CS 106X, Autumn 2018
Practice Final Exam (Based on Autumn 2017 CS 106X Final Exam)

Your Name: _____

Section Leader: _____

Honor Code: *I hereby agree to follow both the letter and the spirit of the Stanford Honor Code. I have not received any assistance on this exam, nor will I give any. The answers I submit are my own work.*

Signature: _____ ← **YOU MUST SIGN HERE!**

Rules: *(same as posted previously to class web site)*

- This is an **individual exam**; you are to complete it yourself without assistance from others.
- You have 3 hours (180 minutes) to complete this exam.
- This test is **open-book**, but **closed notes**. You may not use any printed paper resources.
- You may *not* use any computing devices, including calculators, cell phones, iPads, or music players.
- Unless otherwise indicated, your code will be graded on proper behavior/output, not on style. Some particular problems have style constraints or other code constraints, so read each problem carefully. We reserve the right to deduct points for solutions that are grossly inefficient or wasteful of resources.
- On code-writing problems, you do not need to write a complete program, nor `#include` statements. Write only the code (function, etc.) specified in the problem statement.
- Please do not **abbreviate** code, such as writing ditto marks ("") or dot-dot-dot marks (...).
- Unless otherwise specified, you may define **helper functions** but you may not declare **global variables**.
- If you wrote your answer on a back page or attached paper, please **label this** clearly to the grader.
- Follow the Stanford **Honor Code** on this exam and correct/report anyone who does not do so.

Good luck! You can do it!

Problem	Description	Earned	Possible
1	Inheritance (read)		10
2	Linked Lists (read)		6
3	Inheritance (write)		8
4	Linked Lists (write)		10
5	Graphs (read)		6
6	Graphs (write)		10
7	Binary Trees (write)		10
TOTAL	**Total Points**		**60**

1. Inheritance and Polymorphism (read)

Consider the following classes; assume that each is defined in its own file.

```
class Jesse : public Tulip {
public:
    virtual void m1() {
        cout << "Js1  ";
        Cassidy::m1();
    }

    virtual void m4() {
        cout << "Js4  ";
        m2();
    }
};

class Cassidy {
public:
    virtual void m1() {
        cout << "Ca1  ";
        m2();
    }

    virtual void m2() {
        cout << "Ca2  ";
    }
};

class Tulip : public Cassidy {
public:
    virtual void m2() {
        cout << "Tu2  ";
        Cassidy::m2();
    }

    virtual void m3() {
        cout << "Tu3  ";
        m1();
    }
};
```

Now assume that the following variables are defined:

```
Cassidy*  var1 = new Tulip();
Tulip*    var2 = new Jesse();
```

In the table below, indicate in the right column the output produced by the statement in the left column.

If the statement does not compile, write "**compiler error**".
If a statement would crash at runtime or cause other unpredictable behavior, write "**crash**".

Statement	**Output**
`var1->m1();`	_____
`var1->m3();`	_____
`var2->m2();`	_____
`var2->m3();`	_____
`var2->m4();`	_____
`((Tulip*) var1)->m3();`	_____
`((Tulip*) var1)->m4();`	_____
`((Jesse*) var1)->m4();`	_____
`((Cassidy*) var2)->m3();`	_____
`((Jesse*) var2)->m4();`	_____

2. Linked Lists (read)

Recall the `ListNode` structure seen in class:

```
struct ListNode {
    int data;
    ListNode* next;
};
```

Consider the following linked list of `ListNode` objects, with a pointer named `front` that points to the first node:

front -> 25 -> 40 -> 50 -> 20 -> 50 -> 10 -> 8 -> 60 -> 60 -> 37 /

Draw the final state of the linked list after the following code runs on it. If a given node is removed from the list, you don't need to draw that node, only the ones that remain reachable in the original list.

```
void linkedListMystery(ListNode*& front) {
    ListNode* curr = front;
    while (curr->next != nullptr) {
        ListNode* temp = curr->next;
        if (curr->data >= curr->next->data) {
            curr->next = temp->next;
            if (curr->data == temp->data) {
                curr->next = temp->next;
                delete temp;
            } else {
                temp->next = front;
                front = temp;
            }
        } else {
            curr = curr->next;
        }
    }
}
```

