Data Structures and Algorithms

XMUT-COMP 103 - 2023 T1

Maps, Sorting

Mohammad Nekooei

School of Engineering and Computer Science

Victoria University of Wellington

Using Sets: Vocabulary, another requirement:

- Vocabulary:
  - Given a file of words (from a book)
  - Count the number of words and the number of distinct words.
  - Print out the vocabulary:
    - (a) all words, alphabetically
    - (b) the top 100 words (by frequency)

  - How can we sort the words?
  - How can we keep track of the frequency of words, and then sort by that?

  - We need to store each word we find PLUS how many times we have seen it.

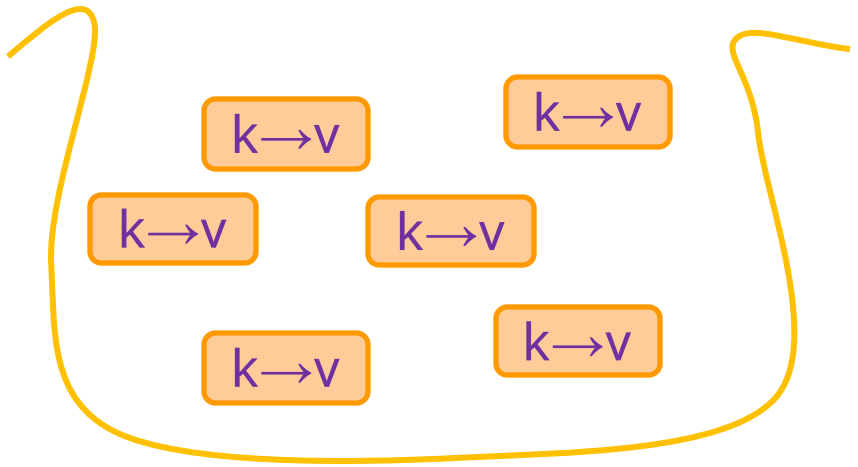
    the:50 we:8 sun:1 play:2 in:22 cat:20 hat:18 play:4

This is a "Map" – a mapping from "keys" to associated "values"
here: mapping from words to their counts

Maps

A Map is

- a collection of data with an index:
  - index → data
- a collection of key → value pairs
- a mapping from keys to values

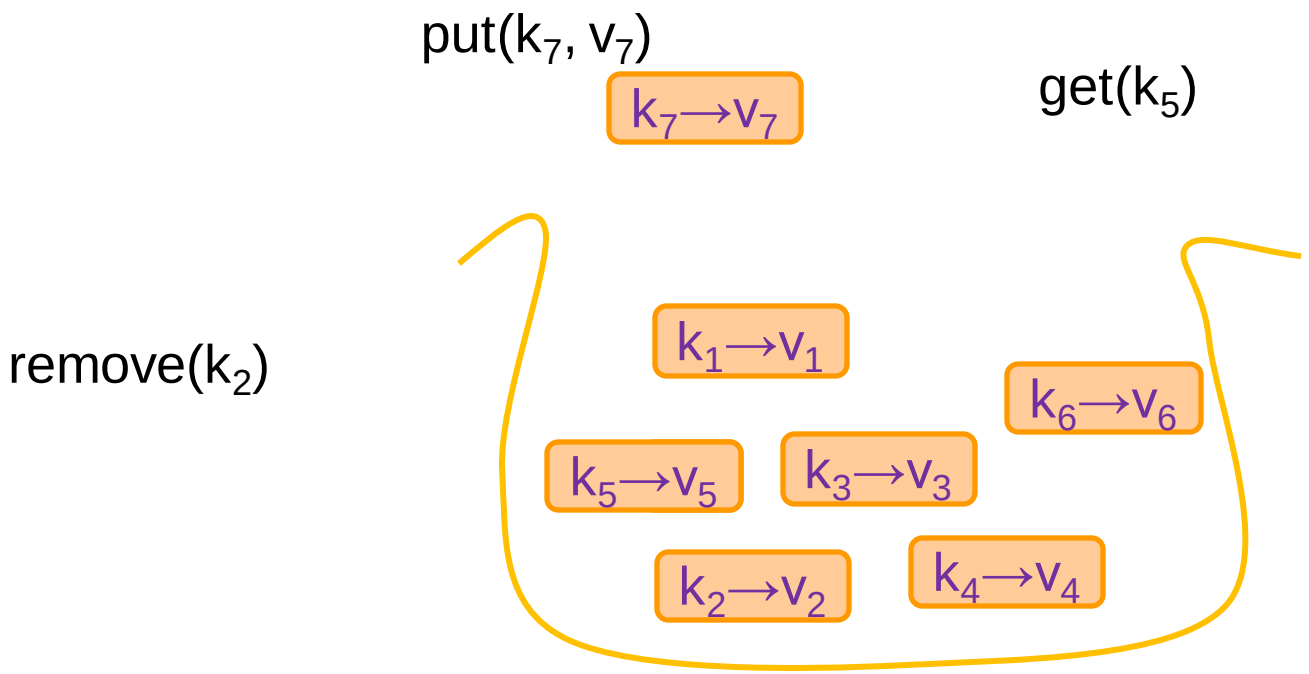


- eg: University: student ID → student
- eg: phonebook: name → phone number
- eg: Bank: account number → account
- eg: Vocab: word → count

- Note: the keys must be unique (the keys are a Set); values can be duplicated

Maps

- Fundamental operations:
  - put(key, value)
  - get(key)
  - remove(key)



Maps in Java

- Map is part of the Collections library (but a *Map* isn't a *Collection*)
- To declare a Map in Java, need two types: key and value
  - `private Map<String, String> dictionary = new HashMap<String, String>();`
    - *given a word, get the definition*
  - `private Map<String, Person> members = new HashMap<String, Person>();`
    - given a name, get the Person object for that person.
  - `private Map<String, Integer> vocab = new HashMap<String, Integer>();`
    - given a word, get the count of that word.
- Stores the key-value pairs in Map.Entry objects
- Implementations: HashMap, TreeMap (like HashSet and TreeSet)
  - HashMap hashes the keys to work out where to store the key-value pair
  - TreeMap uses the ordering of the keys to construct the binary search tree to store the pairs.

Maps in Java: Operations

- more operations:
  - get(key) → value (or null, if none)
  - put(key, value) → old value (or null if no old value)

  - remove(key) → value removed (or null, if none)
  - replace(key, newvalue) → old value (if no old value, doesn't put, and returns null)
  - containsKey(key),
  - containsValue(value) *(expensive! – has to search through all the pairs)*
  - clear(), isEmpty(), size()

  For iterating through a Map: [can't write: **for** (?? item : myMap){….]
  - keySet() → Set<keytype>
  - values() → Collection<valuetype>
  - entrySet() → Set<Map.Entry<keytype, valuetype>>

TreeMap - example

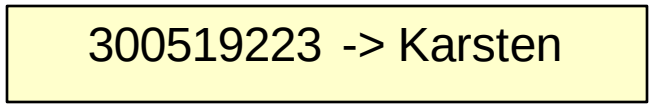
- I want to have a map that stores students preferred name using their student id as a key in a field of a class.

```
private Map<Integer, String> dictionary  = new TreeMap<Integer, String>();
```

- I start adding Karsten

```
dictionary.put(300519223, “Karsten”);
```

TreeMap - example

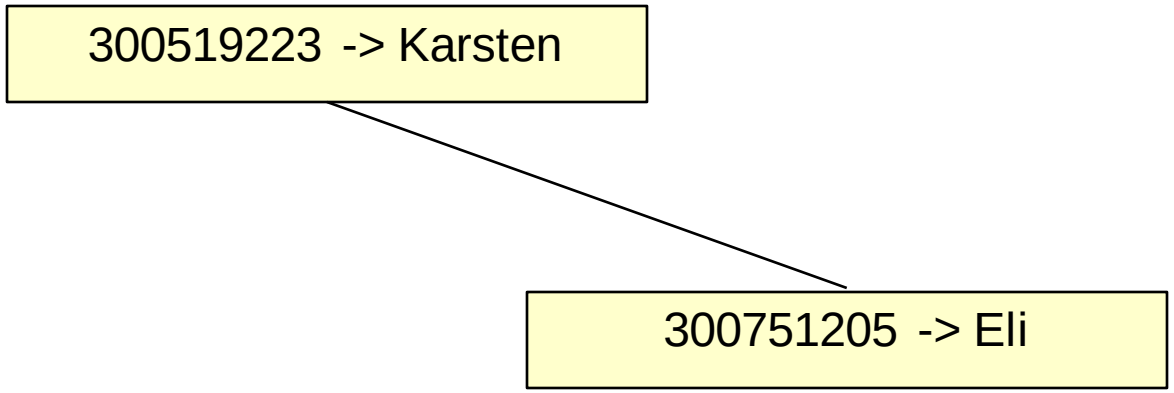


TreeMap - example

- I then add Eli

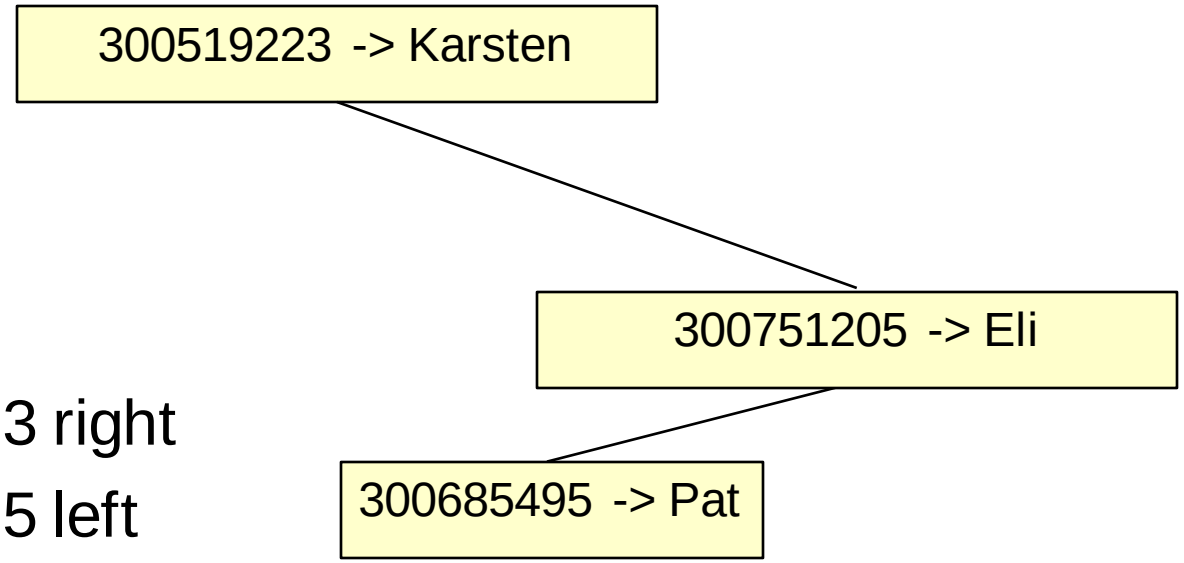
```
dictionary.put(300751205, “Eli");
```

300751205 > 300519223



300751205 > 300519223, so added on the right branch.

