

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА
по дисциплине «Алгоритмы и Структуры Данных»
Тема: Динамическое кодирование и декодирование по Хаффману

Студент гр. 8304

Холковский К.В.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2019

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Холковский Константин Владимирович

Группа 8304

Тема работы: кодирование и декодирование

Исходные данные:

Написать программу для кодирования и декодирования некоторого текста адаптивным методом Хаффмана и продемонстрировать его работу, файловое считывание и файловая запись.

Содержание пояснительной записки:

- Содержание
- Введение
- Кодирование
- Декодирование
- Тестирование
- Исходный код
- Использованные источники

Дата выдачи задания: 11.10.2019

Дата сдачи реферата: 17.10.2019

Дата защиты реферата: 17.10.2019

Студент

Холковский К.В.

Преподаватель

Фирсов М.А.

АННОТАЦИЯ

В данной работе была создана программа на языке программирования C++, которая сочетает в себе несколько функций: кодирования/декодирования текста. Были использованы преимущества C++ для минимизации кода. Для лучшего понимания кода было в нем приведено большое кол-во отладочных выводов. Также была проведена его оптимизация с целью экономии выделяемой в процессе работы памяти и улучшения быстродействия программы.

SUMMARY

In this work, a program was created in the C ++ programming language, which combines several functions: encoding / decoding text. The benefits of C ++ were used to minimize code. For a better understanding of the code, it contained a large number of debugging outputs. Also, its optimization was carried out in order to save the memory allocated in the process of working and improve the speed of the program.

СОДЕРЖАНИЕ

Введение	5
1. Адаптивный алгоритм Хаффмана	6
Кодирование	7
Декодирование.....	8
2. Функции и структуры данных.....	10
3. Интерфейс программы	12
4. Тестирование	12

Введение

Целью данной курсовой работы является реализация адаптивного алгоритма кодирования/декодирования текста методом Хаффмана и демонстрация работы алгоритма. Алгоритм использует коды переменной длины: часто встречающийся символ кодируется кодом меньшей длины, редко встречающийся — кодом большей длины. Адаптивное кодирование Хаффмана позволяет строить кодовую схему в поточном режиме (без предварительного сканирования данных), не имея никаких начальных знаний из исходного распределения, что позволяет за один проход сжать данные. Преимуществом этого способа является возможность кодировать на лету. Данный метод сжатия имеет большое сходство с Фано-Шеннона, который появился на несколько лет раньше и является логическим продолжением алгоритма Шеннона. Кодирование Хаффмана широко применяется при сжатии данных, в том числе при сжатии фото- и видеоизображений (JPEG, MPEG), в популярных архиваторах (PKZIP, LZH и др.), в протоколах передачи данных HTTP (Deflate), MNP5 и MNP7 и других.

1. Адаптивный алгоритм Хаффмана

Адаптивное сжатие по Хаффману

Обеспечивает избыточность, сравнимую с алгоритмом сжатия с использованием динамических таблиц, и позволяет обойтись без передачи частот символов вместе со сжатым сообщением, т. е. частоты символов кодера и декодера автоматически подстраиваются по мере поступления символов или сжатых кодов. Алгоритм широко применяется при сжатии сообщений с заранее неизвестной длиной, в том числе длинных серий коротких сообщений, и во многом по сложности программной/аппаратной реализации и требованиям к производительности аппаратуры соответствует алгоритму сжатия с использованием динамических таблиц.

Статическое сжатие по Хаффману

Часто даст наибольшую избыточность, т. к. при сжатии сообщения используются усредненные частоты (по ним один раз строятся соответствующие дерево кодов и таблицы), полученные на репрезентативной выборке сходных по содержанию данных, и величина этой избыточности во многом будет зависеть от качества выборки. Данный способ широко и эффективно применяется при сжатии изображений, например в стандарте JPEG, и снижает сложность программной/аппаратной реализации алгоритма в целом, позволяет значительно снизить требования к производительности процессора. Исходя из вышесказанного, процедура сжатия будет состоять из одного шага - формирование сжатого сообщения на основании статических табличных кодов. Процедура распаковки также состоит из одного шага - формирование несжатого сообщения на основании статических табличных кодов.

