Informatics 2: Introduction to Algorithms and Data Structures
Notes on Lecture 7
Classic data structures: Lists, stacks, queues

John Longley
© University of Edinburgh 2026

In Lecture 6, we looked at how program data is organized in memory. Among many other things, we touched on different ways in which emphlists can be represented: the things that are called 'lists' in Python are actually implemented using arrays, whereas the things called 'lists' in Haskell are implemented as linked lists. In this lecture, we'll look at this more systematically, and discuss the pros and cons of different implementations in terms of the efficiency of the various operations we want to perform. We'll then do a similar thing for two other classic data types called *stacks* and *queues*—both of which we can think of as lists that are only used in certain restricted ways.

Lists and list operations: an abstract view

How might we go about comparing different implementations of lists? One place to start would be simply to write down a (meta-)list of the basic operations that we might expect any decent implementation of lists to provide. We can then enquire about the efficiency of those operations for different implementations. Here, for example, are some obvious basic operations on lists. (I've given them certain names here, but many of them might have their own special syntax in particular programming languages by which you'll be more familiar with them.) For simplicity, we're thinking here about lists whose items all have some type X: we can have lists of integers, lists of strings, and so on.

- First, we'd like to be able to look up the item at a given position i in a list (remembering that indexing is conventionally from 0). Since this operation takes an integer i as argument and returns an element of the list, we can consider it as an operation

  $$\mathbf{get} \ : \ \mathtt{int} \to X \ .$$

  Here we're thinking of lists as something like *objects* in Java, and of **get** as a *method* associated with a list: if L is a list, then in Java-style syntax we might write $L.\mathbf{get}[5]$ as an expression that returns the element of L at index 5. In Python, this would be written much more briefly as $L[5]$; however, it will be helpful for our purposes to think of the various list operations as methods associated with lists.

- We also have a 'write' operation corresponding to what in Python would appear as '$L[i] = x$':

  $$\textbf{set} \ : \ \texttt{int} * X \to \texttt{void} \, .$$

  As the type suggests, this takes both an integer i and a potential list item x, and writes the value of x into L at position i, overwriting whatever was there before, and leaving the length of the list unchanged. Note that the **set** operation doesn't *return* any particular information—we've indicated this by using a Java-style `void` as the return type.

- We might also want two operations for adding a new item to a list, without losing any of the existing items:

  $$\textbf{cons} \ : \ X \to \texttt{void} \, , \qquad \textbf{append} \ : \ X \to \texttt{void} \, .$$

  The difference between these is that **cons**(x) will add the new element at the *start* of the list, so that x then appears at index 0, whereas **append**(x) adds x at the *end* of the list. (Note that **cons** is the name often given to this operation in functional programming.) In either case, the length of the list will increase by 1.

- More generally, we might expect an operation

  $$\textbf{insert} \ : \ \texttt{int} * X \to \texttt{void} \, ,$$

  such that **insert**(i, x) inserts the new item x at index i without overwriting any existing items (this will increase the length by 1). Related to this, we might have an operation

  $$\textbf{delete} \ : \ \texttt{int} \to \texttt{void} \, ,$$

  where **delete**(i) deletes the element at index i. Assuming i is a valid index for the list, this will decrease the length by 1.

- Finally, we'd expect an operation

  $$\textbf{length} \ : \ \texttt{void} \to \texttt{int}$$

  for asking a list what its length is. The `void` represents the fact that no argument information is required: in Java-style notation, we'd write just L.**length**().

Collectively, these amount to something much like an *interface* in Java: we're saying what methods ought to be present in an implementation of lists, and specifying their argument and return types, but we're not at this point giving any actual implementations of them. This is a useful way of thinking about list operations regardless of whether our programming language actually has such things as 'interfaces',[1] or whether lists are officially 'objects' in the language in question.

[1] Python doesn't have them, for instance, though the machinery of *abstract base classes* plays a somewhat similar role.

There's nothing very special about the particular selection of list operations I've chosen above: We could have added others, such as a **reverse** operation on lists, or contented ourselves with fewer operations. For instance, we might observe that in the presence of **insert**, both **cons** and **append** are in one sense unncessary, since one could simply define

$$\begin{aligned} \mathbf{cons}(x) &\equiv \mathbf{insert}(0, x)\ , \\ \mathbf{append}(x) &\equiv \mathbf{insert}(\mathbf{length}(), x)\ . \end{aligned}$$

On the other hand, we might lose something if we do that, because it's possible that a particular implementation might implement **cons** or **append** in some way that's more efficient than via the general **insert** operation. So we might want to include **cons** and **append** in our interface as basic operations in their own right to allow for this possibility.

