Data structures and algorithms

Queues

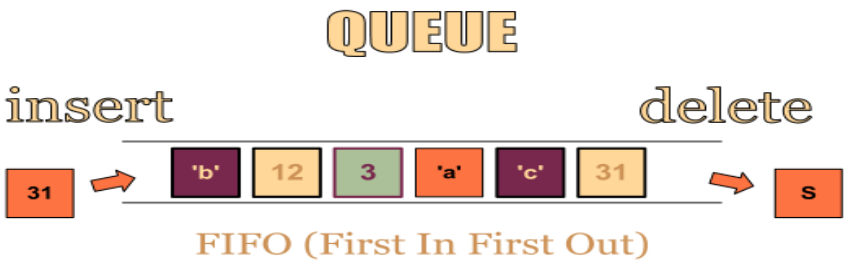
Queues

A *queue* is an ordered collection of homogenous data items such that:

- Items can be removed only at one end (*front* of the queue)
- Items can be added only at the other end (*rear* of the queue)

Accessing the elements of queues follows a *First In, First Out (FIFO)* order

Example: Customers standing in a check-out line in a store model a queue, the first customer in is the first customer served.



Queue as an ADT

A Queue is a collection of items of type T on which following operations are defined:

- **Initialize** the queue to be the *empty queue*.
- Determine whether or not the queue is **empty**.
- Determine whether or not the queue is **full**.
- If Queue is not *full*, **insert** a new item from the ***rear*** of the queue.
- If Queue is not *empty*, **remove** an item from the ***front*** of the queue.
- Retrieve the value at the ***front*** of the queue

Enqueue and Dequeue

Primary queue operations: Enqueue and Dequeue

Enqueue – insert an element at the rear of the queue.

Dequeue – remove an element from the front of the queue.

Applications of queues

Operating system

- Multi-user/multitasking environments, where several users or task may be requesting the same resource simultaneously

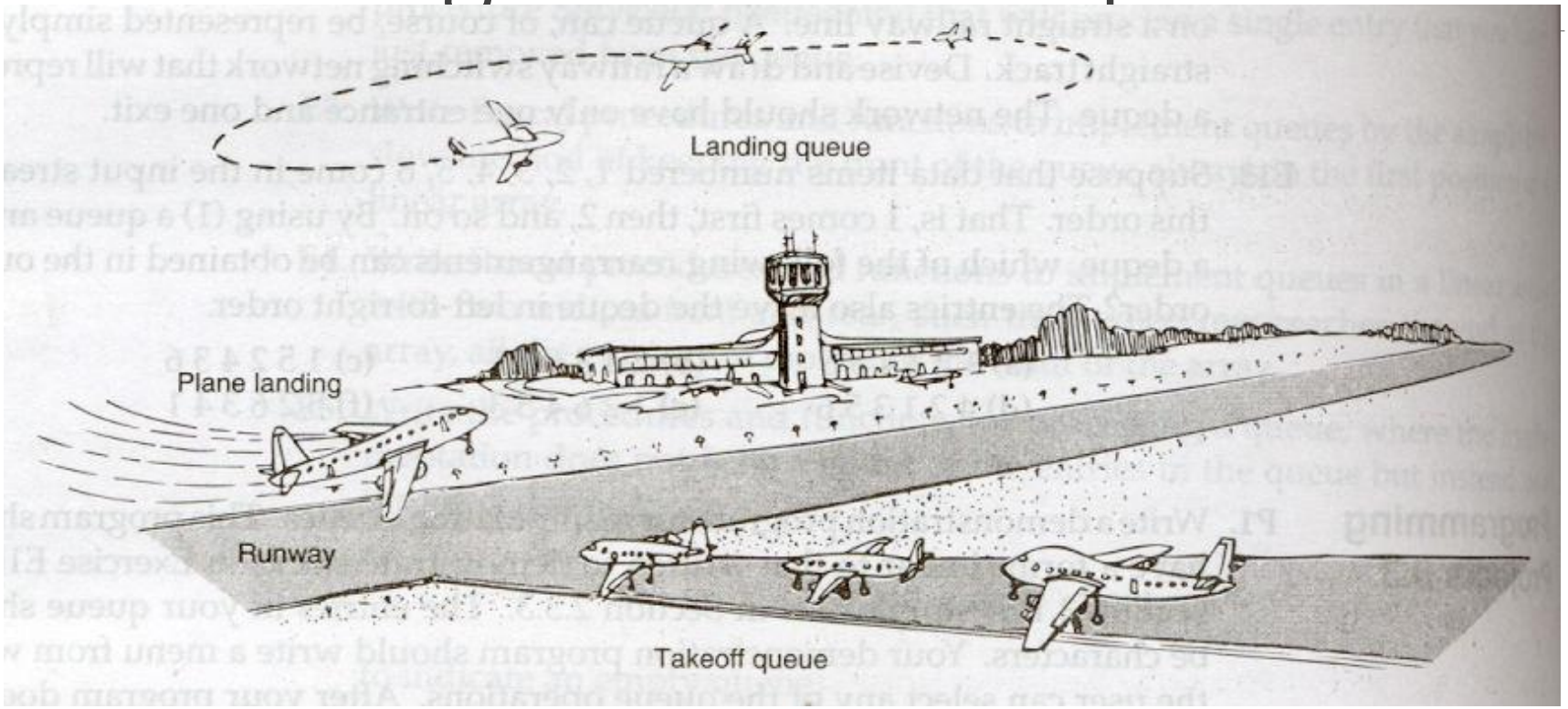
Printer Queue

- Documents to be printed in a queue

Communication Software

- Queues to hold *information* received over networks
- Information can be transmitted faster than it can be processed, so is placed in a queue waiting to be processed

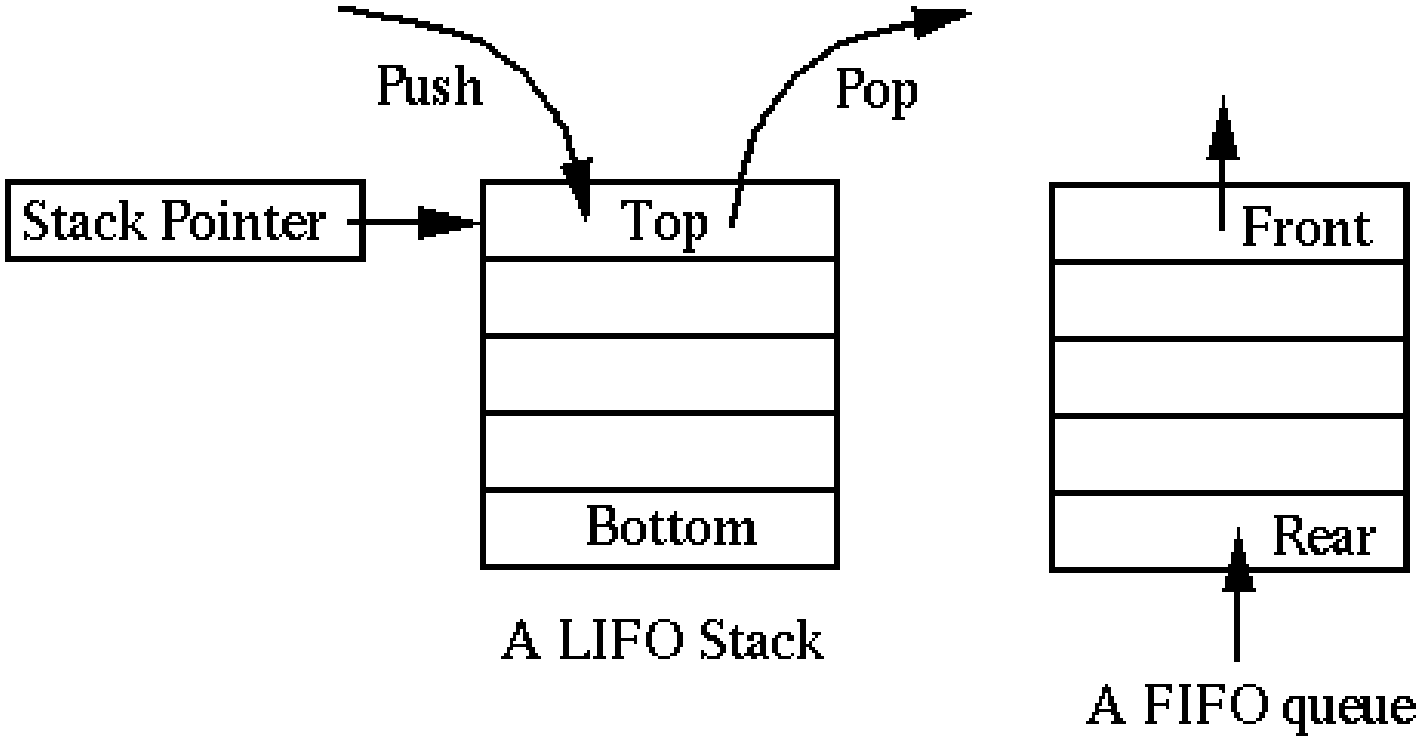
Landing and takeoff queues



Queues vs. Stacks

- Stacks are a LIFO container: Store data in the reverse of order received
- Queues are a FIFO container: Store data in the order received
- Stacks then suggest applications where some sort of reversal or unwinding is desired.
- Queues suggest applications where service is to be rendered relative to order received.
- Stacks and Queues can be used in conjunction to compare different orderings of the same data set.
- From an ordering perspective, then, Queues are the "opposite" of stacks: Easy solution to the *palindrome* problem

