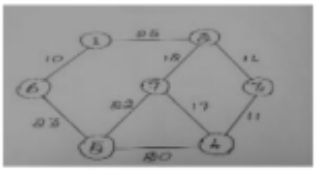
Scheme of Evaluation

Internal Assessment Test 3 -August 2025

Sub:	Data Structures and Algorithms							Sub Code:	MMC203
Date:	13-08-25	Duration:	90 mins	Max Marks:	50	Sem:	II	Branch:	MCA

Q.NO	Description	Marks Distribution	Max Marks
1	What is backtracking? Apply backtracking to solve the below instance of sum of subset problem S={5,10,12,13,15,18} d=30 • Find the solution	[10]	10
2	Describe about the B trees in detail notes • Explanation of B trees with neat diagram	[10]	10
3	Describe about the AVL Tree in detail notes • Explanation of AVL trees with neat diagram	[10]	10
4	Define dynamic programming. How is it different from recursion? Explain with any two example • Definition of dynamic programming • Difference between recursion and dynamic programming • Explanation of any two examples	[2] [3] [4]	10
5	Explain about the Trie and its types with neat example • Explanation of Trie with neat diagram • Explanation of types of Trie	[5] [5]	10
6	Apply Prims algorithm to the following given graph to construct minimum spanning tree  • Find out the solution of the given graph	[10]	10
7	Describe real-world applications of advanced data structures • Explanation of real-world applications of advanced data structures	[10]	10

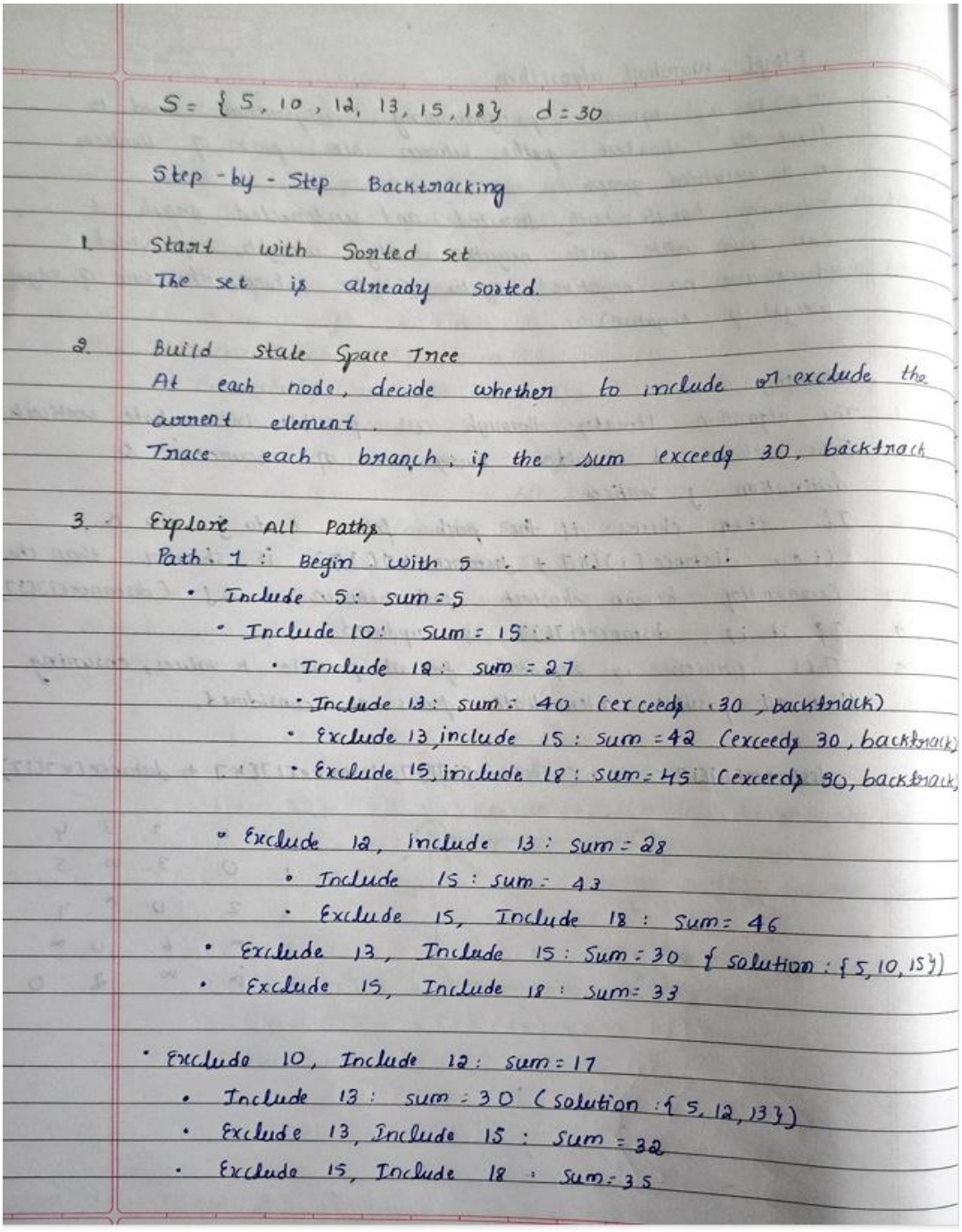
8	Define a segment tree. How is it used for range minimum/maximum/sum queries? • Definition of Segment tree • Explanation for range minimum/maximum/sum queries used in segment tree.	**[2]** **[8]**	**10**
9	What is a Fenwick Tree (Binary Indexed Tree)?How does it differ from a segment tree • Definition of Segment tree • Explain the difference from segment tree	**[2]** **[8]**	**10**
10	Considerthefollowingweightedgraph:Vertices:{1,2,3,4}Edges:1→2(weight2) 1 →3(weight4)2→ 3(weight1)2→4 (weight7)3 →4(weight 3).Apply Floyd-Warshall algorithm and give the final shortest path matrix. • Find out the solution of the given graph	**[10]**	**10**

Internal Assessment Test 3 – August 2025

Data Structures and Algorithms							Sub Code:	MMC203
13/08/2025	Duration:	90 min's	Max Marks:	50	Sem:	II	Branch:	MCA

PART I

1. **What is backtracking? Apply backtracking to solve the below instance of sum of subset problem S={5,10,12,13,15,18} d=30**



