CMSC 420: Spring 2023

CMSC 420 (0201) - Final Exam

This exam is closed-book and closed-notes. You may use three sheets of notes (front and back). Write all answers on the exam paper. You may use any algorithms or results given in class. If you have a question, either raise your hand or come to the front of class. Total point value is 120 points. Good luck!

Problem 1. (30 points) Short answer questions. Unless requested, explanations are not required but may be given to help with partial credit.

(a) (2 points) You have an inorder-threaded binary tree with $n \geq 1$ nodes. As a function of n, how many threads are there? (Include the null threads in the leftmost and righmost nodes.)

(b) (2 points) You are given a leftist max-heap containing $n \geq 3$ keys (priorities), where all the keys are distinct. **True or False**: The node with the smallest key must be a leaf of the tree.

(c) (4 points) What are the min and max number of keys in a 2-3 tree of height 2? (Remember, *height* is the number of edges from the root to the deepest leaf. Note that we are asking about *keys*, not *nodes*.)

(d) (4 points) You have just performed an insertion into a 2-3 tree of height $h \geq 0$. As a function of h, what are the minimumm and maximum number of splits that might be needed as a result?

(e) (6 points) You have just inserted a large number n of keys into a scapegoat tree (no deletions). The next operation may be a find, insert, or delete. Which of the following statements is true? (Select all that apply.)

(1) The next operation is *find*, the worst-case (not expected-case) time is $O(\log n)$

(2) The next operation is *insert*, the worst-case (not expected-case) time is $O(\log n)$

(3) The next operation is *delete*, the worst-case (not expected-case) time is $O(\log n)$

(f) (4 points) In the memory management system described in class, which of the following elements are stored in each *allocated block*? (Select all that apply)

(1) The header, containing the `inUse` bit, `prevInUse` bit, and `size`

(2) Links to the `prev` and `next` elements in the available-block list

(3) The `size2` field at the end of the block

(4) The `time` field indicating when the block was most recently accessed

(g) (3 points) You have a splay tree with n keys. You perform a splay on an arbitrary key x and then immediately following this, you perform a splay on the very next key in ascending order. Which of the following is most accurate? (Select one.)

(1) The second splay takes $O(1)$ time in the worst case.

(2) The second splay takes $O(\log n)$ time in the worst case.

(3) One of the two splays takes $O(\log n)$ time, but we cannot predict which.

(4) The second splay takes $O(n)$ time in the worst case.

(h) (5 points) You have a Bloom filter that stores a set with a large number n of elements. The table has m entries and there are k hash functions. Which of the following is true? (Select all that apply.)

(1) The worst-case insertion time is (most accurately) $O(1)$.

(2) The worst-case insertion time is (most accurately) $O(m)$.

(3) The worst-case insertion time is (most accurately) $O(k)$.

(4) Assuming k is fixed, increasing m (generally) decreases the chance of a false positive.

(5) Assuming m is fixed, increasing k (generally) decreases the chance of a false positive.

Problem 2. (10 points) Recall that the *depth* of a node in a binary tree is the number of edges between this node and the root. In any binary tree, there are at most 2^d nodes at depth d. Given any AVL tree T and an integer $d \geq 0$, we say that T is *full at depth d* if it has the maximum possible number of nodes (namely, 2^d) at depth d.

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 in Fig. 1 has height 4, and it is full at levels 0, 1, and 2, but it is not full at levels 3 and 4.)

Hint: Prove this by induction on the height of the tree.

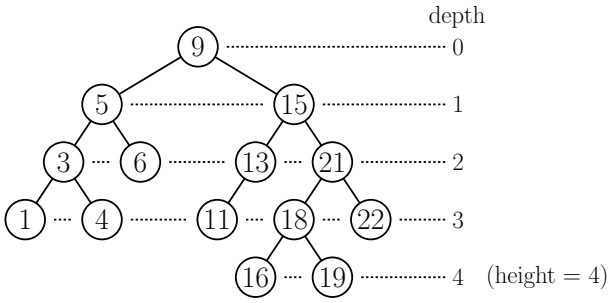

Figure 1: Full at depth.

Problem 3. (10 points) Consider the B-trees of order $m = 4$ shown in Fig. 2. When answering the questions below, assume the following conventions. Key rotation/adoption (when possible) takes precedence over splitting/merging. Second, when splitting a node, use the convention given in the lecture, that the left node has $\lceil m/2 \rceil - 1$ keys and the right node has $\lfloor m/2 \rfloor$ keys. (**Hint:** To save time, you need only redraw the portion of the tree that is affected by the operations.)

(a) (5 points) Show the final tree that results after inserting the key `13`.

(b) (5 points) Starting with the *original tree*, show the final tree that results after deleting the key `42`.

Problem 4. (15 points) In this problem we will build a suffix tree for $S =$ `"ababab$"`.

(a) (5 points) List the seven substring identifiers of S (from the last to the first).

