- **Singly Linked List**

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

template <typename T>
class Node {
private:
    T data;
    Node* next;

public:
    // Constructor
    Node(T element) : data(element), next(nullptr) {}

    // Setters
    void setData(T element) {
        data = element;
    }

    void setNext(Node* node) {
        next = node;
    }

    // Getters
    T getData() {
        return data;
    }

    Node* getNext() {
        return next;
    }
};


template <typename T>
class List {
private:
    Node<T>* head;

public:
    // Constructor
    List() : head(nullptr) {}

    // Insert at the beginning
    void InsertBeginning(Node<T>* pNew) {
        pNew->setNext(head);
        head = pNew;
    }

    // Insert at the end
    void InsertEnd(Node<T>* pNew) {
        if (head == nullptr) {
            head = pNew;
        }
        else {
            Node<T>* temp = head;
            while (temp->getNext() != nullptr) {
                temp = temp->getNext();
            }
            temp->setNext(pNew);
        }
    }

```

```
62.     // Insert in the middle (after pBefore)
63.     void InsertMiddle(Node<T>* pBefore, Node<T>* pNew) {
64.         if (pBefore == nullptr) {
65.             head = pNew;
66.         }
67.         else {
68.             pNew->setNext(pBefore->getNext());
69.             pBefore->setNext(pNew);
70.         }
71.     }
72.
73.     // Delete from the beginning
74.     void DeleteFromBeginning() {
75.         if (head == nullptr) {
76.             cout << "List is empty" << endl;
77.             return;
78.         }
79.         Node<T>* temp = head;
80.         head = head->getNext();
81.         delete temp;
82.     }
83.
84.     // Delete from the end
85.     void DeleteFromEnd() {
86.         if (head == nullptr) {
87.             cout << "List is empty" << endl;
88.             return;
89.         }
90.         if (head->getNext() == nullptr) {
91.             delete head;
92.             head = nullptr;
93.         }
94.         else {
95.             Node<T>* temp = head;
96.             while (temp->getNext()->getNext() != nullptr) {
97.                 temp = temp->getNext();
98.             }
99.             delete temp->getNext();
100.            temp->setNext(nullptr);
101.        }
102.    }
103.
104.    // Delete from the middle using a specific node
105.    void DeleteFromMiddle(Node<T>* pToBeDeleted) {
106.        if (pToBeDeleted == head) {
107.            DeleteFromBeginning();
108.        }
109.        else {
110.            Node<T>* temp = head;
111.            while (temp->getNext() != pToBeDeleted) {
112.                temp = temp->getNext();
113.            }
114.            temp->setNext(pToBeDeleted->getNext());
115.            delete pToBeDeleted;
116.        }
117.    }
118.
119.    // Delete the middle node using the slow and fast pointer technique
120.    void DeleteFromMiddle() {
121.        if (head == nullptr || head->getNext() == nullptr) {
122.            DeleteFromBeginning();
123.            return;
124.        }
125.
126.        Node<T>* slow = head;
```

```
        Node<T>* fast = head;
        Node<T>* prev = nullptr;

        // Traverse the list with fast and slow pointers
        while (fast != nullptr && fast->getNext() != nullptr) {
            prev = slow;
            slow = slow->getNext();
            fast = fast->getNext()->getNext();
        }

        // 'slow' is the middle node, and 'prev' is the node before the middle node
        if (prev != nullptr) {
            prev->setNext(slow->getNext());
            delete slow;
        }
    }

    // Delete the minimum value node
    void DeleteMinimum() {
        if (head == nullptr) return;

        Node<T>* minNode = head;
        Node<T>* temp = head;
        Node<T>* prev = nullptr;
        Node<T>* prevMin = nullptr;

        while (temp != nullptr) {
            if (temp->getData() < minNode->getData()) {
                prevMin = prev;
                minNode = temp;
            }
            prev = temp;
            temp = temp->getNext();
        }

        if (minNode == head) {
            DeleteFromBeginning();
        }
        else if (prevMin != nullptr) {
            prevMin->setNext(minNode->getNext());
            delete minNode;
        }
    }

