Agenda (3/13)

- Dynamic data structures
  - Linked Lists

  - or -

"recursive" data structures.

database of Students.

- don't know how big the database will be
- the set of students in the database keeps changing
  - add new students
  - remove Students.

can we use an array?

Student db[100] → what if I have more than 100 students

what about a dynamically sized array?

Student * db = malloc(sizeof(Student) * num_students)

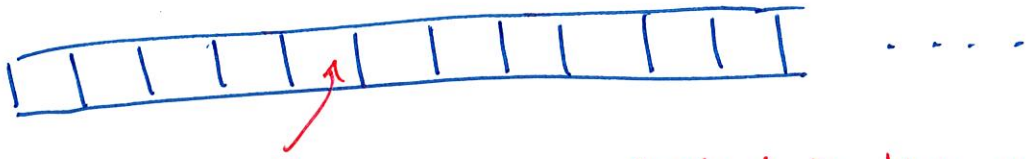
→ If the number of students changes, would have to reallocate (# students 55K → 60K) + copy over the data.

some times I'm ok with this.

"vector libraries" do this. When reallocating an array to make it bigger, <u>double</u> the size. copy over the old data

can <u>maybe</u> get away with not knowing the size. But there's still a problem.

remove a student from the db:



if I remove a student in the middle of the array, it leaves a hole.

add a student to the db:

2 options: 1) add to the end of the array
2) look for a hole in the array and fill the hole.

→ this option leaves a bunch of gaps in the db, takes up more space (removed students take up space forever)

→ if you fill in gaps wherever you can, there is no way to keep the database organized or sorted. (makes it hard to find data). It's also "hard" to find the gaps in the first place!

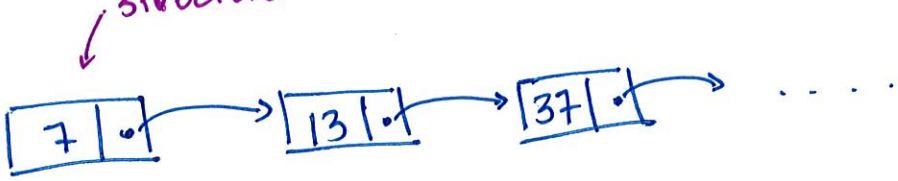
to handle these sorts of situations, we use dynamic data structures.

- growing + shrinking in size / amount of data
- searching for data
- adding/removing certain pieces of data.

Simple dynamic data structure : Linked List

create a linked set of structures, where each structure has two pieces:

1. A piece of data you want to store
2. A link (pointer) to the next structure in the list.



when I think about a struct in C;

```
struct Blah {
    int x;
    float f;
    double y;
    :
}
```

each field has a type.

each one of the building blocks has the same type – it has to be the same kind of structure
→ they all store the same kind of stuff.

what is the type of one of these blocks?
How do I write down a structure definition?

structure definition needs two pieces:

1. field that stores the data
2. pointer to the same kind of structure!

this is weird, but doable.

```
struct Node {
    int val;
    [box] next;
}
```

field that stores the data

pointer that points to the next Node.

struct * Node *

pointer to the struct type we are in the middle of defining!

```
struct Node {
    int val;
    struct Node * next;
}
```

what does the last element in the List point to?

by convention, NULL. (the next pointer equals NULL)

Add ~~an~~ ~~#~~ ~~*~~ a 4 to the linked list.

here is what I want the list to look like:

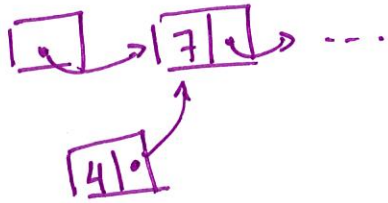
think in terms of rewiring/rerouting pointers

1. create the new Node.

```
struct Node * ~~newNode~~ newNode = malloc(sizeof(struct Node))
newNode -> val = 4;
```

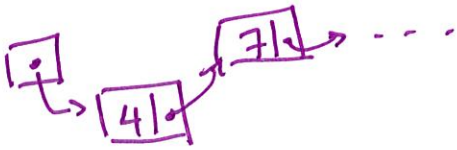
where does 4's next pointer need to point? 7

```
newNode -> next = head; // whatever head is pointing
                           at, so does this next
                           field.
```



now head needs to point where newNode points.

```
~~new~~ head = newNode.
```



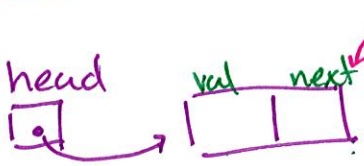
with 4 steps, I added a new number to the beginning of the list!

lifting this into a function is interesting, but tricky

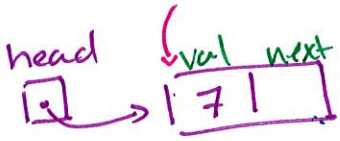
=> every time I do these four steps, it will add a new number to the beginning of the list

Let's build a list of integers:

```
struct Node * head = malloc (sizeof (struct Node));
```

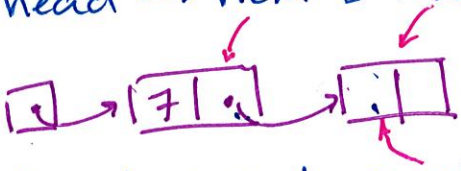



```
head -> val = 7; // remember how -> works
```



to put another number into the list, the Node containing 7 should point to a new Node.

```
head -> next = malloc(sizeof(struct Node))
```



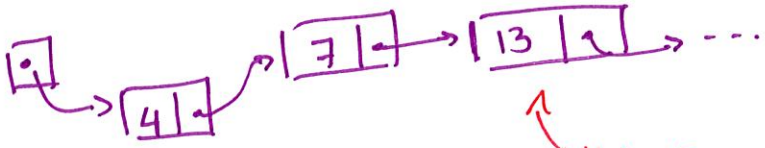

```
head -> next -> val = 13
```


```
head -> next -> next = malloc( sizeof(struct Node));
```


```
head -> next -> next -> val = 27
```


```
head -> next -> next -> next = NULL
```

remove the ~~first~~ second element from the list.



this is pointed at by head → next → next

want:

(7 is removed)

this pointer needs to point here

```
head -> next = head -> next -> next;
```

~~cause~~ causes a memory leak!

fix it by making a temp pointer point at the Node we're removing so we can ~~delete~~ free it later

```
struct Node * toDelete = head -> next;
head -> next = head -> next -> next;
free(toDelete);
```

this points at the node we want to delete

deletes the node

---

to delete the first node in the list:

```
struct Node * toDelete = head;
head = head -> next;
free(toDelete)
```

the pointer in the purple box is the one you want to rewire.