Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 2: Lists

Reading materials

- Goodrich, Tamassia, Goldwasser (6th), chapter 3
- OpenDSA (https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/): chapter 9.1-7

Contents

Joanna Klukowska
joannakl@cs.nyu.edu

1 Abstract Data Types

What is the abstraction? **Abstraction** is a model of a system that includes only the details needed by the viewer/user of such system. The complexity and details of actual implementation should be hidden from the user - that is called **information hiding**. Why? Because the complex design details are not relevant to the user and may make it harder to understand the system. Imagine what would happen if you needed to know and understand every detail of how cars work, before you could use them.

An **abstract data type (ADT)** provides a collection of data and a set of operations that act on the data. An ADT's operations can be used without knowing their implementations or how the data is stored, as long as the interface to the ADT is precisely specified. An ADT is implementation independent and, in fact, can be implemented in many different ways (and many different programming languages).

In this course we will use two different ways of specifying an ADT:

- **informal specification** - an English description that provides list of all available operations on the data with their inputs and outputs;
- **formal specification** - a Java `interface` definition that later can be implemented by concrete classes.

We will be using ADTs from three different perspectives:

- **application level** (or user, or client): using a class for which you only know the formal ADT specification;
- **logical level**: designing the ADT itself given a set of requirements specified by the non-programmer user (this may involve asking questions);
- **implementation level**: writing code for a class that implements a given ADT.

1.1 Collection

A collection object is an object that holds other objects. Typical operations provided by collection classes are:

- insert,
- remove,
- iterate through the collection.

A formal ADT for a generic collection is provided by Java's `Collection<E>` interface:
http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html.
An `ArrayList<E>` class implements this ADT.

1.2 List ADT

There are many different possible List abstract data types that require different sets of operations to be defined on them. The ADT for the list that we defie in this lecture is a very general one. We will use it (after slight revisions) in several future lectures and provide different ways of implementing the list interface.

For now, we will specify List ADTs that expect objects of a specific type. Later, we will revise the List ADT to be general enough to work with any chosen type of objects, i.e., we will define a generic List ADT.

2 Array Based Implementation of List

The **`ArrayList<E>`** class implements Java's **`List<E>`** interface:
http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html.

You could easily (well, it should be easy) create a class that contains an ordinary array as its data field and provides methods that implement all the operations required by Java's **`List<E>`** interface.

The important things to notice is that the **`List<E>`** interface t does not mention a possibility of a full list.

Joanna Klukowska
joannakl@cs.nyu.edu

- So, how can we ensure that we always have enough room to add another item? After all the arrays in Java are fixed size, aren't they?
- How does `ArrayList<E>` class do it? Does it really use an array?

How big should the array be with an empty list? The quick answer is that it depends on the application. The array should start with some non-zero size. The initial size can be provided by the user as an option to the constructor.

Resizing an array. In order to "resize" an array that is full, we need to

- allocate a larger array,
- copy all the data from the full array to the new larger array,
- "abandon" the smaller array and use the larger array from now on.

This has to happen without user intervention. Any method that adds elements to the List needs to verify if there is enough room. If there isn't, it needs to resize the array before performing the addition.

How much extra space should we add to the array?

One extra element. This is a very bad choice! Copying of all the items from one array to another takes time. It should be avoided if possible. Increasing the array size by 1 means that we have to do it often.

Fixed number of elements. This generally works well. The actual number of elements should depend on the application.

Doubling current size. This is often implemented and produces good results assuring that the copying of the data is done infrequently. But for very large arrays, this may lead to a very large increase in memory usage.

3 Linked Structures

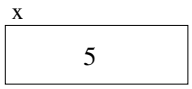
3.1 Review of Java Reference Variables

In Java, when you allocate a variable of a primitive data type (see http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html for listing of types and descriptions) the memory location associated with that variable contains the value assigned to that variable.

Example:

The following lines of code

```
int x;
x = 5;
```



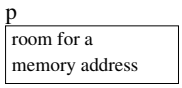
result in a 32-bit block of memory being associated with the variable x and then the bits are used to represent the value of 5 (for simplicity, I will just write the number 5 instead of its binary equivalent).

With reference type variables the memory allocation is different. When you create a reference variable it does not have any room (bits/bytes in memory) for storing data. It only contains enough memory to store a reference or an address of memory location at which the actual data is stored.

The following line of code creates a reference variable, unless it is followed by creation of an actual object, there is no room to store data.

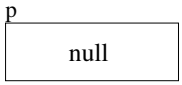
Joanna Klukowska
joannakl@cs.nyu.edu

```
Person  p;
```



In fact, you should make a habit of always assigning the null value to a reference variable that does not actually reference any valid data:

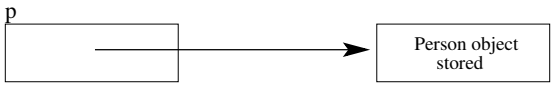
```
Person  p = null;
```



The `new` operator is used to create the actual object and to assign its memory address to a reference variable.

```
p = new Person();
```

And since we do not really care about the hexadecimal value of the memory address (nor do we really have a way of knowing it in Java), you'll see arrows used to show that a reference variable refers to or "points to" a memory location that contains the actual object.

