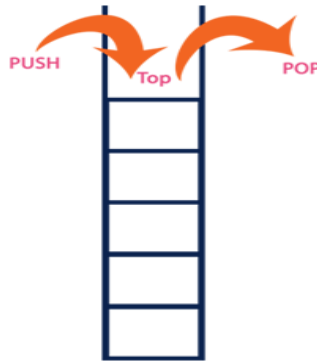


Chapter 3

What is Stack?

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at a single position which is known as "**top**". That means, a new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.



In the figure, PUSH and POP operations are performed at a top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

- It is type of **linear data structure**.
- It follows **LIFO** (Last In First Out) property.
- It has only one pointer **TOP** that points the last or top most element of Stack.
- Insertion and Deletion in stack can only be done from top only.
- Initially, the top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.
- Insertion in stack is also known as a **PUSH operation**.
- Deletion from stack is also known as **POP operation** in stack.

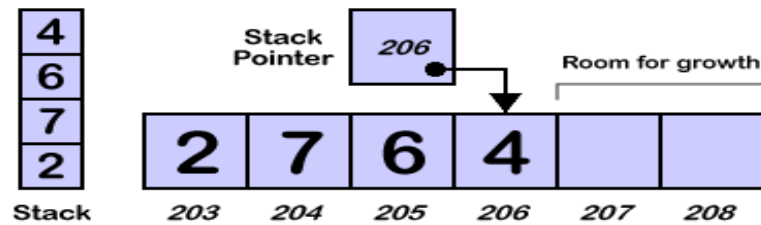
Memory Representation of Stacks

There are two ways to represent stacks in memory:

1. Array representation (Static data structure)
2. Linked list representation (Dynamic data structure)

Array representation of Stacks

Stacks may be represented in the computer in various ways, usually by means of a one-way list or a linear array. Unless otherwise stated or implied, each of our stacks will be maintained by a linear array **STACK**; a pointer variable **TOP**, which contains the location of the top element of the stack; and a variable **MAXSTK** which gives the maximum number of elements that can be held by the stack. The condition **TOP = 0** or **TOP = NULL** will indicate that the stack is empty.

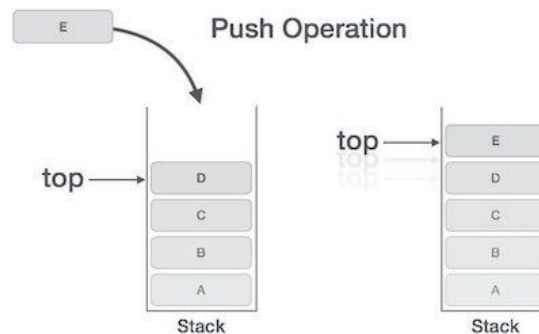


Memory Representation of Stacks

1. Insertion in Stack/Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, display “stack is FULL” and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.



Example

```
void push(int data) {
    if(!isFull())
    {
        top = top + 1;
        stack[top] = data;
    }
    else
    {
        printf("Could not insert data, Stack is full.\n");
    }
}
```

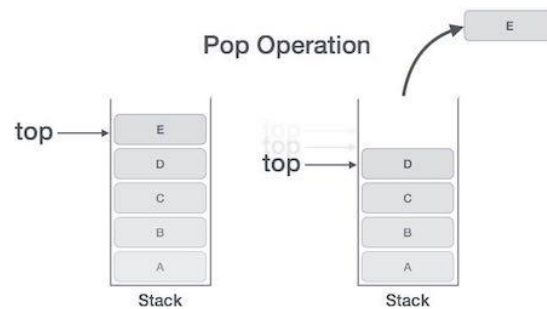
2. Deletion/Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value.

A Pop operation may involve the following steps –

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.

- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



Example

```
int pop(int data) {
    if(!isempty()) {
        data = stack[top];
        top = top - 1;
        return data;
    }
    else
    {
        printf("Could not retrieve data, Stack is empty.\n");
    }
}
```

display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

- **Step 1** - Check whether **stack** is **EMPTY**. (**top == -1**)
- **Step 2** - If it is **EMPTY**, then display "**Stack is EMPTY!!!!**" and terminate the function.
- **Step 3** - If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).
- **Step 3** - Repeat above step until **i** value becomes '0'.

Implementation of Stack using Array

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10
void push(int);
void pop();
void display();

int stack[SIZE], top = -1;
void main()
{
    int value, choice;
    clrscr();
    while(1){
```

```

printf("\n\n***** MENU *****\n");
printf("1. Push\n2. Pop\n3. Display\n4. Exit");
printf("\nEnter your choice: ");
scanf("%d",&choice);
switch(choice){
    case 1: printf("Enter the value to be insert: ");
            scanf("%d",&value);
            push(value);
            break;
    case 2: pop();
            break;
    case 3: display();
            break;
    case 4: exit(0);
    default: printf("\nWrong selection!!! Try again!!!");
}
}
}
void push(int value){
    if(top == SIZE-1)
        printf("\nStack is Full!!! Insertion is not possible!!!");
    else{
        top++;
        stack[top] = value;
        printf("\nInsertion success!!!");
    }
}
void pop(){
    if(top == -1)
        printf("\nStack is Empty!!! Deletion is not possible!!!");
    else{
        printf("\nDeleted : %d", stack[top]);
        top--;
    }
}
void display(){
    if(top == -1)
        printf("\nStack is Empty!!!");
    else{
        int i;
        printf("\nStack elements are:\n");
        for(i=top; i>=0; i--)
            printf("%d\n",stack[i]);
    }
}
}

```

Applications of Stack

Expression Evaluation

Stack is used to evaluate prefix, postfix and infix expressions.

Expression Conversion

An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

- i. Infix to Postfix
- ii. Infix to Prefix
- iii. Postfix to Infix
- iv. Prefix to Infix

Syntax Parsing

Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

Backtracking

Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

Parenthesis Checking

Stack is used to check the proper opening and closing of parenthesis.

String Reversal

Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

Function Call

Stack is used to keep information about the active functions or subroutines.

Reverse String using STACK in C –

This program will **read a string and reverse the string using Stack push and pop operations** in C programming Language.

Reversing string is an operation of Stack by using Stack we can reverse any string, here we implemented a program in C - this will reverse given string using Stack.

The logic behind to implement this program:

1. Read a string.
2. **Push all characters until NULL is not found** - Characters will be stored in stack variable.
3. **Pop all characters until NULL is not found** - As we know stack is a **LIFO** technique, so last character will be pushed first and finally we will get reversed string in a variable in which we store inputted string.

`/*C program to Reverse String using STACK*/`

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

```
#include <string.h>
```

```
#define MAX 100    /*maximum no. of characters*/
```

```
/*stack variables*/
```

```
int top=-1;
```

```

int item;
/*****/
/*string declaration*/
char stack_string[MAX];
/*function to push character (item)*/
void pushChar(char item);
/*function to pop character (item)*/
char popChar(void);

/*function to check stack is empty or not*/
int isEmpty(void);
/*function to check stack is full or not*/
int isFull(void);

int main()
{
    char str[MAX];
    int i;
    printf("Input a string: ");
    scanf("%^[^\n]s",str); /*read string with spaces*/
    /*gets(str);-can be used to read string with spaces*/
    for(i=0;i<strlen(str);i++)
        pushChar(str[i]);
    for(i=0;i<strlen(str);i++)
        str[i]=popChar();
    printf("Reversed String is: %s\n",str);
    return 0;
}

/*function definition of pushChar*/
void pushChar(char item)
{
    /*check for full*/
    if(isFull())
    {
        printf("\nStack is FULL !!!\n");
        return;
    }
    /*increase top and push item in stack*/
    top=top+1;
    stack_string[top]=item;
}

/*function definition of popChar*/
char popChar()
{
    /*check for empty*/
    if(isEmpty())

```

