Informatics 2: Introduction to Algorithms and Data Structures
Notes on Lecture 6
Representation of program data in memory

John Longley
© University of Edinburgh 2026

Having talked about *algorithms* a fair bit in Lectures 2 and 5, in this lecture we'll shift gear and start to talk about *data structures*. In due course, we will be seeing how we might implement interesting data structures for ourselves, such as hash-probe tables or red-black trees. But here we'll start at the bottom level by looking at the 'primitive' data structures that today's programming languages usually have already built into them—things like arrays and lists of various kinds. These are the basic data structures which we use all the time when we're programming, and which we would use as a foundation for building more interesting and complex data structures of our own. We'll be saying a bit about how these basic data structures are typically represented or implemented inside a computer, and what their efficiency characteristics are—this is all part of what we needs to know to be good programmers.

Actually, we'll begin even one small step further back than that. Even before we look at any specific structures like arrays or lists, we'll say a bit about how program data *in general* is organized in memory. This will perhaps make contact with things that are covered in the Computer Systems course—although here we'll be looking at things from the perspective of high-level programming languages like Java and Python.

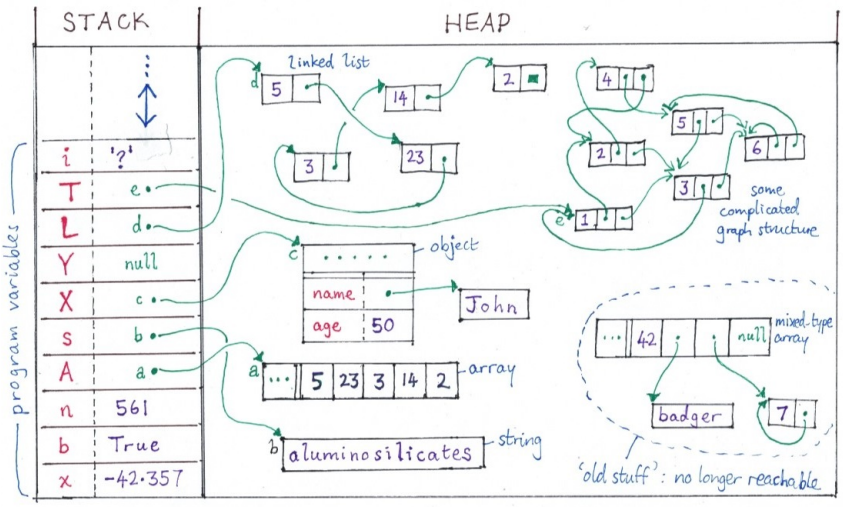
Simplifying a little bit, we'll think of a computer's memory as consisting of *words*—which might be 32- or 64-bit words, say—where each word has an *address* or location in memory. And it'll be convenient to suppose here that each of these addresses can itself be stored in a single word: in other words, the number of bits it takes to write an address doesn't exceed the word size. This means that the contents of one word might play the role of a *reference*, or *pointer*, to another word somewhere else in memory.

The Stack and the Heap

Below is a picture that shows broadly how program memory is organized in most modern high-level programming languages, including Java, Python, and Haskell. In particular, the picture shows that memory is organized into two distinct regions known as the *Stack* and the *Heap*.[1] As we'll see, grasping this

[1]Warning: Both terms *stack* and *heap* are also used in this course to refer to specific data structures that might be implemented *within* program memory—see Lectures 7 and 11

picture is very important for understanding what programs actually do when they're run, and this can be extremely helpful for tracking down various kinds of bugs in your code.



Some of the details here may vary a little from one language to another (we'll mention some examples below), but this broadly illustrates the kinds of things that can go on. Intuitively, we can think of the Stack as a bit like a formal garden, with everything neatly regimented and organized into rows—whereas the Heap is more like a wild, overgrown jungle where things can appear wherever they like and all kinds of messiness are allowed.

As a starting point, the *Stack* is essentially the region of memory where we store the values of the variables that are mentioned within our program—things that have variable names, such as `n` or `A`. The reason it's called a Stack is that is growing and shrinking all the time as variables appear and disappear. For instance, the variables shown as living near the bottom of the Stack might be *global variables* that are allocated at the start of our program run and which remain there right through to the end; while other variables higher up might be more 'temporary' or local to a particular block of code, such as a function or method call—such variables will be added the top of the Stack when we enter the relevant code block, and will expire when we exit it, so will be removed from the Stack at that point.

Another important point about the Stack is that the variables are allocated contiguously in memory, i.e. one immediately next to another. (Here I've drawn a Stack that grows 'upwards', but it could equally well be downwards.) This has the important consequences that each of these variables occupies a *fixed* amount of memory—perhaps 1, 2 or 4 words, depending on the kind of variable it is (e.g. its type). This memory size is fixed when the variable is allocated, and although we can modify the *value* stored in the variable as much as we like (e.g. we could certainly change the value stored in `n` from 561 to 562), we can't change the amount of space it takes up. This is because there will very likely be

respectively. In these notes, we'll use capital letters when referring to *the* Stack and *the* Heap in the sense we have in mind here—however, this isn't a standard convention.

further Stack variables sitting above the variable, and we certainly don't want to shift all of these upwards just to make a bit more room for this one variable.

