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?

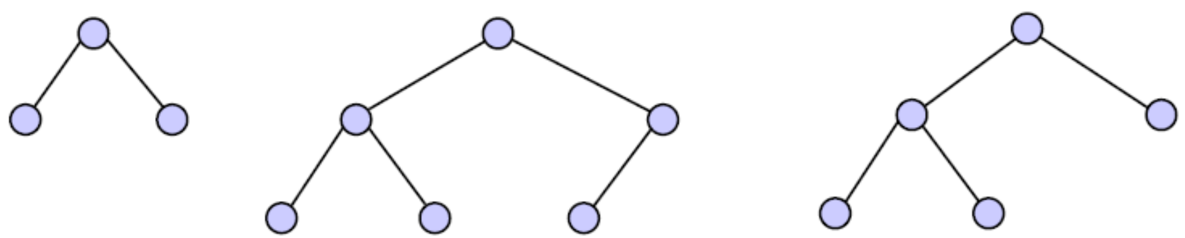
- Operations:
  - insert: add an item to the priority queue (with a priority value)
  - remove: remove a highest-priority item
    - an item in the queue with the largest associated priority value
  - …

- 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)*

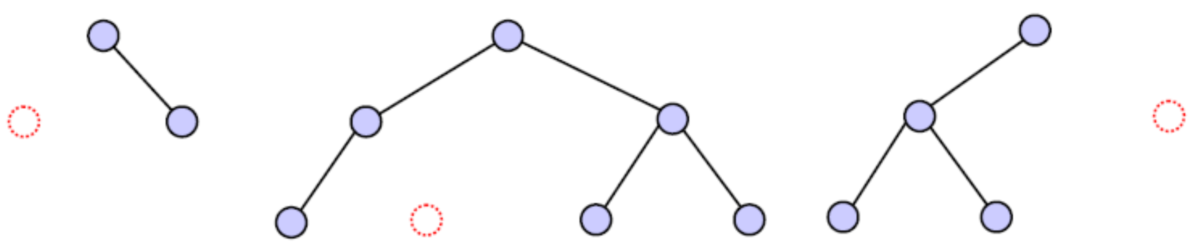
Structure Types of Binary Trees

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

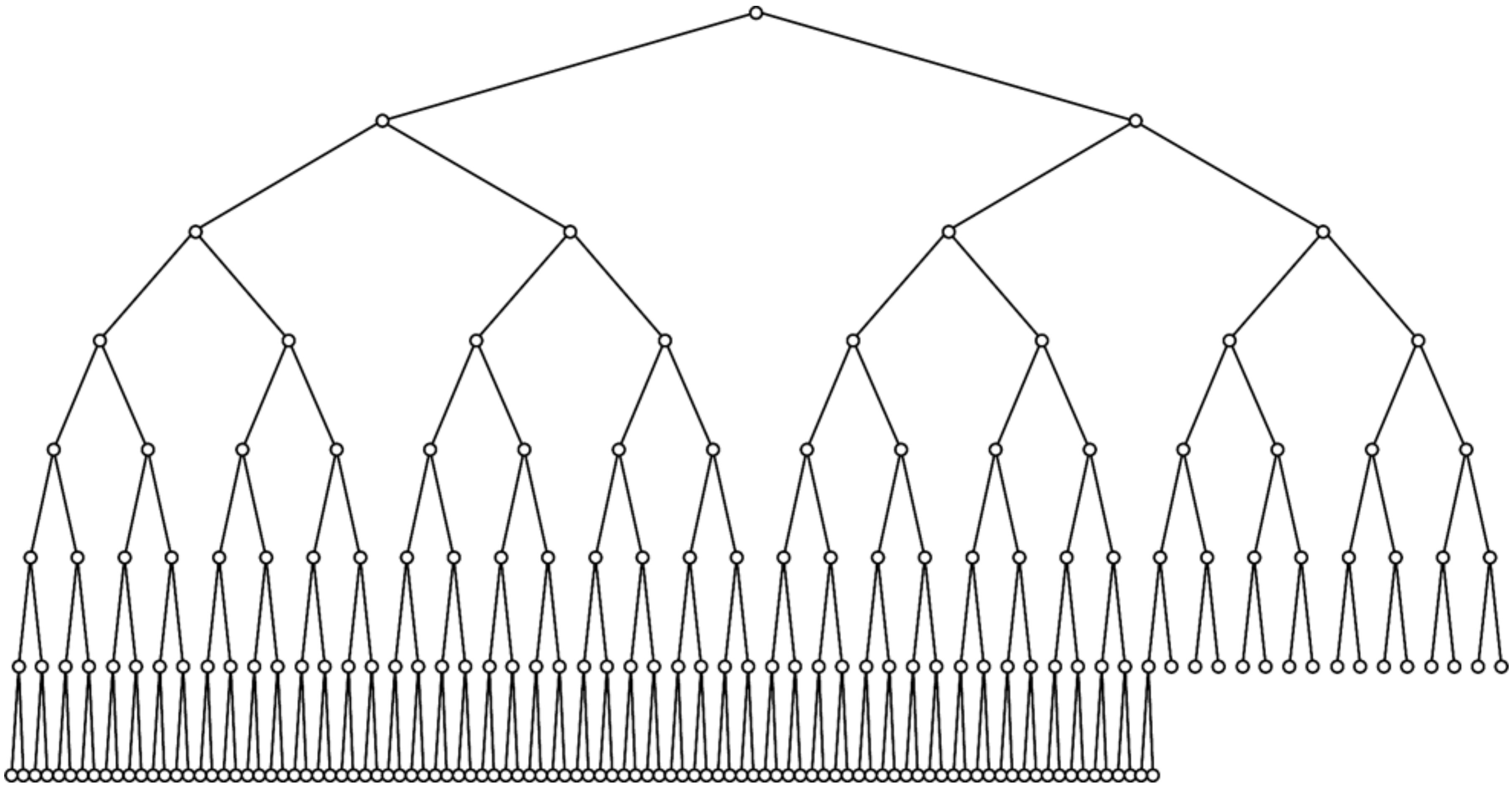
Complete:



Not complete (= missing node):

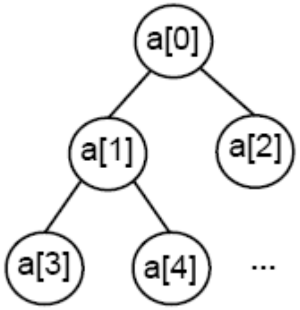


A Large Complete Binary Tree



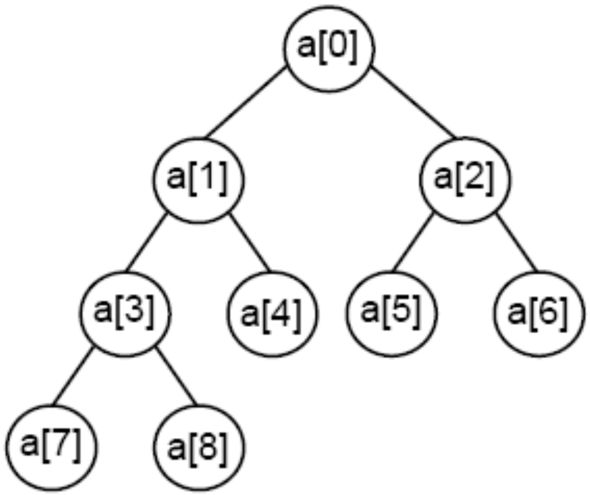
Array Representation of A Complete Binary Tree

- 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], for i!=0

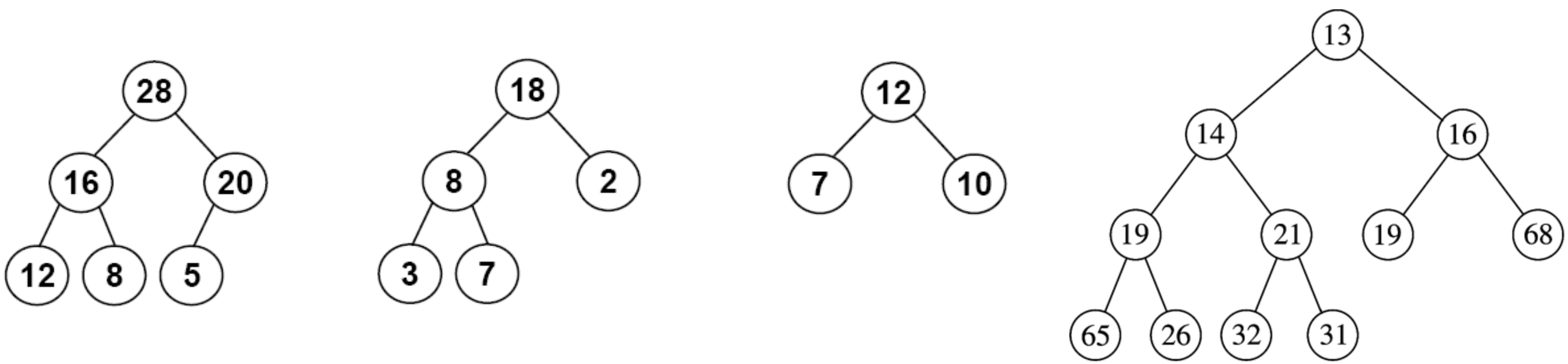


A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

- 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.

