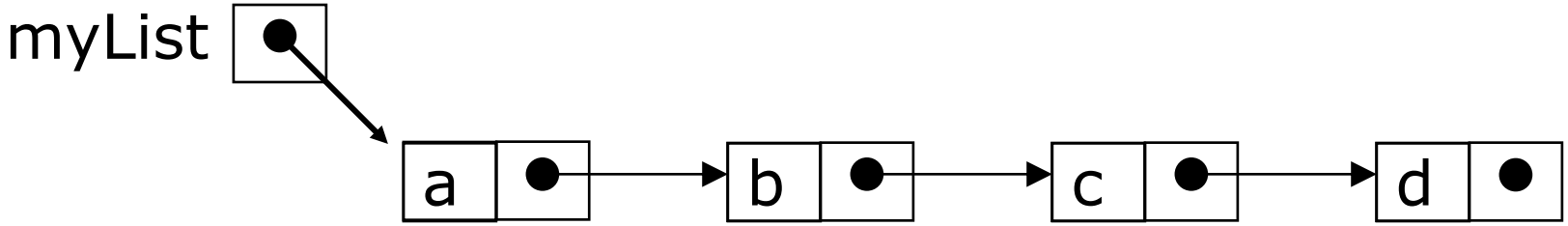
Chapter 6

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 have a header

Singly Linked Lists and Arrays

Singly linked list	Array
Elements are stored in linear order, accessible with links.	Elements are stored in linear order, accessible with an index.
Do not have a fixed size.	Have a fixed size.
Cannot access the previous element directly.	Can access the previous element easily.
No binary search.	Binary search.

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 (Abstract data type)

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**

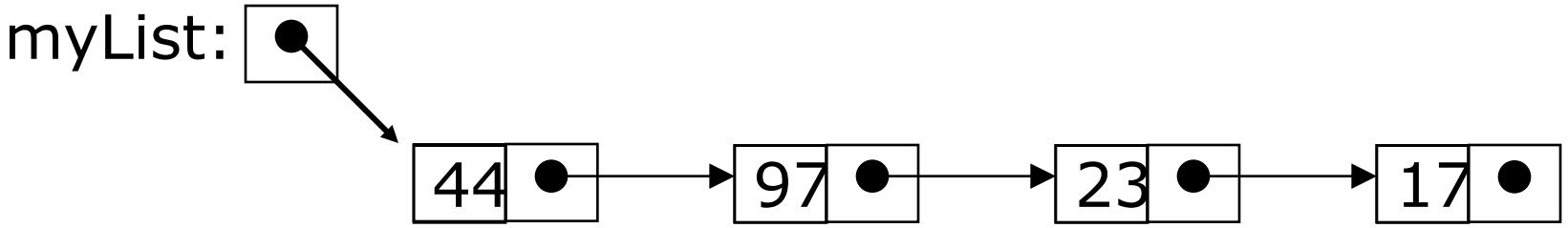
Creating references

- 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

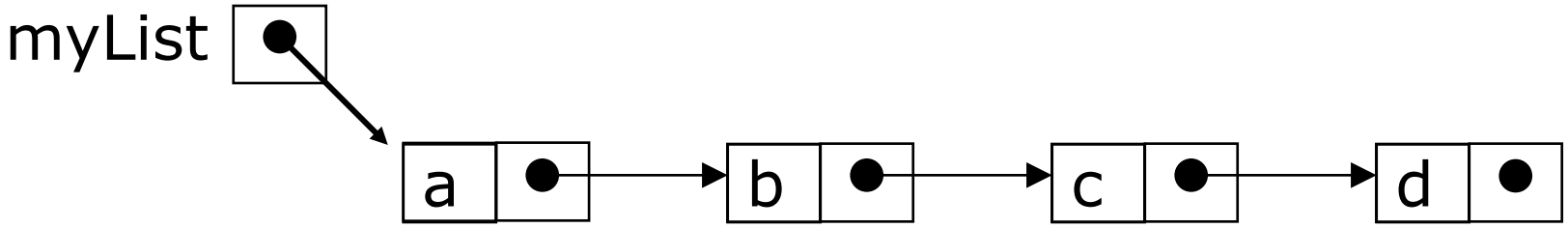
Creating links in Java




```
class Cell { int value;
             Cell next;
     Cell (int v, Cell n) { // constructor
        value = v;
        next = n;
     }
}
Cell temp = new Cell(17, null);
temp = new Cell(23, temp);
temp = new Cell(97, temp);
Cell myList = new Cell(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)

Singly-linked lists in Java

