University of Illinois at Urbana-Champaign
Department of Computer Science

Second Examination RUBRIC

CS 225 Data Structures and Software Principles
Spring 2013
7-10p, Tuesday, April 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 18 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.
- Please put your name at the top of each page.

Problem	Points	Score	Grader
1	25		scantron
2	25		scantron
3	17		
4	10		
5	23		
Total	100		

1. **[Miscellaneous – 25 points].**

MC1 (2.5pts)

Consider a sequence of push and pop operations used to push the integers `0` through `9` on a stack. The numbers will be pushed in order, however the pop operations can be interleaved with the push operations, and can occur any time there is at least one item on the stack. When an item is popped, it is printed to the terminal.

Which of the following could NOT be the output from such a sequence of operations?

(a) `0 1 2 3 4 5 6 7 8 9`
(b) `4 3 2 1 0 5 6 7 8 9`
(c) `5 6 7 8 9 0 1 2 3 4`
(d) `4 3 2 1 0 9 8 7 6 5`
(e) All of these output sequences are possible.

MC2 (2.5pts)

Suppose you implement a `queue` with a singly linked list containing both head and tail pointers. In order to achieve $O(1)$ running times for both the `enqueue()` and `dequeue()` functions you place the entry to the `queue` at the _____________.

Choose the best answer to fill in the blank.

(a) "head of the list"
(b) "tail of the list"
(c) Both (a) and (b) are correct.
(d) Neither (a) nor (b) is correct.
(e) It is not possible to implement a `queue` with a singly linked list in such a way that both `queue` operations are constant time.

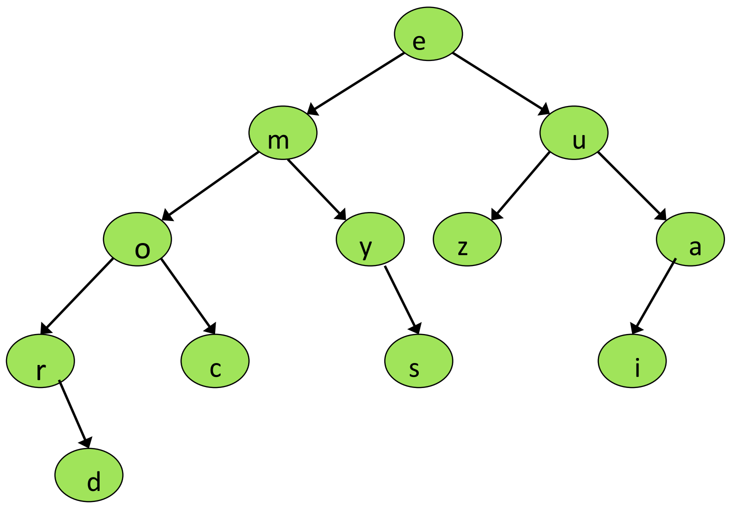
MC3 (2.5pts)

Which of the following sequences of keys is the inOrder traversal of a BST?

(a) 1 3 6 8 12 15 17 19
(b) 50 120 40 130 30 120 20 110
(c) 20 15 10 5 25 30 35 40
(d) More than one of these are VALID inOrder traversals.
(e) All of these are INVALID inOrder traversals.

MC4 (2.5pts)

Consider a level order traversal of the following binary tree. Which node is the last node *dequeued* before the node containing `y` is *enqueued*?



(a) The node containing `c`.

(b) The node containing `o`.

(c) The node containing `m`.

(d) The node containing `s`.

(e) None of these is the correct answer.

MC5 (2.5pts)

How many data structures in this list can used to implement a *Dictionary* so that all of its functions have *better* than $O(n)$ running time (worst case)?

stack   queue   binary tree   binary search tree   AVL tree

(a) 1

(b) 2

(c) 3

(d) 4

(e) 5

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 could invoke `mysteryFunction`?

(a) `insert(key);`

(b) `remove(key);`

(c) `find(key);`

(d) Exactly two of these could invoke `mysteryFunction`.

