Chapter 2 - Selection Sort

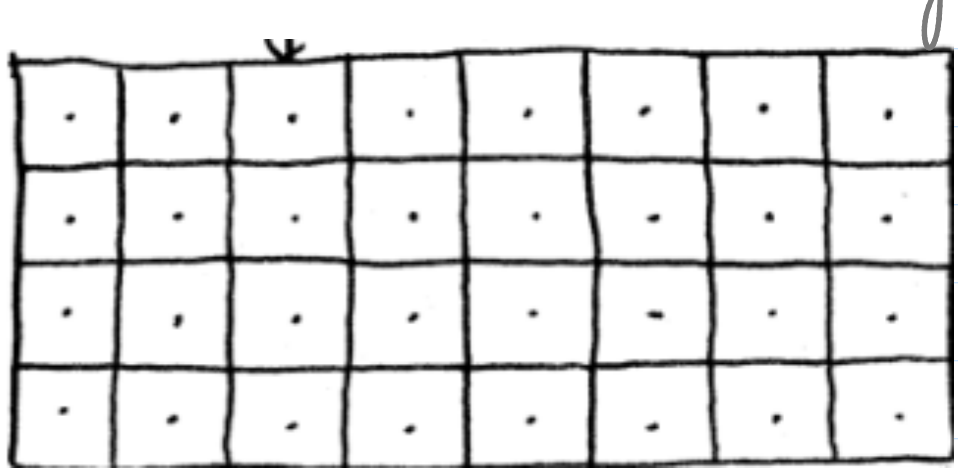
↳ What's Covered?

↳ Arrays
↳ Linked Lists
} Most basic data structures

↳ Learning your first sorting algorithm — selection sort

How does memory work?

↳ Think of your computer's RAM (random access memory) as a big set of drawers...



Each drawer represents a slot, you can store anything in these slots.

↳ Each slot also has an address that allows us to find the slot.

Each time there's more memory needed, then more slots will be used.

If you wish to store more than 1 item:

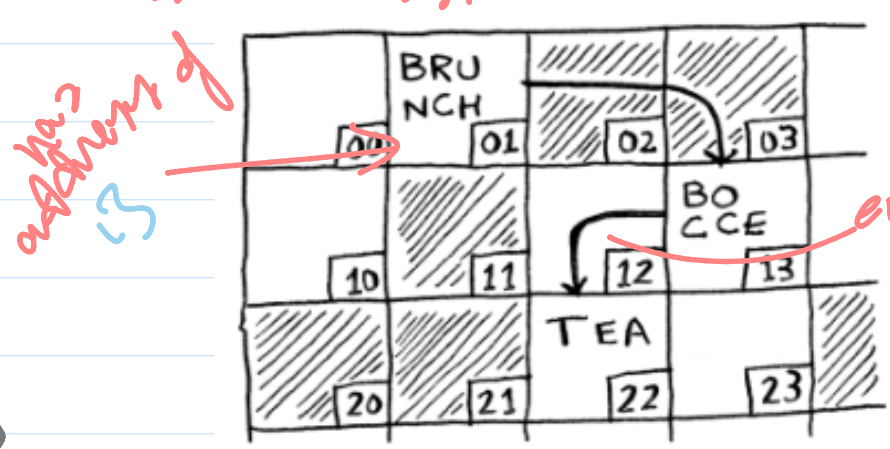
↳ Arrays
↳ Linked Lists

Using Arrays

- Stores elements contiguously (right next to each other) in memory.
  - Requires for all chunks of elements to be stored in chunky assault.
    - Think of this idea to be similar to finding seatings with your friends right next to each other.
- Can be a pain especially when adding new elements.
  - A good way to fix this issue would be to hold spaces in the array so it can avoid modifying the array (move/delete data).
    - Downsides
      - You may not need extra slots the entire time meaning: wasted space
      - You may add more space to hold, but could have to move inefficient

The Solution: Linked Lists

- Data can be stored anywhere in memory.
  - Each element holds the address of the next element in the list



Linked memory addresses

Like a chain of elements.

If your memory has enough space for n elements, then then linked lists can chain & connect all those elements.

