MODULE II-STACKS, QUEUES, RECURSION

Dr. Ganga Holi, Professor & Head, ISE Dept.

Global Academy of Technology
Dept. of Information Science & Engineering

[COMPANY NAME]
[Company address]

Global Academy of Technology

Dept. of Information Science & Engineering

Data Structure Notes

On

Module II

Stacks, Queues & Recursion

Dr. Ganga Holi,

Professor & Head

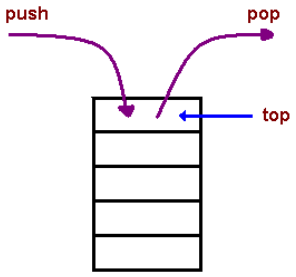
Contents

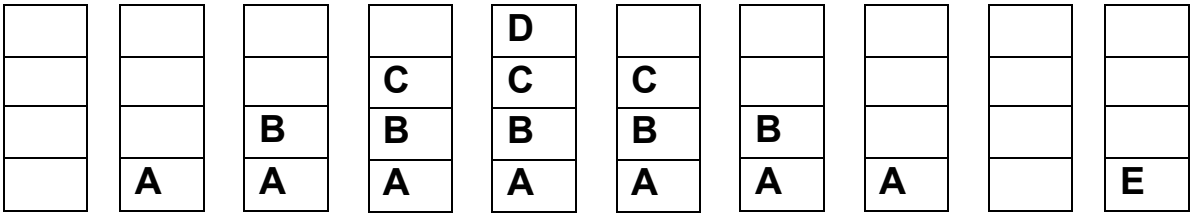
Stacks

Definition: A stack is linear Data Sturcture in which items are added (pushed, inserted) and removed(pop, deleted) from one end called top.

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. Push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

Given a stack S=(a_0,a_1, ….a_{n-1}), we say a_0 is the bottom element, a_{n-1} is the top element and a_i , is on the top of element a_{i-1} , 0<i<n.





Top=-1	Top=0	Top=1	Top=2	Top=3	Top=2	Top=1	Top=0	Top=-1	Top=0
	Push	Push	Push	Push	Pop	Pop	Pop	Pop	Push

Figure 1. Stack operations.

Stack can be represented in the following ways Static implementation using array representation.

Array representation of stacks

Stacks may be represented using one way list ot linear representation that is 1D array. Stack will be maintained by 1 D array of elements. Elements can be of any data types: Basic, derived, user defined data types.

Index variable TOP or pointer TOP is used to represent the location of the top of the stack element.

Stack Operations

1. Stack full
2. Stack empty
3. Push
4. Pop
5. Display
6. Top of stack

Algorithms for stack operations

```
 int stackFull()
{       if(top==(max_size-1))
                Return 1;
        else
                Return 0;
}

int stackEmpty()
{       if(top==-1))
                return 1;
        else
                return 0;
}

void push(int  ele)     //Inserting element into the stack
{       if(stackFull())
        {
                printf("\nStack Overflow:");  return;
        }
        stack[++top]=item;
}

int pop() //deleting an element from the stack
{       if(stackFull())
        {       printf("\nStack Overflow:");  return -1;
        }
```

```
        return stack[top--];
}

void display()
{       int i;
        if(stackEmpty())
        {       printf("\nStack is Empty:"); return;          }
        printf("\nThe stack elements are:\n" );
        for(i=top;i>=0;i--)
                printf("%d\n",stack[i]);
}

int topStack()
{       return(stack[top];
}
```

// Complete program for stack operations

```
#include<stdlib.h>
#define max_size 5
int stack[max_size],top=-1;
void push();
void pop();
void display();
int main()
{       int choice;
        while(1)
        {       printf("\n\n--------STACK OPERATIONS----------\n");
                printf("1.Push\n");
                printf("2.Pop\n");
                printf("3.Display\n");
                printf("4.Exit\n");
                printf("\nEnter your choice:\t");
                scanf("%d",&choice);
                switch(choice)
                {       case 1 : printf("Enter the element to be inserted:\t");
                                 scanf("%d",&item);
                                 push(ele); break;
                        case 2 : ele=pop();
                                 if(ele==-1)
                                         printf("stack Underflow\n”);
                                 else
                                         printf("Poped element =%d\n”, ele);
                                 break;
                        case 3 : display(); break;
                        case 4 : return 0;
                        default : printf("\nInvalid choice:\n") ;   break;

                }
```

```
    } return 0;

}
```

Applications of Stack

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.
- Evaluation of Expressions
- Conversion from Infix to Postfix, prefix
- Language processing:
    - space for parameters and local variables is created internally using a stack.
    - compiler's syntax check for matching braces is implemented by using stack.
    - support for recursion

Infix expression- The expression in which operator is placed in between operands
Ex. a+b
Polish Notation- names after the Polish Mathematician Jan Lukasiewiez, refers to the expression in which the operator is placed before the operands also called prefix expression.
Ex. +ab
Reverse Polish Notation- refers to the expression in which the operator is placed after the operands also called postfix expression or suffix Expression.
Ex. ab+

Evaluation of Postfix expression

In high level languages, infix notation cannot be used to evaluate expressions. Instead compilers typically use a parenthesis free notation to evaluate the expression. A common technique is to convert a infix notation into postfix notation, then evaluating it.

Before knowing the function that translates infix expression to postfix expression, we need to know how to evaluate the postfix expression.

45+ will be evaluate as 4+5=9

To evaluate an expression we make single left to right scan (read) of it. We place the operands on a stack until we find an operator. We then remove, from the stack, correct number of operands for the operator, perform operation, and place the result back on the stack. We continue until we reach the end of the expression. We remove the result top of the stack.

