Stack and Queue

There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of such data structures that are useful are:

- *Stack.*
- *Queue.*

Linear lists and arrays allow one to insert and delete elements at anyplace in the list i.e., at the beginning, at the end or in the middle.

• STACK:

A stack is a list of elements in which an element may be inserted or deleted only at oneend, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out)lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

Push: is the term used to insert an element into a stack.

Pop: is the term used to delete an element from a stack.

―Push‖ is the term used to insert an element into a stack. ―Pop‖ is the term used to delete an element from the stack.

All insertions and deletions take place at the same end, so the last element added tothe stack will be the first element removed from the stack. When a stack is created, thestack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element isthe bottom of the stack.

• Representation of Stack:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack.If we attempt to add new element beyond the maximum size, we will encounter a *stackoverflow* condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a *stack underflow* condition.

When an element is added to a stack, the operation is performed by push(). Figure 4.1 shows the creation of a stack and addition of elements using push().

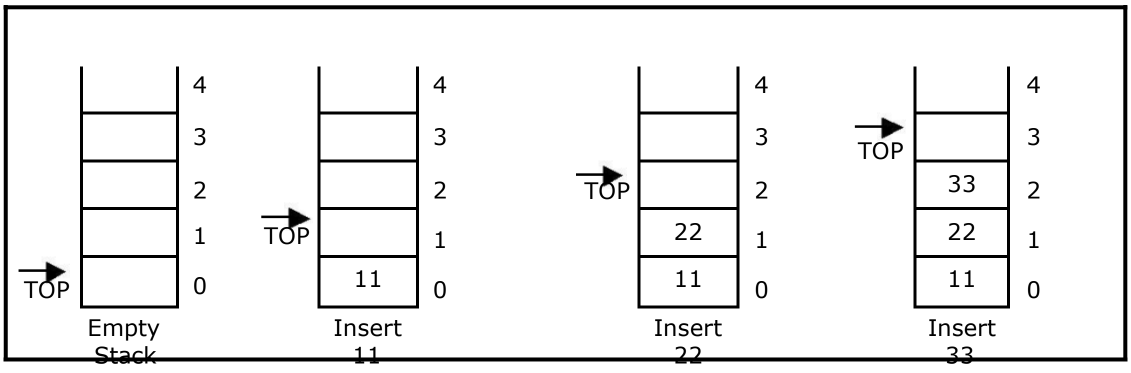


Figure 4.1. Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). Figure 4.2 shows a stack initially with three elements and shows the deletion of elements using pop().

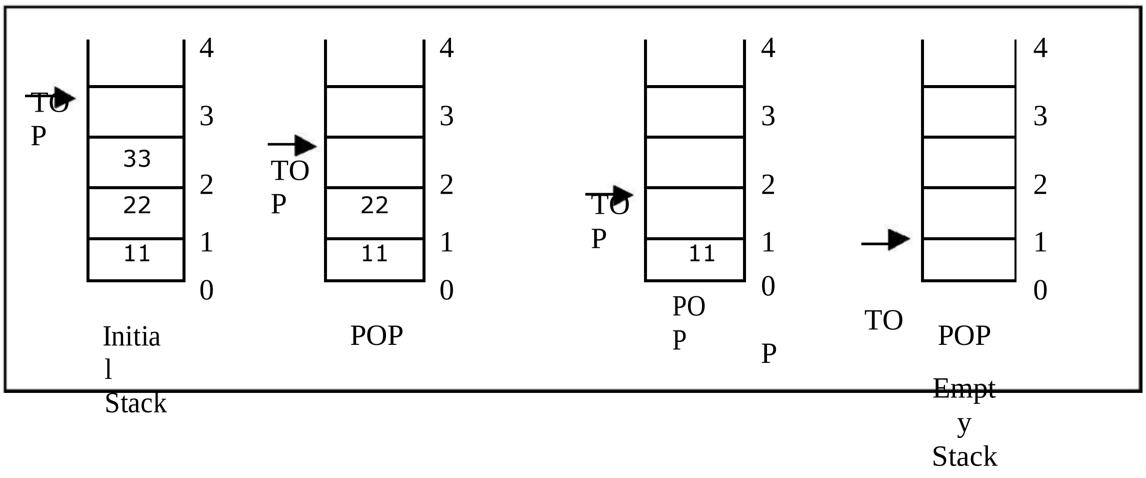


Figure 4.2. Pop operations on stack

Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using *top* pointer. The linked stack looks as shown in figure 4.3.

