PES University
Department of Computer Science and Engineering

UE19CS202- Data Structures and its Applications(4-0-0-4-4)

UNIT - 2 : STACKS AND QUEUES

Note : Pointers, Arrays, Structures, Linked List to be revised for clear understanding.

Course Content for Unit 2

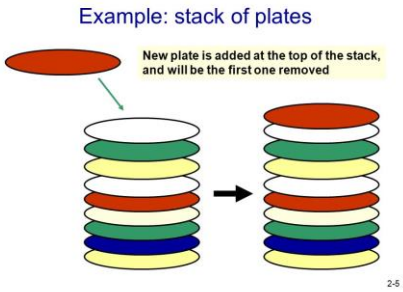
Stacks: Basic structure of a Stack, Implementation of a stack using Arrays & Linked list. **Applications of Stack:** Function execution, Nested functions, Recursion : Tower of Hanoi. Conversion & Evaluation of an expression: Infix to postfix, Infix to prefix, Evaluation of an Expression, Matching of Parentheses. **Queues & Deque:** Basic Structure of a Simple Queue, Circular Queue, Priority Queue, Deque and its implementation using Arrays and Linked List. **Applications of Queue: Case Study – Josephus problem, CPU scheduling- Implementation using queue(simple /circular).**

Contents :

6. Queues
7. Basic Structure of Queue
8. Implementation of queue using Arrays and Linked List
9. Circular queue
10. Priority Queue
11. Application of Queues
12. Case Study
13. References
14. Sample Code.

1. Introduction to Stacks:

- Consider a restaurant serving buffet lunch / dinner where plates are placed one above another. Every new plate is placed on the top.When a plate is required by a customer, it is taken off from the top and used. This is called stacking of plates.
- Consider another example of a driveway in a parking lot in a small apartment complex where all the cars will be parked as they enter and the new car will be parked at the last. To take out the car from the parking area, the last car which was parked should be taken out.
- In the above 2 scenarios, it is observed that the last plate added on top or the last car parked at the parking lot will be taken out first.
- Using this analogy, stack is defined as a linear data structure where items are inserted from one end called the **top** of the stack and removed from the same end.
- Last item inserted will always be on the top of the stack. Deletion is also done from the same end.
- Last item inserted first is the first item removed. Hence Stack is called **Last In First Out (LIFO) Data Structure.**



2. STACK - AS ADT

2 major operations that can be performed on stack.

- Insert a element into the stack - void push ()
- Delete an element from stack -int pop ().

2. 1 INSERTION OPERATION ON A STACK

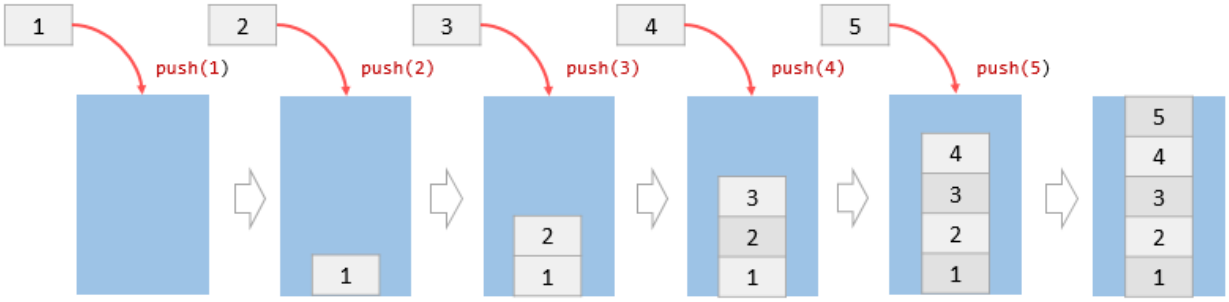


Figure 1 : Stack insertion operation

- Inserting an element into stack is called as PUSH Operation
- From the above figure, it is seen that the stack is fixed with size 5 and initially the stack is empty.
- Five elements : 1,2,3,4,5 are to be placed on the stack.
- Elements are inserted as shown above.
- Initially since the stack is empty, **top**(variable) points to the bottom of the stack.
- The first element 1 is pushed to the stack. At the same time top is incremented by 1.
- The second element 2 is pushed onto the stack, the top is incremented.
- Each time the elements are inserted, top is incremented.
- Similarly elements 3,4,5 are inserted one after the other on the stack.
- After inserting 5, the stack is full and there is no space left to insert.
- The situation in which stack is full and not possible to insert any new item is called **Stack Overflow or Stack FULL.**

