CMSC 420: Spring 2023

Practice Problems for the Final Exam

Our final exam will be held on **Mon, May 15, 4-6pm** in **IRB 0324, the Antonov Auditorium**. It will be closed-book, closed-notes, but you will be allowed three sheets of notes, front and back.

Disclaimer: The exam will be comprehensive, emphasizing material in the latter half of the semester. 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. Since the exam is comprehensive, please look back over the previous homework assignments, the two midterm exams, and the practice problems for both midterms. You should expect at least one problem that involves tracing through an algorithm or construction given in class.

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

(a) You have an inorder-threaded binary tree with n nodes. Let u be an arbitrary non-leaf node in this tree. **True or False**: There must be at least one thread that points into u.

(b) Let T be an extended binary search tree (that is, one having internal and external nodes). You visit the nodes of T according to one of the standard traversals (preorder, postorder, or inorder). Which of the following statements is necessarily true? (Select all that apply.)

(1) *Preorder traversal*: All the internal nodes appear *before* any of the external nodes
(2) *Inorder traversal*: Internal and external nodes *alternate* with each other
(3) *Postorder traversal*: The *first* node visited is an *external node*
(4) *Postorder traversal*: The *last* node visited is an *internal node*

(c) Given a binary max-heap with n entries ($n \geq 3$), you want to return the *third largest* item in the heap (without modifying its contents). What is the minimum number of heap entries that you might need to inspect to be certain that the third largest item is among them?

(d) When we delete an entry from a simple (unbalanced) binary search tree, we sometimes need to find a replacement key. Suppose that `p` is the node containing the deleted key. Which of the following statements are true? (Select all that apply.)

(1) A replacement is needed whenever `p` is the root
(2) A replacement is needed whenever `p` is a leaf
(3) A replacement is needed whenever `p` has two non-null children
(4) It is best to take the replacement exclusively from `p`'s right subtree
(5) At most one replacement is needed for each deletion operation

(e) You have an AVL tree containing n keys, and you insert a new key. As a function of n, what is the maximum number of rotations that might be needed as part of this operation? (A double rotation is counted as two rotations.) Explain briefly.

(f) Repeat the previous question in the case of deletion. (Give your answer as an asymptotic function of n.)

(g) What are the min and max number of nodes in a 2-3 tree of height 2? (Remember, *height* is the number of edges from the root to the deepest leaf.)

(h) You have just performed a deletion from a 2-3 tree of height h. As a function of h, what is the maximum number of key-rotations (adoptions) that might be needed as a result?

(i) The AA-tree data structure has the following constraint: "*Each red node can arise only as the right child of a black node.*" Which of the two restructuring operations (`skew` and `split`) enforces this condition?

(j) You have just inserted a key into an AA tree having L levels. As a function of L, what is the maximum number of skew operations that might be needed as a result? (Here we are only counting skew operations that have an effect on the structure, in the sense that a rotation is performed.)

(k) Splay trees are known to support efficient finger search queries. What is a "finger search query"?

(l) In class, we mentioned that when using double hashing, it is important that the second hash function $g(x)$ should not share any common divisors with the table size m. What might go wrong if this were not the case?

(m) Hashing is widely regarded as the fastest of all data structures for basic dictionary operations (insert, delete, find). Give an example of an operation that a tree-based search structure can perform *more efficiently* than a hashing-based data structure, and explain briefly.

(n) In the (unstructured) memory management system discussed in class, each available block of memory stored the size of the block both at the beginning of the block (which we called `size`) and at the end of the block (which we called `size2`). Why did we store the block size at both ends?

(o) You build a suffix tree for a string having m characters (including the `"$"` at the end). What is the maximum number of internal nodes in the tree?

(p) You build a Bloom filter for a set X that uses k hash functions. Each hash function can be computed in $O(1)$ time. You query the data structure on an element x. Which of the following hold? (Select all that apply.)

