Банкузов Михаил 7 группа Лабораторная работа №13 Вариант 1



Heap.h

#pragma once // Директива препроцессора, которая указывает компилятору на то, чтобы включить этот файл только один раз.

struct AAA // Объявляем структуру AAA.

{

int x;

void print(); // Объявляем метод print().

int getPriority() const; // Объявляем метод getPriority().

};

namespace heap // Объявляем пространство имен heap.

{

enum CMP // Перечисление, которое определяет три возможных результата сравнения двух элементов кучи.

{

LESS = -1, // Меньше.

EQUAL = 0, // Равно.

GREAT = 1 // Больше.

};

struct Heap // Объявляем структуру Heap, которая представляет собой бинарную кучу.

{

int size; // Текущий размер кучи.

int maxSize; // Максимальный размер кучи.

void\*\* storage; // Указатель на массив данных.

CMP(\*compare)(void\*, void\*); // Указатель на функцию сравнения двух элементов кучи.

// Конструктор.

Heap(int maxsize, CMP(\*f)(void\*, void\*))

{

size = 0;

storage = new void\* [maxSize = maxsize]; // Выделяем память под массив данных.

compare = f; // Инициализируем указатель на функцию сравнения.

};

// Методы для получения индексов дочерних и родительского узлов.

int left(int ix);

int right(int ix);

int parent(int ix);

// Методы для проверки полноты и пустоты кучи.

bool isFull() const

{

return (size >= maxSize);

};

bool isEmpty() const

{

return (size <= 0);

};

// Методы для проверки отношения порядка между двумя элементами кучи.

bool isLess(void\* x1, void\* x2) const

{

return compare(x1, x2) == LESS;

};

bool isGreat(void\* x1, void\* x2) const

{

return compare(x1, x2) == GREAT;

};

bool isEqual(void\* x1, void\* x2) const

{

return compare(x1, x2) == EQUAL;

};

// Метод для обмена элементов двух узлов.

void swap(int i, int j);

// Метод для приведения поддерева с корнем в узле ix к виду кучи.

void heapify(int ix);

// Метод для вставки нового элемента в кучу.

void insert(void\* x);

// Методы для извлечения максимального и минимального элементов из кучи.

void\* extractMax();

void\* extractMin();

// Метод для извлечения элемента с заданным индексом из кучи.

void\* extractI(int index);

// Метод для объединения двух куч.

Heap\* unionHeap(Heap\* heap);

// Метод для вывода на экран содержимого кучи.

void scan(int i) const;

};

// Функция для создания новой кучи.

Heap create(int maxsize, CMP(\*f)(void\*, void\*));

};

Heap.cpp

#include "Heap.h" // Подключаем заголовочный файл с объявлением кучи.

#include <iostream> // Подключаем библиотеку для работы с вводом/выводом.

#include <iomanip> // Подключаем библиотеку для работы с форматированным выводом.

void AAA::print() // Определение метода print() для структуры AAA.

{

std::cout << x; // Выводим значение переменной x на экран.

}

int AAA::getPriority() const // Определение метода getPriority() для структуры AAA.

{

return x; // Возвращаем значение переменной x.

}

namespace heap // Определяем функции и методы в пространстве имен heap.

{

Heap create(int maxsize, CMP(\*f)(void\*, void\*)) // Функция для создания новой кучи.

{

return \*(new Heap(maxsize, f)); // Создаем новый объект кучи и возвращаем его.

}

int Heap::left(int ix) // Метод для получения индекса левого потомка узла с индексом ix.

{

return (2 \* ix + 1 >= size) ? -1 : (2 \* ix + 1); // Если индекс левого потомка находится за пределами кучи, возвращаем -1, иначе возвращаем его значение.

}

int Heap::right(int ix) // Метод для получения индекса правого потомка узла с индексом ix.

{

return (2 \* ix + 2 >= size) ? -1 : (2 \* ix + 2); // Если индекс правого потомка находится за пределами кучи, возвращаем -1, иначе возвращаем его значение.

}

int Heap::parent(int ix) // Метод для получения индекса родительского узла для узла с индексом ix.

{

return (ix + 1) / 2 - 1; // Вычисляем индекс родительского узла и возвращаем его значение.

