1. **Construct an AVL tree for a given set of elements and implement insert and delete operations on the constructed tree.**

```
#include<stdio.h>
#include<stdlib.h>
typedef struct AVLNode{
        int key;
        struct AVLNode* left;
        struct AVLNode* right;
        int height;
}AVLNode;
AVLNode* leftRotate(AVLNode* x);
AVLNode* rightRotate(AVLNode* y);
int height(AVLNode* node);
AVLNode* createNode(int key);
int max(int a,int b);
int getBalanced(AVLNode* node);
AVLNode* insert(AVLNode* node,int key){
        if(node==NULL)
                return createNode(key);

        if(key<node->key)
                node->left=insert(node->left,key);
        else if(key>node->key)
                node->right=insert(node->right,key);
        else
                return node;
        node->height=1+max(height(node->left),height(node->right));
        int balance=getBalanced(node);
        if(balance>1&&key<node->left->key)
                return rightRotate(node);
        if(balance<-1&&key>node->right->key)
                return leftRotate(node);
        if(balance>1 && key>node->left->key){
                node->left=leftRotate(node->left);
                return rightRotate(node);

        }
        if(balance<-1 && key<node->right->key){
                node->right=rightRotate(node->right);
                return leftRotate(node);
        }
                return node;
        }
AVLNode* rightRotate(AVLNode* y){
        AVLNode* x=y->left;
        AVLNode* T2=x->right;
        x->right=y;
        y->left=T2;
        y->height=max(height(y->left),height(y->right))+1;
        x->height=max(height(x->left),height(x->right))+1;
                return x;
                }
AVLNode* leftRotate(AVLNode* x){
        AVLNode* y=x->right;
        AVLNode* T2=y->left;
        y->left=x;
        x->right=T2;
```

```
        x->height=max(height(x->left),height(x->right))+1;
        y->height=max(height(y->left),height(y->right))+1;
        return y;
        }
AVLNode* createNode(int key){
AVLNode* node=(AVLNode*)malloc(sizeof(AVLNode));
node->key=key;
node->left=NULL;
node->right=NULL;
node->height=1;
return node;
}
int getBalanced(AVLNode* node){
if(node==NULL)
        return 0;
return height(node->left)-height(node->right);
}
int height(AVLNode* node){
if(node==NULL)
        return 0;
return node->height;
}
int max(int a, int b){
        return (a>b) ? a:b;
}
AVLNode* minValueNode(AVLNode* node){
        AVLNode* current=node;
        while(current->left!=NULL)
                current=current->left;
        return current;
}
AVLNode* deleteNode(AVLNode* root,int key){
        if(root==NULL)
                return root;
        if(key<root->key)
                root->left=deleteNode(root->left,key);
        else if(key>root->key)
                root->right=deleteNode(root->right,key);
        else{
                if((root->left==NULL)||(root->right==NULL)){
                                AVLNode* temp=root->left? root-
>left:root->right;
                                if(temp==NULL){
                                temp=root;
                                root=NULL;
                                }else
                                        *root=*temp;
                                free(temp);
                                }else{
                                AVLNode* temp=minValueNode(root-
>right);
                                root->key=temp->key;
                                root->right=deleteNode(root-
>right,temp->key);
                                }
                                }
        if(root==NULL)
        return root;
```

```
        root->height=1+max(height(root->left),height(root->right));
        int balance=getBalanced(root);
        if(balance>1&&getBalanced(root->left)>=0)
        return rightRotate(root);
        if(balance>1&&getBalanced(root->left)<0){
                root->left=leftRotate(root);
                return rightRotate(root);
        }
        if(balance<-1&&getBalanced(root->right)<=0)
                return leftRotate(root);
        if(balance<-1&&getBalanced(root->right)>0){
                root->right=rightRotate(root->right);
                return leftRotate(root);
        }
        return root;
}
void inOrder(AVLNode* root){
        if(root!=NULL){
                inOrder(root->left);
                printf("%d ",root->key);
                inOrder(root->right);
        }
}
int main(){
        AVLNode* root=NULL;
        int elements[]={30,20,40,10,25,35,50};
        int size=sizeof(elements)/sizeof(elements[0]);
        for(int i=0;i<size;i++){
                root=insert(root,elements[i]);
        }
        printf("In-order traversal of the constructed AVL Tree:\n");
        inOrder(root);
        printf("\n");
        int keytodelete;
        printf("Enter the key to delete: ");
        scanf("%d",&keytodelete);
        root=deleteNode(root,keytodelete);
        printf("In-Order traversal after Deleting %d\n",keytodelete);
        inOrder(root);
        printf("\n");
        return 0;
}
```