2.2 DELETION OPERATION ON A STACK - POP

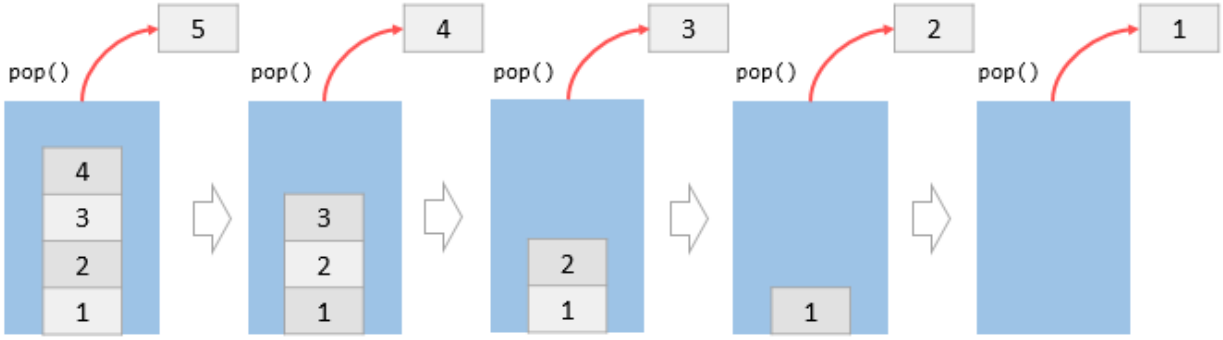


Figure 3 : Deletion an element from stack.

- Deletion of element from the stack is called as POP OPeration
- Deletion of elements from stack should be done from the same end as insertion.
- Elements are deleted from the top of the stack.
- Once the top most element is deleted from stack, the element below it becomes the new top element.
- Each time an element is decremented from the stack, the top is decremented.
- From the above diagram, it can be seen that the first element popped from the stack is 5. Also in parallel top is also decremented to the next highest position.
- After the elements 4,3,2,1 are removed, the top is decremented to the bottom of the stack and the stack is empty.
- Situation when the stack is empty and not possible to delete anymore elements is called **Stack Empty or Stack Underflow.**

3 IMPLEMENTATION OF STACK

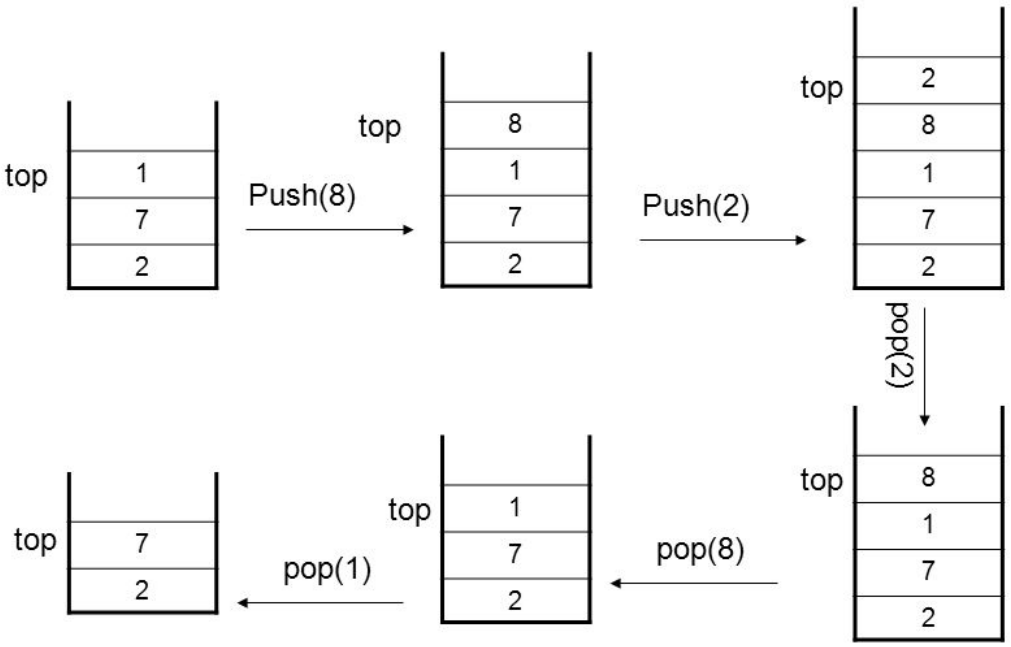
Stack Data structure can be implemented in two ways:

1. Using Arrays.
2. Using Linked List.

3.1 Implementation using Arrays:

- Stack is implemented using 1D Array.
- When a stack is implemented using arrays, its size remains fixed. It is not possible to increase or decrease the size.
- Define a 1D array to insert / delete using LIFO principle, initialize a variable top 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.

An Example of Stack



Following steps are to be followed for creating empty stack:

1. Create a 1- D Array with fixed size
2. Initialize a variable top to -1.

Insert an element into stack - push()

- 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 a parameter and inserts that value into the stack.

  Step 1 - Check whether stack is FULL. (top == SIZE-1)

  Step 2 - If it is FULL, then display "Stack overflow" 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)

Delete an element from stack - pop()

- pop() is a function used to delete an element from the stack.

- The element is always deleted from the top position.
- Pop function does not take any value as a parameter.

