

# 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 Checking It Thrice [ / 22 ]

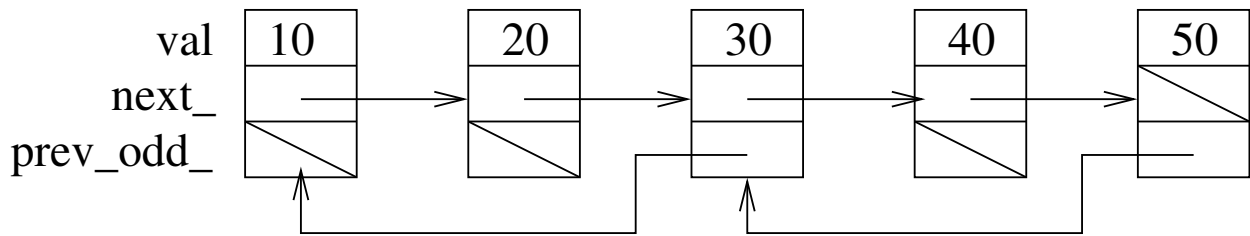
Each row in the table below contains two statements labeled A and B. Put a checkmark  in up to three boxes per row. Each correct checkmark is worth +1, each blank checkbox is worth +0, and to discourage random guessing each incorrect checkmark is worth -2 points. Your score on this problem will not go below 0. Only two A statements are true because of their corresponding B statements.

A is		B is		A is true because of B	Statements
True	False	True	False		
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<p><b>A:</b> STL vector erase() may invalidate an iterator.</p> <p><b>B:</b> For programs that need to do a lot of erasing, you should use an STL list instead of an STL vector.</p>
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<p><b>A:</b> A memory debugger like Valgrind or Dr. Memory can help find leaked memory.</p> <p><b>B:</b> A memory debugger like Valgrind or Dr. Memory shows you the line you should have written delete/delete[] on to fix the memory leak.</p>
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<p><b>A:</b> Vec push_back() on average takes O(n)</p> <p><b>B:</b> Vec push_back() sometimes has to allocate a new array that's 2*m_alloc</p>
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<p><b>A:</b> If std::list&lt;int&gt;::iterator itr points to a valid location in list l1, writing l1.erase(itr)++ may cause a memory error.</p> <p><b>B:</b> Incrementing the end() iterator in any STL list has undefined behavior</p>
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<p><b>A:</b> STL lists / dslist do not have operator[] defined</p> <p><b>B:</b> Using operator[] on a pointer is the same as using pointer arithmetic and then dereferencing the result.</p>
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<p><b>A:</b> Assuming std::vector&lt;int&gt;::iterator itr points to a valid location in vector v1, writing v1.insert(itr,5); std::cout &lt;&lt; *itr; will always cause a memory error.</p> <p><b>B:</b> std::vector&lt;T&gt;::insert() returns an iterator because insert() may invalidate the iterator passed in.</p>
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<p><b>A:</b> If std::vector&lt;int&gt; v1 is an empty vector, v1.end()-- will result in undefined behavior.</p> <p><b>B:</b> Decrementing the end() iterator in any STL vector has undefined behavior.</p>
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<p><b>A:</b> If a recursive function does not use array indexes or pointers, and there's a segmentation fault in that function, it means you must have infinite recursion.</p> <p><b>B:</b> Infinite recursion can result in a segmentation fault.</p>
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<p><b>A:</b> Writing int* x=5; will result a compiler error.</p> <p><b>B:</b> Memory addresses are not numbers, so you cannot store a number in a pointer.</p>
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<p><b>A:</b> Writing dslist&lt;int&gt; x; x.push_back(5); dslist&lt;int&gt; y = x; will not result in calling y.operator=(x) even though there is an = sign in the last statement.</p> <p><b>B:</b> operator= can only be called on objects that are already constructed.</p>

## 2 Hop Lists [ / 37 ]

In this problem we will consider “hop lists”, a made-up data structure based on “skip lists”. A hop list is a linked list using the Node class listed below. For simplicity we will not make it templated and our Nodes will only contain integers. The *next\_* pointer works the same way as in the singly and doubly-linked lists we have studied so far. However, the *prev\_odd\_* pointer is NULL for the 1<sup>st</sup> element in the list, and for all other elements in an odd position (3<sup>rd</sup> in the list, 5<sup>th</sup> in the list, and so on), the *prev\_odd\_* pointer points two elements back. There is a diagram below to illustrate how this would look for a list with five nodes containing the values 10, 20, 30, 40, 50 in that order.

```
class Node{
public:
    Node(int v) : val(v), next_(NULL), prev_odd_(NULL) {}
    Node(int v, Node* next) : val(v), next_(next), prev_odd_(NULL) {}
    int val;
    Node* next_;
    Node* prev_odd_;
};
```



In the following sections you will write two different versions of `AddPrevOddPointers`. Both versions take a Node pointer called *head* which points to the first Node in a hop list, and a Node pointer called *tail*. After the function is done, all of the *prev\_odd\_* pointers should be set correctly, and *tail* should point to the last element (furthest from *head*) that is in an odd position. If there are not at least 3 Nodes in the list, then *tail* should be NULL. You can assume that all Nodes have their *next\_* pointer correctly set before `AddPrevOddPointers` is called.

