Data Structures 1

Table Of Contents

What are data structures

Genrally, a data structure can be just a class or some aggregate of data. This usually relates to an (ADT) Abstract Data Type, where we can construct a type (class, type, interface) which are some kind of abstract on top of the primitives.

However, when we talk about data structures, we are usually referring to how we can efficienctly access, insert, update and/or allocate/position data.

Why is it foundational?

Everything you have been using, `Array`s in JS, `List` and `Dictionary` in C# are data structures. They are powerful types that allow us to build more complex things and not bog down the computation.

What are data structures

As part of understanding and building data structures, we need to understand how we can measure time and space complexity.

- You can think of time complexity as "How long will this program run for if we have N as input"
- You can think of space complexity as "How much ram will this program take up if we have N as input"

Time Complexity

As stated above, time complexity of an algorithm will grow with the size of the input. Now depending on the algorithm, that complexity could change drammatically.

This is typically expressed using big O notation. Common time-complexities you will see with algorithms are going to be:

- O(1)
- O(log n)
- O(n)
- O(n log n)
- O(n^2)

Time Complexity

Lets go through some basics on time-complexity, lets look into these three examples and categorise the time-complexity associated.

1. *We have a loop that goes over each element*
2. *For each element, we loop over the rest of the array*
3. *We know the index and can find the element quickly*

What is a good example?

Do you remember writing a binary search?

Why does that search exist?

What benefit do we get from that?

Space Complexity

Like time complexity, space complexity is another concept to consider.

Your algorithm may be O(1) but if it requires more ram than entire data-centre, it is useless

So, we have another constraint to consider, how do we ensure that we remain efficient with ram?

Space Complexity

The same way we can reason with time-complexity, we can also observe space-complexity changes. For example, if we need to have N objects for element in our collection, we may have a space complexity of O(n^2).

Domains and Usage

What do I need to know this stuff?

The quick answer is that, without building and understanding how these things work, you will be limited in what you can create and their efficiency.

Where is it used?

- Databases
  Tables with an index, usually has a Tree data structure supporting it.
  Tables are also just arrays but we need to be flexible with representation
  of the data in the columns
- Compilers
  So, C# and Typescript compilers can be efficient we want to index files for
  quickly access.
- Mobile Applications
  Doing searches on data in memory or on disk, making sure you only display what
  is in view... with desktop and mobile applications, anything can show up.

Scenarios where it is important

- *The loading of the mobile app is very slow*
- *Why does it take so long to find something*
- *Every time I use this app, it makes my computer slow down*
- *I have to stop other apps because this uses so much ram*

...

I think you get the picture.

You can usually trace the source of the issue to a data structure or algorithm issue. This is sometimes at a time where we only know what we know but sometimes these are decisions that are outside of your control.

List Data Structures

We will be visiting `List` data structures. These will be.

- Linked List
- Queue & Stack
- Dynamic Array

Linked List

Lets start off with a problem we are trying to solve.

We want to only create as many elements as we need, these elements can be created any any time and may be removed and deleted at any time as well

Okay, so lets start with what a linked list is.

Linked List

A Linked List is a List data structure which isn't allocated in a contiguous block of memory. Instead it has two components:

- Node(s)
- Container

The Nodes are used to hold the data and refer to the next place for data.

The container holds onto the first node in the list (if it has been created).

Linked List

Lets define Node since that is the important one.

```
type Node = {
    value: number
    next: Node | null
}
```

WAIT! How can you use `Node` in `Node`'s definition?

This is because `next: Node | null` is not holding onto the value of `Node` but where another allocation exists in ram.

Remember: `next` could be assigned to `null`.

Okay, how does this work?

Let's show some code.

```
let a: Node = { value: 1, next: null };

let b: Node = { value: 2, next: a };

//How do I access 'a' without using 'a' directly?

let c = b.next; //This is 'a'
```

Linked List

We can probably see how we are able to chain `Node`s together in this fashion.

In the above example, we could access the value held by 'a' via 'b' because b.next was assigned to the value that 'a' also holds.

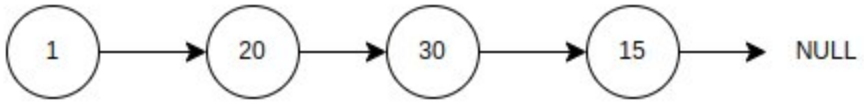
Okay, so how about the Container part?

Linked List

The container can be defined like so.

```
type Container = {
    first: Node | null
}
```

And that is all, it just simply holds onto the first node.

