Final Exam Practice Problems

Here's a sampler of practice problems you can use to prepare for the final exam. This collection of problems is drawn from past CS106B final exams given over the years. It's not meant to be a representative sample of the *length* of the final exam - this collection of problems is far bigger than any exam I've ever given in any class - but rather to give you a sense of what sorts of questions you might expect to see.

Problem One: Data Structure Sleuthing

On Assignment 5, you gained experience writing code that met specific runtime bounds. On Assignments 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 below.

```
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

1. For each of these pieces of code, tell us its big-O runtime as a function of n. No justification is required.

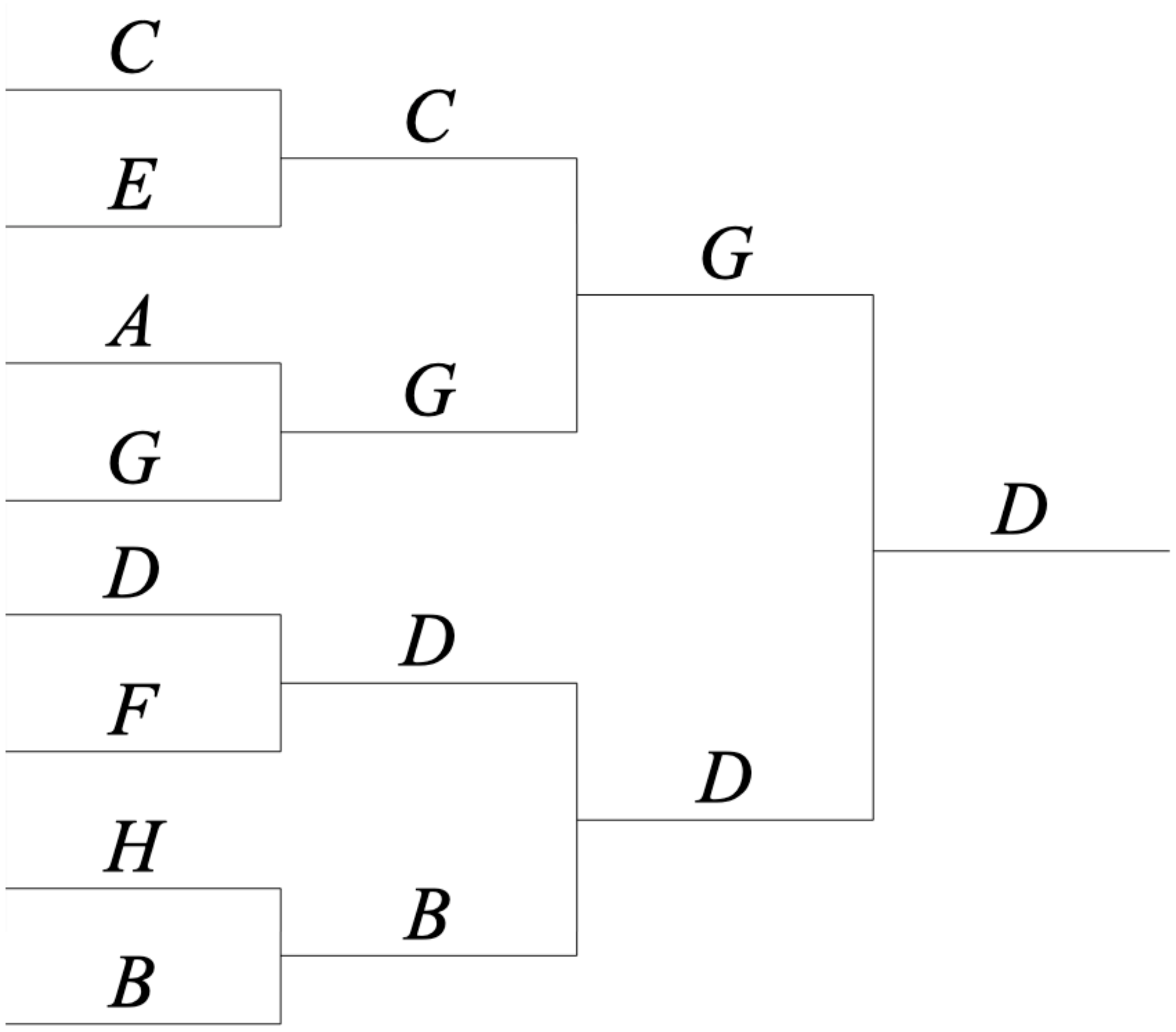
2. For each of these pieces of code, tell us whether that function could have given rise to the return values reported in the rightmost column of the table. No justification is required.
3. Which of the four functions could be the one we ran? If there's just a single option, tell us why that option is right and all the others are wrong. If there's multiple options, tell us why those options are possible and the others aren't. Justify your answer in at most fifty words.

Solution

Problem Two: Rigging a Tournament

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.



Part 1: Overall Winner Of

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.

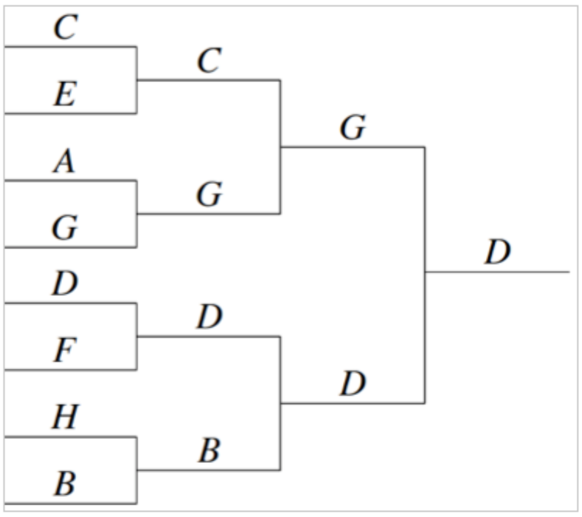
Some notes on this problem:

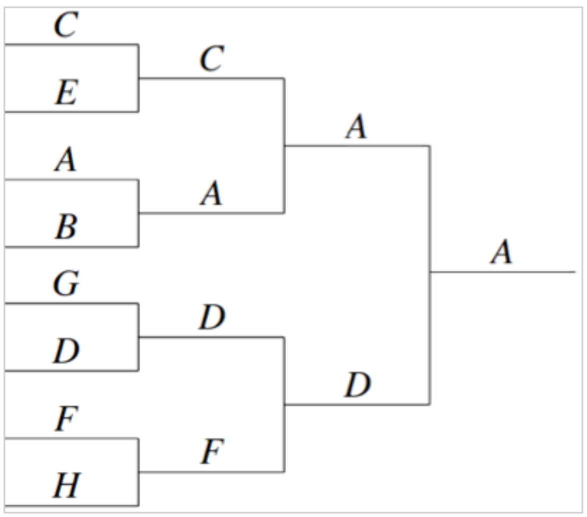
- You can assume the number of players is a perfect power of two (for example, 1, 2, 4, 8, 16, 32, 64, etc.), so there will never be a case where there's an "odd player out" who isn't assigned to play a game in some round. This also means you'll never get a list of zero players.
