[1. Структуры данных: двоичная куча (binary heap)](#_Toc515771112)

[1.1. Введение](#_Toc515771113)

[1.2. Реализация двоичной кучи (binary heap)](#_Toc515771114)

[1.2.1. Реализация класса кучи](#_Toc515771115)

[1.2.2. Конструктор кучи](#_Toc515771116)

[1.2.3. Добавление элемента кучи](#_Toc515771117)

[1.2.4. Вывод элементов кучи](#_Toc515771118)

[1.2.5. Упорядочение кучи](#_Toc515771119)

[1.2.6. Удаление вершины кучи (максимального элемента)](#_Toc515771120)

[1.3. Примеры бинарной кучи](#_Toc515771121)

[1.3.1. Бинарная куча #01. Реализация на C (arry)](#_Toc515771122)

[1.3.2. Бинарная куча #02. Реализация на C++ (class)](#_Toc515771123)

[1.3.3. Бинарная куча #03. Реализация на C++ (class)](#_Toc515771124)

[1.3.4. Бинарная куча #04. Реализация на C (struct)](#_Toc515771125)

[1.3.5. Бинарная куча #05. Реализация на C (struct) (#04=#05)](#_Toc515771126)

[1.3.6. Бинарная куча #06. Реализация на C++ (class) (#03=#06)](#_Toc515771127)

[1.3.7. Бинарная куча #07. Лабораторная работа № 12](#_Toc515771128)

**https://prog-cpp.ru/data-heap/**

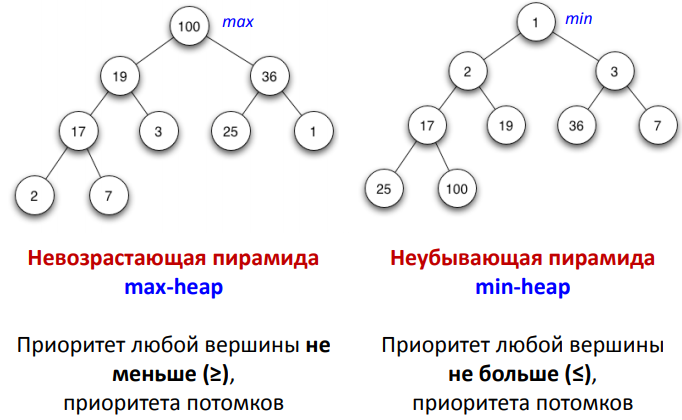
1. Структуры данных: двоичная куча (binary heap)

**Двоичная куча (binary heap)** –структура данных, позволяющая быстро (за логарифмическое время) добавлять элементы и извлекать элемент с максимальным приоритетом (например, максимальный по значению).

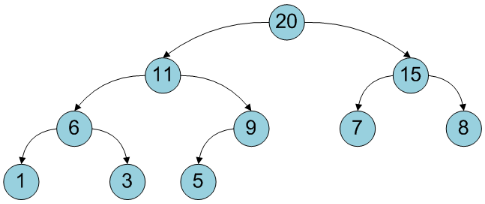
## Введение

Бинарная куча (пирамида, сортирующее дерево, binary heap) – это двоичное дерево, удовлетворяющее следующим условиям:

* Приоритет любой вершины не меньше ( ≥ ), приоритета ее потомков
* Дерево является полным двоичным деревом (complete binary tree) – все уровни заполнены слева направо (возможно за исключением последнего)



Двоичная куча представляет собой полное бинарное дерево, для которого выполняется основное свойство кучи: **приоритет каждой вершины больше приоритетов её потомков**. В простейшем случае приоритет каждой вершины можно считать равным её значению. В таком случае структура называется max-heap, поскольку корень поддерева является максимумом из значений элементов поддерева. Здесь для простоты используется именно такое представление. Напомним также, что **дерево называется полным бинарным, если у каждой вершины есть не более двух потомков, а заполнение уровней вершин идет сверху вниз (в пределах одного уровня – слева направо).**



**Двоичную кучу удобно хранить в виде одномерного массива, причем левый потомок вершины с индексом i имеет индекс 2\*i+1, а правый 2\*i+2. Корень дерева – элемент с индексом 0.** Высота двоичной кучи равна высоте дерева, то есть log2 N, где N – количество элементов массива.

## Реализация двоичной кучи (binary heap)

**Полный код примера приведен в разделе «Бинарная куча #02. Реализация на C++ (class)»**

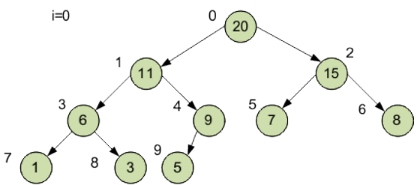
Двоичная куча представляет собой полное бинарное дерево, для которого выполняется основное свойство кучи: **приоритет каждой вершины больше приоритетов её потомков**.

В простейшем случае приоритет каждой вершины можно считать равным её значению. В таком случае структура называется **max-куча**, поскольку корень поддерева является максимумом из значений элементов поддерева.

В качестве альтернативы, если сравнение перевернуть, то наименьший элемент будет всегда корневым узлом, такие кучи называют **min-кучами**.

Двоичную кучу удобно хранить в виде одномерного массива, причем

* левый потомок вершины с индексом i имеет индекс 2\*i+1,
* правый потомок вершины с индексом i имеет индекс 2\*i+2,
* корень дерева – элемент с индексом 0,
* высота двоичной кучи равна высоте дерева, то есть log2 N, где N – количество элементов массива



Корень дерева (кучи) – элемент с индексом 0.

Высота двоичной кучи равна высоте дерева, то есть

**log2 (N+1)↑**,

где **N** – количество элементов массива, **↑** – округление в большую сторону до ближайшего целого.

Для представленной кучи

**log2 (10+1)↑ = 3,46↑ = 4**

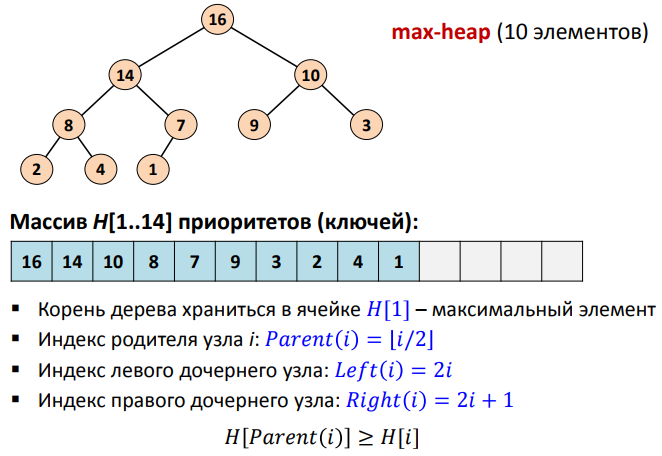
Способ построить кучу из неупорядоченного массива – это по очереди добавить все его элементы. Временная оценка такого алгоритма оценивается как

**N·log2N**.

**Можно построить кучу за N шагов.** Для этого сначала следует построить дерево из всех элементов массива, не заботясь о соблюдении основного свойства кучи, а потом вызвать метод упорядочения для всех вершин, у которых есть хотя бы один потомок (так как поддеревья, состоящие из одной вершины без потомков, уже упорядочены).

