Practice Final Exam

☞ ***Note:*** This handout has been updated with new exam policies. ☜

You are welcome to consult any non-human sources you would like when completing the final exam. You are not allowed to communicate with any other humans during the exam except the course staff. This means, for example, that it is perfectly permissible to visit the course website, to open Qt Creator and run code, and to Google search for anything you'd like. However, you are not allowed to communicate with other humans in any way during the exam, nor are you permitted to post the questions anywhere where other humans could see them or offer advice or suggestions on them (for example, by posting on a public Q&A website or emailing questions to other people).

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you get partial credit if they help us determine what you were trying to do. You do not need to worry about efficiency unless a problem states otherwise. You don't need to prototype helper functions you write, and you don't need to explicitly `#include` libraries you want to use.

This exam is "self-contained" in the sense that if you're expected to reference code from the lectures, textbook, assignments, or section handouts, we'll explicitly mention what that code is and provide sufficient context for you to use it.

This practice exam was originally designed as a closed-book, closed-computer, and limited-note exam, though as mentioned above that is not the case with this quarter's exam. Each problem here was used on an actual CS106B final exam within the past few years. There are 56 total points.

It's been a pleasure teaching CS106B this quarter. Best of luck on the final exam!

Problem One: Data Structures (8 Points)

On Assignment 5, you gained experience writing code that met specific runtime bounds. On Assignments 5, 6, and 7, you saw how to implement a number of common container types. This question is designed to let you demonstrate what you've learned about algorithmic analysis and data structure design.

Below are four functions. We picked one of those functions and ran it on many different values of *n*. We captured the output of that function on each of those inputs, along with the runtime. That information is printed in the table to the right.

```
int function1(int n) {
   Stack<int> values;
   for (int i = 0; i < n; i++) {
      values.push(i);
   }

   int result;
   while (!values.isEmpty()) {
      result = values.pop();
   }

   return result;
}
```

```
int function2(int n) {
   Queue<int> values;
   for (int i = 0; i < n; i++) {
      values.enqueue(i);
   }

   int result;
   while (!values.isEmpty()) {
      result = values.dequeue();
   }

   return result;
}
```

```
int function3(int n) {
   Set<int> values;
   for (int i = 0; i < n; i++) {
      values.add(i);
   }

   int result;
   for (int value: values) {
       result = value;
   }

   return result;
}
```

```
int function4(int n) {
   Vector<int> values;
   for (int i = 0; i < n; i++) {
      values.add(i);
   }

   int result;
   while (!values.isEmpty()) {
      result = values[0];
      values.remove(0);
   }

   return result;
}
```

n	*Time*	*Return Value*
100,000	0.137s	`99999`
200,000	0.274s	`199999`
300,000	0.511s	`299999`
400,000	0.549s	`399999`
500,000	0.786s	`499999`
600,000	0.923s	`599999`
700,000	0.960s	`699999`
800,000	1.198s	`799999`
900,000	1.335s	`899999`
1,000,000	1.472s	`999999`

The actual questions are on the next page.

i. **(4 Points)** For each of these pieces of code, tell us its big-O runtime as a function of n. No justification is required.

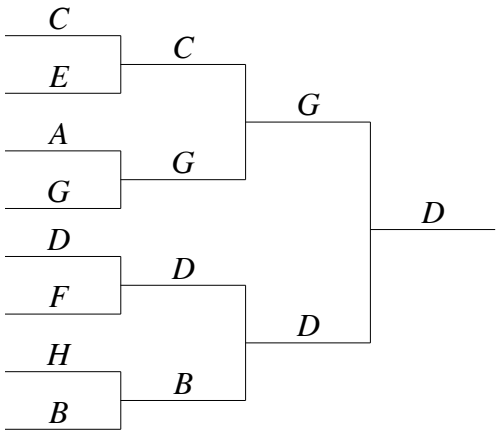
ii. **(2 Points)** For each of these pieces of code, tell us whether that function could have given rise to the return values reported in the rightmost column of the table. No justification is required.

iii. **(2 Points)** Which piece of code did we run? How do you know? Justify your answer in at most fifty words.

Problem Two: Recursive Problem-Solving I (12 Points)

This question explores how to use recursion to determine who would win an elimination tournament and, somewhat mischievously, how to set up a tournament so that your favorite player ends up winning.