3. Inheritance / Collection Implementation (write)

Suppose that an existing class named **ArrayStack** has already been written. An ArrayStack is an implementation of a stack of integers using an array as its internal representation. It has the following implementation:

```
class ArrayStack {
public:
    ArrayStack();                   // construct empty stack
    ~ArrayStack();                  // free memory
    virtual bool isEmpty() const;   // true if stack has no elements
    virtual int peek() const;       // return top element (error if empty)
    virtual int pop();              // remove/return top element (error if empty)
    virtual void push(int n);       // add to top of stack, resizing if needed
private:
    int* elements;                  // array of stack data (index 0 = bottom)
    int size;                       // number of elements in stack
    int capacity;                   // length of array
};
```

Define a new class called SortedStack that extends ArrayStack through inheritance. Your class represents a stack of integers that is always stored in sorted non-decreasing order, regardless of the order in which items are pushed onto the stack. Your class should provide the same member functions as the superclass. Your code must work with the existing ArrayStack as shown, unmodified.

For example, if the following elements are added to an empty SortedStack:

```
42, 27, 39, 3, 55, 81, 11, 9, 0, 72
```

The following lines show the stack's state before and after adding each element, in bottom-to-top (left-to-right) order:

```
{}
{42}
{27, 42}
{27, 39, 42}
{3, 27, 39, 42}
{3, 27, 39, 42, 55}
{3, 27, 39, 42, 55, 81}
{3, 11, 27, 39, 42, 55, 81}
{3, 9, 11, 27, 39, 42, 55, 81}
{0, 3, 9, 11, 27, 39, 42, 55, 81}
{0, 3, 9, 11, 27, 39, 42, 55, 72, 81}
```

Write the **.h** and **.cpp** parts of the class separately with a line between to separate them. The majority of your score comes from implementing the correct behavior. You should also appropriately utilize the behavior you have inherited from the superclass and not re-implement behavior that already works properly in the superclass.

Recall that subclasses are **not** able to access private members of the superclass.

You may create ArrayStack objects in your code if it is helpful to do so, but otherwise you should not create any **auxiliary data structures** (arrays, vectors, queues, maps, sets, strings, etc.) in your code.

Write your answer on the next page.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3. Inheritance / Collection Implementation (write)
Writing Space

4. Linked Lists (write)

Write a function **`switchEvens`** that swaps even-valued elements at the same index between two linked lists. Your function is passed two parameters, references to `ListNode` pointers representing the front of each linked list. Recall the `ListNode` structure:

```
struct ListNode {
    int data;
    ListNode* next;
};
```

You should traverse both lists, find every index where **both lists store an even value** (one that is divisible by 2) at that same index, and swap the nodes between the two lists. If one list has more even values than the others, any extra ones are left unmoved. The relative order of the elements must be preserved in both lists.

For example, if `ListNode` pointers named `front1` and `front2` point to the front of lists storing the following values:

```
index      0     1     2     3     4     5     6     7     8     9

front1 ->  2 ->  4 ->  3 ->  7 ->  8 ->  4 ->  6 -> 12 -> 22 -> 10 /
front2 -> 66 -> 55 -> 16 -> 43 -> 22 -> 90 -> 39 -> 44 /
```

After the call of `switchEvens(front1, front2);` , the lists should store the following elements:

```
front1 -> 66 ->  4 ->  3 ->  7 -> 22 -> 90 ->  6 -> 44 -> 22 -> 10 /
front2 ->  2 -> 55 -> 16 -> 43 ->  8 ->  4 -> 39 -> 12 /
```

Notice that at indexes 0, 4, 5, and 7, both lists contained an even value. Such values were swapped between the two lists.

Your function should work properly for a list of any size, including either list being null.

Note: The goal of this problem is to modify the list by modifying **pointers**. It might be easier to solve it in other ways, such as by changing nodes' `data` values or by rebuilding an entirely new list, but such tactics are forbidden.

Constraints: For full credit, obey the following restrictions in your solution. A violating solution can get partial credit.

