CSE 143
Lecture 4

Stacks and Queues

reading: "Appendix Q" (see course website)

slides adapted from Marty Stepp and Hélène Martin
http://www.cs.washington.edu/143/
Stacks and queues

- Today we will examine two specialty collections:
  - **stack**: Retrieves elements in the reverse of the order they were added.
  - **queue**: Retrieves elements in the same order they were added.
  - Less powerful, but optimized to perform certain operations quickly.

![Stack and queue diagram](image-url)
Abstract data types (ADTs)

• **abstract data type (ADT):** A specification of a collection of data and the operations that can be performed on it.
  – Describes *what* a collection does, not *how* it does it

• We don't know exactly how a stack or queue is implemented, and we don't need to.
  – We just need to understand the idea of the collection and what operations it can perform.

(Stacks are usually implemented with arrays; queues are often implemented using another structure called a linked list.)
Stacks

- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - Elements are stored in order of insertion.
    - We do not think of them as having indexes.
  - Client can only add/remove/examine the last element added (the "top").

- **basic stack operations**:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
  - **peek**: Examine the top element.
Stacks in computer science

• Programming languages and compilers:
  – method calls are placed onto a stack \((\text{call}=\text{push}, \text{return}=\text{pop})\)
  – compilers use stacks to evaluate expressions

• Matching up related pairs of things:
  – find out whether a string is a palindrome
  – examine a file to see if its braces \{ \} match
  – convert "infix" expressions to pre/postfix

• Sophisticated algorithms:
  – searching through a maze with "backtracking"
  – many programs use an "undo stack" of previous operations
Stack\langle E \rangle ()  
constructs a new stack with elements of type \( E \)

**push** (value)  
places given value on top of stack

**pop**()  
removes top value from stack and returns it;  
throws **EmptyStackException** if stack is empty

**peek**()  
returns top value from stack without removing it;  
throws **EmptyStackException** if stack is empty

**size**()  
returns number of elements in stack

**isEmpty**()  
returns true if stack has no elements

---

Stack\langle Integer \rangle  
s = new Stack\langle Integer \rangle ();
s.push(42);
s.push(-3);
s.push(17);  // bottom [42, -3, 17] top
System.out.println(s.pop());  // 17

- Stack has other methods, but we forbid you to use them.
You cannot loop over a stack in the usual way.

```java
Stack<Integer> s = new Stack<Integer>();
...
for (int i = 0; i < s.size(), i++) {
    do something with s.get(i);
}
```

Instead, you pull elements out of the stack one at a time.

- common idiom: Pop each element until the stack is empty.

```java
// process (and destroy) an entire stack
while (!s.isEmpty()) {
    do something with s.pop();
}
```
Exercise

• Consider an input file of exam scores in reverse ABC order:

  Yeilding      Janet      87
  White         Steven     84
  Todd          Kim        52
  Tashev        Sylvia     95
  ...

• Write code to print the exam scores in ABC order using a stack.
  – What if we want to further process the exams after printing?
Let's consider a method `max` that accepts a `Stack` of integers and returns the largest integer in the stack:

```
// Precondition: !s.isEmpty()
public static void max(Stack<Integer> s) {
    int maxValue = s.pop();
    while (!s.isEmpty()) {
        int next = s.pop();
        maxValue = Math.max(maxValue, next);
    }
    return maxValue;
}
```

The algorithm is correct, but what is wrong with the code?
What happened to my stack?

The code destroys the stack in figuring out its answer. To fix this, you must save and restore the stack's contents:

```java
public static void max(Stack<Integer> s) {
    Stack<Integer> backup = new Stack<Integer>();
    int maxValue = s.pop();
    backup.push(maxValue);
    while (!s.isEmpty()) {
        int next = s.pop();
        backup.push(next);
        maxValue = Math.max(maxValue, next);
    }
    while (!backup.isEmpty()) { // restore
        s.push(backup.pop());
    }
    return maxValue;
}
```
Queues

• **queue**: Retrieves elements in the order they were added.
  – First-In, First-Out ("FIFO")
  – Elements are stored in order of insertion but don't have indexes.
  – Client can only add to the end of the queue, and can only examine/remove the front of the queue.

• **basic queue operations:**
  – **add** (enqueue): Add an element to the back.
  – **remove** (dequeue): Remove the front element.
  – **peek**: Examine the front element.
Queues in computer science

• Operating systems:
  – queue of print jobs to send to the printer
  – queue of programs / processes to be run
  – queue of network data packets to send

• Programming:
  – modeling a line of customers or clients
  – storing a queue of computations to be performed in order

• Real world examples:
  – people on an escalator or waiting in a line
  – cars at a gas station (or on an assembly line)
Programming with Queues

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(value)</td>
<td>places given value at back of queue</td>
</tr>
<tr>
<td>remove()</td>
<td>removes value from front of queue and returns it; throws a NoSuchElementException if queue is empty</td>
</tr>
<tr>
<td>peek()</td>
<td>returns front value from queue without removing it; returns null if queue is empty</td>
</tr>
<tr>
<td>size()</td>
<td>returns number of elements in queue</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>returns true if queue has no elements</td>
</tr>
</tbody>
</table>

Queue<Integer> q = new LinkedList<Integer>();
q.add(42);
q.add(-3);
q.add(17);  // front [42, -3, 17] back
System.out.println(q.remove());  // 42

- IMPORTANT: When constructing a queue you must use a new LinkedList object instead of a new Queue object.
  • This has to do with a topic we'll discuss later called interfaces.
Queue idioms

• As with stacks, must pull contents out of queue to view them.

```java
// process (and destroy) an entire queue
while (!q.isEmpty()) {
    do something with q.remove();
}
```

– another idiom: Examining each element exactly once.

```java
int size = q.size();
for (int i = 0; i < size; i++) {
    do something with q.remove();
    (including possibly re-adding it to the queue)
}
```

• Why do we need the size variable?
We often mix stacks and queues to achieve certain effects.

- Example: Reverse the order of the elements of a queue.

```java
Queue<Integer> q = new LinkedList<Integer>();
qu.add(1);
qu.add(2);
qu.add(3); // [1, 2, 3]

Stack<Integer> s = new Stack<Integer>();
while (!q.isEmpty()) { // Q -> S
    s.push(q.remove());
}
while (!s.isEmpty()) { // S -> Q
    q.add(s.pop());
}
System.out.println(q); // [3, 2, 1]
```
Exercise

• Modify our exam score program so that it reads the exam scores into a queue and prints the queue.

  – Next, filter out any exams where the student got a score of 100.

  – Then perform your previous code of reversing and printing the remaining students.

• What if we want to further process the exams after printing?
Exercises

• Write a method `stutter` that accepts a queue of integers as a parameter and replaces every element of the queue with two copies of that element.
  
  - `front [1, 2, 3] back`
  
  becomes
  
  `front [1, 1, 2, 2, 3, 3] back`

• Write a method `mirror` that accepts a queue of strings as a parameter and appends the queue's contents to itself in reverse order.
  
  - `front [a, b, c] back`
  
  becomes
  
  `front [a, b, c, c, b, a] back`