```
/*Implement circular linked list and perform operations
on it. */
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

class CircularLinkedList {
    private:
        Node* head;
    public:
        CircularLinkedList() {
            head = NULL;
        }

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

        void insertAtBeginning(int data) {
            Node* newNode = new Node();
            newNode->data = data;
            newNode->next = head;
            Node* temp = head;
            if (head != NULL) {
                while(temp->next != head) {
                    temp = temp->next;
                }
                temp->next = newNode;
            } else {
                newNode->next = newNode;
            }
            head = newNode;
        }

        void deleteAtEnd() {
            if (head == NULL) {
                return;
            }
            Node* temp = head;
            if (temp->next == head) {
                head = NULL;
                delete temp;
                return;
            }
            while(temp->next->next != head) {
                temp = temp->next;
            }
            Node* toDelete = temp->next;
            temp->next = head;
            delete toDelete;
        }

        void deleteAtBeginning() {
            if (head == NULL) {
                return;
            }
            Node* temp = head;
            if (temp->next == head) {
                head = NULL;
                delete temp;
                return;
            }
            while(temp->next != head) {
                temp = temp->next;
            }
            Node* toDelete = head;
            head = head->next;
            temp->next = head;
            delete toDelete;
        }

        void display() {
            Node* temp = head;
            if (head != NULL) {
                do {
                    cout << temp->data << " ";
                    temp = temp->next;
                } while(temp != head);
            }
        }
};

int main() {
    CircularLinkedList cll;
    cll.insertAtEnd(1);
    cll.insertAtEnd(2);
    cll.insertAtEnd(3);
    cll.insertAtBeginning(0);
    cll.deleteAtEnd();
    cll.deleteAtBeginning();
    cll.display();
    return 0;
}
```

```
output:
3 2
```

```
/*Implement Binary Search.*/
#include <iostream>
using namespace std;

int binarySearch(int arr[], int n, int key) {
    int left = 0, right = n - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] == key) {
            return mid;
        }
        else if (arr[mid] < key) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }
    return -1;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 3;
    int index = binarySearch(arr, n, key);
    if (index != -1) {
        cout << "Element found at index " << index << endl;
    }
    else {
        cout << "Element not found" << endl;
    }
    return 0;
}

/*Create binary tree and perform recursive traversals*/
#include <iostream>
using namespace std;

// Structure for a node of a binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Function to create a new node and return its address
Node* getNewNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Recursive function to do pre-order traversal of the binary tree
void preOrder(Node* root) {
    if (root == NULL) return;
    cout << root->data << " ";
    preOrder(root->left);
    preOrder(root->right);
}

// Recursive function to do in-order traversal of the binary tree
void inOrder(Node* root) {
    if (root == NULL) return;
    inOrder(root->left);
    cout << root->data << " ";
    inOrder(root->right);
```

```
}

// Recursive function to do post-order traversal of the binary tree
void postOrder(Node* root) {
    if (root == NULL) return;
    postOrder(root->left);
    postOrder(root->right);
    cout << root->data << " ";
}

int main() {
    Node* root = getNewNode(1);
    root->left = getNewNode(2);
    root->right = getNewNode(3);
    root->left->left = getNewNode(4);
    root->left->right = getNewNode(5);

    cout << "Pre-order traversal: ";
    preOrder(root);
    cout << endl;

    cout << "In-order traversal: ";
    inOrder(root);
    cout << endl;

    cout << "Post-order traversal: ";
    postOrder(root);
    cout << endl;

    return 0;
}
```

```
output:
Pre-order traversal: 1 2 4 5 3
In-order traversal: 4 2 5 1 3
Post-order traversal: 4 5 2 3 1
```

```
/*Implement circular linked list and perform operations
on it. */
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

class CircularLinkedList {
    private:
        Node* head;
    public:
        CircularLinkedList() {
            head = NULL;
        }

        void insertAtEnd(int data) {
            Node* newNode = new Node();
            newNode->data = data;
            newNode->next = head;
            Node* temp = head;
            if (head != NULL) {
```

```
        while(temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
