DSA Fundamentals and Stack Tutorial

1. DSA Fundamentals

What is DSA?

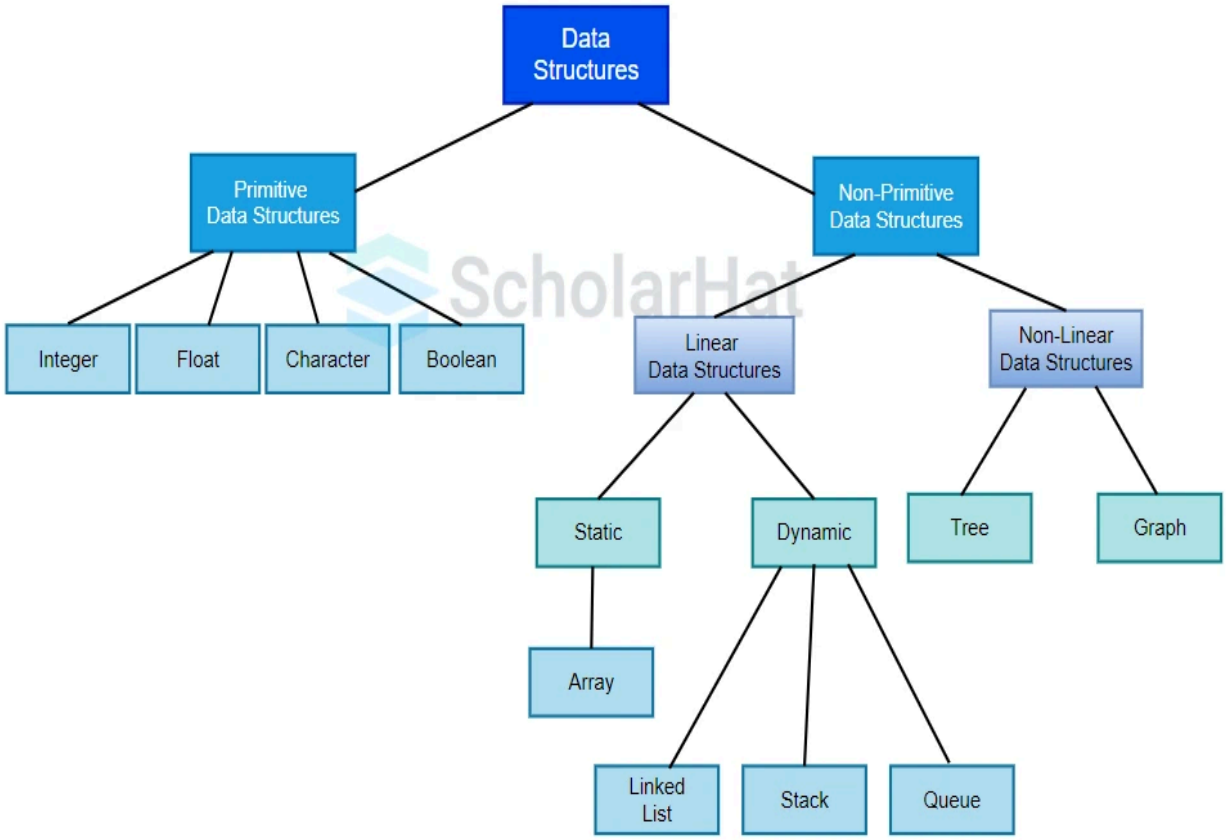
Data Structures and Algorithms (DSA) are the building blocks of computer science.

- **Data Structures:** Techniques to **organize** and **store** data **efficiently**.
- **Algorithms:** Step-by-step **procedures** to perform tasks or solve problems.

