CS2040 Data Structures and Algorithms
Lecture Note #6

Stacks and Queues

Two basic linear data structures

Objectives

1
- Able to define a Stack ADT, and to implement it with array and linked list

2
- Able to define a Queue ADT, and to implement it with array and linked list

3
- Able to use stack and queue in applications

4
- Able to use Java API Stack class and Queue interface

Programs used in this lecture

- Stacks
  - StackADT.java, StackArr.java, StackLL.java, StackLLE.java
  - TestStack.java
- Queues
  - QueueADT.java, QueueArr.java, QueueLL.java, QueueLLE.java
  - TestQueue.java

Outline

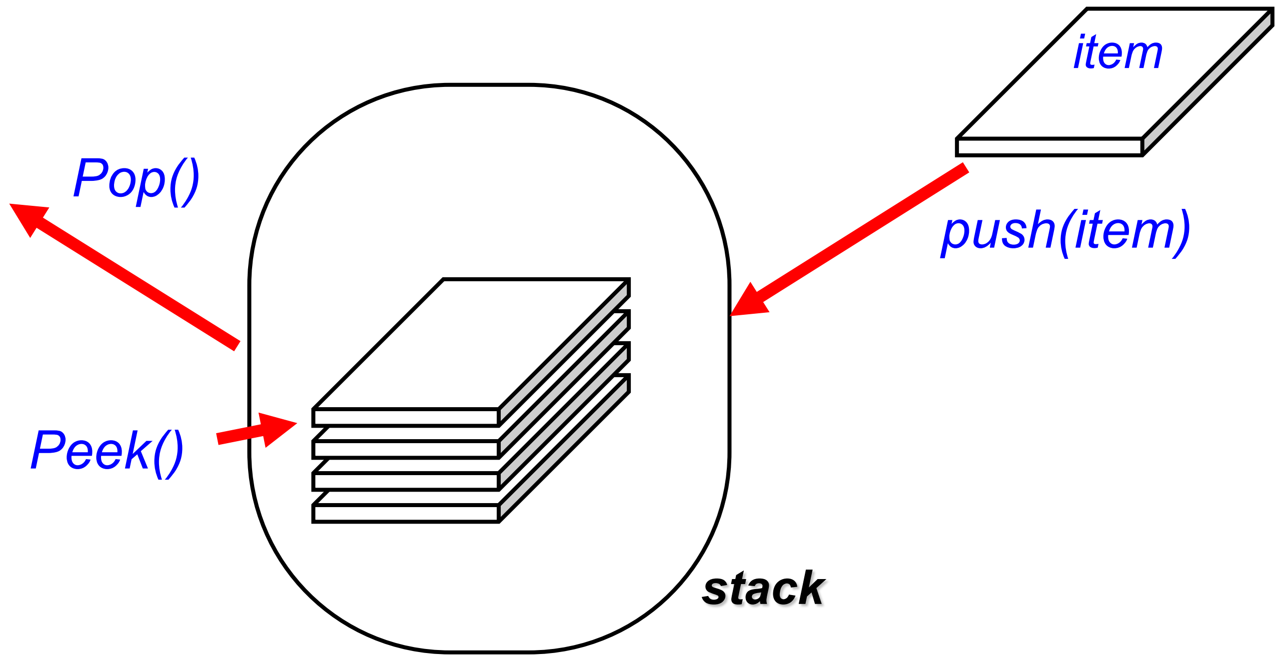
1. Stack ADT (Motivation)
2. Stack Implementation via Array
3. Stack Implementation via Linked List
4. java.util.Stack <E>
5. Stack Applications
   - Bracket matching
   - Postfix calculation
6. Queue ADT (Motivation)
7. Queue Implementation via Array
8. Queue Implementation via Tailed Linked List
9. java.util.interface Queue <E>
10. Application: Palindromes
11. Summary

1-5 Stacks

Last-In-First-Out (LIFO)

1 Stack ADT: Operations

- A Stack is a collection of data that is accessed in a last-in-first-out (LIFO) manner
- Major operations: "push", "pop", and "peek".



1 Stack ADT: Uses

- Calling a function
  - Before the call, the state of computation is saved on the stack so that we will know where to resume
- Recursion
- Matching parentheses
- Evaluating arithmetic expressions (e.g. a + b – c) :
  - postfix calculation
  - Infix to postfix conversion
- Traversing a maze → Will look at this later in the course

1 Stack ADT: Interface

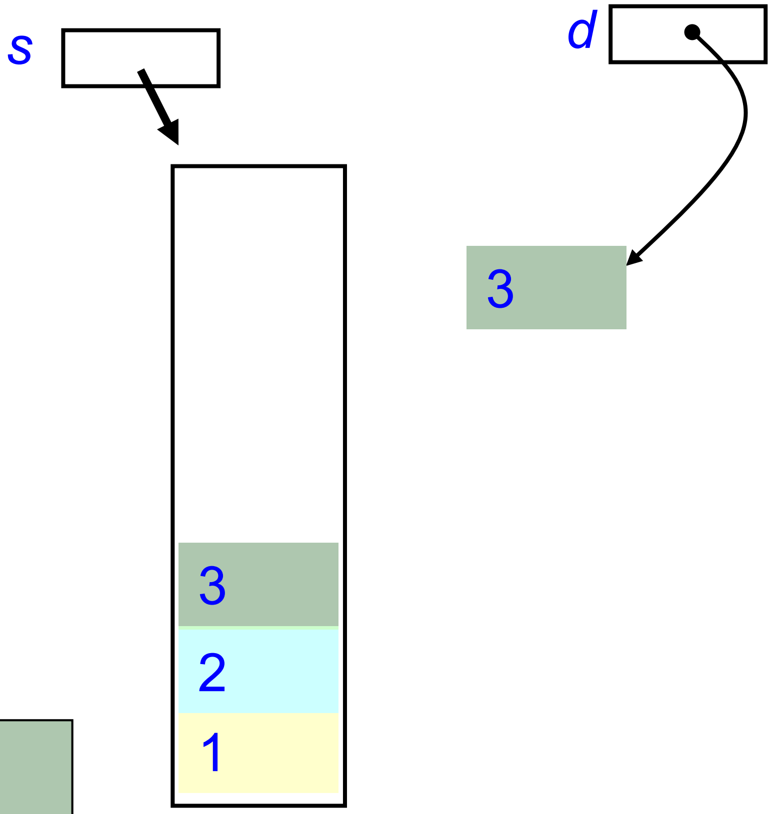
- For the purpose of the lecture we will only look at a stack that stores integer values