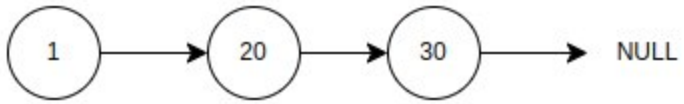


Okay fine, it isn't just that. The Container here should be a class have the methods:

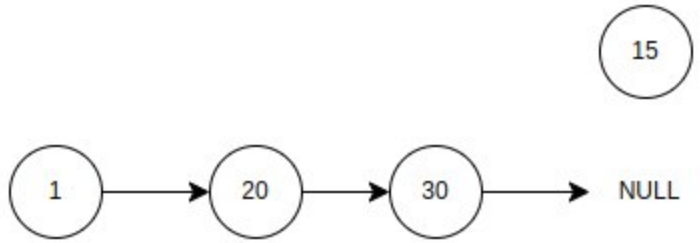
- `appendElement(value: number)`
- `getElement(index: number)`
- `setElement(value: number, index: number)`
- `removeElement(index: number)`

Linked List - Append

We want to add an element to the end of the list, in this case we need to start from `1`, move to `20` then `30` and add a new element from `30`.



After we have found the last element, we add it to the end, making `30`'s `.next` field assigned to a new node. `15` will be inserted after `30`.



We can start off with the current scaffold.

```
class Container {
    first: Node | null = null;

    getElement(index: number) {}

    appendElement(value: number) {}

    setElement(value: number, index: number) {}

    removeElement(index: number) {}

}
```

Lets go through implementing it

Dynamic Array

What would be a good problem for a dynamic array?

The user will keep inputting more strings and I want to keep as much as I can in ram, however I can only use arrays

What are one of the drawbacks with arrays? (at least with C#)

In JS/TS land, using the array type actually gives us a dynamic array.

We have three options:

- Pretend it doesn't exist (If you can't resist see the other option)
- Use a specific module created for this exercise (or if that ins't enough, you can go through the last one).
- Use `ArrayBuffer` which allows you to specify a strict size

Dynamic Array

How do we deal with the problem?

So, firstly, the input size is not known ahead of time, it will grow assuming input from the user.

- Attempt to create a dynamic array in C#. Try to describe and observe what is happening on the next page.

Consider the following pattern and what it is doing.

```
int size = 0;
int capacity = 4;

string[] lines = new string[capacity];
string? line = Console.ReadLine();

while(line != null) {

    lines[size] = line;
    size++;

    if(size >= capacity) {

            string[] temp = new string[capacity*2];

            for(int i = 0; i < (capacity); i++) {
                temp[i] = lines[i];
            }
            capacity = capacity * 2;
            lines = temp;
    }

    line = Console.ReadLine();
}
```

Dynamic Array

The pattern above demonstrates that we can allocate a larger space and re-assign the original array `lines` with a new capacity.

The `capacity` itself keeps track of the number of available spots while `size` keeps track of how many spots are used.

- Consider how costly `remove`-ing an element from a dynamic array could be if it is from the middle of it?
- With the copying of elements into the dynamic array when it resizes for a larger capacity, does this make insertions O(n) instead of O(1)?

Queue & Stack

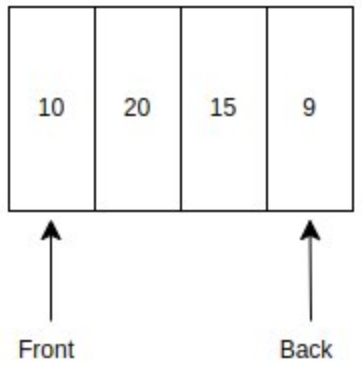
Returning back to a linked list, we can construct what are known are FIFO and LIFO data structures.

A Queue is a kind of FIFO data structure, the first elements put into the queue are the first one to be removed.

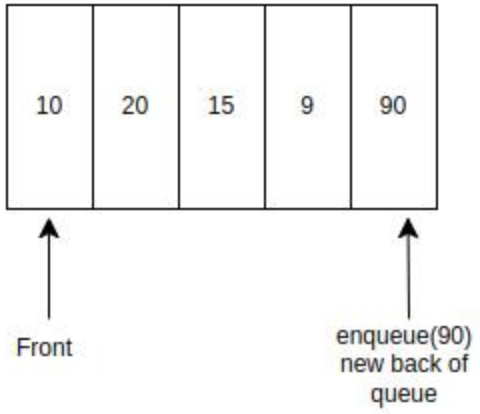
A Stack is a kind of LIFO data structure, the last element put on a stack, is the first one to be removed.

