Submitted By

Umme Kalsoom

Submitted To

Sir Azib mehmood

Roll No

22011556-014

Section

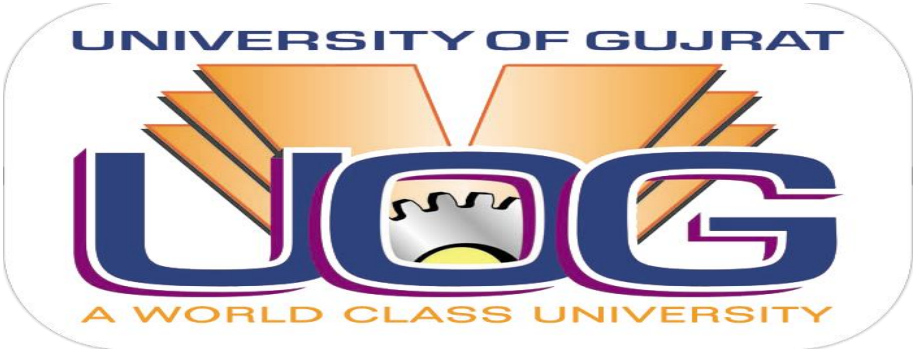
A

Course Title

Data structure and algorithm

Course Code

IT 209



```
#include <iostream>
#include <limits>  // For input validation
using namespace std;
// Define a Node structure struct
Node
 {  int data;       // Data of the node
    Node* next;     // Pointer to the next node
};
// Class for Linked List operations
class LinkedList
{ private:
    Node* headnode;     // Head node of the linked list
    Node* lastnode;     // Last node of the linked list
public:
    // Constructor to initialize an empty linked list
LinkedList()
 {
       headnode = NULL;
lastnode = NULL;
    }
    // Function to insert a node at the end
void insertAtEnd(int value)
{
 // Create a new node
  Node* tempnode = new Node;
tempnode->data = value;
tempnode->next = NULL;
```

```
      // If the list is empty, set the new node as the head and last node
if (headnode == NULL)
{     headnode = tempnode;
     lastnode = tempnode;
      }
Else
{
         // Otherwise, add the new node to the end and update the last node
lastnode->next = tempnode;
  lastnode = tempnode;
      }
   }
   // Function to display the linked list using a while loop
void displayList()
{
      Node* currentnode = headnode;
      // Traverse the list and print each node's data using a while loop
while (currentnode != NULL)
{
cout << currentnode->data << " ";
  currentnode = currentnode->next;
      }
      cout << endl;
   }
   // Function to search for a specific value in the linked list
bool search(int value)
{
```

```
        Node* currentnode = headnode;
        // Traverse the list to find the value
        while (currentnode != NULL)
 {
  if (currentnode->data == value)
{
  return true;  // Value found
    }
  currentnode = currentnode->next;
    }
   return false;  // Value not found
    }
    // Function to update the value at a specific position
void updateAtPosition(int position, int newValue)
 {
        Node* currentnode = headnode;
        // Traverse to the specified position
        int i = 0;
        while (i < position && currentnode != NULL)
 {
 currentnode = currentnode->next;
  i++;
        }
        // If the position is valid, update the value
    if (currentnode != NULL) {
           currentnode->data = newValue;
     cout << "Value updated at index " << position << endl;
        }
```

```
else
 {
          cout << "Invalid index!" << endl;
       }
   }
   // Function to delete a node from the beginning
void deleteFromBeginning() {
       // If the list is empty, do nothing
     if (headnode == NULL) {
 cout << "List is empty. Nothing to delete." << endl;
          return;
       }

       // Otherwise, delete the first node and update the head
Node* tempnode = headnode;
 headnode = headnode->next;
 delete tempnode;
   cout << "Deleted from the beginning." << endl;
   }
   // Function to delete a node from the end
void deleteFromEnd() {
       // If the list is empty, do nothing
   if (headnode == NULL) {
        cout << "List is empty. Nothing to delete." << endl;
          return;
       }
       // If there's only one node, delete it and update head and lastnode
if (headnode == lastnode) {
  delete headnode;
```

```
        headnode = NULL;
        lastnode = NULL;
            cout << "Deleted from the end." << endl;
            return;
        }
        // Otherwise, traverse to the second last node
        Node* previousnode = NULL;
Node* currentnode = headnode;
while (currentnode->next != NULL) {
previousnode = currentnode;
currentnode = currentnode->next;
        }
        // Delete the last node and update the lastnode
delete currentnode;
previousnode->next = NULL;
 lastnode = previousnode;
 cout << "Deleted from the end." << endl;
    }
    // Function to delete a node from a specific position
void deleteFromPosition(int position) {
  // If the list is empty, do nothing
     if (headnode == NULL) {
     cout << "List is empty. Nothing to delete." << endl;
            return;
        }
        // If the position is 0, delete from the beginning if
        (position == 0) {
            deleteFromBeginning();
```

```
            return;
        }
        // Traverse to the specified position
        Node* previousnode = NULL;
Node* currentnode = headnode;
        int i = 0;
        while (i < position && currentnode != NULL) {
previousnode = currentnode;
 currentnode = currentnode->next;
            i++;
        }

        // If the position is valid, delete the node and update the pointers
if (currentnode != NULL) {
previousnode->next = currentnode->next;
delete currentnode;
 cout << "Deleted from index " << position << endl;
        } else {
            cout << "Invalid index!" << endl;
        }
    }
