Heaps, Heapsort, Priority Queues

So Far

Insertion Sort: $O(n^2)$ worst case
Linked List: $O(n)$ search, some operations $O(n^2)$

Heap: Data structure and associated algorithms,
Not garbage collection context

Binary Tree

- A binary tree is a structure that can be visualized as an upside-down tree where the root is at the top and the leaves are at the bottom
  - Each node in the tree has at most two children
  - If each level is completely filled-in then it is called a **complete** binary tree
- We can create binary trees in several ways!

Sample Binary Tree

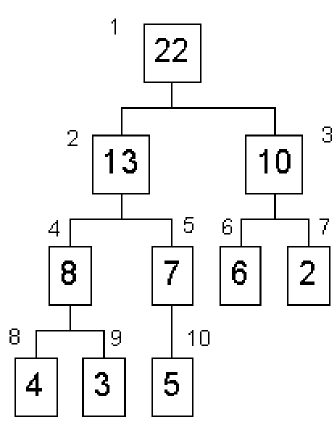
Parent/Child Relationship, Terminology

Heap Structure

- 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 (could flip this if desired)

Heap 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
Arrays are 1-based not 0-based

Height of the Heap

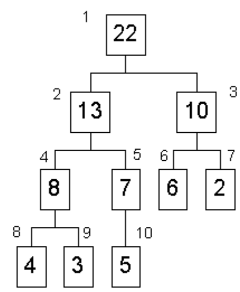
- The **height** of a node in the tree is the number of edges on the longest simple downward path from the node to a leaf; e.g. height of node 6 is 0, height of node 4 is 1, height of node 1 is 3.
- The height of the tree is the height from the root. As in any complete binary tree of size n, this is lg n.
- Caveats: 2^h nodes at level h. 2^{h+1}-1 total nodes in a complete binary tree.

Heap Attributes

- A heap represented as an array A has two attributes:
  - Length(A) – Size of the array
  - HeapSize(A) - Size of the heap
- The property
  Length(A) ≥ HeapSize(A) must be maintained. (why ?)
- The heap property is stated as
  A[parent(I)] ≥ A[I]

Computing Parents, Children

- The root of the tree is A[1].
- Formula to compute parents, children in an array:
  - Parent(I) = A[floor(I/2)]
  - Left Child(I) = A[2I]
  - Right Child(I) = A[2I+1]



Priority Queues

- 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.

Extracting Max

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

Note: Arrays in this example are 1-based, not 0-based
Successive removals will result in items in reverse sorted order!

Heapify Routine

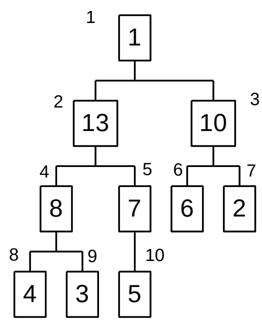
- Heapify maintains heap property by "floating" a value down the heap that starts at I until it is in the right position.

```
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
                then largest ← 2I
                else largest ← I
        If 2I+1 ≤ n and A[2I+1]>A[largest]
                then largest ← 2I+1
        If largest ≠ I
                then swap A[I] ↔ A[largest]
                     Heapify(A,largest,n)
```

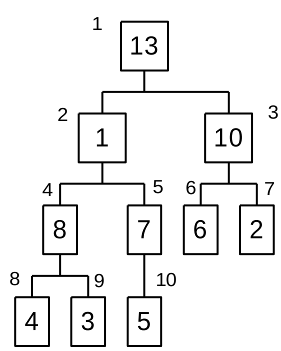
Heapify Example

- Heapify(A,1,10).

  A=[1 13 10 8 7 6 2 4 3 5]



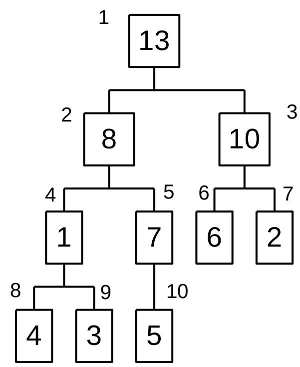
Heapify Example



- Next is Heapify(A,2,10).

