Programming Abstractions

CS106B

Cynthia Lee

Topics:

- **This week: Memory and Pointers**
  - Monday: revisit some topics from last week in more detail:
    - Deeper look at new/delete dynamic memory allocation
    - Deeper look at what a pointer is
  - Today:
    - Finish up the music album example
    - Linked nodes
  - Friday:
    - Linked List data structure
    - (if we have time) priority queues and binary trees

Hat tip to Victoria Kirst of Google for some of today's slides!

Pointers recap

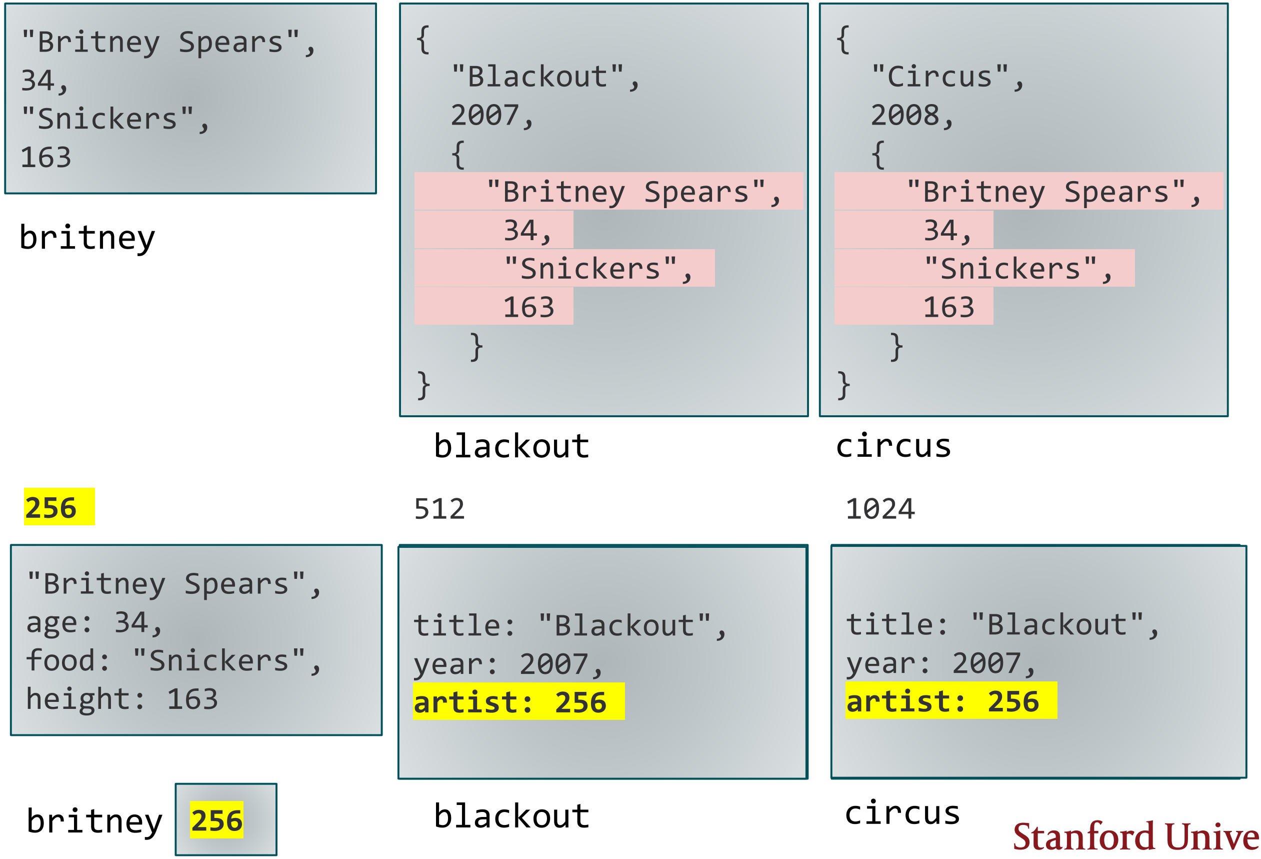
Pointers recap so far (bookmark this slide)

