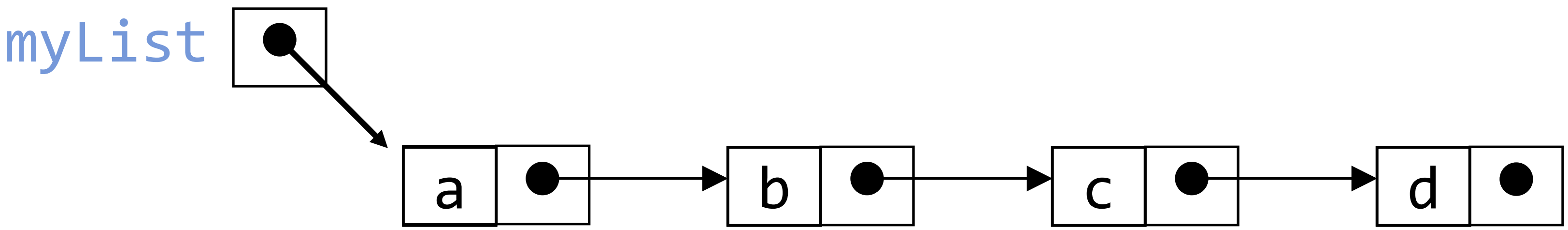
Linked Lists

Anatomy of a Linked List

- A linked list consists of:
  - A sequence of nodes



Each node contains a value
and a link (pointer or reference) to some other node

The last node contains a null link

The list may (or may not) have a header

More terminology

- A node's successor is the next node in the sequence
  - The last node has no successor
- A node's predecessor is the previous node in the sequence
  - The first node has no predecessor
- A list's length is the number of elements in it
  - A list may be empty (contain no elements)

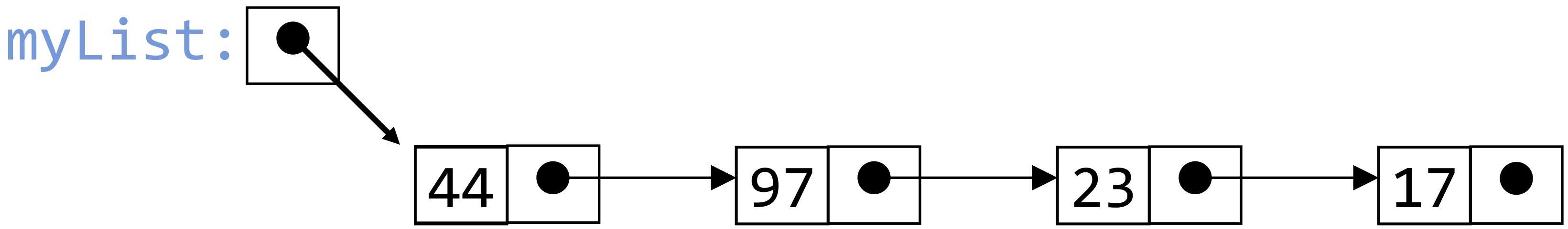
POINTERS AND REFERENCES

- In C and C++ we have "pointers," while in Java we have "references"
  - These are essentially the same thing
    - The difference is that C and C++ allow you to modify pointers in arbitrary ways, and to point to anything
  - In Java, a reference is more of a "black box," or ADT
    - Available operations are:
      - dereference ("follow")
      - copy
      - compare for equality
    - There are constraints on what kind of thing is referenced: for example, a reference to an `array of int` can *only* refer to an `array of int`

Creating references

- The keyword `new` creates a new object, but also returns a *reference* to that object
- For example, `Person p = new Person("John")`
  - `new Person("John")` creates the object and returns a reference to it
  - We can assign this reference to `p`, or use it in other ways