Output:

In-order traversal of the constructed AVL Tree:

10 20 25 30 35 40 50

Enter the key to delete: 30

In-Order traversal after Deleting 30

10 20 25 35 40 50

8.Implement Job Sequencing with deadlines using Greedy strategy.

```
#include <stdio.h>
#include <stdlib.h>
struct Job {
    int deadline;
    int profit;
};
int compareJobs(const void *a, const void *b) {
    return ((struct Job *)b)->profit - ((struct Job *)a)->profit;
}
void jobSequencingWithDeadlines(struct Job jobs[], int n) {
    qsort(jobs, n, sizeof(struct Job), compareJobs);
    int maxDeadline = 0;
    for (int i = 0; i < n; ++i) {
        if (jobs[i].deadline > maxDeadline) {
            maxDeadline = jobs[i].deadline;
        }
    }
    int slots[maxDeadline];
    for (int i = 0; i < maxDeadline; ++i) {
        slots[i] = -1; // -1 indicates slot is empty
    }
    int totalProfit = 0;
    int jobCount = 0;
    for (int i = 0; i < n; ++i) {
        int deadline = jobs[i].deadline;
        while (deadline > 0 && slots[deadline - 1] != -1) {
            deadline--;
        }
        if (deadline > 0) {
            slots[deadline - 1] = i; // Assign job index to slot
            totalProfit += jobs[i].profit;
            jobCount++;
        }
    }
    printf("Maximum profit: %d\n", totalProfit);
    printf("Jobs scheduled:");
    for (int i = 0; i < maxDeadline; ++i) {
        if (slots[i] != -1) {
            printf(" %d", slots[i] + 1); // +1 to convert to 1-based
index
        }
    }
    printf("\n");
}

// Main function
int main() {
    // Example jobs array
    struct Job jobs[] = { {4, 70}, {2, 60}, {4, 50}, {3, 40}, {1,
30}, {4, 20} };
    int n = sizeof(jobs) / sizeof(jobs[0]);
    jobSequencingWithDeadlines(jobs, n);
    return 0;
}  OUTPUT:
Maximum profit: 220
Jobs scheduled: 4 2 3 1
```

8.Write a program to solve 0/1 Knapsack problem Using Dynamic Programming

```
#include <stdio.h>

int max(int a, int b) {
    return (a > b) ? a : b;
}

int knapsack(int capacity, int weights[], int values[], int n) {
    int dp[n + 1][capacity + 1];
    for (int i = 0; i <= n; ++i) {
        for (int w = 0; w <= capacity; ++w) {
            if (i == 0 || w == 0) {
                dp[i][w] = 0;
            } else if (weights[i - 1] <= w) {
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i
- 1]], dp[i - 1][w]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    return dp[n][capacity];
}

int main() {
    int values[] = {60, 100, 120};
    int weights[] = {10, 20, 30};
    int capacity = 50;
    int n = sizeof(values) / sizeof(values[0]);

    int maxProfit = knapsack(capacity, weights, values, n);

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

    return 0;
}
OUTPUT:
Maximum profit: 220
```

10.Implement N-Queens Problem Using Backtracking.

```
#include<stdio.h>
#include<stdlib.h>
int t[8] = {-1};
int sol = 1;
void printsol()
{
        int i,j;
        char crossboard[8][8];
        for(i=0;i<8;i++)
        {
                for(j=0;j<8;j++)
                {
                        crossboard[i][j]='_';
                }
        }
        for(i=0;i<8;i++)
        {
                crossboard[i][t[i]]='q';
        }
        for(i=0;i<8;i++)
        {
                for(j=0;j<8;j++)
                {
                        printf("%c ",crossboard[i][j]);
                }
                printf("\n");
        }
}
int empty(int i)
{
        int j=0;
        while((t[i]!=t[j])&&(abs(t[i]-t[j])!=(i-j))&&j<8)
                j++;
        return i==j?1:0;
}
void queens(int i)
{
        for(t[i] = 0;t[i]<8;t[i]++)
        {
                if(empty(i))
                {
                        if(i==7)
                        {
                                printsol();
                                printf("\n solution %d\n",sol++);
                        }
                        else
                                queens(i+1);
                }
        }
}
int main()
{
    queens(0);
    return 0;
}   OUTPUT:
```

```
q _ _ _ _ _ _ _
_ _ _ _ q _ _ _
_ _ _ _ _ _ _ q
_ _ _ _ _ q _ _
_ _ q _ _ _ _ _
_ _ _ _ _ _ q _
_ q _ _ _ _ _ _
_ _ _ q _ _ _ _

 solution 1
q _ _ _ _ _ _ _
_ _ _ _ _ q _ _
_ _ _ _ _ _ _ q
_ _ q _ _ _ _ _
_ _ _ _ _ _ q _
_ _ _ q _ _ _ _
_ q _ _ _ _ _ _
_ _ _ _ q _ _ _

 solution 2
q _ _ _ _ _ _ _
_ _ _ _ _ _ q _
_ _ _ q _ _ _ _
_ _ _ _ _ q _ _
_ _ _ _ _ _ _ q
_ q _ _ _ _ _ _
_ _ _ _ q _ _ _
_ _ q _ _ _ _ _
```

