COMP1002
Data Structures and Algorithms

Lecture 4: Linked Lists

Department of Computing

Last updated: August 21, 2017

Copyright Warning

This Week

- Linked Lists
  - Simple linked lists
- Variants
  - Double-ended
  - Doubly-linked
  - Sorted linked lists
- Time complexity analysis
  - Big-O notation

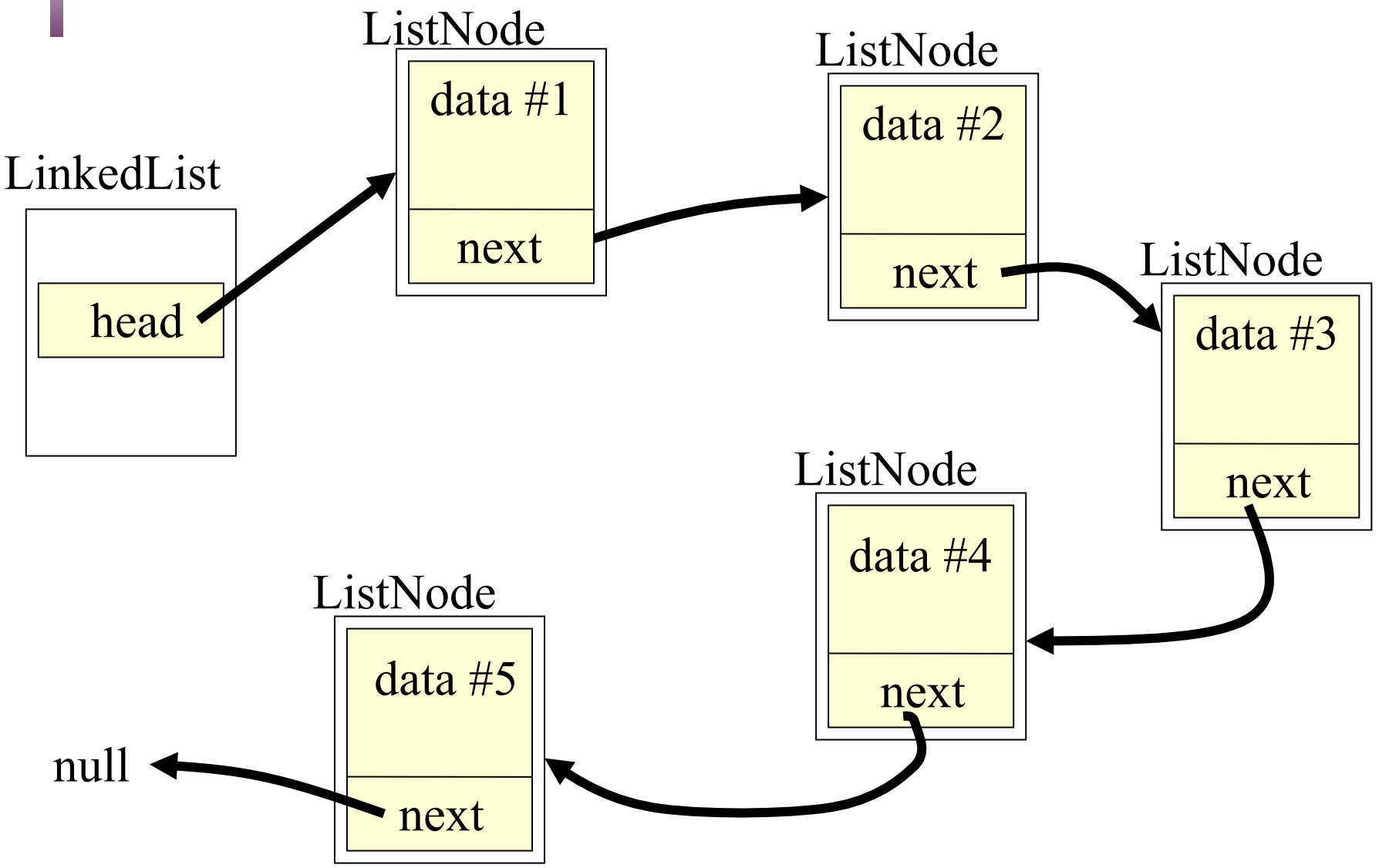
Arrays Aren't Everything

- As a data structure, arrays have their disadvantages
  - Fixed size – cannot grow/shrink to fit num elements
    - » If we don't know how many elements in advance, we'll need to overestimate the array's length, and thus waste space
  - Insertion – inserting an element between two existing elements will require shuffling later elements up by one
    - » Similar issues plague removal from the front/middle (*e.g.*, queues)
  - Contiguous – all elements must fit in a contiguous block
    - » This is sometimes a disadvantage – it's a bit inflexible
- We'd like a data structure that doesn't suffer from these
  - … but still does a similar job to arrays (storing list of data)
  - Fortunately, it's already been invented: the linked list

Linked Lists

- The idea behind a linked list is that each element stores a value *and* a reference to the *next* element
  - The list is essentially a chain of links
  - The list itself only stores a reference to the *first* element
    - » This is often called the 'head'
  - Since each element only points to the next, it is *singly-linked*
  - Since the list only points at the head, it is *single-ended*
- More complicated variants on the basic single-ended singly-linked list
  - Double-ended, doubly-linked, etc
  - We'll examine some of these variants later

Simple Linked List



Anatomy of a Linked List

- Each element is not just data but also a reference to the next element
  - Thus we need to create a composite of `value` + `next` ref
    - » Classes are ideally suited to this – make `value`, `next` class fields
      - – Typically called ListNode, or less commonly ListLink
    - » In pre-OO languages, use structs (C) or records (Pascal)
- Head is the only member field of LinkedList class
  - Points at the first ListNode in the list
- Last node's `next` points at null
  - Indicates the end of the linked list

ListNode Members

- ListNode only exists as a container for the data and the next pointer
- Thus it is really only a couple of member fields with associated getters/setters:
  - One member field for the data value, usually an Object so that the list can be general-purpose
  - A member `ListNode next` that is a reference to the next node in the list's chain
    - » This points at null if the ListNode is the last node in the chain

Traversing Linked Lists

- A linked list is just a chain of connected-but-independent nodes
  - Each node could be anywhere in memory
  - Only a node's predecessor knows where to find a given node (in memory)
  - So the only way to get to the fourth node is to first get to the third node
    - » which in turns requires us to have made it to the second node
    - » … and we can only get to that via the first node

Traversing Linked Lists

- Only the first node is available directly (via `head`)
  - Thus getting to a particular node requires that we traverse from the `head` (first) node, through all the `next` pointers until we reach the desired node
    - » This obviously can be pretty slow
  - Compare this to an array, which can get to *any* element in *one step*
    - » via a little bit of arithmetic – it works because the array elements are all stored in a contiguous block
    - » But linked lists aren't contiguous, so we have to do traversing
  - This is the main disadvantage of linked lists: access time

Linked List Methods

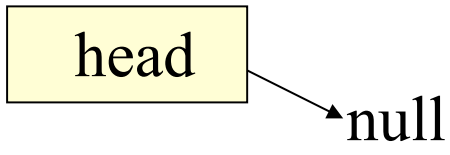
- isEmpty()
- insertFirst(), insertLast(), insertBefore(<dataType> valueToFind)
  - Insert a new item into the list
  - Should require a data item as import, NOT a ListNode
    - » ListNode is an internal detail of LinkedList's implementation
- removeFirst(), removeLast(), remove(<dataType> valueToFind)
  - Delete an item from the list
- peekFirst(), peekLast(), peek(<dataType> valueToFind)
  - Return the data value of an item in the list (not ListNode)
- find(<dataType> valueToFind)
  - Search whether a given data value exists in the list

Inserting a ListNode

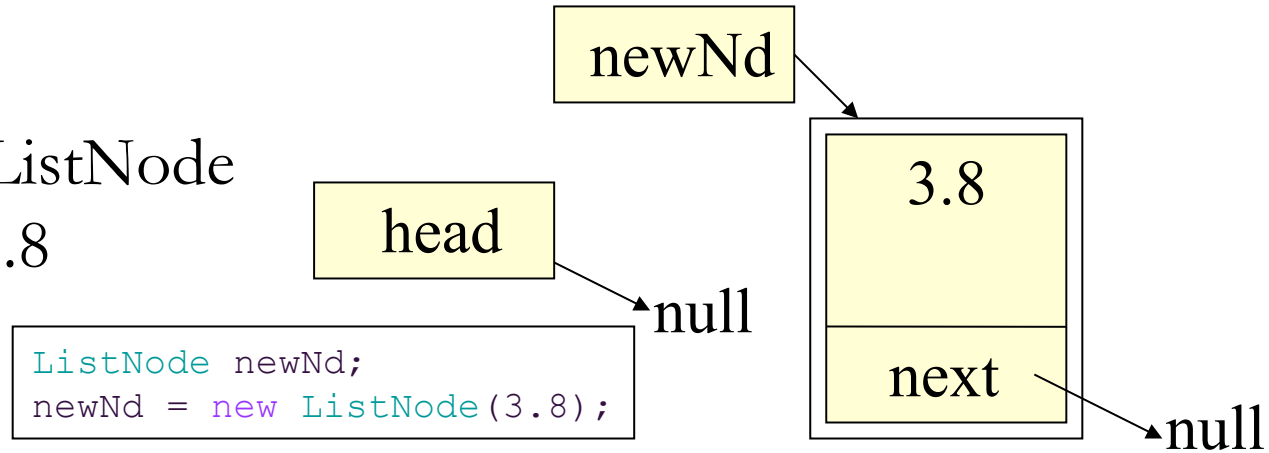
- There are four possible scenarios for node insertion:
  - The list is empty and we are inserting the first node
  - We are inserting before the head node
  - We are inserting after the tail of the list
  - We are inserting somewhere in the middle of the list
- In code, some of these cases turn out to be the same thing, but when designing a solution you should *always* be thinking through all the possibilities