Queues vs. stack



Palindrome Check

```
#include "Stack.h";
#include "Queue.h";
#include <iostream.h>

int main()
{
    Stack S;
    Queue Q;
    std::cout << "Enter string to be tested as
                          palindrome: ";
    std::cin >> inputString;

    // Push each character of the input string onto
    // the stack and add it to the queue
    for (char ch : inputString) {
        S.push(ch);
        Q.addq(ch);
    }
    bool pal = true;
    while (!Q.empty()){
        pal = Q.removeq() = = S.pop();
        if(pal==false) break;
    }
    if (pal)
        cout << "Palindrome!!" << endl;
    else cout<<"Not a Palindrom"<<endl;}
```

Implementation of queues

Static

- Queue is implemented by an array, and size of queue remains fix

Dynamic

- A **queue** can be implemented as a **linked list**, and *expand* or *shrink* with each *enqueue* or *dequeue* operation.

Static implementation

An implementation of a queue requires:

- Storage for the data
- Markers for the *front* and *rear* of the queue

An array-based implementation would need structures like:

- **`myArray`**, an array to store the elements of the queue
- **`front`**, an index to track the front queue element
- **`rear`**, an index to track the last queue element

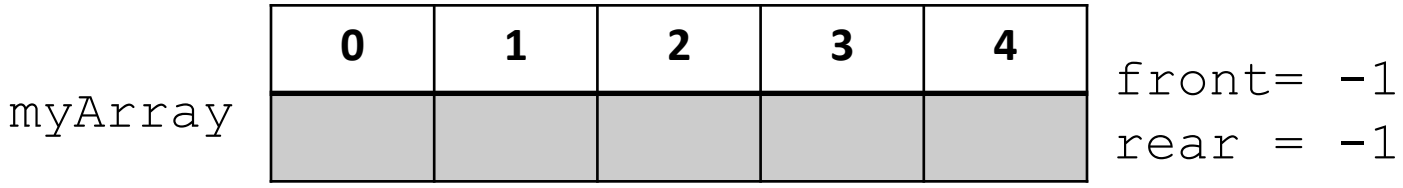
Implementation of queues

- When an item is enqueued, the rear index moves forward.
- When an item is dequeued, the front index also moves forward by one element

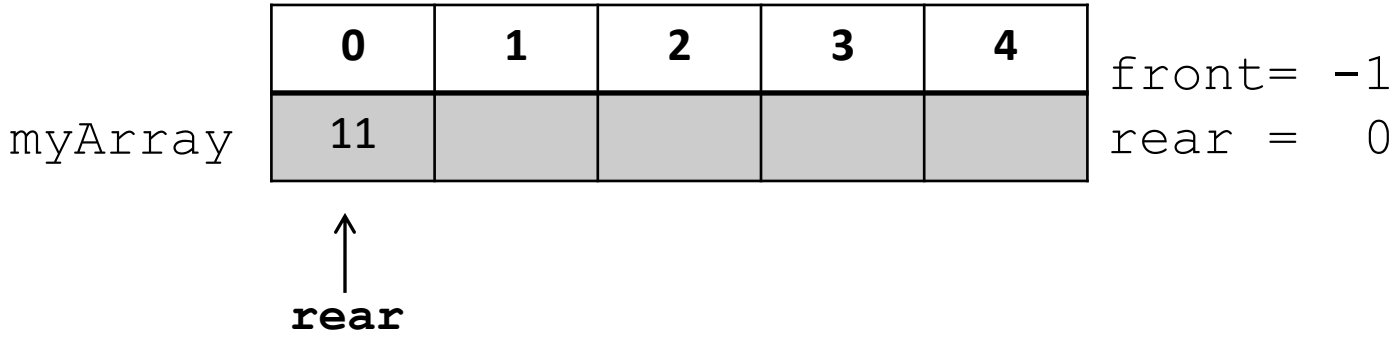
Example: **X = occupied, and O = empty**

- (front) XXXXOOOOO (rear)
- OXXXXOOOO (after 1 dequeue, and 1 enqueue)
- OOXXXXXOO (after another dequeue, and 2 enqueues)
- OOOOXXXXX (after 2 more dequeues, and 2 enqueues)

Implementation of queues

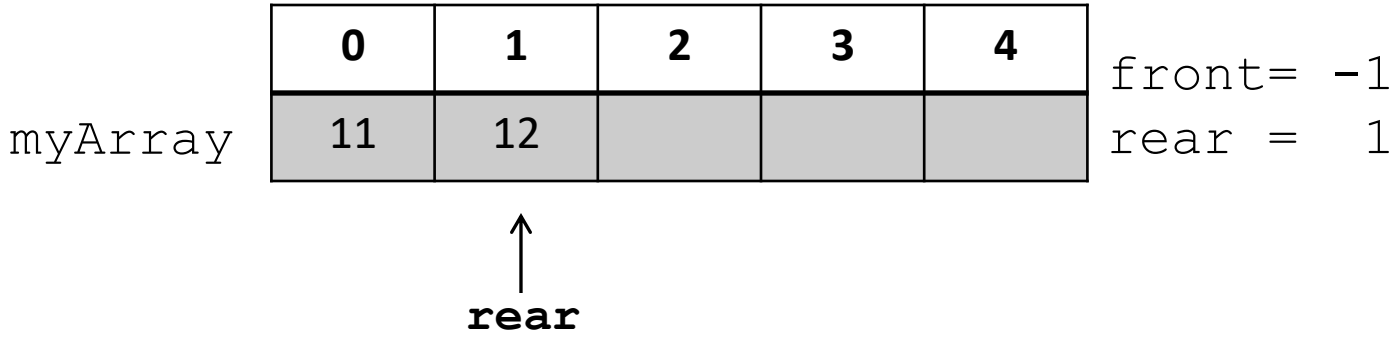



```
Q.enqueue(11);
```

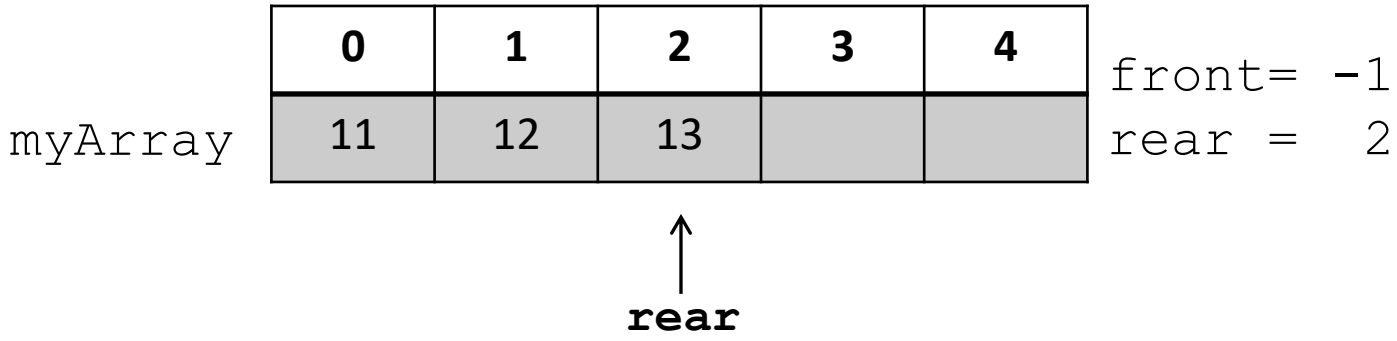


Implementation of queues

Q.enqueue(12);

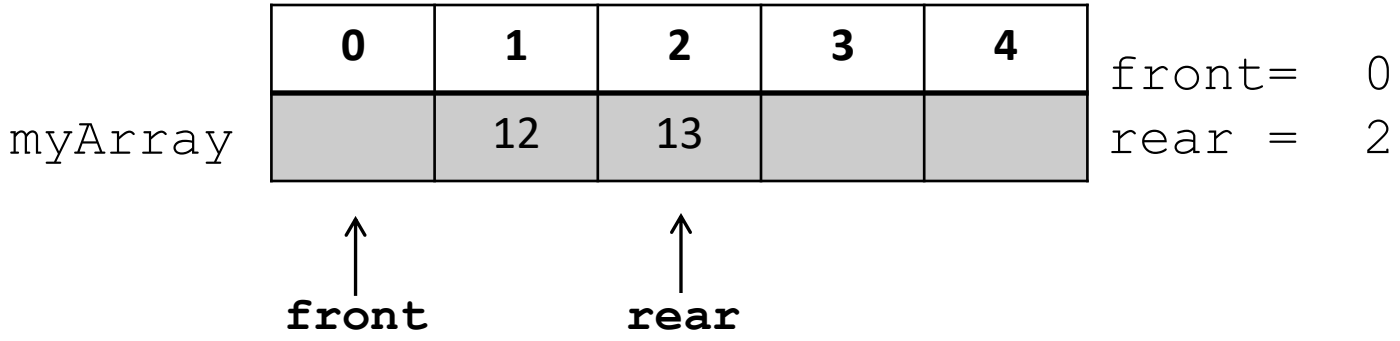


Q.enqueue(13);

