Elyse Cornwall
Amrita Kaur

August 18, 2023

CS106B Final Exam **SOLUTIONS**

You have 3 hours (180 minutes) to complete this exam. The exam is 180 points, meant to correspond to one point per minute of the exam. You may want to use the point allocation for each problem as an indicator for pacing yourself on the exam.

This is a closed-book, closed-device exam. The only materials you may consult during this exam are the provided reference sheet and one 8.5” by 11” inch page (front and back) of notes.

The majority of the points on this exam will be awarded for showing a mastery of the concepts taught in CS106B. There is no guarantee that you will receive points for using concepts not covered in CS106B. You do not need to `#include` any libraries, comment your code, or worry about minor syntactical issues. You may use helper functions if they help you solve the problem, but decomposition for the sake of style is not required.

We will only grade one answer per problem. We cannot grade multiple solution attempts for a single problem, so please clearly indicate your final solution if there are any notes or previous attempts on the page.

If you need extra space to work on a problem, you may use scratch paper. However, we will not grade or accept any work done on scratch paper. All final answers must be written on the exam booklet in the designated sections.

Problem	**Points**
1. Recursive Backtracking	40
2. Classes and Dynamic Memory	45
3. Linked Lists	20
4. Trees	40
5. Hashing	20
6. Graphs	15
Total	180

I acknowledge and accept the letter and spirit of the honor code.

Signature: ______________________________

Name (print): ______________________________

SUNetID (i.e. your @stanford.edu email): ______________________________

1 Recursive Backtracking [40 points]

Inspired by Stanford's nickname, "The Farm," Elyse and Amrita decide it is time to leave their careers as computer science educators and start their new lives as farmers. They create a plan of what crops they will be growing during different months of the year, known as a "growing plan." They also decide that every year, they will take one month off for vacation, during which they won't grow any crops.

Implement the function:

```
Set<Map<int, Set<string>>> reassignCrops(Map<int, Set<string>>& growingPlan,
                                         int vacationMonth)
```

This function takes in two parameters:

- A reference to a Map where each key is a month of the year (represented as an integer from 1 to 12) and each value is the group of crops that will be grown that month (represented as a Set of strings).
- An integer representing the month that Elyse and Amrita will be on vacation.

Elyse and Amrita would like to redistribute the crops that would have been grown during the vacation month to other months of the year. This function will return a Set with all of the new valid "growing plans" where the crops from the vacation month have been reassigned to other months.

Here is an example of how this function would work. Let's say the Map `growingPlan` stores the following information:

```
{
     1: {"apples", "corn"},
     3: {"bananas", "corn"},
     9: {"strawberries"}
}
```

The call to the function `reassignCrops(growingPlan, 9)` would return the following Set:

```
{
    {1: {"apples", "corn", "strawberries"},
     3: {"bananas", "corn"},
     9: {}},

    {1: {"apples", "corn"},
     3: {"bananas", "corn", "strawberries"},
     9: {}}
}
```

Some additional stipulations for this problem:

- The Map `growingPlan` is not guaranteed to have all 12 months of the year as keys. Your function should not redistribute crops to months that do not appear in the Map.
- You may assume that `vacationMonth` appears in the `growingPlan` Map, and that its corresponding Set has at least one crop in it.
- You may not reassign a crop to a month during which that crop is already being grown. Assume there is always at least one other month to which you may reassign a crop.
- The order in which crops appear in each Set does not matter.
- To receive full credit, you must solve this problem using recursive backtracking.
- To receive full credit, any ADTs passed into helper functions must be passed by reference.

Here is another example of how `reassignCrops` would work, using the same Map.

```
{
     1: {"apples", "corn"},
     3: {"bananas", "corn"},
     9: {"strawberries"}
}
```

The call to the function `reassignCrops(growingPlan, 3)` would return the following Set:

```
{
    {1: {"apples", "corn", "bananas"},
     3: {},
     9: {"strawberries", "corn"}},

    {1: {"apples", "corn"},
     3: {},
     9: {"strawberries", "bananas", "corn"}}
}
```

Please write your solution to Problem 1 on the following page. You may use this page for scratch work, which will not be graded.

