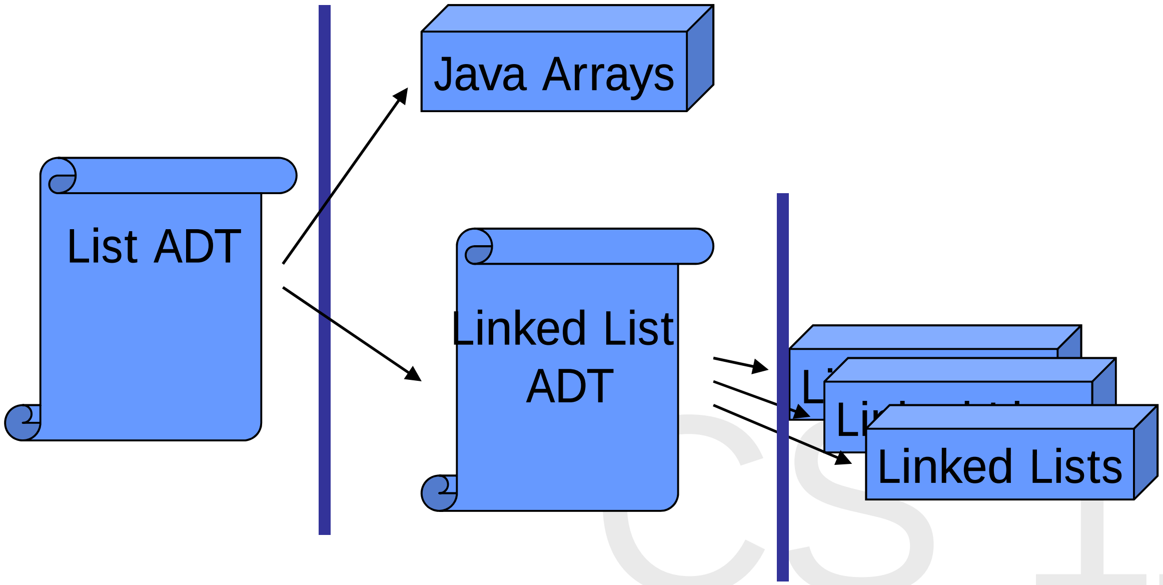
Lecture 5

Stacks and Queues

Recap: ADTs for List and Linked List

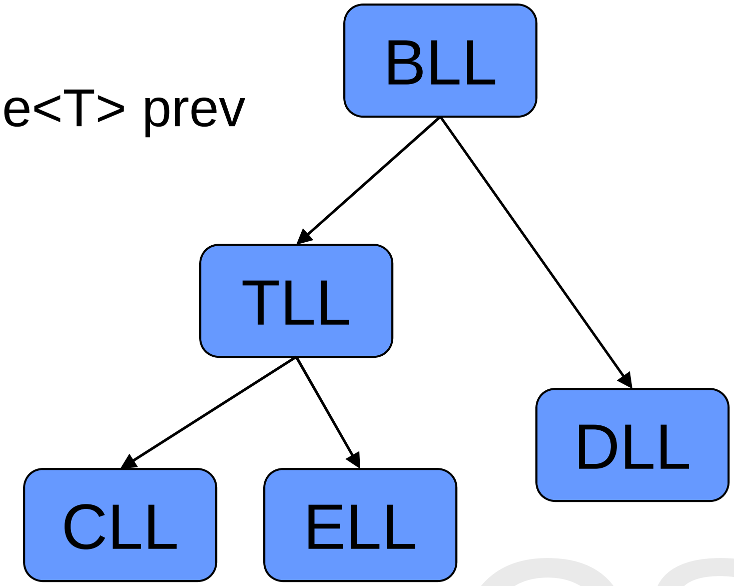
- When to use arrays, when to use linked lists
- Variants: tail pointer, doubly linked, circular
- Implementing with Object or Generics



Recap: Class Hierarchy

- ListNode
  - ListNode<T> next
  - T element
- DListNode

  …
  - DListNode<T> prev



- BasicLinkedList
  - ListNode<T> head
  - Int num_nodes
- ExtendedLinkedList
- TailLinkedList
  - ListNode<T> tail
- DoublyLinkedList
  - DListNode<T> head
  - DListNode<T> tail
- CircularLinkedList

Readings

- Chapter 7: Stacks

  Pages 327-364

  (Leaves out recursion)

- Chapter 8: Queues

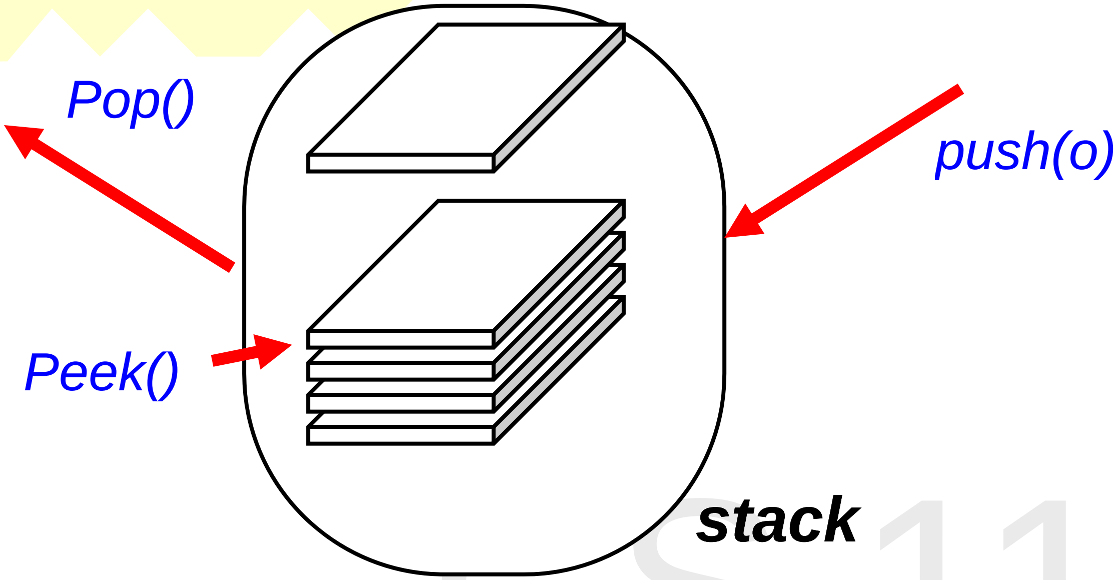
  Pages 381-413

Stack Outline

- What is a Stack?
- Stack ADT
- Various Stack implementations
- Applications
  - Bracket Matching
  - Postfix Calculation

What is a Stack?

