Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering (SCOPE)

CSE2011-DATASTRUCTURES LAB RECORD

FALL 21-22

Name of the Student	**:**	**Mayank Gupta**
Register Number	**:**	**20BCE1538**
Slot	**:**	**L27+L28**
Faculty Name	**:**	**Dr.V.Vani**

INDEX

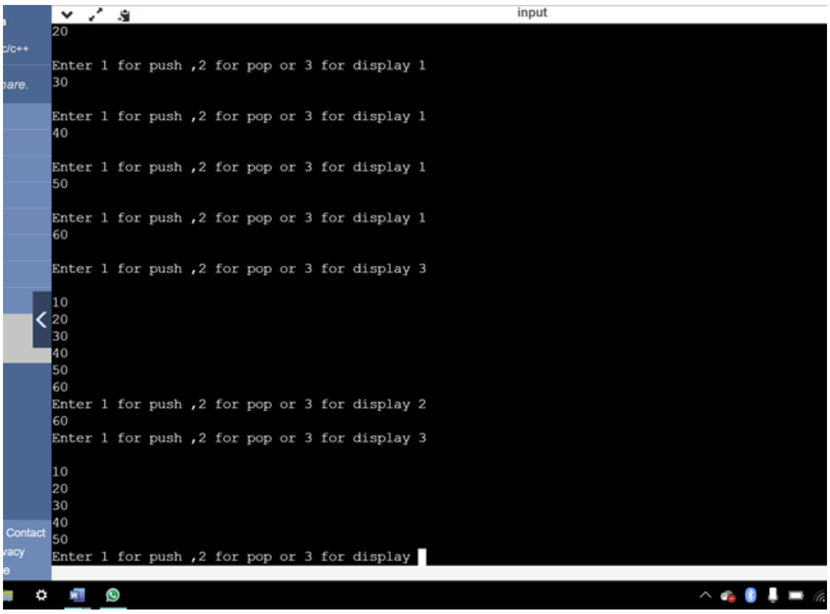
LIST OF EXPERIMENTS

AIM Implementation of Stack Data Structure

CODE

```
#include <stdio.h>
int n = 10;
int stack[10];
int top = -1;
int push ()
{
 int X;
 if(top > n-1)
 printf ("stack is full");
 else{
  scanf ("%d",&X) ;
  top++;
  stack[top]=X;}
}

int pop ()
{

 if (top==-1)
 printf("stack is empty");
 else{


    top --;
    printf("%d",stack[top+1]);

 }
}

int display ()
{
 for (int i=0 ; i<=top ; i++)
 printf ("\n%d",stack[i]);
}
```

```
int
main ()
{
 int j=0;
 while (j != 4){
 printf ("\nEnter 1 for push ,2 for pop or 3 for display ");
 scanf ("\n%d", &j);

 switch (j)
  {
  case 1:
    push ();
    break;
  case 2:
    pop ();
    break;
  case 3:
    display ();
    break;
  }
 }
return 0;
}
```



isfull()

Algorithm of isfull() function –

```
begin procedure isfull

  if top equals to MAXSIZE
    return true
  else
    return false
  endif

end procedure
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty

  if top less than 1
    return true
  else
    return false
  endif

end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- Step 1 – Checks if the stack is full.
- Step 2 – If the stack is full, produces an error and exit.
- Step 3 – If the stack is not full, increments top to point next empty space.
- Step 4 – Adds data element to the stack location, where top is pointing.

- Step 5 – Returns success.

Stack Push Operation

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation
A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data

  if stack is full
    return null
  endif

  top ← top + 1
  stack[top] ← data
```

end procedure

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

- Step 1 – Checks if the stack is empty.
- Step 2 – If the stack is empty, produces an error and exit.
- Step 3 – If the stack is not empty, accesses the data element at which top is pointing.
- Step 4 – Decreases the value of top by 1.
- Step 5 – Returns success.

