DATA STRUCTURE NOTES

Unit 1: Introduction

Data Structure aur uske Types:

- ❖ Data Structure ek aisa tareeka hai jisme hum data ko organize, store aur manage karte hain taaki use efficiently access aur modify kiya ja sake.
- ❖ Data ko alag-alag ways me organize kiya ja sakta hai; kisi specific organization ka logical ya mathematical model hi **Data Structure** kehlata hai.
- ❖ Ye ek framework hota hai jo decide karta hai ki memory me data kahan aur kis form me store hoga taaki operations jaise searching, inserting, deleting aur updating fast ho sake.
- ❖ Ye data aur operations ke beech ka relationship define karta hai.

Operations on Data Structure:

1. **Insertion:** Naya element add karna.
2. **Deletion:** Koi element remove karna.
3. **Traversing:** Har element ko ek-ek karke access karna.
4. **Searching:** Kisi element ka location find karna.
5. **Sorting:** Elements ko ek order me arrange karna.

Types of Data Structure:

1. Primitive Data Structures

Ye basic data types hote hain jo programming languages provide karti hain:

- **Integer:** Whole numbers store karne ke liye.
- **Float/Double:** Decimal numbers ke liye.
- **Character:** Single character store karne ke liye.
- **Boolean:** True/False values ke liye.
- **Pointer:** Memory address store karne ke liye.

---

2. Non-Primitive Data Structures

Ye complex hote hain aur primitive types se banaye jaate hain.

A. Linear Data Structures

Elements sequentially store hote hain.

a) Array

- Same type ke elements ka collection.
- Contiguous memory me store hote hain.
- Example: int arr[5] = {1, 2, 3, 4, 5};

b) Linked List
Ye ek linear data structure hai jisme elements continuous memory me nahi store hote.
Har node me **data + next node ka address (pointer)** hota hai.

Types:

- Singly Linked List → har node sirf next node ka address rakhta hai.
- Doubly Linked List → har node previous aur next dono ka address rakhta hai.
- Circular Linked List → last node dobara head node ki taraf point karta hai.

Advantages:

- Memory ka better use.
- Insertion aur deletion fast aur flexible.

Disadvantages:

- Random access possible nahi.
- Extra memory pointers ke liye chahiye.

c) Stack

- Stack LIFO (Last In, First Out) principle par kaam karta hai.
- Matlab jo element last me gaya wahi sabse pehle niklega. (Jaise thaliyon ka stack).

Operations:

- Push(x) → Top par element add karna.
- Pop() → Top element remove karna.
- Peek()/Top() → Sirf top element dekhna bina remove kiye.
- isEmpty() → Check karna stack empty hai ya nahi.

Important Terms:

- Top pointer
- Overflow (jab stack full ho)

- Underflow (jab stack empty ho)

Example:
Push(10), Push(20), Push(30)
Stack: [30, 20, 10]
Pop() → 30 niklega.

d) Queue

- Queue FIFO (First In, First Out) principle par kaam karta hai.
- Matlab jo element pehle aaya wahi pehle niklega (jaise bus stand ki line).

Operations:

- Enqueue(x) → Element ko rear (end) me add karna.
- Dequeue() → Element ko front se remove karna.
- Peek()/Front() → Sirf front element dekhna bina remove kiye.
- isEmpty() → Check karna queue empty hai ya nahi.

Types of Queue:

- Simple Queue
- Circular Queue
- Priority Queue
- Deque (Double Ended Queue)

B. Non-Linear Data Structures

Elements sequentially store nahi hote.

a) Tree
Tree ek hierarchical structure hai jo nodes se banta hai. Real life tree ki tarah hota hai, bas iska root upar hota hai aur branches neeche.

Concepts:

- Node → Ek element.
- Root Node → Sabse top wala node.
- Parent aur Child Nodes → Jo node kisi ko point kare wo parent, jisko point kiya gaya wo child.
- Leaf Node → Jinke aage koi child nahi.
- Edge → Parent aur child ko connect karne wali line.
- Height/Depth → Root se leaf tak ki lambi path / root se kisi node tak ki distance.

