CMSC 420: Spring 2022

Practice Problems for Midterm 2

The exam will be held in class on **Tue, Apr 12**. It is close-book, closed-notes, but you will be allowed one sheet of notes, front and back.

Disclaimer: These practice problems have been extracted from old homework assignments and exams. Material changes from semester to semester. These do **not** necessarily reflect the actual coverage, difficulty, or length of the midterm exam.

Problem 0. Expect at least one problem that involves working through some operations on a data structure that we have covered since the previous exam.

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

(a) What is the purpose of the *next-leaf* pointer in B+ trees?

(b) Suppose you know that a very small fraction of the keys in an ordered dictionary data structure are to be accessed most of the time, but you do not know which these keys are. Among the various data structures we have seen this semester, which would be best for this situation? Explain briefly.

(c) Both scapegoat trees and splay trees provide $O(\log n)$ amortized time for standard dictionary operations (insert, delete, and find). Suppose that your application involves many more find operations than insertions or deletions. Which of these two structures would you prefer and why?

(d) What is the maximum number of points that can be stored in a 3-dimensional point quadtree of height h? Express your answer as an exact (not asymptotic) function of h. (**Hint:** It may be useful to recall the formula for any $c > 1$, $\sum_{i=0}^{m} c^i = (c^{m+1})-1)/(c-1)$.)

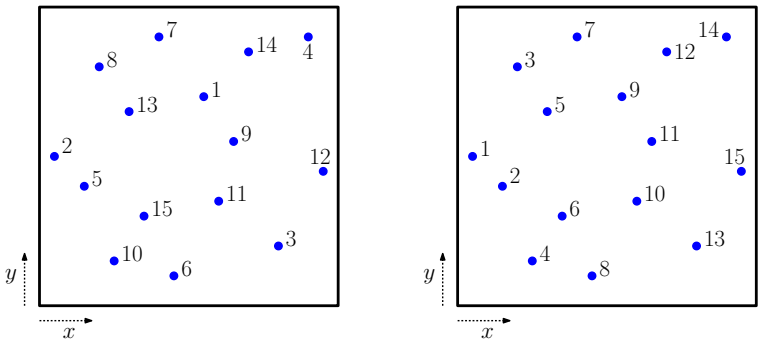

Figure 1: Height of kd-tree.

(e) We have n uniformly distributed points in the unit square, with no duplicate x- or y-coordinates. Suppose we insert these points into a kd-tree in *random* order (see the left side of Fig. 1). As in class, we assume that the cutting dimension alternates between x and y. As a function of n what is the expected height of the tree? (No explanation needed.)

(f) Same as the previous problem, but suppose that we insert points in *ascending* order of x-coordinates, but the y-coordinates are *random* (see the right side of Fig. 1). What is the expected height of the tree? (No explanation needed.)

(g) (**Note:** This problem is only applicable for the probing methods we discuss up to the time of the midterm.) You are using hashing with open addressing. Suppose that the table has just one empty slot in it. In which of the following cases are you *guaranteed* to succeed in finding the empty slot? (Select all that apply.)

(1) Linear probing (under any circumstances)
(2) Quadratic probing (under any circumstances)
(3) Quadratic probing, where the table size m is a prime number
(4) Double hashing (under any circumstances)
(5) Double hashing, where the table size m and hash function $h(x)$ are relatively prime
(6) Double hashing, where the table size m and secondary hash function $g(x)$ are relatively prime

Problem 2. Suppose that you are given a treap data structure storing n keys. The node structure is shown in Fig. 2. You may assume that *all keys and all priorities are distinct.*

```
class TreapNode {
    Key key           // key
    int priority      // priority
    TreapNode left    // left child
    TreapNode right   // right child
}
```

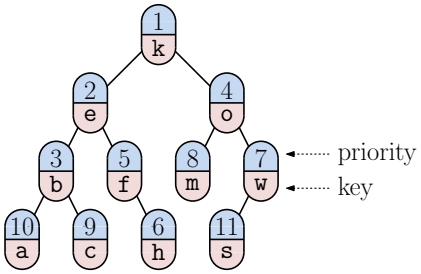

Figure 2: Treap node structure and an example.

(a) Present pseudocode for the operation `int minPriority(Key x0, Key x1)`, which is given two keys x_0 and x_1 (which may or may not be in the treap), and returns the lowest priority among all nodes whose keys x lie in the range $x_0 \le x \le x_1$. If the treap has no keys in this range, the function returns `Integer.MAX_VALUE`. Briefly explain why your function is correct.

