Assignment No. 05

Title: Convert Given Binary Tree into a Threaded Binary Tree and Analyze Time & Space Complexity

Objectives:

- Understand the concept of a Threaded Binary Tree.
- Learn the implementation of converting a binary tree into a one-way threaded binary tree.
- Analyze the time and space complexity of the algorithm.

Introduction: A binary tree is a hierarchical data structure where each node has at most two children: left and right. Traditional binary trees require additional storage and traversal techniques to efficiently navigate through the tree. A **Threaded Binary Tree (TBT)** helps in optimizing tree traversal by utilizing empty right child pointers to store in-order successors, thus reducing the need for stack or recursive traversal.

A **One-Way Threaded Binary Tree** is a special type of binary tree where the right NULL pointers are converted into threads pointing to the in-order successor of the node. This allows in-order traversal without using additional stack space or recursion.

Theory:

1. **Threading in Binary Trees:**
    - A NULL right child is replaced with a pointer to its in-order successor.
    - This allows an efficient in-order traversal without recursion.
2. **Properties of One-Way Threaded Binary Trees:**
    - Right threads replace NULL pointers and point to the in-order successor.
    - Left pointers remain unchanged.
    - Traversal is more efficient compared to a traditional binary tree.
3. **Advantages:**
    - Eliminates the need for stack or recursion during traversal.
    - Reduces space complexity.
    - Improves traversal efficiency.

Implementation Details: The given C++ implementation follows these steps:

1. **Node Structure:** Each node consists of:

- data: Stores the value.
- left: Pointer to the left child.
- right: Pointer to the right child or in-order successor.
- rightThread: Boolean flag indicating if right is a thread.

2. **Insertion:**
   - If the right pointer is NULL, it is replaced with a thread pointing to the in-order successor.
   - The tree is traversed using standard binary search tree (BST) insertion logic.
3. **Leftmost Function:**
   - Finds the leftmost node in a subtree.
4. **Inorder Traversal:**
   - Starts from the leftmost node and follows in-order traversal using threads.
   - If a node has a thread, it follows it instead of recursively traversing.

Algorithm Analysis:

1. **Time Complexity:**
   - **Insertion:** O(h), where h is the height of the tree (in a balanced tree, h = O(log n)).
   - **Traversal (In-order):** O(n), as each node is visited once.
2. **Space Complexity:**
   - **O(n)** for storing n nodes.
   - **O(1)** additional space for traversal since no recursion or stack is used.

Conclusion: A **One-Way Threaded Binary Tree** optimizes traversal by replacing NULL right pointers with in-order successor links, reducing space complexity while maintaining efficient traversal operations. The implemented algorithm efficiently inserts nodes and performs in-order traversal without using extra stack space.

Assignment No. 06

Title:

Represent a given graph using an adjacency matrix/list to perform DFS and using an adjacency list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent landmarks as nodes and perform DFS and BFS on that.

Objectives:

1. To understand the concept of Graph Traversal techniques.
2. To learn and implement Depth-First Search (DFS) and Breadth-First Search (BFS).

Learning Objectives:

1.Understand graph traversal techniques using DFS and BFS.

2.Implement adjacency matrix and adjacency list representations.

3.Apply DFS using an adjacency matrix or list.

4.Apply BFS using an adjacency list.

Learning Outcomes:

- Define a class for the graph using Object-Oriented Programming (OOP) concepts.
- Analyze the working of DFS and BFS functions.
- Represent real-world scenarios using graphs.

Theory:

Graphs are fundamental data structures used to model relationships between different entities. They consist of **nodes (vertices)** and **edges (connections between nodes).**

Graph Representation:

1. **Adjacency Matrix:**
    - A 2D array where matrix[i][j] is 1 if there is an edge between node i and node j, otherwise 0.
    - Best for dense graphs but consumes more space for sparse graphs.
2. **Adjacency List:**
    - Each node maintains a list of all its neighboring nodes.

- Efficient for sparse graphs, using less memory than an adjacency matrix.

Graph Traversal Techniques:

1. Depth-First Search (DFS)

DFS is a recursive graph traversal algorithm that explores as far as possible along one branch before backtracking. It uses **stack (or recursion)** for traversal.

Algorithm for DFS:

1. Start at a given node.
2. Mark the node as visited and explore an adjacent unvisited node.
3. Repeat the process for each unvisited neighbor until no more nodes remain.
4. Backtrack if no adjacent unvisited nodes are left.

Time Complexity: O(V + E), where V = vertices, E = edges.

2. Breadth-First Search (BFS)

BFS explores all neighbors of a node before moving to the next level of neighbors. It uses a **queue** for traversal.

Algorithm for BFS:

1. Start at a given node and enqueue it.
2. Dequeue a node, mark it as visited, and enqueue all its unvisited neighbors.
3. Repeat until the queue is empty.

