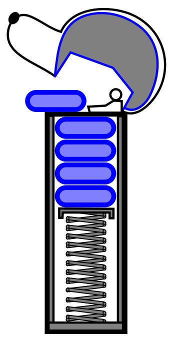
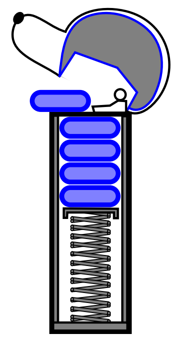
STACKS, QUEUES, AND LINKED LISTS

- Stacks
- Queues
- Linked Lists
- Double-Ended Queues
- Case Study: A Stock Analysis Applet



Stacks

- A stack is a container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as “pushing” onto the stack. “Popping” off the stack is synonymous with removing an item.
- A PEZ® dispenser as an analogy:



The Stack Abstract Data Type

- A stack is an abstract data type (ADT) that supports two main methods:
  - push(*o*): Inserts object *o* onto top of stack
    Input: Object; *Output*: none
  - pop(): Removes the top object of stack and returns it; if stack is empty an error occurs
    Input: none; *Output*: Object
- The following support methods should also be defined:
  - size(): Returns the number of objects in stack
    Input: none; *Output*: integer
  - isEmpty(): Return a boolean indicating if stack is empty.
    Input: none; *Output*: boolean
  - top(): return the top object of the stack, without removing it; if the stack is empty an error occurs.
    Input: none; *Output*: Object

A Stack Interface in Java

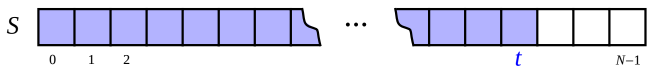
- While, the stack data structure is a “built-in” class of Java’s java.util package, it is possible, and sometimes preferable to define your own specific one, like this:

```
public interface Stack {
    // accessor methods
  public int size(); // return the number of
                     // elements in the stack
  public boolean isEmpty(); // see if the stack
                            // is empty
  public Object top() // return the top element
    throws StackEmptyException; // if called on
                                // an empty stack
                                // update methods

  public void push (Object element); // push an
                     // element onto the stack
  public Object pop() // return and remove the
                     // top element of the stack
    throws StackEmptyException; // if called on
                                // an empty stack
}
```

An Array-Based Stack

- Create a stack using an array by specifying a maximum size N for our stack, e.g. N = 1,000.
- The stack consists of an N-element array S and an integer variable t, the index of the top element in array S.



- Array indices start at 0, so we initialize t to -1
- Pseudo-code

Algorithm size():
  return t +1

Algorithm isEmpty():
  return (t<0)

Algorithm top():
  if isEmpty() **then**
    throw a StackEmptyException
  return $S[t]$

...

An Array-Based Stack (contd.)

- Pseudo-Code (contd.)

Algorithm push(o):
  if size() = N **then**
    throw a StackFullException
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$

Algorithm pop():
  if isEmpty() **then**
    throw a StackEmptyException
  $e \leftarrow S[t]$
  $S[t] \leftarrow$ **null**
  $t \leftarrow t$-1
  return e

- Each of the above method runs in constant time (O(1))
- The array implementation is simple and efficient.
- There is an upper bound, N, on the size of the stack. The arbitrary value N may be too small for a given application, or a waste of memory.

Array-Based Stack: a Java Implementation

```
public class ArrayStack implements Stack {
  // Implementation of the Stack interface
  // using an array.

  public static final int CAPACITY = 1000;  // default
                    // capacity of the stack
  private int capacity; // maximum capacity of the
                    // stack.
  private Object S[]; // S holds the elements of
                    // the stack
  private int top = -1; // the top element of the
                    // stack.

  public ArrayStack() { // Initialize the stack
                      // with default capacity
    this(CAPACITY);
  }
  public ArrayStack(int cap) { // Initialize the
                // stack with given capacity
    capacity = cap;
    S = new Object[capacity];
  }
```


Array-Based Stack in Java (contd.)

```
public int size() { //Return the current stack
                  // size
  return (top + 1);
}
public boolean isEmpty() {  // Return true iff
                      // the stack is empty
  return (top < 0);
}
public void push(Object obj) {  // Push a new
                      // object on the stack
  if (size() == capacity)
    throw new StackFullException("Stack overflow.");
  S[++top] = obj;
}
public Object top() // Return the top stack
                  // element
 throws StackEmptyException {
if (isEmpty())
  throw new StackEmptyException("Stack is empty.");
return S[top];
}
```

Array-Based Stack in Java (contd.)

```
public Object pop() // Pop off the stack element
    throws StackEmptyException {
  Object elem;
  if (isEmpty())
     throw new StackEmptyException("Stack is Empty.");
  elem = S[top];
  S[top--] = null; // Dereference S[top] and
                  // decrement top
  return elem;
  }
}
```


Casting With a Generic Stack

- Have an ArrayStack that can store only Integer objects or Student objects.
- In order to do so using a generic stack, the return objects must be cast to the correct data type.
- A Java code example:

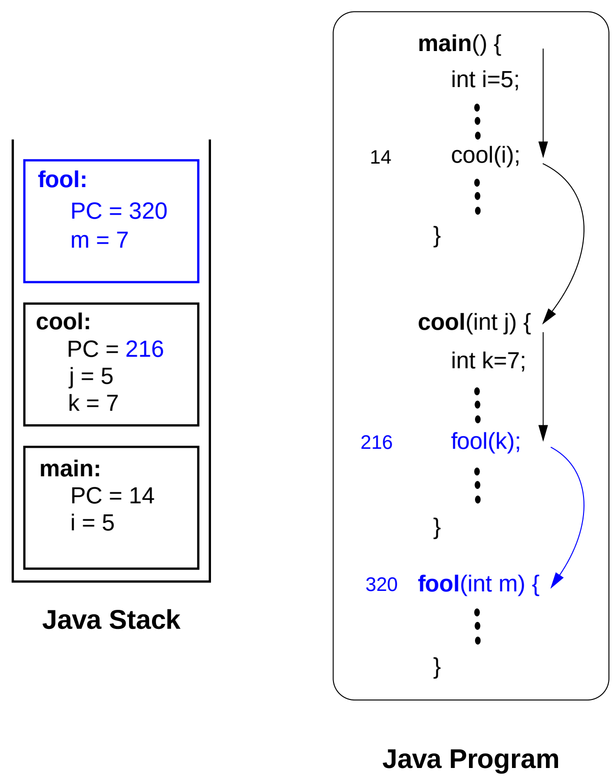
```
public static Integer[] reverse(Integer[] a) {
  ArrayStack S = new ArrayStack(a.length);
  Integer[] b = new Integer[a.length];
  for (int i = 0; i < a.length; i++)
    S.push(a[i]);
  for (int i = 0; i < a.length; i++)
    b[i] = (Integer)(S.pop());
return b;
```


Stacks in the Java Virtual Machine

- Each process running in a Java program has its own Java Method Stack.
- Each time a method is called, it is pushed onto the stack.
- The choice of a stack for this operation allows Java to do several useful things:
  - Perform recursive method calls
  - Print stack traces to locate an error
- Java also includes an operand stack which is used to evaluate arithmetic instructions, i.e.

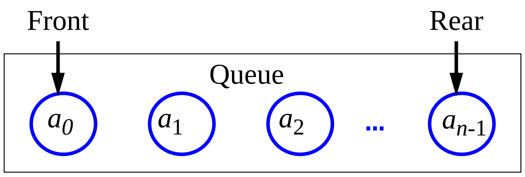
```
Integer add(a, b):
  OperandStack Op
  Op.push(a)
  Op.push(b)
  temp1 ← Op.pop()
  temp2 ← Op.pop()
  Op.push(temp1 + temp2)
  return Op.pop()
```


Java Method Stack



Queues

- A queue differs from a stack in that its insertion and removal routines follows the first-in-first-out (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the *rear* (enqueued) and removed from the *front* (dequeued)

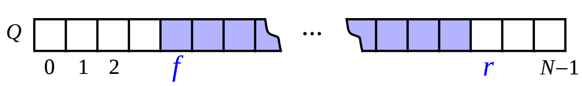


The Queue Abstract Data Type

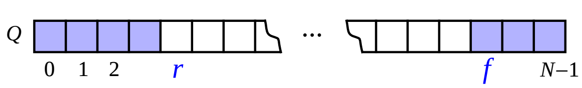
- The queue supports two fundamental methods:
  - enqueue(o): Insert object o at the rear of the queue
    Input: Object; *Output*: none
  - dequeue(): Remove the object from the front of the queue and return it; an error occurs if the queue is empty
    Input: none; *Output*: Object
- These support methods should also be defined:
  - size(): Return the number of objects in the queue
    Input: none; *Output*: integer
  - isEmpty(): Return a boolean value that indicates whether the queue is empty
    Input: none; *Output*: boolean
  - front(): Return, but do not remove, the front object in the queue; an error occurs if the queue is empty
    Input: none; *Output*: Object

An Array-Based Queue

- Create a queue using an array in a circular fashion
- A maximum size N is specified, e.g. N = 1,000.
- The queue consists of an N-element array Q and two integer variables:
  - f, index of the front element
  - r, index of the element after the rear one
- "normal configuration"



- "wrapped around" configuration



- what does f=r mean?

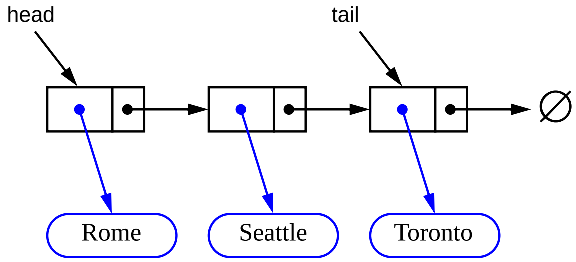
An Array-Based Queue (contd.)

- Pseudo-Code (contd.)

Algorithm size():
  return (N - f + r) mod N
Algorithm isEmpty():
  return (f = r)
Algorithm front():
  if isEmpty() **then**
    throw a QueueEmptyException
  return $Q[f]$
Algorithm dequeue():
  if isEmpty() **then**
    throw a QueueEmptyException
  temp ← $Q[f]$
  $Q[f]$ ← **null**
  f ← (f + 1) mod N
  return *temp*
Algorithm enqueue(o):
  if size = N - 1 **then**
    throw a QueueFullException
  $Q[r]$ ← o
  r ← (r +1) mod N

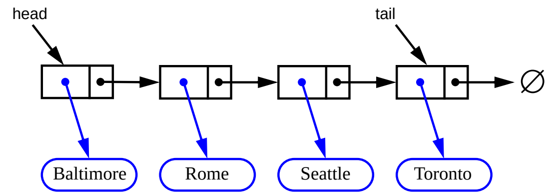
Implementing a Queue with a Singly Linked List

- nodes connected in a chain by links

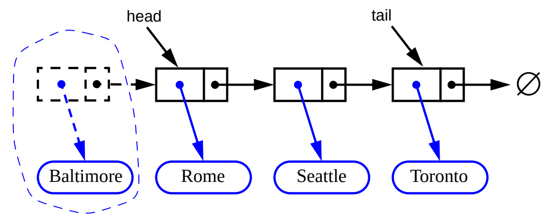


- the head of the list is the front of the queue, the tail of the list is the rear of the queue
- why not the opposite?

Removing at the Head



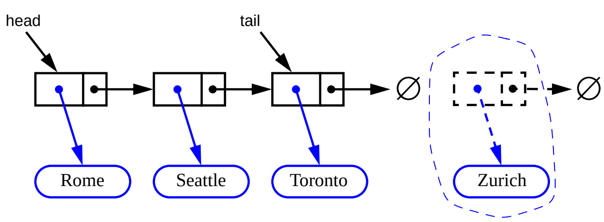
- advance head reference



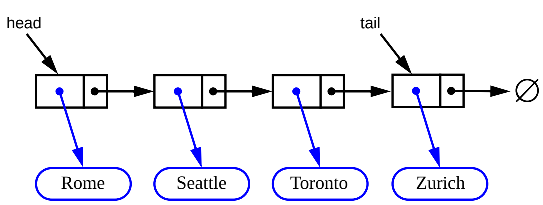
- inserting at the head is just as easy

Inserting at the Tail

- create a new node



- chain it and move the tail reference



- how about removing at the tail?