```
public class SLL {

    private SLLNode first;

    public SLL() {
        this.first = null;
    }

    // methods...
}
```

- This class actually describes the *header* of a singly-linked list
- However, the entire list is accessible from this header
- Users can think of the SLL as *being* the list
  - Users shouldn't have to worry about the actual implementation

SLL nodes in Java

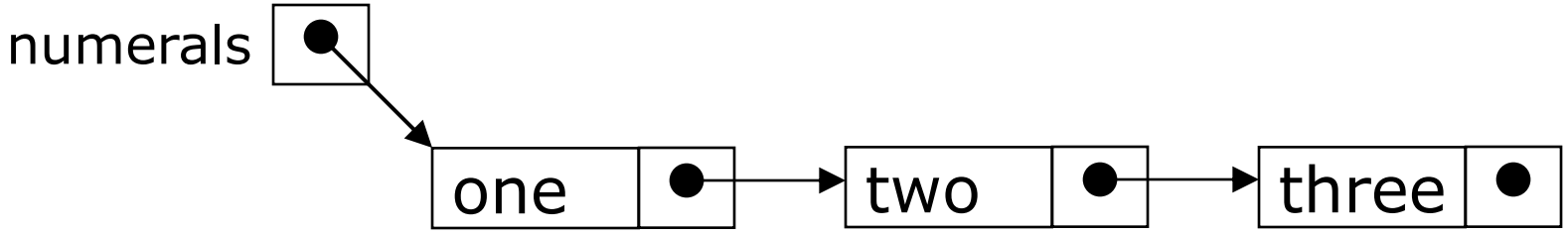
```
public class SLLNode {
    protected Object element;
    protected SLLNode succ;

    protected SLLNode(Object elem,
                      SLLNode succ) {
        this.element = elem;
        this.succ = succ;
    }
}
```

Creating a simple list

- To create the list ("one", "two", "three"):

