The Stack ADT

A Stack is a collection of objects inserted and removed according to the Last In First Out (LIFO) principle. Think of a stack of dishes.

Push and **Pop** are the two main operations

Browsers, while displaying a new webpage, **push** the address of the current page into a stack. The address of the previous page can be **popped** out of the stack.

Think of the *undo operation* of an editor. The recent changes are pushed into a stack, and the undo operation pops it from the stack.

An array based stack implementation

Main update methods:

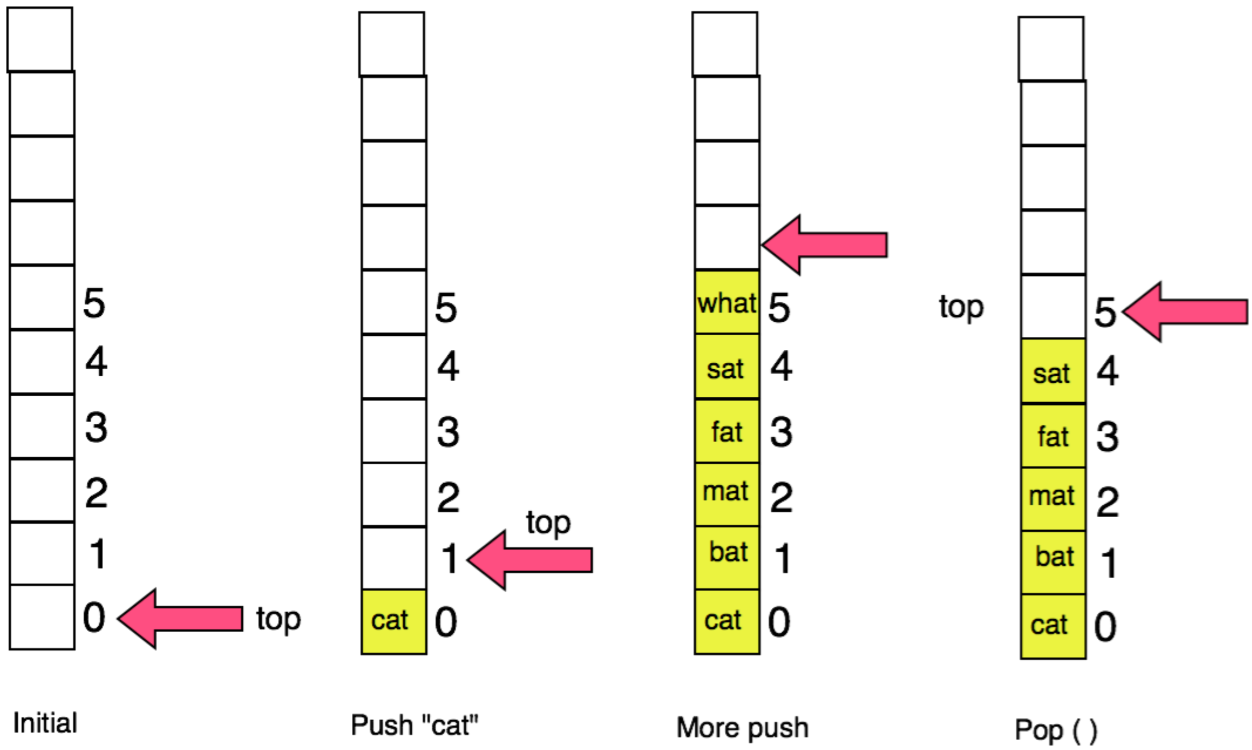
Push (e)

Pop ( )

Additional useful methods

Peek ()	Same as pop, but does not remove the element
Empty()	Boolean, true when the stack is empty
Size ()	Returns the size of the stack

```
public class Stack {
}
    public  Stack {
    }
    public Boolean empty() {
    }
    public void push (String str) {
    }
    public String pop() {
    {
    public String peek ( ) {
    }
}
```


5
4
3
2
1
0
top
Initial
cat
top
Push "cat"
what
sat
fat
mat
bat
cat
More push
top
sat
fat
mat
bat
cat
Pop ( )