Creating links in Java



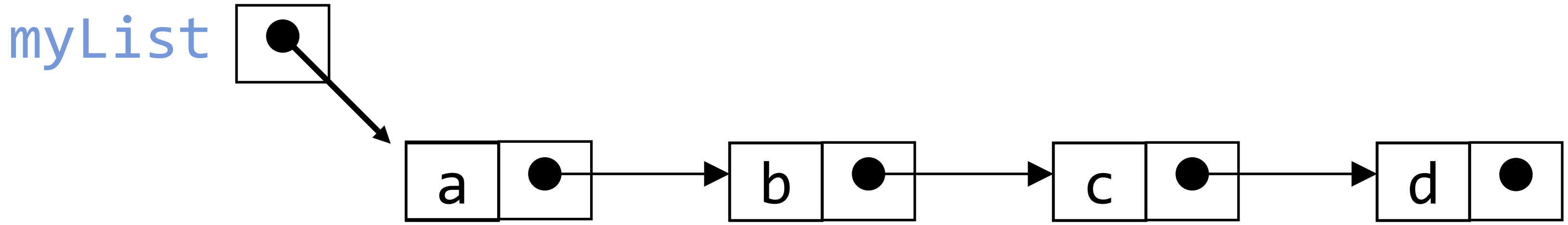

```
class Node {
      int value;
      Node next;

   Node (int v, Node n) { // constructor
         value = v;
         next = n;
   }
}

Node temp = new Node(17, null);
temp = new Node(23, temp);
temp = new Node(97, temp);
Node myList = new Node(44, temp);
```

Singly-linked lists

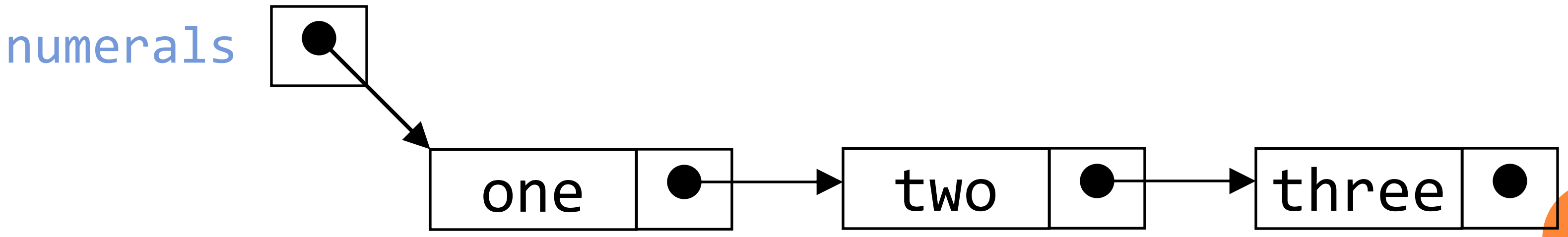
- Here is a singly-linked list (SLL):



- Each node contains a value and a link to its successor (the last node has no successor)
- The header points to the first node in the list (or contains the null link if the list is empty)

