202: Computer Science II

Northern Virginia Community College

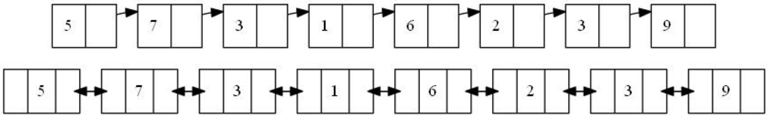
Lists

Cody Narber, M.S.

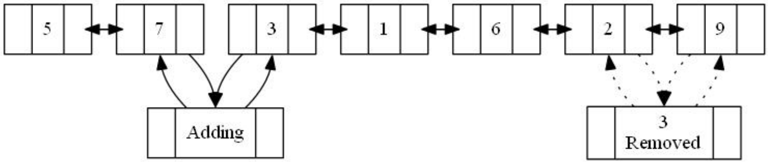
October 14, 2017

Data Structures - List

List: is a data structure that is expandable in size, where each element is linked to either one or two other elements. Below shows a single-linked and double-linked lists constructed with the data: 5,7,3,1,6,2,3,9



To modify the list you need to update the links and add/remove the elements in any location within the list.



Data Structures - List

A **List** is desgined to be the most flexible of the data structures as it allows for the following methods:

- **add** - adds an element to the list (possibly ata a specific index)
- **remove** - finds a specified element and removes it
- **get** - accesses the value at a specific spot
- **contains** - true/false whether the specified element is in the list
- **size** - the number of elements within the list

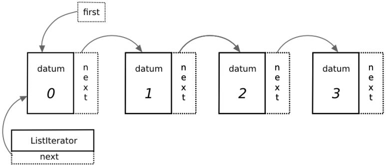
Lists also make use iterators, which are essentially references to spots in between elements, which are used to step through the list. ListIterators have the following methods:

- **hasNext/hasPrevious** - checks if there is a next/previous element (i.e. not null)
- **next/previous** - the iterator is moved to reference the next/previous link in the list, returning the element data of the node passed over.

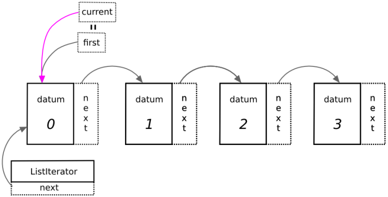
Single Linked List - get(int index)

Illustration of **get(2)**:

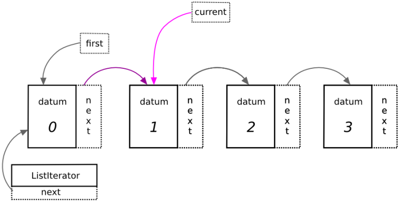
Step 1:



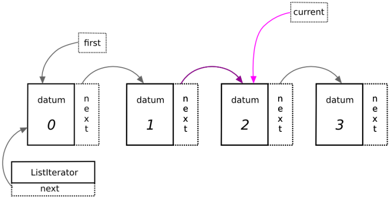
Step 2:



Step 3:



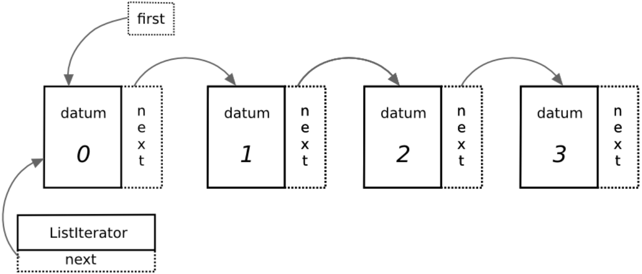
Step 4:



Single LL - add(int index, T element) (1)

Illustration of **add(2, n2)**:

