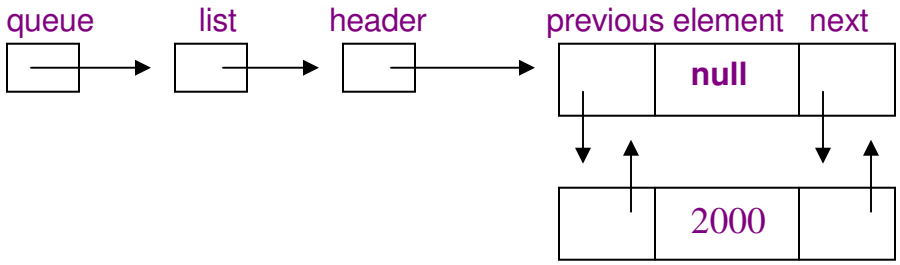
8.2 Suppose we define:

```
PureQueue<Integer> queue = new LinkedListPureQueue<Integer>();
```

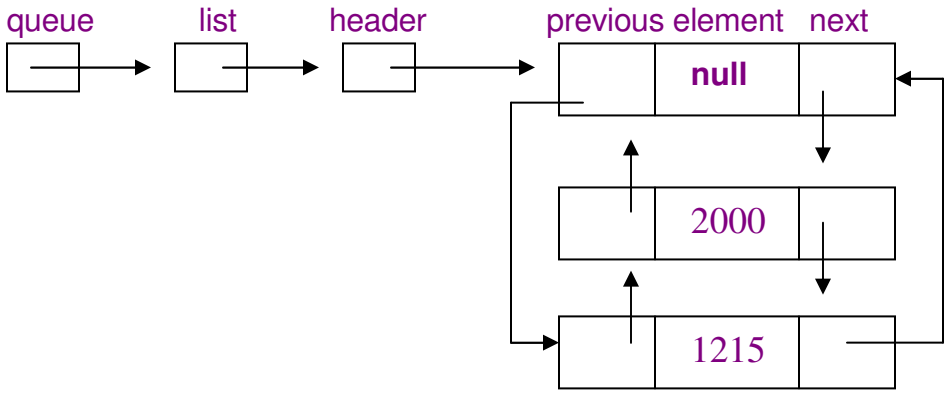
Show what the LinkedList object (referenced by) queue will look like after each of the following messages is sent:

a. queue.enqueue (2000);

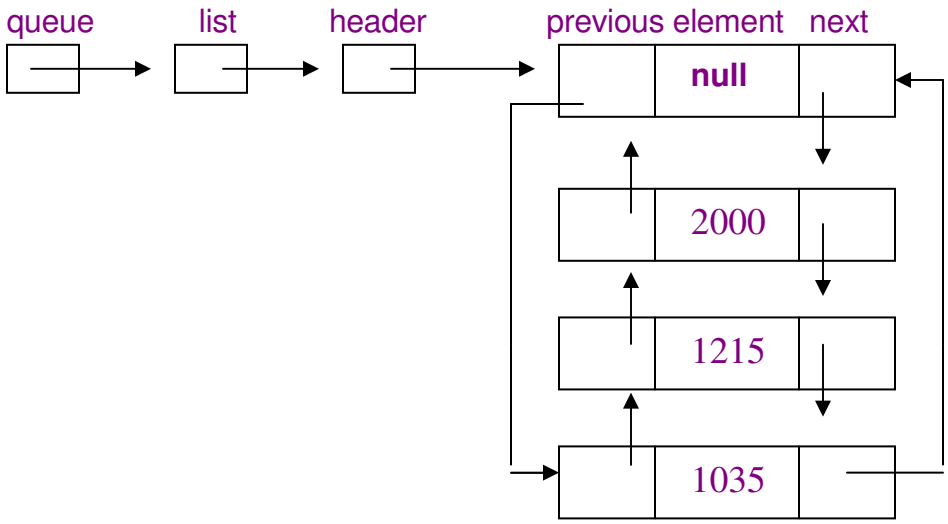


The element field in each entry is of type reference to Integer. For simplicity, we pretend that the element field is of type **int**.

