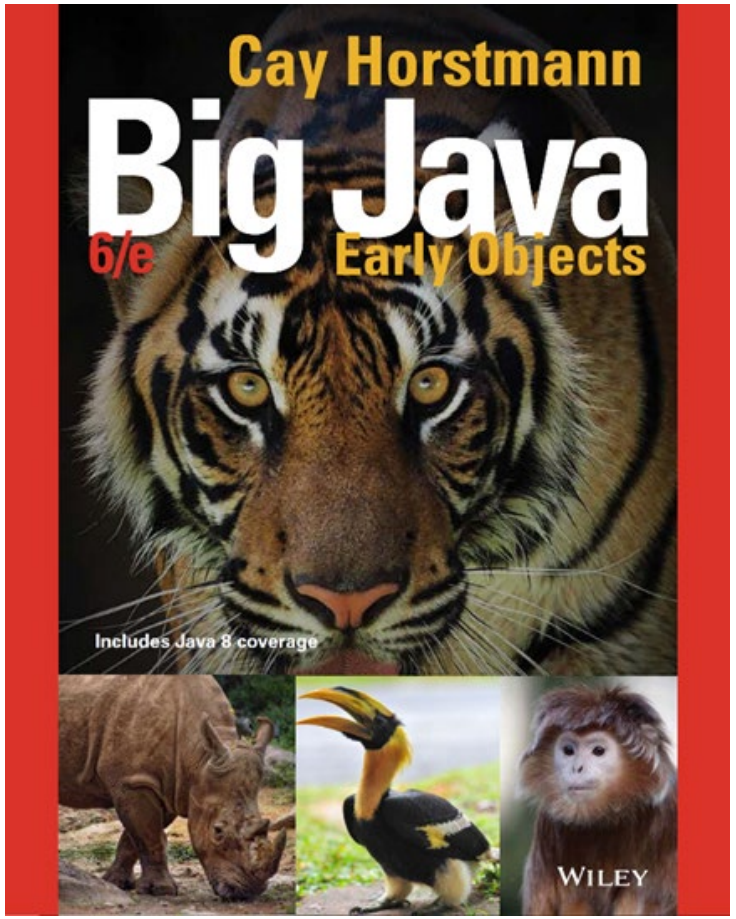

Big Java Early Objects by Cay Horstmann

Chapter 16 – Part III Implementing Array Lists, Stacks and Queues

Implementing Array Lists

- An array list maintains a reference to an array of elements.
- The array is large enough to hold all elements in the collection.
- When the array gets full, it is replaced by a larger one.
- An array list has an instance field that stores the current number of elements.

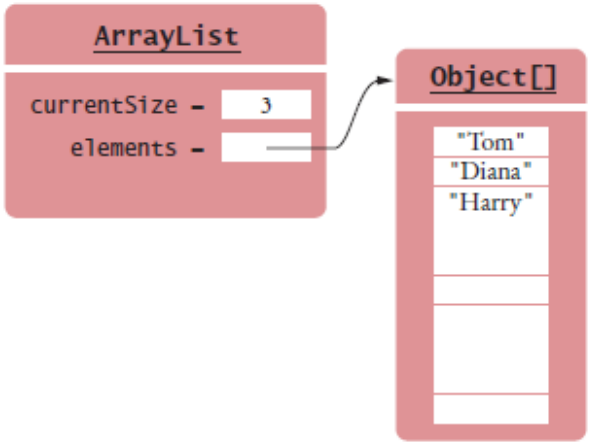

Figure 7 An Array List Stores its Elements in an Array

Implementing Array Lists

- Our `ArrayList` implementation will manage elements of type `Object`:

```
public class ArrayList
{
   private Object[] elements;
   private int currentSize;

   public ArrayList()
   {
      final int INITIAL_SIZE = 10;
      elements = new Object[INITIAL_SIZE];
      currentSize = 0;
   }

   public int size() { return currentSize; }
   . . .
}
```

Implementing Array Lists - Getting and Setting Elements

- Providing `get` and `set` methods: Check for valid positions
- Access the internal array at the given position
- Helper method to check bounds:

```
private void checkBounds(int n)
{
   if (n < 0 || n >= currentSize)
   {
      throw new IndexOutOfBoundsException();
   }
}
```

Implementing Array Lists - Getting and Setting Elements

- The `get` method:

```
public Object get(int pos)
{
   checkBounds(pos);
   return element[pos];
}
```

- The `set` method:

```
public void set(int pos, Object element)
{
   checkBounds(pos);
   elements[pos] = element;
}
```

- Getting and setting an element can be carried out with a bounded set of instructions, independent of the size of the array list.
- These are *O*(1) operations.

