Файл Heap.h

#ifndef HEAP\_H

#define HEAP\_H

#include <iostream>

//перечень кодов ошибок

enum class HeapError {

Size, NO\_ELEMENTS, Index, isFull, isEmpty

};

//класс пирамиды

template <class T>

class Heap {

private:

// hlist указывает на массив, который может быть динамически создан

// конструктором (inArray == 0) или передан как параметр (inArray == 1)

T\* hlist;

int inArray;

// максимальный и текущий размеры пирамиды

int maxheapsize;

// определяет конец списка

int heapsize;

//утилита восстановления пирамидальной структуры начиная с индекса i,

//поднятие вверх по дереву,

//менять элементы мечтаит, если сын меньше родителя

void FilterUp(int i) {

int currentpos, parentpos; //currentpos - индекс текущей позиции на пути предков

T target; //target - вставляемое значение, для которого выбирается

// правильная позиция в пирамиде

currentpos = i;

parentpos = (i - 1) / 2;

target = this->hlist[i];

while (currentpos != 0) { // подниматься к корню по пути родителей

if (this->hlist[parentpos] <= target) // если родитель <= target, то все в

порядке.

break;

else {

// поменять местами родителя с сыном и обновить индексы

// для проверки следующего родителя

// переместить данные из родительской позиции в текущую.

// назначить родительскую позицию текущей.

// проверить следующего родителя

this->hlist[currentpos] = hlist[parentpos];

currentpos = parentpos;

parentpos = (currentpos - 1) / 2;

}

}

this->hlist[currentpos] = target; // правильная позиция найдена. поместить

туда target

}

//восстановление пирамидальности от определённого индекса

//для метода удаления

void FilterDown(int i) {

int currentpos, childpos;

T target;

// начать с узла і и присвоить его значение переменной target

currentpos = i;

target = this->hlist[i];

// вычислить индекс левого сына и начать движение вниз по пути,

// проходящему через меньших сыновей до конца списка

childpos = 2 \* i + 1;

while (childpos < this->heapsize) { // пока не конец списка

// индекс правого сына равен childpos+1. присвоить переменной

// childpos индекс наименьшего из двух сыновей

if ((childpos + 1 < this->heapsize) &&

(this->hlist[childpos + 1] <= this->hlist[childpos]))

childpos = childpos + 1;

// если родитель меньше сына, пирамида в порядке. выход

if (target <= this->hlist[childpos])

break;

else {

// переместить значение меньшего сына в родительский узел.

// теперь позиция меньшего сына не занята

this->hlist[currentpos] = this->hlist[childpos];

// обновить индексы и продолжить сканирование

currentpos = childpos;

childpos = 2 \* currentpos + 1;

}

}

this->hlist[currentpos] = target;

}

public:

// конструкторы и деструктор

Heap(int maxsize) {

this->maxheapsize = maxsize;

this->heapsize = 0;

this->hlist = new T[maxsize];

this->inArray = false;

}

// создать пустую пирамиду

Heap(T arr[], int n) {

if (n <= 0) // если является недопустимым размером массива

throw HeapError::Size;

// использовать п для установки размера пирамиды и максимального размера пирамиды.

// копировать массив arr в список пирамиды

this->maxheapsize = n;

this->heapsize = n;

this->hlist = arr;

// присвоить переменной currentpos индекс последнего родителя.

// вызывать FilterDown в цикле с индексами currentpоз…0

int currentpos = (this->heapsize - 2) / 2;

while (currentpos >= 0) {

// выполнить условие пирамидальности для поддерева

// с корнем hlist[currentpos)

FilterDown(currentpos);

currentpos--;

}

// присвоить флажку inArray значение True

this->inArray = true;

}

// преобразовать arr в пирамиду

Heap(const Heap<T>& H) {

this = H;

}

//дестрктор

~Heap(void) {

ClearList();

delete[] this->hlist;

}

// перегруженные операторы: "=", "[]"

Heap<T> operator= (const Heap <T> heap) {

int n = heap.maxheapsize;

this->heapsize = heap.heapsize;

this->maxheapsize = n;

this->hlist = new T[n];

if (this->hlist == NULL)

throw HeapError::NO\_ELEMENTS;

// копировать элементы массива в текущий объект

T\* srcptr = heap.hlist; // адрес начала источника

T\* destptr = this->hlist; // адрес начала копии

while (n--) // копировать список

\*destptr++ = \*srcptr++;

return \*this;

