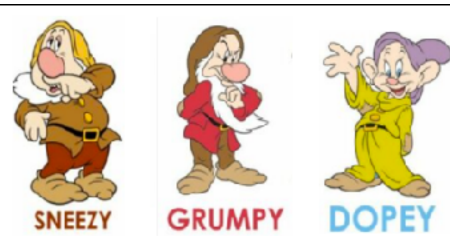
Class 27 Notes
The Queue

A queue is a FIFO (First In – First Out) list. A queue is nothing more than a waiting line. Suppose you want to buy a movie ticket. You get in the line at the end and you wait your turn until you reach the front.
Here is the formal definition:
Definition:
A **queue** is an ordered list of data into which data may be inserted at one end (the rear) and removed from the other end (the front).

The basic operations are :

- Insert(x) → places x at the end of the queue
- remove() → removes and returns the first item in the queue
- peek() → returns the first item in the queue but does not remove it
- size() → returns the number of items in the queue
- empty() →returns true id the queue is empty; otherwise false

Here is an example of how a queue works:

insert("DOPEY") insert("DOC") insert("HAPPY")	(front)  (rear)
remove() remove()	(front)  (rear)
insert("SNEEZY") insert("GRUMPY") insert("DOPEY")	(front)  (reart)
remove()	(front)  (rear)

How does a queue differ from a stack?
With a stack all access is at one end → the top
With a queue there is access in two places → the front and the rear

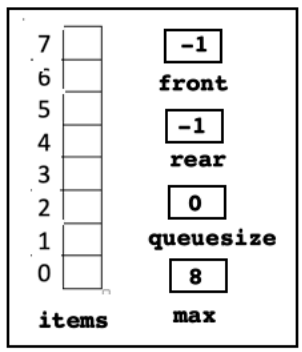
Like a stack we will have to implement a queue for ourselves. (The FIFO list implementation you did for homework was a bit inefficient)

I will implement a queue two different ways:

1. an array implementation
2. a linked (node) implementation

The array implementation:

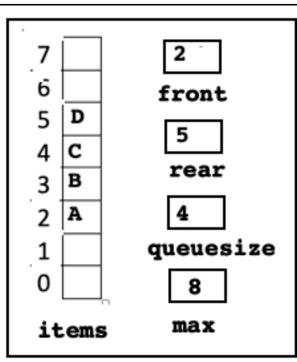
An empty queue object will look like this.



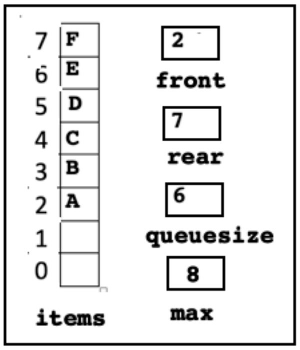
,

- An array,***items***, tholds the data,
- front holds the index of the front item in the queue → initially -1
- rear holds the index of the last item in the queue→ initially -1
- queuesize holds the number of items in the queue → initially 0
- max holds the maximum size of the queue→ 8 in THIS picture

Here is a picture of a queue with data: D C B A, where A is at the front and D is last:

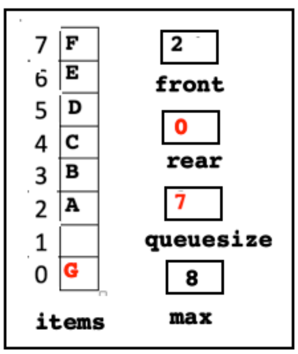
	Queue is D C B A → A is at the front front holds index 2 (position of A) rear holds index 5 (position of D) queuesize is 4 (just 4 items in the queue) max is 8

Now suppose we insert two more items: insert(E) and insert (F). The queue would have the following form:

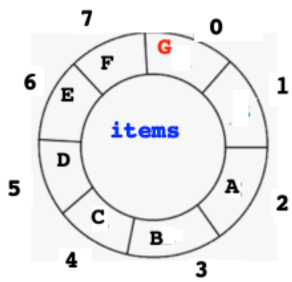


The queue now holds 6 items
E and F were added at the rear. A is at the front.
So the rear == 7
And front == 2
queuesize == 6
and max is 8

Now suppose we want to add another item. Technically, there are two available positions. We could shift all the data down to make room for another item. But that would be inefficient. (Actually, that is what you did on your homework when you built a FIFO list).
Instead we think of the array as “circular.” We will put the new item in position 0. In other words, after position 7 go back to position 0. So after insert(G) the picture is:

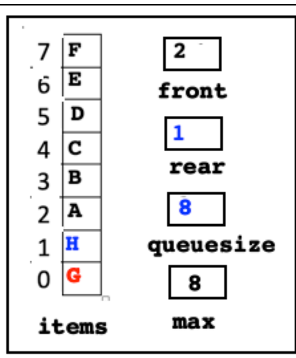


Notice G is is stored in items[0]

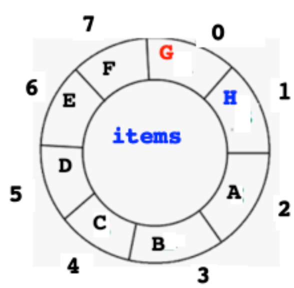


You can picture the array, items, as circular. Items[7] is followed by items[0]. front ==2; rear == 0

Now suppose we want to insert H. H would be stored in items [1] :

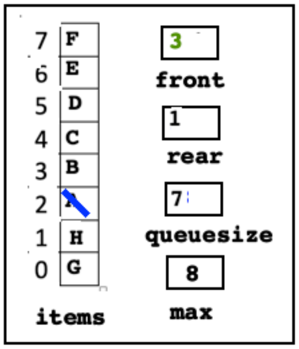


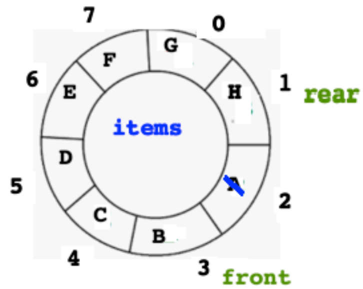
The queue is full now
front is at 2 (A) rear is at 1 (H)



The queue is full now
front is at 2 (A) rear is at 1 (H)

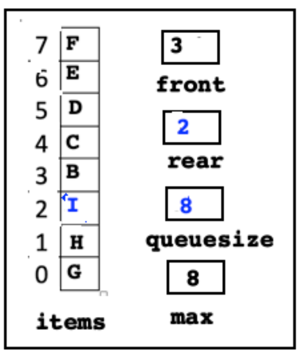
Now suppose we do a remove() operation. The remove() operation takes items from the front. Since front == 2. The remove() deletes A from the queue and the new front is at position 3. So the queue (**front to rear**) is B C D E F G H (B is first, H is last)



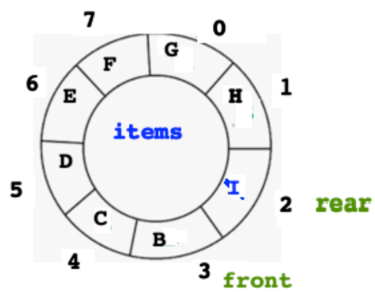


The first item (B) is at position 3. And the last is at position 1.

Suppose that we want to insert(I). It would be placed after H at items [2]. And again the queue would be full. The indices are 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 etc. circular



B C D E F HG H I



Now let's see how to implement a queue using a (circular) array.

```
public class Queue<E>   // E is a generic, as with a stack
{
        private int front, rear, queueSize, max;
        private E[] items;


        public Queue()  //Default Constructor sets maximum size 10 10
        {
                front = rear = -1;
                items = (E[]) new Object[10]; // NOTE the cast, as with the stack
                max = 10;
                queueSize = 0;
        }

        public Queue(int max)  //One argument constructor sets the maximum to max
        {
                front = rear = -1;
                queueSize = 0;
                this. max = max;
                items = (E[]) new Object[max];

        }

        public int size()
        {
                return queueSize;
        }

        public boolean empty()
        {
                return queueSize == 0;
        }
```

```
	public void insert(E x)  //
	{
		if (queueSize == max)
		{
			System.out.println("Queue Overflow");
			System.exit(0);
		}
		if (queueSize == 0)  // if initially empty, and front and rear are both -1
			 front = rear = 0;
		else
			rear = (rear + 1) % max; // move rear up one, circling around if nec.
			 // if max is 10 and rear is 9 then (rear+1)%10 = 10%10= 0  → back to 0
			// if max is 10 and rear is 3 then (rear+1)%10 = 4%10 = 4 → the next slot

		items[rear] = x;
		queueSize ++;
	}
	public E remove()
	{
		if (queueSize == 0)
		{
			System.out.println("Queue Underflow");
			System.exit(0);
		}
		E temp = items[front];  // to return the value
		queueSize --;
		if (queueSize == 0)  // if the queue is now empty
			front = rear = -1;
		else
			front = (front + 1) % max;
			 // move to the next position  but wrap around if necessary
		return temp;
	}
	public E peek()
	{
		if (queueSize == 0)
		{
			System.out.println("Queue Underflow");
			System.exit(0);
		}
		return items[front];
	}
}
```

