CSCI 136
Data Structures & Advanced Programming

Lecture 7

Fall 2018

Instructors: Bill & Bill

Last Time

- Associations
- Code Samples
  - WordFreq, Dictionary (Associations, Vectors)
- Generic Data Types
- Lab 2 Design and Strategies

Today's Outline

- Vector Implementation
- Miscellany: Wrappers
- Condition Checking
  - Pre- and post-conditions, Assertions
- Asymptotic Growth & Measuring Complexity

Recall: Vectors

- Vectors are collections of Objects
- Methods include:
  - `add(Object o), remove(Object o)`
  - `contains(Object o)`
  - `indexOf(Object o)`
  - `get(int index), set(int index, Object o)`
  - `remove(int index)`
  - `add(int index, Object o)`
  - `size(), isEmpty()`
- Remove methods preserve order, close "gap"

Implementing Vectors (Parametrized)

- A Vector holds an array of Objects
- Key difference is that the number of elements can grow and shrink dynamically
- How are they implemented in Java?
  - What instance variables do we need?
  - What methods? (start simple)
- We'll focus on the generic version
- Let's explore the implementation….

Class Vector : Instance Variables

```
public class Vector<E> {
private Object[] elementData;     // Underlying array
protected int elementCount;       // Number of elts in Vector
protected final static int defaultCapacity;
protected int capacityIncrement;  // How much to grow by
protected E initialValue;         // A default elt value
}
```

- Why Object[]?
  - Java restriction: Can't use type variable, only actual type
- Why elementCount?
  - size won't usually equal capacity
- Why capacityIncrement?
  - We'll "grow" the array as needed

Basic Vector<E> Methods

```
public class Vector<E> {
public Vector()              // Make a small Vector
public Vector(int initCap) // Make Vector of given capacity
public void add(E elt)      // Add elt to (high) end of Vector
public void add(int i, E elt)     // Add elt at position i
public E remove(E elt)      // Remove (and return) elt
public E remove(int i)      // Remove (and return) elt at pos i
public int capacity()       // Return capacity
public int size()           // Return current size
public boolean isEmpty()    // Is size == 0?
public boolean contains(E elt) // Is elt in Vector?
public E get(int i)         // Return elt at position i
public E set(int i, E elt) // Change value at position i
public int indexOf(E elt)  // Return earliest position of elt
}
```

Class Vector : Basic Methods

- Much work done by few methods:
  - indexOf(E elt, int i) // find first occurrance of elt at/after pos. I
    - Used by indexOf(E elt)
    - remove methods use indexOf(E elt)
  - firstElement(), lastElement() use get(int i)
  - *Principle: Factor out common code!*
- Method names/functions in spirit of Java classes
  - indexOf has same behavior as for Strings
- Methods are straightforward except when array is full
- How do we add to a full Vector?
  - We make a new, larger array and copy values to it

Extending the Array

- How should we extend the array?
- Possible extension methods:
  - Grow by fixed amount when capacity is reached
  - Double array when capacity is reached
- How could we compare the two techniques?
  - Run speed tests?
    - Hardware/system dependent
  - Count operations!
  - We'll do this soon

ensureCapacity

- How to implement `ensureCapacity(int minCapacity)`?

```
// post: the capacity of this vector is at least minCapacity
public void ensureCapacity(int minCapacity) {
   if (elementData.length < minCapacity) {
      int newLength = elementData.length; // initial guess
      if (capacityIncrement == 0) {
      // increment of 0 suggests doubling (default)
         if (newLength == 0) newLength = 1;
            while (newLength < minCapacity) {
              newLength *= 2;
            }
       } else {
      // increment != 0 suggests incremental increase
         while (newLength < minCapacity) {
            newLength += capacityIncrement;
         }
      }
```

```
    // assertion: newLength > elementData.length.
       Object newElementData[] = new Object[newLength];
       int i;

    // copy old data to array
       for (i = 0; i < elementCount; i++) {
          newElementData[i] = elementData[i];
       }

       elementData = newElementData;
          // garbage collector will pick up old elementData
    }
    // assertion: capacity is at least minCapacity
}
```

Wrappers/AutoBoxing/Unboxing

- In Vector<E>, E cannot be a primitive type
- How to make a Vector of a primitive type?
- Java provides wrapper classes
- Examples:
  - Vector<Integer>
  - Association<String, Character>
- Each has a valueOf() method to return primitive
- Often Java will convert automatically

```
Association<String, Integer> a =
    new Association<String, Integer>("Bill", 97);
    int grade = a.getValue();
```

Wrappers/AutoBoxing/Unboxing

Primitive type	**Wrapper class**
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Pre and Post Conditions

