Practice Final 5

CS106B, Spring 2024

Last (**Family**) Name	
First (**Given**) Name	
Stanford E-mail	@stanford.edu

Exam Instructions

There are **5** questions worth a total of **100** 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)

In this problem, you will be given a vector of *pirateT* structs. The struct definition is as follows:

```
struct pirateT
{
  string pirateName;  // All names are guaranteed to be unique.
  int timeOwed;       // timeOwed is given in minutes.
};
```

Each pirate in the vector is indebted to Captain Bianca Trost and is a member of her crew. Their debts are measured in time owed (in minutes) performing favors for the captain.

The captain is curious whether, given a particular vector of pirates (all of whom have *timeOwed* > 0), she could call in their favors to create a rotation where some subset of them would take turns standing watch throughout the night over a valuable treasure she has recently brought aboard her ship. She has some stipulations for the watch rotation, though:

- The treasure must be under continuous watch through the night, which lasts for 480 minutes (8 hours).
- There should be only one pirate on watch at a time, so as not to double-dip into the debts they owe her.
- No pirate can spend more time on watch than what they owe to the captain.
- No pirate can spend more than 60 minutes on watch regardless of how much time they owe the captain, as she's worried about them dozing off if required to stand watch for too long.
- No pirate can serve more than one shift on watch. So, there is no benefit to us having a pirate spend less than the maximum time they are able to spend on their watch.
- Not all of the pirates are on good terms, and Captain Trost intends to use this to her advantage. No pirate can immediately follow another pirate in the watch rotation if those two pirates are on decent terms; they must instead be rivals. Captain Trost often does that to ensure that a changing of the guard has less of a chance of leading to a friendly conversation where two pirates might end up starting to plot a mutiny against her.
  - Luckily for her, the pirates on her ship are a dastardly group and have many bitter and deep-seated rivalries. She maintains a *rivals* map of type *Map<string, Map<string, bool>>*. In this map, *rivals*[*name1*][*name2*] is *true* if the two pirates are rivals. Otherwise, the value is *false*.
  - So, if our watch rotation is {A, B, C, D, E, F, G, H}, then A and B must be rivals, and B and C must be rivals, but it doesn't matter whether A and C are rivals or not.
- We don't necessarily have to use all the pirates in the vector. It might be possible to accomplish this task with a subset of them.

Write a recursive backtracking function that takes a vector of pirates and determines if it's possible to set up a 480-minute watch that satisfies the conditions above. Return *true* if so, *false* otherwise.

In solving this problem, you must abide by the following guidelines and restrictions:

- Your algorithm must be **recursive** and must use **backtracking** techniques to generate its results.
- You can use a vector to keep track of the pirate watch rotation you are building, if you wish. Beyond that, you must not create any auxiliary data structures.
- You can never modify the contents of a *pirateT* struct. You may add and remove structs from the *crew* vector, but you cannot change the contents of an individual struct. You also cannot modify *copies* of any of these structs.
- You should only explore sequences that could potentially lead to valid results. As soon as it becomes clear that a sequence you have generated cannot lead to any valid results, stop exploring that dead-end path.

- You should not iterate over the *rivals* map. Use it to determine whether two given pirates are rivals – not to get a list of rivals for a given pirate.

```
bool canGuard(Vector<pirateT> crew, Map<string, Map<string, bool>>& rivals) {
```

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

We will not grade anything on this page.

2. Classes (20 pts) In this question, you will create a *CharGobbler* class with a dynamically expanding and contracting array of characters. The basic class definition is:

```
const int kInitialCapacity = 10;  // use this as default array capacity

class CharGobbler {
public:
   CharGobbler();
   ~CharGobbler();
   void gobble(string s);
   void digest(int n);
   void trim();
   void resize(int n);
private:
   int _capacity;  // the capacity (overall length) of the internal char array
   int _size;      // number of cells occupied in char array (think _numFilled)
   char *_array;   // pointer to char array
   // use this space to declare any other private variables or member functions you need
}
```

Constructor: Dynamically allocate a *char* array and initialize all member variables as appropriate.

Destructor: Delete any dynamically allocated memory that is still lingering in heap space.

