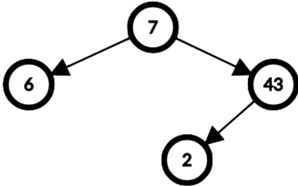
Section 05: AVL Trees + Heaps

AVL Tree Problems

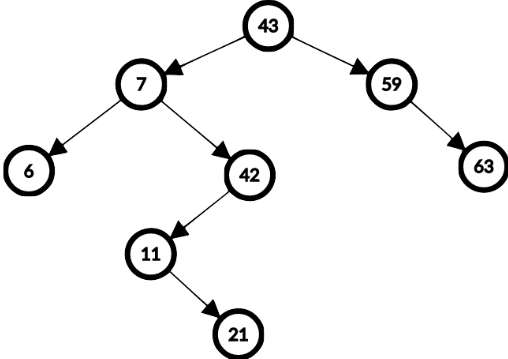
1. Valid BSTs and AVL Trees

For each of the following trees, state whether the tree is (i) a valid BST and (ii) a valid AVL tree. Justify your answer.

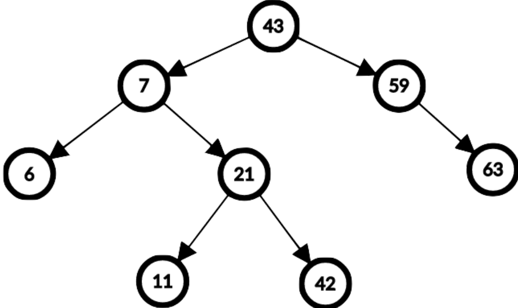
(a)



(b)



(c)



2. 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)

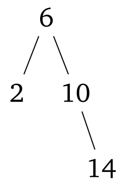
$$\{6, 43, 7, 42, 59, 63, 11, 21, 56, 54, 27, 20, 36\}$$

(c)

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

3. AVL tree rotations

Consider this AVL tree:



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

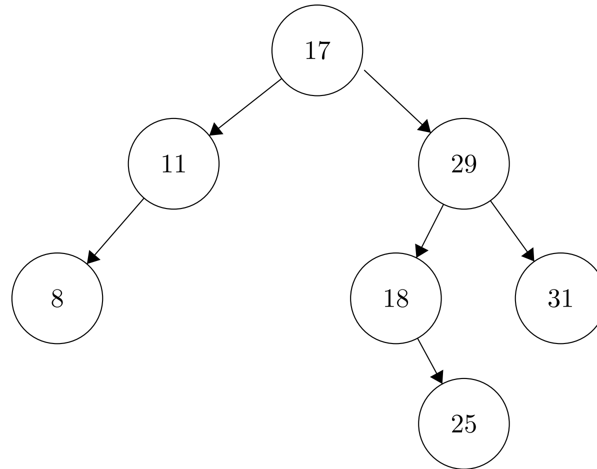
(a) A single rotation

(b) A double rotation

(c) No rotation

4. Inserting keys and computing statistics

In this problem, we will see how to compute certain statistics of the data, namely, the minimum and the median of a collection of integers stored in an AVL tree. Before we get to that let us recall insertion of keys in an AVL tree. Consider the following AVL tree:



(a) 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.

(b) Recall that if we use an unsorted array to store n integers, it will take us $O(n)$ runtime in order to compute the minimum element in the array. This can be done by running a loop that scans the array from the first index to the last index, which keeps track of the minimum element that it has seen so far. Now we will see how to compute the minimum element of a set of integers stored in an AVL tree which runs *much* faster than the procedure described above.

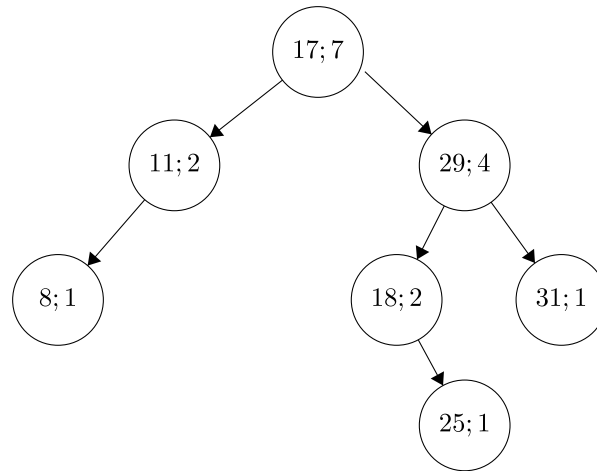
   i. Given an AVL tree storing numbers, like the one above, describe a procedure that will return the minimum element stored in the tree.

   ii. Supposing an AVL tree has n elements, what is the runtime of the above procedure in terms of n? How does this runtime compare with the $O(n)$ runtime of the linear scan of the array?

(c) In the next few problems, we will see how to compute the median element of the set of elements stored in the AVL tree. The median of a set of n numbers is the element that appears in the $\lceil n/2 \rceil$-th position, when this set is written in sorted order. When n is even, $\lceil n/2 \rceil = n/2$ and when n is odd, $\lceil n/2 \rceil = (n+1)/2$. For example, if the set is $\{3, 2, 1, 4, 6\}$ then the set in sorted order is $\{1, 2, 3, 4, 6\}$, and the median is 3.

   If we were to simply store n integers in an array, one way to compute the median element would be to first sort the array and then look up the element at the $\lceil n/2 \rceil$-th position in the sorted array. This procedure has a runtime of $O(n \log n)$, even when we use a clever sorting algorithm like Mergesort. We will now see how to compute the median, when the data is stored in a rather modified AVL tree *much* faster.

   For the time being, assume that we have a modified version of the AVL tree that lets us maintain, not just the key but also the number of elements that occur below the node at which the key is stored plus one (for that node). The use of this will become apparent very soon. As an example, the modified version of the AVL tree above, would like so (the number after the semi-colon in each node accounts for the number of nodes below that node plus one).



i. We now again add the keys $\{21, 14, 20, 19\}$ (in that order) to the modified AVL tree. How does the modified AVL tree look after the insertions are done?

ii. Given a modified AVL tree, like the one above, describe a procedure that will return the median element stored in the tree. Note that in the modified tree, you can access the number of elements lying below a node in addition to the number stored in that node. Can you use this extra information to find the median more quickly?

iii. Supposing a modified AVL tree has n elements, what is the runtime of the above procedure in terms of n? How does this runtime compare with the $O(n \log n)$ runtime described earlier?

iv. Bonus: After every insertion, the number of nodes that lie below a given node need not remain the same. For example, after four insertions, the number of nodes below the root increased and the number of nodes below the node where the key ”29” was stored, decreased. Describe a procedure that takes as input a modified AVL tree T with n nodes, an integer key k and, returns the modified AVL T', that has the key k inserted in T. What is the runtime of this procedure?

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).

Heap Problems

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. 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).