

# Introduction to Algorithms and Data Structures

## Lecture 6: Representation of program data in memory

John Longley

School of Informatics  
University of Edinburgh

3 October 2024

# Representations of data

Having seen a few **algorithms**, we now turn to **data structures**: i.e. ways of representing/structuring data in memory.

In due course, we'll see how to implement our own data structures. But we start at the bottom, with the '**primitive**' data structures that programming languages typically provide as built-in.

Actually, we'll begin one step further back:

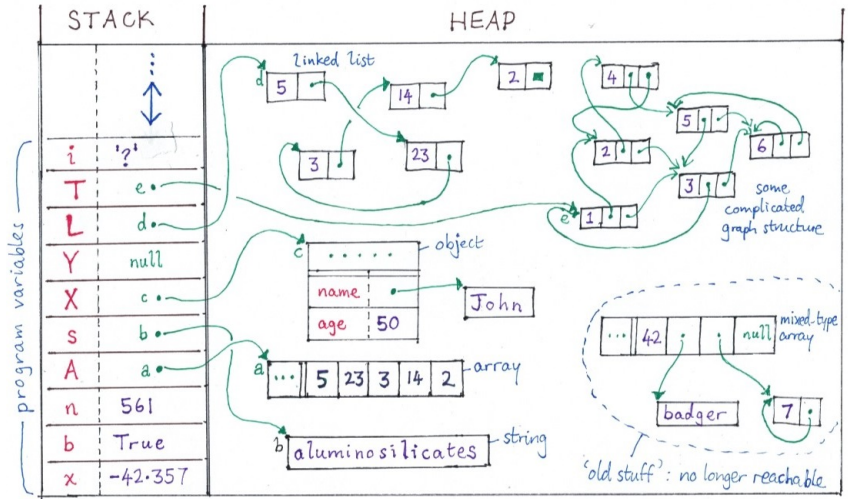
How, in general, is program data organized in memory?

Picture is broadly similar for most modern programming languages (Java, Python, Haskell, . . . ): **Stack** and **Heap**.

Remember: simplifying a bit, we think of memory as consisting of **words** each with an **address** (i.e. location).

Any address can be itself be stored in a single word — though there may be fewer addresses than word values.

# Typical organization of program data in memory



## Program memory: summary

- ▶ Contents of program variables are stored on a **stack**, which grows and shrinks as variables come in and out of **scope**.
- ▶ Stack items are contiguously arranged in memory, so must have **fixed size** (e.g. 1 or 2 words).
- ▶ Typically, a stack item contains either a **basic value** (e.g. 561, True) or a **reference** to something on the **heap**.
- ▶ Heap objects can live anywhere in memory, be of any size, and may contain references to other heap objects.
- ▶ When heap objects are created (**allocated**), the memory manager will decide where to put them. But references to other objects can be changed later – so we can end up with a real mess!
- ▶ As execution proceeds, some heap objects may become **unreachable**. In many languages (e.g. Java, Python, Haskell), a **garbage collector** (i.e. **memory recycler**) detects this and reclaims the space.

## Details may vary . . .

Our picture is mostly 'Java-like' (except for the mixed-type array).

- ▶ In Java, anything of **reference type** (including 'objects' and 'arrays') lives on the heap.

In C, built-in 'arrays' live on the stack, and their size is **static**: an array variable **A** has an associated size fixed throughout its lifetime.

In Python, pretty much everything lives on the heap.

- ▶ Python also offers 'lists' and 'arrays', both implemented much like the heap array in our picture. (Difference: 'lists' allow mixed types.) In functional programming languages (Haskell, ML, Lisp), 'lists' are more typically implemented as **linked lists**.

Java offers many classes for 'lists', e.g. **ArrayList**, **LinkedList**.

- ▶ In Java, all reference types have special value **null** ('pointer to nowhere'). Default initial value for any variable/field of ref type. Closest Python equivalent is **None** – this is actually a reference to a specific object, with no fields or methods.

**Idea:** Once we have the basic picture, we have a framework for understanding such differences.

# Assignment by reference

In Java or Python, **assignment** statements have the form

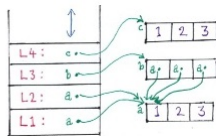
$$\text{variable} = \text{expression}$$

E.g. `s = "smiles"[1:5]` # assigns ref to s

What does an expression actually evaluate to? Either a **basic value** or a **reference** to a heap object. (Or **null** in Java.) List example:



```
L1 = [1, 2, 3]
L2 = L1
L3 = [L1, L1, L1]
L4 = L1[:]
```



**This matters!** Think what happens when we do `L1[2] = 5`.

A heap object will be 'copied' only if we request it (e.g. using `'[:]`' for lists in Python).

Similarly in Java (copying often done by a `clone()` method).

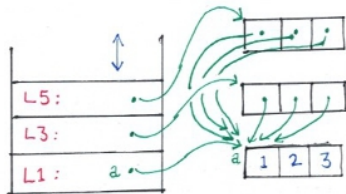
**Geek point:** Technically, a small integer like 2 is itself a reference to a **pre-allocated** heap object. But harmless to write this ref just as '2'.

## Shallow vs. deep cloning

In Python, '[':]' makes only a **shallow clone**: copies 'top level only'.  
E.g. think about **lists of lists**:



```
L1 = [1,2,3]
L3 = [L1,L1,L1]
L5 = L3[:]
```



Again, think what happens with  $L1[2] = 5$ .

For a **deep clone** (fresh copy of entire structure, with no sharing), we could in this case write  $L6 = [L1[:],L1[:],L1[:]]$ .

In general, may need to write a (possibly **recursive**) program to deep-clone the data structures in question.

## Equality testing

Equality testing can be ...

- ▶ **by reference** ('are the addresses the same?'), or
- ▶ **by value** ('do we find identical things at those addresses?')

In Python, `is` means reference eq (a.k.a. **identity**),

`==` means (deep) value eq.

**Exercise:** After executing column 1, what does column 2 give?

<code>L1 = [1,2,3]</code>	<code>L2 is L1</code>
<code>L2 = L1</code>	<code>L4 is L1</code>
<code>L3 = [L1,L1,L1]</code>	<code>L2 == L1</code>
<code>L4 = L1[:]</code>	<code>L4 == L1</code>
<code>L5 = L3[:]</code>	<code>L6 is L3</code>
<code>L6 = [L1[:],L1[:],L1[:]]</code>	<code>L6 == L3</code>

**Warning:** In Java, `==` means reference equality!

For value equality, typically use an `.equals` method.

For numbers and strings, always use value equality.

(Identity is unpredicable!)

## About those NullPointerExceptions

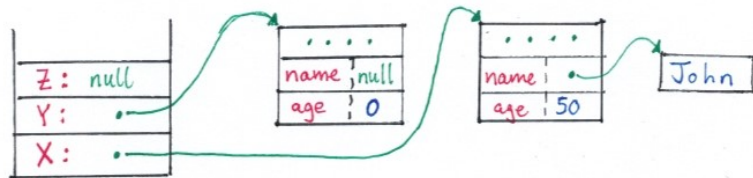
Consider a Java expression of form *expr.fieldname*. E.g. *X.age*.

The *expr* evaluates to a **reference**, or perhaps to **null**.

But when we pass the '.', we follow the reference to reach what it points to (**dereferencing**) ... and so risk a **NullPointerException** if *expr* evaluates to **null**!

Same goes for the '.' in *expr.methodname*(arguments).

E.g. *X.name.length()*.



Understanding this (and drawing pictures) can go a long way towards rooting out those pesky **NullPointerExceptions**.

## Basic operations

The following operations (among others) may all be assumed to work within **constant time** (i.e.  $\Theta(1)$  time):

- ▶ Reading / writing contents of program variables (basic or ref type).

```
n          n = 341          X          Y = X
```

- ▶ Accessing / updating a field in a given object (involves deref).

```
X.age      X.age = 51      X.name = s
```

- ▶ Accessing / updating an entry in a given array (may involve deref).

```
A[42]      A[42] = 51
```

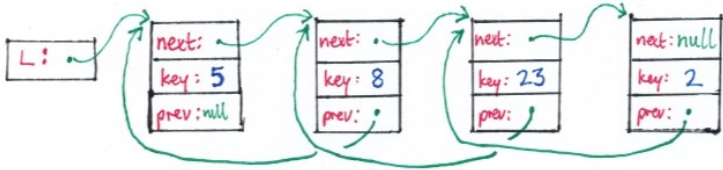
- ▶ Allocating a new object (e.g. of a given class) on the heap (not counting initialization of fields).

```
X = new Person()
```

- ▶ Allocating a new array on the heap, not counting initialization of all its entries.

## Linked lists

In Java/Python, **Linked list** cells would typically be simple **objects**, e.g. of class **Cell**, with fields called **key**, **next** and maybe **prev**.  
E.g. a **doubly linked list**:



In functional languages, **singly-linked lists** are everywhere, but presentation may look more abstract. E.g.

L.key	written as	head L
L.next	written as	tail L
new Cell(x,L)	written as	cons(x,L) or x:L
null	written as	nil or []

Anyway, find  $n$ th element of a linked list **L** takes time  $\Theta(n)$ .



**Reading:**

`https://docs.python.org/3/reference/datamodel.html`  
(just 3.1);  
CLRS chapter 10, especially 10.2 and 10.3.

**Photograph:** Glen Etive from path to Lairig Gartain.  
Kyriakos Kalorkoti (by kind permission)