(1) If $x \in X$, the data structure will correctly report this
(2) If $x \in X$, the data structure may or may not correctly report this
(3) If $x \notin X$, the data structure will correctly report this
(4) If $x \notin X$, the data structure may or may not correctly report this
(5) The query is answered in $O(k)$ worst-case time
(6) The query is answered in $O(k)$ expected time (but it might take longer)

Problem 2. 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            // key
    Node left, right   // children
    int size           // number of entries in this subtree
}
```

(a) Present pseudocode for a function `printMaxK(int k)`, which is given $0 \le k \le n$, and prints the values of the k largest keys in the binary search tree (see, for example, Fig. 1).

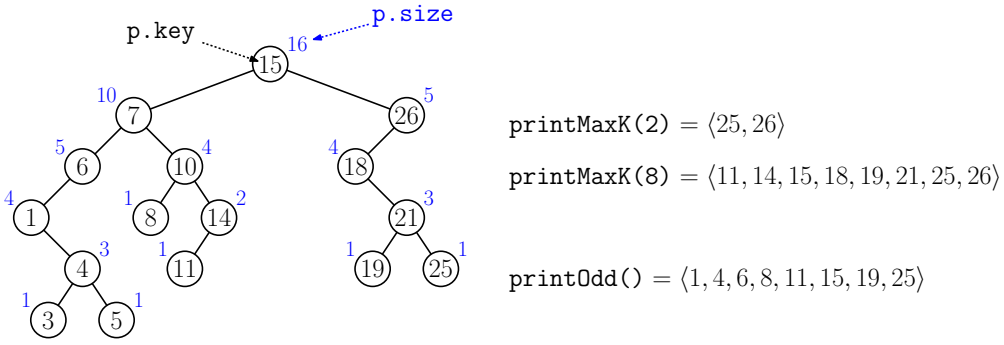

Figure 1: The functions printMaxK and printOdd.

You should do this in a single pass by traversing the relevant portion of the tree. It would be considered cheating to store all the elements of in a list, and then just print the last k entries of the list.

For fullest credit, the keys should be printed in *ascending order*, and your algorithm should run in time $O(k + \log n)$ (see part (b) below). Briefly explain your algorithm.

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`.

(b) Derive the running time of your algorithm in (a).

(c) Give 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 $\langle x_1, x_3, x_5, \ldots \rangle$ (see Fig. 1).

Again, you should do this in a single pass by traversing the tree. (For example, it would be considered cheating to traverse the tree and construct a list with all the entries, and then only print the odd entries of your list.) Your function should run in time $O(n)$. Briefly explain your algorithm.

Problem 3. In this problem you will write a program to check the validity of an AVL tree. The node structure is given below. All members are public.

```
class AVLNode {
    public int key                 // key
    public int height              // height of this subtree
    public AVLNode left, right     // left and right children
}
```

In order to be valid, every node `p` of the tree must satisfy the following conditions (see Fig. 2):

- `p.height` is correct given the heights of its children. (Recall: `height(null) == -1`.)
- The absolute height difference between `p`'s left and right subtrees is at most 1
- An inorder traversal of the tree encounters keys in *strictly* ascending order

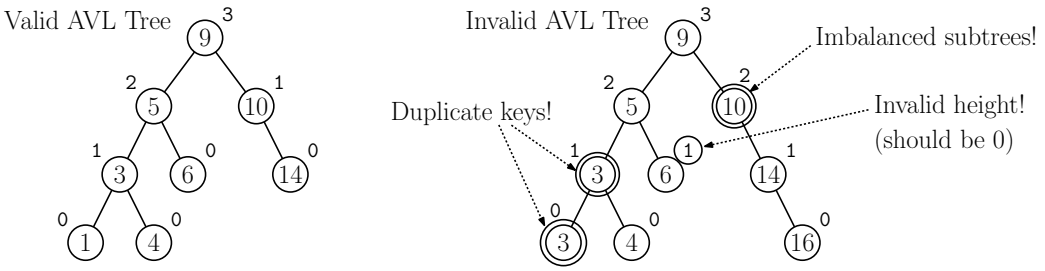

Figure 2: Valid and invalid AVL trees.

Problem 4. Assume that you are given a kd-tree storing a set P of n points in $\mathbb{R}^2$ that satisfies the *standard assumptions*. (That is, the cutting dimension alternates between x and y, subtrees are balanced, and the tree stores a bounding box `bbox` containing all the points of P.)

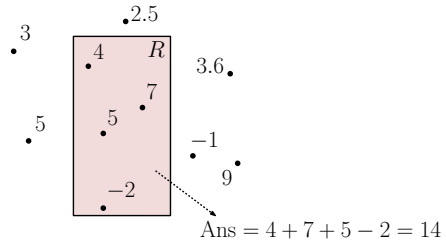

Figure 3: Weighted range query.

(a) In a standard range-counting query, we want to count the number of points in the range. Suppose that each point $p_i \in P$ has an associated real-valued weight w_i. In a *weighted orthogonal range query*, we are given a query rectangle R, given by its lower-left corner r_{lo} and upper-right corner r_{hi}, and the answer is the sum of the weights of the points that lie within R (see Fig. 3(b)). If there are no points in the range, the answer is 0.

Explain how to modify the kd-tree (by adding additional fields to the nodes, if you like) so that weighted orthogonal range queries can be answered efficiently. Based on your modified data structure, present an efficient algorithm in pseudo-code for answering these queries and explain. (For full credit, your algorithm should run in $O(\sqrt{n})$ time).

You may handle the edge cases (e.g., points lying on the boundary of R) however you like. **Hint:** You may use whatever helper function(s) you like, but I would suggest using:

```
double weightedRange(Rectangle R, KDNode p, Rectangle cell)
```

where `p` is the current node in the kd-tree, `cell` is the associated cell.

(b) Briefly analyze the running time of your algorithm, assuming that the tree is balanced. (You may apply/modify results proved in class.)

Problem 5. Assume that you are given a kd-tree storing a set P of n points in $\mathbb{R}^2$ that satisfies the *standard assumptions*. (That is, the cutting dimension alternates between x and y, subtrees are balanced, and the tree stores a bounding box `bbox` containing all the points of P.)

In a *fixed-radius nearest neighbor query*, we are given a point $q \in \mathbb{R}^d$ and a radius $r > 0$. Consider a circular disk centered at q whose radius is r. If no points of P lie within this disk, the answer to the query is `null`. Otherwise, it returns the point of P within the disk that is closest to q (see Fig. 4). Present (in pseudo-code) an efficient kd-tree algorithm that answers such a query.

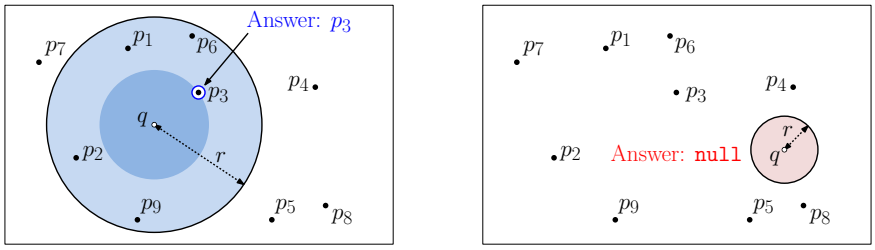

Figure 4: Fixed-radius nearest-neighbor query.

Hint: You may use whatever helper function(s) you like, but I would suggest using:

```
Point frnn(Point q, double r, KDNode p, Rectangle cell, Point best)
```

where `p` is the current node in the kd-tree, `cell` is the associated cell, and `best` is the best point seen so far.

You *do not* need to analyze your algorithm's running time, but explain it briefly. Your algorithm should not waste time visiting nodes that cannot possibly contribute to the answer.

Problem 6. 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 7. Suppose that you are given a treap data structure storing n keys. The node structure is shown in Fig. 5. You may assume that *all keys and all priorities are distinct.*

(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. 5 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"`.

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

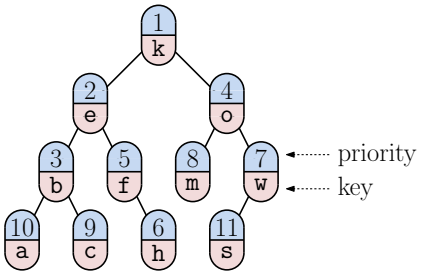

Figure 5: Treap node structure and an example.

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

Problem 8. In this problem we will build a suffix tree for the string $S =$ `baabaababababaa$`.

(a) 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.

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

(c) 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.

Problem 9. Suppose you have a large span of memory, which starts at some address `start` and ends at address `end-1` (see Fig. 6). (The variables `start` and `end` are generic pointers of type `void*`.) As the dynamic memory allocation method of Lecture 15, this span is subdivided into blocks. The block starting at address `p` is associated with the following information:

- `p.inUse` is 1 if this block is in-use (allocated) and 0 otherwise (available)
- `p.prevInUse` is 1 if the block immediately preceeding this block in memory is in-use. (It should be 1 for the first block.)
- `p.size` is the number of words in this block (including all header fields)
- `p.size2` each available block has a copy of the size stored in its last word, which is located at address `p + p.size - 1`.

(For this problem, we will ignore the available-list pointers `p.prev` and `p.next`.)

In class, we said that in real memory-allocation systems, blocks cannot be moved, because they may contain pointers. Suppose, however, that the blocks are movable. Present pseudo-code for a function that compacts memory by copying all the allocated blocks to a single contiguous span of blocks at the start of the memory span (see Fig. 6). Your function `compress(void* start, void* end)` should return a pointer to the head of the available block at the end. Following these blocks is a single available block that covers the rest of the memory's span.

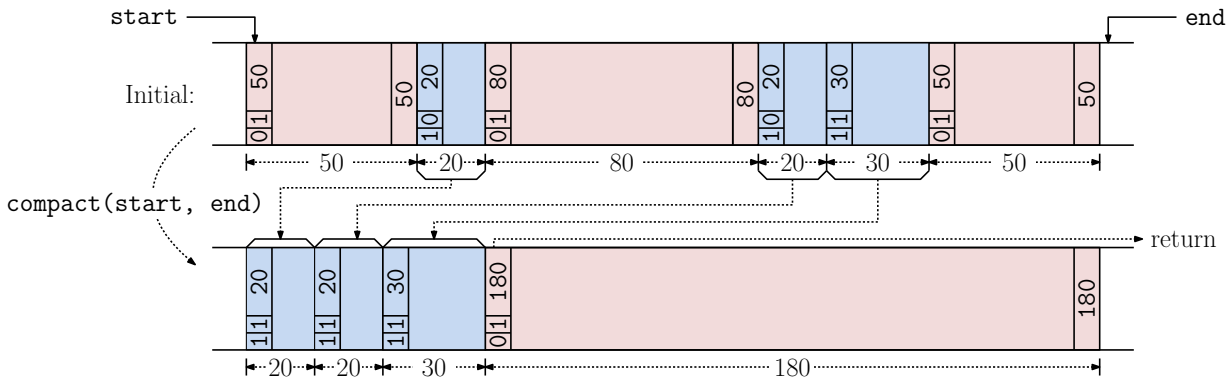

Figure 6: Memory compactor.

To help copy blocks of memory around, you may assume that you have access to a function `void* memcpy(void* dest, void* source, int num)`, which copies `num` words of memory from the address `source` to the address `dest`.

Problem 10. 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.

(a) 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.

(b) 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.

(c) 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 11. You are designing an expandable hash table using open addressing. Let m denote the current table size. Initially $m = 4$. Let us make the ideal assumption that each hash operation takes exactly 1 time unit. After each insertion, if the number of entries in the table is greater than or equal to $3m/4$, we expand the table as follows. We allocate a new table of size $4m$, create a new hash function, and rehash all of the elements from the current table into the new table. The time to do this expansion is $3m/4$.

(a) Derive the amortized time to perform an insertion in this hash table (assuming that m is very large). State your amortized running time and explain how you derived it. (For fullest credit, your running time should as tight as possible.)

Hint: The amortized time need not be an integer.

(b) One approach to decrease the amortized time is to modify the table expansion factor, which in this case is 4. In order to reduce the amortized time, should we *increase* or *decrease* this factor? If you make this adjustment, what negative side effect (if any) might you observe regarding the space and time performance of the data structure? Explain briefly.

‘