Types of Tree:

- General Tree
- Binary Tree
- Binary Search Tree (BST)
- AVL / Red-Black Tree (Self balancing)
- Heap Tree

Uses:

- File systems
- Databases me indexing
- Searching aur sorting
- Decision making (AI)
- Network routing

C. Hash-Based Structures

- Hash Table / Hash Map
- Data key-value pairs me store hota hai using hash function.
- Search, Insert aur Delete operations bahut fast hote hain.

D. Specialized Data Structures

- **Trie:** Prefix based search (autocomplete).
- **Heap:** Priority queues implement karne ke liye.
- **Disjoint Set (Union-Find):** Graph algorithms me use hota hai.
- **Segment Tree / Fenwick Tree:** Arrays ke range queries ke liye.

Function of data structure:

It is **Organize, manage, and store data efficiently** to enable easy access and modification.

Primary Functions of Data Structures

1. **Efficient Data Storage**
    - Allows data to be stored in a way that optimizes space and access time.
    - Example: Arrays store elements in contiguous memory, making indexing fast.

2. **Easy Data Access**
   - Facilitates retrieving data quickly and efficiently.
   - Ex: Hash tables provide constant-time access (O(1)) to values using keys.
3. **Data Manipulation**
   - Enables operations such as insertion, deletion, updating, and searching.
   - Example: Linked lists allow efficient insertion and deletion.
4. **Data Organization**
   - Helps arrange data logically (e.g., hierarchical, sequential, or associative).
   - Example: Trees organize data in a hierarchical structure, useful for databases.
5. **Data Abstraction**
   - Provides a clear interface to work with data while hiding implementation details.
   - Example: Stacks and queues abstract common use cases like backtracking or scheduling.
6. **Improved Algorithm Efficiency**
   - Supports faster and more optimized algorithms.
   - Example: Binary search trees speed up search operations compared to linear structures.

Examples of Common Data Structures and Their Functions

Data Structure	**Function/Use**
Array	Fixed-size, fast index-based access
Linked List	Dynamic size, efficient insertion/deletion
Stack	Last-In-First-Out (LIFO) operations
Queue	First-In-First-Out (FIFO) operations
Hash Table	Fast lookup via key-value pairs
Tree	Hierarchical data representation (e.g., file systems)
Graph	Representing networks (e.g., social media, routes)

Memory Allocation:

It refers to how memory is reserved and managed when creating and using data structures like arrays, linked lists, stacks, queues, trees, etc. There are two main types of memory allocation in data structures:

🔷 1. Static Memory Allocation

- Memory is allocated **at compile-time**.
- Size and type of data structures are fixed.
- Examples: Arrays in C/C++.

Pros:

- Faster access due to fixed size.
- No memory fragmentation.

Cons:

- Wastes memory if the allocated size is too big.
- Cannot handle dynamic size changes.

Example (C):

int arr[10]; Allocates memory for 10 integers at compile time

---

🔷 2. Dynamic Memory Allocation

- Memory is allocated **at run-time**.
- Size can be changed during execution.
- Examples: Linked lists, dynamic arrays, trees.

Pros:

- Efficient use of memory.
- Data structures can grow or shrink as needed.

Cons:

- Slightly slower due to overhead.
- Must manage memory manually (e.g., using free() in C).

Functions in C:

- malloc() – allocate memory
- calloc() – allocate and initialize memory

- realloc() – reallocate memory
- free() – free allocated memory

Example (C):

int* ptr = (int*) malloc(10 * size of(int)); Allocates memory for 10 integers

---

Memory Allocation in Common Data Structures

Data Structure	Allocation Type	Description
Array	Static or Dynamic	Fixed size (static), or dynamic with malloc
Linked List	Dynamic	Nodes are created dynamically
Stack	Static or Dynamic	Can use array (static) or linked list (dynamic)
Queue	Static or Dynamic	Similar to stack
Tree / Graph	Dynamic	Nodes/edges created dynamically

Array:

Single Dimensional Array:

Single dimensional array ek **linear data structure** hai jisme ek hi row me elements ko **contiguous memory locations** me store kiya jata hai.

Iska matlab hai ki array ke saare elements ek ke baad ek memory me rakhe jaate hain.