Inserting First ListNode

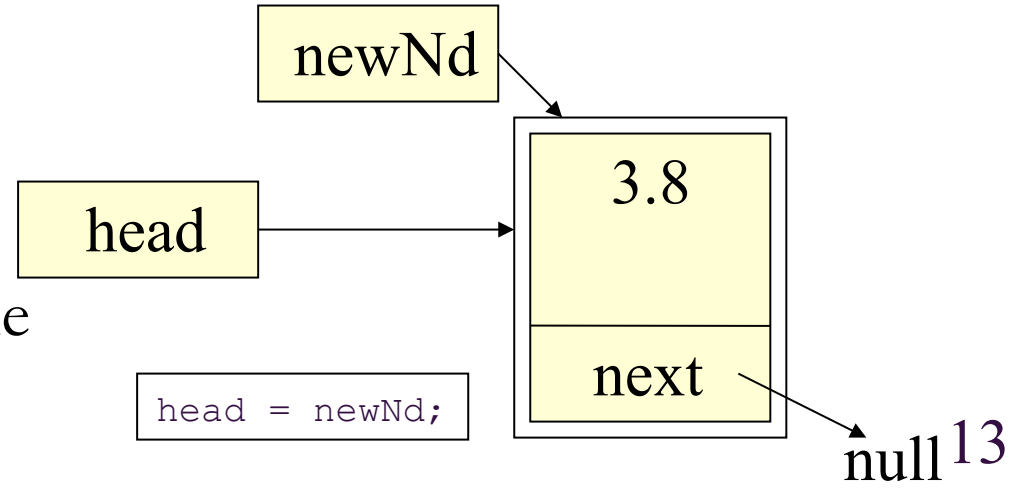
- Initial: empty list
  (*i.e.*, head points at null)



- Step 1: Create new ListNode with (say) value of 3.8

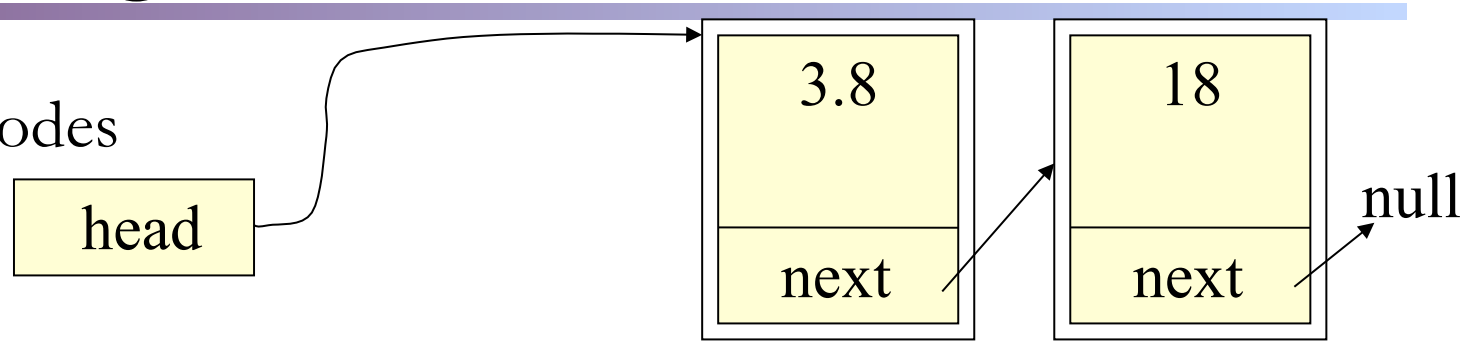


- Step 2: Point head at the newNd to make it the first node



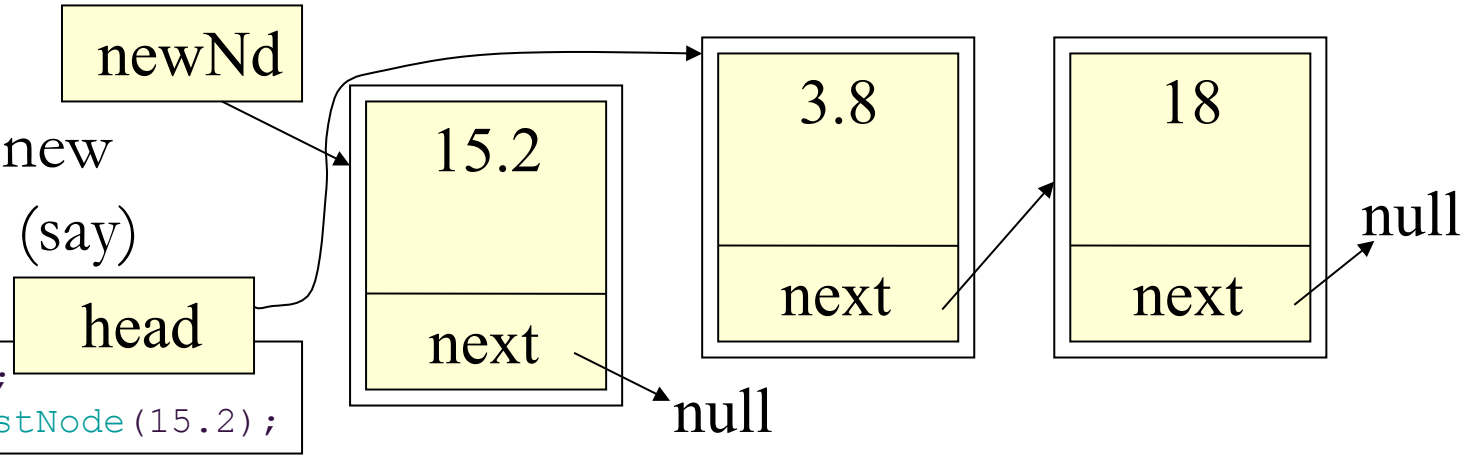
Inserting ListNode Before Head

- Initial: a few nodes in the list



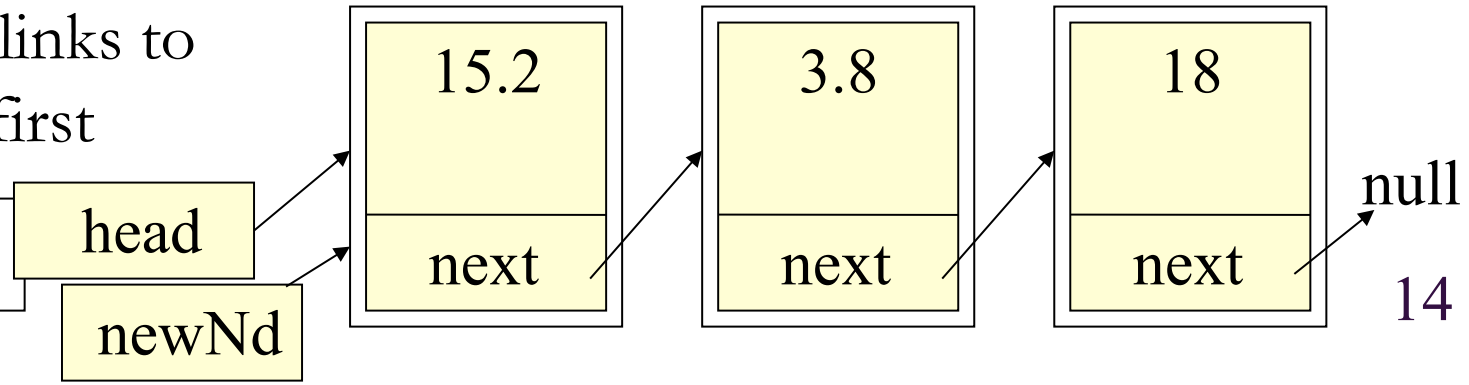
- Step 1: Create new ListNode with (say) value of 15.2

```
ListNode newNd;
newNd = new ListNode(15.2);
```



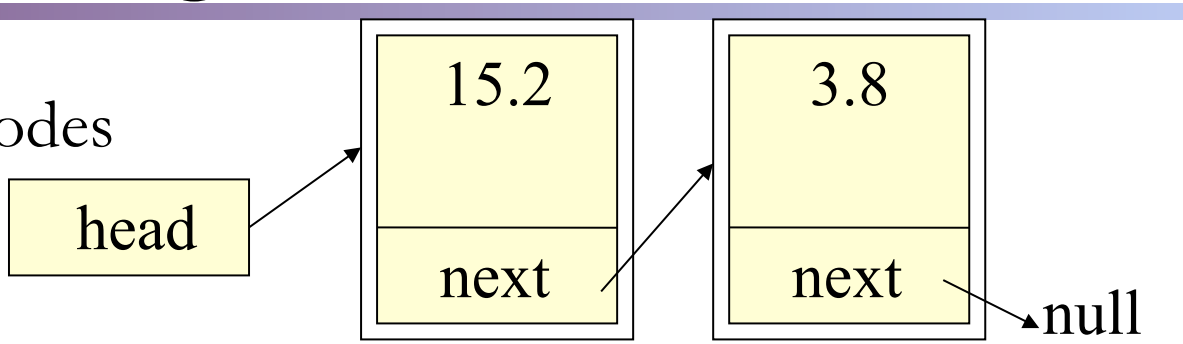
- Step 2&3: Fix links to make newNd first