Please write your solution to Problem 1 (including any helper functions) on this page.

```
Set<Map<int, Set<string>>> reassignCrops(Map<int, Set<string>>& growingPlan,
                                          int vacationMonth) {
    Set<Map<int, Set<string>>> result;
    reassignCropsHelper(growingPlan, result, vacationMonth);
    return result;
}

void reassignCropsHelper(Map<int, Set<string>>& growingPlan,
                         Set<Map<int, Set<string>>>& result,
                         int vacationMonth) {

    // base case - all crops have been reassigned
    if (growingPlan[vacationMonth].isEmpty()) {
        result.add(growingPlan);
        return;
    }
    // choose a crop
    string crop = growingPlan[vacationMonth].first();
    growingPlan[vacationMonth].remove(crop);

    // explore reassigning to different months
    for (int month: growingPlan) {
        if (month != vacationMonth && !growingPlan[month].contains(crop)) {
            growingPlan[month].add(crop);
            reassignCropsHelper(growingPlan, result, vacationMonth);
            // unchoose this month
            growingPlan[month].remove(crop);
        }
    }

    // unchoose this crop
    growingPlan[vacationMonth].add(crop);
}

// -- alternate solution without helper function --
Set<Map<int, Set<string>>> reassignCrops(Map<int, Set<string>>& growingPlan,
                                          int vacationMonth) {
    Set<Map<int, Set<string>>> allPlans;
    if (growingPlan[vacationMonth].isEmpty()) {
        allPlans.add(growingPlan);
        return allPlans;
    }

    Vector<int> keys = growingPlan.keys();
    for (int i = 0; i < growingPlan[vacationMonth].size(); i++) {
        string crop = growingPlan[vacationMonth].first();
        growingPlan[vacationMonth].remove(crop);

        for (int key : keys) {
            if (key != vacationMonth && !growingPlan[key].contains(crop)) {
                growingPlan[key].add(crop);
                Set<Map<int, Set<string>>> somePlans = reassignCrops(growingPlan,
                                                                     vacationMonth);
                allPlans = allPlans.unionWith(somePlans);
                growingPlan[key].remove(crop);
            }
        }
        growingPlan[vacationMonth].add(crop);
    }

    return allPlans;
}
```

2 Classes and Dynamic Memory [45 points]

Your task is to write a Deck class. This class will simulate a standard 52-card deck of playing cards. You can assume you have access to the following Card struct:

```
struct Card {
    string suit;
    int faceValue;
};
```

Each Card has a suit (“Club”, “Diamond”, “Heart”, or “Spade”) and a face value (1 through 13). This shouldn’t matter for your implementation, but you can consider 1 to be an Ace (lowest card), 2-10 to be numbered cards, 11 to be a Jack, 12 to be a Queen, and 13 to be a King (highest card). A full deck consists of every possible combination of suit and face value - this is 52 cards total. You will implement the Deck class to allow a client to manipulate a standard deck of cards.

Here’s the header file for Deck, which must store all the cards in the Deck as a dynamically-allocated array of Card structs. Each method (also known as a member function) is described in more detail on the following page. You may assume you have a working implementation of shuffleDeck and swapCards, which you can use when implementing the other methods.

```
// deck.h

class Deck {

public:
    Deck();

    ~Deck();

    bool isEmpty();

    void removeCard(std::string suit, int faceValue);

    void sortDeck();

    // PROVIDED METHOD: shuffles cards into random order
    void shuffleDeck();

private:
    // PROVIDED METHOD: swaps cards at index1 and index2 in the Deck
    void swapCards(int index1, int index2);
};
```

Your task is to:

1. Complete the `private` section of the header file.
2. Write the full class implementation (i.e. what would go in the `deck.cpp` file).

To receive full credit, you must meet these implementation requirements:

1. You may not change the public interface of the `Deck` class, but you may add whatever private member variables or methods you'd like to the `private` section.
2. You may not allocate any new memory on the heap except in the constructor (you should only use `new` in the constructor).
3. You may not use any ADTs - no Sets, Maps, Vectors, Queues, etc. However, as mentioned previously, you will store the cards in an array of `Card` structs.
4. You must meet the Big-O runtime requirements for each method as described below. You can assume that N refers to the number of cards in the `Deck`.

