**1) What are the two components that are most commonly found in a node struct of a linked list?**

(a) Some ability to hold data

(b) A link to the next node which is usually a pointer to the linked list struct.

**2) What role does the head pointer play?**

The head pointer is an external pointer that anchors the start of the linked list. We must always keep this pointer at the front so that we do not "lose" the list and end up with inaccessible nodes.

**3) What value do we give a pointer to indicate that it points to nothing?**

We use NULL (from C) or nullptr (from C++) to indicate a pointer that points to nothing. Both of these are defined as value 0, so a pointer with address 0 points to nothing.

**4) Explain why a linked list structure may be better than an array structure when we are inserting something in the middle of the list.**

If we want to insert something in the middle of a List, then we need to make room for it. If the implementation is an array, then inserting something in the middle involves copying (moving) all the items from the insert location through the end of the array down one spot so we can make room for the inserted item.

If the implementation of the List is a linked list, then an insert might be easier. We need a pointer to the link **before** the insert point. We create a new item and insert it into that location, linking the remaining items in the list after the new item. This does not involve "moving" any data. But we do need to traverse the list from the front to the insert point whereas in an array we can "jump" straight to the insert point.

If the data is small (like an int) then the copy/move process isn't too bad. If the data is big (like a giant array or structure) then the copy/move process can become time consuming.

**5) Explain why a linked list structure might use more memory than an array structure when implementing a List.**

Each node in a linked list has two parts: the data and the pointer. That adds an additional 8 bytes (for the pointer) to each item in the list. So a linked list requires more memory than an array of the same size because the array only needs to store the values and not the pointers.

**6) Explain why a linked list structure might use less memory than an array structure when implementing a List.**

However, an array is a fixed sized structure. If we have an array of 100 spots but only use 4 of them, then the array still needs enough memory allocated to hold 100 items. A linked list would only need to allocate the memory required for 4 spots. So if the array is mostly empty, then the linked list implementation might actually be using less memory space.

**7) Assume you have a linked list List structure using the following struct. Assume the list is already built and contains 10 items. Show how you can insert a link at the 4th position in the structure with the value 100.**

struct Node {

int value;

Node\* next;

};

Node\* head;

// code not shown to build the link lists with 10 items

// you sketch the steps below (or write code) to insert a new item with value 100 in the 4th spot in the List.

1) We must create a pointer and set it at the head. We call this the traversal pointer.

Node\* traversal = head;

2) We then use this traversal pointer to traverse the linked list to the spot right before the insert location. In this case, the insert happens in the fourth spot, so we traverse to the third node in the list (two after the head).

for (int i = 0; i < 2; i++)

traversal = traversal -> next;

3) We use another pointer to create a new struct and add the value 100 in that new struct.

Node\* toInsert = new Node;

toInsert -> value = 100;

4) We now set the next link of the new struct to point to the rest of the chain (after the insert point) by using the traversal pointer's next field.

toInsert -> next = traversal -> next;

5) Finally we set the traversal pointer's next field to point to the new item.

traversal -> next = toInsert;

Thus it is now fully inserted into the linked list.

1. Assume you have your complete List class from Project 10 using a dynamically allocated array. The class declaration (with one extra method declaration) is shown below.

#define DEFAULT\_LIST\_SIZE 10

template <typename T>

class List

{

public:

List ( );

List ( const List<T> &l );

~List ( );

List<T> operator= ( const List<T> &l );

void append ( T data );

void insert ( T data, int position );

int length ( ) const;

T & operator[] ( int position );

void remove ( int position );

bool isEmpty ( ) const;

List<T> operator+ ( const List<T> &l ) const;

List<T> operator\* ( const List<T> &l ) const;

void clear ( );

friend ostream & operator<< (ostream &os, const List<T> &list);

private:

// the maximum capacity of the array storing the list

int capacity;

// the current number of items in the list

int size;

// the dynamically allocated array storing the list

T \*list;

// make the array twice as big to hold more data

void reallocate ( );

};

You will add a new interleave operator by overloading \*. This operator will interleave the two lists into a new third list. For example:

if these two lists have these values currently:

list1 = [1, 2, 3]

list2 = [10, 20, 30, 40, 50]

then a call like this:

list3 = list1 \* list2;

will fill in list3 with these values: list3 = [1, 10, 2, 20, 3, 30, 40, 50]

The interleave operation alternately takes items from list1 and list2 as they are added to list3. This alternate selection continues while there are still elements left in both lists. When one list is exhausted, then we simply add all the remaining elements from the other list (as shown in the example). This is very similar to operator+ except that the values are interleaved instead of operator+ which places all values of list1 first and then all values from list2 second.

You are permitted to call any of the other List public methods in the List ADT in your new function.

Write the code for operator\* below (on the following page).

template <class T>

List<T> List<T>::operator\* ( const List<T> &mylist ) const { }

2. Is it more efficient to add new items at the beginning or end of an array? Is it more efficient to add new items at the beginning or end of a linked list?

3. Consider the List class using a linked list that you are writing for Project 11. Part of the class declaration is shown below. On the next page, you are to write a new-method called isSorted () which returns true if the list is in sorted order from smallest to biggest and false otherwise. You can assume the template type T has operators for > and for < so that you can compare two items appropriately. An empty list or a list with one item is considered sorted.

template <class T>

class List

{

private:

struct Node

{

T item;

Node \* next;

};

Node \*head; // head pointer to linked list

public:

... more code not shown

}

Here is the method you are to complete:

template <class T>

bool List<T>::isSorted() const { }