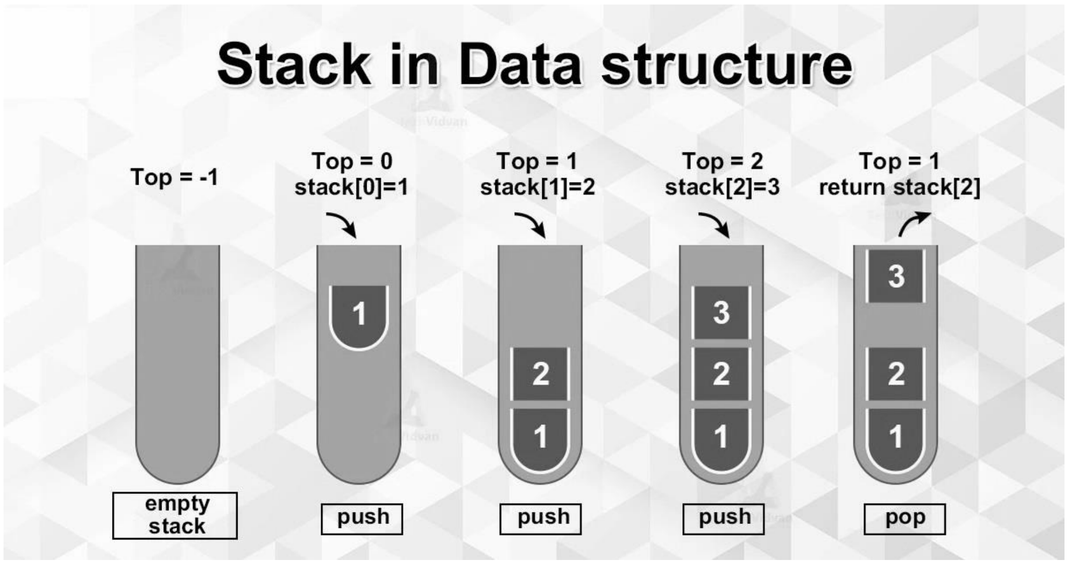
UNIT – IV

1.1 Stack:

A stack is an abstract data type that holds an ordered, linear sequence of items. In contrast to a queue, a stack is a **last in, first out** (LIFO) structure. A real-life example is a stack of plates: you can only take a plate from the top of the stack, and you can only add a plate to the top of the stack. If you want to reach a plate that is not on the top of the stack, you need to remove all of the plates that are above that one.

In the same way, in a stack data structure, you can only access the element on the top of the stack. The element that was added last will be the one to be removed first. Therefore, to implement a stack, you need to maintain a pointer to the **top** of the stack (the last element to be added).



There are some basic operations that allow us to perform different actions on a stack.
Push: Add an element to the top of a stack
Pop: Remove an element from the top of a stack
Peek: Get the value of the top element without removing it.
IsEmpty: Check if the stack is empty.
IsFull: Check if the stack is full.

1.2 Array implementations of Stack:

a) **Creation:**

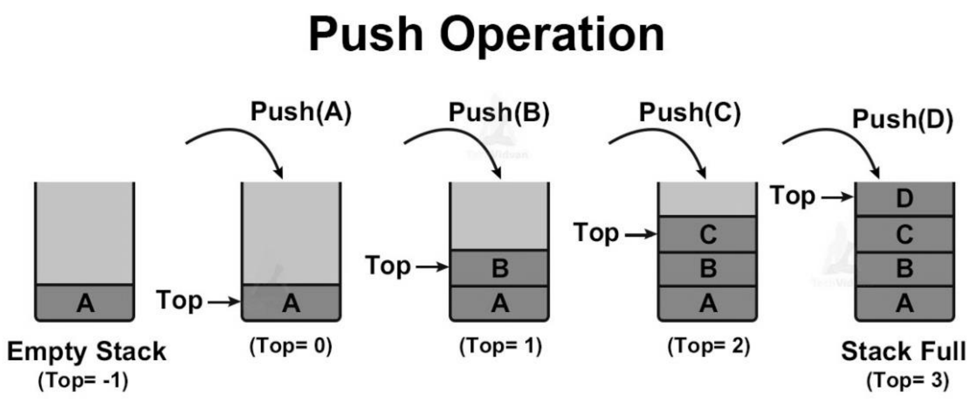
Before implementing actual operations, first follow the below steps to create an empty stack.

