**Lecture 44 & 45 – LINKED LISTS**

Arrays

Why are arrays good?

* They can store multiple things
* They are indexed:
  + Allows us to get any elements in the array at the same time (Random Addressing)
  + With arrays, we’re able to get any element in the array in the same amount of time (one step) i.e:
    - **pointerArrayStart + ( sizeof(arrayElements) \* n )** to find the nth element of an array.

Why are arrays bad?

* You can’t easily insert elements without **shuffling everything around.**
* Not as meaningful to view it as a linear sequence. There are **other relationships between some elements** that aren’t captured.
* Arrays have a **defined ending (fixed size)**
* Elements must be **all the same size and type**.
* For the compiler to find an index, it has to know:
  + What the index is
  + How big each element is
* Inefficient use of memory:
  + Each element must have enough memory to store the largest amount of data that the array will ever receive.

Linked Lists

LINKED LISTS = Treasure Hunt 🡪 One clue / step at a time

ARRAYS = Scavenger Hunt 🡪 Access everything at once

**Lecture 46– PROJECT DESIGN**

Agile Programming

Focus on the principle of decoupling and coherence

* **Decoupling**
  + Decoupled functions are not heavily interrelated with each other and can be easily re-used with other functions when necessary
* **Coherence**
  + Things in the same area work together while things in different areas do different things.

When the spec changes, just change our code to get it to work.

* OLD Richard
  + Why would you write code in a way such that when you change, you would break everything. Everything would be a nightmare because the cost of change is so high it’s terrifying
* PRESENT Richard
  + Make a suite of tests, that tell us when the code is right or not, so just make small changes that happily breaks our code. Following the principles of decoupling and cohesion, it is actually quite fast to change the code, especially with the help of our tests.

You can quickly rebuild your code to make sure your tests work (functionality is back in running order)

* Then you change it so that the code quality becomes awesome (refactoring) i.e. go through the code, decouple and move coherence around.

Richard’s Rule of Thumb

Make it concrete the first time

* When you see two instances of it, make a call whether to make it general or concrete.
* If you see three instances of it, you should really make it generalised.

The philosophy of Agile:

* Make your program flexible, but not at the cost of simplicity
* E.g. if the test only tests linked lists of length 1, you’re not forced to write tests of longer than length one

DRY (DON’T REPEAT YOURSELF)

* Think about refactoring and making things more coherent.

Why waste time making features of far greater complexity than it needs to be for a simple feature that the client needs?

You should not write your program so that it anticipates changes in spec 🡪 Just change as you go.