Linked Lists

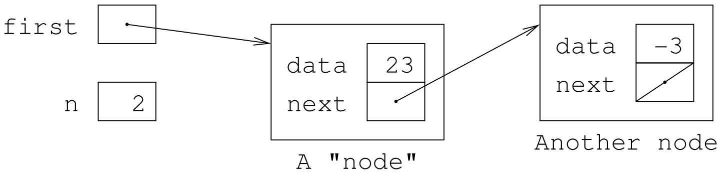
ESC190, Winter 2023

February 28, 2023

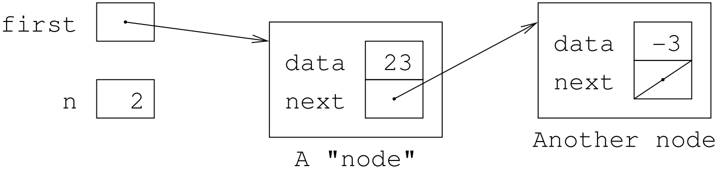
Some slides from Francois Pitt

Linked Lists

- Cannot add an element to an array/block of memory because there may not be space there. (Could move the entire block to a new location with enough space)
- To remove an element from an array/block, need to potentially shift almost the entire block to the left in memory
- Use a "linked-list" structure to store the data instead

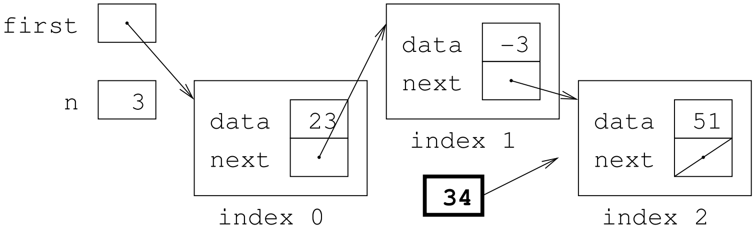


- Each item is stored in a *node* that contains:
  - the value of the item (called the node's *data*)
  - a pointer to the *next* node
- A list consists of two pieces of information:
  - a pointer to the first node
  - the number of elements in the list

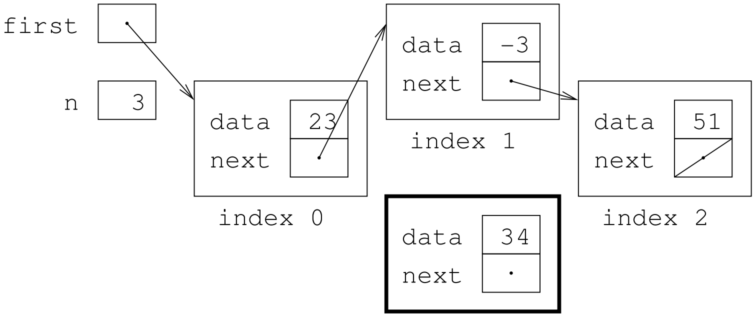


Linked Lists insert

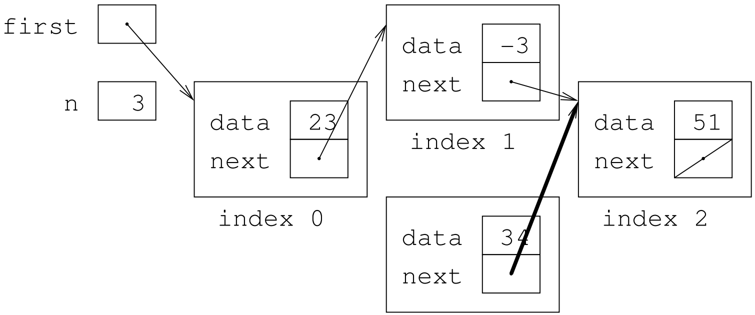
- Suppose we want to insert value 34 at index 2 in the linked list below (*the index of each node is NOT stored in the linked list—it is indicated in the picture for convenience*)



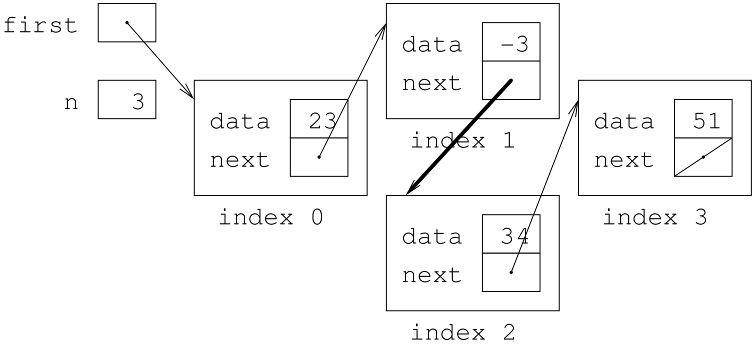
- First, we create a new node to store the new value



