Data Structures and Algorithms

W3- Lecture 1,2

Stacks and Queues

Engr. Bushra Tahir
Department of Electrical Engineering
Iqra National University

Definition of Stack

- A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations occur at one end only, called the top of the stack.

Example

Stack of Plates

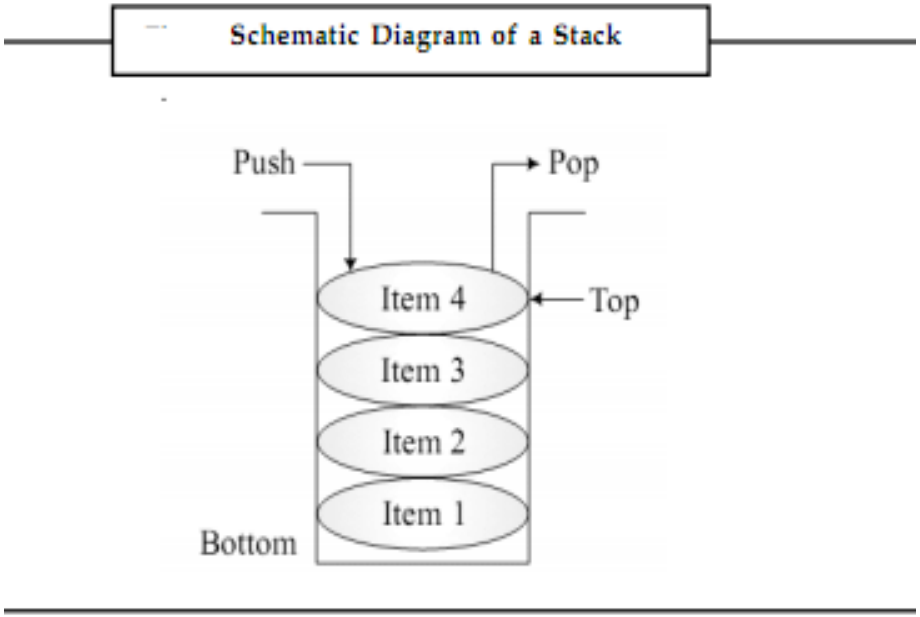
- Here, one plate is placed on top of another, thus creating a stack of plates.
- Suppose, a person takes a plate off the top of the stack of plates. The plate most recently placed on the stack is the first one to be taken off.
- The bottom plate is the first one placed on the stack and the last one to be removed.

Operations on Stack

The primitive operations that can be performed on a stack are given below:

1. Inserting an element into the stack (PUSH operation)
2. Removing an element from the stack (POP operation)
3. Determining the top item of a stack without removing it from the stack (PEEP/Top operation)

