Обробка зображень і мультимедіа

Олег Гутік

Лекція 6: Статистичні методи стиснення зображень, II

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

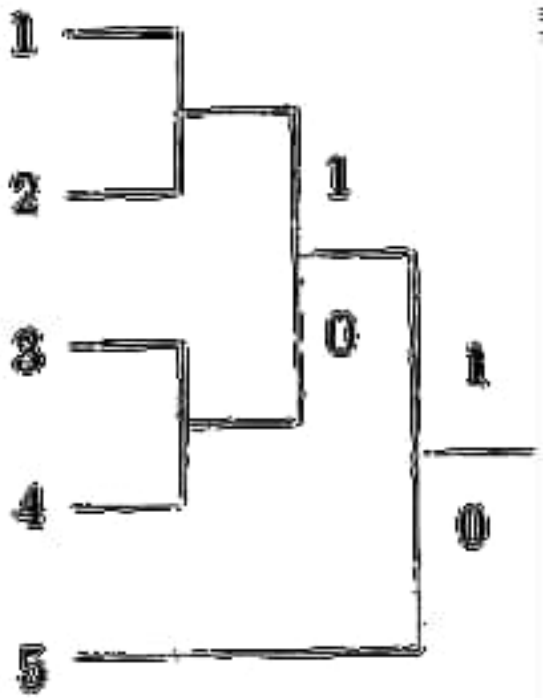
У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

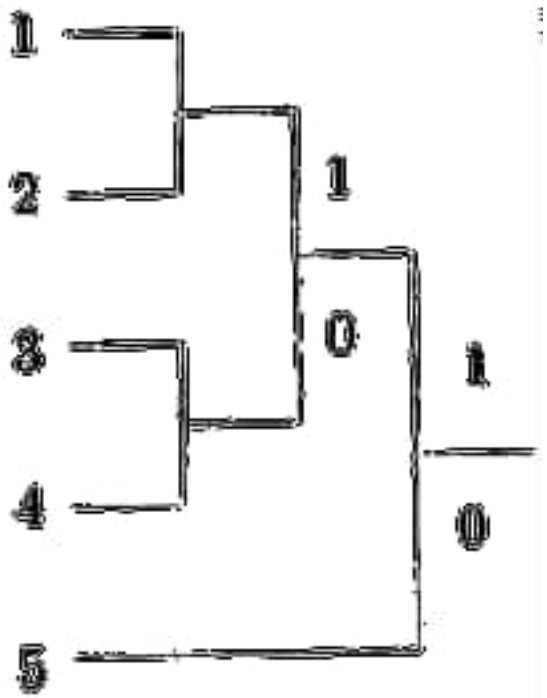
У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.

Перед початком стиснення потоку даних, компресор (кодер) повинен побудувати коди. Це робиться за допомогою ймовірностей (або частот появи) символів. Імовірності та частоти слід записати в стислий файл для того, щоб декомпресор (декодер) Гаффмана міг зробити декомпресію даних. Це легко зробити, оскільки частоти є цілими числами, а ймовірності також є цілими числами. Зазвичай це призводить до додавання кількох сотень байтів у стислий файл. Можна, звичайно, записати в стислий файл самі коди, проте це вносить додаткові труднощі, оскільки коди мають різні довжини. Ще можна записувати у файл саме дерево Гаффмана, але це вимагатиме більшого обсягу, ніж простий запис частот.

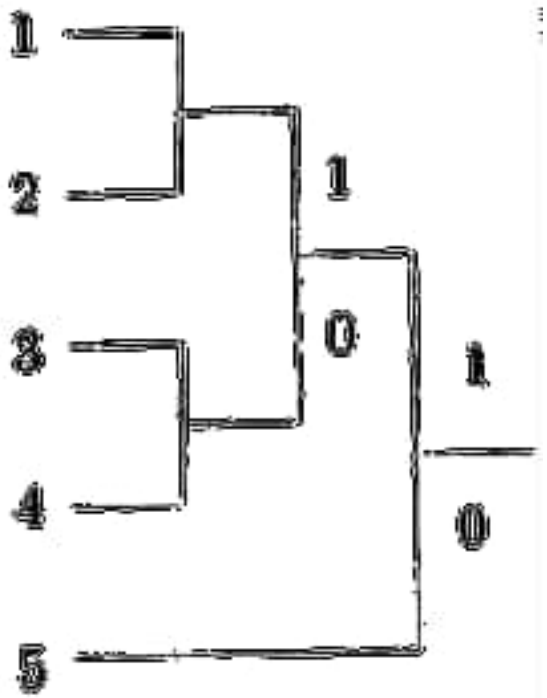
У будь-якому випадку декодер повинен прочитати початок файлу і побудувати дерево Гаффмана для алфавіту. Тільки після цього він може читати та декодувати весь файл. Алгоритм декодування дуже простий. Почати з кореня і прочитати перший біт стисненого файлу. Якщо це нуль, слід рухатися нижньою гілкою дерева; якщо це одиниця, то рухатися треба верхньою гілкою дерева. Далі читається другий біт і відбувається рух наступною гілкою у напрямку листям. Коли декодер досягне листа дерева, він дізнається код першого стисненого символу (зазвичай це символ ASCII). Процедура повторюється для наступного біта, починаючи знову з кореня дерева.



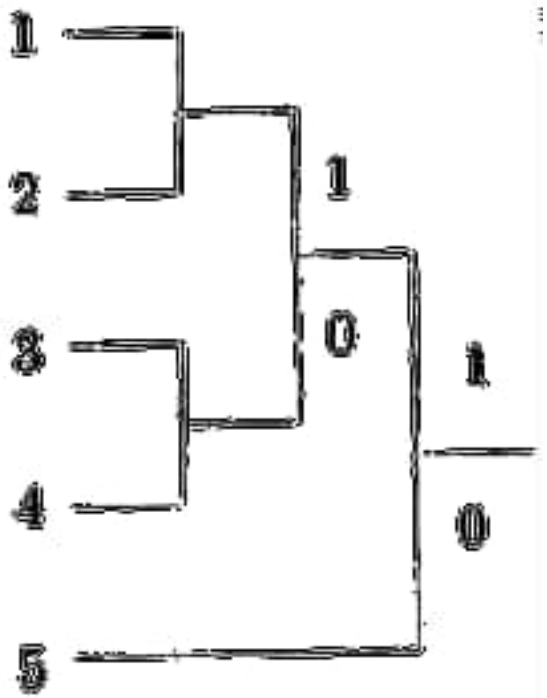
Описану процедуру проілюстровано на рис. для алфавіту із 5 символів. Вхідний рядок "$a_4a_2a_5a_1$" кодується послідовністю 1001100111. Декодер починає з кореня, читає перший біт "1" і йде вгору. Другий біт "0" спрямовує його вниз. Те саме робить третій біт. Це наводить декодер до листка a_4. Отримано перший стиснутий символ. Декодер повертається в корінь і читає 110, рухається вгору, вгору і вниз і отримує символ a_2 і т.д.



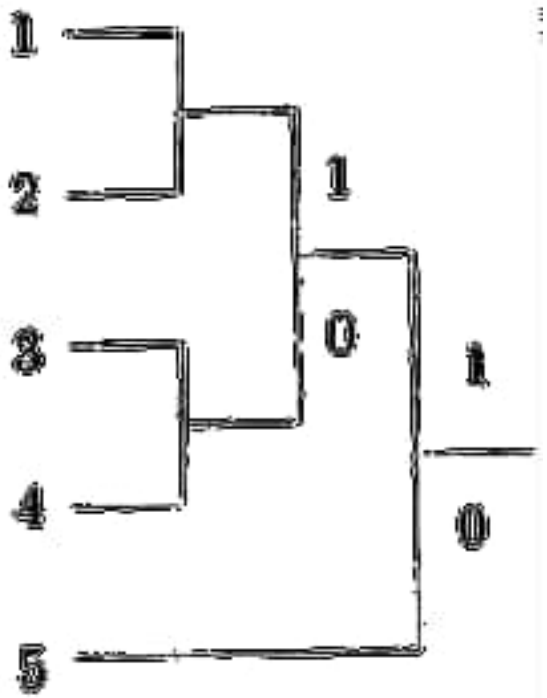
Описану процедуру проілюстровано на рис. для алфавіту із 5 символів. Вхідний рядок "$a_4a_2a_5a_1$" кодується послідовністю 1001100111. Декодер починає з кореня, читає перший біт "1" і йде вгору. Другий біт "0" спрямовує його вниз. Те саме робить третій біт. Це наводить декодер до листка a_4. Отримано перший стиснутий символ. Декодер повертається в корінь і читає 110, рухається вгору, вгору і вниз і отримує символ a_2 і т.д.



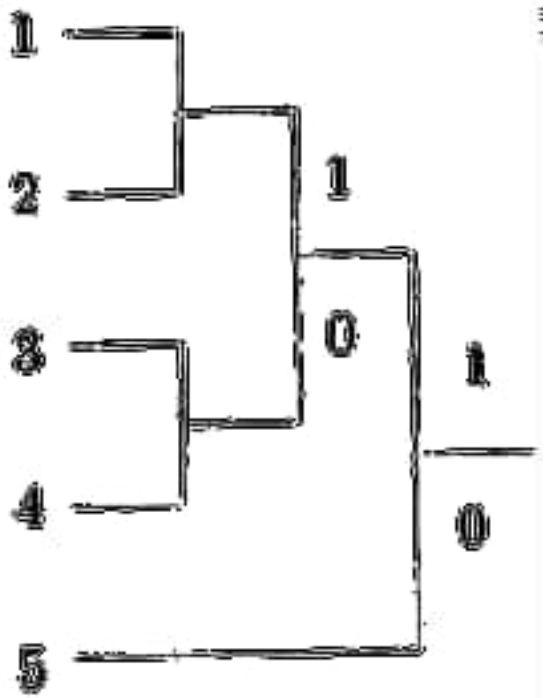
Описану процедуру проілюстровано на рис. для алфавіту із 5 символів. Вхідний рядок "$a_4a_2a_5a_1$" кодується послідовністю 1001100111. Декодер починає з кореня, читає перший біт "1" і йде вгору. Другий біт "0" спрямовує його вниз. Те саме робить третій біт. Це наводить декодер до листка a_4. Отримано перший стиснутий символ. Декодер повертається в корінь і читає 110, рухається вгору, вгору і вниз і отримує символ a_2 і т.д.



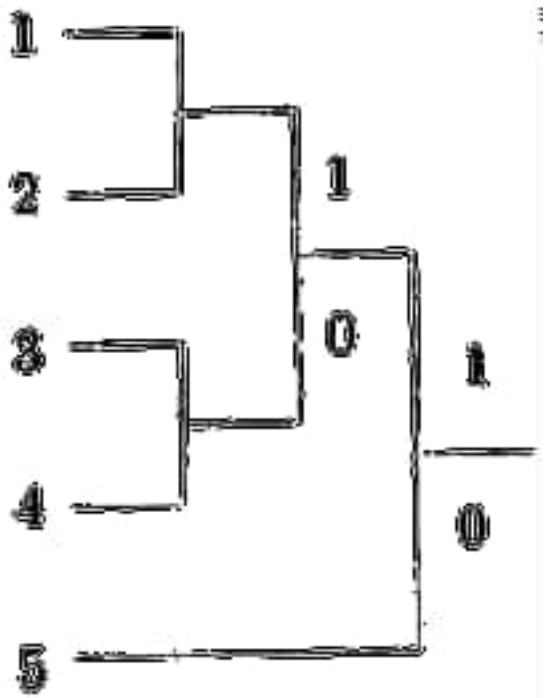
Описану процедуру проілюстровано на рис. для алфавіту із 5 символів. Вхідний рядок "$a_4a_2a_5a_1$" кодується послідовністю 1001100111. Декодер починає з кореня, читає перший біт "1" і йде вгору. Другий біт "0" спрямовує його вниз. Те саме робить третій біт. Це наводить декодер до листка a_4. Отримано перший стиснутий символ. Декодер повертається в корінь і читає 110, рухається вгору, вгору і вниз і отримує символ a_2 і т.д.



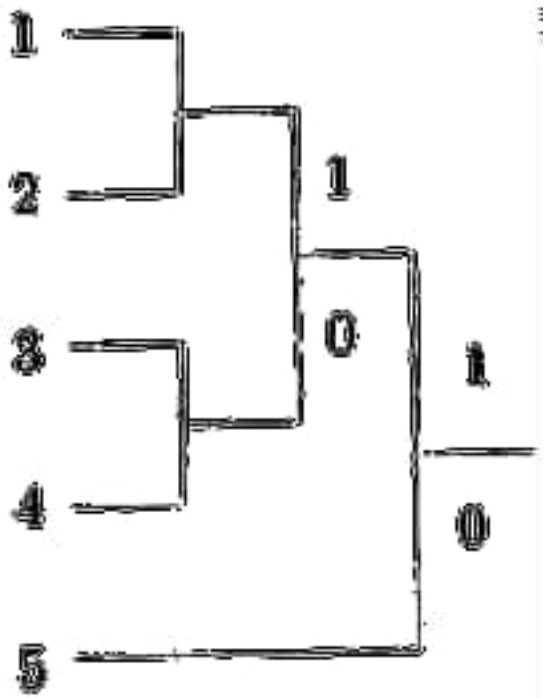
Описану процедуру проілюстровано на рис. для алфавіту із 5 символів. Вхідний рядок "$a_4a_2a_5a_1$" кодується послідовністю 1001100111. Декодер починає з кореня, читає перший біт "1" і йде вгору. Другий біт "0" спрямовує його вниз. Те саме робить третій біт. Це наводить декодер до листка a_4. Отримано перший стиснутий символ. Декодер повертається в корінь і читає 110, рухається вгору, вгору і вниз і отримує символ a_2 і т.д.



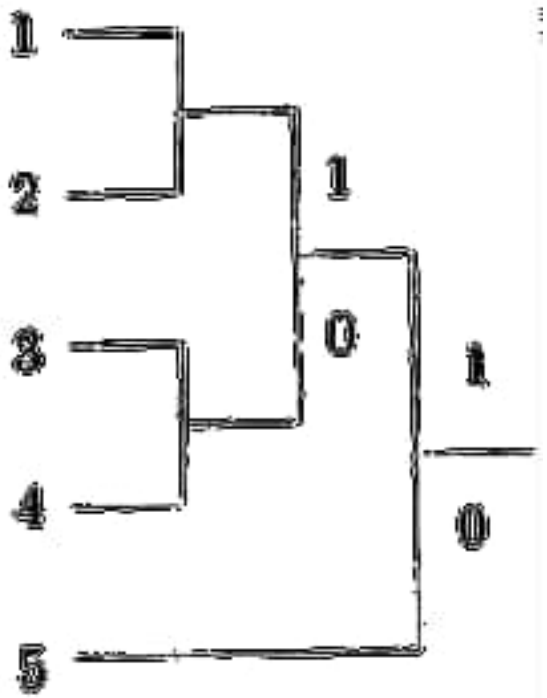
Описану процедуру проілюстровано на рис. для алфавіту із 5 символів. Вхідний рядок "$a_4a_2a_5a_1$" кодується послідовністю 1001100111. Декодер починає з кореня, читає перший біт "1" і йде вгору. Другий біт "0" спрямовує його вниз. Те саме робить третій біт. Це наводить декодер до листка a_4. Отримано перший стиснутий символ. Декодер повертається в корінь і читає 110, рухається вгору, вгору і вниз і отримує символ a_2 і т.д.



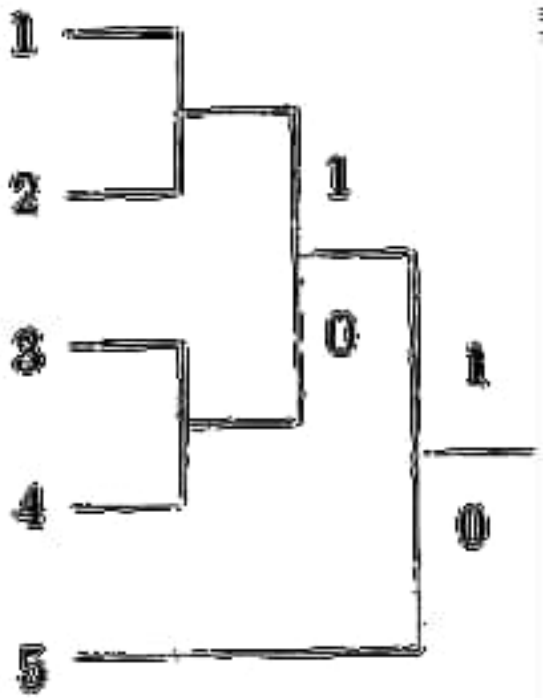
Описану процедуру проілюстровано на рис. для алфавіту із 5 символів. Вхідний рядок "$a_4a_2a_5a_1$" кодується послідовністю 1001100111. Декодер починає з кореня, читає перший біт "1" і йде вгору. Другий біт "0" спрямовує його вниз. Те саме робить третій біт. Це наводить декодер до листка a_4. Отримано перший стиснутий символ. Декодер повертається в корінь і читає 110, рухається вгору, вгору і вниз і отримує символ a_2 і т.д.



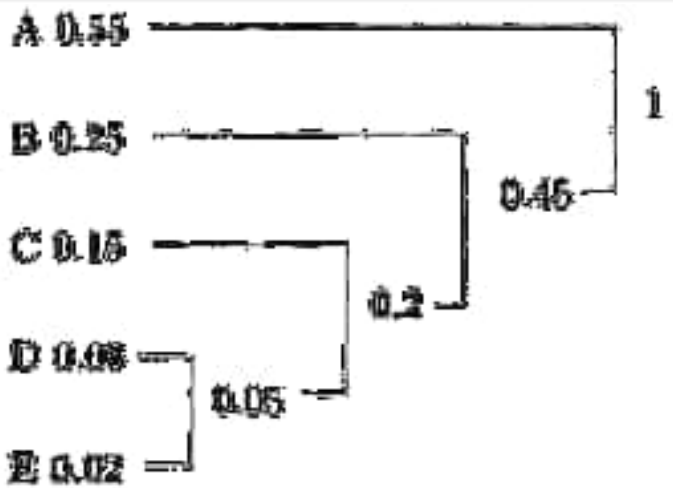
Описану процедуру проілюстровано на рис. для алфавіту із 5 символів. Вхідний рядок "$a_4a_2a_5a_1$" кодується послідовністю 1001100111. Декодер починає з кореня, читає перший біт "1" і йде вгору. Другий біт "0" спрямовує його вниз. Те саме робить третій біт. Це наводить декодер до листка a_4. Отримано перший стиснутий символ. Декодер повертається в корінь і читає 110, рухається вгору, вгору і вниз і отримує символ a_2 і т.д.



Описану процедуру проілюстровано на рис. для алфавіту із 5 символів. Вхідний рядок "$a_4a_2a_5a_1$" кодується послідовністю 1001100111. Декодер починає з кореня, читає перший біт "1" і йде вгору. Другий біт "0" спрямовує його вниз. Те саме робить третій біт. Це наводить декодер до листка a_4. Отримано перший стиснутий символ. Декодер повертається в корінь і читає 110, рухається вгору, вгору і вниз і отримує символ a_2 і т.д.



Описану процедуру проілюстровано на рис. для алфавіту із 5 символів. Вхідний рядок "$a_4a_2a_5a_1$" кодується послідовністю 1001100111. Декодер починає з кореня, читає перший біт "1" і йде вгору. Другий біт "0" спрямовує його вниз. Те саме робить третій біт. Це наводить декодер до листка a_4. Отримано перший стиснутий символ. Декодер повертається в корінь і читає 110, рухається вгору, вгору і вниз і отримує символ a_2 і т.д.



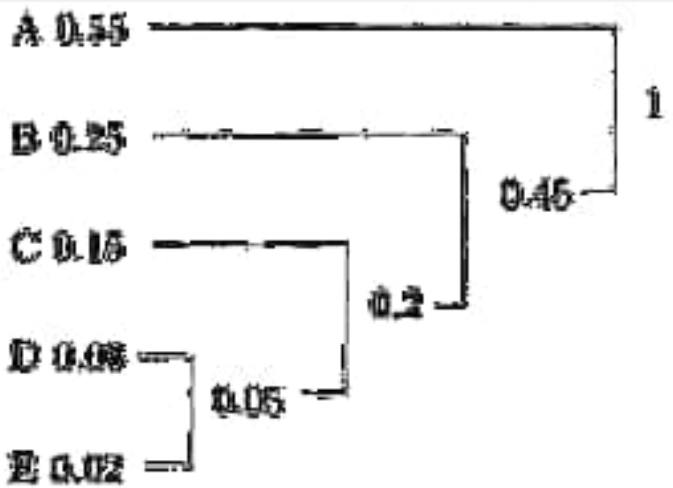
(a)

На рис.(а) представлено множину із п'яти символів з їхніми ймовірностями, а також типове дерево Гаффмана. Символ A виникає в 55% випадків і йому присвоюється однобітовий код, що робить внесок $0.55 \cdot 1$ в середню довжину. Символ E виникає лише у 2% випадків. Йому присвоюється код довжини 4, та його внесок дорівнює $0.02 \cdot 4 = 0.08$. Тоді середня довжина коду дорівнює

$$0.55 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.03 \cdot 4 + 0.02 \cdot 4 = 1.7$$

біт на символ. Дивно, але той самий результат отримаємо, якщо скласти значення ймовірностей чотирьох внутрішніх вузлів кодового дерева:

$$0.05 + 0.2 + 0.45 + 1 = 1.7.$$



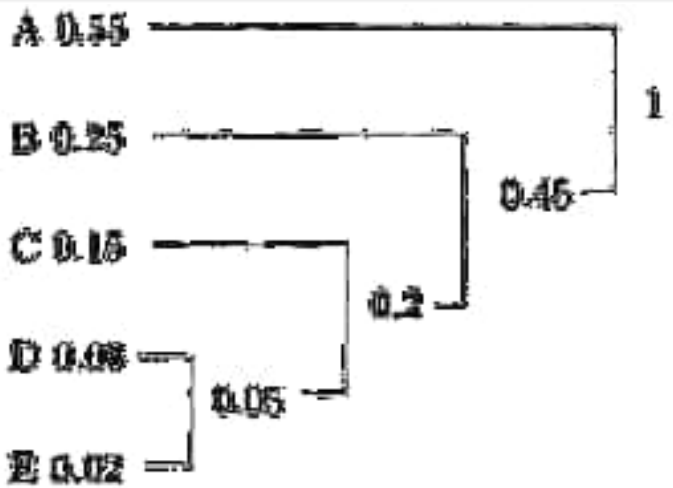
(a)

На рис.(а) представлено множину із п'яти символів з їхніми ймовірностями, а також типове дерево Гаффмана. Символ A виникає в 55% випадків і йому присвоюється однобітовий код, що робить внесок $0.55 \cdot 1$ в середню довжину. Символ E виникає лише у 2% випадків. Йому присвоюється код довжини 4, та його внесок дорівнює $0.02 \cdot 4 = 0.08$. Тоді середня довжина коду дорівнює

$$0.55 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.03 \cdot 4 + 0.02 \cdot 4 = 1.7$$

біт на символ. Дивно, але той самий результат отримаємо, якщо скласти значення ймовірностей чотирьох внутрішніх вузлів кодового дерева:

$$0.05 + 0.2 + 0.45 + 1 = 1.7.$$



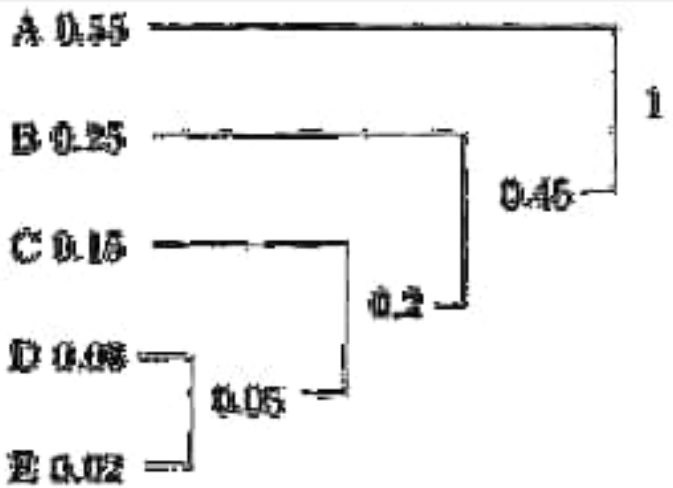
(a)

На рис.(а) представлено множину із п'яти символів з їхніми ймовірностями, а також типове дерево Гаффмана. Символ A виникає в 55% випадків і йому присвоюється однобітовий код, що робить внесок $0.55 \cdot 1$ в середню довжину. Символ E виникає лише у 2% випадків. Йому присвоюється код довжини 4, та його внесок дорівнює $0.02 \cdot 4 = 0.08$. Тоді середня довжина коду дорівнює

$$0.55 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.03 \cdot 4 + 0.02 \cdot 4 = 1.7$$

біт на символ. Дивно, але той самий результат отримаємо, якщо скласти значення ймовірностей чотирьох внутрішніх вузлів кодового дерева:

$$0.05 + 0.2 + 0.45 + 1 = 1.7.$$



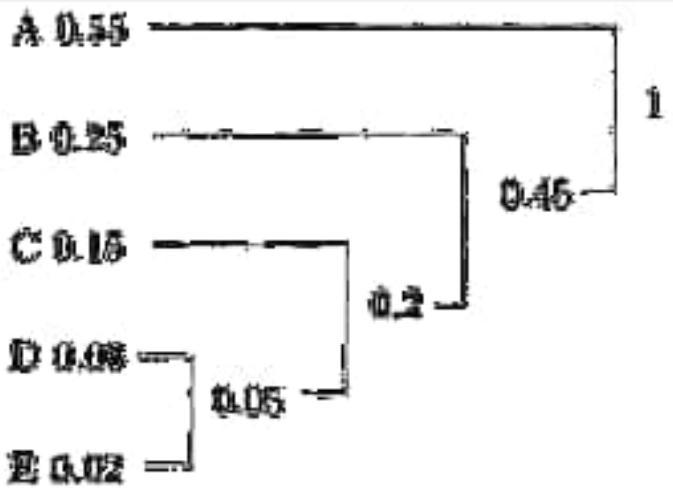
(a)

На рис.(а) представлено множину із п'яти символів з їхніми ймовірностями, а також типове дерево Гаффмана. Символ A виникає в 55% випадків і йому присвоюється однобітовий код, що робить внесок $0.55 \cdot 1$ в середню довжину. Символ E виникає лише у 2% випадків. Йому присвоюється код довжини 4, та його внесок дорівнює $0.02 \cdot 4 = 0.08$. Тоді середня довжина коду дорівнює

$$0.55 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.03 \cdot 4 + 0.02 \cdot 4 = 1.7$$

біт на символ. Дивно, але той самий результат отримаємо, якщо скласти значення ймовірностей чотирьох внутрішніх вузлів кодового дерева:

$$0.05 + 0.2 + 0.45 + 1 = 1.7.$$



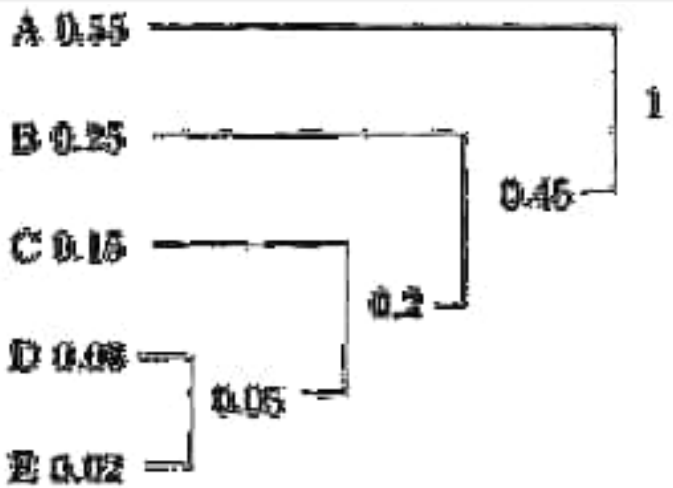
(a)

На рис.(а) представлено множину із п'яти символів з їхніми ймовірностями, а також типове дерево Гаффмана. Символ A виникає в 55% випадків і йому присвоюється однобітовий код, що робить внесок $0.55 \cdot 1$ в середню довжину. Символ E виникає лише у 2% випадків. Йому присвоюється код довжини 4, та його внесок дорівнює $0.02 \cdot 4 = 0.08$. Тоді середня довжина коду дорівнює

$$0.55 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.03 \cdot 4 + 0.02 \cdot 4 = 1.7$$

біт на символ. Дивно, але той самий результат отримаємо, якщо скласти значення ймовірностей чотирьох внутрішніх вузлів кодового дерева:

$$0.05 + 0.2 + 0.45 + 1 = 1.7.$$



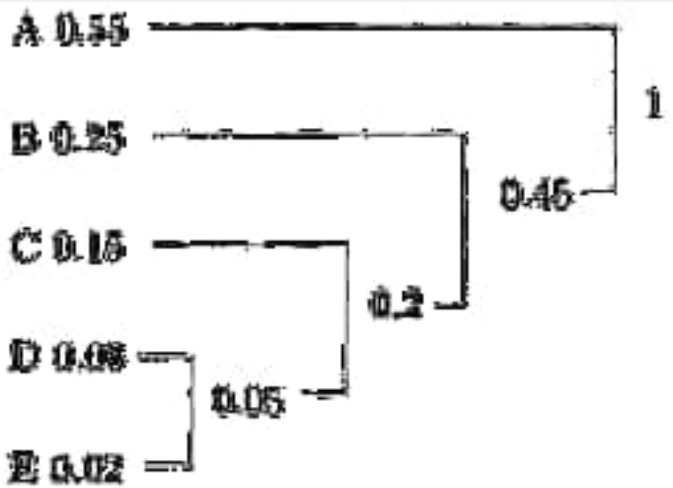
(a)

На рис.(а) представлено множину із п'яти символів з їхніми ймовірностями, а також типове дерево Гаффмана. Символ A виникає в 55% випадків і йому присвоюється однобітовий код, що робить внесок $0.55 \cdot 1$ в середню довжину. Символ E виникає лише у 2% випадків. Йому присвоюється код довжини 4, та його внесок дорівнює $0.02 \cdot 4 = 0.08$. Тоді середня довжина коду дорівнює

$$0.55 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.03 \cdot 4 + 0.02 \cdot 4 = 1.7$$

біт на символ. Дивно, але той самий результат отримаємо, якщо скласти значення ймовірностей чотирьох внутрішніх вузлів кодового дерева:

$$0.05 + 0.2 + 0.45 + 1 = 1.7.$$



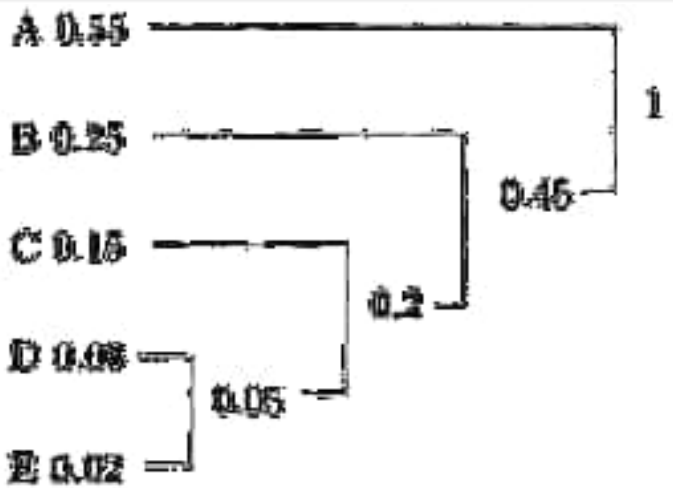
(a)

На рис.(а) представлено множину із п'яти символів з їхніми ймовірностями, а також типове дерево Гаффмана. Символ A виникає в 55% випадків і йому присвоюється однобітовий код, що робить внесок $0.55 \cdot 1$ в середню довжину. Символ E виникає лише у 2% випадків. Йому присвоюється код довжини 4, та його внесок дорівнює $0.02 \cdot 4 = 0.08$. Тоді середня довжина коду дорівнює

$$0.55 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.03 \cdot 4 + 0.02 \cdot 4 = 1.7$$

біт на символ. Дивно, але той самий результат отримаємо, якщо скласти значення ймовірностей чотирьох внутрішніх вузлів кодового дерева:

$$0.05 + 0.2 + 0.45 + 1 = 1.7.$$



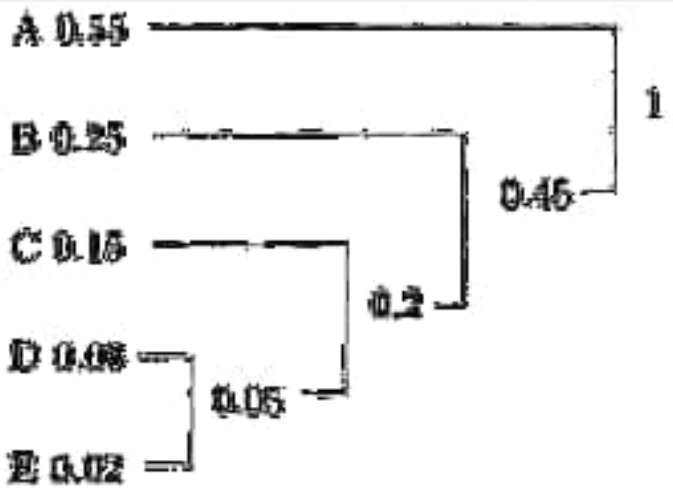
(a)

На рис.(а) представлено множину із п'яти символів з їхніми ймовірностями, а також типове дерево Гаффмана. Символ A виникає в 55% випадків і йому присвоюється однобітовий код, що робить внесок $0.55 \cdot 1$ в середню довжину. Символ E виникає лише у 2% випадків. Йому присвоюється код довжини 4, та його внесок дорівнює $0.02 \cdot 4 = 0.08$. Тоді середня довжина коду дорівнює

$$0.55 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.03 \cdot 4 + 0.02 \cdot 4 = 1.7$$

