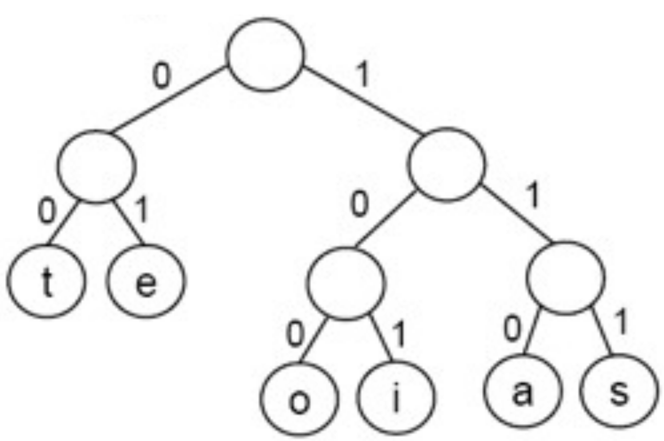
Programming Assignment #2: Using Huffman Encoding to Compress A File

- Read through the input file and build its Huffman tree.
- Traverse the Huffman tree to create a table containing the encoding of each character. Print out the table.
- Read through the input file a second time, and write the Huffman code for each character to the output file. Modify TA-supplied routines to write bits “0” and “1”, to output file. Compute the space savings.
- Write routine to read Huffman code and write out text to output file.



a	110
e	01
i	101
o	100
s	111
t	00

Heaps and Priority Queues

EECS 233

Queue ADT Revisited

- A queue is a sequence in which:
  - items are added at the rear and removed from the front
    - first in, first out (FIFO) (vs. a stack, which is last in, first out)
  - we can only access the item that is currently at the front
- Operations:
  - boolean insert(T item); add an item at the rear of the queue
  - T remove(); remove the item at the front of the queue
  - T peek(); get the item at the front of the queue, but don't remove it
  - boolean isEmpty(); test if the queue is empty
  - boolean isFull(); test if the queue is full

Priority Queues

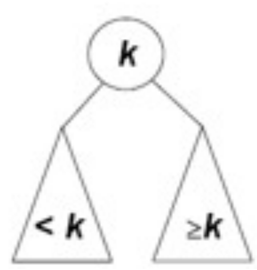
- A priority queue is a collection of items, each of which has an associated *priority* (a number).
  - Applications?

- Operations:
  - insert: add an item to the priority queue (with a priority value)
  - remove: remove the highest-priority item
    - ✓ the item in the queue with the largest associated priority value
  - …

- How can we efficiently implement a priority queue?
  - Unsorted list, sorted list, sorted array?
  - AVL-tree?
  - (A new type of binary tree known as a *heap)*

Tree Types and Characteristics

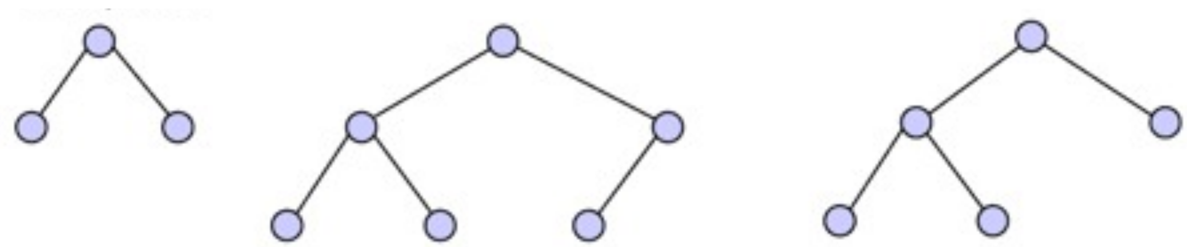
- Structural constraints
  - Constraints on number of children
    - ✓ Binary trees (at most 2 children)
    - ✓ B-trees (between M/2 and M children)
  - Balance constraints
    - ✓ Binary search trees - unconstrained
    - ✓ AVL-trees - differ in subtree heights by at most 1
- Key order constraints
  - Binary search trees and AVL trees (keys in the left subtree are less then, and keys in the right subtree are greater than or equal to a node’s key)



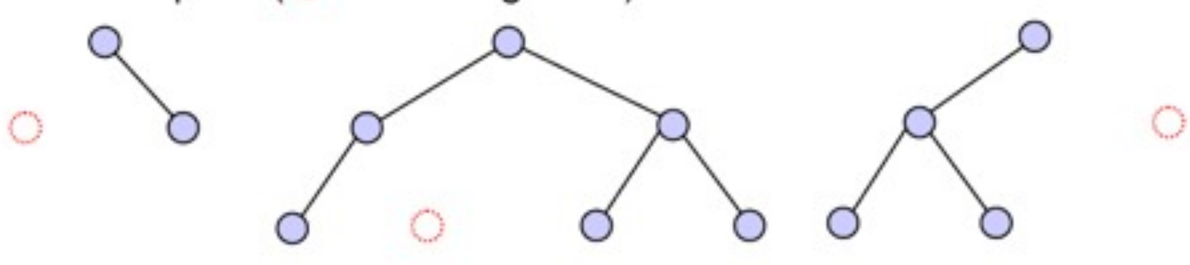
Structure Types of Binary Trees

- Full binary trees
  - Every node has exactly two or zero children
- Complete binary trees
  - Balanced, and at the same level, filled left to right

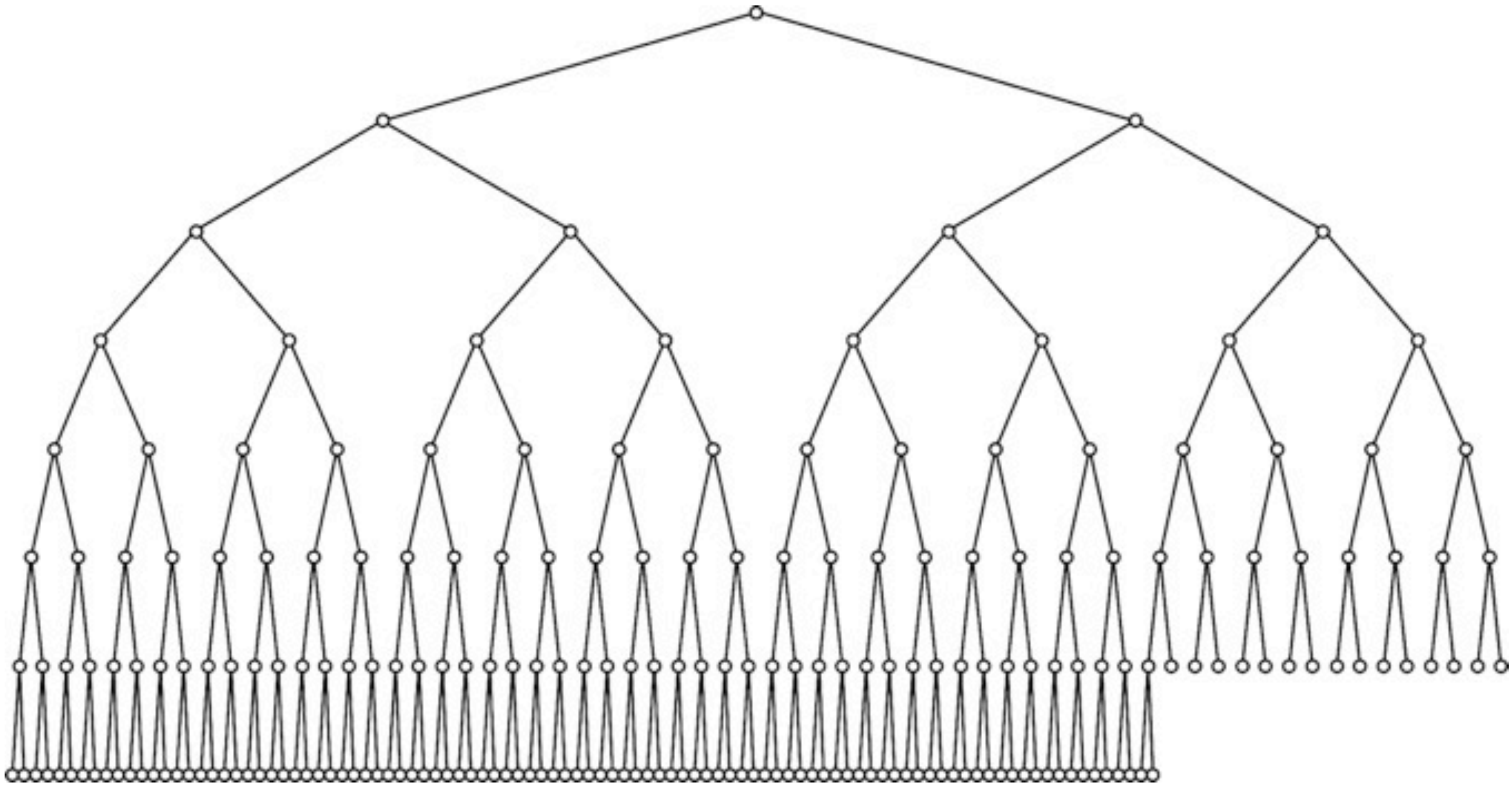
Complete:



