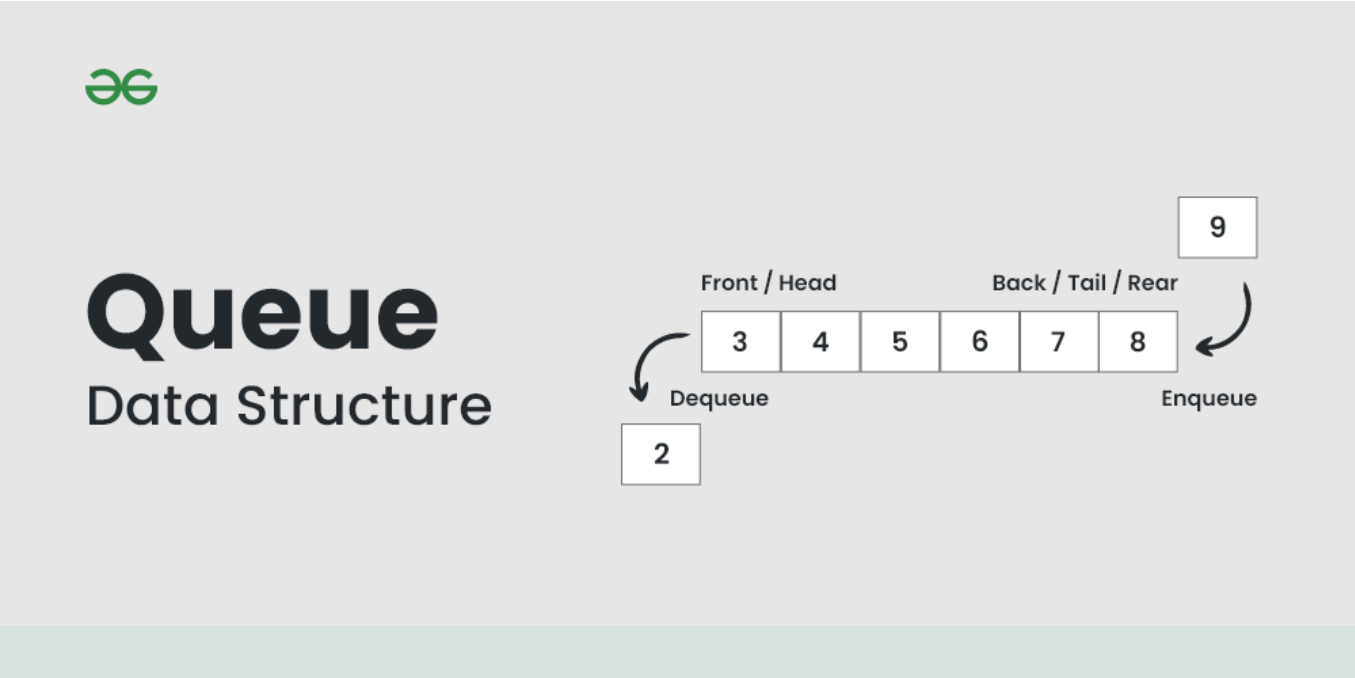
Queue Data Structure

A ***Queue Data Structure*** is a fundamental concept in computer science used for storing and managing data in a specific order. It follows the principle of "***First in, First out*" *(FIFO)***, where the first element added to the queue is the first one to be removed. Queues are commonly used in various algorithms and applications for their simplicity and efficiency in managing data flow.



Basic Operations

1. **Enqueue (Insertion):** Add an element at the rear of the queue.
2. **Dequeue (Deletion):** Remove an element from the front of the queue.
3. **Peek/Front:** Retrieve the front element without removing it.
4. **isEmpty:** Check if the queue is empty.
5. **isFull:** Check if the queue is full (only for array implementation).