- **Step 1 -** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2 -** Declare all the **functions** used in stack implementation.
- **Step 3 -** Create a one dimensional array with fixed size (**int stack[SIZE]**)
- **Step 4 -** Define a integer variable **'top'** and initialize with **'-1'**. (**int top = -1**)
- **Step 5 -** In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

b) push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

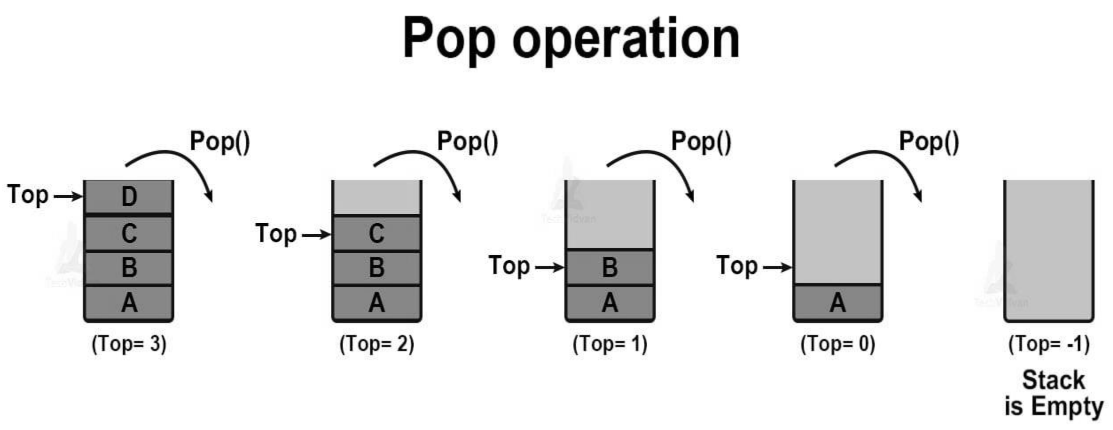
- **Step 1 -** Check whether **stack** is **FULL**. (**top == SIZE-1**)
- **Step 2 -** If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).



c) pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

- **Step 1 -** Check whether **stack** is **EMPTY**. (**top == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).



d) Peek() - ***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'.

e) IsEmpty: Check if the stack is empty.

Isempty() It returns TRUE if the stack is empty. Otherwise, it will return FALSE.

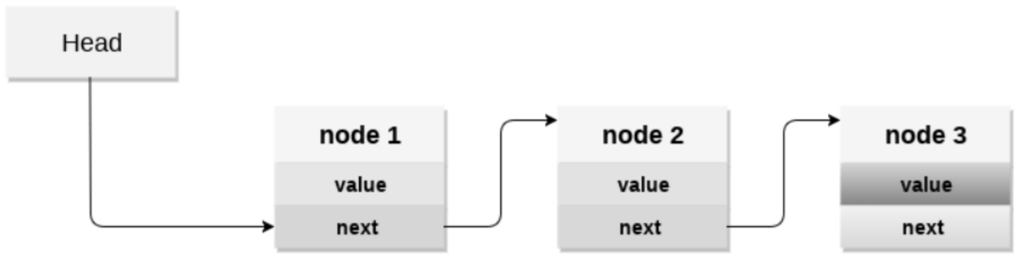
f) IsFull: Check if the stack is full.

IsFull( ) returns TRUE if the stack is full, else it returns FALSE.

1.3 Linked List Implementation of Stack:

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.



push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set **newNode → next** = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set **newNode → next** = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

```
void push ()
{
  int val;
  struct node *ptr =(struct node*)malloc(sizeof(struct node));
  if(ptr == NULL)
  {
    printf("not able to push the element");
  }
  else
  {
    printf("Enter the value");
    scanf("%d",&val);
    if(head==NULL)
    {
        ptr->val = val;
        ptr -> next = NULL;
        head=ptr;
    }
    else
    {
        ptr->val = val;
        ptr->next = head;
        head=ptr;
    }
    printf("Item pushed");
  }
}
```

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top = top → next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

```
void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
    else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped");

    }
}
```

display() - Displaying stack of elements

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

- **Step 1 -** Check whether stack is **Empty** (**top** == **NULL**).
- **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 Node pointer **'temp'** and initialize with **top**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

```
void display()
{
   int i;
   struct node *ptr;
   ptr=head;
   if(ptr == NULL)
   {
      printf("Stack is empty\n");
   }
   else
   {
      printf("Printing Stack elements \n");
      while(ptr!=NULL)
      {
         printf("%d\n",ptr->val);
         ptr = ptr->next;
      }
   }
}
```

1.4 Polish Notation:

Polish Notation is a general form of expressing mathematical, logical and algebraic equations. The compiler uses this notation in order to evaluate mathematical expressions depending on the order of operations. There are in general three types of Notations used while parsing Mathematical expressions:

- Infix Notation
- Prefix Notation
- Postfix Notation

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

Infix Notation or Expression is where the operators are written in between every pair of operands. It is the usual way to write an expression generally written with parentheses. For Ex: An expression like X+Y is an Infix Expression, where + is an Operator and X, Y are Operands.

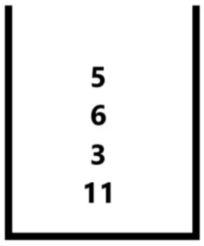
Polish Notation is also known as Prefix Notation or Expression. In this type of arithmetic expression, the operators precede the operands i.e. the operators are written before the Operands. The operators are placed left for every pair of operands. So, for the above Infix X+Y, its equivalent Polish or Prefix Notation is +XY.

Example

Now, let us look at an example on how to evaluate a Polish Notation or Prefix Expression to get the result.
Consider this Expression: **/ * + 5 6 3 11.** While evaluating the expression we take decision for two cases: When the Character is an Operand or When the Character is an Operator. Let us look at the steps.