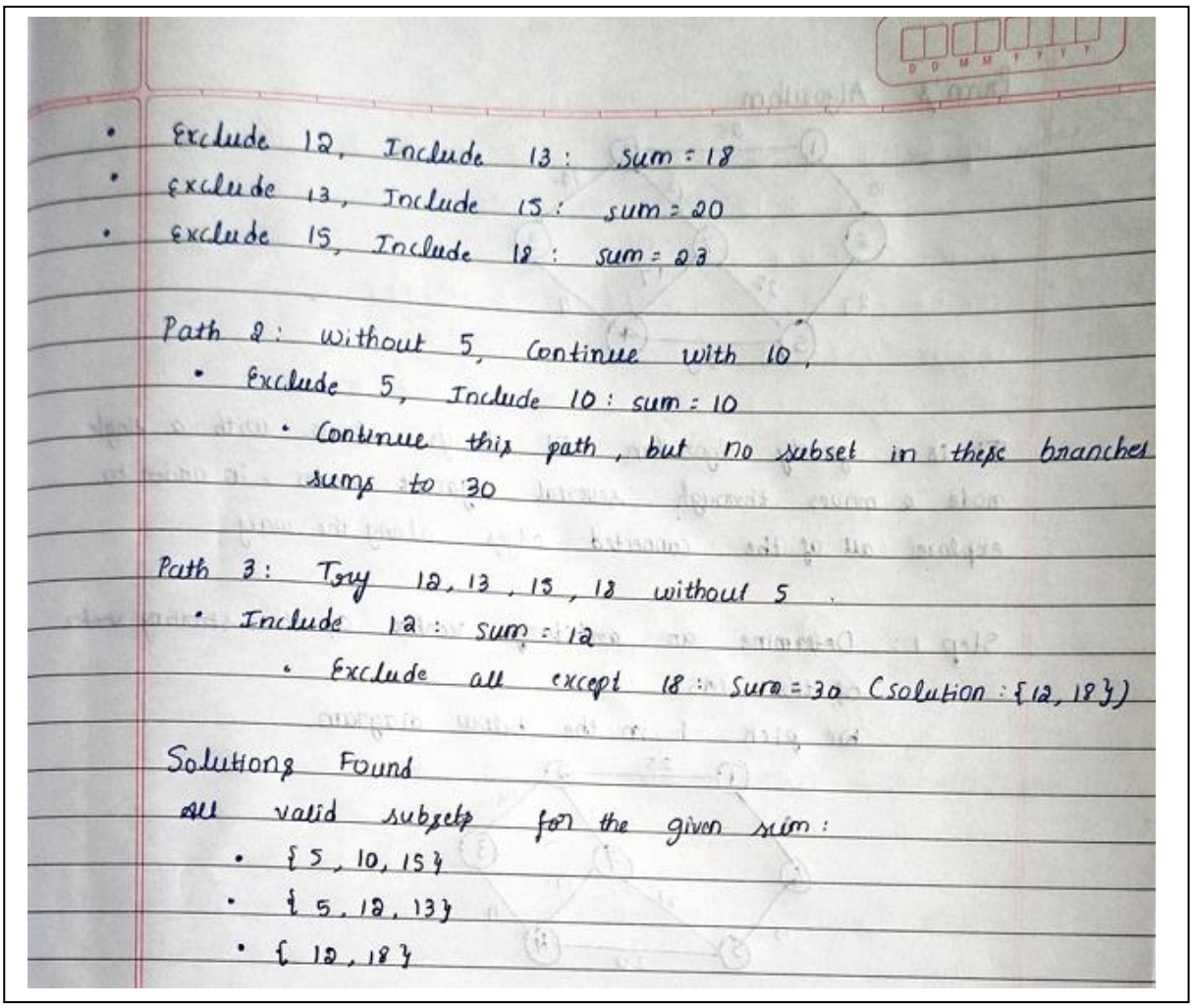
S = {5, 10, 12, 13, 15, 18} d = 30

Step - by - Step Backtracking

1. Start with Sorted set
The set is already sorted.

2. Build State Space Tree
At each node, decide whether to include or exclude the current element.
Trace each branch; if the sum exceeds 30, backtrack

3. Explore All Paths
Path 1: Begin with 5
- Include 5: sum = 5
  - Include 10: sum = 15
    - Include 12: sum = 27
      - Include 13: sum = 40 (exceeds 30, backtrack)
      - Exclude 13, include 15: sum = 42 (exceeds 30, backtrack)
      - Exclude 15, include 18: sum = 45 (exceeds 30, backtrack)
    - Exclude 12, include 13: sum = 28
      - Include 15: sum = 43
      - Exclude 15, Include 18: Sum = 46
  - Exclude 13, Include 15: Sum = 30 ✓ solution: {5, 10, 15}
  - Exclude 15, Include 18: sum = 33
- Exclude 10, Include 12: sum = 17
  - Include 13: sum = 30 (solution: {5, 12, 13})
  - Exclude 13, Include 15: Sum = 32
  - Exclude 15, Include 18: sum = 35

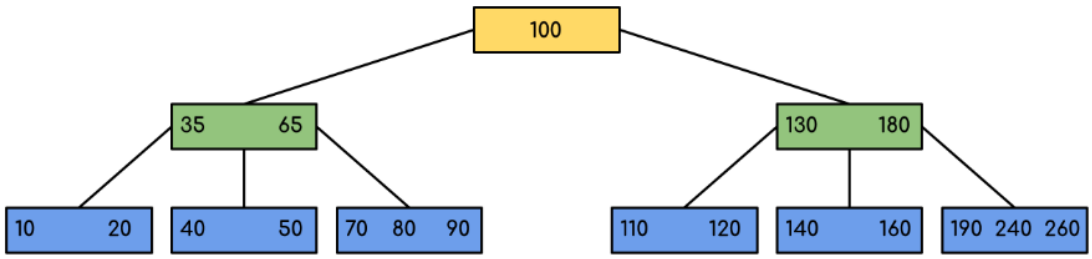


2. Describe about the B trees in detail notes.

A B-Tree is a specialized m-way tree designed to optimize data access, especially on disk-based storage systems.

- In a B-Tree of order m, each node can have up to m children and m-1 keys, allowing it to efficiently manage large datasets.
- The value of m is decided based on disk block and key sizes.
- One of the standout features of a B-Tree is its ability to store a significant number of keys within a single node, including large key values. It significantly reduces the tree's height, hence reducing costly disk operations.
- B Trees allow faster data retrieval and updates, making them an ideal choice for systems requiring efficient and scalable data management. By maintaining a balanced structure at all times,
- B-Trees deliver consistent and efficient performance for critical operations such as search, insertion, and deletion.

Following is an example of a B-Tree of order 5 .



Properties of a B-Tree

- A B Tree of order m can be defined as an m-way search tree which satisfies the following properties:
-
- All leaf nodes of a B tree are at the same level, i.e. they have the same depth (height of the tree).
- The keys of each node of a B tree (in case of multiple keys), should be stored in the ascending order.
- In a B tree, all non-leaf nodes (except root node) should have at least m/2 children.
- All nodes (except root node) should have at least m/2 - 1 keys.
- If the root node is a leaf node (only node in the tree), then it will have no children and will have at least one key. If the root node is a non-leaf node, then it will have at least 2 children and at least one key.
- A non-leaf node with n-1 key values should have n non NULL children.
- We can see in the above diagram that all the leaf nodes are at the same level and all non-leaf nodes have no empty sub-tree and have number of keys one less than the number of their children.

Operations on B-Tree

- B-Trees support various operations that make them highly efficient for managing large datasets. Below are the key operations:

Sr. No.	Operation	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)
4.	Traverse	O(n)

Search Operation in B-Tree

- Search is similar to the search in Binary Search Tree. Let the key to be searched is k.
- 1.Start from the root and recursively traverse down.
  2.For every visited non-leaf node
- If the current node contains k, return the node.
- Otherwise, determine the appropriate child to traverse. This is the child just before the first key greater than k.
- 3.If we reach a leaf node and don't find k in the leaf node, then return NULL.
- Searching a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimized as if the key value is not present in the range of the parent then the key is present in another branch. As these values limit the search they are also known as limiting values or separation values. If we reach a leaf node and don't find the desired key then it will display NULL.

