CS106B

Winter 2023

Practice Final Exam 1

This is a printable version of the first set of practice problems from the practice final exam. It's designed to have the same format and style as the actual CS106B final exam.

Solutions can be found by visiting the Additional Practice Problems resource on the CS106B website and looking up solutions to the corresponding problems.

Problem One: Word Walks

Your first task in this question is to finish an implementation of a function

```
bool wordsOverlap(const string& left, const string& right);
```

that takes as input two words, then returns whether the last letters of the string `left` overlap with the first letters of the string `right`. For example, the strings "wonder" and "derby" overlap like this:

```
wonder
   derby
```

The words "orange" and "angel" overlap like this:

```
orange
  angel
```

The words "violin" and "naqareh" overlap like this:

```
violin
     naqareh
```

The words "aa" and "aardvark" can overlap in two different ways:

```
aa              aa
 aardvark       aardvark
```

Any word overlaps with itself; for example, here's "springbok" overlapping itself:

```
springbok
springbok
```

We've given a partial implementation of this function in the space below. Please fill in the rest of this implementation by filling in the blanks. Do not add, remove, or edit code anywhere else.

```
bool wordsOverlap(const string& left, const string& right) {
    for (int i = 0; _____________________; i++) {
        if (left.substr(________) == right.substr(_______)) {
            return true;
        }
    }
    return false;
}
```

We will grade whatever you write in this box. Use the surrounding space for scratch work.

If you have a collection of words, you can form a ***word walk*** by ordering the words so that each pair of consecutive words overlap. Here's a few sample word walks; I promise these are all English words.

```
absorbants
     antsiest
        siestas
           stash
             ashiness
                  essentials
                         also
```

Another example:

```
euchromatic
       ticktacking
              kingbolt
                  boltonia
                       niacin
                          cinnamyl
                               mylonite
```

And another:

```
tenorite
     termers
         erst
           startlingly
                     yogas
                         synaloephas
                                  simpering
```

In extreme cases, you might have one word fully subsume another; here's an example of this:

```
preponderate
preponderated
      derated
        rated
          tediously
                slyest
                    stepson
```

Your task is to write a function

```
Optional<Vector<string>> makeWordWalkFrom(const Set<string>& words);
```

that takes in a nonempty set of words, then returns a word walk that uses all the words in that set exactly once. If there is no way to make such a word walk, this function returns `Nothing`.

Some notes on this problem:

- Feel free to use the `overlapsWith` function from the previous part of this problem in your solution. You can assume that function works correctly.
- Your function simply needs to tell us whether it's possible to form a word walk using each of the given words exactly once. It doesn't need to tell us what that walk is, if it exists.
- You can assume the input `Set` contains at least one word and don't need to handle the case where the `Set` is empty.
- Your solution does not need to be as efficient as possible, but solutions that contain large inefficiencies will not receive full credit.
- You do need to solve this problem recursively; that's kinda what we're testing here.

```
/* From above. You can assume this function works as intended. */
bool wordsOverlap(const string& left, const string& right);

Optional<Vector<string>> makeWordWalkFrom(const Set<string>& words) {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's additional space for your answer on the next page.

Additional space for your answer to Problem One, if you need it.

Problem Two: Perfect Hashing

We've seen three types of hash tables this quarter: chained hashing, linear probing, and Robin Hood hashing. Each of these hashing strategies is designed to handle the hash collisions that result when items are added. But what if there were no collisions at all? That would make it a lot easier to build a hash table.

Suppose that, somehow, we know in advance all the items that we're ever going to store in the hash table. If we can find a hash function where no two of the items we're storing have the same hash code (a ***perfect hash function***), we can build a ***perfect hash table*** using the following strategy:

- As in linear probing and Robin Hood hashing, maintain an array of slots, each of which is either empty or holds a single item.
- Compute each item's hash code and place each item in the slot associated with its hash code. Since no two items have the same hash code, there are no collisions and we don't need to worry about elements "spilling out" of their home location.
- To check if an item is in the table, compute its hash code and jump to the spot in the table where it should be. If that slot is full and contains the item, great! The item is present. Otherwise it isn't, and we don't need to look anywhere else.

As you can see, this is a lot easier than regular linear probing!

Of course, this presupposes that we have a perfect hash function (a hash function that, given our set of items to store, produces no collisions). How might we find one? In practice, we use a cute strategy. Suppose we have a function