```

{
    printf("\nStack is EMPTY!!!\n");
    return 0;
}
/*pop item and decrease top*/
item = stack_string[top];
top=top-1;
return item;
}
/*function definition of isEmpty*/
int isEmpty()
{
    if(top==-1)
        return 1;
    else
        return 0;
}
/*function definition of isFull*/
int isFull()
{
    if(top==MAX-1)
        return 1;
    else
        return 0;
}

```

Polish Notation (PN)

Polish notation is a notation form for expressing arithmetic, logic and algebraic equations. Its most basic distinguishing feature is that operators are placed on the left of their operands. If the operator has a defined fixed number of operands, the syntax does not require brackets or parenthesis to lessen ambiguity.

The name comes from the Polish mathematician/logician Lukasiewicz, who introduced it. There are 3 different ways to write an algebraic expression:

- Infix form
- Prefix form
- Postfix form

Infix form: The binary operator is between the two operands.

infix-expression := (infix-expression operand infix-expression)

Examples

$(3 * 7)$

$((1 + 3) * 2)$

$((1 + 3) * (2 - 3))$

Prefix form: the operator precedes the two operands.

prefix-expression := (operand prefix-expression prefix-expression)

Examples

(\ast 3 7) or simply \ast 3 7

(\ast (+ 1 3) 2) or simply \ast + 1 3 2

(\ast (+ 1 3) (- 2 3)) or simply \ast + 1 3 - 2 3

Postfix form: the operator is after the two operands.

postfix-expression := (operand postfix-expression postfix-expression)

Examples

(3 7 \ast) or simply 3 7 \ast

((1 3 +) 2 \ast) or simply 1 3 + 2 \ast

((1 3 +) (2 3 -) \ast) or simply 1 3 + 2 3 - \ast

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in the expression $a + b - c$, both + and - have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and - are left associative, so the expression will be evaluated as $(a + b) - c$.

Precedence and associativity determine the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) –

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (\ast) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

The above table shows the default behaviour of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis.

For example –

In $a + b \ast c$, the expression part $b \ast c$ will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for $a + b$ to be evaluated first, like $(a + b) \ast c$.

Examples of Infix, Prefix, Postfix

- Infix $A + B$, $3 \ast x - y$
- Prefix $+AB$, $- \ast 3xy$
- Postfix $AB+$, $3x \ast y-$

Converting to Infix to Postfix

$A + B \ast C$

$\Rightarrow A + [B \ast C]$

$\Rightarrow A + [BC \ast]$

$\Rightarrow A [BC \ast] +$

$\Rightarrow ABC \ast +$

Converting to Infix to Prefix

$A + B * C$

$\Rightarrow A + [B * C]$

$\Rightarrow A + [*BC]$

$\Rightarrow + A [*BC]$

$\Rightarrow + A * BC$

Why?

Why use PREFIX and POSTFIX notations when we have simple INFIX notation?

INFIX notations are not as simple as they seem especially while evaluating them. To evaluate an infix expression we need to consider Operators' Priority and Associative property

• E.g. expression $3+5*4$ evaluate to 32 i.e. $(3+5)*4$ or to 23 i.e. $3+(5*4)$.

To solve this problem Precedence or Priority of the operators was defined. Operator precedence governs the evaluation order. An operator with higher precedence is applied before an operator with lower precedence.

The following table briefly tries to show the difference in all three notations –

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	$a + b$	$+ a b$	$a b +$
2	$(a + b) * c$	$* + a b c$	$a b + c *$
3	$a * (b + c)$	$* a + b c$	$a b c + *$
4	$a / b + c / d$	$+ / a b / c d$	$a b / c d / +$
5	$(a + b) * (c + d)$	$* + a b + c d$	$a b + c d + *$
6	$((a + b) * c) - d$	$- * + a b c d$	$a b + c * d -$

1. Conversion of Infix to Postfix

Algorithm for Infix to Postfix

Step 1: Consider the next element in the input.

Step 2: If it is operand, display it.

Step 3: If it is opening parenthesis, insert it on stack.

Step 4: If it is an operator, then

- If stack is empty, insert operator on stack.
- If the top of stack is opening parenthesis, insert the operator on stack
- If it has higher priority than the top of stack, insert the operator on stack.
- Else, delete the operator from the stack and display it, repeat Step 4.

Step 5: If it is a closing parenthesis, delete the operator from stack and display them until an opening parenthesis is encountered. Delete and discard the opening parenthesis.

Step 6: If there is more input, go to Step 1.

Step 7: If there is no more input, delete the remaining operators to output.

A summary of the rules follows:

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Program for Infix to Postfix Conversion

```
#include <stdio.h>
#include <ctype.h>
#define SIZE 50
char s[SIZE];
int top=-1;
push(char elem)
{
    s[++top]=elem;
    return 0;
}
char pop()
{
    return(s[top--]);
}
int pr(char elem)
{
    switch(elem)
    {
        case '#': return 0;
        case '(': return 1;
        case '+':
        case '-': return 2;
        case '*':
        case '/': return 3;
    }
    return 0;
}
void main()
```