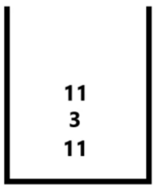
Step 1:

We will use a Stack for this evaluation. We scan the Expression from right to left, if the current character is an Operand we push it into the stack. So from 11 to 5 we push the elements into the stack. The stack looks:



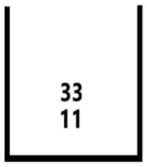
Step 2:

As soon as we get an operator we multiply its previous two elements, so continuing traversing from right to left we first get **'+'** operator so we pop two elements from stack (5 & 6) compute their result with the operator i.e. 5+6 = 11, and push the result back into the stack for future evaluation. The Stack now is:



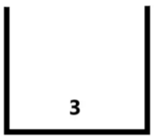
Step 3:

The next Operator is **'*'** Operator (Multiply), so we again pop the two elements from stack and repeating the process of Step 2. So we compute the result from their operation (11 * 3 =33) and push it back to the stack again. The stacks now look like:



Step 4:

Finally, we have the **'/'** operator so we pop 33 and 11 compute the result push it back to the stack. Now we have reached the leftmost or start index of the expression so at this point our stack will contains only one value which will be our Resultant Evaluated Prefix Expression.



1.5 **Postfix Evaluation:**

Postfix Expression is also known as **Reverse Polish Notation**. These are the expression where the Operands precede the Operators i.e. the Operands are written before the Operators. For Example: The Infix **X+Y** will be represented in Postfix or Reverse Polish as **XY+**

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation –

```
Step 1 – scan the expression from left to right
Step 2 – if it is an operand push it to stack
Step 3 – if it is an operator pull operand from stack and perform operation
Step 4 – store the output of step 3, back to stack
Step 5 – scan the expression until all operands are consumed
Step 6 – pop the stack and perform operation
```

Associativity:

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in 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**.

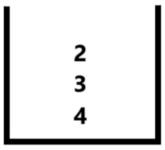
Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest)

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

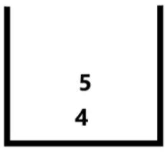
The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis.

Example
Now, let us see how to evaluate a given Postfix Expression. Consider this Reverse Polish or Postfix Expression: **4 3 2 + * 5 –** .

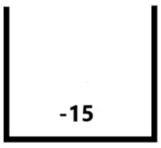
Step 1:
We will again use a Stack for this evaluation. Here, we scan the Expression from left to right, if the current character is an Operand we push it into the stack. So from 4 to 2 we push the elements into the stack. The stack looks:



Step 2
Now, on traversing next we get '+' operator, so we pop two elements from the stack compute their result and push it back again for future evaluation. The steps here are same as above discussed example. The difference is that in this case we traverse from left to right. The overall algorithm remains same. The stack now is:



Step 3:
Now, computing all the steps for each operator we get '*' so we pop 5 and 4 and push 5 * 4 = 20 into stack and then we get 5 so we push into stack then finally we get '-' operator so we compute their result 5-20 = -15, then we push it again, at the end index of the string we get the result of our Postfix evaluation. The stack finally has -15.



1.6 Infix to postfix Conversion:
Any expression can be represented using three types of expressions (Infix, Postfix, and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.
To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

1. Find all the operators in the given Infix Expression.
2. Find the order of operators evaluated according to their Operator precedence.
3. Convert each operator into required type of expression (Postfix or Prefix) in the same order.

Example

Consider the following Infix Expression to be converted into Postfix Expression...

*D = A + B * C*

- Step 1 - The Operators in the given Infix Expression : = , + , *
- Step 2 - The Order of Operators according to their preference : * , + , =
- Step 3 - Now, convert the first operator * ----- D = A + B C *
- Step 4 - Convert the next operator + ----- D = A BC* +
- Step 5 - Convert the next operator = ----- D ABC*+ =

Finally, given Infix Expression is converted into Postfix Expression as follows...

*D A B C * + =*

Infix to Postfix Conversion using Stack Data Structure

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
5. If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Example

Consider the following Infix Expression...

*(A + B) * (C - D)*

The given infix expression can be converted into postfix expression using Stack data Structure as follows...

The final Postfix Expression is as follows...

*A B + C D - **

Program:

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

char stack[100];
int top = -1;

void push(char x)
{
   stack[++top] = x;
}

char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}

int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
    return 0;
}

int main()
{
    char exp[100];
    char *e, x;
    printf("Enter the expression : ");
    scanf("%s",exp);
    printf("\n");
    e = exp;
        while(*e != '\0')
    {
        if(isalnum(*e))
            printf("%c ",*e);
        else if(*e == '(')
            push(*e);
        else if(*e == ')')
        {
            while((x = pop()) != '(')
                printf("%c ", x);
        }
        else
        {
            while(priority(stack[top]) >= priority(*e))
                printf("%c ",pop());
            push(*e);
        }
        e++;
    }