An example of calling the function might look like:

```
//Start a new list
Node* head = new Node(2561);
/////Code to make the rest of the nodes/link forward omitted

PrintListForward(head); //"Printing forward list..."
Node* tail;
AddPrevOddPointers(head,tail);
PrintHopListBackwards(tail); //"Printing backwards..."
std::cout << std::endl;
```

Here are two examples:

```
Printing forward list: 10 20 30 40 50 60 70 80 90 100 110
Value at tail: 110
Printing backwards every-other list: 110 90 70 50 30 10
```

```
Printing forward list: 31 32 33 34 35 36 37 38 39 301
Value at tail: 39
Printing backwards every-other list: 39 37 35 33 31
```

2.1 AddPrevOddPointers (Recursive Version) [ / 18 ]

Write a version of AddPrevOddPointers that uses recursion.

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

## 2.2 AddPrevOddPointers (Iterative Version) [ / 15 ]

Write a version of `AddPrevOddPointers` that does not use any recursion.

*sample solution: 20 line(s) of code*

## 2.3 AddPrevOddPointers Complexity [ / 4 ]

If the hop list has  $n$  elements, what is the running time order notation of the iterative version of `AddPrevOddPointers`? What about for the recursive version?

### 3 Shape Overlays [ / 14 ]

Given a series of Shape objects in a `vector<Shape> shapes`, the code below should allocate and fill a 2D dynamic array of integers, `overlay` that represents an overlay of all the shapes. If all of the shapes were to be put on top of each other, an overlay would show for every position (x,y) how many shapes had a pixel on at that position. You can assume that `shapes` is not empty and each shape is at least 1x1 in size. The coordinates use the Cartesian coordinate system, where (0,0) is the origin,  $x = 1$  is to the right of the origin, and  $y = 1$  is above the origin. You can also assume non-negative x,y coordinates for everything and that the overlay starts with the bottom left corner at (0,0). The example shapes and overlay output have (0,0) in the bottom left corner for easy visualization.

Shape 1 (outline rectangle)

Printing 4 x 4 overlay:

```
XXXX
X..X
X..X
XXXX
```

Lower left is at (0, 1)

Printing 5 x 5 overlay:

```
13321
12111
12111
13321
01100
```

```
class Point{
public:
    Point(int x=0, int y=0) : m_x(x), m_y(y) {}
    int m_x, m_y;
};
```

```
class Shape{
public:
    bool pixelOn(int x, int y) const;
    void getDimensions(int& width, int& height, Point& corner) const;
    ...
};
```

```
void MakeOverlay(int**& overlay, int& height, const std::vector<Shape>& shapes){
    int shape_height, shape_width, width;
    height = width = 0; //Start by assuming a 0x0 overlay
    Point shape_corner; //Holds the lower left corner of the shape
```

```
//Cycle through each shape and find the further-upper-right point
for(unsigned int i=0; i<shapes.size(); i++){
    shapes[i].getDimensions(shape_width, shape_height, shape_corner);
    height = std::max(height, shape_corner.m_y + shape_height);
    width = std::max(width, shape_corner.m_x + shape_width);
}
```

```
//STUDENT PORTION #1: ALLOCATE OVERLAY, first box on next page goes here
```

```
//STUDENT PORTION #2: FILL IN OVERLAY, second box on next page goes here
```

```
PrintOverlay(overlay, width, height); //DO NOT WRITE THIS FUNCTION
```

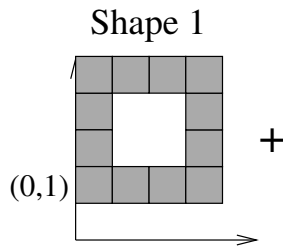
```
}
```

Shape 2 (outline rectangle)

Printing 4 x 4 overlay:

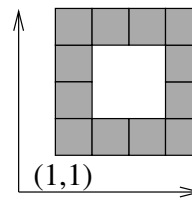
```
XXXX
X..X
X..X
XXXX
```

Lower left is at (1, 1)



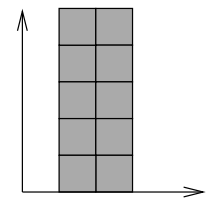
+

Shape 2



+

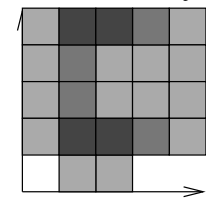
Shape 3



(1,0)

int\*\* overlay

=



(0,0)

### 3.1 #1: Allocate Overlay [ / 8 ]

Write the code that should go under the “STUDENT PORTION #1” comment in MakeOverlay.