Array Based Implementation of Stack

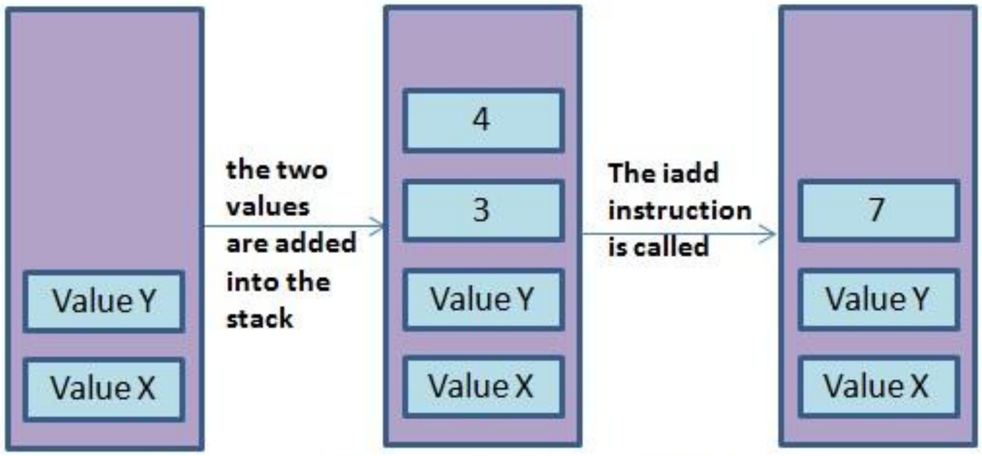
```
public class Stack {
    int  maxSize;
    int  top;
    String arr[];
}

    public  Stack {int n} {
        maxsize = n ;
        arr = new String [maxSize];
        top = 0;
    }

    public Boolean empty() {
        if (top == 0)
            return true;
        } else {
            return false;
        }
    }

    public void push (String str) {
        array[top] = str;
        top++;
    }

    public String pop() {
        if (top > 0) {
            return arr[top-1];
            arr[top-1] = null;
            top --;
        }
        else {
            return null
    }
    public String peek () {
    }
}
```

```
public static void main (String args []) {
stack myStack = new Stack(7);
    myStack.push("cat");
    System.out.println(myStack.peek( ));
    myStack.push("dog");
    System.out.println(myStack.empty( ));
    myStack.push("horse");
    etc etc
```

Uses of Stack

Other than implementing undo and browser back buttons, stacks have many applications.

- You can reverse a string using a stack. How?
- Checking if the parentheses are well formed
  [( ) ( )] is well-formed, but [(( ])) is not.
- Expression evaluation by JVM. How will it compute 3+4 = 7 or (3+4)* (6-9) + 18? (More to be discussed in the class)



State of the operand stack during the addition of 3 and 4

- Activation records at runtime

```
Class Xyz {
        firstMethod {
        int b;
        }
        secondMethod {
        int c;
        }
        thirdMethod {
        }
}
```