- **Do not modify the `data` field** of any existing nodes.
- **Do not create any new nodes** by calling `new ListNode(...)`.
  You may create as many `ListNode*` pointers as you like, though.
- Do not use any auxiliary **data structures** such as arrays, vectors, queues, maps, sets, strings, etc.
- Your code must run in no worse than **O(*N*) time**, where *N* is the sum of the lengths of the lists.

Write your answer on the next page.

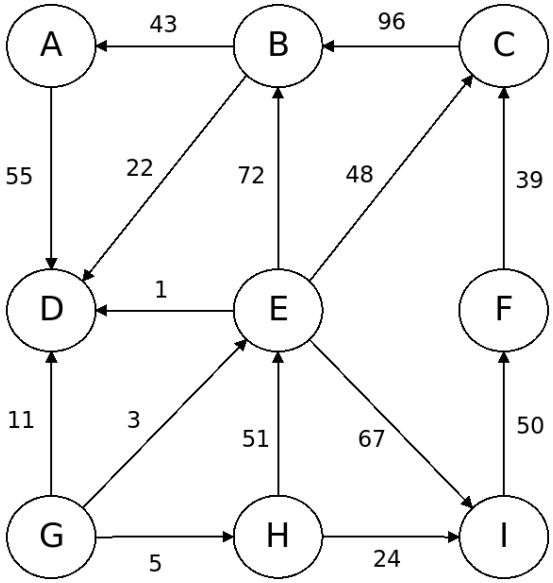
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

---

4. Linked Lists (write)
Writing Space

5. Graphs (read)

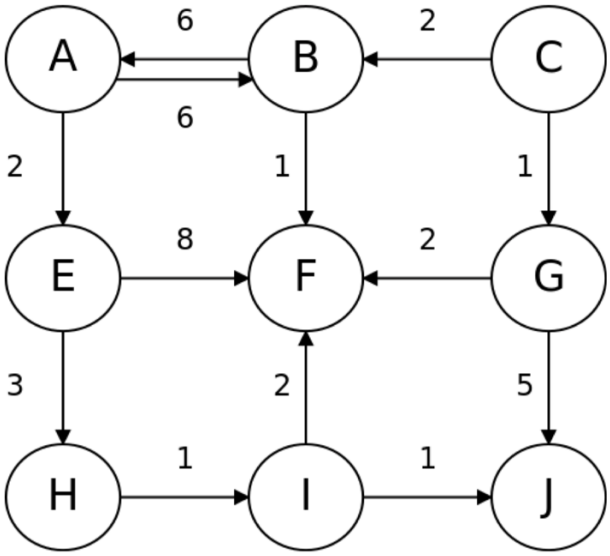
Answer the following three questions below about the included graphs:



a) Write a valid **topological sort** of the vertexes in the graph above. If there are multiple valid sort orders, any will be fine.

- **Sort Order:** _____

For the next two parts, refer to the graph shown below:



b) Write the order that a *depth-first search* (DFS) would visit vertexes if it were looking for a path from vertex A to vertex I. Also write the path it would return. Assume that any "for-each" loop over neighbors returns them in ABC order.

- **Visit Order:** _____
- **Returned Path:** _____

c) Write the order that a *breadth-first search* (BFS) would visit vertexes if it were looking for a path from vertex C to vertex H. Also write the path it would return. Assume that any "for-each" loop over neighbors returns them in ABC order.

- **Visit Order:** _____
- **Returned Path:** _____

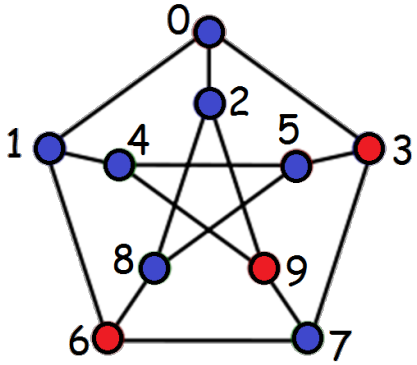
6. Graphs (write)

This problem has **two parts** and asks you to write two functions.

