

DSA PRACTICE

NAME: Abdullah Mohsin

ALL DSA CONTENT:

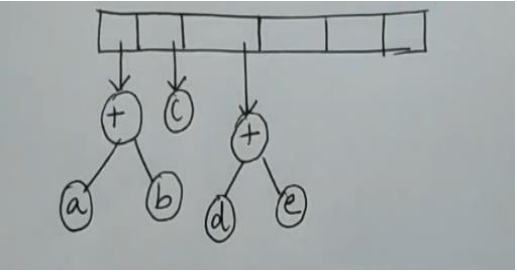
DATA STRUCTURE AND ALGORITHM:	3
Fig 1.1: DATA STRUCTURE CLASSIFICATION	3
• ALGORITHM:	3
Fig 1.2: APPLES ARE IN THE BASKET.	3
• TUTORIAL:	5
How to write an algorithm?	5
EXAMPLE:	5
ALGORITHM:	5
DRY RUN ALGORITHM:	5
1. PRIMITIVE DATA:	5
2. NON-PRIMITIVE DATA:	6
Fig 1.4:Arrays Data Structure	6
Algorithm: (Bubble sort) BUBBLE (DATA, N)	7
Algorithm 1.1: QUICK (A, N, BEG, END, LOC)	7
Fig 1.5: Linked List	8
Fig 1.22:Simple Linked List	8
Fig 1.23:Doubly Linked list	8
Fig 1.24:Circular Linked list	8
Fig 1.6:Stack	9
Fig 1.7:Queue	10
Fig 1.14: Types of trees.	10
Based on Structure:	11
Figure 1.15:Full Binary Tree	11
Figure 1.16: Complete Binary Tree	11
Figure 1.17: Perfect Binary Tree	12
Figure 1.18:Degenerate Tree.	12
Basic Terminologies In Tree Data Structure:	12
Figure 1.19:Types of Graphs:	12
Advanced Graph Terminology:	13
1. Directed Graph (Digraph):	13
2. Undirected Graph:	13
3. Weighted Graph:	13
4. Unweighted Graph:	13
5. Connected Graph:	13
6. Acyclic Graph:	13
7. Cyclic Graph:	13
8. Connected Graph	13
9. Disconnected Graph:	13
▪ STRING MANIPULATION:	13
Fig1.8: FINITE AUTOMATA	13
Fig 1.10: Table	14
Fig 1.9: Algorithm	14
▪ RECURSION:	14
Fig 1.11:Factorial	14
Tower of Honie is one of the best example of recursion.	14
Fig 1.12:Tower of Honai	14
STACK APPLICATION:	14
1. Prefix Notation (Polish Notation)	14
2. Postfix Notation (Reverse Polish Notation)	15
3. Infix Notation.	15
Why Do We Use Prefix/Postfix?	15
Expression Example: 2 3 + 4 *	15
Example: Convert Infix to Postfix	16
EXPRESION TREE EXAMPLE:	16

Fig 1.A:Expression Tree.....

16

Fig 1.B:Expression Tree.....

16



.....

16

Fig 1.C:Expression Tree.....

16

.....

17

Fig 1.D:Expression Tree.....

17

Fig 1.E:Expression Tree.....

17

Heap:

17

Fig 2.1: Heap

17

2 | Page

DATA STRUCTURE AND ALGORITHM:

Here, In this note we will discuss how data structure and algorithm works, how to create algorithm and implement it by using C++.We will also discuss the primitive data structure and non-primitive data structure. We will also understand how static and dynamic memory works.

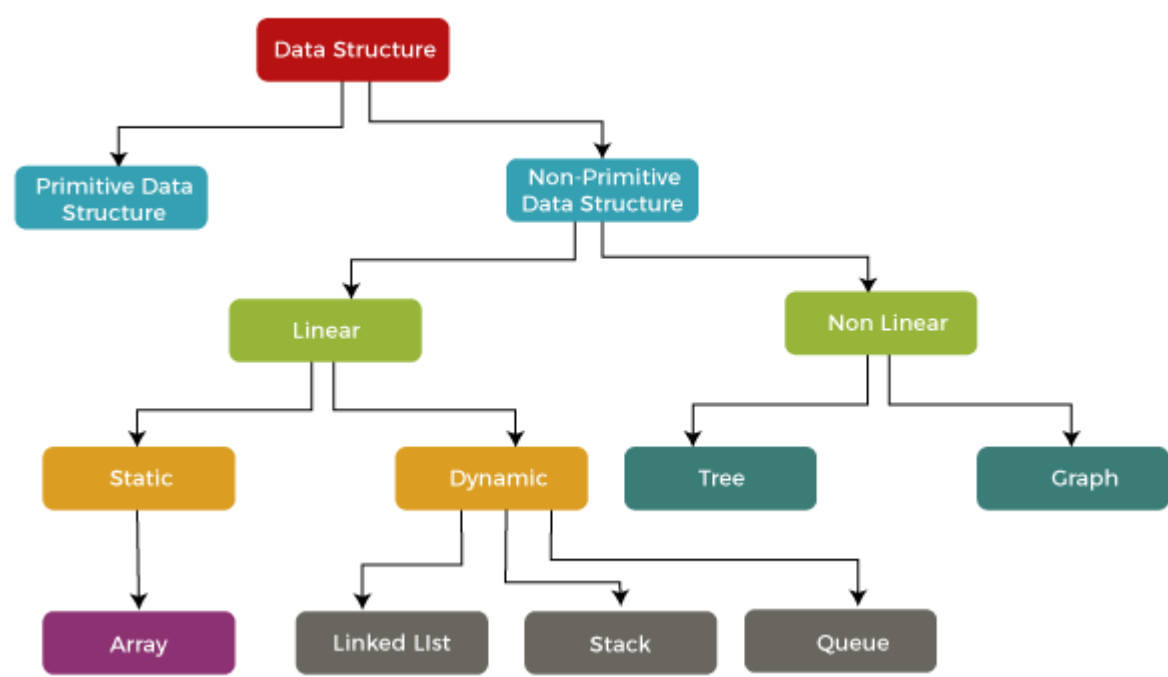


Fig 1.1: DATA STRUCTURE CLASSIFICATION

• **ALGORITHM:**

The word Algorithm means ” A set of finite rules or instructions to be followed in calculations or other problem-solving operations ” Or ” A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations”.



Fig 1.2: APPLES ARE IN THE BASKET.

Here in the fig 1.2, you can see the apples are in the basket, it is in organized form, similar in programming, data need to be organized in structured form so it can easily get the output and it will be efficient also. Huge quantity of data must need to be organized like we talk example of huge system like Nasa system, it having a huge amount of data in order to store data in structured form, they must need an efficient algorithm.

There are two factors that affect performance of algorithm:

- 1.Time Complexity
- 2.Space Complexity

1. Time Complexity

Time complexity measures how the runtime of an algorithm changes as the size of the input grows. It helps us understand how efficiently an algorithm performs. It is typically expressed in **Big-O Notation**, which gives the upper bound of the runtime.

Common Time Complexities:

- **O(1):** Constant time – The algorithm's runtime does not depend on the input size.
- **O(log n):** Logarithmic time – The runtime grows logarithmically as the input size increases (e.g., binary search).
- **O(n):** Linear time – The runtime grows proportionally with the input size (e.g., single loop).
- **O(n log n):** Log-linear time – Common in efficient sorting algorithms (e.g., merge sort, quicksort).
- **O(n²):** Quadratic time – The runtime grows with the square of the input size (e.g., nested loops).
- **O(2^n):** Exponential time – Runtime doubles with each additional input size (e.g., recursive solutions to the Tower of Hanoi).
- **O(n!):** Factorial time – Extremely slow growth, seen in problems like the traveling salesman.