Heap ADT - Generics

- We want a heap that can store data items of different types (just like generic lists, stacks, and queues)
  - A “priority” can be a more general concept than a “search key”
  - We would like to derive “priority” from the object rather than have a distinct “key” attribute
- Writing a generic heap (as well as BST and AVL trees) 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, use compareTo() method.
  - 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 our own classes

    public int compareTo(*Classname* other)

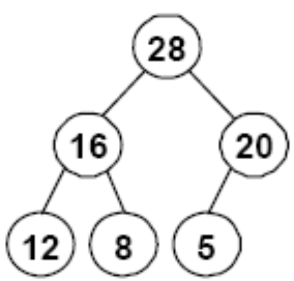
- Then, o1.compareTo(o2) should return:
  - a negative integer, e.g., -1, if o1 is less than o2
  - 0 if o1 equals o2
  - a positive integer, e.g, +1, if o1 is greater than o2

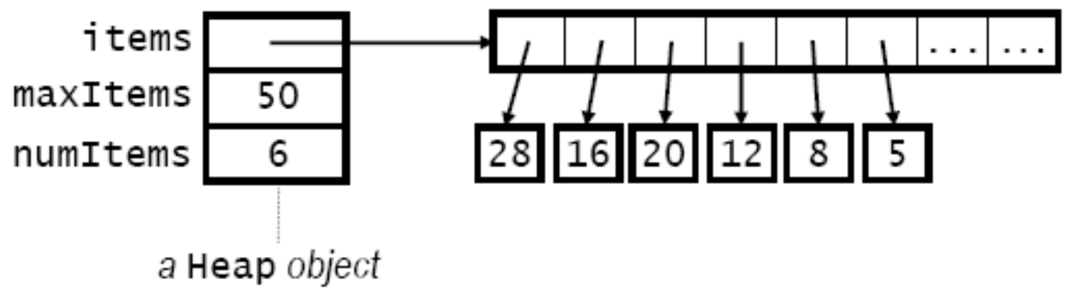
- We also should indicate that our class has this method by adding the following to the class header:

  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;
    }
    …
}
```

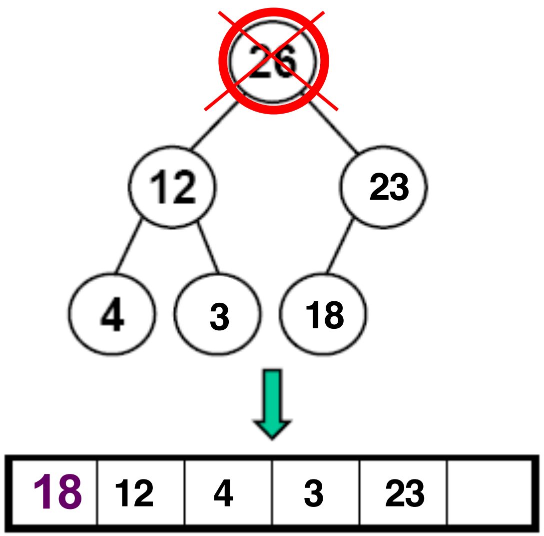




- Suitable for objects of *any* type T that has a compareTo() method and contains "implements Comparable<T>" in its class header. The array actually stores the references

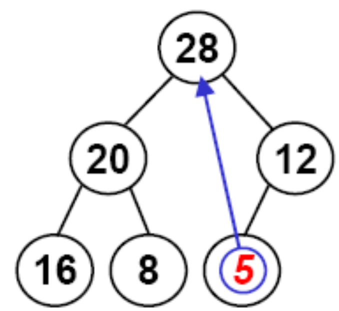
Removing An Item (the largest)

- 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

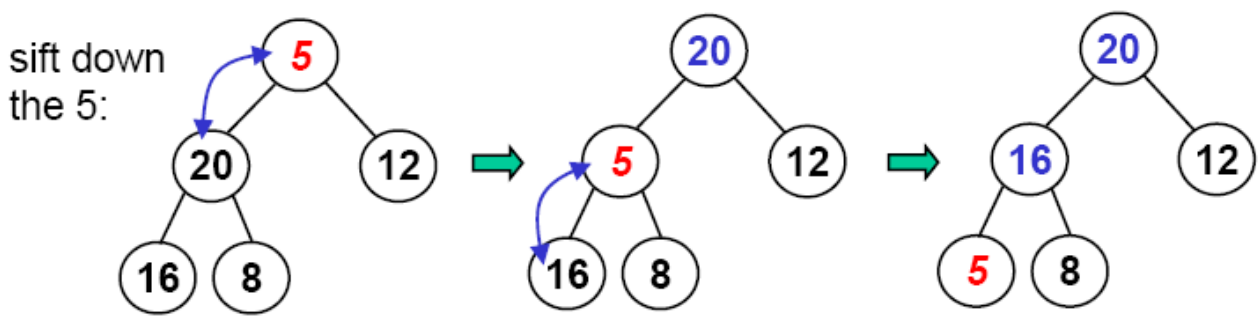


Removing An Item (the largest)

- 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