Not complete (○ = missing node):



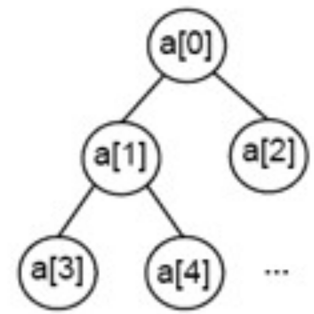
A Large Complete Binary Tree

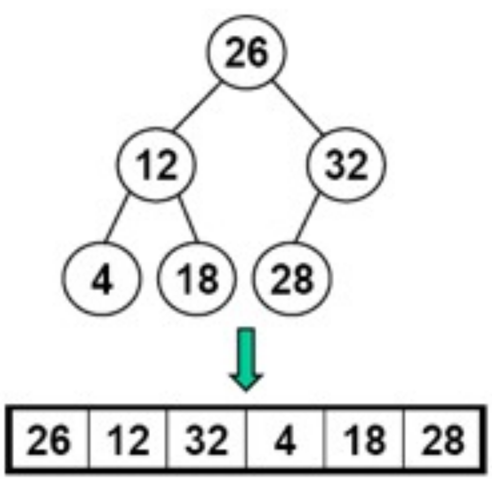


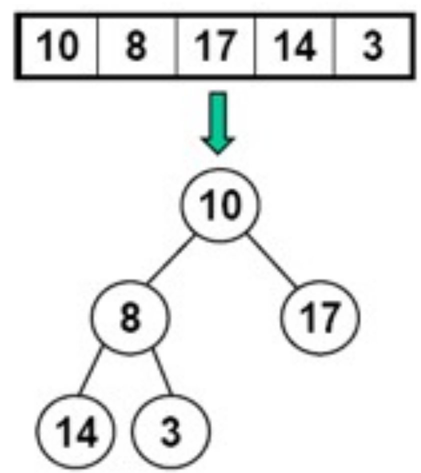
--

Array Representation of A Complete Binary Tree

- A complete binary tree has a simple array representation.
- The nodes of the tree are stored in the array in the order in which they would be visited by a level-order traversal (i.e., top to bottom, left to right).

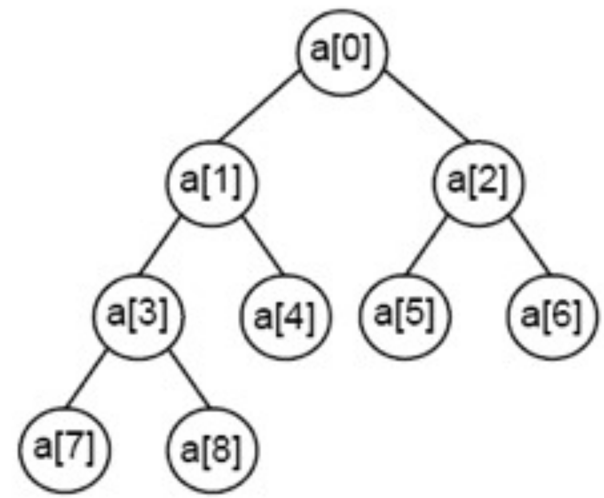






Navigating in A Complete Binary Tree

- The root node is in a[0]
- Given the node in a[i]:
  - its left child is in a[2*i + 1]
  - its right child is in a[2*i + 2]
  - its parent is in a[(i – 1)/2], i!=0

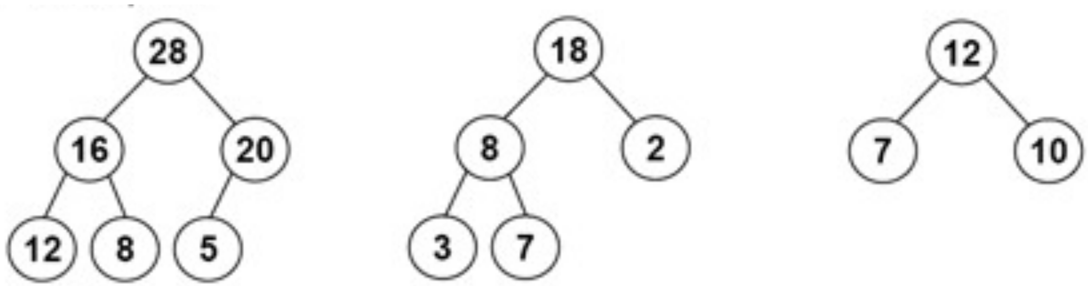


A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

- Examples:
  - the left child of the node in a[1] is in a[2*1 + 1] = a[3]
  - the right child of the node in a[3] is in a[2*3 + 2] = a[8]
  - the parent of the node in a[4] is in a[(4–1)/2] = a[1]
  - the parent of the node in a[7] is in a[(7–1)/2] = a[3]

Heaps for Priority Queues

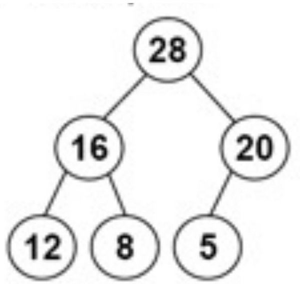
- A complete binary tree
  - The only ordering constraint: a node's key >= keys of children (if any)
- The largest value is always at the root of the tree.
- The smallest value can be in *any* leaf node – there's no guarantee about which one it will be.

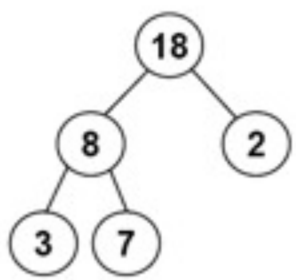


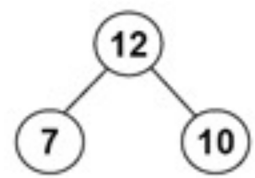
- These are max-at-top heaps.
- One can also define a min-at-top heap, in which every interior node is less than or equal to its children.

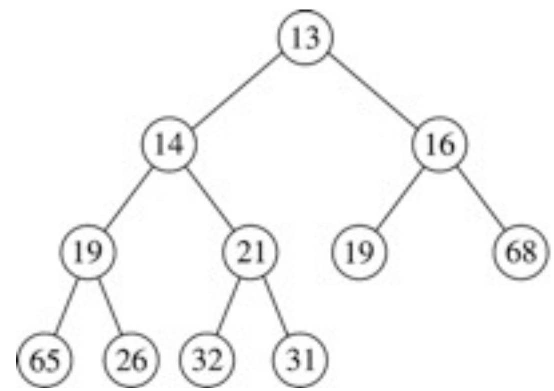
Heaps for Priority Queues

- A complete binary tree
  - The only ordering constraint: a node's key >= keys of children (if any)
- The largest value is always at the root of the tree.
- The smallest value can be in *any* leaf node – there's no guarantee about which one it will be.









- These are max-at-top heaps.
- One can also define a min-at-top heap, in which every interior node is less than or equal to its children.