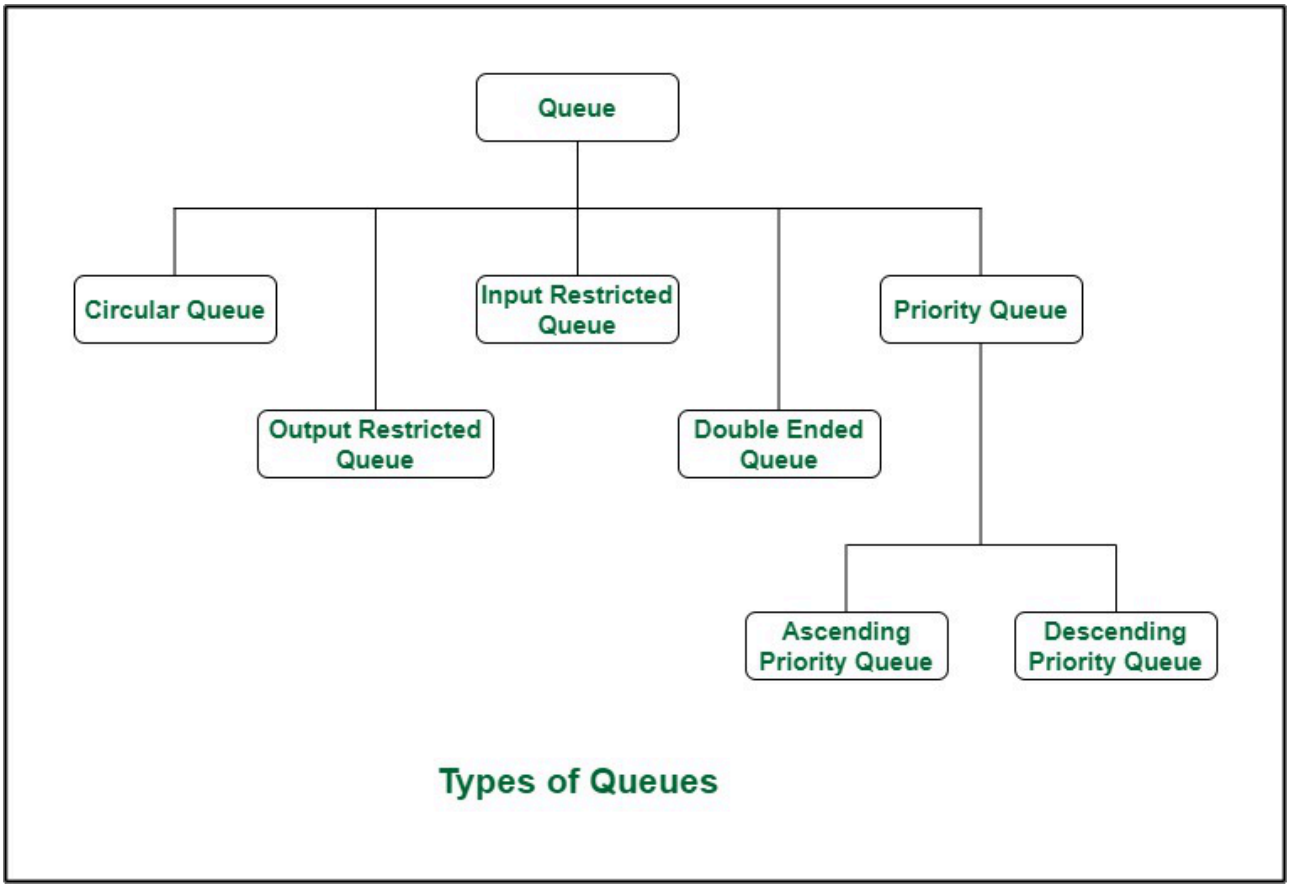
Different Types of Queues and its Applications

Types of Queues:

There are **five different types of queues** that are used in different scenarios. They are:

1. Input Restricted Queue (this is a Simple Queue)
2. Output Restricted Queue (this is also a Simple Queue)
3. Circular Queue

4. Double Ended Queue (Deque)
5. Priority Queue
   - Ascending Priority Queue
   - Descending Priority Queue



Types of Queues

1. **Simple Queue (Linear Queue):** Elements are added at the rear and removed from the front. It has a fixed size.
2. **Circular Queue:** The last position connects back to the first, efficiently utilizing storage.
3. **Priority Queue:** Each element is assigned a priority, and elements with higher priority are dequeued first.
4. **Double-Ended Queue (Deque):** Elements can be added or removed from both end.

Applications of Queues

- **CPU scheduling** in operating systems.
- **Data buffering**, e.g., I/O Buffers, Network Queues.
- **Print spooling**.

- **Breadth-First Search (BFS)** in graph traversal.
- **Simulation systems**, such as customer service or traffic systems.

Queue Implementation

1. Using Arrays

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE 5

typedef struct {
    int items[SIZE];
    int front, rear;
} Queue;

void initialize(Queue *q) {
    q->front = -1;
    q->rear = -1;
}

int isFull(Queue *q) {
    return q->rear == SIZE - 1;
}

int isEmpty(Queue *q) {
    return q->front == -1 || q->front > q->rear;
}

void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue is Full!\n");
        return;
    }
    if (q->front == -1) q->front = 0; // First element
    q->items[++q->rear] = value;
    printf("%d enqueued.\n", value);
}

int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is Empty!\n");
```

```
        return -1;
    }
    return q->items[q->front++];
}

void display(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is Empty!\n");
        return;
    }
    printf("Queue: ");
    for (int i = q->front; i <= q->rear; i++)
        printf("%d ", q->items[i]);
    printf("\n");
}

int main() {
    Queue q;
    initialize(&q);

    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    display(&q);

    printf("Dequeued: %d\n", dequeue(&q));
    display(&q);

    return 0;
}
```

2. **Using Linked List**

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node *next;
} Node;

typedef struct {
    Node *front, *rear;
} Queue;
```

```c
void initialize(Queue *q) {
    q->front = q->rear = NULL;
}

int isEmpty(Queue *q) {
    return q->front == NULL;
}

void enqueue(Queue *q, int value) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation error!\n");
        return;
    }
    newNode->data = value;
    newNode->next = NULL;

    if (q->rear == NULL) {
        q->front = q->rear = newNode;
    } else {
        q->rear->next = newNode;
        q->rear = newNode;
    }
    printf("%d enqueued.\n", value);
}

int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is Empty!\n");
        return -1;
    }
    Node *temp = q->front;
    int data = temp->data;
    q->front = q->front->next;

    if (q->front == NULL) q->rear = NULL;
    free(temp);
    return data;
}

void display(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is Empty!\n");
        return;
    }
    Node *temp = q->front;
```

```
    printf("Queue: ");
    while (temp) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    Queue q;
    initialize(&q);

    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    display(&q);

    printf("Dequeued: %d\n", dequeue(&q));
    display(&q);

    return 0;
}
```

Comparison

- **Array Implementation:** Fixed size, prone to overflow unless circular.
- **Linked List Implementation:** Dynamic size but needs more memory for pointers.

Queue Algorithm

1. Enqueue (Insert Element)

- **Input:** Queue, Element to insert.
- **Steps:**
  1. Check if the queue is full:
     - For an array: Check if `rear == SIZE - 1`.
     - For a linked list: The queue is never full unless memory is exhausted.
  2. If the queue is full:
     - Output: "Queue Overflow" (for array).