For example, in Fig. 2 the query `minPriority("c", "g")` would return 2 from node `"e"`, since it is the lowest priority among all keys x where `"c"` $\le x \le$ `"g"`.

(b) Assuming that the treap stores n keys and has height $O(\log n)$, what is the running time of your algorithm? (Briefly justify your answer.)

Problem 3. Define a new treap operation, `expose(Key x)`. It finds the key `x` in the tree (throwing an exception if not found), sets its priority to $-\infty$ (or more practically `Integer.MIN_VALUE`), and then restores the treap's priority order through rotations. (Observe that the node containing `x` will be rotated to the root of the tree.) Present pseudo-code for this operation.

Problem 4. For this problem, assume that the structure of a node in a skip list is as follows:

```
class SkipNode {
    Key key;           // key
    Value value;       // value
    SkipNode[] next;   // array of next pointers
}
```

The height of a node (that is, the number of levels to which it contributes) is given by the Java operation `p.next.length`.

Often, when dealing with ordered dictionaries, we wish to perform a sequence of searches in sorted order. Suppose that we have two keys, $x < y$, and we have already found the node `p` that contains the key x. In order to find y, it would be wasteful to start the search at the head of the skip list. Instead, we want to start at `p`. Suppose that there are k nodes between x and y in the skip list. We want the expected search time to be $O(\log k)$, not $O(\log n)$.

(a) Present pseudo-code for an algorithm for a function `Value forwardSearch(p, y)`, which starting a node `p` (whose key is smaller than y), finds the node of the skip list containing key y and returns its value. (For simplicity, you may assume that key y appears within the skip list.) In addition to the pseudo-code, briefly explain how your method works.

Show that the expected number of hops made by your algorithm is $O(\log k)$, where k is the number of nodes between x and y. The proof involves showing two things:

(b) Prove that the maximum level reached is $O(\log k)$ in expectation (over random coin tosses).

(c) Prove that the number of hops per level is $O(1)$ in expectation.

Problem 5. In this problem we will consider an enhanced version of a skip list. As usual, each node `p` stores a key, `p.key`, and an array of next pointers, `p.next[]`. To this we add a parallel array `p.span[]`, which contains as many elements as `p.next[]`. This array is defined as follows. If `p.next[i]` refers to a node `q`, then `p.span[i]` contains the distance (number of nodes) from `p` to `q` (at level 0) of the skip list.

For example, see Fig. 3. Suppose that `p` is third node in the skip list (key value "10"), and `p.next[1]` points to the fifth element of the list (key value "13"), then `p.span[1]` would be $5 - 3 = 2$, as indicated on the edge between these nodes.

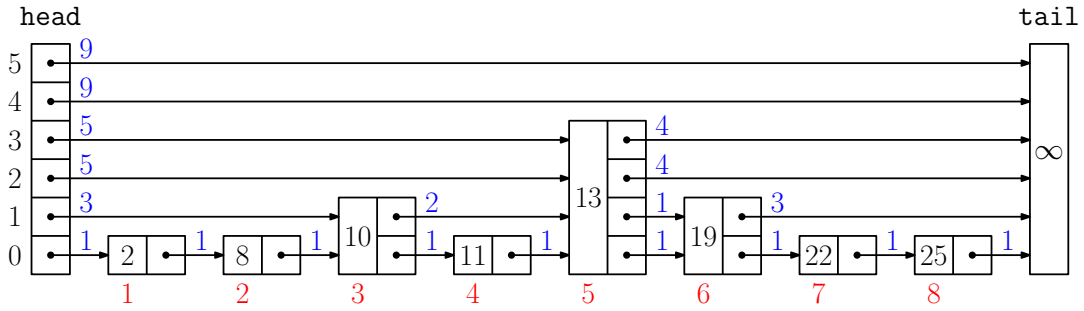

Figure 3: Skip list with span counts (labeled on each edge in blue).

(a) Present pseudo-code for a function `int countSmaller(Key x)`, which returns a count of the number of nodes in the entire skip list whose key values are strictly smaller than `x`. For example, in Fig. 3, the call `countSmaller(22)` would return 6, since there are six items that are smaller than 22 (namely, 2, 8, 10, 11, 13, and 19).