Heap ADT - Generics

- We want a heap that can store data items of different types (just like generic lists, stacks, and queues)

- Writing a generic heap is somewhat tricky: we need to make sure that we can compare the data items.

  if (item1 < item2)

  …

- Problem
  - In Java, if item1 and item2 refer to objects, the condition above will compare the memory locations of objects, not the actual items of the objects.

Generic Heaps – Comparable Objects in Java

- To compare objects in Java, we need to use a method called compareTo().
  - many built-in classes have a version of this method, e.g., String, Integer, Double, etc.
  - we can define a version of this method in classes that we write

    public int compareTo(*Classname* other)

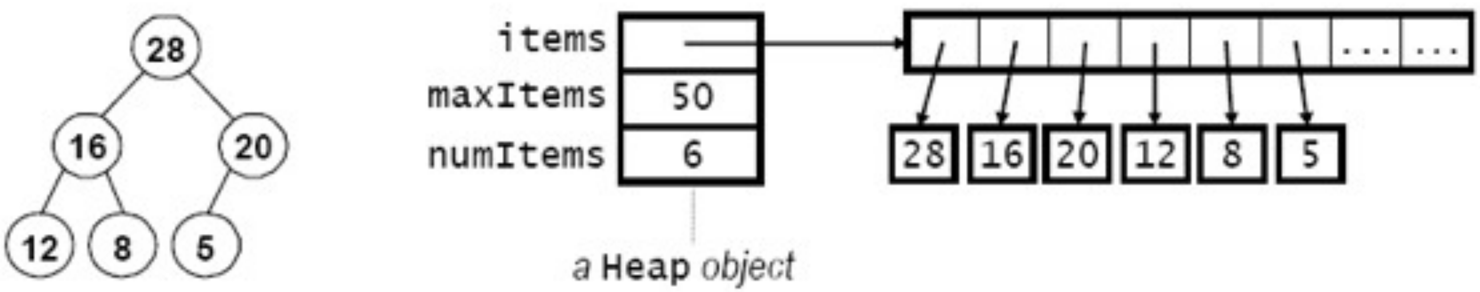
- Then, o1.compareTo(o2) should return:
  - a negative integer, e.g., -1, if o1 is less than o2
  - 0 if o1 equals o2
  - a positive integer, e.g., +1, if o1 is greater than o2

- We also need to indicate that our class has this method by adding the following to the class header:

    class *Classname* implements Comparable<*Classname*>

Generic Heaps in Java

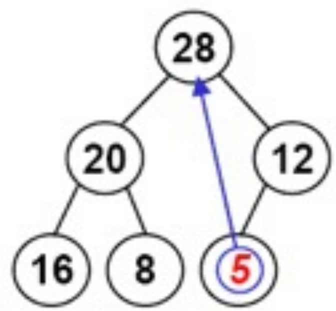
```
public class Heap<T extends Comparable<T>> {
        private T[] items;
        private int maxItems;
        private int numItems;
        public Heap(int maxSize) {
                items = (T[])new Comparable[maxSize];
                maxItems = maxSize;
                numItems = 0;
        }
        …
}
```



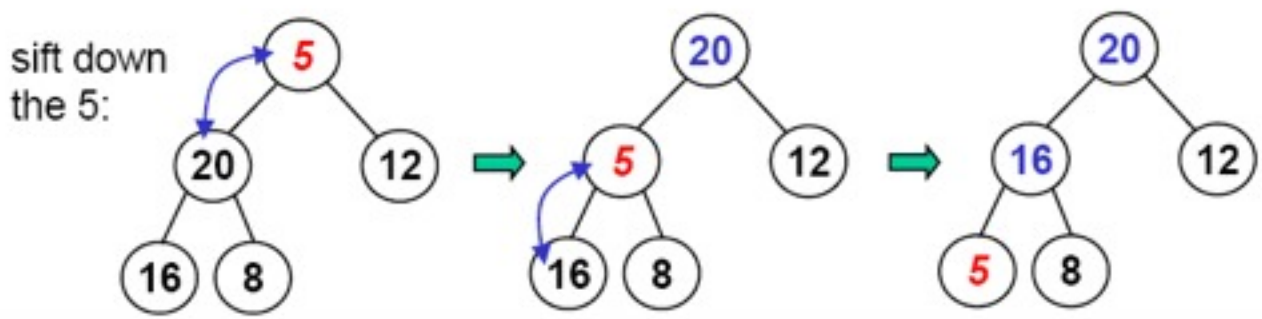
- Suitable for objects of *any* type T that has a compareTo() method and contains “implements Comparable<T>” in its class header. The array actually stores the references

Removing An Item (the largest)

- Remove and return the item in the root node.
  - In addition, we need to move the largest remaining item to the root, while maintaining a complete tree with each node >= children
- Method:
  - make a copy of the largest item
  - move the last item in the heap to the root (see diagram at right)
  - “sift down” the new root item until it is >= its children (or it’s a leaf)
  - return the largest item

