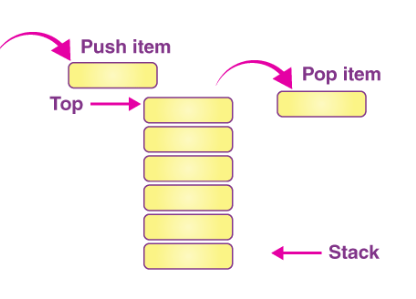
**3.1 STACK**

**3.1.1 INTRODUCTION TO STACK DATA STRUCTURE**

* A Stack is a linear data structure that holds a linear, ordered sequence of elements. It is an abstract data type. A Stack works on the LIFO process (Last In First Out), i.e., the element that was inserted last will be removed first. To implement the Stack, it is required to maintain a pointer to the top of the Stack, which is the last element to be inserted because we can access the elements only on the top of the Stack.



1. **PUSH:**

 PUSH operation implies the insertion of a new element into a Stack. A new element is always inserted from the topmost position of the Stack; thus, we always need to check if the top location is empty or not, i.e., TOP=Max if this condition goes true, it means the Stack is full, and no more elements can be inserted, and even if we try to insert the element, a Stack overflow message will be displayed. Here top shows the no of elements in our stack. If top is 0 , there are zero elements in the stack , if top is max there are max noof elements in the stack , they are from 0 to max-1.

Algorithm :

Step1: if (top==max)

OVERFLOW

Step2: stack[top]=new value;

Step3: top=top+1;

Step4: stop

1. **POP:** POP means to delete an element from the Stack. Before deleting an element, make sure to check if the Stack Top is 0, i.e., TOP=0 If this condition goes true, it means the Stack is empty, and no deletion operation can be performed, and even if we try to delete, then the Stack underflow message will be generated.

**Algorithm:**

Step1 : if (top==0) UNDERFLOW

Step2: top=top-1

Step3: out=stack[top];

Step4: print out and stop

3. **PEEK**: When we need to return the value of the topmost element of the Stack without deleting it from the Stack, the Peek operation is used. This operation first checks if the Stack is empty, i.e., TOP = 0; if it is so, then an appropriate message will display, else the value will return.

**Algorithm:**

Step1 : if (top==0) UNDERFLOW

Step2: out=stack[top-1];

Step4: print out and stop

**3.1.2 SOURCE CODE OF STACK USING ARRAYS**

# include <stdio.h>

int menu()

{

int ch;

printf("MENU STACK\n");

printf("1. PUSH \n");

printf("2. POP \n");

printf("3. DISPLAY \n");

printf("4. CHECK FOR STACK FULL \n");

printf("5. CHECK FOR STACK EMPTY \n");

printf("0. EXIT\n");

printf("Enter Your Option ");

scanf("%d",&ch);

return(ch);

}

void main()

{

int stack[30],top=0,i,nv,op=1,limit;

printf("Ener the capacity of stack ");

scanf("%d",&limit);

while (op!=0)

{

op=menu();

switch(op)

{

case 1:printf("Enter new values");

scanf("%d",&nv);

if (top<limit)

stack[top++]=nv;

break;

case 2: if (top!=0)

printf("The popped value is %d",stack[--top]);

break;

case 3: printf("The stack is as follows ..\n");

for (i=0;i<top;i++)

printf("%d ",stack[i]);

break;

case 4: if (top==limit)

printf("The STACK is FULL ");

else

printf("The STACK IS NOT FULL");

break;

case 5: if (top==0)

printf("THE STACK IS EMPTY");

else

printf("The STACK IS NOT EMPTY");

break;

}

}

}

**3.1.3 SOURCE CODE OF STACK USING LINKED LIST**

# include <stdio.h>

# include <conio.h>

# include <stdlib.h>

typedef struct lstack

{

int data;

struct lstack \*next;

}lstack;

int menu()

{

int ch;

printf("\n M E N U \n");

printf("1: PUSH \n");

printf("2: POP \n");

printf("3: STACK FULL \n");

printf("4: STACK EMPTY \n");

printf("5: DISPLAY \n");

printf("0: Exit \n");

printf("Enter Your Choice \n");

scanf("%d",&ch);

return(ch);

