Lecture 7 - Dynamic Arrays

Review

A **data structure** is a way to store data with a collection of supported operations that allow us to manipulate that data.

The collection of supported operations is called an **interface** (Abstract Data Type).

The difference between a data structure and its interface can be thought of as follows:

- An **interface** is a specification (what the data structure does)
- A **data structure** is a concrete implementation (how it's done!).

Sequence Interface

- Maintains a sequence of n items, e.g. `34,25,35,54` or `"sam", "joe", 1`
- Supports sequence operations

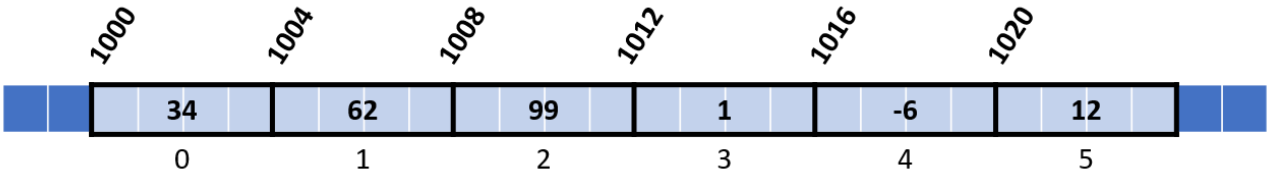
Sequence Operations

Note that this is not written as Python, we are just describing the type of things (operations) we should be able to do to the sequence

Name	Description
`create(X)`	create a sequence from items in `X`
`size()`	return the length of the sequence
`get(i)`	return the item at index `i`
`set(i,x)`	replace the item at index `i` with `x`
`insert(i,x)`	add `x` to position `i` (this will move all previous items at index `i`,`i+1`,... etc up 1)
`delete(i)`	delete the item at index `i` (this will move all previous items at index `i`,`i+1`,... etc down 1)
`insert_first(x)`	add `x` as the first item. Same as `insert(0,x)`
`delete_first()`	delete the first item. Same as `delete(0)`
`insert_last(x)`	add `x` as the last item. Same as `insert(size(),x)`
`delete_last()`	delete the last item. Same as `delete(size()-1)`

The last 4 entries of the table may seem like overkill, but they will allow us to refer to these specific operations when the time comes. It turns out that for different data structures combinations of these are easy.

Array Sequence



- Arrays are fixed-length sequences that store only one type of fixed size.
- The time complexity of creating an array is $O(n)$.
- The space complexity of an array if $O(n)$.
- The worse-case time complexity of the operations on an array is as follows:

Data Structure	`create(X)`	`get(i)`	`set(i,x)`	`insert(i,x)`	`delete(i)`
Array	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$

Worse-case Complexity

Limitations of Arrays

The above table has already outlined the limitations of arrays when it comes to dynamic operations such as `insert()` and `delete()` these are costly.

Can we do better?

Dynamic Arrays

The key idea of dynamic arrays is a very simple one.

- Fix the array size U and track the number of items n
- Resize the array when necessary.

We then have rules for:

- `insert()`
- `delete()`

Insert Rule

When inserting an item we increase the number of items to $n+1$, we have two cases:

1. Array has space ($n+1 \leq U$)
    - If there is space in the array perform the `insert()`
2. Array is full ($n+1 > U$)
    - If the array is full then resize (grow) the array $U = U'$, and then perform `insert()`

Delete Rule

When deleting an item we decrease the number of items to $n-1$, we have two cases:

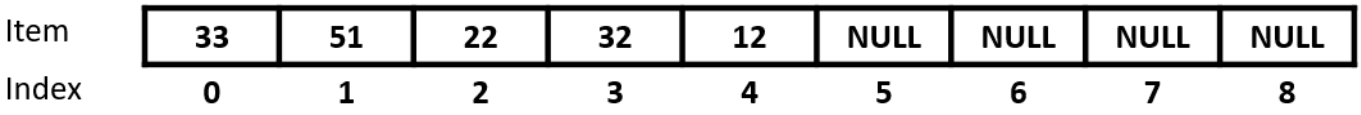
1. Number of items is greater than ($n - 1 > L$)
    - If greater than the threshold L then perform `delete()`
2. Number of items is less than or equal to given threshold L, (i.e. $n - 1 \leq L$)
    - If less than or equal to the threshold L resize (shrink) the array and then perform `delete()`

To explain these we will look at each case using a series of images.

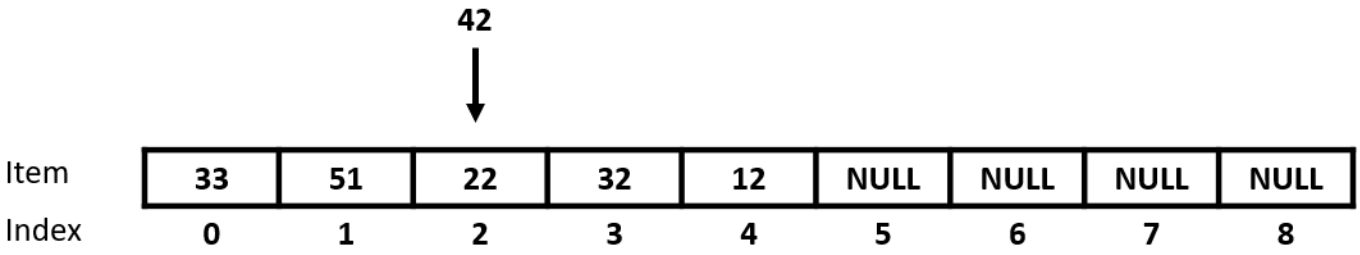
These are also available as an animation which you may find easier to follow.

Insert: Case 1 ($n + 1 \leq U$)

Let's suppose we have an array of fixed size 10, i.e. $U = 10$ which contains 5 items, i.e. $n = 5$. The other items have a `NULL` value contained in them representing that they are not used.

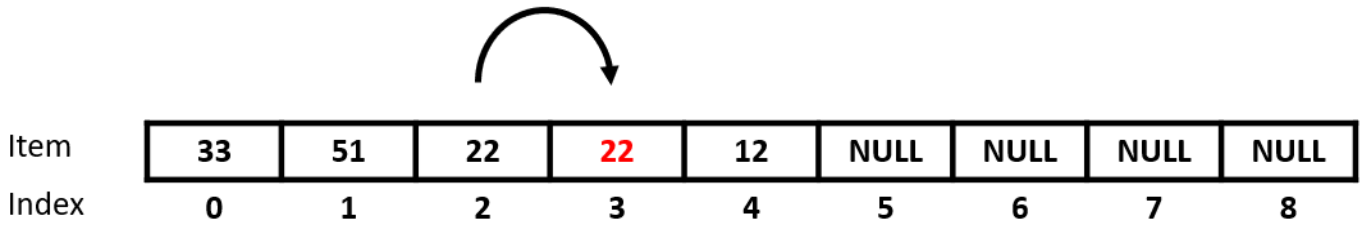


We wish to insert an item at index 2.