```

{
    char infx[50], pofx[50], ch, elem;
    int i=0, k=0;
    printf("\n\nEnter Infix Expression: ");
    scanf("%s", infx);
    push('#');
    while( (ch=infx[i++]) != '\0')
    {
        if( ch == '(') push(ch);
        else
            if(isalnum(ch)) pofx[k++]=ch;
        else
            if( ch == ')')
            {
                while( s[top] != '(')
                    pofx[k++]=pop();
                elem=pop();
            }
            else
            {
                while( pr(s[top]) >= pr(ch) )
                    pofx[k++]=pop();
                push(ch);
            }
    }
    while( s[top] != '#')
        pofx[k++]=pop();
    pofx[k]='\0';
    printf("\n\n Given Infix Expression: %s \n Postfix Expression: %s\n", infx, pofx);
}

```

Advantage of Postfix Expression over Infix Expression

1. An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence). Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

Example: Convert $A * (B + C) * D$ to postfix notation.

Move	Current Token	Stack	Output
1	A	empty	A
2	*	*	A
3	((*	A
4	B	(*	A B
5	+	+(*	A B
6	C	+(*	A B C

7)	*	A B C +
8	*	*	A B C + *
9	D	*	A B C + * D
10		empty	

EXAMPLE: $A + (B * C - (D / E - F) * G) * H$

Stack	Input	Output
Empty	$A + (B * C - (D / E - F) * G) * H$	-
Empty	$+ (B * C - (D / E - F) * G) * H$	A
+	$(B * C - (D / E - F) * G) * H$	A
+($B * C - (D / E - F) * G) * H$	A
+($* C - (D / E - F) * G) * H$	AB
+(*	$C - (D / E - F) * G) * H$	AB
+(*	$- (D / E - F) * G) * H$	ABC
+(-	$(D / E - F) * G) * H$	ABC*
+(- ($D / E - F) * G) * H$	ABC*
+(- ($/ E - F) * G) * H$	ABC*D
+(- (/	$E - F) * G) * H$	ABC*D
+(- (/	$- F) * G) * H$	ABC*DE
+(- (-	$F) * G) * H$	ABC*DE/
+(- (-	$F) * G) * H$	ABC*DE/
+(- (-	$) * G) * H$	ABC*DE/F
+(-	$* G) * H$	ABC*DE/F-
+(-*	$G) * H$	ABC*DE/F-
+(-*	$) * H$	ABC*DE/F-G
+	$* H$	ABC*DE/F-G*-
++	H	ABC*DE/F-G*-
++	End	ABC*DE/F-G*-H
Empty	End	ABC*DE/F-G*-H*+

Example: Suppose we are converting $3 * 3 / (4 - 1) + 6 * 2$ expression into postfix form.

Following table shows the evaluation of Infix to Postfix:

Expression	Stack	Output
3	Empty	3
*	*	3
3	*	33
/	/	33*
(/(33*
4	/(33*4
-	/(-	33*4

1	/(-	33*41
)	-	33*41-
+	+	33*41-/
6	+	33*41-/6
*	+	33*41-/62
2	+	33*41-/62
	Empty	33*41-/62*+

So, the Postfix Expression is **33*41-/62*+**

Example: Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where ^ is an exponential operator.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(Start
2.	A	(A	
3.	+	(+	A	
4.	((+(A	
5.	B	(+(AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	((+(-(ABC*	
10.	D	(+(-(ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.)	(+(-	ABC*DEF^/	Pop from top on Stack , that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.)	(+	ABC*DEF^/G*-	Pop from top on Stack , that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.)	Empty	ABC*DEF^/G*-H*+	END

Resultant Postfix Expression: ABC*DEF^/G*-H*+

Example: 1. $A * B + C$ becomes $A B * C +$

The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '*', so the '*' must be printed first.

We will show this in a table with three columns. The first will show the symbol currently being read. The second will show what is on the stack and the third will show the current contents of the postfix string. The stack will be written from left to right with the 'bottom' of the stack to the left.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * {pop and print the '*' before pushing the '+'}
5	C	+	A B * C
6			A B * C +

The rule used in lines 1, 3 and 5 is to print an operand when it is read. The rule for line 2 is to push an operator onto the stack if it is empty. The rule for line 4 is if the operator on the top of the stack has higher precedence than the one being read, pop and print the one on top and then push the new operator on. The rule for line 6 is that when the end of the expression has been reached, pop the operators on the stack one at a time and print them.

2. $A + B * C$ becomes $A B C * +$

Here the order of the operators must be reversed. The stack is suitable for this, since operators will be popped off in the reverse order from that in which they were pushed.

	current symbol	operator stack	postfix string
1	A		A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6			A B C * +

In line 4, the '*' sign is pushed onto the stack because it has higher precedence than the '+' sign which is already there. Then when they are both popped off in lines 6 and 7, their order will be reversed.

3. $A * (B + C)$ becomes $A B C + *$

A subexpression in parentheses must be done before the rest of the expression.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(* (A B

4	B	* (A B
5	+	* (+	A B
6	C	* (+	A B C
7)	*	A B C +
8			A B C + *

Since expressions in parentheses must be done first, everything on the stack is saved and the left parenthesis is pushed to provide a marker. When the next operator is read, the stack is treated as though it were empty and the new operator (here the '+' sign) is pushed on. Then when the right parenthesis is read, the stack is popped until the corresponding left parenthesis is found. Since postfix expressions have no parentheses, the parentheses are not printed.

4. $A - B + C$ becomes $A B - C +$

When operators have the same precedence, we must consider association. Left to right association means that the operator on the stack must be done first, while right to left association means the reverse.

	current symbol	operator stack	postfix string
1	A		A
2	-	-	A
3	B	-	A B
4	+	+	A B -
5	C	+	A B - C
6			A B - C +

In line 4, the '-' will be popped and printed before the '+' is pushed onto the stack. Both operators have the same precedence level, so left to right association tells us to do the first one found before the second.

5. $A * B ^ C + D$ becomes $A B C ^ * D +$

Here both the exponentiation and the multiplication must be done before the addition.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	^	* ^	A B
5	C	* ^	A B C
6	+	+	A B C ^ *
7	D	+	A B C ^ * D

8			A B C ^ * D +
---	--	--	---------------

When the '+' is encountered in line 6, it is first compared to the '^' on top of the stack. Since it has lower precedence, the '^' is popped and printed. But instead of pushing the '+' sign onto the stack now, we must compare it with the new top of the stack, the '*'. Since the operator also has higher precedence than the '+', it also must be popped and printed. Now the stack is empty, so the '+' can be pushed onto the stack.

6. A * (B + C * D) + E becomes A B C D * + * E +

	current symbol	operator stack	postfix string
1	A		A
2	*	*	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)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

1. Conversion from infix to postfix:

Character Scanned	Stack	Expression
((Empty
(((Empty
a	((a
*	((*	a
b	((*	ab
-	((-	ab*
(((-(ab*
c	((-(ab*c
+	((-(+	ab*c
d	((-(+	ab*cd

Character Scanned	Stack	Expression
/	((-(+ /	ab*cd
e	((-(+ /	ab*cde
^	((-(+ / ^	ab*cde
f	((-(+ / ^	ab*cdef
)	((-	ab*cdef^/+
-	((-	ab*cdef^/+-
g	((-	ab*cdef^/+-g
)	(ab*cdef^/+-g-
*	(*	ab*cdef^/+-g-
h	(*	ab*cdef^/+-g-h
)	Empty	abcdef^/+-g-h

Thus, the postfix expression is: $abcdef^/+-g-h$

Evaluating Postfix Expressions

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression has following general structure...

Operand1 Operand2 Operator

Example



Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+, -, *, / etc.), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Evaluation rule of a Postfix Expression states:

1. While reading the expression from left to right, push the element in the stack if it is an operand.

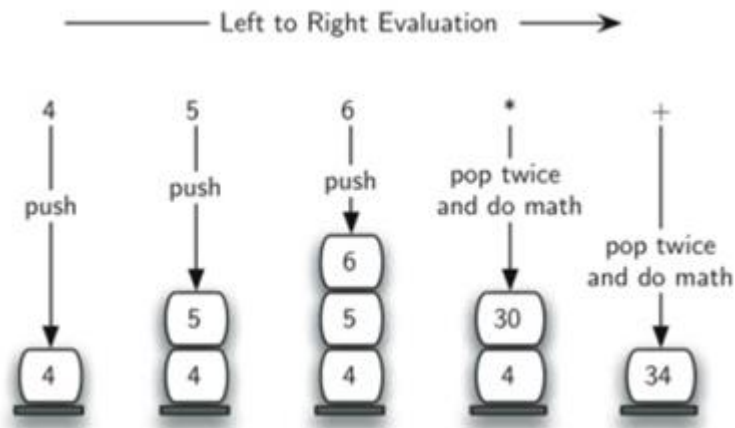
2. Pop the two operands from the stack, if the element is an operator and then evaluate it.
3. Push back the result of the evaluation. Repeat it till the end of the expression.

Algorithm

- 1) Add) to postfix expression.
- 2) Read postfix expression Left to Right until) encountered
- 3) If operand is encountered, push it onto Stack
[End If]
- 4) If operator is encountered, Pop two elements
 - i) A -> Top element
 - ii) B -> Next to Top element
 - iii) Evaluate B operator A
push B operator A onto Stack
- 5) Set result = pop
- 6) END

Let's see an example to better understand the algorithm:

Expression: 456*+



Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	$5*6=30$
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	$4+30=34$
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Result: 34










Example

Consider the following Expression...

Infix Expression **(5 + 3) * (8 - 2)**

Postfix Expression **5 3 + 8 2 - ***

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty 	Nothing
5	push(5) 	Nothing
3	push(3) 	Nothing
+	value1 = pop() value2 = pop() result = value2 + value1 push(result) 	value1 = pop(); // 3 value2 = pop(); // 5 result = 5 + 3; // 8 Push(8) (5 + 3)
8	push(8) 	(5 + 3)
2	push(2) 	(5 + 3)
-	value1 = pop() value2 = pop() result = value2 - value1 push(result) 	value1 = pop(); // 2 value2 = pop(); // 8 result = 8 - 2; // 6 Push(6) (8 - 2) (5 + 3), (8 - 2)
*	value1 = pop() value2 = pop() result = value2 * value1 push(result) 	value1 = pop(); // 6 value2 = pop(); // 8 result = 8 * 6; // 48 Push(48) (6 * 8) (5 + 3) * (8 - 2)
\$ End of Expression	result = pop() 	Display (result) 48 As final result

Infix Expression **(5 + 3) * (8 - 2) = 48**

Postfix Expression **5 3 + 8 2 - * value is 48**

$$2 * (3 + 4) - 6$$

The exact steps of the algorithm are put in the table below:

Input token	Operation	Stack contents (top on the right)	Details
2	Push on the stack	2	
3	Push on the stack	2, 3	
4	Push on the stack	2, 3, 4	
+	Add	2, 7	Pop two values: 3 and 4 and push the result 7 on the stack
*	Multiply	14	Pop two values: 2 and 7 and push the result 14 on the stack
6	Push on the stack	14, 6	
–	Subtract	8	Pop two values: 14 and 6 and push the result 8 on the stack
(End of tokens)	(Return the result)	8	Pop the only value 8 and return it

The contents of the stack in the *Stack contents* ... column is represented from left to right with the rightmost values being on the top of the stack. When there are no more tokens in the input, the contents of the stack is checked. If there is only one value, it is the result of the calculation. If there are no values or if there are many, the passed input expression was not a valid postfix expression.

Example:

Evaluate the expression 2 3 4 + * 5 * which was created by the previous algorithm for infix to postfix.

Move	Current Token	Stack (grows toward left)	
1	2		2
2	3		3 2
3	4		4 3 2
4	+		7 2
5	*		14
6	5		5 14
7	*		70

Notes:

Move 4: an operator is encountered, so 4 and 3 are popped, summed, then pushed back onto stack.

Move 5: operator * is current token, so 7 and 2 are popped, multiplied, pushed back onto stack.

Move 7: stack top holds correct value.

Notice that the postfix notation has been created to properly reflect operator precedence. Thus, postfix expressions never need parentheses.

Infix to Prefix Conversion:

Algorithm of Infix to Prefix

- Step 1. Push “)” onto STACK, and add “(“ to end of the A
- Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
- Step 3. If an operand is encountered add it to B
- Step 4. If a right parenthesis is encountered push it onto STACK
- Step 5. If an operator is encountered then:
- Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator.
 - Add operator to STACK
- Step 6. If left parenthesis is encountered then
- Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encountered)
 - Remove the left parenthesis
- Step 7. Exit

Expression = $(A+B^*C)*D+E^5$

Step 1. Reverse the infix expression.

$5^*E+D*(C^*B+A($

Step 2. Make Every '(' as ')' and every ')' as '('

$5^*E+D*(C^*B+A)$

Step 3. Convert expression to postfix form.

Step 4. Reverse the expression.

$+*+A^*BCD^*E5$

$A+(B*C-(D/E-F)*G)*H$

Expression	Stack	Output	Comment
$5^*E+D*(C^*B+A)$	Empty	-	Initial
$^*E+D*(C^*B+A)$	Empty	5	Print
$E+D*(C^*B+A)$	^	5	Push
$+D*(C^*B+A)$	^	5E	Push
$D*(C^*B+A)$	+	5E^	Pop And Push
$*(C^*B+A)$	+	5E^D	Print
(C^*B+A)	+*	5E^D	Push
$C^*B+A)$	+*(5E^D	Push
$^*B+A)$	+*(5E^DC	Print
$B+A)$	+*(^	5E^DC	Push
$+A)$	+*(^	5E^DCB	Print
$A)$	+*(+	5E^DCB^	Pop And Push
)	+*(+	5E^DCB^A	Print
End	+*	5E^DCB^A+	Pop Until '('
End	Empty	5E^DCB^A+*+	Pop Every element

Reverse the given expression string to obtain $h(g-(f^e/d+c)-ba)$. Enclosing the reversed string in brackets to obtain $(h(g-(f^e/d+c)-ba))$

Calculating the postfix expression of the reversed expression:

Character Scanned	Stack	Expression
((Empty
h	(h
*	(*	h
((*(h
g	(*(hg
-	(*(-	hg
((*(-(hg
f	(*(-(hgf
^	(*(-(^	hgf
e	(*(-(^	hgfe
/	(*(-(/	hgfe^
d	(*(-(/	hgfe^d
+	(*(-(+	hgfe^d/
c	(*(-(+	hgfe^d/c
)	(*(-	hgfe^d/c+
-	(*(-	hgfe^d/c+-
b	(*(-	hgfe^d/c+-b
*	((-	hgfe^d/c+-b
a	((-	hgfe^d/c+-ba
)	(*	hgfe^d/c+-ba*-
)	Empty	hgfe^d/c+-ba-

Thus, the postfix expression obtained is: $hgfe^d/c+-ba-$

Reversing the postfix expression obtained gives the prefix expression.

Prefix expression: $-ab-+c/d^efgh$

Infix to prefix implementation in c : without Pointer

```
# include <stdio.h>
# include <string.h>
# define MAX 20
```

```

void infixtoprefix(char infix[20],char prefix[20]);
void reverse(char array[30]);
char pop();
void push(char symbol);
int isOperator(char symbol);
int prcd(symbol);
int top=-1;
char stack[MAX];
main() {
    char infix[20],prefix[20],temp;
    printf("Enter infix operation: ");
    gets(infix);
    infixtoprefix(infix,prefix);
    reverse(prefix);
    puts((prefix));
}
//-----
void infixtoprefix(char infix[20],char prefix[20]) {
    int i,j=0;
    char symbol;
    stack[++top]='#';
    reverse(infix);
    for (i=0;i<strlen(infix);i++) {
        symbol=infix[i];
        if (isOperator(symbol)==0) {
            prefix[j]=symbol;
            j++;
        } else {
            if (symbol=='(') {
                push(symbol);
            } else if(symbol == '(') {
                while (stack[top]!='(') {
                    prefix[j]=pop();
                    j++;
                }
                pop();
            } else {
                if (prcd(stack[top])<=prcd(symbol)) {
                    push(symbol);
                } else {
                    while(prcd(stack[top])>=prcd(symbol)) {
                        prefix[j]=pop();
                        j++;
                    }
                    push(symbol);
                }
            }
            //end for else
        }
    }
    //end for else
}
//end for for
while (stack[top]!='#') {

```