A ***tournament bracket*** is a type of tournament structure for a group of players. The players are lined up in some initial order (here, ***C, E, A, G, D, F, H, B***, as you can see on the left column). The players are paired off by their positions, with the first player competing against the second, the third player competing against the fourth, etc. The winner of each game advances to the next round, and the loser is eliminated. For example, in the first round, player *C* won her game against player *E*, player *A* lost his game against player *G*, player *D* won her game against player *F*, and player *H* lost his game against player *B*. Those players are again paired off, making sure to preserve their relative ordering. Thus players *C* and *G* and players *D* and *B* face off in the second round, with players *G* and *D* winning and advancing to the next round. Finally, players *G* and *D* face off, and player *D* emerges victorious. Since she's the last player remaining, player *D* is the overall winner of the tournament.



Your task in the first part of this problem is to write a ***recursive*** function

```
string overallWinnerOf(const Vector<string>& initialOrder);
```

that takes as input a vector representing the ordering of the players in the initial tournament bracket, then returns the name of player who ends up winning the overall tournament.

In the course of implementing this function, assume you have access to a helper function

```
string winnerOf(const string& p1, const string& p2);
```

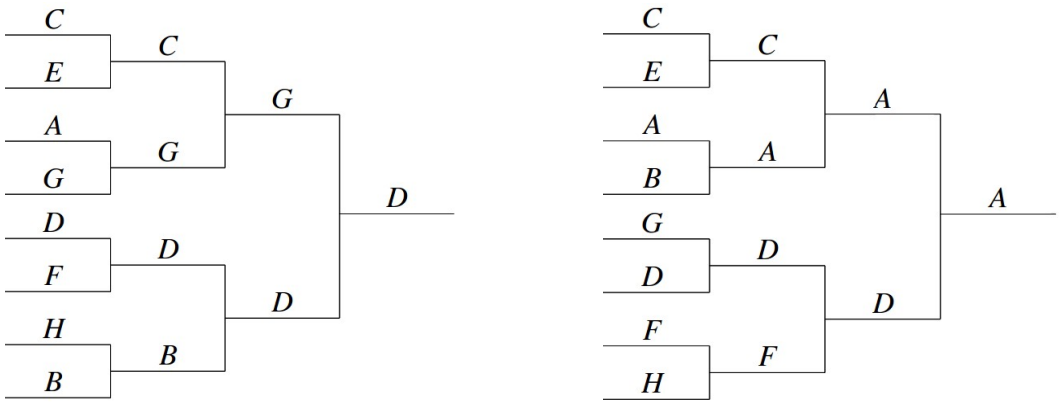
that takes as input the names of two players, then returns which of those two players would win in a direct matchup.

Some notes on this problem:

- You can assume the number of players is a perfect power of two (for example, 1, 2, 4, 8, 16, 32, 64, etc.), so there will never be a case where there's an "odd player out" who isn't assigned to play a game in some round. This also means you'll never get a list of zero players.
- You should not make any assumptions about how a matchup between two players would go based on previous matchups. To determine how a match would go, call the `winnerOf` function.
- This part of the problem ***must*** be implemented recursively – that's what we're testing here. 😄

```
string overallWinnerOf(const Vector<string>& initialOrder);
string winnerOf(const string& p1, const string& p2) {
```

Changing the initial order of players in a tournament can change the outcome of that tournament. For example, imagine that player *A* is a very strong player who would win against every player except player *G*. In the tournament bracket shown to the left, player *A* immediately gets eliminated from the tournament. On the other hand, in the tournament bracket shown to the right, player *A* ends up winning the entire tournament, since player *G* get eliminated before she gets a chance to play a game against player *A*.



Because the winner of a tournament depends on the player ordering, in some cases it is possible to "rig" the outcome of a tournament by changing the initial player ordering. For example, if you were a huge fan of player *D* and wanted her to be the overall winner, and if you knew in advance which opponents player *D* would win against, you could try different orderings and come up with the bracket to the left. If you wanted player *A* to be the overall winner, then you could set up the players in the order to the right. In some cases, there's nothing you can do to ensure someone will win the tournament. For example, a player who you know will lose every game they play will always lose their first game and be eliminated.