- A Stack is a collection of data that is accessed in a last-in-first-out (LIFO) manner.
- Two operations: ‘push’ and ‘pop’.



Stacks are useful

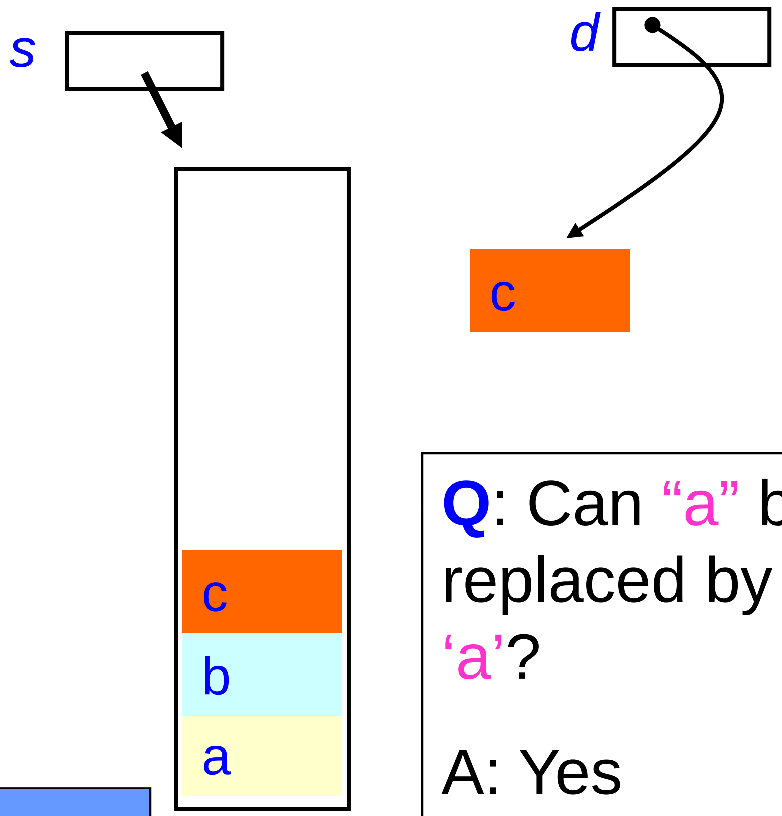
- Calling a function
  - Before the call, the state of computation is saved on the stack so that we will know where to resume
- Recursion (we’ll see this next lecture)
- Matching parentheses
- Evaluating algebraic expressions (e.g. a+b-c)
- Traversing a maze

Stack ADT as an interface

```
public interface StackADT {
// A collection of objects managed by  the following methods:

// true if empty
public boolean isEmpty ();
// insert object o into stack
public void push (Object o);
// remove and return topmost item
public Object pop () throws Underflow;
// retrieves topmost item
public Object peek () throws Underflow;
}

public class Underflow extends Exception { // Companion Exception
    public Underflow (String s) { super(s); }
}
```

Example of 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 ();
```



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

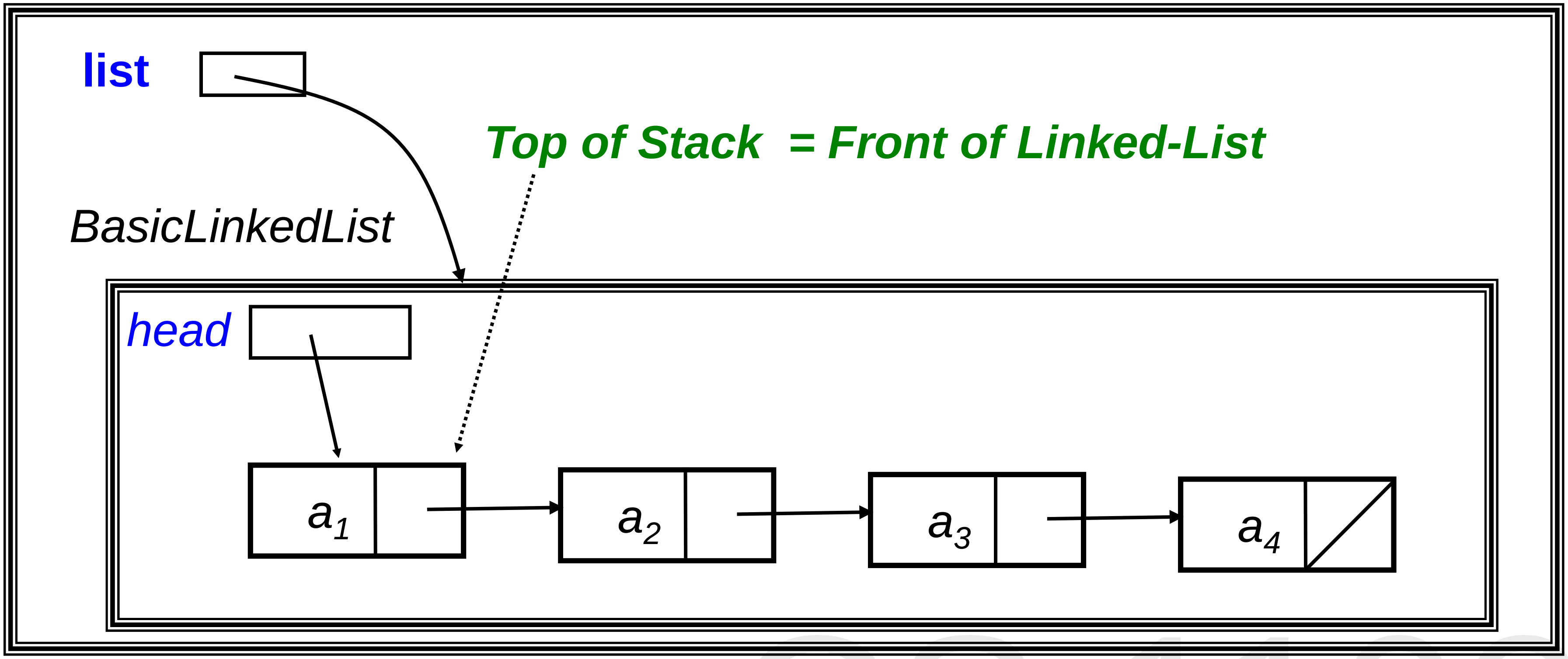
A: Yes

B: No

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

Stacks Implemented with Linked Lists

***Stack**LL*



Defining a class

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
```