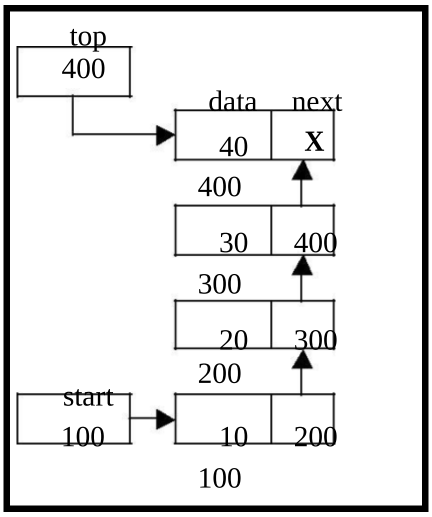


Figure 4.3. Linked stackrepresentation

Source code for stack operations, using linked list:

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

struct stack
{
        int data;
        struct stack *next;
};


void push();
void pop();
void display();
typedef struct stack node;
node *start=NULL;
node *top = NULL;

node* getnode()
{
        node *temp;
        temp=(node  *)  malloc(  sizeof(node))  ;
        printf("\n Enter data ");
        scanf("%d",  &temp  ->  data);
        temp -> next = NULL; return
        temp;
}
void push(node *newnode)
{
        node *temp;
        if( newnode == NULL )
        {
                printf("\n Stack Overflow..");
                return;
        }
```

```
        if(start == NULL)
        {
                start = newnode;
                top = newnode;
        }
        else
        {
                temp = start;
                while( temp -> next !=
                        NULL) temp = temp
                        -> next;
                temp -> next = newnode;
        }
                top = newnode;
        printf("\n\n\t Data pushed into stack");
}
void pop()
{
        node *temp;
        if(top ==
        NULL)
        {
                printf("\n\n\t Stack
                underflow"); return;
        }
        temp = start;
        if( start -> next == NULL)
        {
                printf("\n\n\t Popped element is %d ", top -> data);
                start = NULL;
                free(top);
        }
                top =
        else
                NULL;
        {

                while(temp -> next != top)
                {
                        temp = temp -> next;
                }
                temp -> next = NULL;
                printf("\n\n\t Popped element is %d ", top -> data);
                free(top);
                top = temp;
        }
}
void display()
{
        node *temp;
        if(top ==
        NULL)
        {
                printf("\n\n\t\t Stack is empty ");
        }
        else
        {
                temp = start;
                printf("\n\n\n\t\t Elements in the stack: \n");
                printf("%5d ", temp -> data);
                while(temp != top)
                {
                        temp = temp -> next;
                        printf("%5d ", temp -> data);
                }
        }}
```

```
char menu()
{
        char ch;
        clrscr();
        printf("\n \tStack operations using pointers.. ");
        printf("\n -----------********** --------------- \n");
        printf("\n 1. Push ");
        printf("\n 2. Pop ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice:
        "); ch = getche();
        return ch;
}

void main()
{
        char ch;
        node *newnode;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case '1' :
                                newnode = getnode();
                                push(newnode); break;

                        case '2' :
                                pop();
                                break;
                        case '3' :
                                display();
                                break;
                        case '4':
                                return;
                }

                getch();
        } while( ch != '4' );
}
```

Algebraic Expressions:

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 prefix notations:

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

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

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmeticoperator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in theyear 1920).

Example: * + A B – C D

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

Example: A B + C D - *

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.

2. The parentheses are not needed to designate the expression un-ambiguously.

3. While evaluating the postfix expression the priority of the operators is nolonger relevant.

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

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

Conversion from infix to postfix:

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

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.

   b) If the scanned symbol is an operand, then place directly in the postfixexpression (output).

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 left parenthesis.

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 the stack.

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 is empty.	

Example 2:

Convert a + b * c + (d * e + f) * g the infix expression into postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
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	+ *	
End of String	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	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 /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 4:

Convert the following infix expression A + (B * C – (D / E ↑ F) * G) * H into itsequivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
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	+ *	
End of String	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

1. Program to convert an infix to postfix expression:

```
# include <string.h>

char postfix[50];
char infix[50];
char opstack[50]; /* operator stack */ int i, j, top =0;

int lesspriority(char op, char op_at_stack)
  {
        int k;
        int pv1;                    /* priority value of op */
        int pv2;                    /* priority value of op_at_stack */
        char operators[] = {'+', '-', '*', '/', '%', '^', '(' };
        int priority_value[] = {0,0,1,1,2,3,4};if(
        op_at_stack == '(' )
                return 0;
        for(k = 0; k < 6; k ++)
        {
                if(op == operators[k])
                        pv1 = priority_value[k];
        }
        for(k = 0; k < 6; k ++)
        {
                if(op_at_stack == operators[k])
                        pv2 = priority_value[k];
        }
        if(pv1 < pv2)
                return 1;
        else
                return 0;
}
```

```
void push(char op)           /* op - operator     */
{
          if(top == 0)
          {
                    opstack[top] = op;
                    top++;
          }
          else
          {
                    if(op != '(' )
                    {
                              while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
                              {
                                                  postfix[j] = opstack[--
                                                  top]; j++;
                              }
                    }
                    opstack[top] = op; /* pushing onto stack */
                    top++;
          }
}
```

```
/* before pushing the operator
'op' into the stack check priority
of op with top of opstack if less
then pop the operator from stack
then push into postfix string else
push op onto stack itself */
```

