Practice Midterm Exam

Here's a practice midterm exam to help you prepare for the upcoming midterm. I've done my best to make the problems here similar in difficulty, style, and structure to those on the actual exam. To save paper, I've eliminated unnecessary whitespace; the actual exam will have much more space for you write your answers.

When designing algorithms and data structures on this exam, you do not have to go through the normal writeup template. To make it easier for us to grade your exams, we recommend doing the following:

- Give a one- or two-sentence overview of the data structure or algorithm.
- Describe the basic operation of the data structure or algorithm.
- Prove your data structure or algorithm is correct.

The exam is closed-book. Although initially we planned for the exam to be closed-note, **you may have one double-sided sheet of notes** with you as you take the exam.

The exam covers material up through and possibly including van Emde Boas trees, but will be focused on material covered on the problem sets.

Problem One: Balanced Trees (6 Points)

i. Describe how to augment each node in a red/black tree with two pointers, *next* and *prev*, that point to the inorder predecessor and successor of that node, so that insertions and deletions still run in time O(log n) each. This makes it possible to build an iterator over the red/black tree that can navigate forwards and backwards in worst-case O(1) time.

ii. Let T be a red/black tree and let $k_1 \le k_2$ be two arbitrary values. Give an O(log $n + z$)-time algorithm for deleting all keys from T that are between k_1 and k_2, where z is the total number of elements deleted this way. You can assume that this runtime includes the time required to deallocate the memory for the z removed elements.

Problem Two: Binomial and Fibonacci Heaps (3 Points)

It's possible to implement ***decrease-key*** in an eager binomial heap in time O(log n) by repeatedly swapping that node with its parent until it either reaches the root or becomes less than or equal to its parent.

i. Consider the analogous implementation of ***increase-key*** in an eager binomial heap. What is the runtime of this operation?

ii. Explain how to implement ***increase-key*** in amortized time O(log n) in a Fibonacci heap.

Problem Three: Splay Trees (6 Points)

Let $S = \{ x_1, x_2, \ldots, x_n \}$ be a set of keys in a splay tree where $x_1 < x_2 < \ldots < x_n$. Prove that if you perform a sequence of lookups of $x_1, x_2, \ldots, x_i$, in that order, then the resulting splay tree will have the following structure:

- The root will be x_i.
- The right subtree consists of an arbitrary key containing all keys greater than x_i.
- The left subtree will be a degenerate tree consisting of a left spine holding keys $x_1, \ldots, x_{i-1}$.

(Hint: Use induction and think about the possible cases for the last step of splaying x_i up to the root.)

Problem Four: Aho-Corasick String Matching (3 Points)

Suppose that you have a set of pattern strings $P_1, \ldots, P_k$ of total length n where no pattern string is a substring of any other. Building an Aho-Corasick automaton for these strings takes time $O(n)$.

Prove that the time required to find all matches of these patterns in a string of length m using the matching automaton is $O(m)$.

Problem Five: Suffix Trees and Suffix Arrays (6 Points)

Suppose that you have a pattern string P consisting of a mixture of characters and *wildcard symbols*, denoted ⋆. The pattern string P matches a string T if there is some way to align P inside of T such that all non-wildcard characters of P match the corresponding characters in T. The wildcard symbol is allowed to match any character.

For example, given the strings

T = `alphabetagammadelta` P = `hab⋆t`

P matches at the following spot in T:

```
alphabetagammadelta
   hab⋆t
```

Design an $O(m + n + km)$-time algorithm for determining whether a pattern string P containing k wildcards matches anywhere in T. As a hint, start with the algorithm you developed in Problem Set Six for k-approximate matching and make appropriate modifications.

Problem Six: Hashing and Sketching (6 Points)

In the count sketch data structure, each row consists of an array w of counters. We then choose two hash functions $h : \mathscr{U} \rightarrow [w]$ and $s : \mathscr{U} \rightarrow \{+1, -1\}$, where h sends each element into a slot in the table and s determines whether we increment or decrement the counter when processing the element. To ***increment***(x), we add $s(x)$ to `count`[$h(x)$]. To ***estimate***(x), we return `count`[$h(x)$] $\cdot$ $s(x)$.

You can interpret the hash function s as assigning a *direction* to each element $x \in \mathscr{U}$, where +1 means "up" and -1 means "down." The ***estimate***(x) procedure then works by returning the net number of steps in the direction indicated by $s(x)$ the counter at position $h(x)$ has taken.

We can generalize this so that each counter is a 2D point as follows. Change s so that it now maps from $\mathscr{U}$ to the set {up, down, left, right} and replace each counter in the array `count` with a point in 2D space. To ***increment***(x), we compute $h(x)$ to determine which slot x belongs in, then adjust the point in that slot by moving it one step in the direction given by s(x). To ***estimate***(x), we compute $h(x)$ and look at the point it corresponds to. We then return the net number of steps in the direction given by $s(x)$ that point has taken from the origin. For example, if x is assigned the direction "left," then we return the net number of steps to the left of the origin that the corresponding point has taken.

Compare this modified version of the count sketch to the original version of the count sketch in terms of time, space, accuracy, and confidence. Justify your answer.