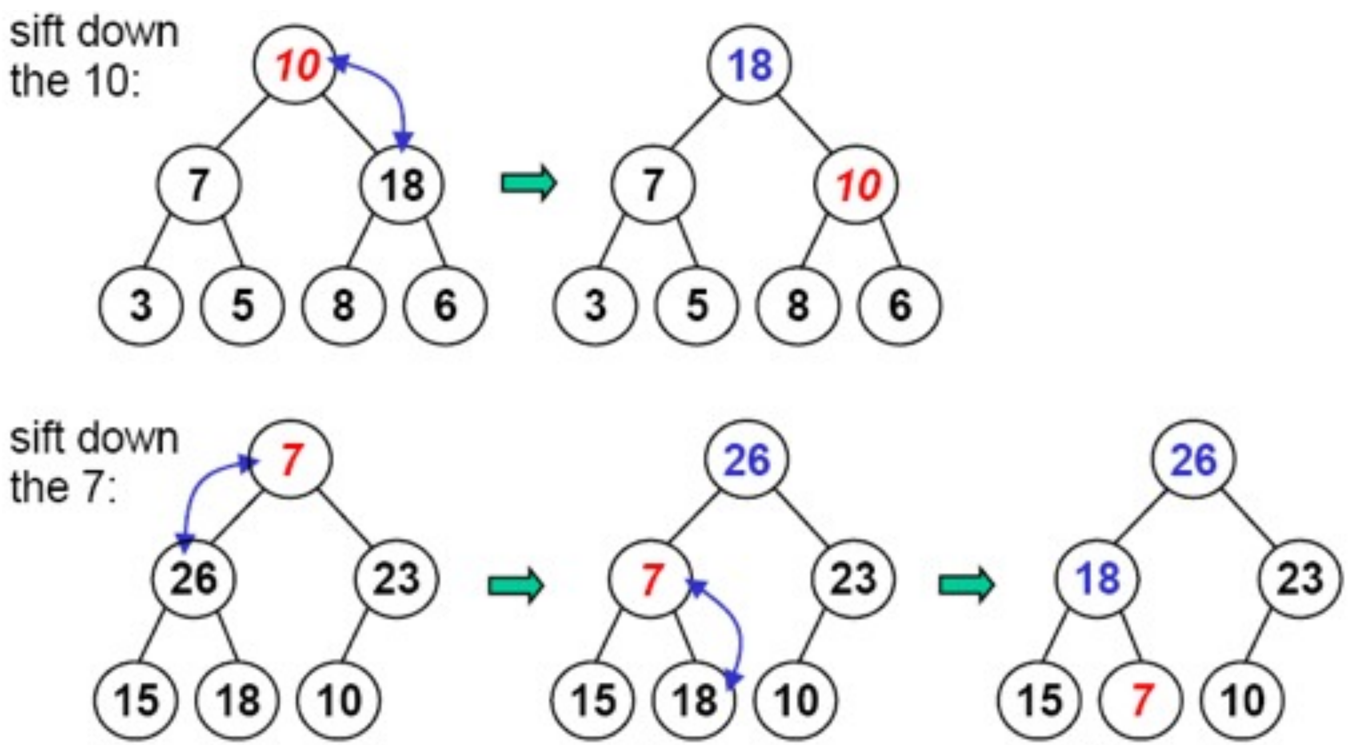


- “sift”: items are filtered such that small ones will fall



"Heapify" – Sifting Down

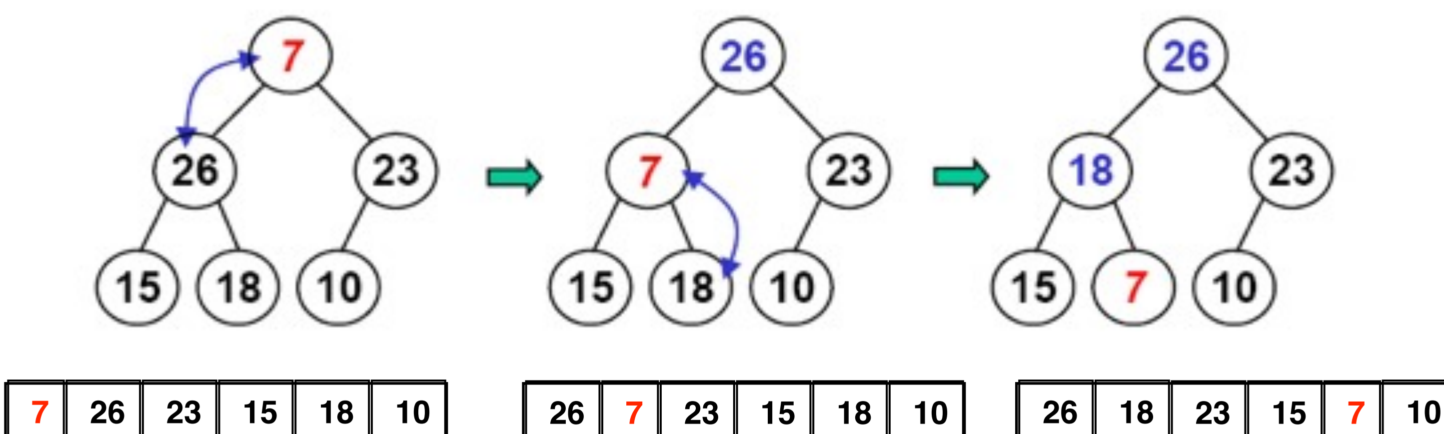
- To sift down item *x* (i.e., the item whose key is *x*):
  - compare *x* with the larger of the item's children, *y*
  - if $x < y$, swap *x* and *y* and repeat



SiftDown Method

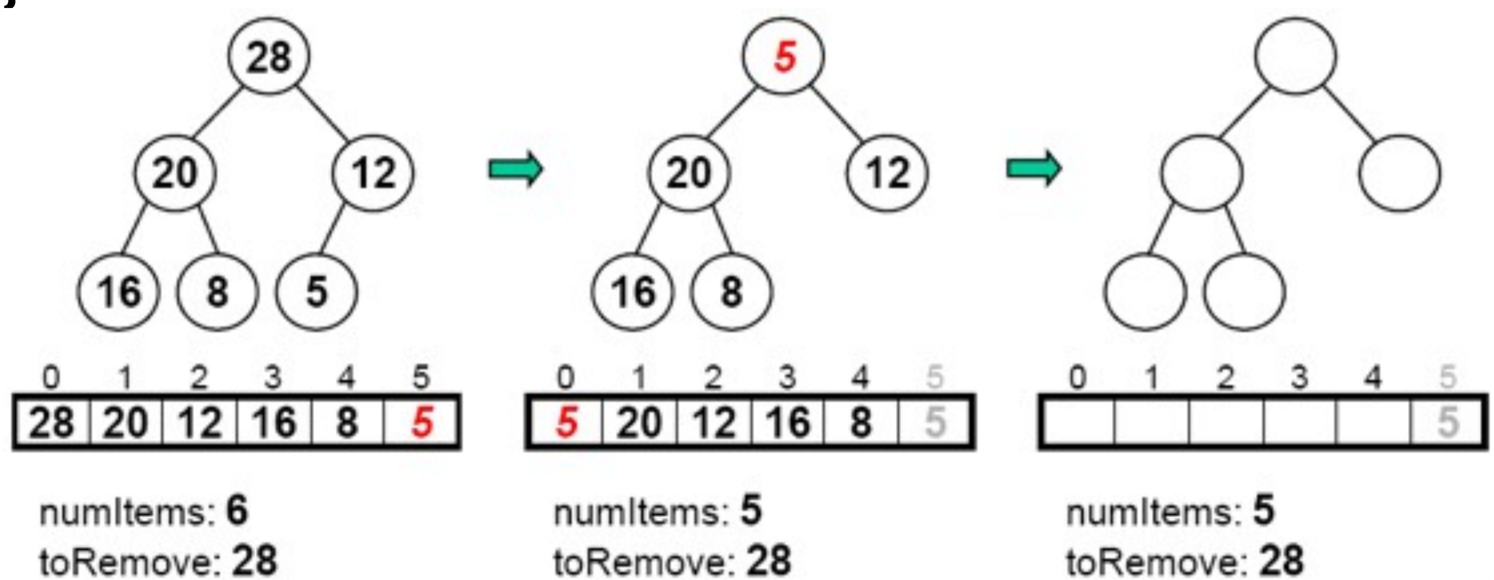
```
private void siftDown(int i) { // Input: the node to sift
        T toSift = items[i];
        int parent = i;
        int child = 2 * parent + 1; // Child to compare with; start with left child
        while (child < numItems) {
                // If the right child is bigger than the left one, use the right child for comparison.
                if (child < numItems - 1 && // if the right child exists
                        items[child].compareTo(items[child + 1]) < 0)  // … and is bigger
                                child = child + 1; // take the right child
                if (toSift.compareTo(items[child]) >= 0)
                        break; // we're done
                // Sift down one level in the tree.
                items[parent] = items[child];
                items[child] = toSift;
                parent = child;
                child = 2 * parent + 1;
        }
        items[parent] = toSift;
}
```

We don't have to put sifted item in place of child. We can wait until the end to put the sifted item in place.



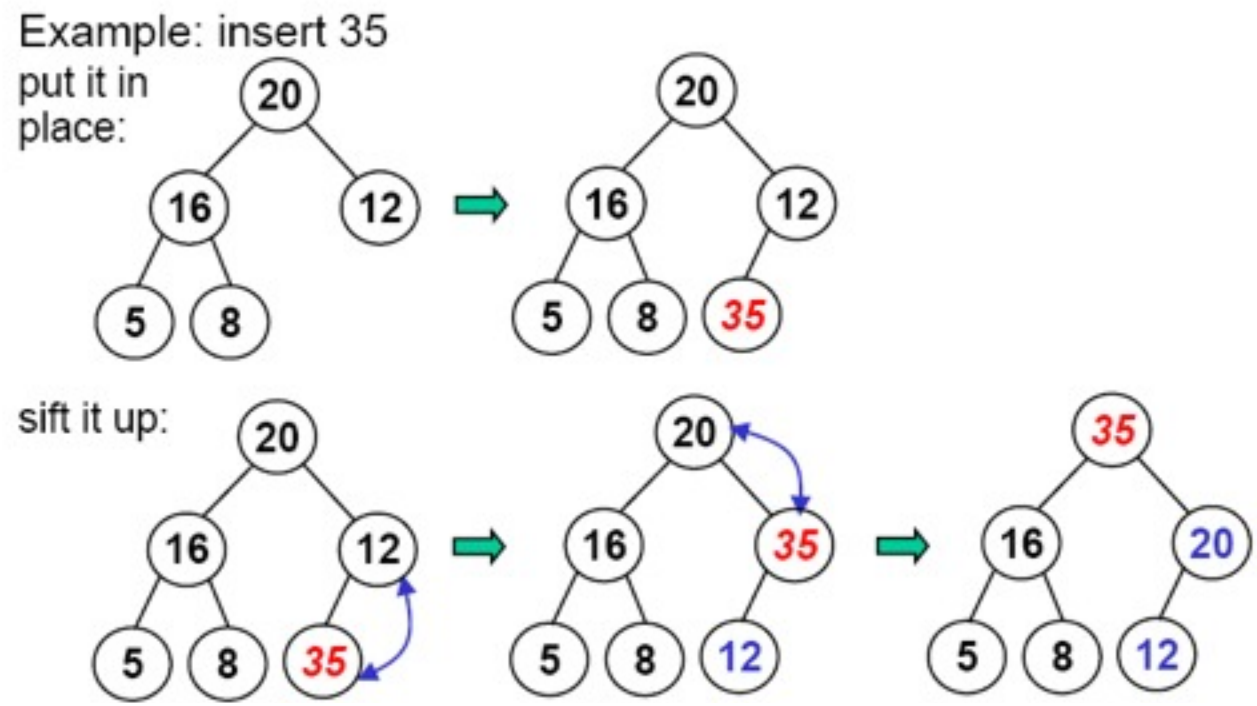
RemoveMax Method

```
public T removeMax() {
    T toRemove = items[0];
    items[0] = items[numItems-1];
    numItems--;
    siftDown[0];
    return toRemove;
}
```