Via Composition

```
class StackLL implements StackADT {
  private BasicLinkedList list;                    // composition

  public StackLL () { list = new BasicLinkedList(); }
  public boolean isEmpty () { return list.isEmpty (); }
  public void push (Object o) { list.addHead (o); }
  public Object pop () throws Underflow {
    Object obj = peek();
    list.deleteHead ();
    return obj; }
  public Object peek () throws Underflow {
    try {
      return list.getHeadElement ();
    } catch (ItemNotFoundException e) {
      throw new Underflow (“Illegal operation on empty stack”);
} } }
```

Via Inheritance

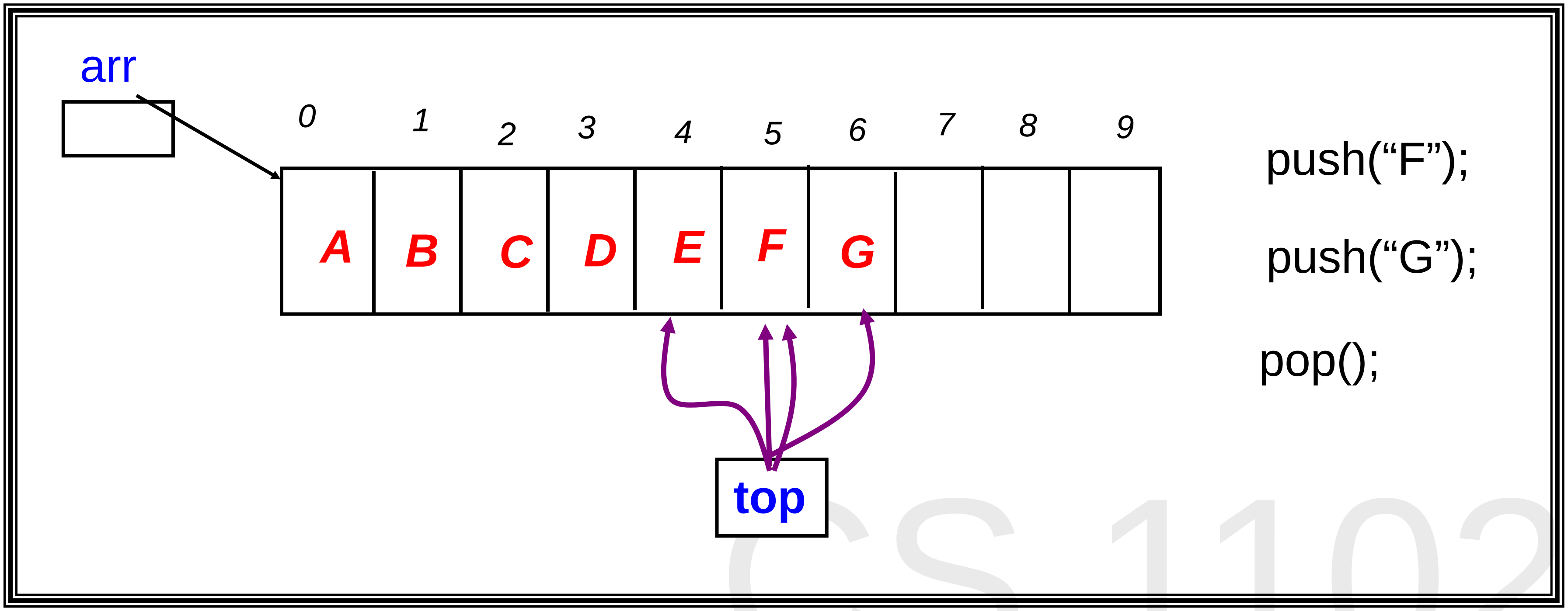
```
class StackLLE extends BasicLinkedList implements StackADT {
  public boolean isEmpty () { return super.isEmpty (); } // can remove too
  public void push (Object o) { addHead (o); }

  public Object pop () throws Underflow {
    Object obj = peek ();
    try { deleteHead (); return obj;
    } catch (ItemNotFoundException e) {
      throw new Underflow (“Illegal operation on empty stack”);
  }

  public Object peek () throws Underflow {
    try {
      return getHeadElement ();
    } catch (ItemNotFoundException e) {
      throw new Underflow (“Illegal operation on empty stack”);
}}}
```

Saying a stack is
a type of List, rather than
a stack has a list inside.

Stack Implemented with Array

- Can use an Array with a top index pointer as an implementation of stack

StackArr



push("F");

push("G");

pop();

Array implementation of Stack

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

  public StackArr () {
    arr = new Object[INITSIZE];
    top = -1;
    maxSize = INITSIZE ;
  }

  public boolean isEmpty () {
      return (top < 0);
  }
  // more on next slide
```

Array implementation of Stack (cont)

```
// continued from last slide
public Object pop () throws Underflow {
  Object obj = peek  ();
  top--;
  return obj;
}

public Object peek () throws Underflow {
  if (!isEmpty ()) { return arr[top]; }
  else throw new Underflow (“Illegal op on empty stack”);
}

public void push (Object obj) {
  if (top >= maxSize-1) enlargeArr();
  top++;
  arr[top] = obj;
}
```

Enlarging the array

```
private void enlargeArr () {
  // double the max size
  int newSize = 2*maxSize;
  Object [] x = new Object[newSize];

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

Implementations of Stacks

- Array based (pages 341-343)
- Linked List based (pages 343-345)
- List ADT based (pages 346-347)

java.util.Stack<E>

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.

