CS106B Final Practice 1

You have 3 hours to complete this exam

This is a closed book, closed computer exam. You are allowed only 1 page (8.5x11, both sides) of notes, and no other aids (and you are not allowed magnifying devices other than regular glasses). You don't need to **`#include`** any libraries, and you needn't use assert to guard against any errors. Understand that the majority of points are awarded for concepts taught in CS106B, and not prior classes. You don't get many points for for-loop syntax, but you certainly get points for proper use of Stanford library containers, efficiency of your code, and solution of the problem.

You will be given a reference sheet. If you need extra space for scratch and/or problem responses, use extra scratch paper. We will not accept any work done on scratch paper.

NAME: ____

SUNet: ____
(e.g., cgregg)

I accept the letter and spirit of the honor code. I will neither give nor receive unauthorized aid on this exam.

signed ____

Problem 1: 15pts

Problem 2: 25pts

Problem 3: 15pts

Problem 4: 15pts

Problem 5: 15pts

Problem 6: 15pts

Total: 100 points

Problem 1: Linked Lists

(15 Points)

Write a function

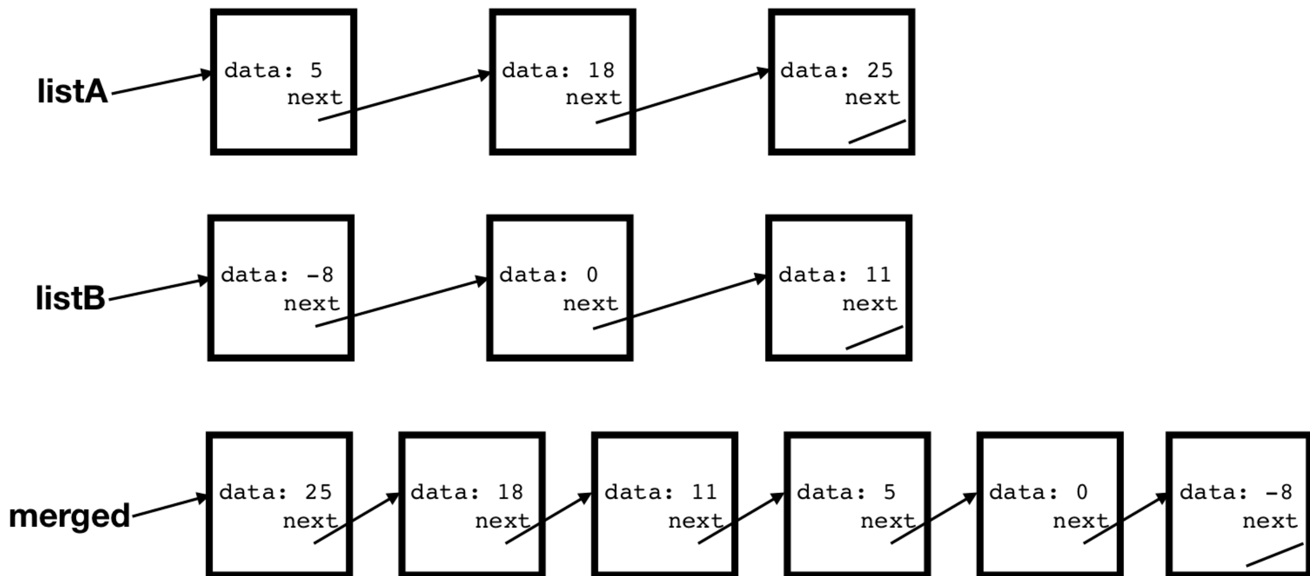
`ListNode *mergeListsAndReverse(ListNode *listA, ListNode *listB);`

that merges two sorted linked lists and also reverses the output, and returns a pointer to the head of the merged and reversed list. The function should not create or delete any **`ListNode`**s. Instead, it should rewire the nodes to form one complete linked list which is sorted in reverse order.

The **`ListNode`** struct is defined as follows:

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

For example, listA and listB below should be merged into a single reversed linked list, and the pointer to the new head should be returned.



