```
// link1.cpp

// to illustrate a linked list
        #include <iostream>
        using namespace std;
        int main()
        {
        typedef float componentType;

        struct NodeType
        {
        componentType   component;
        NodeType *   link;
        };

typedef  NodeType*  NodePtr;

NodePtr  head;
NodePtr  currentPtr;
NodePtr newNodePtr;

        head = new NodeType; // create the first node and store the address in
head
        head->component = 12.8; // value of component in the first node
        newNodePtr = new NodeType; // create another node and store the
address in
                                                                // newNodePtr
        newNodePtr->component = 45.2 ; // value of the next component
        head->link = newNodePtr; // the first node will contain the address of the
                                                                // second node
        currentPtr = newNodePtr ; //save the address of the second pointer
        newNodePtr = new NodeType ; // create another node and store the
address in
                                                                // newNodePtr
        newNodePtr->component = 70.1; // value of the next component
        currentPtr->link = newNodePtr ;// the second node will contain the address
of the
                                                                // third node
```

```
        newNodePtr->link = NULL ;// the third node will contain the NULL;

        // print out link list
        currentPtr = head ; // point to the beginning of the list

        while (currentPtr != NULL)
        {
                cout << currentPtr-> component << endl;
                currentPtr = currentPtr->link ; // point to the next component
        }

}
```

output

12.8
45.2
70.1

```
//*************************************************************************
// link2.cpp
// This program will
//create a dynamic linked list of integers, PLUS it goes on
// to print out the resulting list
// Assumption: The user types in at least one number
//*************************************************************************
#include <iostream>
#include <cstddef>   // For NULL

using namespace std;

typedef int ComponentType;

struct  NodeType;             // Forward declaration
typedef NodeType* NodePtr;

struct NodeType
{
   ComponentType component;
   NodePtr       link;
```

```
};

int main()
{
   NodePtr       head;           // External pointer to list
   NodePtr       newNodePtr;     // Pointer to newest node
   NodePtr       currPtr;        // Pointer to last node
   ComponentType inputVal;

   head = new NodeType;
   cout << "Enter an integer: ";
   cin >> head->component;
   currPtr = head;

   cout << "Enter an integer (or EOF to quit): ";
   cin >> inputVal;
   while (cin)
   {
      newNodePtr = new NodeType;         // Create new node
      newNodePtr->component = inputVal;  // Set its component value
      currPtr->link = newNodePtr;        // Link node into list
      currPtr = newNodePtr;              // Set currPtr to last node
      cout << "Enter an integer (or EOF to quit): ";
      cin >> inputVal;
   }
   currPtr->link = NULL;                 // Mark end of list

   // Print out the resulting linked list

   cout << endl << "Resulting list is:" << endl;
   NodePtr ptr = head;
   while (ptr != NULL)
   {
      cout << ptr->component << ' ';
      ptr = ptr->link;
   }
   cout << endl;
   return 0;
```

```
}
output
Enter an integer: 6
Enter an integer (or EOF to quit): 3
Enter an integer (or EOF to quit): 5
Enter an integer (or EOF to quit): 6
Enter an integer (or EOF to quit): 7
Enter an integer (or EOF to quit):
Resulting list is:
6 3 5 6 7
//link 3

#ifndef SORTEDLIST_H_
#define SORTEDLIST_H_


#include <iostream>
#include <iomanip>
using namespace std;

typedef int ItemType;

struct  NodeType
{
        ItemType  item; // Data
        NodeType* link; // Link to next node in list
};

typedef NodeType* NodePtr;

class  SortedList
{
public :
        bool IsEmpty()const;
        void Print()const;
        void InsertTop(/* in */ ItemType item);
        void Insert(/* in */  ItemType  item);
        void DeleteTop(/* out  */ ItemType& item);
```

```
        void Delete(/* in */ ItemType item);

        SortedList();                                          // Constructor
        ~SortedList();                                         // Destructor
        SortedList(const SortedList&  otherList); // Copy-constructor

private :
        NodeType* head;
} ;


#endif
```

```
SortedList::SortedList() // Constructor
// Post: head == NULL
{
        head = NULL;
}


SortedList::~SortedList() // Destructor
// Post:  All linked nodes deallocated
{
        ItemType temp;

        // Keep deleting top node until list is empty.
        while(!IsEmpty())
        {
                DeleteTop(temp);
        }
}

bool SortedList::IsEmpty()const
```

```
// Post: Returns true IF head == NULL, ELSE false
{
        return head == NULL;
}

void SortedList::Print()const
// Prints out the linked list.
{
        NodePtr currPtr;

        // Points to the beginning of the list.
        currPtr = head ;

        while(currPtr != NULL)
        {
                cout << currPtr-> item << endl;

                // Points to the next component.
                currPtr = currPtr->link;
        }
}


void SortedList::Insert(/* in */ ItemType item)
// Pre:  Item is assigned
//              &&  list components are stored in ascending order.
// Post: New node containing item is in its proper place && list components in
//      ascending order
{
        NodePtr currPtr;
        NodePtr prevPtr;
        NodePtr newNodePtr;

    newNodePtr = new NodeType;

        newNodePtr->item = item;

        prevPtr = NULL;
```

```
        currPtr = head;

        while(currPtr != NULL && item > currPtr->item)
        {
                // Advance both pointers.
                prevPtr = currPtr;
                currPtr = currPtr->link;
        }

        // Insert new node here.
        newNodePtr->link = currPtr;

        if(prevPtr == NULL)
        {
                head = newNodePtr;
        }
        else
        {
                prevPtr->link = newNodePtr;
        }
}


void SortedList::DeleteTop(/* out */ ItemType& item)
// Pre:  List is not empty
//              && list elements in ascending order.
// Post: Item == element of first list node @ entry
//              &&  node containing item is no longer in linked list
//      &&  list elements in ascending order
{
        try
        {
                if(IsEmpty())
                {
                        throw string("\nThe list is empty! - unable to delete the first
node!\n\n");
                }
```

```
                NodePtr tempPtr = head;

                // Obtain item and advance head
                item = head->item;

                head = head->link;

                delete  tempPtr;
        }
        catch(string &emptyList)
        {
                cout << emptyList;
        }
}

void SortedList::Delete(/* in */ ItemType item)
// Pre:  list is not empty
//              && list elements in ascending order
//         &&  item == component member of some list node
// Post: item == element of first list node @ entry
//      &&  node containing first occurrence of item is no longer in linked list
//              &&  list elements in ascending order
{
        NodePtr delPtr;
        NodePtr currPtr;

        bool found = false;

        //  Is item in first node?
        if(item == head->item)
        {
                // If so, delete first node.
                delPtr = head;
                head = head->link;
        }
        else
        {
```

```
		currPtr = head;

		// TRY to delete node from linked list, THROW exception if value is
not
		// in the list, CATCH invalid value and prompt user to enter valid
value.
		try
		{
			// Searches for item in the rest of list.
			while(currPtr->item != item && !found && currPtr->link != NULL)
			{
				if(currPtr->link->item == item)
				{
					currPtr->link = currPtr->link->link;

					found = true;
				}
				else
				{
					currPtr = currPtr->link;
				}

			}

			if(!found)
			{
				throw item;
			}
		}
		catch(ItemType newItem)
		{
			cout << endl << item << " is not a value in the list - "
				<< "please enter a valid value!\n\n";

			cout << "New value to delete: ";
			cin  >> newItem;
			cin.ignore();
```

```
                    Delete(newItem);
                }
        }

        delete delPtr;
}

/*****************************************************************************
*******
 * LINK 3
 * ---------------------------------------------------------------------------
 * This program uses a SortedList class to manipulate the nodes in a linked list.
 *
 * ---------------------------------------------------------------------------
 * INPUT:
 *       newItem - Item that is in the list that allows the user to recover from
 *                          entering an invalid value when deleting from the list.
 *
 * OUTPUT:
 *   Linked list every time a node is inserted or deleted.

*****************************************************************************
*****/

void PrintHeader(string labName, char labType, int labNum);

int main()
{
        PrintHeader("Link 3", 'E', 2);

        SortedList list;
        ItemType mainItem;

        list.Insert(352);
        list.Insert(48);
        list.Insert(12);
        list.Print();
```

```
        if(!list.IsEmpty())
        {
                list.DeleteTop(mainItem);  // delete the first node
                cout << "node delete was " << mainItem << endl << endl;
        }

        cout << "\nprint out list after delete" << endl;
        list.Print();

        list.Insert(1);  // insert at the top of the list
        list.Insert(500); //insert at the bottom of the list
        list.Insert(77); // insert in the middle
        cout << "\nprint the list after inserting nodes"<< endl;
        list.Print();

        list.Delete(48);  // delete in the middle
        cout << "\nprint the list deleting a middle node"<< endl;
        list.Print();

        list.Delete(1);  // delete the first node
        cout << "\nprint the list deleting the first node" << endl;
        list.Print();

        list.Delete(500);  // delete the last node
        cout << "\nprint the list deleting the last node" << endl;
        list.Print();

        cout << "\nAttempting to delete a nonexistent value" << endl;
        list.Delete(1);

        cout << "\nprint the list after deleting a valid node" << endl;
        list.Print();
}

/*************************************************************************
*******
 * PrintHeader
 * ------------------------------------------------------------------------
```

```
 * This function prints the project header.
 * -----------------------------------------------------------------------
 * PRE-CONDITIONS:
 *     labName - Lab Name has to be preciously defined
 *     labType -  Lab Type has to be preciously defined
 *     labNum  - Lab Number has to be preciously defined
 *
 * POST-CONDITIONS:
 *     This function will print the class heading.

*************************************************************************
*****/
void PrintHeader(string labName, char labType, int labNum)
{
        cout << left;

        cout << right;
}
```

