Section 05: Advanced AVL, Heaps

1. Constructing AVL trees

Draw an AVL Tree as each of the following keys are added in the order given. Show intermediate steps.

(a)

{“penguin”, “stork”, “cat”, “fowl”, “moth”, “badger”, “otter”, “shrew”, “lion”, “raven”, “bat”}

(b)

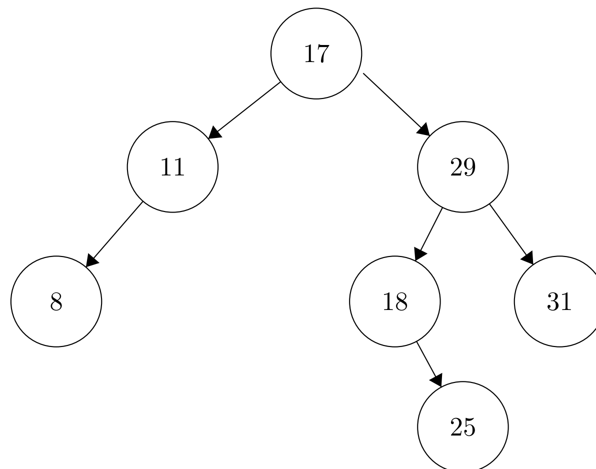
$$\{6, 43, 7, 42, 59, 63, 11, 21, 56, 54, 27, 20, 36\}$$

(c)

{“indigo”, “fuscia”, “pink”, “goldenrod”, “violet”, “khaki”, “red”, “orange”, “maroon”, “crimson”, “green”, “mauve”}

2. Inserting keys

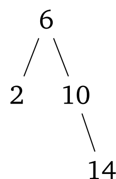
Consider the following AVL tree:



We now add the keys $\{21, 14, 20, 19\}$ (in that order). Show where these keys are added to the AVL tree. Show your intermediate steps.

3. AVL tree rotations

Consider this AVL tree:



Give an example of a value you could insert to cause:

(a) A single rotation

(b) A double rotation

(c) No rotation

4. True or false?

(a) An insertion in an AVL tree with n nodes requires $\Theta(\log(n))$ rotations.

(b) A set of numbers are inserted into an empty BST in sorted order and inserted into an empty AVL tree in random order. Listing all elements in sorted order from the BST is $\mathcal{O}(n)$, while listing them in sorted order from the AVL tree is $\mathcal{O}(\log(n))$.

(c) If items are inserted into an empty BST in sorted order, then the BST's `get()` is just as asymptotically efficient as an AVL tree whose elements were inserted in unsorted order.

(d) An AVL tree will always do a maximum of two rotations in an insert.

5. Big-$\mathcal{O}$

Write down a tight big-$\mathcal{O}$ for each of the following. Unless otherwise noted, give a bound in the worst case.

(a) Insert and find in a BST.

(b) Insert and find in an AVL tree.

(c) Finding the minimum value in an AVL tree containing n elements.

(d) Finding the k-th largest item in an AVL tree containing n elements.

(e) Listing elements of an AVL tree in sorted order

6. Analyzing dictionaries

(a) What are the constraints on the data types you can store in an AVL tree?

(b) When is using an AVL tree preferred over a hash table?

(c) When is using a BST preferred over an AVL tree?

(d) Consider an AVL tree with n nodes and a height of h. Now, consider a single call to `get(...)`. What's the maximum possible number of nodes `get(...)` ends up visiting? The minimum possible?

(e) **Challenge Problem:** Consider an AVL tree with n nodes and a height of h. Now, consider a single call to `insert(...)`. What's the maximum possible of nodes `insert(...)` ends up visiting? The minimum possible? Don't count the new node you create or the nodes visited during rotation(s).

7. 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 $0 \leq j < 3$.

8. 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` on the min heap stored as the following array: $[2, 5, 7, 8, 10, 9]$

9. 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?

10. Project Prep: Contains

You just finished implementing your heap of `ints` when your boss tells you to add a new method called `contains`. Your solution should not, in general, examine every element in the heap(do it recursively!)

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

    // Other heap methods here....

    /**
     * examine whether element k exists in the heap
     * @param int k, the element to find.
     * @return true if found, false otherwise
     */
    public boolean contains(int k) {
        // TODO!
    }
}
```

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

(b) Write code for `contains`. Remember that `heapArray` starts at index 0!

11. 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 $\Theta(n \log n)$.

12. Design Decisions

Finally! You've garnered a coveted interview for an internship at Kelp. Determine which data structure(s) are best suited for the scenarios below in terms of **typical** performance, taking into account the specific types of inputs listed in each problem. Your descriptions of the data structure(s) chosen and your algorithm should be brief but sufficiently detailed so that the runtime is unambiguous. Give a simplified, tight Θ bound for the **worstcase** runtime of your solution.

(a) Someone has loaded all the reviews stored on Kelp into a text document of n words. Print out the number of occurrences of each unique word. You **do** need to count the time required to build your data structure, but don't worry about resizing for any of the data structures you pick

(b) Answer the first question, but assume now that we care more about optimizing for memory usage than runtime.

(c) Answer the first question, but assume now that we also want the ability to print the unique words in alphabetical order.

(d) Kelp has a collection of n reviews. Each review has a unique date, author name, number of stars and the text contents of the review. We want to query the number of reviews within a certain datetime range. You don't have to count the time required to construct your data structure this time.

(e) Kelp is now trying to get into the field of visual computing. Your boss gives you n images of size 256x256, each represented as an `int[][]` array, that you'll need to store in a collection. Each image has associated with it a saturation value, which can be calculated from the pixels. Support the following three operations: `add(int[][] img)`, `getAllImgWithSaturation(int saturation)`, and `remove(int[][] img)`. When providing worst-case runtimes for each operation, don't worry about resizing for any of the data structures you pick, or the time required to construct those data structures.

13. Challenge: 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`.

14. Challenge: 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.

```
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).

15. Challenge: Design

Imagine a database containing information about all trains leaving the Washington Union station on Monday. Each train is assigned a departure time, a destination, and a unique 8-digit train ID number.

What data structures you would use to solve each of the following scenarios. Depending on scenario, you may need to either (a) use multiple data structures or (b) modify the implementation of some data structure.

Justify your choice.

(a) Suppose the schedule contains 200 trains with 52 destinations. You want to easily list out the trains by destination.

(b) In the question above, trains were listed by destination. Now, trains with the same destination should further be sorted by departure time.

(c) A train station wants to create a digital kiosk. The kiosk should be able to efficiently and frequently complete look-ups by train ID number so visitors can purchase tickets or track the location of a train. The kiosk should also be able to list out all the train IDs in ascending order, for visitors who do not know their train ID.

Note that the database of trains is not updated often, so the removal and additions of new trains happen infrequently (aside from when first populating your chosen DS with trains).