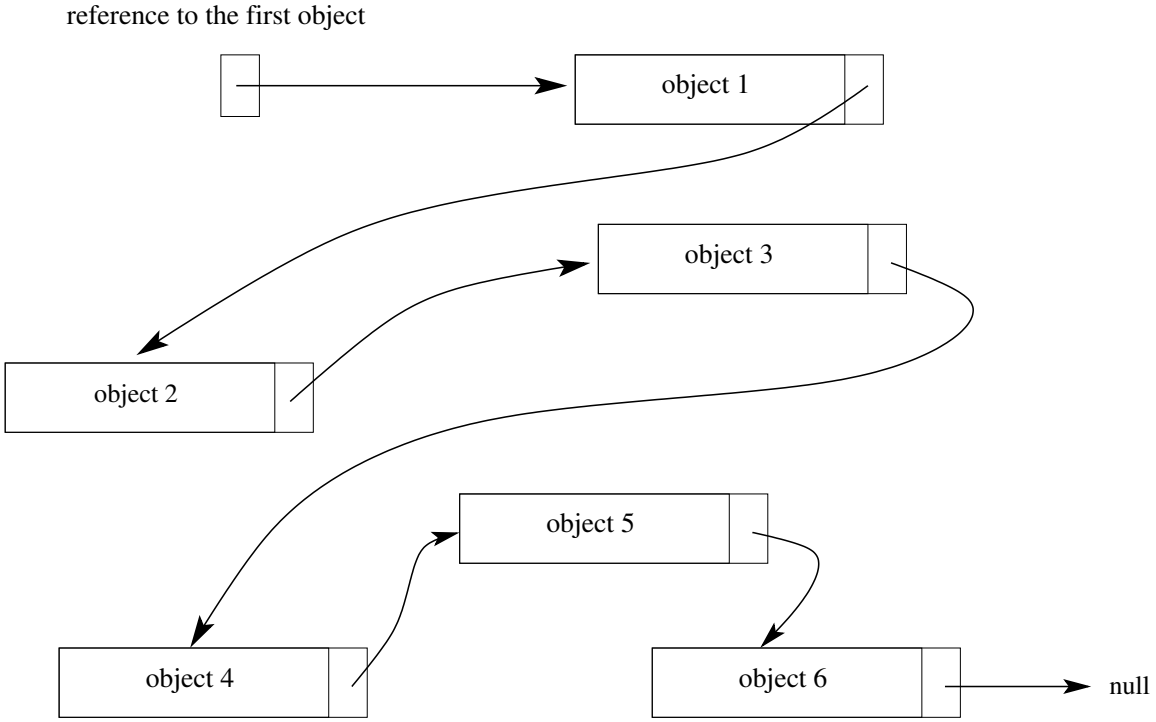


3.2 Linked Structures

The idea of creating linked structures is based on the reference variables.

When we create an array, a consecutive block of memory locations is assigned to it. This allows us to easily access each element using subscript operators.

Linked structures are composed of individual objects that are linked to one another in some way. For example, the following six objects are connected, but they do not necessarily exist in consecutive memory locations.



Joanna Klukowska
joannakl@cs.nyu.edu

Notice that each object has an extra "box" with an arrow coming out of it. The linked structures are composed of special kind of objects, usually referred to as **nodes**. Each node contains the actual data, but also a reference (possibly more than one) to another node. Linked structures get connected through these references to the next node.

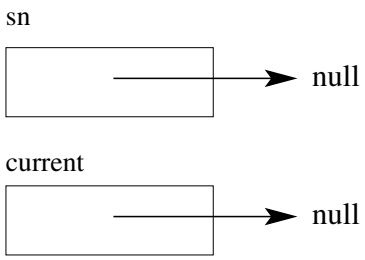
3.2.1 Linking Nodes Together

We want to connect several nodes together in order to create a linked structure. Assume that our node will store a arbitrary type of objects as the data. This means that each node contains data part of type `E` (the commonly used generic specifier), and the reference to another node:

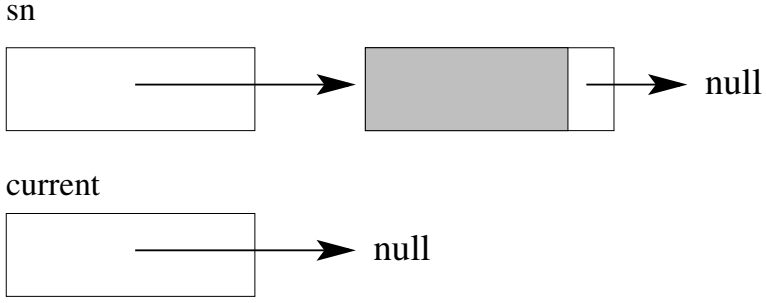
```
class Node <E>{
  private E data;

  private Node<E> next;

  public E getData () {
    return data;
  }

  public Node<E> getNext () {
    return next;
  }

  public void setData (E data ) {
    this.data = data;
  }

  public void setNext (Node<E> next )
    this.next = next;
  }
}
```

The table below shows the steps needed to create several nodes and link them together into a linked structure.