Wrong way

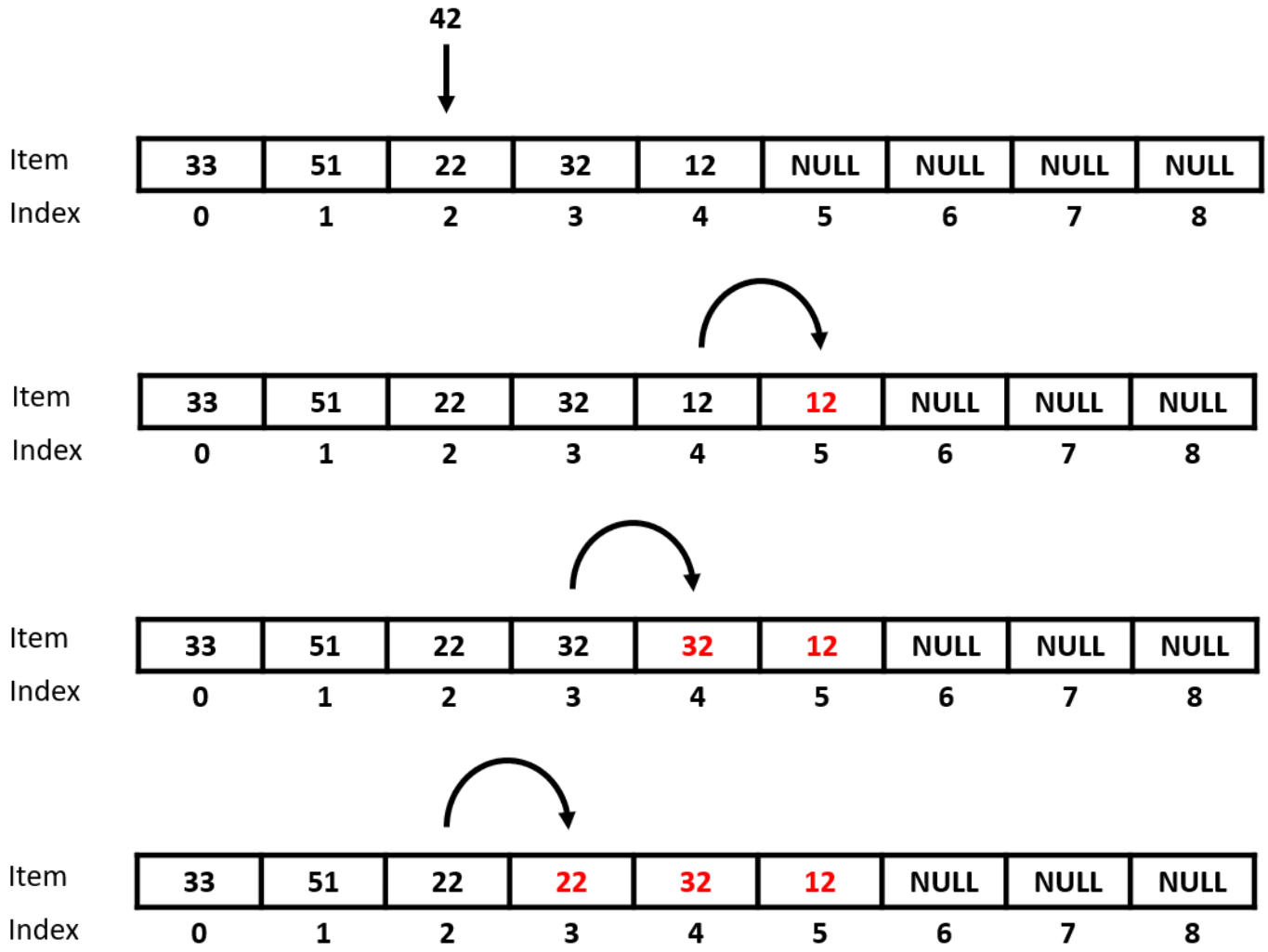
If we begin to copy items to the right from index i, we will overwrite $i + 1$.



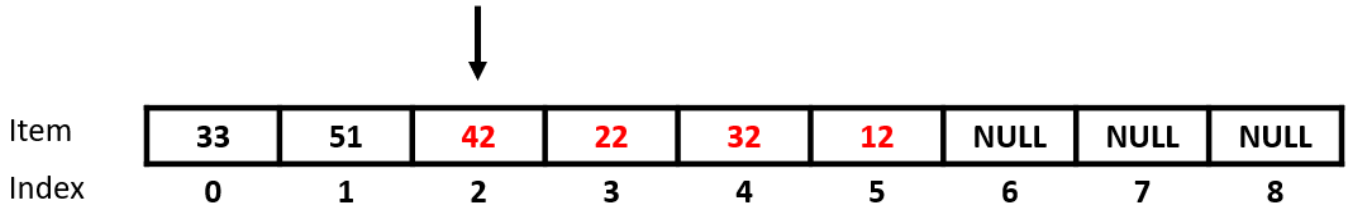
As you can see we have overwritten 32.

Right way

Instead, we copy items to the right starting from index $n - 1$ and ending in index i. We will call this a **backward right copy**.



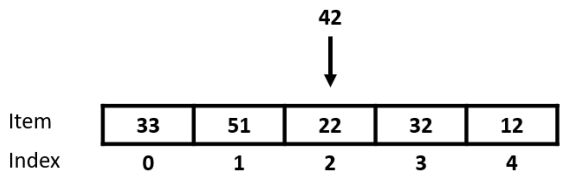
Finally, we can perform the insert into index i.



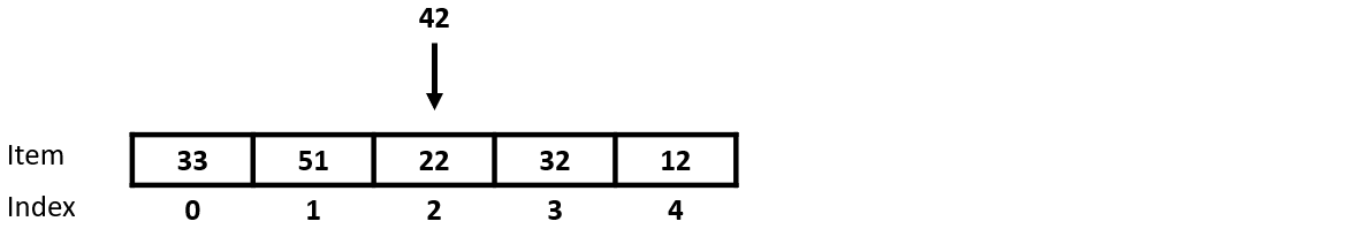
Insert: Case 2 ($n + 1 > U$)

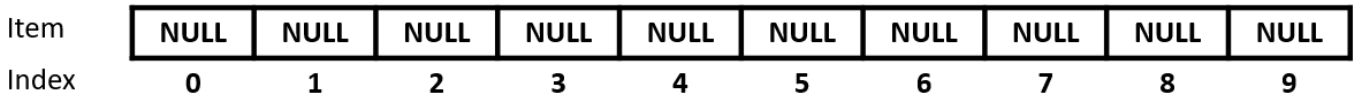
Let's suppose we have an array of fixed size 10, i.e. $U = 10$ which contains 5 items, i.e. $n = 5$. Now we have no space in the array.

If we try and insert, then $n + 1 > U$ and therefore we need to resize and then perform the insert.

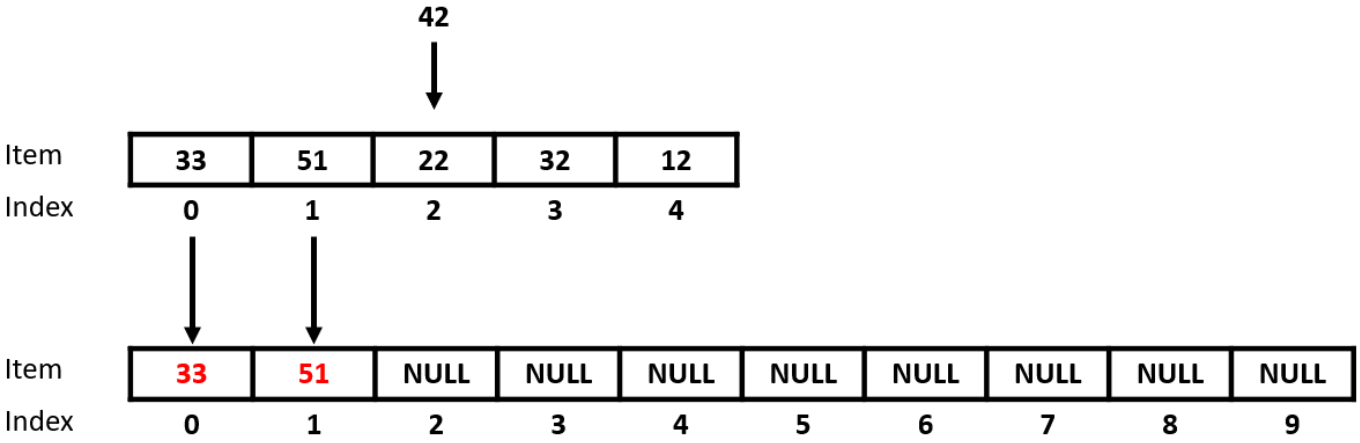


Create a resize copy (not that this can be done by initialising a new array)

