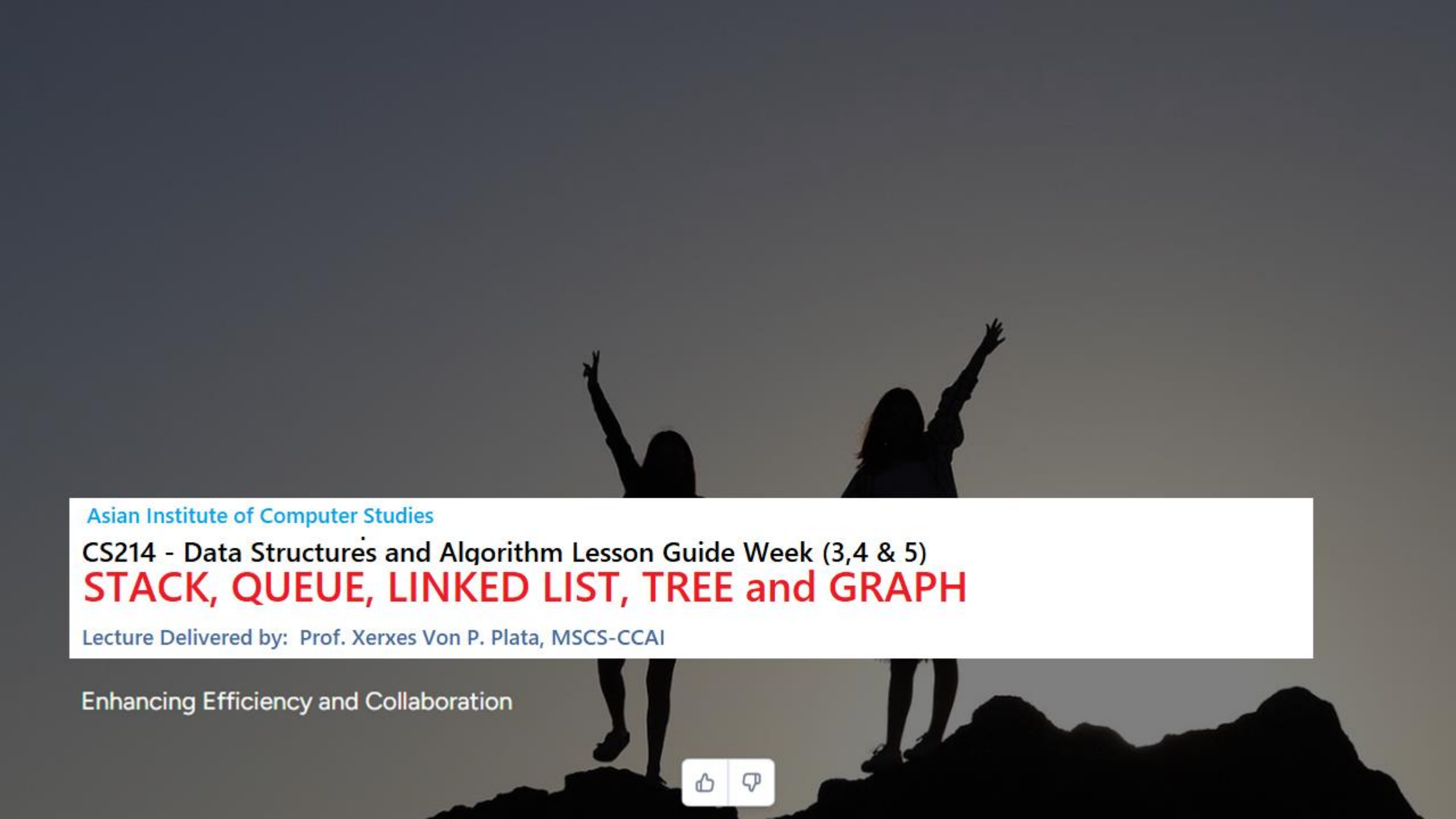


Asian Institute of Computer Studies

CS214 - Data Structures and Algorithm Lesson Guide Week (3,4 & 5)

