Алгоритм относится к группе “жадных” (greedy) алгоритмов и использует только частоту появления одинаковых символов во входном блоке данных. Ставит в соответствие символам входного потока, которые встречаются чаще, цепочку битов меньшей длины. И напротив, встречающимся редко - цепочку большей длины. Для сбора статистики требует двух проходов по входному блоку (также существуют однопроходные адаптивные варианты алгоритма).

Характеристики алгоритма Хаффмана [1]:

- Степени сжатия: 8, 1.5, 1 (лучшая, средняя, худшая степени)
- Симметричность по времени: 2:1 (за счёт того, что требует двух проходов по массиву сжимаемых данных)
- Характерные особенности: один из немногих алгоритмов, который не увеличивает размера исходных данных в худшем случае (если не считать необходимости хранить таблицу перекодировки вместе с файлом).

Основные этапы алгоритма сжатия с помощью кодов Хаффмана

1. Сбор статистической информации для последующего построения таблиц кодов переменной длины
2. Построение кодов переменной длины на основании собранной статистической информации
3. Кодирование (сжатие) данных с использованием построенных кодов

Описанный выше алгоритм сжатия требует хранения и передачи вместе с кодированными данными дополнительной информации, которая позволяет однозначно восстановить таблицу соответствия кодируемых символов и кодирующих битовых цепочек.

Следует отметить, что в некоторых случаях можно использовать постоянную таблицу (или набор таблиц), которые заранее известны как при кодировании, так и при декодировании. Или же строить таблицу адаптивно в процессе сжатия и восстановления. В этих случаях хранение и передача дополнительной информации не требуется, а также отпадает необходимость в предварительном сборе статистической информации (этап 1).

В дальнейшем мы будем рассматривать только двухпроходную схему с явной передачей дополнительной информации о таблице соответствия кодируемых символов и кодирующих битовых цепочек.

Первый проход по данным: сбор статистической информации

Будем считать, что входные символы - это байты. Тогда сбор статистической информации будет заключаться в подсчёте числа появлений одинаковых байтов во входном блоке данных.

Построение кодов переменной длины на основании собранной статистической информации

Алгоритм Хаффмана основывается на создании двоичного дерева [3].

Изначально все узлы считаются листьями (конечными узлами), которые представляют символ (в нашем случае байт) и число его появлений. Для построения двоичного дерева используется очередь с приоритетами, где узлу с наименьшей частотой присваивается наивысший приоритет.

Алгоритм будет включать в себя следующие шаги:

Шаг 1. Помещаем все листья с ненулевым числом появлений в очередь с приоритетами (упорядочиваем все листья в порядке убывания числа появления)

Шаг 2. Пока в очереди больше одного узла, выполняем следующие действия:

Шаг 2a. Удаляем из очереди два узла с наивысшим приоритетом (самыми низкими числами появлений)

Шаг 2b. Создаём новый узел, для которого выбранные на шаге 2a узлы являются наследниками. При этом число появлений нового узла (его вес) полагается равным сумме появлений выбранных на шаге 2a узлов

Шаг 2c. Добавляем узел, созданный на шаге 2b, в очередь с приоритетами.

Единственный узел, который останется в очереди с приоритетами в конце работы алгоритма, будет корневым для построенного двоичного дерева.

Чтобы восстановить кодирующие слова, нам необходимо пройти по рёбрам от корневого узла получившегося дерева до каждого листа. При этом каждому ребру присваивается бит “1”, если ребро находится слева, и “0” - если справа (или наоборот).