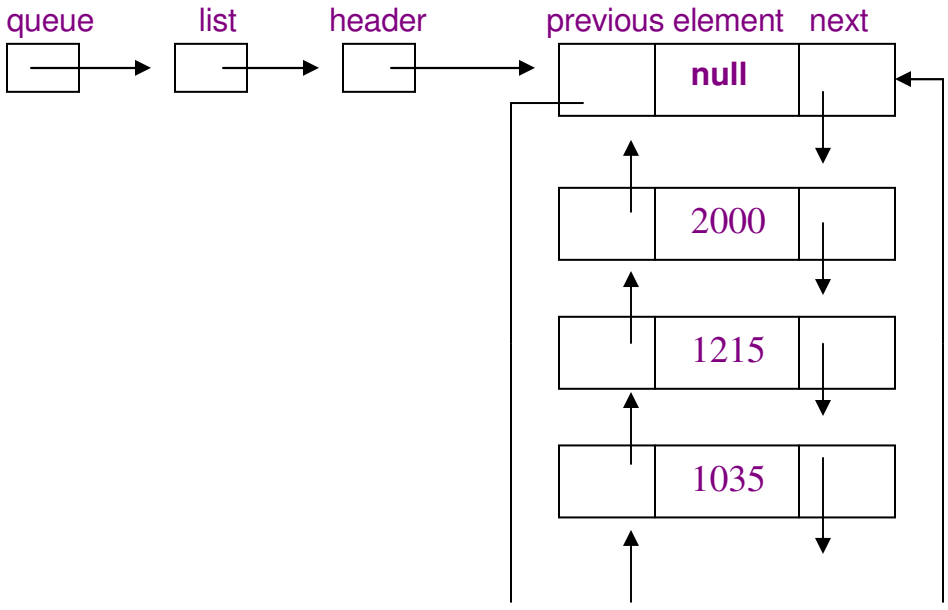
b. queue.enqueue (1215);



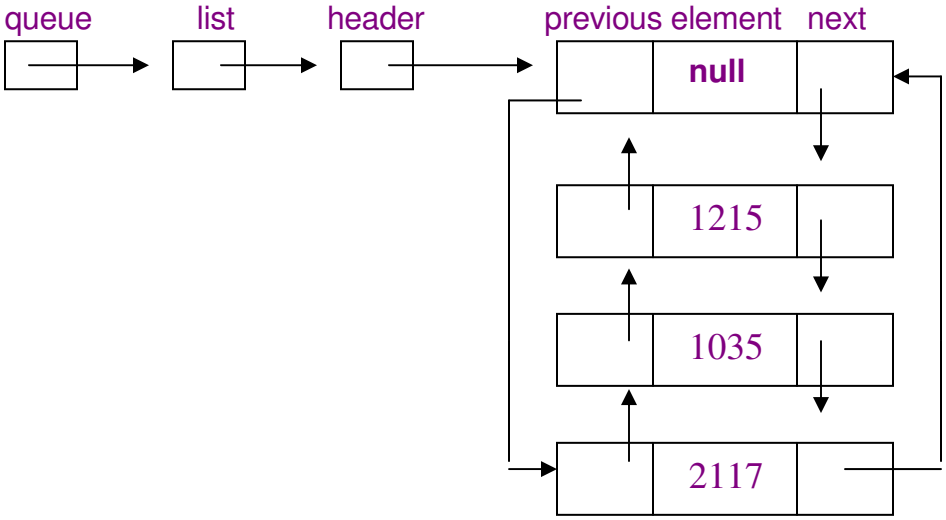
c. queue.enqueue (1035);