Methods to implement:

- `Constructor` - Initializes a full deck of 52 cards along with any other member variables, then shuffles the deck. Note that the `shuffleDeck` function is provided, so you can just call `shuffleDeck()` at the end of the constructor. *No runtime requirement.*
- `Destructor` - Frees all memory associated with the `Deck`. *No runtime requirement.*
- `bool isEmpty()` - Returns `true` if there are zero cards in the deck, `false` otherwise. *O(1).*
- `void removeCard(string suit, int faceValue)` - Removes the card specified by the parameters `suit` and `faceValue`, or throws an error if this card is not in the deck. You can use the `error()` function to throw an error. $O(N)$.
- `void sortDeck()` - Sorts the deck with runtime $O(N^2)$ using the Selection Sort algorithm:

  ```
  for each index i:
      find the smallest card starting at index i
      swap the smallest card with the card currently at index i
  ```

  You should sort the deck so that the lowest cards are at the front of the array, and the highest cards are at the end. You will need to sort the cards by both suit and face value. The face values from lowest to highest are in ascending order, 1 to 13. The suits from lowest to highest are specified in the global constant `suits = {"Club", "Diamond", "Heart", "Spade"}`. Take note that these suits are in ascending alphabetical order! When comparing cards for sorting, you should give precedence to the card's suit over the card's face value. For example:

  - `{"Heart", 1}` goes before `{"Heart", 5}`, within the same suit we sort by face value
  - `{"Club", 10}` goes before `{"Heart", 5}`, suit takes precedence over face value

Please write your solution to Problem 2 on this page and the following pages.

```
// deck.h     Please note: there are many ways to implement this class
              depending on what data you chose to store and how you
class Deck {  removed cards from your deck. We just need the expected
              behavior for all methods as described on the previous page.
public:       This is one of several possible solutions.
    // ... see page 5
private:
    // PROVIDED METHOD: swaps cards at index1 and index2 in the Deck
    void swapCards(int index1, int index2);

    // write any other private variables and/or methods here

    Card* _cards;       // array of cards

    int _numCards;      // store number of cards currently in deck

    int findSmallest(int start);    // helper function for sortDeck


};
```

```
// deck.cpp

#include "deck.h"
#include "vector.h"
using namespace std;

const Vector<string> suits = {"Club", "Diamond", "Heart", "Spade"};

Deck::Deck() {

    _cards = new Card[52];
    int cardInd = 0;
    for (string suit: suits) {
        for (int i = 1; i <= 13; i++) {
            _cards[cardInd].suit = suit;
            _cards[cardInd].faceValue = i;
            cardInd++;
        }
    }
    _numCards = 52;
    shuffleDeck();



}
```

Please continue writing your solution to Problem 2 on this page.

```
// deck.cpp (continued)

Deck::˜Deck() {
    delete[] _cards;






}

bool Deck::isEmpty() {
    return _numCards == 0;

    // can also use an if/else
    statement



}

void Deck::removeCard(string suit, int faceValue) {
     // as mentioned before, there are many ways to do this as long as
     // it yields the desired behavior from the client's perspective

     bool found = false;
     int cardIndex = 0;

     while (cardIndex < _numCards && !found) {
         // if we find the card we're looking for
         if (_cards[cardIndex].faceValue == faceValue
                 && _cards[cardIndex].suit == suit) {
             found = true;
             // shift over the remaining cards
             for (int i = cardIndex + 1; i < _numCards; i++) {
                 _cards[i-1] = _cards[i];
             }
             _numCards--;
         }
         cardIndex++;
     }

     if (!found) {
         error("That card is not in the deck");
     }


}
```

Please continue writing your solution to Problem 2 on this page.

```
// deck.cpp (continued)

void Deck::sortDeck() {
     for (int i = 0; i < _numCards; i++) {
         int indexOfSmallest = findSmallest(i);
         swap(i, indexOfSmallest);
     }
 }

}

// write any other methods you might want here
 int Deck::findSmallest(int start) {
     // set initial values
     int smallestIndex = 0;
     string smallestSuit = "X";
     int smallestFaceValue = 14;

     // loop over all values starting at index start
     for (int i = start; i < _numCards; i++) {

         // if the suit is smaller OR the suit is the same and
         // the face value is smaller, update current smallest
         if (_cards[i].suit < smallestSuit
                 || (_cards[i].suit == smallestSuit
                 && _cards[i].faceValue < smallestFaceValue)) {

             smallestSuit = _cards[i].suit;
             smallestFaceValue = _cards[i].faceValue;
             smallestIndex = i;
         }
     }

     return smallestIndex;
 }
```

This page is intentionally blank. You may use this page for scratch paper, which will not be graded.