Queue - Enqueue and Dequeue

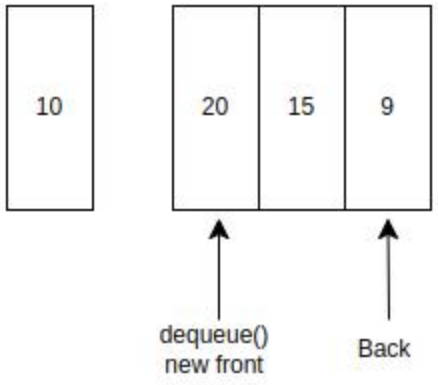
Given that a queue is a FIFO data structure, we can view the data as kind of line-up. Where we see one come after the other.



We have a `front` and `back` of a queue. `.enqueue(...)` will result in an element placed at the back.

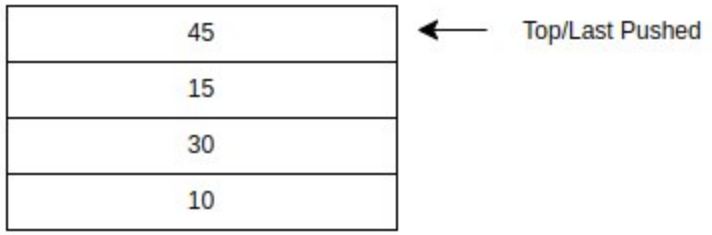


A `dequeue(...)` will result in an element being removed from the front.

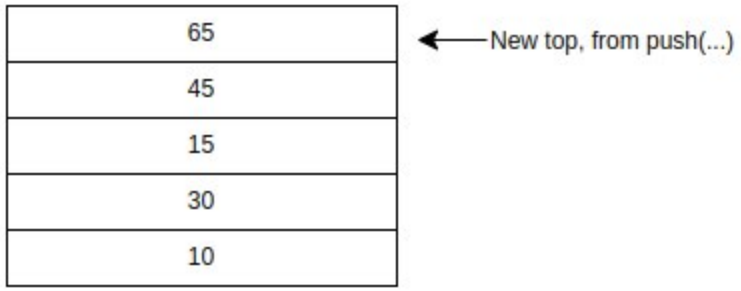


Stack - Push and Pop

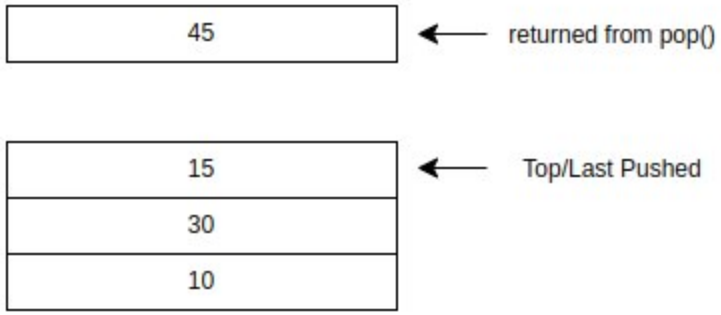
We now have a LIFO data structure a `stack`. An analogy would be to consider a set of coasters sitting on top of each other. You would only take the top one, leaving the one below as the new top.



When `push`ing onto the stack, the element wil be the new top element.



When `pop`ing an element off the stack, it will reveal a new top element.



Demo of Queue & Stack

Quick Recap on Typescript

- What does the `type` keyword do?
- What does the `class` keyword do?
- What are some primitive types in typescript?

Generics

While, JS is a dynamically typed language and we can lean into it, we want to be able to build types that have generic qualities.

It would be quite constraining if the only things we could do is have a `number[]` or `string[]`.

Hang on, do they need to duplicate the array type for different types?

Generics

No, but we have a feature in typescript (and you've seen it in C#) called Generics. Generics allow you to have a **type parameter** as part of your construct.

How do we use generics. Lets start with a simple case.

I want to create a `class` called `Thing` that will contain `T`, where `T` can be some `type`.

```
class Thing<T> {
    // snipped the rest
}
```

Generics

```
class Thing<T> {
    // snipped the rest
}
```

The above class specifies a type parameter `T`, which can be used within the definition of the class `Thing`.

Let's expand on this.

```
class Thing<T> {

    thing: T;

    constructor(obj: T) {
        this.thing = obj;
    }

    getThing(): T {
        return this.thing;
    }

    setThing(obj: T) {
        this.thing = obj;
    }

}
```

Let's work on some exercises

Resources

1. Linked List, NIST, https://xlinux.nist.gov/dads/HTML/linkedList.html
2. Stack, NIST, https://xlinux.nist.gov/dads/HTML/stack.html
3. Queue, NIST, https://xlinux.nist.gov/dads/HTML/queue.html
4. time/space complexity, NIST, https://xlinux.nist.gov/dads/HTML/timespace.html
5. time-complexity, Python Wiki, https://wiki.python.org/moin/TimeComplexity