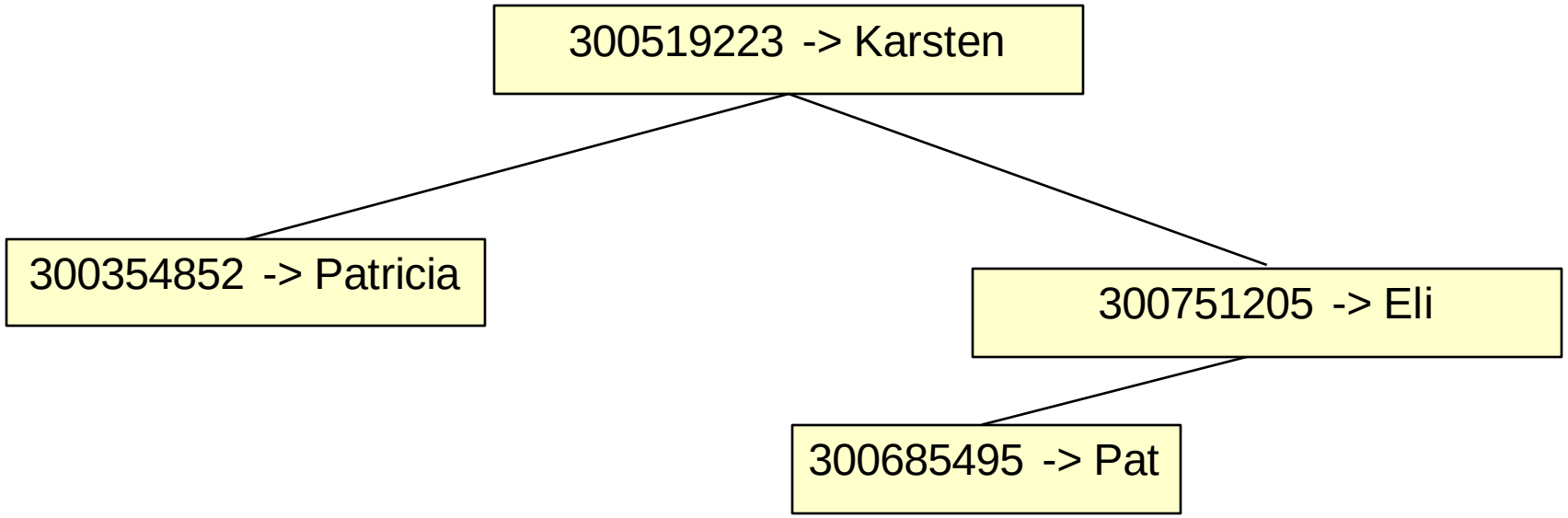
dictionary.put(300685495, “Pat");



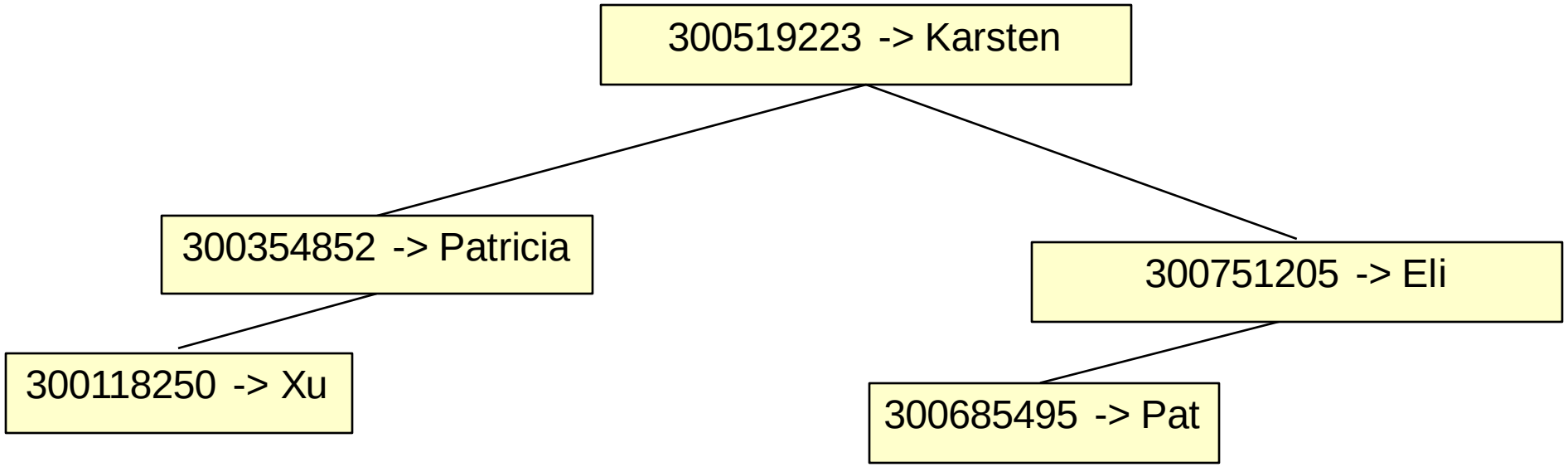
- 300685495 > 300519223 right
- 300685495 < 300751205 left

```
dictionary.put(300354852, “Patricia");

dictionary.put(300118250, “Xu");

dictionary.put(300679800, “Mark");

dictionary.put(300014172, “Joe");

dictionary.put(300424999, “Bo");

dictionary.put(300749861, “Bo"); //not the same Bo

dictionary.put(300845901, “Katherine");
```

