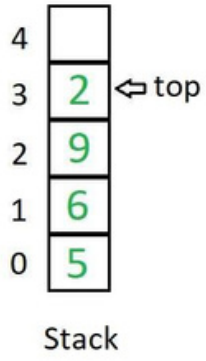
Unit 2

Stack

Definition: A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. This means the last element added to the stack is the first one to be removed.



Operations:

a. Push: Adds an element to the top.
b. Pop: Removes the top element.
c. Peek/Top: Returns the top element without removal.

Explain push and pop operations:

Push: Push operationin stackuse to add an element to the top of the stack. It place new element at the top, shifting the existing element down.

This operation increases the size of the stack by 1 and

Pop: Pop operation in stack used to remove the top element from the stack. The previous element becomes the new top of the stack.

This operation decreases the size of the stack by 1.

Characteristics:

1. LIFO Order: Last In, First Out principle.
2. Constant-time Operations: Push and pop operations have constant time complexity.
3. Limited Access: Only the top element can be accessed.
4. Efficient for Recursive Algorithms: Ideal for managing function calls and recursion.
5. Limited Capacity: Size is constrained; may lead to overflow.
6. Dynamic Resizing: Some implementations dynamically adjust size for flexibility.

Applications:

1. Function Calls and Recursion: Stack manages the execution flow of function calls, supporting recursive algorithms efficiently.
2. Infix to Postfix Conversion: Stacks assist in converting infix expressions to postfix notation, handling operators and their precedence.
3. Postfix Evaluation: Stacks are used to evaluate expressions in postfix notation by managing operands and operators.
4. Undo Mechanisms in Software: Stacks track changes, enabling easy reversal of actions in applications.
5. Memory Management: Handles local variables and function calls, aiding in efficient memory allocation.
6. Depth-First Search Algorithms: Stack is used in graph traversal methods DFS.
7. Backtracking Algorithms: Essential for managing choices and exploring solutions in a systematic way.

Advantages:

1.**Simplicityand Efficiency:** Stack provides a simple, efficient way to manage data, particularly in scenarios where the Last In, First Out (LIFO) order is beneficial.

2. **Constant-Time Operations:** Push and pop operations have constant time complexity, ensuring swift data manipulation.

3. **Well-Suited for Certain Algorithms:** Stack's LIFO nature is advantageous in algorithms relying on a last-in-first-out approach.

4. **Memory Management in Recursion**: Facilitates efficient memory management during recursive calls, optimizing space usage.

5. **Easy Implementation:** The stack is easy to implement, making it a versatile choice for various applications.

Disadvantages:

1. **Limited Access:** Restricted to accessing only the top element, lacking random access capabilities.
2. **Limited Capacity:** The size is constrained, potentially leading to stack overflow issues.
3. **Dynamic Resizing Overhead:** emplementations that dynamically resize may incur additional overhead.
4. **Inefficiency for Some Operations**: While efficient for certain tasks, stacks may not be optimal for operations requiring different access patterns.
5. **Potential for Stack Overflow:** Recursive scenarios can lead to a stack overflow, especially in resource-intensive applications.

Write Push algorithm

Step 1: Begin

Step 2: If Top = Size-1, then print Overflow & exit.

Step 3: Input new item.

Step 4: Top ← Top + 1.

Step 5: Stack[Top] ← Item.

Step 6: Exit

Write pop algorithm

Step 1: Begin

Step 2: If Top = -1, then print Underflow & exit.

Step 3: Item ← Stack[Top].

Step 4: Top ← Top - 1.

Step 5: Print Item.

Step 6: End

Write Stack Traversal algorithm

Step 1: Begin

Step 2: If Top = -1, then print Stack is empty & exit.

Step 3: Set current position to Top.

Step 4: While current position is greater than or equal to 0, do steps 5-7.

Step 5: Print Stack[current position].

Step 6: Decrease current position by 1.

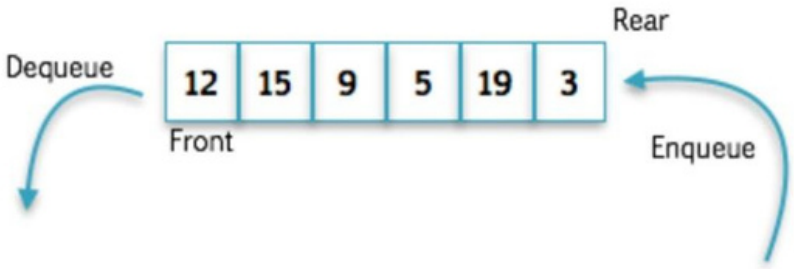
Step 7: Repeat step 4.

Step 8: End

Queue

Definition:

A queue is a linear data structure that follows the First In, First Out (FIFO) principle, where elements are added at the rear and removed from the front.



Operations:

- Enqueue: Add element to the rear.
- Dequeue: Remove element from the front.

Characteristics

- FIFO Order: Queue follows a First In, First Out order, ensuring that the element inserted first is the one to be removed first.
- Two Pointers - Front and Rear: Queues are managed using two pointers – front and rear. Front points to the first element, and rear points to the last element.
- Limited Access (Front and Rear): Elements can only be added at the rear and removed from the front, limiting direct access to elements in the middle of the queue.

Application

1. Task Scheduling: Queues are used in operating systems to manage tasks scheduled for execution, ensuring a fair and orderly processing sequence.
2. Breadth-First Search in Graphs: Queues play a crucial role in algorithms like Breadth-First Search (BFS), exploring graph structures level by level.
3. CPU Scheduling: Operating systems utilize queues for managing the execution of processes in a fair and efficient manner.
4. Call Center Systems: Queues are employed in call centers to organize incoming calls, ensuring they are handled based on their arrival time.