Creating a simple list

- To create the list ("one", "two", "three"):
- `Node numerals = new Node();`
- ```
  numerals =
      new Node("one",
          new Node("two",
              new Node("three", null)));
  ```


  ```

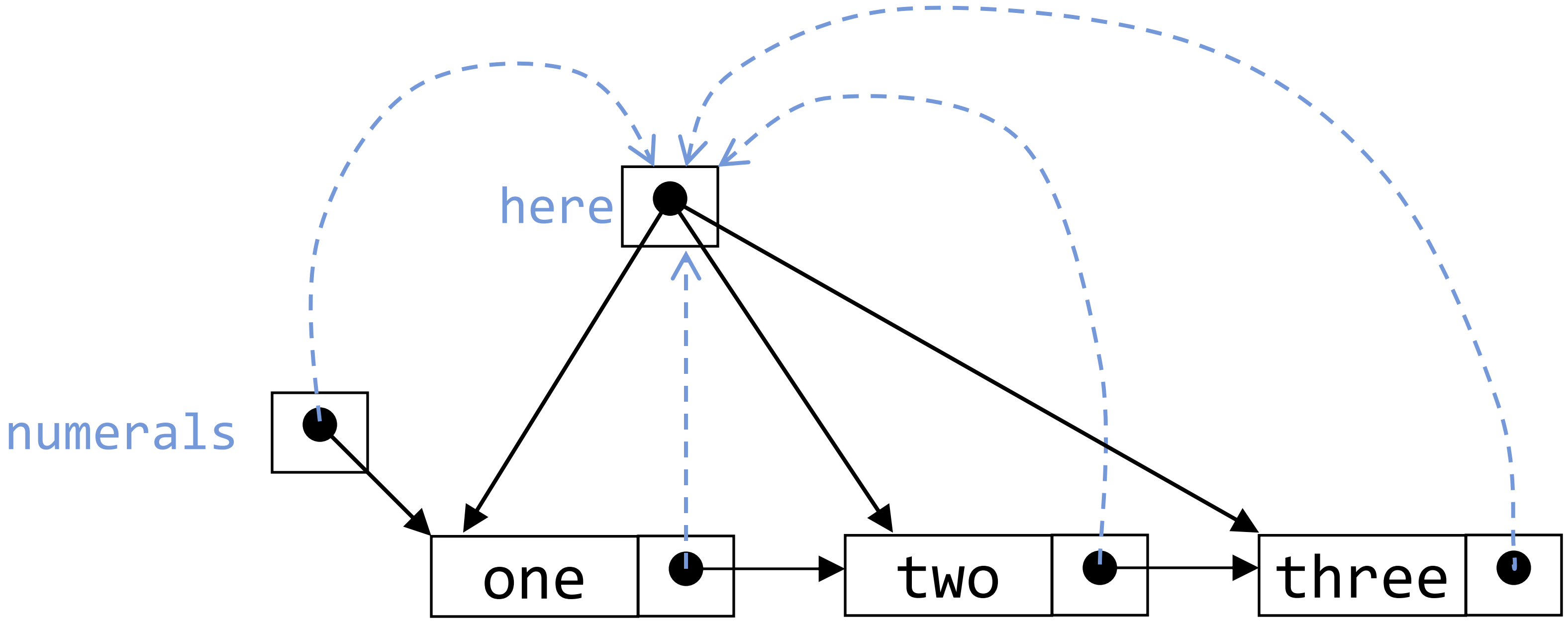
TRAVERSING A SLL

- The following method traverses a list (and prints its elements):

```
public void printFirstToLast(Node here) {
  while (here != null) {
     System.out.print(here.value + "  ");
     here = here.next;
    }
}
```

- You would write this as an instance method of the `Node` class

Traversing a SLL (animation)



INSERTING A NODE INTO A SLL

- There are many ways you might want to insert a new node into a list:
  - As the new first element
  - As the new last element
  - Before a given node (specified by a *reference*)
  - After a given node
  - Before a given value
  - After a given value
- All are possible, but differ in difficulty

INSERTING AS A NEW FIRST ELEMENT

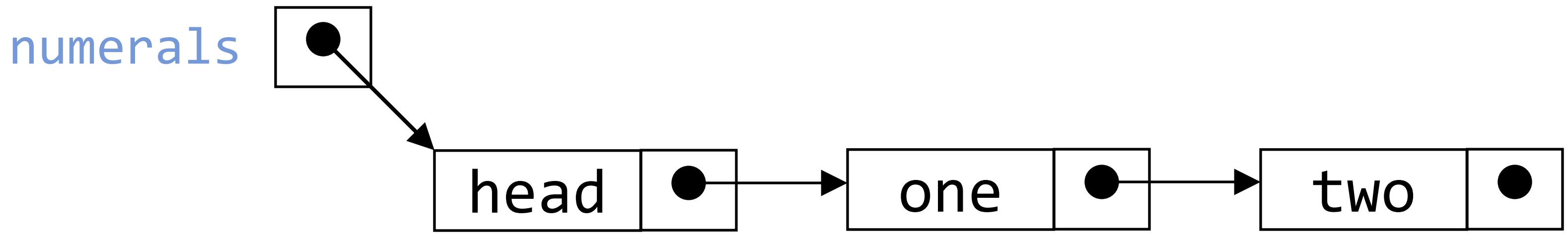
- This is probably the easiest method to implement
- In class `Node`:

```
Node insertAtFront(Node oldFront, Object value) {
    Node newNode = new Node(value, oldFront);
    return newNode;
}
```

- Use this as: `myList = insertAtFront(myList, value);`
- Why can't we just make this an instance method of `Node`?

USING A HEADER NODE

- A header node is just an initial node that exists at the front of every list, even when the list is empty
- The purpose is to keep the list from being `null`, and to point at the first element

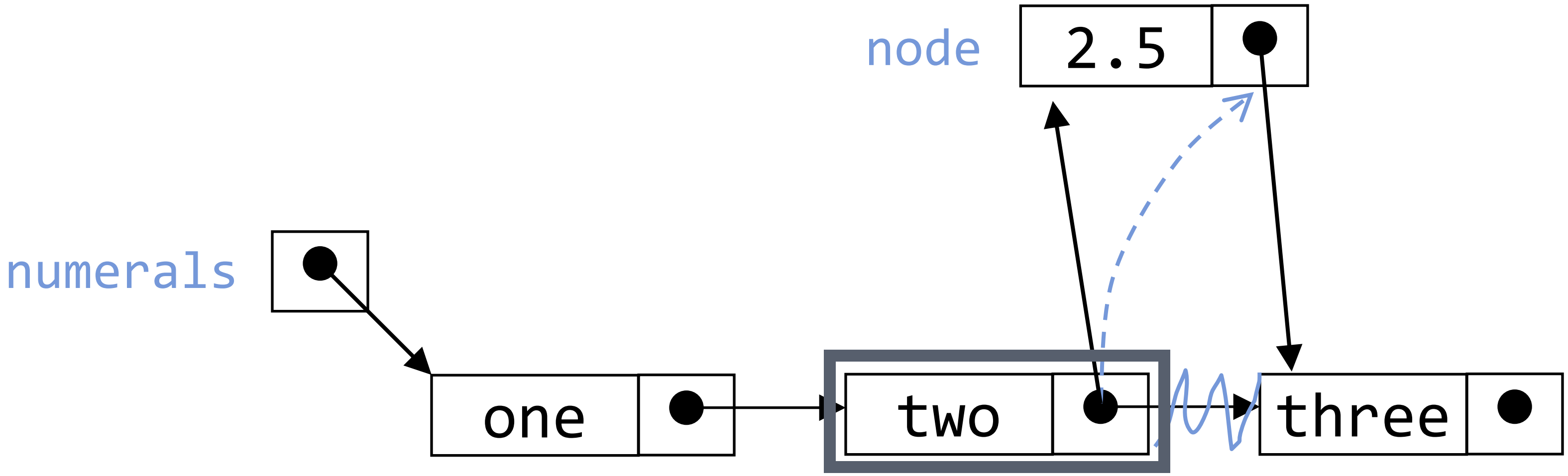



```
void insertAtFront(Object value) {
     Node front = new Node(value, this);
     this.next = front;
 }
```

INSERTING A NODE AFTER A GIVEN VALUE