    while(top != -1)
    {
        printf("%c ",pop());
    }return 0;
}
```

1.7 Stacks and Recursion:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily.

Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. Many more recursive calls can be generated as and when required.

It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.

Need of Recursion

Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. It has certain advantages over the iteration technique which will be discussed later. A task that can be defined with its similar subtask, recursion is one of the best solutions for it. For example; The Factorial of a number.

Properties of Recursion:

- Performing the same operations multiple times with different inputs.
- In every step, we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion otherwise infinite loop will occur.

Algorithm: Steps

The algorithmic steps for implementing recursion in a function are as follows:
Step1 - Define a base case: Identify the simplest case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.
Step2 - Define a recursive case: Define the problem in terms of smaller sub problems. Break the problem down into smaller versions of itself, and call the function recursively to solve each sub problem.
Step3 - Ensure the recursion terminates: Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.
step4 - Combine the solutions: Combine the solutions of the sub problems to solve the original problem.

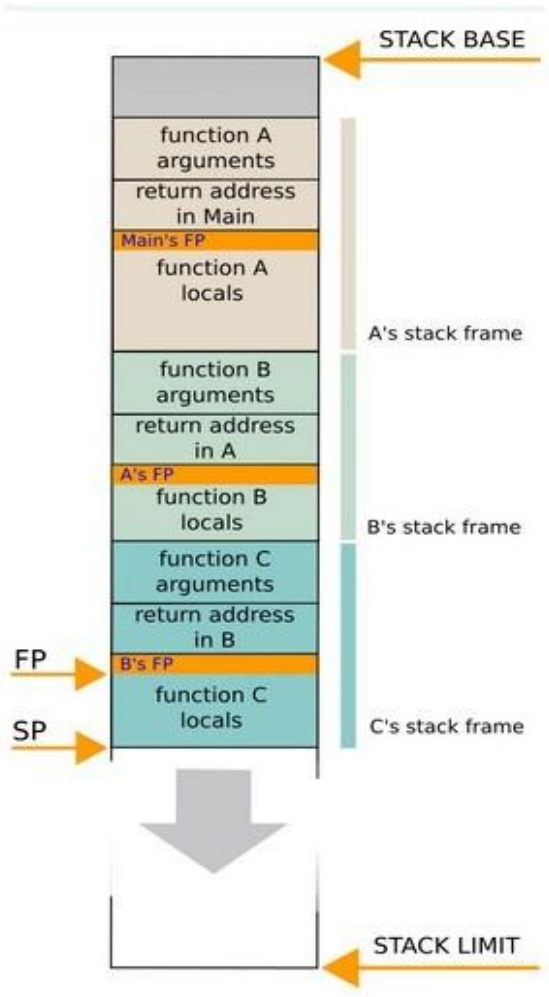
A Mathematical Interpretation

Let us consider a problem that a programmer has to determine the sum of first n natural numbers, there are several ways of doing that but the simplest approach is simply to add the numbers starting from 1 to n. So the function simply looks like this,

approach(1) – Simply adding one by one
$f(n) = 1 + 2 + 3 + \ldots\ldots + n$
but there is another mathematical approach of representing this,

approach(2) – Recursive adding
$f(n) = 1 \quad n=1$
$f(n) = n + f(n-1) \quad n>1$

There is a simple difference between the approach (1) and approach(2) and that is in **approach(2)** the function " **f( )** " itself is being called inside the function, so this phenomenon is named recursion, and the function containing recursion is called recursive function, at the end, this is a great tool in the hand of the programmers to code some problems in a lot easier and efficient way.



How are recursive functions stored in memory?

Recursion uses more memory, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished. The recursive function uses LIFO (LAST IN FIRST OUT) Structure just like the stack data structure. https://www.geeksforgeeks.org/stack-data-structure/

What is the base condition in recursion?

In the recursive program, the solution to the base case is provided and the solution to the bigger problem is expressed in terms of smaller problems.

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

In the above example, the base case for n < = 1 is defined and the larger value of a number can be solved by converting to a smaller one till the base case is reached.

Why Stack Overflow error occurs in recursion?

If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
  // wrong base case (it may cause
  // stack overflow).
  if (n == 100)
    return 1;

  else
    return n*fact(n-1);
}
```

If fact(10) is called, it will call fact(9), fact(8), fact(7), and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

What is the difference between direct and indirect recursion?

A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly. The difference between direct and indirect recursion has been illustrated in Table 1.

// An example of direct recursion

```
void directRecFun()
{
  // Some code....
  directRecFun();

  // Some code...
}
```

// An example of indirect recursion

```
void indirectRecFun1()
{
  // Some code...
  indirectRecFun2();
  // Some code...
}
void indirectRecFun2()
{
  // Some code...
  indirectRecFun1();
  // Some code...
}
```

1.8 Stack Reversing using Recursive function:

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

struct node
{
   int a;
   struct node *next;
};

void generate(struct node **);
void display(struct node *);
void stack_reverse(struct node **, struct node **);
void delete(struct node **);

int main()
{
   struct node *head = NULL;
   generate(&head);
   printf("\nThe sequence of contents in stack\n");
   display(head);
   printf("\nInversing the contents of the stack\n");
   if (head != NULL)
   {
      stack_reverse(&head, &(head->next));
   }
   printf("\nThe contents in stack after reversal\n");
   display(head);
   delete(&head);
   return 0;
}

void stack_reverse(struct node **head, struct node **head_next)
{
   struct node *temp;
   if (*head_next != NULL)
   {
       temp = (*head_next)->next;
      (*head_next)->next = (*head);
      *head = *head_next;
      *head_next = temp;
      stack_reverse(head, head_next);
   }
}

void display(struct node *head)
{
   if (head != NULL)
   {
      printf("%d  ", head->a);
```

