Data Structures & Algorithms 1

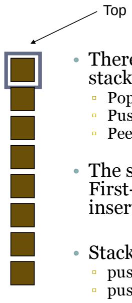
Topic 7 – Stacks and Queues

Abstract Data Types (ADTs)

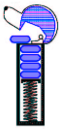
- Stacks and queues are abstract data types – they are more conceptual in nature than concrete data types such as arrays
- The ideas of stacks and queues is described by their interface – we're not interested in how they're actually implemented
- Underlying mechanism is typically not visible to the user – we just want to know how to use them

Stacks

Stack Interface

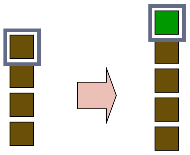


- There are several things you can do with a stack
  - Pop ( ) – pop the top item off the stack
  - Push ( ) – put another item onto the top
  - Peek ( ) – look at the top item and copy it
- The stack mechanism is known as Last-In-First-Out (LIFO) because the last item inserted is the first one removed
- Stacks are also referred to as
  - pushdown stack
  - pushdown list



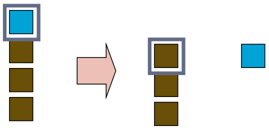
Push

- push (element)
  insert an element at the top of the stack



Pop

- pop ( )
  remove an element from the top of the stack

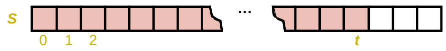


Other Stack Operations

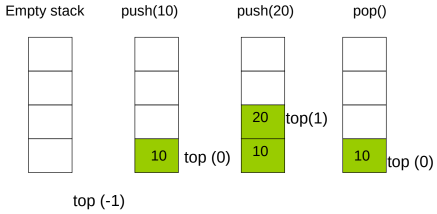
- MakeEmpty( )
  - Remove all items from the stack
- IsEmpty( )
  - True if stack is empty, false otherwise
- IsFull( )
  - True if stack is full, false otherwise

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

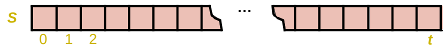


Push and Pop



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT



Implementing a Stack

- Stack object will need several instance fields
  - *maxSize* to store the size of the array
  - an array of numbers *stackArray* to store the stack
  - a variable called *top* to track where the top of the stack is
- Constructor will take in size of array, initialize it and set *top* to -1 since there's nothing in the stack to start with

Java Implementation

```
public class Stack{

   private int maxSize;         // size of stack array
   private long[] stackArray;
   private int top;             // top of stack

public Stack(int s) {          // constructor

      maxSize = s;              // set array size
      stackArray = new long[maxSize]; // create array
      top = -1;                 // no items yet
}
```

Methods

```
public void push(long j) {     // put item on top of stack

        top++;
       stackArray[top] = j;   // increment top, insert item
}

public long pop() {            // take item from top of stack

      return stackArray[top--]; //access item, decrement top
}

public long peek() {           // peek at top of stack

      return stackArray[top];
}
```

Methods

```
public boolean isEmpty() {    // true if stack is empty

       return (top == -1);
}

public boolean isFull() {      // true if stack is full

       return (top == maxSize-1);
}

public void makeEmpty() {      // empty stack

        top=-1;

}
```

Result

```
Stack theStack = new Stack(10);  // make new stack
theStack.push(20);      // push items onto stack
theStack.push(40);
theStack.push(60);
theStack.push(80);
while  (!theStack.isEmpty()){
     System.out.println(theStack.pop());
}
```

- Output would be :   80   60   40   20

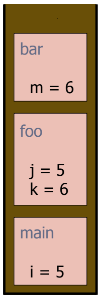
Stack Structures

- In which of the following situations could you feasibly use a stack?
  - Storing fruit
  - Storing milk cartons
  - Storing cans
- Can you think of anything else using a stack structure?
  - Web browsers
  - Undo sequence in a text editor
  - Java Virtual Machine

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```

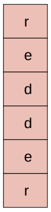


What can we use stacks for?