}

void Heap::swap(int i, int j) // Метод для обмена элементами с индексами i и j в куче.

{

void\* buf = storage[i]; // Временно сохраняем значение элемента с индексом i в буфер.

storage[i] = storage[j]; // Перезаписываем элемент с индексом i значением элемента с индексом j.

storage[j] = buf; // Перезаписываем элемент с индексом j значением из буфера.

}

void Heap::heapify(int ix) // Метод для приведения поддерева с корнем в узле ix к виду кучи.

{

int l = left(ix), r = right(ix), irl = ix; // Получаем индексы левого и правого потомков узла ix, а также индекс узла с наибольшим значением.

if (l > 0)

{

if (isGreat(storage[l], storage[ix])) irl = l; // Если значение левого потомка больше значения узла ix, обновляем индекс узла с наибольшим значением.

if (r > 0 && isGreat(storage[r], storage[irl])) irl = r; // Если значение правого потомка больше значения узла с наибольшим значением, обновляем этот индекс.

if (irl != ix) // Если индекс узла с наибольшим значением отличается от ix, обмениваем элементы в узлах ix и irl и вызываем метод heapify для узла irl.

{

swap(ix, irl);

heapify(irl);

}

}

}

void Heap::insert(void\* x) // Метод для вставки нового элемента x в кучу.

{

int i;

if (!isFull()) // Если куча не заполнена, добавляем новый элемент.

{

storage[i = ++size - 1] = x; // Добавляем x в конец массива данных кучи и сохраняем его индекс в i.

while (i > 0 && isLess(storage[parent(i)], storage[i])) // Пока индекс i не достигнет корня кучи и значение родительского узла больше значения добавленного элемента, меняем местами эти узлы.

{

swap(parent(i), i);

i = parent(i);

}

}

}

void\* Heap::extractMax() // Метод для извлечения максимального элемента из кучи.

{

void\* rc = nullptr; // Инициализируем указатель rc значением nullptr.

if (!isEmpty()) // Если куча не пуста, извлекаем максимальный элемент.

{

rc = storage[0]; // Сохраняем значение максимального элемента в rc.

storage[0] = storage[size - 1]; // Перезаписываем корень кучи значением последнего элемента в куче.

size--; // Уменьшаем размер кучи на 1.

heapify(0); // Приводим поддерево с корнем в узле 0 к виду кучи.

}

return rc; // Возвращаем указатель на извлеченный максимальный элемент.

}

void\* Heap::extractMin() // Метод для удаления минимального элемента из кучи.

{

void\* rc = nullptr; // Инициализируем указатель rc значением nullptr.

if (!isEmpty()) // Если куча не пуста, удаляем минимальный элемент.

{

// Вычисляем индекс минимального элемента.

int index = pow(2, floor(log2(size))) - 1;

rc = storage[index]; // Сохраняем значение минимального элемента в rc.

storage[index] = storage[size - 1]; // Перезаписываем элемент с индексом index значением последнего элемента в куче.

size--; // Уменьшаем размер кучи на 1.

int i = size;

while (i > 0 && isLess(storage[parent(i)], storage[i])) // Пока индекс i не достигнет корня кучи и значение родительского узла больше значения удаляемого элемента, меняем местами эти узлы.

{

swap(parent(i), i);

i = parent(i);

}

heapify(0); // Приводим поддерево с корнем в узле 0 к виду кучи.

}

return rc; // Возвращаем указатель на удаленный минимальный элемент.

}

void\* Heap::extractI(int index) // Метод для удаления элемента по индексу из кучи.

{

void\* rc = nullptr; // Инициализируем указатель rc значением nullptr.

if (!isEmpty() && index < size) // Если куча не пуста и индекс находится в пределах кучи, удаляем элемент.

{

rc = storage[index]; // Сохраняем значение удаляемого элемента в rc.

storage[index] = storage[size - 1]; // Перезаписываем элемент с индексом index значением последнего элемента в куче.

size--; // Уменьшаем размер кучи на 1.

heapify(0); // Приводим поддерево с корнем в узле 0 к виду кучи.

}

return rc; // Возвращаем указатель на удаленный элемент.

}

Heap\* Heap::unionHeap(Heap\* heap) // Метод для объединения двух куч.

