University of Illinois at Urbana-Champaign
Department of Computer Science

Second Examination

CS 225 Data Structures and Software Principles
Spring 2011
7p-9p, Tuesday, April 5

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 17 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	25		
4	15		
5	10		
Total	100		

1. **[Miscellaneous – 25 points].**

MC1 (2.5pts)

Suppose a binary tree holds 127 keys. Then our node-based implementation of that tree has how many `NULL` pointers?

(a) 64

(b) 128

(c) 256

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

(e) None of these is the correct response.

MC2 (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) 50 120 160 172 183 205 200 230

(c) 12 14 20 22 24 27 40 45

(d) More than one of these are not valid inOrder traversals.

(e) All of these are valid inOrder traversals.

MC3 (2.5pts)

How many data structures in the following list are *dictionaries*?

stack   queue   binary search tree   AVL tree   B-Tree

(a) 1

(b) 2

(c) 3

(d) 4

(e) 5

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) {
   Iter fast = front;  Iter slow = front;

   while (fast != end) {
      fast++;  fast++;  slow++;
   }

   while (slow != front) {
      fast--;  slow--;
      doSomething(*fast, *slow);
    }
}

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

int main() {

   vector<int> v;
   ... // insert an even number of elements into the vector

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

   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.

(b) If the vector consists of the integers 1, 2, 3, 4, 5, 6 *in that order, with* `v[0] == 1`, then the output is 1, 2, 3, 4, 5, 6.

(c) If the vector consists of the integers 1, 2, 3, 4, 5, 6 *in that order, with* `v[0] == 1`, then the output is 4, 5, 6, 1, 2, 3.

(d) If the vector consists of the integers 1, 2, 3, 4, 5, 6 *in that order, with* `v[0] == 1`, then the output is 6, 5, 4, 3, 2, 1.

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

MC5 (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) 1

(b) 2

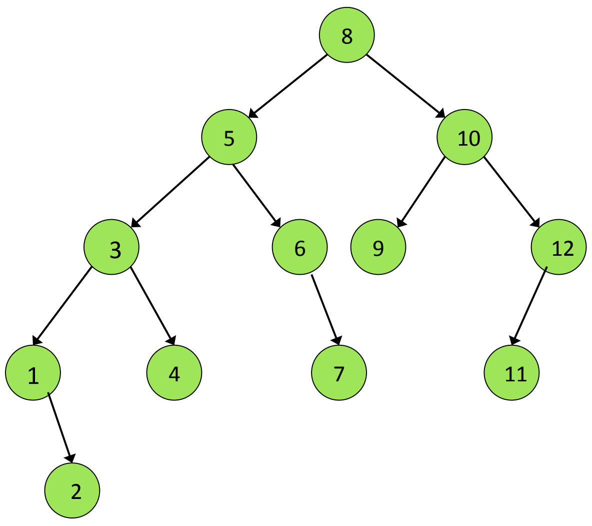
(c) 3

(d) 4

(e) 5 or more

MC6 (2.5pts)

Suppose we remove the node containing key 10 from the AVL tree below.



What sequence of rotations will restore the balance of the tree? (Assume IOP is used for 2-child removal.)

(a) `leftRotate` about 9, followed by `rightRotate` about 8.

(b) `rightRotate` about 8.

(c) `leftRotate` about 12, followed by `rightRotate` about 8.

(d) `rightLeftRotate` about 9, followed by `rightRotate` about 8.

(e) None of these choices will rebalance the tree.

MC7 (2.5pts)

Examine the `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`.

MC8 (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.

MC9 (2.5pts)

What is the minimum number of keys that can be stored in a B-Tree of order 64 and height 5?

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

(b) $2^{25} + 1$

(c) $2^{26} - 1$

(d) $2^{30} - 1$

(e) $2^{30} + 1$

Correction: The exams were originally graded with choice **d** as the correct answer. However, the correct answer is actually choice **c**. The exam scores will be corrected to reflect this (e.g., if you choose choice **c** you will get the points. Otherwise, you will not get the points.).

MC 10 (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 $O(m \log_m n)$ and this bounds the total number of disk seeks.