Stack Pop Operation
Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

  if stack is empty
    return null
  endif

  data ← stack[top]
  top ← top - 1
  return data

end procedure
```

Pseudo code:

Push:

Top= top+1[new top value o] A[top]=x
If top >n-1
Printf stack is full Else
Top++ A[top]=x

pop:

if top==-
1 printf stack is empty
else
top--
printf A[top+1]

display:

forloop to displ
ay

AIM Implementation of Queue Data Structure

CODE

```
#include <stdio.h>
void insert();
void delete();
void display();
int queue[100];
int back=- 1;
int front=- 1;
int main()
{
    int a;
    char val='Y';
    while(val=='Y')
    {
        printf("1.Insert\n2.Delete\n3.Display\nEnter your option\n");
        scanf("%d",&a);
            switch(a)
            {
                case 1:
                    insert();
                    break;
                case 2:
                    delete();
```

```
                break;
            case 3:
                display();
                break;
            default:
                printf("Try again using 1 or 2 or 3\n");
        }
        printf("Enter Y if you want to proceed else enter N\n");
        scanf("%s",&val);
    }
}
void insert()
{
    int value;
    if(back==99)
        printf("Queue Overflow\n");
    else
    {
        if(front==- 1||front==0)
        {
            front=0;
            printf("Enter the element to the queue\n");
            scanf("%d", &value);
            back=back+ 1;
            queue[back]=value;
        }
    }
}
void delete()
{
    if(front==- 1||front>back)
    {
        printf("Queue Underflow\n");
    }
    else
    {
        printf("Element deleted from queue is %d\n", queue[front]);
        front=front + 1;
    }
}
void display()
{
    int i;
    if(front==- 1)
        printf("Queue is empty\n");
    else
    {
        printf("Queue is");
        for(i=front;i<=back;i++)
        {
            printf(" %d ",queue[i]);
```

```
        }
        printf("\n");
    }
}
```

```
1.Insert
2.Delete
3.Display
Enter your option
1
Enter the element to the queue
4
Enter Y if you want to proceed else enter N
Y
1.Insert
2.Delete
3.Display
Enter your option
1
Enter the element to the queue
5
Enter Y if you want to proceed else enter N

Y
1.Insert
2.Delete
3.Display
Enter your option
1
Enter the element to the queue
7
Enter Y if you want to proceed else enter N
Y
1.Insert
2.Delete
3.Display
Enter your option
8
Try again using 1 or 2 or 3
Enter Y if you want to proceed else enter N
Y
```

```
Enter your option
1
Enter the element to the queue
8
Enter Y if you want to proceed else enter N
Y
1.Insert
2.Delete
3.Display
Enter your option
3
Queue is 4  5  7  8
Enter Y if you want to proceed else enter N
Y
1.Insert
2.Delete
3.Display
Enter your option
2
Element deleted from queue is 4
Enter Y if you want to proceed else enter N
Y
1.Insert
2.Delete
3.Display
Enter your option
2
Element deleted from queue is 5
Enter Y if you want to proceed else enter N
N


...Program finished with exit code 0
```

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- enqueue() − add (store) an item to the queue.
- dequeue() − remove (access) an item from the queue.
- Few more functions are required to make the above-mentioned queue operation efficient. These are −
- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek
  return queue[front]
end procedure
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull

  if rear equals to MAXSIZE
    return true
  else
    return false
  endif

end procedure
```

isempty()

Algorithm

```
begin procedure isempty

  if front is less than MIN  OR front is greater than rear
    return true
  else
    return false
  endif
```

```
end procedure
```

Enqueue Operation

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- Step 1 − Check if the queue is full.
- Step 2 − If the queue is full, produce overflow error and exit.
- Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
- Step 4 − Add data element to the queue location, where the rear is pointing.
- Step 5 − return success.

Insert Operation

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation
procedure enqueue(data)

```
 if queue is full
   return overflow
 endif

 rear ← rear + 1
 queue[rear] ← data
 return true

end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation −

- Step 1 − Check if the queue is empty.
- Step 2 − If the queue is empty, produce underflow error and exit.
- Step 3 − If the queue is not empty, access the data where front is pointing.
- Step 4 − Increment front pointer to point to the next available data element.
- Step 5 − Return success.

Remove Operation
Algorithm for dequeue operation
procedure dequeue

```
  if queue is empty
    return underflow
  end if

  data = queue[front]
  front ← front + 1
  return true

end procedure
```