**Куча (heap)**

**Посмотрите видео-объяснение темы**

Преподаватель Фоксфорда всё подробно расскажет и покажет. Чтобы посмотреть,

Рассмотрим структуру данных, которая поддерживает следующие операции:

1. Добавить элемент в структуру данных.
2. Извлечь из структуры данных наибольший (вариант - наименьший) элемент. Извлеченный элемент удаляется из структуры

При этом в структуре могут храниться одинаковые элементы.

Если реализовать такую структуру на базе списка, то добавлять элементы можно в конец списка  . Но поиск наибольшего элемента будет занимать много времени, так как придется просматривать все элементы списка и выбирать из них наибольший.

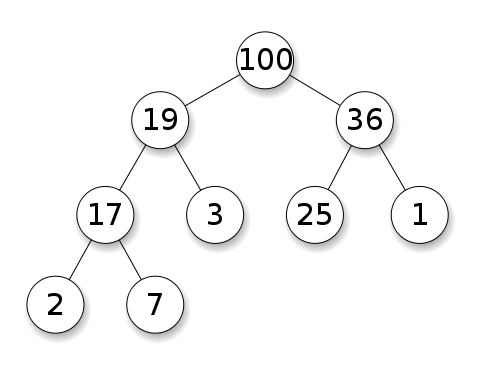
Если же хранить элементы в списке упорядочив их по неубыванию, то извлечение наибольшего будет занимать тоже много времени , так как придется сдвигать элементы, уже находящиеся в списке.

Специальная структура данных «Куча» (англ. heap) позволяет эти операции выполнять за меньшее время.

В куче элементы хранятся в виде двоичного дерева, то есть у элементов есть два потомка - левый и правый. В вершине кучи находится один элемент, у него - два потомка на следующем уровне, у них, в свою очередь, по два потомка на третьем уровне (итого - 4 элемента на третьем уровне) и т. д. Уровни заполняются в порядке увеличения номера уровня, а сам уровень заполняется слева направо. У элементов последнего уровня нет ни одного потомка, возможно, что и у некоторых элементов предпоследнего уровня нет потомков. Также в куче может быть один элемент, у которого только один потомок (левый).

При этом для элементов кучи верно следующее свойство - каждый из элементов кучи большее или равен всех своих потомков. В частности это означает, что в вершине кучи хранится наибольший элемент.

На картинке приведен пример правильной кучи из 9 элементов.



Удобно элементы кучи хранить в списке, начиная с корневого элемента. Для простоты нумерации пропустим нулевой элемент списка, то есть вершина кучи будет храниться в элементе списка с индексом 1. Остальные элементы кучи хранятся подряд в элементах списка с индексами 2, 3, 4 и т. д. То есть для примера выше:

**H[1] == 100**

**H[2] == 19**

**H[3] == 36**

**H[4] == 17**

**H[5] == 3**

**H[6] == 25**

**H[7] == 1**

**H[8] == 2**

**H[9] == 7**

Легко видеть, что у элемента H[i] левым потомком является элемент H[2\*i], а правым потомком - элемент H[2\*i+1]. А родителем элемента H[i] является элемент H[i//2].

**Добавление элемента в кучу**

Элемент добавляется в кучу следующим образом. Сначала он добавляется в самый конец кучи, то есть становится последним элементов (это можно сделать при помощи метода append списка). При этом возможно нарушение главного свойства кучи (каждый элемент больше своих потомков). Свойство могло нарушиться для элемента, который является родителем добавляемого. В этом случае нужно поменять элемент с его родителем. Процесс нужно повторять до тех пор, пока условие нарушается, то есть у добавленного элемента есть родитель (то есть элемент не корневой) и этот родитель меньше добавляемого. То есть добавляемый элемент «поднимается» вверх к вершине кучи, пока не займет надлежащее место.