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

### 3.2 #2: Fill In Overlay [ / 6 ]

Write the code that should go under the “STUDENT PORTION #2” comment in MakeOverlay.

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

## 4 Pivoting Lists [ / 24 ]

Write a void function `MovePivots` that takes an STL `list<string>` `a` and a constant STL `list<string>` `pivots`. The function should rearrange `a` so that all instances of strings that are in both `a` and `pivots` are at the front of `a`. The order of other elements in `a` should not be changed, and the elements that were moved to the front should be in the order they were found in originally in `a`. The elements in `a` are not guaranteed to be unique. Do not create any new lists/vectors/arrays. Do not use anything from `<algorithm>`. Your function does not need to print anything.

Here are some examples of before and after the function, with five strings in `pivots`: `ant bear cat dog eel`

Before List (size 3): `bob ant cod`

After List (size 3): `ant bob cod`

Before List (size 3): `eel ant cat`

After List (size 3): `eel ant cat`

Before List (size 9): `bob blob cod eel cod ant eel eel ant`

After List (size 9): `eel ant eel eel ant bob blob cod cod`

### 4.1 MovePivots Implementation [ / 20 ]

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

### 4.2 MovePivots Complexity [ / 4 ]

If there are  $p$  strings in `pivots`, and  $w$  strings in `a`, what is the running time order notation for `MovePivots`? Explain your analysis.



## 5 Time Complexity [ / 20]

For each of the functions, write the time complexity assuming there are  $n$  elements stored in the container. If there is a difference between C++98 and C++11, you should assume C++11. For the singly-linked list, assume that we only have the *head\_* pointer and no other member variables. For the singly-linked list, assume that *erase()* is given a pointer to the node we want to erase **and** a pointer to the node before the one we want to erase.

	STL vector or <code>Vec&lt;T&gt;</code>	Singly-linked List	STL list or <code>dslist&lt;T&gt;</code>
<code>size()</code>	<input type="text"/>	<input type="text"/>	<input type="text"/>
<code>push_back()</code>	<input type="text"/>	<input type="text"/>	<input type="text"/>
<code>erase()</code>	<input type="text"/>	<input type="text"/>	<input type="text"/>
<code>insert()</code>	<input type="text"/>	<input type="text"/>	<input type="text"/>
<code>pop_back()</code>	<input type="text"/>	<input type="text"/>	<input type="text"/>

Write 2-3 complete sentences about one of the above methods which is more efficient for STL **lists** than for STL **vectors** and why this is the case.

## 6 Pokémon Battles [ / 16]

Players, known as trainers, collect teams of Pokémon. Trainers then have battles against each other where they use one Pokémon at a time from their team to try and defeat their opponent's team. Each face-off between two Pokémon is considered a "fight". A series of fights between two trainers is called a "battle". When a Pokémon loses a fight, it cannot be used again in the same battle. A battle is not over until one of the trainers has no more usable Pokémon, at which point their opponent wins.

Each monster is represented by an instance of the class `Pokemon`. You do not need to know the details of the class. To determine which Pokémon will win in a fight, the following function is used - it returns a negative number if *p1* wins, and a positive number if *p2* wins. There are no ties.

```
int pokemonFight(const Pokemon & p1, const Pokemon & p2);
```

In this problem you will be completing an implementation for the recursive function `TrainerOneWins()`. The function takes in two `Node` pointers which represent two lists of Pokémon, `trainer1` represents the Pokémon belonging to Trainer 1, and `trainer2` represents the Pokémon belonging to Trainer 2. The function returns `true` if Trainer 1 wins, and `false` if Trainer 2 wins. A trainer wins if they still have usable Pokémon but their opponent does not. If a trainer's Pokémon loses, the trainer will use the next Pokémon in their list.

In this problem, the `Node` class is defined as follows:

```
template <class T> class Node {
public:
    T data;
    Node* next;
};
```

As an example consider the following case. You do not need to know anything about specific Pokémon to solve this problem.