```

        prefix[j]=pop();
        j++;
    }
    prefix[j]='\0';
}
////-----
void reverse(char array[30]) // for reverse of the given expression {
    int i,j;
    char temp[100];
    for (i=strlen(array)-1,j=0;i+1!=0;--i,++j) {
        temp[j]=array[i];
    }
    temp[j]='\0';
    strcpy(array,temp);
    return array;
}
//-----
char pop() {
    char a;
    a=stack[top];
    top--;
    return a;
}
//-----
void push(char symbol) {
    top++;
    stack[top]=symbol;
}
//-----
int prcd(symbol) // returns the value that helps in the precedence {
    switch(symbol) {
        case '+':
            case '-':
                return 2;
            break;
        case '*':
            case '/':
                return 4;
            break;
        case '$':
            case '^':
                return 6;
            break;
        case '#':
            case '(':
            case ')':
                return 1;
            break;
    }
}
//-----
int isOperator(char symbol) {
    switch(symbol) {

```

```

        case '+':
            case '-':
            case '*':
            case '/':
            case '^':
            case '$':
            case '&':
            case '(':
            case ')':
            return 1;
        break;
    default:
        return 0;
    // returns 0 if the symbol is other than given above
}
}

```

Evaluation of Prefix Expression

- One stack (call it Evaluation) holds the operators (like +, sin etc) and operands (like 3,4 etc) and the other stack (call it Count) holds a tuple of the number of operands left to be seen + the number of operands an operator expects.
- Anytime you see an operator, you push the operator onto the Evaluation stack and push the corresponding tuple onto the Count stack.
- Anytime you see an operand (like 3,5 etc), you check the top tuple of the Count stack and decrement the number of operands left to be seen in that tuple.
- If the number of operands left to be seen becomes zero, you pop the tuple from the Count stack. Then from the Evaluation stack you pop off the number of operands required (you know this because of the other value of the tuple), pop off the operator and do the operation to get a new value, (or operand).
- Now push the new operand back on the Evaluation stack. This new operand push causes you to take another look at the top of the Count stack and you do the same thing we just did (decrement the operands seen, compare with zero etc).
- If the operand count does not become zero, you continue with the next token in the input.

For example say you had to evaluate $+ 3 + 4 / 20 4$

The stacks will look like (left is the top of the stack)

Count	Evaluation	Input
		$+ 3 + 4 / 20 4$
(2,2)	+	$3 + 4 / 20 4$
(2,1)	3 +	$+ 4 / 20 4$
(2,2) (2,1)	+ 3 +	$4 / 20 4$
(2,1) (2,1)	4 + 3 +	$/ 20 4$
(2,2) (2,1) (2,1)	/ 4 + 3 +	20 4
(2,1) (2,1) (2,1)	20 / 4 + 3 +	4

(2,0) (2,1) (2,1) 4 8 / 4 + 3 +

Since it has become zero, you pop off two operands, the operator / and evaluate and push back 5. You pop off (2,0) from the Count stack.

(2,1) (2,1) 5 4 + 3 +

Pushing back you decrement the current Count stack top.

(2,0) (2,1) 5 4 + 3 +

Since it has become zero, you pop off 5,4 and + and evaluate and push back 9. Also pop off (2,0) from the count stack.

(2,0) 9 3 +

12

Example: - * + 4 3 2 5

Symbol	opnd1	opnd2	value	opndstack
5				5
2				5, 2
3				5, 2, 3
4				5, 2, 3, 4
+	4	3	7	5, 2
				5, 2, 7
*	7	2	14	5
				5, 14
-	14	5	9	
				9

result

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#define Max 20
int st[Max], top=-1;
```

```
void push(int ch)
{
    if (top == Max-1)
    {
        printf("Stack is full\n");
    }
    else
    {
        top++;
        st[top]=ch;
    }
}
```

```

int pop()
{
    int ch;
    if (top==-1)
    {
        printf("Stack is empty\n");
    }
    else
    {
        ch=st[top];
        top--;
    }
    return ch;
}

void dispstack()
{
    int k;
    printf("stack Content: ");
    for (k=top; k>=0; k--)
    {
        printf("%d, ", st[k]);
    }
    printf("\n");
}

int PreEval(char s[25])
{
    char temp[25];
    int i,val=0,ch1,ch2,j=0;
    i=0; top=-1;
    while (s[i]!='\0')
    {
        /*if operand is countered print it*/
        if ( (s[i]>=48 && s[i]<=57) )
        {
            j=0;
            temp[j]=s[i];
            j++;
            temp[j]='\0';
            push(atoi(temp));
        }
        else
        {
            ch2=pop();
            ch1=pop();
            switch(s[i])

```

```

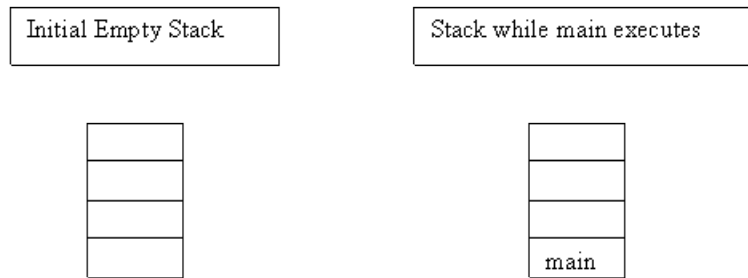
    {
    case '+':{
        val=ch2+ch1;
        break;
    }
    case '-':{
        val=ch2-ch1;
        break;
    }
    case '*':{
        val=ch2*ch1;
        break;
    }
    case '/':{
        val=ch2/ch1;
        break;
    }
    }
    push(val);
}
i++;
}
val=pop();
return val;
}
void main()
{
    char s[25],s1[25];
    int val;
    clrscr();
    printf("enter a Prefix expression for evaluation\n");
    scanf("%s",s);
    strcpy(s1,strev(s));
    val= PreEval(s1);
    printf("Value of Prefix Expression=%d\n", val);
    getch();
}

```

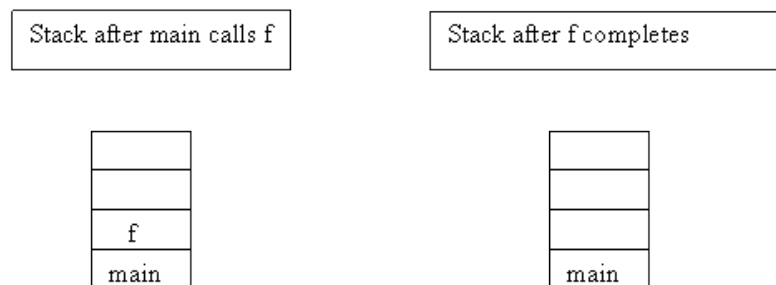
Recursion

How stack applies to recursion...

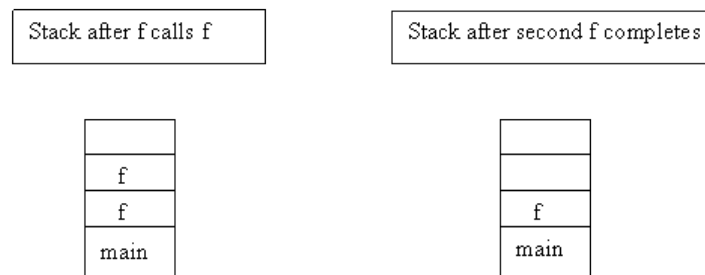
Every executable's main is loaded into a program stack at the beginning of execution. It remains there until it completes, at which time it is popped off of the stack.



If main calls a function, that function is loaded onto the top of the stack and remains there until it is complete at which point it is popped off of the stack.



Now, a recursive function calls itself. That means another instance of the function will be placed on the stack and will remain there until the function completes.



You need to look at a recursive function in terms of the program stack. Lets use factorial as an example. 5 factorial is $5 \times 4 \times 3 \times 2 \times 1 = 120$ and this can be implemented recursively.

```
int f(int x){
    if(x == 1) return 1; // line 1
    return f(x-1)*x;    // line 2
}
void main(){
    int y = f(5); // main call
    // y should get 120
}
```

So lets watch the stack and see what happens.

	f(5) // line 1 false return 5*f(4)
main() y = f(5)	main() y = f(5)

main calls the function f with a value of 5, so on the stack we get f(5). Line 1 is false so we go to line 2 which calls f(4), etc. Note that f(4) must complete before f(5) can complete. f(5) will complete by returning 5*f(4). The stack will look like:

f(1) true return 1;
f(2) false return 2*f(1)
f(3) false return 3*f(2)
f(4) false return 4*f(3)
f(5) // line 1 false return 5*f(4)
main() y = f(5)

So at this point none of the functions have yet returned! The first to return will be f(1) which will return 1. Then f(2) will return 2. Then f(3) will return 6. As in:

f(1) true return 1;	POP		
f(2) false return 2*f(1)	f(2) false return 2*1	POP	
f(3) false return 3*f(2)	f(3) false return 3*f(2)	f(3) false return 3*2	POP
f(4) false return 4*f(3)	f(4) false return 4*f(3)	f(4) false return 4*f(3)	f(4) false return 4*6
f(5) // line 1 false return 5*f(4)	f(5) // line 1 false return 5*f(4)	f(5) // line 1 false return 5*f(4)	f(5) // line 1 false return 5*f(4)
main() y = f(5)	main() y = f(5)	main() y = f(5)	main() y = f(5)

POP		
f(5) // line 1 false return 5*24	POP	
main() y = f(5)	main() y = 120	POP

Find the sum of elements of an array.