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

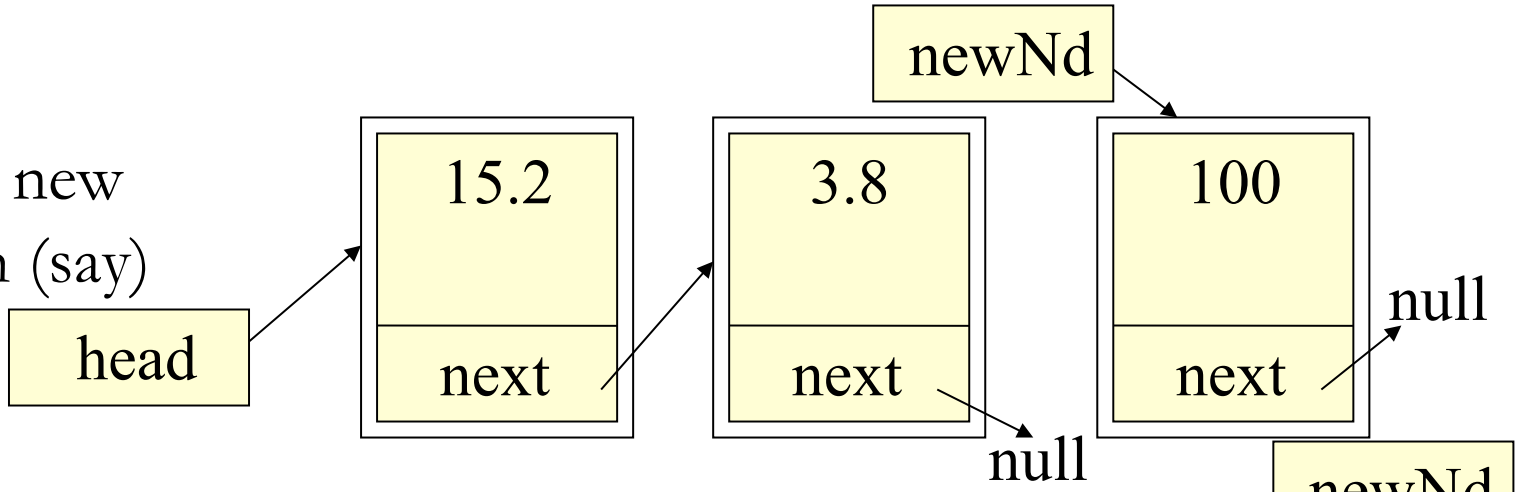


Inserting ListNode After Tail

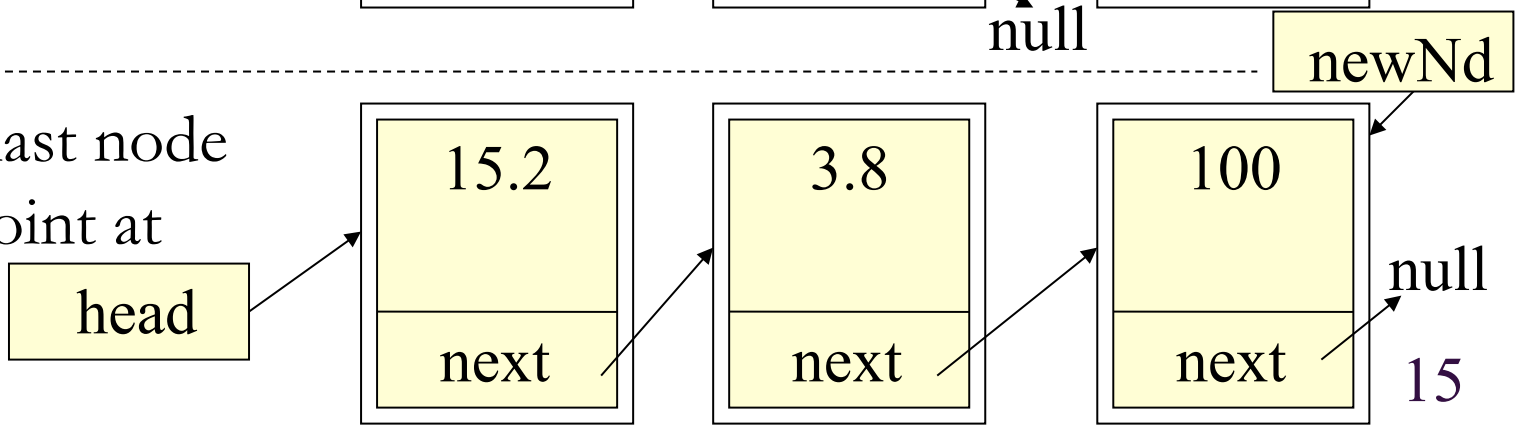
– Initial: a few nodes in the list



– Step 1: Create new ListNode with (say) value of 100

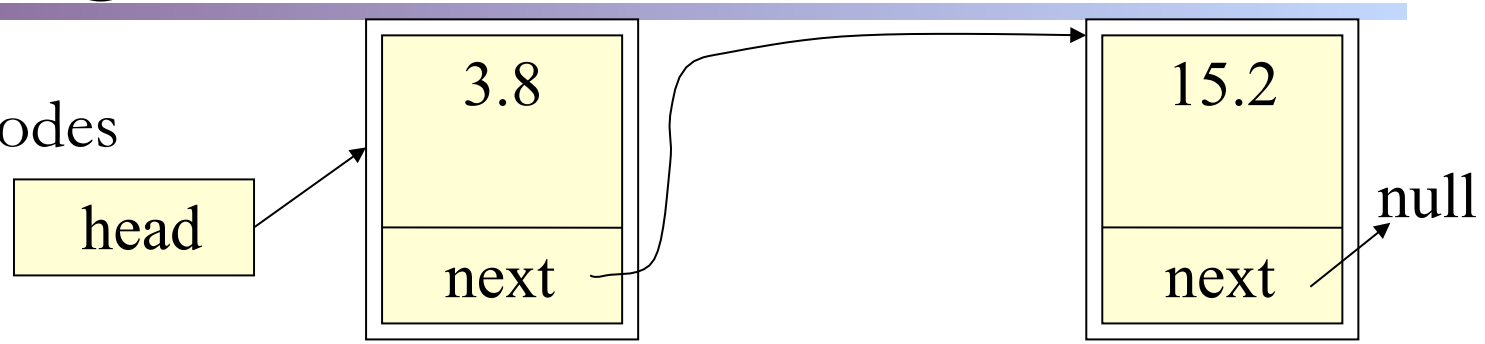


– Step 2: Make last node in the chain point at the newNd

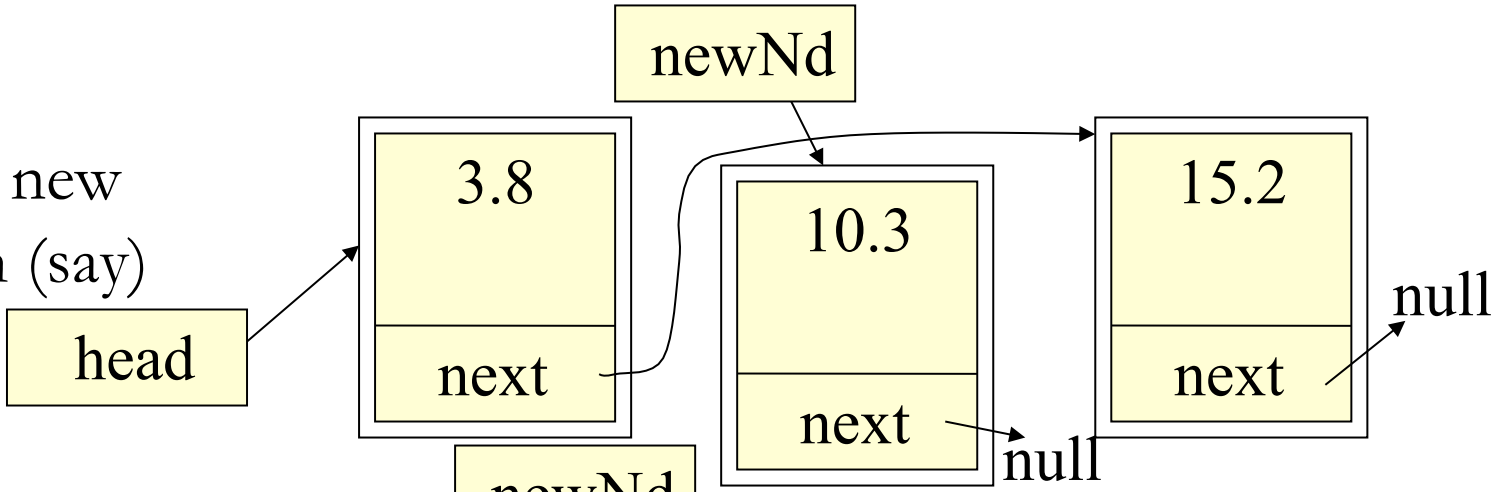


Inserting ListNode In The Middle

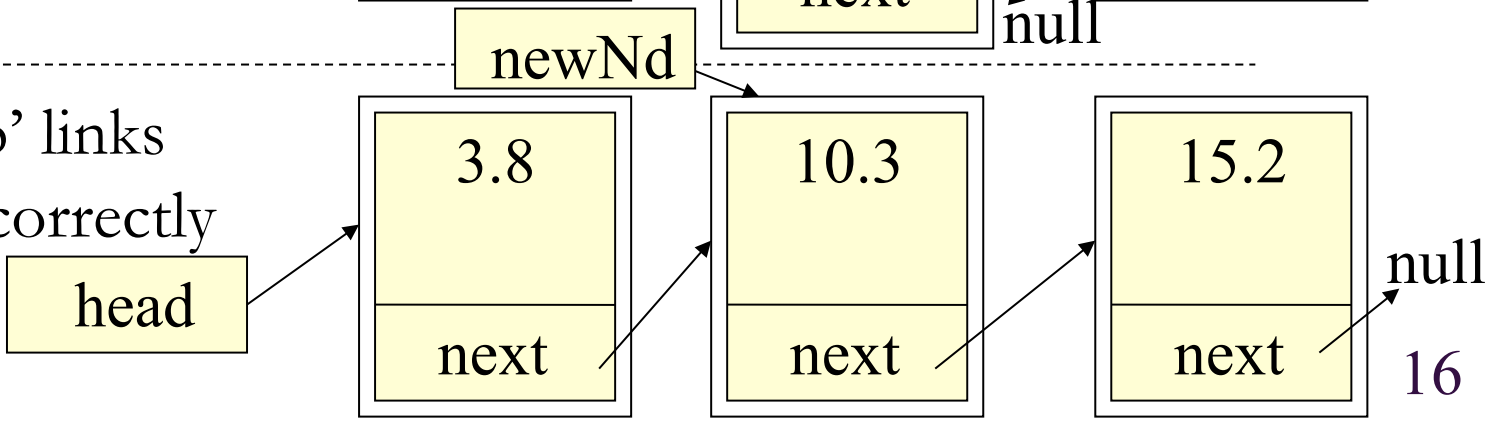
– Initial: a few nodes in the list



– Step 1: Create new ListNode with (say) value of 10.3



– Step 2: 'Fix up' links to make it fit correctly in the list

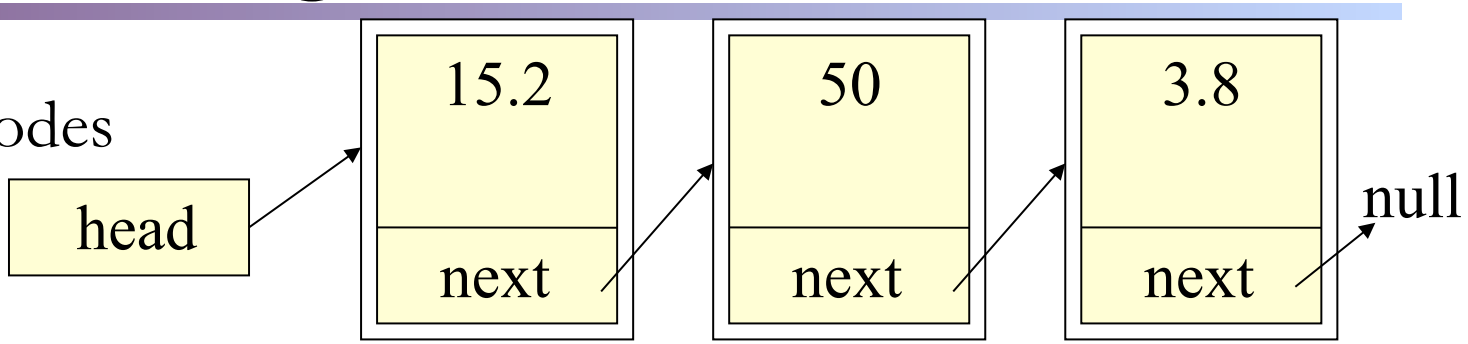


Removing a ListNode

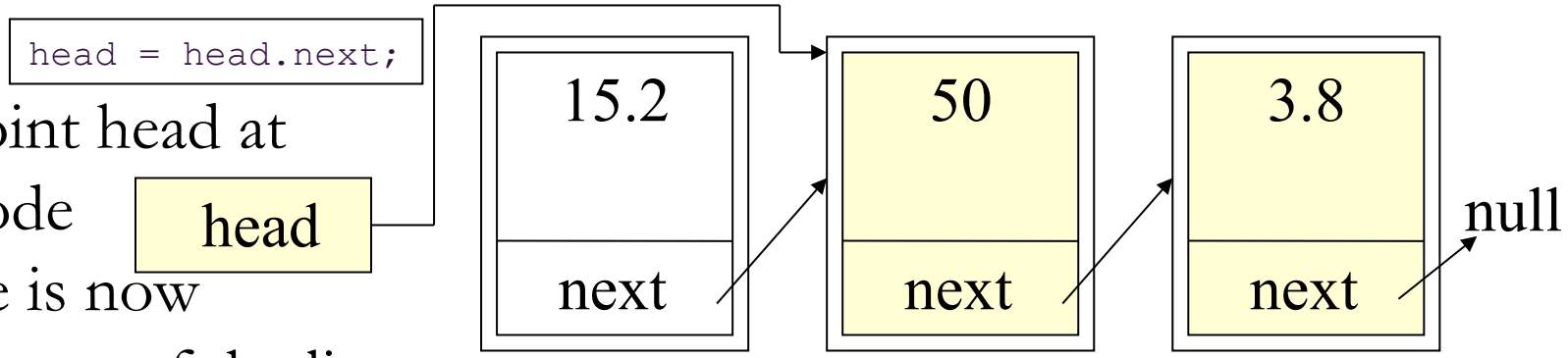
- As with insertion, there are four cases:
  - Removing the head ListNode
  - Removing the tail ListNode