Output

```
2
48
352
node delete was 12


print out list after delete
48
352

print the list after inserting nodes
1
48
77
352
500

print the list deleting a middle node
1
```

```
77
352
500

print the list deleting the first node
77
352
500

print the list deleting the last node
77
352

Attempting to delete a nonexistent value

1 is not a value in the list - please enter a valid value!

New value to delete:
```

//link4

```
#ifndef LINK4_H_
#define LINK4_H_
#include <iostream>
#include <iomanip>
using namespace std;

template<typename T>
struct  NodeType
{
        T          item ; // Data
        NodeType<T> *link ;        // Link to the next node in the list.
} ;

template<typename T>
class SortedList
{
```

```
public :
        int  IsEmpty()const;
        void Print();
        void InsertTop(/* in */ T item);
        void Insert(/* in */ T item);
        void DeleteTop (/* out */ T &item);
        void Delete(/* in */ T item);
        void RevPrint (NodeType<T> *head);
        SortedList();
        ~SortedList();
        SortedList(const SortedList &otherList);

private :
        NodeType<T> *head;
};

template<typename T>
void SortedList<T>::RevPrint(NodeType<T>* head)
// Pre:  Head points to an element of a list.
// Post: All elements of list pointed to by head have been printed
//       out in reverse order using recursion.
{
        // IF there are still nodes in the list, perform recursive call until end
        // of list is reached and print values in reverse order once end is reached.
    if(head != NULL)
        {
                RevPrint(head->link);

                cout << head->item << endl;
        }
}

template<typename T>
SortedList<T>::SortedList()
// Post: head == NULL
{
        head = NULL;
}
```

```
template<typename T>
SortedList<T>::~SortedList()
// Post: All linked nodes deallocated
{
	T temp; // Value of any type.

	// Delete the first element until the list is empty.
	while(!IsEmpty())
	{
		DeleteTop(temp);
	}
}

template<typename T>
int SortedList<T>::IsEmpty ( ) const
// Post: Returns true if head == NULL, else returns false.
{
	return head == NULL;
}

template<typename T>
void SortedList<T>::Print()
// Prints out the linked list in reverse order using recursion.
{
	// IF the list is not empty, calls the recursive RevPrint function, ELSE
	// prints a message indicating that the list is empty.
	if(head != NULL)
	{
		RevPrint(head);
	}
	else
	{
		cout << "There is nothing to print\n";
	}
}

template<typename T>
```

```
void  SortedList<T>::Insert(/* in */ T item)
// Pre:  Item is assigned && list components in ascending order
// Post: New node containing item is inserted in its proper place
//               && list components in ascending order
{
        NodeType<T> *currPtr;    // Pointer to current node being accessed.
        NodeType<T> *prevPtr;    // Pointer to node before currPtr.
        NodeType<T> *newNodePtr; // Pointer to a new node.

        // Creates a new node.
        newNodePtr = new  NodeType<T>;

        // Assigns the value in item to the item data member of newNodePtr.
        newNodePtr->item = item;

        prevPtr = NULL;

        // Starts currPtr at the front of the list.
        currPtr = head;

        // Traverses the list to find the proper place to insert the value.
        while(currPtr != NULL && item > currPtr->item)
        {       prevPtr = currPtr;
                currPtr = currPtr->link;
        }

        // Inserts the new node into the list by connecting newNodePtr to currPtr
        // since the new node will be the new next greater value than currPtr.
        newNodePtr->link = currPtr;

        // IF the new node is the new smallest value, makes head point to it, ELSE
        // connects the node pointed to by prevPtr to the new node.
        if(prevPtr == NULL)
        {
                head = newNodePtr;
        }
        else
        {
```

```
                prevPtr->link = newNodePtr;

        }
}

template<typename T>
void SortedList<T>::DeleteTop(/* out */ T &item)
// Pre:  list is not empty && list elements in ascending order
// Post: item == element of first list node @ entry
//       && node containing item is no longer in linked list
//       && list elements in ascending order
{
        NodeType<T>  *tempPtr = head ; // Points to first node in list.

        // Returns the value stored in head to main by reference.
    item = head->item;

    // Makes the next node the new head.
        head = head->link;

        // Deallocates the memory occupied by the old head.
        delete  tempPtr;
}

template<typename T>
void  SortedList<T>::Delete ( /* in */  T  item )
//  Pre:       list is not empty && list elements in ascending order
//           &&  item == component member of some list node
// Post:          item == element of first list node @ entry
//         &&  node containing first occurrence of item is no longer
//                in linked list   &&  list elements in ascending order
{

        // IF list is empty prints error message, ELSE searches list for value.
        if(IsEmpty())
        {
                cout << "Can\'t delete from an empty list.\n";
        }
```

```
        else
        {
                NodeType<T>  *delPtr;  // Pointer used to deallocate memory if
value is
                                                        // found.
                NodeType<T>  *currPtr; // Points the the node being checked.

                bool found = false; // Whether or not the value is found.

                // IF value is in the first node of the list, makes head the next node
                // and deletes the first node, ELSE searches the rest of the list.
                if (item == head->item)
                {
                        // Makes delPtr point to current head of list.
                        delPtr = head;

                        // Moves head to the next node in the list since the current
head
                        // will be deleted.