Insertion Operation

To insert a new key, we go down from root to leaf. Before traversing down to a node, we first check if the node is full. If the node is full, we split it to create space. Following is the complete algorithm.

- The algorithm starts with a node x and a key k to insert.
- Find the position i in the node x where k should be inserted:
  - Locate the first key in x greater than k, or move to the end if no such key exists.
- If x is a leaf node, directly insert k at position i in sorted order.
- If x is not a leaf node, proceed to the child node at position i:
  - Check if the child node is full (has the maximum number of keys allowed).
  - If the child is full:
    - Split the child node into two nodes.
    - Move the middle key of the child node up to the parent node (x).
    - Adjust the position i if k is greater than the promoted key.
- Recursively call the B-Tree-Insert procedure on the appropriate child node to continue the insertion.
- The process ends when k is successfully inserted into a leaf node, ensuring the tree remains balanced and within its key limit.

Deletion Process in B-Trees

Deletion from a B-tree is more complicated than insertion because we can delete a key from any node-not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children.

As in insertion, we must make sure the deletion doesn't violate the B-tree properties. Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (except that the root is allowed to have fewer than the minimum number t-1 of keys). Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The deletion procedure deletes the key k from the subtree rooted at x. This procedure guarantees that whenever it calls itself recursively on a node x, the number of keys in x is at least the minimum degree t. Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b then we delete x, and x's only child x.c1 becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

Applications of B-Trees

- It is used in large databases to access data stored on the disk
- Searching for data in a data set can be achieved in significantly less time using the B-Tree
- With the indexing feature, multilevel indexing can be achieved.
- Most of the servers also use the B-tree approach.
- B-Trees are used in CAD systems to organize and search geometric data.
- B-Trees are also used in other areas such as natural language processing, computer networks, and cryptography.

Advantages of B-Trees

- B-Trees have a guaranteed time complexity of O(log n) for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
- B-Trees are self-balancing.
- High-concurrency and high-throughput.
- Efficient storage utilization.

Disadvantages of B-Trees

- B-Trees are based on disk-based data structures and can have a high disk usage.
- Not the best for all cases.
- For small datasets, the search time in a B-Tree might be slower compared to a binary search tree, as each node may contain multiple keys.

PART II

3. Describe about the AVL Tree in detail notes

Introduction

An AVL Tree (Adelson-Velsky and Landis, 1962) is a self-balancing Binary Search Tree (BST). It ensures that the heights of the left and right subtrees of any node differ by at most 1. This guarantees O(log n) performance for search, insert, and delete operations.

Balance Factor

(BF) BF(node) = Height(left subtree) - Height(right subtree)

• BF = -1, 0, +1 → Balanced

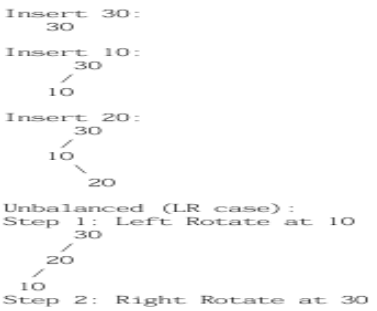
• BF < -1 or BF > +1 → Unbalanced, requires rotation

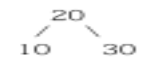
Rotations in AVL Tree

1. LL Rotation (Right Rotation)
2. RR Rotation (Left Rotation)
3. LR Rotation (Left-Right Rotation)
4. RL Rotation (Right-Left Rotation)

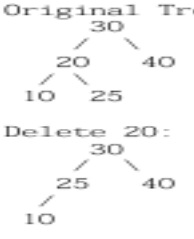
Example 1: Insertion (10, 20, 30)

Example 2: LR Case (30, 10, 20)



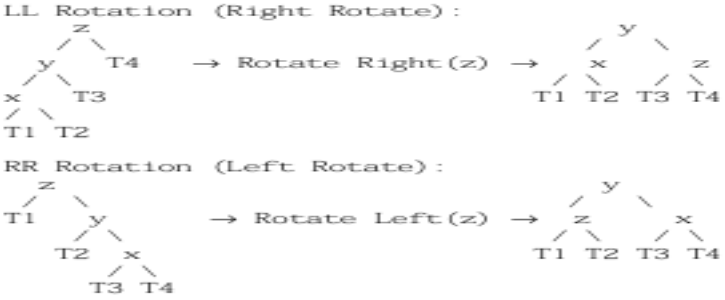


Example 3: Deletion



Tree remains balanced.

Rotations Summary



4. Define dynamic programming. How is it different from recursion? Explain with any two example.

Definition

- **Dynamic Programming** is a method of solving problems by **breaking them down into overlapping subproblems** and solving each subproblem only once, storing the results (usually in a table or array) for future use.
- It is mainly used to optimize **recursive problems** that recompute the same results multiple times

Key Characteristics

1. **Optimal Substructure:**
   - A problem has optimal substructure if its solution can be constructed from solutions of subproblems.
2. **Overlapping Subproblems:**
   - A problem has overlapping subproblems if the same subproblems occur multiple times during recursion.

Difference between Recursion and Dynamic Programming

Aspect	Recursion	Dynamic Programming
Definition	Problem solved by breaking into subproblems, solved via function calls.	Enhances recursion by storing subproblem results to avoid recomputation.
Efficiency	May recompute the same subproblem many times → exponential time (O(2^n) for Fibonacci).	Avoids recomputation by storing results → polynomial time (O(n) for Fibonacci).
Memory Usage	Uses call stack (may lead to stack overflow for deep recursions).	Uses additional memory (table/array) to store results.
Approach	*Top-down only*.	Can be implemented *Top-down (Memoization)* or *Bottom-up (Tabulation)*.

Example 1: Fibonacci Numbers

Recursive approach (without DP):

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

- Time Complexity: O(2^n) (recomputes many subproblems).

DP approach (Memoization – Top-down):

```
memo = {}
def fib_dp(n):
    if n <= 1:
        return n
    if n not in memo:
        memo[n] = fib_dp(n-1) + fib_dp(n-2)
    return memo[n]
```

- Time Complexity: O(n).
- Stores results of subproblems in `memo`.

Example 2: 0/1 Knapsack Problem

- **Problem:** Given `n` items with weights and values, and a knapsack capacity `W`, maximize the total value without exceeding the capacity.

Recursive approach:

```
def knapsack(wt, val, W, n):
    if n == 0 or W == 0:
        return 0
    if wt[n-1] > W:
        return knapsack(wt, val, W, n-1)
    else:
        return max(val[n-1] + knapsack(wt, val, W-wt[n-1], n-1),
                   knapsack(wt, val, W, n-1))
```

- Time Complexity: O(2^n).

DP approach (Bottom-up – Tabulation):

```
def knapsack_dp(wt, val, W, n):
    dp = [[0]*(W+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for w in range(1, W+1):
            if wt[i-1] <= w:
                dp[i][w] = max(val[i-1] + dp[i-1][w-wt[i-1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][W]
```

- Time Complexity: O(nW).
- Stores solutions in a **2D table**.

