Федеральное государственное бюджетное образовательное учреждение

высшего образования «Сибирский государственный университет телекоммуникаций и информатики»

Факультет ИВТ

Кафедра вычислительных систем

**Курсовая работа**

на тему «Сжатие данных»

Вариант 4.2 «Алгоритм Хаффмана»

Выполнил:студент гр. ИС-241Кулик П.Е.

Проверил:Старший преподаватель Кафедры ВС

Фульман В.О.

Новосибирск, 2023

Оглавление

[Коэффициент сжатия 3](#_Toc135385225)

[Работа с битовым массивом 3](#_Toc135385226)

[Задание 5](#_Toc135385227)

[Критерии оценки 5](#_Toc135385228)

[Указание к выполнению задания 5](#_Toc135385229)

[Структура проекта 7](#_Toc135385230)

[Способ представления данных 8](#_Toc135385231)

[Основные алгоритмы 8](#_Toc135385232)

[Коэффициент сжатия 11](#_Toc135385233)

Сжатие данных

Заданиями данной главы предусматривается сжатие данных на основе различных известных алгоритмов.

# Коэффициент сжатия

Коэффициент сжатия – основная характеристика алгоритмов сжатия. Она определяется как отношение объёма исходных несжатых данных к объёму сжатых, то есть:

,

где k — коэффициент сжатия, st — объём исходного сообщения, а sc — объём сжатого сообщения. Таким образом, чем выше коэффициент сжатия, темалгоритм эффективнее. Если k = 1, то алгоритм не производит сжатия, то естьвыходное сообщение оказывается по объёму равным входному. Если k < 1, тоалгоритм порождает сообщение большего размера, нежели несжатое, то есть,совершает «вредную» работу.

# Работа с битовым массивом

Алгоритмы Шенона-Фано и Хаффмана предполагают построение новых кодов для символов входного алфавита, основанных на частоте встречаемости этих символов. При этом длина кодов часто встречающихся символов будет наименьшей (1 – 4 бит), в то время, как коды редко встречающихся символов могу составлять 8, 9, 10 и более бит. Хранение символов с короткими (< 8 бит) кодами в 8-битном байте оказывается бесполезным, так как эффект от сжатия кода нивелируется наличием неиспользуемых бит. С другой стороны, не существует возможности сохранить 10-битное слово в одном байте, потребуется использовать ячейки двухбайтного типа.

Таким образом, возникает необходимость компактного хранения кодов, имеющих различную длину в массивах, элементы которых имеют фиксированный размер. Для решения этой задачи может использоваться битовый массив. Для работы с этим массивом необходимо разработать две функции:

setbits(array, offset, value, value\_len) и getbit(array, offset). Рассмотрим далее принцип действия этих функций.

Пусть дана таблица α кодов, построенная некоторым алгоритмом сжатия (см. Таблицу 1).

Таблица 1. Таблица α кодов.

|  |  |  |
| --- | --- | --- |
| Символ | Код | Длина кода |
| a | 010 | 3 |
| b | 1100 | 4 |
| c | 010111 | 6 |
| d | 1110001111 | 10 |

Необходимо закодировать сообщение: "abaabcd". На рисунке 1 показано исходное сообщение.

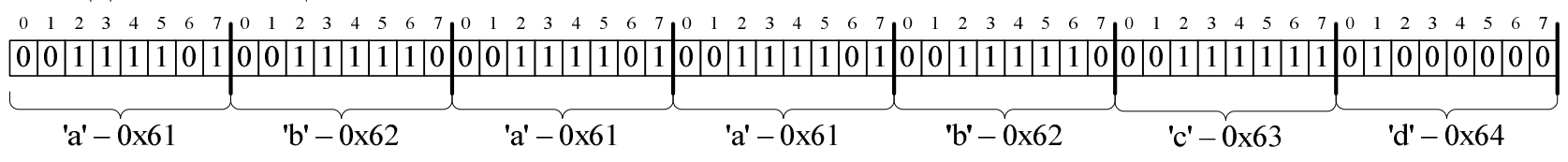


Рисунок 1. Исходное сообщение в формате ASCII

Тогда закодированное сообщение представлено на рисунке 2.

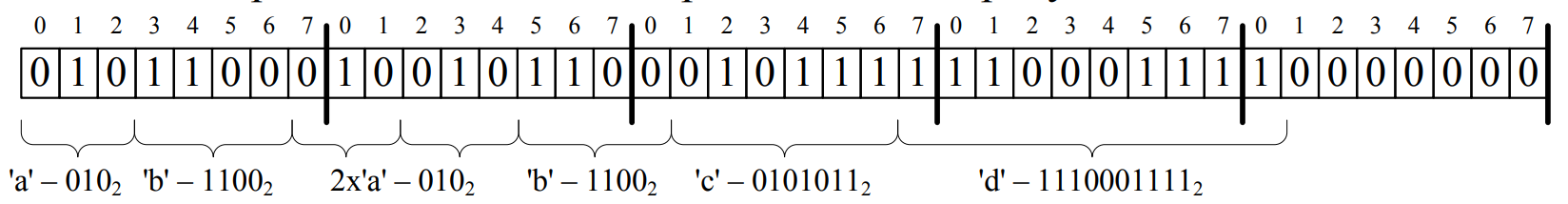


Рисунок 2. Закодированное сообщение

Видно, что исходное сообщение занимает 7 байт, а сжатое – 5 байт, при этом 7 бит последнего байта свободны, т.е. можно сказать, что длина сжатого сообщения близка к 4-м байтам. Таким образом коэффициент сжатия для данного кодирования будет:

Далее в листинге 1 представлен псевдокод алгоритма формирования битового массива С, хранящего закодированное (сжатое) сообщение T.

|  |
| --- |
| Листинг 1. Псевдокод алгоритма формирования битового массива С |
| ENCODE\_MSG(T, a)  offs = 0  C = “”  n = len(T)  for i = 1 to n do  v = a[T[i]].code  l = a[T[i]].len  offs = setbits(C, offs, v, l)  return C |

Функция setbits должна по номеру бита определить номера байтов, в которых будет сохранено значение v. Например, если offs = 10, а len = 15, то первый байт, в который будут записаны разряды v, имеет номер offs div 8, при этом если offs mod 8 ≠ 0, то необходимо прибавить еще один байт.

Для рассмотренного примера 10 div 8 = 1, т.к. 10 mod 8 = 2 ≠ 0, то номер первого байта будет 2. Далее необходимо разместить биты значения v в байтах массива как показано на рисунке 3.

Для решения задач по выделению фрагментов значения и помещения их в соответствующие байты необходимо использовать поразрядные операции.