}

lstack \*push(lstack \*start)

{

lstack \*x;

int val;

x=(lstack \*) malloc (sizeof(lstack));

printf("enter new value ");

scanf("%d",&val);

x->data=val;

x->next=start;

start=x;

return(start);

}

void display(lstack \*start)

{

lstack \*y;

y=start;

printf("linked list is \n");

while (y!=NULL)

{

printf("%d ",y->data);

y=y->next;

}

}

lstack \*pop(lstack \*start)

{

lstack \*y;

if (start!=NULL)

{

y=start;

start=y->next;

printf("The deleted one is %d \n",y->data);

free(y);

}

return(start);

}

void main()

{

int k1=1;

lstack \*x,\*y,\*start=NULL;

while (k1!=0)

{

k1=menu();

switch(k1)

{

case 1: start=push(start);break;

case 2: start=pop(start);break;

case 3: printf("No Size Limit ");break;

case 4: if (start==NULL)

printf("Stack is EMPTY ");

break;

case 5: display(start);break;

}

}

}

**3.1.4 APPLICATIONS OF STACK**

A Stack can be used for evaluating expressions consisting of operands and operators.

1. Stacks can be used for Backtracking, i.e., to check parenthesis matching in an expression.
2. It can also be used to convert one form of expression to another form.
3. It can be used for systematic Memory Management.

Advantages of Stack

1. A Stack helps to manage the data in the ‘Last in First out’ method.
2. When the variable is not used outside the function in any program, the Stack can be used.
3. It allows you to control and handle memory allocation and deallocation.
4. It helps to automatically clean up the objects.

Disadvantages of Stack

1. It is difficult in Stack to create many objects as it increases the risk of the Stack overflow.
2. It has very limited memory.
3. In Stack, random access is not possible.

### AlgebraicExpressions:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, \*, /, ^etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefixnotations:

**Infix:** It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the twooperands.

Example: (A + B) \* (C - D)

**Prefix:** It is the form of an arithmetic notation in which we fix (place) the arithmetic operatorbefore(pre)itstwooperands.Theprefixnotationiscalledas polish notation.

Example: \* + A B – C D

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polishnotation*.

Example: A B + C D - \*

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infixexpression.
2. The parentheses are not needed to designate the expression un- ambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: +, -, \*, / and $ or  (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

|  |  |  |
| --- | --- | --- |
| OPERATOR | PRECEDENCE | VALUE |
| Exponentiation ($ or  or ^) | Highest | 3 |
| \*, / | Next highest | 2 |
| +, - | Lowest | 1 |

### Converting expressions usingStack:

Let us convert the expressions from one type to another. These can be done as follows:

1. Infix topostfix
2. Postfix Evaluation

**3.1.5CONVERSIN FROM INFIX TO POSTFIX**

Procedure to convert from infix expression to postfix expression is as follows:

* + - 1. Scan the infix expression from left toright.
      2. a) If the scanned symbol is left parenthesis, push it onto thestack.
         1. If the scanned symbol is an operand, then place directly in the postfix expression(output).
         2. if the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching leftparenthesis.
         3. If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or greater than or equal*) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto thestack.

### Example 1:

Convert ((A – (B + C)) \* D)  (E + F) infix expression to postfix form:

|  |  |  |  |
| --- | --- | --- | --- |
| SYMBOL | POSTFIX STRING | STACK | REMARKS |
| ( |  | ( |  |
| ( |  | ( ( |  |
| A | A | ( ( |  |
| - | A | ( ( - |  |
| ( | A | ( ( - ( |  |
| B | A B | ( ( - ( |  |
| + | A B | ( ( - ( + |  |
| C | A B C | ( ( - ( + |  |
| ) | A B C + | ( ( - |  |
| ) | A B C + - | ( |  |
| \* | A B C + - | ( \* |  |
| D | A B C + - D | ( \* |  |
| ) | 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 + |  |  |
| End of string | A B C + - D \* E F +  | The input is now empty. Pop the output symbols from the stack until it isempty. | |