```
HashFunction<string> randomHashForSize(int numSlots);
```

that, given a number of slots, returns a randomly-selected hash function that outputs hash codes between 0 and `numSlots` - 1, inclusive. We can find a perfect hash function as follows:

1. Choose a random hash function.
2. Distribute the elements according to that hash function.
3. If there was a hash collision, go back to (1) and try again.

One last detail: how big should our array of slots be? For technical reasons beyond what we can cover on this exam, if you are building a perfect hash table for a set of n elements, you should make the table size exactly n^2 to improve the odds that no two items collide.

Your task is to implement the constructor, destructor, and contains functions for a perfect hash table. Here's a preliminary class definition. Implement the three functions we asked for, adding whatever you'd like to the private section of the class that you need.

As a note, aside from the `Set<string>` given in the constructor, you must not use any container types (e.g. `Vector`, `Map`, etc.) and must do all your own memory management.

```
/* This page contains the class definition for PerfectHashTable, along with some
 * details of how the type works. Do not write anything here; there's space for
 * your answer on the next pages.
 */

class PerfectHashTable {
public:
    /* Constructor: PerfectHashTable(const Set<string>& values);
     * Usage: PerfectHashTable(values);
     * ------------------------------------------------------------------------
     * Constructs a perfect hash table storing the specified set of values.  The
     * number of buckets will be n^2, where n is the number of values.
     */
    PerfectHashTable(const Set<string>& values);

    /* Destructor: ~PerfectHashTable();
     * Usage: (implicit)
     * ------------------------------------------------------------------------
     * Frees all resources allocated by this PerfectHashTable.
     */
    ~PerfectHashTable();

    /* Member function: contains(const string& value) const;
     * Usage: if (pht.contains("dikdik")) { … }
     * ------------------------------------------------------------------------
     * Returns whether the specified item contained in the hash table.
     */
    bool containsKey(const string& key) const;

private:
    /* Up to you! */

};

/* Returns a randomly-generated hash function for the given number of slots. This
 * function is provided to you and you do not need to implement it yourself.
 */
HashFunction<string> randomHashForSize(int numSlots);
```

```
/* Use the space here to write out the contents of the private section of the
 * PerfectHashTable class.
 */
private:




/* Implement the constructor here. Remember that a table holding n elements
 * should have n^2 slots.
 */
PerfectHashTable::PerfectHashTable(const Set<string>& elems) {