This has a very important consequence. If we're wanting a program variable that can store a string, or an array, or some other item of data whose size could vary a lot throughout the lifetime of the variable, that data value will have to live on the *Heap* rather than the Stack. What we will then store in the Stack entry for the variable will then be simply an *address*, i.e. a reference to the location in Heap memory where the value itself can be found. (This is illustrated by the variables s and A in the picture above.) This works fine, because however long a string itself might be, the *address* where it lives will occupy a fixed memory size (e.g. a single word) and so can be stored on the Stack.

What emerges, then, is that there may be two kinds of variables living on the Stack (this is indeed very much how things work in Java, for instance). First, there are variables of what we may call *basic types*, with might include 64-bit integers, floating-point reals, booleans, single Unicode characters, etc., which have the property that they occupy a small, fixed amount of memory determined by their type. The value of such a variable can therefore be stored *in-place* within the Stack entry for that variable (this is illustrated by the variables `x`, `n`, `b`, `i` in our picture). Secondly, there are variables of *reference type*, such as strings, arrays and other objects (in the sense of object-oriented programming): here the data item itself lives on the Heap, and what is stored within a Stack entry is simply a reference to it (this is illustrated by the variables `A`, `s`, `X` in the picture). This means that every time our program wants to access the data associated with one of these variables, it will have to 'follow' the pointer to find the location in Heap memory where the data resides. A further thing to note about variables of reference type is that there's also the possibility (in Java, for instance) that such a variable may contain the special value 'null', which we can think of as a 'pointer to nowhere': this will, for example, be the default value for any variable of reference type to which we haven't yet assigned anything else. (Anyone who has done any Java coding will have encountered NullPointerExceptions - more on these below!)

Note that on the above scheme, it's quite possible to update the contents of `s` or `A` to make them refer to a longer string or array than before. What will happen is that a new string or array will be created somewhere else on the Heap, and the Stack entry for `s` or `A` is then updated to contain the *reference* to this new item.

There will of course be certain conventions governing how these Heap objects are structured. For example, an *array* is likely to consist of a contiguous block of *cells* in memory, each of the same memory size, prefaced by some 'header' information giving perhaps the length of the array, the size of each cell, and any other book-keeping information needed by the language implementation. This means that to lookup, say, the contents of the cell `A[3]` in the picture, we will retrieve the address a where the array lives, add some appropriate offset to a to get the address a' where the cell `A[3]` is expected to live, then lookup what's at the address a' to find the desired value (14 in this case).

Likewise, an *object* is likely to be a record containing a bunch of *fields* which themselves behave like variables—for instance, a 'name' field of type `String` and an 'age' field of type `Integer`—perhaps also with some header information for internal system use. These fields may themselves be of two kinds: there are fields of basic type, whose values occupy a predictable amount of memory and

so may be stored in-place within the object; and there are fields of reference type, in which we store a reference to the item itself which will live somewhere else on the Heap. The point is that when we allocate an object on the Heap, we set aside a fixed amount of memory for each of its fields, so the same rules apply to these fields as apply to the Stack variables themselves.

We've seen already that items on the Heap can contain references to other items on the Heap. This idea can be carried as far as we like: for instance, the variable `L` in the picture contains a reference to a *linked list*, which consists of a bunch of cells in Heap memory each containing a value (an integer in this case) along with a reference to the location where the next cell in the list can be found. (We can suppose that for the last cell in the list, this reference is simply 'null'.) We can also have even more complicated graph structures (as illustrated by the variable `T`) containing a bunch of nodes that point to one another and even perhaps to themselves in complicated ways. In summary, data structures on the Heap can get as complicated and tangled as we like.

Unreachable data and memory recycling

There is one further idea illustrated by the diagram above. In the bottom-right corner, there are a handful of structures that involve references to each other, but which aren't *reachable* from the Stack entries by chasing any sequence of references. This means that these structures can't actually be referred to or manipulated by any expression within our program at all.

How did this unreachable stuff get there? We can imagine, for example, that there used to be a reference to the array here—perhaps from the Stack variable `A` which is of array type—but that at some point the value of `A` got switched to make it point to a different array (e.g. the one at location a), leaving the original array structure stranded. And because these stranded items are no longer reachable by our program, they are no longer needed at all, and we would like to be able to *reclaim* the memory space they occupy so that it can be re-used for other purposes. Indeed, most modern language implementations nowadays have a built-in facility for doing this automatically, so that the programmer does not have the burden of keeping track of which data structures are still reachable. This process of reclaiming space from 'old' objects is traditionally known as *garbage collection*; but in these environmentally conscious times, I'd like to suggest that we rebrand it as *memory recycling*.

