Program 1

1. InsertElement(arr[], n, k, item):
   1.1 Initialize j to n.
   1.2 While j >= k, move arr[j] to arr[j + 1] and decrement j.
   1.3 Place item at arr[k] and increment n.

2. main():
   2.1 arr[20], n, lb=0, ub=9, pos, ele.
   2.2 Input elements into arr from lb to ub.
   2.3 Input pos and ele.
   2.4 Call InsertElement with arr, address of ub, pos - 1, and ele.
   2.5 Display upgraded array from lb to ub.

3. End

Program 2

1. INSERT(LA[], N, K, ITEM):
   1.1 Initialize J to N.
   1.2 While J >= K, move LA[J] to LA[J + 1] and decrement J.
   1.3 Place ITEM at LA[K] and increment N.

2. main():
   2.1 LA[20], i, K, LB=0, UB=9, POS, ELE.
   2.2 Input elements into LA from LB to UB.
   2.3 Input ELE to be inserted in LA.
   2.4 Find POS where ELE should be inserted.
   2.5 Call INSERT with LA, address of UB, POS, and ELE.
   2.6 Initialize K to LB.
   2.7 While K <= UB, display K and LA[K], then increment K.

3. End

Program 3

1. DeleteElement(arr[], ub, element):
   1.1 Initialize pos to -1.
   1.2 Loop through arr to find pos of element.
   1.3 If pos != -1, shift elements from pos to ub to the left and decrement ub.

2. main():
   2.1 arr[20], lb=0, ub=9, ele.
   2.2 Input elements into arr from lb to ub.
   2.3 Input ele to be deleted from arr.
   2.4 Call DeleteElement with arr, address of ub, and ele.
   2.5 Loop through arr from lb to ub and display elements.

3. End

Program 4

1. linearSearch(arr[], lb, ub, ele):
   1.1 Loop arr from lb to ub, return i + 1 if arr[i] equals ele; otherwise, return -1.

2. main():
   2.1 arr[20], lb=0, ub=9, size, ele, pos.
   2.2 Input arr elements from lb to ub.
   2.3 Input ele to be searched in arr.
   2.4 Set pos as linearSearch result with arr, lb, ub, and ele.
   2.5 Display "Element ele found at position pos" if pos != -1; otherwise, display "Element ele not found".

3. End

Program 5

1. BinarySearch(arr[], lb, ub, ele):
   1.1 While lb <= ub, compute mid = lb + (ub - lb) / 2.
   1.2 If arr[mid] == ele, return mid + 1.
   1.3 If arr[mid] < ele, set lb = mid + 1; otherwise, set ub = mid - 1.
   1.4 Return -1.

2. main():
   2.1 arr[] = {2, 5, 8, 12, 16, 23, 38, 42, 47, 50}, lb = 0, ub = 9, ele, pos.
   2.2 Display arr elements.
   2.3 Input ele.
   2.4 Set pos = BinarySearch(arr, lb, ub, ele).
   2.5 Display "Element ele found at position pos" if pos != -1; otherwise, display "Element ele not found".

Program 6

1. BinarySearch(arr[], lb, ub, ele):
   1.1 If lb <= ub:
      1.1.1 Compute mid = lb + (ub - lb) / 2.
      1.1.2 If arr[mid] == ele, return mid + 1.
      1.1.3 If arr[mid] < ele, return BinarySearch(arr, mid + 1, ub, ele).
      1.1.4 Else, return BinarySearch(arr, lb, mid - 1, ele).
   1.2 Return -1.

2. main():
   2.1 arr[] = {2, 5, 8, 12, 16, 23, 38, 42, 47, 50}, lb = 0, ub = 9, ele, position.
   2.2 Display arr elements.
   2.3 Input ele.
   2.4 Set position as BinarySearch result with arr, lb, ub, and ele.
   2.5 Display "Element ele found at position position" if position != -1; otherwise, display "Element not found".

3. End

Program 7

1. struct Term: int coefficient, exponent.

2. inputPolynomial(poly[]):
   2.1 Read num.
   2.2 Loop i: Read poly[i].coefficient, poly[i].exponent.
   2.3 Set poly[num].exponent = -1.

