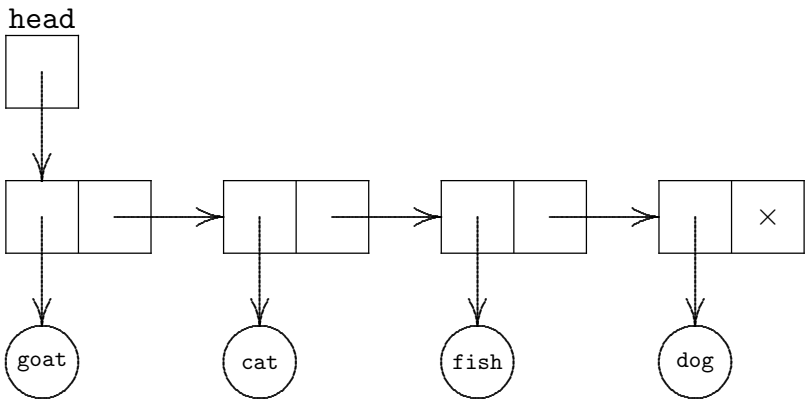
The Linked List Technique

So far we have seen, in considerable detail, how arrays can be used to store collections of objects. Now we want to study a general technique for storing a collection of objects without using an array.

The basic idea is simple but clever: each *node* object holds some data—the item in the collection stored at that node—together with a reference to the next node in the collection.

This picture captures the idea of this technique, used to store the sequence of items `goat, cat, fish, dog`:



There is only one named memory cell in this picture, namely `head`, which contains a reference to a node instance. The 2 by 1 rectangles represent node instances with the first cell being the instance variable named `data` containing a reference to a data item, and the second cell being the instance variable named `next` containing a reference to the next node. The circles represent data instances, perhaps `String` instances in this example. The $\times$ in the last cell represents the `null` reference.

$\Leftarrow$ At the beginning of this course we implemented the list ADT using an array to hold the items in the list. Now we will implement a list of strings using the above idea instead of an array. We'll name the class that holds the list the `SLSList` (for "singly linked string list") class, and we'll name the node class `Node`. We will implement the methods we had in our version of the `ArrayList` class, with exactly the same headers, but totally different internals.

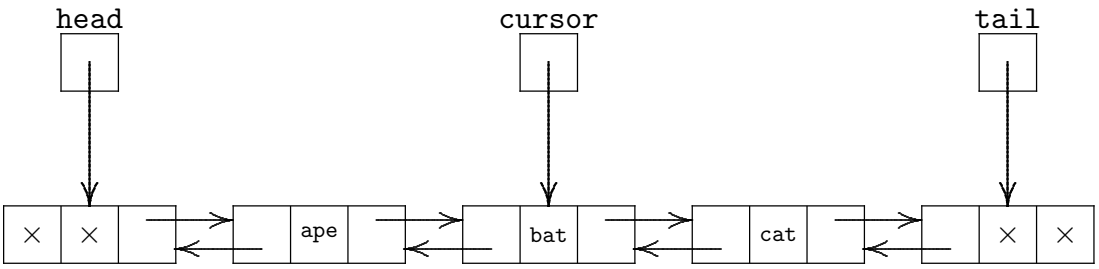
$\Leftarrow$ As we implement the methods, we will perform worst-case efficiency analysis with n items stored in the list, where the representative operation will be storage of a node reference.

Implementing the Editable List ADT by a Doubly-Linked List

Now we want to think about implementing the *editable list ADT* by using a fancier kind of linked list. The editable list ADT consists of a list of items, but instead of being randomly accessible as with the usual list ADT, the editable list can only be accessed with respect to the current item, known as the cursor. For simplicity (and to match Exercise 3), we will assume that the items in the editable list are strings.

⇐ Design the editable list ADT, bearing in mind that the goal is to allow for typical operations like we have seen with the list ADT and the sack ADT, but all happening with respect to the cursor. And, bear in mind that we want to be able to implement a text editor, like in Exercise 1, using an editable list to store the list of lines.

Now, to efficiently implement the methods of the editable list ADT, we will use this picture:



Being able to understand, draw, and implement diagrams like this is a crucial skill in this course, so we should state precisely what things in this sort of diagram represent:

> Individual squares represent simple memory cells that hold a single value, which can be either a reference to some object or a primitive value. If a cell holds `null`, we write × to show its value. If a cell holds a non-null reference to an object, we draw an arrow from inside the cell to the edge of the object it refers to.
>
> Sub-divided rectangles represent objects, with the individual instance variables being represented by the component squares.
>
> For clarity, we draw the strings directly in the `data` instance variable, even though really those squares should hold a reference to the data objects, as in our diagram of the singly-linked list.

⇐ Now we will begin the process of implementing the editable list using the diagram above (you will be asked to finish the implementation in Exercise 3, along with converting your Exercise 1 text editor to use an editable list to store the document). Specifically, we will design and implement the instance variables and sketch out some of the method bodies with this exciting new scheme, and we will create the `DLLNode` class (for "doubly-linked list node").

⇐ Do efficiency analysis of this new approach and compare to the efficiency of the array-based version.

Exercise 3 [10 points] Due in the class period 2 weeks after we reach this point in class

Complete the `EditableList` class and create a class `Ex3` that is like `Ex1` (done correctly!) but uses an `EditableList` instance to store the strings of the document.

Email me your `EditableList.java` and `Ex3.java` files. These classes must use the `DLLNode` class created in class, and the text editor must work as specified for Exercise 1 (I will be critical of failures to meet the specification precisely)