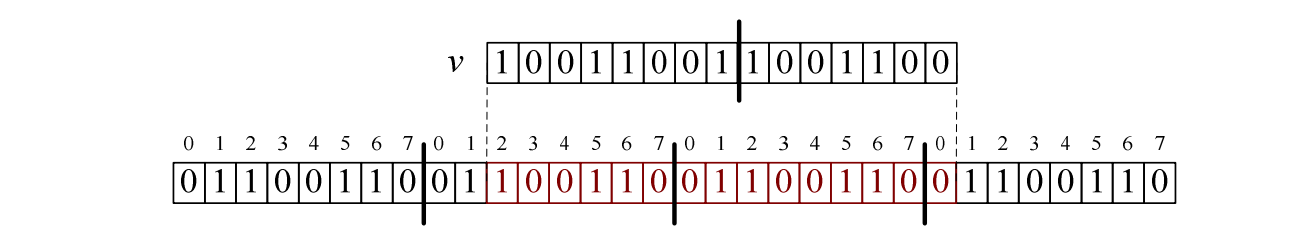


Рисунок 3. Запись в битовый массив

Процесс декодирования сообщения представлен в листинге 2.

|  |
| --- |
| Листинг 2. Процесс декодирования сообщения |
| DECODE\_MSG(C, a)  T = “”  n = len(C)  tmp = a  v = 0  for i = 1 to n \* 8 do  v = v << 1 | getbit(C, i)  tmp = clear(tmp, v)  if find\_code(tmp, v, s) = 1 then  T = Ts  tmp = C  v = 0  else if |tmp| = 0 then  print “Ошибка декодирования”  return T |

ВАРИАНТ 4.2 Алгоритм Хаффмана

# Задание

Реализовать программу hcompress сжатия текстовых файлов на английском языке алгоритмом Хаффмана. Сжатие осуществляется с аргументом командной строки -c (compress), а распаковка – с аргументом -d (decompress). Опция -o указывает имя выходного файла. Например:

|  |  |
| --- | --- |
| $ ./hcompress –c –o file.hc file.txt | #сжатие file.txt в file.hc |
| $ ./hcompress –d –o file1.txt file.hc | #распаковка file.hc в file1.txt |

# Критерии оценки

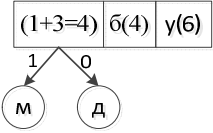
- Оценка «хорошо»: реализован алгоритм сжатия, не обеспечено компактное расположение сжатого текста в выходном файле – код каждого символа занимает целое число байт.

- Оценка «отлично»: закодированный текст компактно упаковывается в файл без учета байтовых границ (с помощью битового массива, как описано в общей информации к разделу).

# Указание к выполнению задания

В алгоритме Хаффмана на основании таблицы частот встречаемости символов в сообщении строится дерево кодирования Хаффмана (Н-дерево). Например, исходный текст: «ббббмдддууууууууу». Частоты встречающихся в исходном тексте символов: б – 4, м – 1, ‘д’ – 3, ‘у’ – 6. Основные этапы алгоритмам Хаффмана:

1. Список X символов упорядочивается по возрастанию частоты встречаемости и записывается в очередь с приоритетом (каждый элемент очереди - узел дерева Хаффмана). 
2. Выбираются два элемента очереди ω1 и ω2 с наименьшими весами (частотами встречаемости). Для рассматриваемого примера ω1 = ‘м’ и ω2 = ‘д’.
3. Создается их родитель ω с весом, равным их суммарному весу (Σ =1+3=4). Родитель добавляется в соответствующее место очереди, а два его потомка удаляются оттуда.
4. Кодовой комбинации символов дописывается 1 (ω1) или 0 (ω2).



1. Если в очереди находится один элемент - дерево Хаффмана построено, иначе переходим на шаг 2.

В листинге 3 представлен высокоуровневый псевдокод алгоритмы построения дерева Хаффмана.

|  |
| --- |
| Листинг 3. Псевдокод алгоритма построения дерева Хаффмана |
| HTREE(queue, sequence)  init\_queue(queue)  while size(queue) > 1 do  ω1=dequeue\_min(queue)  ω2=dequeue\_min(queue)  ω.left = ω1  ω.right = ω2  ω.weight = ω1.weight + ω2.weight  enqueue(queue, ω)  return dequeuer(queue) |

Для формирования таблицы кодов необходимо выполнит обход построенного дерева в глубину.

Для распаковки сжатых данных необходимо знать все использованные коды, необходимо сохранять полученную таблицу соответствия «символ-код» в файл со сжатой информацией. Итоговое дерево Хаффамана для сообщения, приведенного выше, изображено на рисунке 4:

Будут получены следующие коды

символов:

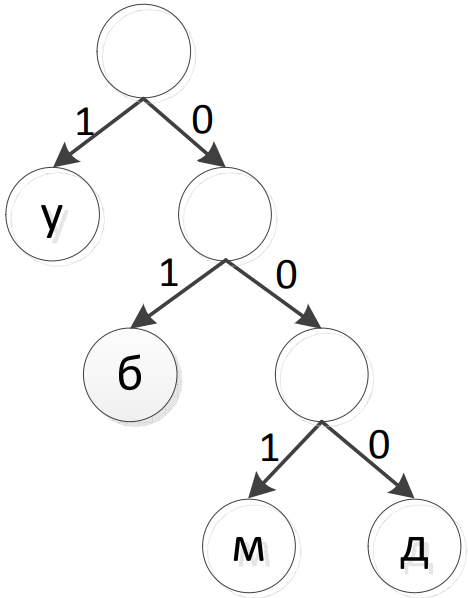


Рисунок 4. Итоговое дерево Хаффмана

у – 1

б – 01

д – 000

м – 001

Текст «ббббмдддууууууууу» будет закодирован в виде двоичной последовательности:

01|01|01|01|001|000|000|000|1|1|1|1|1|1|1|1|1

Рисунок 5. Закодированная двоичная последовательность

Выполнение работы

# Структура проекта

Исходный код программы имеет множество различных функций и разделён не несколько тематических файлов. Процесс обмена данными в программе можно увидеть на рисунке 6.