2. Space Complexity

Space complexity measures the amount of memory an algorithm needs as the input size grows. It includes:

- **Auxiliary space:** Extra space used during computation (e.g., temporary variables, recursion stack).
- **Input space:** Space taken by the input data itself.

Types of Space Complexity:

- 1. **Fixed Part (Constant Space):**
 - Memory required that does not depend on input size.

- Example: Variables, pointers, constants.
2. **Variable Part:**
- Memory required that grows with input size.
 - Includes data structures like arrays, lists, or recursion stacks.

Further Types Based on Usage:

1. **$O(1)$: Constant Space**
 - No extra memory is used, except for a few variables.
 - Example: Swapping two variables.
2. **$O(\log n)$: Logarithmic Space**
 - Space grows logarithmically with input size.
 - Example: Recursive binary search.
3. **$O(n)$: Linear Space**
 - Memory grows proportionally with input size.
 - Example: Storing an array of size n .
4. **$O(n^2)$: Quadratic Space**
 - Space grows as the square of the input size.
 - Example: Storing a 2D matrix for $n \times n$ elements.
5. **$O(2^n)$: Exponential Space**
 - Space requirements double with each additional input size.
 - Example: Recursive solutions with multiple calls like subsets of a set.

Asymptotic Notations

Asymptotic notations describe the behavior of an algorithm's complexity as the input size n grows to infinity. They provide a way to express the time or space complexity of an algorithm in a simplified manner.

1. Big-O Notation (O)

- Represents the **worst-case** complexity of an algorithm.
- Provides an upper bound on the time/space required.
- Ensures that the algorithm will not exceed this runtime.

Example:

If an algorithm has a complexity of $T(n)=3n^2+2n+1$, the Big-O is $O(n^2)$, as n^2 dominates for large n .

2. Omega Notation (Ω)

- Represents the **best-case** complexity of an algorithm.
- Provides a lower bound on the time/space required.
- Indicates the minimum time the algorithm will take.

Example:

For $T(n)=3n^2+2n+1$, the Omega is $\Omega(n^2)$ because n^2 is the fastest-growing term.

3. Theta Notation (Θ)

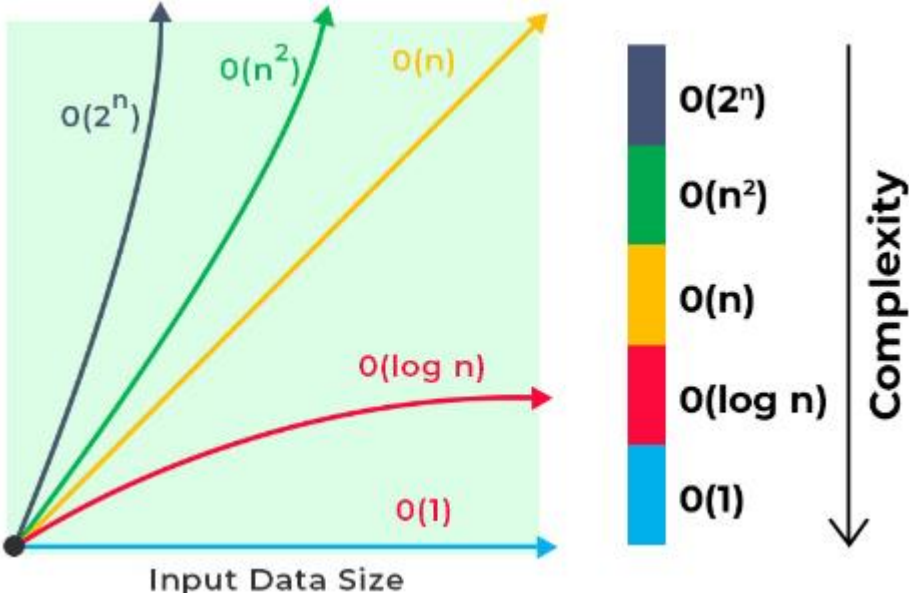
- Represents the **average-case** complexity or tight bound of an algorithm.
- Indicates that the algorithm's runtime grows at the same rate as the given function for both lower and upper bounds.

Example:

For $T(n)=3n^2+2n+1$, the Theta is $\Theta(n^2)$, as the function is bounded both above and below by n^2 .

All Cases

1. **Worst Case:**
 - Captures the maximum time an algorithm could take.
 - Expressed using **Big-O Notation (O)**.
 - Example: Searching for an element that doesn't exist in a list.
2. **Average Case:**
 - Describes the expected runtime for typical inputs.
 - Expressed using **Theta Notation (Θ)**.
 - Example: Searching for a random element in an unsorted list.
3. **Best Case:**
 - Represents the minimum time the algorithm could take.
 - Expressed using **Omega Notation (Ω)**.
 - Example: Searching for the first element in a list.



• **TUTORIAL:**

We are familiar with programming fundamental course, here we understanding how to organized data in well-organized or structured form. if any doubt contacts me :03363736231.

How to write an algorithm?

Writing the algorithm is difficult because it follows some condition like the input should be well defined, output should be well defined,it should not be finite and it should not be unambiguous, feasibility and independent.

Here, I am giving you example of writing algorithm and how to dry-run algorithm.

EXAMPLE:

Q1. Write the algorithm, where you can find MAX value from array.

ALGORITHM:

First step in algorithm writing is to figure the problem and then write it`s description

[A linear array DATA[]with N size of element is defined, K is set as the position of index and MAX is declared as the first index, LOC is the position .This algorithm is used to check the maximum value of array]

- [] **These two brackets are treated as comment.** (//) Which is mainly to describe.

Second Step is to add logic here we already declared a linear array in the description and K and LOC.

Step 1: [Initialize Counter] Set K: =1 and MAX: =DATA [1]

Step 2: [REPEAT STEP 3 AND STEP 5] While K<=N

Step 3: [CONDITION] if K>N, then:

Write: LOC, MAX [END CONDITION]

[EXPLANATION]

Write: is used for printing.

Read: is for taking output.

Step 4: [CONDITION] if MAX<=DATA[K], then:

[SET]Set LOC: =K and MAX: =DATA[K] [END CONDITION]

Step 5: [INCREMENT COUNNTER] SET K: =K+1

Step 6: [END LOOP]

Step 7: [EXIT]

DRY RUN ALGORITHM:

Consider an array with DATA [5] = {15,20,19,13,14}, Remember index start from 1 if you talk about algorithm and N=5

We have to find the maximum number of element, so we need to draw trace table here.

K	K [VALUE]	MAX	MAX[VALUE]	While K<=N	If K>N	LOC	If MAX<=DATA[K]	MAX:=DATA[K]	K:=K+1	RESULT
1	15	1	15	TRUE	FALSE	1	TRUE	1	2	!FOUND
2	20	2	20	TRUE	TRUE	2	FALSE	``	3	FOUND
3	``	``	``	``	``	``	``	``	``	``
4	``	``	``	``	``	``	``	``	``	``
5	``	``	``	``	``	``	``	``	``	``

Here 20 is in [2] index of DATA[K], it is found as K>N (It traverse whole array but find maximum number it is found in the 2 [INDEX].

OPERATIONS:

- 1. **Insertion**
- 2. **Deletion**
- 3. **Traversing**
- 4. **Searching**
- 5. **Merging**
- 6. **Sorting**

1. PRIMITIVE DATA:

Primitive data types are the basic types of data that are directly supported by a programming language and are not composed of other types. They represent simple values like numbers, characters, and logical states.

1. Integer

Description: Stores whole numbers (positive, negative, or zero).

Examples:

int in C/C++, Java

int in Python

Examples of values: 10, -45, 0

2. Float (or Real Numbers)

Description: Stores decimal numbers or fractions.