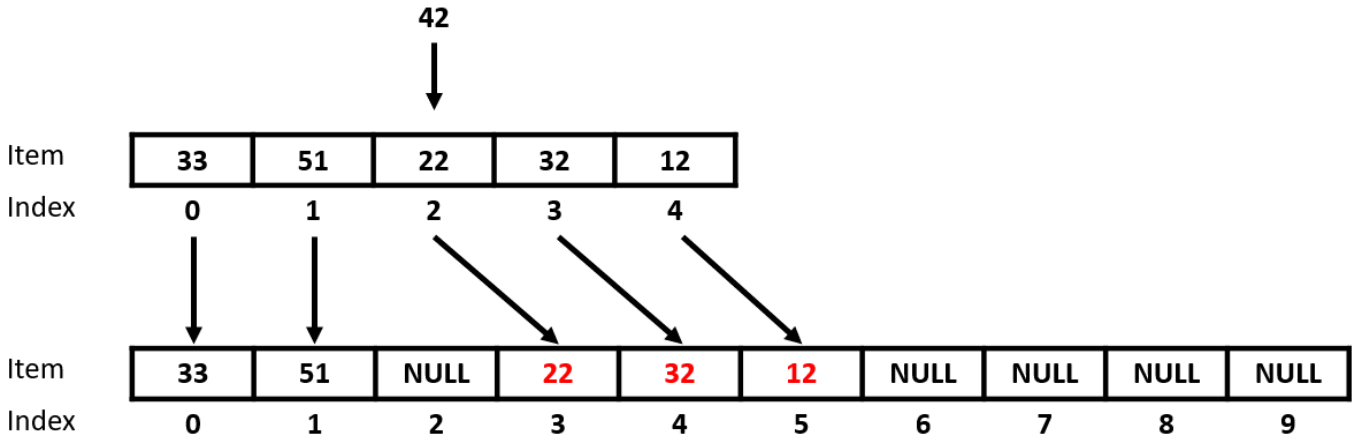




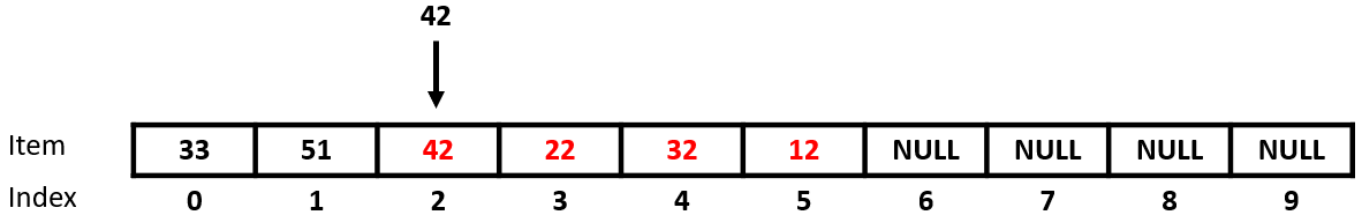
Copy items across before insert.



Copy items after insert.



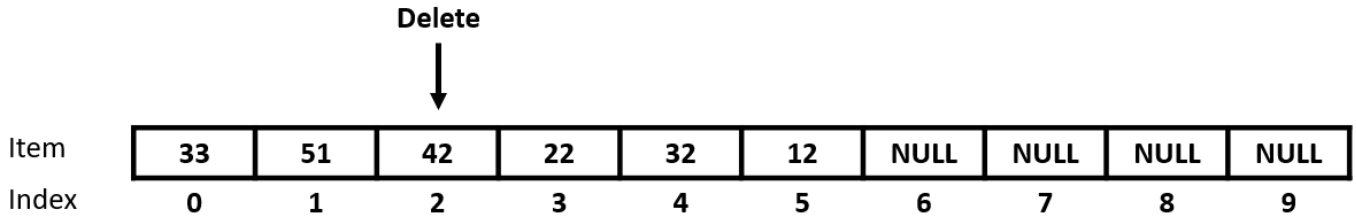
Perform the insert at index 2.



Delete: Case 1 ($n - 1 > L$)

Let's suppose we have an array of fixed size 10, i.e. $U = 10$ which contains 6 items, i.e. $n = 6$. We also have a lower threshold of $L = 4$.

If we try to delete an item at say index $i = 1$

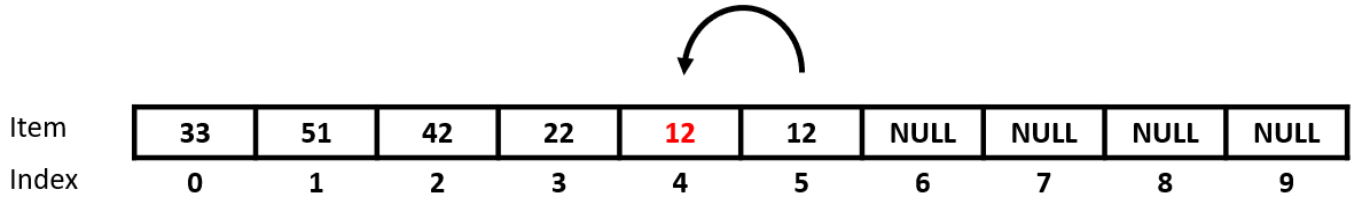


then $n - 1 = 6 > 5$, i.e. we are above the threshold and do **not** resize.

We then copy (shift) the items to the left.

Wrong Way

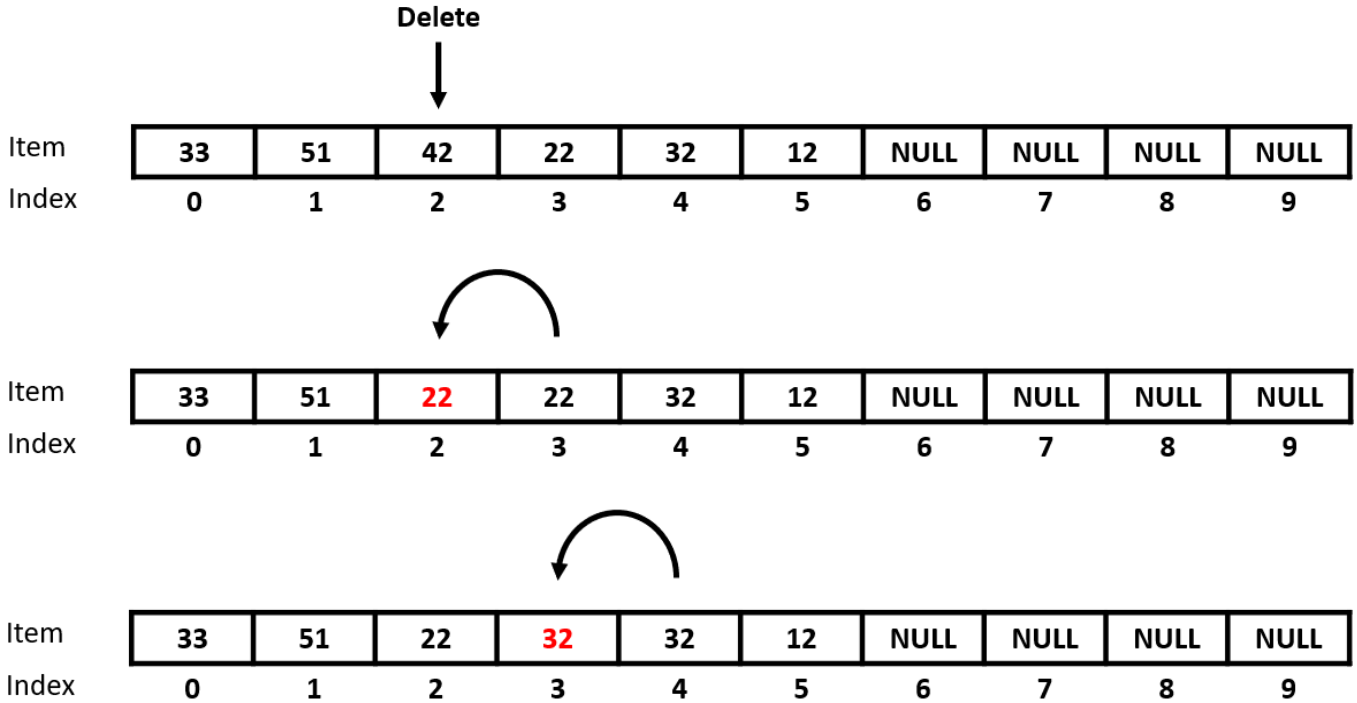
If we try and do this from the back (last item n)