12.Use Backtracking strategy to solve 0/1 Knapsack problem.

```
#include <stdio.h>
int max(int a, int b) {
    return (a > b)? a: b;
}

void knapsack(int capacity, int weights[], int values[], int n, int
currentWeight, int currentValue, int *maxValue) {
    if (n == 0 || capacity == 0) {
        *maxValue = max(*maxValue, currentValue);
        return;
    }
    if (weights[n-1]> capacity) {
    knapsack(capacity, weights, values, n - 1, currentWeight,
currentValue, maxValue);
    } else {
    knapsack(capacity - weights[n-1], weights, values, n-1,
currentWeight + weights[n-1], currentValue +
values[n-1], maxValue);
    knapsack(capacity, weights, values, n - 1, currentWeight,
currentValue, maxValue);
    }
}
int main() {
    // Example data
    int values[] = {60, 100, 120};
    int weights[] = {10, 20, 30};
    int capacity = 50;
    int n = sizeof(values) / sizeof(values[0]);
    // Variable to store maximum profit
    int maxProfit = 0;
    // Solve the 0/1 Knapsack problem using Backtracking
    knapsack(capacity, weights, values, n, 0, 0, &maxProfit);
    // Output the maximum profit
    printf("Maximum profit: %d\n", maxProfit);
    return 0;
}
```

Output:

Maximum profit: 220

6.Implement Quick sort

```
#include <stdio.h>

void quickSort(int arr[], int low, int high);

int partition(int arr[], int low, int high);

void swap(int *a, int *b);

void displayArray(int arr[], int size);

int main() {

   int arr[100], n, i;

   printf("Enter the number of elements: ");

   scanf("%d", &n);

   printf("Enter the elements:\n");

   for (i = 0; i < n; i++) {

      scanf("%d", &arr[i]);

   }

   quickSort(arr, 0, n - 1);

   printf("Sorted array:\n");

   displayArray(arr, n);

   return 0;

}

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 partition(int arr[], int low, int high) {

   int pivot = arr[high];

   int i = (low - 1);

   for (int j = low; j < high; j++) {

      if (arr[j] <= pivot) {
```

```
      i++;
      swap(&arr[i], &arr[j]);
    }
  }
  swap(&arr[i + 1], &arr[high]);
  return (i + 1);
}
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
void displayArray(int arr[], int size) {
  for (int i = 0; i < size; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
}
```

OUTPUT:

```
Enter the number of elements: 5
Enter the elements:
5
8
98
4
2
Sorted array:
2 4 5 8 98
```

7.Implement Merge sort

```
#include <stdio.h>

void mergeSort(int arr[], int left, int right);

void merge(int arr[], int left, int mid, int right);

void displayArray(int arr[], int size);

int main() {

  int arr[100], n, i;

  printf("Enter the number of elements: ");

  scanf("%d", &n);

  printf("Enter the elements:\n");

  for (i = 0; i < n; i++) {

    scanf("%d", &arr[i]);

  }

  mergeSort(arr, 0, n - 1);

  printf("Sorted array:\n");

  displayArray(arr, n);

  return 0;

}

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);

  }

}

void merge(int arr[], int left, int mid, int right) {

  int n1 = mid - left + 1;

  int n2 = right - mid;

  int L[n1], R[n2];

  for (int i = 0; i < n1; i++) {
```

```
        L[i] = arr[left + i];
    }
    for (int i = 0; i < n2; i++) {
        R[i] = arr[mid + 1 + i];
    }
    int i = 0, j = 0, k = left;
    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 displayArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
```

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

OUTPUT:

Enter the number of elements: 5

Enter the elements:

85

96

67

68

56

Sorted array:

56 67 68 85 96

2. **Construct Min Heap using arrays, delete any element and display the content of the Heap.**

```
#include<stdio.h>

#include<stdlib.h>

#define MIN_HEAP_SIZE 100

void swap(int* a,int* b){

    int temp=*a;

    *a=*b;

    *b=temp;

}

void minHeapify(int heap[],int n,int i){

    int smallest=i;

    int left=2*i+1;

    int right=2*i+2;

    if(left,n&&heap[left]<heap[smallest])

        smallest=left;

    if(right<n&&heap[right]<heap[smallest])

        smallest=right;

    if(smallest!=i){

        swap(&heap[i],&heap[smallest]);

        minHeapify(heap,n,smallest);

    }

}

void buildMinHeap(int heap[],int n){

    int startIdx=(n/2)-1;

    for(int i=startIdx;i>=0;i--){

        minHeapify(heap,n,i);

    }

}

void deleteFromMinHeap(int heap[],int* n,int key){

    int i;
```

```
    for(i=0;i<*n;i++){
        if(heap[i]==key){
            break;
        } }
    if(i==*n){
        printf("Element %d not found in the Min Heap.\n");
        return;
    }
    heap[i]=heap[*n-1];
    *n=*n-1;
    buildMinHeap(heap,*n);
}
void displayMinHeap(int heap[],int n){
    printf("Min Heap elements: ");
    for(int i=0;i<n;i++){
        printf("%d ",heap[i]);
    }
    printf("\n");
}
int main(){
    int minHeap[MIN_HEAP_SIZE]={12,7,1,3,10,17,19};//Example MIX-Heap
    int n_minHeap=7;//Number of elements in Min-heap
    int key;
    displayMinHeap(minHeap,n_minHeap);
    printf("Enter element to delete from Min Heap: ");
    scanf("%d",&key);
    deleteFromMinHeap(minHeap,&n_minHeap,key);
    displayMinHeap(minHeap,n_minHeap);
    return 0;
}
```

OUTPUT: Min Heap elements: 12 7 1 3 10 17 19 Enter element to delete from Min Heap: 7
Min Heap elements: 1 3 12 0 10 17

3. **Construct Max Heap using arrays, delete any element and display the content of the Heap.**

```
#include<stdio.h>
#include<stdlib.h>
#define MAX_HEAP_SIZE 100
void swap(int* a,int* b){
    int temp=*a;
    *a=*b;
    *b=temp;
}
void maxHeapify(int heap[],int n,int i){
    int largest=i;
    int left=2*i+1;
    int right=2*i+2;
    if(left<n&&heap[left]>heap[largest])
        largest=left;
    if(right<n&&heap[right]>heap[largest])
        largest=right;
    if(largest!=i){
        swap(&heap[i],&heap[largest]);
        maxHeapify(heap,n,largest);
    }
}
void buildMaxHeap(int heap[],int n){
    int startIdx=(n/2)-1;
    for(int i=startIdx;i>=0;i--){
        maxHeapify(heap,n,i);
    }
}
void deleteFromMaxHeap(int heap[],int* n,int key){
    int i;
```

```
    for(i=0;i<*n;i++){
        if(heap[i]==key)
            break;
    }
    if(i==*n){
        printf("Element %d is not found in the Max-Heap.\n");
        return;
    }
    heap[i]=heap[*n-1];
    *n=*n-1;
    buildMaxHeap(heap,*n);
}
void displayMaxHeap(int heap[],int n){
    printf("Max Heap elements: ");
    for(int i=0;i<n;i++){
        printf("%d ",heap[i]);
    }
    printf("\n");
}
int main(){
    int maxHeap[MAX_HEAP_SIZE]={19,17,12,3,10,1,7};
    int n_maxHeap=7;
    int key; displayMaxHeap(maxHeap,n_maxHeap);
    printf("Enter element to delete from Max Heap: ");
    scanf("%d",&key);
    deleteFromMaxHeap(maxHeap,&n_maxHeap,key);
    displayMaxHeap(maxHeap,n_maxHeap);
    return 0;
}
```

OUTPUT:

Max Heap elements: 19 17 12 3 10 1 7

Enter element to delete from Max Heap: 17   Max Heap elements: 19 10 12 3 7 1

4.Implement BFT and DFT for given graph, when graph is represented by Adjacency Matrix

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

#define MAX_VERTICES 5
int graphMatrix[MAX_VERTICES][MAX_VERTICES];
int numVertices = MAX_VERTICES;
void initGraphMatrix()
{
    for (int i = 0; i<numVertices; i++)
    {
        for (int j = 0; j <numVertices; j++)
        {
            graphMatrix[i][j] = 0;
        }
    }
}
void addEdgeMatrix(int u, int v)
{
    graphMatrix[u][v] = 1;
    graphMatrix[v][u] = 1; // For undirected graph
}

void BFTMatrix(int start)
{
    bool visited[MAX_VERTICES] = { false };
    int queue[MAX_VERTICES];
    int front = 0, rear = 0;

    visited[start] = true;
    queue[rear++] = start;

    printf("Breadth-First Traversal (BFT) starting from vertex %d: ",
start);
    while (front < rear)
    {
        int current = queue[front++];
        printf("%d ", current);

        for (int i = 0; i<numVertices; i++)
        {
            if (graphMatrix[current][i] && !visited[i])
            {
                visited[i] = true;
                queue[rear++] = i;
            }
        }
    }
    printf("\n");
}

void DFTMatrixRecursive(int vertex, bool visited[])
{
    visited[vertex] = true;
    printf("%d ", vertex);
```