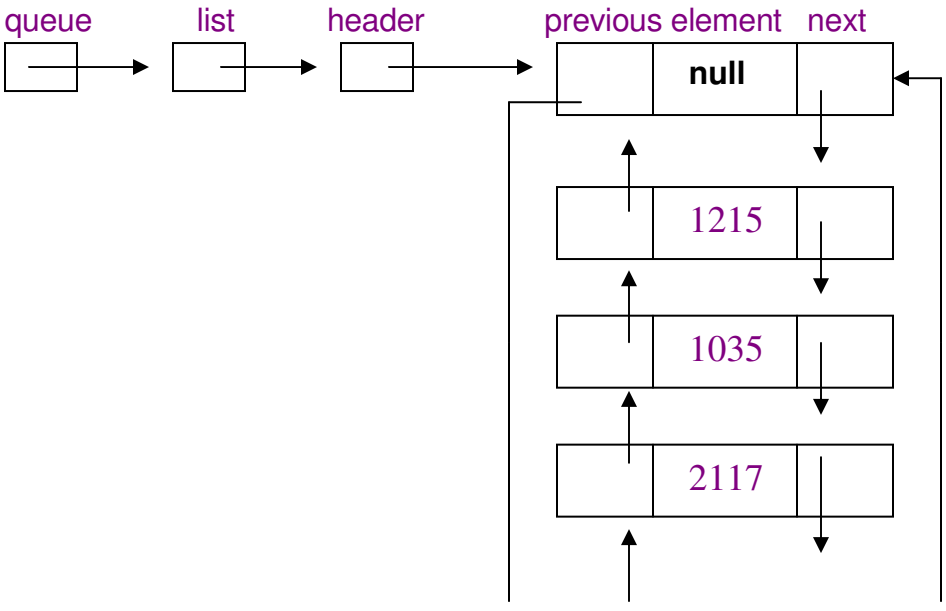
d. queue.enqueue (2117);



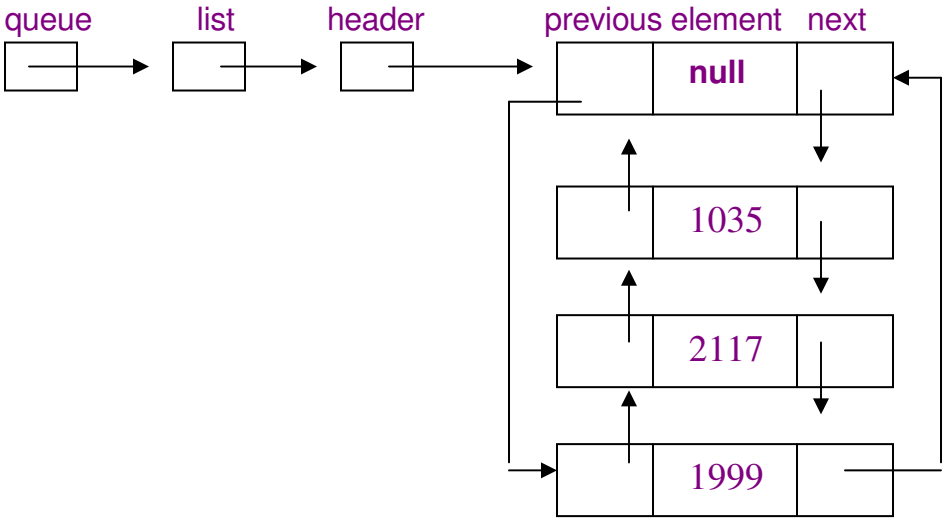
e. queue.dequeue();



f. queue.enqueue (1999);

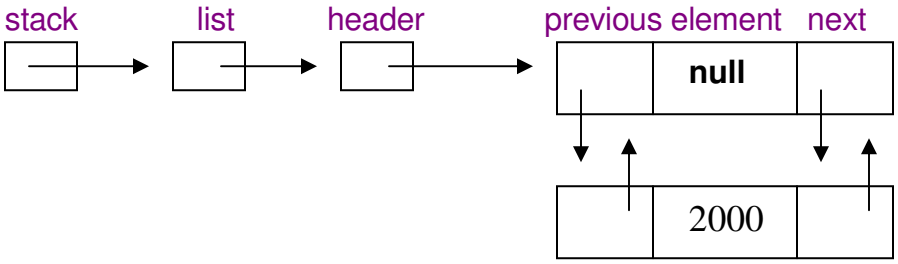


g. queue.dequeue();



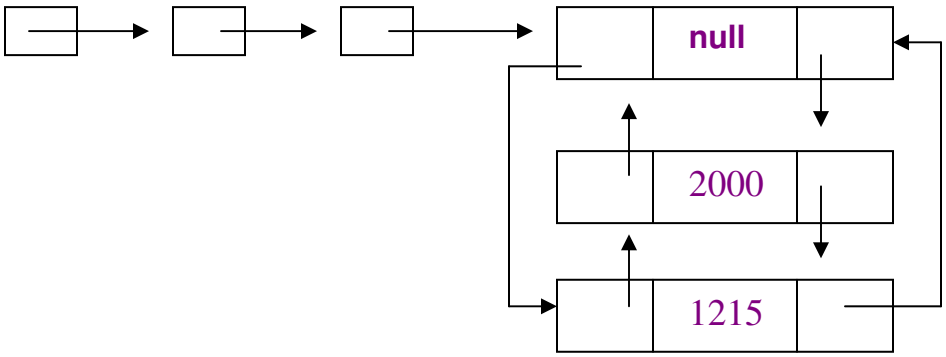
8.3 Re-do Exercise 8.2, parts a through g, for a stack instead of a queue.

a. stack.push (2000);

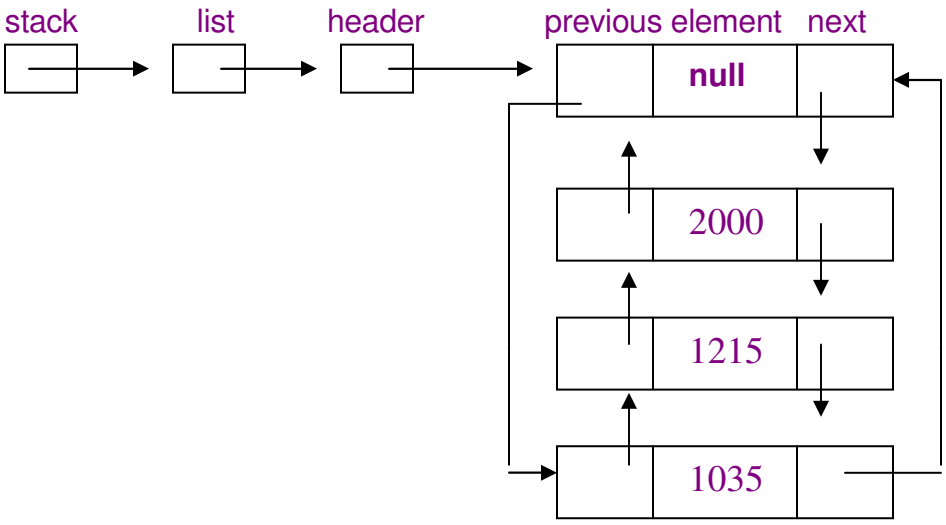


The element field in each entry is of type reference to Integer. For simplicity, we pretend that the element field is of type **int**.

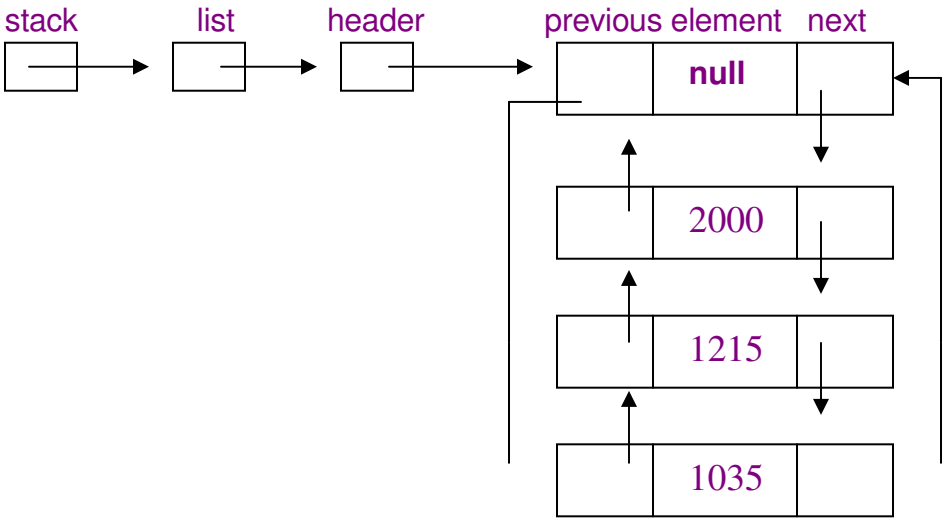
b. stack.push (1215);

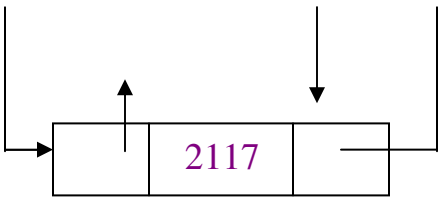


c. stack.push (1035);

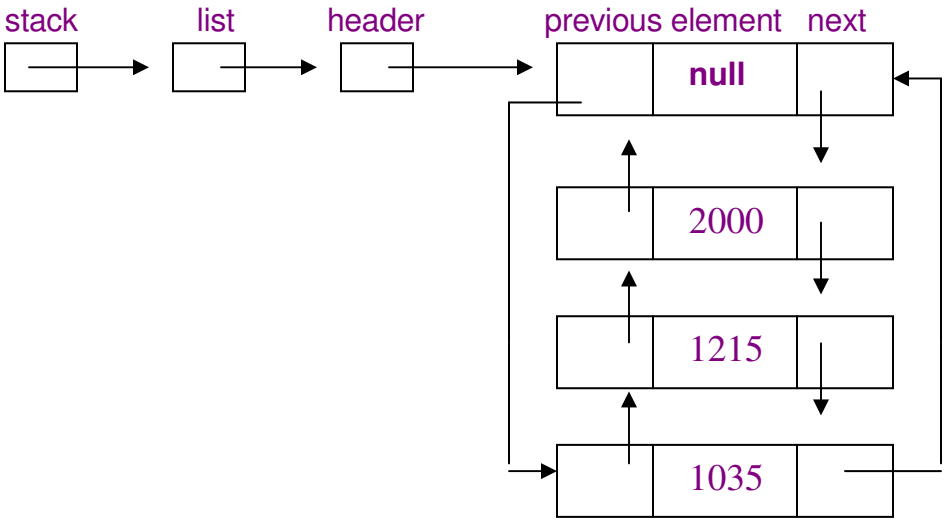


d. stack.push (2117);

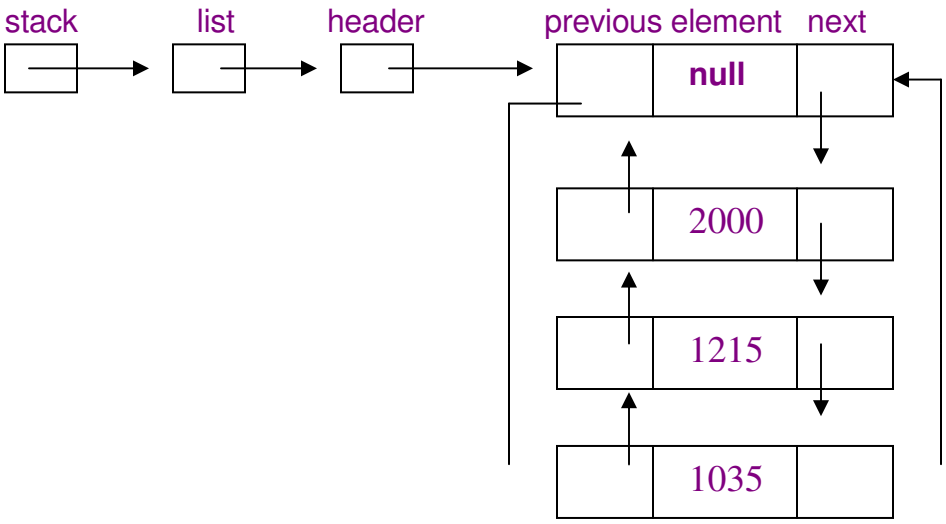


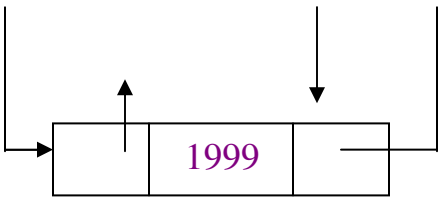


e. stack.pop();

