Data Structures

Iñaki Lakunza

March 14, 2024

Contents

1 Introduction

What is a Data Structure? A data structure (DS) is a way of organizing data so that it can be used effectively. It is just a way of organizing data, so that it can then be accessed and updated easily.

Why Data Structures? They are essential ingredients in creating fast and powerful algorithms. They help to manage and organize data in a very natural way. And, they make the code cleaner and easier to understand. Data Structures can make a difference between having an okay product and an outstanding one.

1.1 Abstract Data Type

An **Abstract data type (ADT)** is an abstraction of a data structure which PROVIDES ONLY THE INTERFACE TO WHICH A DATA STRUCTURE MUST ADHERE TO. The interface does not give any specific details about how something should be implemented or in what programming language.

As an example, we suppose that our abstract data type is for a mode of transportation, to get from pint A to point B. There are different modes of transportation, like walking, or going by train. These specific modes of transportation would be analogous to the data structures themselves. We want to get from one place to another, that is our abstract data type. And, how did we do that? That is our data structure.

These are some data type examples:

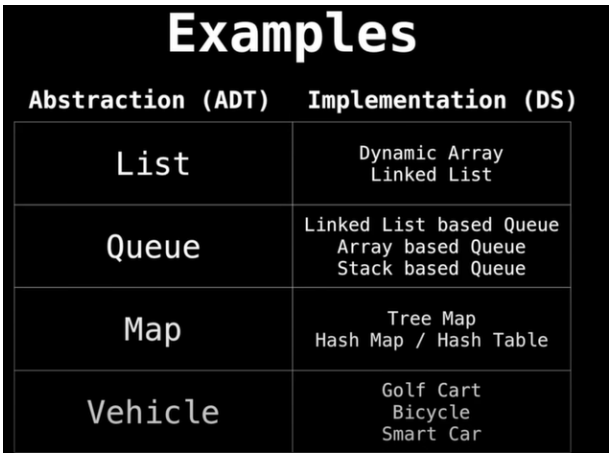

Figure 1:

As we can see, lists can be implemented in two ways, we can have a dynamic array, or a linked list. And the same for the rest, we can implement any abstraction in very different ways. **Abstract data types only defines how a data structure should behave, and what methods it should have, but not the details surrounding how those methods are implemented**.

1.2 Computational Complexity

We must take into account the **time** and the **space** needed by our algorithm.

Big-O Notation gives an upper bound of the complexity in the **worst** case, helping to quantify performance as the input size becomes **arbitrarily large**.

We care when our input becomes large, so because of that reason we will be ignoring constants, for example.

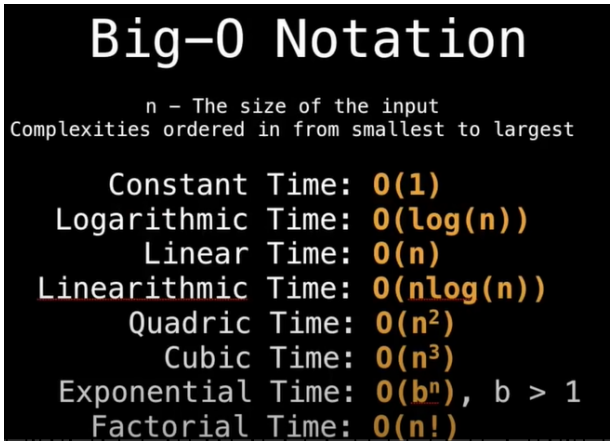

Figure 2:

n will usually be the size of the input coming in to our algorithm.

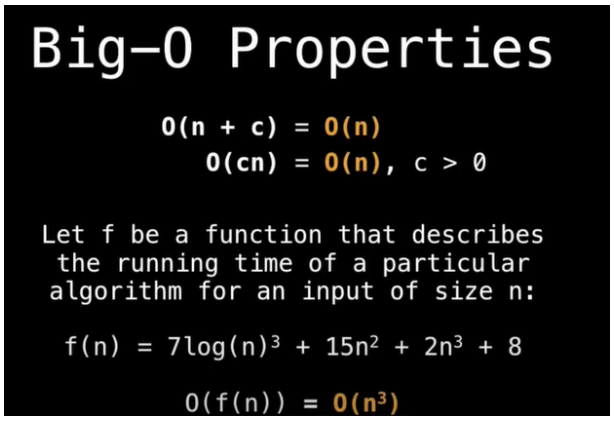

Figure 3:

O really cares when our notation becomes very big, that is why remove constant values, when adding or when multiplying. (But when the constant is very very large in practice we should consider it).

