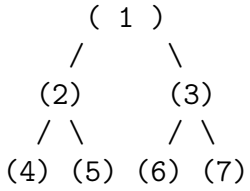
Сегодня, как ни странно, мы будем делать кучу! 🙄 Причём не обычную, а бинарную! Для начала разберёмся, что же такое бинарная куча. Рассмотрим на примере:

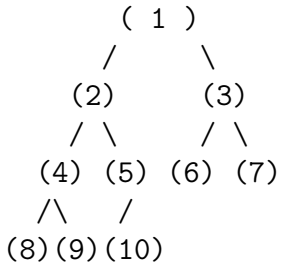


Нам дано дерево с корнем в вершине с номером 1. У вершины 1 два ребёнка: 2 и 3. У вершины 2 и у вершины 3 тоже по два ребёнка (4;5 и 6;7 соответственно). Каждый элемент кучи состоит из двух составляющих: номер элемента и его значение. Пускай для простоты в нашей куче номера и будут являться значениями.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

1) У каждой вершины должно быть ровно по 2 ребёнка

2) Каждый ребёнок "хуже" своего отца.

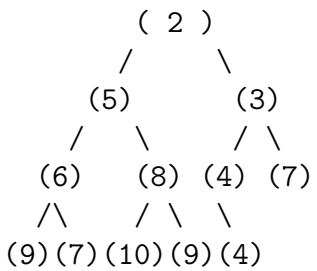
Исключением является нижний ряд кучи - у этих вершин нет детей. В тоже время, нижний ряд не обязательно будет полностью заполнен - элементов может просто не хватить. В таком случае может случиться так, что у одной вершины предпоследнего ряда не будет хватать одного ребёнка. Вот пример:



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

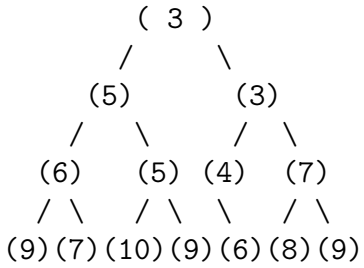
Теперь разберёмся, что же значит "хуже". В нашем случае, "хуже" означает больше. Т.е. дети по значению больше отца, а отец в нашей куче всегда меньше своих детей. И действительно, 1<2 и 1<3 (вершина 1 - отец) ; 2<4 и 2<5 ; 3<6 и 3<7 и т.д. Параметр "хуже" в различных задачах может быть различным, это 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:

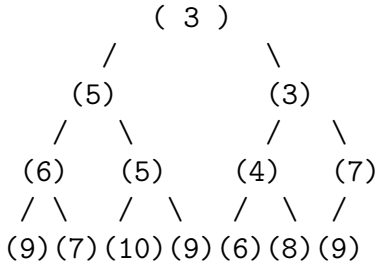


Итак, что вы скажете, правильная ли это куча? В любом случае, ответ давать нельзя, пока не известен параметр сравнивая детей с родителями. Пускай в данной куче ребёнок должен быть больше либо равен своему отцу. А вот теперь ответ однозначный - данная куча правильная. Почему же? Ведь левый ребёнок вершины 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...

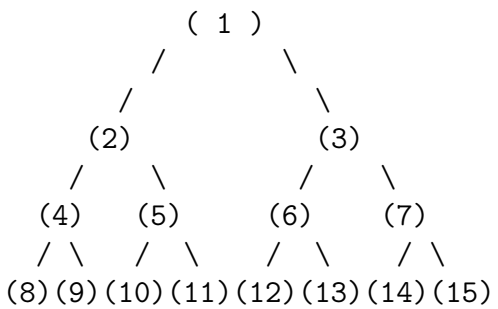
Теперь разберёмся с нумерацией вершин. Вернёмся к первым двум пирамидам - у них значения совпадают с номерами вершин. На них можно увидеть, что нумерация вершин идёт слева направо. Это как будто мы выписываем все номера от 1 до количества вершин в линейку и в определённый момент строчка кончается - мы 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:

Номер:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Значение:	3	5	3	6	5	4	7	9	7	10	9	6	8	9	0

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:

C++ код:

```
#include <stdio.h>

const long int MaxV = 5000;

long int a[MaxV];
long int n,i;

//Our heap variables
long int heap[MaxV];
long int nheap, tmp;
```

Delphi код:

```
const
  MaxV=5000;
var
 a: array [1..MaxV] of longint;
 n,i: longint;
 {Our heap variables}
 Heap: array [1..MaxV] of longint;
 nheap, tmp: longint;
```

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 ис ху разобрались, теперь научимся инициализировать кучу:

C++ код:

```
void InitHeap(void){
  nheap = 0;
}
```

Delphi код:

```
procedure InitHeap;
begin
  nheap := 0;
end;
```

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:

C++ код:

```
void HeapAdd(long int x){
  nheap++;
  heap[nheap] = x;
  MoveUp(nheap);
}
```

