

# CSCI-1200 Data Structures

## Test 2 — Practice Problems

*Note: This packet contains practice problems from three previous exams. Your exam will contain approximately one third as many problems.*

### 1 Dynamic Tetris Arrays [ /26]

#### 1.1 HW3 Tetris Implementation Order Notation [ /6]

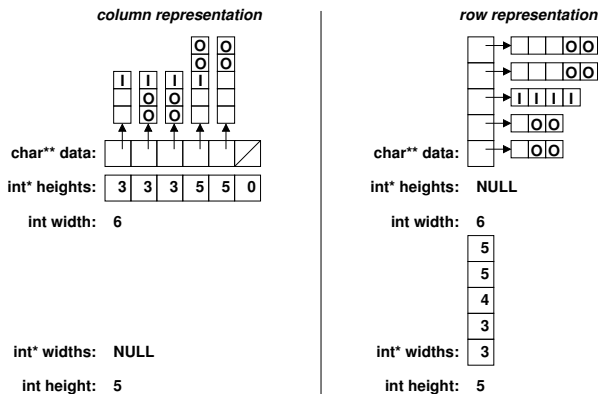
Match up the Tetris class member functions from HW3 with the appropriate order notation, where  $w$  is the width of the board and  $h$  is the maximum height of any column. Assume the solution is efficient, but uses only the 3 member variables specified in the original assignment (`data`, `heights`, and `width`).

*Note: Some letters may be used more than once or not at all.*

<div style="border: 1px solid black; width: 60px; height: 60px; margin: 0 auto;"></div>	<code>void add_piece(char piece,int rotation,int position);</code>	a) $O(1)$
<div style="border: 1px solid black; width: 60px; height: 60px; margin: 0 auto;"></div>	<code>int get_width();</code>	b) $O(w)$
<div style="border: 1px solid black; width: 60px; height: 60px; margin: 0 auto;"></div>	<code>int remove_full_rows();</code>	c) $O(h)$
<div style="border: 1px solid black; width: 60px; height: 60px; margin: 0 auto;"></div>	<code>int get_max_height();</code>	d) $O(w + h)$
<div style="border: 1px solid black; width: 60px; height: 60px; margin: 0 auto;"></div>	<code>void destroy();</code>	e) $O(w * h)$

#### 1.2 Tetris Representation Conversion [ /20]

Now let's revisit the details of the dynamic memory representation for the game of Tetris. Your task is to convert a Tetris board from the *column representation* we used for HW3 to a *row representation*. In addition to the three member variables in our HW3 Tetris class: `data`, `heights`, and `width`, we add 2 additional member variables: `widths` and `height`. In the column representation we don't need the `widths` variable, so it is set to `NULL`. Each time the board is modified to add Tetris pieces or score full rows the `height` variable is updated as necessary to store the maximum height of any column.



The diagram on the left shows an example Tetris board first in *column representation* and then in *row representation* — the “before” and “after” diagrams for a call to the new Tetris class member function `convert_to_row_representation`. Note that once in row representation the `heights` variable isn't needed and we set it to `NULL`. The `convert_to_row_representation` function takes no arguments.

Now write the `Tetris` class member function `convert_to_row_representation` as it would appear in the `tetris.cpp` implementation file. You may assume that before the call the board is in the column representation and the member variables are all set correctly. Make sure your code properly allocates new memory as needed and does not have memory leaks.

*sample solution: 23 line(s) of code*

## 2 Mystery Recursion [ /9]

For each function or pair of functions below, choose the letter that best describes the program purpose or behavior.

