CMSC 420: Spring 2021

Final Exam

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 120 points. Good luck!

Problem 1. (35 points) Short answer questions. Except where noted, explanations are not required but may be given for partial credit.

(1.1) (2 pts) What was play on words with the data structure named "deque"?

(1.2) (6 pts) Recall that a *free tree* is an unrooted tree. A *leaf* in a free tree is a node that has exactly one incident edge. Consider a free tree with $n \geq 200$ nodes. As a function of n, what is the *minimum* number of leaves the tree can have? What is the *maximum* number of leaves the tree can have? (Draw two pictures to illustrate your answers.)

(1.3) (3 pts) You just inserted a key into an AVL tree, where the newly inserted key is larger than any other key in the tree. **True or false**: If a rotation was needed to balance the tree after the insertion, it was a *single rotation* (not a double rotation). Briefly explain.

(1.4) (3 pts) It is a fact that any 2-3 tree storing a set of n keys has height less than or equal to the height of an AVL tree storing the same set of keys. **True or false:** Given a 2-3 tree and AVL tree, both storing the same set of keys, the number of comparisons needed to perform `find(x)` in the 2-3 tree is less than or equal to the number of comparisons needed to perform the same operation in the AVL tree. Briefly explain.

(1.5) (3 pts) The splay operation promotes a key to the root of the splay tree through rotations. While we could have achieved this through simple Zig rotations, we instead preferred Zig-Zag and Zig-Zig rotations. Why does using Zig rotations alone fail to produce a data structure that guarantees efficient amortized performance? (You may want to explain with the help of a drawing.)

(1.6) (6 pts) Given a B-tree of order $m = 15$, what is the minimum and maximum number of children that any internal node might have, assuming this node is not the root of the tree. What if the node is the root?

(1.7) (6 pts) We showed in class that if we alternate splitting between x and y in a height-balanced kd-tree storing n points in $\mathbb{R}^2$, the number of nodes whose associated cell is stabbed by any axis parallel line is $O(\sqrt{n})$.

- Show (by drawing a picture) that this may fail if the tree is *not height balanced*.
- Show (by drawing a picture) that this may fail if the splitting directions *do not alternate*.

(1.8) (3 pts) The fastest known algorithm for building a binary search tree from a set of n keys, runs in time $O(n \log n)$. But, with scapegoat trees, we claimed that we could rebuild a subtree containing m keys in time $O(m)$. How is this possible?

(1.9) (3 pts) The *Cartesian tree* by Jean Vuillemin and the *priority search tree* of Edward McCreight are both closely related to what other data structure that we have studied this semester?

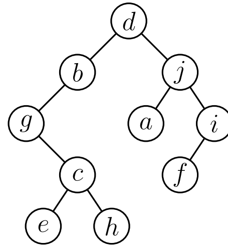

Figure 1: Tree traversals.

Problem 2. (15 points) This problem involves the binary tree shown in Fig. 1.

(2.1) (3 points) List the nodes according to a *preorder traversal*

(2.2) (3 points) List the nodes according to an *inorder traversal*

(2.3) (4 points) List the nodes according to the order given in the pseudocode block below. The initial call is `crazyTraverse(root)`, where `root` is the root of the tree.

```
void crazyTraverse(Node p) { // a crazy way to traverse a tree
    if (p != null) {
        print(p.key)
        crazyTraverse(p.right)
        if (p.left != null) {
            print(p.key)
            crazyTraverse(p.left)
        }
    }
}
```

(2.4) (5 points) Present pseudocode for a tree traversal procedure that produces the output shown below[1] when run on the tree of Fig. 1. Notice that each key is printed twice.

`d j i f f i a a j b g c h h e e c g b d`

Problem 3. (15 points) In this problem we will build a suffix tree for the string $S =$ `baabaabababaa$`.

(3.1) (5 points) List the substring identifiers for the 14 suffixes of S. For the sake of uniformity, list them in order (either back to front or front to back). For example, you could start with "`$`" and end with the substring identifier for the entire string.

(3.2) (2 points) List the substring identifiers again together with their indices (0 through 13), but this time in alphabetical order (where `"a"` $<$ `"b"` $<$ `"$"`).

