Lecture 18

Heaps and HeapSort

Last time:

We looked at TreeMap and TreeSet in the Java Collections Framework. They 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 pages.

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 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 class (type) that you intend to put into TreeMap or TreeSet.

Today, we will go over our last data structure, Heap, and one more sorting algorithm!

In this lecture, we will look at 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 (bottom) 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?

```
          73
        /     \
      32       45
     /  \        \
   26    11       29
  /  \
15    8
```

```
          69
         /  \
       55    28
      /  \
    86    43
```

```
          73
        /     \
      32       45
     /  \     /  \
   26    11  10   29
  /  \
15    8
```

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 _________________.
- 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

```
              73
            /     \
         32         45
        /  \       /  \
      26    11   10    29
     /  \
   15    8
```

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 max-heap).

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

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

Remove the maximum key from Max-Heap

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

1. Remove root (the maximum key) and replace it with the last node of the bottom level to make sure the tree **stays complete.**
2. Swap the newly promoted node 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).

```
            73
          /      \
        32         45
       /   \      /   \
     26    11  10    29
    /   \
  15     8
```

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

Step 2: Implementation View

So far, we have 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?

```
            73
         /       \
       32         45
     /   \      /   \
   26    11   10    29
  /   \
15     8
```

0	1	2	3	4	5	6	7	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 at 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 {
    /**
     * Inserts a new key into a heap.
     * @param key key to insert
     * @return boolean to check whether it was added or not
     */
    boolean insert(int key);

    /**
     * Removes the highest priority key (maximum for max heap).
     * @return removed key. Exception if empty.
     */
    int removeMax();
}
```

Here is a skeleton of the max-heap class.

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

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

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

    public int 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 int key;
    Node(int k) {
        key = k;
    }
}
```

Inserting a new data (key) 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. Swap node with its parent (if necessary) until the tree is restored to be a heap (*Percolating up*!)

```
public boolean insert(int key) {
    if (currentSize == ____________________) {
        return false;
    }
    heapArray[__________________________] = new Node(key);
    percolateUp(______________________________);
    __________________________________________;
    return true;
}

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

    // find the initial index value of parent
    int parent = __________________________;

    // while parent's key is smaller than the new key
    while (______________________________________ &&
heapArray[_____________________].key < bottom.key) {
        // parent node comes down
        heapArray[index] = heapArray[____________________];
        index = ___________________________; // index moves up
        parent = __________________________; // new parent index
    }

    // finally, insert newly added node into proper position
    heapArray[index] = ____________________;
}
```

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 int removeMax() {
    if (________________________________________________) {
        throw new NoSuchElementException("The heap is empty");
    }
    Node root = heapArray[_______________];
    currentSize--;
    heapArray[0] = heapArray[________________]; // root<-last one
    heapArray[______________________] = __________________;
    percolateDown(__________________); // percolate down
    return root.key;
}
private void percolateDown(int index) {
    Node top = heapArray[________________________];
    int largerChild; // larger child's index
    // while there is as at least left child
    while (___________________________________________________) {
        int leftChild = _______________________________________;
        int rightChild = ______________________________________;

        // find which one is larger child
        if (_________________________________________________ &&
                      heapArray[leftChild].key <
heapArray[rightChild].key) {
            largerChild = rightChild;
        } else {
            largerChild = leftChild;
        }
        // no need to go down any more
        if (heapArray[_______________________].key <= top.key) {
            break;
        }
```

```
        // move the nodes up
        heapArray[index] = heapArray[__________________________];
        // index goes down toward larger child
        index = ______________________;
    }

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

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

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

As we saw before in simple sorting discussion, 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 array is long (or heap is tall).

However, *the main point here is that the number of comparisons and necessary copies is bounded by the ______________________ of a 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 worst-case running time complexity of the two major operations, in Big O notation, is ____________________.

Now that we know those major operations of heaps 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 obvious reasons, ***HeapSort***.

A very basic idea is:

- First, build a heap using insert() method.
- 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, it is 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 (Min)	N/A	