Operations on Arrays

Here is a partial list of some common operations:

- Read A[i]
- Get the length of A
- Sort A in the ascending or descending order
- Update A[i] to x
- Delete A[i]

Java.util methods for Arrays support some of the following operations

equals (A, B) {When are two arrays equal?)

fill (A, x)

copyOf (A, n) (What if n ≠ A.length)

toString(A)

sort(A)

binarySearch(A, x)

```
class Test {
        public static void main (String[] args) {
                int arr1[] = {1, 2, 3};
                int arr2[] = {1, 2, 3};
                if (arr1 == arr2) // Same as arr1.equals(arr2)
                        System.out.println("Same");
                else
                        System.out.println("Not same");
        }
}
```

What should be the output?

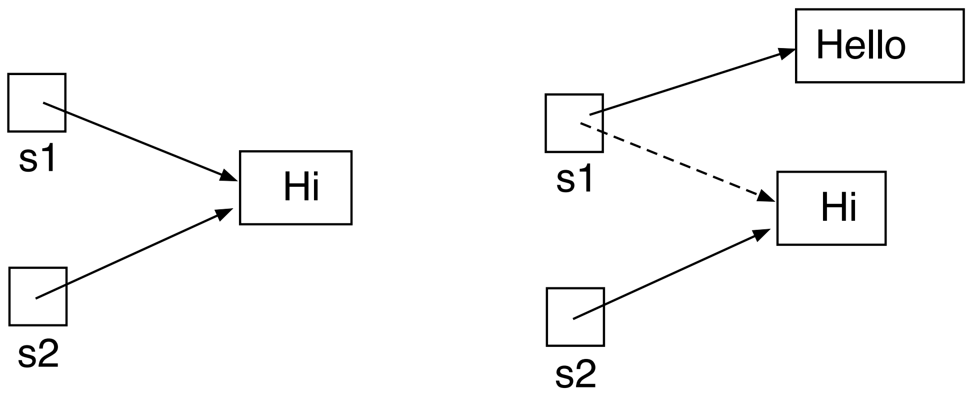
```
import java.util.Arrays;
class Test {
   public static void main (String[] args) {
      int arr1[] = {1, 2, 3};
      int arr2[] = {1, 2, 3};
      if (Arrays.equals(arr1, arr2))
         System.out.println("Same");
      else
         System.out.println("Not same");
   }
}
```

What should be the output?

Strings (not arrays) are immutable

```
String s1 = "Hi";
String s2 = s1; // s1 and s2 now point at the same string - "Hi"
```

Now, you can do nothing to s1 that would affect the value of s2. They refer to the same object - the string "Hi" - but that object is immutable and thus cannot be altered.



Before After

If we do something like this:

```
s1 = "Hello!";
System.out.println(s2); // still prints "Hi"
```

Example of Insertion sort

Algorithm InsertionSort(A)

Input:

An array of comparable elements

Output:

The array with elements arranged in the non-decreasing order

Main idea:

```
for k from 1 to (n-1) do
Insert A[k] at its proper location within A[0], A[1}, ..., A[n-1]
```

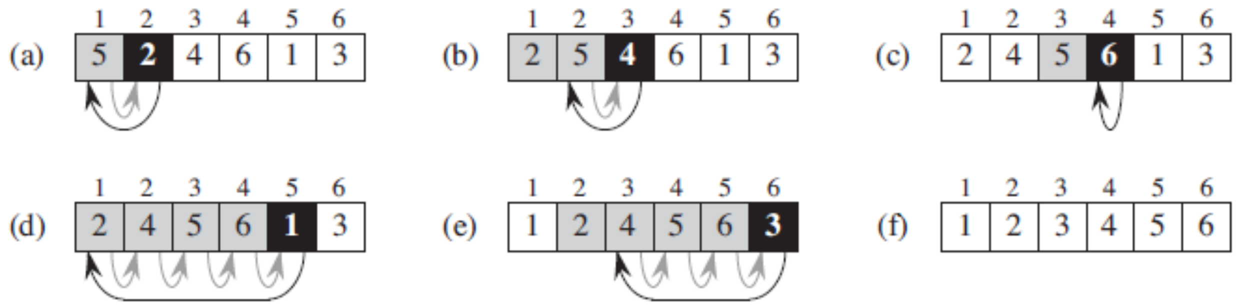


Image source: "Introduction to Algorithms", The MIT Press

Things to observe

After *i* iterations the first *i* elements are ordered.