Given array

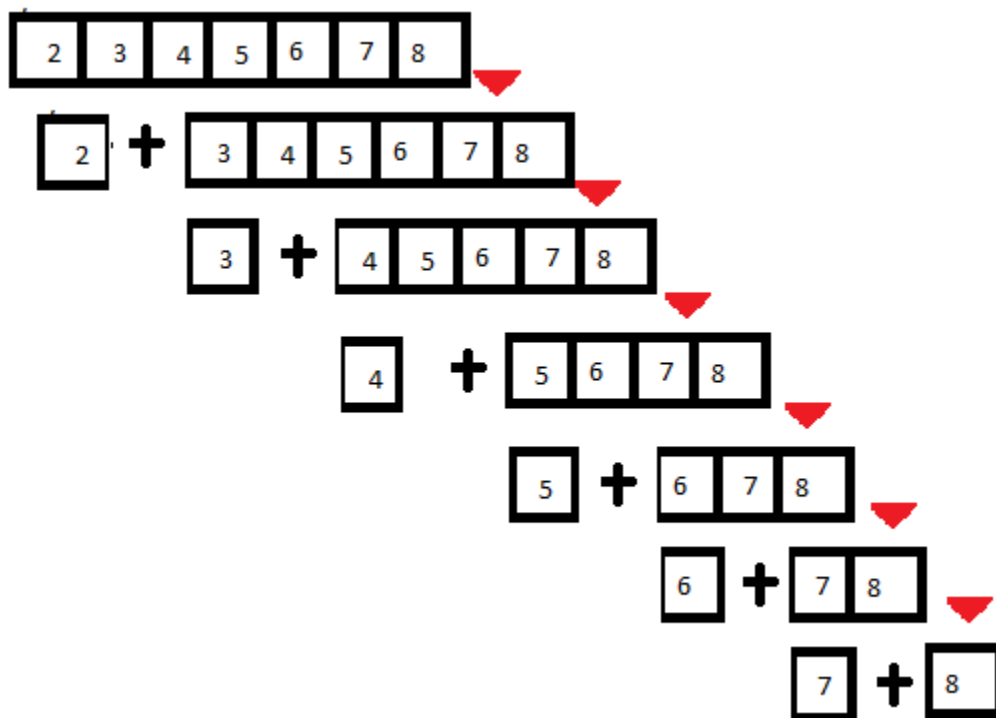
1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

We can easily deduce that sum of array is equal to :

First element + sum of the remaining elements of the array i.e

1	+	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

Here we get two parts: **first element+ an array of rest of elements**. Now the second part itself is an array whose sum is again **its first element +sum of the remaining elements of the array**. and so on.



We can easily visualize that our original problem has been successfully expressed in terms of itself. and that is recursion. We will code this example later below but first let's take one more example to confirm we have actually understood the concept behind recursion.

(Problem 2) : find the factorial of a number.

We know :

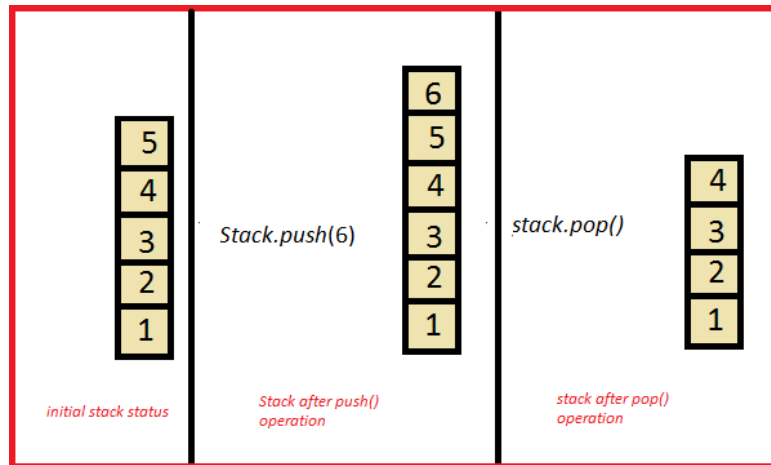
$$5! = (5 \times 4 \times 3 \times 2 \times 1) \Rightarrow (5) \times (4 \times 3 \times 2 \times 1) \Rightarrow (5) \times (4!)$$

we can see that in general $n! = n \times (n - 1)!$

Here also we can find that the problem of finding factorial can be expressed in terms of itself
i.e: **number X factorial(number-1)**

what happens behind the scenes(in the memory) when a function calls itself?

To understand this the understanding of a very basic data structure (STACK) is required. The below diagram shows push () and pop() operation on the stack. In this addition(**called as push**) and deletion(**called as pop**) is performed only on the top of the stack.

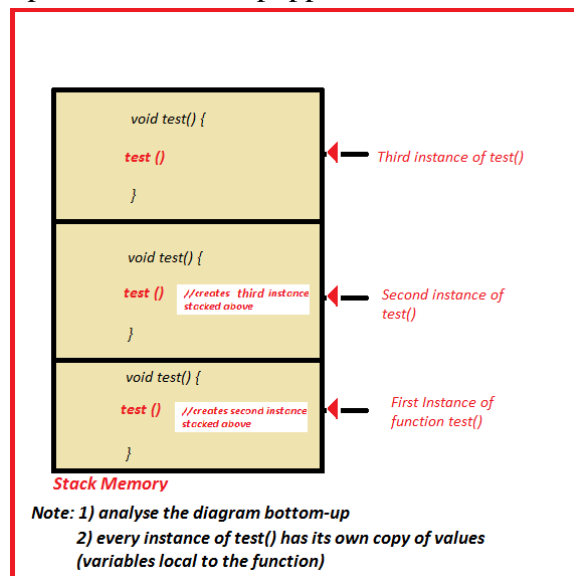


consider the code below :

```
Test(){
    Test();
```

```
}
```

Every time the function calls itself, a copy of it is created and pushed onto the stack and this keeps on going until the condition for breaking out of recursion is met or the stack memory is full. The below diagram depicts the status of stack in case of a recursive call of a function *test()*. Then the function instance from the top is executed and popped out until all instances are executed.



Two ways of thinking

For something simple to start with – let's write a function `pow(x, n)` that raises `x` to a natural power of `n`. In other words, multiplies `x` by itself `n` times.

`pow(2, 2) = 4`

```
pow(2, 3) = 8
pow(2, 4) = 16
```

There are two ways to implement it.

Iterative thinking: the for loop:

```
function pow(x, n)
{
    let result = 1;
    // multiply result by x n times in the loop
    for (let i = 0; i < n; i++)
    {
        result *= x;
    }
    return result;
}

alert( pow(2, 3) ); // 8
```

Recursive thinking: simplify the task and call self:

```
function pow(x, n) {
    if (n == 1) {
        return x;
    } else {
        return x * pow(x, n - 1);
    }
}

alert( pow(2, 3) ); // 8
```

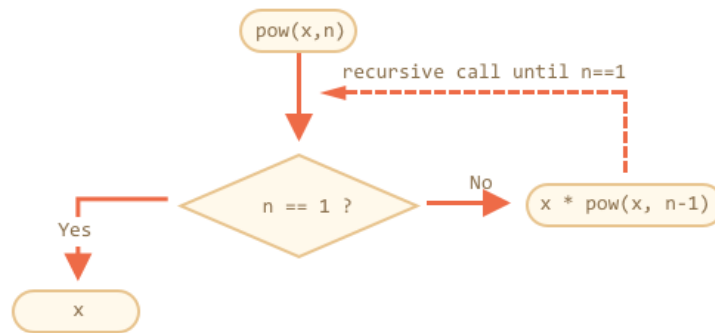
Please note how the recursive variant is fundamentally different.

When `pow(x, n)` is called, the execution splits into two branches:

```
      if n==1  = x
      /
pow(x, n) =
      \
      else    = x * pow(x, n - 1)
```

1. If `n == 1`, then everything is trivial. It is called *the base* of recursion, because it immediately produces the obvious result: `pow(x, 1)` equals `x`.
2. Otherwise, we can represent `pow(x, n)` as `x * pow(x, n - 1)`. In maths, one would write $x^n = x * x^{n-1}$. This is called *a recursive step*: we transform the task into a simpler action (multiplication by `x`) and a simpler call of the same task (`pow` with lower `n`). Next steps simplify it further and further until `n` reaches 1.

We can also say that `pow` *recursively calls itself* till `n == 1`.



For example, to calculate `pow(2, 4)` the recursive variant does these steps:

1. `pow(2, 4) = 2 * pow(2, 3)`
2. `pow(2, 3) = 2 * pow(2, 2)`
3. `pow(2, 2) = 2 * pow(2, 1)`
4. `pow(2, 1) = 2`

So, the recursion reduces a function call to a simpler one, and then – to even more simpler, and so on, until the result becomes obvious.

Recursion is usually shorter

A recursive solution is usually shorter than an iterative one.

Here we can rewrite the same using the conditional operator `?` instead of `if` to make `pow(x, n)` more terse and still very readable:

```
function pow(x, n) {
  return (n == 1) ? x : (x * pow(x, n - 1));
}
```

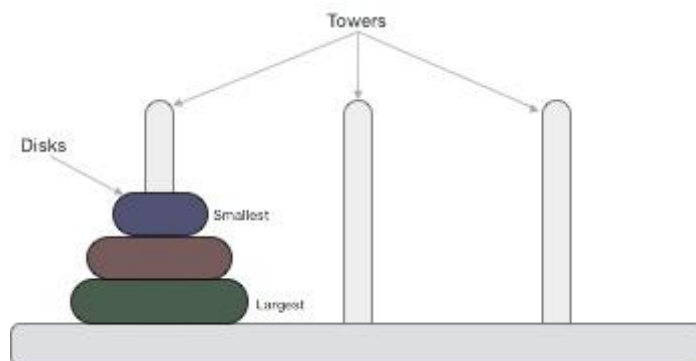
The maximal number of nested calls (including the first one) is called *recursion depth*. In our case, it will be exactly `n`.

The maximal recursion depth is limited by JavaScript engine. We can make sure about 10000, some engines allow more, but 100000 is probably out of limit for the majority of them. There are automatic optimizations that help alleviate this (“tail calls optimizations”), but they are not yet supported everywhere and work only in simple cases.

That limits the application of recursion, but it still remains very wide. There are many tasks where recursive way of thinking gives simpler code, easier to maintain.