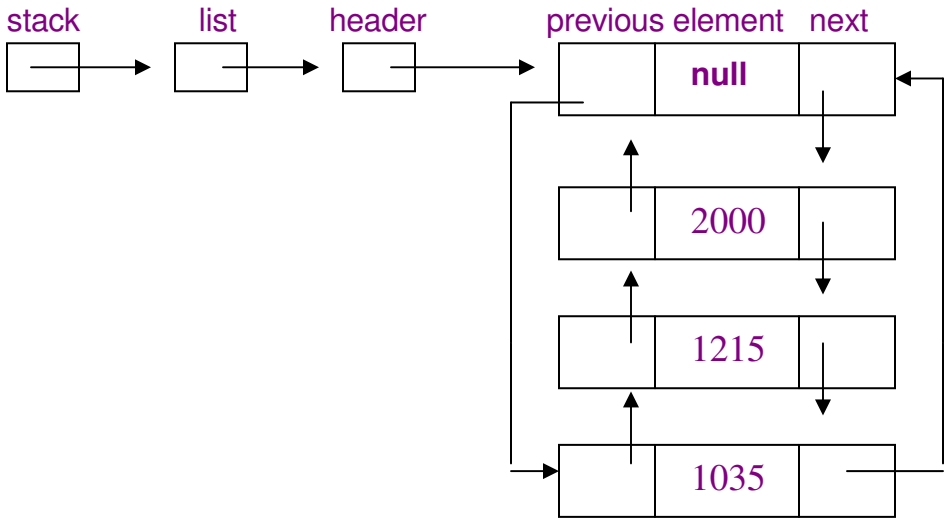


f. stack.push (1999);





g. stack.pop();



8.4 Suppose that elements "a", "b", "c", "d", "e" are pushed, in that order, onto an initially empty stack, which is then popped four times, and as each element is popped, it is enqueued into an initially empty queue. If one element is then dequeued from the queue, what is the *next* element to be dequeued?

The next element to be dequeued is "d".

8.5 The array-based implementation of the PureQueue interface had data, size, head and tail fields. Show that the size field is redundant. Specifically, show that the size() method can be implemented using only the data, head and tail fields.

```
public int size( )
{
        if (tail == – 1 || head <= tail)
                return tail – head + 1;
```

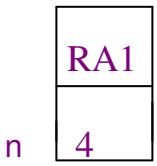
```
        return data.length + tail – head + 1;
} // method size
```

8.6 Use a stack of activation records to trace the execution of the recursive factorial method after an initial call of factorial (4). Here is the method definition:

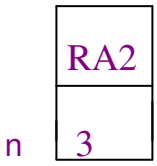
```
/**
 * Calculates the factorial of a non-negative integer, that is, the product of
 *  all integers between 1 and the given integer, inclusive.
 *  The worstTime(n) is O(n).
 *
 *  @param n the non-negative integer whose factorial is calculated.
 *
 *  @return the factorial of n
 *
 *  @throws IllegalArgumentException if n is less than 0.
 *
 */
public static long factorial (int n)
{
      if (n < 0)
            throw new IllegalArgumentException( );
      if (n <= 1)
            return 1;
      return n * factorial (n - 1);
} // method factorial
```

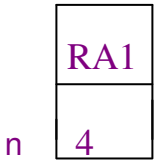
Let RA1 be the address of the instruction after the initial call, and RA2 be the address of the instruction after the recursive call.

Each activation record has two components: the return address and the value of the parameter n.

