Lab-4

Graphs and trees

Discrete Mathematics

Andrew Safwat Fawzy – 21010314
Armia Joseph Hakim - 21010229

GitHub Repo

Click here

Part-1

Airline Network Shortest-Path Finder

Problem Statement

Imagine you are tasked with developing a tool to assist airline passengers in finding the most efficient route between two airports within the airline's network. Create a Java program that models an airline network as a graph, where airports are nodes and flights are edges. Your program should enable users to input details of the flight connections between airports and find the shortest path between a specified source and destination airport.

- Implement a class to represent the airline network graph. Each airport is a node, and flights between airports are edges. You can choose an adjacency matrix or adjacency list to represent the graph.
- Implement Dijkstra's algorithm to find the shortest path between two specified airports in the airline network.
- Display the optimal route details, including the sequence of airports to visit and the total distance or time required for the journey.
- Implement error handling mechanisms to handle cases where the specified source or destination airport is not in the network or when there is no direct flight between them

Data Structures used

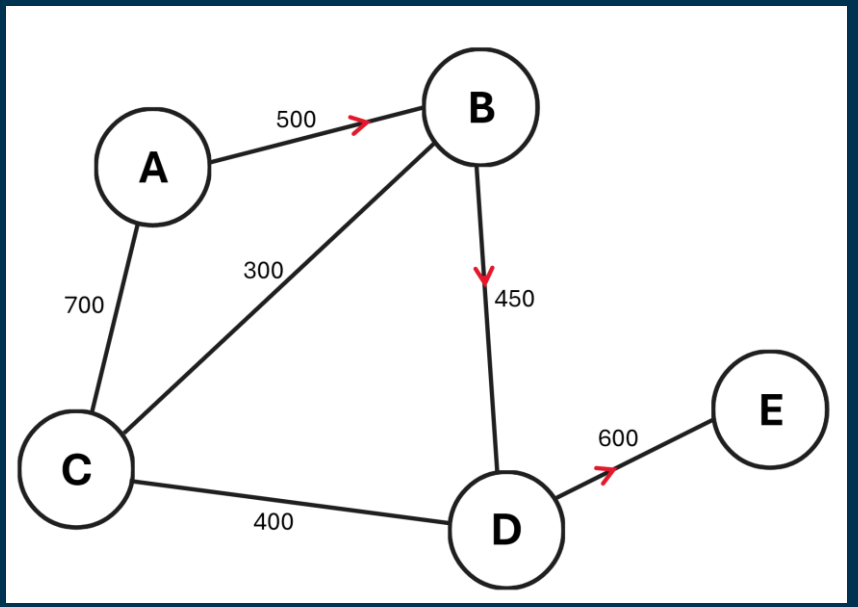
- 2d primitive int array – adjacency matrix
- x2 1d primitive int array – parent array, distance array.
- 1d primitive Boolean array – visited array.
- Array List – path array
- x2 Hashmap – intToChar, charToInt

Sample Runs

```
Enter list of airports: A, B, C, D, E
Enter the flights: A-B, A-C, B-C, C-D, B-D, D-E
The distance for each flight (in miles):
A to B: 500
A to C: 700
B to C: 300
C to D: 400
B to D: 450
D to E: 600
Enter source airport: A
Enter destination airport: E
Shortest path from A to E is: A B D E
Total distance: 1550 miles

Process finished with exit code 0
```