Example3.

Postfix		Stack
234*+		
34*+		**2**
4*+		**2 3**
***+**		**2 3 4 (3*4=12)**
+		**2 12 (2+12=24)**
		24

Algorithm/ Function to evaluate Postfix expression

```
int op(int op1,char sym,int op2)  //op=evaluate
    {   switch(sym)
        {   case '+': return op1+op2;
            case '-': return op1-op2;
            case '*': return op1*op2;
            case '/': return op1/op2;
            case '%': return op1%op2;
            case '^':
            case '$': return pow(op1,op2);
        }
        return 0;
    }

void evaluatePostfix()
{   for(i=0;i<strlen(p);i++)
    {
        sym=p[i];
        if(sym>='0' && sym<='9')
            s[++top]=sym-'0';
        else
        {
            op2=s[top--];
            op1=s[top--];
            s[++top]=op(op1,sym,op2);
        }
    }
    printf("\nThe result is %d",s[top]);
}
```

Conversion to Infix expression to Postfix expression

Infix to Postfix Notation. Normally you will code your expressions in the programmes using infix notation, but compiler understands and uses only postfix notation and hence it has to convert infix notation to post fix notation. For this activity, stack data structure is used. We will consider following binary operators and their precedence rules:

^ Exponentiation

*** / Multiplication and Division. Both have same priority-- Execution is from left to right.**
+ - Addition and Subtraction. Both have same priority--- Execution is from left to right.

Other Operators : order is from left to right
Example D+E-G means (D+E)-G
Let us solve a problem
Infix notation : A +B * C – D
Based on priorities of operators, parenthesize the expression. You know, the priority for the expression given is (* or /) and followed by (+ or -).

((A + (B * C)) – D)

Check your brackets are correct and opening and closing brackets match.
Opening brackets = closing brackets = 3
Step 3. Number your brackets starting from Right Hand side. Give the same number to governing bracket on left hand side.

((A + (B * C)) – D)
1 2 3 3 2 1

Post Fix Notation : A B C * + D -

Example: Infix Expression -- A + B * C - D / E

S.No	Infix	Stack(top)	Postfix
1.	A + B * C -D / E	#	
2.	+ B * C -D / E	#	A
3.	B * C -D / E	# +	A
4.	*C -D / E	#+	AB
5.	C -D / E	#+*	AB
6.	-D / E	#+*	ABC
7.	-D / E	#+	A B C *
8.	D / E	#+-	A B C *
9.	/ E	#+-	A B C * D
10.	E	#+-/	A B C * D
11.		#+-/	A B C * D E
12.		#+-	A B C * D E /
13.		#+	A B C * D E /-
14.		#	A B C * D E/- +

Example 2:
A * B - (C + D) + E

S.No	Infix	Stack(bot->top)	Postfix
1.	A * B - (C - D) + E	empty	empty
2.	* B - (C + D) + E	empty	A
3.	B - (C + D) + E	*	A
4.	- (C + D) + E	*	A B
5.	- (C + D) + E	empty	A B *
6.	(C + D) + E	-	A B *

7.	C + D) + E	- (	A B *
8.	+ D) + E	- (	A B * C
9.	D) + E	- (+	A B * C
10.	) + E	- (+	A B * C D
11.	+ E	-	A B * C D +
12.	+ E	empty	A B * C D + -
13.	E	+	A B * C D + -
14.		+	A B * C D + -E
15.		Empty	A B * C D + -E+

Algorithm or function for conversion infix to postfix expression

```
void postfix(void)
{        // assuming  variables are declared as global
        char ch;   int i;
        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();     /* Remove ( */
                }
                else     /* Operator */
                {
                        while (pr(s[top]) >= pr(ch))
                        pofx[k++] = pop();
                        push(ch);
                }
        }
        while (s[top] != '#') /* Pop from stack till empty */
                pofx[k++] = pop();
        pofx[k] = '\0';          /* Make pofx as valid string */
        printf("Given Infix    Expn: %s      Postfix Expn: %s\n", infx, pofx);
}
```

// pr() is a function for finding the operator precedence value

```
 int pr(char elem) /* Function for precedence */
{       switch (elem)
        {       case   '#' :  return 0;
```

```
            case  '(' :  return 1;
            case  '+' :
            case  '-' :  return 2;
            case  '*' :
            case  '/' :
            case  '%' :  return 3;
            case  '^':   return 4;
        }
}
```

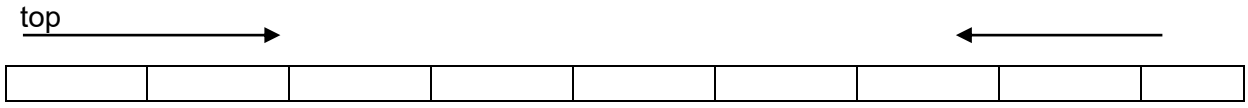
Exercise Problems from Sahni

1. **Write postfix and prefix expression of the following expressions**
   a. **A*b*c**
   b. **–a+b-c+d**
   c. **a*(-b)+c -→ ans: ab-*c+ here no. of operators equal to no. of operands. So there has to be one unary operator… So –b will be evaluated first, then * will be performed, then +.**
   d. **a&&b||c||!(e>f) (assuming C precedence Refer C text book)**

Multiple stacks

Multiple stacks can be implemented using multiple arrays, one array to represent on stack. Multiple stacks can be implemented using single array.

Single array can be used to represent two stacks: one will go from beginning from starting towards end and second stack will go in opposite direction.