- You should not make 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. 😃

Solution

Part 2: Can Rig For

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 Set<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:

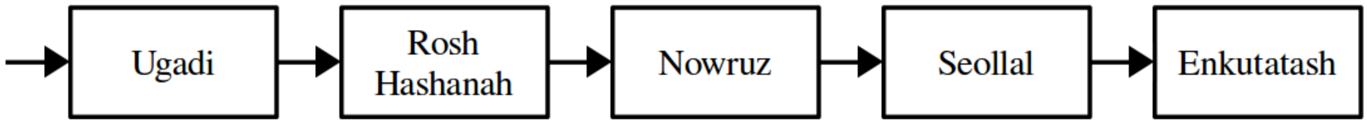
- You should make all the same assumptions about the input as in the first part of this problem: the number of players is always going to be a perfect power of two, that you should use the winnerOf function to determine who would win in a matchup, etc.
- Feel free to use the overallWinnerOf function 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.

Solution

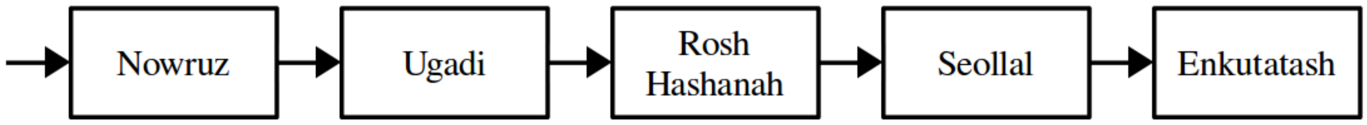
Problem Three: Self-Organizing Lists

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:

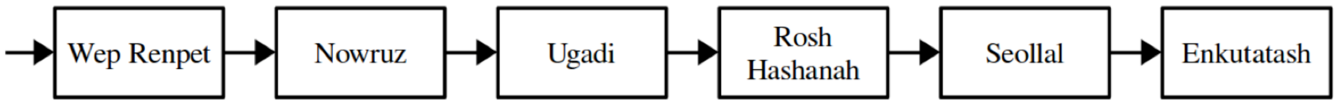


If we look up Nowruz, we'd move the cell containing Nowruz to the front of the list, as shown here:

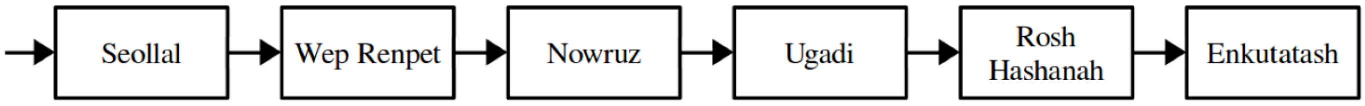


If we now do a look up for Nowruz again, since it's at the front of the list, we'll find it instantly, without having to scan anything else in the list.

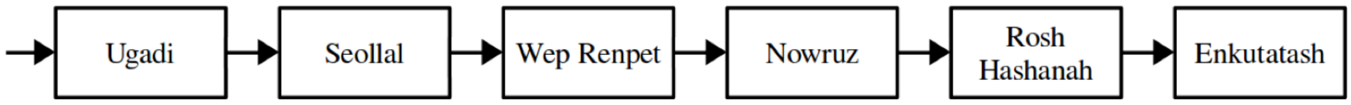
If we now insert the new element Wep Renpet, we'd insert it at the front of the list, as shown here:



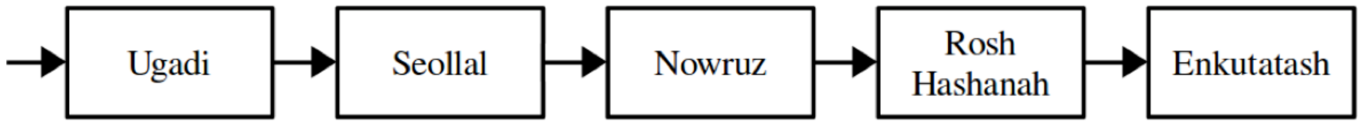
Now, if we do a lookup for Seollal, we'd reorder the list as follows:



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 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.

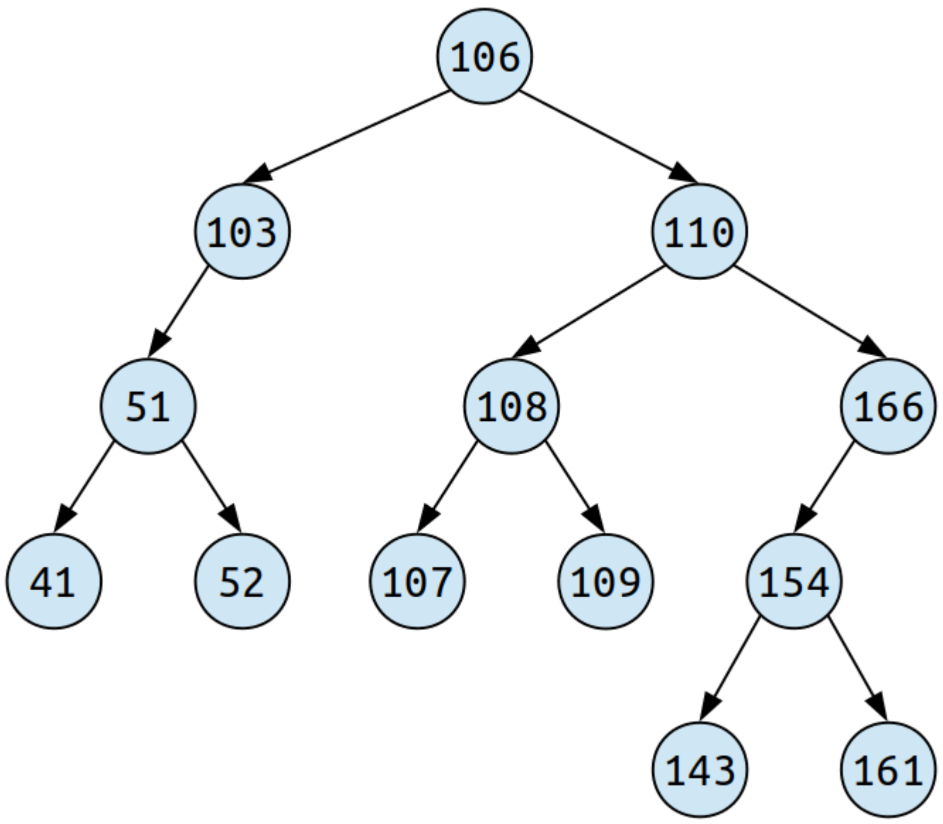
As a hint, you may find it useful to have your **`add`** and **`remove`** implementations call your **`contains`** member function and use the fact that it reorganizes the list for you.