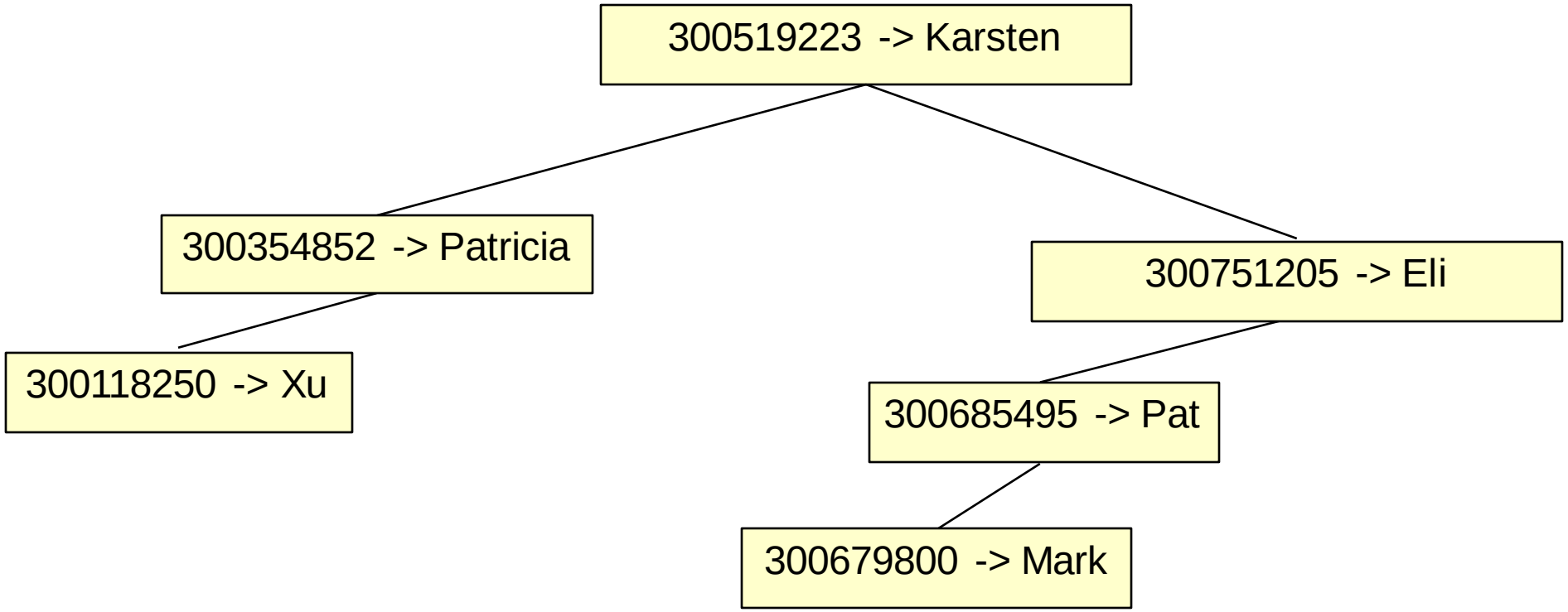
300354852 Patricia



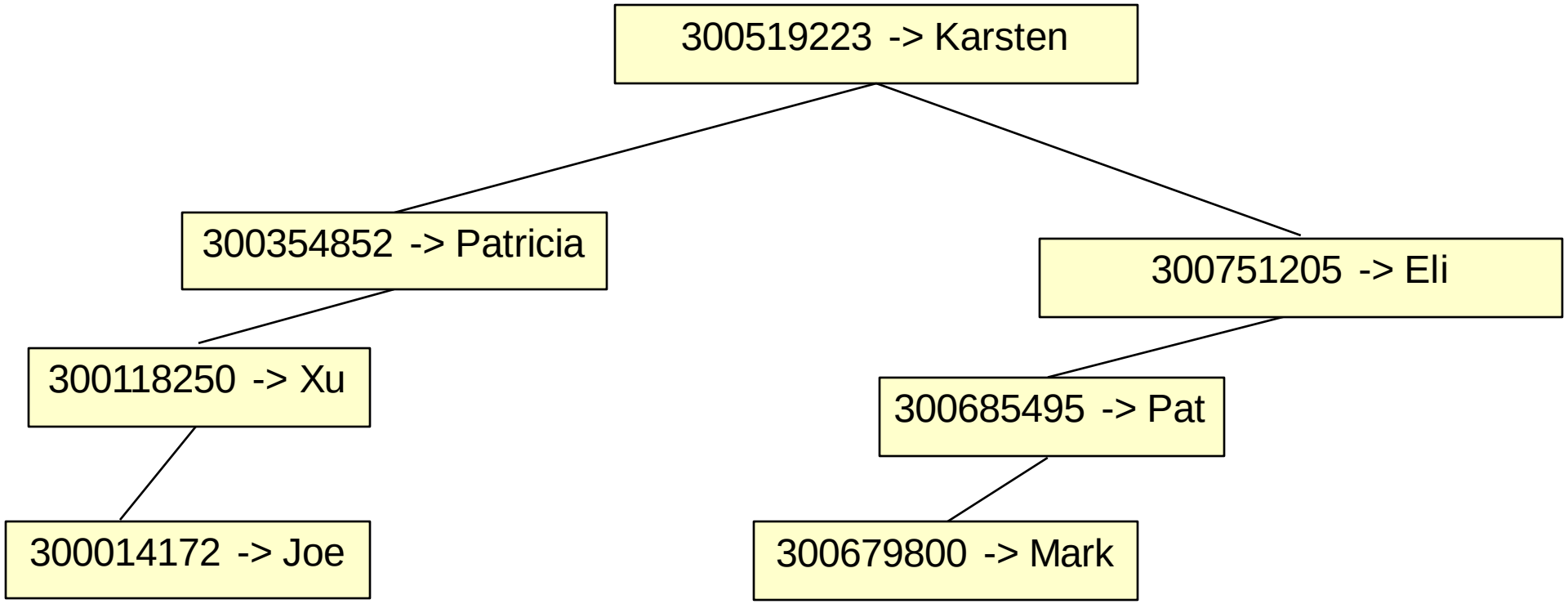
300118250 Xu



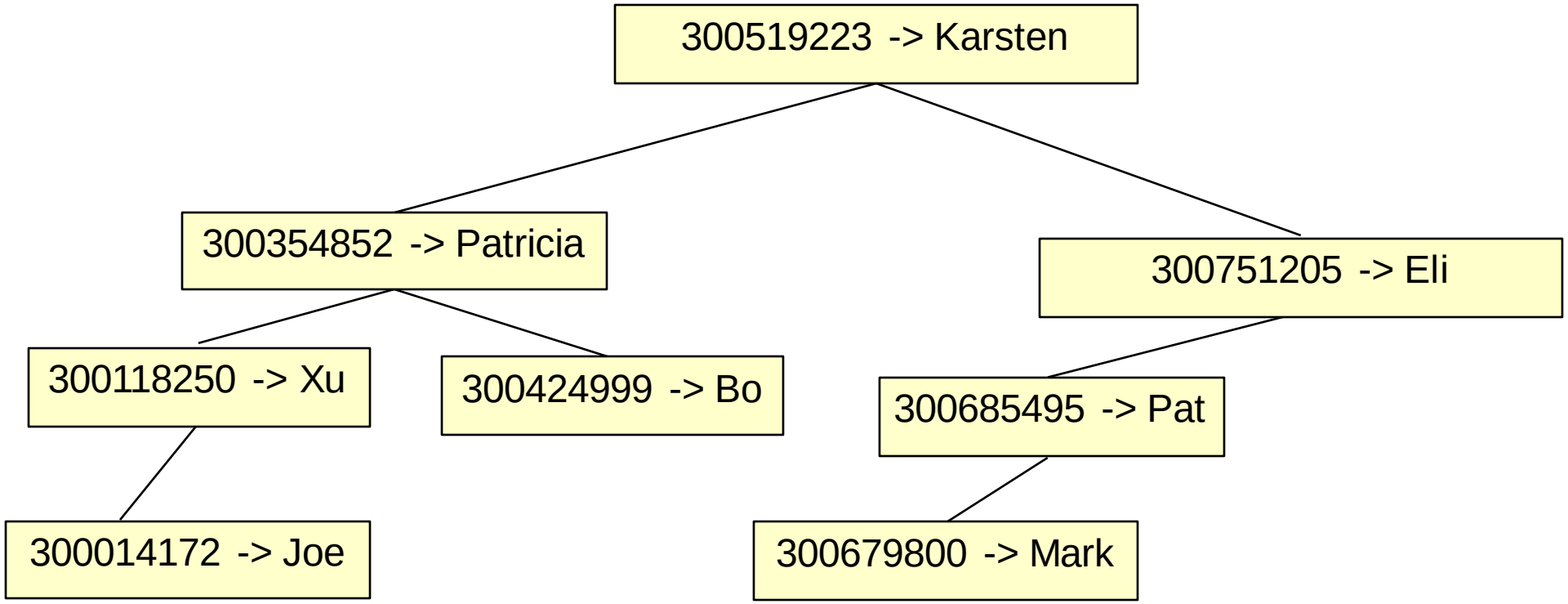
300679800 Mark



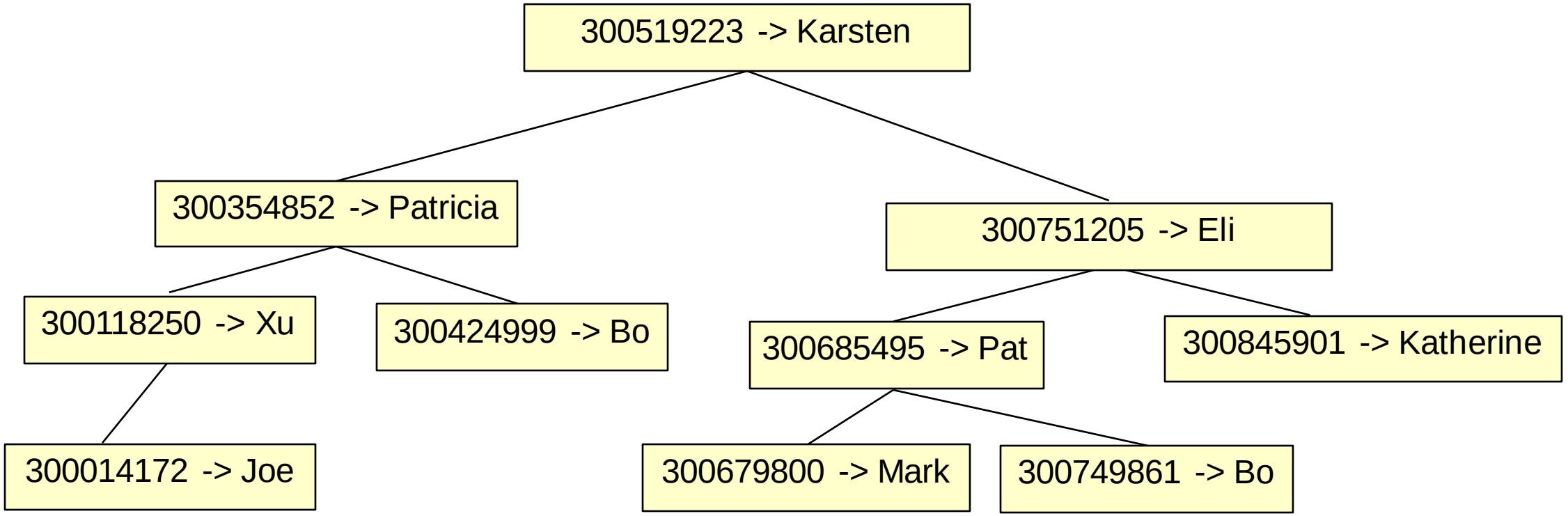
300014172 Joe



300424999 Bo

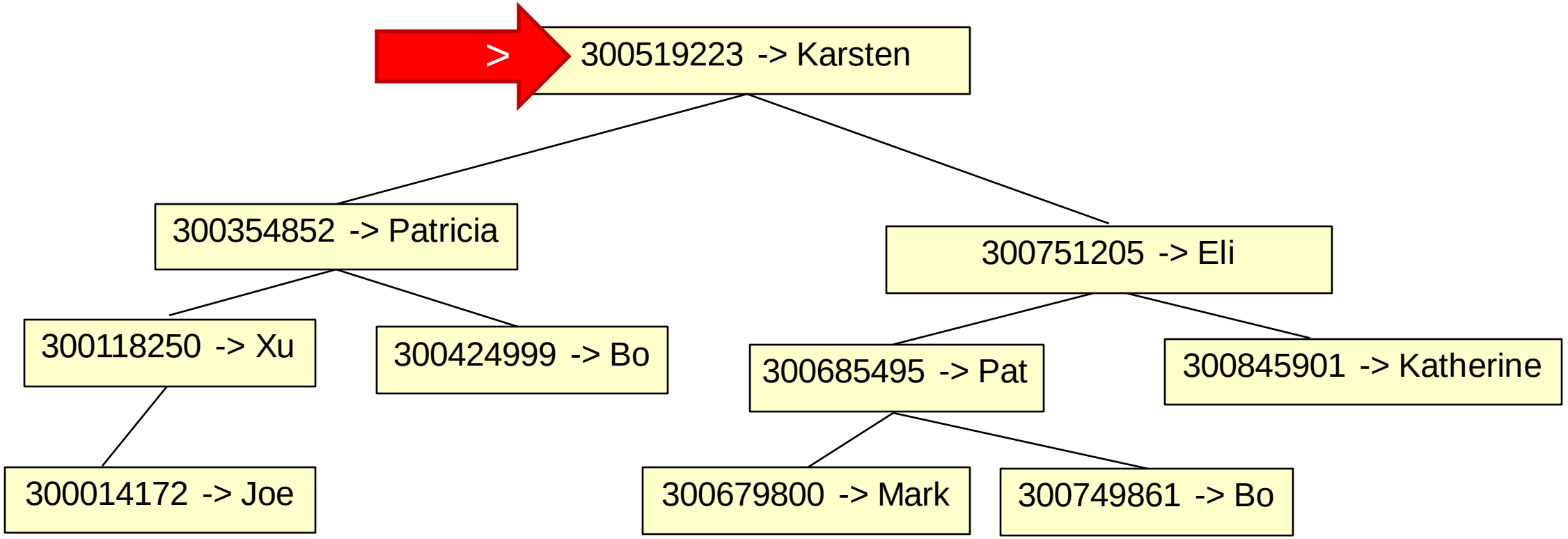


300845901 Katherine



```
dictionary.put(300845901, “Kat"); // Oops her preferred name is Kat