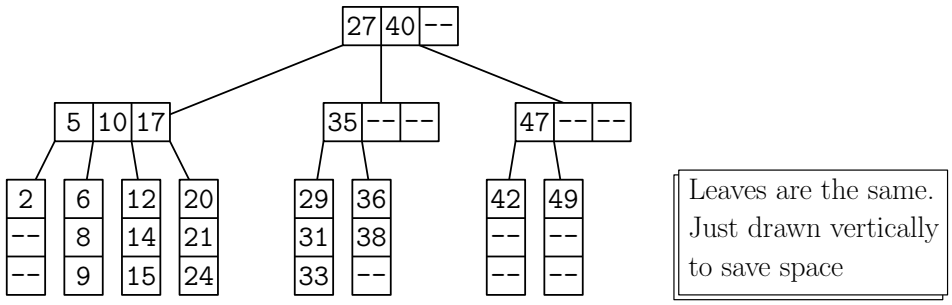

Figure 2: B-Tree Operations.

(b) (10 points) Draw the suffix tree for S. Use the same form as in the lectures. When drawing your tree, order the children of each node from left to right in the order `a` < `b` < `$`. (**Hint:** Be sure to compress paths wherever possible.)

Problem 5. (15 points) You are given a collection of vertical line segments, where the lower endpoints all lie on the x-axis and the upper endpoints lie in the first quadrant. This is presented to you as a set $P = \{p_1, \ldots, p_n\}$ of the upper endpoints of these segments, where each point `p` has coordinates `(p.x,p.y)`.

In a *segment-hitting query*, we are given a horizontal ray that is shot to the left of a query point $q = (q.x, q.y)$, and the problem is to count the number of segments that are hit by this ray (see Fig. 3).

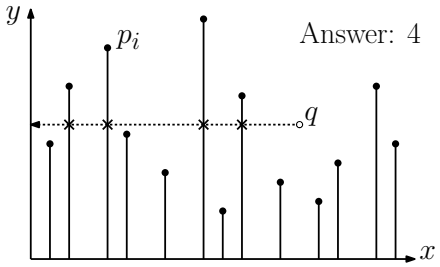

Figure 3: Segment-hitting queries.

(a) (3 points) You are asked to design an efficient kd-tree-based data structure for answering these queries. Explain what information is stored in the nodes of the tree, and what assumptions you make about the tree's structure (e.g., the choice of cutting dimensions and cutting values, tree balance, etc.)

(b) (8 points) Present pseudocode for an efficient function that answers segment hitting queries for a given query point `(q.x q.y)`. **Hint:** Here is a suggested helper:

```
int segHit(Point q, KDNode p, Rect cell)
```

where `q` is the query point, `p` is the current node of the kd-tree, `cell` is the rectangular cell associated with `p`.

(c) (4 points) What is your algorithm's running time? Explain briefly. (You may use facts proven in class about kd-trees.)

Problem 6. (20 points) Consider strings over the alphabet $\Sigma = \{0, 1, 2\}$. We say that a string is *monotone* if its characters (in left to right order) are monotonically non-decreasing. That is, given a string "$s_0 s_1 \ldots s_{k-1}$", we have $s_i \geq s_{i-1}$, for $1 \leq i \leq k-1$.

For example, the strings `"012"`, `"02"`, and `"22222"` are all monotone because, when read from left to right, the digit values never decrease. However, the string `"00102"` is not (because the fourth character `"0"` is smaller than its predecessor `"1"`).

The objective of this problem is, given a trie containing a set of strings, count the number monotone strings in the set. Throughout, you may assume that the set of strings is *prefix free*, that is, no string is a prefix of any other.

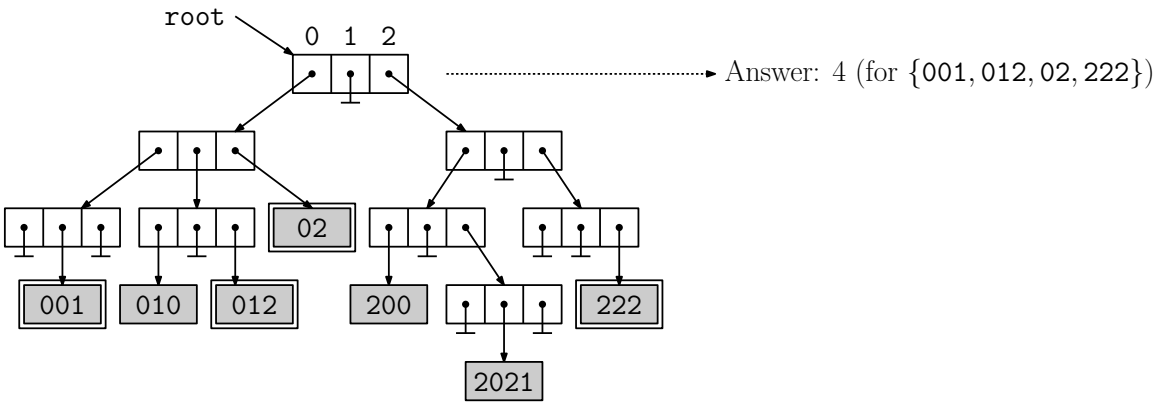

Figure 4: Monotone Strings: Standard Trie.

