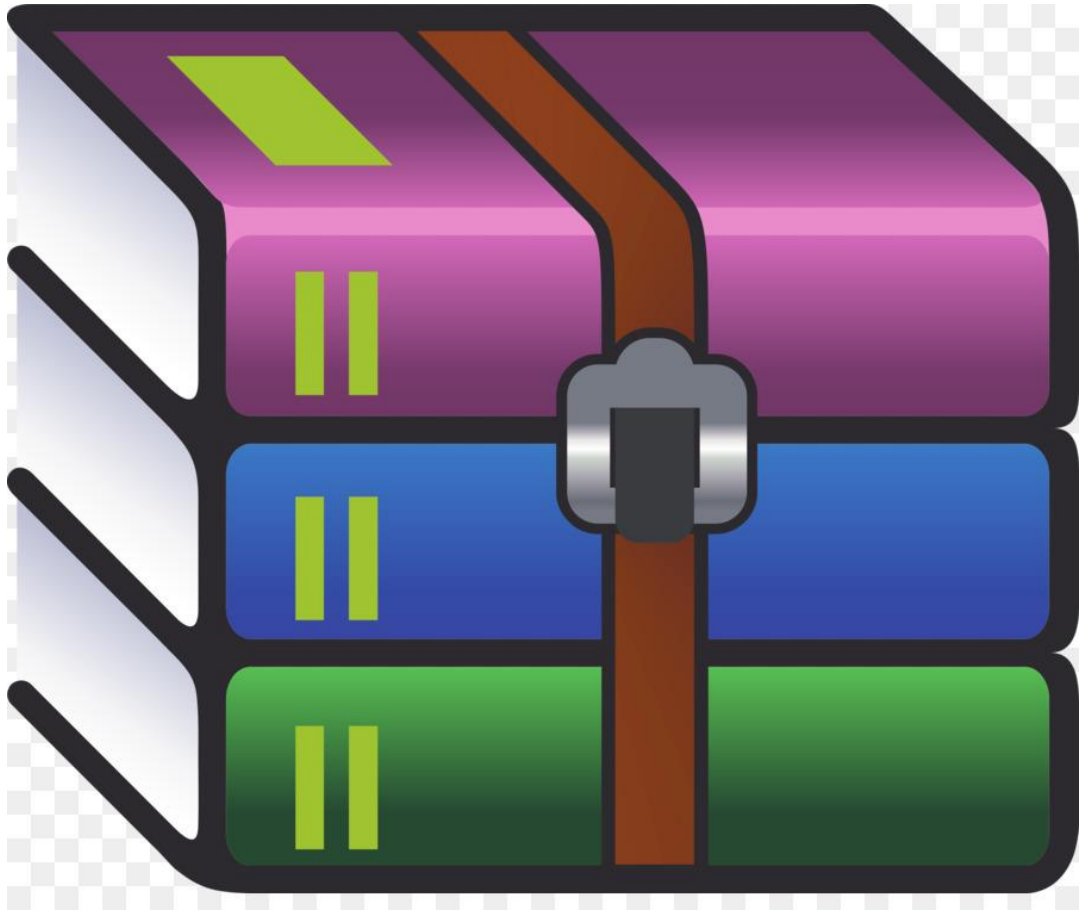
Архиватор



Задание

Реализовать программу, кодирующая и сжимающая файлы по алгоритму Хаффмана
Реализовать программу для декодирования файлов

```
git clone https://github.com/SergeyBalabaev/Archiver
```

Основы теории информации

Информация (Information) — содержание сообщения или сигнала; сведения, рассматриваемые в процессе их передачи или восприятия, позволяющие расширить знания об интересующем объекте

Информация – первоначально — сведения, передаваемые одними людьми другим людям устным, письменным или каким-нибудь другим способом

Информация - как коммуникацию, связь, в процессе которой устраняется неопределенность. (К. Шеннон)

Мера информации

Пусть X – источник дискретных сообщений. Число различных состояний источника – N.
Переходы из одного состояния в другое не зависят от предыдущих состояний, а вероятности перехода в эти состояния $p_j=P\{X = x_j\}$

Тогда за меру количества информации примем следующую величину:

$$H(X) = -\sum_{k=1}^{N} p_k \log(p_k)$$

Эта величина называется **энтропией**

Энтропия

Свойства энтропии

1) Энтропия неотрицательна
2) Максимально возможное значение энтропии равно log(N)
3) Энтропия нескольких независимых файлов равна сумме энтропий каждого из них

Битовые затраты – среднее число бит приходящееся на один символ сообщения

$$R=\sum_{k=1}^{N} p_k R_k$$

R_k - число бит в коде символа x_k

Пример

Задача:

Пусть пришло следующее сообщение: «мамамылараму»

Рассчитаем энтропию сообщения и битовые затраты. Будем считать, что один символ кодируется 1 байтом.

