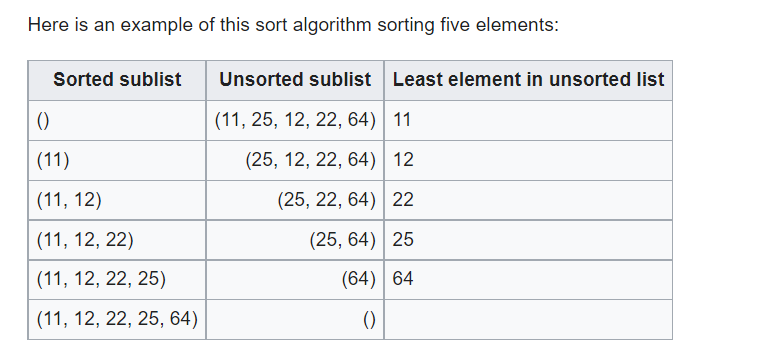
SELECTION SORT

In computer science, selection sort is an in-place comparison sorting algorithm. It has an **O(n2) time complexity**, which makes it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

The algorithm divides the input list into two parts: a sorted sublist of items which is built up from left to right at the front (left) of the list and a sublist of the remaining unsorted items that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The **algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist**, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.



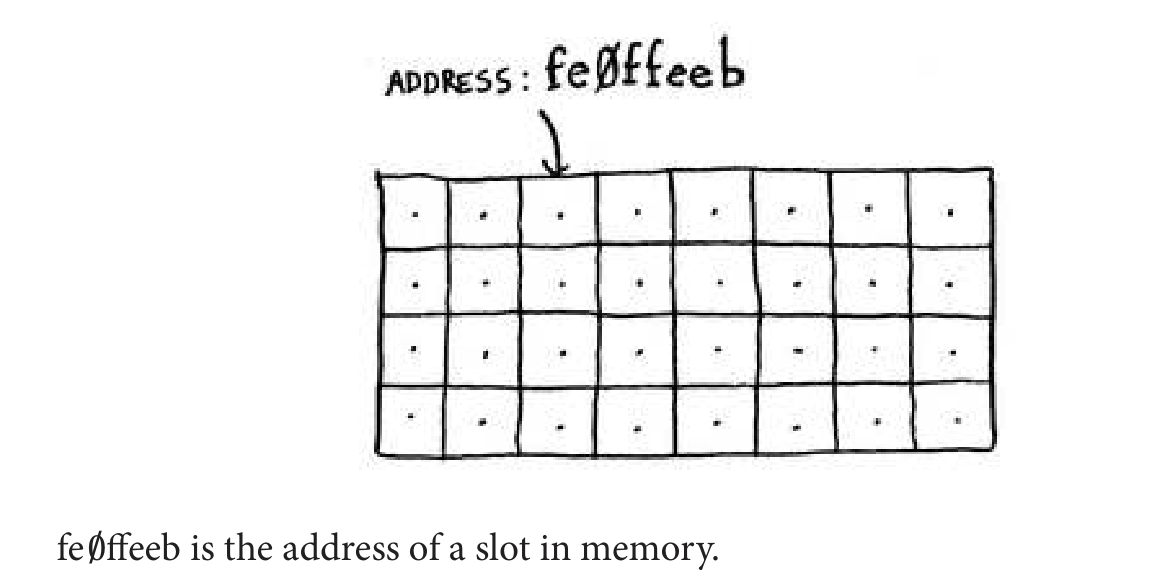
The time efficiency of selection sort is quadratic, so there are a number of sorting techniques which have better time complexity than selection sort. One thing which distinguishes selection sort from other sorting algorithms is that it makes the minimum possible number of swaps, n − 1 in the worst case.

Each time you want to store an item in memory, you ask the computer

for some space, and it gives you an address where you can store your

item. If you want to store multiple items, there are two basic ways to

do so: arrays and lists.



**Arrays and linked lists**

It’s like going to a movie with your friends and finding a place to sit—

but another friend joins you, and there’s no place for them. You have to

move to a new spot where you all it. In this case, you need to ask your

computer for a different chunk of memory that can it four tasks. Then

you need to move all your tasks there.

If another friend comes by, you’re out of room again—and you all have

to move a second time! What a pain. Similarly, **adding new items to**

**an array can be a big pain**. If you’re out of space and need to move to a new spot in memory every time, adding a new item will be really slow. One easy fix is to “hold seats”: even if you have only 3 items in your task list, you can ask the computer for 10 slots, just in case. hen you can add 10 items to your task list without having to move. his is a good workaround, but you should be aware of a couple of downsides:

• You may not need the extra slots that you asked for, and then that

memory will be wasted. You aren’t using it, but no one else can use

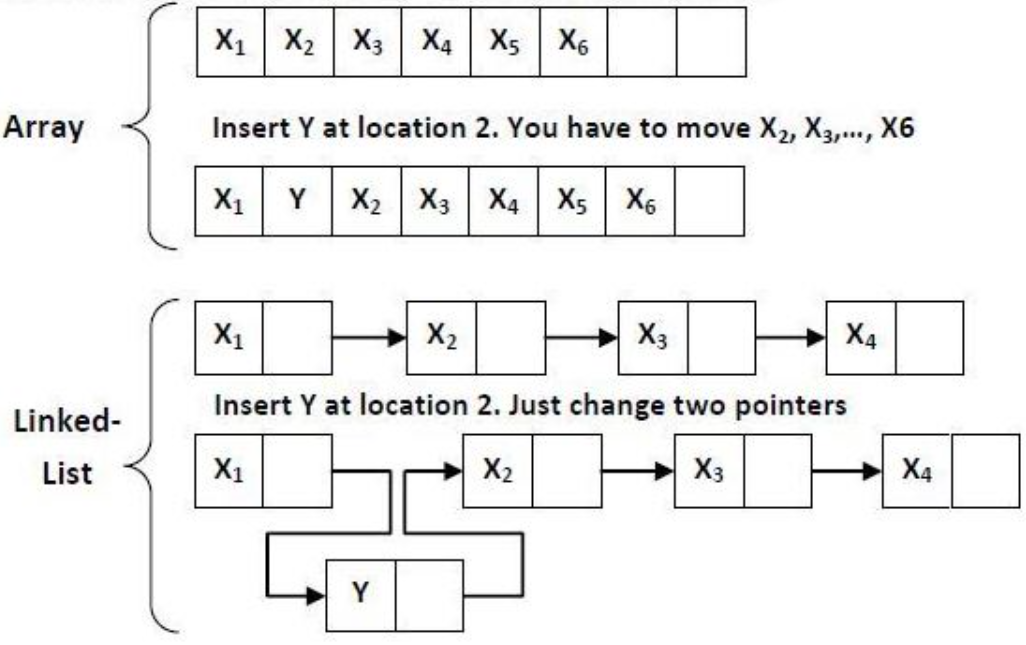
it either.

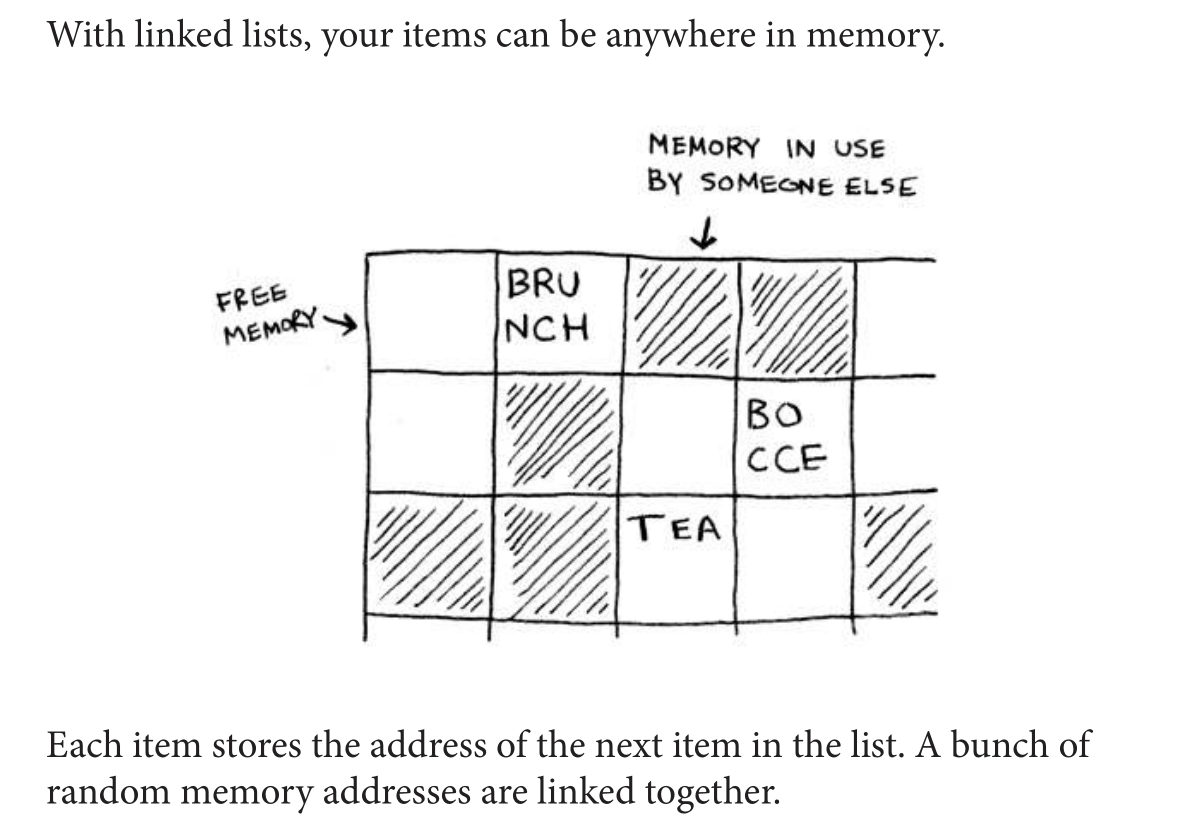
• You may add more than 10 items to your task list and have to

move anyway.

So it’s a good workaround, but it’s not a perfect solution. **Linked lists**

**solve this problem of adding items.**

****



It’s like a treasure hunt. You go to the first address, and it says, “he next

item can be found at address 123.” So you go to address 123, and it says,

“he next item can be found at address 847,” and so on. Adding an item

to a linked list is easy: you stick it anywhere in memory and store the

address with the previous item.

