COMP1002
DATA STRUCTURES AND ALGORITHMS

LECTURE 4: LINKED LISTS

Curtin University
Discipline of Computing

Last updated: March 16, 2020

Copyright Warning

This Week

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

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

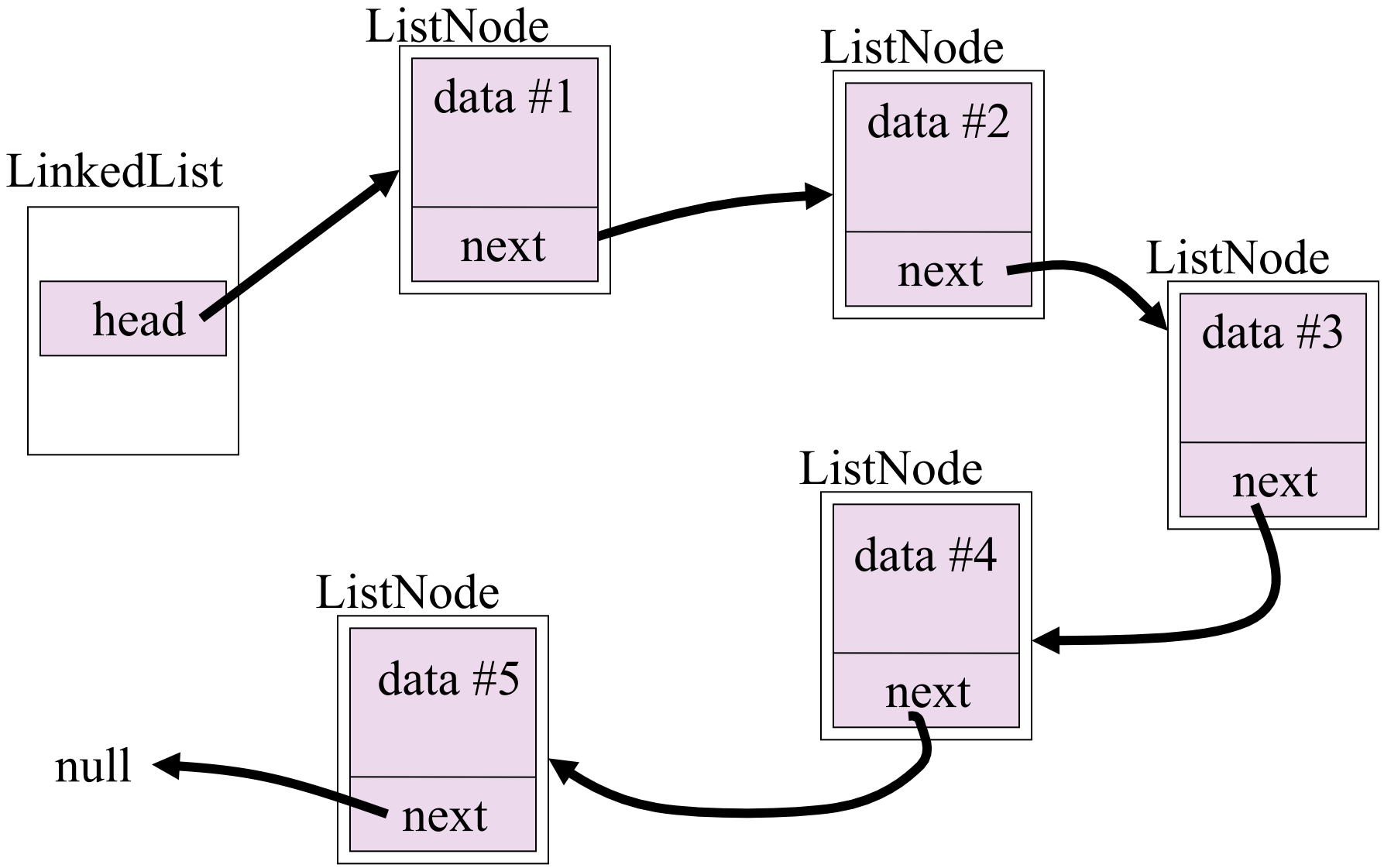
Linked Lists

- 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 reference
- Thus it is really only a couple of member fields with associated getters/setters (Java):
  - One member field (instance variable) for the data value, usually an Object so that the list can be general-purpose
  - A member `next` that is a reference to the next node in the list's chain
    - This points at null/None 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(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(),
  - Delete an item from the list
- peekFirst(), peekLast(),
  - 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

remove(valueToFind)
(less common)
peek(valueToFind)

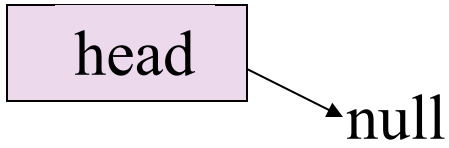
Inserting a ListNode

- 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

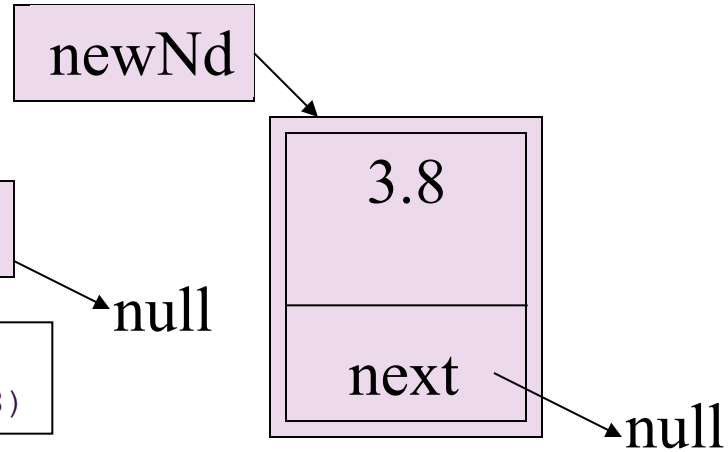
- In code, some of these cases turn out to be the same thing, but when designing a solution you should *always* be thinking through all the possibilities

Inserting First ListNode

- Initial: empty list
  (*i.e.*, head points at null/None)



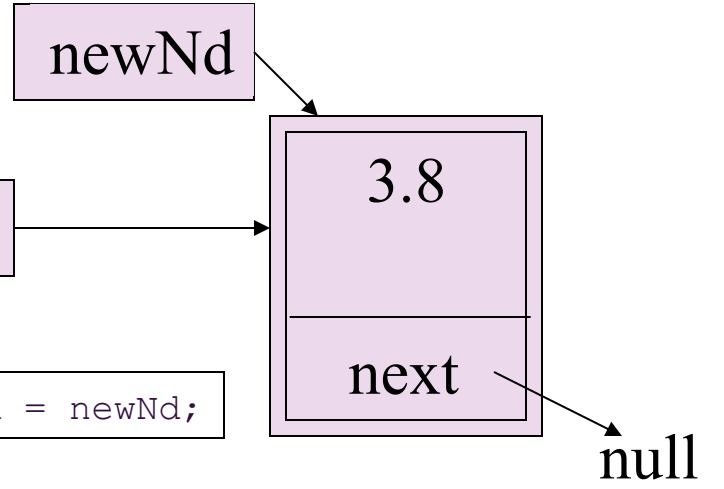
- Step 1: Create new ListNode with (say) value of 3.8




```
ListNode newNd;
newNd = new ListNode(3.8);
```

```
# Python
newNd = ListNode(3.8)
```

