19) Write a program to implement a Doubly Linked List (DLL). Your program should support the following operations: Insert a node at the beginning, at the end, and at any given position, delete a node from the beginning, from the end, and from any given position, display the list in forward and reverse order, search for an element in the list. Also analyse how DLL helps in easier forward–backward navigation compared to a singly linked list.

```
#include<iostream>

using namespace std;

class Node {

public:

   int data;

   Node* prev;

   Node* next;


   Node(int d) {

      data = d;

      prev = NULL;

      next = NULL;

   }

};

// Display DLL forward

void displayForward(Node* head) {

   cout << "Forward DLL: ";

   Node* temp = head;


   while(temp != NULL) {

      cout << temp->data << " ";

      temp = temp->next;

   }

   cout << endl;

}


// Display DLL reverse
```

```
void displayReverse(Node* tail) {
   cout << "Reverse DLL: ";
   Node* temp = tail;

   while(temp != NULL) {
      cout << temp->data << " ";
      temp = temp->prev;
   }
   cout << endl;
}

// Insert at beginning
void insertAtHead(Node* &head, Node* &tail, int d) {

   Node* temp = new Node(d);

   if(head == NULL) {
      head = tail = temp;
   }
   else {
      temp->next = head;
      head->prev = temp;
      head = temp;
   }

   cout << d << " inserted at beginning.\n";
}

// Insert at end
void insertAtTail(Node* &head, Node* &tail, int d) {
```

```
    Node* temp = new Node(d);

    if(tail == NULL) {
        head = tail = temp;
    }
    else {
        tail->next = temp;
        temp->prev = tail;
        tail = temp;
    }

    cout << d << " inserted at end.\n";
}

// Insert at given position
void insertAtPosition(Node* &head, Node* &tail, int pos, int d) {

    if(pos == 1) {
        insertAtHead(head, tail, d);
        return;
    }

    Node* temp = head;
    int cnt = 1;

    while(cnt < pos - 1 && temp != NULL) {
        temp = temp->next;
        cnt++;
    }

    if(temp == NULL) {
```

```
        cout << "Invalid position!\n";
        return;
    }

    if(temp->next == NULL) {
        insertAtTail(head, tail, d);
        return;
    }

    Node* newNode = new Node(d);

    newNode->next = temp->next;
    newNode->prev = temp;

    temp->next->prev = newNode;
    temp->next = newNode;

    cout << d << " inserted at position " << pos << ".\n";
}

// Delete node at given position (your style)
void deleteNode(int position, Node* &head, Node* &tail) {

    // Case 1: delete first node
    if(position == 1) {

        Node* temp = head;

        if(head->next == NULL) {   // only 1 node
            head = tail = NULL;
            delete temp;
```

```
        return;
    }

    head = head->next;
    head->prev = NULL;

    temp->next = NULL;
    delete temp;

    cout << "Deleted node from beginning.\n";
    return;
}

// Case 2: delete middle or last node
Node* curr = head;
Node* prevNode = NULL;

int cnt = 1;
while(cnt < position && curr != NULL) {
    prevNode = curr;
    curr = curr->next;
    cnt++;
}

if(curr == NULL) {
    cout << "Invalid position!\n";
    return;
}

// Case: Last node
if(curr->next == NULL) {
```

```
        tail = prevNode;
        tail->next = NULL;

        curr->prev = NULL;
        delete curr;

        cout << "Deleted node from end.\n";
        return;
    }

    // Case: Middle node
    prevNode->next = curr->next;
    curr->next->prev = prevNode;

    curr->next = curr->prev = NULL;
    delete curr;

    cout << "Deleted node at position " << position << ".\n";
}

// Search element
void search(Node* head, int key) {
    Node* temp = head;
    int pos = 1;

    while(temp != NULL) {
        if(temp->data == key) {
            cout << key << " found at position " << pos << endl;
            return;
        }
        temp = temp->next;
```

```
        pos++;
    }

    cout << key << " not found.\n";
}

int main() {

    Node* head = NULL;
    Node* tail = NULL;

    insertAtHead(head, tail, 10);
    insertAtHead(head, tail, 20);

    insertAtTail(head, tail, 30);
    insertAtTail(head, tail, 40);

    insertAtPosition(head, tail, 3, 99);  // insert in middle

    displayForward(head);
    displayReverse(tail);

    deleteNode(1, head, tail);     // delete head
    deleteNode(4, head, tail);     // delete end
    deleteNode(2, head, tail);     // delete middle

    displayForward(head);
    displayReverse(tail);

    search(head, 30);
    search(head, 100);
```

```
    return 0;
}
```

18)