```
SLL numerals = new SLL();
numerals.first =new SLLNode("one“,new SLLNode("two“,new SLLNode("three", null)));
```

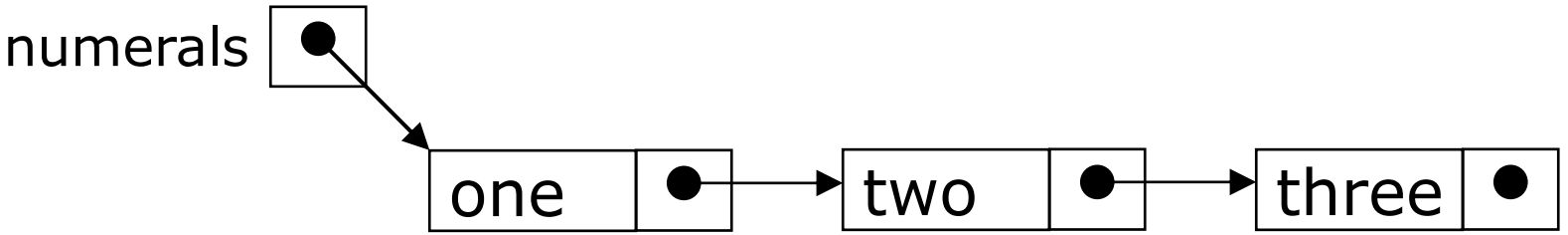


Traversing a SLL

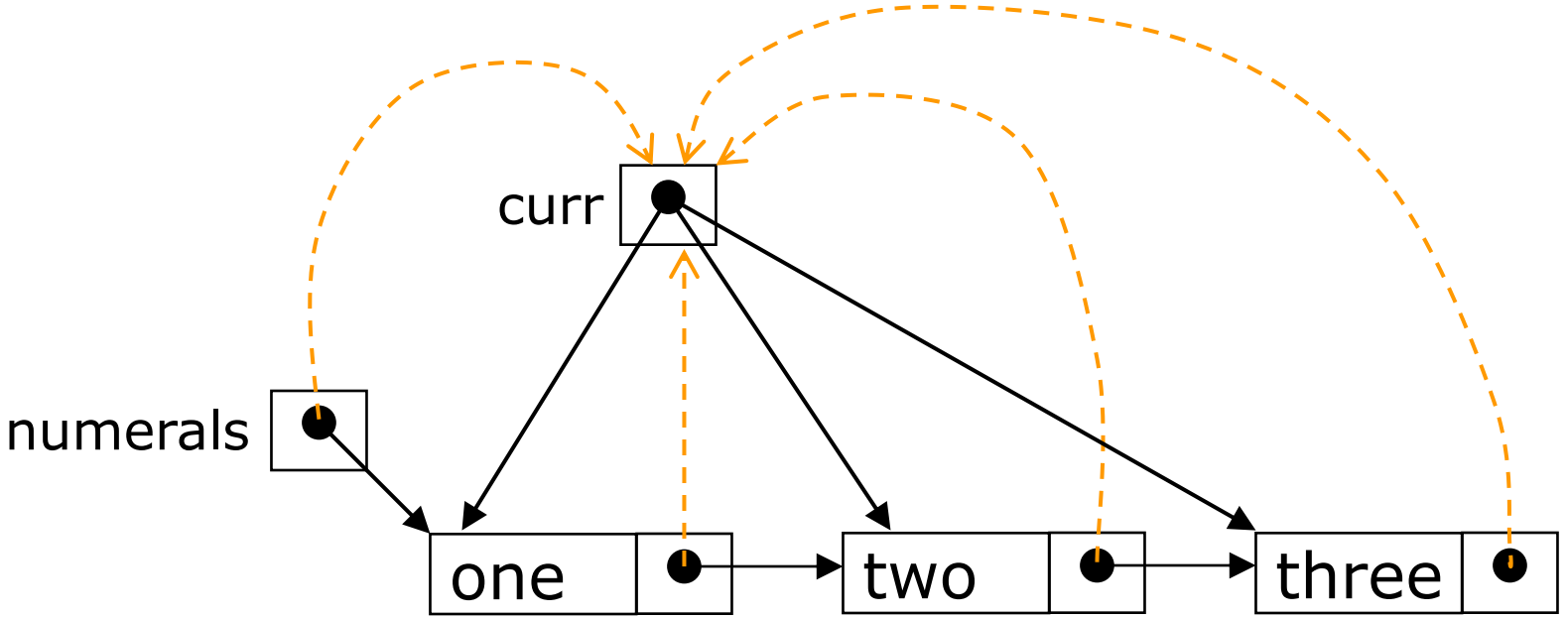
- The following method traverses a list (and prints its elements):