Definition:

Single dimensional array ek **list of elements of same data type** hoti hai, jise ek hi index (position number) se access kiya ja sakta hai.

Ex. int arr[5] = {10, 20, 30, 40, 50};

- Yahan arr ek single dimensional array hai jisme **5 integer elements** store hain.
- Index **0 se start hota hai**:

- arr[0] = 10
- arr[1] = 20
- arr[2] = 30
- arr[3] = 40

- arr[4] = 50

Characteristics:

1. **Fixed size** – ek baar array ka size decide ho gaya toh change nahi hota.
2. **Same data type** – saare elements ek hi type ke hone chahiye (int, char, float, etc).
3. **Random access** – kisi bhi element ko uske index se directly access kiya ja sakta hai.
4. **Contiguous memory allocation** – array ke elements ek ke baad ek memory me stored hote hain.

Multi-Dimensional Array:

multi-dimensional array ko ek **table (2D)** ya **cube (3D)** ki tarah visualize kiya ja sakta hai. Ye bhi **same data type ke elements ka collection** hota hai, par isme elements ko **multiple indexes** ke through access kiya jata hai.

1. Two-Dimensional Array (2D Array)

- Isay aap **rows aur columns wali table** ki tarah soch sakte hain.
- Har element ko **2 indexes** ki madad se access kiya jata hai: arr[row][column].

```
Ex. int arr[2][3] = {
  {10, 20, 30},
  {40, 50, 60}
};
```

2. Three-Dimensional Array (3D Array)

- Isay ek **cube** ki tarah visualize kiya jata hai — multiple tables stacked together.
- Har element ko **3 indexes** se access kiya jata hai: arr[x][y][z].

```
Ex. int arr[2][2][3] = {
  {
    {1, 2, 3},
    {4, 5, 6}
  },
  {
    {7, 8, 9},
    {10,11,12}
  }
```

};

Sparse matrices:

Aisi matrix jisme **zyada tar elements zero hote hain** aur sirf **kuch hi non-zero elements** hote hain.

Agar kisi matrix me **zero elements ki sankhya > non-zero elements**, to use sparse matrix bolte hain.

Ex. 4x5 matrix:

0 0 0 5 0

0 8 0 0 0

0 0 0 0 0

0 6 0 0 0

Sparse Matrix ek **special type ka 2D matrix** hota hai jisme **zyadatar elements zero hote hain**. Normal matrix representation (2D array) use karne se memory waste hoti hai, isliye **special representation** use kiya jata hai.

Sparse Matrix

A matrix jisme **non-zero elements kam hote hain aur zero elements jyada**, usse sparse matrix kehte hain.
Agar matrix me total elements m × n hain aur non-zero elements ki ginti T hai, aur T << (m × n) hai, tab sparse matrix bola jata hai.

Representation of Sparse Matrix

1. Array Representation (Triplet Form / 3-tuple form)

- Ek 2D array banta hai jisme 3 columns hote hain:
  - Row number
  - Column number
  - Value (non-zero element)

Ex.

Matrix:

0 0 3

0 0 0

4 0 0

0 5 0

Triplet Representation:

```
Row Col Value
0   2   3
2   0   4
3   1   5
```

Aksar pehli row me metadata store karte hain (rows, cols, non-zeros):

```
[4 3 3]   ← (rows=4, cols=3, nonzero=3)
[0 2 3]
[2 0 4]
[3 1 5]
```

Pros: Easy to implement, less memory than full 2D array

Cons: Insertion/Deletion thoda costly

2. Linked List Representation

Har non-zero element ek node me store hota hai.

Node structure:

```
struct Node {
   int row;
   int col;
   int value;
   Node* next;
};
```

Har node me us element ka (row, col, value) aur ek pointer next non-zero node ki taraf hota hai.

Example ke liye upar wale matrix ka linked list:

(0,2,3) → (2,0,4) → (3,1,5)

Pros: Insertion/Deletion fast, flexible

Cons: Extra memory chahiye (pointers ke liye), thoda complex

Jab Use Karte Hain Sparse Matrix?

