Daffodil International University

Daffodil Smart City (DSC), Savar, Dhaka

Department of Computing and Information System

Semester Project on

Algorithms [CIS - 132]

Semester Fall-2024

Prepared By	Submitted To
Name: Abdullah ID: 241-16-008 Batch: 19 (A) Department of CIS	Name: Ms. Kazi Fatema Designation: Lecturer Department of CIS Daffodil International University

Acknowledgement

I express my heartfelt gratitude to the Almighty Allah (SWT) for guiding me throughout this project and providing me the strength to complete it.

I would like to extend my sincere appreciation to my teacher, Ms. Kazi Fatema mam for providing me with the opportunity to work on this project and for his valuable guidance and feedback that have helped me grow and learn.

I would also like to thank my parents for their unwavering support, encouragement, and motivation throughout my academic journey. Their love and belief in me have been my biggest source of inspiration.

I am grateful to Daffodil International University for providing me with an environment that has helped me develop my knowledge and skills.

Lastly, I would like to dedicate this project to those who are differently abled and have a thirst for knowledge. Their determination and resilience are a true inspiration to us all.

Contents

Theory Part:

Lab Part:

Theory Part

Task – 1

a. Design an algorithm for scenario-1, a ride-sharing system that assigns drivers to passengers in a way that reduces customer wait times and maximizes efficiency. The system should consider different approaches, including greedy for driver assignment, dynamic programming for fleet management, shortest path algorithms for navigation, and backtracking for managing cancellations or modifications.

Ans:

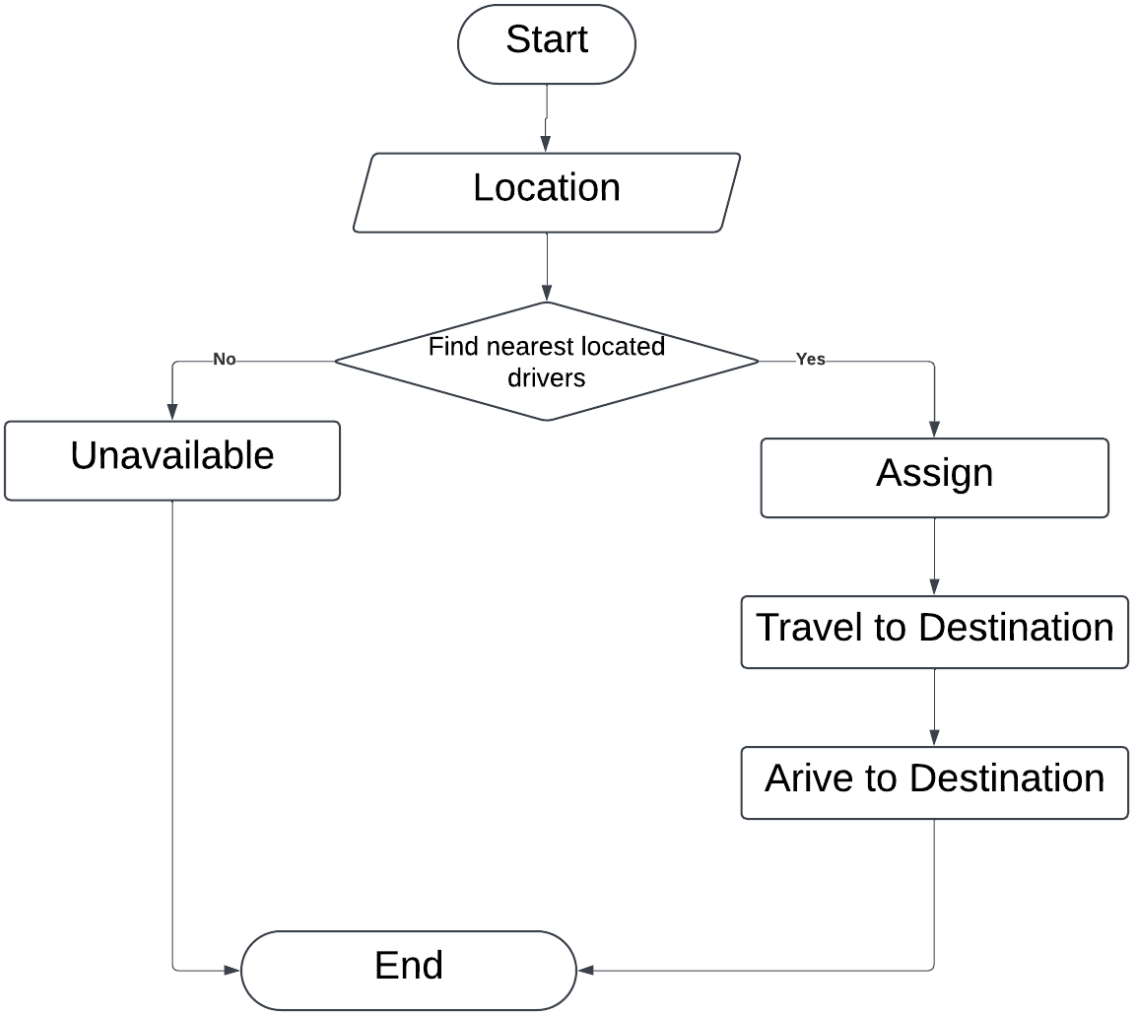
This algorithm integrates the **greedy approach**, **dynamic programming**, **shortest path algorithms**, and **backtracking** to create an efficient ride-sharing system.