біт на символ. Дивно, але той самий результат отримаємо, якщо скласти значення ймовірностей чотирьох внутрішніх вузлів кодового дерева:

$$0.05 + 0.2 + 0.45 + 1 = 1.7.$$



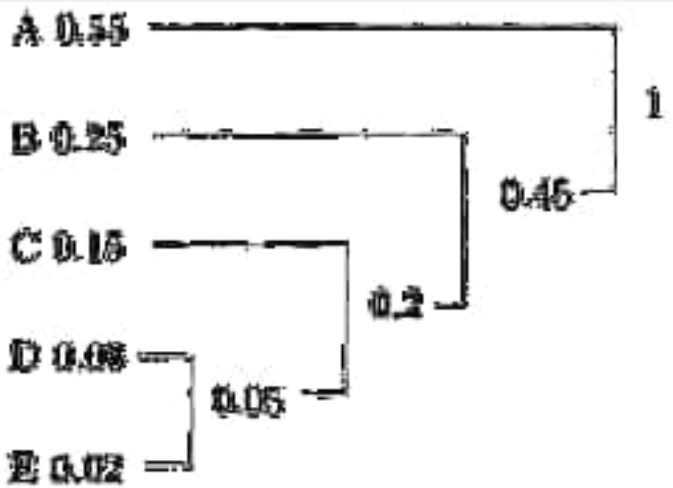
(a)

На рис.(а) представлено множину із п'яти символів з їхніми ймовірностями, а також типове дерево Гаффмана. Символ A виникає в 55% випадків і йому присвоюється однобітовий код, що робить внесок $0.55 \cdot 1$ в середню довжину. Символ E виникає лише у 2% випадків. Йому присвоюється код довжини 4, та його внесок дорівнює $0.02 \cdot 4 = 0.08$. Тоді середня довжина коду дорівнює

$$0.55 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.03 \cdot 4 + 0.02 \cdot 4 = 1.7$$

біт на символ. Дивно, але той самий результат отримаємо, якщо скласти значення ймовірностей чотирьох внутрішніх вузлів кодового дерева:

$$0.05 + 0.2 + 0.45 + 1 = 1.7.$$



(a)

На рис.(а) представлено множину із п'яти символів з їхніми ймовірностями, а також типове дерево Гаффмана. Символ A виникає в 55% випадків і йому присвоюється однобітовий код, що робить внесок $0.55 \cdot 1$ в середню довжину. Символ E виникає лише у 2% випадків. Йому присвоюється код довжини 4, та його внесок дорівнює $0.02 \cdot 4 = 0.08$. Тоді середня довжина коду дорівнює

$$0.55 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.03 \cdot 4 + 0.02 \cdot 4 = 1.7$$

біт на символ. Дивно, але той самий результат отримаємо, якщо скласти значення ймовірностей чотирьох внутрішніх вузлів кодового дерева:

$$0.05 + 0.2 + 0.45 + 1 = 1.7.$$

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

$0.05 =$		$= 0.02 + 0.03$
$0.20 =$	$0.05 + 0.15$	$= 0.02 + 0.03 + 0.15$
$0.45 =$	$0.20 + 0.25$	$= 0.02 + 0.03 + 0.15 + 0.25$
$1.00 =$	$0.45 + 0.55$	$= 0.02 + 0.03 + 0.15 + 0.25 + 0.55$

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

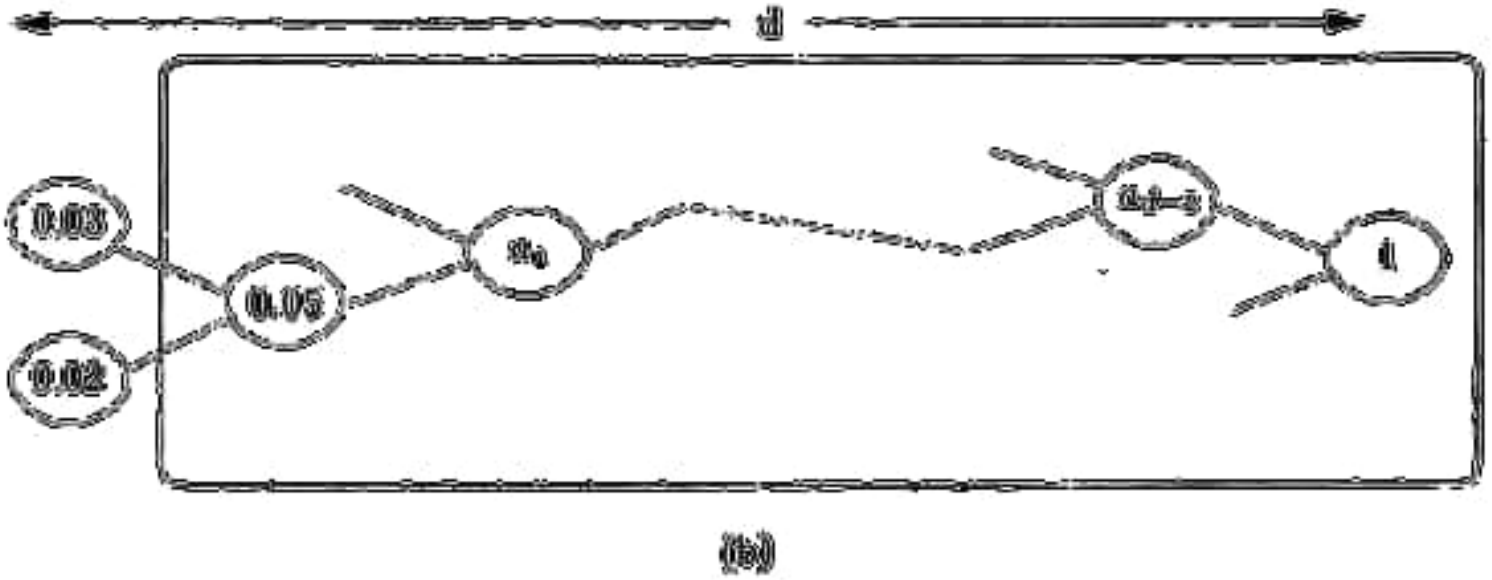
Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.

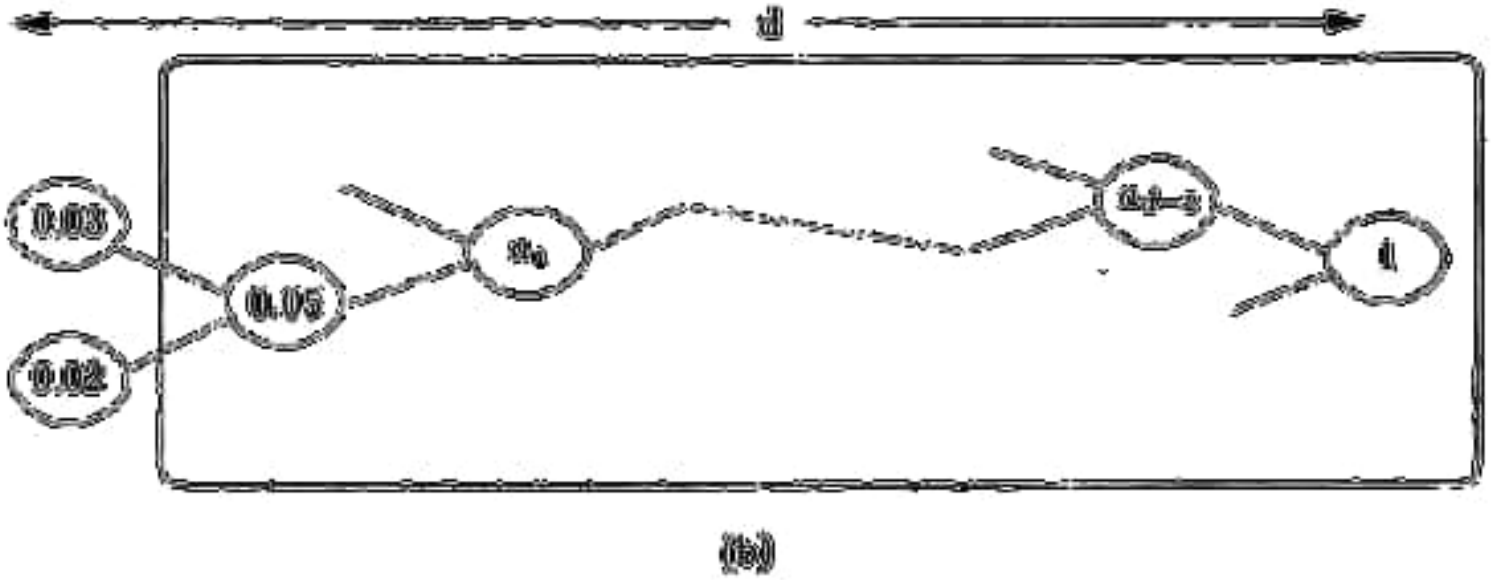
0.05 =		= 0.02 + 0.03
0.20 =	0.05 + 0.15	= 0.02 + 0.03 + 0.15
0.45 =	0.20 + 0.25	= 0.02 + 0.03 + 0.15 + 0.25
1.00 =	0.45 + 0.55	= 0.02 + 0.03 + 0.15 + 0.25 + 0.55

Це спостереження дає простий спосіб обчислення середньої довжини кодів Гаффмана без використання операції множення. Потрібно просто скласти значення внутрішніх вузлів дерева. Табл. ілюструє, чому цей метод працюватиме. У табл. внутрішні вузли виділено. У лівому стовпці виписані величини всіх внутрішніх вузлів. У правих стовпцях показано, як величини складаються з величин попередніх вузлів та з величин листя. Якщо скласти числа в лівому стовпці, то вийде 1.7, а складаючи числа в інших стовпцях, переконуємося, що це число 1.7 є сумою чотирьох чисел 0.02, чотирьох чисел 0.03, трьох чисел 0.15, двох чисел 0.25 та одного числа 0.55.

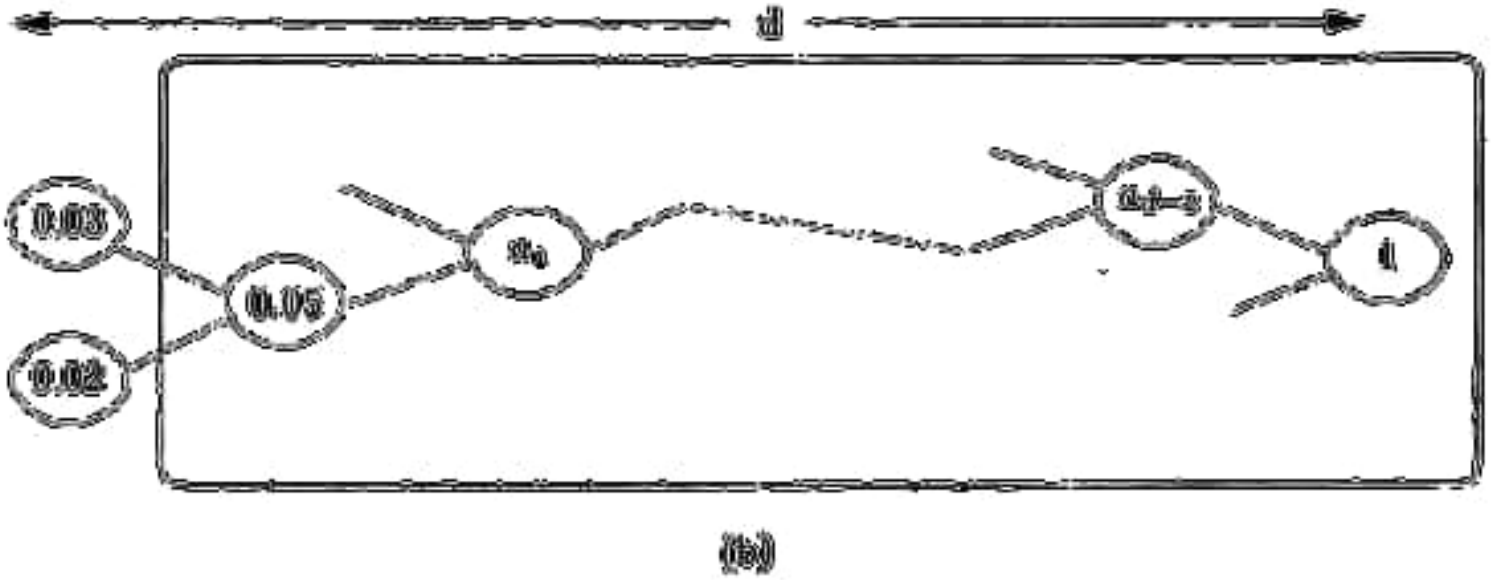
Ці міркування годиться й у загальному випадку. Легко бачити, що в дереві типу Гаффмана (тобто дереві, в якому кожен вузол є сума своїх нащадків) зважена сума листя, де вага листа — це його відстань до кореня, дорівнює сумі всіх внутрішніх вузлів.



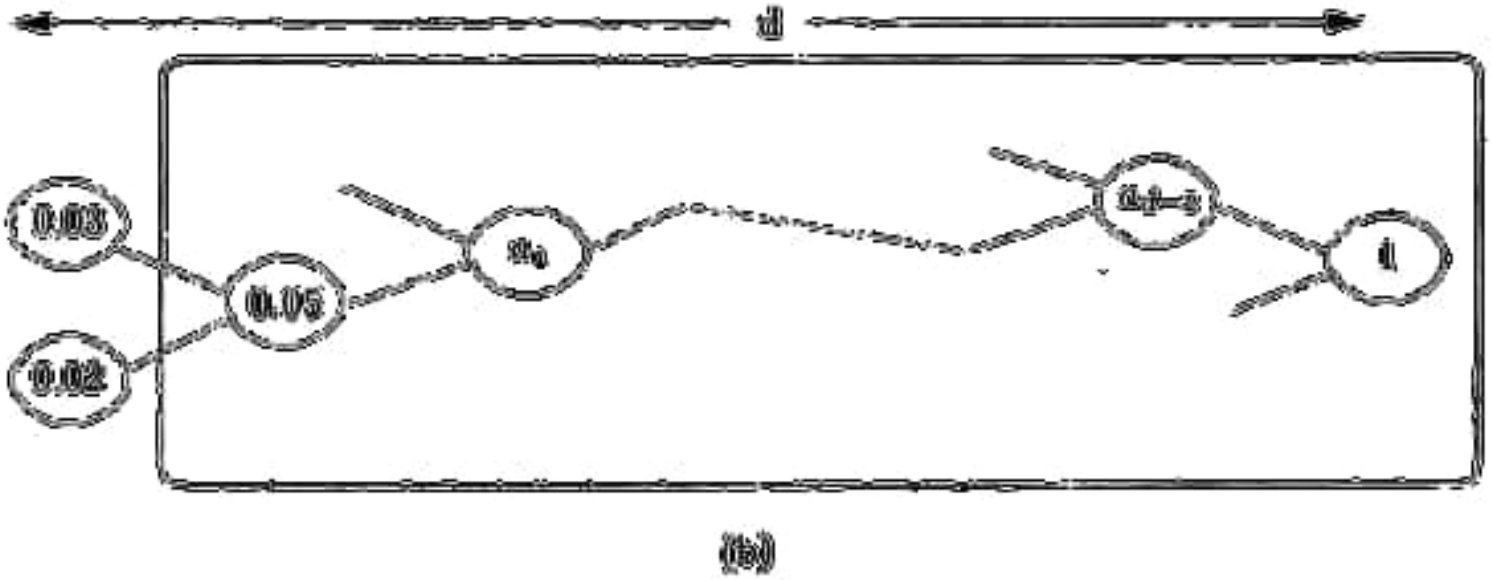
На рис.(b) зображено таке дерево, де передбачається, що два листки 0.02 та 0.03 мають коди Гаффмана довжини d. Усередині дерева це листя є нащадками внутрішнього вузла 0.05, який, у свою чергу, пов'язаний з коренем за допомогою $d-2$ внутрішніх вузлів від a_1 до a_{d-2}.



На рис.(b) зображено таке дерево, де передбачається, що два листки 0.02 та 0.03 мають коди Гаффмана довжини d. Усередині дерева це листя є нащадками внутрішнього вузла 0.05, який, у свою чергу, пов'язаний з коренем за допомогою $d-2$ внутрішніх вузлів від a_1 до a_{d-2}.



На рис.(b) зображено таке дерево, де передбачається, що два листки 0.02 та 0.03 мають коди Гаффмана довжини d. Усередині дерева це листя є нащадками внутрішнього вузла 0.05, який, у свою чергу, пов'язаний з коренем за допомогою $d-2$ внутрішніх вузлів від a_1 до a_{d-2}.



(b)

На рис.(b) зображено таке дерево, де передбачається, що два листки 0.02 та 0.03 мають коди Гаффмана довжини d. Усередині дерева це листя є нащадками внутрішнього вузла 0.05, який, у свою чергу, пов'язаний з коренем за допомогою $d-2$ внутрішніх вузлів від a_1 до a_{d-2}.

$0.05 =$		$= 0.02 + 0.03 + \cdots$
$a_1 =$	$0.05 + \cdots$	$= 0.02 + 0.03 + \cdots$
$a_2 =$	$a_1 + \cdots$	$= 0.02 + 0.03 + \cdots$
$\vdots =$		
$a_{d-2} =$	$a_{d-3} + \cdots$	$= 0.02 + 0.03 + \cdots$
$1.0 =$	$a_{d-2} + \cdots$	$= 0.02 + 0.03 + \cdots$

Табл. складається з d рядків і стверджує, що дві величини 0.02 і 0.03 включаються в різні внутрішні вузли рівно d разів. Додавання величин всіх внутрішніх вузлів дає вклад від цих двох листків, який дорівнює $0.02d + 0.03d$. Оскільки це листя обрано довільно, то зрозуміло, що ця сума включає в себе аналогічний внесок від усіх інших вузлів, тобто дорівнює середній довжині коду. Ця кількість також дорівнює сумі лівого стовпця або сумі всіх внутрішніх вузлів, яка визначить середню довжину коду.

Зазначимо, що у доведенні ніде не припускалося, що дерево — бінарне. Тому ця властивість виконується для будь-якого дерева, де вузли є сумою своїх нащадків.

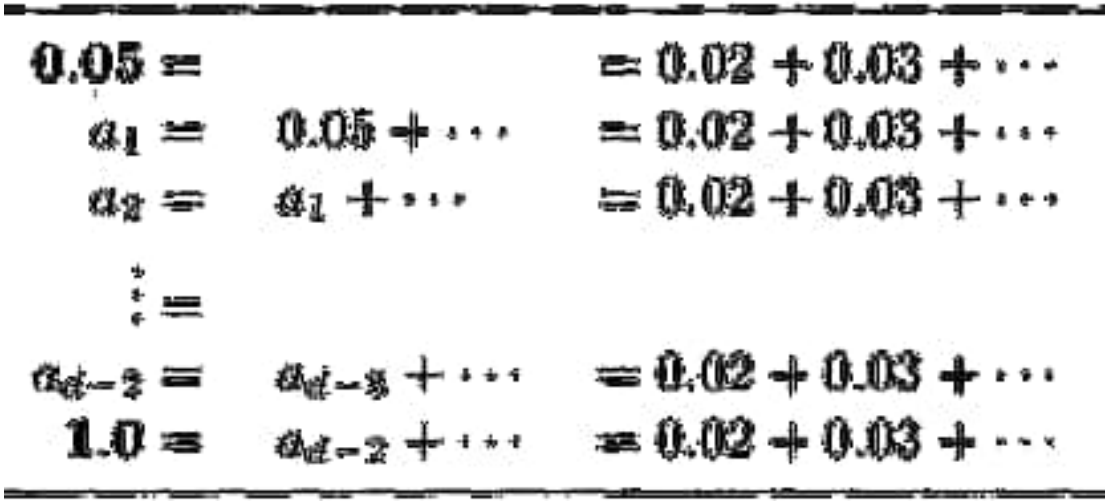
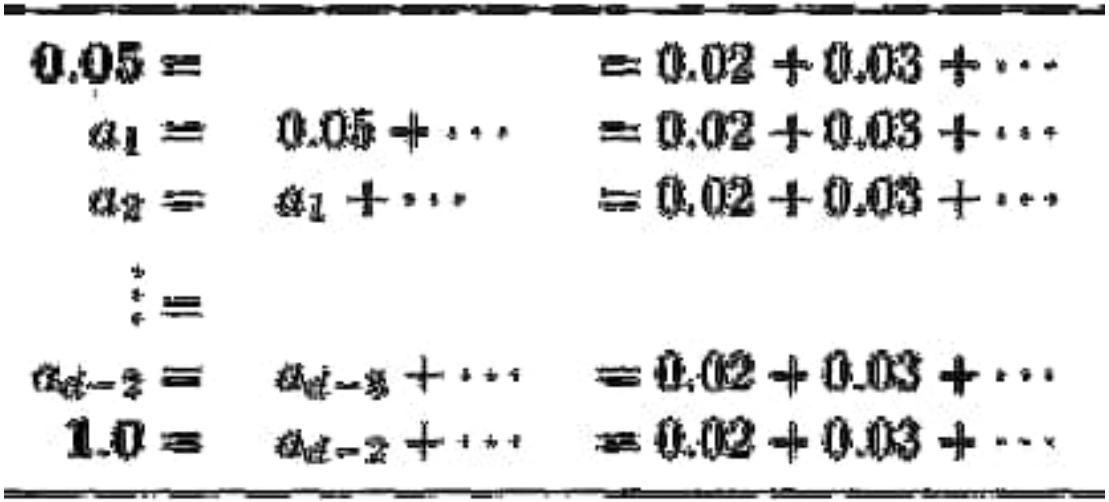


Табл. складається з d рядків і стверджує, що дві величини 0.02 і 0.03 включаються в різні внутрішні вузли рівно d разів. Додавання величин всіх внутрішніх вузлів дає вклад від цих двох листків, який дорівнює $0.02d + 0.03d$. Оскільки це листя обрано довільно, то зрозуміло, що ця сума включає в себе аналогічний внесок від усіх інших вузлів, тобто дорівнює середній довжині коду. Ця кількість також дорівнює сумі лівого стовпця або сумі всіх внутрішніх вузлів, яка визначить середню довжину коду.

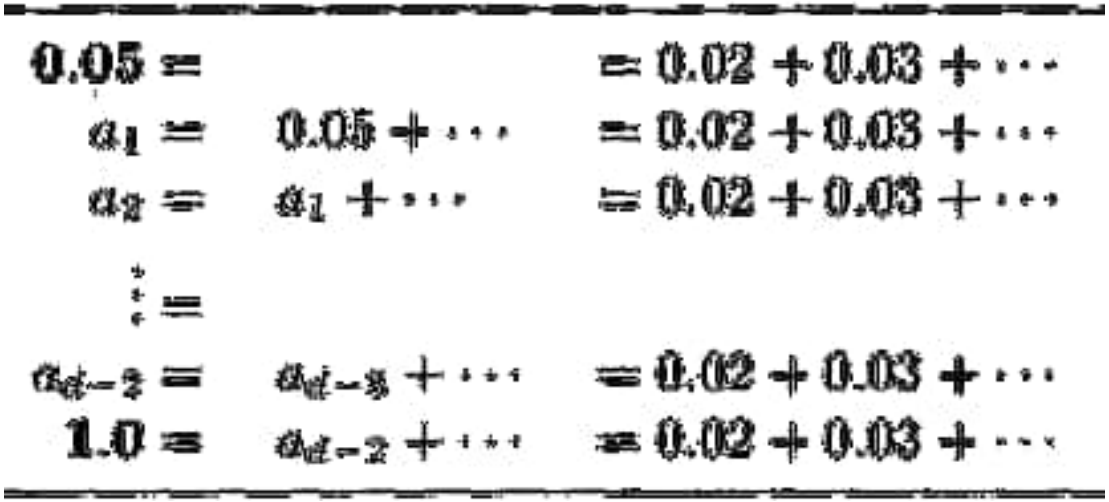
Зазначимо, що у доведенні ніде не припускалося, що дерево — бінарне. Тому ця властивість виконується для будь-якого дерева, де вузли є сумою своїх нащадків.



$0.05 =$		$= 0.02 + 0.03 + \cdots$
$a_1 =$	$0.05 + \cdots$	$= 0.02 + 0.03 + \cdots$
$a_2 =$	$a_1 + \cdots$	$= 0.02 + 0.03 + \cdots$
$\vdots =$		
$a_{d-2} =$	$a_{d-3} + \cdots$	$= 0.02 + 0.03 + \cdots$
$1.0 =$	$a_{d-2} + \cdots$	$= 0.02 + 0.03 + \cdots$

Табл. складається з d рядків і стверджує, що дві величини 0.02 і 0.03 включаються в різні внутрішні вузли рівно d разів. Додавання величин всіх внутрішніх вузлів дає вклад від цих двох листків, який дорівнює $0.02d + 0.03d$. Оскільки це листя обрано довільно, то зрозуміло, що ця сума включає в себе аналогічний внесок від усіх інших вузлів, тобто дорівнює середній довжині коду. Ця кількість також дорівнює сумі лівого стовпця або сумі всіх внутрішніх вузлів, яка визначить середню довжину коду.

Зазначимо, що у доведенні ніде не припускалося, що дерево — бінарне. Тому ця властивість виконується для будь-якого дерева, де вузли є сумою своїх нащадків.



$0.05 =$		$= 0.02 + 0.03 + \cdots$
$a_1 =$	$0.05 + \cdots$	$= 0.02 + 0.03 + \cdots$
$a_2 =$	$a_1 + \cdots$	$= 0.02 + 0.03 + \cdots$
$\vdots =$		
$a_{d-2} =$	$a_{d-3} + \cdots$	$= 0.02 + 0.03 + \cdots$
$1.0 =$	$a_{d-2} + \cdots$	$= 0.02 + 0.03 + \cdots$

Табл. складається з d рядків і стверджує, що дві величини 0.02 і 0.03 включаються в різні внутрішні вузли рівно d разів. Додавання величин всіх внутрішніх вузлів дає вклад від цих двох листків, який дорівнює $0.02d + 0.03d$. Оскільки це листя обрано довільно, то зрозуміло, що ця сума включає в себе аналогічний внесок від усіх інших вузлів, тобто дорівнює середній довжині коду. Ця кількість також дорівнює сумі лівого стовпця або сумі всіх внутрішніх вузлів, яка визначить середню довжину коду.

Зазначимо, що у доведенні ніде не припускалося, що дерево — бінарне. Тому ця властивість виконується для будь-якого дерева, де вузли є сумою своїх нащадків.

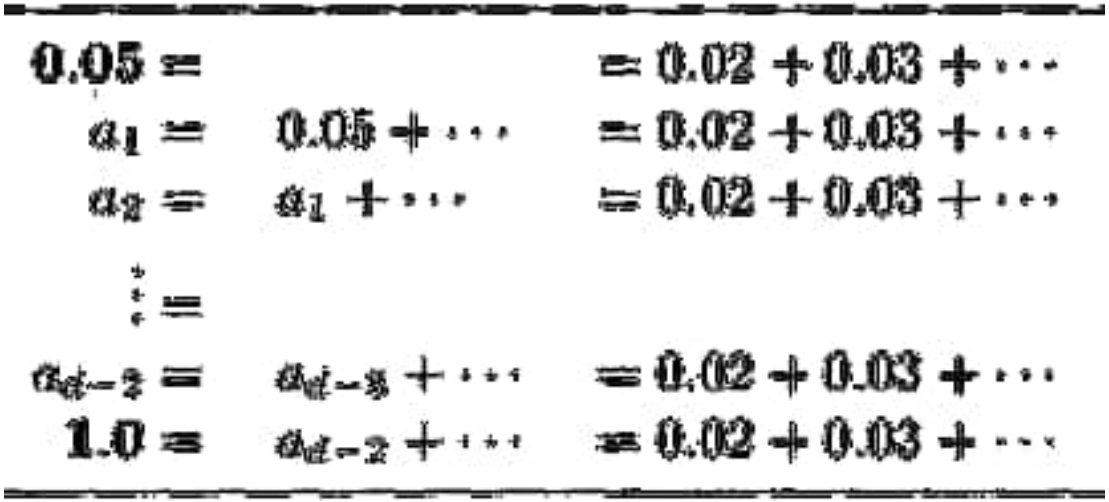
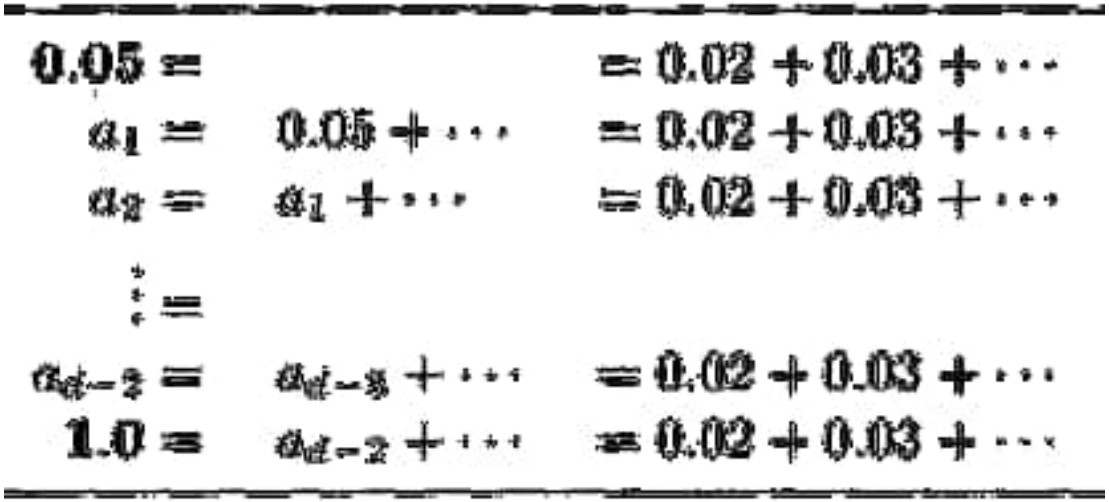


Табл. складається з d рядків і стверджує, що дві величини 0.02 і 0.03 включаються в різні внутрішні вузли рівно d разів. Додавання величин всіх внутрішніх вузлів дає вклад від цих двох листків, який дорівнює $0.02d + 0.03d$. Оскільки це листя обрано довільно, то зрозуміло, що ця сума включає в себе аналогічний внесок від усіх інших вузлів, тобто дорівнює середній довжині коду. Ця кількість також дорівнює сумі лівого стовпця або сумі всіх внутрішніх вузлів, яка визначить середню довжину коду.

Зазначимо, що у доведенні ніде не припускалося, що дерево — бінарне. Тому ця властивість виконується для будь-якого дерева, де вузли є сумою своїх нащадків.



$0.05 =$		$= 0.02 + 0.03 + \cdots$
$a_1 =$	$0.05 + \cdots$	$= 0.02 + 0.03 + \cdots$
$a_2 =$	$a_1 + \cdots$	$= 0.02 + 0.03 + \cdots$
$\vdots =$		
$a_{d-2} =$	$a_{d-3} + \cdots$	$= 0.02 + 0.03 + \cdots$
$1.0 =$	$a_{d-2} + \cdots$	$= 0.02 + 0.03 + \cdots$

Табл. складається з d рядків і стверджує, що дві величини 0.02 і 0.03 включаються в різні внутрішні вузли рівно d разів. Додавання величин всіх внутрішніх вузлів дає вклад від цих двох листків, який дорівнює $0.02d + 0.03d$. Оскільки це листя обрано довільно, то зрозуміло, що ця сума включає в себе аналогічний внесок від усіх інших вузлів, тобто дорівнює середній довжині коду. Ця кількість також дорівнює сумі лівого стовпця або сумі всіх внутрішніх вузлів, яка визначить середню довжину коду.

Зазначимо, що у доведенні ніде не припускалося, що дерево — бінарне. Тому ця властивість виконується для будь-якого дерева, де вузли є сумою своїх нащадків.

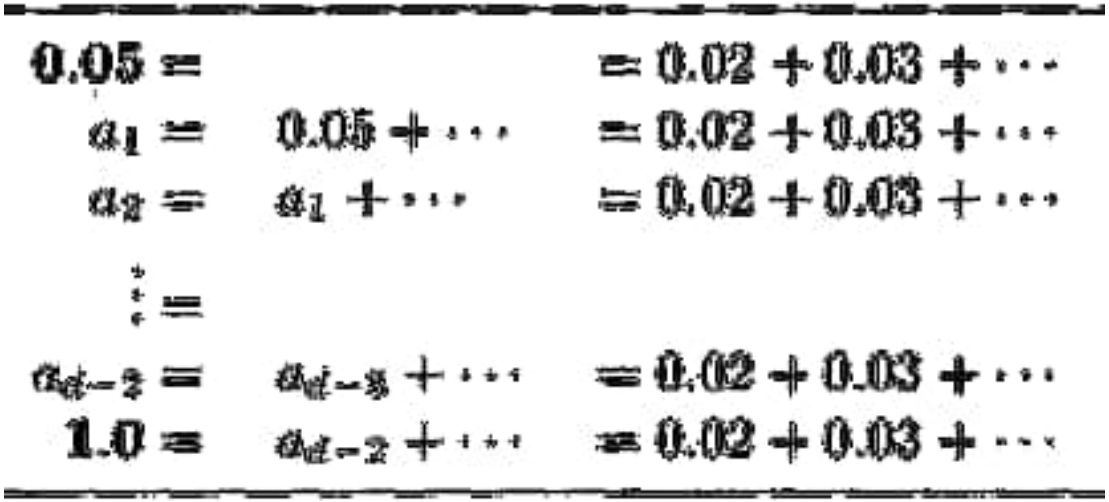
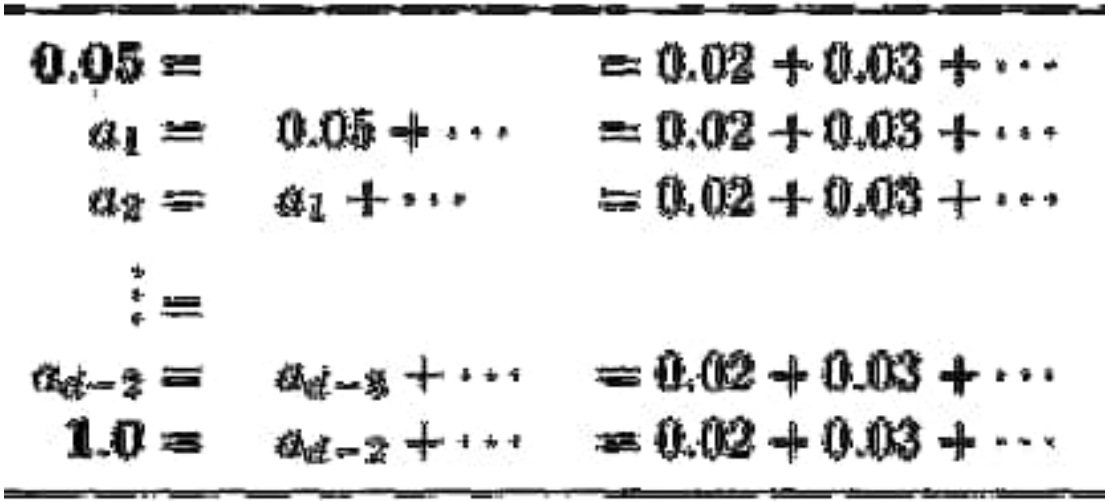


Табл. складається з d рядків і стверджує, що дві величини 0.02 і 0.03 включаються в різні внутрішні вузли рівно d разів. Додавання величин всіх внутрішніх вузлів дає вклад від цих двох листків, який дорівнює $0.02d + 0.03d$. Оскільки це листя обрано довільно, то зрозуміло, що ця сума включає в себе аналогічний внесок від усіх інших вузлів, тобто дорівнює середній довжині коду. Ця кількість також дорівнює сумі лівого стовпця або сумі всіх внутрішніх вузлів, яка визначить середню довжину коду.