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

(iii) Every AVL 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 statements are true.

(e) All statements (i), (ii), and (iii) are false.

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) $O(n^2)$

(MC 11) __a__ `Enqueue` for a `Queue` implemented with an array.

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

(MC 13) __c__ Find a 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 largest key in a Binary Search Tree.

(MC 16) __b__ Find the largest key in an AVL Tree.

(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.

(MC 19) __c__ Determine if a binary tree is a Binary Search Tree.

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

3. **[Quadtrees – 25 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 all non-leaf nodes' `RGBApixel` elements already hold the component-wise average of their children's `RGBApixel` elements, and that the `Quadtree` resolution is set to *d*. Just as in MP5, you may assume that *d* is a power of 2.

```
class Quadtree
{
  public:
    // constructors and destructor, big three, other public methods

    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 * 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 pruneHelper(QuadtreeNode * croot, tolerance);
    void clear(QuadtreeNode *& croot);
    int  squaredDist(const RGBApixel & a, const RGBApixel & b);
    int  max(int a, int b, int c, int d)

    // other helper functions
};
```

(a) In this part you will implement a helper to the `prune` function above (`pruneHelper`).

A node n can be pruned if there is no leaf in the subtree rooted at node n which has a color whose squared distance from the color of n in RGB-space is greater than tolerance. If a node n can be pruned, the subtrees rooted at the children of n are cleared from the `Quadtree`.

Background information: Here we have given you an implementation of the `prune` function in the public portion of the `Quadtree` definition above. You will be writing the function `pruneHelper` called here.

```
void Quadtree::prune(int tolerance)
{
        if (root == NULL) return;
        pruneHelper( root, tolerance );
}
```

If you have an implementation that differs significantly from the code provided here, you may change the above function and parameters to `pruneHelper`.

`pruneHelper` may call the provided functions in the `Quadtree` class above. Specifically, these may be useful:

`void clear(QuadtreeNode * & croot):`
This function clears a node and all descendent nodes in this node's subtree. It also deletes `croot` itself and sets croot to point to NULL.

`int squaredDist(const RGBApixel & a, const RGBApixel & b):`
This function returns the squared distance in RGB-space between `RGBApixels` `a` and `b`.

`int max(int a, int b, int c, int d):`
This function returns maximum of 4 integers.

i. (10 points) Write the main private helper function that actually does the work for the `prune` function in part (a). Your solution will be graded on comments and efficiency, in addition to correctness. Note that there is room for other helper functions on the next page. *Note: This is a long question. You may want to move on to other questions and come back to this part.*

```
void Quadtree::pruneHelper(QuadtreeNode * croot, int tolerance) {
  if (croot == NULL) {  // no nodes to prune
    return;
  } else if (croot->nwChild == NULL) {  // no children to prune
    return;
  }

  if (canPrune(croot, croot->element, tolerance)) {
    /* If we can prune clear all the subtrees,
     * and set their pointers to NULL */
    clear(croot->nwChild);
    clear(croot->neChild);
    clear(croot->swChild);
    clear(croot->seChild);
  } else {
    /* Else if we can prune at this node.
     * Try pruning the root of the subtrees
     * by recursing on them */
    pruneHelper(croot->nwChild);
    pruneHelper(croot->neChild);
    pruneHelper(croot->swChild);
    pruneHelper(croot->seChild);
  }
}
```

You may implement any other helper functions here. As a suggestion, we have a single stub function below, but you may implement more helper functions if you need to. If you run out of space, you may put a note here and continue your solution on the back of this page.

```
bool Quadtree::canPrune(QuadtreeNode * croot,  // current node
                        const RGBApixel & rootColor,  // root's color
                        int tolerance) {
  /* If we are at a leaf check if it is within the tolerance
  if (croot->nwChild == NULL) {
    return squaredDist(croot->element, rootColor) < tolerance;
  }
  /* Else recurse.
   * If all subtrees can prune (i.e., their leaves are within the
   * tolerance) then this function can prune. */
  bool nwCanPrune = canPrune(croot->nwChild, rootColor, tolerance);
  bool neCanPrune = canPrune(croot->neChild, rootColor, tolerance);
  bool swCanPrune = canPrune(croot->swChild, rootColor, tolerance);
  bool seCanPrune = canPrune(croot->seChild, rootColor, tolerance);
  return (nwCanPrune && neCanPrune && swCanPrune && seCanPrune);
}
```

