Priority Queue

- Typically the following operations:
  - find element with highest priority
  - delete element with highest priority
  - insert element with assigned priority

- Enhance with
  - Delete a given element
  - Change key for a given element – usually decrease or increase key for a given application
- Key is to be able to find the element in the heap in constant time! Maintain an additional array for all the entities of interest e.g. vertices in a graph, that contains their position in the (heap) priority queue (handle)

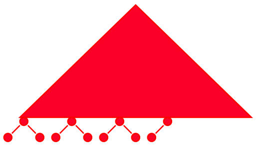
Binary Heap

Operations

- BinaryHeap() creates a new, empty, binary heap.
- insert(k) adds a new item to the heap.
- find_min() returns the item with the minimum key value, leaving item in the heap.
- del_min() returns the item with the minimum key value, removing the item from the heap.
- is_empty() returns true if the heap is empty, false otherwise.
- size() returns the number of items in the heap.
- Build_heap(list) builds a new heap from a list of keys.

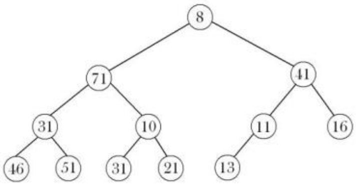
Heaps and Heapsort

- Definition A heap is a binary tree with keys at its nodes (one key per node) such that:
- It is essentially complete (note online text is slightly different), i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing – **shape property**.



- The key at each node is ≤ keys at its children (MinHeap)– **heap order (structure) property** (≥ MaxHeap)

Illustration of the heap's definition



- Min Heap

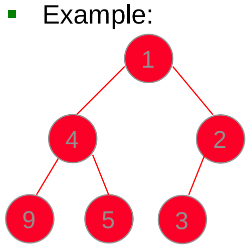
Note: Heap's elements are ordered top down (along any path down from its root), but they are not ordered left to right

Some Important Properties of a Heap (MaxHeap)

- Given n, there exists a unique binary tree with n nodes that is essentially complete, with $h = \lfloor \log_2 n \rfloor$
- The root contains the largest key
- The subtree rooted at any node of a heap is also a heap
- A heap can be represented as an array

Heap's Array Representation

- Store heap's elements in an array (whose elements indexed, for convenience, 1 to n) in top-down left-to-right order
- Example:



1	2	3	4	5	6
1	4	2	9	5	3

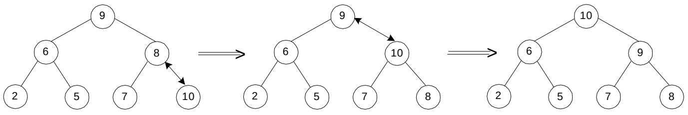
- Left child of node j is at 2j   Right child of node j is at 2j+1
- Parent of node j is at $\lfloor j/2 \rfloor$
- Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations

Insertion of a New Element into a Heap

- Insert the new element at last position in heap.
- Compare it with its parent and, if it violates heap condition, exchange them (Drift up)
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example: Insert key 10

Efficiency: O(log *n*)



Insertion into heap: perc_up (drift_up, sift_up)

1: Put new element into first open position, this maintains the structure property

```
def insert(self,k):
    self.heapList.append(k)
    self.currentSize = self.currentSize + 1
    self.percUp(self.currentSize)
```

2: Drift the element up until the heap property is restored

```
def percUp(self,i):
    while i // 2 > 0:
      if self.heapList[i] < self.heapList[i // 2]:
         tmp = self.heapList[i // 2]
         self.heapList[i // 2] = self.heapList[i]
         self.heapList[i] = tmp
      i = i // 2
```

Top-down heap construction

- Start with empty heap and repeatedly insert elements

Heap Construction (bottom-up)

High level pseudo-code

Initialize the array structure with keys in the order given
(structure property)

Loop: node = rightmost parental node to root
if it node doesn't satisfy the heap condition:
loop: exchange it with its smallest child until the heap condition holds ("Drift Down")

Bottom-up heap construction

1. Insert elements into array respecting the structure property
2. Rearrange elements to enforce the heap order property

```
def buildHeap(self,alist):
    i = len(alist) // 2
    self.currentSize = len(alist)
    self.heapList = [0] + alist[:]
    while (i > 0):
        self.percDown(i)
        i = i - 1
```