For a given graph G, a *dominating set* is a set of vertexes D where every vertex in G either is an element of D or is adjacent (a neighbor) to an element of D.

Another way of phrasing this is that a given set of vertexes D is a dominating set of G if there does not exist any vertex in G outside of D that cannot reach a vertex in D by traveling across a single edge.

For example, in the graph pictured at right, {V3, V6, V9} is a dominating set, because every vertex in the graph either is one of those vertexes or a neighbor of one of those three vertexes. Other example dominating sets would be {V0, V4, V6} and {V2, V4, V5, V7} and {V2, V4, V5, V8, V9}.



A) Write a function named **`isDominatingSet`** that checks whether a given set of vertexes represents a dominating set for a given graph. Your function accepts two parameters: a reference to a `BasicGraph`, and a reference to a `Set` of strings representing vertex names. You should return `true` if the given set is a dominating set for the given graph, and `false` if not. You may assume that every string in the given set corresponds to the name of a vertex found in the graph. You may also assume that the graph is undirected, in other words, that the edges are bidirectional between neighboring vertexes. If the given graph and set are both empty, you should return `true`.

For example, given the graph above, the call of `isDominatingSet(graph, {"V3", "V6", "V9"})` would return `true`.

B) Write a function **`findDominatingSet`** that looks for a dominating set of a given size in a given graph. Your function accepts three parameters: a reference to a `BasicGraph`, an integer *K*, and a reference to a `Set` of strings for storing your dominating set (an "output parameter"). Your function should check whether the given graph has any dominating sets that contain no more than *K* vertexes. If the graph has any such dominating sets, your function should store one of the dominating sets in your output parameter and should return `true`. If there is no dominating set of the given size, you should return `false`; the contents of the output parameter set do not matter and will be ignored by the caller in such a case. If the graph contains multiple dominating sets of the given size, you may emit any one of them.

For example, given the graph above, the call of `findDominatingSet(graph, 3, set)` would return `true`, and would modify the given set to store a trio of vertexes such as `{"V3", "V6", "V9"}`. For the same graph, the call of `findDominatingSet(graph, 2, set)` would return `false`, because there is no dominating set with ≤ 2 vertexes.

It is fine (and encouraged) for your `findDominatingSet` function to call your `isDominatingSet` function from Part A.

Efficiency: Note that in order to be certain that you have exhaustively found a suitable dominating set (or verified with certainty that there is no such set), you must try every possible subset of its vertexes. This can be a slow process. If your code finds a suitable dominating set, you should stop searching without exploring more possibilities.

Assumptions: You may assume that the graph's state is valid, and that it contains no self-edges (e.g. from *V1* to *V1*). You may assume that there is at most one edge from any vertex *V1* to any other vertex *V2*. You may not assume that the graph is connected, nor make any other assumptions about the number or names of its vertexes or edges.

Constraints: Do not modify the contents of the graph such as by adding or removing vertexes or edges from the graph. You may define **private helper** functions if so desired, and you may construct auxiliary **collections** as needed to solve this problem.

Write your answer on the next page.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