Example:

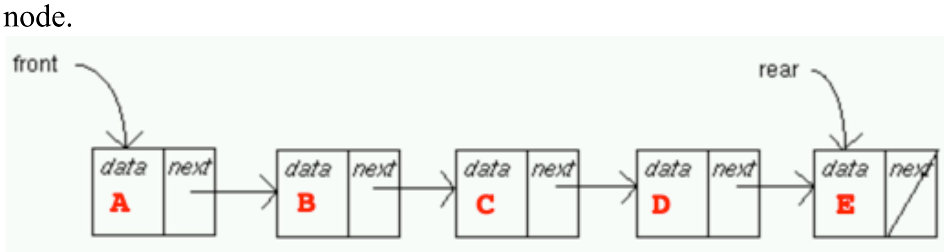
```
public class ExampleQueue
{
        public static void main(String[] args)
        {
                    1.  Queue<String> q = new Queue<String> (8) ; // max is 8
                    2.  q.insert("A");
                    3.  q.insert("B");
                    4.  q.insert("C");
                    5.       // queue is A  B  C →   A  is at  the front
                    6.  System.out.println(q.remove());
                    7.  System.out.println(q.peek());
                    8.  q.insert("D");
                    9.  System.out.println("The remaing elements are: ");
                    10. while (!q.empty())
                      System.out.println(q.remove());
        }
```

Statements 1-3 add A B and C to the queue→ front to rear is “A” “B “ “C”
Statement 6 removes the front item (“A”) and prints it → queue is now B C
Statement 7 peeks at the front item(B) and prints it → peek does not remove the item
Statement 8 adds D to the end of the cur → so now the queue is “B” “C” “D”
Statement 10 removes each item until empty, remove is from front → “B” “C” “D”

So the output is
A
B
B
The remaining elements are:
B
C
D

Dynamic or linked implementation

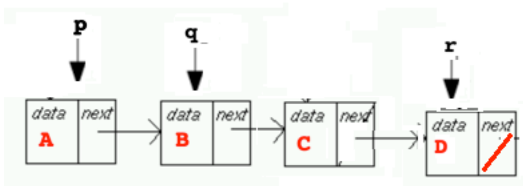
Like the stack , you can implement a queue with a chain of nodes.
Here is a picture. Notice there are two references: *front* and *rear*. Like the stack implementation, each node has two fields: *data* and *next*. And, *next* holds the address of another node.



From **front to rear** the queue is A B C D E

But before I build a Queue class, here is a little review of Nodes

Suppose you have a chain of nodes that looks like this:



- p is the address of the first node; the "whole node"
  System.out.print(p) prints an address such as Node@1cad7d80
  You really never need to know the address
- q is the address of the second node
- r is the address of the fourth node

- p.data stores "A"→ the value stored in the data field
  System.out.print(p.data) prints A
- q.data stores "B"
- r.data stores "D"
  r.data = "X" changes the string in the data field to "X"

- p.next is the address of the "next node", that is the node storing "B" .
  Notice that p.next == q. p.next and q have the same value→ the address of the 2nd node
- q.next holds the address of the third node in the list
- r.next is null

Now look at the picture

- q.next.data is "C" → the data field of the node after node(q)
  (q.next references the third node, and its data is "C")

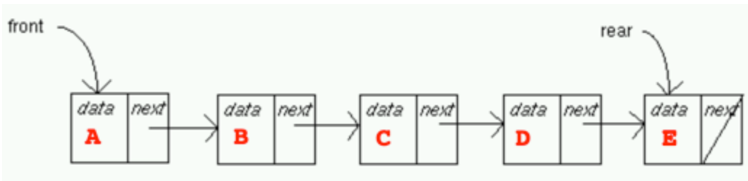
- The statement p = p.next moves p down one node in the list.

