- **Ques 1**

```cpp
class Node {
public:
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

class LinkedList {
public:
    Node* head;
    LinkedList() : head(nullptr) {}

    void insertAtEnd(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next) temp = temp->next;
        temp->next = newNode;
    }

    // 1. Delete the second node
    void deleteSecondNode() {
        if (!head || !head->next) {
            cout << "Exception: List has less than 2 nodes." << endl;
            return;
        }
        Node* second = head->next;
        head->next = second->next;
        delete second;
    }

    // 2. Delete the second last node
```

```
11  class LinkedList {
38      // 2. Delete the second last node
39      void deleteSecondLastNode() {
40          if (!head || !head->next) {
41              cout << "Exception: List has less than 2 nodes." << endl;
42              return;
43          }
44          Node* temp = head;
45          Node* prev = nullptr;
46          while (temp->next && temp->next->next) {
47              prev = temp;
48              temp = temp->next;
49          }
50          if (!prev) {
51              // Only 2 nodes
52              head = head->next;
53          }
54          else {
55              prev->next = temp->next;
56          }
57          delete temp;
58      }
59
60      // 3. Swap two nodes
```

Output

```
Original list: 1 2 3 4 5
After deleting second node: 1 3 4 5
After deleting second last node: 1 3 5
Exception: One or both nodes not found.
After swapping nodes 1 and 4: 1 3 5
After reversing the list: 5 3 1
```

```
class LinkedList {
    // 3. Swap two nodes
    void swapNodes(int x, int y) {
        if (x == y) return;

        Node* prevX = nullptr, * currX = head;
        while (currX && currX->data != x) {
            prevX = currX;
            currX = currX->next;
        }

        Node* prevY = nullptr, * currY = head;
        while (currY && currY->data != y) {
            prevY = currY;
            currY = currY->next;
        }

        if (!currX || !currY) {
            cout << "Exception: One or both nodes not found." << endl;
            return;
        }

        if (prevX) prevX->next = currY;
        else head = currY;

        if (prevY) prevY->next = currX;
        else head = currX;

        Node* temp = currX->next;
        currX->next = currY->next;
        currY->next = temp;
    }

    // 4. Reverse the list
```

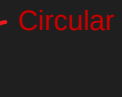
```
    // 4. Reverse the list
    void reverseList() {
        Node* prev = nullptr;
        Node* current = head;
        Node* next = nullptr;
        while (current) {
            next = current->next;
            current->next = prev;
            prev = current;
            current = next;
        }
        head = prev;
    }

    void printList() {
        Node* temp = head;
        while (temp) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};
```

```
116  int main() {
117      LinkedList list;
118      list.insertAtEnd(1);
119      list.insertAtEnd(2);
120      list.insertAtEnd(3);
121      list.insertAtEnd(4);
122      list.insertAtEnd(5);
123
124      cout << "Original list: ";
125      list.printList();
126
127      // Delete second node
128      list.deleteSecondNode();
129      cout << "After deleting second node: ";
130      list.printList();
131
132      // Delete second last node
133      list.deleteSecondLastNode();
134      cout << "After deleting second last node: ";
135      list.printList();
136
137      // Swap two nodes
138      list.swapNodes(1, 4);
139      cout << "After swapping nodes 1 and 4: ";
140      list.printList();
141
142      // Reverse the list
143      list.reverseList();
144      cout << "After reversing the list: ";
145      list.printList();
146
147      return 0;
148  }
```

