University of Illinois at Urbana-Champaign
Department of Computer Science

Second Examination - RUBRIC

CS 225 Data Structures and Software Principles
Fall 2011
9a-11a, Wednesday, November 2

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 15 pages. The last sheet is scratch paper; you may detach it while taking the exam, but must turn it in with the exam when you leave. Use scantron forms for Problems 1 and 2.
- Unless otherwise stated in a problem, assume the best possible design of a particular implementation is being used.
- Unless the problem specifically says otherwise, assume the code compiles, and thus any compiler error is an exam typo (though hopefully there are not any typos).
- 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	25		scantron
2	25		scantron
3	15		
4	20		
5	15		
Total	100		

1. **[Miscellaneous – 25 points].**

MC1 (2.5pts)

Suppose you implement a `queue` using a singly linked list with head and tail pointers so that the front of the `queue` is at the tail of the list, and the rear of the `queue` is at the head of the list. What is the best possible worst-case running time for `enqueue` and `dequeue` in this situation? (As a reminder, `enqueue` occurs at the rear of the queue.)

(a) O(1) for both functions.

(b) O(1) for enqueue and O(n) for dequeue.

(c) O(n) for enqueue and O(1) for dequeue.

(d) O(n) for both functions.

(e) None of these is the correct response.

MC2 (2.5pts)

Think of an algorithm that uses a `Stack` to efficiently check for unbalanced parentheses. What is the maximum number of parentheses that will appear on the stack at any time when the algorithm analyzes the string `(()(())(()))`?

(a) 3

(b) 4

(c) 5

(d) 6

(e) None of these is correct.

MC3 (2.5pts)

Suppose a binary tree holds 357 keys. Then our node-based implementation of that tree has how many `NULL` pointers?

(a) 178

(b) 358

(c) 712

(d) The answer cannot be determined from the information given.

(e) None of these is the correct response.

MC4 (2.5pts)

Which of the following sequences of keys CANNOT be the inOrder traversal of an AVL tree?

(a) 1 3 6 8 12 15 17 19

(b) 50 120 160 172 183 205 214 230

(c) 12 14 20 22 24 27 40 45

(d) More than one of these are INVALID inOrder traversals.

(e) All of these are valid inOrder traversals.

MC5 (2.5pts)

How many data structures in this list can reasonably be used to implement a *Dictionary*?

stack queue binary search tree AVL tree B-Tree Hash Table

(a) 2

(b) 3

(c) 4

(d) 5

(e) 6

MC6 (2.5pts)

Examine `mysteryFunction` below. (Note that in context, `t->right` will not be `NULL`.)

```
void mysteryFunction(treeNode * & t) {

   treeNode * y = t->right;
   t->right = y->left;
   y->left = t;
   y->height = max( height(y->right), height(y->left)) + 1;
   t->height = max( height(t->right), height(t->left)) + 1;
   t = y;

}
```

Which of the following `Dictionary` functions will certainly **not** invoke `mysteryFunction`?

(a) `insert(key);`

(b) `remove(key);`

(c) `find(key);`

(d) Two or more of these will certainly **not** invoke `mysteryFunction`.

(e) All of these could employ `mysteryFunction`.

MC7 (2.5pts)

Consider the AVL Tree built by inserting the following sequence of integers, one at a time: $5, 4, 7, 9, 8, 3, 1$. What is 8's left child?

(a) `NULL`

(b) The node containing 4.

(c) The node containing 5.

(d) The node containing 7.

(e) None of these answers is correct.

MC8 (2.5pts)

Find the minimum number of keys that can be stored in a B-Tree of order 64 and height 4.

(a) $2^{20} + 1$

(b) $2^{20} - 1$

(c) $2^{21} + 1$

(d) $2^{21} - 1$

(e) $2^{25} - 1$

MC9 (2.5pts)

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 $O(\log_m n)$.

(ii) A node contains a maximum of $m - 1$ keys, and this is an upper bound on the number of key comparisons at each level of the tree during a search.

(iii) For fixed n, decreasing m decreases the number of disk seeks.

(a) Only (i) is false.

(b) Only (ii) is false.

(c) Only (iii) is false.

(d) At least two of (i), (ii) and (iii) are false.

(e) None of these characteristics is false.