```
public void printFirstToLast() {
    for (SLLNode curr = first; curr != null; curr = curr.succ) {
      System.out.print(curr.element + "  ");
    }
}
```

- You would write this as an instance method of the SLL 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 SLL (not SLLNode):

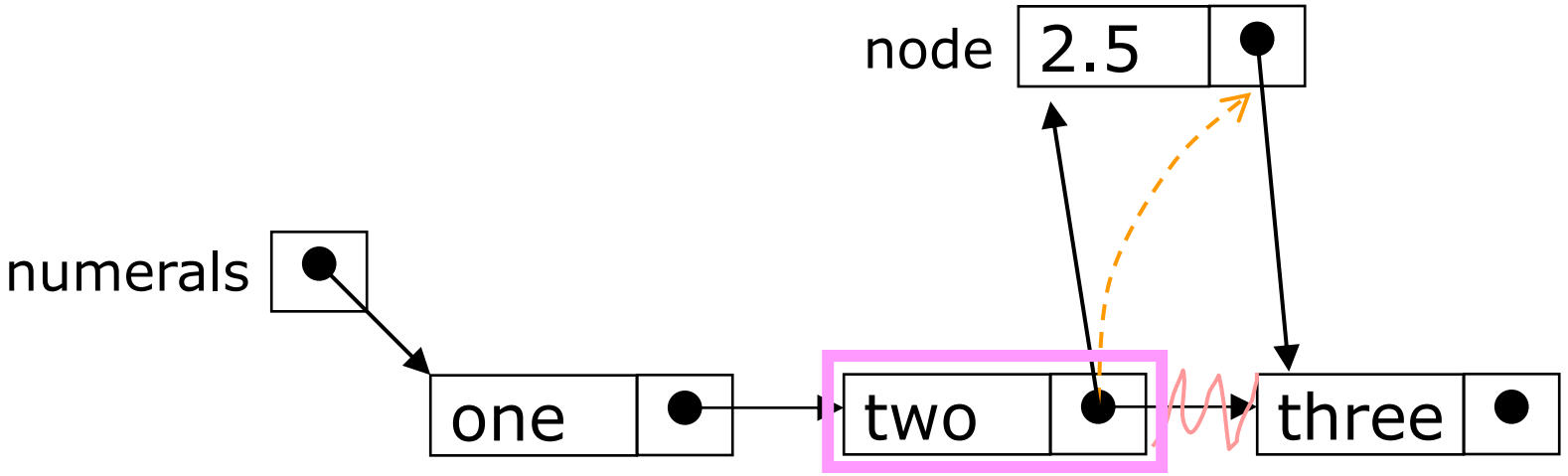
```
void insertAtFront(SLLNode node) {
        node.succ = this.first;
        this.first = node;
}
```

- Notice that this method works correctly when inserting into a previously empty list

Inserting a node after a given value