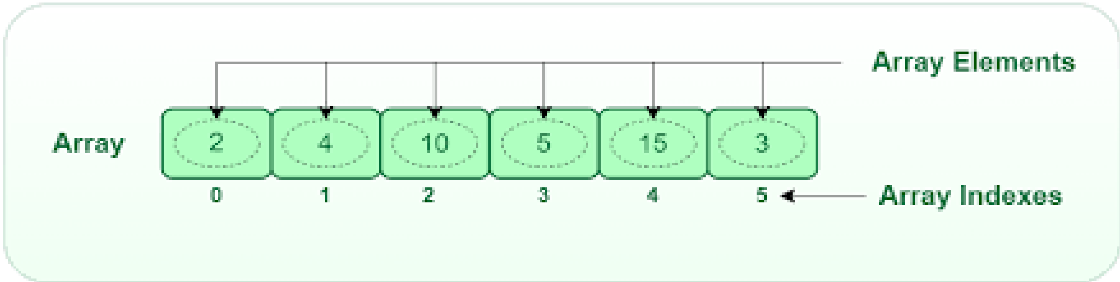
Why Learn DSA?

1. **Problem Solving:** Enhances logical thinking and approach to problem-solving.
2. **Optimized Solutions:** Helps design efficient algorithms to save **time** and **memory**.
3. **Job Interviews:** Most coding interviews heavily focus on DSA.

Type of DataStructure



1. **Linear Data Structures:** store data in a linear way!
    - **Array:** Fixed-size structure to store elements of the same type.

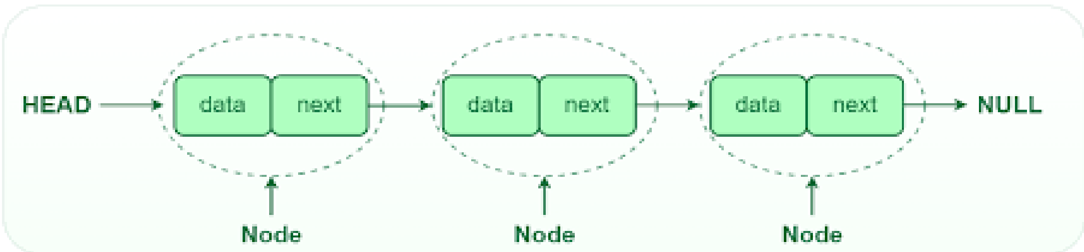


Example : User Photos or Video

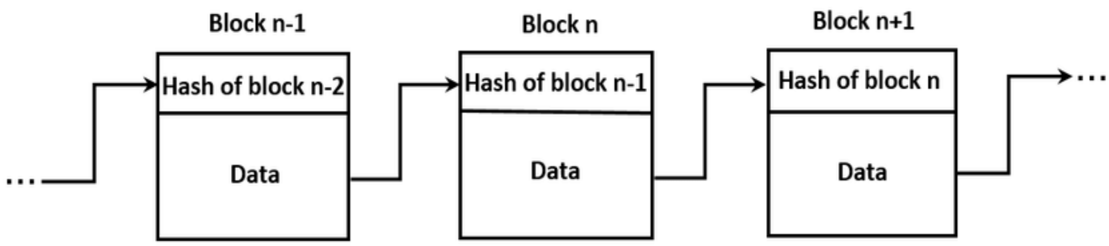
Use Case: Storing **Photos** or **Videos** uploaded by users. On Instagram or Snapchat, stories or images are stored in arrays for easy retrieval and **display** in the **sequence** they were **uploaded**.

- **Linked List:** A sequence of elements, where each element points to the next.

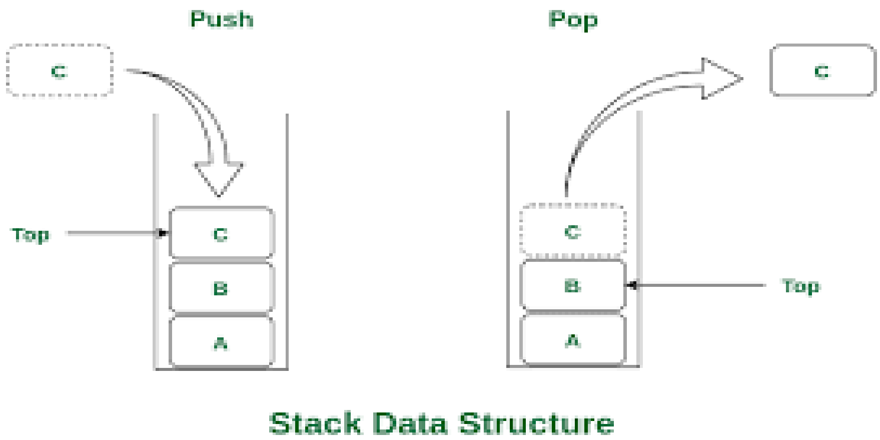
Pointer use for travel and access nodes of list.



