Heaps, Heapsort, Priority Queues

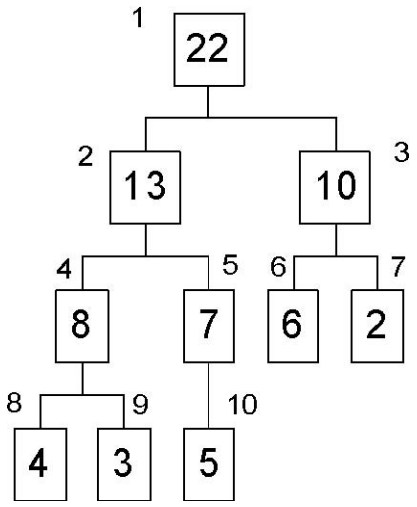
Heaps: A data structure and associated algorithms, NOT GARBAGE COLLECTION

A heap data structure is an array of objects than can be viewed as a complete binary tree such that:

1. Each tree node corresponds to elements of the array
2. The tree is complete except possibly the lowest level, filled from left to right

The heap property is defined as an ordering relation *R* between each node and its descendants. For example, *R* could be smaller than or bigger than. In the examples that follow, we will use the bigger than relation.

Example: Given array [22 13 10 8 7 6 2 4 3 5]



Note that the elements are not sorted, only max element at root of tree.

The **height** of a node in the tree is the number of edges on the longest simple downward path from the node to a leaf; e.g. height of node 6 is 0, height of node 4 is 1, height of node 1 is 3.

The height of the tree is the height from the root. As in any complete binary tree of size n, this is lg n.

There are 2^h nodes at level h and 2^{h+1} - 1 total nodes in a complete binary tree.

We can represent a heap as an array A that has two attributes:

1 Length(A) – Size of the array
2 HeapSize(A) - Size of the heap

The property Length(A) ≥HeapSize(A) must be maintained.
The heap property is stated as A[parent(i)] ≥A[i]

The root of the tree is A[1].
Formula to compute parents, children in an array:

Parent(i) = A[$\lfloor i/2 \rfloor$]

Left Child(I) = A[2i]

Right Child(I) = A[2i+1]

Where might we want to use heaps? Consider the Priority Queue problem: Given a sequence of objects with varying degrees of priority, and we want to deal with the highest-priority item first.

Managing air traffic control - want to do most important tasks first.
Jobs placed in queue with priority, controllers take off queue from top

Scheduling jobs on a processor - critical applications need high priority

Event-driven simulator with time of occurrence as key. Use min-heap, which keeps smallest element on top, get next occurring event.

To support these operations we need to extract the maximum element from the heap:

```
HEAP-EXTRACT-MAX(A)
        remove A[1]
        A[1] =A[n]      ; n is HeapSize(A), the length of the heap, not array
        n = n-1         ; decrease size of heap
        Heapify(A,1,n) ; Remake heap to conform to heap properties
```

Runtime: Q(1) +Heapify time

Note: Successive removals will result in items in reverse sorted order!

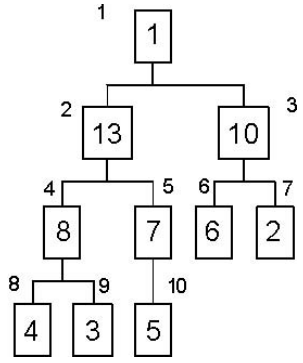
We will look at:

Heapify : Maintain the heap property
Build Heap : How to initially build a heap
Heapsort : Sorting using a heap

Heapify: Maintain heap property by "floating" a value down the heap that starts at I until it is in the right position.

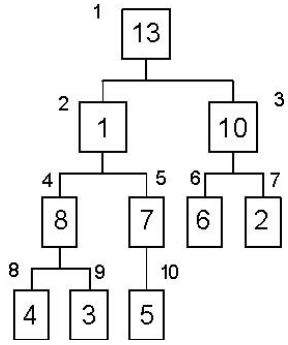
```
Heapify(A,i,n) ; Array A, heapify node i, heapsize is n
        ; Note that the left and right subtrees of i are also heaps
        ; Make i's subtree be a heap.
        If (2i ≤ n) and A[2i]>A[i]
                ; see which is largest of current node and its
                children largest = 2i
        else
                largest = i
        If (2i+1 ≤ n) and A[2i+1]>A[largest]
                largest =  2i+1
        If largest  ≠ i
                swap (A[i], A[largest])
        Heapify(A,largest,n)
```

Example: Heapify(A,1,10). A=[1 13 10 8 7 6 2 4 3 5] Find largest of children and swap. All subtrees are valid heaps so we know the children are the maximums.

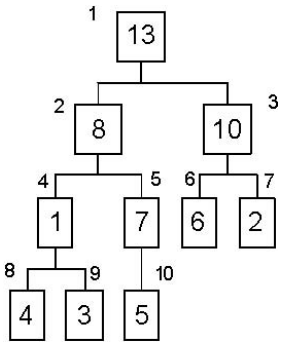


Find largest of children and swap. All subtrees are valid heaps so we know the children are the maximums.