Notes:

1. Either or both of the original lists can be empty.
2. Your solution should travese the lists only once. A correct solution that traverses the list more than once will lose 50% of the points for this problem.
3. You are not allowed to create any additional data structures (e.g., **`Vector`**, etc.) for your solution.
4. You should not create any **`ListNode`** objects, although you are allowed to create as many **`ListNode *`** pointers as you need.
5. You are not allowed to change any of the **`data`** fields of the **`ListNode`** objects.

Write your **`mergeListsAndReverse`** function here:

```
ListNode *mergeListsAndReverse(ListNode *listA, ListNode *listB) {
```

Problem 2: Binary Search Trees

(25 Points)

For this problem, you will be writing a simplified Binary Search Tree class, **BST** that has the following header file:

```
// bst.h
#pragma once

struct BST_Node {
    int data;
    BST_Node *left;
    BST_Node *right;
};

class BST {
public:
    BST();  // constructor
    ~BST(); // destructor

    // insert value into BST, ignoring duplicates
    void insert(int value);

    // removes all nodes less than min and greater than max
    void removeOutOfRangeNodes(int min, int max);

    // prints the in-order representation of the tree
    // in the following form:
    // 1,2,3,4
    // with an endl after the list of values.
    void printTreeInOrder();

private:
    BST_Node *root;

    // helper function for insert
    void insert(BST_Node *&n, int value);

    // cleans up the BST
    void deleteTree(BST_Node *n);

    // add more private functions if needed



};
```

Complete the code for each function, and add private helper functions as necessary.

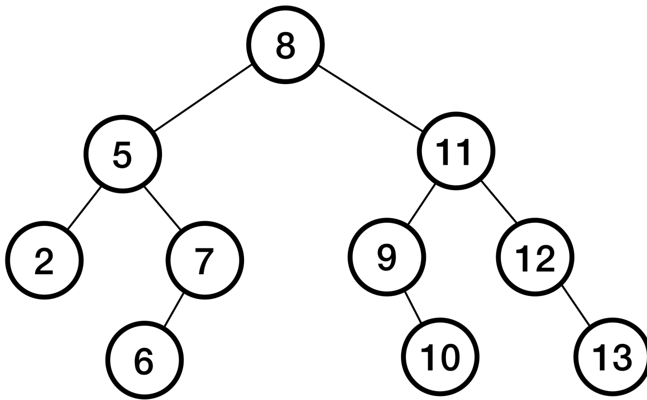
The following program is an example of how the class is used:

```
 1 #include <iostream>
 2 #include "bst.h"
 3 using namespace std;
 4
 5 int main() {
 6     Vector<int> values = {8,5,11,2,7,9,12,6,10,13};
 7     BST tree;
 8
 9     for (int v : values) {
10         tree.insert(v);
11     }
12
13     tree.printTreeInOrder();
14     tree.removeOutOfRangeNodes(3,9);
15     tree.printInOrder();
16     return 0;
17 }
```

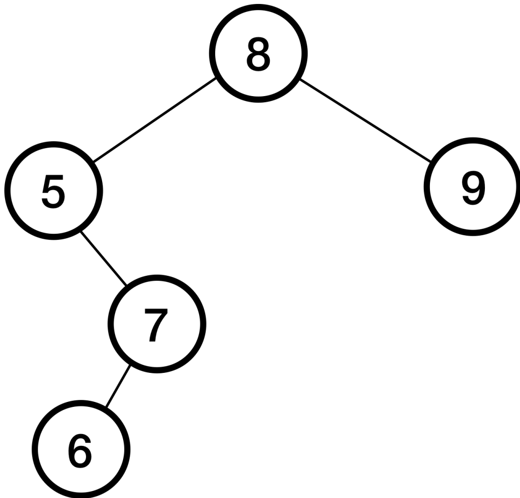
The output from the program is:

```
2,5,6,7,8,9,10,11,12,13
5,6,7,8,9
```

The tree formed by the **`for`** loop should look like this:



The tree after removing the nodes should look like this:



Notes:

1. The **BST** class is not a balanced tree.
2. The destructor should clean up all nodes in the tree.
3. The private **insert** function should create a new **BST_Node** as necessary. If a value is already represented in the tree, **insert** should not duplicate it.
4. The **removeOutOfRangeNodes** function removes all nodes less than **min** and greater than **max**. You should think this function through carefully -- it does not require multiple special-case removal strategies.
5. The **removeOutOfRangeNodes** function should not leak memory. If it removes a node, it should call **delete** on that node.
6. If all nodes are removed from the tree, the **root** of the tree should be **nullptr**.
7. You are not allowed to add new instance variables to the class.

