**Министерство науки и высшего образования Российской Федерации**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

**Факультет безопасности информационных технологий**

**Дисциплина:**

«Алгоритмы и структуры данных»

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 1**

«Пирамидальная сортировка»

**Выполнили:**

Ахраров Али Рустамович, студент группы N3250

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(подпись)

**Проверил:**

Ерофеев Сергей Анатольевич

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(отметка о выполнении)

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(подпись)

Содержание

[ВВЕДЕНИЕ 4](#_Toc177496176)

[1. БЛОК-СХЕМА ПРОГРАММЫ 5](#_Toc177496177)

[2. БЛОК-СХЕМА ФУНКЦИЙ И СОРТИРОВКИ 6](#_Toc177496178)

[3. ПЕРЕМЕННЫЕ И ФУНКЦИИ, КОТОРЫЕ НАХОДЯТСЯ В ПРОГРАММЕ 8](#_Toc177496179)

[4. КОД ПРОГРАММЫ 9](#_Toc177496180)

[5. РАСЧЁТ СЛОЖНОСТИ АЛГОРИТМА 12](#_Toc177496181)

[6. ТЕСТИРОВАНИЕ ПРОГРАММЫ 13](#_Toc177496182)

[8. ЗАКЛЮЧЕНИЕ 13](#_Toc177496183)

ВВЕДЕНИЕ

Цель работы – разработать программу, реализующую пирамидальную сортировку (Heap Sort). Оценить сложность алгоритма посредством подсчета количества элементарных операций в зависимости от количества элементов в массиве.

Задачи работы:

* Составить общий алгоритм (блок-схему) пирамидальной сортировки.
* Реализовать алгоритм на языке программирования C++.
* Провести тестирование программы.
* Оценить сложность алгоритма.

Пирамидальная сортировка – это алгоритм сортировки, основанный на структуре данных "куча" (heap), которая представляет собой бинарное дерево, удовлетворяющее свойству кучи: каждый узел больше или равен (для максимальной кучи) или меньше или равен (для минимальной кучи) своим потомкам.

**Алгоритм пирамидальной сортировки включает следующие шаги:**

1. **Построение кучи из исходных данных**

Для начала массив преобразуется в структуру кучи. Куча — это бинарное дерево, где каждый родительский узел больше (или меньше, если это минимальная куча) своих дочерних узлов. В контексте пирамидальной сортировки строится **максимальная куча**.

Построение начинается с последнего узла, имеющего потомков (средний элемент массива). Затем, начиная с него, проверяется каждый узел: если он меньше своего потомка, выполняется операция "просеивания" (heapify), которая перестраивает локальную кучу.

* **Алгоритм "просеивания" (heapify):**
  1. Если у текущего узла есть потомки, сравниваем его с ними.
  2. Если потомок больше текущего узла, меняем их местами.
  3. Повторяем процесс для изменённого узла, пока не будет восстановлено свойство кучи.

Этот процесс выполняется для всех узлов, начиная с середины массива и до первого элемента.

1. **Удаление максимального элемента (на вершине кучи) и его размещение в конце массива:**

После того как построена максимальная куча, максимальный элемент массива всегда находится на её вершине (первый элемент массива).

* 1. Меняем местами первый элемент (максимальный) с последним элементом массива.
  2. Теперь последний элемент является отсортированным и больше не участвует в процессах перестройки кучи.

1. **Восстановление свойства кучи для оставшихся элементов:**

После удаления максимального элемента (который был перемещён в конец), необходимо восстановить свойство кучи для оставшихся элементов. Для этого применяется тот же процесс "просеивания" (heapify), начиная с вершины кучи, чтобы вновь получить максимальный элемент на первой позиции.

1. **Повторение шагов 2 и 3, пока в куче не останется элементов:**

Шаги 2 и 3 повторяются до тех пор, пока в куче не останется элементов. Каждый раз максимальный элемент перемещается в конец массива, и оставшиеся элементы снова преобразуются в максимальную кучу. Этот процесс продолжается, пока весь массив не будет отсортирован.

Преимуществом пирамидальной сортировки является ее гарантированная временная сложность O(n log n) в худшем случае, что делает ее эффективной для сортировки больших объемов данных.

# БЛОК-СХЕМА ПРОГРАММЫ

Рисунок 1 – Блок-схема программы.