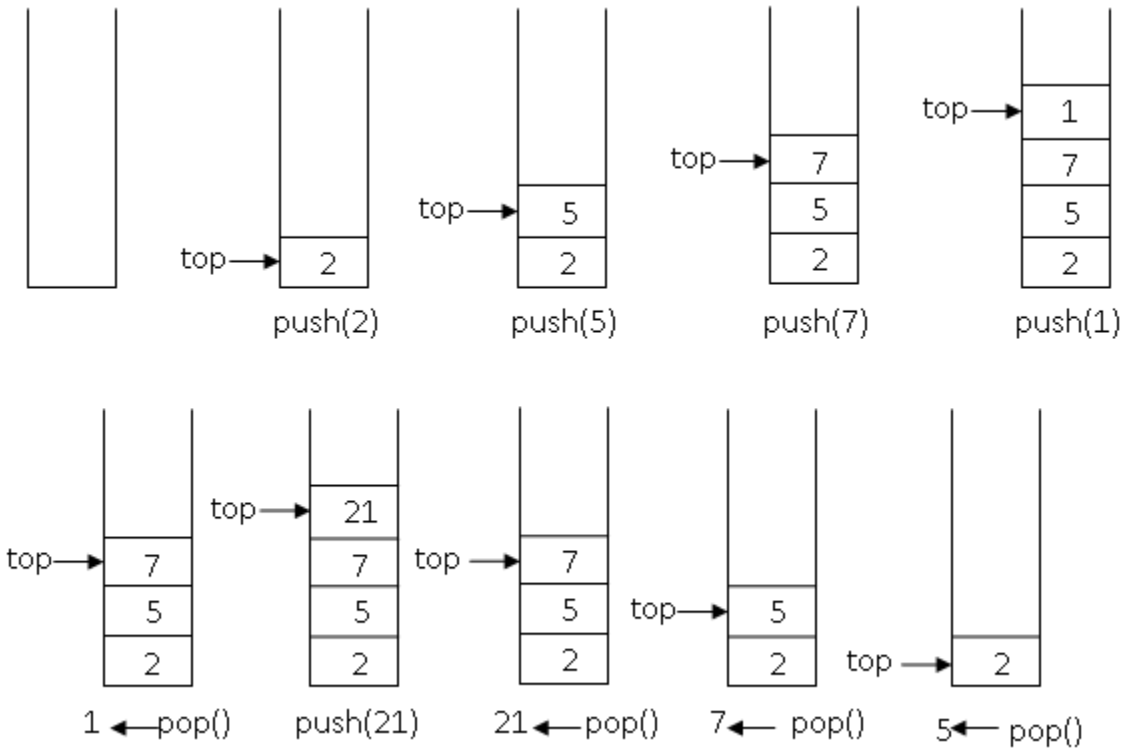
Schematic Diagram



Operation on Stack

- Practically saying, a stack can be an array where elements are inserted in LIFO fashion
- In case of list, we have used *add*, *remove*, *get*, *set* as the suitable names.
- However, for stack, we are using *push*, *pop* and *top/peep*.

Working of Stack



LIFO Structure

- The last element to go into the stack is the first to come out. That is why, a stack is known as *LIFO* (Last In First Out) structure.

Push Operation

- Push function is used for inserting the elements into a stack.
- We have an array named *A* while *current* is its index.

```
void push(int x)
{
   A[++current] = x;
}
```

- *++current* means that add one to the *current* and then use it.
- That also shows that element *x* should be inserted at *current* plus one position.
- Before using the *push* method, the user must call *isFull()* method.

POP Operation

- POP function is used for deleting the elements from a stack.
- Stack follows a mechanism of LIFO, hence the last element to be inserted is the first element to be deleted.
- Before using the *pop* method, the user must call *isEmpty()* method.

POP Operation

The code of *pop()* method is as:

```
int pop()
{
   return A[current--];
}
```

In this method, the recent element is returned to the caller, reducing the size of the array by 1.

PEEP Operation

- It is also possible to verify the item placed at the top of the stack without removing it.
- This operation is called PEEP/top.
- The code of the *top()* method is:

```
int top()
{
   return A[current];
}
```

- This method returns the element at the current position. We are not changing the value of *current* here. We simply want to return the top element.

isEmpty() function

- What happens if we call *pop()* while there is no element?
- One possible way-out is that we have *isEmpty()* function that returns true if stack is empty and false otherwise.

isEmpty() function

- This method returns the element at the current position. We are not changing the value of *current* here. We simply want to return the top element.

```
int isEmpty()
{
    return ( current == -1 );
}
```

isEmpty() function

- This method also tests the value of the *current* whether it is equal to -1 or not.
- Initially when the stack is created, the value of *current* will be -1.
- If the user calls the *isEmpty()* method before pushing any element, it will return true.

isFull() Function

- It is possible that the array may ‘fill-up’ if we push enough elements. Now more elements cannot be pushed.
- To avoid this, we write *isFull()* method that will return a Boolean value.
- If this method returns true, it means that the stack (array) is full and no more elements can be inserted.
- Therefore before calling the *push(x)*, the user should call *isFull()* method.
- If *isFull()* returns false, it will depict that stack is not full and an element can be inserted.

isFull() Function

```
int isFull()
{
    return ( current == size-1);
}
```

This method checks that the stack is full or not.

The variable *size* shows the size of the array.

If the *current* is equal to the *size* minus one, it means that the stack is full and we cannot insert any element in it.

