Binary Heaps

COL 106

Shweta Agrawal and Amit Kumar

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?
  - Hash Tables: 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)
- Hash Tables
  - Insert O(1) but no hope for FindMin
- BSTs look good but…
  - BSTs are efficient for all Finds, not just FindMin
  - We only need FindMin

Better than a speeding BST

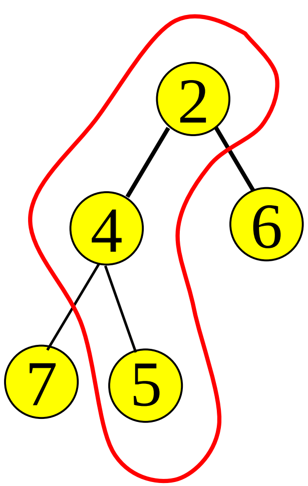
- We can do better than Balanced Binary Search Trees?
  - Very limited requirements: Insert, FindMin, DeleteMin. The goals are:
  - FindMin is O(1)
  - Insert is O(log N)
  - DeleteMin is O(log N)

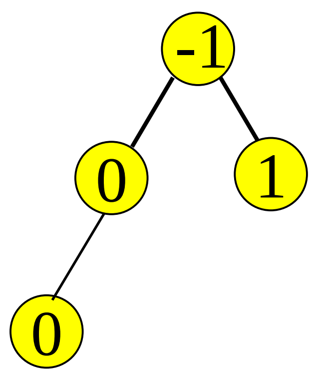
Binary Heaps

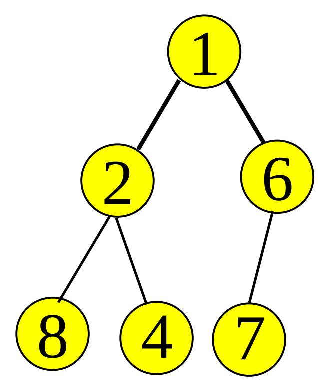
- A binary heap is a binary tree (NOT a BST) that is:
  - Complete: the tree is completely filled except possibly the bottom level, which is filled from left to right
  - Satisfies the heap order property
    - every node is less than or equal to its children
    - or every node is greater than or equal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

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



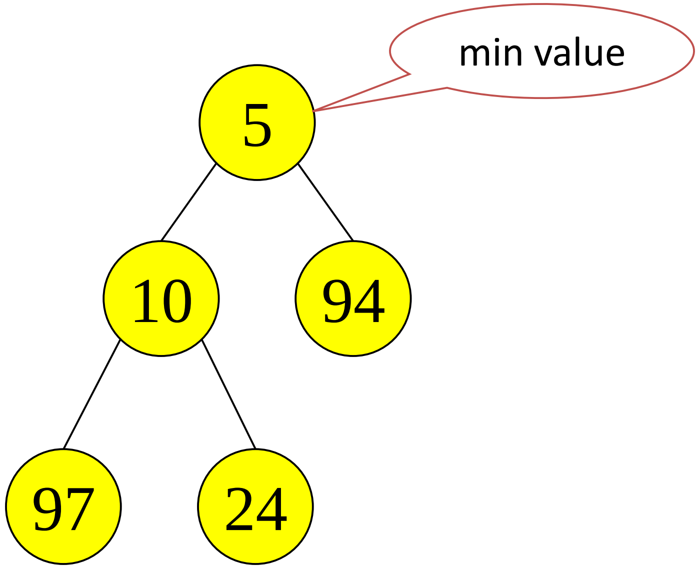




These are all valid binary heaps (minimum)