Result



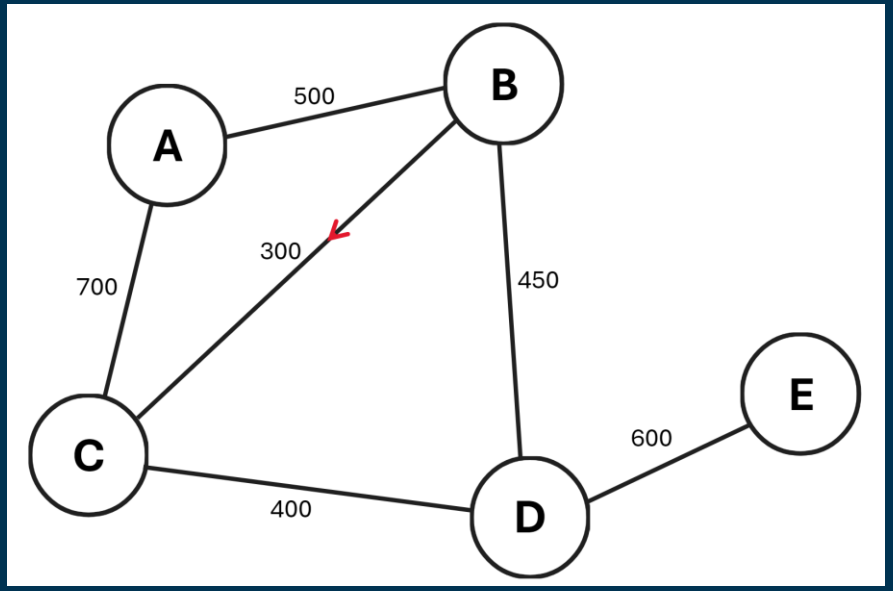
Visualization of result

Sample Runs

```
Enter list of airports: A, B, C, D, E
Enter the flights: A-B, A-C, B-C, C-D, B-D, D-E
The distance for each flight (in miles):
A to B: 500
A to C: 700
B to C: 300
C to D: 400
B to D: 450
D to E: 600
Enter source airport: B
Enter destination airport: C
Shortest path from B to C is: B C
Total distance: 300 miles

Process finished with exit code 0
```

Result

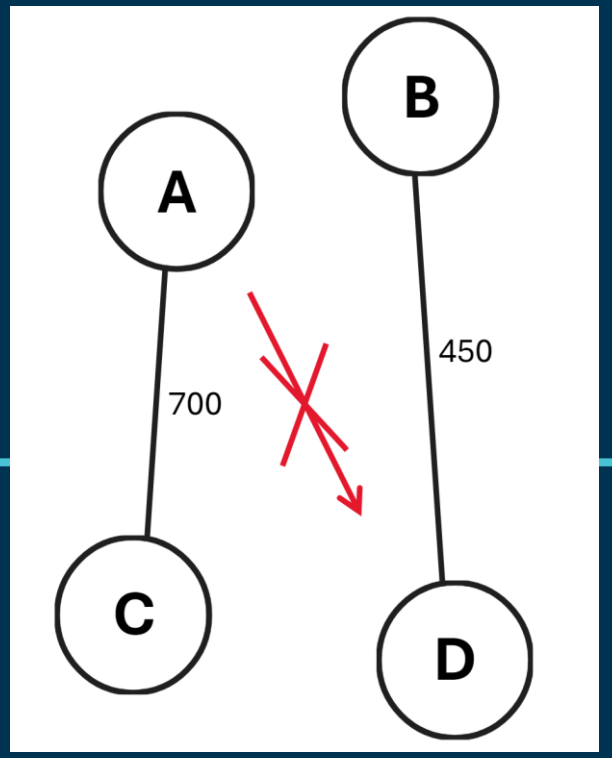


Visualization of result

Sample Runs

```
Enter list of airports: A, B, C, D
Enter the flights: A-B, C-D
The distance for each flight (in miles):
A to B: 500
C to D: 600
Enter source airport: A, D
Enter destination airport: There is no path from A to D
```