```

Find 300845901 and change to Kat



300519223 -> Karsten

300354852 -> Patricia

300751205 -> Eli

300118250 -> Xu

300424999 -> Bo

300685495 -> Pat

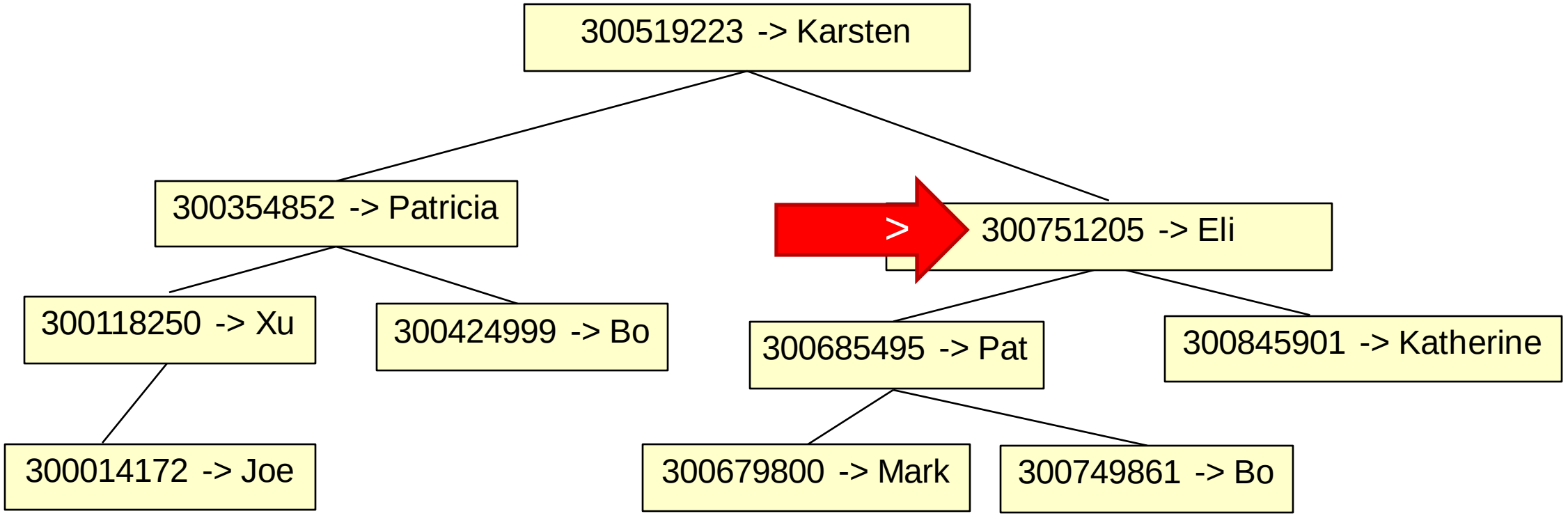
300845901 -> Katherine

300014172 -> Joe

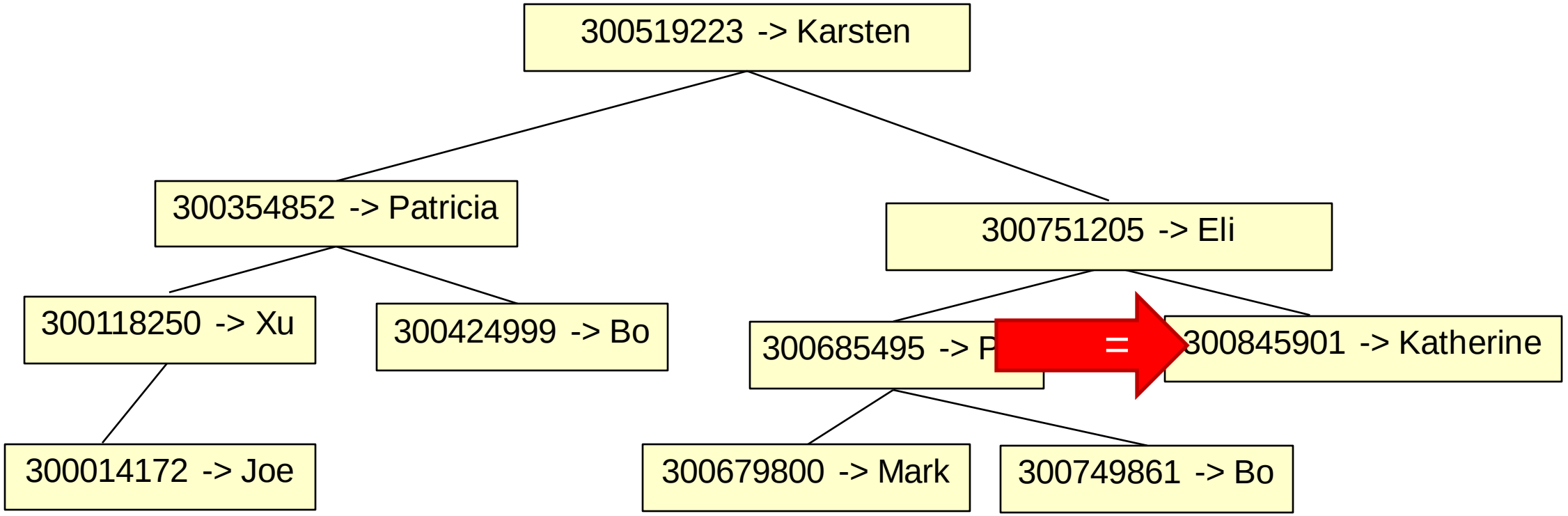
300679800 -> Mark

300749861 -> Bo

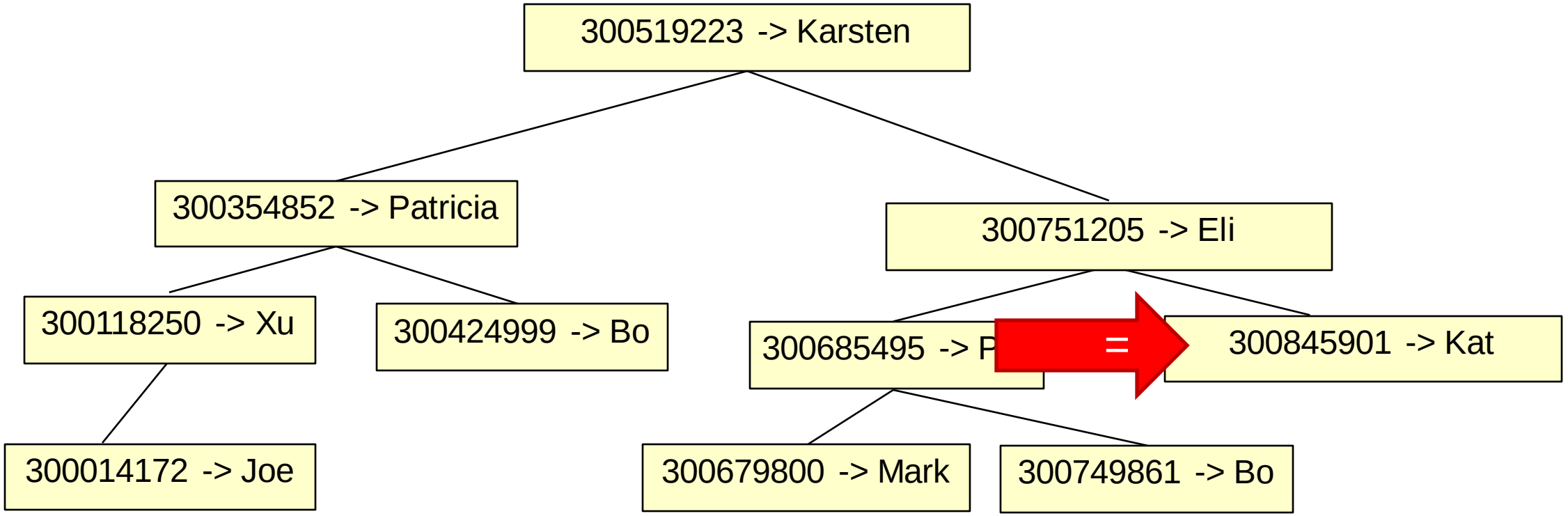
