Linked Lists II

The rest of the story
Adding/removing in the middle of a list, part deux

• Up to now, we have performed operations on nodes in the middle of a list with a crucial assumption: we have an external reference to a node preceding the space we want to insert a new node, or the node we want to remove

• We need a way to create such an external reference
One approach

• One way to tackle the problem involves using three of the methods we have previously examined: listLength, listSearch and listPosition:
  – listSearch finds a target value in a list
  – listLength returns the number of Nodes in a list
  – listPosition returns a reference to a particular Node

• We can use these methods, along with the assumption that the data in our list are kept in order, to come up with the code on the next slide
Implementation of findPrevious

```java
public static Node findPrevious (Node head, int target) {
    Node previous=null, cursor;
    if (head == null)
        return previous;
    for(cursor = head; cursor!=null && cursor.getData() < target;
        cursor = cursor.getLink());
    if(cursor == null)
        previous = Node.listLocate(head, listLength(head));
    else {
        int  shortLen = listLength(cursor);
        previous = listLocate(listLength(head) – shortLen);
    }
    return previous;
}
```
Copying a linked list

• You may have noticed that we didn’t choose to make the Node class implement Cloneable

• In terms of copying an entire list, this makes sense; again, a linked list is not a Node, but rather a collection of Nodes

• To make a copy of a list, we will employ another static method
How not to do it

Because we always have a reference to a head pointer, it’s tempting to try copying a list using just the insertion technique we already know:

```java
Node newList = null;
for (Node cursor = head; cursor != null; cursor = cursor.getLink())
    newList = new Node (cursor.getData(), newList);
```
A better way

• The code on the previous slide will traverse the original list (referenced by head) and copy all of the nodes, but they will end up in reverse of their original order

• A better way involves using a tail pointer for the new list, adding each new node after the last node instead of before the first one

• The listCopy method is presented on the next slide
Code for listCopy

```java
public static Node listCopy(Node source) {
    Node copyHead;
    Node copyTail;
    if (source == null)
        return null;
    copyHead = new Node(source.data, null);
    copyTail = copyHead;
    while (source.link != null) {
        source = source.link;
        copyTail.addNodeAfter(source.data);
        copyTail = copyTail.link;
    }
    return copyHead;
}
```
Copying a list with a tail reference

• As mentioned before, it is sometimes useful to keep track of both the head and tail of a linked list
• Unfortunately, Java doesn’t permit us to have more than one return value from a method – or does it?
• Not really, but kind of – we can return an array from the method, and that array can contain multiple values
public static Node[ ] listCopyWithTail(Node source) {
    Node copyHead;
    Node copyTail;
    Node[ ] answer = new Node[2];
    if (source == null)
        return answer;
    copyHead = new Node(source.data, null);
    copyTail = copyHead;
    while (source.link != null) {
        source = source.link;
        copyTail.addNodeAfter(source.data);
        copyTail = copyTail.link;
    }
    answer[0] = copyHead;
    answer[1] = copyTail;
    return answer;
}
Copying part of a linked list

• Sometimes we only need a portion of a linked list, rather than a copy of the entire list.

• The listPart method, shown on the next slide, handles this situation.

• The two parameters, start and end, are references to the beginning and end of the part of the list we wish to copy.
listPart method

```java
public static Node[ ] listPart(Node start, Node end){
    Node copyHead;
    Node copyTail;
    Node cursor;
    Node[ ] answer = new Node[2];
    copyHead = new Node(start.data, null);
    copyTail = copyHead;
    cursor = start;

    // continued on next slide
```
while (cursor != end)  {
    cursor = cursor.link;
    if (cursor == null)
        throw new IllegalArgumentException
            ("end node was not found on the list");
    copyTail.addNodeAfter(cursor.data);
    copyTail = copyTail.link;
}
answer[0] = copyHead;
answer[1] = copyTail;
return answer;
Bag ADT using linked list

• Invariant:
  – Items are stored in a linked list of Nodes; member variable head refers to the beginning of the list
  – Member variable manyNodes stores the current size of the Bag (in # of nodes)
List-based Bag vs. Array-based Bag

- In the list-based version of the Bag, most of the same methods are defined, but with different implementations.
- An exception is the ensureCapacity method; since a linked list can grow and shrink easily (without the need to make a temporary copy of a data array).
- For this implementation, a new method is added for the heck of it: the grab() method returns a random value from the Bag’s contents.
Bag member methods

