CMSC 420: Spring 2022

Practice Problems for Midterm 1

The exam will be held in class on **Tue, Mar 8**. It is close-book, closed-notes, but you will be allowed one sheet of notes, front and back.

Problem 0. Expect at least one question of the form "apply operation X to data structure Y," where X is a data structure that has been presented in lecture. (Likely targets: AVL trees, 2-3 trees, AA trees, treaps, skiplists). Here is an example from last semester.

(a) Consider the 2-3 tree shown the figure below. Show the **final tree** that results after the operation `insert(6)`. When rebalancing, use only splits, *no adoptions* (key rotations).

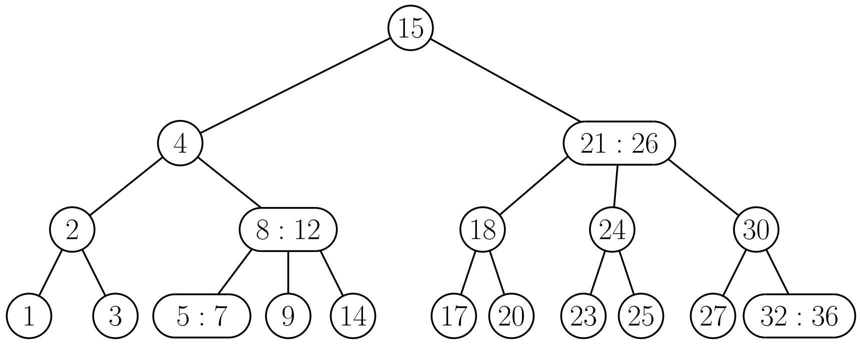


Figure 1: 2-3 tree insertion and deletion.

(b) Returning to the original tree, show the **final tree** that results after the operation `delete(20)` When rebalancing, you may use *both merge and adoption* (key rotation). If either operation can be applied, give priority to adoptions.

In both cases, you should use the algorithm presented in class. (You will receive partial credit if you produce a valid 2-3 tree, but not using the algorithm from class.)

Hint: Don't waste too much time showing intermediate results. You can return to this if you have spare time.

Problem 1. Short answer questions. Except where noted, explanations are not required, but may be given to help with partial credit.

(a) A binary tree is *full* if every node either has 0 or 2 children. Given a full binary tree with n total nodes, what is the maximum number of leaf nodes? What is the minimum number? Give your answer as a function of n (no explanation needed).

(b) **True or false?** Let T be extended binary search tree (that is, one having internal and external nodes). In an inorder traversal, internal and external nodes are encountered in *alternating order*. (If true, provide a brief explanation. If false, show a counterexample.)

(c) **True or false?** In every extended binary tree having n external nodes, there exists an external node of depth $\leq \lceil \lg n \rceil$. Explain briefly. **Hint: Read this carefully before answering.**

(d) 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.)

(e) What are the minimum and maximum number of levels in a 2-3 tree with n nodes. (Define the number of levels to be the height of the tree plus one.) Hint: It may help to recall the formula for the geometric series: $\sum_{i=0}^{m-1} c^i = (c^m - 1)/(c - 1)$.

(f) You are given a 2-3 tree of height h, which has been converted into an AA-tree. As a function of h, what is the *minimum* number of *red nodes* that might appear on any path from a root to a leaf node in the AA tree? What is the *maximum* number? Briefly explain.

(g) Unbalanced search trees, treaps and skiplists all support dictionary operations in $O(\log n)$ "expected time." What difference is there (if any) in the meaning of "expected time" in these contexts?

(h) You have a valid AVL tree with n nodes. You insert two keys, one smaller than all the keys in the tree and the other larger than all the keys in the tree, but you do no rebalancing after these insertions. **True or False**: The resulting tree is a valid AVL tree. (Briefly explain.)

(i) By mistake, two keys in your treap happen to have the same priority. Which of the following is a possible consequence of this mistake? (Select one)

   (i) The `find` algorithm may abort, due to dereferencing a `null` pointer.

   (ii) The `find` algorithm will not abort, but it may return the wrong result.

   (iii) The `find` algorithm will return the correct result if it terminates, but it might go into an infinite loop.

   (iv) The `find` algorithm will terminate and return the correct result, but it may take longer than $O(\log n)$ time (in expectation over all random choices).

   (v) There will be no negative consequences. The find algorithm will terminate, return the correct result, and run in $O(\log n)$ time (in expectation over all random choices).

(j) 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)

(k) You are given a skip list storing n items. What is the expected number of nodes that are at level 3 and higher in 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 2. You are given a degenerate binary search tree with n nodes in a left chain as shown on the left of Fig. 2, where $n = 2^k - 1$ for some $k \geq 1$.

(a) Derive an algorithm that, using only single left- and right-rotations, converts this tree into a perfectly balanced complete binary tree (right side of Fig. 2).

