GROUP 3
BSIT


Calderon


Dimasacat


Galope


Go


Naoe


Odrunia

SUBJECT OF DISCUSSION

/01 STACK

L.I.F.O, Stack Operations, Code Implementation

/02 EXPRESSION PARSING

Notations,Precedence & Associativity, Parsing using stack

/03 QUEUE

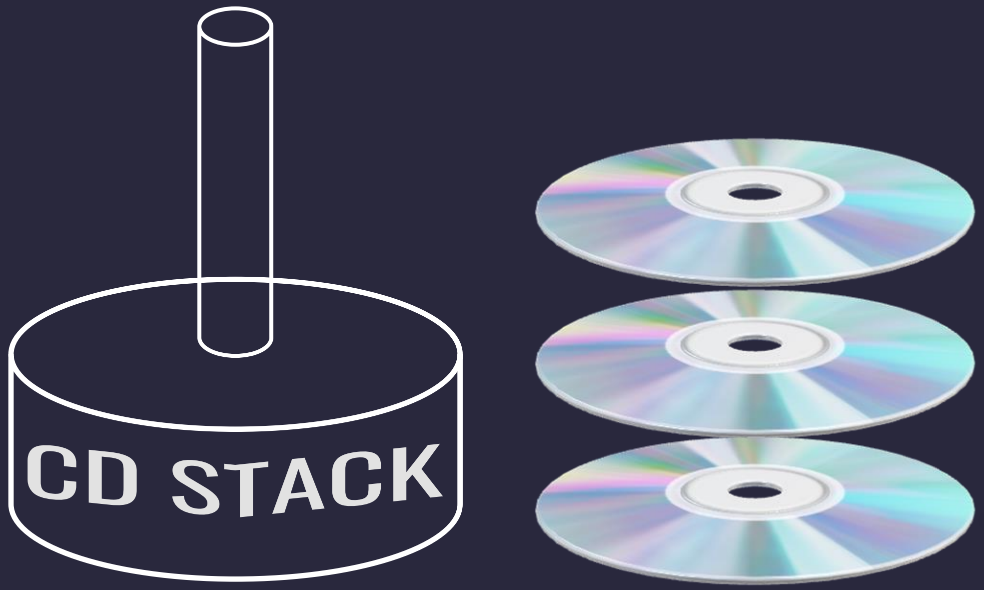
F.I.F.O, Queue Operations, Code Implementation

<STACK>

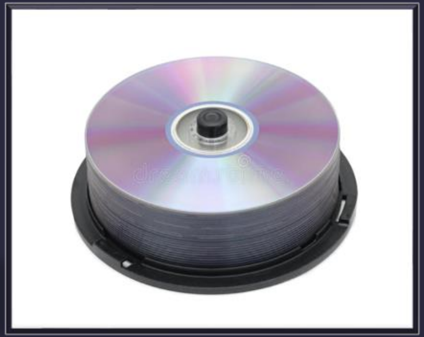
GROUP 3_BSIT

STACK.PPTX

LOGICAL REPRESENTATION OF STACK



RULE: Insertion and Deletion is possible from only one end.



L.I.F.O

LAST
IN
FIRST
OUT

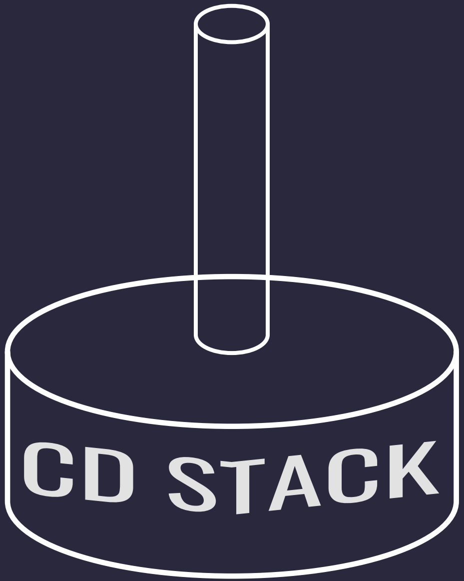
F.I.L.O

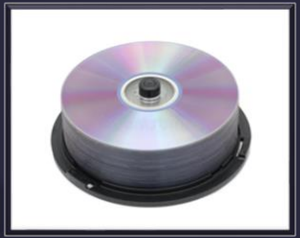
FIRST
IN
LAST
OUT

PUSH

POP

RULE: Insertion and Deletion is possible from only one end.