Зазначимо, що у доведенні ніде не припускалося, що дерево — бінарне. Тому ця властивість виконується для будь-якого дерева, де вузли є сумою своїх нащадків.



$0.05 =$		$= 0.02 + 0.03 + \cdots$
$a_1 =$	$0.05 + \cdots$	$= 0.02 + 0.03 + \cdots$
$a_2 =$	$a_1 + \cdots$	$= 0.02 + 0.03 + \cdots$
$\vdots =$		
$a_{d-2} =$	$a_{d-3} + \cdots$	$= 0.02 + 0.03 + \cdots$
$1.0 =$	$a_{d-2} + \cdots$	$= 0.02 + 0.03 + \cdots$

Табл. складається з d рядків і стверджує, що дві величини 0.02 і 0.03 включаються в різні внутрішні вузли рівно d разів. Додавання величин всіх внутрішніх вузлів дає вклад від цих двох листків, який дорівнює $0.02d + 0.03d$. Оскільки це листя обрано довільно, то зрозуміло, що ця сума включає в себе аналогічний внесок від усіх інших вузлів, тобто дорівнює середній довжині коду. Ця кількість також дорівнює сумі лівого стовпця або сумі всіх внутрішніх вузлів, яка визначить середню довжину коду.

Зазначимо, що у доведенні ніде не припускалося, що дерево — бінарне. Тому ця властивість виконується для будь-якого дерева, де вузли є сумою своїх нащадків.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.

Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

Метод Гаффмана передбачає, що частоти символів алфавіту відомі декодеру. Насправді це буває дуже рідко. Один можливий розв'язок для компресора — це прочитати вихідні дані двічі. Вперше, щоб обчислити частоти, а вдруге зробити стиснення. У проміжку компресор будує дерево Гаффмана. Такий метод називається *напівадаптивним*, і він працює повільно для практичного використання. Насправді застосовується метод адаптивного (чи динамічного) кодування Гаффмана. Цей метод є основою програми compact операційної системи UNIX.
Основна ідея полягає в тому, що і компресор, і декомпресор починають з пустого дерева Гаффмана, а потім модифікує його за порядком читання й обробки символів (у разі, якщо обробка компресора означає стиснення, а для декомпресора — декомпресію). І компресор, і декомпресор повинні модифікувати дерево абсолютно одинаково, щоб весь час використовувати один і той же код, який може змінюватися під час процесу. Будемо говорити, що компресор і декомпресор синхронізовані, якщо їхня робота жорстко узгоджена (хоча і не обов'язково виконується в один і той же час). Слово *дзеркально*, можливо, краще позначає суть їхньої роботи. Декодер дзеркально повторює операції кодера.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

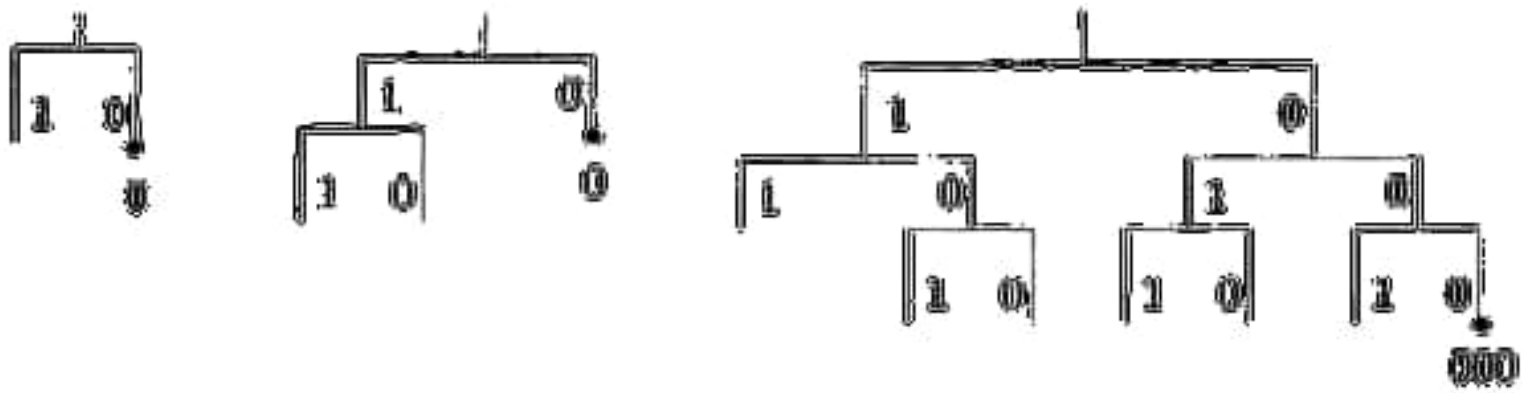
Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.

На початку кодер будує порожнє дерево Гаффмана. Жодному символу код ще не присвоєно. Перший вхідний символ просто записується у вихідний файл у нестисненій формі. Потім цей символ поміщається на дерево і йому присвоюється код. Якщо він зустрінеться в наступний раз, то його поточний код буде записаний у файл, а його частота буде збільшена на одиницю. Оскільки ця операція модифікувала дерево, його потрібно перевірити, чи є воно деревом Гаффмана (що дає найкращі коди). Якщо немає, це тягне за собою перебудову дерева і зміну кодів.

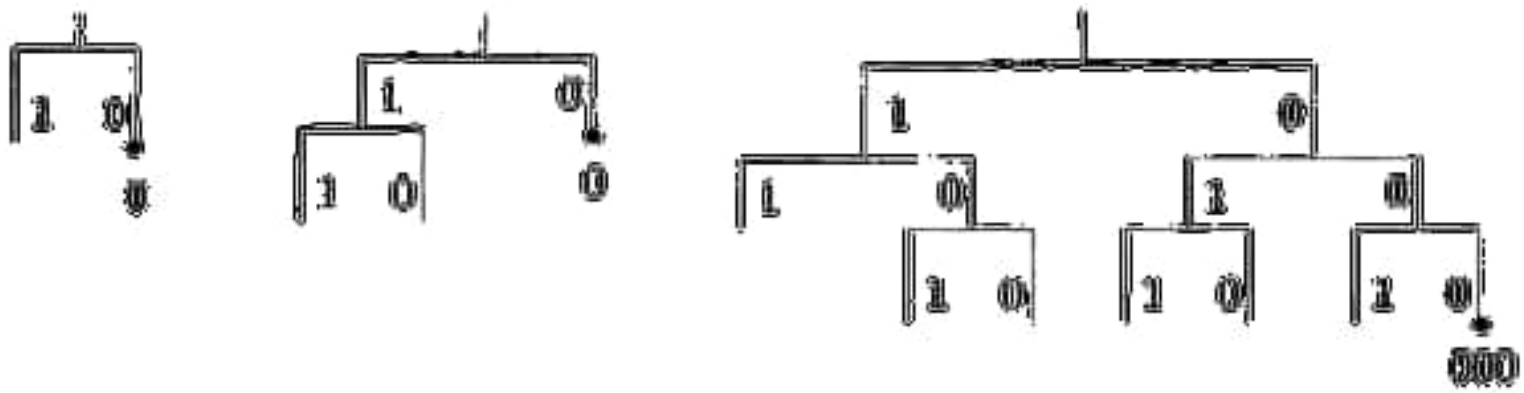
Декомпресор дзеркально повторює цю дію. Коли він читає нестиснений символ, то він додає його на дерево і присвоює йому код. Коли він читає стиснений код (змінної довжини), то він використовує поточне дерево, щоб визначити, який символ відповідає цьому коду, після чого модифікує дерево таким же чином, що і кодер.

Однак, існує одне тонке місце. Декодер повинен знати, чи цей утворений просто нестисненим символом (звичайно, це 8-бітний код ASCII) або це код змінної довжини. Для подолання багатозначності кожному нестисненому символу буде запропоновано спеціальний esc (escape) код змінної довжини. Якщо декомпресор читає цей код, то він точно знає, що за ним слідує 8-розрядний код ASCII, який вперше з'явився на вході.



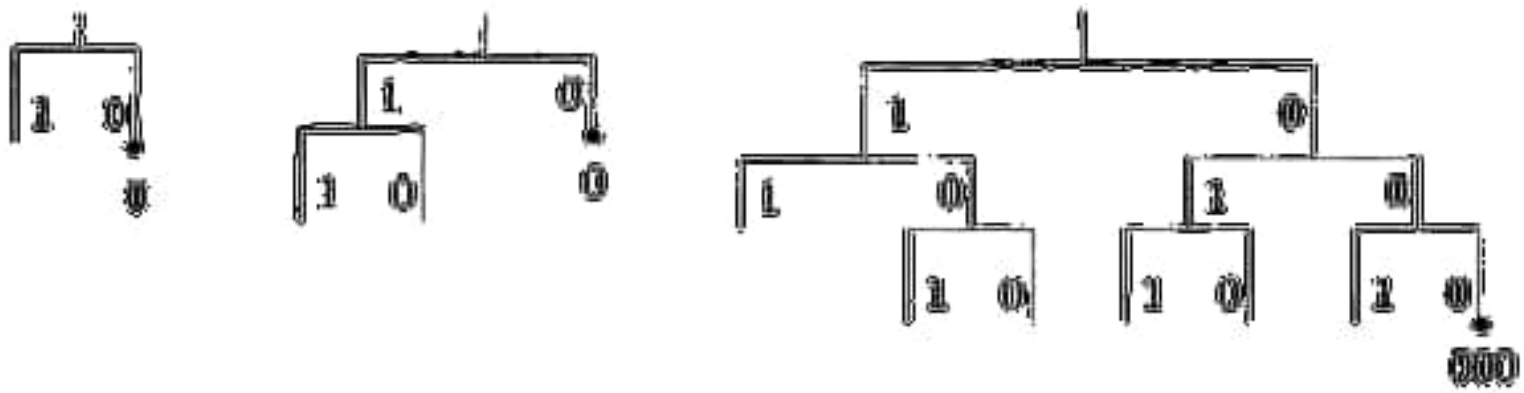
Занепокоєння викликає те, що код esc не повинен бути змінним кодом, використовуваним для символів алфавіта. Оскільки ці коди весь час зазнають зміни, то і код для esc слід також модифікувати. Природний шлях — це додати до дерева ще один порожній листок з постійною нульовою частотою, що б йому весь час присвоювалась гілка з одних нулів. Оскільки цей листок буде весь час присутнім на дереві, то йому весь час буде присвоюватися код змінної довжини. Це і буде код esc, попередній кожному новому нестисненому символу. Дерево буде весь час модифікуватися, і буде змінюватися положення на пустому листку та його коді, але сам цей код буде ідентифікувати кожен новий нестиснений символ у стиснутому файлі. На рис. зображено рух цього коду esc по мірі зростання дерева.

Цей метод також використовується у відомому протоколі V.32 передачі даних по модему зі швидкістю 14400 бод.



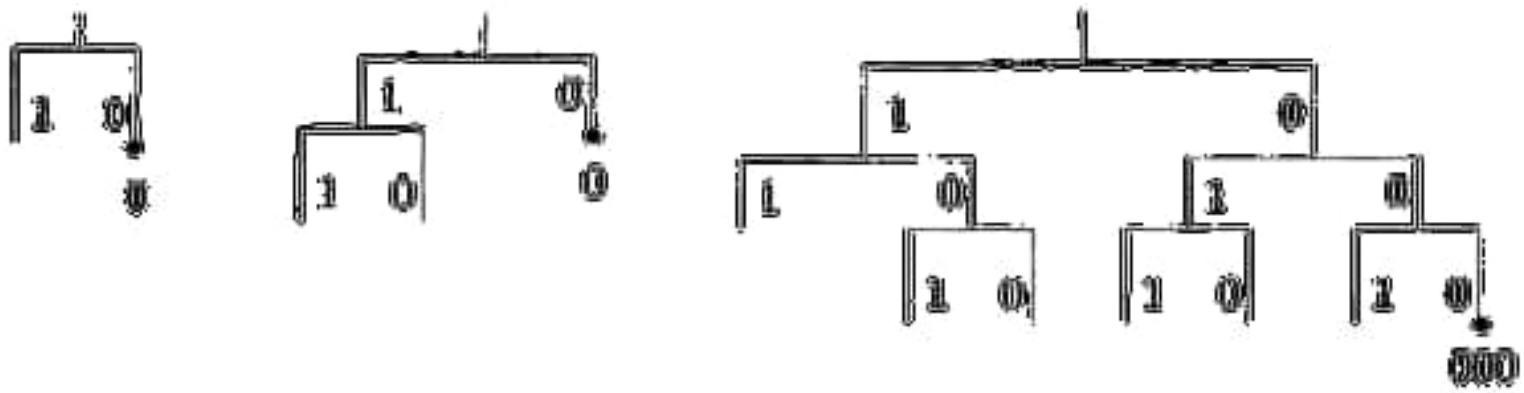
Занепокоєння викликає те, що код esc не повинен бути змінним кодом, використовуваним для символів алфавіта. Оскільки ці коди весь час зазнають зміни, то і код для esc слід також модифікувати. Природний шлях — це додати до дерева ще один порожній листок з постійною нульовою частотою, що б йому весь час присвоювалась гілка з одних нулів. Оскільки цей листок буде весь час присутнім на дереві, то йому весь час буде присвоюватися код змінної довжини. Це і буде код esc, попередній кожному новому нестисненому символу. Дерево буде весь час модифікуватися, і буде змінюватися положення на пустому листку та його коді, але сам цей код буде ідентифікувати кожен новий нестиснений символ у стиснутому файлі. На рис. зображено рух цього коду esc по мірі зростання дерева.

Цей метод також використовується у відомому протоколі V.32 передачі даних по модему зі швидкістю 14400 бод.



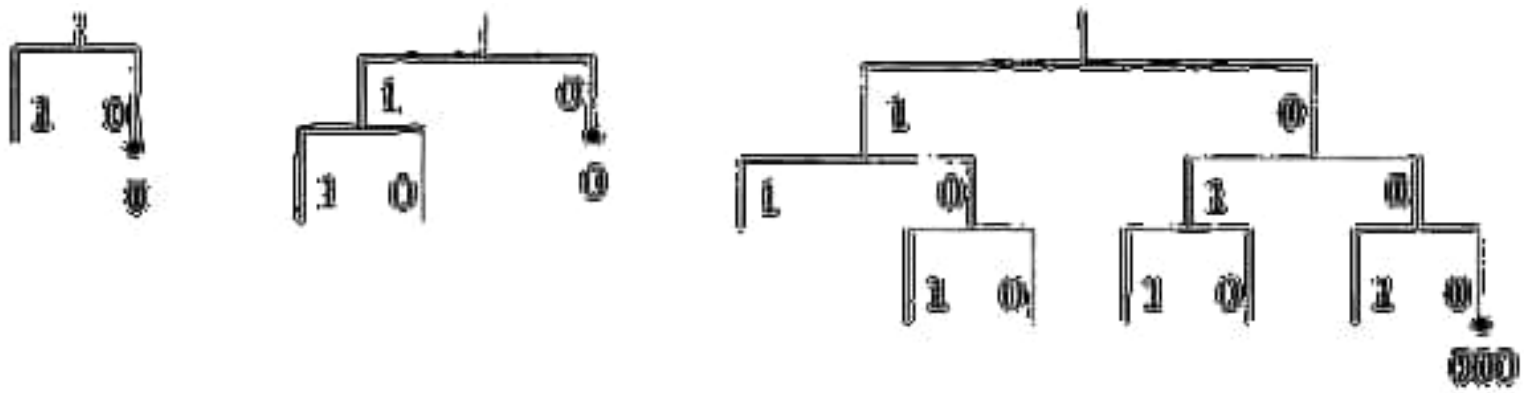
Занепокоєння викликає те, що код esc не повинен бути змінним кодом, використовуваним для символів алфавіта. Оскільки ці коди весь час зазнають зміни, то і код для esc слід також модифікувати. Природний шлях — це додати до дерева ще один порожній листок з постійною нульовою частотою, що б йому весь час присвоювалась гілка з одних нулів. Оскільки цей листок буде весь час присутнім на дереві, то йому весь час буде присвоюватися код змінної довжини. Це і буде код esc, попередній кожному новому нестисненому символу. Дерево буде весь час модифікуватися, і буде змінюватися положення на пустому листку та його коді, але сам цей код буде ідентифікувати кожен новий нестиснений символ у стиснутому файлі. На рис. зображено рух цього коду esc по мірі зростання дерева.

Цей метод також використовується у відомому протоколі V.32 передачі даних по модему зі швидкістю 14400 бод.



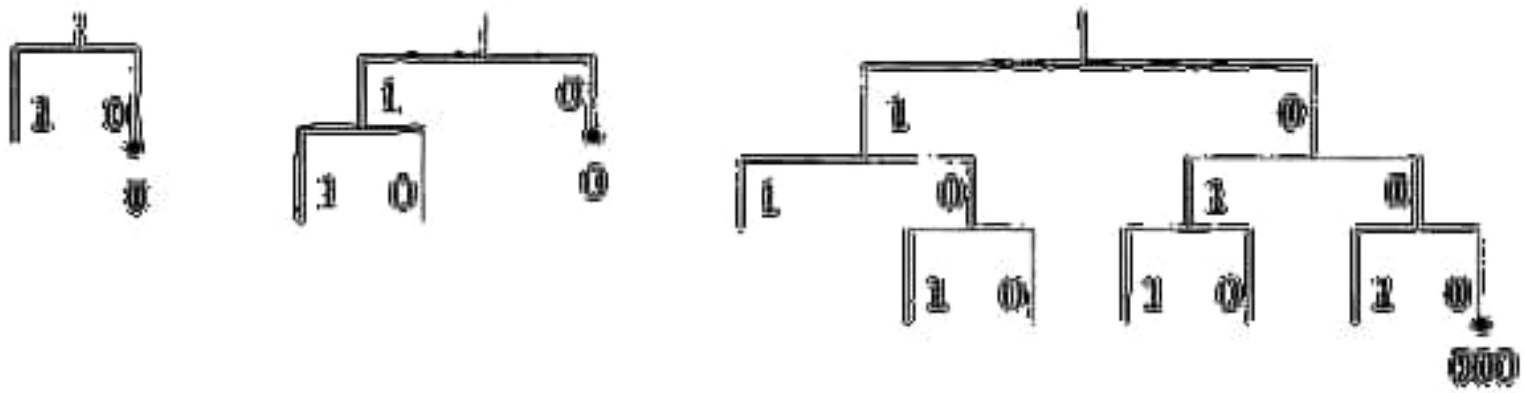
Занепокоєння викликає те, що код esc не повинен бути змінним кодом, використовуваним для символів алфавіта. Оскільки ці коди весь час зазнають зміни, то і код для esc слід також модифікувати. Природний шлях — це додати до дерева ще один порожній листок з постійною нульовою частотою, що б йому весь час присвоювалась гілка з одних нулів. Оскільки цей листок буде весь час присутнім на дереві, то йому весь час буде присвоюватися код змінної довжини. Це і буде код esc, попередній кожному новому нестисненому символу. Дерево буде весь час модифікуватися, і буде змінюватися положення на пустому листку та його коді, але сам цей код буде ідентифікувати кожен новий нестиснений символ у стиснутому файлі. На рис. зображено рух цього коду esc по мірі зростання дерева.

Цей метод також використовується у відомому протоколі V.32 передачі даних по модему зі швидкістю 14400 бод.



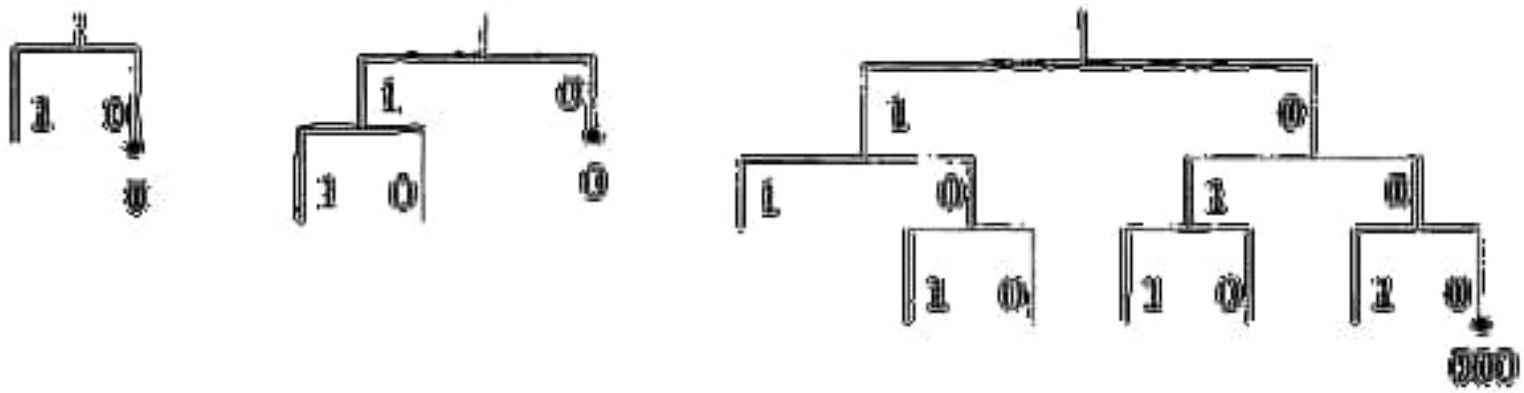
Занепокоєння викликає те, що код esc не повинен бути змінним кодом, використовуваним для символів алфавіта. Оскільки ці коди весь час зазнають зміни, то і код для esc слід також модифікувати. Природний шлях — це додати до дерева ще один порожній листок з постійною нульовою частотою, що б йому весь час присвоювалась гілка з одних нулів. Оскільки цей листок буде весь час присутнім на дереві, то йому весь час буде присвоюватися код змінної довжини. Це і буде код esc, попередній кожному новому нестисненому символу. Дерево буде весь час модифікуватися, і буде змінюватися положення на пустому листку та його коді, але сам цей код буде ідентифікувати кожен новий нестиснений символ у стиснутому файлі. На рис. зображено рух цього коду esc по мірі зростання дерева.

Цей метод також використовується у відомому протоколі V.32 передачі даних по модему зі швидкістю 14400 бод.



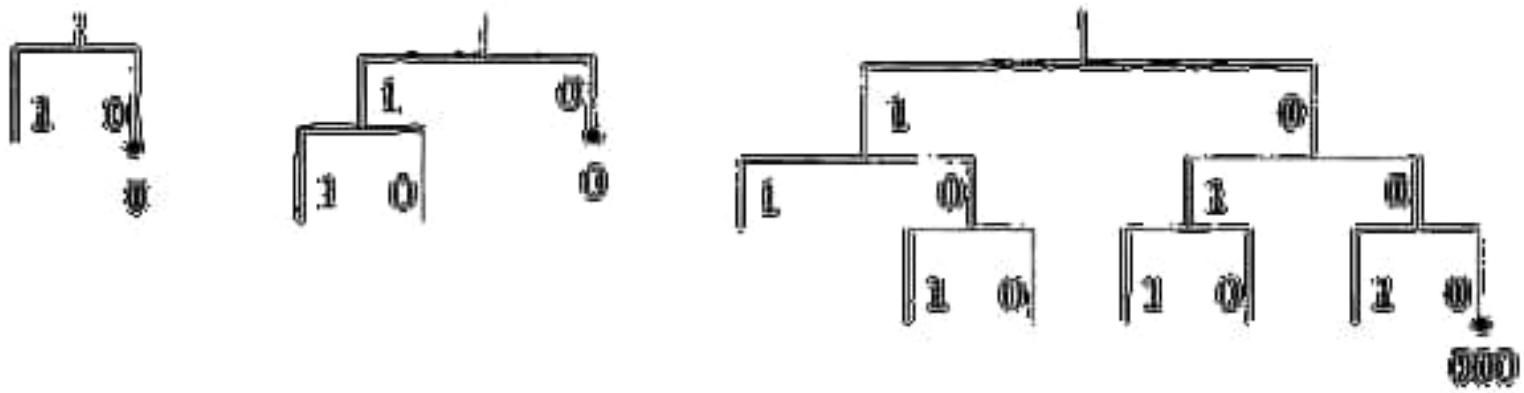
Занепокоєння викликає те, що код esc не повинен бути змінним кодом, використовуваним для символів алфавіта. Оскільки ці коди весь час зазнають зміни, то і код для esc слід також модифікувати. Природний шлях — це додати до дерева ще один порожній листок з постійною нульовою частотою, що б йому весь час присвоювалась гілка з одних нулів. Оскільки цей листок буде весь час присутнім на дереві, то йому весь час буде присвоюватися код змінної довжини. Це і буде код esc, попередній кожному новому нестисненому символу. Дерево буде весь час модифікуватися, і буде змінюватися положення на пустому листку та його коді, але сам цей код буде ідентифікувати кожен новий нестиснений символ у стиснутому файлі. На рис. зображено рух цього коду esc по мірі зростання дерева.

Цей метод також використовується у відомому протоколі V.32 передачі даних по модему зі швидкістю 14400 бод.



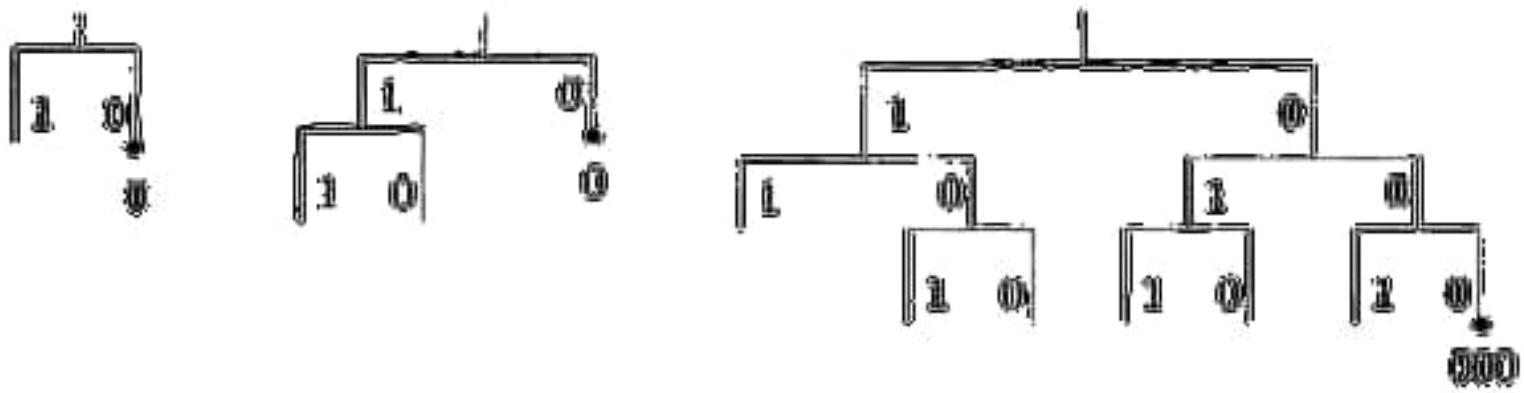
Занепокоєння викликає те, що код esc не повинен бути змінним кодом, використовуваним для символів алфавіта. Оскільки ці коди весь час зазнають зміни, то і код для esc слід також модифікувати. Природний шлях — це додати до дерева ще один порожній листок з постійною нульовою частотою, що б йому весь час присвоювалась гілка з одних нулів. Оскільки цей листок буде весь час присутнім на дереві, то йому весь час буде присвоюватися код змінної довжини. Це і буде код esc, попередній кожному новому нестисненому символу. Дерево буде весь час модифікуватися, і буде змінюватися положення на пустому листку та його коді, але сам цей код буде ідентифікувати кожен новий нестиснений символ у стиснутому файлі. На рис. зображено рух цього коду esc по мірі зростання дерева.

Цей метод також використовується у відомому протоколі V.32 передачі даних по модему зі швидкістю 14400 бод.



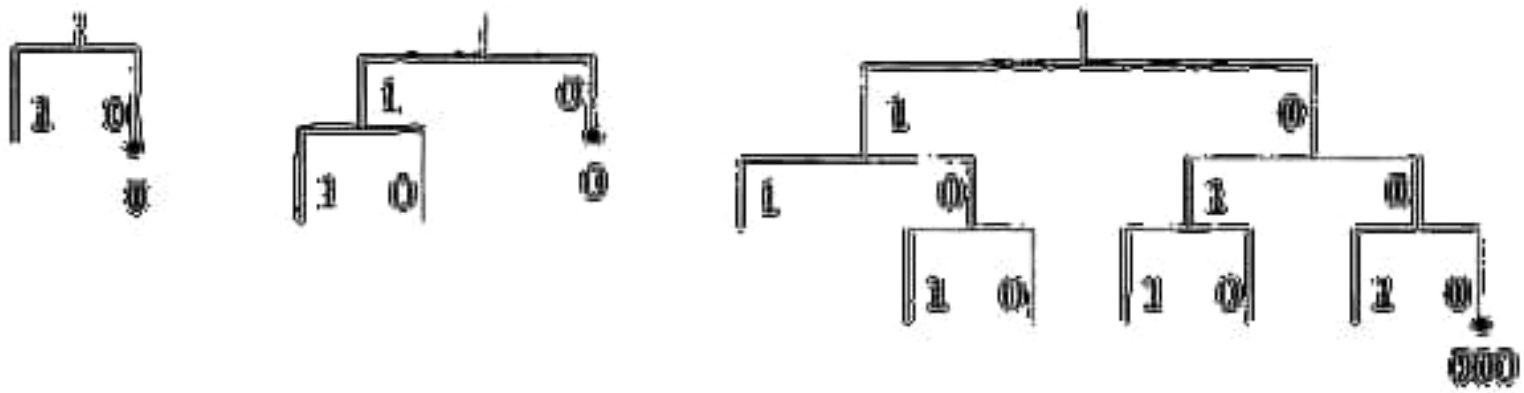
Занепокоєння викликає те, що код esc не повинен бути змінним кодом, використовуваним для символів алфавіта. Оскільки ці коди весь час зазнають зміни, то і код для esc слід також модифікувати. Природний шлях — це додати до дерева ще один порожній листок з постійною нульовою частотою, що б йому весь час присвоювалась гілка з одних нулів. Оскільки цей листок буде весь час присутнім на дереві, то йому весь час буде присвоюватися код змінної довжини. Це і буде код esc, попередній кожному новому нестисненому символу. Дерево буде весь час модифікуватися, і буде змінюватися положення на пустому листку та його коді, але сам цей код буде ідентифікувати кожен новий нестиснений символ у стиснутому файлі. На рис. зображено рух цього коду esc по мірі зростання дерева.

Цей метод також використовується у відомому протоколі V.32 передачі даних по модему зі швидкістю 14400 бод.



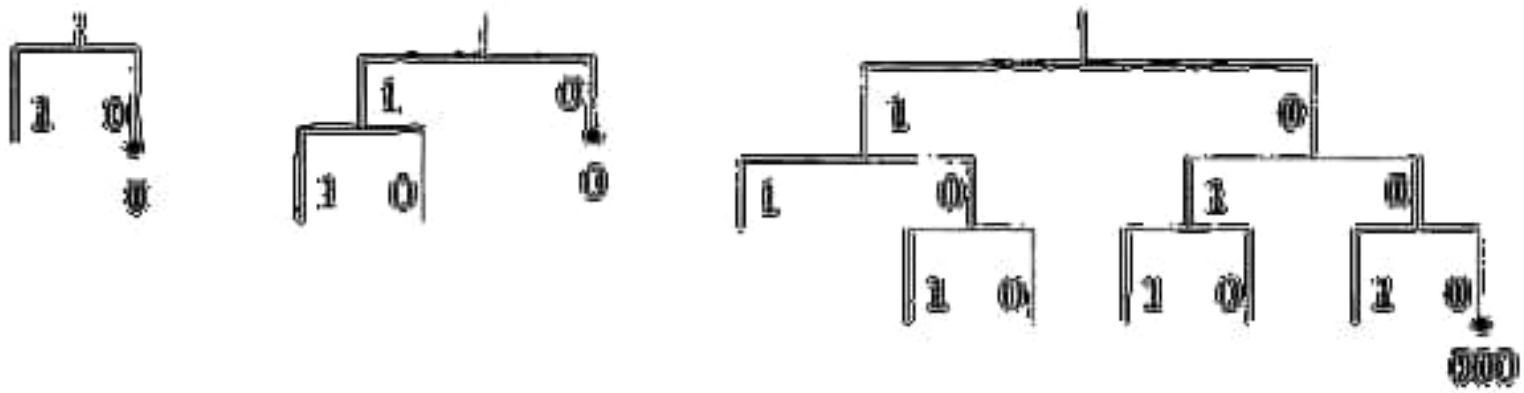
Занепокоєння викликає те, що код esc не повинен бути змінним кодом, використовуваним для символів алфавіта. Оскільки ці коди весь час зазнають зміни, то і код для esc слід також модифікувати. Природний шлях — це додати до дерева ще один порожній листок з постійною нульовою частотою, що б йому весь час присвоювалась гілка з одних нулів. Оскільки цей листок буде весь час присутнім на дереві, то йому весь час буде присвоюватися код змінної довжини. Це і буде код esc, попередній кожному новому нестисненому символу. Дерево буде весь час модифікуватися, і буде змінюватися положення на пустому листку та його коді, але сам цей код буде ідентифікувати кожен новий нестиснений символ у стиснутому файлі. На рис. зображено рух цього коду esc по мірі зростання дерева.