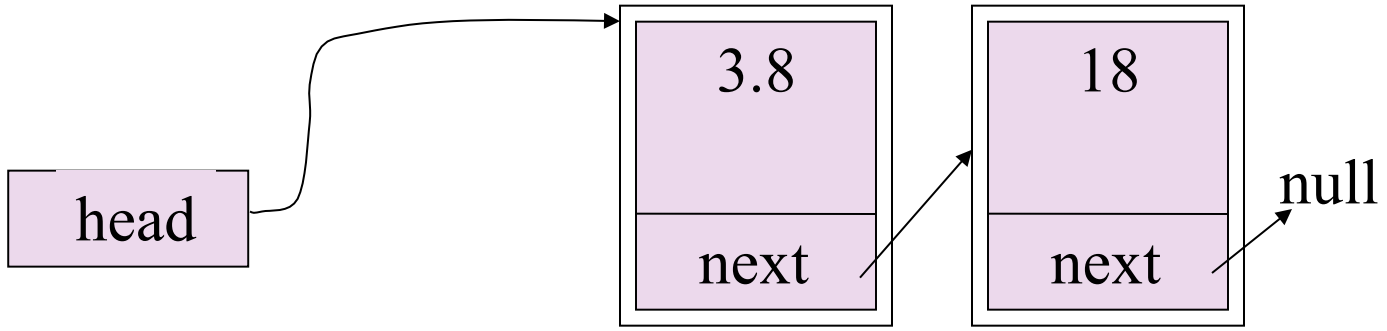
- Step 2: Point head at the newNd to make it the first node




```
head = newNd;
```

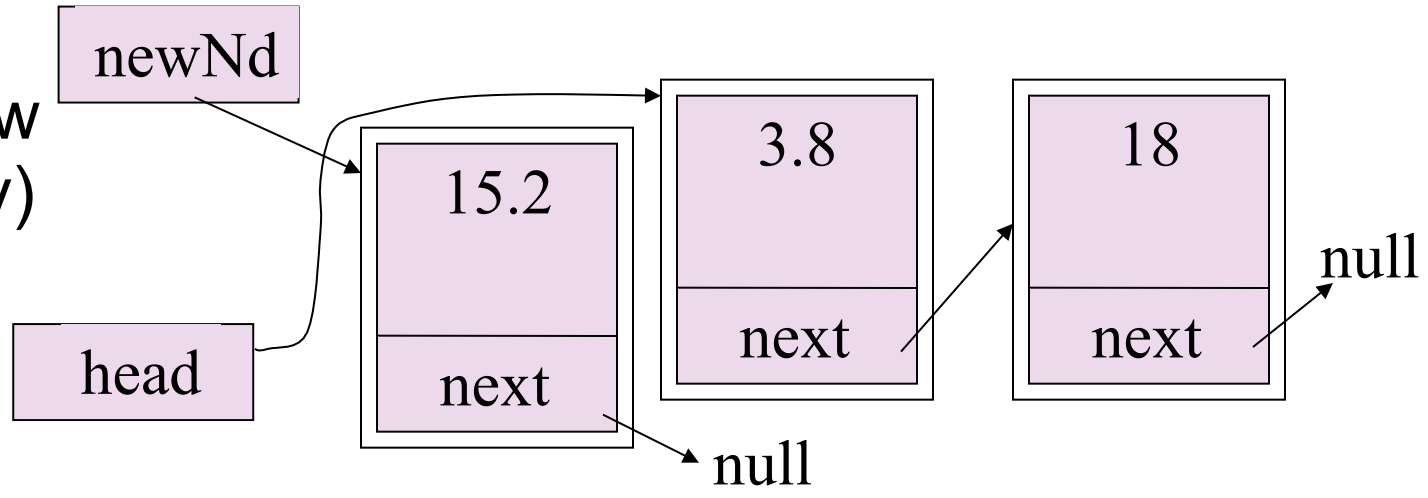
Inserting ListNode Before Head

- Initial: a few nodes in the list



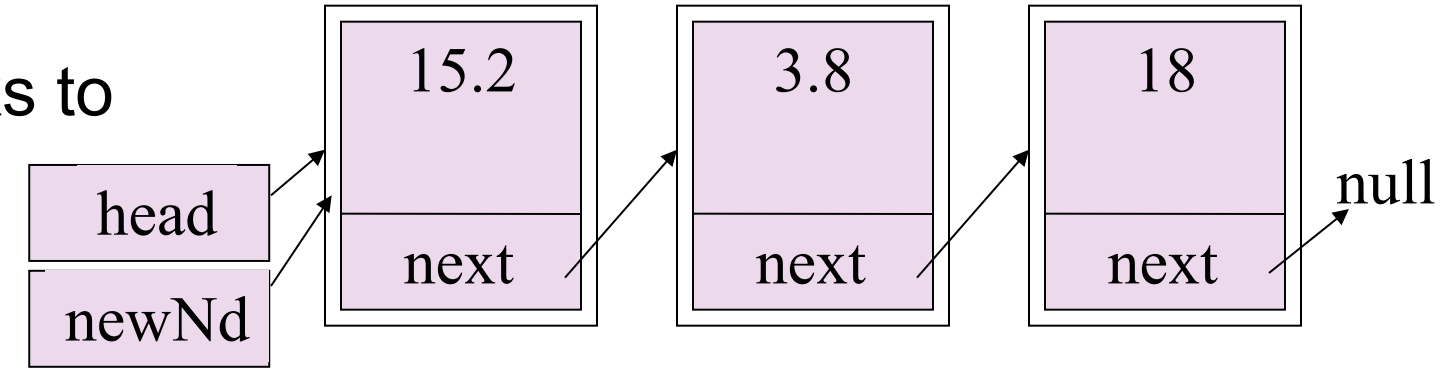
- Step 1: Create new ListNode with (say) value of 15.2

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



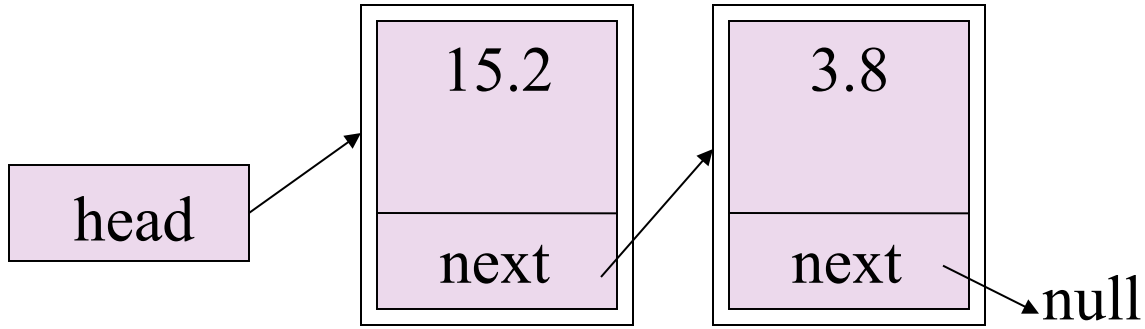
- Step 2&3: Fix links to make newNd first