                        head = head->link;

                        // Deallocates the memory pointed to by delPtr.
                        delete  delPtr ;
                }
                else
                {
                        currPtr = head; // Begins the search at the first node of the
list.

                        // Searches the list for the value until it is found, or until the
                        // end of the list is reached.
                        while(!found && currPtr->link != NULL)
                        {
                                // IF value is found in the next node, makes delPtr point
to that
                                // node, makes currPtr point to the node after the next
node,
```

```
                        // and sets found to true, ELSE advances currPtr to the
next node.
                        if(currPtr->link->item == item)
                        {
                                delPtr = currPtr->link;

                                currPtr->link = currPtr->link->link;

                                found = true;
                        }
                        else
                        {
                                currPtr = currPtr->link;
                        }

                }

                // IF value was not found, print error message, ELSE
deallocate the
                // memory pointed to by delPtr.
                if(!found)
                {
                        cout << "\nThere is no such node \"" << item
                                << "\" in the list.\n";
                }
                else
                {
                        delete delPtr;
                }
        }
    }
}

#endif


/*************************************************************************
*******
```

```
 * LINK 4
 * ------------------------------------------------------------------------
 * This program uses a SortedList template class to manipulate the nodes in a
 * linked list.
 *
 * EXTRA CREDIT - Change delete function to remove the memory leak and
actually
 *                                delete the node containing the value passed into
Delete().
 * ------------------------------------------------------------------------
 * INPUT:
 *        <There are no inputs>.
 *
 * OUTPUT:
 *    Linked list every time a node is inserted or deleted in descending order.

*************************************************************************
*****/

void PrintHeader(string labName, char labType, int labNum);

int main()
{
        PrintHeader("Link 4", 'E', 4);

        SortedList<int> testList; // Linked list of integers.

        // Test delete on an empty list.
        cout << "Testing deleting from empty list:\n";
        testList.Delete(54);
        cout << endl;

        // Inserts values into the list in ascending order.
        cout << "Filling list with an assortment of integers:\n";
        testList.Insert(55);
        testList.Insert(60);
        testList.Insert(70);
        testList.Insert(10);
```

```
testList.Insert(90);

// Prints the list with all inserted values.
cout << "\nPrinting List:\n";
testList.Print();

// Test delete on an invalid value.
cout << "Testing delete of non-found item:\n";
testList.Delete(23);
cout << endl;

// Tests delete on valid values and prints the list upon successful deletion.
cout << "Testing normal, found, delete method:\nDeleting 10\n";
testList.Delete(10);

cout << "\nPrinting List:\n";
testList.Print();
cout << endl;

cout << "Deleting 60\n";
testList.Delete(60);

cout << "\nPrinting List:\n";
testList.Print();
cout << endl;

cout << "Deleting 55\n";
testList.Delete(55);

cout << "\nPrinting List:\n";
testList.Print();
cout << endl;

cout << "Deleting 70\n";
testList.Delete(70);

cout << "\nPrinting List:\n";
testList.Print();
```

```
        cout << endl;

        cout << "Deleting 80\n";
        testList.Delete(80);

        cout << "\nPrinting List:\n";
        testList.Print();
        cout << endl;

        cout << "Deleting 90\n";
        testList.Delete(90);

        cout << "\nPrinting List:\n";
        testList.Print();
        cout << endl;

        cout << "Deleting 60\n";
        testList.Delete(100);

        cout << "\nPrinting List:\n";
        testList.Print();
        cout << endl;

        return 0;
}

/*****************************************************************************
*******
 * PrintHeader
 * ---------------------------------------------------------------------------
 * This function prints the project header.
 * ---------------------------------------------------------------------------
 * PRE-CONDITIONS:
 *    labName - Lab Name has to be preciously defined
 *    labType -  Lab Type has to be preciously defined
 *    labNum  - Lab Number has to be preciously defined
 *
 * POST-CONDITIONS:
```

```
 *     This function will print the class heading.

*******************************************************************************
*****/
void PrintHeader(string labName, char labType, int labNum)
{
        cout << left;
        cout << "************************************************\n";

        cout << "\n************************************************\n\n";
        cout << right;
}

//output

************************************************

************************************************

Testing deleting from empty list:
Can't delete from an empty list.

Filling list with an assortment of integers:

Printing List:
90
70
60
55
10
Testing delete of non-found item:

There is no such node "23" in the list.

Testing normal, found, delete method:
Deleting 10

Printing List:
90
```

```
70
60
55

Deleting 60

Printing List:
90
70
55

Deleting 55

Printing List:
90
70

Deleting 70

Printing List:
90

Deleting 80

There is no such node "80" in the list.

Printing List:
90

Deleting 90

Printing List:
There is nothing to print

Deleting 60
Can't delete from an empty list.

Printing List:
```