```
void insertAfter(Object obj, SLLNode node) {
    for (SLLNode here = this.first;
          here != null;
          here = here.succ) {
        if (here.element.equals(obj)) {
            node.succ = here.succ;
            here.succ = node;
            return;
        } // if
    } // for
    // Couldn't insert--do something reasonable!
}
```

Inserting after (animation)



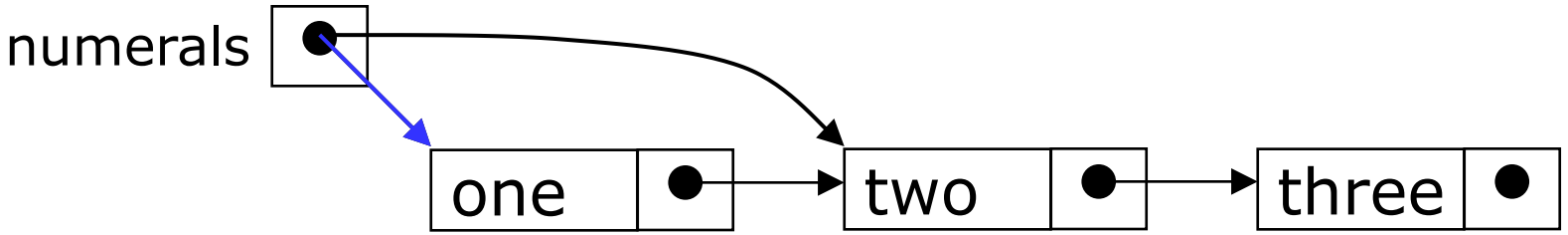
- Find the node you want to insert after
- ***First,*** copy the link from the node that's already in the list
- ***Then,*** change the link in the node that's already in the list

Deleting a node from a SLL

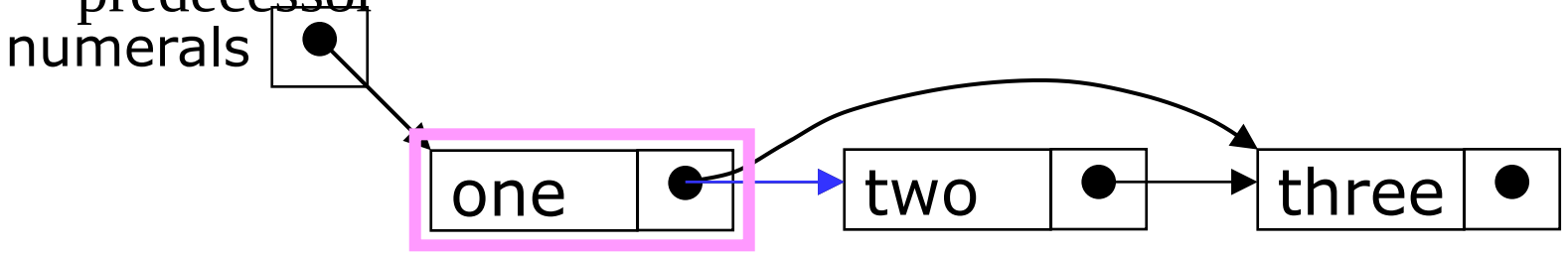
- In order to delete a node from a SLL, you have to change the link in its *predecessor*
- This is slightly tricky, because you can't follow a pointer backwards
- Deleting the first node in a list is a special case, because the node's predecessor is the list header

Deleting an element from a SLL

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



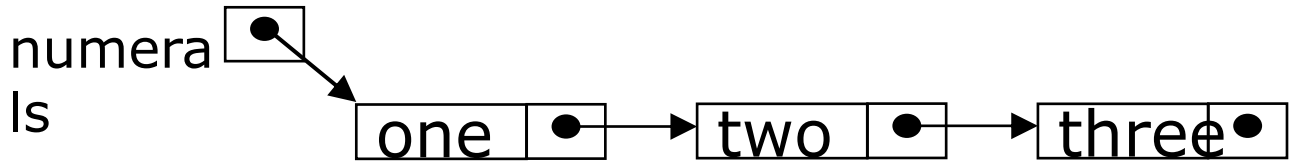
- To delete some other element, change the link in its predecessor



- Deleted nodes will eventually be garbage collected

Deleting from a SLL

```
public void delete(SLLNode del) {
    SLLNode succ = del.succ;
    // If del is first node, change link in header
    if (del == first) first = succ;
    else { // find predecessor and change its link
        SLLNode pred = first;
        while (pred.succ != del) pred = pred.succ;
        pred.succ = succ;
    }
}
```



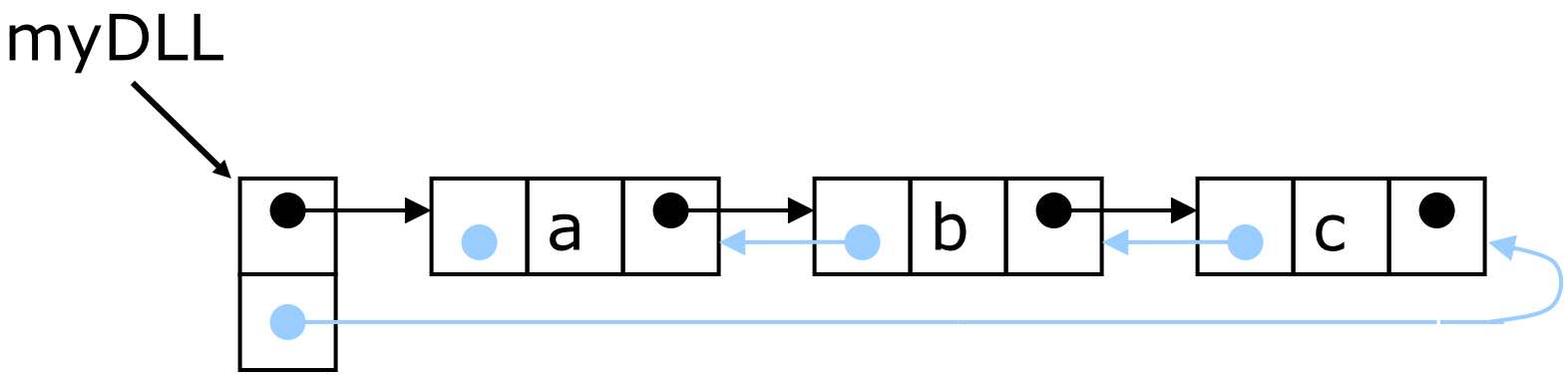
Limitations of a singly-linked list

- Insertion at the front is O(1)
- insertion at other positions is O(n)
- Removing a node requires a reference to the previous node
- We can traverse the list only in the forward direction

- How to overcome these limitations?
  - Double-linked list

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)

Constructing SLLs and DLLs