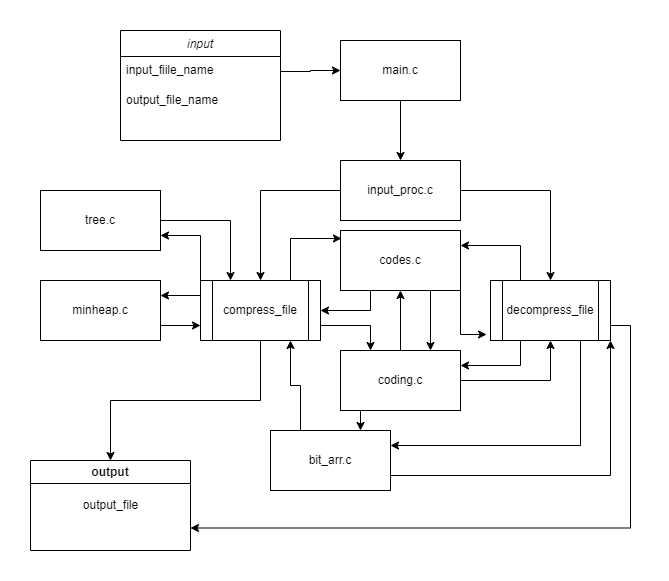


Рисунок 6. Структура проекта

После того, как пользователь вводит команд для запуска программы, в зависимости от аргументов, имена файлов передаются в функцию compress\_file, либо decompress\_file. Обе функции используют файлы coding.c и codes.c. Функция compress\_file дополнительно подключает файлы tree.c и minheap.c для построения сжатых кодов, после чего передаёт данные в coding.c, который возвращает битовый массив обратно в compress\_file. Далее происходит запись сжатого сообщения. Функция decompress\_file обменивается данными с coding.c и bit\_arr.c, чтобы расшифровать сжатые данные, после чего записывает их в файл.

# Способ представления данных

Для битового массива используется массив типа uint8\_t, запись в который производится, начиная с любого бита этого массива, с помощью функции setbits. При помощи функции get\_bit можно получить значение любого бита из этого массива.

Чтобы построить таблицу кодов Хаффмана используется структура данных, представляющая собой очередь с приоритетом, в которой каждый узел является бинарным деревом. Используя такую структуру, построение дерева Хаффмана выглядит примерно следующим образом.

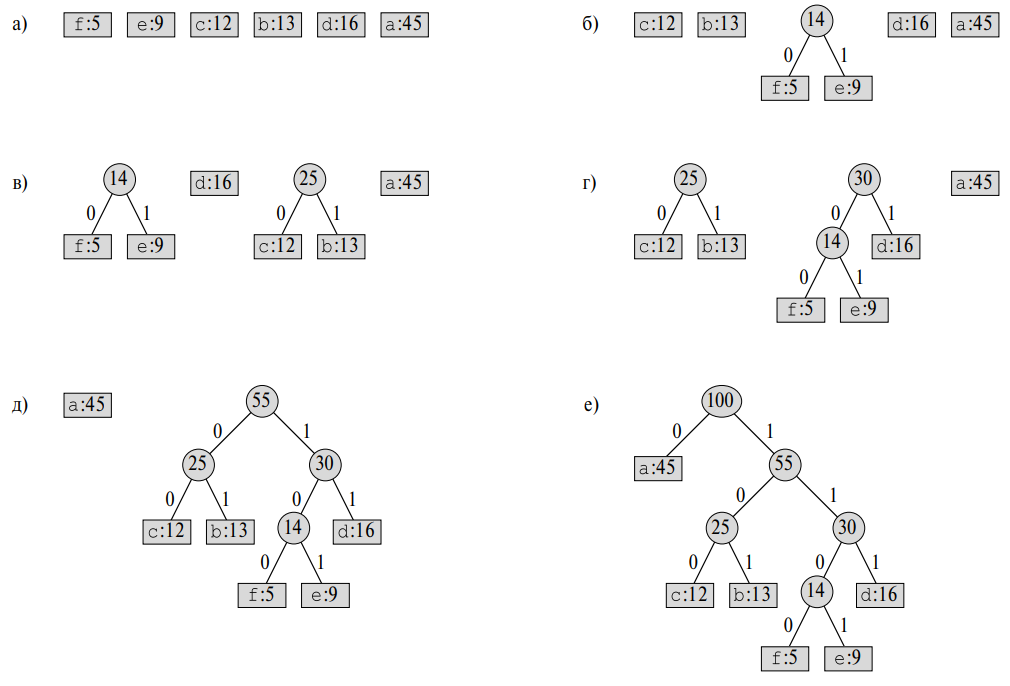


Рисунок 7. Построение дерева Хаффмана.

Для того, чтобы посчитать и временно сохранить частоту встречаемости каждого символа, используется массив типа uint64\_t, значения в котором соответствуют частоте, а индексы кодам символов в ascii таблице.

# Основные алгоритмы

Псевдокод таких алгоритмов как: ENCODE\_MSG, DECODE\_MSG и HTREE, уже представлен в описании главы и задания. В данной реализации программы структура алгоритмов была соблюдена настолько, насколько это было возможно, чтобы они были наиболее похожи на их описание, поэтому нет необходимости описывать их псевдокод снова.

|  |
| --- |
| Листинг 4. Псевдокод функции setbits |
| setbits(bit\_arr, offs, v, v\_len)  byte\_n = offs / 8  byte\_offs = offs % 8  v = v << (16 – v\_len)  for i = v\_len to 1 do  bit\_arr[byte\_n] = bit\_arr[byte\_n] | ((v & 0x8000) >> (byte\_offs + 8))  byte\_offs++  if byte\_offs % 8 = 0 then  byte\_n++  byte\_offs = 0  v = v << 1  return offs + v\_len |

Конкретная функция setbits предназначена для записи битовой последовательности заданного значения в массив байтов. Она принимает следующие параметры:

1. bit\_arr: Указатель на массив байтов, в котором будет производиться запись.
2. offs: Смещение, с которого начнется запись в массиве bit\_arr.
3. value: Значение, биты которого будут записаны в массив bit\_arr.
4. value\_len: Количество бит, которые нужно записать из значения value.

Функция выполняет следующие шаги:

1. Вычисляется номер байта (byte\_n), с которого начнется запись, путем деления смещения offs на 8.
2. Вычисляется смещение внутри байта (byte\_offs), показывающее, с какого бита начнется запись, путем вычисления остатка от деления offs на 8.
3. Значение value сдвигается влево на разницу между 16 и value\_len, чтобы выровнять его по левому краю 16-битного числа.
4. Цикл выполняется value\_len раз.
5. В каждой итерации цикла берется старший бит значения value с помощью маски 0x8000 и сдвигается на правильное смещение, чтобы совпадать с текущим байтом и его смещением.
6. Оператор | выполняет логическое ИЛИ между текущим значением байта bit\_arr[byte\_n] и установленными битами из значения value, обновляя текущий байт.
7. Смещение внутри байта увеличивается для перехода к следующему биту.
8. Если достигнут конец текущего байта, происходит переход к следующему байту.
9. По окончании цикла возвращается новое смещение offs + value\_len, указывающее на следующую доступную позицию для записи в массиве bit\_arr.

