University of Illinois at Urbana-Champaign
Department of Computer Science

Second Examination Grading Rubric

CS 225 Data Structures and Software Principles
Fall 2010
7p-9p, Tuesday, 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, (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	20		scantron
2	25		scantron
3	30		
4	15		
5	10		
Total	100		

1. **[Miscellaneous – 20 points].**

MC1 (2.5pts)

Which of the following is not a reasonable choice for implementing the Dictionary Abstract Data Type?

(a) AVL Tree

(b) B-Tree

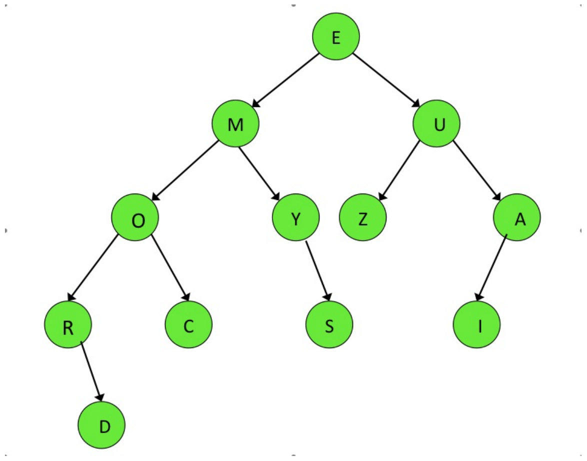
(c) Binary Search Tree

(d) Stack

(e) All of these are reasonable choices for implementing a Dictionary.

MC2 (2.5pts)

Suppose we do a level order traversal of the following binary tree. What is the maximum number of non-null nodes on the queue at one time in the execution of the algorithm?



(a) 4

(b) 5

(c) 6

(d) 12

(e) None of these is the correct number.

MC3 (2.5pts)

Suppose we implement a stack using a dynamically allocated array. If the array fills, we are faced with a dilemma. From among the following choices, select the one that best describes our strategy for handling a filled array.

(a) We declare a new array, twice as big as the original, and copy the data into the new space for a total cost of $O(n)$, over a sequence of n pushes.

(b) We declare another array and keep track of which of the two (or more) arrays contain the current top of the stack in a private member of the stack class. This costs us $O(1)$ per push.

(c) For some fixed k, we create a new array of size $n + 2^k$ and copy the data into the new space for an average cost of $O(1)$ per push operation.

(d) We avoid implementing a stack using a dynamically allocated array, because it is inefficient to have to reallocate memory.

(e) None of these answers is a reasonable response.

MC4 (2.5pts)

Consider the following partial C++ code:

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

template <class Iter, class Formatter>
void mystery(Iter front, Iter end, Formatter doSomething) {

   while (front != end) {
      doSomething(*front, *end);
      front++;  end--;}
}

template <class T> class dunno {
public:
   void operator()(T & a, T & b) {
      a = a + b;}
}

int main() {
   vector<int> v;   ...  // insert an odd number of elements into the vector

   vector<int>::iterator it1 = v.begin();
   vector<int>::iterator it2 = v.end();
   it2--;

   dunno<int> d;
   mystery<vector<int>::iterator,dunno<int> >(it1, it2, d);

   for (int i = 0; i < v.size(); i++) cout << v[i] << endl;
   return 0; }
```

If you assume that all iterators are bi-directional and non-constant, 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 vector consists of the integers 1, 2, 3, 4, 5, 6, 7 *in that order, with the first item on the left*, then the output is 1, 2, 3, 4, 5, 6, 7. (i.e. the code does not change the vector.)

(d) If the vector consists of the integers 1, 2, 3, 4, 5, 6, 7 *in that order, with the first item on the left*, then the output is 8, 8, 8, 4, 5, 6, 7.

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

MC5 (2.5pts)

What is the real name of `mysteryFunction`? 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;

}
```

(a) `leftRotate`

(b) `rightRotate`

(c) `rightLeftRotate`

(d) `leftRightRotate`

(e) None of these choices are correct.

MC6 (2.5pts)

Consider the Binary Search Tree built by inserting the following sequence of integers, one at a time: $5, 4, 7, 9, 8, 6, 2, 3, 1$. If the node containing 5 were removed from the tree, what would be the right child of the node containing 2?

(a) `Null`

(b) The node containing 3.

(c) The node containing 4.

(d) The node containing 6.

(e) None of these answers is correct.

MC7 (2.5pts)

What is the maximum number of keys that can be stored in a B-Tree of order 32 and height 6?

