CSCI-1200 Data Structures
Test 2 — Practice Problem Solutions

1 Dynamic Tetris Arrays [/26]

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

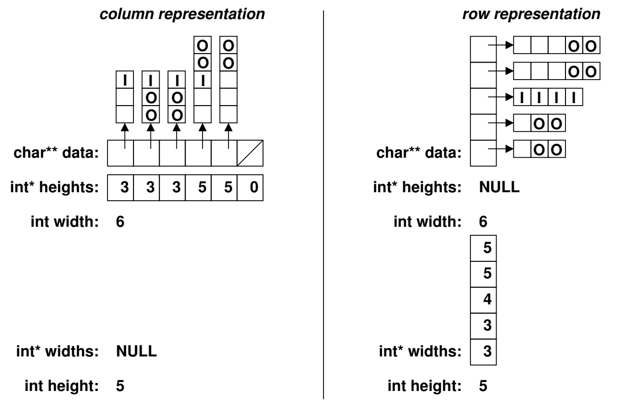
Grading Note: -1.5pts each unanswered or incorrect.

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.

Solution: **c**	`void add_piece(char piece,int rotation,int position);`	a)	$O(1)$
a	`int get_width();`	b)	$O(w)$
e	`int remove_full_rows();`	c)	$O(h)$
b	`int get_max_height();`	d)	$O(w + h)$
e	`void destroy();`	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.

Solution:

```
void Tetris::convert_to_row_representation() {
  // allocate the top level arrays
  widths = new int[height];
  char** tmp = new char*[height];
  // for each row...
  for (int h = 0; h < height; h++ ) {
    // calculate the width of each row
    widths[h] = 0;
    for (int w = 0; w < width; w++ ) {
      if (heights[w] > h && data[w][h] != ' ') widths[h] = w+1;
    }
    // allocate a row of the correct width in the tmp structure
    assert (widths[h] > 0);
    tmp[h] = new char[widths[h]];
    // fill in the row character data
    for (int w = 0; w < widths[h]; w++) {
      if (heights[w] > h)
        tmp[h][w] = data[w][h];
      else
        tmp[h][w] = ' ';
    }
  }
  // cleanup the old structure
  delete [] heights;
  heights = NULL;
  for (int i = 0; i < width; i++) {
    delete [] data[i];
  }
  delete [] data;
  // point to the new data
  data = tmp;
}
```

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	E) function is not recursive	I) reverse the digits
B) factorial	F) sum of the digits	J) multiplication
C) integer power	G) syntax error	K) greatest common divisor
D) the answer is 42	H) modulo 2	L) other

Solution: K

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

Solution: F

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

Solution: H

```
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);
}
```

Solution: J

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

Solution: I

```
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);
}
```

Solution: B

```
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 `list`s of STL `string`s 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 $O(n + m)$.

Solution:

```
void collect(std::list<std::string> &threes, std::list<std::string> &candidates) {
  // start an iterator at the front of each list
  std::list<std::string>::iterator itr = threes.begin();
  std::list<std::string>::iterator itr2 = candidates.begin();
  // loop over all of candidate words
  while (itr2 != candidates.end()) {
    // if the candidate is length 3
    if ((*itr2).size() == 3) {
      // find the right spot for this word
      while (itr != threes.end() && *itr < *itr2) {
        itr++;
      }
      // modify the two lists
      threes.insert(itr,*itr2);
      itr2 = candidates.erase(itr2);
    } else {
      // only advance the pointer if the length is != 3
      itr2++;
    }
  }
}
```

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   }
* 8   std::string& getName() const {
  9     return name;
 10   }
*11   const std::string& getYear() {
 12     if (entryYear == 2014) {
 13       return "freshman"; }
 14     } else if (entryYear == 2013) {
```

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

3 **Solution:** `DSStudent(const std::string &n, int y)`

5 **Solution:** `entryYear = y;`

6 **Solution:** `days = 0;`

8 **Solution:** `const std::string& getName() const {`

11 **Solution:** `std::string getYear() const {`

22 **Solution:** `void incrLateDaysUsed() {`

25 **Solution:** `int getLateDaysUsed() const {`

28 **Solution:** `std::string& FavoriteColor() {`

5 Efficient Occurrences [/ 22]

Write a *recursive* function named `Occurrences` that takes in a *sorted* STL `vector` of STL `string`s 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`.