Write a program to implement a linked-list-based Music Playlist Manager.
Your program should allow users to: Add a new song, delete a song, move to the next or previous song, Display the current playlist. Explain why a linked list is better than an array for this application (dynamic size, fast insert/delete).

```
#include<iostream>
using namespace std;
class Node {
public:
    string song;
    Node* prev;
    Node* next;
    Node(string name) {
        song = name;
        prev = NULL;
        next = NULL;
    }
};
// Display Playlist
void displayPlaylist(Node* head) {
    cout << "Playlist: ";
    if (head == NULL) {
        cout << "Empty\n";
        return;
    }
```

```
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->song << "  ";
        temp = temp->next;
    }
    cout << endl;
}
// Add song at END of playlist
void addSong(Node*& head, Node*& tail, string name) {
    Node* temp = new Node(name);
    if (head == NULL) {
        head = tail = temp;
    }
    else {
        tail->next = temp;
        temp->prev = tail;
        tail = temp;
    }
    cout << "Added: " << name << endl;
}
// Delete a song by name
void deleteSong(Node*& head, Node*& tail, string name) {
    if (head == NULL) {
        cout << "Playlist is empty!\n";
        return;
    }
    Node* curr = head;
    while (curr != NULL && curr->song != name)
        curr = curr->next;
    if (curr == NULL) {
        cout << "Song not found.\n";
```

```
        return;
    }
    // deleting first song
    if (curr == head) {
        head = head->next;
        if (head != NULL) head->prev = NULL;
        if (curr == tail) tail = NULL;
    }
    // deleting last song
    else if (curr == tail) {
        tail = tail->prev;
        tail->next = NULL;
    }
    // deleting middle song
    else {
        curr->prev->next = curr->next;
        curr->next->prev = curr->prev;
    }

    curr->next = curr->prev = NULL;
    delete curr;

    cout << "Deleted: " << name << endl;
}

// Move to next song
void nextSong(Node*& current) {
    if (current == NULL) {
        cout << "No current song.\n";
        return;
    }
```

```
    if (current->next == NULL) {
        cout << "Already at the LAST song.\n";
        return;
    }

    current = current->next;
    cout << "Now playing: " << current->song << endl;
}

// Move to previous song
void prevSong(Node*& current) {
    if (current == NULL) {
        cout << "No current song.\n";
        return;
    }

    if (current->prev == NULL) {
        cout << "Already at the FIRST song.\n";
        return;
    }

    current = current->prev;
    cout << "Now playing: " << current->song << endl;
}

// Show current song
void showCurrent(Node* current) {
    if (current == NULL) {
        cout << "No song selected.\n";
        return;
```

```
    }
    cout << "Currently playing: " << current->song << endl;
}

int main() {

    Node* head = NULL;
    Node* tail = NULL;

    // Adding songs
    addSong(head, tail, "Song1");
    addSong(head, tail, "Song2");
    addSong(head, tail, "Song3");
    addSong(head, tail, "Song4");

    displayPlaylist(head);

    Node* current = head;  // Start with first song
    showCurrent(current);

    nextSong(current);
    nextSong(current);
    prevSong(current);

    deleteSong(head, tail, "Song3");
    displayPlaylist(head);

    showCurrent(current);

    return 0;
}
```

17)Write a program to implement a singly linked list and perform the following operations: Insert a node at the beginning, end, and at a specific position, delete a node from the beginning, end, and a specific position, Search for an element in the linked list, Display the linked list

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

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

   Node(int data) {
      this->data = data;
      this->next = NULL;
   }
};

// Print the list
void print(Node* head) {
   if (head == NULL) {
      cout << "List is empty\n";
      return;
   }