Потомки гарантированно есть у первых **heapSize/2** вершин, где heapSize – размер кучи.

**Реализация бинарной кучи на основе массива**



### Реализация класса кучи

//Реализация класса кучи

class Heap {

static const int SIZE = 100; // максимальный размер кучи

int \*h; // указатель на массив кучи

int HeapSize; // размер кучи

public:

Heap(); // конструктор кучи

void addelem(int); // добавление элемента кучи

void outHeap(); // вывод элементов кучи в форме кучи

void out(); // вывод элементов кучи в форме массива

int getmax(); // удаление вершины (максимального элемента)

void heapify(int); // упорядочение кучи

};

### Конструктор кучи

//Конструктор кучи

Heap :: Heap() {

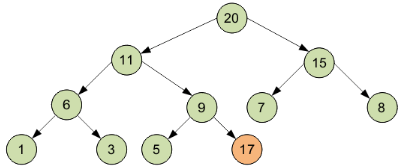
h = new int[SIZE];

HeapSize = 0;

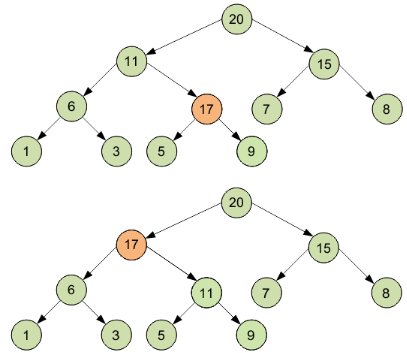
}

### Добавление элемента кучи

Новый элемент добавляется на последнее место в массиве, то есть позицию с максимальным индексом.



Возможно, что при этом будет нарушено основное свойство кучи, так как новый элемент может быть больше родителя. В таком случае новый элемент «поднимается» на один уровень (менять с вершиной-родителем) до тех пор, пока не будет соблюдено основное свойство кучи.



Иначе говоря, новый элемент «всплывает», «проталкивается» вверх, пока не займет свое место. Сложность алгоритма не превышает высоты двоичной кучи (так как количество «подъемов» не больше высоты дерева), то есть равна O(log2 N).

void Heap :: addelem(int n) {

int i, parent;

i = HeapSize;

h[i] = n;

parent = (i-1)/2;

while(parent >= 0 && i > 0) {

if(h[i] > h[parent]) {

int temp = h[i];

h[i] = h[parent];

h[parent] = temp;

}

i = parent;

parent = (i-1)/2;

}

HeapSize++;

}

### Вывод элементов кучи

//Вывод элементов кучи

//Вывод элементов в форме кучи

void Heap:: outHeap(void) {

int i = 0;

int k = 1;

while(i < HeapSize) {

while((i < k) && (i < HeapSize)) {

cout << h[i] << " ";

i++;

}

cout << endl;

k = k \* 2 + 1;

}

}

//Вывод элементов кучи в форме массива

void Heap:: out(void) {

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

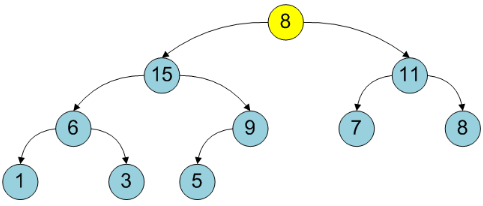
cout << h[i] << " "; }

cout << endl;

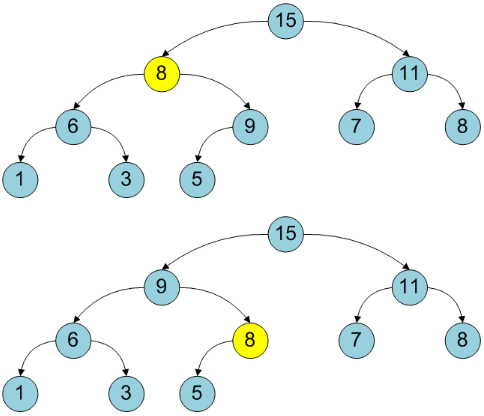
}

### Упорядочение кучи

В ходе других операций с уже построенной двоичной кучей также может нарушиться основное свойство кучи: вершина может стать меньше своего потомка.



Метод heapify восстанавливает основное свойство кучи для дерева с корнем в i-ой вершине при условии, что оба поддерева ему удовлетворяют. Для этого необходимо «опускать» i-ую вершину (менять местами с наибольшим из потомков), пока основное свойство не будет восстановлено (процесс завершится, когда не найдется потомка, большего своего родителя). Нетрудно понять, что сложность этого алгоритма также равна O(log2 N).



//Упорядочение кучи

void Heap:: heapify(int i) {

int left, right;

int temp;

left = 2\*i+1;

right = 2\*i+2;

if(left < HeapSize) {

if(h[i] < h[left]) {

temp = h[i];

h[i] = h[left];

h[left] = temp;

heapify(left);

}

}

if(right < HeapSize) {

if(h[i] < h[right]) {

temp = h[i];

h[i] = h[right];

h[right] = temp;

heapify(right);

}

}

}

В упорядоченном max-heap максимальный элемент всегда хранится в корне. Восстановить упорядоченность двоичной кучи после удаления максимального элемента можно, поставив на его место последний элемент и вызвав метод упорядочения для корня, то есть упорядочив все дерево.

### Удаление вершины кучи (максимального элемента)

//Удаление вершины кучи (максимального элемента)

int Heap:: getmax(void) {

int x;

x = h[0];

h[0] = h[HeapSize-1];

HeapSize--;

heapify(0);

return(x);

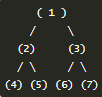
}

## Примеры бинарной кучи

### Бинарная куча #01. Реализация на C (arry)

**Этот пример и в разделе «Бинарная куча #07. Лабораторная работа № 12»**

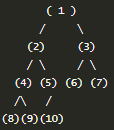
Для начала разберёмся, что же такое бинарная куча. Рассмотрим на примере:



Нам дано дерево с корнем в вершине с номером 1. У вершины 1 два ребёнка: 2 и 3. У вершины 2 и у вершины 3 тоже по два ребёнка (4;5 и 6;7 соответственно). Каждый элемент кучи состоит из двух составляющих: номер элемента и его значение. Пускай для простоты в нашей куче номера и будут являться значениями. Итак, основные правила кучи:

* У каждой вершины должно быть ровно по 2 ребёнка
* Каждый ребёнок “хуже” своего отца.