STACK, QUEUE, LINKED LIST, TREE and GRAPH

Lecture Delivered by: Prof. Xerxes Von P. Plata, MSCS-CCAI

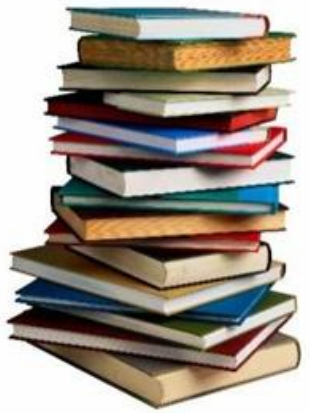
Enhancing Efficiency and Collaboration

Learning Outcomes

After completing this course you are expected to demonstrate the following:

1. Define the definition of stack, its operations, and disadvantages applications. It also helps you to understand expressions and how does it evauates using postfix.

Graphics Analysis:



After completing this course you are expected to demonstrate the following:

1. Identify the definition and structure of queue, its operations, types and applications in real-word. It also gives you an idea to determine the structure of linked list, its characteristics, its types and representation.

- Deletion

After completing this course you are expected to demonstrate the following:

1. Identify the definition of tree and its types. It discuss also the graph, its reprentation, uses of traversal and its application in programming.

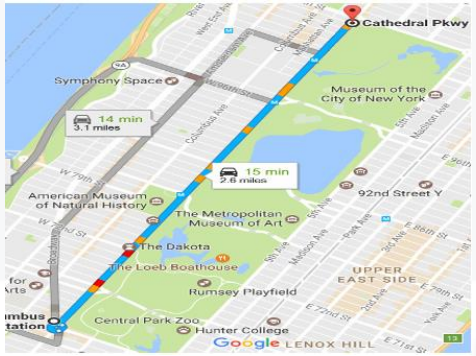


Table of contents

A. STACKS in Data Structures

Introduction to Data Structures

- **Definition**: A data structure is a way to store and organize data efficiently for various operations.
- **Types**:
  - Linear Data Structures (e.g., Arrays, Linked Lists, Stacks, Queues)
  - Non-linear Data Structures (e.g., Trees, Graphs)

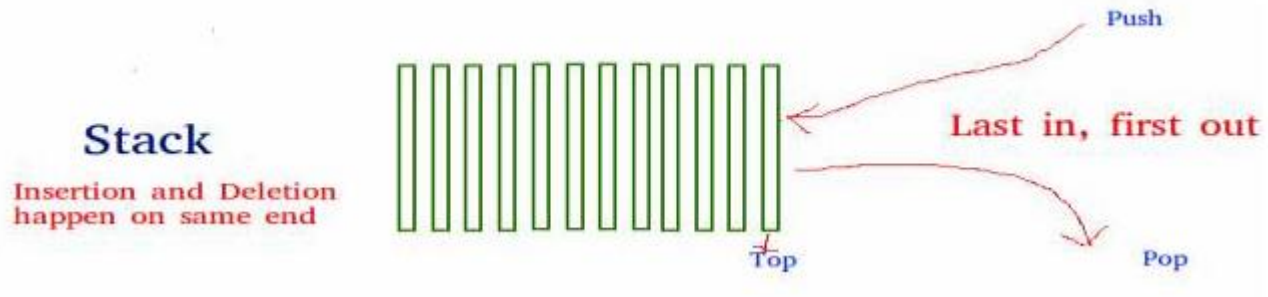
What are Stacks?

- **Definition**: A stack is a linear data structure that follows the **Last In, First Out (LIFO)** principle.
- **Real-life analogy**: Think of a stack of plates; the last plate placed on top is the first one to be taken off.
- **Key Operations**:
  - **Push**: Add an element to the top of the stack.
  - **Pop**: Remove an element from the top of the stack.
  - **Peek/Top**: View the top element without removing it.