Examples:

float, double in C/C++

float in Python

Examples of values: 3.14, -0.5, 2.718

3. Character

Description: Stores a single character.

Examples:

char in C/C++, Java

str (single-character strings) in Python

Examples of values: 'A', 'b', '\$', '3'

4. Boolean

Description: Stores logical values, either True or False.

Examples:

bool in Python, C++, Java

Examples of values: true, false

5. String

Description: Stores a sequence of characters. Although sometimes considered non-primitive, strings are treated as a basic data type in many languages.

Examples:

String in Java

char[] in C/C++

str in Python

Examples of values: "Hello", "123", "@world"

2. NON-PRIMITIVE DATA:

In C++, non-primitive data types are categorized into linear and non-linear structures based on their organization and relationships among elements. Here's an overview:

- a) **Linear Data Structures:** These structures organize data elements sequentially, where each element has a unique predecessor and successor. They are straightforward to implement and traverse. Common linear data structures include:
 - **Arrays: Fixed-size collections of elements of the same type.**

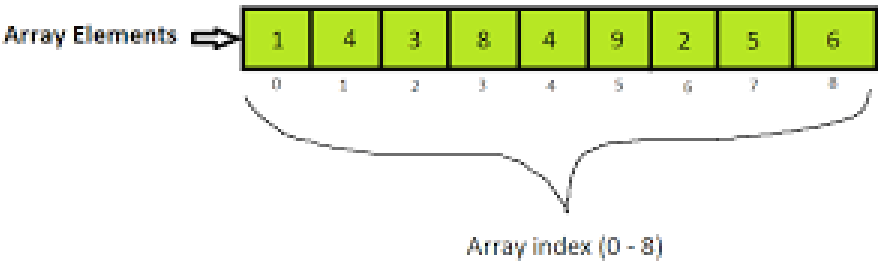


Fig 1.4:Arrays Data Structure

❑ TRAVERSING:

[Here, **LA** is a linear array with lower bound **LB**. The algorithm traverses **LA**, applying an operation **PROCESS** to each element of **LA**.]

1. Set $K := LB$ $K := LB$
2. Repeat steps 3 and 4 while $K \leq UB$ to UB
3. [Visit element] Apply **PROCESS** to $LA[K]$
4. [Increase counter] Set $K := K + 1$ $K := K + 1$ [End of step 2 loop]
3. Exit.

❑ INSERTION:

[A linear array **LA** with **N** elements, a positive integer **K**, and an item **ITEM** to insert at position **K**.
Output: The array **LA** with **ITEM** inserted at the **K**th position]

Set $J := N$ $J := N$

Repeat steps 3 and 4 while $J \geq K$ $J \geq K$:

[Move the **J**-th element downward] Set $LA[J] := LA[J - 1]$ $LA[J] := LA[J - 1]$

Decrease the counter: Set $J := J - 1$ $J := J - 1$

Insert the element: Set $LA[K] := ITEM$ $LA[K] := ITEM$

Update the size of the array: Set $N := N + 1$ $N := N + 1$

Exit.

❑ DELETION:

DELETION ALGORITHM: DELETE (LA, N, K, ITEM)

[**LA** is a linear array with **N** elements and **K** is $K \leq N$. This algorithm deletes **K**th element from **LA**].

1. Set $ITEM := LA[K]$
2. Repeat for to $N - 1$:
[Move the **J** + 1st element upward] Set $LA[J] := LA[J + 1]$
3. [Reset the number **N** of elements in **LA**] Set $N := N - 1$
4. Exit

❑ SEARCHING:

There are two type of searching.

- 1.Linear Search
- 2.Binary Search

1.Linear Search:

LINEAR SEARCH ALGORITHM: LINEAR (DATA, N, ITEM, LOC)

[A linear array **DATA** with elements and a specific **ITEM** of information This algorithm finds the location **LOC** of **ITEM** in the array **DATA**]

1. [Initialize] Set $K = 1$, $LOC = 0$. [Start searching from the first element and initialize the location to 0.]
2. Repeat Step 3 and 4 while $LOC = 0$ and $K \leq N$.
[Continue searching while the item is not found and there are elements left to check.]

- 3. IF ITEM = DATA[K], then:
- 4. Set LOC = K. [If the item is found, store its position in LOC.]
- 5. [Increment counter] Set K = K + 1. [Move to the next element in the array.][Successful]
- 6. If LOC = 0, then ITEM is not in the array DATA.
- 7. Else, print: ITEM is found at LOC. [Check if the item was found, and display the result.]
- 8. Exit. [End the search.]

2.Binary Search:

- 1. (Binary Search) BINARY(DATA, LB, UB, ITEM, LOC)
- 2. [initialize] Set BEG = LB, END = UB and MID = INT(BEG + END) / 2. [Initialize the beginning (BEG), end (END), and middle (MID) indices.]
- 3. Repeat steps 3 and 4 while BEG <= END and DATA[MID] ≠ ITEM.
[Continue searching as long as the item is not found and the search range is valid.]
- 4. If ITEM < DATA[MID],
[Reset END] Set END = MID - 1. [If the item is smaller than the middle element, adjust the search range to the lower half.]
- 5. Else:
[Reset BEG] Set BEG = MID + 1. [If the item is larger than the middle element, adjust the search range to the upper half.]
- 6. Set MID = INT(BEG + END) / 2. [Recalculate the middle index after adjusting the search range.]
- 7. If DATA[MID] = ITEM, then:
[Success] Set LOC = MID. [If the item is found, store the location in LOC.]
- 8. [Failure] Set LOC = NULL. [If the item is not found, set LOC to NULL.]
- 9. Exit. [End the search.]

❑ SORTING:

There are many type of sorting algorithms, we will have discussed here only two first is bubble sort and second is quick sort.

1. Bubble Sorting:

Algorithm: (Bubble sort) BUBBLE (DATA, N)

[Here DATA is an array with elements. This algorithm sorts the elements in DATA.]

- 1. Repeat Steps 2 and 3 for PASS = 1 to N - 1.
- 2. Set **POS := 1**. [Initialize pass pointer POS.]
- 3. Repeat while **POS < PASS**: [Executes pass.]
 - (a) If **DATA[POS] > DATA[POS + 1]**, then:
 - Interchange **DATA[POS]** and **DATA[POS + 1]**. [End of if structure.]
 - (b) Set **POS := POS + 1**.
[End of inner loop.]
- 4. [End of outer loop.]

2. Quick Sorting:

Algorithm 1.1: QUICK (A, N, BEG, END, LOC)

- 1. Set **LEFT := BEG, RIGHT := END, LOC := BEG**.
[Initialize the left and right pointers, and set LOC to BEG.]
- 2. **[Scan from right to left]**
 - a) Repeat While **A[RIGHT]** and **LOC ≠ RIGHT**:
- 3. **RIGHT := RIGHT - 1**.
[Move the right pointer until the condition is satisfied.]
- 4. b) If **LOC = RIGHT**, then:
[If LOC reaches RIGHT, exit the scanning process.]
- 5. **[Scan from left to right]**
 - c) If **A[LOC] > A[RIGHT]**, then:
 - a) Repeat While **A[LOC]**:
 - i. **A[LOC] < A[RIGHT]**.
 - ii. Set **LOC := RIGHT**.
[Scan from the left and check if LOC needs to be updated.]
- 6. b) If **LOC = LEFT**, then:
Return.
[If LOC reaches LEFT, the process is complete.]
- 7. c) If **A[LOC] < A[LEFT]**, then:
 - i. Swap **A[LOC]** and **A[LEFT]**.
 - ii. Set **LOC := LEFT**.
 - iii. Go to Step 2.
[If LOC is less than LEFT, swap the values and restart scanning from step 2.]