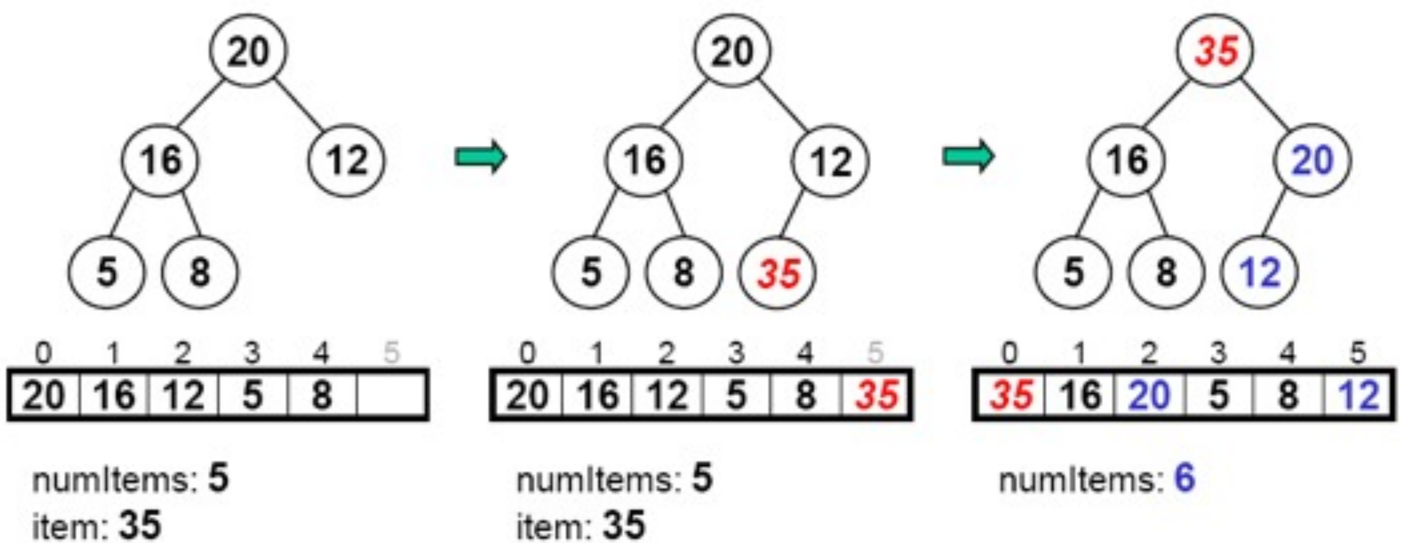
Inserting An Item

- Algorithm:
  - put the item in the next available slot (grow array if needed)
  - “sift up” the new item until it is <=its parent (or becomes the root)



Insert Method

```
public void insert(T item) {
    if (numItems == maxItems) {
        // code to grow the array goes here…
    }
    Items[numItems] = item;
    numItems++;
    siftUp[numItems-1];
}
```



SiftUp Method

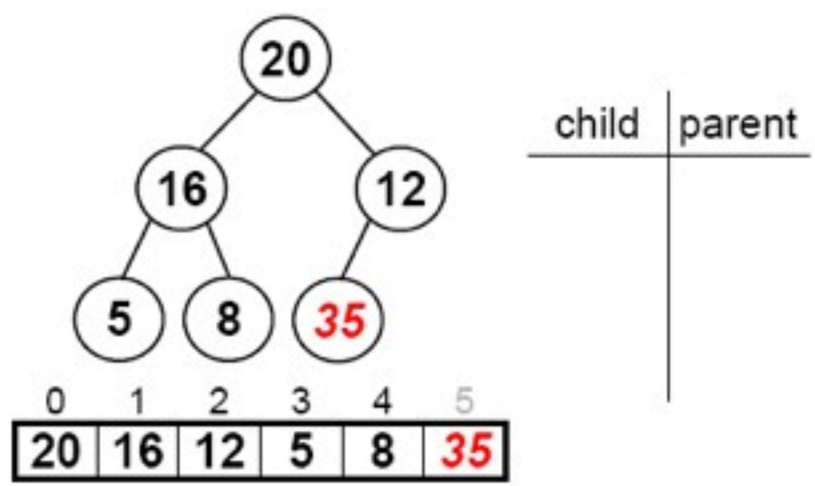
```
private void siftUp(int i) {
      int parent = (i-1)/2;
      int child = i;
      while ( ? ) {



            ?




      }
}
```

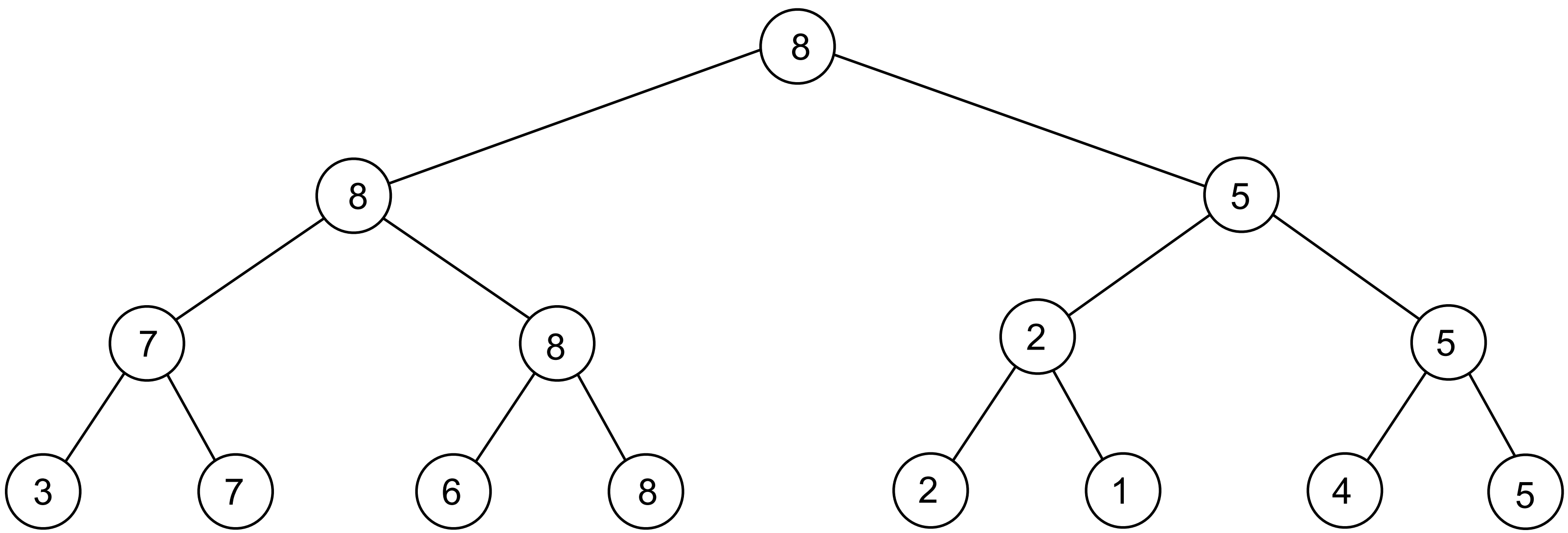


Running Time of Insert / Remove

- Insert() / RemoveMax()
- SiftUp and SiftDown
  - Height $h = \log_2 N$
  - Running time $O(\log_2 N)$ in the worst case
- Both insert and remove (max at the root) are fast
- But search is inefficient due to limited ordering information

Binary Heaps and Tournament Trees

- A field of 2^h players (teams), each with a strength value
- A player with a higher strength value will win and advance to the next round; the loser goes home (playoff events, e.g., baseball)
- Limited ordering information: we can only tell who is the champion; the runner-up may not be the 2nd best.