Tower of Hanoi

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted –



These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

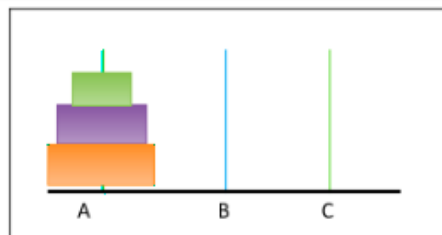
Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are –

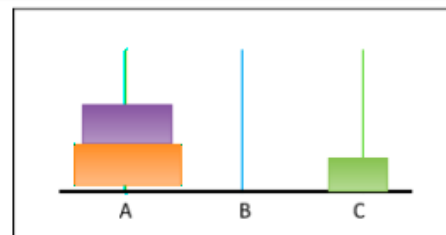
- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Understanding the problem and solution:

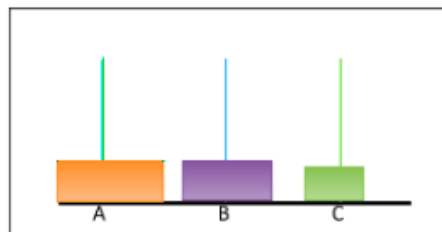
- In our case, let us take $n = 3$. Let us name the poles serially as A, B, C with A being the source pole and C being the destination. B will be used as a spare pole.



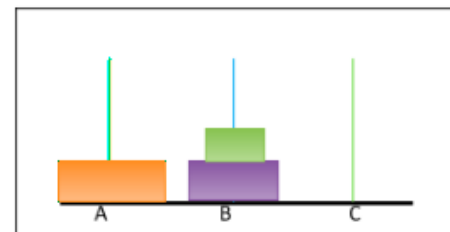
Initial State of the problem



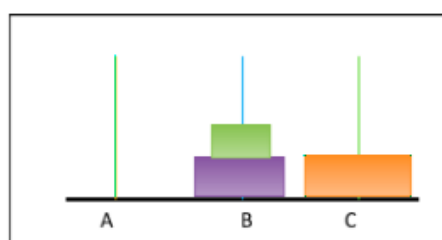
Step 1: Move The topmost disk to pole C



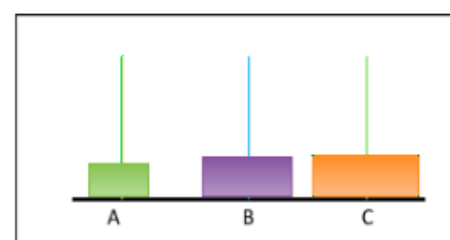
Step 2: Move the second disk to Pole B



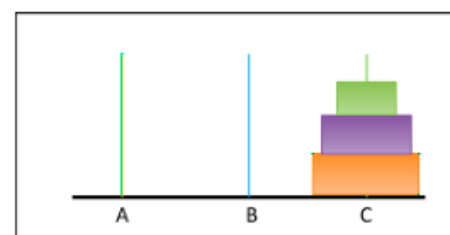
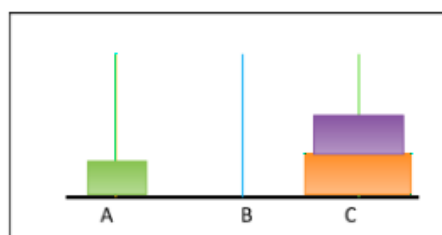
Step 3: Move the disk from pole C to pole B



Step 4: Move the largest disk from pole A to pole C



Step 5: Move the top disk from pole C to A



As seen above, we have moved the 3 disks following all the rules in the minimum number of steps for $n = 3$ which is 7.

The recursive task is to keep moving the disks from one pole to another pole.

Let us see how we shall arrive at our base case and the recursive case for our program.

As stated initially, our objective is to move all the disks placed on pole A – source pole to pole C – destination pole.

To transfer all disks from A to C, we would have to move the two upper disks i.e n-1 disks, from A to B which is the spare pole.

Now, once n-1 disks are placed on spare peg we can now move the largest/last i.e nth disk onto the destination pole C.

The last set of the task would involve moving disks from spare peg B to destination peg C.

Thus, our cases have been derived and they are as given below:

a.) Base case: if $n = 1$; Move the disk from A to C using B as spare.

b.) Recursive Case:

- Move n-1 rings from A to B using C as spare.
 - Move nth ring from A to C using B as spare.
-
- Move the n-1 rings from B to c using A as spare.

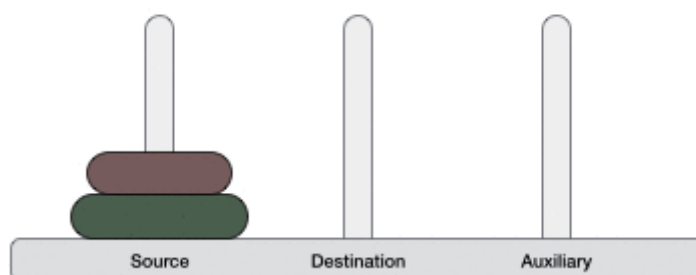
Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say $\rightarrow 1$ or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks –

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.

Step: 0



So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other ($n-1$) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other ($n-1$) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are –

Step 1 – Move n-1 disks from **source** to **aux**

Step 2 – Move n^{th} disk from **source** to **dest**

Step 3 – Move n-1 disks from **aux** to **dest**

A recursive algorithm for Tower of Hanoi can be driven as follows –

```
START
Procedure Hanoi(disk, source, dest, aux)
  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest)  // Step 1
    move disk from source to dest      // Step 2
    Hanoi(disk - 1, aux, dest, source)  // Step 3
  END IF
END Procedure
STOP
```

Program:

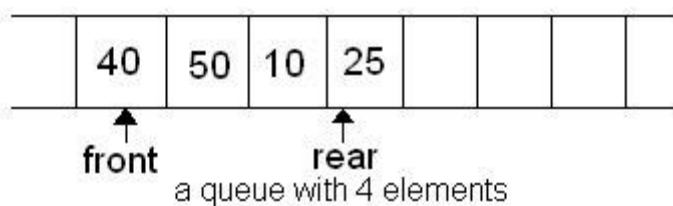
```
#include<stdio.h>
#include<conio.h>
int main ()
{
    clrscr();
    int n;
    printf("Enter number of disks required: \n");
    scanf ("%d", &n);
    TOH (n, 'A', 'B',' C');
    getch();
    return 0;
}

void TOH (int n, char src, char spare, char dest)
{
    if (n==1)
        printf("Move from %c to %c \n", src, dest);
    else
    {
        TOH(n-1, src, dest, spare) ;
        TOH(1, src, spare, dest);
        TOH(n-1, spare, src, dest);
    }
}
```

QUEUE(USING ARRAY)

A queue is a linear collection or linear list in which insertion can take place only at one end, called rear of the list, and deletions can take place only at other end, called front of the list,. The behaviour of a queue is like a First-In-First-Out (FIFO) system. In queue terminology, the insertion and deletion operations are known as enqueue and dequeue operations. For example- Queue can be implemented in following two ways:-

1. Queue (Using Array)
2. Queue (Using Linked List)



Memory Representation of Linear Queue(Using Array)

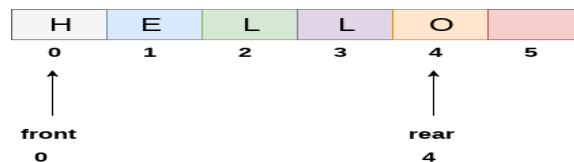
A stack in memory using array is defined as:-

```
#define CAPACITY 50
typedef struct nodetype
{
    int front, rear ;
    int element[CAPACITY];
} q;
```

Operations on Queue (Using Array)

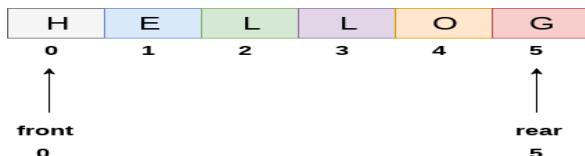
The following operations can be performed on Queue:-

- **Createqueue** - To create an empty queue
- **Enqueue** - To add (insert) and element in the queue
- **Dequeue** - To access and remove an element of the queue
- **Peek** - To access the first element of the queue without removing it.
- **Isfull** - To check whether the queue is full
- **Isempty** - To check whether the queue is empty



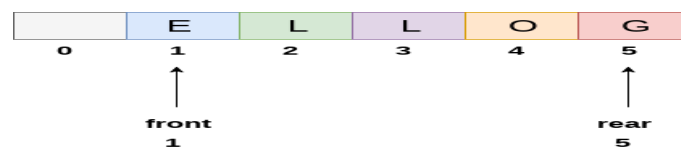
Queue

The above figure shows the queue of characters forming the English word "HELLO". Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.

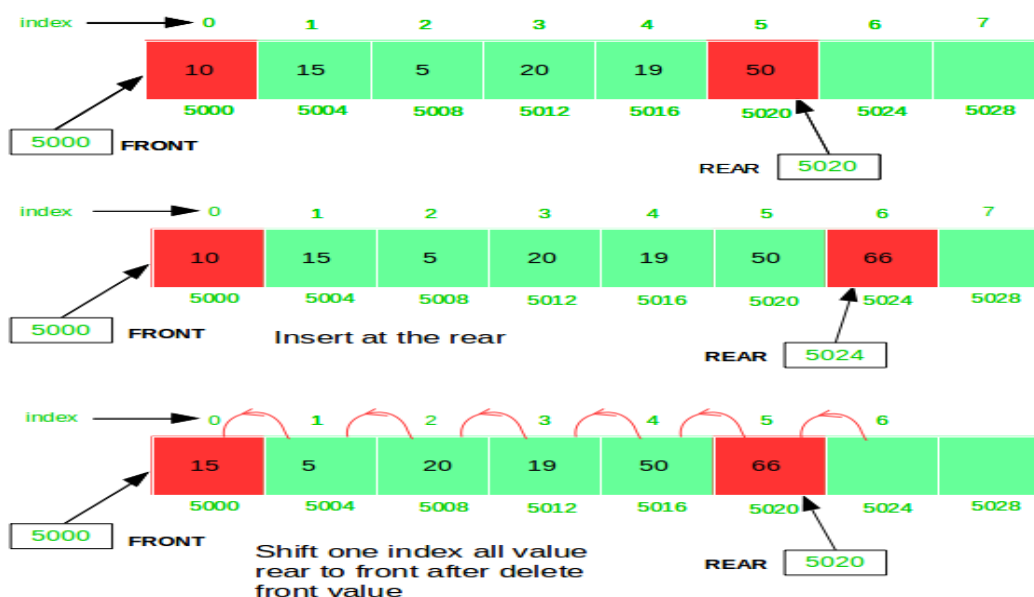


Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element



Algorithm to insert any element in a queue

- Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.
- If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.
- Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