STACKS in Data Structures

Visualizing Stack Operations

- **Diagram**: Show a stack with elements being pushed and popped (visualize LIFO).
- **Example**:
- Initial stack: [1, 2, 3]
- **Push(4)** → Stack: [1, 2, 3, 4]
- **Pop()** → Stack: [1, 2, 3]



Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Stack Applications

- **Function Call Stack**: Manages function calls in programming (recursion).
- **Undo/Redo Functionality**: Used in text editors or drawing applications.
- **Balancing Parentheses**: Check if parentheses in an expression are balanced.
- **Backtracking Algorithms**: E.g., maze solving.

B. QUEUE in Data Structures

What are Queues?

- **Definition**: A queue is a linear data structure that follows the **First In, First Out (FIFO)** principle.
- **Real-life analogy**: Think of a queue of people at a ticket counter; the first person in line is the first person served.
- **Key Operations**:
  - **Enqueue**: Add an element to the rear of the queue.
  - **Dequeue**: Remove an element from the front of the queue.
  - **Peek/Front**: View the front element without removing it.

Visualizing Queue Operations

- **Diagram**: Show a queue with elements being enqueued and dequeued (visualize FIFO).
- **Example**:
  - Initial queue: [1, 2, 3]
  - **Enqueue(4)** → Queue: [1, 2, 3, 4]
  - **Dequeue()** → Queue: [2, 3, 4]

QUEUE in Data Structures

Queue Applications

- **Scheduling Systems**: Like job scheduling in operating systems.
- **Buffer Management**: Used in printers, IO buffers, and CPU scheduling.
- **Breadth-First Search (BFS)**: Algorithm in graph traversal.

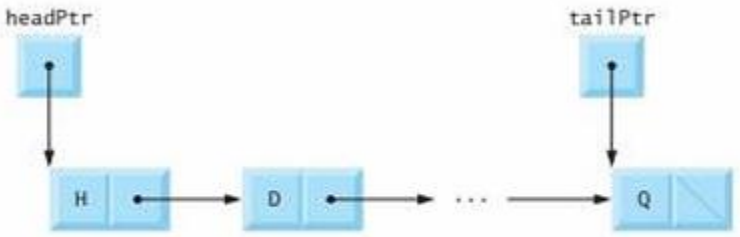


Figure 4.2: A queue with several nodes

Types of Queues

- **Simple Queue**: FIFO model (as explained earlier).
- **Circular Queue**: The last position is connected to the first to form a circle.
- **Priority Queue**: Each element has a priority; elements with higher priority are dequeued first.
- **Double-Ended Queue (Deque)**: Insertion and deletion can happen from both ends.

QUEUE in Data Structures

Slide 11: Key Differences

Feature	Stack	Queue
Insertion	Top (Push)	Rear (Enqueue)
Removal	Top (Pop)	Front (Dequeue)
Order	LIFO	FIFO
Example	Undo in editors	Print queue

Slide 12: Code Example: Stack in Python

python

```python
stack = []

# Push elements
stack.append(1)
stack.append(2)
stack.append(3)

# Pop elements
stack.pop()  # Outputs 3
stack.pop()  # Outputs 2
```