Your task is to write a function

```
bool canRigFor(const string& player, const HashSet<string>& allPlayers,
               Vector<string>& initialOrder);
```

that takes as input the name of a player and a set containing the names of all the players in the tournament, then returns whether there's some initial ordering of the players that will cause that player to be the overall winner. If so, your function should fill in `initialOrder` with one such possible ordering.

Some notes on this problem:

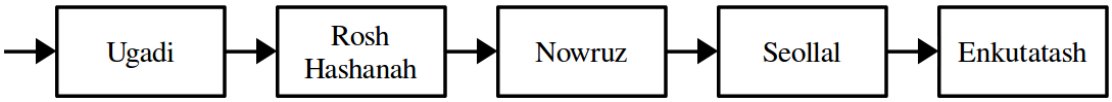
- You should make all the same assumptions about the input as in the first part of this problem: the number of players is always going to be a perfect power of two, that you should use the `winnerOf` function to determine who would win in a matchup, etc.
- Feel free to use the `overallWinnerOf` function from part (i) of this function in the course of solving this problem, even if you weren't able to get a working solution.
- Don't worry about efficiency. We're expecting you to use brute force here, and no creative optimizations are necessary.
- This part of the problem must be done recursively. Again, that's what we're aiming to test here.

```
bool canRigFor(const string& player, const HashSet<string>& allPlayers,
               Vector<string>& initialOrder) {
```

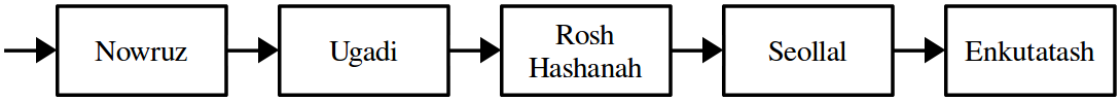
Problem Three: Linear Structures (12 Points)

YouTube and Facebook have tons of data (literally, if you weigh all the disk drives they use to store things), though most of that data is rarely accessed. When you visit YouTube, for example, the videos that will show up will likely be newer videos or extremely popular older videos, rather than random videos from a long time ago. Your Facebook feed is specifically populated with newer entries, though you can still access the older ones if you're willing to scroll long enough.

More generally, data sets are not accessed uniformly, and there's a good chance that if some piece of data is accessed once, it's going to be accessed again soon. We can use this insight to implement the set abstraction in a way that speeds up lookups of recently-accessed elements. Internally, we'll store the elements in our set in an unsorted, singly-linked list. Whenever we insert a new element, we'll put it at the front of the list. Additionally, and critically, whenever we *look up* an element, we will reorder the list by moving that element to the front. For example, imagine our set holds the strings Ugadi, Rosh Hashanah, Nowruz, Seollal, and Enkutatash in the following order:

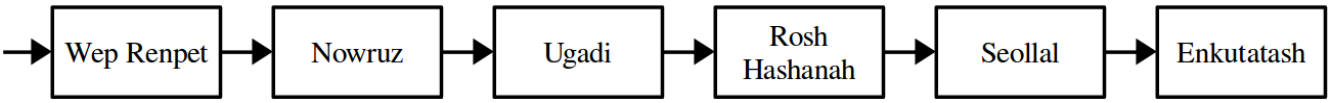


If we look up Nowruz, we'd move the cell containing Nowruz to the front of the list, as shown here:

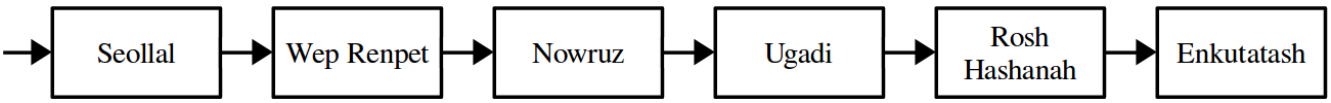


If we now do a look up for Nowruz again, since it's at the front of the list, we'll find it instantly, without having to scan anything else in the list.

