DS CODES(DEV C++)

1)Max heap

```
#include<stdio.h>
#include<stdlib.h>
int heap_size=0;
#define MAXSIZE 100
void swap(int*a,int*b)
{
	int temp=*a;
	*a=*b;
	*b=temp;
}
void heapify(int arr[],int i)
{
	int largest=i;
	int left=2*i+1;
	int right=2*i+2;
	if(left<heap_size&&arr[left]>arr[largest])
	{
		largest=left;
	}
	if(right<heap_size&&arr[right]>arr[largest])
	{
		largest=right;
	}
	if(largest!=i)
	{
		swap(&arr[i],&arr[largest]);
		heapify(arr,largest);
```

```
        }
}
void insert(int arr[],int key)
{
        if(heap_size==MAXSIZE)
        {
                printf("heap is overflow\n");
                return;
        }
        heap_size++;
        arr[heap_size-1]=key;
        int i=heap_size-1;
        while(i>0&&arr[(i-1)/2]<arr[i]){
                swap(&arr[(i-1)/2],&arr[i]);
                i=(i-1)/2;
        }
}
void delete_max(int arr[]) {
 if (heap_size == 0) {
        printf("Heap Underflow\n");
   return;
 }
 arr[0] = arr[heap_size - 1];
 heap_size--;
 heapify(arr, 0);
}
void display(int arr[]) {
        int i;
 if (heap_size == 0) {
```

```
 printf("Heap is empty\n");
 return;
 }
 for ( i = 0; i < heap_size; ++i)
 printf("%d ", arr[i]);
 printf("\n");
}
int main()
{
        int key,choice;
        int arr[MAXSIZE];
        while(1)
        {
                printf("\n1.insert\n2.delete\n3.display\n4.exit\n");
                printf("enter a choice:\n");
                scanf("%d",&choice);
                switch(choice){
                        case 1:printf("enter a key to insert\n");
                                        scanf("%d",&key);
                                        insert(arr,key);
                                        break;
                        case 2:delete_max(arr);
                                        break;
                        case 3:display(arr);
                                        break;
                        case 4:exit(0);
                        default:printf("invalid choice\n");
                }
        }
```

```
	return 0;
}
```

2)Min heap

```
#include<stdio.h>
#include<stdlib.h>
int heap_size=0;
#define MAXSIZE 100
void swap(int*a,int*b)
{
	int temp=*a;
	*a=*b;
	*b=temp;
}
void heapify(int arr[],int i)
{
	int smallest=i;
	int left=2*i+1;
	int right=2*i+2;
	if(left<heap_size&&arr[left]<arr[smallest])
	{
		smallest=left;
	}
	if(right<heap_size&&arr[right]<arr[smallest])
	{
		smallest=right;
	}
	if(smallest!=i)
	{
		swap(&arr[i],&arr[smallest]);
```

```
                heapify(arr,smallest);
        }
}
void insert(int arr[],int key)
{
        if(heap_size==MAXSIZE)
        {
                printf("heap is overflow\n");
                return;
        }
        heap_size++;
        arr[heap_size-1]=key;
        int i=heap_size-1;
        while(i>0&&arr[(i-1)/2]>arr[i]){
                swap(&arr[(i-1)/2],&arr[i]);
                i=(i-1)/2;
        }
}
void delete_min(int arr[]) {
 if (heap_size == 0) {
        printf("Heap Underflow\n");
   return;
 }
 arr[0] = arr[heap_size - 1];
 heap_size--;
 heapify(arr, 0);
}
void display(int arr[]) {
        int i;
```

```
 if (heap_size == 0) {
 printf("Heap is empty\n");
 return;
 }
 for ( i = 0; i < heap_size; ++i)
 printf("%d ", arr[i]);
 printf("\n");
}
int main()
{
        int key,choice;
        int arr[MAXSIZE];
        while(1)
        {
                printf("\n1.insert\n2.delete\n3.display\n4.exit\n");
                printf("enter a choice:\n");
                scanf("%d",&choice);
                switch(choice){
                        case 1:printf("enter a key to insert\n");
                                        scanf("%d",&key);
                                        insert(arr,key);
                                        break;
                        case 2:delete_min(arr);
                                        break;
                        case 3:display(arr);
                                        break;
                        case 4:exit(0);
                        default:printf("invalid choice\n");
                }
```

```
        }
        return 0;
}
```