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

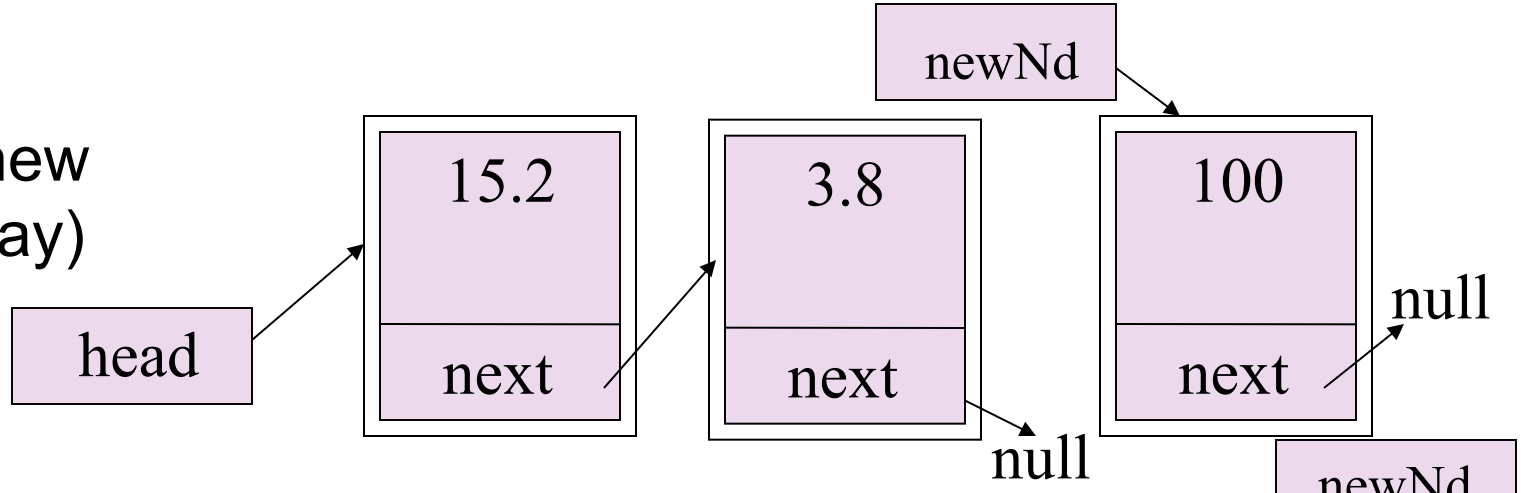


Inserting ListNode After Tail

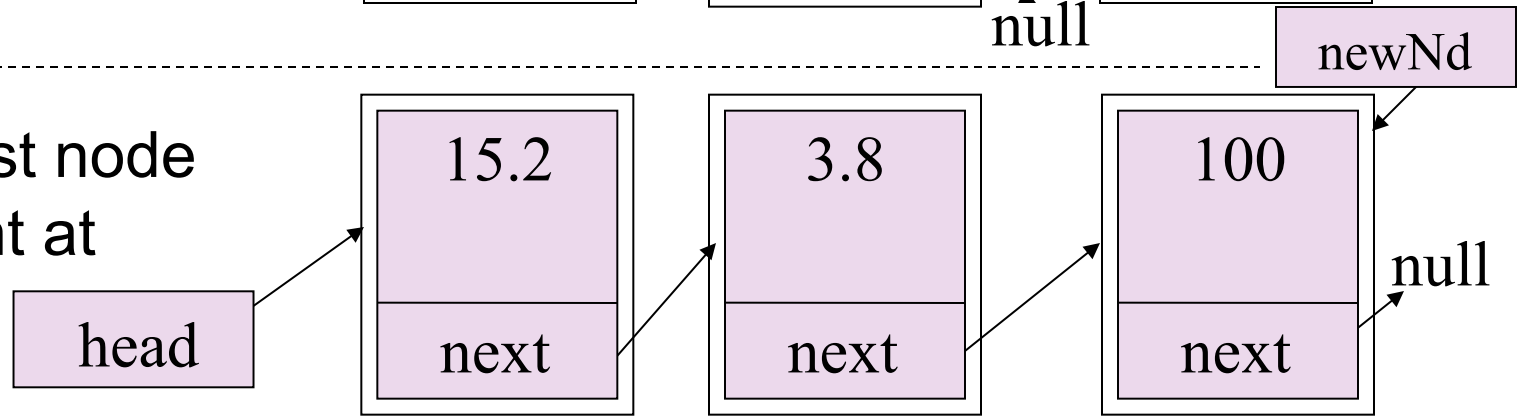
- Initial: a few nodes in the list



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

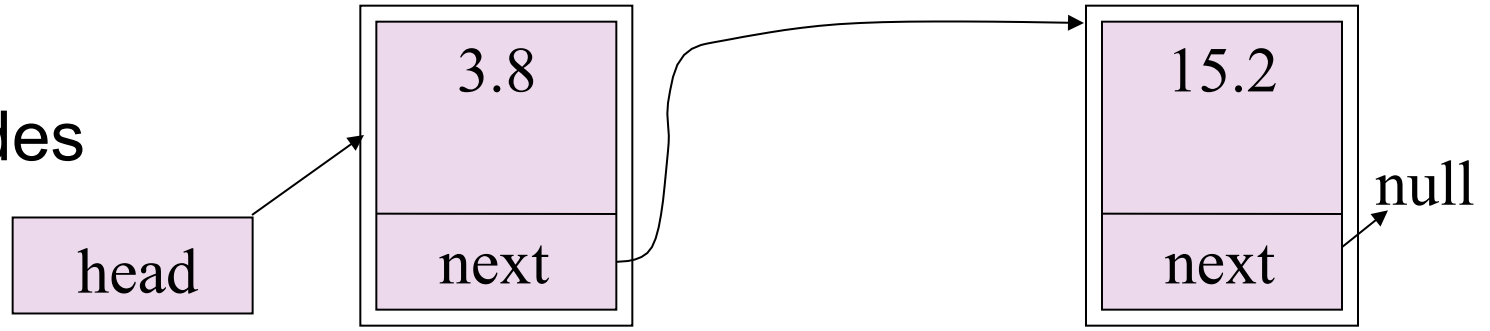


- Step 2: Make last node in the chain point at the newNd

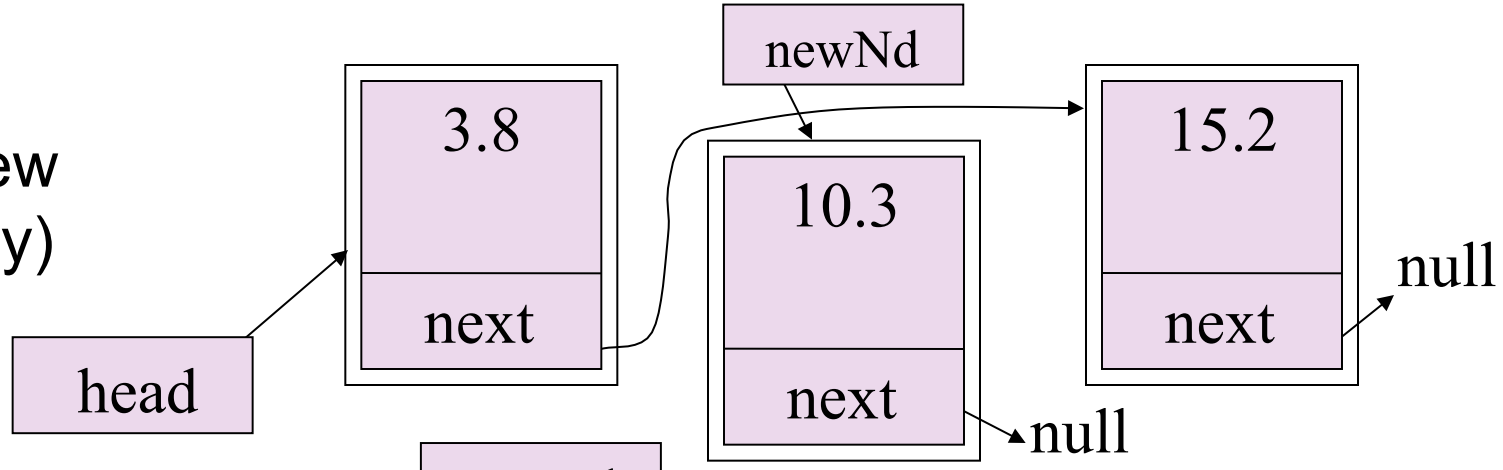


Inserting ListNode In The Middle

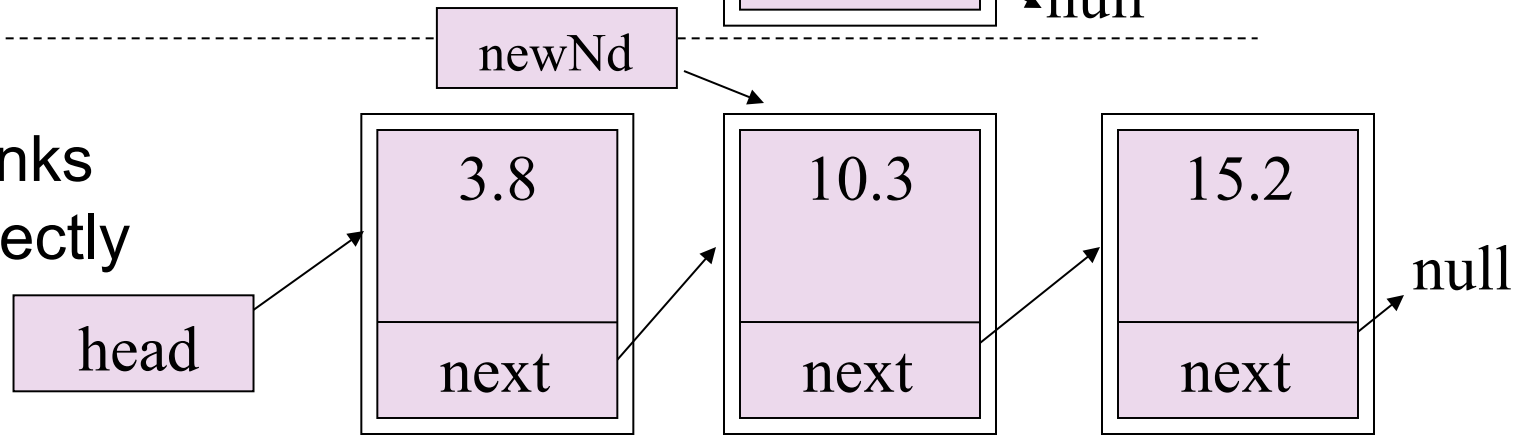
- Initial: a few nodes in the list



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



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

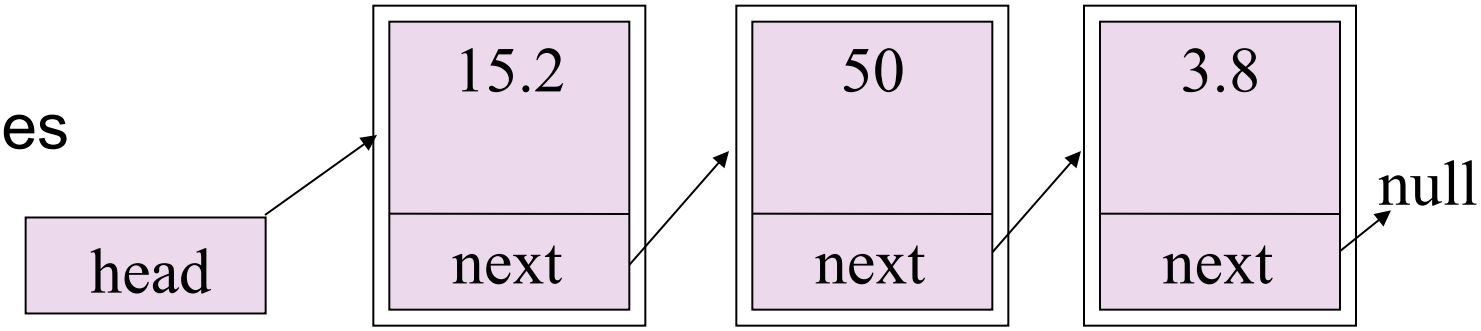


