CS411 Lecture Notes
1/30/01

Heaps, Heapsort, Priority Queues

Sorting problem so far:

Insertion Sort: In Place, $O(n^2)$ worst case
Merge Sort : Not in place, $O(n\lg n)$ worst case
Quicksort : In place, $O(n^2)$ worst case, $O(n\lg n)$ expected case
Heapsort : In place, $O(n\lg n)$ worst case

Heap:

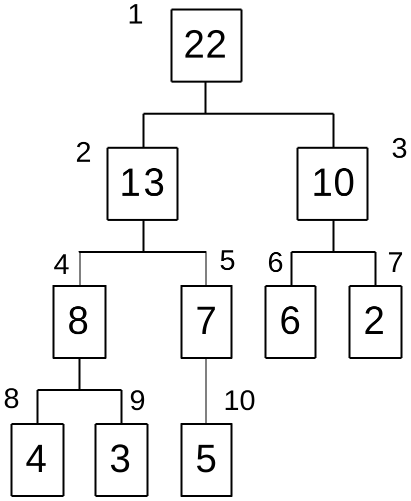
A data structure and associated algorithms, NOT GARBAGE COLLECTION

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

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

The heap property for all nodes I in the tree must be maintained except for the root:
Parent node(I) ≥ I

Example: Given array [22 13 10 87 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.

Caveats: 2^h nodes at level h. $2^{h+1}-1$ total nodes in a complete binary tree.

A heap represented as an array A represented has two attributes:

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

The property Length(A) $\geq$ HeapSize(A) must be maintained. (why ?)

The heap property is stated as A[parent(I)] $\geq$ 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]

(Show how to represent the above tree as an array in the example)