After each iteration, the next element bubbles through the sorted section until it reaches the right spot:

sorted | unsorted

1 3 5 8 | **4** 6 7 9 2

1 3 **4** 5 8 | 6 7 9 2

Pseudocode for insertion sort

```
for i in 1 to n
    for j in i downto 2
        if array[j - 1] > array[j]
            then swap(array[j - 1], array[j])
        else
            break
```

What is the maximum number of swap operations?

```
public static void insertionSort(int[] A){
      int temp;
      for (int i = 1; i < A.length; i++) {
         for(int j = i ; j > 0 ; j--){
            if(A[j] < A[j-1]){
               temp = A[j];
               A[j] = A[j-1];
               A[j-1] = temp;
            }
         }
      }
      return A;
   }
```

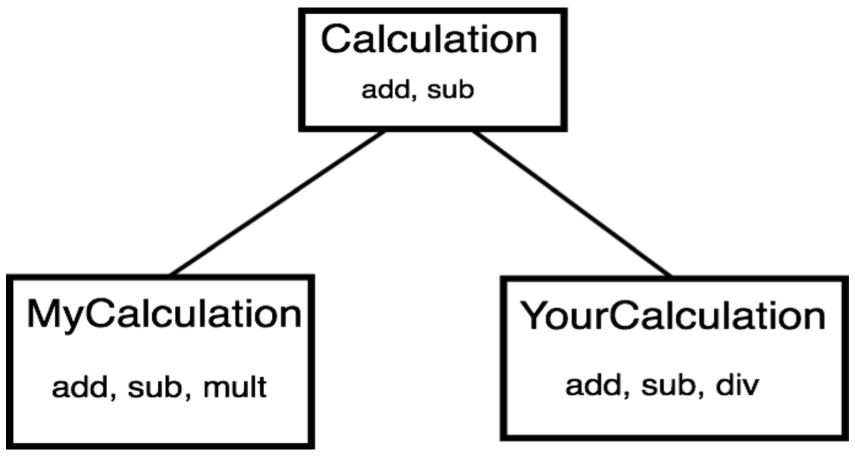
Now, use java.util.Arrays to sort

```
import java.util.Arrays;
         public class HowToSortAnArray {
         public static void main(String[] args) {
         //Array of Integers
         int[] myIntArray = new int[]{31,108,69,4, 81};
         //Sorts array in non-decreasing order
         Arrays.sort(myIntArray);
         //Loop to print Array values to console
         for (int i = 0; i < myIntArray.length; i++) {
         System.out.println(myIntArray[i]);
         }
   }
}
```

Output: 4, 31, 69, 81, 108

Inheritance

One class inherits the properties (methods and fields) of another class.



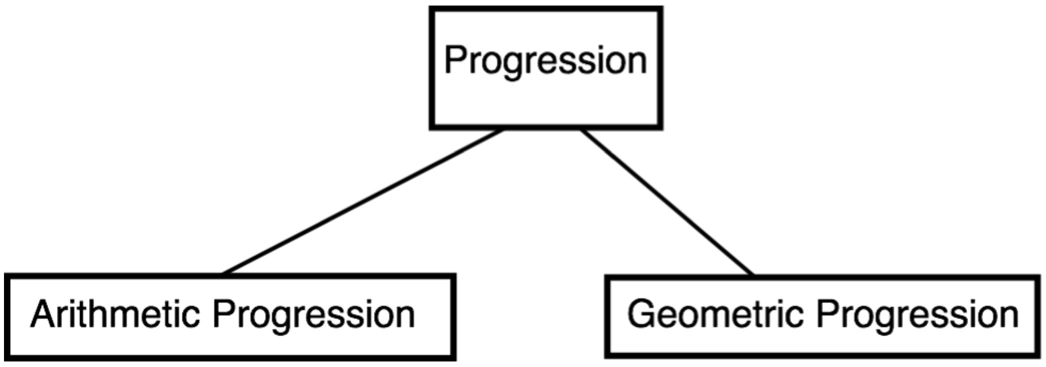
Example 1

```
public class Calculation{
        int z;
        public void addition(int x, int y){
                z = x+y;
                System.out.println("The sum is:"+z);
        }
        public void Subtraction(int x, int y){
                z = x-y;
                System.out.println("The difference is:"+z);
        }
 }

public class My_Calculation extends Calculation{
        public void multiplication(int x, int y){
                z = x*y;
                System.out.println("The product is:"+z);
        }

public static void main(String args[]){
        int a = 20, b = 10;
        My_Calculation demo = new My_Calculation();
                demo.addition(a, b);
                demo.Subtraction(a, b);
                demo.multiplication(a, b);
        }
}
```