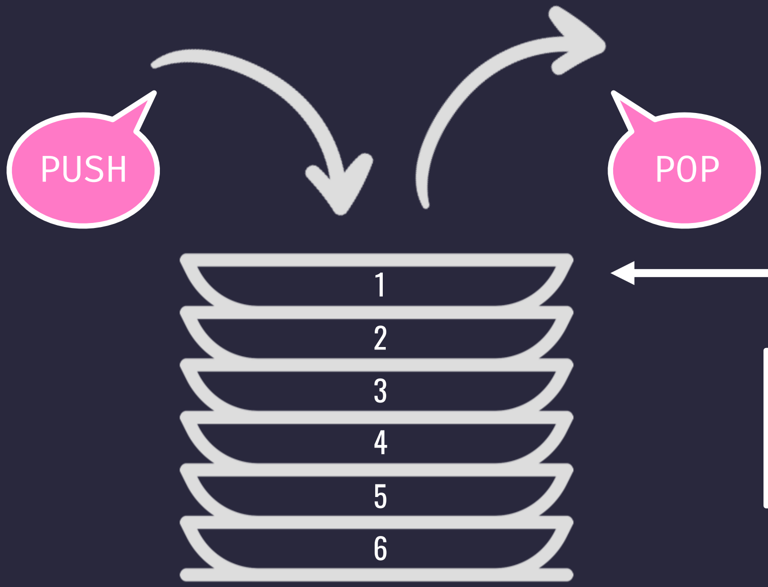
2 FUNDAMENTAL STACK OPERATION

PUSH

Inserting or putting data into the stack.

POP

Taking out or deleting the top most data from the stack.



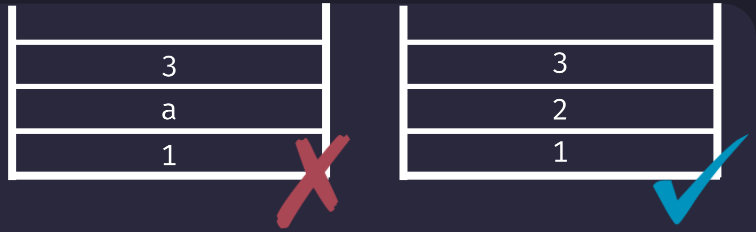
Top or Peek

Only place in Stack where you can push(insert) or pop(delete) data is called ***<u>"Top" or "Peek"</u>***

STACK OPERATIONS

Remember: Stack is a collection of similar data type only! You can only insert a data of similar data type, either integer, character, float.

push(x) – Operation you need to INSERT a data into a stack.



pop() – The top most element will always be the one that pops out or deleted.
[It will return the top most element from the stack and delete that element as well]

peek() or top() – It will going to return the top most element in the stack without removing that element from the stack.

isEmpty() – It will return **True** if the Stack is “**Empty**”, otherwise it will return False.

isFull() – It will return **True** if the Stack is “**Full**”, otherwise it will return False.

APPLICATIONS OF STACK

REVERSE A STRING

UNDO

STACK OPERATIONS

/01 push()

Storing an element to a stack

/02 pop()

Removing an element from the stack

/03 peek()

Get the top element of the data without removing it

/04 isFull()

Check if stack is full

/05 isEmpty()