    } else {
        newNode->next = newNode;
    }
    head = newNode;
}

void insertAtBeginning(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = head;
    Node* temp = head;
    if (head != NULL) {
        while(temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
    } else {
        newNode->next = newNode;
    }
    head = newNode;
}

void deleteAtEnd() {
    if (head == NULL) {
        return;
    }
    Node* temp = head;
    if (temp->next == head) {
        head = NULL;
        delete temp;
        return;
    }
    while(temp->next->next != head) {
        temp = temp->next;
    }
    Node* toDelete = temp->next;
    temp->next = head;
    delete toDelete;
}

void deleteAtBeginning() {
    if (head == NULL) {
        return;
    }
    Node* temp = head;
    if (temp->next == head) {
        head = NULL;
        delete temp;
        return;
    }
    while(temp->next != head) {
        temp = temp->next;
    }
    Node* toDelete = head;
    head = head->next;
    temp->next = head;
    delete toDelete;
```

```
        }

        void display() {
            Node* temp = head;
            if (head != NULL) {
                do {
                    cout << temp->data << " ";
                    temp = temp->next;
                } while(temp != head);
            }
        }
};

int main() {
    CircularLinkedList cll;
    cll.insertAtEnd(1);
    cll.insertAtEnd(2);
    cll.insertAtEnd(3);
    cll.insertAtBeginning(0);
    cll.deleteAtEnd();
    cll.deleteAtBeginning();
    cll.display();
    return 0;
}
```

```
output:
3 2
```

```
/*Implement circular queue using arrays. */
#include <iostream>
using namespace std;

class CircularQueue {
    int *queue, size, front, rear;

public:
    CircularQueue(int s) {
        size = s;
        queue = new int[size];
        front = rear = -1;
    }
    void enqueue(int x);
    int dequeue();
    void display();
};

void CircularQueue::enqueue(int x) {
    if ((front == 0 && rear == size - 1) || (front == rear + 1)) {
        cout << "Queue is full\n";
        return;
    }
    else if (front == -1) {
        front = rear = 0;
    }
    else if (rear == size - 1 && front != 0) {
        rear = 0;
    }
    else {
        rear++;
    }
    queue[rear] = x;
}
```

```
int CircularQueue::dequeue() {
    if (front == -1) {
        cout << "Queue is empty\n";
        return -1;
    }

    int x = queue[front];
    if (front == rear) {
        front = rear = -1;
    }
    else if (front == size - 1) {
        front = 0;
    }
    else {
        front++;
    }
    return x;
}

void CircularQueue::display() {
    if (front == -1) {
        cout << "Queue is empty\n";
        return;
    }
    if (rear >= front) {
        for (int i = front; i <= rear; i++)
            cout << queue[i] << " ";
    }
    else {
        for (int i = front; i < size; i++)
            cout << queue[i] << " ";
        for (int i = 0; i <= rear; i++)
            cout << queue[i] << " ";
    }
}

int main() {
    CircularQueue q(5);
    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);
    q.enqueue(4);
    q.enqueue(5);
    q.enqueue(6);
    q.display();
    cout << endl;
    q.dequeue();
    q.dequeue();
    q.display();
    cout << endl;
    return 0;
}
```

```
output:
Queue is full
1 2 3 4 5
3 4 5
```

```
/*Create binary tree. Find height of the tree and print
leaf nodes. Find mirror image, print original and
```

```
mirror image using level-wise printing. */
#include <iostream>
using namespace std;

struct Node
{
    int data;
    Node *left, *right;
};

Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

int height(Node* root)
{
    if (root == NULL)
        return 0;
    return 1 + max(height(root->left), height(root->right));
}

void printLeafNodes(Node* root)
{
    if (root == NULL)
        return;
    if (root->left == NULL && root->right == NULL)
        cout << root->data << " ";
    printLeafNodes(root->left);
    printLeafNodes(root->right);
}

Node* mirror(Node* root)
{
    if (root == NULL)
        return NULL;
    Node* left = mirror(root->left);
    Node* right = mirror(root->right);
    root->left = right;
    root->right = left;
    return root;
}

void printLevelWise(Node* root)
{
    if (root == NULL)
        return;
    cout << root->data << " ";
    printLevelWise(root->left);
    printLevelWise(root->right);
}

int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Height of tree: " << height(root) << endl;
    cout << "Leaf nodes: ";
    printLeafNodes(root);
    cout << endl;

    Node* mirrorRoot = mirror(root);
    cout << "Original tree: ";
    printLevelWise(root);
    cout << endl;
    cout << "Mirror image: ";
    printLevelWise(mirrorRoot);

    return 0;
}
```

```
output:
Height of tree: 3
Leaf nodes: 4 5 3
Original tree: 1 3 2 5 4
Mirror image: 1 3 2 5 4
```

```
/*Create two doubly linked lists. Sort them after
creation using pointer manipulation. Merge these two