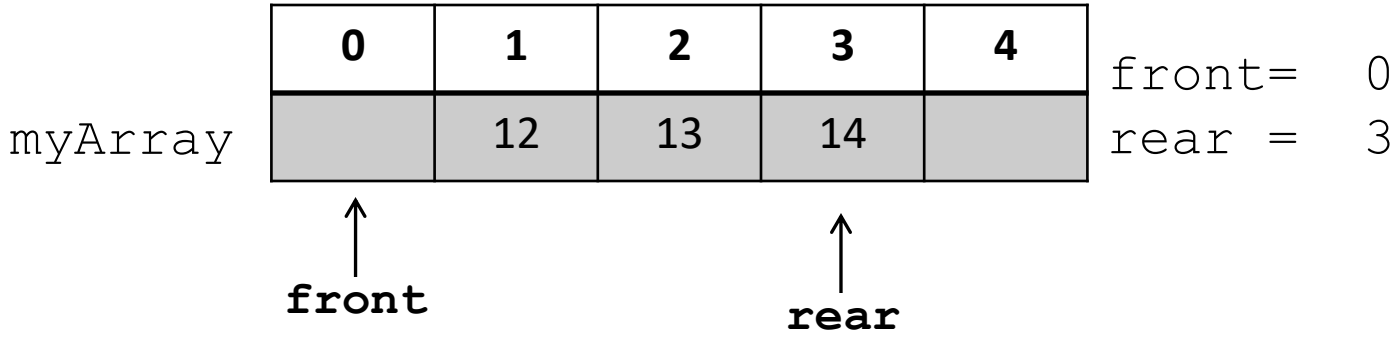


Implementation of queues

Q.dequeue()

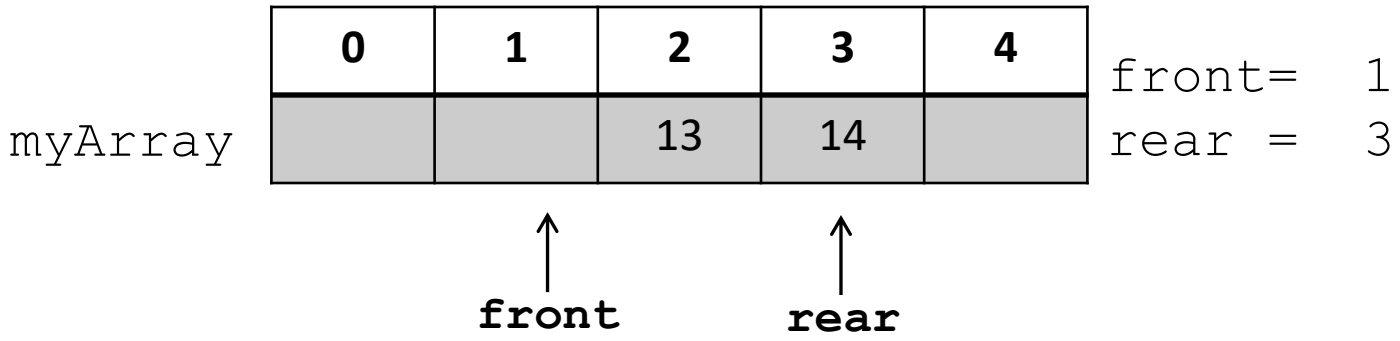


Q.enqueue(14)

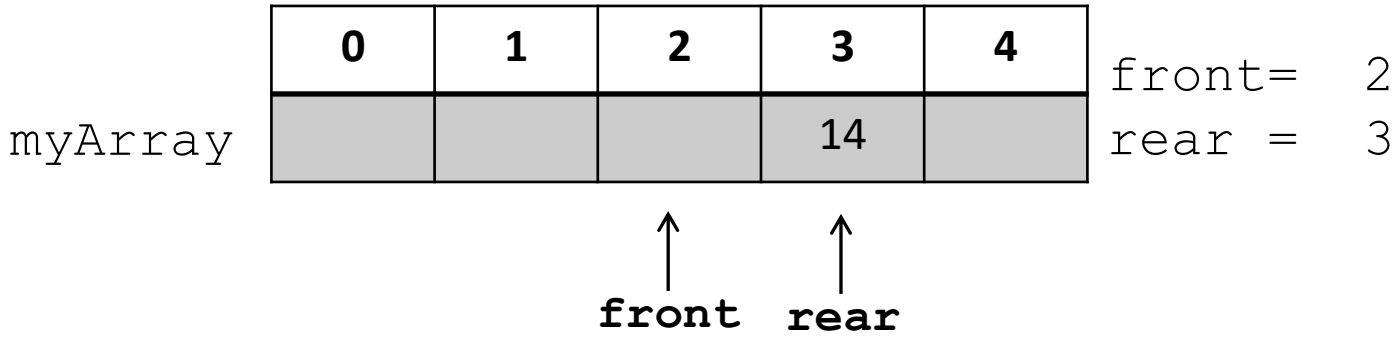


Implementation of queues

Q.dequeue()

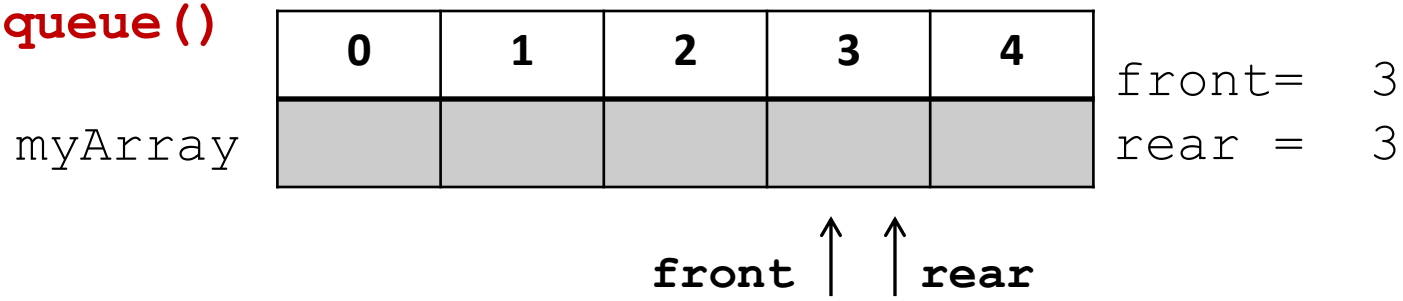


Q.dequeue()



Implementation of queues

Q.dequeue()



For *enqueue*: increment *rear* and assign value:

```
myArray[++rear] = value;
```

For *dequeue*: increment *front* and return value:

```
return myArray[++front];
```

Queue is empty when:

```
front == rear;
```

Implementation of queues

```
class Queue
{
/***** Data Members *****/
  private:
        int *myArray;
        int size;
        int front, rear;
  /***** Function Members *****/
  public:
        Queue(int);
        bool isEmpty();
        bool isFull();
        void enqueue(int value);
        int getFront();
        int dequeue();
};
```

Implementation of queues

```
Queue::Queue(int s)
{
      myArray = new int[s];
      size = s;
       front = rear = -1;}

bool Queue::isFull()
{
       if (rear == size - 1)
             return true;
       else
             return false;}
```

Implementation of queues

```
bool Queue::isEmpty()
{
      if (front == rear)
            return true;
      else
            return false;
}
```

Implementation of queues

```
void Queue::enqueue(int value)
{
      if (isFull())
            cout<<"Queue is full"<<endl;
      else
            myArray[++rear] = value;

}
```

Implementation of queues

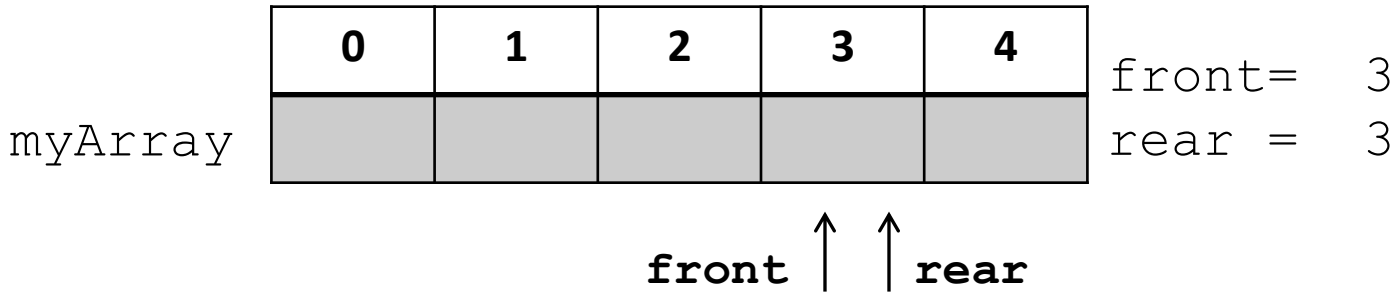
```
int Queue::dequeue()
{
      if (isEmpty())
      {
            cout<<"Queue is empty"<<endl;
            return -1;
      }
      else
            return  myArray[++front];

}
```