***Gobble*:** Add the characters from this string (in order, one by one) to the char array, starting at the next (leftmost) open position in the array. If the array is too small, you should use dynamic memory management to expand it to length 2*k*, where *k* is the number of occupied cells you'll have in the array *after* the string in question is added.

For example, if we have an array of length 10 that contains 7 characters (so, 3 spaces free) and we try to add "squash" to the array, we will have a total of 7 + 6 = 13 characters (too many for the array of length 10 to hold), and so we need to first expand the array to length 13 * 2 = 26. Be sure to avoid memory leaks if you expand an array, and be sure this function always updates all member variables as appropriate. (See the following page for an additional example.)

Digest: Remove the first *n* strings from the char array, shifting all remaining characters over to take their place. This requires you to keep track somehow of how many characters each string contains when it is added. In doing so, you may use an auxiliary data structure to track the length of each string you have added to the char array, but not the strings themselves. Be sure to keep that data structure up to date any time you *gobble()* or *digest()*, and be sure this function updates all member variables as appropriate.

Consider the following example (which is also depicted on the following page): suppose we start with an empty *CharGobbler* and add "fork", "bot", and "sage" (in that order). We would want to know that the respective string lengths are 4, 3, and 4. If someone calls *digest(2)*, we would remove the first 4 + 3 = 7 characters ("forkbot") and be left with "sage". The "sage" characters shift over to the first four cells of the array, and the remaining cells in the array can contain any values. You can let garbage characters accumulate in those positions, because *_size* tells us they're not valid.

If *n* exceeds the number of strings in the *CharGobbler*, you should simply remove all the strings it contains.

If this function results in the number of occupied cells being ≤ 25% of the array's capacity, use dynamic memory management to cut the array length in half.

Trim: Use dynamic memory management to reduce the length of the array to match the number of cells currently occupied. Update any member variables as appropriate.

Resize: Use dynamic memory management to change the length of the character array to *n*. This function could either grow or shrink the array, but if *n* < *_size*, throw an error so that we don't lose valid characters that have been added to the CharGobbler. If expanding the array, you can fill unused cells with any garbage character you want, or even leave them uninitialized. This function should update any member variables as appropriate.

Throughout this problem, be sure to avoid memory leaks (orphaned memory) using *delete* as necessary.

If there is a straightforward place where one of these functions can call another one, do that instead of duplicating code.

Sample usage:

```
CharGobbler c;          // [?, ?, ?, ?, ?, ?, ?, ?, ?, ?]
                        // ^ use kInitialCapacity for initial array length

c.gobble("fork");       // [f, o, r, k, ?, ?, ?, ?, ?, ?]
c.gobble("bot");        // [f, o, r, k, b, o, t, ?, ?, ?]
c.gobble("sage");       // [f, o, r, k, b, o, t, s, a, g, e, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]
                        // ^ array expanded to length (4 + 3 + 4) * 2 = 22

c.digest(2);            // [s, a, g, e, ?, ?, ?, ?, ?, ?, ?]
                        // ^ digested both "fork" and "bot"; 4/22 cells are occupied (less than
                        //   25% full), so array length contracts to half (22/2 = 11)

c.gobble("rose");       // [s, a, g, e, r, o, s, e, ?, ?, ?]
c.digest(1);            // [r, o, s, e, ?, ?, ?, ?, ?, ?, ?]

c.trim();               // [r, o, s, e]
                        // ^ array length is reduced to match _size (4)
```

Implement this bare-bones *CharGobbler* class:

- The declaration of any additional private member variables or functions should go in the box on the previous page.
- The full implementation of the member functions should go on the following two pages.

This space is just for scratch work.

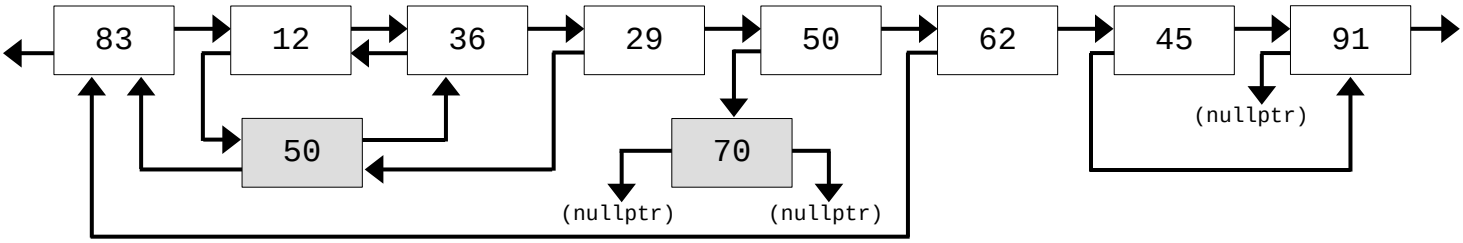
We will not grade anything on this page.

