1 Short Answer (7 Points Each)

1. Given the following function, what operations will need to be overloaded in the class T for this code to compile?

   ```
   template <class T>
   T square(T n)
   {
     return n * n;
   }
   ```

   Solution: The * operator and the copy constructor.

2. What is a container and what is an iterator?

   Solution: The most important data structures in the STL are containers and iterators. A container is a class that stores data and organizes it in some fashion. An iterator is an object that behaves like a pointer. It is used to access the individual data elements in a container.

3. What does LIFO mean and what data structure uses this type of access?

   Solution:

   Last In First Out, the stack.

4. What does FIFO mean and what data structure uses this type of access?

   Solution:

   First In First Out, the queue.

5. Describe two operations that stacks must perform.

   Solution:

   (a) push: which inserts an element onto the top of the stack.

   (b) pop: which removes and returns an element from the top of the stack.

6. Describe two operations that queues must perform.

   Solution:

   (a) enqueue: which inserts an element onto the back of the queue.

   (b) dequeue: which removes and returns an element from the front of the queue.

7. What are some of the advantages that linked lists have over arrays?

   Solution:

   (a) Insertion and deletion operations are faster since there is no need to shift array data.

   (b) Linked lists take up only the amount of memory it needs. There is no lost memory from allocations that are not being used.

   (c) No need to resize the linked list like you would need to do with an array if the array is full. The linked list does not have space restrictions.

8. Write an implementation for the linked list version of the ListCollection class for the concatenation operator +. Recall that the nodes of the list are `ListNode` type and that the ListCollection class had the following functions.

   void setElement(int, T); T getElement(int); void clear(); int size(); int capacity(); void pushFront(T); void pushBack(T); T popFront(); T popBack(); void insertOrdred(T); void removeElement(T); void insert(int, T); void remove(int);

```
template <class T>
const ListCollection<T> ListCollection<T>::operator+(const ListCollection &right)
```

Solution:

```
template <class T>
const ListCollection<T> ListCollection<T>::operator+(const ListCollection &right)
{
    ListCollection<T> newList;

    ListNode *nodeptr = head;
    while (nodeptr)
    {
        newList.pushBack(nodeptr->value);
        nodeptr = nodeptr->next;
    }

    nodeptr = right.head;
    while (nodeptr)
    {
        newList.pushBack(nodeptr->value);
        nodeptr = nodeptr->next;
    }

    return newList;
}
```

2 Coding (10 Points Each)

Given the following specification for the LinkedList class and ListNode class, write the implementations of all the functions.

```cpp
using namespace std;

template <class T>
class ListNode
{
  public:
    T value;
    ListNode<T> *next;

    ListNode(T nodeValue)
    {
        value = nodeValue;
        next = nullptr;
    }
};

template <class T>
class LinkedList
{
  private:
    ListNode<T> *head;

  public:
    LinkedList()
    {
        head = nullptr;
    }

    ~LinkedList();
    void appendNode(T);
    void insertNode(T);
    void deleteNode(T);
    void displayList() const;
};
```

- `~LinkedList()` The destructor removes all elements from the list without memory leaks.
- `appendNode(T)` This function will append a node onto the end of the list.
- `insertNode(T)` This function will insert a node into the list so that if the list is currently ordered the new list will also be ordered.
- `deleteNode(T)` This function deletes the node that has the same value as the input parameter. If the input value is not in the list the original list is unaltered.
- `displayList() const` Writes the list contents to the screen.