For example the two following blocks run in constant time, because they do not depend on the input size n:

```
a := 1
b := 1
c := a + 5*b
```

```
i := 0
While i<11 Do
    i = i + 1
```

In the second case, we are doing a loop, but the loop does not depend on the input size, so it needs a constant time.

On the other hand, the following run in **linear** time: $O(n)$

```
i := 0
While i<n Do
    i = i+1
# f(n) = n
# O(f(n)) = O(n)
```

```
i := 0
While i<n Do
    i = i+3

# f(n) = n/3
# O(f(n)) = O(n)
```

In the second case, we are incrementing by 3, so we will end up ending the loop 3 times faster, so we will end up doing $n/3$ iterations, but, since we do not care about constants, the time complexity is n.

Next, both of the following examples run in **quadratic** time. The first one may be obvious since n work done n times is $n * n = O(n^2)$.

```
For (i := 0; i<n; i = i+1)
    For (j := 0; j<n; j = j+1)

# f(n) = n*n = n^2 , O(f(n)) = O(n^2)
```

```
For (i := 0; i<n; i = i+1)
    For (j := 0; j<n; j = j+1)
    # We have replaced in the second For the 0 with i
```

The first block is obvious, but, in the second case, focusing just in the second loop, since i goes from $[0, n)$, the amount of looping is directly determined by what i is. Remark that if $i = 0$, we do n work, if $i = 1$, we do $n - 1$ work, if $i = 2$, we do $n - 2$ work, etc... So, we end up having the following: $(n) + (n - 1) + (n - 2) + (n - 3) + ... + 3 + 2 + 1$ This turns out to be $n(n + 1)/2$, which can be considered as n^2, because we do not care about constants. So, $O(n^2)$.

Now, here is a more complex example, where we do a search in a binary tree:

Here is another example:

```
i := 0
While i<n Do
```

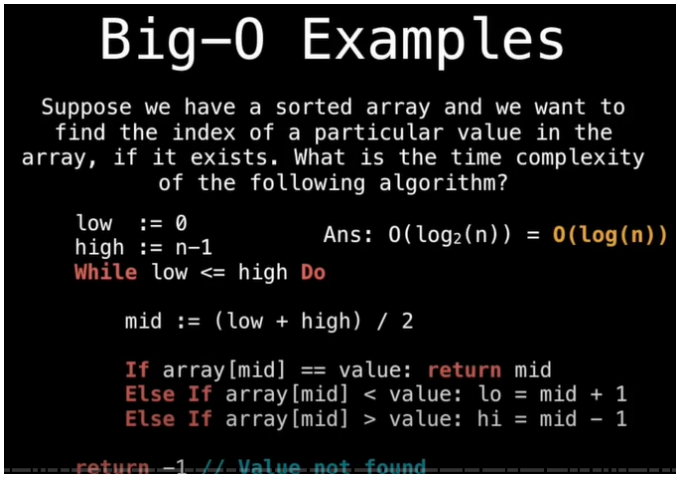

Figure 4:

```
j=0
While j<3*n Do
    j = j+1
j=0
While j<2*n Do
    j = j+1
i = i+1

# f(n) = n*(3n + 2n) = 5n^2
# O(f(n)) = O(n^2)
```

We have two inner loops, so, we add the time of the loops that are the same, and multiply different level loops.

Another example:

```
i := 0
While i<3*n Do
    j := 10
    While j<=50 Do
        j = j+1
    j=0
    While j< n*n*n Do
        j = j+2
    i = i+1

# f(n) = 3n + (40 + n^3 /2) = 3n/40 + 3n^4 /2
# O(f(n)) = O(n^4)
```

We have i going from 0 to $3 * n$ in the outside. So we have to multiply that with what it is going on in the inside. Inside, j goes from 10 to 50, so that does 40 loops exactly every loop. So that is a constant 40 amount of loop. In the second loop, j is less than n^3, but $j = j + 2$ so it is accelerated, so in the inside we are going to get $(40 + n^3/2)$, and we have to multiply that by $3n$.

2 Static and Dynamic Arrays

2.1 Introduction

Arrays are probably the most used Data Structure, probably because it forms the fundamental building block of all data structures. With arrays and pointers, we could be able to construct any data structure.