Double-Ended Queues

- A double-ended queue, or deque, supports insertion and deletion from the front and back.
- The Deque Abstract Data Type
  - insertFirst(*e*): Insert e at the beginning of deque. Input: Object; Output: none
  - insertLast(*e*): Insert e at end of deque Input: Object; Output: none
  - removeFirst(): Removes and returns first element Input: none; Output: Object
  - removeLast(): Removes and returns last element Input: none; Output: Object
- Additionally supported methods include:
  - first()
  - last()
  - size()
  - isEmpty()

Implementing Stacks and Queues with Deques

- Stacks with Deques:

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

- Queues with Deques:

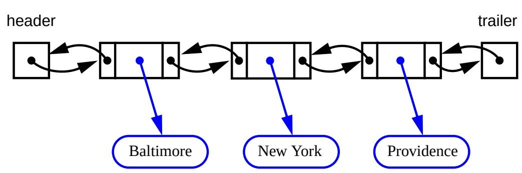
Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()

The Adaptor Pattern

- Using a deque to implement a stack or queue is an example of the adaptor pattern. Adaptor patterns implement a class by using methods of another class
- In general, adaptor classes specialize general classes
- Two such applications:
  - Specialize a general class by changing some methods.
    Ex: implementing a stack with a deque.
  - Specialize the types of objects used by a general class.
    Ex: Defining an IntegerArrayStack class that adapts ArrayStack to only store integers.

Implementing Deques with Doubly Linked Lists

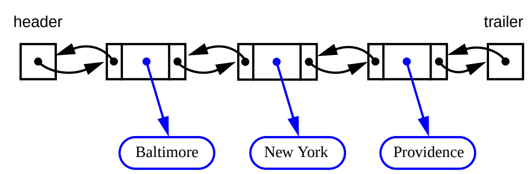
- Deletions at the tail of a singly linked list cannot be done in constant time.
- To implement a deque, we use a doubly linked list. with special header and trailer nodes.



- A node of a doubly linked list has a next and a prev link. It supports the following methods:
  - setElement(Object e)
  - setNext(Object newNext)
  - setPrev(Object newPrev)
  - getElement()
  - getNext()
  - getPrev()
- By using a doubly linked list to, all the methods of a deque have constant (that is, O(1)) running time.

Implementing Deques with Doubly Linked Lists (cont.)

- When implementing a doubly linked lists, we add two special nodes to the ends of the lists: the header and trailer nodes.
  - The header node goes before the first list element. It has a valid next link but a null prev link.
  - The trailer node goes after the last element. It has a valid prev reference but a null next reference.
- The header and trailer nodes are sentinel or “dummy” nodes because they do not store elements.
- Here’s a diagram of our doubly linked list:



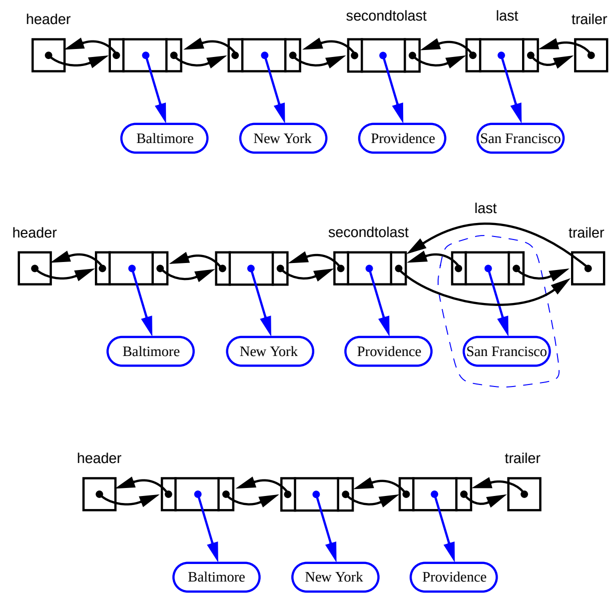
Implementing Deques with Doubly Linked Lists (cont.)