Removing a ListNode

- 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

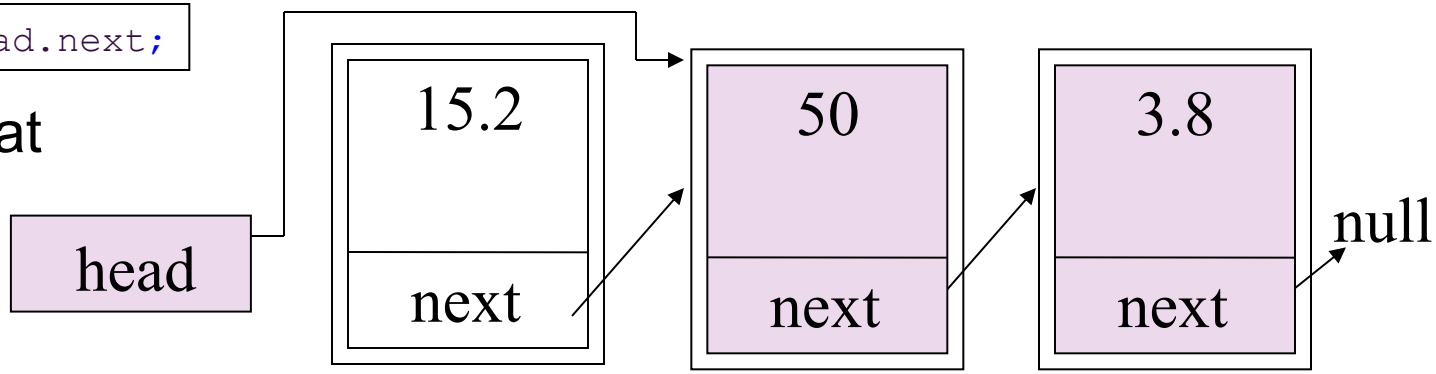
Removing Head ListNode

- Initial: a few nodes in the list




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

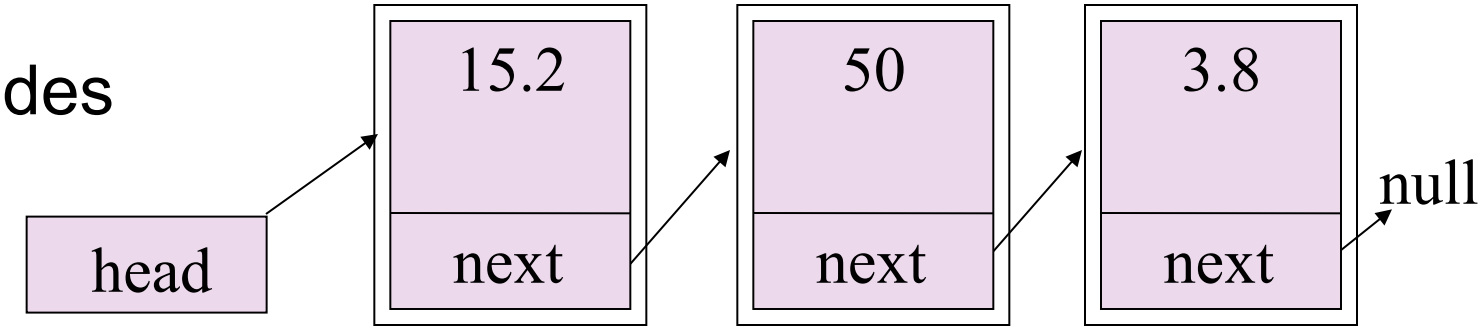
- Step 1: Point head at second node



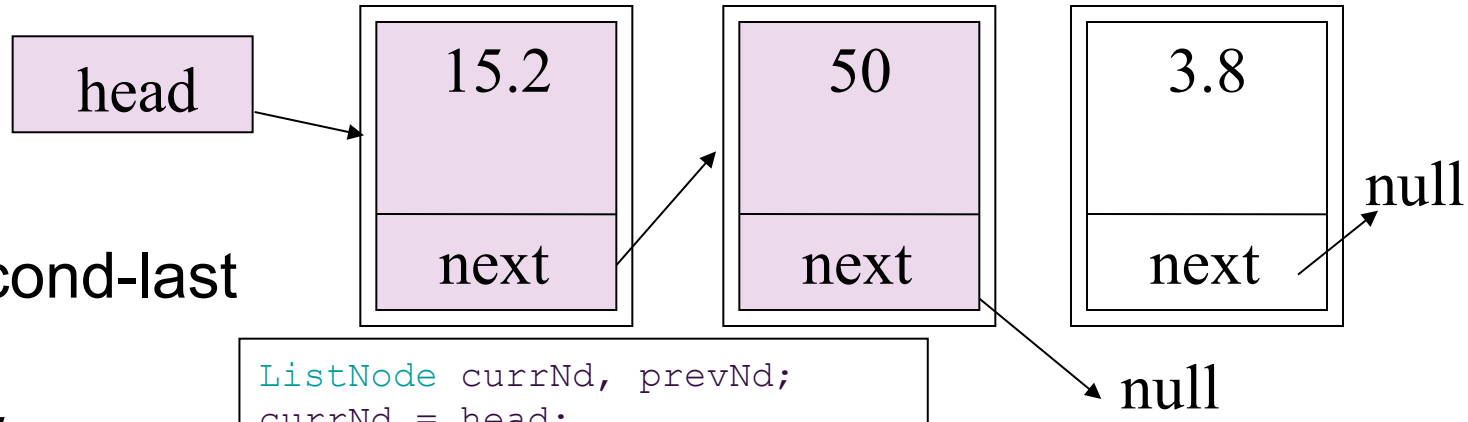
- First node is now no longer part of the list and its memory usage will be garbage-collected by Java (in C/C++, you would have to explicitly free it)
  - The fact that Node 15.2 still points at the second node is not relevant: nothing points *at* Node 15.2, therefore it will be garbage-collected