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

MC7 (2.5pts)

Consider the Binary Search Tree built by inserting the following sequence of integers, one at a time, in the given order: $5, 4, 7, 9, 8, 3, 1$. What is the height of the tree?

(a) 2

(b) 3

(c) 4

(d) 5

(e) We do not have enough information to answer this question.

MC8 (2.5pts)

Consider the following recursive C++ function, and assume our standard node-based implementation of the `BinaryTree` class which includes a private definition of a `Node` class.

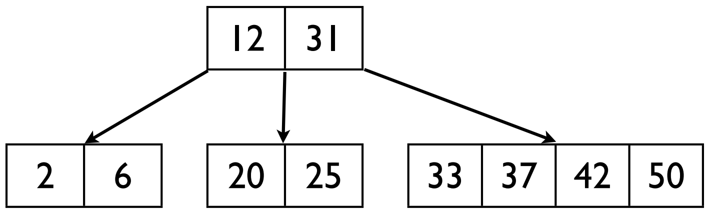
```
template <typename T>
typename BinaryTree<T>::Node * BinaryTree<T>::unk(Node * cRoot)
{
     if (cRoot != NULL) {
          Node * temp    = cRoot->left;
          cRoot->left  = unk(cRoot->right);
          cRoot->right = unk(temp);
     }
     return cRoot;
}
```

Among the following choices, print the best name for the function. (That is, we are asking for a name that describes what the function does.)

(a) **`findMirror` - creates the mirror image of the binary tree.**
(b) `copyTree` - makes a copy of the binary tree.
(c) `goLevel` - traverses each node in the tree exactly once in level order from bottom to top.
(d) `rightRotate` - performs a right rotation around `cRoot`.
(e) None of these options is a good name for the function.

MC9 (2.5pts)

Consider the BTree in the figure below.



How many disk seeks are required during the execution of `Find(42)`? Please assume that none of the data exists in memory when the function call is made.

(a) 1
(b) **2**
(c) 4
(d) 5
(e) The number of disk seeks cannot be determined because we do not know the order of the tree.

MC 10 (2.5pts)

Use the following 3 code examples to answer the question below. Please assume that all arrays and images have been properly initialized to hold valid data.

(i)
```
int table [10][10];
loadInts(table); // initialize array values
#pragma omp parallel for
for(int i = 0; i < 9; i++)
     for(int j = 0; j < 9; j++)
          table[i][j] = table[i+1][j+1];
```

(iI)
```
RGBAPixel colorArray[100];
loadColors(colorArray); // initialize array values
RGBAPixel temp = colorArray[20];
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
     colorArray[i].red = temp.red * (0.9);
}
```

(iIi)
```
PNG image("dogPic.PNG");  // load from file
RGBAPixel temp;
#pragma omp parallel for
for (int i = 0; i < image.width(); i++) {
     for (int j = 0; j < image.height()/2; j++) {
            temp = *image(i, j);
            *image(i, j) = *image(i, image.height() - 1 - j);
            *image(i, image.height() - 1 - j) = temp;
     }
}
```

Which of the code examples above is/are correctly parallelized?

(a) All statements (i), (ii), and (iii) are incorrect.

(b) Only item (i) is correct.

(c) Only item (ii) is correct.

(d) Only item (iii) is correct.

(e) Two of the above examples are correct.

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 worst case 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. 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) <u>C</u> Total cost of a sequence of n `enqueue` operations on a `queue` that is implemented with an array. You may resize the array however you like.

(MC 12) <u>A</u> `Pop` for a `Stack` implemented with an array.

(MC 13) <u>C</u> Find the maximum key in a Binary Tree (not necessarily BST).

(MC 14) <u>C</u> Insert a key into a Binary Search Tree.

(MC 15) <u>B</u> Find the *In Order Predecessor* of a given key in a height balanced Binary Search Tree (if it exists).

(MC 16) <u>C</u> Count the number of leaves that are greater than a given key in a Binary Search Tree.

