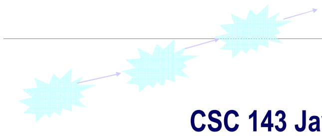


CSC 143 Java

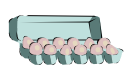
Linked Lists

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: Lists via Links

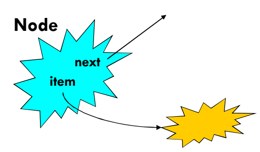
Instead of packing all elements together in an array,



create a *linked chain* of all the elements

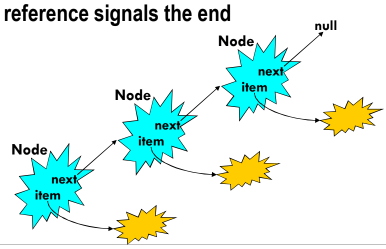
Nodes

- 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 (i.e. contains a link to the next Node in the list)



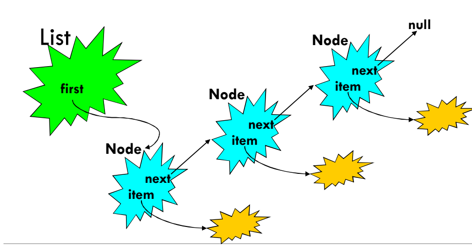
Linked Nodes

- Each Node points to the next
- 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



Node<E> Class: Data

```
/** Node for a simple list */
class Node<E> {
    E item;                    // data associated with this node
    Node<E> next;              // next Node, or null if no next node
    //no more instance variables but
    //maybe some methods, constructors
} //end Node
```

Note 1: This class does NOT represent the chain, only one link of a chain.

Note 2: This class could be an inner class in our SimpleLinkedList class, or it could be a package private class accessible by SimpleLinkedList.

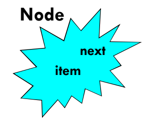
Note 3: Direct access of instance variables violates the normal rules, but is appropriate in this situation.

Note 4: The nodes are NOT part of the data. The data is totally unaware that it is part of a chain.

Node<E> Constructor

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

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



Exercise: Add a Node (1)

- Suppose we've got a linked list containing "lion", "tiger", and "bear" in that order, with a variable pointing to the head of the list

```
Node head;        // first node in the list, or null if list is empty
```

- Draw a picture of the list

Exercise: Add a Node (2)

- Now, write the code needed to insert "wolf" between "tiger" and "bear"

Exercise: Delete a Node (1)

- Suppose we've got a list containing "IBM", "Dell", "Compaq", and "Apple" in that order

```
Node head;        // first node in the list, or null if list is empty
```

- Draw a picture

Exercise: Delete a Node (2)

- Now, write the code needed to delete "Compaq" from the list

LinkedList Data

```
/** Simple version of LinkedList for CSC 143 lecture example */
public class SimpleLinkedList<E> implements List<E> {
    // instance variables
    private Node<E> first;        // first node in the list, or null if list is empty
    …

}
```

LinkedList Data & Constructor

```
/** Simple version of LinkedList for CSC 143 lecture example */
public class SimpleLinkedList<E> implements List<E> {
    // instance variables
    private Node<E> first;        // first node in the list, or null if list is empty
    …

    /** construct new empty list */
    public SimpleLinkedList( ) {
        first = null;             // no nodes yet!
    }

    …
```

List<E> Interface (review)

- Operations to implement:

```
int size( )
boolean isEmpty( )
boolean add( E o )
boolean addAll( Collection<? extends 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 )
void add( int pos, E o )
Iterator<E> iterator( )
```

- What isn't included here?? Notice: No Nodes!!