- The first pointer we saw was a **dynamically allocated** array
  - `type* name = new type[length];`
  - `int* data = new int[10];`
- A pointer is a variable that stores a **memory address**
  - A memory address is just a number, like an array index but where the array is the entire memory of the computer
- You can use the **address-of operator &** to ask for the address of any named variable in your program
  - `int x = 3;`
  - `int* ptr = &x;`
- Many common types of variables (like ints) consume 4 **bytes** of memory, so addresses increment by 4 between adjacent variables. Other types (like doubles and pointers) might take 8 or more bytes to store because they are more complex.
  - In general, you don’t really need to worry about this detail for this course, but it’s good to be aware of it.
- **Sharing information** between several objects is a common use case for a pointer
  - Each album object contains a pointer to the artist object, so they can all share the artist information instead of many copies

Next steps with pointers and structs/classes/objects

Pointers replace redudant copies with a “please see,” like a book/paper citation

Redundancy:
☹



```
"Britney Spears",
34,
"Snickers",
163
```

britney

```
{
  "Blackout",
  2007,
  {
    "Britney Spears",
     34,
     "Snickers",
     163
   }
}
```

blackout

```
{
  "Circus",
  2008,
  {
    "Britney Spears",
     34,
     "Snickers",
     163
   }
}
```

circus

Sharing/
efficiency:
☺

256

```
"Britney Spears",
age: 34,
food: "Snickers",
height: 163
```

britney 256

512

```
title: "Blackout",
year: 2007,
artist: 256
```

blackout

1024

```
title: "Blackout",
year: 2007,
artist: 256
```

circus

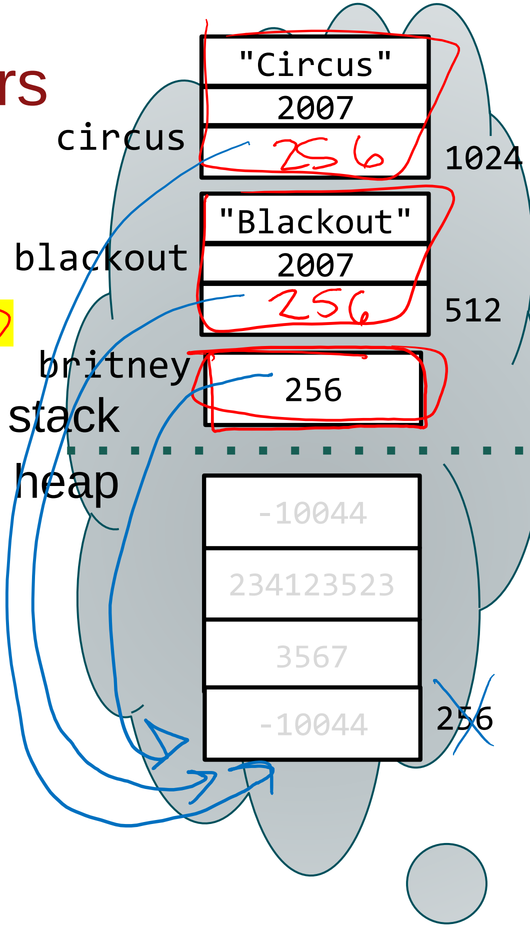
Fixing the Album/Artist example with pointers

```
struct Artist {
  string name;
  int age;
  string favorite_food;
  int height; // in cm
};
```

```
struct Album {
  string title;
  int year;
  Artist* artist;
};
```

```
Artist* britney = new Artist;
// TODO: now we need to set the fields of britney

Album blackout = { "Blackout", 2007, britney };
Album circus = { "Circus", 2008, britney };
```



circus

"Circus"

2007

256

1024

blackout

"Blackout"

2007

256

512

britney

256

stack

heap

-10044

234123523

3567

-10044

~~256~~

Fixing the Album/Artist example with pointers

```
struct Artist {                      struct Album {
  string name;                         string title;
  int age;                             int year;
  string favorite_food;                Artist* artist;
  int height; // in cm               };
};
```

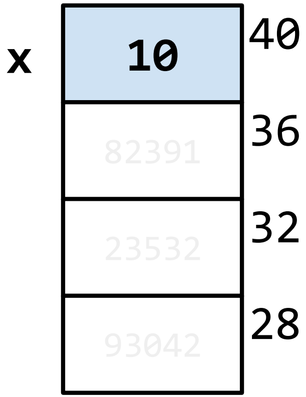
```
Artist* britney = new Artist;
// TODO: now we need to set the fields of britney
britney.name = "Britney Spears"; // no! type is Artist* not Artist
                                 // we need a new tool that says
                                 // “follow the pointer”
Album blackout = { "Blackout", 2007, britney };
Album circus = { "Circus", 2008, britney };
```