A ) infinite loop

B ) factorial

C ) integer power

D ) the answer is 42

E ) function is not recursive

F ) sum of the digits

G ) syntax error

H ) modulo 2

I ) reverse the digits

J ) multiplication

K ) greatest common divisor

L ) other

```
int mysteryONE(int x, int y) {
    if(y == 0)
        return x;
    else
        return mysteryONE(y, x % y);
}
```

```
int mysteryFOUR(int x, int y) {
    if (x == 0)
        return 0;
    else
        return y +
            mysteryFOUR(x-1,y);
}
```

```
int mysteryTWO(int x) {
    if (x == 0)
        return 0;
    else
        return mysteryTWO(x/10)
            + x%10;
}
```

```
int mysteryFIVEa(int x, int y) {
    if (x == 0)
        return y;
    else
        return mysteryFIVEa
            (x/10, y*10 + x%10);
}
```

```
int mysteryFIVEb(int x) {
    return mysteryFIVEa(x,0);
}
```

```
int mysteryTHREEa(int x);

int mysteryTHREEb(int x) {
    if (x == 0)
        return 1;
    else
        return mysteryTHREEa(x-1);
}

int mysteryTHREEa(int x) {
    if (x == 0)
        return 0;
    else
        return mysteryTHREEb(x-1);
}
```

```
int mysterySIX(int x) {
    if (x == 0)
        return 1;
    else
        return x *
            mysterySIX(x-1);
}
```

### 3 Collecting Words [ / 18 ]

Write a function named `Collect` that takes in two *alphabetically sorted* STL lists of STL strings named `threes` and `candidates`. The function searches through the second list and removes all three letter words and places them in the first list in alphabetical order. For example, given these lists as input:

```
threes:      cup dog fox map
candidates:  ant banana egg goat horse ice jar key lion net
```

After the call to `Collect(threes, candidates)` the lists will contain:

```
threes:      ant cup dog egg fox ice jar key map net
candidates:  banana goat horse lion
```

If there are  $n$  and  $m$  words in the input lists, the order notation of your solution should be  $O(n + m)$ .

*sample solution: 15 line(s) of code*

## 4 Constantly Referencing DSStudent [ / 12 ]

The expected output of the program below is:

```
chris is a sophomore, his/her favorite color is blue, and he/she has used 1 late day(s).
```

However, there are a number of small but problematic errors in the `DSStudent` class code. Hint: This problem's title is relevant! Only one completely new line may be added (line 6), and the 7 other lines require one or more small changes. These lines are tagged with an asterisk, \*. Your task is to rewrite each incorrect or missing line in the appropriately numbered box. *Please write the entire new line in the box.*

```
1 class DSStudent {
2     public:
3     DSStudent(std::string n, int y)
4         : name(n) {
5         int entryYear = y;
6     }
7     std::string& getName() const {
8         return name;
9     }
10    const std::string& getYear() {
11        if (entryYear == 2014) {
12            return "freshman"; }
13        } else if (entryYear == 2013) {
14            return "sophomore";
15        } else if (entryYear == 2012) {
16            return "junior";
17        } else {
18            return "senior";
19        }
20    }
21    void incrLateDaysUsed() const {
22        days++;
23    }
24    int& getLateDaysUsed() const {
25        return days;
26    }
27    std::string FavoriteColor() {
28        return color;
29    }
30    private:
31    std::string name;
32    std::string color;
33    int entryYear;
34    int days;
35 };
36
37 int main() {
38     DSStudent s("chris",2013);
39     s.FavoriteColor() = "blue";
40     s.incrLateDaysUsed();
41     std::cout << s.getName()
42               << " is a " << s.getYear()
43               << ", his/her favorite color is " << s.FavoriteColor()
44               << ", and he/she has used " << s.getLateDaysUsed()
45               << " late day(s)." << std::endl;
46 }
47 }
```

3

5

6

8

11

22

25

28

## 5 Efficient Occurrences [ / 22 ]

Write a *recursive* function named `Occurrences` that takes in a *sorted* STL `vector` of STL `strings` named `data`, and an STL `string` named `element`. The function returns an integer, the number of times that `element` appears in `data`. Your function should have order notation  $O(\log n)$ , where  $n$  is the size of `data`.

