Binary Heaps

CSE 326
Data Structures
Unit 8

Reading: Sections 6.1-6.4

Revisiting FindMin

- Application: Find the smallest (or highest priority) item quickly
  - › Operating system needs to schedule jobs according to priority instead of FIFO
  - › Event simulation (bank customers arriving and departing, ordered according to when the event happened)
  - › Find student with highest grade, employee with highest salary etc.

Priority Queue ADT

- Priority Queue can efficiently do:
  - › FindMin (and DeleteMin)
  - › Insert
- What if we use…
  - › Lists: If sorted, what is the run time for Insert and FindMin? Unsorted?
  - › Binary Search Trees: What is the run time for Insert and FindMin?

Less flexibility → More speed

- Lists
  - › If sorted: FindMin is O(1) but Insert is O(N)
  - › If not sorted: Insert is O(1) but FindMin is O(N)
- Balanced Binary Search Trees (BSTs)
  - › Insert is O(log N) and FindMin is O(log N)
- BSTs look good but…
  - › BSTs are efficient for all Finds, not just FindMin
  - › We only need FindMin

Better than a speeding BST

- We can do better than Balanced Binary Search Trees.
  - › Very limited requirements: Insert, FindMin, DeleteMin. The goals are:
  - › FindMin is O(1)
  - › Insert is O(log N)
  - › DeleteMin is O(log N)

Binary Heaps (minimum)

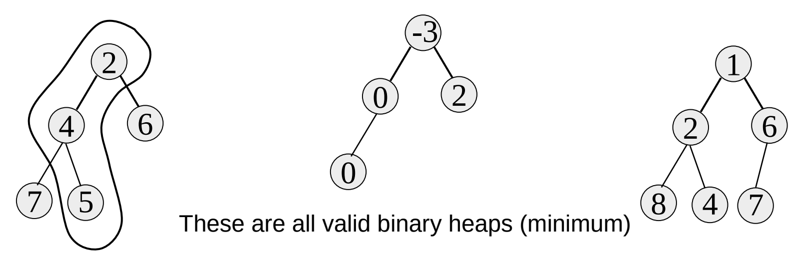
- A binary heap is a binary tree (NOT a BST) that is:
  - › Complete: the tree is completely filled except possibly the bottom level, which is filled from left to right
  - › Satisfies the heap order property:
    - every node is less than or equal to its children
    - In particular, the root node is always the smallest node

Binary Heaps (maximum)

- A binary heap is a binary tree (NOT a BST) that is:
  - › Complete: the tree is completely filled except possibly the bottom level, which is filled from left to right
  - › Satisfies the heap order property
    - every node is greater than or equal to its children
    - In particular, the root node is always the largest node

Heap order property

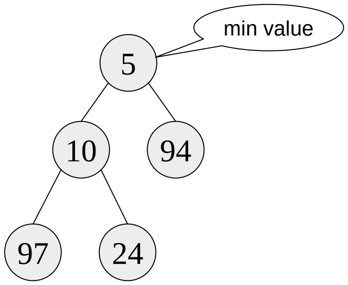
- A heap provides limited ordering information
- Each *path* is sorted, but the subtrees are not sorted relative to each other
  - › A binary heap is NOT a binary search tree



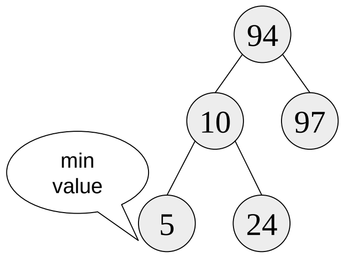
These are all valid binary heaps (minimum)

