Lecture 18

# Heaps and HeapSort

**Last time:**

We looked at TreeMap and TreeSet of the Java Collections Framework. More specifically, our focus was when they can be useful.

TreeMap and TreeSet are useful to have all the elements in order and perform non-exact match searches.

Due to the fact that the elements are ordered, TreeMap and TreeSet provide many useful methods that can be found on their API doc.

Since they are implemented based on self-balanced tree (Red-Black tree), their major operations are guaranteed to have O(log n) worst-case running time complexity.

*Another important thing to remember is that all of the objects that are being added into TreeMap or TreeSet data structures must be sortable.*

In other words, you need to implement Comparable or Comparator interface for the object class that you intend to put into TreeMap or TreeSet.

Today, we will look into our last data structure, Heap!

In this lecture, we will look into Max-Heap.

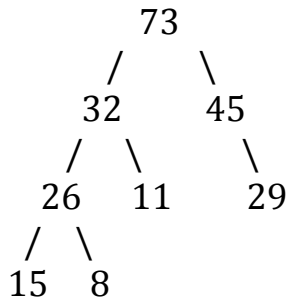
**Step 1: Conceptual View**

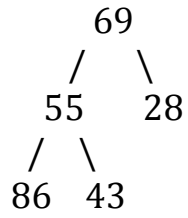
A max-heap is a binary tree with the following characteristics:

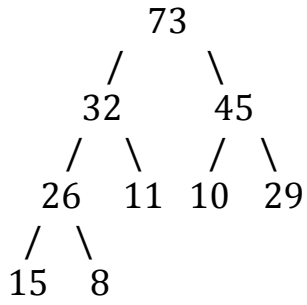
- It's complete (almost). Every level of the tree has the maximum number of nodes except possibly the last level. It's filled in reading from left to right across each row.
- The largest data is at the root of the tree.
- For every node in the max-heap, its children contain smaller keys.

**Examples**

Are these heaps?



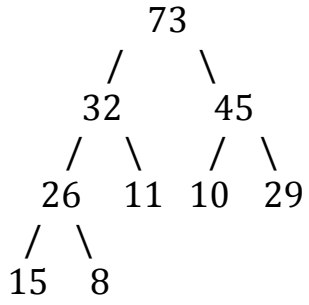



A few analyses

- A heap is **NOT** a sorted structure and can be considered as **PARTIALLY ordered**.
- What can we learn from the fact that a heap is a complete binary tree? It should always have the smallest possible _____ which is O(_____).
- It is a useful data structure when we need to keep getting the object with the highest priority.

**Add (insert) a new data into Max-Heap**




Given the max-heap above, insert 51.

Here is the algorithm (***Percolate up***)!

1. Insert a new data into the next available tree position so that the tree remains **complete**.
2. Swap the data with its parent(s), if necessary, and continue to do so until the tree is **restored to be a heap** (Heap-order property of the max-heap.).

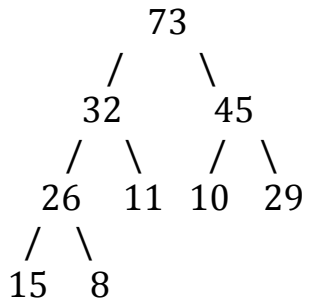
Knowing the insertion algorithm, how do we build a max-heap?

Practice: Build a max-heap with the given values.
26, 32, 45, 10, 29, 8, 11, 9, 73, 15

## Remove the maximum data from Max-Heap

Here is the algorithm (***Percolate down***)!

1. Remove root (the maximum key) and replace it with the last node of the lowest level to make sure the tree **stays complete**.
2. Swap the data with its larger child, if necessary, and continue to do so until the tree is **restored to be a heap** (Heap-order property of the max-heap).

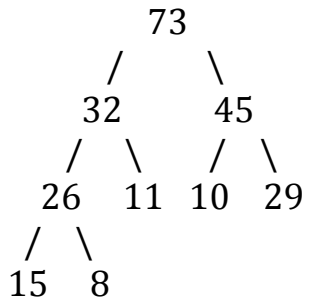



Given the max-heap above, remove the maximum data (root).

## Step 2: Implementation View

So far, we looked at max-heap in the form of a binary tree and its two major operations (insertion and remove max). But the heap in the form of a binary tree is only a conceptual representation.

How can it be implemented?




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 73 | 32 | 45 | 26 | 11 | 10 | 29 | 15 | 8 |

What do we see here?

- A heap is complete which means there cannot be any hole in the array. Every cell should be filled (from 0 to N-1).
- Maximum key is in the root. That means it is at the $0^{th}$ index of the array.
- Once again, a heap is not a sorted structure which means keys in the array are not sorted. (*partially ordered*.)