With linked lists, you never have to move your items. You also avoid

another problem. Let’s say you go to a popular movie with five of your

friends. the six of you are trying to find a place to sit, but the theater

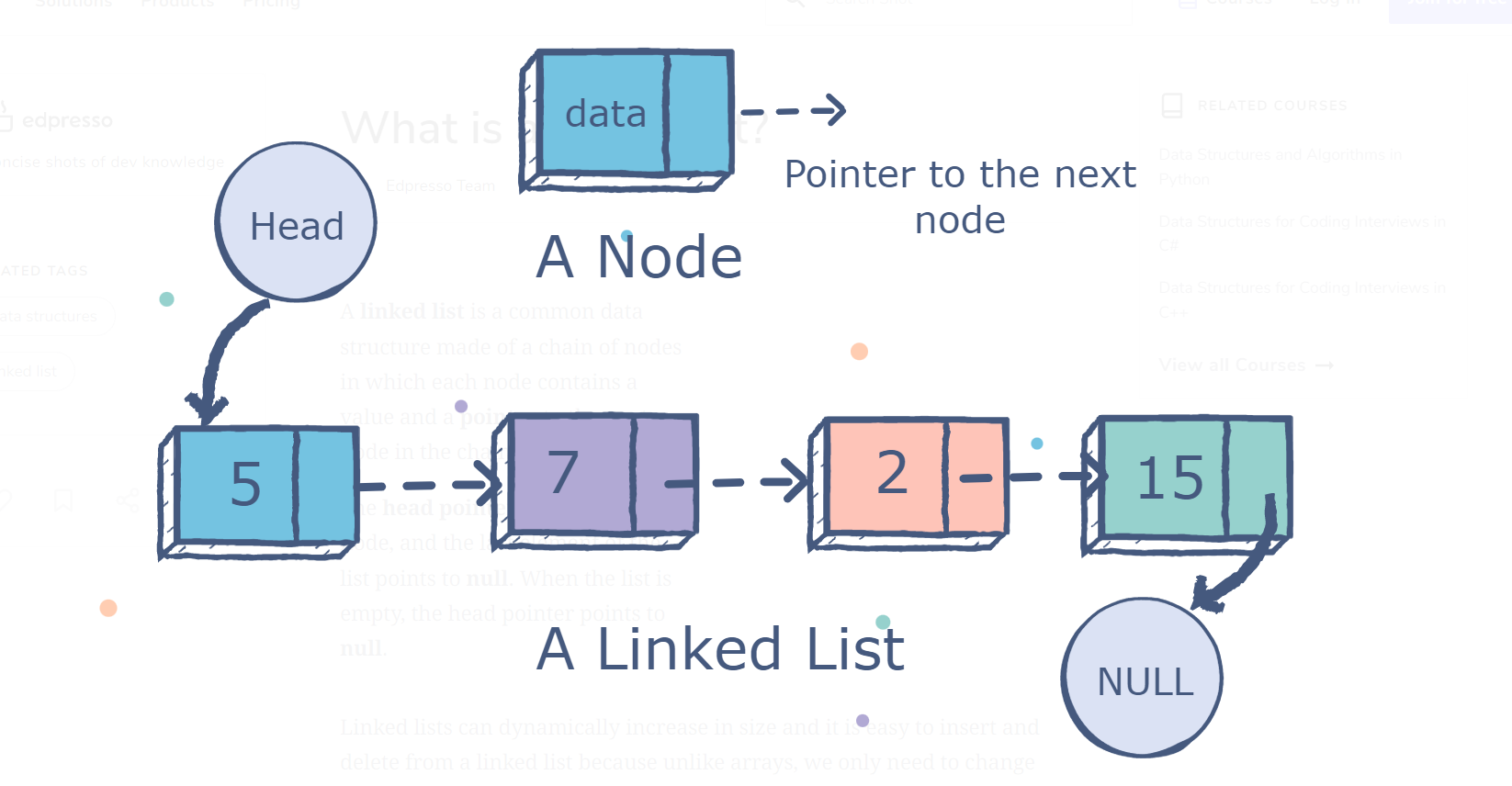
is packed. here aren’t six seats together. Well, sometimes this happens

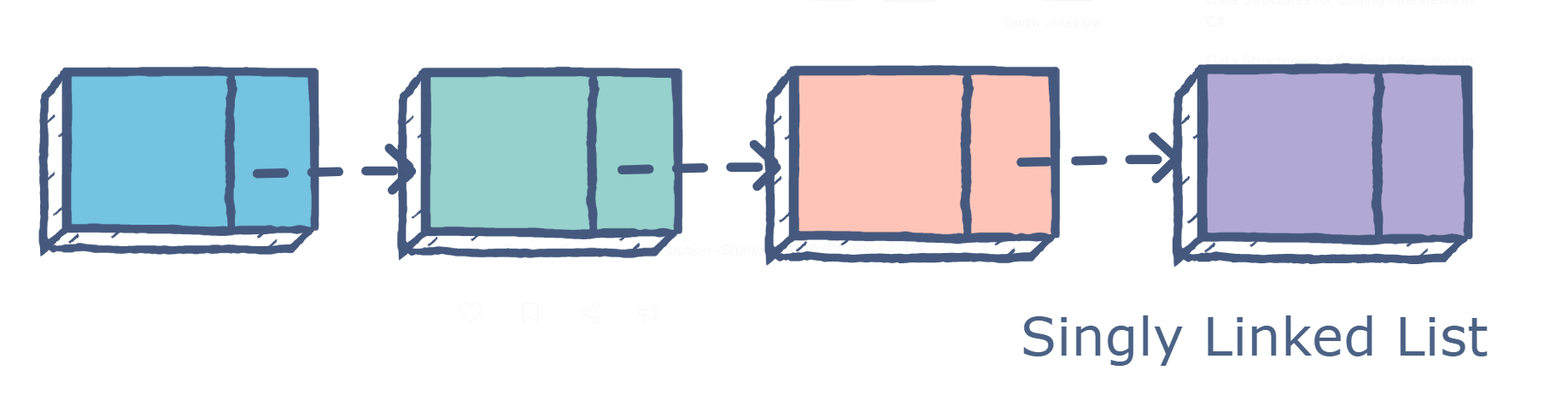
with arrays. Let’s say you’re trying to find 10,000 slots for an array. Your