(MC 17) <u>A</u> Perform `rightLeftRotate` around a given node in an AVL Tree.

(MC 18) <u>C</u> Determine if a given Binary Search Tree is an AVL Tree.

(MC 19) <u>D</u> Build an AVL tree with keys that are the numbers between n and 0, in that order, by repeated insertions into the tree.

(MC 20) <u>C</u> Perform a level order traversal of an AVL Tree.

3. **[MP4ish – 17 points].**

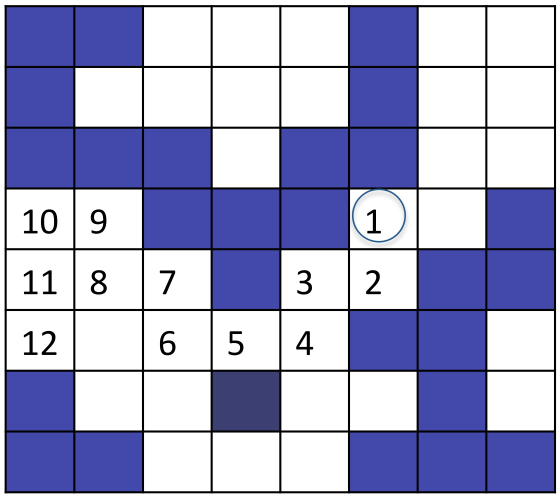
   In MP4 we asked you to implement the `Stack` and `Queue` data structures, and use them to facilitate a variety of flood fill algorithms. This problem will revisit several components of that assignment. In each part, be careful to notice which fill, *Depth*-first or *Breadth*-first, we're asking you about.

   (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 as it would appear in queue.cpp. 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. Your algorithm will be graded for running time efficiency.

```
T Queue<T>::dequeue()
{
    if(outStack.isEmpty())         1 point
    {
      while(! inStack.isEmpty())        1 point
         {
             outStack.push(inStack.pop());    2 points
         }
    }
return outStack.pop();         1 point
}
```



(b) (4 points) Suppose we execute function `DFSfillSolid`, one of two main fill routines from MP4, on the above 8x8 pixel image, beginning at the circled node, (5,3), 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 subsequently add the four neighboring pixels to the structure clockwise beginning on the right, followed by down, left, and up, if appropriate. As a reminder, the fill should change the color when a cell is *removed* from the ordering structure.

RUBRIC:
Full points if correct.
2 points deducted if the order is messed up halfway through.
No points if completely wrong.

(c) (3 points) Suppose we want to fill some part of a arbitrary image containing n pixels. What is the worst-case running time of `DFSfillSolid` if we start from an arbitrary location? Please give your running time in terms of n, the number of pixels in the image.

Solution: O(n), Rubric: all or nothing.

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

Solution: Stack, Rubric: all or nothing.

(e) (2 points) Suppose we implemented the data structure you chose in part (d) using a dynamic array. Briefly explain what we would do if that array filled during the execution of the algorithm?

1) Make a new dynamic array of twice the size, and 2) copy all the elements into the new array. (1 point each)

4. **[Quadtrees – 10 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 (percent is between 0 and 1)
   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, that it has been built from an image that has size $2^k \times 2^k$, and that the `resolution` member variable has been initialized by the constructor to be the number of pixels on one side of the image. 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) (5 points) Write a *private* member function `int Quadtree::tallyNeighbors(RGBApixel const & target, QuadtreeNode const * curNode, int tolerance)`, which calculates and returns the number of leaves in the tree rooted at `curNode` with element less 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::tallyNeighbors(RGBApixel const & target,
           QuadtreeNode const * curNode, int tolerance) _const_ {

   // function not called with curNode == NULL;
   if (curNode->_nwChild_ == _NULL_) {  // check for leaf

      RGBApixel current = curNode->element;

      if (distance(current, target) _>_ tolerance)   return _1_;
      else return 0;

