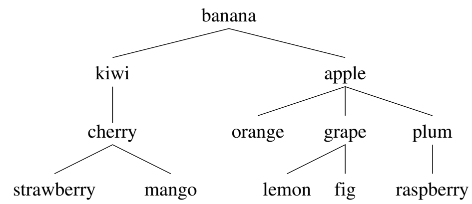
CSCI-1200 Data Structures
Test 3 — Practice Problems

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 Recursing Breadth-First Fruit Tree Traversal [38 pts]

Louis B. Reasoner is verifying the output of his company's breadth-first traversal code for fruit trees. Each node in the tree can have zero or more children, and the tree will never have duplicate items. Remember that in a breadth-first traversal the children at each level can appear in any order. So both of the following orderings are valid breadth-first traversals of this sample tree:




```
banana kiwi apple cherry orange grape plum strawberry mango lemon fig raspberry
banana kiwi apple plum grape cherry orange mango lemon strawberry raspberry fig
```

The `Fruit` node class is on the right. Louis proposes we write three helper functions plus the overall `is_breadth_first` function. Your task is to implement these functions and analyze the Big O Notation for memory & running time.

```
class Fruit {
public:
  Fruit(const std::string &v) { value = v; }
  std::string value;
  std::vector<Fruit*> children;
};
```

1.1 Counting [3 pts]

Before we jump into the code... How many different valid breadth-first traversal orderings are there for the sample tree shown above? Write 1-2 sentences justifying your answer.

a) `2` b) `12` c) `24` d) `60` e) `74` f) `144` g) `1440` h) `6024` i) `12!` j) *other*

1.2 Helper Function `num_appearances` [7 pts]

Implement a *recursive* function `num_appearances` with two required arguments: an STL `string` and an STL `vector` of STL `strings`. A third *optional* argument with a default value facilitates the recursive implementation. The function returns the number of times the string argument appears within the vector.

sample solution: 6 lines of code

For a vector with v strings: Big O Notation for memory: running time:

1.3 Helper Function `everything_appears_once` [7 pts]

Next, implement a *recursive* helper function `everything_appears_once` that takes two arguments: a pointer to the root `Fruit` node and an STL `vector` of STL `string`s. The function should return true if every element in the tree appears exactly once in the vector and false otherwise.

sample solution: 8 lines of code

For a tree with n nodes and height h, and a vector with v strings: Big O Notation for memory: running time:

1.4 Helper Function `which_level` [7 pts]

Write `recursive` function `which_level` that takes two arguments: a pointer to a `Fruit` node and an STL `string`. The function returns an integer indicating on which level of the tree the string is located: where 0 is the root, 1 is a direct child of the root, etc. The function returns -1 if the string is not in the tree.

sample solution: 9 lines of code

For a tree with n nodes and height h: Big O Notation for memory: running time:

1.5 Implement is_breadth_first [10 pts]

Finally, implement the `is_breadth_first` function that takes two arguments: a pointer to the root `Fruit` tree node and a candidate breadth-first ordering stored as an STL `vector` of STL `strings`. The function returns true if the ordering it is a valid breadth-first traversal of the tree and false otherwise. It should use (directly or indirectly) all of the helper functions you defined above.

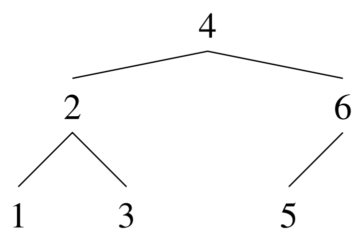
sample solution: 11 lines of code

1.6 Analyze is_breadth_first Memory and Running Time [4 pts]

What is the overall Big O Notation for the memory usage and running time of your `is_breadth_first` function? Assume the `Fruit` tree has n nodes and height h, and the proposed traversal order has $v == n$ elements. Write 2-3 concise and well-written sentences justifying your answer.