- **Step 1:** IF REAR = MAX - 1
Write OVERFLOW
Go to step
[END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** EXIT

Algorithm to delete an element from the queue

- If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.
- Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

- Step 1: IF FRONT = -1 or FRONT > REAR
Write UNDERFLOW
ELSE
SET VAL = QUEUE[FRONT]
SET FRONT = FRONT + 1
[END OF IF]
- Step 2: EXIT

```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
    int choice;
    while(choice != 4)
    {
        printf("\n*****Main Menu*****\n");
        printf("\n=====");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d",&choice);
        switch(choice)
```

```

    {
        case 1:
            insert();
            break;
        case 2:
            delete();
            break;
        case 3:
            display();
            break;
        case 4:
            exit(0);
            break;
        default:
            printf("\nEnter valid choice??\n");
    }
}
}
void insert()
{
    int item;
    printf("\nEnter the element\n");
    scanf("\n%d",&item);
    if(rear == maxsize-1)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    if(front == -1 && rear == -1)
    {
        front = 0;
        rear = 0;
    }
    else
    {
        rear = rear+1;
    }
    queue[rear] = item;
    printf("\nValue inserted ");
}
void delete()
{
    int item;
    if (front == -1 || front > rear)
    {
        printf("\nUNDERFLOW\n");
    }
}

```

```

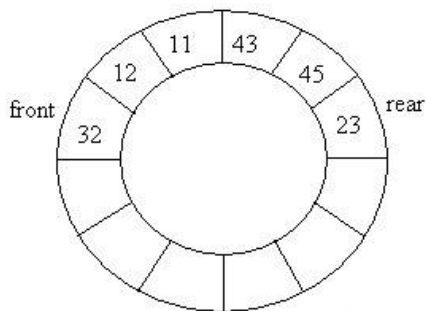
        return;
    }
    else
    {
        item = queue[front];
        if(front == rear)
        {
            front = -1;
            rear = -1 ;
        }
        else
        {
            front = front + 1;
        }
        printf("\nvalue deleted ");
    }
}

void display()
{
    int i;
    if(rear == -1)
    {
        printf("\nEmpty queue\n");
    }
    else
    {
        printf("\nprinting values ..... \n");
        for(i=front; i<=rear; i++)
        {
            printf("\n%d\n", queue[i]);
        }
    }
}

```

CIRCULAR QUEUE(Using Array)

Circular queue is a linear data structure as a linear queue in the form of circle.



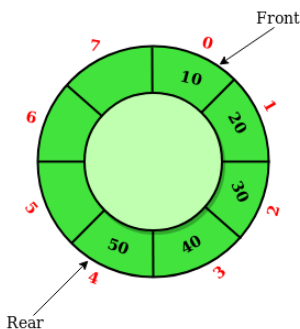
Circular queue with some values

Memory Representation of Circular Queue

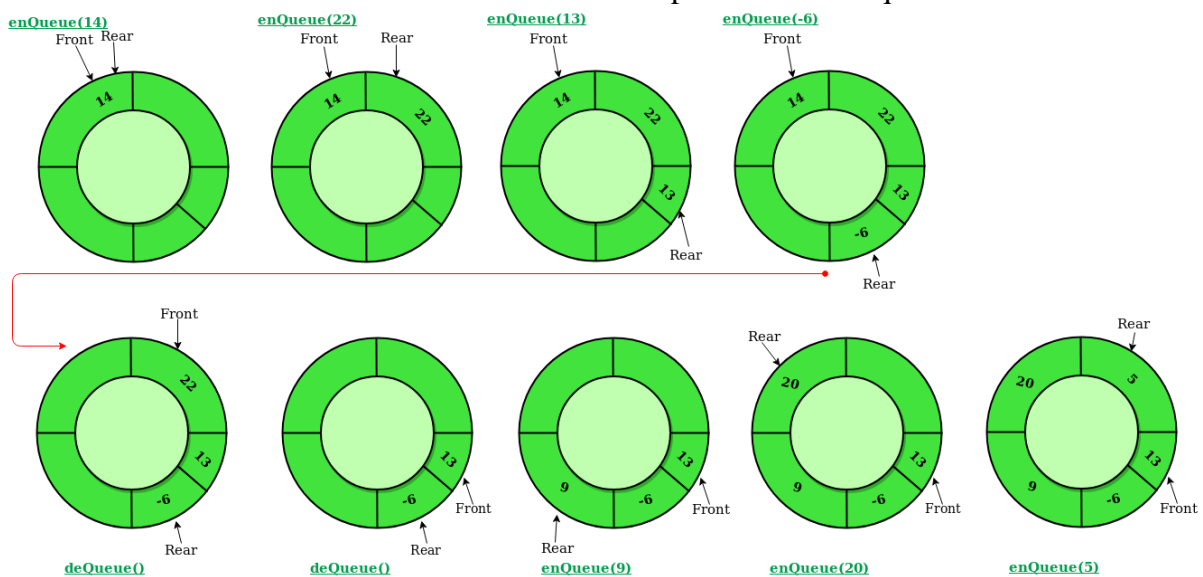
A Circular Queue in memory using array is represented as (same as Linear Queue):-

```
#define CAPACITY[50];  
struct cqueue  
{  
    int front, rear;  
    int element[CAPACITY];  
} cq;
```

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called '**Ring Buffer**'.

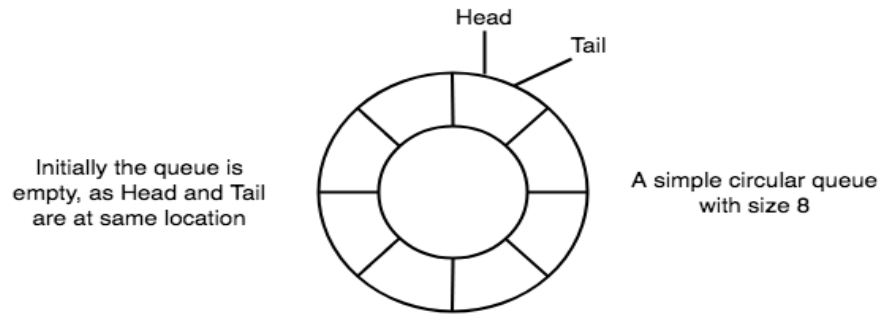


In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

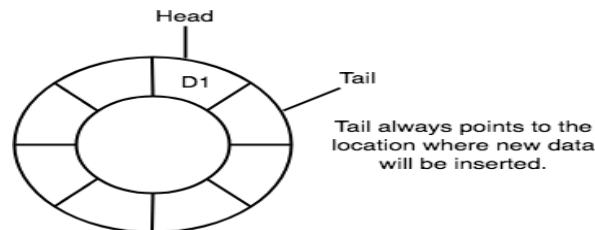


Basic features of Circular Queue

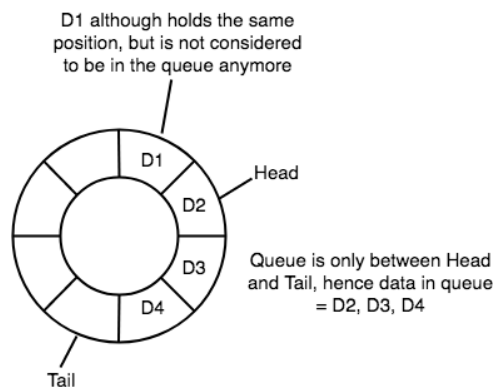
1. In case of a circular queue, head pointer will always point to the front of the queue, and tail pointer will always point to the end of the queue.
2. Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.



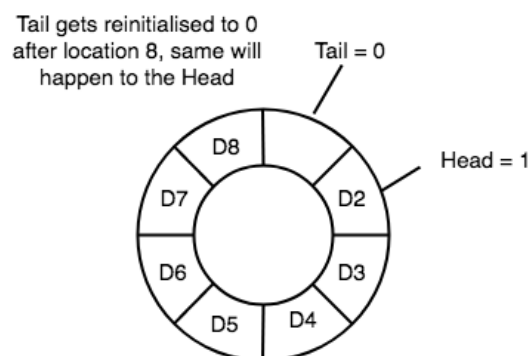
3. New data is always added to the location pointed by the tail pointer, and once the data is added, tail pointer is incremented to point to the next available location.



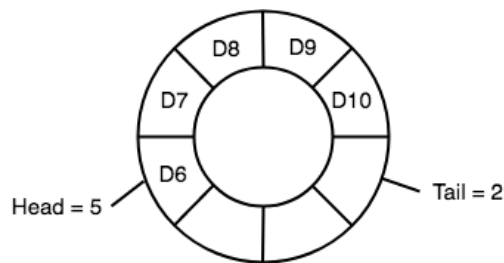
4. In a circular queue, data is not actually removed from the queue. Only the head pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between head and tail, hence the data left outside is not a part of the queue anymore, hence removed.



The head and the tail pointer will get reinitialised to 0 every time they reach the end of the queue.



5. Also, the head and the tail pointers can cross each other. In other words, head pointer can be greater than the tail. Sounds odd? This will happen when we dequeue the queue a couple of times and the tail pointer gets reinitialised upon reaching the end of the queue.