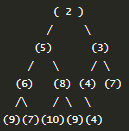
Исключением является нижний ряд кучи - у этих вершин нет детей. В тоже время, нижний ряд не обязательно будет полностью заполнен - элементов может просто не хватить. В таком случае может случиться так, что у одной вершины предпоследнего ряда не будет хватать одного ребёнка. Вот пример:



Как видите, у вершины 5 всего один ребёнок (10) , но, тем не менее, данная куча правильна.

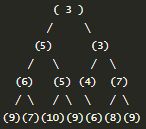
Теперь разберёмся, что же значит “хуже”. В нашем случае, “хуже” означает больше. Т.е. дети по значению больше отца, а отец в нашей куче всегда меньше своих детей. И действительно, 1<2 и 1<3 (вершина 1 - отец) ; 2<4 и 2<5 ; 3<6 и 3<7 и т.д. Параметр “хуже” в различных задачах может быть различным, это зависит от того, что мы хотим получить. Дети по значению могут быть больше или меньше своих родителей. Т.е. “хуже” будет либо меньше, либо больше. В других задачах параметр “хуже” может быть и другим…

Итак, посмотрите на следующую кучу и скажите, правильная ли она:

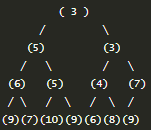


В любом случае, ответ давать нельзя, пока не известен параметр сравнивая детей с родителями. Пускай в данной куче ребёнок должен быть больше либо равен своему отцу. А вот теперь ответ однозначный - данная куча правильная. Почему же? Ведь левый ребёнок вершины со значением 2 - 5 и он меньше правого ребёнка (3)! Это, конечно, так, но в правилах кучи нет никаких правил сравнивания детей! Главное, чтобы оба ребёнка были “хуже” своего родителя, а как они относятся между собой и на каких позициях стоят - всё равно…

С этой кучей разобрались, для закрепления знаний давайте рассмотрим ещё одну кучу:



Параметр “хуже” у этой кучи такой же, как и у предыдущей кучи. Но эта куча неправильна, т.к. у вершины со значением 4 всего один ребёнок. Исправим эту кучу так, чтобы она стала правильной:



Вот теперь эта куча правильная…

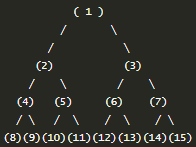
Теперь разберёмся с нумерацией вершин. Вернёмся к первым двум пирамидам - у них значения совпадают с номерами вершин. На них можно увидеть, что нумерация вершин идёт слева направо. Это как будто мы выписываем все номера от 1 до количества вершин в линейку и в определённый момент строчка кончается - мы переходим на новую. Причём, как вы уже заметили, на каждом новом уровне (на строке) будет ровно 2 в степени (i-1) вершин, где i - номер уровня.

Теперь мы знаем основные правила кучи, знаем, как нумеруются вершины. Но как же хранить кучу в памяти компьютера? А вот для этого мы и учились правильно расстанавливать номера. При правильной расстановке у каждого элемента будет свой уникальный номер - почему бы нам ни хранить кучу линейным массивом? Запишем в линейный массив нашу последнюю кучу:



Вот и прекрасно. Советую вам самостоятельно попробовать восстановить бинарную кучу из этого массива. Ну как, получилось? Если не получилось, желательно всё-таки разобраться с этим моментом (попробовать самому записать кучу в линейный массив и восстановить из него). А мы идём дальше.

При работе с бинарной кучей очень часто нужно узнать ребёнка/отца данной вершины. На рисунке всё это хорошо видно, но как же быть, если куча записана в линейный массив? Конечно, можно восстановить рисунок, и по нему посмотреть нужные данные. Но от этого легче не станет - компьютер сам не сможет посмотреть на картинку и сказать, кто отец, а кто ребёнок, а каждый раз спрашивать человека не получится… В этом случае нам помогает одно интересное свойство кучи: **k1=2*i*** *и* ***k2=2*i+1** (где i - номер вершины отца, k1 и k2 - номера вершин детей). По построению это свойство выполняется всегда. Можно посмотреть на примере кучи номеров:



Свойство проверить на любой вершине куче. Например, найдём детей вершины 6 => i = 6 => k1 = 2 \* i = 2 \* 6 = 12, k2 = 2 \* i + 1 = 2 \* 6 + 1 = 13 => дети вершины с номером 6 имеют номера 12 и 13. Это действительно так! :) Можете проверить на любой вершине - свойство выполнится. Однако за существованием вершины с таким номером должны следить мы сами - если номер вершины превысил количество вершин в куче, то такая вершина не существует.

Мы умеем находить детей вершины. А как же найти родителя? Очень просто! **pr = i div 2** (номер отца вершины равен номеру ребёнка, целочисленно делённому на 2). Это свойство обратно предыдущему, проверьте его самостоятельно.

Ну вот, на этом теория закончена. Теперь осталось научиться правильно реализовывать это всё на практике…

Сразу прейдём к коду и постепенно будут даваться объяснения.

Для начала объявим глобальные переменные:

#include<stdio.h>

#include<malloc.h>

#include<iostream>

constlongint **MaxV = 5000;**

longint **a[MaxV];**//массив для временного хранения элементов

longint **n,i;**//n-количество элементов; i-счётчик цикла

//Our heap variables

longint **heap[MaxV];** //основной массив кучи

longint **nheap, tmp;** //nheap-размер нашей кучи; tmp-временное хранение данных

**heap** - основной массив нашей кучи; **nheap** - размер нашей кучи (количество элементов, находящихся на данный момент в куче); **a** - массив для временного хранения элементов (он нужен лишь для примера работ с кучей); **n** - количество элементов, которые нужно считать (также нужно для примера); i - счётчик цикла; **tmp** - переменная, для временного хранения каких-либо данных.

Вот и прекрасно, ху из ху разобрались, теперь научимся инициализировать кучу:

//инициализация кучи

void **InitHeap(**void){

**nheap = 0;**

}

Здесь ничего сложного нет - просто обнуляем текущее количество элементов. Если хотите - можно также сделать и обнуление всего массива кучи, но это не обязательно.

Теперь научимся добавлять элементы в кучу:

void **HeapAdd(**longint **x){**

**nheap++;**

**heap[nheap] = x;**

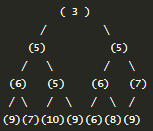
**MoveUp(nheap);**

}

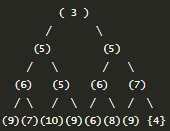
Сначала мы увеличиваем на 1 текущее количество элементов. Потом ложим на последнюю позицию элемент x.

Но ведь после того, как мы положили элемент куча может испортиться, т.е. вновь положенный элемент может быть “лучше” своего предка. В таком случае нам нужно исправить кучу. Для этого мы используем процедуру MoveUp. В неё передаётся номер добавленного элемента.