Solution:

```
// the recursive helper function
int occurrences(const std::vector<std::string> &data, const std::string &element,
               int s1, int s2, int e1, int e2) {
  // s1 & s2 are the current range for the start / first occurence
  // e1 & e2 are the current range for the end / last occurence (+1)
  assert (s1 <= s2 && e1 <= e2);
  if (s1 < s2) {
    // first use binary search to find the first occurrence of element
    int mid = (s1 + s2) / 2;
    if (data[mid] >= element)
      return occurrences(data,element,s1,mid,e1,e2);
    return occurrences(data,element,mid+1,s2,e1,e2);
  } else if (e1 < e2) {
    // then use binary search to find the last occurrence of element (+1)
    int mid = (e1 + e2) / 2;
    if (data[mid] > element)
      return occurrences(data,element,s1,s2,e1,mid);
    return occurrences(data,element,s1,s2,mid+1,e2);
  } else {
    // the simply subtract these indices
    assert (s1 == s2 && e1 == e2 && e1 >= s1);
    return e1 - s1;
  }
}

// "driver" function
int occurrences(const std::vector<std::string> &data, const std::string &element) {
  // use binary seach twice to find the first & last occurrence of element
  return occurrences(data,element,0,data.size(),0,data.size());
}
```

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;
```

Solution: Any iterators attached to an STL vector should be assumed to be invalid after a call to `push_back` (or `erase` or `resize`) because the internal dynamically allocated array may have been relocated in memory (or the data shifted). Dereferencing the pre-`push_back` iterators to print the data is dangerous since that memory may have been deleted/freed.

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 << ")";
  }
}
```

Solution:

```
void printer (Node* n) {
  int count = 0;
  while (n != NULL) {
    if (n->next != NULL) {
      std::cout << "(" << n->value << "+";
      count++;
    } else {
      std::cout << n->value;
    }
    n = n->next;
  }
  std::cout << std::string(count,')');
}
```

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

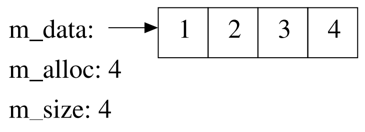
```
template <class T> class Node {
public:
  Node(const T& v):
    value_(v),next_(NULL),prev_(NULL){}
  T value_;
  Node<T>* next_;
  Node<T>* prev_;
};

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.)

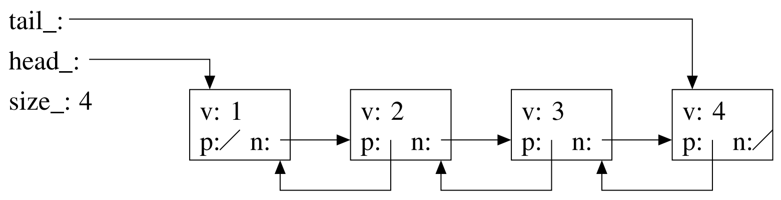
7.1 Diagrams [/ 8]

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

Solution:



Solution:



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.

Solution:

```
template <class T> Vec<T>::Vec(const dslist<T>& lst) {
  m_alloc = m_size = lst.size();
  if (m_alloc > 0)
    m_data = new T[m_alloc];
  else
    m_data = NULL;
  int i = 0;
  Node<T> *tmp = lst.head_;
  while (tmp != NULL) {
    m_data[i] = tmp->value_;
    tmp = tmp->next_;
    i++;
  }
}


template <class T> dslist<T>::dslist(const Vec<T>& v) {
  head_ = tail_ = NULL;
  size_ = v.size();
  Node<T> *tmp = NULL;
  for (int i = 0; i < size_; ++i) {
    tail_ = new Node<T>(v.m_data[i]);
    if (tmp != NULL) {
      tail_->prev_ = tmp;
      tmp->next_ = tail_;
    }
    if (i == 0) head_ = tail_;
    tmp = tail_;
  }
}
```

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?

Solution: O(m + log n)

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

Solution:

Solution:	**(d) e**	duplicates are not allowed
	c d e	efficient (sublinear) erase
	a b	allows random access
	b c d e	automatically resizes as necessary
	d e	requires definition of `operator<`
	(d) e	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.

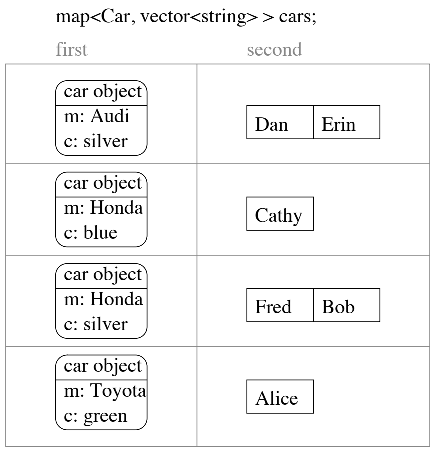
Solution: We must define `operator<` for `Car` objects so that we can sort the keys of the map.

