Final Exam

This exam is closed-book and closed-notes. You may use one sheet of notes (front and back). Write all answers in the exam booklet. 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 150 points. Good luck!

Problem 1. (50 points) Short answer questions. Except where noted, explanations are not required but may be given for partial credit.

(1.1) (5 pts) Given an *extended binary tree* with n *internal nodes*, how many *external nodes* does this tree have?

(1.2) (5 pts) 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.)

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

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

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

(d) None of the above is true

(1.3) (5 pts) 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.

(1.4) (5 pts) Repeat (1.3) in the case of deletion. (You can give your answer as an asymptotic function of n.)

(1.5) (5 pts) You are given a 2-3 tree of height h. As a function of h, what is the *minimum* number of *red nodes* that might appear on any path from a root to a leaf node in corresponding AA tree? What is the *maximum* number?

(1.6) (6 pts) Suppose you know that a very small fraction of the keys in a data structure are to be accessed most of the time, but you do not know which these keys are. Among the data structures we have seen this semester, which would be best for this situation? Explain briefly.

(1.7) (8 pts) You insert the sequence of n keys $\langle x_1, \ldots, x_n \rangle$ into a treap data structure, but something goes wrong, and your random number generator sets the priority of x_i to i (not a random value). The tree that is generated by the treap insertion algorithm is equivalent to which of the following?

(a) An standard (unbalanced) binary search tree

(b) An AVL tree

(c) An AA tree

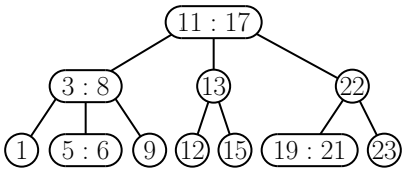
(d) A scapegoat tree

(e) None of these

(1.8) (6 pts) Both the *unbalanced binary search tree* and the *skip list* support dictionary operations in $O(\log n)$ expected-case time and $O(n)$ worst-case time. If you were asked (in a future job, say) to recommend one of these data structures, which of these two options is preferred, or are they essentially equivalent? Briefly justify your answer. (A couple of sentences is sufficient.)

(1.9) (5 pts) 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. (20 points) Consider the 2-3 tree shown in the figure below.



(2.1) (10 points) Show the AA tree corresponding to this tree. (Indicate each red node by having a dashed line coming in from its parent.)

(2.2) (10 points) Show the 2-3 tree that results by inserting the key 7. (Remember that key rotations are *not* performed when inserting into 2-3 trees, only splits. You need only show the final result for full credit, but intermediate results can be given to help with partial credit.)

Problem 3. (20 points) Recall the code block below, which was used to find a key in a hash table `table[0...m-1]` assuming quadratic probing:

```
Value find(Key x) {
    int c = h(x)               // initial probe location
    int i = 0                  // probe offset
    while (table[c].key != empty) && (table[c].key != x) {
        c += 2*(++i) - 1       // next position
        c = c % m              // wrap around
    }
    return table[c].value      // return associated value (or null if empty)
}
```

The purpose of this problem is to explain how this function works.

(3.1) (5 points) Suppose that $h(x) = 0$, $m > 100$, and the while loop above executes at least 5 times. What are the indices of the first five table entries visited by the above algorithm? (For example $0, 1, \ldots$).

(3.2) (5 points) What is the purpose of the line `c = c % m`? (What would go wrong if it were not there?)

(3.3) (10 points) Give a mathematical justification for the assertion that this algorithm visits the indices $h(x) + i^2$, for $i = 0, 1, 2, \ldots$.

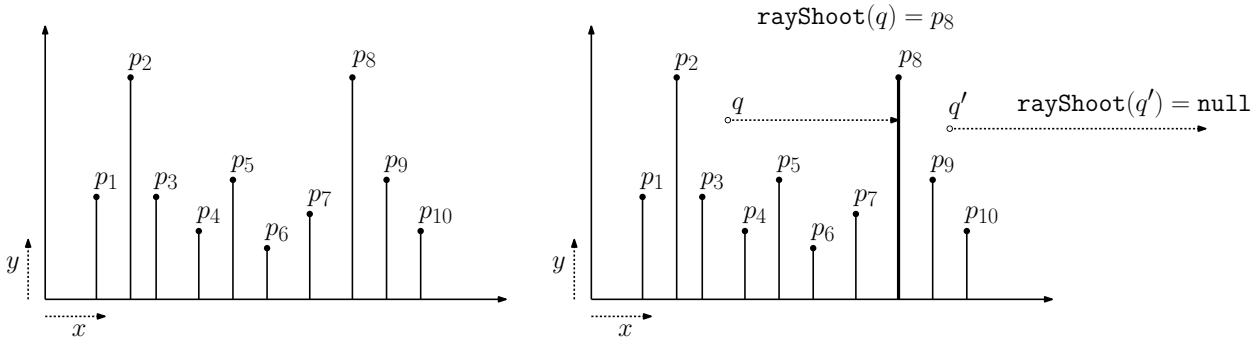
Problem 4. (15 points) You are given a binary search tree where, in addition to the usual fields `p.key`, `p.left`, and `p.right`, each node `p` has a *parent link*, `p.parent`. This points to `p`'s parent, and is `null` if `p` is the root. Given such a tree, present pseudo-code for a function

`Node preorderPredecessor(Node p)`,

which is given a non-null reference `p` to a node of the tree and returns a pointer to `p`'s *preorder predecessor* in the tree (or `null` if `p` has no preorder predecessor). Your function should run in time proportional to the height of the tree. Briefly explain how your function works.