}
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work.

```
/* Implement the destructor here. */
PerfectHashTable::~PerfectHashTable() {









}

/* Implement contains here. */
bool PerfectHashTable::contains(const string& elem) const {









}
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's more space for your answer on the next page.

Additional space for your answer to Problem Two, if you need it.

Problem Three: Social Network Scaling

While most researchers agree that the value of a social network grows as the number of users grows, there's significant debate about precisely how that value scales as a function of the number of users.

Suppose there's a social network whose value, with its current number of users, is $10,000,000.

i. ***Sarnoff's Law*** states that the value of a network is $O(n)$, where n is the number of users on the network. Assuming Sarnoff's law is correct, estimate how much the social network needs to grow to have value $160,000,000. Justify your answer. Your answer should be no more than 50 words long.

Write your answer here. Anything else will be considered scratch work.

ii. ***Metcalfe's Law*** states that the value of a network is $O(n^2)$, where n is the number of users on the network. Assuming Metcalfe's law is correct, estimate how much the social network needs to grow to have value $160,000,000. Justify your answer. Your answer should be no more than 50 words long.

Write your answer here. Anything else will be considered scratch work.

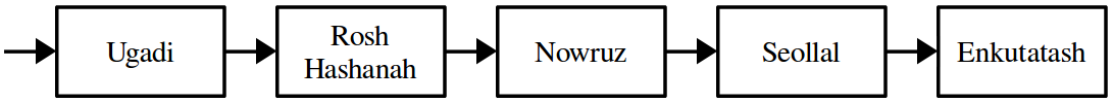
iii. ***Reed's Law*** states that the value of a network is $O(2^n)$, where n is the number of users on the network. Assuming Reed's law is correct, estimate how much the social network needs to grow to have value $160,000,000. Justify your answer. Your answer should be no more than 50 words long.

Write your answer here. Anything else will be considered scratch work.

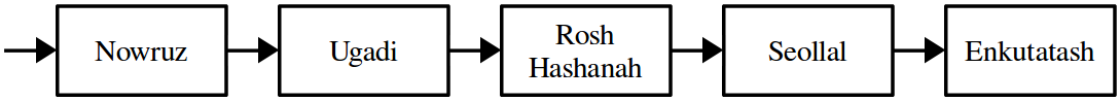
Problem Four: Self-Organizing Lists

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:

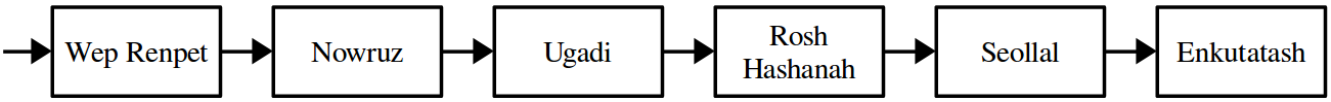


If we look up Nowruz, we'd move the cell containing Nowruz to the front of the list, as shown here:

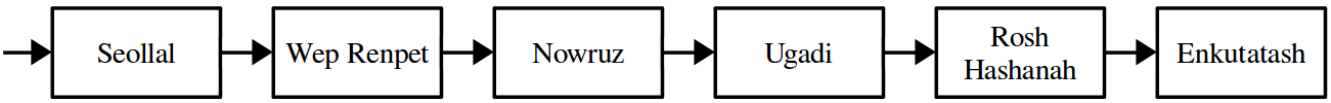


If we now do a look up for Nowruz again, since it's at the front of the list, we'll find it instantly, without having to scan anything else in the list.

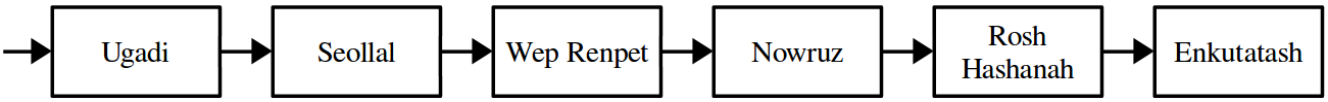
If we now insert the new element Wep Renpet, we'd insert it at the front of the list, as shown here:



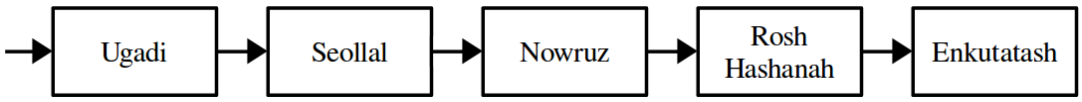
Now, if we do a lookup for Seollal, we'd reorder the list as follows:



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 values 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.

Below is the class definition for `MoveToFrontSet`. Do not write anything on this page that you want us to grade; there's space for your answer on the following pages.

```
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:
    /* Up to you to decide. */

};
```

```
/* Write the contents of the private section of MoveToFrontSet here. */
private:









/* Implement the constructor here. */
MoveToFrontSet::MoveToFrontSet() {









}
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work.

```
/* Implement the destructor here. */
MoveToFrontSet::~MoveToFrontSet() {









}

/* Implement contains here. */
bool MoveToFrontSet::contains(const string& str) {

```

```
}
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work.