*sample solution: 21 line(s) of code*

## 6 Short Answer [ / 8 ]

### 6.1 What's Wrong? [ / 4 ]

Write 1-2 complete and concise sentences describing the problem with this code fragment:

```
std::vector<std::string> people;
people.push_back("sally");
people.push_back("brian");
people.push_back("monica");
people.push_back("fred");
std::vector<std::string>::iterator mom = people.begin() + 2;
std::vector<std::string>::iterator dad = people.begin() + 1;
people.push_back("paula");
std::cout << "My parents are " << *mom << " and " << *dad << std::endl;
```

### 6.2 Fear of Recursion [ / 4 ]

Rewrite this function without recursion:

```
class Node {
public:
    std::string value;
    Node* next;
};
```

```
void printer (Node* n) {
    if (n->next == NULL) {
        std::cout << n->value;
    } else {
        std::cout << "(" << n->value << "+";
        printer (n->next);
        std::cout << ")";
    }
}
```

*sample solution: 13 line(s) of code*

## 7 Converting Between Vec and dslist [ / 26 ]

Ben Bitdiddle is working on a project that stores data with two different data structures: our `Vec` and `dslist` classes. Occasionally he needs to convert data from one format to the other format. Alyssa P. Hacker suggests that he write a copy-constructor-like function for each class that takes in a single argument, the original format of the data. For example, here's how to convert data in `Vec` format to `dslist` format:

```
// create a Vec object with 4 numbers
Vec<int> v; v.push_back(1); v.push_back(2); v.push_back(3); v.push_back(4);
// create a dslist object that initially stores the same data as the Vec object
dslist<int> my_lst(v);
```

Here are the relevant portions of the two class declarations (and the `Node` helper class):

```
template <class T> class Node {
public:
    Node(const T& v):
        value_(v),next_(NULL),prev_(NULL){}
    T value_;
    Node<T>* next_;
    Node<T>* prev_;
};

template <class T> class Vec {
public:
    // conversion constructor
    Vec(const dslist<T>& lst);
    /* other functions omitted */
    // representation
    T* m_data;
    unsigned int m_size;
    unsigned int m_alloc;
};

template <class T> class dslist {
public:
    // conversion constructor
    dslist(const Vec<T>& vec);
    /* other functions omitted */
    // representation
    Node<T>* head_;
    Node<T>* tail_;
    unsigned int size_;
};
```

Ben asks about access to the private member variables of one class from a member function of the other. Alyssa says he can write the functions assuming he has full access to the private member variables. (She promises to teach him how to use the `friend` keyword to make that work after Test 2.)

### 7.1 Diagrams [ / 8 ]

First, draw the detailed internal memory representations for a `Vec` object and a `dslist` object, each storing the numbers: 1 2 3 4.

`m_data:`

`m_alloc:`

`m_size:`

`tail_:`

`head_:`

`size_:`



## 7.2 Implementing the Conversion Constructors [ / 18 ]

Now write the two conversion constructors. You may not use `push_back`, `push_front`, `insert` or iterators in your answer. Instead, demonstrate that you know how to construct and manipulate the low level memory representation.

```
template <class T> Vec<T>::Vec(const dslist<T>& lst) {
```

*sample solution: 13 line(s) of code*

```
template <class T> dslist<T>::dslist(const Vec<T>& v) {
```

*sample solution: 13 line(s) of code*

## 8 Short Answer [ /11]

### 8.1 Map Efficiency [ /2]

Suppose you have a map data structure with  $n$  key/value entries, and the value of each entry is a list of at most  $m$  items. What is the order notation for the worst case running time to lookup and print all of the items stored in the list associated with a particular key?

### 8.2 Container Objects [ /12]

For each of the characteristics below, indicate by letter the container object(s) that have that characteristic. The container objects we have learned:

a) array      b) vector      c) list      d) map      e) set

duplicates are not allowed

efficient (sublinear) erase

allows random access

automatically resizes as necessary

requires definition of `operator<`

entries cannot be modified after they are inserted

## 9 Valet Parking Maps [ /38]

You have been asked to help with a valet parking system for a big city hotel. The hotel must keep track of all of the cars currently stored in their parking garage and the names of the owners of each car. *Please read through the entire question before working on any of the subproblems.* Here is the simple `Car` class they have created to store the basic information about a car:

```

class Car {
public:
    // CONSTRUCTOR
    Car(const string &m, const string &c) : maker(m), color(c) {}
    // ACCESSORS
    const string& getMaker() const { return maker; }
    const string& getColor() const { return color; }
private:
    // REPRESENTATION
    string maker;
    string color;
};