- `Node<E> sn = null;`
  `Node<E> current = null;`

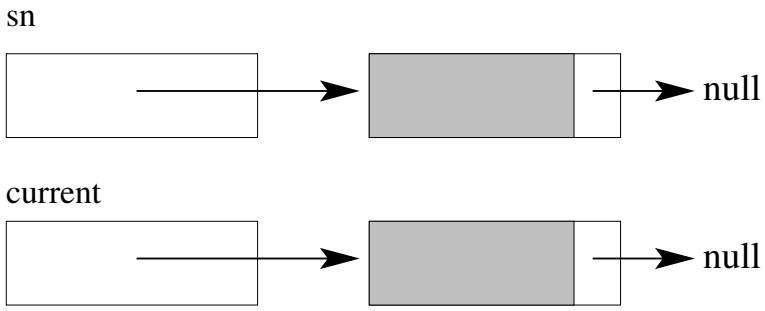


- `sn = new Node<E>();`

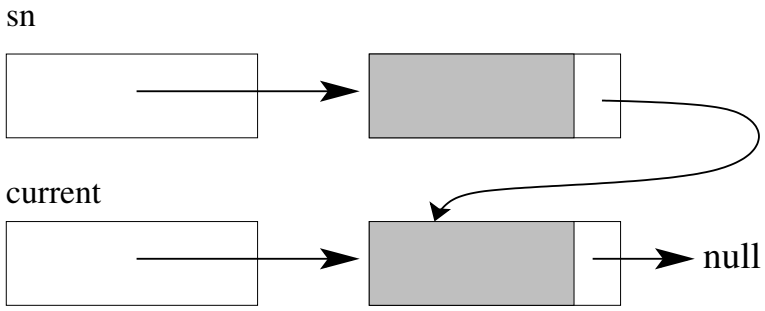


Joanna Klukowska
joannakl@cs.nyu.edu

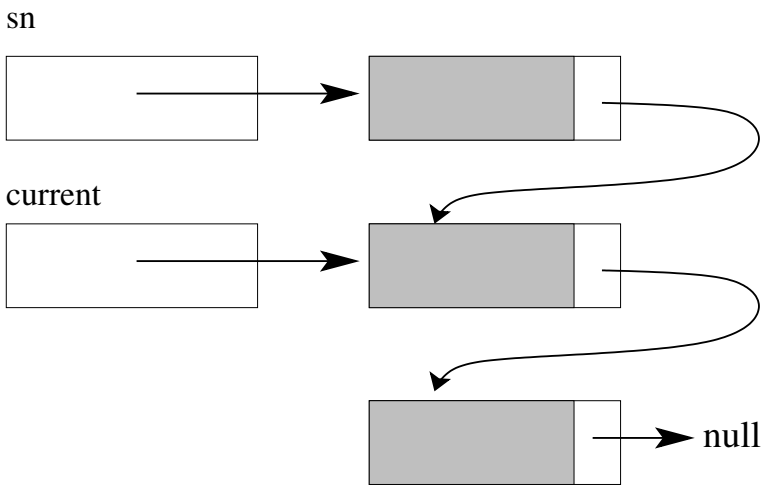
- `current = new Node<E>();`



- `sn.setNext( current );`

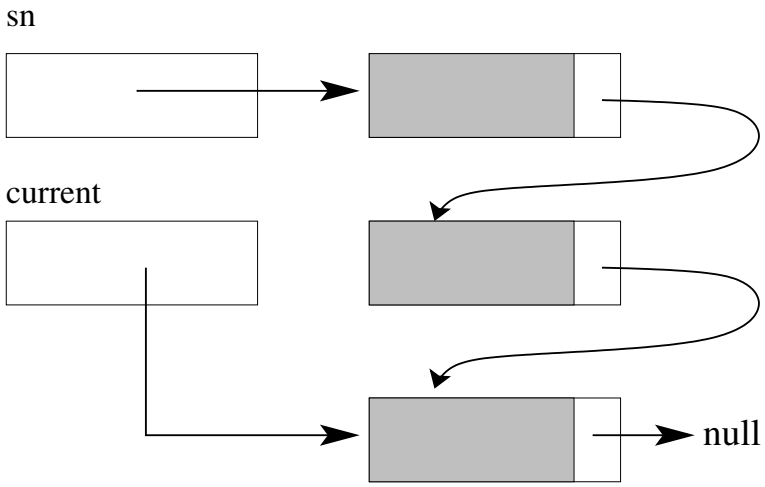


- `current.setNext( new Node<E>());`

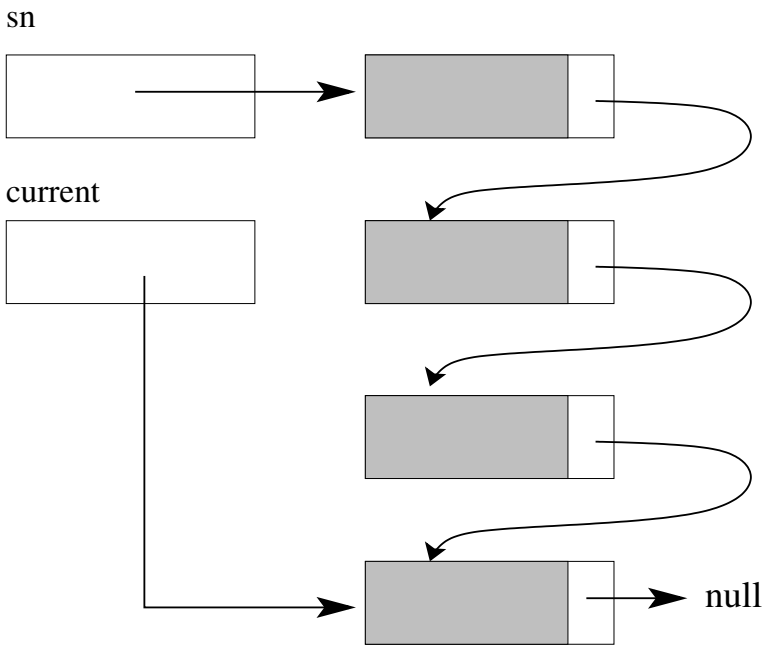


Joanna Klukowska
joannakl@cs.nyu.edu

- `current = current.getNext();`



- `current.setNext( new Node<E>());`
  `current = current.getNext();`



4 Linked-Based Implementation of a List

4.1 Linked Lists

A linked list can be simply thought of as a reference to the first node of a linked structure: from there we can follow the ”links” to get to every other node. Our first linked list class contains a reference to the first node - we call it **head**, and several methods that allow us to work with a list. Here is an example of a Java interface for a list.

```
1
2 public interface List<E> {
3   void insert ( E item );
4   void remove ( E item );
5   void clear ( );
6   boolean contains ( E item );
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
7   int indexOf ( E item );
8   E get ( int index );
9   int size ( );
10  String toString ();
11 }
```

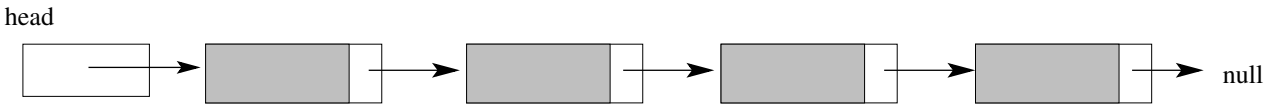
The next few sections discuss the way to implement some of the methods using the linked list approach.