"Dereferencing" a pointer

You can follow ("**dereference**") a pointer by writing

******variable_name***

```
int x = 10
int* ptr_to_x = &x;
cout << *ptr_to_x << endl; // 10
```



Fixing the Album/Artist example with pointers

```
struct Artist {                          struct Album {
  string name;                             string title;
  int age;                                 int year;
  string favorite_food;                    Artist* artist;
  int height; // in cm                   };
};
```

```
       ptr_to_
Artist* britney = new Artist;
// TODO: now we need to set the fields of britney
(*britney).name = "Britney Spears"; // this works but really clunky

Album blackout = { "Blackout", 2007, britney };
Album circus = { "Circus", 2008, britney };
```

-> **operator**: Dereferencing and accessing a member

```
struct Artist {                          struct Album {
  string name;                             string title;
  int age;                                 int year;
  string favorite_food;                    Artist* artist;
  int height; // in cm                   };
};
```

```
Artist* britney = new Artist;
// TODO: now we need to set the fields of britney
britney->name = "Britney Spears"; // ptr->member is the exact same as (*ptr).member

Album blackout = { "Blackout", 2007, britney };
Album circus = { "Circus", 2008, britney };
```

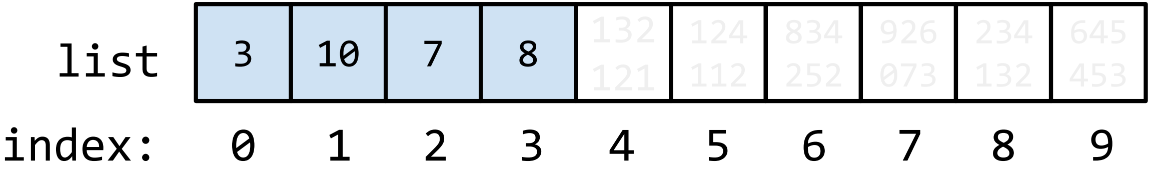
Linked Nodes

Another important application of pointers

We’ll start by looking at a limitation of the array

Arrays

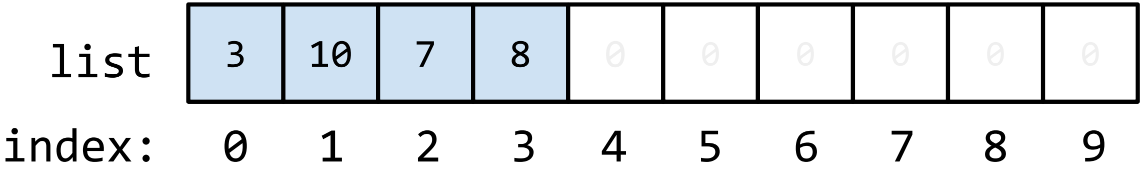
What are arrays good at? What are arrays bad at?



list	3	10	7	8	132 121	124 112	834 252	926 073	234 132	645 453
index:	0	1	2	3	4	5	6	7	8	9

ULYSSES
WAR AND PEACE

Memory is a giant array...



list	3	10	7	8	0	0	0	0	0	0
index:	0	1	2	3	4	5	6	7	8	9

What are the most annoying operations on a tightly packed book shelf, liquor cabinet, shoe closet, etc?

Insertion - **O(n)**

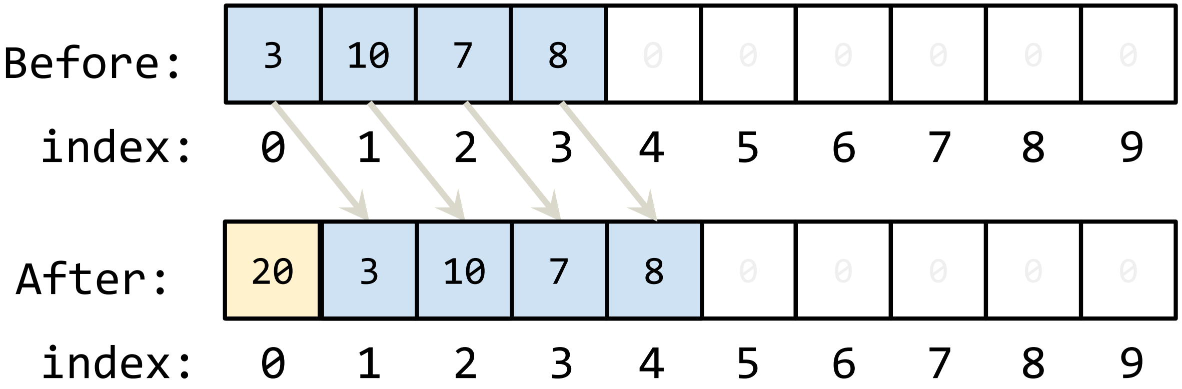
Deletion - **O(n)**

Lookup - **O(1)**

Let's brainstorm ways to improve insertion and deletion....

Add to front

What if we were trying to add an element "20" at index 0?

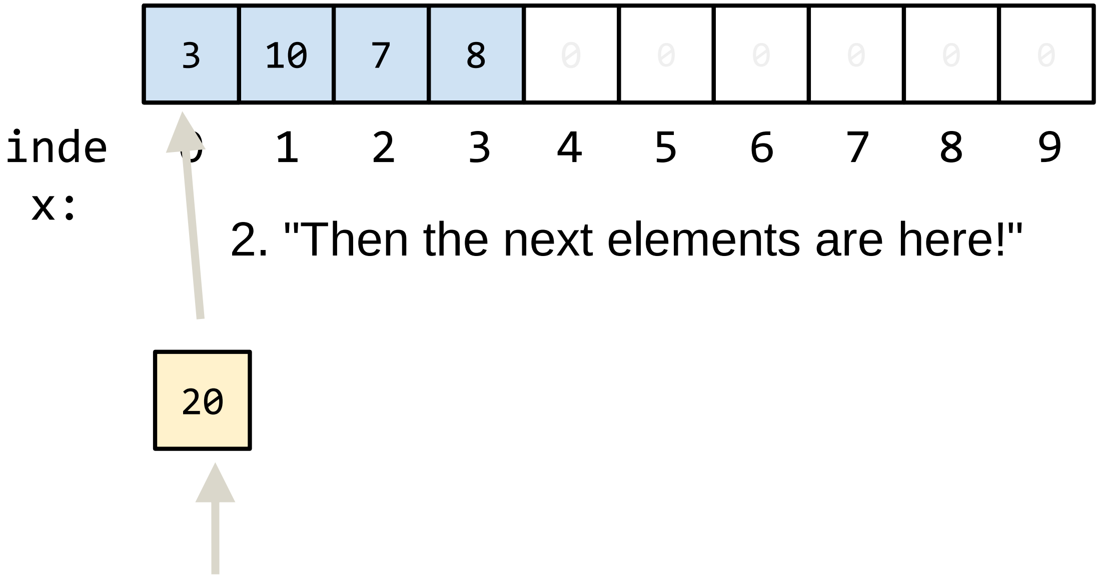


Before:	3	10	7	8	0	0	0	0	0	0
index:	0	1	2	3	4	5	6	7	8	9

After:	20	3	10	7	8	0	0	0	0	0
index:	0	1	2	3	4	5	6	7	8	9

Add to front

Wouldn't it be nice if we could just do something like:



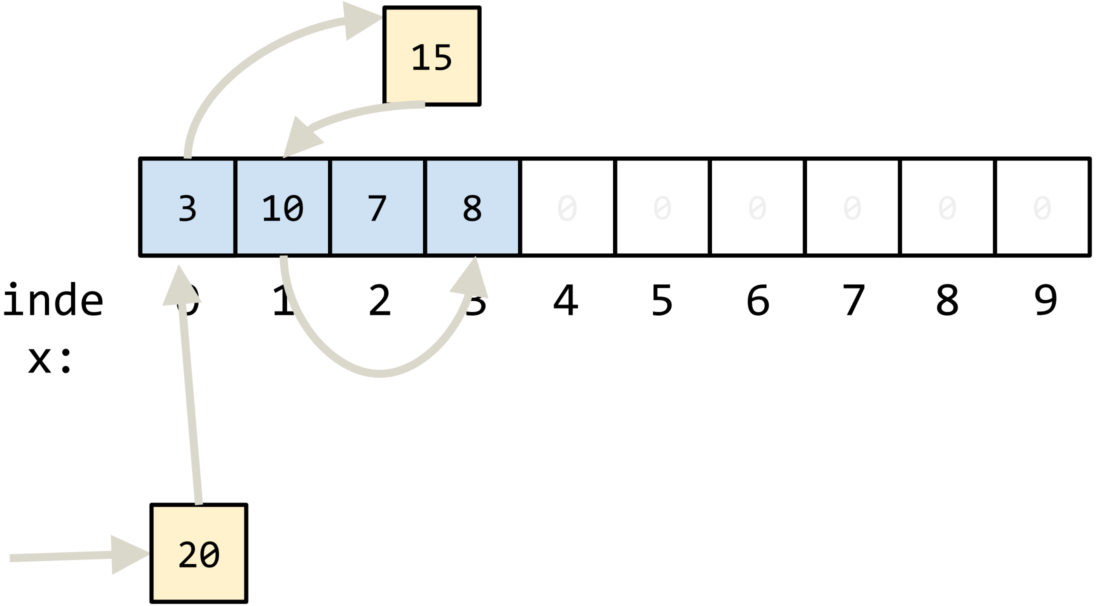
	3	10	7	8	0	0	0	0	0	0
index:	0	1	2	3	4	5	6	7	8	9

2. "Then the next elements are here!"

20

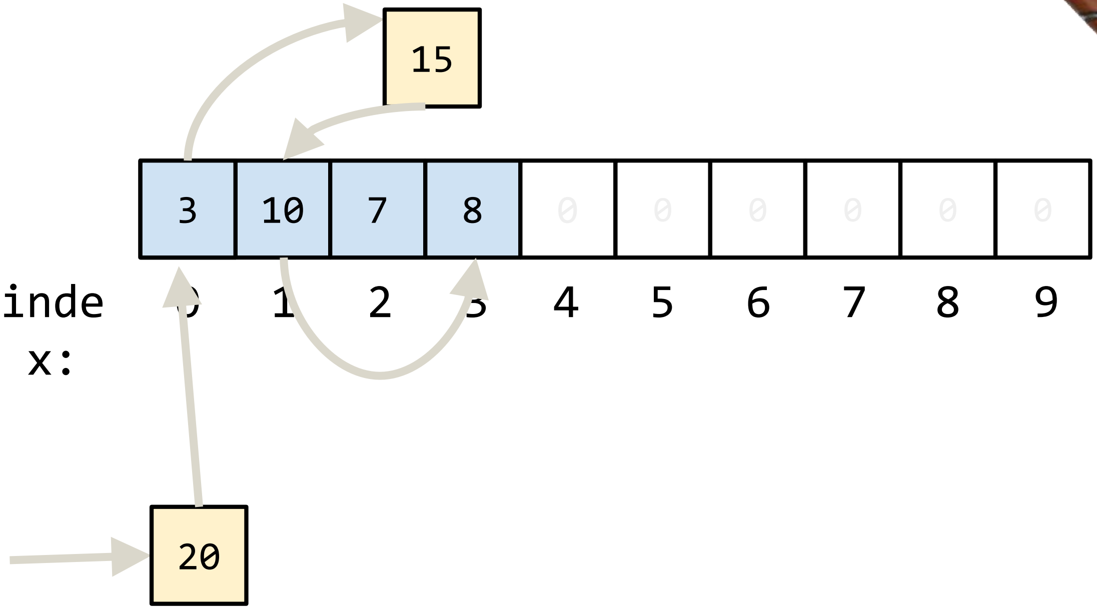
1. "Start here instead!"

Now we add to the front again:
Arrows everywhere!

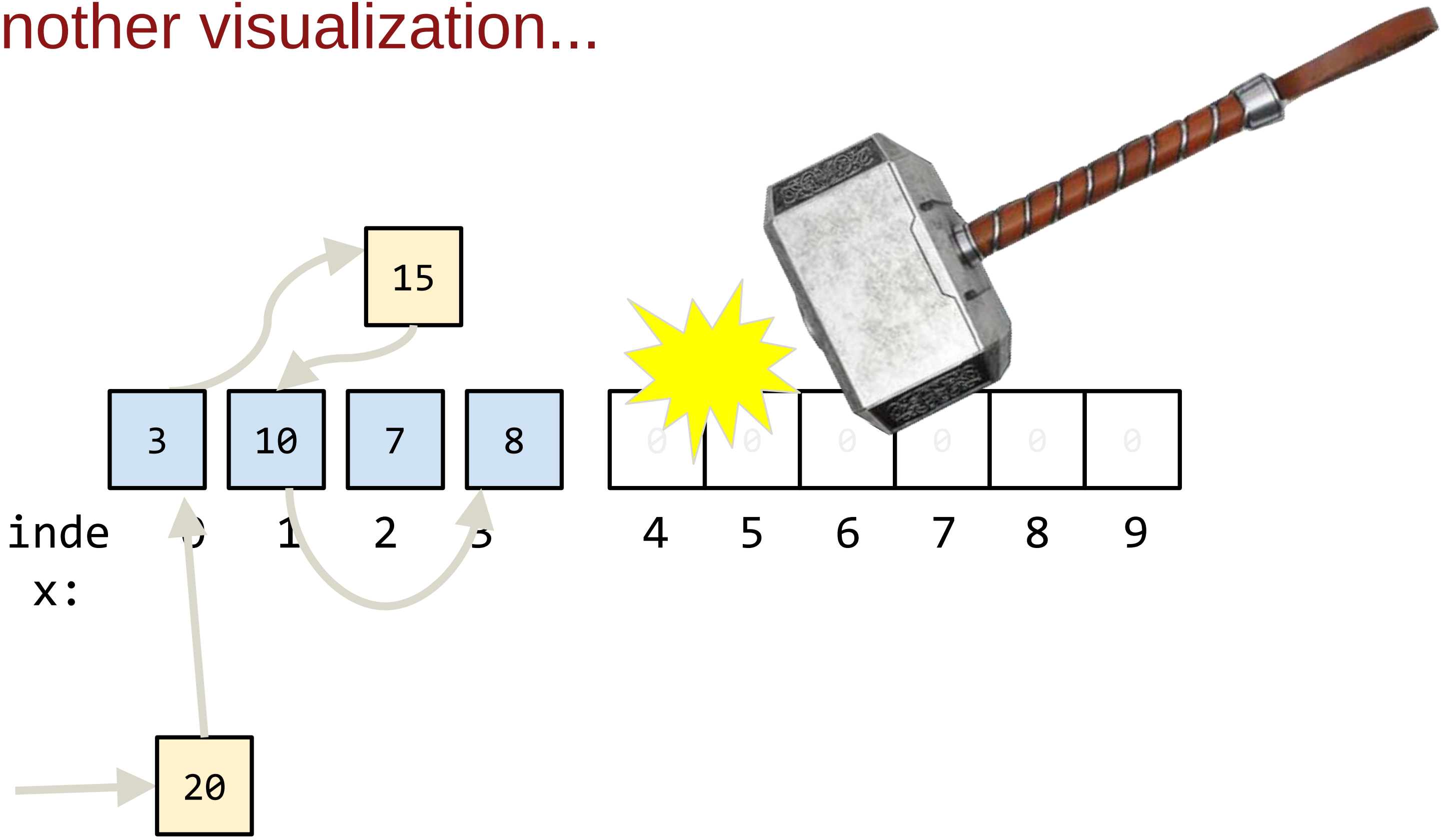


	3	10	7	8	0	0	0	0	0	0
index:	0	1	2	3	4	5	6	7	8	9

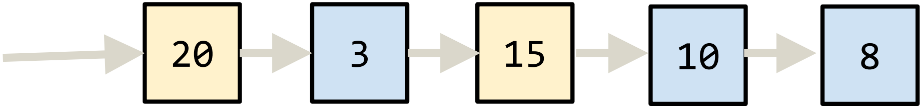
Another visualization...



Another visualization...



This is a **list of linked nodes**!



- A list of linked nodes (or a **linked list**) is composed of interchangeable **nodes**
- Each element is stored separately from the others (vs contiguously in arrays)
- Elements are chained together to form a one-way sequence using pointers

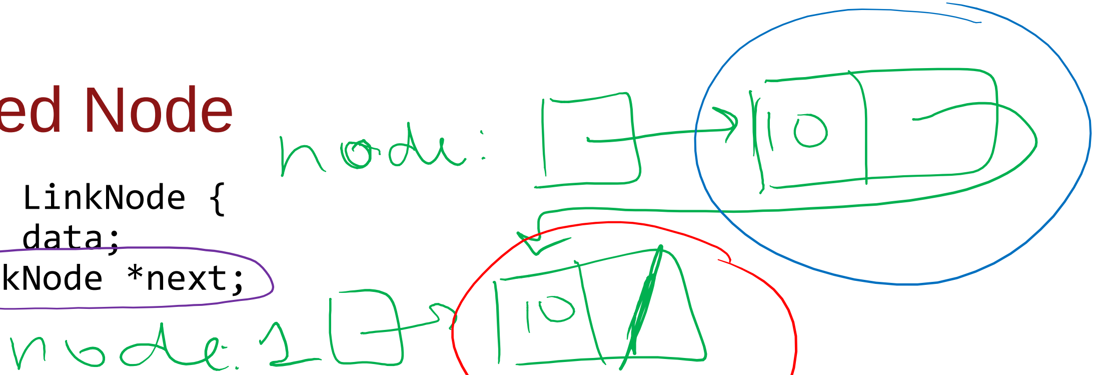
Linked Nodes

A great way to exercise your pointer understanding

Stanford University

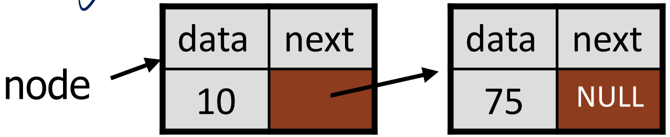
Linked Node

```
struct LinkNode {
    int data;
    LinkNode *next;
}
```



- We can chain these together in memory:

unchanged



```
LinkNode *node1 = new LinkNode;        // complete the code to make picture
node1->data = 10;
node1->next = NULL;
LinkNode *node = new LinkNode;
node->data = 10;
node->next = node1;
```

#ec

node1->data=75;

node->next->data= 75;

FIRST RULE OF LINKED NODE/LISTS CLUB:

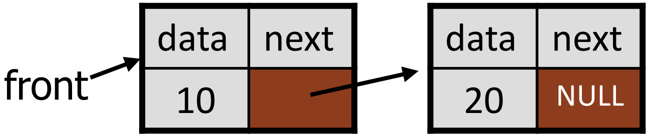
DRAW A PICTURE OF LINKED LISTS

Do no attempt to code linked nodes/lists without pictures!

List code example: Draw a picture!

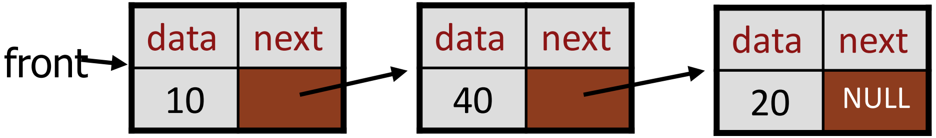
```
struct LinkNode {
  int data;
  LinkNode *next;
}
```

Before:

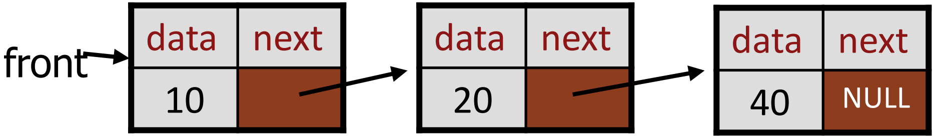


```
front->next->next = new LinkNode;
front->next->next->data = 40;
```

A. After:



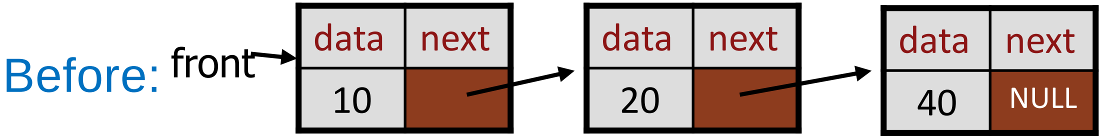
B. After:



C. Using “next” that is NULL gives error

D. Other/none/more than one

List code example: Draw a picture!



```
struct LinkNode {
   int data;
   LinkNode *next;
}
```

Write code that will put these in the reverse order.