    // Delete the maximum value node
    void DeleteMaximum() {
        if (head == nullptr) return;

        Node<T>* maxNode = head;
        Node<T>* temp = head;

        while (temp != nullptr) {
            if (temp->getData() > maxNode->getData()) {
                maxNode = temp;
            }
            temp = temp->getNext();
        }

        DeleteFromMiddle(maxNode);
    }

    // Print the list
    void PrintList() {
        Node<T>* temp = head;
        while (temp != nullptr) {
```

```
            cout << temp->getData() << "\t";
            temp = temp->getNext();
        }
        cout << endl;
    }

    // Recursive function to print list in reverse order
    void PrintReverse(Node<T>* root) {
        if (root == nullptr) return;
        PrintReverse(root->getNext());
        cout << root->getData() << "\t";
    }

    // Public function to call recursive print
    void PrintInReverse() {
        PrintReverse(head);
        cout << endl;
    }
};

int main() {
    Node<int>* a = new Node<int>(1);
    Node<int>* b = new Node<int>(2);
    Node<int>* c = new Node<int>(3);
    Node<int>* d = new Node<int>(4);
    Node<int>* e = new Node<int>(5);
    Node<int>* f = new Node<int>(6);
    Node<int>* k = new Node<int>(7);
    Node<int>* p = new Node<int>(8);
    Node<int>* z = new Node<int>(9);

    List<int>* list = new List<int>();

    list->InsertBeginning(a);  // Insert first node at the beginning
    list->InsertEnd(b);        // Insert at the end
    list->InsertEnd(c);
    list->InsertMiddle(a, d);  // Insert in the middle
    list->InsertMiddle(b, e);
    list->InsertEnd(f);
    list->InsertEnd(k);
    list->InsertEnd(p);
    list->InsertEnd(z);

    cout << "List after insertions:" << endl;
    list->PrintList();

    list->DeleteFromBeginning();
    cout << "\nAfter deleting from beginning:" << endl;
    list->PrintList();

    list->DeleteFromEnd();
    cout << "\nAfter deleting from end:" << endl;
    list->PrintList();

    list->DeleteFromMiddle();
    cout << "\nAfter deleting from Middle:" << endl;
    list->PrintList();

    list->DeleteMinimum();
    cout << "\nAfter deleting minimum value node:" << endl;
    list->PrintList();

    list->DeleteMaximum();
    cout << "\nAfter deleting maximum value node:" << endl;
    list->PrintList();
```

```
257.
258.     cout << "\nPrinting list in reverse order:" << endl;
259.     list->PrintInReverse();
260.
261.     return 0;
262. }
263.
264.
265.
```

- **Doubly Linked List**

```
 1. #include <iostream>
 2. using namespace std;
 3.
 4. template <typename T>
 5. class Node {
 6. private:
 7.     T data;
 8.     Node* next;
 9.     Node* prev;
10.
11. public:
12.     // Constructor
13.     Node(T element) : data(element), next(nullptr), prev(nullptr) {}
14.
15.     // Setters
16.     void setData(T element) {
17.         data = element;
18.     }
19.
20.     void setNext(Node* node) {
21.         next = node;
22.     }
23.
24.     void setPrev(Node* node) {
25.         prev = node;
26.     }
27.
28.     // Getters
29.     T getData() {
30.         return data;
31.     }
32.
33.     Node* getNext() {
34.         return next;
35.     }
36.
37.     Node* getPrev() {
38.         return prev;
39.     }
40. };
41.
42. template <typename T>
43. class DList {
44. private:
45.     Node<T>* head;
46.
```

