Data Structures and Algorithms

Stacks and Queues

Two basic linear data structures

Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy, and Dr. Low Kok Lim for kindly sharing these materials.

Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Recording of modifications

- Course website address is changed to http://sakai.it.tdt.edu.vn
- Course codes cs1010, cs1020, cs2010 are placed by 501042, 501043, 502043 respectively.

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

References

Book

- **Stacks:** Chapter 7 (recursion excluded)
- **Queues:** Chapter 8

IT-TDT Sakai → 501043 website → Lessons

- http://sakai.it.tdt.edu.vn

Programs used in this lecture

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

Outline

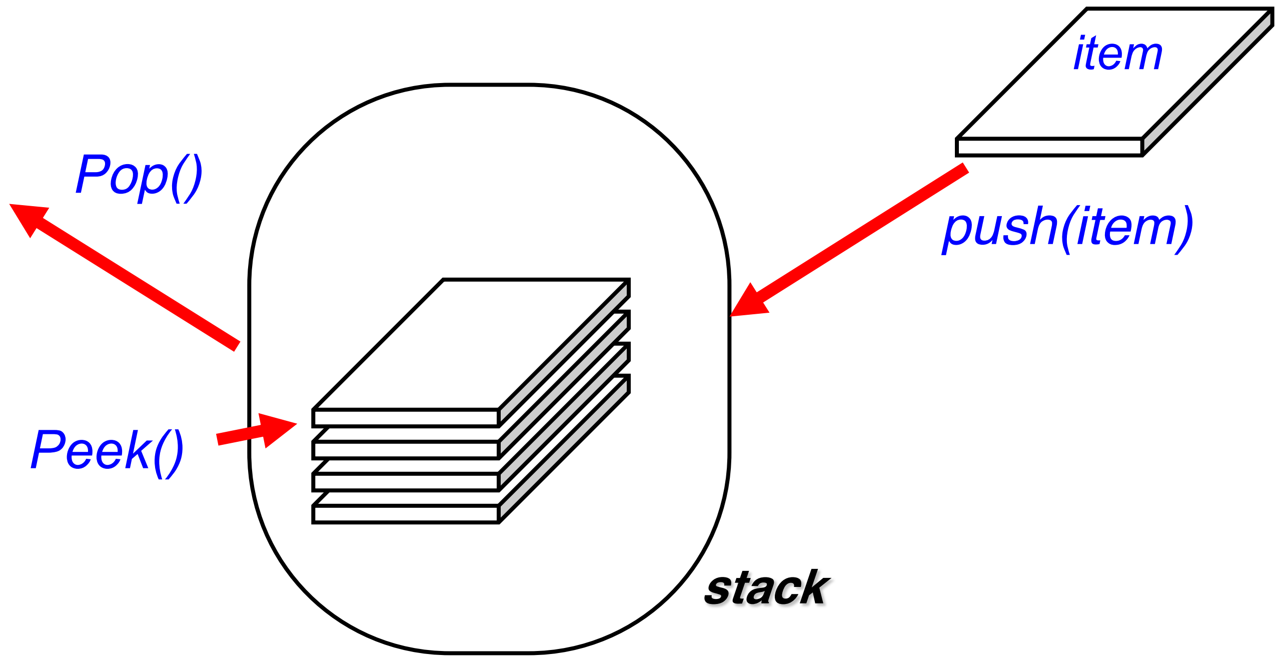
1. Stack ADT (Motivation)
2. Stack Implementation via Array
3. Stack Implementation via Linked List
4. java.util.Stack <E>
5. Stack Applications
   - Bracket matching
   - Postfix calculation
6. Queue ADT (Motivation)
7. Queue Implementation via Array
8. Queue Implementation via Tailed Linked List
9.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