3. addPolynomials(p1[], p2[], res[]):
   3.1 Initialize i, j, k to 0.
   3.2 While p1[i].exponent != -1 and p2[j].exponent != -1:
      3.2.1 If p1[i].exponent > p2[j].exponent, res[k++] = p1[i++].
      3.2.2 If p1[i].exponent < p2[j].exponent, res[k++] = p2[j++].
      3.2.3 Else, res[k] = p1[i] + p2[j], increment k, i, and j.
   3.3 While p1[i].exponent != -1, res[k++] = p1[i++].
   3.4 While p2[j].exponent != -1, res[k++] = p2[j++].
   3.5 Set res[k].exponent = -1.

4. displayPolynomial(poly[]):
   4.1 Loop i while poly[i].exponent != -1: Print poly[i].coefficient, poly[i].exponent.
   4.2 Print newline.

5. main():
   5.1 struct Term poly1[30], poly2[30], result[60].
   5.2 Call inputPolynomial(poly1).
   5.3 Call inputPolynomial(poly2).
   5.4 Call addPolynomials(poly1, poly2, result).
   5.5 DisplayPolynomial(poly1).
   5.6 DisplayPolynomial(poly2).
   5.7 DisplayPolynomial(result).

Program 8

1. struct sparse: int row, col, ele.

2. main():
   2.1 Initialize tot_ele, tot_row, tot_col, ptr=1, i, j, k, flag=0, data.
   2.2 Read tot_row and tot_col, set s[0].row = tot_row, s[0].col = tot_col.
   2.3 Loop i from 0 to tot_row:
      2.3.1 Loop j from 0 to tot_col:
         2.3.1.1 Read data.
         2.3.1.2 If data != 0, set s[ptr].row = i, s[ptr].col = j, s[ptr].ele = data, increment ptr and tot_ele.
   2.4 Set s[0].ele = tot_ele.
   2.5 Print "ROW   COL   DATA".
   2.6 Loop i from 0 to tot_ele:
      2.6.1 Print s[i].row, s[i].col, s[i].ele.
   2.7 Set ptr=1.
   2.8 Print "DISPLAYING THE FINAL MATRIX".
   2.9 Loop i from 0 to tot_row:
      2.9.1 Loop j from 0 to tot_col:
         2.9.1.1 If s[ptr].row==i and s[ptr].col==j, print s[ptr].ele and increment ptr.
         2.9.1.2 Else, print " 0 ".
      2.9.2 Print newline.

3. End

Program 9

1. struct Student: char name[50], int rollNo, struct Student *next.

2. InsertNode(name, rollNo, start, pos):
   2.1 Allocate memory for ele.
   2.2 Copy name and rollNo to ele.
   2.3 Set ele->next to NULL.
   2.4 If *start is NULL, set *start to ele.
   2.5 Else,
      2.5.1 If pos == -1, traverse to the end and add ele.
      2.5.2 Else, insert ele at specified pos.

3. DeleteNodeWithRollNumber(start, rollNo):
   3.1 Set ele_next to (*start)->next, prev to *start.
   3.2 While ele_next is not NULL,
      3.2.1 If rollNo == ele_next->rollNo, remove and free ele_next.
      3.2.2 Update pointers and return.
   3.3 Print "Node not found."

4. ReverseLinkedList(start):
   4.1 Set prev, current, and nextNode to NULL.
   4.2 While current is not NULL,
      4.2.1 Update pointers for reversing the linked list.
   4.3 Set *start to prev.
   4.4 Print "Linked List Reversed."

5. main():
   5.1 struct Student* start = NULL.
   5.2 InsertNode("ABC", 1, &start, -1).
   5.3 InsertNode("EFG", 2, &start, -1).
   5.4 InsertNode("HIK", 3, &start, -1).
   5.5 InsertNode("IJK", 4, &start, 2).
   5.6 DeleteNodeWithRollNumber(&start, 4).
   5.7 ReverseLinkedList(&start).

Program 10

1. struct Employee: char name[50], int empId, struct Employee* next, struct Employee* prev.

2. InsertAtFront(start, name, empId):
   2.1 Allocate memory for ele.
   2.2 Copy name, empId to ele.
   2.3 Set ele->next, ele->prev to NULL.
   2.4 If *start is NULL, set *start to ele.
   2.5 Else, set ele->next to *start, set (*start)->prev to ele.

