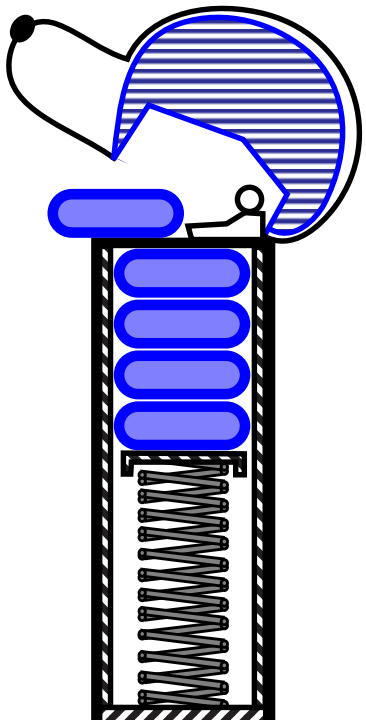
Stacks, Queues, and Linked Lists

- Abstract Data Types (ADTs)
- Stacks
- Example: Stock Analysis
- Queues
- Linked Lists
- Double-Ended Queues

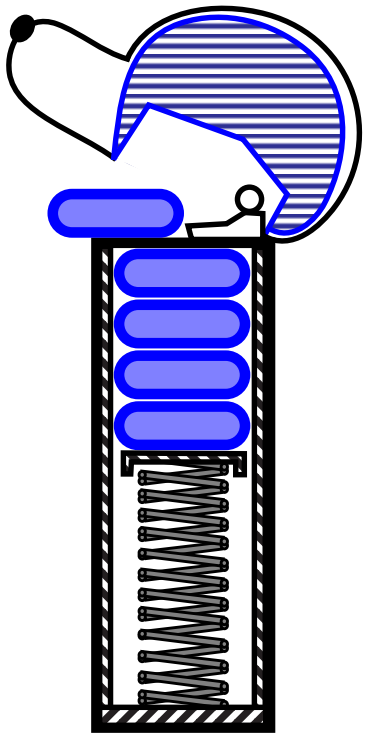


Abstract Data Types (ADTs)

- An **Abstract Data Type** is an abstraction of a data structure: no coding is involved.
- The ADT specifies:
  - what can be stored in the ADT
  - what operations can be done on/by the ADT
- For example, if we are going to model a bag of marbles as an ADT, we could specify that
  - this ADT stores marbles
  - this ADT supports putting in a marble and getting out a marble.
- There are lots of formalized and standard ADTs. A bag of marbles is not one of them.
- In this course we are going to learn a lot of different standard ADTs. (stacks, queues, trees...)

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:

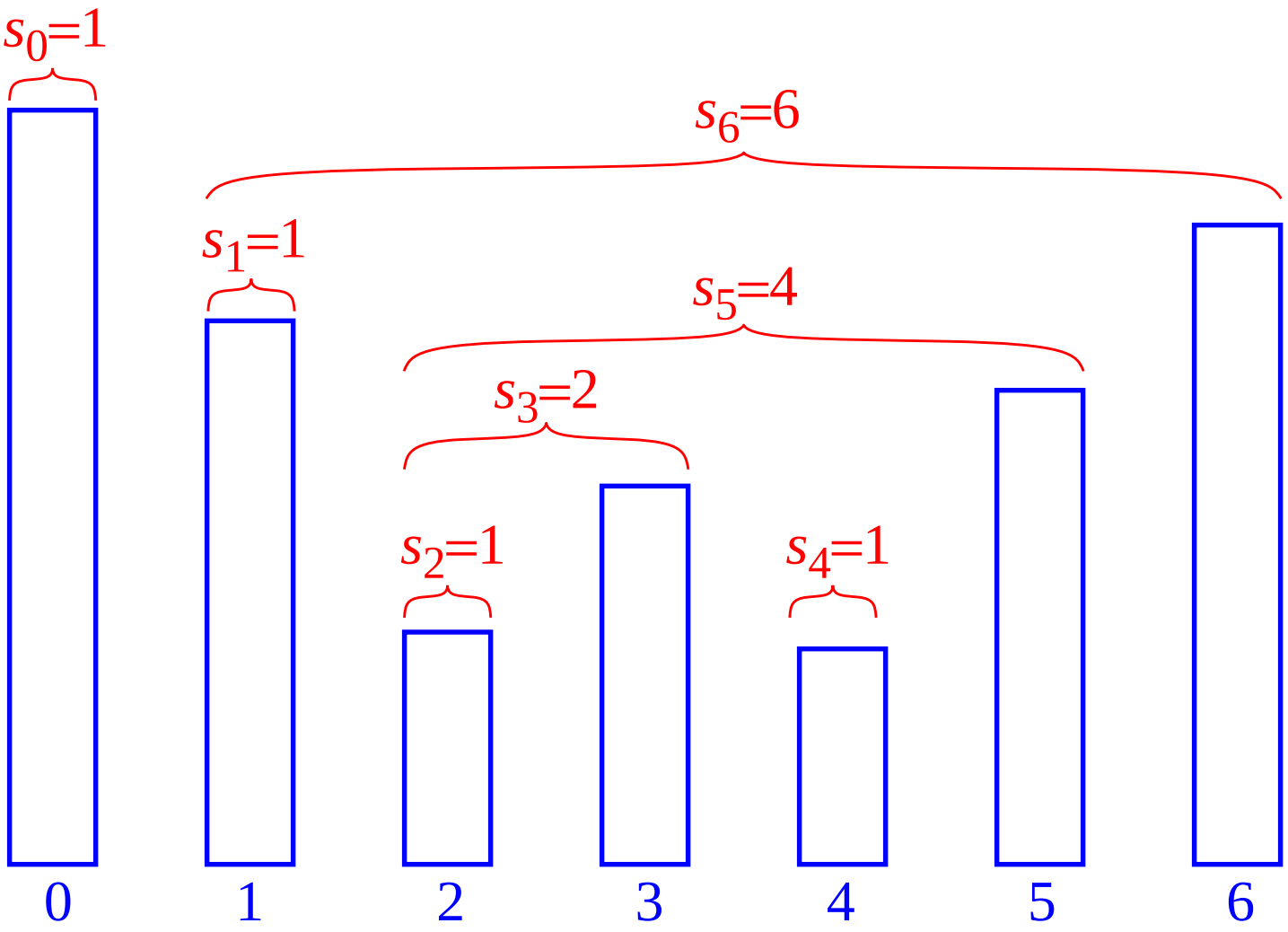


The Stack Abstract Data Type

- A stack is an abstract data type (ADT) that supports two main methods:
  - push(*o*): Inserts object *o* onto top of stack
  - pop(): Removes the top object of stack and returns it; if stack is empty an error occurs
- The following support methods should also be defined:
  - size(): Returns the number of objects in stack
  - isEmpty(): Return a boolean indicating if stack is empty.
  - top(): return the top object of the stack, without removing it; if the stack is empty an error occurs.

Example

- 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*.



An Inefficient Algorithm

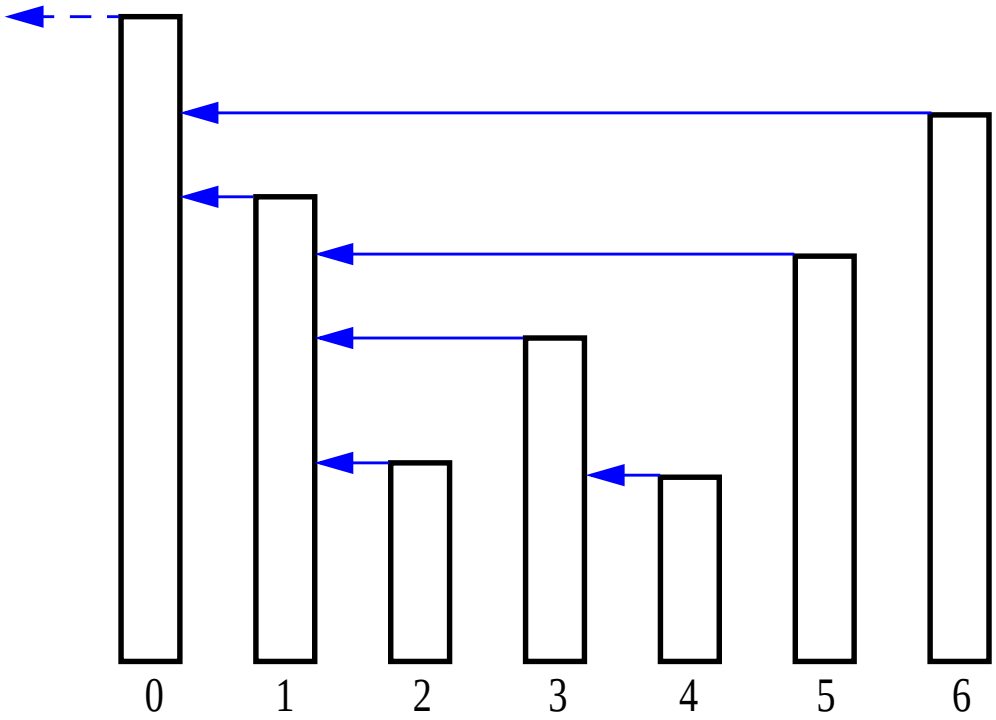
- There is a straightforward way to compute the span of a stock on a given day for n days:

```
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 Stack Can Help

- We see that s_i 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 $s_i = i - h_i$



We use a ***stack*** to keep track of 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...

Java Stuff

- Given the stack ADT, we need to code the ADT in order to use it in the programs.
- You need to understand two program constructs: **interfaces** and **exceptions**.
- An interface is a way to declare what a class is to do. It does not mention how to do it.
- For an interface, you just write down the method names and the parameters. When specifying parameters, what really matters is their **types**.
- Later, when you write a class for that interface, you actually code the content of the methods.
- Separating interface and implementation is a useful programming technique.
- Interface example:

```
public interface radio {
   public void play();
   public void stop();
}
```

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. Note that
       // the type of the parameter is
       // specified as an Object

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

}
```

Exceptions

- **Exceptions** are yet another useful programming construct.
- This is useful for handling errors. When you find an error (or an ***exceptiona*l** case), you just ***throw*** an exception.
- Example

```
public void eatPizza() throws StomachAcheException
{
   ...

   if (ateTooMuch)
      throw new StomachAcheException("Ouch");

   ...
}
```

- As soon as the exception is thrown, the flow of control exits from the current method.
- So when StomachAcheException is thrown, we exit from method eatPizza() and go to where that method was called from.

More Exceptions

- Say the following code fragment called the method eatPizza() in the first place.