3 Linked Lists [20 points]

You can assume you have access to the following `ListNode` struct:

```
struct ListNode {
    int data;
    ListNode* next;
};
```

Given some non-empty linked list of `ListNode`s, our goal is to reduce this list to length one (a single node) by removing every third node. Importantly, this list is **circular**; the last node of the list points back to the head of the list, rather than `nullptr`.

Implement the function:

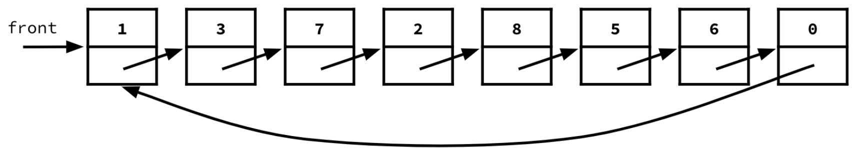
```
void reduce(ListNode*& front)
```

that takes in `front`, a pointer to the head of a non-empty **circular** linked list. Note that `front` is passed by reference. Your function should not return anything. Instead, it should modify the list so that `front` points to the single remaining node in the list after it has been reduced.

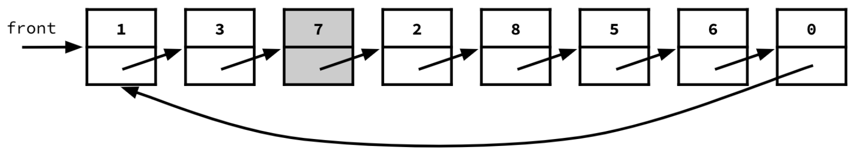
Some notes on this problem:

- You may assume that no two `ListNode`s have the same data (no duplicates).
- You may assume that our input list `front` is at least length 1.
- To receive full credit, you must free the memory associated with any nodes you remove.
- To receive full credit, you must write this function iteratively.

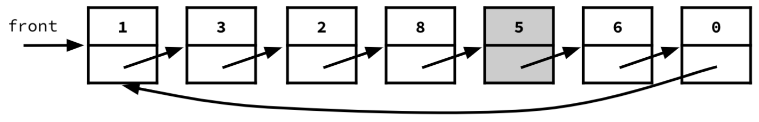
For example, let's say we have the following circular linked list:



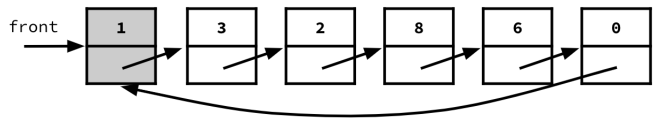
Calling `reduce` on this list will first delete the third node from the front, which is the 7:



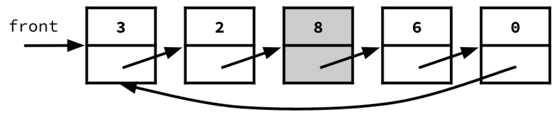
Then, it will delete what originally was the sixth node from the front, which is the 5:



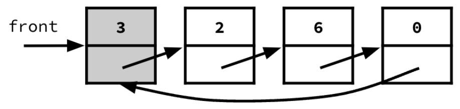
Then, it will wrap around to delete what was originally the ninth node from the front, the 1:

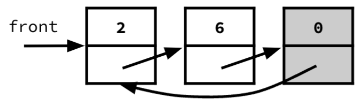


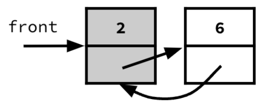
When the node that `front` points to is removed, `front` is updated to point to the next node.

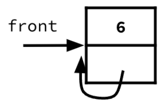


We continue traversing our circular list, deleting every third node:









Now that we're down to one node, our `reduce` function is done. Note that the list is still circular.

Please note: there are many ways to implement this
function – here are just two common solutions we saw!