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



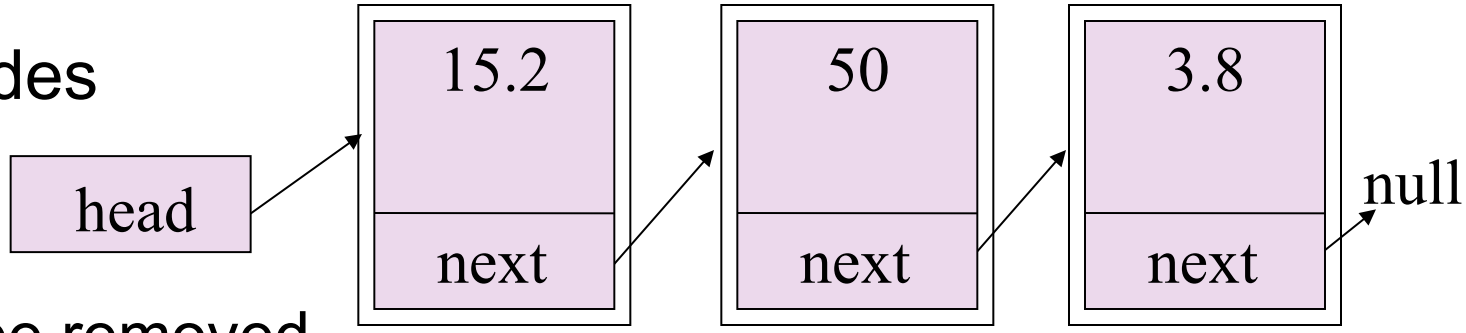

```
ListNode currNd, prevNd;
currNd = head;
prevNd = null;

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

```
# Python
currNd = head
prevNd = null
// Traverse to last node
while (currNd.next)
   prevNd = currNd
   currNd = currNd.next
prevNd.next = None
```

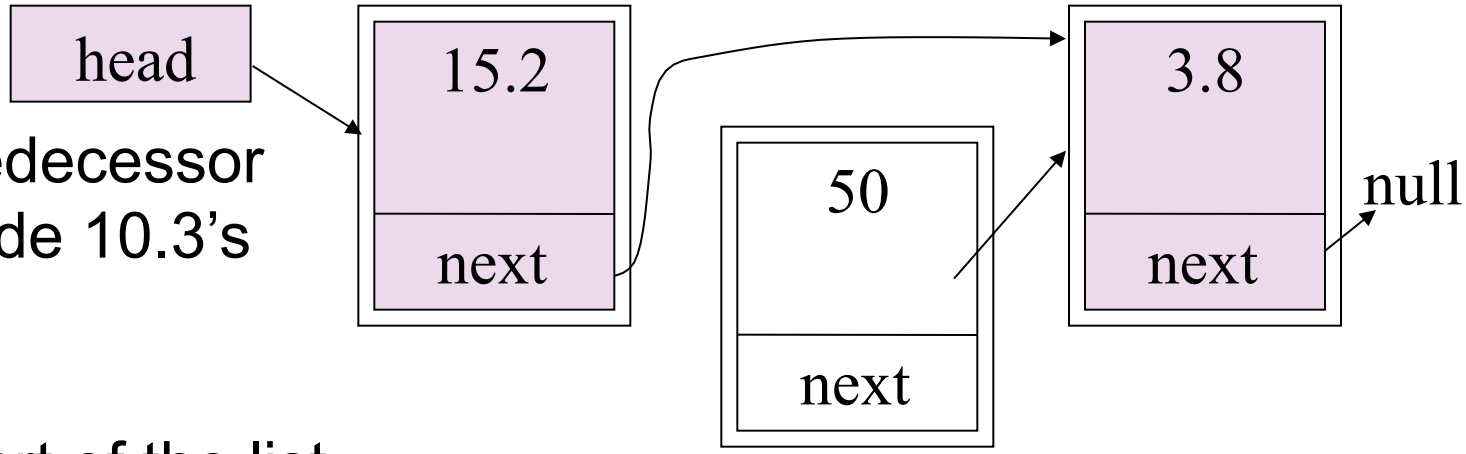
Removing ListNode In The Middle

- Initial: a few nodes in the list;



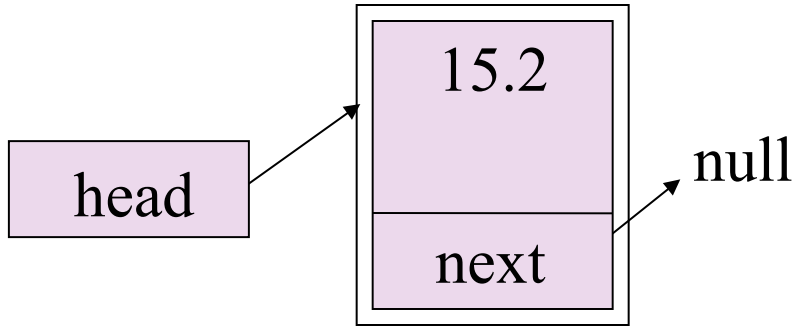
- Node 10.3 is to be removed

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

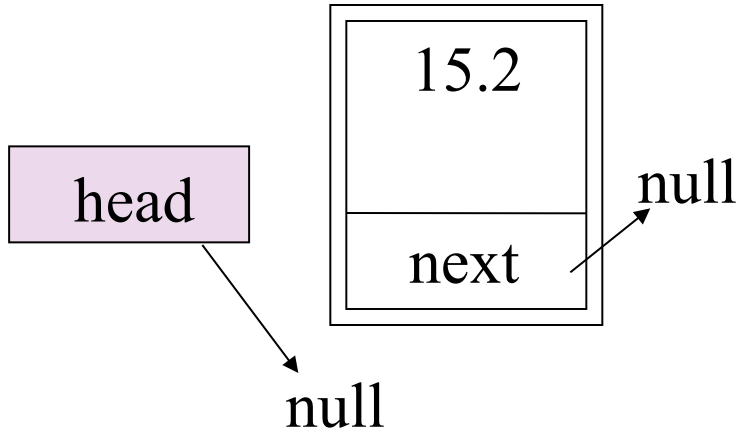


Removing Last Remaining ListNode

- 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   < > is "not equal"
      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

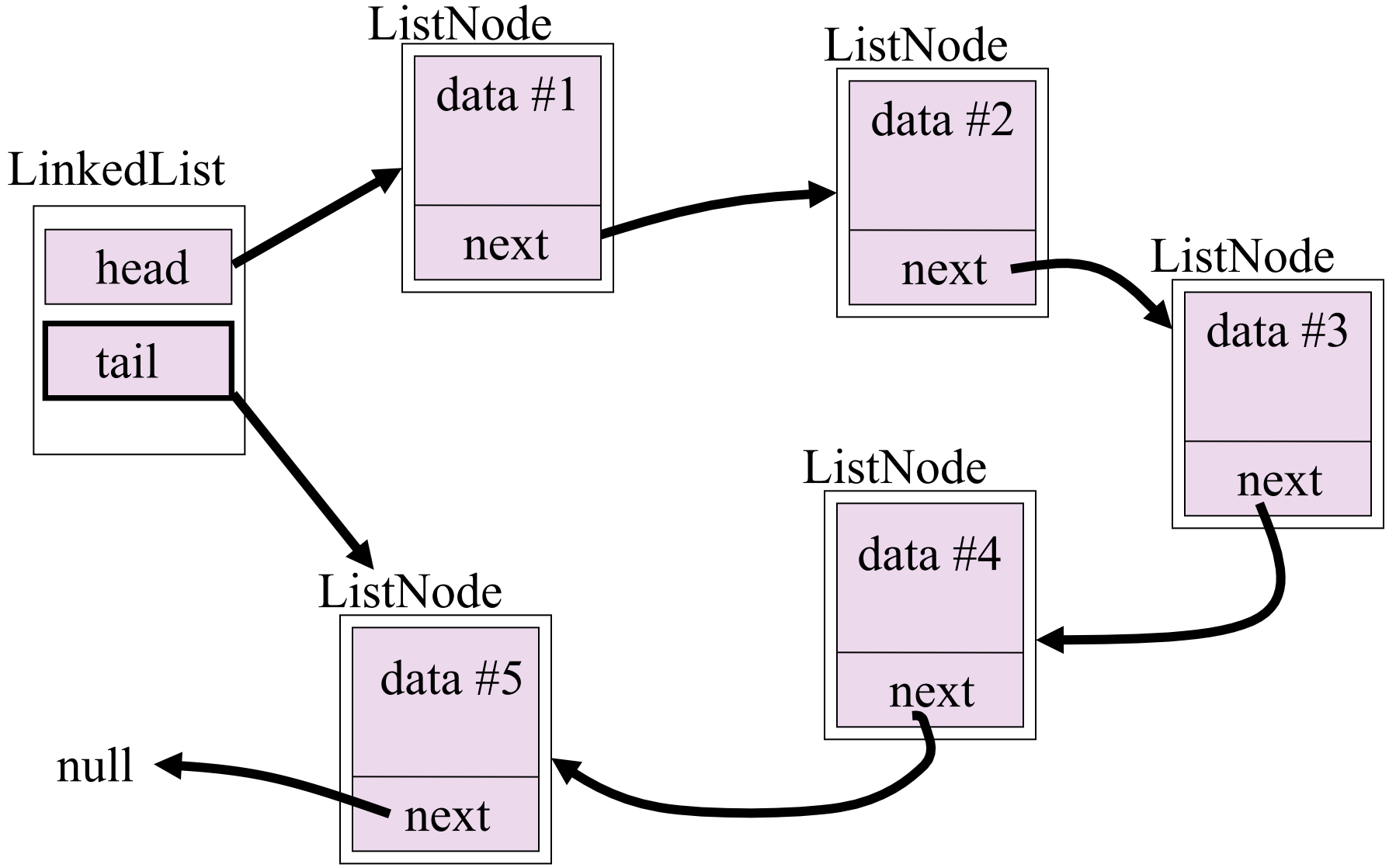
Double-Ended Linked Lists

- 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()!

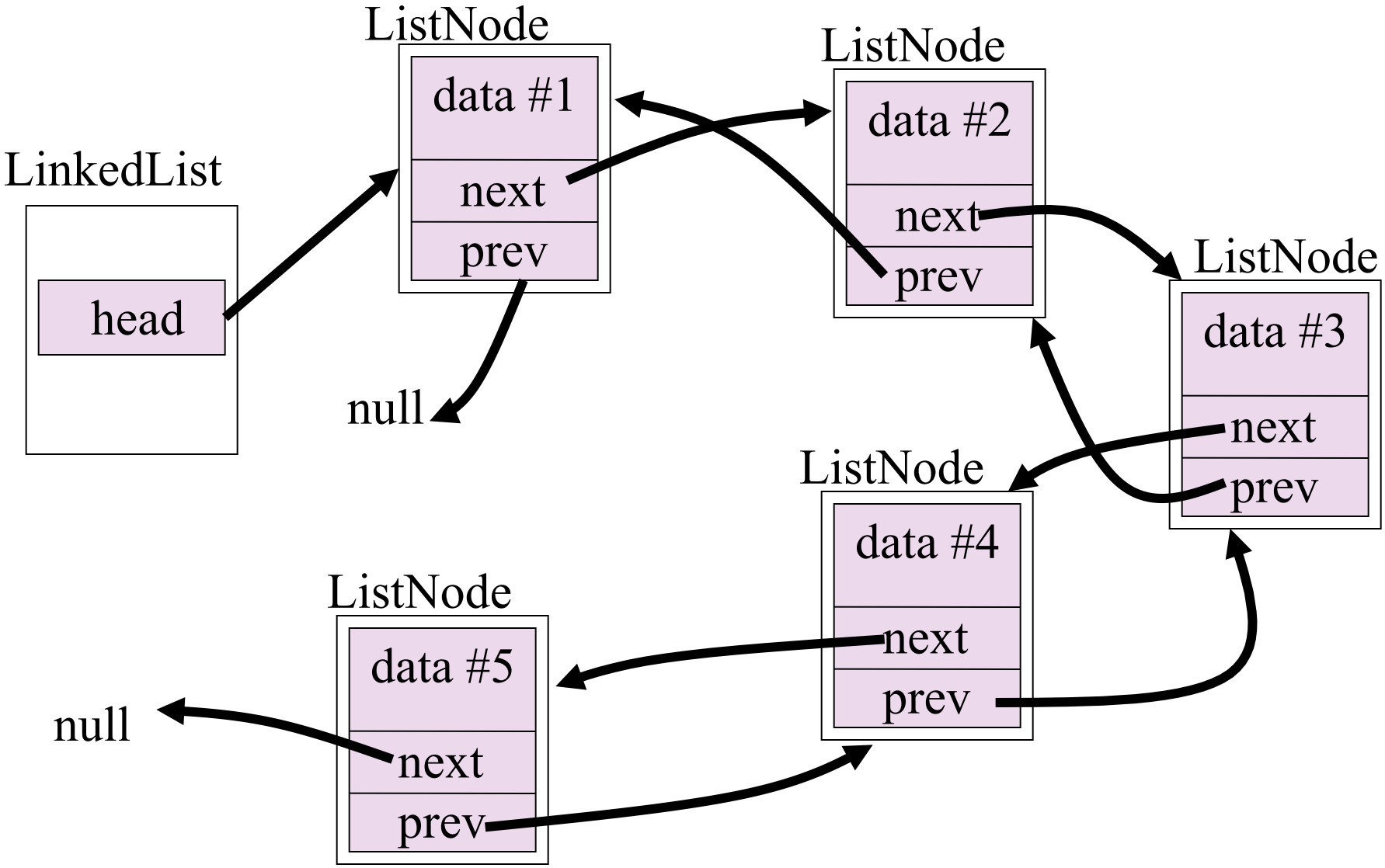
Double-Ended Linked List



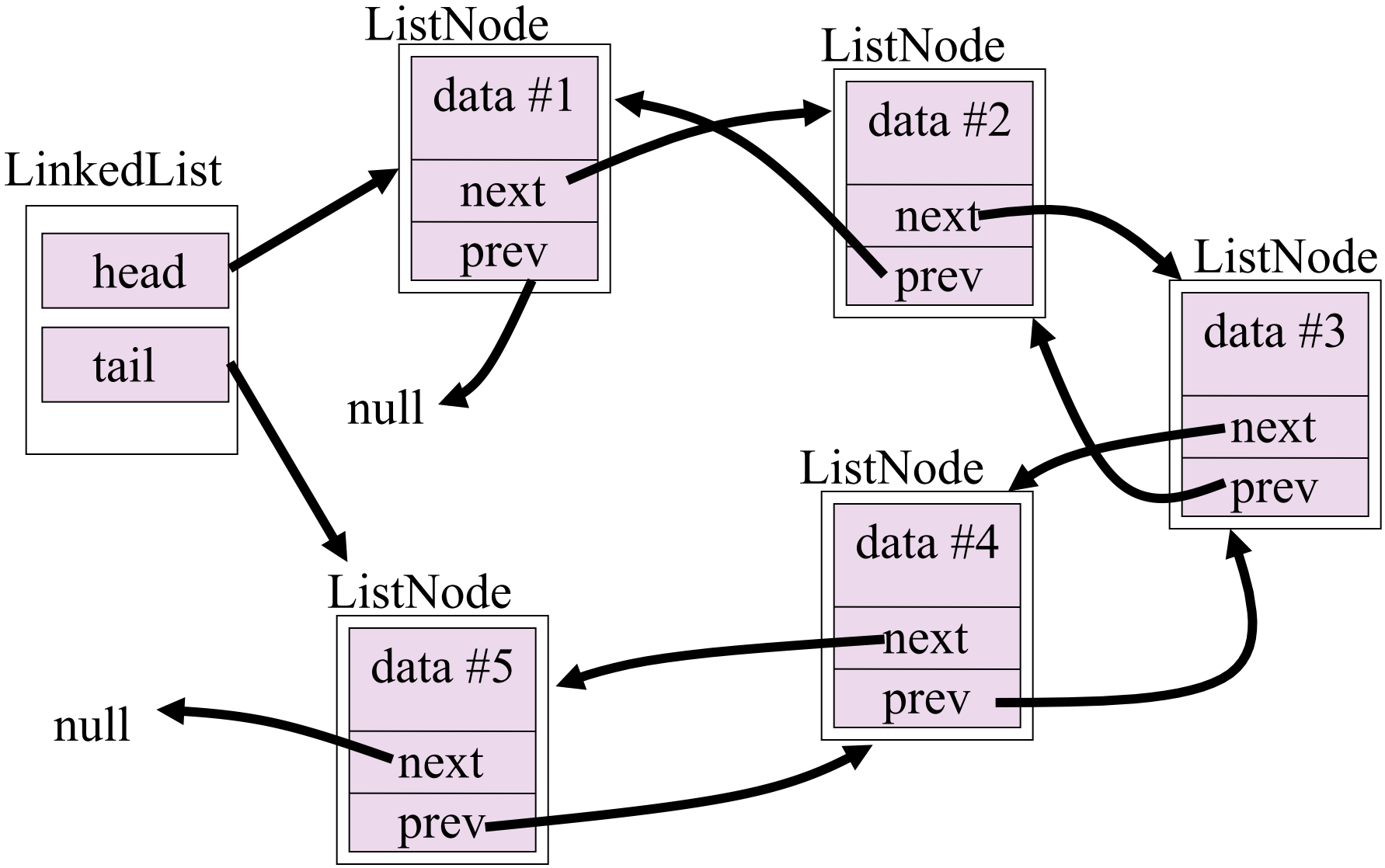
Doubly-Linked Lists

- 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!

Linked List Methods – for sorted list

- isEmpty()
- insertFirst(), insertLast(), **insertBefore(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(),
  - Delete an item from the list
- peekFirst(), peekLast(),
  - 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

remove(valueToFind)
(less common)
peek(valueToFind)

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)	At front	O(1)	O(1)	O(1)	O(1)
	At end	O(N)	O(N)	O(N)	O(N)
Same as singly-linked single-ended!	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

SERIALIZATION

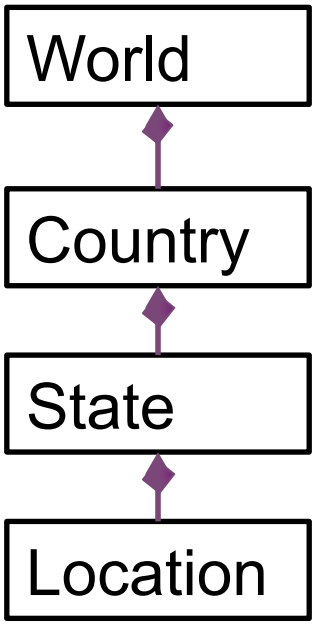
A cool way to save Objects

Persistence

- Persistence: The saving and loading application data so that it lasts between executions of the application
  - *i.e.,* the task of storing an application's state to disk and later loading it back to continue running
- We've already had a look at using text (and binary) formats to save data to disk and read it in later
  - Loading may involve parsing (text) saved data to translate it into a form that is suitable for manipulation by the application
  - Not all persistence requires parsing, but most do simply because storage is static whereas manipulation is dynamic

The Need for Serialization

- What if a ContainerClass (e.g.State) has a classfield which is an object of another class (aggregation)?
  - it does! Location class
- What if ContainerClass had an inheritance relationship that has classfields in the super class? e.g. Region in Country and State



COUNTRY:NAME=Australia:SHORTNAME=AUS:LANGUAGE=English:AREA=7692024:POPULATION=23401892:POPREF=Census2016
COUNTRY:NAME=Fiji:SHORTNAME=FIJ:LANGUAGE=English,Fijian,Hindi:AREA=18274:POPULATION=837271:POPREF=Census2016_Statoi
STATE:NAME=Western Australia:COUNTRY=Australia:SHORTNAME=WA:AREA=2529875:AREAUNIT=km2:POPULATION=2474410:POPREF
STATE:NAME=South Australia:COUNTRY=Australia:SHORTNAME=SA:AREA=983482:AREAUNIT=km2:POPULATION=1676653:POPREF=Ce
STATE:NAME=Victoria:COUNTRY=Australia:SHORTNAME=VIC:AREA=227416:AREAUNIT=km2:POPULATION=5926624:POPREF=Census2
STATE:NAME=New South Wales:COUNTRY=Australia:SHORTNAME=NSW:AREA=800642:AREAUNIT=km2:POPULATION=7480228:POPREF
STATE:NAME=Queensland:COUNTRY=Australia:SHORTNAME=QLD:AREA=1730648:AREAUNIT=km2:POPULATION=4703193:POPREF=Ce
