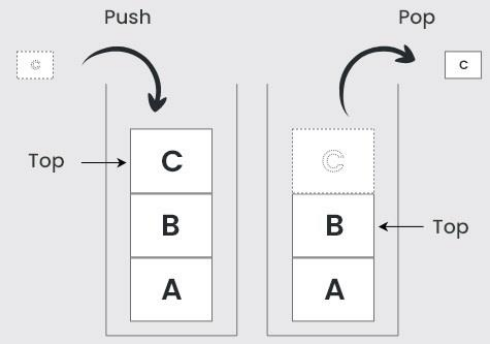
# Stacks

A Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).




**Key Operations on 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).

**Applications of Stack Data Structures**

- Recursion
- Conversion of infix expression to postfix/prefix
- Postfix Expression Evaluation and Parsing
- Undo/Redo Operations
- Depth-First Search (DFS) in graphs
- Browser History
- Function Calls

**What is infix expression?**

- The traditional method of writing mathematical expressions is called infix expressions.
- It is of the form **<operand><operator><operand>**.
- As the name suggests, here the operator is fixed inside between the operands. e.g. A+B here the plus operator is placed inside between the two operators, **(A*B)/Q.**
- Such expressions are easy to understand and evaluate for human beings. However, the computer finds it difficult to parse - Information is needed about operator precedence and associativity rules, and brackets that override these rules.
- Hence we have postfix and prefix notations which make the computer take less effort to solve the problem.

**What is postfix expression?**

- The postfix expression as the name suggests has the operator placed right after the two operands.

- It is of the form **<operand><operand><operator>**
- In the infix expressions, it is difficult to keep track of the operator precedence whereas here the postfix expression itself determines the precedence of operators (which is done by the placement of operators)i.e. the operator which occurs first operates on the operand.
- E.g. **PQ-C/**, here – operation is done on P and Q and then / is applied on C and the previous result.
- A postfix expression is a parenthesis-free expression. For evaluation, we evaluate it from left to right.

| Infix expression | Postfix expression |
|---|---|
| ( A + B ) * ( C - D ) | AB + CD - * |
| ( A + B ) / ( C - D ) - ( E * F ) | AB + CD - / EF * - |

## What is prefix expression?

An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

| Infix Expression | Prefix Expression | Postfix Expression |
|---|---|---|
| A + B | + A B | A B + |
| A + (B * C) | + A * B C | A B C * + |
| (A + B) * C | * + ABC | A B + C * |

**Approach: To convert Infix expression to Postfix**

1. Scan the infix expression from **left to right**.
2. If the scanned character is an operand, Print it.
3. Else,
   - If the precedence of the scanned operator is greater than the precedence of the operator in the stack or the stack is **empty** or the stack contains a **'('**, push the character into the stack.
   - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack.
4. If the scanned character is an **'('**, push it into the stack.
5. If the scanned character is an **')'**, pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-5 until the entire infix expression is scanned.
7. Print the output.
8. Pop and print the output from the stack until it is not empty.

**Approach: To convert Infix expression to Postfix**

Note: Same as postfix procedure but after popping the operator, its placed before the half-done prefix expression

1. Scan the infix expression from **left to right**.
2. If the scanned character is an operand, Print it.
3. Else,
    - If the precedence of the scanned operator is greater than the precedence of the operator in the stack or the stack is **empty** or the stack contains a **'('**, push the character into the stack.
    - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack.
4. If the scanned character is an **'('**, push it into the stack.
5. If the scanned character is an **')'**, pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-5 until the entire infix expression is scanned.
7. Print the output.
8. Pop and print the output from the stack until it is not empty.