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

1 Stack ADT: Interface

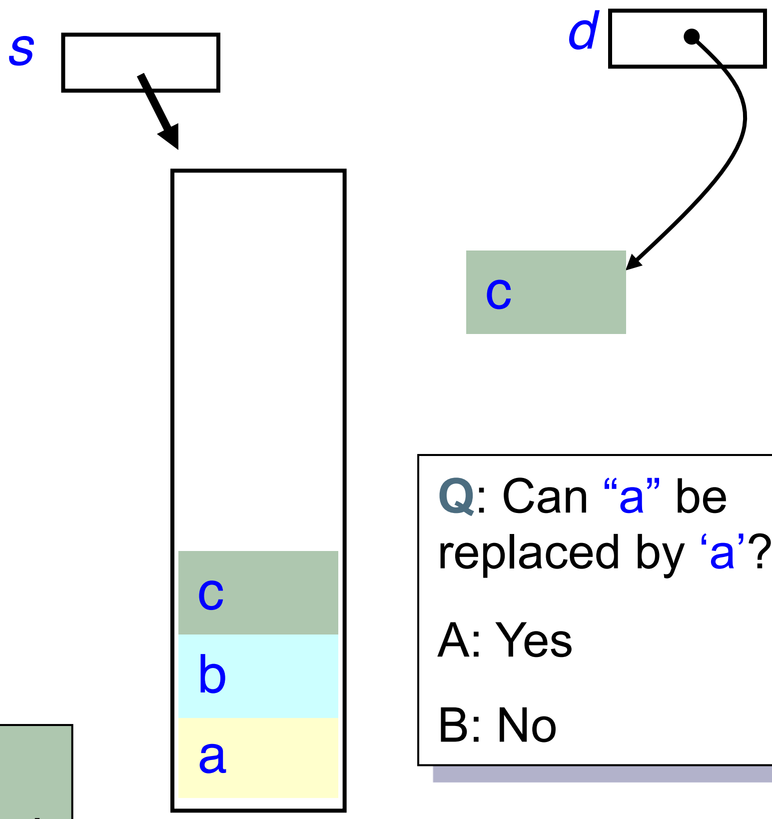
StackADT.java

```
import java.util.*;

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

  // retrieve topmost item on stack
  public E       peek() throws EmptyStackException;

  // remove and return topmost item on stack
  public E       pop() throws EmptyStackException;

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

1 Stack: Usage

- Stack s = new Stack();
- s.push ("a");
- s.push ("b");
- s.push ("c");
- d = s.peek ();
- s.pop ();
- s.push ("e");
- s.pop ();

To be accurate, it is the references to "a", "b", "c", …, being pushed or popped.



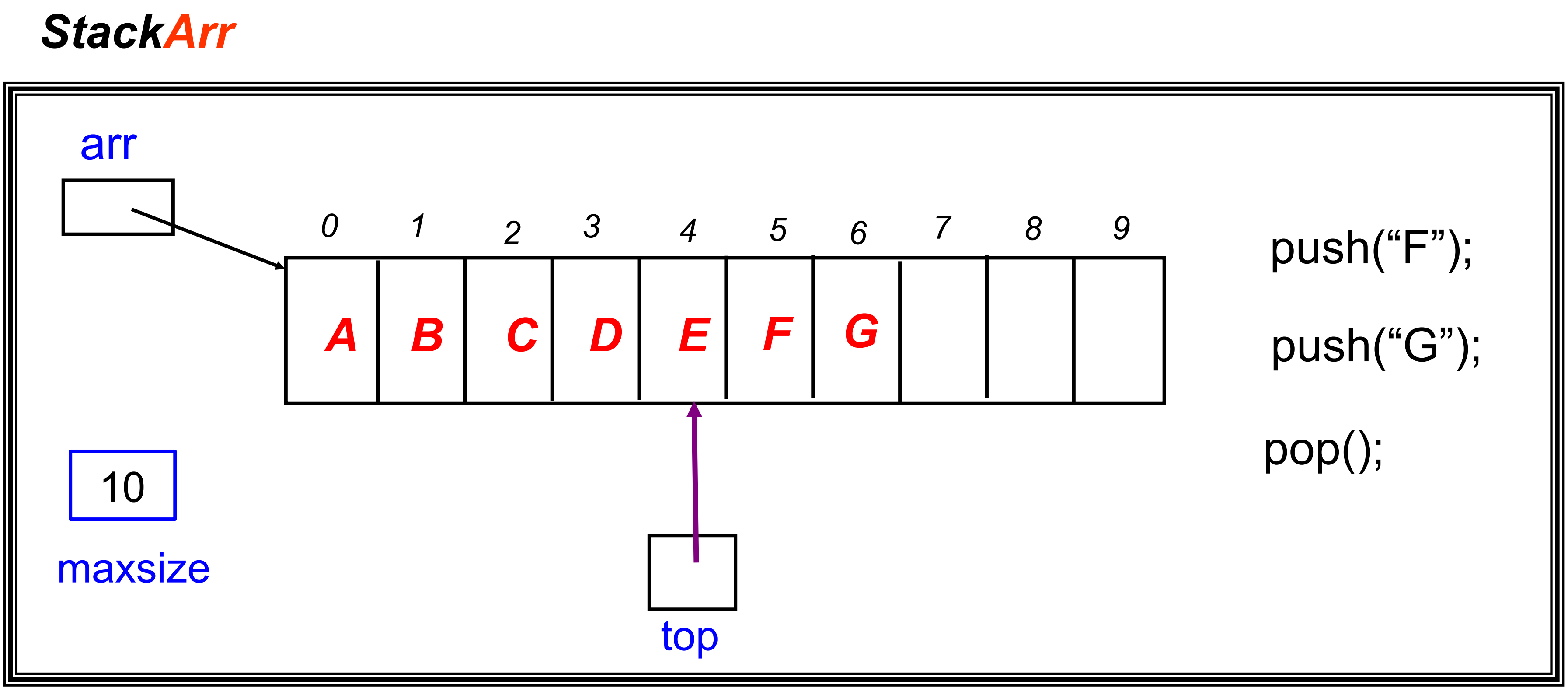
Q: Can "a" be replaced by 'a'?

A: Yes

B: No

2 Stack Implementation: Array (1/4)

- Use an Array with a top index pointer



2 Stack Implementation: Array (2/4)

StackArr.java

```
import java.util.*;

class StackArr <E> implements StackADT <E> {
  private E[] arr;
  private int top;
  private int maxSize;
  private final int INITSIZE = 1000;

  public StackArr() {
    arr = (E[]) new Object[INITSIZE]; // creating array of type E
    top = -1;  // empty stack - thus, top is not on an valid array element
    maxSize = INITSIZE;
  }

  public boolean empty() { if (top < 0) return true; else return
false;
    //return (top < 0);
  }
```

2 Stack Implementation: Array (3/4)

- pop() reuses peek()

StackArr.java

```
  public E peek() throws EmptyStackException {
    if (!empty()) return arr[top]; //if (empty() == false)
return arr[top];
    else throw new EmptyStackException();
  }

  public E pop() throws EmptyStackException {
    E obj = peek();
    top--;
    return obj;
  }
```

2 Stack Implementation: Array (4/4)

- push() needs to consider overflow

StackArr.java

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

 private void enlargeArr() {
   // When there is not enough space in the array
   // we use the following method to double the number
   // of entries in the array to accommodate new entry
   int newSize = 2 * maxSize;
   E[] x = (E[]) new Object[newSize];

   for (int j=0; j < maxSize; j++) {
     x[j] = arr[j];
   }
   maxSize = newSize;
   arr = x;
 }
}
```

private method

3 Stack Implementation: Linked List (1/6)

- A class can be defined in 2 ways:

via composition:

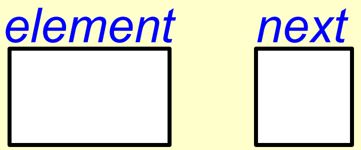
```
class A {
  B  b = new B (…);  // A is composed of instance of B
 …
 }
```

via inheritance:

```
class A extends B {   // A is an extension of B
    ….
}
```

Recall: ListNode (last week)

ListNode.java




```java
class ListNode <E> {
  /* data attributes */
  private E element;
  private ListNode <E> next;

  /* constructors */
  public ListNode(E item) { this(item, null); }

  public ListNode(E item, ListNode <E> n) {
    element = item;
    next = n;
  }

  /* get the next ListNode */
  public ListNode <E> getNext() { return next; }

  /* get the element of the ListNode */
  public E getElement() { return element; }

  /* set the next reference */
  public void setNext(ListNode <E> n) { next = n };
}
```

Recall: Basic Linked List (1/2) (last week)

BasicLinkedList.java

```
import java.util.*;
class BasicLinkedList <E> implements ListInterface <E> {
  private ListNode <E> head = null;
  private int num_nodes = 0;

  public boolean isEmpty() { return (num_nodes == 0); }

  public int size() { return num_nodes; }

  public E getFirst() throws NoSuchElementException {
    if (head == null)
      throw new NoSuchElementException("can't get from an empty list");
    else return head.getElement();
  }

  public boolean contains(E item) {
    for (ListNode <E> n = head; n != null; n = n.getNext())
      if (n.getElement().equals(item)) return true;
    return false;
  }
```