}

//операция индексации

const T operator[] (int i) {

if (i<0 || i>(this->heapsize - 1)) //если индекс вне диапазона

throw HeapError::Index;

return this->hlist[i];

}

// методы обработки списков

int ListSize() const {

return this->heapsize;

}

int MaxSize() const {

return this->maxheapsize;

}

bool ListEmpty() const {

return (this->heapsize <= 0);

}

bool ListFull() const {

return (this->heapsize == this->maxheapsize);

}

void Insert(const T& item) {

// проверить, заполнена ли пирамида и выйти, если да

if (this->heapsize == this->maxheapsize)

throw HeapError::isFull;

// записать элемент в конец пирамиды и увеличить heapsize.

// вызвать FilterUp для восстановления пирамидального упорядочения

this->hlist[this->heapsize] = item;

FilterUp(this->heapsize);

this->heapsize++;

}

T Delete(void) {

T tempitem;

//проверить, пуста ли пирамида

if (heapsize == 0)

throw HeapError::isEmpty;

// копировать корень в tempitem. заменить корень последним элементом

// пирамиды и произвести декремент переменной heapsize

tempitem = hlist[0];

hlist[0] = hlist[heapsize - 1];

heapsize--;

// вызвать FilterDown для установки нового значения корня

FilterDown(0);

// возвратить исходное значение корня

return tempitem;

}

void ClearList(void) {

this->hlist = new T[maxheapsize];

this->heapsize = 0;

}

};

#endif // HEAP\_H

Файл Sort.h

#ifndef SORT\_H

#define SORT\_H

#include <math.h>

#include "Heap.h"

//класс узла

template <class T>

class DataNode

{

public:

T data; //данные

int index; //индекс

bool active; //состояние активности

friend bool operator < (const DataNode<T>& X, const DataNode<T>& Y) {

return X.data < Y.data; //сравнение данных

}

};

int PowerOfTwo(int n) {

int res = log2(n);

return res + (log2(n) > res);

}

//обновление дерева

template <class T>

void UpdateTree(DataNode<T>\* tree, int i) {

int j; //индекс сравниваемого элемента

// определить соперника победителя. позволить ему продолжить

// турнир, копируя его в родительский узел.

if (i % 2 == 0)

tree[(i - 1) / 2] = tree[i - 1]; // соперник - левый узел

else

tree[(i - 1) / 2] = tree[i + 1]; // соперник - правый узел

// переиграть те матчи, в которых принимал участие

// только что исключенный из турнира игрок

i = (i - 1) / 2;

while (i > 0) {

// правый или левый узел?

(i % 2 == 0) ?

j = i - 1 : j = i + 1;

// проверить, является активным

if (!tree[i].active || !tree[j].active)

(tree[i].active) ?

tree[(i - 1) / 2] = tree[i] : tree[(i - 1) / 2] = tree[j];

// устроить соревнование.

// победителя скопировать в родительский узел

else

(tree[i] < tree[j]) ?

tree[(i - 1) / 2] = tree[i] : tree[(i - 1) / 2] = tree[j];

// перейти к следующему кругу соревнования (родительский уровень)

i = (i - 1) / 2;

} //турнир с новым соперником закончен

}

//турнирная сортировка

template <class T>

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

DataNode<T>\* tree; // корень дерева

DataNode<T> item;

// минимальная степень двойки, большая или равная п

int bottomRowSize;

// число узлов в полном дереве, нижний ряд которого

// имеет bottomRowSize узлов

int treesize;

// начальный индекс нижнего ряда узлов

int loadindex;

int i, j = 0;

// определить требуемый размер памяти для нижнего ряда узлов

bottomRowSize = PowerOfTwo(n);

// вычислить размер дерева и динамически создать его узлы

treesize = pow(2, bottomRowSize + 1) - 1;

tree = new DataNode<T>[treesize];

//вычисляем начальный индекс элементов

loadindex = treesize - pow(2, bottomRowSize);

// скопировать массив в дерево объектов типа DataNode

for (i = loadindex; i < treesize; i++) {

item.index = i;

if (j < n) { //если индекс не превышает кол-во элементов

item.active = true; //определяем, что элемент участвует

item.data = a[j++]; //копируем данные

}

else item.active = false; //иначе элемент не участвует

tree[i] = item;