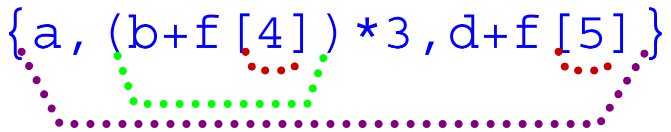
Stack Applications

- Many stack applications:
- line editing (see textbook)
- function call stack
- bracket matching
- postfix calculation
- infix to postfix conversion

Application: Bracket Matching

Ensures that pairs of brackets are properly matched

An example:



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

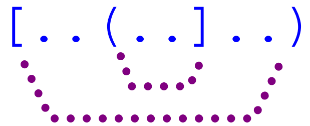
Incorrect examples:

```
(..)..)
```

// too many close brackets

```
(..(..)
```

// too many open brackets



```
[..(..]..)
```

// mismatched brackets

Bracket Matching

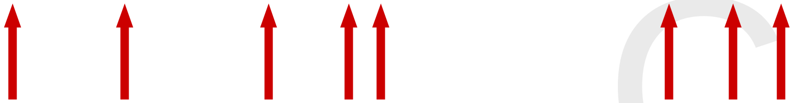
```
create empty stack
for every char read
  if open bracket then
    push onto stack
  if close bracket, then
    pop from the stack
    if doesn’t match or underflow then
      flag error
if stack is not empty then flag error
```

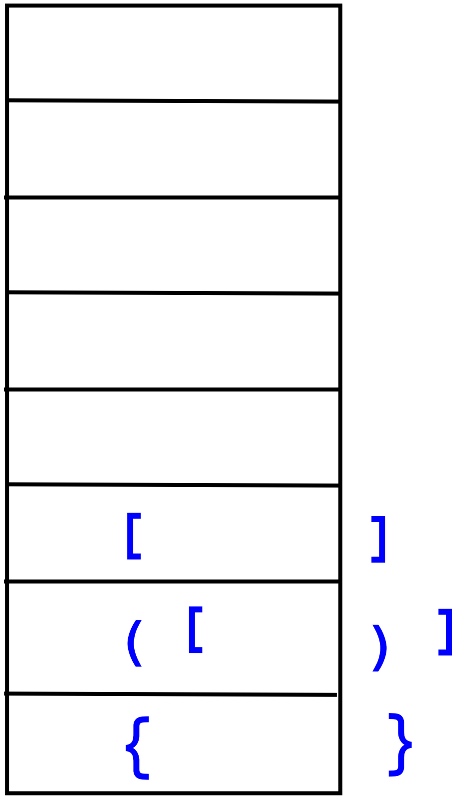
Q: What type of error does the last line test for?

A: too many closing brackets
B: too many opening brackets
C: bracket mismatch

Example

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





Stack

Expression Parsing

Expression:	a = b + c
Operands:	a, b, c
Operators:	=,+

Other operators:

- +, -, *, /, %
- =, !

Infix, Prefix, and Postfix Notation

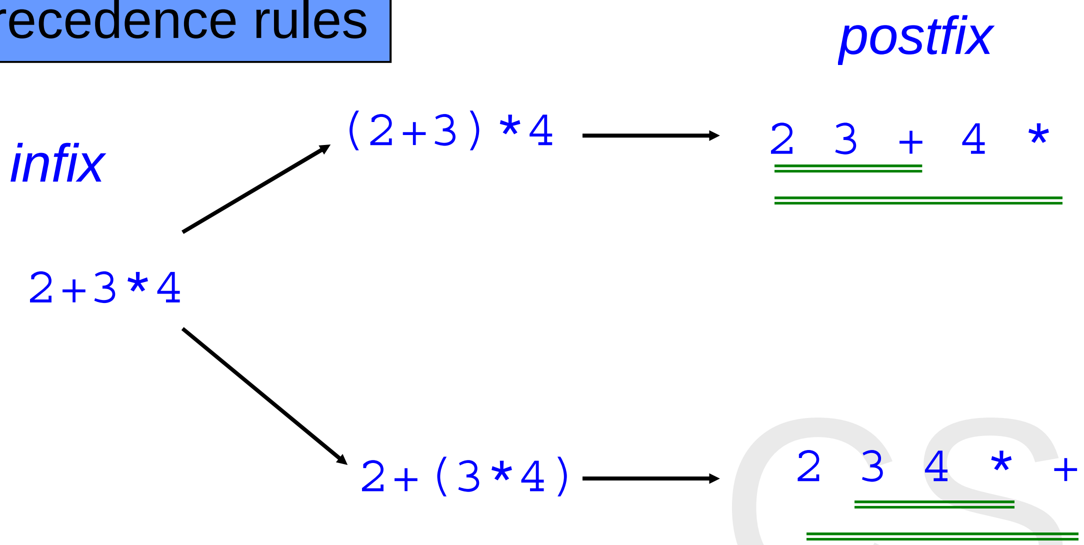
Infix - operand1 operator operand2

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 * +

Q: What is the bottom line an example of?

A: Prefix notation
B: Infix ambiguity
C: Postfix ambiguity

Postfix Calculation

Arithmetic expressions can be efficiently computed for postfix notation, with the help of a 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
```

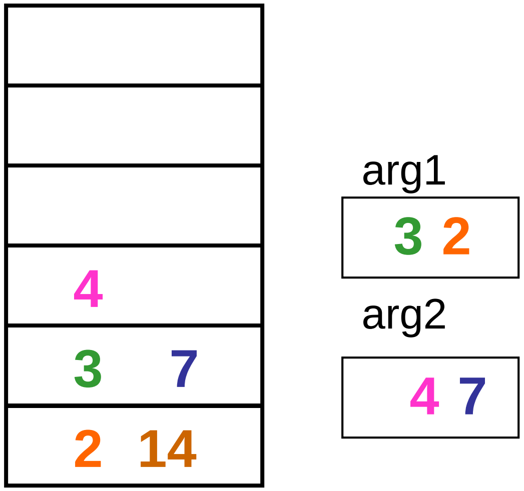
Q: Is there anything wrong with this last line?

A: Nope. It’s correct
B: It should be outdented to
    be aligned with the “If”s
C: It shouldn’t be there at all

Evaluating Postfix Expressions

2 * (3 + 4) ⟶ **2 3 4 + ***

Expression	
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)



Stack

Precedence Rules

- The precedence rules can be implemented in a table by assigning an appropriate level number to each operator
- This table can be found in many books

*** /** have higher precedence over **+ -**

Operators at the same level:
Associate from left to right

Operator	Level no.
*	5
/	5
+	3
-	3

Converting Infix to 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 (top of stack is not ‘(‘)
        postfixExp = postfixExp + stack.pop();
      stack.pop(); break;                      // remove ‘(‘
    case operator:
      while (!stack.isEmpty() && top of stack is not ‘(‘ &&
              precedence(ch) <= precedence(top of stack) )
        postfixExp = postfixExp + stack.pop();
      stack.push(ch); break;
  } // end switch
} // end for
while (!stack.isEmpty())
      postfixExp = postfixExp + stack.pop();
```

Example: Infix to 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

To think about: What about conversion to prefix?