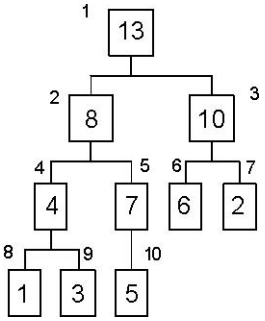
Next is Heapify(A,2,10). A=[13 1 10 8 7 6 2 4 3 5]



Next is Heapify(A,4,10). A=[13 8 10 1 7 6 2 4 3 5]



Next is Heapify(A,8,10). A=[13 8 10 4 7 6 2 1 3 5]



On this iteration we have reached a leaf and are finished. (Consider if started at node 3, n=7)

Runtime: Intuitively this is Q(lg n)since we make one trip down the path of the tree and we have an almost complete binary tree.

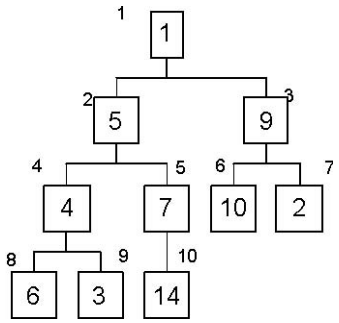
Building The Heap:

Given an array A, we want to build this array into a heap.
Note: Leaves are already a heap! Start from the leaves and build up from there.

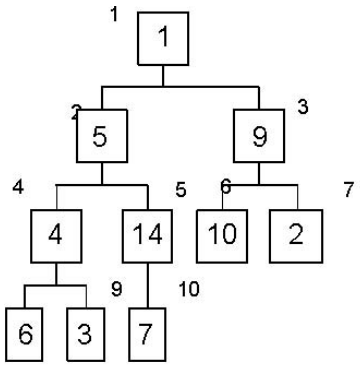
```
Build-Heap(A,n)
    for i =  n downto 1 ; could start at n/2
     Heapify(A,i,n)
```

Start with the leaves (last ½ of A) and consider each leaf as a 1 element heap. Call Heapify on the parents of the leaves, and continue recursively to call Heapify, moving up the tree to the root.

Example: Build-Heap(A,10). A=[1 5 9 4 7 10 2 6 3 14]

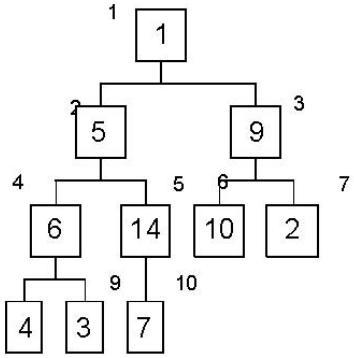


Heapify(A,10,10) exits since this is a leaf.
Heapify(A,9,10) exits since this is a leaf.
Heapify(A,8,10) exits since this is a leaf.
Heapify(A,7,10) exits since this is a leaf.
Heapify(A,6,10) exits since this is a leaf.
Heapify(A,5,10) puts the largest of A[5] and its children, A[10] into A[5]:



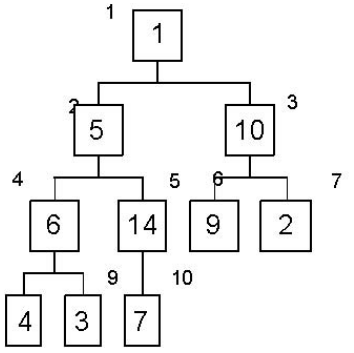
A=[1 5 9 4 14 10 2 6 3 7]

Heapify(A,4,10):



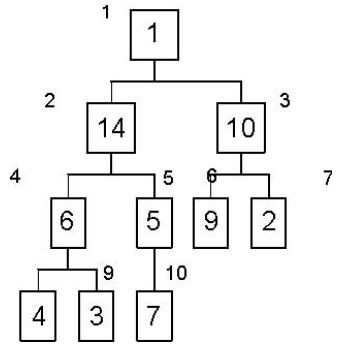
A=[1 5 9 6 14 10 2 4 3 7]

Heapify(A,3,10):

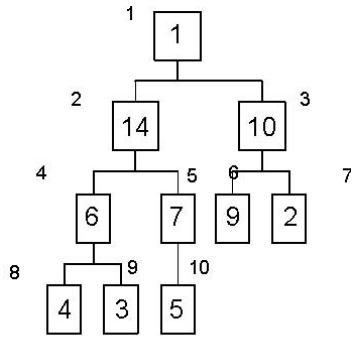


A=[1 5 10 6 14 9 2 4 3 7]

Heapify(A,2,10): First iteration:

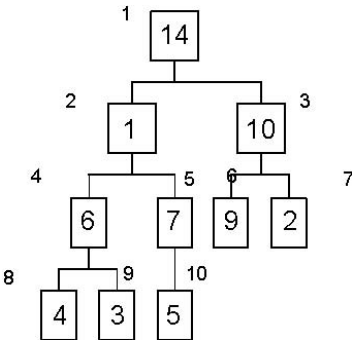


this calls Heapify(A,5,10):

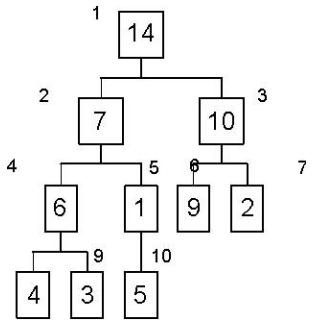


A=[1 14 10 6 7 9 2 4 3 5]

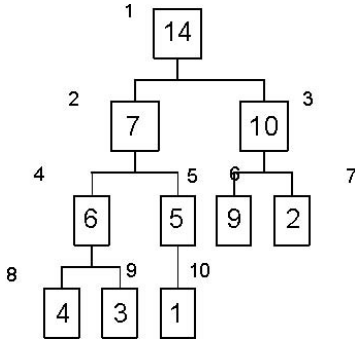
Heapify(A,1,10):



Calls Heapify(A,2,10):



Calls Heapify(A,5,10):



Finished heap: A=[14 7 10 6 5 9 2 4 3 1]

Running Time: We have a loop of n times, and each time call heapify which runs in Θ (lgn). This implies a bound of O(nlgn).

HeapSort: Once we can build a heap and heapify a heap, sorting is easy. Idea is to:

```
HeapSort(A,n)
    Build-Heap(A,n)
    for i = n downto 2
        Swap(A[1], A[i])
        Heapify(A,1,i-1)
```

Runtime is O(nlgn) since we do Heapify on n-1 elements, and we do Heapify on the whole tree. .

Variation on heaps: Heap could have min on top instead of max,
Heap could be k-ary tree instead of binary

Priority Queues: A priority queue is a data structure for maintaining a set of S elements each with an associated key value.

Operations:

Insert(S,x) puts element x into set S
Max(S,x) returns the largest element in set S
Extract-Max(S) removes the largest element in set S

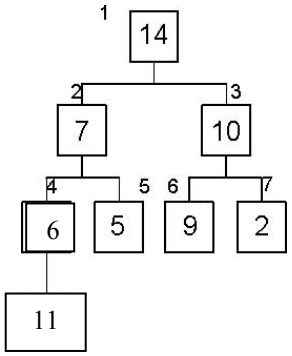
Uses: Job scheduling, event driven simulation, etc.

We can model priority queues nicely with a heap. This is nice because heaps allow us to do fast queue maintenance.

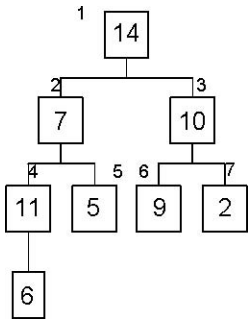
Max(S,x) : Just return root element. Takes O(1) time.

```
Heap-Insert(A,key)
        n=n+1
        i= n
        while (i > 1) and A[⌊i /2⌋] < key
                A[i] =A[⌊i / 2⌋]
                i = ⌊i/2⌋
        A[i] = key
```

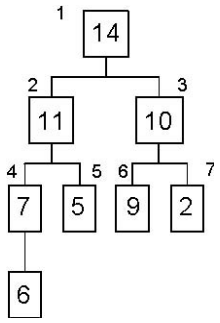
Idea: same as heapify. Start from a new node, and propagate its value up to right level.
Example: Insert new element "11" starting at new node on bottom, i=8



Bubble up:



i=4, bubble up again



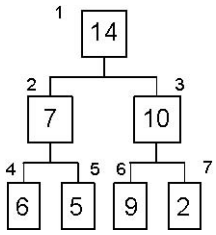
At this point, the parent of 2 is larger so the algorithm stops.

Runtime = O(lgn) since we only move once up the tree levels.

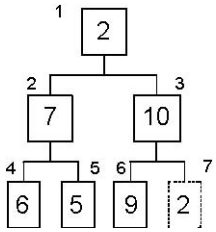
```
Heap-Extract-Max(A,n)
        max = A[1]
        A[1]= A[n]
        n= n-1
        Heapify(A,1,n)
        return max
```

Idea: Make the nth element the root, then call Heapify to fix.
Uses a constant amount of time plus the time to call Heapify, which is O(lgn).
Total time is then O(lgn).

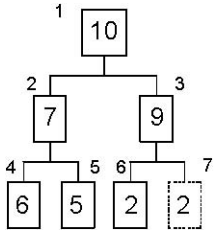
Example: Extract(A,7):



Remove 14, so max=14. Stick 7 into 1:



Heapify (A,1,6):



We have a new heap that is valid, with the max of 14 being returned. The 2 is sitting in the array twice, but since n is updated to equal 6, it will be overwritten if a new element is added, and otherwise ignored.