S1[top]=-1 stack empty condition if(top==-1)

S2[bot]=N stack empty condition if(bot==N)

Stack full if (top==bot) for both

Stack 1 :Push -- S1[++top]=ele Pop -→ ele=S1[top--]

Stack 2 :Push -- S2[--bot]=ele Pop -→ ele=S2[top++]

Single array can be used to represent multiple stacks by dividing into equal number of parts.

i=0	I=N…….	I=2N……. I=3N-	I=3N……. …….

…….i<N	I=2N-1	1	I=4N-1

Recursion

A recursive function is a function which either calls itself or is in a potential cycle of function calls. As the definition specifies, there are two types of recursive functions. Consider a function which calls itself: we call this type of recursion immediate recursion and indirect recursion.

Ex: Factorial is immediate recursive function.

```
void A() {
  B();
  return;
}

void B() {
  C();
  return;
}

void C() {
  A();
  return;
}
```

A recursive function must have the following two properties.

1. There must be certain criteria, called base criteria, for which the function does not call itself. Also called terminating condition to terminate the recursive call.
2. Each time function does call itself with the reduced value or the value must be closer to the base criteria.

A Recursive function with these two properties is said to be **well defined.**

Example. Factorial function

1. If n=0 then n!=1 → base criteria
2. If n>0 the n!=n.(n-1)!

```
fact(n)

{   if(n<1) return 1;

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

}
```

Fibonacci Sequence

Fibonacci series 0,1,1,2,3,5,8,13,21,…..

1. if n=0 or n=1, then F_n =n
2. if n>1 then Fn=F_{n-1}+F_{n-2}

```
fib(n)
{
    if(n==0) return 0;
    if(n==1) return 1;
    else
    return( fib(n-1)+fib(n-2));
}
```

Ackermann–Péter function

The two-argument **Ackermann–Péter function**, is defined as follows for nonnegative integers *m* and *n*:

$$A(m,n) = \begin{cases} n+1 & if\, m = 0 \\ A(m-1,1) & if\ m > 0\ \ and\ n = 0 \\ A\big(m-1, A(m,n-1)\big) & if\ m > 0\ and\ n > 0 \end{cases}$$

A(0,1)=2

A(1,1)=A(m-1,A(m,n-1))=A(0,A(1,0))=A(0,A(0,1))=A(0,2)=3

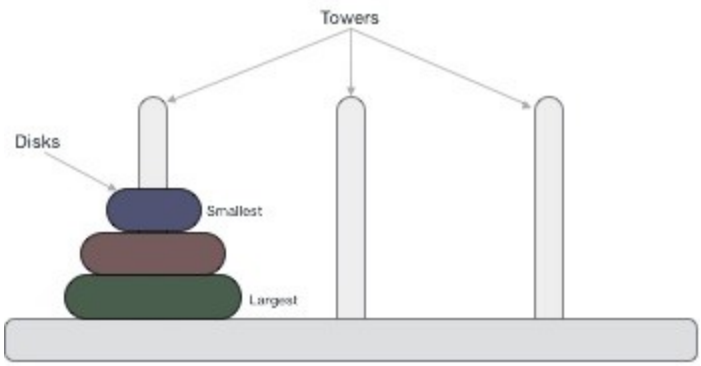
A(2,1)=A(1,A(2,0))=A(1,A(1,1))=A(1,3)=A(0,A(1,2))=A(0,A(0,A(1,1)))

=A(0,A(0,3))=A(0,4)=5

It grows very fast and non primitive recursive function.

Tower of Hanoi

Tower of Hanoi, is a mathematical puzzle which consists of three tower (pegs) and more than one rings; as depicted below −



These rings are of different sizes and stacked upon in ascending order i.e. the smaller one sits over the larger one. There are other variations of 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. The below mentioned are few rules which are to be followed for tower of hanoi −

- 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.

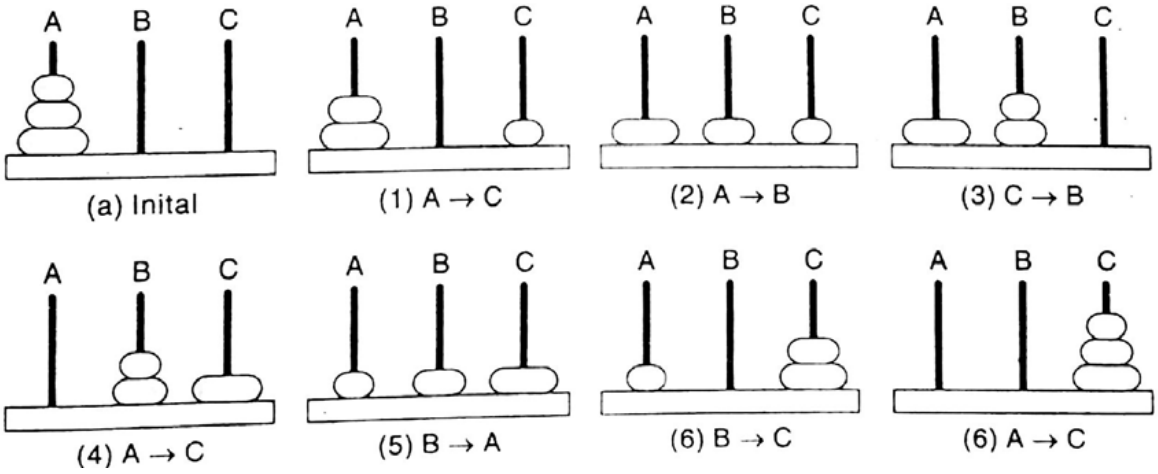
Tower of hanoi puzzle with **n** disks can be solved in minimum 2^n-1 steps. This presentation shows that a puzzle with 3 disks has taken $2^3-1 = 7$ steps.