- **Ques 2**

```cpp
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int val) : data(val), next(nullptr) {}
};

class CircularLinkedList {
```

```cpp
class CircularLinkedList {
private:
    Node* head;

public:
    CircularLinkedList() : head(nullptr) {}

    // Function to append data to the list
    void append(int data) {
        Node* newNode = new Node(data);
        if (head == nullptr) {
            head = newNode;
            head->next = head; // Circular link
        }
        else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head; // Maintain circular structure
        }
    }
```



```
36      // Function to delete the second node
37      void deleteSecondNode() {
38          if (head == nullptr || head->next == head) {
39              cout << "Exception: List is too small or empty!" << endl;
40              return;
41          }
42          Node* second = head->next;
43          head->next = second->next;
44          delete second;
45      }
46
```

```
class CircularLinkedList {
    // Function to delete the second last node
    void deleteSecondLastNode() {
        // Check if the list has less than 2 nodes
        if (!head || head->next == head) {
            cout << "Exception: List has less than 2 nodes." << endl;
            return;
        }

        Node* temp = head;
        Node* prev = nullptr;

        // Traverse the list to find the second last node
        do {
            prev = temp;
            temp = temp->next;
        } while (temp->next->next != head);

        // Case where the list has only 2 nodes
        if (prev == head && head->next == head) {
            // Set head to the last node since we are deleting the second node
            head = temp;
            head->next = head; // maintain circular structure
        }
        else {
            prev->next = temp->next; // Skip over the second last node
        }

        delete temp; // Free the memory of the second last node
    }
```

```
77      // Function to swap two nodes
78      void swapNodes(int val1, int val2) {
79          if (val1 == val2) {
80              cout << "The nodes have the same value, no need to swap." << endl;
81              return;
82          }
83
84          Node* prev1 = nullptr, * curr1 = head;
85          Node* prev2 = nullptr, * curr2 = head;
86
87          // Find the first node with value val1
88          while (curr1 && curr1->data != val1) {
89              prev1 = curr1;
90              curr1 = curr1->next;
91          }
92
93          // Find the second node with value val2
94          while (curr2 && curr2->data != val2) {
95              prev2 = curr2;
96              curr2 = curr2->next;
97          }
98
99          if (curr1 == nullptr || curr2 == nullptr) {
00              cout << "Exception: One or both nodes not found!" << endl;
01              return;
02          }
03
04          // Swap the previous nodes' next pointers
05          if (prev1) prev1->next = curr2;
06          else head = curr2;
07
08          if (prev2) prev2->next = curr1;
```

```
99          if (curr1 == nullptr || curr2 == nullptr) {
00              cout << "Exception: One or both nodes not found!" << endl;
01              return;
02          }
03
04          // Swap the previous nodes' next pointers
05          if (prev1) prev1->next = curr2;
06          else head = curr2;
07
08          if (prev2) prev2->next = curr1;
09          else head = curr1;
10
11          // Swap next pointers
12          Node* temp = curr1->next;
13          curr1->next = curr2->next;
14          curr2->next = temp;
15      }
16
17      // Function to reverse the list
```

```
118     void reverseList() {
119         if (head == nullptr || head->next == head) {
120             cout << "Exception: List is too small or empty!" << endl;
121             return;
122         }
123 
124         Node* prev = nullptr;
125         Node* current = head;
126         Node* next = nullptr;
127 
128         Node* tail = head;
129 
130         // Reverse the list until it comes back to the head
131         do {
132             next = current->next;
133             current->next = prev;
134             prev = current;
135             current = next;
136         } while (current != head);
137 
138         // Adjust the head and tail pointers
139         head->next = prev;
140         head = prev;
141     }
142 
```

```
    // Function to print the list
    void printList() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }
        Node* temp = head;
        do {
            cout << temp->data << " ";
            temp = temp->next;
        } while (temp != head);
        cout << endl;
    }
};
int main() {
```

```cpp
int main() {
    CircularLinkedList list;

    list.append(1);
    list.append(2);
    list.append(3);
    list.append(4);
    list.append(5);
    list.append(6);
    list.append(7);
    list.append(8);
    list.append(9);

    cout << "Original List: ";
    list.printList();

    // Delete second node
    list.deleteSecondNode();
    cout << "After deleting second node: ";
    list.printList();

    // Delete second last node
    list.deleteSecondLastNode();
    cout << "After deleting second last node: ";
    list.printList();

    // Swap two nodes
    list.swapNodes(3, 5);
    cout << "After swapping 3 and 5: ";
    list.printList();

    // Reverse the list
    list.reverseList();
    cout << "After reversing the list: ";
    list.printList();

    return 0;
}
```

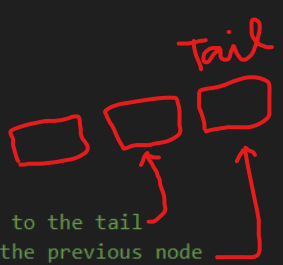
```
Original List: 1 2 3 4 5 6 7 8 9
After deleting second node: 1 3 4 5 6 7 8 9
After deleting second last node: 1 3 4 5 6 7 9
After swapping 3 and 5: 1 5 4 3 6 7 9
After reversing the list: 9 7 6 3 4 5 1
```