Example

```
/* Stack implementation using array */

#include <iostream.h>

/* The Stack class */

class  Stack
{
   public:
      Stack() { size = 10; current = -1;}       //constructor
      int pop(){ return A[current--];}          // The pop function
      void push(int x){A[++current] = x;} // The push function
      int top(){ return A[current];}            // The top function
      int isEmpty(){return ( current == -1 );} // Will return true when stack is empty
      int isFull(){ return ( current == size-1);} // Will return true when stack is full

    private:
      int   object;                                          // The data element
      int   current;                                         // Index of the array
      int   size;                                            // max size of the array
      int   A[10];                                           // Array of 10 elements
};
```

Example

```
// The main method
int main()
{
   Stack stack;                                                    // creating a stack object
   // pushing the 10 elements to the stack
   for(int i = 0; i < 12; i++)
   {
       if(!stack.isFull())                    // checking stack is full or not
              stack.push(i);      // push the element at the top
       else
              cout <<"\n Stack is full, can't insert new element";
   }

   // pop the elements at the stack
   for (int i = 0; i < 12; i++)
   {
      if(!stack.isEmpty())                    // checking stack is empty or not
              cout << "\n The popped element = " << stack.pop();
      else
              cout <<"\n Stack is empty, can't pop";
   }
}
```

Example (Output)

```
Stack is full, can't insert new element
The popped element = 9
The popped element = 8
The popped element = 7
The popped element = 6
The popped element = 5
The popped element = 4
The popped element = 3
The popped element = 2
The popped element = 1
The popped element = 0
Stack is empty, can't pop
```

QUEUES

Definition

- A queue is a linear data structure into which items can only be inserted at one end and removed from the other.
- In contrast to the stack, which is a LIFO (Last In First Out) structure, a queue is a FIFO (First In First Out) structure.

Example

- For example, we queue up while depositing a utility bill or purchasing a ticket. The objective of that queue is to serve persons in their arrival order; the first coming person is served first. The person, who comes first, stands at the start followed by the person coming after him and so on. At the serving side, the person who has joined the queue first is served first.

Queue Operations

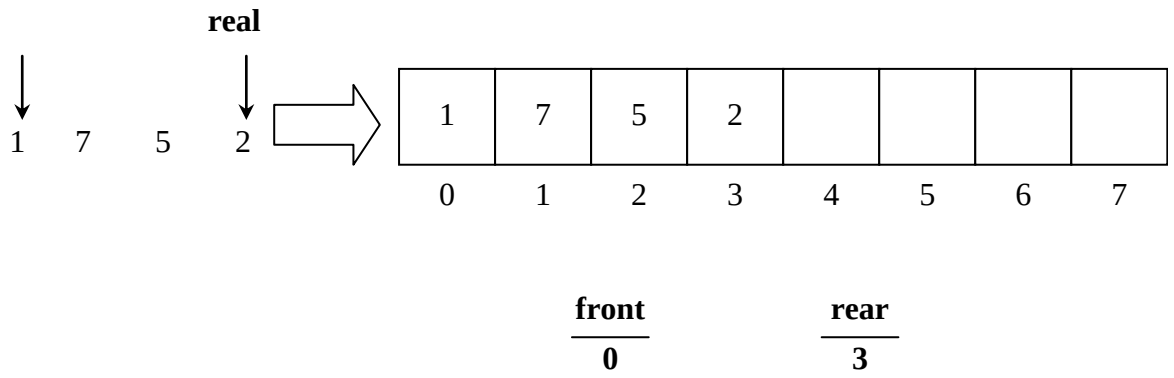
- The queue data structure supports the following operations

Operation	Description
enqueue(X)	Place X at the *rear* of the queue.
dequeue()	Remove the *front* element and return it.
front()	Return *front* element without removing it.
isEmpty()	Return TRUE if queue is empty, FALSE otherwise

