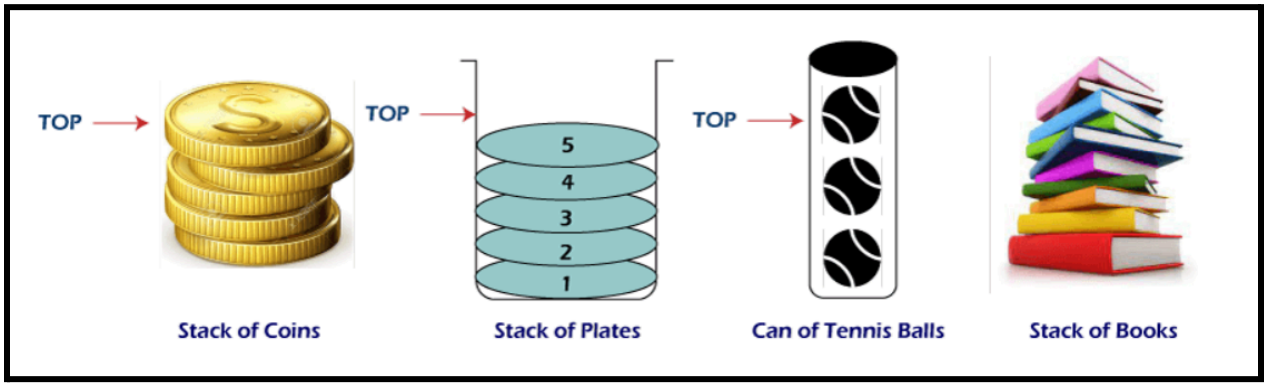
UNIT-III : STACKS

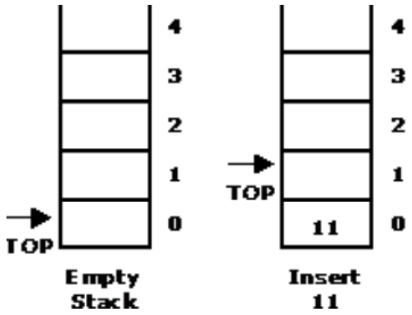
Introduction to stacks:

A Stack is a widely used linear data structure in modern computers in which insertions and deletions of an element can occur only at one end, i.e., top of the Stack. It is used in all those applications in which data must be stored and retrieved in the last.

An everyday analogy of a stack data structure is a stack of books on a desk, Stack of plates, table tennis, Stack of bootless, Undo or Redo mechanism in the Text Editors, etc.



Stack representation:



Basic Operations of Stack Data Structures:

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes the top element from the stack.
- **Peek**: Returns the top element without removing it.
- **IsEmpty**: Checks if the stack is empty.
- **IsFull**: Checks if the stack is full (in case of fixed-size arrays)

Implementation of stacks using arrays:

Algorithm :

```
begin
   if top = n it means the stack is full
   else
      top = top + 1
      stack (top)  = new_item;
end
```

Program :

```
#include <stdio.h>

// Function declarations
void push();
void pop();
void peek();

// Taking stack array of size 50

int stack[50],i,j,option=0,n,top=-1;
void main ()
{

//  Size of stack
   printf("Enter the number of elements in the stack ");
   scanf("%d",&n);
   printf("Stack implementation using array\n");
```

```
//    Taking user input for the operation to be performed
   while(option != 4)
   {
      printf("Choose one from the below options...\n");
      printf("\n1.Push\n2.Pop\n3.Peek\n4.Exit");
      printf("\n Enter your option \n");
      scanf("%d",&option);
      switch(option)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            peek();
            break;
         }
         case 4:
         {
            printf("Exiting");
            break;
         }
         default:
         {
            printf("Please Enter valid option");
         }
      };
```

```
// Push operation function
void push ()
{
   int val,value;
   if (top == n )
   printf("\n Overflow");
   else
   {
      printf("Enter the value?");
      scanf("%d",&val);
      top = top +1;
      stack[top] = value;
   }
}

// Pop operation function
void pop ()
{
   if(top == -1)
   printf("Underflow");
   else
   top = top -1;
}

// peek operation function
void peek()
{
   for (i=top;i>=0;i--)
   {
      printf("%d\n",stack[i]);
   }
   if(top == -1)
   {
      printf("Stack is empty");
   }
}
```