5. Multi programming: Multi programming means when multiple programs are running in the main memory. It is essential to organize these multiple programs and these multiple programs are organized as queues.

Advantage

1. Simple and Easy Implementation: Queues have a straightforward structure, making them easy to implement and understand.
2. Efficient for Managing Tasks Sequentially: Queues are highly effective for scenarios where tasks need to be processed in a sequential manner, following the FIFO order.
3. Supports Dynamic Size: Queues can dynamically adjust their size based on the number of elements, providing flexibility in handling varying workloads.

Write insertion algorithm:

Step 1: Begin

Step 2: If (Rear = MaxSize - 1), then print Queue is full & exit.

Step 3: If Front = -1, set Front to 0.

Step 4: Input new item.

Step 5: Increment Rear ← Rear + 1.

Step 6: Set Queue[Rear] ← Item.

Step 7: End

Write Deletion algo

Step 1: Begin

Step 2: If Front = -1, then print Queue is empty & exit.

Step 3: Set Item to Queue[Front].

Step 4: If Front = Rear, set Front and Rear to -1 (empty queue).

Step 5: Otherwise, increment Front ← Front + 1.

Step 6: Print Item.

Step 7: End

Traversing algo

Step 1: Begin

Step 2: If Front = -1, then print Queue is empty & exit.

Step 3: Set current position to Front.

Step 4: While current position is less than or equal to Rear, do steps 5-7.

Step 5: Print Queue[current position].

Step 6: Increment current position ← current position + 1.

Step 7: Repeat step 4.

Step 8: End

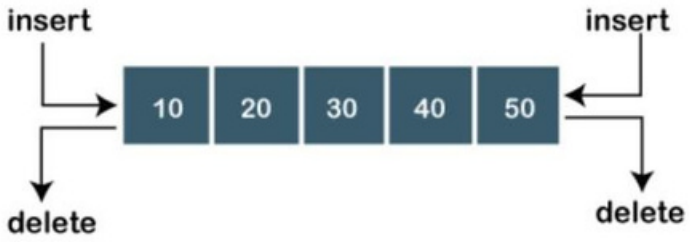
Types:

1. De-queue Queue
2. Circular Queue
3. Priority Queue

De-Queue or Deque (Double ended queue)

Definition:

Dequeue (Double Ended Queue) is a linear data structure that allows insertion and deletion at both ends, providing front and rear access.



• Operations:

- Enqueue Front: Add element to the front.
- Enqueue Rear: Add element to the rear.
- Dequeue Front: Remove element from the front.
- Dequeue Rear: Remove element from the rear.

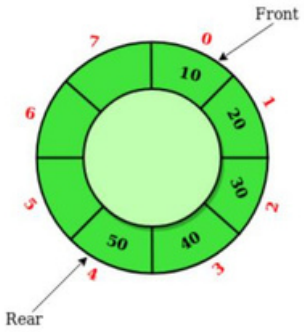
• Types:

- Input-Restricted Dequeue: An input-restricted deque is a type of deque where deletion can be made from both ends, but insertion can be made at one end only
- Output-Restricted Dequeue: An output-restricted deque is a type of deque where insertion can be made at both ends, but deletion can be made from one end only.

Circular Queue

Definition:

CircularQueue is a type of a linear queue with the front and rear pointers connected, forming a circular structure, allowing efficient use of memory.



• Operations:

1. Enqueue: Add element to the rear.
2. Dequeue: Remove element from the front.

Characteristics

1. Follows FIFO Order: Circular Queue maintains the First In, First Out order, ensuring that the element inserted first is the first to be removed.
2. Efficiently Utilizes Memory: The circular structure of the queue allows for efficient memory usage, as elements can wrap around the queue, utilizing available space more effectively.
3. Two Pointers - Front and Rear: Circular Queues use two pointers – front and rear. Elements are enqueued at the rear and dequeued from the front, and the pointers are connected in a circular fashion.
4. Provides a Continuous Loop: The circular nature of the queue provides a continuous loop, offering a seamless transition from the last to the first element, optimizing space utilization.
5. Efficient for Real-Time Applications: Circular Queues find applications in real-time systems where a continuous loop and efficient memory usage contribute to timely and predictable processing of tasks.

Application

1. Resource Allocation: Used in computer systems for efficient resource allocation, ensuring a continuous loop for task processing.
2. Round-Robin Task Handling: Circular Queues play a crucial role in scheduling algorithms like Round Robin, where tasks are assigned time slices in a circular manner for fair processing.
3. Memory Management in Embedded Systems: Circular Queues optimize memory usage in embedded systems, providing seamless data handling in a loop.
4. Simulation of Processes: Employed in simulations for cyclic processing of tasks, replicating real-world scenarios efficiently.

5. Inter-process communication: Circular queue can be used for communication between different processes.

Advantage

1. Efficient Memory Usage: Circular Queues optimize memory by efficiently utilizing available space, contributing to better memory management.
2. Continuous Loop Structure: The circular nature allows for a seamless loop, promoting efficient handling of tasks in a consistent and predictable manner.
3. Simple Implementation: Circular Queues are relatively simple to implement compared to dynamic linear queues, offering a balance between simplicity and functionality.
4. Complexity: All operations occur in O(1) constant time

Disadvantages of Circular Queue:

- In a circular queue, the number of elements you can store is only as much as the queue length, you have to know the maximum size beforehand.
- Some operations like deletion or insertion can be complex in circular queue.
- The implementation of some algorithms like priority queue can be difficult in circular queue.
- Circular queue has a fixed size, and when it is full, there is a risk of overflow if not managed properly.
- In the array implementation of Circular Queue, even though it has space to insert the elements it shows that the Circular Queue is full and gives the wrong size in some cases.