  - “sift down” the new root item until it is >= its children (or it’s a leaf)
  - return the largest item

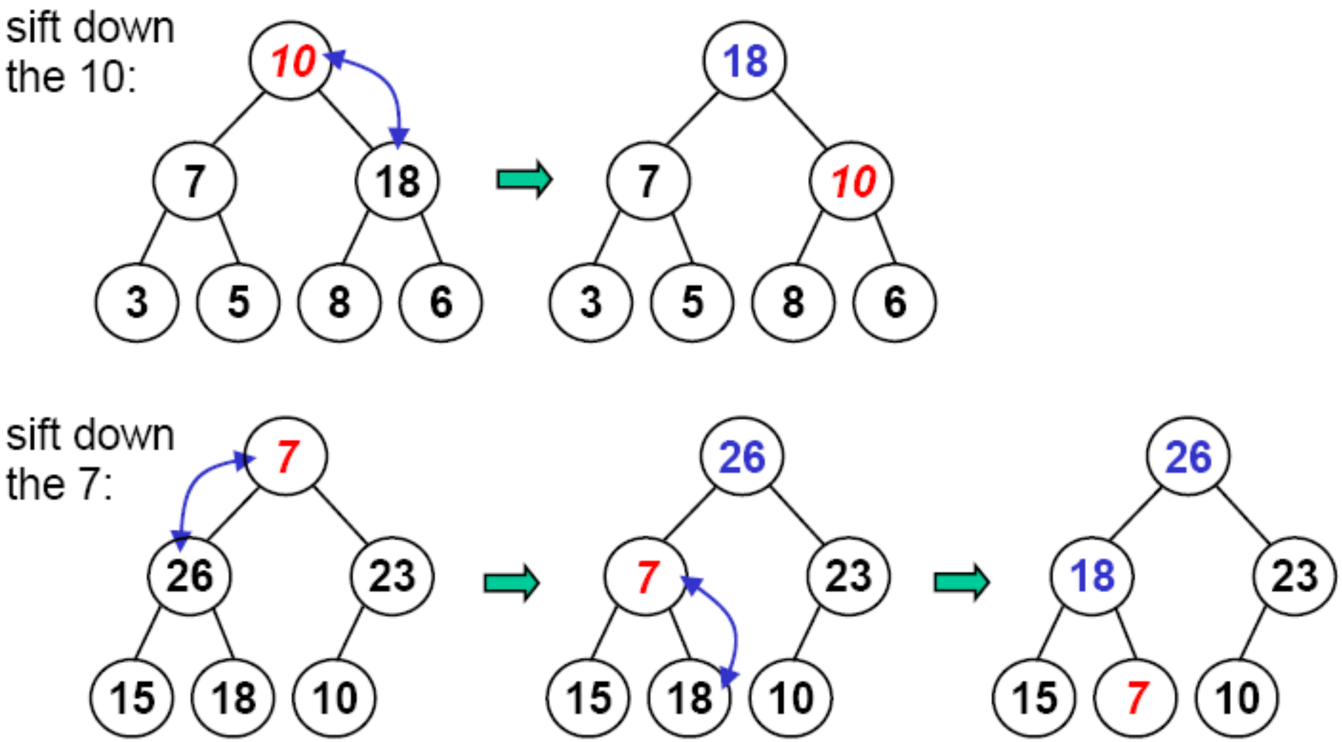


- “sift”: items are filtered such that small ones will fall



"Heapify" – Sifting Down

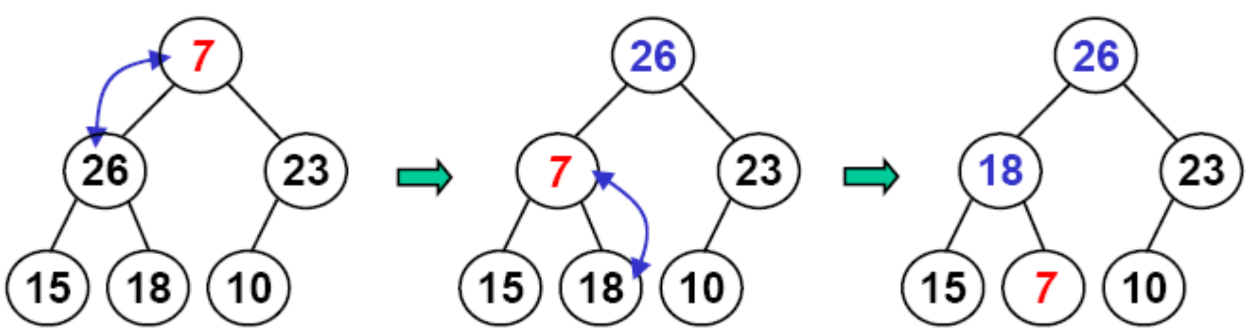
- To sift down item *x* (i.e., the item whose key is *x*):
  - compare *x* with the larger of the item's children, *y*
  - if $x < y$, swap *x* and *y* and repeat



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 instead.
                if (child +1 < numItems  && // if the right child exists