PART III

5. Explain about the Trie and its types with neat example
TRIES DATA STRUCTURE

Trie is an efficient information re***Trie***val data structure. The term tries comes from the word retrieval

Definition of a Trie

· Data structure for representing a collection of strings

· In computer science , a trie also called digital tree or radix tree or prefix tree. · Tries support fast string matching.

Properties of Tries

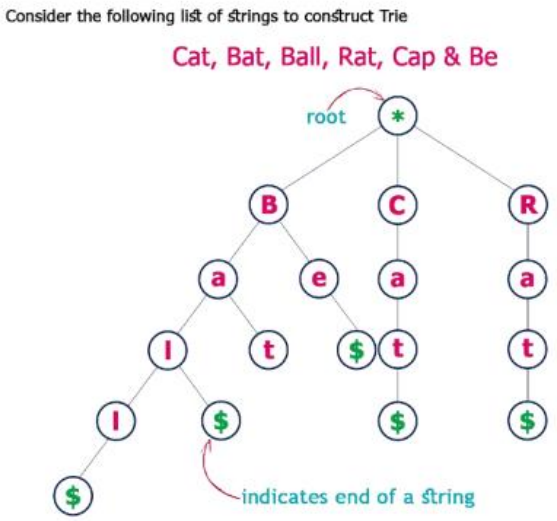
· A Multi way tree

· Each node has from 1 to n children

· Each edge of the tree is labeled with a character

· Each leaf node corresponds to the stored string which is a concatenation of characters on a path from the root to this node.

EXAMPLE

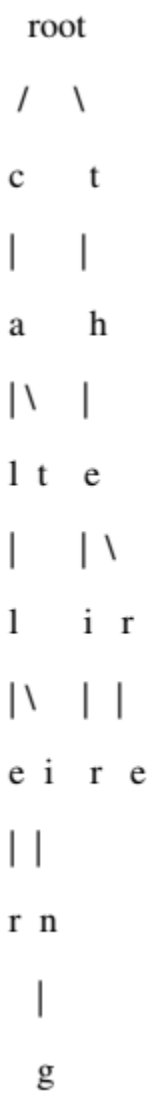


Trie | (Insert and Search)

Trie is an efficient information retrieval data structure. Using Trie, search complexities can be brought to an optimal limit (key length).Given multiple strings. The task is to insert the string in a Trie

Examples:

Example 1: str = {"cat", "there", "caller", "their", "calling", “bat”}



Approach: An efficient approach is to treat every character of the input key as an individual trie node and insert it into the trie. Note that the children are an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array of children. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth.

Trie deletion Here is an algorithm how to delete a node from trie.

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.

2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.

3. Key is prefix key of another long key in trie. Unmark the leaf node.

4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

Time Complexity: The time complexity of the deletion operation is O(n) where n is the key length

Advantages of Trie Data Structure

Tries is a tree that stores strings. The maximum number of children of a node is equal to the size of the alphabet. Trie supports search, insert and delete operations in O(L) time where L is the length of the key.

Hashing:- In hashing, we convert the key to a small value and the value is used to index data. Hashing supports search, insert and delete operations in O(L) time on average.

Self Balancing BST : The time complexity of the search, insert and delete operations in a self-balancing Binary Search Tree (BST) (like Red-Black Tree, AVL Tree, Splay Tree, etc) is O(L * Log n) where n is total number words and L is the length of the word. The advantage of Self-balancing BSTs is that they

maintain order which makes operations like minimum, maximum, closest (floor or ceiling) and kth largest faster.

APPLICATIONS OF TRIES

String handling and processing are one of the most important topics for programmers. Many real time applications are based on the string processing like:

1. Search Engine results optimization
2. Data Analytics
3. Sentimental Analysis

The data structure that is very important for string handling is the Trie data structure that is based on prefix of string

TYPES OF TRIES

Tries are classified into three categories:

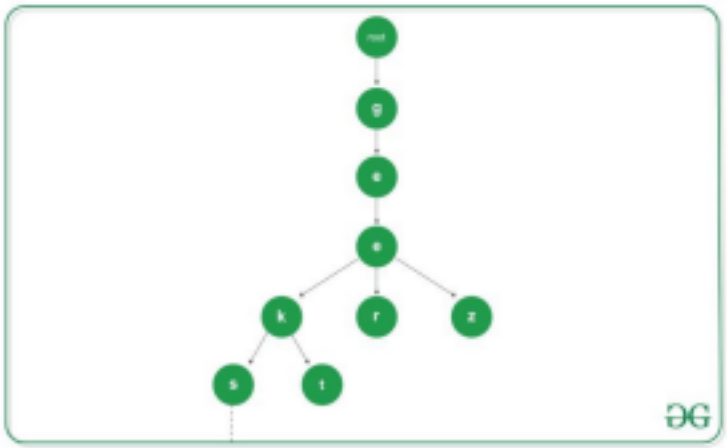
1. Standard Tries
2. Compressed Tries
3. Suffix Tries

STANDARD TRIES

A standard trie have the following properties:}

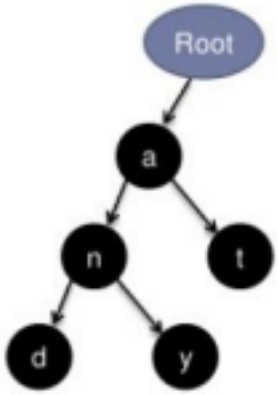
· It is an ordered tree like data structure.

· Each node(except the root node) in a standard trie is labeled with a character. · The children of a node are in alphabetical order.
· Each node or branch represents a possible character of keys or words. · Each node or branch may have multiple branches.
· The last node of every key or word is used to mark the end of word or node. · The path from external node to the root yields the string of S. Below is the illustration of the Standard Trie



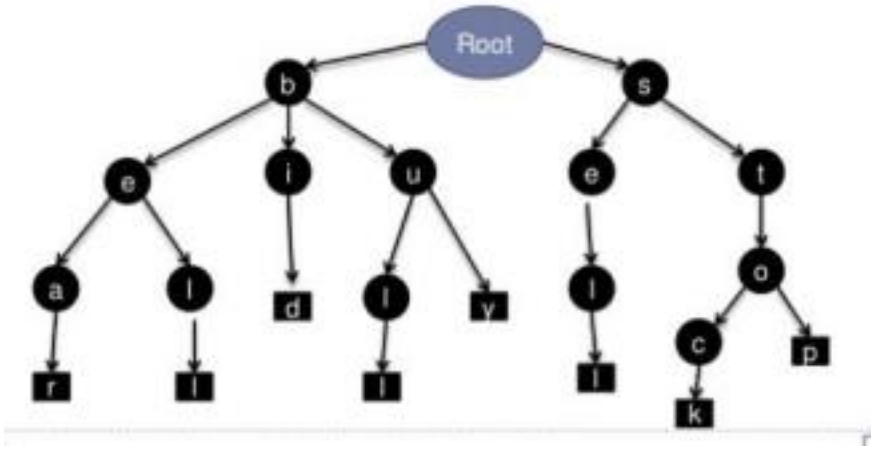
Standard Trie Insertion

Strings={ a,an,and,any}



Example of Standard Trie

Standard trie for the following strings S={ bear, bell, bid, bull, buy, sell, stock, stop}



Handling Keys(strings)

· When a key is prefix of another key

How can we know that “an “ is a word

Example : an, and

COMPRESSED TRIE

A Compressed trie have the following properties:

1. A Compressed Trie is an advanced version of the standard trie.

2. Each nodes(except the leaf nodes) have atleast 2 children.

3. It is used to achieve space optimization.

4. To derive a Compressed Trie from a Standard Trie, compression of chains of redundant nodes is performed.

It consists of grouping, re-grouping and un-grouping of keys of characters.