Result

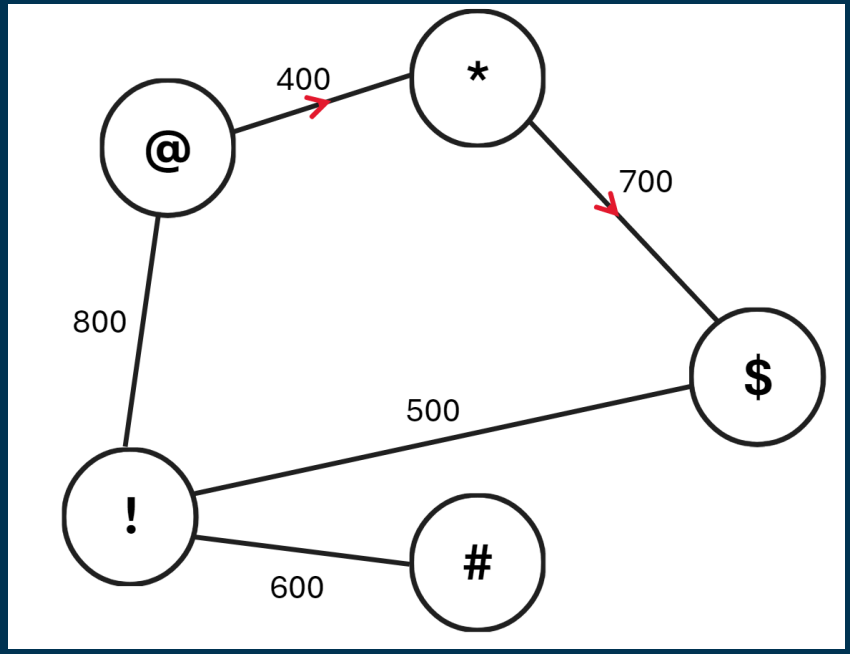


Visualization of result

Sample Runs

```
Enter list of airports: *, @, !, $, #
Enter the flights: *-@, !-$, #-!, *-$, @-!
The distance for each flight (in miles):
* to @: 400
! to $: 500
# to !: 600
* to $: 700
@ to !: 800
Enter source airport: @
Enter destination airport: $
Shortest path from @ to $ is: @ * $
Total distance: 1100 miles
```

Result



Visualization of result

Part-2

Class Schedule Optimization

Problem Statement

Imagine a school with multiple classes and subject timings where certain classes cannot occur simultaneously due to shared resources or teacher availability. Develop a Java program that generates an optimized class schedule by assigning time slots to classes, ensuring that no conflicting classes occur at the same time.

- Represent the schedule information as a graph where nodes represent classes, and edges denote conflicting timings between classes.
- Implement a graph coloring algorithm to assign distinct colors (time slots) to nodes (classes) in the graph. Ensure that adjacent nodes (classes) linked by edges (conflicting timings) do not share the same color (time slot) to avoid scheduling conflicts.
- Display the timetable with color-coded class timings, ensuring that conflicting classes have different colors (non-overlapping timings).
- Use any color names as you like.

Data Structures used

- Node class – Has color and value that can be assigned to it
- Hash map – To map every symbol that user enters to a number to make graph easier to deal with.
- Array of Strings – To store colors
- Array list of Nodes – As an adjacency list of vertices for graph representation

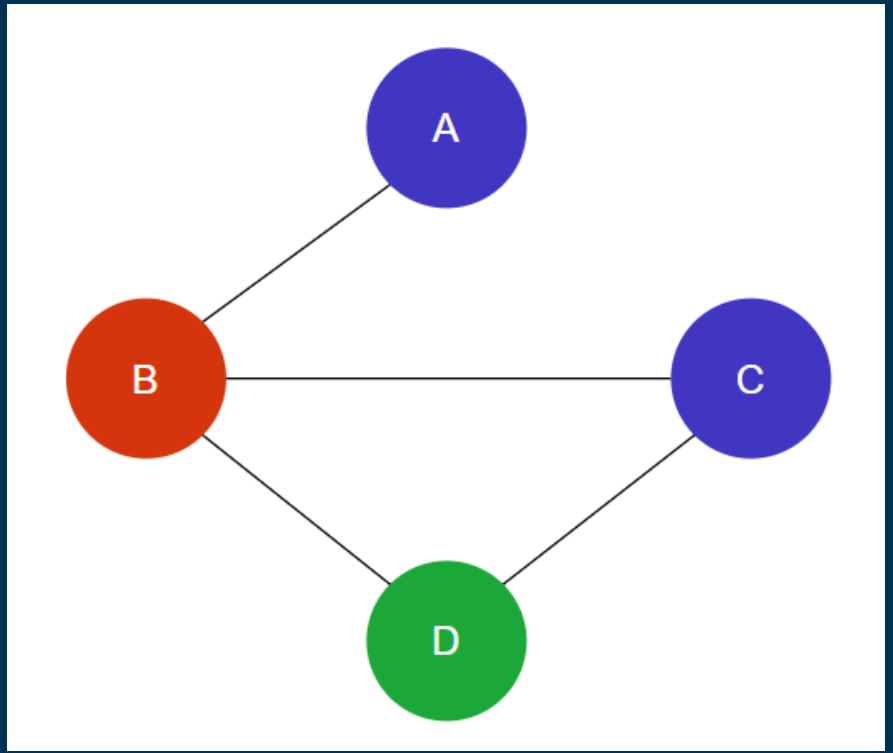
Sample Runs

```
Classes: A, B, C, D
Conflicting classes (cannot occur simultaneously) (put '-' between different nodes):
A-B
B-C
C-D
B-D

Optimized Class Schedule:
A - Blue
B - Red
C - Blue
D - Green

Process finished with exit code 0
```