Implementation of queues

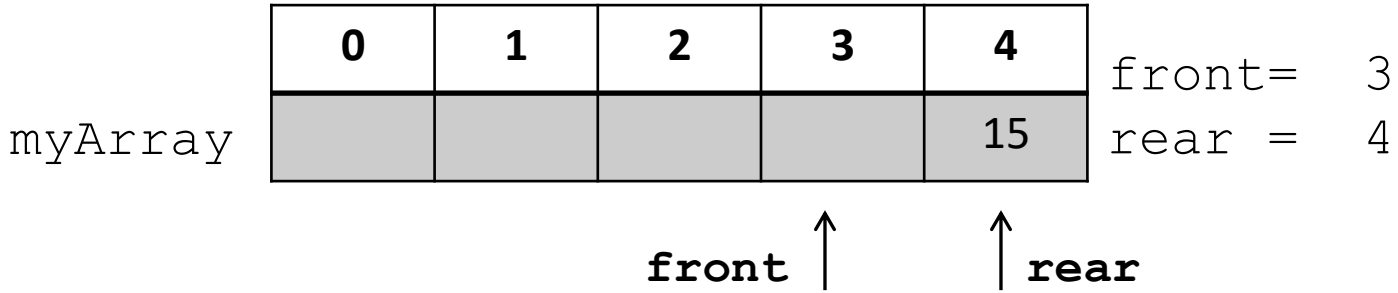
```
int Queue::getFront()
{
      if (isEmpty())
      {
            cout<<"Queue is empty"<<endl;
            return -1;
      }
      else
            return  myArray[front+1];

}
```

Previous example

Q.dequeue()

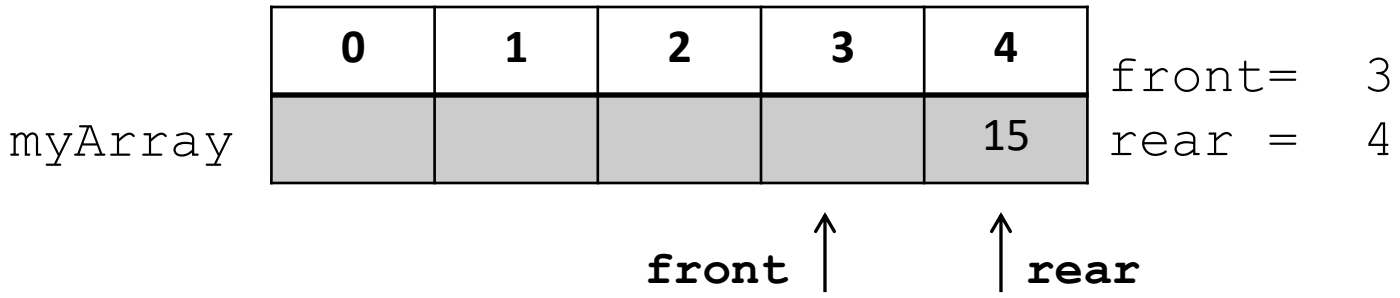


Q.enqueue(15)



Queue full?

Q.enqueue(15)




```
rear ==  size - 1; //Queue Full
```

Though the array has 4 empty slots, the program will signal that the queue is full.

Circular queue

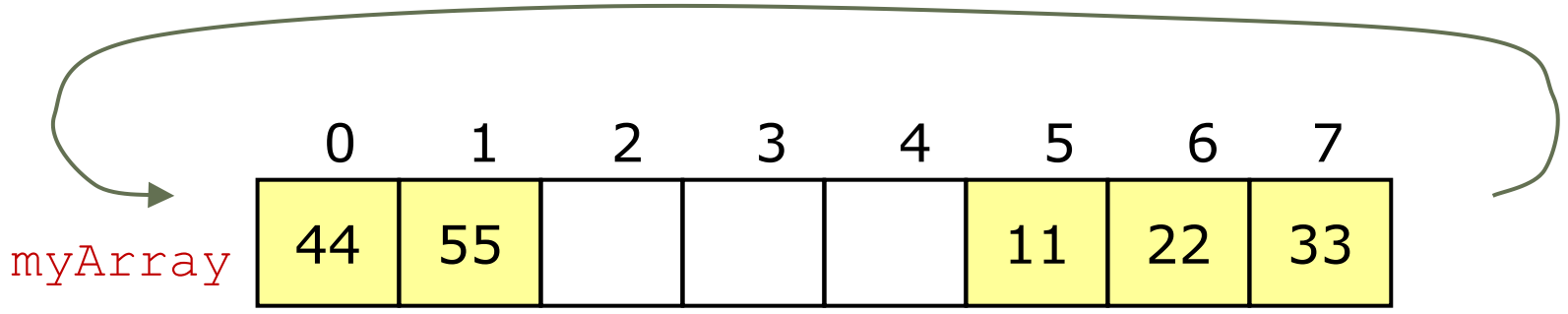
Solution: Shifting of elements?

- Shifting – Inefficient esp. when elements are large records

Would prefer to let the data "*wrap-around*"

Use a *circular array* to implement the queue

In this implementation, first position follows the last



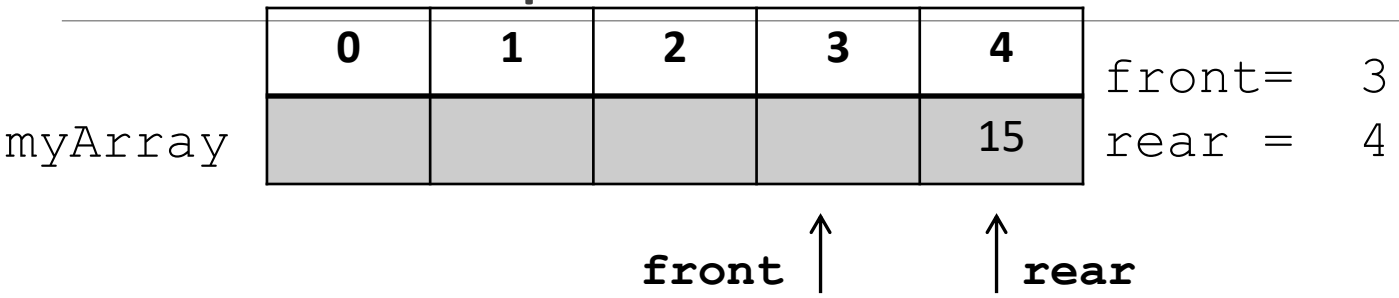
Circular queue

When an element moves past the end of a circular array, it wraps around to the beginning

Example:

- OOOOO7963
- **`Enqueue(4)`** : 4OOOO7963

Circular queue



Allow rear to wrap around the array.

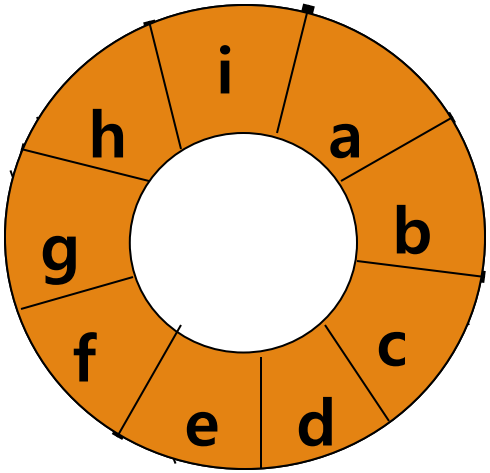
```
if(rear == size-1)
      rear = 0;
else
      rear++;
```

Equivalently:

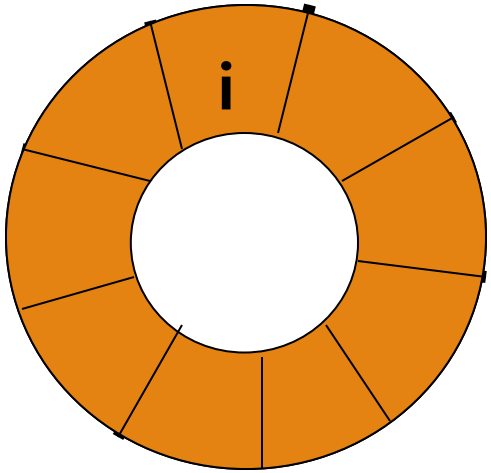
```
rear = (rear + 1) % size;
```

Analogous treatment of *front*.