Input

1. D : List of drivers with current locations.
2. R : List of active ride requests (pickup and drop-off locations).
3. Z : List of city zones with demand forecasts.
4. G(V, E) : Graph representing the city's road network (nodes V , edges E).
5. Real-time data for driver availability and traffic conditions.

Output

1. Driver assignments to ride requests.
2. Optimized driver routes and fleet distribution.



Steps

Step 1: System Initialization

1. Input:

• List of drivers and their current locations.

• List of ride requests (pickup and drop-off locations).

• Map data of the city.

2. Precompute shortest paths between key zones in the city.

Step 2: Assign Drivers to Passengers (Greedy Approach)

1. For each new ride request:

• Find the nearest available driver to the pickup location.

• Assign that driver to the ride request.

• Update the driver's status to "unavailable."

Step 3: Optimize Driver Distribution (Dynamic Programming)

1. Periodically (e.g., every 10 minutes):

• Check ride demand in each zone.

• Move idle drivers from low-demand zones to high-demand zones to prepare for future requests.

Step 4: Navigate Drivers to Destinations (Shortest Path Algorithm)

1. For each driver:

• Use a shortest path algorithm to find the quickest route:

• From the driver's current location to the pickup point.

• From the pickup point to the drop-off location.

2. Provide navigation instructions to the driver.

Step 5: Handle Cancellations or Modifications (Backtracking)

1. If a ride is canceled:

• Mark the assigned driver as "available."

• Check for other nearby ride requests and reassign the driver.

2. If a ride is modified:

• Recalculate driver assignment and update routes.

Step 6: Monitor and Update

1. Continuously monitor:

• Average wait times.

• Number of available drivers in each zone.

2. Adjust assignments and fleet distribution as needed.

b. How does dynamic programming help avoid overcrowding in certain zones, and what is the time complexity of this approach if the city is divided into n zones?

Ans:

Dynamic programming (DP) can effectively manage the distribution of drivers across zones to reduce overcrowding and minimize wait times. Here's how it works:

1. **Create zone:** The city is divided into n zones, and a DP state is defined to represent the number of drivers in each zone at a given time.
    - **Balancing Driver Distribution**:
        - Zones with higher demand get more drivers.
        - Zones with lower demand still receive enough drivers to avoid shortages.
2. **Transition Decisions**: The DP approach determines the optimal movement of drivers between zones based on:
    - Current driver availability in each zone.
    - Anticipated future demand (based on historical or real-time data).
    - Travel time or cost of moving drivers from one zone to another.
3. **Optimization Process**:

- DP calculates the **minimum "penalty" or "wait time"** by evaluating all possible distributions of drivers across zones.
- The penalty is typically proportional to the difference between demand and the number of assigned drivers for each zone.

The time complexity of the dynamic programming approach is:

O(T . k^n . n^2)

where: • n : Number of zones. • k : Maximum drivers per zone. • T : Number of time intervals. • n^2 : Accounts for transitions between zones.

Transitions Per State (m):

• m depends on the number of zones n because driver movements involve zone pairs.

• For simplicity, m can be considered O(n^2) (pairwise transitions between n zones).

c. What are the limitations of using Dijkstra's algorithm in scenario-1 if the city has dynamic traffic conditions (e.g., traffic jams or road closures)?

Ans:

Limitations of Using Dijkstra's Algorithm in Scenario-1 with Dynamic Traffic Conditions:

1. **Static Edge Weights**:

- **Issue**: Assumes fixed travel times, ignoring real-time changes like traffic or closures.
    - Dijkstra's algorithm finds the shortest path based on initial edge weights (distances)
    - Cannot dynamically adapt to real-time traffic changes like sudden road closures or traffic jams
    - Provides an optimal route at the moment of calculation, which may become suboptimal quickly
- **Why**: The algorithm uses static edge weights.
- **Impact**: Suboptimal routes during changing conditions.

2. **High Computational Cost for Updates**:

- **Issue**: Requires recomputation after every traffic change.
    - Recalculating routes for every traffic update is computationally expensive
    - Frequent rerouting can cause significant processing overhead
    - May lead to increased response times and reduced system performance

- **Why**: The algorithm doesn't update dynamically.
- **Impact**: Delays in route updates, increasing wait times.

3. **No Predictive Capabilities**:

- **Issue**: Doesn't account for future traffic events.
    - Cannot anticipate future traffic conditions
    - Provides a snapshot-based route without considering potential upcoming changes
    - Misses nuanced routing strategies that might avoid predictable congestion
- **Why**: The algorithm only uses current graph data.
- **Impact**: Routes may become inefficient mid-trip.

4. **Limited Real-Time Adaptability:**

- **Issue**: Dijkstra's algorithm struggles to handle temporary changes in road conditions, such as:
    - Temporary road blockages
    - Accident zones
    - Sudden congestion
    - Construction work
- **Why**: The algorithm assumes roads are always accessible.
- **Impact**: Drivers may be routed through closed roads.

5. **Potential Performance Bottlenecks:**

- **Issue**: Explores all possible paths, which is slow in dense cities.
    - In large, complex urban networks, Dijkstra's algorithm can become slow
    - Time complexity of $O(V^2)$ makes it less suitable for real-time, large-scale routing
- **Why**: The algorithm performs exhaustive searches.
- **Impact**: High computational cost, especially in large cities.

Alternative Recommendations:

- A* algorithm with dynamic heuristics
- Time-dependent routing algorithms
- Machine learning-based predictive routing models

d. When integrating these algorithms, what consequences would you face and how would you handle problems like Dhaka city's rush hour demand, fluctuating driver availability, and dynamic traffic patterns?