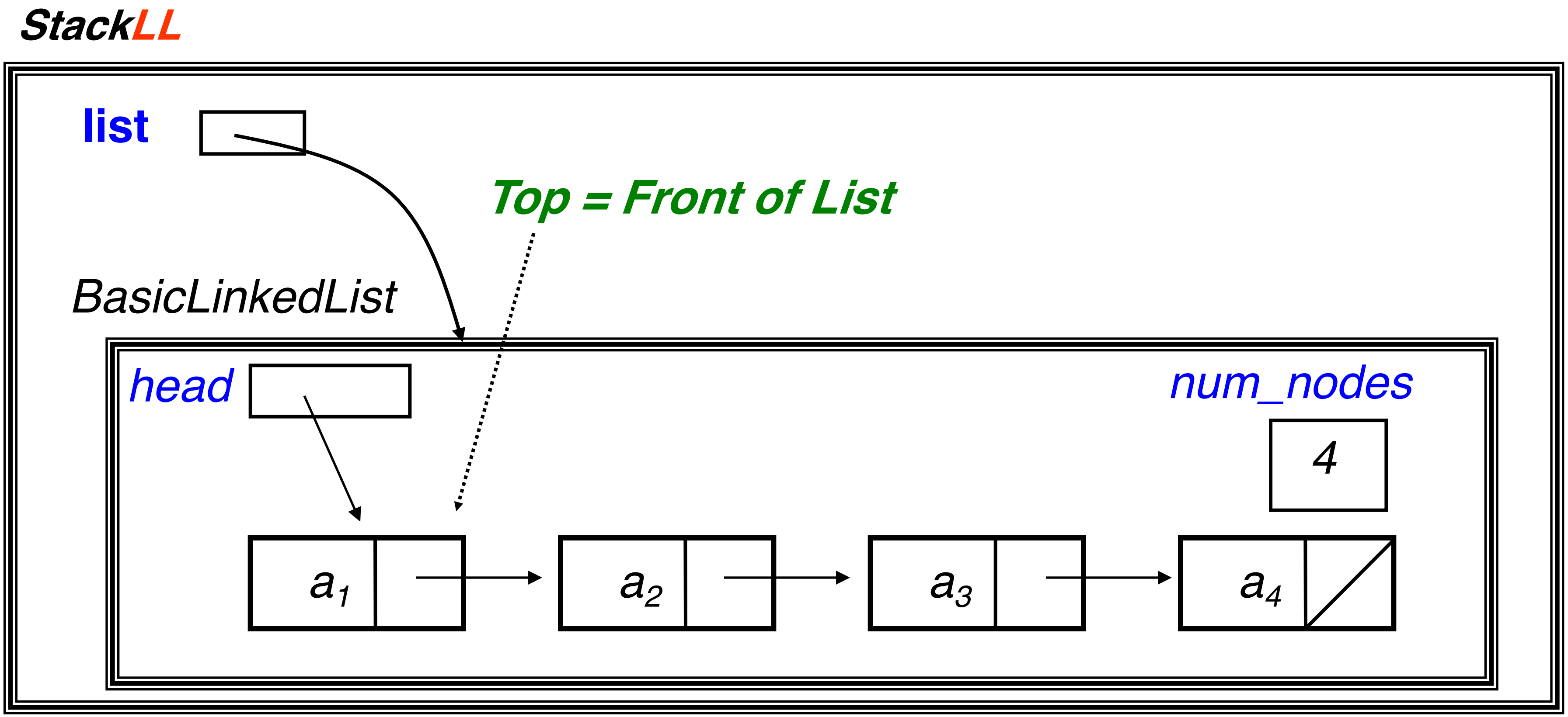
Recall: Basic Linked List (2/2) (last week)

BasicLinkedList.java

```
  public void addFirst(E item) {
    head = new ListNode <E> (item, head);
    num_nodes++;
  }

  public E removeFirst() throws NoSuchElementException {
    ListNode <E> ln;
    if (head == null)
      throw new NoSuchElementException("can't remove from empty list");
    else {
      ln = head;
      head = head.getNext();
      num_nodes--;
      return ln.getElement();
    }

  public void print() throws NoSuchElementException {
    // ... Code omitted
  }
}
```

3 Stack Implementation: Linked List (2/6)

- Method #1 (Composition): Use BasicLinkedList



3 Stack Implementation: Linked List (3/6)

- Method #1 (Composition): Use BasicLinkedList

StackLL.java

```
import java.util.*;

class StackLL <E> implements StackADT <E> {
  private BasicLinkedList <E> list; // Why private?

  public StackLL() {
    list = new BasicLinkedList <E> ();
  }

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

  public E peek() throws EmptyStackException {
    try {
      return list.getFirst();
    } catch (NoSuchElementException e) {
      throw new EmptyStackException();
    }
  }
```

3 Stack Implementation: Linked List (4/6)

- Method #1 (Composition): Use BasicLinkedList

StackLL.java

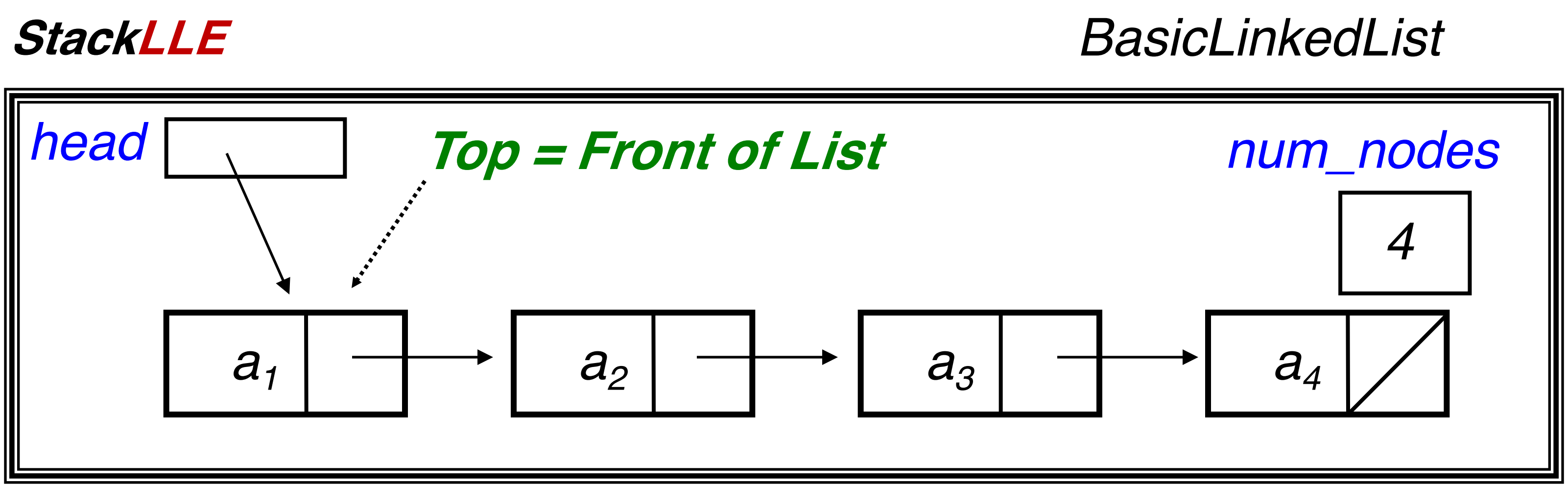
```
  public E pop() throws EmptyStackException {
    E obj = peek();
    list.removeFirst();
    return obj;
  }

  public void push(E o) {
    list.addFirst(o);
  }
}
```