3. Else:
    - Increment the `rear` pointer.
    - Insert the element at the `rear` position (array) or create a new node and link it (linked list).
4. If it is the first element, also set the `front` pointer.

- **Output:** Success message.

---

2. Dequeue (Remove Element)

- **Input:** Queue.
- **Steps:**
    1. Check if the queue is empty:
        - For an array: Check if `front > rear` or `front == -1`.
        - For a linked list: Check if `front == NULL`.
    2. If the queue is empty:
        - Output: "Queue Underflow".
    3. Else:
        - Retrieve the element at the `front`.
        - Increment the `front` pointer (array) or move `front` to the next node (linked list).
        - If the queue becomes empty after removal, reset `front` and `rear`.
- **Output:** Dequeued element.

---

3. Peek (View Front Element)

- **Input:** Queue.
- **Steps:**
    1. Check if the queue is empty:
        - For an array: Check if `front > rear` or `front == -1`.
        - For a linked list: Check if `front == NULL`.
    2. If the queue is empty:
        - Output: "Queue is empty".
    3. Else:
        - Return the value at the `front`.

- **Output:** Front element.

---

4. isEmpty (Check if Queue is Empty)

- **Input:** Queue.
- **Steps:**
  1. For an array:
     - Check if `front > rear` or `front == -1`.
  2. For a linked list:
     - Check if `front == NULL`.
- **Output:** Boolean value (`true` or `false`).

---

5. isFull (Check if Queue is Full)

- **Input:** Queue.
- **Steps:**
  1. For an array:
     - Check if `rear == SIZE - 1`.
  2. For a linked list:
     - The queue is never full unless memory is exhausted.
- **Output:** Boolean value (`true` or `false`).

Pseudocode for Queue (Using Array)

```
Enqueue(Queue, Element):
    IF rear == SIZE - 1:
        PRINT "Queue Overflow"
        RETURN
    IF front == -1:  # First element
        front = 0
    rear = rear + 1
    Queue[rear] = Element

Dequeue(Queue):
    IF front == -1 OR front > rear:
```

```
        PRINT "Queue Underflow"
        RETURN -1
    Element = Queue[front]
    front = front + 1
    IF front > rear:  # Queue becomes empty
        front = rear = -1
    RETURN Element

Peek(Queue):
    IF front == -1 OR front > rear:
        PRINT "Queue is Empty"
        RETURN -1
    RETURN Queue[front]

isEmpty(Queue):
    RETURN front == -1 OR front > rear

isFull(Queue):
    RETURN rear == SIZE - 1
```

Linked List Data Structure

A linked list is a fundamental data structure in computer science. It mainly allows efficient insertion and deletion operations compared to arrays. Like arrays, it is also used to implement other data structures like stack, queue, and deque.

Linked List vs Arrays

Linked List:

- **Data Structure**: Non-contiguous
- **Memory Allocation**: Typically allocated one by one to individual elements
- **Insertion/Deletion**: Efficient
- **Access**: Sequential

Array:

- **Data Structure**: Contiguous
- **Memory Allocation**: Typically allocated to the whole array
- **Insertion/Deletion**: Inefficient
- **Access**: Random

Linked List:

Advantages:

- **Dynamic Size**: Can grow or shrink during execution.
- **Efficient Insertion/Deletion**: Easy to add or remove elements without shifting.

Disadvantages:

- **Memory Overhead**: Extra memory is required for pointers.
- **Sequential Access**: Cannot directly access elements like in arrays.

Basic Terminologies of Linked List

A **Linked List** is a linear data structure in which elements are not stored at a contiguous location but are linked using pointers. It forms a series of connected nodes, where each node stores the data and the address of the next node.

What is Linked List?

Node Structure

A node in a linked list typically consists of two components:

- **Data**: Holds the actual value or data associated with the node.
- **Next Pointer or Reference**: Stores the memory address (reference) of the next node in the sequence.

Head and Tail

- The **head node** points to the first node in the list and is used to access the linked list.
- The **tail node** is the last node in the list and points to `NULL` or `nullptr`, indicating the end of the list.

Why is the Linked List Data Structure Needed?

Linked lists are preferred over arrays in cases where insertion and deletion operations are frequent, as they are more efficient.

Example:

Consider maintaining a sorted list of IDs in an array:
`id[] = [1000, 1010, 1050, 2000, 2040]`.

- To **insert** a new ID `1005`, all elements after `1000` (excluding `1000`) must be shifted to maintain the sorted order.
- To **delete** `1010`, all elements after `1010` must be shifted, making deletion equally expensive.

In such scenarios, linked lists offer better efficiency as insertion and deletion only require pointer adjustments.

Operations on Linked Lists

1. Insertion

Adding a new node to a linked list involves adjusting the pointers of the existing nodes to maintain the proper sequence.
Insertion can be performed:

- At the beginning
- At the end
- At any position within the list

2. Deletion

Removing a node requires adjusting the pointers of neighboring nodes to bridge the gap left by the deleted node.
Deletion can be performed:

- At the beginning
- At the end
- At any position within the list

3. Searching

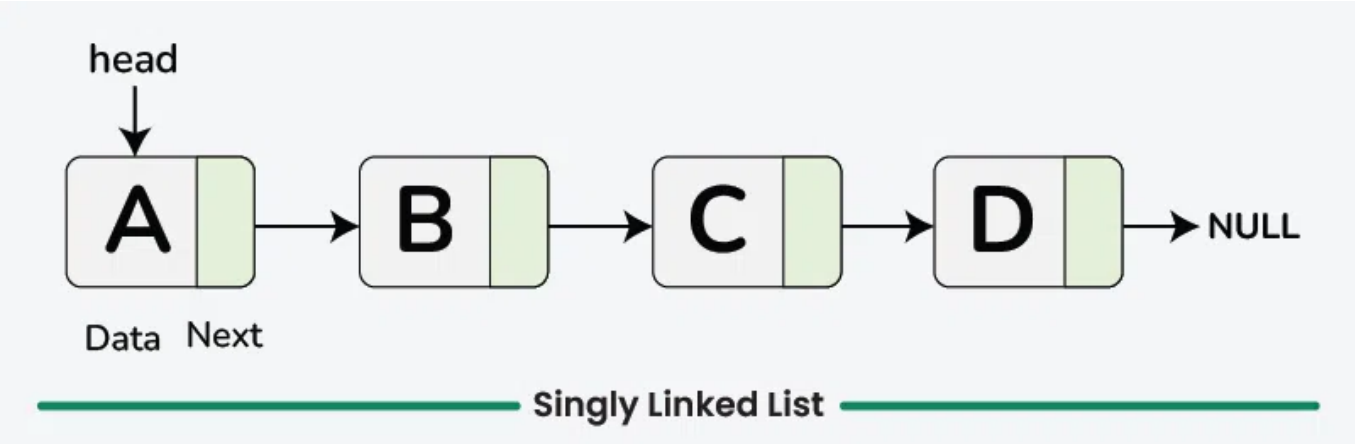
Searching for a specific value involves traversing the list from the head node until the value is found or the end of the list is reached.

Types of Linked Lists

Linked lists are categorized based on how nodes are connected and how data can be traversed. The main types of linked lists are:

1. Singly Linked List

A **Singly Linked List** is a type of linked list where each node points to the next node in the sequence. The last node points to `NULL`, indicating the end of the list.



Example Code:

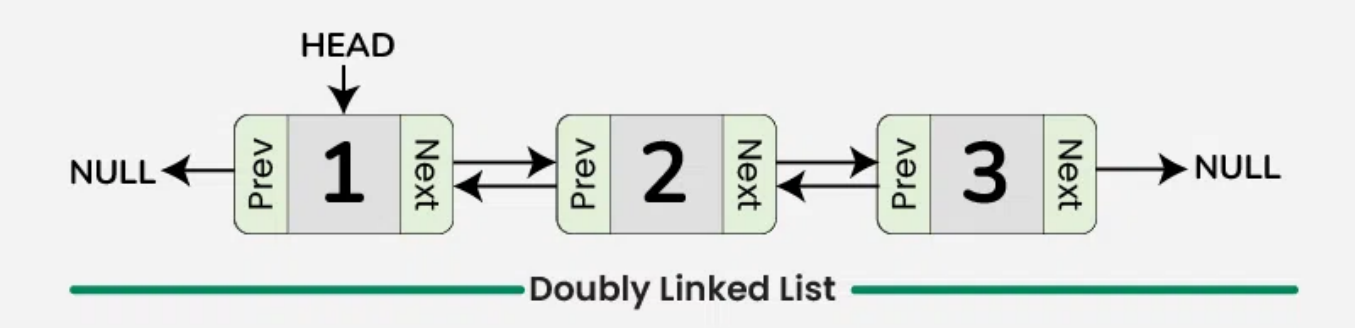
```c
#include <stdio.h>
#include <stdlib.h>

// Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to print the linked list
void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
```

```c
}

int main() {
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // Allocate nodes
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    head->data = 1;
    head->next = second;

    second->data = 2;
    second->next = third;

    third->data = 3;
    third->next = NULL;

    printList(head);
    return 0;
}
```

---

2. Doubly Linked List

A **Doubly Linked List** is a type of linked list where each node contains pointers to both its previous and next nodes. This allows traversal in both directions.



Example Code:

```c
#include <stdio.h>
#include <stdlib.h>
```

```c
// Node structure
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Function to print the linked list
void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d <-> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // Allocate nodes
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    head->data = 1;
    head->prev = NULL;
    head->next = second;

    second->data = 2;
    second->prev = head;
    second->next = third;

    third->data = 3;
    third->prev = second;
    third->next = NULL;

    printList(head);
    return 0;
}
```

3. Circular Linked List

A **Circular Linked List** is a type of linked list where the last node points back to the first node, forming a circular chain. It can be either singly or doubly linked.

Example Code:

```c
#include <stdio.h>
#include <stdlib.h>

// Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to print the circular linked list
void printList(struct Node* head) {
    struct Node* temp = head;
    if (head != NULL) {
        do {
            printf("%d -> ", temp->data);
            temp = temp->next;
        } while (temp != head);
    }
    printf("HEAD\n");
}

int main() {
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // Allocate nodes
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));
```

```
    head->data = 1;
    head->next = second;

    second->data = 2;
    second->next = third;

    third->data = 3;
    third->next = head;  // Circular link

    printList(head);
    return 0;
}
```

Summary of Differences:

Type	Traversal Direction	Last Node Points To
Singly Linked List	Forward Only	`NULL`
Doubly Linked List	Forward & Backward	`NULL`
Circular Linked List	Forward/Backward	First Node

Applications of Linked List

Linked lists are versatile data structures that are widely used in various applications due to their dynamic nature and efficient insertion and deletion capabilities. Below are some of the common applications of linked lists:

1. Dynamic Memory Allocation

Linked lists are used to manage dynamic memory in operating systems and other software applications, where memory blocks are linked together and allocated as needed.

2. Implementing Other Data Structures

Linked lists are the building blocks for various data structures, such as:

- **Stacks**: Implemented using singly linked lists.