- Let's look at some code for removeLast()

```
public class MyDeque implements Deque{
   DLNode header_, trailer_;
   int size_;
   ...
   public Object removeLast() throws
      DequeEmptyException{
      if(isEmpty())
         throw new DequeEmptyException("Ilegal
            removal request.");
      DLNode last = trailer_.getPrev();
      Object o = last.getElement();
      DLNode secondtolast = last.getPrev();
      trailer_.setPrev(secondtolast);
      secondtolast.setnext(trailer_);
      size_ --;
      return o;
   }
   ...
}
```

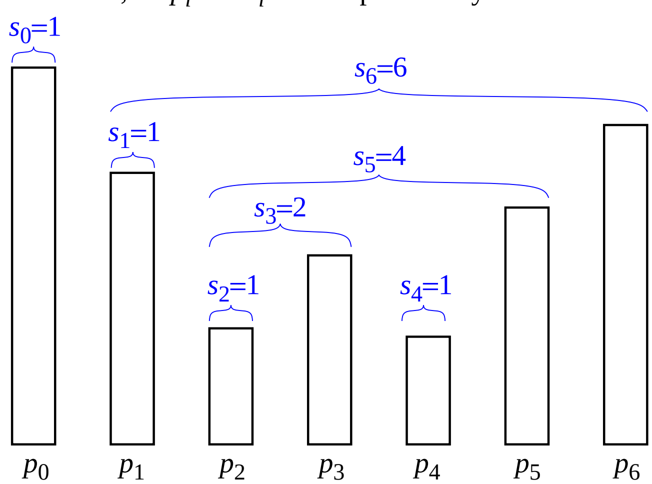

Implementing Deques with Doubly Linked Lists (cont.)

- Here's a visualization of the code for removeLast().



A Stock Analysis Applet

- The span of a stock's price on a certain day, *d*, is the maximum number of consecutive days (up to the current day) the price of the stock has been less than or equal to its price on *d*.
- Below, let p_i and s_i be the span on day *i*



A Case Study: A Stock Analysis Applet (cont.)

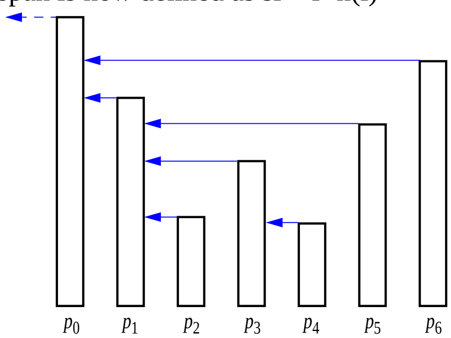
- Quadratic-Time Algorithm: We can find a straightforward way to compute the span of a stock on a given day for *n* days:

Algorithm computeSpans1(*P*):
  Input: An *n*-element array *P* of numbers
  Output: An *n*-element array *S* of numbers such that
      S[*i*] is the span of the stock on day *i*.
  Let *S* be an array of n numbers
  for *i*=0 **to** *n*-1 **do**
    k ←0
    done←**false**
    repeat
      if *P*[*i*-*k*] ≤*P*[*i*] **then**
        k←*k*+1
      else
        done←**true**
    until (*k*=*i*) **or** *done*
    S[*i*]←*k*
  return array *S*

- The running time of this algorithm is (ugh!) O(n^2). Why?

A Case Study: A Stock Analysis Applet (cont.)

- Linear-Time Algorithm: We see that si on day i can be easily computed if we know the closest day preceding i, such that the price is greater than on that day than the price on day i. If such a day exists let's call it h(i).
- The span is now defined as si = i -h(i)



The arrows point to h(i)

A Case Study: A Stock Analysis Applet (cont.)

- The code for our new algorithm:

```
Algorithm computeSpan2(P):
  Input: An n-element array P of numbers
  Output: An n-element array S of numbers such that
          S[i] is the span of the stock on day i.
  Let S be an array of n numbers and D an empty stack
  for i=0 to n-1 do
    done←false
    while not(D.isEmpty() or done) do
      if P[i]≥P[D.top()] then
        D.pop()
      else
        done←true
      if D.isEmpty() then
        h← -1
      else
        h←D.top()
      S[i]←i-h
      D.push(i)
    return array S
```

- Let's analysize computeSpan2's run time...

A Case Study: A Stock Analysis Applet (cont.)

- The total running time of the while loop is

$$O\left(\sum_{i=0}^{n-1}(t_i+1)\right)$$

- However, once an element is popped off the stack, it is never pushed on again. Therefore:

$$\sum_{i=0}^{n-1} t_i \le n$$

- The total time spent in the while loop is O(n).
- The run time of computeSpan2 is the summ of three O(n) terms. Thus the run time of computeSpan2 is O(n).