Outline

- What’s wrong with Enumerations?
- Indexing
- Searching
- Self-Organizing Vectors
- Binary Searching
- Sorting
- Command line Arguments
What’s wrong with Enumerations?

- Nothing is *wrong* with them, but they’re limited…
- They always give you elements in the *same order* in which they appear in the Vector you got them from. *
- What if you want to print the list in reverse?

* The book seems to suggest that the order of an Enumeration is up to the Vector implementer. This is *not true.*
Each element in a Vector has an index.

The first element is at index 0.

If the size of the Vector is n, the last element is at index n-1.

(Example: 4 elements numbered 0,1,2,3)

  This should remind you of substrings! 😊

Vectors store references to objects.
Vector v = new Vector();
v.addElement("What'll");
v.addElement("you do");
v.addElement("When you");
v.addElement("get lonely?");
Indexing: `elementAt`

```java
Vector v;
String s;

// Use elementAt method
s = (String) v.elementAt(2);
System.out.println(s);

// Use size method
System.out.println(v.size());
```

```
"What'll"
"you do"
"when you"
"get lonely?"
```
Why is this useful?

- Many reasons! Some may already be obvious.
- Let’s assume you just read in a sorted list of incomes, and stored them in a Vector.
- Want to know the median?

```java
int median = ( (Integer) myVector.elementAt(myVector.size() / 2 )).intValue();
```
Why is this useful?

- Remember our “print in reverse problem?"

- Solve it now with a while loop:
  
  ```java
  line = in.readLine();
  while (line != null) {
    v.addElement(line);
    line = f.readLine();
  }

  while (/*condition*/){
    /*body*/
  }
  ```
Why is this useful?

Remember our “print in reverse problem?”

```java
line = in.readLine();
while (line!=null) {
    v.addElement(line);
    line = f.readLine();
}
```

Try this with a `for` loop!

```java
int k;
for ( k = v.size()-1;  k >= 0;  k-- )
    System.out.println(v.elementAt(k));
```
Searching

- Easy method:
  - Start at 0, keep looking until you find what you’re looking for or until you run out of options.

- We want to find the index of the String “Java” in a Vector of Strings.
  - *What variables do we need?*

```java
private Vector v;  // instance variable!

private int search(String s){  // search v for s
  // return index of s
}
```

- Come up with an informal procedure
- Choose / define variables
- Initialize variables
- Guarantee termination
- Finish the loop body
private Vector v;  // instance variable!

private int search(String s){  // search v for s
    // return index of s
    int k = 0;
    // do the search!
    while ( ! ( k == v.size() || s.equals(v.elementAt(k)) ) )
        k);

    if ( k == v.size() )
        return -1;  // invalid index, use as “not found”
    else
        return k;
}
Concept is useful for some search engine systems…

Imagine you have a Vector of 2000 items for sale. Each shows # in stock

90% of requests are for the 100 most popular items.

\( \frac{100}{2000} = 5\% \)

We don’t want to spend 90% of our time searching through the least popular 95% of items.

Whenever an item is sought (and found), move it to the front!
Self-Organizing Vectors

Vector has 2 methods to support this:

```java
int index);  // moves elements
// after index up 1

Object o, int index);;

private void moveToFront( Vector v, int k ){
  v.insertElementAt(  v.elementAt(k) , 0  );
  v.removeElementAt(  k + 1  );
}
```

Original Vector: ABCDEFGHIJ

- ABCDEFGHIJ Seek D: ➔
- DABCEFGHIJ Seek D: ➔
- DABCEFGHIJ Seek G: ➔
- GDABCEFGHIJ Seek I: ➔
- IGDABCEFHJ Seek D: ➔
- DIGABCEFHJ
Self-Organizing Vectors

- Problem – if an unpopular item is accessed once, it takes a long time to work its way back toward the end of the list
- “Transpose” method – swap the accessed element with the element immediately in front of it.

```java
private void transpose(Vector v, int k) {
    if (k != 0) {
        v.insertElementAt(v.elementAt(k), k - 1);
        v.removeElementAt(k + 1);
    }
}
```