- **Queues**: Implemented using singly or doubly linked lists.
- **Deque (Double-ended queue)**: Often implemented using doubly linked lists.
- **Graphs**: Used to represent adjacency lists in graph structures.

3. Real-time Applications

- **Music Playlist**: Tracks are linked such that moving to the next or previous song is easy.
- **Image Viewer**: Images are linked for forward and backward navigation.
- **Undo Feature in Applications**: Doubly linked lists are used to store states for undo/redo operations.

4. Dynamic Data Management

Linked lists are ideal for applications requiring frequent addition and deletion of data, such as:

- **Database Management Systems**: Managing data entries dynamically.
- **Text Editors**: Handling lines of text dynamically.

5. Polynomial Representation

Linked lists are used to represent polynomials, where each node contains the coefficient and exponent of a term, and the terms are connected in sequence.

Example:

Representation of `4x^3 + 3x^2 + 2x + 1` using a linked list:

```
[4, 3] -> [3, 2] -> [2, 1] -> [1, 0] -> NULL
```

6. Memory-efficient Storage

For applications where memory is a constraint, linked lists are used instead of arrays because they do not require contiguous memory allocation.

7. File Systems

Many operating systems use linked lists to manage directories and files. For example:

- **FAT (File Allocation Table)** uses a linked list to keep track of file blocks on disk.

8. Network Packet Management

In networking, packets of data can be stored in linked lists, allowing for easy reordering and management.

9. Hash Tables

Linked lists are often used to handle collisions in hash tables by chaining.

Summary Table of Applications:

Application Area	Type of Linked List Used	Use Case
Dynamic Memory Allocation	Singly/Doubly	Managing memory blocks
Data Structures	Singly/Doubly/Circular	Stacks, Queues, Deques
File Systems	Singly/Doubly	Directory and file management
Polynomial Representation	Singly	Representing polynomial terms
Undo/Redo Operations	Doubly	Navigation through history states
Networking	Singly	Storing and reordering packets

Unit-IV: Trees

1. Preliminaries

A **tree** is a non-linear data structure with hierarchical relationships between its elements, called nodes.

- **Root**: The topmost node.
- **Child**: Nodes connected to a parent node.
- **Leaf**: Nodes with no children.
- **Height**: Length of the longest path from the root to a leaf.

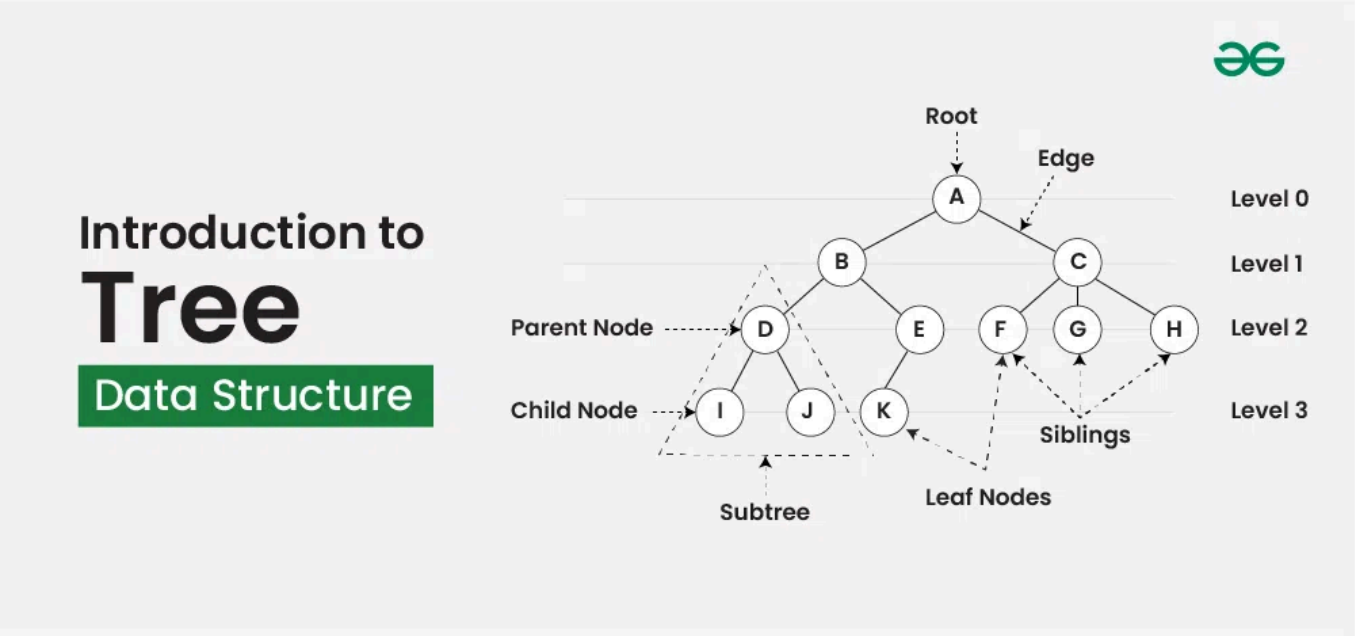
---

2. Trees

A **tree** is a collection of nodes connected by edges. It is used to represent hierarchical data like file systems, organization charts, etc.

Tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the ***root***, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.



Basic Terminologies in Tree Data Structure

In the tree data structure, several terms are used to describe the properties and relationships between nodes. Below is a comprehensive list of the basic terminologies with their definitions and examples.

---

1. Parent Node

The node that is an immediate predecessor of a node is called the parent node.