3. DeletetionAtEnd(start):
   3.1 If *start is not NULL:
      3.1.1 Set ele_curr to *start.
      3.1.2 While ele_curr->next is not NULL, update ele_curr.
      3.1.3 Set temp to ele_curr->prev, temp->next to NULL.
      3.1.4 Print "Node Deleted" and free ele_curr.
   3.2 Else, print "Linked List Empty."

4. main():
   4.1 struct Employee* start = NULL.
   4.2 InsertAtFront(&start, "ABC", 001).
   4.3 InsertAtFront(&start, "EFG", 002).
   4.4 InsertAtFront(&start, "IJK", 003).
   4.5 DeletetionAtEnd(&start).

Program 11

1. struct Node: char collegeName[50], int rank, struct Node* next.

2. insertAtFront(start, name, rank):
   2.1 Allocate memory for N.
   2.2 Copy name and rank to N.
   2.3 If *start is NULL, set N->next to N, *start to N.
   2.4 Else, set N->next to (*start)->next, (*start)->next to N.

3. insertAtEnd(start, name, rank):
   3.1 Allocate memory for N.
   3.2 Copy name and rank to N.
   3.3 If *start is NULL, set N->next to N, *start to N.
   3.4 Else, set N->next to (*start)->next, (*start)->next to N, *start to N.

4. main():
   4.1 struct Node* start = NULL.
   4.2 Call insertAtFront(&start, "College A", 1).
   4.3 Call insertAtEnd(&start, "College B", 5).

Program 12

1. int stack[100], top = -1.

2. push(rollNumber):
   2.1 If top is 99, print "Stack Overflow".
   2.2 Else, increment top, set stack[top] to rollNumber, and print "Pushed: rollNumber".

3. pop():
   3.1 If top is -1, print "Stack Underflow" and return -1.
   3.2 Else, set PV to stack[top], decrement top, and return PV.

4. display():
   4.1 If top is -1, print "Stack is empty".
   4.2 Else, print "Stack elements:" followed by stack values.

5. main():
   5.1 Call push(10), push(20), display().
   5.2 Set PV to pop(), print "Popped Value: PV" if PV is not -1.
   5.3 Call display().

Program 13

1. struct Node: int rollNumber, struct Node* next.

2. push(start, rollNumber):
   2.1 Allocate memory for N.
   2.2 Set N->rollNumber to rollNumber, N->next to *start, *start to N.
   2.3 Print "Pushed: rollNumber".

3. pop(start):
   3.1 If *start is NULL, print "Stack Underflow" and return -1.
   3.2 Set PN to *start, PV to PN->rollNumber, *start to PN->next, free PN, and return PV.

4. display(start):
   4.1 If start is NULL, print "Stack is empty".
   4.2 Else, print "Stack elements:" followed by rollNumber values.

5. main():
   5.1 struct Node* start = NULL.
   5.2 Call push(&start, 101), push(&start, 102), display(start).
   5.3 Set PV to pop(&start), print "Popped Value: PV" if PV is not -1.
   5.4 Call display(start).

Program 14

1. int empNumbers[100], front = -1, rear = -1.

2. Insert(empNumber):
   2.1 If rear is 99, print "Queue Overflow".
   2.2 If front is -1, set front and rear to 0.
   2.3 Else, increment rear.
   2.4 Set empNumbers[rear] to empNumber.
   2.5 Print "Inserted: empNumber".

3. Delete():
   3.1 If front is -1, print "Queue Underflow".
   3.2 Set DV to empNumbers[front].
   3.3 If front is equal to rear, set front and rear to -1.
   3.4 Else, increment front.
   3.5 Print "Deleted: DV".

4. display():
   4.1 If front is -1, print "Queue is empty".
   4.2 Else, print "Queue elements:" followed by empNumbers values from front to rear.

5. main():
   5.1 Call Insert(1004), Insert(1005), display().
   5.2 Call Delete(), display().

Program 15

1. struct Node: int empNumber, struct Node* next.

2. Insert(front, rear, empNumber):
   2.1 Allocate memory for N, set N->empNumber to empNumber, N->next to NULL.
   2.2 If *rear is NULL, set *front and *rear to N.
   2.3 Else, set (*rear)->next to N, *rear to N.
   2.4 Print "Inserted: empNumber".