(a) $6 * 2^5 - 1$
(b) $2^{25} - 1$
(c) $2^{30} - 1$
(d) $2^{35} - 1$
(e) None of the above.

MC 8 (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 $O(\log_m n)$ 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) An order 2 B-tree is also a Binary Search 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) All choices (i), (ii), and (iii) are true.

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. 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) __(a)__ `Enqueue` for a `Queue` implemented with a Singly Linked List with a tail pointer, where the front of the queue is at the end (tail) of the list. (`Enqueue` occurs at the head.)

(MC 10) __(c)__ `Dequeue` for a `Queue` implemented with a Singly Linked List with a tail pointer, where the front of the queue is at the end (tail) of the list. (`Enqueue` occurs at the head.)

(MC 11) __(c)__ Worst case for finding a given key in a Binary Search Tree.

(MC 12) __(b)__ Worst case for inserting a single key into an AVL Tree.

(MC 13) __(c)__ Worst case for an algorithm to find the smallest key that is within fixed distance d from a given key in a Binary Search Tree (if such a key exists).

(MC 14) __(b)__ Worst case for an algorithm to find the smallest key that is within fixed distance d from a given key in an AVL Tree (if such a key exists).

(MC 15) __(c)__ Given a Binary Tree, check to see if it is a Binary Search Tree.

(MC 16) __(c)__ Remove all the nodes from an AVL Tree (as is done by the destructor).

(MC 17) __(c)__ Compute the height of every subtree in a Binary Search Tree.

(MC 18) __(c)__ Given a Binary Search Tree whose subtree heights have been computed determine if the Binary Search Tree is an AVL Tree.

3. **[Quadtrees – 30 points].**

For this question, consider the following partial class definition for the Quadtree class, which uses a quadtree to represent a d-by-d square bitmap image as in MP5. As a simplifying assumption for this problem, you may assume that only leaf nodes contain valid `RGBApixel` elements, and that the `element` field in all non-leaf nodes is not initialized to any particular value (we will not be doing any pruning of this tree, though it may already have been pruned). Just as in MP5, you may assume that d is a power of 2.

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

   void buildTree(BMP const & source, int newResolution);
   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:
     struct 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(RGBApixel const & elem);
        QuadtreeNode();
     };

     QuadtreeNode* root;    // ptr to root of quadtree, NULL if tree is empty.
     int resolution; // number of pixels on a side of the image (assume 2^k)

     void copy(QuadtreeNode *& firstNode, QuadtreeNode * secondNode);
     void clear(QuadtreeNode *& curNode);
     // plus many helper functions of your own design

};
```

You may assume that each child pointer in a new `QuadtreeNode` is NULL, and that the `Quadtree` has been accurately constructed so that the `resolution` is set.

(a) (3 points) Write the `getPixel` function declared in the public section above. This function takes two arguments,`x` and `y`, in that order, both nonnegative integers. It returns the RGBApixel corresponding to the pixel at coordinates (x, y) in the bitmap image which the quadtree represents (where $(0, 0)$ is in the upper left corner of the image). Note that the quadtree may not contain a node specifically corresponding to this pixel since the tree may have been pruned. In this case, `getPixel` retrieves the pixel (i.e. the color) of the smallest square region within which point (x, y) would lie. If the supplied coordinates fall outside of the bounds of the underlying bitmap, or if the current `Quadtree` is "empty" then the returned `RGBApixel` should be the one which is created by the default RGBApixel constructor. Write the function as it would appear in the quadtree.cpp file for the `Quadtree` class. (Note: part (b) of this problem asks you to write a helper function that you may call here.)

```
RGBApixel Quadtree::getPixel(int x, int y) const {  //+0.5pt for definition
                            //including const
   if (x >= resolution || y >= resolution          //+0.5pt for conditional
|| root == NULL) //+0.5pt for null check
      return RGBApixel(); //+0.5pt for RGBApixel()

   return getPixel(root, x, y, resolution); //+0.5pt for two return
//statements and
//argument correctness
//+0.5pt for overall
//correctness (typos,
//pointers and memory
//leaks, etc.
}
```

(b) (7 points) Write the private helper function that actually does the work for the `getPixel` function in part (a). (Note that if your approach to this problem is markedly different than ours, just write your solution on the back of the previous page. Your entire solution to `getPixel` will then be graded out of ten points.)