```
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) {

}
```

Solution

Problem Four: Lower and Upper Bounds

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 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.).

Solution

Problem Five: Workout Playlists

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.

Solution

Problem Six: Binary Heaps

In Assignment 6, you implemented the **`HeapPQueue`** type using a binary heap. As a refresher from that assignment, the

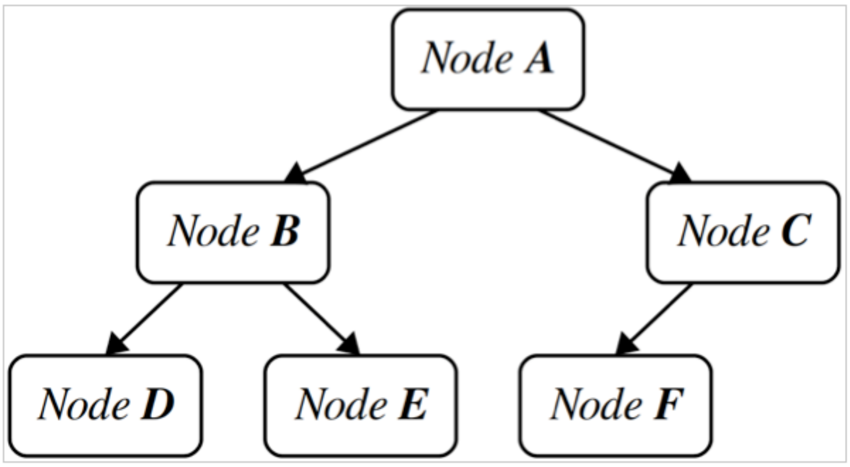
HeapPQueue stored objects of type **DataPoint**, where **DataPoint** is given as follows:

```
struct DataPoint {
    string name;
    int weight;
};
```

We've created a binary heap storing these six **DataPoint** objects:

```
{ "Emu", 1 }        { "Auk", 2 }        { "Cow", 3 }
{ "Yak", 4 }        { "Doe", 5 }        { "Ewe", 6 }
```

These elements were inserted into the binary heap in some order, but you don't know what that order is. Below is a picture of the shape of the binary heap holding these elements, but not what DataPoints are stored in each of the nodes. Each node has been assigned a letter so that we have a nice way to refer to it later; otherwise, the letters have no special meaning.



Regardless of the order in which we inserted the above six elements into the binary heap, we are guaranteed that **{ "Emu", 1 }** will end up in Node ***A*** due to the rules of how binary heaps work. Your task is to determine where other elements may end up.

Answer each of the following questions. No justification is needed. Incorrect answers will not receive points, but otherwise there is no penalty for an incorrect guess.

1. **(1 Point)** In the space below, list the letters of all nodes that may contain **{ "Auk", 2 }**.
2. **(1 Point)** In the space below, list the letters of all nodes that may contain **{ "Doe", 5 }**.

Solution

Problem Seven: Word Walks

Part 1: Overlapping Words

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;
}
```

Solution

Part 2: Word Walk

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:

```
bool canMakeWordWalkFrom(const Set<string>& words);
```

that takes in a ***nonempty*** set of words, then returns whether it's possible to form a word walk that uses all the words in that set exactly once.

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. 😃

Solution

Problem Eight: Linear Probing Sleuthing

In Assignment 7, you explored linear probing hash tables. In this problem, we will be exploring a linear probing table of integers. The hash code for each integer is formed by taking its last digit; for example, the hash code of 137 is 7, and the hash code of 106 is 6. Empty slots in the table are represented as blank spots, filled slots with the number they contain, and tombstones with the 墓 symbol.

18	37	墓			95	16	5	56	39
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

We filled this table in by starting with an empty table and using the standard linear probing algorithm from lecture without any modifications. The specific sequence of commands we issued to the table is shown below in the order in which those commands were performed. As you can see, some of the arguments to the commands have not been given.

Fill in the blanks such that executing the sequence of commands shown below builds the linear probing table shown above. As a hint, *you'll never need to insert something twice*, and *you'll never need to remove something that isn't already in the table*. No justification is needed. Incorrect answers will not receive points, but otherwise there is no penalty for an incorrect guess.

1. Insert 28.
2. Insert 32.
3. Insert __.
4. Remove __.
5. Insert 18.
6. Remove __.
7. Insert 95.
8. Insert __.
9. Insert __.
10. Insert __.
11. Insert __.

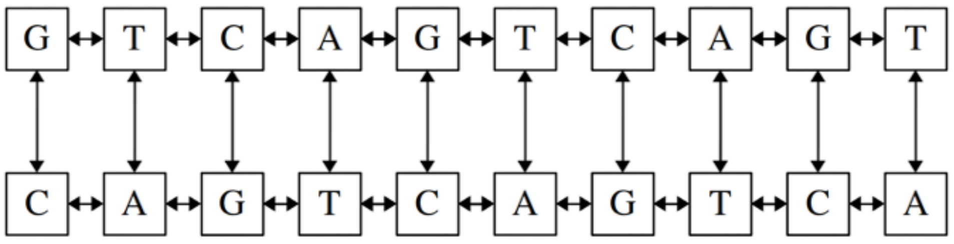
We recommend that you draw out an empty, ten-slot linear probing table like the one shown here for scratch work as you're working through this problem.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Solution

Problem Nine: Complementary Strands

In Assignment 7, you explored how to encode DNA as linked lists of nucleotides. In humans, DNA strands are always paired with a **complementary strand**. This is a new strand of DNA linked alongside the original strand, except with different nucleotides. Specifically, any time there's an A in the initial strand there's a T in the complementary strand, and any time there's a C in the initial strand there's a G in the complementary strand (and vice-versa). Here's an example of what this might look like:



We can represent a double-stranded DNA sequence as a modified doubly-linked list of nucleotides, where each nucleotide links to the nucleotides before and after it in the sequence, as well as to the nucleotide across from it:

```
struct Nucleotide {
    char value;          // 'A', 'C', 'G', or 'T'
    Nucleotide* next;    // To the right
    Nucleotide* prev;    // To the left
    Nucleotide* across; // Vertically, to the other strand.
};
```

For example, in the picture shown above, the nucleotide G in the upper-left corner would have its **prev** pointer set to **nullptr**, its next pointer set to the T nucleotide next to it, and its across pointer set to the C beneath it. The C nucleotide beneath it would have its next pointer set to the A to its right, its **prev** pointer set to **nullptr**, and its across pointer back to the G in the top strand.

Your task is to finish writing a function that works as follows. The function takes as input a single strand of DNA whose across pointers have not been initialized. The function then constructs the complementary strand, then links the two strands together by appropriately wiring the across pointers of the two strands.

Here is the implementation we've provided to you. As indicated by the comment, your task is to fill in the contents of the while loop. You must not add, remove, or edit the code outside the body of the while loop.

```
char complementOf(char base) {
    if (base == 'A') return 'T';
    if (base == 'C') return 'G';
    if (base == 'G') return 'C';
    if (base == 'T') return 'A';
    error("Unknown nucleotide.");
}
void addComplementaryStrand(Nucleotide* dna) {
    while (dna != nullptr) {
        /* TODO: Add your code here! */
    }
}
```

Some notes on this problem:

- The **across** pointers of each of the nucleotides in the input sequence have not been initialized when this function is called. You should not assume anything about where they point.
- For full credit, your solution must run in time *O(n)*, where *n* is the number of nucleotides in **dna**.
- The input pointer will always point to the first nucleotide in the strand, or will be **nullptr** if the input DNA strand is empty.
- You can assume all letters in the DNA strand are either A, C, G, or T.
- You ***may not*** use any container types (e.g. **Vector**, **Set**, etc.) in the course of solving this problem. This includes the **string** type.