Обобщив все вышесказанное, можно составить следующую таблицу (см табл.1)

Таблица 1 - Сравнение характеристик реализаций алгоритма Хаффмана

Тип алгоритма	Избыточность	Эффективность	Потоковое сжатие	Сложность реализации	Вычислительная сложность
Статический	Высокая	Низкая	Нет	Низкая	Низкая
Динамический	Средняя	Высокая	Да	Высокая	Высокая

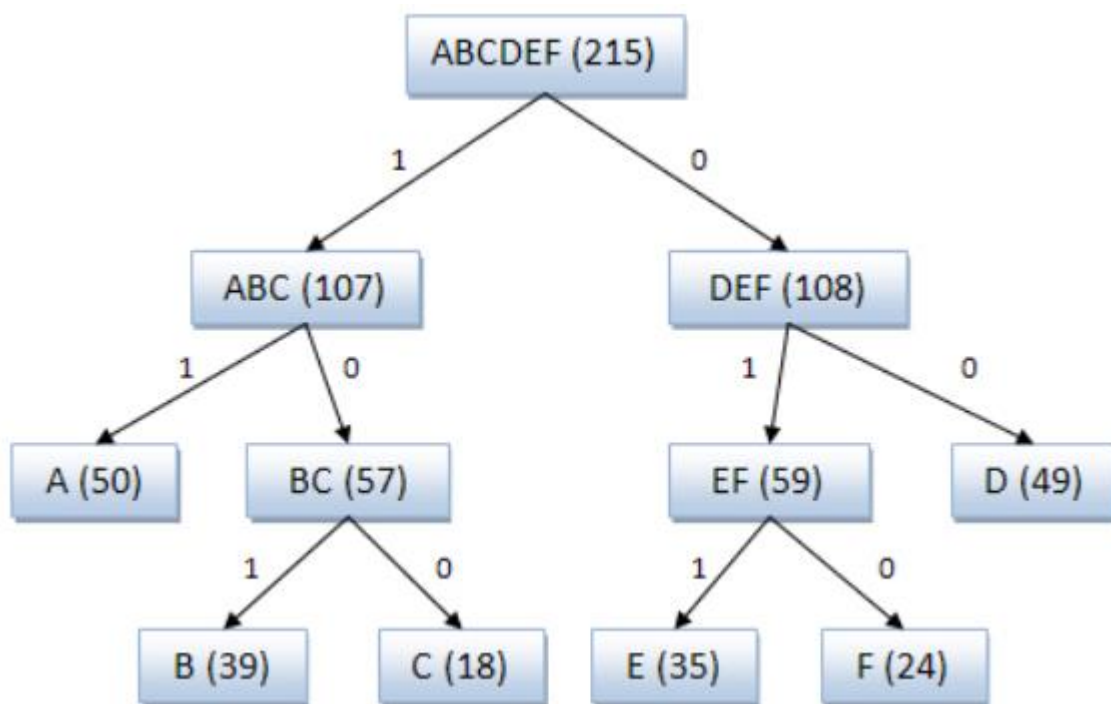


Рисунок 1 – Пример работы алгоритма

1.1. Кодирование

Для решения поставленной подзадачи было реализовано бинарное дерево для хранения информации о символах из входного текста. В зависимости от того, есть или нет некоторый символ в дереве, алгоритм ведет себя по-разному. Если символ уже имелся в дереве, то вызывается функция перестройки дерева, для данного символа, которая отвечает сразу за две задачи, увеличения кол-ва вхождений нужных символов и за то, чтобы дерево было упорядоченным. Если же символа нет в дереве, то он добавляется и вызывается функция перестройки дерева для этого символа. Разберем на примере кодирования текста: “baa” (см. рис 1)