StackADT.java

```
import java.util.*;

public interface StackADT {
  // check whether stack is empty
  public boolean   empty();

  // retrieve topmost item on stack
  public Integer   peek(); // returns obj ver. of int

  // remove and return topmost item on stack
  public Integer   pop(); // returns obj ver. of int

  // insert item onto stack
  public void      push(Integer item);
}
```

1 Stack: Usage

```
Stack s = new Stack();
s.push (1);
s.push (2);
s.push (3);
d = s.peek ();
s.pop ();
s.push (4);
s.pop ();
```

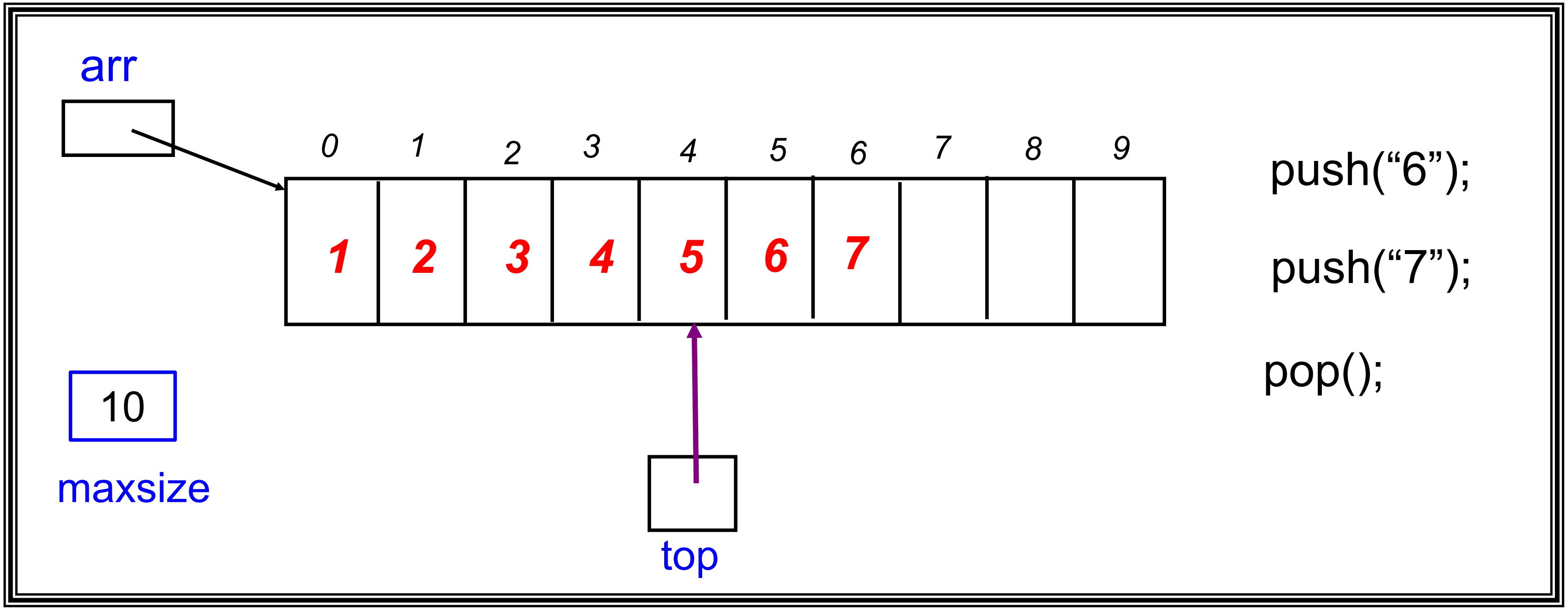


To be accurate, it is the references to 1, 2, 3, …, being pushed or popped.

2 Stack Implementation: Array

- Use an Array with a top index pointer

StackArr



2 Stack Implementation: Array

StackArr.java

```
import java.util.*;

class StackArr implements StackADT {
  public int[] arr;
  public int top;
  public int maxSize;
  public final int INITSIZE = 1000;

  public StackArr() {
    arr = new int[INITSIZE];
    top = -1; // in empty stack top is not on an valid array element
    maxSize = INITSIZE;
  }

  public boolean empty() { return (top < 0); }
```

2 Stack Implementation: Array

- pop() reuses peek()

StackArr.java

```
  public Integer peek() {
    if (!empty())
      return arr[top];
    return null; // use null to represent empty stack
  }

  public Integer pop() {
    Integer item = peek();
    if (item != null)
      top--;
    return item;
  }
}
```