3)Avl tree

```
#include<stdio.h>
#include<stdlib.h>
struct AVLNode{
        int key,height;
        struct AVLNode*left;
        struct AVLNode*right;
};
int height(struct AVLNode*node)
{
        return node?node->height:0;
}
void updateheight(struct AVLNode*node)
{
        node->height=1+(height(node->left)>height(node->right)?height(node->left):height(node->right));
}
struct AVLNode*rightRotate(struct AVLNode*y)
{
        struct AVLNode*x=y->left;
        y->left=x->right;
        x->right=y;
        updateheight(y);
        updateheight(x);
        return x;
}
struct AVLNode*leftRotate(struct AVLNode*x)
{
        struct AVLNode*y=x->right;
```

```
        x->right=y->left;
        y->left=x;
        updateheight(x);
        updateheight(y);
        return y;
}
int getbalance(struct AVLNode*node){
        return node?height(node->left)-height(node->right):0;
}
struct AVLNode*insert(struct AVLNode*node,int key)
{
        if(!node)
        {
        struct AVLNode*newNode=(struct AVLNode*)malloc(sizeof(struct AVLNode));
        newNode->key=key;
        newNode->height=1;
        newNode->left=newNode->right=NULL;
        return newNode;
   }
 if(key<node->key)
 {
        node->left=insert(node->left,key);
 }
 else if(key>node->key)
 {
        node->right=insert(node->right,key);
 }
 else
 {
        return node;
 }
updateheight(node);
int balance=getbalance(node);
```

```
if(balance>1&&key<node->left->key)
{
        return rightRotate(node);
}
if(balance<-1&&key>node->right->key)
{
return leftRotate(node);
}

  // Left-Right case
if (balance > 1 && key > node->left->key) {
    node->left = leftRotate(node->left);
    return rightRotate(node);
  }

  // Right-Left case
if (balance < -1 && key < node->right->key) {
    node->right = rightRotate(node->right);
    return leftRotate(node);
  }

return node;
}

// Inorder traversal to print the tree
void inorder(struct AVLNode* root) {
  if (root) {
    inorder(root->left);
    printf("%d ", root->key);
    inorder(root->right);
  }

}
```

```
int main() {
   struct AVLNode* root = NULL;
   printf("1.Insert\n2.Exit\n3.Display\n");
   while(1)
   {
      int choice,key;
      printf("\nenter choice:");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:printf("enter key to insert:");
            scanf("%d",&key);
            root = insert(root, key);
            break;
         case 2:printf("Exiting");
         exit(0);
         break;
         case 3:printf("Inorder traversal of the AVL tree: ");
   inorder(root);
   break;
      }
   }
   // Inorder traversal
   printf("\n");
}
```

4)QuickSort

```
#include <stdio.h>
void swap(int* a, int* b) {
 int temp = *a;
 *a = *b;
 *b = temp;
}
int partition(int arr[], int low, int high) {
```

```
 int p = arr[low];
 int i = low;
 int j = high;
 while (i < j) {
 while (arr[i] <= p) i++;
 while (arr[j] > p) j--;
 if (i < j) swap(&arr[i], &arr[j]);
 }
 swap(&arr[low], &arr[j]);
 return j;
}
void quickSort(int arr[], int low, int high) {
 if (low < high) {
 int pi = partition(arr, low, high);
 quickSort(arr, low, pi - 1);
 quickSort(arr, pi + 1, high);
 }
}
int main() {
 int n;
 int i;
 printf("Enter array size: ");
 scanf("%d", &n);
 int arr[n];
 printf("Enter %d elements: ", n);
 for (i = 0; i < n; i++) scanf("%d", &arr[i]);
 quickSort(arr, 0, n - 1);
 i=0;
 printf("Sorted array: ");
 for (i = 0; i < n; i++) printf("%d ", arr[i]);
 return 0;
}
```

5)MergeSort

```
#include <stdio.h>
void merge(int arr[], int left, int mid, int right)
 {
   int n1 = mid - left + 1, n2 = right - mid;
   int leftArr[n1], rightArr[n2];
   int i,j;
   for ( i = 0; i < n1; i++)
   {
            leftArr[i] = arr[left + i];
   }
   for ( j = 0; j < n2; j++)
   {
     rightArr[j] = arr[mid + 1 + j];
   }
   int k = left;
   i=0;
   j=0;
   while (i < n1 && j < n2)
   {
      arr[k++] = (leftArr[i] <= rightArr[j]) ? leftArr[i++] : rightArr[j++];
   }
  while (i < n1)
  {
   arr[k++] = leftArr[i++];
  }
  while (j < n2)
   {
            arr[k++] = rightArr[j++];
   }
}
void mergeSort(int arr[], int left, int right)
{
```

```
 if (left < right)
  {
     int mid = left + (right - left) / 2;
     mergeSort(arr, left, mid);
     mergeSort(arr, mid + 1, right);
     merge(arr, left, mid, right);
  }
}
int main()
{
 int n,i;
 printf("Enter array size: ");
 scanf("%d", &n);
 int arr[n];
 printf("Enter %d elements: ", n);
 for (i = 0; i < n; i++) scanf("%d", &arr[i]);
 mergeSort(arr, 0, n - 1);
 printf("Sorted array: ");
 for (i = 0; i < n; i++) printf("%d ", arr[i]);
 return 0;
}
```