Write an algorithm to insert an item in circular

Step 1: Begin

Step 2: If (rear + 1) % SIZE = front, then print Queue is full & exit.

Step 3: If front = -1, set front and rear to 0.

Step 4: Input new item.

Step 5: Rear ← (rear + 1) % SIZE.

Step 6: Queue[rear] ← Item.

Step 7: End

Write an deletion algo of circular queue

Step 1: Begin

Step 2: If Front = -1, then print Queue is empty & exit.

Step 3: Set Item to Queue[Front].

Step 4: If Front = Rear, set Front and Rear to -1 (empty queue).

Step 5: Otherwise, set Front to (Front + 1) % MaxSize.

Step 6: Print Item.

Step 7: End

Write traversing algorithm of circular Queue

Step 1: Begin

Step 2: If Front = -1, then print Queue is empty & exit.

Step 3: Set current position to Front.

Step 4: While current position is less than or equal to Rear, do steps 5-7.

Step 5: Print Queue[current position].

Step 6: Set current position to (current position + 1) % MaxSize.

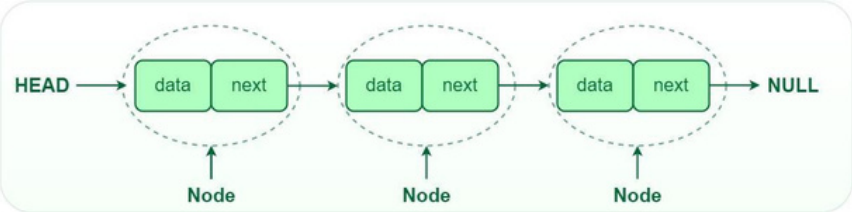
Step 7: Repeat step 4.

Step 8: End

Unit 3

Introduction to Linkedlist

What is Linked list: A linked list is a linear data structure that consists of a collection of nodes, each node containing a data field and a reference (link/next) to the next node in the list. The first node in the list is called the head node, and the last node is called the tail node.

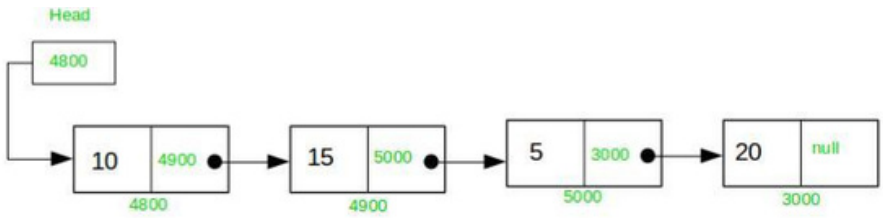


OperationsonLinkedlist:

- Insertion
- Deletion
- Search
- Traversal

Singly linked list:

Asinglylinkedlist isaspecial type of linked list in which each node has only one link that points to the next node in the linked list.



- **Operations**:

1. Insertion.
2. Deletion.
3. Traversal.
4. Searching
5. Concatenation

Characteristics:

1. Node Structure:
   Each node holds both data and a reference (next-pointer) to the next node in the sequence.
2. Dynamic Size:
   Singly linked lists can dynamically adjust in size, allowing for efficient insertion and deletion of nodes.
3. Sequential Access Only:
   Traversal starts from the head node, progressing sequentially through each node, limiting access to a linear fashion.
4. Memory Efficiency:
   Requires less memory compared to arrays due to the absence of fixed-size allocation; nodes use only the necessary memory.
5. No Wasted Memory Gaps:
   Unlike arrays, there are no gaps or unused memory slots, as each node consumes memory only for data and the reference to the next node.

Advantages:

1. Dynamic Size:

- Singly linked lists can dynamically adjust in size, accommodating varying data requirements without the need for pre-allocation.

2. Efficient Insertions and Deletions:

- Inserting or deleting a node is efficient as it involves updating only the pointers, without the need to shift or reallocate other elements.

3. No Pre-Allocation of Memory:

- Memory is allocated on-demand, minimizing wasted memory and making it more adaptable to changing data sizes.

4. Ease of Implementation:

- Implementation is straightforward, making it a simpler choice for scenarios where a basic, flexible data structure is sufficient.

5. Memory Utilization Efficiency:

- Singly linked lists use memory efficiently, especially in dynamic scenarios, as they allocate memory as needed, reducing unnecessary overhead.

Disadvantages:

1. Inefficient Random Access:
   Accessing a specific element requires traversing the list sequentially from the beginning, making random access operations less efficient compared to direct indexing in arrays.
2. Extra Memory for Pointers:
   Each node contains a reference to the next node, adding extra memory overhead. This can be a concern when memory usage is a critical factor.
3. Traversal Needed for Access:
   To access or locate a particular element, the entire list must be traversed from the head, which can be time-consuming for large lists.
4. Memory Overhead:
   The need for maintaining pointers increases memory overhead compared to other data structures, especially in scenarios where memory is a limiting factor.
5. Reverse Traversal is Impractical:
   Unlike arrays, singly linked lists do not support efficient reverse traversal, requiring additional mechanisms if backward access is a common requirement.

Operations of Singly Linked

Exam tips ## in exam you can find a question asking for writing an algorithm. Memorizing this algorithm will be easy if you understand them by watching tutorials on YouTube.

Algorithm of insertion at the Beginning:

Step 1: Begin

Step 2: Create a new node (newNode) and allocate memory for it.

Step 3: Input the value to be inserted at the beginning (element).

Step 4: Set newNode -> data to element.

Step 5: Set newNode -> link to the head -> link.

Step 6: Set head -> link to newNode.

Step 7: End

Insertion at the end

Step 1: Begin

Step 2: Create a new node (`newNode`) and allocate memory for it.