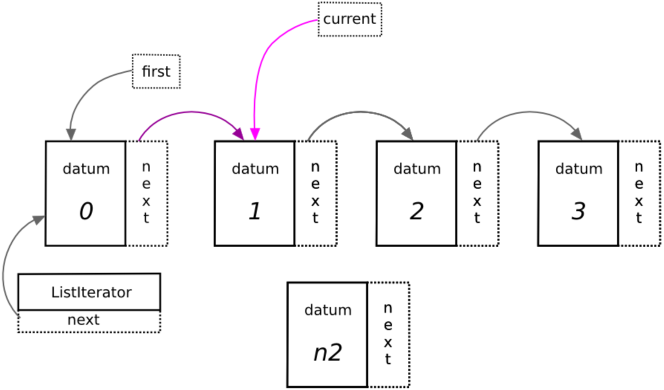
Step 1:



Single LL - add(int index, T element) (2)

Illustration of **add(2, n2)**:

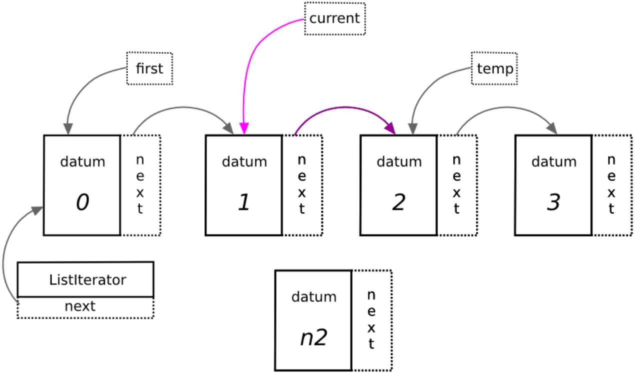
Step 2 - Access the node prior to the space we want to add the new node:



Single LL - add(int index, T element) (3)

Illustration of **add(2, n2)**:

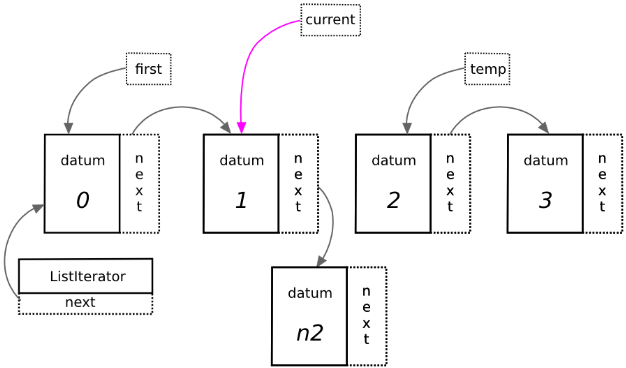
Step 3 - Create a reference to the node currently at index 2, as we will be breaking the link and will need to remember what was there:



Single LL - add(int index, T element) (4)

Illustration of **add(2, n2)**:

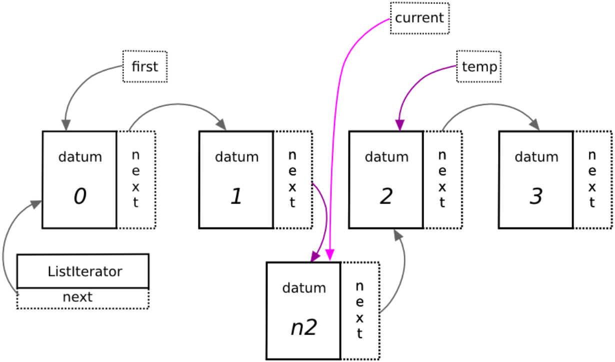
Step 4 - Adjust node at 1 to reference new node as the next node:



Single LL - add(int index, T element) (5)

Illustration of **add(2, n2)**:

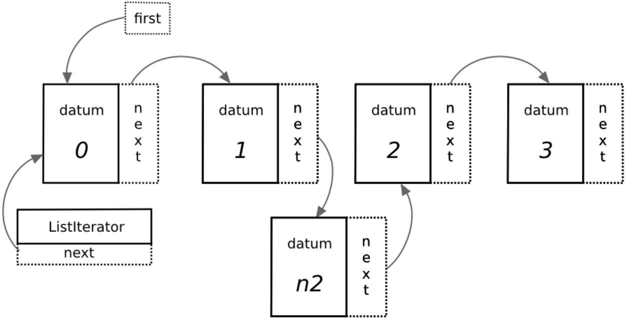
Step 5 - Looking at the new node adjust its next reference to our temporary reference:



Single LL - add(int index, T element) (6)

Illustration of **add(2, n2)**:

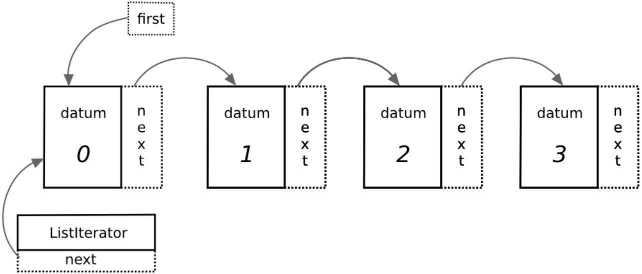
Step 6 - Node added:



Single LL - remove(int index) (1)

Illustration of **remove(1)**:

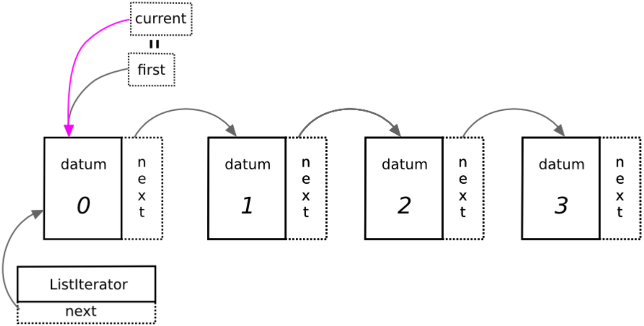
Step 1:



Single LL - remove(int index) (2)

Illustration of **remove(1)**:

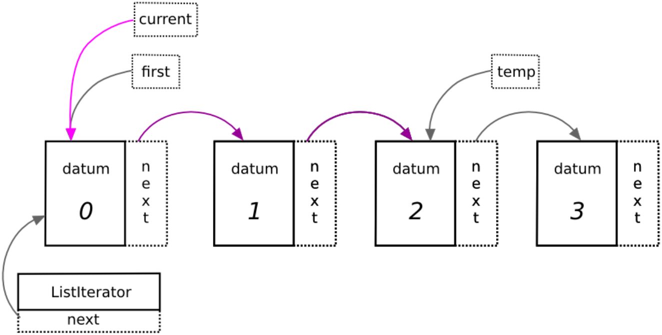
Step 2 - Access the node prior to the space we want to remove the new node:



Single LL - remove(int index) (3)

Illustration of **remove(1)**:

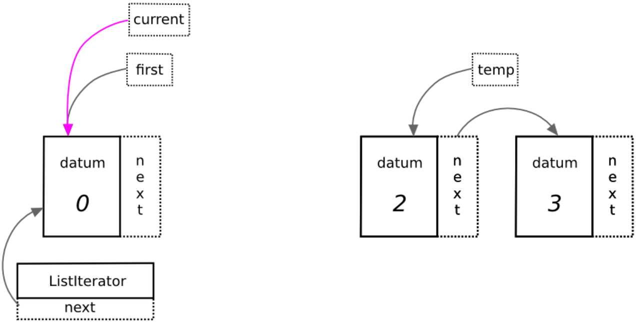
Step 3 - Follow two links to create a temporary reference to the node following the one we wish to delete/remove:



Single LL - remove(int index) (4)

Illustration of **remove(1)**:

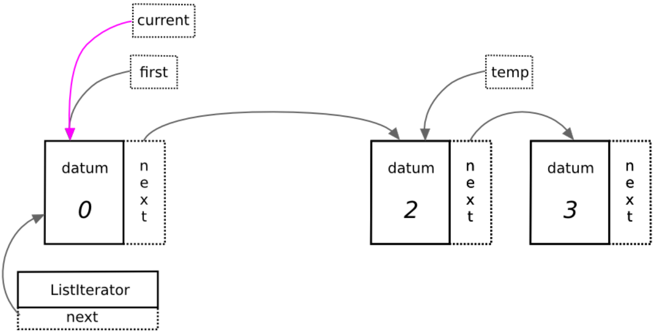
Step 4 - Remove the node (delete)...java will handle this via garbage collection when you move onto next step:



Single LL - remove(int index) (5)

Illustration of **remove(1)**:

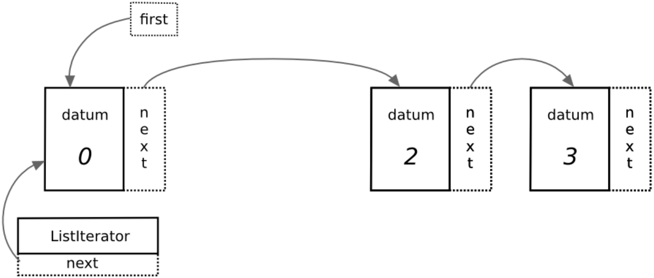
Step 5 - adjust the reference of the current node's next to be the temporary reference:



Single LL - remove(int index) (6)

Illustration of **remove(1)**:

Step 6 - Node Removed:

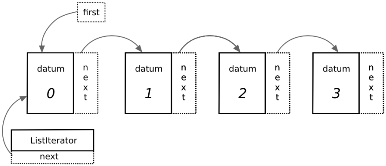


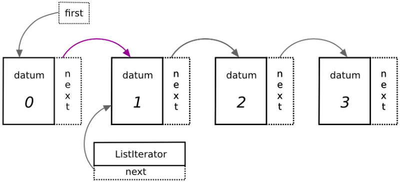
Single Link List Operations

- **add** - adds an element to the list (possibly ata a specific index)
- **remove** - finds a specified element and removes it
- **get** - accesses the value at a specific spot
- **contains** - true/false whether the specified element is in the list
  - **contains(T element)**
  - like accessing/getting a node, you iterate through each node checking each datum of the nodes to see if they are equivalent.
- **size** - the number of elements within the list
  - **size()**
  - an attribute should within the list to keep track of the size, as elements are added or removed the size attribute should be increased and decreased respectively.

Single Link List Iterator

An iterator is created that has references to node(s) in the list, and with each call to next/previous on the iterator, it follows the links on the nodes and updates the iterators references. hasNext/hasPrevious just checks to see if the references are null.

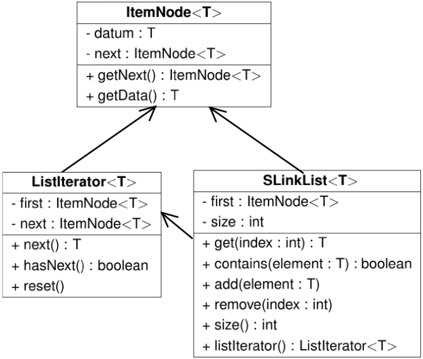




Single Link List Classes

Several things to note here:

- reset functionality given to the iterator, so we need to retain the first when constructed
- Some implementations of Linked Lists have the iterator within the list itself, and this can be easily seen by moving all the attributes and methods from the iterator over to the list class...though we could only have one iterator at a time...



- Separating iterator into another class, we could have multiple iterators going through the linked list at a single time. (WARNING! Allowing this we could have concurrency issues!!!)

Amortized Analysis

Amortized Analysis - amortization is the process of decreasing or accounting for an amount over a period of time. For data structures this relates to looking at the all of the operation/methods running times (Big-Oh) that will be done on a data structure and relate them to the amount of times they will be performed...for example:

Toy Class:

Assume we only construct/terminate the data structure once. Will in use for every call to Process we do approximately 5 CheckStates...

Operation	Running Time
Construction	$O(n^2)$
CheckState	O(1)
Process	O(n)
Terminate	O(n)

Toy Class:

Assume we only construct/terminate the data structure once. Will in use for every call to Process we do approximately 5 CheckStates...