Notes:

1. isEmpty(), getFirst(), removeFirst(), and addFirst() are public methods of BasicLinkedList.
2. **NoSuchElementException** is thrown by getFirst() or removeFirst() of BasicLinkedList.

3 Stack Implementation: Linked List (5/6)

- Method #2 (Inheritance): Extend BasicLinkedList



3 Stack Implementation: Linked List (6/6)

- Method #2 (Inheritance): Extend BasicLinkedList

StackLLE.java

```
import java.util.*;

class StackLLE <E> extends BasicLinkedList <E> implements StackADT <E> {
  public boolean empty() { return isEmpty(); }

  public E peek() throws EmptyStackException {
    try {
      return getFirst();
    } catch (NoSuchElementException e) {
      throw new EmptyStackException();
    }
  }

  public E pop() throws EmptyStackException {
    E obj = peek();
    removeFirst();
    return isEmpty();
  }

  public void push (E o) { addFirst(o); }
}
```

3 Uses of Stack

TestStack.java

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

    // You can use any of the following 4 implementations of Stack
    StackArr <String> stack = new StackArr <String>();  // Array
    //StackLL <String> stack = new StackLL <String>(); // LinkedList composition
    //StackLLE <String> stack = new StackLLE <String>(); // LinkedList inheritance
    //Stack <String> stack = new Stack <String>();  // 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());
  }
}
```

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 (1/2)

- 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

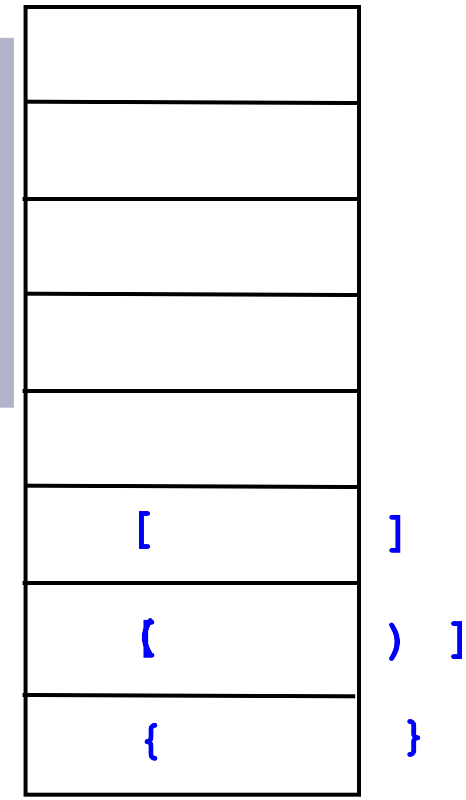
`[..(..]..)` // mismatched brackets

5 Application 1: Bracket Matching (2/2)

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



Stack

Example

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

5 Applic^{n} 2: Arithmetic Expression (1/7)

- 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: Arithmetic Expression (2/7)

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 * +

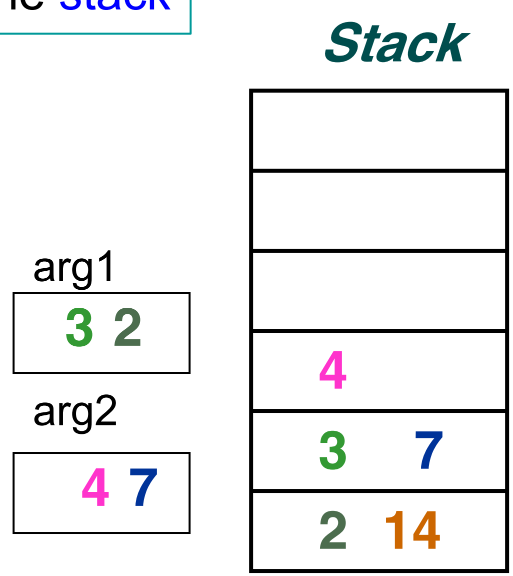
5 Applic^n 2: Arithmetic Expression (3/7)

Algorithm: Calculating 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)



5 Applic^n 2: Arithmetic Expression (4/7)

Brief steps for Infix to Postfix Conversion

1. Scan infix expression from left to right
2. If an operand is found, add it to the postfix expression.
3. If a "(" is found, push it onto the stack.
4. If a ")" is found
   a) repeatedly pop the stack and add the popped operator to the postfix expression until a "(" is found.
   b) remove the "(".
5. If an operator is found
   a) repeatedly pop the operator from stack which has higher or equal precedence than/to the operator found, and add the popped operator to the postfix expression.
   b) add the new operator to stack
6. If no more token in the infix expression, repeatedly pop the operator from stack and add it to the postfix expression.

5 Applic^{n} 2: Arithmetic Expression (5/7)

Algorithm: Converting Infix to an equivalent Postfix

