Basic Data Structures

Outline

Generics in Java

Generics (aka parametrized types) enable us to implement collection ADTs that can store any type of data

Example

```
1  LinkedStack<String> s1 = new LinkedStack<String>();
2  LinkedStack<Date> s2 = new LinkedStack<Date>();
3  s1.push("03/14/1879");
4  s2.push(new Date(3, 14, 1879));
5  String s = s1.pop();
6  Date d = s2.pop();
```

Generics in Java

Java automatically converts a primitive type to the corresponding reference type (auto boxing) and vice versa (auto unboxing)

Example

```
1 LinkedStack<Integer> stack = new LinkedStack<Integer>();
2 stack.push(42);       // auto boxing (int -> Integer)
3 int i = stack.pop(); // auto unboxing (Integer -> int)
```

Wrapper types

boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
float	Float
long	Long
double	Double

Linked List

A linked list is a data structure that is either empty (`null`) or a reference to a node having a generic item and a reference to the rest of the linked list

Example

Linked list ADT

```
1  private class Node {
2      private Item item;
3      private Node next;
4  }
```

Linked List

Traversing a linked list:


```
1  for (Node x = first; x != null; x = x.next) {
2      StdOut.print(x.item + " ");
3  }
```

```
to be or not to be
```

Linked List

Building a linked list:

```
1  Node first = new Node();
2  first.item = "be";
```

Linked List

Inserting at the beginning:


```
1  Node oldfirst = first;
2  first = new Node();
3  first.item = "to";
4  first.next = oldfirst;
```

Linked List

Inserting at the end:


```
1  Node oldlast = last;
2  last = new Node();
3  last.item = "or";
4  oldlast.next = last;
```

Linked List

Remove from the beginning:


```
first = first.next;
```

Linked List

Operation	$T(n)$
Insert at the beginning	1
Insert at the end	1
Remove from the beginning	1

Bag

A `Bag` is an iterable collection that stores generic items

Bag<Item> extends Iterable<Item>

`boolean isEmpty()`	returns `true` if this bag is empty, and `false` otherwise
`int size()`	returns the number of items in this bag
`void add(Item item)`	adds `item` to this bag
`Iterator<Item> iterator()`	returns an iterator to iterate over the items in this bag

Bag

Program: `Stats.java`

- ⇝ Standard input: a sequence of doubles
- ⇝ Standard output: their mean and standard deviation

```
>_ ~/workspace/dsa/programs
```

```
$ java Stats
1 3 5 7 9
<ctrl-d>
Mean:    5.00
Std dev: 3.16
$ _
```

Bag

Stats.java

```
import dsa.LinkedBag;

import stdlib.StdIn;
import stdlib.StdOut;

public class Stats {
    public static void main(String[] args) {
        LinkedBag<Double> bag = new LinkedBag<Double>();
        while (!StdIn.isEmpty()) {
            bag.add(StdIn.readDouble());
        }
        int n = bag.size();
        double sum = 0.0;
        for (double x : bag) {
            sum += x;
        }
        double mean = sum / n;
        sum = 0.0;
        for (double x : bag) {
            sum += (x - mean) * (x - mean);
        }
        double stddev = Math.sqrt(sum / (n - 1));
        StdOut.printf("Mean:    %.2f\n", mean);
        StdOut.printf("Std dev: %.2f\n", stddev);
    }
}
```

Bag · Linked Bag

Instance variables:

- ⇝ Reference to the front of the bag: `Node first`

- ⇝ Number of items in the bag: `int n`

Bag · Linked Bag

LinkedBag.java

```
1  package dsa;
2
3  import java.util.Iterator;
4  import java.util.NoSuchElementException;
5
6  import stdlib.StdIn;
7  import stdlib.StdOut;
8
9  public class LinkedBag<Item> implements Bag<Item> {
10     private Node first;
11     private int n;
12
13     public LinkedBag() {
14         first = null;
15         n = 0;
16     }
17
18     public boolean isEmpty() {
19         return first == null;
20     }
21
22     public int size() {
23         return n;
24     }
25
26     public void add(Item item) {
27         Node oldfirst = first;
28         first = new Node();
29         first.item = item;
30         first.next = oldfirst;
31         n++;
32     }
33
34     public Iterator<Item> iterator() {
35         return new ListIterator();
```

LinkedBag.java

```
36      }
37
38      private class Node {
39          private Item item;
40          private Node next;
41      }
42
43      private class ListIterator implements Iterator<Item> {
44          private Node current;
45
46          public ListIterator() {
47              current = first;
48          }
49
50          public boolean hasNext() {
51              return current != null;
52          }
53
54          public Item next() {
55              if (!hasNext()) {
56                  throw new NoSuchElementException("Iterator is exhausted");
57              }
58              Item item = current.item;
59              current = current.next;
60              return item;
61          }
62
63          public void remove() {
64              throw new UnsupportedOperationException("remove() is not supported");
65          }
66      }
67
68      public static void main(String[] args) {
69          LinkedBag<String> bag = new LinkedBag<String>();
70          while (!StdIn.isEmpty()) {
```

Bag · Linked Bag

LinkedBag.java

```
71              String item = StdIn.readString();
72              bag.add(item);
73          }
74          StdOut.println("size of bag = " + bag.size());
75          for (String s : bag) {
76              StdOut.print(s + " ");
77          }
78          StdOut.println();
79      }
80  }
```

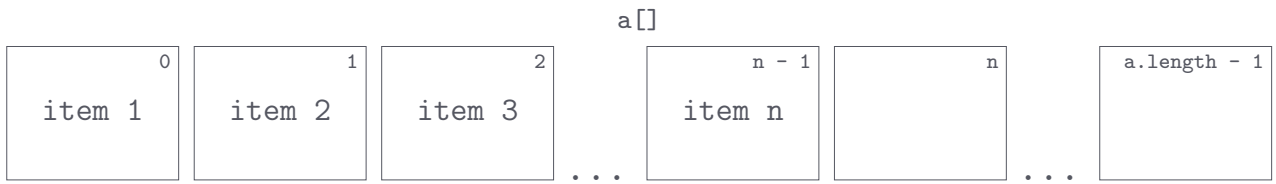
Bag · Linked Bag

Operation	$T(n)$
`LinkedBag()`	1
`boolean isEmpty()`	1
`int size()`	1
`void add(Item item)`	1
`Iterator<Item> iterator()`	1

Bag · Resizing Array Bag

Instance variables:

- ⇝ Array of items in the bag: `Item a[]`



- ⇝ Number of items in the bag: `int n`

Bag · Resizing Array Bag

ResizingArrayBag.java

```
1  package dsa;
2
3  import java.util.Iterator;
4  import java.util.NoSuchElementException;
5
6  import stdlib.StdIn;
7  import stdlib.StdOut;
8
9  public class ResizingArrayBag<Item> implements Bag<Item> {
10     private Item[] a;
11     private int n;
12
13     public ResizingArrayBag() {
14         a = (Item[]) new Object[2];
15         n = 0;
16     }
17
18     public boolean isEmpty() {
19         return n == 0;
20     }
21
22     public int size() {
23         return n;
24     }
25
26     public void add(Item item) {
27         if (n == a.length) {
28             resize(2 * a.length);
29         }
30         a[n++] = item;
31     }
32
33     public Iterator<Item> iterator() {
34         return new ArrayIterator();
35     }
```

Bag · Resizing Array Bag

ResizingArrayBag.java

```
36
37      private void resize(int capacity) {
38          Item[] temp = (Item[]) new Object[capacity];
39          for (int i = 0; i < n; i++) {
40              temp[i] = a[i];
41          }
42          a = temp;
43      }
44
45      private class ArrayIterator implements Iterator<Item> {
46          private int i;
47
48          public ArrayIterator() {
49              i = 0;
50          }
51
52          public boolean hasNext() {
53              return i < n;
54          }
55
56          public Item next() {
57              if (!hasNext()) {
58                  throw new NoSuchElementException("Iterator is exhausted");
59              }
60              return a[i++];
61          }
62
63          public void remove() {
64              throw new UnsupportedOperationException("remove() is not supported");
65          }
66      }
67
68      public static void main(String[] args) {
69          ResizingArrayBag<String> bag = new ResizingArrayBag<String>();
70          while (!StdIn.isEmpty()) {
```

Bag · Resizing Array Bag

ResizingArrayBag.java

```
71              String item = StdIn.readString();
72              bag.add(item);
73          }
74          StdOut.println("size of bag = " + bag.size());
75          for (String s : bag) {
76              StdOut.print(s + " ");
77          }
78          StdOut.println();
79      }
80  }
```

Bag · Resizing Array Bag

Operation	$T(n)$
`ResizingArrayBag()`	1
`boolean isEmpty()`	1
`int size()`	1
`void add(Item item)`	1 (amortized)
`Iterator<Item> iterator()`	1

Queue

A `Queue` is an iterable collection that stores generic items in first-in-first-out (FIFO) order

Queue<Item> extends Iterable<Item>	
`boolean isEmpty()`	returns `true` if this queue is empty, and `false` otherwise
`int size()`	returns the number of items in this queue
`void enqueue(Item item)`	adds `item` to the end of this queue
`Item peek()`	returns the item at the front of this queue
`Item dequeue()`	removes and returns the item at the front of this queue
`Iterator<Item> iterator()`	returns an iterator to iterate over the items in this queue in FIFO order

Queue

Program: `KthFromLast.java`

- ⇝ Command-line input: k (int)
- ⇝ Standard input: a sequence of integers
- ⇝ Standard output: the kth integer from the end

```
>_ ~/workspace/dsa/programs

$ java KthFromLast 4
1 2 3 4 5 6 7 8 9 10
<ctrl-d>
7
$ _
```

Queue

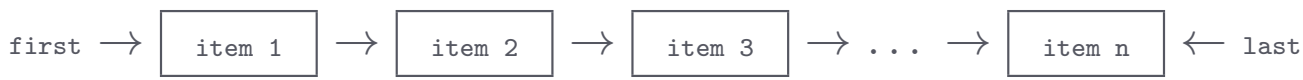
KthFromLast.java

```java
import dsa.LinkedQueue;

import stdlib.StdIn;
import stdlib.StdOut;

public class KthFromLast {
    public static void main(String[] args) {
        int k = Integer.parseInt(args[0]);
        LinkedQueue<Integer> queue = new LinkedQueue<Integer>();
        while (!StdIn.isEmpty()) {
            queue.enqueue(StdIn.readInt());
        }
        int n = queue.size();
        for (int i = 1; i <= n - k; i++) {
            queue.dequeue();
        }
        StdOut.println(queue.peek());
    }
}
```

Queue · Linked Queue

Instance variables:

- ⇝ References to the front and back of the queue: `Node first` and `Node last`



- ⇝ Number of items in the queue: `int n`

Queue · Linked Queue

LinkedQueue.java

```java
package dsa;

import java.util.Iterator;
import java.util.NoSuchElementException;

import stdlib.StdIn;
import stdlib.StdOut;

public class LinkedQueue<Item> implements Queue<Item> {
    private Node first;
    private Node last;
    private int n;

    public LinkedQueue() {
        first = null;
        last = null;
        n = 0;
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public int size() {
        return n;
    }

    public void enqueue(Item item) {
        Node oldlast = last;
        last = new Node();
        last.item = item;
        last.next = null;
        if (isEmpty()) {
            first = last;
        } else {
```

Queue · Linked Queue

LinkedQueue.java

```
36              oldlast.next = last;
37          }
38          n++;
39      }
40
41      public Item peek() {
42          if (isEmpty()) {
43              throw new NoSuchElementException("Queue is empty");
44          }
45          return first.item;
46      }
47
48      public Item dequeue() {
49          if (isEmpty()) {
50              throw new NoSuchElementException("Queue is empty");
51          }
52          Item item = first.item;
53          first = first.next;
54          n--;
55          if (isEmpty()) {
56              last = null;
57          }
58          return item;
59      }
60
61      public Iterator<Item> iterator() {
62          return new ListIterator();
63      }
64
65      private class Node {
66          private Item item;
67          private Node next;
68      }
69
70      private class ListIterator implements Iterator<Item> {
```

LinkedQueue.java

```
71          private Node current;
72
73          public ListIterator() {
74              current = first;
75          }
76
77          public boolean hasNext() {
78              return current != null;
79          }
80
81          public Item next() {
82              if (!hasNext()) {
83                  throw new NoSuchElementException("Iterator is exhausted");
84              }
85              Item item = current.item;
86              current = current.next;
87              return item;
88          }
89
90          public void remove() {
91              throw new UnsupportedOperationException("remove() is not supported");
92          }
93      }
94
95      public static void main(String[] args) {
96          LinkedQueue<String> queue = new LinkedQueue<String>();
97          while (!StdIn.isEmpty()) {
98              String item = StdIn.readString();
99              if (!item.equals("-")) {
100                 queue.enqueue(item);
101             } else if (!queue.isEmpty()) {
102                 StdOut.print(queue.dequeue() + " ");
103             }
104         }
105         StdOut.println("(" + queue.size() + " left on queue)");
```

Queue · Linked Queue

LinkedQueue.java

```
106         }
107     }
```

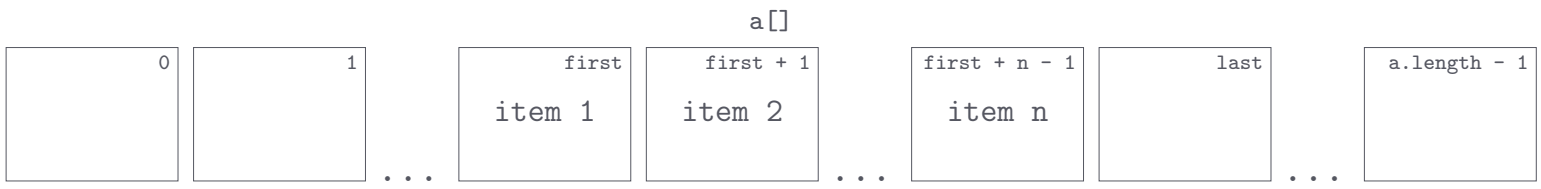
Queue · Linked Queue

Operation	$T(n)$
`LinkedQueue()`	1
`boolean isEmpty()`	1
`int size()`	1
`void enqueue(Item item)`	1
`Item peek()`	1
`Item dequeue()`	1
`Iterator<Item> iterator()`	1

Queue · Resizing Array Queue

Instance variables:

- ⇝ Array of items in the queue: `Item a[]`



- ⇝ Index of the first item: `int first`
- ⇝ Index of the next new item: `int last`
- ⇝ Number of items in the queue: `int n`

Queue · Resizing Array Queue

ResizingArrayQueue.java

```
1  package dsa;
2
3  import java.util.Iterator;
4  import java.util.NoSuchElementException;
5
6  import stdlib.StdIn;
7  import stdlib.StdOut;
8
9  public class ResizingArrayQueue<Item> implements Queue<Item> {
10     private Item[] a;
11     private int first;
12     private int last;
13     private int n;
14
15     public ResizingArrayQueue() {
16         a = (Item[]) new Object[2];
17         n = 0;
18         first = 0;
19         last = 0;
20     }
21
22     public boolean isEmpty() {
23         return n == 0;
24     }
25
26     public int size() {
27         return n;
28     }
29
30     public void enqueue(Item item) {
31         if (n == a.length) {
32             resize(2 * a.length);
33         }
34         a[last++] = item;
35         if (last == a.length) {
```

Queue · Resizing Array Queue

ResizingArrayQueue.java

```
36              last = 0;
37          }
38          n++;
39      }
40
41      public Item peek() {
42          if (isEmpty()) {
43              throw new NoSuchElementException("Queue is empty");
44          }
45          return a[first];
46      }
47
48      public Item dequeue() {
49          if (isEmpty()) {
50              throw new NoSuchElementException("Queue is empty");
51          }
52          Item item = a[first];
53          a[first] = null;
54          n--;
55          first++;
56          if (first == a.length) {
57              first = 0;
58          }
59          if (n > 0 && n == a.length / 4) {
60              resize(a.length / 2);
61          }
62          return item;
63      }
64
65      public Iterator<Item> iterator() {
66          return new ArrayIterator();
67      }
68
69      private void resize(int capacity) {
70          Item[] temp = (Item[]) new Object[capacity];
```

ResizingArrayQueue.java

```
71          for (int i = 0; i < n; i++) {
72              temp[i] = a[(first + i) % a.length];
73          }
74          a = temp;
75          first = 0;
76          last = n;
77      }
78
79      private class ArrayIterator implements Iterator<Item> {
80          private int i;
81
82          public ArrayIterator() {
83              i = 0;
84          }
85
86          public boolean hasNext() {
87              return i < n;
88          }
89
90          public Item next() {
91              if (!hasNext()) {
92                  throw new NoSuchElementException("Iterator is exhausted");
93              }
94              Item item = a[(i + first) % a.length];
95              i++;
96              return item;
97          }
98
99          public void remove() {
100             throw new UnsupportedOperationException("remove() is not supported");
101         }
102     }
103
104     public static void main(String[] args) {
105         ResizingArrayQueue<String> queue = new ResizingArrayQueue<String>();
```