Step 3: Input the value to be inserted at the end (`element`).

Step 4: Set `newNode -> data` to `element`.

Step 5: Set `newNode -> link` to `NULL`.

Step 6: Initialize a pointer (`ptr`) to `head`.

Step 7: While `ptr -> link` is not `NULL`, update `ptr` to `ptr -> link`.

Step 8: Set `ptr -> link` to `newNode`.

Step 9: End

Insertion after a particular element

Step 1: Begin

Step 2: Input the value to be inserted after (keyElement).

Step 3: Create a new node (newNode) and allocate memory for it.

Step 4: Set newNode -> data to the value to be inserted.

Step 5: Initialize a pointer (ptr) to the head node.

Step 6: Traverse the linked list until finding a node with data equal to keyElement or reaching the end.

Step 7: If the keyElement is found:

- Set newNode -> link to ptr -> link.
- Set ptr -> link to newNode.

Step 8: If the keyElement is not found, print "Element not found."

Step 9: End

Insertion before a particular element:

Step 1: Begin

Step 2: Input the value to be inserted before (keyElement).

Step 3: Create a new node (newNode) and allocate memory for it.

Step 4: Set newNode -> data to the value to be inserted.

Step 5: Initialize two pointers, ptr and prev, both initially pointing to the head node.

Step 6: While ptr is not NULL and ptr -> data is not equal to keyElement:

- Set prev to ptr.
- Move ptr to the next node (ptr = ptr -> link).

Step 7: If ptr is not NULL:

- Set prev -> link to newNode.
- Set newNode -> link to ptr.

Step 8: If ptr is NULL, print "Element not found."

Step 9: End

Deletion

Algo of Deletion at beginning

Step 1: Begin

Step 2: If head -> link is NULL, print "List is empty" and exit.

Step 3: Create a pointer temp and set it to head -> link.

Step 4: Set head -> link to temp -> link.

Step 5: Free the memory allocated for temp.

Step 6: End

Deletion after a particular element

Step 1: Begin

Step 2: Input the value to be deleted after (keyElement).

Step 3: Initialize two pointers, ptr and prev, both initially pointing to the head node.

Step 4: While ptr is not NULL and ptr -> data is not equal to keyElement:

- Set prev to ptr.
- Move ptr to the next node (ptr = ptr -> link).

Step 5: If ptr is not NULL:

- Set prev -> link to ptr -> link.
- Free the memory allocated for the node pointed by ptr.

Step 6: If ptr is NULL, print “Element not found.”

Step 7: End

Deletion at the End

Step 1: Begin

Step 2: Check if head -> link is NULL (empty list).

- If true, print "List is empty" and exit.
- If false, proceed to the next step.

Step 3: Initialize two pointers, ptr and prev, both initially pointing to the head node.

Step 4: While ptr -> link is not NULL:

- Set prev to ptr.
- Move ptr to the next node (ptr = ptr -> link).

Step 5: Set prev -> link to NULL.

Step 6: Free the memory allocated for the node pointed by ptr.

Step 7: End

Traversing:

Step 1: Begin

Step 2: Initialize a pointer (ptr) to the head node.

Step 3: While ptr-> link is not NULL:

- Move ptr to the next node (ptr = ptr -> link).
- Print the data in the current node (ptr -> data).

Step 4: End

Searching algorithm

Step 1: Begin

Step 2: Input the value to be searched (searchElement).

Step 3: Initialize a pointer (ptr) to the head node.

Step 4: Initialize a variable (position) to keep track of the position (starting from 1).

Step 5: While ptr is not NULL and ptr -> data is not equal to searchElement:

- Move ptr to the next node (ptr = ptr -> link).
- Increment position by 1.

Step 6: If ptr is not NULL, print "Element found at position: position."

- If ptr is NULL, print "Element not found."

Step 7: End

Implement Queue using Linkedlist.

Insertion (Enqueue)

Step 1 - Create a newNode with given value and set 'newNode → next' to NULL.

Step 2 - Check whether queue is Empty (rear == NULL)

Step 3 - If it is Empty then, set front = newNode and rear = newNode.

Step 4 - If it is Not Empty then, set rear → next = newNode and rear = newNode.

Deletion(Dequeue)

Step 1 - Check whether queue is Empty (front == NULL).

Step 2 - If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function

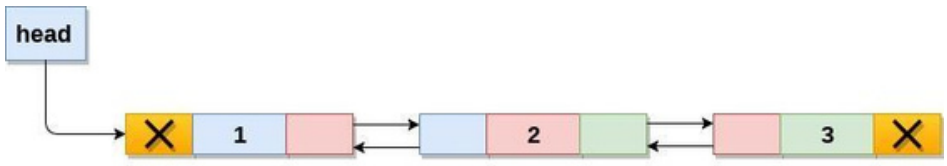
Step 3 - If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.

Step 4 - Then set 'front = front → next' and delete 'temp' (free(temp)).

Doubly Linkedlist

Definition:

A doubly linked list (DLL) is a special type of linked list in which each node contains a pointer to the previous node as well as the next node of the linked list.



Doubly Linked List

Operations:

1. Insertion: Add a new node.
2. Deletion: Remove a node.
3. Traversal: Visit each node sequentially.
4. Reverse Traversal: Move backward through nodes.

Characteristics:

1. Nodes store data and references to both next and previous nodes.

2. Supports bidirectional traversal.
3. Dynamic size: Efficiently grows or shrinks.
4. Allows for more flexible operations than singly linked lists.
5. Requires more memory due to storing both next and previous pointers.
6. Complex implementation compared to singly linked lists.

