Theory: Dynamic array

20 minutes 0 / 5 problems solved

Skip this topic Start practicing

3734 users solved this topic. Latest completion was 9 minutes ago.

§1. Introduction

Many programs need to store and process sequences of elements of the same type like numbers, strings, or even more complex objects. An **array** is a widely used structure to represent such data sequences since an element can be accessed in constant time by index. However, regular arrays suffer from a significant drawback – they have a fixed size. This does not allow one to create an array if the number of elements is unknown in advance. In such cases, using a **dynamic array** is a possible solution.

§2. Essentials

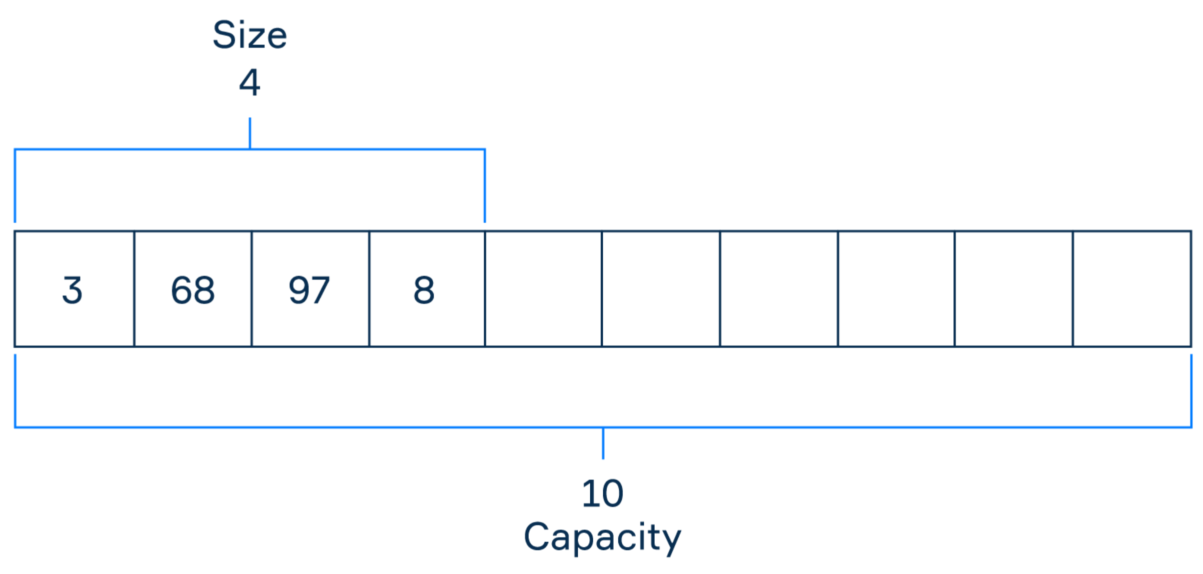
Dynamic array is a linear data structure that is able to increase and, in some implementations, shrink when their size changes. As a rule, it has an internal regular array that actually stores data under the hood and provides some additional operations on top of it.

A dynamic array has two important properties:

- **size** – the number of elements already stored in it;
- **capacity** – a possible number of elements to be stored that corresponds to the size of the internal regular array.

Usually, there are two paths: either to specify a **capacity** for a new dynamic array or to set a constant default value (e.g. 10). In contrast to basic arrays, dynamic arrays have operations for adding/removing elements to or from any position. This way, we can add and remove elements one by one after a dynamic array has been created.

The picture below demonstrates a dynamic array to which we added four numbers. The actual size is 4 and the capacity is 10 (initial):



§3. Scaling factor

If the number of elements exceeds the **capacity**, all elements will be copied to a new internal array of a bigger size. There are a number of different scaling strategies for the size of it. The most common ones are the multiplication of the initial capacity by 1.5 (Java) or 2 (C++, the GCC STL implementation). There are also more unique cases like Golang dynamic array ("slice"), which doubles the size until 1024 elements (after that the ratio is 5/4).

It is a trade-off between time and space complexities. With a bigger growth factor, we have more insertions before we would have to extend an array, thus decreasing time complexity.

But what is the best scaling factor? That is, what value will have both time and space complexities? It turns out, that the value must be equal to the golden ratio, 1.61803. As you may notice, 1.5 is as close to it as it can get. If

Current topic:

Dynamic array

Topic depends on:

✕ Fixed-size array

Topic is required for:

Dynamic array in Java

ArrayList

Table of contents:

you're curious to know more, you can read it up there.

It may also be necessary to support the shrinking of the internal array when removing elements to reduce the required size.