Time Complexity: O(V + E), where V = vertices, E = edges.

Graph Representation for the Given Problem:

For this assignment, we consider a **real-world map around the college** where:

- **Nodes** represent prominent landmarks (e.g., College, Library, Canteen, Bus Stop, Hostel, etc.).
- **Edges** represent roads connecting these landmarks.

Example Graph (Landmarks & Paths):

- **Nodes (Landmarks):**
  - A: College
  - B: Library
  - C: Canteen
  - D: Bus Stop
  - E: Hostel
- **Edges (Connections between landmarks):**
  - A ↔ B
  - A ↔ C
  - B ↔ D
  - C ↔ D
  - D ↔ E

Graph Representation:

1. **Adjacency Matrix Representation (for DFS)**

```
   A  B  C  D  E
A  [0, 1, 1, 0, 0]
B  [1, 0, 0, 1, 0]
C  [1, 0, 0, 1, 0]
D  [0, 1, 1, 0, 1]
E  [0, 0, 0, 1, 0]
```

2. **Adjacency List Representation (for BFS)**

```
A -> B, C
B -> A, D
C -> A, D
D -> B, C, E
E -> D
```

Software Required:

- g++ / gcc compiler (Linux/Windows).
- IDE: Eclipse / CodeBlocks / VS Code.

Input:

1. Number of landmarks (nodes).
2. Connections between landmarks (edges).

Output:

- Adjacency matrix representation (for DFS).
- Adjacency list representation (for BFS).

- DFS and BFS traversal sequence.

Conclusion:

This program provides insights into graph traversal techniques and their real-world applications. DFS is implemented using an adjacency matrix or list, and BFS is implemented using an adjacency list for efficient memory usage.

Outcome:

Upon completion, students will be able to:

- Understand graph representations and traversal techniques.
- Implement adjacency matrix and adjacency list structures.
- Perform DFS and BFS traversals on real-world graphs.

Possible University Exam Questions:

1. What are the differences between DFS and BFS?
2. Explain adjacency matrix and adjacency list with an example.
3. What is the time complexity of DFS and BFS?

Assignment no 07

Title:

There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph. Justify the storage representation used.

Objectives:

1. To understand concept of Graph data structure.
2. To understand concept of representation of graph.

Learning Objectives:

✓ To understand concept of Graph data structure

✓ To understand concept of representation of graph.

Learning Outcome:

- Define class for graph using Object Oriented features.
- Analyze working of functions.

Theory:

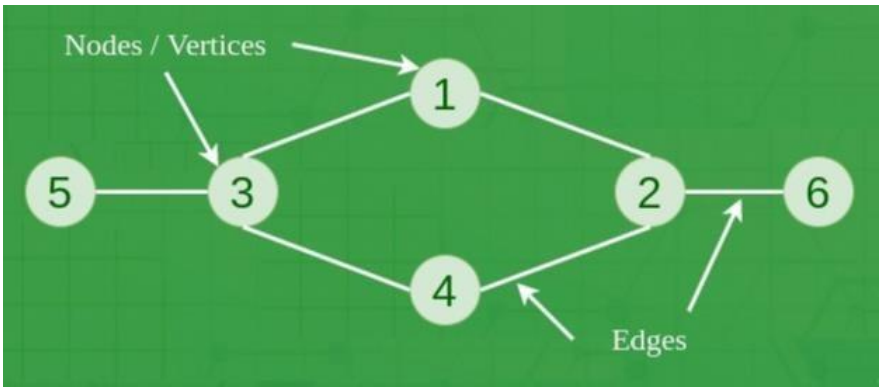
Graphs are the most general data structure. They are also commonly used data structures.

Graph definitions:

- A non-linear data structure consisting of nodes and links between nodes.

Undirected graph definition:

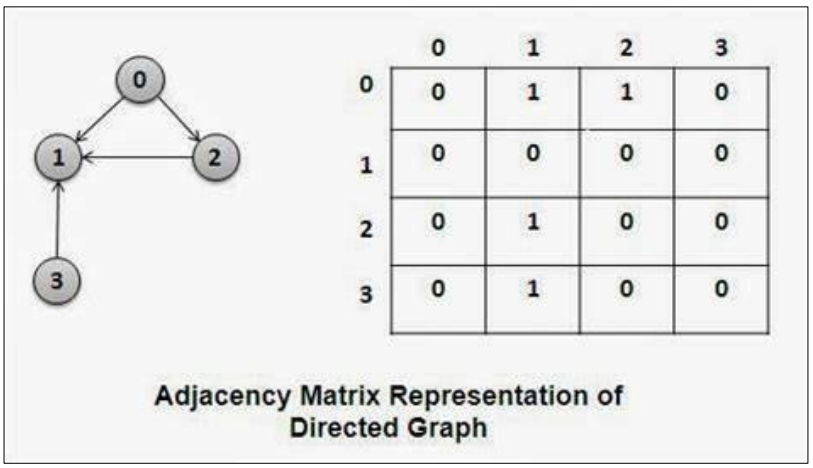
- An undirected graph is a set of nodes and a set of links between the nodes.
- Each node is called a **vertex**, each link is called an **edge**, and each edge connects twovertices.
- The order of the two connected vertices is unimportant.
- An undirected graph is a finite set of vertices together with a finite set of edges. Both setsmight be empty, which is called the empty graph.



