# CSCI-1200 Data Structures
# Test 3 — Practice Problems

*Note: This packet contains selected practice problems from Test 3 from three previous years. Your test will contain approximately one third as many problems (totalling ∼100 pts).*

## 1 Reducing Fractions with Pair Maps [ 30 pts ]

Let's construct a data structure that explicitly stores the correspondence between a fraction and its simplified or reduced form (e.g., $\frac{2}{6} \rightarrow \frac{1}{3}$ ). Here's the code to construct the first of two map data structures we use in this problem. Note that the STL `pair` struct interface and implementation does include the definition of the `operator<(const pair &a, const pair &b)` function, which returns true if the first element of `a` is less than the first element of `b`, and false if the first element of `b` is less than the first element of `a`. If neither of these is the case, then `operator<` returns the result of comparing the second elements of `a` and `b`.

```
typedef ***PART_1*** map1_type;
map1_type map1;
map1[std::make_pair(2,4)] = std::make_pair(1,2);
map1[std::make_pair(4,8)] = std::make_pair(1,2);
map1[std::make_pair(3,6)] = std::make_pair(1,2);
map1[std::make_pair(2,6)] = std::make_pair(1,3);
map1[std::make_pair(3,9)] = std::make_pair(1,3);
map1[std::make_pair(4,6)] = std::make_pair(2,3);
map1[std::make_pair(2,8)] = std::make_pair(1,4);
```

### 1.1 The `map1` Data Type [ 3 pts ]

What is the type for the `map1` data structure? Fill in the blank marked `***PART_1***`.

### 1.2 Visualizing the Data Structure [ 6 pts ]

Draw a picture to represent the `map1` data structure that has been constructed by the commands above. As much as possible use the conventions from lecture and lab for drawing these pictures. Please be neat when drawing the picture so we can give you full credit.

Now we'll convert the data to another format. In the second version, we want to associate the reduced form of the fraction with one or more unsimplified fractions. Here's code to declare the second map:

```
typedef ***PART_3*** map2_type;
map2_type map2;

// code to initialize map2 from the data stored in map1
*** PART 4 ***
```

| | |
|---|---|
| (1,2) | (2,4) (3,6) (4,8) |
| (1,3) | (2,6) (3,9) |
| (1,4) | (2,8) |
| (2,3) | (4,6) |

And on the right is a diagram of the `map2` data structure storing the information from the initial example.

## 1.3 The `map2` Data Type [ 4 pts ]

What is the type for the `map2` data structure? Fill in the blank marked `***PART_3***`.

## 1.4 Map Conversion [ 8 pts ]

Now write the fragment of code to fill in `*** PART_4 ***` that converts data stored in the variable `map1` into the second map data structure format, storing it in variable `map2`. Study the example above, but your code should work for all examples of this type.

*sample solution: 3 line(s) of code*

Let's say the `map1` data structure stores $n$ unreduced fractions, but when reduced there are only $m$ different fractions in reduced form, and the most common reduced form has $k$ unreduced fractions. What is the Big O Notation for the code you just wrote? Write 2-3 sentences justifying your answer.

### 1.5 Counting using `map1` [ 4 pts ]

Write a function named `count_reduce_to_map1` that takes 3 arguments: `map1` (of type `map1_type`), and 2 integers: `numer` and `denom`. The function should return the number of fractions stored in the map that reduce to the fraction $\frac{\texttt{numer}}{\texttt{denom}}$. For example, `count_reduce_to_map1(map1,1,3)` should return 2.

*sample solution: 8 line(s) of code*

In terms of $n$, $m$, and $k$ as defined above, what is the Big O Notation for the `count_reduce_to_map1` function? Write 1-2 sentences justifying your answer.

### 1.6 Counting using `map2` [ 5 pts ]

Now, write a very similar function named `count_reduce_to_map2` that uses `map2` instead of `map1`.

*sample solution: 8 line(s) of code*

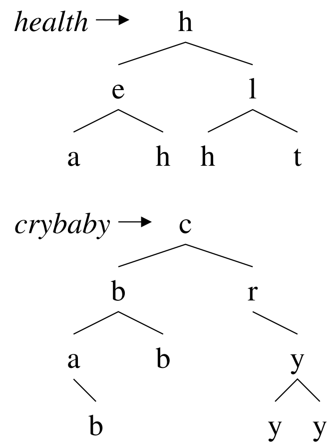
What is the Big O Notation for the `count_reduce_to_map2`? Justify your answer.