Please write your solution to Problem 3 (including any helper functions) on this page.

```
void reduce(ListNode*& front) {
     ListNode* cur = front;

     // if the node points back to itself, we're down to a single node
     while (cur->data != cur->next->data) {

         // get third node
         ListNode* toDelete = cur->next->next;
         ListNode* after = toDelete->next;

         // rewire to skip over the node we want to delete
         cur->next->next = after;

         // if we're deleting the front node
         if (front->data == toDelete->data) {
             front = after;
         }
         cur = after;
         delete toDelete;
     }
 }
```

```
 // -- alternate solution --

 void reduce(ListNode*& front) {
     ListNode* cur = front;

     // if the node points back to itself, we're down to a single node
     while (cur->data != cur->next->data) {

         // get third node
         ListNode* toDelete = cur->next->next;
         ListNode* after = toDelete->next;

         // rewire to skip over the node we want to delete
         cur->next->next = after;

         cur = after;
         delete toDelete;
     }

     // can also reassign front at the end
     front = cur;


}
```

4 Trees [40 points]

For parts A and B, you can assume you have access to the following `TreeNode` struct:

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

Part A

In lecture, we learned how to remove nodes from binary search trees. Removing a leaf node was easy, but removing internal nodes with two children proved to be much more difficult. When removing such a node, we needed to replace that node with either the inorder predecessor or inorder successor. Implement the function:

```
    TreeNode* findInorderSuccessor(TreeNode* nodeToRemove)
```

that takes in `nodeToRemove`, a `TreeNode`. Your function should find and return a pointer to the inorder successor of `nodeToRemove`. As a reminder, the inorder successor of a node is the smallest node in its right subtree.

Some notes on this problem:

- If a node does not have an inorder successor, then your function should return `nullptr`.
- To receive full credit, you must write this function recursively.
- To receive full credit, your function must have a runtime of *O(log N)*, where *N* is the number of nodes in the right subtree of `nodeToRemove`.
- Your function should not alter the tree in any way. Remember, you should not be removing the inorder successor from the tree, only returning a pointer to it.
- We recommend drawing out a few sample binary search trees to get a feel for how to find the inorder successor. Feel free to use the rest of this page for scratch work, which will not be graded.

Please write your solution to Problem 4, Part A (including any helper functions) on this page.

```
TreeNode* findInorderSuccessor(TreeNode* nodeToRemove) {
    if (nodeToRemove == nullptr) { // we didn’t deduct for missing this
        return nullptr;
    }

    return findSmallestNode(nodeToRemove->right);
}


 TreeNode* findSmallestNode(TreeNode* tree) {

     // base case: no more smaller nodes
     if (tree == nullptr || tree->left == nullptr) {
         return tree;
     }

     // recursive case: keep searching left
     return findSmallestNode(tree->left);
 }
```

```
}
```

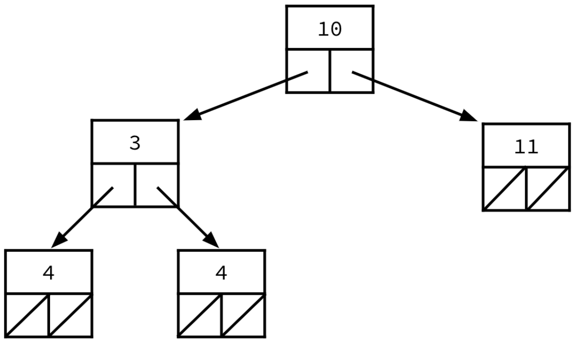
Part B

In lecture, you learned about balanced binary trees, in which the height of the left and right subtrees of each node differ by at most one. You feel inspired, and decide to create your own type of binary tree, called a “sum balanced” tree. A node within a binary tree is “sum balanced” if the sums of the values in the left and right subtrees of that node are equal. Implement the function:

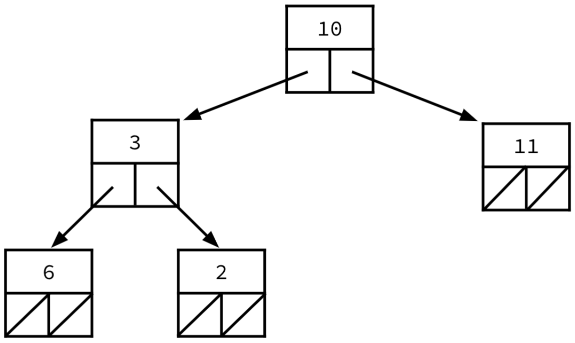
```
bool isSumBalanced(TreeNode* tree)
```

which returns `true` if all of the nodes in a tree are “sum balanced,” and `false` otherwise. This function accepts a pointer to a `TreeNode` as its only parameter.

For example, let’s say `tree` is a pointer to the root node of the following tree:



The call to the function `isSumBalanced(tree)` would return `true`, since each node in this tree is “sum balanced.” On the other hand, let’s say `tree` is a pointer to the root node of this tree:



The call to the function `isSumBalanced(tree)` would return `false`, since one of the nodes in this tree (the 3 node) is not “sum balanced.” Confirm this for yourself in the diagram above.