- **Ques 3**

```cpp
//Suppose we have a doubly linked linear list with a pointer to the head and tail nodes.
// ...
//Apply the following :
//1. Insert an item to maintain a sorted list.
//2. Delete the second last node of the list.
//3. Delete all occurrences of an item from the list.

#include <iostream>
using namespace std;

// Definition of a Node in the doubly linked list
struct Node {
    int data;      // Data stored in the node
    Node* prev;    // Pointer to the previous node
    Node* next;    // Pointer to the next node

    // Constructor to initialize the node with given value
    Node(int val) {
        data = val;
        prev = nullptr;
        next = nullptr;
    }
};

// Doubly Linked List class to manage the list
class DoublyLinkedList {
public:
    Node* head;    // Pointer to the first node in the list
    Node* tail;    // Pointer to the last node in the list

    // Constructor to initialize an empty list
    DoublyLinkedList() {
        head = nullptr;
        tail = nullptr;
    }
```

```cpp
class DoublyLinkedList {

    // Function to insert a new node into the list in a sorted manner
    void insertSorted(int data) {
        Node* newNode = new Node(data);  // Create a new node with the given data

        // If the list is empty, insert the first node
        if (head == nullptr) {
            head = tail = newNode;
        }
        // If the new data is smaller than the first node, insert at the beginning
        else if (data <= head->data) {
            newNode->next = head;     // Link new node to the current head
            head->prev = newNode;     // Link the current head back to the new node
            head = newNode;           // Update the head pointer
        }
        // If the new data is greater than the last node, insert at the end
        else if (data >= tail->data) {
            tail->next = newNode;     // Link current tail to the new node
            newNode->prev = tail;     // Link the new node back to the current tail
            tail = newNode;           // Update the tail pointer
        }
        // Otherwise, insert the new node in the correct sorted position
        else {
            Node* temp = head;
            // Traverse the list to find the right position
            while (temp != nullptr && temp->data < data) {
                temp = temp->next;
            }
            // Insert the new node between two nodes
            newNode->next = temp;         // New node points to the next node
            newNode->prev = temp->prev;   // New node points to the previous node
            temp->prev->next = newNode;   // Previous node points to the new node
            temp->prev = newNode;         // Next node's previous pointer updated
        }
    }

```

```cpp
class DoublyLinkedList {

    // Function to delete the second last node of the list
    void deleteSecondLast() {
        // Check if the list has less than 2 nodes
        if (head == nullptr || head == tail) {
            cout << "List is too short to delete the second last node." << endl;
            return;
        }

        Node* secondLast = tail->prev;  // Find the second last node

        // If the list has only two nodes, delete the head
        if (secondLast == head) {
            head = tail;  // Update head to point to the last node
            delete secondLast;  // Delete the second last node
        }
        // Otherwise, unlink and delete the second last node
        else {
            secondLast->prev->next = tail;  // Link the previous node to the tail
            tail->prev = secondLast->prev;  // Link the tail back to the previous node
            delete secondLast;  // Delete the second last node
        }
    }

