University of Illinois at Urbana-Champaign
Department of Computer Science

Second Examination

CS 225 Data Structures and Software Principles
Fall 2009
7p-9p, Tuesday, November 3

Name:
NetID:
Lab Section (Day/Time):

- This is a **closed book** and **closed notes** exam. No electronic aids are allowed, either.
- You should have 5 problems total on 18 pages. The last two sheets are scratch paper; you may detach them while taking the exam, but must turn them in with the exam when you leave.
- Unless otherwise stated in a problem, assume the best possible design of a particular implementation is being used.
- Unless the problem specifically says otherwise, (1) assume the code compiles, and thus any compiler error is an exam typo (though hopefully there are not any typos), and (2) assume you are NOT allowed to write any helper methods to help solve the problem, nor are you allowed to use additional arrays, lists, or other collection data structures unless we have said you can.
- We will be grading your code by first reading your comments to see if your plan is good, and then reading the code to make sure it does exactly what the comments promise. In general, complete and accurate comments will be worth approximately 30% of the points on any coding problem.
- Please put your name at the top of each page.

Problem	Points	Score	Grader
1	16		
2	20		
3	20		
4	14		
5	10		
6	10		
7	10		
Total	100		

1. **[Miscellaneous – 16 points].**

MC1 (2pts)

Suppose that a client performs an *intermixed* sequence of stack push and pop operations. The push operations push the integers 0 through 9 in order on to the stack; the pop operations print out the return value. Which of the following sequences could not occur?

(a) 4 3 2 1 0 9 8 7 6 5

(b) 2 1 4 3 6 5 8 7 9 0

(c) 0 4 6 5 3 8 1 7 2 9

(d) 4 6 8 7 5 3 2 9 1 0

(e) All of these sequences are possible.

MC2 (2pts)

A queue cannot be implemented using only _____ for holding data.

(a) a stack

(b) a linked list

(c) an array

(d) More than one of a), b), c) can be used to fill in the blank.

(e) None of a), b), c) can be used to fill in the blank.

MC3 (2pts)

Suppose we have implemented the `Stack` ADT as a singly-linked-list with head and tail pointers and no sentinels. Which of the following best describe the running times for the functions `push` and `pop`, assuming there are $O(n)$ items in the list, and that the bottom of the stack is at the head of the list (all pushing and popping occurs at the tail)?

(a) $O(1)$ for both functions.

(b) $O(n)$ for both functions.

(c) $O(1)$ for `push` and $O(n)$ for `pop`.

(d) $O(n)$ for `push` and $O(1)$ for `pop`.

(e) None of these is the correct choice.

MC4 (2pts)

Consider the following partial C++ code:

```
#include <iostream>
#include <list>
using namespace std;

template< class Iter, class Formatter>
bool mystery(Iter first, Iter second, Formatter runit) {
   Iter temp = first;
   bool p = true;
   while (!(second == temp)) {
      p = (p && runit(*first, *second));
      first++;
      second--;
   }
   return p;
}

class noClue {
public:
   bool operator()(int a, int b) {
      return (a==b);
   }
};

int main() {

   list<int> s;

   ... // list insertions here

   list<int>::iterator it1 = s.begin();
   list<int>::iterator it2 = s.end();
   it2--;

   noClue nc;
   if ( mystery<list<int>::iterator, noClue>(it1, it2, nc))
      cout << "yes" << endl;
   else
      cout << "no" << endl;

   return 0;
}
```

Which of the following statements is true?

(a) This code does not compile because of a type mismatch in the `mystery` parameter list.

(b) This code does not compile because of a syntax error in the template instantiation for `mystery`.

(c) If the list consists of the integers 1, 2, 3, 2, 1 *in that order, with the first item on the left*, then the output is "yes".

(d) If the list consists of the integers 1, 2, 3, 2, 1, 4 *in that order, with the first item on the left*, then the output is "yes".

(e) None of these options describes the behavior of this code.

MC5 (2pts)

Which of the following statements is true for a B-tree of order m containing n items?

(i) The height of the B-tree is $O(\log_m n)$ and this bounds the total number of disk seeks.