Result



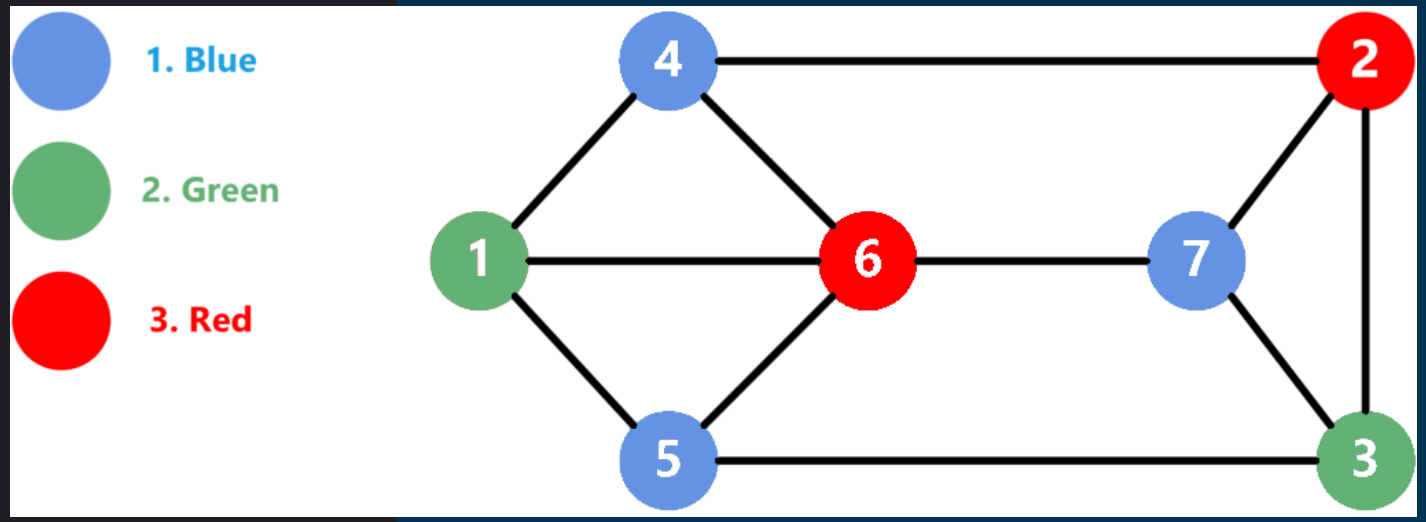
Visualization of result

Sample Runs Cont.

```
Classes: 1 2 3 4 5 6 7
Conflicting classes (cannot occur simultaneously) (put '-' between different nodes):
1-4
1-6
1-5
6-4
6-5
4-2
5-3
6-7
2-7
3-7
2-3

Optimized Class Schedule:
1 - Green
2 - Red
3 - Green
4 - Blue
5 - Blue
6 - Red
7 - Blue
```

Result



Visualization of result

Part-3

Tree Traversal

Problem Statement

Implement three algorithms for Binary Tree traversal recursively or iteratively:

- Preorder
- Inorder
- Postorder

We implemented it both recursively and iteratively !

