# Week 5, continued

This is CS50. Harvard University. Fall 2014.

Cheng Gong

## Table of Contents



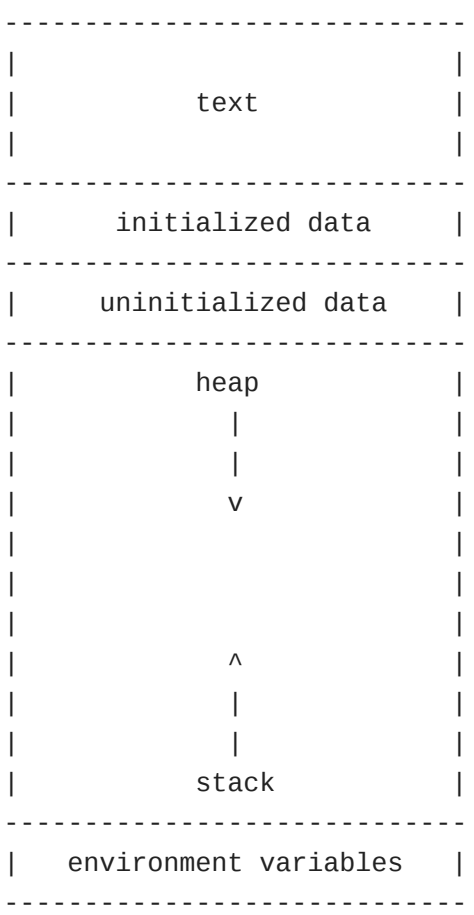
# News

- Good news everyone! CS50 Lunch is again this Friday, RSVP at the usual http://cs50.harvard.edu/rsvp.
- Even better news, no lecture on Monday, 10/13, for Columbus Day!
- Less better news, Quiz 0[1] is Wednesday, 10/15, more details to come.
- Slightly better news, review session on Monday 10/13, and sections will meet as usual.
- Best news, there will be lecture on Friday 10/17, with constant time data structures, trees, tries, hash tables, and all sorts of fun stuff.

# Buffer Overflow

- Remember that we've been looking into a computer's memory, or where a program lives when it's running. When you click on a program to open it, it's copied from the hard drive (or SSD nowadays) to RAM, random-access memory, where it lives until the program quits, the power goes out, or you shut down your laptop.
- That memory is laid out like this, for each program:

[1] http://cs50.harvard.edu/quizzes/0




# This is a conceptual model where we have the **stack** at the bottom and other things at the top.

# The thing at the very top, the **text** segment, is the actual ones and zeroes that make up your compiled program. When you run `./mario`, those bits are copied to the RAM in the text segment.

# Below that, **initialized** and **uninitialized data** are things like global variables, that we've not used many of, or statically defined strings, that are hardcoded into your program.

# Then we have the **stack**, where we've used it for functions by copying over variables, that are passed in as arguments, to the stack. Local variables within each function are also stored in the stack.

# Finally, the **environment variables**, which we've seen when we did something like `x[1000]` when we shouldn't have, are the global settings for the program.

- Today, we'll focus on the **heap**, another chunk of memory (and all of these bytes are stored on the same hardware, only we designate them for different purposes), where other variables and allocated memory from the operating system are stored. You can see that there's a problem if the heap and the stack collide if your program starts to use too much memory, and bad things will happen.
- Last time we looked at:

```
#include <string.h>

void f(char* bar)
{
    char c[12];
    strncpy(c, bar, strlen(bar));
}

int main(int argc, char* argv[])
{
    f(argv[1]);
}
```

  # It has no functional purpose, other than to demonstrate how a poorly written program might lead to your whole computer being taken over. Notice `main` takes `argv[1]` and passes it in to `f`, so whatever word the user types in after the name of the program, and then `f` takes it, which we've called `bar`, and copies it into `c`, which can hold 12 characters. `c` is a local variable of 12 `char`s so it will live on the stack. `strncpy` copies a string, but only `n` letters, in this case `strlen(bar)`, or the length of the user-inputted string.
  # The problem is that we're not checking for the length of `bar`, so if it were 20 characters long, it would overflow and take up 8 more bytes than it should.