2 Stack Implementation: Array

- push() needs to consider overflow

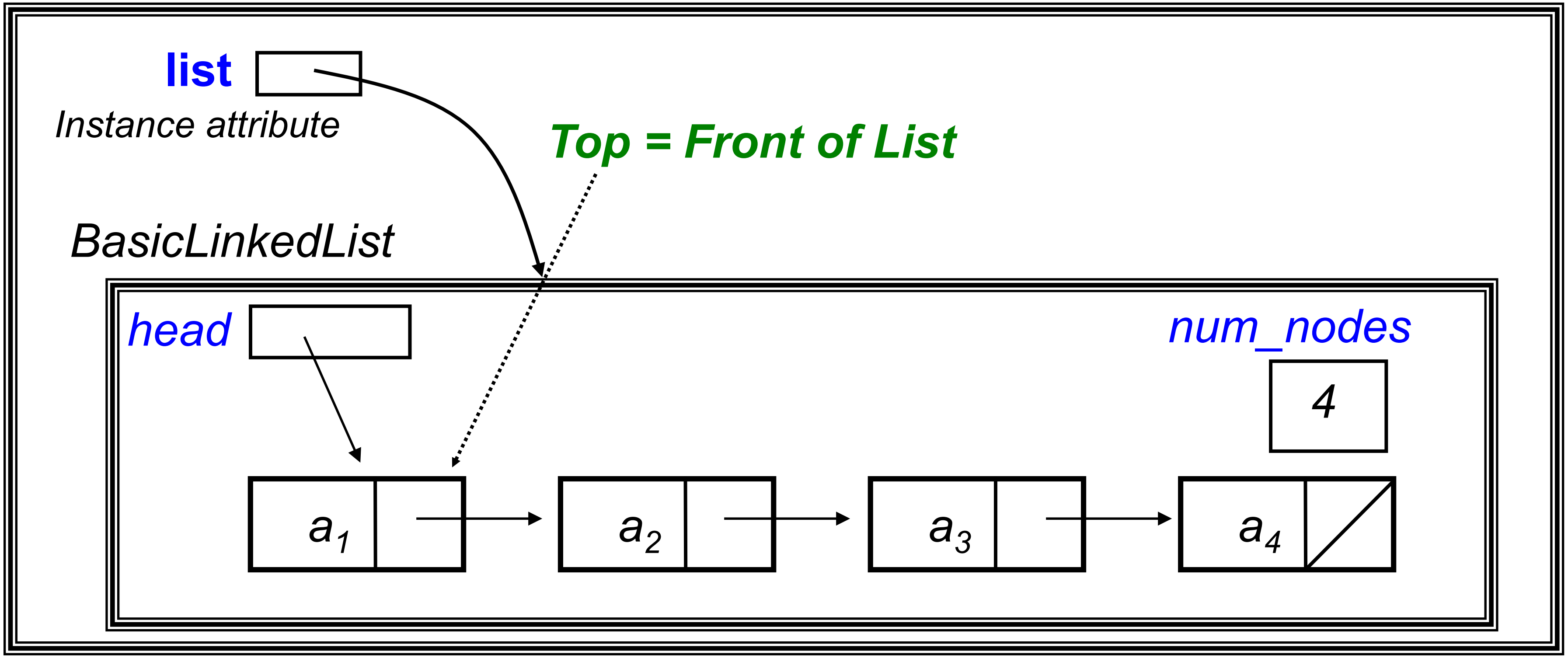
StackArr.java

```
public void push(Integer item) {
    if (top >= maxSize - 1) enlargeArr(); //array is full, enlarge it
    top++;
    arr[top] = item;
}

public void enlargeArr() {
  int newSize = capacity * 2; // double size
  int[] temp = new int[newSize];

  if (temp == null) {
    System.out.println("run out of memory!");
    System.exit(1);
  }
  for (int j=0; j <= top; j++)
    temp[j] = arr[j];
  arr = temp; // point arr to the new array
  capacity = newSize;
}
}
```

3 Stack Implementation: Linked List (1/6)

- Have a BasicLinkedList as an instance attribute

***StackLL* – class for our linked list implementation**



3 Stack Implementation: Linked List (2/6)

- (Composition): Use BasicLinkedList

StackLL.java

```
class StackLL implements StackADT {
  public BasicLinkedList list;

  public StackLL() { list = new BasicLinkedList(); }
  public boolean empty() { return list.isEmpty(); }

  public Integer peek() {
    if (!list.empty())
      return list.getFirst();
    return null; // use null to represent empty stack
  }

  public Integer pop() {
    Integer item = peek();
    if (!list.empty())
      list.removeFront();
    return item;
  }

  public void push(Integer item) { list.addFront(item); }
}
```

3 Uses of Stack

TestStack.java

```
import java.util.*;
public class TestStack {
  public static void main (String[] args) {

    // You can use any of the following 3 implementations of Stack
    StackArr stack = new StackArr();  // Array
    //StackLL stack = new StackLL(); // LinkedList composition
    //Stack <Integer> stack = new Stack <Integer>();  // Java API

    System.out.println("stack is empty? " + stack.empty());
    stack.push(1);
    stack.push(2);
    System.out.println("top of stack is " + stack.peek());
    stack.push(3);
    System.out.println("top of stack is " + stack.pop());
    stack.push(4);
    stack.pop();
    stack.pop();
    System.out.println("top of stack is " + stack.peek());
  }
}
```

```
stack is empty? true
top of stack is 2
top of stack is 3
top of stack is 1
```

4 java.util.Stack <E> (1/2)

Constructor Summary
Stack() Creates an empty Stack.

Method Summary	
boolean	empty() Tests if this stack is empty.
E	peek() Looks at the object at the top of this stack without removing it from the stack.
E	pop() Removes the object at the top of this stack and returns that object as the value of this function.
E	push(E item) Pushes an item onto the top of this stack.
int	search(Object o) Returns the 1-based position where an object is on this stack.

Note: The method "int search (Object o)" is not commonly known to be available from a Stack.