- **Example**: `{B}` is the parent node of `{D, E}`.

---

2. Child Node

The node that is the immediate successor of a node is called the child node.

- **Example**: `{D, E}` are the child nodes of `{B}`.

---

3. Root Node

The topmost node of a tree or the node that does not have any parent node is called the root node.

- **Example**: `{A}` is the root node of the tree.
- A non-empty tree must contain exactly one root node, and there must be exactly one path from the root to every other node in the tree.

---

4. Leaf Node (External Node)

The nodes that do not have any child nodes are called leaf nodes.

- **Example**: `{I, J, K, F, G, H}` are the leaf nodes of the tree.

---

5. Ancestor of a Node

Any predecessor node on the path from the root to that node is called an ancestor of the node.

- **Example**: `{A, B}` are the ancestor nodes of `{E}`.

---

6. Descendant

A node `x` is a descendant of another node `y` if and only if `y` is an ancestor of `x`.

---

7. Sibling

Nodes that share the same parent are called siblings.

- **Example**: `{D, E}` are siblings as they share the same parent `{B}`.

8. Level of a Node

The level of a node is defined as the number of edges on the path from the root node to the node.

- **Example**: The root node has level `0`, and its immediate children have level `1`.

---

9. Internal Node

A node with at least one child is called an internal node.

---

10. Neighbor of a Node

The parent or child nodes of a node are called its neighbors.

---

11. Subtree

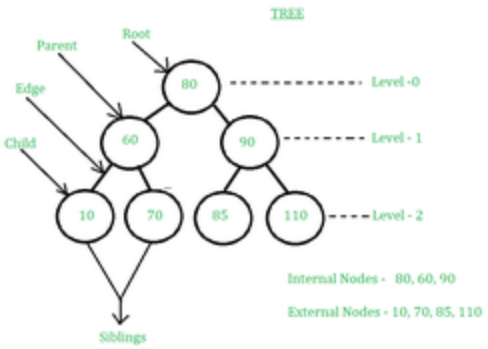
A subtree consists of any node in the tree along with all its descendants.

Visual Representation of a Tree Example:

Key Examples Based on the Above Tree:

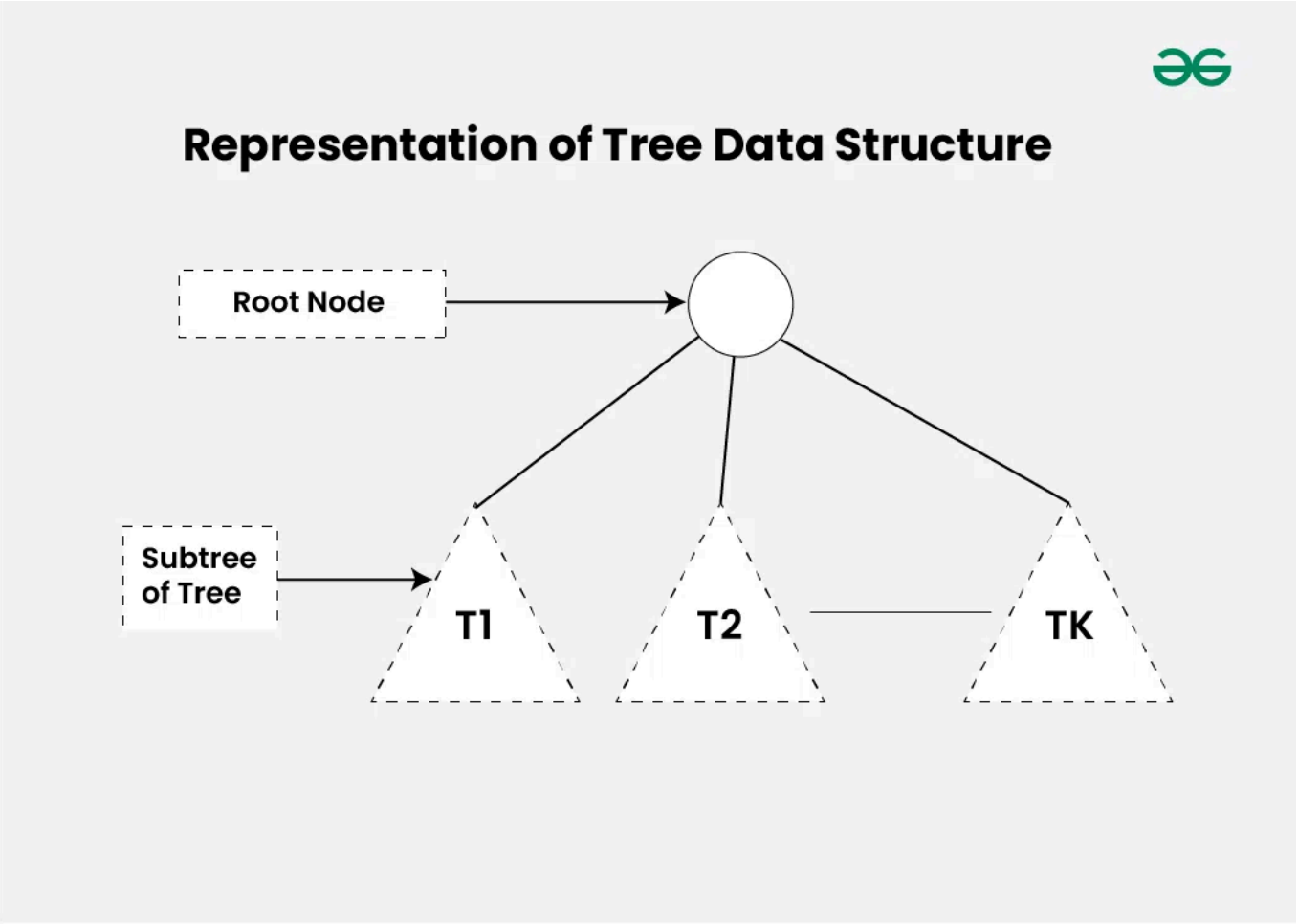
- **Parent Node**: `{B}` is the parent of `{D, E}`.