```
void insertAfter(Object target, Object value) {
    for (Node here = this; here != null; here = here.next)
  {
        if (here.value.equals(target)) {
            Node node = new Node(value, here.next);
            here.next = node;
            return;
        }
  }
    // Couldn't insert--do something reasonable here!
}
```

INSERTING AFTER (ANIMATION)



Find the node you want to insert after

First, copy the link from the node that's already in the list

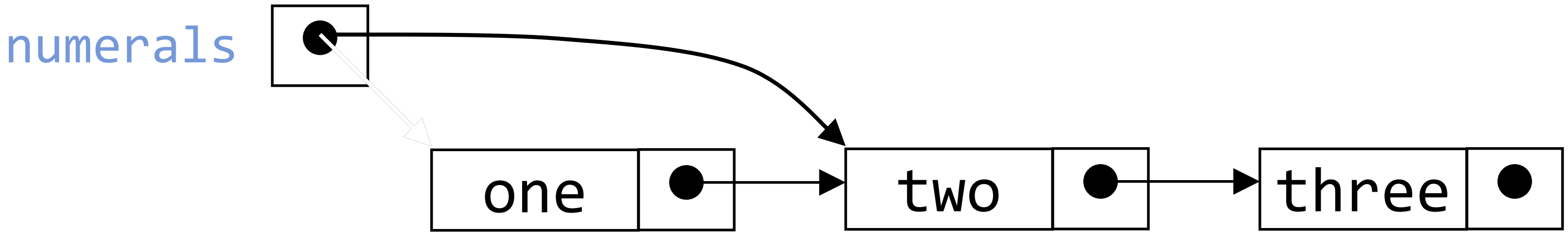
Then, change the link in the node that's already in the list

Deleting a node from a SLL

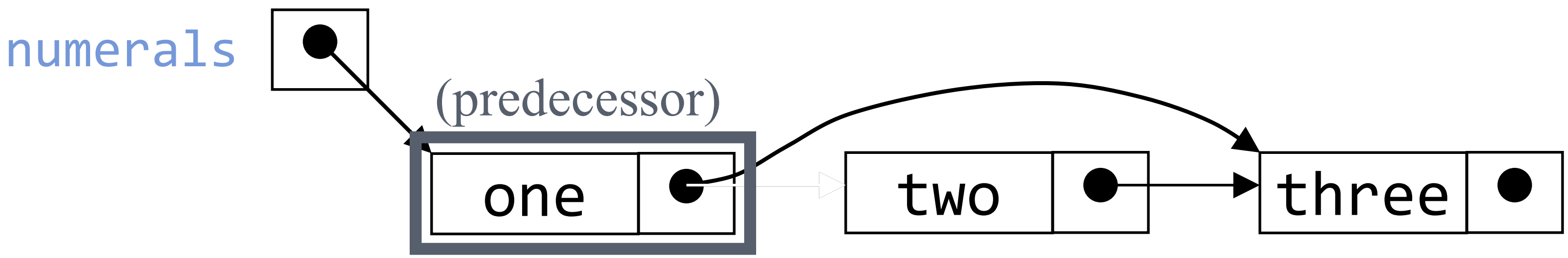
- In order to delete a node from a SLL, you have to change the link in its *predecessor*
- This is slightly tricky, because you can't follow a pointer backwards
- Deleting the first node in a list is a special case, because the node's predecessor is the list header

Deleting an element from a SLL

- To delete the first element, change the link in the header



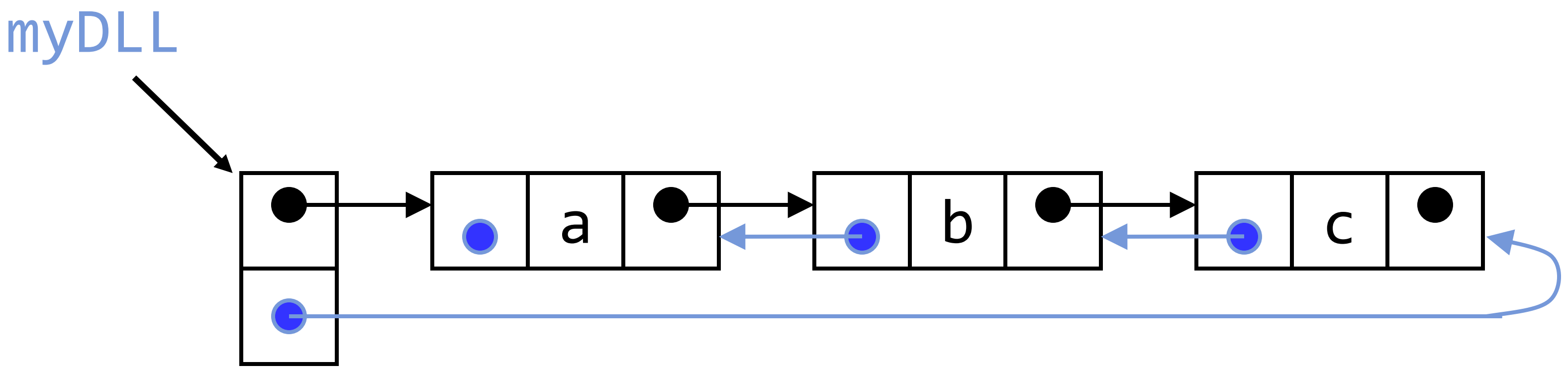