```
        display(head->next);
    }
}

void generate(struct node **head)
{
    int num, i;
    struct node *temp;
    printf("Enter length of list: ");
    scanf("%d", &num);
    for (i = num; i > 0; i--)
    {
        temp = (struct node *)malloc(sizeof(struct node));
        temp->a = i;
        if (*head == NULL)
        {
            *head = temp;
            (*head)->next = NULL;
        }
        else
        {
            temp->next = *head;
            *head = temp;
        }
    }
}

void delete(struct node **head)
{
    struct node *temp;
    while (*head != NULL)
    {
        temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}
```

1.9 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.

Tower of Hanoi puzzle with n disks can be solved in minimum $\mathbf{2^n - 1}$ steps. This presentation shows that a puzzle with 3 disks has taken $\mathbf{2^3 - 1 = 7}$ steps.

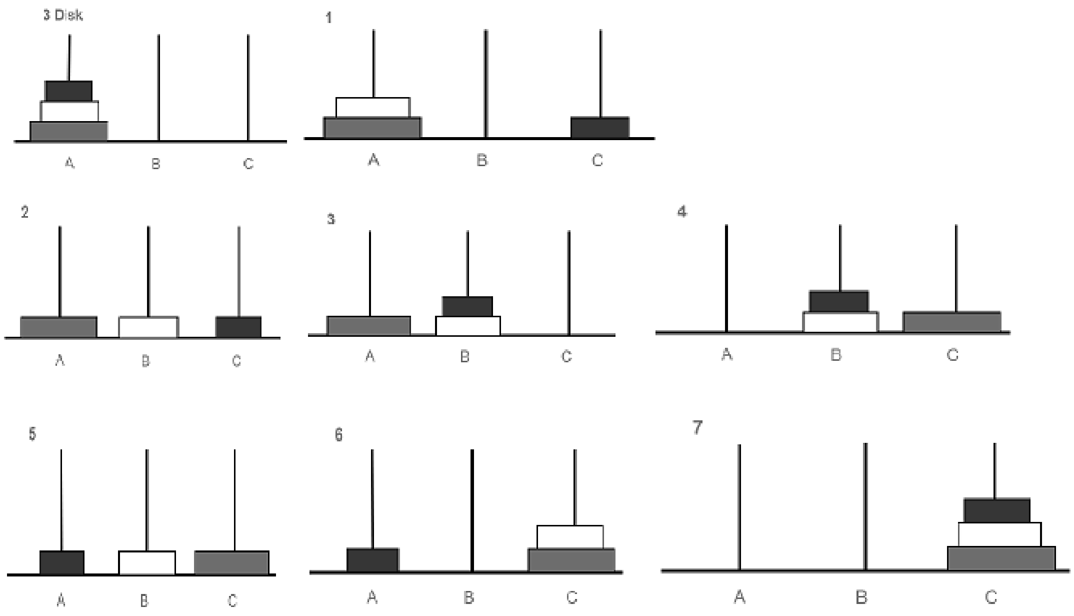
Algorithm

- To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 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.

  Step 1 – Move n-1 disks from **source** to **aux**
  Step 2 – Move nth 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
```



Sample Program

```
#include <stdio.h>
void toH(int n, char rodA, char rodC, char rodB)
{
        if (n == 1)
        {
                printf("\n Move disk 1 from rod %c to rod %c",rodA ,rodC );
                return;
        }
        toH(n-1, rodA, rodB, rodC);
        printf("\n Move disk %d from rod %c to rod %c", n, rodA, rodC);
        toH(n-1, rodB, rodC,rodA);
}

int main()
{
        int no_of_disks ;
        printf("Enter number of disks: ");
        scanf("%d", &no_of_disks);
        toH(no_of_disks, 'A','C','B');
        return 0;
}
```

Output:

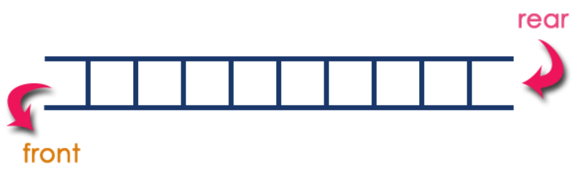
```
Enter number of disks: 3
 Move disk 1 from rod A to rod C
 Move disk 2 from rod A to rod B
 Move disk 1 from rod C to rod B
 Move disk 3 from rod A to rod C
 Move disk 1 from rod B to rod A
 Move disk 2 from rod B to rod C
 Move disk 1 from rod A to rod C
```

2.0 Queue:

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions.

The insertion is performed at one end and deletion is performed at another end. In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'.

In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.



"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle".