There is nothing to print

//link5

```
#ifndef LINK5_H_
#define LINK5_H_

#include <iostream>
#include <iomanip>
using namespace std;

template <class Type>
struct nodeType
{
        Type info;
        nodeType<Type> *link;
};

template<class Type>
class linkedListType
{
public:
   const linkedListType<Type> & operator =(const linkedListType<Type> &);
   void initializeList();
   bool isEmptyList();
   void print();
   int length();
   void destroyList();
   void retrieveFirst(Type &firstElement);
   void search(const Type &searchItem);
   void insertFirst(const Type &newItem);
   void insertLast(const Type &newItem);
   void deleteNode(const Type &deleteItem);
   linkedListType();
```

```
    linkedListType(const linkedListType<Type> &otherList);
    ~linkedListType();

protected:
    nodeType<Type> *first; //pointer to the first node of the list
    nodeType<Type> *last;  //pointer to the last node of the list
};

// Overloads the assignment operator.
template<class Type>
const linkedListType<Type>& linkedListType<Type>::operator=
                                                (const linkedListType<Type>
&otherList)
{
        nodeType<Type> *newNode; // Pointer to create a node.
        nodeType<Type> *current; // Pointer to traverse the list.

        // IF attempting to self-copy, destroy the list, ELSE copy the list to
        // the calling object.
        if(this != &otherList)
        {
                // IF the list is not empty, destroy the list.
                if(first != NULL)
                {
                        destroyList();
                }

                // IF list to be copied is empty, assign NULL to first and last of
calling
                // object, ELSE copy contents of list to be copied in calling object.
                //
                if(otherList.first == NULL)
                {
                        first = NULL;
                        last = NULL;
                }
                else
                {
```

```
                    // Points current to the first node in the other list.
                    current = otherList.first;

                    // Copies the first element by first creating a new node.
                    first = new nodeType<Type>;

                    // Copies the info in the list to the calling object's first node.
                    first->info = current->info;        //copy the info

                    // Link points to NULL.
                    first->link = NULL;

                    // Last points to first (because at this point there is only one
node).
                    last = first;           //make last point to the first node

                    // Make current point to the next node in the list being copied
from.
                    current = current->link;

                    // Copies the rest of the list.
                    while(current != NULL)
                    {
                            // Creates new node.
                            newNode = new nodeType<Type>;

                            // Assigns info in current to new node's info.
                            newNode->info = current->info;

                            // Assigns NULL to newNode's link to make it the last
node in
                            // the list.
                            newNode->link = NULL;

                            // Connects the previous node to the new node.
                            last->link = newNode;

                            // Has last point to the new node.
```

```
                        last = newNode;

                        // Moves current to the next node in the list being
copied from.
                        current = current->link;
                    }
                }
        }

  return *this;
}

//Initialize the list to an empty state
//Post: first = NULL, last = NULL
template<class Type>
void linkedListType<Type>::initializeList()
{
        // Deletes any nodes in the list.
        destroyList();
}

// IF the list is empty, returns true, ELSE it returns false.
template<class Type>
bool linkedListType<Type>::isEmptyList()
{
        return(first == NULL);
}

// Outputs the data contained in each node.
// Pre: The list must exist.
template<class Type>
void linkedListType<Type>::print()
{
        nodeType<Type> *current; // Pointer to traverse the list.

        // Sets the pointer to the first node of the list.
        current = first;
```

```
        // Prints the value in the node being pointed to by current while current
        // does not point to NULL
        while(current != NULL)
        {
          cout << current->info << " ";

          current = current->link;
        }

        cout << endl;
}

// Returns the number of elements in the list.
template<class Type>
int linkedListType<Type>::length()
{
        int count = 0; // Counter that will be incremented every time the while
                                // loop is entered.

        nodeType<Type> *current; // Pointer to traverse the list.

        // Points current to the first node in the list.
        current = first;

        // Counts the number of nodes in the list while current does not point to
NULL.
        while(current!= NULL)
        {
                count++;

                current = current->link;
        }

        return count;
}

// Deletes all nodes from the list.
// Post: first = NULL, last = NULL
```

```
template<class Type>
void linkedListType<Type>::destroyList()
{
	nodeType<Type> *temp; // Pointer used to deallocate the memory occupied
by
                                        // the node

	// Deletes nodes in the list while there are still nodes.
	while(first != NULL)
	{
		// Sets temp to the current node.
		temp = first;

		// Advances first to the next node.
		first = first->link;

		// Deallocates the memory occupied by temp.
		delete temp;
	}

	// Assigns NULL to last (first was set to NULL in the while loop).
	last = NULL;
}

// Returns the info contained in the first node of the list.
// Post: firstElement = first element of the list
template<class Type>
void linkedListType<Type>::retrieveFirst(Type &firstElement)
{
	// Assigns the info stored in the first node of the linked list to
firstElement.
	firstElement = first->info;
}

// IF item is found, prints "Item is found in the list" if searchItem is in the
// list", ELSE prints "Item is not in the list".
template<class Type>
void linkedListType<Type>::search(const Type &item)
```

```
{
	nodeType<Type> *current; // Pointer used to traverse the list.

	bool found = false;

	// IF list is empty, prints error message, ELSE searches the list for the
	// item.
	if(first == NULL)
	{
		cout << "Cannot search an empty list. " << endl;
	}
	else
	{
		// Makes current point to the first node in the list.
		current = first;

		// Advances the pointer through the list while the value entered is
not
		// found.
		while(!found && current != NULL)
		{
			// IF info in current node is equal to value passed in, found
becomes
			// true, ELSE moves to next node in list.
			if(current->info == item)
			{
				found = true;
			}
			else
			{
				current = current->link;
			}
		}

		// IF found, prints message that it was found, ELSE prints not found
		// message.
		if(found)
		{
```

```
                cout << "Item is found in the list."<< endl << endl;
            }
            else
            {
                cout << "Item is not in the list." << endl << endl;
            }
    }
}

// Default constructor.
template<class Type>
linkedListType<Type>::linkedListType()
{
    first = NULL;
    last = NULL;
}

// A new item is inserted at the front of the list.
// Post: First points to the new list and the newItem inserted at the beginning
// of the list
template<class Type>
void linkedListType<Type>::insertFirst(const Type &newItem)
{
    // Pointer to the new node.
    nodeType<Type> *newNode = new nodeType<Type>;

    // Stores the info the new node.
    newNode->info = newItem;

    // Inserts the node before the current first node and connects them.
    newNode->link = first;

    // Makes first point to the new node, making it the new first node.
    first = newNode;

    // IF list is empty, makes last point to the new node.
    if(last == NULL)
    {
```

```
                last = newNode;
        }
}

// New item is inserted to the end of the list.
// Post: First points to the new list and the newItem is inserted at the end of
// the list last points to the last node in the list
template<class Type>
void linkedListType<Type>::insertLast(const Type &newItem)
{
        // Pointer to the new node.
    nodeType<Type> *newNode = new nodeType<Type>;

        // Stores the info the new node.
    newNode->info = newItem;

    // Sets the node's link field to NULL to indicate that it is the last node
    // in the list.
    newNode->link = NULL;

    // IF the list is empty, makes the new node both the first and the last node,
    // ELSE inserts node at the end of the list.
        if(first == NULL)
        {
                first = newNode;
                last = newNode;
        }
        else
        {
                // Makes the current last link point to the new last link.
                last->link = newNode; //insert newNode after last

                // Makes last point to the new last link.
                last = newNode;
        }
}

// If found, the node containing deleteItem is deleted from the list.
```

```
// Post: First points to the first node and last points to the last node of the
// updated list
template<class Type>
void linkedListType<Type>::deleteNode(const Type &deleteItem)
{
	nodeType<Type> *current;	// Pointer to traverse the list.
	nodeType<Type> *trailCurrent; // Pointer to node just before current.

	bool found = false; // Keeps track of whether or not item was found.

	// IF list is empty, returns to main, ELSE searches the list for the item
	// and deletes it if it is found.
	if(first == NULL)
	{
		cout << "\nCannot delete from an empty list.\n\n";

		return;
	}
	else
	{
		// IF the item to be deleted is the first item, deletes it, ELSE
searches
		// the rest of the list for the item.
		if(first->info == deleteItem)
		{