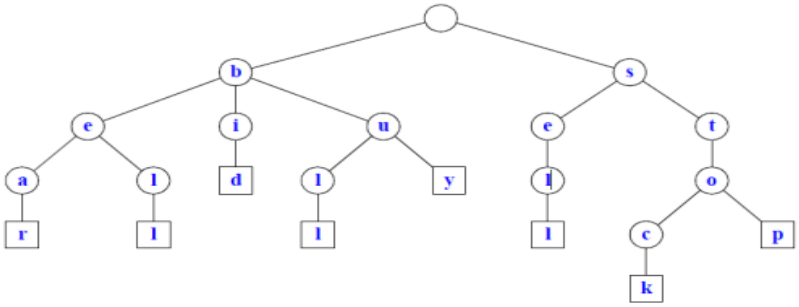
6. While performing the insertion operation, it may be required to un-group the already

grouped characters.

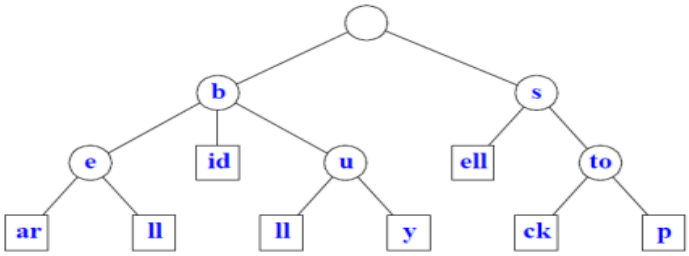
7. While performing the deletion operation, it may be required to re-group the already grouped characters.

Compressed trie is constructed from standard trie

- Standard Trie:



- Compressed Trie:



Storage of Compressed Trie

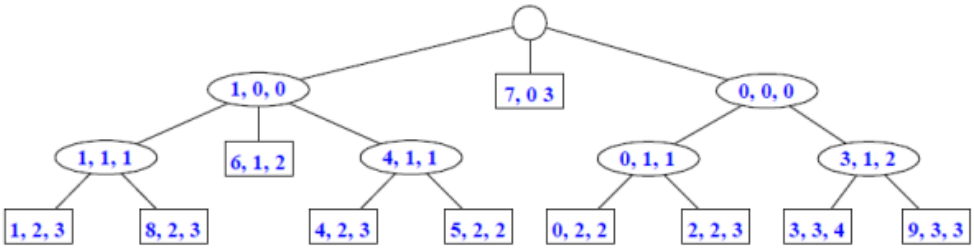
A compressed Trie can be stored at O9s) where s= | S| by using O(1) Space index ranges at the nodes

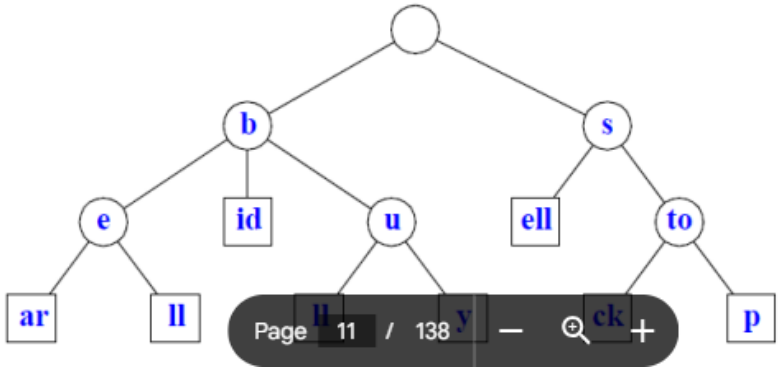
In the below representation each node is represented with (I,j,k) value
I ---- indicate index of the string
j—starting index of the character of string I
k--- ending index of the character of the string I
Ex: In the given diagram node (4,2,3) having the characters**(ll)** which belongs to s[4] so i=4, index of l character in s[4] is 2 so j=2 and ending index is 3 so k=3

	0	1	2	3	4
S[0] =	s	e	e		
S[1] =	b	e	a	r	
S[2] =	s	e	l	l	
S[3] =	s	t	o	c	k

	0	1	2	3
S[4] =	b	u	l	l
S[5] =	b	u	y	
S[6] =	b	i	d	

	0	1	2	3
S[7] =	h	e	a	r
S[8] =	b	e	l	l
S[9] =	s	t	o	p





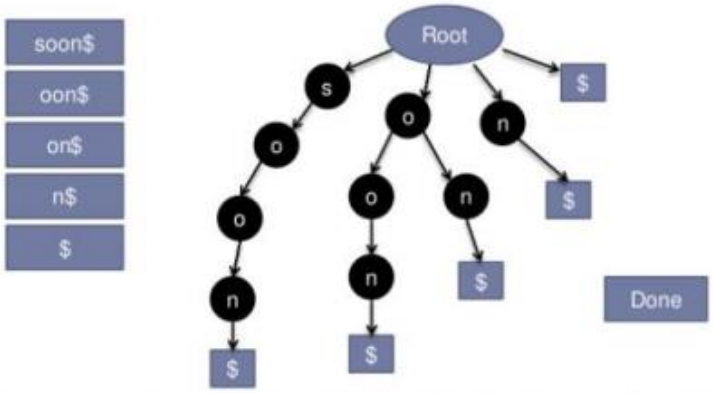
SUFFIX TRIES

A Suffix trie have the following properties:

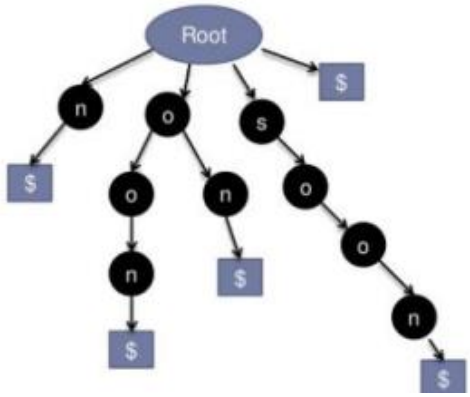
1. Suffix trie is a compressed trie for all the suffixes of the text
2. Suffix trie are space efficient data structure to store a string that allows many kinds of queries to be answered quickly.