Step 1 − Move n-1 disks from **source** to **temp**
Step 2 − Move n^{th} disk from **source** to **dest**
Step 3 − Move n-1 disks from **temp** to **dest**

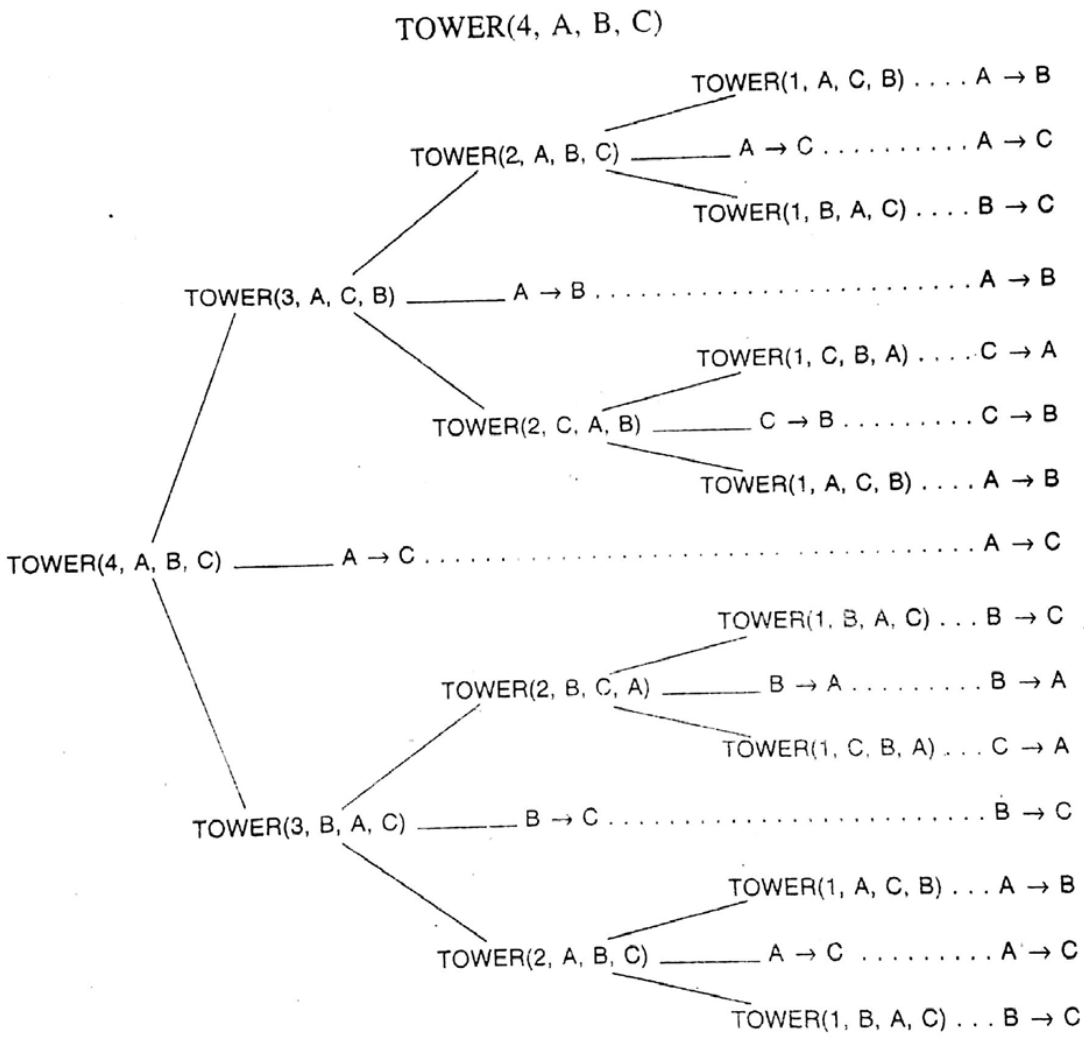
N=1
Step 1 − Move n^{th} disk from **source(A)** to **dest(C)**

N=2
Step 1 − Move n-1=1 disk from **source(A)** to **temp(B)**
Step 2 − Move n^{th} =2 disk from **source(A)** to **dest(C)**
Step 3 − Move n-1=1 disk from **temp(B)** to **dest(C)**

$n = 3$: Move top disk from peg A to peg C.
Move top disk from peg A to peg B.
Move top disk from peg C to peg B.
Move top disk from peg A to peg C.
Move top disk from peg B to peg A.
Move top disk from peg B to peg C.
Move top disk from peg A to peg C.



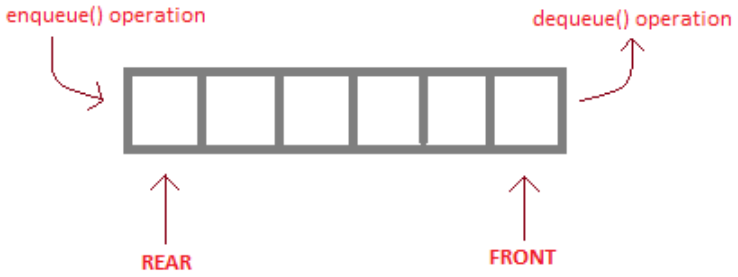

```
void  tower(n, source, temp, dest)
{
  if(n == 0)
      printf("move  %d disk from %c  to %c", source,dest);

  else
  {
    tower(n - 1, source, dest,temp)     // Step 1
    printf("move  %d disk from %c  to %c", source,dest);        // Step 2
    tower(n - 1, temp, source, dest)     // Step 3
  }

}
```



Queue Data structure

In our daily life, to catch a bus, to with draw money from ATM or to buy a cinema ticket, we form a queue. Queue is a first in first out FIFO linear data structure Queue is an important data structure for
computer applications even.

Queue is linear data structure in which deletion of element can take place only at one end called the front end and insertion can take place at the other end called the rear.



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Basic features of Queue

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. **peek( )** function is oftenly used to return the value of first element without dequeuing it.

Applications of Queue

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there 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, Call Center phone systems will use 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, First come first served.

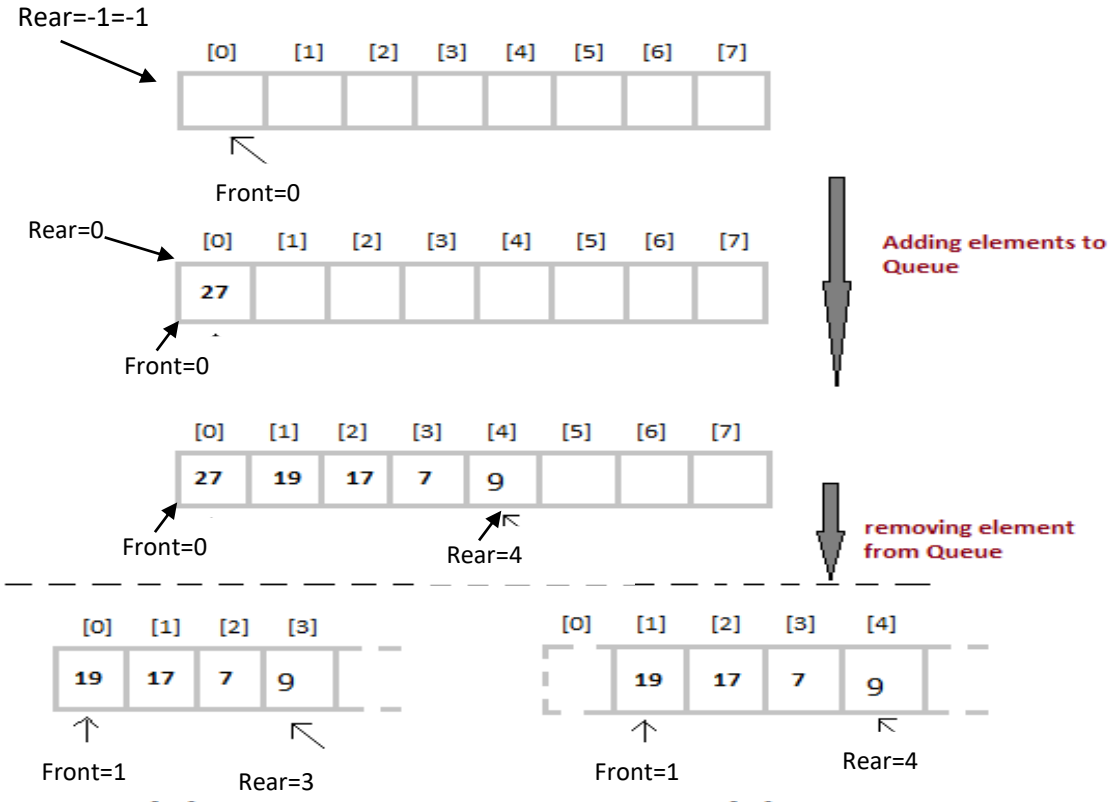
Implementation of Queue

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array. Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the **rear** keeps on moving ahead, always

pointing to the position where the next element will be inserted, while the **front** remains at the first index.

When we remove element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **front** position, and then one by one move all the other elements on position forward. In approach [B] we remove the element from **front** position and then move **front** to the next position.

In approach [A] there is an overhead of shifting the elements one position forward every time we remove the first element. In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the size on Queue is reduced by one space each time.



/*SIMPLE QUEUE

Design, Develop and Implement a menu driven Program in C for the following operations on
Simple QUEUE of Characters (Array Implementation of Queue with maximum size MAX)
a. Insert an Element on to QUEUE
b. Delete an Element from QUEUE
c. Demonstrate Overflow and Underflow situations on QUEUE
d. Display the status of QUEUE
e. Exit

Support the program with appropriate functions for each of the above operations
*/

```
#include<stdio.h>
#define SIZE 5
int i,rear=-1,front=0,option,j;
char q[SIZE];
int qFull()
{       if(rear==SIZE-1)
                return 1;
        else
                return 0;
}
int qEmpty()
{       if(front>rear)
                return 1;
        else
                return 0;
}

void enqueue(char ch)
{       if(qFull())
        {       printf(" Queue overflow\n");  return;  }
        q[++rear]=ch;
        return;
}

char dequeue()
{               return q[front++];     }

void display()
{       int i;
        for(i=front;i<=rear;i++)
                printf("%c  ",q[i]);
        printf("\n");
}
```

```
int main()
{       char ch;
        for(;;)
        {       printf("\n SIMPLE QUEUE\n");
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Display\n");
                printf("4.Exit");
                printf("\nEnter your option:");
                scanf("%d",&option);
                switch(option)
                {
                        case 1 :        printf("\nEnter the char:");
                                ch=getchar();
                                ch=getchar();
                                enqueue(ch);
                                break;
                        case 2 :        if(qEmpty())
                                        printf("\nQ is empty\n");
                                else
                                        printf("\nDeleted item is: %c",dequeue());
                                break;
                        case 3 :        if(qEmpty())
                                        printf("\nQ is empty\n");
                                else
                                        display();
                                break;
                        default  :   return 0;
                }
        }
}
```

Disadvantage Simple Queue

When the first element is serviced, the front is moved to next element. However, the position vacated is not available for further use. Thus, we may encounter a situation, wherein program shows that queue is full, while all the elements have been deleted are available but unusable, though empty.

Circular Queue

In a standard queue data structure re-buffering problem occurs for each dequeue operation. To solve this problem by joining the front and rear ends of a queue to make

the queue as a circular queue. Circular queue is a linear data structure. It follows FIFO principle.

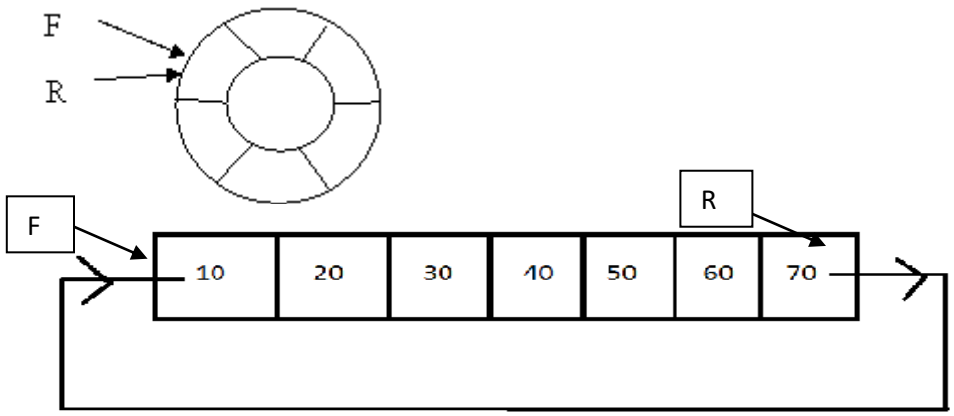
- In circular queue the last node is connected back to the first node to make a circle.
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as “Ring buffer”.

Circular Queue can be created in three ways they are

· Using linked list · Using arrays

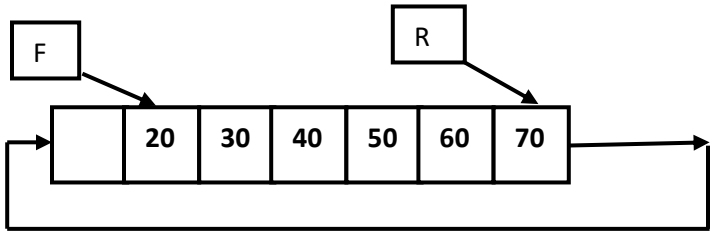
In arrays the range of a subscript is 0 to n-1 where n is the maximum size. To make the array as a circular array by making the subscript 0 as the next address of the subscript n-1 by using the formula subscript = (subscript +1) % maximum size. In circular queue the front and rear pointer are updated by using the above formula.

The following figure shows circular array:



Circular Queue

Queue shown in above figure is full.



Queue shown in above figure has one empty slot at the beginning, as first element is deleted from the queue. Below figure shows two elements deletion and queue has two empty slots at the beginning. Now element can be inserted from the beginning making use of rear pointer pointing to beginning location.

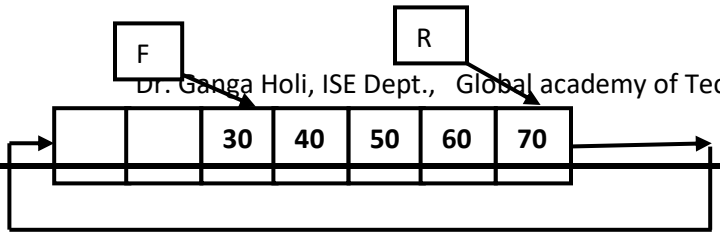
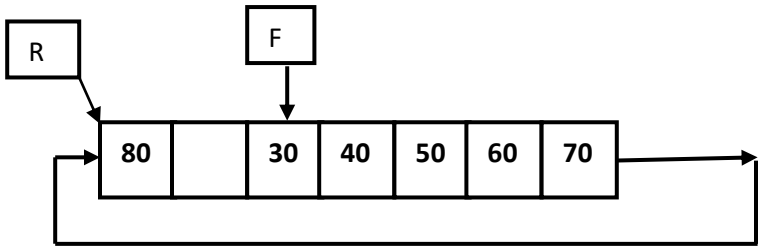


Figure shown below shows that element 80 is added at the beginning location and rear is now pointing to 0th location.



Every time rear is incremented by the value =(rear+1)%SIZE.
Every time front is incremented by the value=(front+1)%SIZE.

