Stacks

Data Structures and Design with Java and JUnit
© Rick Mercer
Stacks

A Stack

- Is a collection with one accessible element--the most recent element "pushed" onto the stack
- Has a Last-In, First-Out (LIFO) characteristic
- Used often in computer systems
  - maintain order of method calls (demo stack of calls)
  - compiler matches openers \{ [ ( and closers ) ] \}
  - converting expressions into postfix notation
- Useful in applications where a last-in-first-out characteristic makes sense to the programmer
Using Java’s Stack Object

// Construct an empty stack
Stack<Integer> stackOfInts = new Stack<Integer>();

// Push three Integer values onto the stack
stackOfInts.push(1);
stackOfInts.push(2);
stackOfInts.push(3);

// Show each element before removed in a LIFO order
while (!stackOfInts.isEmpty()) {
    // Print the value as it's removed from top of stack
    System.out.println(stackOfInts.pop());
}
The Java Stack\(<E>\>

- Stack methods:
  - `push` elements onto the "top" of a stack.
  - `pop` elements off the "top" of the stack
  - `isEmpty` true if there are no elements
  - `peek` provides access only the top element
    - the most recently pushed element

- Some methods throw `EmptyStackException`
  - `pop` and `peek`
/**
 * Check if stack is empty to help
 * avoid popping an empty stack.
 * @returns true if there are 0 elements on this stack
 */
public boolean isEmpty()

/**
 * Put element on "top" of this Stack object.
 * @param element to be placed at the top of this stack
 */
public void push(E element)
/**
 * Return reference to the element at the top of stack
 * @returns A reference to the top element.
 * @throws EmptyStackException if the stack is empty.
 */

public E peek() throws EmptyStackException

/**
 * Remove element at top and return reference to it
 * @returns reference to the most recently pushed element
 * @throws EmptyStackException
 * @throws EmptyStackException if the stack is empty.
 */

public E pop() throws EmptyStackException
}
The memory shown to the right after executing the code on the left:

```java
Stack<String> s = new Stack<String>();
s.push("A");  // strings ok, chars not
s.push("B");
s.push("C");
```

What is the value of `s.peek()` now?

Here's what happens with `s.pop();`
Example 1:
Compiler checks for Balanced symbols

- Imagine a compiler scanning code when you forget `)` or `[` or `]`
- It's difficult to find the source of the error
  - compilers check to see if things aren't balanced
- Push opening symbols onto a stack and see if closing symbols make sense
  - compile a Java with several errors
Balancing Symbols

openers [ ( { and closers ] ) ]

public class A
    public static void main(String[] args
        System.out println( "Hello" );
        for( int j = 0; j < 6 j++ ) j++
        double x = 0.0;
        Integer j = 0;
        System.out.println( "Goodbye" );
    }
}

- Java says 2 errors, but how many can you find?

A.java:1: '{' expected.
A.java:12: Unbalanced parentheses
2 errors
Checks Balanced Symbols First

- Java's compiler apparently first checks to see if certain symbols are balanced \[ \] \{ \} ( )
- It ignores others errors by making a run over the entire source code just looking for unbalanced symbols
- If it finds none, it will begin a new pass
  - starts reading character by character again
- Fix the unbalanced errors of the previous slide one by one to observe this behavior
  - it probably uses a Stack and an algorithm like this
Example

- Process these characters, which represent *only* the openers and closers in a short Java program: `{ {{ [ ] ) } }`

- As the first four characters are read — all openers — push each onto the stack

```java
[ 
( 
{ 
{ 
```
Pop the first closer
Do they match?

- The next symbol read is a closer: ']').
- Pop '[' and compare to ']'.
- Since they match, no error would be reported.
  The stack would now look like this:

```
( {
  {
  )
```
Then ' )' is found, so pop the stack
Since the top symbol '( matches the closer ')', no error needs to be reported.
The stack now has two opening curly braces

```plaintext
{
{
  Still need to process
}
}
```
Pop last two. They match.
All is okay

- The two remaining closing curly braces would cause the two matching openers to be popped with no errors
- It is the last in first out nature of stacks that allows the first '{' to be associated with the last closer '}'. 
When do errors occur?

- If a closer is found and the stack is empty, you could also report an error.
  - For example with } }, where the opening { was not incorrectly typed, the stack has no openers.

- If you process all symbols and the stack is not empty, an error should be reported,
  - In the case of } } ( [ ] ) there is a missing right curly brace. Symbols are processed without error.
  - Except at the end, the stack should be empty. It is not and an error gets reported.
Algorithm

Make an empty stack named $s$
Read symbols until end of file
  if it's an opening symbol
    push it
  otherwise
    if it is a closing symbol && $s$.empty
      report error
    otherwise
      pop the stack
      if symbol is not a closer for pop's value
        report error
 3. At end of file, if !$s$.empty, report error(s)
Example 2:
Evaluating postfix expressions

- Stacks set up the evaluation of expressions.
  
  \[ 4 + 8 / 2 \] is different if evaluated left to right.

- Evaluating "infix" expressions is not easy.
  
  - So compilers convert what we write in infix into the equivalent postfix expression.

- The expression 2 plus 2 in postfix \[ 2 \ 2 \ + \]

- Postfix of \[ 1 - 2 + 3 \times 3 / 4 \] is
  
  \[ 1 \ 2 \ - \ 3 \ 3 \ * \ 4 \ / \ + \]
Evaluating Postfix Expression

- How do we evaluate these expressions?
- Steps for evaluating postfix expressions
  
  Make an empty stack \( s \)
  
  For each token (operators or operands) in the infix expression
  
  if token is an operand (valid integers only)
  
  \[ s\text{.push(operand)} \]
  
  if token is an operator such as + * - /
  
  \[ \text{right} = s\text{.pop()} \]
  \[ \text{left} = s\text{.pop()} \]
  
  push the value of the operator applied to the left and right
Evaluate \[3 \ 4 - 5 \ 3 \ * - \]

Found operand so push
Found operand so push
Found operator so pop two values, apply operand, and push the result

s.push(3);
s.push(4);
right = s.pop(); // found operator - so pop
left = s.pop();
s.push(left - right);

\[3 - 4\]

The stack now has one value -1
The remainder of the expression: \[5 \ 3 \ * -\]
Continue with  5 3 * -

Found operand so push
Found operand so push
Found operator so pop two values, apply operand, and push the result

s.push(5);
5
s.push(3);
3
right = s.pop();
-1
left = s.pop();
15
s.push(left * right);
5 * 3
-1

The Stack has two values
Only one token remains
Continue with –

right = s.pop();
left = s.pop();
s.push(left - right);
$-1 - 15$

<table>
<thead>
<tr>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
</tr>
</tbody>
</table>

The expression has been processed
The value at the top of the stack is the value of the expression is -16
You try it. Trace with a stack

// Use s to evaluate 2 3 4 * 5 * -
Stack<Integer> s = new Stack<Integer>();

For each token (operators or operands) in the infix expression
if token is an operand (valid integers only)
   s.push(operand)
if token is an operator (+ * -, or /)
   right = s.pop()
   left = s.pop()
   push the value of the operator applied to the left and right
Example 3:
Converting Infix to Postfix

\[1 + 2 \times 3 ^ 4\] in postfix \[1 \ 2 \ 3 \ 4 \ ^ \ * \ +\]

Note: \(^\) is a symbol in some languages for exponentiation

- Operators are in reverse order
  - So we need to store them on a stack
  - When an operand is encountered, pop higher order operands before pushing the lower order operand

- Algorithm on the next slide
Part of an algorithm for creating a postfix "String"

Let postfix be a String that is "" and stack be an empty Stack
For each token in the input stream {
  if operand: Immediately concatenate operand to postfix
  if open parentheses: Push '(' on the stack
  if close parentheses: Pop stack symbols and attach to postfix until peek is an open parentheses, then pop '(
  if operator: While the stack is not empty, pop all operators as long as they are of equal or higher precedence and the top is not '. Concatenate each operator from stack to postfix. Then push the current operator.
}
Pop any remaining operators from the stack, concatenating them to the postfix expression

Note: Left parentheses are treated as a high precedence operator on input but as a low precedence operator when on the stack

Example: 2* ((3 + 4) / 2– 6)
Queues
A Queue ADT

First in first out access

- A Queue is another name for waiting line
- Queues provide **First In First Out (FIFO)** access to elements *could also say Last In Last Out (LILO)*
Example of Queue usage
First in first out access

- Submit jobs to a network printer
  - Print the least recently submitted \textit{no priorities given}
  - Add new print jobs at the end of the queue
- Packets (chunks of bits) come into a router and get sent out
- Wait for incoming requests to a server
- Waiting line simulations
- 335 Jukebox PlayList to play mp3s in order
The Java Queue class?

