Lab Sheet 4

"The best way out is always through."
-- Robert Frost

.

1. Write a program to implement a *Linked List* data structure to store *integer* numbers. Name of the class must be `List_Int`.

   A *node* in the list must be represented by the class `Node`. It should have a *data* part (for storing integer numbers) and a *link* part (to point to the next node in the *linked list*). Details of the Node class are given below:

   Data members:

Name of data type	Description
`data`	Holds the data part (integer) of the node.
`next`	Pointer to the next node in the list.

   Constructors:

Constructor	Description
`Node()`	Set `data` to 0 and `next` to `NULL`.
`Node(int d);`	Set `data` to `d` and `next` to `NULL`.

   Member functions:

Function name	Description
`int get_data() const;`	To get the data part of the node.
`Node* get_next_ptr() const;`	To get the pointer to the successor node in the list.
`void set_data(int d);`	To set the data part of the node.
`void set_next_ptr(Node *ptr);`	To set the pointer to the successor node in the list.

   Friend functions/classes:

Function/Class name	Description
`friend class Iterator_Node;`	Make the `Iterator_Node` class a friend of `Node` class so that it can access the data and pointer part of the node directly. (Do you think this is a wrong design decision?)

   Create a class by the name `Iterator_Node` to represent an *iterator* to a node in the list (equivalent to a pointer to Node). You may design the `Iterator_Node` class as follows:

   Data members:

Name of data type	Description
`node_ptr`	To store the pointer to a `Node` object.

   Constructors:

Constructor	Description
`Iterator_Node (Node* p)`	To create an `Iterator_Node` object that points to a specific `Node` object.

   Member functions:

Function name	Description
`Node * get_node_ptr();`	To get the address of the node to which the iterator object is pointing.
`int& operator*() const int& operator*() const;`	Overload the *de-reference* operator to get the data part of the node.
`int* operator->() const int* operator->() const`	Overload the *arrow* operator (const and non-const version).
`bool operator== (const Iterator_Node&); bool operator!= (const Iterator_Node&);`	To compare two iterators.
`Node& operator++(); Node operator++(int);`	To advance the iterator to the next node in the list (prefix and postfix version).

Do we have to overload the decrement (--) operator? What is the use of overloading the arrow (->) operator if the data part of a node is an integer type? Why do you need `const` and `non-const` versions of the *dereference* (*) and *arrow* (->) operators? Why didn't we have a *default* constructor for this class?

The `List_Int` class should be designed as follows:

Data types:

Name of data type	Description
`typedef size_t size_type;`	Type for storing the size of the list (make it equivalent to `size_t`.
`typedef Iterator_Node iterator;` `typedef const Iterator_Node const_iterator;`	Constant and non-constant versions of iterator to a node. (equivalent to a 'safe' pointer to a node).

Data members:

Name of data type	Description
`head`	Pointer to the first node in the list. NULL if empty.
`tail`	Pointer to the last node in the list. NULL if empty.
`list_size`	To hold the number of elements in the list.

Constructors:

Constructor	Description
`explicit List_Int();`	*Default constructor*: to create an empty list. Make the head and tail point to NULL. Size of the list should be initialized to 0.
`explicit List_Int (size_type n, const int& value = 0);`	*Repetitive sequence constructor*: initializes the container object with its content set to repetition, *n* times, of copies of *value*. Example: List_Int(4, 100) will create a list containing 4 nodes, with value 100 in each of them. If the second argument is missing, the value is taken to be 0.
`explicit List_Int (const_iterator first, const_iterator last);`	*Iteration constructor*: Iterates between *first* and *last*, setting a copy of each of the sequence of elements as the content of the container object. Example: List_Int lst1(4, 100); List_Int lst2(l1.begin( ), l1.end( )); will create lst2 as a copy of lst1 (by traversing lst1 from begin( ) to end( )).
`List_Int (const List_Int& x);`	*Copy constructor*: the object is initialized to have the same contents and properties as the x list object.

Member functions:

Function name	Description
`void push_back(const int &);`	To insert an element to the end of the list.
void push_front(const int &d);	To insert an element to the front of the list.
void pop_back();	Removes the last element in the list container, effectively reducing the list size by one.
void pop_front();	Removes the first element in the list container, effectively reducing the list size by one.
bool empty() const;	Returns true if the list is empty and false otherwise.
friend ostream& operator<< (ostream& out, const Stack_Int& s);	Overload the output operator (<<) to print the contents of the list.
size_type size() const ;	Returns the size of the list.
iterator begin( ); const_iterator begin( ) const;	Returns the iterator to the first element of the list.
iterator end( ); const_iterator end( ) const;	Returns an iterator referring to the past-the-end element in the list container.
void clear();	Clear the list. All the elements in the list container are dropped: their destructors are called, and then they are removed from the list container, leaving it with a size of 0.
List_Int::iterator List_Int::insert (List_Int::iterator position,	The list container is extended by inserting new elements before the element at 'position'.

const int& x);	
void swap(List_Int& lst);	Swap content. Exchanges the content of the container by the content of argument lst, which is another list object containing elements of the same type. Sizes may differ. After the call to this member function, the elements in this container are those which were in lst before the call, and the elements of lst are those which were in this. All iterators, references and pointers remain valid for the swapped objects.
iterator erase (iterator position); iterator erase (iterator first, iterator last);	Removes from the list container either a single element (position) or a range of elements ([first,last)).
void remove (const int& value);	Removes from the list all the elements with a specific value. This calls the destructor of these objects and reduces the list size by the amount of elements removed. Unlike member function List_Int::erase, which erases elements by their position (*iterator*), this function removes elements by their *value*.
void unique ( );	Remove duplicate values: removes all but the first element from every consecutive group of equal elements in the list container. Notice that an element is only removed from the list if it is equal to the element immediately preceding it. Thus, this function is especially useful for sorted lists.
void splice (iterator position, List_Int& x); void splice (iterator position, List_Int& x, iterator i); void splice (iterator position, List_Int& x, iterator first, iterator last);	Move elements from list to list: moves elements from list x into the list container at the specified position, effectively inserting the specified elements into the container and removing them from x. *position*: the position within this container where the elements of x are inserted. *x*: the List_Int object from which the elements will be removed. *i*: the iterator to an element in x. Only this single element is removed. *first*, *last*: Iterators specifying a range of elements in x. Moves the elements in the range [*first*, *last*) to *position*. The range includes all the elements between first and last, including the element pointed by *first*, but not the one pointed by *last*. This increases the container size by the amount of elements inserted, and reduces the size of x by the same amount. The operation does not involve the construction or destruction of any element object.

2. Convert the `List_Int` class to represent a *doubly linked list*. The name of the class should be `List_double_ptr_Int`. The interface of the class remains the same.

   Add the following extra member functions in the class:

void sort ( );	Sort elements in container: sorts the elements in the container from lower to higher. The comparisons are performed using the operator< between the elements being compared. The entire operation does not involve the *construction*, *destruction* or *copy* of any element object. They are moved within the container by updating the node links.
void reverse ( );	Reverses the order of the elements in the list container. All iterators and references to elements remain valid.

3. Convert the above `List_double_ptr_Int` class to a *template* class that can store any kind of data types and not just *integers*. The name of the class should be `List`. The *interface* of the class remains the same.