Queue · Resizing Array Queue

ResizingArrayQueue.java

```
106         while (!StdIn.isEmpty()) {
107             String item = StdIn.readString();
108             if (!item.equals("-")) {
109                 queue.enqueue(item);
110             } else if (!queue.isEmpty()) {
111                 StdOut.print(queue.dequeue() + " ");
112             }
113         }
114         StdOut.println("(" + queue.size() + " left on queue)");
115     }
116 }
```

Queue · Resizing Array Queue

Operation	$T(n)$
`ResizingArrayQueue()`	1
`boolean isEmpty()`	1
`int size()`	1
`void enqueue(Item item)`	1 (amortized)
`Item peek()`	1
`Item dequeue()`	1 (amortized)
`Iterator<Item> iterator()`	1

Stack

A `Stack` is an iterable collection that stores generic items in last-in-first-out (LIFO) order

Stack<Item> extends Iterable<Item>	
`boolean isEmpty()`	returns `true` if this stack is empty, and `false` otherwise
`int size()`	returns the number of items in this stack
`void push(Item item)`	adds `item` to the top of this stack
`Item peek()`	returns the item at the top of this stack
`Item pop()`	removes and returns the item at the top of this stack
`Iterator<Item> iterator()`	returns an iterator to iterate over the items in this stack in LIFO order

Stack

Program: `Reverse.java`

- ⇝ Standard input: a sequence of strings
- ⇝ Standard output: the strings in reverse order

```
>_ ~/workspace/dsa/programs
```

```
$ java Reverse
b o l t o n
<ctrl-d>
n o t l o b
$ _
```

Stack

Reverse.java

```java
import dsa.LinkedStack;

import stdlib.StdIn;
import stdlib.StdOut;

public class Reverse {
    public static void main(String[] args) {
        LinkedStack<String> stack = new LinkedStack<String>();
        while (!StdIn.isEmpty()) {
            String s = StdIn.readString();
            stack.push(s);
        }
        for (String s : stack) {
            StdOut.print(s + " ");
        }
        StdOut.println();
    }
}
```

Stack · Linked Stack

Instance variables:

- ⇝ Reference to the top of the stack: `Node first`

- ⇝ Number of items in the stack: `int n`

Stack · Linked Stack

LinkedStack.java

```java
package dsa;

import java.util.Iterator;
import java.util.NoSuchElementException;

import stdlib.StdIn;
import stdlib.StdOut;

public class LinkedStack<Item> implements Stack<Item> {
    private Node first;
    private int n;

    public LinkedStack() {
        first = null;
        n = 0;
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public int size() {
        return n;
    }

    public void push(Item item) {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
        n++;
    }

    public Item peek() {
        if (isEmpty()) {
```

Stack · Linked Stack

LinkedStack.java

```
36              throw new NoSuchElementException("Stack is empty");
37          }
38          return first.item;
39      }
40
41      public Item pop() {
42          if (isEmpty()) {
43              throw new NoSuchElementException("Stack is empty");
44          }
45          Item item = first.item;
46          first = first.next;
47          n--;
48          return item;
49      }
50
51      public Iterator<Item> iterator() {
52          return new ListIterator();
53      }
54
55      private class Node {
56          private Item item;
57          private Node next;
58      }
59
60      private class ListIterator implements Iterator<Item> {
61          private Node current;
62
63          public ListIterator() {
64              current = first;
65          }
66
67          public boolean hasNext() {
68              return current != null;
69          }
70
```

Stack · Linked Stack

LinkedStack.java

```
71          public Item next() {
72              if (!hasNext()) {
73                  throw new NoSuchElementException("Iterator is exhausted");
74              }
75              Item item = current.item;
76              current = current.next;
77              return item;
78          }
79
80          public void remove() {
81              throw new UnsupportedOperationException("remove() is not supported");
82          }
83      }
84
85      public static void main(String[] args) {
86          LinkedStack<String> stack = new LinkedStack<String>();
87          while (!StdIn.isEmpty()) {
88              String item = StdIn.readString();
89              if (!item.equals("-")) {
90                  stack.push(item);
91              } else if (!stack.isEmpty()) {
92                  StdOut.print(stack.pop() + " ");
93              }
94          }
95          StdOut.println("(" + stack.size() + " left on stack)");
96      }
97  }
```