# БЛОК-СХЕМА ФУНКЦИЙ И СОРТИРОВКИ

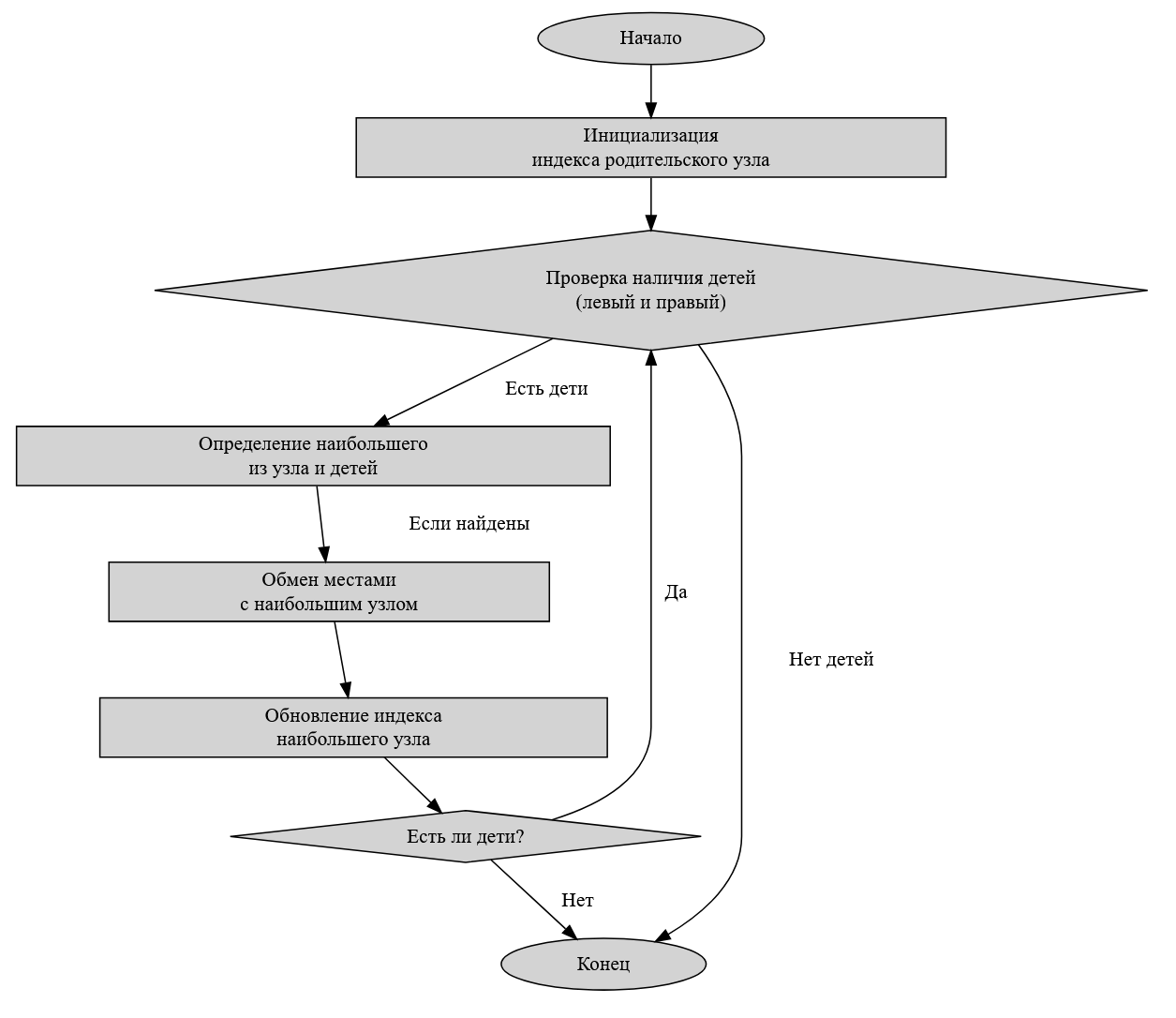


Рисунок 2 – Блок-схема функции Heapify.

# ПЕРЕМЕННЫЕ И ФУНКЦИИ, КОТОРЫЕ НАХОДЯТСЯ В ПРОГРАММЕ

* **int a[]** – массив целых чисел, который нужно отсортировать.
* **int size** – размер массива.
* **void Heap(int a[], int size, int i)** – функция для поддержания свойства кучи для поддерева с корнем в индексе i.
  1. **int largest** – индекс наибольшего элемента среди корня, левого и правого потомков.
  2. **int left** – индекс левого потомка.
  3. **int right** – индекс правого потомка.
* **void Sort(int a[], int n)** – функция для выполнения пирамидальной сортировки.
* **void Randomizer(int a[], int size)** – функция для заполнения массива случайными числами и вывода их на экран.
* **int main()** – главная функция программы.

**Диапазон значений переменных:**

* Целые числа в массиве a[] находятся в диапазоне от 1000 до 5999 (генерируются случайно).

