ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

**БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ**

**ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

(НИУ «БелГУ»)

ИНСТИТУТ ИНЖЕНЕРНЫХ И ЦИФРОВЫХ ТЕХНОЛОГИЙ

КАФЕДРА МАТЕМАТИЧЕСКОГО И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

**Отчет по лабораторной работе 2**

**по дисциплине: «Алгоритмы и анализ сложности»**

**Тема работы «Оценка временной сложности алгоритмов»**

студента очного отделения

3 курса 12001801 группы

Капустина Виктора Сергеевича

Проверил(а):

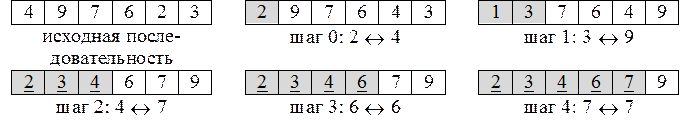
Гайворонский Виталий Александрович

Белгород 2021

Алгоритм сортировки выбором и алгоритм пирамидальной сортировки

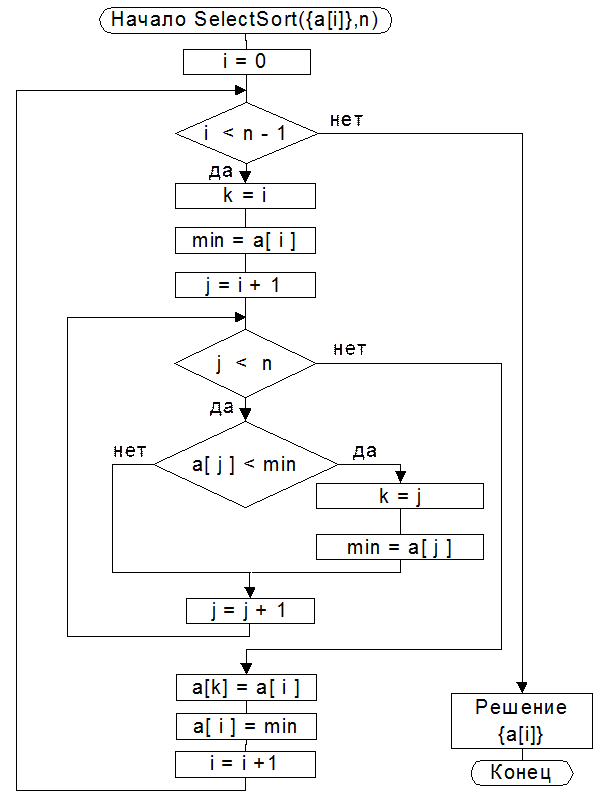
## Анализ сортировки обменом

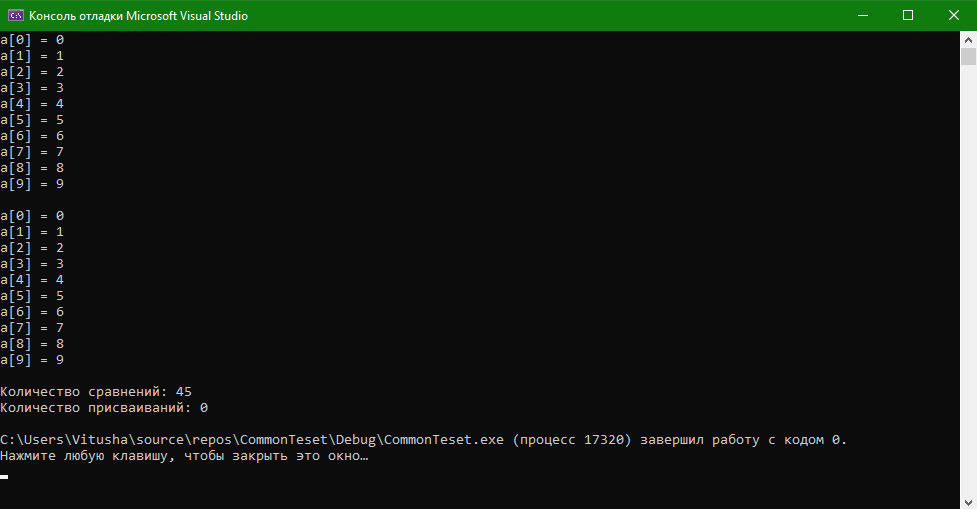
Идея метода состоит в том, чтобы создавать отсортированную последовательность путем сравнивая каждого элемента с другим и их перестановкой, если первый оказался меньше, что приводит массив в состояние упорядоченности. Следуя этой идее, будем строить готовую последовательность, начиная с левого конца массива. Алгоритм состоит из *n-*1 последовательных шагов, начиная от нулевого и заканчивая (*n-*2)-м. На *i*-м шаге выбираем наименьший из элементов *a[i] ... a[n-1]* и меняем его местами с *a[i]*.

