Section 06: Heaps and Heaps of Heaps

1. Ternary Heaps

Consider the following sequence of numbers:

5, 20, 10, 6, 7, 3, 1, 2

(a) Insert these numbers into a min-heap where each node has up to *three* children, instead of two.

(So, instead of inserting into a binary heap, we're inserting into a ternary heap.)

Draw out the tree representation of your completed ternary heap.

(b) Draw out the array representation of the above tree. In your array representation, you should start at index 0 (not index 1).

(c) Given a node at index i, write a formula to find the index of the parent.

(d) Given a node at index i, write a formula to find the j-th child. Assume that $0 \leq j < 3$.

2. Heaps – More Basics

(a) Insert the following sequence of numbers into a *min heap*:

[10, 7, 15, 17, 12, 20, 6, 32]

(b) Now, insert the same values into a *max heap*.

(c) Now, insert the same values into a *min heap*, but use Floyd's `buildHeap` algorithm.

(d) Insert 1, 0, 1, 1, 0 into a *min heap*.

(e) Call `removeMin` three times on the min heap stored as the following array: $[1, 5, 10, 6, 7, 13, 12, 8, 15, 9]$

3. Food For Thought: More Heaps

3.1. Running Times

Let's think about the best and worst case for inserting into heaps.

You have elements of priority $1, 2, \ldots, n$. You're going to insert the elements into a min heap one at a time (by calling `insert` not `buildHeap`) in an order that you can control.

(a) Give an insertion order where the total running time of all insertions is $\Theta(n)$. Briefly justify why the total time is $\Theta(n)$.

(b) Give an insertion order where the total running time of all insertions is $\Theta(n \log n)$.

3.2. Sorting and Reversing (with Heaps)

(a) Suppose you have an array representation of a heap. Must the array be sorted?

(b) Suppose you have a sorted array (in increasing order). Must it be the array representation of a valid min-heap?

(c) You have an array representation of a min-heap. If you reverse the array, does it become an array representation of a max-heap?

(d) Describe the most efficient algorithm you can think of to convert the array representation of a min-heap into a max-heap. What is its running time?

4. HW5 Prep: Delete

You just finished implementing your heap of `int`s when your boss tells you to add a new method called `delete`.

```java
public class DankHeap {
    // NOTE: Data starts at index 0!
    private int[] heapArray;
    private int heapSize;

    // Other heap methods here....

    /**
     * Removes the element k from the heap, and restores the heap
     * property. If no element equal to k exists in the heap, does
     * nothing.
     * @param int k, the element to remove.
     */
    public void delete(int k) {
        // TODO!
    }

    /**
     * You can assume this method correctly percolates the element at
     * the given index of heapArray up/down (if needed) until the heap
     * property is satisfied. You do *not* need to implement this!
     * @param int index, index of the heap member to percolate.
     */
    private void percolate(int index) {
        ...
    }
}
```

(a) How efficient do you think you can make this method?

(b) Write code for `delete`. Remember that `heapArray` starts at index 0! **Hint:** You can use the `percolate` method defined in the `DankHeap` class above.

5. Food For Thought: Recurrences and Heaps

Suppose we have a min heap implemented as a tree, based on the following classes:

```
class HeapNode {
    HeapNode left;
    HeapNode right;
    int priority;

    // constructors and methods omitted.
}

class Heap {
    HeapNode root;
    int size;

    // constructors and methods omitted.
}
```

You just finished implementing your min heap and want to test it, so you write the following code to test whether the heap property is satisfied.

```
boolean verify(Heap h) {
    return verifyHelper(h.root);
}

boolean verifyHelper(HeapNode curr) {
    if (curr == null)
        return true;
    if (curr.left != null && curr.priority > curr.left.priority)
        return false;
    if (curr.right != null && curr.priority > curr.right.priority)
        return false;
    return verifyHelper(curr.left) && verifyHelper(curr.right);
}
```

In this problem, we will use a recurrence to analyze the worst-case running time of `verify`.

(a) Write a recurrence to describe the worst-case running time of the function above. **Hint:** our recurrences need an input integer, use the height of the subtree rooted at `curr`.

(b) Find an expression (using summations but no recursion) to describe the running time using the tree method. Leave the overall height of the tree h as a variable in your expression.

(c) Simplify to a closed form.

(d) If a complete tree has height h, how many nodes could it have? Use this to determine a formula for the height of a complete tree on n nodes.

(e) Use the formula from the last part to find the big-$\mathcal{O}$ of the `verify`.

6. Debugging Heaps of Problems

For this problem, we will consider a hypothetical hash table that uses linear probing and implements the `IDictionary` interface. Specifically, we will focus on analyzing and testing one potential implementation of the `remove` method.

(a) Come up with at least 4 different test cases to test this `remove(...)` method. For each test case, describe what the expected outcome is (assuming the method is implemented correctly).

Try and construct test cases that check if the `remove(...)` method is correctly using the key's hash code. (You may assume that you can construct custom key objects that let you customize the behavior of the `equals(...)` and `hashCode()` method.)

(b) Now, consider the following (buggy) implementation of the `remove(...)` method. List all the bugs you can find.

```java
public class LinearProbingDictionary<K, V> implements IDictionary<K, V> {
    // Field invariants:
    //
    // 1. Empty, unused slots are null
    // 2. Slots that are actually being used contain an instance of a Pair object

    private Pair<K, V>[] array;

    // ...snip...

    public V remove(K key) {
        int index = key.hashCode();

        while ((this.array[index] != null) && !this.array[index].key.equals(key)) {
            index = (index + 1) % this.array.length;
        }

        if (this.array[index] == null) {
            throw new NoSuchKeyException();
        }
        V returnValue = this.array[index].value;
        this.array[index] = null;
        return returnValue;
    }
}
```

(c) Briefly describe how you would fix these bug(s).