# КОД ПРОГРАММЫ

#include <ctime>

#include <iostream>

#include <cstdlib>

// Функция Heapify для поддержания свойства кучи

void Heap(int a[], int size, int i) {

while (true) {

int largest = i; // Инициализируем наибольший элемент как корень

int left = 2 \* i + 1; // Левый потомок

int right = 2 \* i + 2; // Правый потомок

// Если левый потомок больше корня

if (left < size && a[left] > a[largest]){

largest = left;

}

// Если правый потомок больше, чем самый большой элемент на данный момент

if (right < size && a[right] > a[largest]){

largest = right;

}

// Если самый большой элемент – корень, то выход из цикла

if (largest == i){

break;

}

// Меняем местами корень и самый большой элемент

int temp = a[i];

a[i] = a[largest];

a[largest] = temp;

// Рекурсивно Heapify поддерево

i = largest;

}

}

// Функция пирамидальной сортировки

void Sort(int a[], int n){

// Построение кучи (перегруппируем массив)

for (int i = n / 2 - 1; i >= 0; i--){

Heap(a, n, i);

}

// Один за другим извлекаем элементы из кучи

for (int i = n - 1; i > 0; i--){

// Перемещаем текущий корень в конец

int temp = a[0];

a[0] = a[i];

a[i] = temp;

// Вызываем Heapify на уменьшенной куче

Heap(a, i, 0);

}

}

// Функция для заполнения массива случайными числами

void Randomizer(int a[], int size) {

std::srand(std::time(nullptr)); // Инициализируем генератор случайных чисел

for (int i = 0; i < size; i++) {

a[i] = std::rand() % 5000 + 1000; // Генерируем числа от 1000 до 5999

}

std::cout << "Source array:\n";

for (int i = 0; i < size; i++) {

std::cout << a[i] << " ";

}

std::cout << "\n";

}

// Главная функция

int main(){

int size = 100; // Размер массива

int \*a = new int[size];

Randomizer(a, size); // Заполняем массив случайными числами

Sort(a, size); // Сортируем массив

std::cout << "\nSorted array:\n";

for (int i = 0; i < size; i++){

std::cout << a[i] << " ";

}

std::cout << std::endl;

delete a;

return 0;

}

# РАСЧЁТ СЛОЖНОСТИ АЛГОРИТМА

Пирамидальная сортировка включает два основных этапа:

1. **Построение максимальной кучи** из неотсортированного массива.
2. **Извлечение максимального элемента** (который всегда находится в корне кучи), перемещение его в конец массива и восстановление структуры кучи для оставшихся элементов.
   1. **Построение кучи: сложность O(n)**

На первый взгляд может показаться, что построение кучи должно занимать время O(n log(n)), так как для каждого элемента на каждом уровне дерева выполняются операции восстановления кучи, которые могут доходить до глубины log(n). Однако на самом деле это занимает **O(n)**, и вот почему:

Восстановление кучи для каждого уровня

1. Нижние уровни: На нижних уровнях дерева у нас меньшее количество узлов, и для каждого узла нужно проделать меньше операций для восстановления кучи. Например, если рассматривать листья дерева (последний уровень), они не требуют восстановления, так как не имеют потомков.
2. Верхние уровни: Хотя на верхних уровнях у нас меньше узлов, восстановление кучи для них требует больше операций, потому что они находятся на большей глубине (до log(n)).

Важно учесть, что:

* На самом нижнем уровне у нас примерно n/2 элементов, но они требуют 0 операций восстановления (листья).
* На втором снизу уровне у нас n/4 элементов, и каждый требует 1 операцию.
* На третьем снизу уровне у нас n/8 элементов, и каждый требует 2 операции и т.д.

Если просуммировать все эти операции для всех уровней дерева, получится, что суммарное время восстановления кучи для всех уровней составляет O(n), потому что на нижних уровнях выполняется большое количество мелких операций, а на верхних уровнях — меньшее количество более "дорогих" операций. В итоге, благодаря сбалансированности этих операций, получается линейная сложность — O(n).

* 1. **Извлечение элементов из кучи: сложность O(n log(n))**

После того как куча построена, мы начинаем процесс сортировки — извлечение максимального элемента и перестройка кучи. Для понимания этой части разделим процесс на два этапа:

* 1. Извлечение максимального элемента

Куча — это бинарное дерево, где максимальный элемент всегда находится в корне (если это max-heap). Когда мы извлекаем этот элемент, нам нужно:

* Поменять местами корень и последний элемент кучи.
* Уменьшить размер кучи на один, чтобы исключить последний элемент.
* Восстановить кучу, начиная с корня.
  1. Восстановление кучи

Восстановление кучи после извлечения максимального элемента требует времени **O(log(n))**, так как мы перемещаем элемент по уровням дерева до тех пор, пока не восстановим свойства кучи. Глубина бинарного дерева — log(n), и на каждом уровне нам нужно принять решение (сравнить и поменять местами элементы), что и приводит к этой сложности.