Check if stack is empty

push() – pseudo code && implementation

```
begin procedure push: stack,
data

 if stack is full
       return null
 endif
      top ← top + 1
      stack[top] ← data

end procedure
```

```cpp
#include <iostream>
#include <stack>
int main()
{
    std::stack<int> myStack;

    myStack.push(23);
}
```

pop() – pseudo code && implementation

```
begin procedure pop: stack

 if stack is empty
      return null
 endif
      data ← stack[top]
      top ← top - 1
      return data

end procedure
```

```cpp
#include <iostream>
#include <stack>
int main()
{
    std::stack<int> myStack;
    myStack.push(23);

    myStack.pop();
}
```

peek() – pseudo code && implementation

```
begin procedure peek

      return stack[top]

end procedure
```

```cpp
#include <iostream>
#include <stack>
int main()
{
    std::stack<int> myStack;
    myStack.push(23);
    myStack.push(24);

    std::cout << myStack.top();
}
```

isFull() – pseudo code

```
begin procedure isFull

 if top equals to MAXSIZE
      return true
 else
      return false

end procedure
```

isFull() – C++ implementation

```cpp
#include <iostream>
#include <stack>
#define MAXSIZE 5

bool isFull(std::stack<int> stack)
{
    return stack.size() == MAXSIZE ? true : false;
}
```

isFull() – C++ implementation

```
int main()
{
    std::stack<int> stack;

    stack.push(32);
    stack.push(34);

    isFull(stack) ?
    std::cout << "full" :
    std::cout << "not full";
}
```

isEmpty() – pseudo code && implementation

```
begin procedure isEmpty

 if top less than 1
       return true
 else
       return false

end procedure
```

```
#include <iostream>
#include <stack>
int main()
{
    std::stack<int> myStack;
    myStack.push(23);
    myStack.pop();

    stack.empty() ?
    std::cout << “empty" :
    std::cout << “not empty" ;
}
```

</STACK>

SUBJECT OF DISCUSSION

/01 STACK

L.I.F.O, Stack Operations, Code Implementation

/02 EXPRESSION PARSING

Notations,Precedence & Associativity, Parsing using stack

/03 QUEUE

F.I.F.O, Queue Operations, Code Implementation

<EXPRESSION_PARSING>

TERMINOLOGIES

EXPRESSION

a statement that generates a value on evaluation.

PARSING

analyzing a string or a set of symbols one by one depending on a particular criterion.

EXPRESSION PARSING

a term used in a programming language to evaluate arithmetic and logical expressions.

NOTATIONS

INFIX

Operators are written in-between their operands.

PREFIX

Operators are written before their operands.

POSTFIX

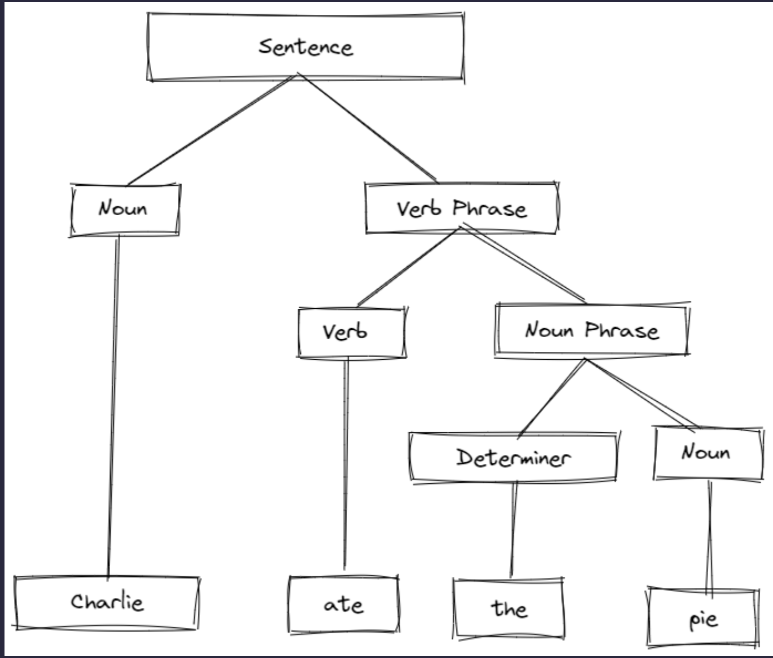
Operators are written after their operands.