- To delete some other element, change the link in its predecessor



- Deleted nodes will eventually be garbage collected

DOUBLY-LINKED LISTS

- Here is a doubly-linked list (DLL):



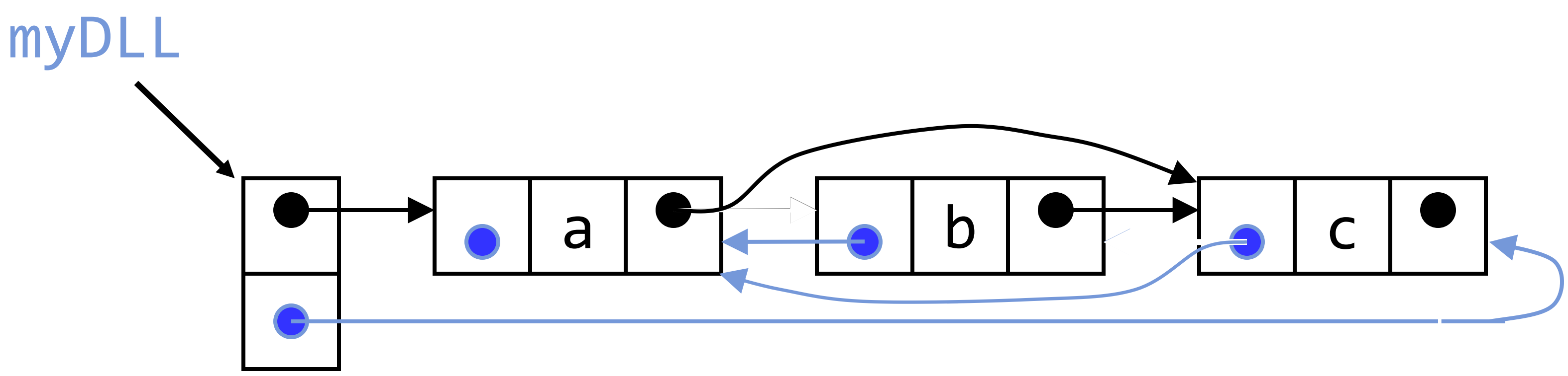
- Each node contains a value, a link to its successor (if any), *and* a link to its predecessor (if any)
- The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)

DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

DELETING A NODE FROM A DLL

- Node deletion from a DLL involves changing *two* links
- In this example,we will delete node b



- We don’t have to do anything about the links in node b
- Garbage collection will take care of deleted nodes
- Deletion of the first node or the last node is a special case

Other operations on linked lists

- Most "algorithms" on linked lists—such as insertion, deletion, and searching—are pretty obvious; you just need to be careful
- Sorting a linked list is just messy, since you can't directly access the n^{th} element—you have to count your way through a lot of other elements