Example : Bitcoin blockchain

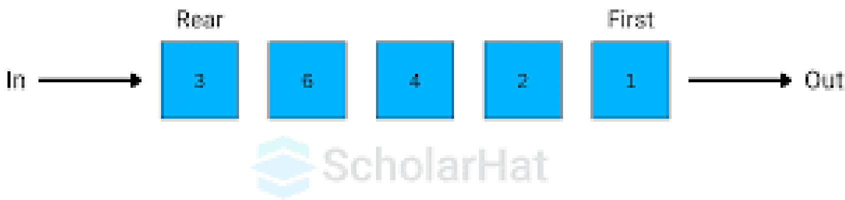


- **Stack:** Follows Last In, First Out (LIFO) principle.



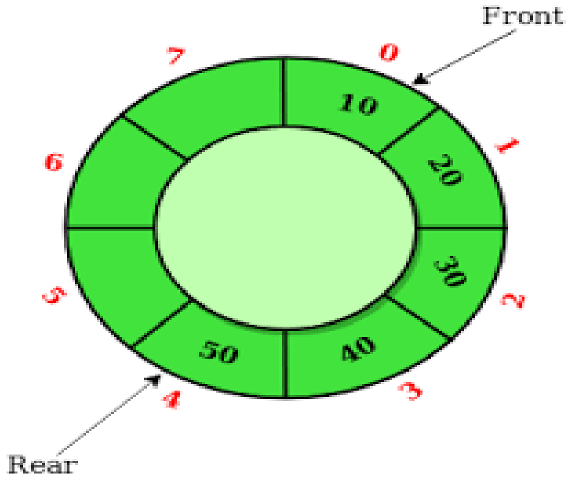
Example : **Browser History, Function Call Stack in Programming etc..**

- **Queue:** Follows First In, First Out (FIFO) principle.



Example : standing for Lunch

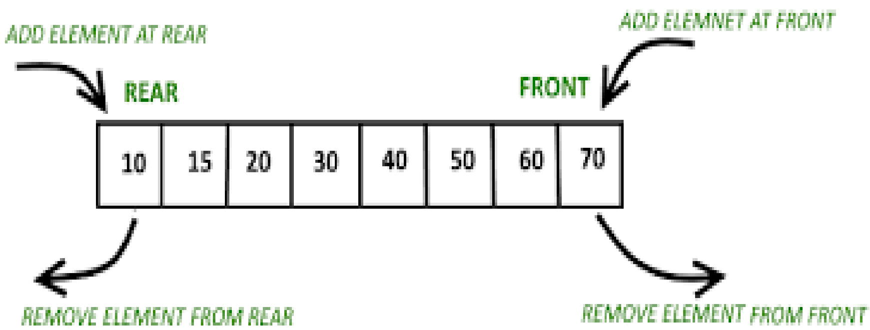
■ **Circular Queue:** A queue where the last position connects back to the first.



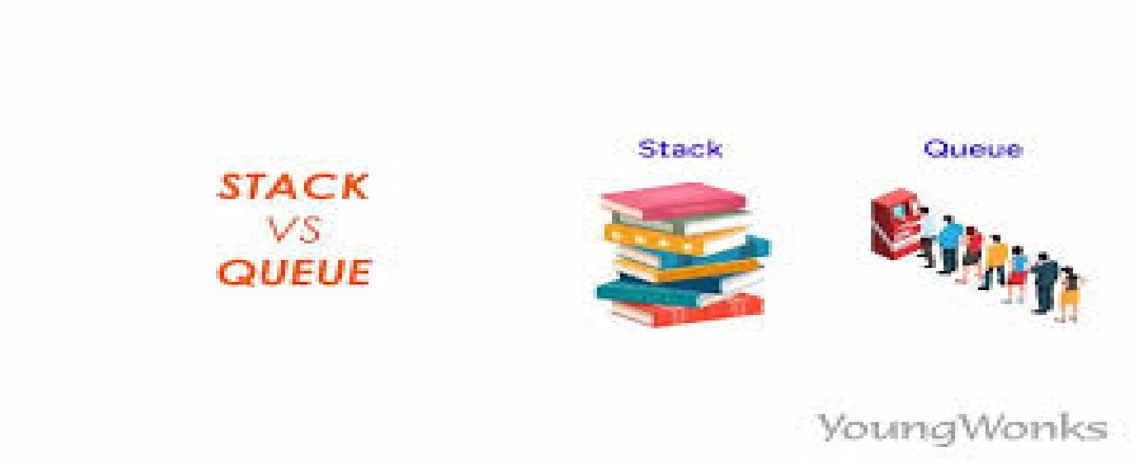
Example : Music Playlists (Repeat Mode)

