CMSC 420: Fall 2022

Practice Problems for the Final Exam

Our final exam will be held on **Thu, Dec 15, 8-10am** 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) Let T be 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.)

(i) In a *postorder traversal*, all the external nodes appear in the order *before* any of the internal nodes

(ii) In a *preorder traversal*, all the internal nodes appear in the order *before* any of the external nodes

(iii) In an *inorder traversal*, internal and external node *alternate* with each other

(iv) None of the above is true

(b) 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.)

(i) A replacement is needed whenever `p` is the root

(ii) A replacement is needed whenever `p` is a leaf

(iii) A replacement is needed whenever `p` has two non-null children

(iv) It is best to take the replacement exclusively from `p`'s right subtree

(v) At most one replacement is needed for each deletion operation

(c) 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.

(d) Repeat (c) in the case of deletion. (Give your answer as an asymptotic function of n.)

(e) 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?

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

(g) 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?

(h) 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.

(i) 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?

(j) Between the classical dynamic storage allocation algorithm (with arbitrary-sized blocks) or the buddy system (with blocks of size power of 2) which is more susceptible to *internal fragmentation*? Explain briefly.

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 $0 \leq k \leq n$, and prints the values of the k largest keys in the binary search tree (see, for example, Fig. 1).

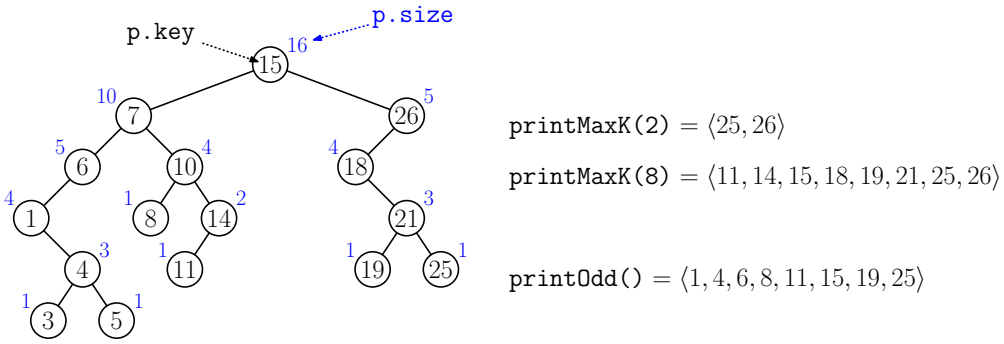

Figure 1: The functions printMaxK and printOdd.

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. Throughout this problem, assume that you are given a kd-tree storing a set P of n points in $\mathbb{R}^2$. Assume the tree 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.) You may also assume that that any geometric computations on primitive objects (distances, disjointness, containment, etc.) can be computed in constant time, without explanation.

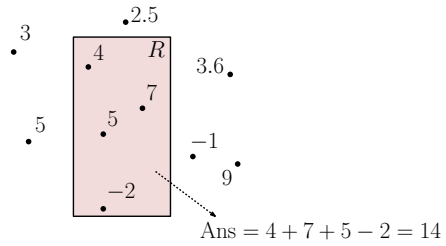

Figure 2: Weighted range query.

(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. 2(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 4. As in the previous problem, assume that you are given a kd-tree storing a set P of n points in $\mathbb{R}^2$ that satisfies the *standard assumptions*. 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. 3). Present (in pseudo-code) an efficient kd-tree algorithm that answers such a query.

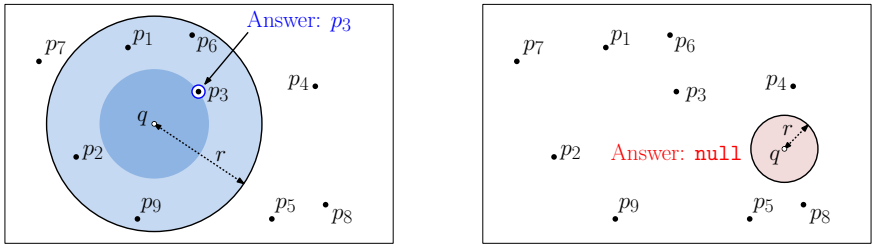

Figure 3: Faxed-radius nearest-neighbor query.

