CMSC 420: Spring 2023

Homework 2: Search Trees

Handed out Tue, Feb 28. Due **Tue, Mar 7, 9:30am** (that is, by the start of class).

Important! Solutions will be discussed in class right after the due date, so **no late submissions will be accepted**. Turn in whatever you have completed by the due date.

Problem 1. (8 points) Perform the following operations on the AVL trees shown in Fig. 1. In each case, show the final tree and list (in order) all the rebalancing operations performed (e.g., "`rotateLeftRight(7)`"). (We only need the final tree, but intermediate results may be shown for the sake of assigning partial credit.) Draw your final tree as in Fig. 1(b) from Lecture 7. Show the balance factors at each node. (Don't bother giving the heights.)

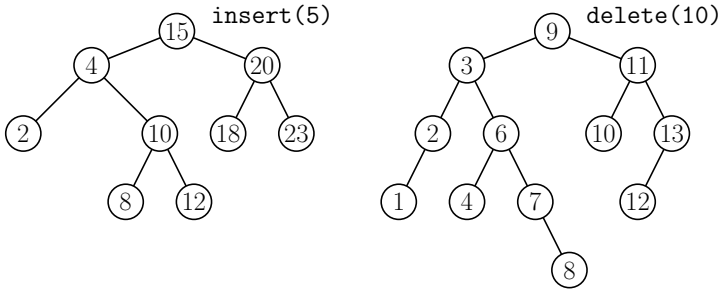

Figure 1: AVL-tree operations.

(a) (4 points) Show the result of executing the operation `insert(5)` to the tree of Fig. 1(a).

(b) (4 points) Show the result of executing the operation `delete(10)` to the tree of Fig. 1(b).

Problem 2. (8 points) Consider the AA trees shown in Fig. 2.

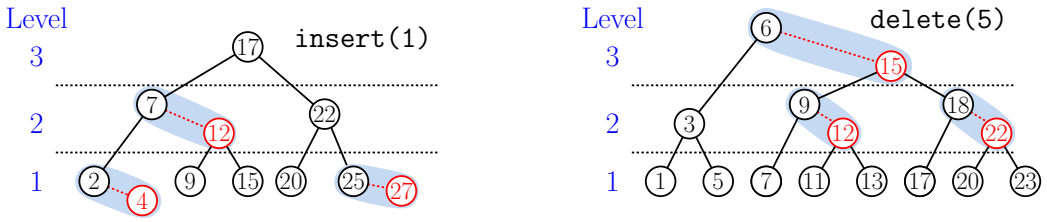

Figure 2: AA-tree operations.

(a) (4 points) Show the result of executing the operation `insert(1)` to the tree on the left.

(b) (4 points) Show the result of executing the operation `delete(5)` to the tree on the right.

In each case, show the final tree and list (in order) all the rebalancing operations (skew, split, and update-level) that result in changes to the tree (e.g., "`skew(13)`"). Intermediate results may be shown for the sake of partial credit.

Draw the tree as in Figs. 6 and 7 from Lecture 9. Indicate both the levels and distinguish red from black nodes. You do not need to color the nodes—a dashed line coming in from the parent indicates that a node is red. (Do not bother drawing `nil`.)

Problem 3. (6 points) Recall the minimal AVL trees from Lecture 7. Formally, we define T_0 and T_1 as shown in Fig. 3(a), and for any $h \geq 2$, T_h consists of a root with T_{h-1} as its left subtree and T_{h-2} as its right subtree (see Fig. 3(b)). Suppose we label the nodes in sequence $\langle 1, 2, 3, 4, 5, \ldots \rangle$ according to an inorder traversal (see Fig. 3(c)). Observe that the nodes along the leftmost chain of the tree will take on values from the Fibonacci sequence. Establish this by giving a (formal) proof for the following theorem.