Advantage

1. Bidirectional traversal: Doubly linked lists allow traversal in both forward and reverse directions due to the presence of two pointers (previous and next) in each node.
2. Enhanced flexibility: The presence of two pointers in each node provides greater flexibility in manipulating the list.
3. Backtracking in algorithms: In certain algorithms or data structures, backtracking is essential. Doubly linked lists allow efficient backtracking as they provide easy access to the previous nodes.

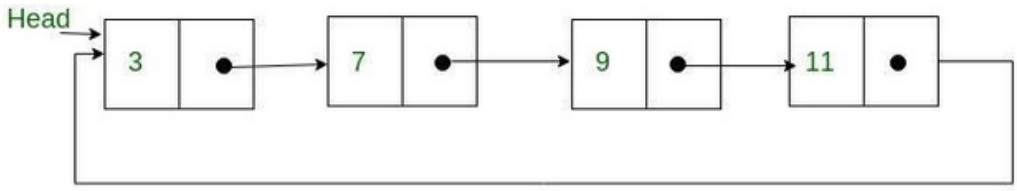
Disadvantage

1. It requires more memory to store the extra pointers.
2. It is a more complex implementation than single-linked lists.
3. It is slower traversal than arrays for random access.
4. Compared to a singly linked list, each node stores an extra pointer which consumes extra
5. memory.
   Operations require more time due to the overhead of handling extra pointers as compared to singly-linked lists

Circular Linkedlist

Definition:

A circular linked list is a variation of a linked list where the last node points to the first, forming a loop.



Operations:

- Insertion:
- Deletion:
- Traversal:
- Searching

Types:

- Singly Circular Linked List
- Doubly Circular Linked List

Characteristics:

1. Nodes have data and a next-pointer, with the last node pointing back to the first.
2. Allows continuous traversal, reaching the start node again.
3. Dynamic size like a regular linked list.
4. Efficient for certain applications with circular patterns.
5. Circular nature simplifies certain algorithms.

Advantage

- Efficient traversal: Circular linked lists can be traversed in both forward and backward directions in constant time, making them ideal for applications where bidirectional traversal is required.
- Efficient insertion and deletion: Insertion and deletion operations can also be performed in constant time at any position in the list.
- Supports Sequential Access Operations:
  Like linear linked lists, circular linked lists support sequential access, making them suitable for applications where sequential data processing is essential. Requires Less Memory Compared to
- Arrays: Circular linked lists utilize memory efficiently, especially when compared to arrays that might require pre-allocation of fixed-size memory, leading to potential wastage.

Disadvantage

- Complex implementation: Circular linked lists can be more complex to implement than other types of linked lists, such as singly linked lists.
- Memory leaks: If the pointers in a circular linked list are not managed properly, memory leaks can occur.
- Traversal can be more difficult: While traversal of a circular linked list can be efficient, it can also be more difficult than linear traversal, especially if the circular list has a complex structure.
- Lack of a natural end: The circular structure of the list can make it difficult to determine when the end of the list has been reached.

Unit 4

What is tree in data structure?

A tree is a non-linear data structure that consists of a collection of nodes in which there is a single unique path between every pair of nodes, establishing a hierarchical relationship.

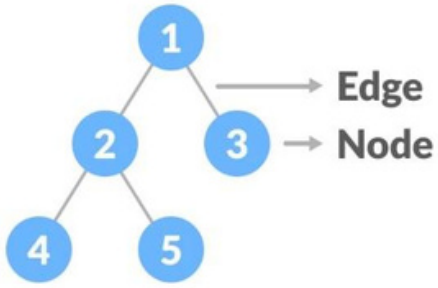
Or.

A tree Is a hierarchical, non-linear data structure where nodes are connected in a way that ensures a unique path between any pair of nodes.

➢ Operation on tree
- Insertion
- Deletion
- Traversing
- Searching

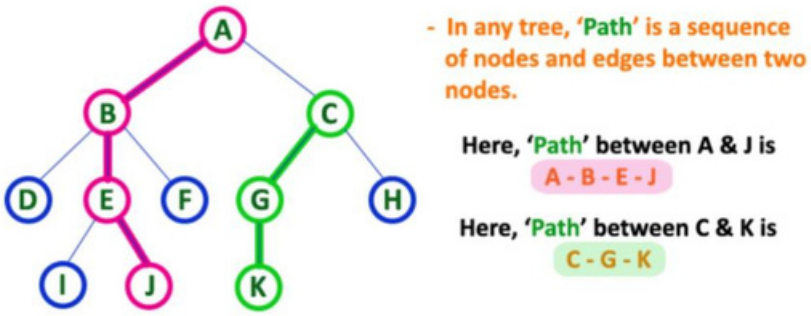
Tree Terminologies:

- Node
  A node is an entity that contains a key or value and pointers to its child nodes. The last nodes of each path are called leaf nodes or external nodes that do not contain a link/pointer to child nodes.
  The node having at least a child node is called an internal node.
- Edge
  It is the link between any two nodes.

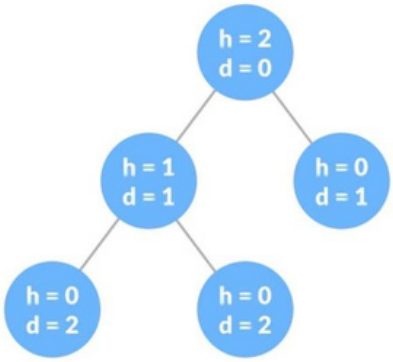


- Root
  It is the topmost node of a tree.

- Path
  The sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes.