lists into one list so that the merged list is in sorted
order. (No new Node should be created.*/
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
    Node* prev;
};

Node* head1 = NULL;
Node* head2 = NULL;

void insert(int data, Node*& head) {
    Node* temp = new Node();
    temp->data = data;
    temp->next = NULL;
    temp->prev = NULL;
    if (!head) {
        head = temp;
    } else {
        Node* ptr = head;
        while (ptr->next) {
            ptr = ptr->next;
        }
        ptr->next = temp;
        temp->prev = ptr;
    }
}

void sortList(Node*& head) {
    Node* ptr = head;
    while (ptr->next) {
        Node* temp = ptr->next;
        while (temp) {
            if (ptr->data > temp->data) {
                swap(ptr->data, temp->data);
            }
            temp = temp->next;
        }
        ptr = ptr->next;
    }
}

void mergeLists() {
    Node* ptr1 = head1;
    Node* ptr2 = head2;
    while (ptr1->next) {
        ptr1 = ptr1->next;
    }
    ptr1->next = ptr2;
    ptr2->prev = ptr1;
}

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

int main() {
    insert(5, head1);
    insert(2, head1);
    insert(4, head1);
    insert(7, head1);
    sortList(head1);
    printList(head1);

    insert(3, head2);
    insert(1, head2);
    insert(6, head2);
    insert(9, head2);
    sortList(head2);
    printList(head2);

    mergeLists();
    printList(head1);

    return 0;
}

output:
2 4 5 7
1 3 6 9
2 4 5 7 1 3 6 9


/*Write a program to implement Heap sort
method.. */
#include <iostream>
using namespace std;
```

```
// Function to heapify the tree
void heapify(int arr[], int n, int i)
{
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    if (l < n && arr[l] > arr[largest])
        largest = l;

    if (r < n && arr[r] > arr[largest])
        largest = r;

    if (largest != i)
    {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

// Main function to sort the array
void heapSort(int arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i=n-1; i>=0; i--)
    {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSort(arr, n);

    for (int i=0; i<n; ++i)
        cout << arr[i] << " ";

    return 0;
}
```

```
output:
5 6 7 11 12 13
```

```
//Implement application of array in polynomial expression.
#include <iostream>
using namespace std;

// Function to evaluate the polynomial
double evaluatePolynomial(double x, int degree, double coeffs[]) {
    double result = 0;
    for (int i = degree; i >= 0; i--) {
        result = result * x + coeffs[i];
    }
    return result;
}
```

```
int main() {
    int degree;
    double x, coeffs[50], result;

    cout << "Enter the degree of the polynomial: ";
    cin >> degree;
    cout << "Enter the value of x: ";
    cin >> x;

    cout << "Enter the coefficients of the polynomial in descending order:
";
    for (int i = 0; i <= degree; i++) {
        cin >> coeffs[i];
    }

    result = evaluatePolynomial(x, degree, coeffs);
    cout << "Result: " << result << endl;

    return 0;
}

output:
Enter the degree of the polynomial: 3
Enter the value of x: 2
Enter the coefficients of the polynomial in descending order: 1
2
3
4
Result: 49


/*. Implement Linked queue*/

#include <bits/stdc++.h>
using namespace std;

struct QNode {
    int data;
    QNode* next;
    QNode(int d)
    {
        data = d;
        next = NULL;
    }
};

struct Queue {
    QNode *front, *rear;
    Queue() { front = rear = NULL; }

    void enQueue(int x)
    {

        // Create a new LL node
        QNode* temp = new QNode(x);

        // If queue is empty, then
        // new node is front and rear both
        if (rear == NULL) {
            front = rear = temp;
            return;
        }

        // Add the new node at
        // the end of queue and change rear
        rear->next = temp;
        rear = temp;
    }

    // Function to remove
    // a key from given queue q
    void deQueue()
    {
        // If queue is empty, return NULL.
        if (front == NULL)
            return;

        // Store previous front and
        // move front one node ahead
        QNode* temp = front;
        front = front->next;

        // If front becomes NULL, then
        // change rear also as NULL
        if (front == NULL)
            rear = NULL;

        delete (temp);
    }
};

// Driver code
int main()
{

    Queue q;
    q.enQueue(10);
    q.enQueue(20);
    q.deQueue();
    q.deQueue();
    q.enQueue(30);
    q.enQueue(40);
    q.enQueue(50);
    q.deQueue();
    cout << "Queue Front : " << (q.front)->data << endl;
    cout << "Queue Rear : " << (q.rear)->data;
}

output:
Queue Front : 40
Queue Rear : 50

/*Write a program to implement Merge sort method*/
#include <iostream>
using namespace std;

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

int main() {
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, n - 1);

    cout << "Sorted array: \n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