{

for (int i = 0; i < heap->size; i++) // Добавляем элементы из второй кучи в первую.

insert(heap->storage[i]);

return this; // Возвращаем указатель на первую кучу.

}

// Определение метода класса Heap для вывода значений элементов на экран

void Heap::scan(int i) const

{

int probel = 20; // Инициализация переменной probel

std::cout << '\n'; // Перевод курсора на новую строку

// Проверка, пуста ли куча

if (size == 0)

{

std::cout << "Куча пуста"; // Вывод сообщения о пустой куче

}

// Цикл по всем элементам кучи

for (int u = 0, y = 0; u < size; u++)

{

// Вывод значения элемента на экран с отступом probel + 10

std::cout << std::setw(probel + 10) << std::setfill(' ');

((AAA\*)storage[u])->print();

// Если текущий индекс равен y, перевод курсора на новую строку и изменение значения y

if (u == y)

{

std::cout << '\n';

if (y == 0)

y = 2;

else

y += y \* 2;

}

// Уменьшение отступа probel вдвое

probel /= 2;

}

std::cout << '\n'; // Перевод курсора на новую строку

}

}

Основной файл

// Подключение заголовочного файла Heap.h

#include "Heap.h"

// Подключение библиотеки ввода-вывода

#include <iostream>

// Использование пространства имен std

using namespace std;

// Определение функции сравнения для типа данных AAA

// Функция возвращает результат сравнения двух объектов AAA

heap::CMP cmpAAA(void\* a1, void\* a2)

{

// Определение макросов для удобства доступа к полям объектов AAA

#define A1 ((AAA\*)a1)

#define A2 ((AAA\*)a2)

heap::CMP rc = heap::EQUAL; // Инициализация результата сравнения

// Сравнение значений поля x двух объектов AAA

if (A1->x > A2->x)

rc = heap::GREAT;

else if (A2->x > A1->x)

rc = heap::LESS;

// Очистка макросов

#undef A2

#undef A1

// Возврат результата сравнения

return rc;

#undef A2

#undef A1

}

//-------------------------------

int main()

{

setlocale(LC\_ALL, "rus");

int k, choice;

// Создание кучи с максимальным размером 30 и функцией сравнения cmpAAA

heap::Heap h1 = heap::create(30, cmpAAA);

// Добавление 10 объектов AAA в кучу h1

for (int i = 0; i < 10; i++)

{

AAA\* a = new AAA;

a->x = i;

h1.insert(a);

}

// Бесконечный цикл для работы с кучей

for (;;)

{

cout << "1 - вывод кучи на экран" << endl;

cout << "2 - добавить элемент" << endl;

cout << "3 - удалить максимальный элемент" << endl;

cout << "4 - удалить минимальный элемент" << endl;

cout << "5 - удалить элемента по индексу" << endl;

cout << "6 - тэстовое объединение" << endl;

cout << "0 - выход" << endl;

cout << "сделайте выбор" << endl; cin >> choice;

switch (choice)

{

case 0: exit(0);

case 1: h1.scan(0);

break;

case 2: { AAA\* a = new AAA;

cout << "введите ключ" << endl; cin >> k;

a->x = k;

h1.insert(a);

}

break;

case 3: h1.extractMax();

break;

case 4: h1.extractMin();

break;

case 5:

{

int index = 0;

cout << "введите индекс" << endl;

cin >> index;

h1.extractI(index);

break;

}

case 6:

{

heap::Heap h2 = heap::create(30, cmpAAA);

h2.insert(new int(100));

h2.insert(new int(200));

h2.insert(new int(300));

h1.unionHeap(&h2);

break;