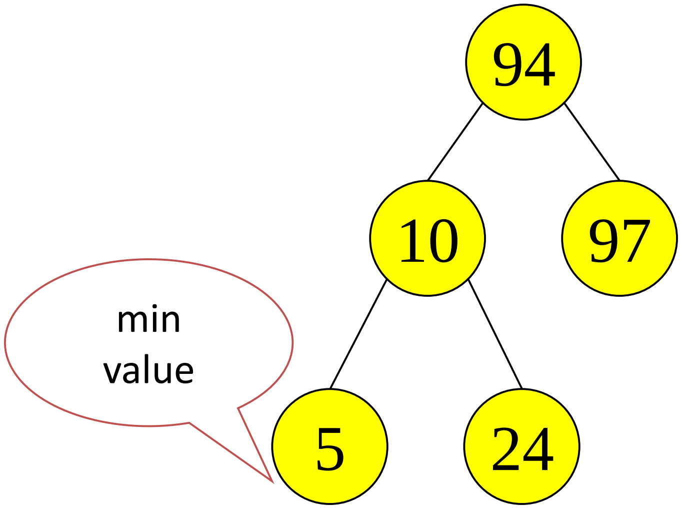
Binary Heap vs Binary Search Tree

Binary Heap



Parent is less than both left and right children

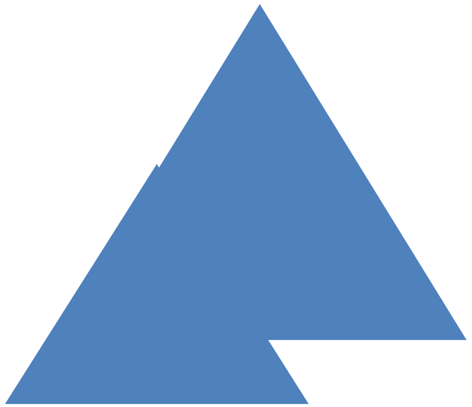
Binary Search Tree



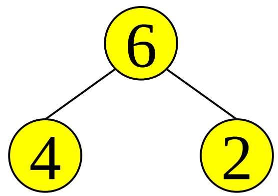
Parent is greater than left child, less than right child

Structure property

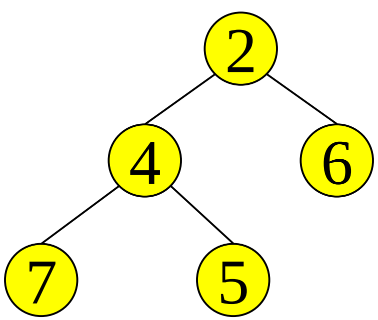
- A binary heap is a complete tree
  - All nodes are in use except for possibly the right end of the bottom row



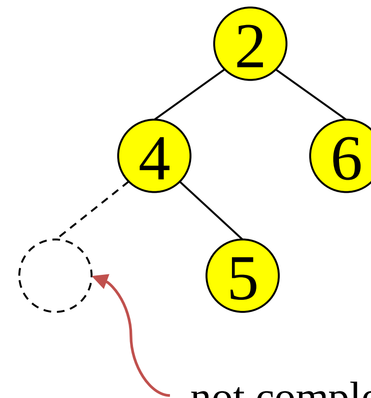
Examples



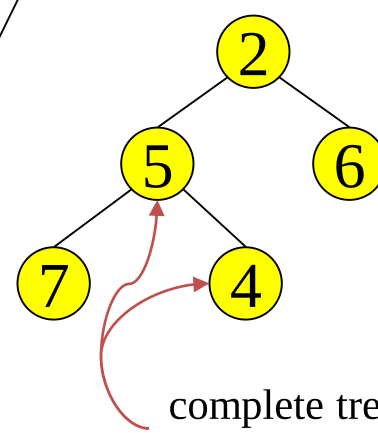
complete tree,
heap order is "max"



complete tree,
heap order is "min"