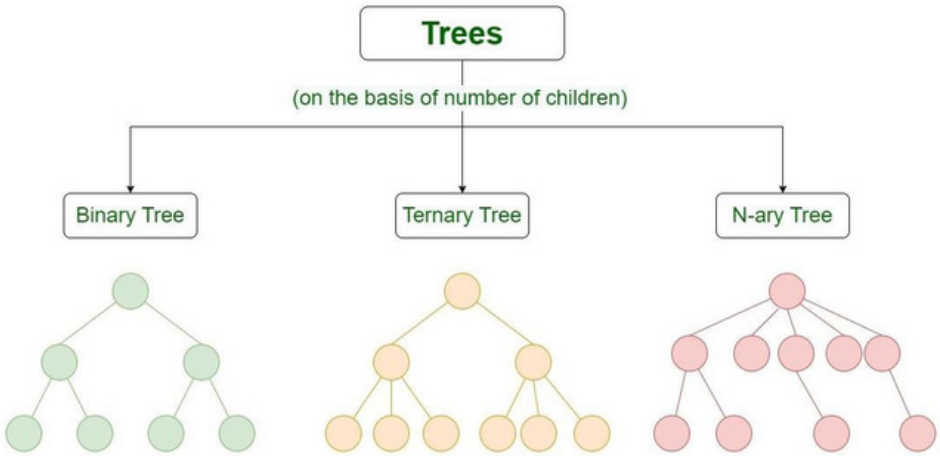


- Height of a Node
  The height of a node is the number of edges from the node to the deepest leaf (i.e. the longest path from the node to a leaf node).
- Depth of a Node
  The depth of a node is the number of edges from the root to the node.
- Height of a Tree
  The height of a Tree is the height of the root node or the depth of the deepest node.



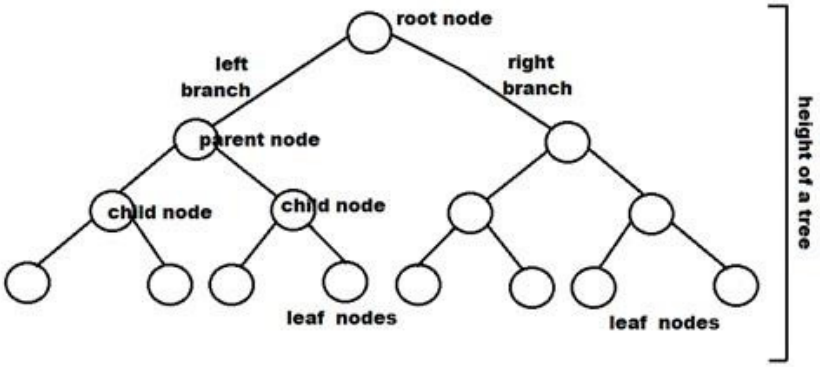
There are 3 types of 3 tree in our Diploma Syllabus

1. Binary tree (Important for Syllabus): tree with at most two child
2. Tarnery tree : a type of tree which have at most three child
3. N ary tree : Tree have end link to the child.



Binary tree:

AbinaryTree is defined as a Tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.



Memoryrepresentation ofbinarytree

Binary tree can be represent using Array and Linkedlist.

Operationonbinarytree

1. Traversal
2. Insertion
3. Deletion

Write an algorithm to insert a new note to binary tree

Step 1: Begin

Step 2: Create a new node with the given item.

Step 3: If the tree is empty, set the new node as the root and end.

Step 4: Traverse the tree to find the appropriate position for the new node:

a. If the item is less, go left.

b. If the item is greater or equal, go right.

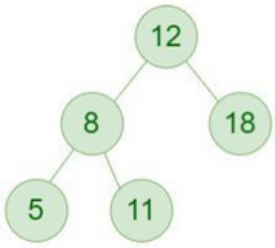
c. If the child in the chosen direction is null, insert the new node and end.

d. If the child is not null, repeat the process from that child.

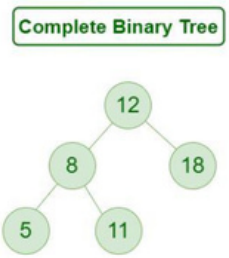
Step 5: End

Types of Binary Tree

1. Full binary tree: A Binary Tree is a full binary tree if every node has 0 or 2 children



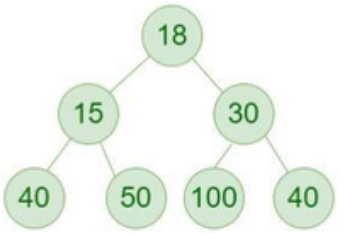
2. Complete Binary Tree: A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.



A complete binary tree is just like a full binary tree, but with two major differences:

- Every level except the last level must be completely filled.
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

3. Perfect Binary Tree: A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.



Tree Traversing

Pre-Order Traversal
In-Order Traversal
Post Order Traversal

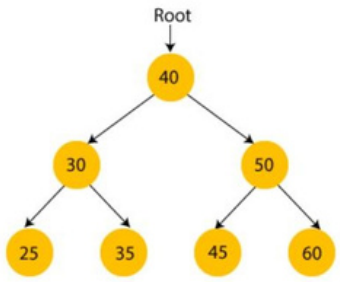
1. **Write the algorithm of Preorder Traversal**
   Step 1: Visit the root
   Step 2: Recursively visit the left sub-tree.
   Step 3: Recursively visit the rights sub-tree.
2. **Write the algorithm of In order Traversal**
   Step 1: Recursively visit Left Sub-tree
   Step 2: Visit Root
   Step 3: Recursively visit Right Sub-tree

3. **Write the algorithm for post order Traversal**
   Step 1 Recursively visit Left Sub-tree
   Step 2 Recursively visit left Right sub-tree
   Step 3 Visit Root

In semester exam you will find out to trouble will be given and you have to construct a binary tree from that.
Also there will be some question where a binary tree will be given and you have to traverse them.