Heap and Stack space allocation to be discussed in the class

Advantages of Array-based Implementation

Fast – all operations are completed in O(1) time

Limitations of Array-based Implementation

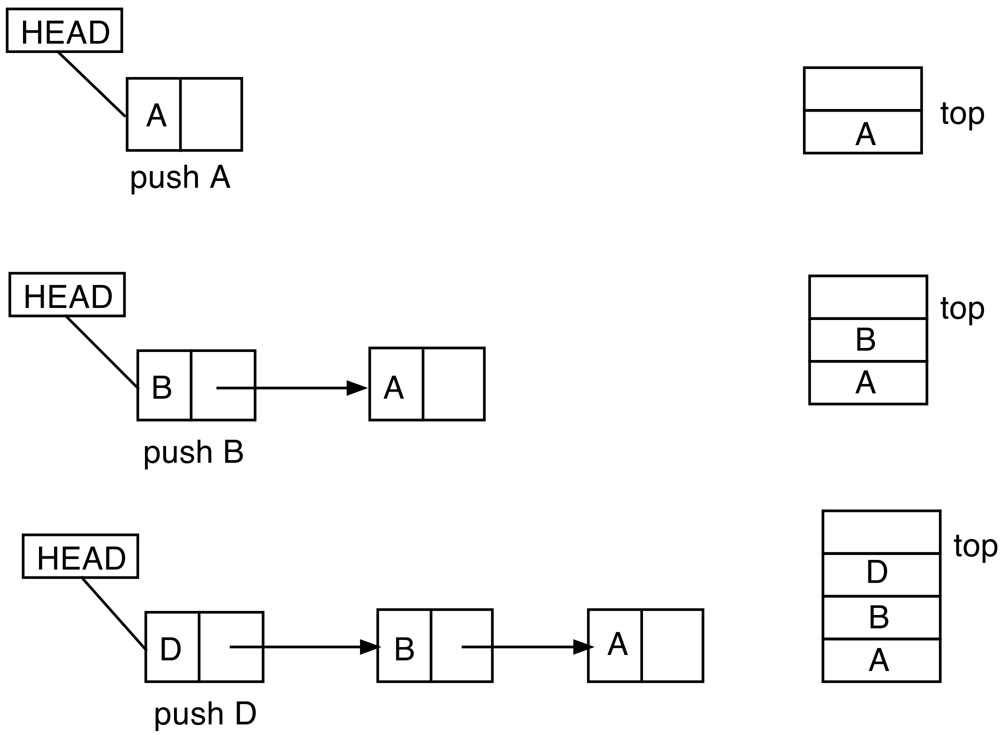
You have to know the upper bound of growth and allocate memory accordingly. If the array if full and there is another push operation then you encounter an exception.

Linked List based Stack Implementation

Can we implement a stack using a Linked List? Yes!

Do not have to worry about the size when the stack grows. Sky (i.e. the entire memory pool) is the limit.

Top of the stack = head of the linked list Bottom of the stack = tail of the linked list Push = add a new head Pop = remove the head



Now, push and pop will take O(1) time.

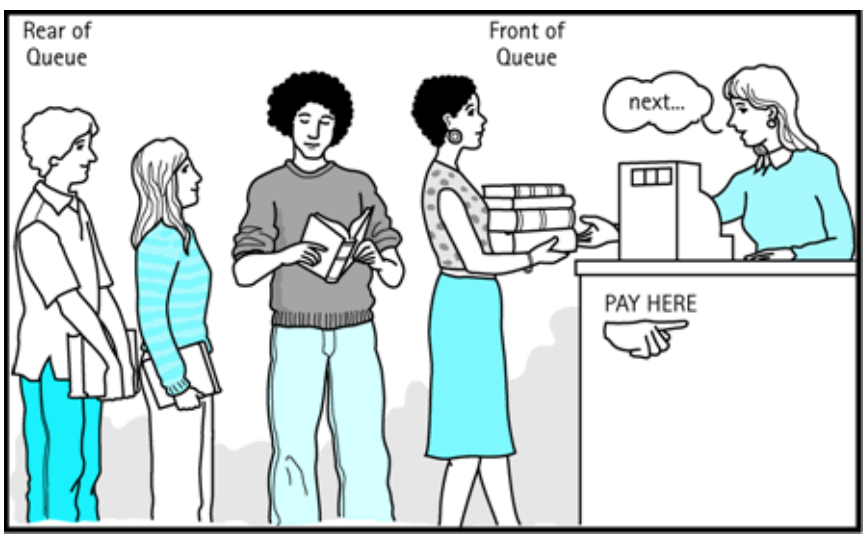
However, size ( ) will take O(n) time

The Queue ADT

Recall the **waiting list** for courses during registration for courses? When a seat opens up, the first one who joined the waiting list is the first to get a chance to add to the course.

This is a **queue**, works on the **first in first out** principle. Two access point: front and rear

Other examples are: Call centers, printer queue, etc



(Taken from http://jcsites.juniata.edu/faculty/kruse/cs240/queues)

The queue ADT supports two main update methods:

Enqueue (e)	Adds element e to the rear
Dequeue ( )	Removes and returns the first element from the front.