(ii) A node contains a maximum of $m - 1$ keys, and this bounds the number of disk seeks at each level of the tree.

(iii) Every Binary Search Tree is also an order 2 B-Tree.

Make one of the following choices.

(a) Only item (i) is true.

(b) Only item (ii) is true.

(c) Only item (iii) is true.

(d) Two of the above choices are true.

(e) None of choices (i), (ii), or (iii) are true.

MC6 (2pts)

To justify the use of arrays for data structures of unbounded size (stacks, queues, lists, etc.), we proved that the following strategy results in $O(n)$ running time over a sequence of n inserts for an average of $O(1)$ per insertion.

(a) When an array of size n fills, create a new array of size n, and maintain all the arrays. Do not copy any data.

(b) When an array of size n fills, create a new array of size $2n$, and maintain all the arrays. Do not copy any data.

(c) When an array of size n fills, create a new array of size $n+d$ for some large fixed constant d, and copy the data into the new array.

(d) When an array of size n fills, create a new array of size $2n$ and copy the data into the new array.

(e) None of these strategies give the performance we describe.

MC7 (2pts)

Objects of type `iterator` promise to implement each of the following *except*...

(a) `operator+`

(b) `operator*`

(c) `operator==`

(d) `operator=`

(e) All of these are implemented in an iterator.

MC 8 (2)

In an array-based implementation of a stack we achieve efficient `push` and `pop` operations by _____.

(i) placing the top of the stack at the start of the array.

(ii) placing the bottom of the stack at the start of the array.

Make one of the following choices.

(a) Only item (i) can be used to fill in the blank.

(b) Only item (ii) can be used to fill in the blank.

(c) Either item (i) or item (ii) can be used to fill in the blank.

(d) Neither item (i) or item (ii) can be used to fill in the blank.

2. **[Efficiency – 20 points].**

Each item below is a description of a data structure, its implementation, and an operation on the structure. In each case, choose the appropriate running time from the list below. The variable n represents the number of items (keys, data, or key/data pairs) in the structure. In answering this question you should assume the best possible implementation given the constraints, and also assume that every array is sufficiently large to handle all items. Please use the scantron sheets for your answers.

(a) $O(1)$
(b) $O(\log n)$
(c) $O(n)$
(d) $O(n \log n)$
(e) None of these running times is appropriate.

(MC 9) _____ Enqueue for a Queue implemented with an array.

(MC 10) _____ Dequeue for a Queue implemented with an array.

(MC 11) _____ Worst case for insertion into a Binary Search Tree.

(MC 12) _____ Worst case for removal from a Binary Search Tree.

(MC 13) _____ Worst case for an algorithm to return all keys that are greater than 20 and that are multiples of 3 in a Binary Search Tree.

(MC 14) _____ Worst case for insertion into an AVL Tree.

(MC 15) _____ Worst case for an algorithm to return all keys that are greater than 20 and that are multiples of 3 in an AVL Tree.

(MC 16) _____ Level order traversal of an AVL Tree.

(MC 17) _____ Build an AVL tree with keys that are the numbers between 0 and n, in that order, by repeated insertion into the tree.

(MC 18) _____ Build a binary search tree with keys that are the numbers between 0 and n, in that order, by repeated insertion into the tree.

3. **[Quadtrees – 20 points].**

For this question, consider the following partial class definition for the Quadtree class, which uses a quadtree to represent a square bitmap image as in MP5.

```
class Quadtree
{
  public:
     // constructors and destructor; all of the public methods from MP5, including:

   void buildTree(BMP const & source, int resolution);
   RGBApixel getPixel(int x, int y) const;
   BMP decompress() const;
   void clockwiseRotate();  // 90 degree turn to the right
   void prune(int tolerance);
   int pruneSize(int tolerance) const;
   int idealPrune(int numLeaves) const;

  private:
     class QuadtreeNode
     {
        QuadtreeNode* nwChild;  // pointer to northwest child
        QuadtreeNode* neChild;  // pointer to northeast child
        QuadtreeNode* swChild;  // pointer to southwest child
        QuadtreeNode* seChild;  // pointer to southeast child

        RGBApixel element;  // the pixel stored as this node's "data"
     };

     QuadtreeNode* root;    // pointer to root of quadtree, NULL if tree is empty.
};
```