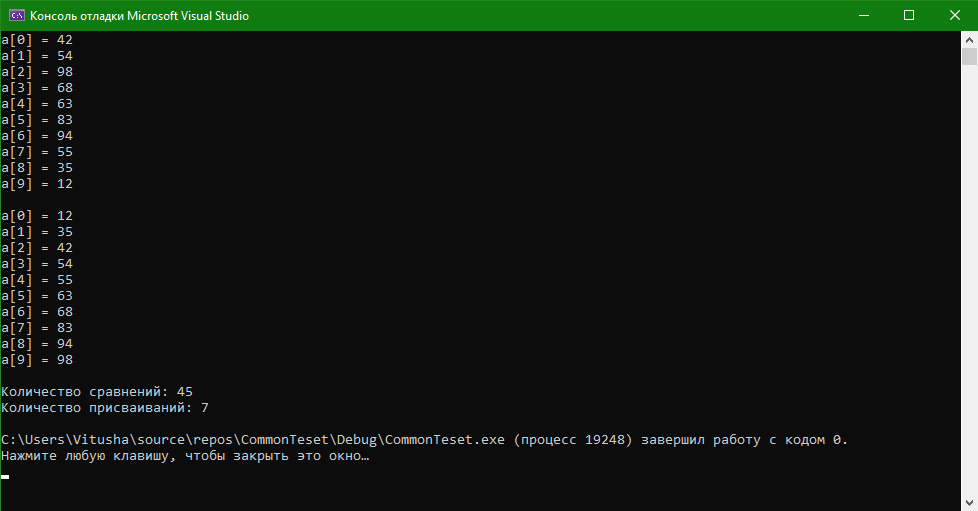
  
Рисунок 1. Пример работы сортировки обменом

В зависимости от номера текущего шага *i*, можно сказать, сколько элементов уже упорядочены. Последовательность *a[0]...a[i]* (на Рис. 1 она выделена серым) является упорядоченной. Таким образом, на (*n-*2)-м шаге вся последовательность, кроме *a[n-1]* окажется отсортированной, а *a[n-1]* останется на последнем месте т.к. все меньшие по значению элементы уже ушли влево и отсортированы.

Таким образом, можно сделать вывод, что в лучшем случае, данный метод просто пройдет по массиву 1 раз, ничего не поменяв. Таким образом сложность алгоритма в лучшем случае - O(n). В худшем же случае, придется проходить массив n вплоть до n раз, и отсюда следует, что в худшем случае сложность будет *O(n2)* Так как сложность алгоритма смотрится по худшему случаю, то и сложность всего метода *O(n2).*

  
Рисунок 2. Блок-схема сортировки обменом

  
Рисунок 3. Результат примера работы программы сортировки выбором по возрастанию.

  
Рисунок 4. Результат примера работы программы сортировки выбором в случайном порядке.

