UNIT-II LINEAR DATA STRUCTURES - STACKS, QUEUES

Stack ADT - Operations - Applications - Evaluating arithmetic expressions - Conversion of Infix to postfix expression - Queue ADT - Operations - Circular Queue - Priority Queue - dequeue - applications of queues.

——— * ——— * ———

Introduction to Stack :-

Stacks are known as LIFO (Last in, first out) lists. It is an ordered list of the same type of elements. A stack is a linear list where all insertions and deletions are permitted only at one end of the list. When the elements are added to stack it grows at one end. Similarly, when the elements are deleted from a stack, it shrinks at the same end.

Here, insertions and deletions are performed is only one position, namely, the end of the list, called the **top**.

* The fundamental operations on a stack are **Push**, which is equivalent to an insert, and **Pop**, which deletes the most recently inserted element.

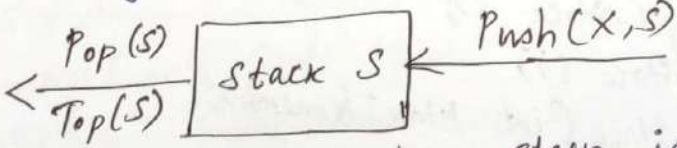


Fig. Stack model: input to a stack is by Push; output is by Pop.

* The most recently inserted element can be examined prior to performing a Pop by use of the Top routine.
* The model depicted in Fig. signifies only that Pushes are i/p operations and Pops and Tops are output.

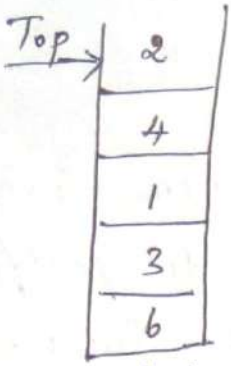


Fig. Stack model: only the top element is accessible.

Fig. shows an abstract stack after several operations.

* The Top is used to keep track of the index of the top most element.

Array Implementation of Stacks:-

In array implementation, if Top is -1, then it is an empty stack.

A stack is defined as a pointer to a structure. The structure contains the TopOfStack and Capacity fields. Once the maximum size is known, the stack array can be dynamically allocated.

```
#ifndef _Stack_h
struct StackRecord;
typedef struct StackRecord *Stack;

int IsEmpty (Stack S);
int IsFull(Stack S);
Stack CreateStack (int MaxElements);
void DisposeStack(Stack S);
void MakeEmpty(Stack S);
void Push(ElementType X, Stack S);
ElementType Top(Stack S);
void Pop(Stack S);
ElementType TopAndPop(Stack S);
#endif /* _Stack_h */
```

```
/* Place is implementation file */
/* Stack implementation is a dynamically allocated array */
#define EmptyTOS (-1)
#define MinStackSize (5)

struct StackRecord
{
  int Capacity;
  int TopOfStack;
  ElementType *Array;
};
```

Stack Creation:

```
Stack CreateStack (int MaxElements)
{
  Stack S;
  if (MaxElements < MinStackSize)
      Error ("Stack size is too small");
  /* Allocation memory */
  S = malloc (sizeof (struct StackRecord));
  if (S == NULL)
      FatalError ("Out of Space!!!");
  /* Allocation of Stack Array */
  S->Array = malloc (sizeof (ElementType) * MaxElements);
  if (S->Array == NULL)
      FatalError ("Out of space!!!");
  /* Initialize TopOfStack and Capacity */
  S->Capacity = MaxElements;
  MakeEmpty (S);
  return S;
}
```

Dispose the Stack:

```
void DisposeStack(Stack S)
{
  if(S != NULL)
  {
    free(S->Array);
    free(S);
  }
}
```

This routine first frees the stack array and then the stack structure.

IsEmpty:

```
int IsEmpty(Stack S)
{
  return S->TopOfStack == EmptyTOS;
}
```

MakeEmpty:

```
void MakeEmpty(Stack S)
{
  S->TopOfStack = EmptyTOS;
}
```

Push:

```
void Push(ElementType X, Stack S)
{
  if(IsFull(S))
    Error("Full stack");
  else
    S->Array[++S->TopOfStack] = X;
}
```

Return Top of Stack:

```
ElementType Top(Stack S)
{
  if(!IsEmpty(S))
    return S->Array[S->TopOfStack];
```

```
    Error ("Empty Stack");
    return 0;  /* Return value used to avoid warning */
}
```

Pop:

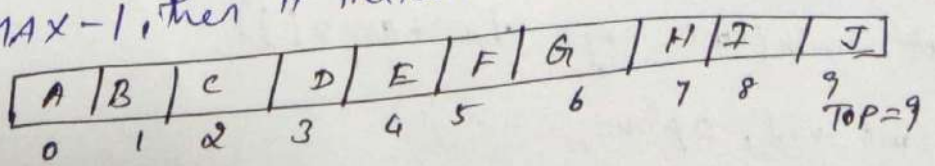
```
void pop(Stack S)
{
    if (IsEmpty(S))
        Error("Empty Stack");
    else
        S->TopOfStack--;
}
```

Routine to give top element and pop a stack:

```
ElementType TopAndPop(Stack S)
{
    if (!IsEmpty(S))
        return S->Array[S->TopOfStack--];
    Error("Empty Stack");
    return 0;
}
```

In Computer's memory, stacks can be represented as a linear array.

* Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack.
* It is the position where the element will be added to or deleted from.
* Another variable MAX is used to store the maximum no. of elements that the stack can hold.
* If TOP = NULL, then it indicates the stack is empty and if TOP = MAX-1, then it indicates the stack is full.



Operations on Stack:

A stack supports three basic operations: push, pop and peek.

push ⇒ adds an element to the top of the stack.
pop ⇒ removes the element from " " " " " .
peek ⇒ returns the value of the topmost element of stack.

Push:

Alg:

1). If TOP = MAX - 1
   Error "Overflow"
   goto step 4
2) TOP = TOP + 1
3) Stack [TOP] = value
4) Stop.

Pop:

Alg:

1). If TOP = NULL
   Error "Underflow"
   goto step 4
2). value = Stack [TOP]
3) TOP = TOP - 1
4) Stop

peek:

Alg:

1). If TOP = NULL
   Error "Stack is empty" goto step 2.
2). Return Stack [TOP]
3) Stop.

Program Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define MAX 3 //Altering this value changes size of stack created.

int st[MAX], top = -1;
void push (int s[], int val);
int pop (int st[]);
int peek (int st[]);
void display (int st[]);
int main (int argc, char *argv[])
{
   int val, option;
```

```
do
{
  printf ("\n **** Main Menu ****");
  printf ("\n 1. PUSH \n 2. POP \n 3. PEEK \n 4. DISPLAY
                                    \n 5. EXIT ");
  printf ("\n Enter the option: ");
  scanf ("%d", &option);
  switch (option)
  {
    case 1:
        printf ("\n Enter the number to be pushed on stack:")
        scanf ("%d", &val);
        push (st, val);
        break;
    case 2:
        val = pop(st);
        if (val != -1)
        printf ("\n the value deleted from stack is %d",
                                    val);
        break;
    case 3:
        val = peek (st);
        if (val != -1)
        printf ("\n The value stored at TOP is: %d", val);
        break;
    case 4: display (st); break;
  } while (option != 5);
  return 0;
}
void push (int s[], int val)
{
  if (top == MAX-1)
  {
    printf ("\n Stack Overflow");
  }
```

```
    else
    {
      top++;
      st[top] = val;
    }
}
int pop(int st[])
{
  int val;
  if (top == -1)
  {
    printf("\n Stack Underflow");
    return -1;
  }
  else
  {
    val = st[top];
    top--;
    return val;
  }
}
void display(int s[])
{
  int i;
  if (top == -1)
    printf("\n stack is empty");
  else
  {
    for(i=top; i>=0; i--)
    printf("\n %d", st[i]);
  }
}
int peek(int st[])
{
  if(top == -1)
  {
    printf("\n Stack is empty");
    return -1;
  }
  else
    return(st[top]);
}
```

Linked List Implementation of Stack :-

Type declaration :-

```
#ifndef _Stack_h
struct node;
typedef struct Node *PtrToNode;
typedef PtrToNode Stack;

int IsEmpty (Stack S);
Stack CreateStack (void);
void DisposeStack (Stack S);
void MakeEmpty (Stack S);
Void Push (ElementType X, Stack S);
ElementType Top(Stack S);
void Pop (Stack S);
#endif   /* _Stack_h */
/* Place in implementation file */
/* Stack implementation is a linked list with a header */
struct Node
{
  ElementType Element;
  PtrToNode Next;
};
```

IsEmpty Routine :

```
int IsEmpty (Stack S)
{
  return S->Next == NULL;
}
```

Create an Empty Stack :

```
Stack CreateStack (void)
{
  Stack S;
  S = malloc (sizeof (struct Node));
  if (S == NULL)
    FatalError (" Out of Space !!! ");
```

```
    MakeEmpty(S);
    return S;
}
void MakeEmpty(Stack S)
{
  if(S==NULL)
      Error("In Must use CreateStack First");
  else while(!IsEmpty(S))
         Pop(S);
}
```

Push:

```
void Push(ElementType X, Stack S)
{
  PtrToNode TmpCell;
  TmpCell=malloc(sizeof(struct Node));
  if(TmpCell==NULL)
      FatalError("Out of Space!!!");
  else
  {
    TmpCell->Element=X;
    TmpCell->Next = S->Next;
    S->Next=TmpCell;
  }}
```

Top:

```
ElementType Top(Stack S)
{
  if(!IsEmpty(S))
      return S->Next->Element;
  Error("Empty Stack");
  return 0;
}
```

Pop:

```
void Pop(Stack S)
{
  PtrToNode FirstCell;
  if (IsEmpty(S))
     Error("Empty Stack");
  else
  {
    FirstCell = S->Next;
    S->Next = S->Next->Next;
    free(FirstCell);
  }
}
```

Representation:

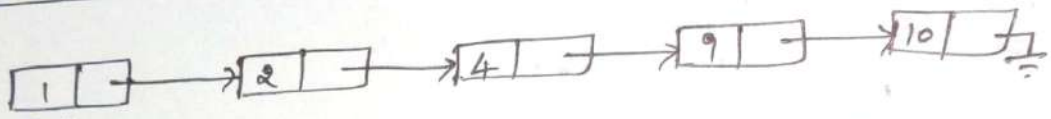


Fig. Linked List representation of Stack

Here if TOP == NULL, then the stack is empty. The head pointer of the linked list is used as TOP. All the insertions (push) and deletions (pop) are done at the node pointed by TOP.

Operations on Stack:

1. Push:

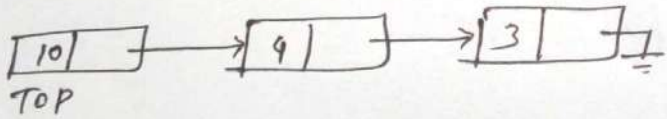


Fig. Before push of 15

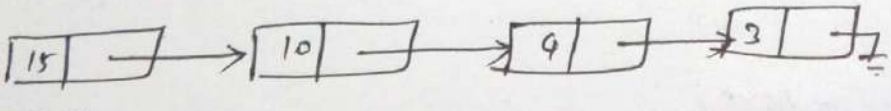


Fig. After push of 15

Algo:

1). Allocate memory for the new node and name it as NEW_NODE

2). NEW_NODE -> Data = Value.

3). If TOP = Null

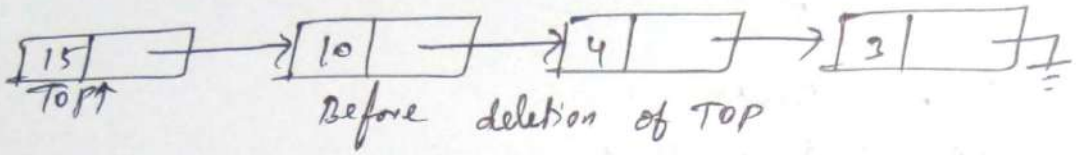
NEW_NODE -> Next = Null

TOP = NEW_NODE

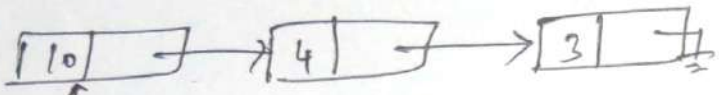
Else
NEW_NODE → Next = TOP
TOP = NEW_NODE

4). Stop

2). Pop :



Before deletion of TOP



After deletion of TOP

Alg :
1). If TOP = Null
print "Underflow"
2). ptr = TOP
3). TOP = TOP → Next
4) free (ptr)
5) stop

Program Code:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
struct stack
{
  int data;
  struct stack *next;
};
struct stack *top = NULL;
struct stack *push(struct stack *, int);
struct stack *display(struct stack *);
struct stack *pop(struct stack *);
int peek(struct stack *);
```

```
int main (int argc, char *argv[])
{
  int val, option;
  do
  {
    printf ("\n *** Main Menu ***");
    printf ("\n 1. PUSH \n 2. POP \n 3. PEEK \n 4. DISPLAY
                    \n 5. EXIT ");
    printf ("\n Enter your choice:");
    scanf ("%d" &option);
    switch (option)
    {
      case 1:
          printf ("\n Enter the number to be pushed: ");
          scanf ("%d", &val);
          top = push (top, val);
          break;
      case 2:
          top=pop(top);
          break;
      case 3:
          val = peek(top);
          if(val != -1)
            printf("\n The value at TOP is %d", val);
          else
            printf("\n Empty stack");
          break;
      case 4:
          top = display (top);
          break;
    }
  } while (option!=5);
  return 0;
}
```

```
struct stack *push (struct stack *top, int val).
{
  struct stack *ptr;
  ptr = (struct stack *) malloc (size of (struct stack));
  ptr -> data = val;
  if (top == NULL)
    {
      ptr -> Next = NULL;
      top = ptr;
    }
  else
    {
      ptr -> next = top;
      top = ptr;
    }
  return top;
}
struct stack * display (struct stack * top)
{
  struct stack *ptr;
  ptr = top;
  if (top == NULL)
    printf (" Empty stack ");
  else
  {
    while (ptr != NULL)
      { printf ("\n %d ", ptr -> data);
        ptr = ptr -> next;
      }
  }
  return top;
}
struct stack * pop (struct stack * top)
  {
    struct stack *ptr;
    ptr = top;
```

```
if (top == NULL)
    printf("\n Stack Underflow");
else
{
  top = top -> next;
  printf("\n The value being deleted is %d", ptr -> data);
  free(ptr);
}
return top;
}
int peek (struct stack *top)
{
if (top == NULL)
    return -1;
else return top -> data;
}
```

Applications of Stack:-

1). Expression conversion
   (a) Infix to postfix
   (b) Infix to prefix
   (c) Postfix to infix
   (d) Prefix to infix
2). Expression Evaluation
3). Parsing
4). Simulation of recursion
5). Function call
6) Balancing Symbols
7) Reversing a list
8) Recursion
9) Tower of Hanoi.

Evaluation of Arithmetic Expressions:

Expression Representation:

(3) methods

1). Infix $x+y$ operator b/n operands
2). prefix $+xy$ operator before operands
3). postfix $xy+$ operator after operands

Postfix:

The expression $(A+B)*C$ can be written as: (infix)

$[AB+]*C$

$AB+C*$ is the postfix notation.

The postfix expression does not even follow the rules of operator precedence.

* The operator first in the expression is operated first on the operands.
* For example, given a postfix expression $AB+C*$. While evaluation, addition will be performed prior to multiplication.
* Thus, operators are applied to the operands that are immediately left to them.

Prefix:

The prefix expression is evaluated from left to right.

* Here, the operator is placed before the operands.
* For example, if $A+B$ is an expression (infix) in infix notation, then the corresponding expression in prefix notation is $+AB$. While evaluating a prefix expression, the operators are applied that are present immediately on the right of the operato.

Conversion of an Infix into a Postfix Expression:

Let I be an algebraic expression written in infix notation. I may contain parentheses, operands and operators. The precedence of these operators can be given as:

Higher priority *, /, %.

Lower priority +, −

Ex: Infix to prefix

1) (A+B) * C
   (+AB) * C
   * +AB

2) (A−B) * (C+D)
   [−AB] * [+CD]
   * −AB+CD

3) (A+B) / (C+D) − (D*E)
   [+AB] / [+CD] − [*DE]
   [/ +AB+CD] − [*DE]
   − /+AB+CD*DE

Infix to postfix

1) (A − B) * (C+D)
   [AB−] * (CD+)
   AB− CD+ *

2). (A+B) / (C+D) − (D*E)
   [AB+] / [CD+] − [DE*]
   [AB+ CD+ /] − [DE*]
   AB+ CD+ / DE*−

* The order of evaluation of these operators can be changed by making use of parenthesis.
* For example, (A+B) * C, will evaluate A+B first and then the result will be multiplied with C.

Alg:

The alg accepts an infix expression that may contain operators, operands, and parenthesis.
* The alg uses a stack to temporarily hold operators.
* The postfix expression is obtained from left-to-right using the operands from the infix expression, and the operators which are removed from the stack.

* The first step in to push a left parenthesis on the stack and to add a corresponding right parenthesis at the end of infix expression.

* The alg in repeated until the stack is empty.

1). Add (Push) "(" on to the stack

2). Repeat until each character in infix notation is scanned

if a "(" is encountered, push it on the stack

if an operand (whether a digit or a character) is encountered, add it to the postfix expression.

if a ")" is encountered then

(a). Repeatedly pop from stack and add it to the postfix expression until a "(" is encountered.

(b). Discard the "(". That is, remove the "(" from stack and do not add it to the postfix expression.

if an operator O is encountered, then

(a). Repeatedly pop from stack and each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than O.

(b) Push the operator O to the stack.

[END IF]

3). Repeatedly pop from stack and add it to postfix expression until the stack is empty.

4) EXIT

Example:
Convert infix to postfix expression using stack

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

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
-	(-	A
(	(- (	A
B	(- (	A B
/	(- (/	A B
C	(- (/	A B C
+	(- (+	A B C /
(	(- (+ (	A B C /
D	(- (+ (	A B C / D
%	(- (+ (%	A B C / D
E	(- (+ (%	A B C / D E
*	(- (+ (% *	A B C / D E
F	(- (+ (% *	A B C / D E F
)	(- (+	A B C / D E F * % +
/	(- (+ /	A B C / D E F * %
G	(- (+ /	A B C / D E F * % G
)	(-	A B C / D E F * % G / +
*	(- *	A B C / D E F * % G / +
H	(- *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * -

Evaluation of Postfix Expression:-

Alg:

1). Add a ")" at the end of the postfix expression
2). Scan every character of the postfix expression and repeat step 3 and 4 until ")" is encountered.
3). If an operand is encountered,
push it on the stack.
If an operator O is encountered, then
(a). Pop the top two elements from the stack as A and B as A and B.
(b). Evaluate B O A, where A is the topmost element and B is the element below A.
(c). Push the result of evaluation on the stack.
[END IF]
4). Set Result equal to the topmost element of the stack.
5). Exit.

Example: Postfix Expression $\underbrace{9-((3+4)+8)/4}_{\text{infix}} \Rightarrow \underbrace{934+8+4/-}_{\text{postfix}}$

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

Here, infix expression is converted into postfix form then this alg is applied.

Pgm: Code to Conversion : Infix to Postfix

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
char st[MAX];
int top=-1;
void push (char st[], char);
char pop( char st[]);
void InfixtoPostfix(char source[], char target[]);
int getPriority(char);
int main()
{
  char infix[100], postfix[100];
  clrscr();
  printf("\n Enter the infix expression: ");
  gets(infix);
  strcpy (postfix, "");
  InfixtoPostfix(infix, postfix);
  printf("\n The corresponding postfix expression is:");
  puts(postfix);
  return 0;
}
void InfixtoPostfix(char source[], char target[])
{
  int i=0, j=0;
  char temp;
  strcpy(target, "");
  while (source[i] != '\0')
  {
    if(source[i] == '(')
    {
      push (st, source[i]);
    }i++;
```

```
else if (source[i] == ')')
{
  while ((top != -1) && (st[top] != '('))
  {
    target[j] = pop(st);
    j++;
  }
  if (top == -1)
  {
    printf("\n Incorect expression.");
    exit(1);
  }
  temp = pop(st); // remove left parenthesis
  i++;
}
else if (isdigit(source[i]) || isalpha(source[i]))
{
  target[j] = source[i];
  j++;
  i++;
}
else if (source[i] == '+' || source[i] == '-' || source[i] == '*'
    || source[i] == '/' || source[i] == '%')
{
  while ((top != -1) && (st[top] != '(') &&
    (getPriority(st[top]) > getPriority(source[i])))
  {
    target[j] = pop(st);
    j++;
  }
  push(st, source[i]);
  i++;
}
```

```
    else
    {
      printf("\n Incorrect element in expression.");
      exit(1);
    }
  }
  while ((top != -1) && (st[top] != '('))
  {
    target[j] = pop(st);
    j++;
  }
  target[j] = '\0';
}
int getPriority(char op)
{
  if (op == '/' || op == '*' || op == '%')
      return 1;
  else if (op == '+' || op == '-')
      return 0;
}
void push(char st[], char val)
{ if (top == MAX-1)
      printf("\n Stack Overflow !!! ");
  else
  {
    top++;
    st[top] = val;
  }
}
char pop(char st[])
{ char val = ' ';
  if (top == -1)
      printf("\n Stack Underflow !!! ");
  else
  { val = st[top]; top--;
  }return val;
}
```

Program Code for Evaluation of Postfix Expression:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define MAX 100
float st[MAX];
int top = -1;
void push(float st[], float val);
float pop(float st[]);
float evaluatePostfix(char exp[]);
int main()
{
  float val;
  char exp[100];
  clrscr();
  printf("\n Enter the postfix expression: ");
  gets(exp);
  val = evaluatePostfix(exp);
  printf("\n Value of Postfix expression = %.2f", val);
  return 0;
}
float evaluatePostfix(char exp[])
{
  int i=0;
  float op1, op2, value;
  while(exp[i] != '\0')
  {
    if(isdigit(exp[i]))
      push(st, (float)(exp[i] - '0'));
    else
    {
      op2 = pop(st);
      op1 = pop(st);
```

Pgm Code for Evaluation of Postfix Expression continuation.

```
switch( exp[i])
{
  case '+':
       value = op1+op2;
       break;
  case '-':
       value = op1-op2;
       break;
  case '/':
       value= op1/op2;
       break;
  case '*':
       value =op1 *op2;
       break;
  case '%':
       value = (int) op1 % (int) op2;
       break;
  };
  push(st, value);
}
i++;
}
return (pop(st));
}
void push (float st[], float val)
{ if (top==MAX-1)
   printf("\n Stack Overflow!!!");
 else
 {  top ++;
    st[top] = val;
} }
float pop( float st[])
{ float val = -1;
  if (top==-1)
     printf ("\n Stack Underflow !!!");
  else
  { val = st[top]; top--; }
   return val; }
```

Conversion of Infix to Prefix Expression:

Alg 1:

1). Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.
2). Obtain the postfix expression of the infix expression obtained in step1.
3). Reverse the postfix expression to get the prefix expression.

Example:

Infix expression ⇒ (A-B/C) * (A/K-L)

Step1:

(L-K/A) * (C/B-A)' ⇒ Reverse the string

Step2:

Postfix expression of (L-K/A) * (C/B-A) is got as

[LKA/-] * [CB/A-]

LKA/-CB/A-*

Step 3:

*-A/BC-/AKL ⇒ Reverse the postfix string.

Alg 2:

1). Scan each character in the infix expression. For this, repeat steps-8 until the end of infix expression.
2). Push the operator into the operand stack, and ignore all the left parenthesis until a right parenthesis is encountered
3). Pop operand 2 from operand stack
4). Pop operand 1 " "
5). Pop operator from operator stack.
6). Concatenate operator and operand1.
7). Concatenate result with operand 2.
8). Push result into the operand stack.
9) END

Pgm Code to convert infix to prefix expression:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
#define MAX 100
char st[MAX];
int top=-1;
void reverse(char str[]);
void push(char st[], char);
char pop(char st[]);
void InfixtoPrefix(char source[], char target[]);
int getPriority(char);
char infix[100], postfix[100], temp[100];
int main()
{
    clrscr();
    printf("\n Enter the infix expression: ");
    gets(infix);
    reverse(infix);
    strcpy(postfix, "");
    InfixtoPrefix(temp, postfix)
    printf("The corresponding postfix expression is: ");
    puts(postfix);
    strcpy(temp, "");
    reverse(postfix);
    printf("\n The prefix expression is: ");
    puts(temp);
    getch();
    return 0;
}
```

```
void reverse (char str[])
{ int len, i=0, j=0;
  len = strlen(str);
  j = len-1;
  while (j>=0)
  { if (str[j]=='(')
      temp[i] = ')';
    else if (str[j]==')')
      temp[i] = '(';
    else
      temp[i] = str[j];
    i++; j--;
  }
  temp[i] = '\0';
}
void Infixto Postfix (char source[], char target[])
{
  int i=0, j=0;
  char temp;
  strcpy(target, "");
  while (source[i] != '\0')
  { if (source[i]=='(')
    { push(st, source[i]);
      i++;
    }
    else if (source[i]==')')
    { while ((top!=-1) && (st[top]!='('))
      {
```

```
        target[j] = pop(st);
        j++;
      }
      if (top == -1)
      {
        printf("\n Incorrect expression");
        exit(1);
      }
      temp = pop(st); //remove left paenthesis
      i++;
    }
    else if (isdigit(source[i]) || isalpha(source[i]))
    {
      target[j] = source[i];
      j++;
      i++;
    }
    else if (source[i] == '+' || source[i] == '-' ||
             source[i] == '*' || source[i] == '/' ||
             source[i] == '%')
    {
      while ((top != -1) && (st[top] != '(') &&
             (getPriority(st[top]) >= getPriority(source[i])))
      {
        target[j] = pop(st);
        j++;
      }
      push(st, source[i]);
      i++;
    }
```

```
else
{
  printf("\n Incorrect elements in expression.");
  exit(1);
}
}while((top!=-1) && (st[top]!='('))
{
   target[j] = pop(st);
    j++;
}
target[j]='\0';
}
int getPriority (char op)
{
  if (op=='/' || op=='*' || op=='%')
      return 1;
  else if(op=='+' || op=='-')
      return 0;
  }
  void push(char st[], char val)
  {
    if(top == MAX-1)
       printf("\n Stack Overflow");
    else
    {
       top++;
       st[top] = val;
    }
  }
```

```
char pop (char st[])
{
  char val = ' ';
  if (top == -1)
     printf ("\n Stack Underflow");
  else
    {
      val = st[top];
      top--;
    }
  return val;
}
```

Evaluation of Postfix Expression:-

Algo:

1) Accept the prefix expression
2) Repeat until all the character in the prefix expression have been scanned.
   (a) Scan the prefix expression from right, one character at a time.
   (b) If the scanned character is an operand, push it on the operand stack.
   (c) If the scanned character is an operator, then
      (i) Pop two values from operand stack
      (ii) Apply the operator on the popped operands
      (iii) Push the result on the operand stack.

Example:
Prefix expression + - 9 2 7 * 8 / 4 12.

Character scanned	Operand Stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
-	24, 5
+	29

Pgm Code to evaluate prefix expression:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
int st[10];
int top=-1;
int pop();
void push(int);
int main()
{
 char prefix[10];
 int len, val, i, opr1, opr2, res;
 clrscr();
 printf("\n Enter the prefix expression:");
 gets(prefix);
 len = strlen(prefix);
 for(i = len - 1; i>=0; i--)
 {
```

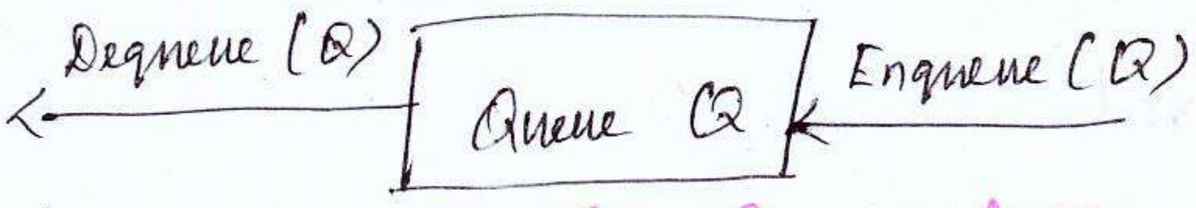
```
switch( get_type(prefix[i]))
{
  case 0:
       val = prefix[i] - '0';
       push (val);
       break;
  case 1:
       opr1 = pop();
       opr2 = pop();
       switch(prefix[i])
       {
          case '+':
               res = opr1 + opr2;
               break;
          case '-':
               res = opr1 - opr2;
               break;
          case '*':
               res = opr1 * opr2;
               break;
          case '/':
               res = opr1 / opr2;
               break;
       }
       push(res);
  }
}
printf("\n Result = %d", st[0]);
return 0;
}
void push(int val)
{
  st[++top] = val;
}
int pop()
{
  return (st[top--]);
}
```

```
int get_type(char c)
{
  if(c=='+' || c=='-' || c=='*'
        || c=='/')
     return 1;
  else
     return 0;
}
```

QUEUE ADT

Queue is a linear data structure with First In, First Out manner. (FIFO). The elements that are inserted first is the first one to be taken out. The elements in a queue are added at one end called the REAR and removed from the other end called the FRONT.

The basic operations on a queue are Enqueue, which inserts an element at the end of the list (rear), and Dequeue, which deletes the element at the start of the list (front)



ARRAY IMPLEMENTATION OF QUEUE ADT:

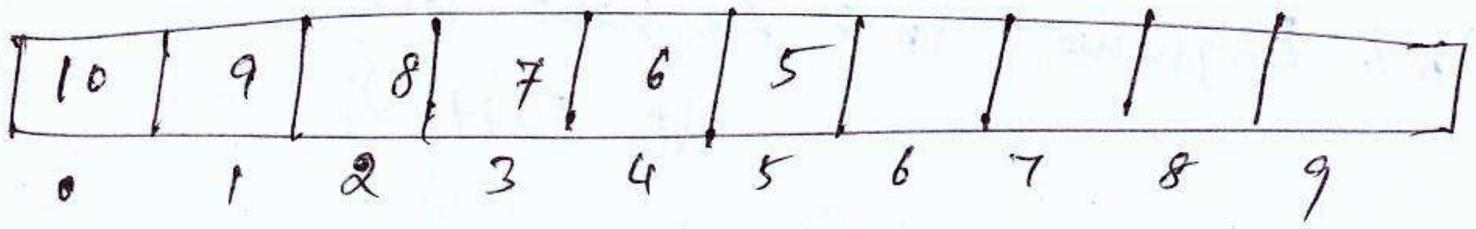
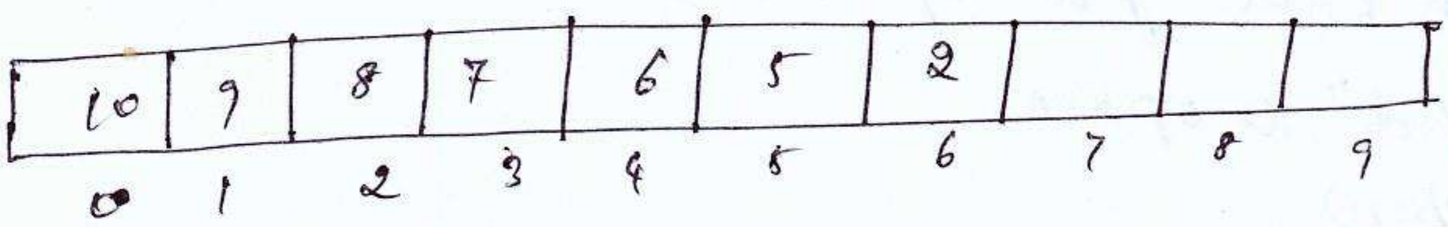
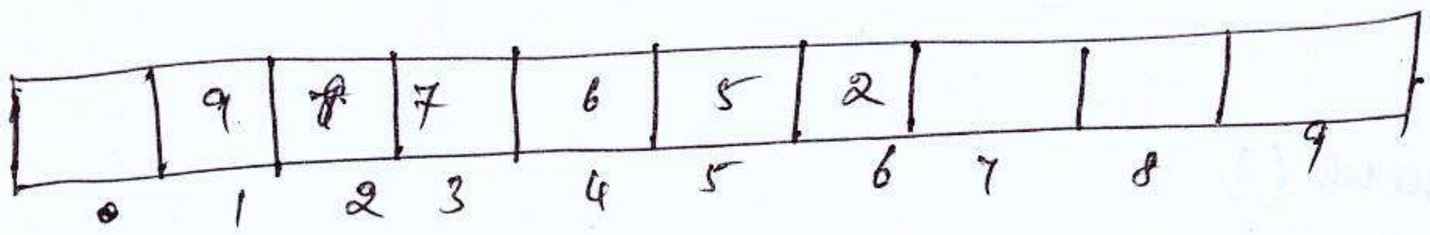


Fig. Queue
Front = 0, Rear = 5



Queue after insertion of a new element
Front = 0, Rear = 6



Queue after deletion of an element
Front = 1, Rear = 6.

Enqueue - Alg

1) If rear = MAX - 1
   write Overflow
   go to step 4
2) If front = -1 and rear = -1
   set front = rear = 0
   else
   set rear = rear + 1
   END If
3) set Queue [rear] = num
4) EXIT

Dequeue - Alg

1) If front = -1 or front > rear
   write Underflow
   else
   set val = Queue [front]
   set front = front + 1
   End If
2) EXIT

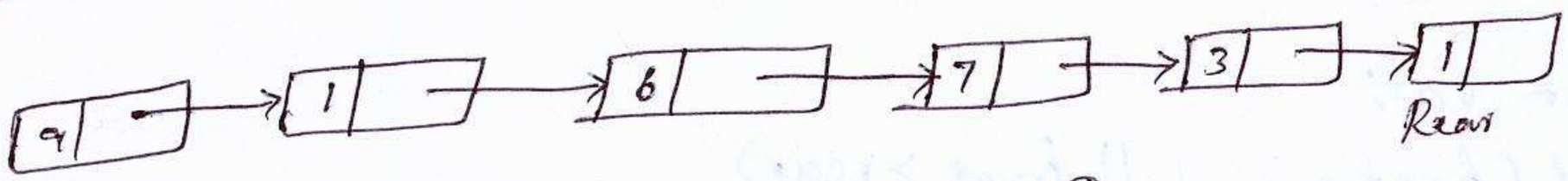
Pgm code :

```
// Array Implementation of Queue ADT
#include <stdio.h>
#include <conio.h>
#define MAX 10
int queue[MAX];
int front = -1, rear = -1;
void enqueue (void);
int dequeue (void);
int peek (void);
void display(void);
int main()
{
    int option, val;
    do
    {
        printf ("\n\n **** Main Menu ****");
        printf ("\n 1. Enqueue \n 2. Dequeue \n 3. Peek \n 4. Display
                \n 5. Exit");
        printf ("\n Enter your option: ");
        scanf ("%d", &option);
        switch (option)
        {
        case 1:
            enqueue ();
            break;
        case 2:
            val = dequeue ();
            if (val != -1)
            printf ("\n The number deleted is : %d", val);
            break;
        case 3:
            val = peek ();
            if (val != -1)
            printf ("\n The front value in queue is : %d", val);
            break;
```

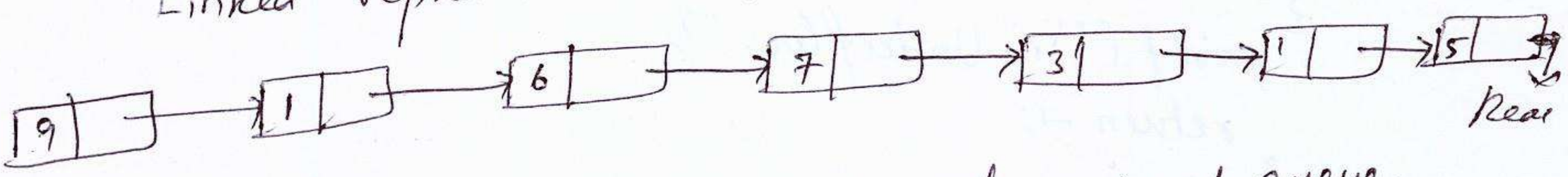
```
        case 4:
            display();
            break;
        }
    } while (option!=5);
    return 0;
}
void enqueue()
{
    int num;
    printf("\n Enter the number to be inserted: ");
    scanf("%d", &num);
    if (rear == MAX-1)
        printf("\n Overflow");
    else if (front==-1 && rear==-1)
        front=rear=0;
    else
        rear++;
    queue[rear] = num;
}
int dequeue()
{
    int val;
    if (front==-1 || front>rear)
    {
        printf("\n Underflow");
        return -1;
    }
    else
    {
        val=queue[front];
        front++;
        if (front>rear)
            front=rear=-1;
        return val;
    }
}
```

```
int peek()
{
  if (front == -1 || front > rear)
  {
    printf("\n Queue is empty");
    return -1;
  }
  else
      return queue[front];
}
void display()
{
  int i;
  printf("\n");
  if (front == -1 || front > rear)
     printf("\n Queue is empty");
  else
  {
    for(i=front; i <= rear; i++)
        printf("\t %d", queue[i]);
  }
}
```

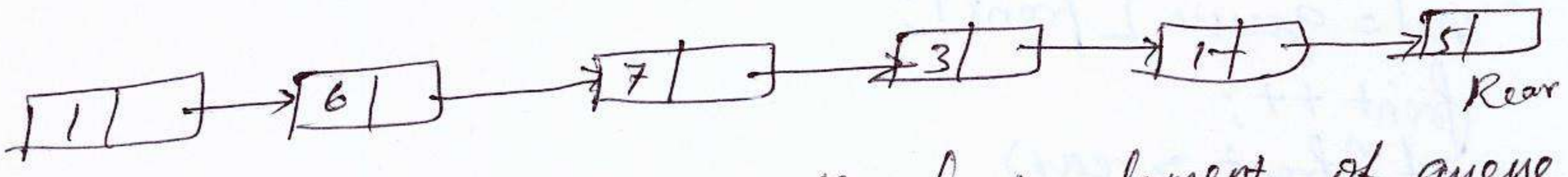
LINKED LIST IMPLEMENTATION OF QUEUE ADT:



Linked representation of Queue.



Insertion of 5 as the last element of queue.



Deletion of 9 as the first element of queue.

The storage requirement of linked representation of a queue with n elements is O(n) and typical time for operations is O(1).

In a linked queue, every element has two parts - one that stores the data and another that stores the address of the next element.

* The START pointer of the linked list is used as FRONT.
* Here, we will also use the another pointer called REAR, which will store the address of the last element in the queue.
* All insertions will be done at the rear end and all the deletions will be done at the front end.
* If FRONT = REAR = NULL, then it indicates that the queue is empty.

Insert (Enqueue) Alg:

1) Allocate memory for the new node and name it as PTR.
2) Set PTR → Data = val
3) If FRONT = NULL
   Set FRONT = REAR = PTR
   Set FRONT → NEXT = REAR → NEXT = NULL
   else
   Set REAR → NEXT = PTR
   Set REAR = PTR
   Set REAR → NEXT = NULL
   [End if]
4) END

Deletion (Dequeue) Alg:-

1) If FRONT = NULL
   write "Underflow"

   [END IF]
2) Set PTR = FRONT
3) Set FRONT = FRONT -> NEXT
4) Free PTR
5) END

Pgm Code:

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
 struct node
{
 int data;
 struct node *next;
};
struct queue
{
 struct node *front;
 struct node *rear;
};
struct queue *q;
void create_queue (struct queue *);
struct queue *insert (struct queue *, int);
struct queue *delete_element (struct queue *);
struct queue *display (struct queue *);
 int peek (struct queue *);
 int main()
{
   int val, option;
   create_queue(q);
   clrscr();
```

```
do
{
  printf("\n **** Main Menu ****");
  printf("\n 1. Insert  \n 2. Delete  \n 3. Peek  \n 4. Display
                                                          \n 5. Exit");
  printf("\n Enter your option:");
  scanf("%d", &option);
  switch(option)
  {
  case 1:
     printf("\n Enter the no. to insert in queue:");
     scanf("%d", &val);
     q= insert(q, val);
     break;
  case 2:
     q = delete_element(q);
     break;
  case 3:
     val = peek(q);
     if(val == -1)
        printf("\n The value at front of
                                queue is %d", val);

     break;
  case 4:
     q=display(q);
     break;
  }
  } while(option!=5)
  getch();
  return 0;
}
```

```
void create-queue (struct queue *q)
{
  q->rear = NULL;
  q -> front = NULL;
}
struct queue * insert (struct queue *q, int val)
 {
   struct node *ptr;
   ptr = (struct node*) malloc (sizeof (struct node));
   ptr -> data = val;
   if (q -> front == NULL)
   {
     q -> front = ptr;
     q -> rear = ptr;
     q -> front -> next = q -> rear -> next = NULL;
   }
   else
   {
     q -> rear -> next = ptr;
     q -> rear = ptr;
     q -> rear -> next = NULL;
   }
   return q;
 }
struct queue *display (struct queue *q)
{
  struct node *ptr;
  ptr = q -> front;
  if (ptr == NULL)
      printf ("\n Queue is empty");
```

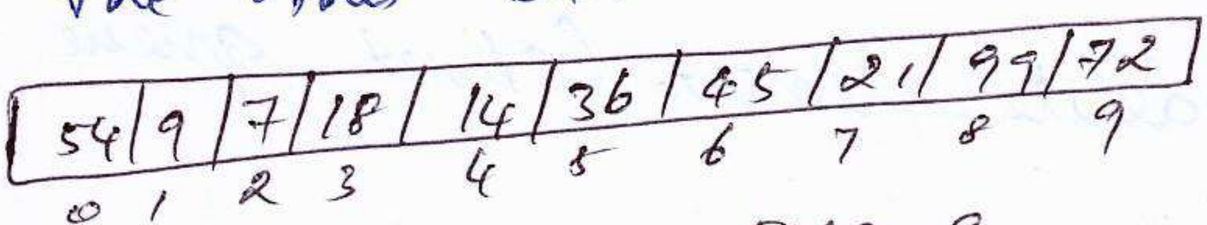
```
else
{
  printf("\n");
  while( ptr != q->rear)
  {
    printf("%d\t", ptr->data);
    ptr = ptr->next;
  }
  printf(" %d\t," ptr->data);
 }
  return q;
}
struct queue *delete_element (struct queue *q)
{
  struct node *ptr;
  ptr = q->front;
  if(q->front == = NULL)
      printf("\n Underflow");
else
{
  q->front = q->front->next;
  printf("\n The value being deleted is %d",
                                  ptr->data);
  free(ptr);
}
return q;
}
int peek (struct queue *q)
{ if(q->front == = NULL)
  { printf("\n Queue is empty");
    return -1;
  }
  else return q->front->data;
}
```

Types of Queues;-

1). Circular Queue
2). Deque (Double Ended Queue)
3). Priority Queue
4). Multiple Queue

Circular Queue

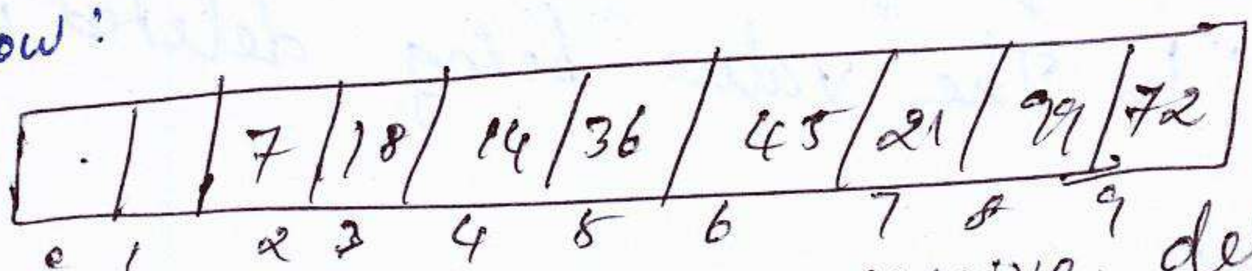
In linear queues, insetion can be done only at one end called the REAR and deletion are always done at the other end called the FRONT.



Here FRONT = 0 , REAR = 9.

Now, if you want to insert another value, it will not be possible. because the queue is completely full. There is no empty space where the value can be inserted.

Comider a scenario in which two successive deletions are made. The queue will then be given as shown below:



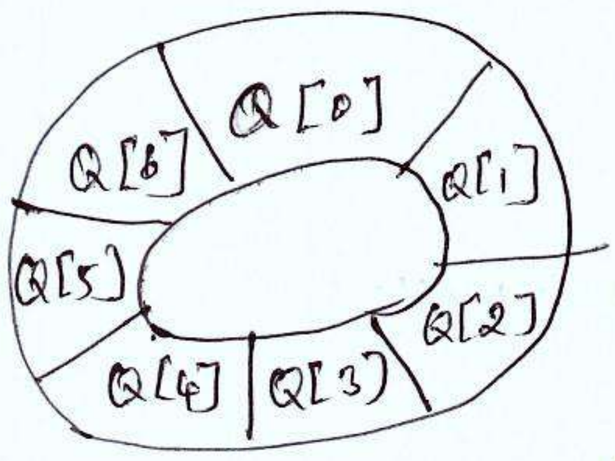
Queue after two successive deletions.

Here FRONT = 2 , REAR = 9.

Suppose we want to insert a new element in the queue. Eventhough there is space available, the overflow condition still exists because the condition REAR = MAX - 1 still hold true. This is a major drawback of a linear queue.

To resolve this pbm, we have (2) solutions:

1) Shift the elements to the left so that the vacant spaces can be occupied and utilized efficiently. But this can be very time-consuming, especially ~~with~~ when the queue is large.
2) Using Circular Queue, the first index comes after the last index.



Circular Queue.

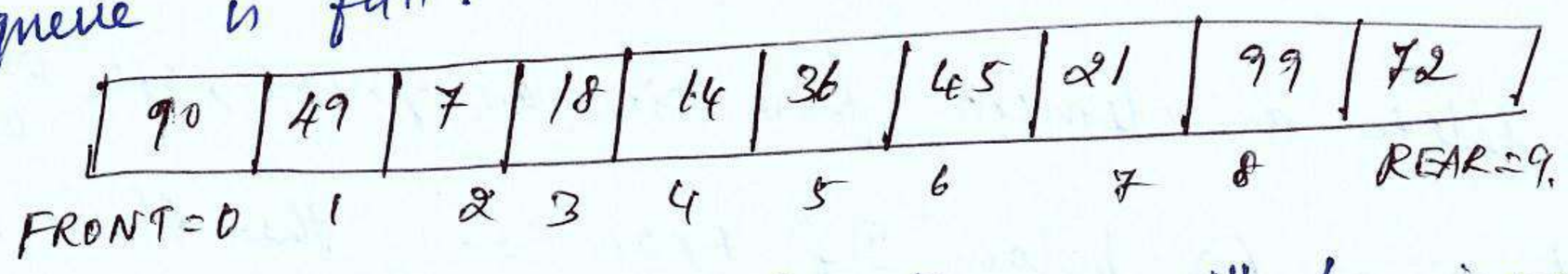
The circular queue will be full only when FRONT=0 & REAR = MAX-1.

* The circular queue is implemented in the same manner as a linear queue is implemented.
* The only difference will be in the code that performs insertion and deletion operations.

Insertion:

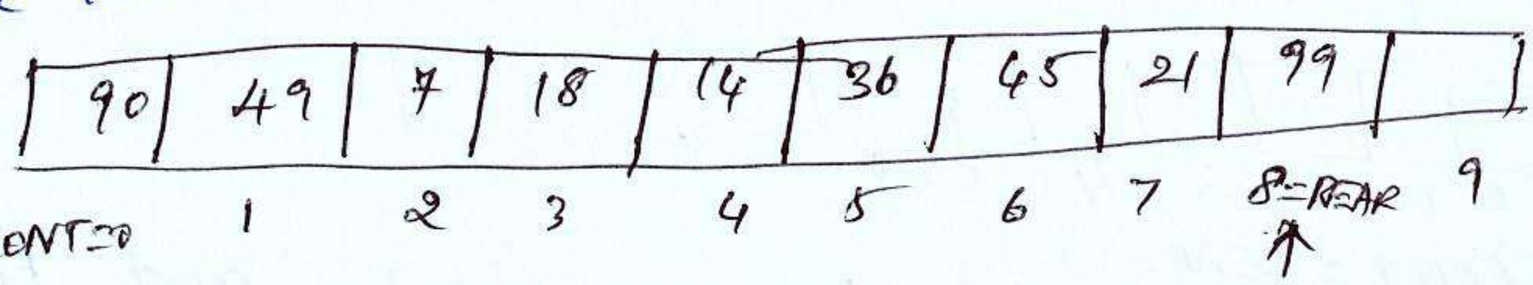
* For insertion, we have to check the (3) conditions:

1) If FRONT=0 and REAR = MAX-1, then the circular queue is full.



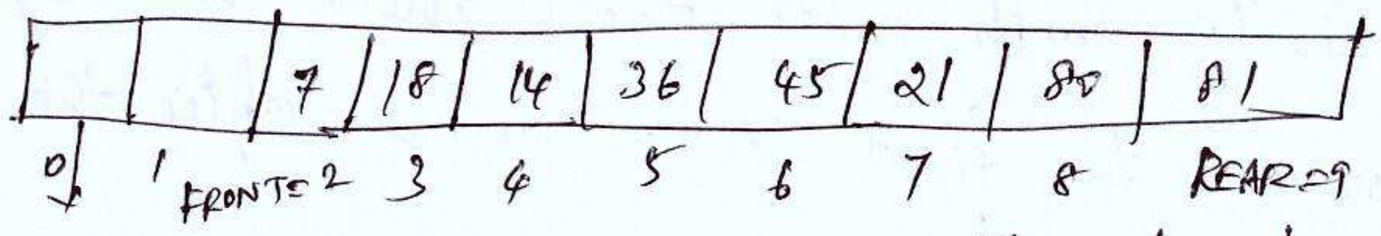
Full Queue.

2) If REAR != MAX-1, then REAR will be incremented and the value will be inserted as



Increment rear so that it points to location 9 and insert the value here.

3) If FRONT != 0 and REAR = MAX-1, then it means that the queue is not full. So set REAR = 0 and insert the new element there as in Fig. below.



Set REAR = 0 and insert the value here.

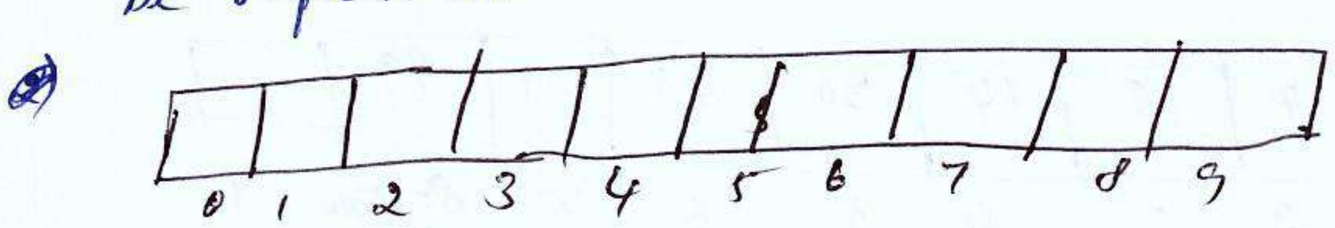
Algorithm:

1) If FRONT = 0 and REAR = MAX-1
   Write "Overflow"
   goto step 4
   END If
2) If FRONT = -1 and REAR = -1
   Set FRONT = REAR = 0.
   Else If REAR = MAX-1 and FRONT != 0
   Set REAR = 0
   Else
   Set REAR = REAR+1
   END IF
3) Set QUEUE[REAR] = VAL
4) EXIT

Deletion:

To delete an element from circular queue, we check ③ conditions:

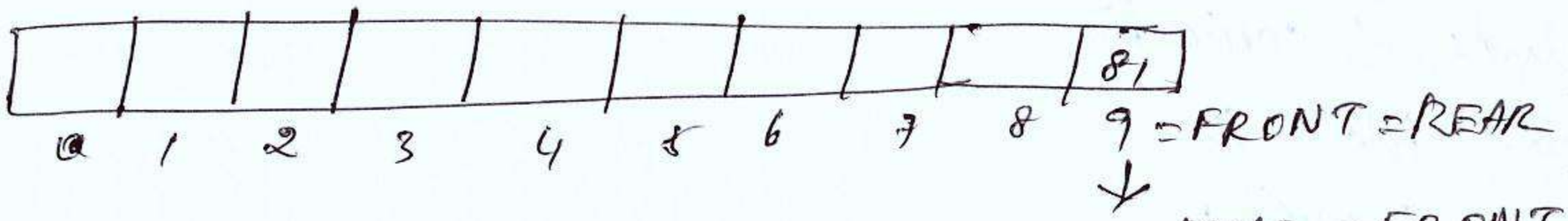
1) Look at Fig below - If FRONT = -1, then there are no elements in the queue. So, an underflow condition will be repeated.



FRONT = REAR = -1

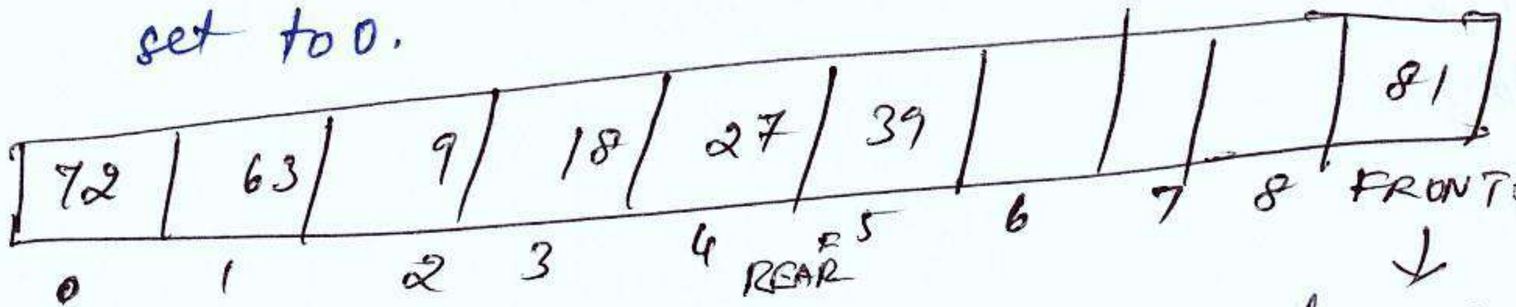
2). If the queue is not empty and FRONT = REAR, then after deleting the element at the front the

queue becomes empty and so FRONT and REAR are set to -1.



Delete this element and set REAR = FRONT = -1.

3). If the queue is not empty. and FRONT = MAX-1, then after deleting the element at the front, FRONT is set to 0.



Delete this element and set FRONT = 0

Algorithm:

1) If FRONT = -1
   Write "Underflow"
   Goto step 4
   END If
2) Set VAL = QUEUE[FRONT]
3) If FRONT = REAR
   Set FRONT = REAR - 1
   Else If FRONT = MAX - 1
   Set FRONT = 0
   Else Set FRONT = FRONT + 1
   END If
4) EXIT

Pgm code :

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int queue[MAX];
int front = -1, rear = -1;
void insert(void);
int delete_element(void);
int peek(void);
void display();
int main()
{
   int option, val;
   clrscr();
   do
   {
      printf("\n **** Main Menu ****");
      printf("\n 1. Insert  \n 2. Delete \n 3. Peek \n 4. Display
                                        \n 5. Exit");
printf("\n Enter you option: ");
scanf("%d", &option);
switch(option)
{
      case 1:
            insert();
            break();
      case 2:
            val = delete_element();
            if(val == -1)
               printf("\n The element deleted is %d",
                                        val);
      break;
case 3:
      val = peek();
      if(val == -1)
         printf("\n The first value is %d", val);
      break;
```

```
        case 4:
            display();
            break;
    }
  }while(option!=5);
  return 0;
}
void insert()
{
  int num;
  printf("\n Enter the number to be inserted in the queue:");
  scanf("%d",&num);
  if(front == 0 && rear == MAX-1)
        printf("\n Overflow");
  else if(front == -1 && rear == -1)
  {
     front=rear=0;
     queue[rear]=num;
  }
  else if(rear=MAX-1 && front!=0)
  {
     rear=0;
     queue[rear]=num;
  }
  else
  {  rear++;
     queue[rear]=num;
  }
}
```

```
void delete_element()
{
  int val;
  if (front == -1 && rear == -1)
  {
    printf("\n Underflow");
    return -1;
  }
  val = queue[front];
  if (front == rear)
      front = rear = -1;
  else
  { ; if (front == MAX-1)
      front = 0;
    else
      front++;
  }
    return val;
  }
int peek()
{ if(front == -1 && rear == -1)
  { printf("\n Queue is empty");
    return -1;
  }
  else return queue[front];
}
```

```
void display()
{
    int i;
    printf("\n");
    if (front == -1 && rear == -1)
        printf("\n Queue is empty");
    else
    {   if (front < rear)
        {   for (i = front ; i <= rear ; i++)
                printf("\t %d", queue[i]);
        }
        else
        {   for ( i=front ; i < MAX ; i++)
                printf("\t %d", queue[i]);
            for ( i=0; i<=rear ; i++)
                printf("\t %d", queue[i]);
        }
    }
}
```

PRIORITY QUEUES :-

A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed.

* The general rules of processing the elements of a priority queue are.

1). An element with higher priority is processed before an element with a lower priority.

2). Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

* A priority queue can be thought of as a modified queue in which an element has to be removed from the queue, the one with the highest-priority is retrieved first.

* These are widely used in operating systems to execute the highest priority process first.

* The priority of the process may be set based on the CPU time it requires to get executed completely.

Implementation of a Priority Queue:

(2) ways to implement a priority queue.

1) We can use a sorted list to store the elements so that when an element has to be taken out, the queue will not have to be searched for the element with the highest priority.

2) We can use unsorted list so that insertions are always done at the end of the list. Every time when an element has to be removed, the highest priority element will be searched & removed.

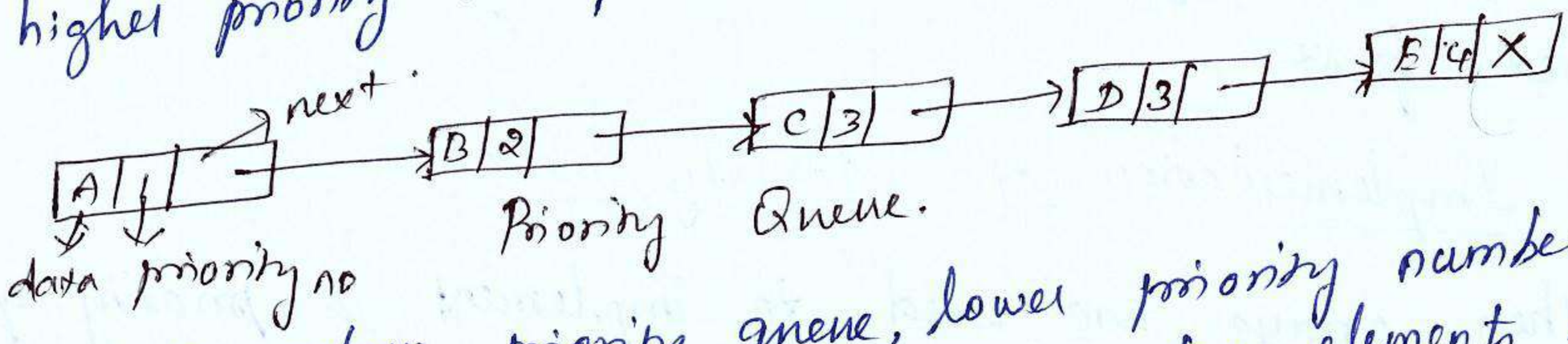
* A sorted list takes O(n) time to insert an element in the list.
* It takes only O(1) time to delete an element.
* An unsorted list will take O(1) time to insert an element and O(n) time to delete an element from the list.

Practically, both the techniques are inefficient and usually a blend of these two approaches is adopted that takes roughly O(log n) time or less.

Linked Representation of Priority Queue;-

When a priority queue is implemented using a linked list, every node of the list consists of (3) parts
1) the information/data part
2) the priority number of an element
3) the address of the next element

* If we use a sorted linked list, then the element with the higher priority will precede the element with lower priority.



Priority Queue.

In the above priority queue, lower priority number means higher priority. For example, if there are two elements A and B, where A has a priority number 1 and B has a priority no 2, then A will be processed before B.

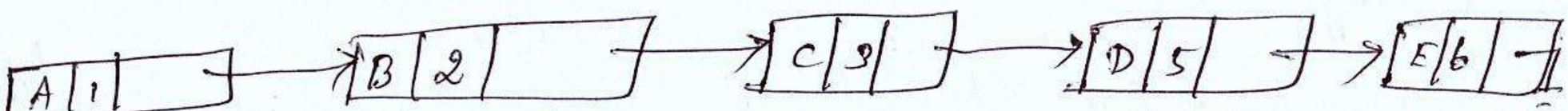
* When two elements have the same priority the elements are arranged and processed in FCFS principle.
↓
First Come First Served.

Insertion:

For insertion of an element, we have to traverse the entire list until we find a node with lower priority than that of the new element.

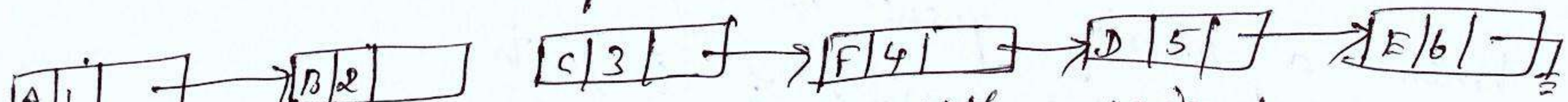
* The new element is inserted before the node with the lower priority.

* Suppose, if two elements has the same priority, the new element is inserted after that element.



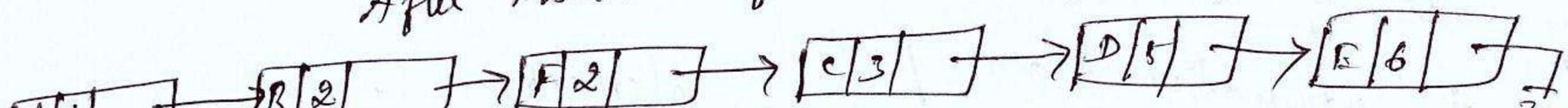
Before insertion.

(i)



After insertion of F with priority 4.

(ii)



After insertion of F with priority 2 (same as B priority)

Deletion:

Deletion is very simple process, where the first node of the list will be deleted and the data of that node will be processed first.

Array Implementation of a Priority Queue:

When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained.

* Each of these queues will be implemented using circular arrays or circular queues.

* Every individual queue will have its own FRONT and REAR pointers.

We use a two-dimensional array where each queue will be allocated the same amount of space.

FRONT	REAR
3	3
1	3
4	5
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3				E	F
4	I			G	H

Fig. Priority Queue Matrix

FRONT[x] and REAR[x] contain the front and rear values of row x, where x is the priority number.

Insertion:

To insert a new element with priority 'x', add the element at the rear end of row x, where x is the row no as well as the priority no of that element.

* For example, if we want to insert an element 'R' with priority '3', then the priority queue is shown below:

FRONT	REAR
3	3
1	3
4	1
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3	R			E	F
4	I			G	H

Fig. Priority queue matrix after insertion of a new element

Deletion:

To delete an element, first we find non-empty queue and then process the front element of the first non-empty queue.

* In our assumption, the first non-empty queue is the one with priority no 'i', so A will be deleted and processed first

Pgm code for Linked Implementation of Priority Queue:

```
#include <stdio.h>
#include <malloc.h>
#include <conio.h>
struct node
{
    int data;
    int priority;
    struct node *next;
};
struct node *start = NULL;
struct node *insert (struct node *);
struct node *delete (struct node *);
void display (struct node *);
int main()
{
    int option;
    clrscr();
do
{
    printf("\n *** Main Menu ***");
    printf("\n 1. Insert \n2. Delete \n3. Display \n4. Exit");
    printf("\n Enter your option:");
    scanf("%d", &option);
    switch (option)
    {
    case 1:
        start = insert (start);
        break;
    case 2:
        start = delete (start);
        break;
    case 3:
        display (start);
        break;
    }
} while (option != 4);
}
```

```
struct node *insert (struct node *start)
{
   int val, pri;
   struct node *ptr, *p;
   ptr= (struct node *) malloc (sizeof (struct node));
   printf ("\n Enter the value and its priority :");
   scanf (" %d %d", &val, &pri);
   ptr→ data = val;
   ptr→ priority = pri;
   if (start == NULL || pri < start→ priority)
   {
      ptr→ next = start;
      start = ptr;
   }
   else
   {
      p = start;
      while (p→ next != NULL && p→ next→ priority <= pri)
            p = p→ next;
      ptr→ next = p→ next;
      p→ next = ptr;
   }
   return start;
}
struct node * delete (struct node *start)
{ struct node *ptr;
   if (start == NULL)
   {
      printf ("\n Underflow");
      return;
   }
```

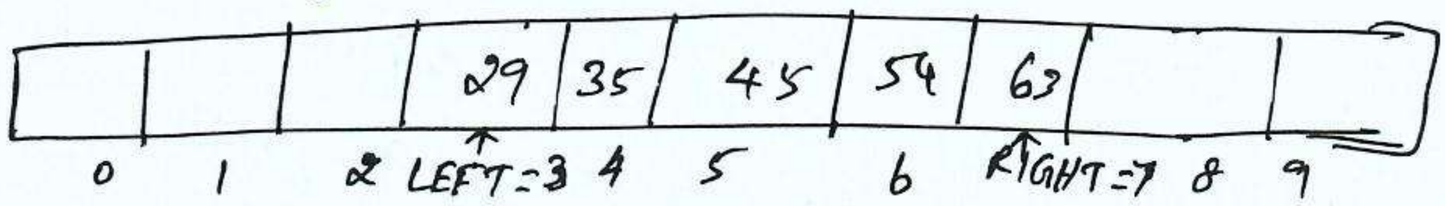
```
else
{
    ptr = start;
    printf("\n Deleted item is : %d", ptr->data);
    start = start->next;
    free(ptr);
}
return start;
}
void display(struct node *start)
{
  struct node *ptr;
  ptr = start;
  if(start == NULL)
      printf("\n Queue is empty");
  else
  {
  printf("\n Priority Queue is:");
  while(ptr != NULL)
  {
      printf("\t %d [priority = %d]", ptr->data,
                                      ptr->priority);

      ptr= ptr->next;
  }
}
}
```

Deques:

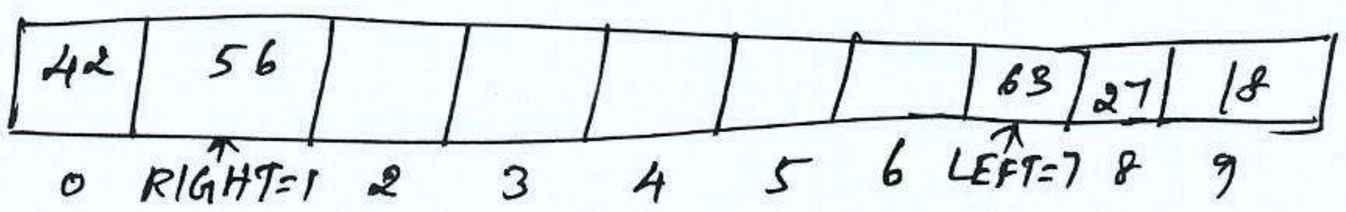
A deque is a list in which the elements can be inserted or deleted at either end. It is also known as head-tail linked list because the elements can be added to or removed from either the front (head) or the back (tail) end.

A deque can be implemented using either a circular array or a circular doubly linked list.

* In deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque.
* The elements in a deque extend from the LEFT end to the RIGHT end since it is circular. Dequeue[N-1] is followed by Dequeue[0].



Double Ended Queues (Deque)



(2) variants of double-ended queue.

1). Input restricted deque:
- Insertions can be done only at one of the ends, while deletions can be done from both ends.

2). Output restricted deque:
- Deletions can be done only at one of the ends, while insertions can be done on both ends.

Pgm code to implement i/p and o/p restricted degues:

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int deque[MAX];
int left = -1, right = -1;
void input_deque(void);
void output_deque(void);
void insert_left(void);
void insert_right(void);
void delete_left(void);
void delete_right(void);
void display(void);
int main()
{ int option;
clrscr();
printf("\n **** Main Menu ****");
printf("\n 1. Input restricted deque");
printf("\n 2. Output restricted deque");
printf("\n Enter you option: ");
scanf("%d", &option);
switch(option)
{
  case 1:
        input_deque();
        break;
  case 2:
        output_deque();
        break;
}
return 0;
}
```

```
void input_deque()
{
  int option;
   do
   {
     printf("\n INPUT RESTRICTED DEQUE");
     printf("\n 1. Insert at right");
     printf("\n 2. Delete from left");
     printf("\n 3. Delete from right");
     printf("\n 4. Display \n 5. Quit");
     printf("\n Enter your option:");
     scanf("%d",&option);
     switch (option)
     {
        case 1:
                insert_right();
                break;
        case 2:
                delete_left();
                break;
        case 3:
                delete_right();
                break;
        case 4:
                display();
                break;
     }
   }while(option!=5);
}
void output_deque()
{
  int option;
  do
  {
     printf("\n OUTPUT RESTRICTED DEQUE");
     printf("\n 1. Insert at right");
```

```
printf ("\n 2. Insert at left");
printf ("\n 3. Delete from left");
printf ("\n 4. Display  \n 5. Quit");
printf ("\n Enter your option: ");
 scanf ("%d", &option);
switch (option)
{
case 1:
        insert_right();
        break;
case 2:
        insert_left();
        break;
case 3:
        delete_left();
        break;
case 4:
        display();
        break;
  }while (option!=5);
}
void insert_right()
{
   int val;
   printf ("\n Enter the value to be added: ");
    scanf ("%d", &val);
   if ((left==0 && right==MAX-1) || (left==right+1))
   {
      printf ("\n Overflow");
      return;
   }
```

```
if (left == -1)   /* if queue is initially empty */
    left = right = 0;
else
{
    if (right == MAX-1) /* right is at last position of
                           queue */
        right = 0;
    else
        right = right + 1;
    }
    deque[right] = val;
}
void insert_left()
{
    int val;
    printf ("\n Enter the value to be added: ");
    scanf ("%d", &val);
    if ((left == 0 && right == MAX-1) || (left == right+1))
    {
        printf ("\n Overflow");
        return;
    }
    if (left == -1) /* If queue is initially empty */
        left = right = 0;
    else
    {
        if (left == 0)
            left = MAX-1;
        else left = left - 1;
    }
    deque[left] = val;
}
```

```
void delete_left()
{
  if (left == -1)
   {
     printf("\n Underflow");
     return;
   }
  printf("\n The deleted element is : %d", deque[left]);
  if (left == right) /* Queue has only one element */
        left = right = -1;
  else
  {  if (left == MAX-1)
        left=0
     else
        left = left+1;
  }
}
void delete_right()
{
  if (left == -1)
  {  printf("\n Underflow");
     return;
  }
  printf("\n The element deleted is : %d", deque[right]);
  if(left == right) /* Queue has only one element */
     left = right = -1;
  else
  {  if(right=0)
        right = MAX-1;
     else
        right = right-1;
  }
}
```

```
void display()
{
  int front = left , rear = right;
  if (front == -1)
    {
      printf ("\n Queue is empty ");
      return;
    }
  printf ("\n The elements of the queue are : ");
  if(front <= rear)
  {
     while (front <= rear)
     {
       printf (" %d ", deque[front]);
       front ++;
     }
  }
else
{ while (front < MAX-1)
  {
    printf (" %d ", deque[front]);
     front ++;
  }
front = 0;
 while (front <= rear)
 {
   printf (" %d", deque[front]);
    front ++;
 }
}
printf("\n");
}
```

Applications of Queues:

* Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
* " " used to transfer data (asynchronously (data not necessarily received at same rate as sent) b/n two processes. (IO buffers), e.g. pipes, file IO, sockets.
* " " used as buffers on MP3 players and portable CD players, iPod playlist.
* " " used in Playlist for jukebox to add songs to the end, play from the front of the list.
* " " used in operating systems for handling interrupts.
  - When programming a real-time system that can be interrupted, for example, by a mouse click, it is necessary to process the interrupts immediately, before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate DS.
* Scheduling of processes
* Spooling
* A queue of client processes waiting to receive the service from the server procesr.
* Various appln s/w using non-linear data structure tree or graph requires a queue for <u>breadth first traversal.</u>

Josephus Problem;-

In Josephus pbm, 'n' people stand in a circle waiting to be executed.

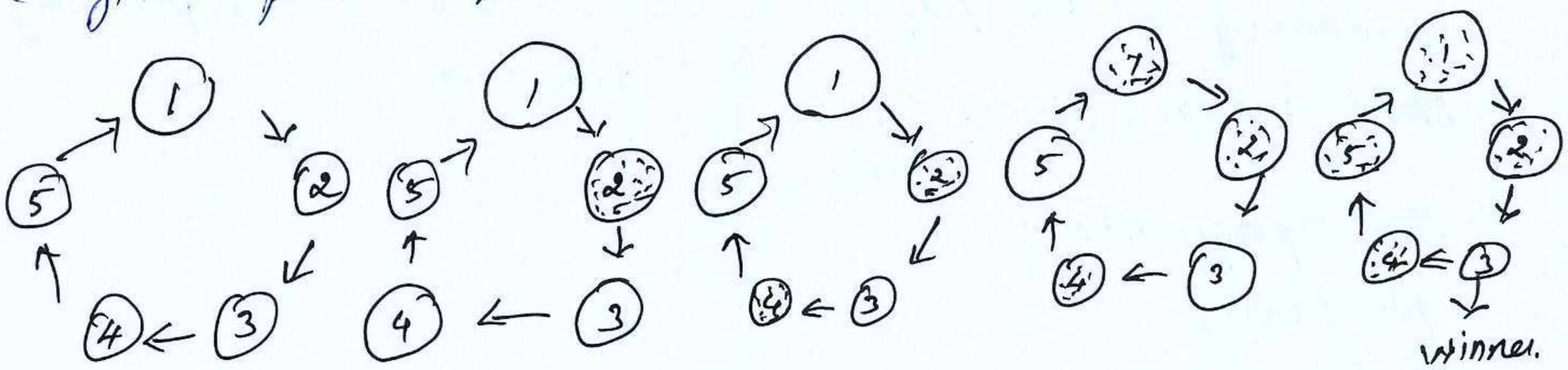
* The counting starts at some point in the circle and proceeds in a specific direction (clockwise / anti-clockwise) around the circle.

* In each step, a certain no. of people are skipped and the next person is executed.

* The elimination of people makes the circle smaller and smaller.

* At the last step, only one person remains who is declared the "winner".

Therefore, if there are 'n' number of people and a number 'k', which indicates that k-1 people are skipped and k-th person in the circle is eliminated, then the pbm is to choose a position in the initial circle so that the given person becomes the winner.



For example, if there are 5 (n) people and every second (k) person is eliminated, then first the person at position 2. is eliminated followed by person at position 4 followed by person at position 1 and finally the person at position 5 is eliminated. ∴, the person at position '3' is the winner.

Pgm Code:

```
#include <stdio.h>
# include <conio.h>
#include <malloc.h>
 struct node
 { int player_id;
   struct node * next;
 }
Struct node *start, *ptr,
                *new_node;

int main()
{
   int n,k, i, cocunt;
   clrscr();
   printf("\n Enter the no. of
      players : ");
   Scanf ("%d", &n);
   printf ("\n Enter the value
            of k : ");
   scanf("%d", &k);
   //create circular linked list
   containing all the players
   start = malloc (sizeof (struct
                    node));
   Start->player_id = 1;
   ptr = start;
   for(i=2; i<n; i++)
   {
     new_node = malloc (sizeof
                (struct node));
     ptr->next = new_node;
     new_node->player_id = i;
     new_node->next = start;
     ptr = new_node;
   }
   for(count = n; count >1; count--)
   {
     for( i=0; i<k-1; ++i)
       {
         ptr = ptr-> next;
       }
     ptr->next = ptr->next->next;
       // Remove the eliminated
          player from the circular
              linked list
   }
   printf("\n The winner is
          player %d", ptr->player_id);
   return 0;
}
```

* — * — * — * — *