Algorithm for Enqueue operation using array

Step 1. start
Step 2. if (front == (rear+1)%max)
Print error “circular queue overflow “
Step 3. else
{ rear = (rear+1)%max
Q[rear] = element;
If (front == -1) f = 0;
}
Step 4. stop

Algorithm for Dequeue operation using array

Step 1. start
Step 2. if ((front == rear) && (rear == -1))
Print error “circular queue underflow “
Step 3. else
{ element = Q[front]
If (front == rear) front=rear = -1
Else
Front = (front + 1) % max
}
Step 4. stop

Circular Queue Program

Design, Develop and Implement a menu driven Program in C for the following operations on Circular QUEUE of Characters (Array Implementation of Queue with maximum size MAX)
a. Insert an Element on to Circular QUEUE
b. Delete an Element from Circular QUEUE
c. Demonstrate Overflow and Underflow situations on Circular QUEUE
d. Display the status of Circular QUEUE
e. Exit
Support the program with appropriate functions for each of the above operations
*/

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define SIZE 5
int q[SIZE],rear=-1,front=0,option,count=0;
int qFull()
{
        if(count==SIZE)
                return 1;
        else
                return 0;
}
int qEmpty()
{
        if(count==0)
                return 1;
        else
                return 0;
}

void enqueue(char ch)
{
        if(qFull())
        {       printf(" Queue overflow\n");  return;
        }
        rear=(rear+1)%SIZE;
        q[rear]=ch;  count++; printf(" rear=%d  count=%d\n",rear,count);
        return;
}

char dequeue()
{       char c;
        c=q[front];  front=(front+1)%SIZE; count--;
        return c;
}

void display()
{   int i; j=front;
        for(i=0;i<count; i++)
        {
                printf("%c  ",q[j]);
```

```
                j=(j+1)%SIZE;
        }
        printf("\n");
}

int main()
{       char ch;
        for(;;)
        {       printf("\n Circular QUEUE\n");
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Display\n");
                printf("4.Exit");
                printf("\nEnter your option:");
                scanf("%d",&option);
                switch(option)
                {
                        case 1 : printf("\nEnter the char:");
                                 ch=getchar();
                                 ch=getchar();
                                 enqueue(ch);
                                 break;
                        case 2 : if(qEmpty())
                                         printf("\nQ is empty\n");
                                 else
                                         printf("\nDeleted item is: %c",dequeue());
                                 break;
                        case 3 : if(qEmpty())
                                         printf("\nQ is empty\n");
                                 else
                                         display();
                                 break;
                        default :   return 0;
                }
        }
}
```

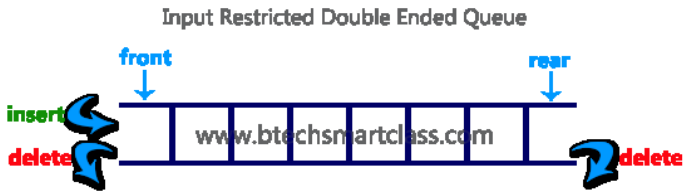
Dequeues- Double Ended Queues

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

- Input Restricted Double Ended Queue
- Output Restricted Double Ended Queue

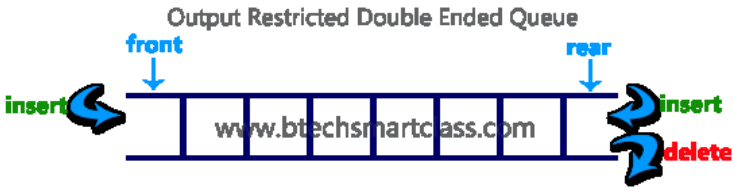
Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Priority Queue

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items 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 are assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

- **insert / enqueue** − add an item to the rear of the queue.
- **remove / dequeue** − remove an item from the front of the queue.

There are various ways of implementing the priority queues.

Double Ended Queue can be represented in various ways.

- Using single array
- Multiple arrays
- Linked list
- Heap

Using single array, we can implemented priority queue as follows...

Using single array

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6

int intArray[MAX];
int itemCount = 0;

int peek(){
   return intArray[itemCount - 1];
}

bool isEmpty(){
   return itemCount == 0;
}

bool isFull(){
   return itemCount == MAX;
}

int size(){
   return itemCount;
}

void insert(int data){
   int i = 0;

   if(!isFull()){
      // if queue is empty, insert the data
      if(itemCount == 0){
         intArray[itemCount++] = data;
```

```
    }else{
      // start from the right end of the queue

      for(i = itemCount - 1; i >= 0; i-- ){
        // if data is larger, shift existing item to right end
        if(data > intArray[i]){
          intArray[i+1] = intArray[i];
        }else{
          break;
        }
      }

      // insert the data
      intArray[i+1] = data;
      itemCount++;
    }
  }
}

int removeData(){
  return intArray[--itemCount];
}

int main() {
  /* insert 5 items */
  insert(3);
  insert(5);
  insert(9);
  insert(1);
  insert(12);

  // ------------------
  // index : 0  1 2 3 4
  // ------------------
  // queue : 12 9 5 3 1
  insert(15);

  // ----------------------
  // index : 0  1 2 3 4  5
  // ----------------------
  // queue : 15 12 9 5 3 1

  if(isFull()){
    printf("Queue is full!\n");
  }

  // remove one item
  int num = removeData();
  printf("Element removed: %d\n",num);

  // ----------------------
  // index : 0  1  2 3 4
  // ----------------------
  // queue : 15 12 9 5 3

  // insert more items
```

```
  insert(16);

  // ----------------------
  // index :  0  1 2 3 4  5
  // ----------------------
  // queue : 16 15 12 9 5 3

  // As queue is full, elements will not be inserted.
  insert(17);
  insert(18);

  // ----------------------
  // index : 0   1  2 3 4 5
  // ----------------------
  // queue : 16 15 12 9 5 3
  printf("Element at front: %d\n",peek());

  printf("----------------------\n");
  printf("index : 5 4 3 2  1  0\n");
  printf("----------------------\n");
  printf("Queue:  ");

  while(!isEmpty()){
    int n = removeData();
    printf("%d ",n);
  }
}
```

Using Multiple arrays

Multiple arrays are used to implement the priority queue. Array number represents the priority value. If the priority of the element is 1 then inserted that element in Q1,if the priority of the element is 2 then inserted that element in Q2, and so on. But deletion will happen from the highest priority queue first. All the elements of the highest priority queue will be deleted first, and then second priority queue elements and so on.