- The LIFO principle can be used in reversing a word
- If we push “h”, “e”, “l”, “l”, “o” and then pop the contents of the stack
- Output is “o”, “l”, “l”, “e”, “h”

Checking for palindromes

- The LIFO principle can be used to check if a word is a palindrome
- Say we have a word like **redder**
- Push the the word onto the stack
- Pop it off the stack
- It has now been reversed – check if it is the same as the original



Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( ))(( )){([( )])})
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (
- Start from the beginning of the sequence
- Opening brackets are placed on a stack
- When the program comes across a closing bracket it pops from the ‘opening bracket stack’ and this should match
- When the program comes to the end, the ‘opening bracket stack’ must be empty

Performance and Limitations

- Performance
  - Let ***n*** be the number of elements in the stack
  - The space used is ***O***(***n***)
  - Each operation runs in time ***O***(1) (e.g. pop, push)
- Limitations for array-based stacks
  - The maximum size of the stack must be defined *a priori* and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Queues

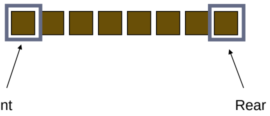
- A queue means to line up for something

Queues

- Queues use the First-In-First-Out system (FIFO)
- Rather than piling items up, the one that has been in the queue the longest is the one that is popped
- Insertions are made at one end, the back of the queue
- Deletions take place at the other end, the front of the queue
- So, the last one added is always the last one available for deletion

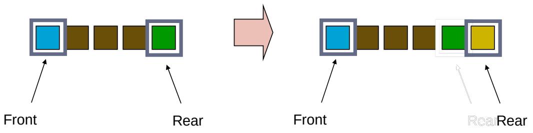
Queues

- Queues are used everywhere e.g. printer queue, multitasking
- Push ( ) is called insert, put, add or enqueue!
- Pop ( ) is called remove, delete, get or dequeue!
- The front and back of the queue are called the front and rear



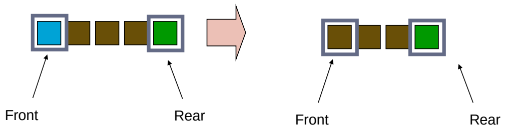
Insert

- insert ( ) method
  - Assumes the queue is not full
  - Inserts at rear
  - If rear is at the top of the array then it wraps around to the bottom of the array



Remove

- remove ( ) method
  - Assumes queue is not empty
  - Obtains the value at the front
  - Increments the front variable
  - If front goes beyond the end of the array it must be wrapped around to 0



Other Methods

- peek ( )
  - Returns the value at the front
- size ( )
  - Assumes queue not empty
  - Returns total number in queue
- isFull ( )
  - Returns true if queue is full
- isEmpty ( )
  - Returns true if queue is empty

Variables

- We need to maintain some variables with our queue class
  - Size of the array
  - The array itself
  - Variables for tracking front and rear
- Empty Queue:
  - Front = 0, Rear = -1
- 3 items in queue:
  - Front = 0, Rear = 2

Examples

Empty Queue

4	
3	
2	
1	
0	

Q.Front = 0
Q.Rear = -1

3 Elements

4	
3	
2	c
1	b
0	a

Q.Rear = 2

Q.Front = 0

4	
3	
2	c
1	
0	

Q.Front = 2
Q.Rear = 2

4	e
3	d
2	c
1	
0	

Q.Rear = 4

Q.Front = 2

Circular Array

4	e
3	d
2	c
1	
0	

Q.Rear = 4

Q.Front = 2

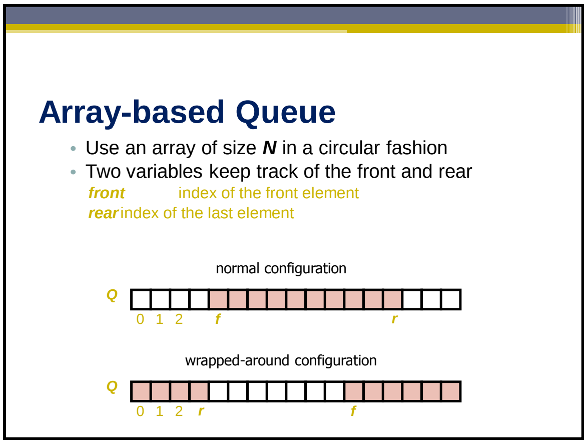
4	e
3	d
2	c
1	
0	

Q.Front = 2

Q.Rear = 0

((4+1)%5)

- We implement wrapround
  - when either back or front reach the end of the array, we reset it to the beginning.
  - To insert (*f*), we reset back to the start of the array and place *f* there.