Step 1 - Check whether stack is EMPTY. (top == -1)

Step 2 - If it is EMPTY, then display "Stack Underflow!" and terminate the function.

Step 3 - If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--)

Display the contents of Stack- display()

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 it with top. Display stack[i] value and decrement i value by one (i--).

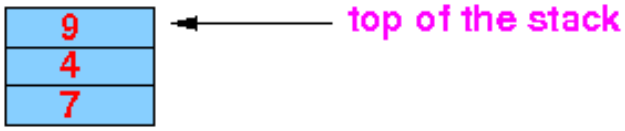
Step 4 - Repeat above step until i value becomes '0'.

3.2 Implementation of Stack Using Linked List

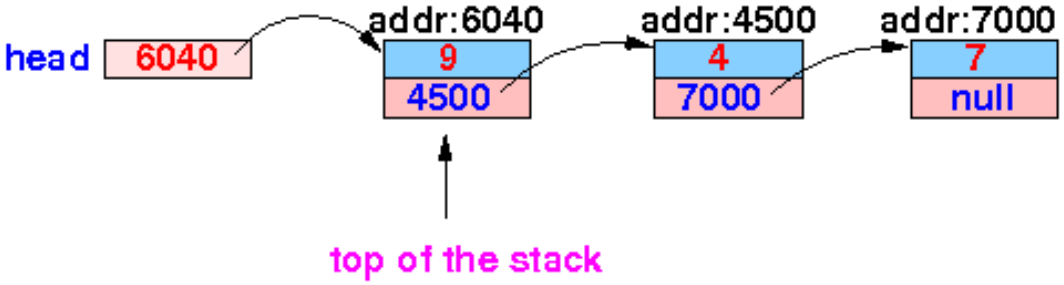
- Drawback of stack using arrays is that it works only with a fixed number of elements.i.e the size should be defined in the beginning itself.
- THis way of implementation is not suitable when the size of the elements are unknown
- To overcome the above drawback, stack can be implemented using the linked list
- Using a linked list based implementation, the size of stack is unlimited, so the data size can be variable.
- There is no need to fix the size initially.
- Here, every new element(Node) is inserted as the **top** in the list. Top is incremented as the node keeps increasing. Similarly as we delete the node, top is decremented.

Representing a stack with a list:

Stack:



List:



Stack Operations using Linked List:

Step 1 - Define a 'STACK' structure with two members - data of type integer and next as type Node.

Step 2 - Define a Node pointer 'top' and set it to NULL or -1.

Inserting an element into a Stack

Step 1- Create an empty node s of type STACK and assign any value.

Step 2 - Check if the stack is empty

Step 3 - If it is Empty, then set s→ next = NULL.

Step 4 - If it is Not Empty, then set p->top++;

p->s[p->top]=x;

Step 5 - Finally, set top = s

Deleting an Element from Stack - POP()

Step 1 - Check if the stack is empty (top == NULL).

Step 2 - If it is Empty, then display "Stack underflow”, terminate the functio.

Step 3 - If it is Not Empty, then define a Node pointer 'q' and set it to 'top'.

Step 4 - Then set 'top = top → next'.

Step 5 - Finally, delete q. (free(temp)) / or return q (return type is int)

Display the contents of the Stack

Step 1 - Check whether stack is Empty (top == NULL).

Step 2 - If it is Empty, then display 'Underflow' and terminate the function.

Step 3 - If it is Not Empty, then define a STACK pointer 'stk' and initialize it with top. Traverse the list by starting from top till u reach the end of list

Step 4 - Display 'stk → data --->' and move it to the next node. Repeat the same until stk reaches to the first node in the stack. (stk → next != NULL).

Step 5 - Finally display the list is empty.

4. APPLICATIONS OF STACK

There are various applications of stack:

- Conversion of Expression.
- Evaluation of Expression.
- Recursion.
- Parentheses Checking.
- Syntax Parsing.
- Backtracking - gaming, finding path.

4.1 CONVERSION OF AN EXPRESSION

- **Expression** : An expression is a mathematical statements consisting of the operators between the operands
- It can be syntactically defined as follows:
    - **A op B**.
- Here A and B are called operands and op is the operator.
- Operators are mainly used to perform mathematical operations between the operands.
- Operators can be any mathematical operators like - Addition, Subtraction, Multiplication and Division.

Types of Expressions:

1. Infix Expressions
2. Prefix Expressions
3. Postfix Expressions.

Infix Expressions: Expression having an operator between two operands

1. a + b * c
2. (2+2)*(4/1)

Prefix expressions : Expressions where the operators precede the operands

1. +a*bc
2. *+22/14

Postfix Expressions: Expressions where the operators succeed the operands

1. bc*a+
2. 41/22+/

4.1 CONVERSION OF INFIX TO POSTFIX EXPRESSION

1. In the given infix expression fill all the operators
2. As per the operator precedence, operators are evaluated accordingly.
3. Convert into the respective postfix form in the same order of evaluation.