```
pop()
{         while(opstack[--top] != '(' )                    /* pop until '(' comes  */
          {
                    postfix[j] = opstack[top];
                    j++;
          }
}

void main()
{
          char ch;
          clrscr();
          printf("\n Enter Infix Expression : ");
          gets(infix);
          while( (ch=infix[i++]) != ‗\0‘)
          {
                    switch(ch)
                    {
                              case ' ' : break;
                              case '(' :
                              case  '+'  :
                              case  '-'  :
                              case  '*'  :
                              case  '/'  :
                              case  '^'  :
                              case '%' :
                                        push(ch); /* check priority and push */ break;

                              case ')' :
                                        pop();
                                        break;
                              default :
                                        postfix[j] = ch;
                                        j++;
                    }
          }
          while(top >= 0)
          {
                    postfix[j] = opstack[--top];
                    j++;
          }                       ntf("\n Infix Expression : %s
          post                    ", infix); printf("\n Postfix
          fix[j                   Expression : %s ", postfix);
          ] =                     getch();
          '\0';           }
          p
          r
          i
                    );
```

4.4. Evaluation of postfix expression:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a _+‘ is read, so 3 and 2 are popped from the stack and their sum 5, is pushed

8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a _*‘ is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a _+‘ is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, _+‘ pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a _*‘ is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example 2:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Program to evaluate a postfix expression:

```
2  include <conio.h>
3  include <math.h>
4  define MAX 20

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

```
void main(void)
{
        char
        postfix[MAX];int
        val;
        char ch;
        int i = 0, top = 0;
        float val_stack[MAX], val1,
        val2,res; clrscr();
        printf("\n Enter a postfix expression: ");
        scanf("%s", postfix);
        while((ch = postfix[i]) != '\0')
        {
                if(isoperator(ch) == 1)
                {
                        val2    =    val_stack[--
                        top]; val1 = val_stack[-
                        -top]; switch(ch)
                        {
                                case '+':
                                        res = val1 +
                                        val2; break;
                                case '-':
                                        res = val1 - val2;
                                        break;
                                case '*':
                                        res = val1 * val2;
                                        break;
                                case '/':
                                        res = val1 / val2;
                                        break;
                                case '^':
                                        res = pow(val1, val2);
                                        break;
                        }
                        val_stack[top] = res;
                }
                else
                        val_stack[top] = ch-48; /*convert character digit to integer digit */

                top++;
                i++;
        }

        printf("\n Values of %s is : %f ",postfix, val_stack[0] );
        getch();
}
```

Applications of stacks:

1. Stack is used by compilers to check for balancing of parentheses, bracketsand braces.

2. Stack is used to evaluate a postfix expression.

3. Stack is used to convert an infix expression into postfix/prefix form.

4. In recursion, all intermediate arguments and return values are stored on theprocessor‘s stack.

5. During a function call the return address and arguments are pushed onto astack and on return they are popped off.

Queue:

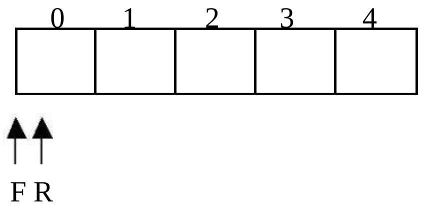
A queue is another special kind of list, where items are inserted at one end called therear and deleted at the other end called the front. Another name for a queue is a ―FIFO‖ or ―First-in-first-out‖ list.

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

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

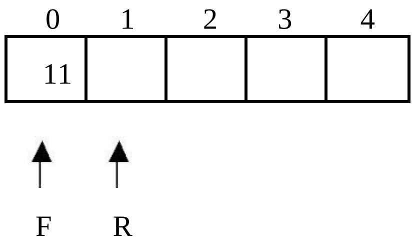
Representation of Queue:

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



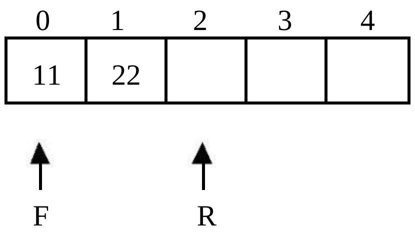
Que u e E mpt y
F RO NT = REA R = 0

Now, insert 11 to the queue. Then queue status will be:



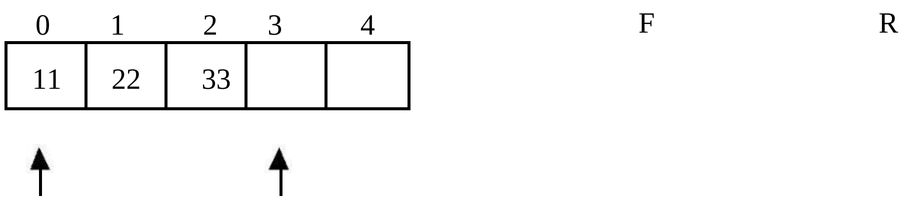
REA R = REA R + 1
= 1F RO NT = 0

Next, insert 22 to the queue. Then the queue status is:



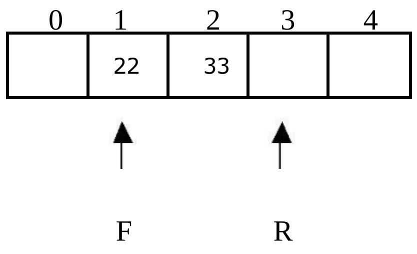
REA R = REA R + 1
= 2F RO NT = 0

Again insert another element 33 to the queue. The status of the queue is:



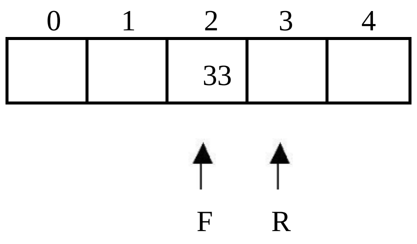
REA R = REA
R + 1 = 3F RO
NT = 0

Now, delete an element. The element deleted is the element at the front of the queue.So the status of the queue is:



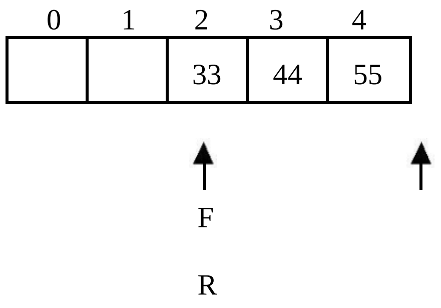
REA R = 3
F RO NT = F R O NT + 1 = 1

Again, delete an element. The element to be deleted is always pointed to by the FRONTpointer. So, 22 is deleted. The queue status is as follows:



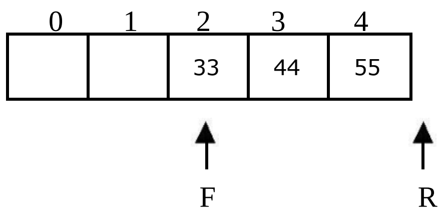
REA R = 3
F RO NT = F R O NT + 1 = 2

Now, insert new elements 44 and 55 into the queue. The queue status is:



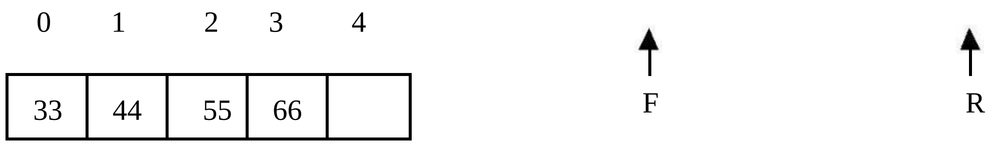
REA R = 5
F RO NT = 2

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



REA R = 5
F RO NT = 2

Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To over come this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



REA R = 4
F RO NT = 0

This difficulty can overcome if we treat queue position with index 0 as a position thatcomes after position with index 4 i.e., we treat the queue as a **circular queue.**

Source code for Queue operations using array:

In order to create a queue we require a one dimensional array Q(1:n) and two variables *front* and *rear*. The conventions we shall adopt for these two variables arethat *front* is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus, front = rear if and only if there are no elements in the queue. The initial condition then is front = rear = 0. The various queue operationsto perform creation, deletion and display the elements in a queue are as follows:

insertQ(): inserts an element at the end of queue Q.

deleteQ(): deletes the first element of Q. displayQ():

displays the elements in the queue.

```
8. include <conio.h>
9. define MAX
6int Q[MAX];
int front, rear;

void insertQ()
{
        int data;
        if(rear == MAX)
        {
                printf("\n Linear Queue is full");
                return;
        }
        else
        {
                printf("\n Enter data: ");
                scanf("%d", &data);
                Q[rear] = data; rear++;

        }       printf("\n Data Inserted in the Queue ");
}

void deleteQ()
{
        if(rear == front)
        {
                printf("\n\n Queue is Empty..");
                return;
        }
        else
        {
                printf("\n Deleted element from Queue is %d",Q[front]);
                front++;

        }
}

void displayQ()
{
        int i;
        if(front == rear)
        {
                printf("\n\n\t Queue is Empty");
                return;
        }
        else
        {
                printf("\n Elements in Queue are:
                "); for(i = front; i < rear; i++)
                {
                        printf("%d\t", Q[i]);
                }
        }
}
int menu()
{
        int ch;
        clrscr();
        printf("\n \tQueue operations using ARRAY..");
        printf("\n -----------**********-----------------\n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice:
        "); scanf("%d",  &ch); return
        ch;
}
void main()
{
        int ch;
        do
        {       ch = menu();
                switch(ch)
                {
                        case 1:
                                insertQ();
                                break;
                        case 2:
                                deleteQ();
                                break;
                        case 3:
                                displayQ();
                                break;
                        case 4:
                                return;
                }
                getch();
        } while(1);
}
```

Linked List Implementation of Queue:

We can represent a queue as a linked list. In a queue data is deleted from the front endand inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers *front* and *rear* for our linked queue implementation.