You may assume that the quadtree is complete and that it has been built from an image that has size $2^k \times 2^k$. As in MP5, the element field of each leaf of the quadtree stores the color of a square block of the underlying bitmap image; for this question, you may assume, if you like, that each non-leaf node contains the component-wise average of the colors of its children. You may not use any methods or member data of the Quadtree or QuadtreeNode classes which are not explicitly listed in the partial class declaration above. You may assume that each child pointer in each leaf of the Quadtree is NULL.

(a) (3 points) Write a *public* member function `void Quadtree::flipVert()`, which reflects an image across the horizontal axis running across the center of the image. For example, if the image is a portrait of a person standing upright, the result will be the image of the person upside down. Your function should call ONE *private* helper function that you will be writing in the next part of the problem. Your implementation must work correctly for `Quadtree`s which have been pruned as described in MP5. Write the function as it would appear in the quadtree.cpp file for the `Quadtree` class.

```
void Quadtree::flipVert(){
// Your code goes here


}
```

(b) (7 points) Write the *private* helper method you invoked in the previous part. For this one, you may choose the return value and the number and types of parameters. Note that our skeleton below should have sufficiently many lines for your solution, but you are welcome to add more if you need to do so. Please try to comment your code inline and to the right of the skeleton.

```
_________   ____________::____________(___________________) {

        ________________________________________________

        ________________________________________________

        ________________________________________________

        ________________________________________________

        ________________________________________________

        ________________________________________________

        ________________________________________________

        ________________________________________________

        ________________________________________________

        ________________________________________________

        ________________________________________________

        ________________________________________________
}
```

(c) (5 points) Now consider the transformation on a bitmap image which moves the pixel at coordinates (i, j) to coordinates (j, i) for each pixel in the image. We call this transformation a "transposition". Under a transposition, the first row of an image becomes the first column, the second row becomes the second column, and so forth. Your task is to add a public method `void Quadtree::transpose()` to the `Quadtree` class, which performs a transposition on the underlying bitmap image.

You you should solve this problem by calling existing *public* Quadtree member functions, including the one you wrote in part (a). Your implementation must work correctly for `Quadtree`s which have been pruned as described in MP5. Write the method as it would appear in the quadtree.cpp file for the `Quadtree` class.

(Hint: it may help to think of a transposition as a reflection across the diagonal running from the top-left corner to the bottom-right corner of the bitmap. BIG hint: there is an elegant solution with no more than 3 lines of code.)

```
void Quadtree::transpose(){
// Your code goes here









}
```

(d) (5 points) Analyze and give a tight asymptotic bound on the running time of your implementation for part (c). Your bound should be stated in terms of N, the number of nodes in the `Quadtree`. Briefly justify your answer.

4. **[Binary Search Trees – 14 points].**

(a) (9 points) Scrutinize the following code and figure out what the function `mystery` does when called on a node in a binary search tree. You may assume that both `croot` and `croot->left` are non-NULL, and that the keys in the BST are unique.

```
treeNode * & BST<T>::mystery(treeNode * & cRoot) {
   return intrigue(cRoot->left);
}

treeNode * & BST<T>::intrigue(treeNode * & cRoot) {
   if (cRoot->right == NULL) return cRoot;
   else return intrigue(cRoot->right);
}
```

Circle every accurate statement in the list below.