Steps in converting infix expression to postfix

1. Scan all the input symbols(operands & Operators) from L-R from the infix expression.
2. If the input symbol is operand, then output and print the operand directly.
3. If the input symbol is left parenthesis '(',push it onto the Stack.
4. If the input symbol is right parenthesis ')', then Pop all the contents of stack until matching left parenthesis is popped and print each popped symbol to the result.
5. If the input symbol is operator i.e + , - , * , / , then Push it onto the Stack.
6. If first pop the operators which are already on the stack that have higher or equal precedence than current operators and print them to the result.
7. Resultant expression will be in the postfix form - AB op where A and B are operands and op is the operator.

Symbol	Current Operator Precedence	Stack Precedence
+, -	1	2
*, /	3	4
Operands	7	8
(	9	0
)	0	-
Top of Stack Initially (#)	-	-1

4.2 EVALUATION OF POSTFIX EXPRESSIONS

- In the previous section, we discussed the conversion of the infix expression to its postfix form.
- In a postfix expression, operator suffixes the operands.
- General Structure of a postfix expression is
    - A B op
- Here, A and B are operands and op is the operator.

Evaluation of postfix Expression using Stack

Following steps are followed while evaluating the postfix expression.

1. Create an empty stack - to store operands.
2. Scan the symbols in the given postfix expression.
3. If the input is operand, then push it onto the Stack.
4. If the input is operator (+ , - , * , / etc.,), then the top two operands are popped and store them as 2 different variables.
5. Then perform scanning the symbol operation using operand1 and operand2 and push the result back on to the Stack.
6. Lastly pop and display the popped value as the final result.

Evaluation of Postfix Expression

Postfix → a b c * + d - Let, a = 4, b = 3, c = 2, d = 5

Postfix → 4 3 2 * + 5 -

Operator/Operand	Action	Stack
4	Push	4
3	Push	4, 3
2	Push	4, 3, 2
*	Pop (2, 3) and 3*2 = 6 then Push 6	4, 6
+	Pop (6, 4) and 4+6 = 10 then Push 10	10
5	Push	10, 5
-	Pop (5, 10) and 10-5 = 5 then Push 5	5

www.geekyshows.com

4.3 PARENTHESIS MATCHING

- Parentheses match also known as balancing of the parentheses is a common coding syntax error that occurs during early programming.
- The following are examples of matching parentheses - ({}) (())({}() ()
- The following are examples of non matching parentheses - {} ({}( ){}) (()
- Following are the steps to the followed while matching the parentheses

Step - 1 -Scan the input string.

Step -2 If character is opening parenthesis `'(', '{','['`, push it on stack.

Step - 3 If character is closing parenthesis `')','}','].`

Step 3.1 Check top of stack, if it is `'(','{','['` , pop and move to next character.If it is not `'('`, return false.

Step - 4 After scanning the entire string, check if the stack is empty. If stack is empty, return true else return false.