Operations on a Queue

The following operations are performed on a queue data structure...

1. **enQueue(value) - (To insert an element into the queue)**
2. **deQueue() - (To delete an element from the queue)**
3. **display() - (To display the elements of the queue)**

Queue data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

2.1 Queue using Arrays:

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. The implementation of queue data structure using array is very simple.

Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and '**rear**'. Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position.

Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

a) Queue Operations using Array

Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below steps to create an empty queue.

- **Step 1 -** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2 -** Declare all the **user defined functions** which are used in queue implementation.
- **Step 3 -** Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)

- **Step 4 -** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
- **Step 5 -** Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

b) enQueue(value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

- **Step 1 -** Check whether **queue** is **FULL**. (**rear == SIZE-1**)
- **Step 2 -** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear]** = **value**.

c) deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

d) display() - Displays the elements of a Queue

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

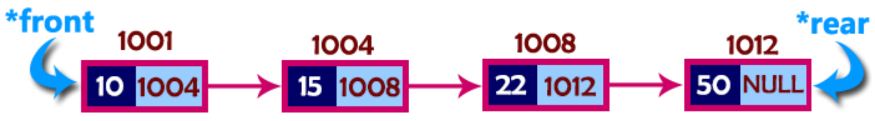
- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
- **Step 4 -** Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

2.2 Queue using Arrays:

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use.

A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation).

The Queue implemented using linked list can organize as many
Data values as we want. In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.



Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

- **Step 1 -** Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
- **Step 2 -** Check whether queue is **Empty** (**rear** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **rear → next** = **newNode** and **rear** = **newNode**.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

- **Step 1 -** Check whether **queue** is **Empty** (**front == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- **Step 4 -** Then set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

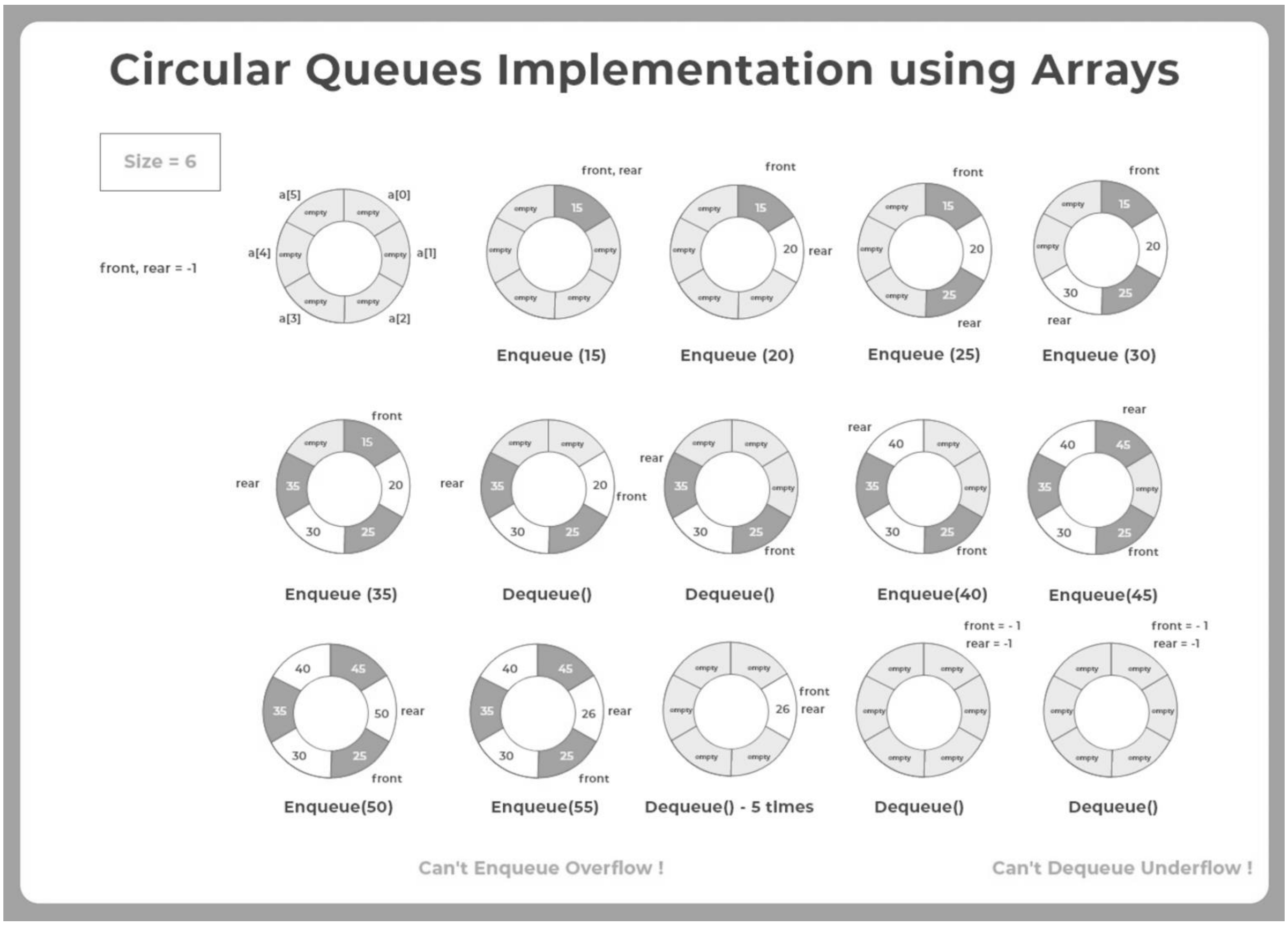
display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- **Step 1 -** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2 -** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

2.3 Circular Queue:

A circular queue is a linear data structure in which the operations are performed based on FIFO principle and the last position is connected back to the first position to make a circle.