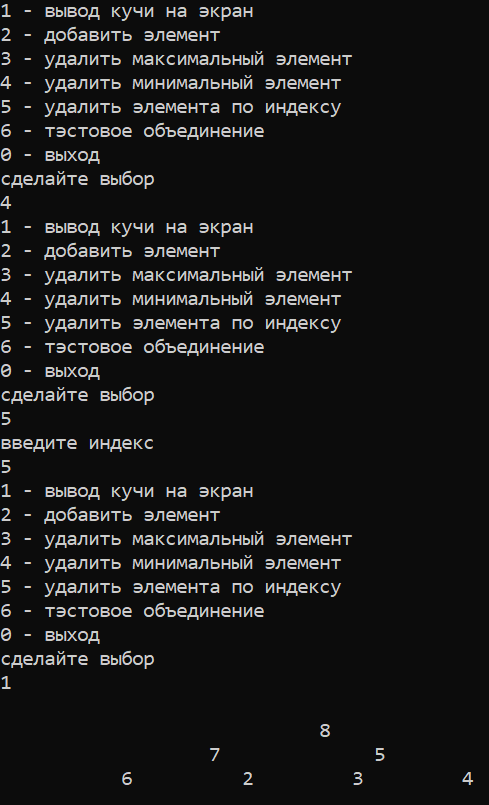
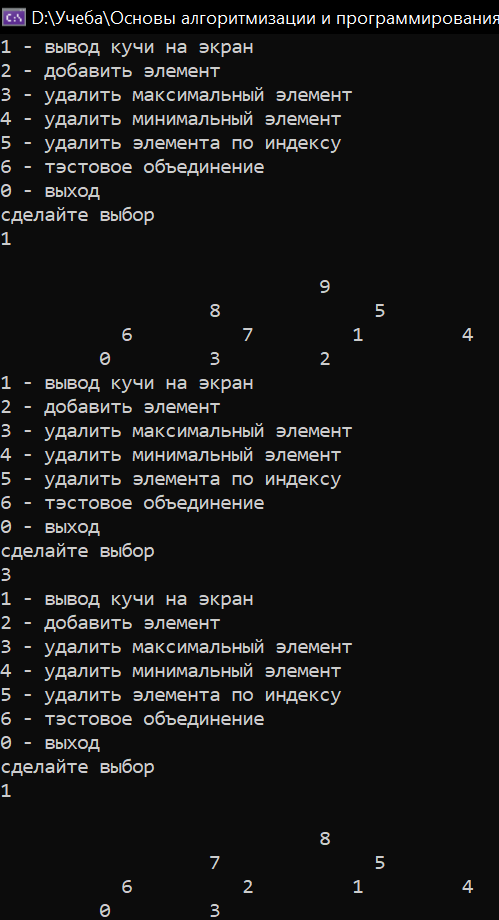
}

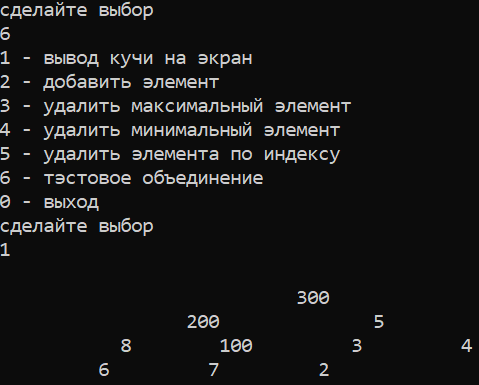
default: cout << endl << "Введена неверная команда!" << endl;

}

} return 0;

}





Дополнительные задания

Задание 1

Реализовать бинарную кучу (max-heap или min-heap) на основе массива. Операции, которые должны быть поддерживаемыми: вставка элемента, удаление корневого элемента (максимального или минимального, в зависимости от типа кучи), построение кучи из неупорядоченного массива.

#include <iostream>

#include <vector>

using namespace std;

class BinaryHeap {

public:

BinaryHeap(bool is\_min\_heap = false) : is\_min\_heap\_(is\_min\_heap) {}

void Insert(int value) {

heap\_.push\_back(value); // добавляем новый элемент в конец массива

int index = heap\_.size() - 1; // запоминаем индекс добавленного элемента

if (is\_min\_heap\_) { // если это минимальная куча

// выполняем восходящую сортировку (min-heap)

while (index > 0 && heap\_[index] < heap\_[(index - 1) / 2]) {

swap(heap\_[index], heap\_[(index - 1) / 2]); // меняем местами элементы

index = (index - 1) / 2; // обновляем индекс

}

}

else { // иначе это максимальная куча

// выполняем восходящую сортировку (max-heap)