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

```
    cout << endl;
}

// Insert at Head
void insertAtHead(Node*& head, int d) {
    Node* temp = new Node(d);
    temp->next = head;
    head = temp;
}

// Insert at Tail
void insertAtTail(Node*& tail, int d) {
    Node* temp = new Node(d);
    tail->next = temp;
    tail = temp;
}

// Insert at any position
void insertAtPosition(Node*& head, Node*& tail, int position, int d) {

    if (position == 1) {
        insertAtHead(head, d);
        return;
    }

    Node* temp = head;
    int cnt = 1;

    while (cnt < position - 1 && temp != NULL) {
        temp = temp->next;
        cnt++;
```

```
    }

    if (temp == NULL) {
        cout << "Invalid position!\n";
        return;
    }

    if (temp->next == NULL) {
        insertAtTail(tail, d);
        return;
    }

    Node* nodeToInsert = new Node(d);
    nodeToInsert->next = temp->next;
    temp->next = nodeToInsert;
}

// Delete Node from given position
void deleteNode(int position, Node*& head, Node*& tail) {

    if (head == NULL) {
        cout << "List empty!\n";
        return;
    }

    // delete first node
    if (position == 1) {
        Node* temp = head;
        head = head->next;

        // if only one node existed
```

```
        if (head == NULL)
            tail = NULL;

        temp->next = NULL;
        delete temp;
        return;
    }

    Node* curr = head;
    Node* prev = NULL;

    int cnt = 1;
    while (cnt < position && curr != NULL) {
        prev = curr;
        curr = curr->next;
        cnt++;
    }

    if (curr == NULL) {
        cout << "Invalid position!\n";
        return;
    }

    // deleting last node
    if (curr->next == NULL)
        tail = prev;

    prev->next = curr->next;
    curr->next = NULL;
    delete curr;
}
```

```
// Search an element
void search(Node* head, int key) {
    Node* temp = head;
    int pos = 1;

    while (temp != NULL) {
        if (temp->data == key) {
            cout << key << " found at position " << pos << endl;
            return;
        }
        pos++;
        temp = temp->next;
    }

    cout << key << " not found.\n";
}

int main() {

    Node* head = NULL;
    Node* tail = NULL;

    // creating first node manually
    Node* node1 = new Node(10);
    head = node1;
    tail = node1;

    insertAtTail(tail, 20);
    insertAtTail(tail, 30);
    insertAtHead(head, 5);
```

```
    print(head);

    insertAtPosition(head, tail, 3, 99);
    print(head);

    deleteNode(1, head, tail);
    print(head);

    deleteNode(4, head, tail);
    print(head);

    search(head, 20);
    search(head, 200);

    cout << "Head = " << (head ? head->data : -1) << endl;
    cout << "Tail = " << (tail ? tail->data : -1) << endl;

    return 0;
}
```

16. Write a program to implement generalized linked list

```
#include<iostream>
using namespace std;
class Node {
public:
    int flag;       // 0 = data, 1 = sublist
    int data;       // valid only if flag = 0
    Node* next;     // next element
    Node* down;     // points to sublist (only if flag = 1)
```

```
    Node(int f, int d = 0) {
        flag = f;
        data = d;
        next = NULL;
        down = NULL;
    }
};
// Create a data node
Node* createDataNode(int x) {
    Node* temp = new Node(0, x);
    return temp;
}

// Create a sublist node
Node* createSublistNode(Node* subHead) {
    Node* temp = new Node(1);
    temp->down = subHead;
    return temp;
}

// Insert data or sublist at end
void insertAtEnd(Node*& head, Node* newNode) {

    if(head == NULL) {
        head = newNode;
        return;
    }

    Node* temp = head;
    while(temp->next != NULL)
        temp = temp->next;
```

```
    temp->next = newNode;
}

// Display GLL
void displayGLL(Node* head) {
    if(head == NULL) {
        cout << "()";
        return;
    }

    cout << "(";
    Node* temp = head;

    while(temp != NULL) {
        if(temp->flag == 0) {
            cout << temp->data;
        }
        else {
            displayGLL(temp->down);   // recursive print of sublist
        }

        if(temp->next != NULL)
            cout << ", ";

        temp = temp->next;
    }

    cout << ")";
}
```

```
int main() {

    Node* head = NULL;

    // Building a GLL of form: (1, 2, (3, 4), 5)

    insertAtEnd(head, createDataNode(1));
    insertAtEnd(head, createDataNode(2));

    // Create sublist (3, 4)
    Node* sub = NULL;
    insertAtEnd(sub, createDataNode(3));
    insertAtEnd(sub, createDataNode(4));

    insertAtEnd(head, createSublistNode(sub));    // insert sublist

    insertAtEnd(head, createDataNode(5));

    cout << "Generalized Linked List: ";
    displayGLL(head);
    cout << endl;

    return 0;
}
```