1) Рассчитаем вероятности появления символов

мамамылараму

Символ	**м**	**а**	**ы**	**л**	**р**	**у**	**Σ**
Количество	4	4	1	1	1	1	**12**
Вероятность	1/3	1/3	1/12	1/12	1/12	1/12	**1**

Пример

2) Рассчитаем энтропию и битовые затраты

$$H(X) = -\sum_{k=1}^{N} p_k \log(p_k)$$

$$H(X) = -\sum_{k=1}^{6} p_k \log(p_k) = -\left(\frac{1}{3}\log\frac{1}{3} + \frac{1}{3}\log\frac{1}{3} + \frac{1}{12}\log\frac{1}{12} + \frac{1}{12}\log\frac{1}{12} + \frac{1}{12}\log\frac{1}{12} + \frac{1}{12}\log\frac{1}{12}\right) \sim \mathbf{2,25}$$

$$R = \sum_{k=1}^{N} p_k R_k$$

$$H(X) = \sum_{k=1}^{6} p_k R_k = \left(\frac{1}{3} * 8 + \frac{1}{3} * 8 + \frac{1}{12} * 8 + \frac{1}{12} * 8 + \frac{1}{12} * 8 + \frac{1}{12} * 8\right) = \mathbf{8}$$

Идея сжатия

Давайте заменим стандартный равномерный ASCII код на неравномерный так, чтобы часто встречающимся символам соответствовали более короткие кодовые последовательности. Если средние битовые затраты будут меньше, чем 8 бит, то сжатие удалось!