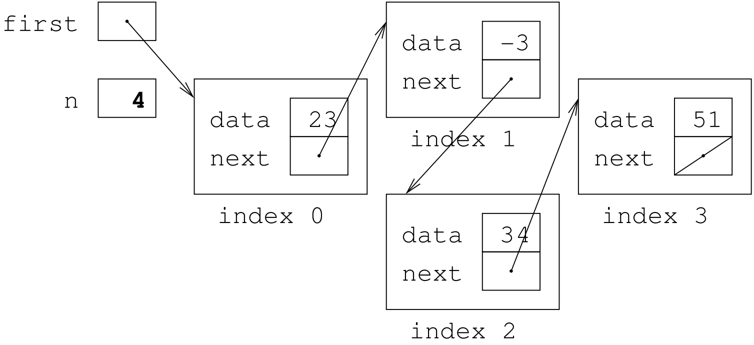
- Next, we set the `next` pointer of the new node to the `next` pointer of the node at index 1



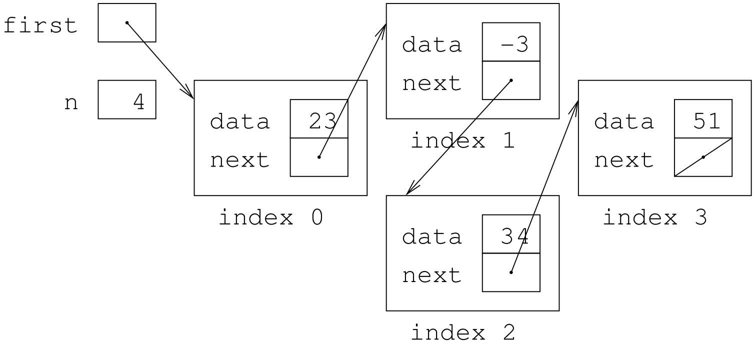
- Next, we set the `next` pointer of the node currently at index 1 to point to the new node



- Finally, we update the value of `n` (it's not necessary to store the number of elements for a linked list, but it is often done for convenience)

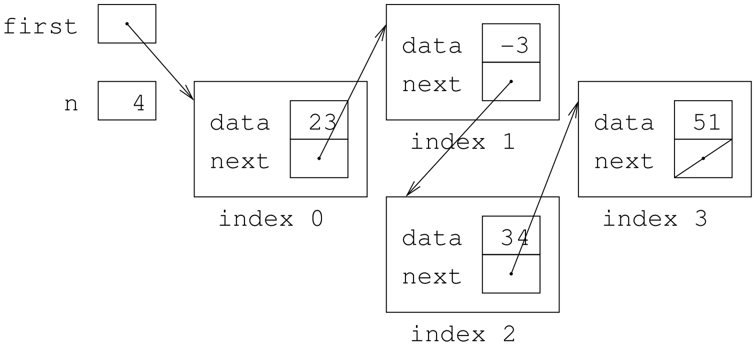


- The complexity is $\mathcal{O}(1)$—*assuming we already have a pointer to the element at index 1*