How does a memory recycler work? This in itself is an interesting topic, and is itself an area where clever algorithms have an important role to play. But the starting point is the idea that we somehow want to identify all the Heap data here that's *reachable* in any number of steps from the starting-points on the Stack. Once we've done that, anything that's not reachable is fair game for being mopped up and reused. This problem of comprehensively visiting everything that's reachable from these starting points is really an instance of a very general problem of *graph traversal*: it's essentially very similar to what we'd like a web crawler to do, as we mentioned Lecture 1. The kinds of algorithms used for traversing graphs are something that will be covered from Lecture 13 onwards.

© University of Edinburgh 2026

Java, Python, C, ...: Some finer points

So far, we've been talking at a very general level—the broad picture we've described is applicable to most modern programming languages. However, some of the details do vary a bit from one language to another, so let's take time to note a few of these differences.[2]

- In Java, arrays live on the Heap, which means it's possible for a program to update an array variable to make it point at an array of different size. However, this contrasts with the C programming language, where arrays may live on the Stack. This brings with it the constraint that arrays in C are *static*: any array variable `A`, say, has an associated array size which is fixed when the variable is first allocated, and which remains fixed throughout the lifetime of that variable. So in C, we couldn't reassign the value of `A` to make it point to a bigger array. This has some efficiency advantages—for instance, when looking up the contents of an array cell, we save the step of having to follow the reference from the Stack entry to the Heap location—and we're also able to reclaim the space occupied by the array as soon as the array variable expired, without having to wait for the memory recycler to detect that it's no longer reachable. Most recent languages, however, have opted for the extra flexibility that comes with storing arrays on the Heap.

- Python offers us both things it calls *lists* and things it calls *arrays*. Both of these live on the Heap, and both look pretty much like the 'array' we showed living at the address a. The differences between lists and arrays in Python aren't huge. For most purposes, it's lists we want to use because they're very flexible. The main difference is that with arrays in Python, all the items stored in the array must have the same type. This means that arrays can be stored quite compactly with no wasted space. Lists in Python, by contrast, are allowed to contain items of different types. Thus, the 'mixed type array' shown in our picture could exist within Python, except that it would be called a list rather than an array.

- In functional languages such as Haskell, things are a little different. Here, the things called *arrays* would again look like the Heap array in the picture. But the things called *lists* in functional languages are most typically implemented as *linked lists*. This is an important difference to be aware of, because as we'll see in the next lecture, arrays and linked lists have very different efficiency characteristics.

- Java has built-in structures called *arrays* (which live on the Heap), but also offers a whole selection of library classes for lists implemented in different ways, e.g. ArrayList and LinkedList. These have the status of *objects* in Java, and they all live on the Heap. In each case, the API documentation for each class describes roughly how it's implemented and give some information on its efficiency properties, sometimes making use of Big-O notation.

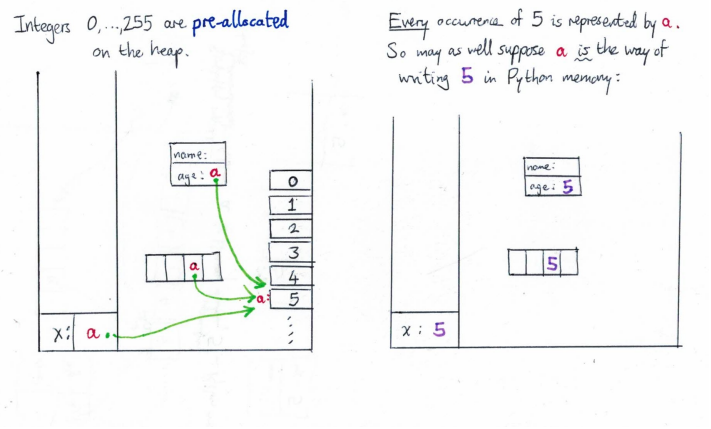
[2] The diagram we've given above is in most respects close to how things are in Java, with its clear distinction between basic types and reference types. The only exception is the mixed-type array in the bottom-right corner, which wouldn't be permitted by Java's fairly strict type system.

- In Java, as we've mentioned all reference types have a special value `null`, which we can think of as a pointer to nowhere, and which will be the default initial value for any variable or field of reference type before we've assigned anything else to it. In Python, the closest equivalent is a special value called `None`. However, this isn't a pointer to nowhere: rather, it's actually a reference to a particular dummy object living on the Heap, surprisingly called 'None', and with no fields and no methods.

- There's also a more radical difference between Java and Python which calls for a bit more explanation. In Python, strictly speaking, there are no such things as 'basic' data values that live in-place on the Stack: rather, *everything* in Python is an object, and as such lives on the Heap. This includes even integers like 5 and -3295849384950113885: indeed, since integers in Python are not limited to 32 or 64 bits but may be of arbitrary size and occupy any number of words, it makes sense that they have to live on the Heap.

