CSE 373

Implementing a Stack/Queue as a Linked List

Reading: Weiss Ch. 3; 3.7

slides created by Marty Stepp
http://www.cs.washington.edu/373/

© University of Washington, all rights reserved.
The Stack\langle E\rangle ADT

- We have now implemented a stack using an array.
  - It is also just fine to implement a stack using a linked list.
  - Let's write a `LinkedStack` class...

```java
public interface Stack<E> {
    void clear();
    boolean isEmpty();
    E peek();
    E pop();
    void push(E value);
    int size();
}

public class LinkedStack<E> implements Stack<E> {
    ...
```
Recall: linked nodes

```java
class Node<E> {
    public E data;
    public Node next;
}
```

- Each node object stores:
  - one piece of data
  - a reference to (at least) one other list node

- Nodes can be "linked" into chains to store a list/stack of values:
Null references

• **null**: A value that does not refer to any object.
  - The elements of a new array of objects are initialized to `null`.
  - Objects' fields are also `null` by default until initialized.

```java
String[] words = new String[5];
private Point myLocation;     // null
```

- not the same as the empty string `""` or the string `"null"
- Why does Java have `null`? What is it used for?
- You cannot *deference* a `null` variable (use a dot `. `on it).
Recall: Linked list

- Previously you have implemented a linked list.
  - The list is internally implemented as a chain of linked nodes.
    - The LinkedList keeps a reference to its front as a field
    - null is the end of the list; a null front signifies an empty list
    - fast O(1) to add/remove at front, slow O(N) at back (opposite of array)

```
front
add(value)
add(index, value)
remove(index)
get(index)
isEmpty()
set(index, value)
size()
toString()
```

Node data | next
--- | ---
42 |  

Node data | next
--- | ---
-3 |  

Node data | next
--- | ---
element 0 | 
element 1 | 
element 2 | 
Inner classes

// outer (enclosing) class
public class name {
    ...

    // inner (nested) class
    private class name {
        ...
    }
}

- Only the outer file can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
- Usually it makes sense to make your linked Node into an inner class.
- An inner class sees the generic type parameters (like \(<E>\)) of its enclosing outer class, so it should not try to re-declare them.
Implementing push

• How do we push an element onto the end of a stack?

```java
public void push(E value) {
    Node newNode = new Node(value);
    newNode.next = front;
    front = newNode;
    size++;
}
```

```
s.push(42);  // client code
```

```
front = 42

```

```
42  next
```

```
-3  next
```

```
17  next
```

Front = 42

Element 0

Element 1

Element 2
Implementing pop

- How do we pop an element off the end of a stack?

```java
public int pop() {
    int top = front.data;
    front = front.next;
    size--;
    return top;
}
```

`s.pop();  // client code;  returns 42`
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.
Let's write our own implementation of a queue.

- As is done in the Java Collection Framework, we will define queues as an ADT by creating a queue interface.

```java
public interface Queue<E> {
    void clear();
    boolean isEmpty();
    E peek();
    E remove(); // remove from back
    void add(E value); // add to front
    int size();
}
```
public class ArrayQueue<E> implements Queue<E> {
    private E[] elements;
    private int size;
    ...

    • A queue is tough to implement efficiently with an unfilled array.
      ▪ The array is fast to add/remove at the end, but slow at the front.

    queue.add(26);    // client code
    queue.add(-9);
    queue.add(14);
    queue.remove();

    +---+---+---+---+---+---+---+---+---+---+
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
    +---+---+---+---+---+---+---+---+---+---+
    | value | 26 | -9 | 14 | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
    +---+---+---+---+---+---+---+---+---+---+
    | size | 3   |     |     |     |     |     |     |     |     |     |
We could implement a queue as a linked list of nodes. Good idea?

- **problem**: fast O(1) to add/remove at front, slow O(N) at back
- queue needs fast O(1) access to both ends

LinkedQueue

```
front
add(value) remove() isEmpty() size() toString()
```

<table>
<thead>
<tr>
<th>Node</th>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>42</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

element 0  element 1  element 2
Doubly linked list

- Nodes in a *doubly linked list* keep references in two directions.
  - The list itself keeps a front and back reference. Fast at both ends!
  - Often implemented with "dummy" nodes at each end to avoid nulls

```
prev  data  next
42     /      /
        |      |
element 0

prev  data  next
42     /      /
        |      |
element 1

prev  data  next
42     /      /
        |      |
element 2
```

```
prev  data  next
/       /      /
        |      |
front

add(value)
remove()
...

prev  data  next
/       /      /
        |      |
back
```
Circular array buffer

- **idea:** when elements are added/removed from front, rather than shifting, array simply alters its definition of the "front" index.

```java
queue.add(26);   // client code
queue.add(-9);
queue.add(14);

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>26</td>
<td>-9</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

queue.remove();  // returns 26
queue.remove();  // returns -9
queue.add(87);
queue.add(35);

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>87</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>3</td>
<td></td>
<td>14</td>
<td>87</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Iterators

- **iterator**: An object that allows a client to traverse the elements of a collection, regardless of its implementation.
  - Remembers a position within a collection, and allows you to:
    - get the element at that position
    - advance to the next position
    - remove the element at that position
  - A common way to examine *any* collection's elements.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>42</td>
<td>-3</td>
<td>17</td>
<td>29</td>
</tr>
</tbody>
</table>

```
set
"the"
"to"
"we"
"from"
```

```
current element: "from"
next element: "the"
```

```
data next
42
```

```
data next
-3
```

```
data next
17
```

```
front
```

```
iterator
current element: -3
current index: 1
```

```
iterator
current element: -3
current index: 1
```

```
```
# Iterator methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNext()</td>
<td>returns <strong>true</strong> if there are more elements to examine</td>
</tr>
<tr>
<td>next()</td>
<td>returns the next element from the collection (throws a <code>NoSuchElementException</code> if there are none left to examine)</td>
</tr>
<tr>
<td>remove()</td>
<td>removes the last value returned by <code>next()</code> (throws an <code>IllegalStateException</code> if you haven't called <code>next()</code> yet)</td>
</tr>
</tbody>
</table>

- **Iterator interface in java.util**
  - every collection has an `iterator()` method that returns an iterator over its elements (usually implemented as an inner class)

```java
Set<String> set = new HashSet<String>();
...
Iterator<String> itr = set.iterator();
...```
Iterator example

Set<Integer> scores = new TreeSet<Integer>();
scores.add(94);
scores.add(38);    // Jenny
scores.add(87);
scores.add(43);    // Marty
scores.add(72);
...

Iterator<Integer> itr = scores.iterator();
while (itr.hasNext()) {
    int score = itr.next();
    System.out.println("The score is " + score);

    // eliminate any failing grades
    if (score < 60) {
        itr.remove();
    }
}
System.out.println(scores); // [72, 87, 94]
Bad linked list usage

• What's bad about this code?

List<Integer> list = new LinkedList<Integer>();
...
for (int i = 0; i < list.size(); i++) {
    int value = list.get(i);
    if (value % 2 == 1) {
        list.remove(i);
    }
}

front =

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

element 0

data | next |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td></td>
</tr>
</tbody>
</table>

element 1

data | next |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

element 2

... (add lots of elements) ...
Iterators and linked lists

- Iterators are particularly useful with linked lists.
  - The previous code is $O(N^2)$ because each call on `get` must start from the beginning of the list and walk to index $i$.
  - Using an iterator, the same code is $O(N)$. The iterator remembers its position and doesn't start over each time.

```
front = [42]

data  next
---   ---
42    

current element: -3
current index: 1
```

```
element 0

data  next
---   ---
42    

data  next
---   ---
-3    

element 1

data  next
---   ---
-3    

data  next
---   ---
17    

element 2
```
Array stack iterator

// Traverses the elements of the stack from top to bottom.
private class ArrayStackIterator implements Iterator<E> {
    private int index;

    public ArrayStackIterator() {
        index = size - 1;
    }

    public boolean hasNext() {
        return index >= 0;
    }

    public E next() {
        E result = elements[index];
        index--;
        return result;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
Linked stack iterator

// Traverses the elements of the stack from top to bottom.
private class LinkedStackIterator<T> implements Iterator<T> {
    private Node position;   // current position in list

    public LinkedStackIterator() {
        position = front;
    }

    public boolean hasNext() {
        return position != null;
    }

    public T next() {
        T result = position.data;
        position = position.next;
        return result;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
for-each loop and Iterable

• Java's collections can be iterated using a "for-each" loop:

```java
List<String> list = new LinkedList<String>();
...
for (String s : list) {
    System.out.println(s);
}
```

  - Our collections currently do not work in this way.

• To fix this, your list must implement the `Iterable` interface.

```java
public interface Iterable<E> {
    public Iterator<E> iterator();
}
```

```java
public class ArrayStack<E> implements Iterable<E> ...
```
### ListIterator

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(value)</td>
<td>inserts an element just after the iterator's position</td>
</tr>
<tr>
<td>hasPrevious()</td>
<td>true if there are more elements before the iterator</td>
</tr>
<tr>
<td>nextIndex()</td>
<td>the index of the element that would be returned the next time next is called on the iterator</td>
</tr>
<tr>
<td>previousIndex()</td>
<td>the index of the element that would be returned the next time previous is called on the iterator</td>
</tr>
<tr>
<td>previous()</td>
<td>returns the element before the iterator (throws a NoSuchElementException if there are none)</td>
</tr>
<tr>
<td>set(value)</td>
<td>replaces the element last returned by next or previous with the given value</td>
</tr>
</tbody>
</table>

ListIterator<String> li = myList.listIterator();

- lists have a more powerful ListIterator with more methods
  - can iterate forwards or backwards
  - can add/set element values (efficient for linked lists)