Graph Implementation:

Different kinds of graphs require different kinds of implementations, but the fundamental concepts of all graph implementations are similar. We'll look at several representations for one particular kind of graph: directed graphs in which loops are allowed.

Representing Graphs with an Adjacency Matrix



Definition:

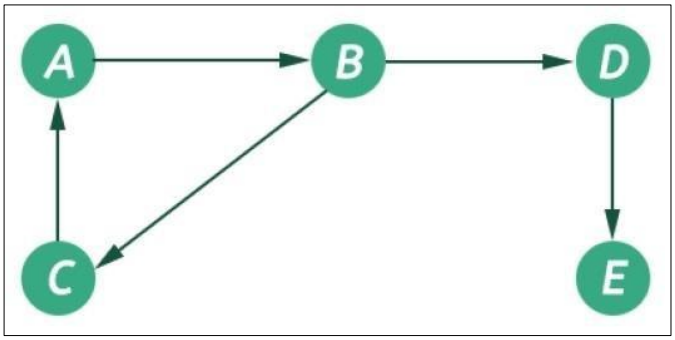
- An adjacency matrix is a square grid of true/false values that represent the edges of agraph.
- If the graph contains *n* vertices, then the grid contains *n* rows and *n* columns.
- For two vertex numbers *i* and *j*, the component at row *i* and column *j* is true if there is anedge from vertex *i* to vertex *j*; otherwise, the component is false.

  We can use a two-dimensional array to store an adjacency matrix:

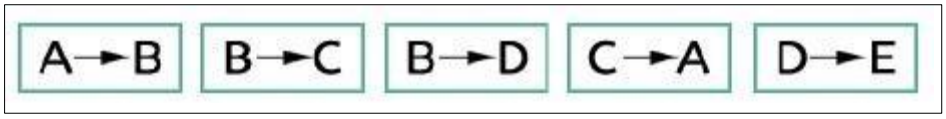
  boolean[][] adjacent = new boolean[4][4];

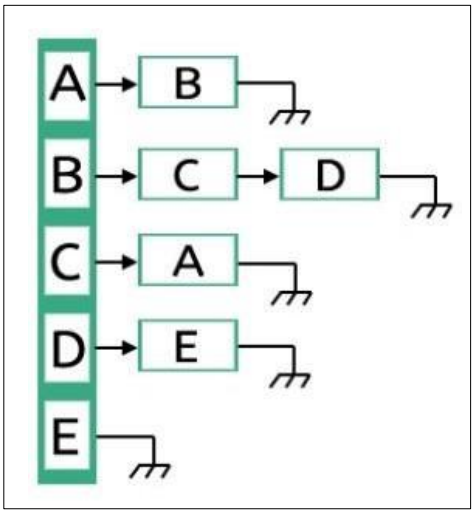
Once the adjacency matrix has been set, an application can examine locations of the matrix to determine which edges are present and which are missing.

Representing Graphs with Edge Lists



	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	1	0	0	0	0
D	0	0	0	0	1
E	0	0	0	0	0





Definition:

- A directed graph with *n* vertices can be represented by *n* different linked lists.
- List number *i* provides the connections for vertex *i*.
- For each entry *j* in list number *i*, there is an edge from *i* to *j*.

Loops and multiple edges could be allowed.

Representing Graphs with Edge Sets

To represent a graph with *n* vertices, we can declare an array of *n* sets of integers. For example:

IntSet[] connections = new IntSet[10]; // 10 vertices

A set such as connections[*i*] contains the vertex numbers of all the vertices to which vertex *i* is connected.

Software Required: g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Input: 1.Number of cities.
2.Time required to travel from one city to another.

Output: Create Adjacency matrix to represent path between various cities.

Conclusion: This program gives us the knowledge of adjacency matrix graph.

OUTCOME

Upon completion Students will be able to:

ELO1: Learn concept of graph data structure.

ELO2: Understand & implement adjacency matrix for graph.

Questions asked in university exam.

1. What are different ways to represent the graph? Give suitable example.
2. What is time complexity of function to create adjacency matrix?