  This means that whenever there is a need to store some new integer such as -3295849384950113885, a Heap object containing this integer will be created, occupying as much space as required. In principle, if our program performs two separate calculations that both happen to yield -3295849384950113885, this *may* result in two identical copies of this object being created on the Heap (though there may sometimes be cases where the system is clever enough to spot that these two numbers are the same and so make do with just one copy). So in principle, we could end up with many copies of the same integer living on the Heap.

  An exception is made in the case of *small* integers, namely those in the range 0,...,255. These small numbers arise so commonly that we wouldn't want to create a separate object e.g. for '5' every time we want to store the number 5 in memory. For this reason, at least in the standard implementation of Python, the objects for the integers 0,...,255 are *pre-allocated*: that is, they're created in some corner of Heap memory at the start of every Python session, and *every* occurrence of 5 that we want to store anywhere on the Stack or Heap will be represented as a reference to this standard, pre-allocated object. This is illustrated by the left-hand diagram below, in which a is the address for the pre-allocated object for 5.



That, at least, is officially how things are in Python. However, having reached this point, there is a helpful simplifying convention we can apply. Since *every* occurrence of 5 as the value of a variable, field or array cell will appear as the reference a, we may as well take the view that this a simply *is* our way of writing the number 5. This convention enables us to simplify our diagrams quite a bit (see the right-hand figure above), and makes the treatment of small integers look much closer to that of integers in Java—basic values that can be stored in-place within a variable or field. In this course, we will frequently adopt this simplifying assumption for the purpose of examples—for instance, in exam questions you may be asked to draw the memory configuration arising from some sequence of Python commands, making use of this 'standard simplifying assumption' regarding small integers.

Assignment by reference

Having a clear mental picture of how program data is arranged and structured in memory is indispensable for understanding certain things that are going on in a programming language. As our first example of this, here is something that applies to both Java and Python.

In both these languages, we make use all the time of *assignment* statements, which in simple cases have the form

variable `=` expression

For example, in Python:

```
s = "smiles"[1:5]
```

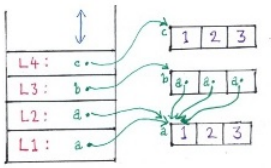
Here the expression on the right is an example of a *slice expression*: it means the substring of `"smiles"` from position 1 (inclusive) to position 5 (exclusive), where indexing starts at 0—that is, the string `"mile"`.

What happens when this assignment statement is executed? Broadly, the expression on the right side is evaluated, and the value is stored in the variable `s`. But what exactly is the 'value' of the right-hand expression? What kind of thing does it evaluate to.

What will happen will be that a new string, `"mile"`, will be created on the Heap, and then what will strictly speaking be returned as the value of the expression will be a *reference* to this new string—the address where it may be found. This, at any rate, is the 'value' that gets assigned to the variable `s`: the Stack entry for `s` won't itself directly contain the string `"mile"`. Indeed, *any* time we evaluate an expression in Python or Java, the value will, strictly speaking, be either a simple basic value like 561 or `True`, or else a *reference* to something living on the Heap. (Or, in the case of Java, the value could be the special `null` reference.)

This has some important consequences for how programs behave. Here's an example that's similar to one on the first Python lab sheet, consisting of a sequence of four assignment statements performed in turn. On the right hand side, we show the memory configuration these statements give rise to when executed:

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



First, when we execute `L1 = [1,2,3]`, a new list will be constructed on the Heap containing the entries 1,2,3, and a *reference* a to this list will be returned as the value of the expression `[1,2,3]`, and this reference will be what is stored in `L1`. Next, when we execute `L2 = L1`, the expression `L1` will be evaluated—and its value will simply be the current content of the variable `L1`, namely (again) the reference a. So this second assignment statement will simply have the effect of copying the reference a from `L1` into `L2`. Note that the list to which a points will not itself be copied: rather, the effect will be that `L1` and `L2`, since they both now contain a, will point at the very same copy of the list 1,2,3 on the Heap. Likewise, if we create a list of lists `L3 = [L1,L1,L1]`, each entry in this new list will be the current value of `L1`, which is again just the reference a. So each of the three cells in this new list will contain a reference to the original copy of 1,2,3.

This has a very important consequence. Suppose that we now modify one of the entries in `L1`, e.g. by executing `L1[2] = 5`. Then the list living on the Heap at location a will now read: 1,2,5. This means that the change will be 'seen' by `L2` and `L3`, both of which involve the reference a. For instance, if after performing the above update we now print the value of `L2`, we'll see `[1,2,5]`; and if we print the value of `L3`, we'll see `[[1,2,5,],[1,2,5],[1,2,5]]`. (You are encouraged to try out this and some similar experiments at the Python prompt to get a feel for what happens.)