NOTATIONS

INFIX	PREFIX	POSTFIX
a + b	+ a b	a b +
(a + b) * c	* + a b c	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	a b + c d + *

PARSING TREE

Charlie ate the pie



EXPRESSION PARSING: PREFIX NOTATION

```
* + a b c

*   x   c

  answer
```

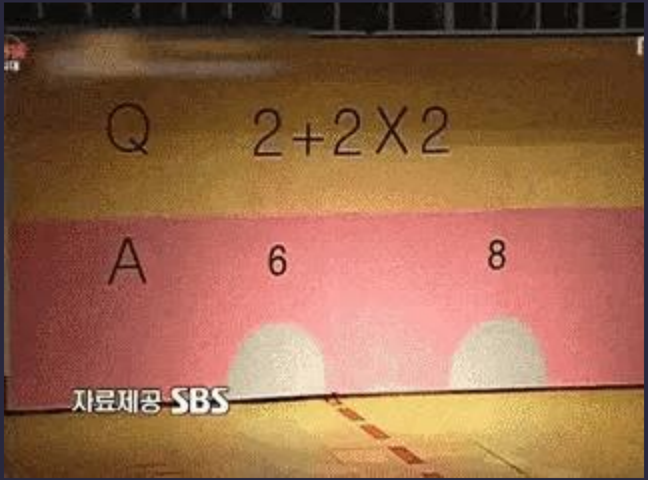
```
(a + b) * c
```

EXPRESSION PARSING: POSTFIX NOTATION

$$\underbrace{a\ \ b\ \ +}_{x}\ \ c\ \ *$$

$$\underbrace{x\ \ c\ \ *}_{\text{answer}}$$

$$(a + b) * c$$



$$3^3 - (9 \times 2) \div 6$$

$$9 + (3 \times 2) - 4$$

$$19 + 40 \div 5 - (8 + 5)$$

P.E.M.D.A.S.

Parenthesis
Exponent
Multiplication
Division
Addition
Subtraction

PRECEDENCE AND ASSOCIATIVITY OF OPERATORS

Precedence of operators come into picture when in an expression we need to decide which operator will be evaluated first. Operator with higher precedence will be evaluated first.

Associativity of operators came into picture when precedence of operators are same and we need to decide which operator will be evaluated first.

```
p + q * r
p + (q * r )
```

SR. NO.	OPERATOR	PRECEDENCE	ASSOCIATIVITY
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

a * b - c / d + e

SR. NO.	OPERATOR	PRECEDENCE	ASSOCIATIVITY
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

$$((a * b) + (c / d)) - e$$

SR. NO.	OPERATOR	PRECEDENCE	ASSOCIATIVITY
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

100 + 200 / 10 - 3 * 10

SR. NO.	OPERATOR	PRECEDENCE	ASSOCIATIVITY
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

100 + (200 / 10) - (3 * 10)

SR. NO.	OPERATOR	PRECEDENCE	ASSOCIATIVITY
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

100 + (20) - (30)

SR. NO.	OPERATOR	PRECEDENCE	ASSOCIATIVITY
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

(100 + (20)) - (30)

SR. NO.	OPERATOR	PRECEDENCE	ASSOCIATIVITY
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

(120) - (30)

SR. NO.	OPERATOR	PRECEDENCE	ASSOCIATIVITY
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

90

SR. NO.	OPERATOR	PRECEDENCE	ASSOCIATIVITY
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

INFIX TO POSTFIX ALGORITHM

1. Scan input string from left to right character by character.

2. If the character is an operand, put it into output stack.

3. If the character is an operator and operator’s stack is empty, push operator into operator’s stack.

4. If the operator’s stack is not empty, there may be following possibilities:

INFIX TO POSTFIX ALGORITHM

- If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operand's stack.
- If the precedence of scanned operator is less than or equal to the top most operator of operator's stack, pop the operators from operand's stack until we find a low precedence operator than the scanned character. Never pop out ('(') or (')') whatever may be the precedence level of scanned character.
- If the character is opening round bracket ('('), push it into operator's stack.
- If the character is closing round bracket (')'), pop out operators from operator's stack until we find an opening bracket ('(').
- Now pop out all the remaining operators from the operator's stack and push into output stack.