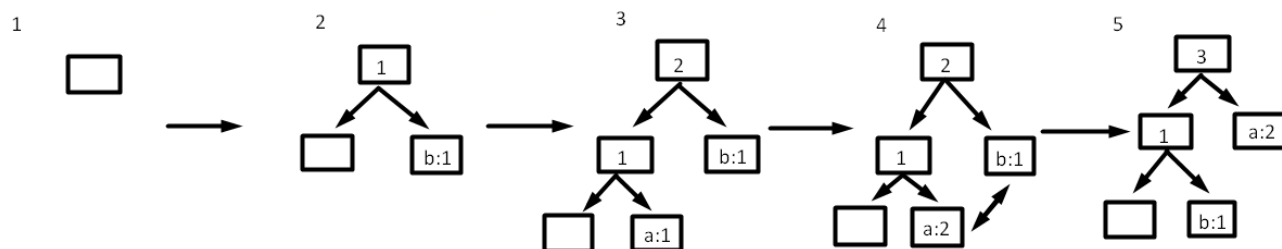


Рисунок 2 – Описание работы алгоритма

Состояние 1:

Имеем пустое дерево, считав первый символ(b) и не найдя его в дереве, добавляем и переходим к состоянию 2.(выводится b)

Состояние 2:

Считав 2-й символ, и не найдя его в дереве, добавляем и переходим к состоянию 3.(выводится 0a)

Состояние 3:

Считав 3-й символ, и найдя его в дереве, переходим к состоянию 4.(выводится 01)

Состояние 4:

Упорядочим наше дерево поменяв местами два, указанных на рисунке 1, элемента.

Состояние 5:

Итоговое состояние дерева.

Результат: b0a01(Демонстрацию, генерируемую программой см. рис 2)


```

Начальное состояние дерева
0

Следующего символа 'b' нет в дереве => добавляем элемент (Выводим: b)
Состояние дерева после добавления:
    'b':0
0
    0
Состояние дерева после перестройки:
    'b':1
1
    0

Следующего символа 'a' нет в дереве => добавляем элемент (Выводим: 0a)
Состояние дерева после добавления:
    'b':1
1
    'a':0
    0
    0
Состояние дерева после перестройки:
    'b':1
2
    'a':1
    1
    0

Следующий символ 'a' есть в дереве (Выводим: 01)
Состояние дерева после перестройки:
    'a':2
3
    'b':1
    1
    0

```

Рисунок 3 – Демонстрация работы кодирования

1.2. Декодирование

Идя по пути, записанному в файле и найдя конечный лист дерева, выполняем следующее, если этот лист пустой, то считываем из входного файла информацию о символе, добавляем его в дерево и вызываем функцию перестройки дерева для данного символа, иначе, встретив лист с некоторым символом, вызываем функцию перестройки дерева для этого символа.

Разберем на примере декодирования: “b0a01” (см. рис 2)

Состояние 1:

Идя по пути(пустой путь) находим пустой лист, добавляем в дерево символ b(выводится b)

Состояние 2:

Идя по пути 0 находим пустой лист, добавляем символ а в дерево (выводится а)

Состояние 3:

Идя по пути 01 находим лист с символом а (выводится а)

Состояние 4:

Упорядочим наше дерево поменяв местами два, указанных на рисунке 2, элемента.

Состояние 5:

Итоговое состояние дерева.

Результат: баа(Демонстрацию, генерируемую программой см. рис 4)

```
Начальное состояние дерева
0

Идя по пути из закодированного файла встретили пустой лист => добавляем в дерево (Выводим: 'b')
Состояние дерева после добавления:
  'b':0
0
  0
Состояние дерева после перестройки:
  'b':1
1
  0

Идя по пути из закодированного файла встретили пустой лист => добавляем в дерево (Выводим: 'a')
Состояние дерева после добавления:
  'b':1
1
  0      'a':0
        0
Состояние дерева после перестройки:
  'b':1
2
  1      'a':1
        0

Идя по пути из закодированного файла встретили символ (Выводим: 'a')
Состояние дерева после перестройки
  'a':2
3
  1      'b':1
        0
```

Рисунок 4 – Демонстрация работы декодирования

2. ФУНКЦИИ И СТРУКТУРЫ ДАННЫХ

1 Структуры:

```
struct kanoha {
    kanoha();
};
```

```

        kanoha(char b, int par);
    char symbol;
    size_t count;
    int parent;
    int left;
    int right;
};