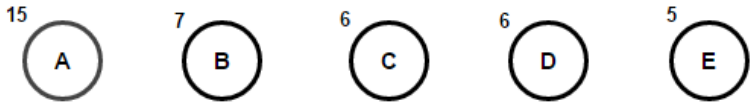
Кодирование Хаффмана

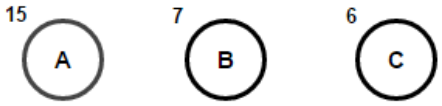
Техника работает, создавая **бинарное дерево** узлов. Узел может быть листовым узлом или внутренним узлом. Изначально все узлы являются листовыми узлами, которые содержат сам персонаж, вес (частоту появления) персонажа. Внутренние узлы содержат вес символов и ссылки на два дочерних узла. По общему соглашению, бит `0` представляет следующий левый дочерний элемент, и немного `1` представляет следующий правильный ребенок. Готовое дерево имеет `n` листовые узлы и `n-1` внутренние узлы. Рекомендуется, чтобы дерево Хаффмана отбрасывало неиспользуемые символы в тексте, чтобы получить наиболее оптимальную длину кода.

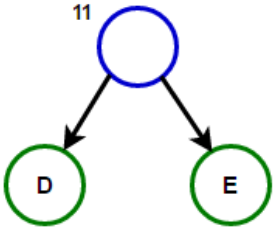
Мы будем использовать **приоритетная очередь** для построения дерева Хаффмана, где узел с наименьшей частотой имеет наивысший приоритет. Ниже приведены полные шаги:

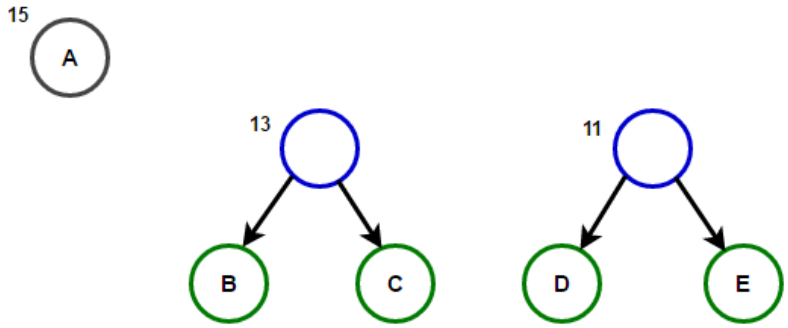
1. Создайте конечный узел для каждого символа и добавьте их в очередь приоритетов.
2. Пока в queue больше одного узла:
    - Удалите из queue два узла с наивысшим приоритетом (самой низкой частотой).
    - Создайте новый внутренний узел с этими двумя узлами в качестве дочерних элементов и частотой, равной сумме частот обоих узлов.
    - Добавьте новый узел в очередь приоритетов.
3. Оставшийся узел является корневым узлом, и дерево завершено.

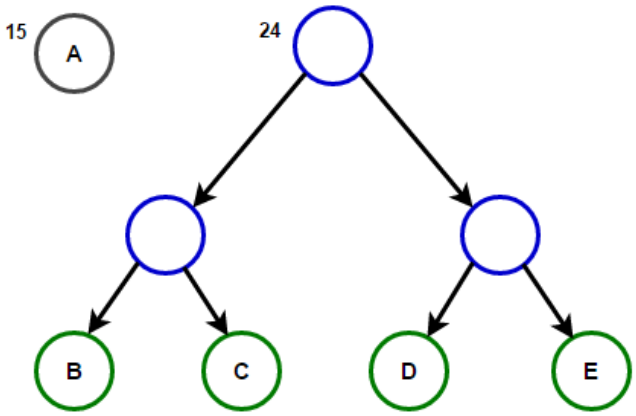
Рассмотрим некоторый текст, состоящий только из `'A'`, `'B'`, `'C'`, `'D'`, а также `'E'` символов, а их частота 15, 7, 6, 6, 5, соответственно. Следующие рисунки иллюстрируют шаги, за которыми следует алгоритм:

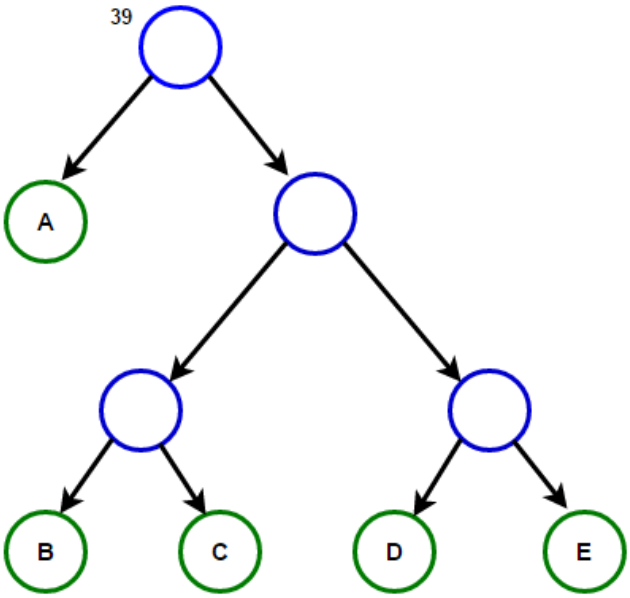




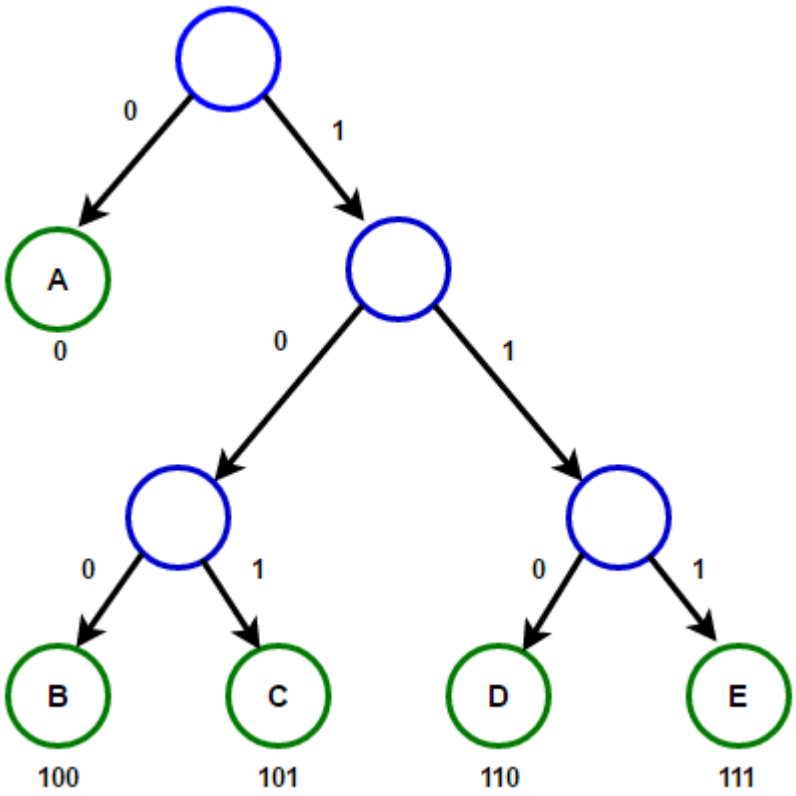








Путь от корня к любому конечному узлу хранит оптимальный код префикса (также называемый кодом Хаффмана), соответствующий символу, связанному с этим конечным узлом.




```
1   import java.util.Comparator;
2   import java.util.HashMap;
3   import java.util.Map;
4   import java.util.PriorityQueue;
5
6   // Узел дерева
7   class Node
8   {
9       Character ch;
10      Integer freq;
11      Node left = null, right = null;
12
13      Node(Character ch, Integer freq)
14      {
15          this.ch = ch;
16          this.freq = freq;
17      }
18
19      public Node(Character ch, Integer freq, Node left, Node
20  right)
21      {
22          this.ch = ch;
23          this.freq = freq;
24          this.left = left;
25          this.right = right;
26      }
27  }
28
```

```
class Main
{
    // Проходим по дереву Хаффмана и сохраняем коды Хаффмана на карте.
    public static void encode(Node root, String str,
                              Map<Character, String> huffmanCode)
    {
        if (root == null) {
            return;
        }

        // Найден листовой узел
        if (isLeaf(root)) {
            huffmanCode.put(root.ch, str.length() > 0 ? str : "1");
        }

        encode(root.left, str + '0', huffmanCode);
        encode(root.right, str + '1', huffmanCode);
    }

    // Проходим по дереву Хаффмана и декодируем закодированную строку
    public static int decode(Node root, int index, StringBuilder sb)
    {
        if (root == null) {
            return index;
        }

        // Найден листовой узел
        if (isLeaf(root))
        {
            System.out.print(root.ch);
            return index;
        }

        index++;

        root = (sb.charAt(index) == '0') ? root.left : root.right;
        index = decode(root, index, sb);
        return index;
    }

    // Вспомогательная функция для проверки, содержит ли дерево Хаффмана только один узел
    public static boolean isLeaf(Node root) {
        return root.left == null && root.right == null;
    }

    // Строит дерево Хаффмана и декодирует заданный входной текст
    public static void buildHuffmanTree(String text)
```

```
    {
        // Базовый случай: пустая строка
        if (text == null || text.length() == 0) {
            return;
        }

        // Подсчитаем частоту появления каждого символа
        // и сохранить его на карте

        Map<Character, Integer> freq = new HashMap<>();
        for (char c: text.toCharArray()) {
            freq.put(c, freq.getOrDefault(c, 0) + 1);