Implementation of Circular Queue

To implement a circular queue data structure using an array, we first perform the following steps before we implement actual operations.

- **Step 1 -** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2 -** Declare all **user defined functions** used in circular queue implementation.
- **Step 3 -** Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)
- **Step 4 -** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
- **Step 5 -** Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

- **Step 1 -** Check whether **queue** is **FULL**. (**(rear == SIZE-1 && front == 0) || (front == rear+1)**)
- **Step 2 -** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.
- **Step 4 -** Increment **rear** value by one (**rear++**), set **queue[rear]** = **value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

deQueue() - Deleting a value from the Circular Queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The deQueue() function doesn't take any value as a parameter. We can use the following steps to delete an element from the circular queue...

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. Then check whether both **front - 1** and **rear** are equal (**front -1** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

display() - Displays the elements of a Circular Queue

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

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
- **Step 4 -** Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.
- **Step 5 -** If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until'**i <= SIZE - 1**' becomes **FALSE**.
- **Step 6 -** Set **i** to **0**.
- **Step 7 -** Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

2.4 Priority Queue:

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

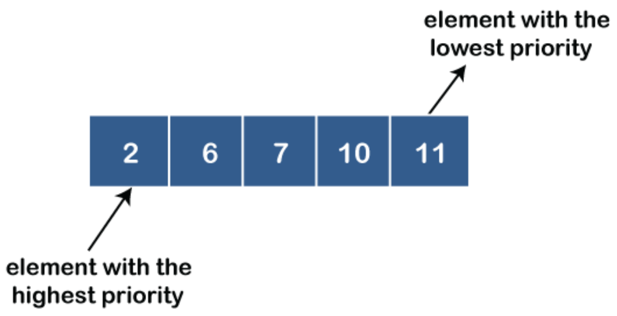
All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

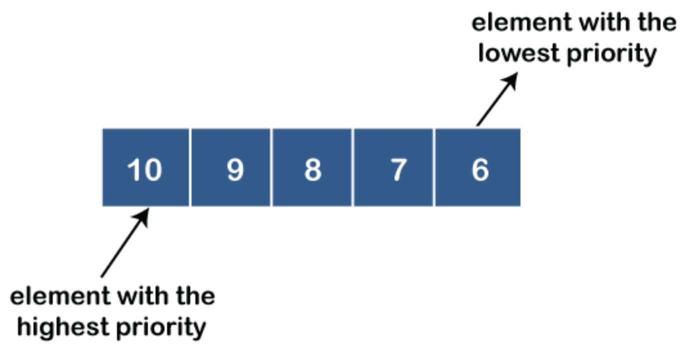
Types of Priority Queue:

There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.

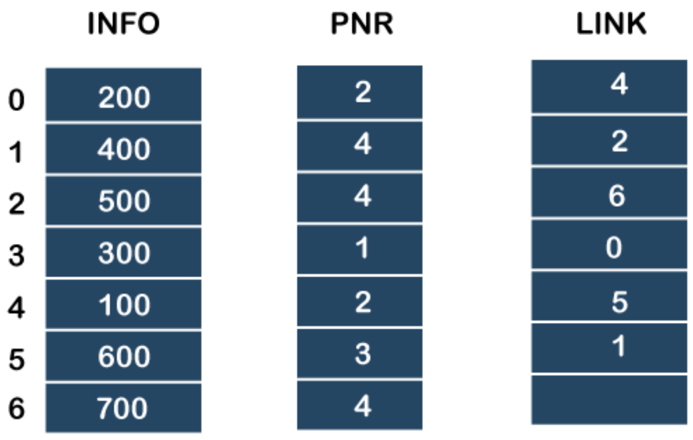

element with the lowest priority
2
6
7
10
11
element with the highest priority

- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.


element with the lowest priority
10
9
8
7
6
element with the highest priority

Representation of priority queue

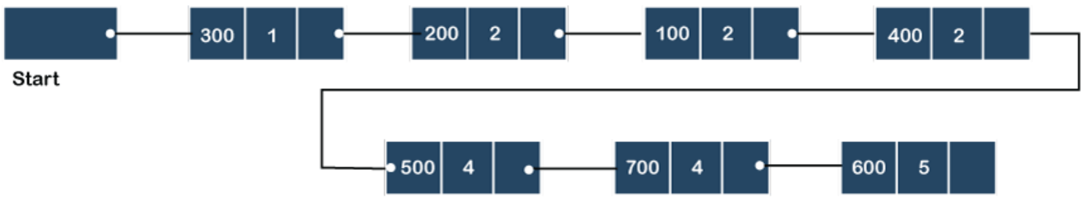
Now, we will see how to represent the priority queue through a one-way list.
We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and LINK basically contains the address of the next node.