- Recall `charAt(int index)` in Java String class
- What are the pre-conditions for charAt?
  - 0 <= index < length()
- What are the post-conditions?
  - Method returns char at position index in string
- We put pre and post conditions in comments above most methods

```
/* pre: 0 ≤ index < length
 * post: returns char at position index
 */
public char charAt(int index) { … }
```

Pre and Post Conditions

- Pre and post conditions "form a contract"
- *Principle: Ensure Post-condition is satisfied if pre-condition is satisfied*
- Examples:
  - `s.charAt(s.length() - 1)`: index < length, so valid
  - `s.charAt(s.length() + 1)`: index > length, not valid
- These conditions document requirements that user of method should satisfy
- But, as comments, they are not enforced

Other Examples

- Other places pre and post conditions are useful

```
// Pre: other is of type Card
// Post: Returns true if suits and ranks match
public boolean equals(Object other) {
    if ( other instanceof Card ) {
        Card oc = (Card) other;
        return this.getRank() == oc.getRank() &&
            this.getSuit() == oc.getSuit();
    }
    else return false;
}
```

Assert Class

- Pre- and post-condition comments are useful as a programmer, but it would be *really* helpful to know as soon as a pre-condition is violated (and return an error)

- The Assert class (in structure5 package) allows us to programmatically check for pre- and post-conditions

Assert Class

The Assert class contains the methods

```
public static void pre(boolean test, String message);
public static void post(boolean test, String message);
public static void condition(boolean test, String message);
public static void fail(String message);
```

If the boolean test is NOT satisfied, an exception is raised, the message is printed and the program halts

Assert Example

- Let's look in CardsWithBaileyAssert

```
// Pre: other is of type Card
// Post: Returns true if suits and ranks match
public boolean equals(Object other) {
      Assert.pre( other instanceof Card,
            "Error: parameter must implement
                  type Card");
      Card oc = (Card) other;
      return this.getRank() == oc.getRank() &&
            this.getSuit() == oc.getSuit();
}
```

General Rules about Assert

1. State pre/post conditions in comments
2. Check conditions in code using "Assert"
3. Use Fail in unexpected cases (such as the default block of a switch statement)

- Any questions?
- You should use Assertions in Lab 2

Measuring Computational Cost

Consider these two code fragments…

```
for (int i=0; i < arr.length; i++)
   if (arr[i] == x) return "Found it!";
```

…and…

```
for (int i=0; i < arr.length; i++)
   for (int j=0; j < arr.length; j++)
     if( i !=j && arr[i] == arr[j]) return "Match!";
```

How long does it take to execute each block?

Measuring Computational Cost

- How can we measure the amount of work needed by a computation?
  - Absolute clock time
    - Problems?
      - Different machines have different clocks
      - Too much other stuff happening (network, OS, etc)
      - Not consistent. Need lots of tests to predict future behavior

Measuring Computational Cost

- Counting computations
  - Count *all* computational steps?
  - Count how many "expensive" operations were performed?
  - Count number of times "x" happens?
    - For a specific event or action "x"
    - i.e., How many times a certain variable changes
- Question: How accurate do we need to be?
  - 64 vs 65? 100 vs 105? Does it really matter??

An Example

```
// Pre: array length n > 0
public static int findPosOfMax(int[] arr) {
        int maxPos = 0 // A wild guess
        for(int i = 1; i < arr.length; i++)
                if (arr[maxPos] < arr[i]) maxPos = i;
        return maxPos;
}
```

- Can we count steps exactly?
  - ”if” makes it hard
- Idea: Overcount: assume “if” block always runs
- Overcounting gives *upper bound* on run time
- Can also undercount for lower bound
- Overcount: 4(n-1) + 4; undercount: 3(n-1) + 4

Measuring Computational Cost

- Rather than keeping exact counts, we want to know the *order of magnitude* of occurrences
  - 60 vs 600 vs 6000, *not* 65 vs 68
  - n, *not* 4(n-1) + 4
- We want to make comparisons without looking at details and without running tests
- Avoid using specific numbers or values
- Look for overall trends

Measuring Computational Cost

- How does algorithm scale with problem size?
  - E.g.: If I double the size of the problem instance, how much longer will it take to solve:
    - Find maximum: $n - 1 \rightarrow (2n) - 1$ (≈ twice as long)
    - Bubble sort: $n(n-1)/2 \rightarrow 2n(2n - 1)/2$ (≈ 4 times as long)
    - Subset sum: $2^{n-1} \rightarrow 2^{2n-1}$ (2^n times as long!!!)
    - Etc.
- We will also measure amount of space used by an algorithm using the same ideas….