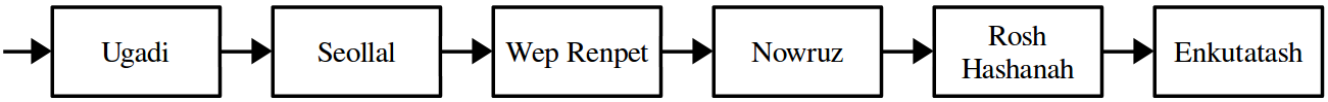
If we now insert the new element Wep Renpet, we'd insert it at the front of the list, as shown here:



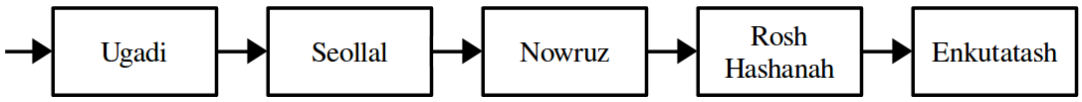
Now, if we do a lookup for Seollal, we'd reorder the list as follows:



If we do an insertion to add Ugadi, since it's already present in the set, we just move it to the front of the list, rather than adding another copy. This is shown here:



Finally, to remove an element from the list, we'd just delete the indicated cell out of the list. For example, deleting Wep Renpet would make the list look like this:



Your task is to implement this idea as a type called `MoveToFrontSet`. The interface is given at the bottom of this page. You're responsible for implementing a constructor and destructor and for implementing the `contains`, `add`, and `remove` member functions.

Some notes on this problem:

- Your implementation ***must*** use a singly-linked list, not a doubly-linked list, but aside from that you can represent the linked list however you'd like.
- When doing a move-to-front, you ***must*** actually rearrange the cells in the list into the appropriate order. Although it might be tempting to simply swap around the strings stored within those cells, this is significantly less efficient than rewiring pointers, especially for long lists.
- You're welcome to add any number of private helper data members, member functions, or member types that you'd like, but you must not modify the public interface provided to you.
- Your implementations of the member functions in this class do not need to be as efficient as humanly possible, but you should avoid operations that are unnecessarily slow or that use an unreasonable amount of auxiliary memory.

As a hint, you may find it useful to have your `add` and `remove` implementations call your `contains` member function and use the fact that it reorganizes the list for you.

```
class MoveToFrontSet {
public:
    MoveToFrontSet();  // Creates an empty set
    ~MoveToFrontSet(); // Cleans up all memory allocated

    bool contains(const string& str); // Returns whether str is present.
    void add(const string& str);      // Adds str if it doesn't already exist.
    void remove(const string& str);   // Removes str if it exists.

private:
    /* Add anything here that you'd like! */









};
```

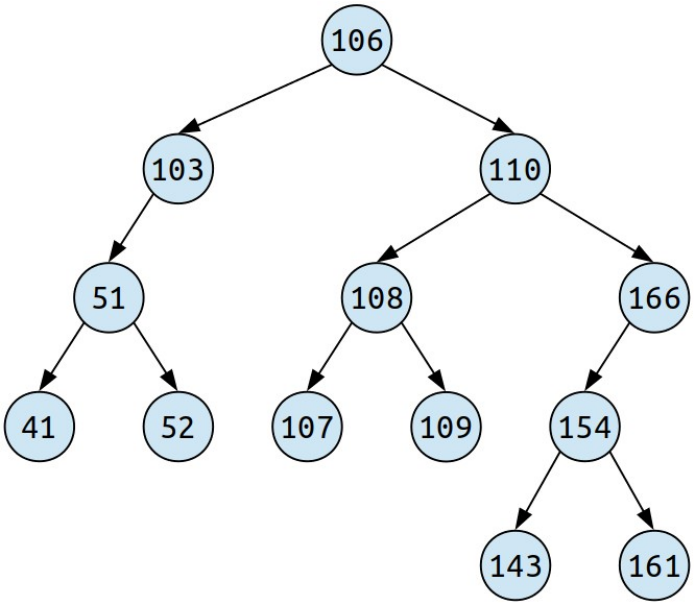
```
/* Initializes the set so that it's empty. */
MoveToFrontSet::MoveToFrontSet() {









}

/* Cleans up all memory allocated by the set. */
MoveToFrontSet::~MoveToFrontSet() {









}
```

```
/* Returns whether the specified element is in the set. If so, reorders the list so
 * that the element is now at the front. If not, the list order is unchanged.
 */
bool MoveToFrontSet::contains(const string& str) {
```

```
/* Adds the specified element to the list, if it doesn't already exist. Either way,
 * the element should end up at the front of the list.
 */
void MoveToFrontSet::add(const string& str) {









}
/* Removes the specified element from the set. If that element doesn't exist, this
 * function should have no effect and should not reorder anything.
 */
void MoveToFrontSet::remove(const string& str) {









}
```

Problem Four: Binary Search Trees (12 Points)

In a binary search tree, the ***lower bound*** of a key is the node in the tree with the smallest value greater than or equal to the key, and the ***upper bound*** of a key is the node in the tree with the largest value less than or equal to the key. For example, in the BST shown here, the lower bound of 137 is the node containing 143, and the upper bound of 137 is the node containing 110.