```
/* Implement add here. */
void MoveToFrontSet::add(const string& str) {









}
/* Implement remove here. */
void MoveToFrontSet::remove(const string& str) {









}
```

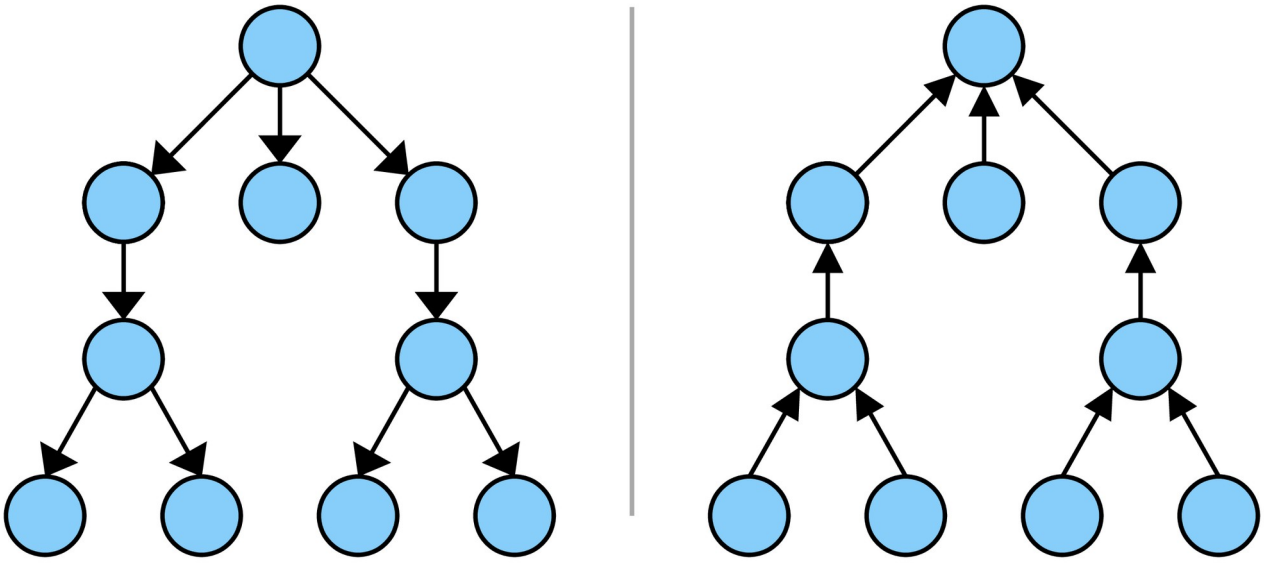
We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work.

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

Problem Five: Spaghetti Stacks

In a typical representation of a tree, we have each node store a pointer to each of its children. This makes it easy to walk down from the root to any leaf node. However, we did not have nodes store pointers back to their parents, making it hard to walk up the tree from any node to the root.

Another representation for a tree adopts the opposite approach – each node stores a pointer to its parent node, but no parent stores a pointer back to its child. This representation is sometimes called a ***spaghetti stack***. For example, here is the a tree represented both top-down and as a spaghetti stack:



Spaghetti stacks differ from normal tree representations in two major ways. First, in our normal top-down tree representation, we can store the tree as a single pointer to the root node. This is sufficient to reach any node in the tree, because every node ultimately descends from the root. In a spaghetti stack, we store one pointer for each leaf in the tree. This enables us to find any node in the tree by beginning at the appropriate leaf and walking upward to the root.

Second, because each node in a spaghetti stack stores just a single pointer to its parent, it is possible to think of a spaghetti stack as a bundle of linked lists whose nodes overlap with one another. Given any specific node in the tree, the set of nodes we can reach by following that node's parent pointer upward forms a linked list connecting that node to the root of the tree. You can try this out with the above tree.

Suppose that you are given the following `struct` representing a node in a tree using the standard top-down representation. Each node can have any number of children:

```
struct TreeNode {
    int value;
    Vector<TreeNode*> children;
};
```

You are also given the following `struct`, which represents a node in a spaghetti stack:

```
struct SpaghettiNode {
    int value;
    SpaghettiNode* parent;
};
```

The problem continues on the next page.

i. Write a function

```
void freeSpaghettiStack(const Set<SpaghettiNode*>& leaves);
```

that accepts as input a set of the leaf nodes in a spaghetti stack, then deallocates all nodes in that spaghetti stack. Be careful when implementing this function not to free the same node twice and to make sure that all nodes of the tree, not just the leaf nodes, are deallocated.

```
void freeSpaghettiStack(const Set<SpaghettiNode*>& leaves) {

}
```

We will grade whatever you write in this box.
Use the surrounding space for scratch work.

ii. Write a function

```
Set<SpaghettiNode*> spaghettify(TreeNode* root);
```

that accepts as input a pointer to the root of a normal, top-down representation of tree, then constructs a copy of that tree as a spaghetti stack. Your function should return a `Set` containing pointers to all of the leaf nodes of that tree, and should not make any changes to the input tree.

```
Set<SpaghettiNode*> spaghettify(TreeNode* root) {
```

We will grade whatever you write in this box.
Use the surrounding space for scratch work.