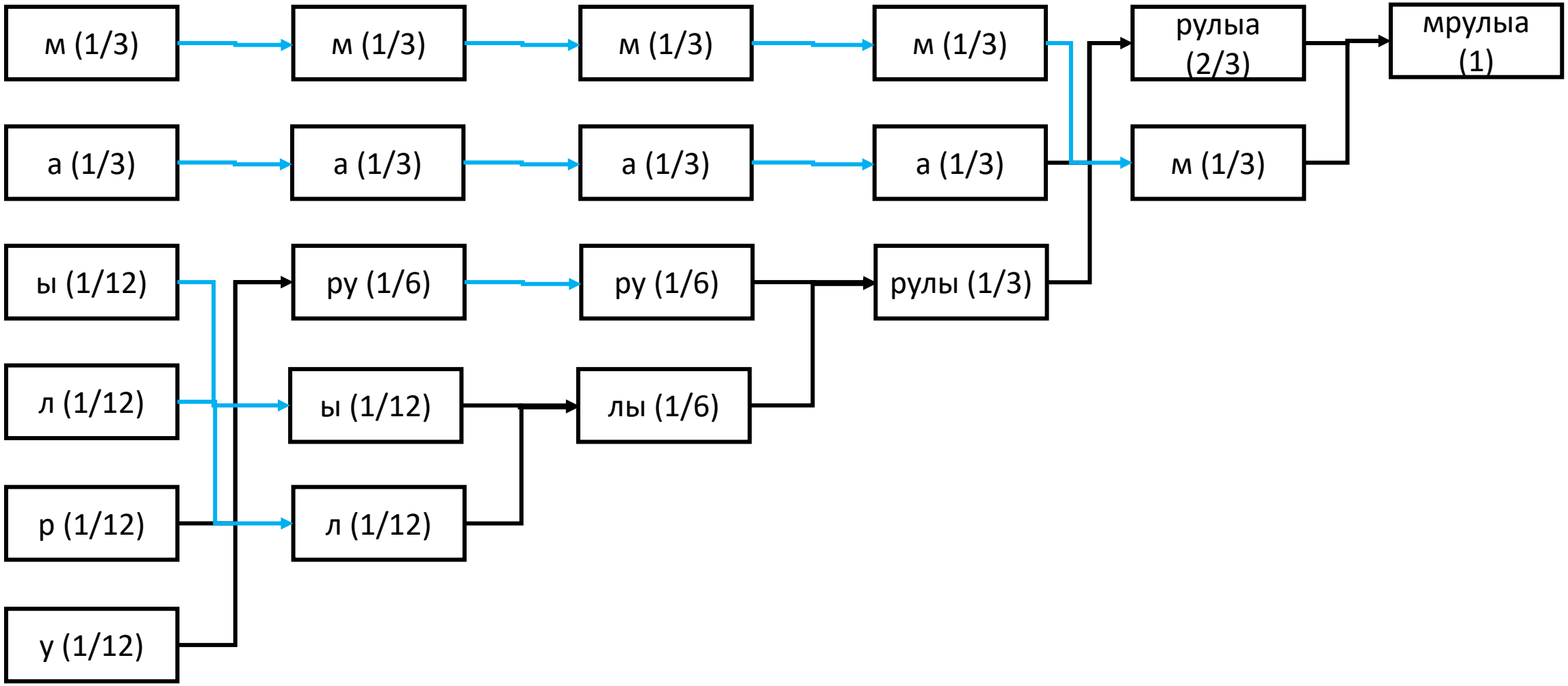
Алгоритм Хаффмана

На вход алгоритма подается таблица символов

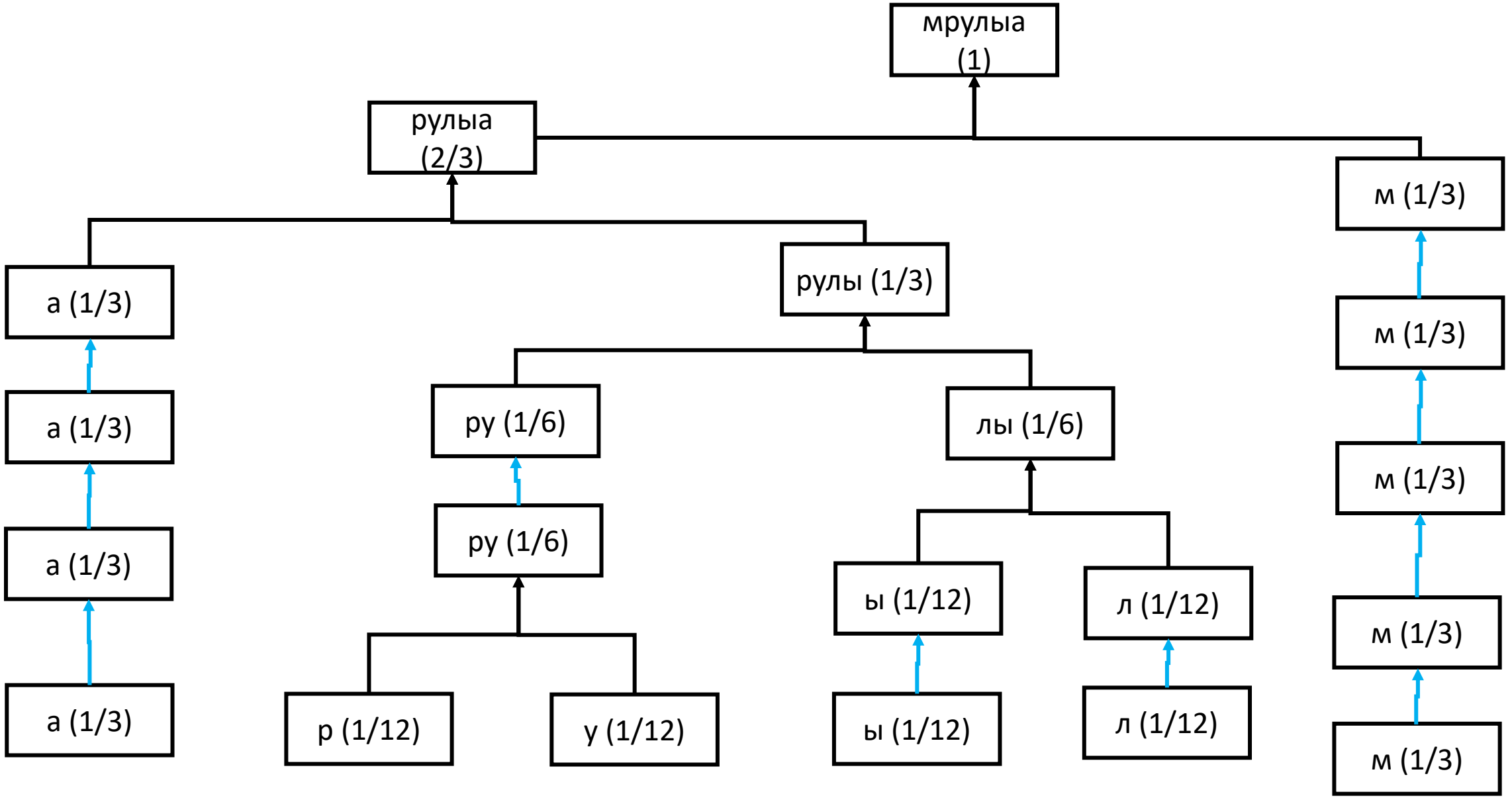
1. Построение дерева Хаффмана
    1. Упорядочиваем таблицу символов в порядке убывания вероятностей
    2. Два последних символа, имеющих наименьшие вероятности появления объединяются в новый символ
    3. Если есть еще символы, то возвращаемся на 1.1
2. Построение битового кода