Building A Heap – A Naïve Algorithm

- Take *N* items from an array and build a heap.
- Naïve algorithm
  - For each item, insert it into a heap, which is initially empty

```
public void buildHeap(T[ ] array, int size )
{
      < initialize the heap here …>
       for( int i = 0; i < size; i++ )
            insert(array[i]);
}
```

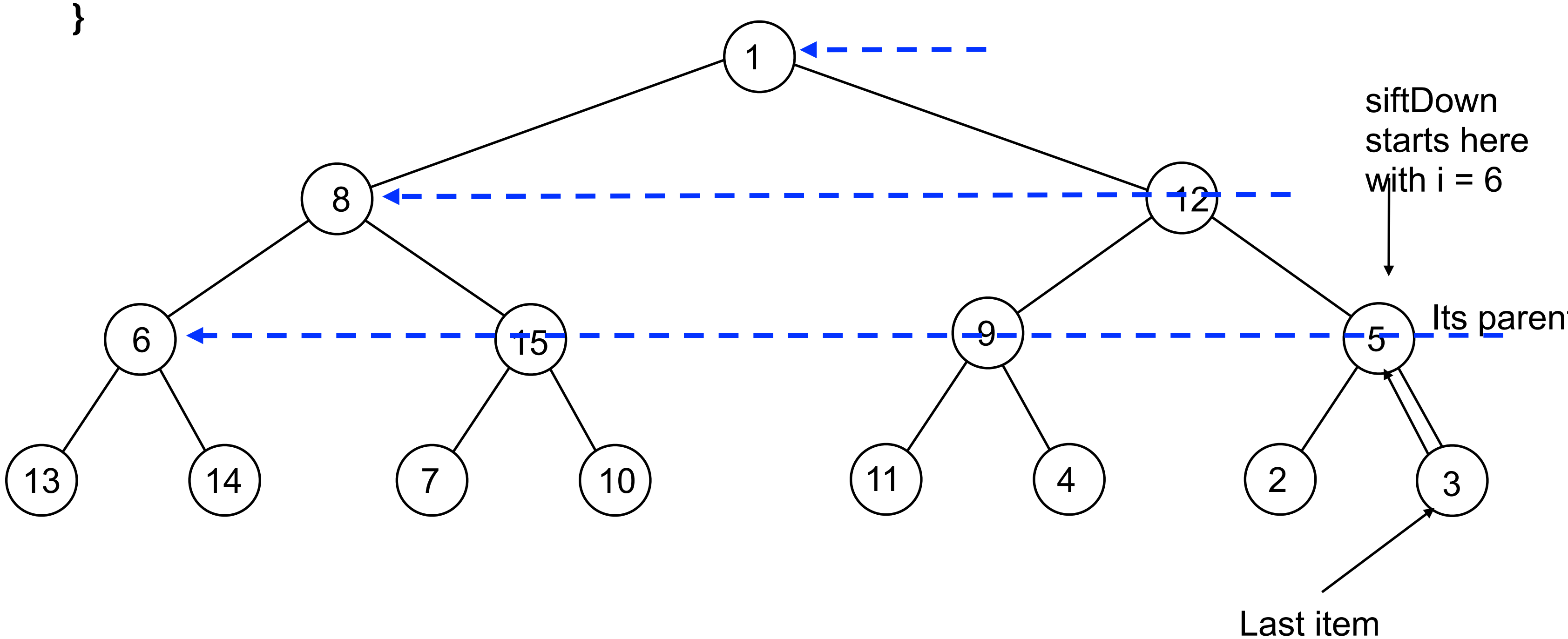
  - Running time: O(log(1) + log(2) + … + log(N)) = O(?)

An Efficient Algorithm: Build the Heap in Place

1. Position = numItems/2 -1; // initial position – the parent of the last item
2. siftDown item at that position.
3. Decrement position by one.
4. Repeat steps 2,3,4 until position is 0.

1	8	12	6	15	9	5	13	14	7	10	11	4	2	3

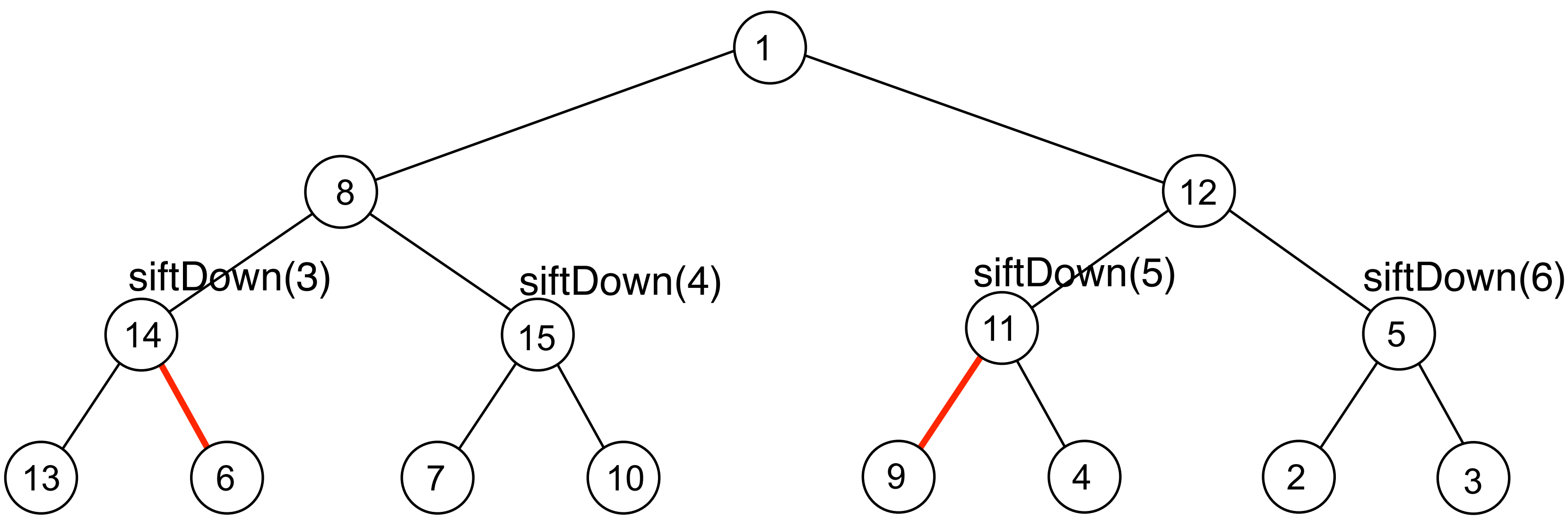
```
public void buildHeap( )
{
        for( int i = numItems /2 - 1; i >= 0; i-- )
                siftDown( i );
}
```