The linked queue looks as shown in figure 4.4:

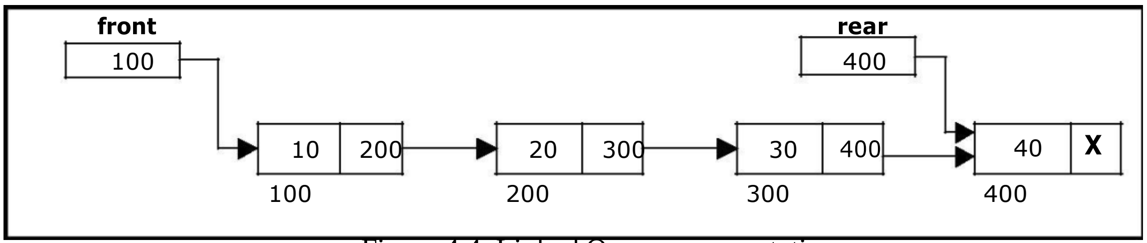


Figure 4.4. Linked Queue representation

Source code for queue operations using linked list:

```
    include <stdlib.h>
    include <conio.h>

struct queue
{
            int data;
            struct queue *next;
};
typedef struct  queue node;
node *front = NULL;node
*rear = NULL;

node* getnode()
{
            node *temp;
            temp   =   (node   *)   malloc(sizeof(node))   ;
            printf("\n Enter data ");
            scanf("%d", &temp -> data);temp
            -> next = NULL; returntemp;
}
void insertQ()
{
            node *newnode; newnode
            = getnode();if(newnode
            == NULL)
            {
                        printf("\n Queue Full");
                        return;
            }
            if(front == NULL)
            {
                        front = newnode;
                        rear = newnode;
            }
            else
            {
                        rear -> next = newnode;
                        rear = newnode;
            }

            printf("\n\n\t Data Inserted into the Queue..");
}
void deleteQ()
{
            node *temp;
            if(front == NULL)
            {
                        printf("\n\n\t Empty Queue..");
                        return;
            }
            temp = front;
            front = front -> next;
            printf("\n\n\t Deleted element from queue is %d ", temp ->data);
}
```

```
free(temp);
            void displayQ()
            {
                        node *temp;
                        if(front == NULL)
                        {
                                    printf("\n\n\t\t Empty Queue ");
                        }
                        else
                        {
                                    temp = front;
                                    printf("\n\n\n\t\t Elements in the Queue are: ");
                                    while(temp != NULL )
                                    {
                                                printf("%5d ", temp -> data);temp
                                                = temp -> next;
                                    }
                        }
            }

            char menu()
            {
                        char ch;
                        clrscr();
                        printf("\n \t..Queue operations using pointers.. ");
                        printf("\n\t            -----------**********-------------
                        \n"); printf("\n 1. Insert ");
                        printf("\n 2. Delete ");
                        printf("\n 3. Display");
                        printf("\n 4. Quit ");
                        printf("\n Enter your choice: ");ch =
                        getche();
                        return ch;
            }

            void main()
            {
                        char ch;
                        do
                        {
                                    ch = menu();
                                    switch(ch)
                                    {
                                                case '1' :
                                                            insertQ();
                                                            break;
                                                case '2' :
                                                            deleteQ();
                                                            break;
                                                case '3' :
                                                            displayQ();
                                                            break;
                                                case '4':
                                                            return;
            }
```

```
                    }
                    getch();
          } while(ch != '4');}
```

```
}
```

Applications of Queue:

1. It is used to schedule the jobs to be processed by the CPU.

2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first infirst out (FIFO) basis.

3. Breadth first search uses a queue data structure to find an element from a graph.

Circular Queue:

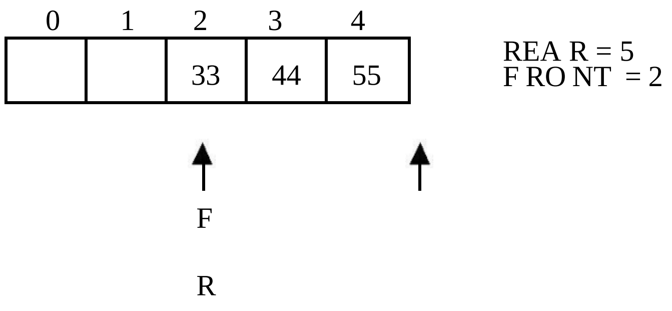
A more efficient queue representation is obtained by regarding the array Q[MAX] as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

There are two problems associated with linear queue. They are:

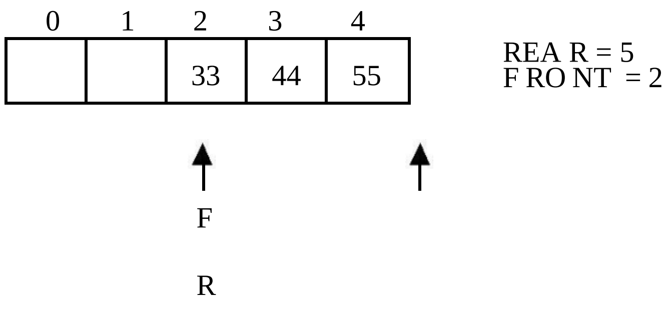
- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.

- Signaling queue full: even if the queue is having vacant position.