Q.15 Write a program to perform addition of polynomial using single linked list

```
#include<iostream>
using namespace std;
class Node {
public:
    int coeff;
```

```
    int power;
    Node* next;
    Node(int c, int p) {
        coeff = c;
        power = p;
        next = NULL;
    }
};
// Insert term at end
void insertAtEnd(Node*& head, Node*& tail, int coeff, int power) {
    Node* temp = new Node(coeff, power);

    if (head == NULL) {
        head = tail = temp;
    }
    else {
        tail->next = temp;
        tail = temp;
    }
}

// Print polynomial
void printPolynomial(Node* head) {
    if (head == NULL) {
        cout << "0\n";
        return;
    }

    Node* temp = head;
    while (temp != NULL) {
        cout << temp->coeff << "x^" << temp->power;
```

```
        if (temp->next != NULL)
            cout << " + ";
        temp = temp->next;
    }
    cout << endl;
}

// Add two polynomials
Node* addPolynomial(Node* p1, Node* p2) {

    Node* head = NULL;
    Node* tail = NULL;

    while (p1 != NULL && p2 != NULL) {

        if (p1->power == p2->power) {
            insertAtEnd(head, tail, p1->coeff + p2->coeff, p1->power);
            p1 = p1->next;
            p2 = p2->next;
        }
        else if (p1->power > p2->power) {
            insertAtEnd(head, tail, p1->coeff, p1->power);
            p1 = p1->next;
        }
        else {
            insertAtEnd(head, tail, p2->coeff, p2->power);
            p2 = p2->next;
        }
    }

    // Add remaining terms of p1
```

```
    while (p1 != NULL) {
        insertAtEnd(head, tail, p1->coeff, p1->power);
        p1 = p1->next;
    }

    // Add remaining terms of p2
    while (p2 != NULL) {
        insertAtEnd(head, tail, p2->coeff, p2->power);
        p2 = p2->next;
    }

    return head;
}

int main() {

    Node* p1 = NULL;
    Node* t1 = NULL;

    Node* p2 = NULL;
    Node* t2 = NULL;

    // Polynomial 1: 5x^3 + 4x^2 + 2x^1 + 1
    insertAtEnd(p1, t1, 5, 3);
    insertAtEnd(p1, t1, 4, 2);
    insertAtEnd(p1, t1, 2, 1);
    insertAtEnd(p1, t1, 1, 0);

    // Polynomial 2: 3x^3 + 6x^1 + 2
    insertAtEnd(p2, t2, 3, 3);
    insertAtEnd(p2, t2, 6, 1);
```

```
    insertAtEnd(p2, t2, 2, 0);

    cout << "Polynomial 1: ";
    printPolynomial(p1);

    cout << "Polynomial 2: ";
    printPolynomial(p2);

    Node* result = addPolynomial(p1, p2);

    cout << "Result (P1 + P2): ";
    printPolynomial(result);

    return 0;
}
```

13 Write a program to implement Shell Sort & Radix Sort to analyse their performance on the same input array. Track and display the time taken, number of swaps, and comparisons, and generate a report with the sorted output and performance insights.

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

int main() {
    int arr[] = {5, 3, 21, 4, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    for (int gap = n / 2; gap >= 1; gap /= 2) {
        for (int j = gap; j < n; j++) {
            for (int i = j - gap; i >= 0; i -= gap) {
                if (arr[i + gap] > arr[i]) {
                    break;
```

```
            } else {
                swap(arr[i], arr[i + gap]);
            }
        }
    }
}

for (int k = 0; k < n; k++) {
    cout << arr[k] << "  ";
}

return 0;
}
```

RADIX SORT :

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

// Function to get maximum element
int getMax(int a[], int n) {
    int max = a[0];
    for (int i = 1; i < n; i++) {
        if (a[i] > max)
            max = a[i];
    }
    return max;
}
// Count Sort for a specific digit position
void countSort(int a[], int n, int pos) {
    int Count[10] = {0};
    int b[n];

    // Count the occurrences of each digit
```

```
    for (int i = 0; i < n; i++)
        Count[(a[i] / pos) % 10]++;

    // Prefix sum (cumulative count)
    for (int i = 1; i < 10; i++)
        Count[i] = Count[i] + Count[i - 1];

    // Build output array (RIGHT TO LEFT for stability)
    for (int i = n - 1; i >= 0; i--) {
        int digit = (a[i] / pos) % 10;
        b[ Count[digit] - 1 ] = a[i];
        Count[digit]--;
    }

    // Copy back to original
    for (int i = 0; i < n; i++)
        a[i] = b[i];
}

// Radix Sort
void radixSort(int a[], int n) {
    int max = getMax(a, n);

    // Apply counting sort for every digit
    for (int pos = 1; max / pos > 0; pos = pos * 10) {
        countSort(a, n, pos);
    }
}
int main() {
    int a[] = {530, 90, 7, 199, 677};
    int n = sizeof(a) / sizeof(a[0]);

    radixSort(a, n);

    cout << "Sorted array: ";
```

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

    return 0;
}
```

Bubble sort

```
#include <iostream>
#include <algorithm>   // for swap()
using namespace std;

int main() {
    int arr[] = {5, 1, 4, 2, 8};
    int n = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }

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

    return 0;
}
```

INSERTION SORT

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