(a) (10 points) Assume that the trie is represented in standard (uncompressed) form as an extended tree. (See Fig. 4.) Each internal node `p` stores an array `p.child[0..2]`. (For example `p.child[2]` points to the subtree of strings whose next character is `"2"`.) External nodes store the final string as `p.key`. To distinguish between node types, each node stores a boolean, `p.isExternal`, which is `true` for external nodes and `false` otherwise. Let `root` denote the root of the tree.

Present pseudocode for an efficient function, `countMonotone(root)`, which returns a count of the number of monotone strings in the trie.

Hint: You may create any helper functions you need. You may use any standard string utility functions. Your answer may be expressed either iteratively or recursively, your choice.)

(b) (10 points) Repeat part (a), but now assuming that the trie is represented using a de la Briandais trie (see Fig. 5). Each internal node `p` stores a single character as the `p.key` and its first-child and next-sibling pointers, `p.firstChild` and `p.nextSibling`, respectively. External nodes store the final string as `p.key`. To distinguish between node types, each node stores a boolean, `p.isExternal`, which is `true` for external nodes and `false` otherwise.

Problem 7. (20 points) In this problem, we will consider a variant of double hashing, called *priority hashing* (or "prashing"). As with double hashing, we are given a hash table with m

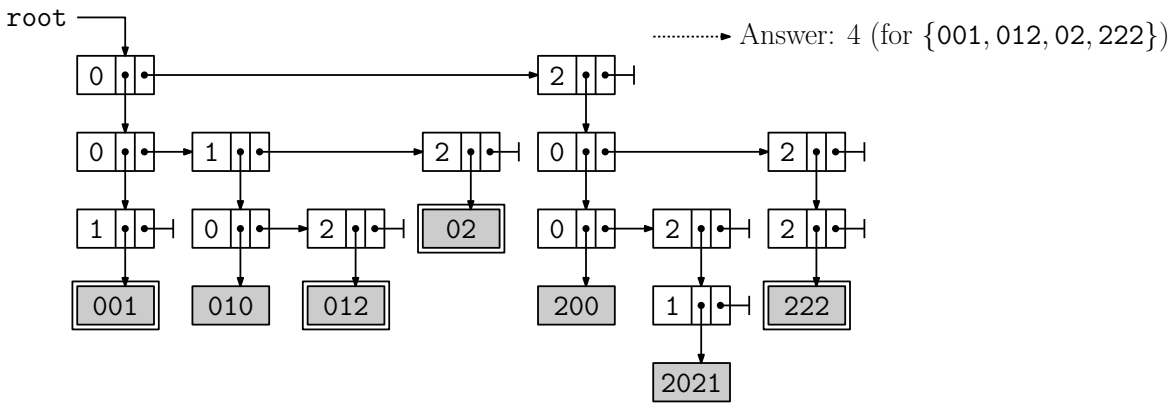

Figure 5: Monotone Strings: de la Briandais Trie.

entries, denoted $T[0..m-1]$. Each key x is associated with two hash functions, $h(x)$ and $g(x)$, and the probe sequence for key x consists of the indices

$$h(x), \quad h(x)+g(x), \quad h(x)+2g(x), \quad h(x)+3g(x), \quad \ldots$$

where all indices are taken modulo m. This is exactly the same as standard double hashing.

The new twist is that each key x is also associated with a numeric *priority function*, $p(x)$. Whenever a collision occurs, we compare the priorities of the two keys. The key with the higher priority gets to stay, and the other is "bumped", that is, it must move to the next position in its probe sequence. To simplify matter, assume that *no two keys have the same priority.* Here is the insert function:

```
void insert(Key x) {                    // insert key x in table T[0..m-1]
    insertHelper(x, h(x))
}

void insertHelper(Key x, int i) {       // try to insert x at T[i]
    i = i % m                           // wrap around
    if (T[i] == null) {                 // empty?
        T[i] = x                        // yes, insert here
    } else {
        Key y = T[i]                    // collision with y
        if (p(x) < p(y)) {              // x has lower priority?
            insertHelper(x, i + g(x))   // bump x to its next position
        } else {                        // x has higher priority?
            T[i] = x                    // x takes y's spot
            insertHelper(y, i + g(y))   // bump y to its next position
        }
    }
}
```

(a) (10 points) Assume that we perform insertions, but *no deletions*. Present pseudocode for an efficient function `find(Key x)`, which searches for a (non-null) key x in the table. If it is in the table, return `true`, and otherwise return `false`.

(b) (5 points) Assume that for all keys x, the values of $g(x)$ and m are relatively prime (that is, they share no common factors other than 1).

True or False: If the hash table has at least one empty entry, the insert function will successfully insert x in the table (as opposed to looping infinitely).

Justify your answer. If true, give a clear argument as to why. If false, give a counterexample showing that it can go into an infinite loop.

(c) (5 points) Repeat part (b), but this time $g(x)$ and m need *not* be relatively prime.

Justify your answer. If true, give a clear argument as to why. If false, give a counterexample showing that it can go into an infinite loop.