			// Makes current point to the first node in the list.
			current = first;

			// Makes first point to the second itemin the list.
			first = first ->link;

			// IF list is empty, makes last point to NULL.
			if(first == NULL)
			{
				last = NULL;
			}

			// Deletes the node.
```

```
                delete current;

                // RECURSIVE CALL that searches the list for any other
instances
                // of the number and deletes them.
                deleteNode(deleteItem);
        }
        else
        {
                // Sets trailCurrent to the first node in the list.
                trailCurrent = first;

                // Sets current to point to the second node in the list.
                current = first->link;

                // Searches the list for the value passed in until it is found, or
                // until the end of the list is reached.
                while((!found) && (current != NULL))
                {
                        // IF item is not found in the current list, trailCurrent
catches
                        // up to current and current moves to the next item,
ELSE found
                        // becomes true and loop is exited.
                        if(current->info != deleteItem)
                        {
                                trailCurrent = current;
                                current = current-> link;
                        }
                        else
                        {
                                found = true;
                        }
                }

                // IF found, deletes the node, ELSE prints not found message.
                if(found)
                {
```

```
                        // Connects the node before the current node to the
node the
                        // current node points to.
                        trailCurrent->link = current->link;

                        // IF node to be deleted is last node, last points to
trailCurrent.
                        if(last == current)
                        {
                                last = trailCurrent;
                        }

                        // Deletes the node from the list.
                        delete current;

                        // RECURSIVE CALL that searches the list for any
other instances
                        // of the number and deletes them.
                        deleteNode(deleteItem);
                }
                else
                {
                        cout << "Item to be deleted is not in the list." << endl;

                        return;
                }
        }
    }
}

// Destructor
// Deletes all nodes from the list
// Post: list object is destroyed
template<class Type>
linkedListType<Type>::~linkedListType() // destructor
{
    nodeType<Type> *temp; // Pointer to traverse the list.
```

```
        // Deletes nodes in the list while there are still nodes.
        while(first != NULL)
        {
                // Makes temp point to the current node.
                temp = first;

                // Advances first to the next node.
                first = first->link;

                // Deallocates the memory stored in temp.
                delete temp;
        }

        // Set last to NULL since first is already NULL
        last = NULL;
}

// Copy constructor
template<class Type>
linkedListType<Type>::linkedListType(const linkedListType<Type> &otherList)
{
        cout << "\nCopy constructor is called\n\n";

   nodeType<Type> *newNode; // Pointer to create a node.
   nodeType<Type> *current; // Pointer to traverse the list.

   if(otherList.first == NULL) //otherList is empty
   {
                first = NULL;
                last = NULL;
        }
        else
        {
                // Makes current point to the list to be copied.
                current = otherList.first;

                // Creates the first node.
                first = new nodeType<Type>;
```

```
            // Copies the information.
            first->info = current->info;

            // Sets the link field of the node to NULL.
            first->link = NULL;

            // Makes last point to first.
            last = first;

            // Makes current point to the next node.
            current = current->link;

            // Copies the remaining nodes in the list.
            while(current != NULL)
            {
                  // Creates a new node.
                  newNode = new nodeType<Type>;

                  // Copies the information.
                  newNode->info = current->info;

                  // Sets the link of the new node to NULL
                  newNode->link = NULL;

                  // Connects the old last node to the new last node.
                  last->link = newNode;

                  // Makes last point to the actual last node.
                  last = newNode;

                  // Makes current point to the next node.
                  current = current->link;
            }
      }
}

#endif
```

```
/*************************************************************************
*******
 * LINK 5
 * ------------------------------------------------------------------------
 * This program uses a linkedListType class to manipulate the nodes in a
 * linked list.
 *
 *
 * ------------------------------------------------------------------------
 * INPUT:
 *        num       - Value to added to the linked list.
 *        searchInt - Integer to be searched for.
 *        deleteInt - Integer to be deleted from the list.
 *
 * OUTPUT:
 *    Linked list every time a node is inserted or deleted.

*************************************************************************
*****/

void PrintHeader(string labName, char labType, int labNum);

int main()
{


        linkedListType<int> list1, list2; // Linked lists of integers.

        int num; // Number to add to the list.

        // Prompts the user to enter the first value in the linked list.
        cout << "Enter numbers ending with -999" << endl;
```

```
    cin  >> num;

    // Creates a linked list of integers until the user enters -999.
    while(num != -999)
    {
        list1.insertLast(num);

        cin >> num;
        cin.ignore();
    }

    // Prints list1.
    cout << endl << "List 1: ";
    list1.print();

    // Prints the length of the list.
    cout << endl << "List 1 length is: " << list1.length() << endl << endl;

    // Calls overloaded assignment operator function to copy list1 into list2
    list2 = list1;

    // Prints list2.
    cout << "List 2: ";
    list2.print();

    // Prints the length of list2.
    cout << endl << "List 2 length is " << list2.length() << endl;

    // Empties list2.
    cout << "All the nodes in list 2 have been destroyed." << endl << endl;
list2.initializeList();

// Confirms that list2 has been emptied.
if(list2.isEmptyList())
{
    cout << "It has been verified that List 2 is empty." << endl << endl;
}
```

```
    int firstInt; // The first integer in the list.

    // Gets the first value in list1.
list1.retrieveFirst(firstInt);

// Prints the first value in the list.
    cout << "The first node in List 1 is " << firstInt << "." << endl << endl;

    int searchInt; // Integer to be searched for.

    // Prompts the user to enter a number to search for.
    cout << "Enter a number to search: ";
    cin  >> searchInt;
    cin.ignore();

    // Searches the list for the number.
list1.search(searchInt);

    // Prompts the user to enter a number to search for.
cout << "Enter another number to search: ";
    cin  >> searchInt;
    cin.ignore();

    // Searches the list for the number.
list1.search(searchInt);

// Prompts the user to enter a number to add to the beginning of the list.
    cout << "Enter a number to add at the beginning of the list: ";
    cin  >> firstInt;
    cin.ignore();

    // Adds the entered number to the beginning of the list.
list1.insertFirst(firstInt);

// Prints list1 after deleting another integer.
cout << "\nList 1: ";
list1.print();
```

```
    int endInt; // Number to be added to the end of the list.

// Prompts the user to enter a number to add to the end of the list.
    cout << "\nEnter a number to add at the end of the list: ";
    cin  >> endInt;
    cin.ignore();

    // Adds the entered number to the end of the list.
list1.insertLast(endInt);

// Prints list1 after deleting another integer.
cout << "\nList 1: ";
list1.print();

int deleteInt; // Number to be deleted from the list.

// Prompts the user to enter a number to delete from the list.
    cout << "\nEnter a number to delete from the list: ";
    cin  >> deleteInt;
    cin.ignore();

    // Deletes all instances of the entered number.
    cout << "\nDeleting all instances of " << deleteInt << "." << endl << endl;
list1.deleteNode(deleteInt);

// Prints list1 after deleting the specified integer.
cout << "\nList 1: ";
list1.print();

// Prompts the user to enter a number to delete from the list.
cout << "\nEnter another number to delete from the list: ";
    cin  >> deleteInt;
    cin.ignore();

    // Deletes all instances of the entered number.
    cout << "\nDeleting all instances of " << deleteInt << ".\n\n";
list1.deleteNode(deleteInt);
```

```
    // Prints list1 after deleting another integer.
    cout << "\nList 1: ";
    list1.print();

    cout << "\nEnd of program.";

        return 0;
}

/*************************************************************************
*******
 * PrintHeader
 * -------------------------------------------------------------------------
 * This function prints the project header.
 * -------------------------------------------------------------------------
 * PRE-CONDITIONS:
 *    labName - Lab Name has to be preciously defined
 *    labType -  Lab Type has to be preciously defined
 *    labNum  - Lab Number has to be preciously defined
 *
 * POST-CONDITIONS:
 *    This function will print the class heading.

*************************************************************************
*****/
void PrintHeader(string labName, char labType, int labNum)
{
        cout << left;
        cout << "\n**************************************************\n\n";
        cout << right;
}
```