## 2 Word BST with Duplicates [ 20 pts ]

Write a function named `word_bst` that takes one argument, an STL `string` named `word`, and returns a pointer to the root `Node` of a binary search tree created by adding the characters of the input word in order. The tree will allow duplicate characters. When a repeated character is added, it is added to both the left and the right subtrees.

You should write a helper function and your solution should use recursion.

```
class Node {
 public:
  char value;
  Node* left;
  Node* right;
};
```




*sample solution: 21 line(s) of code*

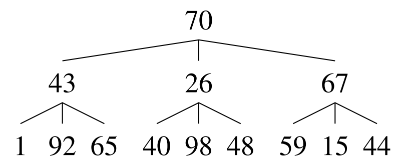
# 3 Constant Memory Breadth-First Traversal [ 31 pts ]

Ben Bitdiddle sat through the Data Structures lecture covering breadth-first tree traversal and is determined to implement a *constant memory implementation* breadth-first traversal of a *perfectly-balanced* trinary tree. To meet the constant-time memory goal, he realizes this means no `vector` or `list` helper variables may be used, and furthermore, the implementation cannot use recursion.

```
class Node {
public:
  int value;
  Node* left;
  Node* middle;
  Node* right;
  Node* parent;
};
```

Below is Ben's initial implementation. You'll write the missing functions and then analyze the algorithm for memory use and running time.

```
void BreadthTraversal(Node *root) {
  int level = 0;
  while (true) {
    Node* tmp = FirstNodeOnLevel(root,level);
    if (tmp == NULL) break;
    std::cout << "level " << level << ": ";
    int level_count = NumNodesOnLevel(level);
    for (int i = 0; i < level_count; i++) {
      std::cout << " " << tmp->value;
      tmp = NextNodeOnLevel(tmp);
    }
    std::cout << std::endl;
    level++;
  }
}
```




```
level 0:  70
level 1:  43 26 67
level 2:  1 92 65 40 98 48 59 15 44
```

## 3.1 Implement and Analyze `FirstNodeOnLevel` [ 6 pts ]

For the above example, when passed `level` = 0, it returns the `Node` storing 70. When passed `level` = 1, it returns the `Node` storing 43. When passed `level` = 2, it returns the `Node` storing 1, etc.

*sample solution: 5 line(s) of code*

For a tree with $n$ nodes: Big O Notation for memory: running time:

## 3.2 Implement and Analyze `NumNodesOnLevel` [ 6 pts ]

Implement the `NumNodesOnLevel` function. Level 0 has 1 node, Level 1 has 3 nodes, etc.

*sample solution: 5 line(s) of code*

For a tree with $n$ nodes: Big O Notation for memory: running time:

### 3.3 Implement and Analyze `NextNodeOnLevel` [ 16 pts ]

Finally implement the `NextNodeOnLevel` function that moves between nodes on a specific level of the tree.

*sample solution: 21 line(s) of code*

For a tree with $n$ nodes: Big O Notation for memory:

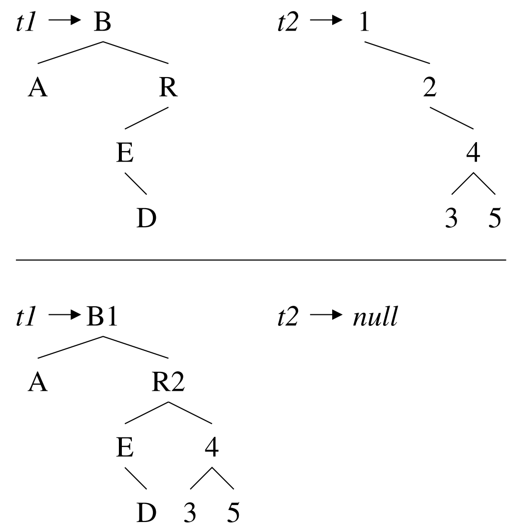
For running time, Best Case: For running time, Worst Case:

### 3.4 Analyze `BreadthTraversal` [ 3 pts ]

What is the overall memory usage and running time of the Breadth Traversal algorithm to print the entire tree? Write 2-3 concise and well-written sentences justifying your answer.

# 4 Tree Overlay [ 16 pts ]

Write a recursive function named `overlay` that takes in two trees, `t1` and `t2`, containing string data. The function will *overlay* and combine the data in the two trees. Where the shapes of the trees are similar, the value in the `t2` `Node` is concatenated to the value in the `t1` `Node`.

