CSC 143 Java

Linked Lists

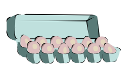
Reading: Ch. 20

Review: List Implementations

- The external interface is already defined
- Implementation goal: implement methods "efficiently"
- ArrayList approach: use an array with extra space internally
- ArrayList efficiency
  - Iterating, indexing (get & set) is fast
    - Typically a one-liner
  - Adding at end is fast, except when we have to grow
  - Adding or removing in the middle is slow: requires sliding all later elements

A Different Strategy: Linked Lists

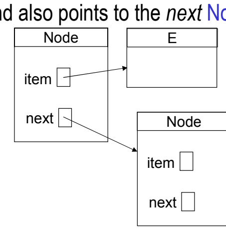
Instead of packing all elements together in an array,



create a *linked chain* of all the elements

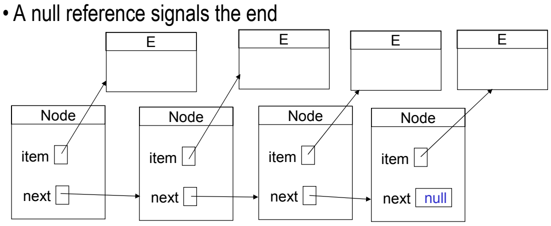
Nodes

- For each element in the list, create a Node object
- Each Node points to the ***data item*** (element) at that position, and also points to the *next* Node in the chain



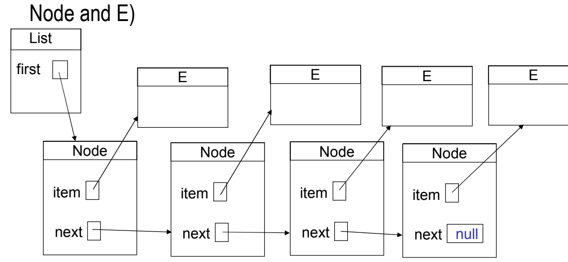
Linked Nodes

- Each node knows where to find the next node
- No limit on how many can be linked!
- A null reference signals the end



Linked List

- The **List** has a reference to the first **Node**
- Altogether, the list involves 3 different object types (List, Node and E)



Node Class: Data

```
/** Node for a simple list (defined within the LinkedList class who knows about
the E type) */
public class Node {
    public E item;       // data associated with this node
    public Node next;    // next Node, or null if no next node
    //no more instance variables
    //but maybe some methods
} //end Node
```

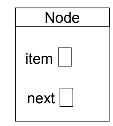
Note 1: This class does NOT represent the list, only one node of a list

Note 2: “public” violates normal practice – will discuss other ways later

Note 3: The nodes are NOT part of the data. The data is totally unaware that it is part of a list.

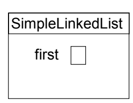
Node Constructor

```
/** Node for a simple list */
public class Node {
    public E item;      // data associated with this node
    public Node next;   // next node, or null if none

    /** Construct new node with given data item and next node (or null if none) */
    public Node( E item, Node next) {
        this.item = item;
        this.next = next;
    }
    …
}
```



LinkedList Data

```
/** Simple version of LinkedList for CSE143 lecture example */
public class SimpleLinkedList<E> implements List<E> {
    // instance variables
    private Node first;   // first node in the list, or null if list is empty
    …

}
```



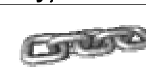
LinkedList Data & Constructor

```
/** Simple version of LinkedList for CSE143 lecture example */
public class SimpleLinkedList<E> implements List<E> {
    // instance variables
    private Node first;   // first node in the list, or null if list is empty
    …

    // construct new empty list
    public SimpleLinkedList( ) {
        this.first = null;          // no nodes yet (statement is not needed since
                                    // null is the default initialization value)
    }

    …
```


List Interface (review)

- Operations to implement:
  - **int size( )**
  - **boolean isEmpty( )**
  - **boolean add(E o)**
  - **boolean addAll(Collection<E> other)**
  - **void clear( )**
  - **E get(int pos)**
  - **boolean set(int pos, E o)**
  - **int indexOf(Object o)**
  - **boolean contains(Object o)**
  - **E remove(int pos)**
  - **boolean remove(Object o)**
  - **boolean add(int pos, E o)**
  - **Iterator iterator( )**
- **What don't we see anywhere here?? (No nodes anywhere)**

Method *add* (First Try)



```
public boolean add( E o) {
    // create new node and place at end of list:
    Node newNode = new Node(o, null);
    // find last node in existing chain: it's the one whose next node is null:
    Node p = this.first;
    while (p.next  != null) {
        p = p.next;
    }
    // found last node; now add the new node after it:
    p.next = newNode;
    return true;   // we changed the list => return true
}
```

Draw the Official CSE143 Picture

- Client code:

```
SimpleLinkedList<Point2D> vertices = new SimpleLinkedList<Point2D>();
Point2D p1 = new Point2D.Double(100.0, 50.0);
Point2D p2 = new Point2D.Double( 250, 310);
Point2D p3 = new Point2D.Double(90, 350.0);
vertices.add(p1);
vertices.add(p2);
vertices.add(p3);
vertices.add(p1);
```


Problems with naïve *add* method

- Inefficient: requires traversal of entire list to get to the end
  - One loop iteration per link
  - Gets slower as list gets longer
  - Solution??
- Buggy: fails when adding first link to an empty list
  - Check the code: where does it fail?
  - Solution??

Improvements to naïve *add* method

- Inefficient: requires traversal of entire list to get to the end
  - A solution:

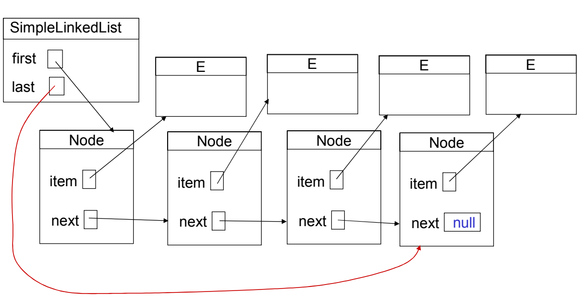
    Remove the constraint that instance variables are fixed.

    Change LinkedList to keep a pointer to *last* node as well as the *first*
- Buggy: fails when adding first link to an empty list
  - A solution: check for this case and execute special code
- Q: “Couldn’t we ....?” Answer: “probably”. There are many ways linked lists could be implemented

List Data & Constructor (revised)

```
public class SimpleLinkedList<E> implements List<E> {
    // instance variables
    private Node first;        // first link in the list, or null if list is empty
    private Node last;         // last link in the list, or null if list is empty
    …

    // construct new empty list
    public SimpleLinkedList( ) {
        this.first = null;     // no links yet
        this.last = null;      // no links yet
    }

    …
```


Linked List with last



Method *add* (Final Version)

```
public boolean add( E o) {
    // create new node to place at end of list:
    Node newNode = new Node(o, null);
    // check if adding the first node
    if (this.first == null) {
        // we're adding the first node
        this.first = newNode;
    } else {
        // we have some existing nodes; add the new node after the current last node
        this.last.next = newNode;
    }
    // update the last node
    this.last = newNode;
    return true;   // we changed the list => return true
}
```

Method *size( )*

- First try it with this restriction: you can't add or redefine instance variables
- Hint: count the number of links in the chain

```
/** Return size of this list */
public int size( ) {
    int count = 0;




    return count;
}
```


Method *size( )*

- Solution: count the number of links in the list

```
/** Return size of this list */
public int size( ) {
    int count = 0;
    for (E e : this) { // use the iterator
        count ++;
    }
    return count;
}
```

- Critique? Very slow!