- The implication is this diagram of a zoomed-in version of the bottom of the program's stack:

```
|  unallocated stack space  |
-----------------------------
| c[0] |                    |
| ---- |                    |
|                           |
|         char c[12]        |
|                           |
```

```
|                  | ----- |
|                  | c[11] |
-----------------------------
|         char* bar         |
-----------------------------
|                           |
-----------------------------
|       return address      |
-----------------------------
|   parent routine's stack  |
```

# At the very bottom is the **parent routine's stack**, in this case `main`, or whichever function that called this one.

# **return address** has always been there, which was copied over along with local variables when the function was called, and this is just where in memory the program should jump back to, once the function returns. In this case, it's somewhere in `main`.

# The top is the stack frame for the function. There's `bar`, an argument to the function, and `c`, an array of characters. And to be clear, on the top left would be `c[0]`, the first character, with the last, `c[11]`, on the bottom right corner.

- What happens if we pass in a `string` with `char* bar` longer than `c`? You would overwrite `char* bar` and, even worse, the return address. Then the program would go back to, well, anywhere in memory that `bar` specified. When bad guys are curious if a program is buggy or exploitable, they send lots of inputs of different lengths, and when it causes your program to crash, then they have discovered a bug.
- In particular, the best case might look something like this:

```
|  unallocated stack space  |
-----------------------------
|  h   |  e   |  l   |  l   |
| ---- | ---- | ---- | ---- |
|  o   |  \0  |             |
| ---- | ---- |             |
|         char c[12]        |
|                           |
|                           |
-----------------------------
|         char* bar         |
```

```
-----------------------------
|                           |
-----------------------------
|       return address      |
-----------------------------
|   parent routine's stack  |
```

# The string passed in is just `hello`, which fits in `c`.

- But what about "attack code" that looks like this?

```
                |  unallocated stack space  |
  Address       -----------------------------
  0x80C03508 -> |  A   |  A   |  A   |  A   |
                | ---- | ---- | ---- | ---- |
                |  A   |  A   |  A   |  A   |
                | ---- | ---- | ---- | ---- |     char c[]
                |  A   |  A   |  A   |  A   |
                -----------------------------
                |  A   |  A   |  A   |  A   |     char* bar
                -----------------------------
                |  A   |  A   |  A   |  A   |
                -----------------------------
                | 0x08 | 0x35 | 0xC0 | 0x80 |    return address
                -----------------------------
                |   parent routine's stack  |
```

# In this picture, the `A`s are arbitrary zeroes and ones that can do anything, maybe `rm -rf` or send spam, and if that person includes those, but also has the last 4 bytes be the precise address of the first character of the `string`, you can trick a computer into going back to the beginning of the `string` and executing the code, instead of simply storing it. The return address, among other things, has been overwritten, and generally it's "shellcode" that gives a bad guy control over a computer, through Bash or some other shell. (And if you've noticed, the return address is the address of the first `A`, with the order of the bytes reversed. This is called little-endianness[2] — an advanced topic we won't need to worry much about yet!)

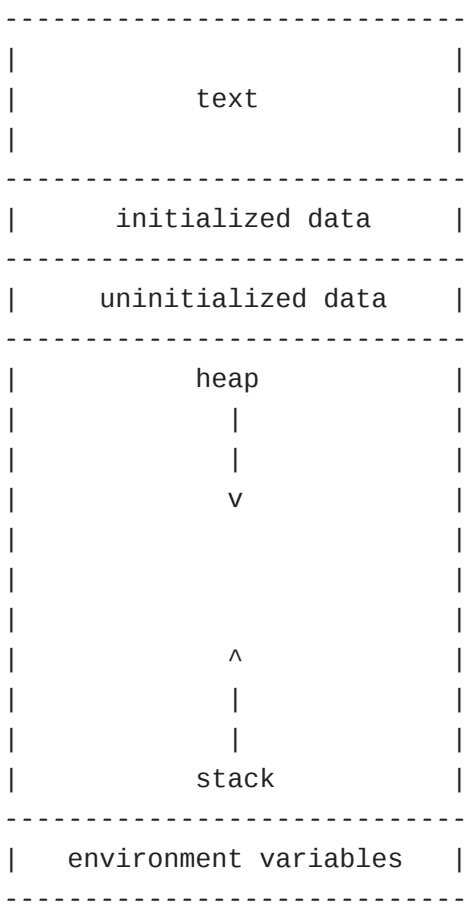
- So in short, this bug came from not checking the boundaries of your array, and since the computer uses the stack from bottom up (think trays from Annenberg being stacked

[2] http://en.wikipedia.org/wiki/Endianness

on a table), while arrays you push on the stack are written to from the top down, this is even more of a concern. And though there are ways and entire languages around this problem (Java is immune to this since they don't give you pointers or direct access to memory addresses), the power of C comes with great responsibility and risk. If you even read articles about "buffer-overflow attacks," then they're probably talking about this.