The main point is that we're distinguishing between two perspectives:

- A general *abstract* view of lists as embodied by our interface above, where we specify what operations we expect lists to have without committing ourselves to any specific way of representing lists in memory or of implementing the operations.
- A *concrete* view in which we consider some specific way of implementing lists which supplies implementations of the operations mentioned in the interface.

We'll now proceed to look at three possible concrete implementations for the abstract list interface given above.

List implementation 1: Fixed-size arrays

Our first implementation is extremely simple. Suppose we create an array A of some fixed size m. Then any list of items $x_0, \ldots, x_{n-1}$ where $n \leq m$ can obviously be stored in A: we simply set $A[i] = x_i$ for $i = 0, \ldots, n-1$. The list $x_0, \ldots, x_{n-1}$ itself can then be represented by the pair (A, n), where n records how much of A the list takes up (which we wouldn't be able to tell just from A alone).

Given this choice of representation, it's easy to see how to implement our various list operations, to the extent that this is possible given the size limit m. Here are implementations of some of them:

```
get(i):              append(x):          insert (i,x):
    return A[i]          A[n] = x            for j = n−1 downto i
                         n = n+1                 A[j+1] = A[j]
                                             A[i] = x
                                             n = n+1
```

The code for **get** speaks for itself. For **append** we note that n records the number of cells in use so far: these are at positions $0, \ldots, n-1$, so the next available cell is at position n. So to add an item x at the end of the list, we simply put it in cell n and increment the value of n. Clearly, there'll be a problem if the whole of A is already in use, that is, if $n = m$, so that there is no room for a new item; otherwise, this simple implementation will work fine. For

insert(i,x), the idea is that we want to insert the new item x at index i, and this means that all the existing items at indices from i upwards will need to be shifted one place to the right to make room for it (again, assuming our array isn't already full). This is the purpose of the for-loop in the above code. The remaining operations **set**, **cons**, **delete**, **length** are likewise easy to implement: we leave this as an exercise.

We're now in a position to ask: how 'efficient' are each of these operations? Or more specifically, what is their asymptotic runtime as a function of the list length n? We now have a strong basis for answering this, because in the previous lecture we noted the basic memory operations which we're happy to regard as working within constant time (i.e. in $\Theta(1)$ time), and these basic operations included reading and writing the values of array cells. So it's now easy to see that both **get** and **append** will work in $\Theta(1)$ time, since they're performing just a small, fixed number of simple line executions—at least, as long as **append** works at all and doesn't overflow the array. (Likewise, it's easy to see that the obvious implementations of **set** and **length** will work within $\Theta(1)$ time.)

By contrast, we can see that **insert** will have to do some more substantial work, as we need to shift $n - i$ elements to the right. The worst case is clearly when $i = 0$, so we see that **insert** will take $\Theta(n)$ time in the worst case. The situation for **cons** and **delete** is similar.

Fixed-size arrays are a very simple way of implementing lists, but they do have advantages in some situations where we know, in advance, some bound m on the length of all the lists we'll ever be dealing with:

- Because the read and write operations for array cells are themselves typically fast, we'll have very fast and simple **get** and **set** operations.
- This will be particularly true in languages such as C where fixed-size arrays can be stored on the stack (see Lecture 6). In this case, we can eliminate the step of having to follow a pointer to the heap to locate the array we're looking for.
- More generally, having arrays of a known, fixed size is good for memory management. For instance, if we can keep such an array on the stack, we'll reclaim the memory space it occupies as soon as the array variable goes out of scope (i.e. reaches the end of its lifetime). We don't need to rely on the memory recycler detecting the the space is no longer in use and then reclaiming it.

However, the disadvantage of fixed-size arrays is very obvious: they have fixed size! That is, we have to decide in advance on a fixed upper limit to the length of each list we allocate, and one we've reached that limit, there'll be no way to add more items. This is particularly an issue if we're dealing with a large number of lists whose sizes may vary widely (in ways not predictable in advance). If the size limit m we choose is too small, there's a chance that some of our lists will overflow; but if we choose m large enough to cater for even the longest lists, we will waste large amounts of memory for those lists that never come anywhere near this length. So while fixed-size arrays might be a good choice in certain special contexts, they're not a good choice for general-purpose lists. This leads us to our second way of implementing lists.