Circular queue



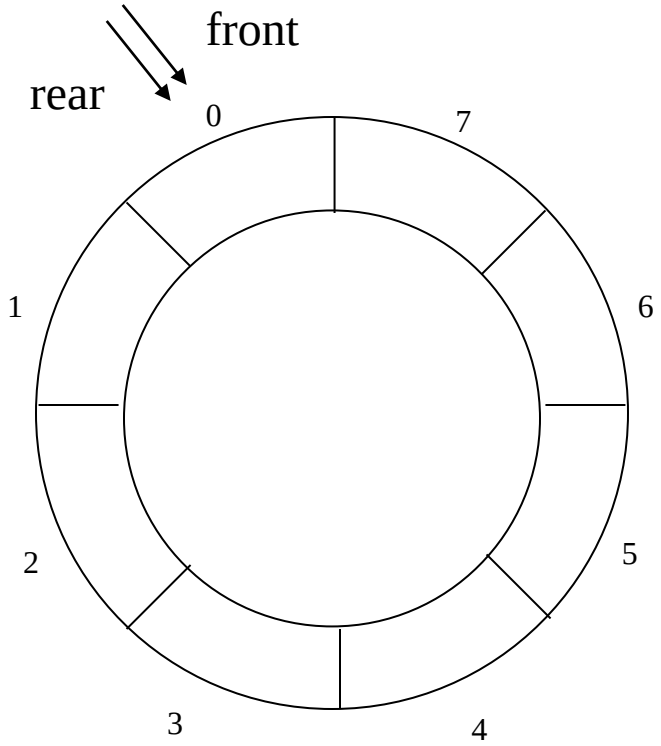
A Completely Filled Queue



A Queue with Only 1 Element

Important

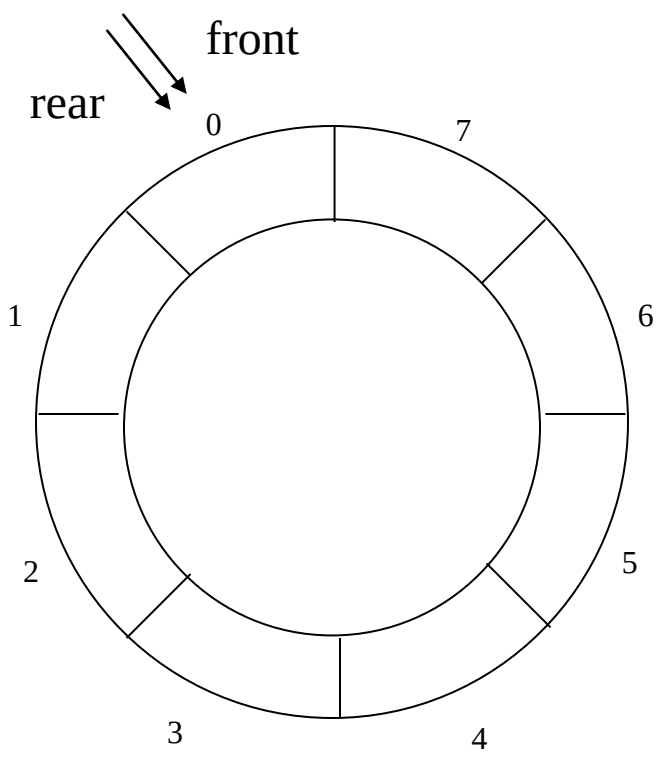
Initialize both *front* and *rear* to 0.

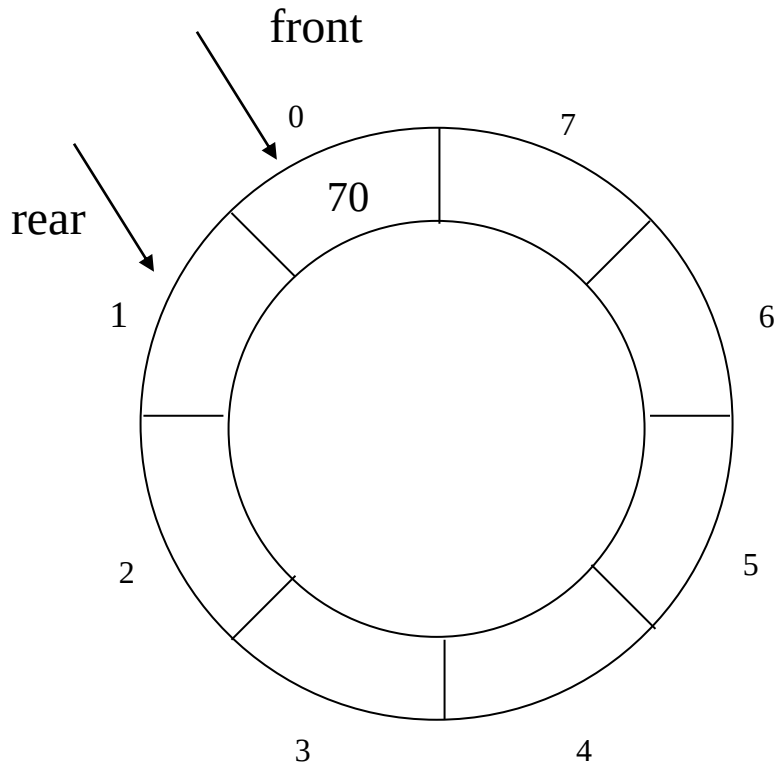



```
Queue::Queue()
{
        front = rear= 0;
}
```

```
enqueue(70);
```

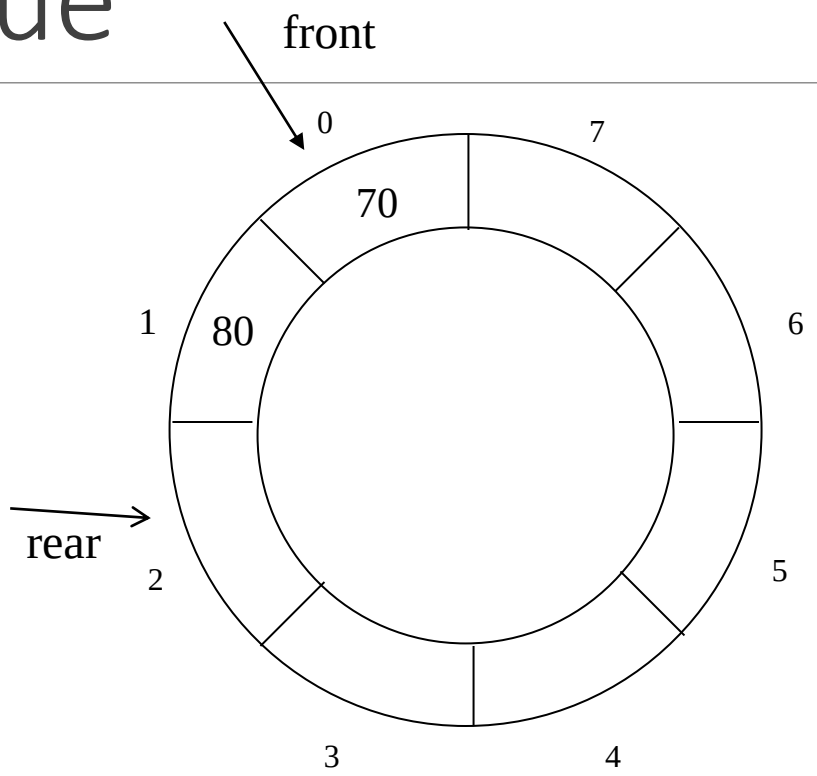
Circular Queue





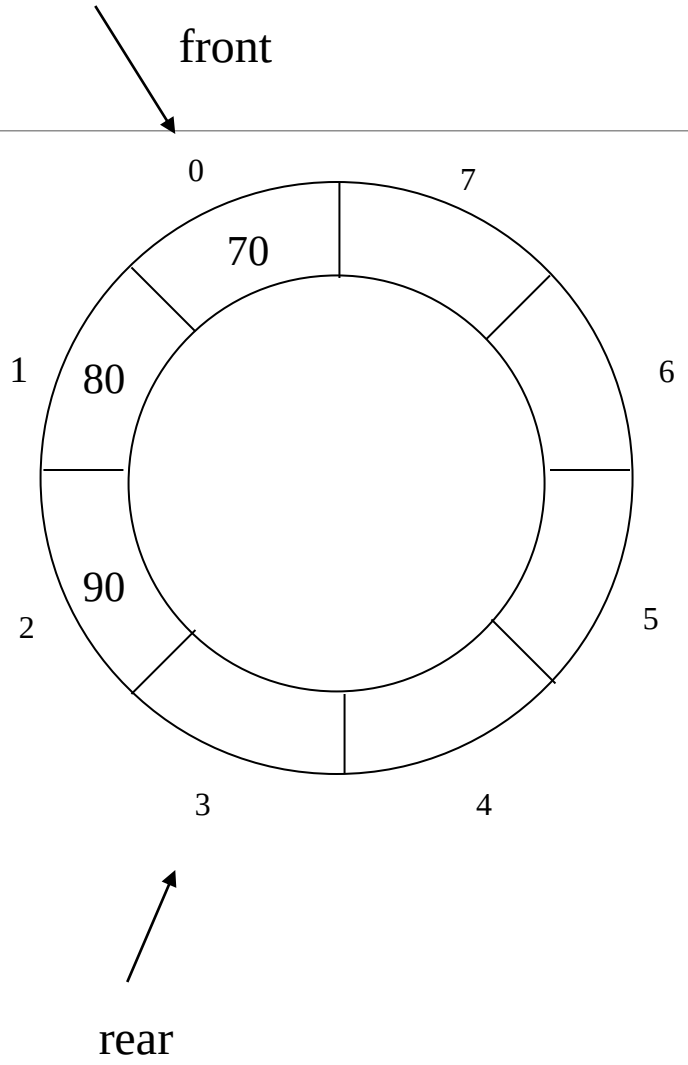
Circular Queue

```
enqueue(70);
enqueue(80);
```