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 5. 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 6. 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.

(a) Assume you are given an n-element point set P in $\mathbb{R}^2$ (see Fig. 4(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. 4(b)). (As an application, imagine you are searching for the k highest rated restaurants in a rectangular region of some city.)

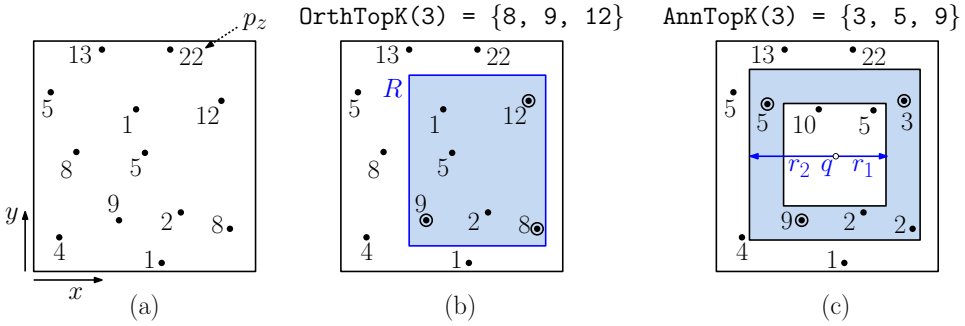

Figure 4: Orthogonal top-k queries and annulus top-k queries.

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.

(b) 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. 4(c)).

Problem 7. In this problem, we are given a set L of n horizontal line segments $\overline{s_i t_i}$ in the plane, where $s_i = (x_i^-, y_i)$ and $t_i = (x_i^+, y_i)$ (Fig. 5(a–b)). We want to preprocess them to answer the following queries efficiently:

Segment stabbing queries: Consider a vertical query line segment with x-coordinate q_x, whose lower endpoint has the y-coordinate q_y^-, and whose upper endpoint has y-coordinate q_y^+. *How many of the segments of L does this segment intersect?* (For example, the vertical segment in Fig. 5(c) intersects 5 segments of L.)

Answer the following for the query vertical query (q_x, q_y^-, q_y^+) and horizontal segment $\overline{s_i t_i}$. (**Hint:** To simplify your answer, you may assume that all the coordinates are distinct, so the endpoint of one segment will never lie in the interior of another.)

(a) Describe a range tree-based data structure for this problem and derive its space bound. (Query processing comes later.)

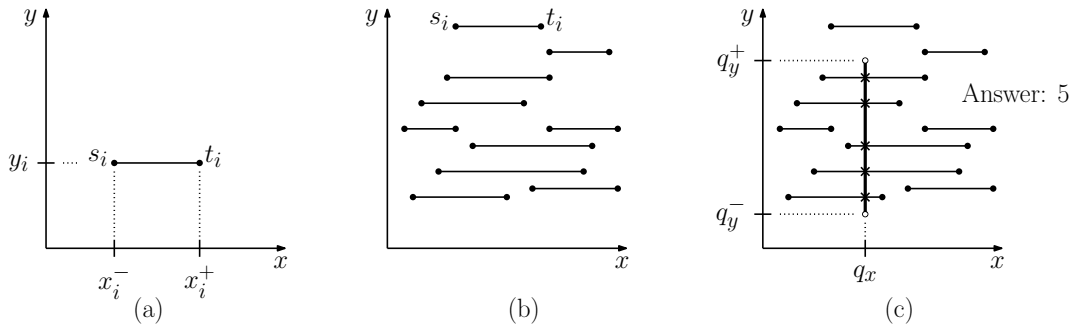

Figure 5: Segment stabbing queries.

(b) Explain how segment-stabbing queries are answered and derive the query time.

Problem 8. 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`.)