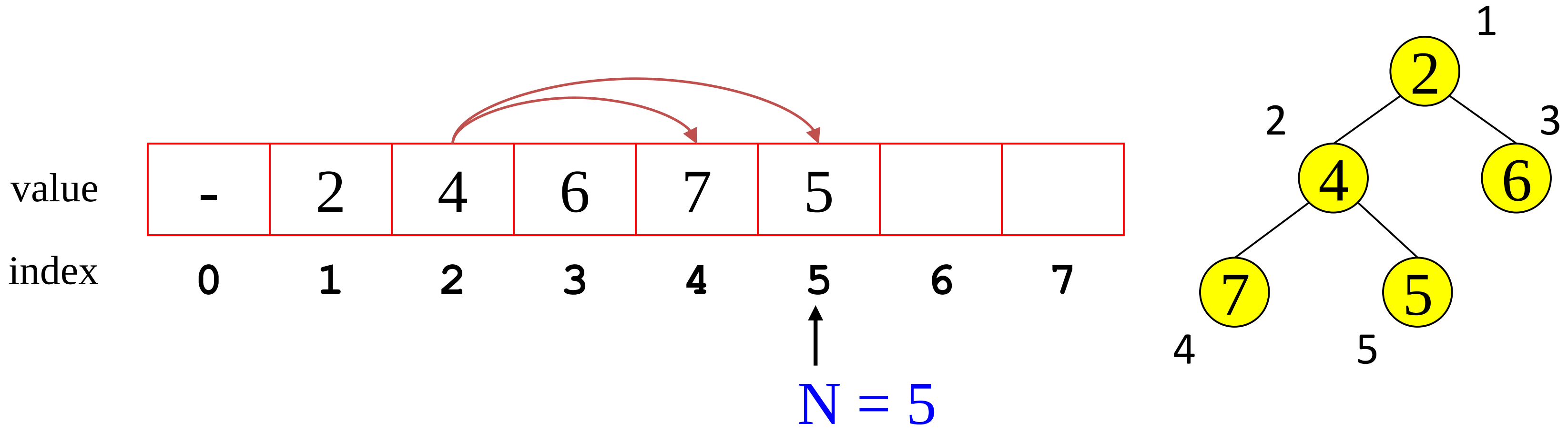
not complete



complete tree, but min
heap order is broken

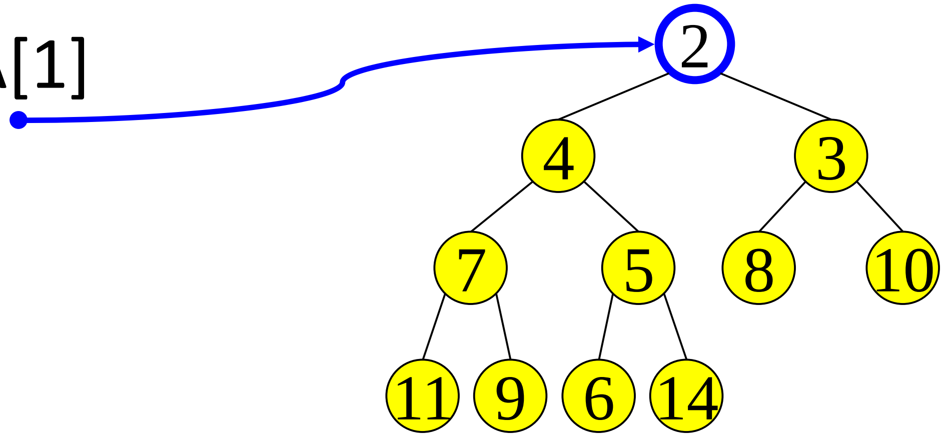
Array Implementation of Heaps (Implicit Pointers)

- Root node = A[1]
- Children of A[i] = A[2i], A[2i + 1]
- Parent of A[j] = A[j/2]
- Keep track of current size N (number of nodes)



FindMin and DeleteMin

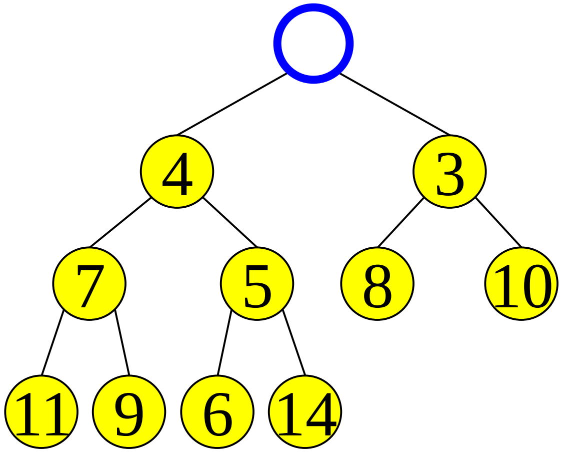
- FindMin: Easy!
  - Return root value A[1]
  - Run time = ?



- DeleteMin:
  - Delete (and return) value at root node

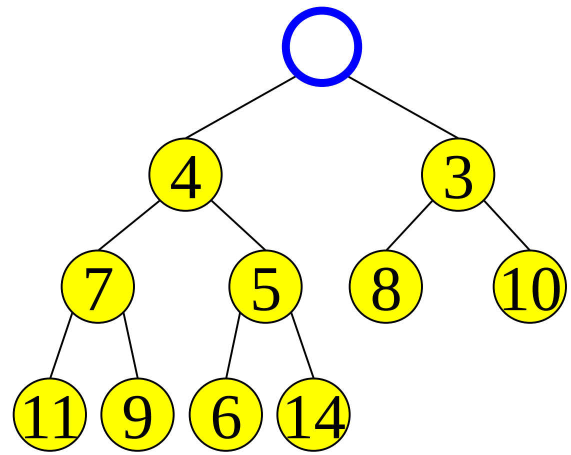
DeleteMin

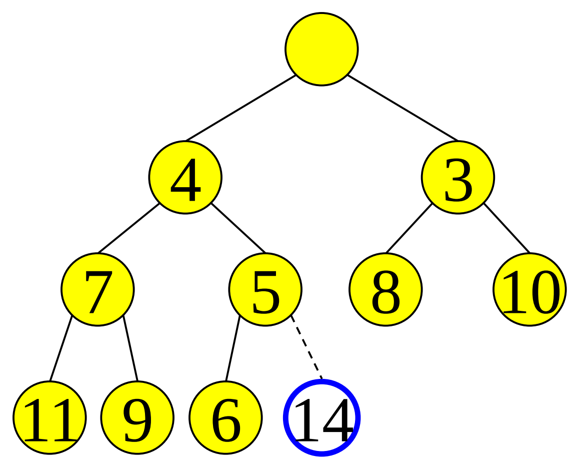
- Delete (and return) value at root node