Solution

Problem Ten: Huffman Sleuthing

Oops! We used the Huffman coding algorithm from Assignment 8 to determine codes for a bunch of characters, but we forgot the encoding of the character G.

Character	Encoding
A	1010
B	110

Character	Encoding
C	100
D	11101
E	1111
F	0
G	¯_(ツ)_/¯
H	11100

Your task is to help us figure out what the code for G is.

To do so, grab a sheet of scratch paper and draw as much of the Huffman coding tree as you can given the codes shown above. Once you've done so, look over that tree and see if anything about it looks unusual. That will let you determine what the code for G is.

Code for G: _______

(No justification is needed. Incorrect answers will not receive points, but otherwise there is no penalty for an incorrect guess.)

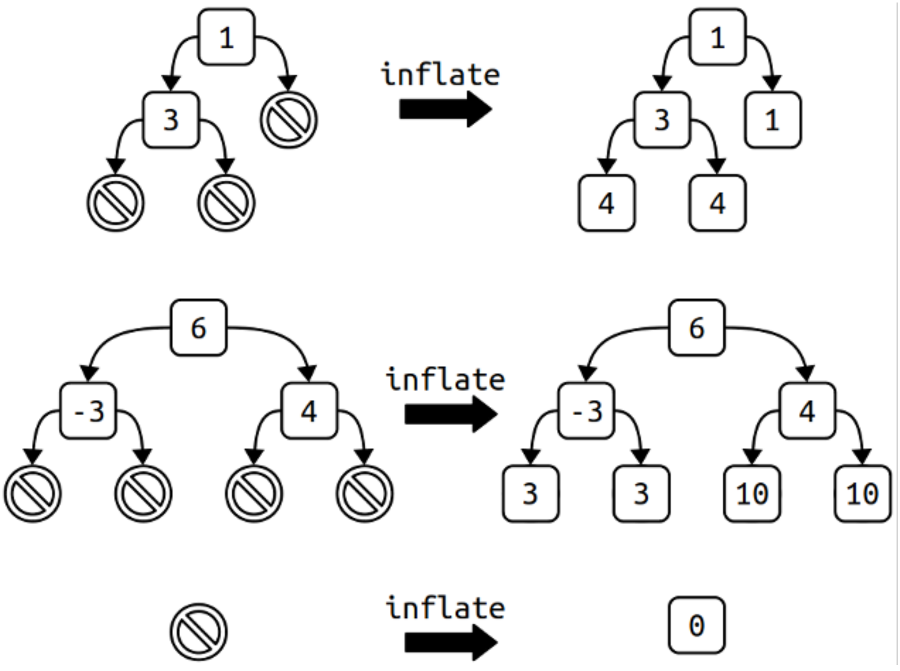
Solution

Problem Eleven: Inflating Trees

Consider the following type representing a node in a binary tree (though not necessarily a binary *search* tree):

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

Given a binary tree represented this way, we can ***inflate*** the binary tree by replacing all null pointers that appear in the original tree with new nodes, where each node's value is the sum of the integers on the path from the root of the tree to that node. (One edge case: if the tree is empty, we replace the root with a new node whose value is zero). For example, here's some sample trees and the result of inflating each of them. We've explicitly drawn in the null left and right pointers on the initial trees to make the transformation clearer. Although the null pointers aren't drawn in on the trees to the right, they are still present.



Write a function

```
void inflate(Node*& root);
```

that takes as input a pointer to the root of a tree (or to **nullptr** if the tree is empty), then inflates the tree. Some notes:

- Your solution does not have to be as efficient as possible, but you should avoid any unnecessary inefficiencies in your solution.
- You may not use any container types (e.g. **Vector**, **Set**, etc.) in the course of solving this problem.

Solution

Problem Twelve: Random Bag Lists

On Assignment 6 and Assignment 7, you gained experience writing classes that worked with dynamically-allocated memory. Assignment 8 also specifically asked you to work with doubly-linked lists. This question will ask you to implement another data structure with a doubly-linked list.

The very first class we designed in CS106B was the **RandomBag**. If you'll recall, the **RandomBag** supported two operations:

- **add**, which adds an element to the random bag, and
- **removeRandom**, which removes and returns a random element out of the **RandomBag**.

When we first implemented **RandomBag**, we layered it on top of the **Vector** type. We'd like you to now go and reimplement **RandomBag**, layered on top of a ***doubly-linked*** list.

For simplicity, we've only asked you to implement the constructor, **add**, and **removeRandom**. You don't need to implement a destructor or any other member functions. You're welcome to add new member functions and data members if you'd like. Your implementation of **removeRandom** should have an equal probability of returning any of the elements currently stored in the **RandomBag** and should not leak any memory. Additionally, **removeRandom** should call error, with whatever error message you think is most amusing, if the **RandomBag** is empty.