Ans:

Ride-Sharing System Optimization: Navigating Dhaka's rush hour Challenges

Key Integration Challenges:

1. Rush Hour Demand
    - Problem: Extreme driver shortage
    - Solution:
        - Predictive demand forecasting
        - Dynamic driver incentives
        - Pre-positioning drivers in high-demand zones
2. Driver Availability
    - Problem: Inconsistent service coverage
    - Solution:
        - Real-time availability tracking
        - Flexible incentive structures
        - Automated driver recruitment during peaks
3. Traffic Pattern Complexity
    - Problem: Inaccurate route estimations
    - Solution:
        - Real-time traffic data integration
        - Adaptive routing algorithms
        - Machine learning for predictive modeling
4. Algorithmic Approach
    - Layered strategy:
        - Greedy matching
        - Dynamic fleet optimization
        - Adaptive routing
        - Request modification handling
5. Technical Infrastructure
    - Implement:
        - Cloud-based system
        - Distributed computing

- Fault-tolerance mechanisms

Core Focus: Create an adaptive, efficient system balancing technological sophistication with practical urban transportation needs.

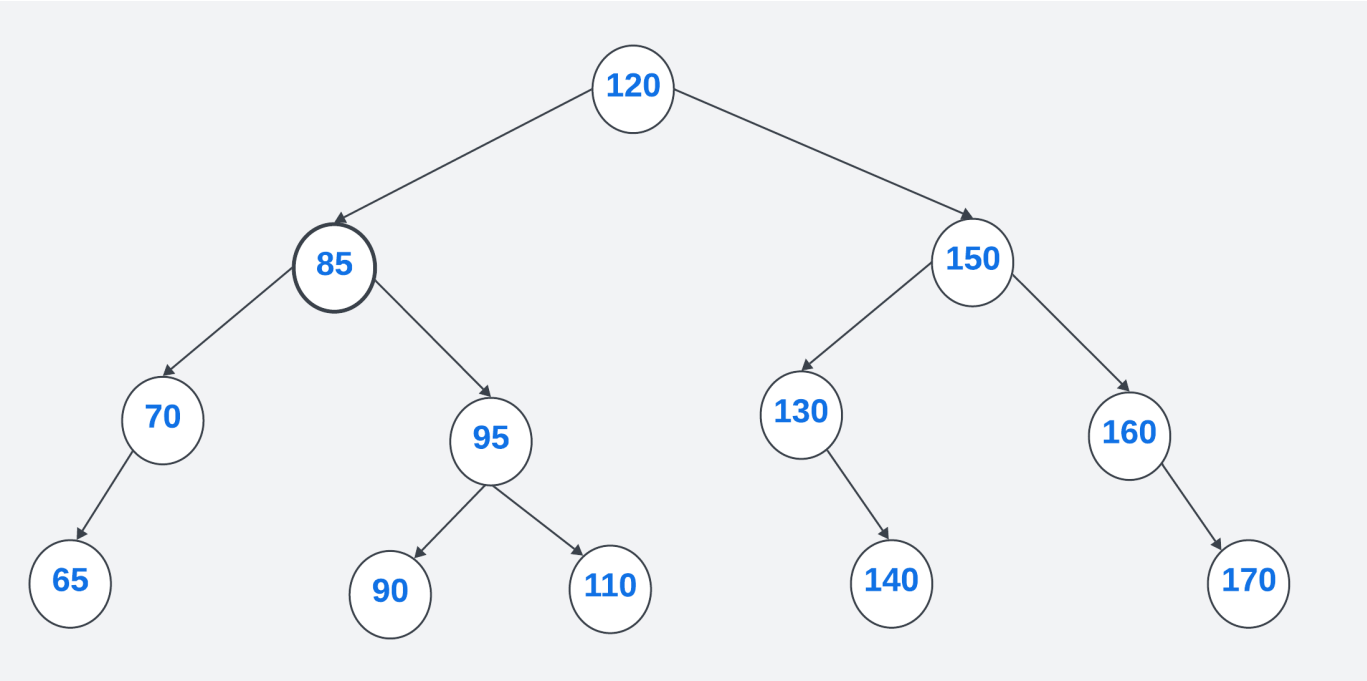
Task – 2

a. Perform an "**in-order traversal**" on the given BST and list the node values in the order they would be visited.

Ans:

In a **Binary Search Tree (BST)**, an **in-order traversal** visits nodes in ascending order of their values. The process for an in-order traversal is as follows:

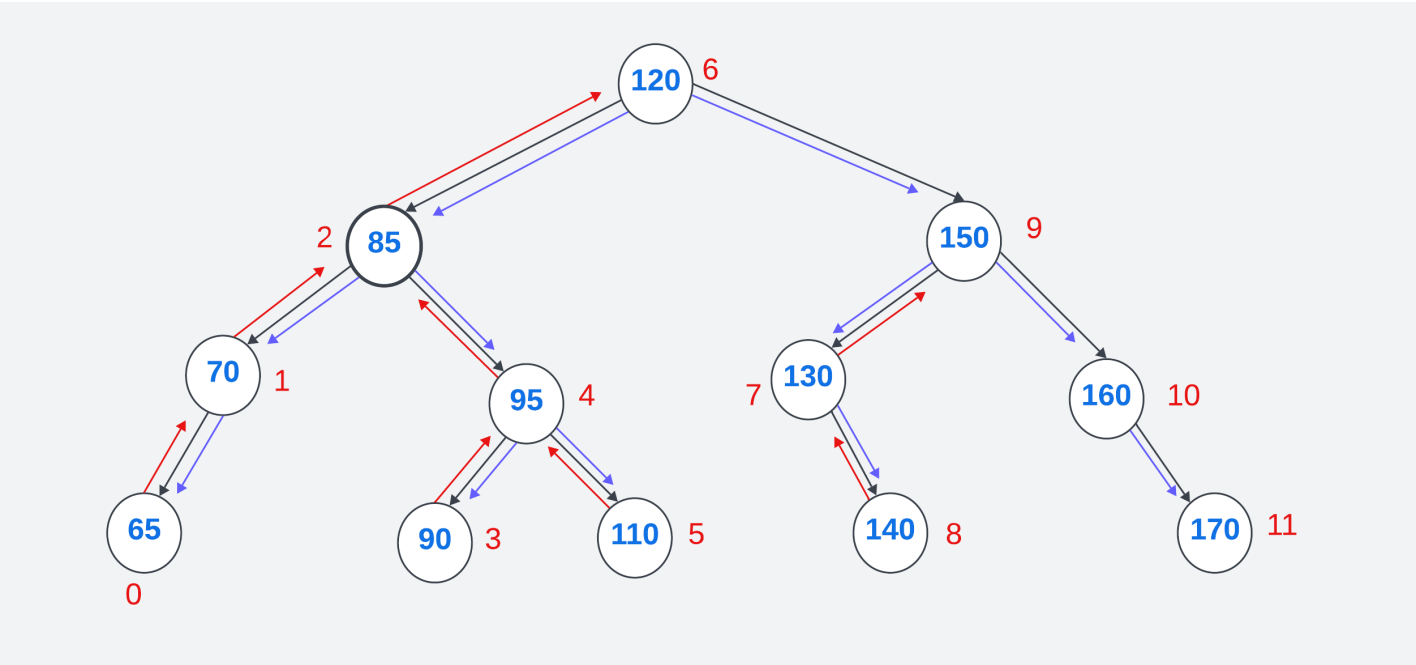
1. **Visit the left subtree** recursively.
2. **Visit the root node**.
3. **Visit the right subtree** recursively.



Traversal Explanation:

Starting from the leftmost node and moving up to the root and then down the right subtree, the sequence of visited nodes would be:

65, 70, 85, 90, 95, 110, 120, 120, 140, 150, 160, 170

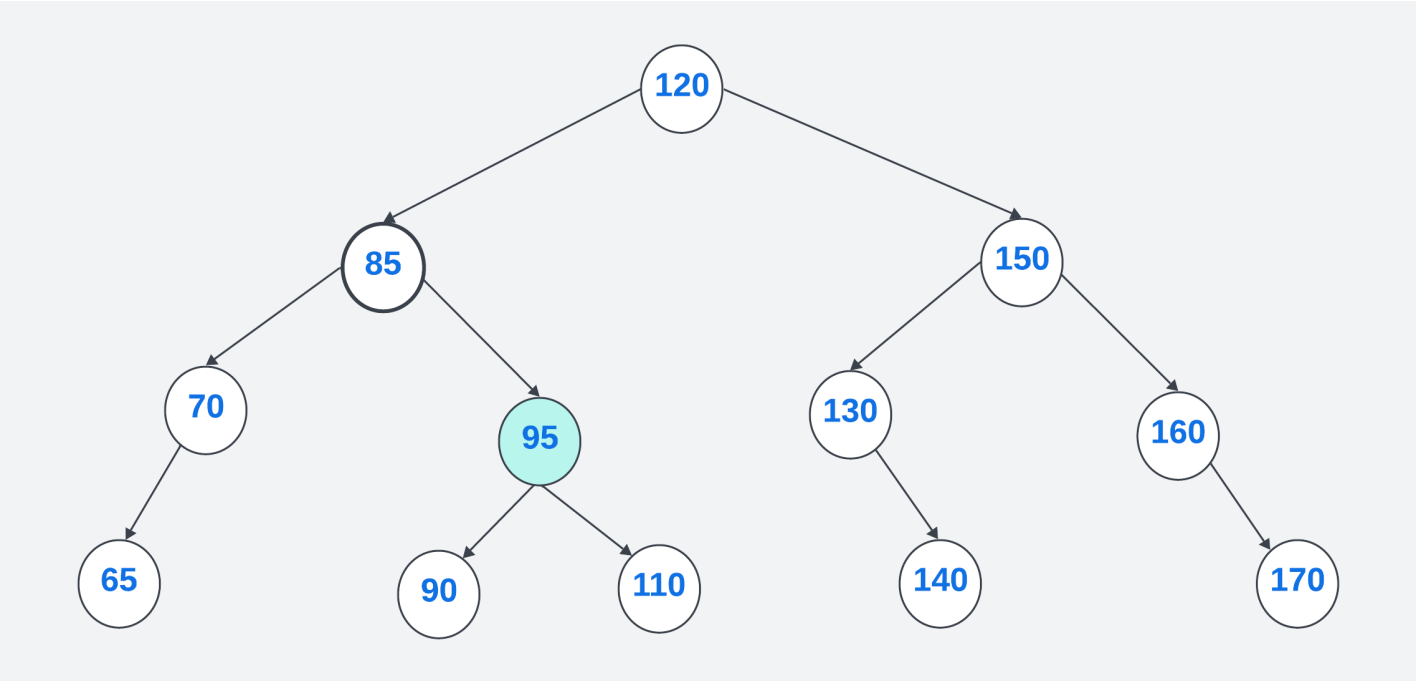


b. How will the BST modify itself if the node "**95**" is deleted? Display the tree's updated structure following this deletion.