---

6. Graphs (write)
Writing Space

6. Graphs (write)
Writing Space

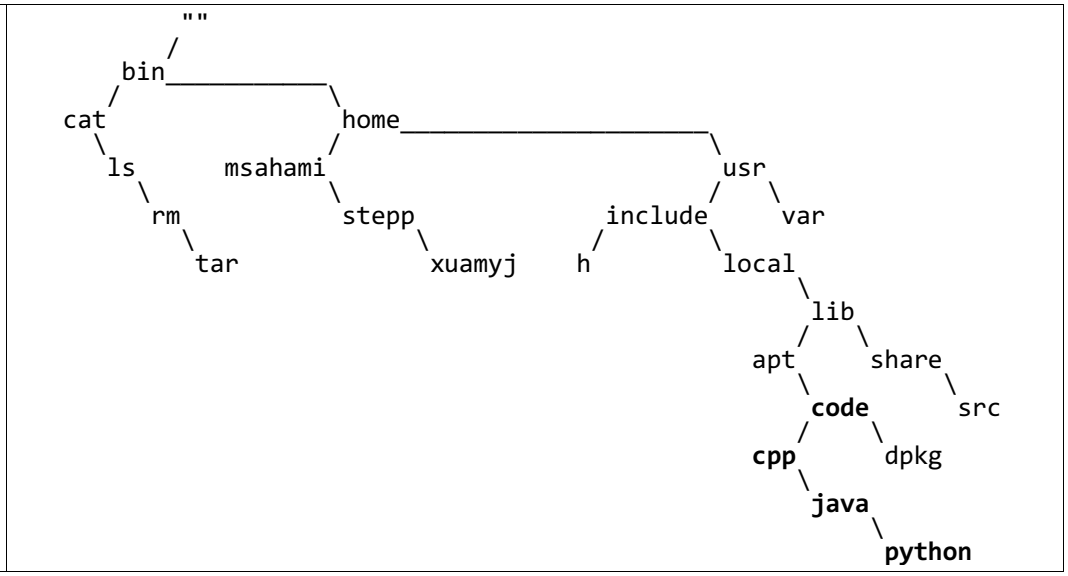
7. Binary Trees (write)

You can use a tree to represent a file system, with each node representing a directory or file. Since a directory can contain more than 2 files, a binary tree may seem like a poor choice. But you can do it using a structure where a node N's "left" child is the first child file inside N, and the "right" child is the next sibling file/dir within the same parent directory of N. (A normal file that isn't a directory will always have a null first child.) The following structure will work:

```
struct FileTreeNode {
    string name;
    FileTreeNode* firstChild;    // "left"
    FileTreeNode* nextSibling;   // "right"
}
```

For example, consider the set of directories and files shown below at left. The file system contains files such as `"/bin/rm"` and `"/usr/lib/code/java"`. You can represent this set of files using the binary tree shown at right. Each "left" line shown below is the `firstChild` of the node above it; each right line shown below is the `nextSibling` of the node before it. The root of the overall file system is a `FileTreeNode` with an empty string as its name, the `bin` node as its `firstChild`, and a null `nextSibling`.

```
bin
    cat
    ls
    rm
    tar
home
    msahami
    stepp
    xuamyj
usr
    include
        h
    local
    lib
        apt
        code
            cpp
            java
            python
        dpkg
    share
    src
var
```



For this problem, write a recursive function named **`removeFile`** that deletes a directory or file from the file system, including all descendent files/dirs inside it. Your function accepts two parameters: a reference to a `FileTreeNode` pointer for the root of the file system binary tree, and a string for the full path of the file or directory to delete, with `/` slashes between directories, such as `"/usr/local/lib"`. Your function should remove the node of the tree that corresponds to that string's file path, along with all of its children, grandchildren, etc. If you remove a node, you should **free its memory** using `delete`. *(You don't need to actually modify the computer's real file system, just the binary tree representing the file system.)* If there is no node in the tree that corresponds to the string path passed in, your function should not modify the tree.

For example, if the file system tree above is represented by a `FileTreeNode` pointer named `fs`, and the call of `removeFile(fs, "/usr/lib/code")` is made, you should modify the tree above to remove the nodes `"code"`, `"cpp"`, `"java"`, and `"python"`, making the next sibling after `"apt"` become `"dpkg"`. If the string to remove were `"/usr/lib"` instead, a total of 7 nodes would be removed: `"lib"`, `"apt"`, `"dpkg"`, plus the four from `"/usr/lib/code"` .

If the string passed is `""` (the empty string), you should delete the entire file system (!).

Constraints: For full credit, obey the following constraints in your solution. A violating solution can get partial credit.

- Your solution must be **recursive** and must not use any **loops**. No exceptions!
- Do not create any **data structures** (arrays, vectors, sets, maps, etc.).
- Do not create any **FileTreeNode objects**. You may create pointers to objects, but not allocate 'new' objects.
- Do not **leak memory**.
- Your solution should be at worst O(*N*) time and must make only a **single pass** over the tree.
- You may define **private helper** functions if you like.

7. Binary Trees (write)
Writing Space

7. Binary Trees (write)
Writing Space