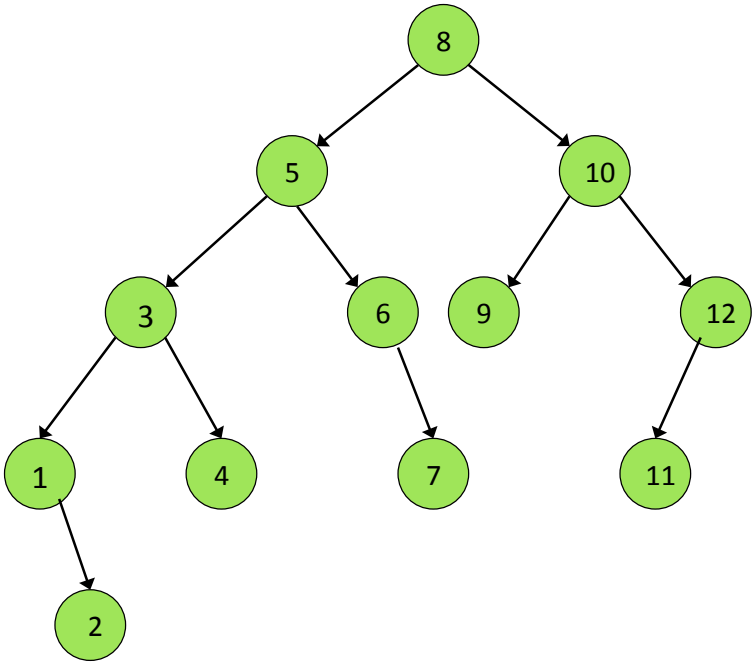
- `mystery` returns the node with minimum key in the tree rooted at `croot`.
- `mystery` returns the node with maximum key in the tree rooted at `croot`.
- `mystery` returns the node with maximum key less than `croot->key` in the tree rooted at `croot`.
- `mystery` returns the node with minimum key greater than `croot->key` in the tree rooted at `croot`.
- `mystery` does an inOrder traversal of the subtree rooted at `croot`.
- `mystery` does a postOrder traversal of the subtree rooted at `croot->left`.
- in the worst case `mystery` runs in time proportional to the height of the tree, $O(h)$.
- in the worst case `mystery` runs in time $O(\log n)$ where n is the number of nodes in the tree.
- in the worst case `mystery` runs in time $O(n)$ where n is the number of nodes in the tree.

(b) (3 points) Which binary search tree *public* member function employs this code?

(c) (2 points) Briefly explain the context in which the code is used.

5. **[AVL Trees – 10 points].**

This tree will become unbalanced by the removal of some of the nodes. Fill in the table below, telling, for each node, whether its removal will unbalance the tree, at what node the first imbalance occurs, what kind of rotation would fix that imbalance, and whether or not a repair of the first imbalance invokes additional imbalance(s). Note that each node is removed from the original tree. (If removal of a node does not unbalance the tree, just leave its entry blank.)



Node to Remove	Unbalanced Node	Rotation to Fix	Additional Imbalance? (Yes or No)
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			

6. **[A simple proof – 10 points].**

A *perfect* binary tree of height h is defined recursively as follows:

- an empty tree is a perfect binary tree of height -1.
- a non-empty tree consisting of a root r, and left and right subtrees T_L and T_R, is a perfect binary tree of height h if and only if T_L and T_R are perfect binary trees of height $h-1$.

(a) (2 points) Draw the perfect binary trees of height 0, 1, 2, and 3 on the lines below.

_____ _____ _____ _____

(b) (3 points) Give an expression for the total number of nodes in a perfect binary tree of height h:

(c) (5 points) Prove that your answer to part (b) is correct by induction:

Consider an arbitrary perfect binary tree of height h.

- If $h = -1$ then the expression in part (b) gives: _____ which is the number of nodes in a tree of height -1.
- otherwise, if $h > -1$ then we denote our tree by a root r together with subtrees T_L and T_R. By an inductive hypothesis that says:

_____

we have _____ nodes in T_L and _____ nodes in T_R

for a total of _____ nodes, which was what we wanted to prove.

7. **[B-Trees – 10 points].**

(a) (3 points) What is the maximum number of keys we can store in a B-Tree of order 128 that has height 3?

(b) (3 points) What is the minimum number of keys we can store in a B-Tree of order 128 that has height 3?

(c) (4 points) In class we proved that the search time for finding a key in a B-Tree is $O(m \log_m n)$. In this problem, we'd like you to explain each of the factors m, and $\log_m n$ in that result:

i. Tell as much as you can about the factor m:

ii. Tell as much as you can about the factor $\log_m n$:

scratch paper