Цей метод також використовується у відомому протоколі V.32 передачі даних по модему зі швидкістю 14400 бод.



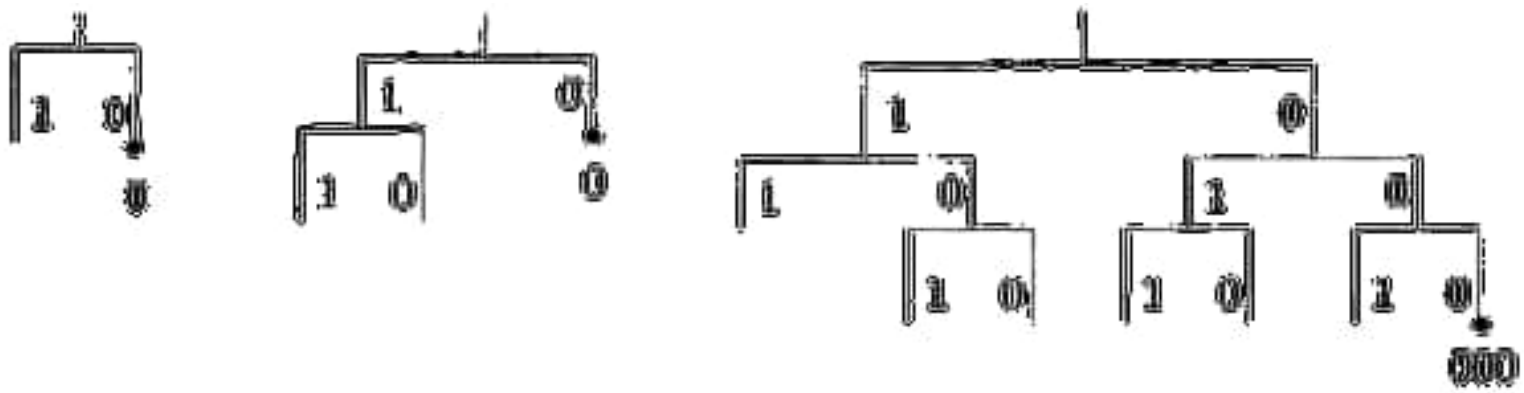
Занепокоєння викликає те, що код esc не повинен бути змінним кодом, використовуваним для символів алфавіта. Оскільки ці коди весь час зазнають зміни, то і код для esc слід також модифікувати. Природний шлях — це додати до дерева ще один порожній листок з постійною нульовою частотою, що б йому весь час присвоювалась гілка з одних нулів. Оскільки цей листок буде весь час присутнім на дереві, то йому весь час буде присвоюватися код змінної довжини. Це і буде код esc, попередній кожному новому нестисненому символу. Дерево буде весь час модифікуватися, і буде змінюватися положення на пустому листку та його коді, але сам цей код буде ідентифікувати кожен новий нестиснений символ у стиснутому файлі. На рис. зображено рух цього коду esc по мірі зростання дерева.

Цей метод також використовується у відомому протоколі V.32 передачі даних по модему зі швидкістю 14400 бод.



Занепокоєння викликає те, що код esc не повинен бути змінним кодом, використовуваним для символів алфавіта. Оскільки ці коди весь час зазнають зміни, то і код для esc слід також модифікувати. Природний шлях — це додати до дерева ще один порожній листок з постійною нульовою частотою, що б йому весь час присвоювалась гілка з одних нулів. Оскільки цей листок буде весь час присутнім на дереві, то йому весь час буде присвоюватися код змінної довжини. Це і буде код esc, попередній кожному новому нестисненому символу. Дерево буде весь час модифікуватися, і буде змінюватися положення на пустому листку та його коді, але сам цей код буде ідентифікувати кожен новий нестиснений символ у стиснутому файлі. На рис. зображено рух цього коду esc по мірі зростання дерева.

Цей метод також використовується у відомому протоколі V.32 передачі даних по модему зі швидкістю 14400 бод.



Занепокоєння викликає те, що код esc не повинен бути змінним кодом, використовуваним для символів алфавіта. Оскільки ці коди весь час зазнають зміни, то і код для esc слід також модифікувати. Природний шлях — це додати до дерева ще один порожній листок з постійною нульовою частотою, що б йому весь час присвоювалась гілка з одних нулів. Оскільки цей листок буде весь час присутнім на дереві, то йому весь час буде присвоюватися код змінної довжини. Це і буде код esc, попередній кожному новому нестисненому символу. Дерево буде весь час модифікуватися, і буде змінюватися положення на пустому листку та його коді, але сам цей код буде ідентифікувати кожен новий нестиснений символ у стиснутому файлі. На рис. зображено рух цього коду esc по мірі зростання дерева.

Цей метод також використовується у відомому протоколі V.32 передачі даних по модему зі швидкістю 14400 бод.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу k дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу k дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.

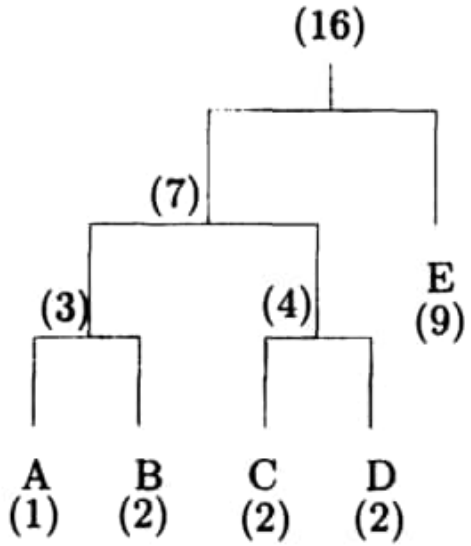
Якщо символи, що стискаються, є кодами ASCII, то їм можна просто присвоїти свої значення для подання в стислому вигляді. У випадку, коли алфавіт має довільний розмір, нестиснені коди двох різних розмірів можна також легко побудувати. Розглянемо, наприклад, алфавіт розміру $n = 24$. Першим 16-и символам можна присвоїти числа від 0 до 15 у їхньому двійковому розкладі. Ці символи вимагатимуть лише 4 біти, але ми закодуємо їх п'ятьма бітами. Символам з номерами від 17 до 24 присвоїмо числа $17 - 16 - 1 = 0$, $18 - 16 - 1 = 1$, і до $24 - 16 - 1 = 7$ у двійковому зображення з 4-х біт. Отже, ми отримаємо шістнадцять 5-бітові коди

$$00000, 00001, \ldots, 01111,$$

за якими слідують вісім 4-бітові коди

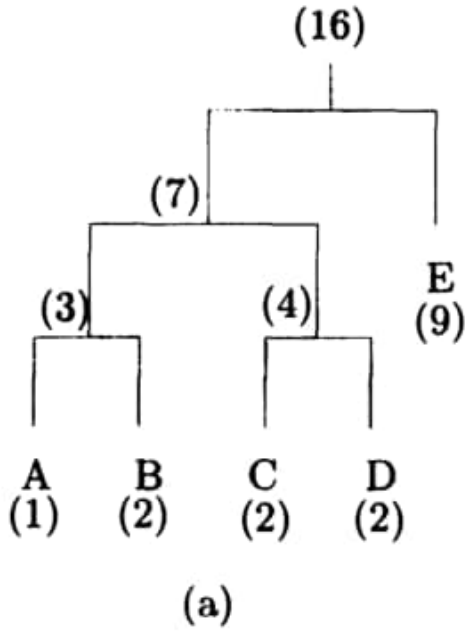
$$0000, 0001, \ldots, 0111.$$

У загальному випадку, якщо є алфавіт $a_1, a_2, \ldots, a_n$, який складається із nп символів, то вибираються такі числа m і r, що $2^m \leqslant n < 2^{m+1}$ і $r = n - 2^m$. Перші 2^m символів кодуються як $(m+1)$-бітові числа від 0 до $2^m - 1$, а інші символи кодуються m-бітовими послідовностями так, що код символу $_k$ дорівнює $k - 2^m - 1$. Такі коди називаються *синфазними двійковими кодами*.



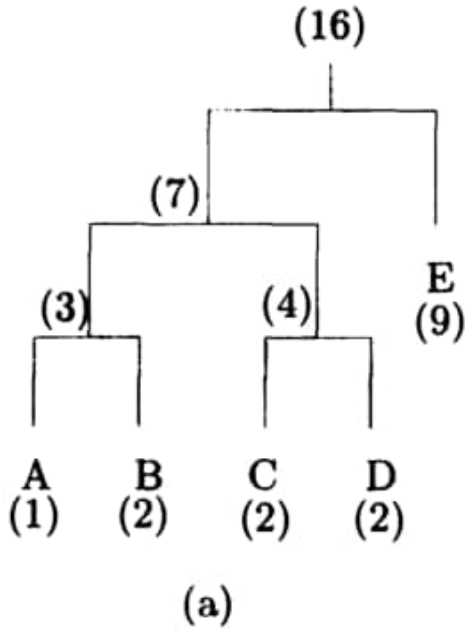
(a)

Основна ідея полягає у перевірці дерева з появою кожного нового вхідного символу. Якщо дерево не є деревом Гаффмана, його слід підправити. Рис. (а) дає уявлення, як це робиться. Дерево на рис. (а) містить п'ять символів A, B, C, D та E. Для всіх символів вказані їхні поточні частоти (у круглих дужках). Властивість Гаффмана означає, що при вивченні дерева по всіх рівнях зліва направо та знизу вгору (від листя до кореня) частоти будуть упорядковані за зростанням (незменшенням). Отже, нижній лівий вузол (A) має найменшу частоту, а верхній правий (корінь) має найбільшу частоту. Цю властивість прийнято називати властивістю *суперництва*.



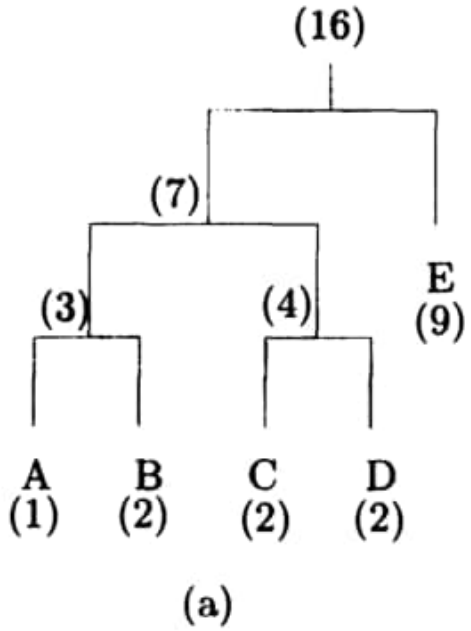
(a)

Основна ідея полягає у перевірці дерева з появою кожного нового вхідного символу. Якщо дерево не є деревом Гаффмана, його слід підправити. Рис. (а) дає уявлення, як це робиться. Дерево на рис. (а) містить п'ять символів A, B, C, D та E. Для всіх символів вказані їхні поточні частоти (у круглих дужках). Властивість Гаффмана означає, що при вивченні дерева по всіх рівнях зліва направо та знизу вгору (від листя до кореня) частоти будуть упорядковані за зростанням (незменшенням). Отже, нижній лівий вузол (A) має найменшу частоту, а верхній правий (корінь) має найбільшу частоту. Цю властивість прийнято називати властивістю *суперництва*.



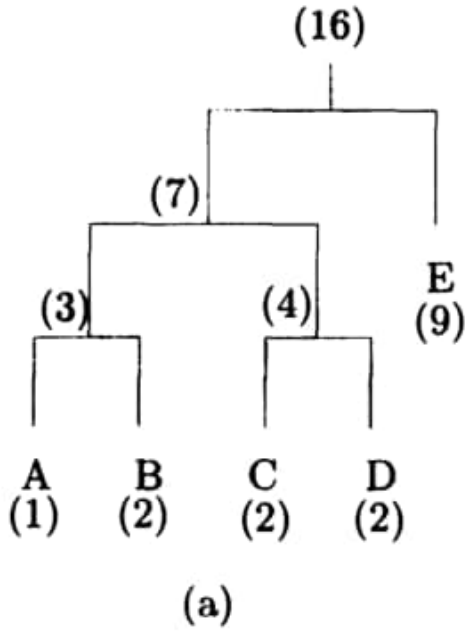
(a)

Основна ідея полягає у перевірці дерева з появою кожного нового вхідного символу. Якщо дерево не є деревом Гаффмана, його слід підправити. Рис. (а) дає уявлення, як це робиться. Дерево на рис. (а) містить п'ять символів A, B, C, D та E. Для всіх символів вказані їхні поточні частоти (у круглих дужках). Властивість Гаффмана означає, що при вивченні дерева по всіх рівнях зліва направо та знизу вгору (від листя до кореня) частоти будуть упорядковані за зростанням (незменшенням). Отже, нижній лівий вузол (A) має найменшу частоту, а верхній правий (корінь) має найбільшу частоту. Цю властивість прийнято називати властивістю *суперництва*.



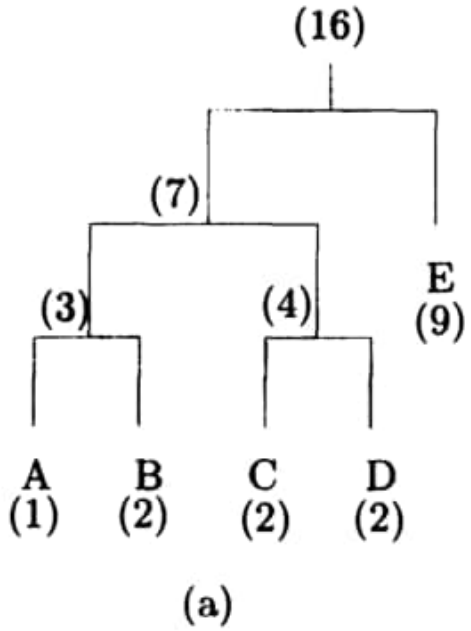
(a)

Основна ідея полягає у перевірці дерева з появою кожного нового вхідного символу. Якщо дерево не є деревом Гаффмана, його слід підправити. Рис. (а) дає уявлення, як це робиться. Дерево на рис. (а) містить п'ять символів A, B, C, D та E. Для всіх символів вказані їхні поточні частоти (у круглих дужках). Властивість Гаффмана означає, що при вивченні дерева по всіх рівнях зліва направо та знизу вгору (від листя до кореня) частоти будуть упорядковані за зростанням (незменшенням). Отже, нижній лівий вузол (A) має найменшу частоту, а верхній правий (корінь) має найбільшу частоту. Цю властивість прийнято називати властивістю *суперництва*.



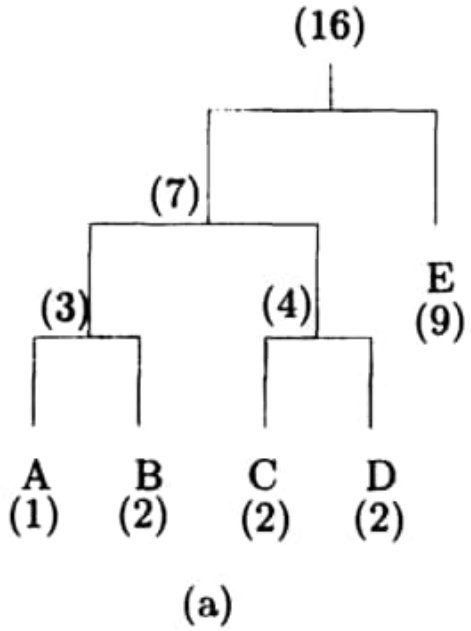
(a)

Основна ідея полягає у перевірці дерева з появою кожного нового вхідного символу. Якщо дерево не є деревом Гаффмана, його слід підправити. Рис. (a) дає уявлення, як це робиться. Дерево на рис. (a) містить п'ять символів A, B, C, D та E. Для всіх символів вказані їхні поточні частоти (у круглих дужках). Властивість Гаффмана означає, що при вивченні дерева по всіх рівнях зліва направо та знизу вгору (від листя до кореня) частоти будуть упорядковані за зростанням (незменшенням). Отже, нижній лівий вузол (A) має найменшу частоту, а верхній правий (корінь) має найбільшу частоту. Цю властивість прийнято називати властивістю *суперництва*.



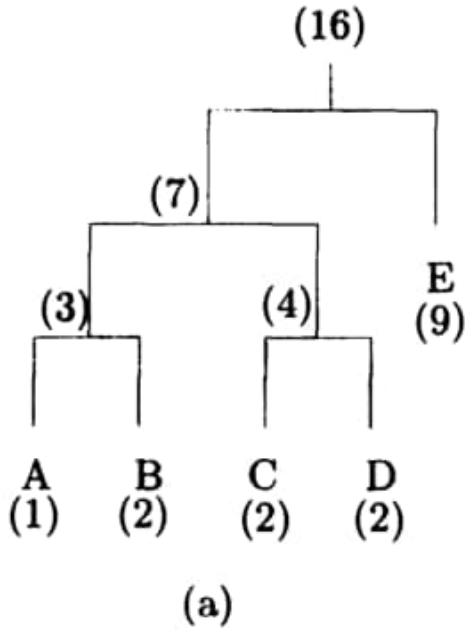
(a)

Основна ідея полягає у перевірці дерева з появою кожного нового вхідного символу. Якщо дерево не є деревом Гаффмана, його слід підправити. Рис. (a) дає уявлення, як це робиться. Дерево на рис. (a) містить п'ять символів A, B, C, D та E. Для всіх символів вказані їхні поточні частоти (у круглих дужках). Властивість Гаффмана означає, що при вивченні дерева по всіх рівнях зліва направо та знизу вгору (від листя до кореня) частоти будуть упорядковані за зростанням (незменшенням). Отже, нижній лівий вузол (A) має найменшу частоту, а верхній правий (корінь) має найбільшу частоту. Цю властивість прийнято називати властивістю *суперництва*.



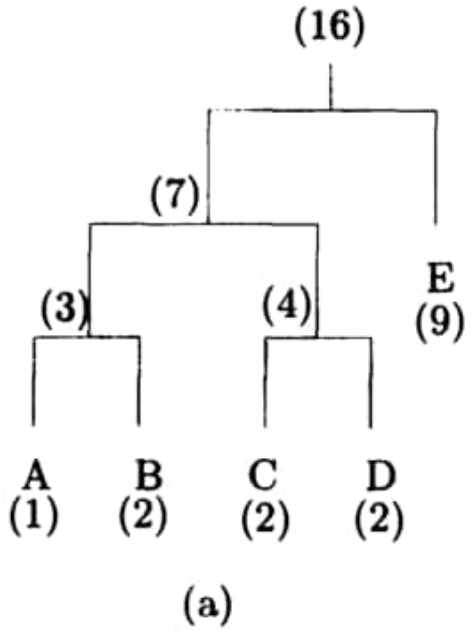
(a)

Основна ідея полягає у перевірці дерева з появою кожного нового вхідного символу. Якщо дерево не є деревом Гаффмана, його слід підправити. Рис. (а) дає уявлення, як це робиться. Дерево на рис. (а) містить п'ять символів A, B, C, D та E. Для всіх символів вказані їхні поточні частоти (у круглих дужках). Властивість Гаффмана означає, що при вивченні дерева по всіх рівнях зліва направо та знизу вгору (від листя до кореня) частоти будуть упорядковані за зростанням (незменшенням). Отже, нижній лівий вузол (A) має найменшу частоту, а верхній правий (корінь) має найбільшу частоту. Цю властивість прийнято називати властивістю *суперництва*.



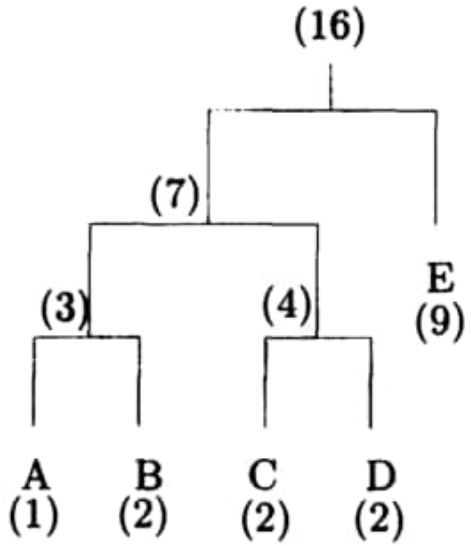
(a)

Основна ідея полягає у перевірці дерева з появою кожного нового вхідного символу. Якщо дерево не є деревом Гаффмана, його слід підправити. Рис. (а) дає уявлення, як це робиться. Дерево на рис. (а) містить п'ять символів A, B, C, D та E. Для всіх символів вказані їхні поточні частоти (у круглих дужках). Властивість Гаффмана означає, що при вивченні дерева по всіх рівнях зліва направо та знизу вгору (від листя до кореня) частоти будуть упорядковані за зростанням (незменшенням). Отже, нижній лівий вузол (A) має найменшу частоту, а верхній правий (корінь) має найбільшу частоту. Цю властивість прийнято називати властивістю *суперництва*.



(a)

Основна ідея полягає у перевірці дерева з появою кожного нового вхідного символу. Якщо дерево не є деревом Гаффмана, його слід підправити. Рис. (а) дає уявлення, як це робиться. Дерево на рис. (а) містить п'ять символів A, B, C, D та E. Для всіх символів вказані їхні поточні частоти (у круглих дужках). Властивість Гаффмана означає, що при вивченні дерева по всіх рівнях зліва направо та знизу вгору (від листя до кореня) частоти будуть упорядковані за зростанням (незменшенням). Отже, нижній лівий вузол (A) має найменшу частоту, а верхній правий (корінь) має найбільшу частоту. Цю властивість прийнято називати властивістю *суперництва*.



(a)

Основна ідея полягає у перевірці дерева з появою кожного нового вхідного символу. Якщо дерево не є деревом Гаффмана, його слід підправити. Рис. (а) дає уявлення, як це робиться. Дерево на рис. (а) містить п'ять символів A, B, C, D та E. Для всіх символів вказані їхні поточні частоти (у круглих дужках). Властивість Гаффмана означає, що при вивченні дерева по всіх рівнях зліва направо та знизу вгору (від листя до кореня) частоти будуть упорядковані за зростанням (незменшенням). Отже, нижній лівий вузол (A) має найменшу частоту, а верхній правий (корінь) має найбільшу частоту. Цю властивість прийнято називати властивістю *суперництва*.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

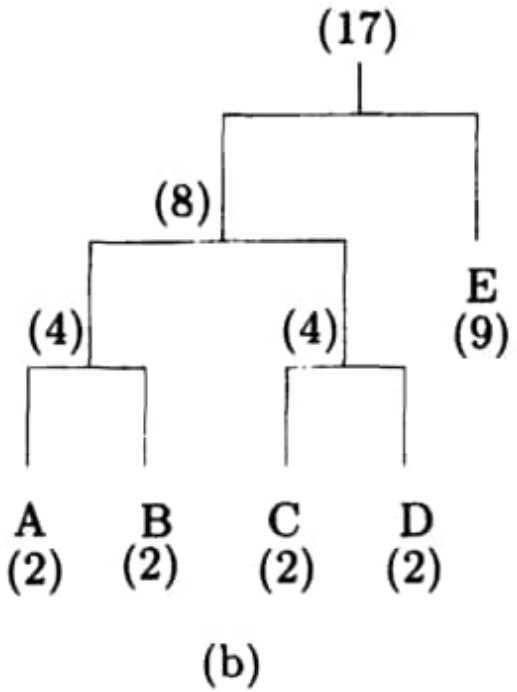
Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.

Причина цієї властивості полягає в тому, що символи з більшими частотами повинні мати коротші коди, а отже, перебувати на дереві на більш високому рівні. Вимога для частот бути впорядкованими одному рівні встановлюється для визначеності. В принципі, без неї можна обійтися, але вона полегшує процес побудови дерева.

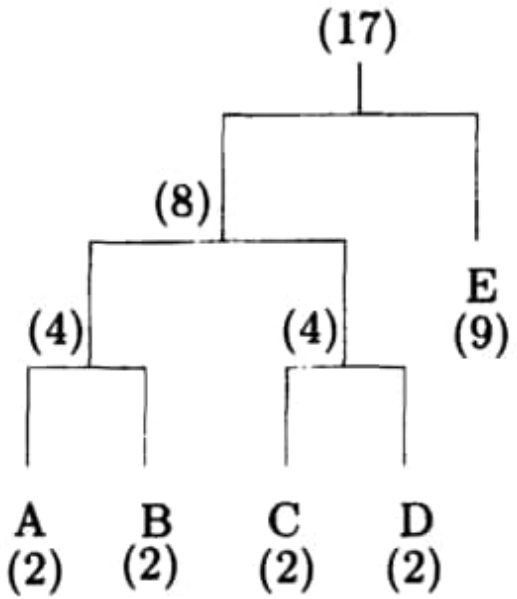
Наведемо послідовність операцій під час модифікації дерева. Цикл починається у поточному вузлі (відповідному до нового вхідного символу). Цей вузол буде листком, який ми позначимо X, а його частота нехай буде F. На кожній наступній ітерації циклу потрібно зробити такі три дії.

1. Порівняти вузол X з його найближчими сусідами на дереві (праворуч і зверху). Якщо безпосередній сусід має частоту $F+1$ або вище, то вузли залишаються впорядкованими і робити нічого не треба. В іншому випадку, якийсь сусід має таку ж або меншу частоту, ніж вузол X. У цьому випадку слід поміняти місцями X і останній вузол у цій групі (за винятком того, що вузол X не треба міняти з його батьком).
2. Збільшити частоту вузла X з F до $F+1$. Збільшити на одиницю частоту всіх його батьків.
3. Якщо X є коренем, то цикл зупиняється. У протилежному випадку він повторюється для вузла, що є батьком вузла X.



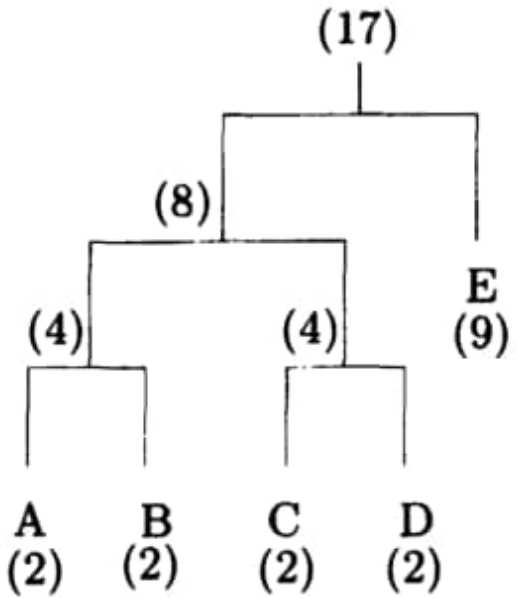
(b)

На рис. (b) зображено дерево після збільшення частоти вузла A з 1 до 2. Легко простежити, як описані вище три дії впливають на збільшення частоти всіх предків вузла A. Місця вузлів не змінюються у цьому простому випадку, оскільки частота вузла A не перевищила частоту його найближчого сусіда праворуч B.



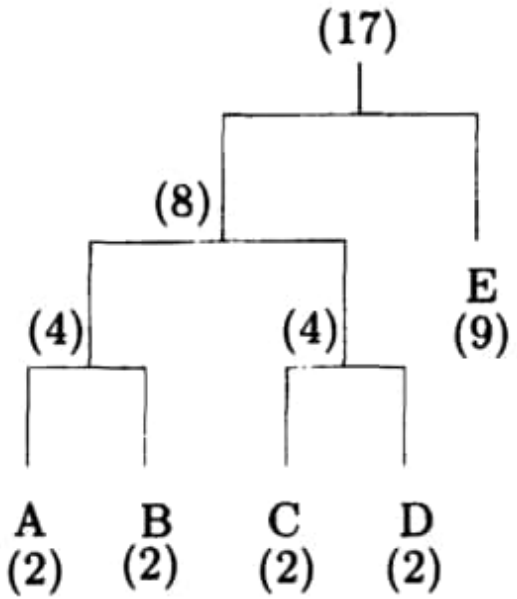
(b)

На рис. (b) зображено дерево після збільшення частоти вузла A з 1 до 2. Легко простежити, як описані вище три дії впливають на збільшення частоти всіх предків вузла A. Місця вузлів не змінюються у цьому простому випадку, оскільки частота вузла A не перевищила частоту його найближчого сусіда праворуч B.



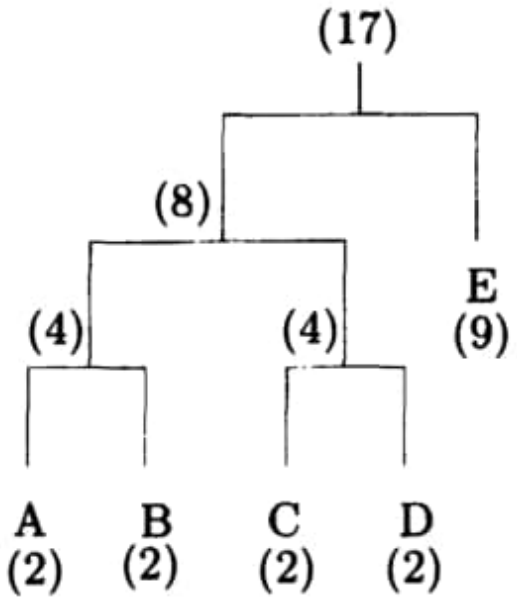
(b)

На рис. (b) зображено дерево після збільшення частоти вузла A з 1 до 2. Легко простежити, як описані вище три дії впливають на збільшення частоти всіх предків вузла A. Місця вузлів не змінюються у цьому простому випадку, оскільки частота вузла A не перевищила частоту його найближчого сусіда праворуч B.



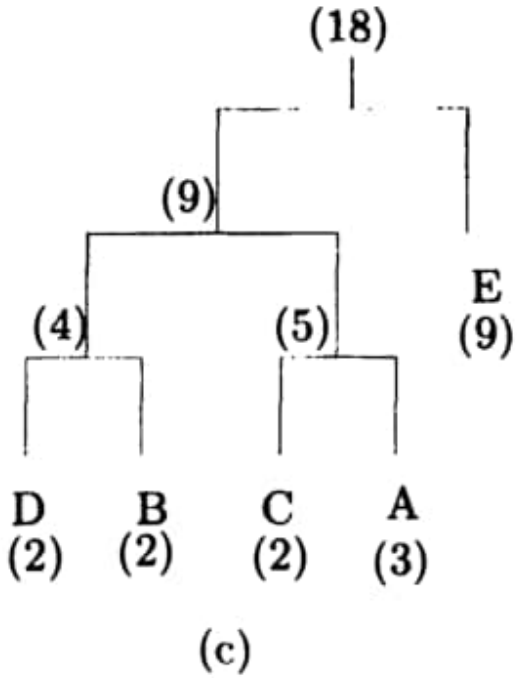
(b)

На рис. (b) зображено дерево після збільшення частоти вузла A з 1 до 2. Легко простежити, як описані вище три дії впливають на збільшення частоти всіх предків вузла A. Місця вузлів не змінюються у цьому простому випадку, оскільки частота вузла A не перевищила частоту його найближчого сусіда праворуч B.



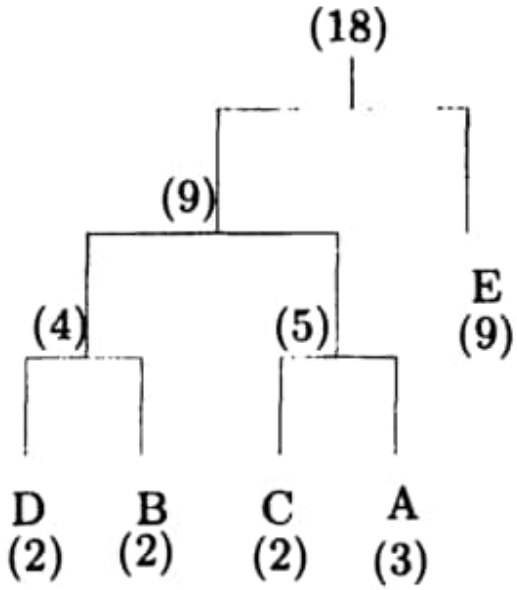
(b)

На рис. (b) зображено дерево після збільшення частоти вузла A з 1 до 2. Легко простежити, як описані вище три дії впливають на збільшення частоти всіх предків вузла A. Місця вузлів не змінюються у цьому простому випадку, оскільки частота вузла A не перевищила частоту його найближчого сусіда праворуч B.



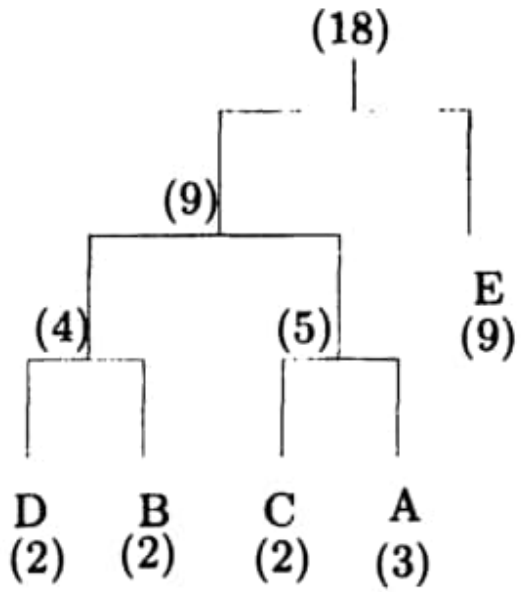
(c)

На рис. (c) показано, що відбудеться при наступному збільшенні частоти вузла A з 2 до 3. Три вузли, що йдуть за A, а саме. B, C і D мали частоту 2, а тому вузол A переставлений з D. Після чого частоти всіх предків вузла A збільшені на 1, і кожен порівняний зі своїми сусідами, але більше на дереві нікого переставляти не треба.