Find 300845901 and change to Kat



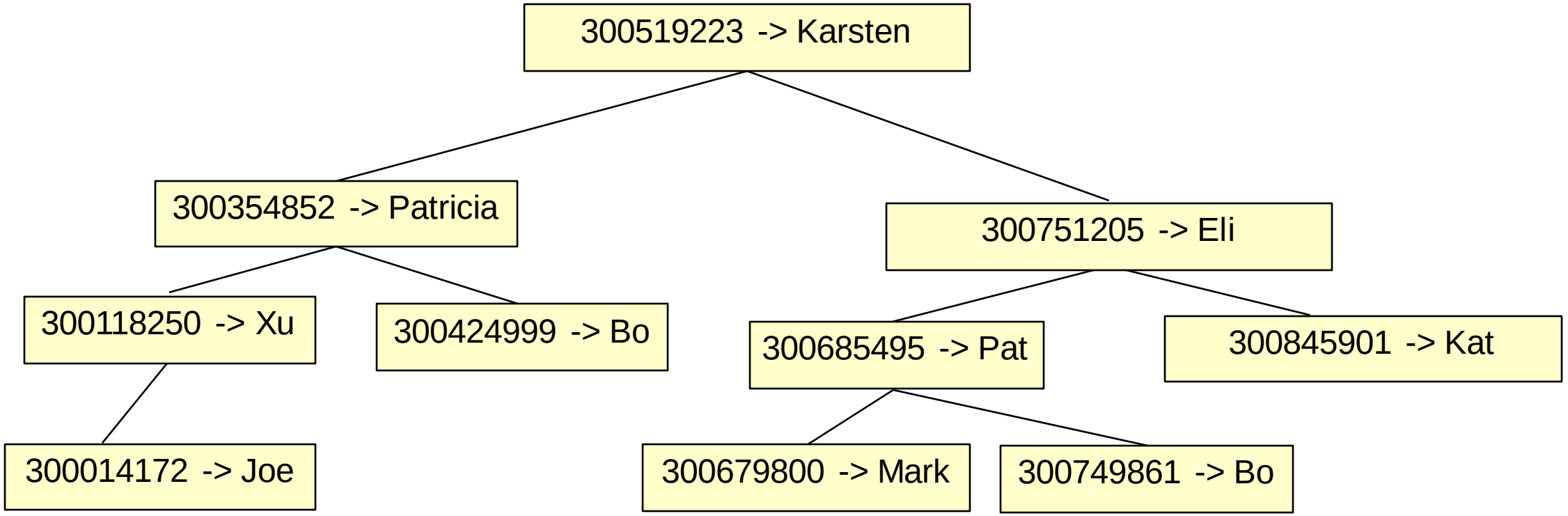
Find 300845901 and change to Kat



Find 300845901 and change to Kat

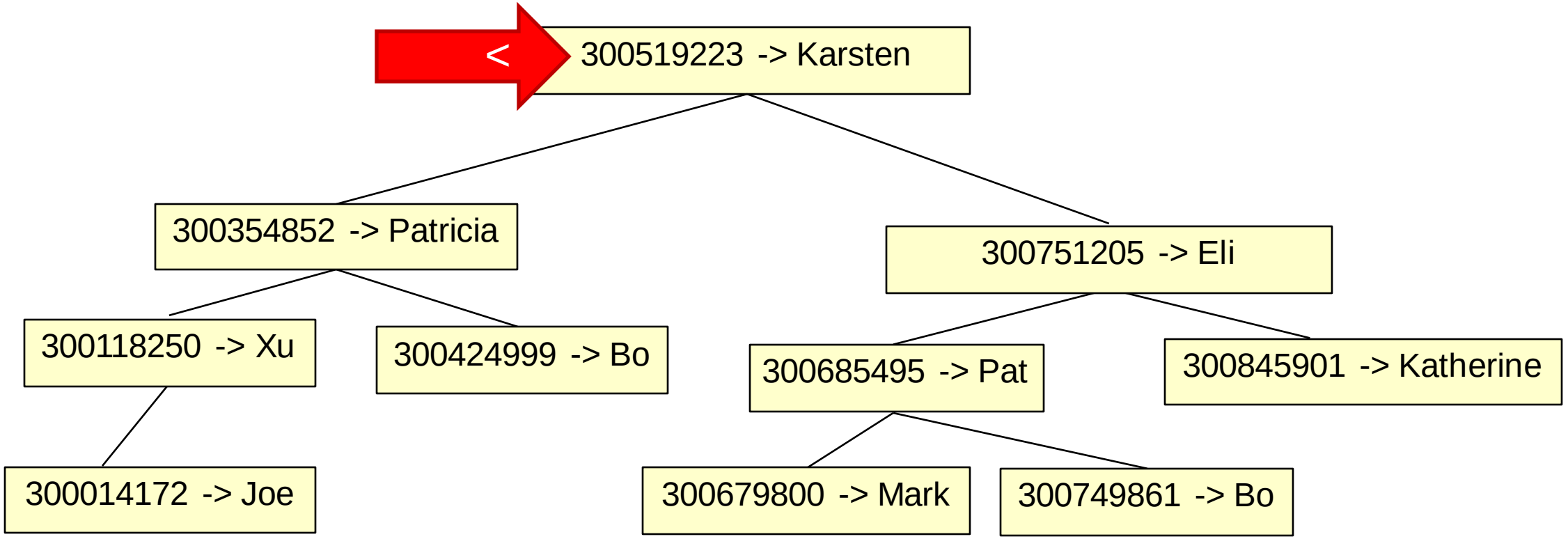


Find 300845901 and change to Kat

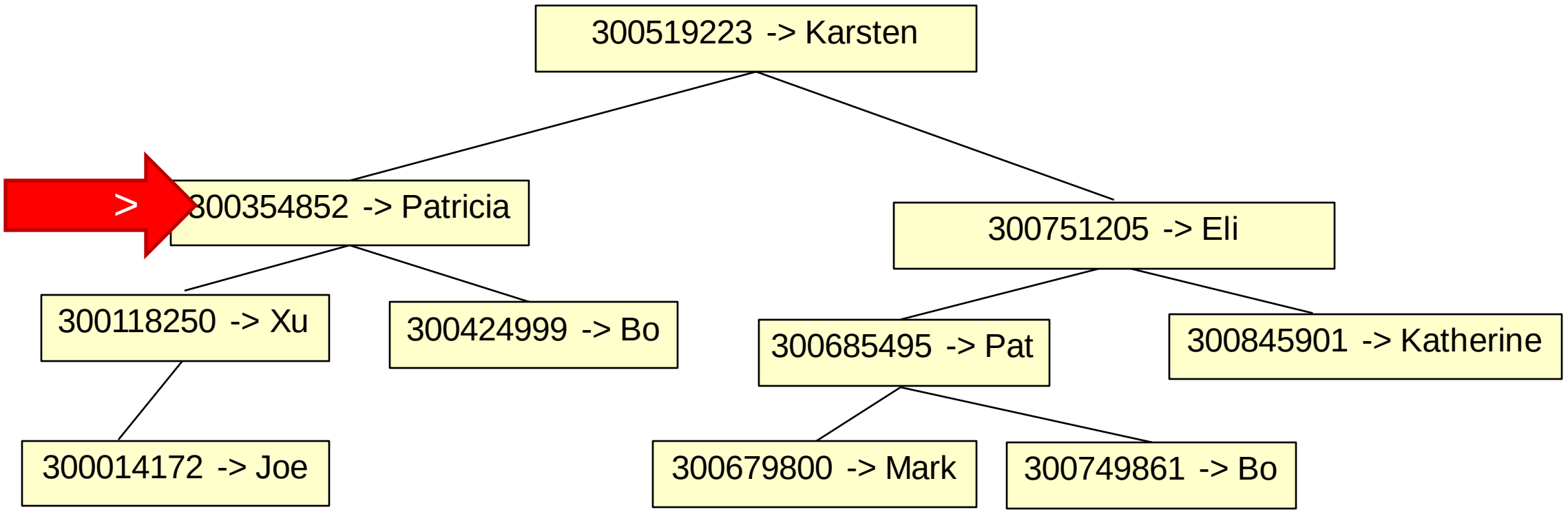


```
String  name = dictionary.get(300424999);
```

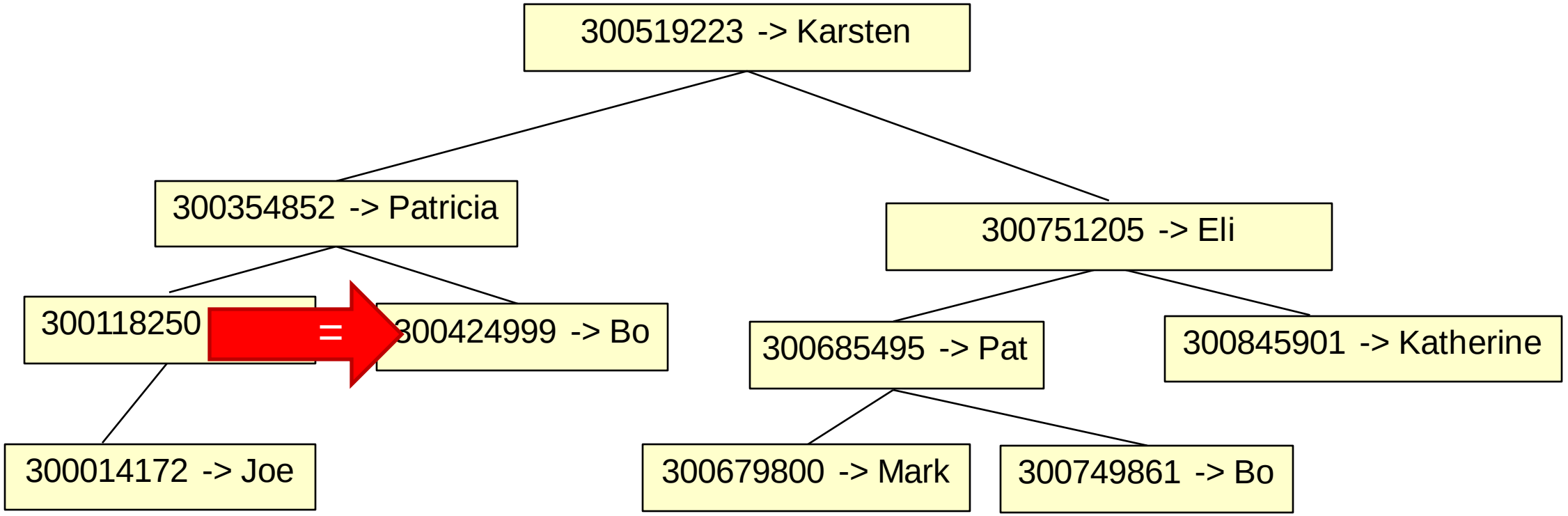
get(300424999)



get(300424999)