  - Removing a ListNode in the middle of the list
  - Removing the sole remaining ListNode in the list
- And as with insertion, some of these cases end up being the same in the code

Removing Head ListNode

– Initial: a few nodes in the list



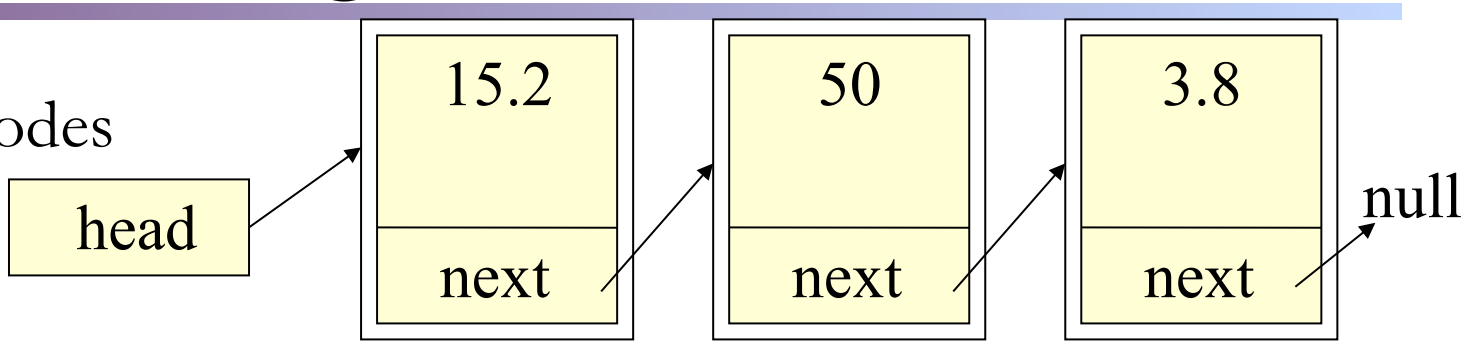
– Step 1: Point head at second node
First node is now no longer part of the list
and its memory usage will be garbage-collected by Java
(in C/C++, you would have to explicitly free it)



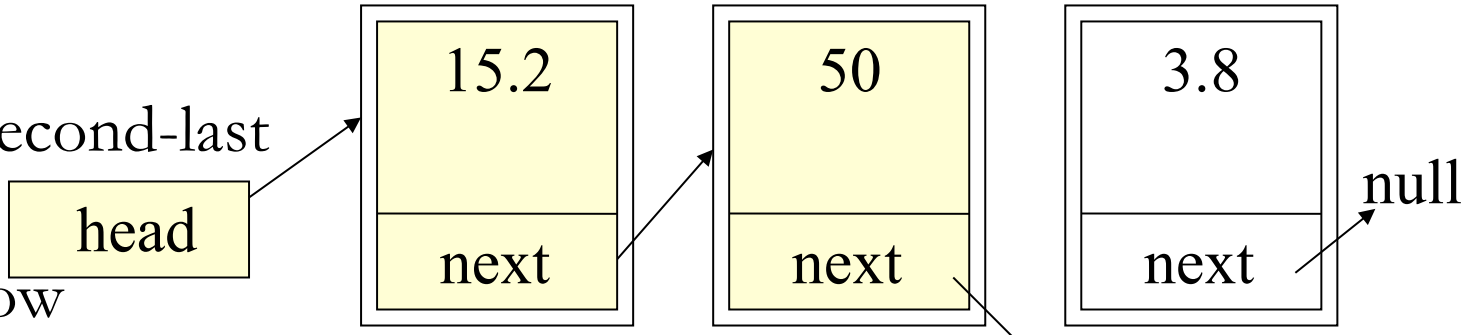
- The fact that Node 15.2 still points at the second node is not relevant: nothing points *at* Node 15.2, therefore it will be garbage-collected

Removing Tail ListNode

– Initial: a few nodes in the list



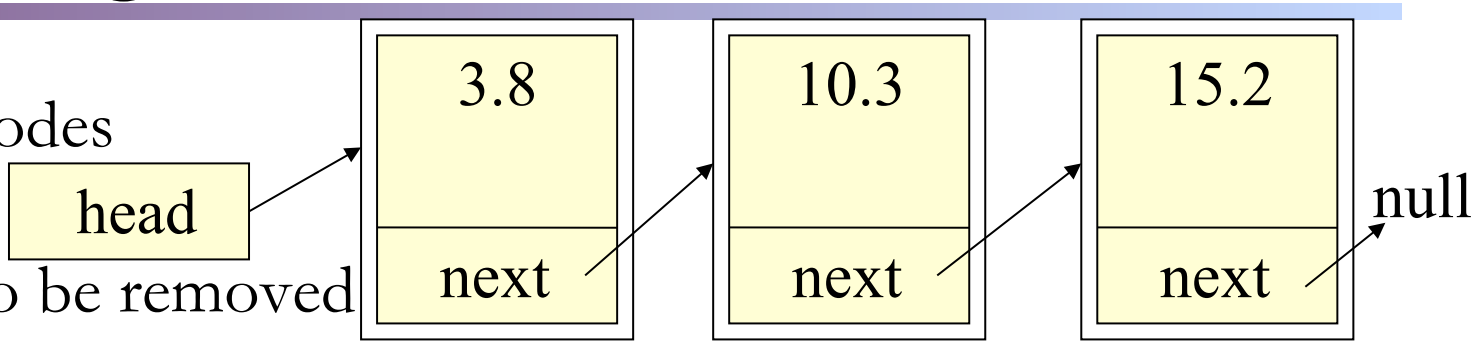
– Step 1: Point second-last node at null. Last node is now no longer part of the list