```
public class SLL {

    private SLLNode first;


    public SLL() {
        this.first = null;
    }


    // methods...
}
```

```
public class DLL {

    private DLLNode first;
    private DLLNode last;

    public DLL() {
        this.first = null;
        this.last = null;
    }

    // methods...
}
```

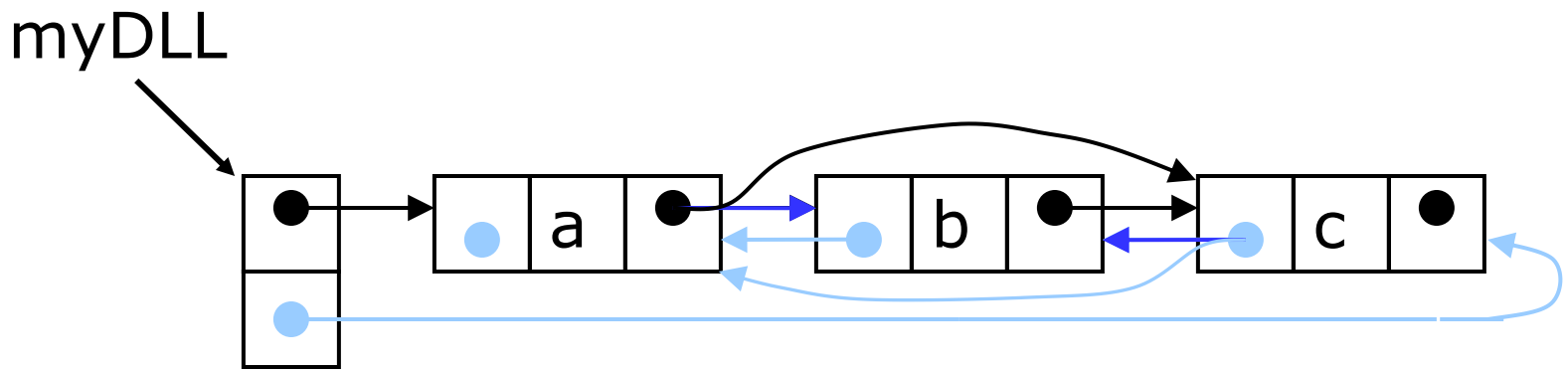
DLL nodes in Java

```
public class DLLNode {
    protected Object element;
    protected DLLNode pred, succ;

    protected DLLNode(Object elem,
                      DLLNode pred,
                      DLLNode succ) {
        this.element = elem;
        this.pred = pred;
        this.succ = succ;
    }
}
```

Deleting a node from a DLL

- Node deletion from a DLL involves changing *two* links



- Deletion of the first node or the last node is a special case
- Garbage collection will take care of deleted nodes

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

Circular Lists

- Circular double-linked list:
  - Link last node to the first node, and
  - Link first node to the last node
- We can also build singly-linked circular lists:
  - Traverse in forward direction only
- **Advantages:**
  - Continue to traverse even after passing the first or last node
  - Visit all elements from any starting point
  - Never fall off the end of a list
- **Disadvantage**: Code must avoid an infinite loop!

END.