```
String postfixExp = "";
for (each character ch in the infix expression) {
  switch (ch) {
    case operand: postfixExp = postfixExp + ch; break;
    case '(': stack.push(ch); break;
    case ')':
      while ( stack.peek() != '(' )
        postfixExp = postfixExp + stack.pop();
      stack.pop(); break;     // remove '('
    case operator:
      while ( !stack.empty() && stack.peek() != '(' &&
              precedence(ch) <= precedence(stack.peek()) ) // Why "<="?
        postfixExp = postfixExp + stack.pop();
      stack.push(ch); break;
  } // end switch
} // end for

while  ( !stack.empty() )
  postfixExp = postfixExp + stack.pop();
```

5 Applic^{n} 2: Arithmetic Expression (6/7)

Algorithm: Converting Infix to an equivalent Postfix

ch	Stack (bottom to top)	postfixExp
a		a
–	–	a
(	– (	a
b	– (	a b
+	– (+	a b
c	– (+	a b c
*	– (+ *	a b c
d	– (+ *	a b c d
)	– (+	a b c d *
	– (	a b c d * +
	–	a b c d * +
/	– /	a b c d * +
e	– /	a b c d * + e
		a b c d * + e / –

Example: a – (b + c * d) / e

Move operators from stack to postfixExp until '('

Copy remaining operators from stack to postfixExp

5 Applic^n 2: Arithmetic Expression (7/7)

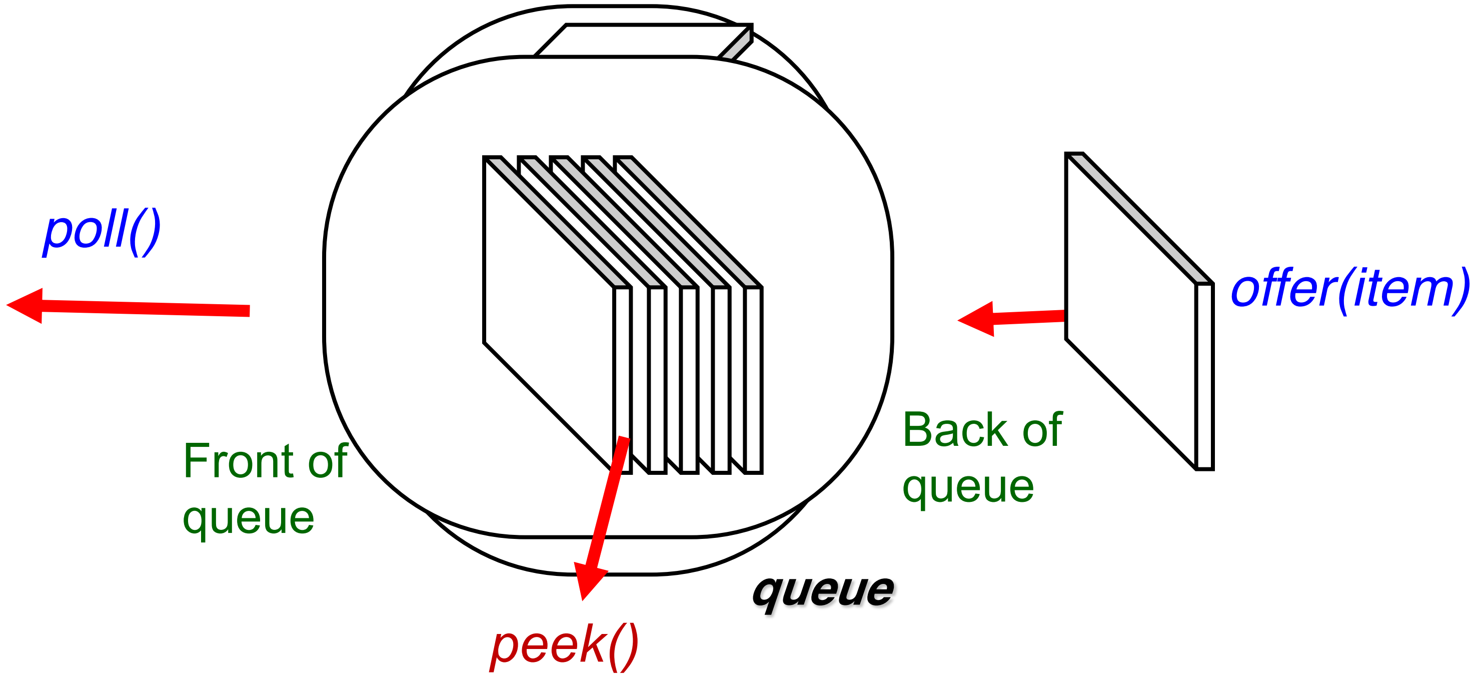
- How to code the above algorithm in Java?
  - Complete PostfixIncomplete.java
  - Answer in subdirectory "/answers", but try it out yourself first.
- How to do conversion of infix to prefix?
  - See Prefix.java

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 trees
- Checking palindromes - for illustration only as it is not a real application of queue

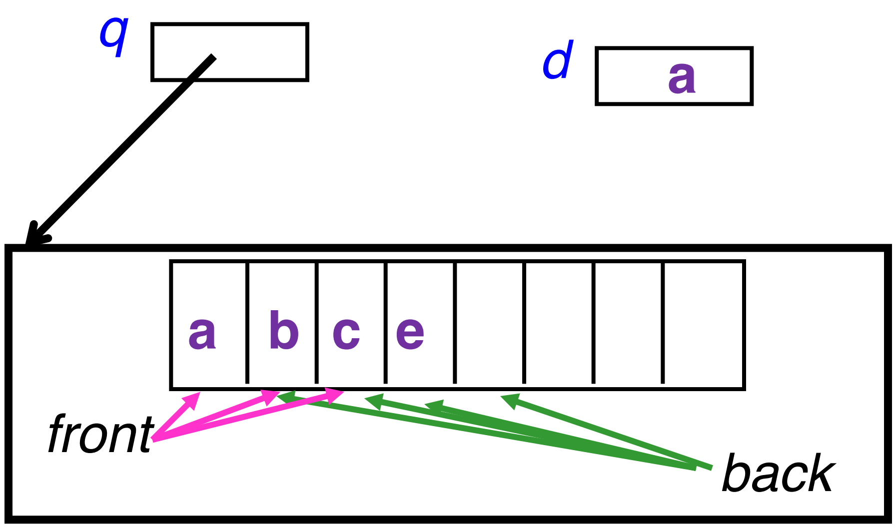
6 Queue ADT: Interface

QueueADT.java

```
import java.util.*;