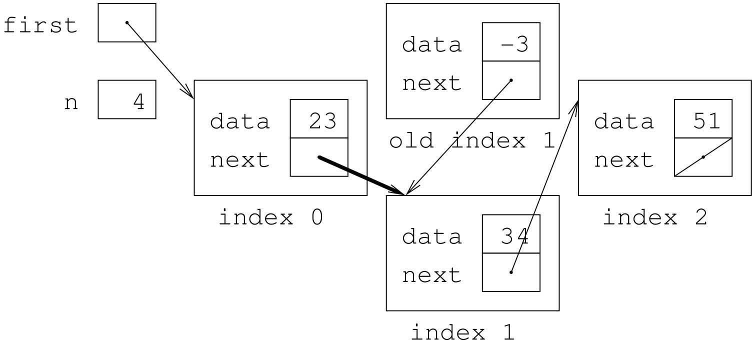


Linked Lists remove

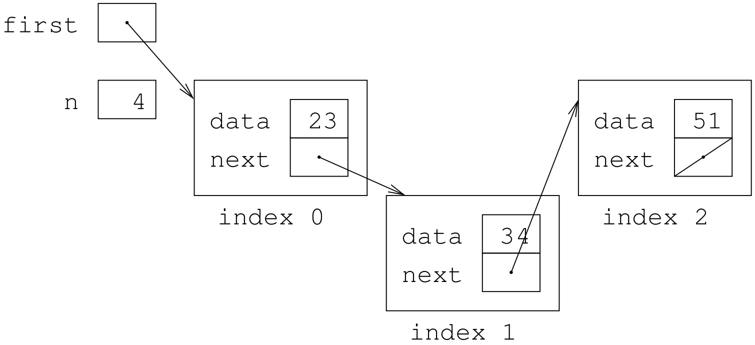
- Now, suppose we want to remove the value at index 1 from the linked list below



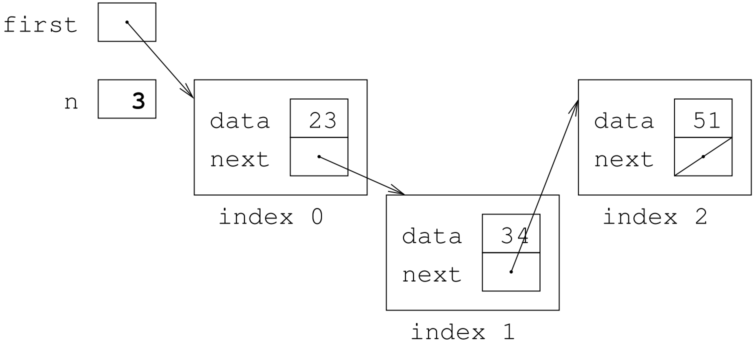
- First, we set the `next` pointer of the node at index 0 to the value of the `next` pointer of the node at index 1



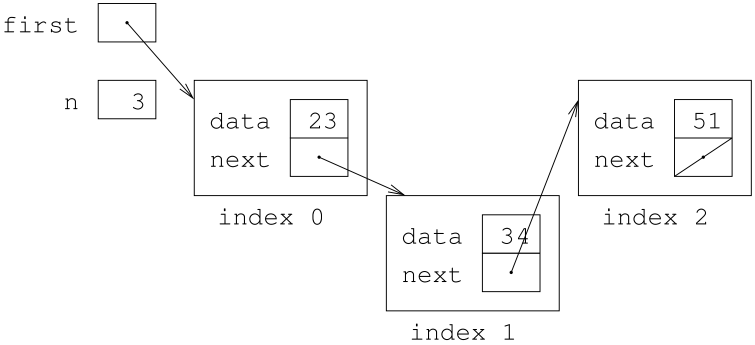
- Next, we "delete" the old node at index 1—meaning we simply release the memory that was allocated for the node



- Finally, we update the value of `n`

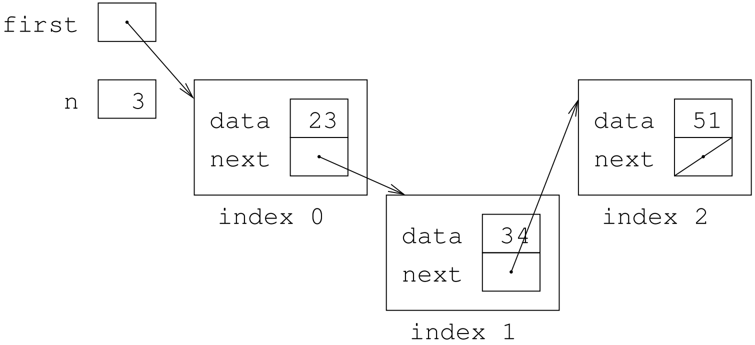


- The complexity is $\mathcal{O}(1)$—*assuming we already have a pointer to the element at index 0*