```
    for (int i = 0; i<numVertices; i++)
    {
        if (graphMatrix[vertex][i] && !visited[i])
        {
            DFTMatrixRecursive(i, visited);
        }
    }
}

void DFTMatrix(int start)
{
    bool visited[MAX_VERTICES] = { false };
    printf("Depth-First Traversal (DFT) starting from vertex %d: ",
start);
    DFTMatrixRecursive(start, visited);
    printf("\n");
}

int main()
{
    initGraphMatrix();
    addEdgeMatrix(0, 1);
    addEdgeMatrix(0, 2);
    addEdgeMatrix(1, 2);
    addEdgeMatrix(1, 3);
    addEdgeMatrix(2, 4);
    addEdgeMatrix(3, 4);
    printf("Adjacency Matrix:\n");
    for (int i = 0; i<numVertices; i++)
    {
        for (int j = 0; j <numVertices; j++)
        {
            printf("%d ", graphMatrix[i][j]);
        }
        printf("\n");
    }

    int startVertex = 0;
    BFTMatrix(startVertex);
    DFTMatrix(startVertex);

    return 0;
}   OUTPUT:
```

Adjacency Matrix:

0 1 1 0 0

1 0 1 1 0

1 1 0 0 1

0 1 0 0 1

0 0 1 1 0

Breadth-First Traversal (BFT) starting from vertex 0: 0 1 2 3 4

Depth-First Traversal (DFT) starting from vertex 0: 0 1 2 4 3

5.Implement BFT and DFT for given graph, when graph is represented by Adjacency Lists

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

   int vertex;

   struct Node* next;

};

struct Graph {

   int numVertices;

   struct Node** adjLists;

   int* visited;

};

struct Node* createNode(int v);

struct Graph* createGraph(int vertices);

void addEdge(struct Graph* graph, int src, int dest);

void BFT(struct Graph* graph, int vertex, int level);

void BFTUtil(struct Graph* graph, int vertex, int level);

void DFS(struct Graph* graph, int vertex);

int main() {

   struct Graph* graph = createGraph(4);

   addEdge(graph, 0, 1);

   addEdge(graph, 0, 2);

   addEdge(graph, 1, 2);

   addEdge(graph, 2, 0);

   addEdge(graph, 2, 3);

   addEdge(graph, 3, 3);

   printf("Breadth First Traversal starting from vertex 2:\n");

   BFT(graph, 2, 1);

printf("Depth First Traversal starting from vertex 2:\n");

   DFS(graph, 2);
```

```
    return 0;
}
struct Node* createNode(int v) {
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}
struct Graph* createGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = malloc(vertices * sizeof(struct Node*));
    graph->visited = malloc(vertices * sizeof(int));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
}
void BFT(struct Graph* graph, int vertex, int level) {
    graph->visited[vertex] = 1;
    printf("%d ", vertex);
    BFTUtil(graph, vertex, level);
    printf("\n");
}
```

```
void BFTUtil(struct Graph* graph, int vertex, int level) {
  if (level >= graph->numVertices) return;

  struct Node* adjList = graph->adjLists[vertex];
  struct Node* temp = adjList;

  while (temp != NULL) {
    int connectedVertex = temp->vertex;

    if (graph->visited[connectedVertex] == 0) {
      graph->visited[connectedVertex] = 1;
      printf("%d ", connectedVertex);
    }
    temp = temp->next;
  }

  temp = adjList;
  while (temp != NULL) {
    BFTUtil(graph, temp->vertex, level + 1);
    temp = temp->next;
  }
}
void DFS(struct Graph* graph, int vertex) {
  graph->visited[vertex] = 1;
  printf("%d ", vertex);

  struct Node* adjList = graph->adjLists[vertex];
  struct Node* temp = adjList;

  while (temp != NULL) {
```

```
        int connectedVertex = temp->vertex;


        if (graph->visited[connectedVertex] == 0) {
          DFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
} OUTPUT:
```

Breadth First Traversal starting from vertex 2:

2 3 0 1

Depth First Traversal starting from vertex 2:

2