Practice Final 1

CS106B, Winter 2024

Last (**Family**) Name

First (**Given**) Name

Stanford E-mail @stanford.edu

Exam Instructions

There are **6** questions worth a total of **105** points. Write all answers directly on the exam paper in the provided spaces for each question. Do not add or remove pages to this exam, and do not remove the staple. This printed exam is closed-book and closed-device; you may refer only to our provided reference sheet. You are required to write your SUID number in the blank at the top of each odd-numbered page.

Unless otherwise restricted in the instructors for a specific problem, you are free to use any of the CS106B libraries and classes. You don't need *#include* statements in your solutions; just assume the required header files (*vector.h*, *strlib.h*, etc.) are visible. You do not need to declare prototypes. You are free to create helper functions unless the problem states otherwise. Comments are not required, but when your code is incorrect, comments could clarify your intentions and help the graders award partial credit.

The Stanford University Honor Code (2023 Revision)

The Honor Code is an undertaking of the Stanford academic community, individually and collectively. Its purpose is to uphold a culture of academic honesty.

Students will support this culture of academic honesty by neither giving nor accepting unpermitted academic aid in any work that serves as a component of grading or evaluation, including assignments, examinations, and research.

Instructors will support this culture of academic honesty by providing clear guidance, both in their course syllabi and in response to student questions, on what constitutes permitted and unpermitted aid. Instructors will also not take unusual or unreasonable precautions to prevent academic dishonesty.

Students and instructors will also cultivate an environment conducive to academic integrity. While instructors alone set academic requirements, the Honor Code is a community undertaking that requires students and instructors to work together to ensure conditions that support academic integrity.

In signing below, I acknowledge, accept, and agree to abide by both the letter and the spirit of the Stanford Honor Code. I will not receive any unpermitted aid on this test, nor will I give any. I do not have any advance knowledge of what questions will be asked on this exam. My answers are my own work.

(**signature**) (required)

1. Recursive Backtracking (20 pts)

Given two strings, `s1` and `s2`, a "drip word" for those strings is any English word that can be constructed by starting with an empty string and then repeating the following two steps any number of times:

1. Remove the **first** character from *either* `s1` or `s2`.
2. *Either* discard the character removed in Step 1, *or* append it to the **end** of the string we are building.

For example, given the strings `s1 = "cat"` and `s2 = "dogs"`, one drip word we can create is "dot" (take the 'd' from `s2`, take the 'o' from `s2`, discard the 'c' from `s1`, discard the 'a' from `s1`, take the 't' from `s1`, and stop). In contrast, "got" is *not* a drip word for `s1` and `s2`; the only way to start our string with 'g' would be to discard the 'd' and 'o' from `s2`, and once they are discarded, we cannot then get them back and place them in our string after the 'g'.

(These are called "drip words" because we can imagine `s1` and `s2` are made of wax, and we are heating them up from the leftmost character and "dripping" the characters one by one into our new string, or discarding the dripping characters as we go.)

Write a function that takes two strings, `s1` and `s2`, and returns a set of all their drip strings.

- You must write a **recursive helper function**.
- Your recursive helper function must use backtracking techniques to generate its results. Namely, as you generate drip string candidates, you should only continue to explore ones that could potentially lead to valid results. As soon as it becomes clear that a string you have generated cannot lead to any valid results, stop exploring that dead-end path.
- The string you are building can be passed to your helper function by value or by reference – whatever you find easier to work with. `s1` and `s2` must be passed by **reference**. Be sure to consider the implications of passing those parameters by reference as you construct your solution.
- You may assume we have a lexicon with valid English words. See function signature on next page.
- You may assume all alphabetic characters in `s1`, `s2`, and our lexicon input file are lowercase.