```
47. public:
48.     // Constructor
49.     DList() : head(nullptr) {}
50. 
51.     // Insert at the beginning
52.     void InsertBeginning(Node<T>* pNew) {
53.         if (head == nullptr) {
54.             head = pNew;
55.         } else {
56.             pNew->setNext(head);
57.             head->setPrev(pNew);
58.             head = pNew;
59.         }
60.     }
61. 
62.     // Insert at the end
63.     void InsertEnd(Node<T>* pNew) {
64.         if (head == nullptr) {
65.             head = pNew;
66.         } else {
67.             Node<T>* temp = head;
68.             while (temp->getNext() != nullptr) {
69.                 temp = temp->getNext();
70.             }
71.             temp->setNext(pNew);
72.             pNew->setPrev(temp);
73.         }
74.     }
75. 
76.     // Insert in the middle (after pBefore)
77.     void InsertMiddle(Node<T>* pBefore, Node<T>* pNew) {
78.         if (pBefore == nullptr) {
79.             head = pNew;
80.         } else if (pBefore->getNext() == nullptr) {
81.             pBefore->setNext(pNew);
82.             pNew->setPrev(pBefore);
83.         } else {
84.             pNew->setNext(pBefore->getNext());
85.             pNew->setPrev(pBefore);
86.             pBefore->getNext()->setPrev(pNew);
87.             pBefore->setNext(pNew);
88.         }
89.     }
90. 
91.     // Delete from the beginning
92.     void DeleteFromBeginning() {
93.         if (head == nullptr) {
94.             cout << "List is empty" << endl;
95.             return;
96.         }
97.         Node<T>* temp = head;
98.         head = head->getNext();
99.         if (head != nullptr) {
100.             head->setPrev(nullptr);
101.         }
102.         delete temp;
103.     }
104. 
105.     // Delete from the end
106.     void DeleteFromEnd() {
107.         if (head == nullptr) {
108.             cout << "List is empty" << endl;
109.             return;
110.         }
111.         if (head->getNext() == nullptr) {
```

```
            delete head;
            head = nullptr;
        } else {
            Node<T>* temp = head;
            while (temp->getNext() != nullptr) {
                temp = temp->getNext();
            }
            temp->getPrev()->setNext(nullptr);
            delete temp;
        }
    }

    // Delete from the middle using a specific node (overload)
    void DeleteFromMiddle(Node<T>* pToBeDeleted) {
        if (pToBeDeleted == head) {
            DeleteFromBeginning();
        } else if (pToBeDeleted->getNext() == nullptr) {
            DeleteFromEnd();
        } else {
            pToBeDeleted->getPrev()->setNext(pToBeDeleted->getNext());
            pToBeDeleted->getNext()->setPrev(pToBeDeleted->getPrev());
            delete pToBeDeleted;
        }
    }

    // Delete the middle node using the slow and fast pointer technique
    void DeleteFromMiddle() {
        if (head == nullptr || head->getNext() == nullptr) {
            // If the list is empty or has only one element, delete the head
            DeleteFromBeginning();
            return;
        }

        Node<T>* slow = head;
        Node<T>* fast = head;

        // Traverse the list with fast and slow pointers
        while (fast != nullptr && fast->getNext() != nullptr) {
            slow = slow->getNext();
            fast = fast->getNext()->getNext();
        }

        // Now 'slow' is pointing to the middle node
        DeleteFromMiddle(slow);
    }

    // Delete the minimum value node
    void DeleteMinimum() {
        if (head == nullptr) return;

        Node<T>* minNode = head;
        Node<T>* temp = head;
        while (temp != nullptr) {
            if (temp->getData() < minNode->getData()) {
                minNode = temp;
            }
            temp = temp->getNext();
        }
        DeleteFromMiddle(minNode);  // Correctly pass the minimum node
    }

    // Delete the maximum value node
    void DeleteMaximum() {
        if (head == nullptr) return;

```

```
        Node<T>* maxNode = head;
        Node<T>* temp = head;
        while (temp != nullptr) {
            if (temp->getData() > maxNode->getData()) {
                maxNode = temp;
            }
            temp = temp->getNext();
        }
        DeleteFromMiddle(maxNode);  // Correctly pass the maximum node
    }

    // Print the list
    void PrintList() {
        Node<T>* temp = head;
        while (temp != nullptr) {
            cout << temp->getData() << "\t";
            temp = temp->getNext();
        }
        cout << endl;
    }