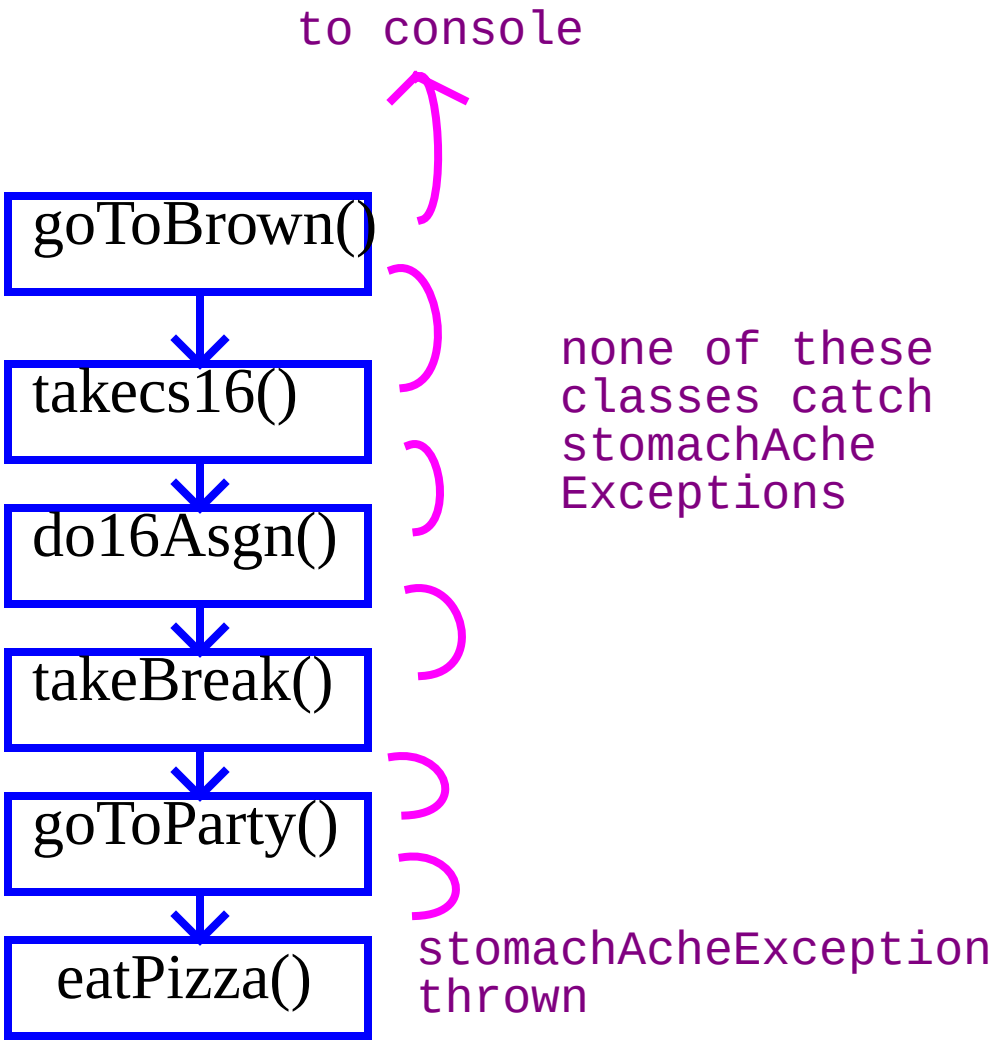
```
private void simulateMeeting()
{
   ...
   try
     {
        aStupidTA.eatPizza();
     }
   catch(StomachAcheException e)
     {
        System.out.println("somebody has a stomach
        ache");
     }
   ...
}
```

Even More Exceptions

- We will get back to aStupidTA.eatPizza(); because, remember, eatPizza() threw an exception.
- The try block and the catch block means that we are listening for exceptions that are specified in the catch parameter.
- Because catch is listening for StomachAcheException, the flow of control will now go to the catch block. And System.out.println will get executed.
- Note that a catch block can contain anything. It does not have to do only System.out.println. You can handle the caught error in any way you like; you can even throw them again.
- Note that if somewhere in your method, you throw an exception, you need to add a throws clause next to your method name.
- What is the point of using exceptions? You can delegate upwards the responsibility of handling an error. Delegating upwards means letting the code who called the current code deal with the problem.

Even More Exceptions

- If you never catch an exception, it will propagate upwards and upwards along the chain of method calls until the user sees it.



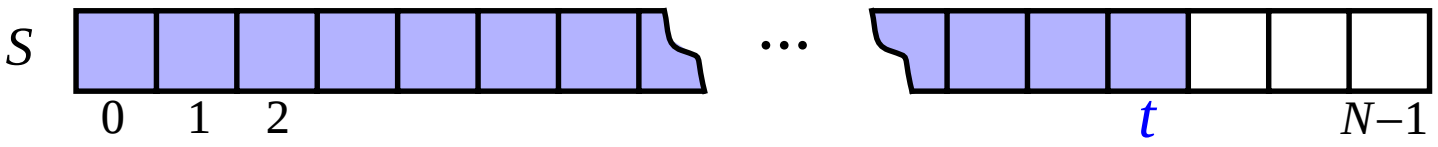
Final Exceptions

- OK, so we threw and caught exceptions. But what exactly are they in Java? Classes.
- Check out the StomachAcheException.

```
public class StomachAcheException extends
   RuntimeException {
   public StomachAcheException(String err)
      {
         super(err);
      }
}
```

An Array-Based Stack

- Create a stack using an array by specifying a maximum size N for our stack, e.g. $N = 1{,}024$.
- 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 ← t + 1
  S[t] ← o

Algorithm pop():
  if isEmpty() then
    throw a StackEmptyException
  e←S[t]
  S[t]←null
  t←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
     this(CAPACITY);// with default 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;
  }
}
```

A Growable Array-Based Stack

- Instead of giving up with a StackFullException, we can replace the array S with a larger one and continue processing push operations.

```
Algorithm push(o):
  if size() = N then
    A ← new array of length f(N)
   for i ← 0 to N – 1
     A[i] ← S[i]
   S ← A
  t ← t + 1
  S[t] ← o
```

- ***How large should the new array be?***
  - ***tight strategy*** (add a constant): $f(N) = N + c$
  - ***growth strategy*** (double up): $f(N) = 2N$

- To compare the two strategies, we use the following cost model

regular push operation: add one element	1
special push operation: create an array of size $f(N)$, copy N elements, and add one element	$f(N) + N + 1$

Tight Strategy (c=4)

- start with an array of size 0
- the cost of a special push is $2N + 5$

push	phase	n	N	cost
1	1	0	0	5
2	1	1	4	1
3	1	2	4	1
4	1	3	4	1
5	2	4	4	13
6	2	5	8	1
7	2	6	8	1
8	2	7	8	1
9	3	8	8	21
10	3	9	12	1
11	3	10	12	1
12	3	11	12	1
13	4	12	12	29

Performance of the Tight Strategy

- We consider k phases, where $k = n/c$
- Each phase corresponds to a new array size
- The cost of phase i is $2ci$
- The total cost of n push operations is the total cost of k phases, with $k = n/c$:

  $2c\ (1 + 2 + 3 + ... + k)$,

  which is $O(k^2)$ and $\mathbf{O(n^2)}$.

Growth Strategy

- start with an array of size 0, then 1, 2, 4, 8, ...
- the cost of a special push is $3N + 1$ for $N > 0$

push	phase	n	N	cost
1	0	0	0	2
2	1	1	1	4
3	2	2	2	7
4	2	3	4	1
5	3	4	4	13
6	3	5	8	1
7	3	6	8	1
8	3	7	8	1
9	4	8	8	25
10	4	9	16	1
11	4	10	16	1
12	4	11	16	1
...	...	...	...	...
16	4	15	16	1
17	5	16	16	49

Performace of the Growth Strategy

- We consider k phases, where $k = \log n$
- Each phase corresponds to a new array size
- The cost of phase i is 2^{i+1}
- The total cost of n push operations is the total cost of k phases, with $k = \log n$

$$2 + 4 + 8 + \ldots + 2^{\log n + 1} =$$

$$2n + n + n/2 + n/4 + \ldots + 8 + 4 + 2 = 4n - 1$$

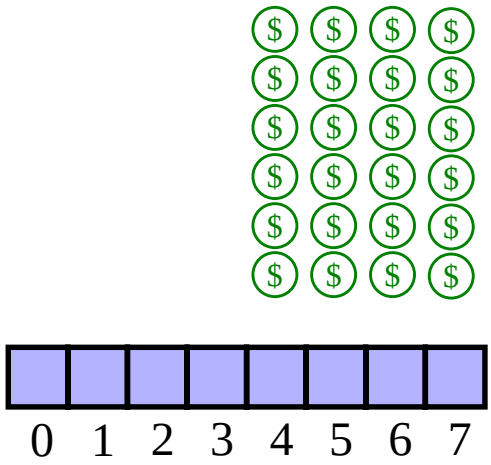
- The growth strategy wins!

Amortized Analysis

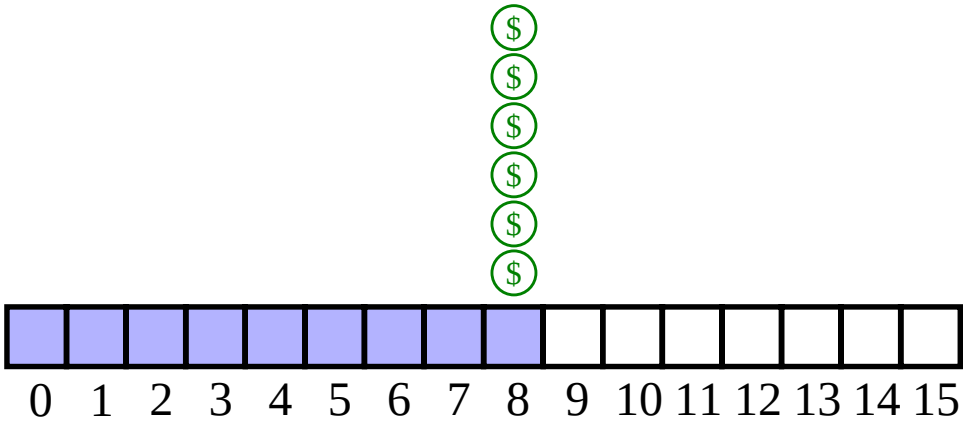
- The **amortized running time** of an operation within a series of operations is the worst-case running time of the entire series of operations divided by the number of operations
- The **accounting method** determines the amortized running time with a system of credits and debits
- We view the computer as a coin-operated appliance that requires one cyber-dollar for a constant amount of computing time.
- We set up a scheme for charging operations. This is known as an ***amortization scheme***.
- We may overcharge some operations and undercharge other operations. For example, we may ***charge each operation the same amount***.
- The scheme must give us always enough money to pay for the actual cost of the operation.
- The total cost of the series of operations is no more than the total amount charged.