Thus, assignment statements in Java and Python work purely *by reference*—only the references to Heap structures are passed around and copied, and the structures themselves won't be copied unless we explicitly request it. This means we can end up with a memory configuration containing many references to the same Heap structure, all consisting of the same address a—a phenomenon known as *aliasing*. This can lead to problems if it's not what we're intending to happen: if two data structures share some portion in common, then a modification of one data structure could unexpectedly result in visible changes to the other, if we haven't thought through whether this sharing is what we actually want. This is why it's important to have a clear grasp of the memory configurations to which our code will give rise.

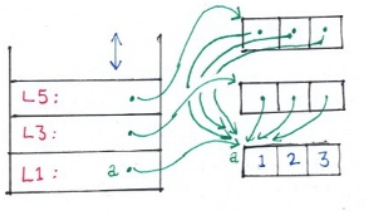
What if we *do* want to make a fresh copy of a Heap structure in itself? In the case of lists in Python, one way to do this is to use the *slice operator*, as in the command `L4 = L1[:]`. In general, `L1[i:j]` means the sublist of `L1` from index `i` (inclusive) to `j` (exclusive)—however, if `i` and `j` are omitted, the default values are 0 and the length of the list respectively, so that `L[:]` means the entire list, or rather a fresh copy of it. The evaluation of `L[:]` will thus create this fresh copy and return a reference to it, and this reference will be what is stored in `L4` (see the diagram above).

Since `L1` and `L4` now refer to distinct copies of 1,2,3 in memory, changes made to `L1` will have no effect on `L4`, and vice versa: if we execute `L1[2]=5` and then print `L4`, we'll still see 1,2,3.

The slice operation is rather specific to list-like structures in Python. In general, if we have some other kind of Heap structure that we're wishing to copy, we may need to write our own custom code to do this—in other words, to implement a *clone operation* for the structures in question. (Likewise, in Java, many kinds of objects already come with a `clone` method that returns a fresh copy of the object—but if not, we may need to implement such a method ourselves.)

One should also be aware here that different 'levels' of cloning are possible. For example, consider the following in Python:

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



Here, as before, `L1` is just the list 1,2,3 and `L3` is a list of lists, all of which happen to be the very same list `L1`. What happens when we perform the list slicing `L3[:]`?

The effect of list slicing is that a new list will be created of the same length as `L3`, then for each position index `i`, the value of `L3[i]` will be copied into the cell `L5[i]` just as if we performed an assignment `L5[i] = L3[i]`. Since, for each `i`, the value of `L3[i]` will actually be the reference a, this is what is written into `L5[i]`. The effect, then, is that each cell of `L5` will still point to the *original* copy of 1,2,3—not to a fresh clone of this list. In other words, we've made a clone of the list `L3` at 'top-level', but haven't cloned any lists deeper down in the structure to which the cells of `L3` refer. This kind of top-level clone operation is often known as *shallow cloning*. In this case, the effect that although a change to one of the elements of `L3` (e.g. `L3[0] = [2,4,6]`) won't be visible as a change to `L5`, a change to one of the *elements of elements* of `L3` will be (e.g. think what `L5` will be after we perform `L3[1][2] = 5`).

Again, this can lead to problems if it's not what we want: if this sharing of data structures isn't what we intend, unwanted aliasing effects may occur. We should therefore consider whether a shallow clone is sufficient for our purposes, or whether we're really wanting a *deep clone* that makes a completely fresh, self-contained copy of the entire structure and everything reachable from it. In this case, we may need to write a program ourselves to do the deep cloning. For complicated data structures, this is likely to be a *recursive* program, since (for example) making a deep clone of a list will involve making a deep clone of each of the elements of the list. Once again, drawing pictures of the the relevant memory structures (or at least having clear pictures of this kind in your head) is indispensable for understanding what's going on here.

Two notions of 'equality'

We turn now to another area where these issues really matter for programming purposes.

What does it mean for two data values to be *equal*? If `x` and `y` are integer variables, then we all know what it means for `x` and `y` to have the same value. But what it `x` and `y` are variables referring to arrays or lists or objects of some other kind that live on the Heap?

There are two possibilities here. Remember that in this situation, the values of `x` and `y` will strictly speaking be *addresses*. So we might be asking: are they the very same address a? In other words, are `x` and `y` pointing to the very same occurrence of some data structure at some location in the Heap? And if that's true, then we say `x` and `y` are equal 'by reference'.

Or we might be asking not whether `x` and `y` point to exactly the same location in memory, but whether what we find at the location a pointed to by `x` is identical to what we find at the location b pointed to by `y`. So it could be the Heap contains two identical copies of some structure (such as a list 1,2,3), and `x` points to one of them while `y` points to the other. And in that case, we'd say `x` and `y` are equal *by value*.[3] Notice that if `x` and `y` are equal by reference, they'll certainly be equal by value, but the converse isn't true.