Original Vector: ABCDEFGHIJ

<table>
<thead>
<tr>
<th>Original Vector</th>
<th>Seek D:</th>
<th>Seek G:</th>
<th>Seek I:</th>
<th>Seek D:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCDEFGHIJ</td>
<td>⇨</td>
<td></td>
<td></td>
<td>⇨</td>
</tr>
<tr>
<td>ABDCEFGHIJ</td>
<td>⇨</td>
<td></td>
<td></td>
<td>⇨</td>
</tr>
<tr>
<td>ADBCEFGHIJ</td>
<td></td>
<td>⇨</td>
<td></td>
<td>⇨</td>
</tr>
<tr>
<td>ADBCEGFHIJ</td>
<td></td>
<td></td>
<td>⇨</td>
<td>⇨</td>
</tr>
<tr>
<td>ADBCEGFIHJ</td>
<td></td>
<td></td>
<td></td>
<td>⇨</td>
</tr>
<tr>
<td>DABCEGFIHJ</td>
<td></td>
<td></td>
<td></td>
<td>⇨</td>
</tr>
</tbody>
</table>
Self-Organizing Vectors

- Problem – with transpose, a popular item at the end of the list takes a long time to move to the front.
  - Move half-way up

```java
private void moveHalfwayUp( Vector v, int k ){
    v.insertElementAt( v.elementAt(k) , k/2 );
    v.removeElementAt( k+1 );
}
```

Original Vector: ABCDEFGHIJ

- Seek D:
  - ABCDEFGHIJ
  - ADBCEFGHIJ
  - DABCEFGHIJ
  - DABGCEFHJ
  - DABGICEFHJ
  - DABGICEFHJ

Seek D: ➔
Seek D: ➔
Seek G: ➔
Seek I: ➔
Seek D: ➔
Seek D: ➔
If the items we’re searching through are **sorted**, the search can be done much faster.

*When you look through the phone book* for “John Smith”, you don’t start with “Abbey Aaron”.

- You probably try to start near the “S” section, somewhere near the middle.
- If you’re too high, you try lower. If you’re too low, you try higher. Then you repeat until “John Smith” is found.
- If you *do* start at the beginning and look at every name sequentially, you probably don’t end up making very many phone calls.
Binary Searching

- Process: divide the range of possible places to search in half. Look at the midpoint.
  
  *If the midpoint is the sought item, you’re done!*
  
- If the sought item is > then the midpoint, the new data set is the right half of the data you just looked at.
  
- If the sought item is < then the midpoint, the new data set is the left half of the data you just looked at.
  
- Repeat the process
- You need to know where your current search data set begins and ends. Do this with indexes!
- The first search area will be from 0 to N – 1, where N == the size of the vector.
• Find “Wittworth”
• left = 0, right = 9
• midpoint = (right + left) / 2 -> 4
• “Williams” comes before “Wittworth”
  • so next search set is on the right.
  • Left was 0; now left is 5.
  • Right doesn’t change.
• left = 5, right = 9
• midpoint = (right + left) / 2  -> 7
• “Woods” comes after “Wittworth”
  • so next search set is on the left.
  • Left was 5; it doesn’t change.
  • Right was 9. Now it is 6.
• left = 5, right = 6
• midpoint = (right + left) / 2    -> 5
• “Wishart” comes before “Wittworth”
  • so next search set is on the right.
  • Left was 5. Now it is 6.
  • Right was 6. It doesn’t change.
Binary Searching

- left = 6, right = 6
- midpoint = (right + left) / 2

-> 6
• “Wittworth” matches “Wittworth”
  • we found what we were looking for,
  • We can return midpoint.
  • *What if we hadn’t found Wittworth here?*
In this example, it took us four “shots” to find the item we were looking for.

In a straight, non-binary search, we’d have to look at 7 items to find the right one.

