Fundamental Tasks and Techniques

- arrays
- linked lists
- searching, sorting, shuffling
- efficiency

Big Picture

One of the most fundamental things in programming is storing and manipulating a collection of elements.

Languages like Java commonly provide two ways to store a collection of elements –

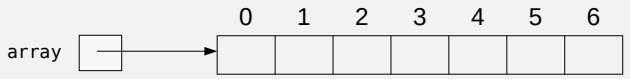
- arrays
- linked lists

These are known as *concrete data structures*, in contrast to *abstract data types* (which we will study later).

We'll also look at some fundamental tasks and concepts –

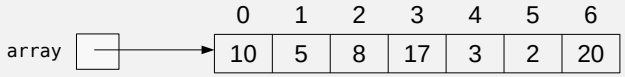
- searching, sorting, shuffling
- efficiency

Array Syntax



- declare an array variable
  ```
  int[] array;
  ```
- create the array itself (compartments)
  ```
  array = new int[7];
  ```
- initialize the compartments
  ```
  for ( int i = 0 ; i < array.length ; i++ ) {
    array[i] = …;
  }
  ```
- access compartment *i*
  ```
  array[i]
  ```
- length of the array (number of compartments)
  ```
  array.length
  ```


Array Syntax



- two ways to create and initialize an array in one step
  - initializer list
    ```
    int[] array = { 10, 5, 8, 17, 3, 2, 20 };
    ```
    - the initializer list *syntax can only be used when declaring a new array variable*
  - array literal
    ```
    array = new int[] { 10, 5, 8, 17, 3, 2, 20 };
    func(new int[] { 10, 5, 8, 17, 3, 2, 20 });
    ```
    - an *array literal* is an expression which can be used anywhere a value of the desired type is allowed

Array Syntax

- the typical way to go through every slot of an array in order to process each element is with a loop that counts through the array indexes

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

- a more compact alternative is the *for-each loop*

```
for ( int elt : array ) {
  System.out.println(elt);
}
```

  - each time through the loop body, `elt` is assigned another value from `array`

  – provides a common way to go through many kinds of collections (not just arrays)
  – can only be used for *traversal* – cannot assign to `elt` to change the contents of the collection

Array Usage

Patterns of usage for arrays used as containers –

- number of things doesn't change, and is known when the array is created
- number of things can change, but the maximum number is known when the array is created

  → *partially full array*, where not all of the slots will be used all the time

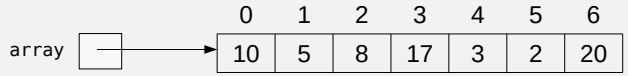
- number of things can change, but the maximum isn't known and/or the maximum is much bigger than the minimum

  → *dynamic array*, which is resized as needed to ensure enough slots without having too many extras

  - "dynamic" refers specifically to resizing as needed, but dynamic arrays will also always be partially full arrays

Fixed-Size Arrays

- number of things doesn't change, and is known when the array is created



  – legal indexes are `0`..`array.length-1`
  – loops accessing all slots go up to slot `array.length-1`

```
int[] array = new int[n];
for ( int i = 0 ; i < array.length ; i++ ) {
  array[i] = 0;
}
```


Partially-Full Arrays

If not all of the slots are used, how do we know which ones have values and which have junk*?

Which of the following is a common way to keep track of the elements in a partially full array?

	Answer	Respondents	Percentage
×	resizing the array every time an element is added	1	7%
×	using a boolean array to mark slots that contain elements	1	7%
×	storing null or 0 in empty slots	4	27%
✓	using only the first n slots of the array and storing the current number of elements (n) in a separate variable	9	60%

resizing every time an element is added means never having to deal with a partially full array (convenient, but it is expensive to resize frequently)

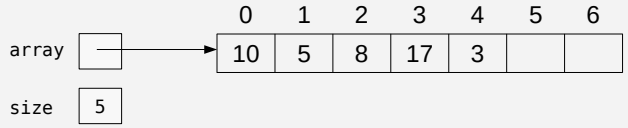
these are ways to distinguish between values and junk, but aren't what "partially full arrays" refers to

* there's no such thing as an empty spot – there's always some value there
you can fill the extra slots with a special value such as `null` or 0, but it isn't necessary because they aren't ever looked at

Partially-Full Arrays

If not all of the slots are used, how do we know which ones have values and which have junk?

- keep all the used slots together (at the beginning is convenient)
- maintain an additional variable to store the number of slots in use



Distinguish *capacity* (the number of slots) from *size* (the number of slots used).

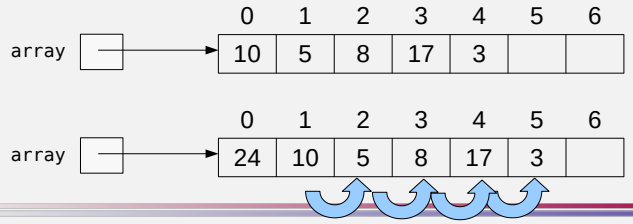
- though be aware that size is often used in place of capacity

Partially Full Arrays

When using a partially full array, what should happen when adding an element?