Some notes on this problem:

- If a subtree of a node is empty, you may assume the sum of that subtree is zero.
- An empty tree is “sum balanced.”
- To receive full credit, you must write this function recursively.
- To receive full credit, your function should not alter the tree in any way.

Please write your solution to Problem 4, Part B (including any helper functions) on this page.

```
bool isSumBalanced(TreeNode* tree) {
    if (tree == nullptr) {
        return true;
    }

    int leftSum = sumTree(tree->left);
    int rightSum = sumTree(tree->right);
    if (leftSum == rightSum) {
        return isSumBalanced(tree->left) && isSumBalanced(tree->right);
    } else {
        return false;
    }
}


int sumTree(TreeNode* tree) {
    if (tree == nullptr) {
        return 0;
    }

    return sumTree(tree->left) + sumTree(tree->right) + tree->data;
}
```

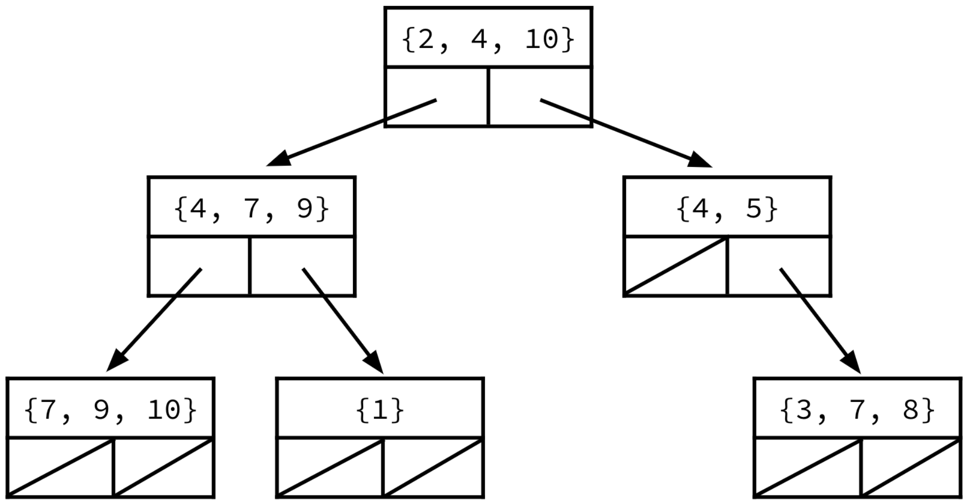
```
}
```

Part C

The CS106B SLs have combined their efforts to write a program that, given a tree where each node's data field is a Vector of sorted integers, is able to count how many times a certain number of interest appears in the entire tree. You can assume that an integer can only appear in a given node's Vector once (no duplicates within a Vector). The SLs have given you the high-level pseudocode for this function. You may assume that this function works correctly.

```
int countAppearances(TreeNode* root, int numOfInterest) {
    if root is null, return 0

    // in-order traversal
    counter = 0

    counter += countAppearances of numOfInterest in left subtree

    do binary search on this node's Vector to search for numOfInterest
    if numOfInterest is in this node's Vector:
        counter++

    counter += countAppearances of numOfInterest in right subtree
    return counter
}
```

Let's see this function in action! Let's say we have the following tree, where `root` is a pointer to the root node:



The call to the function `countAppearances(root, 4)` will return 3. The call to the function `countAppearances(root, 7)` will also return 3.

Let’s say our tree has N `TreeNodes` in it, and every Vector in a `TreeNode` has a maximum of M elements. **What is the Big-O runtime of `countAppearances`?**

```
N log(M)
```

Part D

Why might we be okay with traversing a tree with N nodes recursively, while we would avoid using recursion to traverse a linked list with N nodes? Answer in 1-2 sentences.

```
In a tree, the height from the root to any node is often much less
than N, so the depth of recursive calls (and thus stack frames) ends
up being much less than N. On the other hand, when we traverse a
linked list with N nodes, we’ll produce N stack frames, which causes
stack overflow for large values of N.
```

5 Hashing [20 points]

Simulate the behavior of a Hash Set of strings, implemented using a Chaining Hash Table.