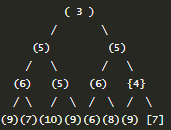
Как же должна работать MoveUp? Давайте рассуждать вместе! Для этого воспользуемся примером кучи:



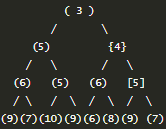
Давайте попробуем добавить в эту кучу элемент 4. Для этого находим первую свободную позицию и присваиваем этой позиции значение 4. В данном случае номер этой позиции будет ранен 15 (у вершины 7 всего один ребёнок - 9. Свободная позиция - позиция второго ребёнка):



Что мы делаем дальше? Правильно, сравниваем с отцом. В данном случае 4 лучше своего отца 7 (т.е. 4 < 7 ). В этом случае мы “поднимаем 4 выше”, т.е. меняем с отцом:



Дальше повторяем то же самое до тех пор, пока 4 не станет “хуже” своего отца:



Т.е. меняем местами вершины со значениями 4 и 5. Опять сравниваем с отцом: 4 > 3, т.е. ребёнок (4) “хуже” своего предка (3). Останавливаемся. Вот как выглядит этот код:

void **MoveUp(**longint **ind){**

longint **k;**

**k = ind / 2;** //номер вершины предка

if( ind > 1 ){ //Если ind < 2 то мы находимся в корне

if( heap[ind] < heap[k] ){

//Если предок "хуже" ребёнка - меняем их местами и

//рекурсивно пытаемся пропихнуть вершину выше

**tmp = heap[ind];**

**heap[ind] = heap[k];**

**heap[k] = tmp;**

**MoveUp(k);**

**}**

**}**

}

Сначала узнаём номер вершины предка. Если ind < 2 то мы находимся в корне - т.е. нужно выходить, т.к. у него нет предка. Если номера ребёнка и предка корректны - сравниваем их. Если предок “хуже” ребёнка - меняем их местами и рекурсивно пытаемся пропихнуть вершину выше. Для обмена элементов местами использовалась глобальная переменная tmp. Её можно было сделать и локальной.

Часто нужно достать самый “лучший” элемент из кучи и работать дальше с ним. При этом чаще всего элемент назад не возвращается, а продолжать работать с кучей приходится. Для этого также нужно исправлять кучу. Вот код получения самого “лучшего” элемента и удаления его из кучи:

longint **ExtractMin(**void){

longint **value;**

**value = heap[1];** //Самый "лучший" элемент лежит на самой верхушке кучи

**heap[1] = heap[nheap];** //последний элемент ложим на 1 место

//опускаем 1 элемент на нужное место

**heap[nheap] = 0;**

**nheap--;**

**MoveDown(1);**

return **value;**

}

Самый “лучший” элемент лежит на самой верхушке куче, т.е. является первым. Запоминаем его. Для того, чтобы кучу можно было восстановить, берём последний элемент и ложим его на 1 место. Теперь, чтобы исправить нашу кучу, нам нужно всего лишь опустить 1 элемент на нужное место. К сожалению, метод исправить кучу быстрее ещё не придуман, так что будем использовать этот. Процедура MoveDown очень похожа на MoveUp. Вот её код:

void **MoveDown(**longint **ind){**

longint **k;**

**k = ind \* 2;** //номер первого ребёнка данной вершины

if(k <= nheap){ //Если номер перевалил за количество элементов - выходим

if( (k+1 <= nheap) && (heap[k] > heap[k+1]) ) k++; //Из двух детей выбираем лучшего

if( heap[ind] > heap[k] ){ //Если предок "хуже" ребёнка - меняем их местами

//рекурсивно пытаемся опустить отца ещё ниже

**tmp = heap[ind];**

**heap[ind] = heap[k];**

**heap[k] = tmp;**

**MoveDown(k);**

**}**

**}**

}

Рассмотрим её пошагово. **k** - номер первого ребёнка данной вершины. Если номер перевалил за количество элементов - выходим, иначе k - первый кандидат на всплывание. Из двух детей должен всплыть “лучший”, иначе свойство кучи после всплывания не будет выполнятся. Из двух детей выбираем лучшего (по умолчанию - 1 ребёнок; если 2 его лучше - запоминаем его, увеличив k на 1). Теперь сравниваем кандидата на всплывание с предком. Если предок “хуже” ребёнка - меняем их местами и рекурсивно пытаемся опустить отца ещё ниже.

Ну вот и всё. Теперь вы умеете реализовывать все процедуры кучи. Сразу отмечу, что опускание/поднимание можно было сделать и не рекурсивно, а циклом. Но рекурсивное написание проще. Если Вы хотите писать бинарную как-то иначе, советую посмотреть вам и другие пособия.

А мы сейчас напишем программку-пример работы с кучей.

Как вы уже заметили, массив кучи нельзя назвать отсортированным. Многие примеры куч были таковыми, так что не думаю, что у кого-то останутся сомнения ;) (если не верите, попробуйте записать наши кучи в виде массивов и увидите, что они не отсортированы). Давайте отсортируем массив при помощи бинарной кучи. Известно, что в корне дерева кучи лежит самый “лучший” элемент, следовательно, последовательно забирая лучшие элементы мы получим отсортированный массив. А вот и код:

void **main(**void){

**setlocale(LC\_ALL,**"Rus");

**InitHeap();**

**printf(**"Insert (N)umber of elements: ");

**scanf(**"%d",&n);

**printf(**"Insert array, please:\n");

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

**printf(**"Element %2d > ", i);

**scanf(**"%d",&tmp);

**HeapAdd(tmp);**

**}**

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

**a[i] = ExtractMin();**

**}**

**printf(**"Sorted array is:\n");

for(i=0; i<n; i++) printf("%d ",a[i]);

**printf(**"\n");

**fflush(stdin);**

//getchar();

**printf(**"\n");

**system(**"PAUSE");

}

С самого начала мы инициализируем кучу. Потом считываем количество элементов массива, которые нужно будет считать. Дальше считываем массив и поэлементно заносим его в кучу. На самом деле массив нам здесь считывать не обязательно - можно просто считывать временную переменную и добавлять её значение в кучу. Т.е. данный момент кода можно заменить на:

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

**printf(**"Element %2d > ", i);

**scanf(**"%d",&tmp);

**HeapAdd(tmp);**

**}**

Дальше мы поэлементно достаём элементы из кучи в массив a и выводим уже отсортированный массив.

