Array Techniques

(adapted by SBJ from slides for Horstmann, Chapter 6)

Goals

- To become more comfortable/familiar with arrays
- To become more comfortable/familiar with using int variables to “point” at array elements
- To see some common algorithms for processing arrays
- Working with partially filled arrays

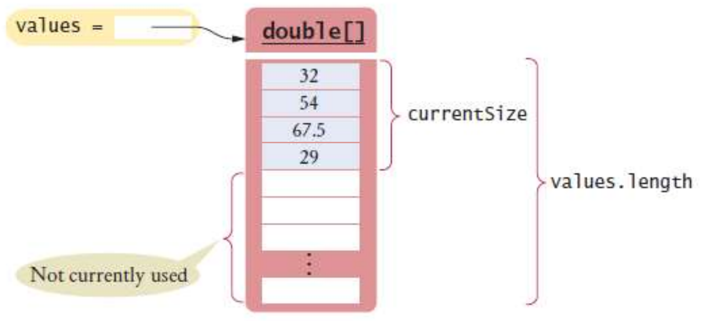
Partially Filled Arrays

- Array length = **maximum number** of elements in array.
- Usually, array is **partially** filled
- Define an array larger than you will need

```
final int LENGTH = 100;
double[] values = new double[LENGTH];
```

- Use companion variable to keep track of current size: call it `currentSize`
- Note: Common technique: use a constant to give the array size
  - And use the constant throughout the program

Partially Filled Arrays



- Note:
  - `currentSize` contains 4
  - The actual data is held at indices `0` – `currentSize-1`
  - Elements indexed `currentSize` – `values.length-1` are currently unused – but they may contain "old data"

Partially Filled Arrays

- To “add a new value to the end”:
  - Assign to the first unused element, and adjust size:

```
values[currentSize] = new value;
currentSize++;
```

- To fetch the last (end) value:

```
values[currentSize-1]
```

- To “throw away the last (end) value”:
  - Simply reduce `currentSize`:

```
currentSize--;
```

  - The old value *remains* in element `currentSize`
  - But the intention is to never access it
  - And it may be overwritten by the next add

Partially Filled Arrays

- A **loop to fill the array**

```
int currentSize = 0;
Scanner in = new Scanner(System.in);
while (in.hasNextDouble())
{
   if (currentSize < values.length)
   {
      values[currentSize] = in.nextDouble();
      currentSize++;
   }
}
```

- At the end of the loop, `currentSize` contains the actual number of elements in the array.
- Note: Stop accepting inputs when `currentSize` reaches the array length.

Partially Filled Arrays

- To process the gathered array elements, use the **companion variable**, not the array length:

```
for (int i = 0; i < currentSize; i++)
{
   System.out.println(values[i]);
}
```

- With a **partially filled array**, *you need to remember **how many elements are filled***

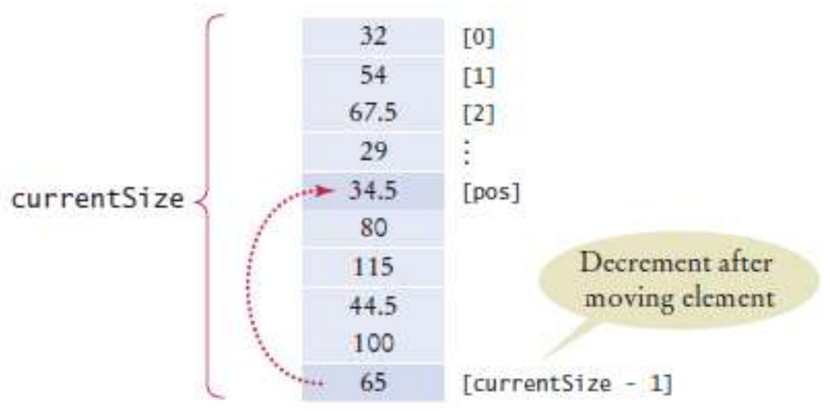
Common Array Algorithm: Removing an Element

Problem: To **remove** the element with **index** `pos` from the array `values` with number of elements `currentSize`.

