

VECTOR

A vector is a data structure that implements a mapping from an unbounded range of integers to objects of a certain type.

I. Language Agnostic Overview

Vectors are similar to arrays – one the most common and certainly one of the simplest data structures used in computer science. Arrays are hampered, however, by their fixed size. A vector data structure solves this problem, by increasing the array's size automatically when it becomes full and another insert is requested. Vectors are used frequently in both research and production environments. As they are versatile, fast, and space-efficient, they are a smart choice for a wide range of applications. Vectors are simple data structures, so correctness is not especially challenging, but it is extremely important.

The source code for the Vector class was developed by Sun, as part of version 1.4 of the Java framework.

Implementations of vectors are typically required to satisfy the following constraints:

1. The underlying array can never be null.
2. The element count can never be negative.
3. The element count can never be greater than the size of the underlying array.
4. No element can have a negative index.

II. Java Specific Overview:

The Vector class in Java is implemented on top of an array, so it still provides very fast random access. When an insert is attempted, and the current array doesn't have sufficient capacity, the array size is either increased by a predetermined number, or if that number isn't set, the capacity is simply doubled. It should also be noted that the Java implementation is also generic to any object (and presumably any primitive).

There are two fields which are used in the invariants:

- `elementData` – This is the array of objects on top of which the Vector class is built. It stores the actual data which is accessed and modified by the various methods of the Vector class.
- `elementCount` – This is an integer which stores the count of elements in the array.

So, a given element would be stored at `elementData[elementCount-1]`.

ensureCapacity

- | |
|--|
| a. This method ensures there is enough room in the underlying array for <code>minCapacity</code> elements. If not, the array size is increased (either doubled, incremented by a set amount, or set to <code>minCapacity</code>) to an acceptable size. |
|--|

b. The preconditions verify that the underlying array has been properly initialized (constraint 1), the number of elements hasn't somehow been set to a negative value (2), and the number of elements isn't larger than the size of the underlying array (3). The postCondition verifies the same properties (1, 2 and 3) and that the underlying array can hold at least minCapacity elements.

```
c. @Assertion(@Case(
pre = "elementData !=null && 0<=elementCount && elementCount<=elementData.length",
post = "elementData !=null && 0<=elementCount && elementCount<=elementData.length &&
    minCapacity<=elementData.length"))
public void ensureCapacity(int minCapacity) {
    modCount++;
    ensureCapacityHelper(minCapacity);
}
```

indexOf

a. This method searches for the given value and then returns its associated key. It searches forward (key sizes increasing by one) from the given index.

b. The preconditions verify that the underlying array has been properly initialized (constraint 1), the number of elements hasn't somehow been set to a negative value (2), and the number of elements isn't larger than the size of the underlying array (3). It also checks that that a negative search position was not given (4) and that the underlying array is not larger than one element (?). The postconditions are that either -1 or a correct key which is larger than the index must be returned.

```
c. @Assertion(@Case(
pre = "elementData !=null && Region.arrayReg(this.elementData,0,1) && Region.setObjReg(elem,1)
    && 0<=elementCount && elementCount<=elementData.length && index >=0 &&
    elementData.length==1",
post = "$ret==-1 || ($ret>=index && elementData[$ret]==elem)")
public int indexOf(Object elem, int index) {
    if (elem == null) {
        for (int i = index ; i < elementCount ; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = index ; i < elementCount ; i++)
            if (elem==elementData[i])
                return i;
    }
    return -1;
}
```

}

lastIndexOf

a. This method is similar to `indexOf` in that it searches for a value and returns the key, however, it searches backward (key sizes decreasing by one) from the given index.

b. The preconditions of this method identical to those of `indexOf` (constraints 1, 2, 3 and 4). The postconditions are also similar, however there is one notable difference. Since `lastIndexOf` searches backwards rather than forwards, the returned key has to be smaller than the given index, instead of larger.