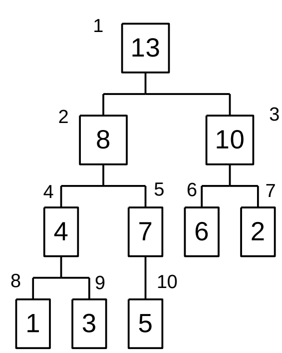
  A=[13 1 10 8 7 6 2 4 3 5]

Heapify Example



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

Heapify Example



- 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.

Heapify Runtime

- Later you will see recurrence relations, which recursively specify how long the algorithm takes to run:

$$T(n) \le T(\frac{2n}{3}) + \Theta(1)$$

- We can always split the problem into at least 2/3 the size.

- Solving this yields Θ(lgn) runtime in the worst case, O(1) in the best case, for O(lgn) overall.

- Basically we start at the top and move toward the bottom; since the tree is balanced we only make lg(n) swaps from the root to a leaf.

Building the Heap

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

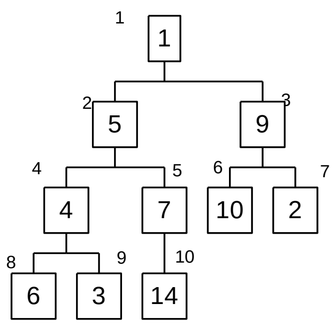
```
Build-Heap(A,n)
    for I = n downto 1                ; could we start at n/2?
      do Heapify(A,I,n)
```

- 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.

Build-Heap Example

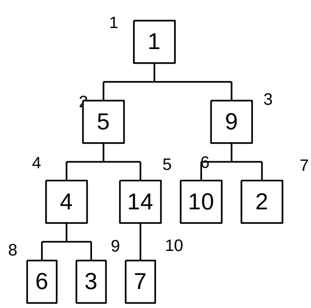
- Build-Heap(A,10)

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



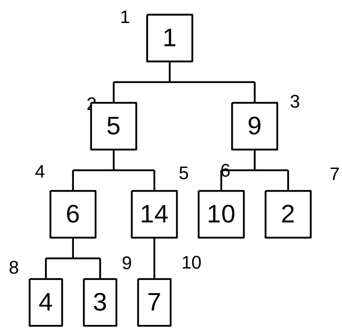
Heapify(A,10,10) exits since this is a leaf.
Heapify(A,9,10) exits since this is a leaf.
Heapify(A,8,10) exits since this is a leaf.
Heapify(A,7,10) exits since this is a leaf.
Heapify(A,6,10) exits since this is a leaf.
Heapify(A,5,10) puts the largest of A[5] and its children, A[10] into A[5]:

Build-Heap Example



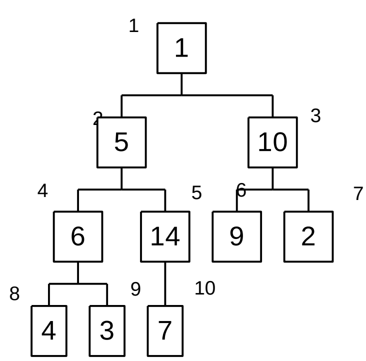
A=[1 5 9 4 14 10 2 6 3 7]
Heapify(A,4,10)

Build-Heap Example



- A=[1 5 9 6 14 10 2 4 3 7]
- Heapify(A,3,10):

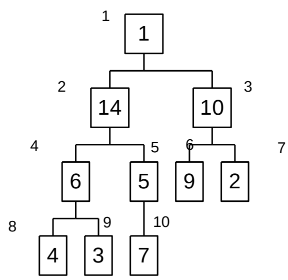
Build-Heap Example



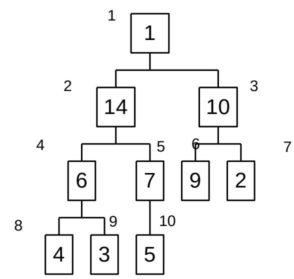
- A=[1 5 10 6 14 9 2 4 3 7]
- Heapify(A,2,10):