```
bool operator<(const Car &a, const Car &b) {
  return (a.getMaker() < b.getMaker() ||
          (a.getMaker() == b.getMaker() && a.getColor() < b.getColor()));
}
```

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.*

Solution:



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.

Solution:

```
void print_cars(const map<Car,vector<string> > &cars) {
  map<Car,vector<string> >::const_iterator itr = cars.begin();
  while (itr != cars.end()) {
    Car c = itr->first;
    cout << "People who drive a " << c.getColor() << " " << c.getMaker() << ":" << endl;
    vector<string>::const_iterator itr2 = itr->second.begin();
    while (itr2 != itr->second.end()) {
      cout << "  " << *itr2 << endl;
      itr2++;
    }
    itr++;
  }
}
```

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
```

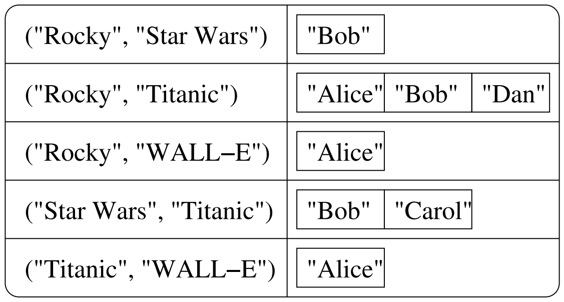
Note that once the only blue Honda stored in the garage has been removed, this color/maker combination is completely removed from the data structure.

Specify the prototype and implement the `remove_car` function.

Solution:

```
bool remove_car(map<Car,vector<string> > &cars,
                const string &name, const string &color, const string &maker) {
  map<Car,vector<string> >::iterator itr = cars.find(Car(maker,color));
  if (itr == cars.end()) return false;
  if (itr->second.size() == 1 && itr->second[0] == name) {
    cars.erase(Car(maker,color));
    return true;
  }
  for (int i = 0; i < itr->second.size(); i++) {
    if (itr->second[i] == name) {
      itr->second.erase(itr->second.begin() + i);
      return true;
    }
  }
  return false;
}
```

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".

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;`

Solution:

```
typedef std::map < std::pair< std::string, std::string >, std::vector<std::string> > 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`.

Solution:

```
// determine which movie comes first alphabetically
if (movie_a < movie_b) {
  count = my_map[make_pair(movie_a,movie_b)].size();
} else {
  count = my_map[make_pair(movie_b,movie_a)].size();
}
```

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?

Solution: There are at most m^2 rows/entries in the map. Accessing a specific row takes log in the number of rows. Querying the number of entries in a vector is constant-time. Overall: $O(\log m^2)$, which can be simplified to $O(\log m)$.

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.

Solution:

```
void AddPerson (const std::string &name,
                const std::list<std::string> &movies,
                MOVIE_MAP &my_map) {
  // two nested for loops to find all pairs in the input movies list
  for (std::list<std::string>::const_iterator itr = movies.begin();
       itr != movies.end(); itr++) {
    std::list<std::string>::const_iterator itr2 = itr;
    itr2++;
    for (; itr2 != movies.end(); itr2++) {
      // determine which movie comes first alphabetically
      if (*itr < *itr2) {
        my_map[make_pair(*itr,*itr2)].push_back(name);
      } else {
        my_map[make_pair(*itr2,*itr)].push_back(name);
      }
    }
  }
}
```

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?

Solution: We must add/edit k^2 rows of the map. Querying for/adding a row takes log m^2 time. `push_back` is constant-time (amortized). Thus overall: $O(k^2 * \log m^2)$, which can be simplified to $O(k^2 * \log m)$.

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`.

Solution:

```
std::list<std::string> DidNotSee(const std::string &movie, const MOVIE_MAP &my_map) {
  // two helper data structures to collect the people
  std::set<std::string> all_people;
  std::set<std::string> did_see;
  for (MOVIE_MAP::const_iterator itr = my_map.begin(); itr != my_map.end(); itr++) {
    bool flag = itr->first.first == movie || itr->first.second == movie;
    for (std::vector<std::string>::const_iterator itr2 = itr->second.begin();
         itr2 != itr->second.end(); itr2++) {
      all_people.insert(*itr2);
      if (flag) did_see.insert(*itr2);
    }
  }
  // loop through the people in the helper sets to construct the final answer
  std::list<std::string> answer;
  for (std::set<std::string>::iterator itr = all_people.begin(); itr != all_people.end(); itr++) {
    if (did_see.find(*itr) == did_see.end()) {
      answer.push_back(*itr);
    }
  }
  return answer;
}
```

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?

Solution: To build the set of all people (and the set of people who have seen the movie), we must visit every row in the map, and every person in each row, that's $m^2 * j$ items and then we must add them to a set of all people, which has maximum size p (each set insert = log p). Once we have the two sets, for each person in the `all_people` set we search the `did_see` set. That's p log p. Thus, overall: $O((m^2 * j + p) *$ log $p)$

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:

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

```
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;
};
```

Solution:

```
template <class T>
void Matrix<T>::transpose() {
  // move the current matrix out of the way
  T **old = values;
  // create a new top level array to store the rows
  values = new T*[cols_];
  for (int i = 0; i < cols_; i++) {
    // create each row
    values[i] = new T[rows_];
    // populate the values
    for (int j = 0; j < rows_; j++) {
      values[i][j] = old[j][i];
    }
  }
  // clean up the old data
  for (int i = 0; i < rows_; i++) {
    delete [] old[i];
  }
  delete [] old;
  // swap the counters for rows & columns
  int tmp = rows_;
  rows_ = cols_;
  cols_ = tmp;
}
```

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.

Solution:

```
int PageWithMostSentencesWithWord(const std::list<std::list<std::list<std::string> > > &book,
                                  const std::string &search) {
  int current = 0;
  int answer = -1;
  int most;
  std::list<std::list<std::list<std::string> > >::const_iterator page;
  std::list<std::list<std::string> >::const_iterator sentence;
  std::list<std::string>::const_iterator word;
  for (page = book.begin(); page != book.end(); page++) {
    current++;
    int count = 0;
    for (sentence = (*page).begin(); sentence != (*page).end(); sentence++) {
      bool found = false;
      for (word = (*sentence).begin(); word != (*sentence).end(); word++) {
        if (*word == search) found = true;
      }
      if (found) count++;
    }
    if (answer == -1 || most < count) {
      answer = current;
      most = count;
    }
  }
  return answer;
}
```

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
```

Solution:

```
int linear_2048(std::list<int> &input) {
  // nothing to do if there aren't at least 2 elements
  if (input.size() <= 1) return -1;
  // start up 2 side-by-side iterators
  std::list<int>::iterator itr = input.begin();
  std::list<int>::iterator itr2 = itr;
  itr2++;
  // walk down the list, looking for 2 neighboring elements with the same value
  while (itr2 != input.end() && *itr != *itr2) {
    itr++;
    itr2++;
  }
  // if we're at the end of the list, nothing to do
  if (itr2 == input.end()) return -1;
  // double the current value
  *itr = (*itr)*2;
  // erase the element under the other iterator
  input.erase(itr2);
  // write down the current value (itr may be changed by recursive call)
  int a = *itr;
  int b = linear_2048(input);
  // return the larger value
  return std::max(a,b);
}
```

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;
}
```

Solution: This function reserves one element at a time from the input vector, recurses on the remaining vector, and then concatenates the reserved element to the front of each item in the recursion output.

Thus, this function generates all *permutations* of the input vector.

By definition, the number of permutations is $n!$. The length of each permutation is $n * k$ (technically $n * k + (n - 1) * 2$ with the commas and spaces). Therefore, the storage space/memory needed for the output vector is $O(n * k * n!)$.

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

In this problem, we will implement the `LeapFrogSplit` function which manipulates a doubly-linked list of `Node`s. This function takes in 3 arguments: pointers to the *head* & *tail* `Node`s 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.

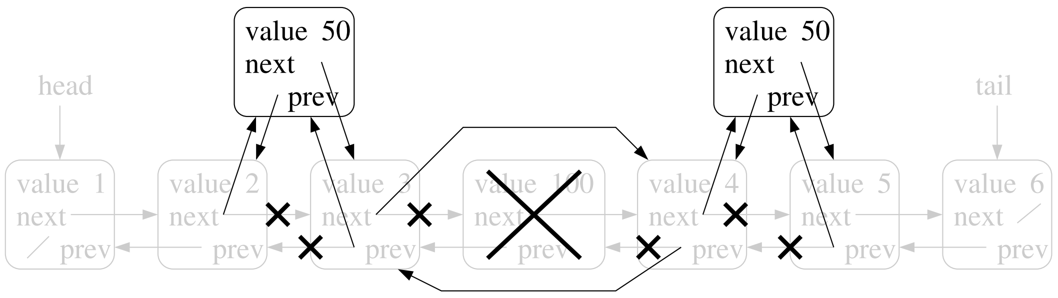
```
class Node {
public:
  Node(int v) :
    value(v),
    next(NULL),
    prev(NULL) {}
  int value;
  Node* next;
  Node* prev;
};
```

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`.

15.1 Diagram [/ 5]

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

Solution:



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.

Solution: We need to handle the case where the node in front of the target node is the head (and/or similarly where the node after the target node is the tail). In this case we must reassign the head (and/or tail) to the newly inserted node:

```
1 100 2 3 4 5 6   =>   50 1 2 50 3 4 5 6
1 2 3 4 5 100 6   =>   1 2 3 4 50 5 6 50
1 100 2           =>   50 1 2 50
```

We need to handle the case where the target Node is the first or last node in the linked list chain. If this is the case, we cannot insert one of the new nodes: *(Note: Alternate definitions for results in these cases are possible.)*

```
100 1 2 3 4   =>   1 50 2 3 4
1 2 3 4 100   =>   1 2 3 50 4
100           =>   <empty list>
```

We should make sure that our solution works when the element is not present in the input list.

```
1 2 3 4 5 6   =>   1 2 3 4 5 6
<empty list>  =>   <empty list>
```

And we could also worry about splitting a node with an odd value...

```
1 2 3 101 4 5 6   =>   1 2 50 3 4 51 5 6
```

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.

Solution:

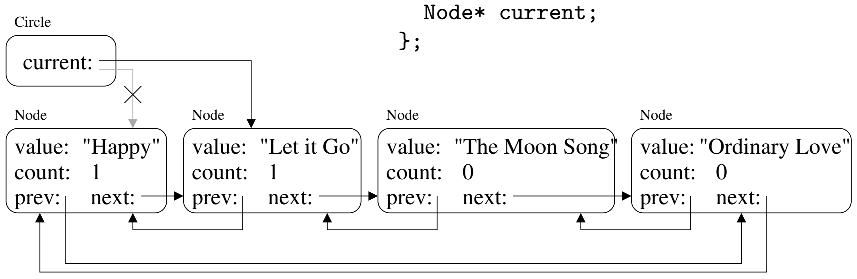
```
void LeapFrogSplit2(Node* &head, Node* &tail, int value) {
  // locate the element
  Node *tmp = head;
  while (tmp != NULL && tmp->value != value) {
    tmp = tmp->next;
  }
  // do nothing if the element was not found
  if (tmp == NULL) return;

  // if there is a previous element to leap backwards over...
  if (tmp->prev != NULL) {
    Node *a = new Node(value/2);
    a->next = tmp->prev;
    a->prev = tmp->prev->prev;
    if (tmp->prev != head) {
      tmp->prev->prev->next = a;
    } else {
      head = a;
    }
    tmp->prev->prev = a;
    tmp->prev->next = tmp->next;
  }

  // if there is a next element to leap forwards over...
  if (tmp->next != NULL) {
    Node *b = new Node(value - value/2);
    b->next = tmp->next->next;
    b->prev = tmp->next;
    if (tmp->next != tail) {
      tmp->next->next->prev = b;
    } else {
      tail = b;
    }
    tmp->next->next = b;
    tmp->next->prev = tmp->prev;
  }


  // reset head & tail if either or both point to the deleted element
  if (head == tmp)
    head = tmp->next;
  if (tail == tmp)
    tail = tmp->prev;
  // clean up the memory
  delete tmp;
}
```

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.

Solution:

```
Circle::Circle(const std::vector<std::string>& data) {
  if (data.size() == 0) {
    // empty input -- create empty play list
    current = NULL;
  } else {
    // create the first node
    current = new Node(data[0]);
    // step through all of the other elements, storing a pointer to the last one.
    Node* tmp = current;
    for (int i = 1; i < data.size(); i++) {
      tmp->next = new Node(data[i]);
      // connect the bidirectional links with the previous node
      tmp->next->prev = tmp;
      tmp = tmp->next;
    }
    // connect the bidirectional links between the first & last nodes
    tmp->next = current;
    current->prev = tmp;
  }
}
```

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.

Solution:

```
bool Circle::remove(const std::string& to_remove) {
  if (current == NULL) return false;
  Node *tmp = current;
  do {
    if (to_remove == tmp->value) {
      if (tmp->next == tmp) {
        // if only one element
        delete tmp;
        current = NULL;
        return true;
      }
      if (current == tmp) {
        // if the head is pointing at the element to be removed
        current = tmp->next;
      }
      // bypass this element in both directions
      tmp->prev->next = tmp->next;
      tmp->next->prev = tmp->prev;
      // cleanup the memory
      delete tmp;
      return true;
    }
    tmp = tmp->next;
  } while (tmp != current);
  return false;
}
```

17 Common Data [/ 20]

Write a templated function `common_data` that takes in two STL `vector`s 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.

Solution:

```
template <class T>
std::vector<T> common_data(const std::vector<T> &a, const std::vector<T> &b) {
  // a local vector variable to store the common elements
  std::vector<T> answer;
  // loop over the first vector
  for (unsigned int i = 0; i < a.size(); i++) {
    bool duplicate = false;
    // check to see if this element is a duplicate of an already
    // processed element in the first vector
    for (unsigned int j = 0; j < i; j++) {
      if (a[i] == a[j]) {
        duplicate = true;
        break;
      }
    }
    if (!duplicate) {
      // loop over the elements in the second vector
      for (unsigned int k = 0; k < b.size(); k++) {
        if (a[i] == b[k]) {
          answer.push_back(a[i]);
          // make sure to break out of this loop so we don't get
          // tricked by a duplicate in the second vector
          break;
        }
      }
    }
  }
  return answer;
}
```

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?

Solution: $O(n*(n+m))$ **or** $O(n^2+nm)$ **– cannot be further simplified without information on the relative sizes of** n **and** m**.**

18 Possessive Grammar [/ 22]

Write a function `convert_to_possessive` that takes in one argument, an STL `list` of `string`s 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.

Solution:

```
void convert_to_possessive (std::list<std::string> &sentence) {
  std::list<std::string>::iterator word = sentence.begin();
  // for each word in the sentence
  while (word != sentence.end()) {
    std::list<std::string>::iterator item = sentence.end();
    std::list<std::string>::iterator owner = sentence.end();
    // check for match to the pattern "the XXXX of XXXX"
    if (*word != "the") { word++; continue; }
    item = word;
    item++;
    if (item == sentence.end()) { word++; continue; }
    owner = item;
    owner++;
    if (owner == sentence.end() || *owner != "of") { word++; continue; }
    owner++;
    if (owner == sentence.end()) { continue; }
    // now make the edits
    word = sentence.erase(word); // erase "the"
    sentence.insert(word,(*owner)+"'s");
    word++;
    word = sentence.erase(word); // erase "of"
    word = sentence.erase(word); // erase owner
  }
}
```

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 $\boxed{\textbf{06}}$. It can be fixed by moving line $\boxed{\textbf{19}}$ immediately after line $\boxed{\textbf{10 or 11}}$. (Note: the line is causing a MEMORY ALREADY FREED error in its current location.) The memory debugger reports use of UNINITIALIZED MEMORY on line $\boxed{\textbf{13 (or 16)}}$. It can be fixed by adding this line of code $\boxed{\texttt{*answer = 0;}}$ immediately after line $\boxed{\textbf{11}}$. A MISMATCHED NEW/NEW[]/DELETE/DELETE[] is reported on line $\boxed{\textbf{17}}$. It is fixed by changing that same line to $\boxed{\texttt{delete [] data;}}$. An INVALID READ is reported on line $\boxed{\textbf{08}}$ and an INVALID WRITE is reported on line $\boxed{\textbf{09}}$. Both errors can be fixed by editing line $\boxed{\textbf{07}}$ to be $\boxed{\texttt{for (int i = 0; i < max\_index-2; i++) \{}}$.

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.

Solution: This program generates the first 20 numbers in the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, and 4181. The program then counts how many of these numbers are odd. The mystery answer = 13.

20 Recursive Order Notation Challenge [/ 13]

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

Solution:

```
int fooA (int n) {
  if (n <= 0) {
    return 0;
  } else {
    return 1 + fooA(n-1);
  }
}
```

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

Solution:

```
int fooB (int n) {
  if (n <= 1) {
    return 0;
  } else {
    return 1 + fooB(n/2);
  }
}
```

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

Solution:

```
int fooC (int n) {
  if (n <= 0) {
    return 1;
  } else {
    return fooC(n-1) + fooC(n-1);
  }
}
```