```

The hotel staff have decided to build their parking valet system using a map between the cars and the owners. This map data structure will allow quick lookup of the owners for all the cars of a particular color and maker (e.g., the owners of all of the silver Hondas in the garage). For example, here is their data structure and how it is initialized to store data about the six cars currently in the garage.

```

map<Car,vector<string> > cars;
cars[Car("Honda","blue")].push_back("Cathy");
cars[Car("Honda","silver")].push_back("Fred");
cars[Car("Audi","silver")].push_back("Dan");
cars[Car("Toyota","green")].push_back("Alice");
cars[Car("Audi","silver")].push_back("Erin");
cars[Car("Honda","silver")].push_back("Bob");

```

The managers also need a function to create a report listing all of the cars in the garage. The statement:

```
print_cars(cars);
```

will result in this report being printed to the screen (`std::cout`):

```

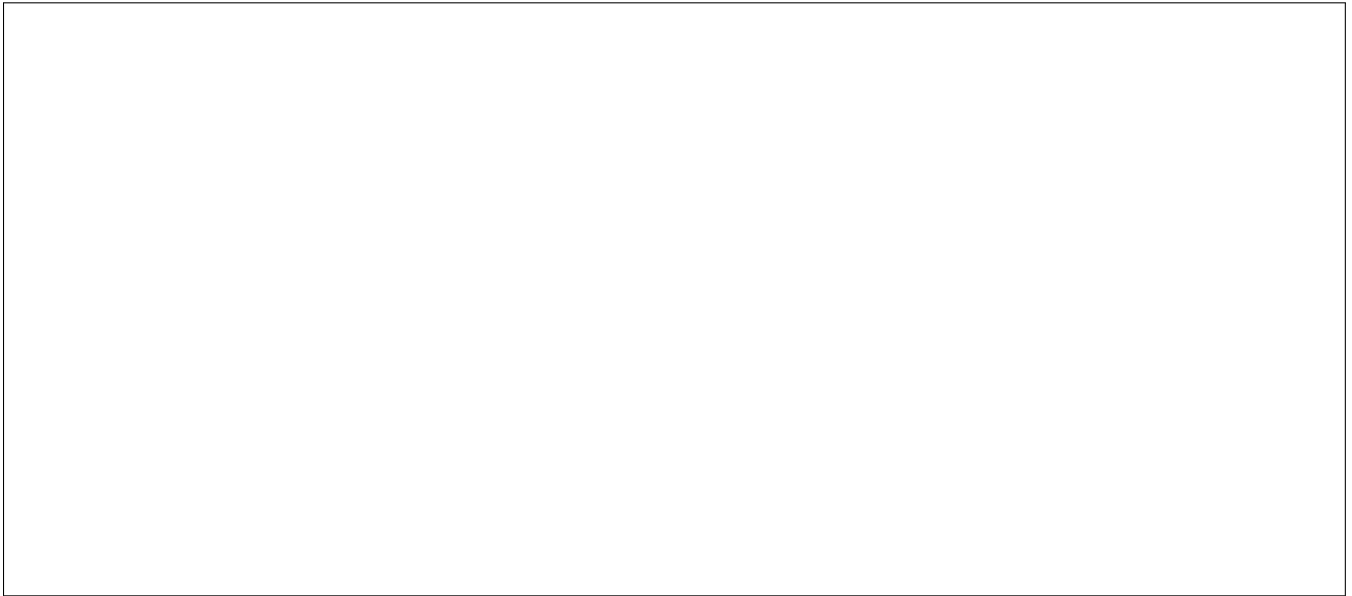
People who drive a silver Audi:
    Dan
    Erin
People who drive a blue Honda:
    Cathy
People who drive a silver Honda:
    Fred
    Bob
People who drive a green Toyota:
    Alice

```

Note how the report is sorted alphabetically by maker, then by car color, and that the owners with similar cars are listed chronologically (the order in which they parked in the garage).

## 9.1 The Car class [ /6]

In order for the `Car` class to be used as the first part of a map data structure, what additional non-member function is necessary? Write that function. Carefully specify the function prototype (using `const` & reference as appropriate). Use the example above as a guide.