Как видите, ничего сложного нет. Теперь поговорим о скорости работы наших процедур:

**InitHeap** - O(1), т.к. мы делаем всего одно действие - обнуляем текущее количество вершин.

**MoveUp** - O(log N) - в самом худшем случае в куче уже будут лежать N элементов, и все они будут “хуже” нашего элемента - тогда нам нужно будет поднять элемент с самого низа до самого верха, т.е. совершить “количество уровней в данной куче” операций - это как раз и будет O(log N)…

**MoveDown** - O(log N) - аналогично MoveUp, только элемент опускается.

**HeapAdd** - O(log N) - добавление мы делаем за О(1), но поднятие элемента “стоит” O(log N) операций.

**ExtractMin** - O(log N) - узнаём и удаляем элемент за О(1), но восстановление кучи требует O(log N) операций.

Теперь поговорим о полезности данного алгоритма. **Задач типа “напишите структуру данных бинарная куча” почти не бывает. Однако есть огромное множество алгоритмов, к которым можно “прикрутить” бинарную кучу - мы с вами уже отсортировали массив с помощью бинарной кучи, хотя кучи по началу там даже видно не было. Вот ещё примеры алгоритмов, которые с бинарной кучей работают во много раз быстрее, чем без неё: алгоритм Дейкстры, Крускала, упорядоченное хранения данных и работа с ними, некоторые алгоритмы сжатия и т.д. На самом деле, бинарную кучу можно использовать почти везде, главное увидеть, как правильно в данной задаче использовать кучу и что в ней хранить.**

Ну вот мы и разобрались, что же такое бинарная куча. Теперь если вас попросят “выйти к доске и сделать кучу” вы смело её сделаете!

Полный код:

#include<stdio.h>

#include<malloc.h>

#include<iostream>

constlongint **MaxV = 5000;**

longint **a[MaxV];**//массив для временного хранения элементов

longint **n,i;**//n-количество элементов; i-счётчик цикла

//Our heap variables

longint **heap[MaxV];** //основной массив кучи

longint **nheap, tmp;** //nheap-размер нашей кучи; tmp-временное хранение данных

void **MoveUp(**longint **ind){**

longint **k;**

**k = ind / 2;** //номер вершины предка

if( ind > 1 ){ //Если ind < 2 то мы находимся в корне

if( heap[ind] < heap[k] ){

//Если предок "хуже" ребёнка - меняем их местами и

//рекурсивно пытаемся пропихнуть вершину выше

**tmp = heap[ind];**

**heap[ind] = heap[k];**

**heap[k] = tmp;**

**MoveUp(k);**

**}**

**}**

}

void **MoveDown(**longint **ind){**

longint **k;**

**k = ind \* 2;** //номер первого ребёнка данной вершины

if(k <= nheap){ //Если номер перевалил за количество элементов - выходим

if( (k+1 <= nheap) && (heap[k] > heap[k+1]) ) k++; //Из двух детей выбираем лучшего

if( heap[ind] > heap[k] ){ //Если предок "хуже" ребёнка - меняем их местами

//рекурсивно пытаемся опустить отца ещё ниже

**tmp = heap[ind];**

**heap[ind] = heap[k];**

**heap[k] = tmp;**

**MoveDown(k);**

**}**

**}**

}

longint **ExtractMin(**void){

longint **value;**

**value = heap[1];** //Самый "лучший" элемент лежит на самой верхушке кучи

**heap[1] = heap[nheap];** //последний элемент ложим на 1 место

//опускаем 1 элемент на нужное место

**heap[nheap] = 0;**

**nheap--;**

**MoveDown(1);**

return **value;**

}

//инициализация кучи

void **InitHeap(**void){

**nheap = 0;**

}

void **HeapAdd(**longint **x){**

**nheap++;**

**heap[nheap] = x;**

**MoveUp(nheap);**

}

void **main(**void){

**setlocale(LC\_ALL,**"Rus");

**InitHeap();**

**printf(**"Insert (N)umber of elements: ");

**scanf(**"%d",&n);

**printf(**"Insert array, please:\n");

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

**printf(**"Element %2d > ", i);

**scanf(**"%d",&tmp);

**HeapAdd(tmp);**

**}**

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

**a[i] = ExtractMin();**

**}**

**printf(**"Sorted array is:\n");

for(i=0; i<n; i++) printf("%d ",a[i]);

**printf(**"\n");

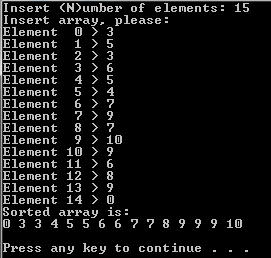
**fflush(stdin);**

//getchar();

**printf(**"\n");

**system(**"PAUSE");

}



### Бинарная куча #02. Реализация на C++ (class)

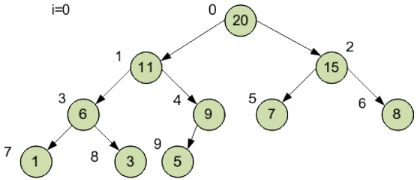
**Двоичная куча** представляет собой полное бинарное дерево, для которого выполняется основное свойство кучи: **приоритет каждой вершины больше приоритетов её потомков**.

В простейшем случае приоритет каждой вершины можно считать равным её значению. В таком случае структура называется **max-куча**, поскольку корень поддерева является максимумом из значений элементов поддерева.

В качестве альтернативы, если сравнение перевернуть, то наименьший элемент будет всегда корневым узлом, такие кучи называют **min-кучами**.

Двоичную кучу удобно хранить в виде одномерного массива, причем

* левый потомок вершины с индексом i имеет индекс 2\*i+1,
* правый потомок вершины с индексом i имеет индекс 2\*i+2.



Корень дерева (кучи) – элемент с индексом 0.

Высота двоичной кучи равна высоте дерева, то есть

**log2 (N+1)↑**,

где **N** – количество элементов массива, **↑** – округление в большую сторону до ближайшего целого.

Для представленной кучи

**log2 (10+1)↑ = 3,46↑ = 4**

Способ построить кучу из неупорядоченного массива – это по очереди добавить все его элементы. Временная оценка такого алгоритма оценивается как

**N·log2N**.

Можно построить кучу за N шагов. Для этого сначала следует построить дерево из всех элементов массива, не заботясь о соблюдении основного свойства кучи, а потом вызвать метод упорядочения для всех вершин, у которых есть хотя бы один потомок (так как поддеревья, состоящие из одной вершины без потомков, уже упорядочены).

Потомки гарантированно есть у первых **heapSize/2** вершин, где heapSize – размер кучи.

**Реализация класса кучи**

class **Heap {**

staticconstint **SIZE = 100;** // максимальный размер кучи

int **\*h;** // указатель на массив кучи

int **HeapSize;** // размер кучи

public:

**Heap();** // конструктор кучи

void **addelem(**int); // добавление элемента кучи

void **outHeap();** // вывод элементов кучи в форме кучи

void **out();** // вывод элементов кучи в форме массива

int **getmax();** // удаление вершины (максимального элемента)

void **heapify(**int); // упорядочение кучи

};

**Конструктор кучи**

Heap :: Heap() {

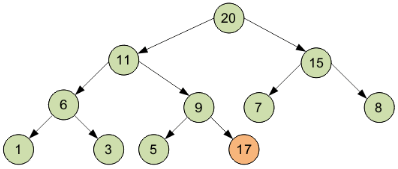
**h =** newint[SIZE];

**HeapSize = 0;**

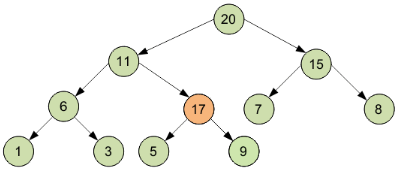
}

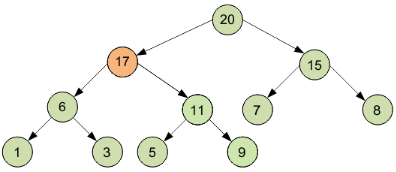
**Добавление элемента кучи**

Новый элемент добавляется на последнее место в массиве, то есть позицию с максимальным индексом.