Таким образом, эта функция позволяет установить определенное количество битов в массиве байтов, начиная с указанного смещения. Сложности алгоритма линейная и зависит только от количества бит, которые нужно записать.

|  |
| --- |
| Листинг 5. Псевдокод функции getbit |
| getbit(bit\_arr, offs)  byte\_n = offs / 8  byte\_offs = offs % 8  mask = 0x80 >> byte\_offs  if bit\_arr[byte\_n] & mask == 0 then  return 0  else  return 1 |

Эта функция позволяет получить значение бита из массива байтов bit\_arr по указанному смещению offs. Она выполняет следующие шаги:

1. Вычисляется номер байта (byte\_n), из которого будет происходить чтение, путем деления смещения offs на 8.
2. Вычисляется смещение внутри байта (byte\_offs), показывающее, с какого бита нужно получить значение, путем вычисления остатка от деления offs на 8.
3. Создается маска (mask), которая будет использоваться для извлечения нужного бита из байта.
4. С помощью операции побитового И (&) между байтом bit\_arr[byte\_n] и маской mask проверяется значение нужного бита.
5. Если результат равен 0, возвращается 0, что означает, что бит равен 0.
6. Иначе возвращается 1, что означает, что бит равен 1.

Таким образом, эта функция позволяет получить значение определенного бита из массива байтов по указанному смещению. Сложность алгоритма константная.

|  |
| --- |
| Листинг 6. Псевдокод функции traverse\_tree |
| traverse\_tree(codes, tree, code, len)  if tree->left != NULL then  tmp\_len = len + 1  tmp\_code = code << 1  traverse\_tree(codes, tree->left, tmp\_code, tmp\_len)  if tree->right != NULL then  tmp\_len = len + 1  tmp\_code = (code << 1) | 1  traverse\_tree(codes, tree->right, tmp\_code, tmp\_len)  if tree->left = NULL and tree->right = NULL then  codes[tree->symbol].code = code  codes[tree->symbol].len = len  return a |

Функция обхода дерева в глубину нужна для того, чтобы сформировать таблицу кодов символов, и вот как она работает:

1. Если у узла tree есть левый потомок (tree->left), то происходит рекурсивный вызов функции traverse\_tree для этого потомка. При этом длина кода увеличивается на 1 (tmp\_len = len + 1), а код tmp\_code сдвигается влево на 1 (tmp\_code = tmp\_code << 1), то есть к нему добавляется 0.
2. Если у узла tree есть правый потомок (tree->right), то происходит рекурсивный вызов функции traverse\_tree для этого потомка. При этом длина кода увеличивается на 1 (tmp\_len = len + 1), а код сдвигается влево на 1 и его младший бит обращается в 1.
3. Если у узла tree нет ни левого, ни правого потомка (tree->left = NULL and tree->right = NULL), значит мы достигли листа дерева. Тогда код и его длина записываются в массив codes для символа, представленного этим листом.
4. Функция возвращает массив codes, который теперь содержит заполненные значения кодов и их длин для символов в дереве.

Таким образом, функция traverse\_tree выполняет рекурсивный обход бинарного дерева и строит коды для символов в листьях этого дерева, сохраняя их в массиве структур Codes.

Для того, чтобы можно было распаковать сжатый файл, необходимо записать в него служебную информацию. В данной программе в качестве служебной информации используются длины кодов символов, коды символов, количество символов в изначальном файле, количество последних бит, которые не были задействованы, и длина битового массива, который был записан в сжатый файл. Запись длин кодов и самих кодов символов происходит по схеме «длина – код». Длины кодов символов различаются и могут занимать от 0 до 16 бит. Если длина какого-либо кода составляет 0 бит, то его запись в файл не требуется и не происходит. Если длина кода составляет не более 8 бит, то для его записи достаточно одного байта, и это тоже учитывается. Таким образом, происходит динамическая запись служебной информации в файл, размер которой может составлять от 143 до 395 байт.

|  |
| --- |
| Листинг 7. Псевдокод функции write\_comp\_file |
| write\_comp\_file(file, bit\_arr, codes, offs, size\_bit\_arr, message\_size)  for i = 1 to 126 do  write(codes[i].len, 1)  if codes[i].len = 0 then  continue  if codes[i].len < 9 then  write(codes[i].code, 1)  continue  write(codes[i].code, 2)  n\_last\_bits = offs % 8  write(message\_size, 8)  write(n\_last\_bits, 1)  write(size\_bit\_arr, 8)  write(bit\_arr, size\_bit\_arr) |

Считывание служебной информации из файла происходит аналогичным образом.

Анализ сложности сжатия и распаковки

Код функций на языке Си представлен в приложении к отчёту.

Прежде, чем выделять память для битового массива, необходимо определить его будущий размер. Для этого требуется выполнить n операций, где n – количество кодируемых символов. После выделения памяти требуется ещё m операций (m – длина битового массива) для того, чтобы инициализировать массив нулями. После этого начинается запись массива. Для того, чтобы записать биты в массив требуется n \* g операций, где g – средняя длина всех кодов символов. Таким образом, сложность формирования битового массива по времени составляет:

Процесс декодирования имеет гораздо большую сложность в силу особенностей реализации. После выделения памяти для message требуется m (количество символов) операций для того, чтобы инициализировать его. После этого происходит копирование кодов во временный массив, которое требует 126 итераций цикла, за которым следует цикл из n итераций, где n – количество бит в битовом массиве. На каждой итерации происходит процесс очищения несовпадающих кодов, требующий ещё 126 операций, за которым следуют, в худшем случае, ещё 126 итераций, для поиска подходящего кода символа в массиве кодов. Но для среднего случая можно предположить, что средним значением будет 63. Примерно каждые g итераций цикла обхода битового массива, где g – средняя длина кода, будет обнаруживаться совпадающий код. В таком случае, нужно будет обновлять таблицу кодов, выполняя 126 шагов цикла копирования. Если сосчитать все факторы, то получится следующее:

Данный анализ показывает, что распаковка сжатого файла занимает гораздо больше времени, чем его сжатие. Этот факт также подтверждается и практикой использования программы.

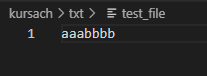
# Подсчёт коэффициента сжатия

При тестировании программы на файлах разных объёмов наблюдается рост эффективности сжатия при увеличении размера файла.