### 9.2 Data structure diagram [ /10]

Draw a picture of the map data structure stored by the `cars` variable in the example. As much as possible use the conventions from lecture for drawing these pictures. Please be neat when drawing the picture. *Optional: You may also write a few concise sentences to explain your picture.*



### 9.3 `print_cars` [ /9]

Write the `print_cars` function. Part of your job is to correctly specify the prototype for this function. Be sure to use `const` and pass by reference as appropriate.

#### 9.4 remove\_cars [ /13]

When guests pick up their cars from the garage, the data structure must be correctly updated to reflect this change. The `remove_car` function returns true if the specified car is present in the garage and false otherwise.

```
bool success;  
success = remove_car(cars, "Erin", "silver", "Audi");  
assert (success == true);  
success = remove_car(cars, "Cathy", "blue", "Honda");  
assert (success == true);  
success = remove_car(cars, "Sally", "green", "Toyota");  
assert (success == false);
```

After executing the above statements the `cars` data structure will print out like this:

```
People who drive a silver Audi:  
  Dan
```

```
People who drive a silver Honda:
  Fred
  Bob
People who drive a green Toyota:
  Alice
```

Note that once the only blue Honda stored in the garage has been removed, this color/maker combination is completely removed from the data structure.

Specify the prototype and implement the `remove_car` function.

## 10 Movies w/ Pair/Map/String/List/Vector/Set [ /41]

In this problem we will organize people and the movies they have seen in the map data structure illustrated on the right. Specifically, we are interested in exploring which people saw different pairs of movies. In the example to the right, we can easily see that three people (“Alice”, “Bob”, and “Dan”) have seen both “Rocky” and “Titanic”, and “Alice” is the only person to have seen “Titanic” and “WALL-E”.

("Rocky", "Star Wars")	"Bob"
("Rocky", "Titanic")	"Alice" "Bob" "Dan"
("Rocky", "WALL-E")	"Alice"
("Star Wars", "Titanic")	"Bob" "Carol"
("Titanic", "WALL-E")	"Alice"

### 10.1 Defining the Map Type [ /5]

First, let’s create a shorthand for the type of the structure illustrated above:

```
typedef    *** PART 1 ANSWER HERE ***    MOVIE_MAP;
```

### 10.2 Counting Combos [ /8]

Now, assuming the map structure is stored in a variable named `my_map`, write a fragment of code to store in the variable `count` the number of people who have seen the two movies stored in the `std::string` variables `movie_a` and `movie_b`.

If we have movie information for  $p$  people and  $m$  movies, and each person has seen no more than  $k$  movies, and each movie is seen by at most  $j$  people, what is the running time of your solution?

### 10.3 Adding Data [ /13]

Next, write the function `AddPerson` which takes 3 arguments: an `std::string` representing the name of a new person to add to the map structure, an `std::list` of the names of the movies that person has seen, and the map structure (type = `MOVIE_MAP`). This function should modify the map structure as needed to record this person's movie-going data.

You may assume the person has not already been added to the database. You may also assume that the movie list contains at least 2 movies and that there are no duplicates in the movie list.

If we have movie information for  $p$  people and  $m$  movies, and each person has seen no more than  $k$  movies, and each movie is seen by at most  $j$  people, what is the running time of your solution?



#### 10.4 You Haven't Seen “*Star Wars*” Yet??? [ /15]

Finally, write the function `DidNotSee` which takes 2 arguments: an `std::string` representing the name of a movie and the map structure, and returns an `std::list` with the names of all people in the structure who have not seen the movie. `DidNotSee` with “*Star Wars*” and the sample data returns a list with 2 people: “*Alice*” and “*Dan*”. Your function should use `std::set`.

If we have movie information for  $p$  people and  $m$  movies, and each person has seen no more than  $k$  movies, and each movie is seen by at most  $j$  people, what is the running time of your solution?