Example 2

```
class Super_class{

int num = 20;
//display method of superclass
public void display(){
    System.out.println("This is the display method of superclass");
}

public class Sub_class extends Super_class {
int num = 10;
//display method of sub class
public void display(){
   System.out.println("This is the display method of subclass");
}

public void my_method(){
//Instantiating subclass
    Sub_class sub = new Sub_class();
//Invoking the display() method of sub class
    sub.display();
//Invoking the display() method of superclass
    super.display();
//printing the value of variable num of subclass
    System.out.println("value of num in sub class:"+ sub.num);
//printing the value of variable num of superclass
    System.out.println("value of num in super class:"+ super.num);
  }

  public static void main(String args[]){
    Sub_class obj = new Sub_class();
    obj.my_method();
   }
}
```

Example 3



(Simple) progression

0, 1, 2, 3, 4 …

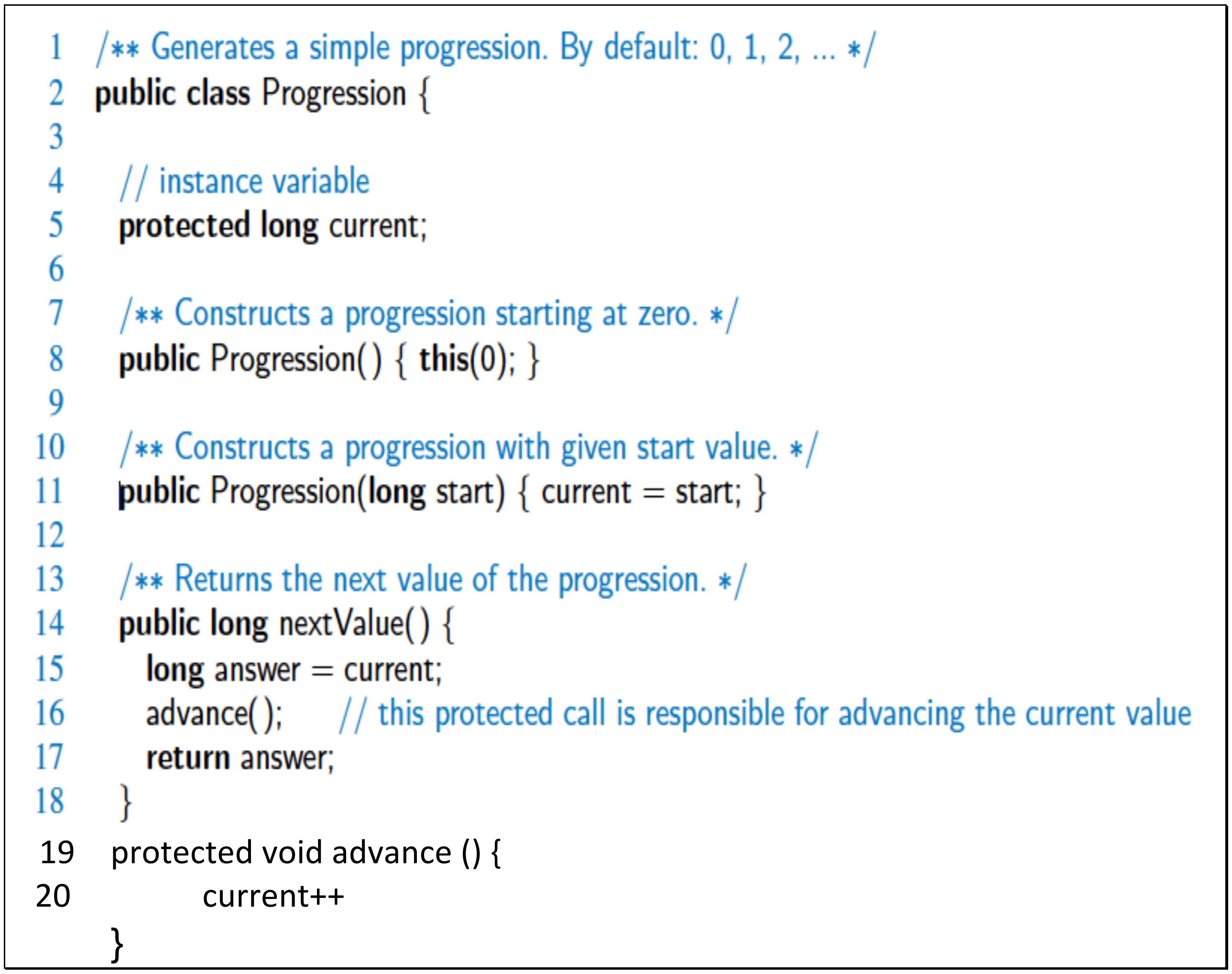
Arithmetic progression

4, 9, 14, 19, 24 …
0, 3, 6, 9, 12,15…

Geometric Progression

1, 2, 4, 8, 16, 32, …
2, 20, 200, 2000, 20000, 200000, …

The difference is how you compute the next value.



```
/** Generates a simple progression. By default: 0, 1, 2, ... */
public class Progression {

  // instance variable
  protected long current;

