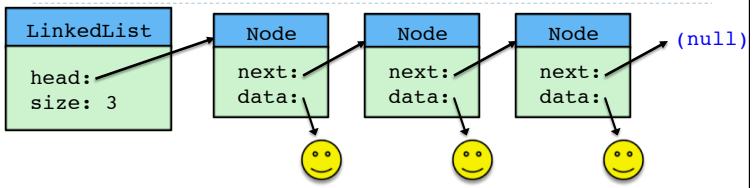
Class #11: Linked Lists

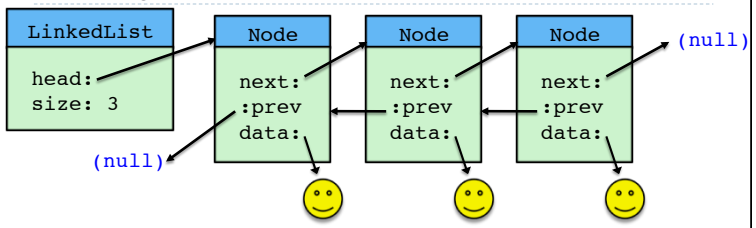
Software Design III (CS 340): M. Allen, 17 Feb. 16

The Linked List ADT



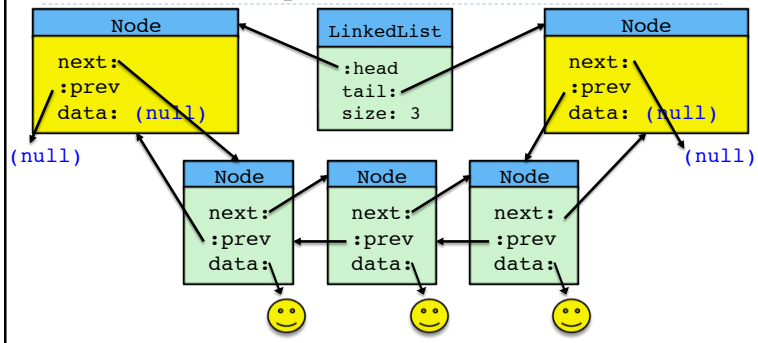
- General idea: an implementation of `List` interface/ADT, using a set of ***linked nodes***, with `size()` == # nodes
- Each node two basic attributes:
  - ***Data***: object that it stores (simple or complex)
  - ***Next link***: a variable reference to the next node in list
  - End of list links to `null` object (can be used as a marker)

Doubly Linked Lists



- Actual `java.util.LinkedList` class is slightly more complex, since each node has two separate links
  - A ***bi-directional, doubly-linked*** list
- Nodes link to both the next and previous nodes in list order
  - Front: `prev` links to `null` object
  - Back: `next` links to `null` object

One Last Complication: Sentinel Nodes



- For convenience, implementations will use one/more sentinel nodes
  - These track the start (`head`) or end (`tail`) of the list (and have ***no data***)
  - Can make code easier to write, but don't generally change basic behavior

LinkedList Nodes

```
private static class Node<T>
{
   private T data;
   private Node<T> prev;
   private Node<T> next;

   public Node( T d, Node<T> p, Node<T> n )
   {
      data = d;
      prev = p;
      next = n;
   }
}
```

- Basic data component of list is a nested, private static class
  - Packaged inside list class itself
  - private to that class, so it cannot be created anywhere else by itself
  - It carries the type of the list class instance, <T>

- Making the class static separates it from the parent class, so it has no requirement for a linked instance of the parent to exist
  - This simplifies things, and can improve memory management
  - This is a good approach, so long as the nested class ***doesn't need direct access*** to other non-static members (variables, methods) of the containing class
  - Containing List class ***does have*** direct access to Node,elements, whether Node is static or not

