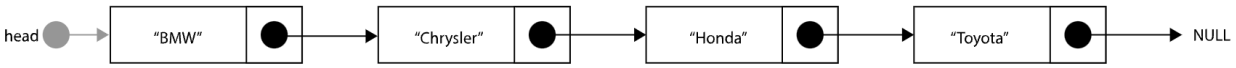
LINKED LISTS

A Linked List is a linear data structure, similar to an array.

It is considered an **Abstract Data Type** because it stores data and has methods to process the list such as add(n), remove(i) for example.

- Arrays require blocks of ***contiguous*** memory. Linked Lists don't.
- Arrays can be processed by using ***indexes***. Linked Lists, conceptually, can't (*however Java's LinkedList class does make this possible!*)

A LinkedList is composed of **Nodes**. Here is a LinkedList of **Nodes** storing Strings:



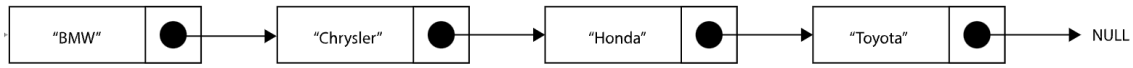
As you can see, each **Node** is composed of **data** and a pointer to the **next** Node in the list.
We need an **external** reference to point to the first Node in order to process it. This is often called the head pointer.
We know we have reached the end of the list when a Node's next pointer points to null.

We could imagine the UML diagram of the Node class to look like this. Let's keep it simple and not incorporate data hiding:

Node
data: String next: Node
Node(data)

Sample Exam Question 1

Here is a LinkedList of Nodes. Explain why it would be impossible to process this list.



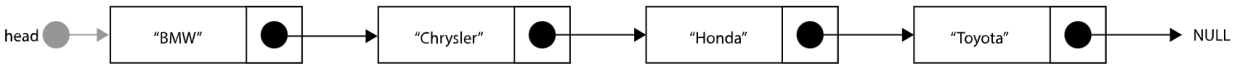
Traversing a LinkedList.

An element in an array data structure can be accessed directly if we know its index eg:

cars[3] = "Toyota"

However, LinkedLists are not processed using indexes because LinkedLists do not exist in **contiguous** blocks of memory.

Therefore, to access a Node, we have to go searching for it, always starting at the **head** of the List.
Using a loop to follow the **next** pointers until null is reached:



Consider this pseudocode:

```
searchItem = "Honda"

loop while head.data!=searchItem and head.next != null
     head = head.next //can you see what this is doing?
end loop

if head.data != searchItem then
     output searchItem, " not found"
else
     output searchItem
end if
```

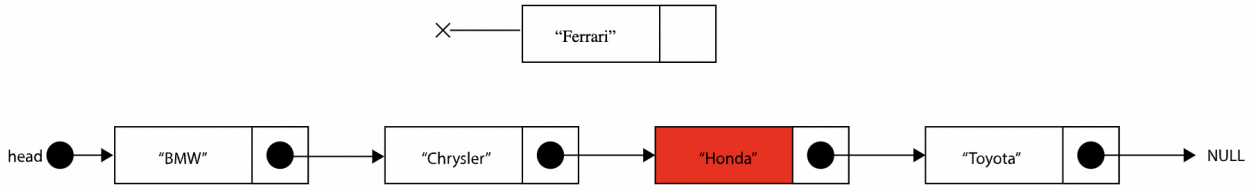
Sample Exam Question 2

Redraw the LinkedList to show the position of the head pointer after the above algorithm is executed.

Sample Exam Question 3

The above algorithm will work, but this is a very risky approach to processing a LinkedList. Why?

Inserting a Node into a LinkedList.



Sample Exam Question 4

An instance of the Node class has been constructed and we want to insert it into a LinkedList.

Order the steps, 1-4

Until the Node containing data greater than the new Node's data is found	
Make the new node's next pointer the same as the next pointer of the previous Node (the one containing "Chrysler")	
Make the previous Node's next pointer (the one containing "Chrysler") point to the new Node, X	
Starting at the head, follow the next pointers	

Sample Exam Question 5

Why is it important to do Step 3 before Step 4 in the above (or any) scenario?

Deleting a Node from a LinkedList.

Similar to inserting a new Node, when deleting a Node from a list we need to search for the Node starting at the head, and when we find it, we need to rearrange pointers! And "*Hey Presto!*"

Let's delete "Honda" from the list above.

Sample Exam Question 6

In your own words, describe how to delete "Honda" from the LinkedList.

RECURSION

Here is a sample algorithm that will traverse a LinkedList, myList, and output all data recursively:

```
recursiveList(myList, nextPointer)

      //base case
      if nextPointer = null
            return
      else
            output nextPointer.data
            recursiveList(myList, nextPointer.next)
      end if

end recursiveList
```

To view more info on LinkedLists, see the unit page on mistermoosi and/or the IBDP CS Course Guide and/or your own research on the Internet.

Also, how about asking ***AI*** to generate a recursive algorithm to process a LinkedList? Ask it to do it 2 ways: once using Java's LinkedList class and once not using Java's LinkedList class!

Things I am wondering about