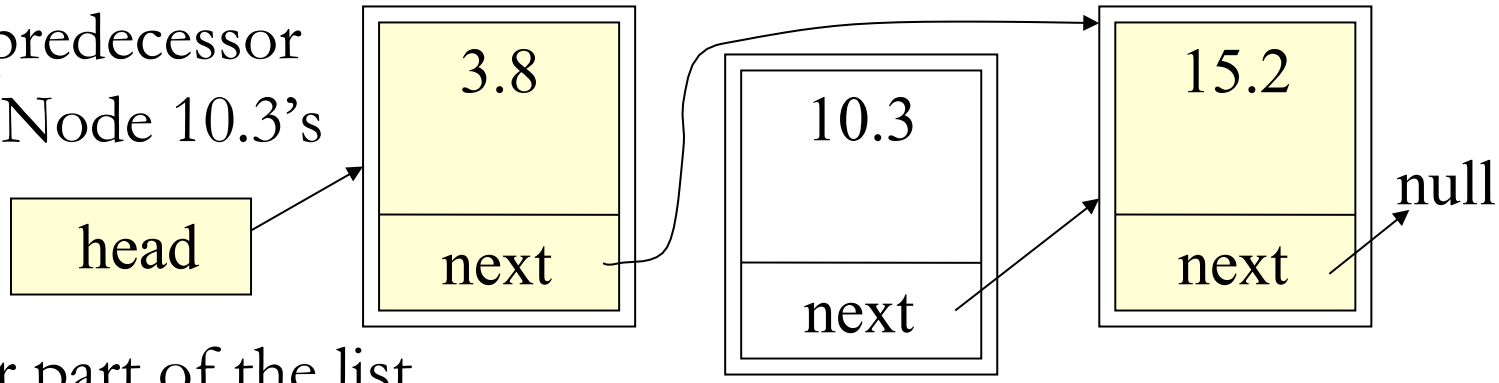

```
ListNode currNd, prevNd;
currNd = head;
prevNd = null;

// Traverse to last node
while (currNd.next != null) {
   prevNd = currNd;
   currNd = currNd.next;
}
prevNd.next = null;
```

Removing ListNode In The Middle

– Initial: a few nodes in the list; Node 10.3 is to be removed

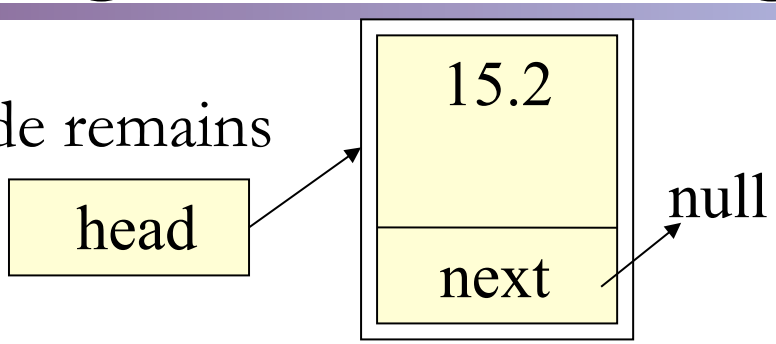


– Step 1: Have predecessor node point at Node 10.3's `next` node. Node 10.3 is now no longer part of the list



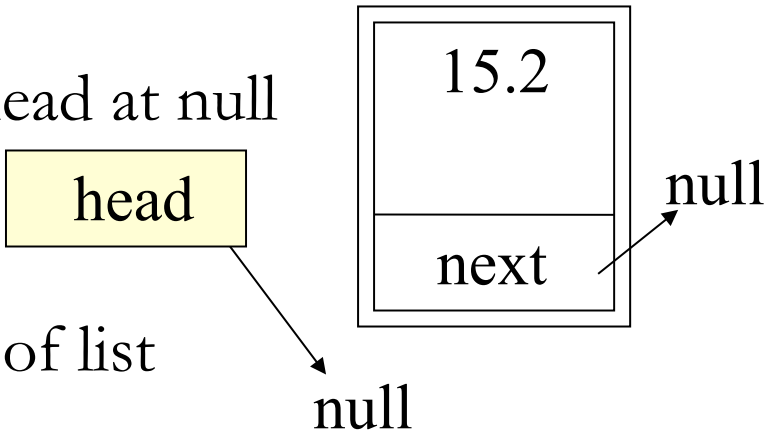
Removing Last Remaining ListNode

– Initial: one node remains in the list



– Step 1: Point head at null

Node is now no longer part of list




```
// Same as removing first node
head = head.next;
```

ListNode - Pseudo-code

```
Class DSAListNode
Class fields : value (Object), next (DSAListNode)    // Could make these public members
                                                     // and avoid the need for getters/setters since
                                                     // ListNode is only used INSIDE LinkedList

Alternate constructor IMPORT inValue (Object)        // Only need one constructor
  value ← inValue                                    // No need for validation
  next ← NULL

ACCESSOR getValue IMPORT none EXPORT value

MUTATOR setValue IMPORT inValue (Object) EXPORT none
  value ← inValue

ACCESSOR getNext IMPORT none EXPORT next

MUTATOR setNext IMPORT newNext (DSAListNode) EXPORT none
  next ← newNext
```

Simple LinkedList - Pseudo-code

```
Class DSALinkedList
Class fields : head (DSAListNode)

Default constructor
  head ← NULL

MUTATOR insertFirst IMPORT newValue (Object) EXPORT none
  newNd ← allocate DSAListNode(newValue)
  IF isEmpty() THEN
    head ← newNd
  ELSE
    newNd.setNext(head)
    head ← newNd
  ENDIF

MUTATOR insertLast IMPORT newValue (Object) EXPORT none
  newNd ← allocate DSAListNode(newValue)
  IF isEmpty() THEN
    head ← newNd
  ELSE
    currNd ← head
    WHILE currNd.getNext() <> NULL DO                    // Traverse to last node
      currNd ← currNd.getNext()
    ENDWHILE
    currNd.setNext(newNd)
  ENDIF
```

<continued next slide>

Simple LinkedList - Pseudo-code

```
ACCESSOR isEmpty IMPORT none EXPORT empty (boolean)
 empty ← (head = NULL)


ACCESSOR peekFirst IMPORT none EXPORT nodeValue (Object)
  IF isEmpty() THEN
    abort
  ELSE
    nodeValue ← head.getValue()
  ENDIF

ACCESSOR peekLast IMPORT none EXPORT nodeValue (Object)
  IF isEmpty() THEN
    abort
  ELSE
    currNd ← head
    WHILE currNd.getNext() <> NULL DO                // Traverse to last node
      currNd ← currNd.getNext()
    ENDWHILE
    nodeValue ← currNd.getValue()
  ENDIF
```

<continued next slide>

Simple LinkedList - Pseudo-code

```
MUTATOR removeFirst IMPORT none EXPORT nodeValue (Object)
  IF isEmpty() THEN
    abort
  ELSE
    nodeValue ← head.getValue()
    head ← head.getNext()
  ENDIF

MUTATOR removeLast IMPORT none EXPORT nodeValue (Object)
  IF isEmpty() THEN
    abort
  ELSEIF head.getNext() = NULL THEN
    nodeValue ← head.getValue()
    head ← NULL
  ELSE
    prevNd ← NULL
    currNd ← head
    WHILE currNd.getNext() <> NULL DO          // Traverse to last node
      prevNd ← currNd                          // We need to get the second-last node
      currNd ← currNd.getNext()                // in order to 'drop' the last node from the list
    ENDWHILE
    prevNd.setNext(NULL)                       // Remove currNd from list
    nodeValue ← currNd.getValue()              // Return value of node that is being removed
  ENDIF
```

Private Inner classes - Java

```
public class DSALinkedList
{
   private class DSAListNode
   {    // Private inner class
      private Object m_value;
      private DSAListNode m_next;
      //could make the classfields public as they can
      //only be seen inside DSALinkedList
      public DSAListNode(Object inValue)
      {
         m_value = inValue;
         m_next = null;
      }
      // normal accessors and mutators if required
   }            // end private inner class DSAListNode
// class DSALinkedList continues
```

Simple LinkedList / ListNode

– NOTE WELL:

- In the practicals, you are supposed to implement a *double-ended* linked list by adjusting this pseudocode
  - » (this pseudocode is just a single-ended linked list)
  - » We will be discussing double-ended lists in a moment

Linked Lists vs Arrays

- Advantages of linked lists over arrays
  - ☑ Can grow and shrink with data - no space wasted
  - ☑ No limits on size (besides memory available)
  - ☑ Easy to insert items at any part of the list
    - » Arrays are only easy to add items to the end
- Disadvantages
  - ☒ Time to access any given element: N/2 steps (on average)
    - » Where N is the number of nodes in the list
    - » Array is direct access: only takes 1 step
  - ☒ `next` pointer is an overhead in terms of memory usage
    - » Adds 1 word (memory address) extra per element in addition to actual data