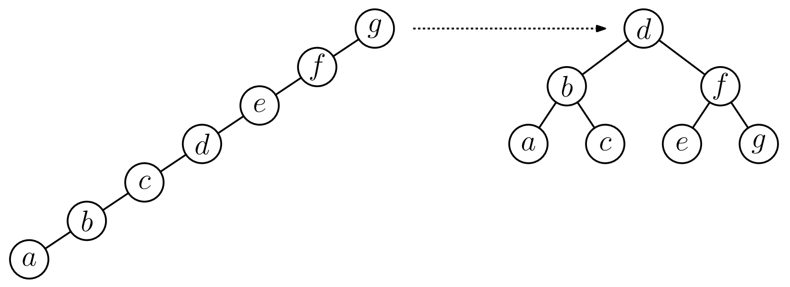


Figure 2: Rotating into balanced form.

(b) As an asymptotic function of n, how many rotations are needed to achieve this? $O(\log n)$? $O(n)$? $O(n \log n)$? $O(n^2)$? Briefly justify your answer.

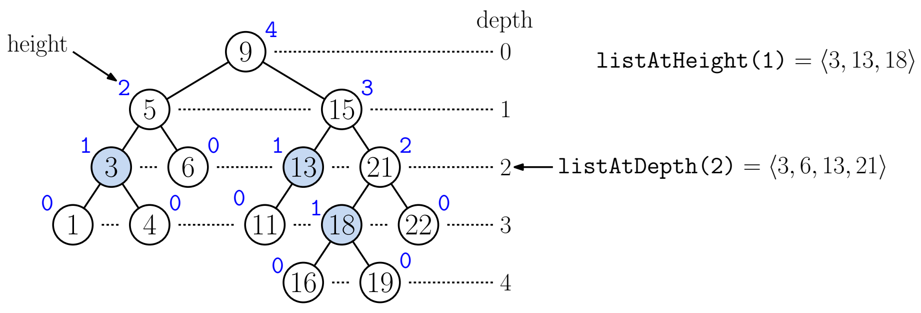
Problem 3. You are given a binary search tree where, in addition to the usual fields `p.key`, `p.left`, and `p.right`, each node `p` has a *parent link*, `p.parent`. This points to `p`'s parent, and is `null` if `p` is the root. Given such a tree, present pseudo-code for a function

```
Node preorderPred(Node p)
```

which is given a non-null reference `p` to a node of the tree and returns a pointer to `p`'s *preorder predecessor* in the tree (or `null` if `p` has no preorder predecessor). Your function should run in time proportional to the height of the tree. Briefly explain how your function works.

Problem 4. 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
0
1
2
3
4
listAtHeight(1) = $\langle 3, 13, 18 \rangle$
listAtDepth(2) = $\langle 3, 6, 13, 21 \rangle$
9
5
15
3
6
13
21
1
4
11
18
22
16
19

Figure 3: AVL tree heights and depths.

(a) 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. 3, 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.)

(b) 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. 3, 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.)

(c) 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".)

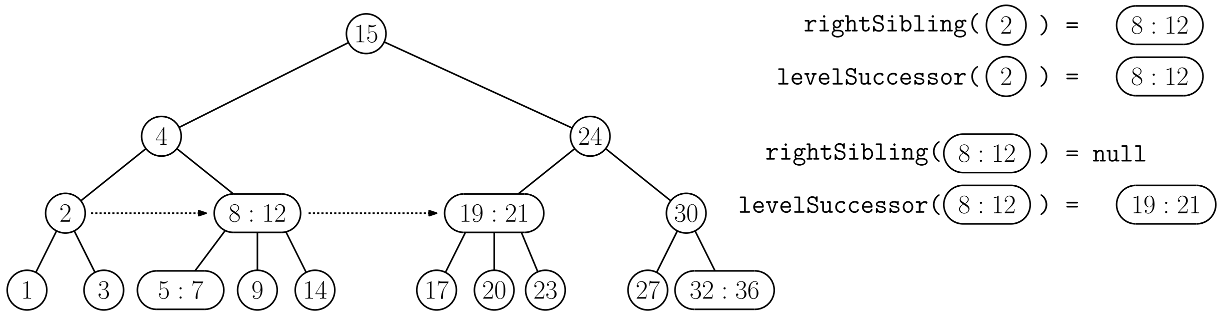
(d) Given any AVL tree T and depth $d \geq 0$, we say that T is *full at depth d* if it has 2^d nodes at depth d. (For example, the tree of Fig. 3 is full at depths 0, 1, and 2, but it is not full at depths 3 and 4.) Prove that for any $h \geq 0$, an AVL tree of height h is full at all depths from 0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

Problem 5. Consider the following possible node structure for 2-3 trees, where in addition to the keys and children, we add a link to the parent node. The root's parent link is `null`.

```
class Node23 {                  // a node in a 2-3 tree
    int       nChildren         // number of children (2 or 3)
    Node23    child[3]          // our children (2 or 3)
    Key       key[2]            // our keys (1 or 2)
    Node23    parent            // our parent
}
```

Assuming this structure, answer each of the following questions:

(a) Present pseudocode for a function `Node23 rightSibling(Node23 p)`, which returns a reference to the sibling to the immediate right of node `p`, if it exists. If `p` is the rightmost child of its parent, or if `p` is the root, this function returns `null`. (For example, in Fig. 4, the right sibling of the node containing "2" is the node containing "8:12". Since the node containing "8:12" is the rightmost node of its parent ("4"), it has no right sibling.)