The implementation details and algorithms below assume that we have a collection of connected nodes (as shown in previous section) with a reference to the first node called **head**.

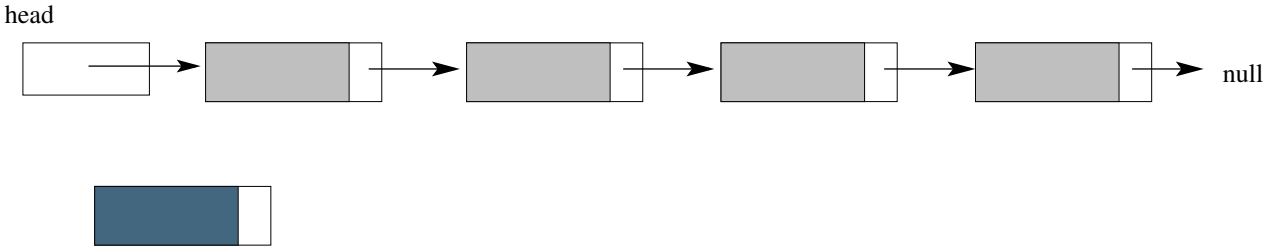
4.1.1 `insert` method

We want to insert a new node into an existing list of nodes. The steps differ slightly depending on where the new node needs to be inserted: front, back or middle of the list.

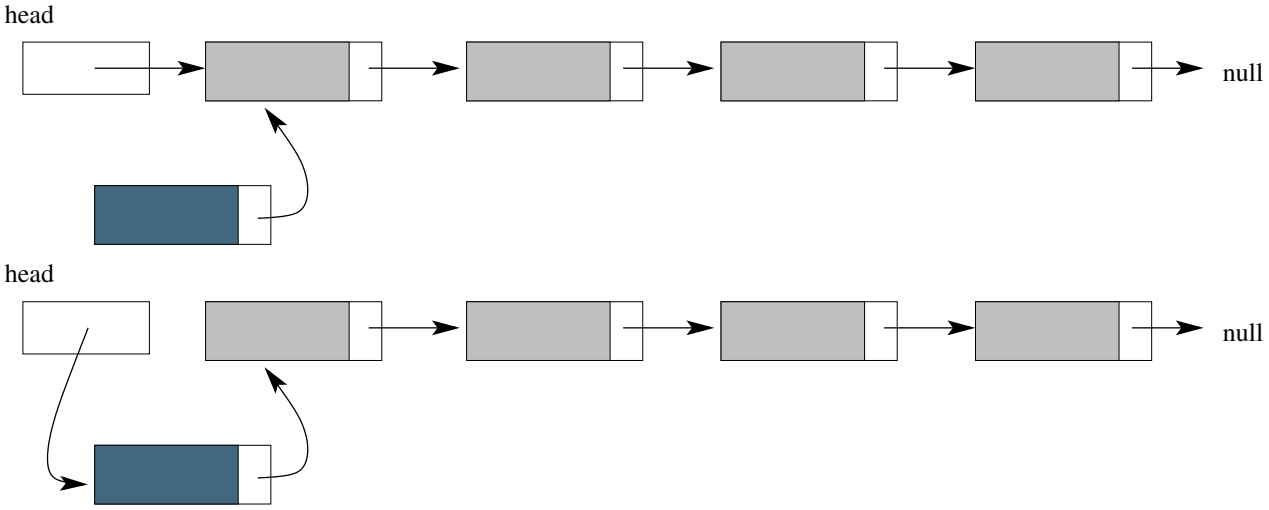
Inserting an element at the beginning Assume that we have a linked list with four nodes:



We want to add a new node at the beginning of this list, i.e., in front of the very first node.



This requires making the new node point to the current first element and then changing where the **head** reference points to.



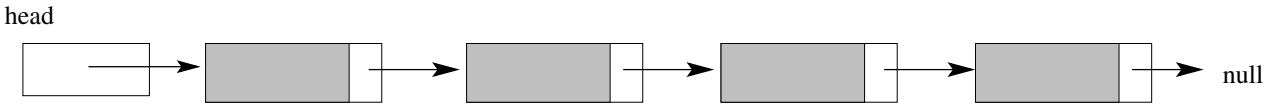
WARNING: If you change where the **head** reference points to first, then the rest of the list will be lost!

Joanna Klukowska
joannakl@cs.nyu.edu

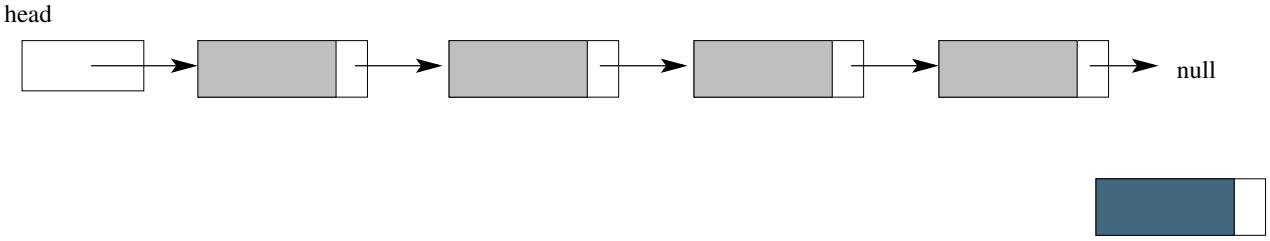
These steps in code:

```
//create new node
Node<E> newNode = new Node<E> ( );

//make the new node point to the current first node
newNode.setNext(head);

//make the head reference point to the new first node
head = newNode;

//update the size variable (if it exists)
size++;
```