    // Recursive function to print list in reverse order
    void PrintReverse(Node<T>* root) {
        if (root == nullptr) return;
        PrintReverse(root->getNext());
        cout << root->getData() << "\t";
    }

    // Public function to call recursive print
    void PrintInReverse() {
        PrintReverse(head);
        cout << endl;
    }
};

int main() {
    Node<int>* a = new Node<int>(1);
    Node<int>* b = new Node<int>(2);
    Node<int>* c = new Node<int>(3);
    Node<int>* d = new Node<int>(4);
    Node<int>* e = new Node<int>(5);
    Node<int>* f = new Node<int>(6);
    Node<int>* k = new Node<int>(7);
    Node<int>* p = new Node<int>(8);
    Node<int>* z = new Node<int>(9);

    DList<int>* list = new DList<int>();

    list->InsertBeginning(a);  // Insert first node at the beginning
    list->InsertEnd(b);        // Insert at the end
    list->InsertEnd(c);
    list->InsertMiddle(a, d);  // Insert in the middle
    list->InsertMiddle(b, e);
    list->InsertEnd(f);
    list->InsertEnd(k);
    list->InsertEnd(p);
    list->InsertEnd(z);

    cout << "List after insertions:" << endl;
    list->PrintList();

    list->DeleteFromBeginning();
    cout << "\nAfter deleting from beginning:" << endl;
    list->PrintList();

```

```
    list->DeleteFromEnd();
    cout << "\nAfter deleting from end:" << endl;
    list->PrintList();

    list->DeleteFromMiddle();
    cout << "\nAfter deleting from Middle:" << endl;
    list->PrintList();

    list->DeleteMinimum();
    cout << "\nAfter deleting minimum value node:" << endl;
    list->PrintList();

    list->DeleteMaximum();
    cout << "\nAfter deleting maximum value node:" << endl;
    list->PrintList();

    cout << "\nPrinting list in reverse order:" << endl;
    list->PrintInReverse();

    return 0;
}
```

- **Doubly With Circular**

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

template <typename T>
class Node {
private:
    T data;
    Node* next;
    Node* previous;

public:
    // Constructor
    Node(T pdata) : data(pdata), next(nullptr), previous(nullptr) {}

    // Setters
    void setData(T pVal) {
        data = pVal;
    }

    void setNext(Node* x) {
        next = x;
    }

    void setPrevious(Node* x) {
        previous = x;
    }

    // Getters
    T getData() {
        return data;
    }