15
4
24
2
8 : 12
19 : 21
30
1
3
5 : 7
9
14
17
20
23
27
32 : 36
rightSibling(2) = 8 : 12
levelSuccessor(2) = 8 : 12
rightSibling(8 : 12) = null
levelSuccessor(8 : 12) = 19 : 21

Figure 4: Sibling and level successor in a 2-3 tree.

Your function should run in $O(1)$ time.

(b) For a node `p` in a 2-3 tree, its *level successor* is the node to its immediate right at the same level. Give pseudocode for a function `Node23 levelSuccessor(Node23 p)`, which returns a reference to `p`'s level successor, if it exists. If `p` is the rightmost node on its level (including the case where `p` is the root), this function returns `null`. (For example, in Fig. 4, the level successor of the node containing "2" is the node containing "8:12", and the level successor of "8:12" is the node containing "19:21".)

Your function should run in $O(\log n)$ time. If you like, you may use `rightSibling`.

(c) Suppose we start at any node `p` in a 2-3 tree with n nodes, and we repeatedly perform `p = levelSuccessor(p)` until `p == null`. What is the (worst-case) total time needed to perform all these operations? (Briefly justify your answer.)

Problem 6. A *social-distanced bit vector* (SDBV) is an abstract data type that stores bits, but no two 1-bits are allowed to be consecutive. It supports the following operations (see Fig. 5):

- `init(m)`: Creates an empty bit vector `B[0..m-1]`, with all entries initialized to zero.
- `boolean set(i)`: For $0 \leq i \leq m$ (where m is the current size of `B`), this checks whether the bit at positions i and its two neighboring indices, $i-1$ and $i+1$, are all zero. If so, it sets the ith bit to 1 and returns `true`. Otherwise, it does nothing and returns `false`. (The first entry, `B[0]`, can be set, provided both it and `B[1]` are zero. The same is true symmetrically for the last entry, `B[m-1]`.)

  For example, the operation `set(9)` in Fig. 5 is successful and sets `B[9] = 1`. In contrast, `set(8)` fails because the adjacent entry `B[7]` is nonzero.

There is one additional feature of the SDBV, its ability to *expand*. If we ever come to a situation where it is impossible set any more bits (because every entry of the bit vector is either nonzero or it is adjacent to an entry that is nonzero), we *reallocate* the bit vector to one of three times the current size. In particular, we replace the current array of size m with an array of size $3m$, and we copy all the bits into this new array, compressing them as much as possible. In particular, if k bits of the original vector were nonzero, we set the entries $\{0, 2, 4, \ldots, 2k\}$ to 1, and all others to 0 (see Fig. 5).

The cost of the operation `set` is 1, unless a reallocation takes place. If so, the cost is m, where m is the size of the bit vector *before* reallocation.

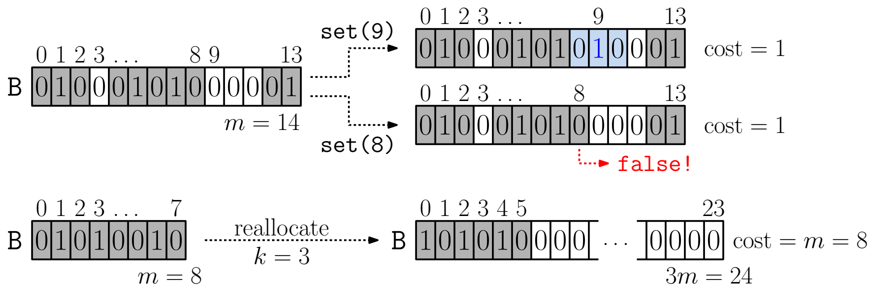


Figure 5: Social-distanced bit vector. (Shaded entries cannot be set to one, due to social-distancing.)

Our objective is to derive an amortized analysis of this data structure.

(a) Suppose that we have arrived at a state where we need to reallocate an array of size m. As a function of m, what is the minimum and maximum number of bits of the SDBV that are set to 1? (Briefly explain.)

(b) Following the reallocation, what is the minimum number of operations that may be performed on the data structure until the next reallocation event occurs? Express your answer as a function of m. (Briefly explain.)

(c) As a function of m, what is the cost of this next reallocation event? (Briefly explain.)

(d) Derive the amortized cost of the SDBV. (For full credit, we would like a tight constant, as we did in the homework assignment. We will give partial credit for an asymptotically correct answer. Assume the limiting case, as the number of operations is very large and the initial size of the bit vector is small.)

Throughout, if divisions are involved, don't worry about floors and ceilings.