- **Linked Lists: Collections of nodes, where each node contains data and a reference to the next node.**

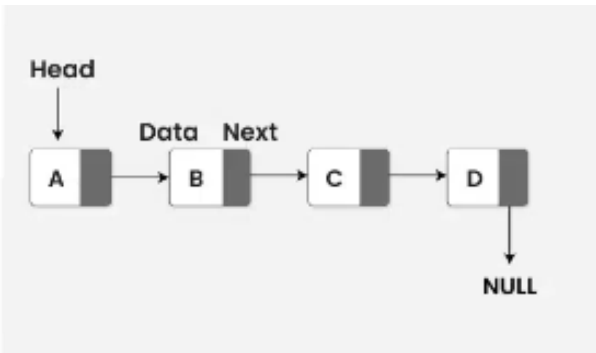


Fig 1.5: Linked List

There are three categories of linked list:

- 1.Linked list(Simple)
- 2.Double Linked list
- 3.Circular Linked list

1.Simple Linked List (Singly Linked List):

- **Structure:** Each node contains two parts:
 - **Data:** The value or information stored in the node.
 - **Next:** A pointer or reference to the next node in the list.
- **Direction:** In a singly linked list, traversal happens in one direction, from the first node (head) to the last node (tail).

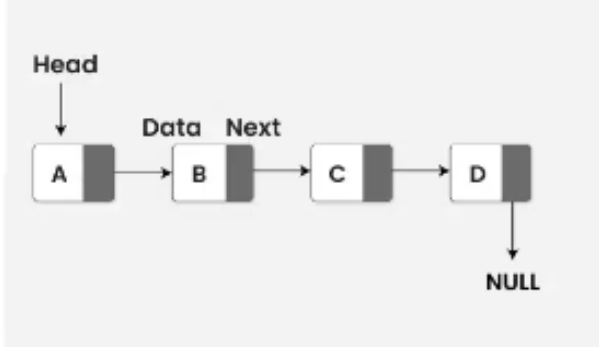


Fig 1.22:Simple Linked List

2,Doubly Linked List:

- **Structure:** Each node contains three parts:
 - **Data:** The value or information stored in the node.
 - **Next:** A pointer to the next node.
 - **Previous:** A pointer to the previous node.
- **Direction:** In a doubly linked list, you can traverse both forward and backward since each node has references to both the next and the previous nodes.



Fig 1.23:Doubly Linked list

3.Circular Linked List:

- **Structure:** A circular linked list can be either singly or doubly linked, but the key feature is that the last node points back to the first node, making the list circular.
 - **Singly Circular Linked List:** The last node's next pointer points to the first node, forming a loop.
 - **Doubly Circular Linked List:** The last node points to the first node, and the first node points back to the last node.
- **Direction:** Circular linked lists can be traversed starting from any node, and they allow looping through the list indefinitely.

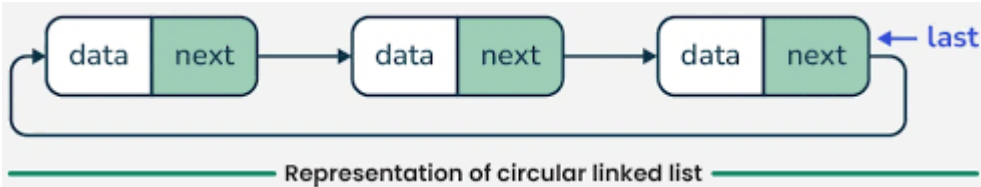


Fig 1.24:Circular Linked list

❑ TRAVERSING:

[Let List be a linked list in memory. This algorithm traverse list, applying process to each element of the list. The variable PTR pointer the currently being proceed]

1. Set PTR: =START
2. [Repeat step 3 and step 4] while PTR≠NULL
3. [Apply Process to info] [PTR]
4. Set PTR: =LINK[PTR][PTR now points to next node]
[End of step 2 loop]
5. Exit

❑ **SEARCHING:**

[Let List be a linked list in memory. This algorithm for searching the item from list]

1. Set PTR:=Start
2. [Repeat Step 3] while PTR≠NULL
3. If ITEM=INFO[PTR]:
Set Loc:=PTR [End Condition]
Else:
Set PTR:=LINK[PTR][PTR now points to next node]
[End of If structure]
[End of Loop]
4. Exit

❑ **DELETION:**

[This algorithm deletes the node N with location LOC,LOCP is the location of node which precedes node N or when N is the first node LOCP=NULL]

1. If LOCP=NULL
2. START:=LINK[START]
[Delete first node]
Else:
3. Set LINK[LOCP]:=LINK[LOC]
[END OF IF SRUCTURE]
[Return deleted node to Avail list]
4. Set LINK[LOC]:=AVAIL and AVAIL:=LOC
5. Exit

❑ **INSERTION:**

[Let List be a linked list in memory. This algorithm insertion the item from list]

[This algorithm inserts ITEM so that ITEM follows the node with location LOC or insert ITEM as the first node when Loc=NULL]

1. If AVAIL=NULL,then:
Write:”Overflow”[END CONDITION]
2. Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]
3. Set INFO[NEW]:=ITEM
4. If LOC=NULL
Set LINK[NEW]:=START and START:=NEW
Else:
Set LINK[NEW]:=LINK[LOC]
Set LINK[LOC]:=NEW
[End of If Structure]
5. Exit.

- **Stacks: Follow the Last-In-First-Out (LIFO) principle; elements are added and removed from the same end.**

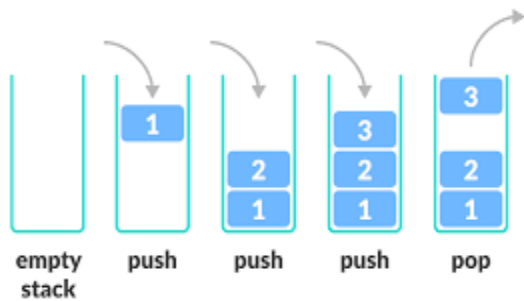


Fig 1.6:Stack

There are stwo operations in Stack:

- 1.Push
- 2.Pop

1.Push:

Algorithm - Push

Algorithm: PUSH(STACK, TOP, MAX, ITEM)

[This procedure pushes an ITEM onto a stack.]

Steps:

1. [Stack full] If TOP = MAX, then:
Write: OVERFLOW [Indicates that the stack is full and no more elements can be added.]
Return.
 2. [Increase TOP] Set TOP := TOP + 1. [Move the TOP pointer to the next position in the stack.]
 3. [Insert element] Set STACK[TOP] := ITEM. [Place the ITEM in the stack at the current TOP position.]
 4. Return. [End of the PUSH operation.]
- 2.POP:

Algorithm: POP (STACK, TOP, ITEM)

[This procedure deletes the top element of the stack and assigns it to variable ITEM.]

1. [Stack empty] If TOP = 0, then:
Write: UNDERFLOW, and return.
2. [Assign top element to ITEM]
Set ITEM = STACK[TOP].
3. [Decrease TOP by 1]
Set TOP := TOP - 1.
4. Return.

- **Queues: Follow the First-In-First-Out (FIFO) principle; elements are added at the rear and removed from the front.**



Fig 1.7:Queue

There are two operations in Queue:

- 1.Enqueue
- 2.Dequeue:

1.ENQUEUE:

Algorithm: Enqueue (QUEUE, N, F, R, ITEM)

[This procedure inserts an ITEM into the queue]

1. [Queue filled] If $F=1$ and $R=N$, or if $F=R+1$, then:

Write: OVERFLOW, and Exit.

2. [Find new value of R] if $F=NULL$, then:

Set $F := 1$ and $R := 1$.

[QUEUE was empty]

Else if $R = N$, then:

Set $R := 1$.

Else: $R := R + 1$.

3. [Insert element] Set $QUEUE[R] := ITEM$.