Use Case: Playing songs in a loop.

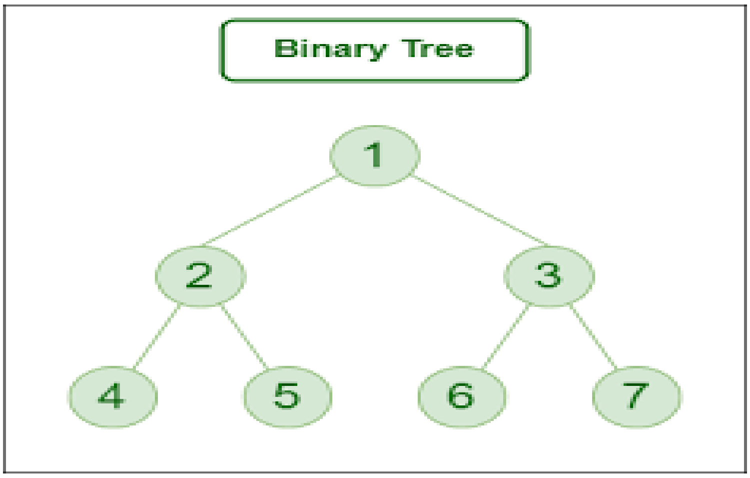
■ **Deque (Double-Ended Queue):** Allows insertion and deletion from both ends.



Example : **Operating process**



2. **Non-Linear Data Structures:**
   - **Tree:** Hierarchical structure with a root node and child nodes (e.g., Binary Tree, Binary Search Tree, AVL Tree).
   - Trees don't have a loop.



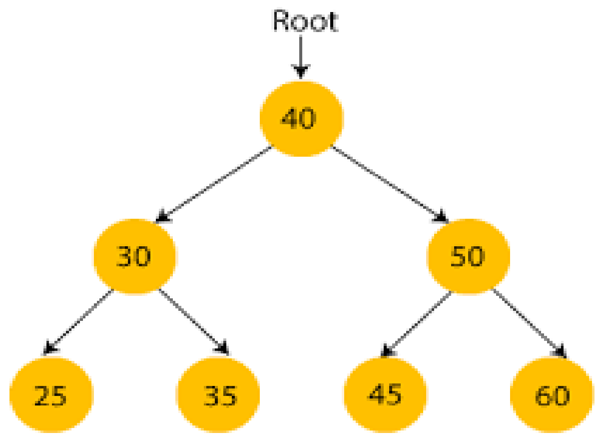
Example : **Blood Relations Tree**, **Organizational Staff Tree**.

1.Binary Tree: every node has at max two childrens.

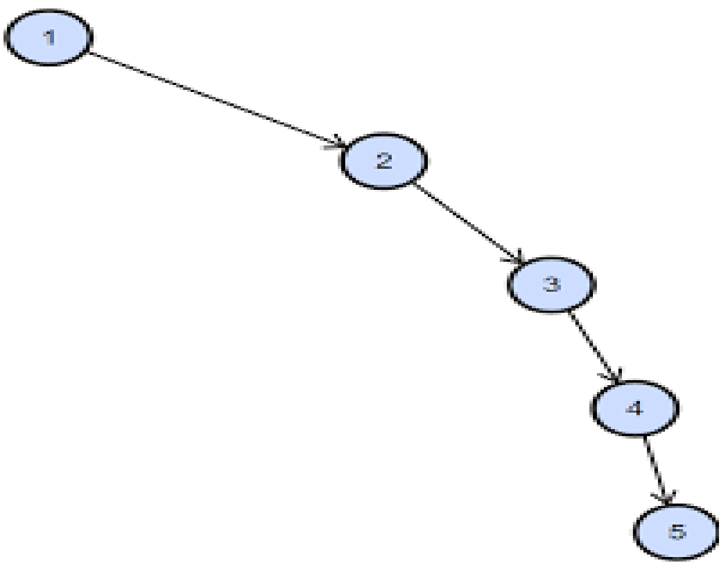
Example : the tree above is a binary tree.