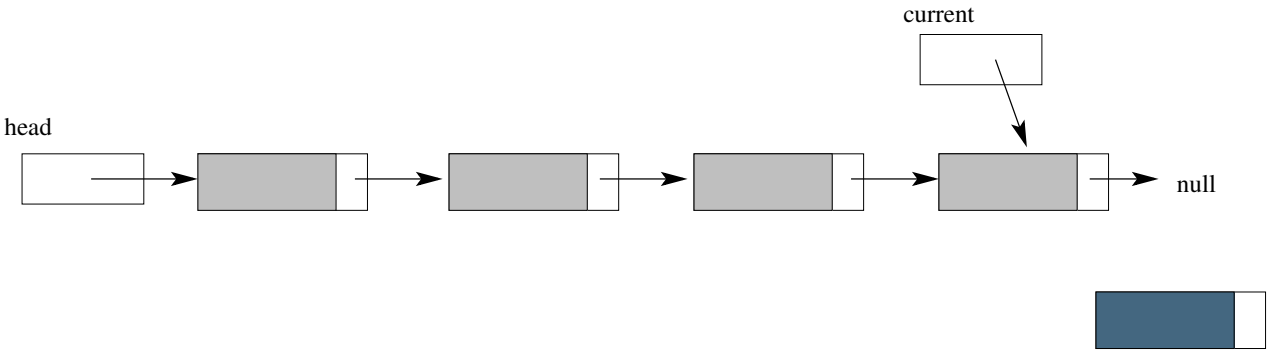
Inserting an element at the end Again, assume that we start with a linked list with four nodes:



We want to add a new node at the end of this list, i.e., the current last node should point to our new node.

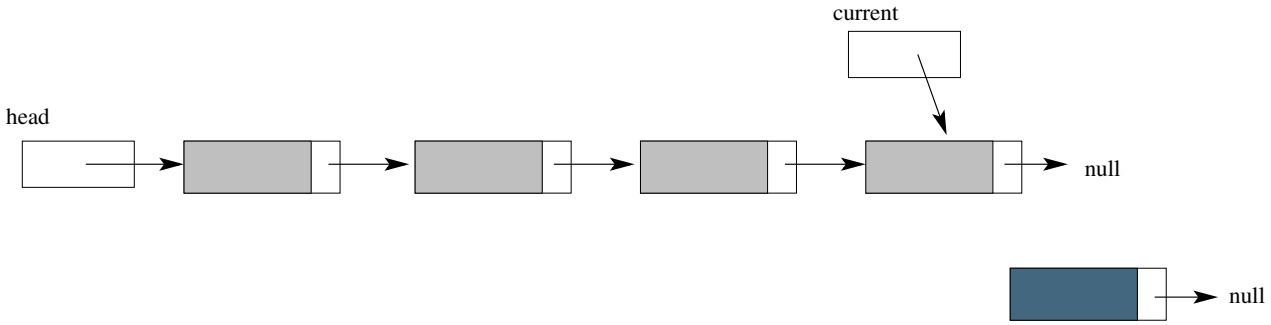


Since, we only have a reference to the first node, we need to find the last node (easy to find since it is the only one pointing to `null`). So we have a new reference, called `current` that now points to the last node.

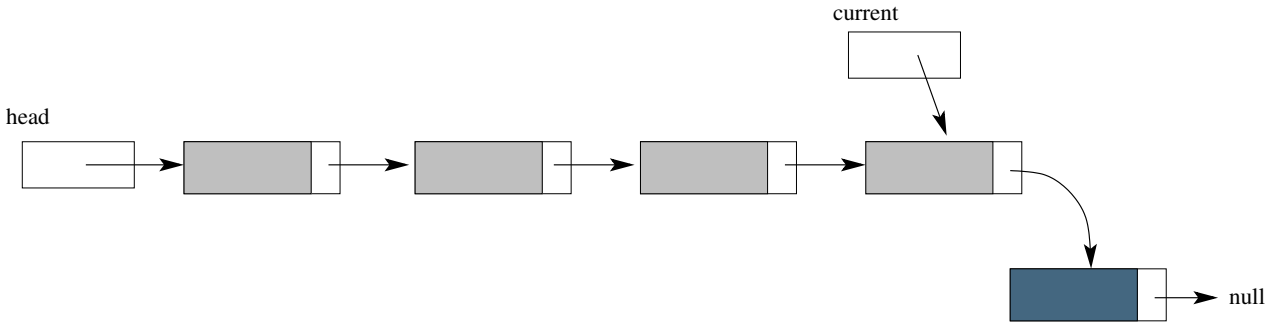


Joanna Klukowska
joannakl@cs.nyu.edu

Our new node should point to **`null`**, since it will become the last node in the list,



and the current last node should point to the new node.

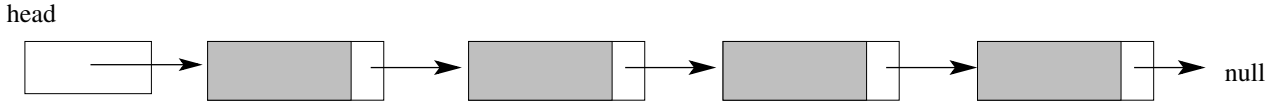


In this case, the order of the last two steps does not matter. (In fact, if the **`Node<E>`** constructor automatically sets the **`next`** reference to **`null`** that step can be omitted.)