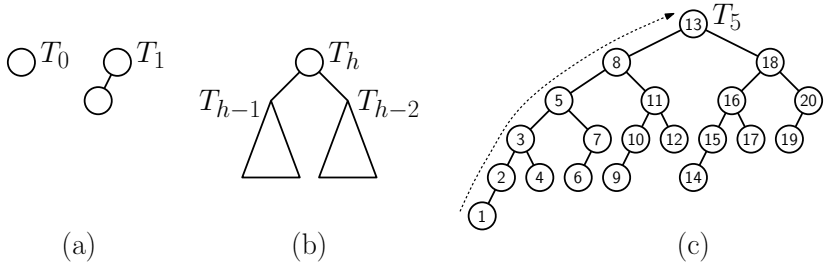

Figure 3: Inorder labeling of a Fibonacci-based trees.

Theorem: For any $h \geq 0$, if the nodes of T_h are labeled according to their position in an inorder traversal of the tree (starting with 1), then the labels along the leftmost chain of tree (from leaf to root) generate the Fibonacci sequence

$$\langle F(2), F(3), F(4), F(5), \ldots, F(h+2) \rangle,$$

where $F(h)$ denotes the hth Fibonacci number.

Hint: If it helps, you may assume the result proved in class about these trees, namely that T_h has $F(h+3) - 1$ nodes. (In the lecture notes, we called this $N(h)$.)

Problem 4. (12 points) Suppose that you are implementing the 2-3 tree as described in class, and you have the node structure shown below. Each node has an additional parent link. As in the lecture, we will "cheat" and allow nodes to have 4 children, but this is only temporarily allowed. Since they do not matter for this exercise, we will only store keys, no values. We also provide two constructors, one for 2-nodes and one for 3-nodes (see Fig. 4).

```
class Node {
    int nChild // number of children (1 through 4)
    Node parent // parent of this node (null if root)
    Node[] child // a 4-element array of child references
    Key[] key // a 3-element array of keys

    // constructors for 2-nodes and 3-nodes
    Node(Node par, Node u, Key x, Node v)
    Node(Node par, Node u, Key x, Node v, Key y, Node w)
}
```

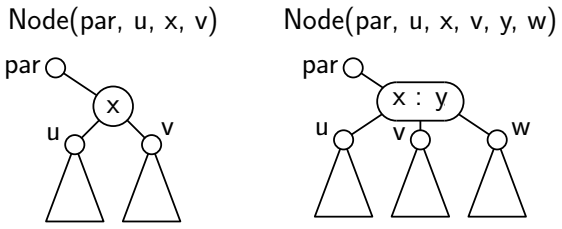

Figure 4: Constructors for 2-3 nodes.

(a) (3 points) Present pseudocode for a function `Node leftSib(Node p)`, which returns a reference to the left sibling of `p`. If `p` has no left sibling, it returns `null`. Your function should run in constant time. You may assume that all nodes other than `p` are valid 2-3 nodes (that is, they have 2 or 3 children.) `Hint:` Be sure to avoid dereferencing `null` pointers or indexing outside of array bounds.

(b) (3 points) Repeat (a) but now for `Node rightSib(Node p)`, which returns the right sibling.

(c) (6 points) Present pseudocode for a function `Node merge(Node p)`. It is given a node `p` that contains only 1 child (and no keys). If it has a left sibling and this sibling is a 2-node, it merges its contents with this sibling node, creating a new node, which it returns. Otherwise, if it has a right sibling, and this sibling is a 2-node, it does the same with this sibling (see Fig. 5). Otherwise, it returns `null`. You may assume that all nodes other than `p` are valid 2-3 nodes (that is, they have 2 or 3 children.) **Do not worry about updating the parent node.** (We'll leave that for a future exercise.)

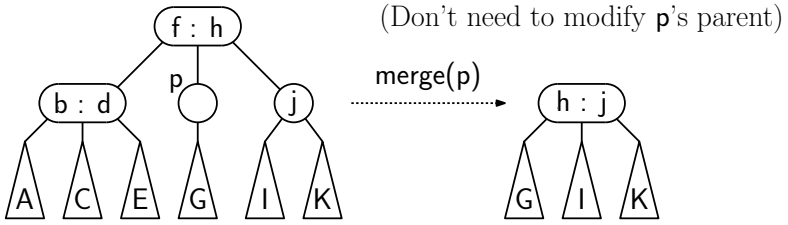

Figure 5: The function `merge(p)`.