while (index > 0 && heap\_[index] > heap\_[(index - 1) / 2]) {

swap(heap\_[index], heap\_[(index - 1) / 2]); // меняем местами элементы

index = (index - 1) / 2; // обновляем индекс

}

}

}

int Extract() {

if (heap\_.empty()) { // если куча пуста

throw logic\_error("Heap is empty"); // выбрасываем исключение

}

int result = heap\_[0]; // запоминаем значение корневого элемента

heap\_[0] = heap\_.back(); // заменяем корневой элемент последним элементом

heap\_.pop\_back(); // удаляем последний элемент

if (heap\_.size() > 1) { // если в куче остались элементы

int index = 0;

while (2 \* index + 1 < heap\_.size()) { // пока есть хотя бы один потомок

int left\_child = 2 \* index + 1; // индекс левого потомка

int right\_child = 2 \* index + 2; // индекс правого потомка

int child = left\_child; // запоминаем индекс потомка

if (right\_child < heap\_.size() &&

((is\_min\_heap\_ && heap\_[right\_child] < heap\_[left\_child]) ||

(!is\_min\_heap\_ && heap\_[right\_child] > heap\_[left\_child]))) {

child = right\_child; // выбираем наименьшего/наибольшего потомка

}

if ((is\_min\_heap\_ && heap\_[index] > heap\_[child]) ||

(!is\_min\_heap\_ && heap\_[index] < heap\_[child])) { // если нарушается порядок кучи

swap(heap\_[index], heap\_[child]); // меняем местами элементы

index = child; // обновляем индекс

}

else {

break; // порядок кучи не нарушается, выходим из цикла

}

}

}

return result; // возвращаем значение корневого элемента

}

void BuildHeap(vector<int>& values) {

heap\_ = values; // копируем элементы из переданного массива

for (int i = heap\_.size() / 2 - 1; i >= 0; i--) {

if (is\_min\_heap\_) {

MinHeapify(i); // выполняем восстановление порядка для каждого поддерева

}

else {

MaxHeapify(i); // выполняем восстановление порядка для каждого поддерева

}

}

}

void Print() {

for (int value : heap\_) {

cout << value << " "; // выводим элементы кучи через пробел

}

cout << endl; // выводим перевод строки

}

private:

vector<int> heap\_; // массив для хранения элементов кучи

bool is\_min\_heap\_; // флаг, указывающий, является ли куча минимальной

void MaxHeapify(int index) {

int left\_child = 2 \* index + 1; // индекс левого потомка

int right\_child = 2 \* index + 2; // индекс правого потомка

int largest = index; // запоминаем индекс наибольшего элемента

if (left\_child < heap\_.size() && heap\_[left\_child] > heap\_[largest]) {

largest = left\_child; // если левый потомок больше текущего элемента, запоминаем его индекс

}

if (right\_child < heap\_.size() && heap\_[right\_child] > heap\_[largest]) {

largest = right\_child; // если правый потомок больше текущего элемента, запоминаем его индекс

}

if (largest != index) { // если нарушается порядок кучи

swap(heap\_[index], heap\_[largest]); // меняем местами элементы

MaxHeapify(largest); // рекурсивно вызываем функцию для потомка

}

}

void MinHeapify(int index) {

int left\_child = 2 \* index + 1; // индекс левого потомка

int right\_child = 2 \* index + 2; // индекс правого потомка

int smallest = index; // запоминаем индекс наименьшего элемента

if (left\_child < heap\_.size() && heap\_[left\_child] < heap\_[smallest]) {

smallest = left\_child; // если левый потомок меньше текущего элемента, запоминаем его индекс

}

if (right\_child < heap\_.size() && heap\_[right\_child] < heap\_[smallest]) {

smallest = right\_child; // если правый потомок меньше текущего элемента, запоминаем его индекс

}

if (smallest != index) { // если нарушается порядок кучи

swap(heap\_[index], heap\_[smallest]); // меняем местами элементы

MinHeapify(smallest); // рекурсивно вызываем функцию для потомка

}

}

};

int main() {

BinaryHeap max\_heap; // создаем новую максимальную кучу

max\_heap.Insert(10); // добавляем элементы

max\_heap.Insert(20);

max\_heap.Insert(15);

max\_heap.Insert(30);

max\_heap.Insert(25);

max\_heap.Print(); // выводим элементы кучи

max\_heap.Extract(); // удаляем корневой элемент

max\_heap.Print(); // выводим элементы кучи

vector<int> values = { 3, 2, 1, 5, 6, 4 }; // создаем массив элементов

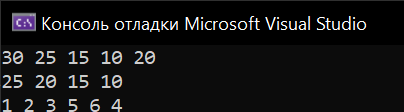
BinaryHeap min\_heap(true); // создаем новую минимальную кучу

min\_heap.BuildHeap(values); // строим кучу на основе массива

min\_heap.Print(); // выводим элементы кучи

return 0;

}



Задание 2

Найти k-ый максимальный элемент в бинарной куче.

