CPT113 – Programming Methodology & Data Structures
Tutorial Week 10
Doubly and Circular Linked Lists

Learning Outcomes:

- Understanding the use of doubly and circular linked lists in class

1. Given the following linked list header file, modify the codes to make it a doubly linked list. Then, write a short main program to test the various functionalities (append, insert, delete, display).

```cpp
// A class template for holding a linked list.
#ifndef LINKEDLIST_H
#define LINKEDLIST_H
#include <iostream>     // For cout
using namespace std;

template <class T>
class LinkedList
{
private:
   // Declare a structure for the list
   struct ListNode
   {
      T value;                // The value in this node
      ListNode *next;  // To point to the next node
   };

   ListNode *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;
};
```

```cpp
//***************************************************
// 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 *newNode;  // To point to a new node
   ListNode *nodePtr;  // To move through the list

   // Allocate a new node and store newValue there.
   newNode = new ListNode;
   newNode->value = newValue;
   newNode->next = nullptr;

   // 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 *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;
   }
}
```

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

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

   // Allocate a new node and store newValue there.
   newNode = new ListNode;
   newNode->value = 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;
      }
   }
}
```

```cpp
//*****************************************************
// 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 *nodePtr;        // To traverse the list
   ListNode *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;
      }
   }
}
```

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

template <class T>
LinkedList<T>::~LinkedList()
{
   ListNode *nodePtr;   // To traverse the list
   ListNode *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;
   }
}

#endif

// Doubly linked list template
#ifndef LINKEDLIST_H
#define LINKEDLIST_H
#include <iostream>     // For cout
using namespace std;

template <class T>
class LinkedList
{
private:
   // Declare a structure for the list
   struct ListNode
   {
      T value;                 // The value in this node
      ListNode *next;  // To point to the next node
      ListNode *back;  // To point to the next node
   };

   ListNode *head;   // List head pointer
   ListNode *tail;   // List head pointer

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

   // Destructor
   ~LinkedList();

   // Linked list operations
   void appendNode(T);
```

```cpp
      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 *newNode;  // To point to a new node
   ListNode *nodePtr;  // To move through the list

   // Allocate a new node and store newValue there.
   newNode = new ListNode;
   newNode->value = newValue;
   newNode->next = nullptr;
   newNode->back = nullptr;

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

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

//**************************************************
// 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 *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.
```

```cpp
      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 *newNode;                              // A new node
   ListNode *nodePtr;                              // To traverse the list

   // Allocate a new node and store newValue there.
   newNode = new ListNode;
   newNode->value = newValue;
   newNode->next = nullptr;
   newNode->back = nullptr;

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

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

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

```cpp
    }
}

//*******************************************************
// 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 *nodePtr;          // To traverse the list

    // 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)
        {
            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)
        {
            nodePtr->back->next = nodePtr->next;
            delete nodePtr;
        }
    }
}
```

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

template <class T>
LinkedList<T>::~LinkedList()
{
   ListNode *nodePtr;   // To traverse the list
   ListNode *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;
   }
}

#endif



#include "LinkedList.h"
#include <iostream>     // For cout
using namespace std;

int main() {
     //Create two different linked lists, one for int, one for double
     LinkedList<double> *list1 = new LinkedList<double>;
     list1->appendNode(4.0);
     list1->insertNode(1.0);
     list1->insertNode(0.5);
     list1->insertNode(2.0);
     list1->insertNode(1.5);
     list1->appendNode(4.0);


     list1->displayList();
     cout << "----" << endl;

     list1->deleteNode(0.5);
     list1->displayList();
     cout << "----" << endl;

     list1->deleteNode(4.0);
     list1->displayList();
     cout << "----" << endl;

     return 0;
}
```
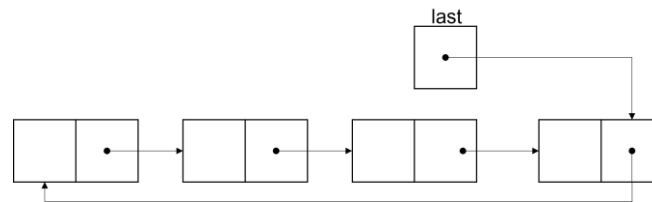
2. An illustration of a circular linked list is shown below. Rather than having a pointer to the first node in the list, we use a pointer called last to point to the last node in the list.

