Arrays

O(1) retrieving

Arrays have Fixed length

```
int[] array = new int[10]
                  ↑
              fixed size of 10
```

resizable arrays can grow as needed

some languages (such as Python) only have resizable arrays

=

Use resizable when you don't know how many elements you need upfront

Underneath it is an array and acts like it, but when the array gets full we create a new array that's double the size of the old array and copy the elements over

Copying all elements is really slow, but cost is ammortized, only happens when we hit capacities so in average adding is still constant time, even though sometimes is linear

Strings

sequence of characters, almost like an array

You can access the ith element immediately like with an array

unlike arrays however, you cannot change the string (in many languages)

When we concatenate a bunch of strings.

```
str = a      "coding" + "is" + "is" + "fun"
str += b       ↑e       ↑b     ↑c    ↑d
str += c
str += d  ← really inefficient!
            create a new string
            every time
```

Create an array or arraylist of all the strings and every time you concatenate one just add it to that list, tracking all the while the total number of characters you need. When actually need the string out, go and merge all the strings together at once

```
StringBuilder sb = new StringBuilder()
sb.append(a)        ↑ most languages
sb.append(b)          should have
sb.append(c)          something like this
sb.append(d)
```

Hash tables

* For any problem, have hash tables at the top of your mind as a possible technique to solve the problem

A hash table, at a high level, is a key value lookup.

So it gives you a way of, given a key, associating a value with it, for very very quick lookups.

hashTable.put("Katy", new Person(...))
hashTable.get("Katy")

In a hashTable, the key as well as the value can be basically any type of data structure, as long as it has a hash function!

=

At a high level we want to store the objects in an array

A hash function takes a string (generally an object), converts it into some sort of integer, and then remaps that integer into an index into that array. That's where we can find our value

The hashcode is not actually the index in this array, we map from the key to the hashcode, and then over to the index

string → hashcode → index

That's because the array that actually stores the data from the hashtable, might be much much smaller than all the available potential hash codes

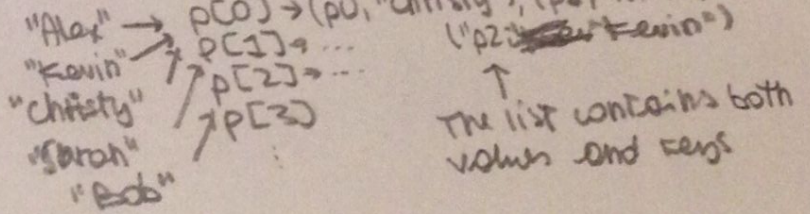
Two strings can have the same hashcode, that's because there are an infinite number of strings, but a finite number of hashcodes. Additionally since we're remapping the hashcode into an even smaller index, two things with different hashcodes could actually wind up mapped to the same index

That's called a collision!

Chaining is the most common way to solve collisions,

with chaining, when there is collisions, just store them in a linked list

When looking for the value of "Alex", you actually need to look all the values in that linked list and pull out the value for "Alex".



O(1) for a "good" hashtable
O(n) for a terrible hashtable ← what we assume
↑ lots of collisions, worst case!

Linked Lists

A linked list is essentially a sequence of elements where each element links to the next element, which links to the next element and so on...

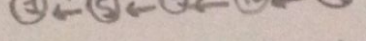
It can contain pretty much any type of data, elements can be sorted or unsorted, it can contain duplicate elements or all unique elements

In an array elements are indexed, a[4]
In a linked list you have to start with the head and work your way through, until you get to the fourth element!
↑
That takes linear time

Advantages
insertion and deletion can be quick!

O(1) prepend
O(n) append

DOUBLY LINKED LIST



Each element also links to the previous element

Stacks and Queues!

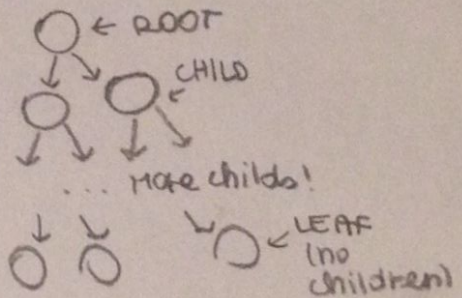
- linear data structures
- Flexible size

The main difference is in how elements are removed:

A stack is LIFO

A queue is FIFO

Trees



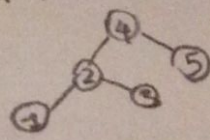
Binary Tree
Each node has no more than two child nodes

Binary search tree
A binary search tree is a binary tree where on any subtree the left nodes are less than the root node, which is less than all of the right nodes

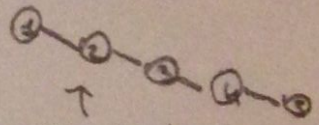
This makes finding a node O(lg n)

Insert
start with some element you want to insert. Check whether is bigger or smaller than the the root, if bigger go to the right, else to the left. Do this over and over until you get to an empty spot and insert the element.

Our tree could get really unbalanced!



An unbalanced tree has slower inserts and finds



↑
Depending on the order of insertion

Traversing

Inorder
left than root then right

Preorder
root than left than right

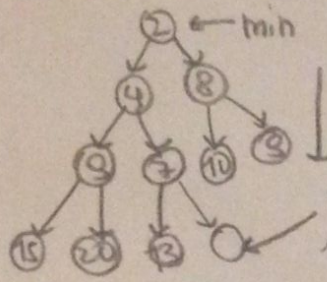
Postorder
left than right then root

Inorder gives us the elements in sorted order in a binary search tree!

MIN or MAX

Heaps

In a min heap, the elements are always smaller than their children



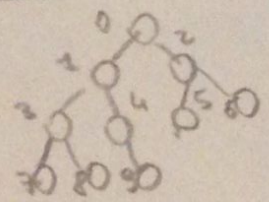
getting bigger

inserting an element, it always goes in the next empty spot looking top to bottom left to right.

.. And then bubble it up until we get to the right spot.
- compare with parent ...
- if out of order, swap them
- keep going up the tree

Removing the minimum element,
- Remove the min element
- Swap empty root with the last element added
- Bubble down root
  - compare with left and right child
  - swap it with the smaller
  - keep going down until the heap property is restored

An heap is visualized as a binary tree, but we can actually use an array to store the values since the heap never has gaps (because of the way elements are added)



PARENT = (index-2)/2

index

LEFT = index*2+1

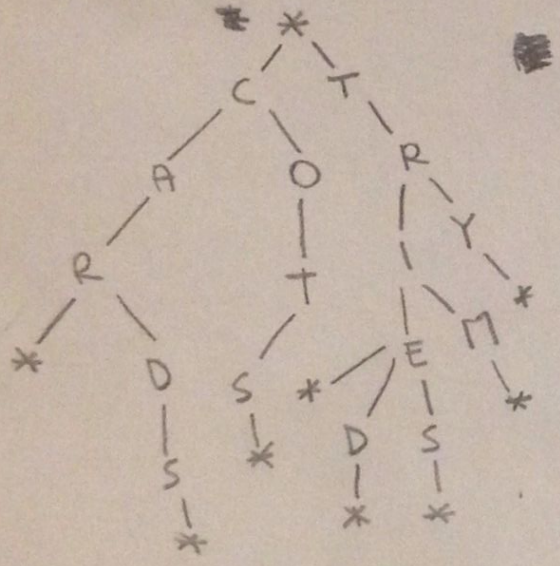
RIGHT = index*2+2

Tries

- Data-structure which is actually a type of tree
- Often used to store characters

Each nodes stores a character and the path from the root to that node is a word or part of a word

Very quick look-ups of a particular kind



Words starting with "car-"
car card cards!

Use where you need some sort of word validation

- Walk through scrabble board and find all the words
- Given list of strings, find all celebrity names

Instead of looking up from the root, every time when checking a string, we want these calls to build on each other

We can keep states in various ways

By:

- Keeping state within trie
- Return the node reference

```
class Node {
    HashMap<Character, Node> children;
    boolean isCompleteWord;
```

Bubble sort

[redacted]

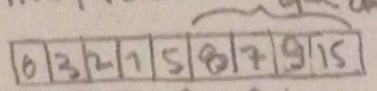
Really naive algorithm, not one we would want to use in the real world

Walk through the array and everytime we see elements that are out of order, we swap them

By doing it enough times, our array will become sorted $O(n^2)$ but in place

For example, you have a sorted array, and one element gets incremented, then one walkthrough a bubblesort could then sort the array efficiently

Quicksort! very efficient (randomly)
- pick some element to be the pivot element
- walk through the array and swap elements around such that the elements less than the pivot come before elements bigger than it
- repeat the process for the left and right portion



choose new pivot and repeat for each half

- Eventually we will wind up with a sorted array as it gets more and more sorted each loop

$O(n \log n)$
each element gets quicksort called on it, $\log n$ times

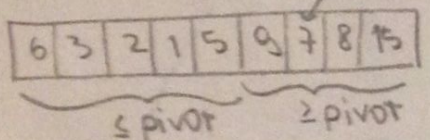
IF we pick a bad pivot everytime (very first or last element in subarray)
$O(n^2)$ runtime

As long as we're smart about how we pick the pivot element, we get [redacted] runtime efficient

QUICKSORT

Really popular and efficient sorting algorithm

- Take array of, say integers
- Pick a pivot element
  (For now we assume is picked randomly)
- Walk through the array and swap elements around such that all elements less than the pivot, come before the elements bigger than it



- Repeat the process to the left and right portion over and over and eventually our array just becomes sorted

EFFICIENCY

In an ideal world in quicksort, we're dividing the array in half each time we pick a great pivot that really is roughly the median, and then half the elements gets pivoted to one side of the array and half of the get pivoted to the other and then apply quicksort to each half

∴ In that case we get O(nlogn)
(each element gets quicksort called on it logn times, and each one of those is one swap. So n elements go through logn swaps)

However in the bad case, we pick a really bad pivot, like, everytime we pick the pivot happens to be the lowest element in that subarray, then we have n^2 calls to quicksort, therefore O(n^2)

As long as we're smart about how we pick the pivot element, we get a pretty efficient runtime, and that's why we implement quicksort in the real world

MERGESORT

Pretty efficient sorting algorithm

The best way to conceptualize merge sort is recursively

Suppose we have this large array, and we want to sort it
what if we could magically sort the left half and the right half?
Then if we wanted to make the whole array sorted, all we have to do is merge those in in sorted order

What we do to sort the left and right half is just apply merge sort again. By doing it over and over we'll get a sorted array eventually.

```
mergesort(array) {
    mergesort(array's left half)
    mergesort(array's right half)
    merge left and right half in sorted order
```

Always gives O(nlogn)

But merging two arrays together generally requires extra space → O(n) space

copy all the elements into a new array, merge them and copy them back