Both of these kinds of equality are important in programming languages. In Python, the keyword `is` (as in `x is y`) can be used to test for reference equality, while `==` tests for value equality. As an exercise, in the left column below are some Python commands repeated from our earlier examples—you may like to draw the memory configurations resulting from these (ideally without looking back in these notes), then try to predict what each of the expressions in the right column will evaluate to. (You're encouraged to check your answers at the Python prompt, and generally to do some experiments to get a feel for how things work.)

```
L1 = [1,2,3]                    L2 is L1
L2 = L1                         L4 is L1
L3 = [L1,L1,L1]                 L2 == L1
L4 = L1[:]                      L4 == L1
L5 = L3[:]                      L6 is L3
L6 = [L1[:],L1[:],L1[:]]        L6 == L3
```

There's a particular subtlety concerning the very last expression, `L6 == L3`. Since these are lists of lists of integers, one can draw a distinction here between *shallow* and *deep* equality by value, analogous to the distinction between shallow and deep cloning. With shallow equality, we'd only look at the 'top level' in the Heap structure: does the structure referenced by `L6` literally match the one referenced by `L3`? This would only be the case if, for each index `i`, the content of `L6[i]` was literally equal (in the strong sense of equality by reference) to the content of `L3[i]`. This would not be the case in the above example, since for instance `L3[0]` and `L6[1]` contain different addresses a, b. On the other hand, with *deep* equality we would explore further, and ask whether what is found at location a matches what's found at b—and so on through as many levels as are present in the structures in question. In the above example, this would indeed

[3] There's admittedly a bit of confusion around the use of the term 'value'. We've already emphasized that, strictly speaking, the *value* of an expression of reference type will be a reference a rather than the thing it points at. Nonetheless, in the context of the phrase 'by value', the *value* is commonly understood to be the entity on the Heap that is pointed at. For a brief discussion of this terminological confusion, see the Wikipedia page for 'Evaluation strategy', especially the short paragraph on 'Semantic drift'.

be the case, since both `L6` and `L3` correspond to representations of the same list of lists: `[[1,2,3], [1,2,3], [1,2,3]]`. In Python, the rule is that `==` tests for *deep equality* in this sense, so that in our example, `L6 == L3` would evaluate to `True`.

The distinction between equality by reference and by value is also relevant in Java. Just to confuse you, though, the symbol `==` in Java is used for *reference equality*, while equality by value is usually tested using a method called `.equals` for the objects in question (as in `x.equals(y)`). Many existing classes in Java already come with such a method, and if you're creating your own Java class, you're free to write your own `.equals` method for it, which captures the precise kind of equality you most care about (e.g. shallow or deep or something in between).

As a final note, in both Python and Java, the issue of equality for *strings* is especially subtle. For instance, we may have a situation in which both `x` and `y` refer to copies of the string `"mile"`—but when will they refer to the same copy of this string, and when will there be two different copies. The rules for this are pretty complicated and may even be dependent on the language implementation, so the main takeaway is that you basically shouldn't write programs that rely on a specific answer to this question. In other words, don't use reference equality for strings at all—it's always safer to use value equality (`==` in Python, `.equals` in Java) which tests whether your strings are identical or different, irrespective of where they live in memory.

Dereferencing and NullPointerExceptions

A third important application of our memory diagrams is in understanding the process of *dereferencing* in Java, and in particular how and when those pesky `NullPointerException`s arise.

Let's suppose we have any expression in Java of the form

expr `.` *fieldname*

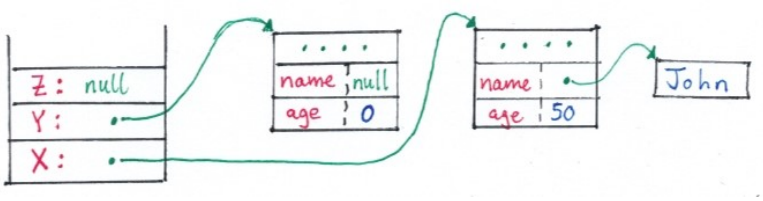
where *expr* is an expression which we expect to evaluate to a reference to an object of some class `C`, and *fieldname* is the name of some field possessed by objects of this class. For example, our Java expression might be `X.age`, where `X` is a variable of type `C`.

We've already made the point that when we evaluate *expr*, what we get, strictly speaking, is some *reference a* to an object on the Heap, or possibly the special reference `null`: in any case, not the object on the Heap itself. However, the value of the field we're interested in *does* live on the Heap as a component of this object, so it's clear that we'll need to follow the reference *a* to see what it's pointing at: a step that's often known as *dereferencing*. Conceptually, the way to think of it is that it's when we pass the '`.`' in our Java expression that the dereferencing occurs—and then, having arrived at the object that's pointed to, we can extract the field value we want. Thus, if our *expr* happens to evaluate to `null`, it's when we pass the '`.`' and attempt to dereference the `null` that a `NullPointerException` will be thrown.