Queue using Array

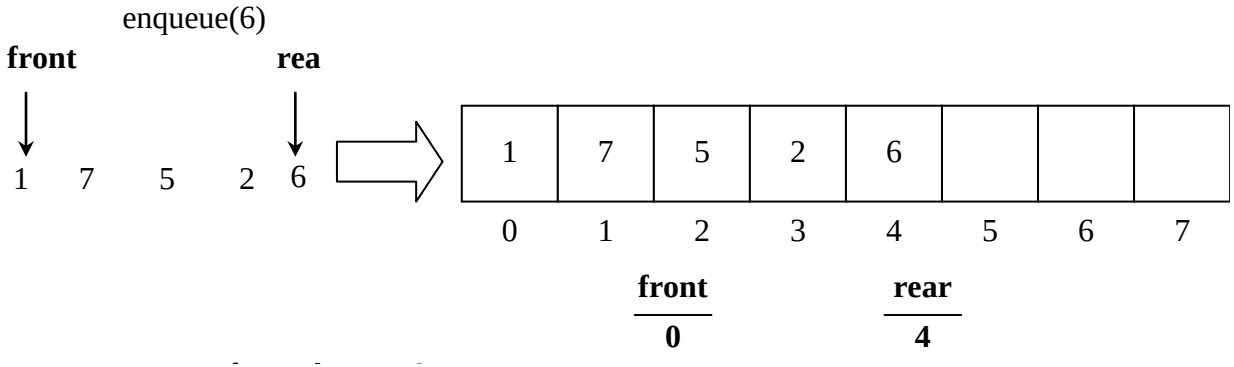
- If we use an array to hold the queue elements, both insertions and removal at the front (start) of the array are expensive. This is due to the fact that we may have to shift up to “n” elements.
- For the stack, we needed only one end
- A queue, requires both.

Queue using Array

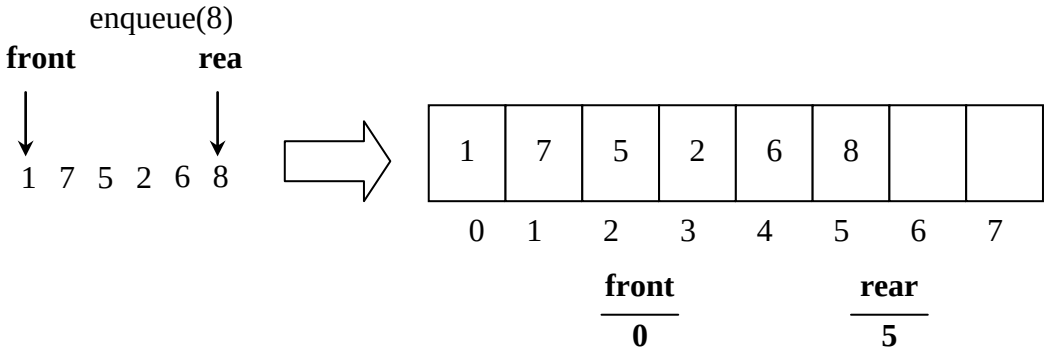


- Array size is *8*
- The *front* and *rear* in this implementation are not pointers but just indexes of arrays. *front* contains the starting index i.e. *0* while *rear* comprises *3*.

enqueue()

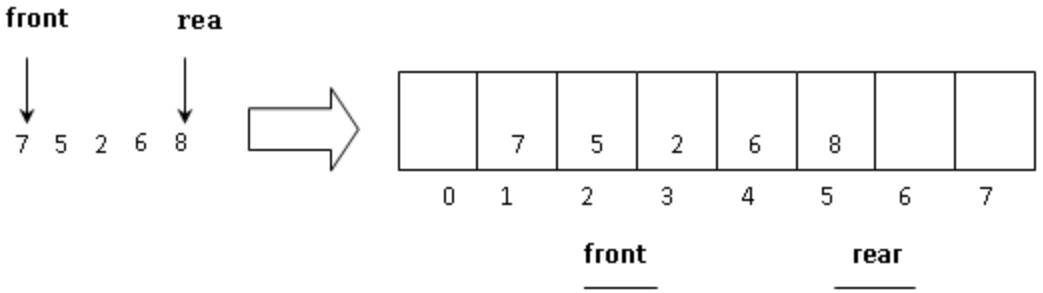


6 has been inserted in the *queue*. Now, the *rear* index is containing *4* while the *front* has the same *0* index. Let's see the figure of the array when another element *8* is inserted in the queue.

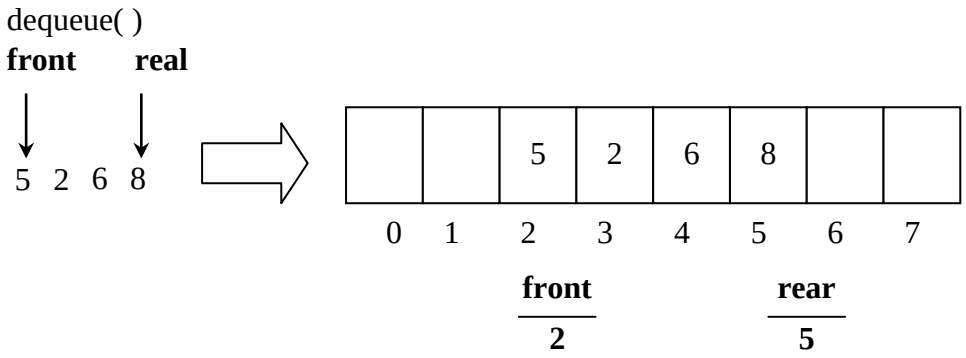


dequeue( )