Removing or Adding Elements

- To remove an element at position *k*, move the elements with higher index values.
- The `remove` method:

```
public Object remove(int pos)
{
   checkBounds(pos);
   Object removed = elements[pos];
   for (int i = pos + 1; i < currentSize; i++)
   {
      elements[i - 1] = elements[i];
   }
   currentSize--;
   return removed;
}
```

- On average, *n* / 2 elements need to move.

Removing or Adding Elements

- Inserting a element also requires moving, on average, n /2 elements.
- Inserting or removing an array list element is an $O(n)$ operation.

Removing or Adding Elements

- Exception: adding an element after the last element
  - Store the element in the array
  - Increment size
- An *O*(1) operation
- The `addLast` method:

```
public boolean addLast(Object newElement)
{
   growIfNecessary();
   currentSize++;
   elements[currentSize - 1] = newElement;
   return true;
}
```

Removing or Adding Elements

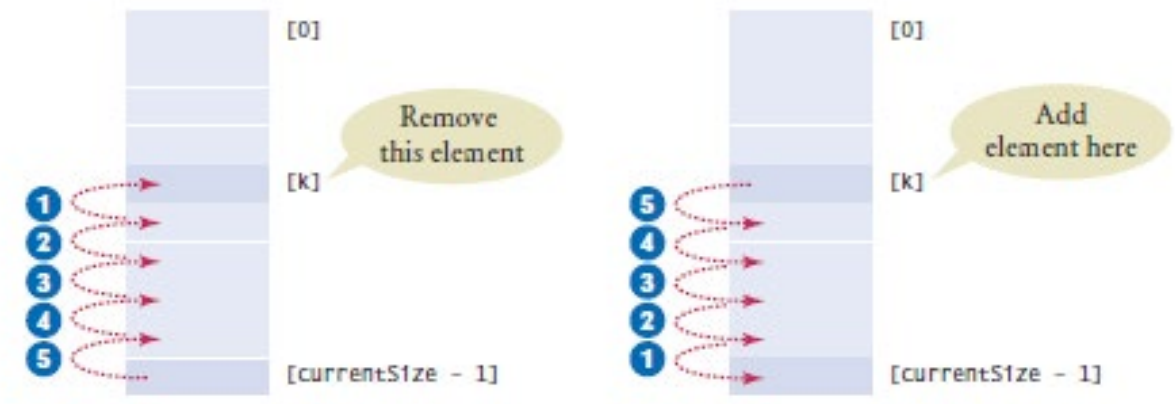

Figure 8 Removing and Adding Elements

Growing the Internal Array

When an array list is completely full, we must move the contents to a larger array.

Growing the Internal Array

- When the array is full: Create a bigger array
- Copy the elements to the new array
- New array replaces old
- Reallocation is $O(n)$.

Growing the Internal Array

- The `growIfNecessary` method:

```
private void growIfNecessary()
{
   if (currentSize == elements.length)
   {
      Object[] newElements =
         new Object[2 * elements.length];
      for (int i = 0; i < elements.length; i++)
      {
         newElements[i] = elements[i];
      }
      elements = newElements;
   }
}
```

1
2
3

Growing the Internal Array

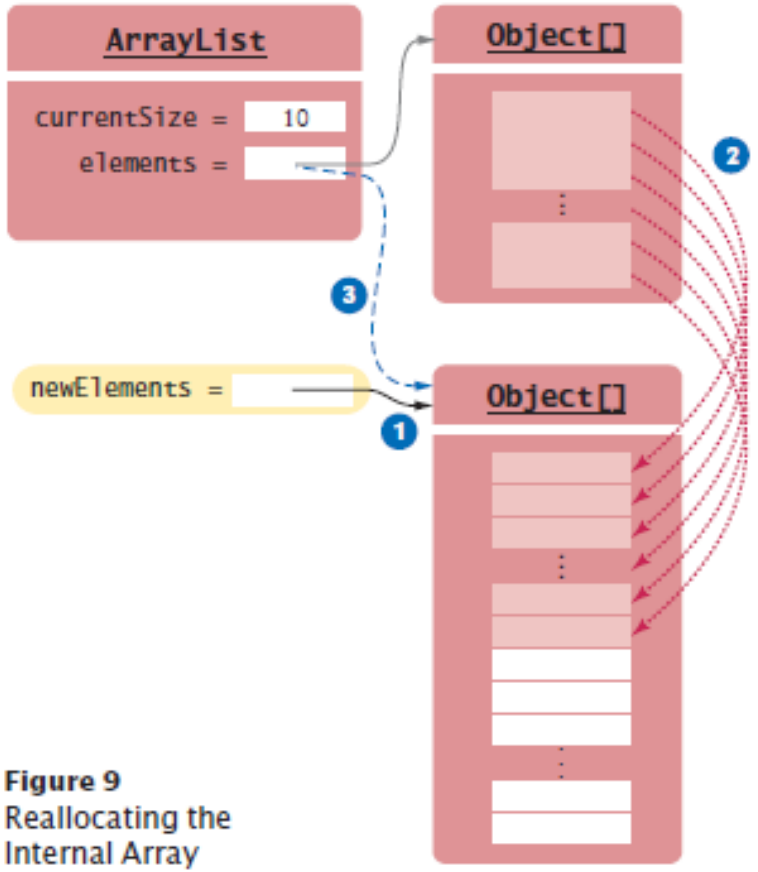

Figure 9
Reallocating the Internal Array

Implementing Stacks and Queues

- Stacks and queues are abstract data types.
- We specify how operations must behave.
- We do not specify the implementation.
- Many different implementations are possible.

Stacks as Linked Lists

- A stack can be implemented as a sequence of nodes.
- New elements are “pushed” to one end of the sequence, and they are “popped” from the same end.
- Push and pop from the least expensive end - the front.
- The `push` and `pop` operations are identical to the `addFirst` and `removeFirst` operations of the linked list.

Stacks as Linked Lists

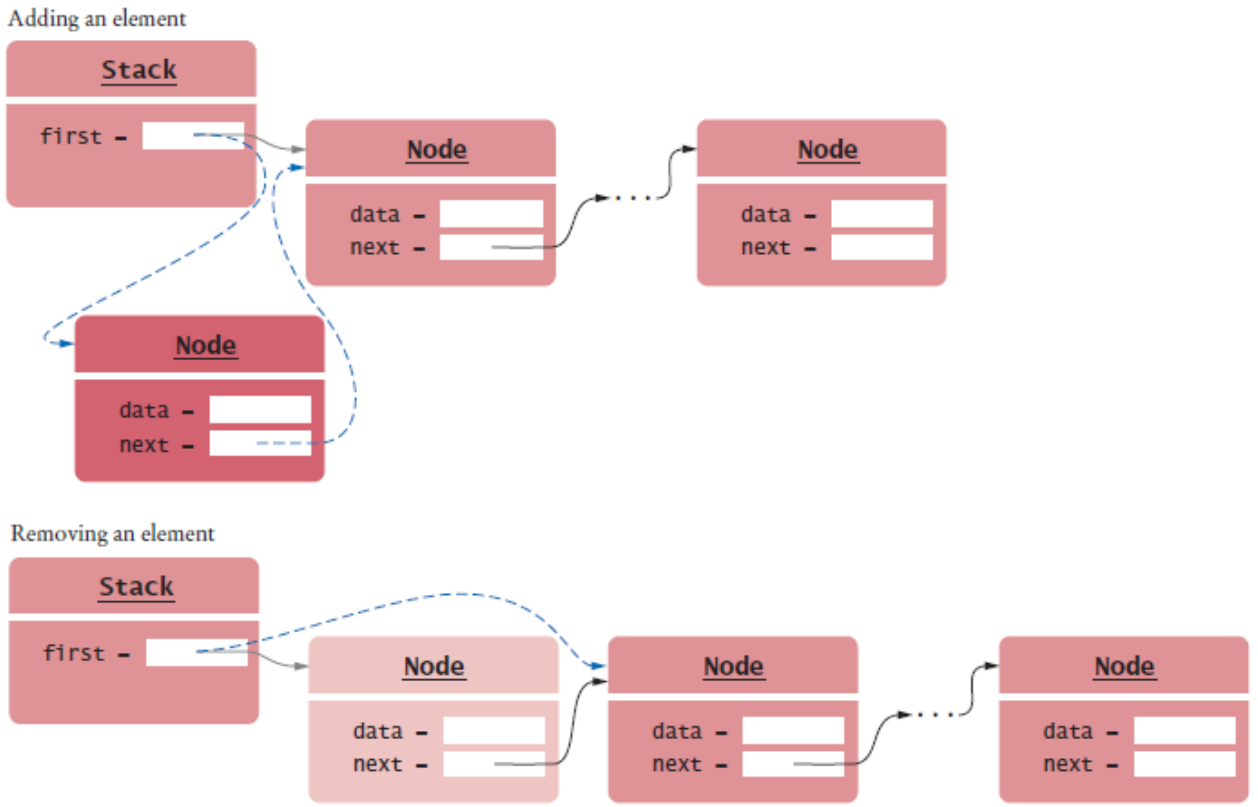

Figure 10 Push and Pop for a Stack Implemented as a Linked List

Stacks as Arrays

- A stack can be implemented as an array.
- Push and pop from the least expensive end - the back.
- The array must grow when it gets full.
- The `push` and `pop` operations are identical to the `addLast` and `removeLast` operations of an array list.
- `push` and `pop` are *O*(1)+ operations.

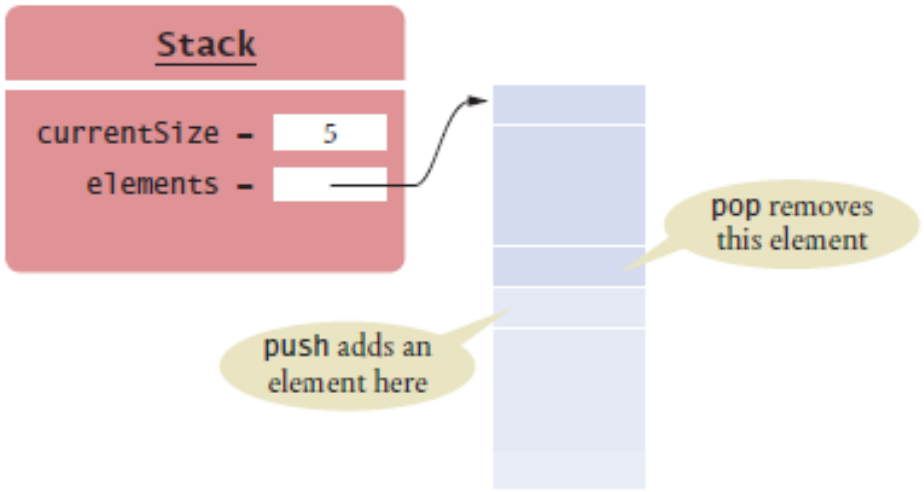

Figure 11 A Stack Implemented as an Array

Queues as Linked Lists

- A queue can be implemented as a linked list:
  - Add elements at the back
  - Remove elements at the front.
  - Keep a reference to last element
- The `add` and `remove` operations are $O(1)$ operations.

Queues as Linked Lists

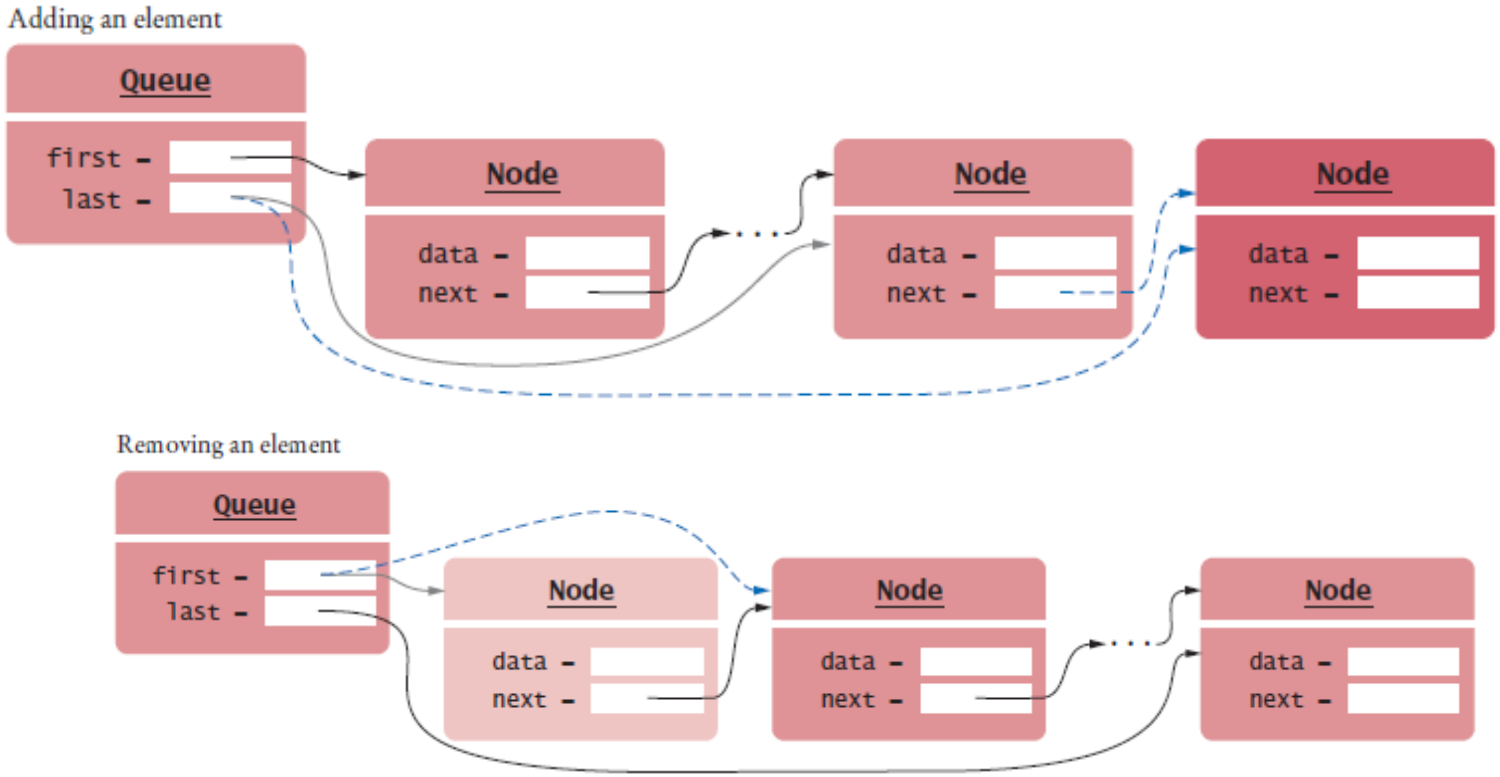

Figure 12 A Queue Implemented as a Linked List

section_3_1/LinkedListStack.java

```
1  import java.util.NoSuchElementException;
2
3  /**
4     An implementation of a stack as a sequence of nodes.
5  */
6  public class LinkedListStack
7  {
8     private Node first;
9
10    /**
11       Constructs an empty stack.
12    */
13    public LinkedListStack()
14    {
15       first = null;
16    }
17
18    /**
19       Adds an element to the top of the stack.
20       @param element the element to add
21    */
22    public void push(Object element)
23    {
24       Node newNode = new Node();
25       newNode.data = element;
26       newNode.next = first;
27       first = newNode;
28    }
29
```

Continued

section_3_1/LinkedListStack.java

```
30      /**
31         Removes the element from the top of the stack.
32         @return the removed element
33      */
34      public Object pop()
35      {
36         if (first == null) { throw new NoSuchElementException(); }
37         Object element = first.data;
38         first = first.next;
39         return element;
40      }
41
42      /**
43         Checks whether this stack is empty.
44         @return true if the stack is empty
45      */
46      public boolean empty()
47      {
48         return first == null;
49      }
50
51      class Node
52      {
53         public Object data;
54         public Node next;
55      }
56   }
```