public interface QueueADT <E> {

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

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

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

  // add item to the back of the queue
  public boolean offer(E item); // also commonly known as enqueue
}
```

6 Queue: Usage

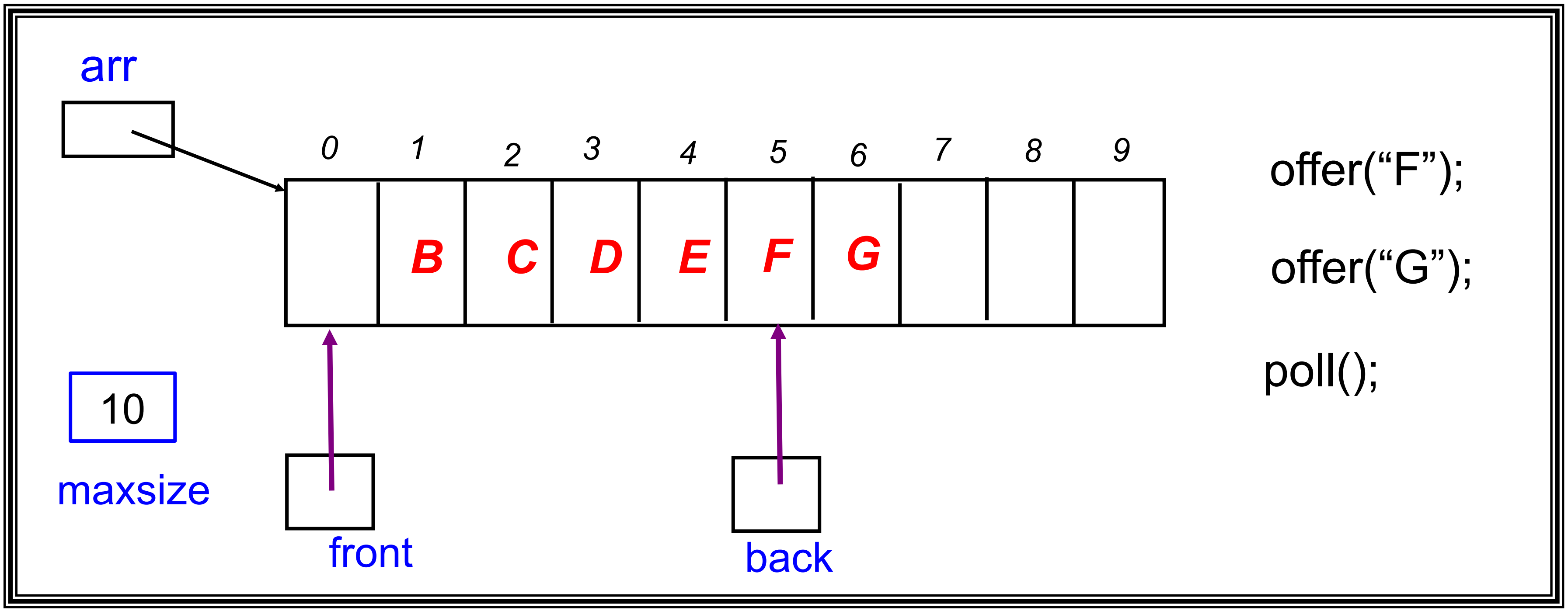
```
Queue q = new Queue ();
q.offer ("a");
q.offer ("b");
q.offer ("c");
d = q.peek ();
q.poll ();
q.offer ("e");
q.poll ();
```



7 Queue Implementation: Array (1/7)

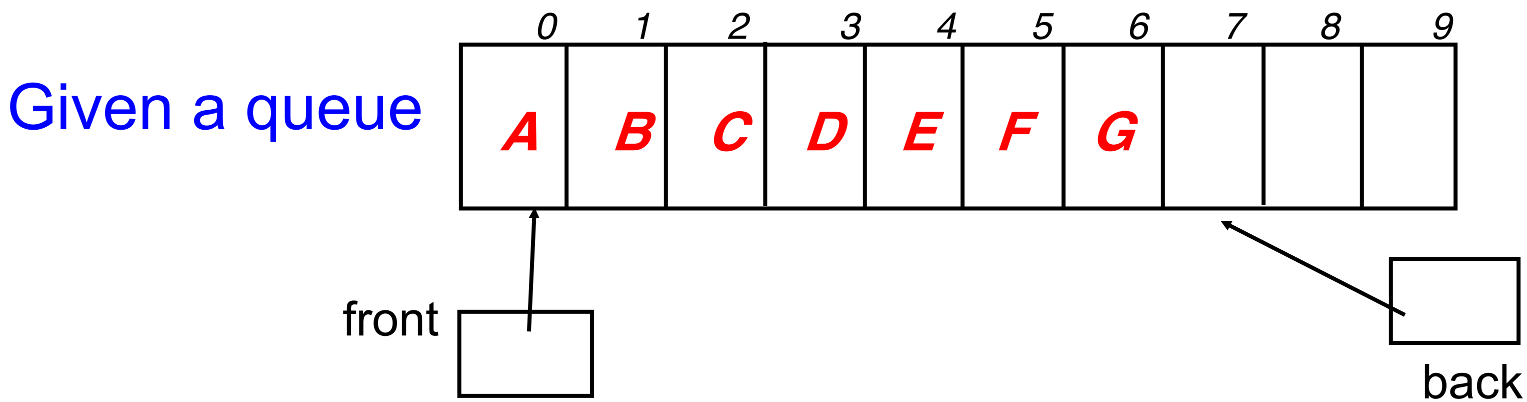
- Use an Array with front and back pointer

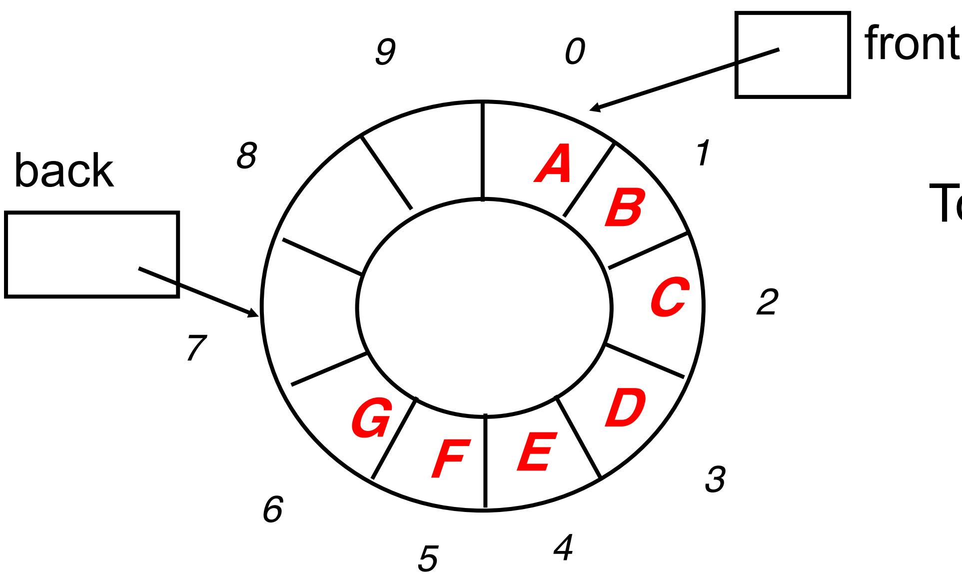
QueueArr



7 Queue Implementation: Array (2/7)

- "Circular"Array needed to recycle space





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

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 **(Preferred and 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 <E> implements QueueADT <E> {
  private E [] arr;
  private int front, back;
  private int maxSize;
  private final int INITSIZE = 1000;

  public QueueArr() {
    arr = (E []) new Object[INITSIZE]; // create array of E objects
    front = 0; // the queue is empty
    back = 0;
    maxSize = INITSIZE;
  }

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

7 Queue Implementation: Array (6/7)

QueueArr.java

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

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

public boolean offer(E o) {  // add item to the back of the queue
  if (((back+1)%maxSize) == front) // array is full
    if (!enlargeArr()) return false; // no more memory to
                                     // enlarge the array
  arr[back] = o;
  back = (back + 1) % maxSize;   // "circular" array
  return true;
}
```