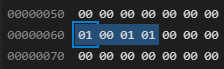
Рисунок 8. Анализ эффективности сжатия с увеличением размера файла

Работа программы

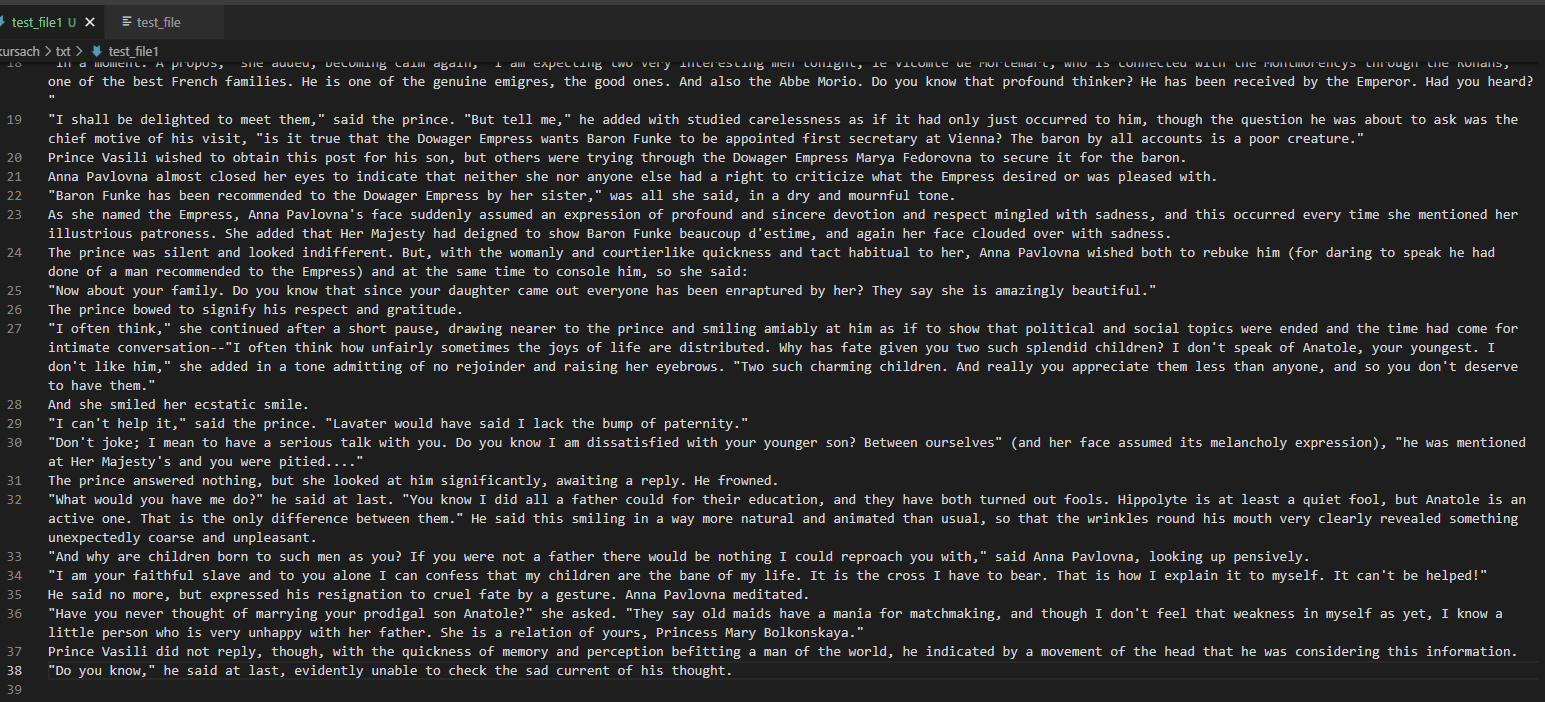
Для начала, тест работы программы для наиболее простого файла, состоящего из нескольких букв.

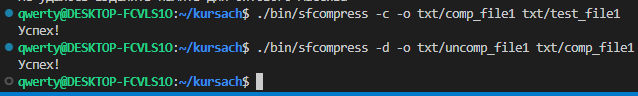


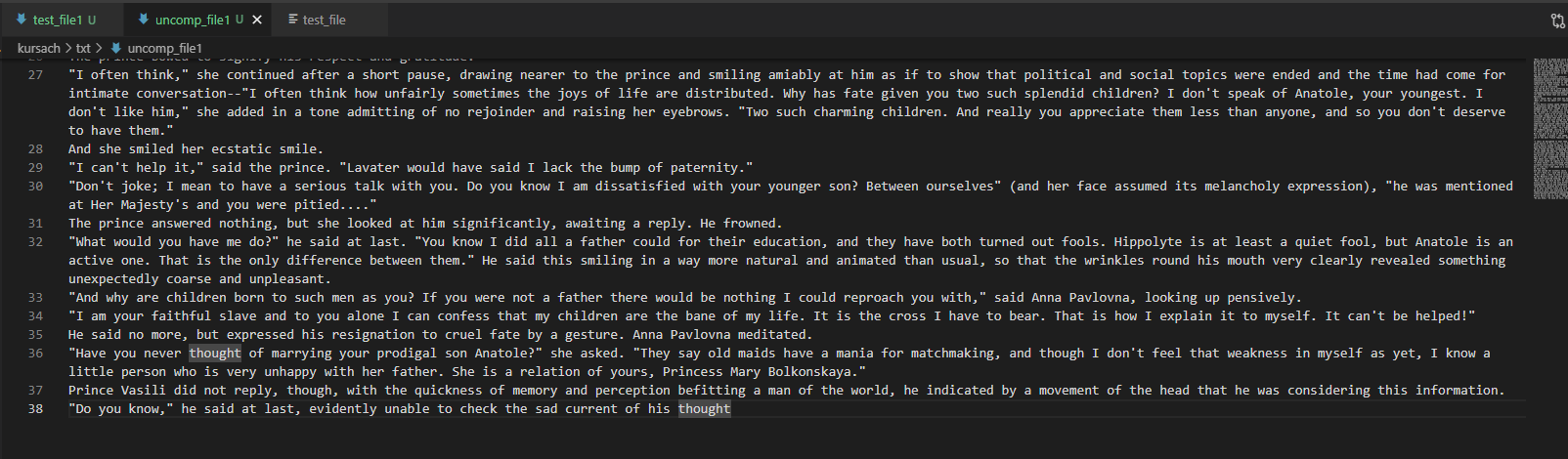
При сжатии данного файла видно, что в последнем байте сжатого файла 3 раза был записан код буквы ‘а’, равный нулю и длиной в 1 бит, и 4 раза код буквы ‘b’, равный 1 и длиной так же в 1 бит.

Теперь более сложный тест. Был подготовлен файл, в котором первая глава романа Л. Н. Толстого «Война и мир» на английском языке.

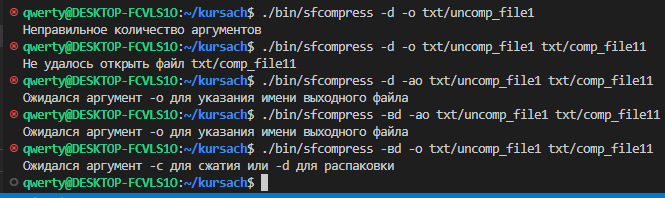






При кодировании и последующем декодировании файла видно, что оба файла полностью совпадает, что говорит о том, что программа успешно выполняет свою работу.