Order of Complexity

– lets analyse LinkedList find() in Big-O notation

- Best case: the head is the node to find
  - » This is one step, so O(1)
- Worst case: last node is the match, which is N node hops
  - » O(N) in Big-O notation
  - » Each hop involves multiple CPU instructions, but we aren't concerned with these details, so we don't talk about O(5N)
- Average case: On average, we must go halfway: N/2 hops
  - » O(N) – again, the constant multiplying factor of ½ is irrelevant
- We are mostly interested in the average and worst cases

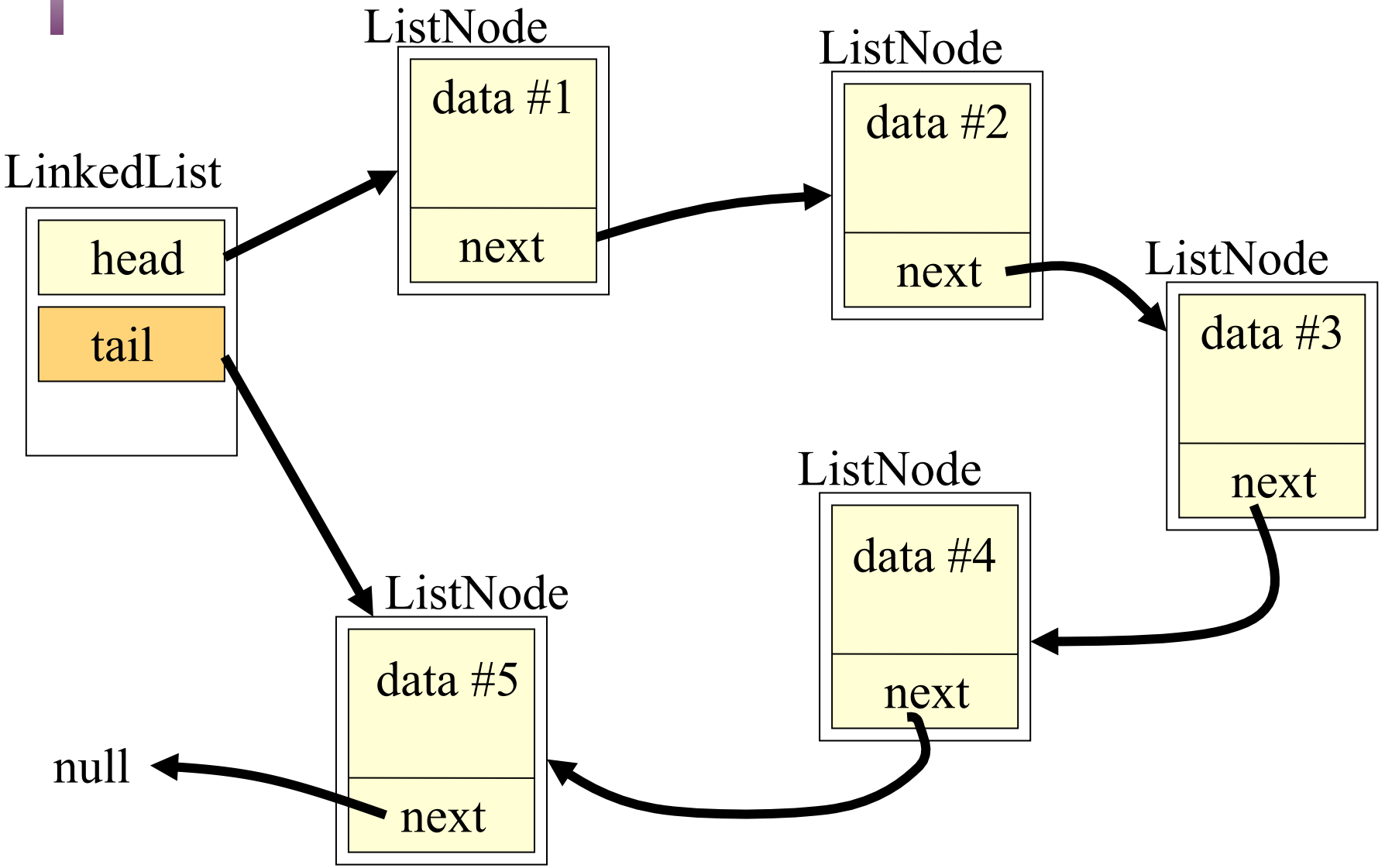
Double-Ended Linked Lists

- We often want to work at the start *and* end of a list
  - *e.g.*, queue adds to end, takes from front
- Simple list is slowest when doing anything at the end
  - Takes O(N) to reach the end – we'd like to do better!
- Solution: add another pointer to point at the last node
  - Call it `tail`
  - Put it with `head` as a member field of the LinkedList
  - That way we can access both head and tail in O(1)
- No disadvantage to this
  - Only adding one pointer to the whole linked list
  - Some code gets more complex, but other code gets easier

Double-Ended vs Single-Ended Lists

- Adding a tail pointer changes how many of the LinkedList methods work
  - insertLast() obviously becomes as easy/fast as insertFirst()
    - » Same with peekLast()
  - insertFirst() and removeFirst() now have to potentially set tail as well in the case of a one-node list
    - » *i.e.*, must maintain tail as well as head everywhere in the code
  - Counter-intuitively, removeLast() isn't helped much
    - » We need to get at the *second*-last node to do removeLast()!