Example

Let us consider an example text "soon$"



After alphabetically order the trie look like



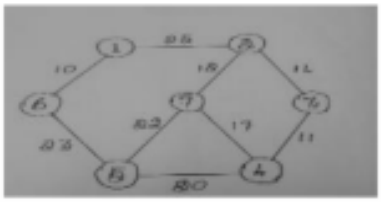
Advantages of suffix tries

1. Insertion is faster compared to the hash table

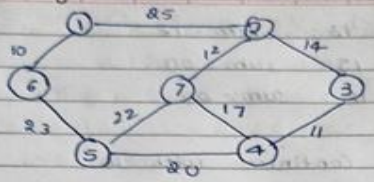
2. Look up is faster than hash table implementation

3. There are no collision of different keys in tries

6. **Apply Prims algorithm to the following given graph to construct minimum spanning tree**

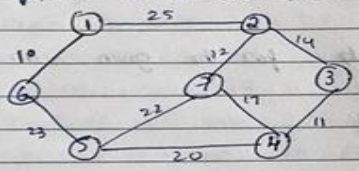


Prim's Algorithm



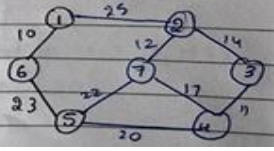
It is a greedy algorithm. It always starts with a single node & moves through several adjacent nodes, in order to explore all of the connected edges along the way.

Step 1: Determine an arbitary vertex as the starting vertex of the MST.
We pick 1 in the below diagram.



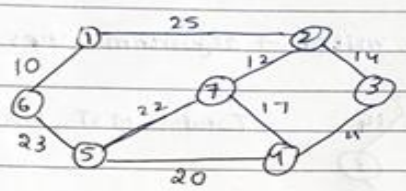
- visited = {1}
- Candidate edges = {1-2 (25), 1-6 (10)}
- choose 1-6 (10) → add to MST (because 10 is smaller than 25)
- MST edges so far: {1-6}
- Total weight = 10

Step 2: Since 6's adjacent vertex is 5, choose 5
- visited = {1, 6}
- Candidate edges = {1-2 (25), 6-5 (23)}
- Choose 6-5 (23)
- MST edges = {1-6, 6-5}
- Total weight = 10 + 23 = 33



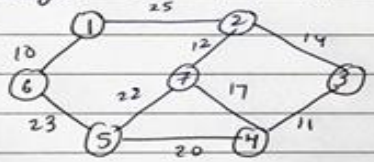
Step 3: Vertex 5 adjacent's are 7 & 4 with the weights 22 & 20.

- visited = {1, 6, 5}
- Candidate edges = {5,7 (2), 5,4 (20)}
- Choose 5-4 (20) (smallest).
- MST edges: {1-6, 6-5, 5-4}
- Total weight = 33+20 = 53



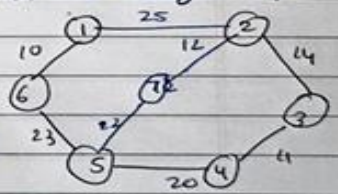
Step 4: Vertex 4 adjacent's are 7 & 3 with the weights 17 & 11.

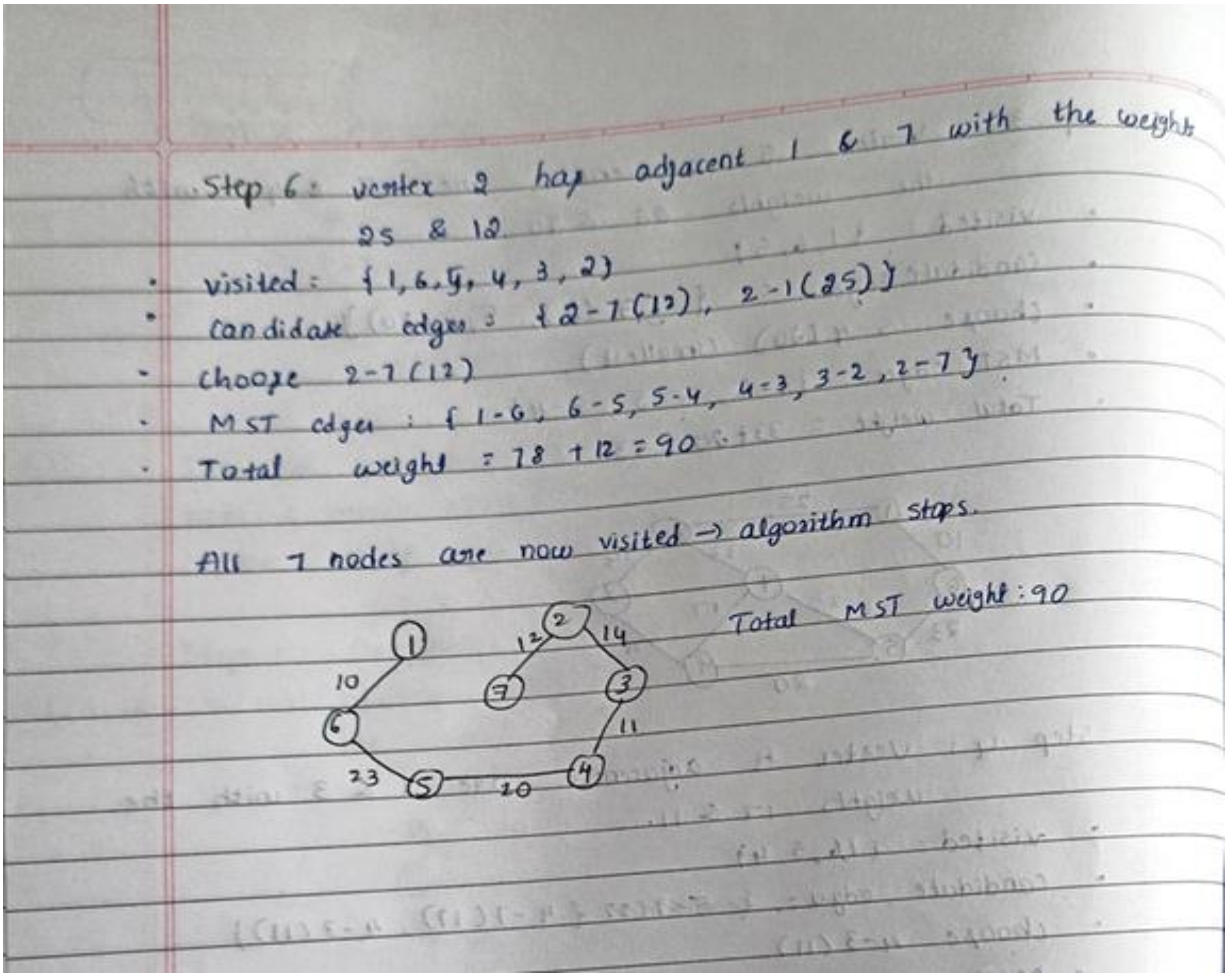
- visited: {1,6,5,4}
- Candidate edges = {4-7 (17), 4-3 (11)}
- choose 4-3 (11).
- MST edges: {1-6, 6-5, 5-4, 4-3}
- Total weight = 53 + 11 = 64



Step 5: Vertex 3 adjacent is 2 with the weight 14.