Operation	Running Time
Construction	$O(n^2)$
CheckState	O(1)
Process	O(n)
Terminate	O(n)

When using the data structure we assume the methods used during use are called a "LARGE" number of times, given our assumptions we can see the amortized running time is about:

$$TotTime = 1 * n^2 + (5 * m) * 1 + m * n + 1 * n \quad (1)$$

$$TotCalls = 1 + (5 * m) + m + 1 \quad (2)$$

$$Amortized = TotTime/TotCalls = \lim_{m \to \infty} \frac{n^2 + n + m * (5 + n)}{2 + 6 * m} = n \quad (3)$$

Amortized Analysis

Stacks (Array & Link):

Operation
Push
Pop

Queues (Array & Link):

Operation
Enqueue
Dequeue

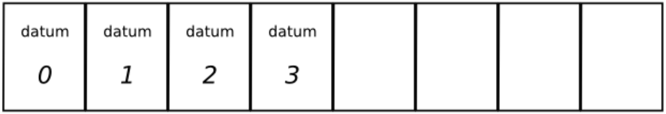
Single Link List:

Operation
get
contains
add
remove

Array-Based List

A **List** is desgined to be the most flexible of the data structures as it allows for the following methods:

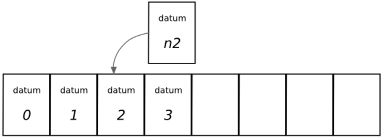
- **add** - adds an element to the list (possibly ata a specific index)
- **remove** - finds a specified element and removes it
- **get** - accesses the value at a specific spot
  - Just access the element within the array
- **contains** - true/false whether the specified element is in the list
  - For loop checking each element
- **size** - the number of elements within the list
  - Access attribute modified on add/remove



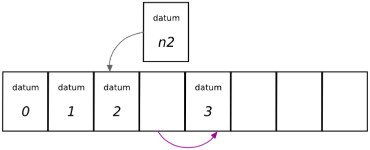
Single LL - add(int index, T element) (1)

Illustration of **add(2, n2)**:

Step 1 - To place in specified index:



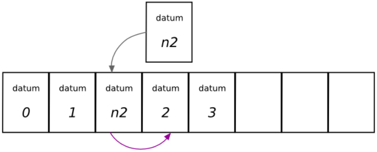
Step 2 - Move elements down:



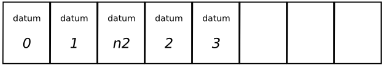
Single LL - add(int index, T element) (2)

Illustration of **add(2, n2)**:

Step 3 - Continue to make room, and put new data in:



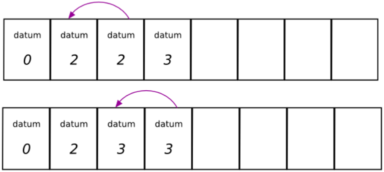
Step 4 - Added:



Single LL - remove(int index)

Illustration of **remove(1)**:

Step 1 - Move elements into space:



Step 2 - Removed:

How to improve runtime...

Variants of the single link list:

- Circular Linked List - the last element links to the first
- Double Linked List - each node has a reference to next and previous
- Maintain Sorted Elements (Comparable)
- Have references to middle and end as well as beginning.

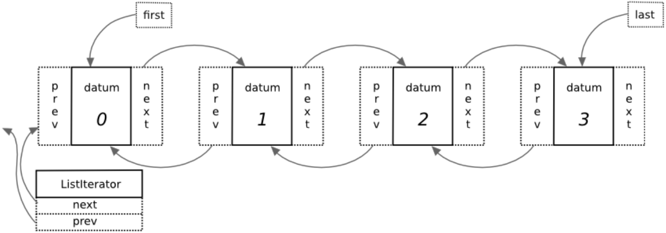
Comparison of functions:

	S. LL	D. LL	Array	Sort S. LL	Sort D. LL	Sort Array
get						
contains						
add						
remove						

Double LList - add(int index, T elem) (1)

Illustration of **add(2, n2)**:

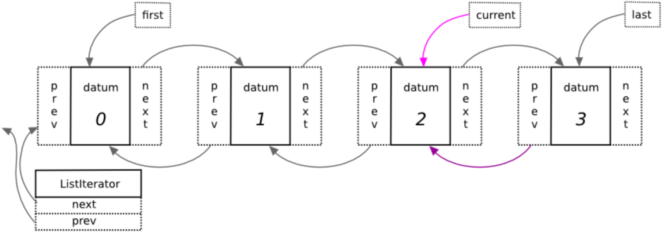
Step 1 - Double Linked List



Double LList - add(int index, T elem) (2)

Illustration of **add(2, n2)**:

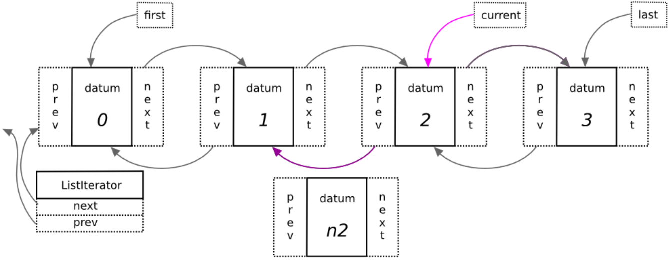
Step 2 - Start at end, since it is closer to index, iterate back to desired point



Double LList - add(int index, T elem) (3)

Illustration of **add(2, n2)**:

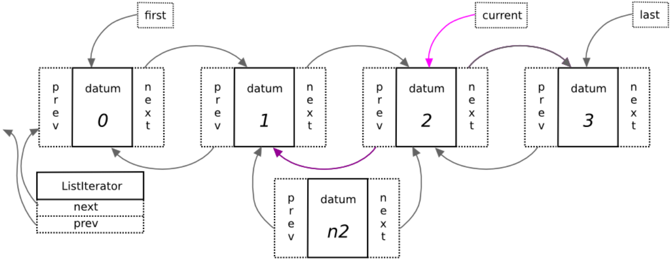
Step 3 - Create new node, and trace from current ind to previous in order to update links



Double LList - add(int index, T elem) (4)

Illustration of **add(2, n2)**:

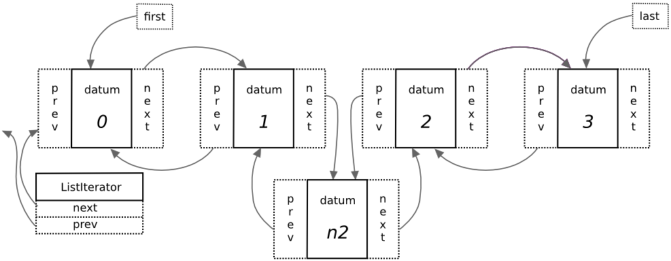
Step 4 - Update references from the new node...



Double LList - add(int index, T elem) (5)

Illustration of **add(2, n2)**:

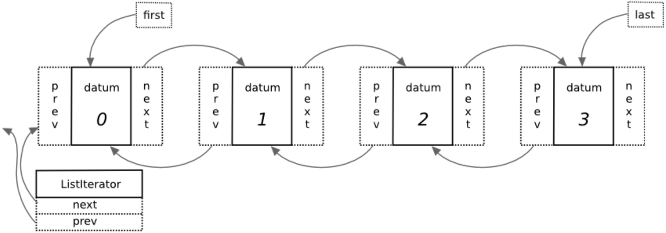
Step 5 - Update references to the new node...Added



Double LList - remove(int index) (1)

Illustration of **remove(2)**:

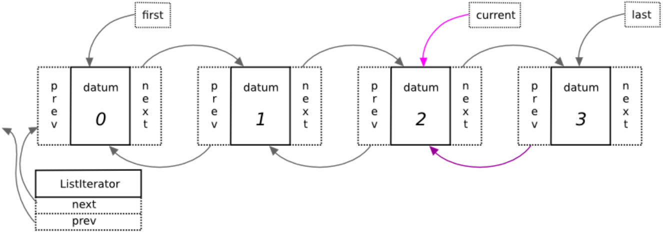
Step 1 - Double Linked List



Double LList - remove(int index) (2)

Illustration of **remove(2)**:

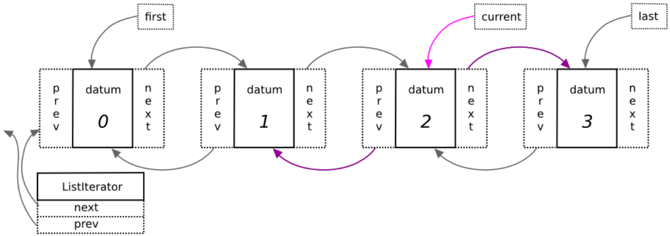
Step 2 - Start at end, since it is closer to index, iterate back to desired point



Double LList - remove(int index) (3)

Illustration of **remove(2)**:

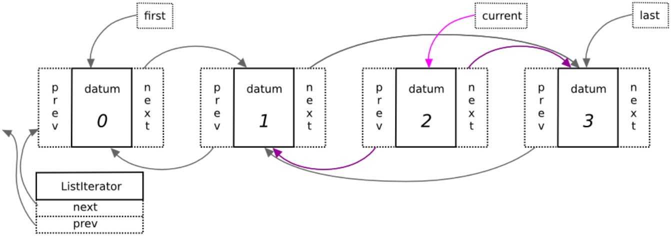
Step 3 - Trace from current ind to previous and next in order to update links



Double LList - remove(int index) (4)

Illustration of **remove(2)**:

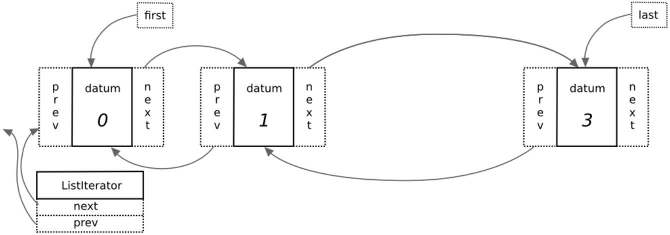
Step 4 - Update links of previous and next to skip over the current node.



Double LList - remove(int index) (5)

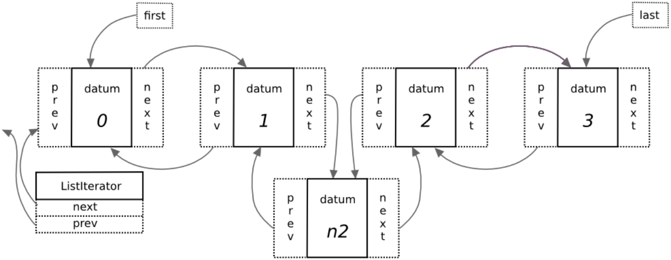
Illustration of **remove(2)**:

Step 5 - Given we still had the reference for the current node, we can remove/delete it (or in the case of Java let garbage collection handle that).



Sorted List - Add

In order to maintain a sorted list, we must modify our add method to no longer take an index, `add(T element)`: this method will iterate the list and as soon as we find a node that has an element higher, we can insert the data just as done before.



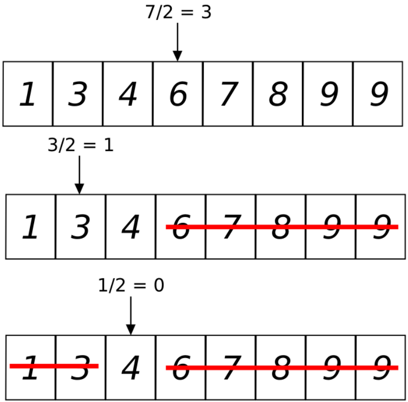
In the example above the location was chosen since :

`datum 1 < datum n2 < datum 2`

Binary Search

Is a search technique that assumes ordered data. Rather than looking at one item at a time, which could take as much as O(n) time, We can use the fact that the data is ordered and we can begin by splitting the data in half and eliminating half of the list as it will either be too large or small.

Example `bsearch(4)`:



Java Collections

Java's Collection Framework is built all around these implementations; Java Collections Framework

- Queue
- List
- ArrayList
- Vector