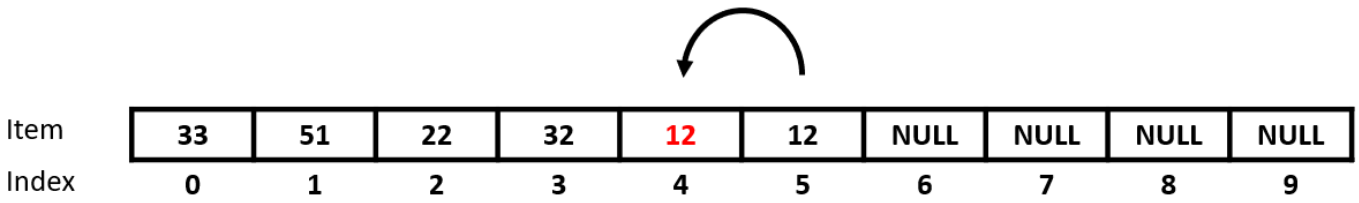


We overwrite 32.

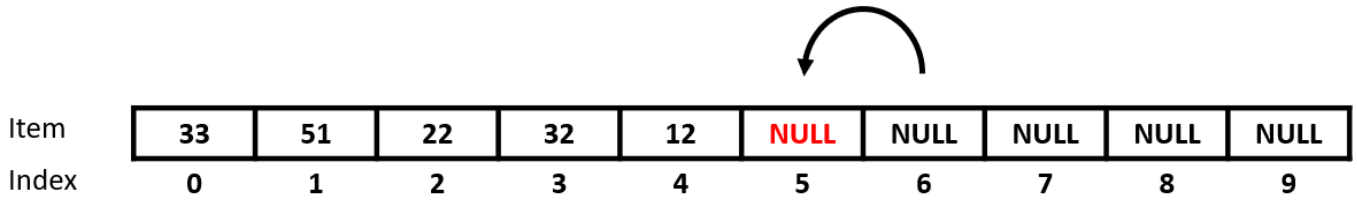
Right way

Therefore we copy items to the left from the front $(i + 1)$

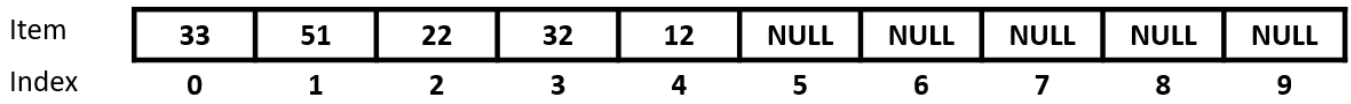




Until $n+1$



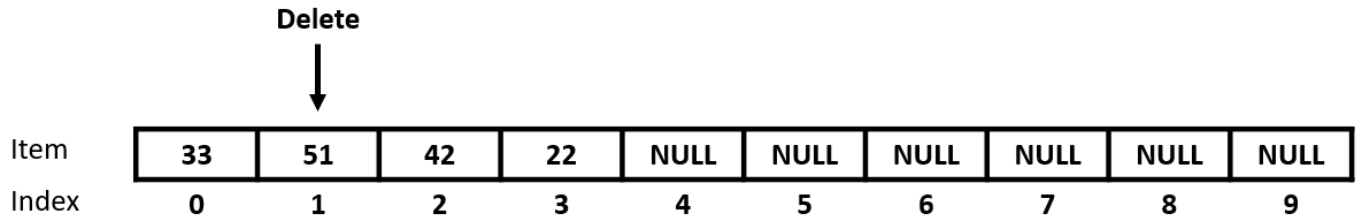
Thus resulting in the deletion of 42 from index 2.



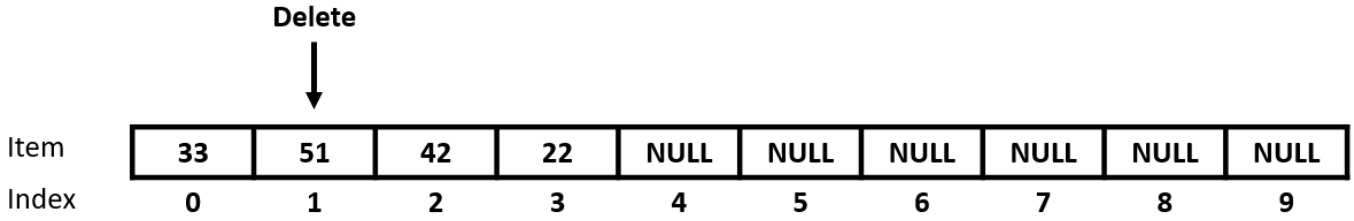
Delete: Case 2 ($n-1 \leq L$)

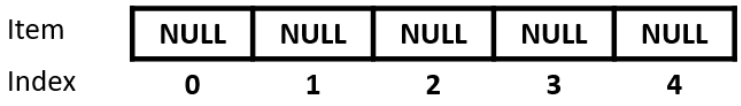
Let's suppose we have an array of fixed size 10, i.e. $U=10$ which contains 4 items, i.e. $n=4$. We also have a lower threshold of $L=3$.

If we try to delete an item at say index $i=1$

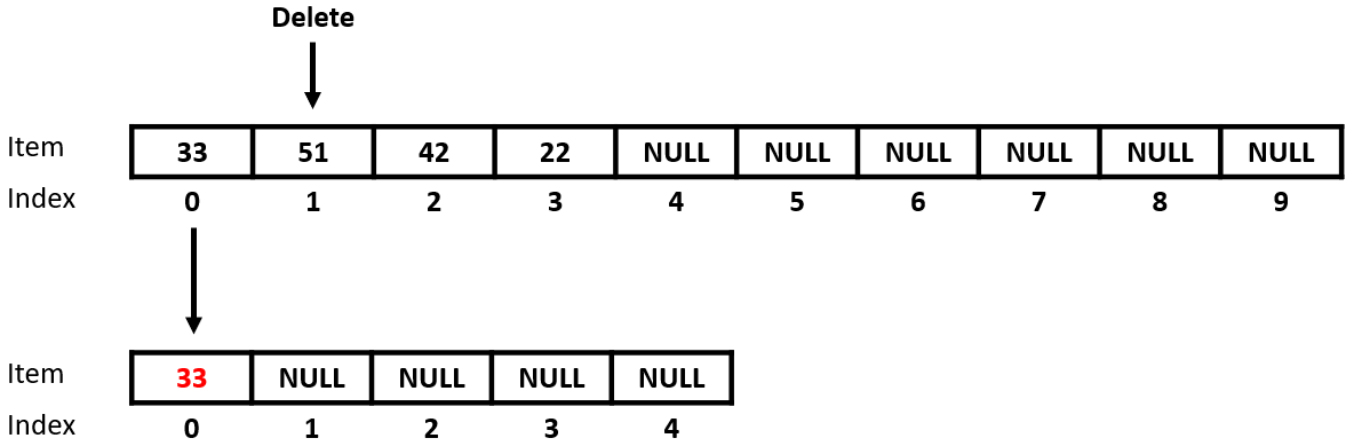


then $n-1=3<3$, i.e. We are below the threshold and therefore need to shrink the array.