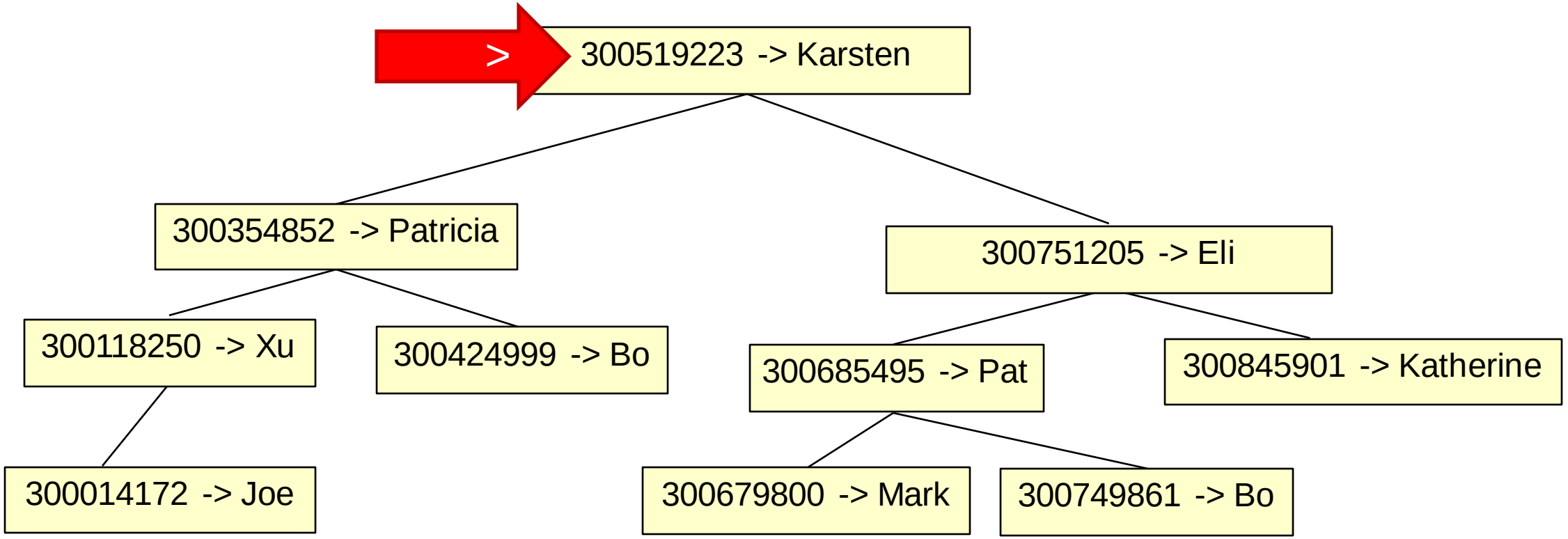


get(300424999) -> “Bo”



```
dictionary.remove(300685495);
```

remove(300685495)



300519223 -> Karsten

300354852 -> Patricia

300751205 -> Eli

300118250 -> Xu

300424999 -> Bo

300685495 -> Pat

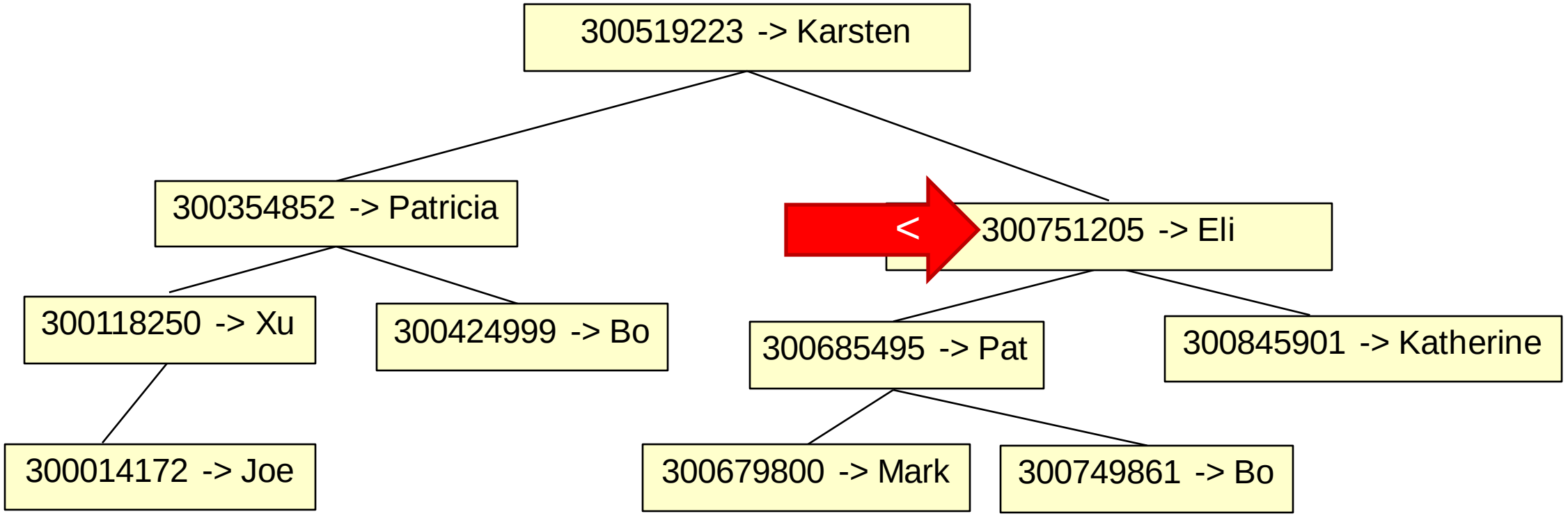
300845901 -> Katherine

300014172 -> Joe

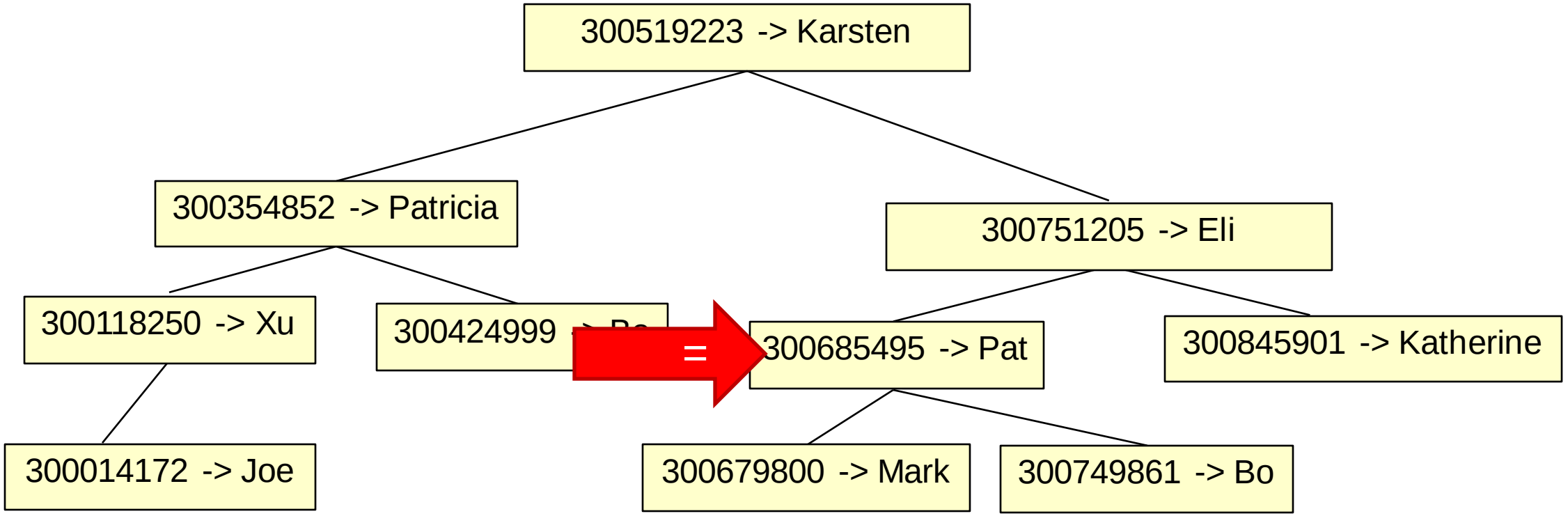
300679800 -> Mark

300749861 -> Bo

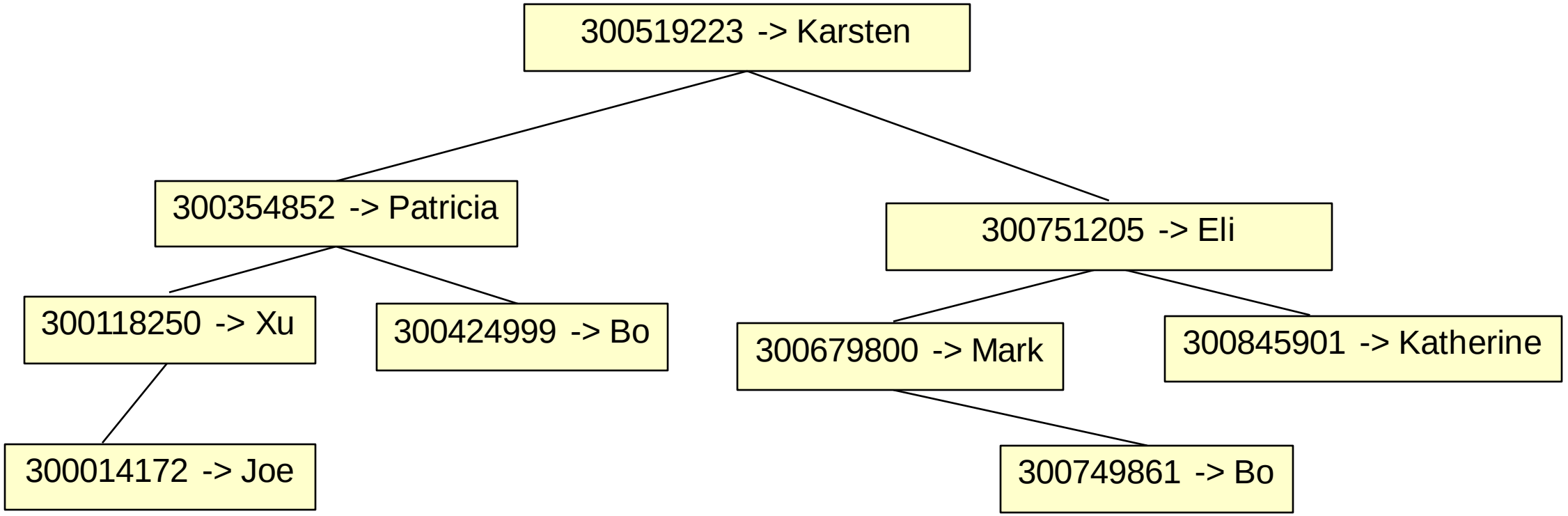
remove(300685495)



remove(300685495)



remove(300685495)



Vocabulary, yet again

- Vocabulary:
  - Given a file of words (from a book)
  - Count the number of words and the number of distinct words.
  - Print out the vocabulary:
    - (a) all words, alphabetically
    - (b) the top 100 words (by frequency)
  - How can we sort the words?
  - How can we keep track of the frequency of words, and then sort by that?