Example 1: **A+B*C-D**

| Sr No | Expression | Stack | Prefix |
|---|---|---|---|
| 0 | A | | A |
| 1 | + | + | A |
| 2 | B | + | A B |
| 3 | * | + * | A B |
| 4 | C | + * | A B C |
| 5 | - | - | +A*BC |
| 6 | D | - | +A*BC D |
| 7 | | | -+A*BCD |

Example 2: **A+B*C/D**

| Sr No | Expression | Stack | Prefix |
|---|---|---|---|
| 0 | A | | A |
| 1 | + | + | A |
| 2 | B | + | A B |
| 3 | * | + * | A B |
| 4 | C | + * | A B C |
| 5 | / | + / | A *BC |
| 6 | D | + / | A *BC D |
| 7 | | | +A/*BCD |

Example 3: **(A+B)*C/D-E**

| Sr No | Expression | Stack | Prefix |
|---|---|---|---|
| 0 | ( | ( | |

| | | | |
|---|---|---|---|
| 1 | A | ( | A |
| 2 | + | ( + | A |
| 3 | B | ( + | A B |
| 4 | ) | | +AB |
| 5 | * | * | +AB |
| 6 | C | * | +AB C |
| 7 | / | / | *+ABC |
| 8 | D | / | *+ABC D |
| 9 | - | - | /*+ABCD |
| 10 | E | - | /*+ABCD E |
| 11 | | | -/*+ABCDE |

Example 4: **(a+b^c)* d/e-f**

| Sr No | Expression | Stack | Prefix |
|---|---|---|---|
| 0 | ( | ( | |
| 1 | a | ( | a |
| 2 | + | ( + | a |
| 3 | b | ( + | a b |
| 4 | ^ | ( + ^ | a b |
| 5 | c | ( + ^ | a b c |
| 6 | ) | | +a^bc |
| 7 | * | * | +a^bc |
| 8 | d | * | +a^bc d |
| 9 | / | / | *+a^bcd |
| 10 | e | / | *+a^bcd e |
| 11 | - | - | /*+a^bcde |
| 12 | f | - | /*+a^bcde f |
| 13 | | | -/*+a^bcdef |

## Evaluation of Postfix Expression

To evaluate a postfix expression, we can use a stack.

*Iterate the expression from left to right and keep on storing the operands into a stack. Once an operator is received, pop the two topmost elements and evaluate them and push the result in the stack again.*

## Recursion

The recursive function should call itself.

The recursive function should have an ending point where it does not make further recursive calls. Also called as the base case.

**Example**

```
int factorial(int n)
{
    if(n <= 1) //base case
    return 1;
 else
   return (n * factorial(n - 1));
}
```

When calculating factorial(5). Here, base case is factorial(0) = 1 || factorial(1) = 1.
Recursive functions should go like factorial(5), factorial(4),factorial(3),factorial(2), finally ends with factorial(1) or factorial(0).

## Tower of Hanoi

Tower of Hanoi consists of three pegs or towers with n disks placed one over the other. The objective of the puzzle is to move the stack to another peg following these simple rules. Only one disk can be moved at a time. No disk can be placed on top of the smaller disk.

The 3 pegs are called Source (S), Temporary (T) and Destination (D).

```
#include < stdio.h >
int tower(int n, char s, char t, char d)
{
  int x, y;
  if(n <= 0) return 0;
  x = tower(n - 1, s, d, t);
  printf("\nMove %d from %c to %c", n, s, d);
  y = tower(n - 1, t, s, d);
  return(x + y + 1);
}
int main(void)
{
  int n, k;
  printf("\nEnter the number of discs : ");
  scanf("%d", &n);
  k = tower(n, 'S', 'T', 'D');
  printf("\n\nThere are %d moves", k);
  return 0;
}
```

**Binary Search (Recursive)**

Binary Search is a search algorithm that is used to find the position of an element (target value ) in a sorted array. The array should be sorted prior to applying a binary search.
Binary search is also known by these names, logarithmic search, binary chop, half interval search.

**Working of Binary Search**

The binary search algorithm works by comparing the element to be searched by the middle element of the array and based on this comparison follows the required procedure.
Case 1 – element = middle, the element is found return the index.

Case 2 − element > middle, search for the element in the sub-array starting from middle+1 index to n.
Case 3 − element < middle, search for element in the sub-array starting from 0 index to middle -1.