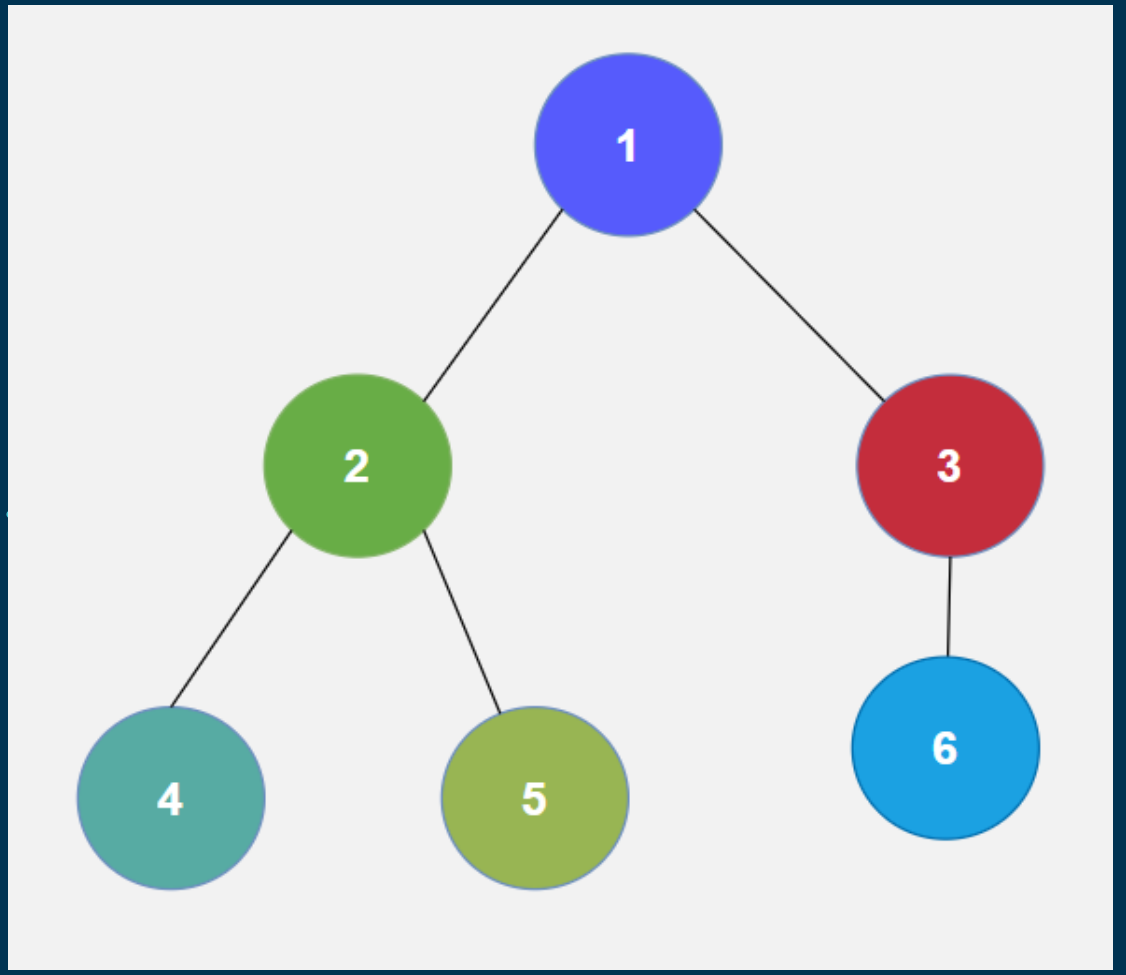
Data Structures used

- Node class – Has left and right children and value that can be assigned to it
- Queue of Nodes – In order to get values of nodes from user in a level order traversal way
- Stack of Nodes – For handling traversals in iterative way