	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

Let's create the priority queue step by step.
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.
Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:
Step 2: After inserting 300, priority number 2 is having a higher priority, and data values associated with this priority are 200 and 100. So, this data will be inserted based on the FIFO principle; therefore 200 will be added first and then 100.
Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 400, 500, 700. In this case, elements would be inserted based on the FIFO principle; therefore, 400 will be added first, then 500, and then 700.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 600, so it will be inserted at the end of the queue.



Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

Analysis of complexities using different implementations

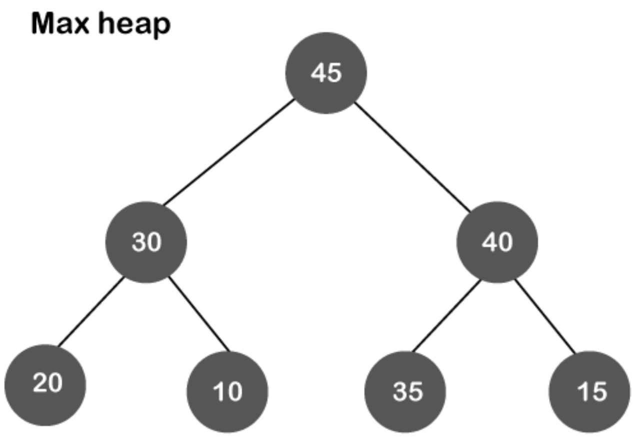
Implementation	add	Remove	peek
Linked list	O(1)	O(n)	O(n)
Binary heap	O(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

What is Heap?

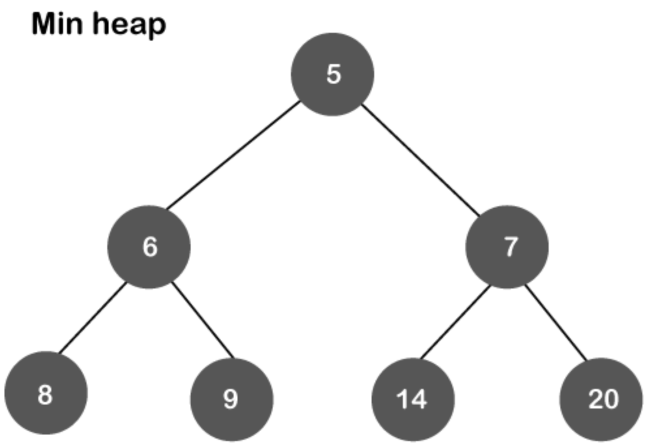
A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap.

It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

- **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.



- **Min heap:** The min heap is a heap in which the value of the parent node is less than the value of the child nodes.



Both the heaps are the binary heap, as each has exactly two child nodes.

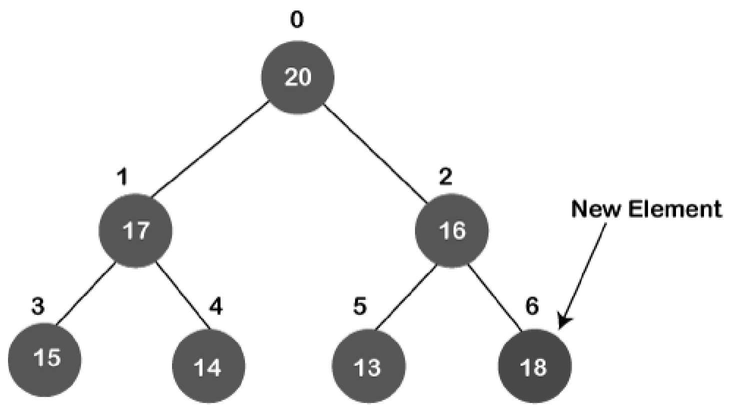
Priority Queue Operations

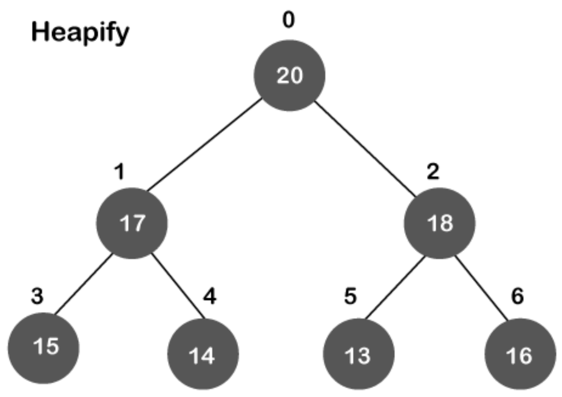
The common operations that we can perform on a priority queue are insertion, deletion and peek. Let's see how we can maintain the heap data structure.

Inserting the element in a priority queue (max heap)

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.





Removing the minimum element from the priority queue
As we know that in a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Applications of Priority queue

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.