If there are N items in the list, we’d need to take (on average) $N/2$ shots in a non-binary, sequential search.
In a binary search, we end up looking at (approximately) $\log_2 N$ items.

- This is a measure of algorithm efficiency, an important part of Comp. Science

<table>
<thead>
<tr>
<th>N</th>
<th>$\log_2 N$</th>
<th>$N/2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
<td>50</td>
</tr>
<tr>
<td>500</td>
<td>9</td>
<td>250</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>500</td>
</tr>
<tr>
<td>5,000</td>
<td>13</td>
<td>2,500</td>
</tr>
<tr>
<td>10,000</td>
<td>14</td>
<td>5,000</td>
</tr>
</tbody>
</table>
```java
static final int MISSING = -1;

private int bsearch(String s) {
    int left, right;
    left = 0; right = v.size();

    while (left < right) {
        int mid = (right + left) / 2
        String smid = (String) v.elementAt(mid);
        if (s.compareTo(smid) < 0)
            right = mid - 1;
        else if (s.compareTo(smid) > 0)
            left = mid + 1;
        else { //found
            return( mid );
        } //end while
    } //NOT found
    return( MISSING );
}
```

String `s1.compareTo(s2)` method:

- Does a lexicographical comparison
- Returns 0 if s1 == s2
- Returns -1 if s1 < s2
- Returns +1 if s1 > s2
Many computer applications involve sorting the items in a list into some specified order.

In order for a binary search to work, the things you’re searching through must be sorted first.

To sort a group of items, the following relationships must be clearly defined over the items to be sorted:

\[
\begin{align*}
    a < b \\
    a > b \\
    a = b
\end{align*}
\]

Ascending order: smallest ... largest

Descending order: largest … smallest

When designing or choosing an algorithm for sorting, one goal is to minimize the amount of work necessary to sort the list of items.

Generally the amount of work is measured by the number of comparisons of list elements and/or the number of swaps of list elements that are performed.
• Start at the beginning
• Walk down the list and find the smallest item.
• Swap them
• Now the *beginning* of the list is sorted...
• Repeat for the next index.
• Now the *beginning* of the list is sorted...
• Repeat for the next index.
Sorting
Sorting
Sorting

Organization

38
This is called a “selection sort” because in each iteration, you select the item that will occupy the next slot.

Its efficiency is proportional to $N^2$. There are more efficient sort algorithms ($N \cdot \log_2 N$), but they’re harder to code.

Note that in this algorithm, if you’ve sorted N-1 items, the last item automatically falls into place.

Formal Definition:

1. Loop (i) from 0 to the (number of elements to be sorted - 2)
   1.1 Assume the smallest remaining item is at the $i^{th}$ position, call this location smallest.
   1.2 Loop (j) through the remainder of the list to be sorted ($i+1 .. \text{size-1}$).
       1.2.1 Compare the $j^{th}$ & smallest elements in the unsorted list.
       1.2.2 If the $j^{th}$ element is < the smallest element then reset the location of the smallest to the $j^{th}$ location.
   1.3 Move the smallest element to the head of the unsorted list, (i.e. swap the $i^{th}$ and smallest elements).
private void sort(Vector v) {
    int k; // index of next slot to fill
    int n = v.size() - 1;

    for (k = 0; k < n; k++) {
        int j = getSmallest(v, k);  // ****
        swap(v, k, j);
    }
}

private void swap(Vector v, int a, int b){
    Object temp = v.elementAt(a);
    v.setElementAt(v.elementAt(b), a);
    v.setElementAt(temp, b);
}

Are the objects in the vector being swapped?
static final int EMPTYVECTOR = -1;
//return index of smallest element in the vector
//starting at the kth index assuming an alphabetical ordering

private int getSmallest( Vector v, int k) {
    if (v == null ||  v.size() <= k)
        return EMPTYVECTOR;   // error value

    int small = k;  // index of smallest found
    for (int i = k + 1; i < v.size(); i++) {
        String current  = (String) v.elementAt(i);
        String smallest = (String) v.elementAt(small);
        if ( current.compareTo(smallest) < 0 )
            small = i;
    }  // i == v.size( ) when loop exits
    return small;
Bubblesort Algorithm proceeds by walking down the list, comparing adjacent elements, and swapping them if they are in the wrong order. The process is continued until the list is sorted.