INFIX TO POSTFIX NOTATION USING STACK

SO > TO → PUSH

SO <= TO → POP

(→ PUSH

) → POP

(a+b) * (c+d)

INFIX TO POSTFIX NOTATION USING STACK

SO > TO → PUSH

SO <= TO → POP

(→ PUSH

) → POP

a/b+c/d

INFIX TO PREFIX ALGORITHM

1. Reverse the infix expression (i.e A+B*C will become C*B+A).

Note: While reversing each ‘(‘ will become ‘)’ and each ‘)’ becomes ‘(‘.

2. Obtain the “nearly” postfix expression of the modified expression (i.e CB*A+).

3. Reverse the postfix expression. Hence in our example prefix is +A*BC.

INFIX TO PREFIX NOTATION USING STACK

SO > TO → PUSH

SO <= TO → POP

) → PUSH

(→ POP

a+b

b+a

INFIX TO PREFIX NOTATION USING STACK

SO > TO → PUSH

SO <= TO → POP

) → PUSH

(→ POP

(a+b)*c

</EXPRESSION_PARSING>

SUBJECT OF DISCUSSION

/01 STACK

L.I.F.O, Stack Operations, Code Implementation

/02 EXPRESSION PARSING

Notations,Precedence & Associativity, Parsing using stack

/03 QUEUE

F.I.F.O, Queue Operations, Code Implementation

<QUEUE>

QUEUE

Queue is an abstract data structure. Somewhat similar to Stacks. It is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

LOGICAL REPRESENTATION OF QUEUE

A QUEUE REMINDER

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

QUEUE OPERATIONS

/01 peek()

Gets the element at the front of the queue without removing it.

/02 isfull()

Checks if the queue is full.

/03 isempty()

Checks if the queue is empty.

/04 enqueue()

add (store) an item to the queue.

/05 dequeue()

remove (access) an item from the queue.

peek() – pseudo code && implementation

```
begin procedure peek

    return queue[front]

end procedure
```

```cpp
int peek() {
        return queue[front];
}
```

isfull() – pseudo code && implementation

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.

```
begin procedure isfull
    if rear equals to MAXSIZE
        return true
    else
        return false
    endif
end procedure
```

```
bool isfull() {
    if(rear == MAXSIZE - 1)
        return true;
    else
        return false;
}
```

isempty() – pseudo code && implementation

```
begin procedure isempty

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

```
bool isempty() {
   if(front < 0 || front >
rear)
      return true;
   else
      return false;
}
```

enqueue() – steps && pseudo code

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.

```
procedure
enqueue(data)

   if queue is full
      return overflow
   endif

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

end procedure
```

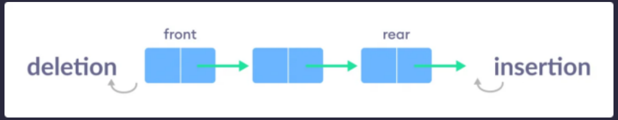
dequeue() – steps && pseudo code

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.

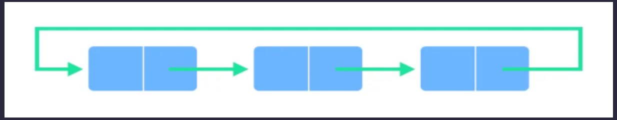
```
procedure dequeue