- When an element is removed from the queue. It is removed from the *front* index



After another call of *dequeue()* function:



dequeue( )

- With the removal of element from the queue, we are not shifting the array elements.
- The shifting of elements might be an expensive exercise to perform and the cost is increased with the increase in number of elements in the array.
- Therefore, we will leave them as it is.

Problem with array implementation

- After insertion of two elements in the queue, the array that was used to implement it, has reached its limit as the last location of the array is in use now. We know that there is some problem with the array after it attained the size limit.

Problem with array implementation

- We can see that two locations at the start of the array are vacant.
- Therefore, we can consider how to use those locations appropriately in to insert more elements in the array.
- The solution to this problem lies in allowing the queue to *wrap around*.

Queue implementation using circular array

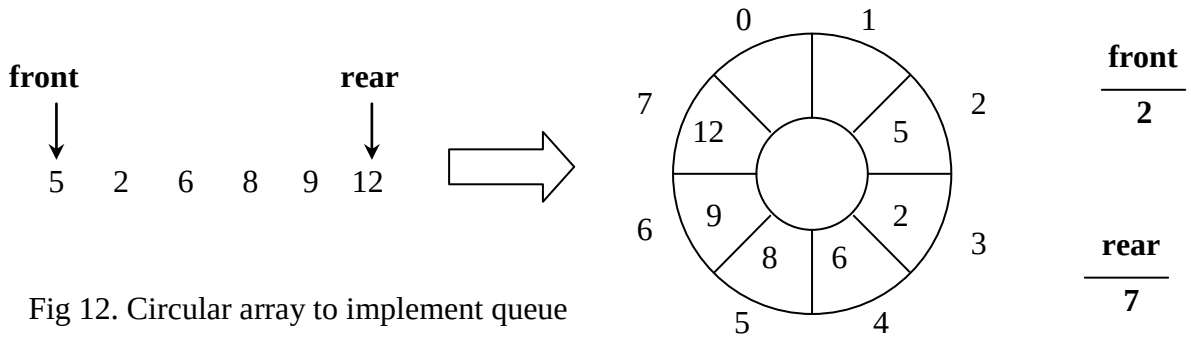


Fig 12. Circular array to implement queue

The number of locations in the above circular array are also eight, starting from index *0* to index *7*. The index numbers are written outside the circle incremented in the clock-wise direction. To insert an element *21* in the array , we insert this element in the location, which is next to index *7*.

enqueue() method

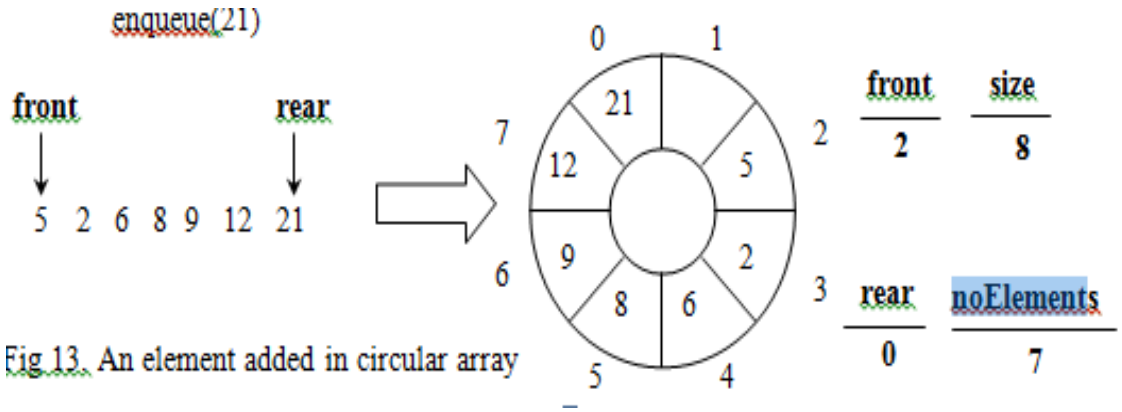


Fig 13. An element added in circular array

Now, we can see that *rear* index has decreased instread of increasing. It has moved from index *7* to *0*. *front* is containing index *2 i.e.* higher than the index in *rear*.

