- **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);
        }
    }

    // Insert in the middle (after pBefore)
    void InsertMiddle(Node<T>* pBefore, Node<T>* pNew) {
        if (pBefore == nullptr) {
            head = pNew;
        }
        else {
            pNew->setNext(pBefore->getNext());
            pBefore->setNext(pNew);
        }
    }

    // Delete from the beginning
    void DeleteFromBeginning() {
        if (head == nullptr) {
            cout << "List is empty" << endl;
            return;
        }
        Node<T>* temp = head;
        head = head->getNext();
        delete temp;
    }

    // Delete from the end
    void DeleteFromEnd() {
        if (head == nullptr) {
            cout << "List is empty" << endl;
            return;
        }
        if (head->getNext() == nullptr) {
            delete head;
            head = nullptr;
        }
        else {
            Node<T>* temp = head;
            while (temp->getNext()->getNext() != nullptr) {
                temp = temp->getNext();
            }
            delete temp->getNext();
            temp->setNext(nullptr);
        }
    }

    // Delete from the middle using a specific node
    void DeleteFromMiddle(Node<T>* pToBeDeleted) {
        if (pToBeDeleted == head) {
            DeleteFromBeginning();
        }
        else {
            Node<T>* temp = head;
            while (temp->getNext() != pToBeDeleted) {
                temp = temp->getNext();
            }
            temp->setNext(pToBeDeleted->getNext());
            delete pToBeDeleted;
        }
    }

    // Delete the middle node using the slow and fast pointer technique
    void DeleteFromMiddle() {
        if (head == nullptr || head->getNext() == nullptr) {
            DeleteFromBeginning();
            return;
        }

        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();

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

    return 0;
}



```

- ## Doubly Linked List

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

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

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

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

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

    void setPrev(Node* node) {
        prev = node;
    }

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

    Node* getNext() {
        return next;
    }

    Node* getPrev() {
        return prev;
    }
};

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

```

```cpp
public:
    // Constructor
    DList() : head(nullptr) {}

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

    // Insert in the middle (after pBefore)
    void InsertMiddle(Node<T>* pBefore, Node<T>* pNew) {
        if (pBefore == nullptr) {
            head = pNew;
        } else if (pBefore->getNext() == nullptr) {
            pBefore->setNext(pNew);
            pNew->setPrev(pBefore);
        } else {
            pNew->setNext(pBefore->getNext());
            pNew->setPrev(pBefore);
            pBefore->getNext()->setPrev(pNew);
            pBefore->setNext(pNew);
        }
    }

    // Delete from the beginning
    void DeleteFromBeginning() {
        if (head == nullptr) {
            cout << "List is empty" << endl;
            return;
        }
        Node<T>* temp = head;
        head = head->getNext();
        if (head != nullptr) {
            head->setPrev(nullptr);
        }
        delete temp;
    }

    // Delete from the end
    void DeleteFromEnd() {
        if (head == nullptr) {
            cout << "List is empty" << endl;
            return;
        }
        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;
    }

    Node* getNext() {
        return next;
    }

    Node* getPrevious() {
        return previous;
    }
};

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

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

    // Inserts node pNew after node pBefore
    void Insert(Node<T>* pBefore, Node<T>* pNew) {

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

            pNew->setNext(head);
            pNew->setPrevious(head->getPrevious());
            head->getPrevious()->setNext(pNew);
            head->setPrevious(pNew);
        }
        else {
            pNew->setNext(pBefore->getNext());
            pNew->setPrevious(pBefore);
            pBefore->getNext()->setPrevious(pNew);
            pBefore->setNext(pNew);
        }
    }

    // Deletes the node pToBeDeleted
    void Delete(Node<T>* pToBeDeleted) {
        if (head == nullptr) {
            cout << "List is empty\n";
            return;
        }
        if (pToBeDeleted == head && head->getNext() == head) {
            delete head; // only one node
            head = nullptr;
        }
         else if (pToBeDeleted == head) {
            head->getPrevious()->setNext(head->getNext());
            head->getNext()->setPrevious(head->getPrevious());
            Node<T>* temp = head;
            head = head->getNext();
            delete temp;
        } else {
            pToBeDeleted->getPrevious()->setNext(pToBeDeleted->getNext());
            pToBeDeleted->getNext()->setPrevious(pToBeDeleted->getPrevious());
            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;
    }

    // Prints the list in reverse order from the head
    void PrintReverse() {
        if (head == nullptr) {
            cout << "List is empty\n";
            return;
        }
        Node<T>* temp = head->getPrevious(); // Start from the last node
        do {
            cout << temp->getData() << "\t";
            temp = temp->getPrevious();
        } while (temp != head->getPrevious());
        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);

    CircularDList<int>* list = new CircularDList<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

    cout << "After inserting all nodes:\n";
    list->PrintList();

    list->Delete(a);
    cout << "After deleting first node:\n";
    list->PrintList();

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

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

    cout << "Printing in reverse order:\n";
    list->PrintReverse();

    return 0;
}

```

- **Singly With Circular**

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

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

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

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

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

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

    Node* getNext() {
        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
```

```cpp

    cout << "After inserting all nodes:\n";
    list->PrintList();

    list->Delete(a);
    cout << "After deleting first node:\n";
    list->PrintList();

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

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

    return 0;
}

```