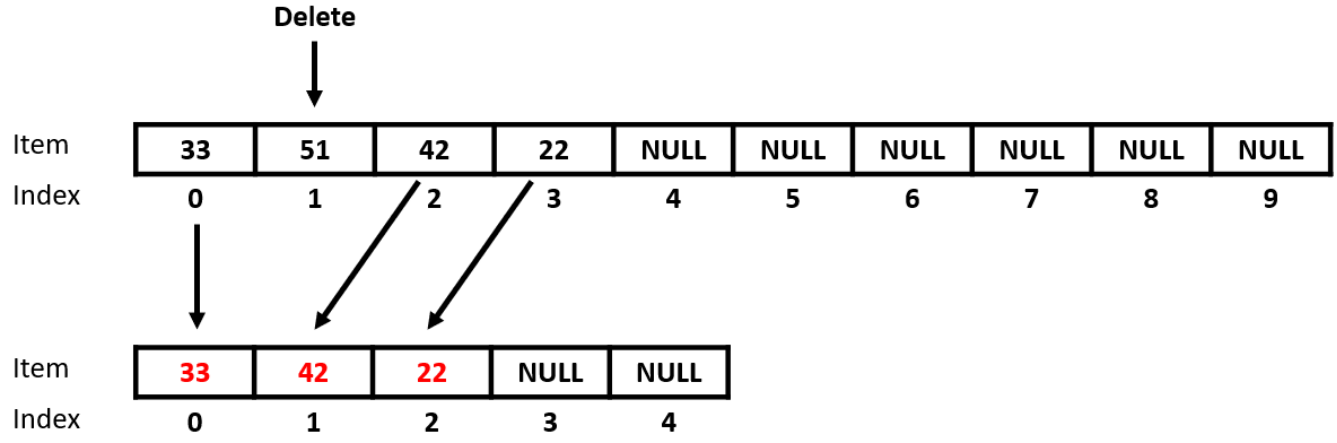




Copy everything before i to new array.



Copy everything after i to new array



You can find a Python implementation of this on the course in the Lecture 7 folder.

It is also a homework exercise to implement your own dynamic array for those of you interested.

Key Questions

- What size should we start with?
- How much do we grow the array by?
- When do we resize for deletion. i.e. lower threshold L
- How much should we shrink the array by?

The answers to these questions require more sophisticated analysis than we will cover in this module.

However, it is common to use **table doubling** which means doubling the size of our array each time we need to resize. We will also shrink the array by a factor of 2. i.e. halve it each time we resize.

Will we also use a simple rule that the lower threshold $L = \lfloor \frac{U}{3} \rfloor$. Note $\lfloor \cdot \rfloor$ is the floor function which rounds down to the nearest integer. e.g. $\lfloor 5.2 \rfloor = 5$.

So if our array is of size $U = 8$, then $L = \lfloor \frac{8}{3} \rfloor = 2$

Growth

It is worth noting that we could have chosen another approach and added a fixed amount each time to the array. This is actually a bad idea and if you are interested you can think about why after you've read the amortized analysis of `insert_last()` and `delete_last()`.

Analysis

Let's now analyse the operations. Note that a key assumption is that allocating a new array of size U is basically free (in a language like C, the array), it is the copying the items into memory that is costly.

Insert

In both cases we have to do approximately n operations and the time-complexity is linear time $O(n)$.

Case 1

The worse-case for an insert is to insert at the front of the list.

As we don't resize then we shift everything over by 1. This is n shifts so as n gets bigger it does so proportionally to n. Therefore the time-complexity is $O(n)$.

Case 2

For the second case we have copy n items to a new array which is $O(n)$ (note we assume we can get an empty array in $O(1)$ time). The insert is just 1 operations so constant time $O(1)$.

In total this is just $O(n)$.

Delete

We can use the same reasoning used for `insert()`, that is:

- deleting the first item requires shifting all the items to the left. i.e $O(n)$.
- a resize such as halving the array will cost $O(n)$.

Thus we have $O(n)$.

Amortized Analysis of `insert_last()` and `delete_last()`

So what is all the fuss about. We haven't gained anything right?

Well not quite, if we specifically analyse `insert_last()` and `delete_last()` in terms of the amortized case we will discover something quite amazing!

Wait what! You said something about the amortized case?

OK, so the amortized case is similar to the average case, but it is different. The average case requires an input distribution and in some cases consideration of random choices.

Amortized analysis is used to analyse a single opertation, e.g. `insert_last()` over a sequence of operations. By this we mean we keep inserting items to the end of the array and consider the average cost of the running time.

For example, imagine an operation took the following number of operations over 10 times.

Whilst `insert_last()` and `delete_last()` still have a worse-case time complexity of $O(n)$, we will find this is not the case for the amortized time complexity.

There are a number of ways we could do this, but we shall use the common way which is using the accounting method.

Amortized Analysis of `insert_last()`

We will assumed here that we are doubling the array everytime we need to resize. However this analysis holds for any constant we choose, e.g. tripling the array.

Before we do this let's consider the number of operations performed when doubling our array.

Key Assumption - We will assume that the cost of doubling our array is the number of copies from the old array to the new array. We will ignore the initialisation of the array. e.g. Doubling the array from 4 to 8 will cost 4 operations

We could debate the above, but one of the issues is depending on machine architecture, programming language, compiler and implementation this will differ. What we do know is that we will have to copy the items in the old array to the new array irrespective of the above.

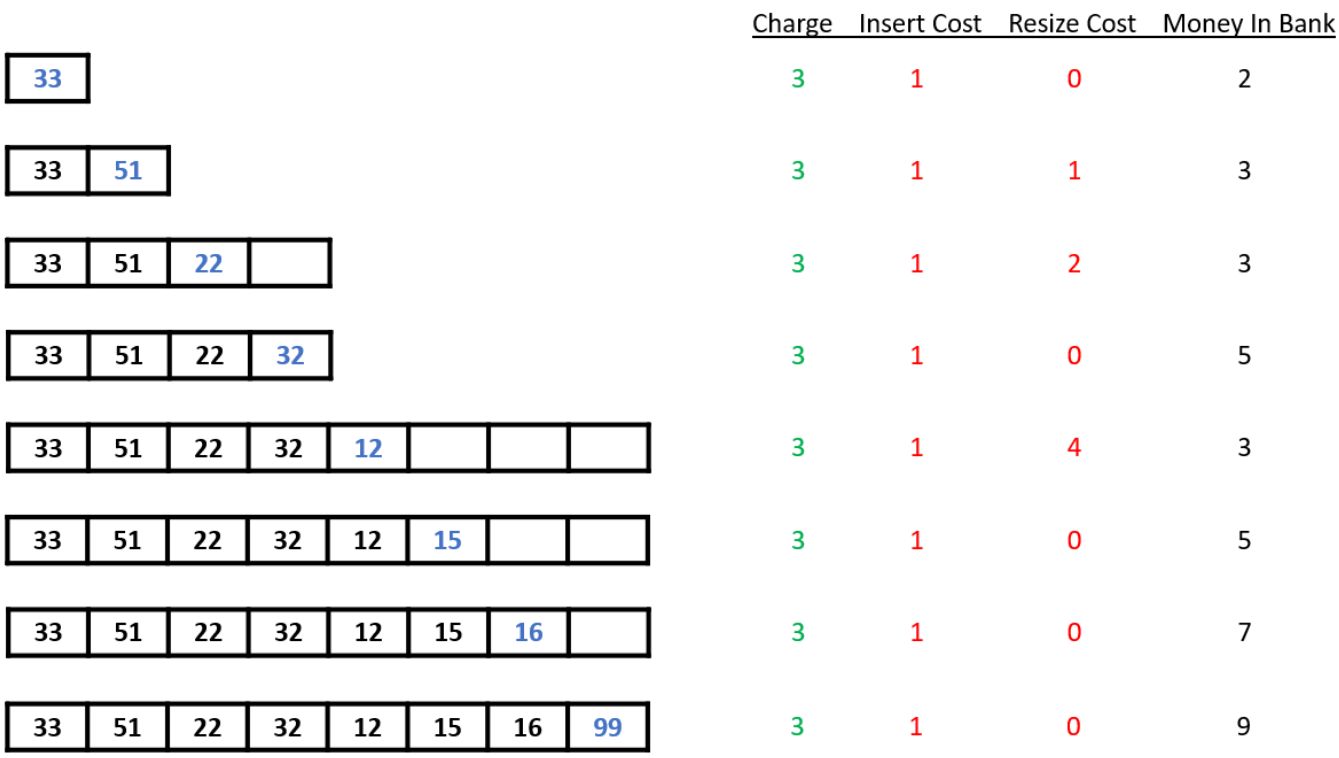
When we insert something as the last element we either:

- insert at the end with **no resize**.
    - Cost: 1 operation for insert
- insert at end with **resize**.
    - Cost: cost of doubling + 1 operation for insert.