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] $\leftarrow$ A[n]        ; n is HeapSize(A), the length of the heap, not array
    n $\leftarrow$ n-1            ; decrease size of heap
    Heapify(A,1,n)  ; Remake heap to conform to heap properties
```

Runtime: $\Theta(1)$ +Heapify time

Differences from book :

no error handling
n instead of HeapSize(A)
slightly higher abstraction
Passing “n” to Heapify routine

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

We will look at:

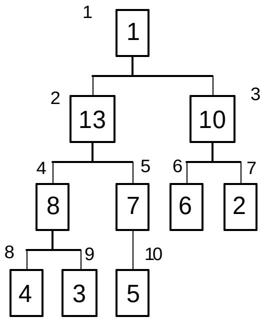
Heapify : Maintain the heap property
Build Heap : How to initially build a heap
Heapsort : Sorting using a heap

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

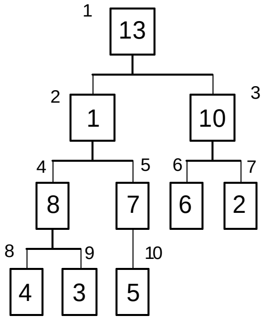
```
Heapify(A,I,n)            ; Array A, heapify node I, heapsize is n
    ; Note that the left and right subtrees of I are also heaps
    ; Make I’s subtree be a heap.
    If $2I \le n$ and $A[2I]>A[I]$
        ; see which is largest of current node and its children
        then largest $\leftarrow$ 2I
        else largest $\leftarrow$ I
    If $2I+1 \le n$ and $A[2I+1]>A[largest]$
        then largest $\leftarrow$ 2I+1
    If largest $\neq$ I
        then swap $A[I] \leftrightarrow A[largest]$
             Heapify(A,largest,n)
```

Differences from book : 2I and 2I+1 instead of left and right, n instead of heapsize

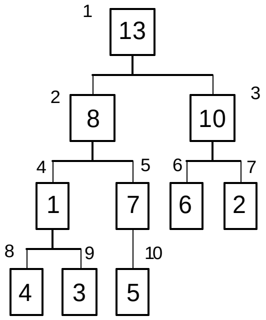
Example: Heapify(A,1,10). A=[1 13 10 8 7 6 2 4 3 5]



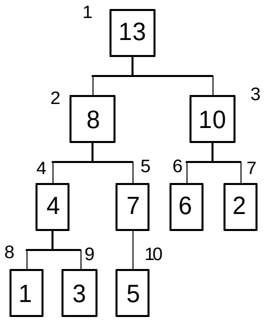
Find largest of children and swap. All subtrees are valid heaps so we know the children are the maximums.



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 $\Theta(\lg n)$ since we make one trip down the path of the tree and we have an almost complete binary tree. This would correspond to : $T(n) = T(\frac{n}{2}) + \Theta(1)$

Worst-case Runtime actually equal to: $T(n) = T(\frac{2n}{3}) + \Theta(1)$. Split problem into at least 2/3 the size. Consider the number of nodes on the left side vs. the right in the most unbalanced state:

In the worst case a heap of height n has all of the bottom leaves of the left child filled and the right child has height n-1. This is the most unbalanced a tree will ever become due to the heap property.

For any complete binary tree of n nodes and l leaves, where the lowest level is full, l=n+1. That is, half of the tree is leaves.

For a tree of height h, the number of leaves in a complete binary tree is 2^h and the number of nodes (not counting leaves) is 2^h -1.

So in the worst case, the left subtree has about $\frac{1}{2}2^h + \frac{1}{2}2^h$ leaves + nodes.

The right subtree has $\frac{1}{2}2^h$ nodes.

If we take the ratio of the left subtree over the total number of nodes: $\dfrac{\frac{1}{2}2^h + \frac{1}{2}2^h}{\frac{1}{2}2^h + \frac{1}{2}2^h + \frac{1}{2}2^h} = \dfrac{2}{3}$

So we are able to split the problem by at least 1/3 each iteration of the loop in the worst case.

