Data Structures and Algorithms
An example

- Program design issues
- Parallel arrays
- Partially filled arrays
- New implementation of partially filled arrays
- An intricate improvement

Program design issues

- (Most) programs' key purpose is to *process data*
- Therefore a core programming activity is:
  - The design of *storage* for the data:
    "Data structures" – variables, arrays, types, ...
  - The design of *algorithms* for processing the data in those structures
  - The two go together
- Usually there are many alternatives
  - Need to consider, evaluate and choose
- Many techniques have been designed
  - Entire text books are devoted to the subject
- But there is always scope for creativity!
- Example: an alternative for partially filled arrays...

Reminder: Parallel arrays

- Quoting from Wikipedia:
  "A form of implicit data structure that uses multiple **arrays** to represent a singular **array** of records. It keeps a separate, homogeneous data **array** for each field of the record, each having the same number of elements."
- Example: From a recent practical: An address book:
  - One array of Strings for names
  - One for addresses
  - One for phone numbers
- All elements with index 0 are related, similarly index 1, etc
- The types of the array elements could be different
  - int for phone number, maybe
- In Object Oriented Programming there is a better solution!

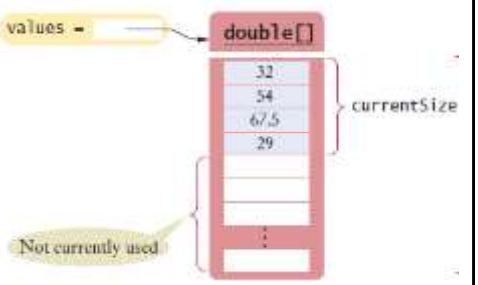
- Example of the address book:

Index	Name	Address	Phone no
0	Simon	4B63	7434
1	David	4B87	7445
2	Savi	4B68	7431
3	John	4B102	7286
4	Bruce	4B76	7432

```
System.out.println(name[i] + " " +
                   address[i] + " " + phone[i]);
```

Partially filled arrays (again)

- Often we need to create an array large enough to cope with *all anticipated data*
  - But it will not always be "full" of actual data
- Previous solution:
  - Use elements index 0 onwards, up to the "current size"
  - Use a companion variable to indicate current size/index of first unused element
  - Always insert at "top"
  - Delete from anywhere and "shuffle" to close gap
  - Diagram from Horstmann:



Alternative solution using parallel arrays

- Use one array for data values
  - Actual data could be *anywhere* in the array
- Use a parallel array of *booleans* to indicate in use/not in use
  - Where true: corresponding element in data array holds actual value
  - Where false: value of corresponding element in data array is irrelevant
- No need for "current size"
- No need to shuffle on deletion
- May need to search to insert! But can "chain" the empties! (Later)

Index	Data	InUse
0	3	true
1	67	true
2	12	false
3	-1	true
4	0	false
5	56	true
6	999	false

Sample algorithm coding

- Declaring the data structure:

```
private int size = 1000;  // Max values
private int[] data = new int[size];
private boolean[] inUse = new boolean[size];
```

- Setting the structure to be "empty":

```
private void setUp()
{
  for (int i = 0; i < size; i++)  // Traverse
  {
    inUse[i] = false;
  }
}
```


- How many actual data values are there?

```
private int getCount()
{
  int count = 0;
  for (int i = 0; i < size; i++) // Traverse
  {
    if (inUse[i])          // Data value here?
    {
      count++;             // Yes
    }
  }
  return count;
}
```

- Delete value at index i:

```
private void delete(int i)
{
  inUse[i] = false;        // That's all!
}
```

- Insert new value (find first unused element, if there is one):

```
private void insert(int value)
{
  for (int i = 0; i < size; i++)  // Traverse
  {
    if (!inUse[i])         // Unused?
    {
      data[i] = value;     // Yes, insert value
      inUse[i] = true;     // Now in use
      return;              // Immediate exit!
    }
  }
}
```

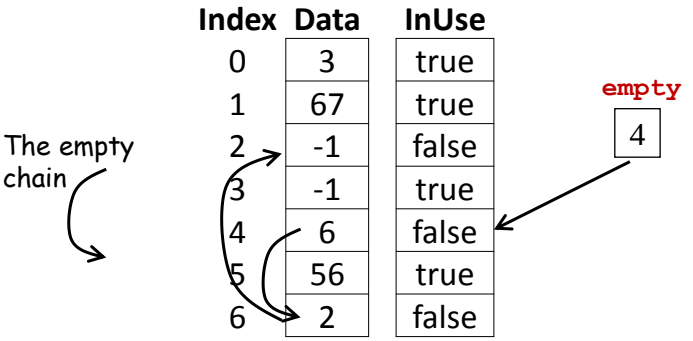

- Search for a given value, returning the index, or -1 if not found:

```
private int find(int value)
{
  for (int i = 0; i < size; i++)  // Traverse
  {
    if (inUse[i] && data[i] == value)  // Got it?
    {
      return i;   // Yes, found and exit
    }
  }
  return -1;      // Not found
}
```

Can we do better? Yes!

- Avoid the loop in `getCount` by keeping a `count` companion variable:
  - `count++` after insert, `count--` after delete
  - `getCount` simply returns `count`
- Avoid the loop in insert by *chaining* the empty elements:
  - Not in use elements contain the *index* of another not in use element – forming a chain
  - A companion variable, `empty`, holds index of first not in use element in the chain
  - The last element holds -1 (not an index!)
  - `insert` uses first in chain, adjusts `empty`
  - `delete` links vacated element to start of chain, adjusts `empty`
  - Example diagram on next slide

- Chained empty elements:



- Deleting element `i`:

```
if (inUse[i])
{ inUse[i] = false;
  data[i] = empty; // Link
  empty = i;
}
```

- Inserting `value`:

```
if (empty != -1)
{ int temp = data[empty];
  data[empty] = value;
  inUse[empty] = true;
  empty = temp; // Link
}
```

End of section