Your procedure should run in time expected-case time $O(\log n)$ (over all random choices). Briefly explain how your function works.

(b) Present pseudo-code for a function `Value getMinK(int k)`, which returns the value associated with the kth smallest key in the entire skip list. For example, in Fig. 3, the call `getMinK(5)` would return 13, since 13 is the fifth smallest element of the skip list. You may assume that $1 \leq k \leq n$, where n is the total number of nodes in the skip list.

Your procedure should run in time expected-case time $O(\log n)$ (over all random choices). Briefly explain how your function works.

Problem 6. It is easy to see that, if you splay twice on the same key in a splay tree (`splay(x); splay(x)`), the tree's structure does not change as a result of the second call.

Is this true when we alternate between two keys? Let T_0 be an arbitrary splay tree, and let x and y be two keys that appear within T_0. Let:

- T_1 be the result of applying `splay(x); splay(y)` to T_0.
- T_2 be the result of applying `splay(x); splay(y); splay(x); splay(y)` to T_0.

Question: Irrespective of the initial tree T_0 and the choice of x and y, is $T_1 = T_2$? (That is, are the two trees structurally identical?) Either state this as a theorem and prove it or provide a counterexample, by giving the tree T_0 and two keys x and y for which this fails.

Problem 7. Consider the following possible node structure for 2-3 trees (that is, a B-tree of order $m = 3$), 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.) 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

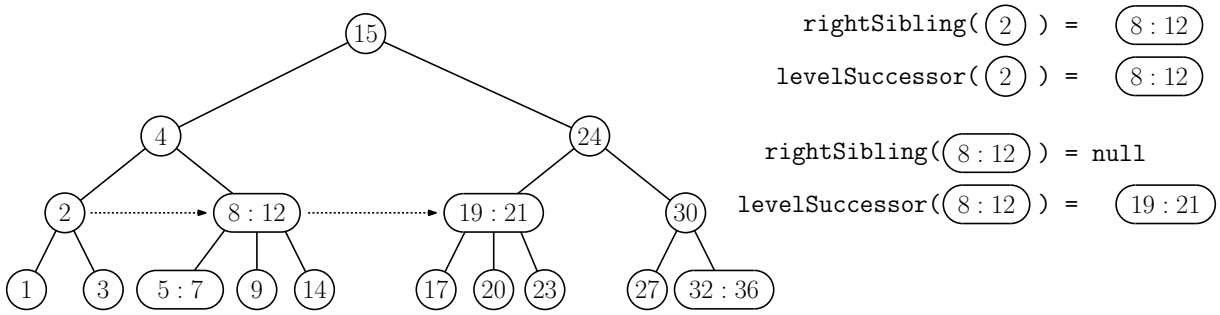

Figure 4: Sibling and level successor in a 2-3 tree.

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 8. Given a set P of n points in the real plane, a *partial-range max query* is given two x-coordinates x_1 and x_2, and the problem is to find the point $p \in P$ that lies in the vertical strip bounded by x_1 and x_2 (that is, $x_1 \leq p.x \leq x_2$) and has the maximum y-coordinate (see Fig. 5).

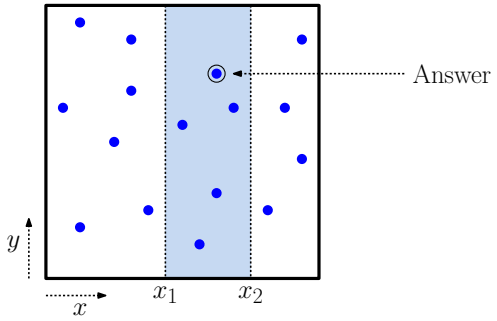

Figure 5: Partial-range max query.

Present pseudo-code for an efficient algorithm to solve partial-range max queries, assuming that the points are stored in a point kd-tree. You may make use of any primitive operations on points and rectangles (but please explain them). **Hint:** A possible signature for your helper function might be

```
Point partialMax(double x1, double x2, KDNode p, Rectangle cell, Point best)
```

Assuming the tree is balanced and the splitting dimension alternates between x and y, show that your algorithm runs in time $O(\sqrt{n})$.

Problem 9. In class we showed that for a balanced kd-tree with n points in the real plane (that is, in 2-dimensional space), any *axis-parallel line* intersects at most $O(\sqrt{n})$ cells of the tree.