## Malloc

- `malloc` is our friend since we can allocate memory if or when we want it, so we don't have to hardcode it.
- All of `malloc`'s memory comes from the heap, if we refer back to this general diagram:



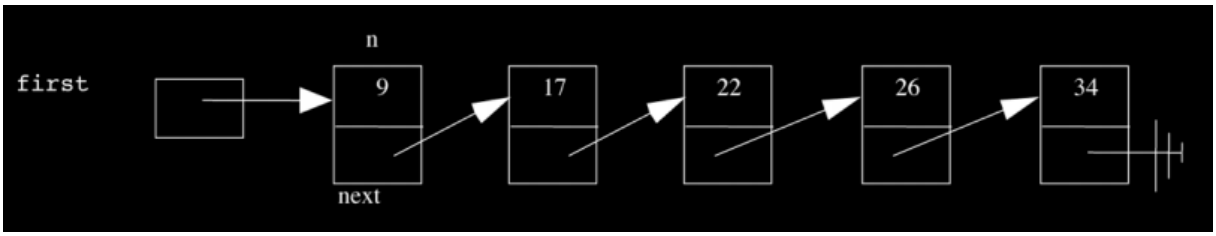

- This is how `GetString` can allocate memory without knowing how much you'll type beforehand, return a pointer to you to that memory, and allow you to keep that memory after `GetString` returns. Remember that the stack goes up and down as functions are called and as they finish, in the bottom half of the diagram. And `malloc` allocates

from the top, so the data stored there isn't impacted by the movements of the stack, so you can use the variables after the function returns.

- The opposite of `malloc` is `free`, and a good rule to start thinking about is that you should be using `free` every time you're done with something you used `malloc` to get.
- All this time we've been writing buggy code, by using the CS50 Library to `GetString`, which leaks memory. We've been asking the operating system for memory, and never returning it. `valgrind` will help us find these leaks!
- `malloc` will also help us solve problems much more effectively. So far, the fanciest data structure (a way of clustering things together beyond just a single `int` or `char`), we've had is the array, which is continuous chunks of memory, each of which is the same type.
  - # But there are downsides to arrays. They have limited size, and once an array is filled, we might try to put another item at the end, but the next chunk of memory following it might be used by another variable. (Think to last time when we had `Zamyla` and `Daven` and `Gabe` in memory as `string`s). We might make another array of one size bigger, and copy over all the elements from the old array to the new array, but that's inefficient. So we have the benefit of random access, but not dynamic resize. We can implement binary search and other algorithms that divide and conquer, but we can't change the size easily once we fill up the array.

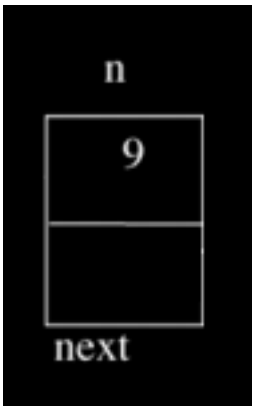
## Linked Lists

- Another data structure is a linked list. Instead of rectangles back-to-back, we have rectangles with a bit of space:




  - # The boxes look orderly in the image, but in reality they might be all over the place, with arrows that link each rectangle to the next.

- We've used pointers to represent an arrow, so instead of an array that only stores numbers, we can store a pointer next to each number that weaves all of these rectangles together.
- If we wanted to implement this, we'd start by noticing that each of these rectangles aren't a single number, but rather an `int` (though they can have anything) and a `pointer`:




- To create our own data structure, we just have to define a `struct` like we've seen before:

```
typedef struct
{
    string name;
    string house;
}
student;
```

- Now we can take that idea and do something like the following:

```
typedef struct node
{
    int n;
    struct node* next;
}
node;
```

# A `node` is a general computer science term for an element in a data structure.

- Our `node` will have the `int` and also a `struct node*`, or pointer to another node.