Please write your solution on the following page.

```
// Finish coding this function, and write your recursive helper below.
Set<string> melt(string s1, string s2) {
   Lexicon lex("EnglishWords.txt");
```

2. Classes (20 pts) Let's implement some graph functionality. You are ready for this! The `Graph` class stores information about an unweighted, undirected graph using an adjacency matrix (in the form of a `Grid<bool>` variable).

```
class Graph {
public:
   Graph(Grid<bool> matrix);
   void addEdge(int n1, int n2);
   void removeEdge(int n1, int n2);
   bool edge(int n1, int n2) const;
   Set<int> neighbors(int n1) const;
   int nodeCount() const;
   int edgeCount() const;
private:
   Grid<bool> _matrix;
   int _numNodes;
   // use this space to declare any other private variables or member functions you need



};
```

Constructor: Receives a grid as described below. Initialize all private class variables to reflect the graph contents. The `Grid<bool> matrix` is guaranteed to be a valid adjacency matrix representation of an undirected, unweighted graph. If `matrix[n1][n2]` is `true`, there is an edge from `n1` to `n2` in the graph (where `n1` and `n2` are node IDs, which are just integers on the range `0` through `_numNodes - 1`). Otherwise, there is no edge from `n1` to `n2`. This matrix is guaranteed to be well-formed: it will be a square (equal number of rows and columns, which is number of nodes in graph), and since the graph is undirected, `matrix[n1][n2]` = `matrix[n2][n1]` for all valid indices `n1` and `n2`. After instantiating a graph, the number of nodes it contains will not change, although edges may be added or removed.

Add edge: In **O(1)** time, add an edge between the given nodes if not already present.

Remove edge: Same as above, but remove the edge (if it exists) instead of adding. Runtime must be **O(1)**.

Edge: In **O(1)** time, `edge(n1, n2)` returns `true` if there is an edge from `n1` to `n2`. Otherwise, return `false`.

Get neighbors: In **O(n log n)** time, `neighbors(n1)` returns a set of IDs (integers) for all nodes adjacent to node `n1`.

Node count: In **O(1)** time, return the number of nodes in the graph.

Edge count: In **O(1)** time, return the number of edges in the graph. Since the graph is undirected, we do not count an edge from `n1` to `n2` and an edge from `n2` to `n1` as two distinct edges. That's just one edge.

```
// sample usage
Grid<bool> grid(3, 3);                    // graph will have 3 nodes, IDs 0 through 2
grid[1][1] = true;
grid[2][1] = true;
grid[1][2] = true;
Graph graph(grid);                        // graph has 3 nodes, 2 edges
graph.addEdge(0, 2);                      // graph has 3 nodes, 3 edges
EXPECT_EQUAL(graph.edge(2, 0), true);     // this edge was added above
EXPECT_EQUAL(graph.neighbors(1), {1, 2}); // 1 is adjacent to itself and 2
graph.removeEdge(2, 1);                   // graph has 3 nodes, 2 edges
EXPECT_EQUAL(graph.edge(1, 2), false);    // this edge was removed above
```

Implement this bare-bones `Graph` class, including the declaration of any additional private member variables or functions (in the box above) and full implementation of the constructor and member functions (on this page).

Note: In the event that a function receives an invalid node ID, throw an error.

This page is intentionally left blank for you to use as scratch paper.

We will not grade anything on this page unless you write a redirect from the original answer area to here.

This page is intentionally left blank for you to use as scratch paper.

We will not grade anything on this page unless you write a redirect from the original answer area to here.