Feel free to use this function to generate random numbers:

```
int randomInteger(int low, int high); // Between low and high, inclusive
```

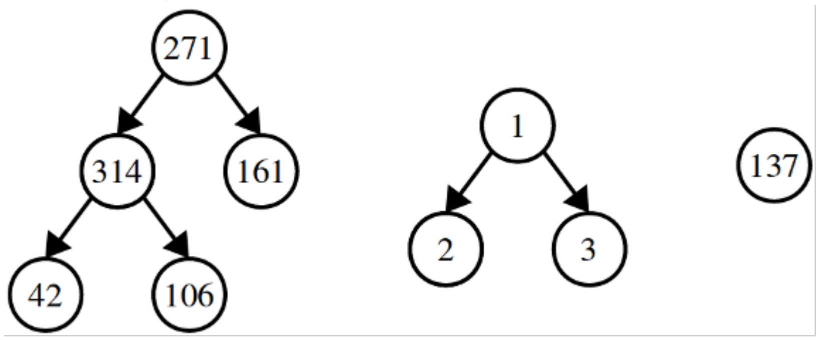
To receive full credit on this problem, you should not use any of the standard container types (**Vector**, **Map**, **Set**, etc.) in your solution, since the purpose of this problem is for you to implement a custom type without layering on top of an existing container.

For full credit, your implementation of add should run in time O(1) and your implementation of **removeRandom** should run in time O(n), where n is the number of elements in the **RandomBag**.

Solution

Problem Thirteen: Agglomerative Clustering

A ***full binary tree*** is a binary tree where each node either has two children or no children. Here are some sample full binary trees:



Note that full binary trees are not necessarily binary *search* trees.

Let's imagine that we have a type representing a node in a full binary tree, which is shown here:

```
struct Node {
    double value;  // The value stored in this node

    Node* left;    // Standard left and right child pointers
    Node* right;
};
```

Your first task in this problem is to write a function

```
Set<double> leavesOf(Node* root);
```

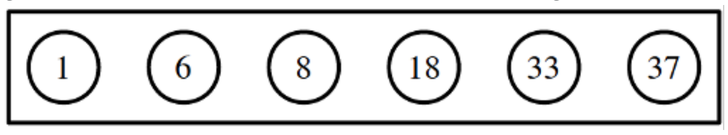
that takes as input a pointer to the root of a full binary tree, then returns a set of all the values stored in the leaves of that tree. For example, calling this function on the leftmost tree above would return a set containing {42, 106, 161}, calling this function on the tree in the middle would return {2, 3}, and calling this function on the tree on the right would return {137}.

Some notes:

- You can assume that the pointer to the root of the tree is not null.
- You should completely ignore the values stored at the intermediary nodes.

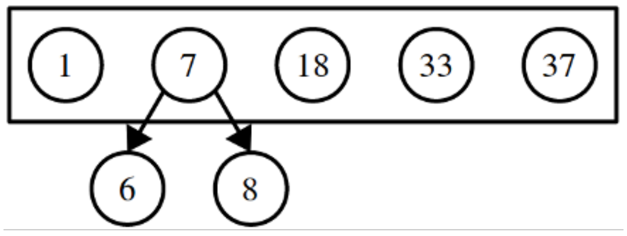
The second part of this problem explores an algorithm called ***agglomerative clustering*** that, given a collection of data points, groups similar data points together into clusters of similar values. The algorithm works by starting with a bunch of singleton nodes and assembling them into full binary trees. For the purposes of this problem, we'll cluster a group of doubles.

The first step in running agglomerative clustering is to create singleton trees for each of the data points. For example, given the numbers 1, 6, 8, 18, 33, and 37, we'd begin with the following trees:

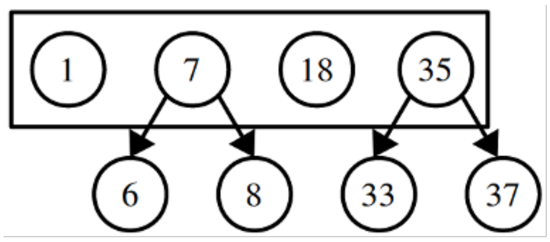


We now choose the two trees whose root nodes' values are closest to one another. Here, we pick the trees holding 6 and 8. We then merge those two trees into a single tree, and give the root of the new tree the average value of all its leaves.

Here, the leaves hold 6 and 8, so the new tree has root value 7:



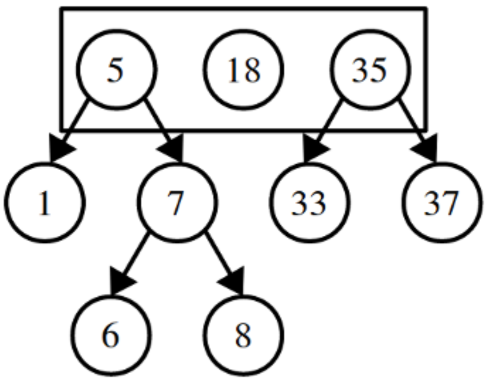
We repeat this process, again selecting the two trees whose root values are as close as possible. In this case, that would be 33 and 37, which get merged into a new tree:



Notice that the new root node has the value 35, the average value of its leaves.

On this next step, we'll find that the trees with the two closest roots are the ones with roots 1 and 7. We merge those trees into a new tree. As before, the root of this tree is then assigned the average value of all its leaves. The leaves have values

1, 6, and 8, so the new root gets the value 5. Here's the result:



Each step in this process reduces the number of trees by one. We'll stop this algorithm once we've reduced things down to a small number of trees; how many trees, exactly, we'll leave as a parameter for the user to specify.