Circular Queue

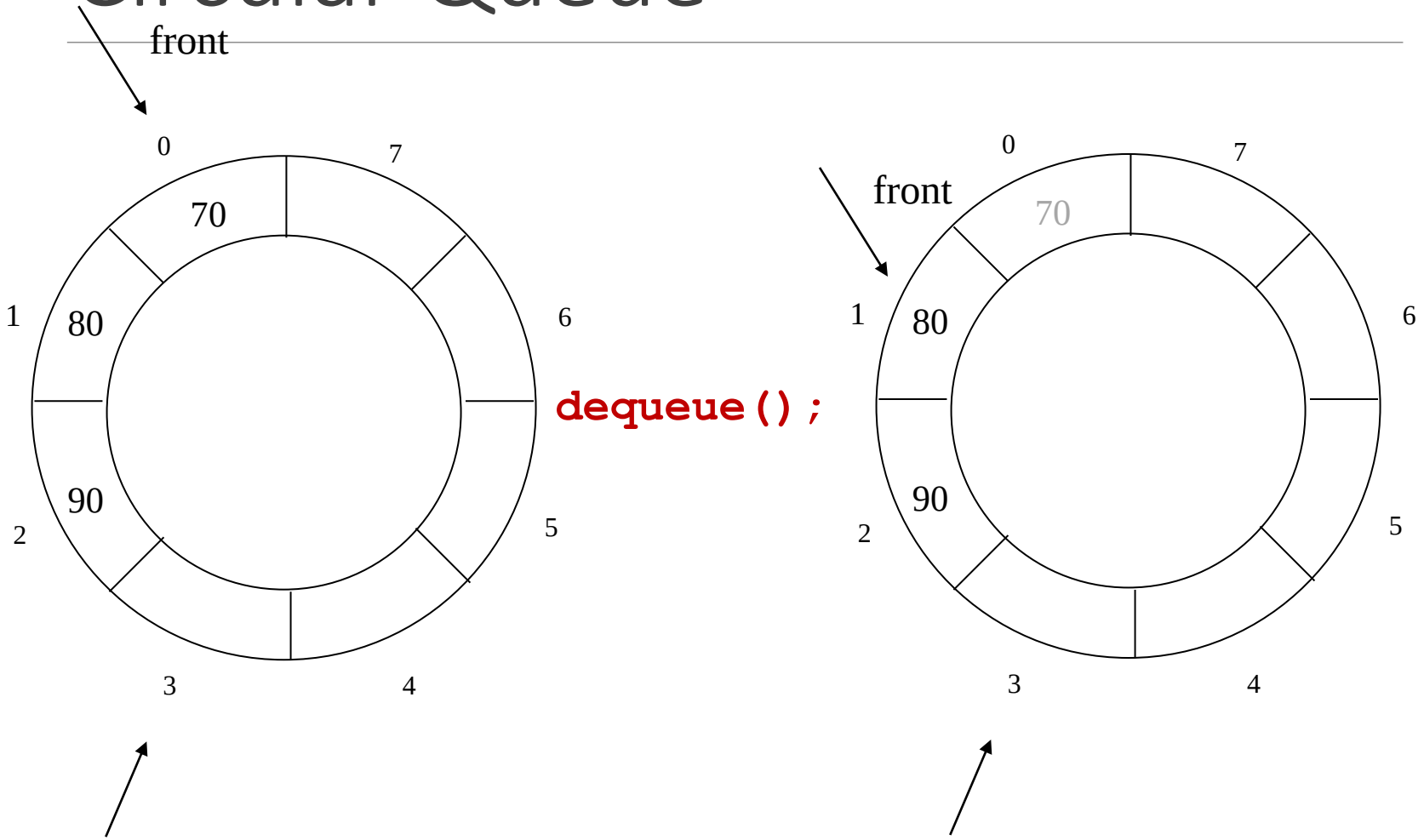
```
enqueue(70);
enqueue(80);
enqueue(90);
```



Circular Queue

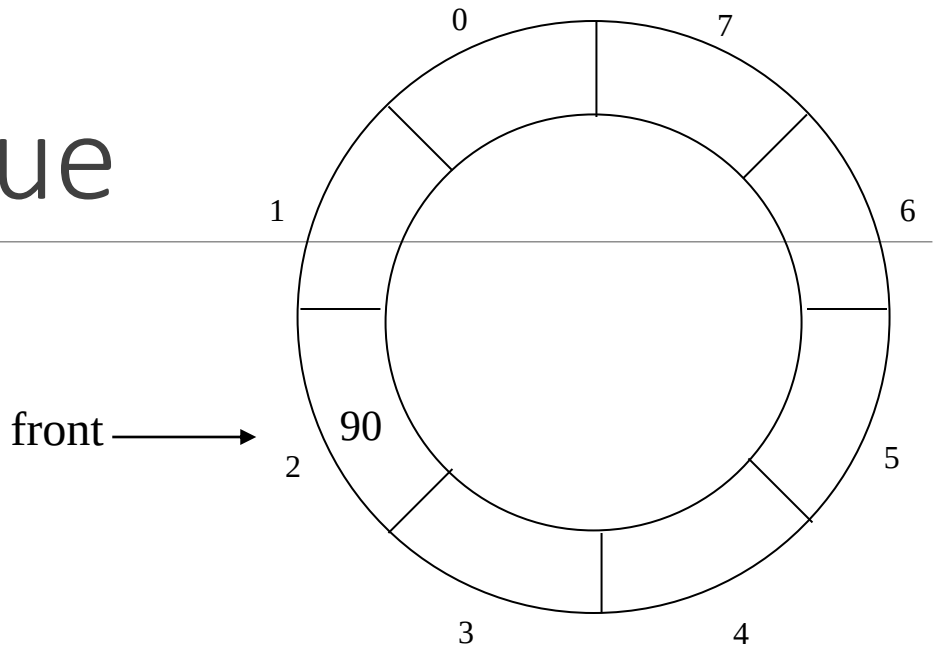
```
void Queue::enqueue(int value)
{
      if (! isFull())
      {
            myArray[rear] = value;
            rear = (rear + 1) % size;
      }
      else
            cout<<"Queue Overflow";
}
```

Circular Queue



Circular Queue

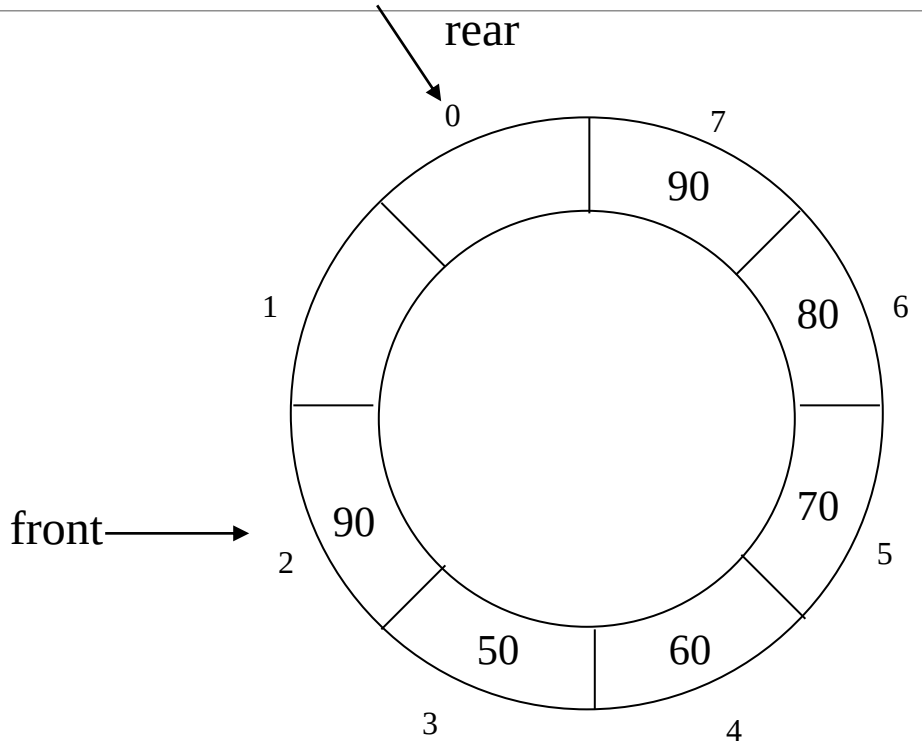
```
dequeue();
dequeue();
```




```
int Queue::dequeue()
{
   if (! isEmpty()) {
        int val= myArray[front] ;
        front = (front + 1) % size;
       return val;}
  else
        cout<<"Queue Underflow";
        return -1;
}
```