}

// выполнить начальные сравнения для определения наименьшего элемента

i = loadindex;

while (i > 0) {

j = i; //определяем индекс начала уровня

while (j < 2 \* i) { // обработать пары соревнующихся

// проведение матча. сравнить tree [j] с его соперником tree [j + 1]

// скопировать победителя в родительский узел

(!tree[j + 1].active || tree[j] < tree[j + 1]) ?

tree[(j - 1) / 2] = tree[j] : tree[(j - 1) / 2] = tree[j + 1];

j += 2;

}

// перейти к следующему кругу соревнования (родительский уровень)

i = (i - 1) / 2;

}

// обработать оставшиеся n-1 элементов, и обновить

// дерево, разрешив сопернику победителя снова войти в турнир

for (i = 0; i < n - 1; i++) {

a[i] = tree[0].data; //копируем победителя

tree[tree[0].index].active = false; //сделаем победителя неактивным

UpdateTree(tree, tree[0].index);

}

// скопировать последнее значение в массив

a[n - 1] = tree[0].data;

}

template <class T>

void HeapSort(T a[],int n) {

// конструктор преобразующий массив а в пирамиду

Heap<T> H(a,n);

T elt;

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

// исключить наименьший элемент и заполнить его в A[i]

elt = H.Delete();

a[i] = elt;

}

}

#endif // SORT\_H

Файл main.cpp

#include <iostream>

#include <fstream>

#include <clocale>

#include <time.h>

#include "Sort.h"

#include "Heap.h"

using namespace std;

//заполнение массива

int\* create\_mas(int n, int min, int max) {

int\* res = new int[n];

srand(time(0)); //задаём условие рандома

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

res[i] = rand() % (max - min) + min; //заполняем в диапазоне

}

return res;

}

//печать массива

void print\_mas(int\* arr, int n, ostream& strm) {

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

if (i != 0 && i % 8 == 0) { //10 элементов в строке

strm << endl;

}

strm << arr[i] << " ";

}

strm << endl;

}

//печать пирамиды

void print\_heap(Heap<int>& heap, ostream& strm) {

for (int i = 0; i < heap.ListSize(); i++) {

strm << "[" << i << "]:" << heap[i] << endl;

}

}

int main() {

setlocale(LC\_ALL, "Rus");

ofstream fout;

fout.open("Result.txt");

//создаём и заполняем массив

int n = 20; //размер

int\* arr1 = create\_mas(n, 100, 1000);

//печать исходного массива

cout << "Начальный массив (n=20)" << endl;

fout << "Начальный массив (n=20)" << endl;

print\_mas(arr1, n, cout);

print\_mas(arr1, n, fout);

//сортируем пирамидальной

HeapSort(arr1, n);

//печать конечного массива

cout << "Конечный массив" << endl;

fout << "Конечный массив" << endl;

print\_mas(arr1, n, cout);

print\_mas(arr1, n, fout);

cout << endl;

fout << endl;

//создаём и заполняем массив

int n2 = 200;

int\* arr2 = create\_mas(n2, 100, 1000);

//печать исходного массива

cout << "Начальный массив (n=200)" << endl;

fout << "Начальный массив (n=200)" << endl;

print\_mas(arr2, n2, cout);

print\_mas(arr2, n2, fout);

//сортируем пирамидальной

HeapSort(arr2, n2);

//печать конечного массива

cout << "Конечный массив" << endl;

fout << "Конечный массив" << endl;

print\_mas(arr2, n2, cout);

print\_mas(arr2, n2, fout);

cout << endl;

fout << endl;

//заполняем первую пирамиду

Heap<int> h1(arr2, n2), h2(n2);

//печатаем содержимое пирамиды

cout << "Пирамида 1: " << endl;

fout << "Пирамида 1: " << endl;

print\_heap(h1, cout);

print\_heap(h1, fout);

while (!h1.ListEmpty()) { //пока в 1-й пирамиде есть элементы

int buf = h1.Delete(); //извлекаем элемент из 1-й

h2.Insert(buf); //и вставляем во 2-ю

}

//печатаем содержимое пирамиды

cout << "Пирамида 2: " << endl;

fout << "Пирамида 2: " << endl;

print\_heap(h2, cout);

print\_heap(h2, fout);

//освобождаем память

delete[] arr1;

delete[] arr2;

return 0;

}