**Formal Definition:**

1. Initialize the size of the list to be sorted to be the actual size of the list.

2. Loop through the list until no element needs to be exchanged with another to reach its correct position.

   2.1 Loop (i) from 0 to size of the list to be sorted - 2.

      2.1.1 Compare the i\(^{th}\) and (i + 1)\(^{st}\) elements in the unsorted list.

      2.1.2 Swap the i\(^{th}\) and (i + 1)\(^{st}\) elements if not in order (ascending or descending as desired).

2.2 Decrease the size of the list to be sorted by 1.

Each pass "bubbles" the largest element in the unsorted part of the list to its correct location.
Trace of the bubblesort algorithm on the list below. Try to keep track of how many comparisons and swaps are performed.

Here is the trace of the bubblesort algorithm on the list below:

<table>
<thead>
<tr>
<th>Pass</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13 7 43 5 3 19 2 23 29 ?? ?? ?? ?? ?? ??</td>
</tr>
<tr>
<td>1</td>
<td>7 13 5 3 19 2 23 29 43</td>
</tr>
<tr>
<td>2</td>
<td>7 5 3 13 2 19 23 29 43</td>
</tr>
<tr>
<td>3</td>
<td>5 3 7 2 13 19 23 29 43</td>
</tr>
<tr>
<td>4</td>
<td>3 5 2 7 13 19 23 29 43</td>
</tr>
<tr>
<td>5</td>
<td>3 2 5 7 13 19 23 29 43</td>
</tr>
<tr>
<td>6</td>
<td>2 3 5 7 13 19 23 29 43</td>
</tr>
<tr>
<td>7</td>
<td>2 3 5 7 13 19 23 29 43</td>
</tr>
<tr>
<td>8</td>
<td>2 3 5 7 13 19 23 29 43</td>
</tr>
<tr>
<td></td>
<td>2 3 5 7 13 19 23 29 43</td>
</tr>
</tbody>
</table>
private void BubbleSort(Vector v) {
    int k; // index of next slot to fill
    int n = v.size() – 1;

    for (int Stop = v.size() – 1; Stop > 0; Stop--) {
        for (int Check = 0; Check < Stop; Check++) {
            String current = (String) v.elementAt(Check);
            String next = (String) v.elementAt(Check + 1);
            if (current.compareTo(next) > 0)
                swap(v, check, check + 1);
        }
    }
}

Bubblesort compares and swaps adjacent elements; simple but not very efficient.

Efficiency note: the outer loop could be modified to exit if the list is already sorted.
Recognize this line?:

```java
public static void main(String args[]) throws Exception{

    Ever wondered what `String args[]` means?

    ... funny you should ask...
```
Java programs can be run with arguments.
- These are similar to arguments of a Java message
- When running from the command line:
  ```java
  Java MyProgramName myArg1 myArg2 myArg3 etc...
  ```
  - arguments must be separated by white space!
Command Line Arguments

- `arg[]` is an Array of Strings
- To find out how many arguments there are:
  ```java
  int numargs = args.length
  ```
- To find any individual argument:
  - First argument
    ```java
    String neededArgument = args[0];
    ```
  - Last argument
    ```java
    neededArgument = args[numargs - 1]
    ```
class FileToScreen{
    public static void main(String args[])
        throws Exception{
        BufferedReader in = new BufferedReader{
            new InputStreamReader{
                new FileInputStream{
                    new File(args[0]))});
        String s = in.readLine();
        while ( s != null ){
            System.out.println(s);
            s = in.readLine();
        }
    }
}