Bottom-up heap construction

1. Insert elements into array respecting the structure property
2. Rearrange elements to enforce the heap order property
   For Max Heap:

```
def percDown(self,i):
    while (i * 2) <= self.currentSize:
        mc = self.minChild(i)
        if self.heapList[i] < self.heapList[mc]:
            tmp = self.heapList[i]
            self.heapList[i] = self.heapList[mc]
            self.heapList[mc] = tmp
        i = mc
```

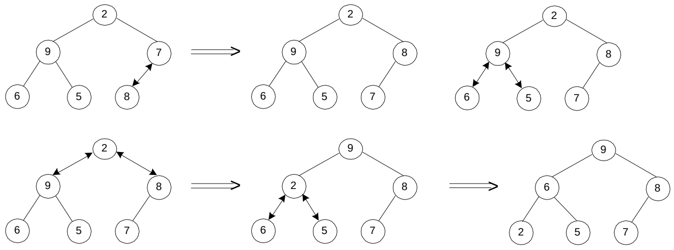
Bottom-up heap construction (Max Heap)

```
def perc_down (self, hole )
    tmp = array[hole]
    while (hole * 2 <= currentSize):
        child = hole * 2;
        if(child != currentSize)and
                     (array[child+1] > array[child] ))
            child = child+1
        if(array[child] > tmp ):
            array[hole] = array[child]
        else:
            break
        hole = child
     array[hole] = tmp
```

Refactoring of text to make less verbose

Example of Heap Construction

Construct a maxheap for the list 2, 9, 7, 6, 5, 8



Which is better Top-down vs. Bottom-up heap construction?

Remove max from heap

```
Swap last entry in heap with first entry in heap
               // store original first entry for return
Reduce heapsize by 1
Drift-down the new first element until the heap property is
        restored
               - see bottom-up construction for drift-down
```

Note: There are only two basic “moves” in the heap.

- *Drift-down* – used in bottom up construction and removal
- *Drift-up* – used in insertion and top-down construction

Priority Queue

- A priority queue is the ADT of a set of elements with numerical priorities and the following operations:
  - find element with highest priority
  - delete element with highest priority
  - insert element with assigned priority

- Heap is a very efficient way for implementing priority queues

- Applications determine what is a priority ordering!
  Sometimes want largest, sometimes smallest element to be found/deleted.

- Many implementation options : list, sorted list, …

Heapsort

Construct a heap for a given list of *n* keys

Repeat operation of root removal *n*-1 times:
- Exchange keys in the root and in the last (rightmost) leaf
- Decrease heap size by 1
- If necessary, swap new root with larger child until the heap condition holds (Drift Down)

Example of Sorting by Heapsort

Sort the list 2, 9, 7, 6, 5, 8 by heapsort

Stage 1 (heap construction)

```
2 9 7 6 5 8
2 9 8 6 5 7
2 9 8 6 5 7
9 2 8 6 5 7
9 6 8 2 5 7
```

Stage 2 (root/max removal)

```
9 6 8 2 5 7
7 6 8 2 5 | 9
8 6 7 2 5 | 9
5 6 7 2 | 8 9
7 6 5 2 | 8 9
2 6 5 | 7 8 9
6 2 5 | 7 8 9
5 2 | 6 7 8 9
5 2 | 6 7 8 9
2 | 5 6 7 8 9
```


Analysis of Heapsort

Stage 1: Build heap for a given list of n keys worst-case

C(n) = $\sum_{i=0}^{h-1} 2(h-i)2^i = 2\big(n - log_2(n+1)\big) \in \Theta(n)$

Stage 2: Repeat operation of root removal n-1 times (fix heap) worst-case

C(n) = $\sum_{i=0}^{n-1} 2\big(log_2(i)\big) \in \Theta(n\ log\ n)$

- Both worst-case and average-case efficiency: Θ(n*logn)

Performance Analysis of Bottom-up Build Heap

- How do binary heaps grow ???
  - A binary heap of height k contains between 2^k and 2^{k+1} -1 keys
  - Roughly half the keys are leaves
  - A quarter would move at most one level
  - …
- Intuitively this means O(n)