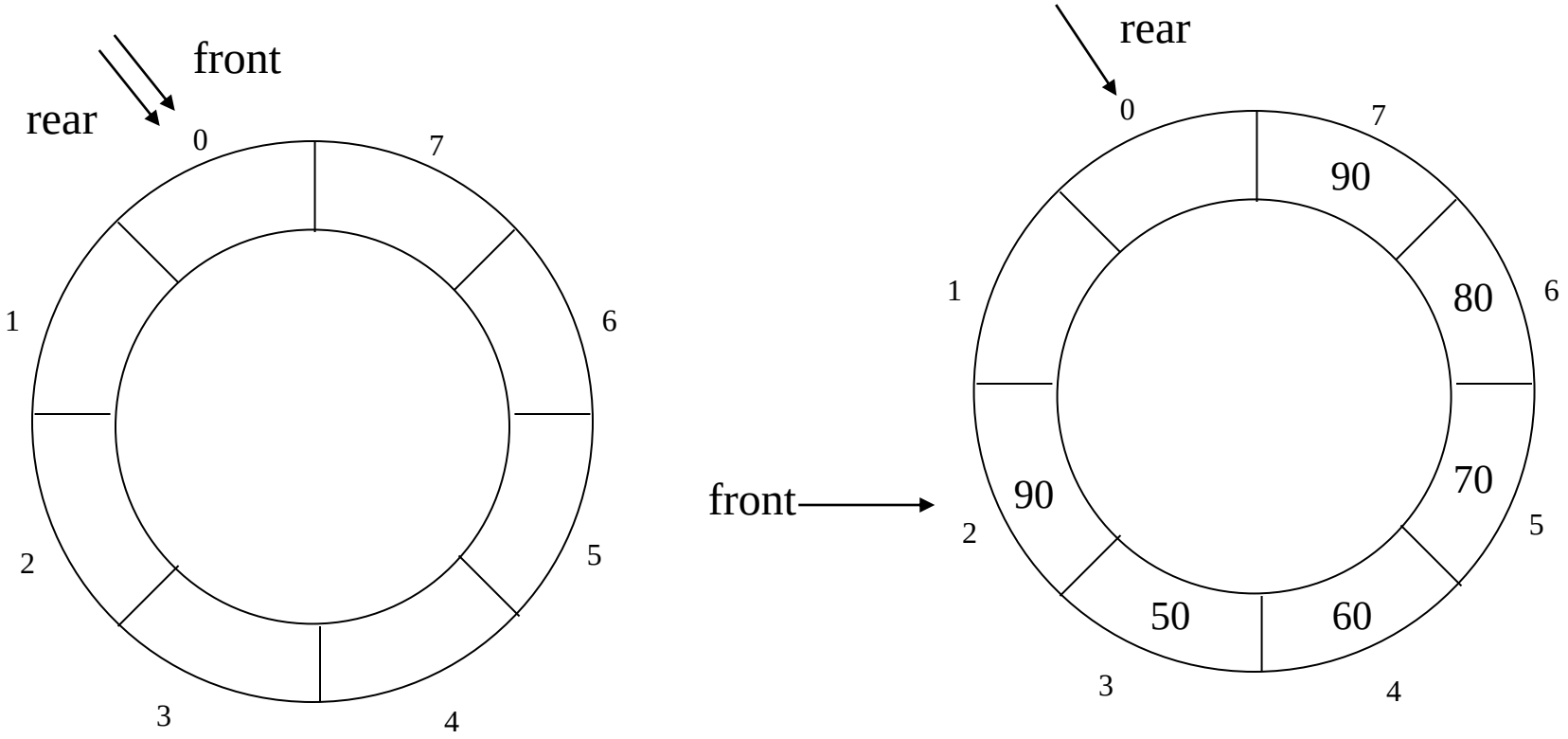
Circular Queue

```
enqueue(50);
enqueue(60);
enqueue(70);
enqueue(80);
enqueue(90);
```

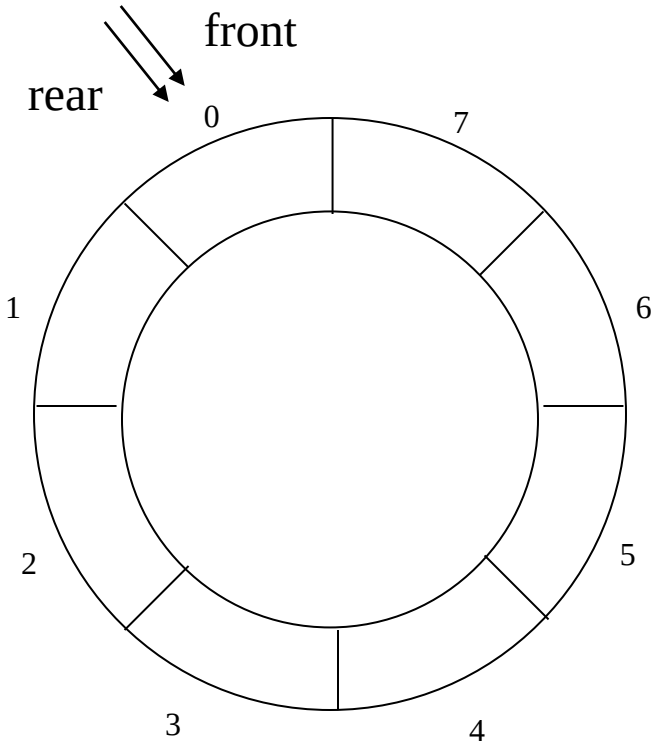


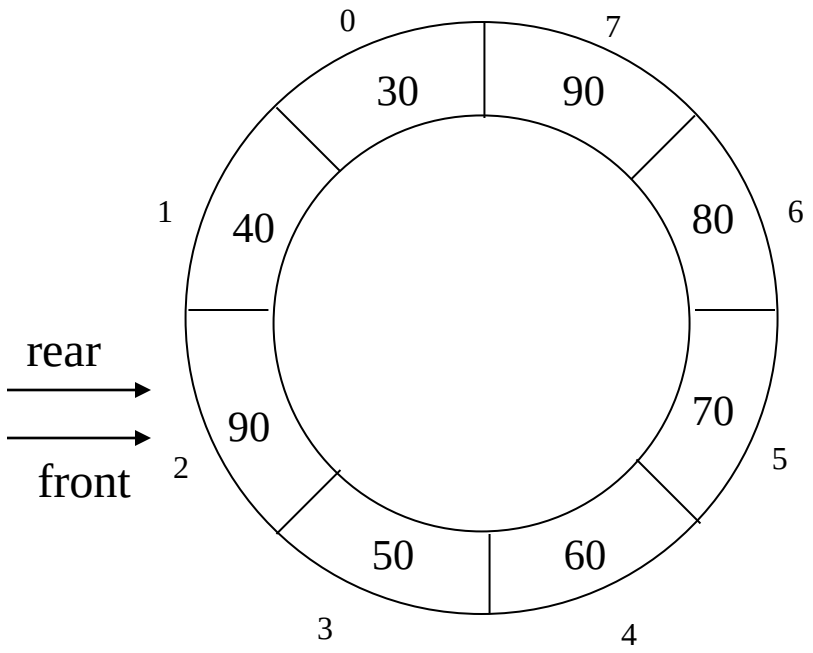
Empty vs. full queue

```
enqueue(30);
enqueue(40);
```



Empty vs. full queue






```
front == rear; //Empty or full?
```

How to distinguish b/w empty and full queues

Can be tricky

Use a counter variable to keep track of number of elements in the queue

```
class Queue
{
private:
          int *myArray;
          int front, rear, size;
          int count;
};
```

```
class Queue
{
  /***** Function Members *****/
  public:
        Queue(int);
        bool isEmpty();
        bool isFull();
        void enqueue(int value);
        int getFront();
        int dequeue();

  /***** Data Members *****/
  private:
        int *myArray;
        int front, rear, count, size;
};
```

```
Queue::Queue(int s)
{
      myArray=new int[s];
      size = s;
     front = rear = count =0;
}



bool Queue::isEmpty()
{
     if (count == 0)
          return true;
     else
          return false;
}
```

```
bool Queue::isFull()
{
      if (count == size)
            return true;
      else
            return false;
}


int Queue::getFront()
{
      if (!isEmpty())
      {
            return myArray[front];
      }
      else
            return -1;
}
```