```

Структура `kanoha` представляет узел дерева, где `left` – номер на левой ветки, `right` – правой в `vector` структуры `bin_tree` (см ниже), `symbol` – символ хранящийся в структуре, `count` – кол-во данных символов в дереве, `parent` – номер родителя в `vector`.

```

class bin_tree {
    struct kanoha {
        kanoha();
        kanoha(char b, int par);
        char symbol;
        size_t count;
        int parent;
        int left;
        int right;
    };

    std::vector<kanoha> arr;
    void swap(int one, int second);

public:

    bin_tree();
    kanoha get(size_t i);
    size_t size();
    size_t find(const char a);
    void add(const char a);
    void print(int index, int deep, std::ofstream& dem);
    void rebuild(const char a);

};

```

Структура `bin_tree` нужна для хранения дерева Хаффмана, где `arr` – хранит узлы; `kanoha get(size_t i)` - нужен для получения копии элемента из `arr`; `size_t size()`

- нужен для получения размера arr; `size_t find(const char a)` - нужен для получения номера в arr для данного a, если элемента нет в arr то возвращает номер пустого элемента; `void add(const char a)` – добавляет на место пустого элемента a; `void print(int index, int deep, std::ofstream& dem)` – нужна для вывода дерева в dem; `void rebuild(const char a)` – нужна для перестройки дерева для данного a.

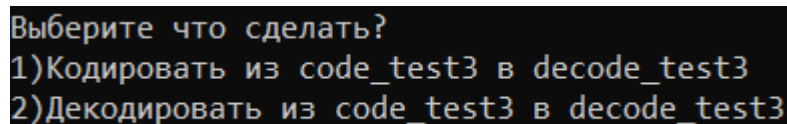
2 Функции:

`void code(std::ifstream& in, std::ofstream& out, std::ofstream& dem)` – кодирует из in в out с выводом демонстрации в dem.

`bool decode(std::ifstream& in, std::ofstream& out, std::ofstream& dem)` – декодирует из in в out с выводом в dem, возвращает true если in соответствует закодированному файлу, false – если не соответствует.

3. ИНТЕРФЕЙС ПРОГРАММЫ

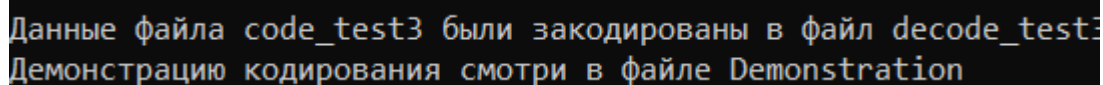
1. Ввод – файловый, пути до файлов передаются как аргументы командной строки.
2. Выбор предоставляемый пользователю в ходе программы представлен на рисунке 5.



```
Выберите что сделать?
1)Кодировать из code_test3 в decode_test3
2)Декодировать из code_test3 в decode_test3
```

Рисунок 5 – Диалоговое окно выбора сортировки

3. Информация о выполнении операции(см рис 6)



```
Данные файла code_test3 были закодированы в файл decode_test3
Демонстрацию кодирования смотри в файле Demonstration
```

Рисунок 6 – Информация о выполнении операции

5. ТЕСТИРОВАНИЕ

INPUT	OUTPUT
КОДИРОВАНИЕ	
baa	b0a01
abcde	a0b00c100d000e
ДЕКОДИРОВАНИЕ	
asdasda	Файл 'имя файла' не является закодированным текстом
a0b00c100d000e	abcde
b0a01	baa

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы была написана программа, содержащая в себе реализацию кодирования и декодирования адаптивным методом Хаффмана. Был получен опыт работы с дополнительными возможностями C++ и эффективной алгоритмизацией на нем. Также были закреплены знания полученные на протяжении семестра. Исходный код программы находится в приложении А.