Maintain the Structure Property

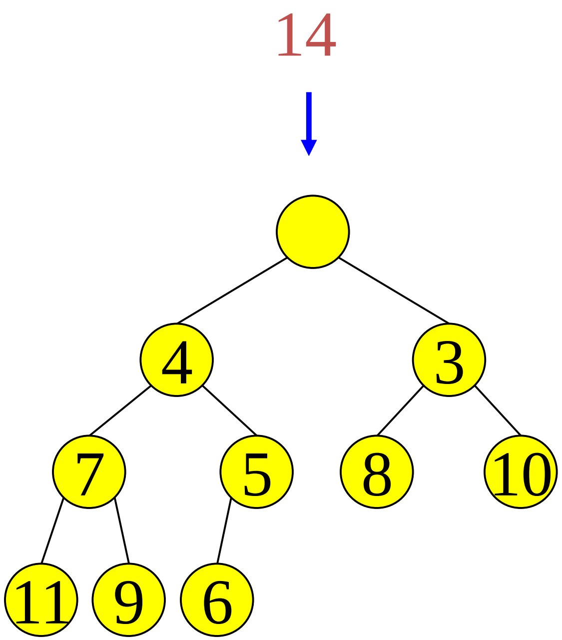
- We now have a “Hole” at the root
  - Need to fill the hole with another value
- When we get done, the tree will have one less node and must still be complete



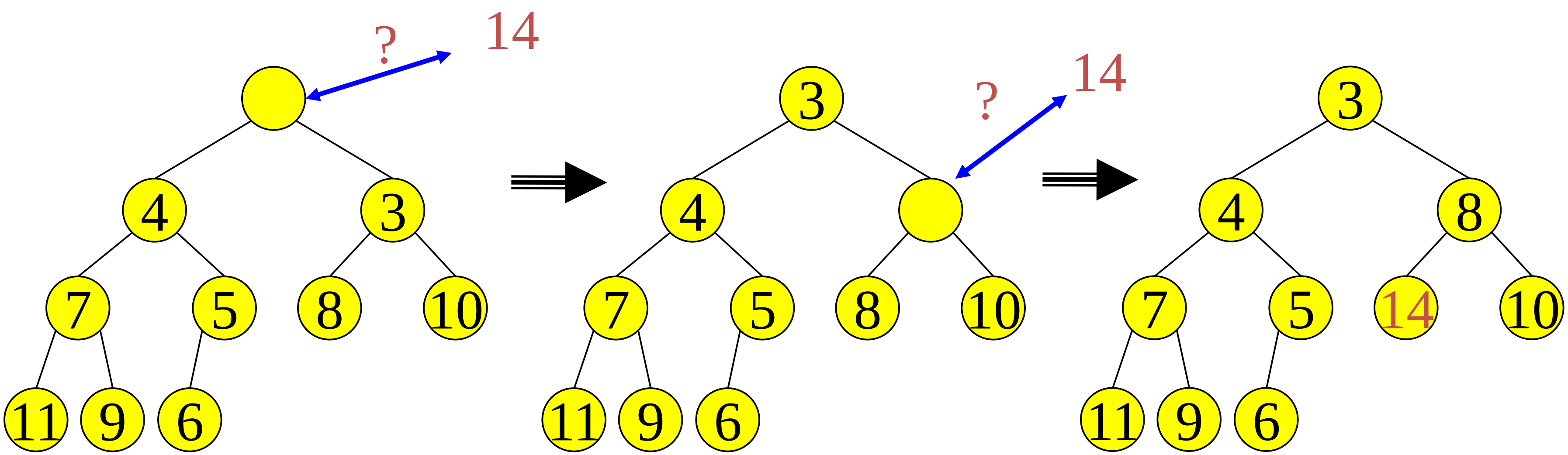


Maintain the Heap Property

- The last value has lost its node
  - we need to find a new place for it



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?