                        items[child].compareTo(items[child + 1]) < 0)  // … and is bigger than the left child
                                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.



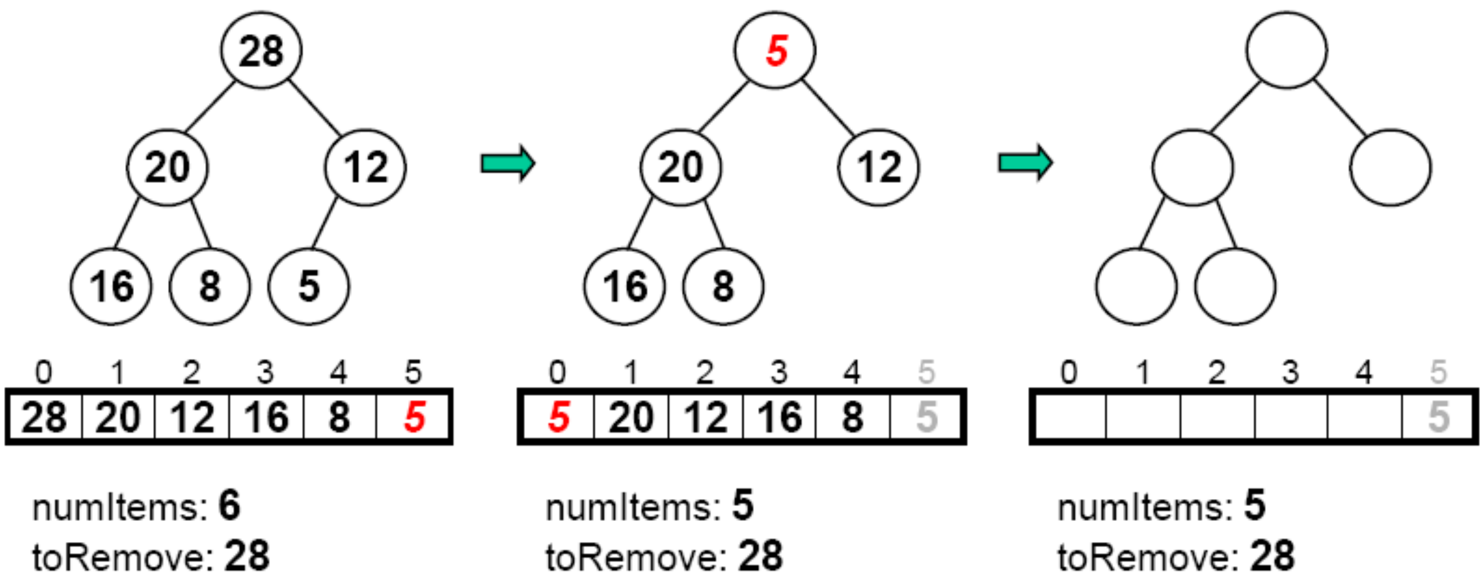
7	26	23	15	18	10

26	7	23	15	18	10

26	18	23	15	7	10

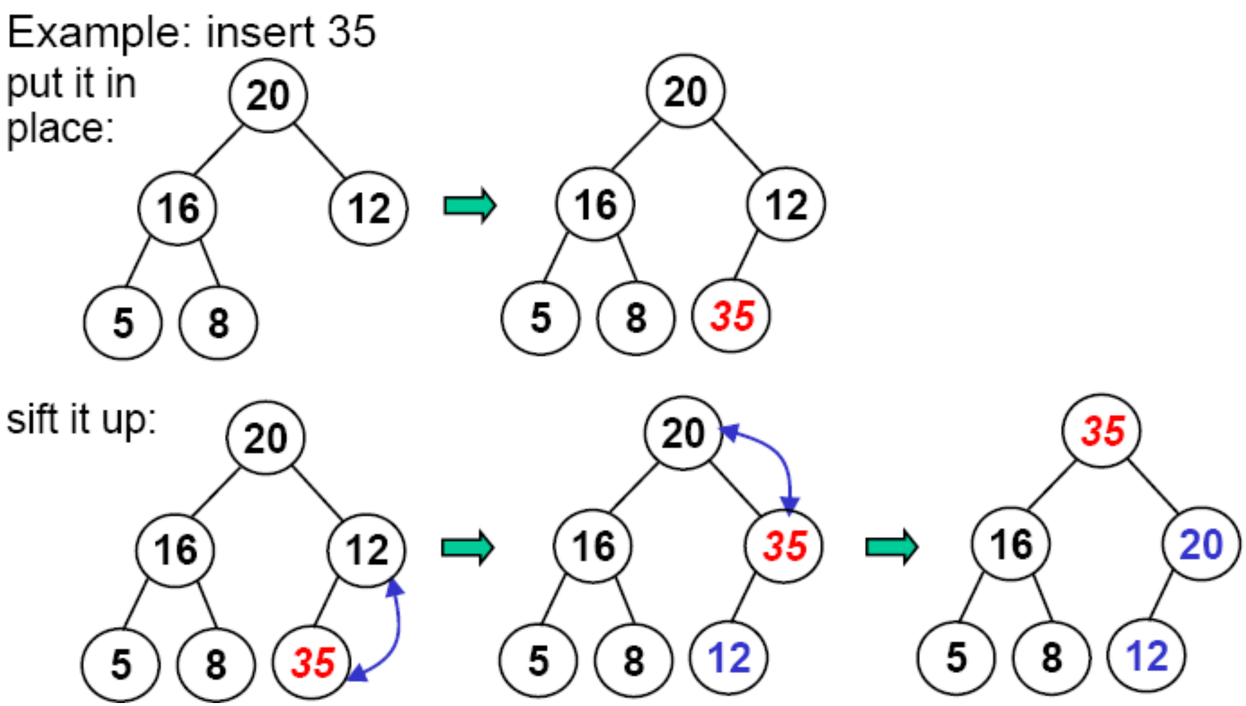
RemoveMax Method

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



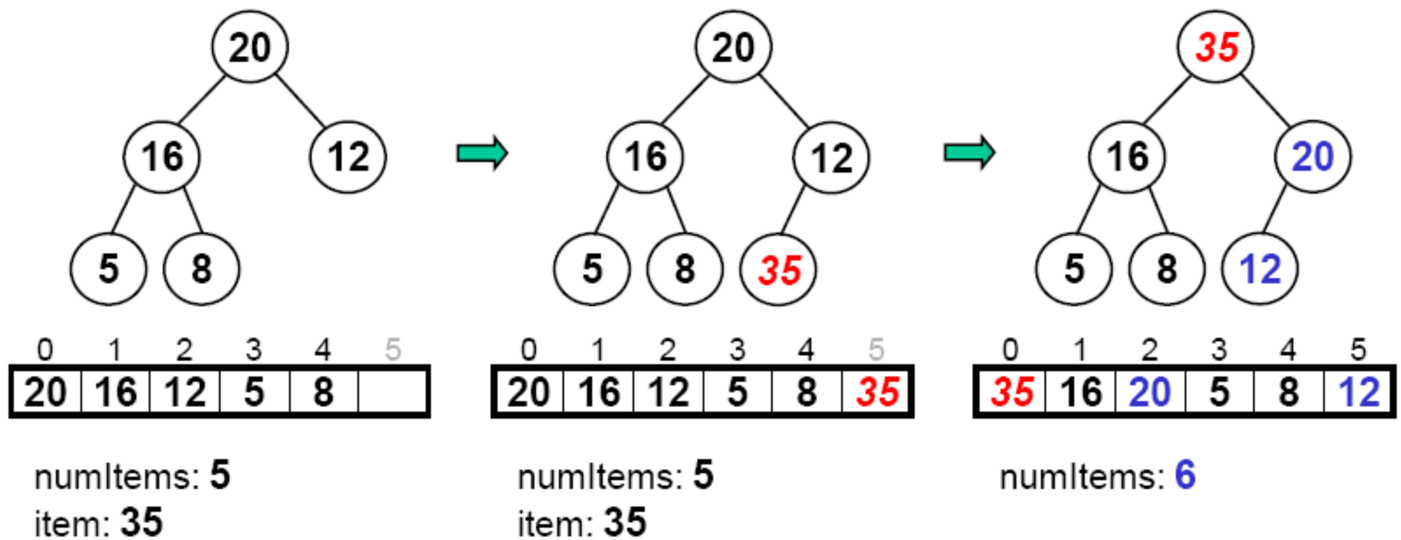
Inserting An Item

- 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);
}
```