## 11 Matrix Transpose [ / 20 ]

First, study the partial implementation of the templated `Matrix` class on the right. Your task is to implement the `transpose` member function for this class (as it would appear outside of the class declaration). Remember from math class that the transpose flips the matrix data along the diagonal from the upper left corner to the lower right corner. For example:

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \xrightarrow{\text{transpose}} \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix}$$

```
template <class T> class Matrix {
public:
    Matrix(int rows, int cols, const T &v);
    ~Matrix();
    int getRows() const { return rows_; }
    int getCols() const { return cols_; }
    const T& get(int r, int c) const
        { return values[r][c]; }
    void set(int r, int c, const T &v)
        { values[r][c] = v; }
    void transpose();
private:
    int rows_;
    int cols_;
    T **values;
};
```

*sample solution: 18 line(s) of code*

## 12 Book, Page, Sentence, & Word Iteration [ / 18 ]

Write a function `PageWithMostSentencesWithWord` that takes in two arguments. The first argument is an STL `list` of STL `lists` of STL `lists` of STL `strings` that represents a book with pages. Each page has multiple sentences. Each sentence has multiple words. The second argument is an STL `string` with the search word. The function should return the page number that has the most sentences that contain the search word. The first page in the book is numbered 1 (not zero). You may assume that any punctuation has already been removed and everything has been converted to lowercase.

*sample solution: 24 line(s) of code*

### 13 Linear 2048 [ / 18 ]

Write a *recursive* function named `Linear2048` that takes in an STL list of integers and plays a single line based version of the 2048 game. If two adjacent numbers are equal to each other in value, those two elements merge and are replaced with their sum. The function returns the maximum value created by any of the merges during play. The example shown on the right reduces the original input list with 17 values to a list with 4 values and returns the value 2048.

```
8 2 2 1024 256 32 16 8 4 1 1 2 32 32 128 512 32
8 4 1024 256 32 16 8 4 1 1 2 32 32 128 512 32
8 4 1024 256 32 16 8 4 2 2 32 32 128 512 32
8 4 1024 256 32 16 8 4 4 32 32 128 512 32
8 4 1024 256 32 16 8 8 32 32 128 512 32
8 4 1024 256 32 16 16 32 32 128 512 32
8 4 1024 256 32 32 32 32 128 512 32
8 4 1024 256 64 32 32 128 512 32
8 4 1024 256 64 64 128 512 32
8 4 1024 256 128 128 512 32
8 4 1024 256 256 512 32
8 4 1024 512 512 32
8 4 1024 1024 32
8 4 2048 32
```

*sample solution: 15 line(s) of code*

## 14 Mystery Function Memory Usage Order Notation [ / 6 ]

What does this function compute? What is the order notation of the size of the memory necessary to store the return value of this function? Give your answer in terms of  $n$ , the number of elements in the input vector, and  $k$ , the average or worst case length of each string in the input vector. Write 3-4 concise and well-written sentences to justify your answer.

```
std::vector<std::string> mystery(const std::vector<std::string> &input) {
    if (input.size() == 1) { return input; }
    std::vector<std::string> output;
    for (int i = 0; i < input.size(); i++) {
        std::vector<std::string> helper_input;
        for (int j = 0; j < input.size(); j++) {
            if (i == j) continue;
            helper_input.push_back(input[j]);
        }
        std::vector<std::string> helper_output = mystery(helper_input);
        for (int k = 0; k < helper_output.size(); k++) {
            output.push_back(input[i] + ", " + helper_output[k]);
        }
    }
    return output;
}
```

## 15 LeapFrogSplit on a Doubly-Linked List [ / 26 ]

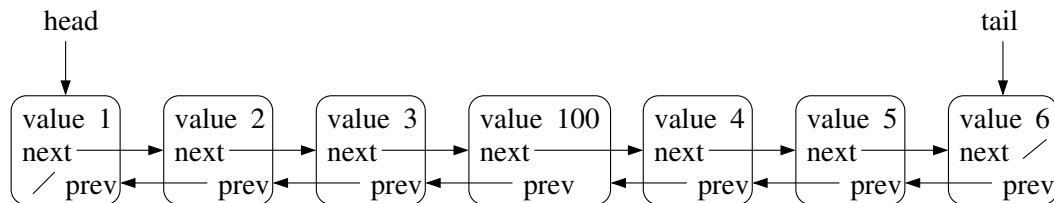
In this problem, we will implement the `LeapFrogSplit` function which manipulates a doubly-linked list of `Nodes`. This function takes in 3 arguments: pointers to the *head* & *tail* `Nodes` of a doubly-linked list, and an integer *value*. The function locates the `Node` containing that value, removes the node, splits the value in half, and re-inserts the half values into the list jumping over both of the original neighbors before and after it in the list.