Помимо этого, в программе предусмотрена обработка ошибок при вводе.



А также обработка битых файлов.



Приложение

Приложение 1 “bit\_arr.c”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | #include <codes.h>  #include <stdint.h>  #include <stdio.h>  **uint64\_t**  **setbits**(**uint8\_t**\* bit\_arr, **uint64\_t** offs, **uint16\_t** value, **uint8\_t** value\_len)  {  **uint64\_t** byte\_n = offs / **8**;  **uint8\_t** byte\_offs = offs % **8**;  value = value << (**16** - value\_len);  **for** (**int** i = value\_len; i > **0**; i--) {  bit\_arr[byte\_n]  = (bit\_arr[byte\_n] | ((value & **0x8000**) >> (byte\_offs + **8**)));  byte\_offs++;  **if** (byte\_offs % **8** == **0**) {  byte\_n++;  byte\_offs = **0**;  }  value = value << **1**;  }  **return** offs + value\_len;  }  **uint8\_t** **getbit**(**uint8\_t**\* bit\_arr, **uint64\_t** offs)  {  **uint64\_t** byte\_n  = offs / **8**;  **uint8\_t** byte\_offs = offs % **8**;  **uint8\_t** mask = **0x80** >> byte\_offs;  **if** ((bit\_arr[byte\_n] & mask) == **0**)  **return** **0**;  **else**  **return** **1**;  }  **size\_t** **bit\_arr\_size**(Codes\* a, **char**\* message)  {  **size\_t** count = **0**;  **for** (**int** i = **0**; message[i] != '\0'; i++)  count += a[(**uint8\_t**)message[i]].len;  **size\_t** size = count / **8**;  **if** (count % **8** > **0**)  size++;  **return** size;  }  **void** **init\_bit\_arr**(**uint8\_t**\* bit\_arr, **size\_t** size)  {  **for** (**size\_t** i = **0**; i < size; i++)  bit\_arr[i] = **0**;  } |

Приложение 2 “bit\_arr.h”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #pragma once  #include <codes.h>  #include <stdint.h>  #include <stdio.h>  **uint64\_t**  **setbits**(**uint8\_t**\* bit\_arr, **uint64\_t** offs, **uint16\_t** value, **uint8\_t** value\_len);  **uint8\_t** **getbit**(**uint8\_t**\* bit\_arr, **uint64\_t** offs);  **size\_t** **bit\_arr\_size**(Codes\* a, **char**\* message);  **void** **init\_bit\_arr**(**uint8\_t**\* bit\_arr, **size\_t** size); |

Приложение 3 “codes.c”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | #include <codes.h>  Codes\* **copy\_codes**(Codes\* dest, Codes\* src)  {  **for** (**int** i = **1**; i < **127**; i++) {  dest[i].code = src[i].code;  dest[i].len = src[i].len;  }  **return** dest;  }  Codes\* **clear**(Codes\* a, **uint16\_t** value, **uint8\_t** value\_len)  {  **uint16\_t** tmp\_code;  **for** (**int** i = **1**; i < **127**; i++) {  **if** (a[i].len == **0**)  **continue**;  tmp\_code = a[i].code >> (a[i].len - value\_len);  **if** (tmp\_code != value)  a[i].len = **0**;  }  **return** a;  }  **\_Bool** **find\_code**(Codes\* tmp, **uint16\_t** value, **uint8\_t** value\_len, **uint8\_t** s)  {  **if** (tmp[s].code == value && tmp[s].len == value\_len)  **return** **1**;  **return** **0**;  }  **\_Bool** **empty\_codes**(Codes\* tmp)  {  **for** (**int** i = **1**; i < **127**; i++)  **if** (tmp[i].len != **0**)  **return** **0**;  **return** **1**;  } |

Приложение 4 “codes.h”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #pragma once  #include <stdint.h>  **typedef** **struct** codes {  **uint16\_t** code;  **uint8\_t** len;  } Codes;  Codes\* **copy\_codes**(Codes\* dest, Codes\* src);  Codes\* **clear**(Codes\* a, **uint16\_t** value, **uint8\_t** value\_len);  **\_Bool** **find\_code**(Codes\* tmp, **uint16\_t** value, **uint8\_t** value\_len, **uint8\_t** s);  **\_Bool** **empty\_codes**(Codes\* tmp); |

Приложение 5 “coding.c”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64 | #include <codes.h>  #include <bit\_arr.h>  #include <stdlib.h>  #include <stdint.h>  #include <stdio.h>  #include <string.h>  **uint8\_t**\*  **ENCODE\_MSG**(**char**\* message, Codes\* a, **uint64\_t**\* offs, **size\_t**\* size\_bit\_arr)  {  **uint64\_t** tmp\_offs = **0**;  **size\_t** size = bit\_arr\_size(a, message);  **uint8\_t**\* bit\_arr = malloc(size);  init\_bit\_arr(bit\_arr, size);  **uint16\_t** value;  **uint8\_t** value\_len;  **for** (**int** i = **0**; message[i] != '\0'; i++) {  value = a[(**uint8\_t**)message[i]].code;  value\_len = a[(**uint8\_t**)message[i]].len;  tmp\_offs = setbits(bit\_arr, tmp\_offs, value, value\_len);  }  \*offs = tmp\_offs;  \*size\_bit\_arr = size;  **return** bit\_arr;  }  **char**\* **DECODE\_MSG**(  **uint8\_t**\* bit\_arr,  Codes\* a,  **uint8\_t** n\_last\_bits,  **size\_t** size\_bit\_arr,  **size\_t** message\_size)  {  **char** letter[**2**] = {**0**};  **char**\* message = malloc(message\_size + **1**);  **for**(**size\_t** i = **0**; i < message\_size + **1**; i++)  message[i] = **0**;  **size\_t** n = size\_bit\_arr - **1**;  Codes tmp[**127**];  copy\_codes(tmp, a);  **uint16\_t** value = **0**;  **uint8\_t** value\_len = **0**;  **for** (**size\_t** i = **0**; i < (**8** \* n + n\_last\_bits); i++) {  value = (value << **1**) | getbit(bit\_arr, i);  value\_len++;  clear(tmp, value, value\_len);  **for** (**uint8\_t** s = **1**; s < **127**; s++) {  **if** (find\_code(tmp, value, value\_len, s)) {  letter[**0**] = s;  message = strcat(message, letter);  copy\_codes(tmp, a);  value = **0**;  value\_len = **0**;  **break**;  }  }  **if** (empty\_codes(tmp)) {  printf("Ошибка декодирования, такого кода не существует!**\n**");  exit(-**1**);  }  }  **return** message;  } |