Solution:

```cpp
// A class template for holding a linked list.
// The node type is also a class template.
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

using namespace std;

//*********************************************
// The ListNode class creates a type used to  *
// store a node of the linked list.           *
//*********************************************

template <class T>
class ListNode
{
  public:
    T value;           // Node value
    ListNode<T> *next; // Pointer to the next node

    // Constructor
    ListNode(T nodeValue)
    {
        value = nodeValue;
        next = nullptr;
    }
};

//*********************************************
// LinkedList class                           *
//*********************************************

template <class T>
class LinkedList
{
  private:
    ListNode<T> *head;   // List head pointer

  public:
    // Constructor
    LinkedList()
    {
        head = nullptr;
    }

    // Destructor
    ~LinkedList();

    // Linked list operations
    void appendNode(T);
    void insertNode(T);
    void deleteNode(T);
    void displayList() const;
};

//**************************************************
// appendNode appends a node containing the value  *
// pased into newValue, to the end of the list.    *
//**************************************************

template <class T>
void LinkedList<T>::appendNode(T newValue)
{
    ListNode<T> *newNode;  // To point to a new node
    ListNode<T> *nodePtr;  // To move through the list

    // Allocate a new node and store newValue there.
    newNode = new ListNode<T>(newValue);

    // If there are no nodes in the list
    // make newNode the first node.
    if (!head)
        head = newNode;
    else  // Otherwise, insert newNode at end.
    {
        // Initialize nodePtr to head of list.
        nodePtr = head;

        // Find the last node in the list.
        while (nodePtr->next)
            nodePtr = nodePtr->next;

        // Insert newNode as the last node.
        nodePtr->next = newNode;
    }
}

//**************************************************
// displayList shows the value stored in each node *
// of the linked list pointed to by head.          *
//**************************************************

template <class T>
void LinkedList<T>::displayList() const
{
    ListNode<T> *nodePtr;  // To move through the list

    // Position nodePtr at the head of the list.
    nodePtr = head;

    // While nodePtr points to a node, traverse
    // the list.
    while (nodePtr)
    {
        // Display the value in this node.
        cout << nodePtr->value << endl;

        // Move to the next node.
        nodePtr = nodePtr->next;
    }
}

//**************************************************
// The insertNode function inserts a node with     *
// newValue copied to its value member.            *
//**************************************************
```

```

template <class T>
void LinkedList<T>::insertNode(T newValue)
{
    ListNode<T> *newNode;               // A new node
    ListNode<T> *nodePtr;               // To traverse the list
    ListNode<T> *previousNode = nullptr; // The previous node

    // Allocate a new node and store newValue there.
    newNode = new ListNode<T>(newValue);

    // If there are no nodes in the list
    // make newNode the first node
    if (!head)
    {
        head = newNode;
        newNode->next = nullptr;
    }
    else  // Otherwise, insert newNode
    {
        // Position nodePtr at the head of list.
        nodePtr = head;

        // Initialize previousNode to nullptr.
        previousNode = nullptr;

        // Skip all nodes whose value is less than newValue.
        while (nodePtr != nullptr && nodePtr->value < newValue)
        {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }

        // If the new node is to be the 1st in the list,
        // insert it before all other nodes.
        if (previousNode == nullptr)
        {
            head = newNode;
            newNode->next = nodePtr;
        }
        else  // Otherwise insert after the previous node.
        {
            previousNode->next = newNode;
            newNode->next = nodePtr;
        }
    }
}

//*****************************************************
// The deleteNode function searches for a node        *
// with searchValue as its value. The node, if found, *
// is deleted from the list and from memory.          *
//*****************************************************

template <class T>
void LinkedList<T>::deleteNode(T searchValue)
{
    ListNode<T> *nodePtr;       // To traverse the list
    ListNode<T> *previousNode;  // To point to the previous node

    // If the list is empty, do nothing.
    if (!head)
        return;

    // Determine if the first node is the one.
    if (head->value == searchValue)
    {
        nodePtr = head->next;
        delete head;
        head = nodePtr;
    }
    else
    {
        // Initialize nodePtr to head of list
        nodePtr = head;

        // Skip all nodes whose value member is
        // not equal to num.
        while (nodePtr != nullptr && nodePtr->value != searchValue)
        {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }

        // If nodePtr is not at the end of the list,
        // link the previous node to the node after
        // nodePtr, then delete nodePtr.
        if (nodePtr)
        {
            previousNode->next = nodePtr->next;
            delete nodePtr;
        }
    }
}

//**************************************************
// Destructor                                      *
// This function deletes every node in the list.   *
//**************************************************

template <class T>
LinkedList<T>::~LinkedList()
{
    ListNode<T> *nodePtr;   // To traverse the list
    ListNode<T> *nextNode;  // To point to the next node

    // Position nodePtr at the head of the list.
    nodePtr = head;

    // While nodePtr is not at the end of the list...
    while (nodePtr != nullptr)
    {
        // Save a pointer to the next node.
        nextNode = nodePtr->next;

        // Delete the current node.
        delete nodePtr;

        // Position nodePtr at the next node.
        nodePtr = nextNode;
    }
    head = nullptr;
}
#endif
```