1	2	3	4	5	6
~~6~~	10	8	13	14	25

Percolate Down

```
PercDown(i:integer, x: integer): {
// N is the number elements, i is the hole,
   x is the value to insert
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;
}}
```

no children

one child
at the end

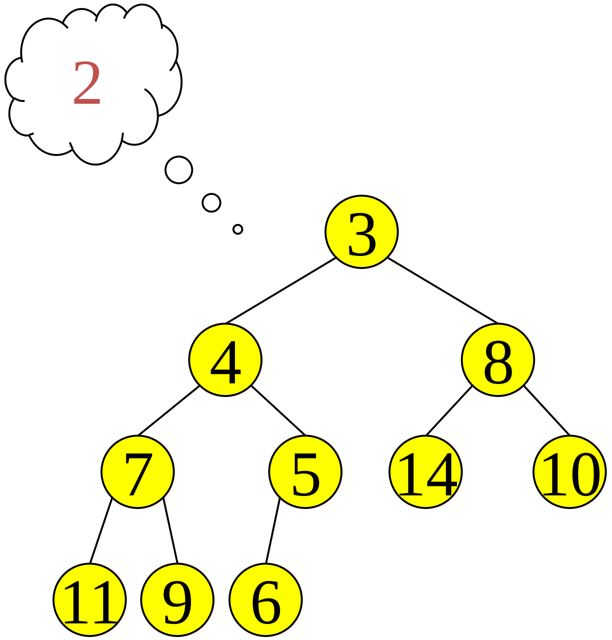
2 children

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 = $\log_2(N)$
- Run time of DeleteMin is O(log N)