4. Exit.

2.DEQUEUE:

Algorithm: Dequeue (QUEUE, N, F, R, ITEM)

[Queue empty] If $F = NULL$, then:

Write: UNDERFLOW. [This procedure deletes an element from the queue and assigns it to ITEM.]

Exit.

[Extract the ITEM to be deleted] Set $ITEM := QUEUE[F]$. [Get the element at the front of the queue.]

[Find new value of F]

If $F = R$, then:

Set $F := NULL$ and $R := NULL$. [If the queue becomes empty after the dequeue operation, reset both F and R to NULL.]

Else if $F = N$, then:

Set $F := 1$. [If F reaches the end of the queue, reset F to the first position.]

Else:

Set $F := F + 1$. [Move F to the next position in the queue.]

Exit. [End of the dequeue operation.]

- b) **Non-Linear Data Structures:** These structures organize data elements in a hierarchical or interconnected manner, allowing for more complex relationships. Common non-linear data structures include:

- **Trees: Hierarchical structures with a root element and sub-elements (children) forming a parent-child relationship.**

Here we will have discussed, trees data structure.

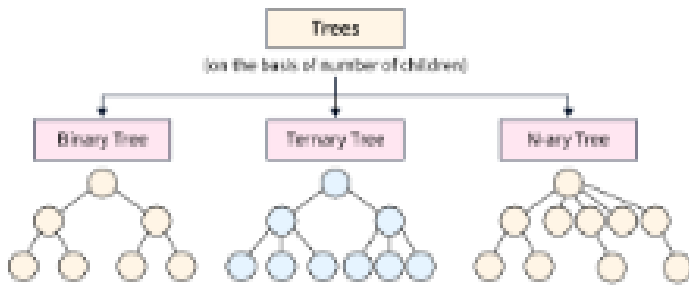


Fig 1.14: Types of trees

□ Binary Tree:

- A tree in which each node has at most two children (left and right).
- Used in various algorithms like searching, sorting, and decision-making.

□ Binary Search Tree (BST):

- A binary tree where for every node, the left child has a smaller value, and the right child has a larger value.
- Used for efficient searching, insertion, and deletion operations.

□ Balanced Binary Tree:

- A binary tree where the height of the left and right subtrees of any node differ by at most one.
- Examples: AVL tree, Red-Black tree.
- Ensures that operations like searching, insertion, and deletion are efficient.

□ Heap:

- A complete binary tree that satisfies the heap property (either max-heap or min-heap).
- Used in priority queues and sorting algorithms (like heapsort).

□ AVL Tree:

- A self-balancing binary search tree where the difference in heights between the left and right subtrees of any node is at most 1.
- Provides faster lookup times by keeping the tree balanced.

- **Red-Black Tree:**
 - A self-balancing binary search tree with an additional property that each node is either red or black.
 - Provides balanced performance for insertion, deletion, and search operations.
 - **B-tree:**
 - A balanced tree used in databases and file systems for efficient data retrieval.
 - It is a self-balancing tree where nodes can have multiple children (more than two).
 - **Trie:**
 - A tree-like data structure used for storing strings where nodes represent characters.
 - Commonly used for fast prefix matching and dictionary implementations.
 - **Segment Tree:**
 - A binary tree used for storing intervals or segments.
 - Efficient for querying and updating intervals or ranges.
 - **Suffix Tree:**
 - A compressed tree used for storing all the suffixes of a given string.
 - Useful in string processing tasks like pattern matching.
 - **N-ary Tree:**
 - A tree where each node can have at most **N** children.
 - Useful for hierarchical data structures like file systems.

■ Classification of Binary tree:

Based on Structure:

- **Full Binary Tree:**
 - A binary tree in which every node has either 0 or 2 children.
 - Example: A tree where all non-leaf nodes have exactly two children.

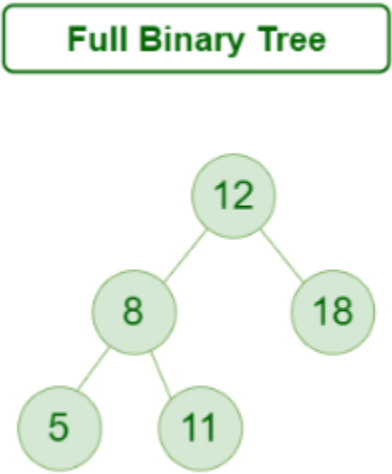


Figure 1.15:Full Binary Tree

- **Complete Binary Tree:**
 - A binary tree where all levels, except possibly the last, are completely filled, and all nodes are as far left as possible.
 - Example: A tree with nodes filled level by level from left to right.

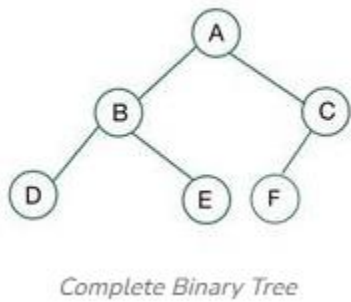


Figure 1.16: Complete Binary Tree

- **Perfect Binary Tree:**
 - A binary tree in which all interior nodes have exactly two children, and all leaf nodes are at the same level.
 - Example: A binary tree where all levels are completely filled, with no missing nodes at any level.

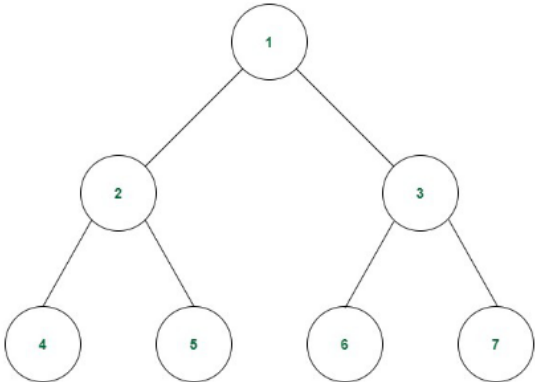


Figure 1.17: Perfect Binary Tree

- **Degenerate (or Pathological) Binary Tree:**
 - A binary tree in which each parent node has only one child, making it essentially a linked list.
 - Example: A tree where every node only has one child (either left or right).

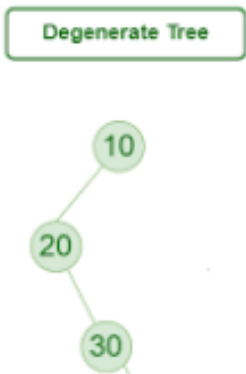


Figure 1.18: Degenerate Tree

Basic Terminologies In Tree Data Structure:

- ❑ **Parent Node:** The node which is an immediate predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.
- ❑ **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.
- ❑ **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- ❑ **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {I, J, K, F, G, H} are the leaf nodes of the tree.
- ❑ **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A,B} are the ancestor nodes of the node {E}
- ❑ **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of x.
- ❑ **Sibling:** Children of the same parent node are called siblings. {D,E} are called siblings.
- ❑ **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
- ❑ **Internal node:** A node with at least one child is called Internal Node.
- ❑ **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- ❑ **Subtree:** Any node of the tree along with its descendant.