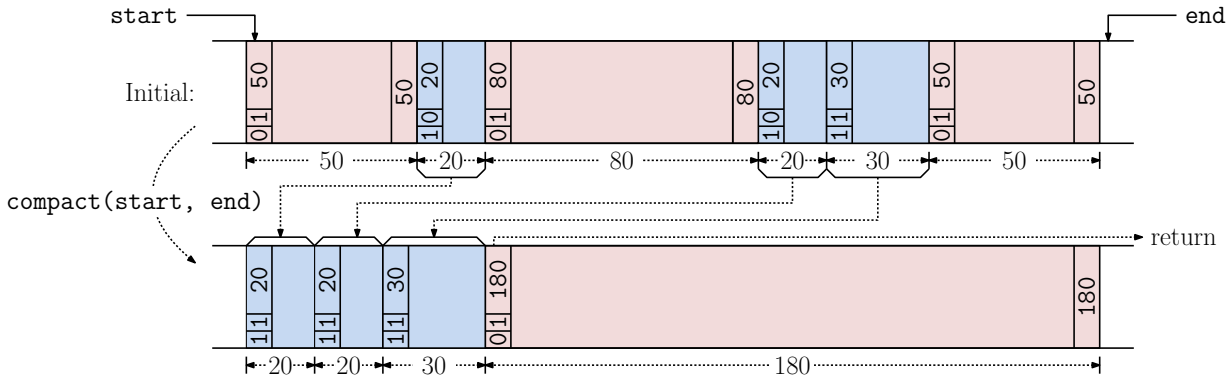

Figure 6: Memory compactor.

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.

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 9. Recall the buddy system of allocating blocks of memory (see Fig. 7). Throughout this problem you may use the following standard bit-wise operators:

`&`	bit-wise "and"	`|`	bit-wise "or"
`^`	bit-wise "exclusive-or"	`~`	bit-wise "complement"
`<<`	left shift (filling with zeros)	`>>`	right shift (filling with zeros)

You may also assume that you have access to a function `bitMask(k)`, which returns a binary number whose k lowest-order bits are all 1's. For example `bitMask(3)` $= 111_2 = 7$.

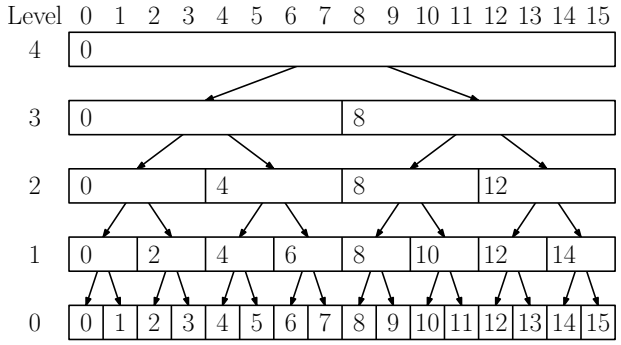

Figure 7: Buddy relatives.

Present a short (one-line) expression for each of the following functions in terms of the above bit-wise functions:

(a) `boolean isValid(int k, int x)`: True if and only if $x \geq 0$ a valid starting address for a buddy block at level $k \geq 0$.

(b) `int sibling(int k, int x)`: Given a valid buddy block of level $k \geq 0$ starting at address x, returns the starting address of its *sibling* (that is, its "buddy").

(c) `int parent(int k, int x)`: Given a valid buddy block of level $k \geq 0$ starting at address x, returns the starting address of its *parent* at level $k + 1$.

(d) `int left(int k, int x)`: Given a valid buddy block of level $k \geq 1$ starting at address x, returns the starting address of its *left child* at level $k - 1$.

(e) `int right(int k, int x)`: Given a valid buddy block of level $k \geq 1$ starting at address x, returns the starting address of its *right child* at level $k - 1$.

For example, given the tree shown in the figure, we have

```
isValid(2, 12) = isValid(2, 01100) = True
isValid(2, 10) = isValid(2, 01010) = False
sibling(2, 12) = sibling(2, 01100) =  8 = 01000
 parent(2, 12) =  parent(2, 01100) =  8 = 01000
   left(2, 12) =    left(2, 01100) = 12 = 01100
  right(2, 12) =   right(2, 01100) = 14 = 01110
```

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.

‘