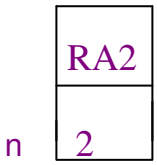


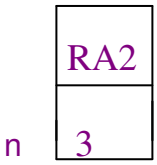
Activation Stack

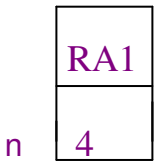




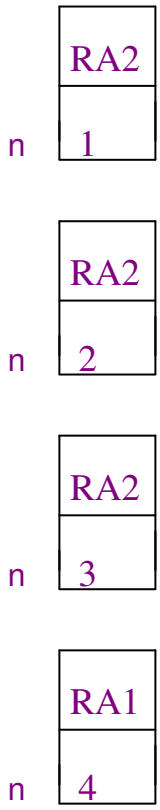
Activation Stack



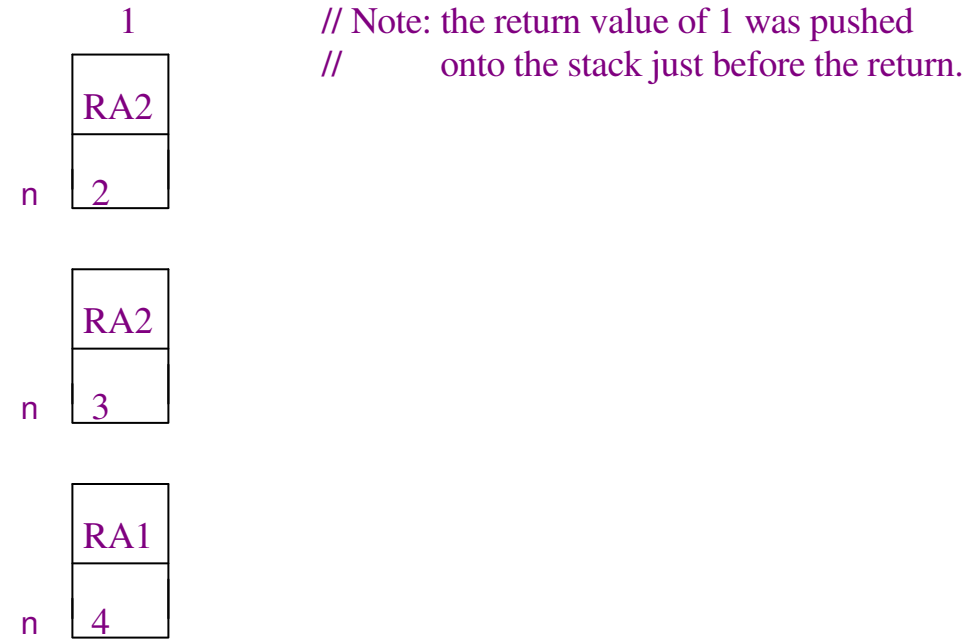




Activation Stack



Activation Stack


1
// Note: the return value of 1 was pushed
// onto the stack just before the return.
RA2
n
2
RA2
n
3
RA1
n
4
Activation Stack
2
RA2
n
3
RA1
n
4
Activation Stack
6
RA1
n
4
Activation Stack

The value now on top of the stack, 24, is the value of factorial (4).

Programming Exercises

8.3 Define an implementation of the PureStack class based on an ArrayList.

Hint: In the ArrayListPureStack class, the top of the stack is at index size() – 1.

```
import java.util.*;

public class ArrayListPureStack<E> implements PureStack<E>
{
        protected ArrayList<E> list;


        /**
         * Initializes this ArrayListPureStack object to have no elements.
         */
        public ArrayListPureStack()
        {
                list = new ArrayList<E>();
        } // default constructor


        /**
         * Initializes this ArrayListPureStack object to contain a shallow
         * copy of otherStack.
         *
         * @param otherStack - a ArrayListPureStack from which the
         *              calling object will be initialized.
         */
        public ArrayListPureStack(ArrayListPureStack<E> otherStack)
        {
                list = new ArrayList<E> (otherStack.list);
        } // default constructor


        /**
         * Returns the number of elements in this ArrayListPureStack
         * object.
         *
         * @return an int containing the number of elements in the
         *              ArrayListPureStack.
         *
```

```
 */
public int size()
{
        return list.size();
} // method size


/**
 * Returns true if this ArrayListPureStack object has no elements.
 * Otherwise, returns false.
 *
 *  @return a boolean indicating if the ArrayListPureStack is empty
 *              or not.
 *
 */
public boolean isEmpty()
{
        return list.isEmpty();
} // method isEmpty



/**
 * Inserts element at the top of this ArrayListPureStack object.
 * The averageTime(n) is constant and worstTime(n)is O(n).
 *
 * @param element a reference of type E to be pushed on the
 *       ArrayListPureStack.
 */
public void push (E element)
{
        list.add (element);
} // method push


/**
 * Removes (and returns) the element that was at the top of this
 * non-empty ArrayListPureStack object just before this method was
 * called.
 *
 * @return the element that was removed from the top of this
 *             ArrayListPureStack.
 *
 * @throws IndexOutOfBoundsException – if this
ArrayListPureStack
 *              object is empty.
```

```
         *
         */
        public E pop()
        {
                return list.remove (list.size() - 1);
        } // method pop


        /**
         * The element at the top of this non-empty ArrayListPureStack
         * object has been returned.
         *
         * @return the element that is the top element of the
         *         ArrayListPureStack.
         *
         * @throws IndexOutOfBoundsException – if this
         ArrayListPureStack
         *                 object is empty.
         *
         */
        public E peek()
        {
                return list.get (list.size() - 1);
        } // method peek

} // ArrayListPureStack class
```