Move operators from stack to postfixExp until "("

Copy remaining operators from stack to postfixExp

Queues

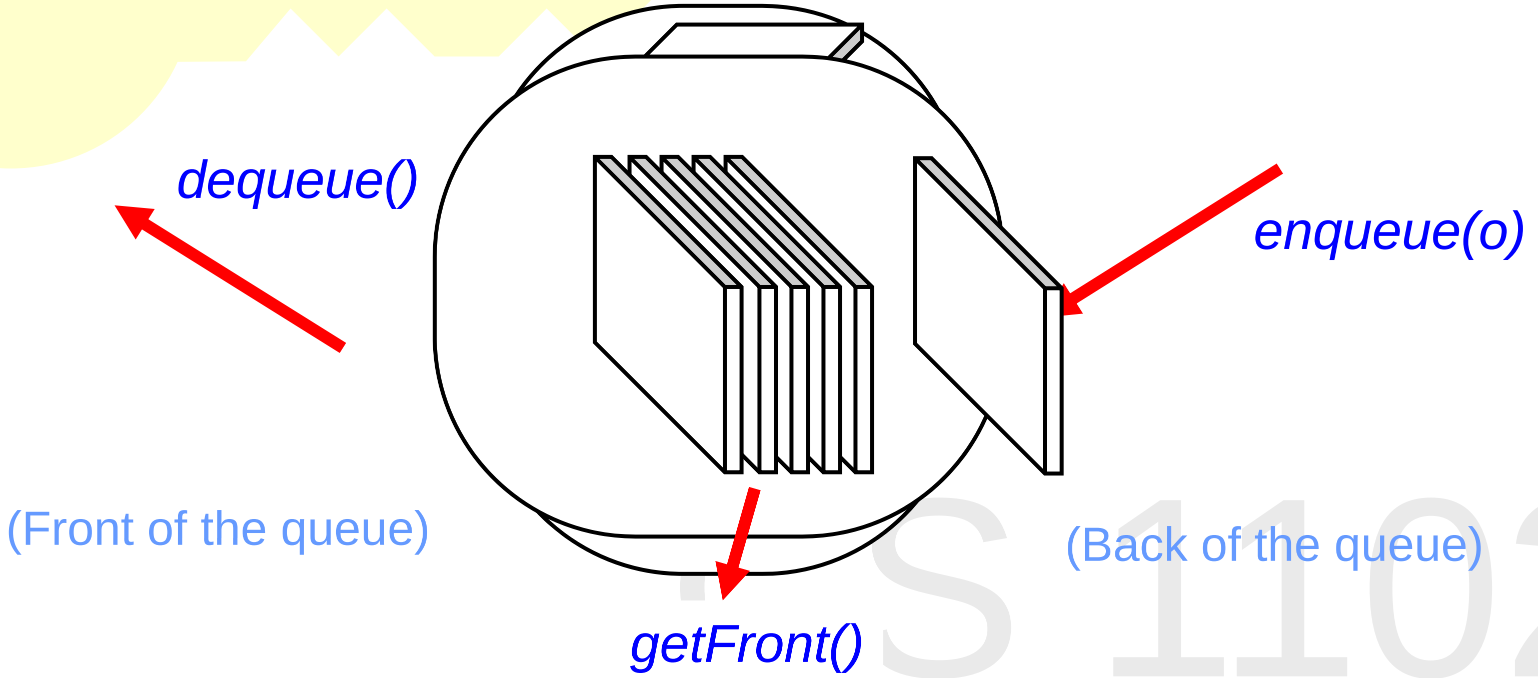
Chapter 8, pages 381-413

Outline

- What is a Queue?
- Queue ADT
- Various Queue Implementations
- Applications

What is a Queue?

- A Queue is a collection of data that is accessed in a first-in-first-out (FIFO) manner.
- Two operators: ‘enqueue’ and ‘dequeue’

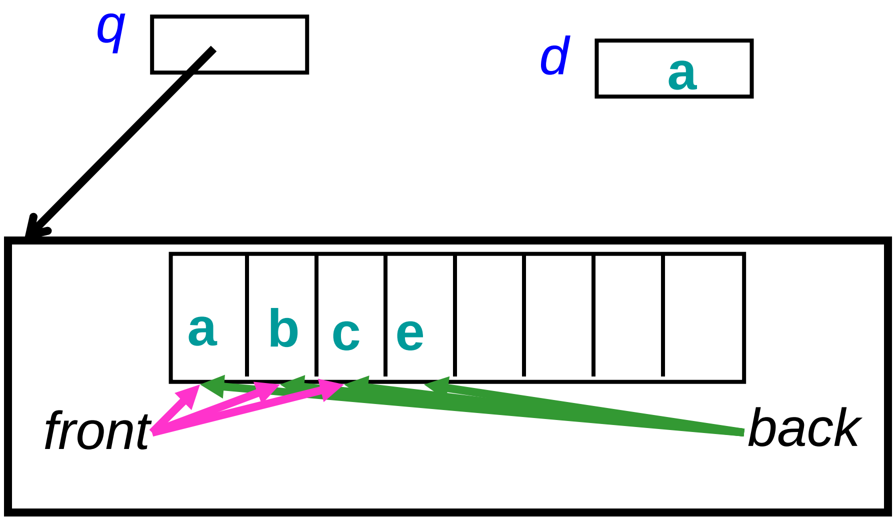


Queue ADT

```
public interface QueueADT {
  // A collection of objects managed by the following methods:

  // Insert element o at rear
  public void enqueue (Object o);
  // Remove and return front element
  public Object dequeue () throws Underflow;
  // Returns front element
  public Object getFront () throws Underflow;
  // Returns true if queue has no elements
  public boolean isEmpty ();
}
public class Underflow extends Exception
{ public Underflow (String s) { super(s); }}
```