I. Graphs (Adjacency Matrix sparse hoti hai)
II. Machine Learning me (high-dimension data)
III. Image processing, optimization problems

1. Array Representation (Triplet Form)

```
#include <stdio.h>
#define MAX 100
int main() {
int rows, cols;
int mat[10][10];
int sparse[MAX][3];  // Triplet form
int i, j, k = 1, nonZero = 0;
printf("Enter rows and cols of matrix: ");
scanf("%d %d", &rows, &cols);
printf("Enter matrix elements:\n");
for (i = 0; i < rows; i++) {
for (j = 0; j < cols; j++) {
      scanf("%d", &mat[i][j]);
      if (mat[i][j] != 0)
        nonZero++;
    }
  }
  // First row: metadata
  sparse[0][0] = rows;
  sparse[0][1] = cols;
  sparse[0][2] = nonZero;
  // Store non-zero elements
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      if (mat[i][j] != 0) {
        sparse[k][0] = i;
        sparse[k][1] = j;
        sparse[k][2] = mat[i][j];
        k++;
      }
```

```
        }
    }
    // Print Triplet form
    printf("\nSparse Matrix (Triplet Form):\n");
    for (i = 0; i <= nonZero; i++) {
        printf("%d\t%d\t%d\n", sparse[i][0], sparse[i][1], sparse[i][2]);
    }
    return 0;
}
```

2. Linked List Representation

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int row, col, val;
    struct Node* next;
};
struct Node* createNode(int r, int c, int v) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->row = r;
    newNode->col = c;
    newNode->val = v;
    newNode->next = NULL;
    return newNode;
}
void display(struct Node* head) {
    struct Node* temp = head;
    printf("\nSparse Matrix (Linked List Form):\n");
    while (temp != NULL) {
        printf("Row=%d Col=%d Value=%d\n", temp->row, temp->col, temp->val);
        temp = temp->next;
    }
}
int main() {
    int rows, cols, i, j, val;
    int mat[10][10];
    struct Node* head = NULL;
    struct Node* tail = NULL;
    printf("Enter rows and cols of matrix: ");
    scanf("%d %d", &rows, &cols);
    printf("Enter matrix elements:\n");
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            scanf("%d", &val);
            mat[i][j] = val;
            if (val != 0) {
                struct Node* newNode = createNode(i, j, val);
                if (head == NULL) {
                    head = tail = newNode;
                } else {
                    tail->next = newNode;
                    tail = newNode;
                }
            }
        }
    }
    display(head);
    return 0;
}
```

Stack Implementing Single/Multiple Stacks in an Array:

🔷 1. Single Stack using Array

Stack ek **LIFO (Last In First Out)** data structure hota hai.
Array ke through stack banane ke liye humein 3 cheezen chahiye:

1. **Array** (jisme elements store karenge)
2. **Top pointer/index** (jo stack ke top element ko track karega)
3. **Size** (maximum capacity)

Operations:

- **push(x)** → element ko top par daalna
- **pop()** → top element ko nikalna
- **peek()** → current top element dekhna
- **isEmpty()** → stack khali hai ya nahi
- **isFull()** → stack full hai ya nahi

C Code (Single Stack in Array):

```
#include <stdio.h>

#define MAX 5

int stack[MAX];

int top = -1;

void push(int x) {

   if (top == MAX - 1) {

      printf("Stack Overflow\n");

   } else {

      stack[++top] = x;

      printf("%d pushed to stack\n", x);

   }

}

void pop() {

   if (top == -1) {

      printf("Stack Underflow\n");

   } else {

      printf("%d popped from stack\n", stack[top--]);
```

```
    }
}
void peek() {
    if (top == -1) {
        printf("Stack is empty\n");
    } else {
        printf("Top element is %d\n", stack[top]);
    }
}
int main() {
    push(10);
    push(20);
    push(30);
    peek();
    pop();
    peek();
    return 0;
}
```

2. Multiple Stacks in a Single Array

Agar ek hi array me **multiple stacks** banane ho, toh 2 tarike hote hain:

(a) Fixed Division Method

Array ko fixed parts me divide kar dete hain.
👉 Example: Agar array size 10 hai aur 2 stacks chahiye, toh first 5 elements stack1 ke liye, aur last 5 stack2 ke liye reserve karenge.

C Code (2 Stacks in Single Array - Fixed Division):