Using Sets: Vocabulary, another requirement:

- Vocabulary:
  - Given a file of words (from a book)
  - Count the number of words and the number of distinct words.
  - Print out the vocabulary:
    - (a) all words, alphabetically
    - (b) the top 100 words (by frequency)

  - How can we sort the words?
  - How can we keep track of the frequency of words, and then sort by that?

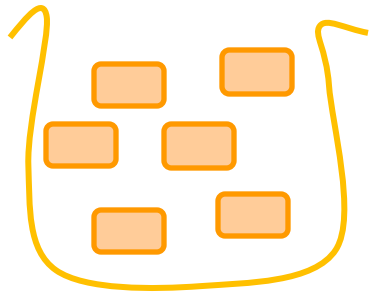
  - We need to store each word we find PLUS how many times we have seen it.

    the:50 we:8 sun:1 play:2 in:22 cat:20 hat:18 play:4

This is a "Map" – a mapping from "keys" to associated "values"
here: mapping from words to their counts

Counting Occurrences

- Vocabulary count:
  - Given a file of words (from a book)
  - Count the number of occurrences of each distinct word.



    - open the file
    - initialise Vocab = new Map <String, Integer>()
    - for each word in the file
      - if the word is not in the vocab, then
        - put(word, 1)
      - else
        - put(word, get(word)+1)
    - close the file

    - sort vocabulary by count and print out first 100 words ??????

Building the vocab map.

```
Map<String, Integer> vocab = new HashMap<String, Integer>();
Scanner sc = new Scanner(Path.of(filename));
while (sc.hasNext()){
    String word = sc.next();
    if (! vocab.containsKey(word)){
        vocab.put(word, 1);
    }
    else {
        vocab.put(word, (vocab.get(word)+1) );
    }
}

// sort vocabulary by count and print out first 100 words     ??????
```

Sorting

- Sorting a list is easy:
  - Collections.sort(myList);
  - Pretty fast!

- But only if the items in the list are Comparable
  - Lists of String, Lists of numbers, …

- Sorting a Set is easy:
  - Use a TreeSet instead of a HashSet, or
  - Copy HashSet to a list and sort:

```
List<String> sortedVocab = new ArrayList<String>(vocab);    or   addAll(vocab);
Collections.sort(sortedVocab);
```

- But only if the items in the list are Comparable (eg, String, Integer, Double)

Sorting items

- What about Lists of items that aren't Comparable?

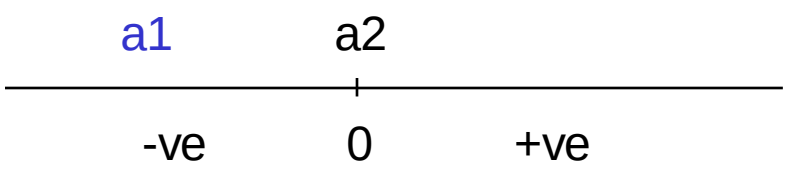
- Sort needs a method to compare two values and say which comes first
  - Comparable objects have such a method (called compareTo)
  - You can give sort such a method for non-Comparable objects
    - parameters are the two values
    - returns an int

      -1 if the first value comes earlier in the ordering (or any negative integer)

      1 if the second value comes earlier in the ordering (or any positive integer)

      0 if the two values are equal in the ordering

  - Collections.sort(myList, (Type a1, Type a2) -> { **if** (*a1 is before a2*) { **return** -1; }
    else if (*a1 is after a2*) { **return** 1; }
    else { **return** 0; } });



Sorting Students by City

- If classList is a List of Student objects, which have a getCity() method:

- Collections.sort(classList,

```
(Student s1, Student s2) -> {
    if      (s1.getCity().compareTo(s2.getCity()) < 0 )  { return -1; }
    else if (s1.getCity().compareTo(s2.getCity()) > 0 )  { return  1; }
    else                                                 { return  0; }
 } );
```

OR

- Collections.sort(classList,

```
(Student s1, Student s2) -> { return ( s1.getCity().compareTo(s2.getCity() );  } );
```

Sorting Students by Country and then City

If classList is a List of Student objects, which have a getCountry() and getCity() methods:

```
Collections.sort(classList,
                 (Student s1, Student s2) -> {
                     if     (s1.getCountry().compareTo(s2.getCountry()) < 0 )  { return -1; }
                     else if (s1.getCountry().compareTo(s2.getCountry()) > 0 )  { return  1; }
                     else {   return ( s1.getCity().compareTo(s2.getCity()) ); }  // same country
                 } );
```

OR

```
Collections.sort(classList, this::compareByCountryCity);

public int compareByCountryCity(Student s1, Student s2) {
    if     (s1.getCountry().compareTo(s2.getCountry()) < 0 )  { return -1; }
    else if (s1.getCountry().compareTo(s2.getCountry()) > 0 )  { return  1; }
    else {   return ( s1.getCity().compareTo(s2.getCity()) ); }  // same country
}
```

Sorting items

- What about TreeSets or TreeMaps of items that aren't Comparable?

- Need to give the Set (or Map) a method to compare two values and say which comes first

  - Set<Face> crowd = **new** TreeSet<Face>((Face f1, Face f2) -> {
    if (f1.size() < f2.size()){ **return** -1; } **else**….}));

  - Set<Face> crowd = **new** TreeSet<Face>((Face f1, Face f2) -> {
    return (f1.size() - f2.size()));

  - Map<Person,Integer> counter = **new** TreeMap<Person,Integer>(
    (Person p1, Person p2) -> {
    return (p1.getName().compareTo(p2.getName()));
    });

Recap: the vocab map.

```
Map<String, Integer> vocab = new HashMap<String, Integer>();
Scanner sc = new Scanner(Path.of(filename));
while (sc.hasNext()){
    String word = sc.next();
    if (! vocab.containsKey(word)){
        vocab.put(word, 1);
    }
    else {
        vocab.put(word, (vocab.get(word)+1) );
    }
}

// sort vocabulary by count and print out first 100 words     ??????
```

Sorting Vocab with a compare method

- Vocab
  - Map of [word-count] pairs
  - Sort by the count (largest first)
  - How do we get a list of the pairs?
  - Maps consist of key-value pairs of type Map.Entry<K, V>
    - Therefore, make a List<Map.Entry<K,V> >
  - How do we sort the List?
    - Using a comparison method:

```
List < Map.Entry<String, Integer> > list = new ArrayList< Map.Entry<String, Integer> >();

Collections.sort(list, (Map.Entry<String, Integer> p1, Map.Entry<String, Integer> p2) ->{
                        return ( p2.getValue()-p1.getValue() );
                    }));
```

Is this the right way round?

[*Run the VocabularyMap program.*]

Sorting Comparable objects vs using a lambda

- Sorting Lists of Comparable values is MUCH easier.
- Strings, Double, Int are all Comparable.
- How could we make Student, Person, City, … Comparable?

Note: it would only work if there is just one obvious way or ordering the items.

Making Objects that are Comparable

What does Comparable mean? How can you make your objects Comparable?

- The class must declare that it is Comparable

```
public class Student implements Comparable<Student>{…
```

- The class must have a compareTo(…) method:

```
public int compareTo(Student other){
    if ( this.sID < other.sID)        { return -1; }
    else if (this.sID > other.sID)) { return  1; }
    else                              { return  0; }    });
```

Now can use sort directly on lists of students:

```
Collections.sort(classList);
```

Making Classes usable with Collections

- Making a class Comparable makes it easier to use with sort, TreeSet, TreeMap,…

- There are other methods that may also be needed to make your class easy to use:
  - compareTo(…) [to make it Comparable]
  - toString() [to make it easy to print out objects]
  - equals(…) [to make everything work, needed if two different objects could be considered to be the same, like Strings]
  - hashCode() [to make HashSet and HashMap work]

- Part of making user-defined Classes usable with Collections
  - Note: compareTo, equals, and hashCode should be consistent!

public String toString(){…

- If an object has a toString method, then print, printf, and println will automatically call the toString method to get a printable description of the object

- Every object has a toString() method – defined in the Object class.
  - The default toString() just returns the "ID" of the object (not very useful!)