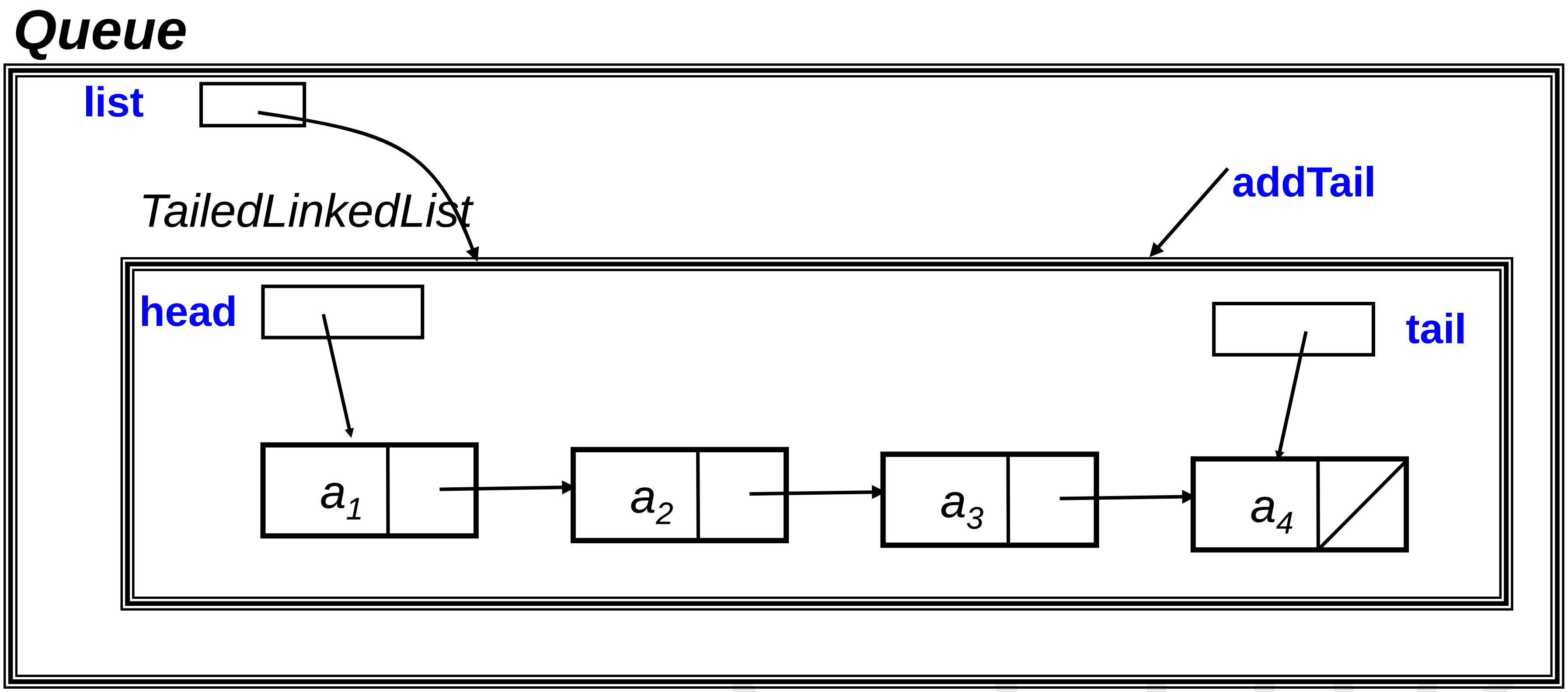
Sample run

```
Queue q = new Queue ();
q.enqueue ("a");
q.enqueue ("b");
q.enqueue ("c");
d = q.getFront ();
q.dequeue ();
q.enqueue ("e");
q.dequeue ();
```



Queue Implemented with Linked List

- Can use TailedLinkedList as underlying implementation of Queues



Via Composition

```
class QueueLL implements QueueADT {
  private TailedLinkedList list;      // composition

  public QueueLL () { list = new TailedLinkedList(); }
  public boolean isEmpty () { return list.isEmpty (); }
  public void enqueue (Object o) { list.addTail (o); }
  public Object dequeue () throws Underflow {
    Object obj = getFront ();
    list.deleteHead ();
    return obj;
  }
  public Object getFront () throws Underflow {
    try {
      return list.getHeadElement();
    } catch (ItemNotFoundException e) {
      throw new Underflow (“Illegal operation on empty queue”);
  } } }
```

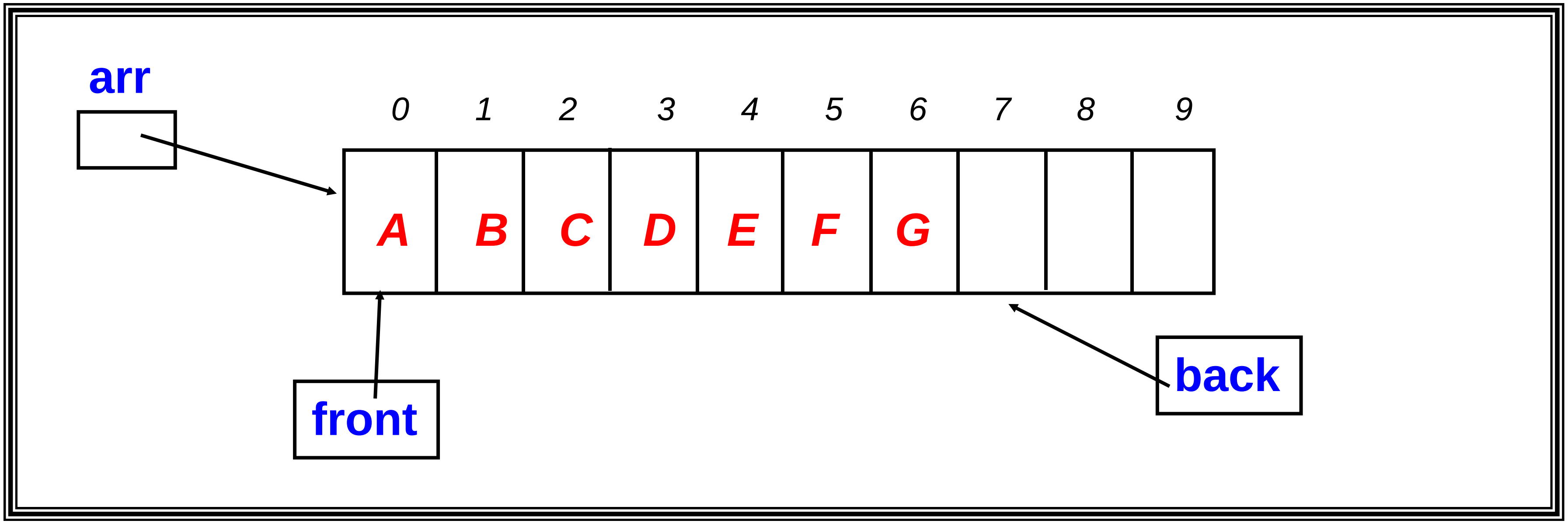
Via Inheritance

```
class QueueLLE extends TailedLinkedList implements QueueADT {
  public void enqueue (Object o) { addTail (o); }
  public Object dequeue () throws Underflow {
    Object obj = getFront ();
    deleteHead ();
    return obj;
  }
  public Object getFront () throws Underflow {
    try {
      return getHeadElement();
    } catch (ItemNotFoundException e) {
      throw new Underflow (“Illegal operation on empty queue”);
    }
  } }
```