Binary Heap vs Binary Search Tree

Binary Heap



Binary Search Tree

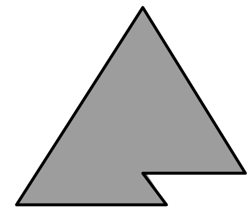


Parent is less than both left and right children

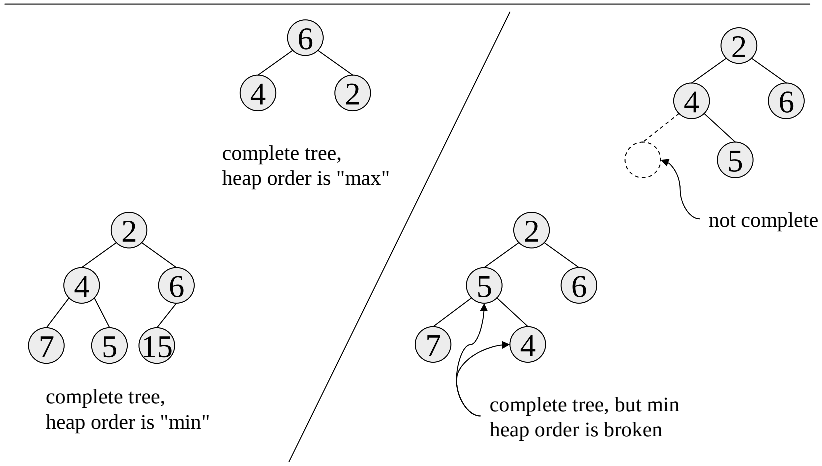
Parent is greater than left child, less than right child

Structure property

- A binary heap is a complete tree
  - › All nodes are in use except for possibly the right end of the bottom row

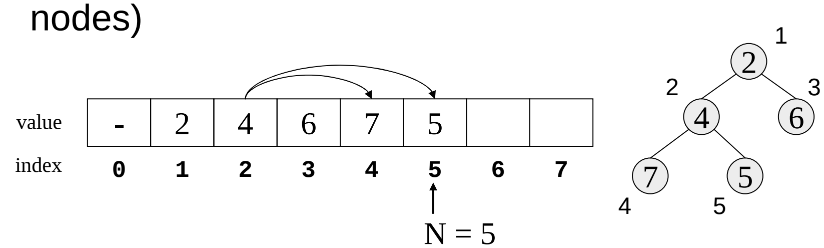


Examples



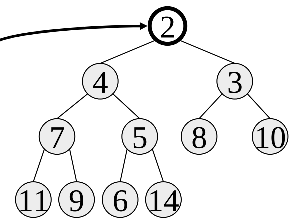
Array Implementation of Heaps

- Root node = A[1]
- Children of A[i] are in A[2i], A[2i + 1]
  - › Proof: By induction on i
- Keep track of current size N (number of nodes)



FindMin and DeleteMin

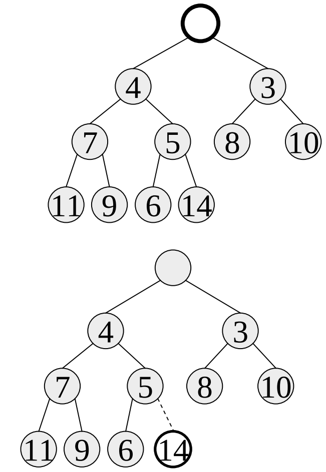
- FindMin: Easy!
  - › Return root value A[1]
  - › Run time = ?



- DeleteMin:
  - › Delete (and return) value at root node
    - How can we delete?

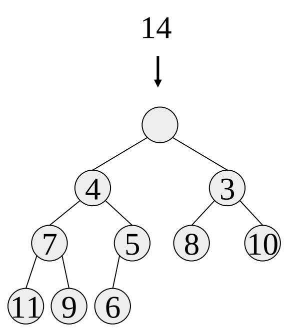
Maintain the Structure Property

- We now have a “Hole” at the root
  - › Need to fill the hole with another value
- When we get done, the tree will have one less node and must still be complete

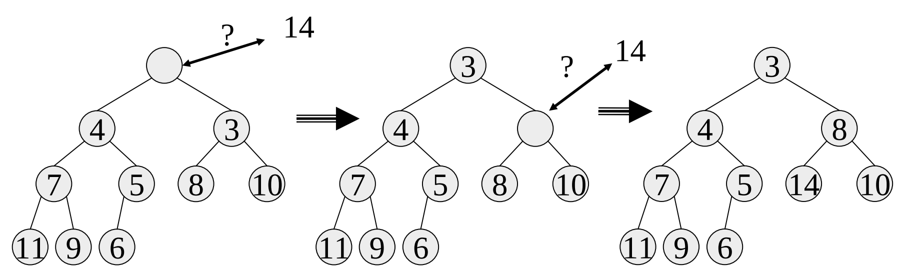


Maintain the Heap Property

- The last value has lost its node
  - › we need to find a new place for it
- We can do a simple insertion sort operation to find the correct place for it in the tree



DeleteMin: Percolate Down



- Keep comparing with children A[2i] and A[2i + 1]
- Copy smaller child up and go down one level
- Done if both children are ≥ item or reached a leaf node
- What is the run time?