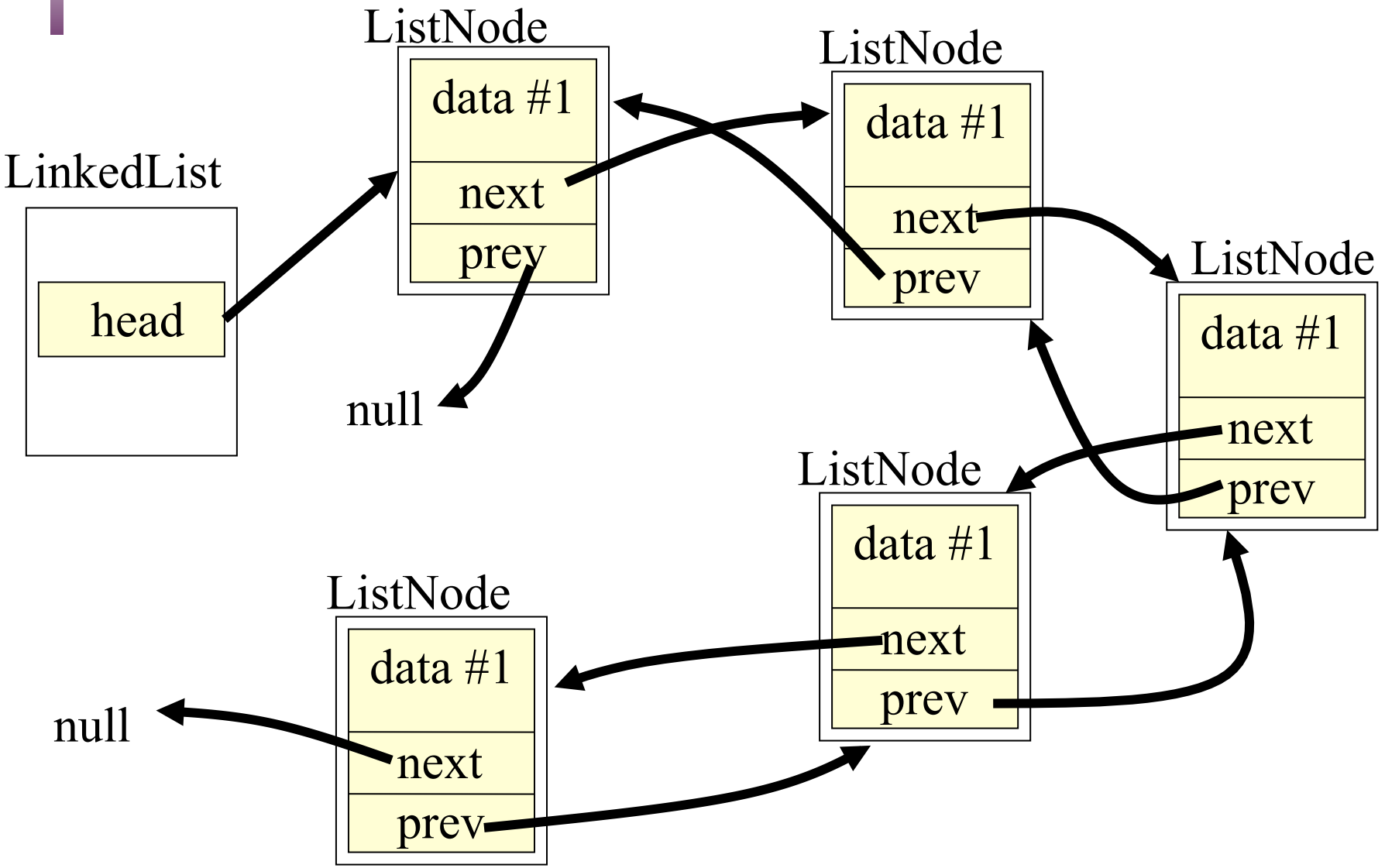
Double-Ended Linked List



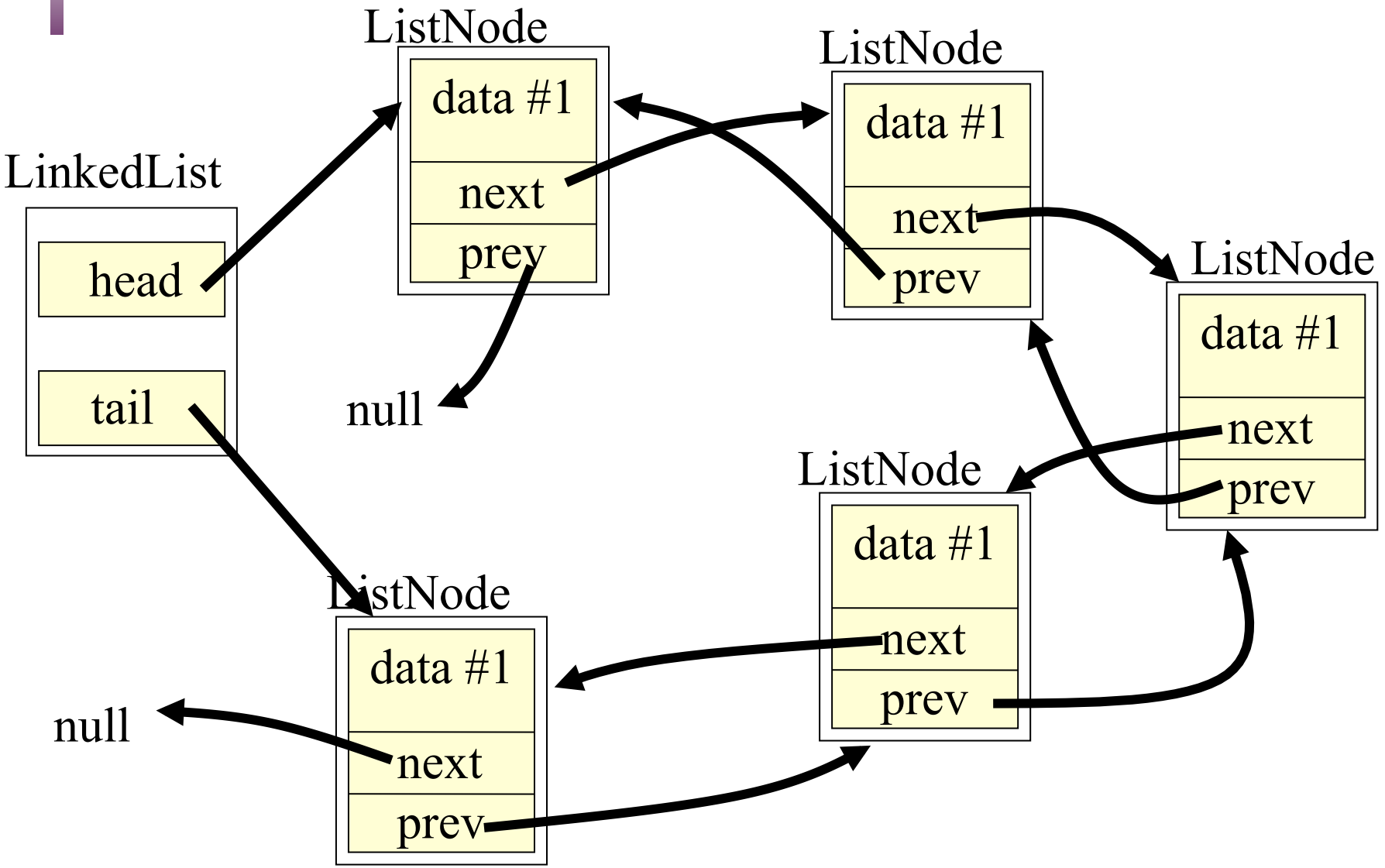
Doubly-Linked Lists

- Sometimes it's useful to be able to traverse both forwards *and backwards* through a list
  - *e.g.*, undo method in an application: in order to allow redo, must be able to go back and forth one node at a time
- Doubly-linked lists add another pointer to each ListNode: `prev`
  - Issue: Now overheads are *two* references per ListNode
    - » That's 2 words extra *per node* (8 bytes in a 32-bit machine; 16 bytes for 64-bit)
    - » Given that ints are only 4 bytes, it can easily *treble* space usage!
  - Benefit: removeLast() is easy if double-ended as well

Doubly-Linked List



Double-Ended, Doubly-Linked List



Sorted Linked Lists

- Since it's easy to insert new elements at any position in a linked list, it can also be convenient to insert them in sorted order
  - When adding a new element, find where it would fit (in sorted order) in the list and insert it there
- This is a variant on the Insertion Sort algorithm
  - A Doubly-Linked List makes this simple
    - » no moving elements – just search backwards then update links!

Saving Linked Lists to file

- We need to serialize our list.
  - we must put all the data together into a single block
- Only a few data structures are organised in blocks
  - Pretty much only arrays, and even then you need to add more info to define what the array's data type is
  - Each node of our linked list can be located in a completely different area of RAM
  - Remember, pointers cannot be made persistent
    - » You cannot save a pointer and expect it to be useful after re-loading it since the pointer says where the data used to be in the old run of the application!
    - » When the application runs again, it won't place objects in the same place – every time new is called, it returns a different address

Saving Linked Lists to file

- And it is not just our Linked List structure we have to worry about
  - our generic Object class for storing the data does not have file I/O methods!
- Create a new Abstract class that forces save/load methods
  - All of our data must then inherit from this class
- Or, simply use Java's object serialization!

Time Complexity Comparison

- Comparison of data structure speeds has many facets
  - Insertion, deletion, accessing and finding
    - » Access = getting the value at a known index
    - » Finding = searching through the data set to find the value
  - Best case, worst case, average case
    - » Usually corresponding to front, rear and middle
      - – … but not necessarily in that order!

Complexity: Array vs Simple LinkedList

Array vs List		Insert	Delete	Access	Find
Array	At front	O(N)	O(N)	**O(1)**	O(1)
	At end	O(1)	O(1)	**O(1)**	O(N)
	In middle	O(N)	O(N)	**O(1)**	O(N)
Linked List (single-ended)	At front	O(1)	O(1)	O(1)	O(1)
	At end	O(N)	O(N)	O(N)	O(N)
	In middle	O(N)	O(N)	O(N)	O(N)

NOTE: Although linked lists are fast to insert/delete, it still takes O(N) to traverse the list and make it to the place to insert/delete

Complexity: List Variants vs Simple List

List variants vs single-ended		Insert	Delete	Access	Find
Linked List (double-ended)	At front	O(1)	O(1)	O(1)	O(1)
	At end	O(1)	O(N)	O(1)	O(N)
	In middle	O(N)	O(N)	O(N)	O(N)
Linked List (doubly-linked) Same as singly-linked single-ended!	At front	O(1)	O(1)	O(1)	O(1)
	At end	O(N)	O(N)	O(N)	O(N)
	In middle	O(N)	O(N)	O(N)	O(N)

Complexity: Sorted List vs Simple List

Sorted list vs single-ended		Insert	Delete	Access	Find
Linked List (sorted) Same as singly-linked single-ended!	At front	O(1)	O(1)	O(1)	O(1)
	At end	O(N)	O(N)	O(N)	O(N)
	In middle	O(N)	O(N)	O(N)	O(N)

- Sorting will help only a little bit for find():
  - If the value *isn't* in the list ,we only need to search up to the point the number *should* be at, and then we can abort
    - » Other lists are forced to search through all elements
- But it doesn't resolve poor scaling with larger N

Notes on Comparisons

- Arrays are unbeatable at O(1) for access time
  - » This makes them indispensable. Why? Because you often only add/remove an element *once*, but access an element *many times*
  - » But if you only need to work on the ends, linked lists are better
- Single-ended lists are only good at working with one end (the head), double-ended lists are good at working with both ends (head and tail)
  - » Otherwise, you might as well be using an array, UNLESS you need the ability to dynamically grow and shrink efficiently
- Doubly-*linked* lists don't help access/find time
  - » How can you know it will be faster to go *backwards*?
  - » Their real strength is application-specific back-and-forthing

Traversal

- Iterating over elements in a data set is a common task
  - *e.g.*, to calculate the average of an array of ages:

```
public double calcAverageAge(double[] ages) {
   double sum = 0.0;
   for (int ii = 0; ii < ages.length; ii++) {
      sum += ages[ii];
   }
   return sum / (double)ages.length;
}
```

    - » An array's O(1) time to access *any* element makes this efficient
- We could do the same with a LinkedList, but only if it provided some way to get at each element
  - Solution 1: Provide a peek(int index) method
  - Solution 2: Provide an *iterator* object

Inefficient LinkList Traversal - Indexing

– The problem with Solution 1 is that it is inefficient

- peek(int index) must start at the beginning of the list every time it is called
  - » *i.e.*, it must traverse through the list from the beginning until it reaches the desired index
- Thus a for loop to visit N elements will cause peek() to make N *traversals* of the list:

```
for (int ii = 0; ii < numAges; ii++)
   sum += ages.peek(ii);
```

  - » Steps: $1 + 2 + 3 + 4 + \ldots + N = N(N+1)/2$ steps $\approx N^2$
- Complexity: $O(N^2)$ (!!)
  - » Truly awful scalability for a simple 'access each element' task

Efficient LinkList Traversal - Iterators

- Solution 2 seeks to solve this inefficiency problem
  - Keep a reference to the *last visited* ('current') node
    - » Sometimes called a cursor, since it indicates where you are
  - Every time the next node is requested, it only needs to make one hop forward
- Naïve approach: make `curr` a member of LinkedList
  - Issue: What if *two* loops access the list at the same time?
    - » Commonly happens with nested for loops
  - But there's only *one* `curr` node in the LinkedList
    - » Can't share it among many for loops
  - You can examine a proper solution to this in the generics slide set

Iterators

- A LinkedList with a single 'current' cursor is limiting
  - Assumes only one 'user/client' of the LinkedList at a time
  - It would be better if every client had its own current cursor
- Iterators are designed to solve this problem
  - Each Java container (list) class has an iterator( ) method
    - » Note that it's not *get*Iterator( ) - a bit inconsistent of Java!
  - Returns an Iterator (or more precisely, an object that inherits from the Java Iterator interface)
    - » You don't need to know the exact object type - knowing it is an Iterator is enough (the power of interfaces and polymorphism!)
    - » Enumeration is an older Java interface for the same idea