   }

   // otherwise...recurse!
   int devTotal = _tallyNeighbors(target, currNode->nwChild, tolerance) + ... +
      tallyNeighbors(target, currNode->seChild, tolerance)_


   return _devTotal_;
}
```

Rubric: Approximately 1 point per line containing blank(s).

(b) (5 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. This helper function will be called by the public version as: `prunish(root, tolerance, resolution, percent);`

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.

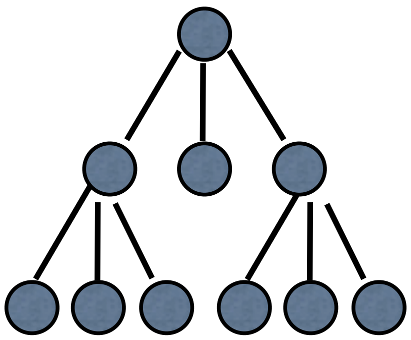
```
void Quadtree::prunish(QuadtreeNode * curNode, int tolerance,
                        int res, double percent) {

   if (curNode == NULL)
      return;

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

   int neighbors = _tallyNeighbors(curNode->element, curNode, tolerance)_; //(1 point

   double percentClose =_neighbors/res*res_; //(1 point) percent between 0 and 1

   // prune conditions
   if (percentClose _ >= percent_) { //(2 points)

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

      return;
   }

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

   _prunish(curNode->nwChild,tolerance,res/2, percent)_   //(1 point)

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

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

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

   return;
}
```

5. **[Ternary Trees – 23 points].**

A ternary tree is a rooted tree where each node has up to 3 children. More formally, a ternary tree tree T is a rooted tree where either T is empty, or T consists of a single node r together with subtrees T_L, T_M, and T_R, each of which is also a ternary tree.

(a) A ternary tree is said to be *full* if it is empty or if it consists of a single node together with subtrees T_L, T_M, and T_R, all of which are empty, or all of which are non-empty, and each of which is also a full ternary tree. (This is a natural extension of our definition of full binary trees.)

i. (3 points) Draw a full ternary tree with 10 nodes.



ii. (3 points) Let $M(h)$ denote the maximum number of nodes in a full ternary tree of height h. Write a recurrence for $M(h)$, including appropriate base case(s).
$M(h) = 3M(h-1) + 1, M(-1) = 0$.

iii. (6 points) A stranger on the street asserts that the solution to the above recurrence is:

$$M(h) = \frac{3^{h+1}-1}{2}, h \geq -1$$

Prove that the formula is correct using induction.
Fill in the blanks to complete the proof: Consider an arbitrary integer $h \geq -1$.

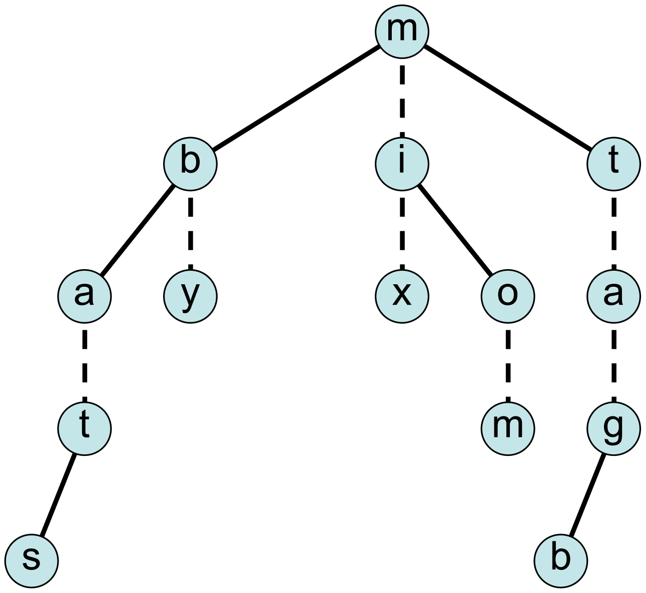
- If $h = -1$ then the closed form gives $M(h) = \underline{\ 0\ }$, which is also the base case of the recurrence.
- If $h \geq 0$, then by an inductive hypothesis that says,

$$\underline{\forall j < h, M(j) = \frac{3^{j+1}-1}{2}}$$

we know that $M(h-1) = \underline{\frac{3^h-1}{2}}$, so that substituting in to the recurrence gives us

$M(h) = \underline{3 \times \frac{3^h-1}{2} + 1} = \underline{\frac{3^{h+1}-1}{2}}$, which was what we wanted to prove.

(b) READ THIS CAREFULLY: A ternary search tree (TST) is used to store a dictionary of English words as shown in the following example. Each node contains a single character. Every path from the root to a node containing no middle child corresponds to a single word. The exact sequence of letters in such a path do not correspond exactly to the letters in a word. Rather, letters in the path that are followed by either a left or right traversal are NOT letters in the word. For example, consider the path in the tree below that traverses the nodes containing `m-t-a-g-b`. The letter `m` is not in the word because the next branch is to the right. The letter `g` is also not in the word because the next branch is to the left. All other letters are in the word, and as such, the word corresponding to that path is `tab`.



i. (3 points) The tree above contains the words `at`, `mix`, `tab` and 4 other english words. What are they?
Solution: as, mom, by, tag

Rubric: 1 correct : 1 point, 2/3 correct : 2 points, 4 correct : 3 points,

The TST can also be used as a query structure. In the next 2 parts of the problem you will write the code to determine if a given `string` is contained in the tree. The formal algorithm for function `bool TST::find(const string & query)` is:

In general you will process the string one character at a time, starting from the first letter in the word, and similarly process the tree one node at a time, beginning with the root. At each stage of the algorithm: If the current node character k matches the current string character c then search for the rest of the string in the middle subtree. Otherwise, search for the string in the left or right subtree, depending on whether c is less or greater than k, respectively. You have found a word if you have found all the characters of your string, and if the current node has no middle child.

(Note that our version of a TST is really a simplification because it only stores words which are not prefixes of other words in the dictionary. That is, our tree cannot contain the words “cap” and “cape” because cap is a prefix of cape. A slight modification of our structure would eliminate this constraint, but we’ll not bother with that here.)

Following is a partial class definition for the `TST` class. You may assume that the `TSTNode` constructor terminates subtrees with nulls, and that an empty `TST` is a `null TSTNode` pointer. Your job is to write the public and private versions of the `find` function which are member functions of the `TST` class. (As a hint, remember that a `string` object can be treated like an array of `char`s and that the `string` class provides a member function `length()` which returns the number of characters in the string.)