- The statement r = r.next makes r null

The Queue class using a chain of Nodes

Node will be a private inner class. This is exactly what I did with a Stack.
The **data** field holds information the **next** field links or points to another node.
So here is a(partial) Queue class with just the inner Node class and the global variables:

- front→ references the first node in the queue and
- rear→ reference the last node



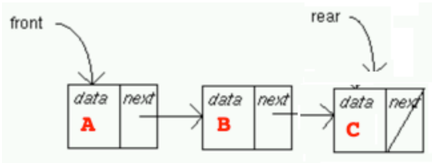

```
public class Queue<E>
{
        private class Node      // a private inner class, same as Stack
        {
                private E data;
                private Node next;  // holds the address of another ("the next") node

                public Node()   // default constructor
                {
                        data = next = null;
                }
                public Node(E x) // one argument constructor, places x in the data field
                {
                        data = x;
                        next = null;
                }
        }
        Node front, rear;       // front holds the address of the first node, real the last
        int queueSize;

        public Queue()                          // default constructor creates empty queue
        {
                front = rear = null;            // initially front and rear point to nothing
                queueSize = 0;
        }
         // the  queue methods go here
        public boolean empty()
        public int size()
        public void insert(E x)
        public E remove()
        public E peek()
}
```

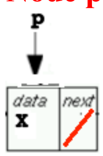
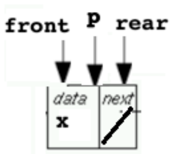
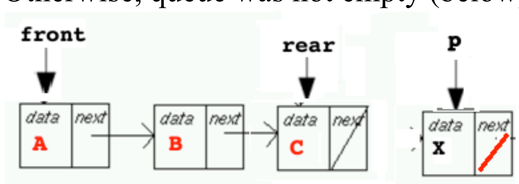
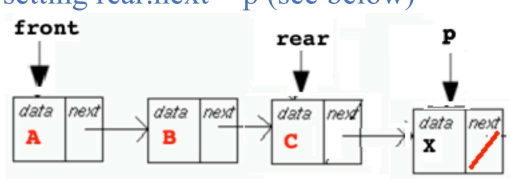
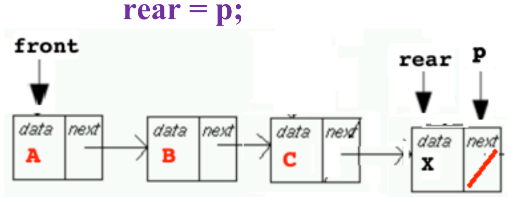
Now I will implement each of the methods that manipulate a queue

Suppose that the queue looks like this (for example)



Or is perhaps the queue empty (in which front and rear are both null)

Description	**The code is color coded with the description**

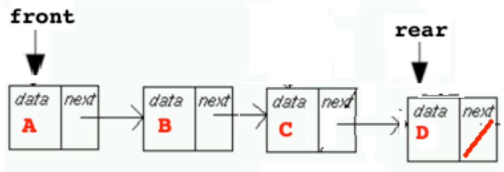
Insert(E x):
Get a new node :
Node p = new Node(x). See pic below

If the queue was empty. Set front and rear to p (the new node) and the queue has just one node. See pic below.

Otherwise, queue was not empty (below)

Attach node(p) to the end of the queue by setting rear.next = p (see below)

And finally move rear to the last node:
rear = p;

And increment queueSize
	public void insert(E x)
	{
	Node p = new Node (x);
	
	If (queueSize == 0)
	front = rear = p;
	else
	{
	rear.next = p;
	rear = p;
	}
	queueSize++;
	}

<table>
<tr><td>

remove()

If the queue was empty, you cannot remove anything. Issue a message and exit.

Now assume the queue is not empty:



</td><td>

```
public E remove()
{
    if (queueSize == 0)
    {
        System.out.println("Queue Underflow");
        System.exit(0);
    }
```

</td></tr>
<tr><td>

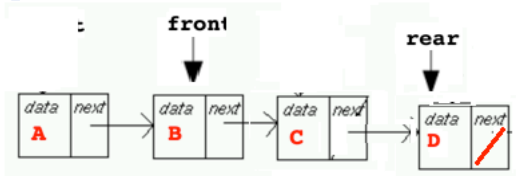
Place the data stored in the first node (front) into a temp so that it can be returned. This data is stored in front.data. So set temp = front.data

</td><td>

```
 E temp = front.data;
     // save value to return
```

</td></tr>
<tr><td>

Move front up to the next node in the queue : front = front.next. Pic below



(The garbage collector will take the original front node)