- Visited = {1,6,5,4,3}
- Candidate edges = {3-2 (14)}
- Choose 3-2 (14)
- MST edges: {1-6, 6-5, 5-4, 4-3, 3-2}
- Total weight = 64 + 14 = 78





PART IV

7. Describe real-world applications of advanced data structures

1. Arrays

- **Applications:**
  - Used in **databases** and **spreadsheets** for storing tabular data.
  - Storing **images and matrices** in computer graphics.
  - Implementing lookup tables and static memory allocation.

2. Linked Lists

- **Applications:**
  - **Dynamic memory allocation** (managing free memory blocks).
  - Implementing **undo/redo** functionality in text editors.
  - **Music/Video playlists** where next and previous items are linked.
  - Chaining in **Hash Tables** for collision resolution.

3. Stacks

- **Applications:**
  - **Function call management** in recursion (system call stack).
  - **Undo feature** in editors.
  - **Expression evaluation and syntax parsing** in compilers.
  - **Backtracking algorithms** (maze solving, DFS traversal).

4. Queues

- **Applications:**
  - **CPU scheduling** (Round Robin, FCFS).
  - **Print spooling** in operating systems.
  - **Order processing** in e-commerce systems.
  - **Network data packet management** (buffer handling).

5. Trees

Binary Search Trees (BST):

- **Applications:**
  - Implementing **search operations** in databases.
  - **Indexing** in file systems.
  - Used in **spell checkers** and **autocomplete systems**.

AVL Trees / Red-Black Trees:

- **Applications:**
  - **Database indexing** (to keep queries efficient).
  - **Memory management** (balanced allocation and search).
  - High-performance **dictionary/map implementations**.

Heaps:

- **Applications:**
  - **Priority queues** (task scheduling, shortest job first).
  - **Heap Sort** algorithm.
  - **Graph algorithms** (Dijkstra's shortest path, Prim's MST).

- **Memory management in operating systems**.

Tries (Prefix Trees):

- **Applications:**
  - **Autocomplete and spell-checking**.
  - **IP routing** in networking.
  - **Search engines** (efficient word lookup).
  - **Dictionary implementation**.

6. Graphs

- **Applications:**
  - **Social networks** (Facebook friend suggestions, LinkedIn connections).
  - **Navigation systems** (Google Maps, GPS shortest path).
  - **Webpage ranking** (Google PageRank).
  - **Computer networks** (routing algorithms, data transmission).
  - **Project scheduling** (PERT/CPM charts).

7. Hashing

- **Applications:**
  - **Database indexing** (fast data retrieval).
  - **Password storage** (secure hashing).
  - **Caching mechanisms** (hash maps in memory).
  - **Load balancing** in distributed systems.
  - **Blockchain and cryptography** (hash functions).

8. Advanced Data Structures

- **Segment Trees / Fenwick Trees (Binary Indexed Trees):**
  - Range queries in databases (e.g., sum, min, max over intervals).
  - **Competitive programming** problems.
  - **Gaming** (range updates and queries).
- **B-Trees / B+ Trees:**
  - Widely used in **database indexing**.
  - **File systems** (NTFS, ext4) for directory structure.
- **Disjoint Set (Union-Find):**
  - **Network connectivity** checking.
  - **Image processing** (connected component analysis).
  - **Kruskal's MST algorithm** in graphs.
- **Bloom Filters:**
  - **Web cache filtering** (check if a page is cached).
  - **Databases** (checking membership quickly).
  - **Spam email detection**.

8. Define a segment tree. How is it used for range minimum/maximum/sum queries?

Definition

- A **Segment Tree** is a **binary tree data structure** used to efficiently perform **range queries** (like sum, minimum, maximum) and **range updates** on arrays.
- It is a **divide-and-conquer** based structure, where:
  - The **root node** represents the entire array.
  - Each **internal node** represents a segment (range) of the array.
  - The **leaf nodes** represent individual array elements.

Properties

1. Built for an array of size **n**, the segment tree has about **2n – 1 nodes**.
2. **Height of tree** = O(log n).
3. **Construction Time:** O(n).
4. **Query Time (sum/min/max in a range):** O(log n).
5. **Update Time (changing an element):** O(log n).

Construction of Segment Tree

- Split the array into **two halves** recursively until individual elements are reached.
- Store results (sum, min, max) at each node.

Example: Array `A = [2, 5, 1, 4, 9, 3]`

```
Segment Tree (for sum):
               [0..5] = 24
              /            \
       [0..2]=8            [3..5]=16
       /     \              /     \
  [0..1]=7  [2]=1     [3..4]=13  [5]=3
   /   \
[0]=2  [1]=5
```

Range Queries

Segment Trees are mainly used for:

1. Range Sum Query (RSQ)

- Example: Sum from index **l to r**.
- Traverse the tree and add only those segments that fully lie inside `[l, r]`.
- Time: **O(log n)**

2. Range Minimum Query (RMQ)

- Example: Find **minimum element** from index **l to r**.
- Traverse the tree, compare only valid segments.
- Time: **O(log n)**

3. Range Maximum Query

- Example: Find **maximum element** from index **l to r**.
- Similar logic as RMQ, but take maximum instead of minimum.
- Time: **O(log n)**

Example

Array: `A = [2, 5, 1, 4, 9, 3]`

- **Range Sum Query (1, 4):**
  → Elements: [5, 1, 4, 9] = 19
- **Range Min Query (1, 4):**
  → Minimum = 1
- **Range Max Query (1, 4):**
  → Maximum = 9

Advantages

- Much faster than naive O(n) approach.
- Supports **dynamic updates** (unlike prefix-sum arrays).

Applications

- Competitive programming (frequent range queries).
- Database indexing.
- Image processing.
- Game leaderboards (top score queries).

PART V

9. What is a Fenwick Tree (Binary Indexed Tree)?How does it differ from a segment tree in use and implementation?

Definition

- A **Fenwick Tree**, also known as a **Binary Indexed Tree (BIT)**, is a data structure that provides efficient methods for **prefix sum queries** and **point updates** on an array.

- It is based on the idea of representing cumulative frequency using binary representation of indices.

Operations

1. **Update(index, value):** Add a value to an element.
   - Time: **O(log n)**
2. **Query(index):** Find the prefix sum from `0` to `index`.
   - Time: **O(log n)**
3. **Range Sum(l, r):**

   Sum(l,r)=Query(r)−Query(l−1)Sum(l, r) = Query(r) - Query(l-1)Sum(l,r)=Query(r)−Query(l−1)

Construction

- Initialize an array `BIT[]` of size `n+1` (1-based indexing).
- Update BIT values using **least significant bit (LSB)** property.
- Each node stores the sum of a specific range of elements.

Example: Array = `[1, 2, 3, 4, 5]`
BIT representation stores cumulative sums at selective ranges depending on LSB of indices.