Slide 13: Code Example: Queue in Python

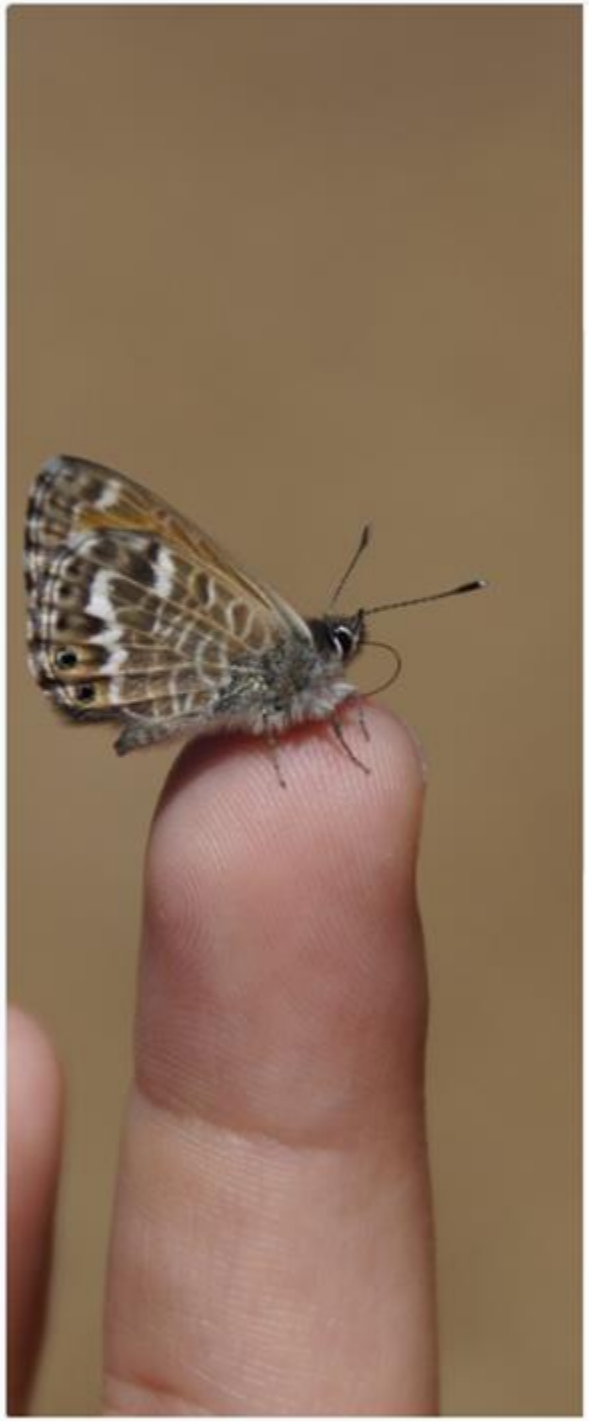
python

```python
from collections import deque

queue = deque()

# Enqueue elements
queue.append(1)
queue.append(2)
queue.append(3)

# Dequeue elements
queue.popleft()  # Outputs 1
queue.popleft()  # Outputs 2
```



C. LINKED LIST in Data Structures

Definition of Linked Lists:

- A linked list is a linear data structure where elements, called nodes, are linked using pointers.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array.

Structure of a Node:

- Each node consists of two parts: data and a reference (or pointer) to the next node.

Comparison to Arrays:

- Unlike arrays, linked lists do not require contiguous memory allocation.
- Linked lists have dynamic size, which makes them more flexible in memory usage.

LINKED LIST in Data Structures

Types of Linked Lists

Singly Linked List:

- Each node points to the next node in the sequence.
- Traversal is one-directional, from the head node to the last node.

Doubly Linked List:

- Each node points to both the next and the previous node.
- Allows traversal in both forward and backward directions.

Circular Linked List:

- The last node points back to the head, forming a circular structure.
- Can be either singly or doubly linked.

Linked list can be visualized as a chain of nodes, where every node point to the next r

Linked List Representation

LINKED LIST in Data Structures

Applications of Linked Lists

Dynamic Memory Management:

- Linked lists are often used to manage free blocks of memory

Graph Implementations:

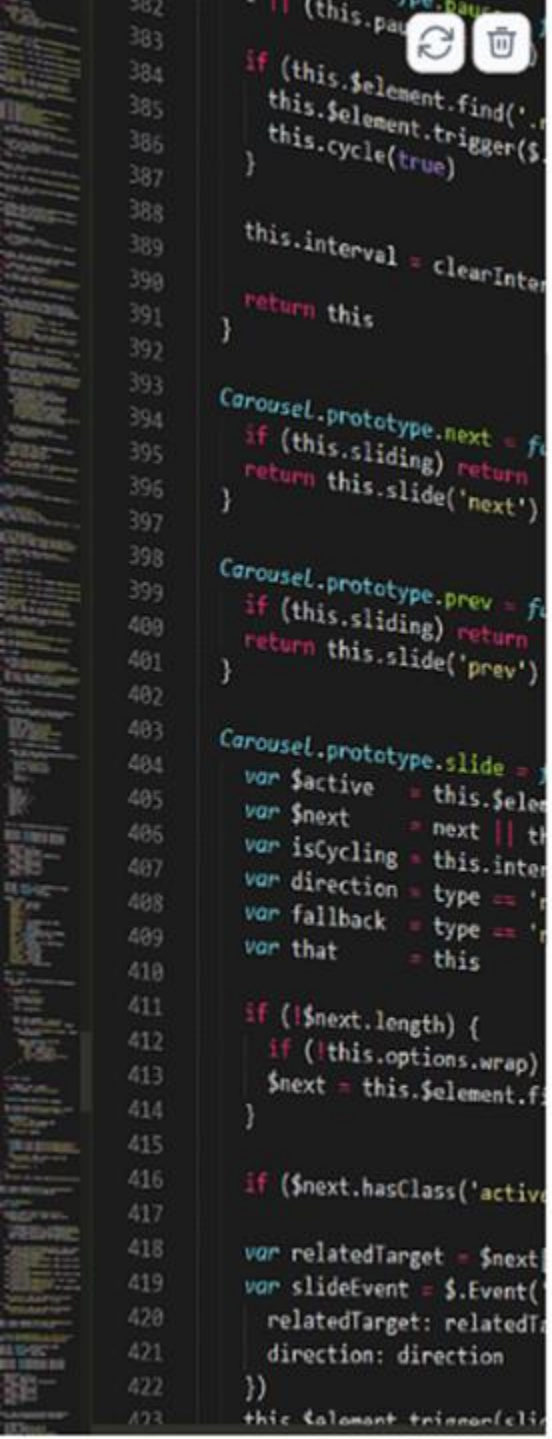
- Linked lists are used to represent adjacency lists for graphs,

Undo Mechanisms:

- In applications like text editors, linked lists track changes for the undo functionality.

Real-Time Systems:

- Circular linked lists are used in round-robin scheduling algorithms in operating systems.



D. TREES in Data Structures

Overview of Trees

What is a Tree?

- A hierarchical data structure with a set of nodes connected by edges.
- Characteristics:
  - Root: The topmost node.
  - Parent-Child Relationship.
  - No cycles or loops.
- Tree Terminology:
  - Root, Leaf, Sibling, Subtree, Depth, Height.

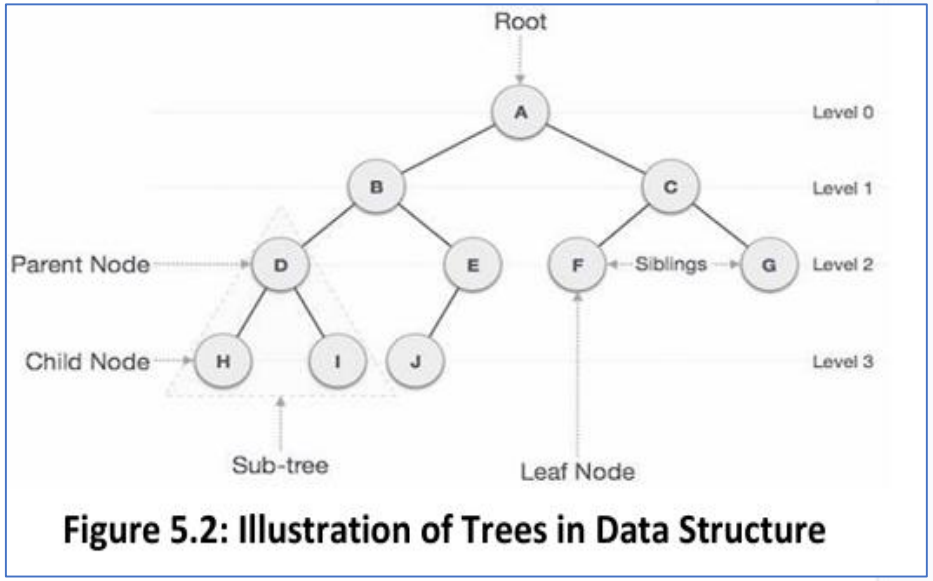


Figure 5.2: Illustration of Trees in Data Structure

Types of Trees

- 1. Binary Tree
  - Each node has at most 2 children (left and right).