    Для каждого узла дерева строим по два ребра, приписываем одному из них 1, другому 0

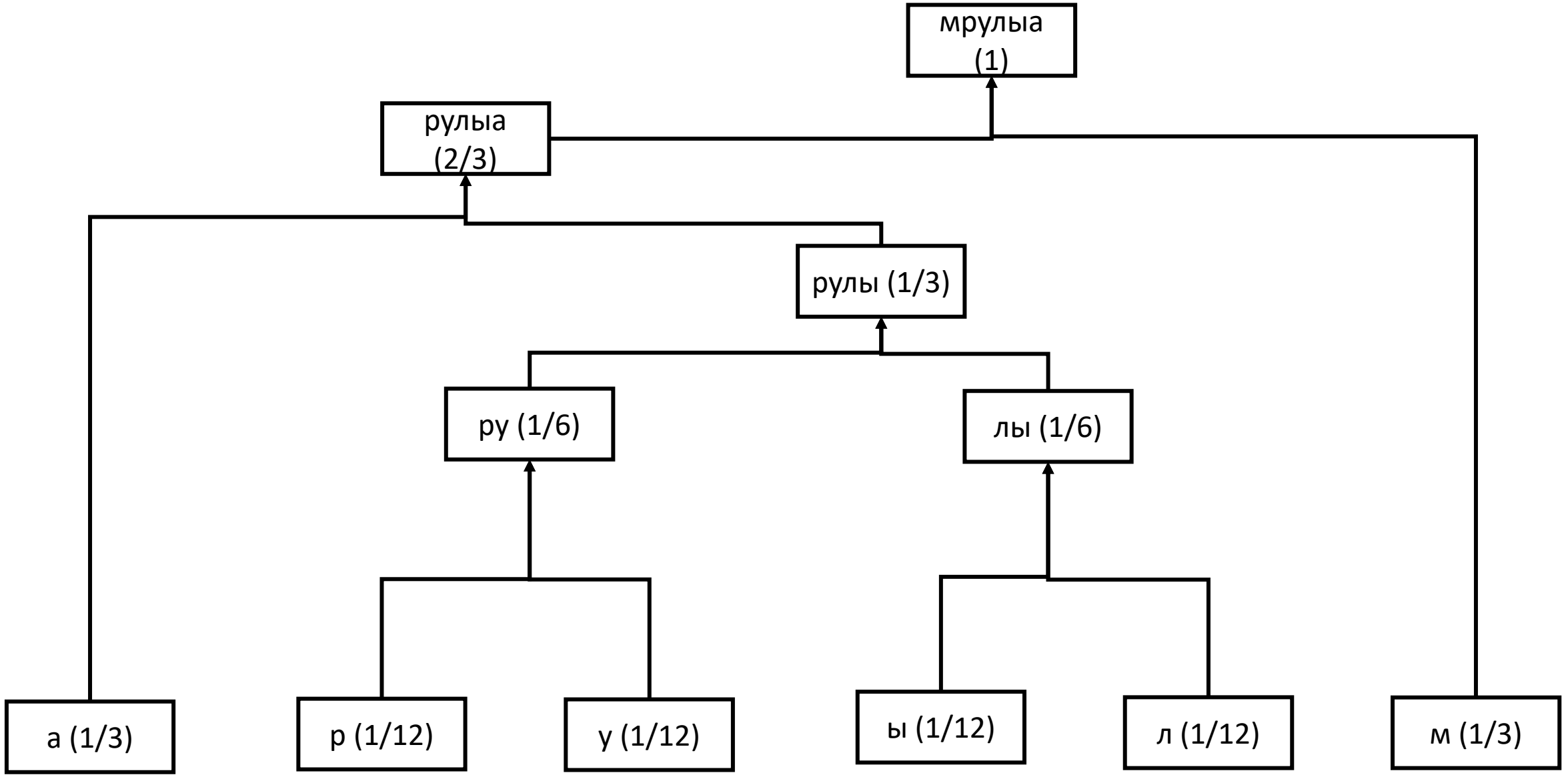
Алгоритм Хаффмана - пример



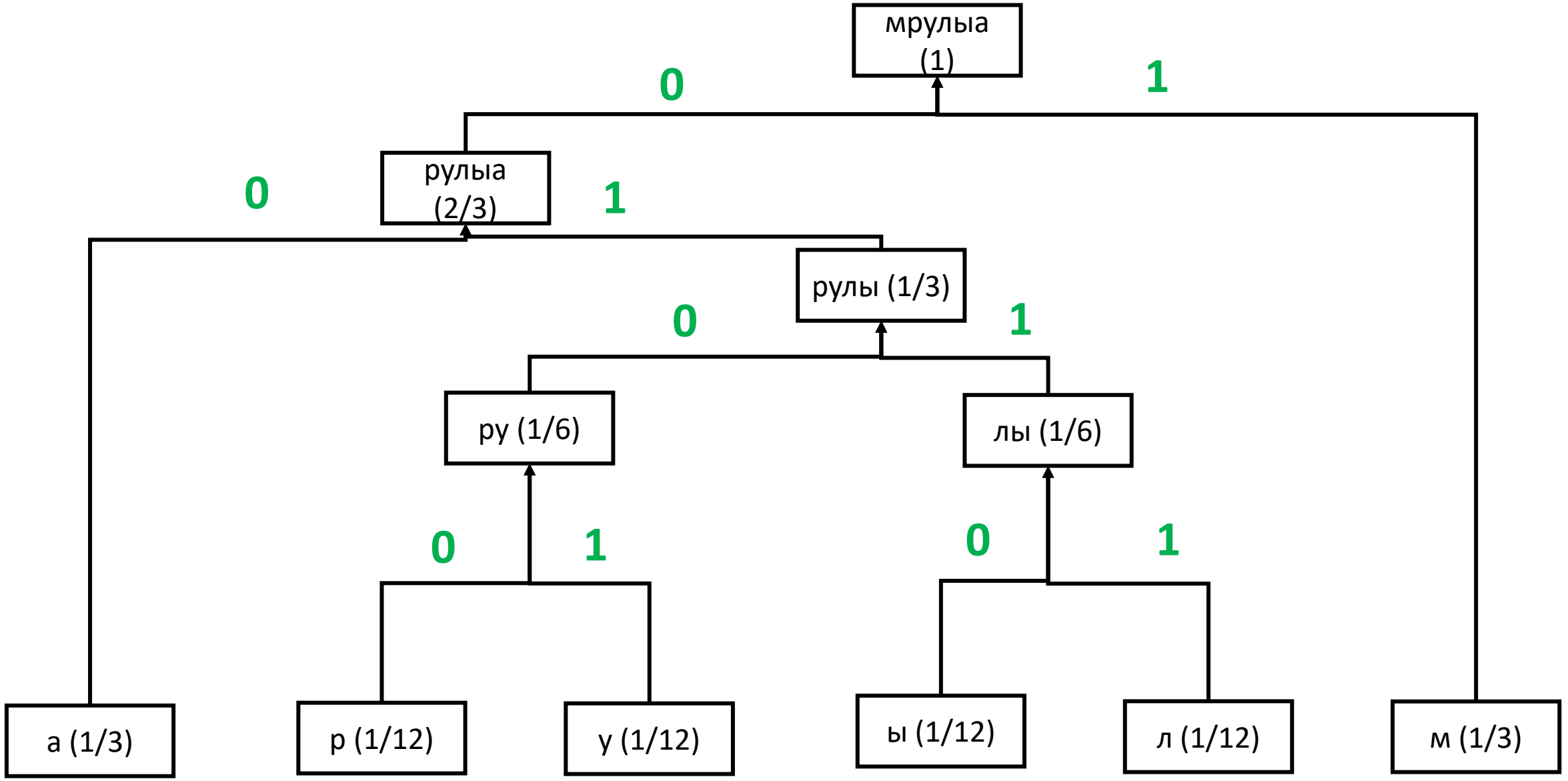