We will start with an array of size 1 and keep performing an `insert_last()`.

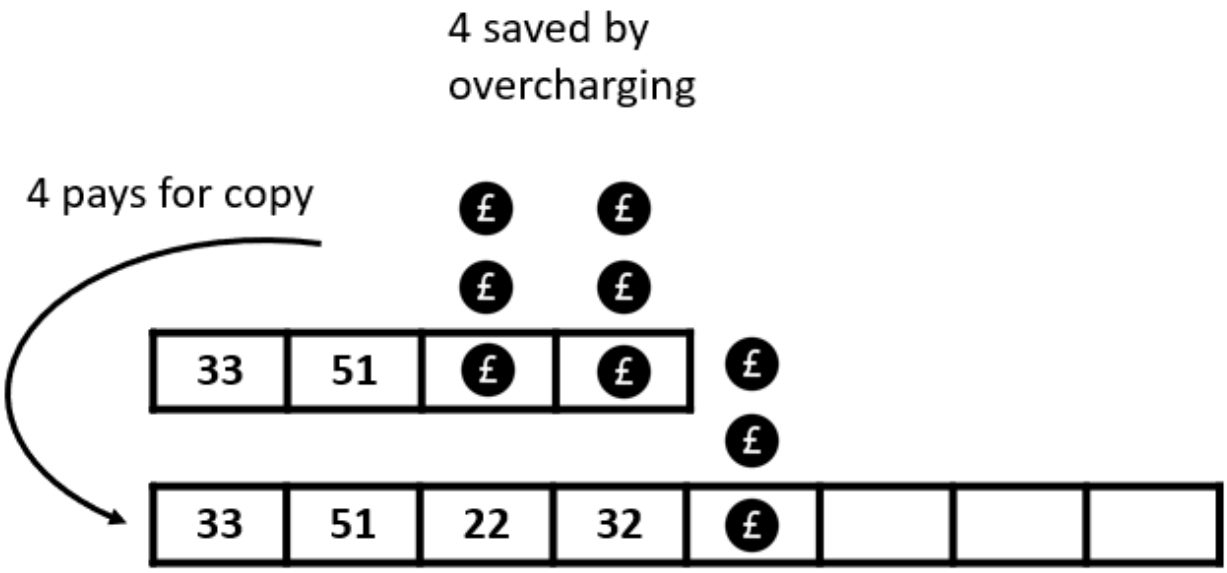
Array Size U	**New Array Size U'**	**No Items n**	**Space Remaining $U' - n$**	**Insert Cost**	**Doubling Cost**	**Total Cost**	**Charge**	**In Bank**
1		1	0	1	0	1	3	**2**
1	2	2	0	1	1	**2**	3	**3**
2	4	3	1	1	2	**3**	3	3
4		4	0	1	0	1	3	**5**
4	8	5	3	1	4	**5**	3	3
8		6	2	1	0	1	3	5
8		7	1	1	0	1	3	7
8		8	0	1	0	1	3	**9**

Array Size U	New Array Size U'	No Items n	Space Remaining $U' - n$	Insert Cost	Doubling Cost	Total Cost	Charge	In Bank
8	16	9	7	1	8	**9**	3	3
16		10	6	1	0	1	3	5
16		11	5	1	0	1	3	7
16		12	4	1	0	1	3	9
16		13	3	1	0	1	3	11
16		14	2	1	0	1	3	13
16		15	1	1	0	1	3	15
16		16	0	1	0	1	3	**17**
16	32	17	15	1	16	**17**	3	3

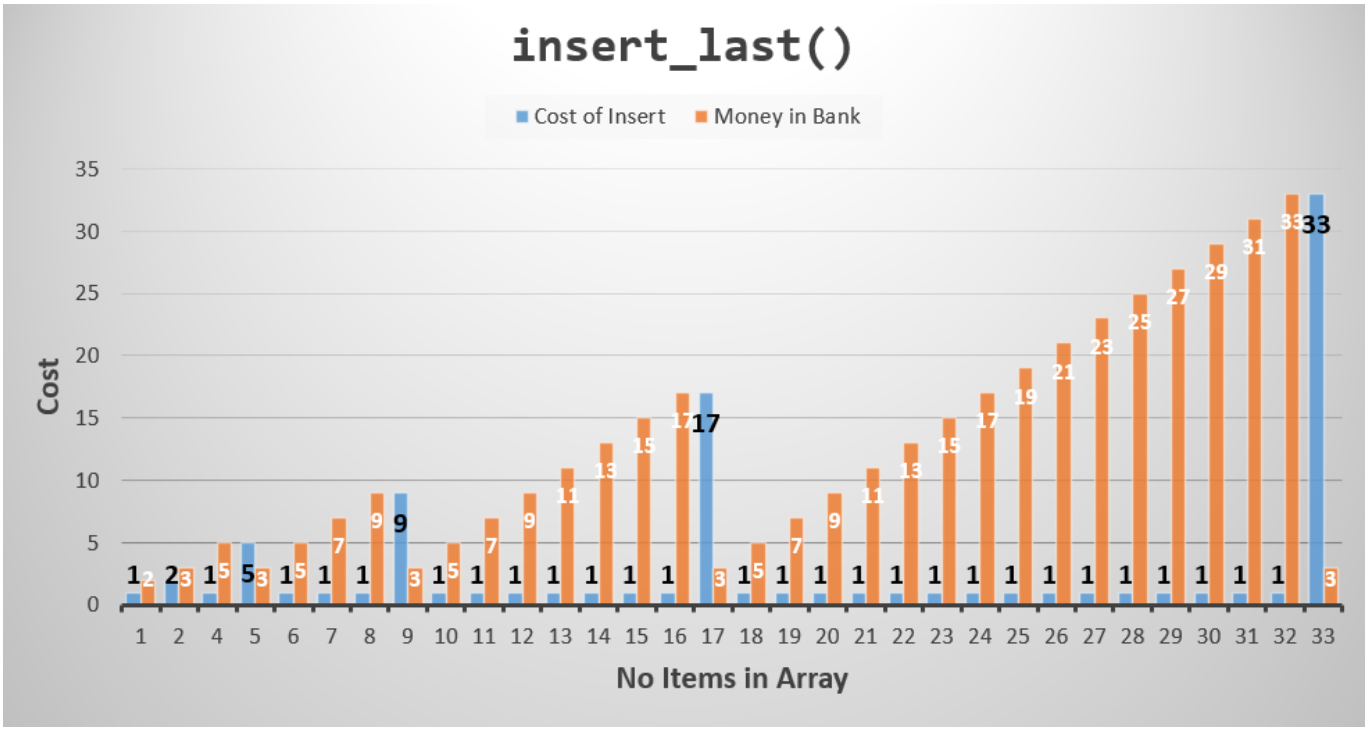


If you look at the Total Cost (No. Operations) you will see that as we start to increase the size of array we get more and more `insert_last()` operations for a cost of 1.

Thus we get an amortized time-complexity of $O(1)$.



A nice way to visualise this is with a plot. The plot shows the cost of each `insert_last()` as the number items in the array increases.



You can see from the plot that for every costly `insert_last()` there are a bunch of cheap inserts before. For example for the inserting item 33 it costs 33. We can pay off 32 using the savings we made by overcharging the previous 16 operations by 2.