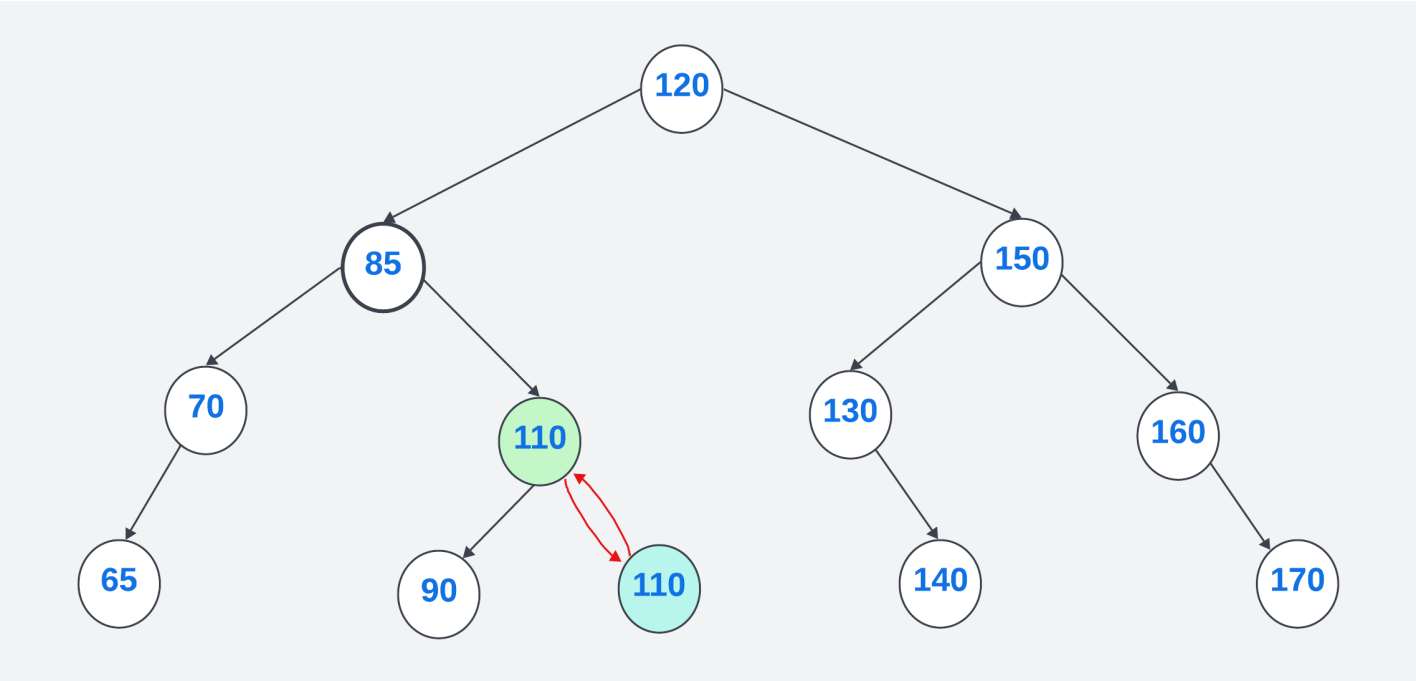
Ans:

Delete the node 95.

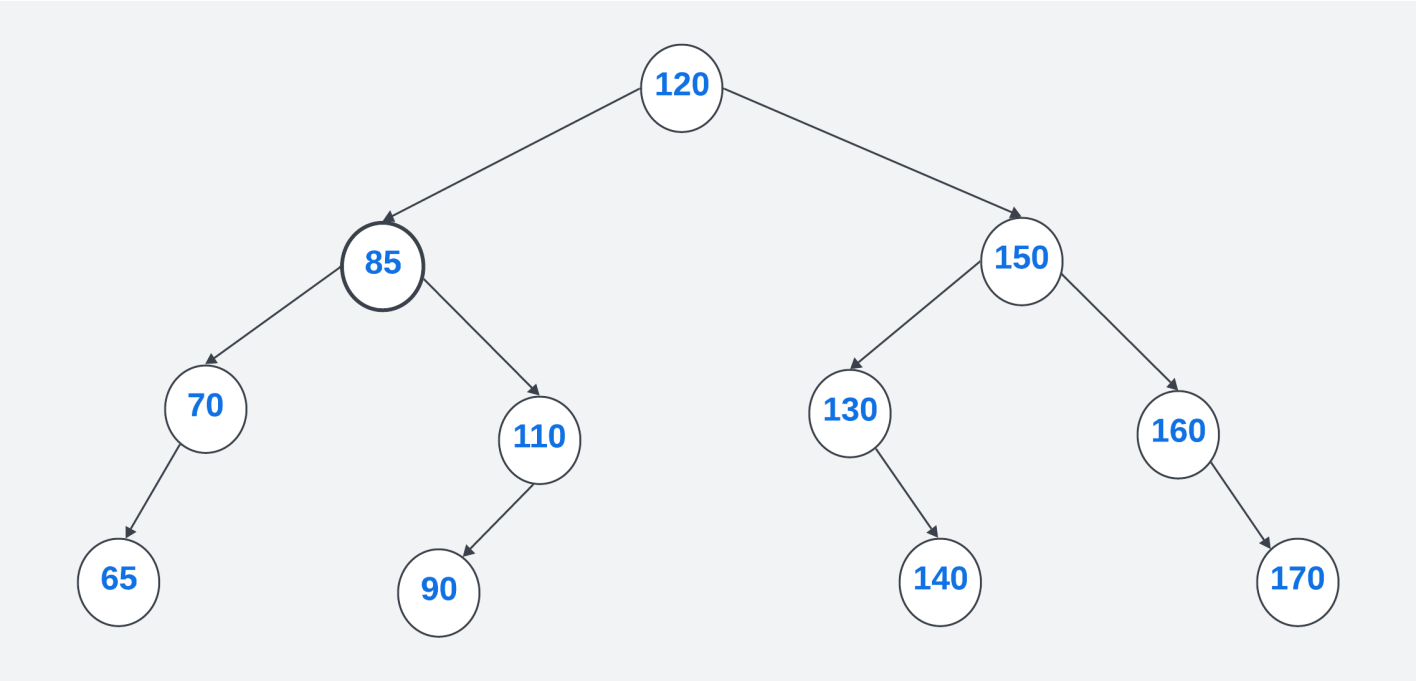
1. Locate the node 95.