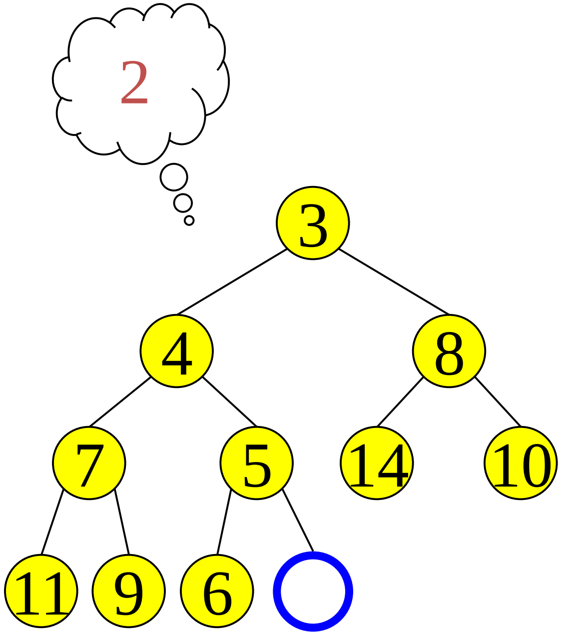
Insert

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



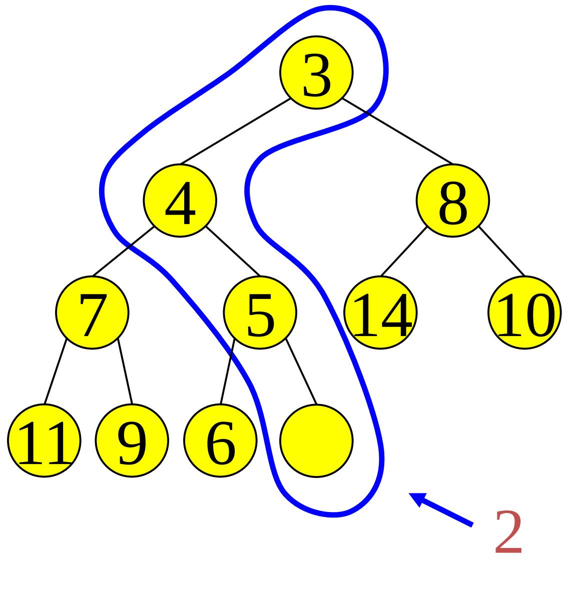
Maintain the Structure Property

- 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

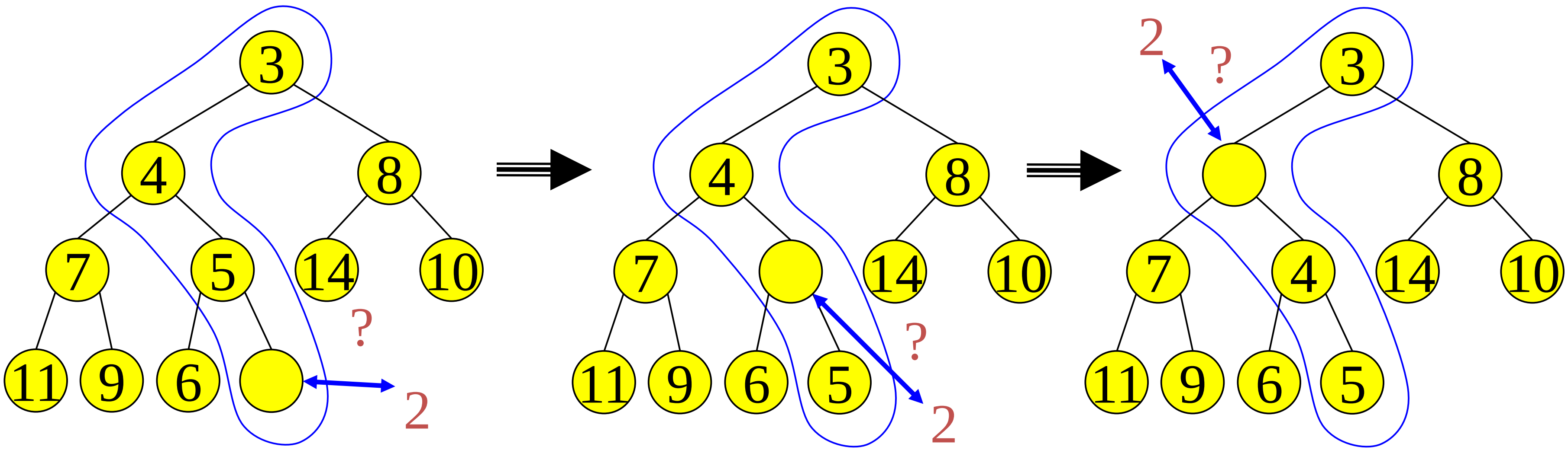


Maintain the Heap Property

- The new value goes where?

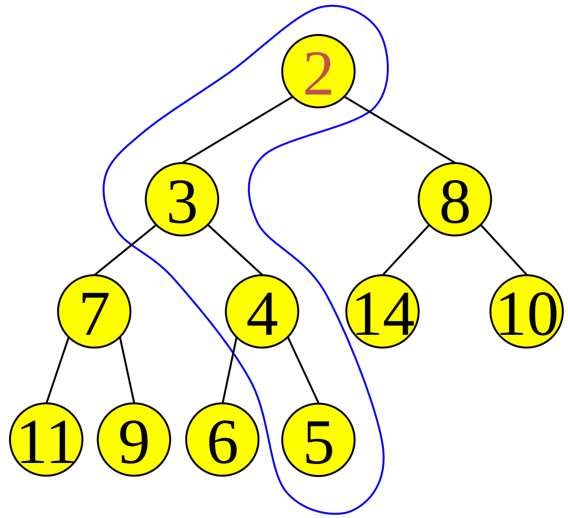


Insert: Percolate Up



- 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]

Insert: Done



- Run time?