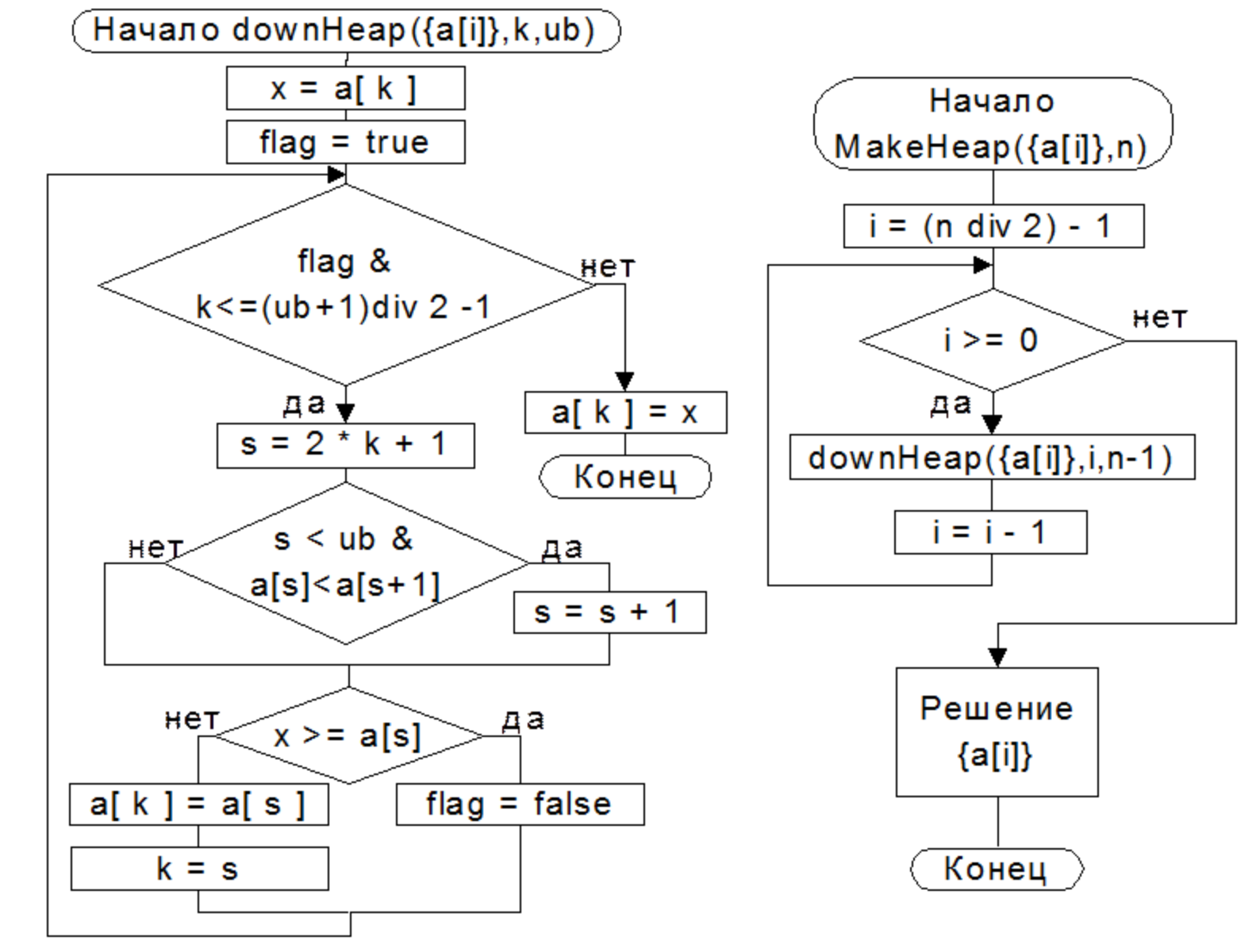
## 2. Пирамидальная сортировка

**Описание алгоритма**

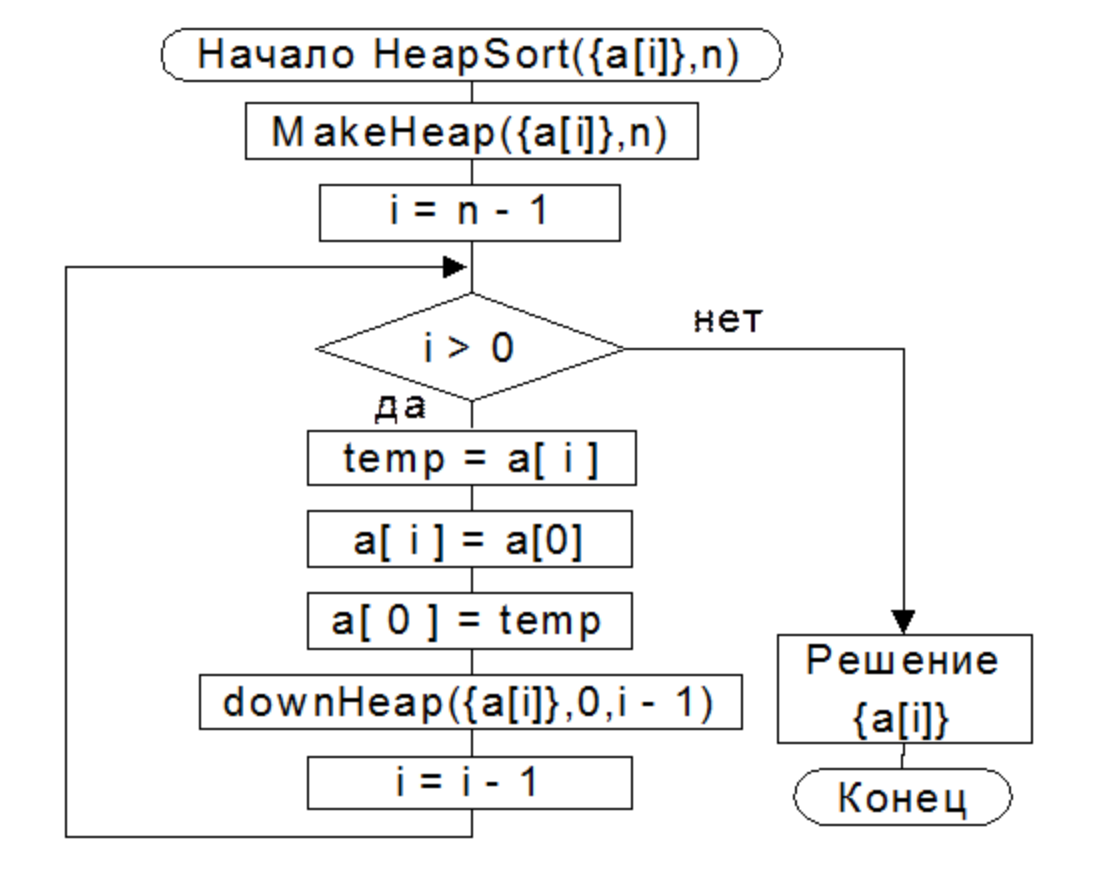
Общая идея пирамидальной сортировки заключается в том, что сначала строится пирамида из элементов исходного массива, а затем осуществляется сортировка элементов.

Выполнение алгоритма разбивается на два этапа.

1. Построение пирамиды. Определяем правую часть дерева, начиная с n/2-1 (нижний уровень дерева). Берем элемент левее этой части массива и просеиваем его сквозь пирамиду по пути, где находятся меньшие его элементы, которые одновременно поднимаются вверх; из двух возможных путей выбираем путь через меньший элемент.

  
Рисунок 5. Блок-схема первого этапа - построения пирамиды

2. Сортировка на построенной пирамиде. Берем последний элемент массива в качестве текущего. Меняем верхний (наименьший) элемент массива и текущий местами. Текущий элемент (он теперь верхний) просеиваем сквозь n-1 элементную пирамиду. Затем берем предпоследний элемент и т.д.

  
Рисунок 6. Блок-схема второго этапа - сортировка по построенной пирамиде.

Данный алгоритм имеет O(𝑛 log𝑛) сложность при лучшем, среднем и худшем случае.