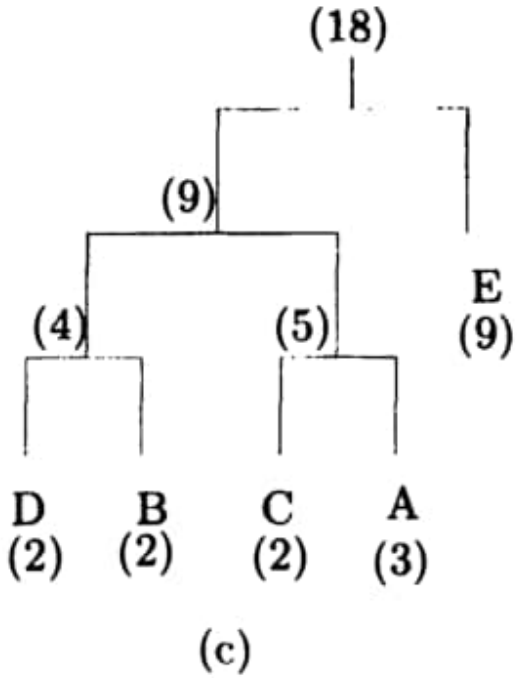
(c)

На рис. (c) показано, що відбудеться при наступному збільшенні частоти вузла A з 2 до 3. Три вузли, що йдуть за A, а саме. B, C і D мали частоту 2, а тому вузол A переставлений з D. Після чого частоти всіх предків вузла A збільшені на 1, і кожен порівняний зі своїми сусідами, але більше на дереві нікого переставляти не треба.



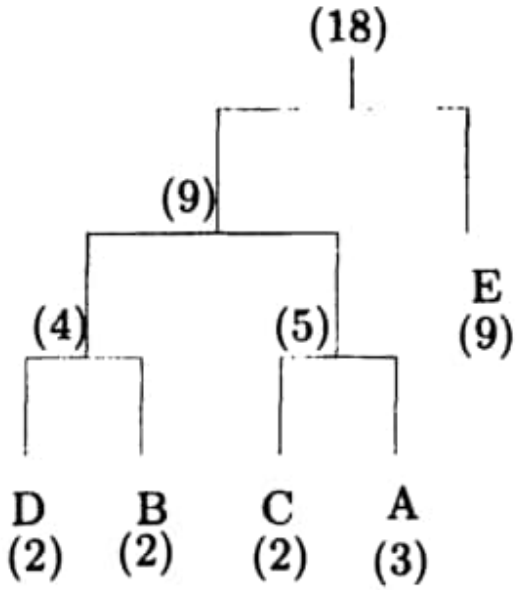
(c)

На рис. (c) показано, що відбудеться при наступному збільшенні частоти вузла A з 2 до 3. Три вузли, що йдуть за A, а саме. B, C і D мали частоту 2, а тому вузол A переставлений з D. Після чого частоти всіх предків вузла A збільшені на 1, і кожен порівняний зі своїми сусідами, але більше на дереві нікого переставляти не треба.



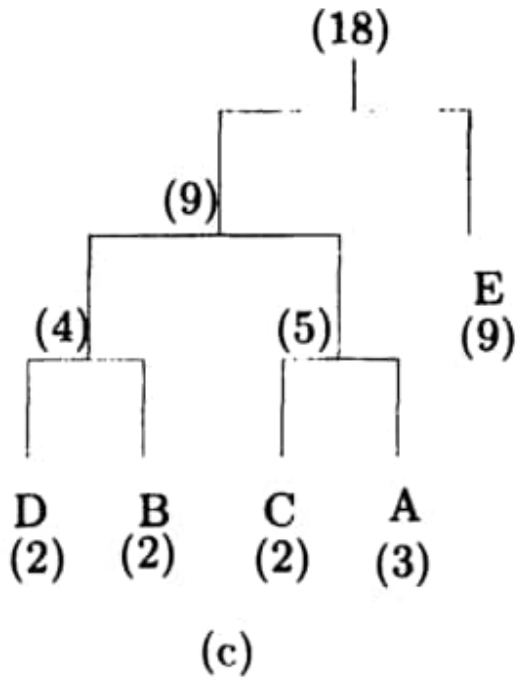
(c)

На рис. (c) показано, що відбудеться при наступному збільшенні частоти вузла A з 2 до 3. Три вузли, що йдуть за A, а саме. B, C і D мали частоту 2, а тому вузол A переставлений з D. Після чого частоти всіх предків вузла A збільшені на 1, і кожен порівняний зі своїми сусідами, але більше на дереві нікого переставляти не треба.



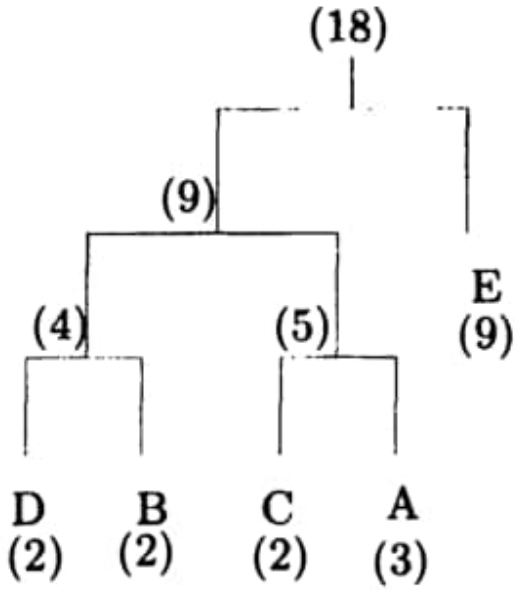
(c)

На рис. (c) показано, що відбудеться при наступному збільшенні частоти вузла A з 2 до 3. Три вузли, що йдуть за A, а саме. B, C і D мали частоту 2, а тому вузол A переставлений з D. Після чого частоти всіх предків вузла A збільшені на 1, і кожен порівняний зі своїми сусідами, але більше на дереві нікого переставляти не треба.



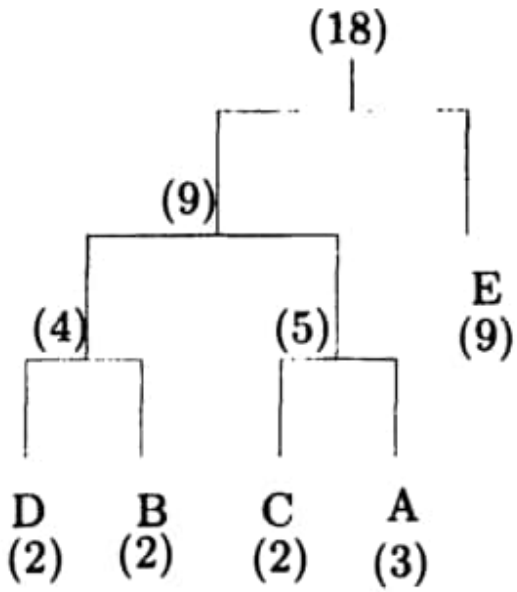
(c)

На рис. (c) показано, що відбудеться при наступному збільшенні частоти вузла A з 2 до 3. Три вузли, що йдуть за A, а саме. B, C і D мали частоту 2, а тому вузол A переставлений з D. Після чого частоти всіх предків вузла A збільшені на 1, і кожен порівняний зі своїми сусідами, але більше на дереві нікого переставляти не треба.



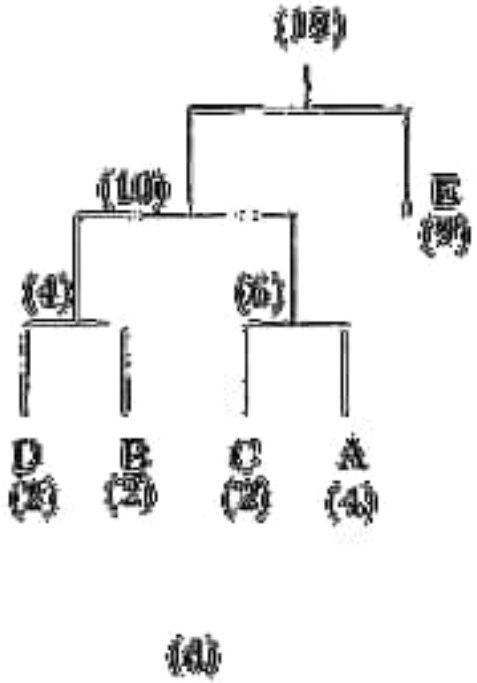
(c)

На рис. (c) показано, що відбудеться при наступному збільшенні частоти вузла A з 2 до 3. Три вузли, що йдуть за A, а саме. B, C і D мали частоту 2, а тому вузол A переставлений з D. Після чого частоти всіх предків вузла A збільшені на 1, і кожен порівняний зі своїми сусідами, але більше на дереві нікого переставляти не треба.



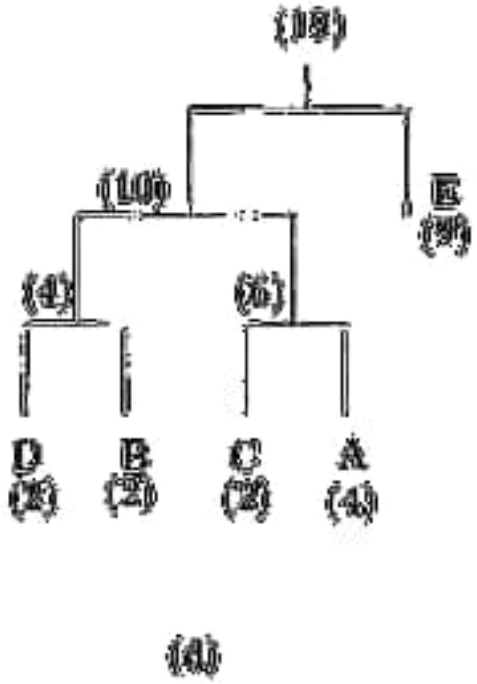
(c)

На рис. (c) показано, що відбудеться при наступному збільшенні частоти вузла A з 2 до 3. Три вузли, що йдуть за A, а саме. B, C і D мали частоту 2, а тому вузол A переставлений з D. Після чого частоти всіх предків вузла A збільшені на 1, і кожен порівняний зі своїми сусідами, але більше на дереві нікого переставляти не треба.



(d)

Рис. (d) зображує дерево після збільшення частоти до 4. Оскільки вузол A є поточним, то його частота (дорівнює поки що 3) порівнюється з частотою сусіда зверху (4), і дерево не змінюється. Частота вузла A збільшується на одиницю разом із частотами всіх нащадків.



(d)

Рис. (d) зображує дерево після збільшення частоти до 4. Оскільки вузол A є поточним, то його частота (дорівнює поки що 3) порівнюється з частотою сусіда зверху (4), і дерево не змінюється. Частота вузла A збільшується на одиницю разом із частотами всіх нащадків.

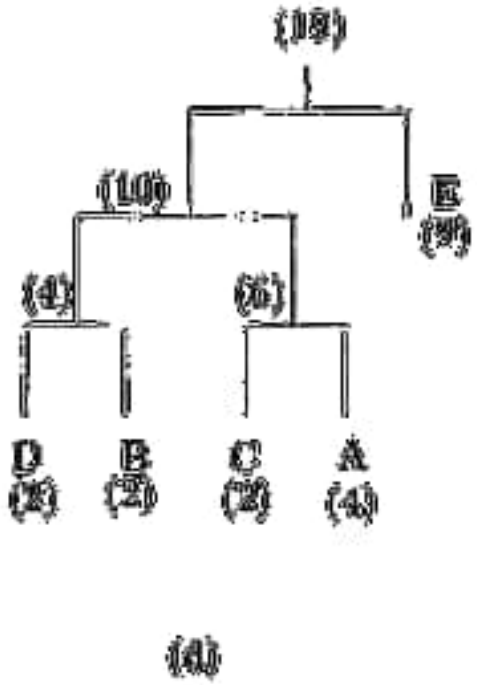


Рис. (d) зображує дерево після збільшення частоти до 4. Оскільки вузол A є поточним, то його частота (дорівнює поки що 3) порівнюється з частотою сусіда зверху (4), і дерево не змінюється. Частота вузла A збільшується на одиницю разом із частотами всіх нащадків.

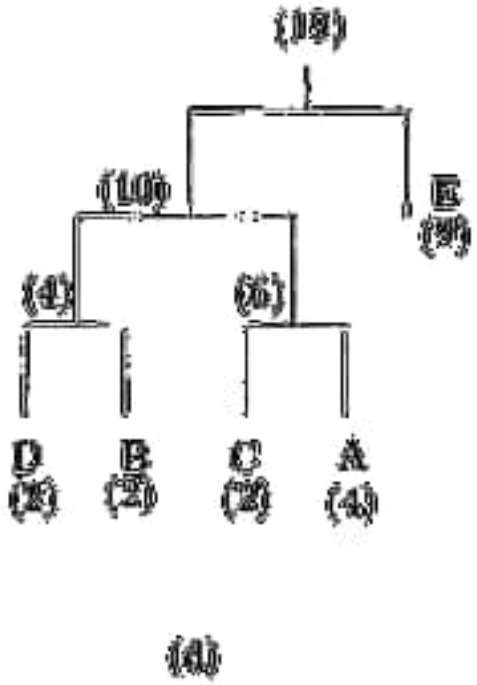


Рис. (d) зображує дерево після збільшення частоти до 4. Оскільки вузол A є поточним, то його частота (дорівнює поки що 3) порівнюється з частотою сусіда зверху (4), і дерево не змінюється. Частота вузла A збільшується на одиницю разом із частотами всіх нащадків.

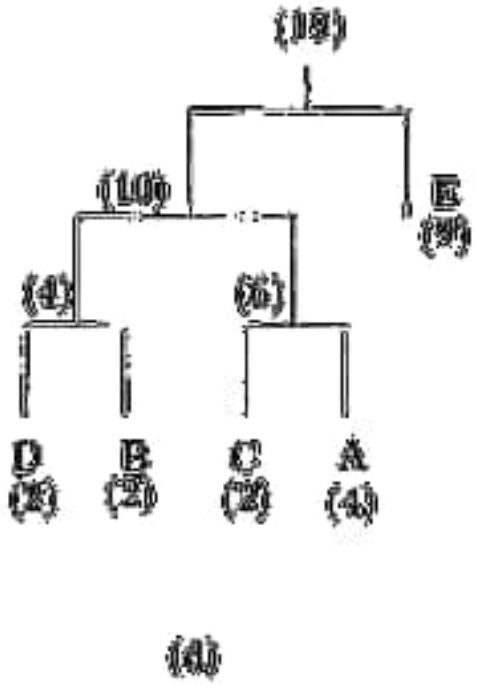


Рис. (d) зображує дерево після збільшення частоти до 4. Оскільки вузол A є поточним, то його частота (дорівнює поки що 3) порівнюється з частотою сусіда зверху (4), і дерево не змінюється. Частота вузла A збільшується на одиницю разом із частотами всіх нащадків.

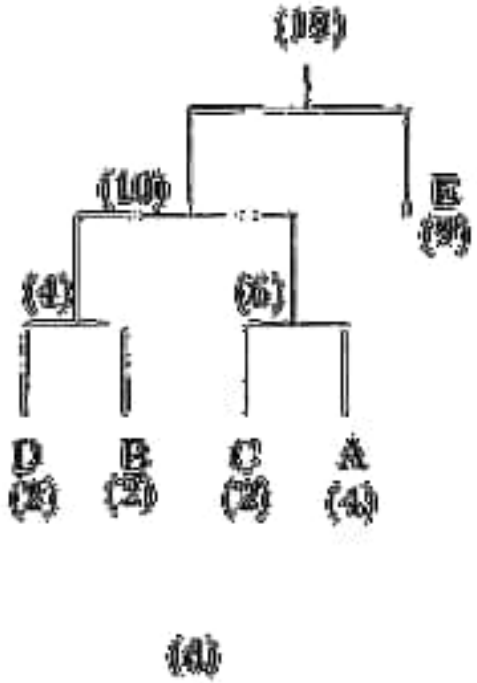
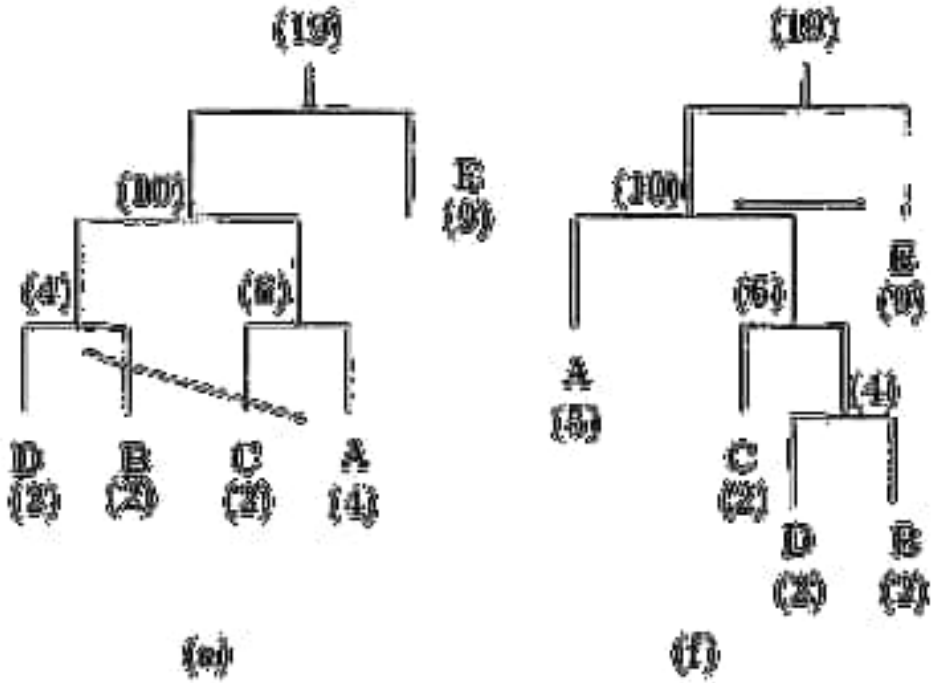
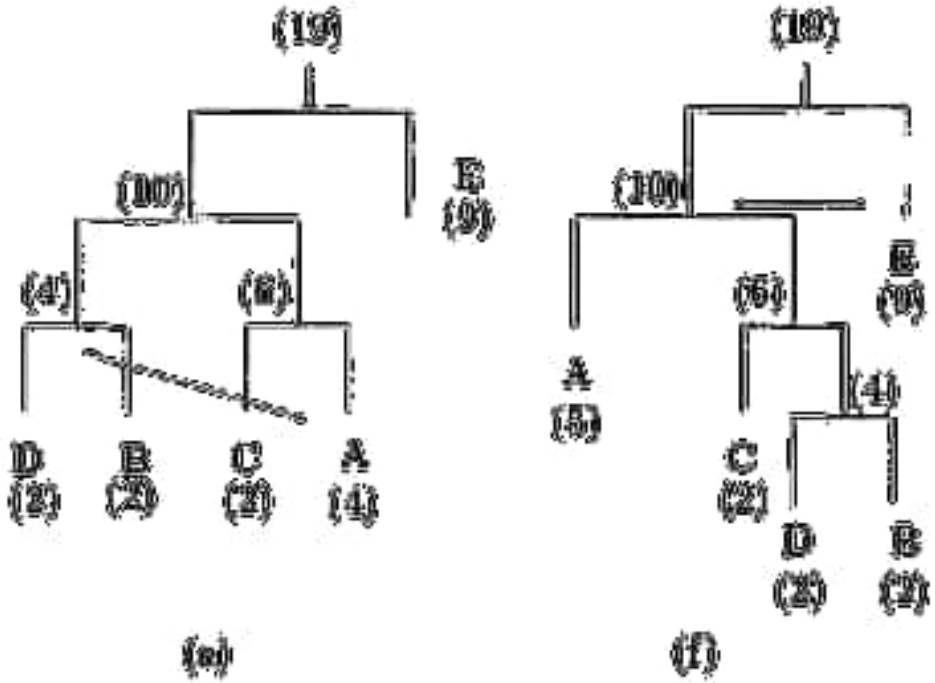


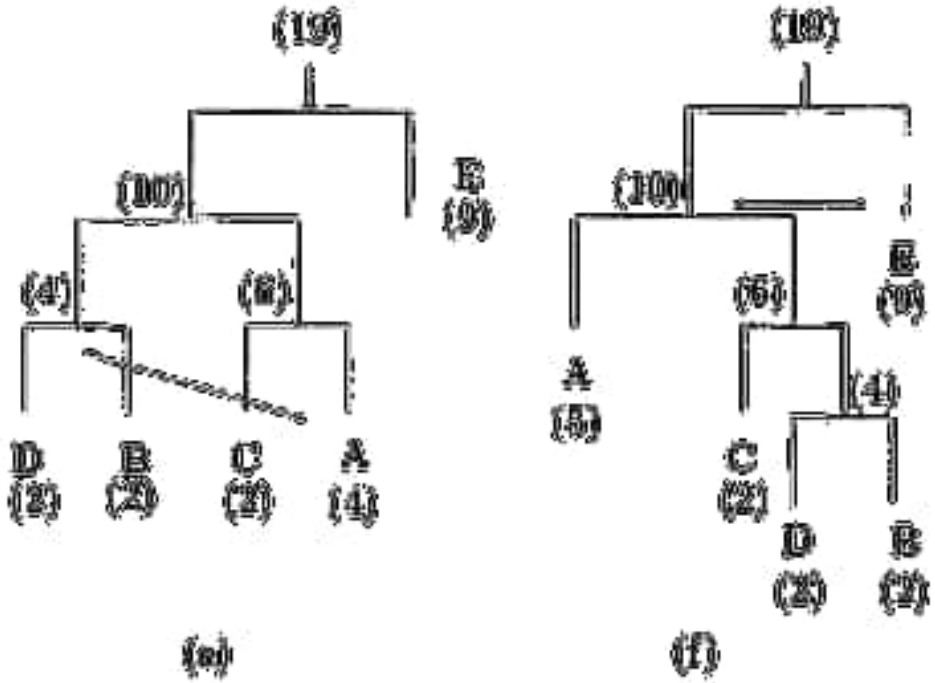
Рис. (d) зображує дерево після збільшення частоти до 4. Оскільки вузол A є поточним, то його частота (дорівнює поки що 3) порівнюється з частотою сусіда зверху (4), і дерево не змінюється. Частота вузла A збільшується на одиницю разом із частотами всіх нащадків.



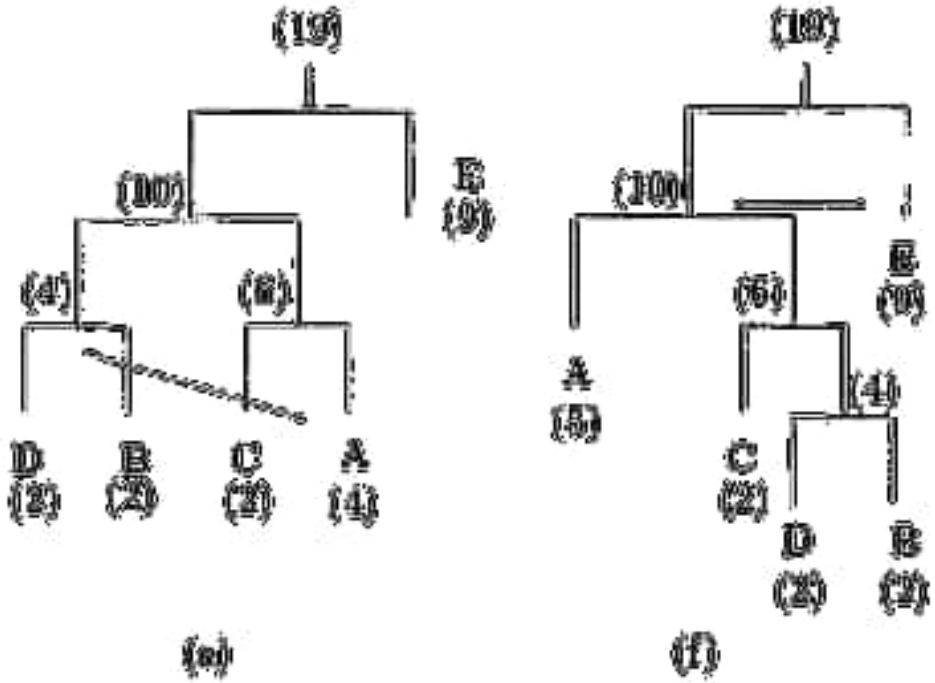
На рис. (e) вузол A знову є поточним. Його частота (4) дорівнює частоті сусіда згори, тому їх слід поміняти місцями. Це зроблено на рис. (f), де частота вузла A вже дорівнює 5.



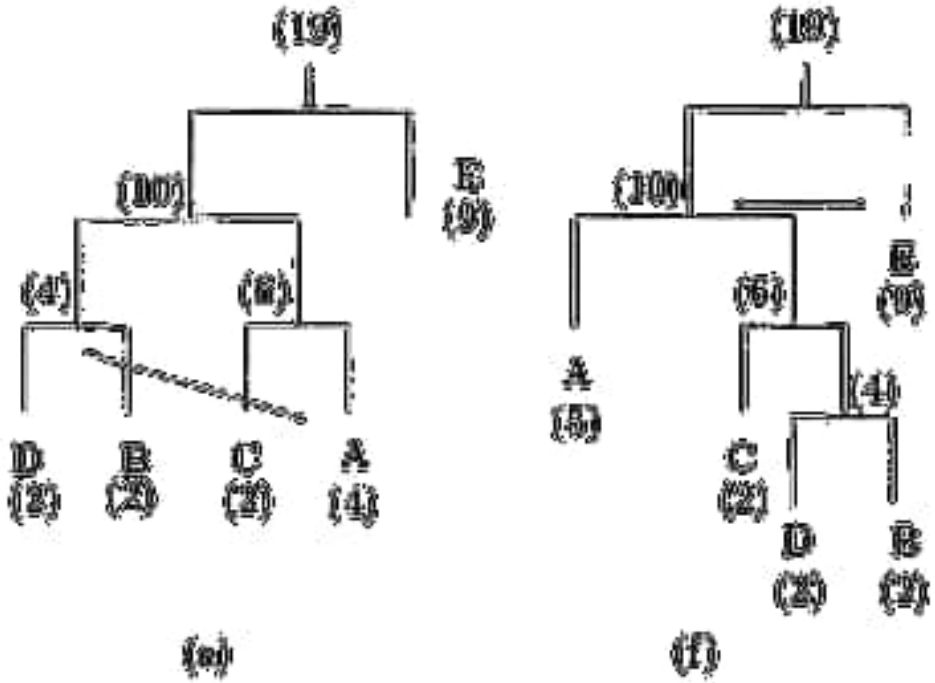
На рис. (e) вузол A знову є поточним. Його частота (4) дорівнює частоті сусіда згори, тому їх слід поміняти місцями. Це зроблено на рис. (f), де частота вузла A вже дорівнює 5.



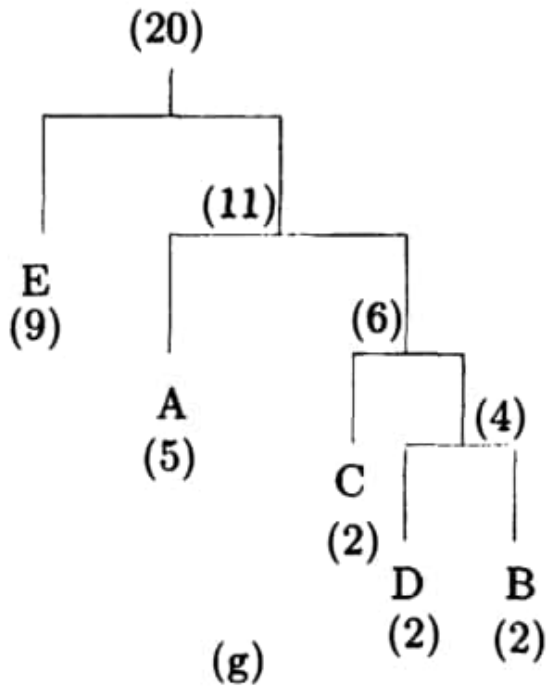
На рис. (e) вузол A знову є поточним. Його частота (4) дорівнює частоті сусіда згори, тому їх слід поміняти місцями. Це зроблено на рис. (f), де частота вузла A вже дорівнює 5.



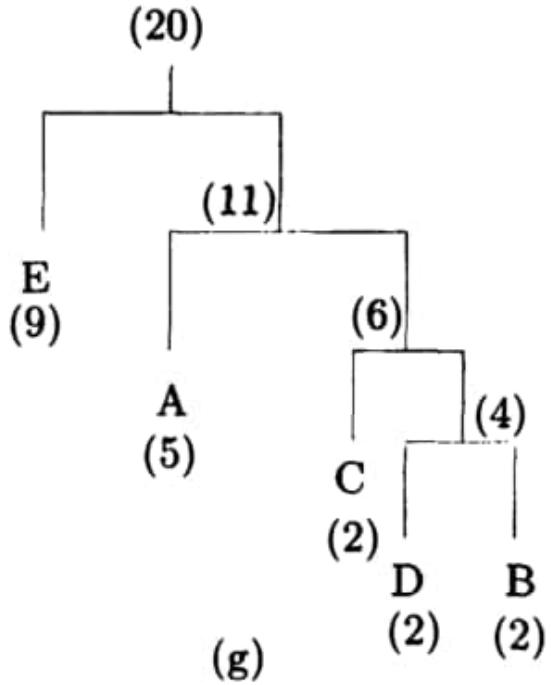
На рис. (e) вузол A знову є поточним. Його частота (4) дорівнює частоті сусіда згори, тому їх слід поміняти місцями. Це зроблено на рис. (f), де частота вузла A вже дорівнює 5.



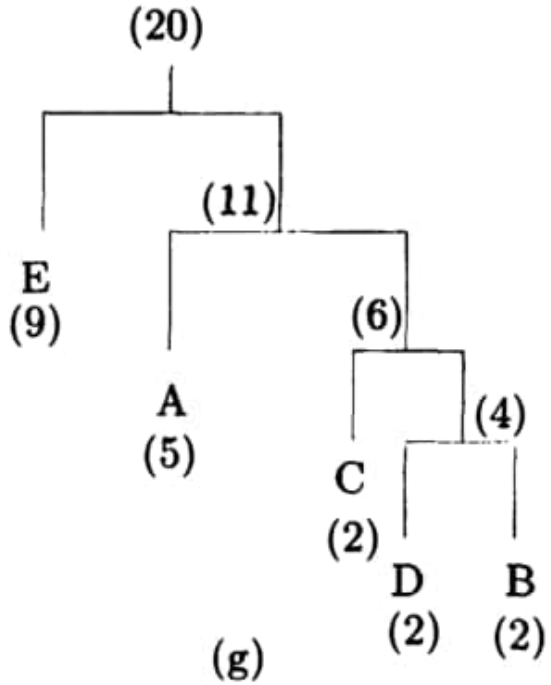
На рис. (e) вузол A знову є поточним. Його частота (4) дорівнює частоті сусіда згори, тому їх слід поміняти місцями. Це зроблено на рис. (f), де частота вузла A вже дорівнює 5.



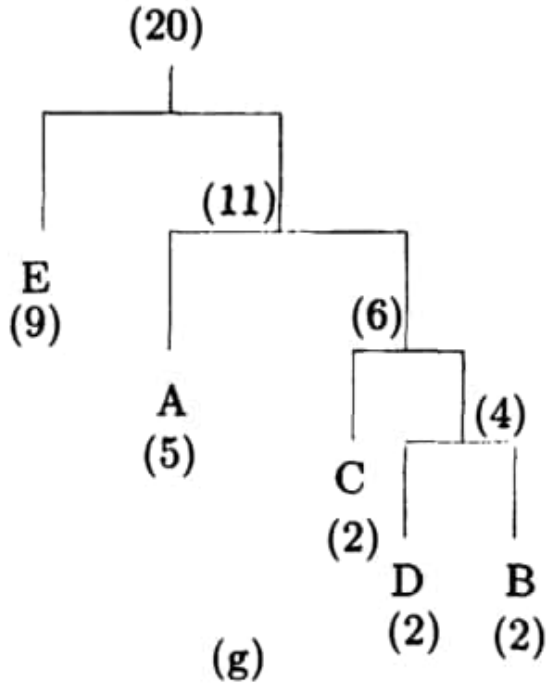
Наступний крок перевіряє батька вузла A з частотою 10. Його слід переставити з його сусідом справа E, у якого частота 9. У результаті виходить скінченне дерево, зображене на рис. (g).



Наступний крок перевіряє батька вузла A з частотою 10. Його слід переставити з його сусідом справа E, у якого частота 9. У результаті виходить скінченне дерево, зображене на рис. (g).



Наступний крок перевіряє батька вузла A з частотою 10. Його слід переставити з його сусідом справа E, у якого частота 9. У результаті виходить скінченне дерево, зображене на рис. (g).



Наступний крок перевіряє батька вузла A з частотою 10. Його слід переставити з його сусідом справа E, у якого частота 9. У результаті виходить скінченне дерево, зображене на рис. (g).

Лічильники частот символів зберігаються на дереві у вигляді чисел із фіксованою розрядністю. Поля розрядів можуть бути переповненими. Число без знака з 16 двійкових розрядів може накопичувати лічильник до числа $2^{16} - 1 = 65535$. Найпростіше вирішення цієї проблеми полягає в спостереженні за зростанням лічильника в корені дерева, і, коли він досягає максимального значення, слід зробити зміну масштабу всіх лічильників шляхом їхнього цілочисельного ділення на 2. Насправді це зазвичай робиться розподілом лічильників листя з наступним обчисленням лічильників всіх внутрішніх вузлів дерева. Кожен внутрішній вузол дорівнює сумі своїх нащадків. Однак при такому методі цілочисельного ділення знижується точність обчислень, що може призвести до порушення якості суперництва вузлів дерева.