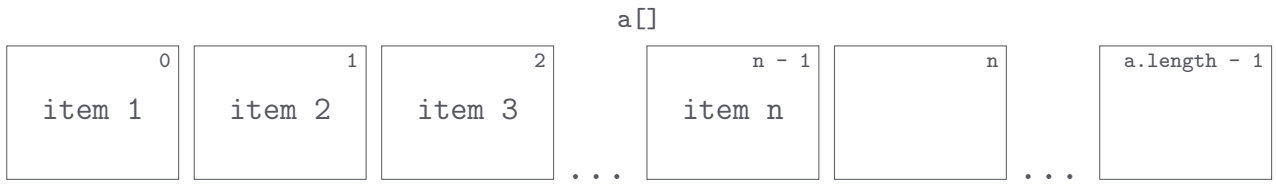
Stack · Linked Stack

Operation	$T(n)$
`LinkedStack()`	1
`boolean isEmpty()`	1
`int size()`	1
`void push(Item item)`	1
`Item peek()`	1
`Item pop()`	1
`Iterator<Item> iterator()`	1

Stack · Resizing Array Stack

Instance variables:

- ⇝ Array of items in the stack: `Item a[]`



- ⇝ Number of items in the stack: `int n`

Stack · Resizing Array Stack

ResizingArrayStack.java

```java
package dsa;

import java.util.Iterator;
import java.util.NoSuchElementException;

import stdlib.StdIn;
import stdlib.StdOut;

public class ResizingArrayStack<Item> implements Stack<Item> {
    private Item[] a;
    private int n;

    public ResizingArrayStack() {
        a = (Item[]) new Object[2];
        n = 0;
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public int size() {
        return n;
    }

    public void push(Item item) {
        if (n == a.length) {
            resize(2 * a.length);
        }
        a[n++] = item;
    }

    public Item peek() {
        if (isEmpty()) {
            throw new NoSuchElementException("Stack is empty");
```

Stack · Resizing Array Stack

ResizingArrayStack.java

```
        }
        return a[n - 1];
    }

    public Item pop() {
        if (isEmpty()) {
            throw new NoSuchElementException("Stack is empty");
        }
        Item item = a[n - 1];
        a[n - 1] = null;
        n--;
        if (n > 0 && n == a.length / 4) {
            resize(a.length / 2);
        }
        return item;
    }

    public Iterator<Item> iterator() {
        return new ReverseArrayIterator();
    }

    private void resize(int capacity) {
        Item[] temp = (Item[]) new Object[capacity];
        for (int i = 0; i < n; i++) {
            temp[i] = a[i];
        }
        a = temp;
    }

    private class ReverseArrayIterator implements Iterator<Item> {
        private int i;

        public ReverseArrayIterator() {
            i = n - 1;
        }
```

Stack · Resizing Array Stack

ResizingArrayStack.java

```
71
72          public boolean hasNext() {
73              return i >= 0;
74          }
75
76          public Item next() {
77              if (!hasNext()) {
78                  throw new NoSuchElementException("Iterator is exhausted");
79              }
80              return a[i--];
81          }
82
83          public void remove() {
84              throw new UnsupportedOperationException("remove() is not supported");
85          }
86      }
87
88      public static void main(String[] args) {
89          ResizingArrayStack<String> stack = new ResizingArrayStack<String>();
90          while (!StdIn.isEmpty()) {
91              String item = StdIn.readString();
92              if (!item.equals("-")) {
93                  stack.push(item);
94              } else if (!stack.isEmpty()) {
95                  StdOut.print(stack.pop() + " ");
96              }
97          }
98          StdOut.println("(" + stack.size() + " left on stack)");
99      }
100 }
```

Stack · Resizing Array Stack

Operation	$T(n)$
`ResizingArrayStack()`	1
`boolean isEmpty()`	1
`int size()`	1
`void push(Item item)`	1 (amortized)
`Item peek()`	1
`Item pop()`	1 (amortized)
`Iterator<Item> iterator()`	1