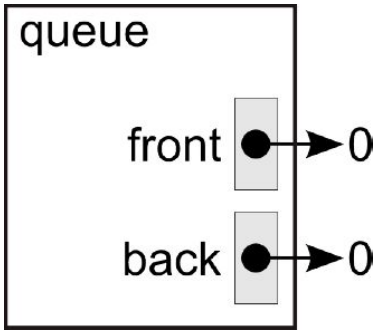
```
void Queue::enqueue(int value)
{
     if (! isFull()){
         myArray[rear] = value;
         rear = (rear + 1) % size;
         count++;
     }
}
int Queue::dequeue()
{
   if (! isEmpty()) {
        int val= myArray[front] ;
        front = (front + 1) % size;
       count--;
       return val;}
   else
         cout<<"Queue Underflow";
         return -1;
}
```

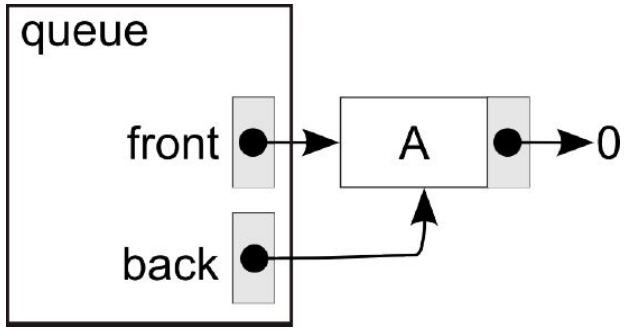
```
void Queue::display()
{
    int f=front, r=rear;
   if (! isEmpty())  {
       while(f!=r)
       {
           cout<<myArray[f];
          f = (f + 1) % size;
       }
   else
         cout<<"Queue is empty";
}
```

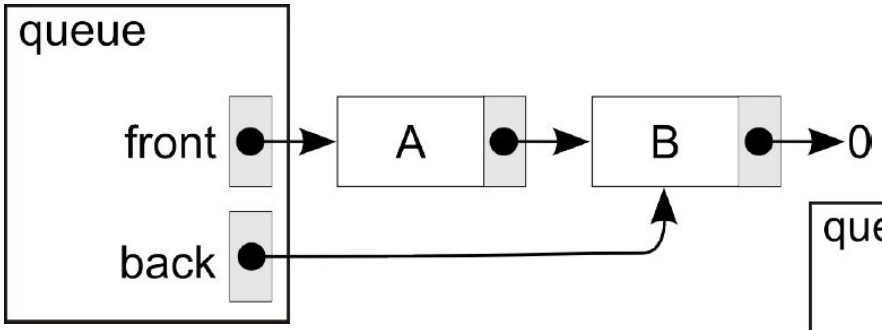
Linked List Based Implementation of Queues?

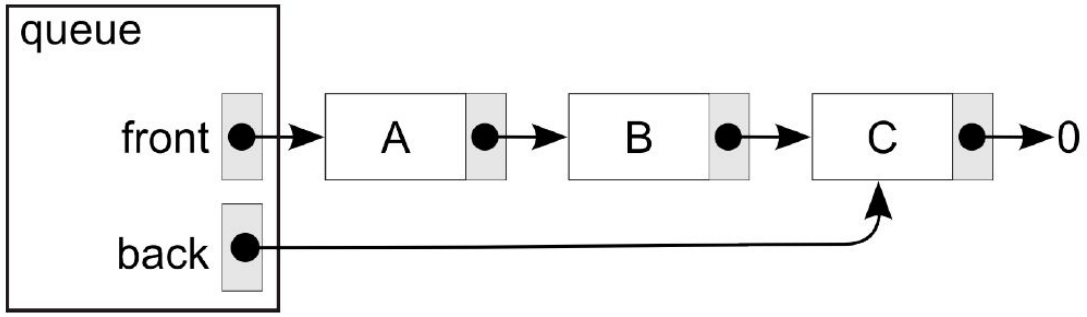
NUST SEECS

Maintain pointer to first and last nodes in a linked list.





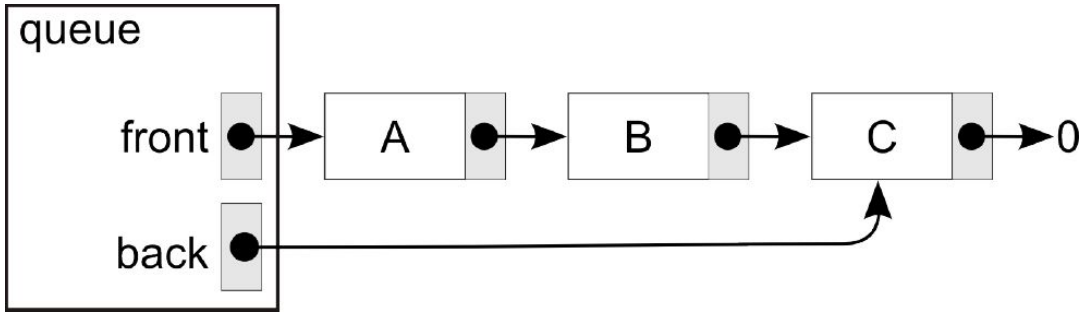


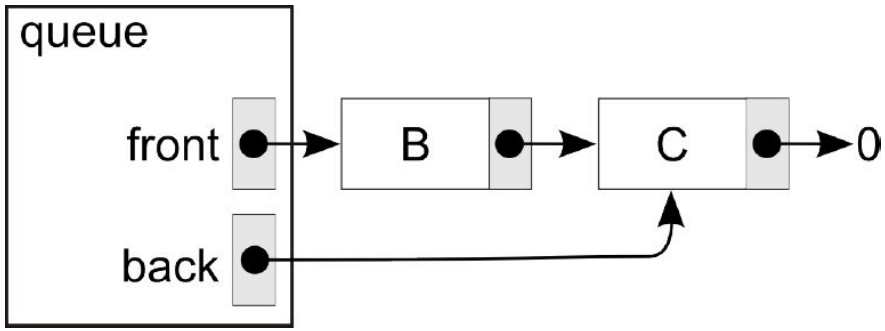


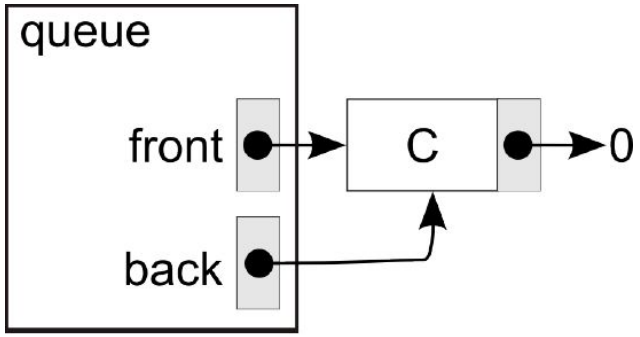
Linked List Based Implementation of Queues?

NUST SEECS

Maintain pointer to first and last nodes in a linked list.







Priority Queue

Priority queues

A priority queue is a data structure in which prioritized insertion and deletion operations on elements can be performed according to their priority values.

There are two types of priority queues

- Ascending Priority queue
- Descending Priority queue

Types of Priority Queue

- **Ascending Priority Queue**: A collection of items into which items can be inserted ***randomly*** but only the ***smallest*** item can be removed

- **Descending Priority Queue**: A collection of items into which items can be inserted ***randomly*** but only the ***largest*** item can be removed

Priority queues

In both the above types, if elements with equal priority are present, the FIFO technique is applied.

Both types of priority queues are similar in a way that both of them remove and return the element with the highest **"Priority"** when the function **`remove()/dequeue()`** is called.

- For an ascending priority queue item with *smallest value* has *maximum priority*
- For a descending priority queue item with *highest value* has *maximum priority*

Priority Queue Issues

In what manner should the items be inserted in a priority queue?

- **Ordered:** Retrieval is simple, but insertion will become complex
- **Arbitrary:** Insertion is simple but retrieval will require elaborate search mechanism

In what manner should the queue be maintained when an item is removed from it?

- Emptied location is kept blank (how to recognize a blank location?)
- Remaining items are shifted

Double-ended-Queue (Deque)

A Deque* is a queue in which elements can be added or removed at either end but not in the middle.

*Pronounced as '*Deck*'

Double-ended-Queue (Deque)

Common Operations:

- **`Left_Enqueue()`**
- **`Right_Enqueue()`**
- **`Left_Dequeue()`**
- **`Right_Dequeue()`**
- **`Get_left()`**
- **`Get_right()`**

Types of deque

Input-restricted Deque

- Elements can be added only at one end
- Elements can be removed at both ends

Output-restricted Deque

- Elements can be added at both ends
- Elements can be removed only at one end

Deque as stack and queue

As Stack

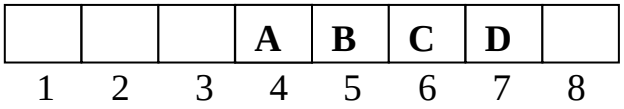
- When insertion and deletion are made at same end

As Queue

- When insertions are made at one end and deletions at the other end

Deque Implementation

A **Deque** is maintained by a circular array with pointers LEFT and RIGHT, which point to the two ends of the Deque.



LEFT = 4
RIGHT = 7