// Подключение необходимых заголовочных файлов

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

// Определение функции для поиска k-го максимального элемента в бинарной куче

int kthLargest(vector<int>& nums, int k) {

// Создание минимальной кучи

priority\_queue<int, vector<int>, greater<int>> minHeap;

// Добавление элементов из вектора nums в кучу

for (int num : nums) {

minHeap.push(num);

// Если размер кучи превышает k, удаляем наименьший элемент

if (minHeap.size() > k) {

minHeap.pop();

}

}

// Возвращение k-го максимального элемента из кучи

return minHeap.top();

}

// Основная функция программы

int main() {

// Инициализация вектора nums и значения k

vector<int> nums = { 3, 2, 1, 5, 6, 4 };

int k = 2;

// Вызов функции для поиска k-го максимального элемента

int kthLargestElement = kthLargest(nums, k);

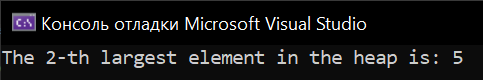
// Вывод результата на экран

cout << "The " << k << "-th largest element in the heap is: " << kthLargestElement << endl;

// Возвращение нулевого значения, чтобы сообщить ОС об успешном завершении программы

return 0;

}



Задание 3

Реализовать операции нахождения среднего арифметического и медианы множества чисел с помощью бинарной кучи.

// Подключение необходимых заголовочных файлов

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

// Определение класса MedianHeap, который использует две бинарные кучи (maxHeap и minHeap) для хранения меньших и больших чисел соответственно

class MedianHeap {

private:

priority\_queue<int> maxHeap; // Максимальная куча для меньших чисел

priority\_queue<int, vector<int>, greater<int>> minHeap; // Минимальная куча для больших чисел

public:

// Добавление числа в кучу

void addNumber(int num) {

if (maxHeap.empty() || num <= maxHeap.top()) {

maxHeap.push(num);

}

else {

minHeap.push(num);

}

// Балансировка куч

if (maxHeap.size() > minHeap.size() + 1) {

minHeap.push(maxHeap.top());

maxHeap.pop();

}

else if (minHeap.size() > maxHeap.size() + 1) {

maxHeap.push(minHeap.top());

minHeap.pop();

}

}

// Нахождение медианы

double findMedian() {

if (maxHeap.size() == minHeap.size()) {

return (maxHeap.top() + minHeap.top()) / 2.0;

}

else if (maxHeap.size() > minHeap.size()) {

return maxHeap.top();

}

else {

return minHeap.top();

}

}

// Нахождение среднего арифметического

double findMean() {

double sum = 0.0;

int count = 0;

priority\_queue<int> maxCopy = maxHeap;

priority\_queue<int, vector<int>, greater<int>> minCopy = minHeap;

// Вычисление суммы всех чисел в кучах и их количества

while (!maxCopy.empty()) {

sum += maxCopy.top();

maxCopy.pop();

count++;

}

while (!minCopy.empty()) {

sum += minCopy.top();

minCopy.pop();

count++;

}

// Вычисление среднего арифметического

return sum / count;

}

};

// Основная функция программы

int main() {

// Создание экземпляра класса MedianHeap

MedianHeap medianHeap;

// Добавление чисел в кучу

medianHeap.addNumber(3);

medianHeap.addNumber(2);

medianHeap.addNumber(1);

medianHeap.addNumber(5);

medianHeap.addNumber(6);

medianHeap.addNumber(4);

// Вычисление медианы и среднего арифметического

double median = medianHeap.findMedian();

double mean = medianHeap.findMean();

// Вывод результатов на экран

cout << "Median: " << median << endl;

cout << "Mean: " << mean << endl;

// Возвращение нулевого значения, чтобы сообщить ОС об успешном завершении программы

return 0;

}