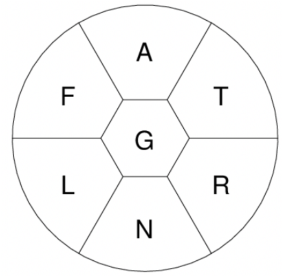



As shown in the example to the right, after the `overlay` function call, the combined tree is stored in the `t1` variable and the `t2` variable is an empty tree.

```
class Node {
 public:
  std::string value;
  Node* left;
  Node* right;
};
```

*sample solution: 15 line(s) of code*

## 5 Spelling Bee Re-Sting [ 33 pts ]




Now that we have learned more sophisticated (and efficient!) data structures, let's revisit the statistics we computed in Homework 2, for the *Spelling Bee* game. Remember that the goal of the puzzle is to make English words using only the 7 letters for the day's puzzle. And furthermore, the words must use the center letter, in this example, the letter 'g'.

While the letter 'g' is required to be in every word, it is not necessarily the most frequently appearing letter across all 23 valid words in the solution. In fact, in this example, the letter 'a' appears 33 times, and the letter 'g' only appears 26 times. Our goal in this problem is to print the following statistics for a specific Spelling Bee puzzle:

```
The letter(s): 'f' & 't' appear 8 times.
The letter(s): 'l' & 'n' appear 11 times.
The letter(s): 'r' appear 12 times.
The letter(s): 'g' appear 26 times.
The letter(s): 'a' appear 33 times.

The letter(s): 'f' appear in 7 words.
The letter(s): 'l' & 't' appear in 8 words.
The letter(s): 'n' & 'r' appear in 11 words.
The letter(s): 'a' & 'g' appear in 23 words.
```

```
rang
ragtag
raga
grant
gran
gall
gaga
gala
gnat
frag
gaff
tang
fragrant
flag
fang
alga
graft
gnarl
algal
flagrant
gang
gallant
agar
```

To get started, carefully study the code below that reads the solution words from an input file stream named `istr`. It prepares two structures named `frequency_of_total_usage` and `frequency_of_use_in_words` that can be simply printed to the screen as shown above. The details of the code to print these variables is is omitted from this problem.

```
total_uses_type total_uses;
used_in_words_type used_in_words;

// read in each solution word from the input file
std::string s;
while (istr >> s) {
  used_letters_type letters_in_this_word = collect_letters(s);
  increment_total_uses(total_uses, s);
  increment_used_in_words(used_in_words, letters_in_this_word);
}

// prepare the frequency statistics for printing
frequency_of_total_usage_type frequency_of_total_usage;
frequency_of_use_in_words_type frequency_of_use_in_words;
for (total_uses_type::iterator tu = total_uses.begin(); tu != total_uses.end(); tu++) {
  frequency_of_total_usage[tu->second].insert(tu->first);
}
for (used_in_words_type::iterator uw = used_in_words.begin(); uw != used_in_words.end(); uw++) {
  frequency_of_use_in_words[uw->second].insert(uw->first);
}
```

You will fill in some of the missing pieces of this implementation on the next few pages.

### 5.1 Complete these Bee Sting typedefs [ 5 pts ]

IMPORTANT: In this problem, you are not allowed to use C-style arrays, STL `vector`, STL `list`, or any classes that create "homemade" versions of these containers.

```
typedef [                                        ] used_letters_type;

typedef [                                        ] total_uses_type;

typedef  total_uses_type  used_in_words_type;

typedef [                                        ] frequency_of_total_usage_type;

typedef  frequency_of_total_usage_type  frequency_of_use_in_words_type;
```

### 5.2 Can you Picture This? [ 5 pts ]

Using conventions from lecture, draw a diagram of the variable `letters_in_this_word` immediately after we process the 2nd word in this example solution, `ragtag`. Also draw the data stored in the `total_uses` and `used_in_words` variables after processing the first two words from the solution file, `rang` and `ragtag`.

| `letters_in_this_word` | `total_uses` | `used_in_words` |
| --- | --- | --- |
| | | |

### 5.3 Implement `collect_letters` [ 7 pts ]

*sample solution: 7 line(s) of code*

### 5.4 Implement increment_total_uses [ 6 pts ]

*sample solution: 5 line(s) of code*

### 5.5 Implement increment_used_in_words [ 7 pts ]

*sample solution: 7 line(s) of code*

### 5.6 And Now Paint the Complete Picture [ 3 pts ]

Again using conventions from lecture, draw diagrams for the data stored in the `frequency_of_total_usage` and `frequency_of_use_in_words` structures after the entire input file is processed and these structures are prepared for printing.