6)JobSequence

```
#include <stdio.h>
#define MAX 100
int main() {
 int n,i,j;
 int jobID[MAX], deadline[MAX], profit[MAX];
 int result[MAX]; // To store the sequence of job IDs
 int slot[MAX] = {0}; // To track free time slots
 int maxProfit = 0; // Total profit initialized to zero
 printf("Enter the number of jobs: ");
 scanf("%d", &n);
 // Input job details
```

```
for (i = 0; i < n; i++) {
printf("Enter details for job %d (ID, Deadline, Profit): ", i + 1);
scanf("%d %d %d", &jobID[i], &deadline[i], &profit[i]);
}
// Simple selection sort to sort jobs by profit in descending order
for (i = 0; i < n - 1; i++) {
for (j = 0; j < n - i - 1; j++) {
if (profit[j] < profit[j + 1]) {
// Swap profits
int tempProfit = profit[j];
profit[j] = profit[j + 1];
profit[j + 1] = tempProfit;
// Swap job IDs
int tempID = jobID[j];
jobID[j] = jobID[j + 1];
jobID[j + 1] = tempID;
// Swap deadlines
int tempDeadline = deadline[j];
deadline[j] = deadline[j + 1];
deadline[j + 1] = tempDeadline;
}
}
}
// Iterate through all jobs in sorted order
for (i = 0; i < n; i++) {
// Find a free slot for this job before its deadline
for ( j = (deadline[i] < MAX ? deadline[i] : MAX) - 1; j >= 0; j--) {
if (slot[j] == 0) { // If the slot is free
slot[j] = 1; // Mark the slot as occupied
result[j] = jobID[i]; // Store the job ID
maxProfit += profit[i]; // Add profit to total
break; // Exit the loop after assigning the job
}
```

```
}
}
// Print the result
printf("Job sequence for maximum profit:\n");
for (i = 0; i < MAX; i++) {
if (slot[i]) { // If the slot is occupied
printf("Job ID: %d\n", result[i]);
}
}
printf("Total profit: %d\n", maxProfit);
return 0;
}
```

7)KnapSack using dynamic programming

```
#include <stdio.h>
#define MAX_ITEMS 100
#define MAX_CAPACITY 1000
int max(int a, int b) {
return (a > b) ? a : b;
}
int main() {
        int n,i; // Number of items
int W; // Maximum capacity of the knapsack
int weights[MAX_ITEMS], values[MAX_ITEMS];
int dp[MAX_ITEMS + 1][MAX_CAPACITY + 1];
// Input number of items and capacity
printf("Enter the number of items: ");
scanf("%d", &n);
printf("Enter the maximum capacity of the knapsack: ");
scanf("%d", &W);
// Input weights and values
for ( i = 0; i < n; i++) {
printf("Enter weight and value for item %d: ", i + 1);
scanf("%d %d", &weights[i], &values[i]);
```

```
}
// Initialize DP table
int j;
for (i = 0; i <= n; i++) {
for (j = 0; j <= W; j++) {
if (i == 0 || j == 0) {
dp[i][j] = 0; // Base case
} else if (weights[i - 1] <= j) {
dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i - 1]] + values[i - 1]);
} else {
dp[i][j] = dp[i - 1][j]; // Item cannot be included
}
}
}
// The maximum value is found at dp[n][W]
printf("Maximum value in knapsack = %d\n", dp[n][W]);
return 0;
}
```