Реализация соответствующего алгоритма (чуть более оптимизированная) приведена ниже:

**def add(Heap, elem):**

**Heap.append(elem)**

**i = len(Heap) - 1**

**while i > 1 and Heap[i // 2] < elem:**

**Heap[i] = Heap[i // 2]**

**i //= 2**

**Heap[i] = elem**

В функцию add передается два параметра — список, в котором хранится куча (Heap) и добавляемый элемент (elem). Сначала в конец кучи добавляется новый элемент, переменной i присваивается индекс добавленного элемента. Затем все предки добавленного элемента должны сдвинуться вниз, если они были меньше добавленного элемента. Это реализовано в цикле while. После окончания цикла на место элемента i записывается значение elem.

**Удаление элемента из кучи**

Из кучи можно удалить наибольший элемент, то есть тот, который хранится в вершине куче. На его место нужно поставить какой-нибудь элемент кучи. Поставим последний элемент кучи, удалив его из конца. Теперь в вершине кучи может нарушиться свойство кучи, значит, верхний элемент нужно сместить вниз, обменяв его с одним из своих потомков. При этом из двух потомков нужно выбрать наибольший и если этот наибольший потомок больше стоящего в вершине кучи, обменяем их местами.

Тем самым элемент, который был взят снизу кучи, спустится на один уровень вниз. Будем дальше опускать этот элемент до тех пор, пока оба его потомка не станут меньше его (или у него не будет потомков, также необходимо аккуратно обработать случай одного потомка.

**def pop(Heap):**

**if len(Heap) == 2:**

**return Heap.pop()**

**retval = Heap[1]**

**Heap[1] = Heap.pop()**

**i = 1**

**while 2 \* i + 1 < len(Heap) \**

**and Heap[i] < max(Heap[2 \* i], Heap[2 \* i + 1]):**

**if Heap[2 \* i] > Heap[2 \* i + 1]:**

**Heap[i], Heap[2 \* i] = Heap[2 \* i], Heap[i]**

**i = 2 \* i**

**else:**

**Heap[i], Heap[2 \* i + 1] = Heap[2 \* i + 1], Heap[i]**

**i = 2 \* i + 1**

**if 2 \* i == len(Heap) - 1 and Heap[i] < Heap[2 \* i]:**

**Heap[i], Heap[2 \* i] = Heap[2 \* i], Heap[i]**

**return retval**

В этом примере сохраняется значение на вершине кучи в переменной retval, затем последний элемент удаляется из кучи и ставится на вершину кучи. Отдельно обрабатывается случай, когда куча состояла ровно из одного элемента, т. е. после удаления она становится пустой. Далее в основном цикле элемент опускается вниз. Цикл продолжается пока у элемента два потомка и хотя бы один из потомков больше текущего элемента. В этом случае элемент меняется местами с наибольшим из потомков и цикл повторяется заново.

После окончания цикла отдельно обрабатывается случай, когда у элемента ровно один потомок (нет правого потомка) и единственный левый потомок больше данного элемента, в этом случае необходимо провести еще один обмен.

**Сложность операций с кучей**

Все операции с кучей (добавление и удаление элемента) требуют  операций, где  - высота кучи. Пусть в куче  элементов, а ее высота равна . Тогда наибольшее число элементов, которое может быть в куче высоты  есть .

Таким образом, , откуда видно, что  примерно равно двоичному логарифму числа , то есть сложность всех операций с кучей - .

**Применения кучи**

Одно из наиболее известных применений кучи — сортировка при помощи кучи или пирамидальная сортировка (англ. heapsort).

В данной сортировке из элементов списка сначала строится куча, потом элементы по одному удаляются из кучи — сначала наибольший элемент, потом - наибольший из оставшихся и т. д. При этом кучу можно хранить там же, где хранятся элементы самого списка, тем самым пирамидальная сортировка имеет сложность , но при этом не требует дополнительной памяти (как сортировка слиянием) и не является вероятностной (как быстрая сортировка Хоара).