        }

        // создаем приоритетную очередь для хранения живых узлов
дерева Хаффмана.
        // Обратите внимание, что элемент с наивысшим
приоритетом имеет наименьшую частоту

        PriorityQueue<Node> pq;
        pq = new PriorityQueue<>(Comparator.comparingInt(l ->
l.freq));

        // создаем конечный узел для каждого символа и добавляем
его
        // в приоритетную очередь.

        for (var entry: freq.entrySet()) {
            pq.add(new Node(entry.getKey(), entry.getValue()));
        }

        // делаем до тех пор, пока в queue не окажется более
одного узла
        while (pq.size() != 1)
        {
            // Удаляем два узла с наивысшим приоритетом
            // (самая низкая частота) из queue

            Node left = pq.poll();
            Node right = pq.poll();

            // создаем новый внутренний узел с этими двумя
узлами в качестве дочерних
            // и с частотой равной сумме обоих узлов
            // частоты. Добавьте новый узел в очередь
приоритетов.

            int sum = left.freq + right.freq;
            pq.add(new Node(null, sum, left, right));
        }

        // `root` хранит указатель на корень дерева Хаффмана
        Node root = pq.peek();

```

```java
        // Проходим по дереву Хаффмана и сохраняем коды Хаффмана на карте
        Map<Character, String> huffmanCode = new HashMap<>();
        encode(root, "", huffmanCode);

        // Выводим коды Хаффмана
        System.out.println("Huffman Codes are: " + huffmanCode);
        System.out.println("The original string is: " + text);

        // Печатаем закодированную строку
        StringBuilder sb = new StringBuilder();
        for (char c: text.toCharArray()) {
            sb.append(huffmanCode.get(c));
        }

        System.out.println("The encoded string is: " + sb);
        System.out.print("The decoded string is: ");

        if (isLeaf(root))
        {
            // Особый случай: для ввода типа a, aa, aaa и т. д.
            while (root.freq-- > 0) {
                System.out.print(root.ch);
            }
        }
        else {
            // Снова проходим по дереву Хаффмана и на этот раз
            // декодируем закодированную строку
            int index = -1;
            while (index < sb.length() - 1) {
                index = decode(root, index, sb);
            }
        }
    }

    // Реализация алгоритма кодирования Хаффмана на Java
    public static void main(String[] args)
    {
        String text = "Huffman coding is a data compression algorithm.";
        buildHuffmanTree(text);
    }
}
```