Thus we only every need to charge 3 per `insert_last()` no matter how big the array is. **It does not depend on** n.

Therefore the amortized time-complexity is $O(1)$.

Key Insight

- Most of the time it is very cheap to add to the end of the list. In fact $O(1)$.

- Occasionally, we have to double the size of the dynamic array which is costly.
- On average doing many `insert_last()` operations will work out to be $O(1)$.
- That means that it does not depend on the size of your list.

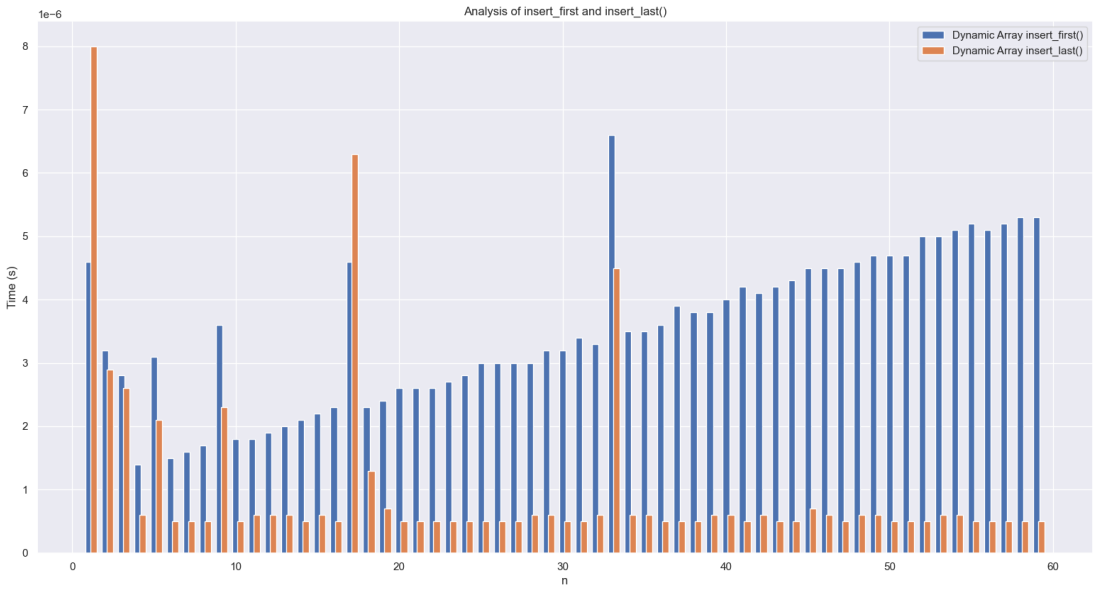
Amortized Analysis of `delete_last()`

This is more complicated, but you can do something similar. The amortized time-complexity of `delete_last()` is also $O(1)$.

Experimental Analysis

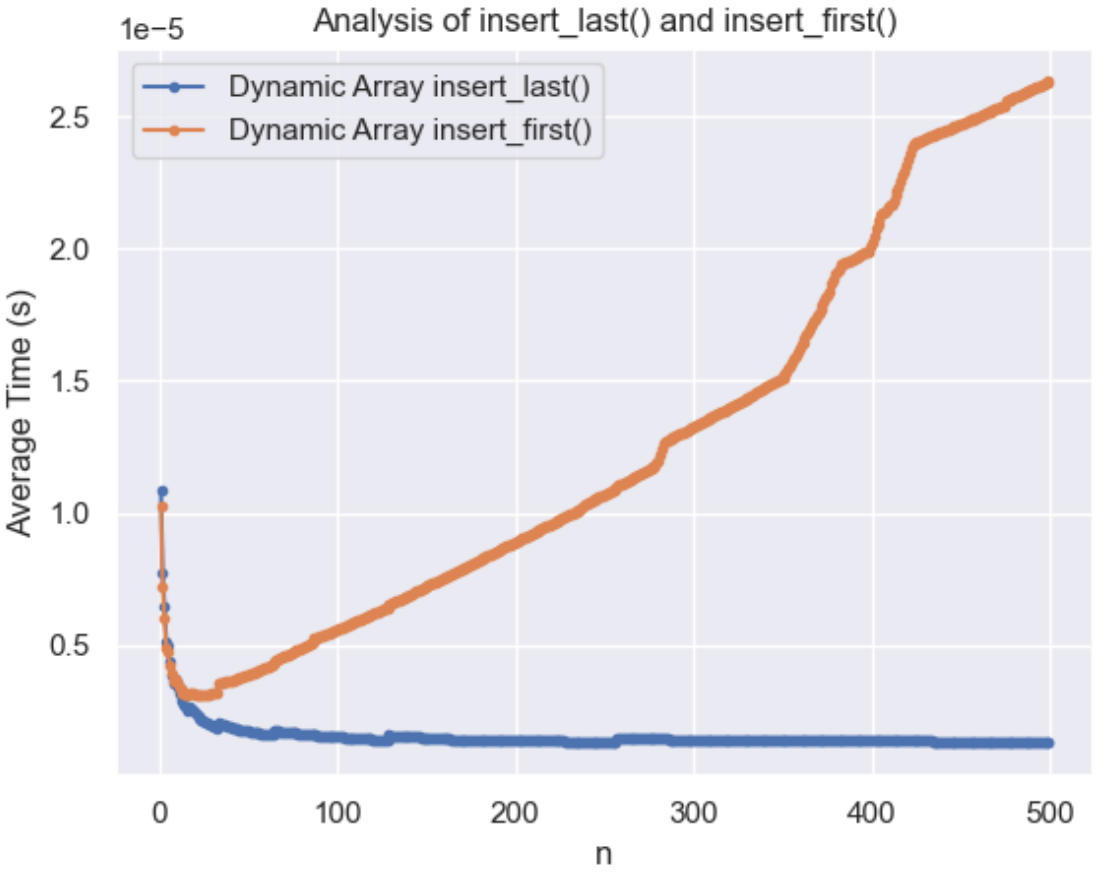
To convince ourselves we will run an experiment with our dynamic array and with our array.

Remember these are timings that may be affected by the CPU doing other things, they are not exact, we are looking at the pattern.



We can see that `insert_last()` seems to at times be very cheap.

If we plot the average.



We can see that `insert_last()` doesn't appear to grow like `insert_first()`.

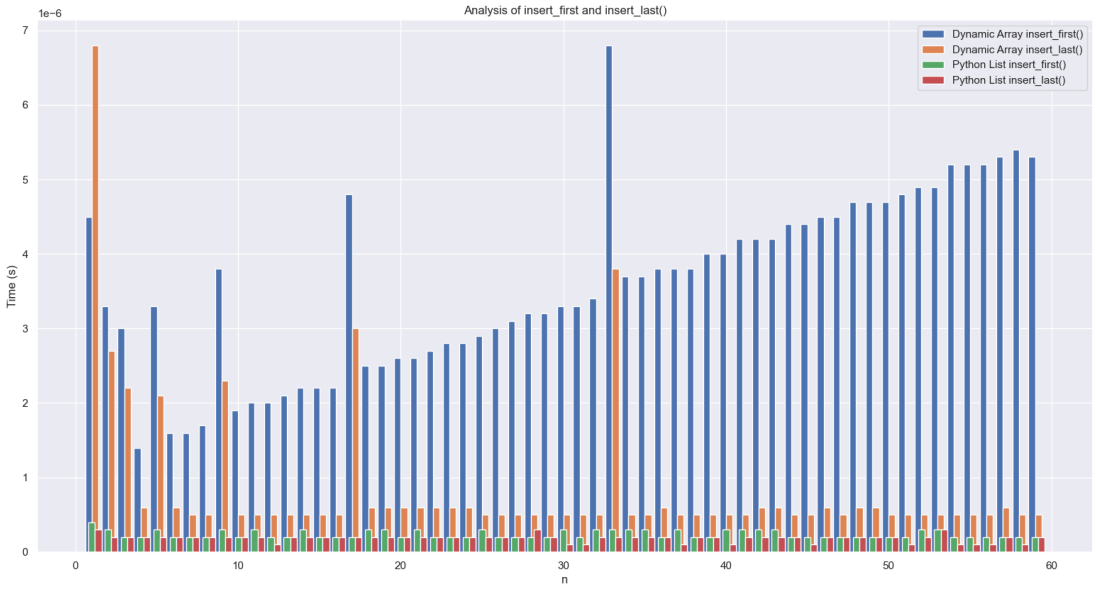
This agrees with our analysis that `insert_last()` does not grow proportionally to n. That is, it's amortized cost (cost over the sequence of `insert_last()` operations) is $O(1)$.