(3.3) (8 points) Draw a picture of the suffix tree for S. For the sake of uniformity, when drawing your tree, use the convention of Fig. 7 in the Lecture 17 LaTeX lecture notes. In particular, label edges of the final tree with substrings, index the suffixes from 0 to 13, and order subtrees in ascending lexicographical order.

[1] An early version of the exam contained an extra character "g". If you had this version of the exam and lost points for this reason, please file a regrade request.

Problem 4. (25 points) This problem involves an input which is a binary search tree having n nodes of height $O(\log n)$. You may assume that each node `p` has a field `p.size` that stores the number of nodes in its subtree (including `p` itself). Here is the node structure:

```
class Node {
    int key;
    Node left;
    Node right;
    int size; // number of nodes in this subtree
}
```

(4.1) (10 points) Present pseudocode for a function `printMaxK(int k)`, which is given $0 \leq k \leq n$, and prints the values of the k largest keys in the binary search tree. (See, for example, Fig. 2.)

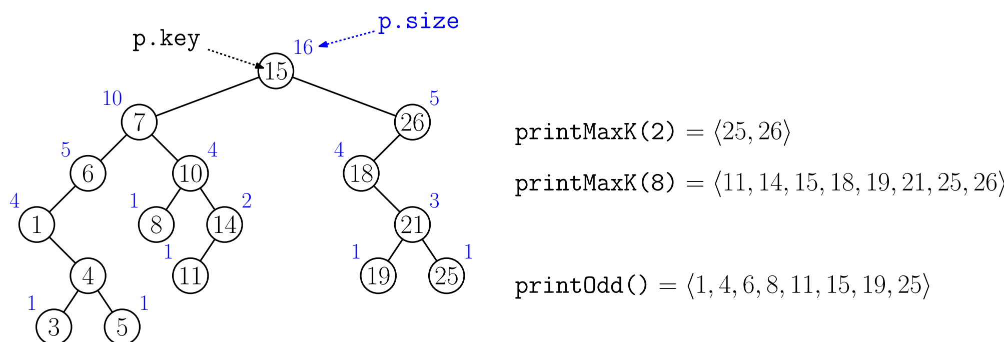

Figure 2: The functions printMaxK and printOdd.

You should do this by traversing the tree. You are not allowed to "cheat" but storing an auxiliary list of sorted nodes.

For fullest credit, the keys should be printed in *ascending order*, and your algorithm should run in time $O(k + \log n)$ (see part (4.2) below). (Partial credit will be given otherwise, but an $O(n)$ time algorithm is not worth anything.)

You may assume that $0 \leq k \leq n$, where n is the total number of nodes in the tree. Briefly explain your algorithm. (The running time will be derived in (4.2).)

Hint: I would suggest using the helper function `printMaxK(Node p, int k))`, where `k` is the number of keys to print from the subtree rooted at `p`.

(4.2) (5 points) Derive the running time of your algorithm in (4.1).

(4.3) (10 points) Present pseudocode for a function `printOdd()`, which does the following. Let $\langle x_1, x_2, \ldots, x_n \rangle$ denote the keys of the tree in ascending order, this function prints every other key, namely $\langle x_1, x_3, x_5, \ldots, x_n \rangle$, if n is odd, and $\langle x_1, x_3, x_5, \ldots, x_{n-1} \rangle$, if n is even.

Beware: We are not printing the "odd-valued" keys, rather we are printing the odd numbered positions in the sorted order. (See Fig. 2.)

Again, you should do this by traversing the tree. You are not allowed to "cheat" by storing auxiliary lists or using global variables. Your program should run in time $O(n)$. Briefly explain your algorithm.

Problem 5. (15 points) This problem involves a data structure called an *erasable stack*. This data structure is just a stack with an additional operation that allows us to "erase" any element that is currently in the stack. Whenever we pop the stack, we skip over the erased elements, returning the topmost "unerased" element. The pseudocode below provides more details be implemented.

```
class EStack {      // erasable stack of Objects
    int top         // index of stack top
    Object A[HUGE] // array is so big, we will never overflow
    Object ERASED  // special object which indicates an element is erased

    EStack() { top = -1 }  // initialize

    void push(Object x) {  // push
       A[++top] = x
    }

    void erase(int i) {    // erase (assume 0 <= i <= top)
        A[i] = ERASED
    }

    Object pop() {         // pop (skipping erased items)
        while (top >= 0 && A[top] == ERASED) top--
        if (top >= 0) return A[top--]
        else return null
    }
}
```

Let $n = \texttt{top} + 1$ denote the current number of entries in the stack (including the `ERASED` entries). Define the *actual cost* of operations as follows: `push` and `erase` both run in 1 unit of time and `pop` takes $k + 1$ units of time where k is the number of `ERASED` elements that were skipped over.