ПРИЛОЖЕНИЕ А

СОДЕРЖИМОЕ ФАЙЛА CW.CPP

```
#include "bin_tree.h"

void code(std::ifstream& in, std::ofstream& out, std::ofstream& dem);
bool decode(std::ifstream& in, std::ofstream& out, std::ofstream& dem);

int main(int argc, char* argv[])
{
    if(argc == 3) {
        std::ifstream in(argv[1]);
        std::ofstream out(argv[2]);
        std::ofstream dem("Demonstration");
        if(!in.is_open() || !out.is_open() || !dem.is_open()) {
            std::cout << "Не удалось открыть файл: (" << std::endl;
            return 0;
        }
        std::cout << "Выберите что сделать?\n1)Кодировать из " << argv[1] << "
в " << argv[2] << "\n2)Декодировать из " << argv[1] << " в " << argv[2] <<
std::endl;

        int option;
        std::cin >> option;
        std::cout << std::endl;
        switch(option) {
            case 1:
                code(in , out, dem);
                std::cout << "Данные файла " << argv[1] << " были
закодированы в файл " << argv[2] << std::endl;
                std::cout << "Демонстрацию кодирования смотри в файле
Demonstration" << std::endl;
                break;
            case 2:
                if(decode(in , out, dem)) {
                    std::cout << "Данные файла " << argv[1] << " были
декодированы в файл " << argv[2] << std::endl;
                    std::cout << "Демонстрацию декодирования смотри в
файле Demonstration" << std::endl;
                }
                else {
                    std::cout << "Файл " << argv[1] << " не является
закодированным текстом" << std::endl;
                }
            }
    }
}
```

```

        return 0;
    }
    break;
default:
    std::cout << "Опции с таким номером нет(" << std::endl;
    break;
}
}
else {
    std::cout << "Введите имена файлов, откуда и куда
кодировать/декодировать!" << std::endl;
}
return 0;
}

void code(std::ifstream& in, std::ofstream& out, std::ofstream& dem) {
    bin_tree my_tree;
    dem << "Начальное состояние дерева" << std::endl;
    my_tree.print(0, 0, dem);
    std::string cur_text, cur_str;
    while (std::getline(in, cur_str)) {
        cur_text += cur_str;
        cur_text += '\n';
    }
    if(cur_text.size()) {
        cur_text.pop_back();
    }
    int cur_parent, num;

    for(char ch: cur_text) {
        num = my_tree.find(ch);
        if(my_tree.size() == (size_t)num + 1) {
            dem << std::endl << "Следующего символа '" << ch << "'" нет в
дереве => добавляем элемент (Выводим: ";
        }
        else {
            dem << std::endl << "Следующий символ '" << ch << "'" есть в
дереве (Выводим: ";
        }
        std::string stack;
        while(num != 0) {
            cur_parent = my_tree.get(num).parent;

```



```

        if(my_tree.get(cur_parent).left == num) stack += '0';
        else stack += '1';
        num = cur_parent;
    }
    for(auto i = stack.rbegin(); i != stack.rend(); ++i) {
        out << *i;
        dem << *i;
    }
    if(my_tree.find(ch) + 1 == my_tree.size()) {
        my_tree.add(ch);
        out << ch;
        dem << ch << ")" << std::endl << "Состояние дерева после
добавления:" << std::endl;
        my_tree.print(0, 0, dem);
    }
    else {
        dem << ")" << std::endl;
    }
    my_tree.rebuild(ch);
    dem << "Состояние дерева после перестройки:" << std::endl;
    my_tree.print(0, 0, dem);
}

}

bool decode(std::ifstream& in, std::ofstream& out, std::ofstream& dem) {
    bin_tree my_tree;
    dem << "Начальное состояние дерева" << std::endl;
    my_tree.print(0, 0, dem);
    std::string cur_text, cur_str;
    while (std::getline(in, cur_str)) {
        cur_text += cur_str;
        cur_text += '\n';
    }
    if(cur_text.size()) {
        cur_text.pop_back();
    }

    int num;
    for(size_t i = 0; i < cur_text.size(); ++i) {
        num = 0;
        while(my_tree.get(num).symbol == '\0') {
            if(my_tree.get(num).count == 0) break;

```

```

        if(cur_text[i] == '1') {
            num = my_tree.get(num).right;
        }
        else {
            if(cur_text[i] == '0') {
                num = my_tree.get(num).left;
            }
            else {
                dem << std::endl << "Файл не является
закодированным текстом, выход из программы...";
                return false;
            }
        }
        ++i;
    }

    if(my_tree.get(num).count == 0) {
        dem << std::endl << "Идя по пути из закодированного файла
встретили пустой лист => добавляем в дерево (Выводим: '\" << cur_text[i] << '\")' <<
std::endl;

        my_tree.add(cur_text[i]);
        out << cur_text[i];
        dem << "Состояние дерева после добавления:" << std::endl;
        my_tree.print(0, 0, dem);
        my_tree.rebuild(cur_text[i]);
        dem << "Состояние дерева после перестройки:" << std::endl;
        my_tree.print(0, 0, dem);
    }
    else {
        dem << std::endl << "Идя по пути из закодированного файла
встретили символ (Выводим: '\" << my_tree.get(num).symbol << '\")' << std::endl;
        out << my_tree.get(num).symbol;
        --i;
        my_tree.rebuild(my_tree.get(num).symbol);
        dem << "Состояние дерева после перестройки" << std::endl;
        my_tree.print(0, 0, dem);
    }
}

return true;
}