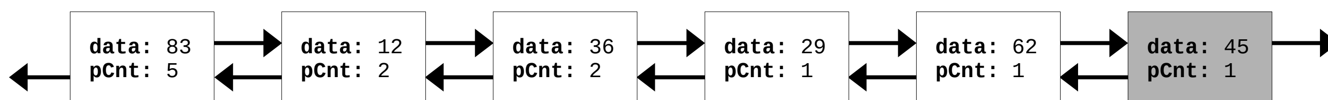
3. Linked Lists (20 pts)

Consider the following `Node` struct, which is designed to implement a variation of a doubly linked list:

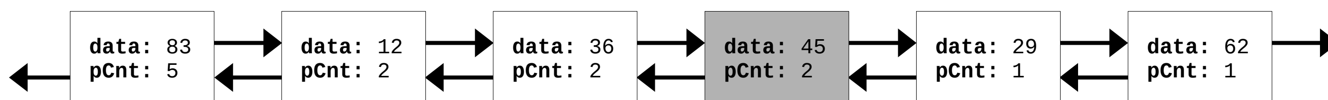
```
struct Node
{
   int data;
   int pCnt;    // count of the number of times this node has been pinged
   Node* next;  // pointer to next node
   Node* prev;  // pointer to previous node
};
```

Write a function called `ping()` that takes the head of a doubly linked list whose nodes are sorted by `pCnt` (from largest to smallest), and some `value` to search for among the `data` fields in the list. If `value` is found in the list, increment the node's `pCnt`, and then rearrange the list as needed to restore the sorted order of nodes. (Thus, our nodes end up sorted by how frequently they've been searched for. Neat!)

For example, suppose we call `ping(head, 45)` on the following list:

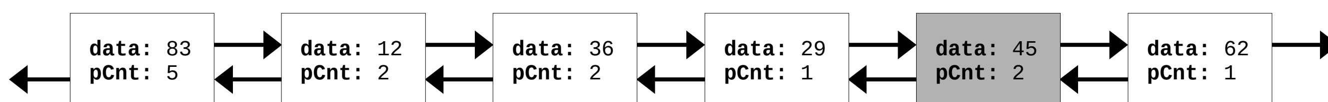


The resulting list would be as follows. Note that we have incremented the `pCnt` for the node with the value 45, and the nodes are still sorted by `pCnt` (largest to smallest):

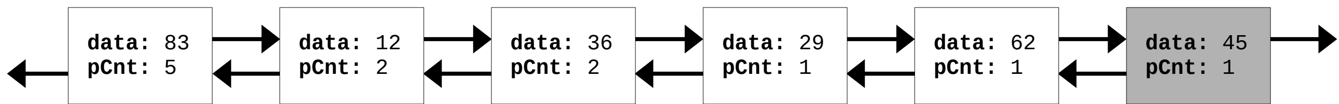


Following are various notes and restrictions for this problem:

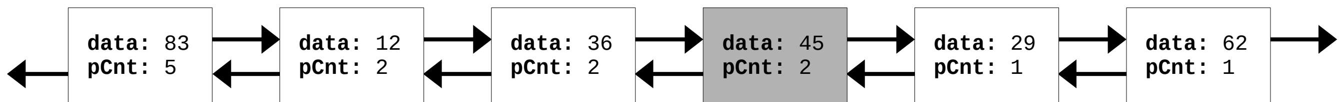
1. There are no duplicate `data` values in the list. The same value will never occur in more than one node.
2. If the `value` passed to this function is not found in the list, the list should not change at all.
3. The list may be empty, in which case `head` will be `nullptr`.
4. The head node's `prev` pointer and the tail node's `next` pointer are both null.
5. You may create `Node*` pointer variables, but you must not allocate (`new`) or deallocate (`delete`) any nodes.
6. You cannot change the `data` value within a node. If a node needs to be repositioned in the list, you must rewire the links between the nodes by modifying `next` and `prev` pointers.
7. Note that because we are incrementing `pCnt`, a node will only ever move left in the list (toward head), not right.
8. If a node needs to move after updating `pCnt`, start at the node in question and move left in the list to find its new position. Please do not sort the whole list or remove the node and then reinsert it by starting at the head of the list.
9. When moving a node, you should not percolate it to the left one by one. Instead, seek the position where it needs to move in the list, and *then* move it there. For instance, in the example shown above, there should **not** be an intermediary step where the list looks like this:



Repeated for your convenience as you start coding, here is the example list before calling `ping(head, 45)`:



Following is the list after calling `ping(head, 45)`. Note that we have incremented the `pCnt` for the node with the value 45, and the nodes are still sorted by `pCnt` (largest to smallest):



NOTE: We have started this function for you, but there is one line to fill in below, and then the rest of your code will be written on the following page. The space above can be used to trace through some of your ideas.

```
// Here is some code to get you started. Return true if we find the value, false otherwise.
bool ping(Node*& head, int value) {
   Node* current = head;
   while (current != nullptr && current->data != value) {
      current = current->next;
   }
   if (current == nullptr)
      return false;
   }

   current->pCnt++;  // If we get here, we must have found what we were looking for!

   // FINISH THE LINE BELOW! What conditions would allow us to leave the function at this time?

   if (
      return true;
   }
```

```
// Now, find the node BEFORE which the current node needs to be placed. In the example above,
// the goal is to find the node with 29. Be careful not to let this pointer become nullptr.
// Once you get this pointer where it needs to be, rewire the nodes as needed. You might need
// additional pointers for that. Be ever on the lookout not to dereference null pointers!
Node* before = current->prev;
```

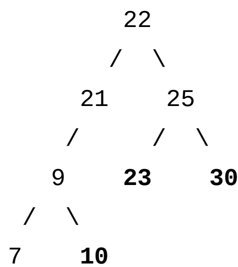
This page is intentionally left blank for you to use as scratch paper.

We will not grade anything on this page unless you write a redirect from the original answer area to here.

4. Binary Search Trees (20 pts)

a. Write a function that takes the `root` of a binary search tree and an integer `threshold` and returns the sum of all leaf node values that meet or exceed the `threshold` value. Avoid unnecessary inefficiency as you traverse the BST.

For example, given the following tree and a `threshold` of 10, the function should return 10 + 23 + 30 = 63.

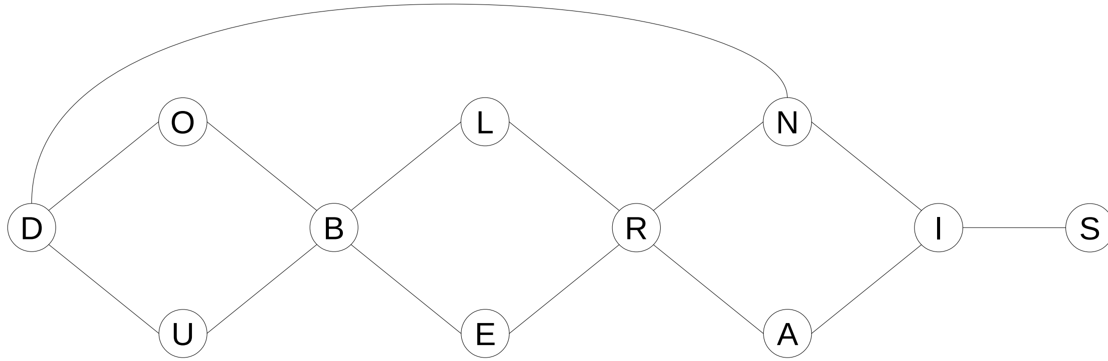



```
// struct for this problem:
struct TreeNode {
  int data;
  TreeNode* left;
  TreeNode* right;
};
```

```
// Complete the code using this function signature. You may not write any helper functions.
int threshSum(TreeNode* root, int threshold) {
```

b. What is the **best-case** runtime for the function above, and what situation would lead to that best-case runtime? (Be brief. Confine your answer to the space below in a fairly reasonable, non-squished font size.)