© *University of Edinburgh 2026*

List implementation 2: Extensible arrays

The basic idea of extensible arrays is again very simple: we work with array representations of lists as before, but if our array A overflows, we simply replace it by a bigger one and carry on.

This would be very easy and cheap to do if we happened to know that the memory space appearing immediately after A (on the heap, say) was free: we could then just set aside a bit more space for A than we did at first, and continue as normal. In general, however, we can't assume this: the memory space immediately after A might *not* be free; and even if it is, detecting this fact might not be so easy. So in general, whenever we try to add a new element to a list whose carrier array is already full, we must be prepared to make a completely fresh array B somewhere else on the heap, where B is larger than A, then *copy* the current contents of A into B.

A common approach to this is to have some fixed *expansion factor* $r > 1$, which gives the ratio by which we expand our array each time we do this. For example, if we choose the expansion factor $r = 2$, then our new array B will be twice as large as A. This should be enough to keep us going for a while; but if B itself later becomes full, we may need to expand again to an array C twice the size of B (i.e. four times the size of the original A). Or, if we want to be more thrifty in our use of memory, we might take a smaller expansion factor, say $r = 1.2$, and then B would be bigger than A by only that factor (rounded to an integer if necessary).

The essential idea is illustrated by the following pseudocode for **append**:

```
append (x):
    if n = |A|
        B = new array (⌈ n × r ⌉)
        copy contents of A into B (n items)
        A = B
    # Now do ordinary append:
    A[n] = x
    n = n+1
```

Note that if our array A isn't already full—i.e. if n is less than the size of A—this will behave just like the ordinary **append** for fixed-size arrays. Otherwise, we allocate a new array B, larger than A by the chosen factor r, and copy each existing element $A[i]$ into $B[i]$. The line $A = B$ then performs an assignment by reference: it copies the *reference to* B into the variable A, so that the new array can still be accessed by the variable A as before; the variable B here is playing only a temporary role. (Recall that this assignment by reference is just a cheap constant-time operation and doesn't *itself* make a fresh copy of B: all the actual copying is done explicitly by the preceding lines of pseudocode.) Finally, the last two lines will simply add the new item x into the newly expanded array as normal and update the length variable n.

How long does this **append operation** take to execute? If it's an ordinary append that doesn't trigger an expansion, only three lines executions will have, so again this works in $\Theta(1)$ time. But from time to time, we'll get a 'bad' append where we have to expand and copy—and because we have n list items to copy across, where n the current length of the list, it's clear that the total runtime will be $\Theta(n)$.

This might seem at first like a rather crude and dirty solution to the problem of array overflow. But it's quite widely used in practice. And there's something interesting and perhaps surprising to say about it, which emerges when we try to analyse its runtime. We'll look at this next.

Amortized cost

We've noted that most **append** operations will be fast, but just occasionally we'll get a slow one when an expand-and-copy is triggered. How much does this matter?

This will depend on the application. There may be some highly time-critical real-time applications for which it's essential that our **append** *always* completes within some small number of microseconds. In such a context, even a single 'bad append' could be fatal, so we probably wouldn't opt for an extensible array implementation at all. However, for many other applications, it might not matter if an append occasionally takes a long time, as long as the *average* time taken over a long sequence of append operations is reasonable. In other words, a bad append might seem quite acceptable if we think of its cost as spread out, or as we say here, *amortized*, over the following run of good appends. That is, we can think of a costly array expansion as an investment of effort that pays for itself over time, because it makes possible a lot of good appends straight after it.

This raises the question: what *is* the amortized cost of a bad append if considered as part of a long sequence of appends? Can we quantify the economics of what's going on here and what the overall cost of appends is, if we consider the cost of the bad ones as spread out over the good ones? The answer is perhaps unexpected.