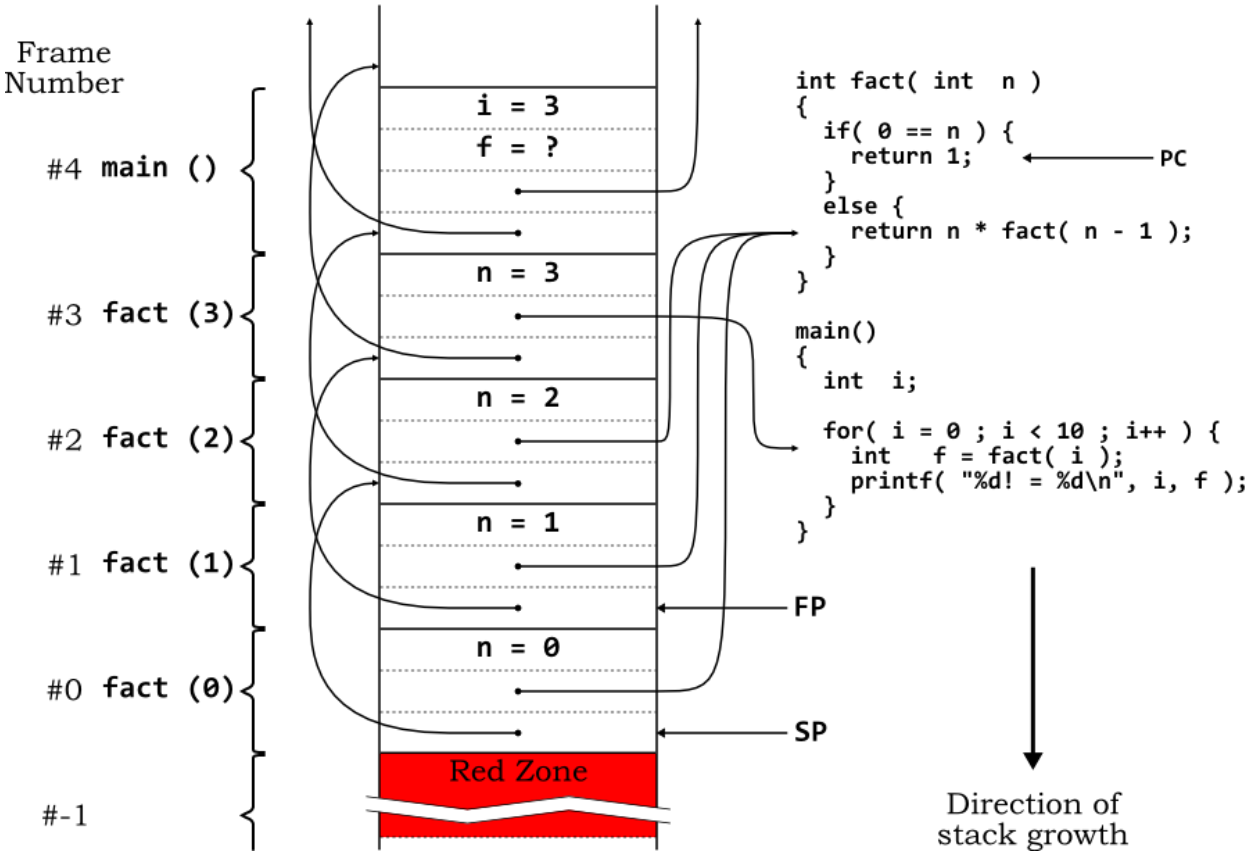
5. RECURSION

- Powerful utility available in most of the programming languages- C, C++, python, Java etc..
- Recursion is a method of solving problems where the solution depends on smaller instances of the same problem.
- In recursion, a function ‘a’ either calls itself directly or calls a function ‘b’ that in turn calls the original function ‘a’.
- A recursive function using has 2 cases:
    - A base case
    - A Recursive case
- A base case in recursion, in which the answer is known when the termination for a recursive condition is to unwind back.
- A recursive case which returns to the answer which is closer

RECURSION Vs ITERATION
Diff between Recursion & Iteration:

Recursion in Data Structures - Implementation of Recursion

- Implementation recursion by means of Stacks Data Structure.
- Whenever a function (caller) calls another function (callee) or itself as callee, the caller function transfers execution control to the callee.
- This transfer process may also involve some data to be passed from the caller to the callee.
- This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function.
- The caller function needs to start exactly from the point of execution where it puts itself on hold.
- It also needs the exact same data values it was working on.
- For this purpose, an activation record (or stack frame) is created for the caller function.



The above diagram shows the stack implementation of a factorial program executed recursively.

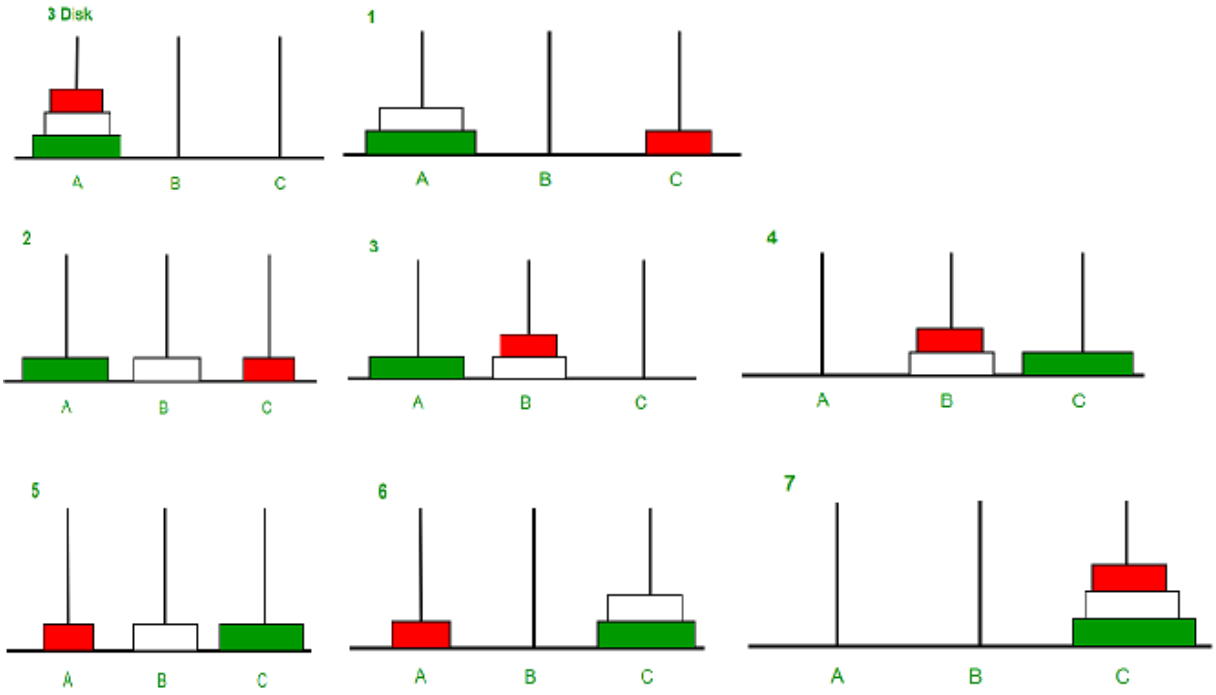
Example : Tower of Hanoi

It is a classical mathematical puzzle where there are three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1) Only one disk can be moved at a time.