- (amortized time) ≤ (total $ charged) / (# operations)

Amortization Scheme for the Growth Strategy

- At the end of a phase we must have saved enough to pay for the special push of the next phase.
- At the end of phase 3 we want to have saved $24.



- The amount saved pays for growing the array.



- We charge **$7** for a push. The $6 saved for a regular push are "stored" in the second half of the array.

Amortized Analysis of the Growth Strategy

- We charge **$5** (introductory special offer) for the first push and **$7** for the remaining ones

push	*n*	*N*	***balance***	***charge***	cost
1	0	0	$0	$5	$2
2	1	1	$3	$7	$4
3	2	2	$6	$7	$7
4	3	4	$6	$7	$1
5	4	4	$12	$7	$13
6	5	8	$6	$7	$1
7	6	8	$12	$7	$1
8	7	8	$18	$7	$1
9	8	8	$24	$7	$25
10	9	16	$6	$7	$1
11	10	16	$12	$7	$1
12	11	16	$18	$7	$1
...	...	...	...	...	...
16	15	16	$42	$7	$1
17	16	16	$48	$7	$49

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:

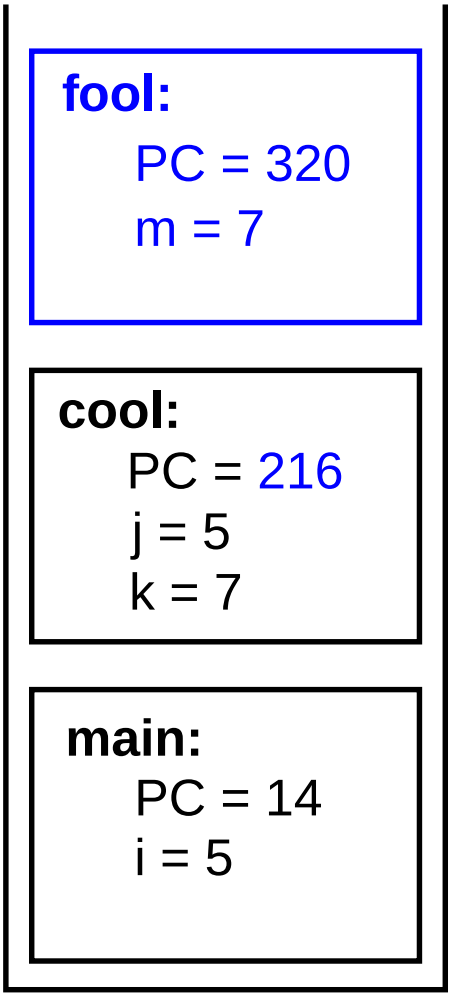
```
public static Integer[] reverse(Integer[] a) {
   ArrayStack S = new ArrayStack(a.length);
   Integer[] b = new Integer[a.length];
   for (int i = 0; i < a.length; i++)
      S.push(a[i]);
   for (int i = 0; i < a.length; i++)
      b[i] = (Integer)(S.pop()); // the popping
        // operation gave us an Object, and we
        // casted it to an Integer before
        // assigning it to b[i].
   return b;
}
```

Stacks in the Java Virtual Machine

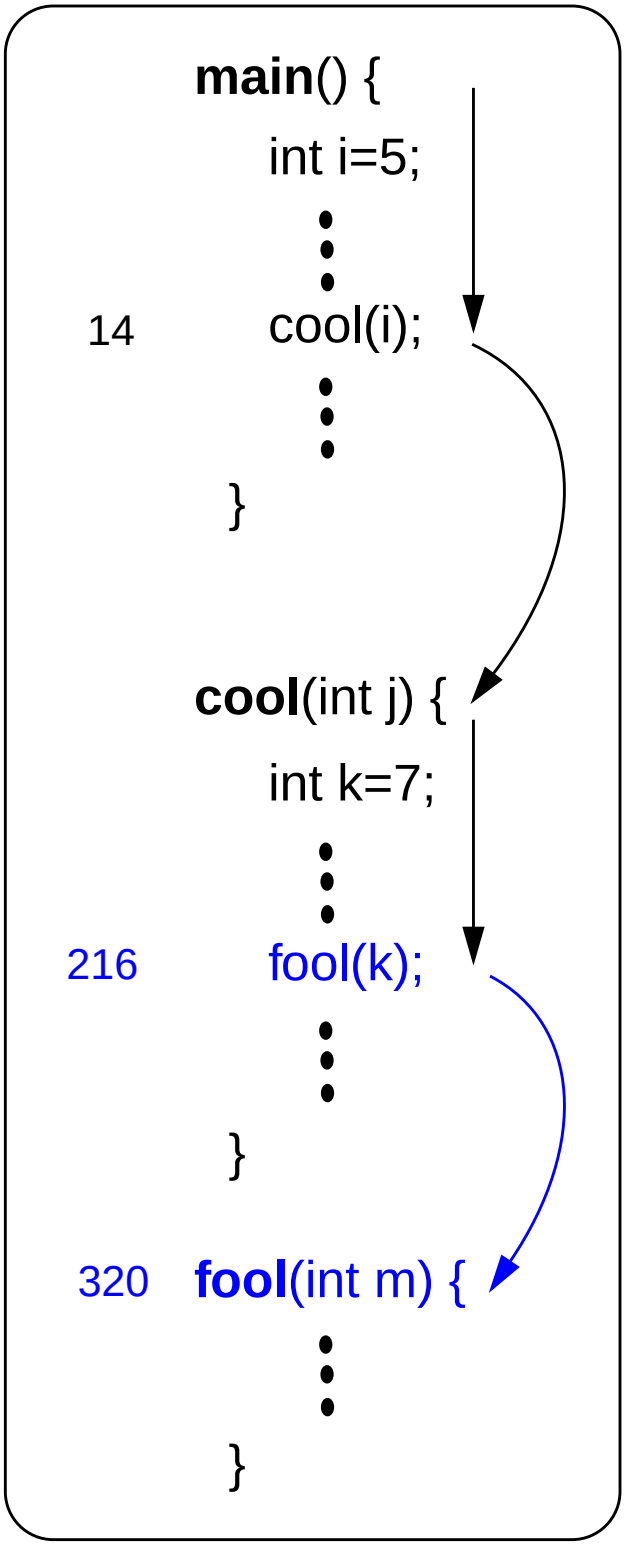
- Each process running in a Java program has its own Java Method Stack.
- Each time a method is called, it is pushed onto the stack.
- The choice of a stack for this operation allows Java to do several useful things:
  - Perform recursive method calls
  - Print stack traces to locate an error
- Java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



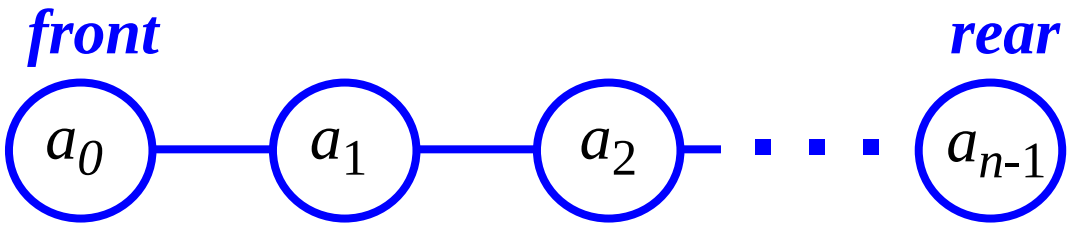
Java Stack



Java Program

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)

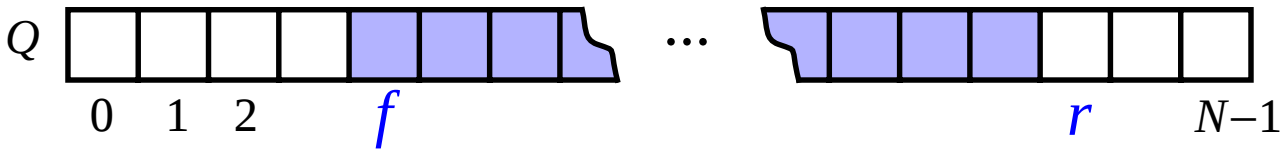


The Queue Abstract Data Type

- The queue supports two fundamental methods:
  - enqueue(*o*): Insert object *o* at the rear of the queue
  - dequeue(): Remove the object from the front of the queue and return it; an error occurs if the queue is empty
- These support methods should also be defined:
  - size(): Return the number of objects in the queue
  - isEmpty(): Return a boolean value that indicates whether the queue is empty
  - front(): Return, but do not remove, the front object in the queue; an error occurs if the queue is empty

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?

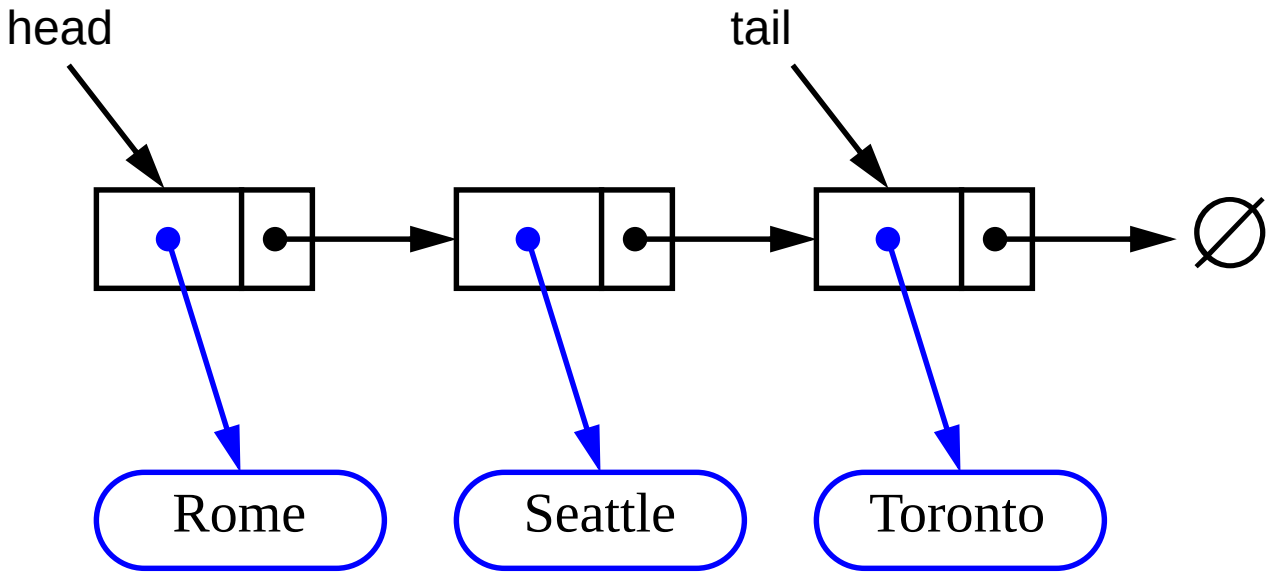
An Array-Based Queue (contd.)

- 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
```

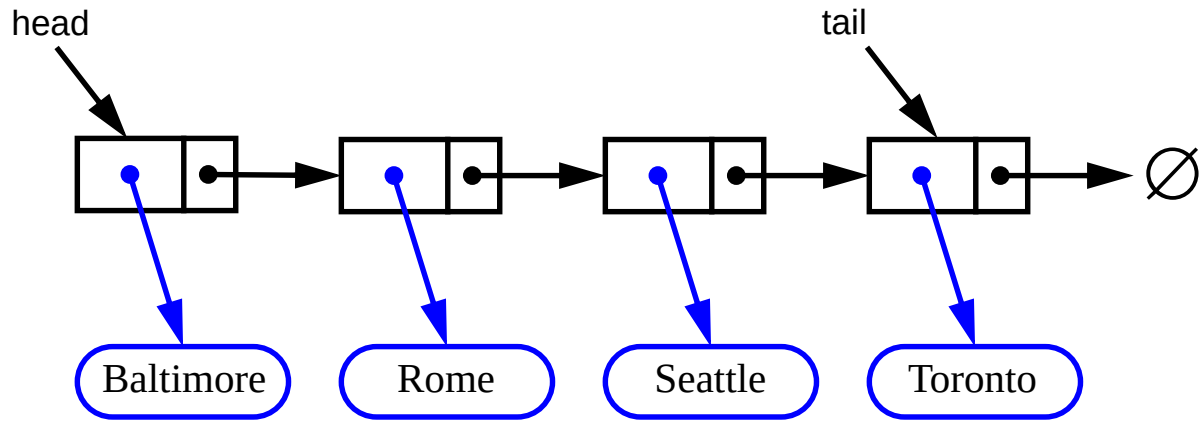
Implementing a Queue with a Singly Linked List