Using linked list we can implement the Priority Queue.

Two dimensional arrays can also be used to represent Priority Queue. Row represents the Priority value and order of arrival of the elements. Size of the array depends on the Priority Value.

Col→ Row | V	1	2	3	4	5	6
Q P -1	Q	P	H			
Q P -2	A	B	C	D	E	
Q P -3	T	P	R	M		

Related Questions on Queue & Stacks

- What is stack? What are the operations on stack?>
- What is the recursion? What are two important criteria for recursive functions?
- How does rear and front work in circular queue in C language?
- What are the applications of queues and stacks?
- How is priority queue implemented in C?
- Can we use an array to create a circular queue?
- What is the application of queues in computer science?
- What is the advance application of circular queue?
- What is queue and a circular queue?
- How does rear and front work in circular queue in C language?
- What is dequeue? How to insert and delete from dequeue.
- Describe the priority queue. How do we implement the priority queues?

Related Questions on Stacks

1. Entries in a stack are "ordered". What is the meaning of this statement?
    - A. A collection of stacks can be sorted.
    - B. Stack entries may be compared with the '<' operation.
    - C. The entries must be stored in a linked list.
    - D. There is a first entry, a second entry, and so on.
2. The operation for adding an entry to a stack is traditionally called:
    - A. add
    - B. append
    - C. insert
    - D. push
3. The operation for removing an entry from a stack is traditionally called:
    - A. delete
    - B. peek
    - C. pop
    - D. remove
4. Which of the following stack operations could result in stack underflow?
    - A. is_empty
    - B. pop
    - C. push
    - D. Two or more of the above answers
5. Which of the following applications may use a stack?
    - A. A parentheses balancing program.
    - B. Keeping track of local variables at run time.
    - C. Syntax analyzer for a compiler.

- D. All of the above.

6. Consider the following pseudocode:

```
   declare a stack of characters
   while ( there are more characters in the word to read )
   {
      read a character
      push the character on the stack
   }
   while ( the stack is not empty )
   {
      write the stack's top character to the screen
      pop a character off the stack
}
```

7. What is written to the screen for the input "carpets"?
   - A. serc
   - B. carpets
   - C. steprac
   - D. ccaarrppeettss
8. Here is an INCORRECT pseudocode for the algorithm which is supposed to determine whether a sequence of parentheses is balanced:

```
 declare a character stack
   while ( more input is available)
   {
      read a character
      if ( the character is a '(' )
         push it on the stack
      else if ( the character is a ')' and the stack is not empty )
         pop a character off the stack
      else
         print "unbalanced" and exit
    }
    print "balanced"
```

9. Which of these unbalanced sequences does the above code think is balanced?
   - A. `((())`
   - B. `())(()`
   - C. `(()()))`
   - D. `((()))()`
10. Consider the usual algorithm for determining whether a sequence of parentheses is balanced. What is the maximum number of parentheses that will appear on the stack AT ANY ONE TIME when the algorithm analyzes: `(()(()))(())`?
    - A. 1
    - B. 2
    - C. 3
    - D. 4
    - E. 5 or more

11. Consider the usual algorithm for determining whether a sequence of parentheses is balanced. Suppose that you run the algorithm on a sequence that contains 2 left parentheses and 3 right parentheses (in some order). What is the maximum number of parentheses that will ever appear on the stack AT ONE TIME during the computation?
    - A. 1
    - B. 2
    - C. 3
    - D. 4
    - E. 5 or more

12. Suppose we have an array implementation of the stack class, with ten items in the stack stored at data[0] through data[9]. The CAPACITY is 42. Where does the push member function place the new entry in the array?

Multiple Choice
Section 7.3
Implementations of the stack ADT

    - A. data[0]
    - B. data[1]
    - C. data[9]
    - D. data[10]

13. Consider the implementation of the stack using a partially-filled array. What goes wrong if we try to store the top of the stack at location [0] and the bottom of the stack at the last used position of the array?
    - A. Both peek and pop would require linear time.
    - B. Both push and pop would require linear time.
    - C. The stack could not be used to check balanced parentheses.
    - D. The stack could not be used to evaluate postfix expressions.
14. In the linked list implementation of the stack class, where does the push member function place the new entry on the linked list?
    - A. At the head
    - B. At the tail
    - C. After all other entries that are greater than the new entry.
    - D. After all other entries that are smaller than the new entry.
15. In the array version of the stack class (with a fixed-sized array), which operations require linear time for their worst-case behavior?
    - A. is_empty
    - B. peek
    - C. pop
    - D. push
    - E. None of these operations require linear time.
16. In the linked-list version of the stack class, which operations require linear time for their worst-case behavior?
    - A. is_empty
    - B. peek
    - C. pop
    - D. push
    - E. None of these operations require linear time.

Multiple Choice

Section 7.4
More Complex
Stack Applications

17. What is the value of the postfix expression 6 3 2 4 + - *:
    - A. Something between -15 and -100
    - B. Something between -5 and -15
    - C. Something between 5 and -5
    - D. Something between 5 and 15
    - E. Something between 15 and 100
18. Here is an infix expression: 4+3*(6*3-12). Suppose that we are using the usual stack algorithm to convert the expression from infix to postfix notation. What is the maximum number of symbols that will appear on the stack AT ONE TIME during the conversion of this expression?
    - A. 1
    - B. 2
    - C. 3
    - D. 4
    - E. 5