Reduce the size

</td><td>

```
front = front.next;
    // change front
```

```
queueSize-- ;
```

</td></tr>
<tr><td>

If the queue is now empty,
set front = rear = null
(Actually, front would have been already set to null by the blue code and you only need to set rear)

</td><td>

```
if( queueSize == 0)    // if queue is empty make
      front = rear = null;
```

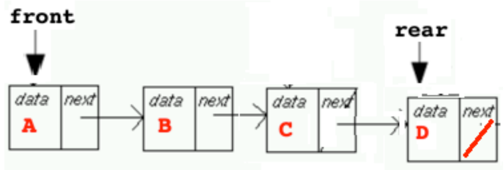
</td></tr>
<tr><td>

Return what was removed and stored in temp

</td><td>

```
return temp;
}
```

</td></tr>
</table>

peek()
 	
If the queue was empty, you cannot peek() at anything. Issue a message and exit.
 	
Now assume the queue is not empty:

Return the data in the first node. This is front.data. Se the above pic.
	public E peek()
	{
	    if queueSize == 0)
	    {
	      System.out.println("Queue Underflow");
	      System.exit(0);
	    }
	
	    return front.data;
	
	}

Size()
    Return queueSize
	public int size()
	{
	    return queueSize;
	}

Empty()
the statement
  queueSize == 0
will be either true or false
	public boolean empty()
	{
	    return queueSize == 0;
	}

Summary:

We have looked at two ways to implement a queue: as a circular array and as a linked chain of nodes.

The advantage of the array implantation is that it is very efficient. However an array cannot expand. The linked implementation is not limited in size (unless you run out of space) but uses more memory. Every Node stores not only data but an address.

With stacks we also used an ArrayList implementation. But using an ArrayList would be a pretty inefficient implementation for a queue.

Adding to the end of an ArrayList is no problem so insert(..) is fine, but every time an item is removed from the front, an ArrayList shifts all the items to fill the gap. If the queue has one million data, that's quite a lot of movement of data for each remove() operation. Pretty inefficient.

A deque

A double ended queue(a ***deque***) is an ordered list of data such that insertions and deletions can occur at both the front and the rear.

In other words a deque is a queue with additional operations that allows insertions into the front of the deque and removals from the rear.
The operations are:

- void insertFront(E x)
- E removeFront() → **works as remove() for a queue**
- void insertRear(E x) → **works as insert (…)for a queue**
- E removeRear()
- E peekFront() → works **as peek() for a queue**
- **E peekRear()**
- int size()
- boolean empty()

The three red operations are the new ones.

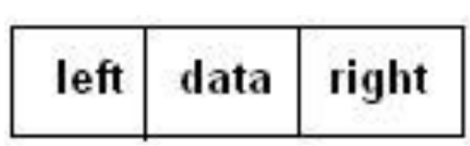
Example :The operations

insertFront(A) → A
insertFront(B) → B A
insertRear(C) → B A C
insertFront(D) → D B A C

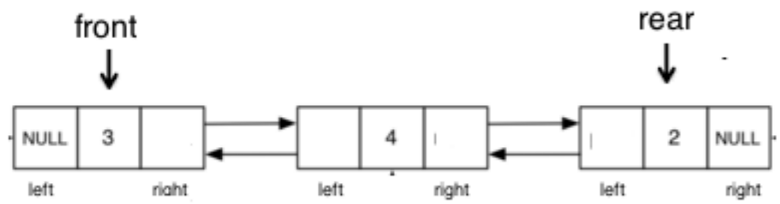
creates a deque that looks like: D B A C (front to rear)

removeFront() leaves the deque as B A C
and removeRear() as B A

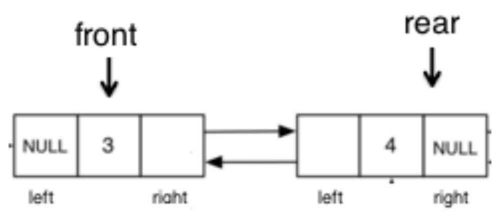
You can implement a deque as a circular array and also as a chain of nodes.
However, unlike an ordinary queue a Node will have two reference fields :
left and right (as well as a data field).



The ***left*** field points to a node to the left (or is null); the ***right*** field points to a node to the right (or is null).



Notice that the removeRear() operation leaves the deque as



So moving the rear pointer to the left is precisely why we use nodes with two reference fields.

What is the reason for the node with two reference fields?