- nodes connected in a chain by links

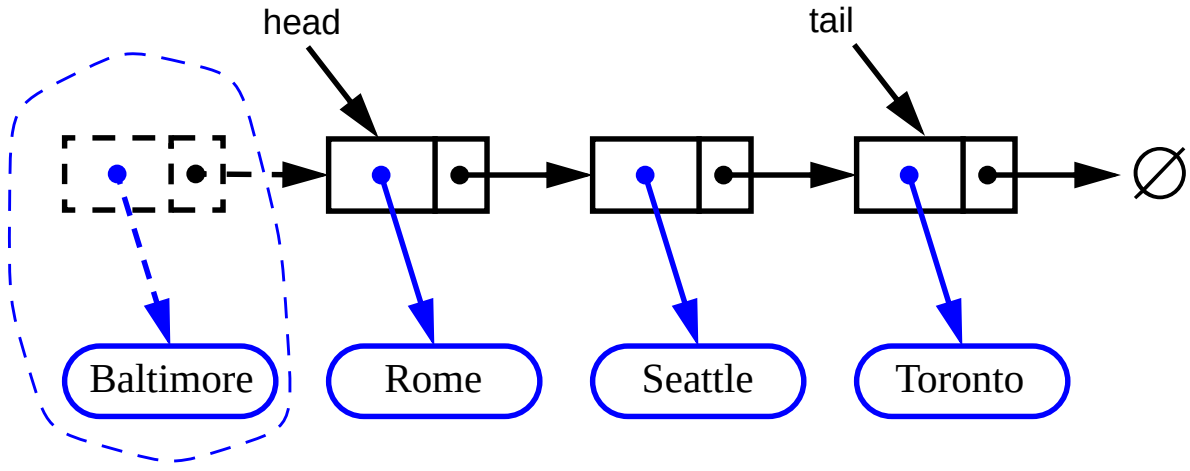


- the head of the list is the front of the queue, the tail of the list is the rear of the queue
- why not the opposite?