- **Child Node**: `{D, E}` are the children of `{B}`.
- **Root Node**: `{A}` is the root node.
- **Leaf Nodes**: `{G, H, I, J, F}`.
- **Internal Node**: `{B, C, D}`.
- **Siblings**: `{D, E}`.
- **Level of Node**: Level of `{A}` is `0`, `{B, C}` is `1`, and `{G, H, I, J}` is `3`.
- **Subtree**: Subtree of `{B}` contains `{B, D, E, G, H}`.



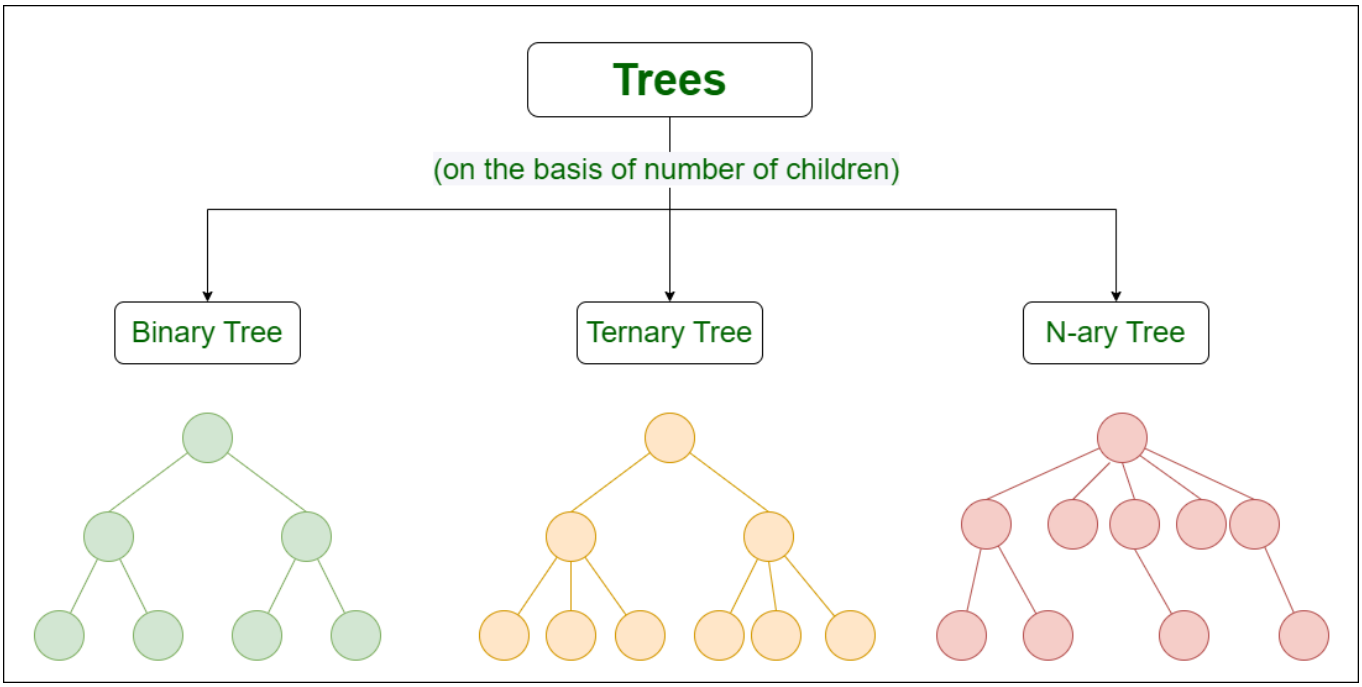
Representation of Tree Data Structure:

A tree consists of a root node, and zero or more subtrees T1, T2, … , Tk such that there is an edge from the root node of the tree to the root node of each subtree. Subtree of a node X consists of all the nodes which have node X as the ancestor node.




```
struct Node {
    int data;
    struct Node* first_child;
    struct Node* second_child;
    struct Node* third_child;
    .
    .
    .
    struct Node* nth_child;
};
```

Types of Trees in Data Structure based on the number of children:

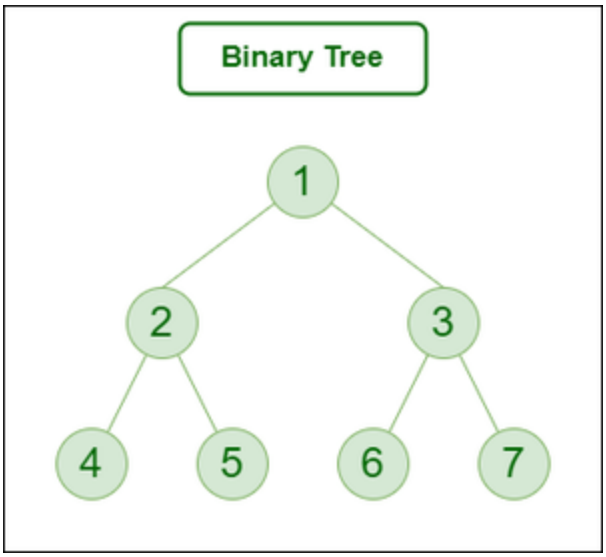


1. Binary Tree

A binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Example:

Consider the tree below. Since each node of this tree has only 2 children, it can be said that this tree is a Binary Tree

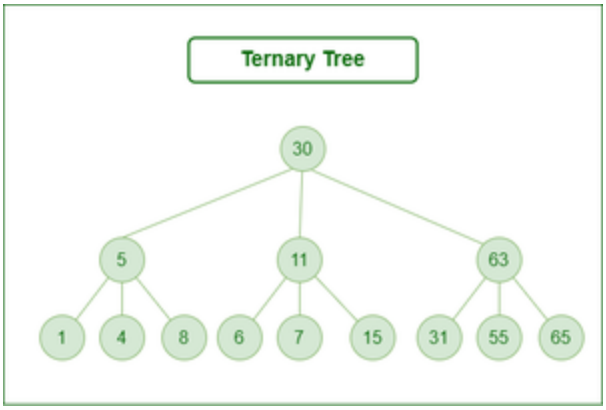


2. Ternary Tree

A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.

Example:

Consider the tree below. Since each node of this tree has only 3 children, it can be said that this tree is a Ternary Tree



Types of Ternary Tree:

Ternary Search Tree
A ternary search tree is a special trie data structure where the child nodes of a standard trie are ordered as a binary search tree.

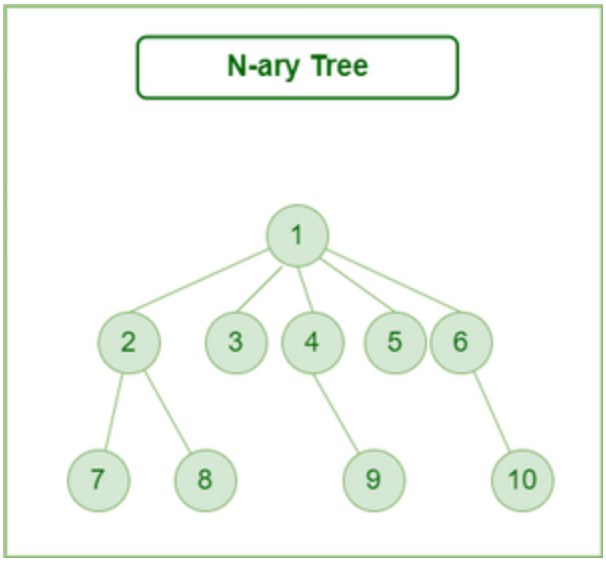
Unlike trie(standard) data structure where each node contains 26 pointers for its children, each node in a ternary search tree contains only 3 pointers:

1. The left pointer points to the node whose value is less than the value in the current node.
2. The equal pointer points to the node whose value is equal to the value in the current node.
3. The right pointer points to the node whose value is greater than the value in the current node.

3. N-ary Tree (Generic Tree)

Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.
Every node stores the addresses of its children and the very first node's address will be stored in a separate pointer called root.

1. Many children at every node.
2. The number of nodes for each node is not known in advance.



3. Forest

A **forest** is a collection of disjoint trees. It can be represented as multiple trees where each tree acts independently.

---

4. Binary Trees

A **binary tree** is a tree where each node has at most two children, referred to as the left child and right child.

Example Code:

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
```

```
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Preorder traversal
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Preorder Traversal: ");
    preorder(root);

    return 0;
}
```

---

5. Binary Search Tree (BST)

A **BST** is a binary tree where each node's left child contains values smaller than the node, and the right child contains values larger than the node.

Example Code:

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
```

```c
// Create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Insert a node in BST
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) return createNode(data);
    if (data < root->data) root->left = insert(root->left, data);
    else if (data > root->data) root->right = insert(root->right, data);
    return root;
}

// Inorder traversal
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);

    printf("Inorder Traversal of BST: ");
    inorder(root);

    return 0;
}
```

---

6. Conversion of Forest to Binary Tree

A **forest** can be represented as a binary tree by linking trees horizontally through the right child and vertically through the left child.

---

7. AVL Trees

An **AVL tree** is a self-balancing binary search tree. For every node, the height difference between its left and right subtrees is at most 1.

---

8. Tree Traversals

Tree traversal refers to visiting each node in a tree.

1. **Preorder**: Root -> Left -> Right
2. **Inorder**: Left -> Root -> Right
3. **Postorder**: Left -> Right -> Root

Example Code for Traversals:

```
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}
```

---

9. Priority Queues (Heaps)

A **priority queue** is an abstract data type where each element has a priority. **Heaps** are used to implement priority queues efficiently.

- **Binary Heap**: A complete binary tree satisfying the heap property:
  - **Max-Heap**: Parent node is always larger than its children.
  - **Min-Heap**: Parent node is always smaller than its children.

Example of a Binary Heap (Insertion):

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

int heap[MAX];
int size = 0;

// Insert into heap
void insert(int value) {
    int i = size;
    heap[size++] = value;

    while (i > 0 && heap[(i - 1) / 2] < heap[i]) {
        int temp = heap[i];
        heap[i] = heap[(i - 1) / 2];
        heap[(i - 1) / 2] = temp;
        i = (i - 1) / 2;
    }
}

// Print the heap
void printHeap() {
    for (int i = 0; i < size; i++) {
        printf("%d ", heap[i]);
    }
    printf("\n");
}

int main() {
    insert(10);
    insert(20);
    insert(5);
    insert(30);
    insert(15);
```

```
    printf("Heap: ");
    printHeap();
    return 0;
}
```

This structured explanation covers all major topics with examples in C. Let me know if you'd like to focus on any specific section!