`Node<Pokemon> * list1` has Bulbasuar, Ivysaur, Geodude

`Node<Pokemon> * list2` has Squirtle, Charmander

Running the following code:

```
if(TrainerOneWins(list1,list2)){
    std::cout << "Trainer 1 wins." << std::endl;
}
else{
    std::cout << "Trainer 2 wins." << std::endl;
}
```

A possible run using might look something like this. The output is handled in `pokemonFight()`, you should not write any output statements:

```
Bulbasuar wins against Squirtle
Charmander wins against Bulbasaur
Charmander wins against Ivysaur
Geodude wins against Charmander
Trainer 1 wins.
```

### 6.1 TrainerOneWins Implementation [ /12]

Fill in the blanks to finish *TrainerOneWins()*:

```
bool TrainerOneWins(  trainer1,  trainer2)
{
    if(trainer1 == NULL || trainer2 == NULL)
    {
        return  ;
    }

    int fight_result = pokemonFight(  );

    if(fight_result < 0){
        return  ;
    }
    else{
        return  ;
    }
}
```

### 6.2 TrainerOneWins Complexity [ /4]

If *pokemonFight()* is  $O(1)$ , and there are  $m$  pokemon in Trainer 1's list and  $n$  pokemon in Trainer 2's list, what is the time complexity for *TrainerOneWins()*?

## 7 Print-and-Modify Functions [ / 20]

In this problem you must infer the behavior of two functions from the code sample and output provided below. You should not hard-code any values. *Hint: Write out the relationship between the numbers before and after `AddByPositionAndPrint()`.*

Running this code:

```
int main(){
    std::list<int> counts = std::list<int>(3,0);
    AddOneAndPrint(counts);
    AddOneAndPrint(counts);
    counts.push_back(9);
    counts.push_front(11);
    AddOneAndPrint(counts);
    std::list<int> add_amounts;
    add_amounts.push_back(4); add_amounts.push_back(1); add_amounts.push_back(-3);
    add_amounts.push_back(0); add_amounts.push_back(4);
    AddByPositionAndPrint(counts,add_amounts);
    AddOneAndPrint(add_amounts);
    return 0;
}
```

Produces this output, with one line coming from each non-STL function call:

```
Elements after updates: 1 1 1
Elements after updates: 2 2 2
Elements after updates: 12 3 3 3 10
Elements after updates: 16 4 0 3 14
Elements after updates: 5 2 -2 1 5
```

**Your answers should go in the boxes on the next page.**

### 7.1 AddByPositionAndPrint [ / 14]

Start by implementing *AddByPositionAndPrint*. Assume that the two arguments have the same size.

*sample solution: 10 line(s) of code*

### 7.2 AddOneAndPrint [ / 6]

Now implement *AddOneAndPrint*. Do not write any duplicate code.

*sample solution: 3 line(s) of code*

## 8 Mystery List Function [ / 20]

This problem focuses on a couple similar functions that are used to manipulate a collection of doubly linked nodes:

```
template <class T>
class Node{
public:
    Node(const T& v) : value(v), next(NULL), prev(NULL) {}
    T value;
    Node<T> *next, *prev;
};
```

We also define a function for printing nodes:

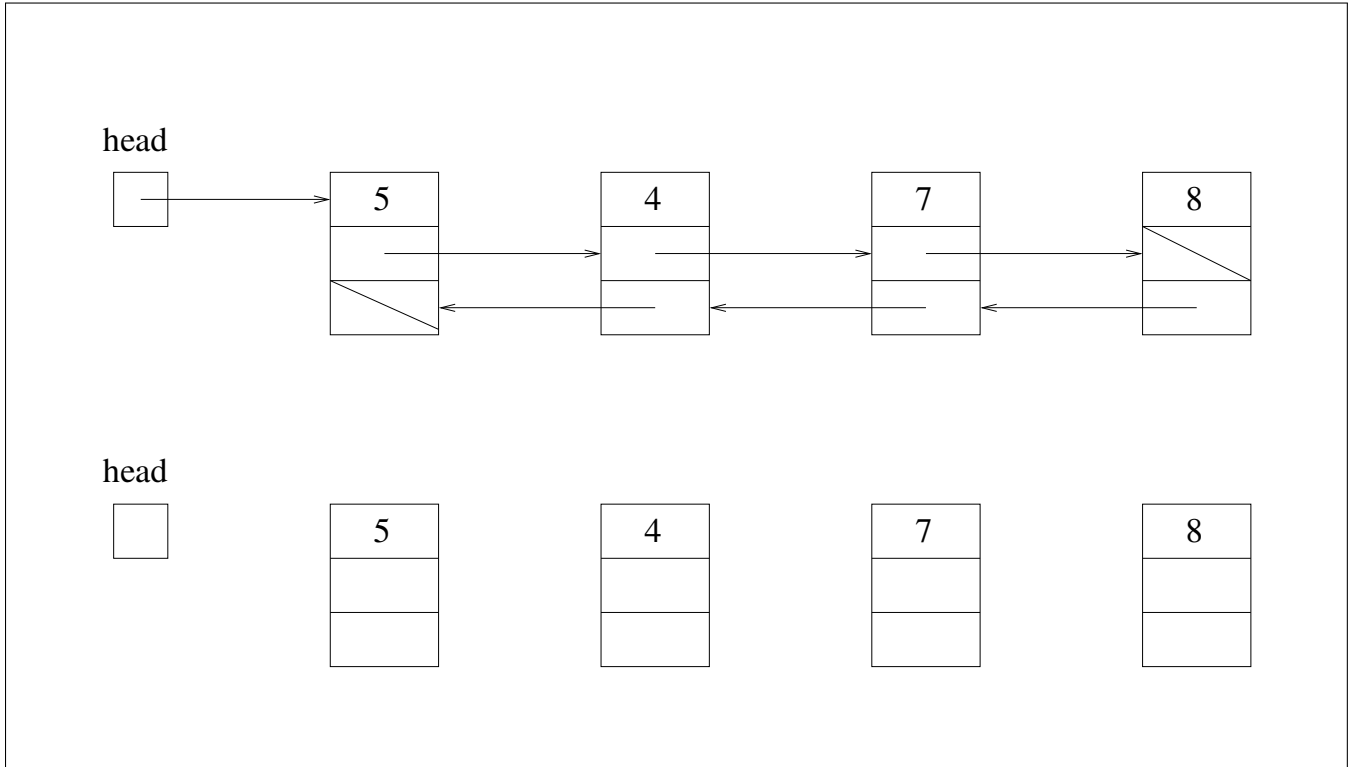
```
template <class T>
void PrintList(Node<T>* head){
    std::cout << "List:";
    while(head){
        std::cout << " " << head->value;
        head = head->next;
    }
    std::cout << std::endl;
}
```