```
/* The rst time getPixel is called d's value is that of the quadtree?s resolution */
RGBApixel Quadtree::getPixel(int x, int y, QuadtreeNode* current, int d) const {
  if (current == NULL) {
    /* If the current pointer is NULL the tree was never built and we need to
     * return a default RGBApixel. */
    RGBApixel temp;
    return temp;
  } else if (current->nwChild == NULL) {
    /* If the nwChild is NULL we are at a leaf node and we can't recurse down
     * any further so return the leaf's element RGBApixel. */
    return current->element;
  }
  /* Chop the current resolution in half and use it to compare against the x and
   * y coordinates */
  d = d / 2;
  /* (d, d) is the center point of the image and we use this center point to
   * gure out which child we need to recurse on next.  Additionally, if we
   * are recursing into the East and/or South part we need to adjust the x
   * and/or y. */
  if (x < d  && y < d) {  // Northwest
    return getPixel(x, y, current->nwChild, d);
  } else if (x >= d && y < d) {  // Northeast
    return getPixel(x-d, y, current->neChild, d);
  } else if (x < d && y >= d) {  // Southwest
    return getPixel(x, y-d, current->swChild, d);
  } else {  // Southeast
    return getPixel(x-d, y-d, current->seChild, d);
  }
}
```

Rubric:

+2 Points for Comments

+1 Point for the method signature

+4 Points for method implementation

Partial credit was given for incomplete and/or inaccurate comments, signature, and method implementation. This is not the only implementation for getPixel but probably the most common one. Credit was given for other correct implementations.

(c) (5 points) Analyze and give a tight asymptotic bound on the running time of your implementation for part (b). Your bound should be stated in terms of d, the number of pixels in the width (or height) of the original image represented by the `Quadtree`. Briefly justify your answer.

The running time of the getPixel helper function is $O(\log d)$ because the algorithm's running time is determined by the height of the tree. That is the algorithm starts at the Quadtree?s root node and then recurses until it reaches a leaf node. At each recursive call it does $O(1)$ work to check to see if it is at a leaf node and then to gure out where it should recuse next. A recurrence for the method would be as follows:

$$T(1) = 1$$

$$T(d) = T(d/2) + O(1)$$

Using unrolling we can see a pattern.

$$= T(d/4) + O(1) + O(1)$$
$$= T(d/8) + O(1) + O(1)$$
$$= T(d/16) + O(1) + O(1) + O(1)$$
$$= ....$$
$$= T(d/2^h) + O(1)$$

So the recurrence bottoms out when $d/2^h = 1$, that is when $h = \log_2 d$.

+3 Points for a correct running time

+2 Points for a correct explanation

(d) (2 points) Suppose a pruned Quadtree has height $h \geq 0$. What is the least number of nodes it contains in terms of h? (Briefly explain or draw a helpful sketch to assure partial credit, and be exact.)

Solution: The least number of nodes in a tree of height h is $4h + 1$. It happens when, for each level, only one of the four children has descendants.

1 pt for showing the case graphically but fail to get the correct answer

(e) (2 points) Suppose a `Quadtree` has n nodes. What is its greatest possible height in terms of n? (Please be exact.)

Solution: The greatest possible height for a Quadtree with n nodes is $\frac{n?1}{4}$. It happens exactly the same case as discussed in the last subproblem. A oor function is not necessary since the quantity $\frac{n?1}{4}$ is always an integer due to the structure of Quadtree.

1 pt for showing the case graphically but fail to get the correct answer

3 pts total for the two problems for providing answers like $3h+1$ and $\frac{n-1}{3}$, or $4h-1$ and $\frac{n+1}{4}$.

(f) In this part of the problem we will derive an expression for the *maximum* number of nodes in a `Quadtree` of height h, and prove that our solution is correct. Let $N(h)$ denote the maximum number of nodes in a `Quadtree` of height h.

i. (3 points) Give a recurrence for $N(h)$. (Don't forget appropriate base case(s).)

$$N(h) = 4 * N(h-1) + 1; (\texttt{2pts})$$

$$N(0) = 1; (\texttt{1pt})$$

$$N(-1) = 0;$$

Number of nodes at h = number of nodes at 4 subtrees at (h-1) + root

We solved the recurrence and found a closed form solution for $N(h)$ to be:

$$N(h) = \frac{4^{h+1} - 1}{3}, h \geq -1$$

ii. (8 points) Prove that our solution to your recurrence from part (a) is correct by induction:
Consider a maximally sized `Quadtree` of arbitrary height h.

- If $h = -1$ then the expression above gives: __0__ which is the maximum number of nodes in a tree of height -1 (briefly explain).
- otherwise, if $h > -1$ then by an inductive hypothesis that says:

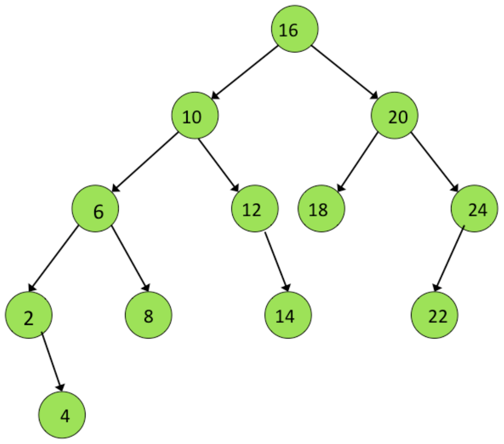
Suppose $N(k) = (4^{(k+1)} - 1)/3$ for all $k < h$.

we have $N(\underline{\ h-1\ }) = \underline{(4^{((h-1)+1)} - 1)/3}$ nodes.

so that $N(h) = \underline{4 * (4^{((h-1)+1)} - 1)/3 + 1} = \underline{\frac{4^{h+1}-1}{3}}$, which was what we wanted to prove.

4. **[AVL Trees – 15 points].**

Parts (a) through (e) below refer to this AVL Tree: (Note that no part is dependent on any other–each operation is performed on the original tree.)



(a) (3 points) If key 15 is inserted into the tree, the result will be a <u>left</u> rotation about the node containing key <u>12</u>.

(b) (3 points) If keys 0 and 1 are inserted into the tree, in that order, the result will be a <u>right</u> rotation about the node containing key <u>6</u>.

(c) (3 points) If key 10 is removed from the tree, the result will be a <u>left-right</u> rotation about the node containing key <u>6</u>.

(d) (3 points) Name a key, that when removed results in a right rotation: <u>12/14/16/18/20/22/24</u>.

(e) (3 points) Name a key, that when removed results in a right-left rotation, followed by a right rotation: <u>18/20</u>.

5. [**Tricky – 10 points**].

Here are the `Stack` and `Queue` class definitions you used in MP4:

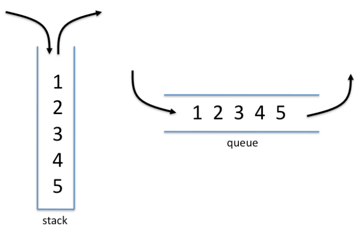
```
template<class T> class Stack {
public:
   void push(T const & newItem);  // adds new item to top of stack
   T pop();  // removes top item from the stack and returns its value
   T peek() const;  // returns value of top item on stack, but does not remove
   bool isEmpty() const;  // returns true if stack is empty, false o/w
private:  ... // not relevant here
};