4 java.util.Stack <E> (2/2)

Methods inherited from class java.util.Vector
add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode, indexOf, indexOf, insertElementAt, isEmpty, lastElement, lastIndexOf, lastIndexOf, remove, remove, removeAll, removeAllElements, removeElement, removeElementAt, removeRange, retainAll, set, setElementAt, setSize, size, subList, toArray, toArray, toString, trimToSize

Methods inherited from class java.util.AbstractList
iterator, listIterator, listIterator

Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.List
iterator, listIterator, listIterator

5 Application 1: Bracket Matching

- Ensures that pairs of brackets are properly matched

An example: `{a,(b+f[4])*3,d+f[5]}`

Incorrect examples:

`(..)..)`	// too many close brackets
`(..(..)`	// too many open brackets
`[..(..]..)`	// mismatched brackets

5 Application 1: Bracket Matching

```
create empty stack
for every char read
{
  if open bracket then
      push onto stack
  if close bracket, then
     if doesn't match or underflow then flag error
     else pop from the stack
 }
if stack is not empty then flag error
```

Q: What type of error does the last line test for?

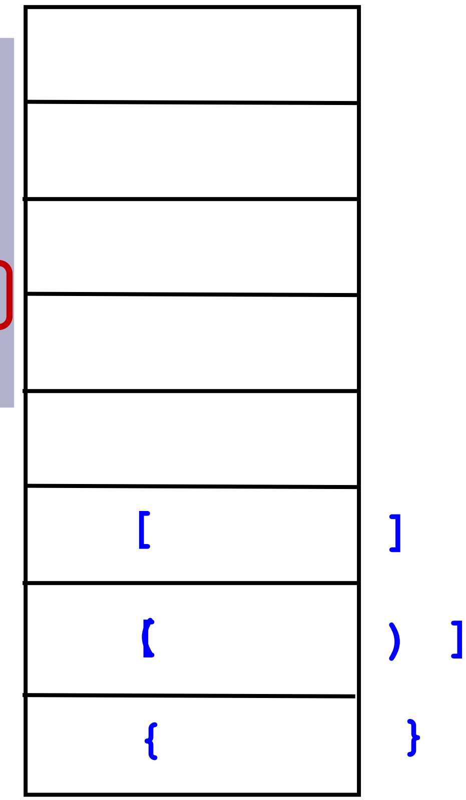
A: too many closing brackets

B: too many opening brackets

C: bracket mismatch

Example

`{ a -( b + f [ 4 ] ) * 3 * d + f [ 5 ] }`



Stack

5 Applic^{n} 2: Evaluating Arithmetic Expression

- Terms
  - Expression: a = b + c * d
  - Operands: a, b, c, d
  - Operators: =, +, –, *, /, %
- Precedence rules: Operators have priorities over one another as indicated in a table (which can be found in most books & our first few lectures)
  - Example: * and / have higher precedence over + and –.
  - For operators at the same precedence (such as * and /), we process them from left to right

5 Applic^{n} 2: Evaluating Arithmetic Expression

Infix	: operand1	operator	operand2
Prefix	: operator	operand1	operand2
Postfix	: operand1	operand2	operator

Ambiguous, need () or precedence rules

Unique interpretation

postfix

infix

`2+3*4`

`(2+3)*4` → `2 3 + 4 *`

`2+(3*4)` → `2 3 4 * +`

5 Applic^{n} 2: Evaluating Arithmetic Expression

Algorithm: Evaluating Postfix expression with stack

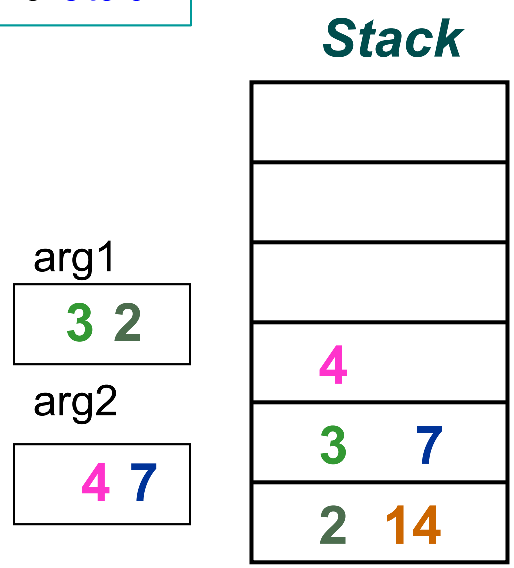
Create an empty stack
for each item of the expression,
  if it is an operand,
    push it on the stack
  if it is an operator,
    pop arguments from stack;
    perform the operation;
    push the result onto the stack

Infix		postfix
2 * (3 + 4)	⟶	**2 3 4 + ***

2 s.push(2)
3 s.push(3)
4 s.push(4)
+ arg2 = s.pop ()
  arg1 = s.pop ()
  s.push (arg1 + arg2)

* arg2 = s.pop ()

  arg1 = s.pop ()
  s.push (arg1 * arg2)