Percolate Down

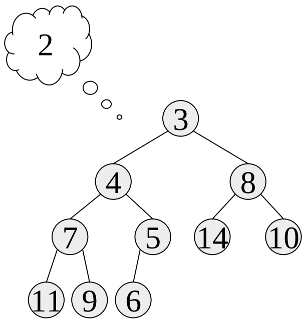
```
PercDown(i:integer, x :integer): {
// N is the number of entries in heap//
j : integer;
Case{
  2i > N : A[i] := x; //at bottom//
  2i = N : if (A[2i] < x) then
              A[i] := A[2i]; A[2i] := x;
           else A[i] := x;
  2i < N : if (A[2i] < A[2i+1]) then j := 2i;
           else j := 2i+1;
           if (A[j] < x) then
              A[i] := A[j]; PercDown(j,x);
           else A[i] := x;
}}
```


DeleteMin: Run Time Analysis

- Run time is O(depth of heap)
- A heap is a complete binary tree
- Depth of a complete binary tree of N nodes?
  - depth = $\lfloor \log_2(N) \rfloor$
- Run time of DeleteMin is O(log N)

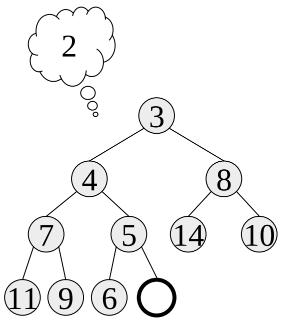
Insert

- Add a value to the tree
- Structure and heap order properties must still be correct when we are done



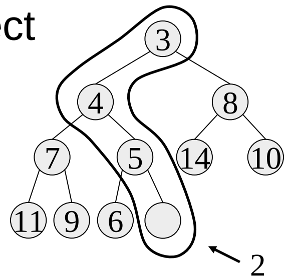
Maintain the Structure Property

- The only valid place for a new node in a complete tree is at the end of the array
- We need to decide on the correct value for the new node, and adjust the heap accordingly

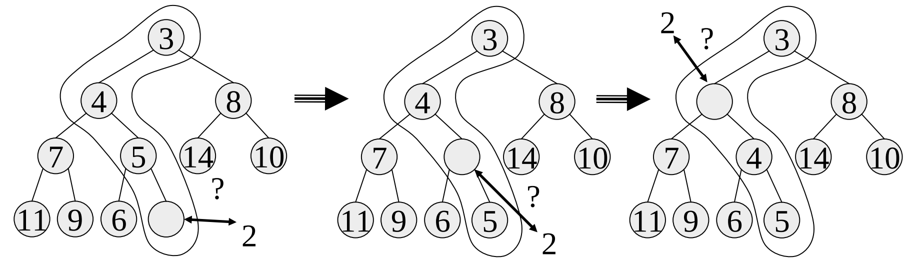


Maintain the Heap Property

- First, the new value goes to A[N+1] (and N is increased)
- Next, we find the correct place for it in the tree

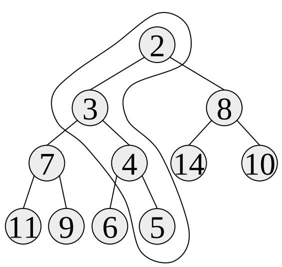


Insert: Percolate Up



- Start at last node and keep comparing with parent A[i/2]
- If parent larger, copy parent down and go up one level
- Done if parent ≤ item or reached top node A[1]
- Run time?

Insert: Done



- Run time?

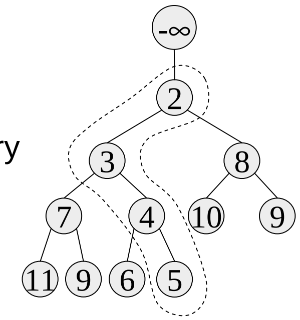
PercUp

- Define PercUp which percolates new entry to correct spot.
- Note: the parent of i is i/2

```
PercUp(i : integer, x : integer): {
????
}
```

Sentinel Values

- Every iteration of Insert needs to test:
  - if it has reached the top node A[1]
  - if parent ≤ item
- Can avoid first test if A[0] contains a very large negative value
  - sentinel -∞ < item, for all items
- Second test alone always stops at top



value	-∞	2	3	8	7	4	10	9	11	9	6	5		
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**