template<class T> class Queue {
public:
   void enqueue(T const & newItem);  // adds new item to back of queue
   T dequeue();  // removes front item from the queue and returns its value
   T peek() const;  // returns value of front item on queue, but does not remove
   bool isEmpty() const;  // returns true if queue is empty, false o/w
private:  ... // not relevant here
};
```

Write the recursive function `verifySame` whose function prototype is below. The function should return `true` if the parameter stack and queue contain only elements of exactly the same values in exactly the same order, and `false` otherwise (see example below). *You may assume the stack and queue contain the same number of items!*

We're going to constrain your solution so as to make you think hard about solving it elegantly: 1) Your function may not use any loops; 2) In your function you may only declare ONE local boolean variable to use in your return statement, and you may only declare TWO local variables of parameterized type `T` to use however you wish–no other local variables can be used; and 3) After execution of `verifySame`, the stack and queue must be unchanged. Be sure to comment your code VERY well.

Example: This stack and queue are considered to be the same. Note that we match the bottom of the stack with the front of the queue. No other queue matches this stack.



```
template <class T> bool verifySame(Stack<T> & s, Queue<T> & q) {
    bool retval = true;
    if (!(s.empty())) {
        T stackTemp = s.pop();
        retval = verifySame(s,q) && (stackTemp == q.peek());
        q.enqueue(q.dequeue());
        s.push(stackTemp);
    }
    return retval;
}
```

Rubric:

Comments: 2 points for good comments, 1 point for sparse comments, 0 points for no comments

Recurssion Correctness: 1 point for correctly terminating recursion, 0 points for infinite recursion or the use of Loops

Parameters Intact: 2 points for both s and q left intact, 1 point if one of them is reversed and the other is fine, 0 points otherwise

Correct Local Variables: 1 point if they only use only 2 T's and 1 bool, 0 otherwise

Correct Output: 4 points if output is always correct, 2 points if output is correct part of the time, 1 point if they compare the wrong elements (top of stack with front of queue, etc), and 0 points if the output is totally wrong.

Syntax errors: -.5 points, up to - 2 points, never going below a total of 0.

scratch paper