Program to convert an infix to postfix expression:

#include<stdio.h>

#include<string.h>

int getval(char op)

{

switch(op)

{

case '(': return(0);

case '+':

case '-': return(1);

case '\*':

case '/': return(2);

}

}

main()

{

int k=0, top=0,val1,val2,l,i;

char stack[50],infix[50],postfix[50];

printf("ENTER THE INFIX EXPRESSION \n");

gets(infix);

strcat(infix,")");

l=strlen(infix);

stack[top++]='(';

printf("\n");

for(i=0;i<l;i++)

{

switch(infix[i])

{

case '+':

case '-':

case '\*':

case '/': val1=1,val2=0;

while(val1>=val2)

{

val1=getval(stack[top-1]);

val2=getval(infix[i]);

if(val1>=val2)

postfix[k++]=stack[--top];

}

stack[top++]=infix[i];break;

case '(': stack[top++]='(';

break;

case ')': while(stack[top-1]!='(')

{

postfix[k++]=stack[--top];

} top--;break;

default : postfix[k++]=infix[i];

break;

}

postfix[k]='\0';

printf("the result is %s\n",postfix);

}

}

**2.7.4POSTFIX EVALUATION**

Ex : 2,3,+ The Answer is 5

Rules :

* + - 1. If value , push it in to stack
      2. If operator

Pop call as op1

Pop call as op2

Res = op2 operator op1

Push res

* + - 1. Repeat step1 and 2 until all the elements are over
      2. Result is at stack[0]

Ex: 2,3,+,8,6,-,\*

|  |  |  |
| --- | --- | --- |
| Symbol | Stack | Action |
| 2 | 2 | Push |
| 3 | 2,3 | Push |
| + | 5 | Pop op1=3 , Pop op2=2 , res= 2+3 is 5 PUSH 5 |
| 8 | 5,8 | Push |
| 4 | 5,8,6 | Push |
| - | 5,2 | Pop op1 = 6 pop op2=8 res= 8-6 is 2 push2 |
| \* | 10 | Pop op1=2 pop op2 =5 res = 5 x 2 =10 push 10 |

The result is 10.

Source code :

# include <stdio.h>

main()

{

char postfix[80];

int val=0,l,i,op1,op2,top=0, stack[20],res,d, num=0, unary;

int index;

clrscr();

printf("enter the postfix expression");

gets(postfix);

l=strlen(postfix);

for(i=0;i<l;i++)

{

switch(postfix[i])

{

case ' ' : continue;

case '\*' : op1=stack[--top];

op2=stack[--top];

res=op2\*op1;

stack[top++]=res;

break;

case '+' : op1=stack[--top];

op2=stack[--top];

res=op2+op1;

stack[top++]=res;

break;

case '-' : if ((postfix[i+1]==' ') || (i+1==l))

{

op1=stack[--top];

op2=stack[--top];

res=op2-op1;

stack[top++]=res;

}

else

unary=1;

break;

case '/': op1=stack[--top];

op2=stack[--top];

res=op2/op1;

stack[top++]=res;

break;

default :

num=0;

while(postfix[i]!=' ')

{

d=postfix[i]-'0';

num=num\*10+d;

i++;

}

if(unary==1)

stack[top++]=-num;

else

stack[top++]=num;

unary=0;

break;

}

printf("\nSTACK IS ....");

for (index=0;index<top;index++)

printf(" %d ",stack[index]);

printf("\n");

}

printf("\n");

printf("the result is%d",stack[0]);

}

**Self-AssessmentQuestions:**

1. Choose correct output for the following sequence of operations.

Push(4)

Push(3)

Pop

Push(5)

Push(1)

Push(2)

Pop

Pop

Pop

Pop

1. 4 3 5 1 2
2. 2 1 5 3 4
3. 3 2 1 5 4
4. 3 2 5 1 4
5. Which of the following is the prefix form of A+B\*C?
6. A+(BC\*)
7. +AB\*C
8. ABC+\*
9. +A\*BC
10. Which of the following is not correct statement for stack data structure.
11. Arrays can be used to implement stack
12. Stack follows FIFO
13. Elements are stored in sequential order
14. Top of the stack contains the last inserted element
15. What are the conditions for FULL and EMPTY , if top is initialized to zero , limit is the limit
16. top==limit , top==0
17. top==limit-1 , top==0
18. top==limit, top==-1
19. none

**2.8 QUEUE**

A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a “FIFO” or “First-in-first-out”list.