Method *add* (First Try)

```
public boolean add( E o ) {
    // create new node and place at end of list:
    Node<E> newNode = new Node<E>( o, null );
    // find last node in existing chain: it's the one whose next node is null:
    Node<E> p = 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 Picture

- Client code:

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


Problems with naïve *add* method

- Inefficient: requires traversal of entire list to get to the end
  - One loop iteration per node
  - Gets slower as list gets longer
  - Solution??
- Buggy: fails when adding first node 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: Change LinkedList to keep a pointer to *last* node as well as the *first*
- Buggy: fails when adding first node 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

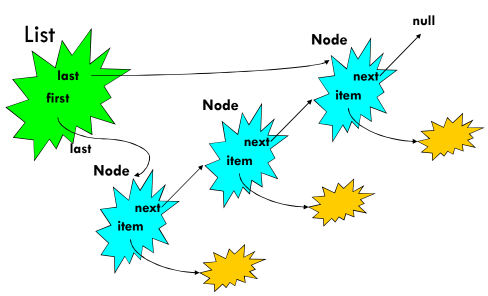
List Data & Constructor (revised)

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

    /** construct new empty list */
    public SimpleLinkedList( ) {
        first = null;           // no nodes yet!
        last = null;            // no nodes yet!
    }

    …
```


Linked List with last



Method *add* (Final Version)

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

```
public int size( ) {
    int count = 0;
            Node<E> p = first;
            while ( p  !=  null ) {
                p = p.next;
                count++;
            }
    return count;
}
```

- Critique? Slow…

Method *size* (faster)

- Add an instance variable to the list class

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

- Add to constructor:

```
numNodes = 0;
```

- Add to method *add*:

```
numNodes ++;
```

- Method *size*

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

- Critique? Good, but don't forget that every method that changes the number of items on the list needs to update numNodes. Need a class invariant!

Method clear( )

- Simpler than with arrays or not?

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

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

    (Some GCs might reclaim the objects quicker if we did null out the nodes, but good ones shouldn't need this)

Method get( )

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

- Critique?
- DO try this at home. Try "set" too

add and *remove* at given position

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


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


Helper for *add* and *remove*

- Possible helper method: get node given its position

```
// Return the node at position pos
// precondition (unchecked): 0 <= pos < size
private Node<E> getNodeAtPos( int pos ) {
    Node<E> p = 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?

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 >= numNodes ) { throw new IndexOutOfBoundsException( ); }
    E removedElem;
    if ( pos == 0 ) {
        removedElem = first.item;          // remember removed item, to return it
        first = first.next;                // remove first node
        if ( first == null ) { last = null; }   // update last, if needed
    } else {
        Node<E> 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 ) { last = prev; }  // update last, if needed
    }
    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 void add( int pos, E o ) {
    if ( pos < 0 || pos > numNodes ) { throw new IndexOutOfBoundsException( ); }
    if ( pos == 0 ) {
        first = new Node<E>( o, first );  // insert new node at the front of the chain
        if ( last == null ) { last = first; } // update last, if needed
    } else {
        Node<E> prev = getNodeAtPos( pos – 1 );  // find node before one to insert
        prev.next = new Node<E>( o, prev.next );  // splice new node btn prev & prev.next
        if ( last == prev ) { last = prev.next; }   // update last, if needed
    }
    numNodes++;       // remember to increment the size!
}
```

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 linked nodes

- Different efficiency trade-offs 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

Postscript 1: Recursion and Linked Lists

- A linked list is a recursive data structure!
- Example: printing a linked list (why are these methods private?):
  - Forward:

```
private void forwardPrint( Node<E> p ) {
    if ( p == null ) return;
    System.out.println( p.item );   // print the current item, …
    forwardPrint( p.next );         // then print the rest of the list.
}
```

  - Reverse (references link only forward – can we do this?):

```
private void reversePrint( Node<E> p ) {
    if ( p == null ) return;
    reversePrint( p.next );         // print the rest of the list, …
    System.out.println( p.item );   // then print the current item.
}
```


Recursion vs. Iteration

- When to use recursion:
  - Processing recursive data structures
  - "Divide & Conquer" algorithms:
    1. Divide problem into sub-problems.
    2. Solve each sub-problem recursively
    3. Combine the sub-problem solutions
- When to use iteration instead:
  - Non-recursive data structures
  - Problems without obvious recursive solution
  - Problems with obvious iterative solution
  - Methods with a large "footprint"
    especially when many iterations are needed

Postscript 2: Linked List Variations

- Several variations to the linked list structure are possible…
  - Circular Linked List:
    - Nodes are linked in a circular structure.
    - Any node could be the "first".
    - No node has a null link.
  - Lists with Dummy Head Nodes:
    - Eliminates the need to test for special cases in add and remove.
  - Doubly Linked Lists:
    - Each node links to 'previous' node in addition to 'next' node.
  - Combinations…