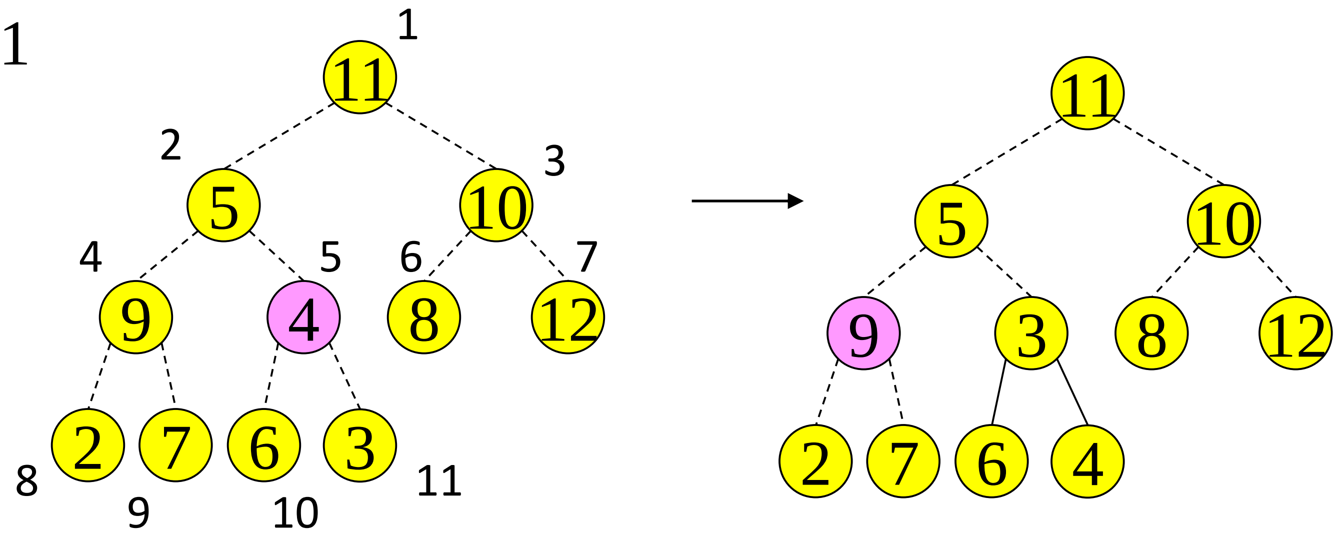
Binary Heap Analysis

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

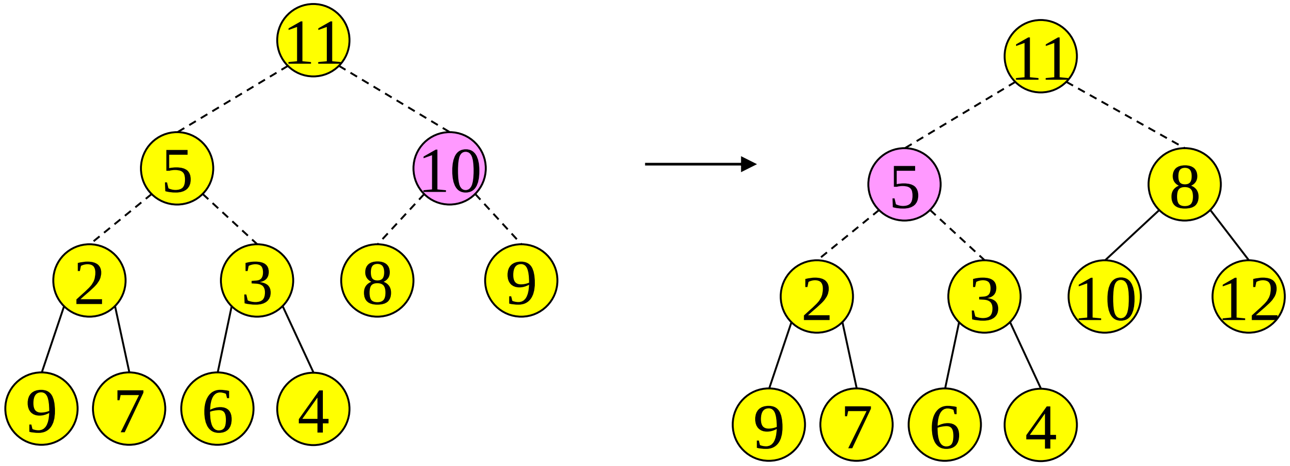
Build Heap

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

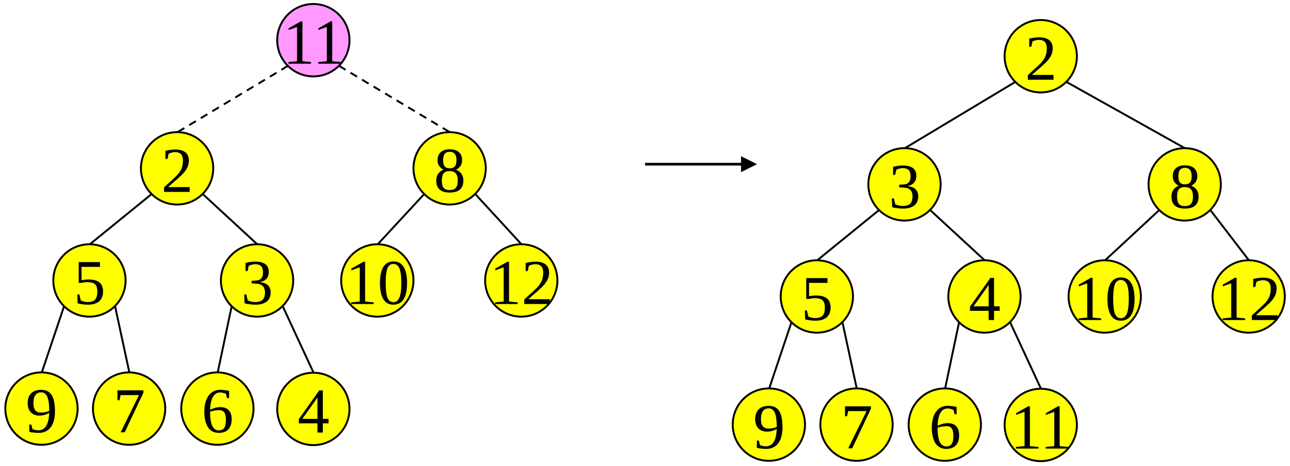
N=11



Build Heap



Build Heap



Time Complexity

- Naïve considerations:
  - **`n/2 calls to PercDown, each takes clog(n)`**
  - **`Total:`** $\mathbf{cn\log(n)}$
- More careful considerations:
  - Only O(n)