5. Short Answer (15 pts)



a. Indicate whether each of the following is a valid BFS for the graph above (write "yes" or "no" in the provided box):

R A I N S L E B O U D [ ] "yes" or "no"

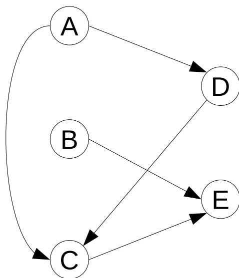
D O U N B I R E L S A [ ] "yes" or "no"

D O U B L E R A N I S [ ] "yes" or "no"

B O L D U E N R I A S [ ] "yes" or "no"

D O U N B E L R A I S [ ] "yes" or "no"

b. Indicate whether each of the following is a valid topological sort for the given graph ("yes" or "no"):



B A D E C [ ] "yes" or "no"

A D C B E [ ] "yes" or "no"

A B C D E [ ] "yes" or "no"

A D B C E [ ] "yes" or "no"

B A C D E [ ] "yes" or "no"

c. With respect to finding shortest paths in a graph, what does Dijkstra's algorithm take into account that BFS does not?

[ ]

6. General Problem Solving (10 pts)

Consider the following backtracking code, which is attempting to find and return a pointer to an exit in a maze:

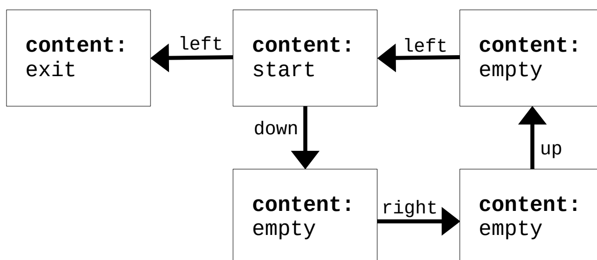
```
struct Cell
{
   string content;
   Cell* up;
   Cell* down;
   Cell* left;
   Cell* right;
};

Cell* findExit(Cell* start) {
   Cell* goal = nullptr;
   findExit(start, goal);
   return goal;
}

bool findExit(Cell* current, Cell*& goal) {
   if (current == nullptr) {
      return false;
   }

   if (current->content == "exit") {
      goal = current;
      return true;
   } else if (findExit(current->up, goal)) {
      return true;
   } else if (findExit(current->down, goal)) {
      return true;
   } else if (findExit(current->left, goal)) {
      return true;
   } else if (findExit(current->right, goal)) {
      return true;
   } else {
      return false;
   }
}
```

Given the following maze, our function is crashing from a stack overflow because it's stuck running in circles:



The root cause of this error is that the code is not keeping track of states it has already visited. On the following page, you will modify this code to solve this problem.

Modify the following code so that it somehow keeps track of which cells it has already visited and returns from any recursive call the brings it back to an already-visited cell.

- Efficiency matters. Implement a solution that is efficient both in terms of runtime and memory usage.
- You may **not** modify the `Cell` struct definition in any way, and you may **not** modify the contents of any of the cells as you journey through the maze.
- You **may**, however, make changes to the parameters being passed to the recursive function.
- You may not **remove** lines of code, but you may **add** lines or **slightly modify** some of the existing lines.
- Do not simply change the order of the recursive calls so that the function works for the maze on the previous page. The goal is to fix up the function so that it works on any maze it might receive.
- We have spaced out the following code so you can make your changes to it directly. Not every space requires a change, however, and you might need to draw some arrows to show where exactly you want to insert certain lines.

```
Cell* findExit(Cell* start) {

   Cell* goal = nullptr;


   findExit(start, goal                );


   return goal;
}

bool findExit(Cell* current, Cell*& goal            ) {

   if (current == nullptr) {
      return false;
   }



   if (current->content == "exit") {
      goal = current;
      return true;
   }


   else if (findExit(current->up, goal           )) {
      return true;
   }
   else if (findExit(current->down, goal         )) {
      return true;
   }
   else if (findExit(current->left, goal         )) {
      return true;
   }
   else if (findExit(current->right, goal        )) {
      return true;
   }


   else {
      return false;
   }
}
```

This page is intentionally left blank for you to use as scratch paper.

We will not grade anything on this page unless you write a redirect from the original answer area to here.