Static Arrays

A static array is a **fixed length** container containing n elements **indexable** from the range $[0, n-1]$.

By being indexable, we mean that each slot or index in the array can be referenced with a number.

On the other hand, static arrays are given **contiguous chunks of memory**, meaning that our chunk of memory will not have holes and gaps, it will be contiguous, all addresses will be adjacent in our static array.

When and where is a static array used?

- Storing and accessing sequential data
- Temporarily storing objects
- Used by IO routines as buffers
- Lookup tables and inverse lookup tables. This way be can retrieve the data easily from a table of information.
- Can be used to return multiple values from a function. This is useful in programming languages where just a single return value is allowed in functions.
- Used in dynamic programming to cache answers to subproblems.

Complexity The access time for static and dynamic arrays is constant because of a property that arrays are indexable.

On the other hand, searching takes a linear time, since we have to transverse all the elements in the array, and, if the element we are looking for does not exist (worst case), we will have to analyze all elements in the array.

Inserting, appending and deleting in a static array does not make sense. This is because the static array is a fixed size container, it cannot grow larger or smaller.

But inserting in a dynamic array will cost a linear time, because we will potentially have to shift all the elements in the array to the right, and recopy all the elements into the new static array (**this is assuming we are implementing dynamic arrays using static arrays**).

Appending though is constant because when we append elements we just have to resize the internal static array containing all those elements. But this happens so rarely that appending becomes constant time.

Deletions are linear for the same reason that insertions are linear, because in the worst case we will have to shift all the elements in the array and potentially recopy the whole static array.

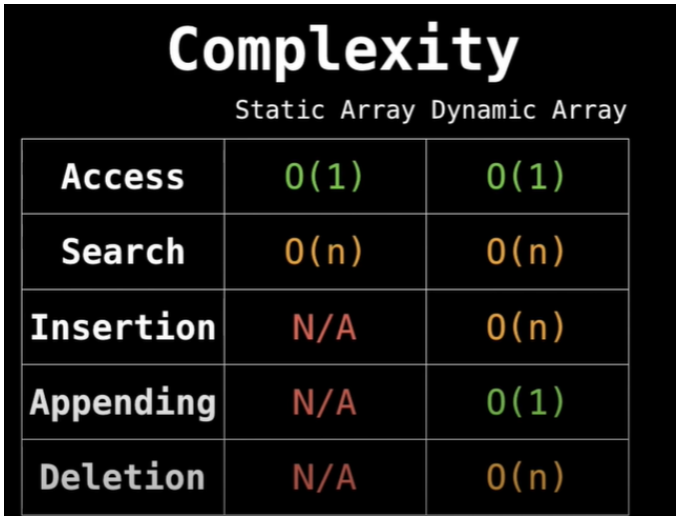

Figure 5:

2.2 Operations in Dynamic Arrays

As we know, dynamic arrays can grow and shrink in size as needed. So the dynamic arrays can do all similar **get** and **set** operations that static arrays can do, but, unlike the static array, it grows inside as dinamically as needed.

How can we implement a dynamic array?
One ways is to **use a static array** (this is not the only way):

1. Create a satic array with an initial capacity.
2. Add elements to the underlying static array, keeping track of the number of elements.
3. If adding another element exceeds the internal capacity of our static array, **create a new static array with twice the capacity and copy the original elements to it**.

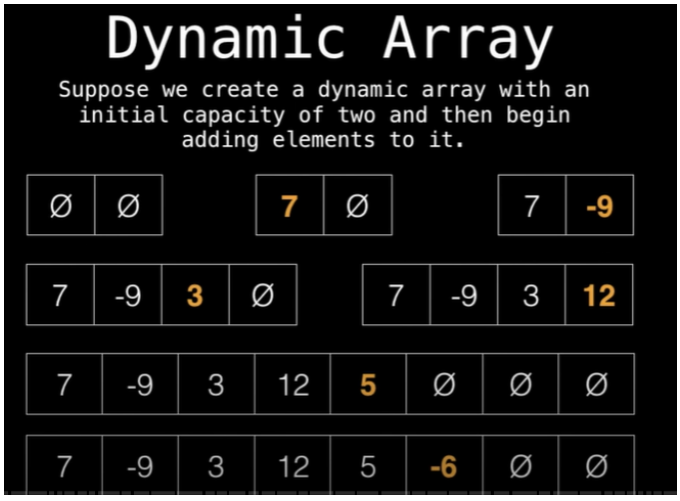

Figure 6:

3 Singly and Doubly Linked Lists

3.1 Introduction

A linked list is a sequential list of nodes that hold data which point to other nodes also containing data.