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

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]);
}
```

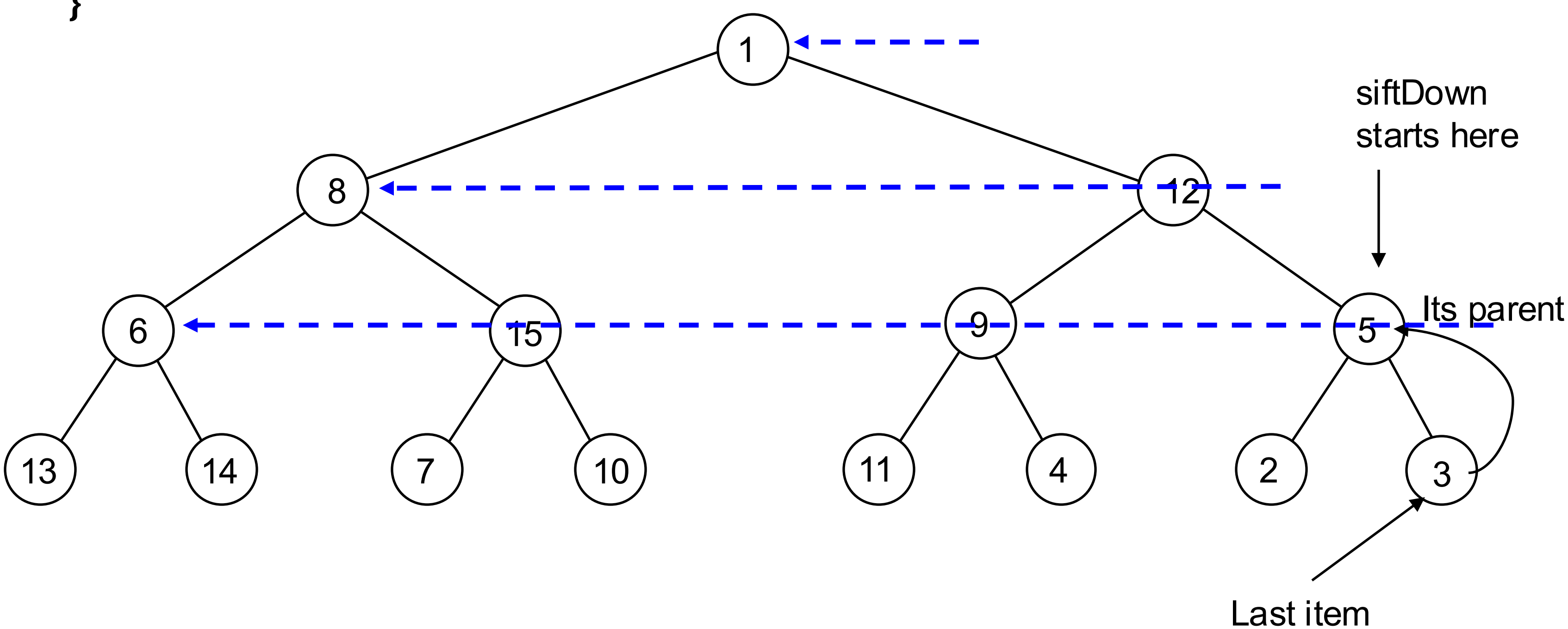
  - Running time: O(log(1) + log(2) + … + log(N)), or O(NlogN)

An Efficient Algorithm: Build the Heap in Place

1. Position = (numItems-2)/2; // 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.

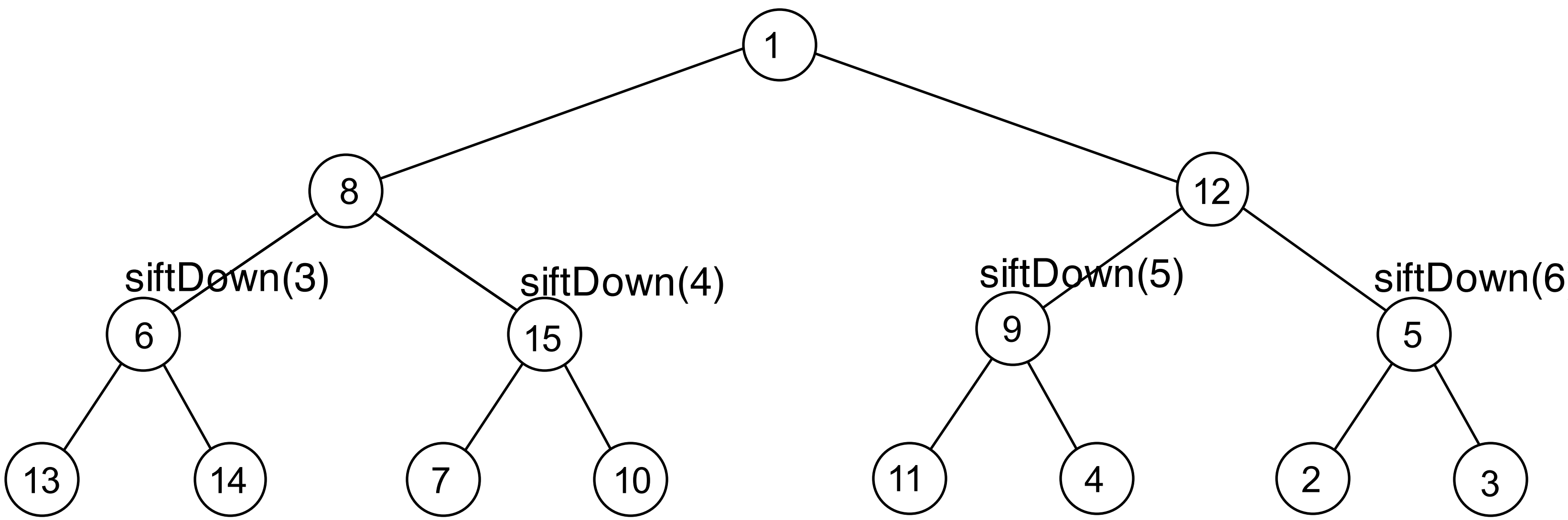