`frequency_of_total_usage` `frequency_of_use_in_words`

# 6 Lightning Doesn't Strike Out Twice [ 20 pts ]

In this problem we declare the following three container objects, and fill the containers with a very large number ($n$) of English words, represented as STL `string`s. The code for filling the structures is omitted.

```
std::vector<std::string> my_vec;
std::list<std::string> my_list;
std::set<std::string> my_set;
```

Below we call the `do_it` function with each version of the data. Assume that $k$ is a positive integer and $k << n$. For each numbered line of the function, for each of these container objects, either:

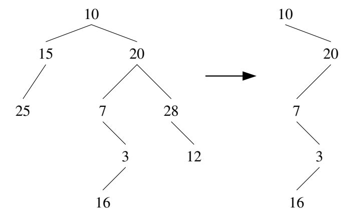
- Indicate that the syntax is incorrect for the container and will not compile by drawing an 'X' in the box. You should then assume this line is commented out for that container. *OR*
- Specify the Big O Notation for that line in terms of $k$ and/or $n$.

```
template <class T> void do_it(T &d) {

01   d.sort();

02   std::sort(d.begin(),d.end());

03   d.push_front("FIRST_THING");

04   d.push_back("LAST_THING");

     T::iterator itr;

05   itr = d.end();

06   for (int i = 0; i < k; i++) { itr--; }

07   itr -= k;

08   itr = d.insert(itr,"A_MIDDLE_THING");

09   itr = std::find(d.begin(),d.end(),"the");

10   itr = d.find("the");

     assert (itr != d.end());

11   itr = d.erase(itr);

12   std::cout << d.size() << std::endl;

}
```

| Line | do_it(my_vec); | do_it(my_list); | do_it(my_set); |
|---|---|---|---|
| 01 | | | |
| 02 | | | |
| 03 | | | |
| 04 | | | |
| 05 | | | |
| 06 | | | |
| 07 | | | |
| 08 | | | |
| 09 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

# 7 Unbalanced Tree Pruning [ 20 pts ]

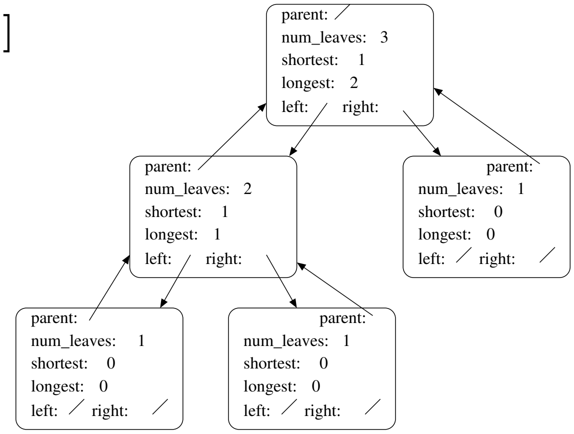
Write a function named `keep_longest` that takes as input a pointer to the root of a binary tree, and removes all of the nodes in the tree except the nodes that form the longest path from root to leaf node. You are encouraged to write helper function(s) as needed.




```
class Node {
public:
  int value;
  Node *left;
  Node *right;
};
```

*sample solution: 24 line(s) of code*

# 8 Leaf Counting Construction [ 15 pts ]




Write a function named `create_tree` that takes one argument, a positive integer named `num_leaves` and returns a pointer to the root `Node` of a binary tree that has the specified number of leaves and is at least approximately balanced. In addition to the number of leaves beneath it, each `Node` should also store the length of the longest and shortest paths from that node to a leaf node.
You may write the constructor for the `Node` class.

```
class Node {
public:
```

```
  int num_leaves;
  int shortest;
  int longest;
  Node *parent;
  Node *left;
  Node *right;
};
```

*sample solution: 16 line(s) of code*

# 9 Tree Traversal Bingo [ 9 pts ]

For each of the diagrams below, write a letter corresponding to one of the following statements that accurately describes the diagram. Each letter should be used exactly once.