The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations onqueues:

* + - *enqueue*: which inserts an element at the end of thequeue.
    - *dequeue*: which deletes an element at the start of thequeue.

### Representation ofQueue:

Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

Initially front and rear are set to zero , showing the queue is empty

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| operation | front | rear | queue | After operation | front | rear | queue | Remarks |
| Check for empty | 0 | 0 | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  |  |  |  | | QUEUE is EMPTY | 0 | 0 | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  |  |  |  | |  |
| Insert 45 | 0 | 0 | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  |  |  |  | |  | 0 | 1 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 45 |  |  |  |  | |  |
| Insert 5 | 0 | 1 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 45 |  |  |  |  | |  | 0 | 2 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 45 | 5 |  |  |  | |  |
| Insert 9 | 0 | 2 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 45 | 5 |  |  |  | |  | 0 | 3 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 45 | 5 | 9 |  |  | |  |
| Insert 4 | 0 | 3 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 45 | 5 | 9 |  |  | |  | 0 | 4 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 45 | 5 | 9 | 4 |  | |  |
| Insert 6 | 0 | 3 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 45 | 5 | 9 | 4 |  | |  | 0 | 5 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 45 | 5 | 9 | 4 | 6 | |  |
| Check for full | 0 | 5 |  | QUEUE is FULL As rear reached max |  |  |  |  |
| delete | 0 | 5 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 45 | 5 | 9 | 4 | 6 | |  | 1 | 5 | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 5 | 9 | 4 | 6 | |  |
| delete | 1 | 5 | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 5 | 9 | 4 | 6 | |  | 2 | 5 | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  | 9 | 4 | 6 | |  |
| Insert | 2 | 5 | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 5 | 9 | 4 | 6 | | Not possible as rear reached maximum | 2 | 5 |  |  |

**2.8.2 SOURCE CODE OF QUEUE USING ARRAYS**

# include <stdio.h>

int insert(int que[40],int rear,int limit)

{

int nv;

if (rear==limit)

printf("Queue is Full ");

else

{

printf("Enter new element ");

scanf("%d",&nv);

que[rear++]=nv;

}

return(rear);

}

int dele1(int que[40],int rear,int front)

{

if (rear==front)

printf("Queue is Empty ");

else

{

printf("The deleted Element is %d ",que[front]);

front++;

}

return(front);

}

int menu(void)

{

int op;

printf("\n MENU \n");

printf("1: INSERTION \n");

printf("2: DELETION \n");

printf("3: Display \n");

printf("4: Queue Full \n");

printf("5: Queue Empty \n");

printf("0: Exit \n");

printf("Enter Your option \n");

scanf("%d",&op);

return(op);

}

void display(int que[40],int rear, int front)

{

int i;

printf("\nThe Contents of Queue is as follows ... \n");

for (i=front;i<rear;i++)

printf("%d ",que[i]);

}

void main()

{

int q[40],rear,front,limit,ch=1;

rear=0;

front=0;

printf("Enter the limit of the queue \n");

scanf("%d",&limit);

while (ch!=0)

{

ch=menu();

switch(ch)

{

case 1: rear=insert(q,rear,limit);

break;

case 2: front=dele1(q,rear,front);

break;

case 3: display(q,rear,front);break;

case 4: if (rear==limit)

printf("QUEUE is FULL ");

else

printf("Queue is Not Full ");

break;

case 5: if (rear==front)

printf("QUEUE is EMPTY ");

else

printf("QUEUE is not empty ");

break;