Some notes on this problem:

- The Hash Set has an initial capacity of 5, which is stored in the variable `tableCapacity`.
- The Hash Set uses linked list chaining to resolve collisions, and new values are prepended to the head of each linked list.
- Rehashing occurs if the load factor exceeds 1.0 *after* a new element is added to the set. Rehashing doubles the table's capacity and rehashes all elements.
- The hash function we'll use to assign strings to buckets is:

  ```
  int hash(string s) {
      return s.length() % tableCapacity;
  }
  ```

Part A

Your task is to:

1. Draw an array diagram to show the final state of the Hash Set after the following operations are performed. You may leave unused buckets blank, assuming they will contain `nullptr`. All values must be in the proper buckets and in the proper order to receive full credit.
2. Write the final number of elements, table capacity, and load factor of the Hash Set.

```
HashSet<string> set;         // initialize empty set
set.add("you");              // length 3
set.add("are");              // length 3
set.add("doing");            // length 5
set.add("amazing");          // length 7
set.add("keep");             // length 4
set.add("it");               // length 2
set.add("up");               // length 2
set.remove("amazing");       // length 7
set.add("exquisitely");      // length 11
set.add("!");                // length 1
```

You should format your table as follows. The number at the beginning of the line is the bucket, and the elements in each bucket are linked with arrows. The head of each list is the leftmost element.

```
0: "here" -> "is"
1:
2: "an" -> "example"
3: "set"
...
```

Please write your solution to Problem 5, Part A below.

```
  // there are two valid solutions based on how you rehashed elements
  0:
  1: ! -> exquisitely
  2: up -> it
  3: you -> are
  4: keep
  5: doing
  6:
  7:
  8:
  9:

  // alternate (swap "are" and "you" depending on how you rehashed)
  0:
  1: ! -> exquisitely
  2: up -> it
  3: are -> you
  4: keep
  5: doing
  6:
  7:
  8:
  9:

Final number of elements: 8
Final table capacity: 10
Final load factor: 8/10 = 4/5 = 0.8
```

Part B

Let's say we're using this Hash Set (same hash function, same load factor as in Part A) to store all of the words in the English dictionary. This is roughly 200,000 words. Based on our definition of a "good" hash function from class, why might we consider `hash` as shown on the previous page to be a bad hash function, at least for this use case?

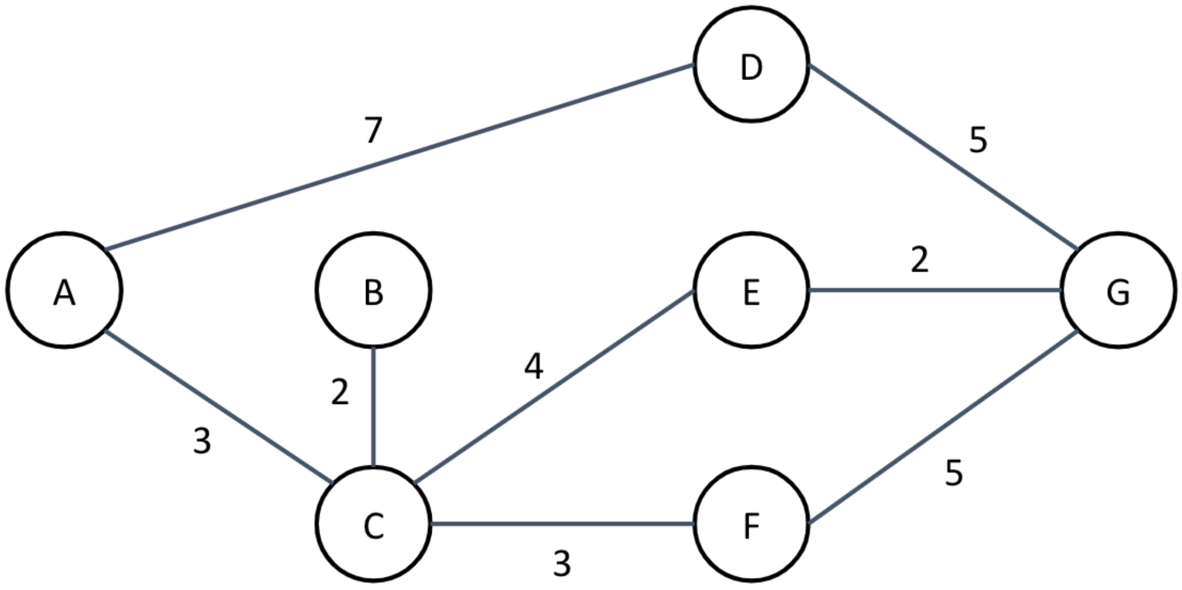
Answer in 1-2 sentences.

```
If we're maintaining a load factor of 1.0, then we can expect to have
roughly 200,000 buckets for our 200,000 words. Since the vast majority
of English words would be hashed into the first 15 or so buckets, this
is not a good hash function because it's not spreading elements evenly
across our buckets.


P.S. the longest English word is only 45 letters long, so all the
buckets after bucket 45 would be completely empty!!
```