Приложение 6 “coding.h”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #pragma once  **uint8\_t**\*  **ENCODE\_MSG**(**char**\* message, Codes\* a, **uint64\_t**\* offs, **size\_t**\* size\_bit\_arr);  **char**\* **DECODE\_MSG**(  **uint8\_t**\* bit\_arr,  Codes\* a,  **uint8\_t** n\_last\_bits,  **size\_t** size\_bit\_arr,  **size\_t** message\_size); |

Приложение 7 “input\_proc.c”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170 | #include <codes.h>  #include <minheap.h>  #include <tree.h>  #include <coding.h>  #include <bit\_arr.h>  #include <stdint.h>  #include <stdio.h>  #include <sys/stat.h>  #include <sys/types.h>  **off\_t** **fsize**(**const** **char**\* filename)  {  **struct** stat st;  **if** (stat(filename, &st) == **0**)  **return** st.st\_size;  **return** -**1**;  }  **uint64\_t**\* **get\_freq**(**uint64\_t**\* symbols, **char**\* message, **off\_t** message\_size)  {  **for** (**int** i = **0**; i < message\_size + **1**; i++)  symbols[(**uint8\_t**)message[i]]++;  **return** symbols;  }  **void** **write\_comp\_file**(  **FILE**\* file,  **uint8\_t**\* bit\_arr,  Codes\* a,  **uint64\_t** offs,  **size\_t** size\_bit\_arr,  **off\_t** message\_size)  {  **uint8\_t** buf;  **for** (**int** i = **1**; i < **127**; i++) {  fwrite(&a[i].len, **1**, **1**, file);  **if** (a[i].len == **0**)  **continue**;  **if** (a[i].len < **9**) {  buf = (**uint8\_t**)a[i].code;  fwrite(&buf, **1**, **1**, file);  **continue**;  }  fwrite(&a[i].code, **2**, **1**, file);  }  **uint8\_t** n\_last\_bits = offs % **8**;  fwrite(&message\_size, **sizeof**(**off\_t**), **1**, file);  fwrite(&n\_last\_bits, **1**, **1**, file);  fwrite(&size\_bit\_arr, **sizeof**(**size\_t**), **1**, file);  fwrite(bit\_arr, **1**, size\_bit\_arr, file);  }  Codes\* **get\_codes**(Codes\* a, **char**\* message, **off\_t** message\_size)  {  **uint64\_t** letters\_freq[**127**] = {**0**};  get\_freq(letters\_freq, message, message\_size);  Heap\* h = NULL;  h = heap\_create(h);  Node\* h\_tree = HTREE(letters\_freq, h);  traverse\_tree(a, h\_tree, **0**, **0**);  heap\_free(h);  free\_tree(h\_tree);  **return** a;  }  **void** **compress\_file**(**char**\* input\_file\_name, **char**\* output\_file\_name)  {  **FILE**\* input\_file = fopen(input\_file\_name, "rb");  **if** (input\_file == NULL) {  printf("Не удалось открыть файл %s**\n**", input\_file\_name);  exit(-**1**);  }  **off\_t** input\_file\_size = fsize(input\_file\_name);  **if** (input\_file\_size == -**1**) {  printf("Не удалось определить размер файла %s**\n**", input\_file\_name);  fclose(input\_file);  }  **FILE**\* output\_file = fopen(output\_file\_name, "wb");  **if** (output\_file == NULL) {  printf("Не удалось открыть файл %s**\n**", output\_file\_name);  fclose(input\_file);  exit(-**1**);  }  **char**\* message = malloc(**sizeof**(**char**) \* input\_file\_size + **1**);  **if** (message == NULL) {  printf("Не удалось выделить память для считывания файла %s**\n**",  input\_file\_name);  exit(-**1**);  }  **for** (**int** i = **0**; i < input\_file\_size + **1**; i++)  message[i] = **0**;  **if** (fread(message, **1**, input\_file\_size, input\_file)  != (**size\_t**)input\_file\_size) {  printf("При чтении файла %s произошла ошибка**\n**", input\_file\_name);  fclose(input\_file);  fclose(output\_file);  free(message);  }  fclose(input\_file);  Codes a[**127**] = {**0**};  get\_codes(a, message, input\_file\_size);  **uint64\_t** offs;  **size\_t** size\_bit\_arr;  **uint8\_t**\* bit\_arr = ENCODE\_MSG(message, a, &offs, &size\_bit\_arr);  free(message);  write\_comp\_file(  output\_file, bit\_arr, a, offs, size\_bit\_arr, input\_file\_size);  free(bit\_arr);  fclose(output\_file);  }  **void** **decompress\_file**(**char**\* input\_file\_name, **char**\* output\_file\_name)  {  **FILE**\* input\_file = fopen(input\_file\_name, "rb");  **if** (input\_file == NULL) {  printf("Не удалось открыть файл %s**\n**", input\_file\_name);  exit(-**1**);  }  **FILE**\* output\_file = fopen(output\_file\_name, "w");  **if** (output\_file == NULL) {  printf("Не удалось открыть файл %s**\n**", output\_file\_name);  fclose(input\_file);  exit(-**1**);  }  Codes a[**127**] = {**0**};  **uint8\_t** buf;  **for** (**int** i = **1**; i < **127**; i++) {  fread(&a[i].len, **1**, **1**, input\_file);  **if** (a[i].len == **0**)  **continue**;  **if** (a[i].len < **9**) {  fread(&buf, **1**, **1**, input\_file);  a[i].code = (**uint16\_t**)buf;  **continue**;  }  fread(&a[i].code, **2**, **1**, input\_file);  }  **size\_t** message\_size;  **uint8\_t** n\_last\_bits;  **size\_t** size\_bit\_arr;  fread(&message\_size, **sizeof**(**size\_t**), **1**, input\_file);  fread(&n\_last\_bits, **1**, **1**, input\_file);  fread(&size\_bit\_arr, **sizeof**(**size\_t**), **1**, input\_file);  **uint8\_t**\* bit\_arr = malloc(size\_bit\_arr);  **if** (bit\_arr == NULL) {  printf("Не удалось выделить память для битового массива**\n**");  fclose(input\_file);  fclose(output\_file);  exit(-**1**);  }  init\_bit\_arr(bit\_arr, size\_bit\_arr);  **if** (fread(bit\_arr, **1**, size\_bit\_arr, input\_file) != size\_bit\_arr) {  printf("Не удалось считать файл %s**\n**", input\_file\_name);  fclose(input\_file);  fclose(output\_file);  free(bit\_arr);  exit(-**1**);  }  fclose(input\_file);  **char**\* message  = DECODE\_MSG(bit\_arr, a, n\_last\_bits, size\_bit\_arr, message\_size);  free(bit\_arr);  fwrite(message, **1**, message\_size, output\_file);  free(message);  fclose(output\_file);  } |