```
#include <stdio.h>
#define MAX 10
int arr[MAX];
int top1 = -1;
int top2 = MAX;
```

```
void push1(int x) {
  if (top1 < top2 - 1) {
    arr[++top1] = x;
    printf("%d pushed to Stack1\n", x);
  } else {
    printf("Stack Overflow\n");
  }
}
void push2(int x) {
  if (top1 < top2 - 1) {
    arr[--top2] = x;
    printf("%d pushed to Stack2\n", x);
  } else {
    printf("Stack Overflow\n");
  }
}
void pop1() {
  if (top1 >= 0) {
    printf("%d popped from Stack1\n", arr[top1--]);
  } else {
    printf("Stack1 Underflow\n");
  }
}
void pop2() {
  if (top2 < MAX) {
    printf("%d popped from Stack2\n", arr[top2++]);
  } else {
    printf("Stack2 Underflow\n");
  }
}

int main() {
  push1(10);
  push1(20);
  push2(100);
  push2(200);
  pop1();
  pop2();
  return 0;
}
```

Yahan **top1 left se grow karta hai** aur **top2 right se grow karta hai**.
Isse space ka zyada efficient use hota hai.

2nd Method: Flexible / Dynamic Sharing Method

Isme ek array ko **fixed parts me divide nahi karte**, balki **do stacks ko opposite directions me grow karte hain**.

Ex:

- Stack1 **left se right** grow karega (index 0 → MAX-1)
- Stack2 **right se left** grow karega (index MAX-1 → 0)

Isse dono stacks **ek dusre ki taraf grow karte hain**.
Aur jab tak top1 < top2 - 1, dono stacks me elements daal sakte ho.

C Code (Two Stacks in One Array – Dynamic Sharing):

```
#include <stdio.h>
#define MAX 10
int arr[MAX];
int top1 = -1;
int top2 = MAX;
void push1(int x) {
   if (top1 < top2 - 1) {
      arr[++top1] = x;
      printf("%d pushed to Stack1\n", x);
```

```
    } else {
        printf("Stack Overflow\n");
    }
}
void push2(int x) {
    if (top1 < top2 - 1) {
        arr[--top2] = x;
        printf("%d pushed to Stack2\n", x);
    } else {
        printf("Stack Overflow\n");
    }
}
void pop1() {
    if (top1 >= 0) {
        printf("%d popped from Stack1\n", arr[top1--]);
    } else {
        printf("Stack1 Underflow\n");
    }
}
void pop2() {
    if (top2 < MAX) {
        printf("%d popped from Stack2\n", arr[top2++]);
    } else {
        printf("Stack2 Underflow\n");
    }
}
int main() {
    push1(10);
    push1(20);
    push2(100);
    push2(200);

    pop1();

    pop2();

    return 0;

}
```

Difference Between the Two Methods:

1. **Fixed Division Method**
    - Array ko fixed parts me divide karna padta hai.
    - Agar ek stack full ho gaya aur dusra empty hai to bhi space waste hoga.
2. **Dynamic Sharing Method** (Top1 left se aur Top2 right se grow karte hain)
    - Space ka **efficient use** hota hai.
    - Jab tak top1 < top2 - 1, dono stacks me elements daale ja sakte hain.

<u>Infix, Prefix and Postfix Expressions:</u>

1. Infix Expression

- Yahi wo form hai jo hum normally math me likhte hain.
- **Operator** apne **operands** ke beech me aata hai.
- Example:

A + B

(A + B) * C

(A + B) * (C - D)

2. Prefix Expression (Polish Notation)

- Isko **Polish Notation** bhi kehte hain.
- Isme **operator** apne **operands** ke pehle likha jata hai.
- Example:

```
+ A B        // Infix: A + B

* + A B C    // Infix: (A + B) * C

* + A B - C D  // Infix: (A + B) * (C - D)
```

Note: Postfix me evaluation **stack** ka use karke easily hota hai.

Conversion Summary

Infix	**Prefix**	**Postfix**
A + B	+ A B	A B +
(A + B) * C	* + A B C	A B + C *
(A + B) * (C-D)	* + A B - C D	A B + C D - *

Evaluation (Example)

Postfix Evaluation:
Expression: 5 6 2 + *
Steps:

1. Push 5 → [5]
2. Push 6 → [5, 6]
3. Push 2 → [5, 6, 2]
4. Encounter + → pop(6,2) → 6+2=8 → push(8) → [5, 8]
5. Encounter * → pop(5,8) → 5*8=40 → push(40) → [40]
   Answer = 40

Utility and Conversion of these expressions from one to another Application of stack:

1. Utility of Infix, Prefix, Postfix

Infix (Normal Expression)

- Hum insaan naturally infix me sochte aur likhte hain.
- Example: (A + B) * C.
- **Problem**: Computer ke liye infix difficult hai kyunki usko **operator precedence** (priority) aur **associativity** samajhna padta hai.

Prefix (Polish Notation)

- Operators operands ke **pehle** hote hain.
- **Brackets ki zarurat nahi hoti**, kyunki order clear hota hai.
- **Efficient evaluation** recursion ya stack se ho sakta hai.
- Use: **Expression parsers, compilers**.

Postfix (Reverse Polish Notation)

- Operators operands ke **baad** hote hain.
- Evaluation **stack** ke through bahut easy hoti hai.
- Brackets ki zarurat nahi.
- **Calculator aur Compiler me sabse zyada use hota hai.**

---

2. Conversion Between Infix, Prefix, Postfix

(A) Infix → Postfix Conversion (using Stack)

Algorithm (Shunting Yard by Dijkstra):

1. Operands → directly output me daalo.
2. Operators → stack me push karo.
3. Agar precedence kam ho to stack se operators nikalke output me bhejo.
4. (→ stack me push karo.
5.) → stack se operators pop karo jab tak (na mile.
6. Expression ke end me stack ke saare operators nikal do.

Example: (A + B) * C

- Infix: (A + B) * C
- Postfix: A B + C *

(B) Infix → Prefix Conversion

Method 1: Reverse + Postfix Method

1. Infix ko **reverse** karo.
2. Parentheses ko ulta karo ((ko) aur) ko ().
3. Reverse infix ko **postfix me convert** karo.
4. Final postfix ko **reverse** karke Prefix banao.

Example: (A + B) * C

- Infix: (A + B) * C
- Reverse: C * (B + A)
- Postfix (reverse infix): C B A + *
- Reverse again → Prefix: * + A B C

(C) Postfix → Infix Conversion

1. Ek stack banao.
2. Operand aaye → stack me push karo.
3. Operator aaye → do operands pop karo, unhe operator ke sath infix banakar wapas push karo.
4. End me ek hi expression bachega = infix.

Example: A B + C *

- Stack: [A], [A,B]
- + → pop(B,A) → (A+B) → push → [(A+B)]
- C → push → [(A+B),C]
- * → pop(C,(A+B)) → ((A+B)*C)
- Result: (A + B) * C

(D) Prefix → Infix Conversion

1. Expression ko **right to left** scan karo.
2. Operand aaye → stack me push karo.
3. Operator aaye → 2 operands pop karo aur infix banao.
4. End me ek hi infix expression bachega.

Example: * + A B C

- Right to Left: C, B, A, +, *
- Stack: [C], [C,B], [C,B,A]
- + → pop(A,B) → (A+B) → push → [C,(A+B)]
- * → pop((A+B),C) → ((A+B)*C)
- Result: (A + B) * C

Summary Table

Conversion	**Method**
Infix → Postfix	Stack (Shunting Yard)

Conversion	**Method**
Infix → Prefix	Reverse + Postfix + Reverse
Postfix → Infix	Stack (Operands combine)
Prefix → Infix	Stack (Right to left)

Jo code maine aapko diya tha **Infix → Postfix, Prefix conversion aur Postfix evaluation** ke liye, usme kuch **limitations** hain. Chaliye detail me dekhte hain:

Limitations

1. Single Character Operands

- Code sirf **single letters/digits** (A, B, C ya 5, 6, 2) ko handle karta hai.
- Agar operand multi-digit number ho (e.g. 25+36) ya variable ka naam lamba ho (num1), to code galat result dega.

---

2. No Spaces Handling

- Input string me agar space ho ((A + B) * C), program crash ho sakta hai.
- Code sirf **continuous string** ko input manta hai ((A+B)*C).

---

3. Limited Operators

- Precedence function me sirf ye operators hain: +, -, *, /, %, ^.
- Agar koi aur operator aayega (e.g. //, &&, ||), to code handle nahi karega.

---

4. Evaluation Only for Digits

- Postfix evaluation part sirf **digits** (0–9) ke liye kaam karta hai.
- Agar operands letters (A, B, C) ho, evaluation possible nahi hai, kyunki unke values assign nahi kiye ja rahe.

---

5. Division Edge Cases

- Divide by zero (6 0 /) evaluate karne par crash hoga.
- Floating point division (5/2 = 2.5) bhi handle nahi karta, sirf integer division hota hai.

---

6. Stack Size Limitation

- MAX = 100 fix hai.
- Agar expression bahut bada ho, stack overflow ho sakta hai.

---

7. Prefix/Postfix → Infix Not Implemented

- Jo code diya hai usme conversion **Infix → Postfix/Prefix** aur **Postfix evaluation** hi hai.
- Lekin agar aapko **Postfix → Infix** ya **Prefix → Infix** chahiye, to alag functions banana padega.

Summary

- Simple expressions ke liye code sahi hai.
- **Real compiler/calculator jaisa code likhne ke liye**:
    - Multi-digit operands,
    - Floating point values,
    - Error handling (divide by zero, invalid expression),
    - Aur advanced operators handle karne padenge.

Data Structure me **Stack ko array ke through** represent karna ek basic aur important topic hai. Chaliye step by step samajhte hain:

Stack – Definition

Stack ek **Linear Data Structure** hai jo **LIFO (Last In First Out)** principle follow karta hai.

- Jo element **sabse last me push** hua, wahi **sabse pehle pop** hoga.
- Example: Plate rack – jo plate upar rakhi, wahi sabse pehle nikalti hai.

Array Representation of Stack

Stack ko ek **1D Array** ke through implement kar sakte hain.

Components:

1. **Array stack[MAX]** → elements store karne ke liye.
2. **top variable** → current top position ko track karega.
    - Initially top = -1.
3. **MAX** → stack ka maximum size.

Basic Operations

1. **Push(x)** → element stack me daalna.
   - Agar top == MAX-1 → Overflow (stack full).
   - Nahi to stack[++top] = x.
2. **Pop()** → top element nikalna.
   - Agar top == -1 → Underflow (stack empty).
   - Nahi to stack[top--] return karo.
3. **Peek() / Top()** → sirf top element dekhna (bina hataye).
4. **isEmpty()** → check if stack empty (top == -1).
5. **isFull()** → check if stack full (top == MAX-1).

C Code: Stack using Array