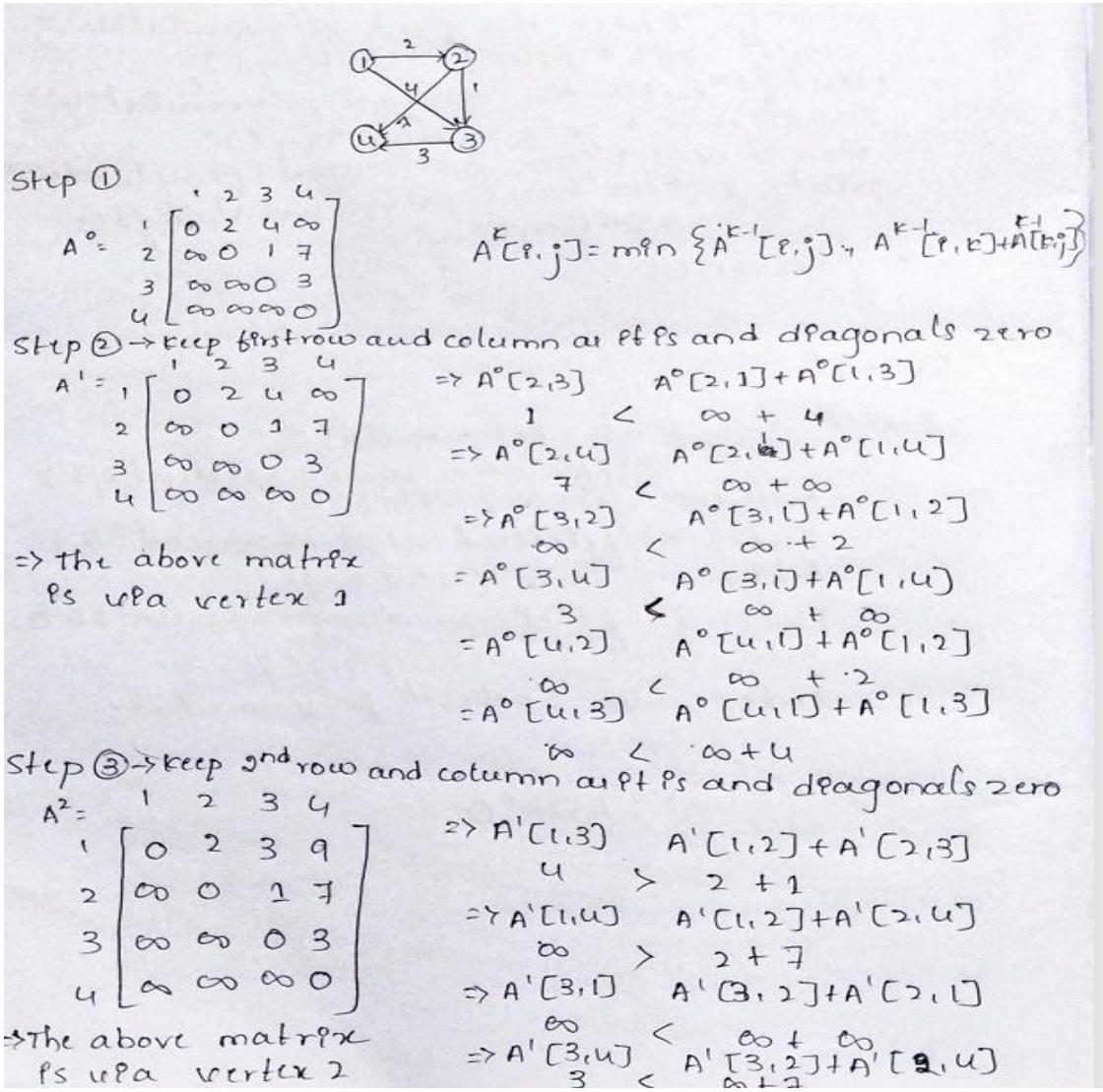
How it Works

- Uses binary decomposition of indices.
- Example: Index `6` (in binary `110`) stores sum of last `2` elements.
- To move to parent, subtract the last set bit.

Fenwick Tree vs Segment Tree

Feature	Fenwick Tree (BIT)	Segment Tree
Use case	Efficient for prefix sums and point updates.	Handles a wider range of queries: sum, min, max, gcd, etc.
Memory Usage	Requires **O(n)** space.	Requires **O(4n)** space (due to tree representation).
Implementation	Easier to implement (array-based, bit manipulation).	More complex (recursive or iterative tree structure).
Query Time	O(log n)	O(log n)
Update Time	O(log n)	O(log n)
Range Queries	Supports range sums (via prefix differences).	Supports many queries: sum, min, max, etc.
Flexibility	Less flexible, mostly prefix-based operations.	Very flexible, supports segment-based operations.
Best Use Case	Frequency tables, cumulative sums, inversion count.	General-purpose range queries, advanced queries in competitive programming.

10. Consider the following weighted graph: Vertices: {1,2,3,4} Edges: 1→2 (weight 2) 1→3 (weight 4) 2→3 (weight 1) 2→4 (weight 7) 3→4 (weight 3). Apply Floyd-Warshall algorithm and give the final shortest path matrix.



$\Rightarrow A^1[4,1] \quad A^1[4,2] + A^1[2,1]$
$\infty \quad < \quad \infty + \infty$

$\Rightarrow A^1[4,3] \quad A^1[4,2] + A^1[2,3]$
$\infty \quad < \quad \infty + 1$

step ③ → keep third row and column as it is and diagonals '0':

$A^3 =$

	1	2	3	4
1	0	2	3	6
2	∞	0	1	4
3	∞	∞	0	3
4	∞	∞	∞	0

$\Rightarrow$ the above matrix is upto vertex 3

$\Rightarrow A^2[1,2] \quad A^2[1,3] + A^2[3,2]$
$2 \quad < \quad 3 + \infty$

$\Rightarrow A^2[1,4] \quad A^2[1,3] + A^2[3,4]$
$9 \quad > \quad 3 + 3$

$\Rightarrow A^2[2,1] \quad A^2[2,3] + A^2[3,1]$
$\infty \quad < \quad 1 + \infty$

$\Rightarrow A^2[2,4] \quad A^2[2,3] + A^2[3,4]$
$7 \quad > \quad 1 + 3$

$\Rightarrow A^2[4,1] \quad A^2[4,3] + A^2[3,1]$
$\infty \quad < \quad \infty + \infty$

$= A^2[4,2] \quad A^2[4,3] + A^2[3,2]$
$\infty \quad < \quad \infty + \infty$

Step ④ → keep 4th row and column as it is and diagonals '0':

$A^4 =$

	1	2	3	4
1	0	2	3	6
2	∞	0	1	4
3	∞	∞	0	3
4	∞	∞	∞	0

$\Rightarrow A^3[1,2] \quad A^3[1,4] + A^3[4,2]$
$2 \quad < \quad 6 + \infty$

$\Rightarrow A^3[1,3] \quad A^3[1,4] + A^3[4,3]$
$3 \quad < \quad 6 + \infty$

$\Rightarrow A^3[2,1] \quad A^3[2,4] + A^3[4,1]$
$\infty \quad < \quad 4 + \infty$

$\Rightarrow A^3[2,3] \quad A^3[2,4] + A^3[4,3]$
$1 \quad < \quad 4 + \infty$

$\Rightarrow A^3[3,1] \quad A^3[3,4] + A^3[4,1]$
$\infty \quad < \quad 3 + \infty$

$\Rightarrow A^3[3,2] \quad A^3[3,4] + A^3[4,2]$
$\infty \quad < \quad 3 + \infty$