Также при помощи кучи можно организовать структуру данных «очередь с приоритетами». В очереди каждому элементу сопоставляется приоритет — некоторое целое число. При удалении элемента из очереди удаляется не тот элемент, который был добавлен раньше (как в обычной очереди), а элемент с наибольшим приоритетом. То есть элементы в очереди с приоритетами можно хранить в куче, сравнивая их при этом по приоритету.

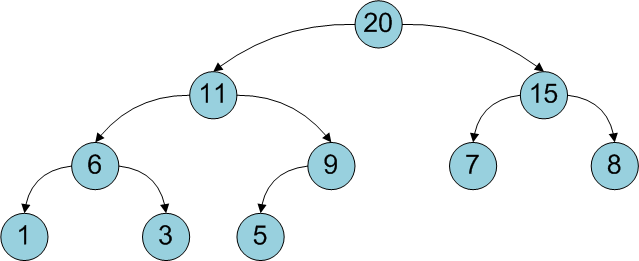
В очереди с приоритетами также есть операция изменения приоритета элемента. Для этого реализованы две функции — повышения и понижения приоритета. При повышении приоритета элемент поднимается вверх, поэтому эта функция реализована аналогично операции добавления элемента. При понижении приоритета элемент спускается вниз, как в операции удаления элемента.

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

Введение

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. В этой статье для простоты используется именно такое представление.

Напомню 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.  
  
Приведу заготовку класса на C#:

public class BinaryHeap

{

private List<int> list;

public int heapSize

{

get

{

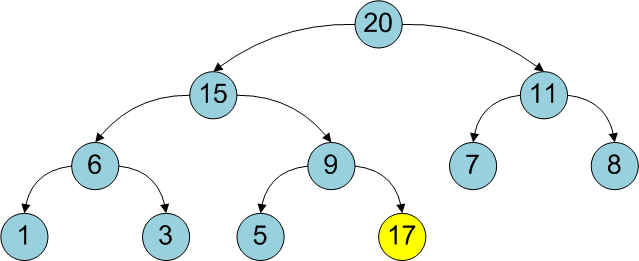
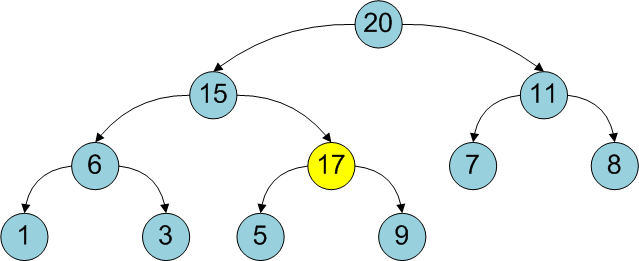
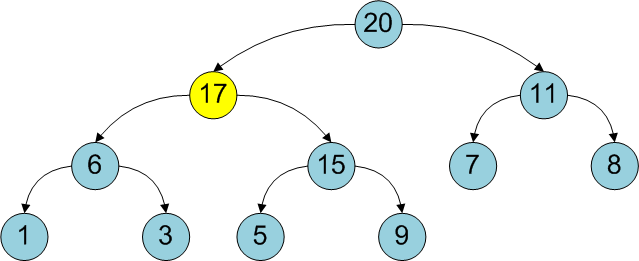
return this.list.Count();

}

}

}

Добавление элемента

Новый элемент добавляется на последнее место в массиве, то есть позицию с индексом **heapSize**:  
  
  
  
Возможно, что при этом будет нарушено основное свойство кучи, так как новый элемент может быть больше родителя. В таком случае следует «поднимать» новый элемент 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).

public void add(int value)

{

list.Add(value);

int i = heapSize - 1;

int parent = (i - 1) / 2;

while (i > 0 && list[parent] < list[i])

{

int temp = list[i];

list[i] = list[parent];

list[parent] = temp;