```
int main() {

    int arr[] = {1, 4, 5, 2, 3};
    int n = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < n - 1; i++) {
        int j = i + 1;

        while (j >= 1 && arr[j] < arr[j - 1]) {
            swap(arr[j], arr[j - 1]);
            j--;
        }
    }

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

    return 0;
}
```

QUICK SORT

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

int partitionArray(int arr[], int low, int high) {
    int pivot = arr[low];
    int i = low + 1;
    int j = high;

    while (true) {
```

```
        // move i forward until an element > pivot is found
        while (i <= j && arr[i] <= pivot) {
            i++;
        }

        // move j backward until an element < pivot is found
        while (i <= j && arr[j] >= pivot) {
            j--;
        }

        // if pointers cross — stop
        if (i > j)
            break;

        swap(arr[i], arr[j]);
    }

    // place pivot in correct position
    swap(arr[low], arr[j]);

    return j;   // new pivot position
}
void quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partitionArray(arr, low, high);

        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}
int main() {
    int arr[] = {10, 7, 9, 3, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quicksort(arr, 0, n - 1);
```

```
    // to print array
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

BINARY SEARCH

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

int main() {

    int arr[] = {5, 10, 15, 20, 25, 30};
    int n = 6;

    int key;
    cout << "Enter element to search: ";
    cin >> key;

    int low = 0, high = n - 1;
    int mid, comp = 0;
    int pos = -1;

    while (low <= high) {

        comp++;
        mid = (low + high) / 2;

        if (arr[mid] == key) {
            pos = mid;
```

```
            break;
        }
        else if (arr[mid] < key) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }

    if (pos != -1)
        cout << "Element found at index: " << pos << endl;
    else
        cout << "Element not found!" << endl;

    cout << "Comparisons = " << comp << endl;

    return 0;
}
```

complexity (Best, Average, Worst Case) using Big-O, Θ, and Ω notations.

5) Write a program to Implement basic array operations such as insert, delete, and merge, while simulating a 2D array using a single-dimensional array to optimize space (using both row-major and column-major formats). Use pointer arithmetic to print memory addresses of elements, helping you understand how arrays are stored in memory.

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

// ------------------ PRINT USING POINTER ARITHMETIC ------------------
void printRowMajor(int *arr, int r, int c) {
    cout << "\nRow-major order:\n";
```

```
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {

            int index = i * c + j;      // row-major
            cout << arr[index] << "(" << (arr + index) << ")  ";
        }
        cout << endl;
    }
}

void printColMajor(int *arr, int r, int c) {
    cout << "\nColumn-major order:\n";
    for (int j = 0; j < c; j++) {
        for (int i = 0; i < r; i++) {

            int index = j * r + i;      // column-major
            cout << arr[index] << "(" << (arr + index) << ")  ";
        }
        cout << endl;
    }
}

// ------------------ INSERT INTO 1D ARRAY ------------------
void insertElement(int arr[], int &n, int pos, int val) {
    for (int i = n; i > pos; i--)
        arr[i] = arr[i - 1];

    arr[pos] = val;
    n++;
}
```

```
// ------------------ DELETE FROM 1D ARRAY ------------------
void deleteElement(int arr[], int &n, int pos) {
    for (int i = pos; i < n - 1; i++)
        arr[i] = arr[i + 1];

    n--;
}

// ------------------ MERGE TWO ARRAYS ------------------
void mergeArrays(int a[], int n1, int b[], int n2, int merged[]) {
    for (int i = 0; i < n1; i++)
        merged[i] = a[i];

    for (int j = 0; j < n2; j++)
        merged[n1 + j] = b[j];
}

// ------------------ MAIN FUNCTION ------------------
int main() {

    int r = 2, c = 3;
    int size = r * c;

    int arr[50] = {1, 2, 3, 4, 5, 6};
    int n = size;

    cout << "2D array stored in 1D array:\n";
    printRowMajor(arr, r, c);
    printColMajor(arr, r, c);

    // ---------------- INSERT ----------------
```

```
    cout << "\nInserting 99 at linear index 2...\n";
    insertElement(arr, n, 2, 99);
    printRowMajor(arr, r, c);   // r,c no longer exact matrix but shows memory

    // ---------- DELETE ----------
    cout << "\nDeleting element at linear index 4...\n";
    deleteElement(arr, n, 4);
    printRowMajor(arr, r, c);

    // ---------- MERGE ----------
    cout << "\nMerging with another array...\n";
    int arr2[3] = {10, 20, 30};
    int merged[100];

    mergeArrays(arr, n, arr2, 3, merged);

    cout << "Merged array: ";
    for (int i = 0; i < n + 3; i++)
        cout << merged[i] << " ";
    cout << endl;

    return 0;
}
```