- **Graphs: Collections of nodes (vertices) connected by edges, representing complex relationships.**

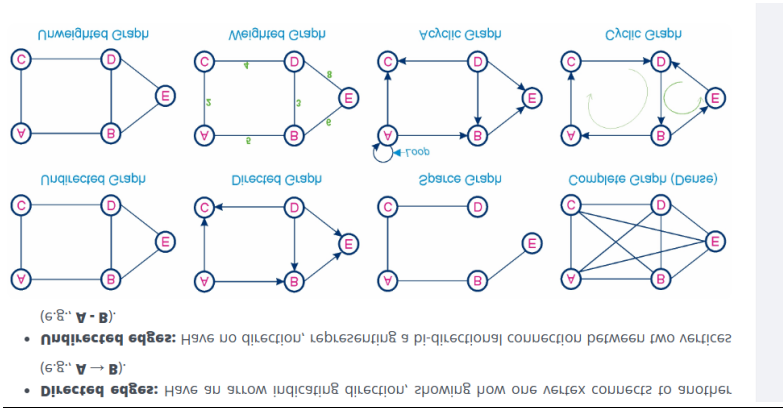


Figure 1.19: Types of Graphs:

1. Graph: A collection of nodes (vertices) and edges (connections between the nodes).
2. Vertex (Node): A point or a position in a graph, often represented as a circle.
3. Edge: A connection between two vertices. It can be directed or undirected.
4. Directed Graph (Digraph): A graph where the edges have a direction, meaning they go from one vertex to another.
5. Undirected Graph: A graph where the edges do not have a direction, meaning the connection between two vertices is mutual.
6. Degree: The number of edges connected to a vertex. In a directed graph, there are two types:
7. In-degree: Number of incoming edges to a vertex.
8. Out-degree: Number of outgoing edges from a vertex.
9. Path: A sequence of vertices connected by edges.
10. Cycle: A path that starts and ends at the same vertex without repeating any other vertices.
11. Connected Graph: A graph in which there is a path between every pair of vertices.
12. Disconnected Graph: A graph that is not connected, meaning there are at least two vertices without a path between them.
13. Subgraph: A graph formed from a subset of the vertices and edges of another graph.
14. Tree: A connected acyclic graph (no cycles).
15. Spanning Tree: A tree that includes all the vertices of the graph.
16. Weighted Graph: A graph in which each edge has a weight or cost associated with it.
17. Neighbor: A vertex that is directly connected to another vertex by an edge.
18. Adjacency: Two vertices are adjacent if there is an edge between them.

- 19. Isolated Vertex: A vertex with no edges connected to it.
- 20. Complete Graph: A graph in which there is an edge between every pair of vertices.
- 21. Bipartite Graph: A graph whose vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set.
- 22. Planar Graph: A graph that can be drawn on a plane without edges crossing each other.

Advanced Graph Terminology:

1. [Directed Graph \(Digraph\)](#):

A Directed Graph consists of nodes (vertices) connected by directed edges (arcs). Each edge has a specific direction, meaning it goes from one node to another. Directed Graph is a network where information flows in a specific order. Examples include social media follower relationships, web page links, and transportation routes with one-way streets.

2. [Undirected Graph](#):

In an Undirected Graph, edges have no direction. They simply connect nodes without any inherent order. For example, a social network where friendships exist between people, or a map of cities connected by roads (where traffic can flow in both directions).

3. [Weighted Graph](#):

Weighted graphs assign numerical values (weights) to edges. These weights represent some property associated with the connection between nodes. For example, road networks with varying distances between cities, or airline routes with different flight durations, are examples of weighted graphs.

4. [Unweighted Graph](#):

An unweighted graph has no edge weights. It focuses solely on connectivity between nodes. For example: a simple social network where friendships exist without any additional information, or a family tree connecting relatives.

5. [Connected Graph](#):

A graph is connected if there is a path between any pair of nodes. In other words, you can reach any node from any other node. Even a single-node graph is considered connected. For larger graphs, there’s always a way to move from one node to another.

6. [Acyclic Graph](#):

An acyclic graph contains no cycles (closed loops). In other words, you cannot start at a node and follow edges to return to the same node. Examples include family trees (without marriages between relatives) or dependency graphs in software development.

7. [Cyclic Graph](#):

A cyclic graph has at least one cycle. You can traverse edges and eventually return to the same node. For example: circular road system or a sequence of events that repeats indefinitely.

8. Connected Graph

A Graph is connected if there is a **path between every pair of vertices** in the graph. In a directed graph, the concept of strong connectivity refers to the existence of a directed path between every pair of vertices.

9. Disconnected Graph:

A disconnected graph has isolated components that are not connected to each other. These components are separate subgraphs.

■ **STRING MANIPULATION:**

Here ,we will discussed two types of algorithms:

- 1.Bruteforce
- 2.Finite Automata

1.**B**ruteforce:

[P and T are strings with lengths R and S respectively,and are stored as array with one character per element.This algorithm finds the INDEX of P in T].

- 1. **[initialize]** Set **K := 1** and **MAXS := R + 1**. [Start the search from the first character and set the maximum limit for searching.]
- 2. **Repeat** **steps** **3** **to** **5** **while** **K** **<** **MAXS.**
[Continue searching while there are characters left in the text.]
- 3. **Repeat for L = 1 to R:**
- 4. If **P[L] = T[K + L]**, then go to step 5. [Check if the current character of **P** matches the corresponding character in **T**.]
- 5. **[Success]** Set **INDEX := K** and Exit. [If the pattern is found, store the index and stop.]
- 6. **Set K := K + 1**. [Move to the next character in **T**.]
- 7. **[Failure]** Set **INDEX := 0**. [If the pattern is not found, return 0 as the index.]
- 8. **Exit**. [End the algorithm.]

2.**F**inite Automata:

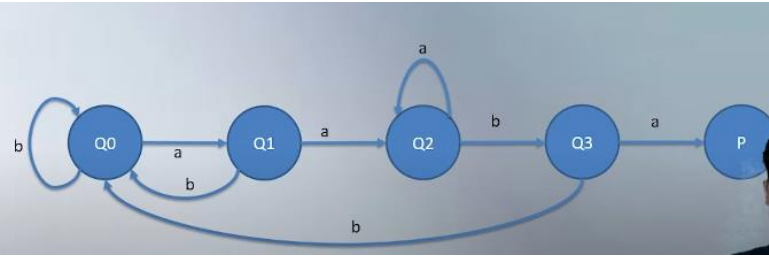


Fig1.8: FINITE AUTOMATA

Step 1 Set $K := 1$ and $S_1 = Q_0$ [Initialization]
Step 2 Repeat steps 3 to 5 while $S_k \neq P$ and $K \neq N$
Step 3 Read T_k
Step 4 Set $S_{k+1} := F(S_k, T_k)$ [Finding next state]
Step 5 Set $K := K + 1$ [Update counter]
[End of Step 2 loop]
Step 6 If $S_k = P$ then
 Set $INDEX := K - LENGTH(P)$ [Successful]
Else:
 Set $INDEX := 0$ [Unsuccessful]
Step 7 Exit