If we stop here, we have three clusters: the cluster {1, 6, 8}, the cluster {18}, and the cluster {33, 37}, which you can see by looking at the leaves of the resulting trees.

Write a function

```
Set<Node*> cluster(const Set<double>& values, int numClusters);
```

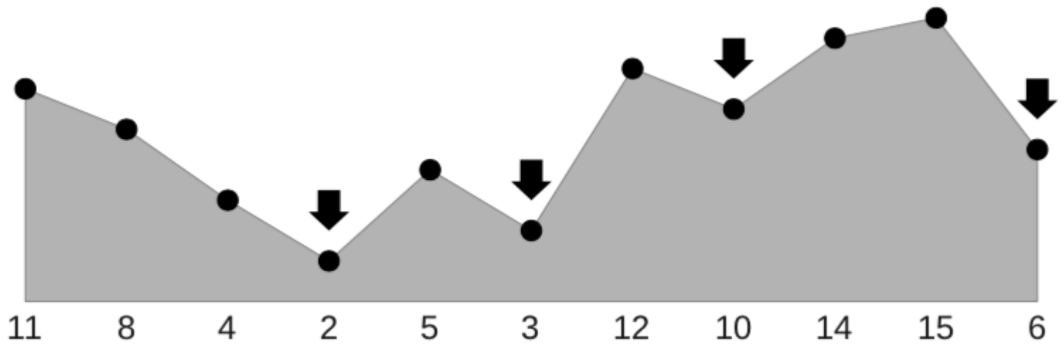
that takes as input a set of values and a desired number of clusters, then runs the agglomerative clustering algorithm described on the previous page to form the specified number of clusters. This function should then return a **`Set<Node*>`** containing pointers to the roots of the trees formed this way. Some notes:

- You can assume the number of clusters is less than or equal to the number of values and greater than or equal to one (the algorithm only works on values in those ranges).
- If there's a tie between which pair of tree roots is closest, break the tie however you'd like.
- You'll almost certainly want to use the **`leavesOf`** function you wrote in the first part of this function in the course of writing up your solution.

Solution

Problem Fourteen: Rainwater Collection

You are interested in setting up a collection point to funnel rainwater into a town's water supply. The town is next to a ridge, which for simplicity we will assume is represented as an array of the elevations of different points along the ridge. When rain falls on the ridge, it will roll downhill along the ridge. We'll call a point where water naturally accumulates (that is, a point lower than all neighboring points) a "good collection point." For example, here is a possible ridge with good collection points identified:



Write a recursive function

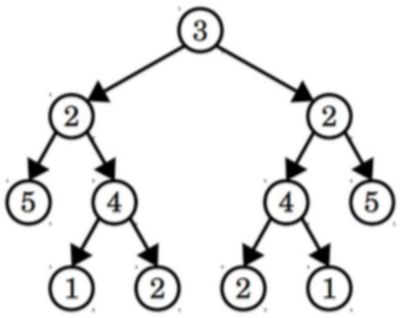
```
int goodCollectionPointFor(const Vector<int>& heights);
```

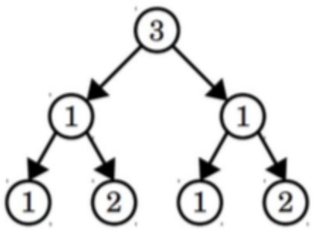
that returns the index of a good collection point. Your solution should run in time O(log *n*). As a hint, think about binary search. You can assume that all elements in the array are distinct.

Solution

Problem Fifteen: Palindromic Trees

A binary tree (not necessarily a binary *search tree*) is called a **palindromic tree** if it's its own mirror image. For example, the tree on the left is a palindromic tree, but the tree on the right is not:





Write a function that takes in a pointer to the root of a binary tree and returns whether it's a palindrome tree.

Solution

Problem Sixteen: The Great Tree List Recursion Problem

This excellent problem by Nick Parlante

A node a binary tree has the same fields as a node in a doubly-linked list: one field for some data and two pointers. The difference is what those pointers mean: in a binary tree, those fields point to a left and right subtree, and in a doubly-linked list they point to the next and previous elements of the list. Write a function that, given a pointer to the root of a binary *search* tree, flattens the tree into a doubly-linked list, with the values in sorted order, without allocating any new cells. You'll end up with a list where the pointer **`left`** functions like the **`prev`** pointer in a doubly-linked list and where the pointer **`right`** functions like the next pointer in a doubly-linked list.

Solution

Problem Seventeen: Changing Passwords

Looking to change your password? Rather than picking a password that's a common word or phrase with a bunch of random numbers thrown in, consider using a multi-word password formed by choosing some sequence of totally random words out of the dictionary. Choosing four totally random words, it turns out, tends to be a pretty good way to make a password. Here's a couple passwords you might make that way:

```
RantingCollegersDenoteClinching
VivificationPandectYawnedCarmine
DetachednessHowlinglySportscastsVapored
UnlearnedMockeriesTuskedChuckles
SharpshootingPreyParaffinsLibeler
```

We generated these particular passwords using the following piece of code:

```
string makeRandomPassword(const Vector<string>& wordList) {
   string result;
   for (int i = 0; i < 4; i++) {
      int wordIndex = randomInteger(0, wordList.size() - 1);
      result += wordList[wordIndex];
   }
   return result;
}
```

When we ran this code, we used a word list containing about 120,000 words. There are other word lists available that we could have picked. The Basic English dictionary, for example, only has about 5,000 words. A more elaborate dictionary called ENABLE has about 180,000 words. It's therefore not all that unreasonable for us to analyze the efficiency of the above code in terms of the number of words n in our word list.

i. Let n denote the number of words in wordList. What is the big-O time complexity of the above code as a function of n? You can assume that the words are short enough that the cost of concatenating four strings together is O(1) and that a random number can be generated in time O(1). Explain how you arrived at your answer. Your answer should be no more than 50 words long.