MC10 (2.5pts)

Suppose a hash table has size 10, and that the search keys are strings consisting of 3 lower case letters. We want to hash 7 unknown values from this keyspace. In the hash function, when we refer to the alphabet positions of the letters, we mean: "a"= 1, "b"= 2, . . . , "z"= 26.

$$h(k) = (\texttt{product of the alphabet positions of } k'\texttt{s letters})^4 \ \ \%10$$

Which of these ideal hash function characteristics are violated by this hash function?

(i) A good hash function distributes the keys uniformly over the array.

(ii) A good hash function is deterministic.

(iii) A good hash function is computed in constant time.

(a) Only (i) is violated.

(b) Only (ii) is violated.

(c) Only (iii) is violated.

(d) At least two of (i), (ii) and (iii) are violated.

(e) None of these characteristics are violated–our hash function is a good one!

2. **[Efficiency – 25 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 (unless otherwise stated). Please use the scantron sheets for your answers.

(a) $O(1)$
(b) $O(\log n)$
(c) $O(n)$
(d) $O(n \log n)$
(e) $O(n^2)$

(MC 11) _A_ Consider a sequence of n `push` operations on a `Stack` implemented using an array. What is the average cost per push, over the sequence, of pushing the sequence into an array of size n if the array is already full at the start of the sequence? You may resize the array however you like.

(MC 12) _A_ `Dequeue` for a `Queue` implemented with an array.

(MC 13) _C_ Find the minimum key in a Binary Tree (not necessarily BST).

(MC 14) _B_ Remove a single key from a height balanced Binary Search Tree.

(MC 15) _C_ Find the *In Order Predecessor* of a given key in a Binary Search Tree (if it exists).

(MC 16) _B_ Find the *In Order Predecessor* of a given key in an AVL Tree (if it exists).

(MC 17) _A_ Perform `rightLeftRotate` around a given node in an AVL Tree.

(MC 18) _C_ Determine if a given Binary Search Tree is an AVL Tree.

(MC 19) _E_ Build a binary search tree with keys that are the numbers between 0 and n, in that order, by repeated insertions into the tree.

(MC 20) _C_ Remove all the nodes from a BST (as is done by the destructor).

3. **[MP4ish – 15 points].**

   In MP4 we asked you to implement the `Stack` and `Queue` data structures, and use them to facilitate a variety of flood fill algorithms. This problem will revisit several components of that assignment. In each part, be careful to notice which fill, *Depth*-first or *Breadth*-first, we're asking you about.

   (a) (5 points) Below we have included the partial class definition for the `Stack` and `Queue` classes.

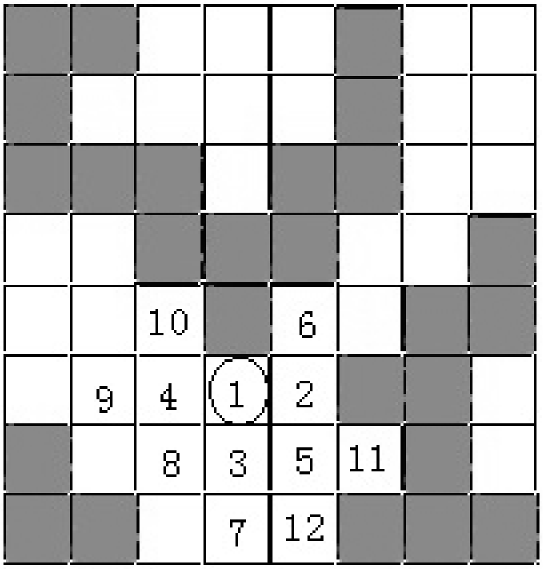
```
template<class T> class Stack {
public:
   //   adds T to the top of the stack
   void push(T const & newItem);

   //   removes the object on top of the stack, and returns it to the caller
   T pop();

   //   finds the object on top of the stack, and returns it to the caller;
   //     unlike pop(), this operation does not alter the stack itself
   T peek() const;

   //   returns true if the stack is empty, and false otherwise
   bool isEmpty() const;

// PRIVATE section not included
};

template<class T> class Queue {
public:
   //   adds T to the back of the queue
   void enqueue(T const & newItem);

   //   removes the object at the front of the queue, and returns it
   T dequeue();

   //   finds the object at the front of the queue, and returns it to
   //     the caller; unlike pop(), this operation does not alter the queue
   T peek();

   //   returns true if the queue is empty, and false otherwise
   bool isEmpty() const;

private:
   Stack<T> inStack;
   Stack<T> outStack;
};
```