7 Queue Implementation: Array (7/7)

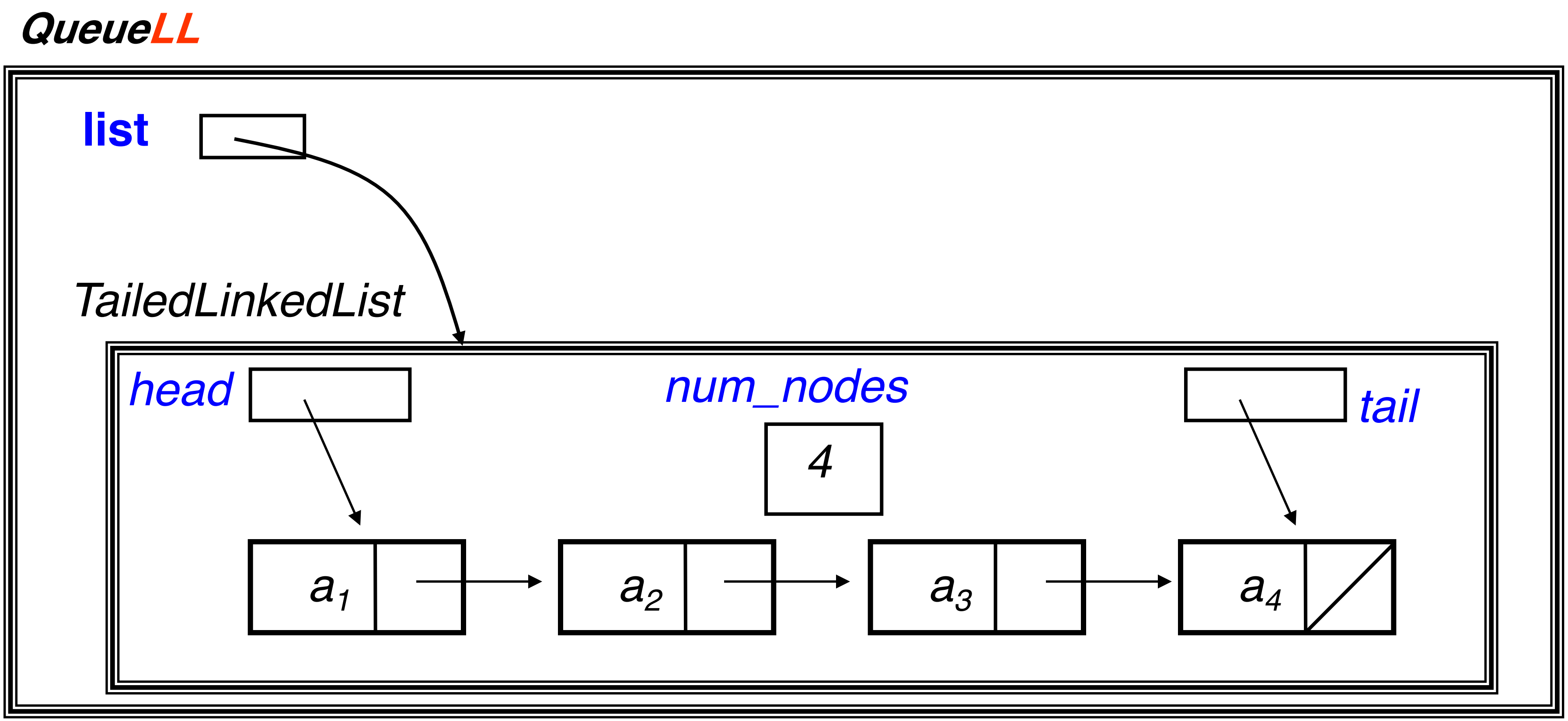
private method

QueueArr.java

```
  private boolean enlargeArr() {
    int newSize = maxSize * 2;
    E[] x = (E []) new Object[newSize];
    if (x == null) // i.e. no memory allocated to array of E objects
      return false;

    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?
      x[j] = arr[(front+j) % maxSize];
    }
    front = 0;
    back = maxSize - 1;
    arr = x;
    maxSize = newSize;
    return true;
  }
}
```

8 Queue Implementⁿ: Linked List (1/4)

- Method #1 (Composition): Use TailedLinkedList
  - Do not use BasicLinkedList as we would like to use addLast() of TailedLinkedList.



8 Queue Implement^{n}: Linked List (2/4)

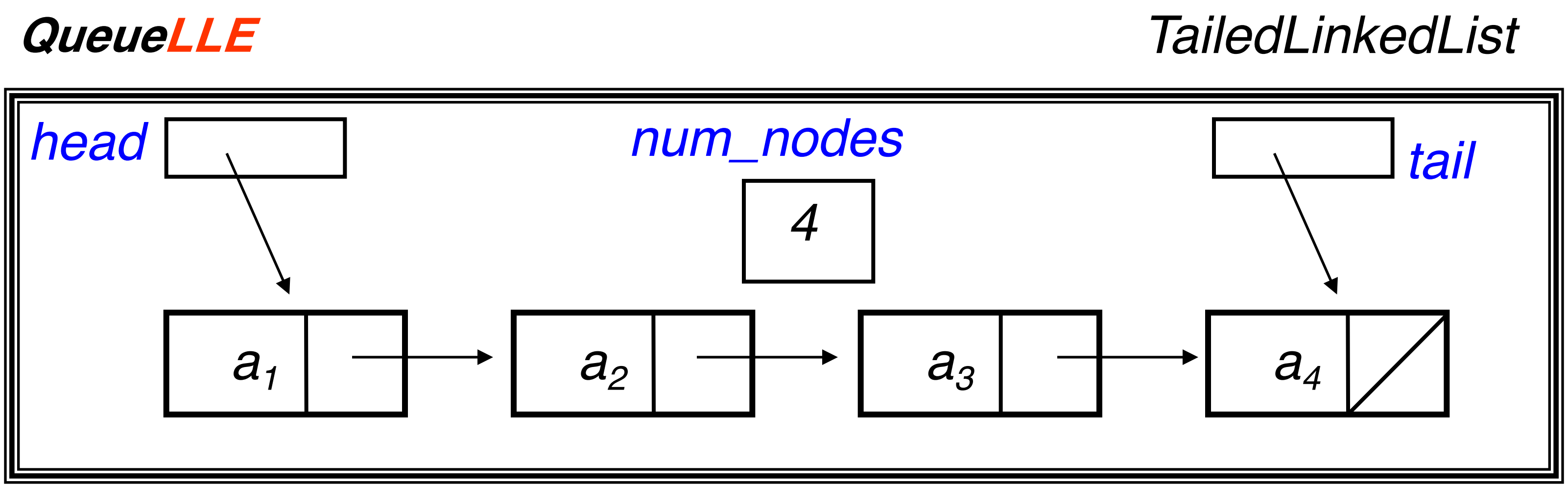
- Method #1 (Composition): Use TailedLinkedList

