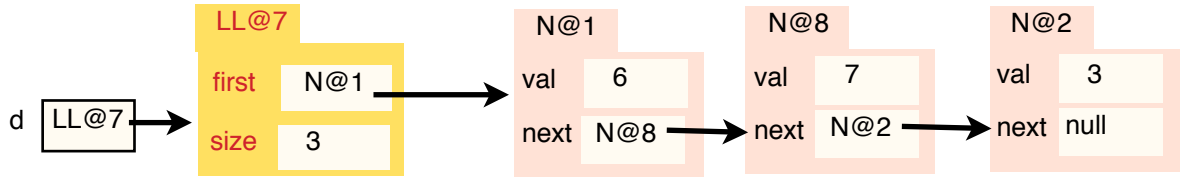
Singly linked lists

The diagram below represents the list of values [6, 7, 3]. The leftmost object, LL@7, is called the *header*. It contains two values: the size of the list, 3, and a pointer to the first *node* of the list. Each of the other three objects, of class `N` (for *Node*) contains a value of the list and a pointer to the next node of the list —or **`null`** if there are no more nodes in the list. This data structure is call a *singly linked list*, or just *linked list*.



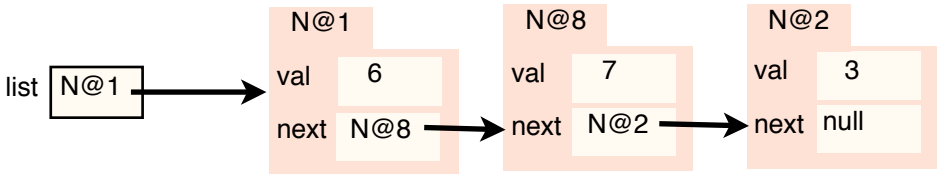
A singly linked list has these advantages: (1) The list can be any size, and (2) Inserting (or removing) a value at the beginning can be done in *constant* time. It takes just a few operations, bounded above by some constant: Create a new object and change a few pointers. On the other hand, to reference element number `i` of the list takes time proportional to `i` —one has to sequence through all the nodes numbered `0..i-1` to find it.

Exercise

At this point, you will gain more understanding by doing the following to construct a linked list that represents the sequence [4, 6, 7, 3]. Do what follows, don't just read it. (1) Copy the linked list diagram shown above. (2) Below that diagram, draw a new object of class `N` , with 4 in field `val` and **null** in field `next`. (3) Change field `next` of the new node to point to node N@1 and change field `first` to point to the new node. (4) Change field `d.size` to 4. The diagram now represents the list [4, 6, 7, 3].

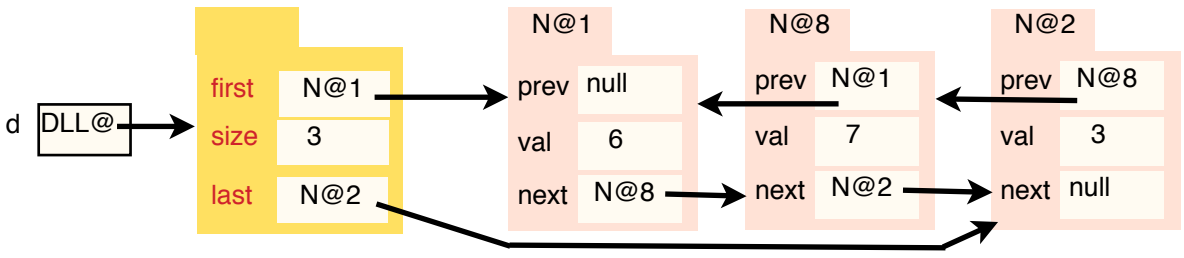
Linked list without a header

Below is a linked list without the header. We sometimes work with linked lists without headers. The disadvantage is that we don't know the size of the list without writing a loop to calculate the number of nodes.



Doubly linked lists

A singly linked list has field `first` in the header and field `next` in each node, as shown above. A *doubly linked list* has, in addition, a field `last` in the header and a field `prev` in each node, as shown below. In the diagram below, one can traverse the list of values in reverse: first `d.last.val`, then `d.last.prev.val`, then `d.last.prev.prev.val`. This doubly linked lists represents the same sequence [6, 7, 3] as the singly linked list given above —but the data structure lets us easily enumerate the values in reverse, [3, 7, 6], as well as forward.



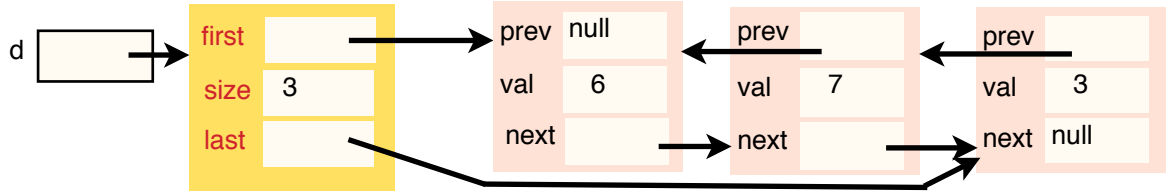
The major advantage of a doubly linked list over a singly linked list is that, given a node `n` (containing something like N@8), one can get to `n`'s prev and next nodes in constant time. For example, removing node `n` from the list can be done in constant time, but in a singly linked list, the time may depend on the length of the list (why?).

Copy the diagram above. Write another variable `n` that points to node N@8. Then figure out what fields have to be changed to delete node `n`, i.e. node N@8, from the linked list. Redraw the new linked list with that node removed.

A doubly linked list allows the following operations to be executed in "constant time" —using just a few assignments and perhaps if-statements, but no loops: Append a value to the list, Prepend a value (insert an element at the beginning of the list), and Insert a value before or after a given element. In an array implementation of such a list, most of these operations could take time proportional to the length of the list in the worst case.

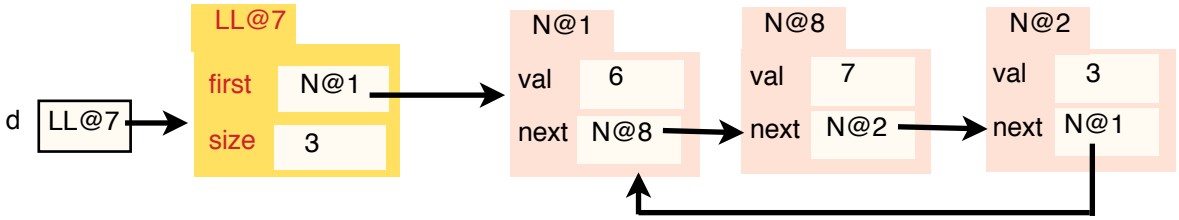
A simplified format

We often write linked lists without the tabs on the objects and even without names in the pointer fields, as shown below. No useable information is lost, since the arrows take the place of the object pointer-names.



Circular linked lists

For some applications, it is useful to use a *circular* linked list, in which the last node points to the first:



Use a circular linked list when it doesn't matter which is the first node. Also, one might use a circular doubly linked list, in which the first node points to the last and the last node points to the first. And one might not need a header node.

Applications

If you are taking a course on data structures, you will see many applications of linked lists. They can be used wherever the maximum size of the list cannot be predetermined and where insertion and deletion of nodes at either end of a list should be efficient operations. Don't be concerned with applications now. Just know that you will see many applications!

Implementations in the Java Collections Framework

An instance of *LinkedList<E>* maintains a doubly linked list with header. This class is used often.