2. Node 95 has two children (90 and 110).
3. Find the in-order successor of 90, which is the smallest node in 90's right subtree. Here, the in-order successor is 110.
4. Replace 95 with 110.



5. Then remove 110 from its original position.



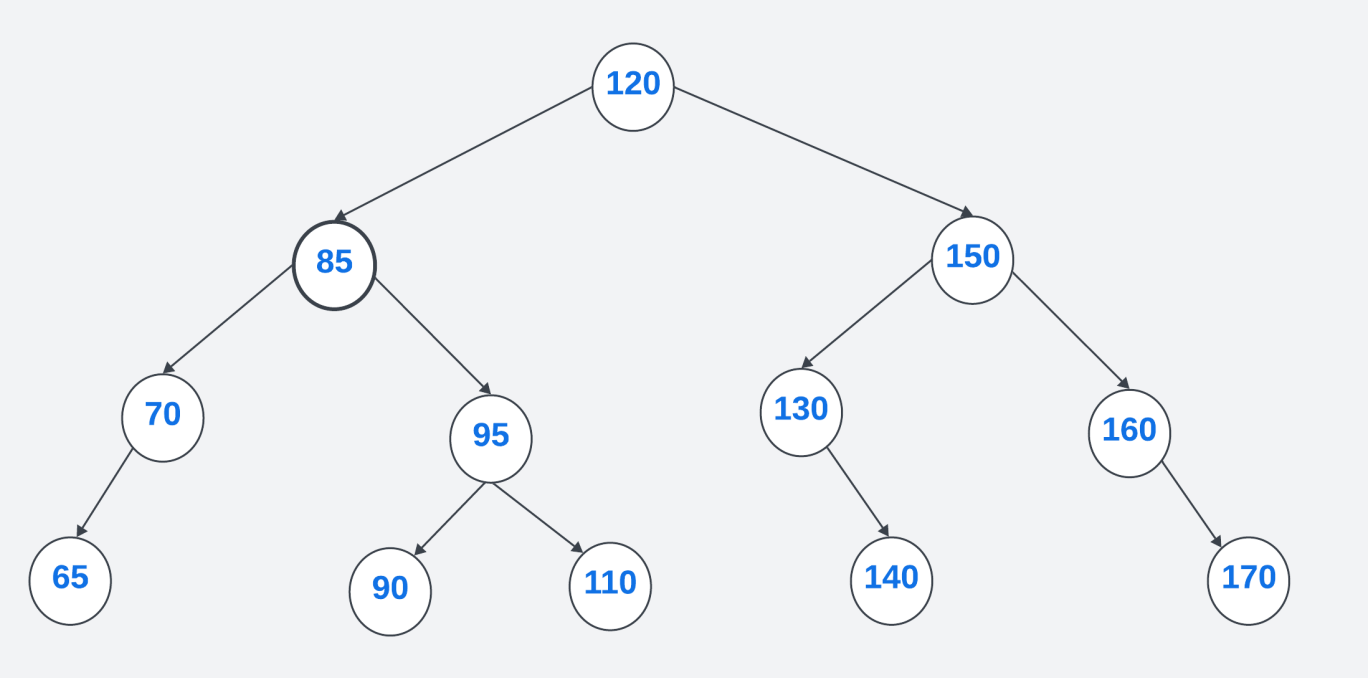
c. How would the traversal order change if you applied **"Breadth-First Search (BFS)"** instead of **"in-order traversal"** ?

Ans:

BFS:

Breadth-First Search (BFS), also known as **level-order traversal**, the nodes are visited level by level, from top to bottom and left to right within each level.

- Starts from the root node.
- Visits all nodes at the current level before moving to the next level.



Given the Binary Search Tree (BST) in the image, here's the **BFS traversal order**:

1. **Level 1**: 120
2. **Level 2**: 85, 150
3. **Level 3**: 70, 95, 130, 160
4. **Level 4**: 65, 90, 110, 140, 170

So, the **BFS traversal order** would be:

120, 85, 150, 70, 95, 130, 160, 65, 90, 110, 140, 170

In-order traversal:

1. **Visit the Left Subtree**: Recursively perform in-order traversal on the left child.
2. Visit the root node.
3. Traverse the right subtree in-order.