- Unordered
  1. **Overwrite the element to be removed with the last element of the array.**
  2. **Decrement** the `currentSize` variable.

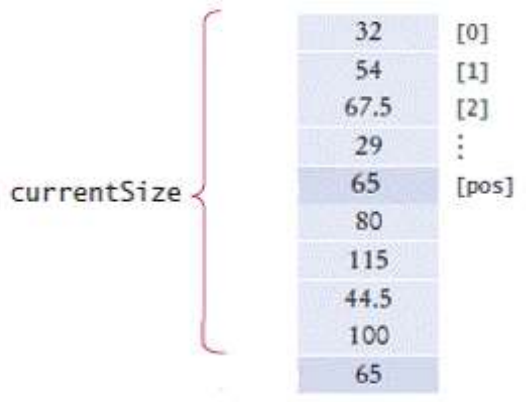
```
values[pos] = values[currentSize - 1];
currentSize--;
```

Common Array Algorithm: Removing an Element



Removing an Element in an Unordered Array
(after is on next slide)

Common Array Algorithm: Removing an Element



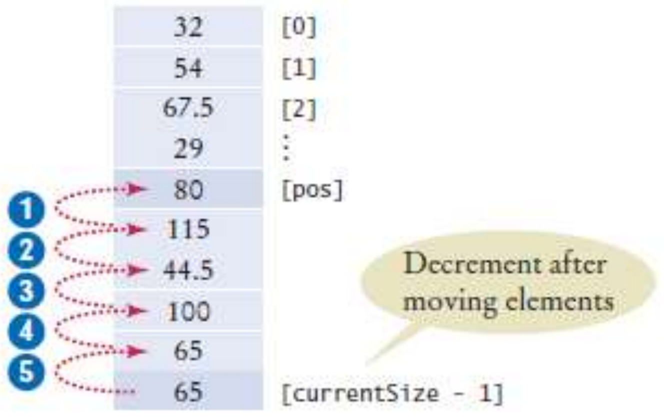
After removal

Common Array Algorithm: Removing an Element

- **Ordered** array
  1. **Move all elements** following the element to **a lower index.**
  2. Decrement `currentSize`

```
for (int i = pos + 1; i < currentSize; i++)
{
   values[i - 1] = values[i];
}
currentSize--;
```

Common Array Algorithm: Removing an Element



Removing an Element in an Ordered Array

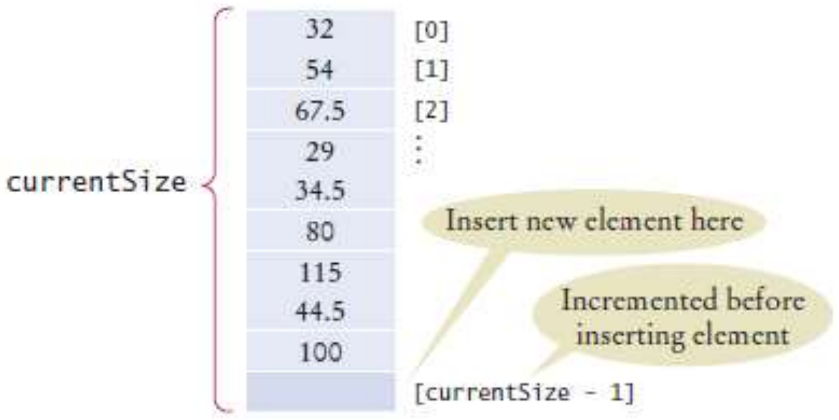
Common Array Algorithm: Inserting an Element

- If **order** does not matter
  and if there is a free space available:
  1. **Increment** the **variable tracking the size of the array**.
  2. **Insert** the new element at the **end of the array**.

```
if (currentSize < values.length)
{
  currentSize++;
  values[ currentSize-1 ] = newElement;
}
```

(with care, can increment the size after)