}

}

}

**2.8.3 SOURCE CODE OF QUEUE USING LINKED LISTS**

# include <stdio.h>

# include <conio.h>

# include <stdlib.h>

typedef struct lque

{

int data;

struct lque \*next;

}lque;

int menu()

{

int ch;

printf("\n M E N U \n");

printf("1: Enqueue \n");

printf("2: DelQueue \n");

printf("3: QUEUE FULL \n");

printf("4: QUEUE Empty\n");

printf("5: Display \n");

printf("0: Exit \n");

printf("Enter Your Choice \n");

scanf("%d",&ch);

return(ch);

}

lque \*enque(lque \*qst)

{

lque \*x;

int val;

x=(lque \*) malloc (sizeof(lque));

printf("enter new value ");

scanf("%d",&val);

x->data=val;

x->next=qst;

qst=x;

return(qst);

}

void display(lque \*qst)

{

lque \*y;

y=qst;

printf("linked list is \n");

while (y!=NULL)

{

printf("%d ",y->data);

y=y->next;

}

}

lque \*delque(lque \*qst)

{

lque \*z,\*y;

// name the last node as y

// name the last but one node as z

// z->next=NULL AND then free y

// name the last node as y

if (qst==NULL)

printf("Deletion is not possible");

else

{

//name the last node as y

y=qst;

while (y->next!=NULL)

y=y->next;

// name the last but one node as z

z=qst;

while (z->next!=y)

z=z->next;

z->next=NULL;

printf("the deleted value is %d ",y->data);

free(y);

}

return(qst);

}

void main()

{

int k1=1;

lque \*x,\*y,\*qst=NULL;

while (k1!=0)

{

k1=menu();

switch(k1)

{

case 1: qst=enque(qst);break;

case 2: qst=delque(qst);break;

case 3: printf("No Size so NO Full");break;

case 4: if (qst==NULL)

printf("Queue is Empty");break;

case 5: display(qst);break;

}

}

}

**2.8.4 CIRCULAR QUEUE**

The problem with queues is once if rear reaches maximum , no new element can be inserted even if an element is deleted . To remove this hiccup , circular queues are used. The idea behind this is modulo incrimination. Incrimination is done circularly.

| **S.no.** | **Linear Queue** | **Circular Queue** |
| --- | --- | --- |
| **1.** | Arranges the data in a linear pattern. | Arranges the data in a circular order where the rear end is connected with the front end. |
| **2.** | The insertion and deletion operations are fixed i.e, done at the rear and front end respectively. | Insertion and deletion are not fixed and it can be done in any position. |
| **3.** | It is inefficient in comparison to a circular queue. | It is more efficient in comparison to linear queue. |
| **4.** | If there are 10 spaces then in the best case all the 10 spaces in the queue can be filled | If there are 10 spaces then in the best case 9 spaces can be filled at a time |
| **5.** | Good for applications where overflow is not a concern. | Good for applications where efficient use of memory is important. |
| **6** | Once if rear reaches limit, no element can be inserted , even if space is available | Problem of queue is rectified |

# include <stdio.h>

int insert(int cque[40],int rear,int front,int limit)

{

int nv;

if ((rear+1)%(limit+1)==front)

printf("Circular Queue is Full ");

else

{

printf("Enter new element ");

scanf("%d",&nv);

cque[rear]=nv;

rear=(rear+1)%(limit+1);

}

return(rear);

}

int dele1(int que[40],int rear,int front,int limit)

{

if (rear==front)

printf("Circular Queue is Empty ");

else

{

printf("The deleted Element is %d ",que[front]);

front=(front+1)%(limit+1);

}

return(front);

}

int menu(void)

{

int op;

printf("\n Circular Queue MENU \n");

printf("1: INSERTION \n");

printf("2: DELETION \n");

printf("3: Display \n");

printf("4: Circular Queue Full \n");

printf("5: Circular Queue Empty \n");

printf("0: Exit \n");

printf("Enter Your option \n");

scanf("%d",&op);

return(op);