3. Delete(front, rear):
   3.1 If *front is NULL, print "Queue Underflow" and return.
   3.2 Set FN to *front, DV to FN->empNumber.
   3.3 Set *front to FN->next, free FN.
   3.4 If *front is NULL, set *rear to NULL.
   3.5 Print "Deleted: DV".

4. display(front):
   4.1 If front is NULL, print "Queue is empty".
   4.2 Else, print "Queue elements:" followed by empNumber values from front to NULL.

5. main():
   5.1 struct Node* front = NULL, *rear = NULL.
   5.2 Call Insert(&front, &rear, 1001), Insert(&front, &rear, 1002), display(front).
   5.3 Call Delete(&front, &rear), display(front).
   5.4 Print "Abhay Raj, 00976803122".

Program 16

1. struct Lab: char name[50], int num, struct Lab* next.

2. Insert(front, rear, name, num):
   2.1 Allocate memory for N, set N->name to name, N->num to num, N->next to NULL.
   2.2 If *rear is NULL, set *front and *rear to N.
   2.3 Else, set (*rear)->next to N, *rear to N.
   2.4 Print "Inserted: Lab Name: name, Lab Number: num".

3. Delete(front, rear):
   3.1 If *front is NULL, print "Queue Underflow" and return.
   3.2 Set FN to *front.
   3.3 Print "Deleted: Lab Name: FN->name, Lab Number: FN->num".
   3.4 Set *front to FN->next, free FN.
   3.5 If *front is NULL, set *rear to NULL.

4. Display(front):
   4.1 If front is NULL, print "Queue is empty" and return.
   4.2 Set c to front, Print "Queue elements:".
   4.3 Do:
      4.3.1 Print "Lab Name: c->name, Lab Number: c->num".
      4.3.2 Set c to c->next.
   4.4 While c is not NULL and c is not equal to front.

5. main():
   5.1 struct Lab* front = NULL, *rear = NULL.
   5.2 Call Insert(&front, &rear, "Physics Lab", 24), Insert(&front, &rear, "Chemistry Lab", 17), Insert(&front, &rear, "Biology Lab", 9), Display(front).
   5.3 Call Delete(&front, &rear), Display(front).

Program 17

1. struct Automobile: char type[50], char company[50], int year.

2. struct Tree: struct Automobile data, struct Tree* left, struct Tree* right.

3. createNode(type, company, year):
3.1 Allocate memory for N, set N->data.type to type, N->data.company to company, N->data.year to year.
3.2 Set N->left and N->right to NULL.
3.3 Return N.

4. Insert(R, type, company, year):
4.1 If R is NULL, return createNode(type, company, year).
4.2 If year < R->data.year, R->left = Insert(R->left, type, company, year).
4.3 Else, R->right = Insert(R->right, type, company, year).
4.4 Return R.

5. findMin(N):
5.1 While N->left is not NULL, set N = N->left.
5.2 Return N.

6. Delete(R, year):
6.1 If R is NULL, return R.
6.2 If year < R->data.year, R->left = Delete(R->left, year).
6.3 Else if year > R->data.year, R->right = Delete(R->right, year).
6.4 Else:
6.4.1 If R->left is NULL, set temp = R->right, free R, return temp.
6.4.2 If R->right is NULL, set temp = R->left, free R, return temp.
6.4.3 Set temp = findMin(R->right), R->data = temp->data, R->right = Delete(R->right, temp->data.year).
6.5 Return R.

7. dispPost(R):
7.1 If R is not NULL, dispPost(R->left), dispPost(R->right), display(R).

8. dispIn(R):
8.1 If R is not NULL, dispIn(R->left), display(R), dispIn(R->right).

9. dispPre(R):
9.1 If R is not NULL, display(R), dispPre(R->left), dispPre(R->right).

10. display(R):
10.1 Print "Type: R->data.type, Company: R->data.company, Year: R->data.year".

11. main():
11.1 struct Tree* R = NULL.
11.2 R = Insert(R, "Sedan", "Tata", 2022).
11.3 R = Insert(R, "SUV", "Nissan", 2020).
11.4 R = Insert(R, "Hatchback", "Ford", 2021).