• Default constructor: creates empty bag
• Code:
  
public Bag ( )
{
  head=null;
  manyNodes=0;
}

Clone() method

• Like the array-based bag, the linked list version implements the Cloneable interface, and thus must provide a clone() method
• Code shown on next slide
Clone method

```java
public Object clone( ) {
    Bag answer;
    try {
        answer = (Bag) super.clone( );
    }
    catch (CloneNotSupportedException e) {
        throw new RuntimeException
            ("This class does not implement Cloneable");
    }
    answer.head = Node.listCopy(head);
    return answer;
}
```
Remove method

• Strategy is similar to removal of target value from an array
• With array, we copied the last index’s data into the slot occupied by the target value, then decremented used
• With linked list, we’ll copy the head Node’s data value into the target Node, then remove the head Node
Code for remove method

```java
public boolean remove(int target) {
    Node targetNode;
    targetNode = Node.listSearch(head, target);
    if (targetNode == null)
        return false;
    else {
        targetNode.setData(head.getData());
        head = head.getLink();
        manyNodes--;
        return true;
    }
}
```
countOccurrences method

• Strategy: create local variable (cursor) that refers to a Node – use listSearch to find the target value, and assign its return value to cursor
• If cursor isn’t null, count one occurrence
• Continue process until cursor becomes null
Code for countOccurrences

public int countOccurrences(int target) {
    int answer = 0;
    Node cursor;
    cursor = Node.listSearch(head, target);
    while (cursor != null) {
        answer++;
        cursor = cursor.getLink();
        cursor = Node.listSearch(cursor, target);
    }
    return answer;
}
Add methods

• Since Bag is unordered ADT, follow path of least resistance and insert all items at front of list
• We follow this procedure for add(), addMany(), and addAll() (next several slides)
public void add(int element)
{
    head = new Node(element, head);
    manyNodes++;
}
addMany method

```java
public void addMany(int... elements) {
    for (int i : elements)
        add(i);
}
```
public void addAll(Bag addend) {
    Node[ ] copyInfo;
    if (addend.manyNodes > 0) {
        copyInfo = Node.listCopyWithTail(addend.head);
        copyInfo[1].setLink(head);
        head = copyInfo[0];
        manyNodes += addend.manyNodes;
    }
}
The union method

• Creates a new Bag that is the union of two other bags
• Method is static so that it can be applied to any two Bag objects (and has no effect on either one)
Code for union

public static Bag union(Bag b1, Bag b2) {
    Bag answer = new Bag();
    answer.addAll(b1);
    answer.addAll(b2);
    return answer;
}
Grab method

• New function for Bag: grab pulls a random item out of the Bag for inspection

• Curiously, it doesn’t actually remove the item -- if you wanted to add this functionality, you’d have to add a call to the remove method
public int grab() {
    int i;
    Node cursor;
    if (manyNodes == 0)
        throw new IllegalStateException("Bag size is zero");
    i = (int)(Math.random() * manyNodes) + 1;
    cursor = Node.listPosition(head, i);
    return cursor.getData();
}
But wait, there’s more! Doubly-linked lists

- Singly-linked lists have one major limitation: you can only traverse the list in one direction

- A variation on the linked list: linking each node to its previous neighbor as well as its next neighbor makes bi-directional list traversal possible
Modified class definition for doubly-linked list

```java
public class Node {
    private int data;
    private Node link; // link to next Node
    private Node back; // link to previous Node
    ...
}
```
Methods that must be modified

- constructor
- addNodeAfter
- remove
Constructors for doubly-linked list

```java
public Node(int d, Node b, Node n) {
    data = d;
    back = b;
    link = n;
}

public Node() {
    this(0, null, null);
}
```
addNodeAfter method

• Strategy is similar to singly-linked addNodeAfter, but with the additional complication that the node has two links
• Important thing to remember: make links before you break them!
Code for add method

public void addNodeAfter (Node previous, int entry) 
{
    Node n = new Node(entry, previous, null);
    n.link = previous.link;
    if (previous.link != null)
        previous.link.back = n;
    previous.link = n;
}
Linked Lists II

The rest of the story