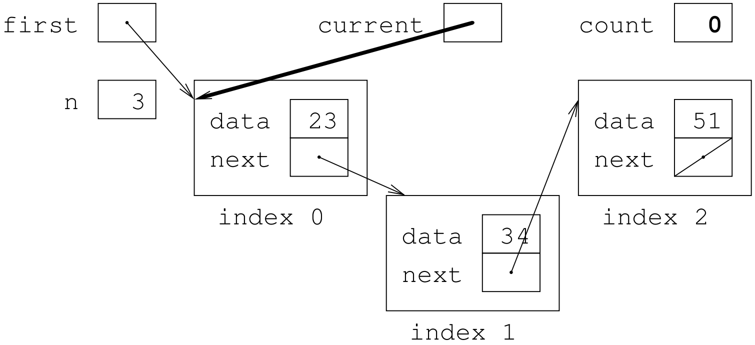


Linked list get

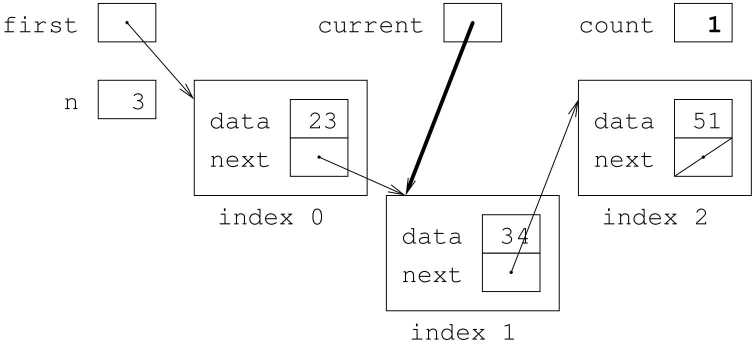
- Finally, suppose we want to get the value at index 2 from the linked list below



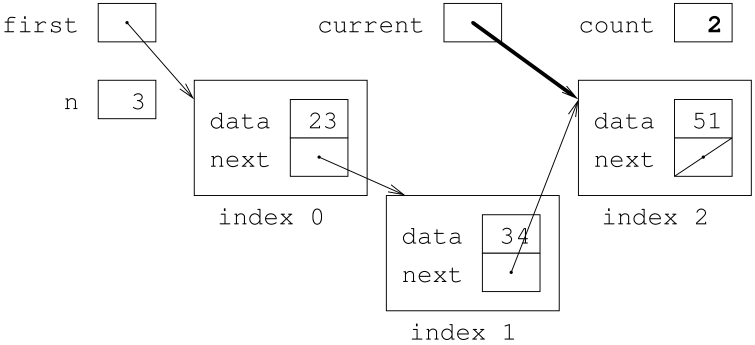
- This requires setting a pointer to point to each node in turn, keeping count, until we reach index 2



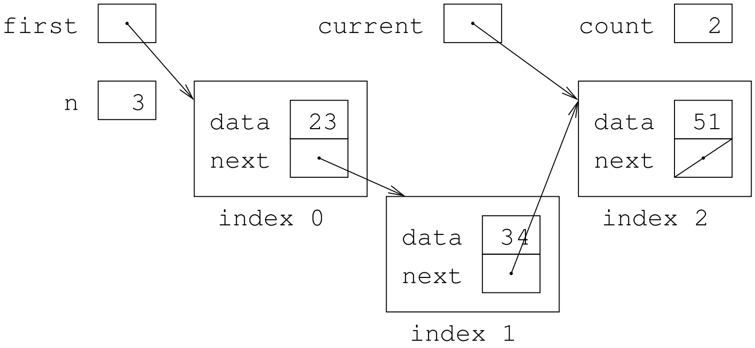
- This requires setting a pointer to point to each node in turn, keeping count, until we reach index 2



- This requires setting a pointer to point to each node in turn, keeping count, until we reach index 2



- The complexity is $\mathcal{O}(n)$ in the worst-case (when retrieving the item at the last index in the list)



Summary

- The worst-case complexity of each list operation for the array data structure and the linked list data structure, where n is the number of items in the list

Operation	Array	Linked List
Insert	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Remove	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Get	$\mathcal{O}(1)$	$\mathcal{O}(n)$ (or $\mathcal{O}(1)$ if index is known)

- The complexity listed for insert and remove for linked lists is only the time taken for the actual insertion or removal—not counting the time required to find the insertion/removal point, which will be $\mathcal{O}(n)$ in the worst-case

- Wait a minute! This means linked lists are no better than arrays, are they?
- Linked lists have one big advantage over arrays: their size is not fixed and can grow and shrink to accommodate exactly the number of values actually stored
- Linked lists are particularly suited to applications where we mostly need to insert or remove values from either end of the list—we'll see examples soon, when we discuss *stacks* and *queues*