Let's consider what happens if we start with the empty list, represented by a carrier array with some initial capacity a, then do some large number m of append operations in succession, expanding our array by a factor of r when necessary. How will the size of the array grow, and how many expansions will be needed. Ignoring tiresome rounding issues, we'll have an array of size ar after the first expansion, then ar^2 after the second expansion, then ar^3, and so on. So if we wish to accommodate m items, the number of expansion steps will be the number s such that $ar^s \geq m$ (that is, s expansions are enough), but $ar^{s-1} < m$ (that is $s-1$ expansions aren't enough). Rewriting these inequalities as $r^{s-1} < m/a \leq r^s$, and taking logs to base r throughout, we see that this number s is $\lceil \log_r(m/a) \rceil$. This could be quite a sizeable number if m is much larger than a and if the expansion factor r is only a little above 1: for example, if $a = 100$, $m = 5000$ and $r = 1.1$, we get $s = \lceil \log_{1.1} 50 = 42$. So the number of expand-and-copy phases involved could be quite significant.

This may suggest the following thought. Consider what happens to those items that were inserted into the array early on, before the first expansion. Each of these items will have to be copied every time we do an expansion, so will be copied s times in the course of appending m items. So it may seem that the efficiency of this whole scheme will gradually degrade as we do more and more appends: the more we do, the more times each of these items will have to get copied, so the *average* time cost per append (defined as the total time for m appends divided by m) will slowly increase, because of the gradual creeping up of the number of copyings per item that we have to do.

That, at least, is what we might intuitively expect: on a run of m appends, the amortized time cost per append will increase asymptotically with m. But is this actually what happens? Let's do the sums and find out.

Amortized time analysis of append

We can get the idea by looking at a specific example. We'll work with the figures mentioned above: $a = 100$, $m = 5000$, $r = 1.1$, from which it follows that $s = 42$. What will be the total cost of this run of 5000 appends, measured (say) by the number of times we'll need to copy some individual item from our 'A' array to our 'B' array (in the notation of the above pseudocode)?

We can work this out as follows. On the first expansion, we'll be expanding from an array of size 100 to one of size $100 * r = 110$, and will be copying across the 100 items from the first array. Then on the second expansion, we'll be expanding from 110 to $100 * r^2 = 121$, and will copy 110 items. On the third expansion, we'll copy 121 items, and so on. So ignoring rounding issues, the total number of item copyings in the course of all 42 expansions will be

$$100 \times (1 + 1.1 + 1.1^2 + \cdots + 1.1^{41}) \; .$$

The expression in brackets is the sum of a geometric progression. As you may recall, there's a formula for this, so that the above expression reduces to

$$100 \times (1.1^{42} - 1)/(1.1 - 1) \; .$$

Note that 1.1^{42} will be the final array size, which is large enough to accommodate all m items, though it won't be *much* larger than m, because we've noted that 1.1^{41} wasn't big enough. Indeed, from the inequality $1.1^{41} < m$ we can infer that $1.1^{42} - 1 < 1.1^{42} < 1.1m$, which allows us to give a simple upper bound on our total number of copyings:

$$100 \times (1.1^{42} - 1)/(1.1 - 1) \;<\; 1.1m/0.1 \;=\; 11m \; .$$

In other words, although *some* items will get copied as many as 42 times, the *average* number of copyings per item will remain below 11. Note that the '11' here is derived only from r (in fact it is $r/(r-1) = 1.1/0.1$), and is independent of the number m of appends or the number s of expansions they trigger. So even if m were made so large that 100 expansions were needed, so that some items were copied as many as 100 times, the average number of copyings per item (or per append operation) would never rise above 11.

This may seem counterintuitive. But one way to think about it is that although the *maximum* number of times an item may get copied can grow without limit—in our example it's 42, but it could get even bigger—it's only a tiny and negligible proportion of the items that do get copied that many times. So these items won't have a significant effect on the *average* number of copyings. We have been through the reasoning here with some specific numbers, but if we work through the algebra, we can see that in general, the average number of copyings stays below $r/(r-1)$, however large m becomes.

We've reached this conclusion mathematically by summing a geometric progression. But having obtained the answer, there's a more conceptual argument for justifying it which you might prefer. For this, let's imagine that copying a

single item from A to B costs us 1p to do. And let's suppose that each time we do an append, we're required to pay 11p into a pension fund which is intended to pay for future copyings. (We're taking advantage of the fact that we already know that 11 is an upper bound on the average number of copyings per item.) How will this work out? Let's pick up the story at the point immediately after we've done our first expansion, so that we have an array of 110 cells of which 100 are currently in use. We can then do 10 appends before our next expansion, and this will pay £1.10 into the fund. But this is precisely enough to pay for our second expansion, which will involve 110 copyings. We then have an array of size 121, with 110 cells currently in use, so can now do 11 more appends before the next expansion. This will pay £1.21 into the fund, which is precisely enough to pay for the third expansion ... and so on. Clearly, we can keep on doing this for as long as we live: if we think of each append as incurring a cost of 11 copyings, the economics works out and we'll continue to break even. This is why, in this example, the amortized number of copyings per append operation will never exceed 11.