Major operations we want to implement in our max-heap are:

- Insert
- Remove Max

Let's have our interface first.

```
public interface MaxHeapInterface {
      /**
       * Insert a new key into a heap
       * @param key key to be inserted
       * @return boolean to check whether it was added or not
       */
      boolean insert(double key);

      /**
       * remove the highest priority key (maximum for max heap)
       * @return removed key
       */
      double removeMax();
}
```

Here is a skeleton of the max-heap class.

```
public class MaxHeap {
      private Node[] heapArray;
      private int currentSize;

      public Heap(int initialCapacity) {
            heapArray = new Node[initialCapacity];
            currentSize = 0;
      }

      public boolean insert(double key) {
           // TODO implement this method
      }

      public double removeMax() {
           // TODO implement this method
      }
}
```

Just like BST, we have a node class as a private static nested class.

```
// private static nested Node class
private static class Node {
      private double key;  // data
      public Node(double key) {
            this.key = key;
      }
}
```

**Inserting a new data into Max heap.**

Steps to insert a new data:

1. Insert a new data into the next available tree (array) position so that the heap remains complete.
2. Exchange data with its parent(s) (if necessary) until the tree is restored to be a heap (*Percolating up*!)

```
public boolean insert(double key) {
      if(currentSize == ____hA.length_____) return false;
      Node newNode = new Node(key);
      heapArray[____currentSize_____] = newNode;
      percolateUp(____currentSize++____);
      return true;
}

private void percolateUp(int index) {
      // save the bottom node, newly added one
      Node bottom = heapArray[index];

      // find the parent index
      int parent = ____(index-1)/2____;

      // parent's key is smaller than the new key
      while(____index>0_____ &&
heapArray[____parent_____].key < bottom.key) {
            heapArray[index] = heapArray[____parent_____];
            index = ____parent_____; // move up
            parent = ____(parent-1)/2____; // new parent
      }

      heapArray[index] = ____bottom_____; // insert into the
proper location
}
```

**Removing the max key from max heap.**

Three basic steps to remove:

1. Remove the root (Node at 0th index)
2. Move the last node up to the root (0th index)
3. Percolate or trickle down the node where it is below the bigger nodes and above the smaller nodes

```
public double removeMax() {
      if (___currentSize == 0_____________________)
            throw new NoSuchElementException("The heap is empty");
      Node root = heapArray[_____0_____];
      currentSize--;
      heapArray[0] = heapArray[currentSize]; // root<-last one
      heapArray[___currentSize__] = ___null____;
      percolateDown(_____0_____); // percolate down
      return root.key;
}
private void percolateDown(int index) {
      Node top = heapArray[_____index_____];
      int largerChild; // larger child's index
      // loop till the index that has a child
      while(index < ____currentSize/2_____) {
            int leftChild = ____2 * index +1_____;
            int rightChild = leftChild + 1;
            // find which one is larger child
            if(_____ &&
                        heapArray[leftChild].key <
heapArray[rightChild].key) largerChild = rightChild;
            else largerChild = leftChild;
            // if top is bigger, then stop
            if(top.key >= heapArray[_____].key)
                  break;
            // move the nodes up
            heapArray[index] = heapArray[_____];
            index = _____; // go down
      }
```

```
      // put top key into proper location to restore the heap
      heapArray[index] = _____________________;
}
```

**Efficiency of *insert(double key)* and *removeMax()* methods**
Where do we spend most of time for these operations?

One important thing you need to notice is that while percolating up or down, we did not really swapped as we mentioned in the Conceptual View.

As we saw before, one swap requires _____ copies. But, in our implementation, we have reduced the number of copies.

For example, we have four nodes and need to swap all of them.
A ←→ B ←→ C ←→ D

How many copies do we need to perform?

How about our implementation? Let's draw its operations!

How many copies do we need to perform?

This will reduce the running time, especially in case heap is big (tall). However, *the main point here is that the number of comparisons and necessary copies is bounded by the _____ of the heap.*

Of course, there are other moves but they are constant for the operations. Thus, we can ignore.

In conclusion, we can say that the two major operations' worst-case running time complexity, in Big O notation, is _____.

Now that we know those major operations of a heap have the worst-case running time complexity of _____, we might have this question:

***"What if we use this data structure for sorting?"***

This sorting algorithm is called, for an obvious reason, ***HeapSort***.

A very basic idea is:

- First, insert items into a heap using insert() method to build a heap.
- And, call removeMax() for max heap or removeMin() for min heap repeatedly so that we get the items in a sorted order.

```
for(int i = 0; i < array.length; i++)
    theHeap.insert(array[i]);
for(int i = 0; i < array.length; i++)
    array[i] = theHeap.removeMax(); // removeMin for min heap
```

Once again, insert() and removeMax() methods' worst-case running time complexity is _____.

Each must be applied **n** times which makes the entire sort run in O(_____) time.

**Priority Queue**
Priority Queues may be used for any application that requires processing items based on some priority, such as task scheduling in computers. The major operations in priority queues are insert(object) and delete().

The Java Collections Framework has PriorityQueue class that is a Heap implementation. (By default, min heap)

Its major operations and their worst-case running time complexities are:

- add(object) or offer(object) : inserting a new element _____
- poll() : removing the highest priority element _____
- peek() : looking at the highest priority element _____
- remove(object) : remove an element _____

The PriorityQueue class of the Java Collections Framework does not guarantee to traverse the elements in any particular order since it is based on heap.

**Comparison with BSTs**
Even though both of them are binary trees, they are conceptually different and their implementations are fundamentally different.

Which tree is for easier searching?

Which tree is for retrieving the maximum key quickly?

Which tree is guaranteed to be complete or almost complete?

Which tree is guaranteed to be balanced?

Worst-case running time complexity

| | Binary Search Tree | Max Heap (Min Heap) |
|---|---|---|
| Insertion | | |
| Deletion | | |
| Search | | |
| Peek Max | N/A | |