2.Binary Search Tree : it must be a binary tree + every node value is greater than its left subtree and lesser or equal than right subtree.

Note : ***''used to search fast''***. In the best case and average case we need to compare ***h*** time, h is height of tree.



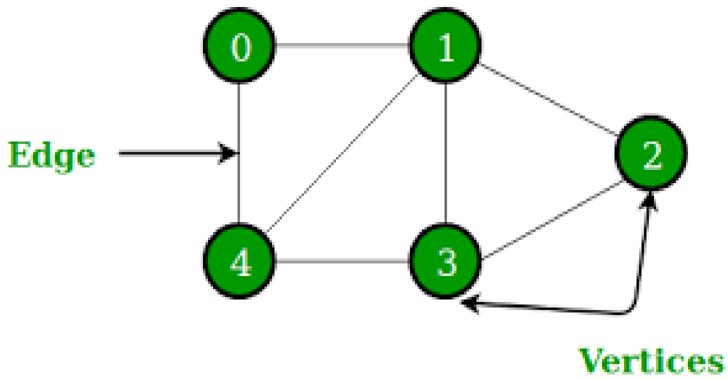
Note : *'' it can be unbalance!''*



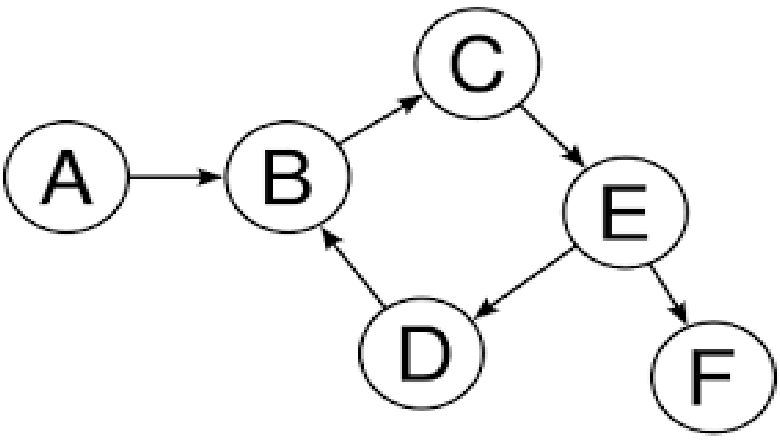
Note : ''AVL tree(Balance tree) alway be balance''

- **Graph:** A collection of nodes connected by edges (e.g., Directed, Undirected, Weighted).
- ''*Every tree is also called a graph. But reverse may not be true.*''
- Graphs can have loop and multi edge also.
- Example : **google map, network etc..**