Возможно, что при этом будет нарушено основное свойство кучи, так как новый элемент может быть больше родителя. В таком случае новый элемент «поднимается» на один уровень (менять с вершиной-родителем) до тех пор, пока не будет соблюдено основное свойство кучи.





Сложность алгоритма не превышает высоты двоичной кучи (так как количество «подъемов» не больше высоты дерева), то есть равна log2 N.

void **Heap :: addelem(**int **n) {**

// Добавление элементов в кучу

int **i, parent;**

**i = HeapSize;**

**h[i] = n;**

**parent = (i-1)/2;**

while(parent >= 0 && i > 0) {

if(h[i] > h[parent]) {

int **temp = h[i];**

**h[i] = h[parent];**

**h[parent] = temp;**

**}**

**i = parent;**

**parent = (i-1)/2;**

**}**

**HeapSize++;**

}

**Вывод элементов кучи**

*Вывод элементов в форме кучи*

void **Heap:: outHeap(**void) {

// Вывод элементов в форме кучи

int **i = 0;**

int **k = 1;**

while(i < HeapSize) {

while((i < k) && (i < HeapSize)) {

**cout << h[i] <<** " ";

**i++;**

**}**

**cout << endl;**

**k = k \* 2 + 1;**

**}**

}

*Вывод элементов кучи в форме массива*

void **Heap:: out(**void) {

// Вывод элементов кучи в форме массива

for(int **i=0; i< HeapSize; i++) {**

**cout << h[i] <<** " "; }

**cout << endl;**

}

**Упорядочение кучи**

void **Heap:: heapify(**int **i) {**

// Упорядочение кучи

int **left, right;**

int **temp;**

**left = 2\*i+1;**

**right = 2\*i+2;**

if(left < HeapSize) {

if(h[i] < h[left]) {

**temp = h[i];**

**h[i] = h[left];**

**h[left] = temp;**

**heapify(left);**

**}**

**}**

if(right < HeapSize) {

if(h[i] < h[right]) {

**temp = h[i];**

**h[i] = h[right];**

**h[right] = temp;**

**heapify(right);**

**}**

**}**

}

В упорядоченном max-heap максимальный элемент всегда хранится в корне. Восстановить упорядоченность двоичной кучи после удаления максимального элемента можно, поставив на его место последний элемент и вызвав метод упорядочения для корня, то есть упорядочив все дерево.

**Удаление вершины кучи (максимального элемента)**

int **Heap:: getmax(**void) {

// Удаление вершины кучи (максимального элемента)

int **x;**

**x = h[0];**

**h[0] = h[HeapSize-1];**

**HeapSize--;**

**heapify(0);**

return(x);

}

***Пример* Создать бинарную кучу из 16 элементов. Определить максимальный элемент.**

#include<stdio.h>

#include<malloc.h>

#include<iostream>

usingnamespace **std;**

class **Heap {**

staticconstint **SIZE = 100;** // максимальный размер кучи

int **\*h;** // указатель на массив кучи

int **HeapSize;** // размер кучи

public:

**Heap();** // конструктор кучи

void **addelem(**int); // добавление элемента кучи

void **outHeap();** // вывод элементов кучи в форме кучи

void **out();** // вывод элементов кучи в форме массива

int **getmax();** // удаление вершины (максимального элемента)

void **heapify(**int); // упорядочение кучи

};

Heap :: Heap() {

**h =** newint[SIZE];

**HeapSize = 0;**

}

void **Heap :: addelem(**int **n) {**

int **i, parent;**

**i = HeapSize;**

**h[i] = n;**

**parent = (i-1)/2;**

while(parent >= 0 && i > 0) {

if(h[i] > h[parent]) {

int **temp = h[i];**

**h[i] = h[parent];**

**h[parent] = temp;**

**}**

**i = parent;**

**parent = (i-1)/2;**

**}**

**HeapSize++;**

}

void **Heap:: outHeap(**void) {

int **i = 0;**

int **k = 1;**

while(i < HeapSize) {

while((i < k) && (i < HeapSize)) {

**cout << h[i] <<** " ";

**i++;**

**}**

**cout << endl;**

**k = k \* 2 + 1;**

**}**

}

void **Heap:: out(**void) {

for(int **i=0; i< HeapSize; i++) {**

**cout << h[i] <<** " "; }

**cout << endl;**

}

void **Heap:: heapify(**int **i) {**

int **left, right;**

int **temp;**

**left = 2\*i+1;**

**right = 2\*i+2;**

if(left < HeapSize) {

if(h[i] < h[left]) {

**temp = h[i];**

**h[i] = h[left];**

**h[left] = temp;**

**heapify(left);**

**}**

**}**

if(right < HeapSize) {

if(h[i] < h[right]) {

**temp = h[i];**

**h[i] = h[right];**

**h[right] = temp;**

**heapify(right);**

**}**

**}**

}

int **Heap:: getmax(**void) {

int **x;**

**x = h[0];**

**h[0] = h[HeapSize-1];**

**HeapSize--;**

**heapify(0);**

return(x);

}

int **main() {**

**Heap heap;**

int **k;**

**setlocale(LC\_ALL,** "Rus");

//system("chcp 1251");

**system(**"cls");

for(int **i=0; i<16; i++) {**

**cout <<** "Введите элемент " **<< i+1 <<** ": ";

**cin >> k;**

**heap.addelem(k);**

**}**

**heap.outHeap();**

**cout << endl;**

**heap.out();**

**cout << endl <<** "Максимальный элемент: " **<< heap.getmax();**

**cout << endl <<** "Новая куча:" **<< endl;**

**heap.outHeap();**

**cout << endl;**

**heap.out();**

**cout << endl <<** "Максимальный элемент: " **<< heap.getmax();**

**cout << endl <<** "Новая куча:" **<< endl;**

**heap.outHeap();**

**cout << endl;**

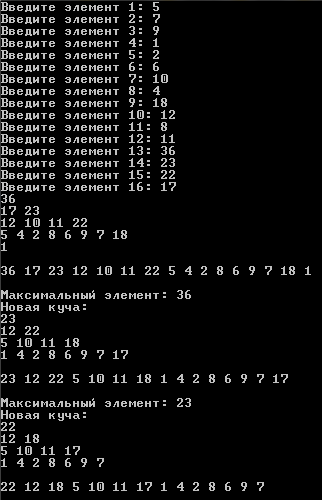
**heap.out();**

**cin.get();cin.get();**

return **0;**

}

Результат выполнения



### Бинарная куча #03. Реализация на C++ (class)

Один из алгоритмов, в котором используется бинарная куча — это [HeapSort](http://acm.mipt.ru/twiki/bin/view/Algorithms/HeapSort) — сортировка методом бинарной кучи, которая в худшем и среднем случае сортирует за время O(N log N) (напомним, что [QuickSort](http://acm.mipt.ru/twiki/bin/view/Algorithms/QuickSort) в худшем случае работает N^2).