}

void display(int que[40],int rear, int front,int limit)

{

int i;

printf("\nThe Contents of Circular Queue is as follows ... \n");

for (i=front;i!=rear;i=(i+1)%(limit+1))

printf("%d ",que[i]);

}

void main()

{

int q[40],rear,front,limit,ch=1;

rear=0;

front=0;

printf("Enter the limit of the queue \n");

scanf("%d",&limit);

while (ch!=0)

{

ch=menu();

switch(ch)

{

case 1: rear=insert(q,rear,front,limit);

break;

case 2: front=dele1(q,rear,front,limit);

break;

case 3: display(q,rear,front,limit);break;

case 4: if ((rear+1)%(limit+1)==front)

printf("Circular QUEUE is FULL ");

else

printf("Circular Queue is Not Full ");

break;

case 5: if (rear==front)

printf("Circular QUEUE is EMPTY ");

else

printf("Circular QUEUE is not empty ");

break;

}

}

}

**2.8.5 DOUBLE ENDED QUEUE**

Double Ended queue (dequeue) is a queue which has two ends, insertion can take place from both ends similarly deletions also can take place from both ends. Input restricted dequeue , Output restricted Dequeue.

**2.8.6 PRIORITY QUEUE**

# include <stdio.h>

typedef struct pqueue

{

int data;

int prior;

}prq;

int insert(prq pr[30],int n)

{

int newpr,newdat,i,k;

printf("ENTER NEW DATA AND PRIORITY \n");

scanf("%d%d",&newdat,&newpr);

if (n!=0)

{

for (i=0;i<n;i++)

{

if (newpr < pr[i].prior)

break;

}

for (k=n;k>i;k--)

pr[k]=pr[k-1];

pr[i].data=newdat;

pr[i].prior=newpr;

}

else

{

pr[0].data=newdat;

pr[0].prior=newpr;

}

return(n+1);

}

int delete(prq pr[30],int n)

{

int k;

if (n!=0)

{

printf("THE DELETED ELEMENT IS %d %d ",pr[0].data,pr[0].prior);

for (k=0;k<n;k++)

pr[k]=pr[k+1];

--n;

}

return(n);

}

display(prq pr[30],int n)

{

int i;

printf("\nTHE PRIORITY QUEUE IS ....\n");

printf("DATA(PRIORITY)...\n");

for (i=0;i<n;i++)

printf("%3d(%d)--- ",pr[i].data,pr[i].prior);

printf("\n");

}

main()

{

int n=0,ch;

prq pr[30];

for (;;)

{

printf("1. INSERTION \n");

printf("2. DELETION \n");

printf("3. Exit \n");

printf("ENTER UR CHOICE \n");

scanf("%d",&ch);

if (ch==3) break;

switch(ch)

{

case 1: n=insert(pr,n);break;

case 2: n=delete(pr,n);break;

}

display(pr,n);

}

}

**Self-AssessmentQuestions:**

1. A data structure where elements can be added or removed at either end but not in the middle is called
2. Stack
3. Queue
4. Linked List
5. Dqueue
6. A Circular queue of (n-1) elements is implemented with array of n elements. Assume that REAR and FRONT are used as indices to carry out INSERTION and DELETION operations. Choose the correct options for CQFull and CQEmpty.
7. (REAR+1) mod n == front and front==0
8. REAR mod n == front and front==0
9. (REAR+1) mod n == front and front==rear
10. REAR mod n == front and front==rear
11. If the elements “A”, “B”, “C” and “D” are placed in a queue and are deleted one at a time, in what order will they be removed?
12. DCBA
13. BACD
14. ABCD
15. CABD
16. Which one of the following is not the application of the Queue data structure?
17. Resource shared between various systems
18. Data is transferred asynchronously
19. Load Balancing
20. Balancing of Symbols

|  |  |
| --- | --- |
| Question | Answer |
| 1 | A |
| 2 | D |
| 3 | C |
| 4 | C |
| 5 | D |
| 6 | C |
| 7 | D |
| 8 | A |
| 9 | A |
| 10 | B |
| 11 | B |
| 12 | A |
| 13 | A |
| 14 | A |
| 15 | C |
| 16 | A |
| 17 | C |
| 18 | D |
| 19 | A |
| 20 | A |
| 21 | B |
| 22 | C |
| 23 | C |
| 24 | D |