Please show us your implementation of the `dequeue` function, whose skeleton is below. Remember that we have constrained you to use ONLY the member variables declared in the `Queue` class above. Comment your code so we know your plan. Feel free to draw illustrations so we know what algorithm you're employing.

```
template<class T>
T Queue<T>::dequeue()
{

   if(outstack.isEmpty()){                  //+1
      while(!instack.isEmpty()){            //+1
          outstack.push(instack.pop());     //+2 (+1 for each statement)
      }
   }

   return outstack.pop();                  //+1

}
```

SCORING:
-2.5 for an extra while loop
-2.5 for a solution that doesn't work (e.g. forgetting the first if statement)

Figure 1: Problem 3(b): BFSfillSolid



▨	▨				▨		
▨					▨		
▨	▨	▨		▨	▨		
		▨	▨	▨			▨
		10	▨	6		▨	▨
	9	4	1	2	▨	▨	
▨		8	3	5	11	▨	
▨	▨		7	12	▨	▨	▨

(b) (3 points) Suppose we execute function `BFSfillSolid`, one of two main fill routines from MP4, on the following 8x8 pixel image, beginning at the circled node, (3,5), and changing white pixels to solid red. Place the numbers 1 through 12, in order, in the first 12 pixels whose colors are changed by the function. Assume that we start the algorithm by adding the circled cell to the ordering structure, and that we add the four neighboring pixels to the structure clockwise beginning on the right, followed by down, left, and up. As a reminder, the fill should change the color when a cell is *removed* from the ordering structure.

SOLUTION:

- 0.25 pt for each single numbering, but we'll round the total to integer or .5
- -0.5 pt for the first 1 or 2 incorrect numberings
- -0.5 pt for not starting from the circled pixel
- no point for only making the first numbering correct

(c) (4 points) Suppose we want to fill some part of a arbitrary n-by-n image. What is the worst-case running time of `DFSfillSolid` if we start from an arbitrary location? Be sure to give your running time in terms of n, the length of one *side* of the square image.

SOLUTION: 4 points for anything $O(n^2)$, 2 points for anything $O(n)$, 0 points otherwise

(d) (3 points) What data structure was used to order the points for filling in function `DFSfillSolid`?

SOLUTION: 3 points for Stack, 1 point for Queue.

4. **[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:
   // ctors and dtor and 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 prune(int tolerance);
   ...
   // a NEW function for you to implement
   void prunish(int tolerance, double percent);

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
  int resolution; // init to be the resolution of the quadtree NEW
  int distance(RGBApixel const & a, RGBApixel const & b); // returns sq dist
  void clear(QuadtreeNode * & cRoot); // free memory and set cRoot to null
  // a couple of private helpers are omitted here.
};
```

   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) (8 points) Write a *private* member function `int Quadtree::tallyDeviants(RGBApixel const & target, QuadtreeNode const * curNode, int tolerance)`, which calculates the number of leaves in the tree rooted at `curNode` with element more than `tolerance` distance from `target`. You may assume that you are working on a complete (unpruned), non-empty Quadtree. Write the method as it would appear in the quadtree.cpp file for the `Quadtree` class. We have included a skeleton for your code below–just fill in the blanks to complete it.

```
int Quadtree::tallyDeviants(RGBApixel const & target,
          QuadtreeNode const * curNode, int tolerance) _const (1pt)_ {

   // function not called with curNode == NULL;
   if (curNode->_xxChild (1pt)_ == _NULL (1pt)_) {  // check for leaf

      RGBApixel current = curNode->element;

      if (distance(current, target) _> (1pt)_ tolerance) return _1 (1pt)_;
      else return 0;