Binary Search Tree (BTS)

Binary Search Tree (BST) isadata structure in which each node has at most two children. It maintains a specific order of values, nodes on the left have smaller values, nodes on the right have larger values, making it efficient for searching.



BSTs have the following properties:

- Each node in the tree has a maximum of two children.
- The value of each node is greater than all the values in its left subtree and less than all the values in its right subtree.
- The time complexity of all three of these operations is O(log n), where n is the number of nodes in the tree.

There are three main operations that can be performed on BSTs:

- Searching: Searching for a particular value in the tree.
- Insertion: Inserting a new value into the tree.
- Deletion: Deleting an existing value from the tree.

Write an algorithm to insert a new note to BST:

Step 1: Begin

Step 2: Create a new node with the given item.

Step 3: If the tree is empty, set the new node as the root and end.

Step 4: Traverse the tree to find the appropriate position for the new node:

a. If the item is less than the value of the current node, go left.

b. If the item is greater than or equal to the value, go right.

c. If the child in the chosen direction is null, insert the new node and end.

d. If the child is not null, repeat the process from that child.

Step 5: End

What is Balance factor?

The balance factor of a node is the difference between the heights of its left and right subtrees.

Abalance factor of 0 indicates that the node is perfectly balanced.

Abalance factor of 1 or -1 indicates that the node is slightly unbalanced.

Abalance factor of greater than 1 or less than -1 indicates that the node is significantly unbalanced.

Advantages of a Binary Search Tree (BST) over a Binary Tree:

1. Efficient Searching: BST maintains a sorted order, allowing for faster search operations. It has a time complexity of O(log n) for search, compared to O(n) in an unsorted binary tree.
2. Improved Data Retrieval: BSTs enable efficient retrieval of the minimum and maximum values, as they are located at the extreme left and right nodes, respectively.
3. Sorted Traversal: In-order traversal of a BST results in a sorted sequence, which is valuable for tasks such as printing data in ascending order or finding data within a specific range.
4. Efficient Insertion and Deletion: When maintaining the BST property during insertion and deletion, these operations also have an average time complexity of O(log n).

Example of binary search trees: AVL tree, Red-black tree, et.

AVL TREE

An AVL tree is a self-balancing binary search tree. It maintains its balance by ensuring that the height difference between the left and right subtrees of any node is at most 1. This helps in optimizing search, insert, and delete operations, keeping the tree relatively balanced and efficient.

Properties of AVL tree:

1. Balancing Factor: The height difference between left and right subtrees is at most 1.
2. Binary Search Tree (BST) Property: Left subtree has keys less than the node, and the right subtree has keys greater than the node.
3. Recursive Balance: All subtrees of each node are AVL trees, maintaining balance.
4. Logarithmic Height: The height of the tree is O(log n), ensuring efficient search, insert, and delete operations.
5. Dynamic Balancing: AVL trees dynamically adjust their structure after insertions and deletions to maintain balance.

Advantage of AVL tree

1. Self-balancing: AVL trees automatically maintain a balanced structure, which ensures efficient search, insertion, and deletion operations.
2. Avoids skewness: Unlike standard binary search trees, AVL trees are not prone to becoming skewed, which can significantly degrade performance.
3. Faster lookups: AVL trees typically outperform red-black trees in terms of lookup times. This is because AVL trees maintain a stricter balance, leading to shorter search paths.

4. Improved search time complexity: AVL trees have a guaranteed worst-case time complexity of O(log n) for search operations, which is superior to the O(n)complexity of unbalanced trees.
5. Height-bounded: The height of an AVL tree is always logarithmic to the number of nodes, ensuring that operations remain efficient even for large data sets.

Disadvantages of AVL Tree:

1. It is difficult to implement.
2. It has high constant factors for some of the operations.
3. Less used compared to Red-Black trees.
4. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
5. Take more processing for balancing.

Write an Algorithm to Insert a Node into an AVL Tree:

Step 1: Begin

Step 2: If the tree is empty, create a new node with the given item and set it as the root. End.

Step 3: If the tree is not empty, insert the node into the AVL tree:

a. If the item is less, go left.

i. Recursively insert the item into the left subtree.

ii. Update the height of the current node.

iii. Check balance and rotate if needed.

b. If the item is greater, go right.

i. Recursively insert the item into the right subtree.

ii. Update the height of the current node.

iii. Check balance and rotate if needed.

Step 4: Update the height of the current node.

Step 5: Check balance and rotate if needed.

Step 6: Update height and balance factors of ancestors recursively.

Step 7: End

Write an Algorithm to Search in AVL Tree

Step 1: Begin

Step 2: If the tree is empty, end the search (value not found).

Step 3: If the target value is equal to the value of the current node, end the search (value found).

Step 4: If the target value is less than the value of the current node, go left:

a. Recursively search in the left subtree.

b. If found, end the search; otherwise, continue.

Step 5: If the target value is greater than the value of the current node, go right:

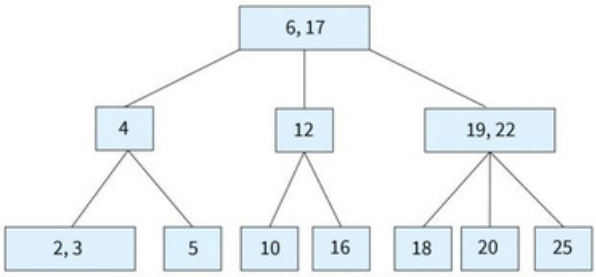
a. Recursively search in the right subtree.

b. If found, end the search; otherwise, continue.

Step 6: End (value not found in the AVL tree).

B tree

B-tree is a sort of self-balancing search tree whereby each node could have more than two children and hold multiple keys. It's a broader version of the binary search tree. It is also usually called a height-balanced m-way tree.



