CMSC 420: Spring 2021

Midterm Exam 1

The exam is asynchronous and online. It is open-book, open-notes, open-Internet, but it must be done on your own without the aid of other people or software. The total point value is 100 points. Good luck!

Problem 1. (20 points) Consider the trees shown in Fig. 1.

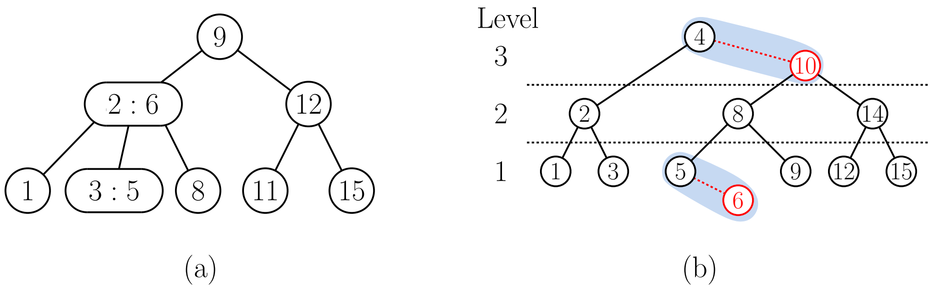


Figure 1: 2-3 and AA Trees.

(1.1) (5 points) Draw a picture of the AA tree corresponding to the 2-3 tree shown in Fig. 1(a). (Similar to Fig. 1(b), indicate which levels the nodes are at and indicate black-red pairs by connecting them with a dashed line.)

(1.2) (5 points) Draw a picture of the 2-3 tree corresponding to the AA tree shown in Fig. 1(b).

(1.3) (10 points) Show the final AA tree that results by inserting 7 into the AA tree of Fig. 1(b). (Intermediate results may be given to help with partial credit.)

Problem 2. (35 points) Short answer questions. Unless requested, explanations are not required, but may always be given to help with partial credit.

(2.1) (5 points) You have a binary tree with inorder threads (for both inorder predecessor and inorder successor). Let u and v be two arbitrary nodes in this tree. **True or false**: There is a path from u to v, using some combination of child links and threads. (No justification needed.)

(2.2) (5 points) Recall that the *depth* of a node in a tree is the number of edges along the path from the root to this node. In an inorder threaded binary tree, if `u.right` is a thread, what can be inferred about the relative depths of these two nodes? (Select one. No explanation needed.)

(a) `depth(u) < depth(u.right)`

(b) `depth(u) <= depth(u.right)`

(c) `depth(u) > depth(u.right)`

(d) `depth(u) >= depth(u.right)`

(e) We cannot infer anything. It depends on the tree's structure.

(2.3) (5 points) Suppose we have an inorder threaded binary tree, which is *full*. If `u.right` is a thread, which of the following are possible? (Select all that apply. No explanation needed.)

(a) Both `u` and `u.right` are internal.

(b) Both `u` and `u.right` are leaves.

(c) Node `u` is a leaf and `u.right` is internal.

(d) Node `u` is internal and `u.right` is a leaf.

(2.4) (5 points) You are given a sorted set of n keys $x_1 < x_2 < \cdots < x_n$ (for some large number n). You insert them all into an AA tree in some arbitrary order. No matter what insertion order to choose, one of these keys *cannot* possibly be a red node. Which is it? (Briefly explain)

(2.5) (5 points) Below we give the AA rebalancing operation `skew`, but we have explicitly commented out the check where `p == nil`). What will the modified function do if we were to call `skew(nil)`? (Select one and briefly explain your answer.)

```
AANode skew(AANode p) {
    // ----> Intentionally omitted: if (p == nil) return p;
    if (p.left.level == p.level) { // red node to our left?
        AANode q = p.left; // do a right rotation at p
        p.left = q.right;
        q.right = p;
        return q; // return pointer to new upper node
    }
    else return p; // else, no change needed
}
```

(a) It has no effect and returns a reference to `nil`.

(b) It will alter the contents of `nil`, but it will not modify any of the other nodes of the tree.

(c) It may alter both `nil` and other nodes the tree as well.

(d) It will abort due to an attempt to dereference a `null` pointer.

(2.6) (5 points) Under what circumstances will the priority value of a treap node change? (Select all that apply. No explanation needed.)

(a) Once created, it will never change (until the node is deleted).

(b) It will be changed periodically according to a random process.

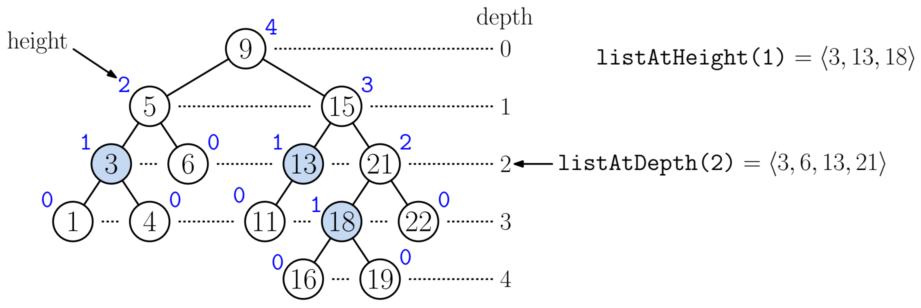
(c) It may change if the node is involved in a rotation.

(d) It may change if the node is used as a replacement for a deleted node.

(2.7) (5 points) You are given a skip list storing n items. What is the expected number of nodes that will contribute to level 3 of the skip list? (Express your answer as a function of n. Assume that level 0 is the lowest level, containing all n items. Also assume that the coin is fair, return heads half the time and tails half the time.)