Array implementation of Queue

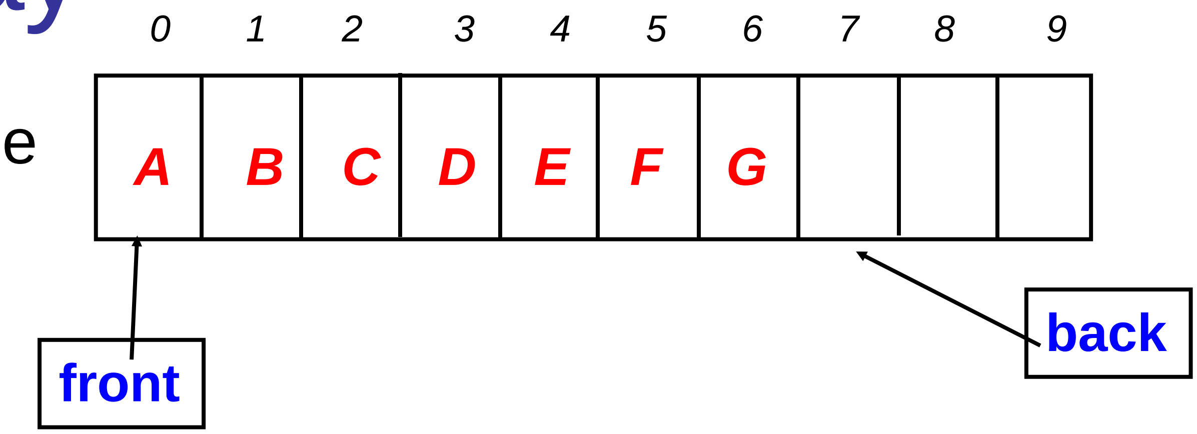
- Can use an array with front and back pointers to implement a queue

QueueArr

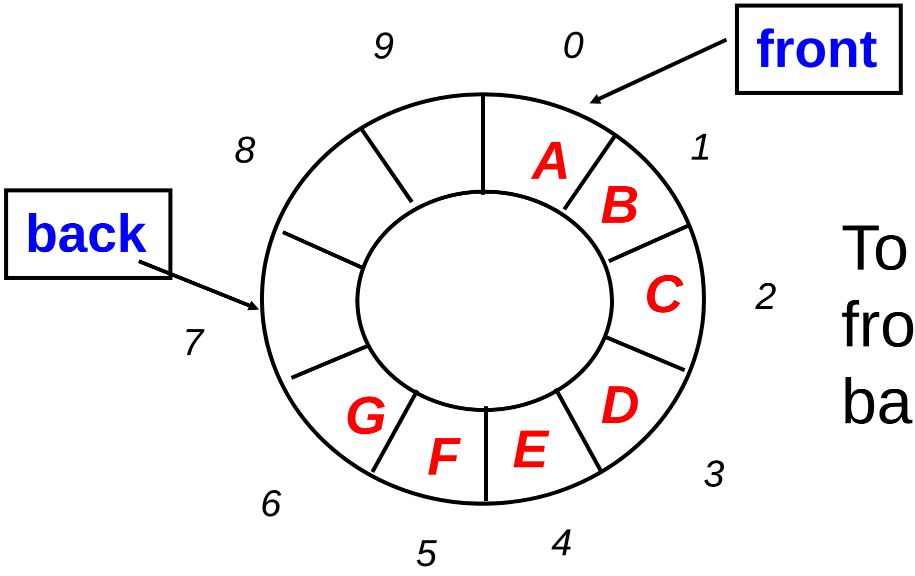


Circular Array

Given a queue



… we can view the array as a circular structure:



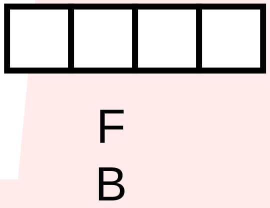
To advance the indexes, use
front = (front+1) % maxsize;
back = (back+1) % maxsize;

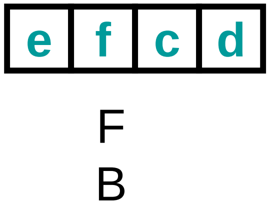
Quiz Time

- **Q:** What does front == back denote?
  - A: Full Queue
  - B: Empty Queue
  - C: Both A and B!
  - D: Neither A nor B!

Ambiguous full/empty state!

Queue
Empty
State





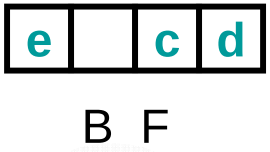
Queue
Full
State

Solution 1 – Maintain queue size or full status

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

Solution 2 – Leave a gap!

Don't need the size field this way



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

Applications

- Many Queue applications:
  - Print queue
  - Simulations
  - Breadth-first traversal of trees
  - Checking palindromes

Return to this in 2nd half of course

Array implementation of Queue

```
class QueueArr implements QueueADT {
  private Object [] arr;
  private int front,back;
  private int maxSize;
  private final int INITSIZE = 1000;

  public QueueArr () {
    arr = new Object[INITSIZE ];
    front = 0;
    back = 0;
    maxSize = INITSIZE ;
  }
  public boolean isEmpty () {
    return (front == back);
  }
  // more on next slide
```

Array implementation (cont)

```
public Object dequeue () throws Underflow {
  Object obj = getFront();
  arr[front] = null;                    // prevent memory leak!
  front = (front + 1) % maxSize;
  return obj;
}
public Object getFront () throws Underflow {
  if (isEmpty()) throw new Underflow (“Invalid operation on empty q”);
  else return arr[front];
}
public void enqueue (Object o) {
  if (((back+1)%maxSize)==front) enlargeArr();
  arr[back] = o;
  back = (back + 1) % maxSize;
}
// more on next slide
```

Array implementation (cont)