Other useful methods are

First ( )

Size ( )

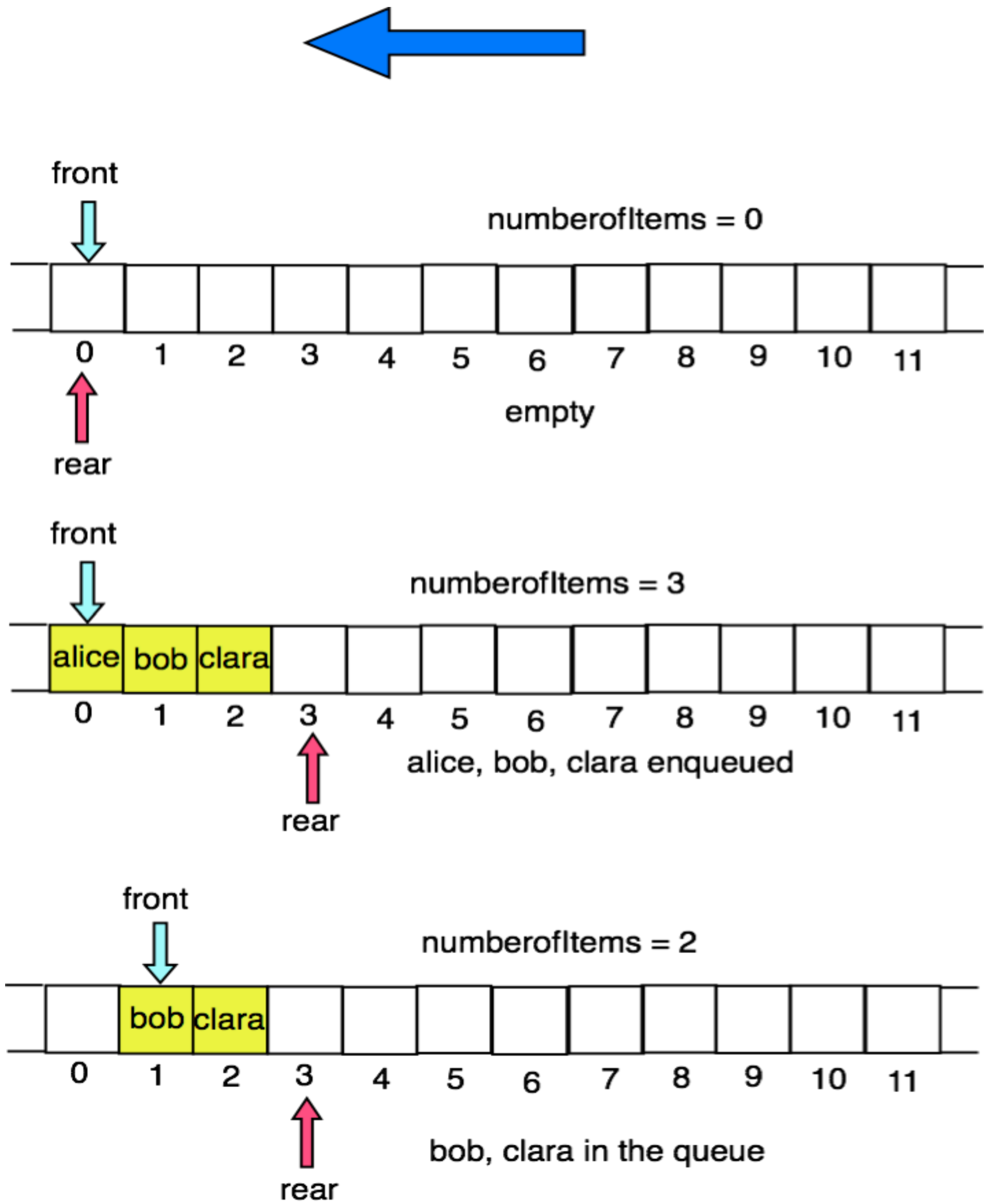
Empty ( )

The obvious implementation uses an array. After several **enqueue** and dequeue operations, both ends will drift.