Removing at the Head



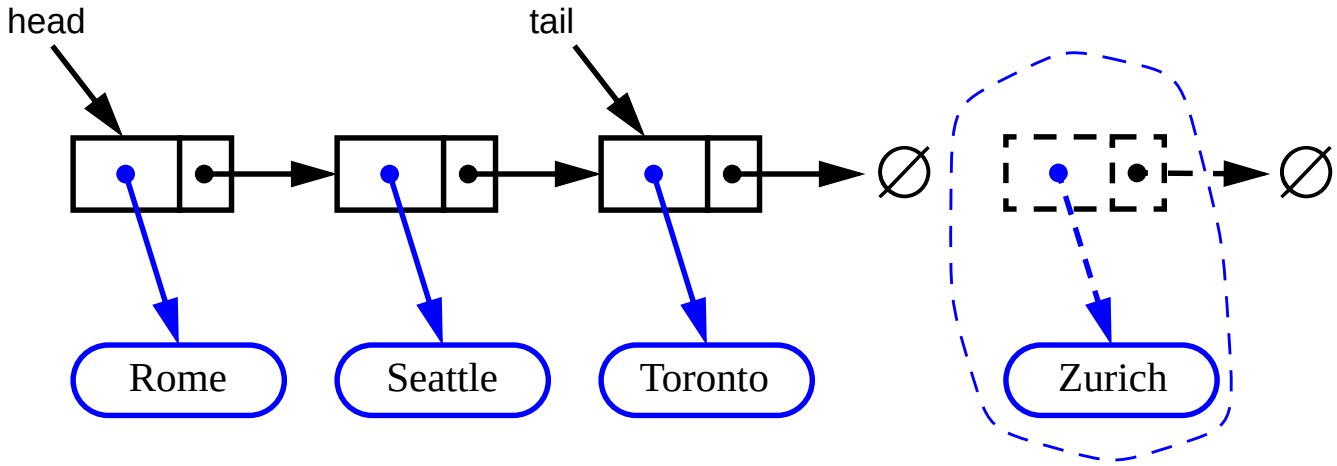
- advance head reference



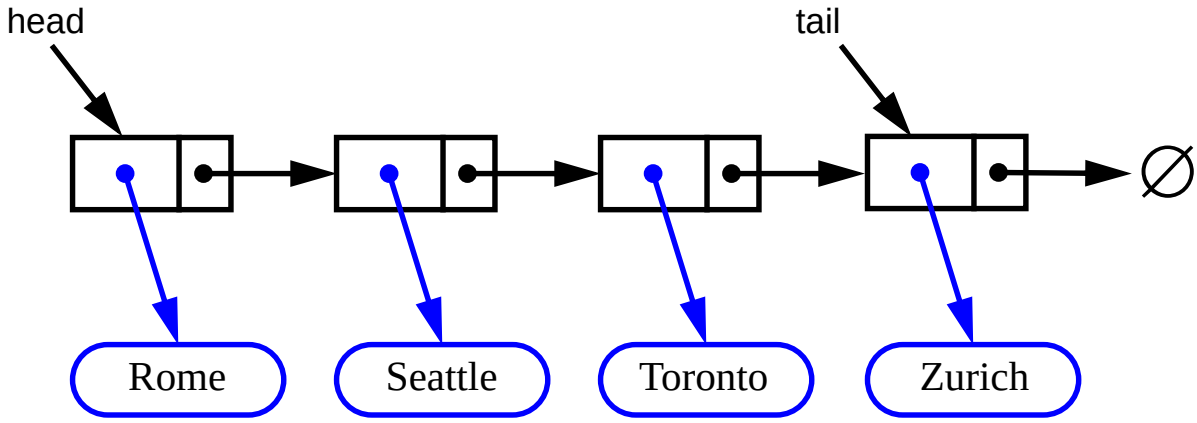
- inserting at the head is just as easy

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 deginning of deque.
  - insertLast(*e*): Insert e at end of deque
  - removeFirst(): Removes and returns first element
  - removeLast(): Removes and returns last element
- 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

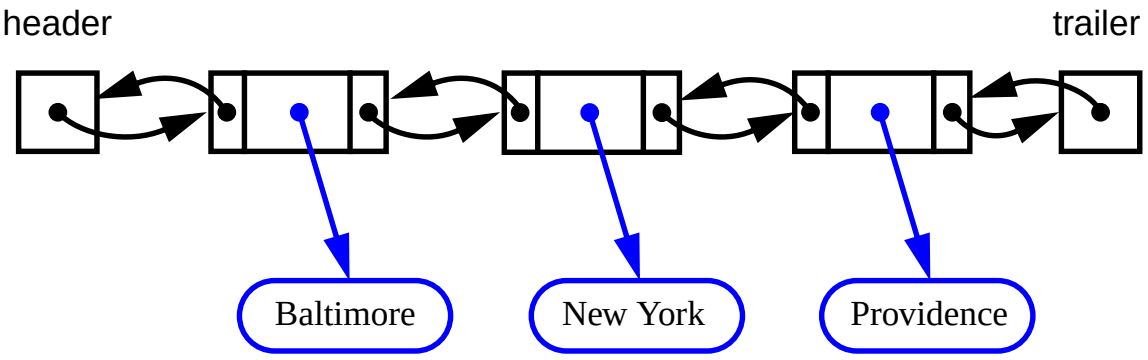
- Using a deque to implement a stack or queue is an example of the adaptor pattern. Adaptor patterns implement a class by using methods of another class
- In general, adaptor classes specialize general classes
- Two such applications:
  - Specialize a general class by changing some methods.

    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

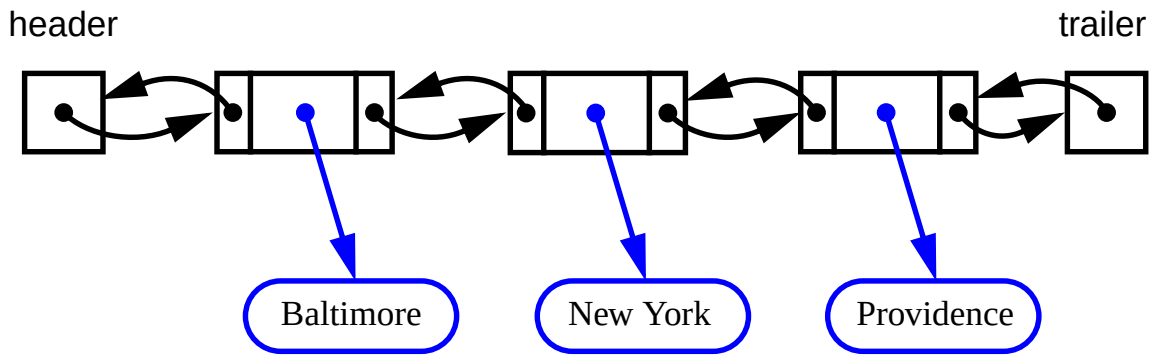
- Deletions at the tail of a singly linked list cannot be done in constant time.
- To implement a deque, we use a doubly linked list. with special header and trailer nodes.



- 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.)

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