(A) has post-order traversal: 1 2 3 4 5 6 7

(B) is not a tree

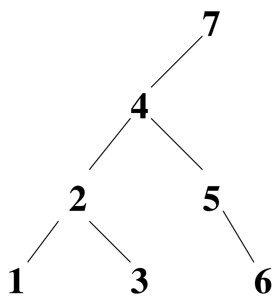
(C) has in-order traversal: 5 4 7 1 6 2 3

(D) is a binary search tree

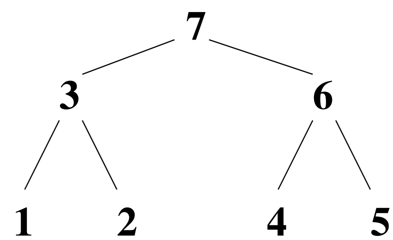
(E) has breadth-first traversal: 7 6 5 4 3 2 1

(F) cannot be colored as a red-black tree

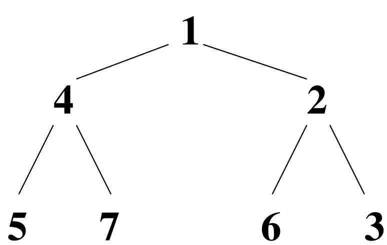
**9.1**



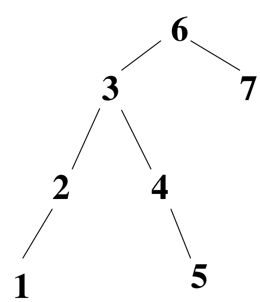

**9.2**



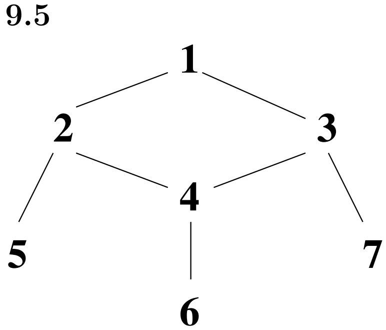

**9.3**



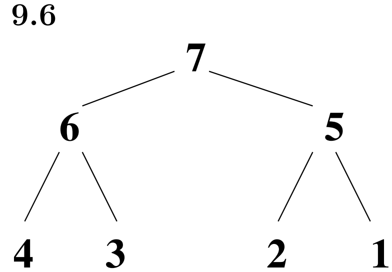

**9.4**




**9.5**




**9.6**



# 10 Order Up! [ / 48 ]

Alyssa P. Hacker is consulting for a restaurant to build a customer and order tracking system. Below is a small sample of the data structure she's designed. She explains that it uses 5 *different* STL data structures, but it does *NOT* use STL `vector`, or the C-style array, or any custom classes.

| Meal | | | Orders | | |
|---|---|---|---|---|---|
| apple_pie | | | Victoria,1 | Ben,2 | Herta,1 |
| happy_meal | | | Ben,1 | | |
| fanta | nuggets | | Jacob,1 | Ben,3 | |
| salad | tea | | Victoria,1 | Herta,2 | Jacob,1 |
| burger | coke | fries | Jacob,1 | Victoria,1 | |
| burger | fries | sprite | Jacob,1 | | |
| coke | fries | salad | Herta,1 | Victoria,1 | |

The design allows us to study the frequency of meals (combinations of items on the menu) ordered by customers, and who has ordered the meal most recently. For example, 3 different customers have ordered the meal "salad & tea". Herta has ordered that meal twice and Jacob has ordered it most recently.

To record a customer order in the structure above, we typically use the following 3 argument function:

`placeOrder( data , "Jacob" ,` *meal with burger, fries, and coke* `);`

Where `data` is the complete structure diagrammed above. This call will update the fifth row of the structure to record that Jacob has now ordered that meal twice, and furthermore, the sequence will be changed so that Jacob is last (swapping places with Victoria) because he has now ordered that meal most recently.

## 10.1 Fast Food `typedefs` [ / 6 ]

After looking ahead through the rest of this problem, let's define a few helpful `typedefs`. You will be graded on the convenience and efficiency of the overall structure. *Also remember that Alyssa said she used 5 different STL types, but NOT* `vector`*.*

`typedef` [ ] `Meal;`

`typedef` [ ] `Orders;`

`typedef` [ ] `Data;`

### 10.2 Organizing the Meals [ / 6 ]

On the previous page we can see the restaurant's desired organization for the meals. We should start with the simple (1 item) meals and progress to more complicated meals. Meals of the same complexity are organized alphabetically. Write the necessary helper function so this happens automatically.

*sample solution: 13 line(s) of code*

### 10.3 Complexity Analysis [ / 6 ]

Let's assume that the restaurant has $i$ items on the menu and $c$ unique customers. Customers have created $m$ different meals (each with at most $k$ menu items). Any one meal has been ordered by at most $d$ different customers, and at most $t$ times by one customer. What is the Big 'O' Notation for the following functions?