STATE:NAME=Tasmania:COUNTRY=Australia:SHORTNAME=TAS:AREA=68401:AREAUNIT=km2:POPULATION=509965:POPREF=Census20
STATE:NAME=Northern Territory:COUNTRY=Australia:SHORTNAME=NT:AREA=1349129:AREAUNIT=km2:POPULATION=228833:POPREF
STATE:NAME=Australian Capital Territory:COUNTRY=Australia:SHORTNAME=ACT:AREA=2358:AREAUNIT=km2:POPULATION=396857:PO
STATE:NAME=Central Division:COUNTRY=Fiji:SHORTNAME=Central:AREA=4293:AREAUNIT=km2:POPULATION=342386:POPREF=Census
STATE:NAME=Western Division:COUNTRY=Fiji:SHORTNAME=Western:AREA=6360:AREAUNIT=km2:POPULATION=319611:POPREF=Cens
LOCATION:NAME=Perth:STATE=WA:COUNTRY=AUS:COORDS=-32.0024145,115.9097847:DESCRIPTION=State capital
LOCATION:NAME=Geraldton:STATE=WA:COUNTRY=AUS:COORDS=-28.7758955,114.5936913:DESCRIPTION=Coastal city in the Mid West
LOCATION:NAME=Albany:STATE=WA:COUNTRY=AUS:COORDS=-35.0258178,117.8754521:DESCRIPTION=Port city in the Great Southern reg
LOCATION:NAME=Port Hedland:STATE=WA:COUNTRY=AUS:COORDS=-20.3458226,118.5947316:DESCRIPTION=Major port in the North We
LOCATION:NAME=Melbourne Airport:STATE=VIC:COUNTRY=AUS:COORDS=-37.6690123,144.8388386:DESCRIPTION=Airport
LOCATION:NAME=Federation Square:STATE=VIC:COUNTRY=AUS:COORDS=-37.8177089,144.9668941:DESCRIPTION=Meeting place
LOCATION:NAME=Luna Park Melbourne:STATE=VIC:COUNTRY=AUS:COORDS=-37.8676749,144.9746723:DESCRIPTION=Fun park in St Kild

Serialization Issues

- Every object is a reference (pointer) – even Strings!
- Pointers let you jump around in memory (*i.e.*, non-contiguous)
  - This is so useful that we simply cannot do without pointers!
- But 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 a new object is created, it is given a different address
  - Furthermore, the object doesn't even exist yet – it is about to be loaded from disk!
- In other words, pointers cannot be made persistent

Saving Linked Lists to file

- To save to a file, 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

How Does Serialization Work

- So how could it possibly work?
  - The idea is pack the data to be saved into a single contiguous block so that it can be written all together
  - Java and Python do this *automatically* via a language feature called reflection
- Reflection is the ability for an object to query the methods and fields of another class *at runtime*
  - This returns the names of fields, methods, parameters, return types, etc, and the ability to actually set/get the value of each field
    - Thus Java/Python can figure out what data fields an object has and pack them up for saving

Java Serialization

- Writing your own serialization code in Java is quite involved
  - The serialization buffer must be an array of bytes
  - Then you must copy every byte of an object into the serialization byte array
- This is why Java's serialization mechanism is so very useful
  - It does all the work of formatting and copying for you
  - From the programmer's viewpoint, it all comes down to the Serializable interface and a couple of Stream classes

Serializable Interface

- To let Java serialize your class, simply make it inherit from the Serializable interface *e.g.,*