2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

3) No disk may be placed on top of a smaller disk



In the above diagram there are 3 rods named A, B, C consisting of 3 disks(Orange 'O' at Top, White 'W' at middle and Green 'G' at Bottom) in Rod A.
Step 1 : Move 'R' to C.
Step 2 : Move 'W' to B.
Step 3 : Move 'R' to B.
Step 4 : Move 'G' to C.
Step 5 : Move 'R' to A.
Step 6 : Move 'W' to C.
Step 7 : Move 'R' to C.

Recursive ALgorithm : disk - # of disks, src - source rod, aux - auxiliary rod, destination- destination rod.

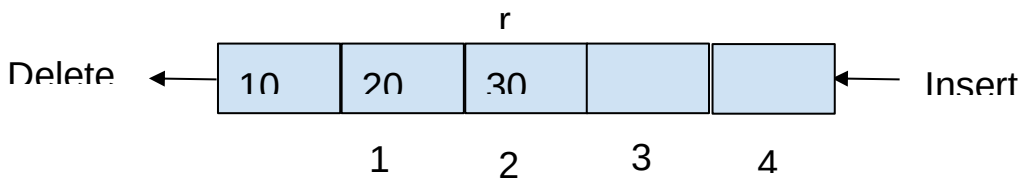
Start
TOH(disk,src,aux).
    If disk == 1 , move from src to destination.
Else
    TOH(disk-1,src,aux,destination). (**Step 1**)
    Move from src to destination. (**Step 2**)
    TOH(disk-1,aux,destination, src). **(Step 3**)

6. QUEUES

In the previous section we discussed about one of the common linear data structures and its applications - Stacks

- In this topic we will be discussing another linear Data Structures and its application and implementation - **Queues**
- Queue is a very familiar term used in day to day life as we see people standing in queues to pay bills, to board buses, to buy tickets in cinema halls etc..
- In these scenarios, a person just enters the queue and stands at the end and the person who is at the front of the first person to pay the bill or get inside the bus or purchase the tickets etc..

- The queue data structure is implemented in the same concept
- A **Queue** is a linear data structure where the elements are inserted from one end and deleted from the other end.
- End from which an element is inserted is called the **rear** end and the end from which the element is deleted is called the **front** end.



Pictorial representation of Queues is as shown above :

- The first element inserted into the queue is 10, the second element inserted in 20 and the last element inserted is 30.
- Any new element to be inserted into the queue has to be inserted towards the right of 30and that element will be the last element in the queue.
- From the above operations on the queue, it can be clearly understood that the first element inserted into the queue is the first element to be deleted out from the queue.
- Hence Queue is known as the **First in First Out (FIFO) Data Structure.**

Applications of Queue in Computer Science:

- CPU Scheduling.
- Time sharing
- Batch Processing

Types of Queue:

1. Ordinary Queue
2. Circular Queue
3. Double Ended Queue
4. Priority Queue

QUEUE - ADT

Operations on Queue

1. Insert into an element from queue - En-queue
2. Delete an element from the queue - Dequeue

Implementation

A queue can be implemented in Two ways :

1. Using Arrays
2. Using LInked List

IMPLEMENTATION OF QUEUE USING ARRAYS:

- Queue can be implemented using 1D Array.
- The queue implemented using an array stores only a fixed number of data values.
- Implementation s very simple -
- Define a one dimensional array of specific size and insert or delete the values into that array by using FIFO mechanism 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.

INSERTING AN ELEMENT - ENQUEUE

EnQueue() is a function used to insert a new element into the queue.

The new element is always inserted at the rear position.

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

DELETING THE ELEMENT FROM QUEUE

DeQueue() function used to delete an element from the queue.

The element is always deleted from the front position.

The deQueue() function does not take any value as parameter.

- **Step 1** - Check whether the queue is **EMPTY**. (**front == rear**)
- **Step 2** - If it is **EMPTY**, then display **"Queue Underflow "** and terminate the function.
- **Step 3** - If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as a 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**).

Queue Using Linked List

- Drawback of queues using arrays is that it will work for an only fixed number of data values.
- That is the amount of data or the size of the queue must be specified at the beginning itself.
- Queue using an array is not suitable when the size of the data is too large or unknown.
- To overcome the above drawback, queue can be implemented using a linked list data structure.
- A Linked list can work for an unlimited number of values- no need to fix the size at the beginning of implementation.
- The Queue implemented using linked lists can organize as many data values as we want.