TREES in Data Structures

- 2. Binary Search Tree (BST)
  - Left child < Parent < Right child.
  - Efficient for search, insert, delete operations.
- 3. AVL Tree (Self-balancing)
  - A balanced BST ensuring O(log n) search and insertion time.
- 4. Heap
  - Min-Heap: Parent node is smaller than children.
  - Max-Heap: Parent node is larger than children.

Slide 6: Applications of Trees

- **Binary Search Trees (BST):**
  - Used in databases for indexing and searching records.
- **Heap:**
  - Used in priority queues.
- **Parse Trees:**
  - Used in compilers to parse expressions.
- **Trie (Prefix Tree):**
  - Used in searching words in a dictionary.

Common Operations on Trees

- Traversal
- Search and Insertion
- Deletion

TREES in Data Structures

Important Terms

Following are the important terms with respect to tree.

1. **Path** – Path refers to the sequence of nodes along the edges of a tree.
2. **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
3. **Parent** – Any node except the root node has one edge upward to a node called parent.
4. **Child** – The node below a given node connected by its edge downward is called its child node.
5. **Leaf** – The node which does not have any child node is called the leaf node.
6. **Subtree** – Subtree represents the descendants of a node.
7. **Visiting** – Visiting refers to checking the value of a node when control is on the node.
8. **Traversing** – Traversing means passing through nodes in a specific order.
9. **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
10. **Keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

Algorithm

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

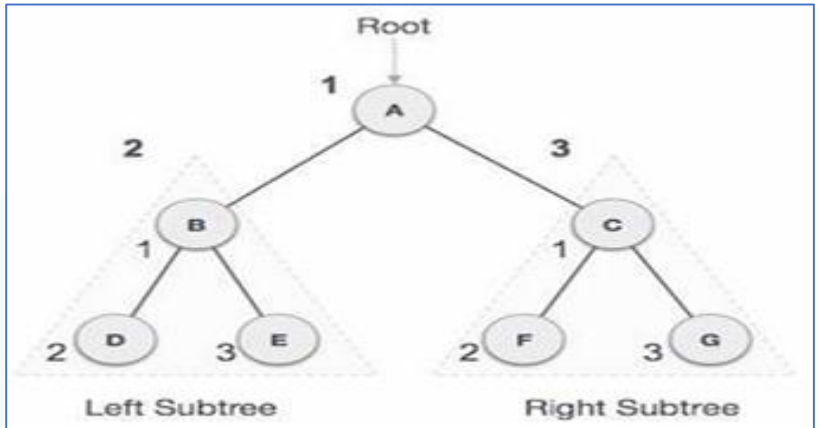
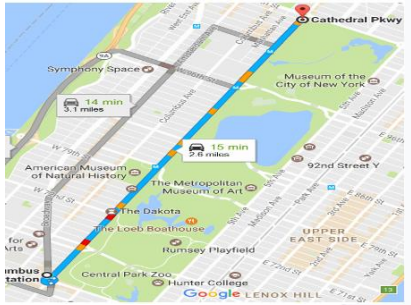


Figure 5.5

E. GRAPHS in Data Structures

Introduction to Graphs

- What is a Graph?
  - A collection of nodes (vertices) and edges that may or may not be directed.
  - Unlike trees, graphs can have cycles.
- Graph Terminology:
  - **Vertex**: A node in a graph.
  - **Edge**: A connection between two vertices.
  - **Directed Graph**: Edges have direction.
  - **Undirected Graph**: Edges have no direction.
  - **Weighted Graph**: Edges carry weights.
  - **Cycle**: A path where the start and end vertices are the same.



GRAPHS in Data Structures

Types of Graphs

- 1. Directed vs Undirected Graphs
  - **Directed**: Edges have a direction (e.g., Webpage links).
  - **Undirected**: Edges have no direction (e.g., Social network connections).
- 2. Weighted vs Unweighted Graphs
  - **Weighted**: Edges have weights (e.g., distance between cities).
  - **Unweighted**: Edges are either present or absent without weights.
- 3. Connected vs Disconnected Graphs
  - **Connected**: There's a path between any pair of nodes.
  - **Disconnected**: Some nodes are isolated.