Write your code for the class below

```
BST::BST() {  // constructor

}

BST::~BST() { // destructor

}

void BST::insert(int value) {

}

void BST::removeOutOfRangeNodes(int min, int max) {

}
```

```
void BST::printTreeInOrder() {

}

void BST::insert(BST_Node *&n, int value) {

}

void deleteTree(BST_Node *n) {

}
```

Problem 3: Recursion -- wildcard matches (15 Points)

Most operating systems and many applications that allow users to work with files support wildcard patterns, in which special characters are used to create filename patterns that can match many different files. The most common special characters used in wildcard matching are **`?`**, which matches any single character, and **`*`**, which matches any sequence of characters. Other characters in a filename pattern must match the same character in a filename. For example, the pattern **`*.*`** matches any filename that contains a period, such as **`US.txt`** or **`pathfinder.cpp`**, but would not match filenames that do not contain a period. Similarly, the pattern **`test.?`** matches any filename that consists of the name test, a period, and a single character; thus, **`test.?`** matches **`test.h`** or **`test.c`**, but not **`test.cpp`**. These patterns can be combined in any way you like. For example, the pattern **`??*`** matches any filename containing at least two characters.

Write a function:

```
bool filenameMatches(string filename, string pattern);
```

that takes two strings, representing a filename and a wildcard pattern, and returns **`true`** if that filename matches the pattern. Thus, **`filenameMatches("US.txt", "*.*")`** returns **`true`** **`filenameMatches("test", "*.*")`** returns **`false`** **`filenameMatches("test.h", "test.?")`** returns **`true`** **`filenameMatches("test.cpp", "test.?")`** returns **`false`** **`filenameMatches("x", "??*")`** returns **`false`** **`filenameMatches("yy", "??*")`** returns **`true`** **`filenameMatches("zzz", "??*")`** returns **`true`**

This problem is easier than it might seem at first, if you choose the right recursive decomposition. To help lead you in that direction, start by thinking about the first character in the pattern and distinguish the following cases:

- The pattern string is empty and therefore has no first character.
- The first character in the pattern string is a ?.
- The first character in the pattern string is a *.
- The first character in the pattern string is any other character.

In each of these cases, think about what the filename string would have to look like in order to be a match. As in any recursive formulation, the strategy you choose has to reduce complex problems into simpler instances of the same problem.

In writing your answer to this problem, you should keep the following ideas in mind:

- The period (.) that typically separates parts of a filename is not a special character as far as **`filenameMatches`** is concerned. A period in pattern means that there must be a period in filename, which is exactly what happens with any other character.
- The * character can match any substring in filename, including an empty substring of length 0.
- If you find yourself getting bogged down in lots of detailed code, you are probably veering off onto the wrong track. Go back and look at the hints on recursive decomposition presented earlier in this problem and think again about your strategy.

Write your **`filenameMatches`** function below:

```
bool filenameMatches(string filename, string pattern) {
```

Problem 4: Break Sentence into Dictionary Words

(15 Points)

Given a sentence that has no spaces between the words, and a Lexicon of words, write a function to print out all possible combinations of valid words:

```
void breakSentence(Lexicon &lex, string str);
```

For example, **breakSentence(lex, "subvertexitingot");** would output the following, given a standard English dictionary:

```
sub vertex i tin got
sub vertex it in got
sub vertex it ingot
subvert exit in got
subvert exit ingot
```

Notes:

1. You may define helper functions to solve this problem.
2. For full credit, your solution must be recursive.

Write your **breakSentence** function below:

```
void breakSentence(Lexicon &lex, string str);
```