Operations:

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

Step 1 - 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** function by displaying a menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

ENQUEUE - INSERTION ELEMENT FROM FRONT

Step 1 - Create a node called **temp** with given value and set '**temp** → **next**' to **NULL**.

Step 2 - Check whether queue is **Empty** (**rear** == **NULL**)

Step 3 - If it is **Empty** then, set **front** = **rear=temp**

Step 4 - If it is **Not Empty** then, set **rear** → **next** = **temp** and **rear** = **temp.**.

DE-QUEUE - DELETION FROM FRONT END

Step 1 - Check whether the queue is Empty **(front == NULL).**

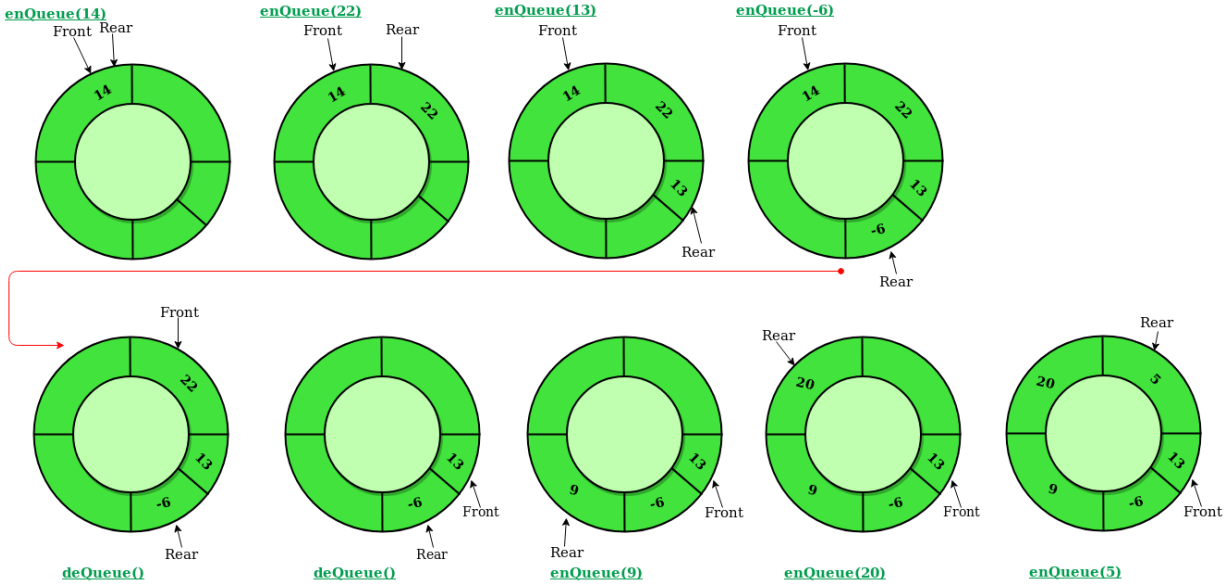
Step 2 - If it is Empty, then display "Queue underflow" and terminate from the function

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)).

CIRCULAR QUEUE

- In an ordinary queue, elements are inserted, the rear end identified by ‘r’ is incremented by 1.
- Once the queue reaches MAX-1 , the queue is full.
- Even If some elements are deleted from the queue, and rear = MAX-1, items cannot be inserted.
- To overcome the above drawback, a circular queue is used.
- A 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.



IMPLEMENTATION OF CIRCULAR QUEUE

Step 1 - Declare all functions used in circular queue implementation.(insert, delete,display)

Step 2 - Create a one dimensional array with a predefined size.

Step 3 - Define two integer variables 'front' and 'rear' and initialize both with '-1'.

Step 4 - Implement the main method by displaying a menu of operations list and make suitable function calls to perform operations selected by the user on a circular queue.

Enqueue - Inserting value into circular Queue

- EnQueue() is a function used to insert an element into the circular queue.
- In a circular queue, the new element is always inserted from its rear position.

Step 1 - Check whether the queue is FULL. ((rear == SIZE-1 && front == 0) || (front == rear+1))

Step 2 - If it is FULL, then display "Queue overflow" 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 value into circular Queue

- DeQueue() is used to delete an element from the circular queue.
- In a circular queue, the element is always deleted from the front position.

Step 1 - Check whether the queue is **EMPTY**. (**front == -1 && rear == -1**)

Step 2 - If it is **EMPTY**, then display **"Queue underflow"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then display **queue[front]** as a 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**).

DOUBLE ENDED QUEUE

Double ended queue is a special type of data structure in which insertions and deletions will be done either at front end or at rear end of the queue.

Operations of Double ended queue

1. Insert an element from the front end.
2. Insert an element from the rear end.
3. Delete an element from the front end.
4. Delete an element from the rear end.
5. Display the contents of the queue.