6 Graphs [15 points]

Part A

You are given the following graph and want to find the shortest weighted distance from D to B, from D to F, and from D to G.



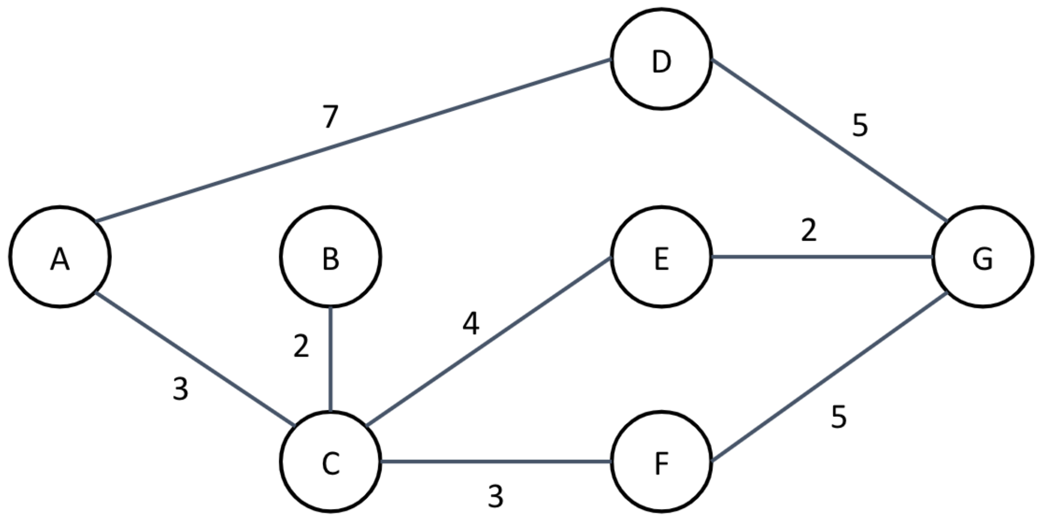
What is the most efficient shortest path algorithm in this case and why? Answer in 1-3 sentences.

Dijkstra's algorithm.

We wouldn't want to use BFS or DFS because these don't work on weighted graphs. We wouldn't want to use A∗ either because it requires that we know some heuristic information about our graph, which we don't. That leaves us with Dijkstra, which can calculate the shortest weighted distance between a node (e.g. node D) and all other nodes in a graph, which is exactly what we need.

Part B

This is the same graph as the one shown on the previous page, replicated for your convenience.



Given this graph, you must now use breadth-first search to find a path from `A` to `G`. You must add neighbor nodes to your Stack and/or Queue in alphabetical order. Answer the following questions about your breadth-first search. You may use this page for scratch work, but any work outside of the answer boxes will not be graded.

What path did you find from `A` to `G` when using breadth-first search?

```
A, D, G
```

List the nodes in the order that you visited them during your breadth-first search.

```
A, C, D, B, E, F, G

// explanation not needed, but if you're curious:
We "visit" a node when we dequeue it and add its neighbors to our queue.
Queue: {A}, Visited: {}
First, we dequeue and visit A, adding its neighbors C and D to our queue in alphabetical order.
Queue: {C, D}, Visited: {A}
Then, we dequeue and visit C, adding its three neighbors B, E, and F to our queue in that order.
Queue: {D, B, E, F}, Visited: {A, C}
Then, we dequeue and visit D, adding its neighbor G to our queue.
Queue: {B, E, F, G}, Visited: {A, C, D}
We dequeue and visit B, then E, then F, none of which have neighbors we haven't already enqueued.
Queue: {G}, Visited: {A, C, D, B, E, F}
Finally, we visit G, which is the node we were searching for.
Queue: {}, Visited: {A, C, D, B, E, F, G}
```