2 Enhanced Tree Copy [13 pts]

Write a function named `EnhancedCopy` that takes in a pointer to the root of a tree built with `Basic` nodes, and returns a pointer to the root of a copy of that tree made with `Enhanced` nodes. `Enhanced` nodes have pointers to their parents and also to their sibling nodes. In the example tree below: nodes 2 & 6 are siblings; nodes 1& 3 are siblings; and nodes 4 and 5 do not have siblings.




```
template <class T> class Basic {
public:
  Basic(const T& v): value(v),
    left(NULL), right(NULL) {}
  T value;
  Basic* left;
  Basic* right;
};

template <class T> class Enhanced {
public:
  Enhanced(const T& v): value(v),parent(NULL),
    left(NULL),right(NULL),sibling(NULL) {}
  T value;
  Enhanced* parent;
  Enhanced* left;
  Enhanced* right;
  Enhanced* sibling;
};
```

sample solution: 14 lines of code

3 Spicy Chronological Sets using Maps [/ 33]

Ben Bitdiddle is organizing his spice collection using an STL `set` but runs into a problem. He needs the fast `find`, `insert`, and `erase` of an STL set, but in addition to organizing his spices alphabetically, he also needs to print them out in chronological order (so he can replace the oldest spices).

Ben is sure he'll have to make a complicated custom data structure, until Alyssa P. Hacker shows up and says it can be done using an STL `map`. She quickly sketches the diagram below for Ben, but then has to dash off to an interview for a Google summer internship.

Alyssa's diagram consists of 3 variables. The first variable, containing most of the data, is defined by a `typedef`. Even though he's somewhat confused by Alyssa's diagram, Ben has pushed ahead and decided on the following interface for building his spice collection:

```
chrono_set cs;
std::string oldest = "";
std::string newest = "";
insert(cs,oldest,newest,"garlic");
insert(cs,oldest,newest,"oregano");
insert(cs,oldest,newest,"nutmeg");
insert(cs,oldest,newest,"cinnamon");
insert(cs,oldest,newest,"basil");
insert(cs,oldest,newest,"sage");
insert(cs,oldest,newest,"dill");
```

chrono_set cs:

"basil"	<"cinnamon", "sage">
"cinnamon"	<"nutmeg", "basil">
"dill"	<"sage", "">
"garlic"	<"","oregano">
"nutmeg"	<"oregano","cinnamon">
"oregano"	<"garlic", "nutmeg">
"sage"	<"basil", "dill">

std::string oldest: "garlic"

std::string newest: "dill"

Ben would like to output the spices in 3 ways:

```
ALPHA ORDER:        basil   cinnamon      dill    garlic     nutmeg    oregano       sage
OLDEST FIRST:      garlic    oregano    nutmeg  cinnamon      basil       sage       dill
NEWEST FIRST:        dill       sage     basil  cinnamon     nutmeg    oregano     garlic
```

If he buys more of a spice already in the collection, the old spice jar should be discarded and replaced. For example, continuing the example above, after calling:

```
insert(cs,oldest,newest,"cinnamon");
```

The spice collection output should now be:

```
ALPHA ORDER:        basil   cinnamon      dill    garlic     nutmeg    oregano       sage
OLDEST FIRST:      garlic    oregano    nutmeg     basil       sage       dill   cinnamon
NEWEST FIRST:    cinnamon       dill      sage     basil     nutmeg    oregano     garlic
```

3.1 The `typedef` [/ 3]

First, help Ben by completing the definition of the typedef below:

```
typedef                                                    chrono_set;
```

3.2 Printing out the spice collection [/ 8]

Next, write the code to output (to `std::cout`) Ben's spices in alphabetical and chronological order:

```
std::cout << "ALPHA ORDER:  ";
```

sample solution: 4 lines of code

```
std::cout << std::endl;
std::cout << "OLDEST FIRST: ";
```

sample solution: 5 lines of code

```
std::cout << std::endl;
```

3.3 Performance Analysis [/ 5]

Assuming Ben has n spices in his collection, what is the order notation for each operation? *Note: You may want to first complete the implementation of the* `insert` *operation on the next page.*

printing in alphabetical order:

printing in chronological order:

`insert`-ing a spice to the collection:

3.4 Implementing `insert` for the `chrono_set` [/ 17]

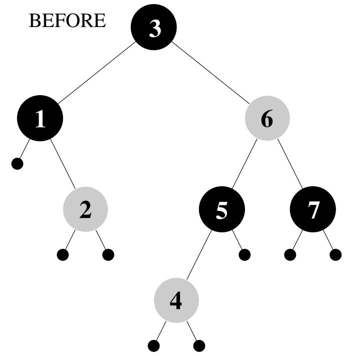
Finally, implement the `insert` function for Ben's spice collection. Make sure to handle all corner cases.

sample solution: 26 lines of code

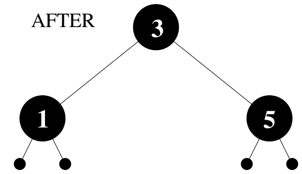
4 Erase Red [13 pts]

Write a function named `erase_red` that takes one argument, a pointer to the root of a red-black binary search tree. The function should erase all of the red nodes in the tree, and everything in the right subtree of those red nodes. An example is shown on the right. *NOTE: The red nodes are shown in grey.* You may write a helper function.

BEFORE



AFTER




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

sample solution: 18 lines of code

What will be true about the number of nodes in the tree after calling `erase_red`? Why? Write 1-2 sentences justifying your answer.

5 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);
```

5.1 The `map1` Data Type [3 pts]

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

5.2 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 ***
```

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

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

5.3 The `map2` Data Type [4 pts]

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

5.4 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 lines 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.

5.5 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 lines 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.

5.6 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 lines of code

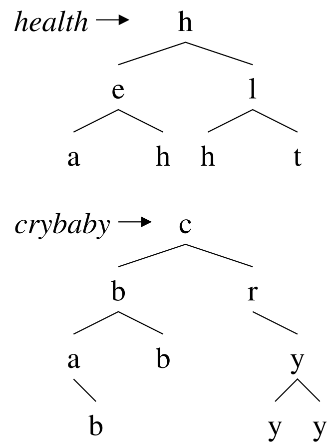
What is the Big O Notation for the `count_reduce_to_map2`? Justify your answer.

6 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 lines of code

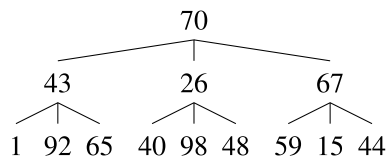
7 Constant Memory Breadth-First Traversal [31 pts]

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
```

7.1 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 lines of code

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

7.2 Implement and Analyze `NumNodesOnLevel` [6 pts]

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

sample solution: 5 lines of code

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

7.3 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 lines of code

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

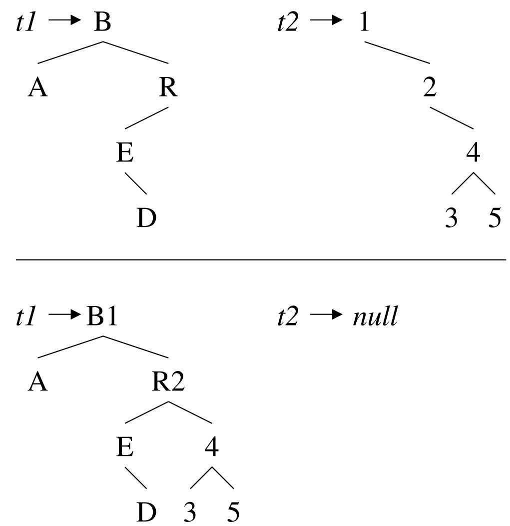
For running time, Best Case: For running time, Worst Case:

7.4 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.

8 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`.

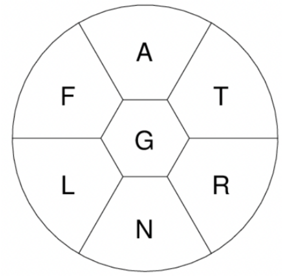


As shown in the example to the right, after the `overlay` function call, the combined tree is stored in the `t1` variable and the `t2` variable is an empty tree.

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

sample solution: 15 lines of code

9 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.

9.1 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;
```