The incomplete header file for the circular linked list is provided below:

```cpp
// A class template for holding a circular linked list.
#ifndef CIRCLIST_H
#define CIRCLIST_H
#include <iostream>      // For cout
using namespace std;

template <class T>
class CircList
{
private:
    // Declare a structure for the list
    struct ListNode
    {
        T value;                 // The value in this node
        ListNode *next;  // To point to the next node
    };

    ListNode *last;   // Pointer for last node in the circular linked list

public:
    // Constructor
    CircList()
        { last = nullptr; }

    // Destructor
    ~CircList();

    // Linked list operations
    void appendNode(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 CircList<T>::appendNode(T newValue)
{
    //Incomplete function
}
```

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

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

   // Position nodePtr at the first of the list.
   nodePtr = last->next;

   // While nodePtr points to a node, traverse
   // the list.
   do
   {
      // Display the value in this node.
      cout << nodePtr->value << endl;
      // Move to the next node.
      nodePtr = nodePtr->next;
   } while (nodePtr != last->next);
}




//*******************************************************
// 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 CircList<T>::deleteNode(T searchValue)
{
   //Incomplete code
}

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

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

   // Position nodePtr at the first of the list.
   nodePtr = last->next;

   // If there is only one node in the list
   if (nodePtr == last)
      delete nodePtr;

   else {
   // While nodePtr is not at the end of the list...
      do
      {
         // Save a pointer to the next node.
         nextNode = nodePtr->next;

         // Delete the current node.
         delete nodePtr;
```

```
        // Position nodePtr at the next node.
        nodePtr = nextNode;
    } while (nodePtr!=last);
    delete last;
    }
}

#endif
```

(a) How would assign the address of the first node to a pointer called `nodePtr`?

```
nodePtr = last->next;
```

(b) The following is the implementation of the appendNode function for a regular linked list. Modify the function so that a new node will be appended to the end of the circular linked list shown above.

```cpp
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;
    newNode->value = newValue;
    newNode->next = nullptr;

    // 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;
    }
}

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

    // Allocate a new node and store newValue there.
    newNode = new ListNode;
    newNode->value = newValue;
    newNode->next = nullptr;

    // If there are no nodes in the list
    // make newNode the first node.
    if (!last) {
            last = newNode;
            newNode->next = newNode;
    }
    else  // Otherwise, insert newNode at end.
    {
```

```
            // Initialize nodePtr as the last node in the list.
            nodePtr = last;

            // Insert newNode as the last node, point to the first node
            newNode->next = last->next;
            nodePtr->next = newNode;
            last = newNode;
    }
}
```

(c) Complete the `deleteNode` function. Note that you will need to take the following
scenarios into consideration:
- Empty list which has nothing to delete
- The first node is the one to be deleted
- The last node is the one to be deleted
- The node to be deleted is neither the first nor last node

Test your code with the following main function:

```
int main() {
    //Create two different linked lists, one for int, one for double
    CircList<int> list1;

    list1.appendNode(2);
    list1.appendNode(3);
    list1.displayList();
    cout << "After deleting first node\n";
    list1.deleteNode(2);
    list1.displayList();
    cout << "---\n";

    list1.appendNode(4);
    list1.appendNode(5);
    list1.displayList();
    cout << "After deleting middle node\n";
    list1.deleteNode(4);
    list1.displayList();
    cout << "---\n";

    list1.appendNode(6);
    list1.appendNode(7);
    list1.displayList();
    cout << "After deleting last node\n";
    list1.deleteNode(7);
    list1.displayList();
    cout << "---\n";

    return 0;
}
```

Your output should be as follows:

```
2
3
After deleting first node
3
---
3
4
5
After deleting middle node
3
5
---
```

```
3
5
6
7
After deleting last node
3
5
6
```

```cpp
template <class T>
void CircList<T>::deleteNode(T searchValue)
{
    ListNode *nodePtr;          // To traverse the list
    ListNode *previousNode;

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

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

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

        //If the node to be deleted is the last one.
        if (nodePtr==last)
        {
            previousNode->next = nodePtr->next;
            last = previousNode;
            delete nodePtr;
        }
        // If nodePtr is not at the end of the list,
        // link the previous node to the node after
        // nodePtr, then delete nodePtr.
        else if (nodePtr!=last->next) //If nodePtr is first, then the target
value does not exist
        {
            previousNode->next = nodePtr->next;
            delete nodePtr;
        }
    }
}
```