Here is the mystery function you will be working with:

```
1 void MysteryA(Node<int>*& ptr1, Node<int>* ptr2){
2     while(ptr2 && (ptr2->value % 2 == 0 || ptr2->value % 7))
3         ptr2 = ptr2->next;
4     if(ptr2 && ptr2 != ptr1){
5         Node<int>* ptr3 = ptr2->next;
6         Node<int>* ptr4 = ptr2->prev;
7         ptr2->next = ptr1;
8         ptr1->prev = ptr2;
9         ptr1 = ptr2;
10        if(ptr3)
11            ptr3->next = ptr4;
12        if(ptr4)
13            ptr4->prev = ptr3;
14        ptr1->prev = NULL;
15    }
16 }
```

8.1 Visualizing *MysteryA* [ / 14]

In the box below, a list is shown. Below it is a copy of the nodes with none of the pointers added. Update the bottom list to illustrate how the list has changed after a call to *MysteryA(head,head)*.



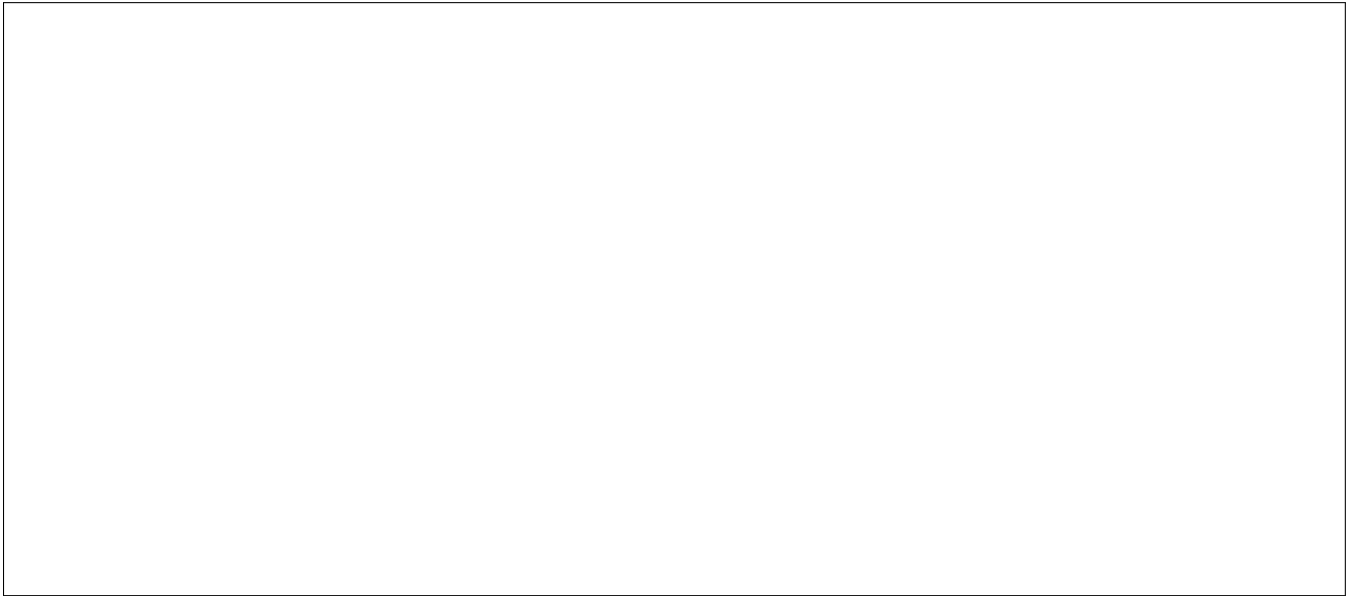
The function is supposed to find the first odd multiple of 7 by using *ptr2* and if it's not already at the head, the function should make *ptr2* the new head by moving it to the front and updating links accordingly. It also has to update the pointer passed in. Assume that *ptr1* is a list head and that *ptr2* belongs to the same list as *ptr1*.