QueueLL.java

```
import java.util.*;
class QueueLL <E> implements QueueADT <E> {
  private TailedLinkedList <E> list;
  public QueueLL() { list = new TailedLinkedList <E> (); }
  public boolean isEmpty() { return list.isEmpty(); }

  public boolean offer(E o) {
    list.addLast(o);    // isEmpty(), addLast(), getFirst(), removeFirst()
                        // are public methods of TailedLinkedList
    return true;
  }
  public E peek() {
    if (isEmpty()) return null;
    return list.getFirst();
  }
  public E poll() {
    E obj = peek();
    if (!isEmpty()) list.removeFirst();
    return obj;
  }
}
```

8 Queue Implement^n: Linked List (3/4)

- Method #2 (Inheritance): Extend TailedLinkedList



8 Queue Implement^{n}: Linked List (4/4)

- Method #2 (Inheritance): Extend TailedLinkedList

QueueLLE.java

```
import java.util.*;

class QueueLLE <E> extends TailedLinkedList <E> implements QueueADT <E> {
  public boolean offer(E o) {
    addLast(o);
    return true;
  }
  public E peek() {
    if (isEmpty()) return null;
    return getFirst();
  }

  public E poll() {
    E obj = peek();
    if (!isEmpty()) removeFirst();
    return obj;
  }
}
```

8 Uses of Queues (1/2)

TestQueue.java

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

   System.out.println("queue is empty? " + queue.isEmpty());
   queue.offer("1");
   System.out.println("operation: queue.offer(\"1\")");
   System.out.println("queue is empty? " + queue.isEmpty());
   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());
```

8 Uses of Queues (2/2)

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());
    }
}
```

9 java.util.interface Queue <E>

Method Summary

E	element() Retrieves, but does not remove, the head of this queue.
boolean	offer(E o) Inserts the specified element into this queue, if possible.
E	peek() Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
E	poll() Retrieves and removes the head of this queue, or `null` if this queue is empty.
E	remove() Retrieves and removes the head of this queue.

Methods inherited from interface java.util.Collection

add, addAll, clear, contains, containsAll, equals, hashCode, isEmpty, iterator, remove, removeAll, retainAll, size, toArray, toArray

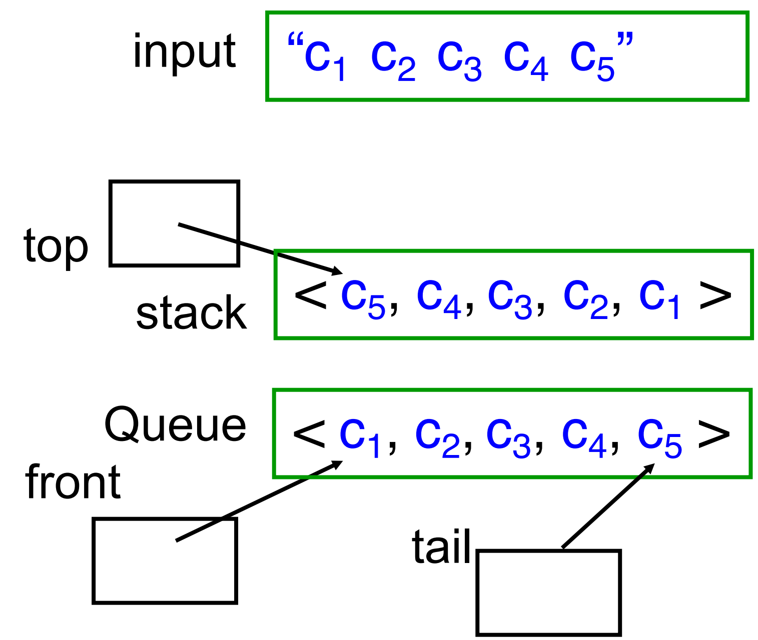
Note: The methods "E element()" and "E remove()" are not in our own Queue ADT .

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

10 Application: Palindromes (2/3)

Palindromes.java

```
import java.util.*;
public class Palindromes {

  public static void main (String[] args) throws NoSuchElementException {
    // you can use any of the following stack/queue implementations
    // and Java classes Stack and LinkedList
    //StackLLE <String> stack = new StackLLE <String> ();
    Stack <String> stack = new Stack <String> (); // Stack is a Java class
    //StackLL <String> stack = new StackLL <String> ();
    //StackArr <String> stack = new StackArr <String> ();
    //QueueLL <String> queue = new QueueLL <String> ();
    //QueueLLE <String> queue = new QueueLLE <String> ();
    //QueueArr <String> queue = new QueueArr <String> ();
    LinkedList <String> queue = new LinkedList <String> ();

    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter text: ");
    String inputStr = scanner.next();
    for (int i=0; i < inputStr.length(); i++) {
      String ch = inputStr.substring(i, i+1);
      stack.push(ch);
      queue.offer(ch);
    }
```

LinkedList is a Java class that implements interface Queue and other interfaces, such as Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>.

10 Application: Palindromes (3/3)

Palindromes.java

```
    boolean ans = true;
    try {
      while (!stack.isEmpty() && ans) {
        if (!(stack.pop().equals(queue.poll())))
              ans = false;
      }
    } catch (NoSuchElementException  e) {
      throw new NoSuchElementException();
    }

    System.out.print(inputStr + " is ");
    if (ans)
      System.out.println("a palindrome");
    else
      System.out.println("NOT a palindrome");
  }
}
```

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 cases still
- Please do not forget that the Java Library class is much more comprehensive than our own – for sit-in lab or exam, please use the one as told.

End of file