```



```
26  class DoublyLinkedList {
95      // Function to delete all occurrences of a specific value from the list
96      void deleteAllOccurrences(int item) {
97          Node* temp = head;
98          // Traverse the list and delete matching nodes
99          while (temp != nullptr) {
100             if (temp->data == item) {
101                 Node* nodeToDelete = temp;  // Store the node to delete
102
103                 // If the node is at the head
104                 if (temp == head) {
105                     head = head->next;      // Move head to the next node
106                     if (head) head->prev = nullptr;  // Update the new head's previous pointer
107                 }
108                 // If the node is at the tail
109                 else if (temp == tail) {
110                     tail = tail->prev;      // Move tail to the previous node
111                     if (tail) tail->next = nullptr;  // Update the new tail's next pointer
112                 }
113                 // If the node is in the middle
114                 else {
115                     temp->prev->next = temp->next;   // Link previous node to the next node
116                     temp->next->prev = temp->prev;   // Link next node back to the previous node
117                 }
118
119                 temp = temp->next;   // Move to the next node
120                 delete nodeToDelete; // Delete the current node
121             }
122             else {
123                 temp = temp->next;   // Move to the next node if no match
124             }
125         }
126     }
127
```

```
26  class DoublyLinkedList {
128     // Function to print the list
129     void printList() {
130         Node* temp = head;  // Start from the head
131         while (temp != nullptr) {
132             cout << temp->data << " ";  // Print the data
133             temp = temp->next;  // Move to the next node
134         }
135         cout << endl;
136     }
137 };
138 
139 int main() {
140     DoublyLinkedList dll;
141 
142     // Insert some values to maintain sorted order
143     dll.insertSorted(5);
144     dll.insertSorted(3);
145     dll.insertSorted(7);
146     dll.insertSorted(2);
147     dll.insertSorted(6);
148      // Print the list after inserting values
149     cout << "List after sorted insertions: ";
150     dll.printList();
151 
152     // Delete the second last node
153     dll.deleteSecondLast();
154     cout << "List after deleting second last node: ";
155     dll.printList();
156     // Insert duplicate values to demonstrate deletion of all occurrences
157     dll.insertSorted(5);
158     dll.insertSorted(5);
159     cout << "List after inserting duplicates: ";
160     dll.printList();
161     // Delete all occurrences of value 5
162     dll.deleteAllOccurrences(5);
163     cout << "List after deleting all occurrences of 5: ";
164     dll.printList();
165 
166     return 0;
```

```
List after sorted insertions: 2 3 5 6 7
List after deleting second last node: 2 3 5 7
List after inserting duplicates: 2 3 5 5 5 7
List after deleting all occurrences of 5: 2 3 7
```

- **Ques 4**

```cpp
#include <iostream>
using namespace std;

// Node structure for the doubly circular linked list
class Node {
public:
    int data;        // Data stored in the node
    Node* prev;      // Pointer to the previous node
    Node* next;      // Pointer to the next node

    // Constructor to create a new node
    Node(int val) {
        data = val;
        prev = nullptr;
        next = nullptr;
    }
};

// Doubly Circular Linked List class
class DoublyCircularLinkedList {
public:
    Node* head;    // Pointer to the first node (head)
    Node* tail;    // Pointer to the last node (tail)

    // Constructor to initialize an empty list
    DoublyCircularLinkedList() {
        head = nullptr;
        tail = nullptr;
    }

