Java For Non-Major CGS3416

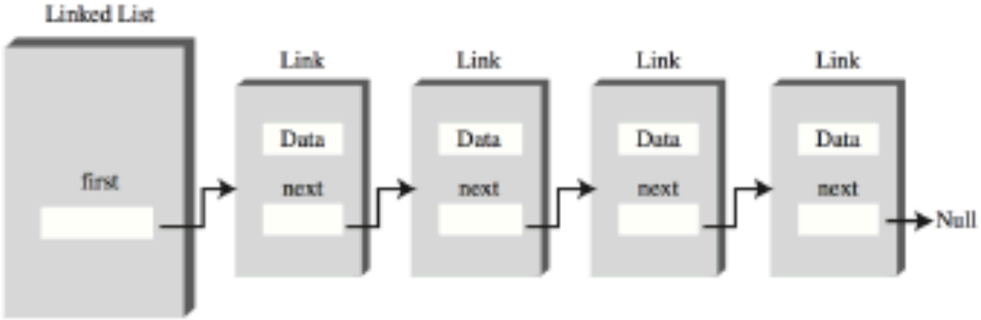
Lecture 16

LinkedList

This lecture was prepared based on the notes from David Fernandez-Baca and Steve Kautz of Iowa State University

Linked Lists

- Linked lists consist of linked nodes.
- Each node is a simple container, holding some piece of data, which has links(references) to one or more other nodes.
- There are many varieties of linked lists.
  - Forward links
  - Backward and forward links
  - Multiple successors
  - "dummy" nodes
  - Circular links
  - …



Singly-Linked Lists

Each node has a reference to the next node in the list.

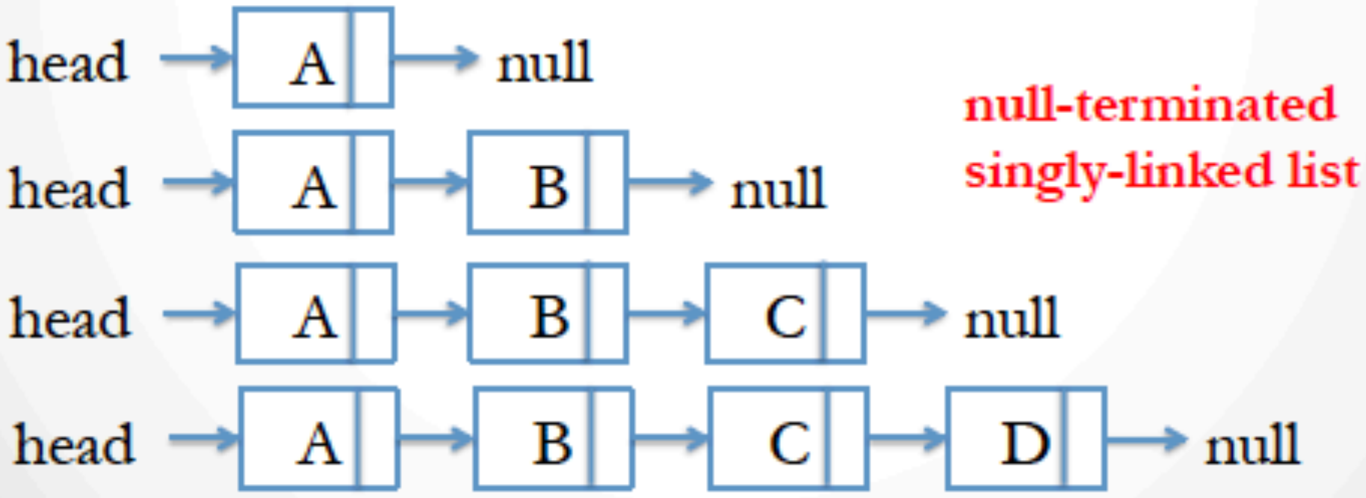
```
public class Node {
    public Object data;
    public Node next;
    public Node(Object data) { this.data = data; }
}

public class LinkList {
    private Node head;
    public LinkList() { head = null; }
    public boolean isEmpty() { return (head==null); }
    …
}
```

Singly-Linked Lists

We can build a list like this:

```
Node head = new Node("A");
head.next = new Node("B");
head.next.next = new Node("C");
head.next.next.next = new Node("D");
```



Access Elements in the List

We can access any element by starting at head:

```
System.out.println(head.data);
System.out.println(head.next.data);
System.out.println(head.next.next.data);
System.out.println(head.next.next.next.data);
```

We can also loop through the list using a temporary variable:

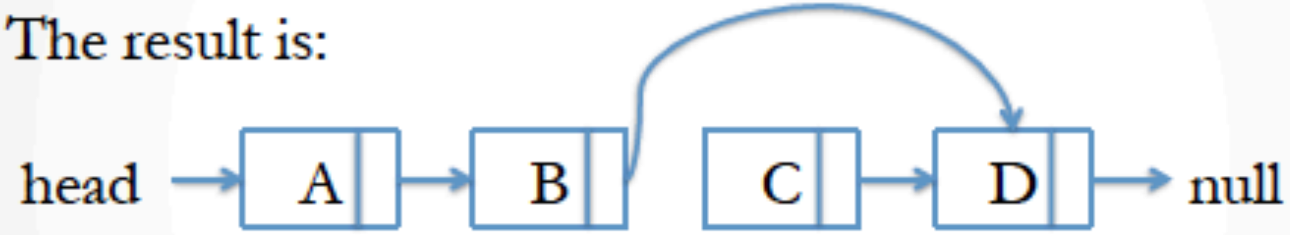
```
Node current = head;
while (current != null) {
    System.out.println(current.data);
    current = current.next;
}
```