In such a situation the value of the Head pointer will be greater than the Tail pointer

Going Round and Round

- Another very important point is keeping the value of the **tail** and the **head** pointer within the maximum queue size.
- In the diagrams above the queue has a size of 8, hence, the value of **tail** and **head** pointers will always be between 0 and 7.
- This can be controlled either by checking every time whether **tail** or **head** have reached the **maxSize** and then setting the value 0 or, we have a better way, which is, for a value x if we divide it by 8, the remainder will never be greater than 8, it will always be between 0 and 0, which is exactly what we want.
- So the formula to increment the head and tail pointers to make them **go round and round** over and again will be, $\text{head} = (\text{head} + 1) \% \text{maxSize}$ or $\text{tail} = (\text{tail} + 1) \% \text{maxSize}$

Operations on Circular Queue

The following operations can be performed on Circular Queue:-

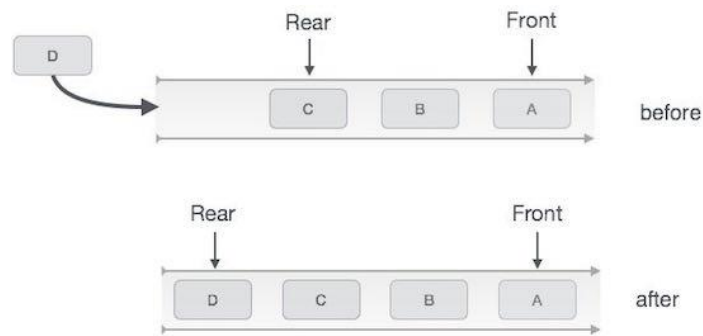
- **Createqueue** - To create an empty queue
- **Enqueue** - To add (insert) and element in the queue
- **Dequeue** - To access and remove an element of the queue
- **Peek** - To access the first element of the queue without removing it.
- **Isfull** - To check whether the queue is full
- **Isempty** - To check whether the queue is empty

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

- **Step 1** – Check if the queue is full.
- **Step 2** – If the queue is full, produce overflow error and exit.
- **Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Implementation of enqueue() in C programming language –

Example

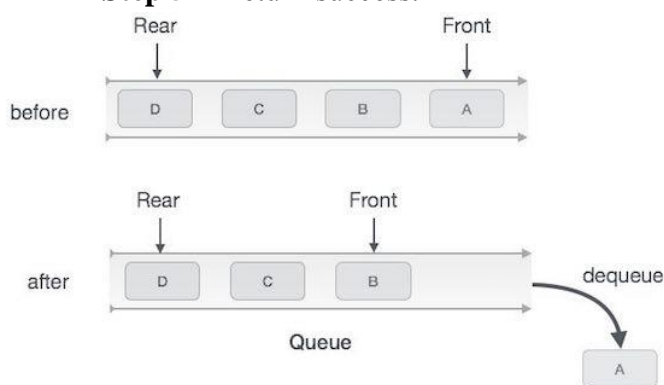
```
int enqueue(int data)
{
    if(isfull())
        return 0;

    rear = rear + 1;
    queue[rear] = data;
    return 1;
}
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Queue Dequeue

Implementation of dequeue() in C programming language –

Example

```
int dequeue() {
    if(isempty())
        return 0;
```

```

int data = queue[front];
front = front + 1;
return data;
}

```

Implementation of Circular Queue Datastructure using array - C Programming

```

#include<stdio.h>
#include<conio.h>
#define SIZE 5

void enQueue(int);
void deQueue();
void display();

int cQueue[SIZE], front = -1, rear = -1;

void main()
{
    int choice, value;
    clrscr();
    while(1){
        printf("\n***** MENU *****\n");
        printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1: printf("\nEnter the value to be insert: ");
                    scanf("%d",&value);
                    enQueue(value);
                    break;
            case 2: deQueue();
                    break;
            case 3: display();
                    break;
            case 4: exit(0);
            default: printf("\nPlease select the correct choice!!!\n");
        }
    }
}

void enQueue(int value)
{
    if((front == 0 && rear == SIZE - 1) || (front == rear+1))
        printf("\nCircular Queue is Full! Insertion not possible!!!\n");
    else{
        if(rear == SIZE-1 && front != 0)
            rear = -1;
        cQueue[++rear] = value;
    }
}

```



```

    printf("\nInsertion Success!!!\n");
    if(front == -1)
        front = 0;
}
}
void deQueue()
{
    if(front == -1 && rear == -1)
        printf("\nCircular Queue is Empty! Deletion is not possible!!!\n");
    else{
        printf("\nDeleted element : %d\n",cQueue[front++]);
        if(front == SIZE)
            front = 0;
        if(front-1 == rear)
            front = rear = -1;
    }
}
void display()
{
    if(front == -1)
        printf("\nCircular Queue is Empty!!!\n");
    else{
        int i = front;
        printf("\nCircular Queue Elements are : \n");
        if(front <= rear){
            while(i <= rear)
                printf("%d\t",cQueue[i++]);
        }
        else{
            while(i <= SIZE - 1)
                printf("%d\t", cQueue[i++]);
            i = 0;
            while(i <= rear)
                printf("%d\t",cQueue[i++]);
        }
    }
}
}

```

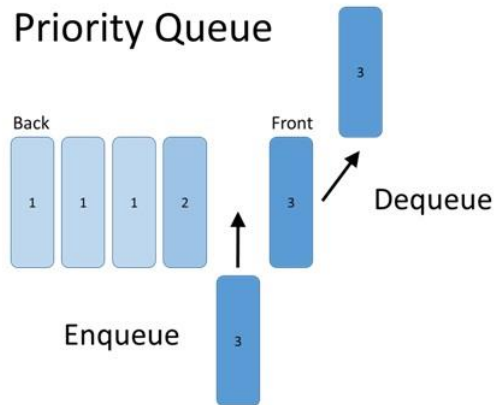
Application of Circular Queue

Below we have some common real-world examples where circular queues are used:

1. Computer controlled **Traffic Signal System** uses circular queue.
2. CPU scheduling and Memory management.

Priority Queue

Priority Queue

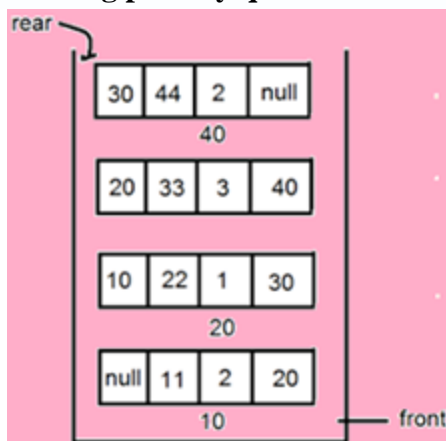


Priority queue is a collection of elements such that each element has been assigned a priority and the order in which elements are deleted and processed comes from following rules:

- An element of higher priority is processed before any elements of lower priority
- Two elements with the same priority are processed according to the order in which they were added to the queue.

An example of priority queue in computer science occurs in timesharing system in which the processes of higher priority are executed before any process of lower priority. There are two types of priority queue:

- **Ascending priority queue:** It is a collection of items in to which items can be inserted arbitrarily and from which only the smallest item can be removed.
- **Descending priority queue:** It is similar but allows deletion of only the largest item.



- Deletion order 22,11,44,33

Example: Program for Priority Queue

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5
```

```
void insert_by_priority(int);
void delete_by_priority(int);
void create();
void check(int);
void display_pqueue();
```

```
int pri_que[MAX];
```

```

int front, rear;

void main()
{
    int n, ch;
    printf("\n1 Insert");
    printf("\n2 Delete");
    printf("\n3 Display");
    printf("\n4 Exit");
    create();
    while (1)
    {
        printf("\nEnter your choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
            case 1:
                printf("\nEnter Element : ");
                scanf("%d", &n);
                insert_by_priority(n);
                break;
            case 2:
                printf("\nEnter Element to Delete : ");
                scanf("%d", &n);
                delete_by_priority(n);
                break;
            case 3:
                display_pqueue();
                break;
            case 4:
                exit(0);
            default:
                printf("\n Invalid Choice");
        }
    }
}

void create() //Create Function
{
    front = rear = -1;
}

void insert_by_priority(int data) //Insert Function
{
    if (rear >= MAX - 1)
    {
        printf("\nQueue is Overflow");
        return;
    }
}

```

```

    }
    if ((front == -1) && (rear == -1))
    {
        front++;
        rear++;
        pri_que[rear] = data;
        return;
    }
    else
        check(data);
    rear++;
}

void check(int data) //Check Function - to check priority and place element
{
    int i,j;
    for (i = 0; i <= rear; i++)
    {
        if (data >= pri_que[i])
        {
            for (j = rear + 1; j > i; j--)
            {
                pri_que[j] = pri_que[j - 1];
            }
            pri_que[i] = data;
            return;
        }
    }
    pri_que[i] = data;
}

void delete_by_priority(int data) //Delete Function
{
    int i;
    if ((front == -1) && (rear == -1))
    {
        printf("\nQueue is empty no elements to delete");
        return;
    }
    for (i = 0; i <= rear; i++)
    {
        if (data == pri_que[i])
        {
            for (; i < rear; i++)
            {
                pri_que[i] = pri_que[i + 1];
            }
            pri_que[i] = -99;
        }
    }
}

```

```

        rear--;
        if (rear == -1)
            front = -1;
        return;
    }
}
printf("\n%d not found in queue to delete", data);
}
void display_pqueue() //Display Function
{
    if ((front == -1) && (rear == -1))
    {
        printf("\nQueue is empty");
        return;
    }
    for (; front <= rear; front++)
    {
        printf(" %d ", pri_que[front]);
    }
    front = 0;
}

```

Applications of Queue

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Centre phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

There are various applications of computer science, which are performed using data structure queue. This data structure is usually used in-

- Simulation
- Various features of operating system
[Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.]
- Multi-programming platform systems
- Different type of scheduling algorithm
- Round robin technique or Algorithm
- Printer server routines
- Various applications software is also based on queue data structure
- Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

Limitations of Linear Queue(Using Array)

					40	50	10	25
					↑ front			
							↑ rear	

Prepared by Mrs. V. R. Sonar