c. `@Assertion(@Case(
pre = "elementData !=null && Region.arrayReg(this.elementData,0,1) && Region.setObjReg(elem,1)
 && 0<=elementCount && elementCount<=elementData.length ",
post = "$ret==-1 || ($ret<=index && elementData[$ret]==elem)"))
public int lastIndexOf(Object elem, int index) {
 if (index >= elementCount)
 throw new IndexOutOfBoundsException(" >= ");
 if (elem == null) {
 for (int i = index; i >= 0; i--)
 if (elementData[i]==null)
 return i;
 } else {
 for (int i = index; i >= 0; i--)
 if (elem==elementData[i])
 return i;
 }
 return -1;
}`

removeElementAt

a. This method removes the element from the vector which is stored at the given index.

b. The preconditions verify that the underlying array has been properly initialized (constraint 1), the number of elements hasn't somehow been set to a negative value (2), and the number of elements isn't larger than the size of the underlying array (3). There are no postconditions.

c. `@Assertion(@Case(
pre = "elementData !=null && 0<=elementCount && elementCount<=elementData.length ",
post = "true"))
public void removeElementAt(int index) {`

```

modCount++;
if (index >= elementCount) {
    throw new ArrayIndexOutOfBoundsException(" >= ");
} else if (index < 0) {
    throw new ArrayIndexOutOfBoundsException("<0");
}
int j = elementCount - index - 1;
if (j > 0) {
    for(int i=0;i<j;i++) {
        elementData[index+i]=elementData[index+i+1];
    }
}
elementCount--;
elementData[elementCount] = null; /* to let gc do its work */
}

```

insertElementAt

- a. This method inserts an element into the vector at a given position. Often, it is not called directly, but by the method `addElement(E obj)`, which inserts the given object at the end of the data structure.
- b. The preconditions verify that the underlying array has been properly initialized (constraint 1), the number of elements hasn't somehow been set to a negative value (2), and the number of elements isn't larger than the size of the underlying array (3). They also check that the index to insert the element into is not a negative position (4), or a position larger than the size of the array.
- c.

```

@Assertion(@Case(
pre = "elementData !=null && 0<=elementCount && elementCount<=elementData.length && index
    >=0 && index<=elementCount",
post = "elementData !=null && 0<=elementCount && elementCount<=elementData.length"))
public void insertElementAt(E obj, int index) {
    modCount++;
    if (index > elementCount) {
        throw new ArrayIndexOutOfBoundsException(">");
    }
    ensureCapacityHelper(elementCount + 1);
    for(int i=0;i<elementCount - index;i++) {
        elementData[index+1+i]=elementData[index+i];
    }
    elementData[index] = obj;

```

```
        elementCount++;  
    }
```

add

- a. This method simply adds the given element to the end of the vector.
- b. The preconditions verify that the underlying array has been properly initialized (constraint 1), the number of elements hasn't somehow been set to a negative value (2), and the number of elements isn't larger than the size of the underlying array (3). The post conditions verify the same things.
- c.

```
@Assertion(@Case(  
pre = "elementData !=null && 0<=elementCount && elementCount<=elementData.length",  
post = "elementData !=null && 0<=elementCount && elementCount<=elementData.length"))  
public boolean add(E o) {  
    modCount++;  
    ensureCapacityHelper(elementCount + 1);  
    elementData[elementCount++] = o;  
    return true;  
}
```

Metrics:

Lines of code for implementation: 254

Methods for implementation: 49

Lines of code for invariants: 60

Methods for invariants: 6

References:

Wikipedia's entry on dynamically resizing arrays: http://en.wikipedia.org/wiki/Dynamic_array

Data Structures, Algorithms and Applications in Java by Sartaj Sahni. Section 5.4: “Vector Representation.” pgs 176 – 180.

Algorithms in Java, 3rd Edition by Robert Sedgewick. Section 3.2: “Arrays.” pg 87.