Build-Heap Example

Heapify(A,2,10)



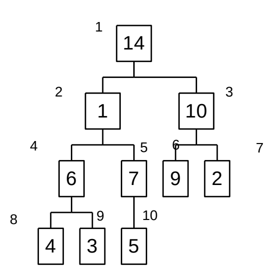
Heapify(A,5,10)



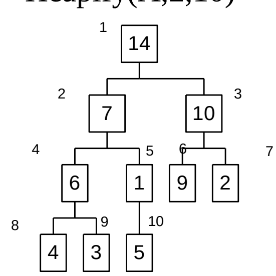
- A=[1 14 10 6 7 9 2 4 3 5]
- Heapify(A,1,10):

Build-Heap Example

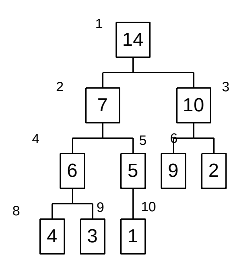
Heapify(A,1,10)



Heapify(A,2,10)



Heapify(A,5,10)



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

Build-Heap Runtime

- Running Time
  - We have a loop of n times, and each time call heapify which runs in (lgn). This implies a bound of O(nlgn). This is correct, but is a loose bound! We can do better.
  - 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.

Build-Heap Runtime

- Property: In an n-element heap there are at most $n/(2^h)$ nodes at height h
  - The leaves are h=1 and root at lgn, this is a slight change from the previous definition (leaves at height 0)
- 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.

Build-Heap Runtime

- Cost of Build-Heap is then:

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

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

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

Build-Heap Runtime

- 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 in our equation, which is safe because the sum from 0 to infinity is LARGER than the sum from 1 to lgn.

$$T(n) = O\left(\sum_{h=1}^{\lg n} \frac{n}{2^h} h \right) < O\left(n \sum_{h=1}^{\infty} \frac{h}{2^h} \right) < O(n2) = O(n)$$

Heapsort

- Once we can build a heap and heapify, sorting is easy… just remove max N times

```
HeapSort(A,n)
        Build-Heap(A,n)
        for I ← n downto 2
                do      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.

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

Heap Variations

- Heap could have min on top instead of max
- Heap could be k-ary tree instead of binary
- Priority Queue
  - Desired 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

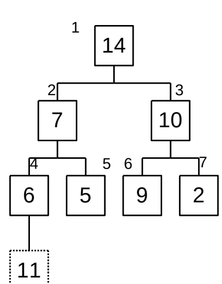
Priority Queue implemented via Heap

- Max(S,x)
  - Just return root element. Takes O(1) time.
- Insert(S,x)
  - Similar idea to heapify, put new element at end, bubble up to proper place toward root

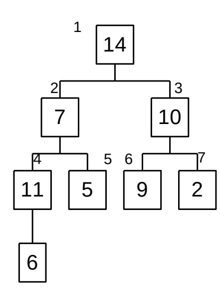
```
Heap-Insert(A,key)
    n←n+1
    I ← n
    while I > 1 and A[⌊i / 2⌋] < key
        do    A[I] ←A[⌊i / 2⌋]
              I ← ⌊i / 2⌋
    A[I] ←key
```

Insert Example

- Insert new element “11” starting at new node on bottom, I=8

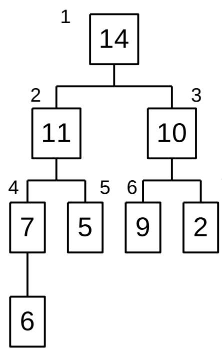


Bubble up



Insert Example

- Bubble up once more



Stop at this point, since parent (index 1, value 14) has a larger value

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

Extract Max

- To extract the max, copy the last element to the root and heapify

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

O(lgn) time

Can implement priority queue operations in O(lgn) time, O(n) to build