For example, let us consider a linear queue status as follows:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue fullsignal. The queue status is as follows:

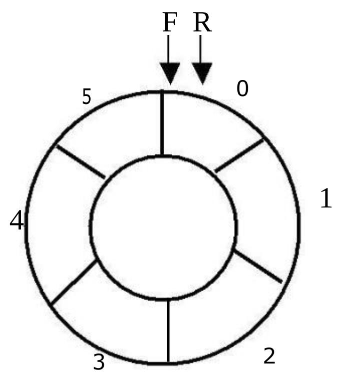


This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a **circular queue.**

In circular queue if we reach the end for inserting elements to it, it is possible to insertnew elements if the slots at the beginning of the circular queue are empty.

Representation of Circular Queue:

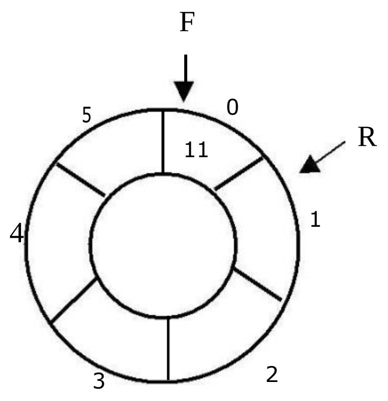
Let us consider a circular queue, which can hold maximum (MAX) of six elements.Initially the queue is empty.



Que u e E mpt y
M A X = 6
F RO NT = REA R =
0CO U NT = 0

Circ ular Que
ue

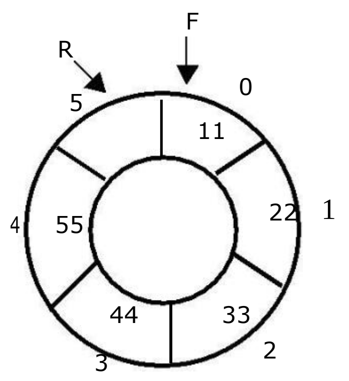
Now, insert 11 to the circular queue. Then circular queue status will be:



F RO NT = 0
REA R = (REA R + 1) % 6 = 1
CO U NT = 1

Circ ular Que ue

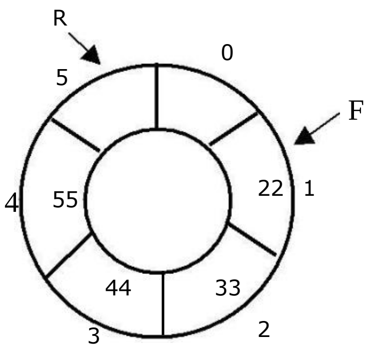
Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queuestatus is:



FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

Circular Queue

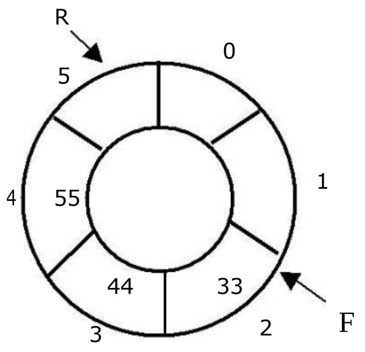
Now, delete an element. The element deleted is the element at the front of the circularqueue. So, 11 is deleted. The circular queue status is as follows:



F RO NT = (F R O NT + 1) % 6 = 1
REA R = 5
CO U NT = CO U NT - 1 = 4

Circ ular Que ue

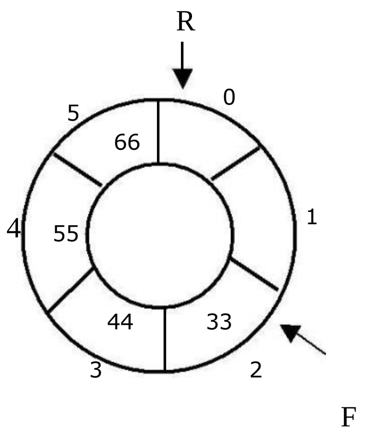
Again, delete an element. The element to be deleted is always pointed to by the FRONTpointer. So, 22 is deleted. The circular queue status is as follows:



F RO NT = (F R O NT + 1) % 6
= 2REA R = 5
CO U NT = CO U NT - 1 = 3

Circ ular Que ue

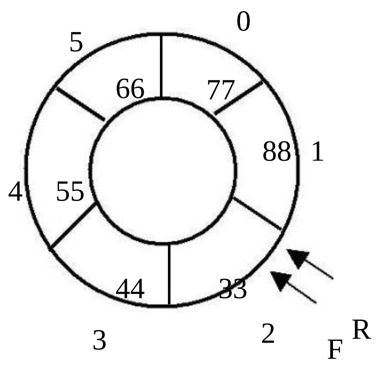
Again, insert another element 66 to the circular queue. The status of the circular queueis:



F RO NT = 2
REA R = (REA R + 1) % 6
= 0CO U NT = CO U NT +
1 = 4

Circ ular Que ue

Now, insert new elements 77 and 88 into the circular queue. The circular queue statusis:



F RO NT = 2, REA R =
2REA R = REA R % 6
= 2 CO U NT = 6

Circ ular Que ue

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add theelement to circular queue. So, the circular queue is *full*.