Let's sum up now. Over a long run of m append operations, the total time we spend doing expanding and copying will be $O(m)$, as shown by the above arguments. To complete our analysis, we should also look at the time taken by the ordinary appends. Because an ordinary append takes $\Theta(1)$ time, the total time for m ordinary appends will also be $O(m)$. So the *total* time cost for m appends will be $O(m) + O(m) = O(m)$: that is, an amortized cost of $O(1)$ per append operation, independently of m. We can read this as saying that the overall efficiency of this extensible array implementation will remain acceptable even if the number of expansions we have to do is very very large: the effiicency of the scheme *doesn't* degrade indefinitely as we might at first have thought.

Of course, we're hiding something here. Saying that the average time is $O(1)$ is saying that it's bounded above by a constant, but we're not saying what the constant is. In our example, we saw that the constant was quite large: at least 11 operations' worth, for example. And of course, the closer our ratio r is to 1, the larger the constant will be, and this is certainly something we want to keep an eye on. So it's more honest, perhaps, to say that the amortized time cost is $O(1)$, but with a potentially large hidden constant.

As a small cultural aside, an inspection of the underlying C code for the implementation of lists in (the standard C implementation of) Python reveals that Python lists are indeed implemented via extensible arrays, essentially with an expansion factor of $9/8 = 1.125$. There is also a further level of sophistication: if we delete a lot of list items so that the proportion of the array in use dips below $1/2$, we can likewise *shrink* our array so as to free up memory space (that is, we create a smaller array and copy the existing items across). The amortized analysis of this more complicated scenario is discussed in Chapter 16 of the CLRS textbook if you're interested (Chapter 17 in earlier editions).

Java also offers an implementation of lists, called `ArrayList`, that uses extensible arrays. In this case, the documentation for this class doesn't prescribe any specific expansion policy—for instance, it doesn't say what the expansion factor should be, so it allows for different implementations of Java to make different choices here. But the definition does prescribe that however the class is implemented, the amortized cost over a long run of appends must be $O(1)$ per operation.

© University of Edinburgh 2026

List implementation 3: Linked lists

A quite different way of implementing lists concretely is as *(singly) linked lists*. We've already discussed in the previous lecture what these are, so let's focus here on the various list operations and their asymptotic efficiency.

The big difference from array representations is that to locate the cell at position i in the list (e.g. for the purpose of evaluating **get**(i) or **set**(i, x)), we have no choice but to start at the head of the list and follow the references for i steps to find the cell we want. For a list of length n, this will take $\Theta(n)$ time in the worst case (although with small hidden constants, as the little for-loop that does the pointer-chasing can be made very fast). Having found the right cell, of course, it then takes just $\Theta(1)$ to read or write the key value for that cell. The situation will be similar for **insert**(i, x) and **delete**(i), because in both cases we need to start by finding our way to position $i - 1$ in the list—although having done so, the remainder of these operations will work within $\Theta(1)$ time.

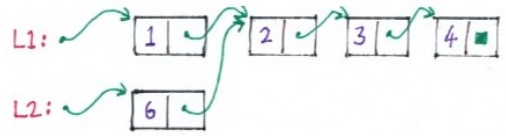
On the other hand, to implement **cons**, we simply have to construct a single new cell holding the relevant key and set some pointers so as to add it to the front of our list. This requires just a fixed, constant number of basic operations, so $\Theta(1)$ time—with a small hidden constant—in contrast to the $\Theta(n)$ time for array representations, in which we have to shuffle all the existing items along in order to make room for the new one at the start. We can also implement **append** in $\Theta(1)$ time if, as part of our linked list structure, we also maintain a reference to the *last* cell in the list.

Here, for ease of comparison, is a summary of what we've learned about the runtimes of list operations for array and linked list representations. The growth rates given here are for the *worst-case* runtimes. The asterisks indicate operations that may potentially fail for fixed-size arrays, or trigger an expansion for extensible ones.