```

СОДЕРЖИМОЕ ФАЙЛА BIN_TREE.H

```
#include <iostream>
```

```

#include <fstream>
#include <vector>
#include <variant>
#include <algorithm>

class bin_tree {
    struct kanoha {
        kanoha();
        kanoha(char b, int par);
        char symbol;
        size_t count;
        int parent;
        int left;
        int right;
    };

    std::vector<kanoha> arr;
    void swap(int one, int second);

public:

    bin_tree();
    kanoha get(size_t i);
    size_t size();
    size_t find(const char a);
    void add(const char a);
    void print(int index, int deep, std::ofstream& dem);
    void rebuild(const char a);

};

```

СОДЕРЖИМОЕ ФАЙЛА BIN_TREE.CPP

```

#include "bin_tree.h"

bin_tree::kanoha::kanoha(): symbol('\0'), count(0), parent(-1), left(-1), right(-1) {}
bin_tree::kanoha::kanoha(char b, int par): symbol(b), count(0), parent(par), left(-1), right(-1) {}

void bin_tree::swap(int one, int second) {
    if(one == second) return;
    std::swap(arr[one].symbol, arr[second].symbol);
    std::swap(arr[one].count, arr[second].count);
    std::swap(arr[one].left, arr[second].left);
    std::swap(arr[one].right, arr[second].right);
    if(arr[one].right != -1)
        arr[arr[one].right].parent = one;
}

```

```

        if(arr[one].left != -1)
            arr[arr[one].left].parent = one;
        if(arr[second].right != -1)
            arr[arr[second].right].parent = second;
        if(arr[second].left != -1)
            arr[arr[second].left].parent = second;
    }

    bin_tree::bin_tree() {
        arr.push_back(kanoha());
    }

    bin_tree::kanoha bin_tree::get(size_t i) {
        return arr[i];
    }

    size_t bin_tree::size() {
        return arr.size();
    }

    size_t bin_tree::find(const char a) {
        for(size_t i = 0; i < arr.size(); ++i) {
            if(arr[i].symbol == a) {
                return i;
            }
        }
        return arr.size() - 1;
    }

    void bin_tree::add(const char a) {
        arr[arr.size() - 1].left = arr.size() + 1;
        arr[arr.size() - 1].right = arr.size();
        arr.push_back(kanoha(a, arr.size() - 1));
        arr.push_back(kanoha());
        arr[arr.size() - 1].parent = arr.size() - 3;
    }

    void bin_tree::print(int index, int deep, std::ofstream& dem) {
        if(index == -1) return;
        print(arr[index].right, deep + 1, dem);
        for(int i = 0; i < deep; ++i) dem << "\t";
        if(arr[index].symbol == '\n')
            dem << "\\n : " << arr[index].count << std::endl;
        else
            if(arr[index].symbol == '\0')
                dem << arr[index].count << std::endl;
            else
                dem << "'" << arr[index].symbol << "':" << arr[index].count
<< std::endl;
        print(arr[index].left, deep + 1, dem);
    }

    void bin_tree::rebuild(const char a) {
        size_t num = find(a);
        if(num + 2 == size() && arr[num].count == 0) {
            arr[num--].count++;
        }
        while(true) {
            arr[num].count++;
            if(num == 0) {
                return;
            }
            int i = 0;

```

```

while(arr[num].count > arr[num - i - 1].count) {
    ++i;
}
if((size_t)arr[num].parent + i == num) {
    num = num - i;
}
else {
    swap(num, num - i);
    num = arr[num - i].parent;
}
}
}

```