[HeapSort](http://acm.mipt.ru/twiki/bin/view/Algorithms/HeapSort) реализован в функции main.

#include <stdio.h>

#include <malloc.h>

typedef struct {

// int value;

int key;

} ITEM;

class HEAP {

public:

ITEM \*h;

int size;

HEAP(unsigned int n) {

size = 0;

h = (ITEM\*)malloc(sizeof(ITEM) \* n);

}

~HEAP() {

if (h) free(h);

}

int add(ITEM x) {

h[++size] = x;

checkup(size);

return 1;

}

int extract\_min(ITEM \*x) {

if (size == 0) return 0;

\*x = h[1];

h[1] = h[size--];

checkdown(1);

return 1;

}

private:

void checkup(int c) {

int p;

p = c / 2;

if (p == 0)return;

if (h[p].key > h[c].key) {

ITEM tmp;

tmp = h[p]; h[p] = h[c]; h[c] = tmp;

checkup(p);

}

}

void checkdown(int p) {

int c;

c = 2 \* p;

if (c > size) return;

if (c + 1 <= size && h[c + 1].key < h[c].key) c++;

if (h[c].key < h[p].key) {

ITEM tmp;

tmp = h[c]; h[c] = h[p]; h[p] = tmp;

checkdown(c);

}

}

};

int main() {

HEAP heap(1000);

int n, i;

ITEM x;

printf("n= ");

scanf\_s("%d", &n);

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

printf("%2d x.key= ", i);

scanf\_s("%d", &x.key);

heap.add(x);

}

while (heap.extract\_min(&x)) {

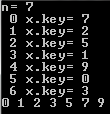
printf("%d ", x.key);

}

scanf\_s("%d", &n);

return 0;

}



### Бинарная куча #04. Реализация на C (struct)

Ниже приведен более "индустриальный код", соответствующий стандартам. В нём реализовано динамическое выделением памяти.

Заложена возможность динамического увеличения размера кучи.

#include<stdio.h>

#include<malloc.h>

#include<iostream>

typedefint **key\_t;**

typedefunsignedint **value\_t;**

typedefstruct **{**

**key\_t key;**

**value\_t value;**

} pair\_t ;

typedefstruct **{**

**pair\_t \*data;**

unsignedint **size;**

unsignedint **data\_size;**

} bheap\_t;

bheap\_t\* bheap\_new( unsignedint **initial\_data\_size ) {**

**bheap\_t \*h = (bheap\_t\*) malloc(** sizeof(bheap\_t) );

**h->data = (pair\_t\*) malloc(** sizeof(pair\_t) \* (1 + initial\_data\_size) );

**h->data\_size = 1 + initial\_data\_size;**

**h->size = 0;**

return **h;**

}

void **bheap\_delete(bheap\_t \*h) {**

if( h ) {

if( h->data ) {

**free( h->data );**

**}**

**free(h);**

**}**

}

void **bheap\_checkup( bheap\_t \*h,** unsignedint **c ) {**

unsignedint **p;**

for( p = c / 2; p > 0 ; c = p , p = c / 2 ) {

if( h->data[p].key > h->data[c].key ) {

**pair\_t tmp = h->data[p]; h->data[p] = h->data[c]; h->data[c] = tmp;**

**}** else **{**

break;

**}**

**}**

}

void **bheap\_checkdown( bheap\_t \*h,** unsignedint **p ) {**

unsignedint **c;**

for( c = 2 \* p ; c <= h->size ; p = c, c = 2 \*p ) {

if( c + 1 <= h->size && h->data[c + 1].key < h->data[c].key ) c++;

if( h->data[c].key < h->data[p].key ) {

**pair\_t tmp;**

**tmp = h->data[c]; h->data[c] = h->data[p]; h->data[p] = tmp;**

**}** else **{**

break;

**}**

**}**

}

void **bheap\_add( bheap\_t \*h, pair\_t v ) {**

if( h->size + 1 >= h->data\_size ) {

**h->data\_size \*= 2;**

**h->data = (pair\_t\*) realloc( h->data, h->data\_size \*** sizeof(pair\_t) );

**}**

**h->data[ ++(h->size) ] = v ;**

**bheap\_checkup( h, h->size );**

}

int **bheap\_extract\_min(bheap\_t \*h, pair\_t \*v) {**

if( h->size == 0 ) return **0;**

**\*v = h->data[1];**

**h->data[1] = h->data[ (h->size)--];**

**bheap\_checkdown( h, 1 );**

return **1;**

}

int **main() {**

**setlocale(LC\_ALL,**"Rus");

**bheap\_t \*h = bheap\_new(1000);**

unsignedint **n, i;**

**pair\_t v = {0, 0};**

**printf(**"Введите количество элементов n= ");

**scanf(**"%u", &n);

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

**printf(**"Элемент %2d > ", i);

**scanf(**"%d", &v.key);

**v.value = i;**

**bheap\_add(h, v);**

**}**

**printf(**"\nРезультат:\n", i);

while( bheap\_extract\_min(h, &v) ) {

**printf(**"%d ", v.key);

**}**

**printf(**"\n");

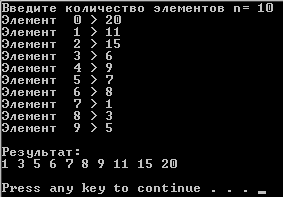
**bheap\_delete( h );**

**printf(**"\n");

**system(**"PAUSE");

return **0;**

}



### Бинарная куча #05. Реализация на C (struct) (#04=#05)

Ниже приведен более "индустриальный код", соответствующий стандартам. В нём реализовано динамическое выделением памяти.

Заложена возможность динамического увеличения размера кучи.