These steps in code (this assumes that data fields in the **`Node<E>`** class are public to simplify the logic of the code):

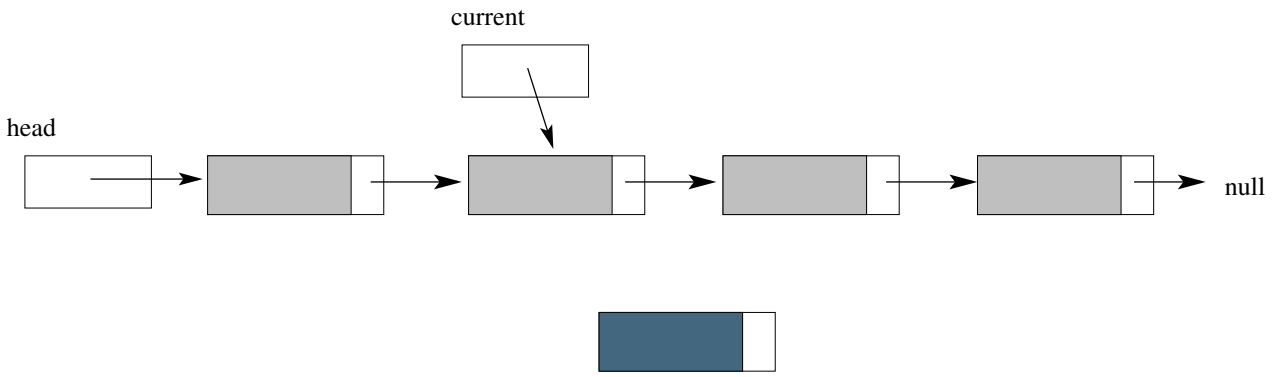
```
//create new node
Node<E> newNode = new Node<E> ( );

//create the current reference and point it to the last node
Node<E> current = head;
while (current.next != null )
  current = current.next;

//make the new node point to null
newNode.next = null;

//make the last node point to the new last node
current.next = newNode;

//update the size variable (if it exists)
size++;
```

Inserting an element in the middle The previous two scenarios covered special cases. Now, we have the same initial linked list, but we want to add a node in the middle. At this point it does not matter how the actual location is selected, so let's add it after the second node.

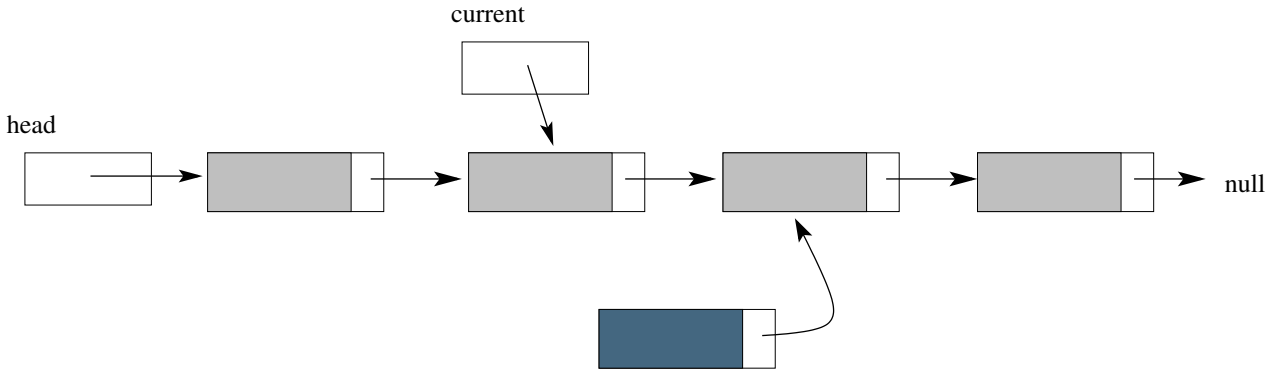


At this point it does not matter how the actual location is selected, so let's add it after the second node. We need the `current` reference to point to the node that should be before the new node after the insertion (the exact details of how to advance the current reference from head to that particular position depend on how the position is selected).

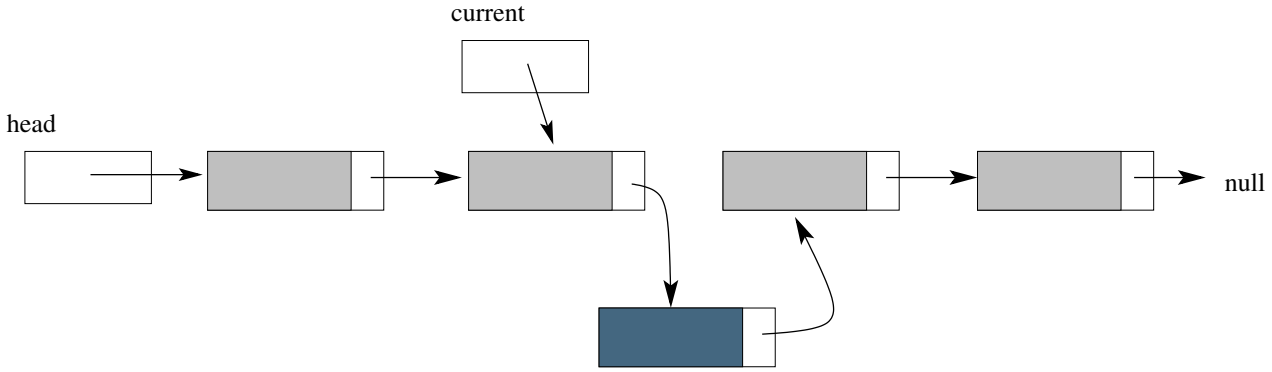
Joanna Klukowska
joannakl@cs.nyu.edu



As with inserting at the front, we need to make sure not to disconnect the rest of the list during the insertion. The new node has to point to its successor



and then its predecessor is pointed to the new node.