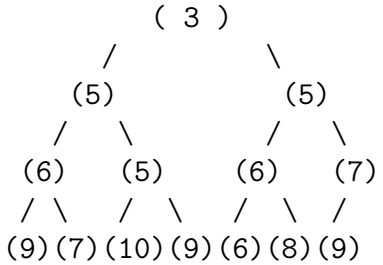
Delphi код:

```
procedure HeapAdd(x: longint);
begin
  inc(nheap);
  heap[nheap] := x;
  MoveUp(nheap);
end;
```

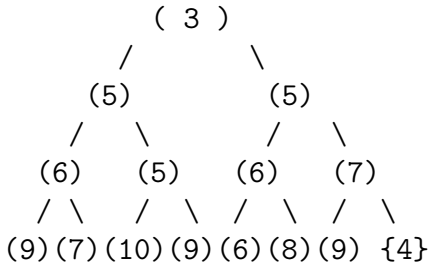
Сначала мы увеличиваем на 1 текущее количество элементов. Потом ложим на последнюю позицию элемент x.

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.

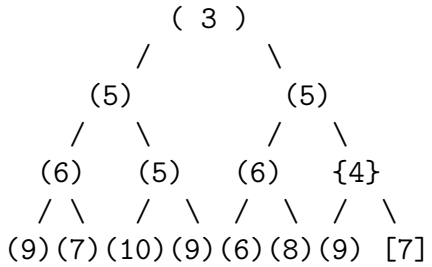
Как же должна работать MoveUp? Давайте рассуждать вместе! Для этого воспользуемся примером кучи:



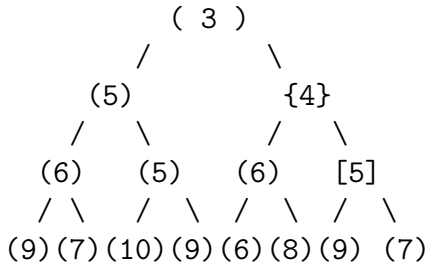
Давайте попробуем добавить в эту кучу элемент 4. Для этого находим первую свободную позицию и присваиваем этой позиции значение 4. В данном случае номер этой позиции будет ранен 15 (у вершины 7 всего один ребёнок - 9. Свободная позиция - позиция второго ребёнка):



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:

C++ код:

```
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);
    }
  }
}
```

Delphi код:

```
procedure MoveUp(ind: longint);
var
  k: longint;
begin
  k := ind div 2;
  if ind > 1 then
  begin
    if heap[ind] < heap[k] then
    begin
      tmp := heap[ind];
      heap[ind] := heap[k];
      heap[k] := tmp;
      MoveUp(k);
```

```
    end;
  end;
end;
```

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:

C++ код:

```
long int ExtractMin(void){
  long int value;
  value = heap[1];
  heap[1] = heap[nheap];
  heap[nheap] = 0;
  nheap--;
  MoveDown(1);
  return value;
}
```

Delphi код:

```
function ExtractMin: longint;
var
  value: longint;
begin
  value := heap[1];
  heap[1] := heap[nheap];
  heap[nheap] := 0;
  dec(nheap);
  MoveDown(1);
  ExtractMin := value;
end;
```

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:

C++ код:

```
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);
    }
  }
}
```

Delphi код:

```
procedure MoveDown(ind: longint);
var
  k: longint;
begin
  k := ind*2;
  if k <= nheap then
  begin
    if (k+1 <= nheap) and (heap[k] > heap[k+1]) then
      k := k + 1;
    if heap[ind] > heap[k] then
    begin
      tmp := heap[ind];
      heap[ind] := heap[k];
      heap[k] := tmp;
      MoveDown(k);
    end;
  end;
end;
```

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:

C++ код:

```
void main(void){
  InitHeap();
  printf("Insert (N)umber of elements:\n");
  scanf("%d",&n);
  printf("Insert array, please:\n");
  for(i=0; i<n; 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();
}
```

Delphi код:

```
begin
  InitHeap;
  writeln('Insert (N)umber of elements:');
  readln(n);
  writeln('Insert array, please:');
  for i:=1 to n do
  begin
    Read(tmp);
    HeapAdd(tmp);
  end;
  for i:=1 to n do
    a[i] := ExtractMin;
  writeln('Sorted array is:');
  for i:=1 to n do
    write(a[i], ' ');
  writeln;
  readln;
end.
```

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:

C++ код:

```
  ...
  for(i=0; i<n; i++){
    scanf("%d",&a[i]);
    HeapAdd(a[i]);
  }
  ...
```

Delphi код:

```
...
 for i:=1 to n do
 begin
   Read(a[i]);
   HeapAdd(a[i]);
 end;
 ...
```

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 O(1), но поднятие элемента "стоит" O(log N) операций.

ExtractMin - O(log N) - узнаём и удаляем элемент за O(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!