This would result in:

65, 70, 85, 90, 95, 110, 120, 130, 140, 150, 160, 170

So, **in-order traversal** results in a sorted list of values, **BFS traversal** explores the tree level-by-level, which produces a different order.

Lab Part

Task – 1

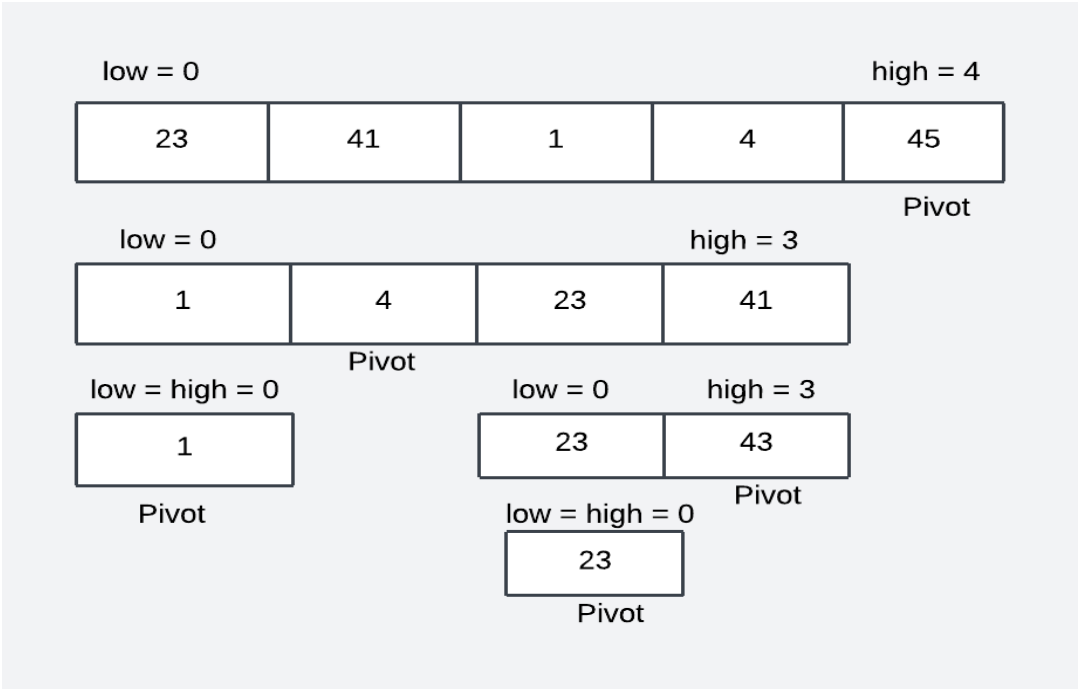
a. Implement the Quick Sort algorithm to sort the collection of elements. Output the order of elements after each complete pass through the collection to show how the sorting progresses.

Ans:

```
#include <bits/stdc++.h>

using namespace std;

int partition(int a[], int low, int high)
{
   int pivot = a[high];
   int i = low-1;

   for (int j = low; j < high; j++)
   {
       if (a[j] < pivot)
       {
           i++;
           swap(a[i], a[j]);
       }
   }
   swap(a[i+1], a[high]);
   return i+1;
}

void quick_sort(int arr[], int low, int high)
```

```
{
    if (low < high)
    {
        int mid = partition(arr, low, high);
        quick_sort(arr, low, mid - 1);
        quick_sort(arr, mid + 1, high);
    }
}

void print(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main()
{
    int n;
    cout << "Enter array size: ";
    cin >> n;

    int a[n];

    cout << "Input elements: ";
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    quick_sort(a, 0, n - 1);

    cout << "Sorted Array : ";
    print(a, n);
}
```

Iteration:



Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

abdullah@Abdullahs-MacBook-Air Vscode % g++ -std=c++20 "/Use
odeRunnerFile
Enter array size: 5
Input elements: 23 41 1 4 45
Sorted Array : 1 4 23 41 45
abdullah@Abdullahs-MacBook-Air Vscode %
```

b. Implement Prim's algorithm to determine the minimum spanning tree (MST) of the graph. Display the edges that are part of the MST, along with their corresponding weights.

Ans:

```
#include <bits/stdc++.h>
using namespace std;

int minKey(vector<int> &key, vector<bool> &mstSet, int n)
```

```
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < n; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

void printMST(vector<int> &parent, vector<vector<int>> &graph,
vector<char> &vertexData, int n)
{
    cout << "Vertices \tWeight\n";
    for (int i = 1; i < n; i++)
        cout << vertexData[parent[i]] << " - " << vertexData[i] <<
"\t\t" << graph[i][parent[i]] << " \n";
}

void primMST(vector<vector<int>> &graph, vector<char> &vertexData,
int n)
{
    vector<int> parent(n);
    vector<int> key(n);
    vector<bool> mstSet(n);

    for (int i = 0; i < n; i++)
    {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < n - 1; count++)
    {
        int u = minKey(key, mstSet, n);
        mstSet[u] = true;

        for (int v = 0; v < n; v++)
        {
            if (graph[u][v] && mstSet[v] == false && graph[u][v] <
key[v])
```