Лічильники частот символів зберігаються на дереві у вигляді чисел із фіксованою розрядністю. Поля розрядів можуть бути переповненими. Число без знака з 16 двійкових розрядів може накопичувати лічильник до числа $2^{16} - 1 = 65535$. Найпростіше вирішення цієї проблеми полягає в спостереженні за зростанням лічильника в корені дерева, і, коли він досягає максимального значення, слід зробити зміну масштабу всіх лічильників шляхом їхнього цілочисельного ділення на 2. Насправді це зазвичай робиться розподілом лічильників листя з наступним обчисленням лічильників всіх внутрішніх вузлів дерева. Кожен внутрішній вузол дорівнює сумі своїх нащадків. Однак при такому методі цілочисельного ділення знижується точність обчислень, що може призвести до порушення якості суперництва вузлів дерева.

Лічильники частот символів зберігаються на дереві у вигляді чисел із фіксованою розрядністю. Поля розрядів можуть бути переповненими. Число без знака з 16 двійкових розрядів може накопичувати лічильник до числа $2^{16} - 1 = 65535$. Найпростіше вирішення цієї проблеми полягає в спостереженні за зростанням лічильника в корені дерева, і, коли він досягає максимального значення, слід зробити зміну масштабу всіх лічильників шляхом їхнього цілочисельного ділення на 2. Насправді це зазвичай робиться розподілом лічильників листя з наступним обчисленням лічильників всіх внутрішніх вузлів дерева. Кожен внутрішній вузол дорівнює сумі своїх нащадків. Однак при такому методі цілочисельного ділення знижується точність обчислень, що може призвести до порушення якості суперництва вузлів дерева.

Лічильники частот символів зберігаються на дереві у вигляді чисел із фіксованою розрядністю. Поля розрядів можуть бути переповненими. Число без знака з 16 двійкових розрядів може накопичувати лічильник до числа $2^{16} - 1 = 65535$. Найпростіше вирішення цієї проблеми полягає в спостереженні за зростанням лічильника в корені дерева, і, коли він досягає максимального значення, слід зробити зміну масштабу всіх лічильників шляхом їхнього цілочисельного ділення на 2. Насправді це зазвичай робиться розподілом лічильників листя з наступним обчисленням лічильників всіх внутрішніх вузлів дерева. Кожен внутрішній вузол дорівнює сумі своїх нащадків. Однак при такому методі цілочисельного ділення знижується точність обчислень, що може призвести до порушення якості суперництва вузлів дерева.

Лічильники частот символів зберігаються на дереві у вигляді чисел із фіксованою розрядністю. Поля розрядів можуть бути переповненими. Число без знака з 16 двійкових розрядів може накопичувати лічильник до числа $2^{16} - 1 = 65535$. Найпростіше вирішення цієї проблеми полягає в спостереженні за зростанням лічильника в корені дерева, і, коли він досягає максимального значення, слід зробити зміну масштабу всіх лічильників шляхом їхнього цілочисельного ділення на 2. Насправді це зазвичай робиться розподілом лічильників листя з наступним обчисленням лічильників всіх внутрішніх вузлів дерева. Кожен внутрішній вузол дорівнює сумі своїх нащадків. Однак при такому методі цілочисельного ділення знижується точність обчислень, що може призвести до порушення якості суперництва вузлів дерева.

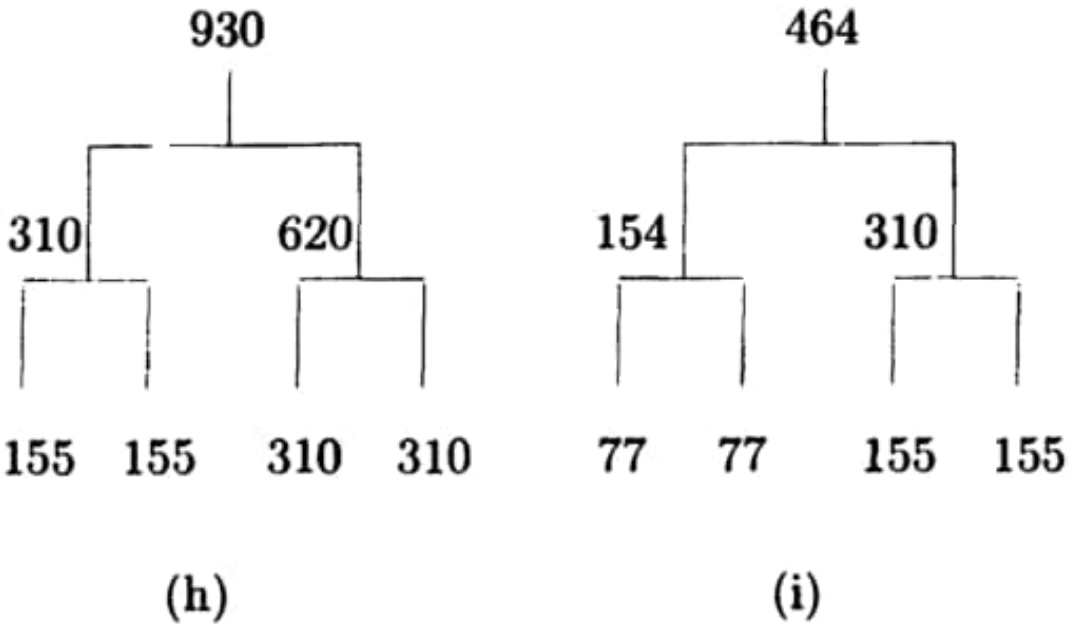
Лічильники частот символів зберігаються на дереві у вигляді чисел із фіксованою розрядністю. Поля розрядів можуть бути переповненими. Число без знака з 16 двійкових розрядів може накопичувати лічильник до числа $2^{16} - 1 = 65535$. Найпростіше вирішення цієї проблеми полягає в спостереженні за зростанням лічильника в корені дерева, і, коли він досягає максимального значення, слід зробити зміну масштабу всіх лічильників шляхом їхнього цілочисельного ділення на 2. Насправді це зазвичай робиться розподілом лічильників листя з наступним обчисленням лічильників всіх внутрішніх вузлів дерева. Кожен внутрішній вузол дорівнює сумі своїх нащадків. Однак при такому методі цілочисельного ділення знижується точність обчислень, що може призвести до порушення якості суперництва вузлів дерева.

Лічильники частот символів зберігаються на дереві у вигляді чисел із фіксованою розрядністю. Поля розрядів можуть бути переповненими. Число без знака з 16 двійкових розрядів може накопичувати лічильник до числа $2^{16} - 1 = 65535$. Найпростіше вирішення цієї проблеми полягає в спостереженні за зростанням лічильника в корені дерева, і, коли він досягає максимального значення, слід зробити зміну масштабу всіх лічильників шляхом їхнього цілочисельного ділення на 2. Насправді це зазвичай робиться розподілом лічильників листя з наступним обчисленням лічильників всіх внутрішніх вузлів дерева. Кожен внутрішній вузол дорівнює сумі своїх нащадків. Однак при такому методі цілочисельного ділення знижується точність обчислень, що може призвести до порушення якості суперництва вузлів дерева.

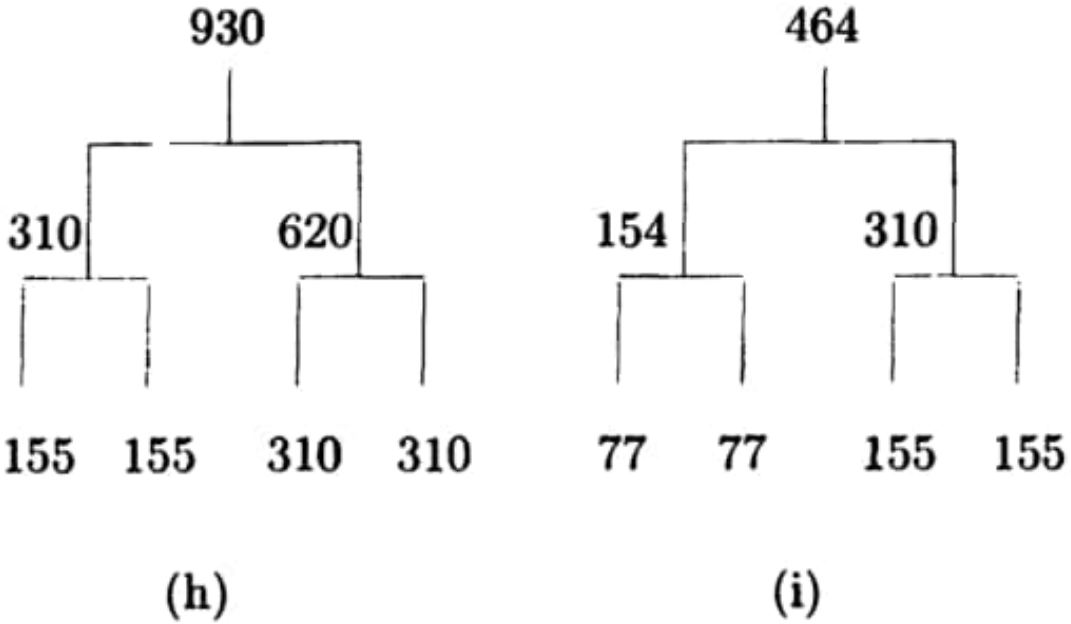
Лічильники частот символів зберігаються на дереві у вигляді чисел із фіксованою розрядністю. Поля розрядів можуть бути переповненими. Число без знака з 16 двійкових розрядів може накопичувати лічильник до числа $2^{16} - 1 = 65535$. Найпростіше вирішення цієї проблеми полягає в спостереженні за зростанням лічильника в корені дерева, і, коли він досягає максимального значення, слід зробити зміну масштабу всіх лічильників шляхом їхнього цілочисельного ділення на 2. Насправді це зазвичай робиться розподілом лічильників листя з наступним обчисленням лічильників всіх внутрішніх вузлів дерева. Кожен внутрішній вузол дорівнює сумі своїх нащадків. Однак при такому методі цілочисельного ділення знижується точність обчислень, що може призвести до порушення якості суперництва вузлів дерева.

Лічильники частот символів зберігаються на дереві у вигляді чисел із фіксованою розрядністю. Поля розрядів можуть бути переповненими. Число без знака з 16 двійкових розрядів може накопичувати лічильник до числа $2^{16} - 1 = 65535$. Найпростіше вирішення цієї проблеми полягає в спостереженні за зростанням лічильника в корені дерева, і, коли він досягає максимального значення, слід зробити зміну масштабу всіх лічильників шляхом їхнього цілочисельного ділення на 2. Насправді це зазвичай робиться розподілом лічильників листя з наступним обчисленням лічильників всіх внутрішніх вузлів дерева. Кожен внутрішній вузол дорівнює сумі своїх нащадків. Однак при такому методі цілочисельного ділення знижується точність обчислень, що може призвести до порушення якості суперництва вузлів дерева.

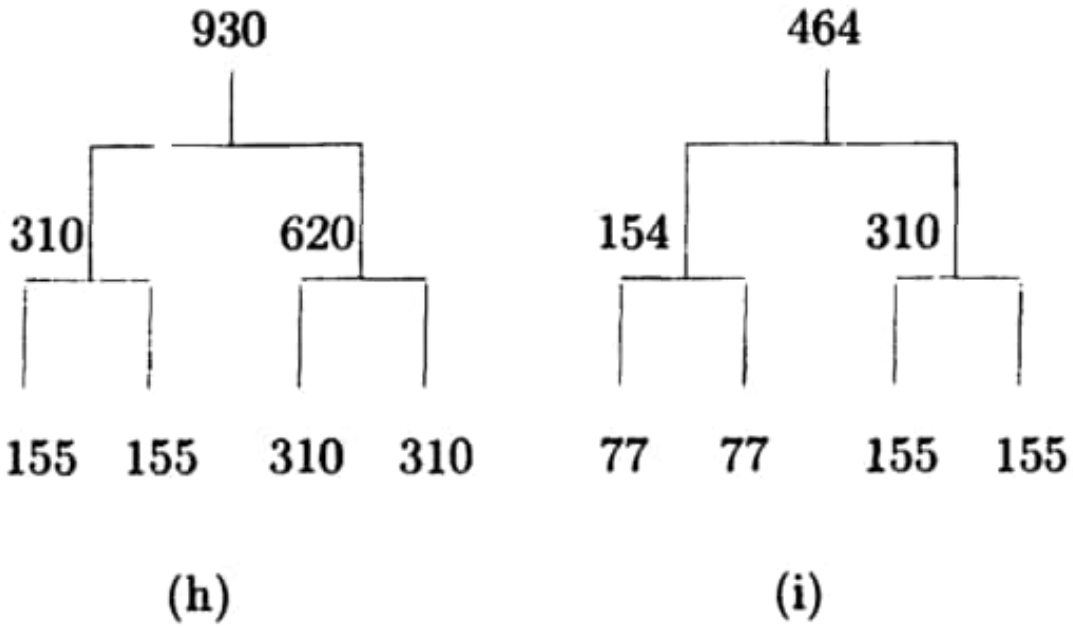
Лічильники частот символів зберігаються на дереві у вигляді чисел із фіксованою розрядністю. Поля розрядів можуть бути переповненими. Число без знака з 16 двійкових розрядів може накопичувати лічильник до числа $2^{16} - 1 = 65535$. Найпростіше вирішення цієї проблеми полягає в спостереженні за зростанням лічильника в корені дерева, і, коли він досягає максимального значення, слід зробити зміну масштабу всіх лічильників шляхом їхнього цілочисельного ділення на 2. Насправді це зазвичай робиться розподілом лічильників листя з наступним обчисленням лічильників всіх внутрішніх вузлів дерева. Кожен внутрішній вузол дорівнює сумі своїх нащадків. Однак при такому методі цілочисельного ділення знижується точність обчислень, що може призвести до порушення якості суперництва вузлів дерева.



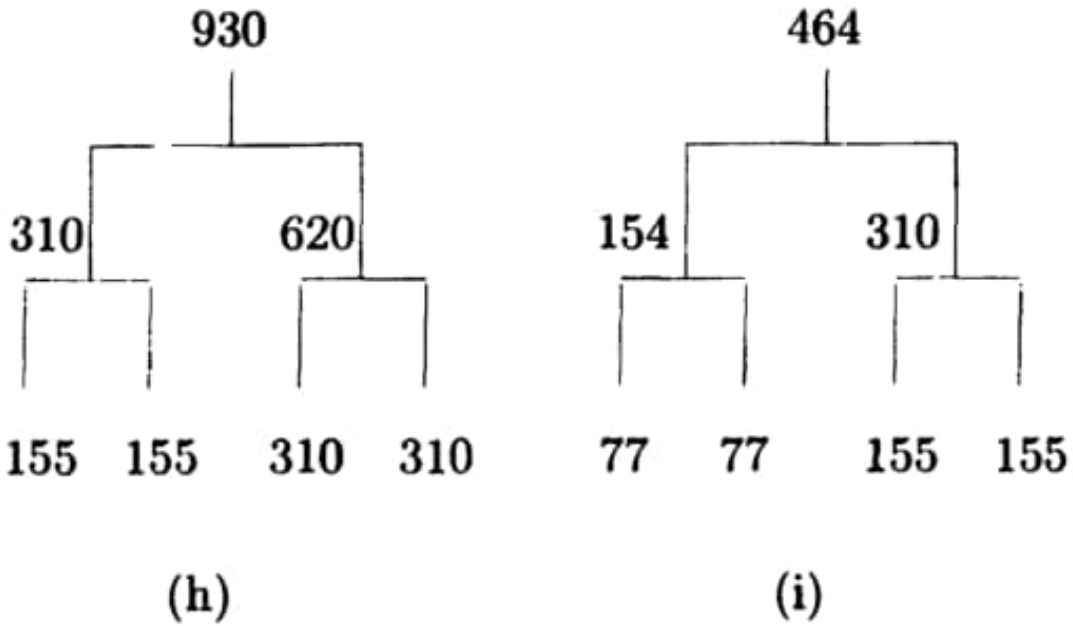
Найпростіший приклад наведено на рис. (h). Після уполовинювання лічильників листя, три внутрішні вузли перераховані, як показано на рис. (i). Отримане дерево, однак, не задовольняє властивості Гаффмана, оскільки лічильники перестали бути впорядкованими. Тому після кожної зміни масштабу потрібна перебудова дерева, яка, на щастя, робитиметься не надто часто. Тому комп'ютерні програми загального призначення, що використовують метод стиснення Гаффмана, повинні мати багаторозрядні лічильники, щоб їхнє переповнення траплялося не надто часто. Лічильники розрядності в 4 байти переповнюватимуться при значенні рівному $2^{32} - 1 \approx 4.3 \cdot 10^9$.



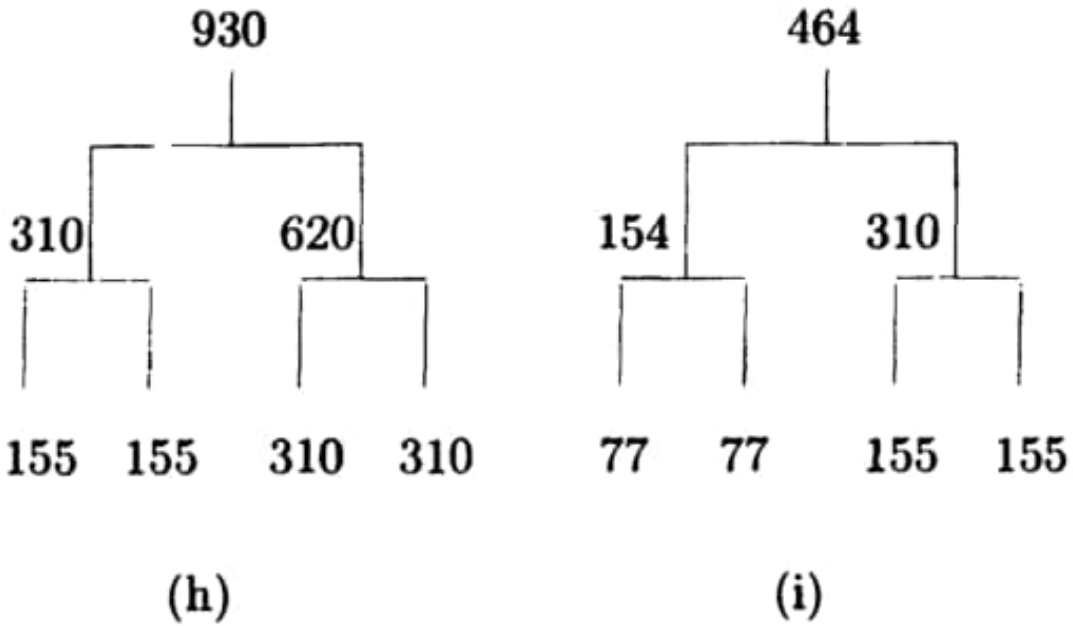
Найпростіший приклад наведено на рис. (h). Після уполовинювання лічильників листя, три внутрішні вузли перераховані, як показано на рис. (i). Отримане дерево, однак, не задовольняє властивості Гаффмана, оскільки лічильники перестали бути впорядкованими. Тому після кожної зміни масштабу потрібна перебудова дерева, яка, на щастя, робитиметься не надто часто. Тому комп'ютерні програми загального призначення, що використовують метод стиснення Гаффмана, повинні мати багаторозрядні лічильники, щоб їхнє переповнення траплялося не надто часто. Лічильники розрядності в 4 байти переповнюватимуться при значенні рівному $2^{32} - 1 \approx 4.3 \cdot 10^9$.



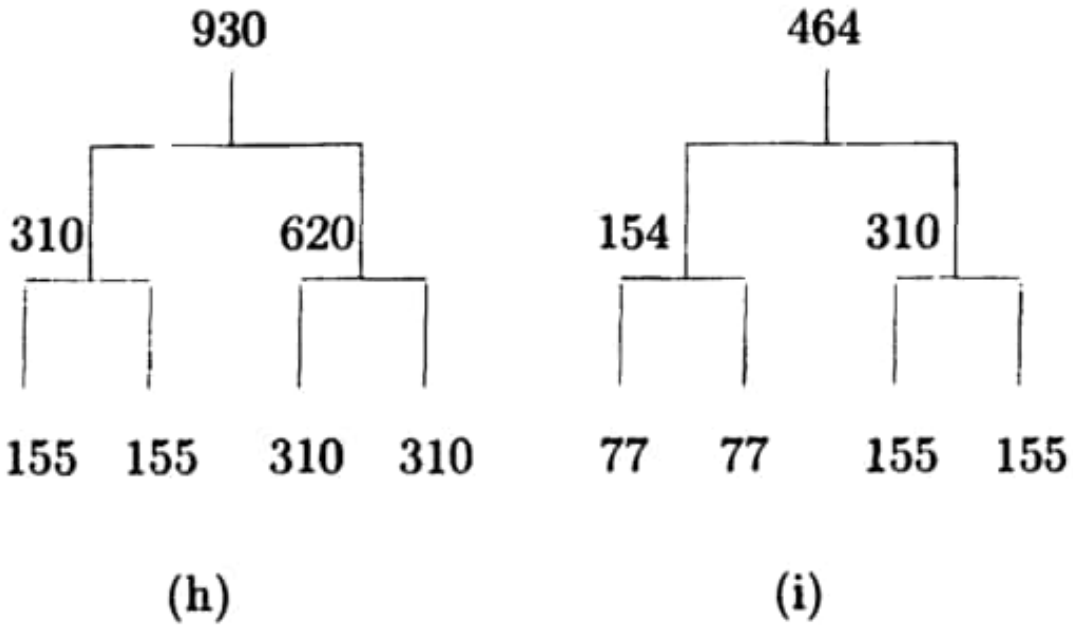
Найпростіший приклад наведено на рис. (h). Після уполовинювання лічильників листя, три внутрішні вузли перераховані, як показано на рис. (i). Отримане дерево, однак, не задовольняє властивості Гаффмана, оскільки лічильники перестали бути впорядкованими. Тому після кожної зміни масштабу потрібна перебудова дерева, яка, на щастя, робитиметься не надто часто. Тому комп'ютерні програми загального призначення, що використовують метод стиснення Гаффмана, повинні мати багаторозрядні лічильники, щоб їхнє переповнення траплялося не надто часто. Лічильники розрядності в 4 байти переповнюватимуться при значенні рівному $2^{32} - 1 \approx 4.3 \cdot 10^9$.



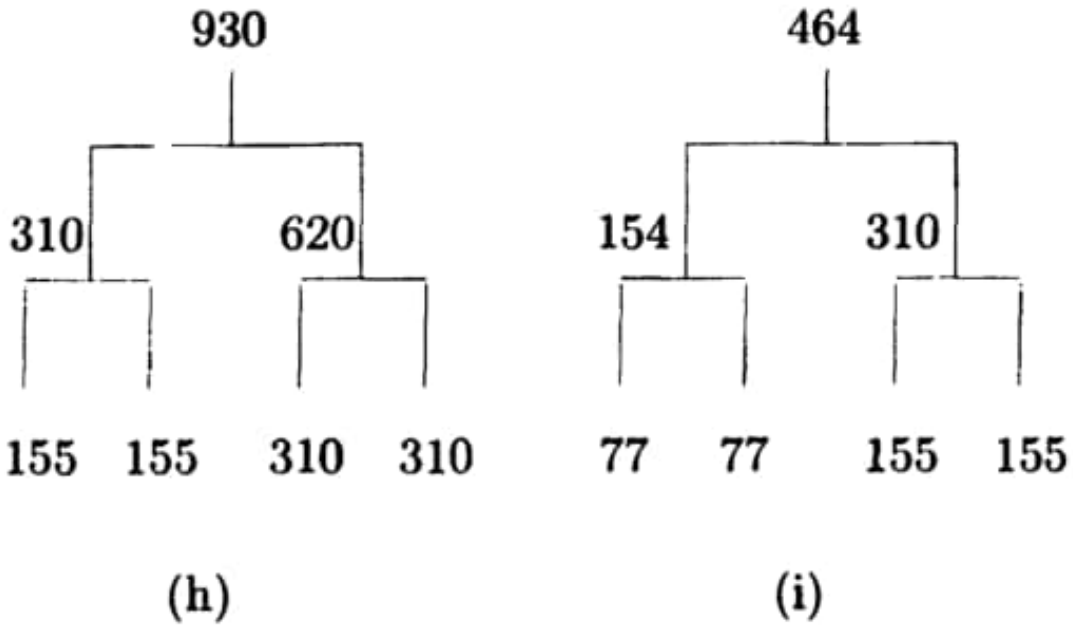
Найпростіший приклад наведено на рис. (h). Після уполовинювання лічильників листя, три внутрішні вузли перераховані, як показано на рис. (i). Отримане дерево, однак, не задовольняє властивості Гаффмана, оскільки лічильники перестали бути впорядкованими. Тому після кожної зміни масштабу потрібна перебудова дерева, яка, на щастя, робитиметься не надто часто. Тому комп'ютерні програми загального призначення, що використовують метод стиснення Гаффмана, повинні мати багаторозрядні лічильники, щоб їхнє переповнення траплялося не надто часто. Лічильники розрядності в 4 байти переповнюватимуться при значенні рівному $2^{32} - 1 \approx 4.3 \cdot 10^9$.



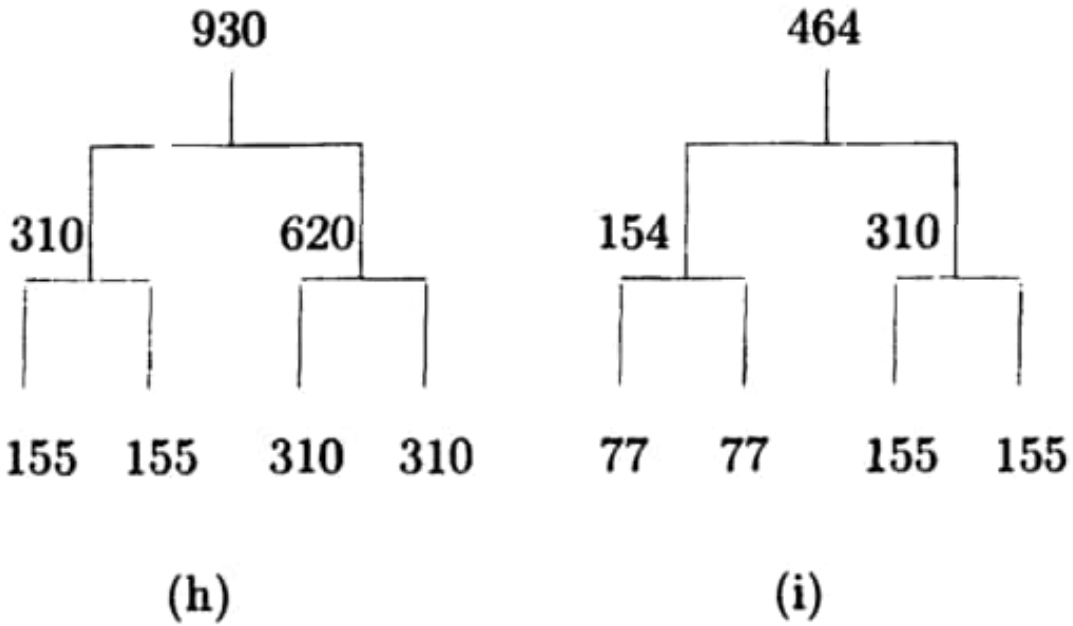
Найпростіший приклад наведено на рис. (h). Після уполовинювання лічильників листя, три внутрішні вузли перераховані, як показано на рис. (i). Отримане дерево, однак, не задовольняє властивості Гаффмана, оскільки лічильники перестали бути впорядкованими. Тому після кожної зміни масштабу потрібна перебудова дерева, яка, на щастя, робитиметься не надто часто. Тому комп'ютерні програми загального призначення, що використовують метод стиснення Гаффмана, повинні мати багаторозрядні лічильники, щоб їхнє переповнення траплялося не надто часто. Лічильники розрядності в 4 байти переповнюватимуться при значенні рівному $2^{32} - 1 \approx 4.3 \cdot 10^9$.



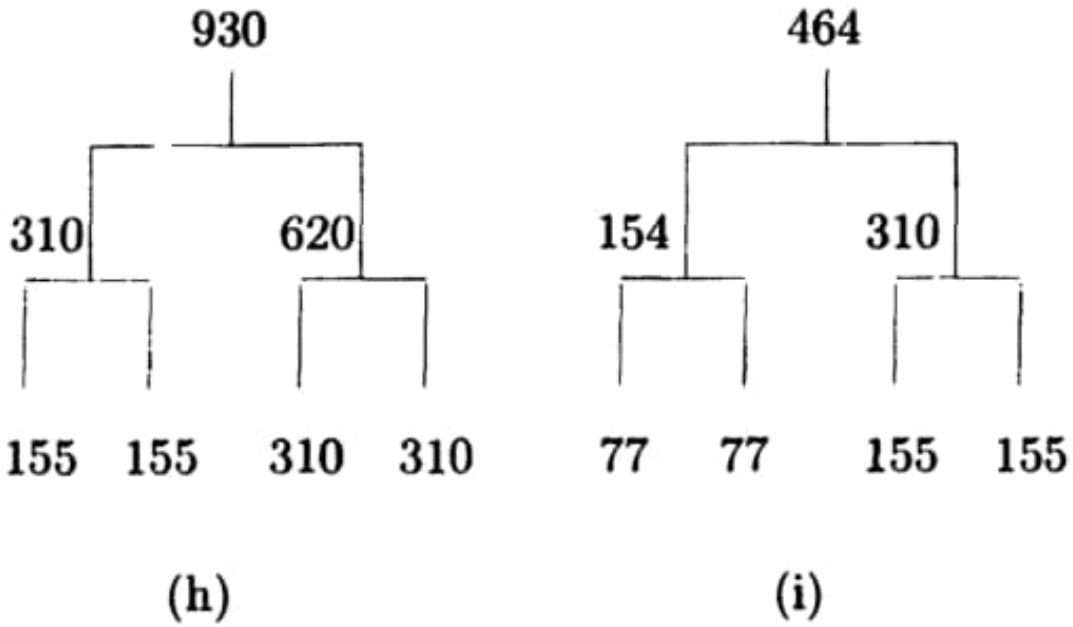
Найпростіший приклад наведено на рис. (h). Після уполовинювання лічильників листя, три внутрішні вузли перераховані, як показано на рис. (i). Отримане дерево, однак, не задовольняє властивості Гаффмана, оскільки лічильники перестали бути впорядкованими. Тому після кожної зміни масштабу потрібна перебудова дерева, яка, на щастя, робитиметься не надто часто. Тому комп'ютерні програми загального призначення, що використовують метод стиснення Гаффмана, повинні мати багаторозрядні лічильники, щоб їхнє переповнення траплялося не надто часто. Лічильники розрядності в 4 байти переповнюватимуться при значенні рівному $2^{32} - 1 \approx 4.3 \cdot 10^9$.



Найпростіший приклад наведено на рис. (h). Після уполовинювання лічильників листя, три внутрішні вузли перераховані, як показано на рис. (i). Отримане дерево, однак, не задовольняє властивості Гаффмана, оскільки лічильники перестали бути впорядкованими. Тому після кожної зміни масштабу потрібна перебудова дерева, яка, на щастя, робитиметься не надто часто. Тому комп'ютерні програми загального призначення, що використовують метод стиснення Гаффмана, повинні мати багаторозрядні лічильники, щоб їхнє переповнення траплялося не надто часто. Лічильники розрядності в 4 байти переповнюватимуться при значенні рівному $2^{32} - 1 \approx 4.3 \cdot 10^9$.

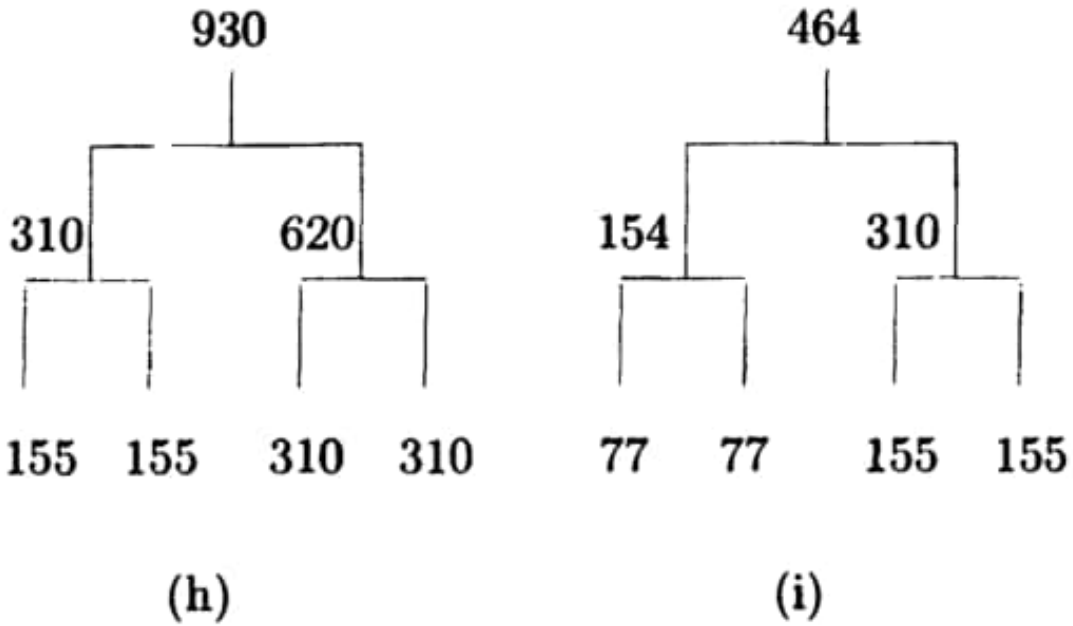


Найпростіший приклад наведено на рис. (h). Після уполовинювання лічильників листя, три внутрішні вузли перераховані, як показано на рис. (i). Отримане дерево, однак, не задовольняє властивості Гаффмана, оскільки лічильники перестали бути впорядкованими. Тому після кожної зміни масштабу потрібна перебудова дерева, яка, на щастя, робитиметься не надто часто. Тому комп'ютерні програми загального призначення, що використовують метод стиснення Гаффмана, повинні мати багаторозрядні лічильники, щоб їхнє переповнення траплялося не надто часто. Лічильники розрядності в 4 байти переповнюватимуться при значенні рівному $2^{32} - 1 \approx 4.3 \cdot 10^9$.