    if queue is empty
        return underflow
    end if

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

end procedure
```

TYPES OF QUEUE

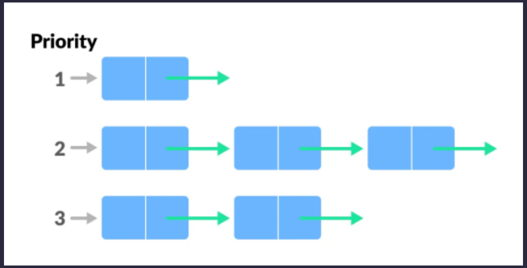
1. Simple Queue



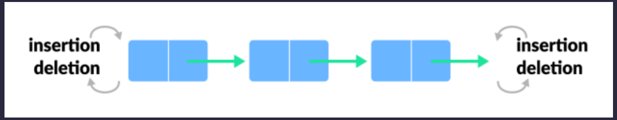
2. Circular Queue



3. Priority Queue



4. Double-Ended Queue



FRONT

REAR

3

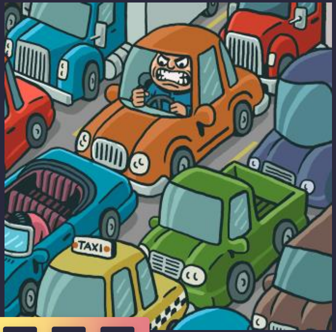
5

2

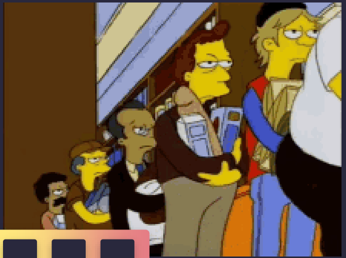
8

10

FIFO - First-In-First-Out methodology



FIFO - First-In-First-Out methodology

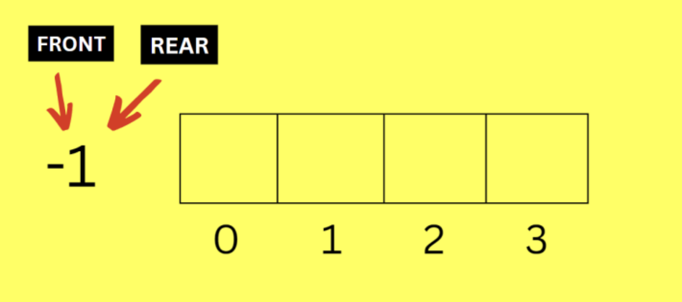


ENQUEUE(ADD)

DEQUEUE(REMOVE)

Declaring Variables

```
#include <iostream>

using namespace std;

int queue1[100];
int n=100, Front = -1, Rear = -1;
```



isEmpty()

```
void isEmpty(){
     if(Rear ==-1 || Front ==-1 ){
          cout<<"Queue is Empty"<<endl;
     }
     else
          cout<<"Queue is Not Empty!"<<endl;
}
```

isFull()

```
void isFull(){
     if(Rear==n-1){
     cout<<"Queue is Full"<<endl;
     }
     else
          cout<<"Queue is Not
Full!"<<endl;
}
```

peek()

```
void peek(){
    if(Front==-1 && Rear==-1){
        cout<<"There is no element inside the queue to
display"<<endl;
    }
    else
         cout<<"The element at the front node is:
"<<queue1[Front]<<endl;
}
```

enqueue()

```
int element;
    if (Rear == n-1){
        cout<<"Overflow Error"<<endl;
    }
    else
    if (Front==-1){
        Front=0;
    }
     cout<<"Enter the element for insertion: ";
     cin>>element;
     Rear++;
     queue1[Rear]=element;
}
```

dequeue()

```
void dequeue(){
    if (Front ==-1 && Rear==-1){
        cout<<"Underflow Error";
    }
    else if(Front == Rear){
        cout<<"The deleted element from the queue is:"<<queue1[Front]<<endl;
         Front = Rear =-1;
    }
    else
         cout<<"The deleted element from the queue
         is:"<<queue1[Front]<<endl;

             Front++;

}
```

display()

```
void display(){
     if(Front==-1){
          cout<<"Queue is Empty!"<<endl;
     }
     else
          cout<<"Queue elements are:"<<endl;
     for(int i =Front; i <= Rear; i++)
          cout<<queue1[i]<<"\n";
          cout<<endl;
}
```

GROUP 3_BSIT

</QUEUE>

QUEUE.PPTX