Implementation of stacks using linked list

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

// Structure to create a node with data and the next pointer
struct node {
   int info;
   struct node *ptr;
}*top,*top1,*temp;

int count = 0;
// Push() operation on a  stack
void push(int data) {
    if (top == NULL)
   {
      top =(struct node *)malloc(1*sizeof(struct node));
      top->ptr = NULL;
      top->info = data;
   }
   else
   {
      temp =(struct node *)malloc(1*sizeof(struct node));
      temp->ptr = top;
      temp->info = data;
      top = temp;
   }
   count++;
   printf("Node is Inserted\n\n");
}
```

```
int pop() {
     top1 = top;

    if (top1 == NULL)
    {
printf("\nStack Underflow\n");
        return -1;
    }

else
    top1 = top1->ptr;
    int popped = top->info;
    free(top);
    top = top1;
    count--;
    return popped;
}

void display() {
    // Display the elements of the stack
    top1 = top;
    if (top1 == NULL)
    {
        printf("\nStack Underflow\n");
        return;
    }

    printf("The stack is \n");
    while (top1 != NULL)
    {
        printf("%d--->", top1->info);
        top1 = top1->ptr;
    }
    printf("NULL\n\n");

}
```

```
int main()
 {
   int choice, value;
   printf("\nImplementation of Stack using Linked List\n");
   while (1) {
      printf("\n1. Push\n2. Pop\n3. Display\n4. Exit\n");
      printf("\nEnter your choice : ");
      scanf("%d", &choice);
      switch (choice)
 {
      case 1:
         printf("\nEnter the value to insert: ");
         scanf("%d", &value);
         push(value);
         break;
      case 2:
         printf("Popped element is :%d\n", pop());
         break;
      case 3:
         display();
         break;
      case 4:
         exit(0);
         break;
      default:
         printf("\nWrong Choice\n");
      }
   }
```

Applications of Stack:

1. Reverse a string :

Follow the steps given below to reverse a string using stack.

- Create an empty stack.
- One by one push all characters of string to stack.
- One by one pop all characters from the stack and put them back to string.

Iterate the input string from left to right and push the current character in the stack

A
T
A
D

Pop from the stack and append to string builder

A	T	D	A

2.checking balanced parentheses:

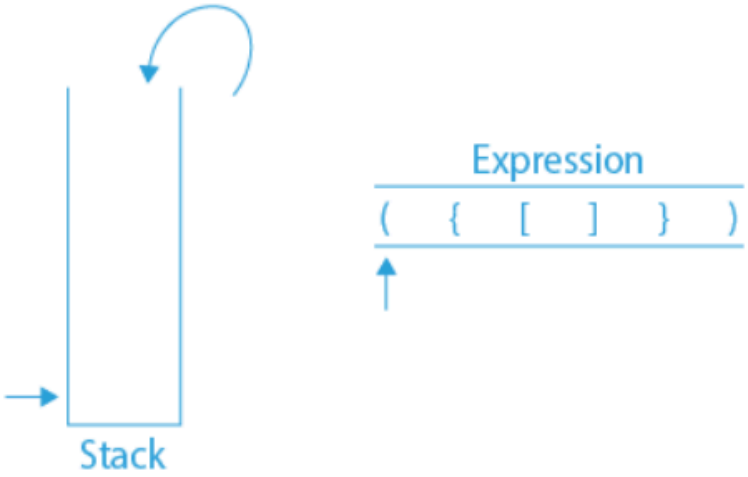