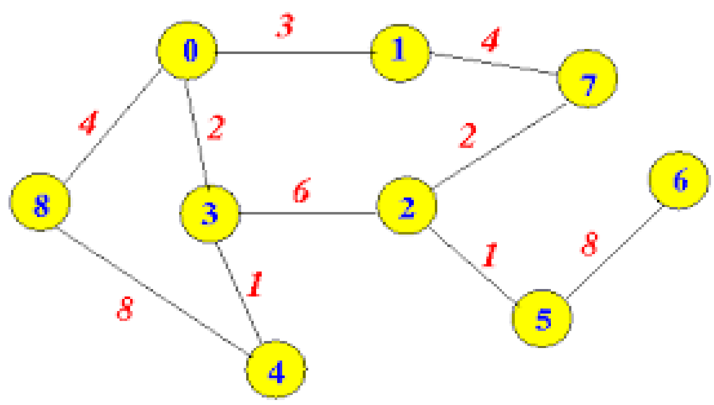
Undirected Graph



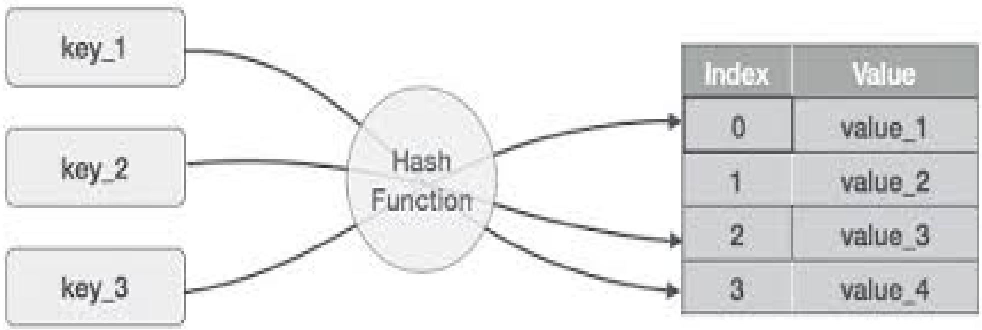
Directed Graph



Weighted Graph



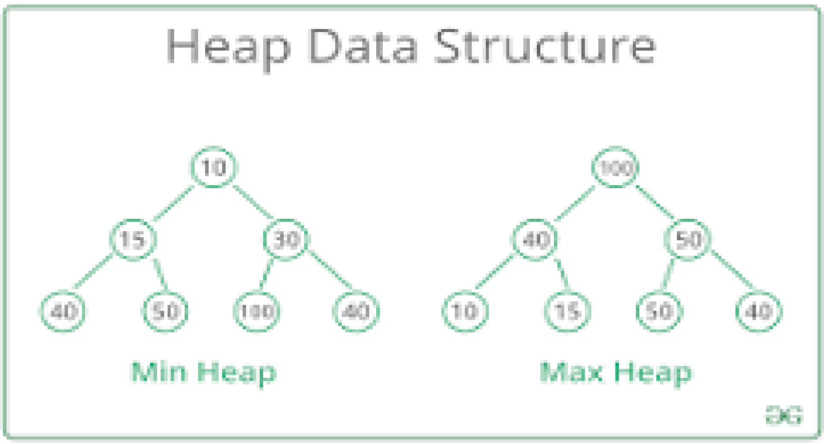
3. **Hash-Based Data Structures:**
   - **Hash Table:** Provides efficient key-value pair storage and retrieval using hash functions.



Note: "***time complexity to search any data is averagely constant time.***"

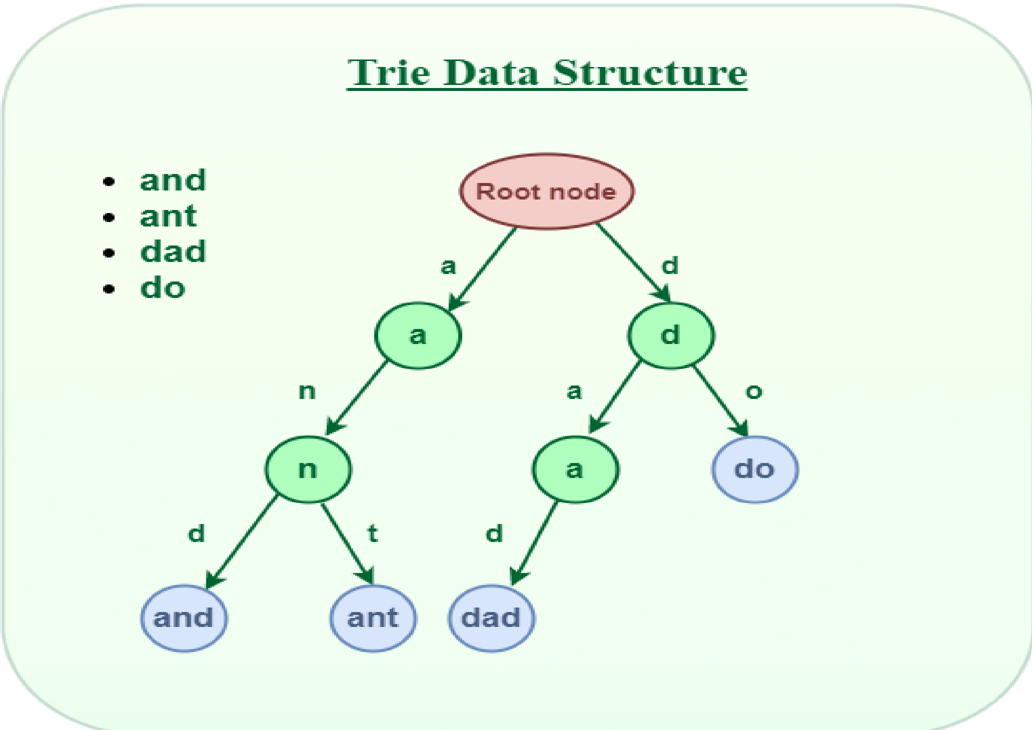
Example **: cache frequently used data**