```
output:
Sorted array:
5 6 7 11 12 13
```

```
/*Implement Sequential Search.*/
#include <iostream>
using namespace std;

int sequentialSearch(int array[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (array[i] == key) {
            return i; // return the index of the key if found
        }
    }
    return -1; // return -1 if key is not found
}

int main() {
    int array[] = {1, 2, 3, 4, 5};
    int size = sizeof(array) / sizeof(array[0]);
    int key = 3;
    int index = sequentialSearch(array, size, key);
    if (index != -1) {
        cout << "Key found at index " << index << endl;
    } else {
        cout << "Key not found" << endl;
    }
    return 0;
}

/*. Write a menu driven program to perform following
operations on singly linked list: Create, Insert,
Delete, and Display.*/
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

Node* head = NULL;

void insert(int x) {
    Node* temp = new Node();
    temp->data = x;
    temp->next = head;
    head = temp;
}

void Delete(int n) {
    Node* temp1 = head;
    if(n == 1) {
        head = temp1->next;
        delete temp1;
        return;
    }
    for(int i=0; i<n-2; i++) {
        temp1 = temp1->next;
    }
    Node* temp2 = temp1->next;
    temp1->next = temp2->next;
    delete temp2;
}
```

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

int main() {
    int choice, x, n;
    while(1) {
        cout << "1. Insert" << endl;
        cout << "2. Delete" << endl;
        cout << "3. Display" << endl;
        cout << "4. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;
        switch(choice) {
            case 1: cout << "Enter the element: ";
                    cin >> x;
                    insert(x);
                    break;
            case 2: cout << "Enter the element you want to delete: ";
                    cin >> n;
                    Delete(n);
                    break;
            case 3: display();
                    break;
            case 4: exit(0);
            default: cout << "Invalid Input" << endl;
        }
    }
    return 0;
}
```

```
output:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter the element: 2
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
2
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:
```

```
//Implement application of array in sparse matrix to
//perform simple and fast transpose.
```

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

struct SparseMatrix {
    int row;
    int col;
    int value;
};

SparseMatrix* transpose(SparseMatrix* mat, int size) {
    SparseMatrix* transposed = new SparseMatrix[size];
    for (int i = 0; i < size; i++) {
        transposed[i].col = mat[i].row;
        transposed[i].row = mat[i].col;
        transposed[i].value = mat[i].value;
    }
    return transposed;
}

int main() {
    SparseMatrix mat[5] = {{0, 0, 15}, {0, 3, 22}, {1, 1, 11}, {2, 2, 27},
{3, 1, 17}};
    int size = sizeof(mat)/sizeof(mat[0]);
    SparseMatrix* transposed = transpose(mat, size);

    cout << "Original Matrix:" << endl;
    for (int i = 0; i < size; i++) {
        cout << "(" << mat[i].row << ", " << mat[i].col << ", " <<
mat[i].value << ")" << endl;
    }

    cout << "Transposed Matrix:" << endl;
    for (int i = 0; i < size; i++) {
        cout << "(" << transposed[i].row << ", " << transposed[i].col << ",
" << transposed[i].value << ")" << endl;
    }

    return 0;
}


//output:-
//Original Matrix:
//(0, 0, 15)
//(0, 3, 22)
//(1, 1, 11)
//(2, 2, 27)
//(3, 1, 17)
//Transposed Matrix:
//(0, 0, 15)
//(3, 0, 22)
//(1, 1, 11)
//(2, 2, 27)
//(1, 3, 17)
```