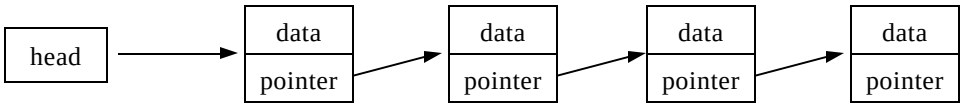
Linked Lists

What's a Linked List?

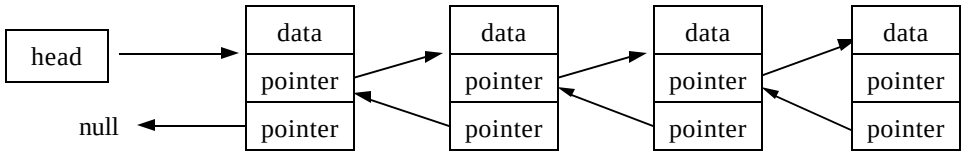
Linked lists are like arrays except that programmer manages the links between data nodes instead of the compiler. They consist of collections of objects, each containing data and methods along with and one or more pointers to the 'next' object[s] in the list.

There are many different types of linked lists! As an example, here are a few:

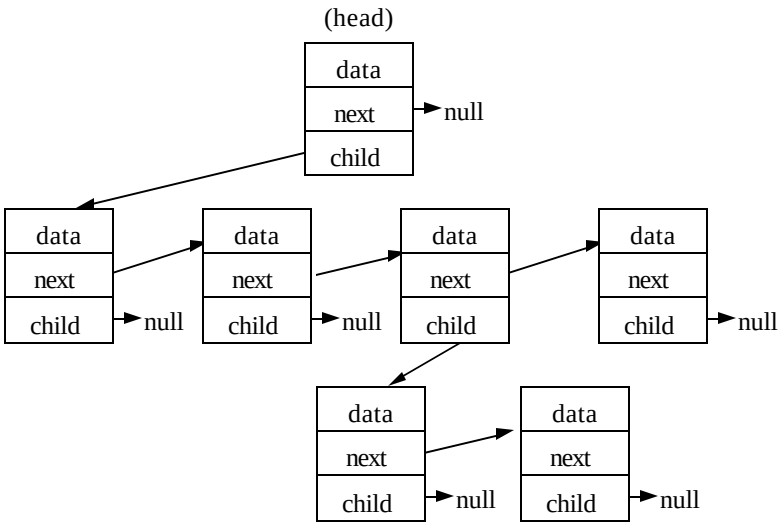
Single Linked Lists: Lists that can only be traversed one way. Very simple to build.



Double Linked Lists: Lists that can be traversed both ways. A little more difficult to build.



Trees: Lists that are built on a 'tree' principle. There are hundreds of different types of trees, and before this class is over we might see a couple of them!



Building a Linked List

The first thing you should do when confronted by a problem is decide whether an array or a linked list would be a better solution. Here's a table to explain what this means:

Worst Case Comparison of Lists and Arrays					
Type of ADT	Creation	Adding	Deleting	Sorting	Memory Use
Arrays	*Easy*	*Hard*	*Hard*	*Easy*	*Fair*
Single Lists	*Easy*	*Easy*	*Hard*	*Hard*	*Good*
Double Lists	*Easy*	*Medium*	*Medium*	*Hard*	*Good*
Trees	*Varies*	*Varies*	*Varies*	*Varies*	*Good*

The hardest part of building a linked list is making sure you keep track of all your pointers and avoid memory leaks. The best way to do it is to spend a LOT of time in the design phase. DRAW PICTURES!

Example:

It is the year 2020, and you've been asked by the Admiral of Fleet Command, Brian Tjaden, to build a program that will keep track of all the starships in the fleet.

You decide that since starships are coming out all the time and its easiest to add to a linked list, that's what you're going to use.

Step One: Design

What type of linked list are we going to use? Why? What will be stored in the data section of each node?

Lets list all the functionality we're going to need for our list.

Now, lets draw some example pictures for these so we know what our code is supposed to do.

Step Two: Code it up!

How will we implement each of our methods?

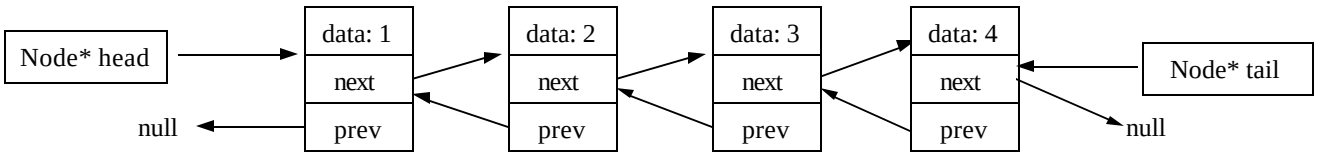
Recursion and Linked Lists

As was said in lecture, linked lists are naturally recursive. This means that it's really easy to write recursive functions to traverse the list!

Take out a sheet of paper. Given the following class:

```
class Node {
        public:
        int data;
        Node* next;
        Node* prev;
}
```

Write two recursive functions dealing with the following list made up of objects of type *Node*:



1) Write a recursive function that prints it from head to tail, and an example of how to call it.

2) Write a recursive function that prints it *in reverse*, or tail to head, and an example of how to call it.

Hint: Remember what we talked about with recursion! What's the base case—the condition that will kick out of the recursion—going to be for these functions?

Don't worry, we'll go through the answers in a moment.

Answers:

1)

```
void printHeadtoTail(Node* ptr){

        if (ptr == NULL) return;
        else {
                cout << ptr->data << endl;
                printHeadtoTail(ptr->next);
                return;
        }
}

printHeadtoTail(head);
```

2) Two possible ways to do this!

I)

```
void printTailtoHead(Node* ptr){

        if (ptr == NULL) return;
        else {
                printTailtoHead(ptr->next);
                cout << ptr->data << endl;
                return;
        }
}

printTailtoHead(head);
```

- OR -

II)

```
void printTailtoHead(Node* ptr){

        if (ptr == NULL) return;
        else {
                cout << ptr->data << endl;
                printTailtoHead(ptr->prev);
                return;
        }
}

printTailtoHead(tail);
```