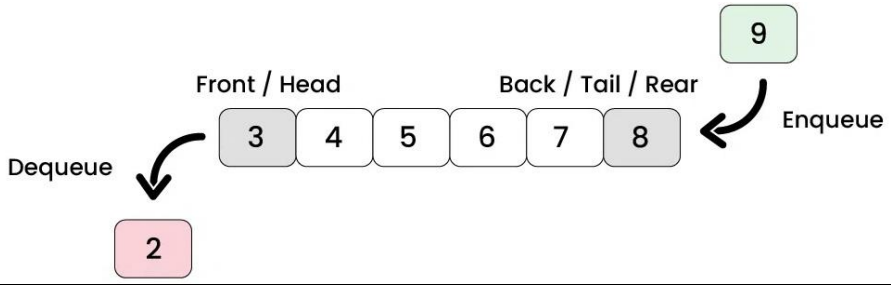
**Algorithm**

Parameters inital_value , end_value
Step 1 : Find the middle element of array. using ,
middle = initial_value + end_value / 2 ;
Step 2 : If middle = element, return 'element found' and index.
Step 3 : if middle > element, call the function with end_value = middle - 1 .
Step 4 : if middle < element, call the function with start_value = middle + 1 .
Step 5 : exit.

```
//Recursive Binary search program
#include <stdio.h>
int recursiveBinSearch(int arr[], int start, int end, int element) {
  if (end >= start){
    int mid = start + (end - start )/2;
    if (arr[mid] == element)
      return mid;
    if (arr[mid] > element)
      return recursiveBinSearch(arr, start, mid - 1, element);
    return recursiveBinSearch(arr, mid + 1, end, element);
  }
  return -1;
}
int main(void) {
  int arr[] = {1, 4, 7, 9, 26, 55, 87,90};
  int n = 8;
  int element = 70;
  int index = recursiveBinSearch(arr, 0, n - 1, element);
  if(index == -1 )
  printf("Element not found in the array ");
  else
    printf("Element found at index : %d", index);
  return 0;
}
```

## Queue

A queue is a linear data structure that follows the First In First Out (FIFO) order of insertion and deletion. It means that the element that is inserted first will be the first one to be removed and the element that is inserted last.



**Implementation of a Queue in C**

We can implement a queue in C using either an array or a linked list. In this article, we will use the array data structure to store the elements. The insertion in the queue is done at the back of the queue and the deletion is done at the front. So we maintain two index pointers front and rear pointers to keep track of the front and back of the queue. The queue consists of two basic operations enqueue which adds elements to the queue (insertion) from the rear pointer and dequeue(deletion) which removes elements from the queue through the front pointer.

```
struct Queue {
    type arr[MAX_SIZE];
    int front;
    int rear;
}
```

The front pointer initial value will be -1 and the rear pointer initial value will be 0.

| Operation | Description |
|---|---|
| *Enqueue* | Inserts an element in the queue through rear pointer. |
| *Dequeue* | Removes an element from the queue through front pointer. |
| *Peek* | Returns the front element of the queue. |
| *IsFull* | Returns true is the queue is full otherwise returns false. |
| *IsEmpty* | Returns true is the queue is empty otherwise returns false. |

**Algorithm of isFull Function**

1. If rear == MAX_SIZE, return true.
2. Else, return false.

**isEmpty Function**

The isEmpty function will check whether the queue is empty or not. When we initialize a queue, we set the front = -1 and rear = 0. So we know that when there are no elements in the queue, the front = rear – 1.

**Algorithm of isEmpty Function**

1. If front == rear – 1, return true.
2. Else, return false

**Algorithm of Enqueue Function**

1. Check whether the queue is full.
2. If the queue is full, display the overflow message.
3. If the queue is not full, add the new element to the position pointed to by the rear pointer.
4. Increment the rear pointer.

**Algorithm of Dequeue**

1. Check whether the queue is empty.
2. If the queue is empty, display the underflow message.
3. If the queue is not empty,
4. Increment the front pointer of the queue.
5. remove the element at the front position.