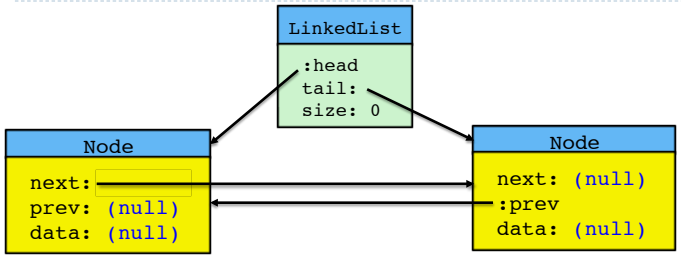
Creating the LinkedList

```
public class MyLinkedList<T> implements Iterable<T>
{
    private int theSize;
    private Node<T> head;
    private Node<T> tail;

    public MyLinkedList()
    {
        clear();
    }

    public void clear()
    {
        head = new Node<T>( null, null, null );
        tail = new Node<T>( null, head, null );
        head.next = tail;

        theSize = 0;
    }

}
```

Use head/tail nodes to indicate start/end of list

Can access private next and prev link variables in Node ***directly***, since it is a nested member class, contained entirely inside the List class itself.

Creating the LinkedList




```
head = new Node<T>( null, null, null );
tail = new Node<T>( null, head, null );
head.next = tail;
```


LinkedList Iterators

This private inner class is ***not*** static

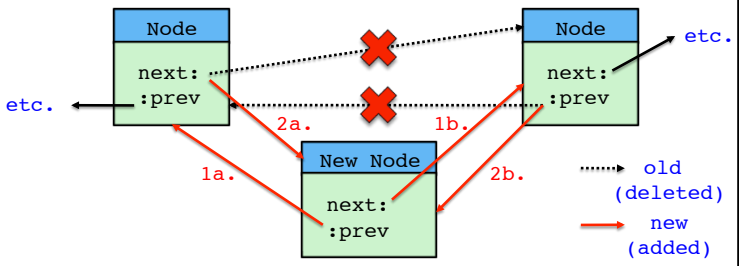
It can easily ***directly access*** any other elements of the List class it needs.

```
private class LinkedListIterator implements java.util.Iterator<T>
{
   private Node<T> current = head.next;

   public boolean hasNext() {
      return current != tail;
   }

   public T next() {
      if( !hasNext() )
         throw new java.util.NoSuchElementException();

      T nextItem = current.data;
      current = current.next;
      return nextItem;
   }

   public void remove() {
      MyLinkedList.this.remove( current.prev );
   }
}
```

Use the head/tail nodes as indicators of the start/end of the list

Use the next/prev variable references to navigate from node to node in the list.

Adding Nodes to Linked Lists



- With array-based lists, adding/removing involves shifting objects around to open/close space in array, leading to ***linear*** O(n) complexity
- With linked lists, we can add nodes by simply creating the new node, then adding/deleting links appropriately (a ***constant-time*** O(1) process)
  - ***Note: Order is important!*** We can lose data if we delete links to a node before adding a new one to keep track of it

Adding Nodes to Linked Lists

```
public boolean add( T x )
{
   add( theSize, x );
   return true;
}

public void add( int idx, T x )
{
   addBefore( getNode( idx, 0, theSize ), x );
}

private void addBefore( Node<T> p, T x )
{
   Node<T> newNode = new Node<T>( x, p.prev, p );
   newNode.prev.next = newNode;
   p.prev = newNode;
   theSize++ ;
}
```

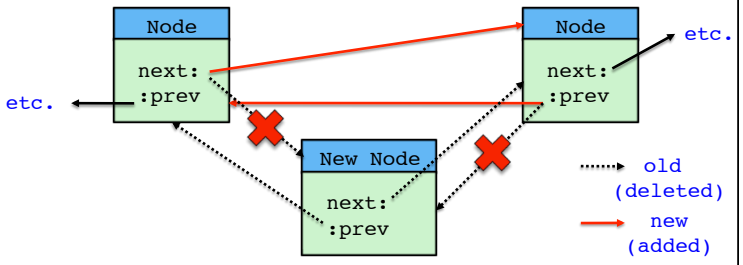
Basic `add( T )` is a ***convenience method***. It calls `add( int, T )`, using size of list as add-position.