Crucially, it's *not an error in itself* for an expression to evaluate to `null`: indeed, we'd be in deep trouble if it were, as we wouldn't even be able to use idioms like `if (X != null) { ... }` to guard against the possibility of null

values: even the evaluation of the condition here would throw and error! Rather, it's when we try to dereference the `null` to see where it points to—when we pass the relevant '`.`'—that the error will be thrown. Exactly same applies when we use a '`.`' to access a *method* associated with an object, as in `X.length()`: it's when we pass the dot that we risk an error if `X` has in fact evaluated to `null`.

This has the consequence that any time you get a `NullPointerException`, there will be one specific occurrence of '`.`' within your program that's causing it. Understanding this can be very helpful for debugging, since drawing the relevant pictures can help us to pinpoint which '`.`' is the culprit. For example, suppose we have three Stack variables `X`, `Y`, `Z`, all of type `Person`, where `Person` is a class with a `name` field of type string and an `age` field of type integer. Suppose, in particular, that we have the following memory configuration:



Here, we see that:

- The evaluation of `X.name.length()` will succeed, and will return 4.
- The evaluation of `Y.name` will also succeed—it will return `null`, but that's fine. However, the evaluation of `Y.name.length()` will throw a `NullPointerException`, triggered by the second of the two dots.
- The evaluation of `Z` will succeed, returning `null`. However, the evaluation of `Z.name` will attempt to dereference this `null` and will throw an exception. So we won't even get as far as being able to try evaluating `Z.name.length()`.

Basic memory operations and their cost

We've now seen several examples of how an understanding of how program memory works can be very helpful when writing and debugging programs. However, in the context of this course, we're also wanting to lay a foundation for constructing our own algorithms and data structures on the foundation of the memory model I've described, and for reasoning about their efficiency. Just as in Lecture 5, we will be interested in gauging the runtime of a piece of code by the number of line executions or basic 'instructions' it executes—so it's worth taking stock of our memory model at this point to see what kinds of operations on memory we're willing to count as taking just a single 'instruction' or computation step—i.e. what we're willing to assume can be done within *constant (i.e.* $\Theta(1)$*) time*.

1. First of all, *reading* or accessing the value of a program variable, whether of basic type or reference type, is certainly something that can be done

in constant time, as we are simply having to look up the contents of the relevant Stack entry.[4]

2. Likewise, *writing* a known value to a program variable, whether of basic or reference type, will also work within $\Theta(1)$ time, as we're simply having to update the contents of the relevant Stack entry. This is illustrated by assignment statements like `n = 341` or `Y = X` (where `X`,`Y` are of reference type). Note especially that in the latter case, the assignment works by reference: we are simply copying a *reference* from `X` to `Y`, which clearly takes just constant time, independent of the size of the Heap structure that `X` points to. If we were copying the Heap structure itself, we wouldn't be able to put any constant upper bound on the time this might take—for instance, we might be dealing with a string or array of any size n—but that's not what a simple assignment statement is doing.

3. Suppose now we wish to read, or write to, a known field in some object to which we already have a reference: for example, to access the value of `X.age`, or to assign the value `X.age = 5`. This will involve dereferencing our reference to find the Heap object in question, then a bit of simple navigation to find our way to where the field in question is stored. Again, we can expect to do all of this within constant time. If we are writing to a field of reference type, e.g. as in `X.name = s`, then as before, we're simply copying the *reference* to `s` into the relevant field, not copying the string `s` itself, so everything works within $\Theta(1)$ time.

4. Suppose we wish to access a cell at a known index in an *array* for which we already have a reference: e.g. `A[42]`. We will then need to add a suitable offset to our reference a to obtain the expected location of the cell we want—this might be something like $a + 4 + (2 \times 42)$—then look in this address to find the value. All this, again, is constant time, and the same goes if we wish to write a new value into an array cell, e.g. `A[42]=5`.

5. A further thing we need to think about is the allocation of new Heap structures: for example, the creation of a new object of some specified class such as `Person`. Here, the process is a little more complex. First, we have to identify a portion of Heap memory of the required size that is currently free and available for use (it will be the job of the *memory management* system to keep track of which parts of Heap memory are free and which are not). We then wish to set aside this portion of memory for our new object, registering it as now 'in use'. Next, there may be some initialization phase in which the new portion of memory is formatted correctly and appropriate header information added. Although all of this may seem quite complicated, for practical purposes we shall be happy to assume that all these phases can be completed within $\Theta(1)$ time.