Problem 5. (16 points) Alice and Bob want to test their implementation of a standard (unbalanced) binary search tree. They know that the tree will perform badly if keys are inserted in sorted order, and it will perform well (in expectation) if the keys are inserted in random order. They decide to analyze the performance of one more insertion order.

Let us assume that the number of keys n is chosen to be a perfect square, that is, $n = k^2$ for some $k \geq 1$. They first write the keys out row-by-row in a matrix. For example, here is what they would generate for $n = 16$:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Then, they insert the keys by reading down the columns, working from right to left. In the above example, the insertion order is $\langle 4, 8, 12, 16, 3, 7, 11, 15, 2, 6, 10, 14, 1, 5, 9, 13 \rangle$.

Answer the following questions based on Alice and Bob's insertion process. **Throughout, you may assume that the number of nodes n is a strictly positive perfect square.**

(a) (3 points) Draw the final binary search tree that results for the above example involving $n = 16$ keys.

(b) (3 points) What is the height of the resulting tree? (Express your answer as a function of n.) **Hint:** To avoid being off by 1, recall the definition of height from the lecture notes. A tree with a single node has height 0, not 1.

(c) (4 points) Letting h denote the height of the final tree. Give a formula $d(i)$, which for $0 \leq i \leq h$ indicates the number of nodes at depth i in the resulting tree. Express your answer as a function of n and i. **Hint:** Recall the definition of depth from the lecture notes. The root is at depth 0, and depths increase from there.

(d) (6 points) Under the reasonable assumption that the time needed to insert a node at depth i is $i + 1$, what is the total time needed to insert all n keys in the tree? Express your answer as a function of n in closed form (no summations or recurrences). For full credit, give the exact formula. For partial credit, give an asymptotically tight bound. `Hint:` It may be useful to recall the summation formulas from the CMSC420 Reference Guide. **Show how you derived your answer.**

Challenge Problem 1: Augment your answer to Problem 4(c) by presenting pseudocode that not only creates the new merged node, but (assuming that the result is not `null`) also updates the contents of the parent node. Note that the parent's degree decreases by one, and so it may become a 1-node.

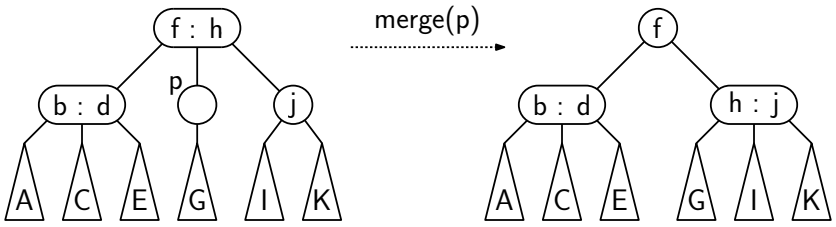

Figure 6: The complete function `merge(p)`.

Challenge Problem 2: In programming languages like C and C++, when a data structure is deallocated, its nodes must be explicitly deallocated one by one. Given a node `p`, the operation `delete(p)` deallocates the node, returning its memory block to the system. The simplest way to do this is to perform a postorder traversal of the tree, and delete nodes as you are backing out.

Performing a postorder traversal requires the use of system memory in the form of the recursion stack. If the tree is really huge, you may not have enough memory to store the recursion stack. Ironically, you may run out of memory in the process of trying to free up memory!

Describe a method for deleting all the nodes of a standard binary tree that uses only $O(1)$ additional working storage. (Note that using recursion, allocating arrays or other data structures, and allocating new nodes all contribute to your working storage.)

Your binary tree is absolutely minimal. Each node just has a left child link and a right child link, and nothing else (no keys, no values, no parent links.) Your algorithm should run in $O(n)$ time, where n is the number of nodes in the tree. **Hint:** You are allowed to modify the contents of the tree.