Given : $T(n) = T(\frac{2n}{3}) + \Theta(1)$ Can solve by the master theorem.

Case 2:

a=1,b=3/2

Is $\Theta(1) = \Theta(n^{\log_{1.5} 1})$?

$\Theta(1) = \Theta(n^0)$?

YES, so T(n)=$\Theta(f(n)\lg n) = \Theta(\lg n)$

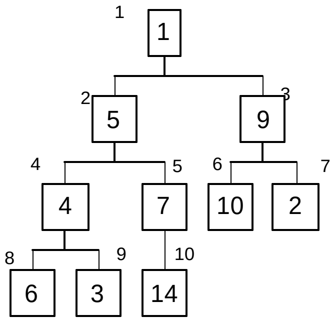
Building The Heap:

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.

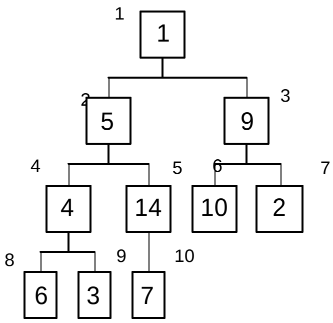
```
Build-Heap(A,n)
    for I ← n downto 1                    ; could start at n/2
        do Heapify(A,I,n)
```

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]

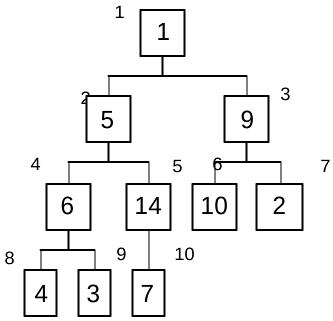


Heapify(A,10,10) exits since this is a leaf.
Heapify(A,9,10) exits since this is a leaf.
Heapify(A,8,10) exits since this is a leaf.
Heapify(A,7,10) exits since this is a leaf.
Heapify(A,6,10) exits since this is a leaf.
Heapify(A,5,10) puts the largest of A[5] and its children, A[10] into A[5]:



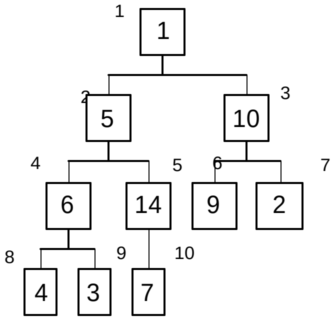
A=[1 5 9 4 14 10 2 6 3 7]

Heapify(A,4,10):



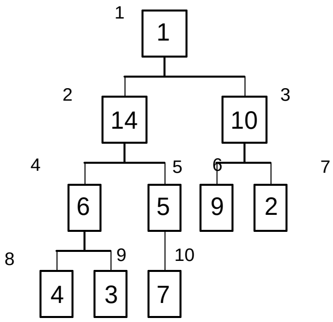
A=[1 5 9 6 14 10 2 4 3 7]

Heapify(A,3,10):

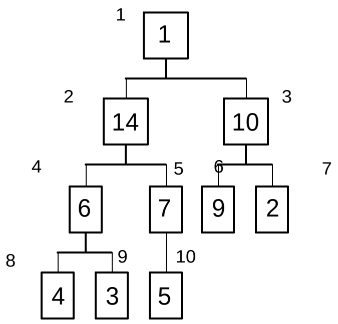


A=[1 5 10 6 14 9 2 4 3 7]

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

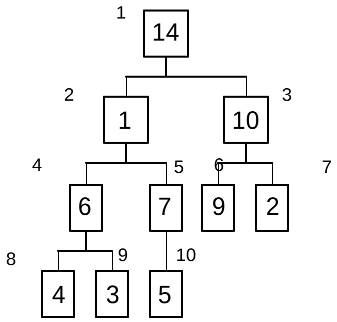


this calls Heapify(A,5,10):

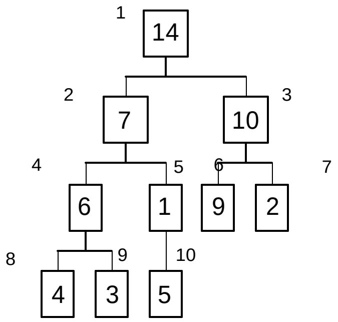


A=[1 14 10 6 7 9 2 4 3 5]

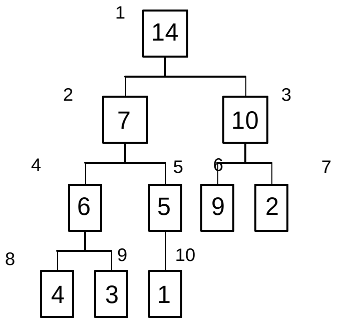
Heapify(A,1,10):



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). This is correct, but is a loose bound! We can do better. Note: This is a good approach in general. Start with whatever bound you can determine, then try to tighten it.

Key observation: Each time heapify is run within the loop, it is not run on the entire tree. We run it on subtrees, which have a lower height, so these subtrees do not take lgn time to run. Since the tree has more nodes on the leaf, most of the time the heaps are small compared to the size of n.

Better Bound for Build-Heap:

Property: In an n-element heap there are at most $\frac{n}{2^h}$ nodes of height h (The leaves are height 1 and root at lgn, this is backwards from normal). The time required by Heapify when called in Build-Heap on a node at height h is O(h); h=lgn for the entire tree.

Cost of Build-Heap is:

$$T(n) = \sum_{h=1}^{heap\_height} (\#nodes\_at\_h)(Heapify - Time)$$

$$T(n) = \sum_{h=1}^{\lg n} \frac{n}{2^h} O(h)$$

$$T(n) = O\left(\sum_{h=1}^{\lg n} \frac{n}{2^h} h \right)$$

We know that $\sum_{n=0}^{\infty} nx^n = \frac{x}{(1-x)^2}$. If x=1/2 then (1/2)^n=1/2^n so:

$$\sum_{n=0}^{\infty} h\left(\frac{1}{2}\right)^h = \frac{1/2}{(1-1/2)^2} = 2$$

Substitute this back in, which is safe because the sum from 0 to infinity is LARGER than the sum from 1 to lgn. This means we are working with a somewhat looser upper bound on the right hand side::

$$T(n) \le O\left(n \sum_{h=0}^{\infty} h \frac{1}{2^h} \right)$$

$$T(n) \le O(n2)$$

$$T(n) = O(n)$$

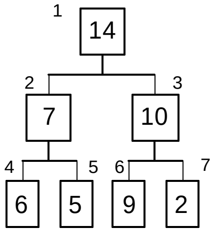
DONE!

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