**Общее время для всех извлечений**

Для того чтобы отсортировать все n элементов, нужно повторить процесс извлечения **n раз**, и каждый раз восстанавливать кучу, что требует **O(log(n))** операций на каждое извлечение. Таким образом, общее время на извлечение всех элементов составит **O(n log(n)).**

Итог:

* Построение кучи: O(n)
* Извлечение элементов: O(n log(n))

Таким образом, общая сложность пирамидальной сортировки **— O(n log(n)),** где построение кучи выполняется за линейное время, а сортировка элементов — за время O(n log(n)).

Так как мы выполняем извлечение для каждого элемента массива (всего n элементов), общее время на этот этап будет:

**Общая сложность пирамидальной сортировки:**

Теперь сложим сложности обоих этапов:

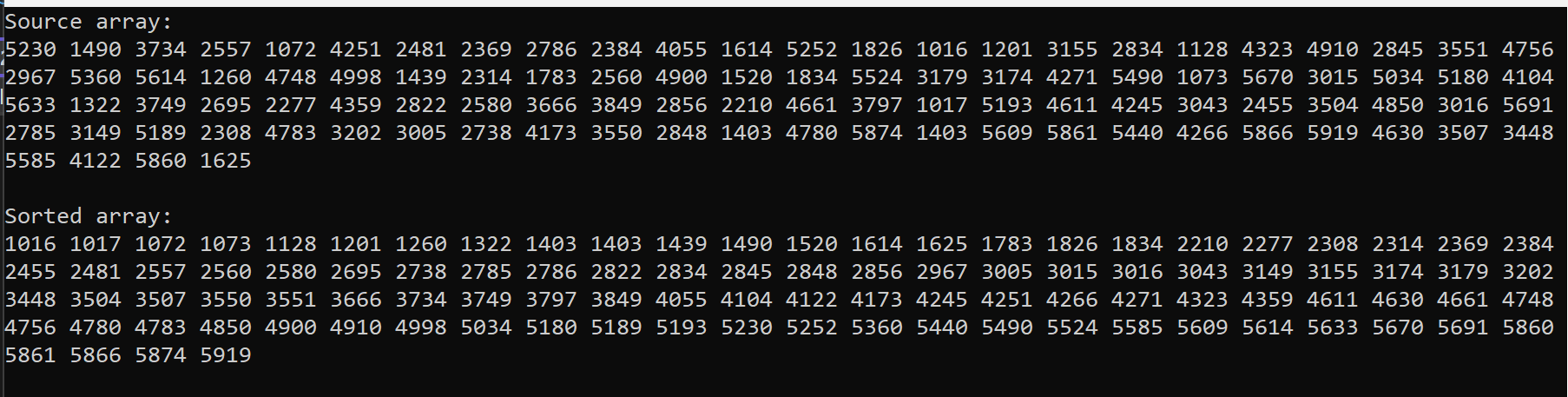
1. Построение кучи: O(n)
2. Извлечение элементов: O(n log(n))

Итоговая сложность пирамидальной сортировки:

O(n) + O(n log(n)) = O(n log(n))

Таким образом, пирамидальная сортировка выполняется за O(n log(n)) в худшем, лучшем и среднем случае. Это делает её эффективным алгоритмом сортировки, особенно когда требуется стабильная производительность для больших массивов данных.

# ТЕСТИРОВАНИЕ ПРОГРАММЫ



1. Приложение 1 – Блок-схема функции сортировки Heap Sort.

## ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была разработана программа, реализующая пирамидальную сортировку. Были изучены принципы работы с кучей, реализованы функции построения кучи и сортировки. Проведенный анализ показал, что алгоритм имеет временную сложность O(n log n) в худшем, среднем и лучшем случаях, что делает его эффективным для сортировки больших массивов данных.

Были выполнены следующие задачи:

* Составлены блок-схемы общего алгоритма и функций.
* Реализован код на языке C++.
* Проведено тестирование программы.
* Оценена сложность алгоритма.

Полученные результаты подтверждают теоретическую эффективность пирамидальной сортировки и ее практическую применимость для обработки больших объемов данных.

**Использованные инструменты:**

* Среда разработки: Visual Studio
* Язык программирования: C++.
* Средства для построения блок-схем: <https://programforyou.ru/block-diagram-redactor>.