

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:

```
1 template <class T>
2 const ListCollection<T> ListCollection<T>::operator+(const ListCollection &right)
3 {
4     ListCollection<T> newList;
5
6     ListNode *nodeptr = head;
7     while (nodeptr)
8     {
9         newList.pushBack(nodeptr->value);
10        nodeptr = nodeptr->next;
11    }
12
13    nodeptr = right.head;
14    while (nodeptr)
15    {
16        newList.pushBack(nodeptr->value);
17        nodeptr = nodeptr->next;
18    }
19
20    return newList;
21 }
```

2 Coding (10 Points Each)

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

```

1 using namespace std;
2
3 template <class T>
4 class ListNode
5 {
6     public:
7         T value;
8         ListNode<T> *next;
9
10        ListNode(T nodeValue)
11        {
12            value = nodeValue;
13            next = nullptr;
14        }
15 };
16
17 template <class T>
18 class LinkedList
19 {
20     private:
21         ListNode<T> *head;
22
23     public:
24         LinkedList()
25         {
26             head = nullptr;
27         }
28
29         ~LinkedList();
30         void appendNode(T);
31         void insertNode(T);
32         void deleteNode(T);
33         void displayList() const;
34 };

```

- `~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:

```

1 // A class template for holding a linked list.
2 // The node type is also a class template.
3 #ifndef LINKEDLIST_H
4 #define LINKEDLIST_H
5
6 using namespace std;
7
8 //*****
9 // The ListNode class creates a type used to *
10 // store a node of the linked list. *
11 //*****
12
13 template <class T>
14 class ListNode
15 {
16     public:
17         T value; // Node value
18         ListNode<T> *next; // Pointer to the next node
19
20         // Constructor
21         ListNode(T nodeValue)
22         {
23             value = nodeValue;
24             next = nullptr;
25         }
26 };
27
28 //*****
29 // LinkedList class *
30 //*****
31
32 template <class T>
33 class LinkedList
34 {

```

```

35 private:
36     ListNode<T> *head;    // List head pointer
37
38 public:
39     // Constructor
40     LinkedList()
41     {
42         head = nullptr;
43     }
44
45     // Destructor
46     ~LinkedList();
47
48     // Linked list operations
49     void appendNode(T);
50     void insertNode(T);
51     void deleteNode(T);
52     void displayList() const;
53 };
54
55 //*****
56 // appendNode appends a node containing the value *
57 // passed into newValue, to the end of the list. *
58 //*****
59
60 template <class T>
61 void LinkedList<T>::appendNode(T newValue)
62 {
63     ListNode<T> *newNode; // To point to a new node
64     ListNode<T> *nodePtr; // To move through the list
65
66     // Allocate a new node and store newValue there.
67     newNode = new ListNode<T>(newValue);
68
69     // If there are no nodes in the list
70     // make newNode the first node.
71     if (!head)
72         head = newNode;
73     else // Otherwise, insert newNode at end.
74     {
75         // Initialize nodePtr to head of list.
76         nodePtr = head;
77
78         // Find the last node in the list.
79         while (nodePtr->next)
80             nodePtr = nodePtr->next;
81
82         // Insert newNode as the last node.
83         nodePtr->next = newNode;
84     }
85 }
86
87 //*****
88 // displayList shows the value stored in each node *
89 // of the linked list pointed to by head. *
90 //*****
91
92 template <class T>
93 void LinkedList<T>::displayList() const
94 {
95     ListNode<T> *nodePtr; // To move through the list
96
97     // Position nodePtr at the head of the list.
98     nodePtr = head;
99
100    // While nodePtr points to a node, traverse
101    // the list.
102    while (nodePtr)
103    {
104        // Display the value in this node.
105        cout << nodePtr->value << endl;
106
107        // Move to the next node.
108        nodePtr = nodePtr->next;
109    }
110 }
111
112 //*****
113 // The insertNode function inserts a node with *
114 // newValue copied to its value member. *
115 //*****

```

```

116
117 template <class T>
118 void LinkedList<T>::insertNode(T newValue)
119 {
120     ListNode<T> *newNode;           // A new node
121     ListNode<T> *nodePtr;           // To traverse the list
122     ListNode<T> *previousNode = nullptr; // The previous node
123
124     // Allocate a new node and store newValue there.
125     newNode = new ListNode<T>(newValue);
126
127     // If there are no nodes in the list
128     // make newNode the first node
129     if (!head)
130     {
131         head = newNode;
132         newNode->next = nullptr;
133     }
134     else // Otherwise, insert newNode
135     {
136         // Position nodePtr at the head of list.
137         nodePtr = head;
138
139         // Initialize previousNode to nullptr.
140         previousNode = nullptr;
141
142         // Skip all nodes whose value is less than newValue.
143         while (nodePtr != nullptr && nodePtr->value < newValue)
144         {
145             previousNode = nodePtr;
146             nodePtr = nodePtr->next;
147         }
148
149         // If the new node is to be the 1st in the list,
150         // insert it before all other nodes.
151         if (previousNode == nullptr)
152         {
153             head = newNode;
154             newNode->next = nodePtr;
155         }
156         else // Otherwise insert after the previous node.
157         {
158             previousNode->next = newNode;
159             newNode->next = nodePtr;
160         }
161     }
162 }
163
164 //*****
165 // The deleteNode function searches for a node *
166 // with searchValue as its value. The node, if found, *
167 // is deleted from the list and from memory. *
168 //*****
169
170 template <class T>
171 void LinkedList<T>::deleteNode(T searchValue)
172 {
173     ListNode<T> *nodePtr;           // To traverse the list
174     ListNode<T> *previousNode;      // To point to the previous node
175
176     // If the list is empty, do nothing.
177     if (!head)
178         return;
179
180     // Determine if the first node is the one.
181     if (head->value == searchValue)
182     {
183         nodePtr = head->next;
184         delete head;
185         head = nodePtr;
186     }
187     else
188     {
189         // Initialize nodePtr to head of list
190         nodePtr = head;
191
192         // Skip all nodes whose value member is
193         // not equal to num.
194         while (nodePtr != nullptr && nodePtr->value != searchValue)
195         {
196             previousNode = nodePtr;

```

```
197         nodePtr = nodePtr->next;
198     }
199
200     // If nodePtr is not at the end of the list,
201     // link the previous node to the node after
202     // nodePtr, then delete nodePtr.
203     if (nodePtr)
204     {
205         previousNode->next = nodePtr->next;
206         delete nodePtr;
207     }
208 }
209 }
210
211 //*****
212 // Destructor
213 // This function deletes every node in the list.
214 //*****
215
216 template <class T>
217 LinkedList<T>::~LinkedList()
218 {
219     ListNode<T> *nodePtr;    // To traverse the list
220     ListNode<T> *nextNode;   // To point to the next node
221
222     // Position nodePtr at the head of the list.
223     nodePtr = head;
224
225     // While nodePtr is not at the end of the list...
226     while (nodePtr != nullptr)
227     {
228         // Save a pointer to the next node.
229         nextNode = nodePtr->next;
230
231         // Delete the current node.
232         delete nodePtr;
233
234         // Position nodePtr at the next node.
235         nodePtr = nextNode;
236     }
237     head = nullptr;
238 }
239 #endif
```