Python Lists

We can now add in Python lists to our experiment.

Python lists are highly opimised dynamic arrays.

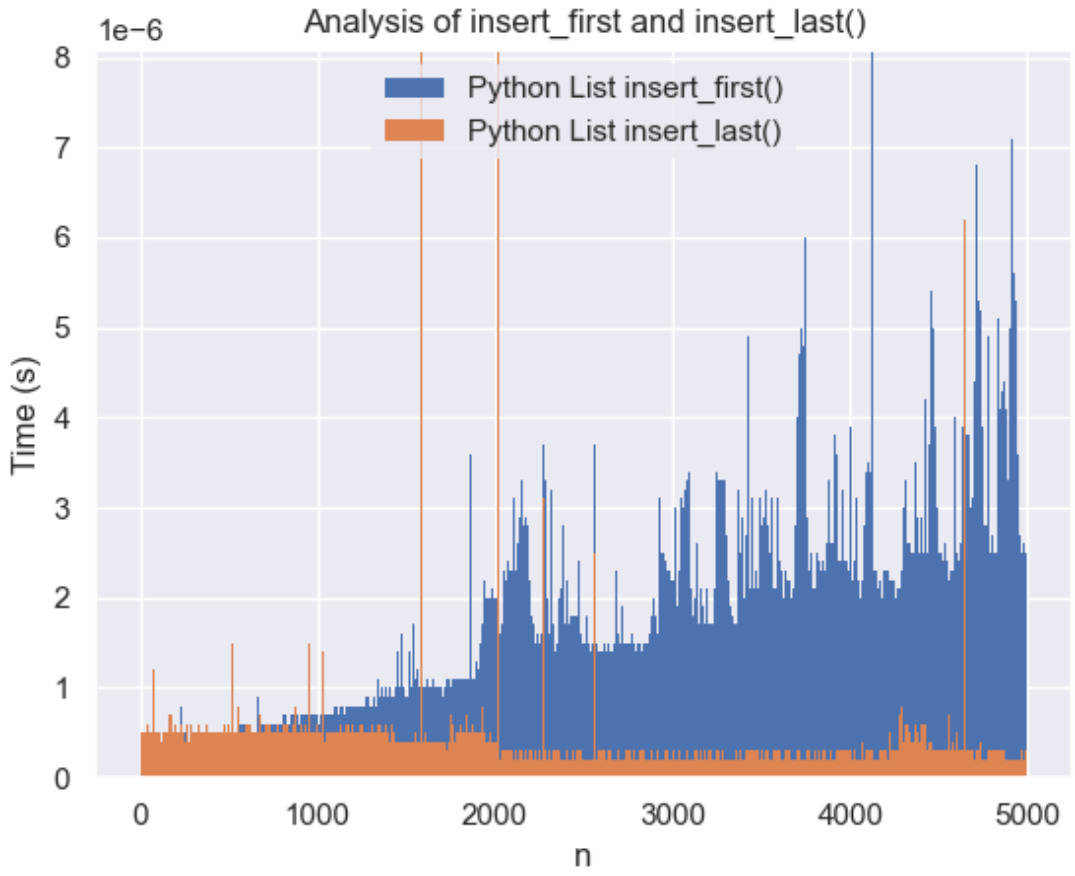
We can analyse their `append()` method (`insert_last()`) method and `pop()` method (`insert_first()`).



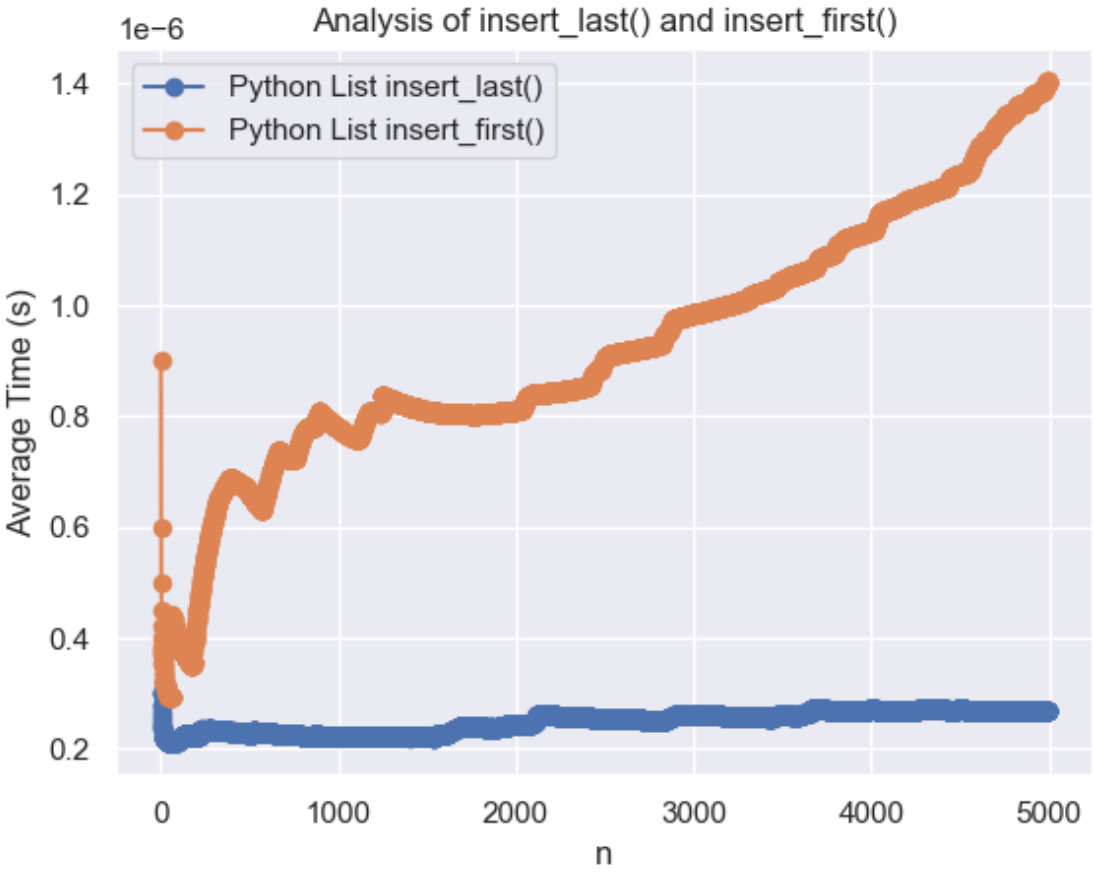
It also looks like `insert_first()` and `insert_last()` are the same for a Python `list`.

Be careful...

If we zoom in and run this for a larger number of operations we will see this isn't the case.



We can also see this by plotting average time taken.



Which agree with our analysis that `append()` (`insert_last()`) have an amortized cost of $O(1)$.

Summary

- Dynamic Arrays resize as necessary
- It is cheap to insert and delete at the end of a Dynamic Array
- Python lists are highly optimised Dynamic Arrays

Data Structure	`create(X)`	`get(i)` `set(i,x)`	`insert(i,x)` `delete(i)`	`insert_first(i,x)` `delete_first()`	`insert_last(i,x)` `delete_last()`	**Space**
Array	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Dynamic Array	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$**	$O(n)$

Worse-case Complexity, ****Amoritized Time**

Key Takeaway

If you are doing a lot of appends use a Dynamic Array.