Алгоритм Хаффмана



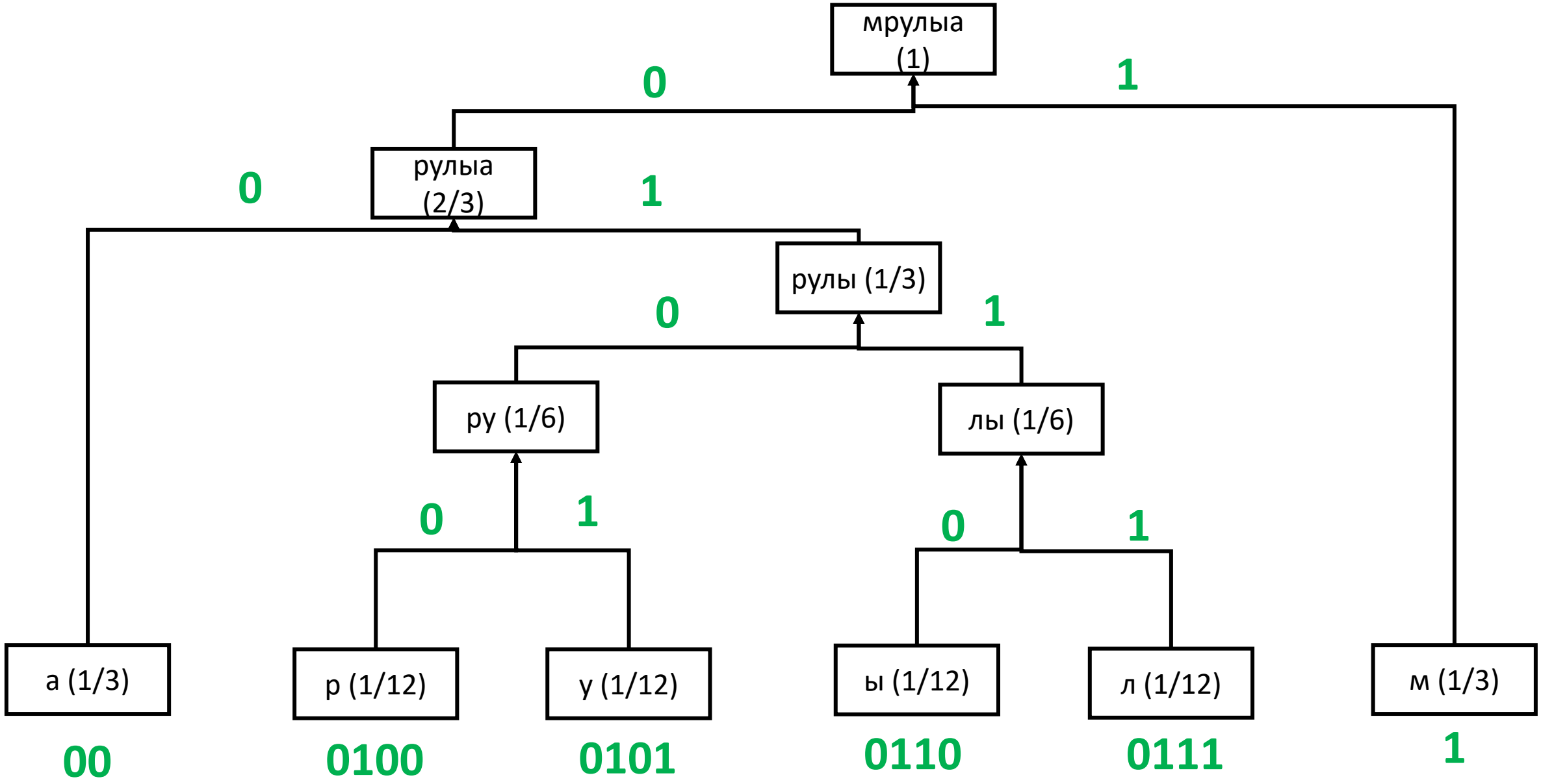
Алгоритм Хаффмана



Алгоритм Хаффмана



Алгоритм Хаффмана



Расчет битовых затрат

Без кодирования	R = 8
Кодирование «в лоб»	R ~ 2,42
Метод Хаффмана	R ~ 2,33
Энтропия	H ~ 2,25

Возможная программная реализация

Программа может быть разделена на 6 модулей (см. следующий слайд). Порядок работы:

1) Открытие и чтение файла
2) Расчёт частоты встречаемости символов
3) Сортировка массива символов по частоте по убыванию
4) Создание дерева Хаффмана
5) Создание кодов Хаффмана
6) Создание промежуточного файла, состоящего из 0 и 1 – закодированный полученным кодом первый файл
7) Запись полученной последовательности в архивный файл

Программная реализация

math_func.c	types.h	arch_logic.c	information.c	file_In_out.c
Описание **математических функций** (расчет энтропии, битовых затрат и т.п.)	Описание **глобальных** типов	Описание **логики работы** архиватора	Вывод информации о работе программы	Работа со вводом и выводом

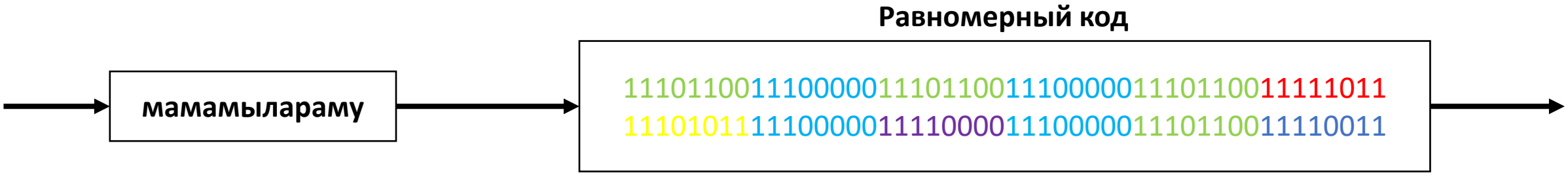
main.c

Комментарии

Программа может быть реализована по следующим пунктам:

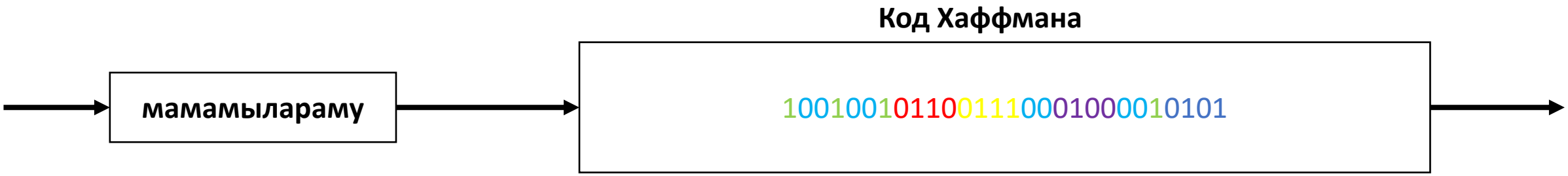
1) Открытие файла в бинарном виде и посимвольное чтение потока байтов
2) Расчет гистограммы появлений символов
3) Функция расчета энтропии по полученной в п. 2 гистограмме
4) Функция построения дерева Хаффмана
5) Функция создания кодов по полученному дереву
6) Кодирование входного файла с помощью полученных кодов и запись результата в текстовый файл. Данный файл представляет собой набор 0 и 1.
7) Создание нового файла, содержащий сжатый файл на основе полученного результата. *(см слайд 20)*

Комментарии по реализации



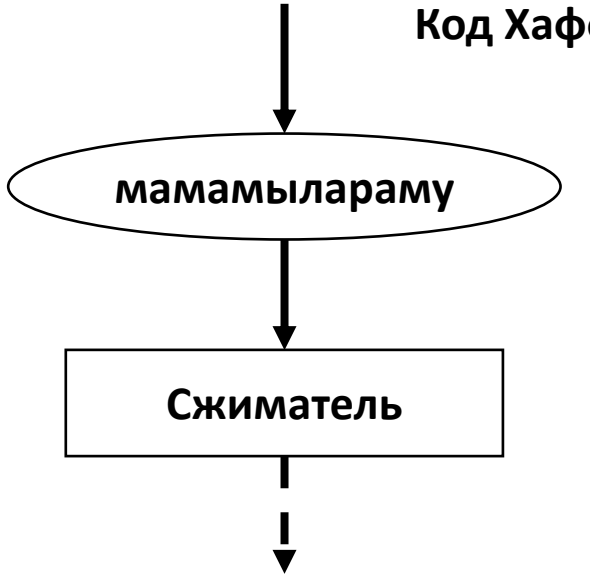
Символ	Код (ASCII Win-1251)
м	11101100
а	11100000
ы	11111011
л	11101011
р	11110000
у	11110011