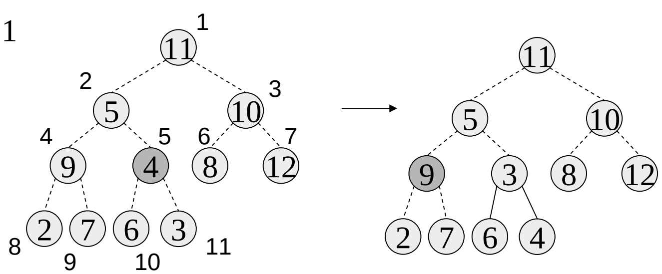
Binary Heap Analysis

- Space needed for heap of at most MaxN nodes: O(MaxN)
  - An array of size MaxN, plus a variable to store the current size N, plus an array slot to hold the sentinel
- Time
  - FindMin: O(1)
  - DeleteMin and Insert: O(log N)
  - BuildHeap from N inputs : O(N)

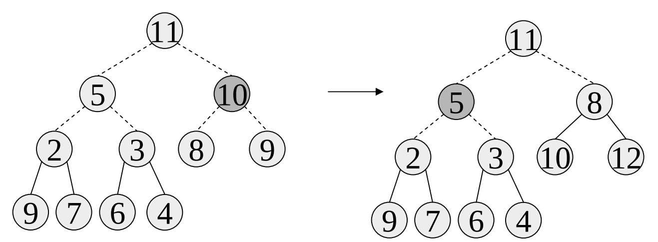
Build Heap

```
BuildHeap {
   for i = N/2 to 1 by –1 PercDown(i,A[i])
}
```

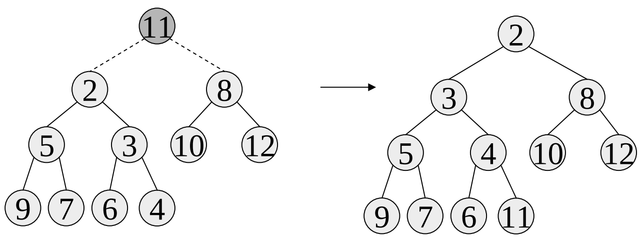
N=11



Build Heap



Build Heap



Analysis of Build Heap

- Assume $N = 2^K - 1$ (a full tree of height k)
  - › Level 1: k -1 steps for 1 item
  - › Level 2: k - 2 steps for 2 items
  - › Level 3: k - 3 steps for 4 items
  - › In general: Level i : k - i steps for 2^{i-1} items
  - › Until Level k-1: 1 step for 2^{k-2} items

$$\text{Total Steps} = \sum_{i=1}^{k-1} (k-i)2^{i-1} = 2^k - k - 1$$
$$= O(N)$$

Other Heap Operations

- Find(X, H): Find the element X in heap H of N elements
  - › What is the running time? O(N)
- FindMax(H): Find the maximum element in H
- Where FindMin is O(1)
  - › What is the running time? O(N)
- We sacrificed performance of these operations in order to get O(1) performance for FindMin
- How can we support FindMax in O(1)?
  - › Hint: double time and space complexity..

Other Heap Operations

- DecreaseKey(P,Δ): Decrease the key value of node at position P by a positive amount Δ, e.g., to increase priority
  - › First, subtract Δ from current value at P
  - › Heap order property may be violated
  - › so percolate up to fix
  - › Running Time: O(log N)

Other Heap Operations

- IncreaseKey(P,Δ): Increase the key value of node at position P by a positive amount Δ, e.g., to decrease priority
  - › First, add Δ to current value at P
  - › Heap order property may be violated
  - › so percolate down to fix
  - › Running Time: O(log N)

Other Heap Operations

- Delete(P): E.g. Delete a job waiting in queue that has been preemptively terminated by user
  - › Use DecreaseKey(P,∞) followed by DeleteMin
  - › Running Time: O(log N)

Other Heap Operations

- Merge(H1,H2): Merge two heaps H1 and H2 of size O(N). H1 and H2 are stored in two arrays.
  - › Can do O(N) Insert operations: O(N log N) time
  - › Better: Copy H2 at the end of H1 and use BuildHeap. Running Time: O(N)

PercUp Solution

```
PercUp(i : integer, x : integer): {
if i = 1 then A[1] := x
else if A[i/2] < x then
       A[i] := x;
     else
       A[i] := A[i/2];
       Percup(i/2,x);
}
```