Приложение 8 “input\_proc.h”

|  |  |
| --- | --- |
| 1 2 | **void** **compress\_file**(**char**\* input\_file\_name, **char**\* output\_file\_name);  **void** **decompress\_file**(**char**\* input\_file\_name, **char**\* output\_file\_name); |

Приложение 9 “main.c”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | #include <bit\_arr.h>  #include <codes.h>  #include <coding.h>  #include <minheap.h>  #include <output.h>  #include <tree.h>  #include <input\_proc.h>  #include <stdio.h>  #include <string.h>  **int** **main**(**int** argc, **char**\* argv[])  {  **if** (argc != **5**) {  printf("Неправильное количество аргументов**\n**");  **return** -**1**;  }  **if** (strcmp(argv[**2**], "-o") != **0**) {  printf("Ожидался аргумент -o для указания имени выходного файла**\n**");  **return** -**1**;  }  **if** ((strcmp(argv[**1**], "-c") != **0**) && (strcmp(argv[**1**], "-d") != **0**)) {  printf("Ожидался аргумент -с для сжатия или -d для распаковки**\n**");  **return** -**1**;  }  **if** (strcmp(argv[**1**], "-c") == **0**) {  compress\_file(argv[**4**], argv[**3**]);  }  **if** (strcmp(argv[**1**], "-d") == **0**) {  decompress\_file(argv[**4**], argv[**3**]);  }  printf("Успех!**\n**");  **return** **0**;  } |

Приложение 10 “minheap.c”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88 | #include "./minheap.h"  #include <stdint.h>  #include <stdio.h>  #include <stdlib.h>  Heap\* **heap\_create**()  {  Heap\* h;  h = malloc(**sizeof**(\*h));  **if** (h != NULL) {  h->nnodes = **0**;  **if** (h->nodes == NULL) {  free(h);  **return** NULL;  }  }  **return** h;  }  **uint8\_t** **heap\_nnodes**(Heap\* h)  {  **return** h->nnodes;  }  **void** **heap\_free**(Heap\* h)  {  free(h);  }  **void** **heap\_swap**(**struct** heapnode\* a, **struct** heapnode\* b)  {  **struct** heapnode tmp = \*a;  \*a = \*b;  \*b = tmp;  }  **int** **heap\_insert**(Heap\* h, **uint64\_t** freq, **uint8\_t** symbol, Node\* left, Node\* right)  {  h->nnodes++;  h->nodes[h->nnodes].freq = freq;  h->nodes[h->nnodes].symbol = symbol;  h->nodes[h->nnodes].left = left;  h->nodes[h->nnodes].right = right;  **for** (**int** i = h->nnodes;  (i > **1**) && (h->nodes[i].freq < h->nodes[i / **2**].freq);  i = i / **2**)  heap\_swap(&h->nodes[i], &h->nodes[i / **2**]);  **return** **0**;  }  **void** **heap\_heapify**(Heap\* h, **int** index)  {  **while** (**1**) {  **int** left = **2** \* index;  **int** right = **2** \* index + **1**;  **int** minimal = index;  **if** ((left <= h->nnodes)  && (h->nodes[left].freq < h->nodes[minimal].freq))  minimal = left;  **if** ((right <= h->nnodes)  && (h->nodes[right].freq < h->nodes[minimal].freq))  minimal = right;  **if** (minimal == index)  **break**;  heap\_swap(&h->nodes[index], &h->nodes[minimal]);  index = minimal;  }  }  **struct** heapnode **heap\_extract\_min**(Heap\* h)  {  **if** (h->nnodes == **0**)  **return** (**struct** heapnode){**0**, **0**, NULL, NULL};  **struct** heapnode minnode = h->nodes[**1**];  h->nodes[**1**] = h->nodes[h->nnodes--];  heap\_heapify(h, **1**);  **return** minnode;  }  Heap\* **init\_heap**(**uint64\_t**\* symbols, Heap\* h)  {  **for** (**int** i = **1**; i < **127**; i++)  heap\_insert(h, symbols[i], i, NULL, NULL);  **return** h;  } |

Приложение 11 “minheap.h”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | #pragma once  #include <stdio.h>  #include <stdlib.h>  #include <stdint.h>  **typedef** **struct** heapnode {  **uint64\_t** freq;  **uint8\_t** symbol;  **struct** heapnode\* left;  **struct** heapnode\* right;  } Node;  **typedef** **struct** heap {  **uint8\_t** nnodes;  **struct** heapnode nodes[**127**];  } Heap;  Heap\* **heap\_create**();  **uint8\_t** **heap\_nnodes**(Heap\* h);  **void** **heap\_free**(Heap\* h);  **int** **heap\_insert**(Heap\* h, **uint64\_t** freq, **uint8\_t** symbol, Node\* left, Node\* right);  **struct** heapnode **heap\_extract\_min**(Heap\* h);  Heap\* **init\_heap**(**uint64\_t**\* symbols, Heap\* h); |

Приложение 12 “makefile”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | APP\_NAME = sfcompress  CFLAGS = -Wall -Wextra -Werror  CPPFLAGS = -I src -MMD  CC = gcc  SRC\_DIR = src  OBJ\_DIR = obj  BIN\_DIR = bin  APP\_PATH = **$(**BIN\_DIR**)**/**$(**APP\_NAME**)**  SRC\_EXT = c  APP\_SOURCES = **$(**wildcard **$(**SRC\_DIR**)**/\*.**$(**SRC\_EXT**))**  APP\_OBJECTS := **$(**patsubst **$(**SRC\_DIR**)**/%.**$(**SRC\_EXT**)**,**$(**OBJ\_DIR**)**/%.o,**$(**APP\_SOURCES**))**  DEPS = **$(**APP\_OBJECTS:.o=.d**)**  **.PHONY**: **all**  **all**: **$(APP\_PATH)**  -include $(DEPS)  **$(**APP\_PATH**)**: **$(**APP\_OBJECTS**)**  **$(**CC**)** **$(**CFLAGS**)** **$(**CPPFLAGS**)** $^ -o $@  **$(**OBJ\_DIR**)**/%.o: **$(**SRC\_DIR**)**/%.**$(**SRC\_EXT**)**  **$(**CC**)** -c **$(**CFLAGS**)** **$(**CPPFLAGS**)** $< -o $@  .PHONY: clean  clean:  rm -rf **$(**APP\_OBJECTS**)**  rm -rf **$(**DEPS**)**  rm -rf **$(**APP\_PATH**)** |