`placeOrder` *Note: We don't ask you to implement this.*

`namelessOrder` *You will implement this on the next page (page 4).*

`orderMyFavorite` *You will implement this in a couple of pages (page 5).*

### 10.4 Convenience for Our Most Loyal Customers: `namelessOrder` [ / 16 ]

This structure allows us to streamline ordering for the most frequent customers. They don't even need to leave their name when they place an order. For example:

`namelessOrder( data ,` *meal with nuggets and fanta* `);`

prints "`Welcome back, Ben!`" and places his order (because Ben has ordered that meal more often than anyone else!) However, attempts to use this function can fail and result in error messages:

`namelessOrder( data ,` *meal with salad, fries, and coke* `);`

will print to STDERR: "`ERROR: What name should we put on this order?`" (because two different people, Herta and Victoria, are tied for most times ordering that meal). And

`namelessOrder( data ,` *meal with happy_meal and coke* `);`

will print to STDERR: "`ERROR: No one's ordered this meal before!`"

*sample solution: ∼ 30 line(s) of code*

### 10.5 We Know What You Want! `orderMyFavorite` [ / 14 ]

Alternately, customers may re-order their personal favorite (most frequently ordered) meal:

```
orderMyFavorite( data , "Herta" );
```

which will respond with the success message: "`Placing your order for salad & tea.`" However,

```
orderMyFavorite( data , "Victoria" );
```

prints the error message "`ERROR: You have multiple, equally-favorite meals.`" (because Victoria has ordered four different meals, one time each). And

```
orderMyFavorite( data , "Louis");
```

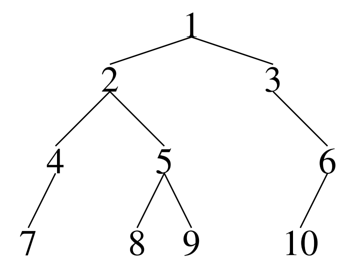
prints "`ERROR: We don't have any prior orders for you.`"

*sample solution: ∼ 30 line(s) of code*

## 11 Each Level, All Pairs [ / 16 ]

```
template <class T>
class Node {
public:
  T value;
  Node* left;
  Node* right;
};
```

Write a function named `each_level_all_pairs` that takes in a pointer to the root `Node` of a binary tree and prints all pairs of values from each "level" of the tree. For example, the function will print the following 10 pairs for the tree on the right, which has 4 levels:

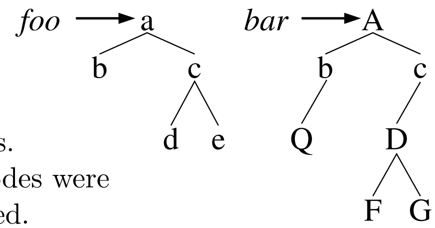


```
(2,3)  (4,5)(4,6)(5,6)   (7,8)(7,9)(7,10)(8,9)(8,10)(9,10)
```

There are no pairs from the top level (because there's only 1 value). There's 1 pair from the second level. We have 3 values and 3 pairs from the third level, and 4 values and 6 pairs from the lowest level. *Note: The extra space between pairs of different levels is optional.*

*sample solution: 18 line(s) of code*

## 12 tREeVISION (Tree Revision) [ / 20 ]




Write a function named `treevision` that takes in two `Node` pointers and modifies the first tree to match the second tree in shape and values. The function returns a `Trio` of three numbers, indicating how many nodes were edited, how many nodes were added, and how many nodes were removed.
For example `treevision(foo,bar)` will return the values `2 3 1` because 2 nodes were edited ('`a`'→'`A`' and '`d`'→'`D`'), 3 nodes were added ('`Q`', '`F`', and '`G`') and 1 node was removed ('`e`').

```
class Node {
public:
  Node(char v):value(v){left=right=NULL;}
  char value;
  Node* left;
  Node* right;
};
```

```
class Trio {
public:
  Trio(int e,int a,int r):edit(e),add(a),remove(r){}
  int edit;
  int add;
  int remove;
};
```

*sample solution: 25 line(s) of code*

## 13 Treeslinger Quick Draw! [ / 13 ]

Draw a balanced binary tree with in-order traversal: `the fox jumped over the lazy dogs`

Draw a binary search tree with leaves: $2^{10}$, 10, $10^2$, and $\sqrt{10}$.

Draw a balanced ternary tree with pre-order traversal: `13 12 11 10 9 8 7 6 5 4 3 2 1`

Draw a red-black tree with values 'a'-'g', with root 'e' and 3 red nodes: 'b', 'd', and 'g'.