   Finally, we may need to initialize the contents of the object's fields to suitable default values. For any specified class such as `Person`, there will of course be only a constant number of such fields; and as long as their types are not too complex, it is probably realistic to assume that all this,

[4] You might wonder also about the time taken to locate the relevant Stack entry, given a variable name such as `n` or `X`. However, if our compiler is doing a reasonable job, we won't have to do more than $\Theta(1)$ work at runtime to figure out where to find this Stack entry.

too, can be done in $\Theta(1)$ time. Thus, we can suppose that the allocation of a new object *of a specified class* can be accomplished within constant time overall.

6. What about the allocation of a new *array* of size n? Once again, we will ask our memory manager to identify and set aside a suitable portion of memory, and we'll assume that this can be done within constant time, independent of the size n and the amount of memory it calls for. This is because merely recording that, say, all of memory from address 0xED150000 to 0xED15FFFF is to be set aside for the array `A` doesn't in itself require anything to be done to each of these words individually.

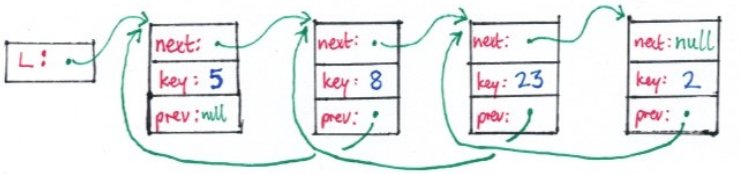
   However, we now come to the question of whether the values of the array cells are to be *initialized*, perhaps to some default value such as 0. This partly depends on the language in question. In C, for instance, it's possible to allocate space for an array without bothering to initialize its entries—this means that if we create an array `A` and then try to access, say `A[42]` before we've written anything into this cell, what we get is liable to be some random content left over from whatever that portion of memory was last used for. Other languages, such as Java, steer clear of such unpredicatable behaviour by insisting that the creation of a new array involves the initialization of its contents, either to default values or to values specified by the programmer. In this case, the initialization phase for an array of size n can be expected to take time $\Theta(n)$. When formulating an algorithm that involves the creation of arrays, therefore, we may need to pay attention to the question of whether these arrays come with an initialization cost.

A quick glance at linked lists

We have already briefly mentioned *linked lists* as one of the Heap data structures illustrated in our big picture on page 2. Let's conclude our survey of the contents of program memory with a little more about these.

As we've seen, the idea is that a linked list consists of separate cells that individually can live anywhere in memory, each one storing a single list item, and where each cell is linked to the next one by means of a reference to its memory location. This means that, starting from the first cell in the list, we can find all the others by repeatedly chasing the pointers. We can also suppose that the last cell in the linked list has a null reference, meaning it doesn't point to anything else. In a *singly linked* list, this is essentially all the structure we have; in a *doubly linked* list, we also have references in the other direction allowing us to get from each cell to the previous cell in the list (if there is one).

Here is a diagram of how doubly linked lists might look if implemented in Java or Python:



Here we're supposing that cells are represented as *objects* of some class `Cell`, each containing three fields: a `key` field containing the main payload of the cell (that is, the list item that the cell is holding), and two fields `next` and `prev`, both themselves of type `Cell`, which contain references to the next and previous cells. A linked list as a whole can then be specified simply by a reference to its first element (or *head element*), so the same type `Cell` can be thought of as both the type of linked lists as a whole and of individual cells within them. This would be a way in which you could implement (singly or doubly) linked lists for yourself in Java or Python—although of course, it would be easy to find existing library implementations in which this kind of thing is already implemented for you.

What about functional languages like Haskell? Here, linked lists are typically used absolutely all the time: indeed so central are they to functional programming, that one of the early functional languages was called Lisp, which is short for *list processing*. The built-in default notion of lists in most functional languages is some version of singly linked lists. And because these lists are native to these languages—one doesn't have to define a special class of objects for them—there is often some special syntax associated with linked lists in functional languages. For example, the operation of taking the first element of a list `L`, which would be written in the above setting as `L.key`, might be written in a functional language as something like `head(L)`. Likewise, the operation of discarding the first element and taking the linked list of all the remaining elements, written in our object-oriented setting as `L.next`, might be written in functional style as `tail(L)`. Another very common operation is that of creating a new list by adding an element `x` onto the front of an existing list `L`: this might appear in object-oriented syntax e.g. as `new Cell(x,L)`, or in functional syntax as `cons(x,L)` or even just `x:L`. Despite the superficial differences in syntax, however, what is going on under the hood in program memory will be very similar for both functional and object-oriented languages.

Although both arrays and linked lists can serve as ways of storing lists of items, they are obviously very different from an efficiency point of view: for example, finding the nth element in an array, as we've noted above, is a $\Theta(1)$ time operation, whereas for a linked list this takes time $\Theta(n)$ as we have no choice but to chase the pointers down the linked list for n steps. In the next lecture, we'll investigate this kind of thing more systematically, also looking at more specialized kinds of lists such as *stacks* and *queues*, and will look in more depth at the pros and cons of different representations.