```
            {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    printMST(parent, graph, vertexData, n);
}

int main()
{
    int n;
    cout << "Enter the number of vertices: ";
    cin >> n;

    vector<char> vertexData(n);
    cout << "Enter the labels of the vertices: ";
    for (int i = 0; i < n; i++)
    {
        cin >> vertexData[i];
    }

    vector<vector<int>> adjmat(n, vector<int>(n));
    cout << "Enter the adjacency matrix: " << endl;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cin >> adjmat[i][j];
        }
    }

    primMST(adjmat, vertexData, n);

    return 0;
}
```

Output:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

abdullah@Abdullahs-MacBook-Air Vscode % g++ -std=c++20 "/Users/abdullah/Vs
Enter the number of vertices: 3
Enter the labels of the vertices: A B C
Enter the adjacency matrix:
0 5 0
5 0 1
0 1 0
Vertices        Weight
A - B           5
B - C           1
abdullah@Abdullahs-MacBook-Air Vscode %
```

c. Implement the scenario-1 using C/C++.

Ans:

```cpp
#include <bits/stdc++.h>

using namespace std;

// Location structure
struct Location
{
   int x, y;
};

// Function to calculate Manhattan distance
int calculateDistance(Location a, Location b)
{
   return abs(a.x - b.x) + abs(a.y - b.y);
}

// Greedy algorithm: Assign closest driver to the passenger
int assignClosestDriver(vector<Location> &drivers, Location
passenger)
{
   int closestDriver = -1;
   int minDistance = INT_MAX;
```

```
    for (int i = 0; i < drivers.size(); i++)
    {
        int dist = calculateDistance(drivers[i], passenger);
        if (dist < minDistance)
        {
            minDistance = dist;
            closestDriver = i;
        }
    }

    return closestDriver; // Return the index of the closest driver
}

// Shortest Path Algorithm using BFS for a grid-like city
int bfsShortestPath(Location start, Location end)
{
    if (start.x == end.x && start.y == end.y)
        return 0; // Already at destination

    queue<pair<Location, int>> q; // Queue stores location and steps
taken
    q.push({start, 0});

    set<pair<int, int>> visited; // To avoid revisiting locations
    visited.insert({start.x, start.y});

    // Directions for up, down, left, right
    vector<pair<int, int>> directions = {{0, 1}, {0, -1}, {1, 0}, {-
1, 0}};

    while (!q.empty())
    {
        auto [current, steps] = q.front();
        q.pop();

        for (auto dir : directions)
        {
            int nx = current.x + dir.first;
            int ny = current.y + dir.second;

            if (nx == end.x && ny == end.y)
                return steps + 1;
```

```
            if (visited.find({nx, ny}) == visited.end())
            {
                q.push({{nx, ny}, steps + 1});
                visited.insert({nx, ny});
            }
        }
    }

    return -1;
}

// Backtracking function
bool backtrack(vector<int> &assignedDrivers, int rideIndex,
vector<bool> &availableDrivers)
{
    if (rideIndex == assignedDrivers.size())
        return true; // All rides assigned successfully

    for (int driver = 0; driver < availableDrivers.size(); driver++)
    {
        if (availableDrivers[driver])
        {
            assignedDrivers[rideIndex] = driver;
            availableDrivers[driver] = false; // Mark driver as
unavailable

            if (backtrack(assignedDrivers, rideIndex + 1,
availableDrivers))
                return true;

            // Undo assignment
            assignedDrivers[rideIndex] = -1;
            availableDrivers[driver] = true;
        }
    }

    return false;
}

// Find the closest point
int findClosestPoint(Location passenger, vector<Location> &points)
{
    int closestIndex = -1;
    int minDistance = INT_MAX;
```

```
    for (int i = 0; i < points.size(); i++)
    {
        int dist = calculateDistance(passenger, points[i]);
        if (dist < minDistance)
        {
            minDistance = dist;
            closestIndex = i;
        }
    }

    return closestIndex;
}

int main()
{
    vector<Location> predefinedPoints = {
        {0, 0},  // Changa
        {5, 5},  // Gate 2
        {10, 10} // Khagan
    };

    vector<string> pointNames = {"Changa", "Gate 2", "Khagan"};

    vector<Location> drivers = {
        {1, 1}, {6, 5}, {4, 3}, {2, 8}, {9, 7}, {11, 12}, {0, 2},
{8, 8}, {3, 4}, {10, 0}};

    Location passenger;
    cout << "Enter passenger location (x y): ";
    cin >> passenger.x >> passenger.y;

    int pickupIndex = findClosestPoint(passenger, predefinedPoints);

    cout << "Closest pickup point: " << pointNames[pickupIndex] << "
at ("
         << predefinedPoints[pickupIndex].x << ", " <<
predefinedPoints[pickupIndex].y << ")\n";

    Location dropoff;
    cout << "Enter drop-off location (choose 0 for Changa, 1 for
Gate 2, 2 for Khagan): ";
    int dropIndex;
    cin >> dropIndex;
```

```
    dropoff = predefinedPoints[dropIndex];

    cout << "Drop-off point: " << pointNames[dropIndex] << " at ("
         << dropoff.x << ", " << dropoff.y << ")\n";

    int driver = assignClosestDriver(drivers, passenger);
    cout << "Assigned Driver: Driver " << driver + 1 << " at ("
         << drivers[driver].x << ", " << drivers[driver].y << ")\n";

    int distance = bfsShortestPath(predefinedPoints[pickupIndex],
dropoff);
    cout << "Shortest path from " << pointNames[pickupIndex] << " to
"
         << pointNames[dropIndex] << ": " << distance << " steps\n";

    return 0;
}
```

Output: