

Bags, stacks and queues

*Please note that these slides are based in part on
material originally developed by Prof. Kevin Wayne of
the CS Dept at Princeton University.*

Bag: API

Note the use of a generic type for the items



```
public interface Bag<Item> extends Iterable<Item> {  
    /**  
     * Update this Bag by adding item.  
     * No guarantee is made regarding the ordering of Items in the iterator  
     * @param item the item to add  
     */  
    void add(Item item);  
    /**  
     * @return true if this bag is empty  
     */  
    boolean isEmpty();  
    /**  
     * @return the number of elements in this bag (not the capacity which is an  
     * implementation-dependent feature)  
     */  
    int size();  
}
```

Note the API is an interface which extends another



Note the javadoc essentially documents the API



Is there anything missing?



Example of usage of iterator:


```
for (Integer t : bag)  
    StdOut.println(t);
```

Stack: API


```
public interface Stack<Item> {
```

```
    /**
     * Update this Stack by adding an item on the top.
     * @param item the item to add
     */
    void push(Item item);
    /**
     * Update this Stack by taking the top item of this Stack.
     * @return the item.
     * @throws Exception
     */
    Item pop() throws Exception;
    /**
     * Take the peek at the item on top of this Stack.
     * @return the item (return null if there is no such item).
     */
    Item peek();
    /**
     * @return true if this stack is empty
     */
    boolean isEmpty();
}
```

Stack does not
implement *Iterable*.



We allow this method in
order to avoid
exception when
popping an empty
Stack



Queue: API

```
public interface Queue<Item> {
```

```
/**
 * Update this Queue by adding an item on the "newest" end.
 * @param item the item to add
 */
```

```
void enqueue(Item item);
```

```
/**
 * Update this Queue by taking the oldest item off the queue.
 * @return the item or null if there is no such item.
 */
```

```
Item dequeue();
```

```
/**
 * @return true if this stack is empty
 */
```

```
boolean isEmpty();
```

```
}
```

Queue does not
implement *Iterable* either
—although it could.

We don't need to throw
an exception on
dequeue when the
queue is empty.

Notice any differences...

- ...from the book?
 - This is the tricky part of designing an API: there are frequently different opinions. My rules:
 1. do not add any signatures that are not absolutely essential;
 2. split mutating and non-mutating methods into separate interfaces;
 3. separate different concerns.
 - Also, notice that I used interfaces, not classes. Unfortunately, the Java designers started out with a lot of concepts which they implemented as classes (or abstract class) which should have been interfaces (IMO, of course).

Bag: Implementation

```
import java.util.Arrays;
import java.util.Iterator;
public class Bag_Array<Item> implements Bag<Item> {
    public Bag_Array() {
        grow((Item[])new Object[0], 32);
    }
    public void add(Item item) {
        if (full())
            grow(items, 2 * capacity());
        items[count++] = item;
    }
    public boolean isEmpty() {
        return count==0;
    }
    public int size() {
        return count;
    }
    public Iterator<Item> iterator() {
        return Arrays.asList(Arrays.copyOf(items, count)).iterator();
    }
    private void grow(Item[] source, int size) {
        items = growFrom(source, size);
    }
    private int capacity() {
        return items.length; // items should always be non-null when this method is called
    }
    private boolean full() {
        return size()==capacity();
    }
    private static <T> T[] growFrom(T[] from, int size) {
        T[] result = (T[])new Object[size];
        System.arraycopy(from, 0, result, 0, from.length);
        return result;
    }
    private Item[] items = null;
    private int count = 0;
}
```

Note the name: concrete classes should have a name that describes how they implemented the interface.

Here we implement the signatures defined by *Bag* and *Iterator*

Note that Java does not provide an iterator for an array :(

Generally, we should put the private stuff at the end of the class.

Bag: Testing

```
import org.junit.Test;

import static org.junit.Assert.*;

public class BagTest {
    /**
     * Test method for Bag
     */
    @Test
    public void testBag() {
        Bag<Integer> bag = new Bag_Array<>();
        assertTrue(bag.size()==0);
        assertTrue(bag.isEmpty());
        assertFalse(bag.iterator().hasNext());
        bag.add(1);
        assertTrue(bag.size()==1);
        assertFalse(bag.isEmpty());
        assertTrue(bag.iterator().hasNext());
        assertEquals(bag.iterator().next(), new Integer(1));
    }
}
```

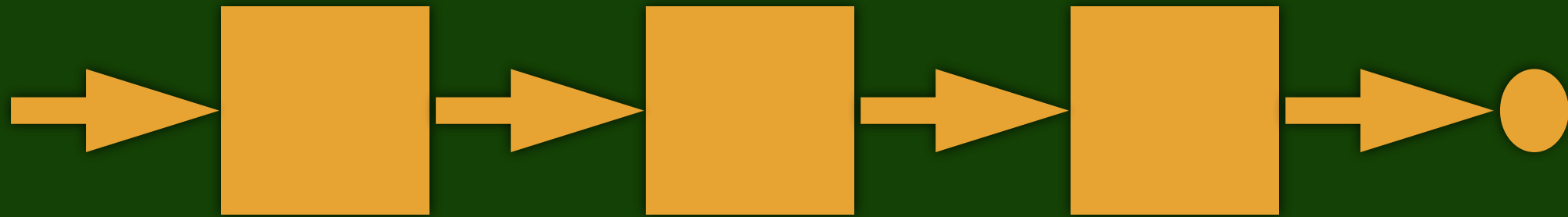
Comparison of storage methods

Technique	Access	Add to Head	Add to Tail	Copy
Array	$O(1)$	$O(1)^*$	$O(1)^*$	$O(N)$
Linked List	$O(N)^\dagger$	$O(1)$	$O(N)$	$O(1)$ or $O(N)$ if mutable
Doubly-linked List	$O(N)^\dagger$	$O(1)$	$O(1)$	$O(1)$ or $O(N)$ if mutable