Common Array Algorithm: Inserting an Element



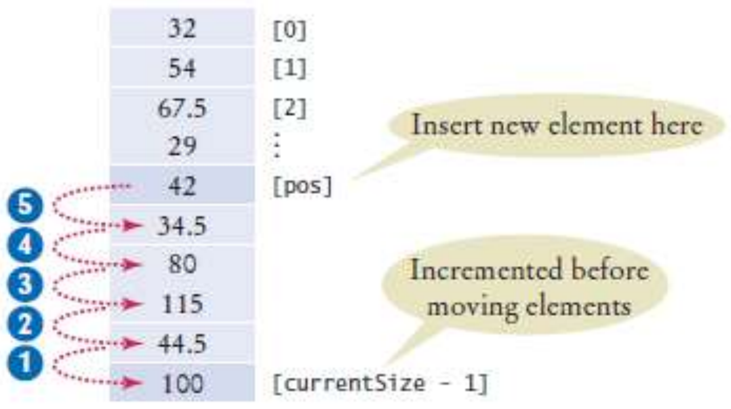
Inserting an Element in an Unordered Array

Common Array Algorithm: Inserting an Element

- If **order** matters, there is space and insertion is at `pos`:
  1. **Increment** the **variable** tracking the **size** of the array.
  2. **Move elements at `pos` and above to a higher index**
  3. Insert the element.

```
if (currentSize < values.length)
{
   currentSize++;
   for (int i = currentSize - 1; i > pos; i--)
   {
      values[i] = values[i - 1];
   }
   values[pos] = newElement;
}
```

Common Array Algorithm: Inserting an Element (42)



Inserting an Element in an Ordered Array

Puzzle

When inserting an element into an array, we moved the elements to larger index values, starting at the end of the array. Why is it wrong to start at the insertion location, like this?

```
for (int i = pos; i < currentSize - 1; i++)
{
   values[i + 1] = values[i];
}
```

Answer: This loop sets all elements to `values[pos]`!

Puzzle

- Assume the array `values` contains data as below. What does it contain after executing the following loop?

```
for (int i = 0; i < currentSize/2; i++)
{
   values[currentSize-1-i] = values[i];
}
```

Before: 2, 3, 5, -1, 45

Answer: 2, 3, 5, 3, 2
- first half "mirrored" in second half

Puzzle

- Assume the array values contains data as below. What does it contain after executing the following loop?

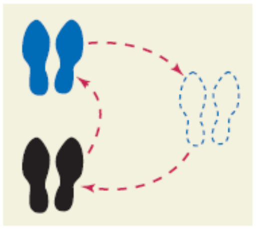
```
for (int i = 0; i < currentSize; i++)
{
   values[i]++;
}
```

Before: 2, 3, 5, -1, 45

Answer: 3, 4, 6, 0, 46

Common Array Algorithm: Swapping Elements

- To **swap two elements**, you need a **temporary variable**.



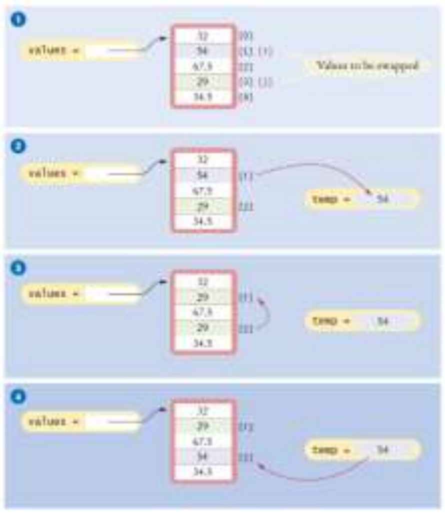
- We need to **save the first** value in the **temporary variable** before replacing it.

```
double temp = values[i];
values[i] = values[j];
```

- Now we can set values[j] to the **saved value**.

```
values[j] = temp;
```

Common Array Algorithm: Swapping Elements



Swapping Array Elements

Puzzle

- What does the following loop do to the array values?

```
int i = 0;
int j = values.length - 1;
while ( i < j )
{
   double temp = values[i];
   values[i] = values[j];
   values[j] = temp;
   i++;
   j--;
}
```

Answer: It reverses the elements in the array!

End of lecture