Arrays

Arrays

An *array* is a data structure that is laid out in memory as a contiguous list of cells, with each cell indexed by the position of the cell in the structure.

Just like in Java, We will always index arrays with the first cell having index 0, and the last cell having index `len - 1`, where `len` is the number of cells in the array (i.e. the length of the array). Some programming languages (e.g. Fortran, R, Matlab) use a different policy of using 1 and `len` for the first and last cell respectively instead.

Cells do not need to be single bytes: the array is declared to contain some underlying type, such as Integers, Floating Point Numbers, Strings, etc. It is even possible to have an array of arrays of integers, where each cell contains a whole array of integers

Array Operations

The basic array data type has very few operations.

- Array creation

```
int[] nums = new int[4]
```

- Getting values from cells in the array

```
val = nums[0]
```

- Assigning values to cells in the array

```
nums[1] = 23
```

- Getting the length of the array

```
len = length(nums)
```

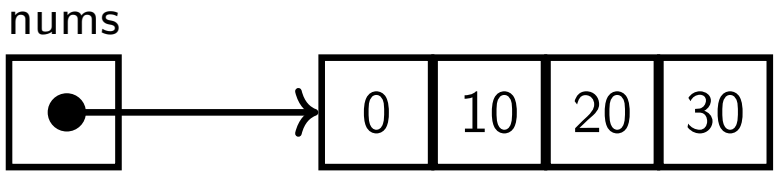
More sophisticated List ADTs often use basic arrays in their implementation, but add more complex operations such as increasing the size of a List, Sorting a list, concatenating Lists etc.

List as an array of a fixed length

In pseudocode, you can create an array using Java-like syntax:

```
1 int[] nums = new int[4]
2 for (i=0, i<length(nums) ; i=i+1)
3     nums[i] = i * 10
```

This can be described in a diagram such as the one below, and results in the memory layout shown to the right



i	Memory
⋮	⋮
3344	23
3340	30271
3336	30
3332	20
3328	10
3324	0
3320	6738
⋮	⋮
nums:3100	3324

array `nums`

Here we assume that the word size is 32 bits or 4 bytes, and integers (and memory pointers) are also 32 bits.

- `nums` is a variable whose cell in memory is at address 3100. The contents of this cell is a memory address, 3324, which is where the array starts
- Every `int` in the array occupies one word or 4 bytes in memory
- Because we declared our array to be an array of integers, the compiler knows that every entry in the array takes 4 bytes, and if the array starts at location 3324, then it knows that:
  - `nums[0]` is at address 3324,
  - `nums[1]` is at address $3324 + (1 \times 4)$,
  - `nums[2]` is at address $3324 + (2 \times 4)$, etc.

More complicated arrays in memory

In its simplist form, a Java class collects variables together into a single structure

```
1 class Point {
2     float x;
3     float y;
4 }
5 Point[] locations = new Point[3];
6 locations[1].x = 25.2;
7 locations[1].y = 38.6;
```

i	Memory
⋮	⋮
4052	0.0
4048	0.0
4044	38.6
4040	25.2
4036	0.0
4032	0.0
⋮	⋮
locations:3100	4032

Given that a float is 4 bytes, the following is what happens in memory:

```
1 Cell at address locations+1*2*4+0 is set to 25.2;
2 Cell at address locations+1*2*4+1 is set to 38.6;
```

In the 1*2*4: the 1 is the index into locations, the 2 is the number of words in a Point object, and the 4 is the size of the word.

Memory Management Reviewed

In Java

- Memory allocation is automatic
- Freeing memory is automatic (by the garbage collector)
- Bounds of arrays are checked

In C or C++

- Allocations are explicit
- Freeing memory is explicit
- Bounds are not checked

Java is slower and safe, C and C++ is fast and dangerous.

A very common mistake is to try to access the last cell in an array incorrectly:

```
int[] a = new int[5];
a[5] = 1000; // Error:  the cells are a[0] to a[4]
```

This leads to an `ArrayIndexOutOfBoundsException` in Java whereas in C (or C++) this goes through without a warning and can lead to a corruption of data in memory!

Inserting into an Array by Shifting Up

To insert a point at position `pos`, where $0 \leq pos \leq size$:

```
1  maxsize = 100
2  Point[] locations = new Point[maxsize];
3  int size = 0;     // number of points currently stored
4
5  void insert(int pos, Point pt) {
6    if (size == maxsize) {
7        throw new ArrayFullException("locations array");
8    }
9    for (int i=size-1; i >= pos; i--) {
10     // Copy entry in pos i one pos towards the end
11     locations[i+1] = locations[i];
12   }
13   locations[pos] = pt;
14   size++;
15 }
```

If we want to insert a value to an array (at a certain position) we can do this in two steps:

1. Create a new array, of size bigger by one.
2. Copy elements of the old array to the new one to the corresponding positions.

However, this requires to copy the whole array every single time. Instead, we can allocate a big array at the beginning (of size `maxsize`) and then always "only" shift elements whenever we are inserting/deleting one.

Exercise: write the corresponding pseudocode to remove an item from an array by shifting down