Hint: Test your algorithm in each of the following cases. When `p` is the root of the tree. When `p` is a left child. When `p` is a right child, and its parent has no left child. When `p` is a right child, and its parent has a non-null left child.

Problem 5. (15 points) 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 the figure below, left). You may assume that both the x- and y-coordinates of all the points of P are strictly positive real numbers.



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: `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 6. (15 points) The objective of this problem is to design an enhanced stack data structure, called `MinStack`. For concreteness, let's assume that the stack just stores integers. Your stack should support the standard stack operations `void push(int x)`, which pushes `x` on top of the stack, and `int pop()`, which removes the element at the top of the stack and returns its value. It must also support the additional operation, `int getMin()`, which returns the smallest value currently in the stack, *without altering the contents of the stack*. Finally, there is a constructor `MinStack(int n)`, which is given the maximum number n of items that will be stored in the stack.

Present pseudocode for a data structure that implements these operations. All operations should run in $O(1)$ time. (We will give partial credit if algorithm is correct, but your running time is worse than this.) Your answer should include the following things:

- Explain what objects are maintained by your data structure.
- Explain how the data structure is initialized (that is, what does the constructor do?)
- Present pseudocode descriptions of `push(x)`, `pop()`, and `getMin()`.

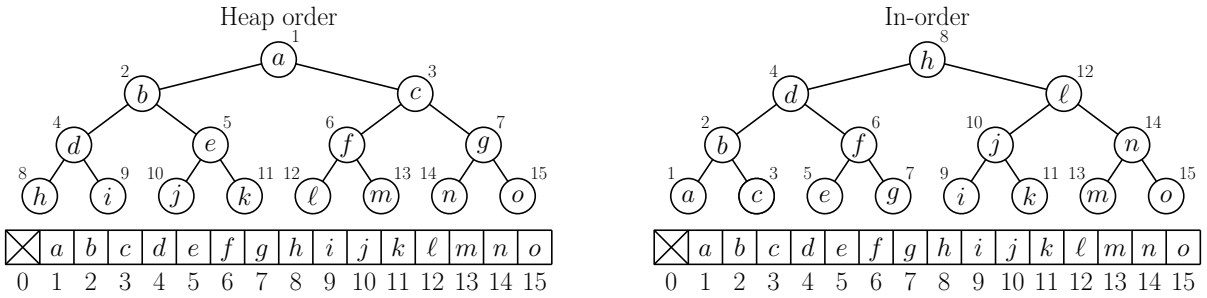
No error checking is needed. (No more than n elements will be in the stack at any time and no `pop` or `getMin` from an empty stack.)

Problem 7. (20 points) One of the ideas that we saw in the buddy system is that a complete binary tree can be represented *without pointers*. The contents of a complete n-node tree are stored in an array $A[1...n]$, and all tree relationships can be computed simply through arithmetic or bit-wise operations on the node indices in A. In this problem, we will consider two such "pointerless" tree representations. In each, we assume that the tree contains $n = 2^k - 1$ nodes for some $k \geq 1$, and all leaves are at the same level.

(7.1) (10 points) Consider the *Heap-Order* tree structure shown in the figure below left (for $n = 15$). Given a node at index x, where $1 \leq x \leq n$, give a short piece of pseudo-code (or a mathematical expression) to compute each of the following operations. You may use standard integer arithmetic or bit-wise operations.

For example, for $n = 15$, `left(3)=6`, `right(3)=7`, `parent(3)=1`, and `sibling(3)=2`.

- `left(int x)`: The index of x's left child (assuming x is not a leaf).
- `right(int x)`: The index of x's right child (assuming x is not a leaf)
- `parent(int x)`: The index of x's parent (assuming x is not the root)
- `sibling(int x)`: The index of x's sibling (assuming x is not the root)



(7.2) (10 points) Consider the *In-Order* tree structure shown in the figure above right (for $n = 15$). Repeat problem (7.1) for this ordering. You may use standard integer arithmetic or bit-wise operations. (Hint: It may be useful to first derive a function `level(x)` that returns the level of the of node x in the tree, where the leaves are at level 0.)

For example, for $n = 15$, `left(12)=10`, `right(12)=14`, `parent(12)=8`, and `sibling(12)=4`.

- `left(int x)`: The index of x's left child (assuming x is not a leaf).
- `right(int x)`: The index of x's right child (assuming x is not a leaf)
- `parent(int x)`: The index of x's parent (assuming x is not the root)
- `sibling(int x)`: The index of x's sibling (assuming x is not the root)