11.5 R = Insert(R, "Convertible", "Chevrolet", 2019).
11.6 Print "Postorder Traversal:"; dispPost(R).
11.7 Print "Inorder Traversal:"; dispIn(R).
11.8 Print "Preorder Traversal:"; dispPre(R).
11.9 R = Delete(R, 2021).
11.10 Print "Node with Year 2021 Deleted".
11.11 Print "Inorder Traversal:"; dispIn(R).

Program 18

1. SelectionSort(arr[], n):
1.1 For i = 0 to n-2:
1.1.1 Set minIndex = i.
1.1.2 For j = i+1 to n-1:
If arr[j] < arr[minIndex], set minIndex = j.
1.1.3 Swap(arr[minIndex], arr[i]).

2. BubbleSort(arr[], n):
2.1 For i = 0 to n-2:
2.1.1 For j = 0 to n-i-2:
If arr[j] > arr[j+1], Swap(arr[j], arr[j+1]).

3. InsertionSort(arr[], n):
3.1 For i = 1 to n-1:
3.1.1 Set key = arr[i].
3.1.2 Set j = i - 1.
3.1.3 While j >= 0 and arr[j] > key:
Move arr[j] to arr[j+1], decrement j.
3.1.4 Set arr[j+1] = key.

4. QuickSort(arr[], low, high):
4.1 If low < high:
4.1.1 Set pi = Partition(arr, low, high).
4.1.2 Call QuickSort(arr, low, pi-1).
4.1.3 Call QuickSort(arr, pi+1, high).

5. Partition(arr[], low, high):
5.1 Set pi = arr[high], i = low - 1.
5.2 For j = low to high-1:
If arr[j] < pi, Swap(arr[++i], arr[j]).
5.3 Swap(arr[i+1], arr[high]).
5.4 Return (i+1).

6. MergeSort(arr[], l, r):
6.1 If l < r:
6.1.1 Set m = l + (r - l) / 2.
6.1.2 Call MergeSort(arr, l, m).
6.1.3 Call MergeSort(arr, m+1, r).
6.1.4 Merge(arr, l, m, r).

7. Merge(arr[], l, m, r):
7.1 Set n1 = m - l + 1, n2 = r - m.
7.2 Declare L[n1], R[n2].
7.3 Copy data to L[0 to n1-1] and R[0 to n2-1].
7.4 Merge the two halves back into arr[l to r].

8. RadixSort(arr[], n):
8.1 Find the maximum number to know the number of digits.

8.2 For each digit place (1s, 10s, 100s, ...):
   8.2.1 Count the occurrences in each bucket.
   8.2.2 Adjust the count to get the correct position.
   8.2.3 Copy the elements to the output array.
   8.2.4 Copy the output array back to arr[].

9. Print(arr[], n):
   9.1 For i = 0 to n-1, print arr[i], followed by a space.
   9.2 Print a newline.

10. Swap(a, b):
   10.1 Set temp = a, a = b, b = temp.

11. main():
   11.1 Declare an array arr with values {56, 77, 23, 99, 68, 11, 9, 29, 33, 45, 10, 87}.
   11.2 Set n as the size of arr.
   11.3 Print "Original array:", call Print(arr, n).
   11.4 Call SelectionSort(arr, n), Print "Selection Sort:", call Print(arr, n).
   11.5 Call BubbleSort(arr, n), Print "Bubble Sort:", call Print(arr, n).
   11.6 Call InsertionSort(arr, n), Print "Insertion Sort:", call Print(arr, n).
   11.7 Call QuickSort(arr, 0, n-1), Print "Quick Sort:", call Print(arr, n).
   11.8 Call MergeSort(arr, 0, n-1), Print "Merge Sort:", call Print(arr, n).
   11.9 Call RadixSort(arr, n), Print "Radix Sort:", call Print(arr, n).

Program 19

1. Function createN(data):
   1.1 Create a new node (N*) with the given data.
   1.2 Set the next pointer of the new node to NULL.
   1.3 Return the new node.

2. Function createGh(V):
   2.1 Create a new graph (Gh*) with V vertices.
   2.2 Allocate memory for the adjacency list array adjL.
   2.3 Initialize each element of adjL to NULL.
   2.4 Return the graph.