```

```
33.     Node* getNext() {
34.         return next;
35.     }
36. 
37.     Node* getPrevious() {
38.         return previous;
39.     }
40. };
41. 
42. template <typename T>
43. class CircularDList {
44. private:
45.     Node<T>* head;
46. 
47. public:
48.     // Constructor
49.     CircularDList() : head(nullptr) {}
50. 
51.     // Inserts node pNew after node pBefore
52.     void Insert(Node<T>* pBefore, Node<T>* pNew) {

53.         if (head == nullptr) {
54.             head = pNew;
55.             head->setNext(head);
56.             head->setPrevious(head);
57.         }
             else if (pBefore == nullptr || pBefore->getNext() == head) {
                  // if pBefore is the last node or nullptr (inserting after the last node)

58.             pNew->setNext(head);
59.             pNew->setPrevious(head->getPrevious());
60.             head->getPrevious()->setNext(pNew);
61.             head->setPrevious(pNew);
62.         }
            else {
63.             pNew->setNext(pBefore->getNext());
64.             pNew->setPrevious(pBefore);
65.             pBefore->getNext()->setPrevious(pNew);
66.             pBefore->setNext(pNew);
67.         }
68.     }
69. 
70.     // Deletes the node pToBeDeleted
71.     void Delete(Node<T>* pToBeDeleted) {
72.         if (head == nullptr) {
73.             cout << "List is empty\n";
74.             return;
75.         }
76.         if (pToBeDeleted == head && head->getNext() == head) {
77.             delete head; // only one node
78.             head = nullptr;
79.         }
             else if (pToBeDeleted == head) {
80.             head->getPrevious()->setNext(head->getNext());
81.             head->getNext()->setPrevious(head->getPrevious());
82.             Node<T>* temp = head;
83.             head = head->getNext();
84.             delete temp;
85.         } else {
86.             pToBeDeleted->getPrevious()->setNext(pToBeDeleted->getNext());
87.             pToBeDeleted->getNext()->setPrevious(pToBeDeleted->getPrevious());
88.             delete pToBeDeleted;
89.         }
90.     }
91. 
```

```
92.     // Prints the list from head
93.     void PrintList() {
94.         if (head == nullptr) {
95.             cout << "List is empty\n";
96.             return;
97.         }
98.         Node<T>* temp = head;
99.         do {
100.            cout << temp->getData() << "\t";
101.            temp = temp->getNext();
102.        } while (temp != head);
103.        cout << endl;
104.    }
105.
106.    // Prints the list in reverse order from the head
107.    void PrintReverse() {
108.        if (head == nullptr) {
109.            cout << "List is empty\n";
110.            return;
111.        }
112.        Node<T>* temp = head->getPrevious(); // Start from the last node
113.        do {
114.            cout << temp->getData() << "\t";
115.            temp = temp->getPrevious();
116.        } while (temp != head->getPrevious());
117.        cout << endl;
118.    }
119.
120.    // Deletes the node with the minimum value
121.    void DeleteMinimum() {
122.        if (head == nullptr) return;
123.        Node<T>* temp = head;
124.        Node<T>* minNode = head;
125.        do {
126.            if (temp->getData() < minNode->getData()) {
127.                minNode = temp;
128.            }
129.            temp = temp->getNext();
130.        } while (temp != head);
131.        Delete(minNode);
132.    }
133.
134.    // Deletes the node with the maximum value
135.    void DeleteMaximum() {
136.        if (head == nullptr) return;
137.        Node<T>* temp = head;
138.        Node<T>* maxNode = head;
139.        do {
140.            if (temp->getData() > maxNode->getData()) {
141.                maxNode = temp;
142.            }
143.            temp = temp->getNext();
144.        } while (temp != head);
145.        Delete(maxNode);
146.    }
147. };
148.
149. int main() {
150.     Node<int>* a = new Node<int>(200);
151.     Node<int>* b = new Node<int>(30);
152.     Node<int>* c = new Node<int>(40);
153.     Node<int>* d = new Node<int>(45);
154.     Node<int>* e = new Node<int>(450);
155.     Node<int>* f = new Node<int>(500);
156.
```

```
157.     CircularDList<int>* list = new CircularDList<int>();
158.
159.     list->Insert(nullptr, a);  // Insert first node
160.     list->Insert(a, b);        // Insert at the end
161.     list->Insert(b, c);
162.     list->Insert(a, d);        // Insert after node a
163.     list->Insert(b, e);        // Insert after node b
164.     list->Insert(c, f);        // Insert after node c
165.
166.     cout << "After inserting all nodes:\n";
167.     list->PrintList();
168.
169.     list->Delete(a);
170.     cout << "After deleting first node:\n";
171.     list->PrintList();
172.
173.     list->DeleteMaximum();
174.     cout << "After deleting maximum value node:\n";
175.     list->PrintList();
176.
177.     list->DeleteMinimum();
178.     cout << "After deleting minimum value node:\n";
179.     list->PrintList();
180.
181.     cout << "Printing in reverse order:\n";
182.     list->PrintReverse();
183.
184.     return 0;
185. }
186.
```