Output

```
Enter numbers ending with -999

2
5
6
```

7
7
8
-999

List 1: 2 5 6 7 7 8

List 1 length is: 6

List 2: 2 5 6 7 7 8

List 2 length is 6
All the nodes in list 2 have been destroyed.

It has been verified that List 2 is empty.

The first node in List 1 is 2.

Enter a number to search: 44
Item is not in the list.

Enter another number to search: 7
Item is found in the list.

Enter a number to add at the beginning of the list: 8

List 1: 8 2 5 6 7 7 8

Enter a number to add at the end of the list: 44

List 1: 8 2 5 6 7 7 8 44

Enter a number to delete from the list: 66

Deleting all instances of 66.

Item to be deleted is not in the list.

```
List 1: 8 2 5 6 7 7 8 44

Enter another number to delete from the list: 44

Deleting all instances of 44.

Item to be deleted is not in the list.

List 1: 8 2 5 6 7 7 8

End of program
```

```
// link6.cpp
// implements linked list as a template

#include <iostream>
using namespace std;


template<class TYPE>                // struct link<TYPE>
struct link                         // one element of list
// ( within this struct def link means link<TYPE> )
        {
  TYPE data;                        // data item
        link* next;                 // pointer to next link
  };

template<class TYPE>                // class linklist<TYPE>
class linklist                      // a list of links
// ( within this class def linklist means linklist<TYPE> )
        {
  private:
                link<TYPE>* first;          // pointer to first link
  public:
                linklist() ;                // no-argument constructor
                // note: destructor would be nice; not shown for simplicity
                void additem(TYPE d);       // add data item (one link)
```

```
            void display();              // display all links
        };

        template<class TYPE>
        linklist<TYPE>::linklist()      // no-argument constructor
                        {
                         first = NULL;          // no first link
                        }

template<class TYPE>
void linklist<TYPE>::additem(TYPE d)  // add data item
        {
  link<TYPE>* newlink = new link<TYPE>;  // make a new link
        newlink->data = d;              // give it data
  newlink->next = first;            // it points to next link
        first = newlink;                // now first points to this
  }

template<class TYPE>
void linklist<TYPE>::display()       // display all links
  {
        link<TYPE>* current = first;       // set ptr to first link
  while( current != NULL )          // quit on last link
                {
                cout << endl << current->data;  // print data
     current = current->next;       // move to next link
                }
        }
int main()
        {
  linklist<double> doubleLinkList; // doubleLinkList is object of class
linklist<double>

  doubleLinkList.additem(151.5);   // add three doubles to list doubleLinkList
  doubleLinkList.additem(262.6);
  doubleLinkList.additem(373.7);
  doubleLinkList.display();       // display entire list doubleLinkList
```

```
  linklist<char> charLinkList;  // charLinkList is object of class linklist<char>

  charLinkList.additem('a');    // add three chars to list charLinkList
  charLinkList.additem('b');
  charLinkList.additem('c');
  charLinkList.display();       // display entire list charLinkList
        }
output

373.7
262.6
151.5
c
b
a

// link7.cpp
// implements linked list as a template
// demonstrates list used with employee class
// the << and >> operators in the employee class are overloaded.
//so the linked list knows how to input and output its data.
//Any operators used in template must be defined in an appropriate
//way for any data type or class used to instantiate the template.

#include <iostream>
using namespace std;


////////////////////////////////////////////////////////////////
// the employee class
////////////////////////////////////////////////////////////////
const int LEN = 80;          // maximum length of names

class employee                    // employee class
  {
  private:
        char name[LEN];              // employee name
        unsigned long number;        // employee number
```

```
  public:
    friend istream& operator >> (istream& s, employee& e);
    friend ostream& operator << (ostream& s, employee& e);
  };

istream& operator >> (istream& s, employee& e)
  {
        cout << "\n   Enter last name: "; cin >> e.name;
  cout << "   Enter number: ";      cin >> e.number;
  return s;
        }
ostream& operator << (ostream& s, employee& e)
  {
        cout << "\n   Name: " << e.name;
  cout << "\n   Number: " << e.number;
  return s;
        }
////////////////////////////////////////////////////////////////
// the linked list template
////////////////////////////////////////////////////////////////
template<class TYPE>                // struct "link<TYPE>"
struct link                         // one element of list
        {
  TYPE data;                         // data item
  link* next;                       // pointer to next link
  };

template<class TYPE>                // class "linklist<TYPE>"
class linklist                      // a list of links
  {
  private:
    link<TYPE>* first;              // pointer to first link
  public:
    linklist()                     // no-argument constructor
                        { first = NULL; }           // no first link
    void additem(TYPE d);           // add data item (one link)
    void display();                 // display all links
  };
```

```
template<class TYPE>
void linklist<TYPE>::additem(TYPE d)  // add data item
  {
  link<TYPE>* newlink = new link<TYPE>;  // make a new link
  newlink->data = d;               // give it data
  newlink->next = first;            // it points to next link
  first = newlink;                 // now first points to this
       }

template<class TYPE>
void linklist<TYPE>::display()       // display all links
  {
  link<TYPE>* current = first;       // set ptr to first link
       while( current != NULL )          // quit on last link
    {
    cout << endl << current->data;  // display data
    current = current->next;        // move to next link
    }
  }
////////////////////////////////////////////////////////////////
// main() creates a linked list of employees
////////////////////////////////////////////////////////////////
int main()
  {                       // employeeLinkList is object of
  linklist<employee> employeeLinkList;   // class "linklist<employee>"
       employee emptemp;          // temporary employee storage
  char ans;                 // user's response ('y' or 'n')

  do
    {
    cin >> emptemp;         // get employee data from user
    employeeLinkList.additem(emptemp);  // add it to linked list employeeLinkList
    cout << "\nAdd another (y/n)? ";
               cin >> ans;
    } while(ans != 'n');    // when user is done,
       employeeLinkList.display();           // display entire linked list
       }
```