Method *size* (revised)

- Add an instance variable to the list class

```
private int numNodes; // number of nodes in this list
```

- Add to constructor: numNodes = 0; (though not necessary)
- Add to method add: numNodes ++;

SimpleLinkedList
first ☐
last ☐
numNodes ☐

- Method size (new version)

```
/** Return size of this list */
public int size( ) {
    return numNodes;
}
```

- Critique? Don't forget to update numNodes whenever the list changes.

clear

- Simpler than with arrays or not?

```
/** Clear this list */
public void clear( ) {
    this.first = null;
    this.last = null;
    this.numNodes = 0;
}
```

- No need to "null out" the elements themselves
  - Garbage Collector will reclaim the Node objects automatically

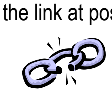
get

```
/** Return object at position pos of this list. 0 <= pos < size, else IndexOOBExn */
public E get( int pos) {
    if (pos < 0 || pos >= this.numNodes) {
        throw new IndexOutOfBoundsException( );
    }
    // search for pos'th link
    Node p = this.first;
    for ( int k = 0; k < pos; k++) {
        p = p.next;
    }
    // found it; now return the element in this link
    return p.item;
}
```

- Critique? Much slower than array implementation. Avoid linked lists if this happens a lot
- DO try this at home.

add and *remove* at given position

- Observation: to add a link at position k, we need to change the next pointer of the link at position k- 1

- Observation: to remove a link at position k, we need to change the next pointer of the link at position k 1


Helper for *add* and *remove*

- Possible helper method: get link given its position

```
// Return the node at position pos
// precondition (unchecked): 0 <= pos < size
private Node getNodeAtPos( int pos) {
    Node p = this.first;
    for ( int k = 0; k < pos; k++) {
        p = p.next;
    }
    return p;
}
```

  - Use this in get, too
  - How is this different from the get(pos) method of the List? It returns the Node and not the item.

remove(pos): Study at Home!

```
/** Remove the object at position pos from this list. 0 <= pos < size, else IndexOOBExn */
public E remove( int pos) {
    if (pos < 0 || pos >= this.numNodes) { throw new IndexOutOfBoundsException( ); }
    E removedElem;
    if (pos == 0) {
        removedElem = this.first.item;       // remember removed item, to return it
        this.first = this.first.next;        // remove first node
        if (this.first == null) { this.last = null; } // update last, if needed
    } else {
        Node prev = getNodeAtPos(pos-1);     // find node before one to remove
        removedElem = prev.next.item;        // remember removed item, to return it
        prev.next = prev.next.next;          // splice out node to remove
        if (prev.next == null) { this.last = prev; } // update last, if needed
    }
    this.numNodes --;    // remember to decrement the size!
    return removedElem;
}
```


add(pos): Study at Home!

```
/** Add object o at position pos in this list. 0 <= pos <= size, else IndexOOBExn */
public boolean add( int pos, E o) {
    if (pos < 0 || pos >= this.numNodes) { throw new IndexOutOfBoundsException( ); }
    if (pos == 0) {
        this.first = new Node(o, this.first);      // insert new link at the front of the chain
        if (this.last == null) { this.last = this.first; } // update last, if needed
    } else {
        Node prev = getNodeAtPos(pos-1);      // find link before one to insert
        prev.next = new Node(o, prev.next);   // splice in new link between prev & prev.next
        if (this.last == prev) { this.last = prev.next; }   // update last, if needed
    }
    this.numNodes ++;   // remember to increment the size!
    return true;
}
```


Implementing *iterator( )*

- To implement an iterator, could do the same thing as with SimpleArrayLists: return an instance of SimpleListIterator
- Recall: SimpleListIterator tracks the List and the position (index) of the next item to return
  - How efficient is this for LinkedLists?
  - Can we do better?

Summary

- SimpleLinkedList presents same illusion to its clients as SimpleArrayList
- Key implementation ideas:
  - a chain of links

- Different efficiency tradeoffs than SimpleArrayList
  - must search to find positions, but can easily insert & remove without growing or sliding
  - get, set a lot slower
  - add, remove faster (particularly at the front): no sliding required