An expression is balanced if each opening bracket is closed by the same type of closing bracket in the exact same order. Stack is used to check for balanced parentheses because it allows us to keep track of opening and closing parentheses in an expression and ensure that they are properly matched and nested.

Follow the steps mentioned below to implement the idea:

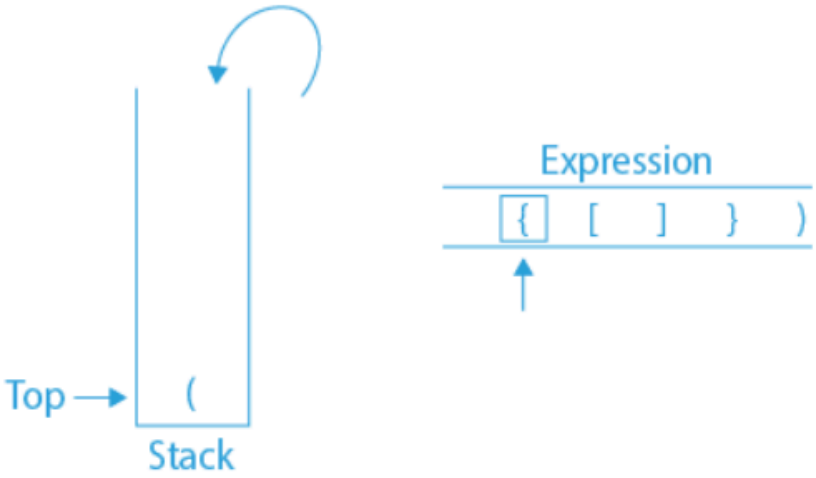
- Declare a character stack (say **temp**).
- Now traverse the string exp.
    - If the current character is a starting bracket (**'(' or '{' or '[')** then push it to stack.
    - If the current character is a closing bracket (**')' or '}' or ']'**) then pop from the stack and if the popped character is the matching starting bracket then fine.
    - Else brackets are **Not Balanced**.
- After complete traversal, if some starting brackets are left in the stack then the expression is **Not balanced**, else **Balanced**.

Algorithm to Check Balanced Parentheses in C using Stack

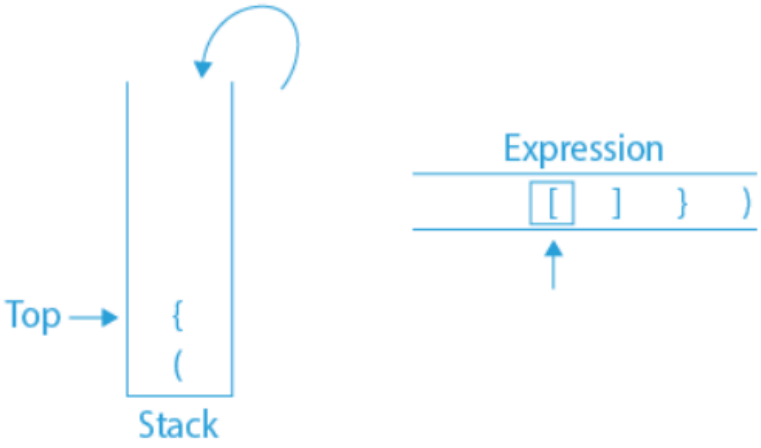
1. Initialize an empty stack.
2. Iterate i from 0 to length(expression).
    - Store the current character in a variable 'ch'.
    - If stack is empty: Push 'ch' to the stack
    - Else if current character is a closing bracket and of the top of the stack contains an opening bracket of the same type then remove the top of the stack: Else, push 'ch' to the stack
3. If the stack is empty, return true, else false



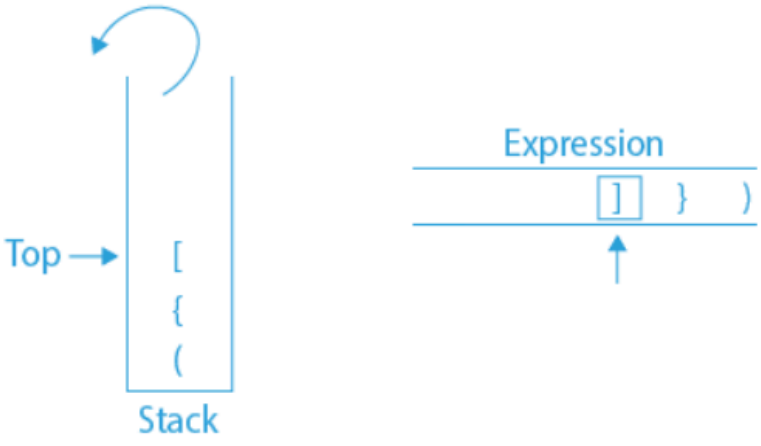
Stack is empty so push current character into the stack



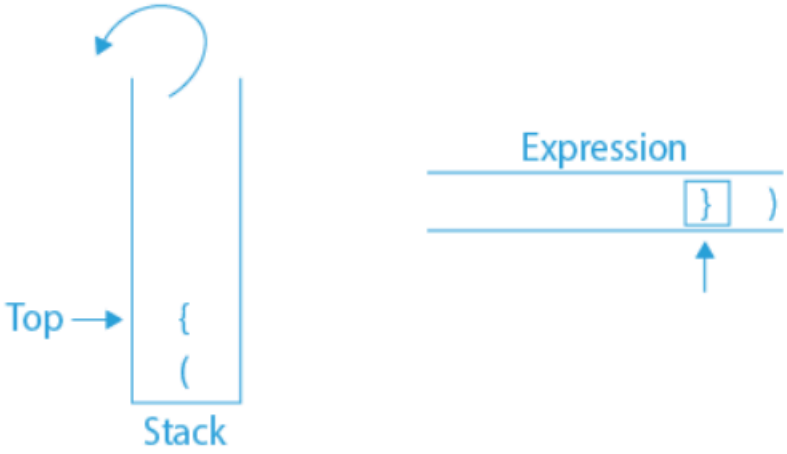
The current character is a opening bracket,
so push it into the stack



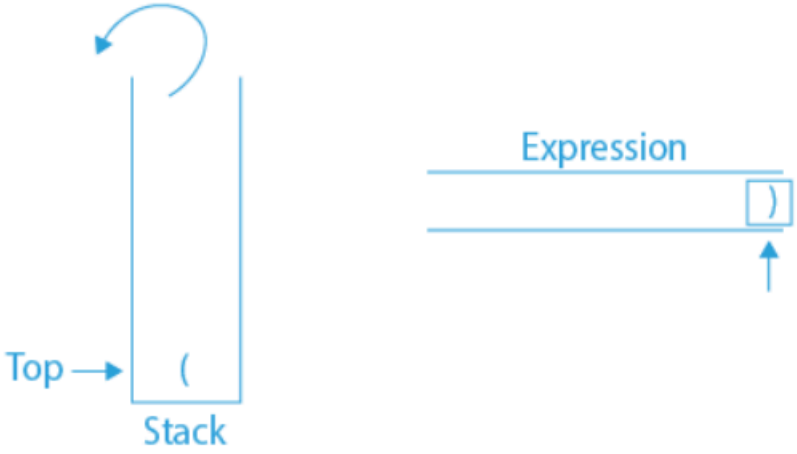
The current character is a opening bracket, so push it into the stack



The current character is a closing bracket and the top is the opening bracket of same type remove top of the stack.



The current character is a closing bracket and the top is the opening bracket of same type remove top of the stack.



The current character is a closing bracket and the top is the opening bracket of same type remove top of the stack.

3.Undo Operations: Stacks are used to implement undo operations in applications such as text editors or graphics software. Each action performed by the user is pushed onto the stack, allowing them to undo the actions by popping them off the stack.

4.Memory Management: Stacks are used in memory management systems to allocate and deallocate memory for variables and data structures. Memory blocks are allocated from the top of the stack, and when they are no longer needed, they are deallocated by moving the stack pointer.