## 3. Сравнение сортировок

Таблица 1. Количество сравнений сортировки выбором

| Размер массива | | По возрастанию | По убыванию | В случайном порядке |
| --- | --- | --- | --- | --- |
| 10 | Количество сравнений | 45 | 45 | 45 |
| Количество присваиваний | 0 | 9 | 7 |
| Количество операций | 45 | 54 | 52 |
| 20 | Количество сравнений | 190 | 190 | 190 |
| Количество присваиваний | 0 | 19 | 16 |
| Количество операций | 190 | 209 | 206 |
| 50 | Количество сравнений | 1225 | 1225 | 1270 |
| Количество присваиваний | 0 | 49 | 45 |
| Количество операций | 1225 | 1274 | 1270 |
| 100 | Количество сравнений | 4950 | 4950 | 4950 |
| Количество присваиваний | 0 | 99 | 94 |
| Количество операций | 4950 | 5049 | 5044 |

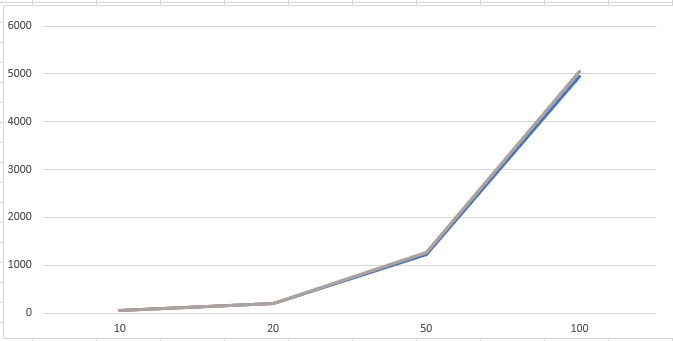
  
Рисунок 7. График сортировки обменом

Таблица 2. Количество сравнений пирамидальной сортировки.

| Размер массива | | По возрастанию | По убыванию | В случайном порядке |
| --- | --- | --- | --- | --- |
| 10 | Количество операций | 79 | 103 | 89 |
|
|
| 20 | Количество операций | 231 | 283 | 245 |
|
|
| 50 | Количество операций | 795 | 976 | 890 |
|
|
| 100 | Количество операций | 1978 | 2363 | 2178 |
|
|

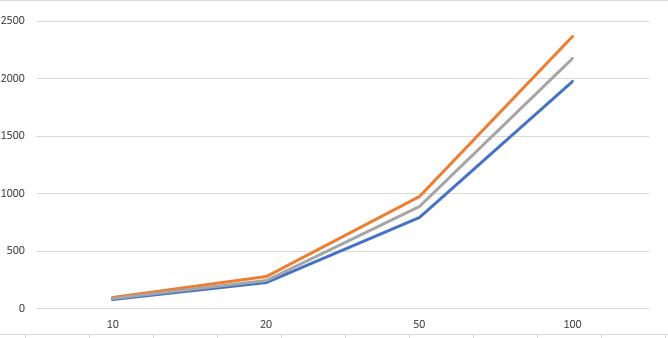


Рисунок 8. График сортировки пирамидальной сортировки

**Вывод**: Проведя анализ и вычислительные опыты были составлены таблицы значений и графики, отражающие зависимость количества проводимых операций от количества входного размера массива. Сравнив графики, можно заметить, что на больших массивах это соотношение лучше у пирамидальной сортировки, т.к. она делает заметно меньше операций. На малых значениях данная сортировка делает существенно больше операций, даже на отсортированных массивах. Для сортировки выбором можем наблюдать ровно обратную ситуацию. На малых значениях она показывает себя лучше всего, особенно если массив уже отсортирован и не будет производиться ни одного присваивания. Однако с увеличением массива входных данных, количество производимых операций резко увеличивается.

**Листинг сортировки выбором**

#include <iostream>

#include <ctime>

using namespace std;

int CmpCount=0;

int MovCount=0;

void selectionSort(int \*num, const int size)

{

int min, temp;

for (int i = 0; i < size - 1; i++)

{

min = i;

for (int j = i + 1; j < size; j++)

{

if (num[j] < num[min]) {

min = j;

CmpCount++;

}

else { CmpCount++; }

}

if (min == i) continue;

temp = num[i];

num[i] = num[min];

num[min] = temp;

MovCount++;

}

}

void Input(int \*arr, const int n)

{

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

{

arr[i] = rand() % 99 + 1;