3. Function AddEdge(gh, src, dest):
   3.1 Create a new node with destination 'dest'.
   3.2 Set the next pointer of the new node to the current adjacency list at index 'src'.
   3.3 Update the adjacency list at index 'src' to point to the new node.
   3.4 Print "Edge added: src -> dest".
   3.5 Repeat steps 3.1-3.4 with reversed roles for an undirected graph.

4. Function BFS(gh, start):
   4.1 Allocate memory for an array 'done' to track visited vertices (initialize to false).
   4.2 Allocate memory for a queue 'Q' to perform BFS.
   4.3 Set front = rear = 0.
   4.4 Mark 'start' as visited and enqueue it.
   4.5 Print "BFS from start: ".
   4.6 While the queue is not empty:
      4.6.1 Dequeue a vertex 'current' from the queue.
      4.6.2 Print 'current'.
      4.6.3 Enqueue all adjacent vertices of 'current' that are not visited.
   4.7 Free allocated memory for 'done' and 'Q'.

5. Function Util(gh, vx, done):
   5.1 Mark vertex 'vx' as visited.
   5.2 Print 'vx'.
   5.3 For each adjacent vertex 'adjVx' of 'vx':
      5.3.1 If 'adjVx' is not visited, recursively call Util with 'adjVx' and 'done'.

6. Function DFS(gh, start):
   6.1 Allocate memory for an array 'done' to track visited vertices (initialize to false).
   6.2 Print "DFS from start: ".
   6.3 Call Util(gh, start, done).
   6.4 Free allocated memory for 'done'.

7. Function main():
   7.1 Create a graph 'gh' with 4 vertices using createGh(4).
   7.2 Add edges to the graph using AddEdge(gh, src, dest).
   7.3 Perform BFS from vertex 2 using BFS(gh, 2).
   7.4 Perform DFS from vertex 2 using DFS(gh, 2).

7.5 Free allocated memory for the graph and its adjacency list.

Program 20

1. Function CreateGh(vx):
   1.1 Create a new graph (Gh*) with vx vertices.
   1.2 Allocate memory for the adjacency matrix adjMat.
   1.3 Initialize all elements of adjMat to 0.
   1.4 Return the graph.

2. Function AddEdge(G, s, d, w):
   2.1 Set the weight w for the edge between vertices s and d in the adjacency matrix.
   2.2 Set the weight w for the edge between vertices d and s in the adjacency matrix.

3. Function ShortestPath(G, src):
   3.1 Create an array dist[V] to store the shortest distance from src to each vertex.
   3.2 Create an array sptSet[V] to track visited vertices.
   3.3 Initialize dist[] to INT_MAX and sptSet[] to false.
   3.4 Set dist[src] to 0.
   3.5 Repeat the following V-1 times:
      3.5.1 Find the vertex with the minimum distance value from the set of vertices not yet included in the shortest path tree (sptSet[]).
      3.5.2 Mark the selected vertex as true in sptSet[].
      3.5.3 Update the distance value of the adjacent vertices of the selected vertex.
   3.6 Print the shortest distances from src to all vertices.

4. Function main():
   4.1 Create a graph G with V vertices using CreateGh(V).
   4.2 Add edges to the graph using AddEdge(G, s, d, w).
   4.3 Find the shortest paths from source vertex 0 using ShortestPath(G, 0).
   4.4 Free allocated memory for the adjacency matrix.
   4.5 Free allocated memory for the graph.

Program 21

1. ShortestPath(graph):
   1.1 Create a 2D array dist[V][V] to store the shortest distances between every pair of vertices.
   1.2 Initialize dist[][] with the same values as the input graph[][], and set dist[i][i] to 0.
   1.3 For each intermediate vertex k from 0 to V-1:
      1.3.1 For each pair of vertices i and j from 0 to V-1:
         1.3.1.1 If vertex k is on the shortest path from i to j, update dist[i][j]:
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
   1.4 Print the final matrix dist[][].

2. main():
   2.1 Create a 2D array graph[V][V] representing the input directed weighted graph.
   2.2 Initialize the graph with appropriate weights (Max for unreachable pairs).
   2.3 Call ShortestPath(graph).