    // Function to insert a node at the beginning
void insertAtBeginning(int value) {
        // Create a new node
        Node* tempnode = new Node;

         tempnode->data = value;

         tempnode->next = headnode;
```

```
        // Update the head to the new node
headnode = tempnode;
        cout << "Inserted at the beginning." << endl;
    }
    // Function to insert a node at a specific position
void insertAtPosition(int position, int value) {
// If the position is 0, insert at the beginning
  if (position == 0) {
insertAtBeginning(value);
 return;
        }
        // Traverse to the specified position
        Node* previousnode = NULL;
Node* currentnode = headnode;
        int i = 0;
        while (i < position && currentnode != NULL) {
previousnode = currentnode;
   currentnode = currentnode->next;
            i++;
        }
        // Create a new node and insert it at the specified position
        Node* tempnode = new Node;

        tempnode->data = value;

        tempnode->next = currentnode;
        previousnode->next = tempnode;
        cout << "Inserted at index " << position << endl;
    }
};
```

```
int main() {
    LinkedList myList;


    int numNodes;
  cout << "Enter the number of nodes you want in the linked list: ";
    // Input validation to handle non-numeric inputs for the number of nodes
 while (!(cin >> numNodes)) {
  cout << "Invalid Numeric Entered! Please enter a number: ";
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
    }
    // Take user input for nodes
 for (int i = 0; i < numNodes; ++i) {
        int value;
     cout << "Enter the value for node " << i << ": ";
        // Input validation to handle non-numeric inputs for node values
        while (!(cin >> value)) {
            cout << "Invalid Numeric Entered! Please enter a number: ";
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
        }
        myList.insertAtEnd(value);
    }
    myList.displayList();  // Display the linked list after inserting nodes
     int choice;     do {
    // Display menu        cout <<
 "\nMenu:\n";
 cout << "1. Search\n";
```

```
 cout << "2. Insert at beginning\n";
cout << "3. Insert at end\n";
 cout << "4. Delete from beginning\n";
cout << "5. Delete from end\n";
  cout << "6. Insert at any index\n";
cout << "7. Delete from any index\n";
cout << "8. Update at any index\n";
cout << "9. Exit\n";
 cout << "Enter your choice: ";
       // Input validation to handle non-numeric inputs for user choice
       while (!(cin >> choice)) {
 cout << "Invalid Numeric Entered! Please enter a number: ";
          cin.clear();
          cin.ignore(numeric_limits<streamsize>::max(), '\n');
       }
       switch (choice) {
      case 1:
{
  int value;
cout << "Enter the value to search: ";
             // Input validation to handle non-numeric inputs for search value
             while (!(cin >> value)) {
                cout << "Invalid Numeric Entered! Please enter a number: ";
                cin.clear();
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
             }
             if (myList.search(value)) {
 cout << "Value " << value << " found in the list." << endl;
```

```
            }
else
 {
                cout << "Value " << value << " not found in the list." << endl;
            }
break;
         }
          case 2:
{              int value;
  cout << "Enter the value to insert at the beginning: ";
            // Input validation to handle non-numeric inputs for insertion value
            while (!(cin >> value)) {
                cout << "Invalid Numeric Entered! Please enter a number: ";
                cin.clear();
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
            }
            myList.insertAtBeginning(value);
myList.displayList();
  break;
         }
         case 3:
 {
    int value;
     cout << "Enter the value to insert at the end: ";
 // Input validation to handle non-numeric inputs for insertion value
while (!(cin >> value)) {
cout << "Invalid Numeric Entered! Please enter a number: ";
cin.clear();
```

```
cin.ignore(numeric_limits<streamsize>::max(), '\n');
 }
 myList.insertAtEnd(value);
myList.displayList();
 break;
        }
        case 4:
           myList.deleteFromBeginning();
           myList.displayList();
break;

        case 5:
 myList.deleteFromEnd();
myList.displayList();
 break;

        case 6:
{
 int index, value;
           cout << "Enter the index where you want to insert: ";
           // Input validation to handle non-numeric inputs for index
           while (!(cin >> index)) {
 cout << "Invalid Numeric Entered! Please enter a number: ";
 cin.clear();
 cin.ignore(numeric_limits<streamsize>::max(), '\n');
   }

  cout << "Enter the value to insert: ";
  // Input validation to handle non-numeric inputs for insertion value
```