Analysis of Build Heap

- Assume $n = 2^{h+1} - 1$ where h is height of the tree
  - Thus, level h has 2^h nodes but there is nothing to PercDown
  - At level h-1 there are 2^{h-1} nodes, each might percolate down 1 level
  - At level h-j, there are 2^{h-j} nodes, each might percolate down j levels

Total Time

$$T(n) = \sum_{j=0}^{h} j 2^{h-j} = \sum_{j=0}^{h} j \frac{2^h}{2^j}.$$

$= O(n)$

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

Other Heap Operations

- DecreaseKey(P,Δ,H): 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,H): 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,H): E.g. Delete a job waiting in queue that has been preemptively terminated by user
  - Use DecreaseKey(P, Δ,H) 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)

Heap Sort

- Idea: `buildHeap` then call `deleteMin` *n* times

```
E[] input = buildHeap(...);
E[] output = new E[n];
for (int i = 0; i < n; i++) {
    output[i] = deleteMin(input);
}
```

- Runtime?

  Best-case ___ Worst-case ___ Average-case ___
- Stable? _____
- In-place? _____

Heap Sort

- Idea: `buildHeap` then call `deleteMin` *n* times

```
E[] input = buildHeap(...);
E[] output = new E[n];
for (int i = 0; i < n; i++) {
    output[i] = deleteMin(input);
}
```

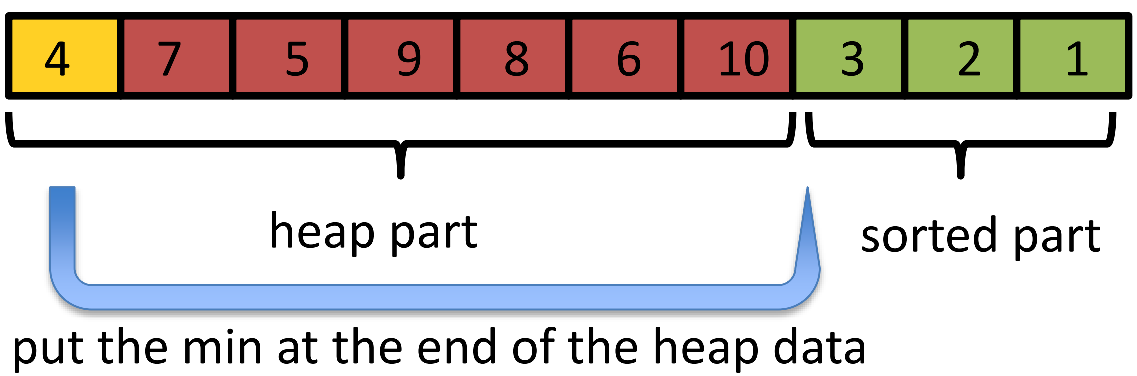
- Runtime?

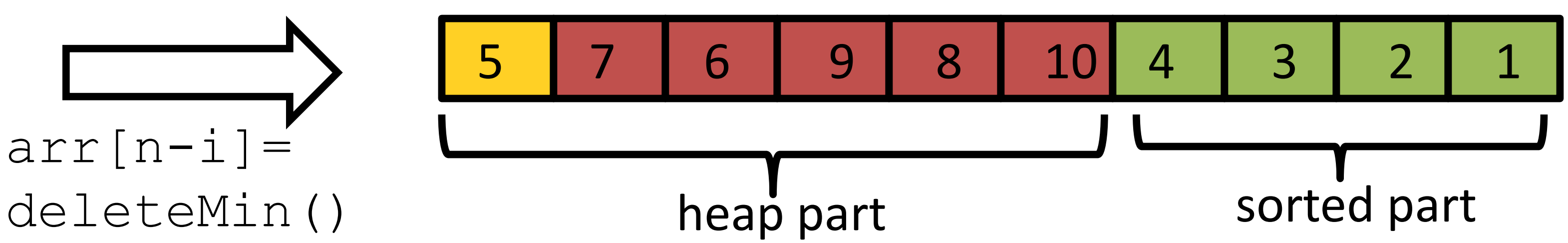
  Best-case, Worst-case, and Average-case: O(n log(n))
- Stable? No
- In-place? No. But it could be, with a slight trick...

In-place Heap Sort

But this reverse sorts – how would you fix that?

- Treat the initial array as a heap (via **`buildHeap`**)
- When you delete the **`i`**th element, put it at **`arr[n-i]`**
  - That array location isn't needed for the heap anymore!





"AVL sort"? "Hash sort"?

AVL Tree: sure, we can also use an AVL tree to:

- `insert` each element: total time $O(n \log n)$
- Repeatedly `deleteMin`: total time $O(n \log n)$
  - Better: in-order traversal $O(n)$, but still $O(n \log n)$ overall
- But this cannot be done in-place and has worse constant factors than heap sort