Write your implementation of the *CharGobbler* class member functions on this and the following page.

Write your implementation of the *CharGobbler* class member functions on this and the previous page.

3. Linked Lists (20 pts) In this problem, you'll write two functions to help salvage a doubly linked list whose *next* pointers are valid, but whose *prev* pointers have been corrupted. In particular, the linked lists you have to handle in this problem will have the following properties:

1. The *next* pointers in the list always form a perfect (singularly) linked list with no cycles/circularity. Hooray!
2. The *prev* pointers are sometimes valid, but sometimes they're corrupted. Specifically, those pointers might be corrupted in the following ways:
   a. Sometimes they point to nodes much earlier in the list, as with 62's *prev* pointer below.
   b. Sometimes they point to nodes later in the list, as with 45's *prev* pointer below.
   c. Sometimes they point to *null* when they should point to an actual node, as with 91's *prev* pointer below.
   d. Sometimes they point to extraneous nodes that are not even reachable when we follow the *next* pointers starting from the head of the list. The gray nodes in the diagram below are extraneous.



Some additional notes:

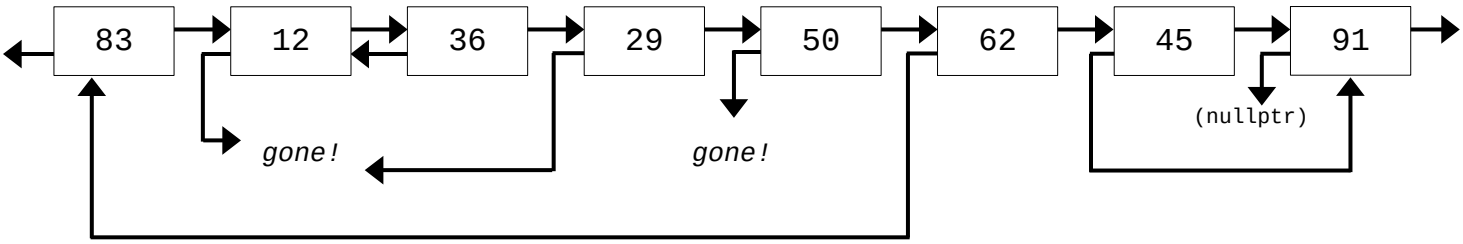
1. The list is not guaranteed to contain any nodes. It could be empty.
2. The head's *prev* pointer isn't guaranteed to be *nullptr*.
3. It's possible for multiple nodes' *prev* pointers to point to the same extraneous node, as with 12 and 29 both pointing to 50 above.
4. It's possible for some nodes to have the same values. Note that 50 appears in a valid node above, as well as an extraneous node. There is no restriction on the values whatsoever, and they should not be used to attempt to track or determine which nodes are extraneous or not.
5. An extraneous node will always be reachable via a *prev* pointer from a valid node in the list. We will never have a situation where there is an extraneous node that is only reachable via another extraneous node. For example, we would not have node 70 in the diagram lead to a third extraneous node that can only be reached through node 70.

The first function you write, *seekAndDestroy()*, takes the head of one of these broken linked lists and deletes all the extraneous nodes it contains (nodes that cannot be reached via *next* pointers when starting at the head of the list). In doing so, take care never to delete the same node twice, even if there are multiple *prev* pointers pointing to the same node (as with 12 and 29 pointing to 50 above), and never delete a valid node (e.g., don't delete 83 simply because 62's *prev* pointer points to it or 91 simply because 45's *prev* points to it; 83 and 91 are valid members of our list!).