Problem 5: Hash Map (15 Points)

Simulate the behavior of a hash map of ints (keys) to strings (values) as described and implemented in lecture. Assume the following:

- the hash map array has an initial capacity of 10.
- the hash map uses linked list chaining to resolve collisions, and key/value pairs are inserted at the head of each linked list.
- the hash/compression function is defined as follows:

```
int hash(int key) {
    return key % tableCapacity;
}
```

- rehashing occurs at the end of a **put** where the load factor is >= 1.0 and doubles the capacity of the hash map. Rehashing should be performed as discussed in lecture.

Draw an array diagram to show the final state of the hash map after the following operations are performed. You may leave unused buckets blank, assuming they will contain **nullptr**. All values must be in the proper buckets and in the proper order to receive full credit. Also write the final number of elements, capacity, and load factor of the hash map.

```
map.put(10,"perform");
map.put(20,"your");
map.put(15,"banana");
map.put(53,"get");
map.put(51,"will");
map.put(13,"full");
map.put(106,"106b");
map.put(6,"map");
map.put(87,"skills");
map.remove(10);
map.put(27,"are");
map.put(40,"hash");
map.remove(106);
map.put(28,"and");
map.put(88,"fantastic");
map.put(78,"credit");
map.put(29,"you");
map.remove(15);
```

Your finished table might look something like this:

```
0: 15,banana -> 88,fantastic -> 40,hash
1:
2: 28,and
3: 6,map -> 78,credit
4:
5:
6: 29,you
etc.

Final number of elements: 7
Capacity: 10
Load factor: 0.7
```

Final state of the array:

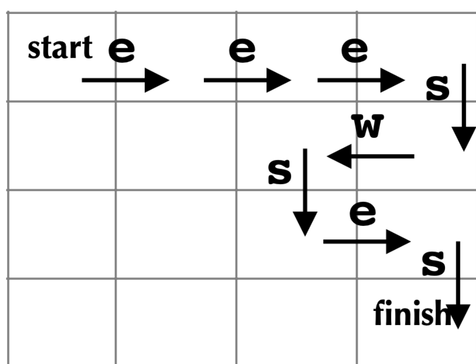
Problem 6: Backtracking: All Paths from Start to Finish (15 points)

Given a Grid<bool> g, write a function **allPaths** that returns a Vector<string> of all possible paths to traverse from the top left corner of the grid (**row=0, col=0**) to the bottom right corner of the grid (**row=g.numRows()-1**, **col=g.numCols()-1**), without re-visiting any grid locations. A path string is defined as a string of compass directions, 'n', 'e', 's', and 'w' corresponding to north, east, south, and west.

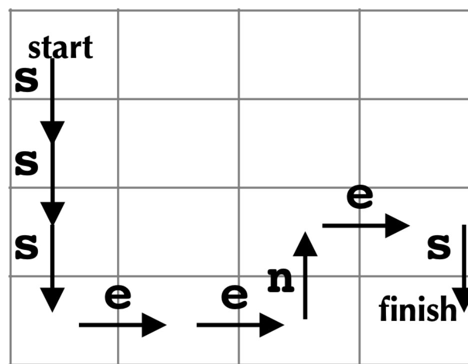
```
Vector<string> allPaths(Grid<bool> &g);
```

You are allowed to create any helper functions you need, and your solution should recursively produce all possible paths as described.

For example, for a 4x4 grid below, the path **"eeeswses"** is drawn on the left grid, and the path **"ssseenes"** is drawn on the right grid.



"eeeswses"



"ssseenes"

Notes:

1. The initial Grid passed into the function will have all of its elements marked as "**false**".
2. The Grid is guaranteed to be at least 1x1 in size (and a 1x1 grid has "" as the path).
3. You should probably use the boolean values in the grid to mark where you have been in the grid as you complete your recursive function. A particular path should not revisit any grid locations.
4. You must use recursion to receive full credit.

Write your **allPaths** function here:

```
Vector<string> allPaths(Grid<bool> &g) {
```