- Some languages have a Queue class or queue is part of a library that works with the language
  - Java 1.4 used class LinkedList to offer FIFO functionality by adding methods addLast(Object) and Object(getFirst)
  - Java 1.5 added a Queue interface and several collection classes: ArrayBlockingQueue<E> and LinkedListBlockingQueue<E>

- Outline of what we'll do
  - Specify a Queue ADT as a Java interface
  - Show a difficult to implement array based implementation
  - Implement a linked version
Queues typically provide these operations

- `add` adds an element at the end of the queue.
- `peek` returns a reference to the element at the front of the queue.
- `remove` removes the element from the front of the queue and returns a reference to the front element.
- `isEmpty` returns false if there is at least one element in the queue.
Specify an interface

- We will use an interface to describe a queue ADT
  - The interface specifies method names, return types, the type of elements to add, and hopefully comments

- interface **OurQueue** declares we must be able to **add** and **remove** any type of element
  - Collection class must have <E> to make it a generic type
public interface OurQueue<E> {
    // Return true if this queue has 0 elements
    public boolean isEmpty();

    // Store a reference to any object at the end
    public void add(E newEl);

    // Return a reference to the object at the
    // front of this queue
    public E peek() throws NoSuchElementException;

    // Remove the reference to the element at the front
    public E remove() throws NoSuchElementException;
}
Let SlowQueue implement the Queue interface

- We need to store an Object[] an array of Object objects
  - avoids having queue of int and people and cars, and...

```java
public class SlowQueue<E> implements OurQueue<E> {
    private int back;
    private Object[] data;
    // ...
}
```

- Now implement all methods of the OurQueue interface as they are written
  - plus a constructor with the proper name
Bad array type queue

- Queue as an array could have
  - the front of the queue is always in \texttt{data [0]}

```java
public SlowQueue(int max) {
    data = new Object[max];
    back = -1;
}
```

\textbf{back} \texttt{== \text{-1}}

\begin{tabular}{cccc}
  null & null & null & null \\
\end{tabular}

So far so good. An empty queue
public void add(E element) {
    // This method will be changed later
    back++;
    data[back] = element;
}

*Send an add message*

    aQueue.add("a");

```
    "a"       null    null    null
```

back == 0

*So far so good. A queue of size 1*
add another

```java
public void add(E element) {
    // This method will be changed later
    back++;
    data[back] = element;
}
```

- Send two more add messages

```java
    aQueue.add("b");
    aQueue.add("c");
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;a&quot;</td>
<td>&quot;b&quot;</td>
<td>&quot;c&quot;</td>
<td>null</td>
</tr>
</tbody>
</table>

`back == 2`

So far so good. A Queue of size 3
Array Implementation of a Queue

- During remove, slide all elements left if size were 999, then 998 assignments would be necessary.

Before remove

```
| a | b | c | null |
```

After remove

```
| b | c | c | null |
```

A poor remove algorithm
Effect of queue operation on an array with a "floating" front

```
add("a")
add("b")
add("c")
remove()
add("d")
remove()
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>front</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>back</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
remove()
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>front</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>back</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What happens next when back equals array.length?

- back indexes the last array index
- However, this queue is not full
- Where do you place "e"?
  - data[0] is available
  - What code would increment back correctly even when back is referencing the last available array index

? _______________________________ ?
A "circular queue" implementation uses wraparound. The queue has "c" "d" "e"

- either increase back by 1
- or set back to 0

add("e") now works in this "circular" queue. It reuses previously used array indexes.
Implementing a Circular Queue

- Still have to work with arrays, not circles
  - In order for the first and last indices to work in a circular manner:
    - increase by one element at a time
    - after largest index in the array, go to zero.
      back = 0 1 2 3 0 1 2 3 0 1 2 3 0 ...
  - could contain code you just wrote on previous slide

- But what is an empty queue?
  - What values should be given to front and back when the queue is constructed?
Problem: A full queue can not be distinguished from an empty queue

One option is to have the constructor place `back` one index before `front` then increment `back` during `add`.

What does `back == front` imply? An empty or full queue?
Corrected Circular Queue

- Use this trick to distinguish between full and empty queues
  - The element referenced by `front` never indexes the front element— the “real” front is located at `nextIndex(front)`

```java
private int nextIndex(int index) {
    // Return an int to indicate next position
    return (index + 1) % data.length;
}
```

- For example, use this during `peek()`
  ```java
  return data[nextIndex(front)];
  ```
Correct Circular Queue
Implementation Illustrated

The front index is always 1 behind the actual front
This wastes one array element  

*but it's no big deal*
Correct Circular remove Implementation Illustrated

remove three times to make this queue empty
Use a singly linked structure

```java
public class LinkedQueue<E> implements OurQueue<E> {

    private class Node {
        private E data;
        private Node next;

        public Node(E element) {
            data = element;
            next = null;
        }
    }

    Keep two external references, front and back

    Empty queue

    front

    back
```