Queue invariants

Acyclic structure

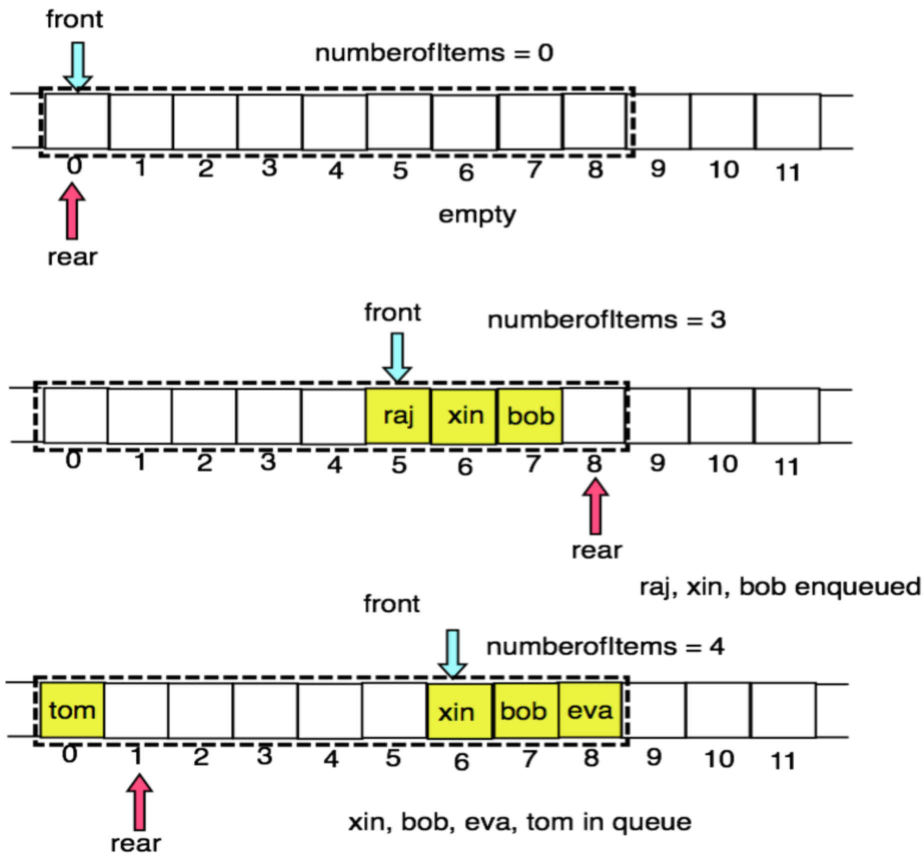
Fists In First Out Property

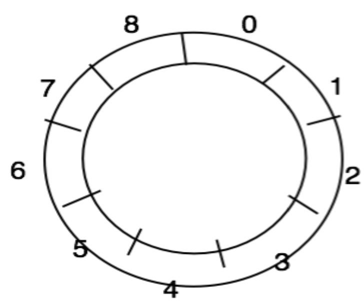


Array based implementation of queue

```
public class Queue {
    public String arr[ ];
    int maxSize;
    int front, rear, numberofItems = 0;

    public Queue (int n){
    maxSize = n
    arr = new String[maxSize]
    }

public void enqueue (String str) {
    if (numberofItems + 1 <= maxSize){
    array[rear] = input;
    rear++;
    numberofItems++} else {
    System.our.println(“Sorry, the Queue is Full”)
        }
    }


public void dequeue () {
    if (numberofItems > 0){
    System.our.println(arr[front] + “Was Removed”);
    front++;
    numberofItems--} else {
    System.our.println(“Sorry, the Queue is Empty”)
        }
    }

public static void main(String[ ] args){

    Queue myQueue = new Queue(8);

    myQueue.enqueue(“alice”)

    myQueue.enqueue(“bob”);

    myQueue.enqueue(“clara”)

    myQueue.dequeue();

    etc etc.

    }
```

Notice any problem with Space management? How will you limit the queue within the allotted space?





Observe the cyclic structure

Possible error conditions

1. Dequeue from an empty buffer

```
    if (numberofItems > 0){
        Normal dequeue action
    } else {
    System.out.println{"Sorry, the queue is empty'}
```

Also, `front` **must be incremented** `modulo maxSize`

2. Enqueue into a full buffer

```
if (numberofItems < maxSize){
        Normal enqueue action
    } else {
    System.out.println{"Sorry, the queue is full"}
```

Also, `rear` **must be incremented** `modulo maxSize`