}

}

void InputSort(int \*arr, const int n)

{

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

{

arr[i] = i;

}

}

void InputUnsort(int\* arr, const int n)

{

for (int i = n - 1, j=0; i >= 0;j++, i--)

{

arr[i] = j;

}

}

int main()

{

//setlocale(LC\_ALL, "RU");

const int rm = 10;

int a[rm];

InputSort(a, rm);

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

{

cout << "a[" << i << "] = " << a[i] << endl;

}

//for (int i = 0; i < rm; i++)

//{

//cout << "a[" << i << "] = " ;

//cin >> a[i];

//}

selectionSort(a, rm);

cout << endl;

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

{

cout << "a[" << i << "] = " << a[i] << endl;

}

cout << endl << "Количество сравнений: " << CmpCount << endl;

cout << "Количество присваиваний: " << MovCount << endl;

return 0;

}

**Конец листинга**

**Листинг пирамидальной сортировки**

#include <iostream>

#define N 1

using namespace std;

#ifndef BASE\_TYPE\_H

#define BASE\_TYPE\_H

unsigned long CmpCount;

unsigned long MovCount;

void ClearCount() {

CmpCount = 0;

MovCount = 0;

}

unsigned long GetAllCount() {

return CmpCount + MovCount;

}

typedef int Key;

typedef int Data;

struct BaseType {

Key key;

Data data;

BaseType& operator=(const BaseType& Var) {

key = Var.key;

data = Var.data;

MovCount++;

return \*this;

}

bool operator<(const BaseType& Var) const {

CmpCount++;

return(key < Var.key);

}

bool operator<=(const BaseType& Var) const {

CmpCount++;

return(key <= Var.key);

}

bool operator>(const BaseType& Var) const {

CmpCount++;

return(key > Var.key);

}

bool operator>=(const BaseType& Var) const {

CmpCount++;

return(key >= Var.key);

}

bool operator==(const BaseType& Var) const {

CmpCount++;

return(key == Var.key);

}

bool operator!=(const BaseType& Var) const {

CmpCount++;

return(key != Var.key);

}

};

void RandArray(BaseType\* a, long n) {

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

a[i].key = rand() % n;

a[i].data = i;

}

}

void BestArray(BaseType\* a, long n) {

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

a[i].key = i;

a[i].data = i;

}

}

void WorstArray(BaseType\* a, long n) {

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

a[i].key = n - i;

a[i].data = i;

}

}

#endif

template<class type>

void Swap(type& x, type& y) {

type temp = x;

x = y;

y = temp;

}

template<class type>

void heapify(type\* a, long n, int i) {

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < n && a[l] > a[largest])

largest = l;

if (r < n && a[r] > a[largest])

largest = r;

if (largest != i) {

Swap(a[i], a[largest]);

heapify(a, n, largest);

}

}

template<class type>

void heapSort(type\* a, long n) {

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

heapify(a, n, i);

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

{

Swap(a[0], a[i]);

heapify(a, i, 0);

}

}

int main() {

//int testArray[N] = { 10, 20, 50, 100, 200, 300, 400, 500, 600, 700 };

int testArray[N] = { 10};

BaseType\* RandomArray[N];

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

RandomArray[i] = new BaseType[testArray[i]];

RandArray(RandomArray[i], testArray[i]);

}

BaseType\* a;

cout << "\nHeapSort:\n";

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

a = new BaseType[testArray[i]];

cout << "Array Size: " << testArray[i] << endl;

for (int j = 0; j < testArray[i]; j++) a[j] = RandomArray[i][j];

ClearCount();

heapSort(a, testArray[i]);

cout << "\nAverage way: " << GetAllCount() << endl;

WorstArray(a, testArray[i]);

ClearCount();

heapSort(a, testArray[i]);

cout << "Best way: " << GetAllCount() << endl;

BestArray(a, testArray[i]);

ClearCount();

heapSort(a, testArray[i]);

cout << "Worst way: " << GetAllCount() << endl << endl;

delete a;

}

for (int i = 0; i < N; i++) delete RandomArray[i];

return 0;

}

**Конец листинга**