--

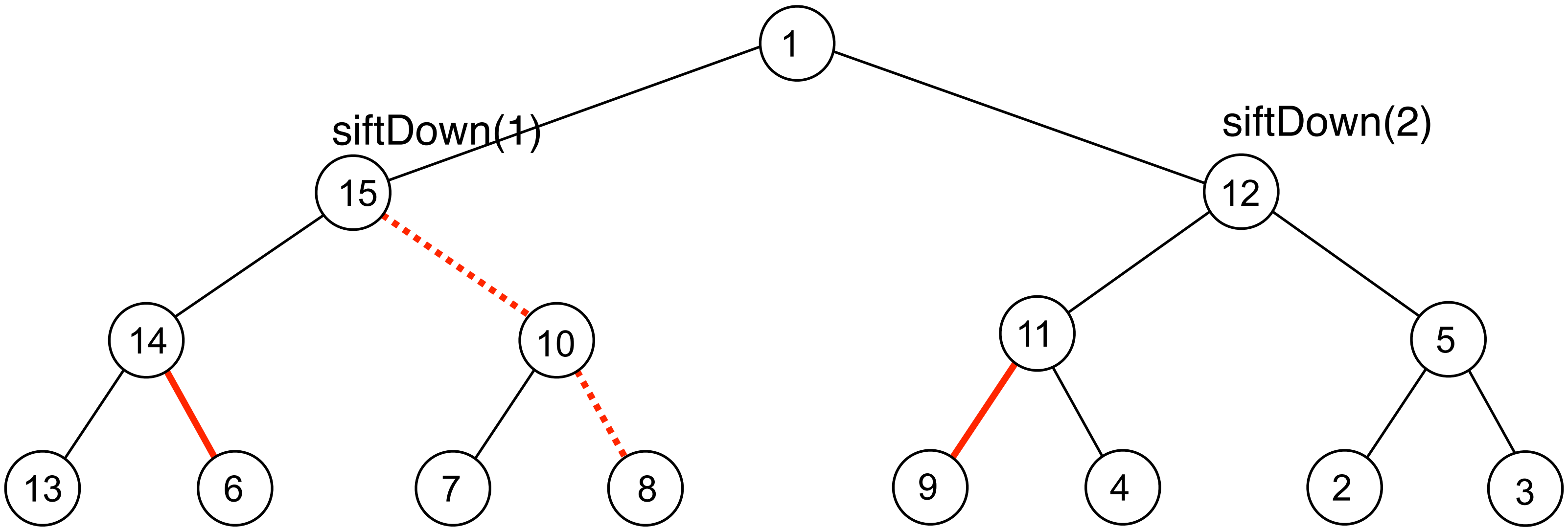
An Efficient Algorithm

```
for( int i = numItems / 2 - 1; i >= 0; i-- )
    siftDown( i );
```



An Efficient Algorithm

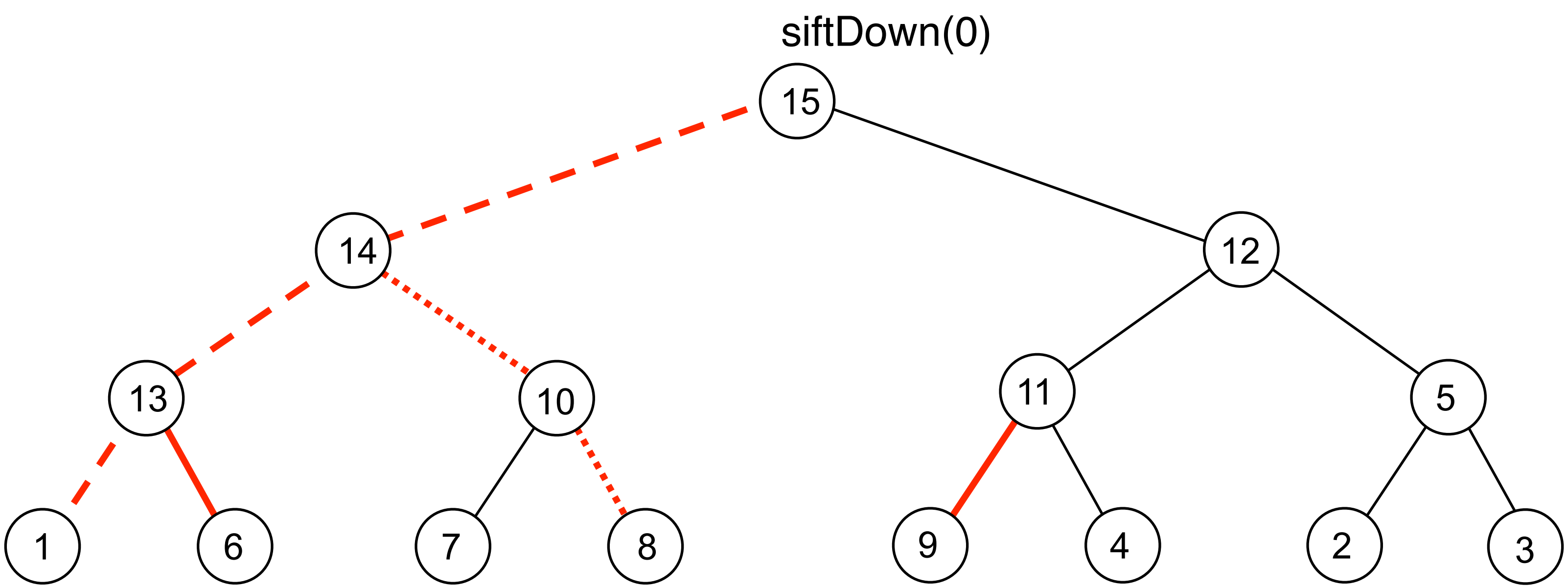
```
for( int i = numItems / 2 - 1; i >= 0; i-- )
    siftDown( i );
```



--

An Efficient Algorithm

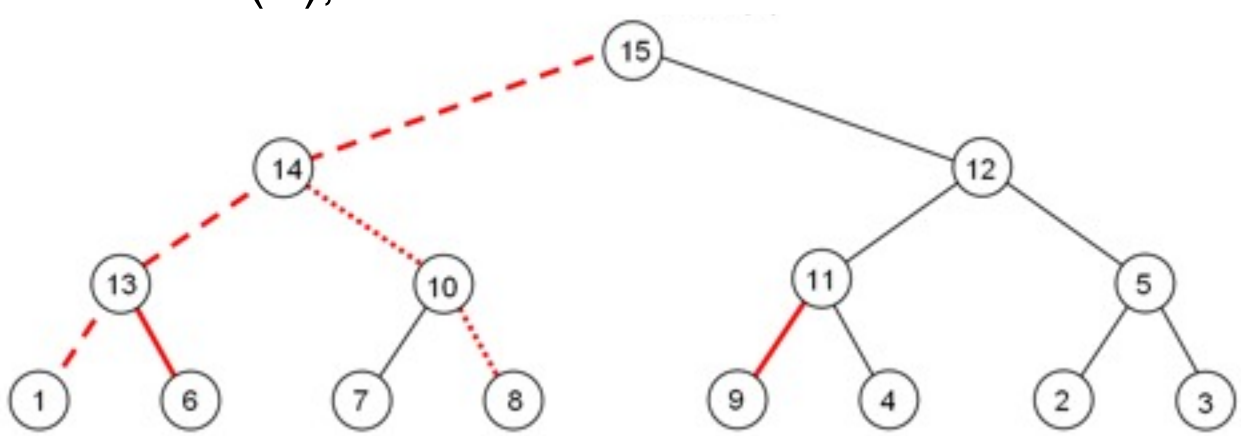
```
for( int i = numItems / 2 - 1; i >= 0; i-- )
    siftDown( i );
```



--

Running Time Analysis

- O(NlogN) as siftDown() is called N/2 times, and each takes no more than O(logN) time.
- However, this running time is not tight. The cost of BuildHeap is bounded by the number of red lines (all red lines), which is bounded by the sum of heights of all nodes of the heap.
  - The red lines may overlap (in this example they don't)
- This sum is O(*N*), where *N* is the number of nodes in the heap.