  /** Constructs a progression starting at zero. */
  public Progression( ) { this(0); }

  /** Constructs a progression with given start value. */
  public Progression(long start) { current = start; }

  /** Returns the next value of the progression. */
  public long nextValue( ) {
    long answer = current;
    advance( );      // this protected call is responsible for advancing the current value
    return answer;
  }
 protected void advance () {
         current++
 }
```

Study how two other sub-classes: Arithmetic Progression and Geometric Progression can be created from this super-class.

What can a subclass do

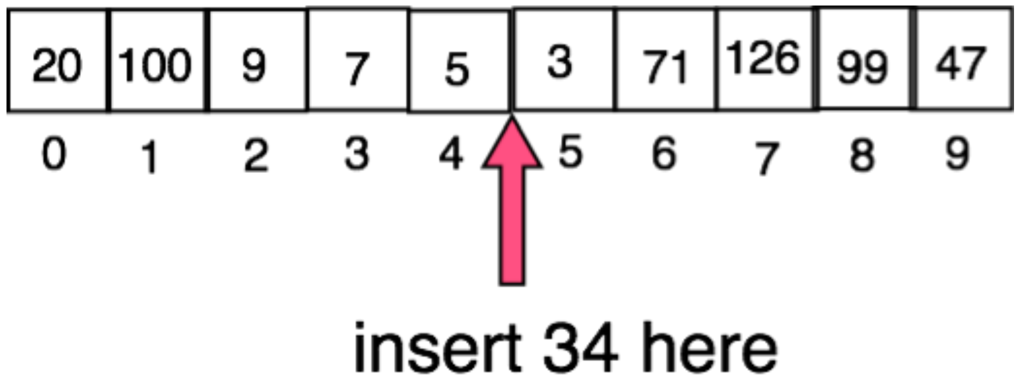
1. It can declare new fields
2. It can declare new methods
3. It can override old methods with new implementations

```
class Base {
int x = 100;
}

class Sup1 extends Base {
        int x = 200;
        void Show()    {
                System.out.println(super.x);
                System.out.println(x);
        }

public static void main(String[] args)    {
        new Sup1().Show();
        }
}
```

What is the output?

Lists

Consider the problem of storing a set of items, where the order is important (as in a ***phone directory***, or ***student records*** at the university). How will you store them? Use an array?



Advantages

Know the index, and locate the item!

Disadvantages

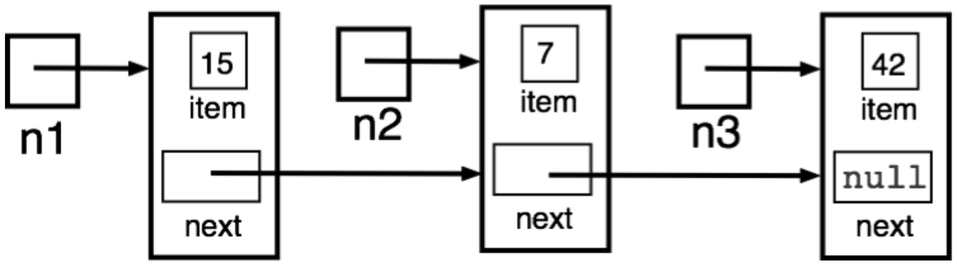
1. Insertion of a new item at the beginning or the middle of the takes time proportional to the length of the array.
2. Arrays, once declared, have a fixed length. To increase the length of the array, there is the overhead of copying the array.

(However, ArrayList facilitates this. More on this later.)

Linked list resolves the problem.

Linked List

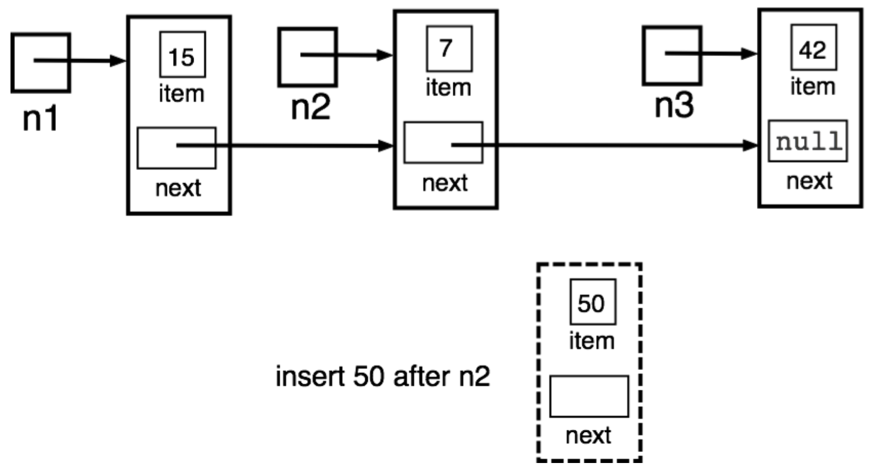
Example of a recursive data type:

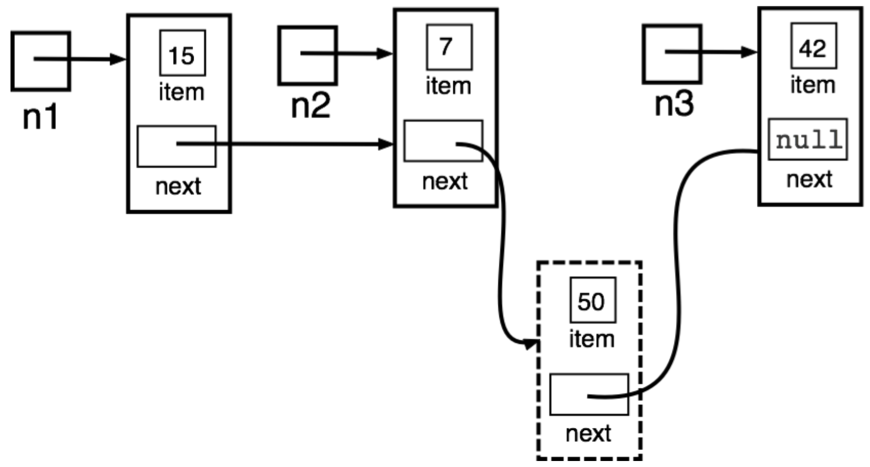


```
public class ListNode {
        {int item;
        ListNode next;        //recursive definition
}
// Construct three ListNodes
ListNode n1 = new ListNode();
ListNode n2 = new ListNode();
ListNode n3 = new ListNode();
n1.item = 15;
n2.item = 7;
n3.item = 42;
n1.next = n2;
n2.next = null;
n3.next = 42;
```

Why is this a good idea?

1. Inserting an item takes "**constant time**."

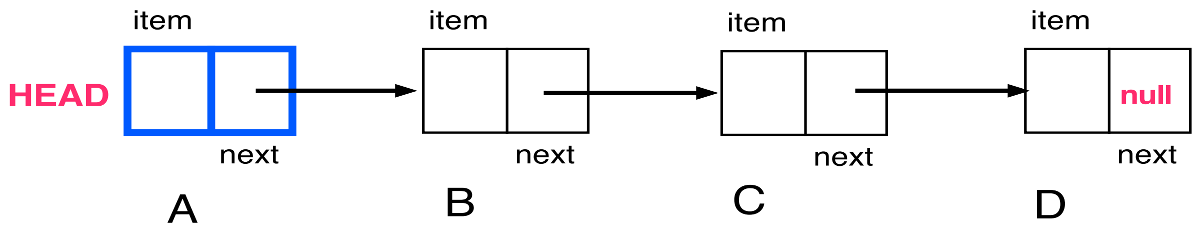




2. Linked lists can grow in size as long as memory is available. Contrast this with the way an array grows.

This is also called Singly Linked List. Why?

The starting node is called the HEAD of the linked list. If the list is empty, then the head is null. The other end is the TAIL node.

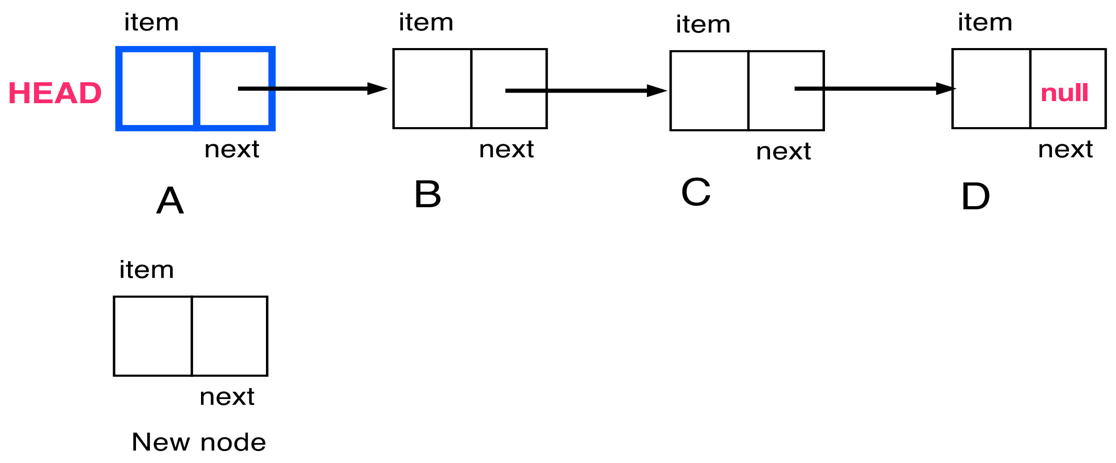


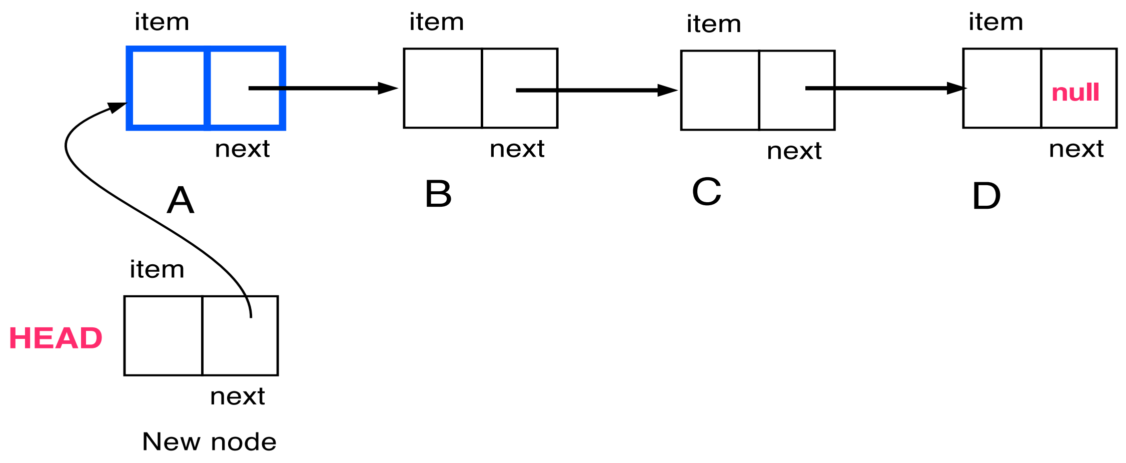
Add a node (N) at the head of a singly linked list

Newest node = N

Newest.next = head

Head = newest





How to locate the **TAIL** of the list? Start from the HEAD node, and reach a node with next=null by link hopping.

Removing the HEAD:

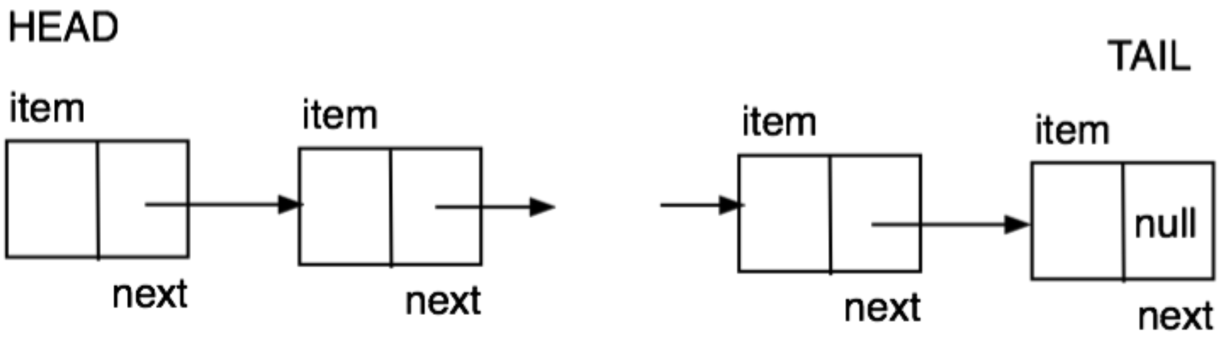
HEAD = HEAD.next

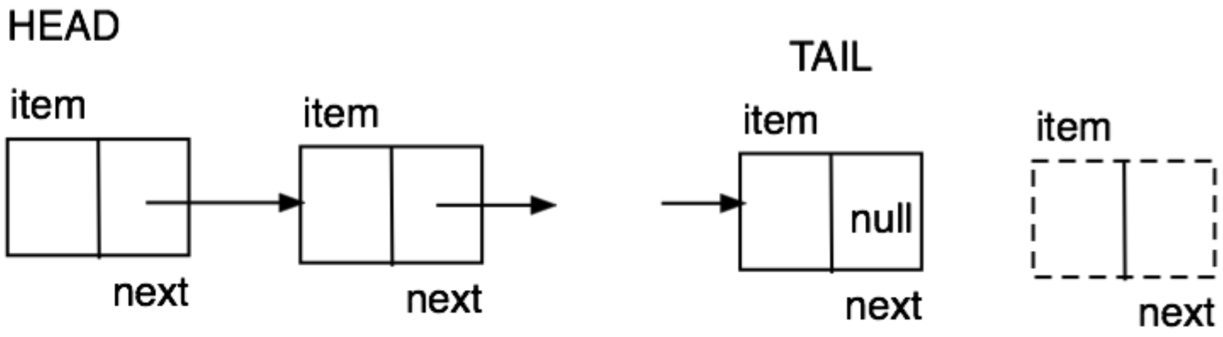
If HEAD is the only node, then the list becomes empty.

How will you remove the TAIL node?

Start from the HEAD node;
Reach the TAIL by link hopping;
Then what?

To remove the tail, you have to change the "next" field of the ListNode "before the last one" to null. But how do you reach there from the tail? Can we walk backwards?





ArrayList implements a Dynamic Array

```
import java.util.*;
public class ArrayListDemo {
public static void main(String args[]) {
        // Create an array list
        ArrayList al = new ArrayList();
        System.out.println("Initial size of al: " + al.size());
        // Add elements to the array list
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        al.add(1, "A2");
        System.out.println("Size of al now: " + al.size());
        // Display the array list
        System.out.println("Contents of al: " + al);
        // Remove elements from the array list
        al.remove("F");
        al.remove(2);
        System.out.println("Size after deletions: " + al.size());
        System.out.println("Contents of al: " + al);    } }
```

Output

```
    Initial size of al: 0
    Size of al now: 7
    Contents of al: [C, A2, A, E, B, D, F]
    Size after deletions: 5
    Contents of al: [C, A2, E, B, D]
```