	Answer	Respondents	Percentage
×	put it at index 0, shifting other elements out of the way to make room	2	13%
×	put it at index 0, overwriting what is there	0	0%

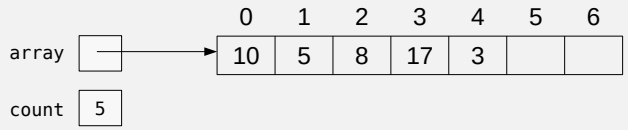
this is possible, but shifting takes time



Partially Full Arrays

When using a partially full array, what should happen when adding an element?

	Answer	Respondents	Percentage
×	put it at index count-1, where count is the number of elements already in the array	1	7%
✓	put it at index count, where count is the number of elements already in the array	9	60%
×	put it at index count+1, where count is the number of elements already in the array	3	20%



Dynamic Arrays

The capacity of an array is fixed when it is created.

What typically happens when a dynamic array becomes full?

	Answer	Respondents	Percentage
✓	allocate a new array with double the previous size and copy elements	9	60%
×	increase the array size by 1 element at a time so as to not waste space	5	33%
×	throw an exception and stop execution	0	0%
×	automatically overwrite the oldest data	1	7%

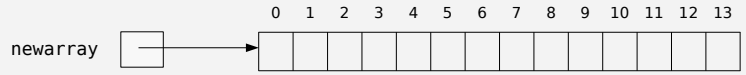
creates space, but immediately have to grow again when there's another insertion – doubling in size means that additional insertions can happen without having to grow again right away

there may be situations where this is appropriate, but it is not a general-purpose strategy (also have to somehow keep track of the oldest thing)

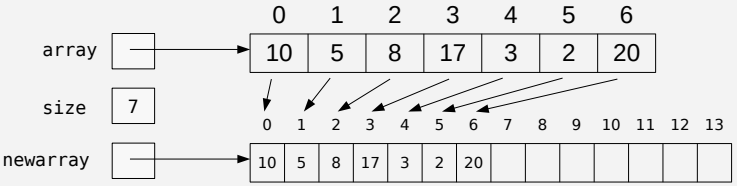
Dynamic Arrays

The capacity of an array is fixed when it is created.
Growing (or shrinking) an array requires –

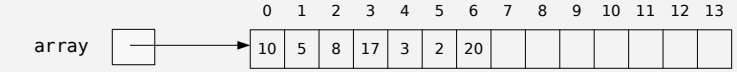
- creating a new array with a larger (or smaller) capacity



- copying the elements from the old array to the new



- replacing the old array with the new



Arrays as Collections

For collections, we typically need four kinds of operations –

- insert a new element
- remove an existing element
- access an element
- size

For arrays, we'll consider –

- insert at end
- remove from the end
- insert at index *i*
- remove index *i*
- get the element at index *i*
- size

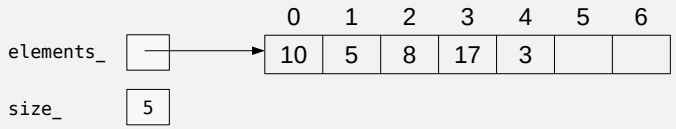
Figuring Out Array Operations

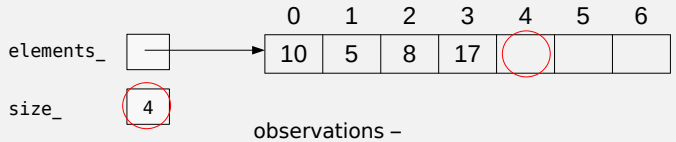
Strategy –

- create an example (not too big, not too small)
- draw before and after pictures
- identify what needs to be changed
- make the changes
- consider special cases – empty, first thing, last thing, …
  - create an example
  - draw before and after pictures
  - trace current algorithm
  - if something goes wrong, fix general case or write code to identify the case and do the right thing

Figuring Out Array Operations

removeAtEnd() –

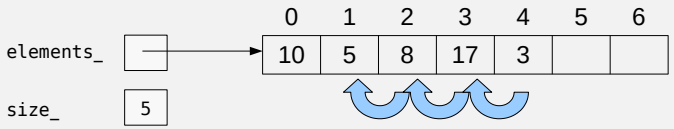


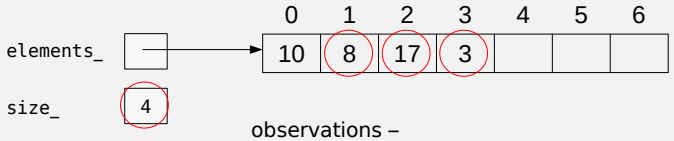


observations –
- size decreases by 1
- only one slot changes in the array

Figuring Out Array Operations

removeAt(1) –





observations –
- size decreases by 1
- many things in the array change → loop
- new value for slot i is what was at slot i+1

Figuring Out Array Operations

removeAt special cases –
- removeAt(0)
- removeAt(4)