Найпростіший приклад наведено на рис. (h). Після уполовинювання лічильників листя, три внутрішні вузли перераховані, як показано на рис. (i). Отримане дерево, однак, не задовольняє властивості Гаффмана, оскільки лічильники перестали бути впорядкованими. Тому після кожної зміни масштабу потрібна перебудова дерева, яка, на щастя, робитиметься не надто часто. Тому комп'ютерні програми загального призначення, що використовують метод стиснення Гаффмана, повинні мати багаторозрядні лічильники, щоб їхнє переповнення траплялося не надто часто.

Лічильники розрядності в 4 байти переповнюватимуться при значенні рівному $2^{32} - 1 \approx 4.3 \cdot 10^9$.



Найпростіший приклад наведено на рис. (h). Після уполовинювання лічильників листя, три внутрішні вузли перераховані, як показано на рис. (i). Отримане дерево, однак, не задовольняє властивості Гаффмана, оскільки лічильники перестали бути впорядкованими. Тому після кожної зміни масштабу потрібна перебудова дерева, яка, на щастя, робитиметься не надто часто. Тому комп'ютерні програми загального призначення, що використовують метод стиснення Гаффмана, повинні мати багаторозрядні лічильники, щоб їхнє переповнення траплялося не надто часто. Лічильники розрядності в 4 байти переповнюватимуться при значенні рівному $2^{32} - 1 \approx 4.3 \cdot 10^9$.

Варто зазначити, що після зміни масштабу лічильників, нові символи, які надійшли для стиснення, будуть впливати на лічильники сильніше, ніж стиснені раніше до зміни масштабу (приблизно з вагою 2). Це не критично, оскільки з досвіду відомо, що ймовірність появи символу більше залежить від безпосередньо попередніх символів, ніж від тих, що надійшли у віддаленому минулому.

Варто зазначити, що після зміни масштабу лічильників, нові символи, які надійшли для стиснення, будуть впливати на лічильники сильніше, ніж стиснені раніше до зміни масштабу (приблизно з вагою 2). Це не критично, оскільки з досвіду відомо, що ймовірність появи символу більше залежить від безпосередньо попередніх символів, ніж від тих, що надійшли у віддаленому минулому.

Варто зазначити, що після зміни масштабу лічильників, нові символи, які надійшли для стиснення, будуть впливати на лічильники сильніше, ніж стиснені раніше до зміни масштабу (приблизно з вагою 2). Це не критично, оскільки з досвіду відомо, що ймовірність появи символу більше залежить від безпосередньо попередніх символів, ніж від тих, що надійшли у віддаленому минулому.

Варто зазначити, що після зміни масштабу лічильників, нові символи, які надійшли для стиснення, будуть впливати на лічильники сильніше, ніж стиснені раніше до зміни масштабу (приблизно з вагою 2). Це не критично, оскільки з досвіду відомо, що ймовірність появи символу більше залежить від безпосередньо попередніх символів, ніж від тих, що надійшли у віддаленому минулому.

Варто зазначити, що після зміни масштабу лічильників, нові символи, які надійшли для стиснення, будуть впливати на лічильники сильніше, ніж стиснені раніше до зміни масштабу (приблизно з вагою 2). Це не критично, оскільки з досвіду відомо, що ймовірність появи символу більше залежить від безпосередньо попередніх символів, ніж від тих, що надійшли у віддаленому минулому.

Варто зазначити, що після зміни масштабу лічильників, нові символи, які надійшли для стиснення, будуть впливати на лічильники сильніше, ніж стиснені раніше до зміни масштабу (приблизно з вагою 2). Це не критично, оскільки з досвіду відомо, що ймовірність появи символу більше залежить від безпосередньо попередніх символів, ніж від тих, що надійшли у віддаленому минулому.

Варто зазначити, що після зміни масштабу лічильників, нові символи, які надійшли для стиснення, будуть впливати на лічильники сильніше, ніж стиснені раніше до зміни масштабу (приблизно з вагою 2). Це не критично, оскільки з досвіду відомо, що ймовірність появи символу більше залежить від безпосередньо попередніх символів, ніж від тих, що надійшли у віддаленому минулому.

Варто зазначити, що після зміни масштабу лічильників, нові символи, які надійшли для стиснення, будуть впливати на лічильники сильніше, ніж стиснені раніше до зміни масштабу (приблизно з вагою 2). Це не критично, оскільки з досвіду відомо, що ймовірність появи символу більше залежить від безпосередньо попередніх символів, ніж від тих, що надійшли у віддаленому минулому.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Більш серйозну проблему може створити кодове переповнення. Це станеться, якщо на дерево надійде занадто багато символів і воно стане надто високим. Самі коди не зберігаються на дереві, тому що вони змінюються весь час, і компресор повинен обчислювати код символу X щоразу при його появі. Ось що при цьому відбувається:

1. Кодер повинен знайти символ X на дереві. Дерево слід реалізувати як масиву структур, які з вузлів. Пошук у цьому масиві буде лінійним.
2. Якщо символ X не знайдено, то виробляється код esc, за яким слідує нестиснений код символу. Потім символ X додається до дерева.
3. Якщо символ X знайдено, то компресор рухається від вузла X назад до кореня, вибудовуючи його код біт за бітом. На кожному кроці, рухаючись вліво від нащадка до батька, він додає до коду "1", а рухаючись вправо, додає "0" (або навпаки, але це має бути чітко визначено, оскільки декодер робитиме те саме). Цю послідовність бітів треба десь зберігати, оскільки вона записуватиметься у зворотному порядку. Коли дерево стане високим, то коди теж подовжаться. Якщо вони накопичуються у вигляді 16-розрядного цілого, то коди довші 16 біт викличуть збій програми.

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

Приклад

Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

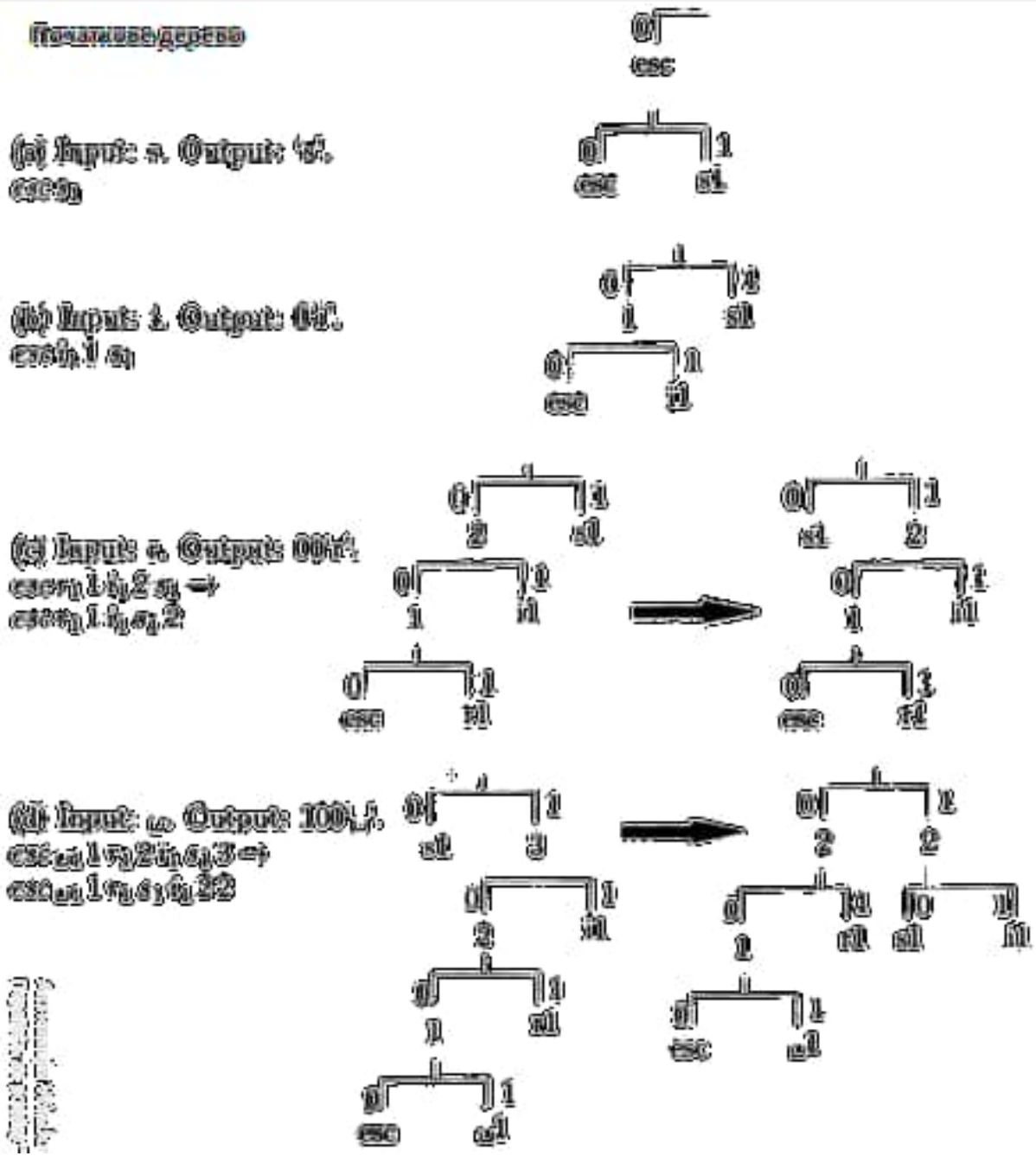
Правильний розв'язок може полягати у накопиченні бітів коду у зв'язаному списку, до якого можна додавати нові вузли. Тоді обмеженням буде лише обсяг доступної пам'яті. Це загальний розв'язок, але повільний. Інший розв'язок полягає у накопиченні коду в довгій цілій змінній (наприклад з 50 розрядів). При цьому слід документувати програму обмеженням 50 біт для максимальної довжини кодів.

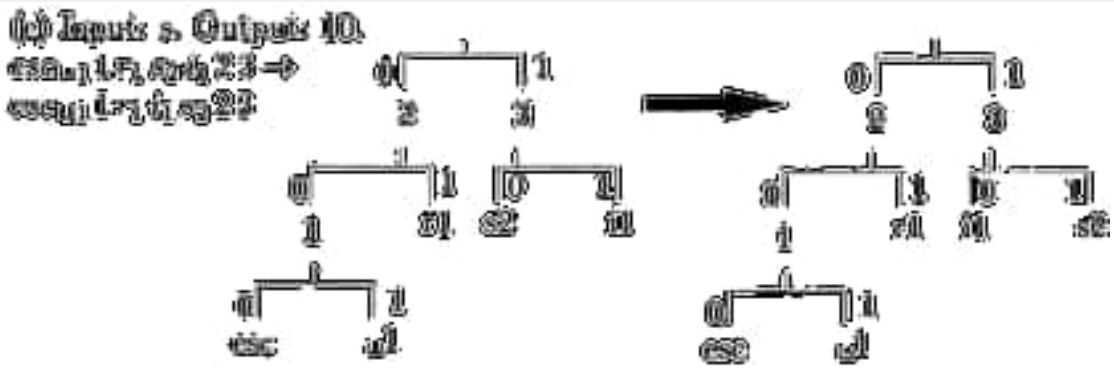
На щастя, ця проблема не впливає на процес декодування. Декодер читає стислий код біт за бітом і використовує кожен біт, щоб йти крок за кроком вліво або вправо вниз по дереву, доки він не досягне листка із символом. Якщо листком служить код esc, то декодер прочитає стиснутий символ зі стисненого файлу (і додасть цей символ на дерево). В іншому випадку нестиснений символ буде на цьому листку дерева.

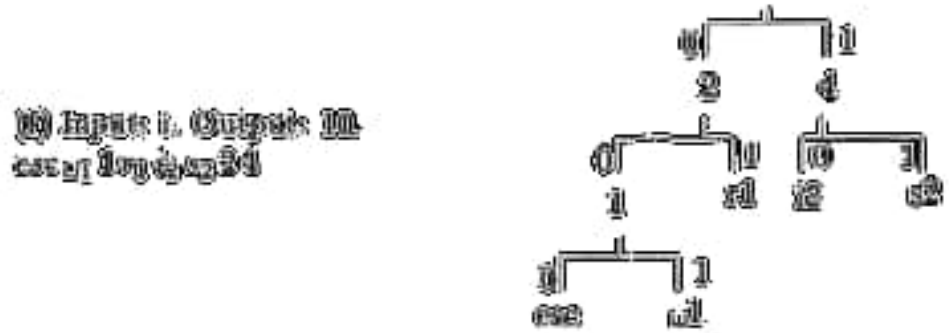
Приклад

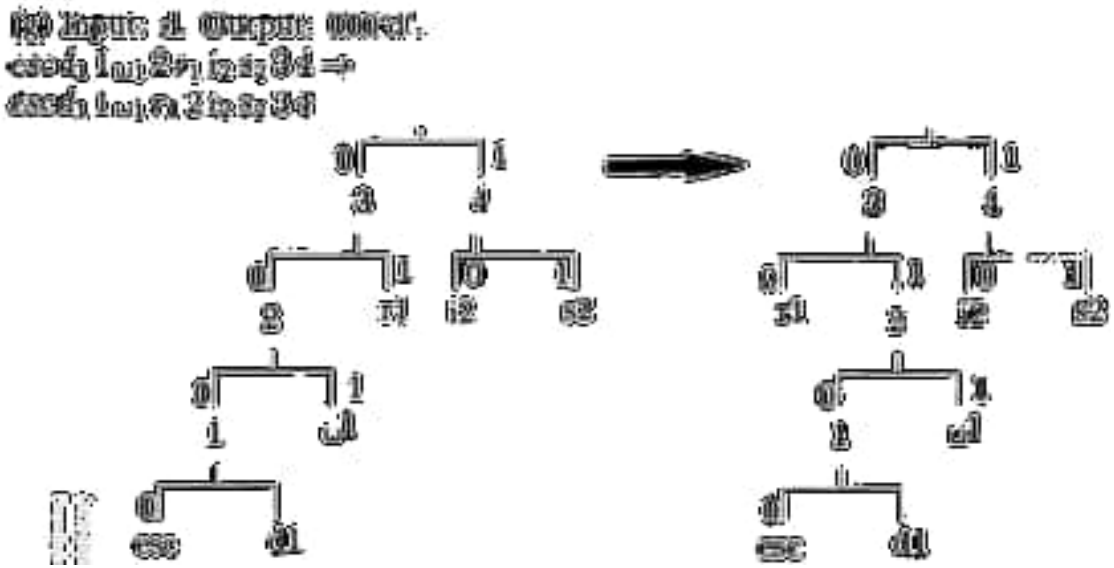
Застосуємо адаптивне кодування Хаффмана до рядка "sir_sid_is". Для кожного вхідного символу знайдено код на виході, дерево після додавання цього символу, дерево після модифікації (якщо необхідно), а також пройдені вузли зліва направо знизу вгору.

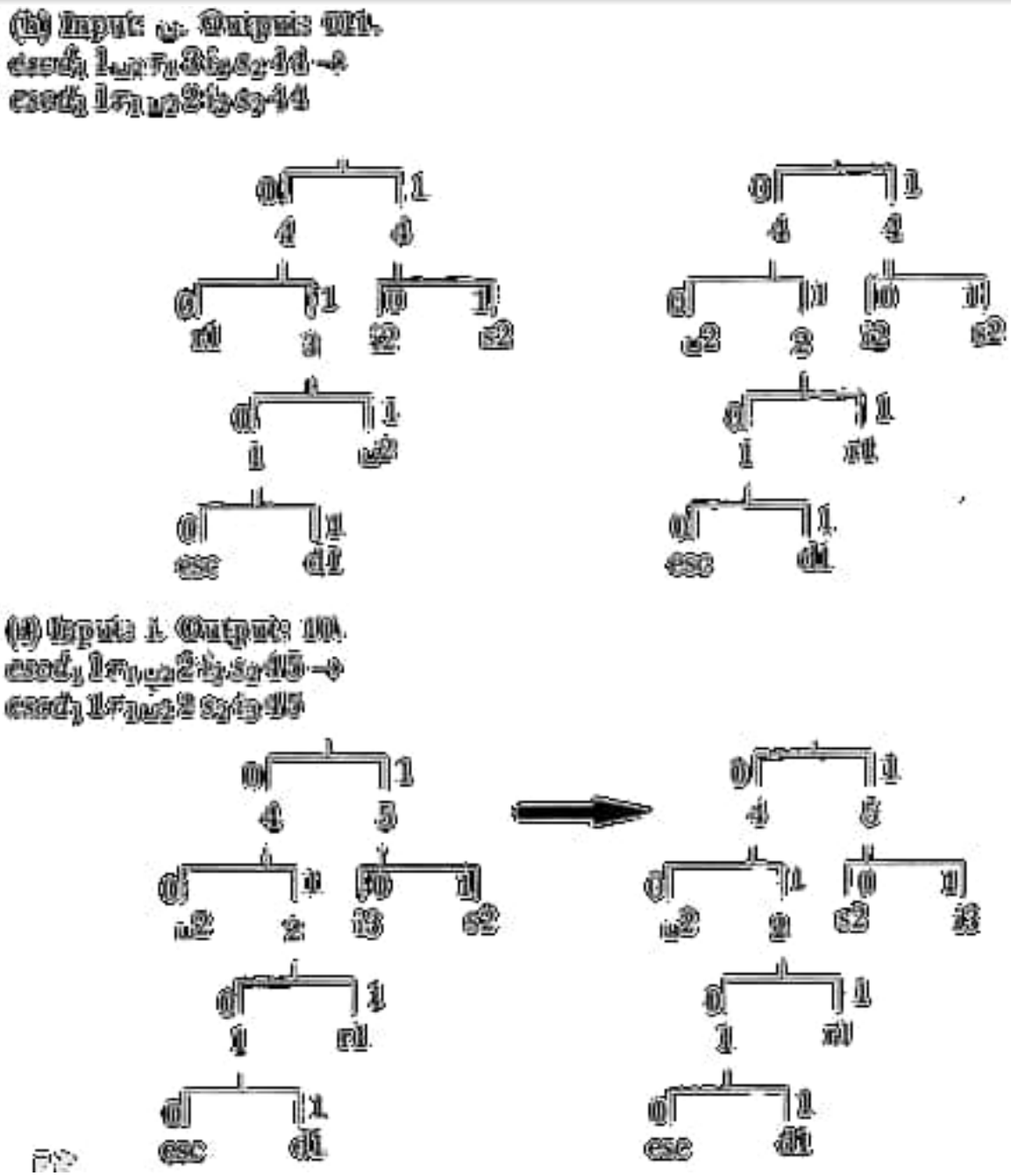
Наступні рисунки зображають початкове дерево і як воно змінювалося за 11 кроків від (а) до (к).

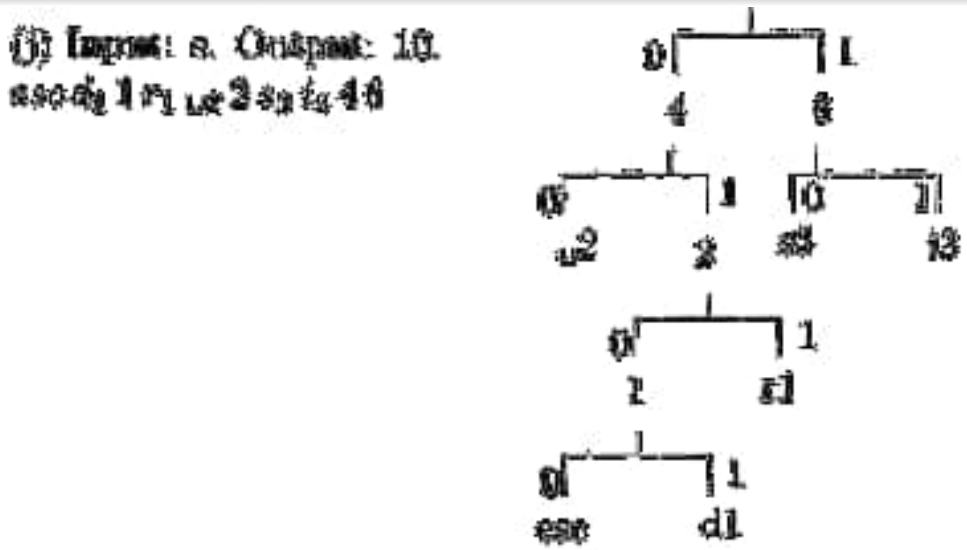


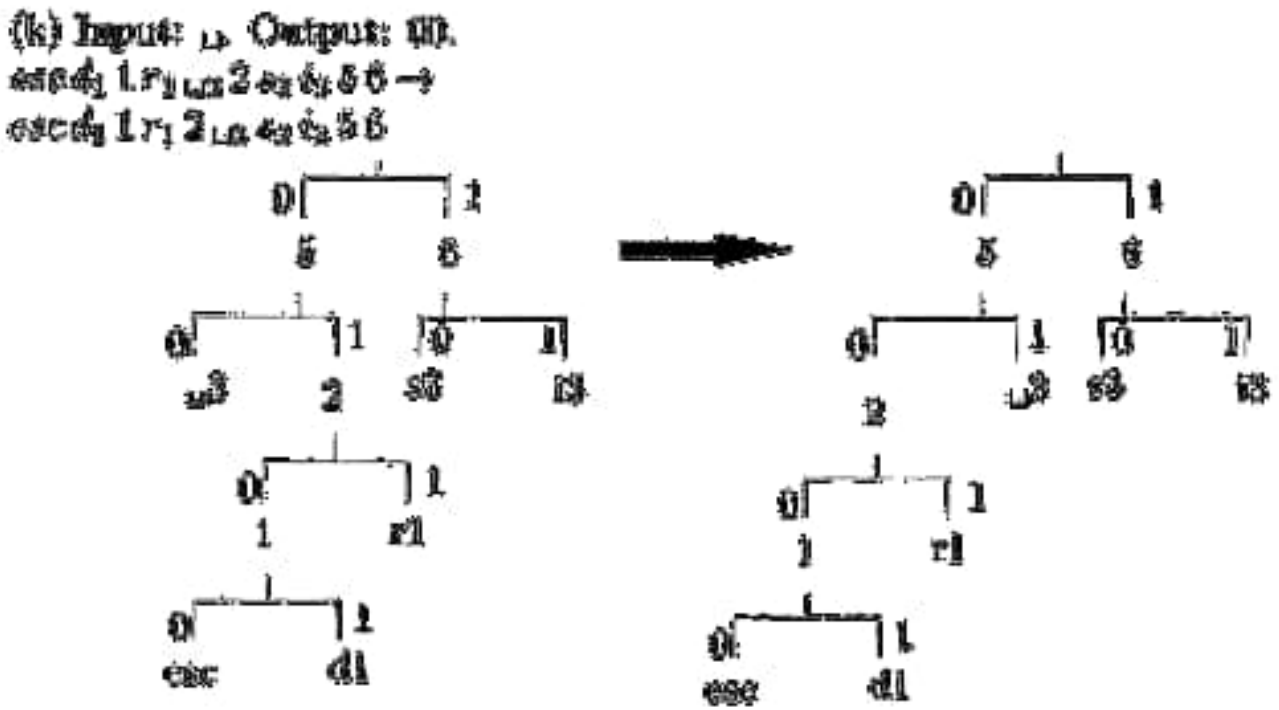


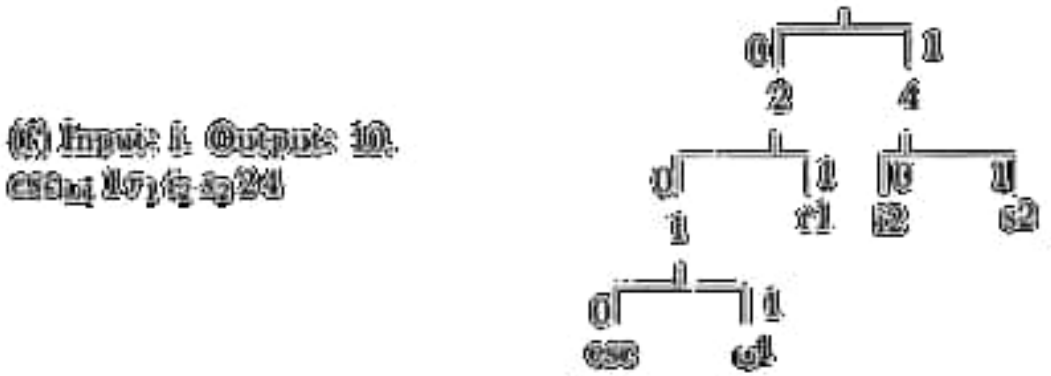


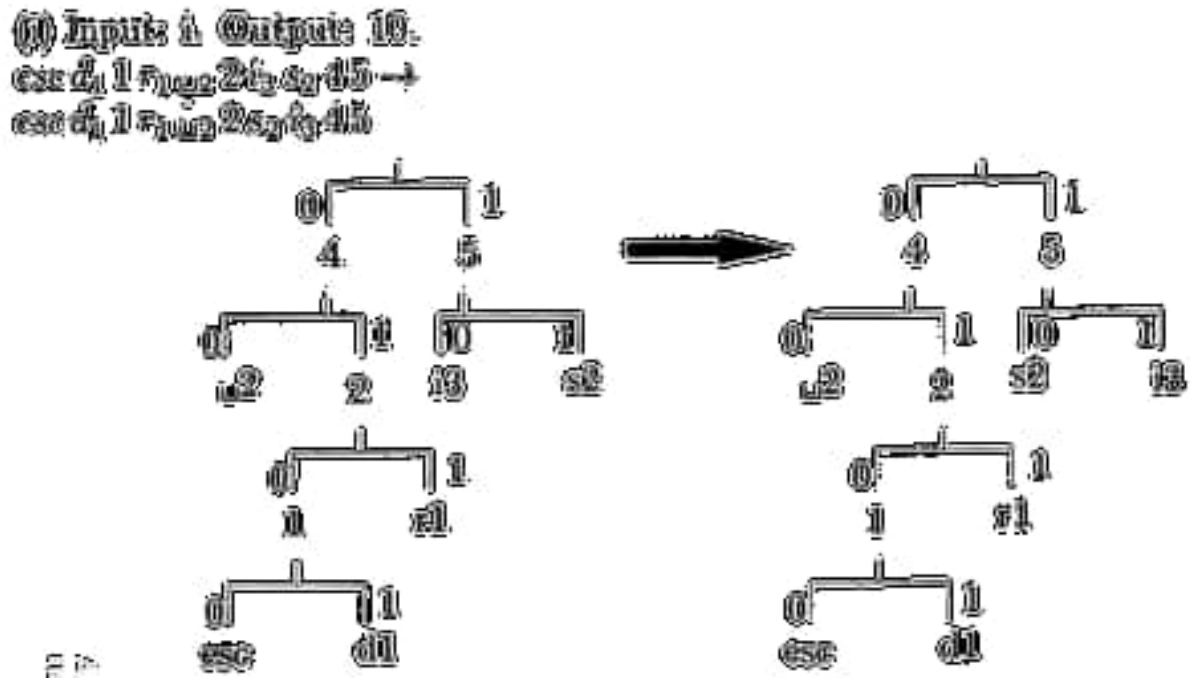




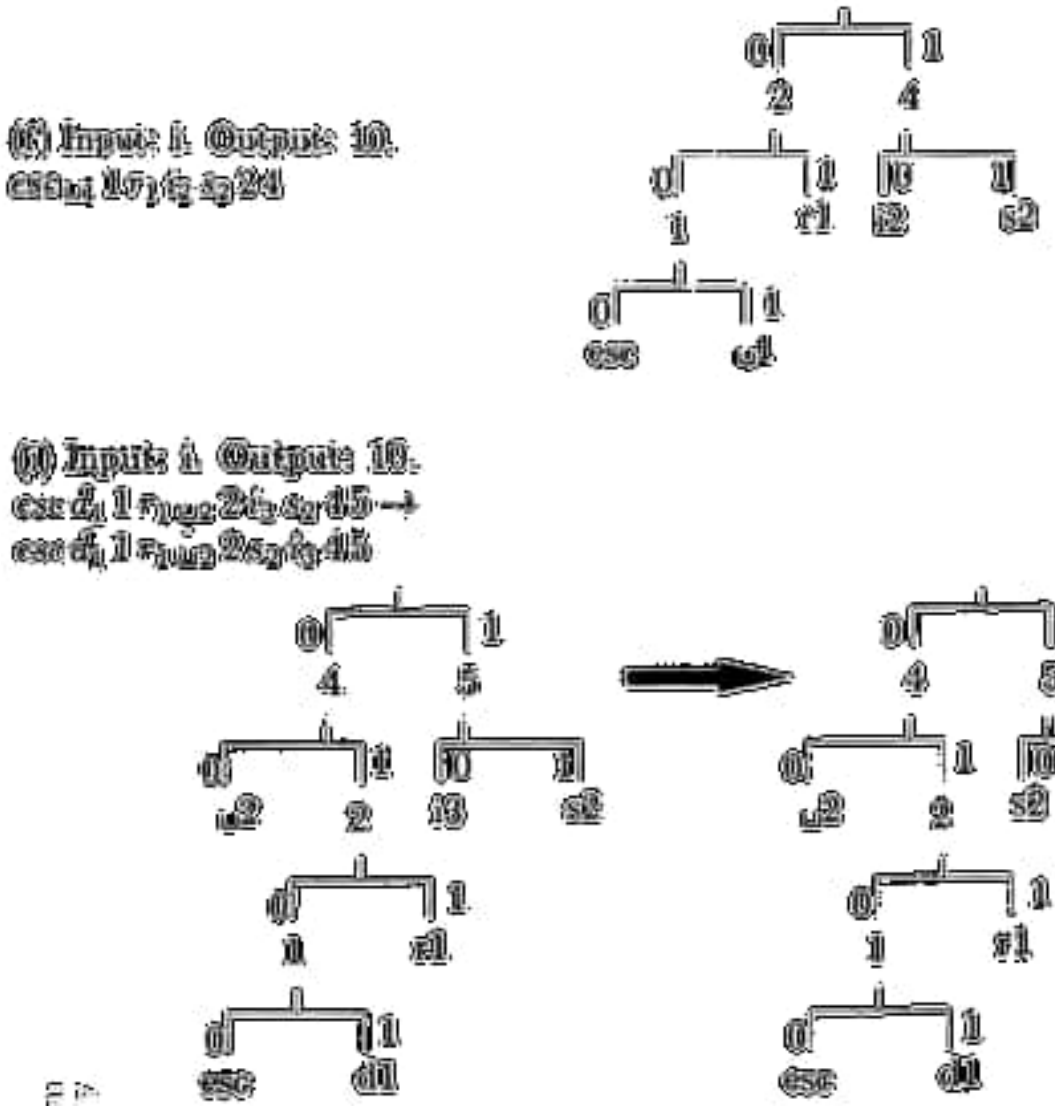




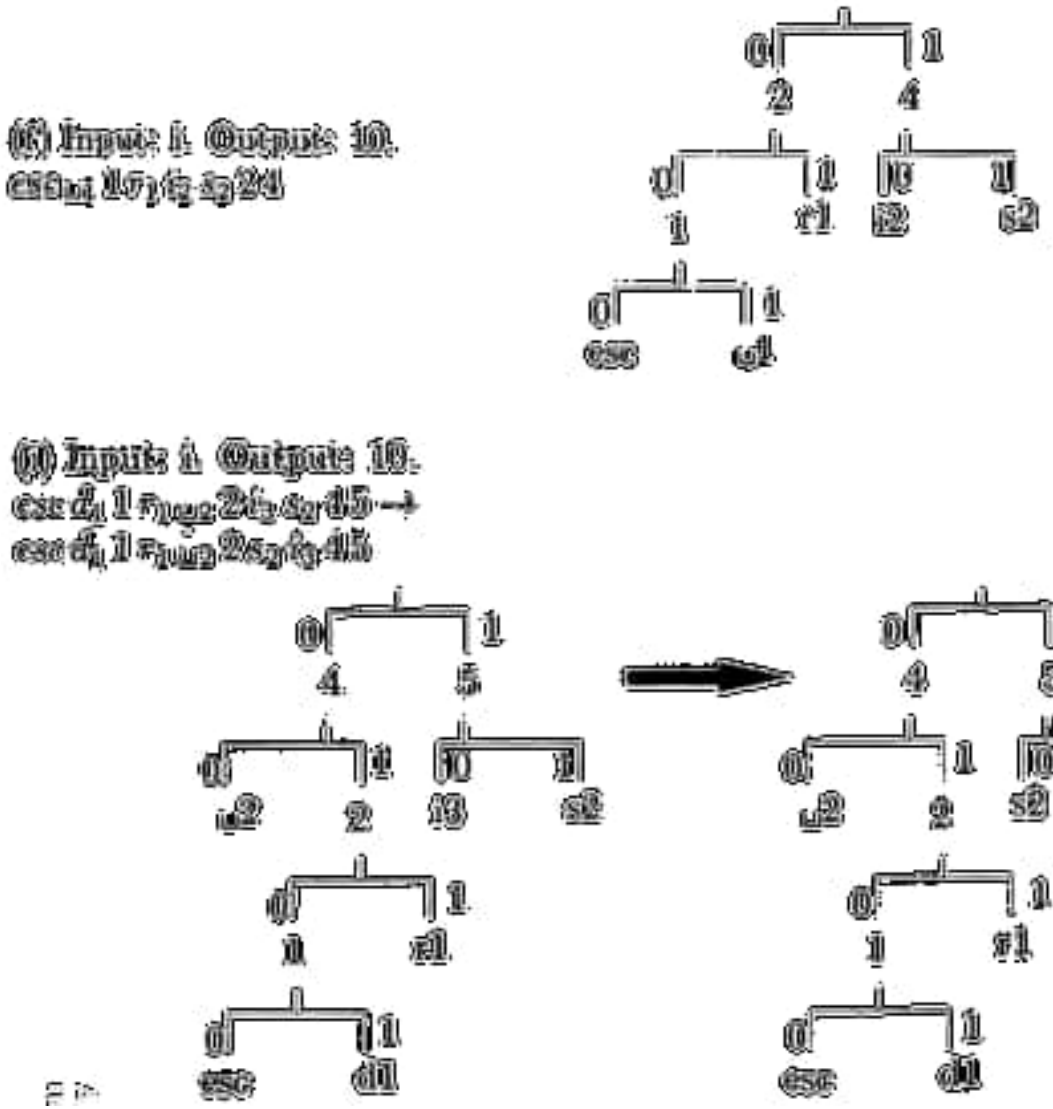




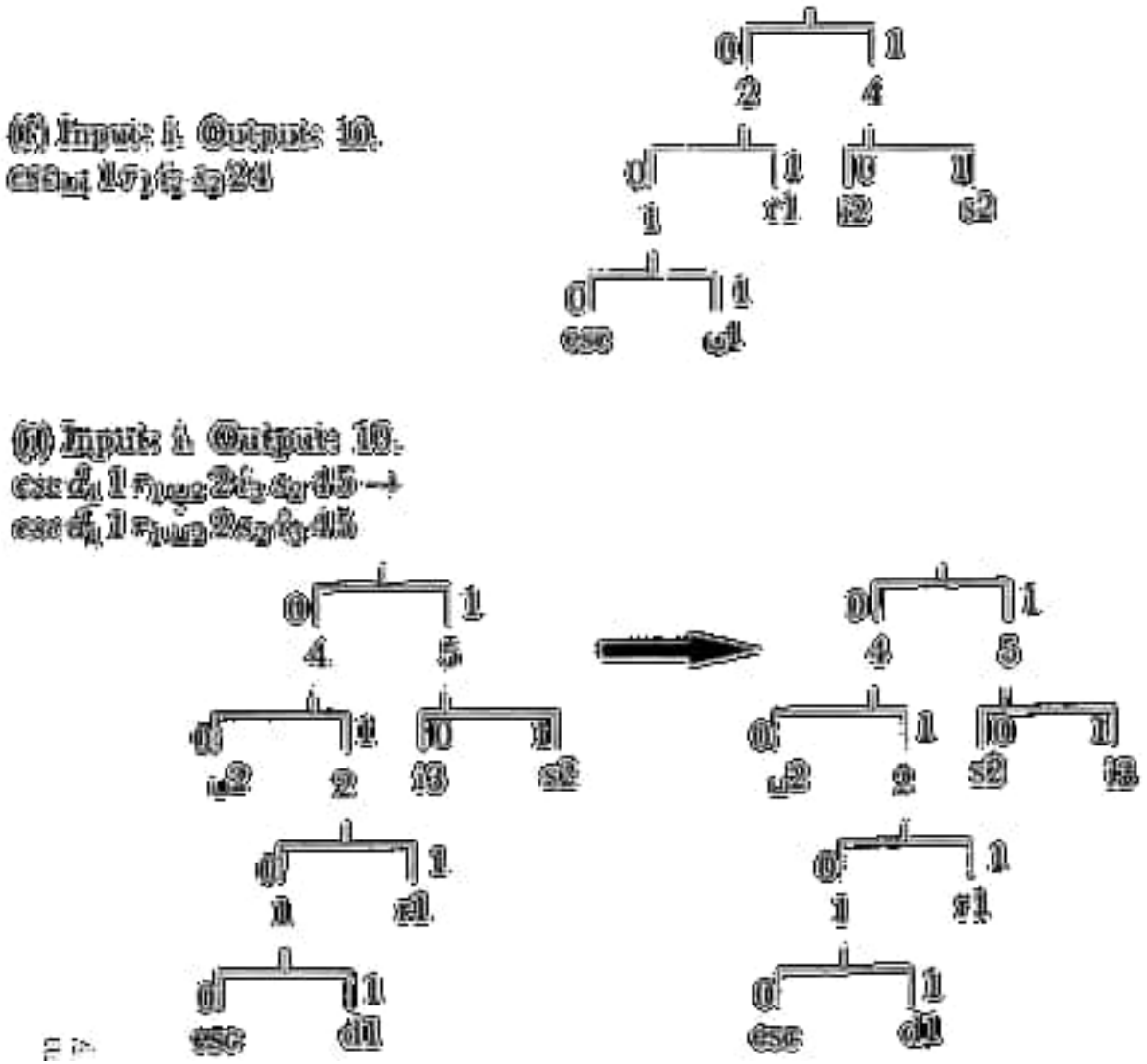
Зверніть увагу на те, як символ esc отримує постійно різні коди, і як різні символи переміщаються по дереву, змінюючи свої коди. Код 10, наприклад, кодує символ “i” на кроках (f) та (i),