8)Nqueens

```
#include <stdio.h>

#define MAX 20

int board[MAX][MAX];

// Function to print the solution
void printSolution(int n) {
   int i, j;
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         if (board[i][j] == 1)
            printf("Q ");
         else
```

```
            printf(". ");
        }
        printf("\n");
    }
}

// Function to check if a queen can be placed at board[row][col]
int isSafe(int row, int col, int n) {
    int i, j;
    // Check this column on upper side
    for (i = 0; i < row; i++) {
        if (board[i][col] == 1)
            return 0; // Not safe
    }

    // Check upper left diagonal
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1)
            return 0; // Not safe
    }

    // Check upper right diagonal
    for (i = row, j = col; i >= 0 && j < n; i--, j++) {
        if (board[i][j] == 1)
            return 0; // Not safe
    }

    return 1; // Safe
}

// Backtracking function to solve the N-Queens problem
int solveNQueens(int row, int n) {
    int col;
```

```
    // If all queens are placed, return 1 (solution found)
    if (row >= n)
        return 1;

    // Consider this row and try all columns
    for (col = 0; col < n; col++) {
        // Check if it's safe to place a queen at board[row][col]
        if (isSafe(row, col, n)) {
            // Place queen
            board[row][col] = 1;

            // Recur to place the next queen
            if (solveNQueens(row + 1, n) == 1)
                return 1; // Solution found

            // If placing queen at board[row][col] doesn't lead to a solution,
            // remove the queen (backtrack)
            board[row][col] = 0;
        }
    }

    // If the queen cannot be placed in any column, return 0 (no solution)
    return 0;
}

int main() {
    int n;

    // Input the size of the board (number of queens)
    printf("Enter the number of queens: ");
    scanf("%d", &n);

    // Initialize the board with 0 (no queens placed)
```

```
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            board[i][j] = 0;

    // Solve the N-Queens problem
    if (solveNQueens(0, n) == 1) {
        printf("Solution:\n");
        printSolution(n);
    } else {
        printf("No solution exists for %d queens.\n", n);
    }

    return 0;
}
```

9)DFT Matrix

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
int visited[MAX];
void dft(int graph[MAX][MAX],int vertex,int n)
{
        printf("%d\t",vertex);
        int i;
        visited[vertex]=1;
        for(i=0;i<n;i++)
        {
                if(graph[vertex][i]==1&& !visited[i])
                {
                        dft(graph,i,n);
                }
        }
}
```

```
int main()
{
        int graph[MAX][MAX];
        int n,i,j;
        printf("enter the no of vertices:\n");
        scanf("%d",&n);
        printf("enter the adjacency matrix:\n");
        for(i=0;i<n;i++){
                for(j=0;j<n;j++){
                        scanf("%d",&graph[i][j]);
         }
        }
        for(i=0;i<n;i++){
                visited[i]=0;
        }
         printf("Depth-First Traversal starting from vertex 0:\n");
        dft(graph,0,n);
        return 0;
}
```

10)source

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <stdbool.h>
// Define a structure for an edge
struct Edge {
 int src, dest, weight;
};
// Function to print the shortest distance from the source
void printDistances(int dist[], int V) {
        int i;
 printf("Vertex\tDistance from Source\n");
 for ( i = 0; i < V; i++) {
```

```
 printf("%d\t%d\n", i, dist[i]);
 }
}
// Function to find the vertex with the minimum distance
int findMinVertex(int dist[], bool visited[], int V) {
        int i;
 int minDistance = INT_MAX;
 int minVertex = -1;
 for ( i = 0; i < V; i++) {
 if (!visited[i] && dist[i] < minDistance) {
 minDistance = dist[i];
 minVertex = i;
 }
 }
 return minVertex;
}
// Dijkstra's algorithm function
void dijkstra(int V, int E, struct Edge edges[], int src) {
        int i,j;
 int dist[V]; // Distance array
 bool visited[V]; // Visited array
 // Initialize distances and visited status
 for ( i = 0; i < V; i++) {
 dist[i] = INT_MAX; // Set all distances to infinity
 visited[i] = false; // Mark all vertices as unvisited
 }
 dist[src] = 0; // Distance to the source is 0
 // Process each vertex
 for ( i = 0; i < V - 1; i++) {
 int u = findMinVertex(dist, visited, V); // Find the closest unvisited vertex
 if (u == -1) break; // If no vertex is reachable, exit the loop
 visited[u] = true; // Mark it as visited
 // Relax all edges originating from u
```

```
 for ( j = 0; j < E; j++) {
 if (edges[j].src == u) {
            int v = edges[j].dest;
 int weight = edges[j].weight;
 if (!visited[v] && dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
 dist[v] = dist[u] + weight;
 }
 }
 }
 }
 // Print the distances
 printDistances(dist, V);
}
int main() {
 int V, E, src,i;
 printf("Enter the number of vertices: ");
 scanf("%d", &V);
 printf("Enter the number of edges: ");
 scanf("%d", &E);
 struct Edge edges[E];
 printf("Enter the edges in the format (source, destination, weight):\n");
 for ( i = 0; i < E; i++) {
 printf("Edge %d: ", i + 1);
 scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);
 }
 printf("Enter the source vertex: ");
 scanf("%d", &src);
  // Call Dijkstra's algorithm
 dijkstra(V, E, edges, src);
 return 0;
}
```