5.Browser History: Stacks are used to implement the back and forward buttons in web browsers. Each page visited by the user is pushed onto the stack, allowing them to navigate back and forward through their browsing history.

6.Backtracking Algorithms: Stacks are used in backtracking algorithms, such as depth-first search (DFS), to keep track of the current state of the search and the path taken. This allows the algorithm to backtrack and explore other paths if necessary.

8.Function Call Management: Stacks are used to manage function calls and execution in programming languages. When a function is called, its arguments and local variables are pushed onto the stack, and when the function returns, they are popped off the stack.

7. Evaluating an Arithmetic Expression ?(converting infix to postfix.)

An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression.

These notations are –
• Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands. E.g.: (A + B) * (C -D)
• Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish
notation. E.g.: * +AB – CD

• Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post)its two operands. The postfix notation is called as suffix
notation and is also referred to as reverse polish notation.
E.g:AB + C D -

OPERATORS	**ASSOCIATIVITY**	**PRECEDENCE**
^ exponentiation	Right to left	Highest followed by *Multiplication and /division
*Multiplication, /division	Left to right	Highest followed by + addition and - subtraction
+ addition, - subtraction	Left to right	lowest

Rules of converting infix to postfix:

1.Read the expression from left to right .
2.The scanning character contains the operand then push into postfix column
3.The scanning character contains the operator then check whether stack is empty or not ,
3.1 if stack is empty then push the operator onto stack column
3.2 If stack is not empty then compare its operator precedence and associativity of the incoming operator with the operator present at top of the stack
3.2.1 if incoming operator > operator at top of the stack then push into stack column
3.2.2 if incoming operator < operator at top of the stack then pop out the operator.
3.2.3 if incoming operator is ")" pop out the operators present in stack till encountered the left parenthesis .
4. Check especially for a condition when the operator at the top of the stack and the scanned operator both are '^'. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.