   }

   // otherwise...recurse!
   int devTotal = _tallyDeviants(target,curNode->nwChild,tolerance)+
            tallyDeviants(target,curNode->neChild,tolerance)+
            tallyDeviants(target,curNode->swChild,tolerance)+
            tallyDeviants(target,curNode->seChild,tolerance) (2pt) _


   return _devTotal (1pt)_;
}
```

(b) (8 points) Our next task is to write a private member function declared as `void Quadtree::prunish(QuadtreeNode * curNode, int tolerance, int res, double percent)` whose functionality is very similar to the `prune` function you wrote for MP5. Rather than prune a subtree if ALL leaves fall within a tolerance of the current node's pixel value, `prunish` will prune if at least `percent` of them do. Parameter `res` is intended to represent the number of pixels on one side of the square represented by the subtree rooted at `curNode`. All the constraints on pruning from the `prune` function apply here, as well. That is, you should prune as high up in the tree as you can, and once a subtree is pruned, its ancestors should not be re-evaluated for pruning. As before, we've given you most of the code below. Just fill in the blanks on the next page.

```
void Quadtree::prunish(QuadtreeNode * curNode, int tolerance,
                        int res, double percent) {
```

```
   if (curNode == NULL)
      return;

   // count the number of leaves more than tolerance distance from curNode

   int deviants = _tallyDeviants(curNode->element, curNode, tolerence)_; //(2 points)

   double percentDeviant =_(double) deviants/(res*res)_; //(2 points)

   // prune conditions
   if (percentDeviant _<= (1-percentDeviant)_) { //(2 points)

      clear(curNode->neChild);
      clear(curNode->nwChild);
      clear(curNode->seChild);
      clear(curNode->swChild);

      return;
   }

   // can't prune here :( so recurse!

   _prunish(curNode->NEChild, tolerance, res/2, percent)_   //(2 points)

   _prunish(curNode->NWChild, tolerance, res/2, percent)_

   _prunish(curNode->SWChild, tolerance, res/2, percent)_

   _prunish(curNode->SEChild, tolerance, res/2, percent)_

   return;
}
```

(c) (4 points) Finally, write the public member function `void Quadtree::prunish(int tolerance, double percent)` that prunes from the `Quadtree` any subtree with more than `percent` leaves within `tolerance` color distance of the subtree's root.

```
void Quadtree::prunish(int tolerance, double percent) {

   _prunish(root, tolerance, resolution, percent);_ // 1pt per param

   return;
}
```

5. **[Binary Tree Rank – 15 points].**

In this problem we are going to define a new characteristic of Binary Trees called *rank*, and then we will ask to prove something about trees of rank r.

The *rank* of a binary tree T is recursively defined as follows:

- if $T = \{\}$, then $rank(T) = 0$.
- otherwise, $T = \{r, T_L, T_R\}$, and
  - if $rank(T_L) = rank(T_R)$, then $rank(T) = rank(T_L) + 1$.
  - otherwise, $rank(T) = \max\{rank(T_L), rank(T_R)\}$.

(a) (3 points) In the spaces below, draw trees of rank 1, 2, and 3, each containing as few nodes as possible.

complete tree of height 0   complete tree of height 1   complete tree of height 2

(b) (2 points) In the spaces below, draw two significantly different trees of rank 3, each containing 10 nodes. Do not just make the two trees mirror images of one another.

many possible answers   many possible answers

(c) (5 points) Let $N(r)$ denote the least number of nodes in a tree of rank r. Give a recurrence for $N(r)$. You must justify your answer completely using the definition of *rank*.

SOLUTION: The least number of nodes is represented by $N(r) = \min\{2N(r-1)+1, N(r)+1\}$, and since $N(r)$ is an increasing function, $N(r) = 2N(r-1)+1$.

$N(0) = 0$ (1 point)
$N(r) = 2N(r-1)+1$ (2 points)
justification (2 points)

(d) (5 points) We solved the recurrence and found a closed form solution for $N(r)$ to be $N(r) = 2^r - 1$, $r \geq 0$. Prove that our solution to your recurrence from part (c) is correct by finishing this inductive proof:

Consider an arbitrary non-negative integer r.

- If $r = 0$ then the closed form gives $N(r) = \underline{\ 0\ }$, which is also the base case of the recurrence.
- otherwise, if $r > 0$ then by an inductive hypothesis that says,

$$\underline{\forall 0 < j < r, N(j) = 2^j - 1}$$

we have $N(\underline{\;r-1\;}) = \underline{\;2^{r-1}-1\;}$,

so that $N(r) = \underline{\;2(2^{r-1}-1)+1\;} = \underline{\;2^{r}-1\;}$, which was what we wanted to prove.

scratch paper