**Summary:**

A linked list is a fundamental data structure used in computer science and programming. It consists of a sequence of elements called nodes, where each node contains data and a reference (or link) to the next node in the sequence. Here are some key summary points on linked lists:

1. Nodes: The basic building blocks of a linked list are nodes. Each node contains two fields:
   * Data: The actual value or information stored in the node.
   * Next: A reference (or pointer) to the next node in the sequence.
2. Head: The first node in the linked list is called the "head." It acts as the starting point to traverse the list.
3. Tail: The last node in the linked list is called the "tail." It points to NULL, indicating the end of the list.
4. Singly Linked List: Each node in a singly linked list has a single reference to the next node. Traversal is unidirectional, starting from the head and progressing to the tail.
5. Doubly Linked List: Each node in a doubly linked list has two references: one to the next node and another to the previous node. Traversal can be bidirectional.
6. Circular Linked List: In this type of linked list, the tail node points back to the head, creating a circular structure. It has no NULL termination.
7. Dynamic Size: Linked lists can grow or shrink dynamically as elements are added or removed, as opposed to arrays that have a fixed size.
8. Insertion: Adding a new node to a linked list involves adjusting the references of the neighboring nodes. It can be done at the beginning (prepend), end (append), or in between (insert).
9. Deletion: Removing a node from a linked list requires reconfiguring the references of its neighboring nodes. Once the links are adjusted, the removed node is deallocated from memory.
10. Searching: To find a specific value in a linked list, you start from the head and traverse through the nodes until the desired value is found or the end of the list is reached.
11. Time Complexity:

* Insertion and Deletion: O(1) if the position is known; O(n) if the position needs to be searched.
* Searching: O(n) as it may require traversing the entire list.

1. Memory Efficiency: Linked lists use memory efficiently because they only allocate memory for each node as it is needed.
2. Memory Overhead: Linked lists have a higher memory overhead than arrays due to the additional memory required for storing the references.
3. Usage: Linked lists are commonly used in various applications, such as implementing stacks, queues, hash tables, and dynamic data structures.
4. Advantages:

* Dynamic size allows efficient memory utilization.
* Insertion and deletion at any position are relatively fast.
* Easy to implement and manage.

1. Disadvantages:

* Slower access time compared to arrays (no direct indexing).
* More memory overhead due to storing pointers/references.
* Extra care is required to handle pointers correctly to avoid memory leaks or segmentation faults.

Terminal Questions

1. What is a linked list, and how does it differ from an array?
2. How do you traverse a linked list from the beginning to the end?
3. How do you insert a new node at the beginning of a linked list ?
4. How do you insert a new node at the end of a linked list ?
5. How do you insert a new node after a specific node in a linked list?
6. Explain the process of deleting a node from a singly linked list.
7. How do you delete the first node from a linked list?
8. How do you delete the last node from a linked list?
9. Can you delete a specific node with a given value from a linked list?
10. How do you reverse a linked list?

**Bibliography:**

1. "Data Structures and Algorithm Analysis in C++" by Mark Allen Weiss - This book covers various data structures, including linked lists, and provides implementation details using C++.
2. "Data Structures and Algorithms Made Easy" by Narasimha Karumanchi - This book focuses on data structures and algorithms with sample problems, including linked list-based exercises.
3. "The Algorithm Design Manual" by Steven S. Skiena - This book covers various algorithms and data structures, including linked lists, with an emphasis on practical implementation.
4. "Algorithms" by Robert Sedgewick and Kevin Wayne - This book provides a detailed exploration of algorithms and data structures, including linked lists, using a modern approach.

**e-References:**

1. <https://www.geeksforgeeks.org/linked-list/>
2. <https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithm.htm>
3. <https://www.programiz.com/dsa/linked-list>