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

```
#include <iostream>


using namespace std;

void heapifyMin(int arr[], int n, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] < arr[smallest])
        smallest = left;
    if (right < n && arr[right] < arr[smallest])
        smallest = right;

    if (smallest != i) {
        swap(arr[i], arr[smallest]);
        heapifyMin(arr, n, smallest);
    }
}

void buildMinHeap(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapifyMin(arr, n, i);
}

void deleteMin(int arr[], int &n, int key) {
    int index = -1;
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            index = i;
            break;
        }
    }

    if (index == -1) {
        cout << "Key not found!" << endl;
        return;
    }

    arr[index] = arr[n - 1];
    n--;
    buildMinHeap(arr, n);
}

int main() {
    int arr[] = {10, 20, 15, 30, 40};
    int n = sizeof(arr) / sizeof(arr[0]);

    buildMinHeap(arr, n);
    cout << "Min Heap:" << endl;
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    deleteMin(arr, n, 10);
    cout << "After deletion:" << endl;
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

9. Write a program to solve0/1Knapsack problem Using Dynamic Programming.

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

int knapSack(int W, int wt[], int val[], int n) {
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));

    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (wt[i - 1] <= w)
                dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }

    return dp[n][W];
}

int main() {
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);

    cout << "Maximum value in Knapsack: " << knapSack(W, wt, val, n) << endl;
    return 0;
}
```

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

Code (Adjacency Matrix):

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

#define V 5

void BFT(int graph[V][V], int start) {
    bool visited[V] = {false};
    queue<int> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";

        for (int i = 0; i < V; i++) {
            if (graph[node][i] == 1 && !visited[i]) {
                q.push(i);
                visited[i] = true;
            }
        }
    }
}

void DFT(int graph[V][V], int start) {
    bool visited[V] = {false};
    stack<int> s;
    s.push(start);
    visited[start] = true;

    while (!s.empty()) {
        int node = s.top();
        s.pop();
        cout << node << " ";

        for (int i = 0; i < V; i++) {
            if (graph[node][i] == 1 && !visited[i]) {
                s.push(i);
                visited[i] = true;
            }
        }
    }
}

int main() {
    int graph[V][V] = {
        {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}
    };

    cout << "Breadth-First Traversal:" << endl;
    BFT(graph, 0);

    cout << "\nDepth-First Traversal:" << endl;
    DFT(graph, 0);

    return 0;
}
```

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

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

int maxProfit = 0;

void knapSackBacktrack(int W, int wt[], int val[], int n, int idx, int currentWeight, int currentProfit) {
    if (idx == n) {
        if (currentProfit > maxProfit)
            maxProfit = currentProfit;
        return;
    }

    if (currentWeight + wt[idx] <= W)
        knapSackBacktrack(W, wt, val, n, idx + 1, currentWeight + wt[idx], currentProfit + val[idx]);

    knapSackBacktrack(W, wt, val, n, idx + 1, currentWeight, currentProfit);
}

int main() {
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);

    knapSackBacktrack(W, wt, val, n, 0, 0, 0);
    cout << "Maximum value in Knapsack: " << maxProfit << endl;
    return 0;}
```

Code (Adjacency List):

```
#include<iostream>
#include<list>
#include<queue>
#include<stack>
using namespace std;

class Graph {
    int V;
    list<int> *adj;

public:
    Graph(int V);
    void addEdge(int v, int w);
    void BFT(int s);
    void DFT(int s);
};

Graph::Graph(int V) {
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w) {
    adj[v].push_back(w);
    adj[w].push_back(v);
}

void Graph::BFT(int s) {
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    queue<int> queue;
    visited[s] = true;
    queue.push(s);

    while (!queue.empty()) {
        int node = queue.front();
        cout << node << " ";
        queue.pop();

        for (auto adjNode : adj[node]) {
            if (!visited[adjNode]) {
                visited[adjNode] = true;
                queue.push(adjNode);
            }
        }
    }
}

void Graph::DFT(int s) {
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    stack<int> stack;
    visited[s] = true;
    stack.push(s);

    while (!stack.empty()) {
        int node = stack.top();
        cout << node << " ";
        stack.pop();

        for (auto adjNode : adj[node]) {
            if (!visited[adjNode]) {
                visited[adjNode] = true;
                stack.push(adjNode);
            }
        }
    }
}

int main() {
    Graph g(5);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 4);

    cout << "Breadth-First Traversal:" << endl;
    g.BFT(0);

    cout << "\nDepth-First Traversal:" << endl;
    g.DFT(0);

    return 0;
}
```

10. Implement N-Queens Problem Using Backtracking

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

#define N 8

void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            cout << board[i][j] << " ";
        cout << endl;}}

bool isSafe(int board[N][N], int row, int col) {
    for (int i = 0; i < col; i++)
        if (board[row][i])
            return false;

    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    for (int i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;
    return true;}

bool solveNQUtil(int board[N][N], int col) {
    if (col >= N)
        return true;

    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;

            if (solveNQUtil(board, col + 1))
                return true;

            board[i][col] = 0; }}

    return false;}

bool solveNQ() {
    int board[N][N] = {0};

    if (!solveNQUtil(board, 0)) {
        cout << "Solution does not exist" << endl;
        return false;
    }

    printSolution(board);
    return true;
}

int main() {
    solveNQ();
    return 0;}
```

12. Implement Travelling Sales Person problem using Branch and Bound approach Objective:

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

#define N 4

int tsp(int graph[N][N], vector<bool>& visited, int pos, int count, int cost, int& ans) {
    if (count == N && graph[pos][0]) {
        ans = min(ans, cost + graph[pos][0]);
        return ans;
    }

    for (int i = 0; i < N; i++) {
        if (!visited[i] && graph[pos][i]) {
            visited[i] = true;
            tsp(graph, visited, i, count + 1, cost + graph[pos][i], ans);
            visited[i] = false;
        } }

    return ans;
}

int main() {
    int graph[N][N] = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    vector<bool> visited(N, false);
    visited[0] = true;
    int ans = INT_MAX;

    tsp(graph, visited, 0, 1, 0, ans);
    cout << "Minimum cost of TSP: " << ans << endl;
    return 0;}
```