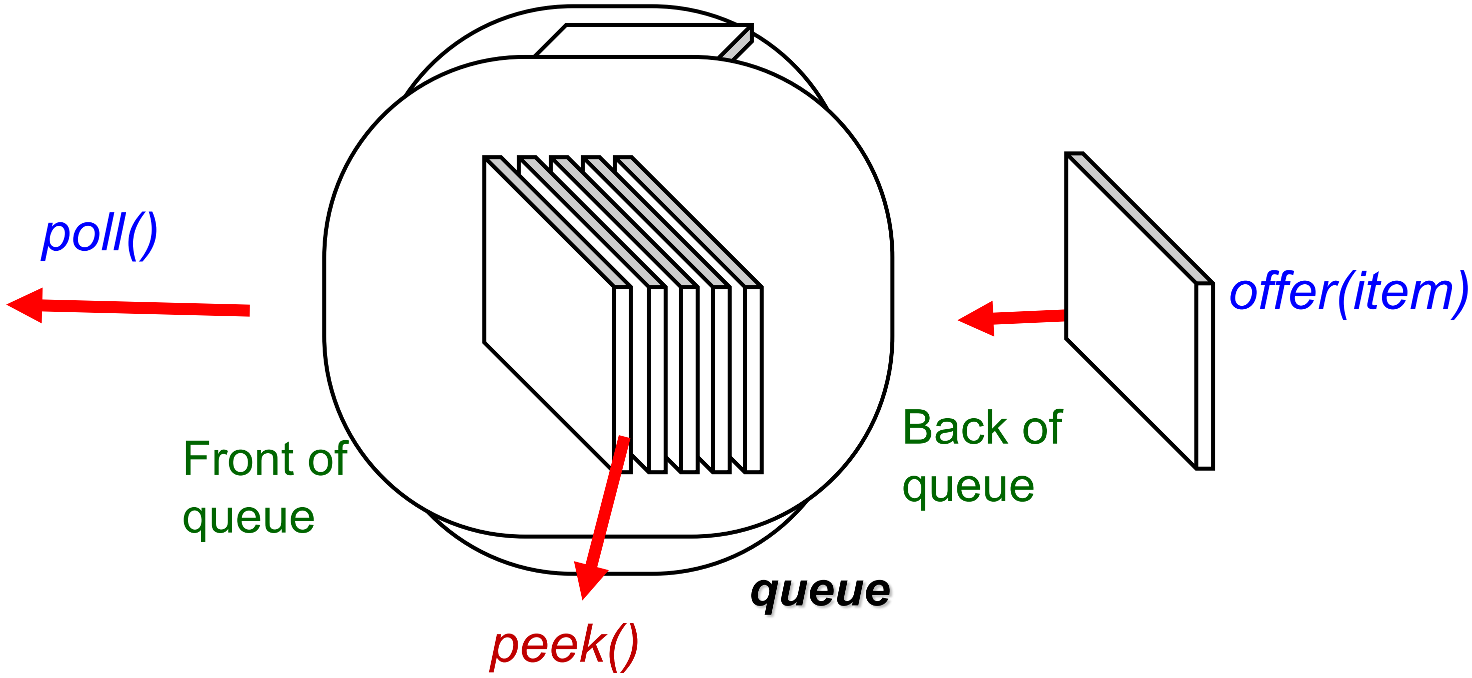


6-9 Queues

First-In-First-Out (FIFO)

6 Queue ADT: Operations

- A Queue is a collection of data that is accessed in a first-in-first-out (FIFO) manner
- Major operations: "poll" (or "dequeue"), "offer" (or "enqueue"), and "peek".



6 Queue ADT: Uses

- Print queue
- Simulations
- Breadth-first traversal of graph → look at this later in the course
- Checking palindromes - for illustration only as it is not a real application of queue

6 Queue ADT: Interface

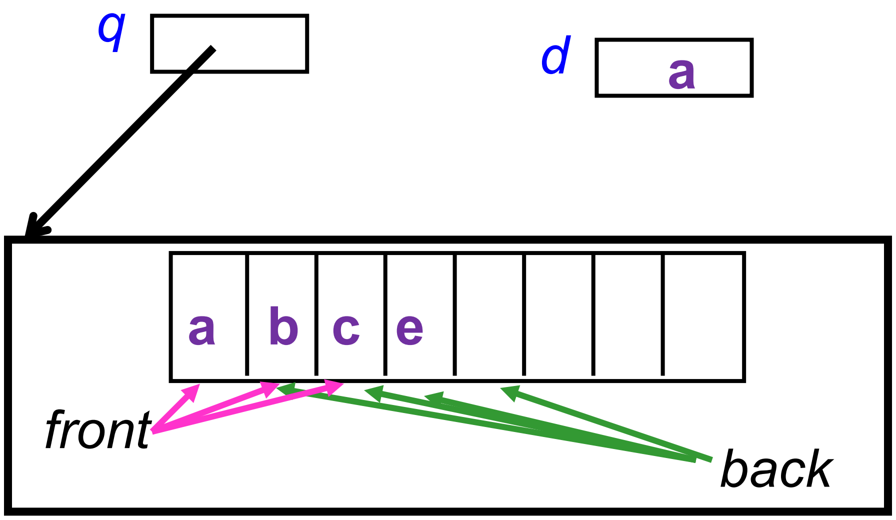
- For the purpose of the lecture we will only look at a queue that stores integer values

QueueADT.java

```
import java.util.*;

public interface QueueADT {

  // return true if queue has no elements
  public boolean empty();

  // return the front of the queue
  public Integer peek();

  // remove and return the front of the queue
  public Integer poll(); // also known as dequeue

  // add item to the back of the queue
  public void    offer(Integer item); // also known as enqueue
}
```