## 8.2 Fixing MysteryA [ / 6]

Consider the original list given in 4.1 and the following code fragment:

```
PrintList(head);  
MysteryA(head,head);  
PrintList(head);
```

Explain what will happen when this code runs. If there are bugs, explain which lines of code need to be changed and what the corrected lines of code should be. Use the line numbers provided in the left margin of the code.





## 9 Acrostic Printing [ / 21]

An “acrostic” is a word that is found by a column reading top-to-bottom when looking at several words stacked on top of each other. For example, if our input is a vector of 4 strings:

```
Had
a
Nice
Day
```

then we could look at the first letter in each word to see that they spell a new word: “HaND”. Similarly, we could look at the second letter in each word and see they spell: “aia”. Your task is to write a function, *acrostics()*, which takes a vector of strings (each string contains exactly one word) and returns an array of C-style strings where each string is one column of the acrostic. For the input shown above, the return array should contain the following strings:

```
HaND
aia
dcy
e
```

You can ignore the null-terminating character, ‘\0’ for simplicity. We have given you a partial implementation, you will need to fill in the rest of it. You should not use any arrays besides *ret*, and you should not call the `delete` keyword. You cannot declare any new vectors/lists in your code. Finally, keep in mind that you should be memory efficient and not allocate more space than you need in the return array.

**Your answer should go in the box on the next page.**

## 9.1 *acrostic* Implementation [ / 16]

```
char** acrostics(const std::vector<std::string>& v){
    unsigned int max_return_length = 0;

    for(unsigned int i=0; i<v.size(); i++){
        max_return_length = std::max(max_return_length, (unsigned int)v[i].size());
    }

    std::vector<int> characters_per_string(max_return_length,0);
    for(unsigned int i=0; i<v.size(); i++){
        for(unsigned int j=0; j<v[i].size(); j++){
            characters_per_string[j]++;
        }
    }
}
```



```
    return ret;
}
```

## 9.2 *acrostic* Performance [ / 5]

If there are  $m$  input strings, the length of the longest input string is  $n$ , and there are  $c$  characters in all input strings combined, what is the time complexity and the space (memory) complexity of *acrostic*?



## 10 Iterating Over a List and Inserting New Points [ /21]

In this question you are asked to write two functions that use iterators to traverse and manipulate STL lists.

Don't worry about `#include` or `#define` statements. You may assume using namespace `std`.

### 10.1 Lists Finding Zeros [ /6]

First, write a function that will be passed a **const** reference to an STL list of ints and returns a new list of ints containing the positions of zeros in the input list. The function should iterate through the list using a list iterator. It should not use `find()` or similar STL functions.

For example, if the original list contained 1 0 11 16 0 0 50 75 85 90 0, the returned list should contain 1 4 5 10.

*sample solution: 12 line(s) of code*

## 10.2 Lists Replacing Zeros[ /15]

Now, write a second function. This **void** function will be passed a reference to an STL list of ints. In this function each zero in the list should be replaced with the sum of the adjacent numbers. A zero in the first or last position of the list should not be replaced. For example, if the list originally contained 1 0 11 16 0 0 50 75 85 90 0, the returned list will contain 1 12 11 16 16 66 50 75 85 90 0. Iterate through the list from left to right and replace the elements sequentially. Notice how consecutive zeros are handled. The first zero is replaced and the replacement value becomes the adjacent value for the next zero. That is, a list containing  $x$  0 0 0  $y$  will become  $x$   $x$   $x$   $x+y$   $y$ , where  $x$  and  $y$  are integers.

The zeros are to be replaced in the original list. Do not make a copy of the list. Iterate through the list and replace the elements. Do not use `std::replace` or `std::find`.