Source code for Circular Queue operations, using array:

```
# include  <stdio.h>#
include   <conio.h> #
define MAX 6

int CQ[MAX];
int front = 0;
int rear = 0;
int count = 0;

void insertCQ()
{
        int data;
        if(count == MAX)
        {
                printf("\n Circular Queue is Full");
        }
        else
        {       printf("\n Enter data: ");
                scanf("%d", &data);
                CQ[rear] = data;
                rear = (rear + 1) % MAX;
                count ++;
                printf("\n Data Inserted in the Circular Queue ");
        }
}



void deleteCQ()
{
        if(count == 0)
        {
                }
        }
        else
        {


        }
```

```
printf("\n\n        ront]);front = (front + 1) % MAX;
Circular            count --;
Queue is
Empty..");
```

```
p
r
i
n
t
f
(
"
\
n
D
e
l
e
t
e
d
e
l
e
m
e
n
t
f
r
o
m
C
i
r
c
u
l
a
r
Q
u
e
u
e
i
s
%
d
"
,
C
Q
[
f
```

```
void displayCQ()
{
        int i, j;
        if(count == 0)
        {
                printf("\n\n\t Circular Queue is Empty ");
        }
        else
        {       printf("\n Elements in Circular Queue are: ");j
                = count;
                for(i = front; j != 0; j--)
                {
                        printf("%d\t",
                        CQ[i]);i = (i + 1) %
                        MAX;
                }
        }
}

int menu()
{
        int ch;
        clrscr();
        printf("\n \t Circular Queue Operations using ARRAY..");
        printf("\n -----------**********----------------\n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter Your Choice:
        "); scanf("%d", &ch);
        return ch;
}

void main()
{
        int ch;
        do
        {       ch = menu();
                switch(ch)
                {
                        case 1:
                                insertCQ();
                                break;
                        case 2:
                                deleteCQ();
                                break;
                        case 3:
                                displayCQ();
                                break;
                        case 4:
                                return;
                        default:
                                printf("\n Invalid Choice ");
                }
                getch();
        } while(1);
}
```

Deque:

In the preceding section we saw that a queue in which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a *deque*. The word *deque* is an acronym derived from *double-ended queue*. Figure 4.5 shows the representation of a deque.

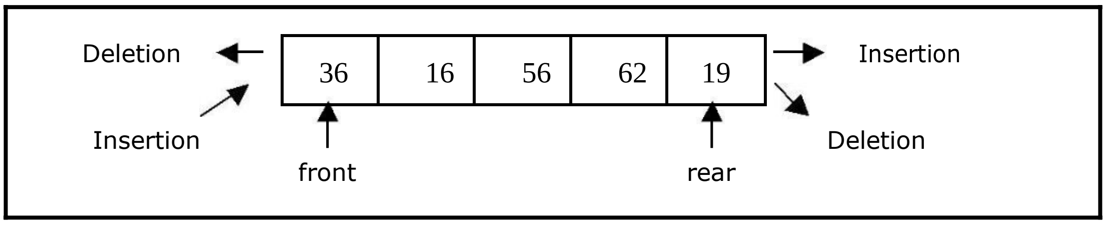


Figure 4.5. Representation of a deque.

A deque provides four operations. Figure 4.6 shows the basic operations on a deque.

- enqueue_front: insert an element at front.
- dequeue_front: delete an element at front.
- enqueue_rear: insert element at rear.
- dequeue_rear: delete element at rear.

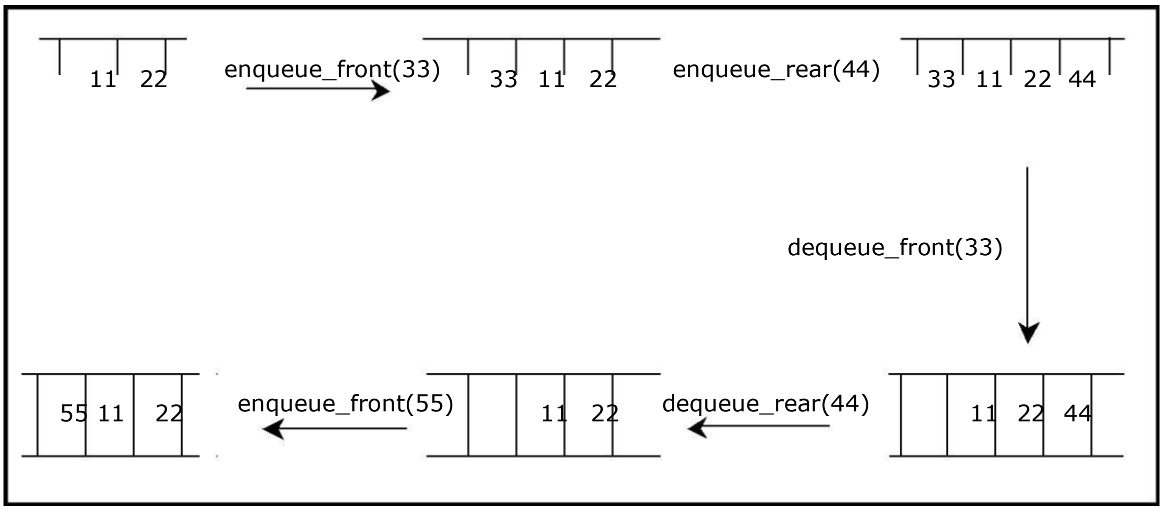


Figure 4.6. Basic operations on deque

There are two variations of deque. They are:

- Input restricted deque (IRD)
- Output restricted deque (ORD)

An Input restricted deque is a deque, which allows insertions at one end but allowsdeletions at both ends of the list.