Implementation: Assignment for Students

PRIORITY QUEUE

- A priority queue is a special type of queue.
- In a priority queue elements are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa.
- So we're assigned priority to the element based on its key value.
- Lower the value, higher will be the priority.

APPLICATIONS OF PRIORITY QUEUE

1. Dijskstra's shortest path algorithm
2. Prims Algorithm
3. Artificial intelligence - A Search Algorithm.

IMPLEMENTATION:

Following are the steps used in the implementation of Priority queue using a list -

INSERTION

Step 1 : If there is no new node, create a node dynamically to allocate memory.

Step 2 : if the node is already present, insert the new node at the end from left to right.

Step 3 : Heapify the array.

DELETION

Step 1 : If the node to be deleted is leaf node, remove the node

Step 2 : Swap the node to be deleted with the last leaf node.

Step 3 : Heapify the array.

Priority Queue Applications

Some of the applications of a priority queue are:

1. Dijkstra's algorithm
2. for implementing stack
3. for load balancing and interrupt handling in an operating system
4. for data compression in Huffman code

SAMPLE CODE :

Implementation of Stacks to perform push and pop operations:

```
#include<stdio.h>
#include<stdlib.h>
int push(int*,int*,int,int);
int pop(int*,int*);
void display(int*,int);
//driver function.
int main()
{
  int top,size,ch,k,x;
  int *s;
  printf("Enter the size of the stack..\n");
  scanf("%d",&size);
  s=(int*)malloc(sizeof(int)*size);
  top=-1;
  while(1)
  {
    display(s,top);
    printf("\n1..push\n");
    printf("2..pop\n");
    printf("3..display\n");
    scanf("%d",&ch);
    switch(ch)
    {
      case 1:printf("Enter the data\n");
```

```
            scanf("%d",&x);
            k=push(s,&top,size,x);
            if(k>0)
              printf("Element pushed successfully\n");
            break;
    case 2:k=pop(s,&top);
           if(k>0)
              printf("\nElement popped=%d\n",k);
           break;
     case 3:display(s,top);
           break;
     case 4:exit(0);
   }
 }
}
//pop
 int pop(int *s, int *t)
 {
   int x;
   //check for stack underflow
   if(*t==-1)
   {
     printf("Stack underflow..cannot delete..\n");
     return -1;
   }
    x=s[*t];
    (*t)--;
```

```
    return x;
  }
//push
 int push(int *s, int *t, int size, int x)
 {
    //check for overflow

    if(*t==size-1)
    {
      printf("Stack overflow..cannot insert\n");
      return -1;
     }
     (*t)++;//or ++*t or *t=*t+1
      s[*t]=x;
     return 1;
   }
//display the contents of stack
  void display(int *s, int t)
  {
    int i;
    if(t==-1)
      printf("Empty stack..\n");
    else
     {
       for(i=t;i>=0;i--)
          printf("%d  ",s[i]);
      }
```

```
}
```

Implementation of QUEUE to perform enqueue and dequeue operations

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

int qinsert(int,int*,int*,int*,int);
 int qdelete(int *,int *, int*);
 void display(int *,int,int);
int main()
{

  int *q;
  int ch,k,x;
   int f,r, size;
  f=r=-1;
  printf("Enter the size of the size..");
  scanf("%d",&size);

  q=(int*)malloc(sizeof(int)*size);
  while(1)
  {
    display(q,f,r);
    printf("\n1..Insert");
    printf("\n2..Delete");
    printf("\n3..Display");
    printf("\n4..EXIT");
    scanf("%d",&ch);
    switch(ch)
```

```
  {
     case 1:printf("Enter the value..");
            scanf("%d",&x);
            k=qinsert(x,q,&f,&r,size);
            if(k>=0)
               printf("Element inserted successfully\n");
             break;
     case 2:k=qdelete(q,&f,&r);
            if(k>=0)
              printf("element deleted=%d\n",k);
            break;
      case 4:exit(0);
  }
 }
}

 int qinsert(int x,int *q,int *f,int *r,int size)
 {
  //check for queue overflow

   if(*r==size-1)
   {
     printf("Queue full..\n");
     return -1;
   }
    (*r)++;
    q[*r]=x;
```

```
   if(*f==-1)//first element
      *f=0;//point to first element
   return 1;
 }


int qdelete(int *q,int *f, int *r)
{
   int x;
   //check for queue empty
  if(*f==-1)
  {
     printf("Queue empty..\n");
      return -1;
   }
   x=q[*f];
   if(*f==*r)//only one element
     *f=*r=-1;
   else
     (*f)++;
   return x;
}


 void display(int *q, int f, int r)
 {
    int i;
```

```
    if(f==-1)
      printf("Empty Queue..\n");
    else
     {
       for(i=f;i<=r;i++)
          printf("%d  ",q[i]);
     }
    printf("\n");
}
```