These steps in code:

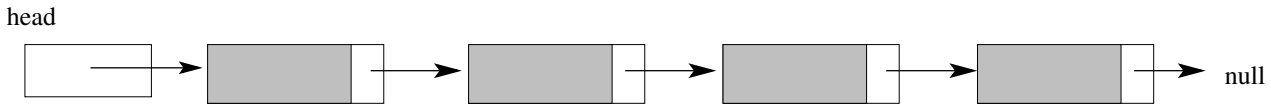
```
//create new node
Node<E> newNode = new Node<E> ( );

//create the current reference and advance it to
//the node after which we want to insert
Node<E> current = head;
... //depends on specifics of the problem

//make the new node point to its successor
newNode.next = current.next;

//make the current node point to the new node
current.next = newNode;
```

Joanna Klukowska
joannakl@cs.nyu.edu

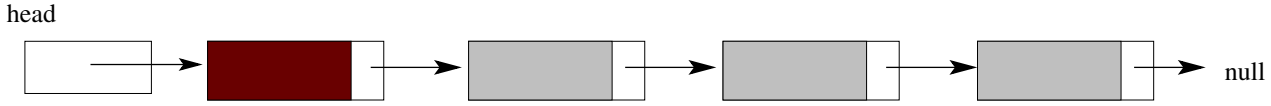
4.1.2 `remove` method

As with the insertion, the removal of a node is slightly different depending on where the node is located in the list.

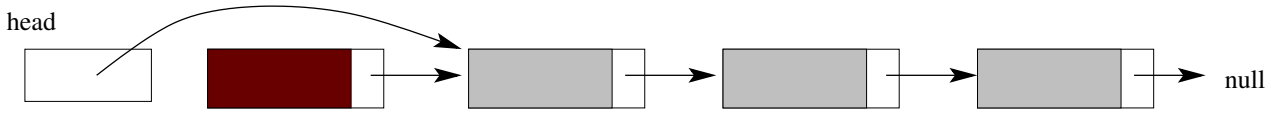
Removing an element at the beginning We start with our four-node linked list



and we want to remove the very first node (the one pointed to by the `head` reference).



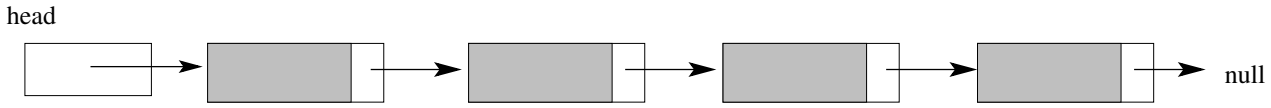
This involves pointing the head reference to the second node. Once the first node is no longer pointed to by anything, it becomes unreachable and memory used by it will be returned to available memory set by Java garbage collector.



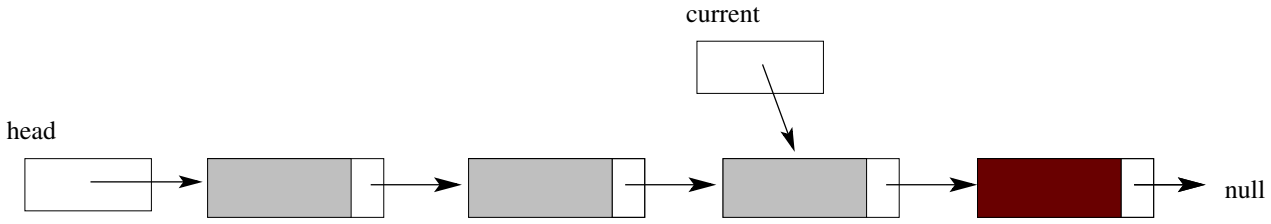
These steps in code:

```
//point the head reference to the second node
if ( head != null )
    head = head.next;

size--;
```

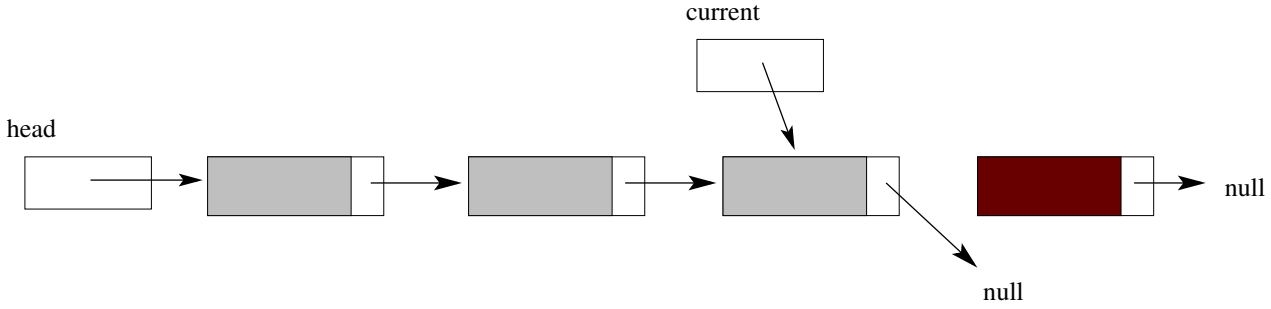
Removing an element at the end We start with our four-node linked list



and we want to remove the very last node (the one that points to `null`). This means that we first need to find the node right before it ...



... and make it point to `null` (make it the last node).



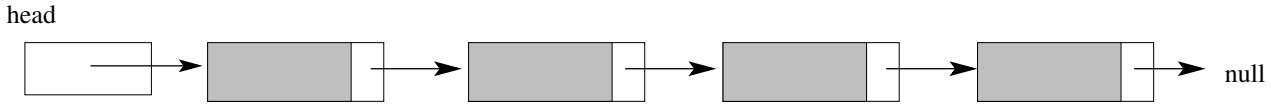
Joanna Klukowska
joannakl@cs.nyu.edu

These steps in code:

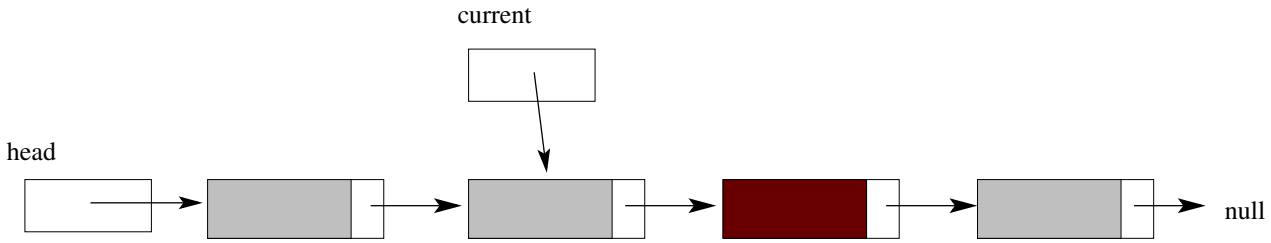
```
//create a current reference and advance it to the one
//before the last node
Node<E> current;
current = head;

//assume that there are at least two nodes in the list
//(otherwise we are deleting the first node)
while ( current.next.next != null )
  current = current.next;

//disconnect the last node by pointing the one
//before it to null
current.setNext(null);

size--;
```

Warning: the above code is not going to work if the list is very short. What might happen if the list has only one node?