Operation	Array impl	Linked-list impl
get	$O(1)$	$O(n)$
set	$O(1)$	$O(n)$
cons	* $O(n)$	$O(1)$
append	* $O(n)$ (amortized $O(1)$)	$O(1)$ (using ref to last element)
insert	* $O(n)$	$O(n)$
delete	$O(n)$	$O(n)$

So broadly speaking, we can see that for either of these implementations, some operations are cheap ($\Theta(1)$) while others are costly ($\Theta(n)$). This might lead us to ask whether there could be some alternative implementation of lists that performs well across the board, so that *all* list operations are cheap. The answer to this may be far from obvious, but in Lecture 9 and Tutorial 3 we'll see how the ideas of *balanced trees* give rise to an implementation of lists for which all the above operations have worst-case time $\Theta(\lg n)$.

Returning to the implementations we already have, It might initially appear from the above table that linked lists don't have very much going for them. However, we'll see below that they're actually a very respectable choice in certain contexts such as the implementation of *stacks* and *queues*. And there's also another advantage of linked lists which we haven't touched on yet. If we've got one linked list that is constructed from another in such a way that only the items near the start are different, then the two lists can actually *share* cells:



So if we already have the first list and want to create the second, we avoid having to make a copy of most of the list, which will mean a saving in both time and space. This isn't something we can naturally do with arrays, because of the way array cells are placed consecutively in memory. This is one part of the reason why functional programmers like linked lists.

Exercise (somewhat hard): Suppose, for a given n, we want to create the list of all 2^n possible binary lists of length n E.g. for $n = 3$:

```
[[0,0,0], [0,0,1], [0,1,0], [0,1,1], [1,0,0], [1,0,1], [1,1,0], [1,1,1]]
```

This will be a list of lists, and if constructed in the obvious way using array representation, the space it occupies will be $\Theta(n \times 2^n)$. Show, however, that with a linked list representation, by taking advantage of sharing we can construct this list using just $\Theta(2^n)$ space—an asymptotic saving of a factor of n.

Stacks and queues

We now turn to two data structures that are essentially just lists that used in certain restricted ways:

- If we only ever want to be able to add new items at one end of the list, and only want to read or remove items from the *same* end, we have what is known as a *stack*. The operation of adding an item to a stack is traditionally known as *pushing* the item, and that of removing an item is traditionally known as *popping* it. A stack is also known as a last-in first-out (LIFO) buffer, since whenever we pop an item, it will be the item at the top of the stack, and hence the current stack item that was most recently pushed.

- If we only want to add new items at one end of the list (the 'back') and read/remove them from the other end (the 'front'), we have what is known as a *queue*. The add and remove operations for queues are again sometimes known as push and pop, although here I'll call them *enqueue* and *dequeue*. A queue is also known as a first-in first-out (FIFO) buffer: any item we remove will be the item in the queue that has been there the longest. So it's just like a queue of people waiting to be served in a shop.

Since both of these are just lists used in certain ways, it's easy to see that *any* implementation of lists will give us a way to implement stacks and queues: e.g. it's a simple exercise to construct implementations of 'push' and 'pop' using operations given by our abstract list interface. However, if we *know* in advance that a given list is only going to be used as a stack or queue, this may affect our choice of implementation: in particular, we'll only care about how efficient the stack or queue operations are, not about list operations in general. So it's worth giving some thought to stacks and queues as data structures in their own right.

To clarify what we're doing here, it's helpful to give abstract interfaces specifying the basic operations we want stacks and queues to support, analogous to our earlier abstract interface for lists:

Stacks:		**Queues:**	
empty	: `void` → `bool`	**empty**	: `void` → `bool`
push	: $X \to$ `void`	**enqueue**	: $X \to$ `void`
peek	: `void` → X	**peek**	: `void` → X
pop	: `void` → X	**dequeue**	: `void` → X

The meanings of **push**, **pop**, **enqueue**, **dequeue** are as explained above. We also have operations for testing whether our stack or queue is empty or not—this is clearly useful, since if it's empty we won't be able to remove an item from it. Finally, I've also chosen to allow **peek** operations that allow us to inspect the item at the top of the stack, or front of the queue, but without removing it. (This is the same item that *would* be removed if we did a **pop** or **dequeue**.)