11)BFT Matrix

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

// Function to perform Breadth First Traversal (BFT)
void breadthFirstTraversal(int graph[MAX][MAX], int startNode, int n) {
   int queue[MAX], visited[MAX] = {0};
   int front = 0, rear = -1;

   // Enqueue the start node and mark it as visited
   visited[startNode] = 1;
   rear++;
   queue[rear] = startNode;

   printf("BFT starting from node %d: ", startNode);

   while (front <= rear) {
      // Dequeue a node and print it
      int currentNode = queue[front];
      front++;
      printf("%d ", currentNode);
      int i;

      // Explore all the adjacent nodes of the current node
      for ( i = 0; i < n; i++) {
         if (graph[currentNode][i] == 1 && !visited[i]) {
            visited[i] = 1;
            rear++;
            queue[rear] = i; // Enqueue the adjacent node
         }
```

```
        }
    }
    printf("\n");
}

int main() {
    int n, startNode,i,j;

    // Input number of nodes
    printf("Enter number of nodes: ");
    scanf("%d", &n);

    int graph[MAX][MAX];

    // Input adjacency matrix
    printf("Enter the adjacency matrix:\n");
    for ( i = 0; i < n; i++) {
        for ( j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    // Input the starting node for BFT
    printf("Enter the starting node: ");
    scanf("%d", &startNode);

    // Perform BFT starting from the given node
    breadthFirstTraversal(graph, startNode, n);

    return 0;
}
```

12)knapsack using branch and bound and backtracking

```
#include <stdio.h>

#define MAX_ITEMS 100

int maxProfit = 0;

void knapsack(int weights[], int values[], int n, int capacity, int index, int currentWeight, int currentProfit) {
   if (index == n) {
      if (currentProfit > maxProfit) {
         maxProfit = currentProfit;
      }
      return;
   }

   if (currentWeight + weights[index] <= capacity) {
      knapsack(weights, values, n, capacity, index + 1, currentWeight + weights[index], currentProfit +
values[index]);
   }

   knapsack(weights, values, n, capacity, index + 1, currentWeight, currentProfit);
}

int main() {
   int n, capacity,i;
   int weights[MAX_ITEMS], values[MAX_ITEMS];

   printf("Enter the number of items: ");
   scanf("%d", &n);

   printf("Enter the capacity of the knapsack: ");
   scanf("%d", &capacity);
```

```
    printf("Enter the weights and values of the items:\n");
    for ( i = 0; i < n; i++) {
        printf("Item %d - Weight: ", i + 1);
        scanf("%d", &weights[i]);
        printf("Item %d - Value: ", i + 1);
        scanf("%d", &values[i]);
    }

    knapsack(weights, values, n, capacity, 0, 0, 0);

    printf("Maximum profit: %d\n", maxProfit);

    return 0;
}
```

13)travelling sales person

```
#include <stdio.h>
#define INF 99999
#define MAX 10
int dp[1 << MAX][MAX], dist[MAX][MAX];
int min(int a, int b) {
    return (a < b) ? a : b;
}
int tsp(int n, int mask, int pos) {
    if (mask == (1 << n) - 1) {
        return dist[pos][0];
    }
    if (dp[mask][pos] != -1) {
        return dp[mask][pos];
    }
    int ans = INF;
    int city;
    for ( city = 0; city < n; city++) {
        if ((mask & (1 << city)) == 0) {
```

```
            int newAns = dist[pos][city] + tsp(n, mask | (1 << city), city);
            ans = min(ans, newAns);
        }
    }
    dp[mask][pos] = ans;
    return ans;
}
int main() {
    int n,i,j;
    printf("Enter number of cities: ");
    scanf("%d", &n);
    printf("Enter the distance matrix:\n");
    for ( i = 0; i < n; i++) {
        for ( j = 0; j < n; j++) {
            scanf("%d", &dist[i][j]);
        }
    }
    for ( i = 0; i < (1 << n); i++) {
        for (j = 0; j < n; j++) {
            dp[i][j] = -1;
        }
    }
    int result = tsp(n, 1, 0);
    printf("Minimum cost of traveling: %d\n", result);

    return 0;
}
```