* Except when full: in which case, a copy is required;

† To be precise, $N/2$

Linked Lists



- Each element has two fields:
 - The value of this element;
 - A pointer/reference to the next element (which may be null).
- Addition/removal of an element:
 - at the head is $O(1)$, i.e. constant;
 - at the tail is $O(N)$, i.e. it varies according to the current length N

Stack using LinkedList

- A linked list is perfectly suited to a Stack, because all addition/removal operations (push, pop) happen at the head.
- You can also implement a Stack with an array.

LinkedList: Implementation

```
public class LinkedList<Item> {  
    public void add(Item item) {  
        Element tail = head;  
        head = new Element(item, tail);  
    }  
  
    public Item remove() throws BQSEException {  
        if (head == null) throw new BQSEException("collection is empty");  
        Item result = head.item;  
        head = head.next;  
        return result;  
    }  
  
    public Item getHead() {  
        return isEmpty() ? null : head.item ;  
    }  
  
    public boolean isEmpty() {  
        return head==null;  
    }  
  
    private class Element {  
        Element(Item x, Element n) {  
            item = x;  
            next = n;  
        }  
        final Item item;  
        final Element next;  
    }  
  
    private Element head = null;  
}
```

For this presentation, at least,
we don't create an interface
out of this. But in our repo it is
an interface.



Private inner class *Element* is
immutable




head is mutable which means
LinkedList is mutable.



Stack: Implementation

```
public class Stack_LinkedList<Item> implements Stack<Item> {  
    public Stack_LinkedList() {  
        list = new LinkedList<>();  
    }  
    public void push(Item item) {  
        list.add(item);  
    }  
    public Item pop() throws RuntimeException {  
        return list.remove();  
    }  
    public Item peek() {  
        return list.getHead();  
    }  
    public boolean isEmpty() {  
        return list.isEmpty();  
    }  
    private final LinkedList<Item> list;  
}
```

All methods are
delegated to
appropriate LinkedList
method (three name
changes)



Note that list is marked
final



Dijkstra's two-stack algorithm

- What's the value of $(1 + ((2 + 3) * (4 * 5)))$?
- If we had a stack, we could use so-called Reverse Polish Notation:
 - 1 2 3 + 4 5 * * +
 - 1
 - 1,2
 - 1,2,3
 - 1,5
 - 1,5,4
 - 1,5,4,5
 - 1,5,20
 - 1,100
 - 101

Movie

13DemoDijkstraTwoStack

Dijkstra's two-stack algorithm

- See repository: `DijkstraTwoStack.java`

LinkedLists and Queues

- A linked list is not perfectly suited to a Queue.
- Why not?

Doubly Linked Lists



- Each element has three fields:
 - The value of this element;
 - A pointer/reference to the next element (which may be null).
 - A pointer/reference to the previous element (which may be null).
- Addition/removal of an element:
 - at the head is $O(1)$, i.e. constant;
 - at the tail is $O(1)$, i.e. constant;

Queue with Elements

- Actually, there's a simpler way to implement a (standard) queue — which only ever enqueues or dequeues a single value at a time:
 - Use Elements (the same as in *LinkedList*)

Queue: Implementation

```
public class Queue_Elements<Item> implements Queue<Item> {  
    /**  
     * Construct a new (empty) queue.  
     */  
    public Queue_Elements() {  
        oldest = null;  
        newest = null;  
    }  
    /**  
     * Enqueue the given item into the linked list referenced by oldest  
     * @param item the item to add  
     */  
    public void enqueue( Item item) {  
        Element<Item> element = new Element<>(item);  
        Element<Item> secondNewest = newest;  
        if (isEmpty()) oldest = element;  
        else {  
            assert secondNewest != null; // Redundant Check  
            secondNewest.next = element;  
        }  
        this.newest = element;  
    }  
    /**  
     * Dequeue an element from the oldest list and return the item.  
     * @return the value of the oldest element.  
     */  
    public Item dequeue() {  
        if (isEmpty()) return null;  
        else {  
            assert oldest != null; // Redundant assertion  
            Item result = oldest.item;  
            oldest = oldest.next;  
            if (isEmpty()) newest = null;  
            return result;  
        }  
    }  
    public boolean isEmpty() {  
        return oldest == null;  
    }  
    // This Element essentially begins a LinkedList of Elements which correspond  
    // to the elements that can be taken from the queue (head points to the oldest element).  
    // However, it is built in manner that requires a pointer to the newest element.  
    private Element<Item> oldest;  
    // This element always points to the newest (tail-most) element in the LinkedList referenced by oldest.  
    private Element<Item> newest;  
    private class Element {  
        // same as for LinkedList  
    }  
}
```

newest always changes
but oldest only when
empty.

oldest always changes
but newest only when
empty.

oldest essentially implements a linked
list while newest points to its last
Element