Iterator Interface

- Quite simple; only three methods
  - hasNext( ) - queries if more items exist in the list
  - next( ) - move the cursor to the next item in the list
  - remove( ) - optional ability to remove the current item
    - » throw UnsupportedOperationException if you don't support it
  - These just give standard names to a common task
- No prev( ) - inherit from interface ListIterator for that
- Only limit is that behaviour is undefined if element removed by one client while another client iterating
  - Depends on the underlying list being iterated over
  - Some can handle this scenario (*e.g.*, linked list) others can't

Using an Iterator

– Using the iterator directly:

```
public void iterateOverList(LinkedList theList) {
   MyClass c;
   Iterator iter = theList.iterator();
   while (iter.hasNext()) {
      c = (MyClass)iter.next();          ← Get next item and cast from Object to MyClass
      doSomething(c);
   }
}
```

– Using the 'for-each' looping structure:

- Added to Java in 1.5 to simplify coding iterations
- Requires list class to implement Iterable interface

```
public void iterateOverList(LinkedList theList) {
   MyClass c;
   for (Object o : theList) {
      c =  (MyClass)o;                   ← Cast to MyClass
      doSomething(c);
}  }
```

- Also handles nested loops without causing iteration errors

Writing Your Own Iterator

– Iterators are well-suited to being implemented by private inner classes - clients only need to know about Iterator

```
public class MyLinkedList implements Iterable {      ← so for-each loop can be used. Only defines iterator() method
   ...
   public Iterator iterator() {                       ← Return a new Iterator of internal type MyLinkedListIterator
      return new MyLinkedListIterator(this);          ← Hook the iterator to this MyLinkedList object
   }

   private class MyLinkedListIterator implements Iterator {   ← Private class inside MyLinkedList
      private MyListNode iterNext;                    ← Cursor (assuming MyListNode is the node class of MyLinkedList)
      public MyLinkedListIterator(MyLinkedList theList) {
         iterNext = theList.head;                     ← NOTE: Able to access private field of MyLinkedList
      )
      // Iterator interface implementation
      public boolean hasNext() { return (iterNext != null) }
      public Object next() {
         Object value;
         if (iterNext == null)
            value = null;
         else {
            value = iterNext.getValue();              ← Get the value in the node
            iterNext = iterNext.getNext();            ← Ready for subsequent calls to next()
         }
         return value;
      }
      public void remove() { throw new UnsupportedOperationException("Not supported"); }
   }
}
```

Private Inner Classes

- Note that the iterator class was able to access the 'head' field of the list class
  - Even though 'head' is a private field!
  - Works because the inner class is also part of the list class
    - » The inner class has access to all private fields of the outer class
  - This is a very useful property of inner classes that are exploited in many contexts
    - » *e.g.*, event handling in GUIs, helper classes, and Iterators
    - » Replaces the terrible concept of friend classes in C++
  - You could always pass the head instead of the list though

Generics

- You'll notice in the Java 1.5/1.6 Docs that Iterator, and indeed most of the container classes, have this funny '<E>' thing appended to their class names
  - *e.g.*, Stack<E>, Iterator<E> ArrayList<E>
  - These represent what are called generic classes
    - » The same concept is in C#, and called templates in C++
- Generics allow you to parameterise the *class itself* by providing the types that the class operates on
  - *e.g.*, Stack<Plate> means "create a Stack of Plates"

Generics - Why Bother?

- When writing a container without generics, to make it general-purpose you must have Objects for fields
  - Otherwise you will be limiting what can be stored
  - But this means that the client of your container must downcast from Object to the class they actually put in
    - » Annoying, and risks a ClassCastException at runtime
  - Also, primitives are not Objects and so cannot be stored
    - » Must use the wrapper classes - Integer, Float, Double, etc
    - » Which requires *both* a downcast *and* a call to (say) .intValue( )

Generics vs Containing an Object

– Example:

```
ArrayList squareNums = new ArrayList();
for (int ii = 0; ii < 5; ii++)        // Set up an array of square values
   squareNums.add(new Integer(ii * ii));                    ← Must wrap int into Integer object

int iSumOfSquares = 0;                // Go thru array and sum up the squares
for (int ii = 0; ii < 5; ii++) {
//      iSumOfSquares += squareNums.get(ii);                ← Compiler error (int assigned an Object)
//      iSumOfSquares += ((Float)squareNums.get(ii)).intValue();      ← Run-time ClassCastException
      iSumOfSquares += ((Integer)squareNums.get(ii)).intValue();     ← Correct
}
```

– Generics allow the client to define what type to store

- Eliminates the need for casting
  - » Code becomes much cleaner and simpler
- The compiler can ensure that everything is the right type
  - » Catches many logic errors at compile time
- Unfortunately, Java generics don't support primitives
  - » Most other languages with generics do support primitives

Using a Generic Class

– *Using* a generic class is pretty easy - just add <type>

```
ArrayList<Integer> squareNums = new ArrayList<Integer>();
for (int ii = 0; ii < 5; ii++)        // Set up an array of square values
   squareNums.add(new Integer(ii * ii));                    ← Still must wrap int into Integer object

int iSumOfSquares = 0;                // Sum up the squares
for (int ii = 0; ii < 5; ii++) {
   iSumOfSquares += squareNums.get(ii).intValue();          ← No need for cast to Integer
}
```

– There are more complicated things you can do with generics, but the basic concept remains the same

- For example: can have more than one data type parameter

» *e.g.*, Hashtable<String, Double>: String = key/lookup type, Double = value/stored type

code: Hashtable<String, Double> h = new Hashtable<String, Double>( );

Creating a Generic Class

– *Building* a generic class is slightly different

- It requires you to put in a 'placeholder' name for the data type instead of the data type itself (like a parameter)
- The clients of your class then later provide the *real* type
- Let's take a hypothetical MyListNode class:

```
public class MyListNode<E> {          ← Class definition with parameter E for generic placeholder type
   public E data;                     ← Can use generic type E like a real type, just don't know exact type yet
   public MyListNode next;
}
```

- The E (any name will do) is a placeholder for the real type that is only defined when MyListNode is actually *used*

```
public static void main() {
   MyListNode<Double> aNode;          ← E now is 'redefined' to be Double, but only for aNode
   // etc
}
```

Writing Your Own Iterator Revisited

– To make it use generics, just add **<E>**s and replace Object

```
public class MyLinkedList<E> implements Iterable<E> {
   ...
   public Iterator<E> iterator() {
      return new MyLinkedListIterator<E>(this);
   }

   private class MyLinkedListIterator<E> implements Iterator<E> {
      private MyLinkedList<E>.MyListNode<E> iterNext;        ← Weird issue: Java generics require full scoping
      public MyLinkedListIterator(MyLinkedList<E> theList) {        ← NOTE: No <E> in c'tor name
         iterNext = theList.head;
      )
      // Iterator interface implementation
      public boolean hasNext() { return (iterNext != null) }
      public E next() {
         E value;
         if (iterNext == null)
            value = null;
         else {
            value = iterNext.getValue();
            iterNext = iterNext.getNext();
         }
         return value;
      }
      public void remove() { throw new UnsupportedOperationException("Not supported"); }
   }
}
```

Writing Your Own Iterator Revisited

– …as compared to how it was back in Slide 6

```
public class MyLinkedList implements Iterable {
   ...
   public Iterator iterator() {
      return new MyLinkedListIterator(this);
   }

   private class MyLinkedListIterator implements Iterator {
      private MyListNode iterNext;
      public MyLinkedListIterator(MyLinkedList theList) {
         iterNext = theList.head;
      )
      // Iterator interface implementation
      public boolean hasNext() { return (iterNext != null) }
      public Object next() {
         Object value;
         if (iterNext == null)
            value = null;
         else {
            value = iterNext.getValue();
            iterNext = iterNext.getNext();
         }
         return value;
      }
      public void remove() { throw new UnsupportedOperationException("Not supported"); }
   }
}
```

Java Generics Under the Covers

- So how do generics actually work underneath it all?
- Java uses *type erasure*
  - The generic type E only exists for duration of compiling
    - » Temporarily replaced with real type (*e.g.*, Double) which is then used to check for type safety
  - After compilation, all the E's are converted to Object
    - » Type safety is lost at runtime – it's a compile-time-only check
    - » Explains why primitives aren't supported!

Java Generics Under the Covers

- Why did Java's creators use type erasure?
  - Backwards compatibility: they wanted to avoid having to change the .class binary format
    - » Compiling generics to Object means only Objects go in the .class
  - Unfortunately, it makes for some weird behaviour
    - » *e.g.*, Can't create arrays of generics: E[] anArray; is illegal
    - » *e.g.*, Don't even have to give the generic parameter: MyListNode and MyListNode<Double> are *both legal*
- Other languages do it differently
  - C#/C++: every used type gets its *own copy of the class*
    - » C++ actually *recompiles* the generic class *source code* each time
    - » C# is smarter: substitutes generic type within compiled code

Auto-boxing/unboxing

- Embedding a primitive into its class equivalent is referred to as ‘boxing’ in Java
  - *e.g.*, new Integer(-5) boxes -5 into an Integer object
- Java will automatically do boxing for you if possible
  - … and also does automatic unboxing too

```
Integer intObj;
int intVal;
intObj = 10;
intVal = intObj;
```

← Auto-boxing hides need to do intObj = new Integer(10)
← Auto-unboxing hides need to call intObj.intValue( )

  - *Slightly* offsets Java’s lack of generic support for primitives
  - Problems: 1) Slow. 2) Integer == differs from int ==
    - » Integer == is a reference comparison, int == is Integer.equals( )
    - » So try not to rely too much on auto-boxing/auto-unboxing!

Next Week

– Trees