Fig 1.9: Algorithm

	<i>a</i>	<i>b</i>
<i>Q0</i>	Q1	Q0
<i>Q1</i>	Q2	Q0
<i>Q2</i>	Q2	Q3
<i>Q3</i>	P	Q0

Fig 1.10: Table

■ **RECURSION:**

```
int factorial(int n) {  
    if(n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

Fig 1.11:Factorial

Here we will have discussed recursion. The difference between iteration and recursion is that in iteration there is a repetition control flow and in recursion we need function to call itself, it is a stack example.
Procedure: FACTORIAL (FACT, N)

This procedure calculates **N!** and returns the value in variable **FACT**.
Steps:

1. If **N = 0**, then Set **FACT := 1**. [Base case: Factorial of 0 is 1.]
2. Set **FACT := 1**. [Initialize FACT to 1 before calculation.]
3. Repeat for **K = 1 to N**, Set **FACT := FACT * K**. [Loop through the numbers from 1 to N, multiplying each value to calculate the factorial.]
4. Return. [End of procedure.]

Tower of Honie is one of the best example of recursion.

Procedure: TOWER (N, SRC, DEST, AUX)
This procedure gives a recursive solution to the Towers of Hanoi problem for N disks.

1. **If N = 1, then:**
 - a) **Write: SRC to DEST.** [Move the disk from the source peg to the destination peg.]
 - b) **Return.** [Stop the procedure when only one disk remains.]
2. **Call TOWER (N-1, SRC, AUX, DEST).** [Move N-1 disks from the source peg to the auxiliary peg.]
3. **Write: SRC to DEST.** [Move the N-th disk from the source peg to the destination peg.]
4. **Call TOWER (N-1, AUX, SRC, DEST).** [Move N-1 disks from the auxiliary peg to the destination peg.]
5. **Return.** [End the procedure.]

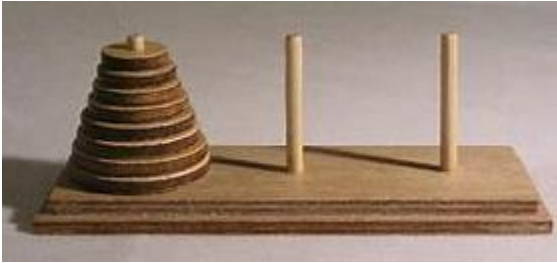


Fig 1.12: Tower of Honai

STACK APPLICATION:

In Data Structures and Algorithms (DSA), **prefix**, **infix**, and **postfix** are notations used to represent expressions (like arithmetic expressions). These notations describe the position of the operator in the expression relative to its operands. Let me explain them with simple examples:

1. Prefix Notation (Polish Notation)

- The operator is written **before** its operands.
- Example: + 3 5 (means 3 + 5)

To evaluate:

1. Start from the right and find an operator with two operands.
2. Perform the operation.
3. Replace it in the expression and continue.

Expression:

+ * 2 3 4

- Step 1: $*\ 2\ 3 = 6$
- Step 2: $+\ 6\ 4 = 10$

2. Postfix Notation (Reverse Polish Notation)

- The operator is written **after** its operands.