Running Time Analysis

- For a perfect binary tree of height h, $N = 2^{h+1}-1$:

  1 node at height h

  2 nodes at height h-1

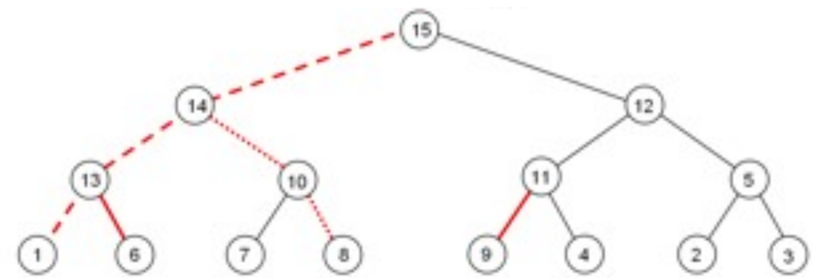
  4 nodes at height h-2

  …

  2^{h-1} nodes at height 1

  2^{h} nodes at height 0

  Total height



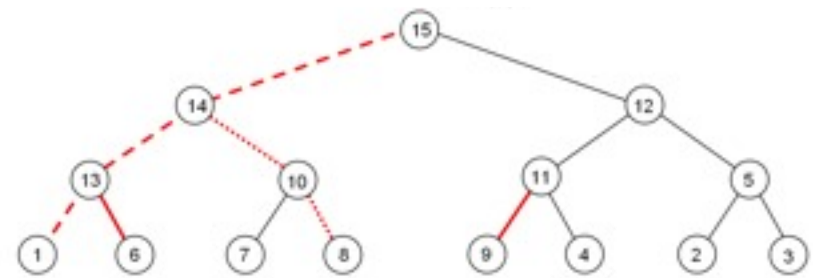
- Although a complete tree is not a full binary tree, but number of nodes in a complete tree of height h is:
  - $2^{h} <= N < 2^{h+1}$

- Thus the above sum is an *upper bound* on the sum of height of all nodes in a complete tree.

Running Time Analysis

- For a perfect binary tree of height h, N = 2^{h+1}-1:

  1 node at height h
  2 nodes at height h-1
  4 nodes at height h-2
  …
  2^{h-1} nodes at height 1
  2^h nodes at height 0



Total height $= h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^{h-1}(1) = S$

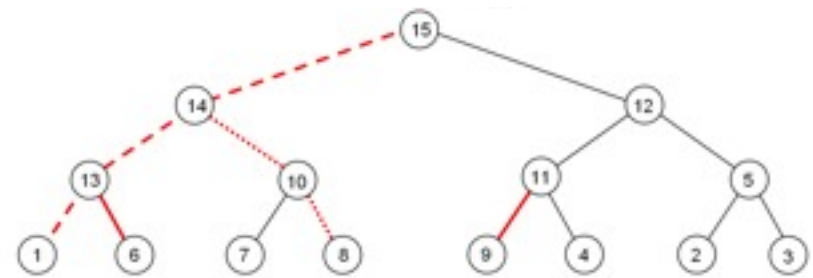
- Although a complete tree is not a full binary tree, but number of nodes in a complete tree of height h is:
  - $2^h <= N < 2^{h+1}$

- Thus the above sum is an *upper bound* on the sum of height of all nodes in a complete tree.

Running Time Analysis

- For a perfect binary tree of height h, $N = 2^{h+1}-1$:

1 node at height h
2 nodes at height h-1
4 nodes at height h-2
…
2^{h-1} nodes at height 1
2^h nodes at height 0



Total height $= h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^{h-1}(1) = S$

$$2h + 4(h-1) + 8(h-2) + 16(h-3) + \ldots + 2^{h}(1) = 2S$$

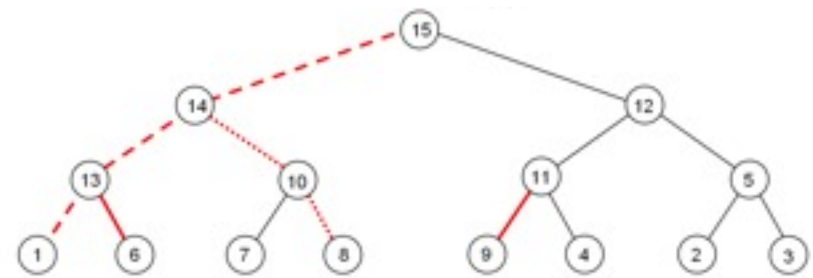
- Although a complete tree is not a full binary tree, but number of nodes in a complete tree of height h is:
  - $2^h <= N < 2^{h+1}$

- Thus the above sum is an *upper bound* on the sum of height of all nodes in a complete tree.

Running Time Analysis

- For a perfect binary tree of height h, $N = 2^{h+1}-1$:

1 node at height h
2 nodes at height h-1
4 nodes at height h-2
…
2^{h-1} nodes at height 1
2^h nodes at height 0



Total height $= h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^{h-1}(1) = S$

$$2h + 4(h-1) + 8(h-2) + 16(h-3) + \ldots + 2^{h}(1) = 2S$$

$$S = -h + 2 + 4 + \ldots + 2^{h} = 2^{h+1} - 1 - (h+1) = O(N)$$

- Although a complete tree is not a full binary tree, but number of nodes in a complete tree of height h is:
  - $2^h <= N < 2^{h+1}$

- Thus the above sum is an *upper bound* on the sum of height of all nodes in a complete tree.

Merge Two Heaps

Merge Two Heaps

- Merge two heaps into a single one.

Merge Two Heaps

- Merge two heaps into a single one.
- Append one heap to the end of the other, and then just like buildHeap, sift down the items (in the interior nodes)

Merge Two Heaps

- Merge two heaps into a single one.
- Append one heap to the end of the other, and then just like buildHeap, sift down the items (in the interior nodes)
  - Running time = O(?)

Running Time of Binary Heap Operations

- Summary of the worst-case running time of binary heap operations (max-at-top)

findMax	O(1)
removeMax()	O(logN)
insert()	O(logN)
delete()	O(logN)
update() the key	O(logN)
buildHeap()	O(N)
merge()	O(N)

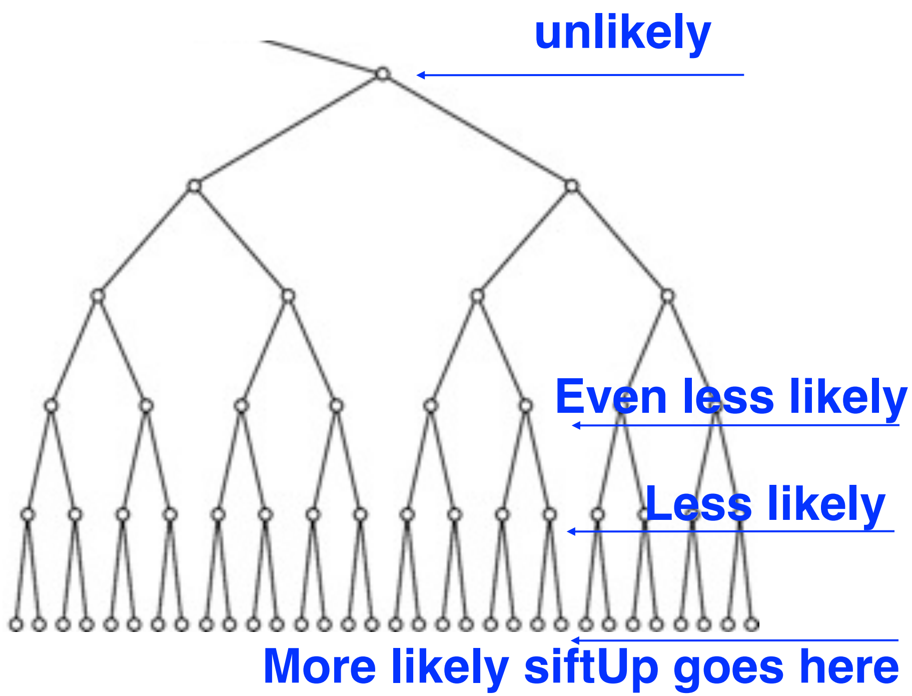
- Average-case running time is different for insertion. It takes O(1) average time (about 2.6 comparisons), why?

Running Time of Binary Heap Operations

- Summary of the worst-case running time of binary heap operations (max-at-top)

findMax	O(1)
removeMax()	O(logN)
insert()	O(logN)
delete()	O(logN)
update() the key	O(logN)
buildHeap()	O(N)
merge()	O(N)

- Average-case running time is different for insertion. It takes O(1) average time (about 2.6 comparisons), why?



Etc.

- Problem of the week:
  - Assume that you want FIFO order within a given priority. Do heaps provide any guarantees in this regard? Why or why not?