Array-based Queue

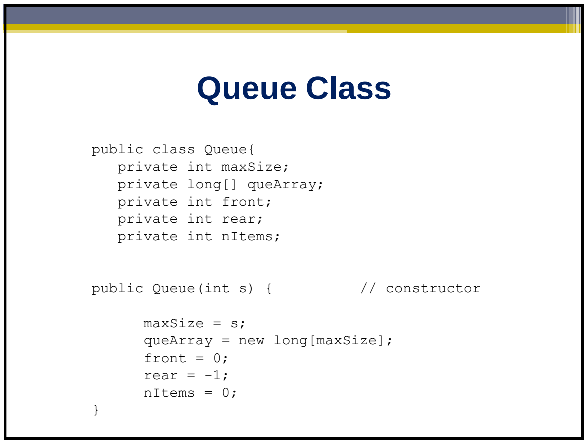
- Use an array of size ***N*** in a circular fashion
- Two variables keep track of the front and rear

front index of the front element

rear index of the last element

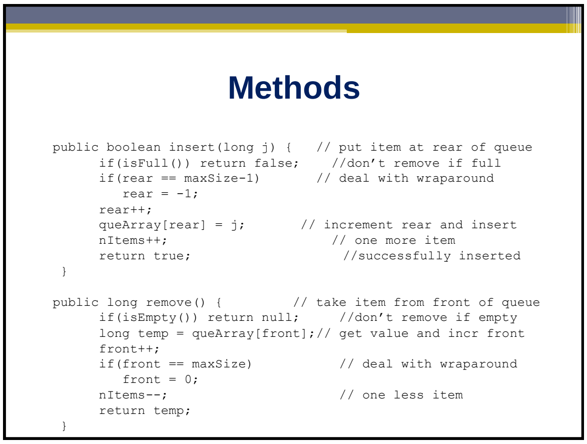
normal configuration

wrapped-around configuration



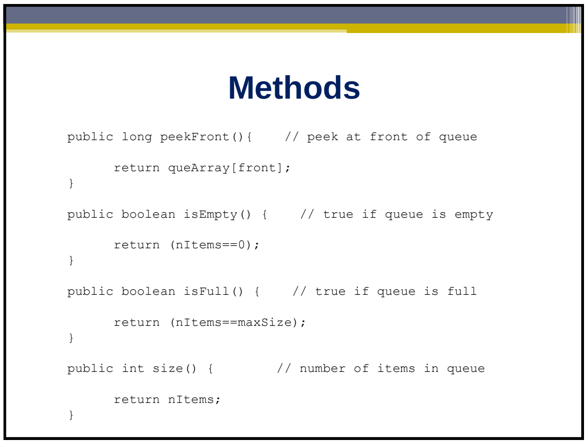
Queue Class

```
public class Queue{
   private int maxSize;
   private long[] queArray;
   private int front;
   private int rear;
   private int nItems;


public Queue(int s) {          // constructor

      maxSize = s;
      queArray = new long[maxSize];
      front = 0;
      rear = -1;
      nItems = 0;
}
```



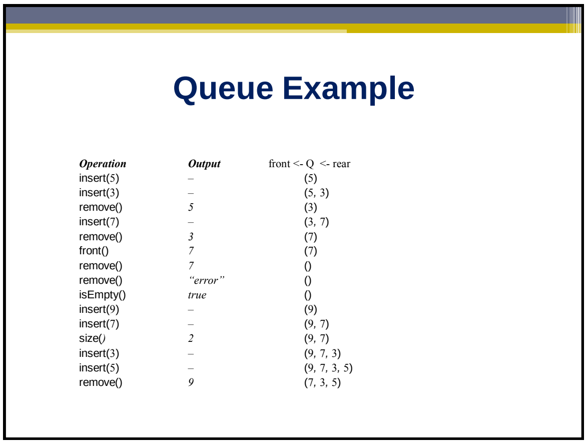
Methods

```
public boolean insert(long j) {   // put item at rear of queue
      if(isFull()) return false;    //don’t remove if full
      if(rear == maxSize-1)       // deal with wraparound
         rear = -1;
      rear++;
      queArray[rear] = j;       // increment rear and insert
      nItems++;                     // one more item
      return true;                   //successfully inserted
 }

public long remove() {         // take item from front of queue
      if(isEmpty()) return null;     //don’t remove if empty
      long temp = queArray[front];// get value and incr front
      front++;
      if(front == maxSize)           // deal with wraparound
         front = 0;
      nItems--;                      // one less item
      return temp;
 }
```