6 Queue: Usage

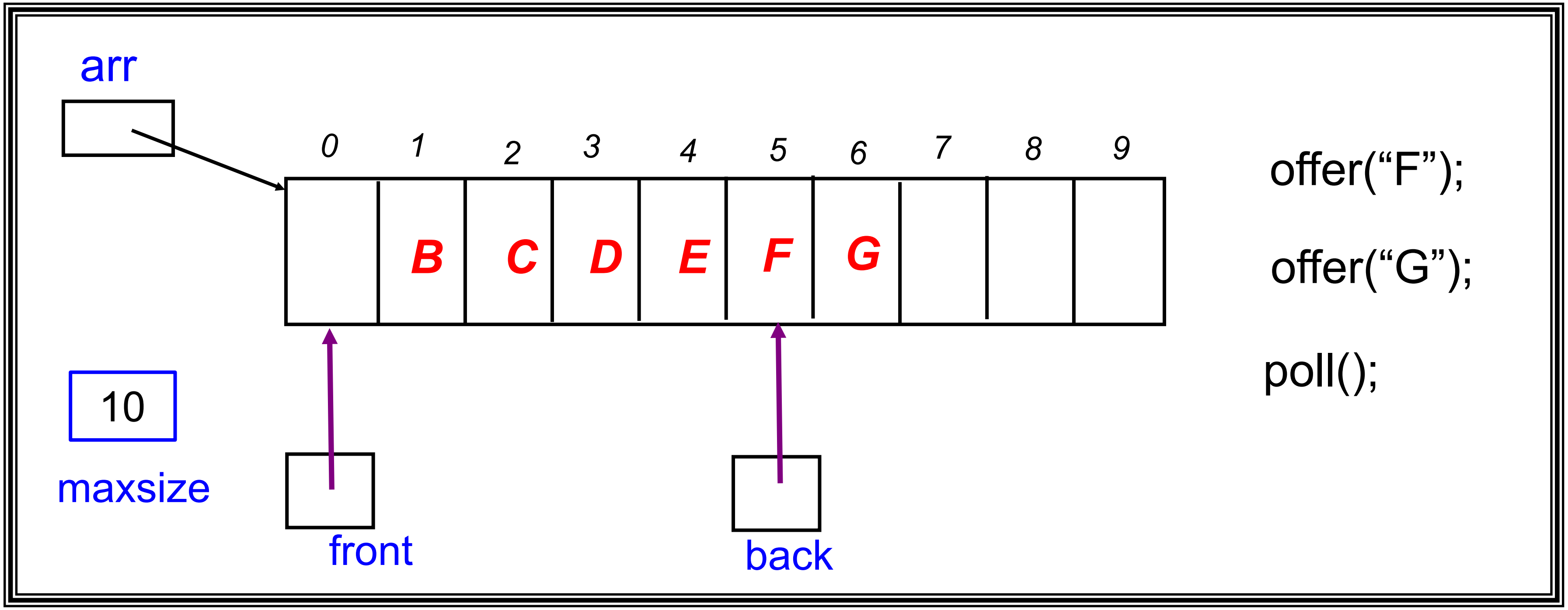
```
Queue q = new Queue ();
q.offer ("a");
q.offer ("b");
q.offer ("c");
d = q.peek ();
q.poll ();
q.offer ("e");
q.poll ();
```



7 Queue Implementation: Array

- Use an Array with front and back indices

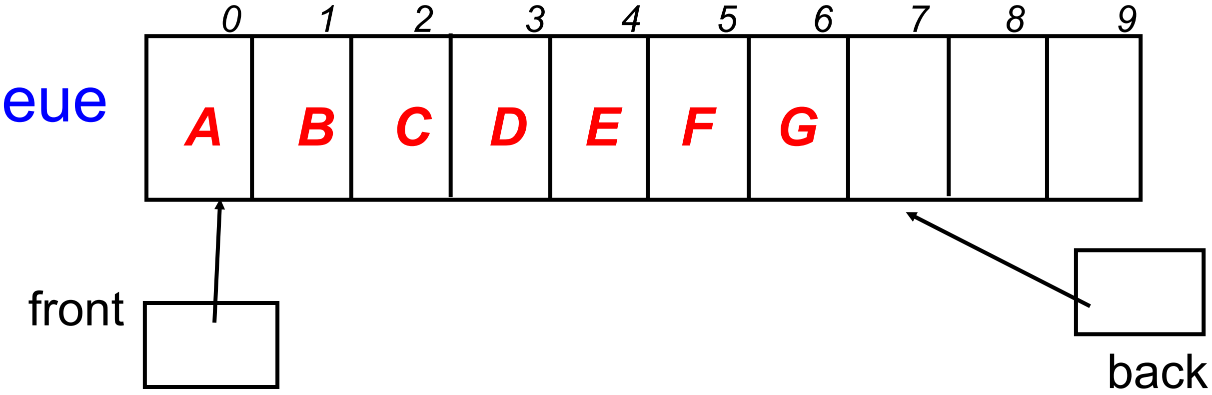
QueueArr – class that implements Queue using array

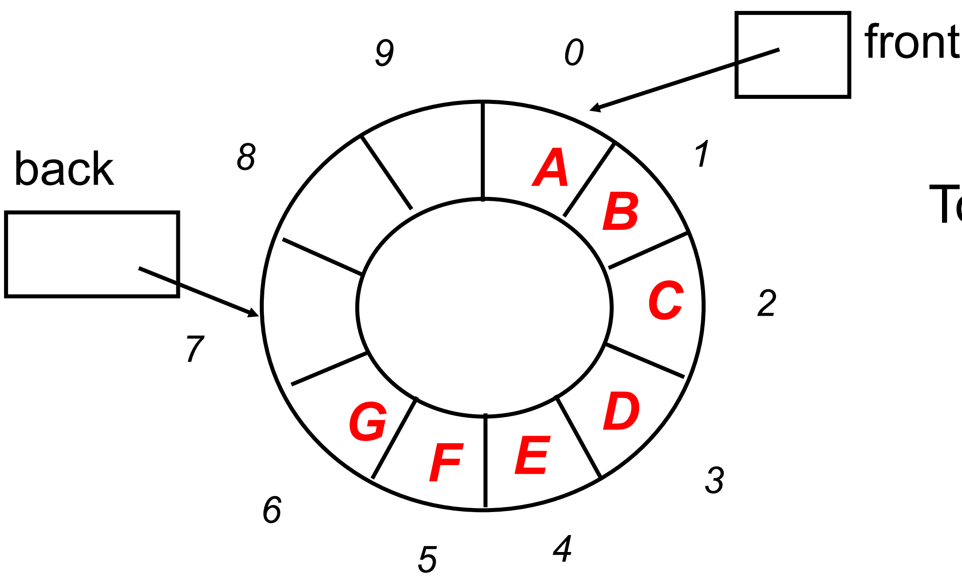


7 Queue Implementation: Array (2/7)

- "Circular" Array needed to recycle space

Given a queue





To advance the indexes, use

front = (front+1) % maxsize;

back = (back+1) % maxsize;

7 Queue Implementation: Array (3/7)

- Question: what does (front == back) mean?