```

```
20  class DoublyCircularLinkedList {
32      void insertSorted(int data) {
33          Node* newNode = new Node(data);  // Create a new node with the given data
34
35          // Case 1: List is empty
36          if (head == nullptr) {
37              head = tail = newNode;       // The new node is both head and tail
38              head->next = head;           // Point to itself (circular)
39              head->prev = head;           // Circular doubly linked list
40          }
41          // Case 2: Insert at the beginning (new data is smaller than head)
42          else if (data <= head->data) {
43              newNode->next = head;        // New node points to current head
44              newNode->prev = tail;        // New node points to current tail
45              head->prev = newNode;        // Head points back to new node
46              tail->next = newNode;        // Tail points to new node (circular link)
47              head = newNode;              // Update head to new node
48          }
49          // Case 3: Insert at the end (new data is greater than tail)
50          else if (data >= tail->data) {
51              newNode->next = head;        // New node points to head (circular)
52              newNode->prev = tail;        // New node points to current tail
53              tail->next = newNode;        // Tail points to new node
54              head->prev = newNode;        // Head points back to new node
55              tail = newNode;              // Update tail to new node
56          }
57          // Case 4: Insert in the middle
58          else {
59              Node* temp = head;
60              // Traverse the list to find the correct position to insert
61              while (temp->data < data) {
62                  temp = temp->next;
63              }
64              // Insert the new node between temp->prev and temp
65              newNode->next = temp;
66              newNode->prev = temp->prev;
67              temp->prev->next = newNode;
68              temp->prev = newNode;
69          }
70      }
```

```
20  class DoublyCircularLinkedList {
73      void deleteSecondLast() {
74          // If the list is empty or has less than 2 nodes, return
75          if (head == nullptr || head->next == head) {
76              cout << "Cannot delete second last node, not enough nodes." << endl;
77              return;
78          }
79
80          Node* secondLast = tail->prev;    // Second last node
81          // If the list has only two nodes, we delete the head
82          if (secondLast == head) {
83              head = tail;                  // Head becomes the tail
84              head->next = head;            // Update the circular link
85              head->prev = head;
86              delete secondLast;            // Delete the original head
87          }
88          else {
89              secondLast->prev->next = tail;  // Link previous node to tail
90              tail->prev = secondLast->prev;  // Link tail back to previous node
91              delete secondLast;              // Delete the second last node
92          }
93      }
94
95      // 3. Delete all occurrences of a specific item from the list
```

```
20  class DoublyCircularLinkedList {
95      // 3. Delete all occurrences of a specific item from the list
96      void deleteAllOccurrences(int item) {
97          if (head == nullptr) return;  // If the list is empty, do nothing
98
99          Node* current = head;
100         Node* toDelete = nullptr;
101
102         // Loop through the list and delete all occurrences of the item
103         do {
104             if (current->data == item) {
105                 toDelete = current;
106
107                 // Case 1: Only one node in the list
108                 if (head == tail && current == head) {
109                     head = tail = nullptr;
110                     delete toDelete;
111                     return;
112                 }
113                 // Case 2: Node to be deleted is the head
114                 else if (current == head) {
115                     head = head->next;
116                     head->prev = tail;
117                     tail->next = head;
118                 }
119                 // Case 3: Node to be deleted is the tail
120                 else if (current == tail) {
121                     tail = tail->prev;
122                     tail->next = head;
123                     head->prev = tail;
124                 }
125                 // Case 4: Node to be deleted is in the middle
126                 else {
127                     current->prev->next = current->next;
128                     current->next->prev = current->prev;
129                 }
130
131                 current = current->next;  // Move to the next node
132                 delete toDelete;          // Delete the current node
133
131                 current = current->next;  // Move to the next node
132                 delete toDelete;          // Delete the current node
133
134             }
135             else {
136                 current = current->next;  // Move to the next node
137             }
138
139         } while (current != head);  // Continue until we loop back to the head
140     }
141
142     // Function to print the list
143     void printList() {
144         if (head == nullptr) {       // If the list is empty
145             cout << "List is empty" << endl;
146             return;
147         }
148
149         Node* temp = head;
150         do {
151             cout << temp->data << " <-> ";  // Print the current node's data
152             temp = temp->next;              // Move to the next node
153         } while (temp != head);             // Continue until back to head
154         cout << " (back to head)" << endl;
155     }
156 };
```

```cpp
int main() {
    DoublyCircularLinkedList dcll;  // Create a doubly circular linked list

    // Insert elements into the list while keeping it sorted
    dcll.insertSorted(10);
    dcll.insertSorted(5);
    dcll.insertSorted(20);
    dcll.insertSorted(15);
    dcll.insertSorted(7);

    // Print the current list
    cout << "List after sorted insertions: ";
    dcll.printList();

    // Delete the second last node
    dcll.deleteSecondLast();
    cout << "List after deleting second last node: ";
    dcll.printList();

    // Insert duplicate values to test deletion of all occurrences
    dcll.insertSorted(5);
    dcll.insertSorted(20);
    cout << "List after inserting duplicates: ";
    dcll.printList();

    // Delete all occurrences of 5
    dcll.deleteAllOccurrences(5);
    cout << "List after deleting all occurrences of 5: ";
    dcll.printList();

    // Delete all occurrences of 20
    dcll.deleteAllOccurrences(20);
    cout << "List after deleting all occurrences of 20: ";
    dcll.printList();

    return 0;
}
```

```
List after sorted insertions: 5 <-> 7 <-> 10 <-> 15 <-> 20 <->  (back to head)
List after deleting second last node: 5 <-> 7 <-> 10 <-> 20 <->  (back to head)
List after inserting duplicates: 5 <-> 5 <-> 7 <-> 10 <-> 20 <-> 20 <->  (back to head
    )
List after deleting all occurrences of 5: 5 <-> 7 <-> 10 <-> 20 <-> 20 <->  (back to
    head)
List after deleting all occurrences of 20: 5 <-> 7 <-> 10 <->  (back to head)
```