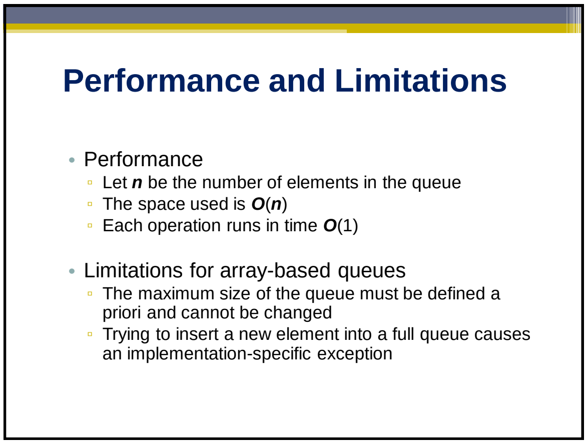
Methods

```
public long peekFront(){    // peek at front of queue

      return queArray[front];
}

public boolean isEmpty() {    // true if queue is empty

      return (nItems==0);
}

public boolean isFull() {    // true if queue is full

      return (nItems==maxSize);
}

public int size() {        // number of items in queue

      return nItems;
}
```



Queue Example

Operation	***Output***	front <- Q <- rear
insert(5)	–	(5)
insert(3)	–	(5, 3)
remove()	*5*	(3)
insert(7)	–	(3, 7)
remove()	*3*	(7)
front()	*7*	(7)
remove()	*7*	()
remove()	*“error”*	()
isEmpty()	*true*	()
insert(9)	–	(9)
insert(7)	–	(9, 7)
size()	*2*	(9, 7)
insert(3)	–	(9, 7, 3)
insert(5)	–	(9, 7, 3, 5)
remove()	*9*	(7, 3, 5)



Performance and Limitations

- Performance
  - Let ***n*** be the number of elements in the queue
  - The space used is ***O***(***n***)
  - Each operation runs in time ***O***(1)
- Limitations for array-based queues
  - The maximum size of the queue must be defined a priori and cannot be changed
  - Trying to insert a new element into a full queue causes an implementation-specific exception

Deque

- A deque is a double-ended queue
- This means you can insert items at either end and delete them at either end
- Essentially, there is no longer a front and rear, simply two ends
- Use methods called
  - insertLeft( )
  - insertRight( )
  - removeLeft( )
  - removeRight( )

Deques, Stacks and Queues

- A stack is actually a deque with only the methods
  - insertRight( )
  - removeRight( )
- A queue is a deque with only the methods
  - insertRight( )
  - removeLeft( )
- A deque is actually a more versatile data structure than either a stack or a queue but is not used as often

Priority Queue

- A priority queue is a queue where items don't just join at the rear, they are slotted into the queue according to their priority
- Imagine a stack of mail which are sorted according to priority
  - Each time a new letter is added, you slot it in according to its priority
  - Every time you pick up a letter to read, you are picking the most important one of the pile

What's Different?

- Don't really need to track front and rear as the rear always stays put at slot 0
- It looks kind of similar to a stack because we only need to track the top - would be better named as a "priority stack"
- We shift elements up to make space rather than just putting the element at the rear
- Insert method has a for loop that shifts elements up
- Remove method simply removes the top (highest priority) element

Priority Queue Insert ( )

```
 public void insert(long item) {              // insert item

     if(nItems==0){                           // if no items,
         queArray[0] = item;                  // insert at 0
      }else{                                   // if some items,
         int j = nItems;              // start at end

        while(j > 0 && queArray[j-1] > item){    // while new
  item larger
               queArray[j] = queArray[j-1];    // shift upward
               j--;                         // decrement j
         }
         queArray[j] = item;                  // insert it
     }
      nItems++;                               // increase items
}
```

Priority Queue

- Insertion is O(n) while deletion is O(1)
- What output do we get following from the following (assuming lower numbers have highest priority?)

```
PQ thePQ = new PQ(10);  // make new priority queue
thePQ.insert(60);      // slot items into queue
thePQ.insert(20);
thePQ.insert(80);
thePQ.insert(40);
while (!thePQ.isEmpty()){
     System.out.println(thePQ.remove());
}
```

20 40 60 80