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

## 11 Recursive Lists [ /25]

In this question, don't worry about `#include` or `#define` statements. You may assume using namespace `std`.

### 11.1 Recursive Lists Delete a List[ /6]

A templated class for Nodes is defined as:

```
template <class T>
class Node {
public:
    T value;
    Node<T>* next;
};
```

First, write a templated **recursive** function to delete a list of `Node <T >s`.

*sample solution: 8 line(s) of code*

## 11.2 Recursive Lists Recursive Merge [ /19]

Merging two or more sorted lists is a common operation. It is a basic part of the merge sort which we recently covered in lecture. The idea behind merging two lists is to travel through the two sorted input lists and produce a third *sorted* list containing all of the elements of the two original lists.

For example, if the first list contains *apple cow rhino tree* and the second list contains *cat dog mongoose zebra*, the merged list should contain *apple cat cow dog mongoose rhino tree zebra*.

Write a templated **recursive** function that takes two pointers to sorted singly linked lists of *Nodes*, defined on the previous page, and a reference to a pointer to a singly linked list of *Nodes*. On return from the function, the third list should contain the merged sorted list. Your merged list must copy the data in the sorted lists.

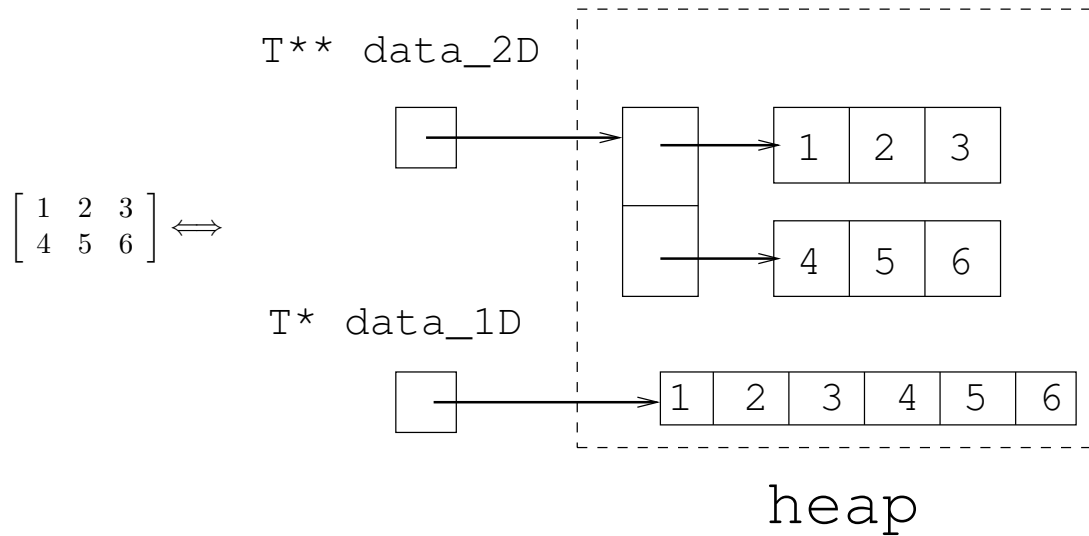
The function must be *recursive*. Do not sort the list. Merge the lists, don't sort. These are not STL lists, use the pointers to iterate through the lists. You may include any helper functions that you find necessary. Do not edit the Node class. You don't have to write a main() function.

*sample solution: 20 line(s) of code*

## 12 Templating and Flattened Matrices [ /19]

In Homework 3 we explored a matrix based around a 2D data structure. We will not be using the *Matrix* class we designed, but instead will be using a templated  $T^{**}$  `data_2D` data structure to represent a matrix. The layout should look familiar.

In addition to the 2D representation, we would like to have the option to work on a “flattened” version of the matrix, which is implemented in a  $T^*$  `data_1D` structure. Both structures are depicted below. We would like to be able to go between the two data structures.



Below is an example of how the two functions we will write might be used, along with the output from this code fragment.

```
int** data_2D = new int*[2];
data_2D[0] = new int[3];
data_2D[1] = new int[3];
data_2D[0][0] = 1; data_2D[0][1] = 2;
data_2D[0][2] = 3; data_2D[1][0] = 4;
data_2D[1][1] = 5; data_2D[1][2] = 6;

int* data_1D = Flatten(data_2D,2,3);

for(int i=0; i<2; i++){
    for(int j=0; j<3; j++){
        int index;
        int readval = Read1D(data_1D,2,3,i,j,index,-1);
        std::cout << "(" << index << "," << readval << ") ";
    }
}

//Assume and Delete1D/2D are already written
Delete2D(data_2D,2,3);
Delete1D(data_1D);
```

Output:

```
(0,1) (1,2) (2,3) (3,4) (4,5) (5,6)
```

## 12.1 Flattening the Matrix [ /10]

Your first task is to implement the templated *flatten* function. *flatten* should take in `data_2D` as shown on the previous page, and two integers `m` and `n`, which are the number of rows and the number of columns in the data respectively. *flatten* should return a pointer to an equivalent 1D data structure.