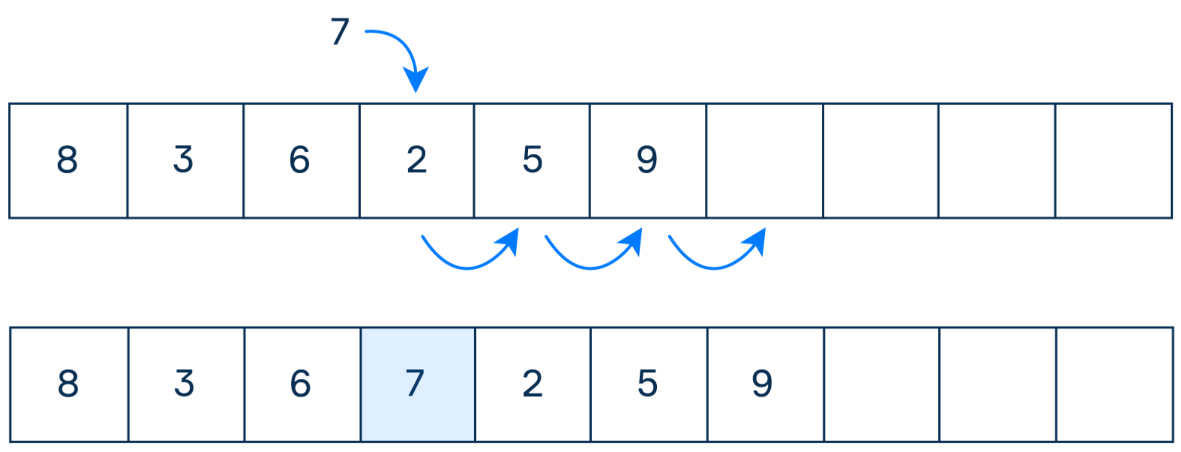
§4. Common operations

Add an element to the end of the array. As discussed above, in the base case scenario where we just add an element to an array without specifying the index, we'll have these complexities:

- $O(1)$ – in average cases, since we just insert an element to already allocated memory (less than capacity);
- $O(n)$ – in the worst case, where we ran out of space and need to allocate a new array and copy every element into it.

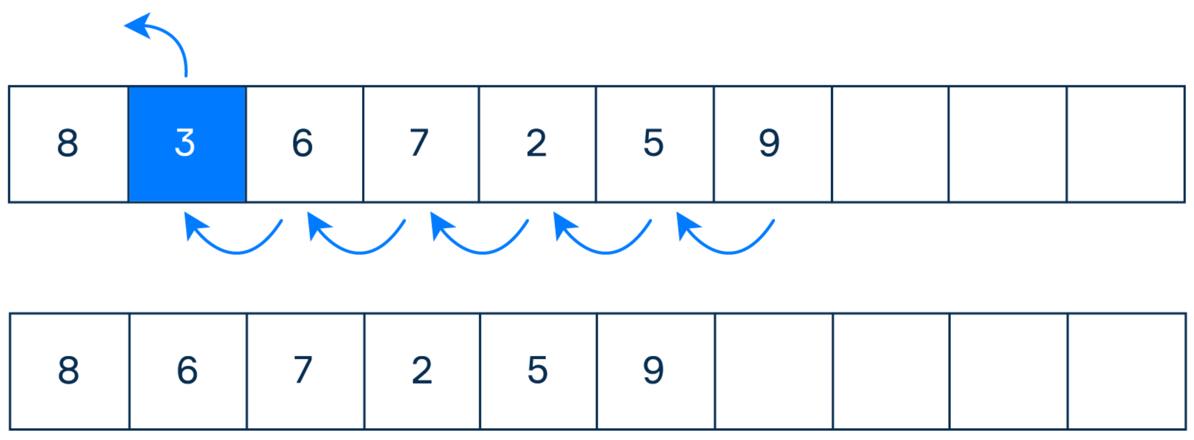
The average estimate for adding an element to the end of the array is called amortized. Since it is rather difficult to tell from the first glance that it is $O(1)$, we have to use a special analysis for that. If anyone is interested they can read about it here.

Add an element at the specified index. This operation is used when we want to add an element between some already placed elements. Its complexities (both average and worst) would be $O(n)$ since on each insertion we must move an element at the index we want and then move every element one index to the right.



Update value at the specified index. This operation replaces the element at the specified index with the element. All this is done in constant time since it is just like the assignment in the basic array, so the complexities are both $O(1)$.

Remove an element by value/index. These methods either remove the first occurrence of an element specified or an element at the index specified. Both are similar to adding an element at the specified index in the sense that we would have to move some (or all) of the remaining elements one index to the left; therefore their complexities would also be $O(n)$.



Clear. Here we just want to remove every element from the array. Since insertion is done via computation on the current array size, we can just reset the size to zero and override the old elements during the following inserts. That would leave the elements hanging out in memory (so the garbage collector won't be able to collect them) until they are overridden. The simplest form would have complexities of $O(1)$, but the right one would have $O(n)$.

Get element by index. Since a dynamic array is basically just a normal array, we can access elements by their index in constant time, so complexities are $O(1)$.

§5. Conclusion

A dynamic array is just like a regular array, but the number of stored elements can be changed. If adding operations run out of space to store elements, a new bigger array is allocated, and every element of the old array is copied to the new one. The scaling factor is a trade-off between time (speed) and space. With a bigger factor we have fewer allocations and less copying, but higher chances of running out of memory. The most common factors are 1.5 and 2. In some implementations, a dynamic array can support shrinking to reduce the used memory after removing elements.