output

```
  Enter last name: Jones
  Enter number: 12345

Add another (y/n)? y

  Enter last name: Smith
  Enter number: 34593

Add another (y/n)? y

  Enter last name: Leno
  Enter number: 98634

Add another (y/n)? n


  Name: Leno
  Number: 98634

  Name: Smith
  Number: 34593

  Name: Jones
  Number: 12345
```

```
//link8.h
#ifndef H_doublyLinkedList
#define H_doublyLinkedList

#include <iostream>

using namespace std;

//Definition of the node
template <class Type>
struct  nodeType
```

```
{
    Type info;
    nodeType<Type>  *next;
    nodeType<Type>  *back;
};

template <class Type>
class doublyLinkedList
{
public:
   void initializeList();
                //Initialize list to an empty state
                //Post: first = NULL
   bool  isEmptyList();
                //Function returns true if the list is empty;
                //otherwise, it returns false
   void destroy();
                //Delete all nodes from the list
                //Post: first = NULL
   void print();
                //Output the info contained in each node
   int length();
                //Function returns the number of nodes in the list
   void search(const Type& searchItem);
                //Outputs "Item is found in the list" if searchItem
                //is in the list; otherwise, outputs "Item not in the list"
   void insertNode(const Type& insertItem);
                //newItem is inserted in the list
                //Post: first points to the new list and the
                //   newItem is inserted at the proper place in the list
   void deleteNode(const Type& deleteItem);
                //If found, the node containing the deleteItem is deleted
                //from the list
                //Post: first points to the first node of the
                //   new list
   doublyLinkedList();
                //Default constructor
                //Initialize list to an empty state
```

```
            //Post: first = NULL
   doublyLinkedList(const doublyLinkedList<Type>& otherList);
            //copy constructor
   ~doublyLinkedList();
            //Destructor
            //Post: list object is destroyed

private:
   nodeType<Type> *first;  //pointer to the list
};



template<class Type>
doublyLinkedList<Type>::doublyLinkedList()
{
       first= NULL;
}


template<class Type>
bool doublyLinkedList<Type>::isEmptyList()
{
   return(first == NULL);
}


template<class Type>
void doublyLinkedList<Type>::destroy()
{
       nodeType<Type>  *temp; //pointer to delete the node

       while(first != NULL)
       {
              temp = first;
              first = first->next;
              delete temp;
       }
```

```
}

template<class Type>
void doublyLinkedList<Type>::initializeList()
{
        destroy();
}


template<class Type>
int doublyLinkedList<Type>::length()
{
        int length = 0;
        nodeType<Type> *current; //pointer to traverse the list

        current = first;  //set current to point to the first node

        while(current != NULL)
        {
                length++;   //increment length
                current = current->next; //advance current
        }

        return length;
}


template<class Type>
void doublyLinkedList<Type>::print()
{
   nodeType<Type> *current; //pointer to traverse the list

        current = first;  //set current to point to the first node

        while(current != NULL)
        {
          cout<<current->info<<"  ";  //output info
          current = current->next;
```

```
        }//end while
}//end printList


template<class Type>
void doublyLinkedList<Type>::search(const Type& searchItem)
{
  bool found;
  nodeType<Type> *current; //pointer to traverse the list

  if(first == NULL)
    cout<<"Cannot search an empty list"<<endl;
  else
  {
       found = false;
              current = first;

              while(current != NULL && !found)
                     if(current->info >= searchItem)
                            found = true;
                     else
                            current = current->next;

              if(current == NULL)
                     cout<<"Item not in the list"<<endl;
              else
                     if(current->info == searchItem) //test for equality
                            cout<<"Item is found in the list"<<endl;
                     else
                            cout<<"Item not in the list"<<endl;
  }//end else
}//end search

template<class Type>
void doublyLinkedList<Type>::insertNode(const Type& insertItem)
{
   nodeType<Type> *current; // pointer to traverse the list
   nodeType<Type> *trailCurrent; // pointer just before current
```

```
nodeType<Type> *newNode;  // pointer to create a node
bool found;

newNode = new nodeType<Type>; //create the node
newNode->info = insertItem;  //store new item in the node
newNode->next = NULL;
newNode->back = NULL;

if(first == NULL) //if list is empty, newNode is the only node
            first = newNode;
else
{
            found = false;
            current = first;

            while(current != NULL && !found) //search the list
                  if(current->info >= insertItem)
                        found = true;
                  else
                  {
                        trailCurrent = current;
                        current = current->next;
                  }

            if(current == first) //insert new node before first
            {
                  first->back = newNode;
            newNode->next = first;
                  first = newNode;
            }
            else
            {
                  //insert newNode between trailCurrent and current
                  if(current != NULL)
                  {
                        trailCurrent->next = newNode;
                        newNode->back = trailCurrent;
                        newNode->next = current;
```

```
                        current ->back = newNode;
                }
                else
                {
                        trailCurrent->next = newNode;
                        newNode->back = trailCurrent;
                }
        }//end else
}//end else
}//end insertNode

template<class Type>
void doublyLinkedList<Type>::deleteNode(const Type& deleteItem)
{
        nodeType<Type> *current; // pointer to traverse the list
        nodeType<Type> *trailCurrent; // pointer just before current

        bool found;

        if(first == NULL)
                cout<<"Cannot delete from an empty list"<<endl;
        else
                if(first->info == deleteItem)     // node to be deleted is the
                                                                // first node
                {
                        current = first;
                        first = first->next;

                        if(first != NULL)
                                first->back = NULL;

                        delete current;
                }
                else
                {
                        found = false;
                        current = first;
```

```
                    while(current != NULL && !found)  //search the list
                            if(current->info >= deleteItem)
                                    found = true;
                            else
                                    current = current->next;

                    if(current == NULL)
                            cout<<"Item to be deleted is not in the list"<<endl;
                    else
                            if(current->info == deleteItem) //check for equality
                            {
                                    trailCurrent = current->back;
                                    trailCurrent->next = current->next;

                                    if(current->next != NULL)
                                            current->next->back = trailCurrent;

                                    delete current;
                            }
                            else
                                    cout<<"Item to be deleted is not in list."<<endl;
    }//end else
}//end deleteNode



template<class Type>
doublyLinkedList<Type>::~doublyLinkedList()
{
        // cout<<"Needs to be written"<<endl;
}
#endif
//link8.cpp
//Program to test the various operations on a doubly linked list

#include <iostream>
#include "link8.h"
```

```
using namespace std;

int main()
{
	doublyLinkedList<int> intlist;
	int num;

	cout<<"Enter a list of positive integers ending "
		<<"with -999: "<<endl;
	cin>>num;

	while(num != -999)
	{
		intlist.insertNode(num);
		cin>>num;
	}

	cout<<endl;
	cout<<"List in ascending order: ";
	intlist.print();
	cout<<endl;

	return 0;
}
```