An output restricted deque is a deque, which allows deletions at one end butallows insertions at both ends of the list.

Priority Queue:

A priority queue is a collection of elements such that each element has been assigned apriority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lowerpriority.
2. two elements with same priority are processed according to the order inwhich they were added to the queue.

A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort.

Multiple Queue:

. Multi queue is data structure in which multiple queues are maintained. This type of data structures are utilized for process scheduling. We might use one dimensional array or multidimensional array to illustrated a multiple queue.

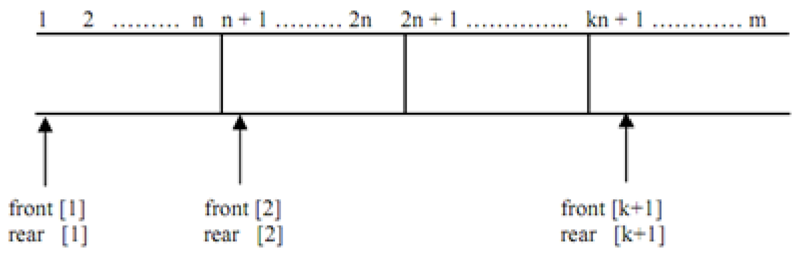

1 2 .... n n + 1 .... 2n 2n + 1 .... kn + 1 .... m
front [1]
rear [1]
front [2]
rear [2]
front [k+1]
rear [k+1]

Figure: Multiple queues in an array

A multi queue implementation by using a single dimensional array along m elements is illustrated in Figure 6. Each of queues contains n elements that are mapped to a liner array of m elements.

C program to add and delete elements from multiple queues.

Solution:

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 20
int insq( int queue[MAX], int qno, int rear[], int limit[], int *ele)
{
   if( rear[qno] == limit[qno] )
      return(-1);
   else
   {
      rear[qno]++; //... rear[qno] = rear[qno] + 1;
      queue[ rear[qno] ] = *ele;
```

```
        return(1);
    }
}
int delq( int queue[MAX], int qno, int front[], int rear[], int *ele)
{
    if( front[qno] == rear[qno] )
        return(-1);
    else
    {
        front[qno]++; //... front[qno] = front[qno] + 1;
        *ele = queue[ front[qno] ];
        return(1);
    }
}
int main()
{
    int queue[MAX],ele;
    int bott[10], limit[10], f[10], r[10];
    int i, n,qno,size,ch,reply;
    printf("How many queues you want to enter? : ");
    scanf("%d", &n);
    size = MAX / n; //Get maximum size for each queue
    //Initialize bottom for each Queue
    bott[0] = -1; //Bottom of first queue is -1
    for(i = 1; i < n; i++)
        bott[i] = bott[i-1] + size; //Initialize limit of each queue
    for(i = 0; i < n; i++)
    {
        limit[i] = bott[i] + size; //Initialize Front & Rear of each Queue
    }
    //Initial value of Front & Rear of each queue is same as its Bottom Value
    for(i = 0; i < n; i++)
        f[i] = r[i] = bott[i];
    printf("------------------------------------\n");
    printf("\tMenu");
    printf("\n-----------------------------------");
    printf("\n 1. Insert in element Queue");
    printf("\n 2. Delete element from a Queue");
    printf("\n 3. Exit \n");
    printf("------------------------------------\n");
    while(1)
    {
        printf("Choose operation : ");
        scanf("%d", &ch);
        witch(ch)
        {
            case 1 : //... Insert
                printf("\nEnter logical queue number (0 to %d) : ", n-1);
                scanf("%d", &qno);
                printf("Element to be entered in queue number %d : ",qno);
                scanf("%d", &ele);
                reply = insq(queue, qno, r, limit, &ele);
                if( reply == -1)
                    printf("Queue %d is full \n", qno);
                else
                    printf("%d is inserted in queue number %d.\n\n", ele, qno);
                break;
            case 2 : //... Delete
                printf("\nEnter logical queue number (0 to %d) : ", n-1);
                scanf("%d", &qno);
                reply = delq(queue, qno, f, r, &ele);
                if( reply == -1)
                    printf("\n Queue %d is empty. \n", qno);
                else
                    printf("%d is deleted from queue number %d \n\n",ele, qno);
                break;
            case 3 : exit(0);
            default: printf("Invalid operation \n");
        }
    }
    return 0;
}
```