(5.1) (2 points) As a function of n, what is the *worst-case running time* of the `pop` operation? (For fullest credit, make your bound as tight as possible.) Justify your answer.

(5.2) (8 points) Starting with an empty stack, we perform a sequence of m `push`, `erase`, and `pop` operations. Give an upper bound on the *amortized running time* of such as sequence. You may assume that all the operations are valid and the array never overflows. (For fullest credit, make your bound as tight as possible.) Justify your answer.

(5.3) (5 points) Given two (large) integers k and m, where $k \leq m/2$, we start from an empty stack, push m elements, and then erase k elements *at random*, finally we perform a single `pop` operation. What is the *expected running time* of the final pop operation. You may express your answer asymptotically as a function of k and m.

In each case, state your answer first, and then provide your justification.

Problem 6. (15 points) In this problem, we will consider how to use/modify range trees to answer two related queries. While the answer should be based on range trees, you may need to

make modifications including possibly transforming the points and even adding additional coordinates. In each case, describe the points that are stored in the range tree and how the search process works. An English explanation (as opposed to pseudocode) is sufficient. Justify your algorithm's correctness and derive its running time.

(6.1) (10 points) Assume you are given an n-element point set P in $\mathbb{R}^2$ (see Fig. 3(a)). In addition to its coordinates (p_x, p_y), each point $p \in P$ is associated with a numeric *rating*, p_z. In an *orthogonal top-k query*, you are given an axis-aligned query rectangle R (given, say, by its lower-left and upper-right corners) and a positive integer k. The query returns a list of the (up to) k points of P that lie within R having the highest ratings (see Fig. 3(b)). (As an application, imagine you are searching for the k highest rated restaurants in a rectangular region of some city.)

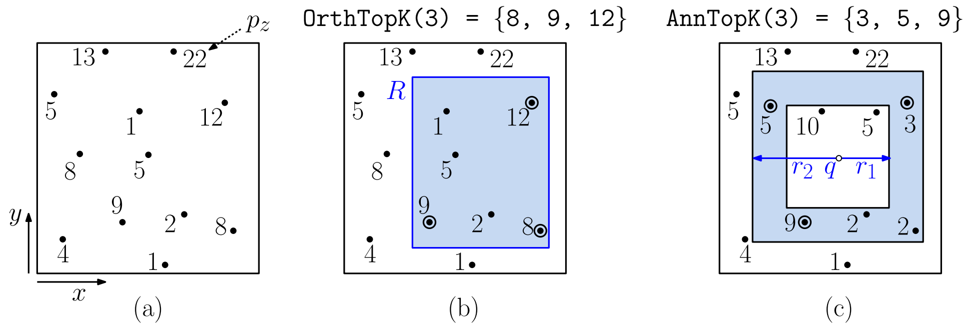

Figure 3: Orthogonal top-k queries and annulus top-k queries.

Describe how to preprocess the point set P into a data structure that can efficiently answer any orthogonal top-k query (R, k). Your data structure should use $O(n \log^2 n)$ storage and answer queries in at most $O(k \log^2 n)$ time. (I don't care how you handle edge cases, such as points lying on the boundary of the rectangle or points having the same rating.) If there are k points or fewer in the query region, the list will contain them all.

(6.2) (5 points) In an *annulus top-k query* a query is given by a query point $q \in \mathbb{R}^2$ and two positive radii $r_1 < r_2$. Let $S_1 = S(q, r_1)$ be the square centered at q whose half side length is r_1 and define S_2 similarly for q and r_2. The square annulus $A(q, r_1, r_2)$ is defined to be the region between these two squares. The query returns a list of the (up to) k points of P that lie within the annulus $A(q, r_1, r_2)$ that have the highest ratings (see Fig. 3(c)).

In Problem 4, we saw how to print the k-largest entries from a balanced binary tree. You may use that result here if you like.