**Algorithm of Peek Function**

1. Check if the queue is empty.
2. If the queue is empty, return -1.
3. If not, return queueArray[front + 1].

## Lab programs from Unit-1

```
//Demonstration of Stack operations
//Here stack is implemented using Array and top is passed as pointer

#include<stdio.h>
#define size 5
void push(int s[],int *top)
{
        if(*top == size-1)
                return;
        printf("Enter the element:");
        scanf("%d",&s[++(*top)]);
}
int pop(int *top)
{
        if(*top == -1)
                return -1;
        return((*top)--);
}
void display(int s[],int *top)
{
        int i;
        if(*top==-1)
        {       printf("THE STACK IS EMPTY");
          return ;
        }
        else
        {
                for(i=*top;i>=0;i--)
                        printf("%d ",s[i]);
                if(*top == size-1)
                        printf(" : THE STACK IS FULL");
        }
}
void main()
{
        int x,ch,s[size],top=-1;
        while(1)
        {
        //      display(s,&top);
                printf("\n1:PUSH  2:POP  3.DISPLAY  OTHER:EXIT\n");
                printf("Enter Your Choice: ");
                scanf("%d",&ch);
                switch(ch)
                {
```

```
                        case 1 : push(s,&top);break;
                        case 2 : x=pop(&top);
                                 if(x!=-1)
                                     printf("Popped Element = %d",x);
break;
                        case 3: display(s,&top);
                        default: return;
                }
        }
}
```

**//Evaluate postfix expression**

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

void error()
{
        printf("\nGiven postfix is incorrect");
        exit(0);
}
void push(float stack[],int *top,float num)
{
        stack[++(*top)]=num;
}
float pop(float stack[],int *top)
{
        if(*top == -1)error();
        return(stack[(*top)--]);
}
float eval(char postfix[20])
{
        float stack[20],op1,op2;
        int i=-1,top=-1;
        while(postfix[++i]!='\0')
        {
                if(isdigit(postfix[i]))
                        push(stack,&top,postfix[i]-48);
                else
                {
                        op2=pop(stack,&top);
                        op1=pop(stack,&top);
                        switch(postfix[i])
                        {
                                case '*':push(stack,&top,op1 * op2);break;
                                case '/':push(stack,&top,op1 / op2);break;
                                case '+':push(stack,&top,op1 + op2);break;
                                case '-':push(stack,&top,op1 - op2);break;
                        }
                }
```

```
        }
        if(top)error();
        return(pop(stack,&top));
}
int main()
{
        char postfix[20];
                printf("\nEnter the postfix expression: ");
        scanf("%s",postfix);
        printf("\nValue of the postfix expression is %.2f",eval(postfix));
        return 0;
}
```

```
#include <stdio.h>
# define SIZE 5
void enqueue();
void dequeue();
void show();
int inp_arr[SIZE];
int Rear = - 1;
int Front = - 1;
main()
{
   int ch;
   while (1)
   {
      printf("\n1.Enqueue Operation\n");
      printf("2.Dequeue Operation\n");
      printf("3.Display the Queue\n");
      printf("4.Exit\n");
      printf("Enter your choice of operations : ");
      scanf("%d", &ch);
      switch (ch)
      {
         case 1:
         enqueue();
         break;
         case 2:
         dequeue();
         break;
         case 3:
         show();
         break;
         case 4:
         exit(0);
         default:
         printf("Incorrect choice \n");
      }
   }
}
```

```
void enqueue()
{
   int insert_item;
   if (Rear == SIZE - 1)
      printf("Overflow \n");
   else
   {
      if (Front == - 1)

      Front = 0;
      printf("Element to be inserted in the Queue\n : ");
      scanf("%d", &insert_item);
      Rear = Rear + 1;
      inp_arr[Rear] = insert_item;
   }
}

void dequeue()
{
   if (Front == - 1 || Front > Rear)
   {
      printf("Underflow \n");
      return ;
   }
   else
   {
      printf("Element deleted from the Queue: %d\n", inp_arr[Front]);
      Front = Front + 1;
   }
}

void show()
{

   if (Front == - 1)
      printf("Empty Queue \n");
   else
   {
      printf("Queue: \n");
      for (int i = Front; i <= Rear; i++)
         printf("%d ", inp_arr[i]);
      printf("\n");
   }
}
```

## References


https://www.calcont.in/Conversion/infix_to_prefix
https://www.geeksforgeeks.org/queue-in-c/