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
Application: Time Series

- The span $s_i$ of a stock’s price on a certain day $i$ 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 day $i$. 

\[ s_0 = 1 \quad s_1 = 1 \quad s_2 = 1 \quad s_3 = 2 \quad s_4 = 1 \quad s_5 = 4 \quad s_6 = 6 \]
An Inefficient Algorithm

• There is a straightforward way to compute the span of a stock on each of \( n \) days:

Algorithm computeSpans1\((P)\):

Input: an \( n \)-element array \( P \) of numbers such that \( P[i] \) is the price of the stock on day \( i \)

Output: an \( n \)-element array \( S \) of numbers such that \( S[i] \) is the span of the stock on day \( i \)

for \( i \leftarrow 0 \) to \( n - 1 \) do

\( k \leftarrow 0 \)

\( done \leftarrow \text{false} \)

repeat

if \( P[i - k] \leq P[i] \) then

\( k \leftarrow k + 1 \)

else

\( done \leftarrow \text{true} \)

until \( (k = i) \) or \( done \)

\( S[i] \leftarrow k \)

return \( S \)

• 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 $h(i)$, otherwise, we conventionally define $h(i) = -1$

• The span is now computed as $s_i = i - h(i)$

We use a stack to keep track of $h(i)$
An Efficient Algorithm

- The code for our new algorithm:

```
Algorithm computeSpan2(P):
    Input: An n-element array P of numbers representing stock prices
    Output: An n-element array S of numbers such that S[i] is the span of the stock on day i

    Let D be 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 S
```

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

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 \( h(i) \), otherwise, we conventionally define \( h(i) = −1 \). The span is now computed as \( s_i = i − h(i) \). We use a stack to keep track of \( h(i) \).
Queue ADT
Queue

- 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).
Queue

• The queue has two fundamental methods:
  - enqueue(o): Inserts object o at rear of the queue
  - dequeue(): Removes object from front of queue and returns it; an error occurs if queue is empty.

• These support methods should also be defined:
  - size(): Returns number of objects in the queue
  - isEmpty(): Returns a boolean value that indicates whether the queue is empty
  - front(): Returns, but not remove, the front object in the queue; an error occurs if queue is empty.
# Queue

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(a)</td>
<td>a</td>
</tr>
<tr>
<td>add(b)</td>
<td>ab</td>
</tr>
<tr>
<td>remove()</td>
<td>b</td>
</tr>
<tr>
<td>add(c)</td>
<td>bc</td>
</tr>
<tr>
<td>add(d)</td>
<td>bcd</td>
</tr>
<tr>
<td>add(e)</td>
<td>bcde</td>
</tr>
<tr>
<td>remove()</td>
<td>cde</td>
</tr>
<tr>
<td>add(f)</td>
<td>cdef</td>
</tr>
<tr>
<td>remove()</td>
<td>def</td>
</tr>
<tr>
<td>add(g)</td>
<td>defg</td>
</tr>
<tr>
<td>OPERATION</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-----</td>
</tr>
<tr>
<td>0123456789</td>
<td></td>
</tr>
<tr>
<td>add(a)</td>
<td>a</td>
</tr>
<tr>
<td>add(b)</td>
<td>ab</td>
</tr>
<tr>
<td>remove()</td>
<td>b</td>
</tr>
<tr>
<td>add(c)</td>
<td>bc</td>
</tr>
<tr>
<td>add(d)</td>
<td>bcd</td>
</tr>
<tr>
<td>add(e)</td>
<td>bcde</td>
</tr>
<tr>
<td>remove()</td>
<td>cde</td>
</tr>
<tr>
<td>add(f)</td>
<td>cdef</td>
</tr>
<tr>
<td>remove()</td>
<td>def</td>
</tr>
<tr>
<td>add(g)</td>
<td>defg</td>
</tr>
<tr>
<td>OPERATION</td>
<td>head</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>add(a)</td>
<td>a</td>
</tr>
<tr>
<td>add(b)</td>
<td>ab</td>
</tr>
<tr>
<td>remove()</td>
<td>b</td>
</tr>
<tr>
<td>add(c)</td>
<td>bc</td>
</tr>
<tr>
<td>add(d)</td>
<td>bcd</td>
</tr>
<tr>
<td>add(e)</td>
<td>bcde</td>
</tr>
<tr>
<td>remove()</td>
<td>cde</td>
</tr>
<tr>
<td>add(f)</td>
<td>cdef</td>
</tr>
<tr>
<td>remove()</td>
<td>def</td>
</tr>
<tr>
<td>add(g)</td>
<td>defg</td>
</tr>
</tbody>
</table>
removeFirst()
    tmp = head;
    head = head.next;
    tmp.next = null;
    size = size - 1;
}

addLast( newNode )
    tail.next = newNode;
    tail = tail.next;
    size = size + 1;

Queue as List

head tail size

[Image of a diagram with nodes and arrows representing the queue structure and operations.]

- Diagram showing the queue data structure with nodes labeled as `head`, `tail`, and `size`.
- Visual representation of the `removeFirst` and `addLast` operations on a queue with nodes connected by arrows.
Queue as Array

Array of shapes:

Size=5
Queue as Array

Size=6
Queue as Array

Size=5
Queue as Array

Size=5
Queue as Array

But why ??

Size=5
Queue as Array

0 (H) 2 3 4 (T) 6 7

[•,•,•,•,•,•, •, •]

head=1, tail=5, (size=5)
Queue as Array

head=3, tail=7, (size=5)
Queue as Array

head=3, tail=7, (size=5)
Queue as Array

head=3, tail=0, (size=6)
Queue as Array

head=3, tail=0, (size=6)
Queue as Array

head=3, tail=2, (size=8)
Queue as Array

head=3, tail=2, (size=8)

FULL!!
Queue as Array

enqueue( element ){    
    // array implementation
    if ( size == length)
        increase length of array // *** SEE BELOW **
    a[ (head + size) % length ] = element
    size = size + 1
}

dequeue(){
    out = a[head]
    head = (head + 1) % length
    size = size - 1
    return out
}
Queue as Array

// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) % small.length ]

head = 0
tail = small.length-1
size = small.length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
  big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
  big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length

array size length

[•, 4, 4]

[•, 4, 4, T, •, •, •, •, •, •, •, •, •, •, •, •]
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[(head + i) % small.length]
head = 0
tail = small.length-1
size = small.length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length

// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[(head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length

array size length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length

// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
  big[i] = small[(head + i) % small.length]
head = 0
tail = small.length-1
size = small.length
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
Running Times and Asymptotic Notation
As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time?

- Charles Babbage
Computational Tractability

**Brute force.** For many non-trivial problems, there is a natural brute force search algorithm that tries every possible solution.

- Typically takes $2^N$ time or worse for inputs of size $N$.
- Unacceptable in practice.

**Desirable scaling property.** When the input size doubles, the algorithm should only slow down by some constant factor $C$.

There exists constants $a > 0$ and $d > 0$ such that on every input of size $N$, its running time is bounded by $aN^d$ steps.

**Def.** An algorithm is poly-time if the above scaling property holds.

even worse: $N!$ for some problems

choose $C = 2^d$
Worst Case Analysis
Worst Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on any input of a given size N.
- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.
- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.
Winter 2016
COMP-250: Introduction to Computer Science
Lecture 8, February 4, 2016