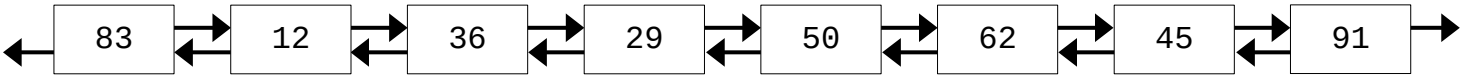
As part of the credit for this function, you must come up with a way to keep track of all the valid nodes in the list and **efficiently** check whether each *prev* pointer points to a valid node or an extraneous one, as well as to ensure that you don't *delete* the same node more than once. Credit will be awarded for correctness as well as efficiency.

The second function you write, *fixPrevLinks()*, will fix the *prev* pointers to point to the correct nodes in the list. Assume we call *seekAndDestroy()* before calling *fixPrevLinks()* and that *seekAndDestroy()* works as intended.

After calling *seekAndDestroy()* on the linked list above, we would have:



Then, after calling *fixPrevLinks()* on that list, we would have:



In both functions:

1. Do not create any new nodes.
2. Do not change the contents of any nodes, even if you plan to delete them.
3. Please do not write any helper functions.

The struct definition is:

```
struct Node
{
   int data;
   Node *next;
   Node *prev;
};
```

Please write your functions on the following two pages.
Sorry for this big, awkward, empty space.

```
void seekAndDestroy(Node *head) {
```

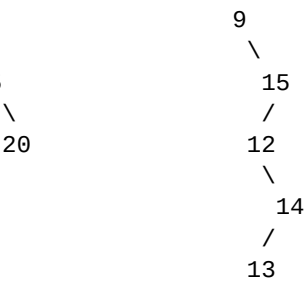
There is space for your fixPrevLinks() function on the following page.

```
void fixPrevLinks(Node *head) {
```

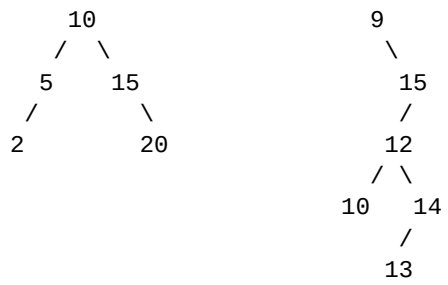
4. Binary Trees (20 pts)

Write a function that takes the root of a binary search tree and determines whether it has devolved into a linked list (i.e., it is maximally tall). If so, return *true*. Otherwise, return *false*. You may assume the given tree is a BST. For example:

The following have devolved into linked lists:



The following are not linked lists:

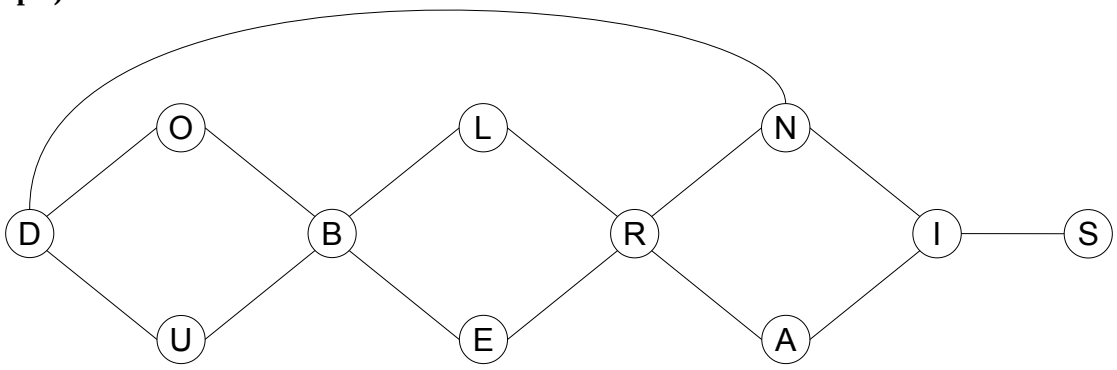


You **cannot** write any helper functions. The *Node* struct and function signature are as follows:

```
struct Node
{
   int data;
   Node *left;
   Node *right;
};

bool isLinkedList(Node *root) {
```

5. Graphs (20 pts)



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

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

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

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

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

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

b. In the box, list all the nodes that could come **first** in a valid topological sort for the directed graph below:

[ ]

c. In the box, list all the nodes that could come **last** in a valid topological sort for the directed graph below:

[ ]

d. In the box, give one valid topological sort for this directed graph:

[ ]