- `typedef struct node` is also at the top, for `node` to be able to refer to itself or another `node`, or self-referential. Notice how we didn't need that for `student` since they don't need to refer to another student.
- Let's take a look at `./list-0`[3]:

```
jharvard@appliance (~): ./list-0

MENU

1 - delete
2 - insert
3 - search
4 - traverse
0 - quit
```

# Notice the interface with a menu that allows the user to input an option.

- Let's recreate the list from the diagram above:

```
jharvard@appliance (~): ./list-0

MENU

1 - delete
2 - insert
3 - search
4 - traverse
0 - quit

Command: 2
Number to insert: 9

LIST IS NOW: 9


MENU

1 - delete
2 - insert
3 - search
```

[3] http://cdn.cs50.net/2014/fall/lectures/5/w/src5w/

```
4 - traverse
0 - quit

Command: 2
Number to insert: 17

LIST IS NOW: 9 17
```

- So we inserted `9` and `17`, but now let's try to insert `26` before `22`:

```
MENU

1 - delete
2 - insert
3 - search
4 - traverse
0 - quit

Command: 2
Number to insert: 26

LIST IS NOW: 9 17 26


MENU

1 - delete
2 - insert
3 - search
4 - traverse
0 - quit

Command: 2
Number to insert: 22

LIST IS NOW: 9 17 22 26
```

# Both were inserted, and the list was still sorted after, so everything seems to work.

- Now let's finish creating the list by inserting `34`, and just to be sure, we can search for `22`:

```
MENU
```

```
1 - delete
2 - insert
3 - search
4 - traverse
0 - quit

Command: 2
Number to insert: 34

LIST IS NOW: 9 17 22 26 34


MENU

1 - delete
2 - insert
3 - search
4 - traverse
0 - quit

Command: 3
Number to search for: 22

Found 22!
```

- To implement this, `list-0.h` might include something like this:

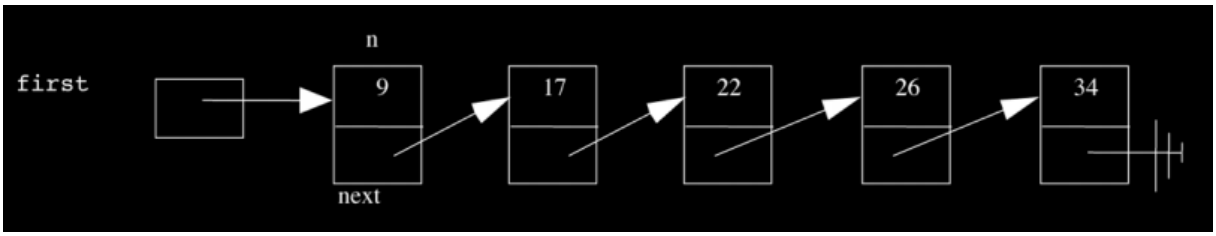
```
typedef struct node
{
    int n;
    struct node* next;
}
node;
```

- And `list-0.c` might include lines like these:

```c
#include "list-0.h"

void search(int n);

int main(void)
{
    // TODO
}

void search(int n)
{

}
```

# `search` would be one of the functions that finds an element in the list, if it exists.

- We also need to make the list itself, so let's we refer back to that diagram:




- So it seems like we only need to remember a pointer to the first node, and the rest will be attached as they are created. So we can write:

```
#include "list-0.h"

// declare linked list up here
node* first = NULL;

void search(int n);

int main(void)
{
    // TODO
}

void search(int n)
{

}
```

`#` and set `first = NULL` because there is no pointer there yet.

- But if we only remember the first node, then we have to make sure the end with a `NULL` is properly implemented, much like a `string` needs a terminating character.
- Since there are no elements in the list, then having the pointer to the `first` node be `NULL` is all we need.
- How long might it take to reach an element in this list? #(*n*), which isn't bad, but is linear. We've given up the random access that arrays allow, since `malloc` gives us memory from wherever it is available in the heap, and the addresses of each node could be spaced far apart.

## Searching

- So if we wanted to implement a `search` function, we start with this:

```
#include "list-0.h"

// declare linked list up here
node* first = NULL;

void search(int n);

int main(void)
{
    // TODO
}

void search(int n)
{
    node* ptr = first;
}
```

# The variable is named `ptr` by convention, and will keep track of where we are in the list.

- And add a bit more:

```
#include "list-0.h"

// declare linked list up here
node* first = NULL;

void search(int n);

int main(void)
{
    // TODO
}

void search(int n)
{
    node* ptr = first;
    while (ptr != NULL) ❶
    {
        if (ptr->n == n) ❷
        {
            // announce that we found n
            break;
        }
        else
        {
            ptr = ptr->next;
        }
    }
}
```

# Lots of things to notice here. Line 16 creates a loop that we will continue to run until we get to `NULL`, or the end of the list.

# Then in line 18, we check if the number stored at `ptr`, `ptr->n`, is what we're looking for. The `->` is syntax for going to the location from pointer `ptr` and retrieving the variable `n` stored within that struct, since `ptr` isn't a `node` itself but rather a `node*`. (You could also use `(*ptr).n`, meaning go to `ptr` and then get the `n` field, since `*ptr` is a struct. And `ptr->n` is exactly the same, just "syntactic sugar," or something that makes the code look better.)

    # If the number isn't what we're looking for, then we set `ptr` to `ptr->next`, which is moving our placeholder to the next `node` in the linked list.

- If we look through the actual implementation of `list-0.c`, the code is not fun to walk through, but the concepts are quite intuitive.
- Let's think about this with help from volunteers from the audience.
- We line up people to represent each rectangle, with Gabe on the far left to represent `first`, which is just a pointer:

```
[]----->[9]--->[17]--->[22]--->[29]--->[34]
                                        |
                                        V
```

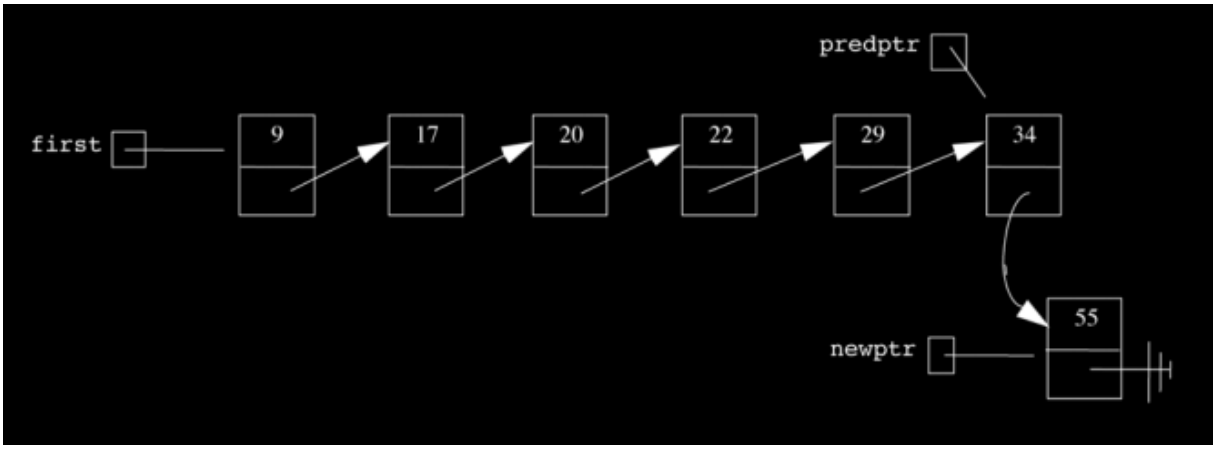
    # And we have everyone pointing to either the next `node`, or in the case of `34`, pointing downward to represent `NULL`.

## Inserting

- Now let's try to insert the element `55`, held by David. Let's consider the different cases, with 3 possible situations: `55` might go at the beginning, in the middle, or at the end.
- Remember that for the `search` function, we start by initializing a `ptr` that points not at `first`, but `9`, since that's what `first` points to. Then it moves down the list following the arrows until it finds the element or reaches the end.
- Our `insert` function will do the same, but comparing if `55` is less than each element, since we want to keep the list sorted. So we get to the end, and the pointer in the node of `55` will be `NULL` and the pointer of `34` will change to point to the node containing `55`:

```
[]----->[9]--->[17]--->[22]--->[29]--->[34]--->[55]
                                                |
                                                V
```

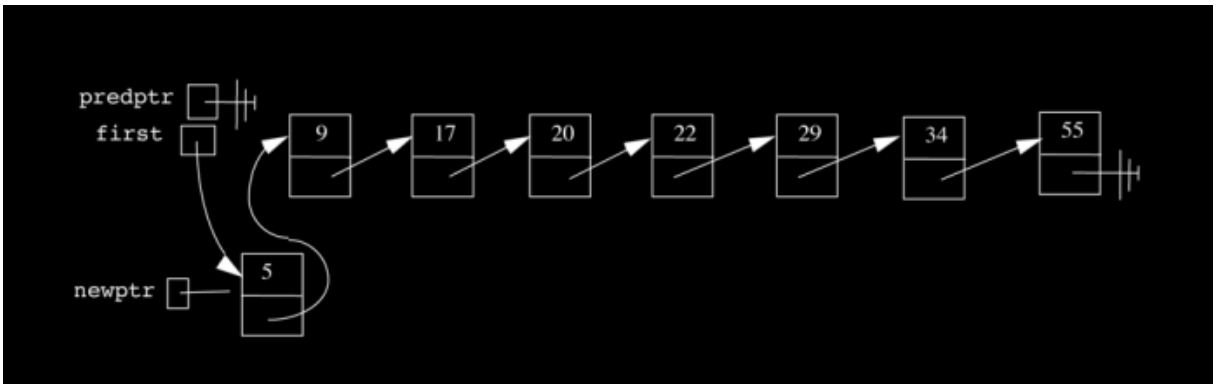
- And this is image is another way to look at it:




- Now let's say we have to insert to the beginning of the list, a number like `5`. We start by intializing our `ptr` to the point to the first element, `9`, and realize that `5` is less than `9`. So now Gabe, `first`, needs to point to the node of `5` and the node of `5` will now point at `9`:

```
[]----->[5]--->[9]--->[17]--->[22]--->[29]--->[34]--->[55]
                                                       |
                                                       V
```

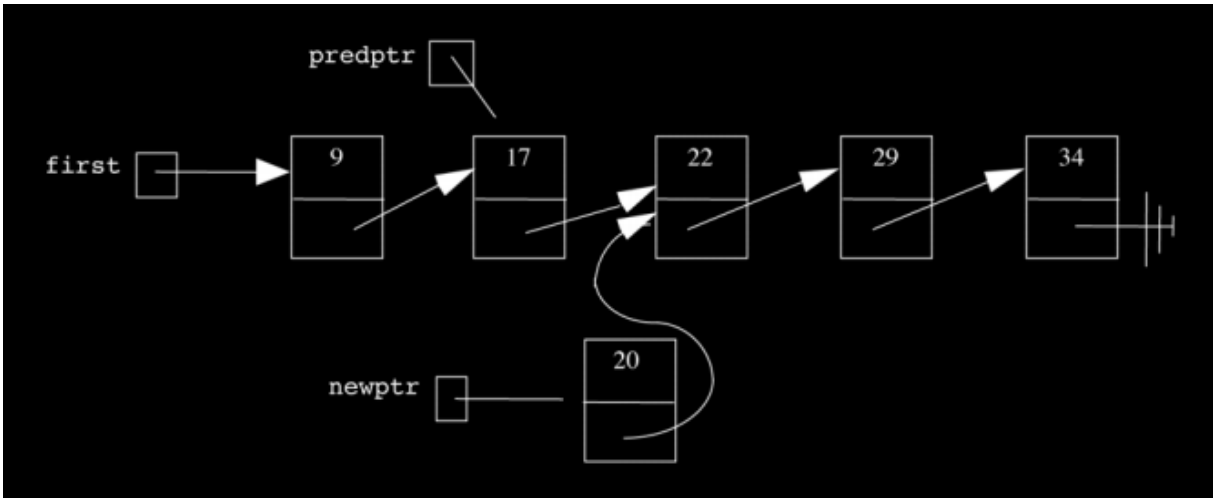
- Here is the corresponding image:



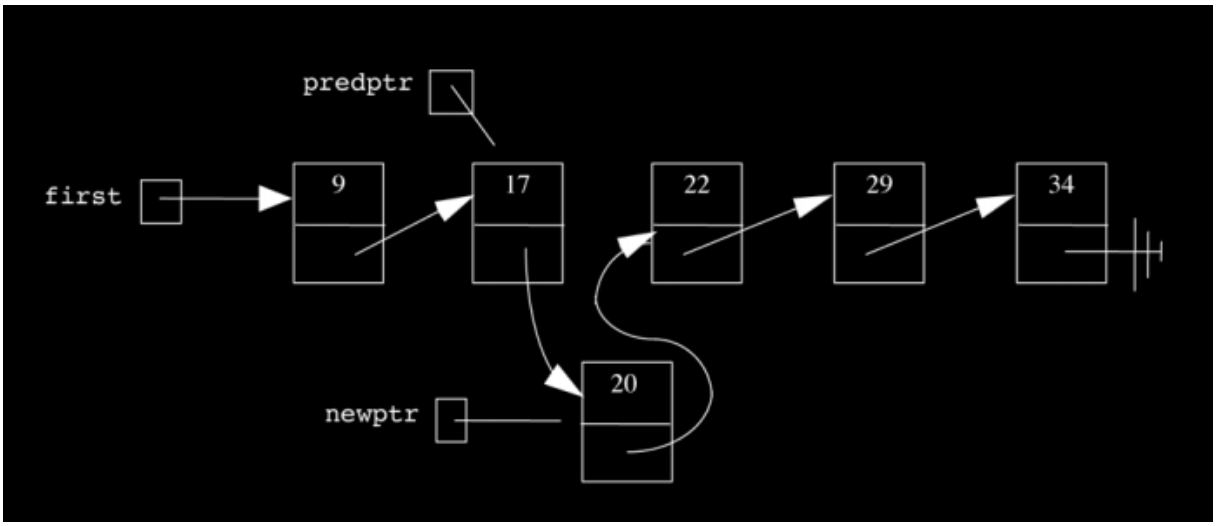

# And in the images, `predptr` and `newptr` are just the names given in the sample code to David's hands when he's pointing around.

- Now let's consider inserting a node into the middle, like the number `20`. We go through the list, and realize that `20` is less than `22`. Now we need `predptr`, predecessor pointer, that points to the element just before. Now when we get to `22`, we can change the pointer in `17`, the element that should be before `20`, to point to `20`:

```
[]----->[5]--->[9]--->[17]--->[20]--->[22]--->[29]--->[34]--->[55]
                                                              |
                                                              V
```



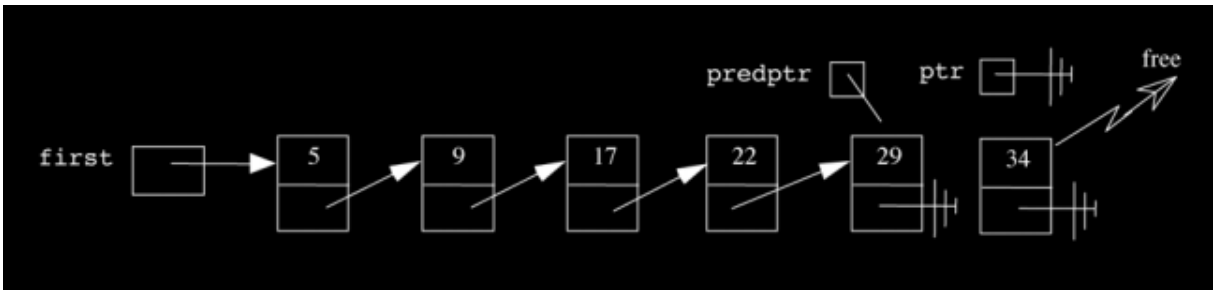

- Notice that `predptr` points to `17`, so we can update it to point to `20`, and then we're done:



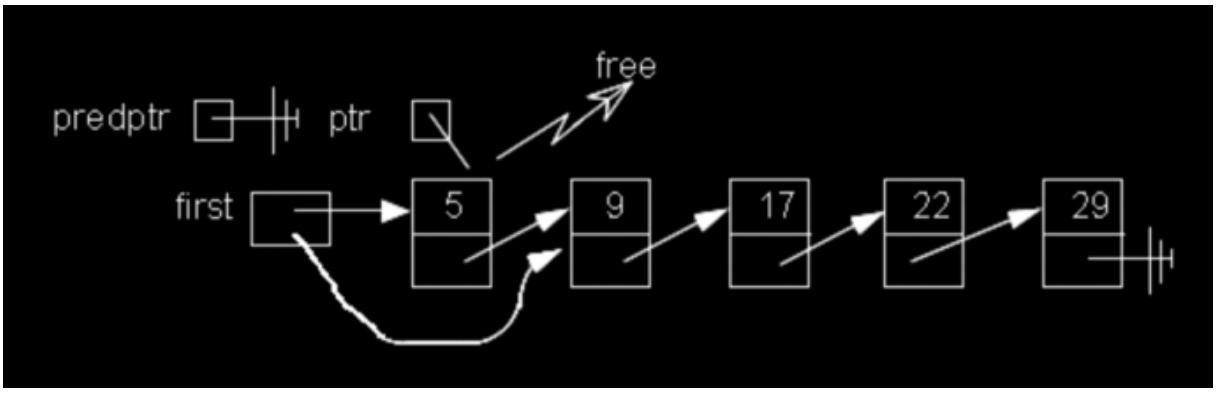

- Even though this seems like a pain to do, the operations are pretty simple with just a few lines of code, but require some logic to figure out where we are and where we should move things.

## Removing

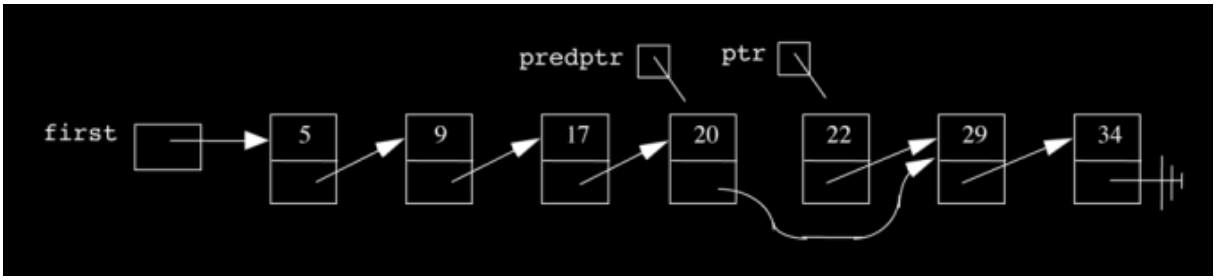
- We can also implement a `remove` function, where, if we wanted to remove `55`, we have to `free` that node, and change the pointer in `34` to point to `NULL`, lest we go back down the list and try to access memory that we've returned




- Removing the head of the list also requires some care. If we wanted to remove `5`, we'd change `first` to point to `9`, but also `free` `5`, or else we'd leak memory by keeping nodes we're no longer using:




- Removing in the middle follows similar logic:




- But the important thing to consider is the running time. We've seen #, upper bound, and $\Omega$, lower bound, of an algorithm, so let's think about linked lists.

    # Running time of `search` has #(*n*) if the element was at the end. And even if our list was sorted, we can't jump around with binary search since we'd have to start from `first` and move through. In the best case, we would have $\Omega(1)$ if the element we were looking for was first.
    # Deleting an element would also have #(*n*) and $\Omega(1)$, the same logic. Even though we had to take multiple steps to remove even the first element, it is still constant time since it takes the same amount of time no longer how big the list is.
    # This seems like a waste of time since we used to have something on the order of `log n` for binary search. But a linked list allows us to add elements, and we'll see this theme of having a tradeoff (just like how merge sort was faster, but needed extra space). A linked list trades time for flexibility, dynamism, which is a positive feature. And it also requires extra space, a pointer for every element, which doubles the space if we're storing `int`s. But if each node represented a word, then adding a pointer wouldn't be as big a deal. Or even if each node was a `struct` with many fields, so as our data types get bigger, adding an additional pointer is even less significant.

- But is there a holy grail of a data structure where search, delete, and insert are all constant time with #(1)? Find out next time.