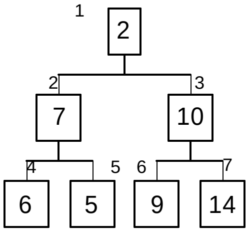
```
HeapSort(A,n)
    Build-Heap(A,n)
    for I $\leftarrow$ n downto 2
        do      Swap(A[1]$\leftrightarrow$A[I]
                Heapify(A,1,I-1)
```

Slightly different from book. The book removes root, puts into sorted list. In this example we are sticking it at the end of the array and the loop decreases the size, so the element is not touched again.

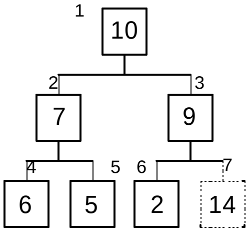
Example: HeapSort(A,7) A=[14 7 10 6 5 9 2] (already a heap)



Swap root with 7:

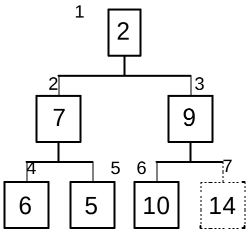


Heapify(A,1,6)

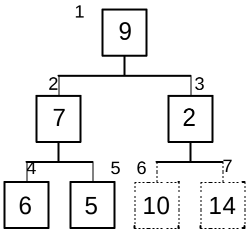


A=[10 7 9 6 5 2 14]

Swap root with 6:

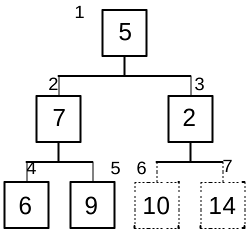


Heapify(A,1,5)

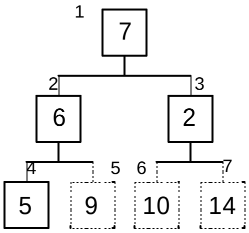


A=[9 7 2 6 5 10 14]

Swap root with 5:

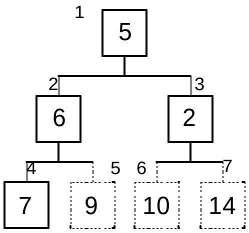


Heapify(A,1,4)

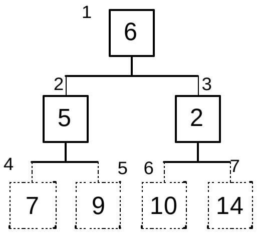


A=[7 6 2 5 9 10 14]

Swap root with 4:

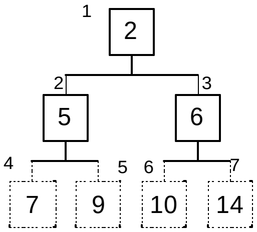


Heapify(A,1,3)

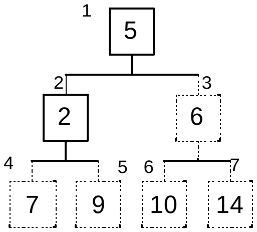


A=[6 5 2 7 9 10 14]

Swap root with 3:

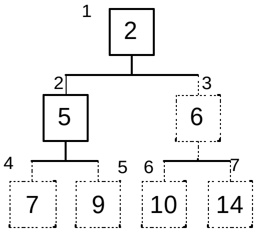


Heapify(A,1,2)



A=[5 2 6 7 9 10 14]

Swap root with 2:



A=[2 5 6 7 9 10 14]

We are done!

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

Note: In-place sort, required no extra storage variables unlike Merge Sort, which used extra space in the recursion.

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.

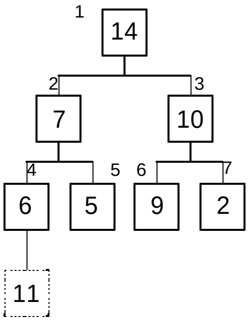
We can model priority queues nicely with a heap. This is nice because heaps allow us to do fast queue maintenance.

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

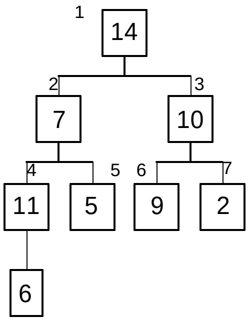
```
Heap-Insert(A,key)
    n ← n+1
    I ← n
    while I > 1 and A[$\lfloor i/2 \rfloor$] < key
        do    A[I] ← A[$\lfloor i/2 \rfloor$]
              I ← $\lfloor i/2 \rfloor$
    A[I] ← key
```

Idea: same as heapify. Start from a new node, and propagate its value up to right level.

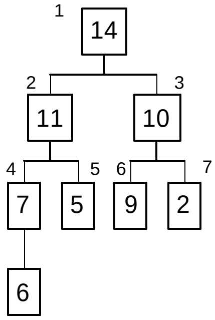
Example: Insert new element "11" starting at new node on bottom, I=8



Bubble up:



I=4, bubble up again



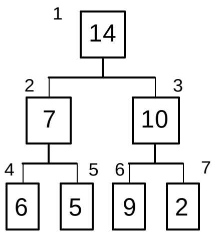
At this point, the parent of 2 is larger so the algorithm stops.

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

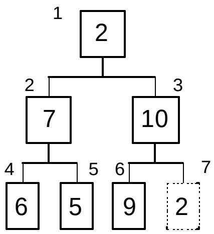
```
Heap-Extract-Max(A,n)
    max← A[1]
    A[1]← A[n]
    n← n-1
    Heapify(A,1,n)
    return max
```

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).

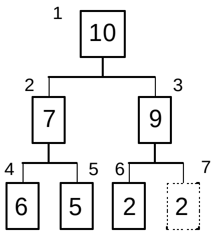
Example: Extract(A,7):



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.

In general: Heaps have log or constant operation on queues as opposed to linear.

Length	O(lgn)	O(n)	
8	3	8	
16	4	16	
32	5	32	
…			
256	8	256	
1024	10	1024	
4096	12	4096	Consider if linked lists, cost to traverse!

Recap of sorts so far:

Insertion, Merge, Heap, Quicksort

Quicksort is actually used most frequently despite $O(n^2)$ worst case runtime