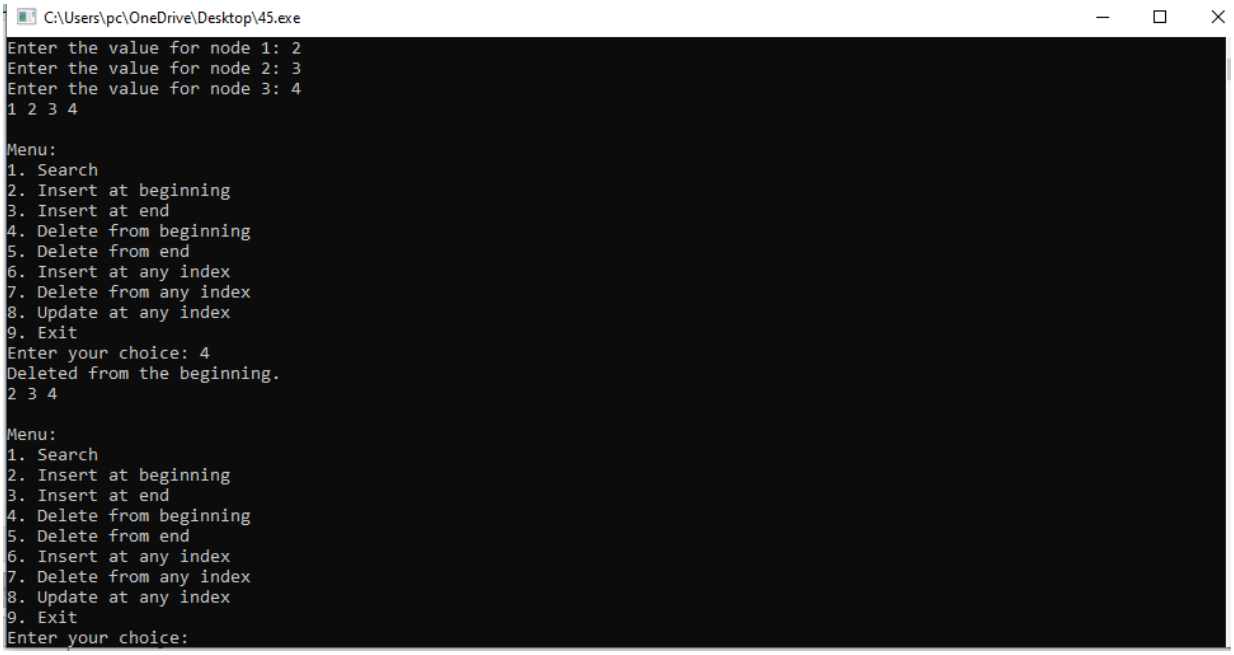
```
            while (!(cin >> value)) {
             cout << "Invalid Numeric Entered! Please enter a number: ";
                cin.clear();
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
            }
            myList.insertAtPosition(index, value);
myList.displayList();
 break;
        }
        case 7:
{
 int index;
            cout << "Enter the index from where you want to delete: ";
            // Input validation to handle non-numeric inputs for index
while (!(cin >> index)) {
                cout << "Invalid Numeric Entered! Please enter a number: ";
                cin.clear();
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
            }
            myList.deleteFromPosition(index);
            myList.displayList();
break;
        }
        case 8:
{
   int index, value;
            cout << "Enter the index where you want to update: ";
            // Input validation to handle non-numeric inputs for index
```

```
            while (!(cin >> index)) {
 cout << "Invalid Numeric Entered! Please enter a number: ";
                cin.clear();
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
            }
            cout << "Enter the new value: ";
            // Input validation to handle non-numeric inputs for new value
while (!(cin >> value)) {
                cout << "Invalid Numeric Entered! Please enter a number: ";
                cin.clear();
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
            }
            myList.updateAtPosition(index, value);
myList.displayList();
          break;
          }

          case 9:
             cout << "Exiting the program.\n";
             break;
default:
             cout << "Invalid choice. Please enter a valid option.\n";
       }
    }
 while (choice != 9);

    return 0;
}
```

Output:


C:\Users\pc\OneDrive\Desktop\45.exe
Enter the value for node 1: 2
Enter the value for node 2: 3
Enter the value for node 3: 4
1 2 3 4
Menu:
1. Search
2. Insert at beginning
3. Insert at end
4. Delete from beginning
5. Delete from end
6. Insert at any index
7. Delete from any index
8. Update at any index
9. Exit
Enter your choice: 4
Deleted from the beginning.
2 3 4
Menu:
1. Search
2. Insert at beginning
3. Insert at end
4. Delete from beginning
5. Delete from end
6. Insert at any index
7. Delete from any index
8. Update at any index
9. Exit
Enter your choice: