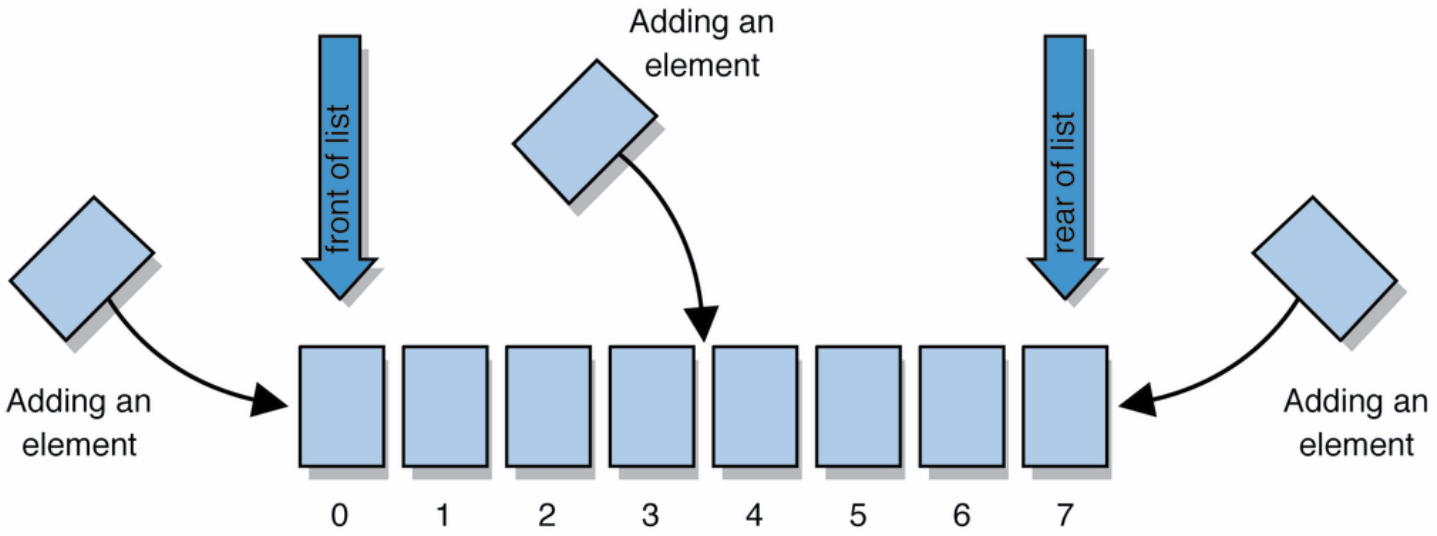
Lists

List

- A collection storing an ordered sequence of elements
  - Each element is accessible by a 0-based **index**
  - A list has a **size** (number of elements that have been added)
  - Elements can be added to the front, back, or elsewhere



Idea of a List

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

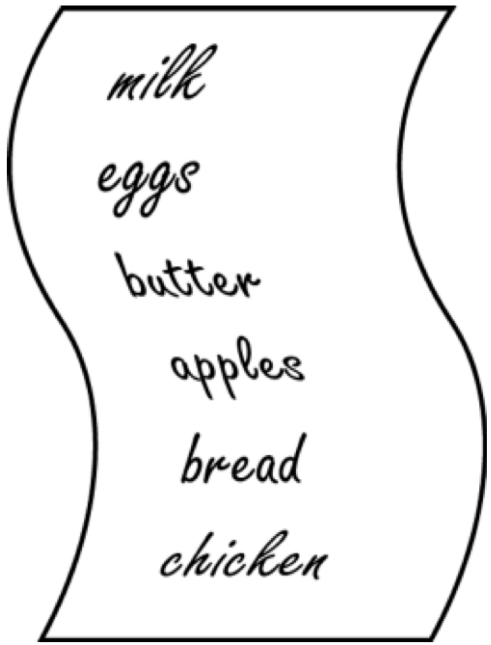
[]

- You can add items to the list.
  - The default behavior is to add to the end of the list.

  [milk, eggs, butter, apples, bread, chicken, meat]

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.

- For example a shopping list!



ADT List operations

`add(`**value**`)`	appends value at the end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`contains(`**value**`)`	returns true if given value is found somewhere in this list

List in Java Collection API

An example, I got from:
https://beginnersbook.com/2013/12/java-arraylist/

```
ArrayList<String> names = new ArrayList<String>();

/*This is how elements should be added to the array list*/
names.add("Ajeet");
 names.add("Harry");
names.add("Chaitanya");
names.add("Steve");
names.add("Anuj");

/* Displaying array list elements */
System.out.println("Currently the array list has "+ names.size() + " elements");

/*Add element at the given index*/
names.add(0, "Rahul");
names.add(1, "Justin");

/*Remove elements from array list like this*/
names.remove("Chaitanya");
names.remove("Harry");

System.out.println("Current array list is:"+ names);
```

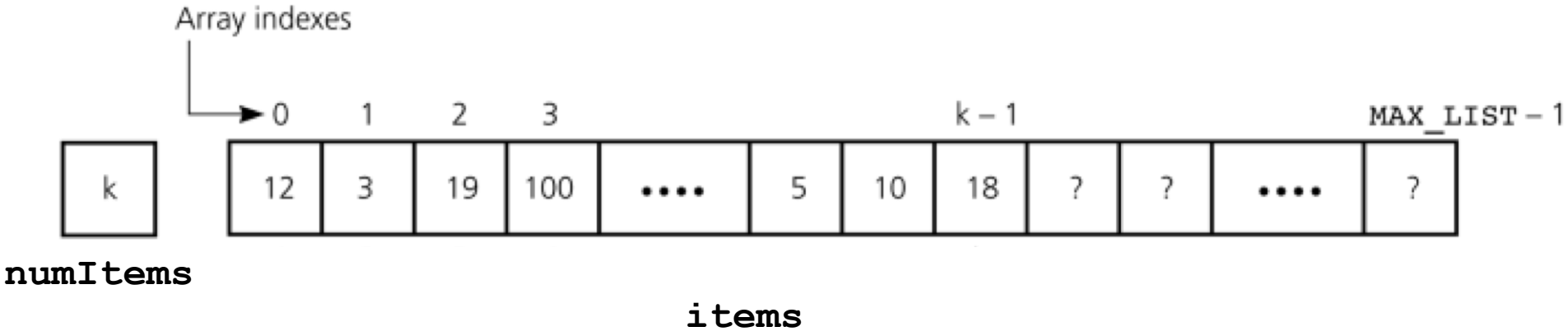
Implementation of List ADT

Array-based Lists

- The natural choice is using *arrays*
  - Both an array and a list identifies their items by number (index)
  - Access to an index is fast: k-th item of list will be stored in index k-1 of the array

Array-based Lists

- Two fields
  - An array (let's call it `items`)
  - An integer variable which keeps the number of items stored in the list (let's call it `numItems`)
    - Note that this variable can be used to find the index of the last item in the list (`numItems-1`)



Array-based Lists

```
public class ArrayBasedList {
   private static int MAX_LIST = 50;
   private int items[];
   private int numItems;
   public ArrayBasedList(){
          items = new int[MAX_LIST];
          numItems = 0;
   }
   public boolean isEmpty(){
      return (numItems == 0);
   }
   public int size() {return numItems;}
   public void add(int index, int item) throws
        IndexOutOfBoundsException {[...]}
   public int get(int index) throws IndexOutOfBoundsException {[...]}
   public int remove(int index) throws NoSuchElementException{[...]}
   public void clear(){[...]}
}
```

Check the source code of ArrayBasedList

Use exceptions!

Using the List ADT

```
public static void main( String [ ] args ) {
      ArrayBasedList list = new ArrayBasedList();
      list.add(0, 5);
      list.add(0, 4);
      list.add(0, 3);
      list.add(0, 2);
      list.add(0, 1);
      list.remove(3);
      System.out.println("index 3 is: "+list.get(3));

}
```

```
int x = list.items[0];  //illegal
```

violates the terms of abstraction

Array-based Lists

- The natural choice is ***arrays***
  - Both an array and a list identifies their items by number (index)
  - Access to index is fast: k-th item of list will be stored index k-1 of the array
- Can you think of any disadvantage?

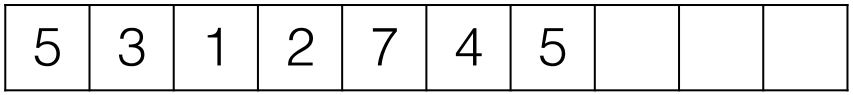
Array-based Lists

- Disadvantages
  1. Insert an item at the beginning or middle
     - Time proportional to the length of array (O(n))

```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages
  1. Insert an item at the beginning or middle
     - Time proportional to the length of array (O(n))

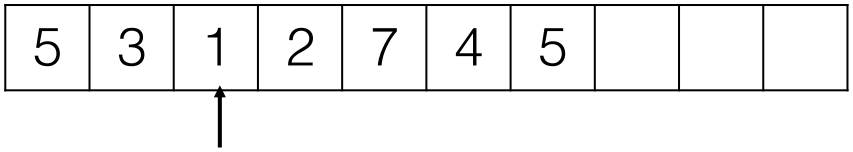



```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages
  1. Insert an item at the beginning or middle
     - Time proportional to the length of array (O(n))

insert 6 in index 2
add(2,6);

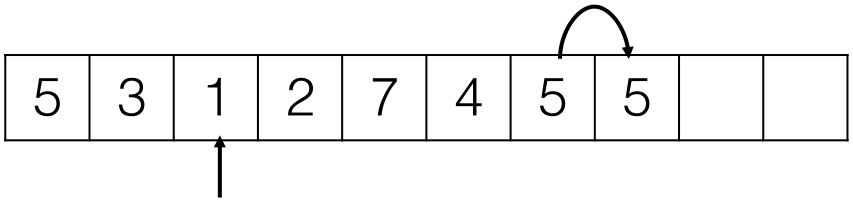



```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages
  1. Insert an item at the beginning or middle
     - Time proportional to the length of array (O(n))

insert 6 in index 2
add(2,6);

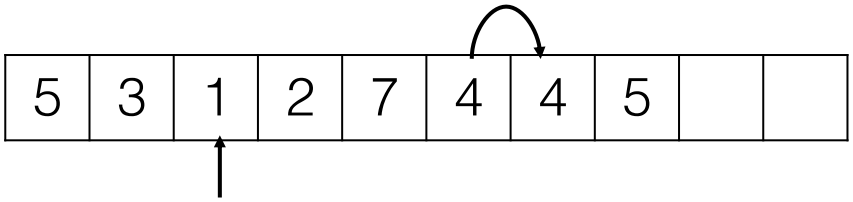



```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages
  1. Insert an item at the beginning or middle
     - Time proportional to the length of array (O(n))

insert 6 in index 2
add(2,6);

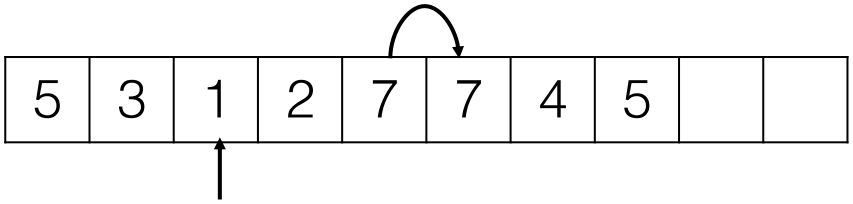



```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages
  1. Insert an item at the beginning or middle
     - Time proportional to the length of array (O(n))

insert 6 in index 2
add(2,6);

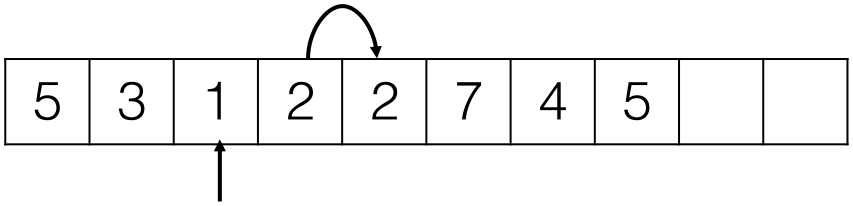



```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages
  1. Insert an item at the beginning or middle
     - Time proportional to the length of array (O(n))

insert 6 in index 2
add(2,6);

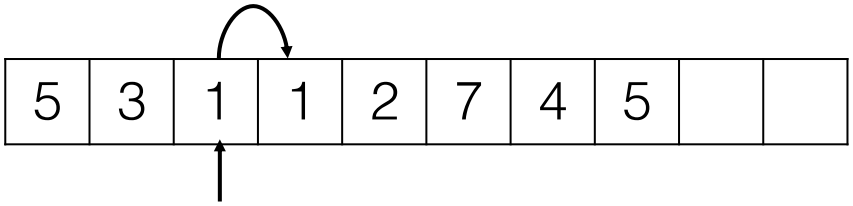



```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages
  1. Insert an item at the beginning or middle
     - Time proportional to the length of array (O(n))

insert 6 in index 2
add(2,6);

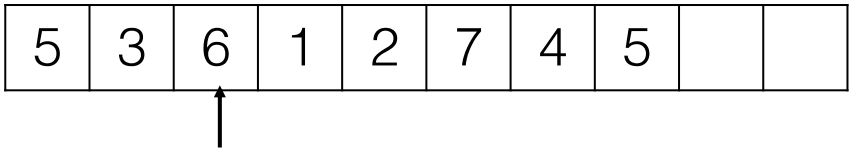



```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages
  1. Insert an item at the beginning or middle
     - Time proportional to the length of array (O(n))

insert 6 in index 2
add(2,6);

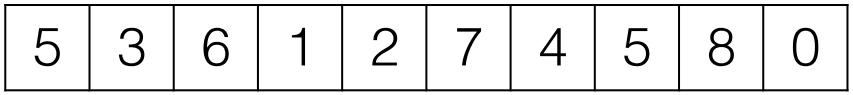



```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages

  2. Arrays have fixed length



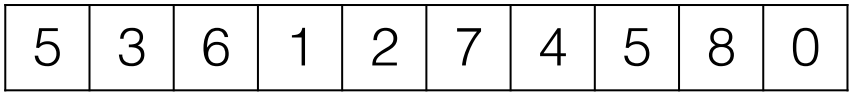

```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages

  2. Arrays have fixed length
     - What happened when the array is full?!

insert 9 in index 2
add(2,9);



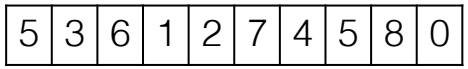

```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages

  2. Arrays have fixed length

     - What happened when the array is full?!



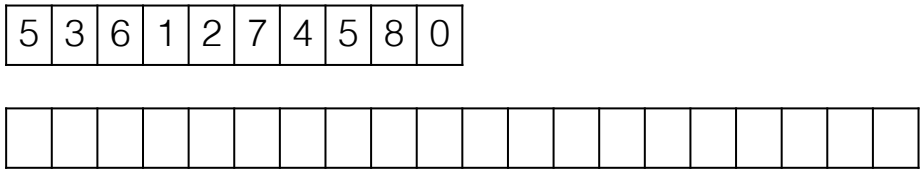

```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      // No room left in the array?
      if (size() == MAX_LIST) {
         int newArray[] = new int[2 * MAX_LIST];  // Allocate a new array, twice as long.
         for (int i = 0; i < numItems; i++)       // Copy items to the bigger array.
            newArray[i] = this.items[i];
         this.items = newArray;                   // Replace the too-small array with the new one.
         MAX_LIST *=2;
      }

      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages

  2. Arrays have fixed length

     - What happened when the array is full?!



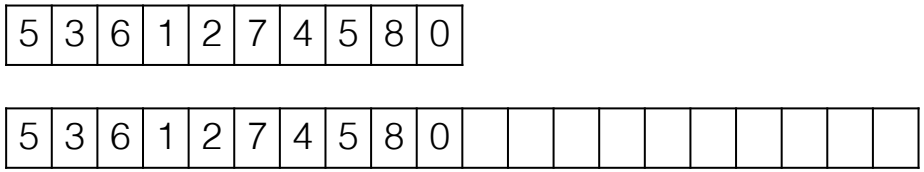

```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      // No room left in the array?
      if (size() == MAX_LIST) {
         int newArray[] = new int[2 * MAX_LIST];  // Allocate a new array, twice as long.
         for (int i = 0; i < numItems; i++)       // Copy items to the bigger array.
            newArray[i] = this.items[i];
         this.items = newArray;                   // Replace the too-small array with the new one.
         MAX_LIST *=2;
      }

      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Array-based Lists

- Disadvantages

  2. Arrays have fixed length

     - What happened when the array is full?!




```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      // No room left in the array?
      if (size() == MAX_LIST) {
         int newArray[] = new int[2 * MAX_LIST];  // Allocate a new array, twice as long.
         for (int i = 0; i < numItems; i++)       // Copy items to the bigger array.
            newArray[i] = this.items[i];
         this.items = newArray;                   // Replace the too-small array with the new one.
         MAX_LIST *=2;
      }

      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

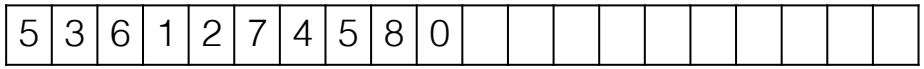
Array-based Lists

- Disadvantages

  2. Arrays have fixed length

     - What happened when the array is full?!

insert 9 in index 2
add(2,9);




```
public class ArrayBasedList {
   private static int MAX_LIST = 10;
   private int items[];
   private int numItems;

   public ArrayBasedList(){
      items = new int[MAX_LIST];
      numItems = 0;
   }

   public void add(int index, int item) throws IndexOutOfBoundsException {
      // No room left in the array?
      if (size() == MAX_LIST) {
         int newArray[] = new int[2 * MAX_LIST];  // Allocate a new array, twice as long.
         for (int i = 0; i < numItems; i++)       // Copy items to the bigger array.
            newArray[i] = this.items[i];
         this.items = newArray;                   // Replace the too-small array with the new one.
         MAX_LIST *=2;
      }

      if (index < 0 || index > numItems)
         throw new IndexOutOfBoundsException("index: "+index);

      for (int i = numItems-1; i >= index; i--)       // Shift items to the right.
         items[i + 1] = items[i];

      items[index] = item;
      numItems++;
   }
```

Generic Types (recap)

Check the source code of ArrayBasedList v1 and compare it to ArrayBasedList v0

```
public class ArrayBasedList<AnyType> {
   private static int MAX_LIST = 50;
   private AnyType items[];
   private int numItems;
   public ArrayBasedList(){
          items = (AnyType[]) new Object[MAX_LIST];
          numItems = 0;
   }
   public boolean isEmpty(){
      return (numItems == 0);
   }
   public int size() {return numItems;}
   public void add(int index, AnyType item) throws
        IndexOutOfBoundsException {[...]}
   public AnyType get(int index) throws IndexOutOfBoundsException {[...]}
   public AnyType remove(int index) throws NoSuchElementException{[...]}
   public void clear(){[...]}
}
```

Generic Types (recap)

- When constructing an `ArrayBasedList`, you must specify the type of elements it will contain between < and >.
- It allows the same `ArrayBasedList` class to store lists of different types.
  - But type should be a class (primitive types are not allowed)!

```
ArrayBasedList<String> groceryList = new ArrayBasedList<String>();
groceryList.add(0, "milk");
groceryList.add(0, "bread");
System.out.println("index 1 is: "+groceryList.get(1));
System.out.println("The whole list is: "+ groceryList);
```

!!NOTE!!

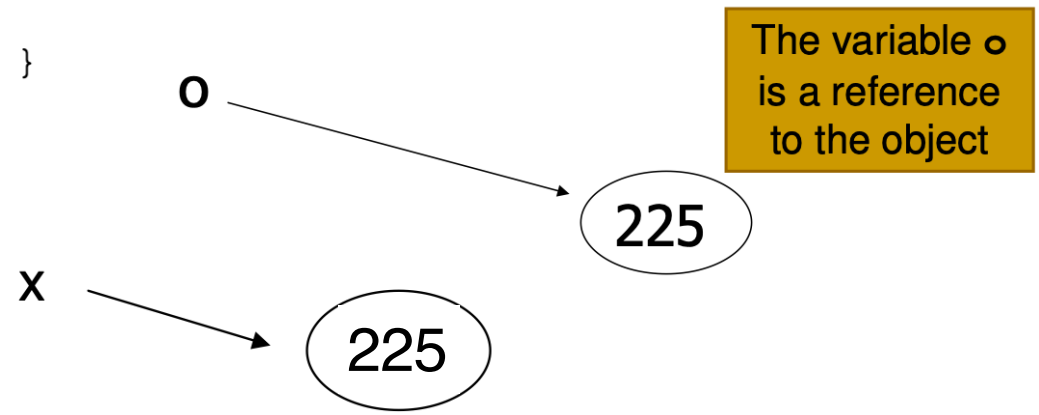
```
ArrayBasedList<int> list = new ArrayBasedList<int>();
// this is illegal! primitive types are not allowed
// rewrite it using class Integer
```

Comparison (recap)

- `== and !=`
  - for primitive types, == evaluates as true if the values are the same
  - for objects, == is true if the references are to the same object.

```
Public class aClass{
   aClass(int value){m=value;   }
   public int m;
}
aClass o = new aClass(225);
aClass x = new aClass(225);
```

x == o is false!!



Comparing Objects (recap)

- The == operator does not work well with objects.
- It only produces `true` when you compare an object to itself.

How can we add a methods like `indexOf()` or `contains()` to our `ArrayBasedList` class?

The `equals()` method

- The `equals` method compares the state of objects.

```
if (str1.equals(str2)) {
   System.out.println("the strings are equal");
}
```

- But if you write a new class (without defining `equals`), if you call `equals` method, it behaves like `==` (compare the references)

```
if (o.equals(x)) {  //false!!
   System.out.println("the objects are equal!”);
}
```

- This is the default behavior we receive from class `Object`.
- Java doesn't understand how to compare new classes by default.

The `equals()` method

- **You have to write the method `equals()` for any class you define!**

```
public class Point {
    private int x;
    private int y;
    …
```

```
public boolean equals(Object other) {
    Point tmp = (Point) other;
    return (tmp.x == this.x and tmp.y == this.y);
}
```

Method `indexOf()`

- Now assuming the `Object x` have the proper method `equals`, we can write the method `indexOf()` for our ArrayBasedList class:

```
public int indexOf(Object x) {
   for (int i = 0; i < size(); i++)
      if (x.equals(items[i]))
         return i;
   return -1;
}
```

ArrayList in Java Collection API

- ArrayList class is an implementation of ADT List in Java Collection API.
- Most of the developers choose ArrayList over Array as it's a very good alternative of traditional java arrays.
- ArrayList is a resizable-array implementation of the List interface. You do not need to be worried about size.
- It implements all optional list operations, and permits all types of elements, including `null`.

ArrayList in Java Collection API

`add(`**value**`)`	appends value at end of list
`add(`**index, value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index, value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

Check the Java API tutorials for the full list of methods:
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

ArrayList in Java Collection API

`addAll(`**list**`)` `addAll(`**index**`, `**list**`)`	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
`contains(`**value**`)`	returns true if given value is found somewhere in this list
`containsAll(`**list**`)`	returns true if this list contains every element from given list
`equals(`**list**`)`	returns true if given other list contains the same elements
`iterator()` `listIterator()`	returns an object used to examine the contents of the list
`lastIndexOf(`**value**`)`	returns last index value is found in list (-1 if not found)
`remove(`**value**`)`	finds and removes the given value from this list
`removeAll(`**list**`)`	removes any elements found in the given list from this list
`retainAll(`**list**`)`	removes any elements *not* found in given list from this list
`subList(`**from**`, `**to**`)`	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
`toArray()`	returns the elements in this list as an array

Check the Java API tutorials for the full list of methods:
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

ArrayList in Java Collection API

An example, I got from:
https://beginnersbook.com/2013/12/java-arraylist/

```
ArrayList<String> obj = new ArrayList<String>();

/*This is how elements should be added to the array list*/
obj.add("Ajeet");
 obj.add("Harry");
obj.add("Chaitanya");
obj.add("Steve");
obj.add("Anuj");

/* Displaying array list elements */
System.out.println("Currently the array list has following elements:"+obj);

/*Add element at the given index*/
obj.add(0, "Rahul");
obj.add(1, "Justin");

/*Remove elements from array list like this*/
obj.remove("Chaitanya");
obj.remove("Harry");

System.out.println("Current array list is:"+obj);
```

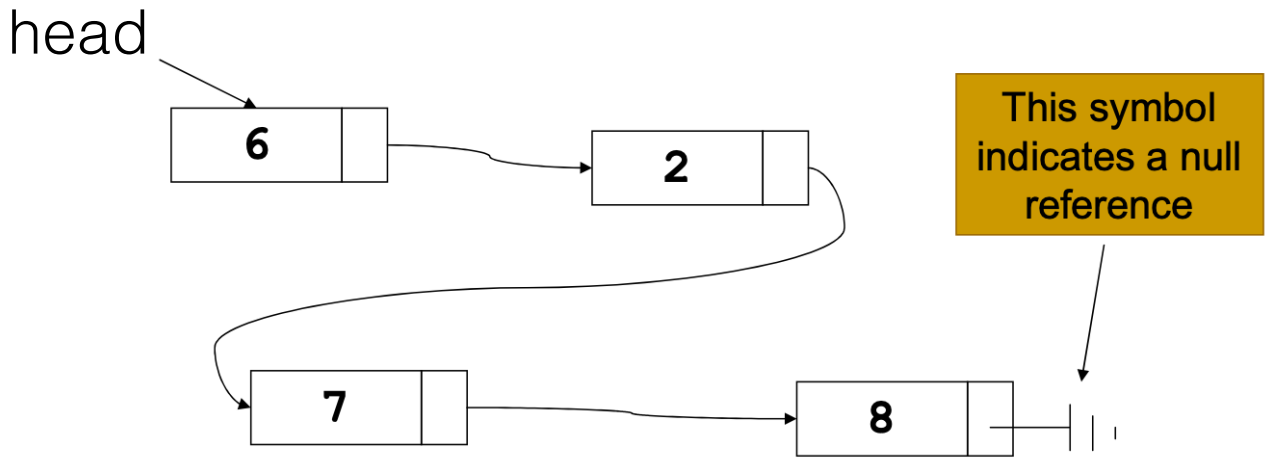
Linked Lists

Options for Implementing ADT List

- Arrays-based
  - Good
    - Fast random access of elements.
    - Efficient in terms of memory usage (almost no memory overhead)
  - Bad
    - Data must be shifted during insertions and deletions
    - Data must be copied during the resizing of the array
- Linked List based
  - Good
    - Is able to grow in size as needed
    - Does not require the shifting of items during insertions and deletions
  - Bad
    - Accessing an element does not take a constant time, it takes linear time!! (it affects insertion and deletion)
    - Need to store links to maintain the logical order of the elements (memory overhead).

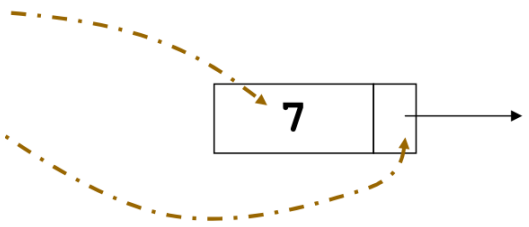
Linked Lists

- A linked list is a dynamic data structure made up of ***nodes***



Linked Lists

- A linked list is a dynamic data structure made up of ***nodes***
- Each node is a data structure that contains two components:
  - An item (that contains the data)
  - A reference that allows us to reach to the ***next*** node in the list

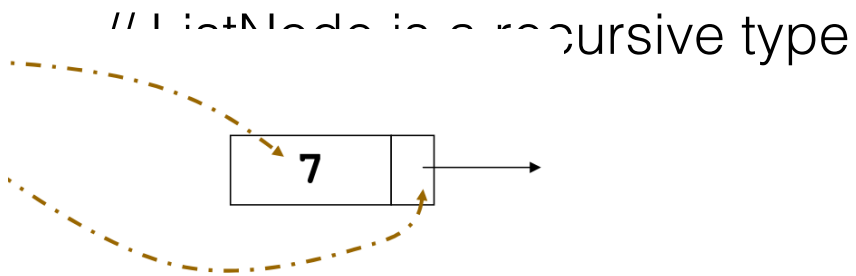


Linked Lists

- A linked list is a dynamic data structure made up of ***nodes***
- Each node has two components:
  - An item (that contains the data)
  - A reference to the ***next*** node in the list
- The physical location of the nodes in the main memory are random (i.e there is no relation between the logical order and the physical order of the elements of a linked list, unlike arrays).
- Therefore the logical order of the elements are maintained through the links.

Linked Lists

```
public class ListNode {
    public int item;
    public ListNode next;
}
```



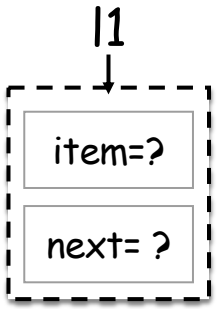
A node points to another node, so the reference must be of type `ListNode`

Linked Lists

```
public class ListNode {          // ListNode is a recursive type
    public int item;              // item can be any object type
    public ListNode next;         // Here we're using ListNode before
}                                 //   we've finished declaring it.
```

l1 is just a reference to a ListNode, it is not an object

```
ListNode l1;
```

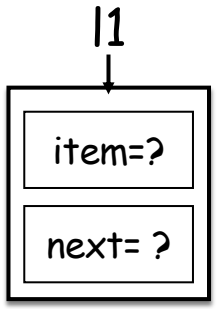


Linked Lists

```
public class ListNode {          // ListNode is a recursive type
    public int item;              // item can be any object type
    public ListNode next;         // Here we're using ListNode before
}                                 //   we've finished declaring it.
```

We have to use `new` to create an object

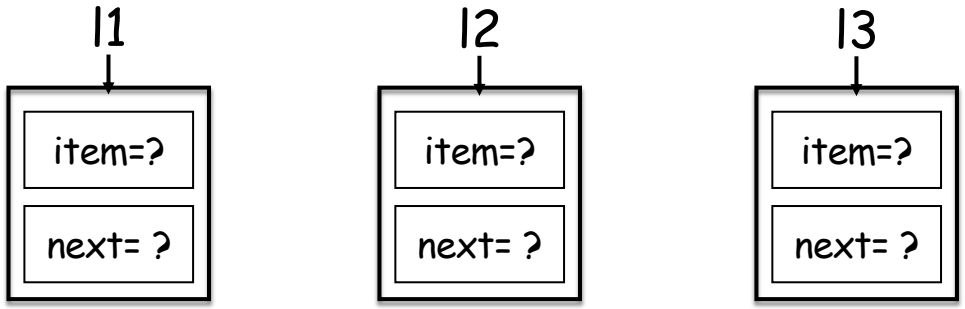
```
ListNode l1;
l1 = new ListNode();
```



Linked Lists

```
public class ListNode {          // ListNode is a recursive type
    public int item;              // item can be any object type
    public ListNode next;         // Here we're using ListNode before
}                                 //   we've finished declaring it.
```

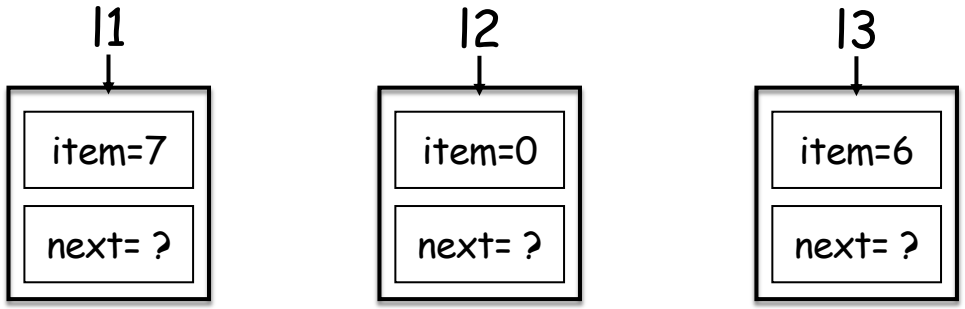
```
ListNode l1 = new ListNode(), l2 = new ListNode(), l3 = new ListNode();
```



Linked Lists

```
public class ListNode {          // ListNode is a recursive type
    public int item;              // item can be any object type
    public ListNode next;         // Here we're using ListNode before
}                                 //   we've finished declaring it.
```

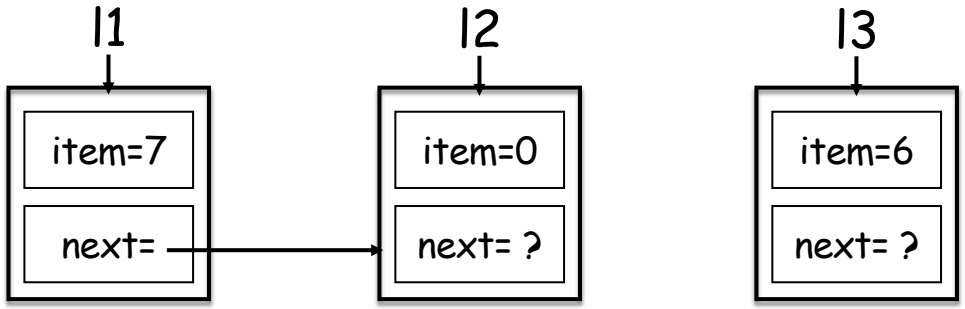
```
ListNode l1 = new ListNode(), l2 = new ListNode(), l3 = new ListNode();
l1.item = 7;
l2.item = 0;
l3.item = 6;
```



Linked Lists

```
public class ListNode {          // ListNode is a recursive type
    public int item;              // item can be any object type
    public ListNode next;         // Here we're using ListNode before
}                                 //   we've finished declaring it.
```

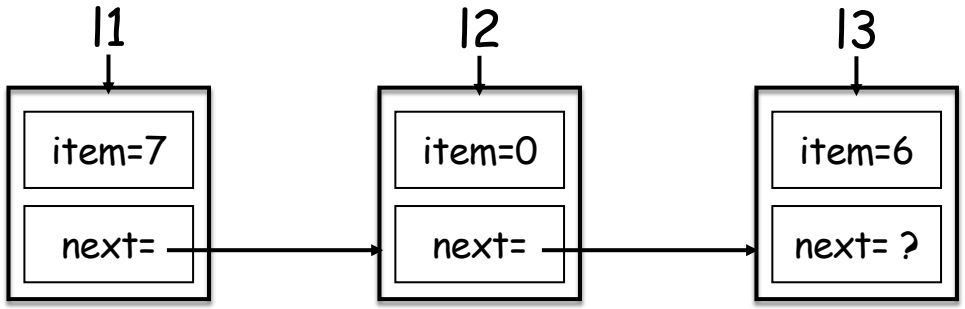
```
ListNode l1 = new ListNode(), l2 = new ListNode(), l3 = new ListNode();
l1.item = 7;
l2.item = 0;
l3.item = 6;
l1.next = l2;
```



Linked Lists

```
public class ListNode {          // ListNode is a recursive type
    public int item;              // item can be any object type
    public ListNode next;         // Here we're using ListNode before
}                                 //   we've finished declaring it.
```

```
ListNode l1 = new ListNode(), l2 = new ListNode(), l3 = new ListNode();
l1.item = 7;
l2.item = 0;
l3.item = 6;
l1.next = l2;
l2.next = l3;
```

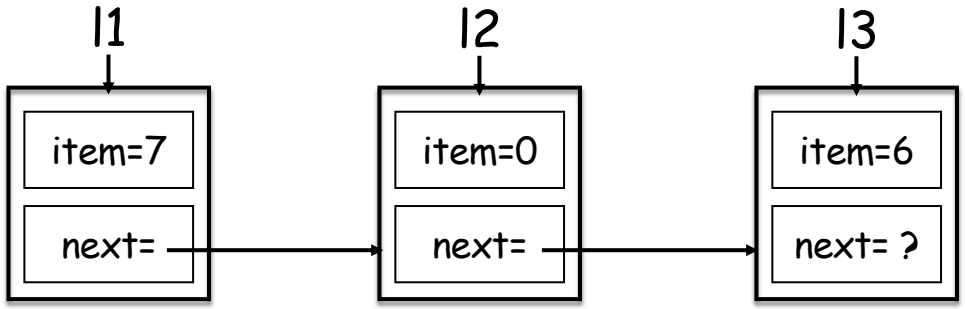


Linked Lists

```
public class ListNode {          // ListNode is a recursive type
    public int item;              // item can be any object type
    public ListNode next;         // Here we're using ListNode before
}                                 //   we've finished declaring it.
```

```
ListNode l1 = new ListNode(), l2 = new ListNode(), l3 = new ListNode();
l1.item = 7;
l2.item = 0;
l3.item = 6;
l1.next = l2;
l2.next = l3;
```

What happens to l3?



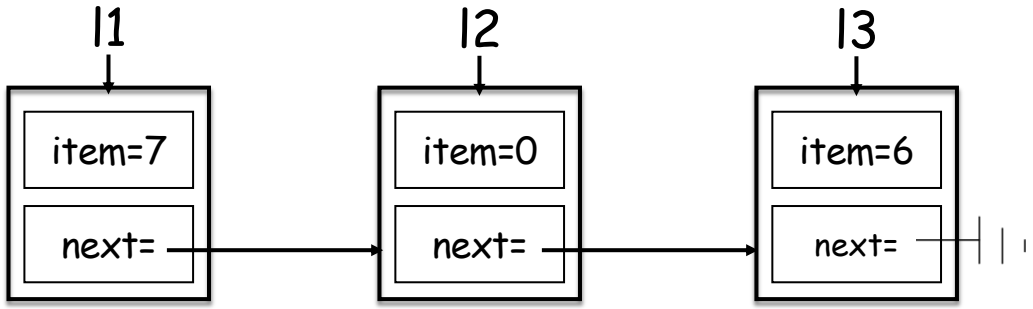
Linked Lists

```
public class ListNode {          // ListNode is a recursive type
    public int item;              // item can be any object type
    public ListNode next;         // Here we're using ListNode before
}                                 //   we've finished declaring it.
```

```
ListNode l1 = new ListNode(), l2 = new ListNode(), l3 = new ListNode();
l1.item = 7;
l2.item = 0;
l3.item = 6;
l1.next = l2;
l2.next = l3;
l3.next = null;
```

What happens to l3?

We should clarify that l3 is the last node in the list



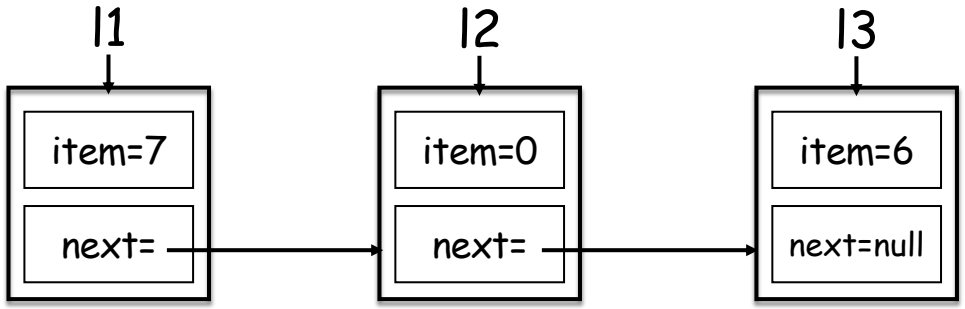
Linked Lists

```
public class ListNode {          // ListNode is a recursive type
    public int item;              // item can be any object type
    public ListNode next;         // Here we're using ListNode before
}                                 //   we've finished declaring it.
```

```
ListNode l1 = new ListNode(), l2 = new ListNode(), l3 = new ListNode();
l1.item = 7;
l2.item = 0;
l3.item = 6;
l1.next = l2;
l2.next = l3;
l3.next = null;
```

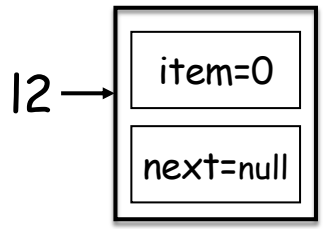
What happens to l3?

We should clarify that l3 is the last node in the list



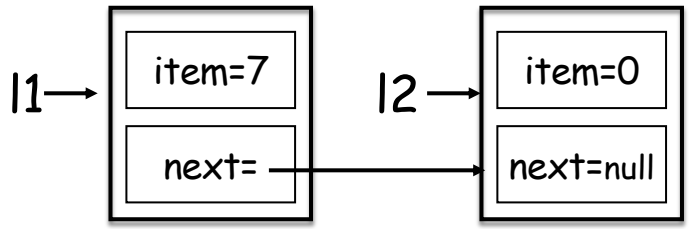
Constructors

```
public class ListNode {          // ListNode is a recursive type
    public int item;               // item can be any object type
    public ListNode next;

    public ListNode(int i, ListNode n) {   // long constructor
      item = i;
      next = n;
    }
    public ListNode(int i) {                  // short constructor
      this(i, null);
    }
}
ListNode l2 = new ListNode(0);
```



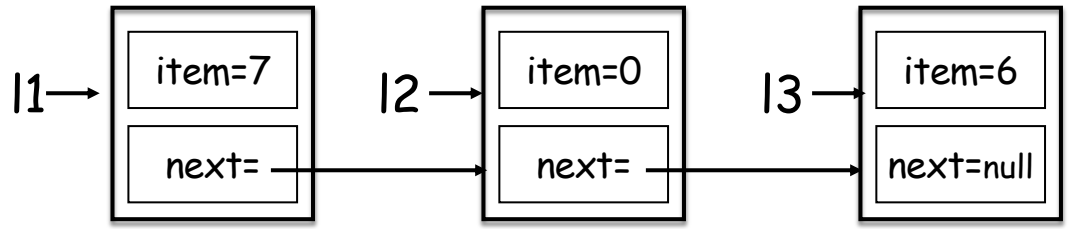
Constructors

```
public class ListNode {          // ListNode is a recursive type
    public int item;              // item can be any object type
    public ListNode next;         // Here we're using ListNode before

    public ListNode(int i, ListNode n) {   // long constructor
      item = i;
      next = n;
    }
    public ListNode(int i) {                // short constructor
      this(i, null);
    }
}
ListNode l2 = new ListNode(0);
ListNode l1 = new ListNode(7, l2);
```



Traversing a Linked Lists

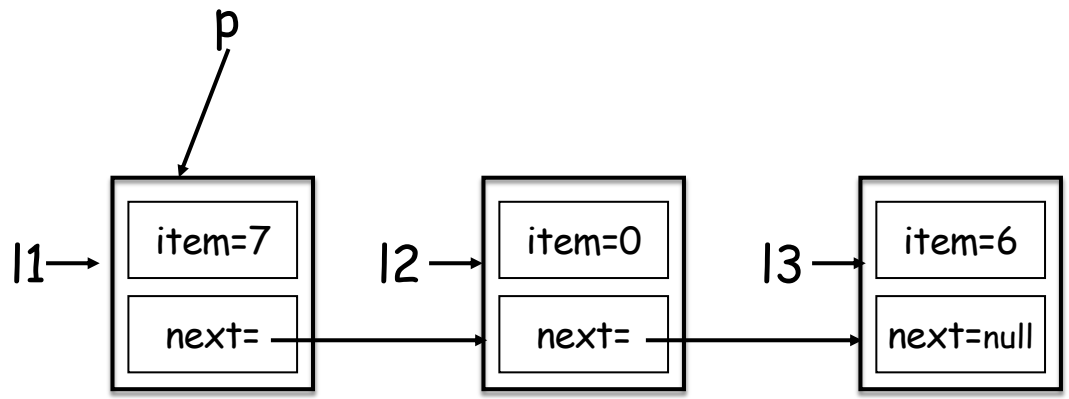
ListNode l1 = new ListNode(7, new ListNode(0, new ListNode(6)));



Traversing a Linked Lists

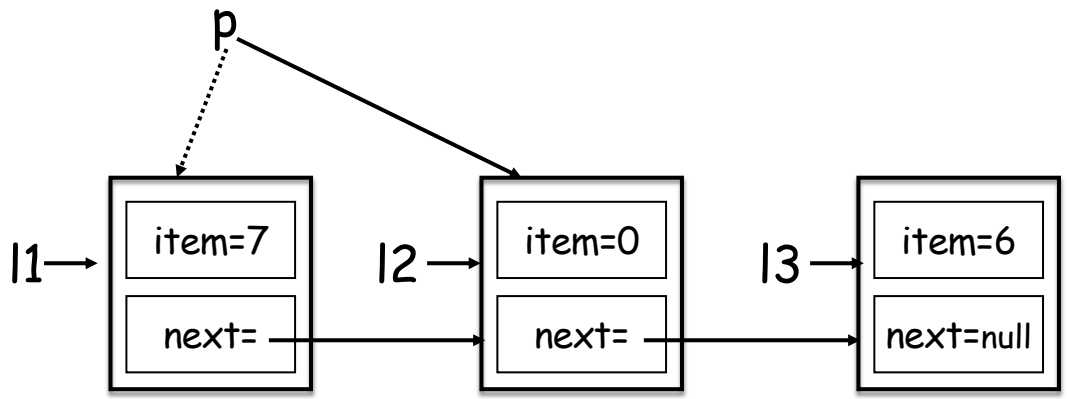
ListNode l1 = new ListNode(7, new ListNode(0, new ListNode(6)));

ListNode p = l1;



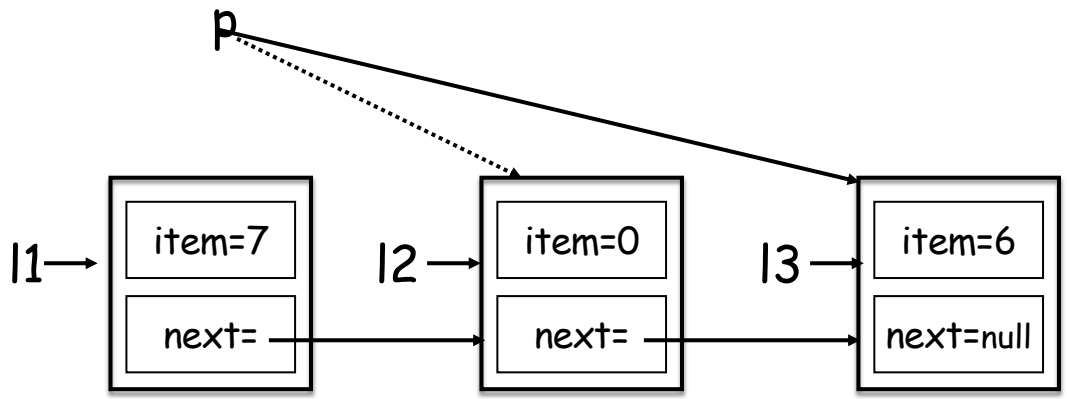
Traversing a Linked Lists

```
ListNode l1 = new ListNode(7, new ListNode(0, new ListNode(6)));

ListNode p = l1;
p = p.next;     // go to the next node
```



Traversing a Linked Lists

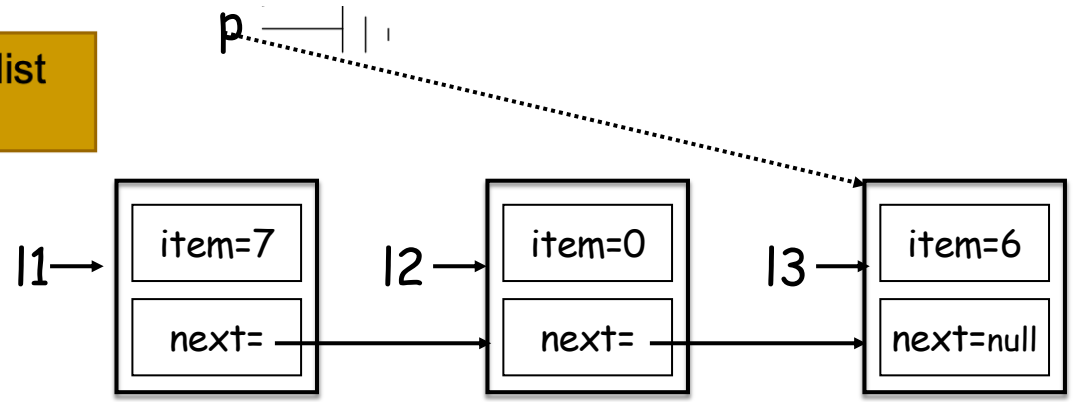
```
ListNode l1 = new ListNode(7, new ListNode(0, new ListNode(6)));

ListNode p = l1;
p = p.next;     // go to the next node
p = p.next;     // go to the next node
```



Traversing a Linked Lists

```
ListNode l1 = new ListNode(7, new ListNode(0, new ListNode(6)));

ListNode p = l1;
p = p.next;     // go to the next node (l2)
p = p.next;     // go to the next node (l3)
p = p.next;     // go to the next node (null)
```

Eventually, p hits the end of the list and becomes null.

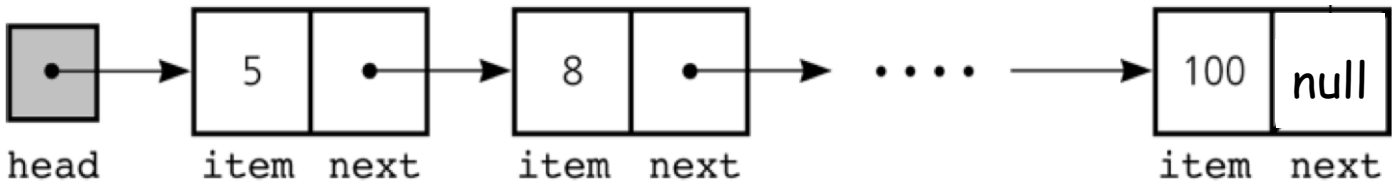


Traversing a Linked Lists

- The link field of the last node in the list is always null.
- Therefore, if you have the reference to the ***first*** node in the list you can traverse the list (i.e visit all the nodes) using a simple while loop

Linked Lists

- To manipulate a linked list we need to have references to the first node in the list
- Data field next in the last node is set to `null` therefore we know when we reached the end of the list.
- `head` reference variable always points to the first node of the list.
  - Reference to the list's first node always exists even if the list is empty



Linked Lists

- We need two classes
- Change `SListNode` to follow generics

```
//“S” stands for singly-linked
public class SListNode<AnyType> {
    public AnyType item;
    public SListNode<AnyType> next;
}
```

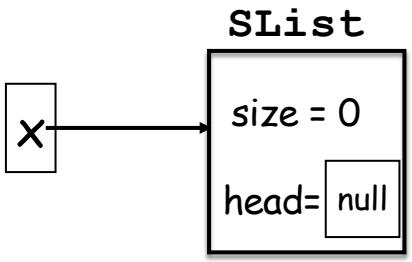
Linked Lists

- We need two classes
- Change `SListNode` to follow generics

```
//"S" stands for singly-linked
public class SListNode<AnyType> {
    public AnyType item;
    public SListNode<AnyType> next;
}


public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }
}
```

Linked List

```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }
}
```

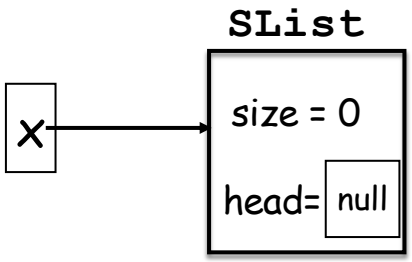


To create a null list, set head to null and size to 0

```
x =  new SList<String>();
```

Linked List

```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }
}
```




```
x =  new SList<String>();
x.addFirst("milk");
```

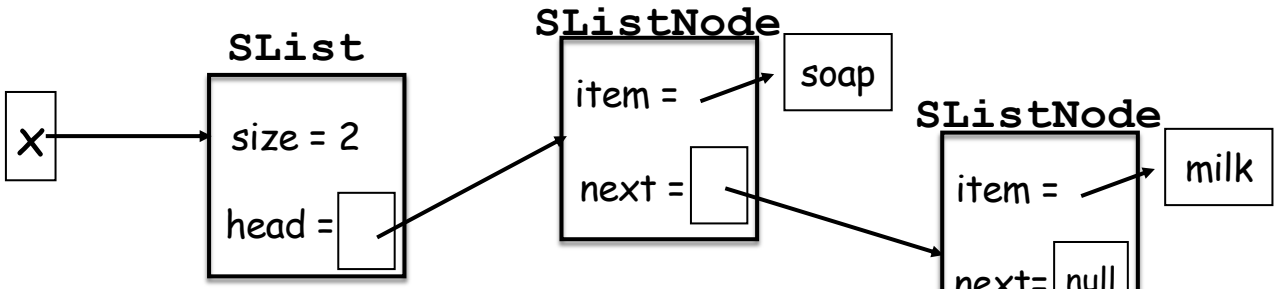
Linked List

```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }
}
```


```
x =  new SList<String>();
x.addFirst("milk");
```

Linked List

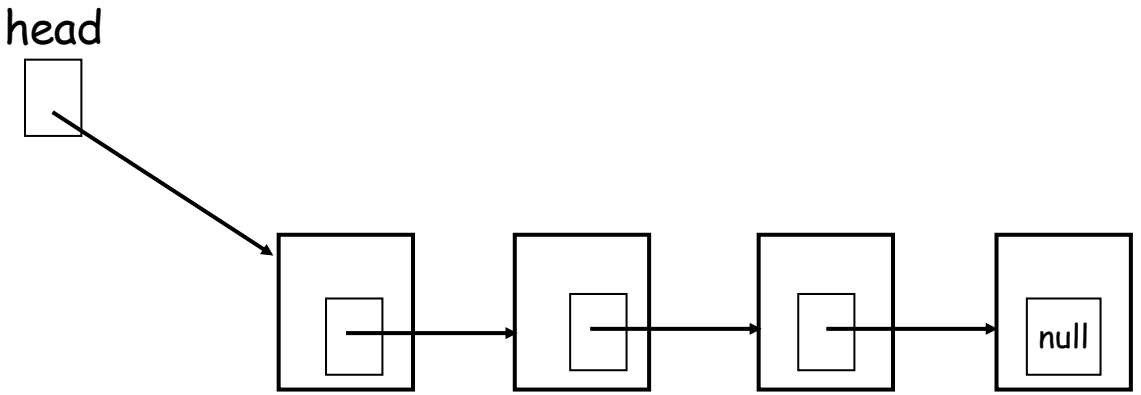
```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }
}
```




```
x =  new SList<String>();
x.addFirst("milk");
x.addFirst("soap");
```

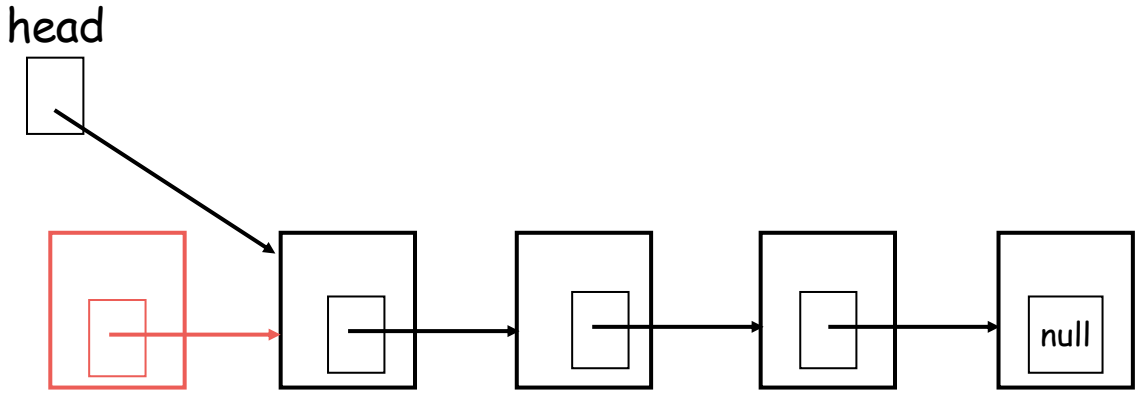
Linked List

```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }

  public void addFirst(AnyType item) {
     SListNode<AnyType> newNode = new SListNode<AnyType>(item, head);
     head = newNode;
     size++;
  }
}
```



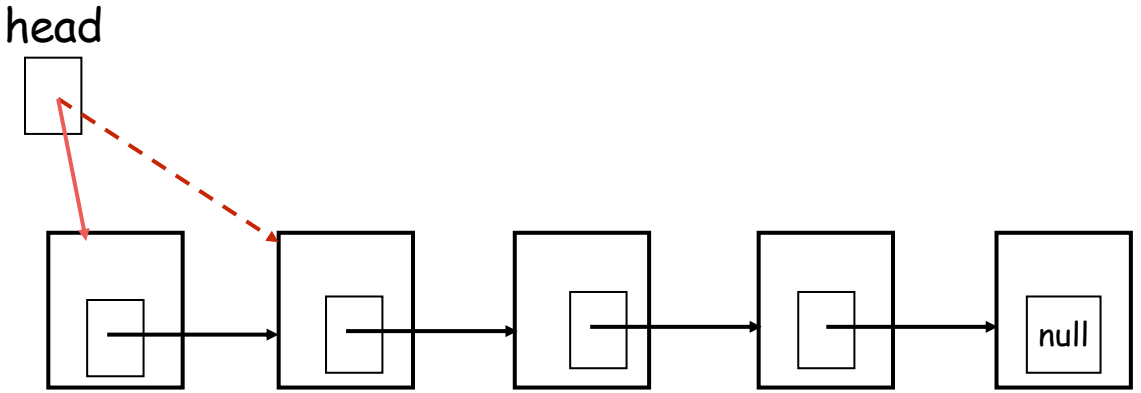
Linked List

```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }

  public void addFirst(AnyType item) {
     SListNode<AnyType> newNode = new SListNode<AnyType>(item, head);
     head = newNode;
     size++;
  }
}
```



Linked List

```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }

  public void addFirst(AnyType item) {
    SListNode<AnyType> newNode = new SListNode<AnyType>(item, head);
    head = newNode;
    size++;
  }
}
```

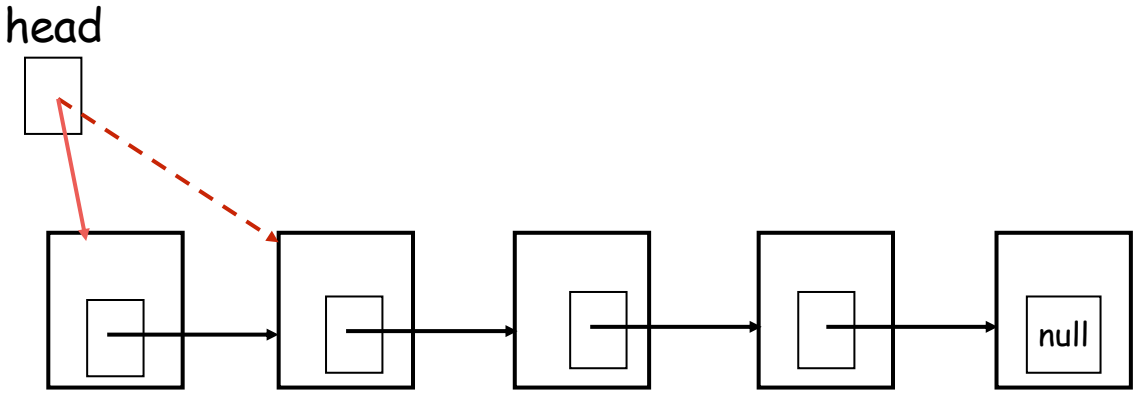


Linked List

```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }
```

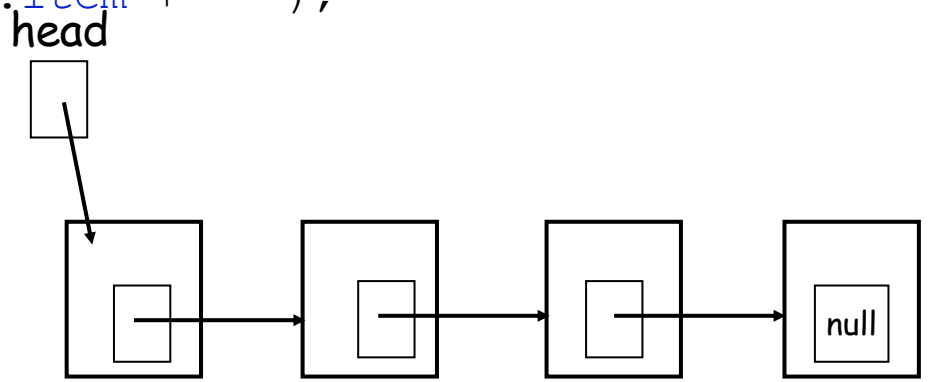
we can rewrite it:

```
  public void addFirst(AnyType item) {
     head = new SListNode<AnyType>(item, head);
     size++;
  }
}
```



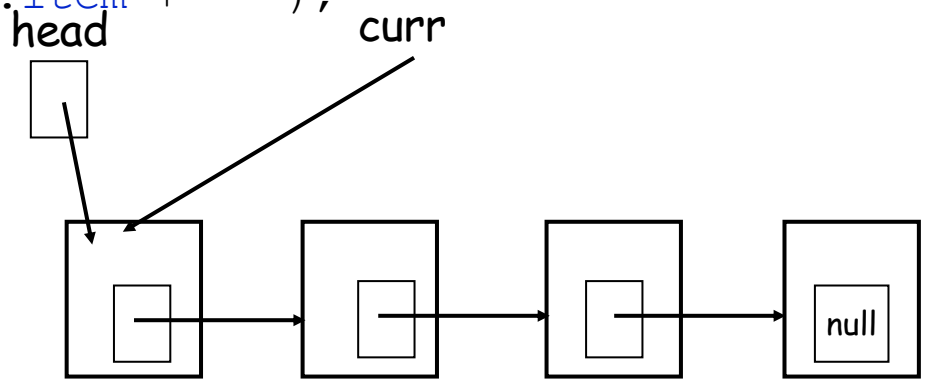
Linked List

```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }

  public void display() {
    SListNode<AnyType> curr = head;
    while (curr != null) {
      System.out.println(curr.item + " ");
      curr = curr.next;
    }
  }
```



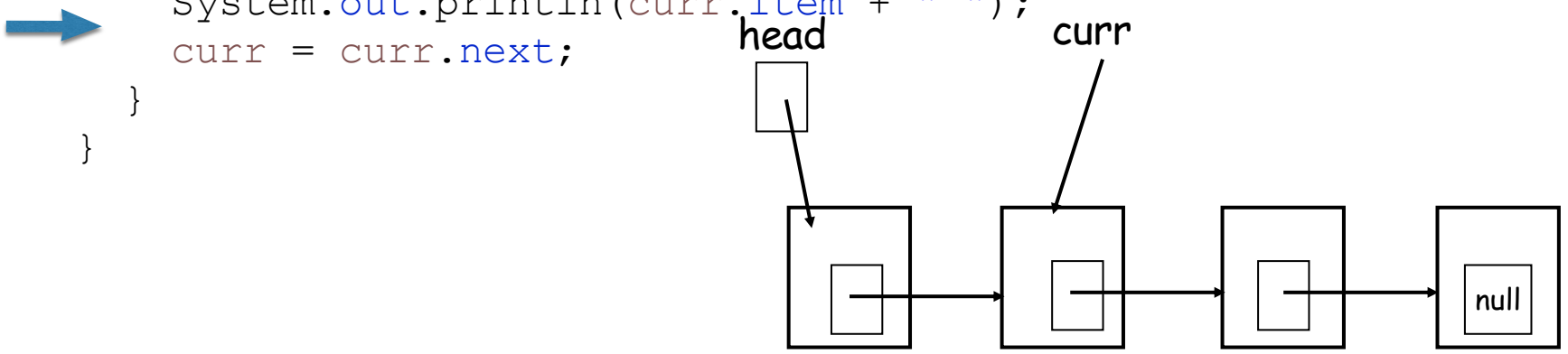
Linked List

```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }

  public void display() {
    SListNode<AnyType> curr = head;
    while (curr != null) {
      System.out.println(curr.item + " ");
      curr = curr.next;
    }
  }
```



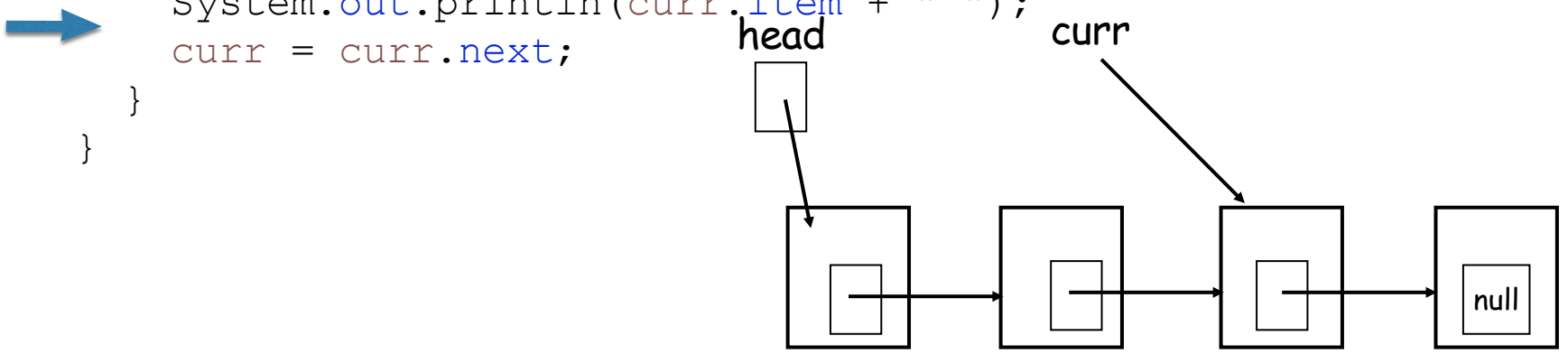
Linked List

```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }

  public void display() {
    SListNode<AnyType> curr = head;
    while (curr != null) {
      System.out.println(curr.item + " ");
      curr = curr.next;
    }
  }
```



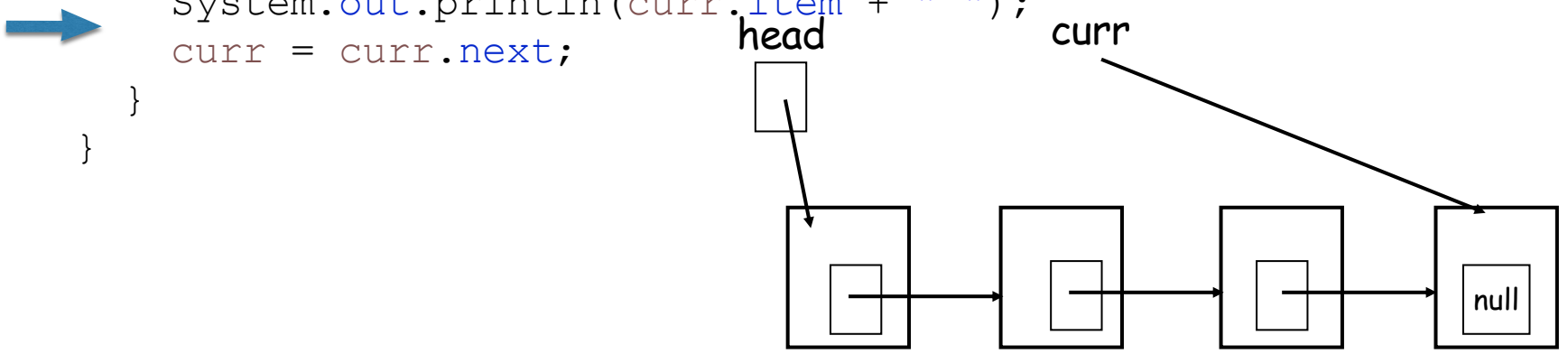
Linked List

```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }

  public void display() {
    SListNode<AnyType> curr = head;
    while (curr != null) {
      System.out.println(curr.item + " ");
      curr = curr.next;
    }
  }
```



Linked List

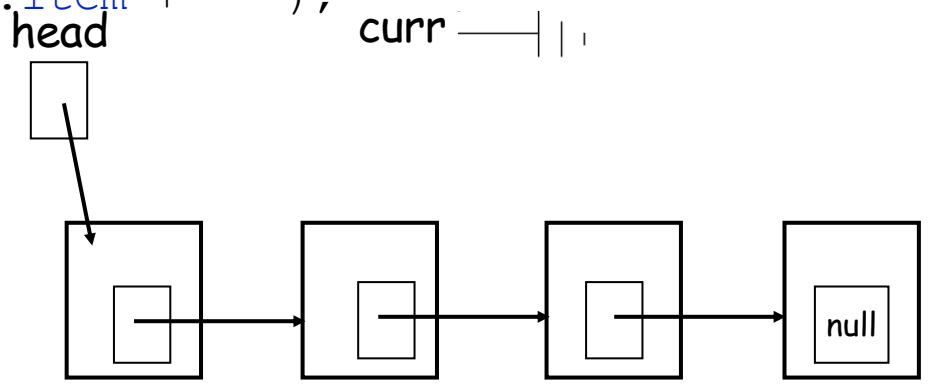
```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }

  public void display() {
    SListNode<AnyType> curr = head;
    while (curr != null) {
      System.out.println(curr.item + " ");
      curr = curr.next;
    }
  }
```



Linked List

```
public class SList<AnyType> {
  private SListNode<AnyType> head;     // First node in list.
  private int size;                    // Number of items in list.

  public SList() {      // Here's how to represent an empty list.
    head = null;
    size = 0;
  }

  public void display() {
    SListNode<AnyType> curr = head;
    while (curr != null) {
      System.out.println(curr.item + " ");
      curr = curr.next;
    }
  }
```

curr is null and we stop.



Insert/Delete at Front

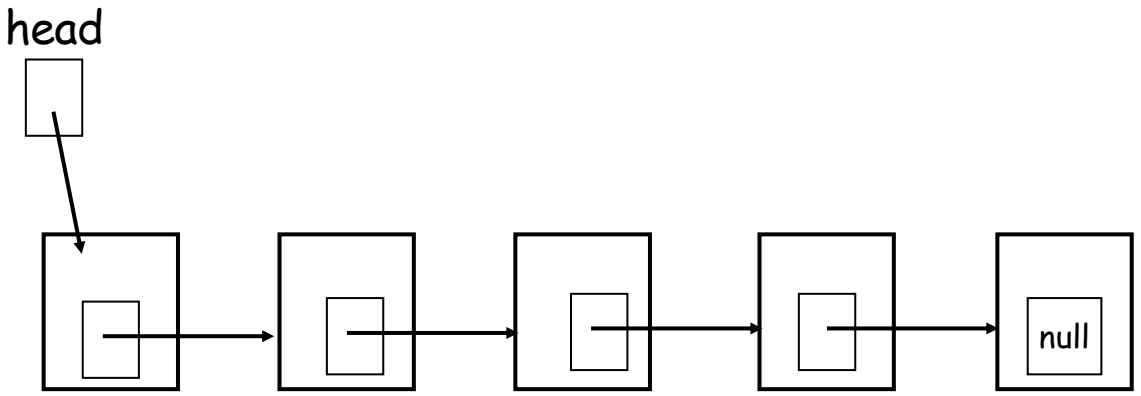
- Inserting/Deleting an item at the front of a linked list is easy.

```
public void removeFirst() {
    if (head != null) {
        head = head.next;
        size--;
    }
}
```

Insert/Delete at Front

- Inserting/Deleting an item at the front of a linked list is easy.

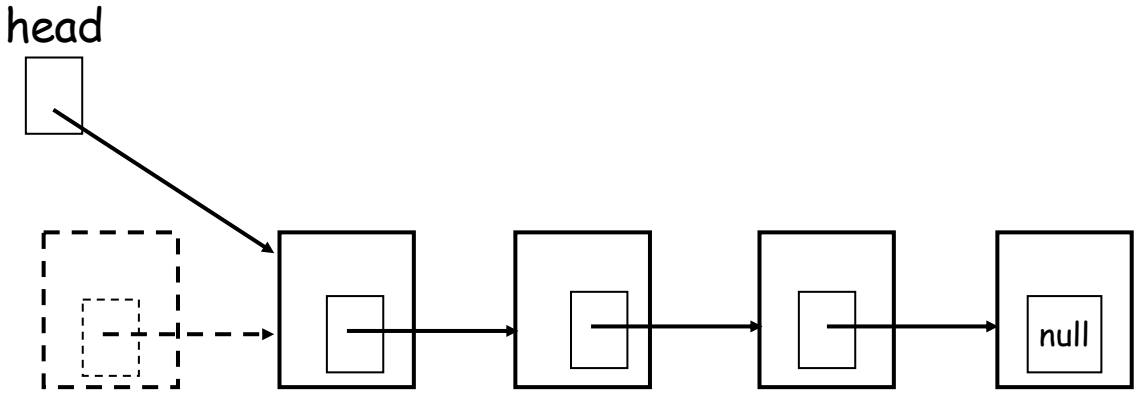
```
public void removeFirst() {
    if (head != null) {
        head = head.next;
        size--;
    }
}
```



Insert/Delete at Front

- Inserting/Deleting an item at the front of a linked list is easy.

```
public void removeFirst() {
    if (head != null) {
        head = head.next;
        size--;
    }
}
```



Insert/Delete at Front

- Inserting/Deleting an item at the front of a linked list is easy.

```
public void removeFirst() {
    if (head != null) {
        head = head.next;
        size--;
    }
}
```

what if the list is empty?

Insert/Delete at Front

- Inserting/Deleting an item at the front of a linked list is easy.

```
public void removeFirst() {
    if (head != null) {
        head = head.next;
        size--;
    }
}
```

what if the list is empty?
our implementation still works!!

Insert/Delete at End

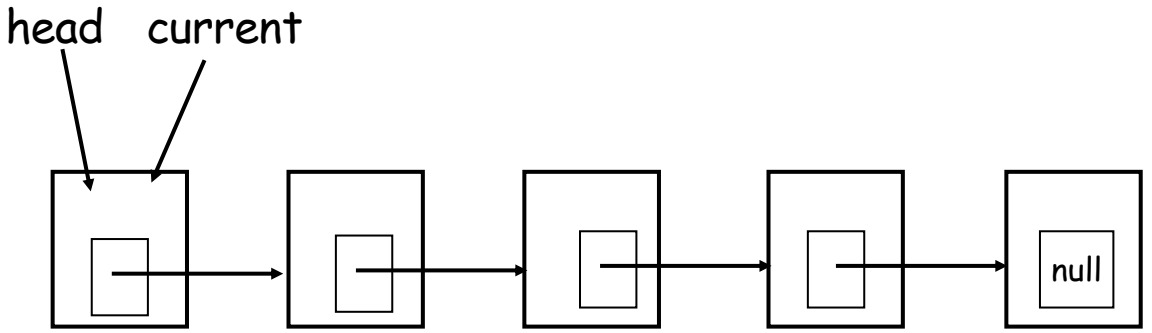
- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

initialize current with head:

SListNode current = head;

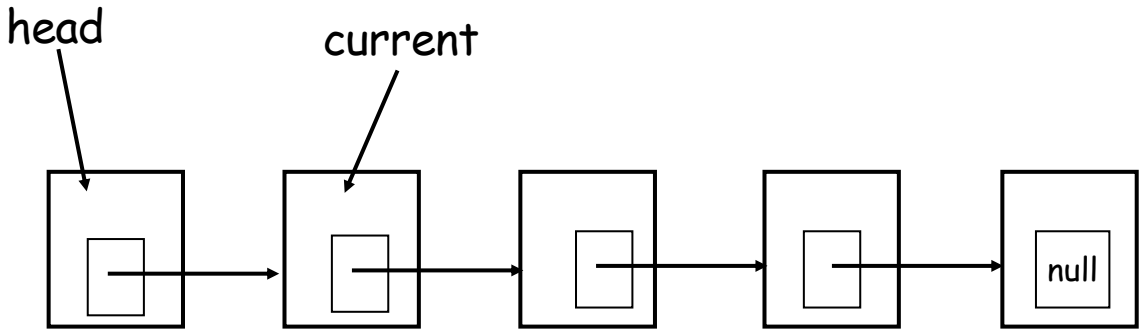


Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

update current to point to the next node in the list

current = current.next;

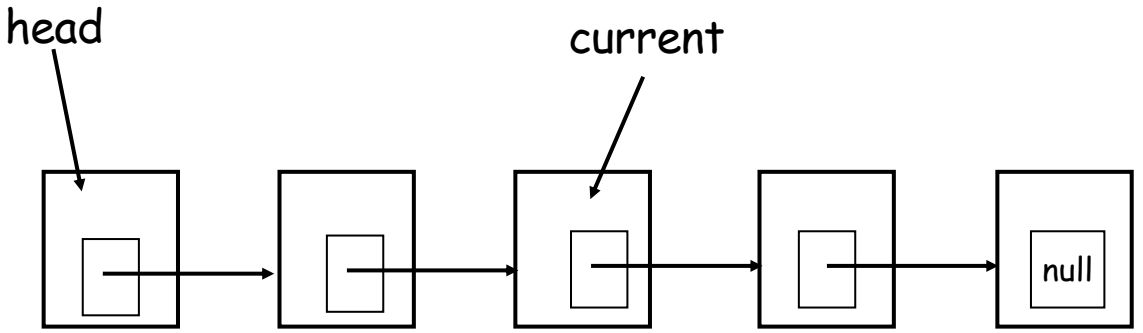


Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

update current to point to the next node in the list

current = current.next;

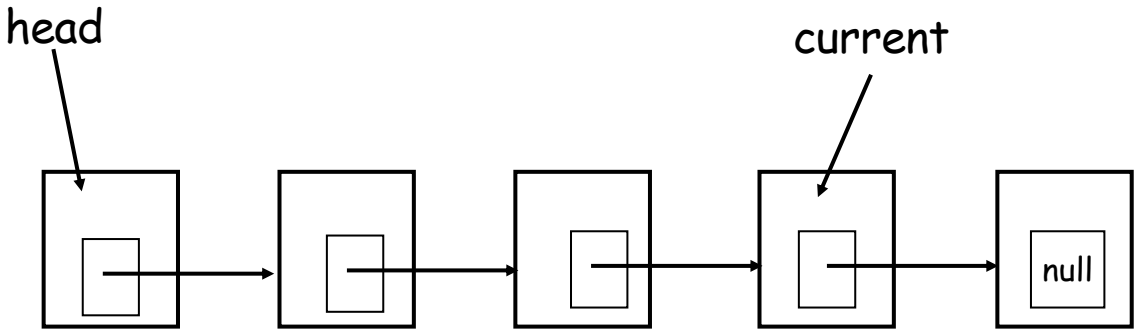


Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

update current to point to the next node in the list

current = current.next;



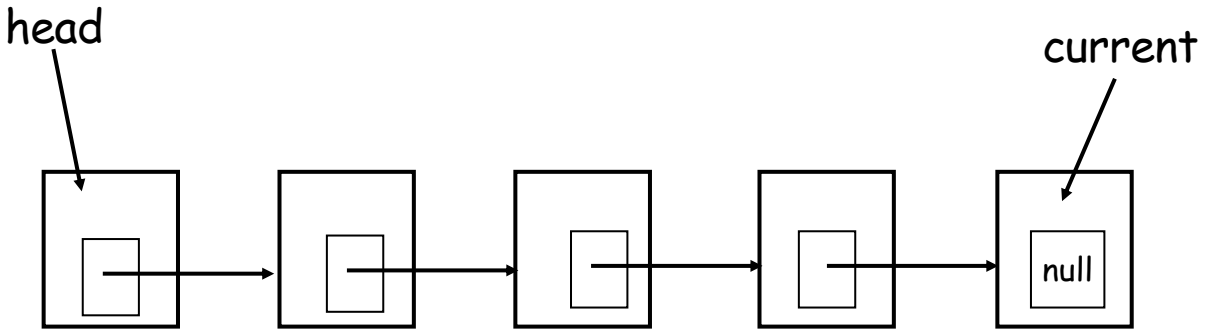
Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

continue until there is no next node

while (current.next != null)

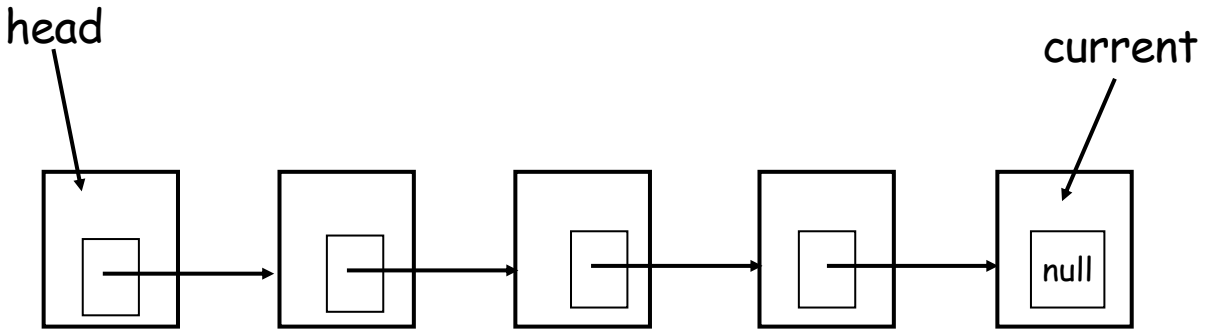
current points to the last node



Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

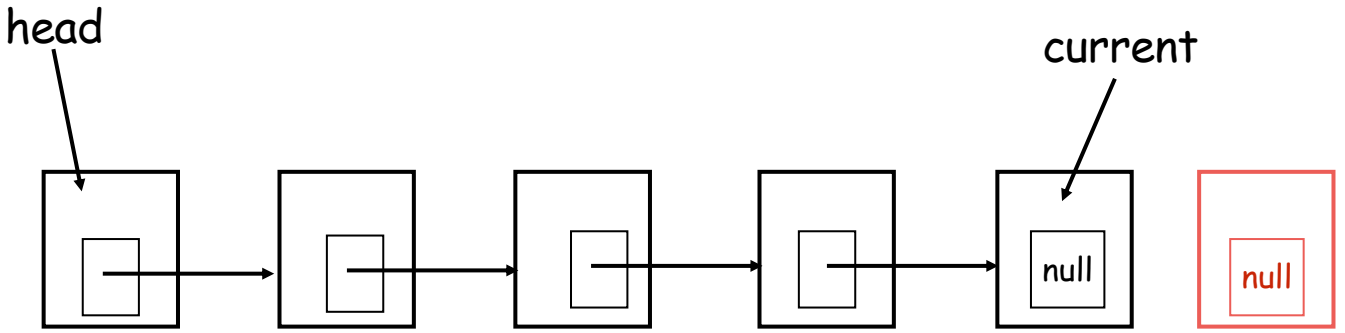
```
public void addLast(AnyType obj) {
    SListNode<AnyType> current = head;
    while (current.next != null) {
        current = current.next;
    }
    current.next = new SListNode<AnyType>(obj);
    size++;
}
```



Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

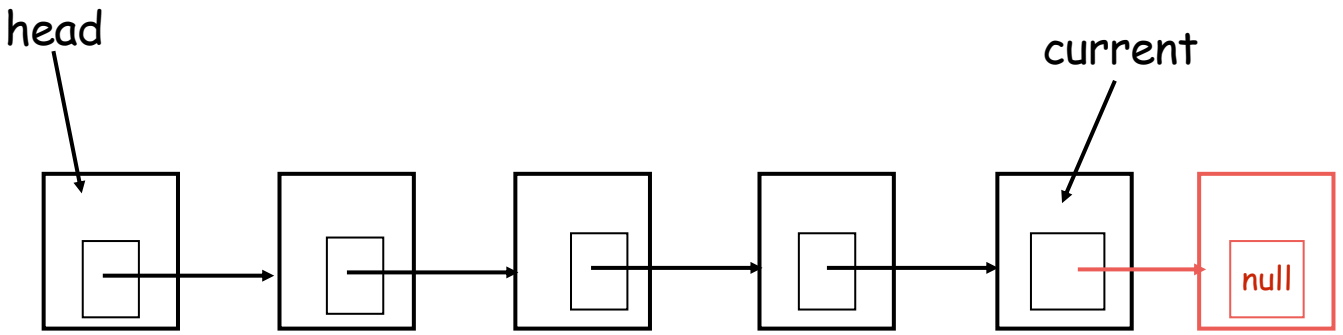
```
public void addLast(AnyType obj) {
    SListNode<AnyType> current = head;
    while (current.next != null) {
        current = current.next;
    }
    current.next = new SListNode<AnyType>(obj);
    size++;
}
```



Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

```
public void addLast(AnyType obj) {
    SListNode<AnyType> current = head;
    while (current.next != null) {
        current = current.next;
    }
    current.next = new SListNode<AnyType>(obj);
    size++;
}
```



Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

```
public void addLast(AnyType obj) {
    SListNode<AnyType> current = head;
    while (current.next != null) {
        current = current.next;
    }
    current.next = new SListNode<AnyType>(obj);
    size++;
}
```

what if the list is empty?

Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

```
public void addLast(AnyType obj) {
    SListNode<AnyType> current = head;
    while (current.next != null) {
        current = current.next;
    }
    current.next = new SListNode<AnyType>(obj);
    size++;
}
```

what if the list is empty? our code will crash!!

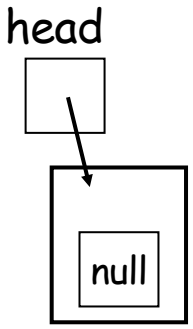
We have to consider this special case.

Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

```
public void addLast(AnyType obj) {
   if (head == null)
      head = new SListNode<AnyType>(obj);
   else {
      SListNode<AnyType> current = head;
      while (current.next != null) {
         current = current.next;
      }
      current.next = new SListNode<AnyType>(obj);
   }
   size++;
}
```

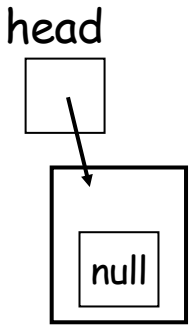
what if the list is empty?
it works now!



Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time

```
public void addLast(AnyType obj) {
   if (head == null)
      head = new SListNode<AnyType>(obj);
   else {
      SListNode<AnyType> current = head;
      while (current.next != null) {
         current = current.next;
      }
      current.next = new SListNode<AnyType>(obj);
   }
   size++;
}
```

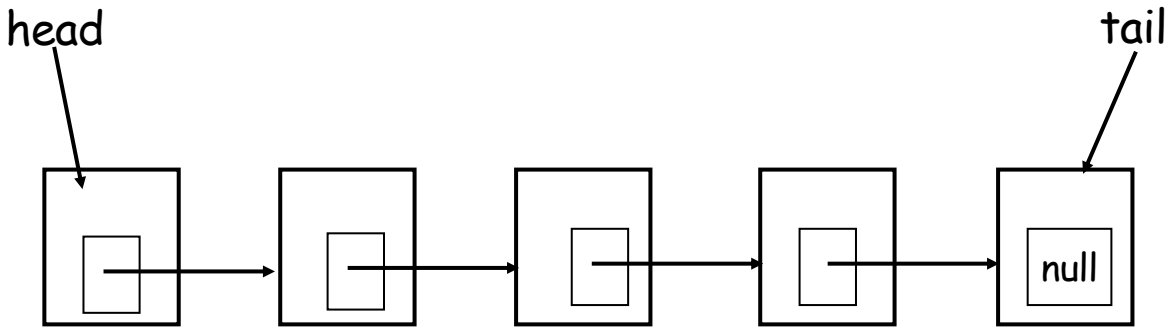


time complexity?

Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time
  - If you have a *tail*,
    - Insert will be easy

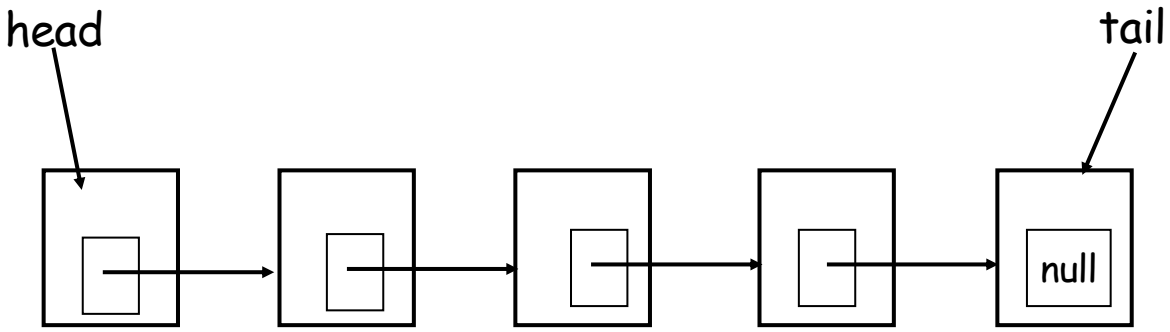
```
public class SList<AnyType> {
  private SListNode<AnyType> head;
  private SListNode<AnyType> tail;
  private int size;
  ...
```



Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time
  - If you have a *tail*,
    - Insert will be easy

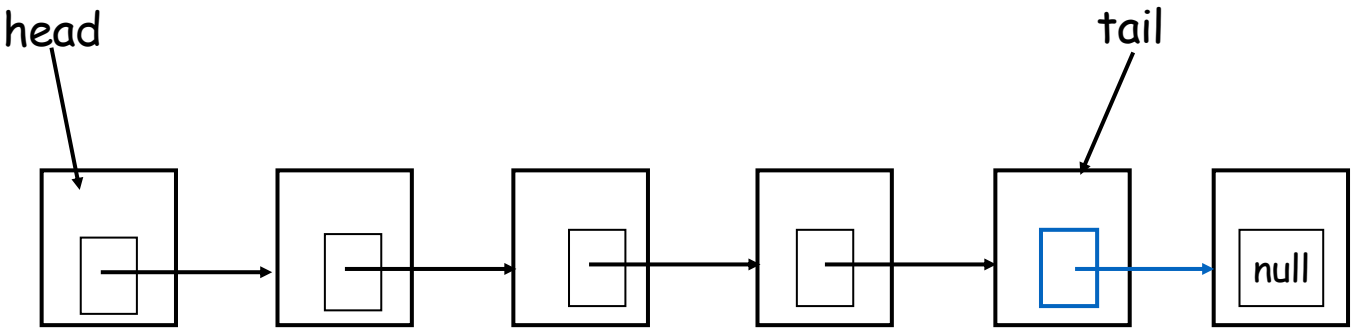
```
public void addLast(AnyType item) {
    tail.next = new SListNode<AnyType>(item);
    tail = tail.next;
    size++;
}
```



Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time
  - If you have a *tail*,
    - Insert will be easy

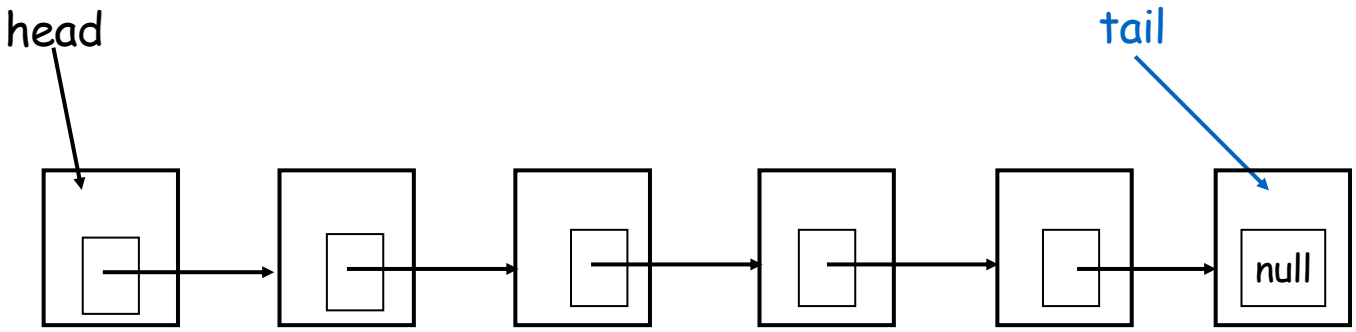
```
public void addLast(AnyType item) {
    tail.next = new SListNode<AnyType>(item);
    tail = tail.next;
    size++;
}
```



Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time
  - If you have a *tail*,
    - Insert will be easy

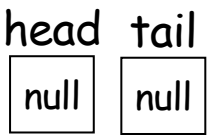
```
public void addLast(AnyType item) {
    tail.next = new SListNode<AnyType>(item);
    tail = tail.next;
    size++;
}
```



Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time
  - If you have a *tail*,
    - Insert will be easy

```
public void addLast(AnyType item) {
    tail.next = new SListNode<AnyType>(item);
    tail = tail.next;
    size++;
}
```

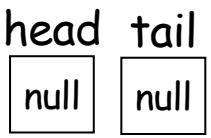


what if the list is empty?

Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time
  - If you have a *tail*,
    - Insert will be easy

```
public void addLast(AnyType item) {
    tail.next = new SListNode<AnyType>(item);
    tail = tail.next;
    size++;
}
```



what if the list is empty?

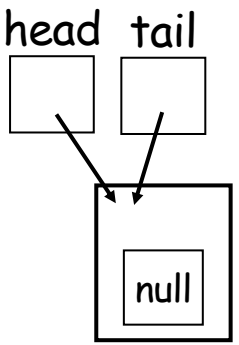
our code will crash!!

head and tail should be updated

Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time
  - If you have a *tail*,
    - Insert will be easy

what if the list is empty?

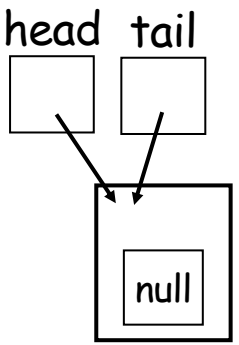



```
public void addLast(AnyType item) {
    if (tail == null){
        tail = new SListNode<AnyType>(item);
        head = tail;
    }
    // Second case: list is not empty
    else{
        tail.next = new
SListNode<AnyType>(item);
        tail = tail.next;
    }
    size++;
}
```

Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time
  - If you have a *tail*,
    - Insert will be easy

time complexity?

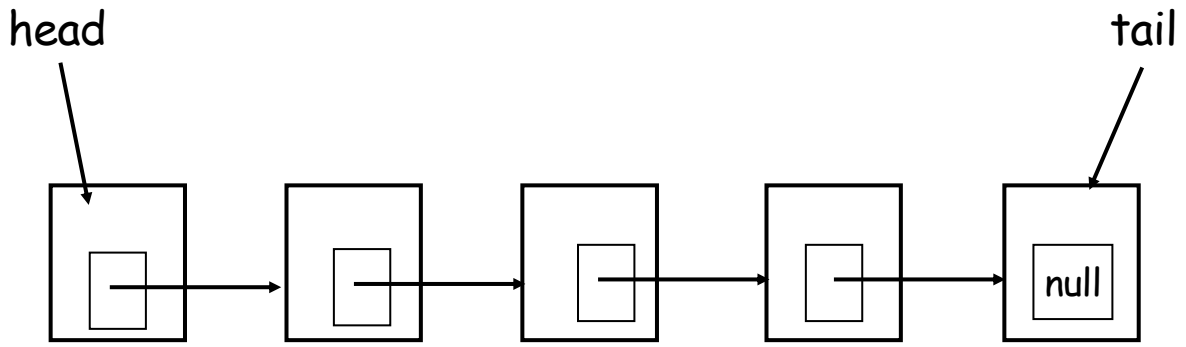



```
public void addLast(AnyType item) {
    if (tail == null){
        tail = new SListNode<AnyType>(item);
        head = tail;
    }
    // Second case: list is not empty
    else{
        tail.next = new
SListNode<AnyType>(item);
        tail = tail.next;
    }
    size++;
}
```

Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time
  - If you have a *tail*,
    - Insert will be easy

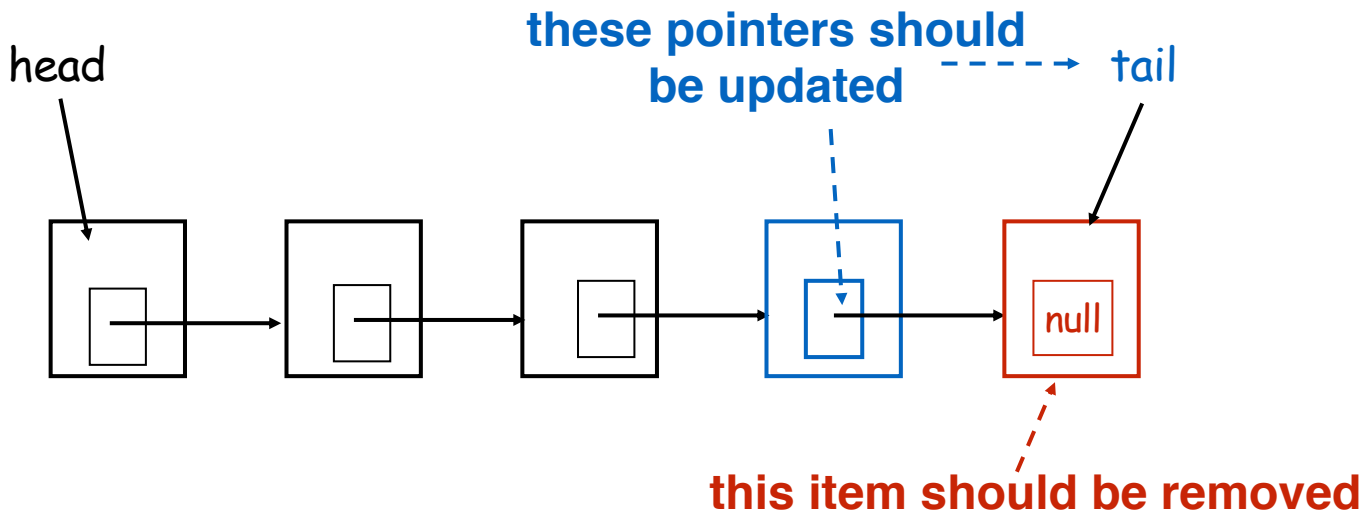
How about delete?



Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time
  - If you have a *tail*,
    - Insert will be easy

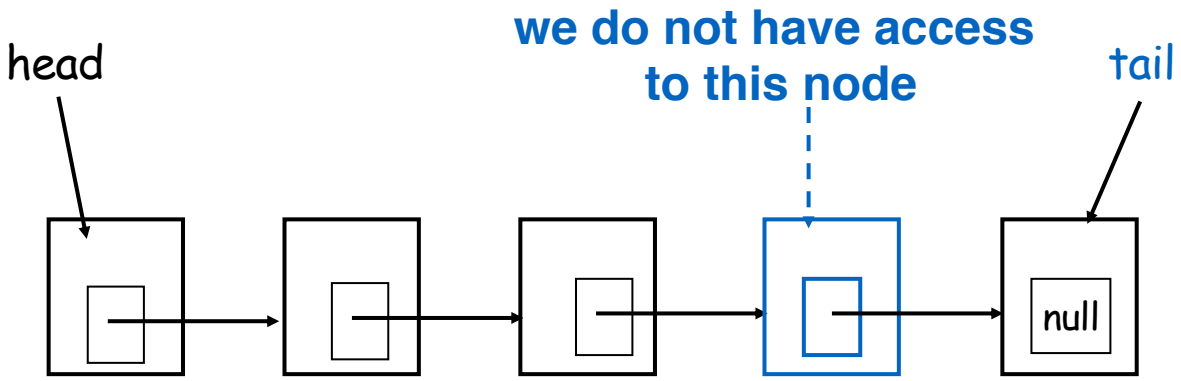
How about delete?



Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time
  - If you have a *tail*,
    - Insert will be easy

How about delete?

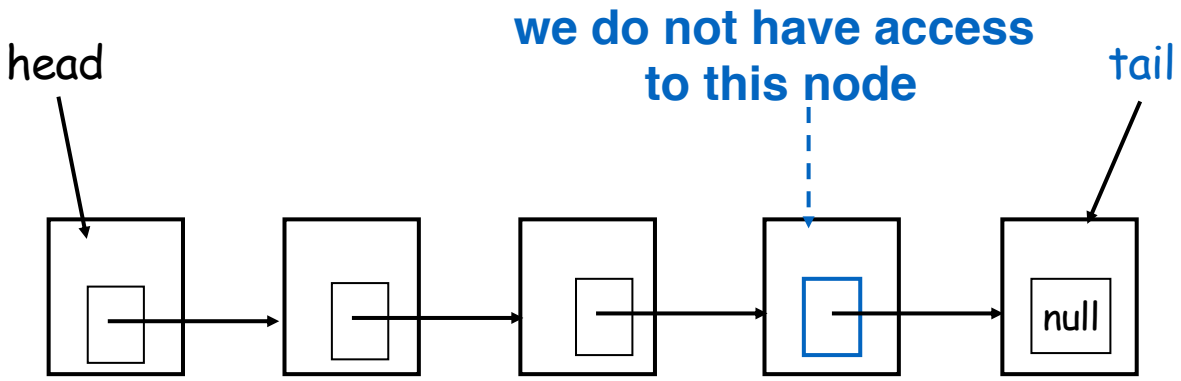


To find that pointer, we need to search through the whole list!

Insert/Delete at End

- Inserting/Deleting an item at the end of a linked list
  - Entails a search through the entire list, which might take a long time
  - If you have a *tail*,
    - Insert will be easy
    - Still delete will need to traverse the list

How about delete?



To find that pointer, we need to search through the whole list!

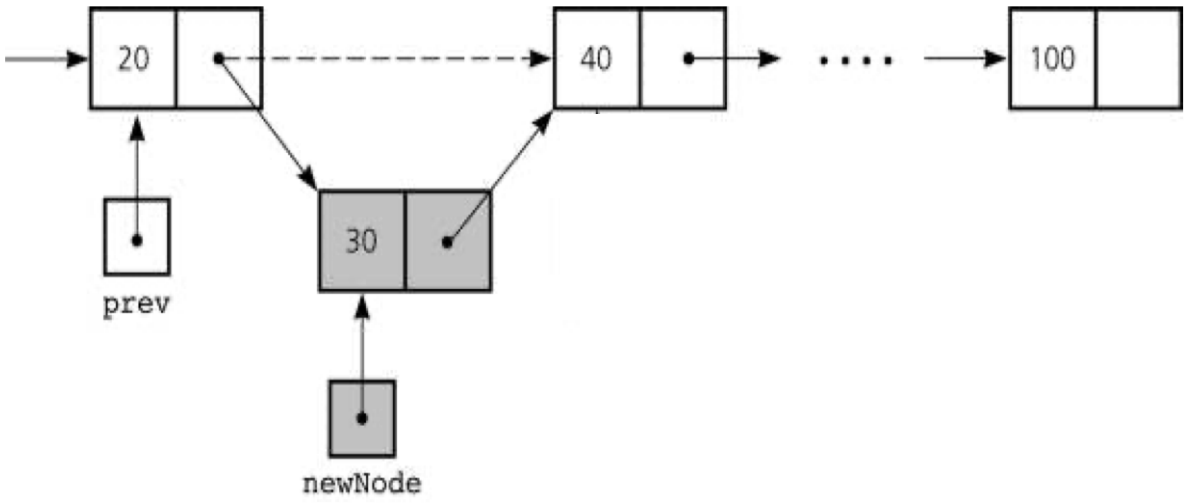
Insertion in arbitrary position

- Insert a new item after a node
  1. create a new node

```
SListNode<AnyType> newNode = new SListNode<AnyType>(item);
```

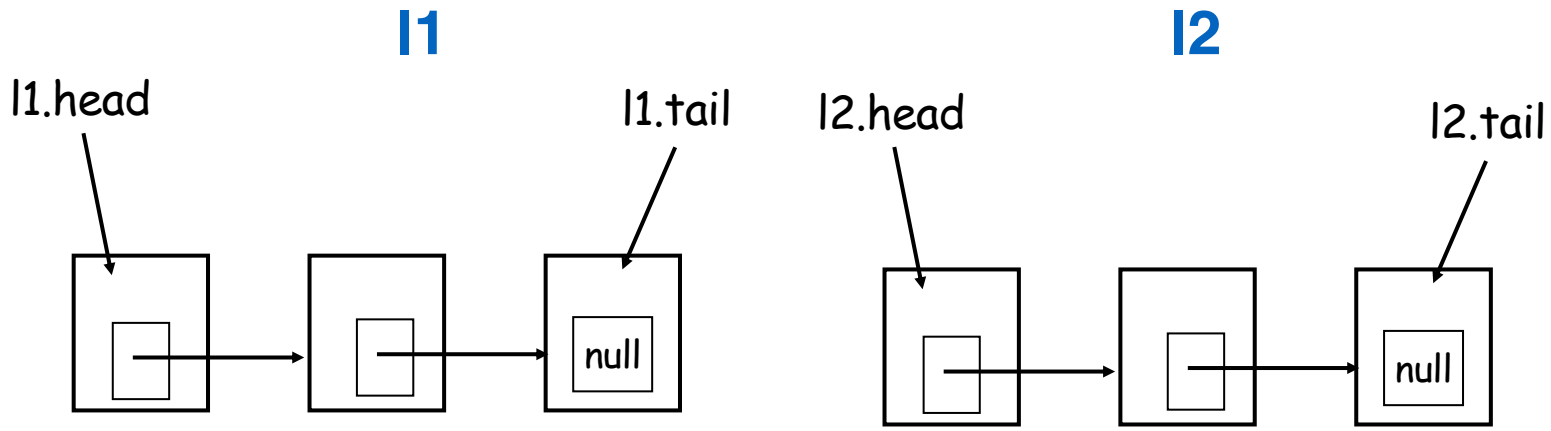
  2.Insert it after `prev`

```
newNode.next = prev.next;
prev.next = newNode;
```



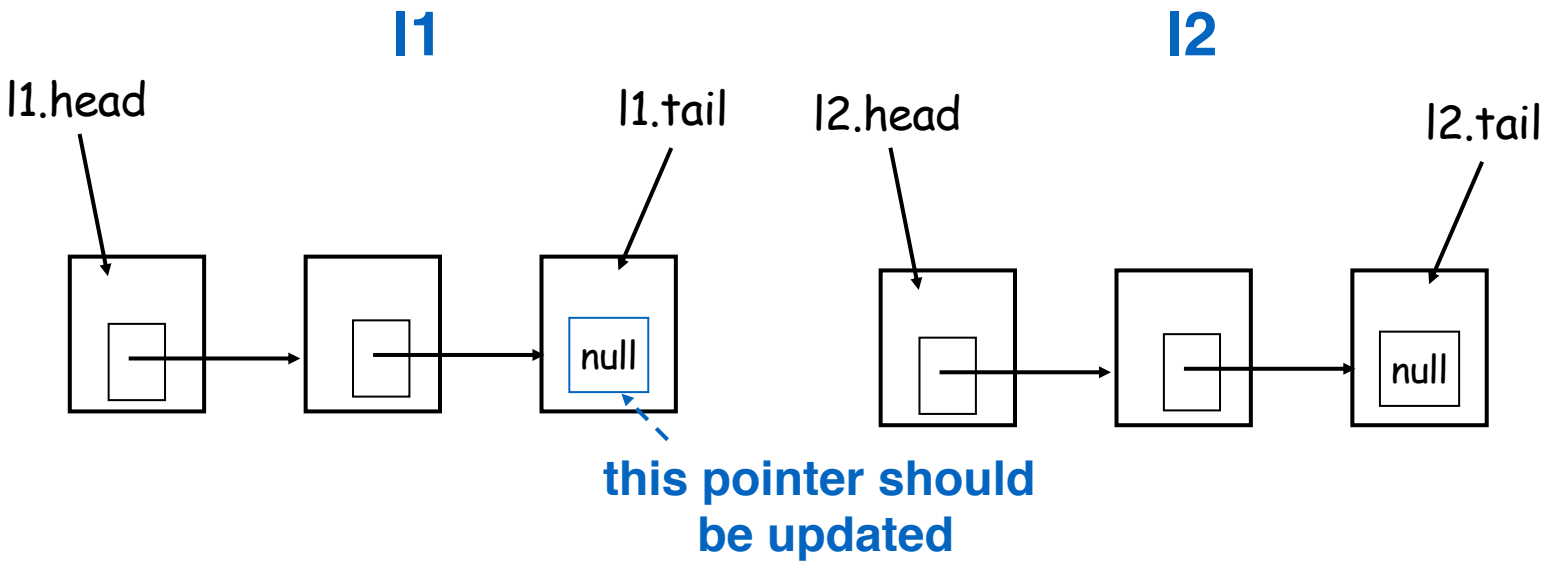
Concatenate two Lists

- Concatenating one list to the end of another list
  - Assume you have a *tail*,



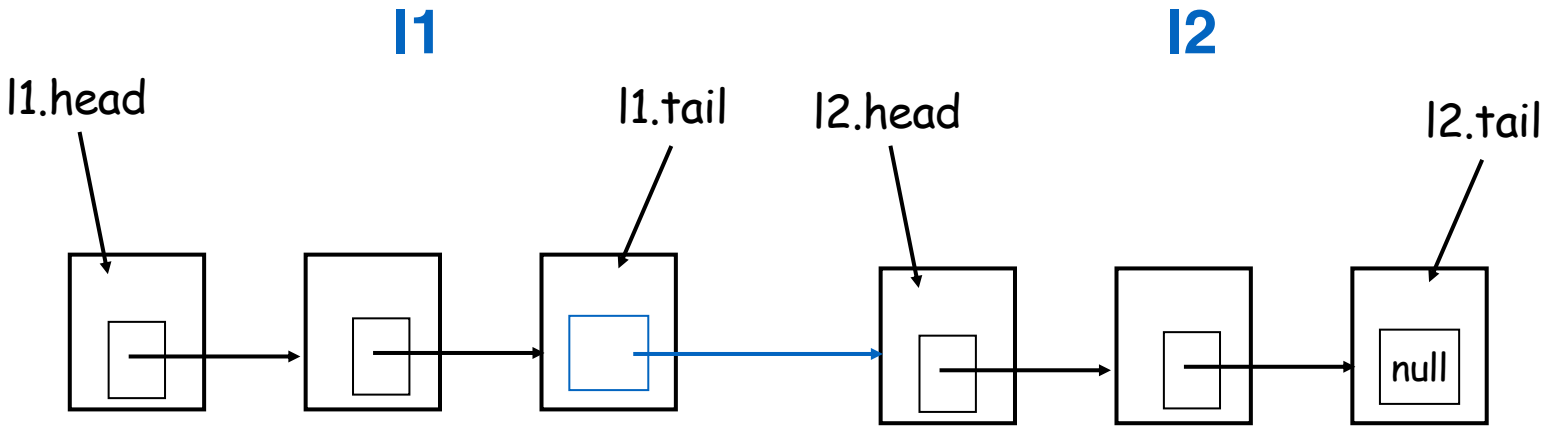
Concatenate two Lists

- Concatenating one list to the end of another list
  - Assume you have a *tail*,



Concatenate two Lists

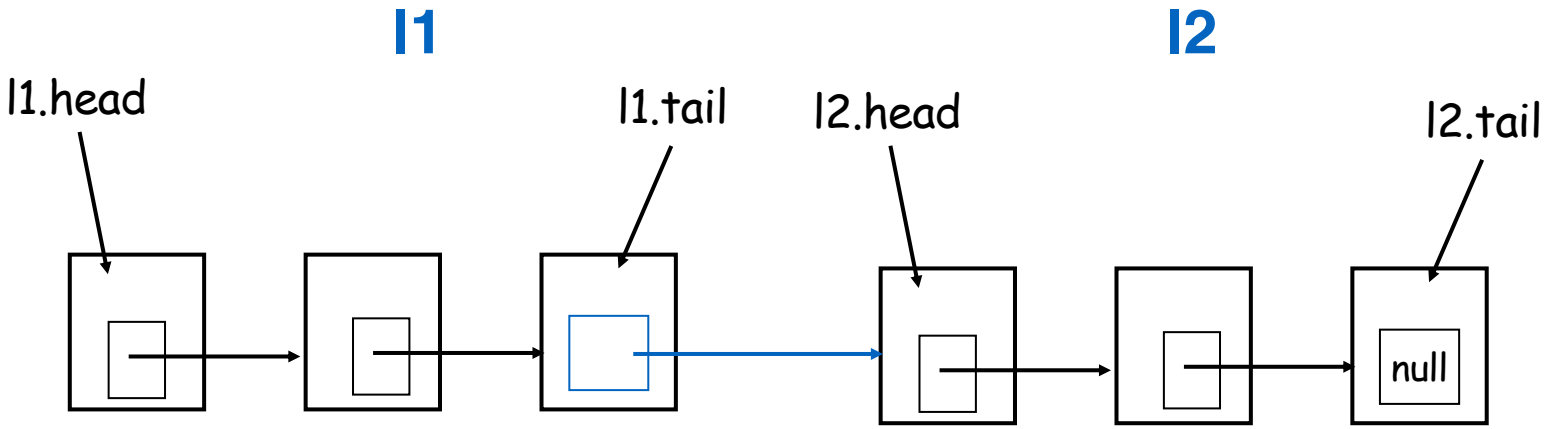
- Concatenating one list to the end of another list
  - Assume you have a *tail*,



Concatenate two Lists

- Concatenating one list to the end of another list
  - Assume you have a *tail*,

What if l1 is empty?
What if l2 is empty?
What if both are empty?



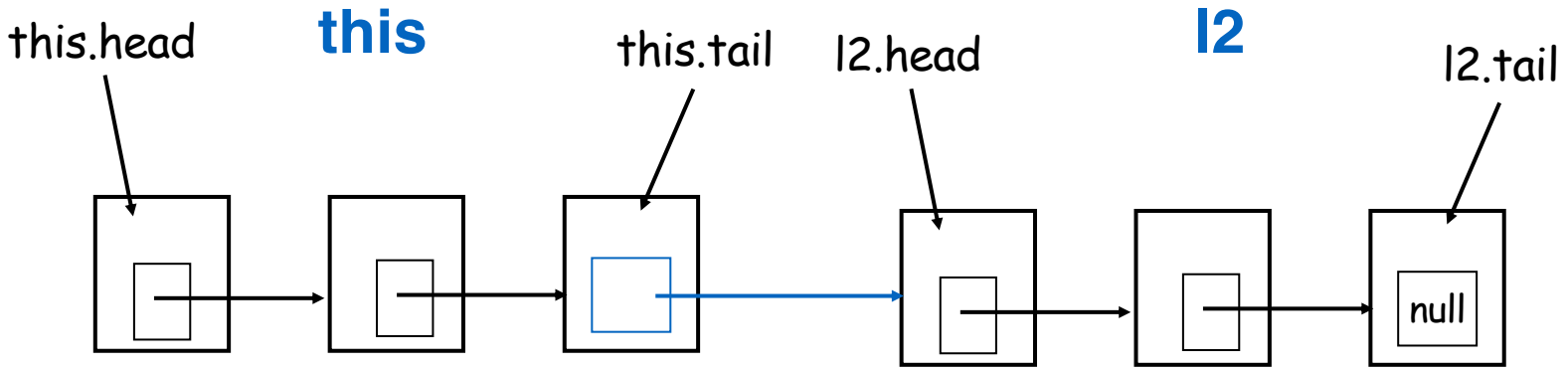
Concatenate two Lists

- Concatenating one list to the end of another list
  - Assume you have a *tail*,
    - We want to add concatenate as a method to the class SList

```
public void concatenate(SList l2);
```

What if l1 is empty?
What if l2 is empty?
What if both are empty?
size should be updated

Practice for you!



Linked Lists

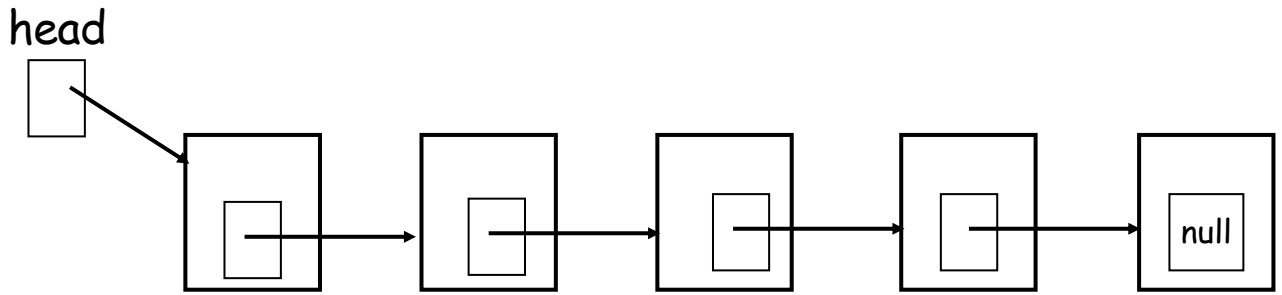
- Advantages
  - Inserting item into the list (linked list)
    - Constant time ***if*** you have reference to the previous node (don't have to walk through the whole list searching for it)
    - List can keep growing until memory runs out.

Arrays vs Linked Lists

- Disadvantages
  - Finding the n-th item of a linked list takes time proportional to the length of the list (n).
    - In arrays you can do it in constant time
- Trade-off between arbitrary lookup (finding n-th item) and arbitrary insertion (inserting a new item)
  - Finding a compromise between arrays and linked lists.
  - Data structures that are fast for both arbitrary lookups (like arrays) and arbitrary insertions (like linked lists)

SList

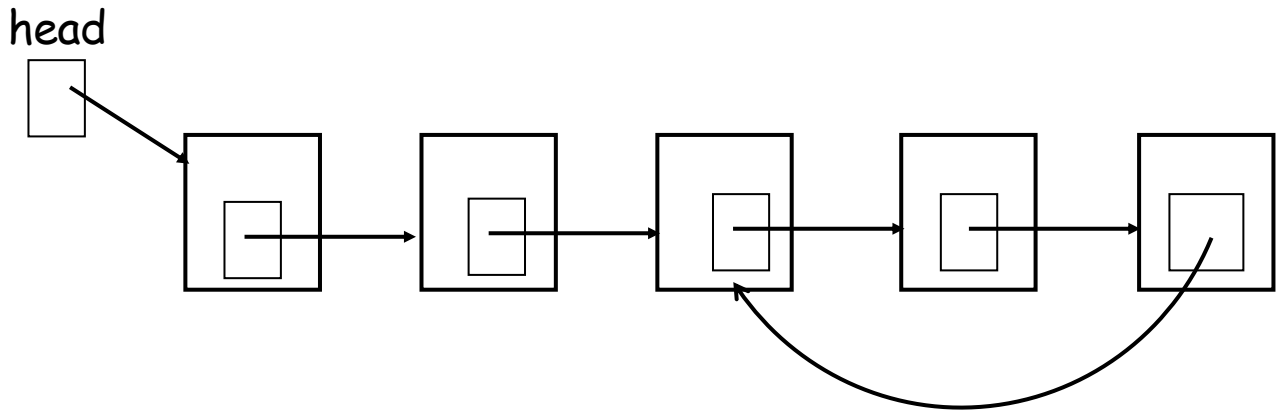
- In implementation of all methods, we should always make sure
  1) `size` is always correct
  2) List is never circularly linked (the last node refers to null)



SList

- In implementation of all methods, we should always make sure
  1) `size` is always correct
  2) List is never circularly linked

What's wrong with circles?

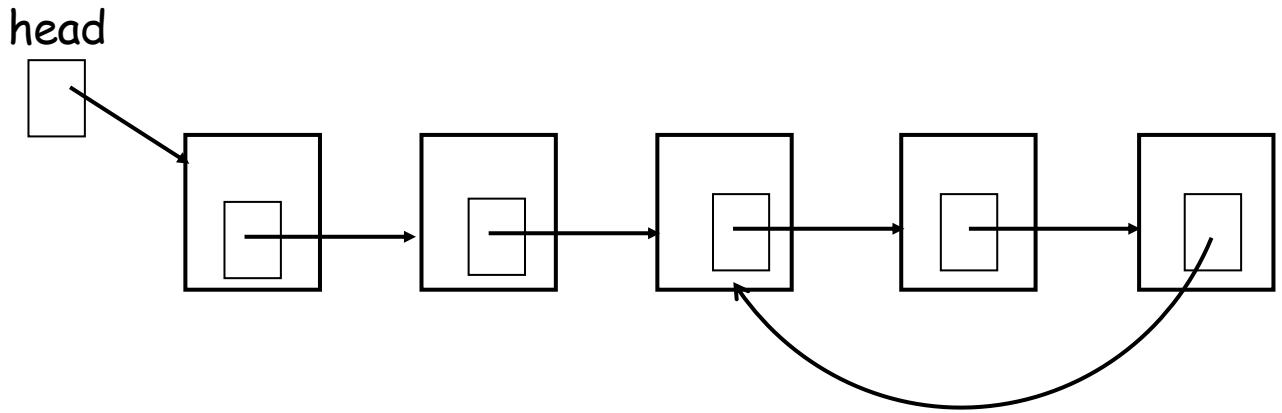


SList

- In implementation of all methods, we should always make sure
  1) `size` is always correct
  2) List is never circularly linked

What's wrong with circles?

Assume we want to print all items. It keeps looping forever!



Doubly Linked Lists

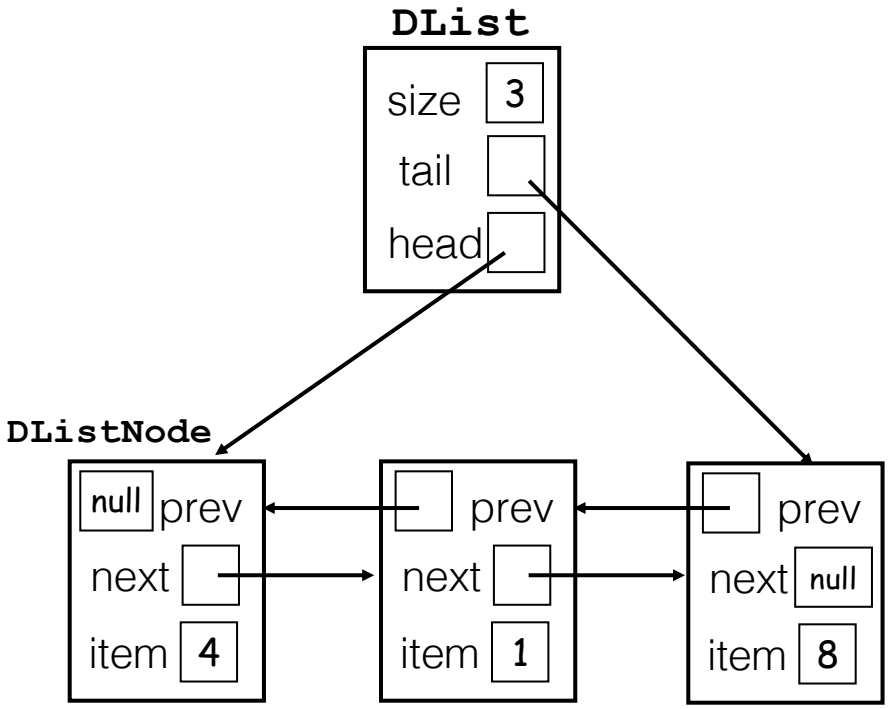
Doubly Linked Lists

- We want to do both insert/delete at both ends in constant time.

```
public class DListNode<AnyType> {
    public AnyType item;
    public DListNode<AnyType> next;
    public DListNode<AnyType> prev;
}


public class DList<AnyType> {
    private int size;
    private DListNode<AnyType> head;
    private DListNode<AnyType> tail;
}
```

Doubly Linked Lists

- We want to do both insert/delete at both ends in constant time.



Doubly Linked Lists

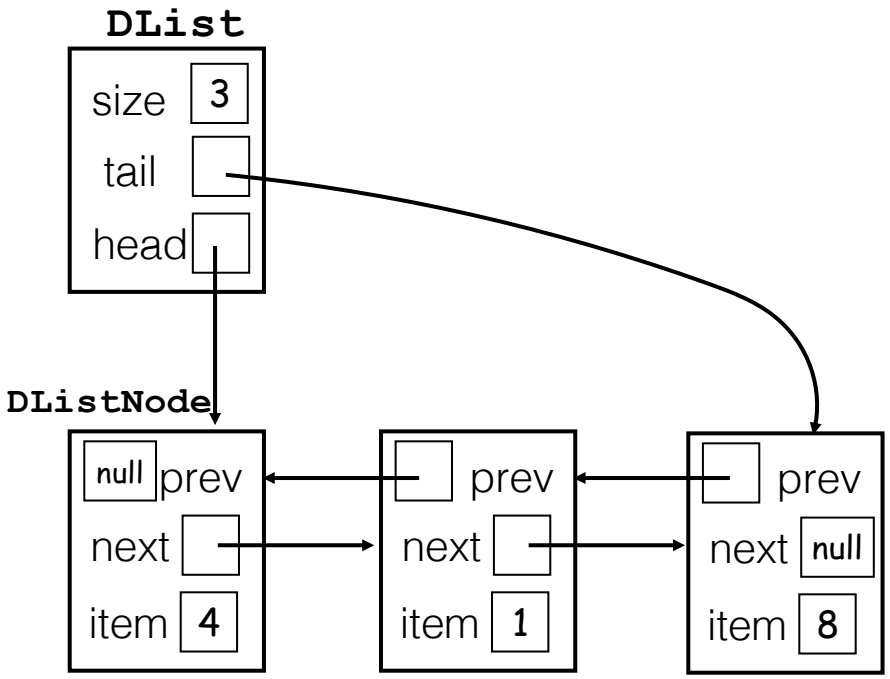
- Remove the tail node (assume there are at least two items in the list)

```
tail.prev.next = null;
tail = tail.prev;
size--;
```

Doubly Linked Lists

- Remove the tail node (assume there are at least two items in the list)

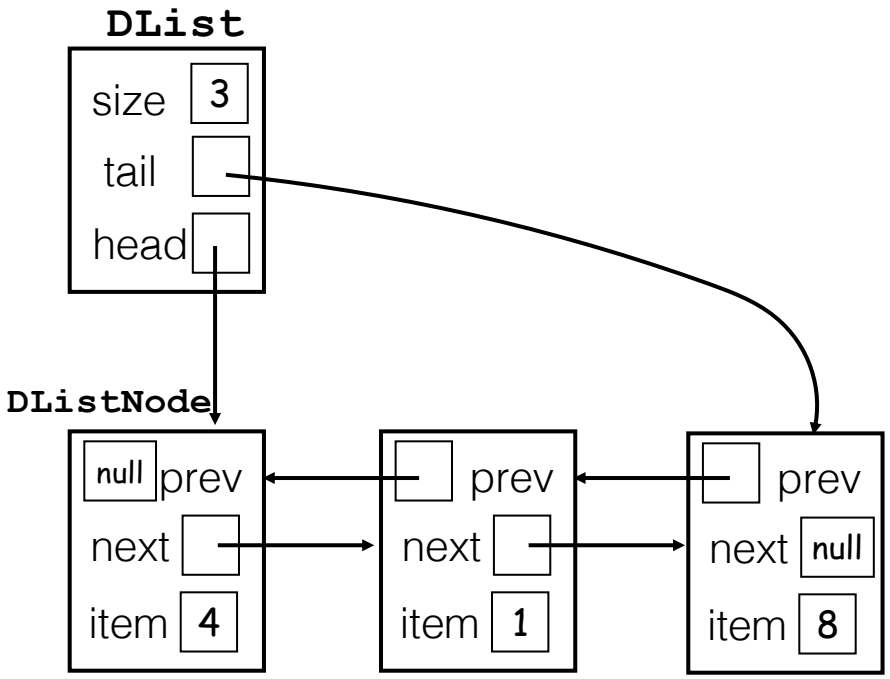
```
tail.prev.next = null;
tail = tail.prev;
size--;
```



Doubly Linked Lists

- Remove the tail node (assume there are at least two items in the list)

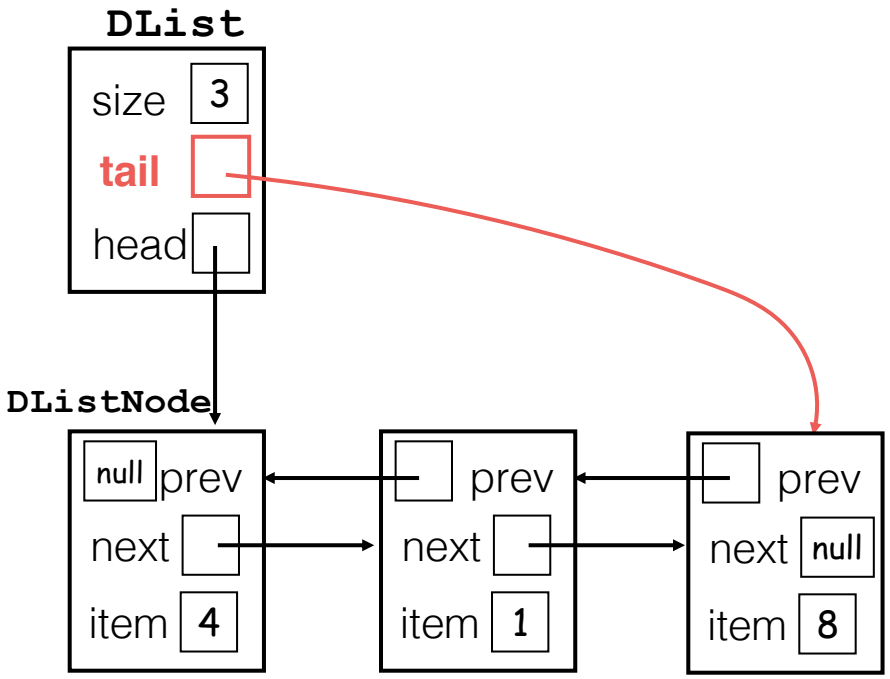
```
tail.prev.next = null;
tail = tail.prev;
size--;
```



Doubly Linked Lists

- Remove the tail node (assume there are at least two items in the list)

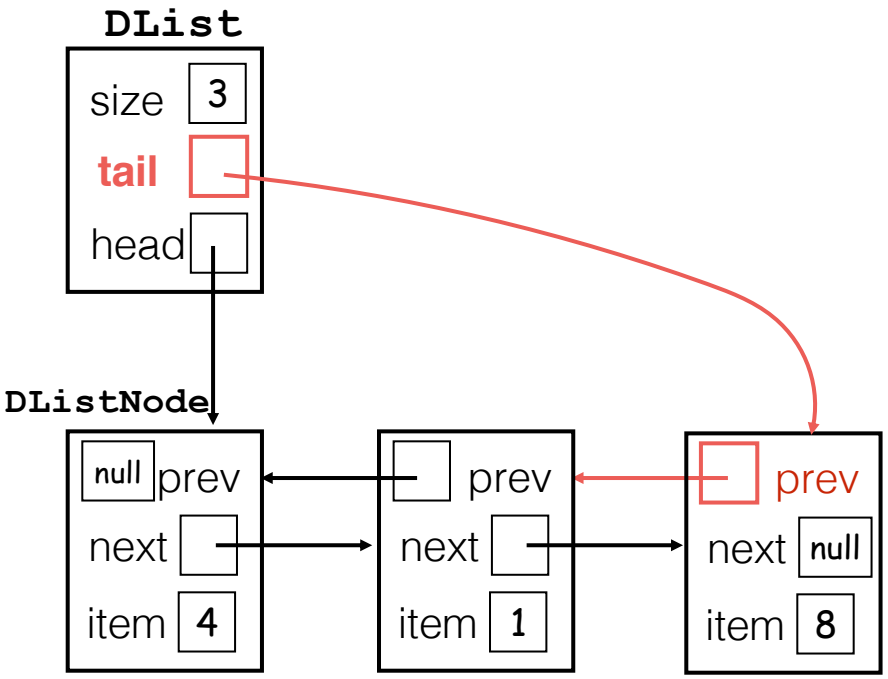
```
tail.prev.next = null;
tail = tail.prev;
size--;
```



Doubly Linked Lists

- Remove the tail node (assume there are at least two items in the list)

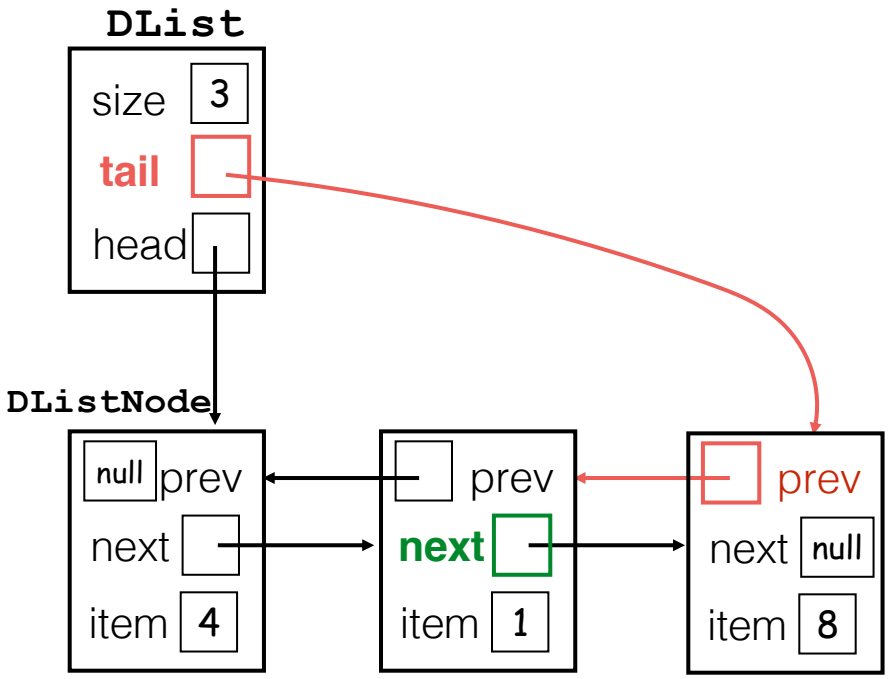
```
tail.prev.next = null;
tail = tail.prev;
size--;
```



Doubly Linked Lists

- Remove the tail node (assume there are at least two items in the list)

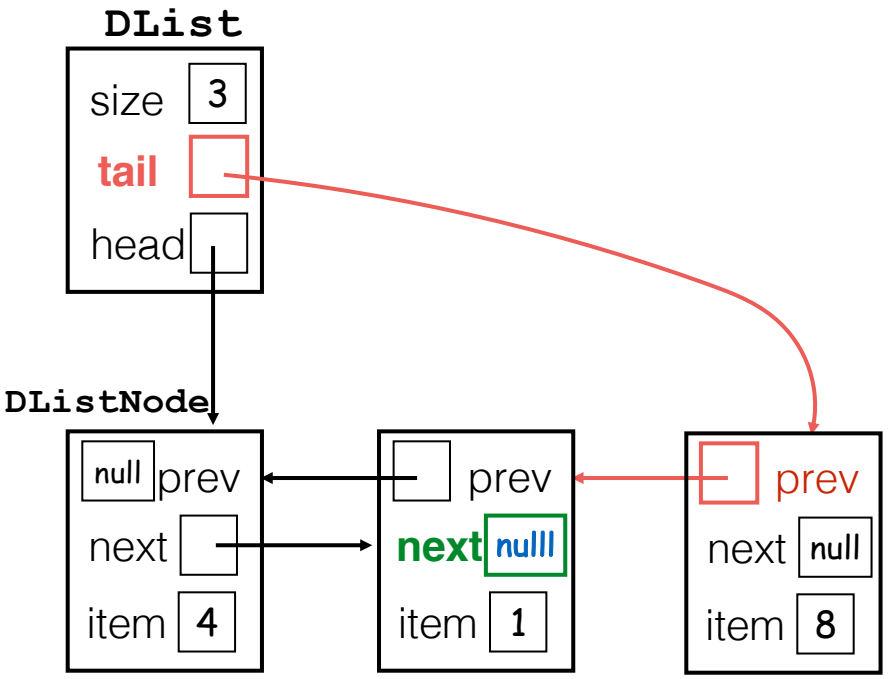
```
tail.prev.next = null;
tail = tail.prev;
size--;
```



Doubly Linked Lists

- Remove the tail node (assume there are at least two items in the list)

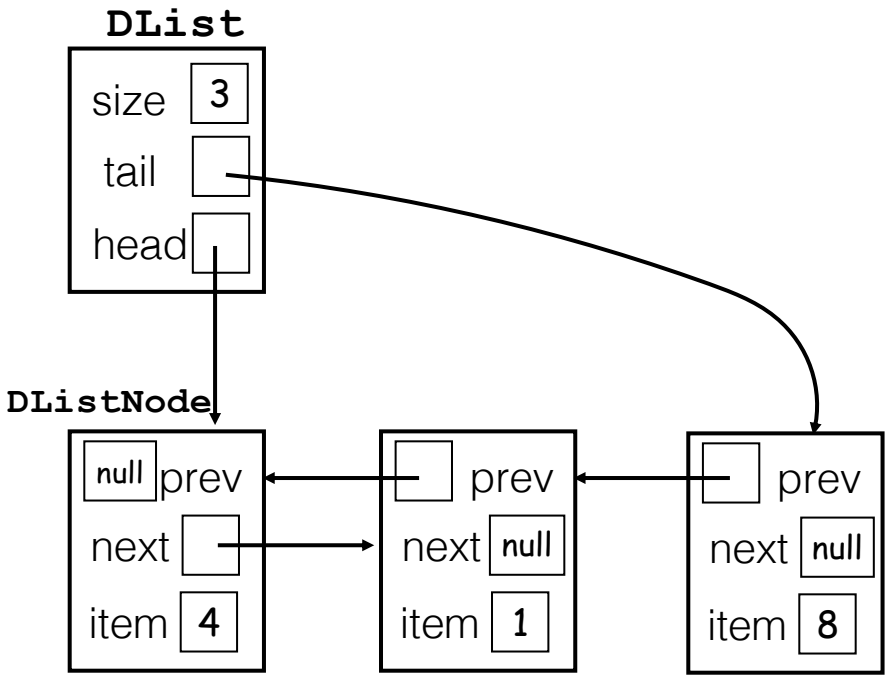
```
tail.prev.next = null;
tail = tail.prev;
size--;
```



Doubly Linked Lists

- Remove the tail node (assume there are at least two items in the list)

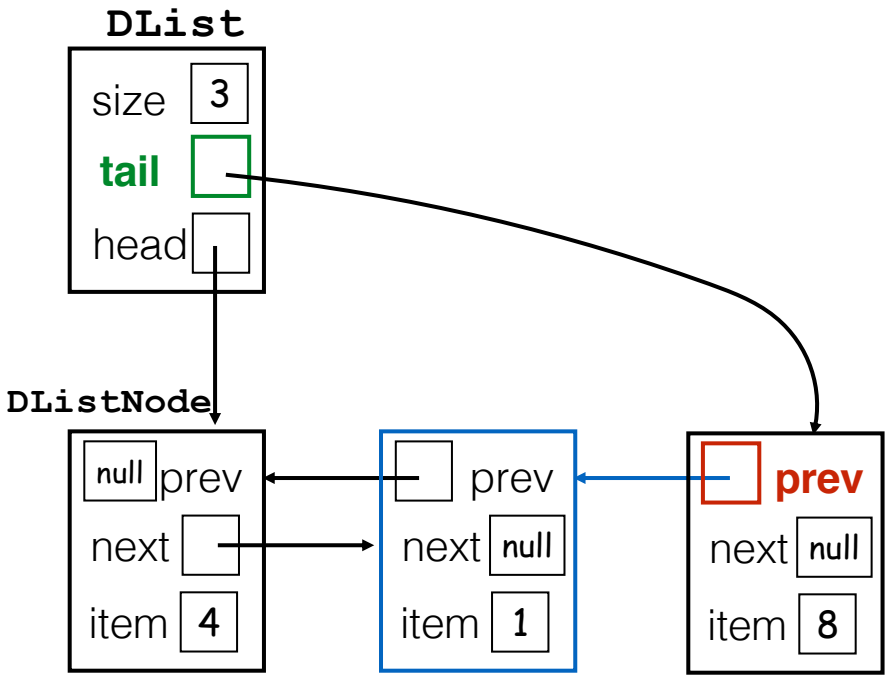
```
tail.prev.next = null;
tail = tail.prev;
size--;
```



Doubly Linked Lists

- Remove the tail node (assume there are at least two items in the list)

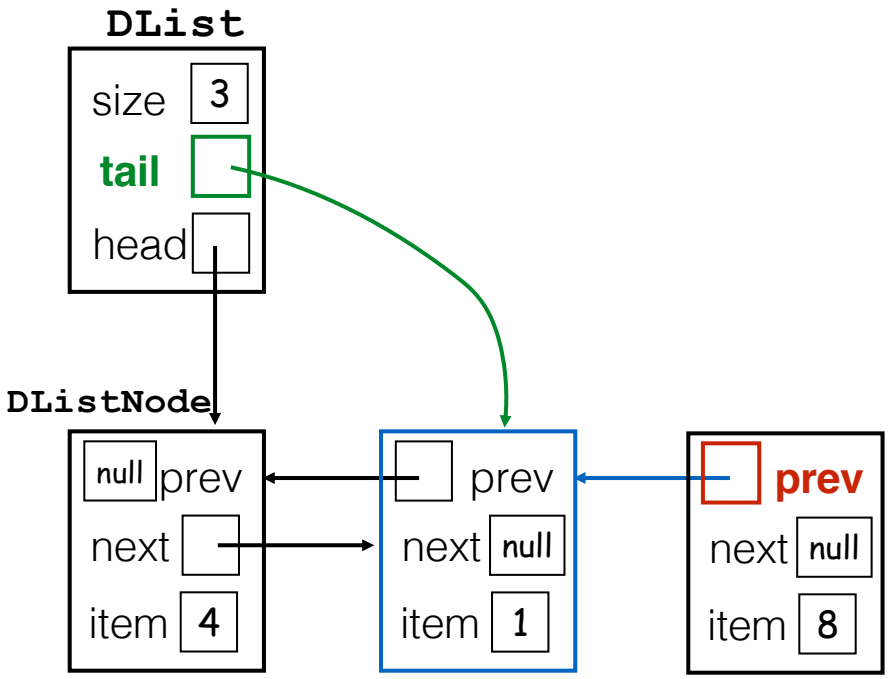
```
tail.prev.next = null;
tail = tail.prev;
size--;
```



Doubly Linked Lists

- Remove the tail node (assume there are at least two items in the list)

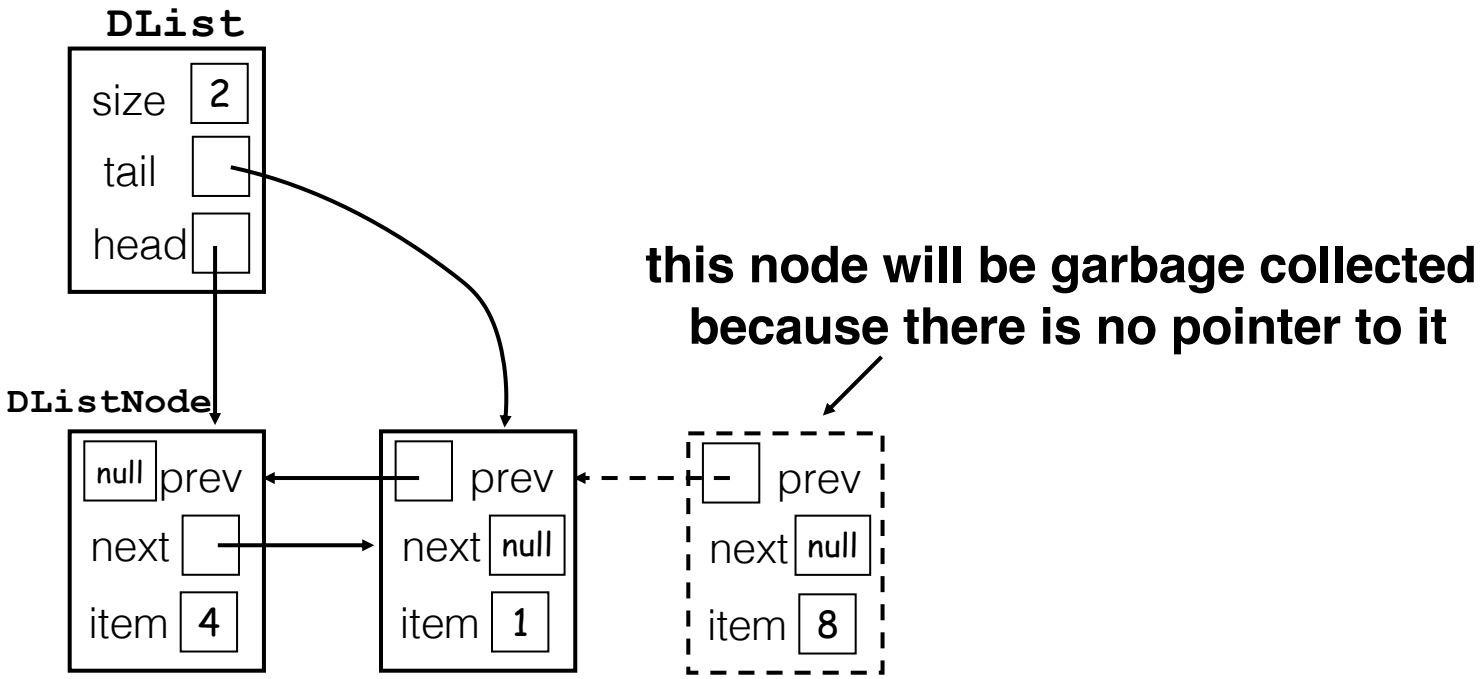
```
tail.prev.next = null;
tail = tail.prev;
size--;
```



Doubly Linked Lists

- Remove the tail node (assume there are at least two items in the list)

```
tail.prev.next = null;
tail = tail.prev;
size--;
```



Special Cases

```
tail.prev.next = null;
tail = tail.prev;
size--;
```

- What if the list has one item? (or zero item?)
- You'll need a special case for a `DList` with no items.
- You'll also need a special case for a `DList` with one item, because `tail.prev.next` does not exist.

LinkedList in Java Collection API

- LinkedList class is a doubly-linked list implementation of ADT List in Java Collection API.
- LinkedList allows sequential access of elements.
- In other words, LinkedList can be searched forward and backward but the time it takes to traverse the list is directly proportional to the size of the list.
- LinkedList is also an implementation of ***Deque***
- It implements all optional list operations (and more), and permits all elements, including null.

LinkedList in Java Collection API

`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`addFirst(`**value**`)`	Inserts value at the beginning of this list.
`addLast(`**value**`)`	appends value at end of list
`clear()`	removes all elements of the list
`get(`**index**`)`	returns the value at given index
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`removeFirst()`	Removes and returns the first element from this list.
`removeFirst()`	Removes and returns the last element from this list.
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

Check the Java API tutorials for the full list of methods:
http://docs.oracle.com/javase/8/docs/api/?java/util/LinkedList.html

LinkedList in Java Collection API

`addAll(`**list**`)` `addAll(`**index**`, `**list**`)`	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
`contains(`**value**`)`	returns true if given value is found somewhere in this list
`containsAll(`**list**`)`	returns true if this list contains every element from given list
`equals(`**list**`)`	returns true if given other list contains the same elements
`iterator()` `listIterator()`	returns an object used to examine the contents of the list
`lastIndexOf(`**value**`)`	returns last index value is found in list (-1 if not found)
`remove(`**value**`)`	finds and removes the given value from this list
`removeAll(`**list**`)`	removes any elements found in the given list from this list
`retainAll(`**list**`)`	removes any elements *not* found in given list from this list
`subList(`**from**`, `**to**`)`	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
`toArray()`	returns the elements in this list as an array

Check the Java API tutorials for the full list of methods:
http://docs.oracle.com/javase/8/docs/api/?java/util/LinkedList.html

LinkedList in Java Collection API

An example:

```
LinkedList<String> obj = new LinkedList<String>();

/*This is how elements should be added to the linked list*/
obj.add("Ajeet");
 obj.add("Harry");
obj.add("Chaitanya");
obj.add("Steve");
obj.add("Anuj");

/* Displaying linked list elements */
System.out.println("Currently the linked list has following elements:"+obj);

/*Add element at the given index*/
obj.add(3, "Rahul");
obj.addLast("Justin");

/*Remove elements from linked list like this*/
obj.remove("Chaitanya");
obj.removeFirst();

System.out.println("Current linked list is:"+obj);
```

Reading

- Chapters 15 and 17 of the textbook
- Goodrich and Tamassia, ***Data Structures and Algorithms in Java***, (The first, third, fourth, fifth, or sixth edition)
  - Sections 3.2, 3.3