Is is important to notice that **every node has a pointer to another node**.

And also notice that the last pointer points to null, meaning that there are no more nodes at this point. The last node always has a null reference.

Figure 7:

Where are linked lists used?

- Used in many List, Queue & Stack implementations, because of their great time complexity for adding and removing elements.
- Great for creating circular lists, making the pointer in the last node point to the first node. Used to model repeating event cycles
- Can easily model real world objects such as trains.
- Used in separate chaining, which is present in certain Hash-table implementations to deal with hashing collisions.
- Often used in the implementation of adjacency lists for graphs..

Terminology

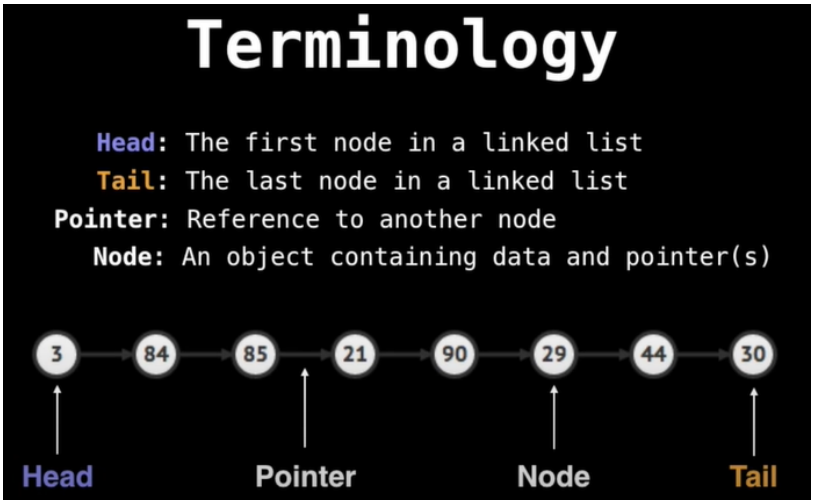

Figure 8:

It is very important to **always maintain a reference to the head of the link lists**. This is because we need somewhere to start when transversing our list.

We also give a name to the last element of the linked list. This is called the **tail of the list**.

Then we have the nodes themselves, which contain pointers (pointers are also sometimes called references), and these pointers always point to the next node.

The nodes themselves are usually represented as structs, or classes, when implemented.

3.2 Singly vs Doubly Linked Lists

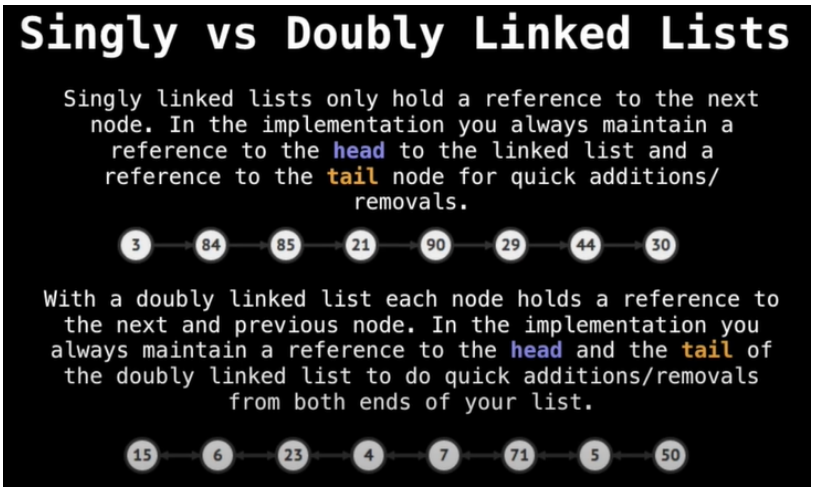

Figure 9:

Singly linked lists only contain a pointer to the next node. While doubly linked lists also contain a pointer to the previous node, which can be quite useful sometimes. (We could also have tiply or quadruply linked lists).

The pros of using a **singly linked list** are the lower memory rquirement and the simpler implementation. But its cons are that the previous element cannot be accesed. On the other hand, **doubly linked lists** can be transversed backwards, but the bad part is that the memory requirement is twice larger than with singly linked lists.

3.3 Implementation details

3.3.1 Implementing Singly Linked Lists