- **Singly With Circular**

```
 1. #include <iostream>
 2. using namespace std;
 3.
 4. template <typename T>
 5. class Node {
 6. private:
 7.     T data;
 8.     Node* next;
 9.
10. public:
11.     // Constructor
12.     Node(T pdata) : data(pdata), next(nullptr) {}
13.
14.     // Setters
15.     void setData(T pVal) {
16.         data = pVal;
17.     }
18.
19.     void setNext(Node* x) {
20.         next = x;
21.     }
22.
23.     // Getters
24.     T getData() {
25.         return data;
26.     }
27.
28.     Node* getNext() {
29.         return next;
```

```
    }
};

template <typename T>
class CircularSinglyList {
private:
    Node<T>* head;

public:
    // Constructor
    CircularSinglyList() : head(nullptr) {}

    // Inserts node pNew after node pBefore
    void Insert(Node<T>* pBefore, Node<T>* pNew) {
        if (head == nullptr) {
            head = pNew;
            head->setNext(head);  // Point to itself
        }
      else if (pBefore == nullptr || pBefore->getNext() == head) {
            // If inserting after the last node or pBefore is nullptr
            Node<T>* last = head;
            while (last->getNext() != head) {
                last = last->getNext();
            }
            last->setNext(pNew);
            pNew->setNext(head);
        }
      else {
            pNew->setNext(pBefore->getNext());
            pBefore->setNext(pNew);
        }
    }

    // Deletes the node pToBeDeleted
    void Delete(Node<T>* pToBeDeleted) {
        if (head == nullptr) {
            cout << "List is empty\n";
            return;
        }

        if (head == pToBeDeleted) {
            if (head->getNext() == head) {  // Only one node in the list
                delete head;
                head = nullptr;
            }
        else {
                Node<T>* last = head;
                while (last->getNext() != head) {
                    last = last->getNext();
                }
                last->setNext(head->getNext());
                Node<T>* temp = head;
                head = head->getNext();
                delete temp;
            }
        } else {
            Node<T>* temp = head;
            while (temp->getNext() != pToBeDeleted && temp->getNext() != head) {
                temp = temp->getNext();
            }
            if (temp->getNext() == pToBeDeleted) {
                temp->setNext(pToBeDeleted->getNext());
                delete pToBeDeleted;
            }
        }
```

```
    }

    // Prints the list from head
    void PrintList() {
        if (head == nullptr) {
            cout << "List is empty\n";
            return;
        }
        Node<T>* temp = head;
        do {
            cout << temp->getData() << "\t";
            temp = temp->getNext();
        } while (temp != head);
        cout << endl;
    }

    // Deletes the node with the minimum value
    void DeleteMinimum() {
        if (head == nullptr) return;

        Node<T>* temp = head;
        Node<T>* minNode = head;
        do {
            if (temp->getData() < minNode->getData()) {
                minNode = temp;
            }
            temp = temp->getNext();
        } while (temp != head);

        Delete(minNode);
    }

    // Deletes the node with the maximum value
    void DeleteMaximum() {
        if (head == nullptr) return;

        Node<T>* temp = head;
        Node<T>* maxNode = head;
        do {
            if (temp->getData() > maxNode->getData()) {
                maxNode = temp;
            }
            temp = temp->getNext();
        } while (temp != head);

        Delete(maxNode);
    }
};

int main() {
    Node<int>* a = new Node<int>(200);
    Node<int>* b = new Node<int>(30);
    Node<int>* c = new Node<int>(40);
    Node<int>* d = new Node<int>(45);
    Node<int>* e = new Node<int>(450);
    Node<int>* f = new Node<int>(500);

    CircularSinglyList<int>* list = new CircularSinglyList<int>();

    list->Insert(nullptr, a);  // Insert first node
    list->Insert(a, b);        // Insert at the end
    list->Insert(b, c);
    list->Insert(a, d);        // Insert after node a
    list->Insert(b, e);        // Insert after node b
    list->Insert(c, f);        // Insert after node c
```

```
157.
158.     cout << "After inserting all nodes:\n";
159.     list->PrintList();
160.
161.     list->Delete(a);
162.     cout << "After deleting first node:\n";
163.     list->PrintList();
164.
165.     list->DeleteMaximum();
166.     cout << "After deleting maximum value node:\n";
167.     list->PrintList();
168.
169.     list->DeleteMinimum();
170.     cout << "After deleting minimum value node:\n";
171.     list->PrintList();
172.
173.     return 0;
174. }
175.
```