9.2 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

9.3 Implement `collect_letters` [7 pts]

sample solution: 7 lines of code

9.4 Implement `increment_total_uses` [6 pts]

sample solution: 5 lines of code

9.5 Implement `increment_used_in_words` [7 pts]

sample solution: 7 lines of code

9.6 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`

10 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 `strings`. 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:

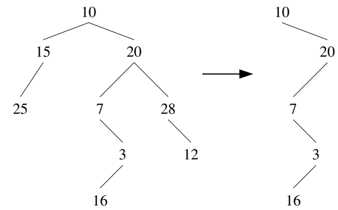
- Indicate that the syntax is incorrect for the container and will not compile by drawing an 'X' in the box. You should then assume this line is commented out for that container. *OR*
- Specify the Big O Notation for that line in terms of k and/or n.

```
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;

}
```

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

11 Unbalanced Tree Pruning [20 pts]

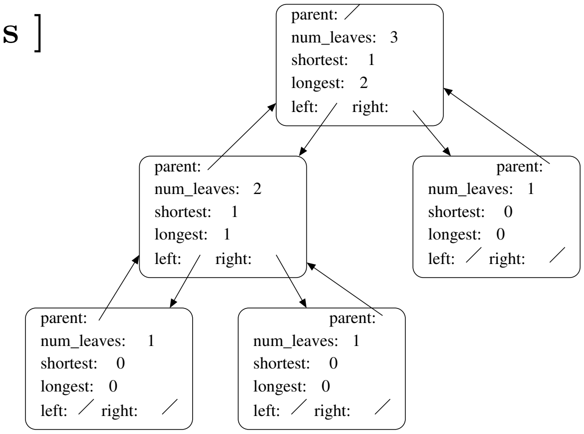
Write a function named `keep_longest` that takes as input a pointer to the root of a binary tree, and removes all of the nodes in the tree except the nodes that form the longest path from root to leaf node. You are encouraged to write helper function(s) as needed.




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

sample solution: 24 lines of code

12 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 lines of code

13 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

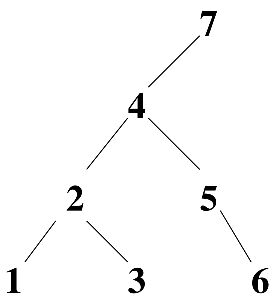
(C) has in-order traversal: 5 4 7 1 6 2 3

(D) is a binary search tree

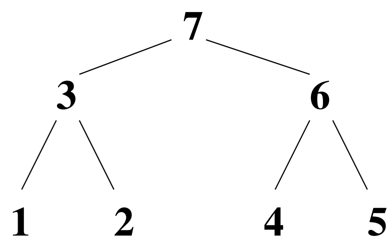
(E) has breadth-first traversal: 7 6 5 4 3 2 1

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

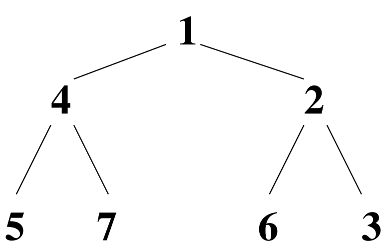
13.1



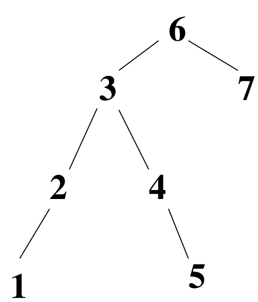
13.2



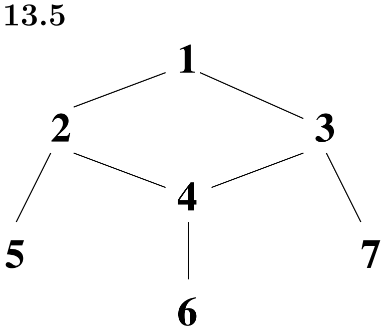
13.3



13.4



13.5



13.6