5.While reaching the null character in the expression and still finding the operators in the stack then pop and add to the postfix expression .

Example 1:

Infix expression: K + L - M*N + (O^P) * W/U/V * T + Q

Input Expression	Stack	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	K L + M N* K L + M N* -
(	+ (	K L + M N *-
O	+ (	K L + M N * - O
^	+ (^	K L + M N* - O
P	+ (^	K L + M N* - O P
)	+	K L + M N* - O P ^

*	+ *	K L + M N* - O P ^
W	+ *	K L + M N* - O P ^ W
/	+ /	K L + M N* - O P ^ W *
U	+ /	K L + M N* - O P ^W*U
/	+ /	K L + M N* - O P ^W*U/
V	+ /	KL + MN*-OP^W*U/V
*	+ *	KL+MN*-OP^W*U/V/
T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*-OP^W*U/V/T*+Q+

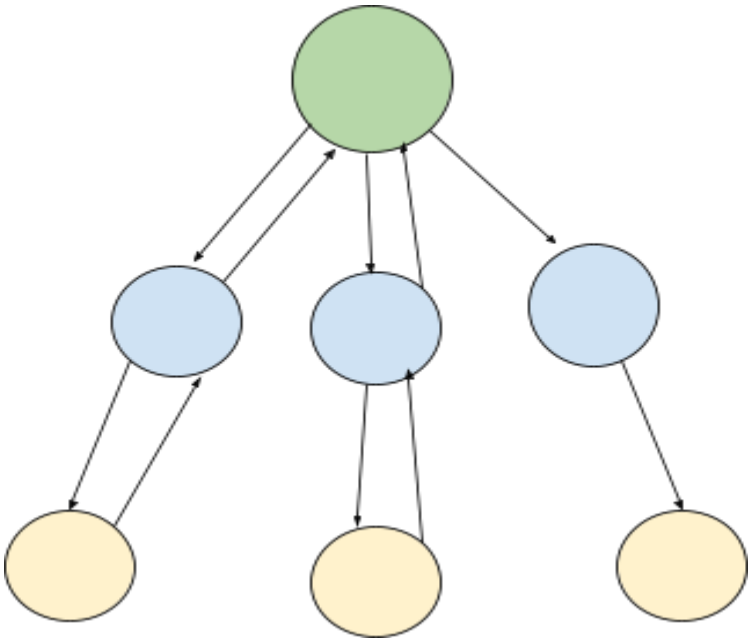
Backtracking:

Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku. When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.

Generally backtracking can be seen having below mentioned time complexities:

- Exponential (O(K^N))
- Factorial (O(N!))



Pseudo code

```
void FIND_SOLUTIONS( parameters):
    if (valid solution):
        store the solution
        Return
    for (all choice):
        if (valid choice):
            APPLY (choice)
            FIND_SOLUTIONS (parameters)
            BACKTRACK (remove choice)
    Return
```

Reversing the list

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

// Define a structure for a node in the linked list
typedef struct Node {
    int data;
    struct Node* next;
} Node;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning of the list
Node* insertAtBeginning(Node* head, int data) {
    Node* newNode = createNode(data);
    newNode->next = head;
    return newNode;
}

// Function to reverse a linked list
Node* reverseList(Node* head) {
    Node *prev = NULL, *current = head, *next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
```

```
    }
    return prev; // new head of the reversed list
}

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

int main() {
    Node* head = NULL;

    // Insert some nodes at the beginning of the list
    head = insertAtBeginning(head, 3);
    head = insertAtBeginning(head, 5);
    head = insertAtBeginning(head, 7);
    head = insertAtBeginning(head, 9);

    printf("Original list: ");
    printList(head);

    // Reverse the list
    head = reverseList(head);

    printf("Reversed list: ");
    printList(head);

    return 0;
}
```