```
#include <stdio.h>

#define MAX 5   // Maximum size of stack

int stack[MAX];

int top = -1;

// Check if stack is full

int isFull() {

   return top == MAX - 1;

}

// Check if stack is empty

int isEmpty() {

   return top == -1;

}

// Push operation

void push(int x) {

   if (isFull()) {

      printf("Stack Overflow! Cannot push %d\n", x);

   } else {
        stack[++top] = x;
        printf("%d pushed to stack\n", x);
    }
}
// Pop operation
int pop() {
    if (isEmpty()) {
        printf("Stack Underflow! Cannot pop\n");
        return -1;
    } else {
        return stack[top--];
    }
}
// Peek operation
int peek() {
    if (isEmpty()) {
        printf("Stack is Empty!\n");
        return -1;
    } else {
        return stack[top];
    }
}
// Display stack
void display() {
    if (isEmpty()) {
        printf("Stack is Empty!\n");
    } else {
        printf("Stack elements: ");
        for (int i = top; i >= 0; i--) {
            printf("%d ", stack[i]);
        }
        printf("\n");
    }
}
int main() {
    push(10);
    push(20);
    push(30);
    display();
    printf("Top element: %d\n", peek());
    printf("%d popped from stack\n", pop());
    display();
    return 0;
}
```

Output Example

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
Stack elements: 30 20 10
Top element: 30
30 popped from stack
Stack elements: 20 10
```