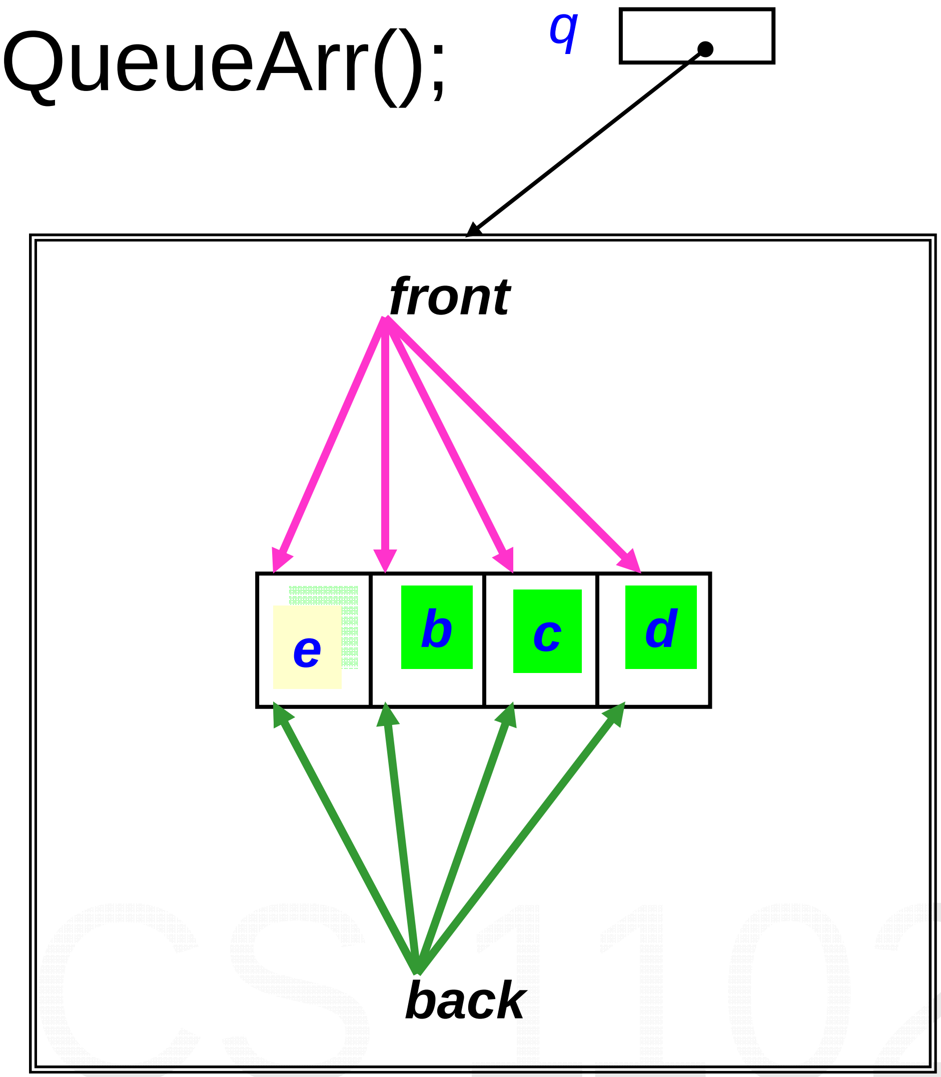
```
private void enlargeArr () {
  int newSize = maxSize * 2;
  Object [] x = new Object[newSize];
  for (int j=0; j < maxSize; j++) {
    x[j] = arr[ (front+j) % maxSize ];
  }
  front = 0; back = maxSize-1;
  maxSize = newSize;
  arr = x;
} } // end class QueueArr
```

Q: How did we solve the F==B problem in this code? (slide 41)

A. Keeping a size variable
B. Letting the maximum size be (capacity – 1)
C. It isn’t solved ☹

Sample run

- ➡ Queue q = new QueueArr();
- ➡ q.enqueue(“a”);
- ➡ q.enqueue(“b”);
- ➡ q.enqueue(“c”);
- ➡ q.dequeue();
- ➡ q.dequeue();
- ➡ q.enqueue(“d”);
- ➡ q.enqueue(“e”);
- ➡ q.dequeue();



Using Stacks with Queues

```
public static boolean MyStackQueueDemo (String v) throws Exception {
  Stack s = new Stack ();
  Queue q = new Queue ();
  int len = v.length ();

  // push string into stack and queue
  for (int j=0; j < len; j++) {
    Character c = new Character (v.charAt (j));
    s.push (c);
    q.enqueue (c);
  }
  // pop, dequeue, and compare
  while (!s.isEmpty())  {
    Character vs = (Character) s.pop();
    Character vq = (Character) q.dequeue();
    if (!vs.equals(vq)) return false;
  }
  return true;
}
```

Recognizing Palindromes

- A string which reads the same either left to right, or right to left is known as a palindrome
  - Palindromes : “r a d a r” and “d e e d”
  - Counterexample : “d a t a”

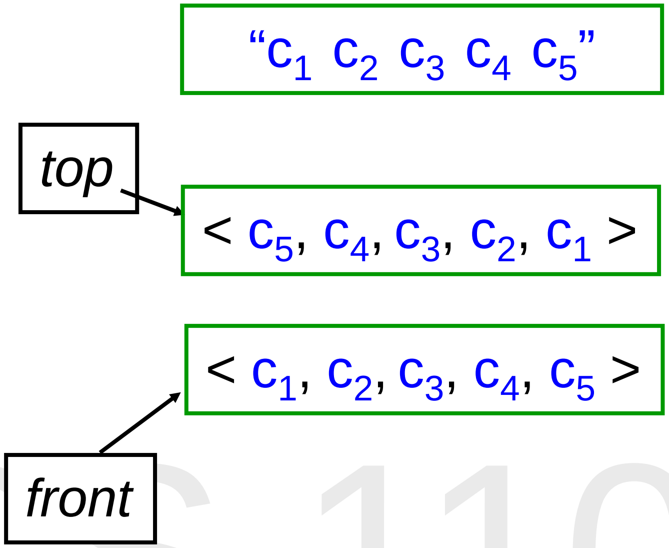
Procedure

Given a string, use:

a Stack to *reverse* its order

a Queue to *preserve* its order

Check if the sequences are the same



“c_1 c_2 c_3 c_4 c_5”

top

< c_5, c_4, c_3, c_2, c_1 >

< c_1, c_2, c_3, c_4, c_5 >

front

java.util interface Queue<E>

boolean offer(E o)

Inserts the specified element into this queue, if possible.

enqueue

E peek()

Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.

getFront

E poll()

Retrieves and removes the head of this queue, or null if this queue is empty.

dequeue

java.util class LinkedList<E>

int size()

Returns the number of elements in this list.

- In addition to those defined in Interface Queue, the above method in the class LinkedList can be used to implement isEmpty()
- Use LinkedList to implement the class Queue!

Summary

- The Stack and Queue ADTs
- LIFO vs. FIFO – a simple difference that leads to very different applications
- Implementations as Linked Lists and Arrays
- Applications
  - Stacks: Matching
  - Queues: Simulations