Solution

ii. Let's suppose that the above code takes 1ms to generate a password when given a word list of length 50,000. Based on your analysis from part (i), how long do you think the above code will take to generate a password when given a word list of length 100,000? Explain how you arrived at your answer. Your answer should be no more than 50 words long.

Solution

Now, let's think about how someone might try to break your password. If they know that you chose four totally random English words, they could try logging in using every possible combination of four English words. Here's some code that tries to do that:

```
string breakPassword(const Vector<string>& wordList) {
  for (int i = 0; i < wordList.size(); i++) {
    for (int j = 0; j < wordList.size(); j++) {
      for (int k = 0; k < wordList.size(); k++) {
        for (int l = 0; l < wordList.size(); l++) {
          string password = wordList[i] + wordList[j] + wordList[k] + wordList[l];
          if (passwordIsCorrect(password)) {
            return password;
          }
        }
      }
    }
  }
}
```

As before, let's assume that the words in our word list are short enough that the cost of concatenating four of them together is O(1). Let's also assume that calling the `passwordIsCorrect` function with a given password takes time O(1).

iii. What is the worst-case big-O time complexity of the above piece of code as a function of n, the number of words in the word list? Explain how you arrived at your answer. Your answer should be no more than 50 words long.

Solution

iv. Imagine that in the worst case it takes 1,000 years to break a four-word password when given a word list of length 50,000. Based on your analysis from part (iii), how long will it take, in the worst case, to break a password when given a word list of length 100,000? Explain how you arrived at your answer. Your answer should be no more than 50 words long.

Solution

Problem Eighteen: 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.

Solution

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.

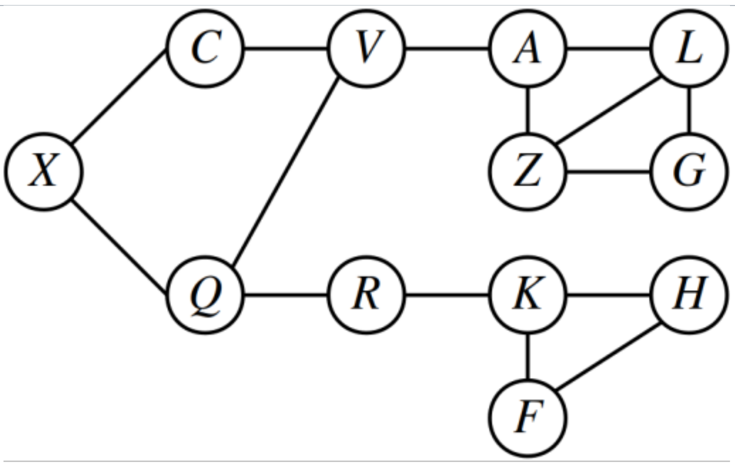
Solution

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.

Solution

Problem Nineteen: Graph Sleuthing

Consider the picture of the graph shown below:



Suppose we run a breadth-first search on this graph, starting at node X. Whenever we dequeue a node from the BFS queue, we print out that node's label.

i. Explain why it is ***impossible*** to get back the following sequence of nodes when running BFS on the above graph, starting at node X.

X, Q, C, R, V, K, Z, A, F, H, L, G

Please limit your answer to fifty words or fewer. As a hint, in what order does BFS visit nodes when exploring a graph?

Solution

Now, a little more backstory. We did get the above sequence by running BFS on some graph. It just wasn't the graph shown above. We'll call the graph that we ran BFS on the ***mystery graph***.

The mystery graph is identical to the above graph except that one additional edge has been added in. That edge, which is an undirected edge, links two nodes that do not appear to be linked in the above diagram. We'll call that edge the ***mystery edge***.

ii. Based on the BFS traversal, the picture given above, and the fact that only one extra edge has been added in, you have enough information to narrow down the mystery edge to one of two possibilities. What are those two possibilities? No justification is required.

Solution

One last piece of information about the mystery graph. We additionally ran a DFS on the mystery graph, also starting from the node X. That DFS was implemented recursively, and whenever we made a recursive call on a node for the very first time, we printed out the label on that node. Here's the sequence of nodes we got back:

X, Q, R, K, F, H, Z, G, L, A, V, C

Just to make sure that we're clear about this, this is a DFS from the mystery graph, the one that has the extra edge added in relative to the above picture. The above sequence couldn't be produced in the graph shown above.

iii. Explain why it is impossible to get the above sequence of nodes as the result of DFS in the graph drawn above. Please limit your answer to fifty words or fewer.

Solution

iv. You now have enough information to identify the mystery edge. Which edge is it, and why? Please limit your answer to fifty words or fewer.

Solution