For example, if the linked list initially contains 7 nodes with the data:  
1 2 3 100 4 5 6, then after executing `LeapFrogSplit(head,tail,100)`  
it will contain 8 nodes: 1 2 50 3 4 50 5 6.

```
class Node {  
public:  
    Node(int v) :  
        value(v),  
        next(NULL),  
        prev(NULL) {}  
    int value;  
    Node* next;  
    Node* prev;  
};
```

### 15.1 Diagram [ / 5 ]

First, modify the diagram below to illustrate the result of `LeapFrogSplit(head,tail,100)`.



### 15.2 Corner Cases & Testing [ / 7 ]

What “corner cases” do you need to consider for this implementation? Give 4 interesting examples of input and what you define as the correct result for each case. Write 2-3 explanatory sentences as needed.

### 15.3 Implementing LeapFrogSplit [ / 14 ]

Finally, write `LeapFrogSplit`. Focus primarily on correctly performing the general case that we diagrammed on the previous page. Corner cases are worth only a small number of points.

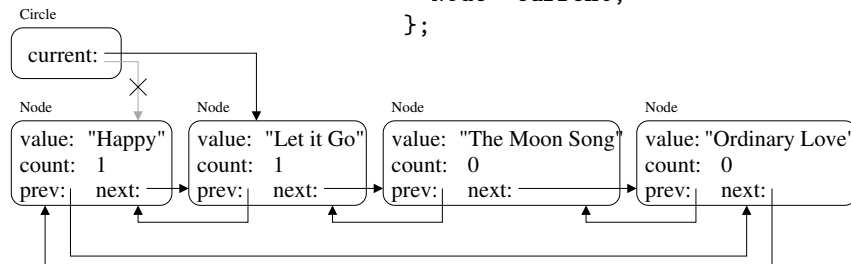
*sample solution: 36 line(s) of code*

## 16 Circular Play List [ / 27 ]

In this problem we will create a simple doubly-linked circular data structure to store a play list of songs represented as STL strings. Here's a portion of the class declaration:

```
class Node {
public:
    Node(const std::string& val) :
        value(val), count(0),
        prev(NULL), next(NULL) {}
    std::string value;
    int count;
    Node* prev;
    Node* next;
};

class Circle {
public:
    //
    // PROTOTYPES OF TWO FUNCTIONS YOU WILL WRITE
    //
    void play();
private:
    Node* current;
};
```



And here is a sample usage of the `Circle` class to store the Oscar nominees for “Best Song”:

```
std::vector<std::string> songs;
songs.push_back("Happy");
songs.push_back("Let it Go");
songs.push_back("The Moon Song");
songs.push_back("Ordinary Love");
Circle oscar_nominees(songs);
for (int i = 0; i < 3; i++) {
    oscar_nominees.play(); }
std::cout << "--- editing the song list ---" << std::endl;
bool success = oscar_nominees.remove("Let it Go");
assert (success == true);
success = oscar_nominees.remove("Atlas");
assert (success == false);
for (int i = 0; i < 6; i++) {
    oscar_nominees.play(); }
```

Which results in this output:

```
now playing: Happy
now playing: Let it Go
    (last song was Happy)
now playing: The Moon Song
    (last song was Let it Go)
--- editing the song list ---
now playing: Ordinary Love
    (last song was The Moon Song)
now playing: Happy, played 1 time(s) previously
    (last song was Ordinary Love)
now playing: The Moon Song, played 1 time(s) previously
    (last song was Happy)
now playing: Ordinary Love, played 1 time(s) previously
    (last song was The Moon Song)
now playing: Happy, played 2 time(s) previously
    (last song was Ordinary Love)
now playing: The Moon Song, played 2 time(s) previously
    (last song was Happy)
```



Here's the implementation of one of the functions used on the previous page. You need to implement the other two missing functions so that the program performs as shown in the example.

```
void Circle::play() {
    if (current == NULL) return;
    std::cout << "now playing: " << current->value;
    if (current->count > 0) {
        std::cout << ", played " << current->count << " time(s) previously";
    }
    std::cout << std::endl;
    if (current->prev->count != 0)
        std::cout << "    (last song was " << current->prev->value << ")" << std::endl;
    current->count++;
    current = current->next;
}
```

## 16.1 Circle constructor [ / 12 ]

First, implement the constructor used in the example on the previous page as it would appear in the .cpp file. Of course, make sure your function also handles input song lists with more or fewer songs.

*sample solution: 15 line(s) of code*

## 16.2 Implementing remove [ / 15 ]

Now, implement the `remove` function as it would appear in the `.cpp`. Study the provided example carefully, but also make sure that your function works for all corner cases as well.

*sample solution: 21 line(s) of code*

## 17 Common Data [ / 20 ]

Write a templated function `common_data` that takes in two STL **vectors** of type `T` and returns an STL **vector** of type `T` that contains all of the common elements; that is, only if an element is in *both* of the input vectors will it be added to the output vector. The input vectors may contain duplicates, but your output vector should not. You are not allowed to edit the input vectors.

*sample solution: 22 line(s) of code*

**Order Notation** If there are  $n$  elements in the first input vector and  $m$  elements in the second input vector, what is the order notation of your solution?

## 18 Possessive Grammar [ / 22 ]

Write a function `convert_to_possessive` that takes in one argument, an STL list of `strings` representing a sentence, and edits the sentence to replace the pattern “the AAA of BBB” with “BBB’s AAA”.

For example,            `i like the hat of sarah`  
is rewritten as        `i like sarah's hat`

And                    `the car of joe is parked between the van of chris and a motorcycle`  
is rewritten as        `joe's car is parked between chris's van and a motorcycle`

You may assume that the words are all lowercase and the input sentence contains no punctuation.

*sample solution: 21 line(s) of code*

## 19 Mysterious Memory Errors [ / 15 ]

The program below contains numerous memory-related errors. Your task is to identify and fix each problem.

```
01 int main() {
02     int max_index = 20;
03     int* data = new int[max_index];
04     data[0] = 0;
05     data[1] = 1;
06     int* tmp = new int;
07     for (int i = 0; i < max_index; i++) {
08         *tmp = data[i] + data[i+1];
09         data[i+2] = *tmp;
10     }
11     int* answer = new int;
12     for (tmp = data; tmp < data+max_index; tmp++) {
13         if (*tmp % 2 == 1) (*answer)++;
14     }
15     tmp = answer;
16     std::cout << "mystery answer => " << *answer << std::endl;
17     delete data;
18     delete answer;
19     delete tmp;
20     return 0;
21 }
```

A MEMORY LEAK is reported for the allocation on line . It can be fixed by moving line

immediately after line . (Note: the line is causing a MEMORY ALREADY FREED error in its current location.) The memory debugger reports use of UNINITIALIZED MEMORY on line

. It can be fixed by adding this line of code

immediately after line . A MISMATCHED NEW/NEW[]/DELETE/DELETE[] is reported on

line . It is fixed by changing that same line to .

An INVALID READ is reported on line  and an INVALID WRITE is reported on line .

Both errors can be fixed by editing line  to be .

Once all of these errors are corrected, the program calculates a simple, yet interesting, statistic. Describe in 1 or 2 sentences the mystery answer calculated by this program.

## 20 Recursive Order Notation Challenge [ / 13 ]

Write a recursive function `FooA` that takes a single integer argument  $n$  that has order notation  $O(n)$ .

*sample solution: 7 line(s) of code*

Write a recursive function `FooB` that takes a single integer argument  $n$  that has order notation  $O(\log n)$ .

*sample solution: 7 line(s) of code*

Write a recursive function `FooC` that takes a single integer argument  $n$  that has order notation  $O(2^n)$ .

*sample solution: 7 line(s) of code*