4. **Specialized Data Structures:**
   - **Heap:** A complete binary tree used for priority-based operations.
   - MinHeap : nodes have a min value then their children.
   - MaxHeap : nodes have a max value then their children.



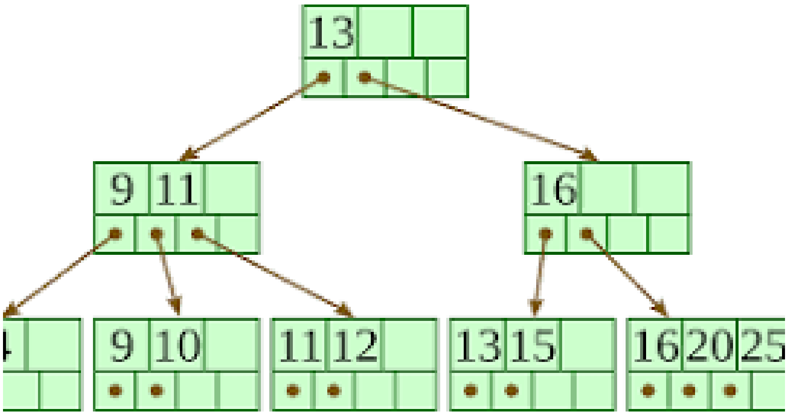
Example : **Real-Time Task Scheduler**

   - **Trie:** Used for efficient **retrieval of strings & storage**, especially in dictionaries.



Example : **Prefix Search**, **Autocompleter features**,**Search Engines (google).**

5. **Advanced Data Structures:**
   - **B-Tree and B+ Tree:** Used in **databases** for optimized storage and retrieval. (**balance tree**).
   - Store data in **ordered** form.



1. Stack

- **LIFO (last in first out)**

How to Use a Stack?

Operations commonly performed on a stack:

1. **Push:** Add an element to the top of the stack.
2. **Pop:** Remove the top element from the stack.
3. **Peek/Top:** View the top element without removing it.
4. **isEmpty:** Check if the stack is empty.

Properties

1. Follows **LIFO** order.
2. Dynamic memory allocation (if implemented using dynamic structures).
3. Can be implemented using **arrays** or **linked lists**.

Time Complexity of Operations

- **Push:** O(1)
- **Pop:** O(1)
- **Peek:** O(1)
- **isEmpty:** O(1)

Implementation of Stack in C++

```
Int MAX = 100;
class Stack {
private:
    int top;
    int arr[MAX];

public:
    Stack() { top = -1; }

    void push(int x) {
        if (top >= MAX - 1) return;
```

```
        arr[top] = x;
         top++;
    }

    int pop() {
        if (top < 0) return -1;
        return arr[top];
         top–;
    }

    int peek() {
        if (top < 0) return -1;
        return arr[top];
    }

    bool isEmpty() {
        return top < 0;
    }
};

int main() {
    Stack s;
    s.push(10);
    s.push(20);
    s.push(30);

    cout << s.pop() << endl;
    cout << s.peek() << endl;
    cout << (s.isEmpty() ? "Yes" : "No") << endl;

    return 0;
}
```