  A: Full queue

  B: Empty queue

  C: Both A and B

  D: Neither A nor B

C: Both A and B

7 Queue Implementation: Array (4/7)

- Ambiguous full/empty state

Queue
Empty
State

F
B

e	f	c	d

F
B

Queue
Full
State

Solution 1 – Maintain queue size or full status

size **0** *size* **4**

Solution 2 **(Used in our codes)** – Leave a gap!

Don't need the size field this way

e		c	d

B F

Full Case: (((B+1) % maxsize) == F)
Empty Case: F == B

7 Queue Implementation: Array (5/7)

QueueArr.java

```
import java.util.*;

// This implementation uses solution 2 to resolve full/empty state
class QueueArr implements QueueADT {
  public int [] arr;
  public int front, back;
  public int maxSize;
  public final int INITSIZE = 1000;

  public QueueArr() {
    arr = new int[INITSIZE];
    front = 0; // the queue is empty
    back = 0;
    maxSize = INITSIZE;
  }

  public boolean empty() {
    return (front == back);        // use solution 2
  }
```

7 Queue Implementation: Array

QueueArr.java

```
public Integer peek() { // return front of the queue
  if (empty()) return null;
  else return arr[front];
}

public Integer poll() { // remove and return front of the queue
  if (empty()) return null;
  Integer item = arr[front];
  front = (front + 1) % maxSize;  // "circular" array
  return item;
}

public void offer(Integer item) {// add item to back of queue
  if (((back+1)%maxSize) == front) // array is full
    enlargeArr(); // no more memory so enlarge the array
  arr[back] = item;
  back = (back + 1) % maxSize;   // "circular" array
}
```

7 Queue Implementation: Array (7/7)

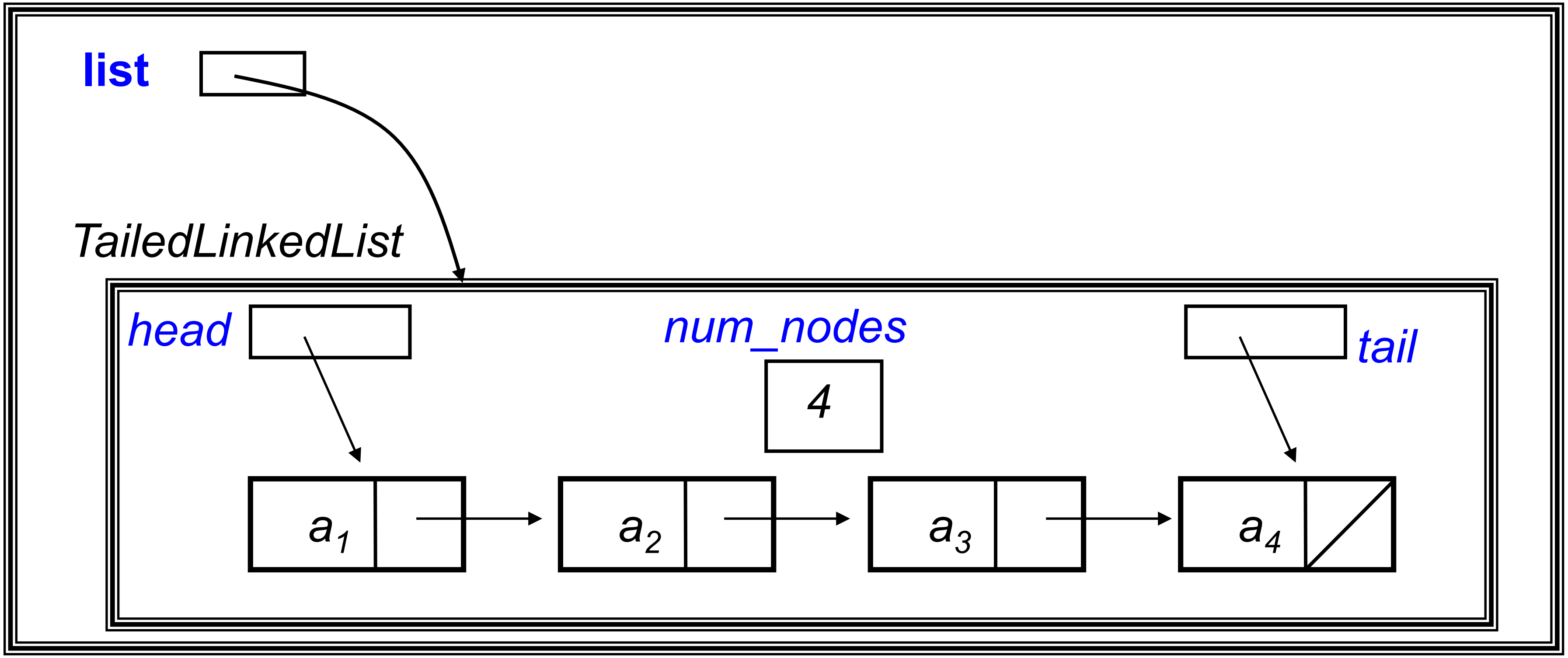
QueueArr.java

```
  public void enlargeArr() {
    int newSize = maxSize * 2;
    int[] temp = new int[newSize];

    if (temp == null) { // not enough memory to create new array
      System.out.println("run out of memory!");
      System.exit(1);
    }
    for (int j=0; j < maxSize; j++) {
      // copy the front (1st) element, 2nd element, ..., in the
      // original array to the 1st (index 0), 2nd (index 1), ...,
      // positions in the enlarged array. Q: Why this way?
      temp[j] = arr[(front+j) % maxSize];
    }
    front = 0;
    back = maxSize - 1;
    arr = temp;
    maxSize = newSize;
  }
}
```

8 Queue Implement^n: Linked List

- Use TailedLinkedList
  - Do not use BasicLinkedList as we would like to use addBack() of TailedLinkedList.