```
class TST {
public:
   ...
   bool find(const string & s);
   ...
private:
   class TSTNode {
   public:
      char k;
      TSTNode* left;
      TSTNode* mid;
      TSTNode* right;
      // constructors
   };

   TSTNode * root;
   bool find(const string & s, TSTNode * t, int strPos);
};
```

Be sure to let the graders know what you’re trying to do by commenting your code completely. Partial credit will be awarded for useful comments, even if the code is broken.

ii. (3 points) Write the public version of the `find` function here:

```
bool TST::find(const string & s)
{

                return find(s, root, 0);
   ------------------------------------------------

}
```

Rubric:
Correct : 3
Partial correct : 2/1

No clue : 0

iii. (5 points) (Note: As a syntax reminder, if `s` is a string, then `s[i]` gives the character in the `i`th position of the string.) Write the private version of the `find` function here:

```
bool TST::find(const string & s, TSTNode* t, int strPos)
{
   if (t == NULL && s.length() == strPos)
                             return true;
   else if(t == NULL && s.length() > strPos)
                             return false;
   else if (t != NULL && s.length() == strPos)
                             return false;
   else if (t->k == s[strPos])
                             return find(s, t->mid, strPos++);
   else if (t->k < s[strPos])
                             return find(s, t->right, strPos);
   else
                             return find(s, t->left, strPos);
}
```

Rubric:
Base cases handled : 3
All 3 recursive cases handled : 3
Overall syntax : 1

scratch paper