Properties of B tree:

1. In a B-Tree, each node has a maximum of m children.
2. Except for the root and leaf nodes, each node in a B-Tree has at least m/2 children.
3. There must be at least two root nodes.
4. The level of all the leaf nodes should be the same.

Operation on B tree:

1. Search operations
2. Insertion
3. Deletion

Time Complexity:

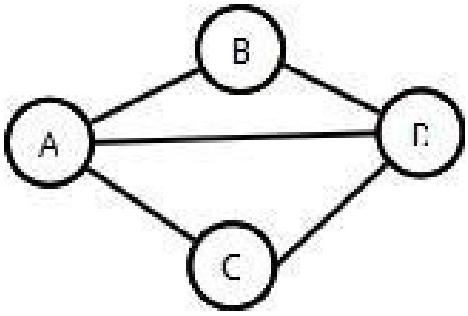
Time complexity for any of the common operations, i.e. Search, insert and delete will be O(log n).

"N" is the total number of elements in the B tree

Graphs

The graph is a no-linear data structure that consistsofvertices (V) connected through edges(E). It can be represented as G(V, E).

The graph is denoted by G(V, E).

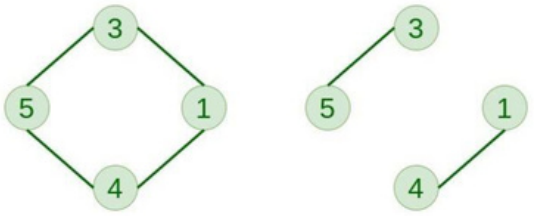


Here V = {A,B,C,D}

E = {(A,B),(A,C),(A,D),(B,D),(C,D)}

Types according to connection:

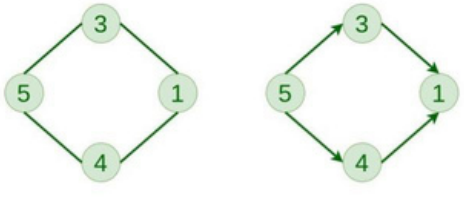
1. Connect graph: Edges are Connected to each other
2. Disconnected: Not connected to every node



Connected Graph Disconnected Graph

Types according to Direction:

1. Directed graph: A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.
2. Undirected graph: A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.

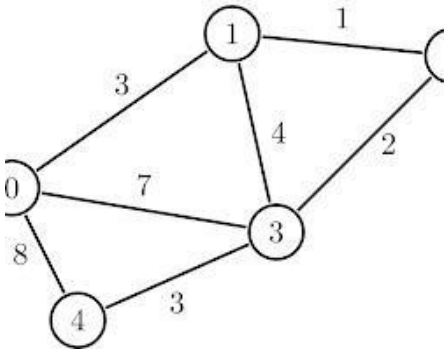


Undirected Graph Directed Graph

Basic Terminology

- Adjacent vertices: how many vertices are connected to a particular vertices.
- Path: path is a sequence of edges that connects two vertices
- Circle: A cycle is a path that starts and ends at the same vertex.
- Degree: The degree of a vertex is the number of edges that are incident to it
- Complete Graph: A complete graph is a graph in which every pair of vertices is connected by an edge.

- Weighted graph: A weighted graph is a graph in which each edge has a weight



Graph representation

There are two types of graph representation:

1. Adjacency Matrix representation
2. Adjacency List representation

There are two main types of graph traversal algorithms:

1. Depth-First Search (DFS):DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It uses a stack to keep track of the visited nodes, making it recursive in nature. DFS is often used in tasks like topological sorting and solving mazes.

2. Breadth-First Search (BFS): BFS explores a graph level by level, visiting all neighbors of a node before moving on to the next level. It uses a queue data structure, making it suitable for finding the shortest path in an unweighted graph. BFS is commonly applied in network analysis and puzzle solving.

Properties:

- DFS: Can be implemented using recursion or a stack. It may not find the shortest path.

- BFS: Utilizes a queue for traversal. It guarantees the shortest path in an unweighted graph.

Writeanalgorithm to travers using BFS

Step1:Begin BFS

Step2:Initialize an empty queue and a set to track visited vertices.

Step3:Enqueue the starting vertex and mark it as visited.

Step4:While the queue is not empty:

a.Dequeue a vertex from the queue.

b.Process the vertex (visit or do some operation).

c.Markthe dequeued vertex as visited.

d.Foreach unvisited neighbor, enqueue it and mark it as visited.

Step5:End BFS

Writeanalgorithm to travers using DFS

Step1:Begin DFS

Step2:Initialize an empty stack and a set to track visited vertices.

Step3:Push the starting vertex onto the stack and mark it as visited.

Step4:While the stack is not empty:

a.Popavertex from the stack.

b.Process the vertex (visit or do some operation).

c.Markthe popped vertex as visited.

d. For each unvisited neighbor, push it onto the stack.

Step 5: End DFS.

Important Question for Semesters

- What is a stack? What is the push() and pop() operation of the stack?
- What is overflow and underflow?
- What is a queue? What is a double-ended queue?
- Write a procedure to insert and delete in a circular queue.
- What is a priority queue? What are the applications of a priority queue?
- What are the applications of a queue and double-ended queue?
- What are the applications of a stack? What is stack top?
- What is malloc and calloc?
- What is the advantage and disadvantage of using linked list over array?
- What are the applications of singly linked list and doubly linked list?
- Write an algorithm to reverse a singly linked list.
- Write an algorithm to insert an element at the end of a singly linked list.
- Write an algorithm to delete a particular element from a singly linked list.
- What is a circular linked list? Write the application of a circular linked list.
- Write an algorithm to concatenate two singly linked lists.