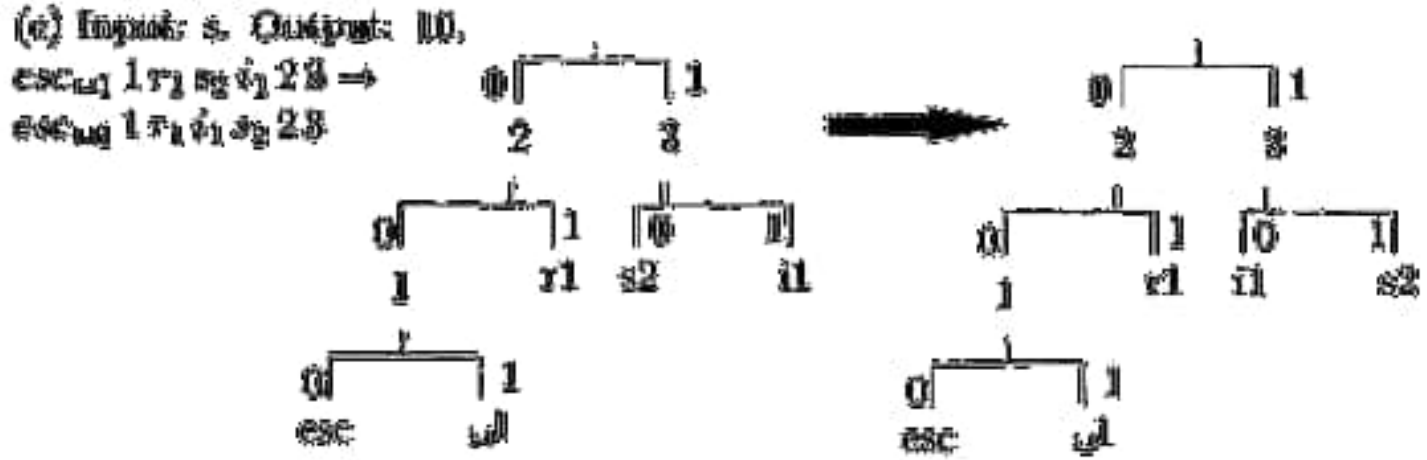
Зверніть увагу на те, як символ esc отримує постійно різні коди, і як різні символи переміщаються по дереву, змінюючи свої коди. Код 10, наприклад, кодує символ “i” на кроках (f) та (i),

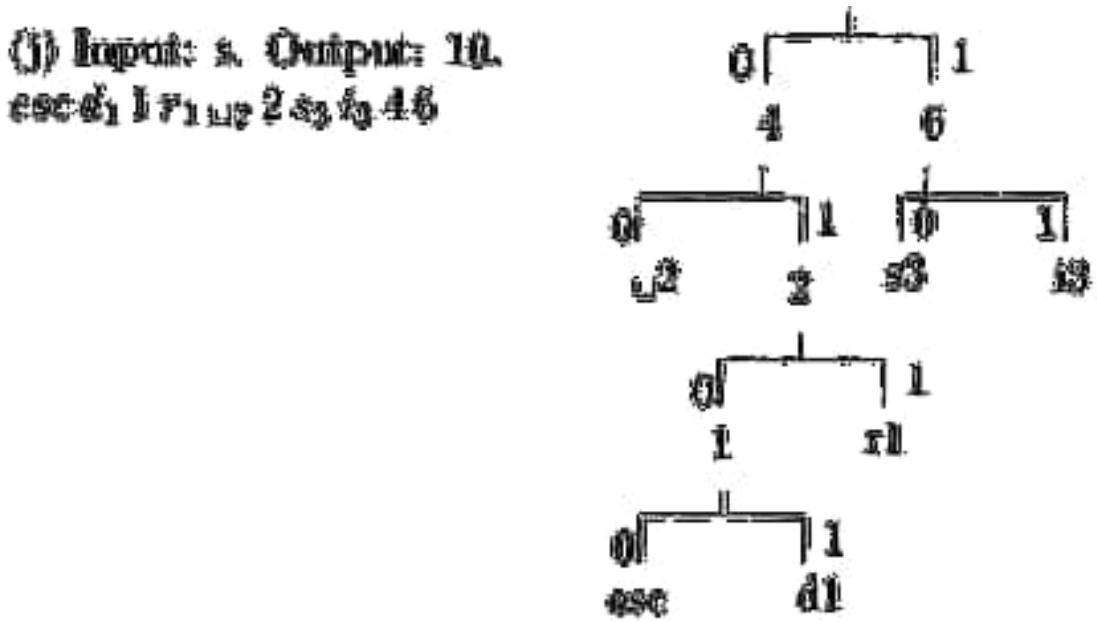


Зверніть увагу на те, як символ esc отримує постійно різні коди, і як різні символи переміщаються по дереву, змінюючи свої коди. Код 10, наприклад, кодує символ “i” на кроках (f) та (i),

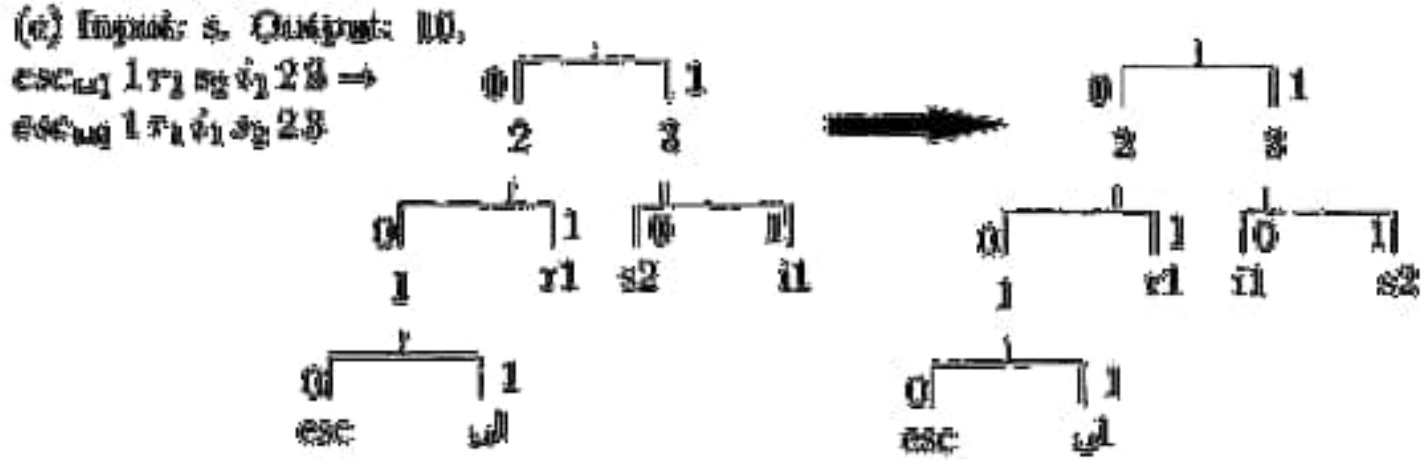


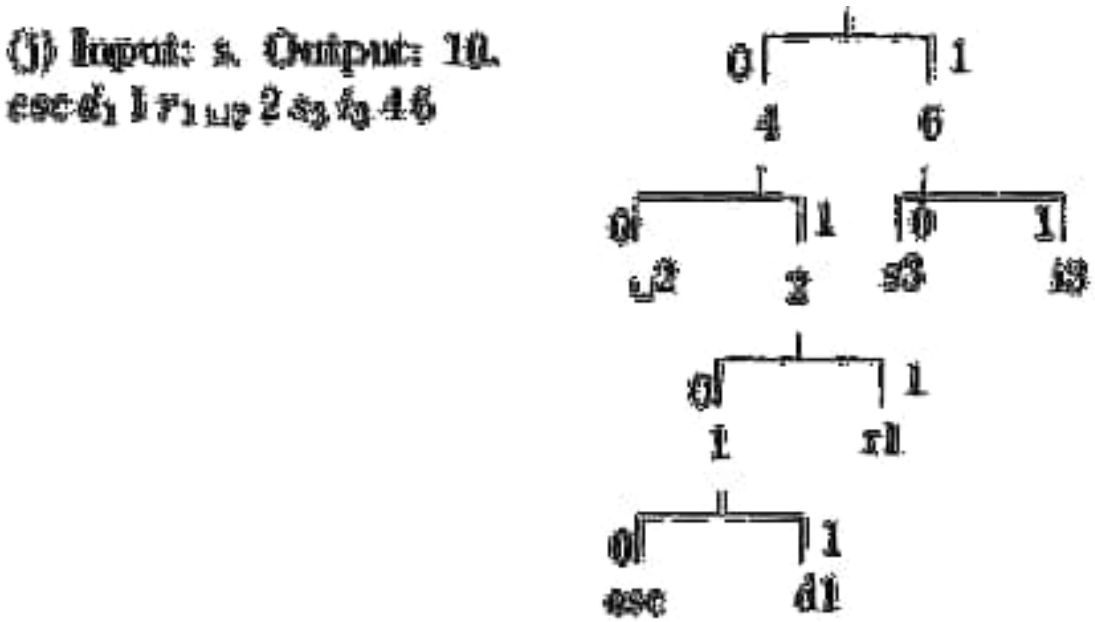
Зверніть увагу на те, як символ esc отримує постійно різні коди, і як різні символи переміщаються по дереву, змінюючи свої коди. Код 10, наприклад, кодує символ “i” на кроках (f) та (i),



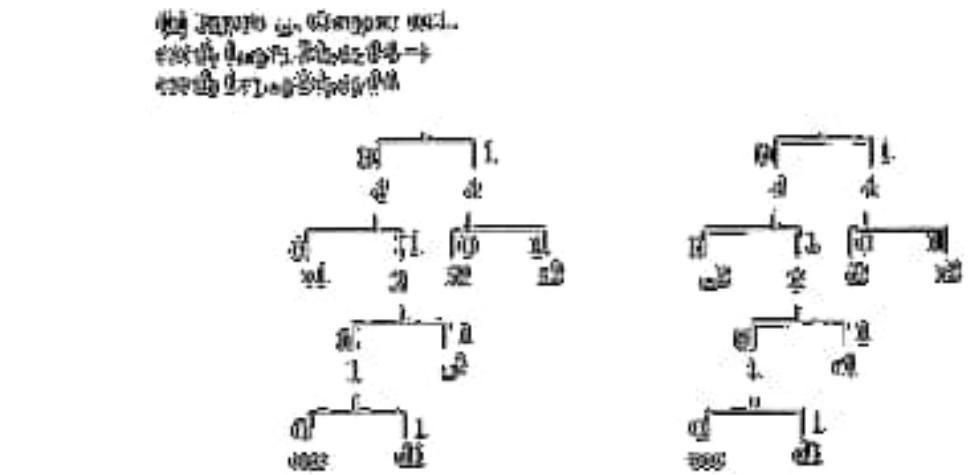


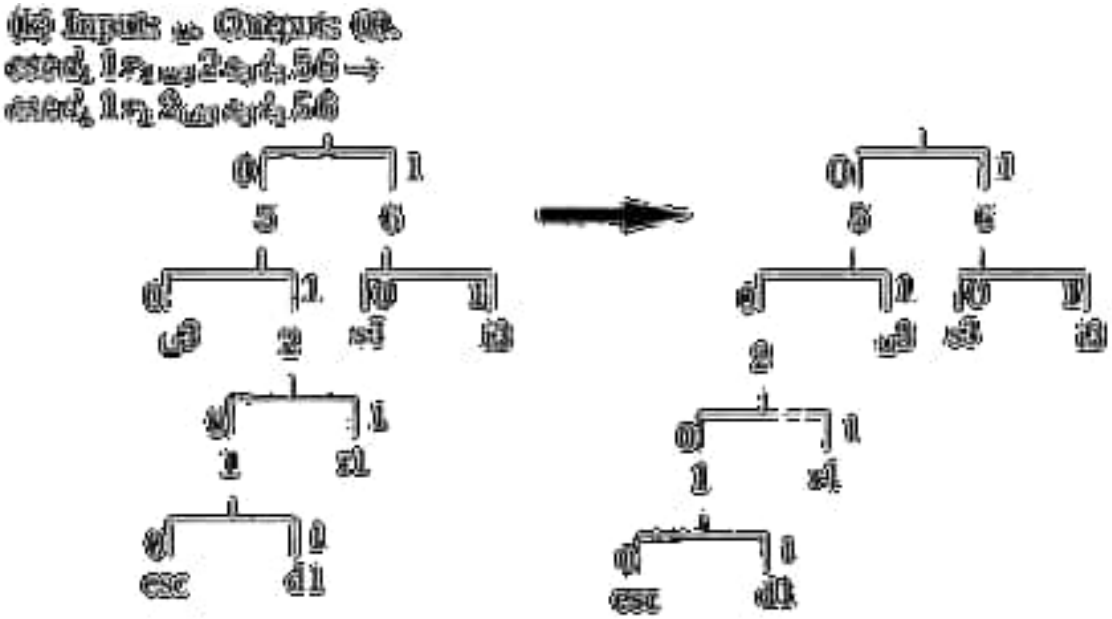
цей ж код надається символу “s” на кроках (e) та (j).



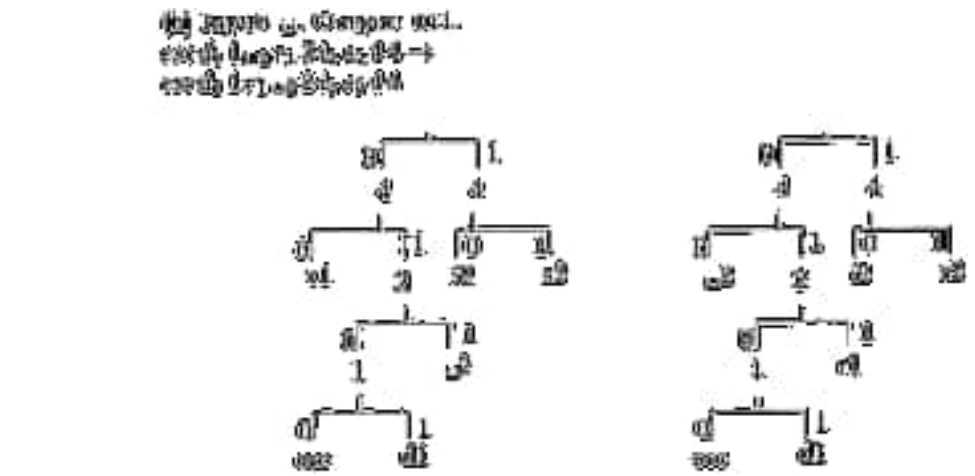


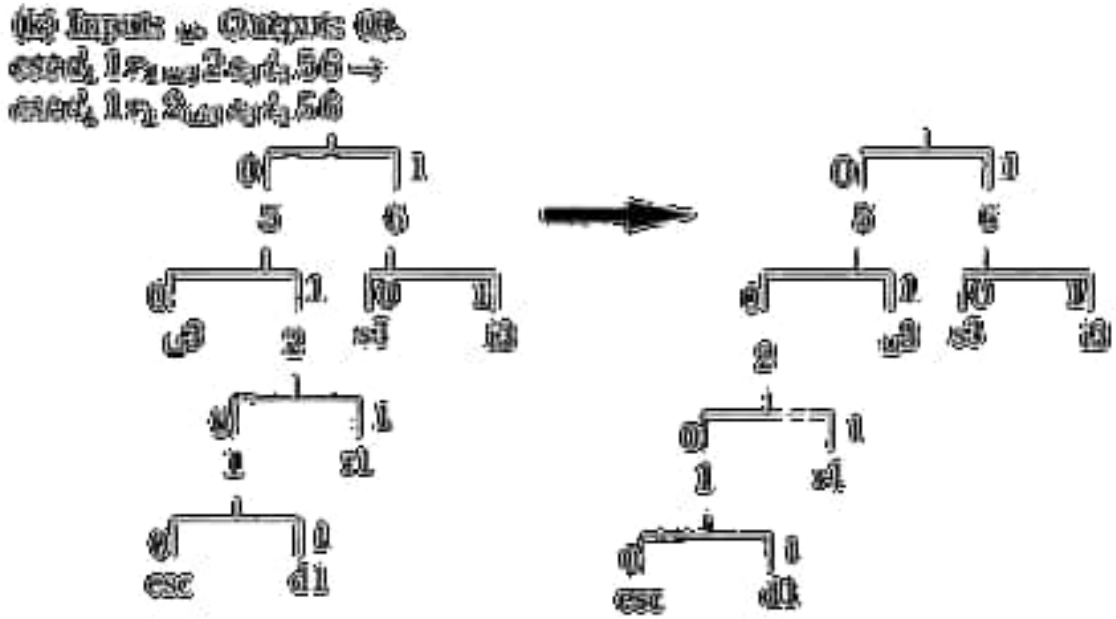
цей ж код надається символу “s” на кроках (e) та (j).



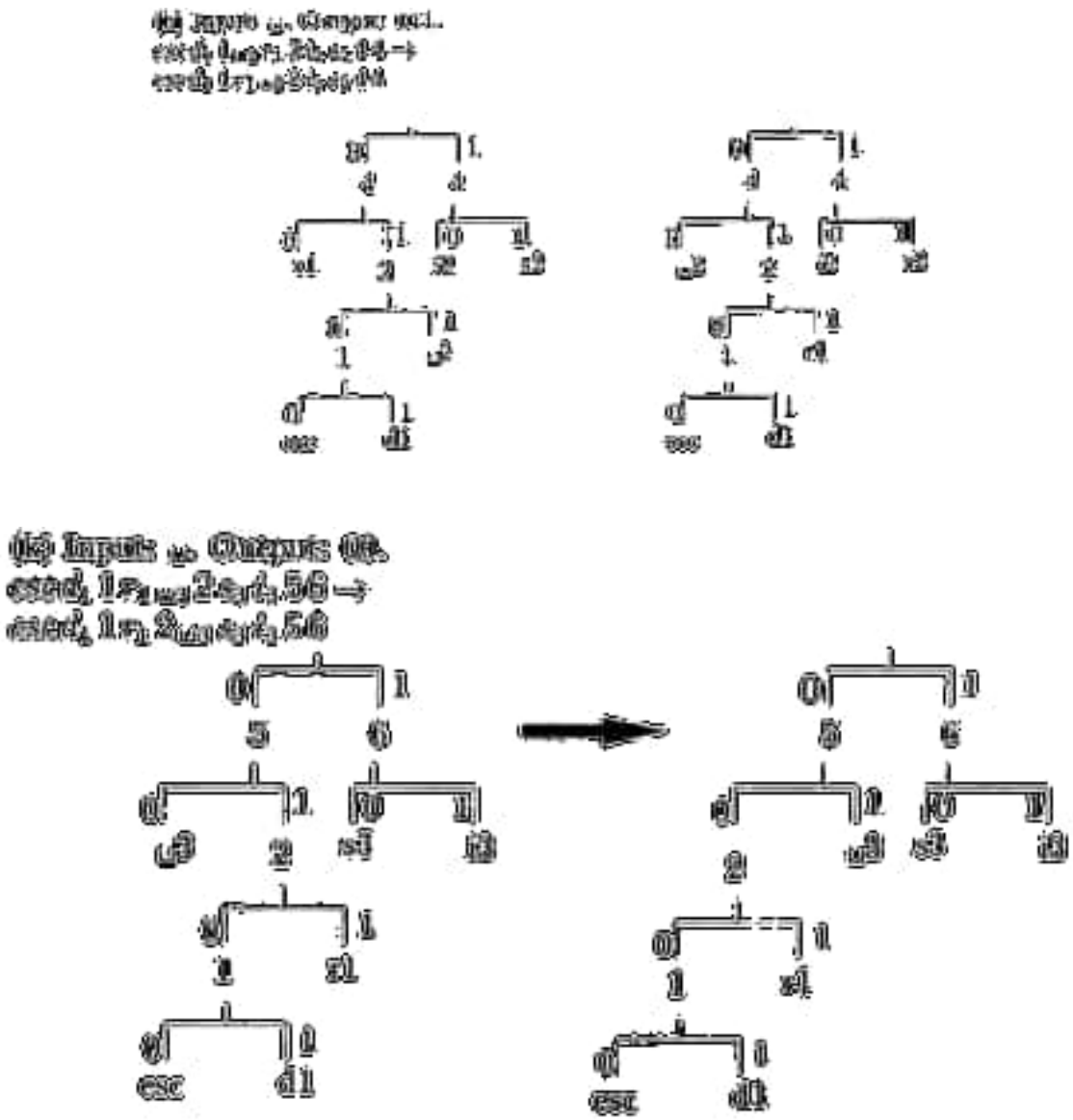


Порожній символ має код 011 на кроці (h), але він має код 00 на кроці (к).





Порожній символ має код 011 на кроці (h), але він має код 00 на кроці (к).



Порожній символ має код 011 на кроці (h), але він має код 00 на кроці (к).

На виході кодера буде рядок

$$\text{"s0i00r100\_l0l0000d011101000".}$$

Загальна кількість бітів дорівнює

$$5 \cdot 8 + 22 = 62.$$

Коефіцієнт стиснення дорівнює

$$\frac{62}{88} \approx 0.7.$$

На виході кодера буде рядок

$$\text{"s0i00r100\_l0l0000d011101000".}$$

Загальна кількість бітів дорівнює

$$5 \cdot 8 + 22 = 62.$$

Коефіцієнт стиснення дорівнює

$$\frac{62}{88} \approx 0.7.$$

На виході кодера буде рядок

$$\text{"s0i00r100\_l0l0000d011101000".}$$

Загальна кількість бітів дорівнює

$$5 \cdot 8 + 22 = 62.$$

Коефіцієнт стиснення дорівнює

$$\frac{62}{88} \approx 0.7.$$

На виході кодера буде рядок

$$\text{"s0i00r100\_l0l0000d011101000".}$$

Загальна кількість бітів дорівнює

$$5 \cdot 8 + 22 = 62.$$

Коефіцієнт стиснення дорівнює

$$\frac{62}{88} \approx 0.7.$$

На виході кодера буде рядок

$$\text{"s0i00r100\_l0l0000d011101000".}$$

Загальна кількість бітів дорівнює

$$5 \cdot 8 + 22 = 62.$$

Коефіцієнт стиснення дорівнює

$$\frac{62}{88} \approx 0.7.$$

На виході кодера буде рядок

$$\text{"s0i00r100\_l0l0000d011101000".}$$

Загальна кількість бітів дорівнює

$$5 \cdot 8 + 22 = 62.$$

Коефіцієнт стиснення дорівнює

$$\frac{62}{88} \approx 0.7.$$

На виході кодера буде рядок

$$\text{"s0i00r100\_l0l0000d011101000".}$$

Загальна кількість бітів дорівнює

$$5 \cdot 8 + 22 = 62.$$

Коефіцієнт стиснення дорівнює

$$\frac{62}{88} \approx 0.7.$$

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Цей варіант адаптивного кодування Гаффмана дуже простий, але менш ефективний. Його ідея полягає у побудові множини з n кодів змінної довжини на основі рівних ймовірностей та випадковому присвоєнні цих кодів n символам. Після чого зміна кодів робиться "на льоту" в залежності від зчитування та стиснення символів. Метод не надто продуктивний, оскільки коди не ґрунтуються на реальних ймовірностях символів вхідного файлу. Однак його простіше реалізувати, і він працюватиме швидше за описаний вище алгоритм, оскільки переставляє рядки таблиці швидше, ніж перший алгоритм перебудовує дерево при зміні частот символів.

Основна структура даних — це таблиця розміру $n \times 3$, в якій три стовпці зберігають, відповідно, імена n символів, частоти символів та їхні коди. Таблиця весь час впорядкована по другому стовпцю. Коли лічильники частот у другому стовпці змінюються, то рядки переставляються, але переміщаються лише перший і другий стовпці. Коди у третьому стовпці не змінюються.

Симв.	Лічильник	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Лічильник	Код
a_2	1	0
a_1	0	10
a_3	0	110
a_4	0	111

(b)

Симв.	Лічильник	Код
a_2	1	0
a_4	1	10
a_3	0	110
a_1	0	111

(c)

Симв.	Лічильник	Код
a_4	2	0
a_2	1	10
a_3	0	110
a_1	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3a_4a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

Симв.	Ім.	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Ім.	Код
a_2	1	0
a_4	0	10
a_3	0	110
a_1	0	111

(b)

Симв.	Ім.	Код
a_2	1	0
a_4	1	10
a_3	0	110
a_1	0	111

(c)

Симв.	Ім.	Код
a_4	2	0
a_2	1	10
a_3	0	110
a_1	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3 a_4 a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

Симв.	Ім.	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Ім.	Код
a_2	1	0
a_1	0	10
a_3	0	110
a_4	0	111

(b)

Симв.	Ім.	Код
a_2	1	0
a_4	1	10
a_3	0	110
a_1	0	111

(c)

Симв.	Ім.	Код
a_4	2	0
a_2	1	10
a_3	0	110
a_1	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3 a_4 a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

Симв.	Ім.	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Ім.	Код
a_2	1	0
a_1	0	10
a_3	0	110
a_4	0	111

(b)

Симв.	Ім.	Код
a_2	1	0
a_4	1	10
a_3	0	110
a_1	0	111

(c)

Симв.	Ім.	Код
a_4	2	0
a_2	1	10
a_3	0	110
a_1	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3 a_4 a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

Симв.	Лічильник	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Лічильник	Код
a_2	1	0
a_4	0	10
a_3	0	110
a_1	0	111

(b)

Симв.	Лічильник	Код
a_2	1	0
a_4	1	10
a_3	0	110
a_1	0	111

(c)

Симв.	Лічильник	Код
a_4	2	0
a_2	1	10
a_3	0	110
a_1	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3a_4a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

Симв.	Ім.	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Ім.	Код
a_2	1	0
a_1	0	10
a_3	0	110
a_4	0	111

(b)

Симв.	Ім.	Код
a_2	1	0
a_4	1	10
a_3	0	110
a_1	0	111

(c)

Симв.	Ім.	Код
a_4	2	0
a_2	1	10
a_3	0	110
a_1	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3 a_4 a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

Симв.	Ім.	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Ім.	Код
a_2	1	0
a_1	0	10
a_3	0	110
a_4	0	111

(b)

Симв.	Ім.	Код
a_2	1	0
a_4	1	10
a_3	0	110
a_1	0	111

(c)

Симв.	Ім.	Код
a_4	2	0
a_2	1	10
a_3	0	110
a_1	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3a_4a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

Симв.	Лічильник	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Лічильник	Код
a_2	1	0
a_1	0	10
a_3	0	110
a_4	0	111

(b)

Симв.	Лічильник	Код
a_2	1	0
a_4	1	10
a_3	0	110
a_1	0	111

(c)

Симв.	Лічильник	Код
a_4	2	0
a_2	1	10
a_3	0	110
a_1	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3a_4a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

Симв.	Ім.	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Ім.	Код
a_2	1	0
a_1	0	10
a_3	0	110
a_4	0	111

(b)

Симв.	Ім.	Код
a_2	1	0
a_4	1	10
a_3	0	110
a_1	0	111

(c)

Симв.	Ім.	Код
a_4	2	0
a_2	1	10
a_3	0	110
a_1	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3 a_4 a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

Симв.	Лічильник	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Лічильник	Код
a_3	1	0
a_4	0	10
a_2	0	110
a_1	0	111

(b)

Симв.	Лічильник	Код
a_3	1	0
a_4	1	10
a_2	0	110
a_1	0	111

(c)

Симв.	Лічильник	Код
a_4	2	0
a_3	1	10
a_2	0	110
a_1	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3a_4a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

Симв.	Ім.	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Ім.	Код
a_2	1	0
a_1	0	10
a_3	0	110
a_4	0	111

(b)

Симв.	Ім.	Код
a_2	1	0
a_4	1	10
a_3	0	110
a_1	0	111

(c)

Симв.	Ім.	Код
a_4	2	0
a_2	1	10
a_3	0	110
a_1	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3a_4a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

Симв.	Лічильник	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Лічильник	Код
a_3	1	0
a_4	0	10
a_3	0	110
a_4	0	111

(b)

Симв.	Лічильник	Код
a_3	1	0
a_4	1	10
a_3	0	110
a_4	0	111

(c)

Симв.	Лічильник	Код
a_4	2	0
a_3	1	10
a_3	0	110
a_4	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3a_4a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

Симв.	Лічильник	Код
a_1	0	0
a_2	0	10
a_3	0	110
a_4	0	111

(a)

Симв.	Лічильник	Код
a_2	1	0
a_1	0	10
a_3	0	110
a_4	0	111

(b)

Симв.	Лічильник	Код
a_2	1	0
a_4	1	10
a_3	0	110
a_1	0	111

(c)

Симв.	Лічильник	Код
a_4	2	0
a_2	1	10
a_3	0	110
a_1	0	111

(d)

На рис. наведено приклади для чотирьох символів та поведінка методу при стисканні рядка "$a_3a_4a_4$".

На рис. (a) зображено початковий стан. Після зчитування символу a_2 його лічильник збільшується, і оскільки він тепер найбільший, то рядки 1 і 2 міняються місцями (див. рис. (b)). Далі зчитується другий символ a_4, його лічильник збільшується, і рядки 2 та 4 переставляються (див. рис. (c)). Нарешті, після зчитування третього символу a_4, його лічильник стає найбільшим, що призводить до перестановки рядків 1 і 2 (див. рис. (d)).

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

У цьому алгоритмі лише одне місце може викликати проблему — це переповнення лічильників. Якщо змінні лічильників мають розрядність k біт, то їхнє максимальне значення дорівнює $2^k - 1$. Тому настане переповнення після 2^k-го збільшення. Це може статися, якщо розмір файлу, що стискається, заздалегідь не відомий, що буває часто. На щастя, нам не треба знати точних значень лічильників. Нам потрібен лише їхній порядок, тому цю проблему переповнення легко розв'язати.

Можна, наприклад, рахувати вхідні символи, і після $2^k - 1$ символу робити ціле ділення лічильників на 2 (або зрушувати їхній вміст на одну позицію вліво, що простіше).

Інший близький спосіб — це перевіряти кожен лічильник після його збільшення та після досягнення максимального значення виконувати ділення на 2 всіх лічильників. Цей підхід потребує більш рідкісного поділу, але складніших перевірок.

У будь-якому випадку, всі операції повинні виконуватися синхронно кодером і декодером.

Дякую за увагу!