The purpose of this problem is to show that does not apply to lines that are not axis-parallel. Show that for every n, there exists a set of points P in the real plane, a kd-tree of height $O(\log n)$ storing the points of P, and a line ℓ, such that *every* cell of the kd-tree intersects this line.

Problem 10. In this problem we will see how to use kd-trees to answer a common geometric query, called *ray shooting*. You are given a collection of vertical line segments in 2D space, each starts at the x-axis and goes up to a point in the positive quadrant. Let $P = \{p_1, \ldots, p_n\}$ denote the upper endpoints of these segments (see Fig. 6). You may assume that both the x- and y-coordinates of all the points of P are strictly positive real numbers.

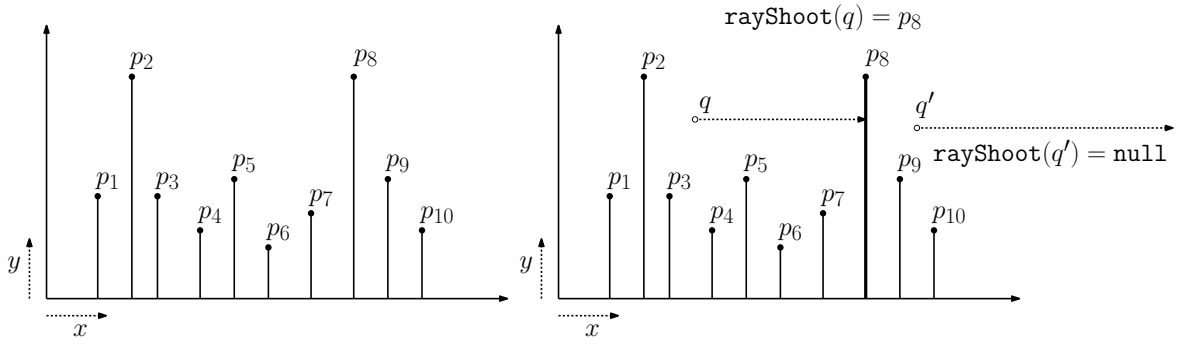

Figure 6: Ray shooting in a kd-tree.

Given a point q, we shoot a horizontal ray emanating from q to the right. This ray travels until it hits one of these segments (or perhaps misses them all). For example, in the figure above, the ray shot from q hits the segment with upper endpoint p_8. The ray shot from q' hits nothing.

In this problem we will show how to answer such queries using a standard point kd-tree for the point set P. A query is given the point $q = (q_x, q_y)$, and it returns the upper endpoint $p_i \in P$ of the segment the ray first hits, or `null` if the ray misses all the segments.

Suppose you are given a kd-tree of height $O(\log n)$ storing the points of P. (It does *not* store the segments, just the points.) Present pseudo-code for an efficient algorithm, `rayShoot(q)`, which returns an answer to the horizontal ray-shooting query (see the figure above, right).

You may assume the kd-tree structure given in class, where each node stores a point `p.point`, a cutting dimension `p.cutDim`, and left and right child pointers `p.left` and `p.right`, respectively. You may make use of any primitive operations on points and rectangles (but please explain them). You may assume that there are no duplicate coordinate values among the points of P or the query point.

Hint: You might wonder how to store segments in a kd-tree. It turns out that to answer this query you do not need to store segments, just points. The function `rayShoot(q)` will invoke a recursive helper function. Here is a suggested form, which you are *not* required to use:

```
Point rayShoot(Point2D q, KDNode p, Rectangle cell, Point best),
```

Be sure to indicate how `rayShoot(q)` makes its initial call to the helper function.

Problem 11. (**Note:** This problem may not be applicable, depending on whether we discuss deletion in hash tables before the exam.) In class we demonstrated a simple idea for deleting keys from a hash table with open addressing. Namely, whenever a key is deleted, we stored a special value "`deleted`" in this cell of the table. It indicates that this cell contained a deleted key. The cell may be used for future insertions, but unlike "`empty`" cells, when the probe sequence searching for a key encounters such a location, it should continue the search.

Suppose that we are using *linear probing* in our hashing system. Describe an alternative approach, which does not use the "`deleted`" value. Instead it moves the table entries around to fill any holes caused by a deleted items.

In addition to explaining your new method, justify that dictionary operations are still performed correctly. (For example, you have not accidentally moved any key to a cell where it cannot be found!)