If the key is less than all the values, its upper bound is `nullptr`, so the the upper bound of 15 is `nullptr`. (The lower bound of 15 is the node containing 41.)

Similarly, if the key is greater than all elements in the BST, its lower bound is `nullptr`. For example, the lower bound of 261 is `nullptr`. (The upper bound of 261 is the node containing 166.)

In the event that the key happens to appear inside the BST, then the key itself is its own lower bound and upper bound. For example, the lower and upper bounds of 106 are each the node containing 106.

It is a bit confusing that a key's lower bound is always at least as large as its upper bound, but, alas, that is the naming convention we use.

Your task is to implement a function

```
Bounds boundsOf(Node* root, int key);
```

that takes in a pointer to the root of a BST, along with an integer key, then returns the lower and upper bound of that key in the tree. Here, `Node` and `Bounds` are `structs` defined as follows:

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

```
struct Bounds {
    Node* upperBound;
    Node* lowerBound;
};
```

Some notes on this problem:

- For full credit, your implementation should run in time O(h), where h is the height of the tree. This means that you can't necessarily explore the whole tree to find the upper and lower bounds. Due to how a BST is structured, though, you shouldn't need to check every node.
- You may want to draw some pictures before diving into this problem. In particular, think about the recursive intuition for how BSTs are structured.
- There can be any number of nodes in the tree, including zero, and there are no restrictions on what the key can be.
- For full credit, you should not use any of the container types (`Map`, `Set`, `Vector`, `Lexicon`, etc.).

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

```
struct Bounds {
    Node* upperBound;
    Node* lowerBound;
};
```

```
Bounds boundsOf(Node* root, int key) {
```

(Extra space for your answer to Problem Four, if you need it.)

Problem Five: Recursive Problem-Solving II (12 Points)

On Assignment 4, you wrote recursive code to determine whether it was possible to find objects of various types. This question serves as a coda to our treatment of recursive exploration and is designed to let you show us what you've learned in the process.

You'd like to put together a playlist for your next workout. You'd like to design the playlist so that

- its length is ***exactly equal*** to the length of your workout, and
- no song appears too many times in the playlist.

How many times is "too many times?" That depends. For a short workout, you might not want to hear the same song multiple times. For a long workout, you might be okay hearing the same song three or even four times. We'll assume that when you sit down to start creating the playlist, you'll have some magic number in mind.

Write a function

```
bool canMakePlaylist(const Vector<int>& songLengths,
                     int workoutLength, int maxTimes);
```

that takes as input a list of the lengths of the songs you're considering putting on your playlist, along with the length of your workout and the maximum number of times you're comfortable hearing a particular song. The function then returns whether there's a playlist of ***exactly*** length `workoutLength` which doesn't include any song more than `maxTimes` times.

Some notes on this problem:

- You must implement this function recursively.
- You can assume that `workoutLength` ≥ 0, that `maxTimes` ≥ 0, and that the length of each song is also greater than or equal to zero. You don't need to handle the case where any of these quantities are negative.
- There can be any number of songs in `songLengths`, including zero.
- You just need to tell us whether there is a possible playlist with the given length, not what songs are on that playlist or what order you'd put them in.
- Each song may appear multiple times, and not all songs necessarily need to be used.
- The total length of all songs in your playlist should equal `workoutLength`. Note that this is not the same thing as saying that the total *number* of songs in your playlist should equal `workoutLength`.
- Be careful – greedy solutions won't work here. There are combinations of song lengths, workout lengths, and maximum repeat counts where the only way to build a playlist of exactly the right length is to use fewer copies of a particular song than you're allowed to.
- Your solution does not need to be as efficient as possible, but to receive full credit your solution must not use a recursive strategy that is needlessly inefficient.

```
bool canMakePlaylist(const Vector<int>& songLengths,
                     int workoutLength, int maxTimes) {
```