```
public class ContainerClass implements Serializable
{
   ...
}
```

  - That's it! **You do not have to write any code** – Java will determine the contents of your class and serialize it for you *entirely automatically* using reflection
  - It is possible to override Java's serialization if needed
- Note that Serializable is an *empty* interface! In Java:

```
public interface Serializable { }
```

  - *i.e.*, it is only used to mark a class as serializable for Java
  - Don't write your own – import the Java one!

Object I/O Streams (Java)

- Any class that implements Serializable can then be written and read via two serialization-specific classes
  - ObjectOutputStream – for serializing an object and writing it out to an underlying stream
  - ObjectInputStream – for reading an underlying stream, deserializing the object and returning it
    - This creates the object and sets its fields to the serialized data
  - The two classes both know the serialization format
- When creating an Object stream, you must provide another stream – the underlying 'physical' stream
  - Usually FileOutputStream and FileInputStream

Object I/O Streams (Java)

- ObjectOutputStream will serialize the given object *and all objects referenced by that object*
  - This is done recursively, so that the entire object tree will be serialized
  - This supports the composition relationship: a car object consists of wheel objects, engine object and chassis object
    - So when serializing a car, we must serialize all contained objects
  - Objects not marked as Serializable won't be serialized
    - You can also add in custom serialization/deserialization steps, but this is only needed in special circumstances
- ObjectInputStream will deserialize all contained objs

Serializing an Object (Java)

```
private void save(ContainerClass objToSave, String filename)
{
   FileOutputStream fileStrm;
   ObjectOutputStream objStrm;

   try {
      fileStrm = new FileOutputStream(filename);      ← Underlying stream
      objStrm = new ObjectOutputStream(fileStrm);     ← Object serialization stream
      objStrm.writeObject(objToSave);                 ← Serialize and save to filename
                                                          This will also save the ContainerClass'
                                                           contained Location object
      objStrm.close();                                ← Clean up
   }
   catch (Exception e) {                              ← should do more here
      throw new IllegalArgumentException("Unable to save object to file");
   }
}
```

Deserializing an Object (Java)

```
private ContainerClass load(String filename) throws IllegalArgumentException
{
   FileInputStream fileStrm;
   ObjectInputStream objStrm;
   ContainerClass inObj;

   try {
      fileStrm = new FileInputStream(filename);        ← Underlying stream
      objStrm = new ObjectInputStream(fileStrm);       ← Object serialization stream
      inObj = (ContainerClass)objStrm.readObject();    ← Deserialize. Note the cast is needed
      objStrm.close();                                 ← Clean up

   }
   catch (ClassNotFoundException e) {
      System.out.println("Class ContainerClass not found" + e.getmessage);
   }
   catch (Exception e) {
      throw new IllegalArgumentException("Unable to load object from file");
   }
   return inObj;
}
```

Serialization in Python

- Python also uses reflection to convert object hierarchies into a bytestream for saving or transferring data
- The module for serialization is called pickle
- "Pickling" uses the dump() method to convert objects to bytestreams
- "Unpickling" uses the load() method to convert bytestreams to objects
- dumps() and loads() create and use strings instead of files

- **Note**: pickle doesn't protect your data, so only load data you trust

Pickling and Unpickling (Python)

```
# Load Object

try:
    with open(placesDAT, "rb") as dataFile:
        inObject = pickle.load(dataFile)
except:
        print("Error: Object file does not exist!")


# Save Object

print("Saving Object to File...")
try:
    with open(placesDAT, "wb") as dataFile:
        pickle.dump(myObject, dataFile)
except:
    print("Error: problem pickling object!")
```

Pickling and Unpickling (Python)

```
# Multiple objects

try:
    with open(placesDAT, "rb") as dataFile:
        countryObjects = pickle.load(dataFile)
        stateObjects = pickle.load(dataFile)
        locationObjects = pickle.load(dataFile)
except:
        print("Error: Object file does not exist!")


# Save Object
print("Saving Object to File...")
try:
    with open(placesDAT, "wb") as dataFile:
        pickle.dump(countryObjects, dataFile)
        pickle.dump(stateObjects, dataFile)
        pickle.dump(locationObjects, dataFile)
except:
     print("Error: problem pickling objects!")
```

XML and JSON Serialization

- The serialization approaches in previous slides use a format that is specific to the language(s)
  - Don’t expect to be able to pass the object to another language and deserialize it, even if the classes are identical!
    - In fact, even backwards compatibility *in Java* has been broken
- An alternative that are the more portable XML and JSON formats
  - Both provide text-based, structured representations of objects
  - Java provides XMLEncoder and XMLDecoder for this, which work similarly to the Object<Input|Output>Stream classes
  - JAXB (Java Architecture for XML Binding) is a more sophisticated alternative for XML, but also more complex
  - JSON in python requires the json package and then is called similarly to pickle

ITERATORS

Next!!!!

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 + … + N = N(N+1)/2 steps ≈ 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

Iterators (Java)

- 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 (Java)

- 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 (Java)

- 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 (Java)

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

```
import java.util.*;

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 (Java)

- 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

Iterators (Python)

- Need to define **__iter__()** and **__next__()** to build an iterator:
  - __iter__ sets up your cursor to hold the current position in the collection
  - __next__ defines how you move on to the next element
  - When you reach the end of the iteration, raise the **StopIteration** exception (e.g. hit the None at the end of a list)

- Once you have these, you can go through a list using a for loop:

```
for item in mylist:
    print(item)
```

- __iter__() and __next__() are predefined functions, similar to __init__() and __str__().

- A good resource for more information is : https://www.w3schools.com/python/python_iterators.asp

Iterator Example (Python)

```
class ListNode():                       # incomplete sample code as guide
    def __init__(self, data):
        self._next = None               # _ can be used to indicate
        self._data = data               # "private" fields

class LinkedList():
    def __init__(self):
        self._root = None

    def insertFirst(self, value):
        if self._root == None:
            self._root = ListNode(value)
        else:
            newNode = ListNode(value)
            newNode._next = self._root
            self._root = newNode
```

Iterator Example (Python)

```
# class ListNode() continued

    def __iter__(self):          # iter method creates iterator
        self._cur = self._root # defines cursor for current
        return self              # returns reference to object

    def __next__(self):          # next method defines traversal
        curval = None            # prepare return value
        if self._cur == None:       # check for end of list
            raise StopIteration     # Exception for end of iter
        else:
            curval = self._cur._data      # this return value
            self._cur = self._cur._next   # move cursor along
        return curval                     # return the value

# Problem: what if there's mutliple iterators on one list?
```

Iterator Example (Python)

```
# using the iterator

ll = LinkedList()
ll.insertFirst("first")
ll.insertFirst("second")
ll.insertFirst("third")
ll.insertFirst("fourth")

# creating list and getting next value
myiter = iter(ll)
value = next(myiter)

# using a for loop to go through list
for value in ll:
    print(value)
```

Another Iterator Approach (Python)

- Python lets you create a **generator** using the **yield** statement
- Maintains the current values of the method/function until next call
- **__iter__()** , **__next__()** and **StopIteration** are automatically implied
- Solves concurrent iterator issue as currNd is local to method

```
# example of iterator using yield
    def __iter__(self):
        currNd = self.head
        while currNd is not None:
            yield currNd._value
            currNd = currNd._nextNode
```

For more info: https://www.programiz.com/python-programming/generator

PRACTICAL 3

Aims:

- To create a general purpose Linked List class.
- To extend the linked list class with an iterator.
- To convert stack/queue to use the linked list.
- To save the linked list using serialization.

Next Week

- Trees