Sample Runs

```
Enter the value of the root node:
1
Enter left child value of 1 (or -1 to skip):
2
Enter right child value of 1 (or -1 to skip):
3
Enter left child value of 2 (or -1 to skip):
4
Enter right child value of 2 (or -1 to skip):
5
Enter left child value of 3 (or -1 to skip):
-1
Enter right child value of 3 (or -1 to skip):
6
Enter left child value of 4 (or -1 to skip):
-1
Enter right child value of 4 (or -1 to skip):
-1
Enter left child value of 5 (or -1 to skip):
-1
Enter right child value of 5 (or -1 to skip):
-1
Enter left child value of 6 (or -1 to skip):
-1
Enter right child value of 6 (or -1 to skip):
-1
PreOrder Traveral: 1 2 4 5 3 6
InOrder Traveral: 4 2 5 1 3 6
PostOrder Traveral: 4 5 2 6 3 1
```

Recursive / Iterative

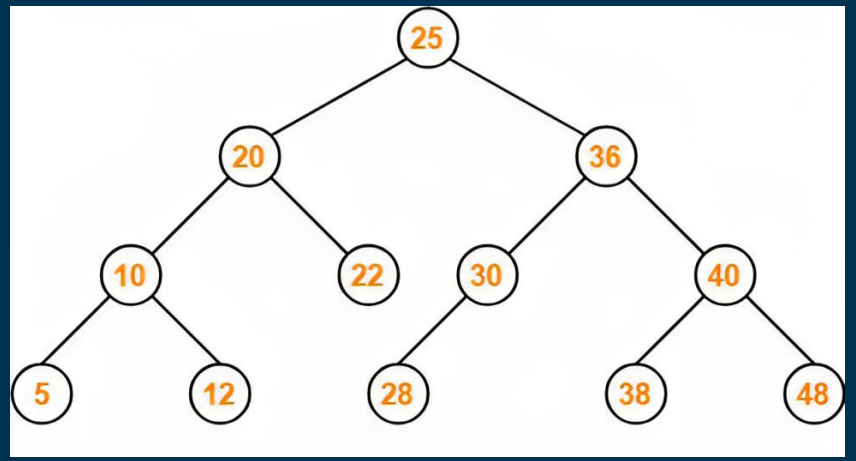


Visualization of result

Sample Runs Cont.

```
Enter 1 for iterative solution and 2 for recursive sol
1
Enter the value of the root node:
25
Enter left child value of 25 (or -1 to skip):
20
Enter right child value of 25 (or -1 to skip):
36
Enter left child value of 20 (or -1 to skip):
10
Enter right child value of 20 (or -1 to skip):
22
Enter left child value of 36 (or -1 to skip):
30
Enter right child value of 36 (or -1 to skip):
40
Enter left child value of 10 (or -1 to skip):
5
Enter right child value of 10 (or -1 to skip):
12
Enter left child value of 22 (or -1 to skip):
-1
Enter right child value of 22 (or -1 to skip):
-1
Enter left child value of 30 (or -1 to skip):
28
Enter right child value of 30 (or -1 to skip):
-1
Enter left child value of 40 (or -1 to skip):
38
Enter right child value of 40 (or -1 to skip):
48
Enter left child value of 5 (or -1 to skip):
-1
Enter right child value of 5 (or -1 to skip):
-1
Enter left child value of 12 (or -1 to skip):
-1
Enter right child value of 12 (or -1 to skip):
-1
Enter left child value of 28 (or -1 to skip):
-1
Enter right child value of 28 (or -1 to skip):
-1
Enter left child value of 38 (or -1 to skip):
-1
Enter right child value of 38 (or -1 to skip):
-1
Enter left child value of 48 (or -1 to skip):
-1
Enter right child value of 48 (or -1 to skip):
-1
PreOrder Traversal: 25 20 10 5 12 22 36 30 28 40 38 48
InOrder Traversal: 5 10 12 20 22 25 28 30 36 38 40 48
PostOrder Traversal: 5 12 10 22 20 28 30 38 48 40 36 25
```

Recursive / Iterative



Visualization of result

Thank You