If either the number of rows or columns is non-positive, the function should return a NULL pointer.

Do not change `data_2D`. **Only** use `const` and `&` when needed. Remember that since *flatten* is templated it should work with **any** datatype. Do not leak any memory, any memory still allocated must be reachable from the returned pointer.

*sample solution: 14 line(s) of code*

If the 2D matrix representation contains  $m$  rows and  $n$  columns, what is the running time of the *flatten* function?



## 12.2 Reading From the Flattened Matrix [ /9]

Another important task is to be able to read from the data structure. Write a function *Read1D* that takes in a 1D representation of data `data_1D`, two integers `m` and `n` which are the number of rows and columns respectively, two integers `row` and `col` which are the row and column position we are trying to extract data from, a reference to an integer `index`, and a `failure_value` which will be returned in case of an error.

Just like in Homework 3, we will number the upper left corner of a 2D structure as (0,0) or `row= 0, col= 0`.

The function should do two things. If the dimensions are legal (i.e. there are a positive number of rows and columns), and the requested position can exist within the given bounds, then the function should return the data stored at that position and set `index` to the index in `data_1D` where the data came from. If the dimensions are illegal or the requested position is out of bounds, the `index` should be set to -1 and `failure_value` should be returned.

Keep in mind that the same `data_1D` object can be viewed different ways. For example, if there is a  $2 \times 3$  `data_2D_example` and `data_1D_example = flatten(data_2D_example,...)`, then after calling `Read1D(data_1D_example,1,6,1,1,index,-1)`, `index` will be -1, because in this example call we specified that `m= 1, n= 6`, and there is no position (1, 1) inside of a  $1 \times 6$  matrix.

On the other hand, using the same `data_1D_example`, `Read1D(data_1D_example,2,3,1,1,index,-1)` would set `index= 4`.

Do not call any STL functions. **Only** use `const` and `&` when needed. Remember that since *Read1D* is templated it should work with **any** datatype.

*sample solution: 11 line(s) of code*

### 13 Memory Errors [ /9]

For each function or pair of functions below, choose the letter that best describes the memory error that you would find. You can assume using namespace std and any necessary #include statements.

- A ) use of uninitialized memory                      C ) memory leak                      E ) no memory error  
B ) mismatched new/delete/delete[]                  D ) already freed memory              F ) invalid write

```
char* a = new char[6];  
a[0] = 'B'; a[1] = 'y';  
a[2] = 'e'; a[3] = '\0';  
cout << a << endl;  
delete a;
```

```
int a[2];  
float** b = new float*[2];  
b[0] = new float[1];  
a[0] = 5; a[1] = 2;  
b[0][0] = a[0]*a[1];  
delete [] b[0];  
b[0] = new float[0];  
delete [] b;
```

```
int a[10];  
int b[5];  
for(int i=10; i>5; i--){  
    a[((i-6)*2+1)] = i*2;  
    a[((i-6)*2)] = b[i-6];  
    cout << a[(i-6)*2] << endl;  
}
```

```
string* str1 = new string;  
string* str2;  
string* str3 = new string;  
*str1 = "Hello";  
str2 = str1;  
*str3 = *str1;  
delete str1;  
delete str3;  
delete str2;
```

```
bool* is_even = new bool[10];  
for(int i=0; i<=10; i++){  
    is_even[i] = ((i%2)==0);  
}  
delete [] is_even;
```

```
int x[3];  
int* y = new int[3];  
for (int i=3; i>=1; i--){  
    y[i-1] = i*i;  
    x[i-1] = y[i-1]*y[i-1];  
}  
delete [] y;
```

## 14 Complexity Code Writing [ / 9 ]

For each of the problems below, write a function `void complexity(int n)` that satisfies the big- $O$  running time. Assume that the input is size  $n$ . You should not use anything that requires a `#include` statement. You should write no more than 7 lines of code per box (including the function prototype).

$O(1)$ :

```
void complexity(int n){
```

*sample solution: 1 line(s) of code*

```
}
```

$O(n^2)$ :

```
void complexity(int n){
```

*sample solution: 5 line(s) of code*

```
}
```

$O(\log n)$

For this one, do not use any loops, do not use math functions such as `log()` or `log2()`:

```
void complexity(int n){
```

*sample solution: 4 line(s) of code*

```
}
```