#include<stdio.h>

#include<malloc.h>

typedef int key\_t;

typedef unsigned int value\_t;

typedef struct {

key\_t key;

value\_t value;

} pair\_t;

typedef struct {

pair\_t \*data;

unsigned int size;

unsigned int data\_size;

} bheap\_t;

bheap\_t\* bheap\_new(unsigned int initial\_data\_size) {

bheap\_t \*h = (bheap\_t\*)malloc(sizeof(bheap\_t));

h->data = (pair\_t\*)malloc(sizeof(pair\_t) \* (1 + initial\_data\_size));

h->data\_size = 1 + initial\_data\_size;

h->size = 0;

return h;

}

void bheap\_delete(bheap\_t \*h) {

if (h) {

if (h->data) {

free(h->data);

}

free(h);

}

}

void bheap\_checkup(bheap\_t \*h, unsigned int c) {

unsigned int p;

for (p = c / 2; p > 0; c = p, p = c / 2) {

if (h->data[p].key > h->data[c].key) {

pair\_t tmp = h->data[p]; h->data[p] = h->data[c]; h->data[c] = tmp;

}

else {

break;

}

}

}

void bheap\_checkdown(bheap\_t \*h, unsigned int p) {

unsigned int c;

for (c = 2 \* p; c <= h->size; p = c, c = 2 \* p) {

if (c + 1 <= h->size && h->data[c + 1].key < h->data[c].key) c++;

if (h->data[c].key < h->data[p].key) {

pair\_t tmp;

tmp = h->data[c]; h->data[c] = h->data[p]; h->data[p] = tmp;

}

else {

break;

}

}

}

void bheap\_add(bheap\_t \*h, pair\_t v) {

if (h->size + 1 >= h->data\_size) {

h->data\_size \*= 2;

h->data = (pair\_t\*)realloc(h->data, h->data\_size \* sizeof(pair\_t));

}

h->data[++(h->size)] = v;

bheap\_checkup(h, h->size);

}

int bheap\_extract\_min(bheap\_t \*h, pair\_t \*v) {

if (h->size == 0) return 0;

\*v = h->data[1];

h->data[1] = h->data[(h->size)--];

bheap\_checkdown(h, 1);

return 1;

}

int main() {

bheap\_t \*h = bheap\_new(1000);

unsigned int n, i;

pair\_t v = { 0, 0 };

printf("n= ");

scanf\_s("%u", &n);

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

printf("%2d v.key= ", i);

scanf\_s("%d", &v.key);

v.value = i;

bheap\_add(h, v);

}

while (bheap\_extract\_min(h, &v)) {

printf("%d ", v.key);

}

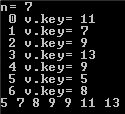
printf("\n");

bheap\_delete(h);

scanf\_s("%d", &n);

return 0;

}



### Бинарная куча #06. Реализация на C++ (class) (#03=#06)

Здесь приведена реализация структуры данных "Бинарная куча" на языках программирования C и C++ (см. далее).

Один из алгоритмов, в котором используется бинарная куча — это [HeapSort](http://acm.mipt.ru/twiki/bin/view/Algorithms/HeapSort) — сортировка методом бинарной кучи, которая в худшем и среднем случае сортирует за время O(N log N) (напомним, что [QuickSort](http://acm.mipt.ru/twiki/bin/view/Algorithms/QuickSort) в хуждем случае работает N^2).

[HeapSort](http://acm.mipt.ru/twiki/bin/view/Algorithms/HeapSort) реализован в функции main.

#include<iostream>

#include<stdio.h>

usingnamespace **std;**

typedefstruct **{**

// int value;

int **key;**

} ITEM;

class **HEAP {**

public:

**ITEM \*h;**

int **size;**

**HEAP(**unsignedint **n) {**

**size = 0;**

**h = (ITEM\*) malloc(** sizeof(ITEM) \* n);

**}**

**~HEAP() {**

if(h) free(h);

**}**

int **add(ITEM x) {**

**h[++size]=x;**

**checkup(size);**

return **1;**

**}**

int **extract\_min(ITEM \*x) {**

if(size ==0) return **0;**

**\*x = h[1];**

**h[1] = h[size--];**

**checkdown(1);**

return **1;**

**}**

private:

void **checkup(**int **c) {**

int **p;**

**p = c / 2;**

if( p == 0 )return;

if(h[p].key > h[c].key) {

**ITEM tmp;**

**tmp = h[p]; h[p] = h[c]; h[c] = tmp;**

**checkup(p);**

**}**

**}**

void **checkdown(**int **p) {**

int **c;**

**c = 2\*p;**

if( c > size ) return;

if( c+1 <= size && h[c + 1].key < h[c].key ) c++;

if( h[c].key < h[p].key ) {

**ITEM tmp;**

**tmp = h[c]; h[c] = h[p]; h[p] = tmp;**

**checkdown(c);**

**}**

**}**

};

int **main() {**

**setlocale(LC\_ALL,**"Rus");

**HEAP heap(1000);**

int **n, i;**

**ITEM x;**

**printf(**"Введите количество элементов n= ");

**scanf(**"%d", &n);

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

**printf(**"Элемент %2d > ", i);

**scanf(**"%d", &x.key);

**heap.add(x);**

**}**

**printf(**"\nРезультат:\n", i);

while( heap.extract\_min(&x) ) {

**printf(**"%2d ", x.key);

//printf("%d ", x.key);

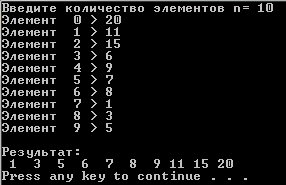
**}**

**cout <<** "\n";

**system(**"PAUSE");

return **0;**

}



### Бинарная куча #07. Лабораторная работа № 12

#include <iostream>

using namespace std;

const long int MaxV = 5000;

long int a[MaxV];

long int n, i;

long int heap[MaxV];

long int nheap, tmp;

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

void initHeap(void)

{

nheap = 0;

}

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

void moveUp(long int ind)

{

long int k;

k = ind / 2;

if (ind > 1)

{

if (heap[ind] < heap[k])

{

tmp = heap[ind];

heap[ind] = heap[k];

heap[k] = tmp;

moveUp(k);

}

}

}

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

void moveDown(long int ind)

{

long int k;

k = ind \* 2;

if (k <= nheap)

{

if ((k + 1 <= nheap) && (heap[k] > heap[k + 1]))

k++;

if (heap[ind] > heap[k])

{

tmp = heap[ind];

heap[ind] = heap[k];

heap[k] = tmp;

moveDown(k);

}

}

}

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

void heapAdd(long int x)

{

nheap++;

heap[nheap] = x;

moveUp(nheap);

}

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

long int extractMin(void)

{

long int value;

value = heap[1];

heap[1] = heap[nheap];

heap[nheap] = 0;

nheap--;

moveDown(1);

return value;

}

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

void main(void)

{

setlocale(LC\_ALL, "Russian");

initHeap();

cout << "Введите количество элементов ";

cin >> n;

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

{

cout << "Введите " << i + 1 << " элемент= ";

cin >> tmp;

heapAdd(tmp);

}

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

{

a[i] = extractMin();

}

cout << "Результат= ";

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

cout << a[i] << ' ';

fflush(stdin);

getchar();

}