Change Reference

Suppose we do:

```
head.next.next = head.next.next.next;
```

The result is:

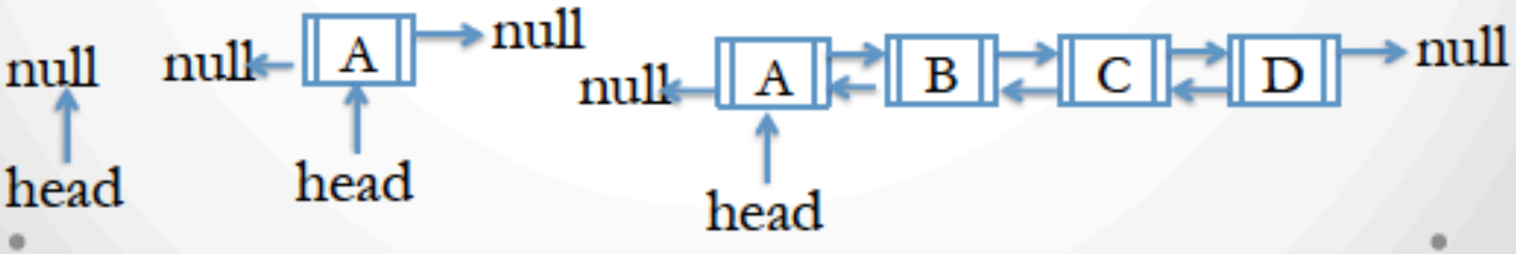


This effectively removes the node containing “C” from the list. Since C is no longer referenced, it becomes “garbage,” which is eventually reclaimed by Java’s garbage collector.

What happens if we do head = null?

Doubly-Linked Lists

- Limitation of singly-linked lists
  - cannot quickly access the *predecessor* of the current element
  - difficult to delete this element
  - can only iterate in one direction
- In **doubly-linked lists**, nodes have backward links as well as forward links.
- Cost: small amount of memory.



Practice 1: Using Doubly-Linked Lists to Implement the Collection Class

```
public class DoublyLinkedCollection<E> extends
                                AbstractCollection<E> {
    private Node head = null;
    private int size = 0;
    private class Node {
        public E data;
        public Node next;
        public Node previous;
        public Node(E data, Node next, Node previous) {
            this.data = data;
            this.next = next;
            this.previous = previous;
        }
    }
```

public boolean add(E item)

```
@Override
public boolean add(E item) {
   // add at beginning
   Node temp = new Node(item, head, null);
   // special case for empty or nonempty list
   if (head != null) { head.previous = temp; }
   head = temp;
   ++size;
   return true;
}
@Override
public int size() { return size; }
```

Since this is a collection, we don't need to worry about maintaining order. Thus, we put new elements at the beginning of the chain.

Iterator for DoublyLinkedCollection

- We implement iterators through an inner class called LinkedIterator. The iterator() method is then implemented as follows.

```
@Override
public Iterator<E> iterator() {
    return new LinkedIterator();
}
```

- Idea: use a Node variable to keep track of the next node to examine.

LinkedIterator

- To keep track of the next node to examine, an iterator will have a cursor field (of type Node) that runs through the list.
  - If the list is empty or there are no more elements, cursor is null.
  - Otherwise, cursor points to the next element to be returned by next().
  - Thus, the proper initial value for cursor is head.

Implementing LinkedIterator

```
private class LinkedIterator implements Iterator<E> {
    private Node cursor;
    public LinkedIterator() { cursor = head; }
    @Override
    public boolean hasNext() { return cursor != null; }
    @Override
    public E next() { //first attempt
        if (!hasNext()) throw new NoSuchElementException();
        E ret = cursor.data;
        cursor = cursor.next;
        return ret;
    }
```

Since LinkedIterator is an inner class within DoublyLinkedCollection, we can refer to the type variable E.

Implementing remove()

- To implement remove(), we need to maintain additional state information, so that an exception is raised if we invoke the method without previously calling next().
- It is not enough to keep a boolean canRemove state as we did for the array-based collection because we need to update links.
  - E.g., when we get to the end of the list, cursor is null.
- Thus, we maintain a Node variable pending that references the node whose removal is "pending".
  - pending is non-null, remove() will delete the node that pending refers to.
  - pending is null, we cannot do a remove().

next()

```
@Override
public E next() {
    if (!hasNext())
        throw new NoSuchElementException();
    pending = cursor;
    cursor = cursor.next;
    return pending.data;
}
```

remove()

```
@Override
public void remove() {
    if (pending == null) throw new IllegalStateException();
    // unlink pending node
    if (pending.previous != null) {
        pending.previous.next = pending.next;
    }
    if (pending.next != null) {
        pending.next.previous = pending.previous;
    }
    // if we're deleting the head, update head reference
    if (pending == head) { head = pending.next; }
    --size; pending = null;
}
}
```