output

```
Enter a list of positive integers ending with -999:
1
3
5
7
2
4
6
8
-999

List in ascending order: 1  2  3  4  5  6  7  8
```

```
//*************************************************************************
//link9.h

#ifndef H_StackType
#define H_StackType

#include <iostream>

using namespace std;

//Definition of the node
template <class Type>
struct nodeType
{
        Type info;
        nodeType<Type> *link;
};

template<class Type>
class linkedStackType
{
public:
  const linkedStackType<Type>& operator= (const linkedStackType<Type>&);
      //overload the assignment operator
   void initializeStack();
          //Initialize the stack to an empty state.
          //Post condition: Stack elements are removed; top = NULL
   bool isEmptyStack();
          //Function returns true if the stack is empty;
          //otherwise, it returns false
   bool isFullStack();
          //Function returns true if the stack is full;
          //otherwise, it returns false
   void push(const Type& newItem);
          //Add the newItem to the stack.
          //Pre condition: stack exists and is not full
          //Post condition: stack is changed and the newItem
          //     is added to the top of stack. top points to
```

```
        //     the updated stack
    void pop(Type& poppedElement);
        //Remove the top element of the stack.
        //Pre condition: Stack exists and is not empty
        //Post condition: stack is changed and the top
        //    element is removed from the stack. The top
        //    element of the stack is saved in poppedElement
    void destroyStack();
        //Remove all elements of the stack, leaving the
        //stack in an empty state.
        //Post condition: top = NULL
    linkedStackType();
        //default constructor
        //Post condition: top = NULL
    linkedStackType(const linkedStackType<Type>& otherStack);
        //copy constructor
    ~linkedStackType();
        //destructor
        //All elements of the stack are removed from the stack

private:
    nodeType<Type> *top; // pointer to the stack
};


template<class Type> //default constructor
linkedStackType<Type>::linkedStackType()
{
        top = NULL;
}

template<class Type>
void linkedStackType<Type>::destroyStack()
{
        nodeType<Type> *temp; //pointer to delete the node

        while(top != NULL)  //while there are elements in the stack
        {
```

```
          temp = top;      //set temp to point to the current node
          top = top->link; //advance top to the next node
          delete temp;     //deallocate memory occupied by temp
        }
}// end destroyStack



template<class Type>
void linkedStackType<Type>:: initializeStack()
{
   destroyStack();
}

template<class Type>
bool linkedStackType<Type>::isEmptyStack()
{
        return(top == NULL);
}

template<class Type>
bool linkedStackType<Type>:: isFullStack()
{
  return 0;
}

template<class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
        nodeType<Type> *newNode; //pointer to create the new node

        newNode = new nodeType<Type>; //create the node
        newNode->info = newElement;   //store newElement in the node
        newNode->link = top;          //insert newNode before top
        top = newNode;           //set top to point to the top node
} //end push
```

```
template<class Type>
void linkedStackType<Type>::pop(Type& poppedElement)
{
  nodeType<Type> *temp;         //pointer to deallocate memory

  poppedElement = top->info;    //copy the top element into
                                //poppedElement
  cout << "Popped item is " << poppedElement << endl;

  temp = top;                   //set temp to point to the top node
  top = top->link;              //advance top to the next node
  delete temp;                  //delete the top node
}//end pop



template<class Type>   //copy constructor
linkedStackType<Type>::linkedStackType(const linkedStackType<Type>&
otherStack)
{
      nodeType<Type> *newNode, *current, *last;

      if(otherStack.top == NULL)
            top = NULL;
      else
      {
            current = otherStack.top;       //set current to point to the
                                            //stack to be copied

                  //copy the top element of the stack
            top = new nodeType<Type>;   //create the node
            top->info = current->info;  //copy the info
            top->link = NULL;           //set the link field of the
                                        //node to null
            last = top;                 //set last to point to the node
            current = current->link;    //set current to point to the
                                        //next node
```

```
                //copy the remaining stack
        while(current != NULL)
        {
                newNode = new nodeType<Type>;
                newNode->info = current->info;
                newNode->link = NULL;
                last->link = newNode;
                last = newNode;
                current = current->link;
        }//end while
    }//end else
}//end copy constructor


template<class Type> //destructor
linkedStackType<Type>::~linkedStackType()
{
    nodeType<Type> *temp;

    while(top != NULL)    //while there are elements in the stack
    {
        temp = top;       //set temp to point to the current node
        top = top ->link;  //advance first to the next node
        delete temp;       //deallocate the memory occupied by temp
    }//end while
}//end destructor

template<class Type>   //overloading the assignment operator
const linkedStackType<Type>& linkedStackType<Type>::operator=
                    (const linkedStackType<Type>& otherStack)
{
    nodeType<Type> *newNode, *current, *last;

    if(this != &otherStack) //avoid self-copy
    {
        if(top != NULL)  //if the stack is not empty, destroy it
            destroyStack();
```

```
        if(otherStack.top == NULL)
            top = NULL;
        else
        {
            current = otherStack.top;        //set current to point to
                                                    //the stack to be
copied

                //copy the top element of otherStack
            top = new nodeType<Type>;        //create the node
            top->info = current->info; //copy the info
            top->link = NULL;                //set the link field of the
                                                    //node to null
            last = top;                    //make last point to the node
            current = current->link;    //make current point to
                                                //the next node

                //copy the remaining elements of the stack
            while(current != NULL)
            {
                newNode = new nodeType<Type>;
                newNode->info = current->info;
                newNode->link = NULL;
                last->link = newNode;
                last = newNode;
                current = current->link;
            }//end while
        }//end else
    }//end if

    return *this;
}//end operator=
#endif
//link9.cpp
//This program tests the various operations of a linked stack

#include <iostream>
#include "link9.h"
```

```
using namespace std;

void testCopy(linkedStackType<int> OStack);

int main()
{
	linkedStackType<int> stack;
	linkedStackType<int> otherStack;
	linkedStackType<int> newStack;
	int num;

	stack.push(34);
	stack.push(43);
	stack.push(27);
	newStack = stack;

	cout<<"After the assignment operator, newStack: "<<endl;

	while(!newStack.isEmptyStack())
	{
		newStack.pop(num);
		cout<<num<<endl;
	}

	otherStack = stack;

	cout<<"Testing the copy constructor"<<endl;

	testCopy(otherStack);

	cout<<"After the copy constructor, otherStack: "<<endl;

	while(!otherStack.isEmptyStack())
	{
		otherStack.pop(num);
		cout<<num<<endl;
	}
```

```
        linkedStackType<int> intStack;
        int poppedInt;
        intStack.push(23);
        intStack.push(45);
        intStack.push(38);

   intStack.pop(poppedInt);
   intStack.pop(poppedInt);
   intStack.pop(poppedInt);


        linkedStackType<float> floatStack;      // floatStack is object of class
Stack<float>
        float poppedFloat;
   floatStack.push(1111.1);      // push 3 floats, pop 3 floats
   floatStack.push(2222.2);
   floatStack.push(3333.3);
   floatStack.pop(poppedFloat);
   floatStack.pop(poppedFloat);
   floatStack.pop(poppedFloat);

   linkedStackType<long> longStack;       // longStack is object of class
Stack<long>
        long poppedLong;
   longStack.push(123123123L);  // push 3 longs, pop 3 longs
   longStack.push(234234234L);
   longStack.push(345345345L);
   longStack.pop(poppedLong);
   longStack.pop(poppedLong);
   longStack.pop(poppedLong);

        return 0;
}

void testCopy(linkedStackType<int> OStack) //function to test the
                                                                    // copy
constructor
```

```
{
        int num;

        cout<<"Stack in the function testCopy:"<<endl;

        while(!OStack.isEmptyStack())
        {
                OStack.pop(num);
                cout<<num<<endl;
        }
}
```

output

```
After the assignment operator, newStack:
Popped item is 27
27
Popped item is 43
43
Popped item is 34
34
Testing the copy constructor
Stack in the function testCopy:
Popped item is 27
27
Popped item is 43
43
Popped item is 34
34
After the copy constructor, otherStack:
Popped item is 27
27
Popped item is 43
43
Popped item is 34
34
Popped item is 38
Popped item is 45
Popped item is 23
Popped item is 3333.3
```

Popped item is 2222.2
Popped item is 1111.1
Popped item is 345345345
Popped item is 234234234
Popped item is 123123123