***QueueLL* – class of LinkedList based implementation of Queue**



8 Queue Implement^n: Linked List

- Use TailedLinkedList

QueueLL.java

```
import java.util.*;
class QueueLL implements QueueADT {
  public TailedLinkedList list;

  public QueueLL() { list = new TailedLinkedList(); }

  public boolean empty() { return list.isEmpty(); }

  public void offer(Integer item) { list.addBack(item); }

  public Integer peek() {
    if (empty()) return null;
    return list.getFirst();
  }

  public Integer poll() {
    Integer item = peek();
    if (!empty())
      list.removeFront();
    return item;
  }
}
```

8 Uses of Queues

TestQueue.java

```
import java.util.*;
public class TestQueue {
   public static void main (String[] args) {
   // you can use any one of the following implementations
   QueueLL queue= new QueueLL(); // LinkedList composition
  //QueueArr queue= new QueueArr(); // Array
  //LinkedList <Integer> queue = new LinkedList<Integer>();


   System.out.println("queue is empty? " + queue.empty());
   queue.offer(1);
   System.out.println("operation: queue.offer(1)");
   System.out.println("queue is empty? " + queue.empty());
   System.out.println("front now is: " + queue.peek());
   queue.offer(2);
   System.out.println("operation: queue.offer(2)");
   System.out.println("front now is: " + queue.peek());
   queue.offer(3);
   System.out.println("operation: queue.offer(3)");
   System.out.println("front now is: " + queue.peek());
```

```
queue is empty? true
operation: queue.offer(1)
queue is empty? false
front now is: 1
operation: queue.offer(2)
front now is: 1
operation: queue.offer(3)
front now is: 1
```

8 Uses of Queues

TestQueue.java

```
    queue.poll();
    System.out.println("operation: queue.poll()");
    System.out.println("front now is: " + queue.peek());
    System.out.print("checking whether queue.peek().equals(1): ");
    System.out.println(queue.peek().equals(1));
    queue.poll();
    System.out.println("operation: queue.poll()");
    System.out.println("front now is: " + queue.peek());
    queue.poll();
    System.out.println("operation: queue.poll()");
    System.out.println("front now is: " + queue.peek());
    }
}
```

(output from previous slide…)

```
queue is empty? true
operation: queue.offer(1)
queue is empty? false
front now is: 1
operation: queue.offer(2)
front now is: 1
operation: queue.offer(3)
front now is: 1
```

(output continues…)

```
operation: queue.poll()
front now is: 2
checking whether queue.peek().equals(1): false
operation: queue.poll()
front now is: 3
operation: queue.poll()
front now is: null
```

9 java.util.interface Queue <E>

- Note that there is no Queue class in the Java API, only a Queue interface
- LinkedList class it one of the classes in the Java API that implements this interface, so use LinkedList class (only those methods defined in the Queue interface) in your assignments/tests if you are not restricted to writing your own Queue class

10 Palindromes

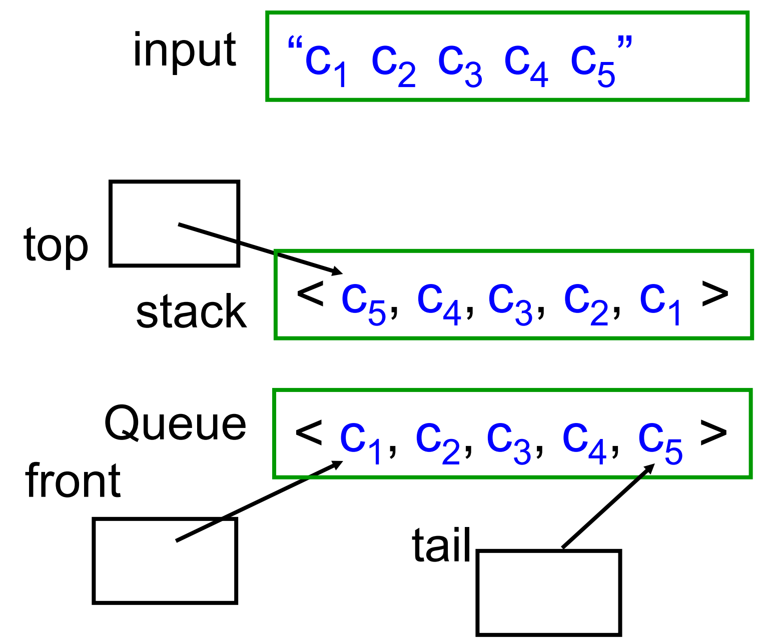
Application using both Stack and Queue

10 Application: Palindromes (1/3)

- A string which reads the same either left to right, or right to left is known as a palindrome
  - Palindromes: “radar”, “deed”, “aibohphobia”
  - Non-palindromes: “data”, “little”

Algorithm
Given a string, use:
    a Stack to *reverse* its order
    a Queue to *preserve* its order

Check if the sequences are the same



11 Summary

- We learn to create our own data structures from array and linked list
  - LIFO vs FIFO – a simple difference that leads to very different applications
  - Drawings can often help in understanding the different cases for operations on the Stack and Queue
- Please do not forget that the Java Library class is much more comprehensive than our own – for lab assignments or exam, please use the one as told.

End of file