- +1 for proper base cases [pruneHelper]
- +1 point for $\leq$ tolerance (i.e., instead of $<$ tolerance) [canPrune]
- +1 for pruning / attempting to prune something [pruneHelper]
- +1 for pruning the whole tree [pruneHelper]
- +1 for using && or max correctly [canPrune]
- +2 for checking leaf nodes against the current node's color [canPrune]
- +1 for clearing the node's children not the node itself [pruneHelper]
- +1 points for a correct (and non-iterative) prune over the whole tree
- +1 points for comments

(b) In this part we will examine the relationship between n, the number of nodes in an unpruned quadtree, h, the height of the quadtree.

i. (1 point) Let $N(h)$ denote the number of nodes in an unpruned `Quadtree` of height h. Give a recurrence for $N(h)$

Solution: Depending on whether you think recursively, there are two alternative solutions for this problem. If you do, then

$$N(h) = 4N(h-1)+1$$

Or you can also write down something like

$$N(h) = N(h-1)+4^h$$

Both should start with $N(0)=1$.

ii. (2 points) Solve this recurrence (i.e. by unrolling) to get a tight Big-O bound on the number of nodes in a `Quadtree` of height h

Solution: Depending on your result in subproblem (i), you can either

$$\begin{aligned} N(h) &= 4N(h-1)+1 \\ N(h)+\frac{1}{3} &= 4(N(h-1)+\frac{1}{3}) \\ N(h) &= 4^h * (N(0)+\frac{1}{3}) - \frac{1}{3} = \frac{4^{h+1}-1}{3} \end{aligned}$$

Or

$$\begin{aligned} N(h) &= N(h-1)+4^h \\ N(h) &= \sum_{i=0}^{h} 4^i \\ N(h) &= \frac{1-4^{h+1}}{1-4} = \frac{4^{h+1}-1}{3} \end{aligned}$$

Therefore

$$N(h) = \mathcal{O}(4^h)$$

$\mathcal{O}(2^h)$ is incorrect as there does not exist a M such that $|\frac{4^{h+1}-1}{3}| \leq M|2^h|$ as h goes to infinite.

iii. (2 points) Give a tight Big-O bound on the height of a `Quadtree` with n nodes

Solution: Since $N(h) = \frac{4^{h+1}-1}{3}$, we have $h = \log_4(3n+1)$, namely

$$h = \mathcal{O}(\log_4(3n+1)) = \mathcal{O}(\log_4(n)) = \mathcal{O}(\log(n))$$

(c) Give the runtimes of the following functions from MP 5, **in terms of** n, the number of nodes in the Quadtree. Assume all functions are called on an unpruned Quadtree. Be sure to give brief justifications of your responses.

i. (2 points) `buildTree`: This function takes two arguments, a BMP object by reference and an integer d, and creates a Quadtree representing the upper-left d by d block of the BMP.

   $O(n)$, build a tree with the leaves representing the image pixels and internal nodes

ii. (2 points) `getPixel`: This function takes two arguments, x and y, and returns the RGBApixel corresponding to the pixel at coordinates (x, y) in the bitmap image which the quadtree represents.

   $O(log(n))$, proportional to the height of the tree, find a node in a height-balanced tree

iii. (2 points) `decompress`: This function returns the underlying BMP object represented by the quadtree.

   $O(n)$, recursive traversal of the tree, constant operation (retrieval of pixel value and storing it in BMP image object) on each node in the tree $O(n * log(n))$ with an explanation of using getPixel() function received a partial credit

iv. (2 points) `clockwiseRotate`: This function rotates the Quadtree object's underlying image clockwise by 90 degrees.