enqueue() method

```
void  enqueue( int  x)
{
1.    rear  =  (rear + 1) % size;
2.    array[rear]  =  x;
3.    noElements  =  noElements + 1;
}
```

dequeue()

```
int  dequeue()
{
   int   x  =  array[front];
         front  =  (front + 1) % size;
         noElements  =  noElements - 1;
         return  x;
}
```

enqueue() method

another element in the queue.

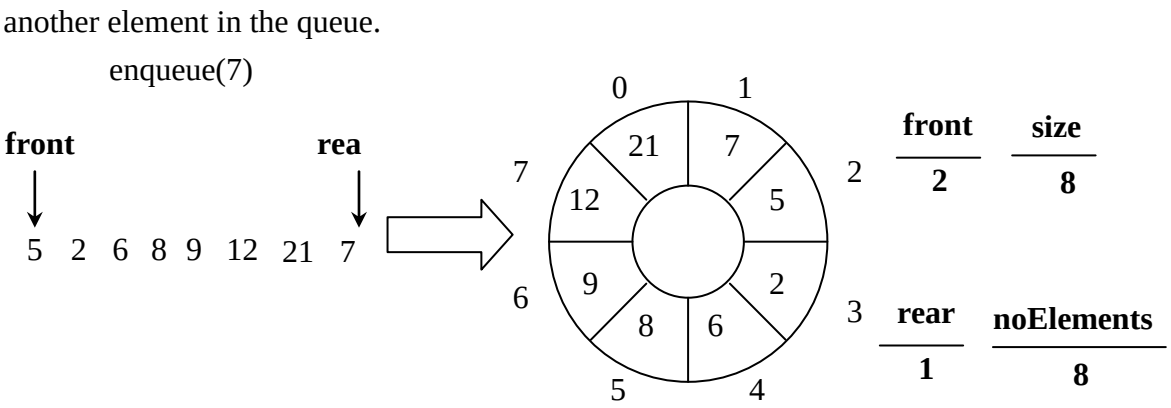


Fig 14. Another element added in circular array

Now, the queue, rather the array has become full. It is important to understand, that queue does not have such characteristic to become full. Only its implementation array has become full.

isFull()

```
int  isFull()
{
   return  noElements  ==  size;
}
```

isFull() returns true if the number of elements (*noElements*) in the array is equal to the *size* of the array. Otherwise, it returns false.

isEmpty()

- *isEmpty()* looks at the number of elements (*noElements*) in the queue.
- If there is no element, it returns true or vice versa.

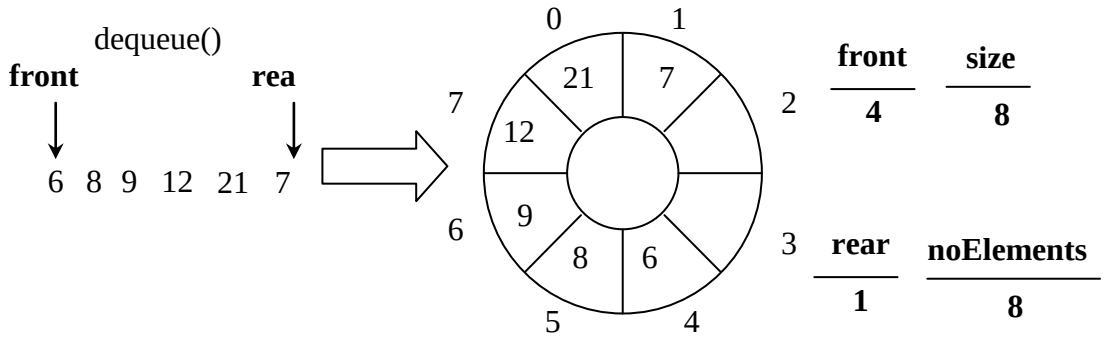


Fig 15. Element removed from the circular array

isEmpty()

```
int  isEmpty()
{
        return  noElements  ==  0;
}
```

Questions?