- Example: $3\ 5\ +$ (means $3 + 5$)

To evaluate:

1. Scan from left to right.
2. Push operands onto a stack.
3. When an operator appears, pop the top two elements, perform the operation, and push the result back onto the stack.

Expression:

2 3 * 4 +

- Step 1: $2\ *\ 3 = 6$
- Step 2: $6\ +\ 4 = 10$

3. Infix Notation

- The operator is written **between** its operands (this is how we normally write expressions).
- Example: $3 + 5$

This notation is easiest for humans to read but can be ambiguous for computers, which is why parentheses or operator precedence rules are required.

Expression:

2 * 3 + 4

- Using precedence, evaluate $2\ *\ 3 = 6$.
- Then, $6\ +\ 4 = 10$.

Why Do We Use Prefix/Postfix?

- Computers and compilers use prefix or postfix notations because they remove ambiguity and don't require parentheses to denote precedence.
- **Postfix** is particularly useful in stack-based computations, as it's easier to evaluate with a stack.

Expression Example: 2 3 + 4 *

Step	Symbol	Stack (Action Taken)	Stack (Current State)
1	2	Push 2 onto the stack.	[2]
2	3	Push 3 onto the stack.	[2, 3]
3	+	Pop 2 and 3, calculate $2 + 3 = 5$, and push 5.	[5]
4	4	Push 4 onto the stack.	[5, 4]
5	*	Pop 5 and 4, calculate $5 * 4 = 20$, and push 20.	[20]

Example: Convert Infix to Postfix

Infix Expression: A + B * C - D

Step	Symbol	Stack	Postfix Expression
1	((
2	A	(A
3	+	(+	A
4	B	(+	A B
5	*	(+ *	A B
6	C	(+ *	A B C
7	-	(-	A B C * +
8	D	-	A B C * + D
9)		ABC*+D-

Final Postfix: A B C * + D -

highest to lowest priority:

- 1. ()` (Parentheses)
- 2. ^ (Exponentiation)
- 3. *, / (Multiplication and Division)
- 4. +, - (Addition and Subtraction)

EXPRESION TREE EXAMPLE:

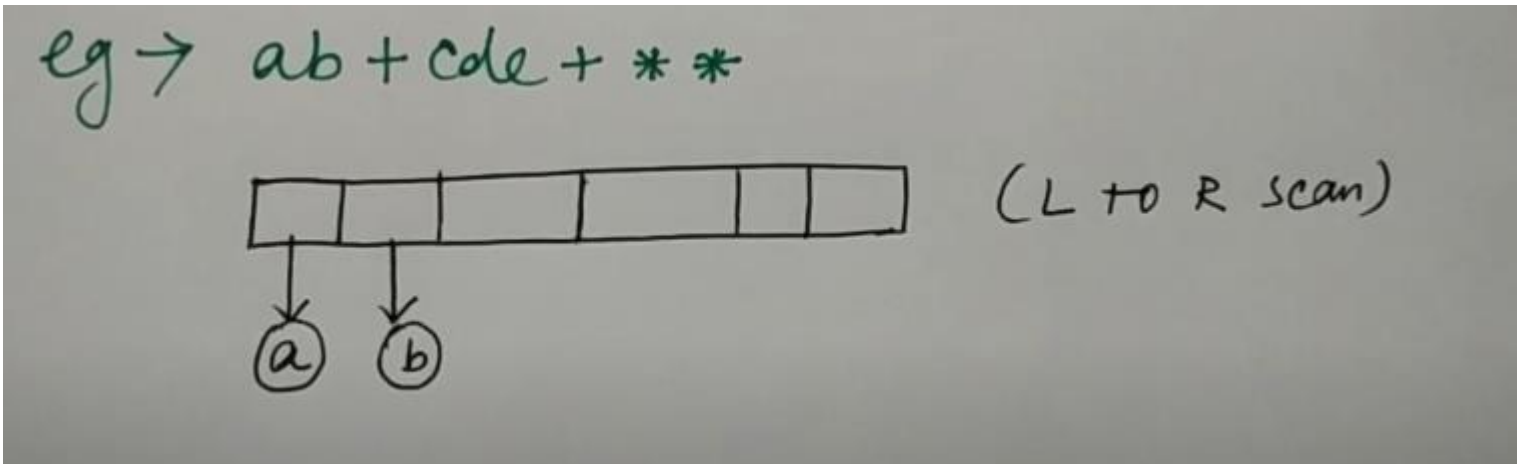


Fig 1.A:Expression Tree

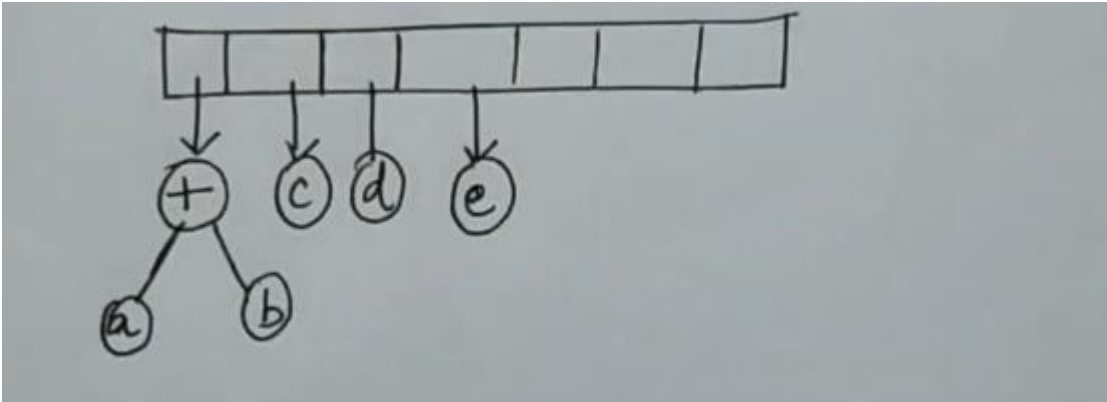


Fig 1.B:Expression Tree

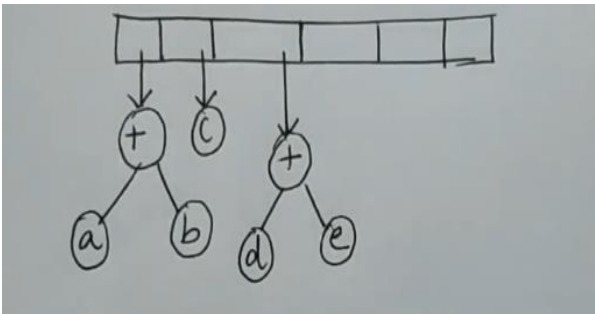


Fig 1.C:Expression Tree

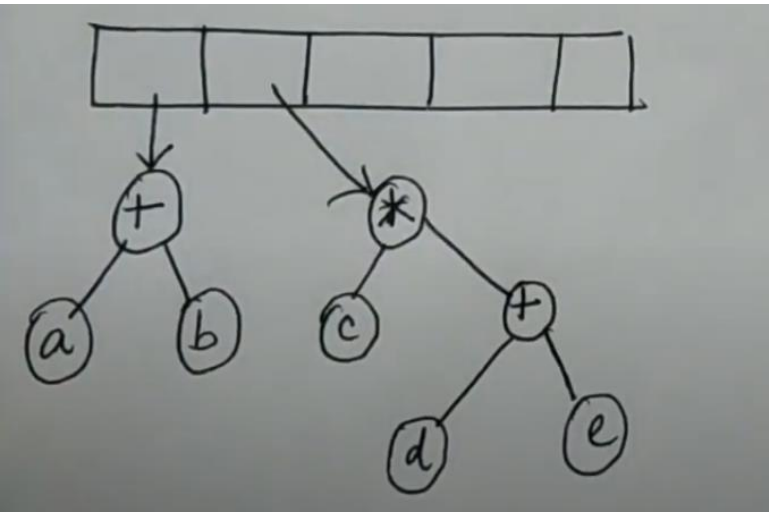


Fig 1.D:Expression Tree

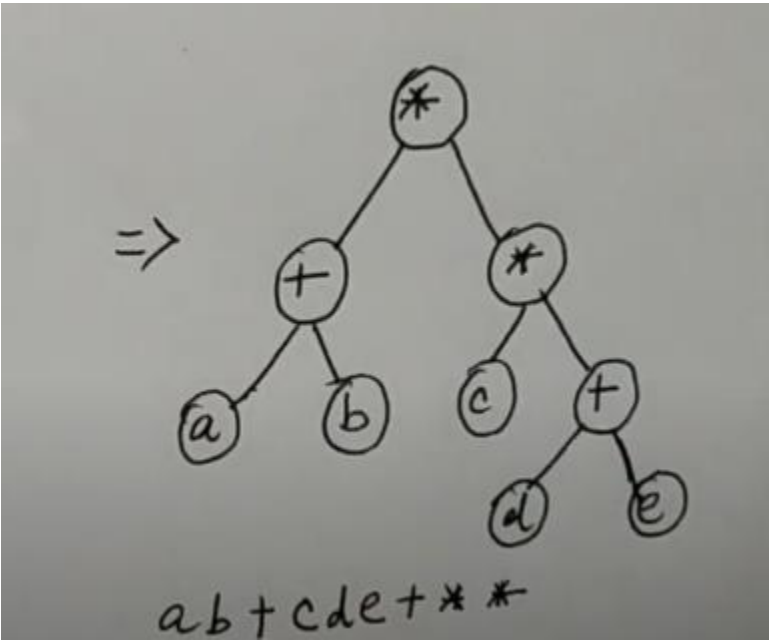


Fig 1.E:Expression Tree

Heap:

A **Heap** is a complete binary tree data structure that satisfies the heap property: for every node, the value of its children is greater than or equal to its own value. Heaps are usually used to implement priority queues, where the smallest (or largest) element is always at the root of the tree.

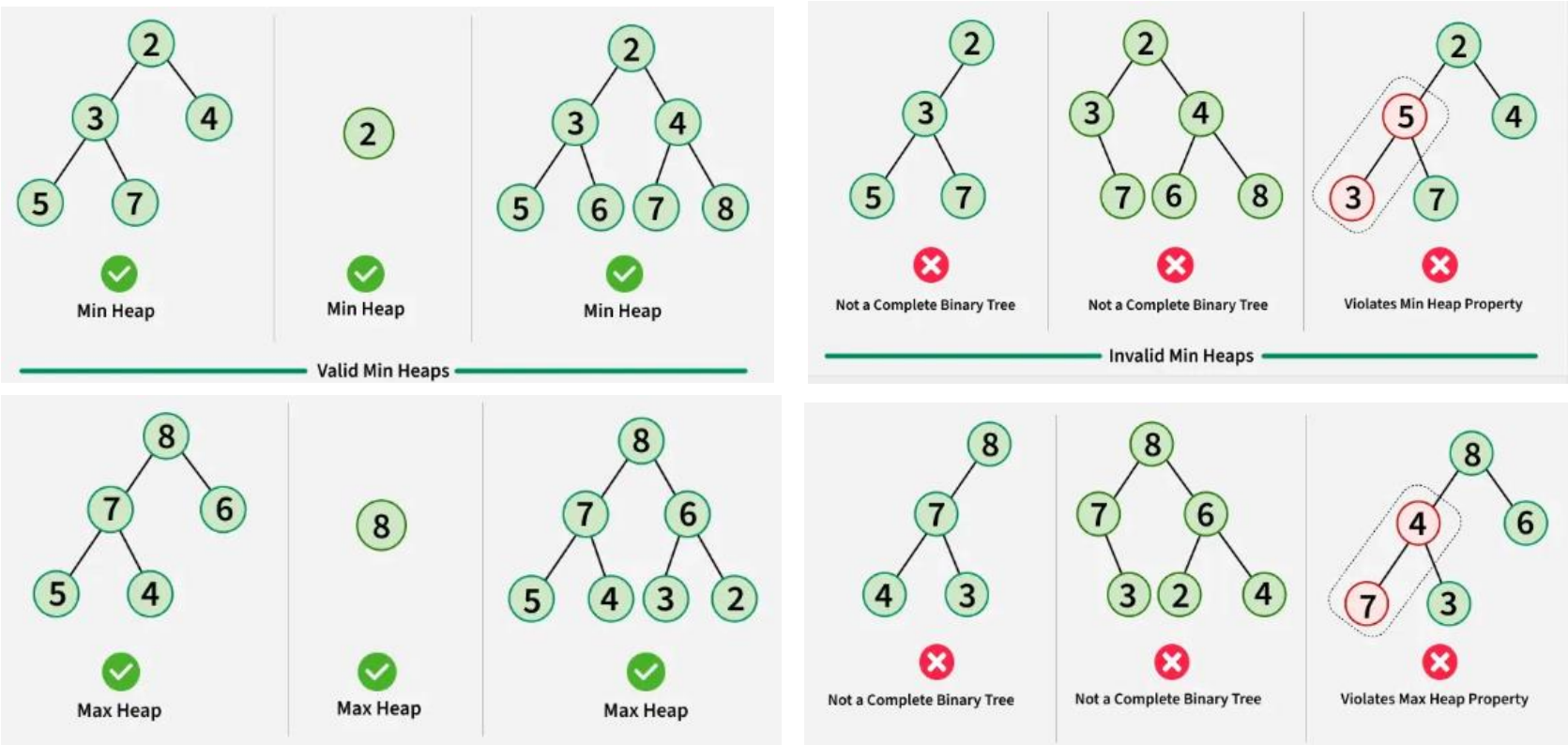


Fig 2.1: Heap