1	8	12	6	15	9	5	13	14	7	10	11	4	2	3

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



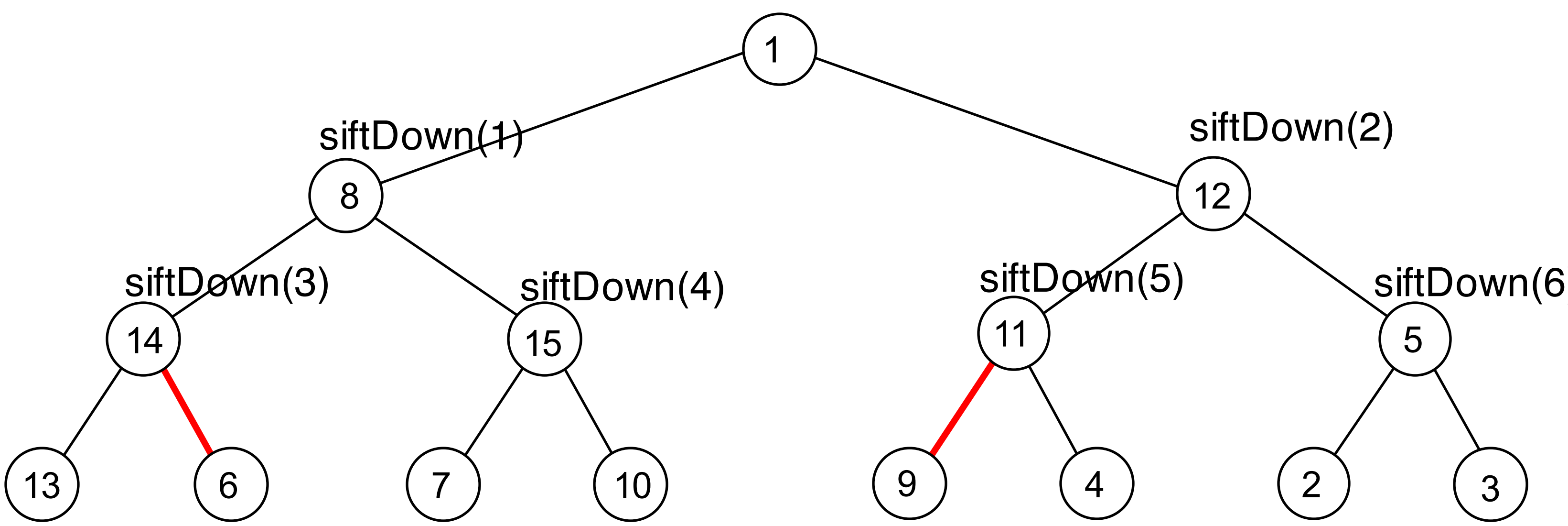
An Efficient Algorithm

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



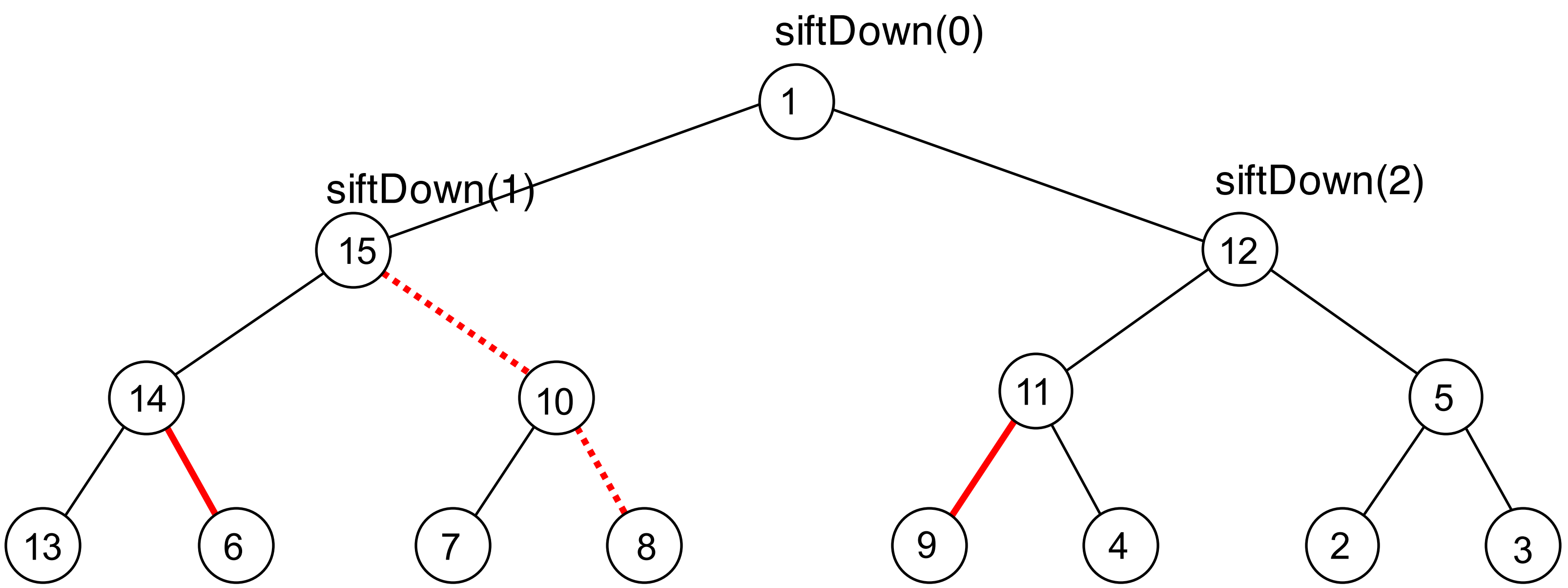
An Efficient Algorithm

```
for( int i = (numItems -2) / 2; 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 );
```

siftDown(0)