Formally, a graph is a pair of sets (V, E),

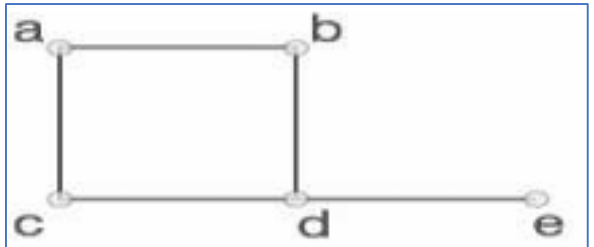
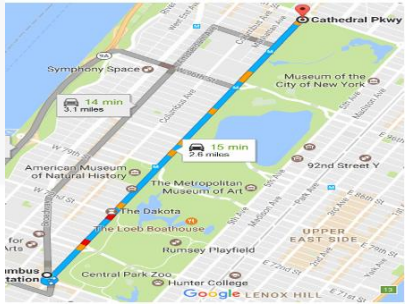


Figure 5.11

where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.

In the given graph,
V = {a, b, c, d, e}
E = {ab, ac, bd, cd, de}



GRAPHS in Data Structures

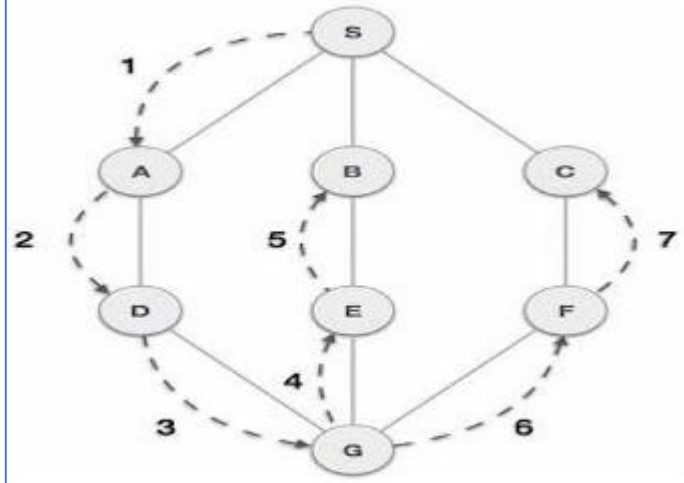


Figure 5.12

Graphs Algorithms

Depth First Search (DFS) Traversal

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

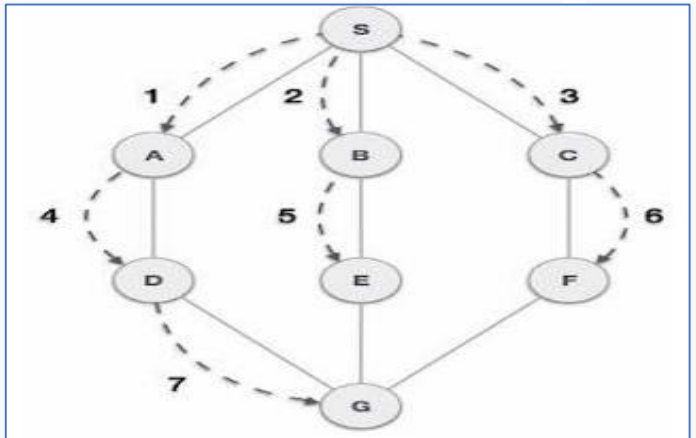
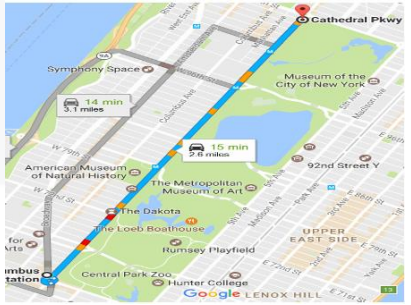