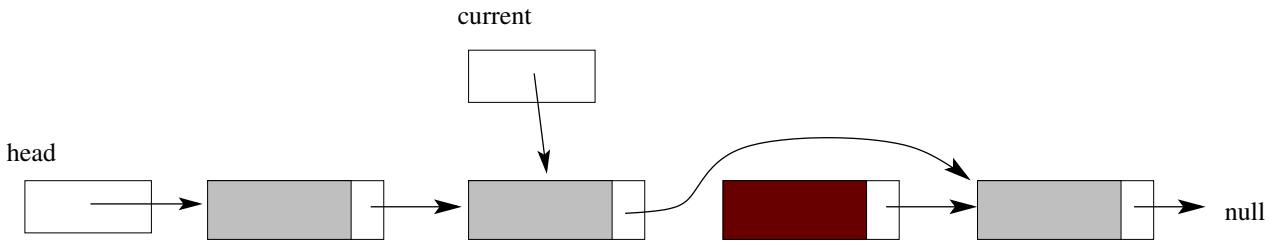
Removing an element in the middle We start with our four-node linked list



and we want to remove a node in the middle (not the first one or the last one). We need the `current` reference to point to the node before the one that needs to be removed (how this is done depends on the specifics of how the node to be deleted is selected).



The node pointed to be current needs to point to the node "one away" in the list (the one that the node to be deleted is pointing to).



These steps in code:

```
//create a current reference and advance it to the one
//before the node to be deleted
Node<E> current;
current = head;

... //depends on specifics of the problem

//connect the node pointed to by current with the node
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
//pointed to be the node being deleted
current.setNext(current.getNext().getNext() );

size--;
```

4.1.3 `clear` method

The `clear()` method of the list ADT empties the list. This can be accomplished by simply pointing the **`head`** reference of our list to **`null`**.

4.1.4 `contains` method

The contains() method determines if a particular data item is in the list or not. With linked lists the data items are stored inside the nodes. When implementing **`contains`** method we need to make sure that we are "looking at" the actual data items, not at the nodes (we need to look inside the boxes).

Here is an algorithm for searching in the linked list:

```
make current reference equal to head

while current != null
  check if current.data is what we are after
  if yes
    return true
  advance current to the next node

if we reach this point we did not find what we were after so return false
```

4.1.5 `indexOf` method

The **`indexOf`** method returns an integer indicating the position of an element in the list (starting with zero as the first position).

The implementation of this method is very similar to the implementation of **`contains`** method: we need to locate the element and return its position rather than returning Boolean value indicating it was found or not.

Here is an algorithm for searching in the linked list and returning the position information:

```
make current reference equal to head

set index to zero

while current != null
  check if current.data is what we are after
  if yes
    return the index
  advance current to the next node
  increment the index by one

if we reach this point we did not find what we were after so return  -1 to indicate that the iter
```

Joanna Klukowska
joannakl@cs.nyu.edu

4.1.6 `get` method

The `get` method returns the data element of the list at a particular location, call it `index`. Since we only have a reference to the first node, we need to jump over the first `index-1` nodes and then return the data element in the `index`$^{\text{th}}$ node.

Here is an algorithm for finding and returning `index`$^{\text{th}}$ node in a linked list:

```
make current reference equal to head
set counter to zero
while current != null and counter != index
  increment the counter by one
if counter == index
  return current.data
otherwise (we reached the end of the list)
  return null
```

4.1.7 `size` method

The most efficient implementation of this method would make use of an auxiliary data field that keeps track of the number of elements in the list: its value is adjusted on every deletion and insertion. The method then simply returns a value of this variable.

But as an exercise we will develop this method to compute a size of a list in two different ways: iteratively and recursively.

Iterative algorithm The details of an iterative algorithm were discussed in previous parts of this section. We need to move through all the nodes incrementing a counter until we find a node that is pointing to null.

Here is an iterative algorithm for determining the size of a linked list:

```
make current reference equal to head
set counter to zero
while current != null
  increment the counter by one
return the value of counter
```

4.2 Circular Singly Linked List

This structure is still singly linked (only forward references from each node), but the last node, instead of "pointing" to **`null`**, points to the first node. This requires that all the methods that need to traverse the entire list have a different loop termination condition or base case for recursion. They cannot look for the **`null`** reference. Instead, the end of the list is indicated by a node that references the same node as the **`head`**.

WARNING: the implementation that looks for the end node which has the same reference as the **`head`** reference has to

- Make sure that it is really the same node that the **`head`** is pointing to, NOT that it just contains the same data.
- Make sure that such node is found after traversing the whole list, NOT when the **`current`** is initialized to **`head`** reference.

What would be an application of a circular linked list?

4.3 Doubly Linked List

The node used in a doubly linked list has references to both its successor and its predecessor.

```
class Node <E> {
  T data;
  Node<E> next;
  Node<E> prev;
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
  //methods
}
```

The first node is pointed to by a **head** reference, the last node is pointed to by a **tail** reference. The first node has **prev** pointing to **null** and the last node has next pointing to **null**.

Java's LinkedList¡E¿ class http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html provides an implementation of a doubly linked list.