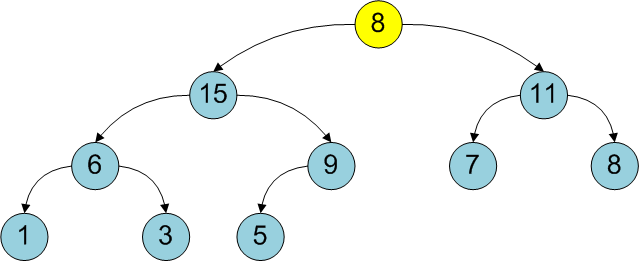
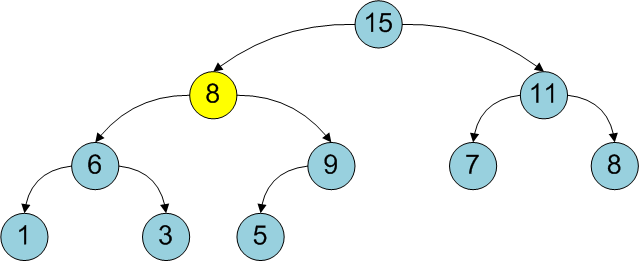
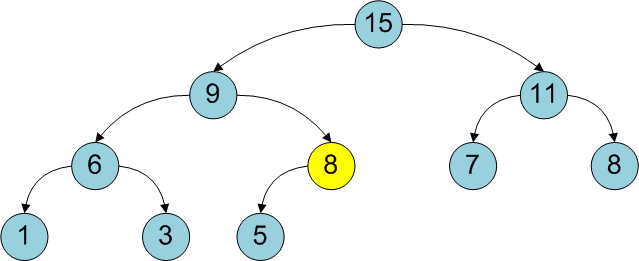
i = parent;

parent = (i - 1) / 2;

}

}

Упорядочение двоичной 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).  
  
  
  


public void heapify(int i)

{

int leftChild;

int rightChild;

int largestChild;

for (; ; )

{

leftChild = 2 \* i + 1;

rightChild = 2 \* i + 2;

largestChild = i;

if (leftChild < heapSize && list[leftChild] > list[largestChild])

{

largestChild = leftChild;

}

if (rightChild < heapSize && list[rightChild] > list[largestChild])

{

largestChild = rightChild;

}

if (largestChild == i)

{

break;

}

int temp = list[i];

list[i] = list[largestChild];

list[largestChild] = temp;

i = largestChild;

}

}

Построение двоичной кучи

Наиболее очевидный способ построить кучу из неупорядоченного массива – это по очереди добавить все его элементы. Временная оценка такого алгоритма O(N log2 N). Однако можно построить кучу еще быстрее — за О(N). Сначала следует построить дерево из всех элементов массива, не заботясь о соблюдении основного свойства кучи, а потом вызвать метод **heapify** 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.

public void buildHeap(int[] sourceArray)

{

list = sourceArray.ToList();

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

{

heapify(i);

}

}

Извлечение (удаление) максимального элемента

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 **heapify** для корня, то есть упорядочив все дерево.

public int getMax()

{

int result = list[0];

list[0] = list[heapSize - 1];

list.RemoveAt(heapSize - 1);

return result;

}

Сортировка с применением двоичной кучи

Заметим, что можно отсортировать массив, сначала построив из него двоичную кучу, а потом последовательно извлекая максимальные элементы. Оценим временную сложность такого элемента: построение кучи – O(N), извлечение **N** элементов – O(N log2 N). Следовательно, итоговая оценка O(N log2 N). При этом дополнительная память для массива не используется.

public void heapSort(int[] array)

{

buildHeap(array);

for (int i = array.Length - 1; i >= 0; i--)

{

array[i] = getMax();

heapify(0);

}

}

Заключение

Таким образом, двоичная куча имеет структуру дерева логарифмической высоты (относительно количества вершин), позволяет за логарифмическое же время добавлять элементы и извлекать элемент с максимальным приоритетом за константное время. В то же время двоичная куча проста в реализации и не требует дополнительной памяти.