Figure 5.13

Breadth First Search (BFS) Traversal

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



GRAPHS in Data Structures

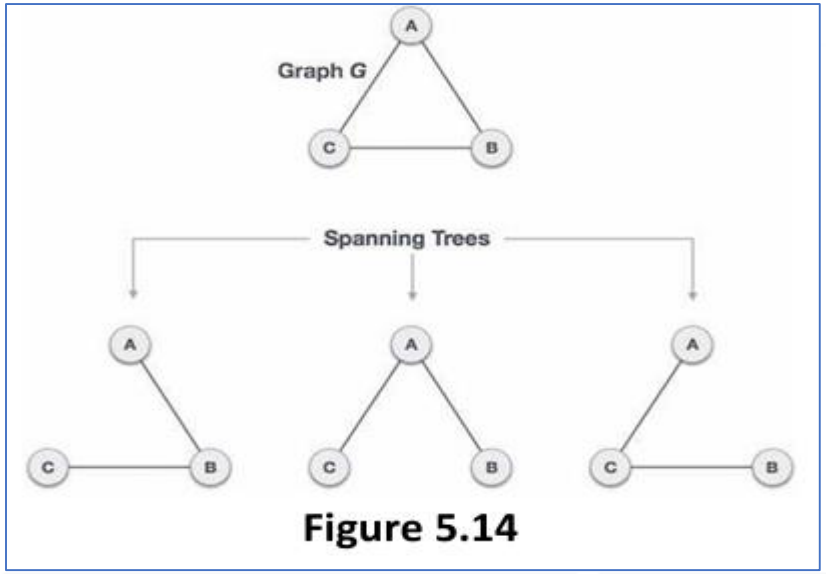


Figure 5.14

Spanning Tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

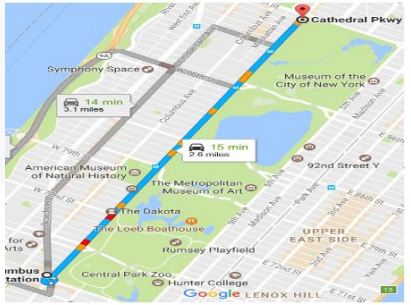
Application of Spanning Tree

Common application of spanning trees is –

1. Civil Network Planning
2. Computer Network Routing Protocol
3. Cluster Analysis

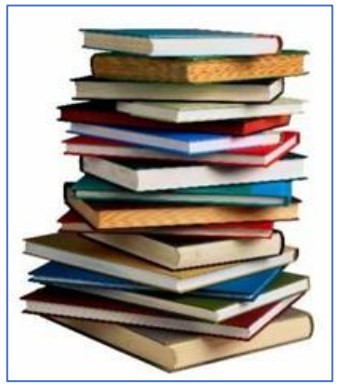
Applications of Graphs in Real-Life

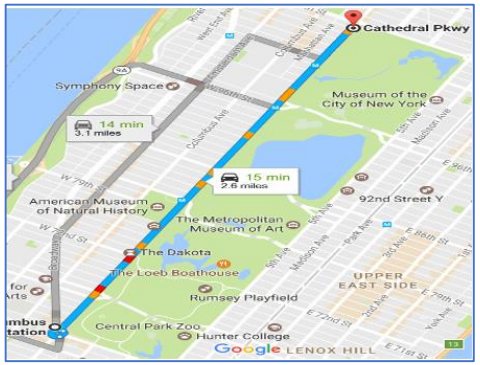
1. Connecting with friends on social media, where each user is a vertex, and when users connect, they create an edge.
2. Using GPS/Google Maps/Yahoo Maps, to find a route based on shortest route.
3. Google, to search for webpages, where pages on the internet are linked to each other by hyperlinks; each page is a vertex and the link between two pages is an edge.
4. On eCommerce websites relationship graphs are used to show recommendations.



Conclusion and Future Trends

Stacks and queues are **array-like data structures that differ only in how elements are added and removed**. Linked lists, trees, and graphs are structures with nodes that keep references to other nodes.





QUIZ NEXT MEETING