Method `add( int, T )` uses another helper method to get the list-node that needs to be linked up with the new one we are adding.

(Helper throws `IndexOutOfBoundsException` if `idx < 0` or `idx > theSize`.)

The final helper method, `addBefore( Node<T>, T )`, actually ***creates*** the new node to store data, and links it to ***existing*** node. If we call basic `add( T )`, this results in new node being added ***just before the*** `tail`.

Deleting Nodes from Linked Lists



- Can also remove from linked structure in ***constant-time*** O(1)
- We ***first*** change over the links for the ***remaining*** list elements, and ***then*** delete the node and its own links

Deleting Nodes from Linked Lists

```
public T remove( int idx )
{
   return remove( getNode( idx ) );
}

private T remove( Node<T> p )
{
   p.next.prev = p.prev;
   p.prev.next = p.next;
   p.next = null;
   p.prev = null;
   theSize-- ;

   return p.data;
}
```

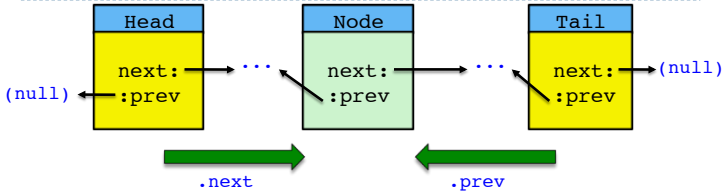
User calls `public remove( int )` and gives a position at which to remove data.

That method uses a helper to get the list node to remove, sends it to ***another*** helper.

The first helper, `getNode()`, throws an `IndexOutOfBoundsException` if `idx < 0` or `idx >= theSize`.)

Second helper, `remove( Node<T> )`, does the work of re-arranging the node links, adjusting the list-size, and returning removed data (in case needed at the original calling site).

Accessing Linked List Elements



- Unlike array-based lists, however, we ***cannot*** directly access a list element in constant time
- Instead, we must start from either the head or tail, and iterate inwards to the position we want in the list
- This gives worst-case (n / 2) operations == O(n)

Accessing Linked List Elements

```
private Node<T> getNode( int idx, int lower, int upper )
{
   Node<T> p;

  if ( idx < lower || idx > upper )
           throw new IndexOutOfBoundsException();

   if ( idx < theSize / 2 )
   {
      p = head.next;
      for ( int i = 0; i < idx; i++ )
         p = p.next;
   }
   else
   {
      p = tail;
      for ( int i = theSize; i > idx; i-- )
         p = p.prev;
   }

   return p;
}
```

When we need to access a node at a (valid) position `idx`, we loop inwards from the ***nearest end*** to that node, forward from `head`, or backward from the `tail`.

Cuts expected practical search time in half. However, it means worst-case performance is still: theSize / 2 = O(theSize)

LinkedList Complexity

Method	Time Complexity
T get(int i)	O(n)
int size()	O(1)
T set(int i, T element)	O(n)
boolean add(T element)	O(1)
void add(int i, T element)	O(n)
T remove()	O(1)
T remove(int i)	O(n)
int indexOf(T element)	O(n)
void clear()	O(1)

Add/Remove to head or tail: ***constant*** time (fast)

Add/Remove to other locations: ***linear*** time (slow)

Comparing LinkedList & ArrayList

Method	ArrayList	LinkedList
Get element at any index *i*	O(1)	O(n)
Set element at any index *i*	O(1)	O(n)
Add/Remove element at tail	O(1)	O(1)
Add/Remove element at head	O(n)	O(1)
Add element at any index *i*	O(n)	O(n)
Remove element at any index *i*	O(n)	O(n)
Get index of given element *e*	O(n)	O(n)

This Week

- **Topic:** Linear Structures
- **Read**: Text, chapter 03
- **In Lab:** Friday, 19 Feb.
- **Homework 02:** due Wednesday, 24 Feb. (5:00 PM)
- **Office Hours:** Wing 210
  - Tuesday & Thursday: 10:00–11:30 AM
  - Tuesday & Thursday: 4:00–5:30 PM