   $O(n)$, recursive traversal of the tree, constant operations (pointer reassignments) on each node in the tree

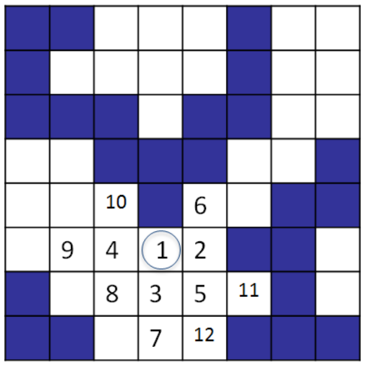
v. (2 points) `prune`: This function is described in part (a) of the problem.

   $O(n * log(n))$, recursive traversal of all nodes in the tree + comparison operations against all leaves of each internal node

- +2 for a correct and complete answer
- +1 for either correct running time but no explanation, or a running time for not best possible implementation
- +0 for incorrect running time (even if the explanation is somewhat correct)

4. **[Flood Fill – 15 points].**

(a) (4 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.

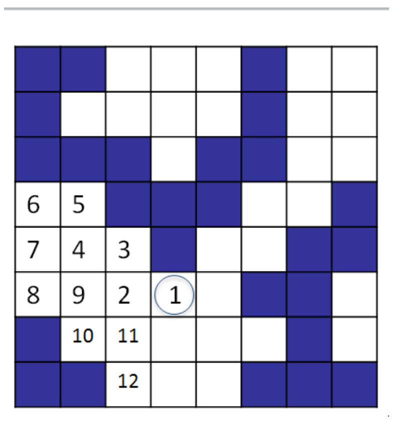


- +4 points if perfect
- +3 points if not more that 2 numbers mere wrongly placed
- +2 points if a lot of mistakes but have got the general idea of the rotation right
- +1 point if concept is clear
- +0 if totally off.

(b) (4 points) Suppose we want to fill some part of a arbitrary n-by-n image. What is the worst-case running time of `BFSfillSolid` 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.

- +4 points if answer is $O(n^2)$
- +2 points if anything else but have given a good explanation
- +0 otherwise

(c) (4 points) Suppose we execute function `DFSfillSolid`, 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.



- +4 points if perfect
- +3 points if not more that 2 numbers mere wrongly placed
- +2 points if a lot of mistakes but have got the general idea of the rotation right
- +1 point if concept is clear
- +0 if totally off

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

- +3 if answer is stack
- +3 if answer is queue with queue implementation in part 3
- +0 otherwise

5. **[AVL Tree Height – 10 points].**

In this problem you will complete the proof of the following theorem:

The height of an n-node AVL Tree is $O(\log n)$.

Fill in the blanks to complete the proof.

Preliminaries: Let $H(n)$ denote the maximum height of an n-node AVL Tree, and let $N(h)$ denote the minimum number of nodes in an AVL tree of height h. To prove that $H(n) = O(\log n)$, we argue that

$$H(n) \leq 2\log_2 n, \text{ for all } n.$$

Rather than prove this inequality directly, we'll show equivalently that

$$N(h) \geq \underline{\quad 2^{h/2} \quad}.$$

Proof: For an arbitrary value of h, the following recurrence holds for all AVL Trees:

$$N(h) = \underline{\quad 1 \quad} + \underline{\quad N(h-1) \quad} + \underline{\quad N(h-2) \quad}$$

$$\text{and } N(0) = \underline{\quad 1 \quad}, N(1) = \underline{\quad 2 \quad}$$

This expression for $N(h)$ simplifies to the following inequality which is a function of $N(h-2)$:

$$N(h) \geq \underline{\quad 2 \quad} * \underline{\quad N(h-2) \quad}$$

By an inductive hypothesis which states:

for all j less than h:

$$\underline{\quad N(j) \geq 2^{h/2} \quad}$$

we now have:

$$N(h) \geq 2 * \underline{\quad 2^{(h/2)-1} \quad} = \underline{\quad 2^{(h/2)} \quad},$$

which was what we wanted to prove.'

Each line is worth 1 point except for the recurrence relation and the inductive hypothesis which are each 3 points. The inductive hypothesis need not be strong, just as long the base cases given are correct.

scratch paper