Problem 3. (25 points) Recall that in a binary tree the *depth* of a node is defined to be the number of edges from the root to the node. The *height* of a node is defined to be the height of the subtree rooted at this node, that is, the maximum number of edges on any path from this node to one of its leaves.

In this problem, we will consider some questions involving nodes of a particular depth and height in an AVL tree. Let us assume (as in class) that an `AVLNode` stores its `key`, `value`, `left`, `right`, and `height`, and let us assume that the `AVLTree` stores a pointer to the `root` node.



height
depth
listAtHeight(1) = ⟨3, 13, 18⟩
listAtDepth(2) = ⟨3, 6, 13, 21⟩

Figure 2: AVL tree heights and depths.

(3.1) (5 points) Present an algorithm `listAtHeight(int h)`, which is given an integer $h \geq 0$, and returns a list (e.g., a Java `ArrayList`) containing all the keys, in increasing order, associated with all the nodes that are at height h in the AVL tree. If there are no nodes at height h, the function returns an empty list.

For example, in Fig. 2, the call `listAtHeight(1)` would return the list $\langle 3, 13, 18 \rangle$.

Briefly explain how your algorithm works, present a description (pseudocode preferred or a clear English explanation), and briefly explain its running time. The running time should be proportional to the number of nodes at height $\geq h$. (For example, in the case of `listAtHeight(1)`, there are 7 nodes of equal or greater height.)

(3.2) (5 points) Present an algorithm `listAtDepth(int d)`, which is given an integer $d \geq 0$, and returns a list (e.g., a Java `ArrayList`) containing all the keys, in increasing order, associated with all the nodes that are at depth d in the AVL tree. If there are no nodes at depth d, the function returns an empty list. **Note:** Nodes do *not* store their depths, only their heights.

For example, in Fig. 2, the call `listAtDepth(2)` would return the list $\langle 3, 6, 13, 21 \rangle$.

In each of the coding problems, briefly explain how your algorithm works, present a description (pseudocode preferred or a clear English explanation), and briefly explain its running time. The running time of your algorithm should be proportional to the number of nodes at depths $\leq d$. (For example, in the case of `listAtDepth(2)`, there are 7 nodes of equal or lesser depth.)

(3.3) (5 points) Prove that in any AVL tree, the maximum number of nodes that there are can be at depth $d \geq 0$ is 2^d. (**Hint:** This is intended to be easy. Even so, please give a short proof, even you think the observation is "obvious".)

(3.4) (10 points) Given any AVL tree T and depth $d \geq 0$, we say that T is *full at depth d* if it has 2^d nodes at depth d. (For example, the tree of Fig. 2 is full at depths 0, 1, and 2, but it is not full at depths 3 and 4.) Prove that for any $h \geq 0$, an AVL tree of height h is full at all depths from 0 up to $\lfloor h/2 \rfloor$. (For example, the AVL tree of Fig. 2 has height 4, and is full at levels 0, 1, and 2.)

Problem 4. (20 points) In this problem, we will consider generalization of the amortized analysis of the dynamic stack algorithm from Lecture 2. Recall that in the dynamic stack from the class, whenever we run out of space using an array of size k, we *expand* by allocating a new array of size $2k$, copy the elements over, and remove the old array. In this problem, we will also consider *contracting*, by halving the size of the array when we have too few elements. Below we give a formal description of our new dynamic stack and the cost of each operation. Throughout, assume that k is a power of 2.

Initialization: We allocate an array of size $k = 1$, which is empty. (Actual Cost $= 0$)

`push(x)`: If $n < k$ (standard case), we push the element onto the stack and increment n. (Actual Cost $= 1$) If $n = k$ (overflow case), we double the array size by setting $k \leftarrow 2k$, allocate a new array of this size, copy the contents of the old array into the new array, and remove the old array. We then do the standard-case push. (Actual Cost $= 2k + 1$)

`pop()`: If $n = 0$, we return `null`. (Actual Cost $= 1$) Otherwise, we pop the stack and decrement n. If (after decrementing) $n \leq \frac{k}{4}$, we halve the array size by setting $k \leftarrow \frac{k}{2}$, allocate a new array of this size, copy the contents of the old array into the new array, and remove the old array. (Actual Cost $= 1 + \frac{k}{2}$)

Note that in either case, just after reallocation, roughly half of the current array is being used. In this problem, you will derive the amortized cost of operations on this data structure. Consider a long sequence of pushes and pops, starting from any empty stack. We will break this sequence into runs, with each run ending just after each reallocation (expanding or contracting).

(4.1) (5 points) Suppose that when the run starts, the current array size is k, and the run ends with an *expansion* to size $2k$. Prove that there is a constant c (which you may choose) so that the amortized cost is c. (That is, show that the sum of actual costs in the run is at most c times the number of operations.) **Hint:** This is pretty much what we did in class. You can just adapt the proof from the lecture, but put it in your own words.

(4.2) (10 points) Suppose that when the run starts, the current array size is k, and the run ends with a *contraction* to size $\frac{k}{2}$. Prove that there is a constant c (which you may choose) so that the amortized cost is c.

(4.3) (5 points) We triggered a contraction when $n \leq \frac{k}{4}$. Suppose instead that we triggered a contraction (halving the size of the array) whenever $n \leq \frac{k}{2}$. Would the amortized cost still be a constant? Briefly justify your answer.

Because we are most interested in asymptotic bounds, you may focus mainly on the case where n and k are both large numbers.