↳ the problem of Linked Lists

↳ Only shows the information of the upcoming element since you only have access to the address of the next element. (info)

Limits access to all elements (depending on element position)

For Arrays, this is NOT an issue.

↳ You get to have access to all elements in a Array.

Linked Lists: NOT ordered → less access
→ easy to input data

Arrays: ordered → hard to manage data space
→ access to ALL elements

Index = Position of Element

Runtimes for common operations on Arrays & Linked Lists

blc	ARRAYS	LISTS
READING	O(1)	O(n)
INSERTION	O(n)	O(1)

easy to read blc full access to all elements

hard to read list blc ith is one by one

fixed size for always!

easy to just insert for Lists

O(n) = LINEAR TIME
O(1) = CONSTANT TIME

Inserting elements to the middle of a list

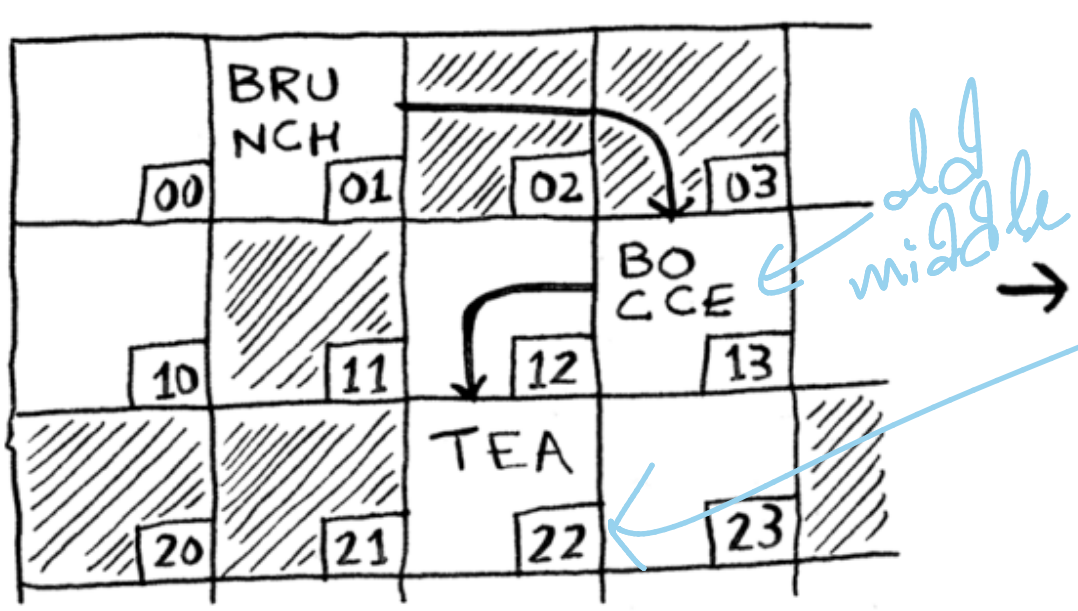
↳ Linked Lists

↳ Easy, change the address from the element that was the middle before to the new element's address

old middle

to become the middle of the list

Diagram for more description



BEFORE

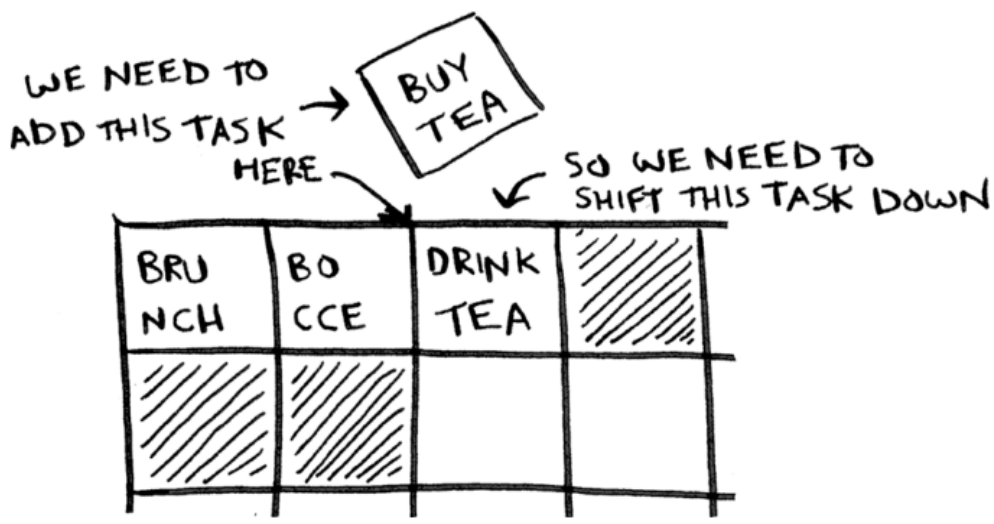
address contained in slot is changed from 22 to 14

new middle

AFTER

For Arrays:

↳ You have to shift all the elements down/up by one slot



No Space?

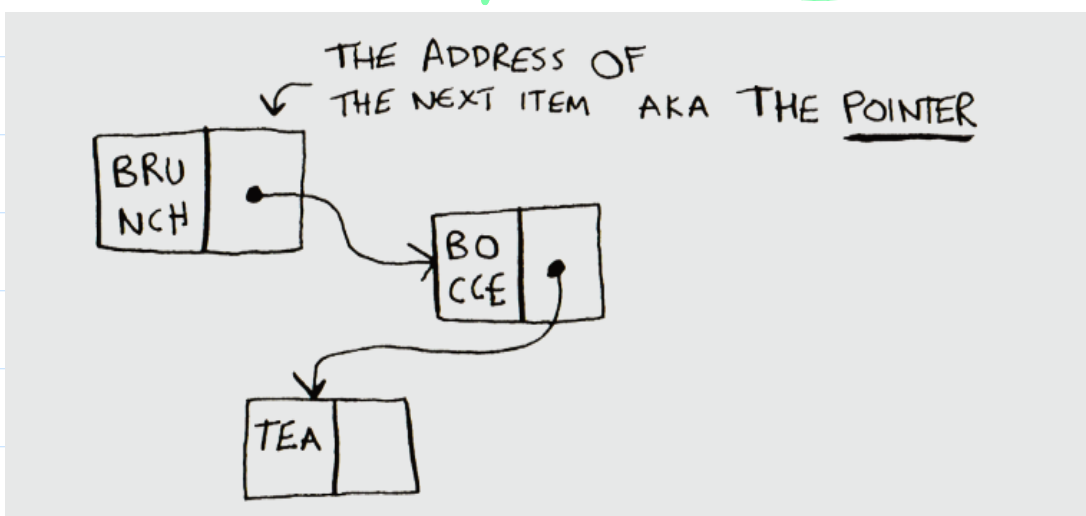
↳ Find a new location in array to fit them all in

Continuing on from Linked Lists

↳ Pointers

↳ The way how each element is "linked up" or pointed to the preceding element

↳ Uses up a miniscule amount of memory to store the next address of the next element



Deletions

↳ Unlike insertions, deletions will always work

- Insertions sometimes fail because of the lack of space in RAM

↳ Runtimes for Reading, Insertion, and Deletion

	ARRAYS	LISTS
READING	O(1)	O(n)
INSERTION	O(n)	O(1)
DELETION	O(n)	O(1)

O(1) (immediate) — why? ↳ independent of input size and bounded to constant "c" i.e. 1

O(n) - linear time

- Insertions & Deletions are O(1) only if you can instantly access the element you wish to delete

↳ It's common practice to keep track of the first and last elements in a linked list in order to get the O(1) time

Arrays vs Linked Lists: Which is used more?

↳ Arrays

↳ Often used because...

- they provide random access (what?) inherently
- Better reads (access-wise)
- Faster than Linked Lists

↳ why? -> Caching

↳ In this sense since the Array consists of putting data as a group (right next to each other) by definition, the CPU can read the Array as a group, meaning it allows faster processing

what is Caching?

↳ Storing data in an easy-to-access location for efficiency

Arrays provide this ↓ Random Access

allows you to jump directly to any element

↳ Lot of times, you'll use this

Linked Lists can only do this ↓ Sequential Access

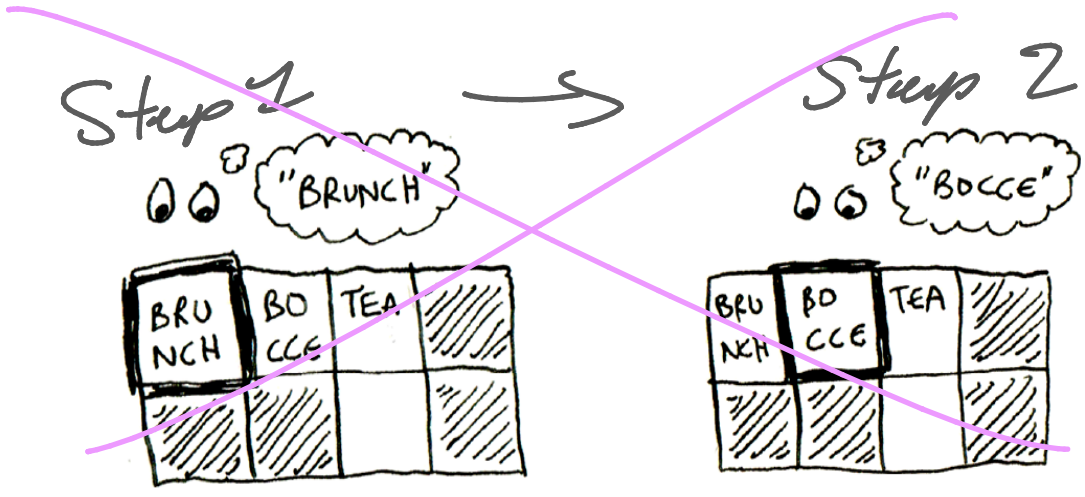
Reading elements one-by-one

- Starts at the first element

The Computer Reads Arrays as Sections (Groups)

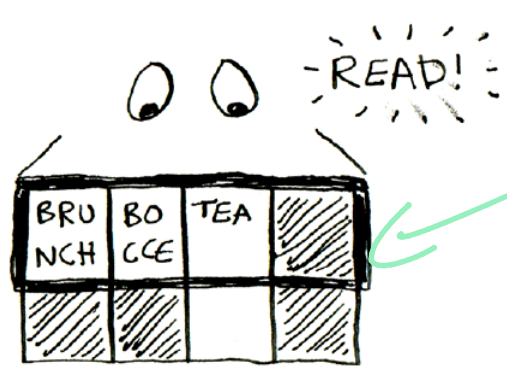
↳ Meaning it does not read the Array's elements one-by-one, but rather as groups

↳ What you may think the process works...



FALSE

↳ How it really works...



Faster! processing ↓ Easy-to-access

fallaway & track elements

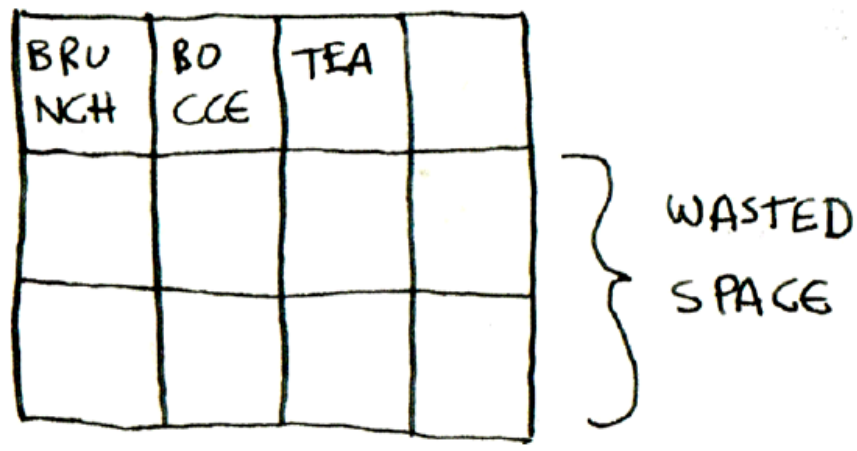
Above ONLY works for Arrays, NOT Linked Lists

↳ Remember, you DON'T know (can't read), what data the next element has, only you know where the next element is at.

memory address

Memory Consumption: Arrays vs Linked Lists

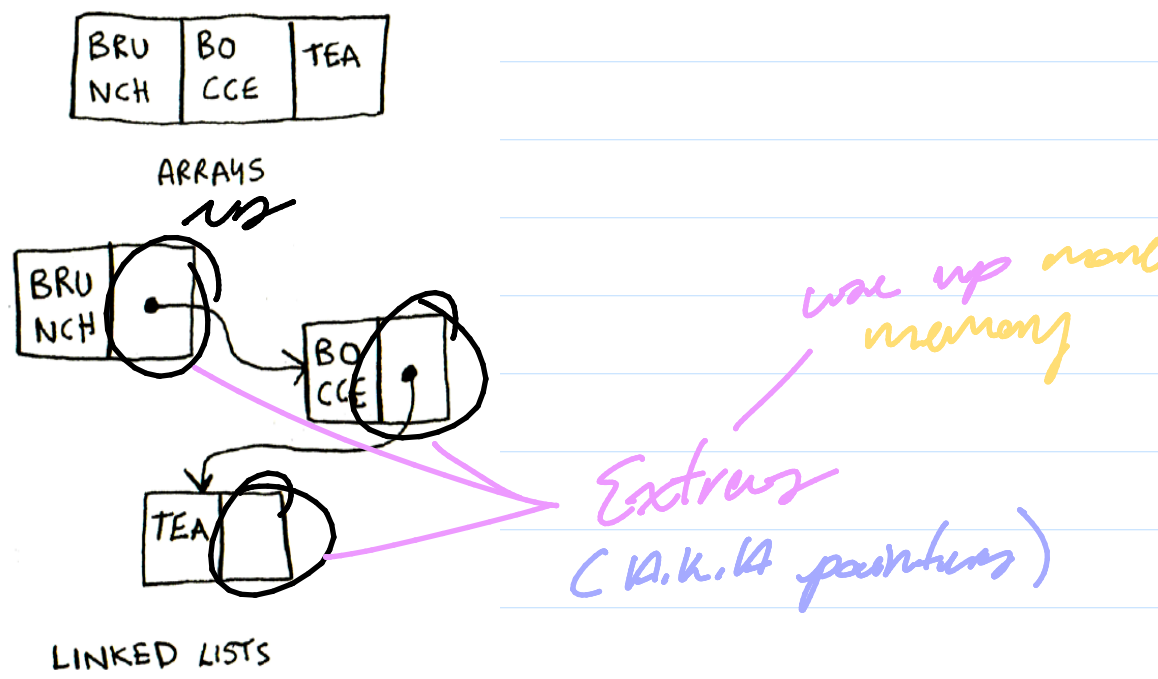
↳ You might think that Arrays will use much more memory space because of potential unused space...



Well not really, there is less wasted space. only if elements contain little to 0 data

But with Linked Lists, you use much more memory per element because you use more memory for pointers

↳ The only remedy for this would be if the data stored in each element is big enough to compensate for the inefficiency-cost. problem



In conclusion for the debate (Arrays vs Linked Lists) == So arrays are used more often than linked lists except in specific use cases.

Selection Sort (Algorithm)

↳ Pre-Req: Arrays & Big O Notation

↳ Suppose you have a list of music you want to organize by most listened to least listened to

greatest → least

~♫~	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

To actually organize this list in the way we want, we would:

1: Read the entire list in search for the current greatest listened to song

2: Add the current greatest to song to a new sorted list and remove the song off the original list

REPEAT until org. list has 0 elements

In add...

new list: sorted

~♫~	PLAY COUNT
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

removed off original list

SORTED	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141

← added second

1. KISHORE KUMAR IS THE NEXT MOST-PLAYED ARTIST — read the org. list first

2. SO IT IS THE NEXT ARTIST ADDED TO THE NEW LIST — added to new list second

Result?

	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
WILCO	111
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
THE BLACK KEYS	35

In conclusion...

↳ We read the original (org. list) in $O(n)$ b/c we needed to read every element first.

linear time

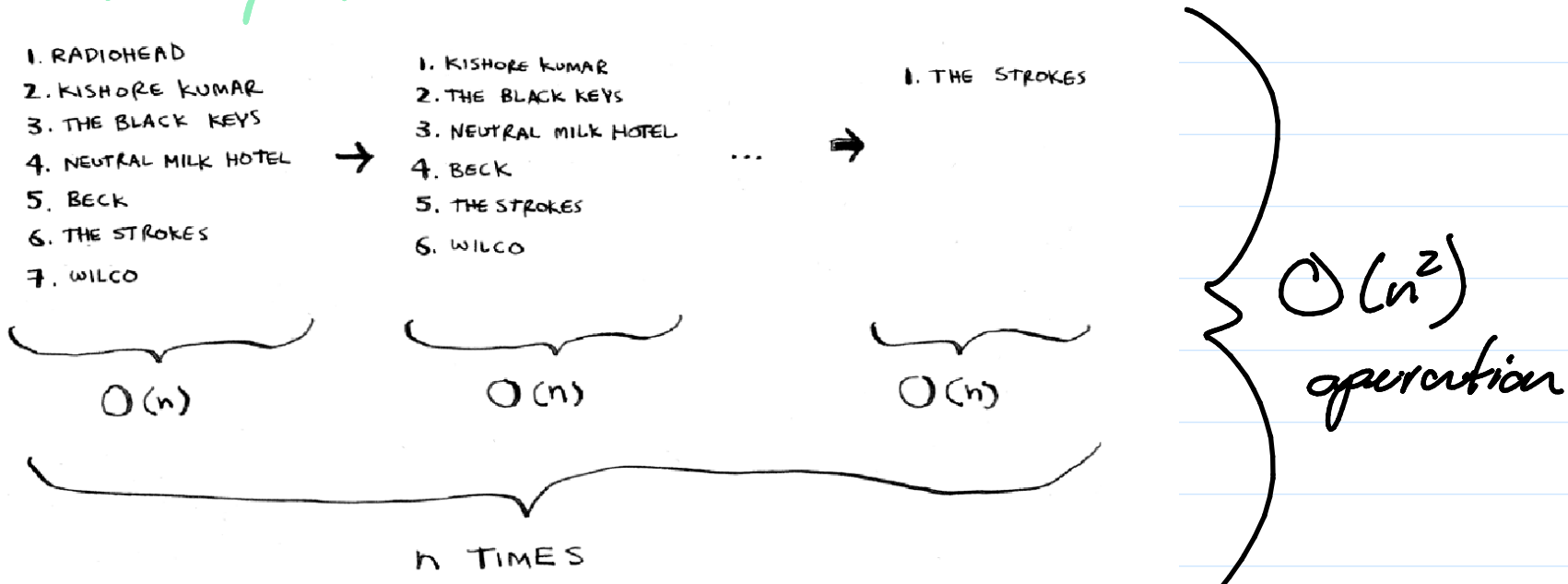
n elements, parse through each? ↳ $O(n)$

Reminds you of something? ↓ Simple Search

Then, we have to parse through the the entirety of the original list by n times. Thus we add first (to the new "sorted list") and delete (from org. list)

↳ b/c of n elements

↳ $O(n)$ to read AND do it n times to parse. $\Rightarrow O(n \cdot n) \Rightarrow O(n^2)$



Selection Sort is very neat to use, however, $O(n^2)$ time is incredibly slow... don't worry though, there is alternative coming in Chapter 4...

Recap

- Your computer's memory is like a giant set of drawers.
- When you want to store multiple elements, use an array or a linked list.
- With an array, all your elements are stored right next to each other.
- With a linked list, elements are strewn all over, and one element stores the address of the next one.
- Arrays allow fast reads.
- Linked lists allow fast inserts and deletes.