Implementing stacks

How might we implement the stack interface? As already mentioned, any implementation of lists will in principle yield an implementation of stacks (in more than one way)—but if we're looking for an *efficient* implementation of stacks, two obvious representations suggest themselves:

- Using arrays. Since for arrays it's easier to add new items at the end of the list than the start (recall **append** is cheaper than **cons**), we declare that the 'top' of the stack, where all the pushing and popping happens, is at the right-hand end of the list as it is stored in the array. It's easy to see that all the stack operations can then be performed within constant time, except of **push** which may occasionally trigger an expansion if our array is already full. As we've already seen, even the cost of this expansion can still be viewed as $\Theta(1)$ if amortized over the following run of easy **push**es.
- Using linked lists. Here we treat the start of the linked list as the top of the stack, because this is where it's easiest to add and remove items. It's then not hard to see that all the stack operations of interest can be performed within constant time.

Summarizing, our worst-case runtimes for these implementations are:

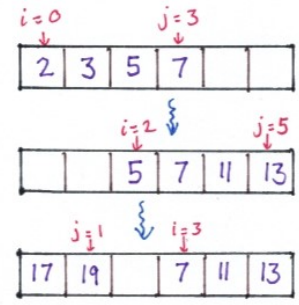
Operation	Extensible array impl	Linked list impl
empty	$O(1)$	$O(1)$
push	* $O(n)$ (amortized $O(1)$)	$O(1)$
peek	$O(1)$	$O(1)$
pop	$O(1)$	$O(1)$

So linked lists are quite competitive as a way of implementing stacks.

Implementing queues

For queues, there's a little more to say. Again, there are two natural approaches:

- Using arrays. The idea is illustrated by the following picture:



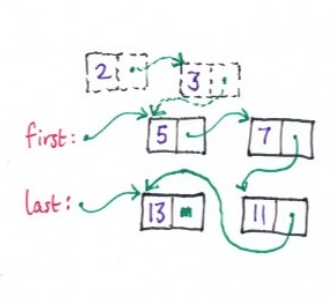
  At any given time, our queue occupies a certain portion of an array. The variables i, j record the position in the array of the front and back of the queue respectively. In the above picture, we first see the effect of first dequeuing the two items 2 and 3 and enqueuing the new items 11 and 13: the portion of the array that's in use gradually works its way along the array like a caterpillar. In the second step, we see the effect of dequeuing 5 and enqueuing 17 and 19: this shows that we're treating our array as a *wraparound* list in which the cell 'after' the last one is the first one. (Think of the caterpillar as disappearing off the right end of the array and reappearing on the left, as might happen in a computer game.)

  Of course, this still leaves the question of what to do when the array is full up. As with lists themselves, if we're working with fixed-size arrays, there's nothing we can do, but if we're working with extensible arrays, we can imagine doing an expand-and-copy that will let us continue—again, this will take $\Theta(n)$ time, but amortizing to $\Theta(1)$ over a long run of enqueues.

  The following pseudocode shows how **enqueue** might be implemented using wraparound arrays (you might like to think for yourself about what implementations of the other queue operations would look like). Notice that the (j+1) mod |A| is what is achieving the wraparound effect.

```
enqueue(x):
    j = (j+1) mod |A|
    if j = i
        fail (or expand)
    else A[j] = x
```

- Using linked lists. Here the front of the queue will be the *head* of the linked list: items will be dequeued from the start of the linked list, and new items will be enqueued at the end of it.



Because we want to do work at both ends of the list, we maintain pointers to the first and last cells. To remove an item from the front of the list, we simply use the pointers to locate the *second* element of the list, then update our 'first' variable to make it point to that. (In the above picture, the ghosted-out cells containing keys 2 and 3 are cells that used to be in the queue before they were dequeued.)

To enqueue a new item, we simply create a new cell for it, add it to the queue by making the current last cell point to it, then update our 'last' variable to make it point to the new cell:

enqueue(x):
    last.next = new Cell(x,null)
    last = last.next

What about the worst-case runtime of all of these? The situation is very similar to the situation for stacks: for both the above implementations, all operations can be performed by a small fixed number of instructions, except for the fact that the array version of **enqueue** will sometimes trigger an expansion taking linear time (though with low amortized cost).

Operation	Wraparound array impl	Linked list impl
enqueue	* $O(n)$ (amortized $O(1)$)	$O(1)$
peek	$O(1)$	$O(1)$
dequeue	$O(1)$	$O(1)$

We thus have two nice implementations of queues, and you can take your pick between them.