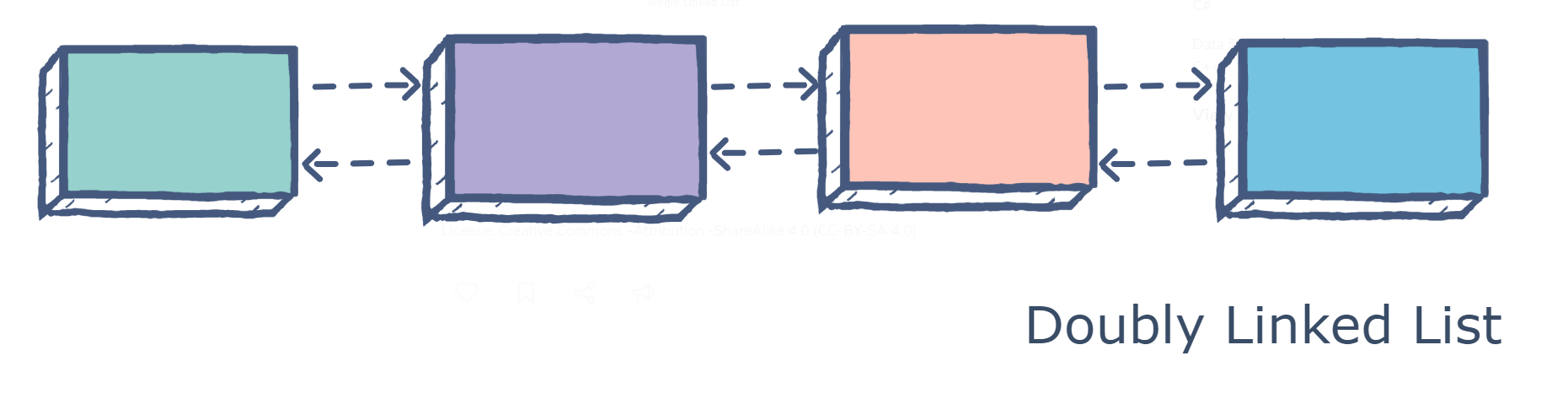
memory has 10,000 slots, but it doesn’t have 10,000 slots together. You

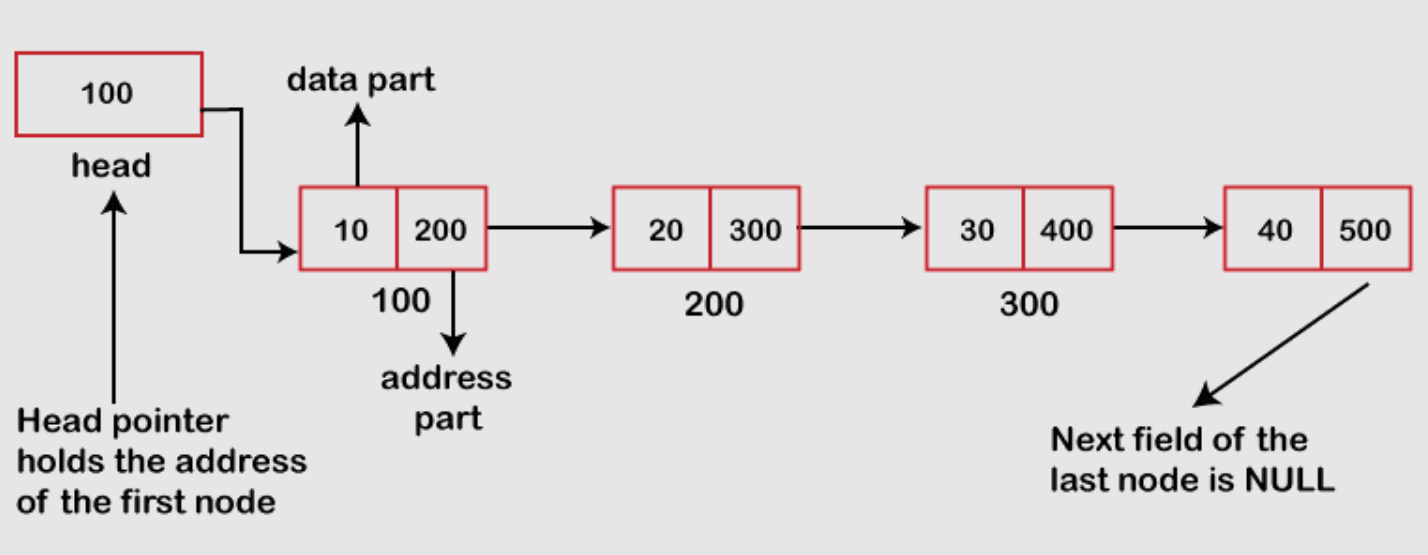
can’t get space for your array! A linked list is like saying, “Let’s split up

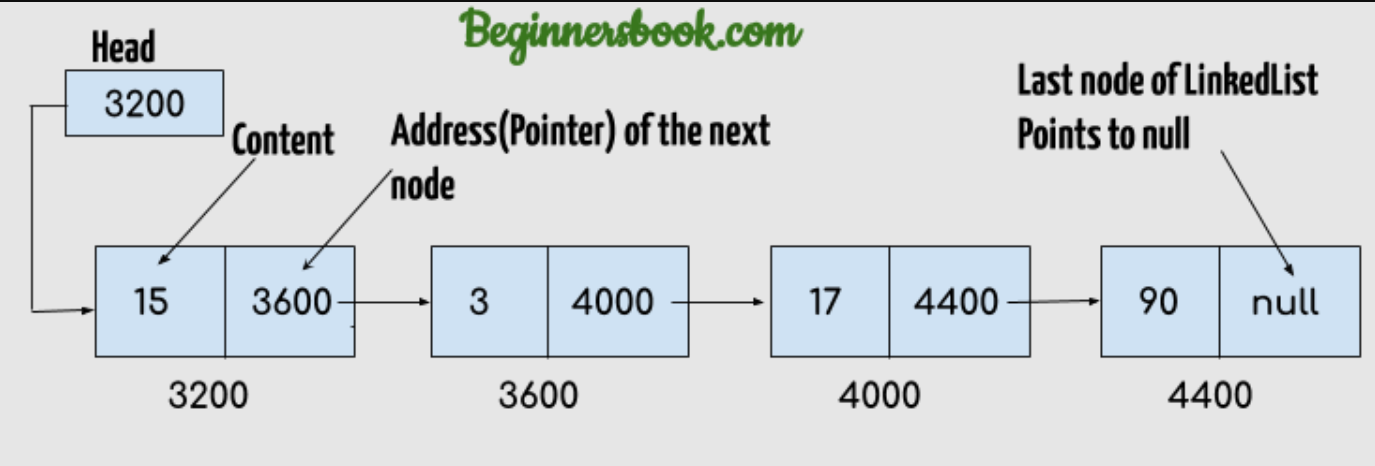
and watch the movie.” **If there’s space in memory, you have space for your linked list.**

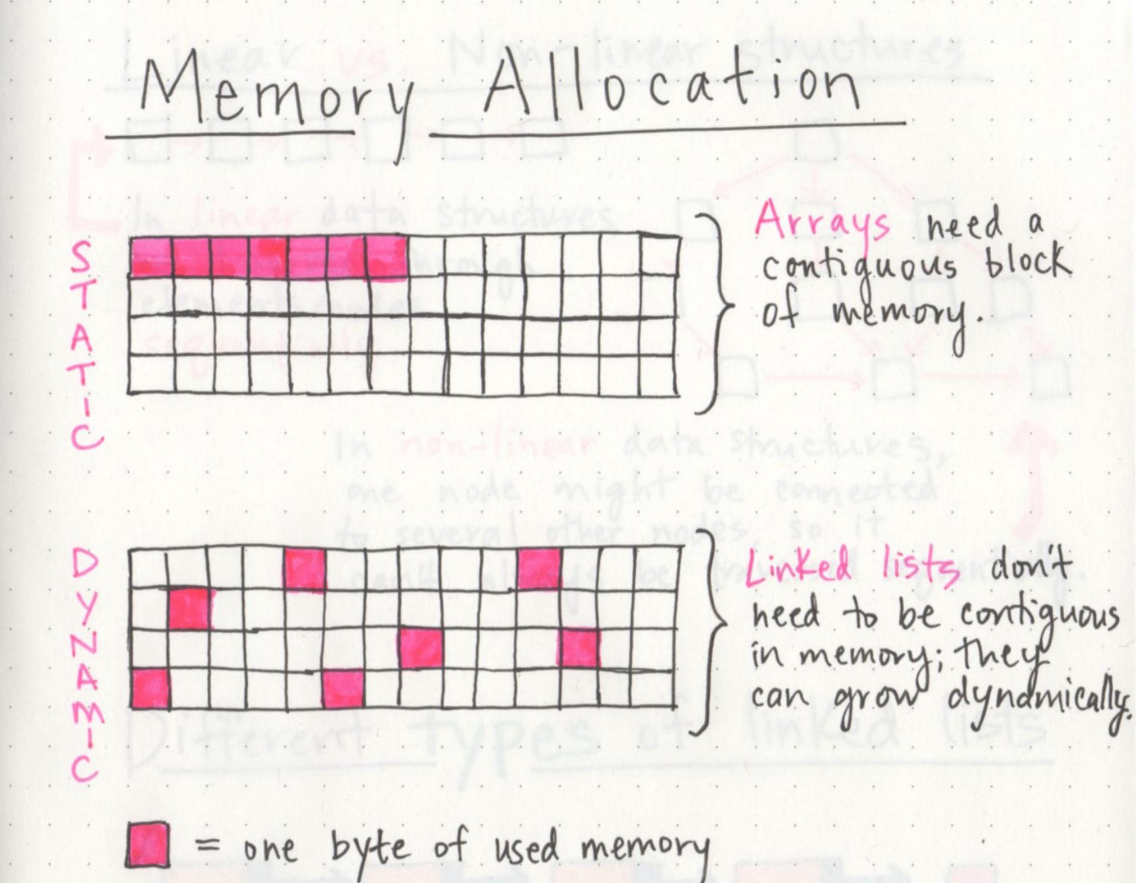
****

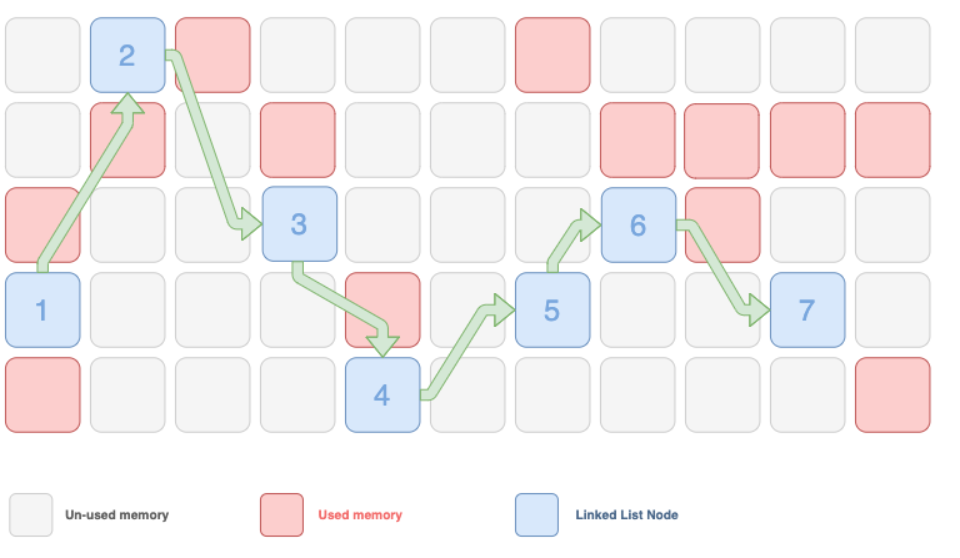
****

****

****

****

****

****

1. Arrays have a fixed size, but linked lists are dynamic and flexible and can expand and contract their size.
2. In arrays, to find the particular element, you can just find it with index, but in a linked list, you have to start your way from head and go through until you get to the fourth element.
3. Accessing an element in an array is fast, while a linked list takes linear time, so it is slower.
4. Operations like insertion and deletion in arrays consume a lot of time. On the other hand, the performance of these operations on linked lists is fast.
5. In addition, memory utilization is inefficient in the array. Conversely, memory utilization is efficient in the linked list.

If linked lists are so much better at inserts, what are arrays good for?

**Arrays**

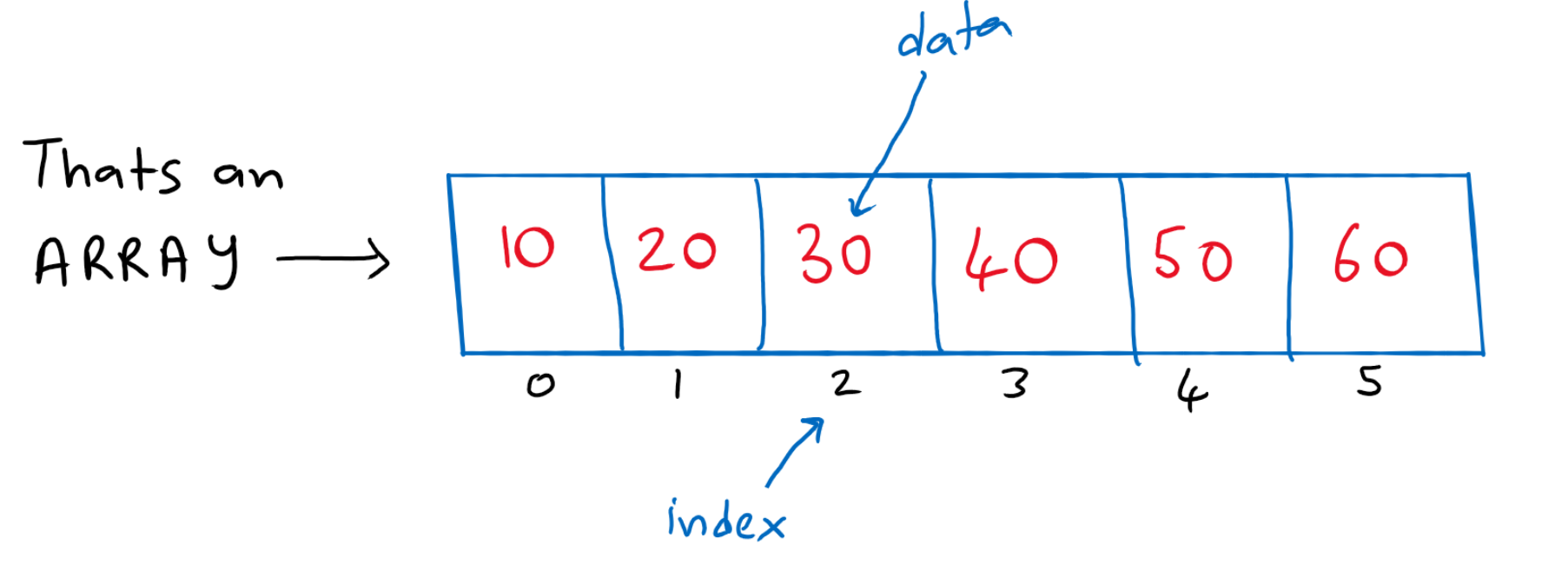
Linked lists have a problem. Suppose you want to read the last

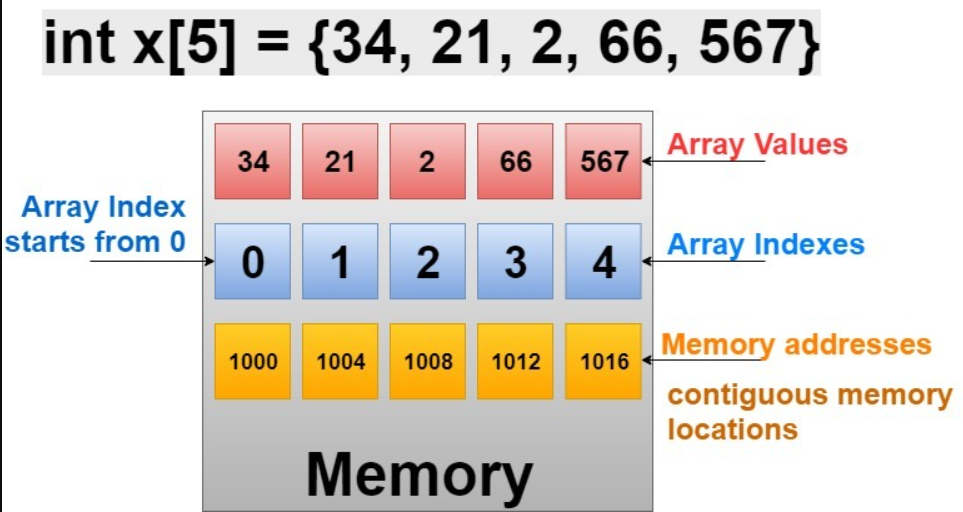
item in a linked list. You can’t just read it, because you don’t know what address it’s at. Instead, you have to go to item #1 to get the address for item #2. then you have to go to item #2 to get the address for item #3. And so on, until you get to the last item. Linked lists are great if you’re going to read all the items one at a time: you can read one item, follow the address to the next item, and so on. But if you’re going to keep jumping around, linked lists are terrible.

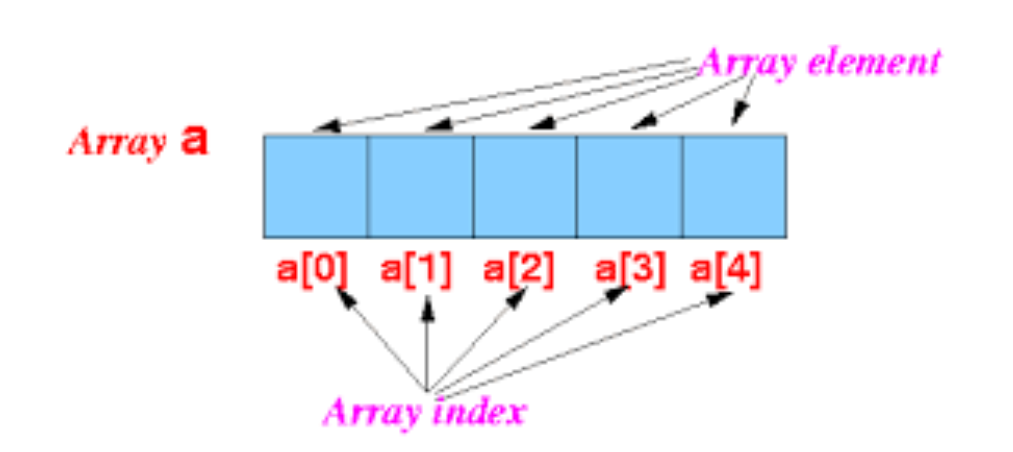
Arrays are different. You know the address for every item in your array.

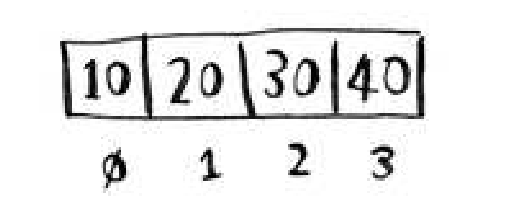
For example, suppose your array contains five items, and you know it

starts at address 00. What is the address of item #5?



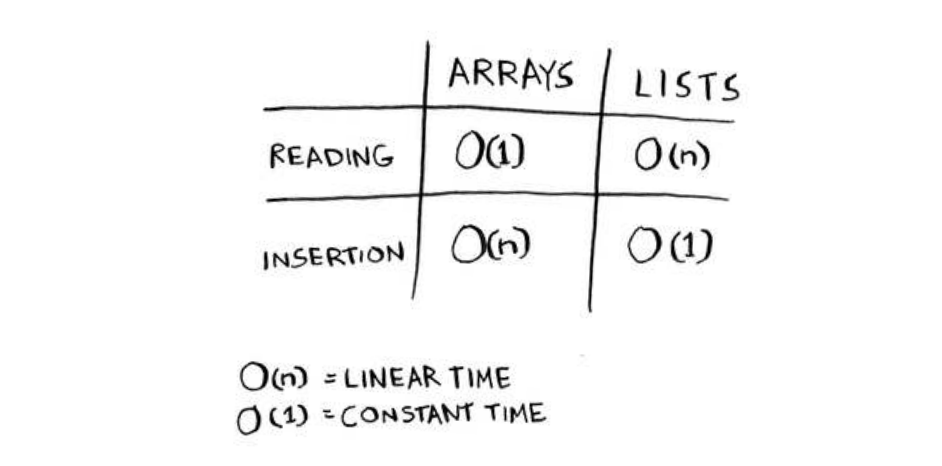






The position of an element is called its index. So instead of saying, “20 is at position 1,” the correct terminology is, “20 is at index 1.” I’ll use index to mean position throughout this book.

Here are the run times for common operations on arrays and lists.



**EXERCISE**

**2.1** Suppose you’re building an app to keep track of your inances.

Every day, you write down everything you spent money on. At the

end of the month, you review your expenses and sum up how much

you spent. So, you have lots of inserts and a few reads. Should you

use an array or a list?

**Ans:** In this case, you’re adding expenses to the list every day and reading all the expenses once a month. **Arrays have fast reads and slow inserts.** Linked lists have **slow reads and fast inserts**. Because you’ll be inserting more often than reading, it makes sense to use a **linked list.** Also, linked lists have slow reads only if you’re accessing random elements in the list. Because you’re reading every element in the list, linked lists will do well on reads too. So a linked list is a good solution to this problem.

**Inserting into the middle of a list**

Lists are better if you want to insert elements into the middle.

**Deletions**

What if you want to delete an element? Again, lists are better, because

you just need to change what the previous element points to. With

arrays, everything needs to be moved up when you delete an element.

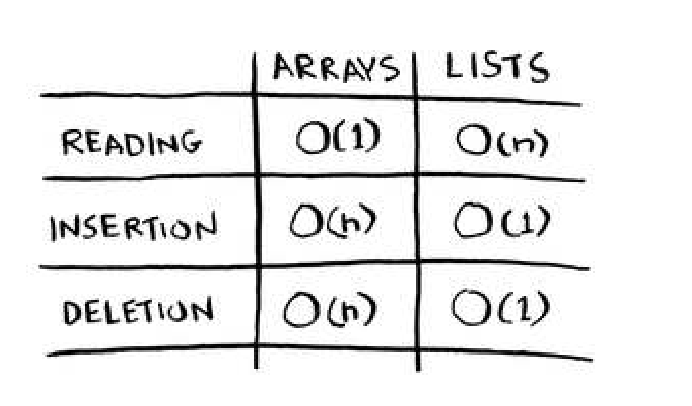
Unlike insertions, deletions will always work. Insertions can fail

sometimes when there’s no space left in memory. But you can always

delete an element.

Here are the run times for common operations on arrays and

linked lists.



It’s worth mentioning that insertions and deletions are O(1) time only

if you can instantly access the element to be deleted. It’s a common

practice to keep track of the first and last items in a linked list, so it

would take only O(1) time to delete those.

Which are used more: arrays or lists? Obviously, it depends on the use

case. But arrays seem a lot of use because they allow random access. here

are two different types of access: **random access and sequential access.**

Sequential access means reading the elements one by one, starting

at the first element. **Linked lists can only do sequential access**. If you

want to read the 10th element of a linked list, you have to read the first

9 elements and follow the links to the 10th element. Random access

means you can jump directly to the 10th element. You’ll frequently

hear me say that arrays are faster at reading. This is because they provide random access. A lot of use cases require random access, so arrays are used a lot. Arrays and lists are used to implement other data structures, too.

**EXERCISES**

**2.2** Suppose you’re building an app for restaurants to take customer

orders. Your app needs to store a list of orders. Servers keep adding

orders to this list, and chefs take orders of the list and make them.

It’s an order queue: servers add orders to the back of the queue, and

the chef takes the irst order of the queue and cooks it.

Would you use an array or a linked list to implement this queue?

(Hint: Linked lists are good for inserts/deletes, and arrays are good

for random access. Which one are you going to be doing here?)

**Ans:** Linked List Based Queue will be the best option for the above asked Question. In the linked list Queue you have the advantage of O(1) operation for insert(placing the orders by servers) and delete (Taking the orders by chef). Also you have dynamic memory allocation (Memory is used up only when it needs to ).

**2.3** Let’s run a thought experiment. Suppose Facebook keeps a list of

usernames. When someone tries to log in to Facebook, a search is

done for their username. If their name is in the list of usernames,

they can log in. People log in to Facebook pretty oten, so there are

a lot of searches through this list of usernames. Suppose Facebook

uses binary search to search the list. Binary search needs random

access—you need to be able to get to the middle of the list of

usernames instantly. Knowing this, would you implement the list

as an array or a linked list?

**Ans:** A sorted array. Arrays give you random access—you can

get an element from the middle of the array instantly. You can’t

do that with linked lists. To get to the middle element in a linked

list, you’d have to start at the first element and follow all the links

down to the middle element.

**2.4** People sign up for Facebook pretty oten, too. Suppose you decided to use an array to store the list of users. What are the downsides of an array for inserts? In particular, suppose you’re using binary search to search for logins. What happens when you add new users to an array?

**Ans:** Inserting into arrays is slow. Also, if you’re using binary

search to search for usernames, the array needs to be sorted.

Suppose someone named Adit B signs up for Facebook. Their

name will be inserted at the end of the array. So you need to sort

the array every time a name is inserted!

**2.5** In reality, Facebook uses neither an array nor a linked list to store

user information. Let’s consider a hybrid data structure: an array

of linked lists. You have an array with 26 slots. Each slot points to a

linked list. For example, the first slot in the array points to a linked

list containing all the usernames starting with a. The second slot

points to a linked list containing all the usernames starting with b,

and so on.

Suppose Adit B signs up for Facebook, and you want to add them

to the list. You go to slot 1 in the array, go to the linked list for slot

1, and add Adit B at the end. Now, suppose you want to search for

Zakhir H. You go to slot 26, which points to a linked list of all the

Z names. hen you search through that list to ind Zakhir H.

Compare this hybrid data structure to arrays and linked lists. Is it

slower or faster than each for searching and inserting? You don’t

have to give Big O run times, just whether the new data structure

would be faster or slower.

**Ans:** Searching—slower than arrays, faster than linked lists.

Inserting—faster than arrays, same amount of time as linked lists.

So it’s slower for searching than an array, but faster or the same

as linked lists for everything. We’ll talk about another hybrid

data structure called a hash table later in the book. This should

give you an idea of how you can build up more complex data

structures from simple ones.

So what does Facebook really use? It probably uses a dozen

different databases, with different data structures behind them:

hash tables, B-trees, and others. Arrays and linked lists are the

building blocks for these more complex data structures.

**Sorting algorithms** are very useful. Now you can sort

• Names in a phone book

• Travel dates

• Emails (newest to oldest)

Selection sort is a neat algorithm, but it’s not very fast. Quicksort is a

a faster sorting algorithm that only takes O(n log n) time.

**Let’s write a function to ind the smallest element**

**in an array:**

*// Selection Sort - O(n^2)*

*function* *findSmallest*(arr) {

*let* smallest *=* arr*[0]*; *// Stores the smallest value*

*let* smallestIndex *= 0*; *// Stores the index of the smallest value*

*for* (*let* i *= 0*; i *<* *arr*.length; i*++*) {

*if* (arr[i] *<* smallest) {

            smallest *=* arr[i]

            smallestIndex *=* i;

        }

    }

*return* smallestIndex;

}

*const selectionSort = (*arr*) => { // Sorts an array*

*let* newArray *= []*

*let* arrLength *= arr.length*

*for (*i *= 0;* i *<* arrLength*;* i*++) {*

*let* smallestIndex *= findSmallest(*arr*); // find the smallest  Element  in the  arr adds to the newArray .*

*newArray.push(arr.splice(*smallestIndex*, 1)[0]);*

*}*

*return* newArray*;*

*}*

myArray *=* [2, 4, 5, 6, 7, 11, 15, 14, 16]

*console*.*log*(*selectionSort*(myArray))