Комментарии по реализации



Символ	Код Хаффмана
м	1
а	00
ы	0110
л	0111
р	0100
у	0101

Комментарии по реализации



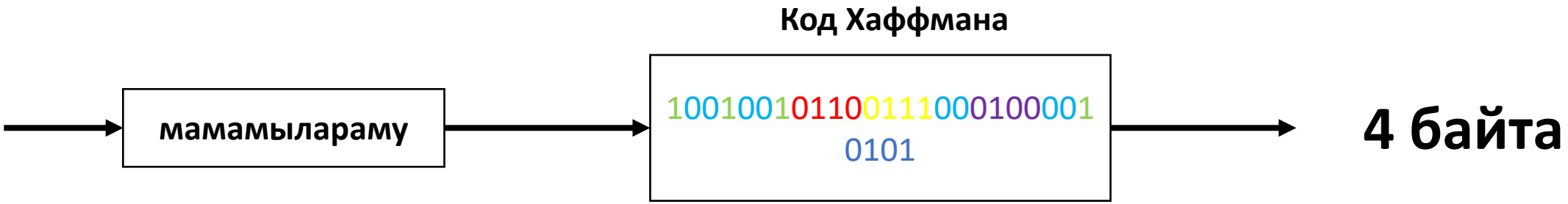
10010010 11001110 00100001 0101

1 байт 2 байт 3 байт 01010000

4 байт

Дополняем нулями до 8 символов

Комментарии по реализации



Комментарии по реализации

Алгоритм построения дерева можно реализовать следующим образом:

1. Создание **рекурсивной функции** построения дерева Хафмана.

   symbol makeTree(symbol* psym[], int k)*

Передаем по указателю значение нового пришедшего элемента. В данном случае, самого нижнего. Также передаем общее количество символов k. Возвращаем - указатель на новую структуру типа symbol

2. Создание временной структуры symbol. В нее мы запишем новый символ, состоящий из суммы двух нижних

```
symbol* temp;
///
//т.к. символ состоит из последних двух, то запишем их как «родительские»
temp->left = psym[k - 2];
temp->right = psym[k - 1];
///
```

3. Заменяем эти два символа на получившийся новый и упорядочиваем дерево
4. Рекурсивно вызываем функцию makeTree, только с меньшим числом символов на 1, т.к. два из них склеились

```
return makeTree(psym, k - 1);
```

Комментарии по реализации

Алгоритм построения кодов можно реализовать следующим образом:

1. Рекурсивная функция кодирования. На вход принимает указатель на полученный корень дерева
   void makeCodes(symbol root)*
2. Обходим дерево начиная с корня. Сначала идем налево. Если есть такой символ, то к уже полученному коду добавляем символ 0
3. Далее рекурсивно вызываем функцию, но уже для элемента, лежащего слева от него

Таким образом мы дойдем до конца дерева по левым ветвям. Далее аналогично добавляем возможность перебирать правые ветви.

Комментарии по реализации

Для записи сжатого файла возможно использовать объединение и битовые поля:

```
union code {
   unsigned char sym_to_write;//переменная содержащая код для записи в сжатый файл

   struct byte //представлена в виде битового поля - каждый unsigned символ занимает 1 бит
   {
      unsigned b1 : 1;
      unsigned b2 : 1;
      unsigned b3 : 1;
      unsigned b4 : 1;
      unsigned b5 : 1;
      unsigned b6 : 1;
      unsigned b7 : 1;
      unsigned b8 : 1;
   }byte;
};
```

Комментарии по реализации

Для записи сжатого файла возможно использовать объединение и битовые поля:

10010010

```
///
union code code1;//инициализируем переменную code1
code1.byte.b1 = 1;
code1.byte.b2 = 0;
code1.byte.b3 = 0;
code1.byte.b4 = 1;
code1.byte.b5 = 0;
code1.byte.b6 = 0;
code1.byte.b7 = 1;
code1.byte.b8 = 0;
///
```

code1. sym_to_write – содержит полученный символ, который записывается в файл

Возможная программная реализация

Порядок работы:

1) Открытие и чтение файла
2) Расчёт частоты встречаемости символов
3) Сортировка массива символов по частоте по убыванию

БДЗ 1

4) Создание дерева Хаффмана
5) Создание кодов Хаффмана
6) Создание промежуточного файла, состоящего из 0 и 1 – закодированный полученным кодом первый файл
7) Запись полученной последовательности в архивный файл

БДЗ 2

Желаю успеха!