**A**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **CS 112—Final Exam—Spring 2016**

There are 8 problems on the exam. The first and last are mandatory, and **you *must* eliminate any one of problems 2 – 7 by drawing an X through it.** Problem 1 is worth 10 points, and all other problems are worth 15. Please write in pen if possible. Circle answers when they occur in the midst of a bunch of calculations. If you need more room, use the back of the sheet and tell me this on the front sheet.

**Problem One.** **(True/False – MANDATORY – 10 points)** Write True or False to the left of each statement.

1. A separate-chaining hash table containing N keys and N buckets (slots in the array) must have a worst-case lookup time of Θ( 1 ).
2. A game of "perfect information" in one in which there are no random events such as the throw of dice.
3. A binary search tree is a special case of an undirected graph.
4. A binary search tree is a special case of a 2-3 tree.
5. The second-largest key in a binary Max Heap always occurs as a child of the root.
6. The largest key in a binary search tree always occurs at a leaf node.
7. The smallest key in a 2-3 tree always occurs at a leaf node.
8. Mergesort is difficult to implement without an auxiliary array (to transfer data from the original array).
9. In a queue implemented as a circular buffer with resizing, a call to dequeue() can never trigger a resize operation.
10. I have put my name at the top of this exam. ☺

**Problem Two (Binary Search Trees – 15 points).**  This problem has multiple parts, all referring to the diagram of the tree on the right; each part is independent and refers to the original diagram, i.e., start all over again with the original tree diagram for each part.

**(A)** Supposing that we can only insert integers into this tree, and no duplicates, what keys could be inserted to the right of the leaf node containing 2?

**(B)** Give a reverse postorder traversal of this tree.

**(C)** Give a left-to-right breadth-first (level order)

 traversal of this tree.

 **(D)** As discussed in lecture, it is useful when deleting nodes from a BST that we alternately move up nodes from the right and left subtrees of the node to be deleted; draw the tree that results from deleting the root of the tree in the diagram four times, alternately moving up a node from the right, left, right, and left subtrees of the root.

**Problem Three** **(Hash Tables – 15 points).**  For a hash table implemented using the technique of **Linear Probing**, assume that you have an array size of 7 and a hash function

 int hash(int key) { return ( 3 \* key % 7 ); }

Suppose that you want to perform the following sequence of operations:

insert( 4 );

insert( 3 );

insert( 7 );

insert( 11 );

insert( 9 );

insert( 2 );

delete( 7 );

delete( 9 );

insert( 9 );

(A) Show in the space above the result of performing these operations on an initially-empty table. Use -1 as a sentinel for “never used” and -2 for “used but currently empty.”

(B) Recall that a hash function generally has the form ( P \* key % M ) for suitable primes P and M. What would be the effect on the hash table’s performance if P = M? Be specific and give the Θ(..... ) estimated cost for looking up an item in such a table.

(C) Precisely how many slots in the array do you have to examine in each of the following tests for the hash table you got at the end of (A)?

member( 9 )

member( 2 )

member( 5 ) [unsuccessful test]

**Problem Four (Max Heaps – 15 points).** Show the Max Heap that would result after the following series of operations; you should work this out by using a tree structure to do the operations, and then (A) fill in the array representation for this final tree, (B) draw your final tree, and (C) answer the general questions below.

1. insert(9);

(A)

1. insert (4);
2. insert (10);

Note: The array will not be entirely filled by the operations given.

1. insert (6);
2. insert (7);
3. n = getMax(); // delete the root and assign its key to n
4. insert(13);
5. insert(8);
6. m = getMax(); // delete the root and assign its key to m

(B) Draw your final tree:

 (C) Complete the following definitions, according to the presentation in lecture on heaps:

// given index k in the array, return index of parent of this key

int parent(int k) { return }

// given index k in the array, return index of right child of this key

int rchild(int k) { return }

// given index k in the array, return index of left child of this key

int lchild(int k) { return }

// given index k in the array, determine if it is a leaf node

boolean isLeafNode(int k) { return }

**Problem Five.** Suppose we consider max-heaps implemented as binary trees using the Nodes as shown (i.e., instead of the normal arrays--I gave examples of these trees when presenting the lecture on heaps). Write a recursive function

 boolean isMaxHeapOrdered(Node p) { …… }

which takes a binary tree and determines if it has the **max-heap ordered** property. Note carefully the following:

class Node {

 int key;

 Node left;

 Node right;

}

* You may not use a loop (the function or the helper function must be recursive).
* You may write a helper-function for this if you wish (meaning, the function above is a wrapper around another function).
* The assumed Node class is shown at right.
* You do NOT need to tell if it has the right shape for a heap, ONLY if the tree has the requisite heap-ordered property for a max-heap.
* You must account for the null pointer; a null tree is assumed to be help-ordered by default (so calling this function on null will return true).

// returns true if each parent is >= children

boolean isMaxHeapOrdered(Node p) {

 if(p == null)

 return true;

 if(!isMaxHeapOrdered(p.left))

 return false;

 if(!isMaxHeapOrdered(p.right))

 return false;

 if(p.left != null && p.key < p.left.key)

 return false;

 if(p.right != null && p.key < p.right.key)

 return false;

 return true;

}

**Problem Six** **(2-3 Trees -- 15 points).** Assume you need to insert the integer keys 2, 8, 16, 11, 6, 4, 15, 9, 17, 23 in that order into an initially-empty 2-3 tree.

1. Show the data structure that would result after inserting these integers (use back of previous page or scrap paper for preliminary steps, and CIRCLE your final tree.
2. Consider the example tree to the right. What is the worst case for a lookup of a key which is present in this specific tree (i.e., the exact number of comparisons)?
3. Answer True or False to the following statement: Inserting a single key into this tree could never cause its height to increase.
4. In general what is the worst-case for looking up a key in a 2-3 tree with N keys? (Express in terms of Θ(….)).

**Problem Seven (MinMax Search -- 15 points).** For the following Min-Max tree, assume the evaluation function produces values at all the leaves as shown.

**(A)** Fill in all the nodes with the value which would be calculated by Min-Max search, and indicate which move the Max player would take.

**(B)** Draw an X through any nodes which would NOT be searched, because of a cutoff due to Alpha-Beta Pruning.

**(C)** We did not study the time complexity of game tree search, but you have all the tools you need to answer the following question: Supposing a MinMax tree has N nodes, what is the Θ( .... ) complexity of the algorithm which you just performed, i.e., calculating the values at each node and performing Alpha-Beta pruning?

1

7

5

6

3

4

8

-1

9

-2

2

3

9

8

4

**Problem Eight (Graph Search – 15 points).** For the undirected graph below, suppose the “master list” of vertices is V = { A, B, C, D, E, F, G, H }. In each case, you would *access the adjacent nodes in alphabetic order*. Show the

1. Order that vertices would be visited by a **recursive depth-first search.**
2. Order that nodes would be visited by depth-first search using an **explicit stack**.
3. Order that nodes would be visited by a **breadth-first** **search**.

D

H

F

C

B

EE

A

G

**Problem Nine.** **(MANDATORY – 15 points)** For the following algorithms, for the first set, state the average- and worst-case time (in terms of Θ ) for performing the indicating operation. The number of items stored in each data structure is N. Assume that none of the operations produces an error. Those entries in the table contain “X” do not have to be answered. For this question, your Θ(......) estimate is counting the number of times you must touch an element (e.g., to compare it or move it or print it out, etc.).

|  |  |  |
| --- | --- | --- |
| **Algorithm/Problem** | **Average Case Θ(......)** | **Worst Case Θ(......)** |
| Find an item in an ordered array using Binary Search |  |  |
| Insert an item into an unordered list if duplicates are *not* permitted. |  |  |
| Insert an item into an unordered list if duplicates *are* allowed.  |  |  |
| Delete an item from a max-queue implemented as a heap. | X |  |
| Dequeue an item from a queue. |  |  |
| Enqueue an item into a queue implemented as a ring buffer with resizing. | X |  |
| Insert an item into a linear-probing hash table.  | X |  |
| Delete an item from a Binary Search Tree. |  |  |
| Insert an item in a 2-3 tree.  | X |  |
| Delete an element from a separate-chaining hash table (with the assumptions from lecture)  |  |  |
| Print out all the nodes in a Binary Search Tree using preorder search.  |  |  |
| Sort an arbitrary list using Quicksort. |  |  |
| Sort an arbitrary list using Mergesort |  |  |
| Sort an arbitrary list using Selection Sort |  |  |
| Sort an arbitrary list using Insertion sort.  |  |  |