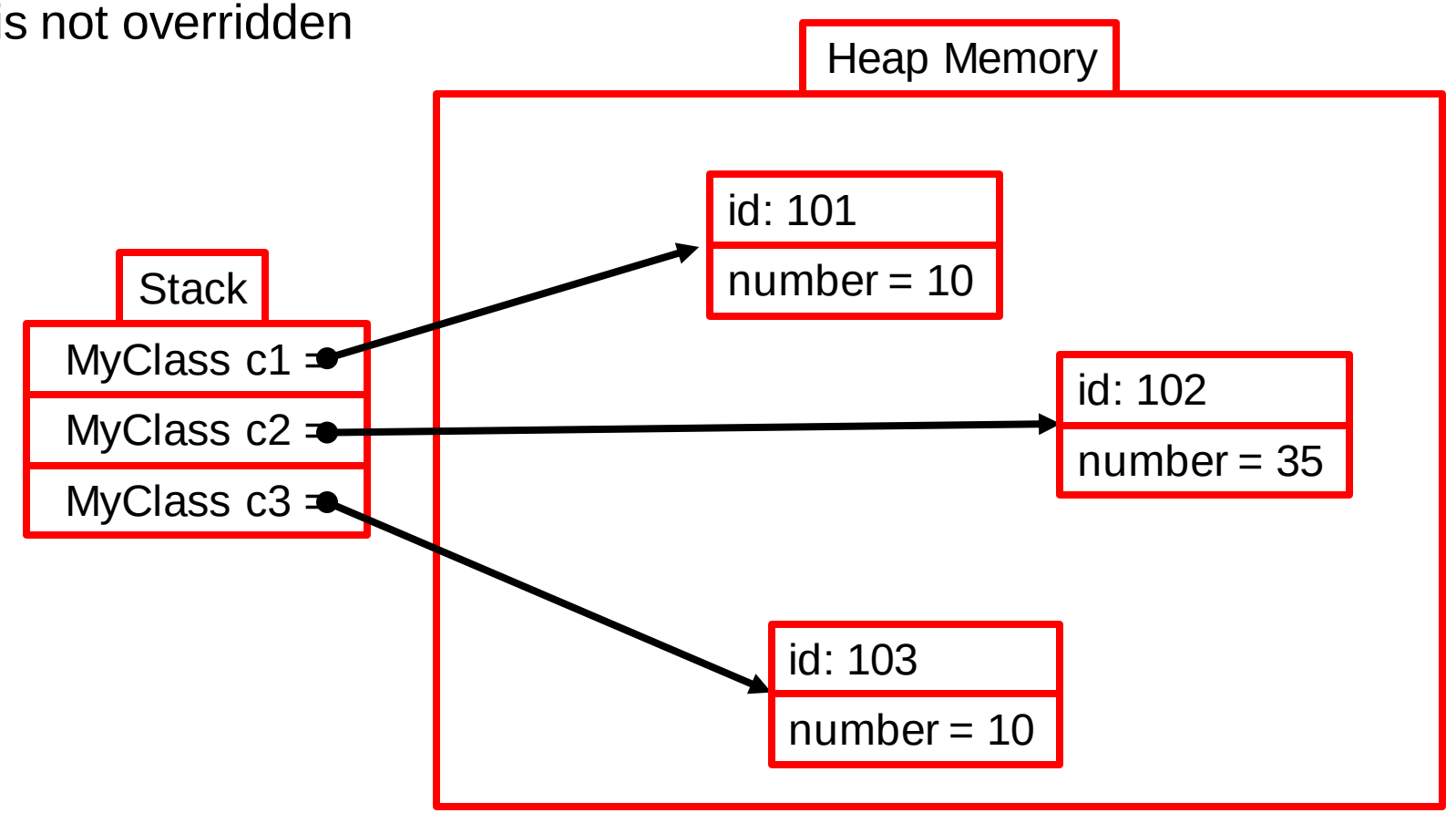
- If a class defines its own (useful) toString() method, then objects of that class can be printed nicely.

- Always define a toString method that gives a useful description, at least for debugging!
  - The String returned should include all the important fields of the object.

- Lists, Sets, *etc* have nice toString() methods

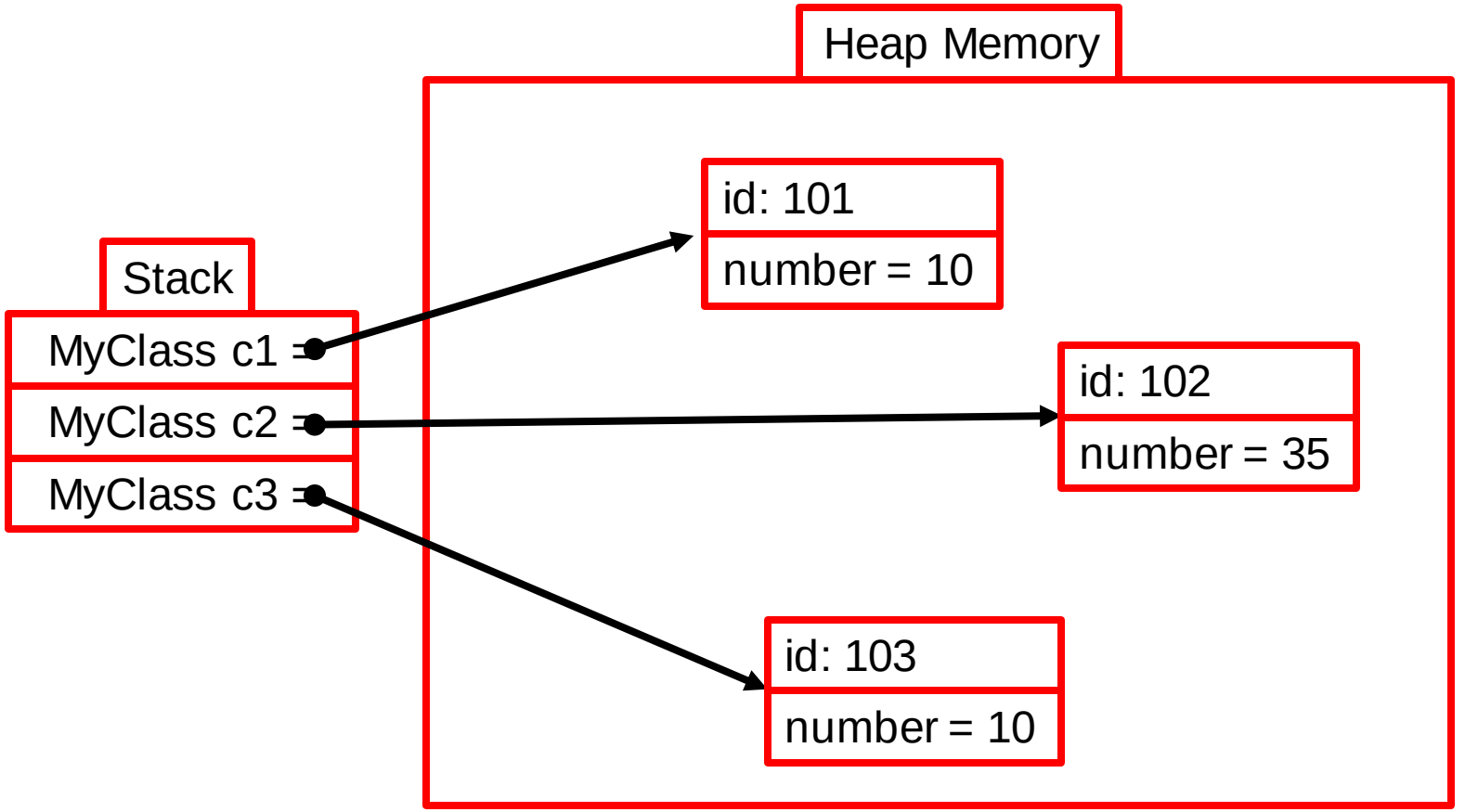
Potential problem with equals(…)

```
public class MyClass {
    private int number;

    //equals is not overridden
}
```



Potential problem with equals(…)

Default equals() checks id of objects. Not the value(s) stored within the object.



public boolean equals(Object obj){…

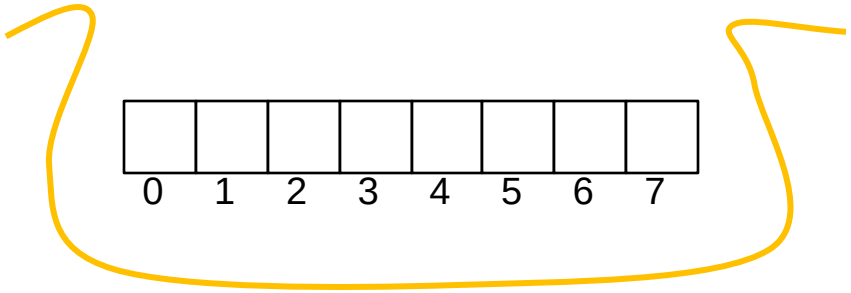
- The Collection classes use the equals(..) method to determine if a value is the same as a value in the collection
- Every object has an equals(…) method – defined in the Object class.
  - The default equals(…) just does ==
    - just compares the references/IDs/pointers to see if the values are exactly the same Object.
    - doesn't look inside the objects to see if the fields are the same
- The String class (and the Collection classes) have useful equals(…)
- Always define an equals method in your classes if two different objects containing the same field values should be considered the same.
  - BUT
    - Need a compatible hashCode() method if using a HashMap
      (equal objects have same hashCode value)
    - Need a compatible compareTo if using a TreeMap
      (equal objects should have compareTo(..) → 0, and vice versa)

Designing an equals(...) method

- The equals method should reflect object identity:
  - If obj1.equals(obj2) then obj1 and obj2 should represent the same entity

- The equals method should not depend on fields that might change
  - Otherwise, objects are changing their identity,
  - two objects that are equal at one time may not be equal later.
  - (Same applies to the compareTo method!)

- The equals method needs to be consistent with the hashCode() method
  - two objects that are equal must have the same hashCode because hashSet and hashMap use both equals() and hashCode() to find values in the collection



hashCode() % data.length to find the bucket

equals(..) to find the item in the bucket.

Fields for equals/compareTo/hashCode:

- What fields should/should not be used for equals, compareTo and hashCode in:
  - a Student class (used in a student records system)
  - a Car class (used in a vehicle registration system)
  - a Shape class (used in a diagram editor)
  - an Earthquake class (used in the GNS earthquake record system)

Designing equals/compareTo/hashCode

The methods should depend

- on fields that are sufficient to identify the entity represented by an object and distinguish it from other entities
- only on fields that will not be changed over time
  - (nice if the fields are declared **final** – so they *can't* be changed.)

If there are no such fields (eg, they might all change over time), or
Every time you create a new object of this type it should be considered unique

Then:

- use the default equals and hashCode which use the object reference/ID/pointer
- compareTo probably doesn't make sense.

Making Classes usable with Collections

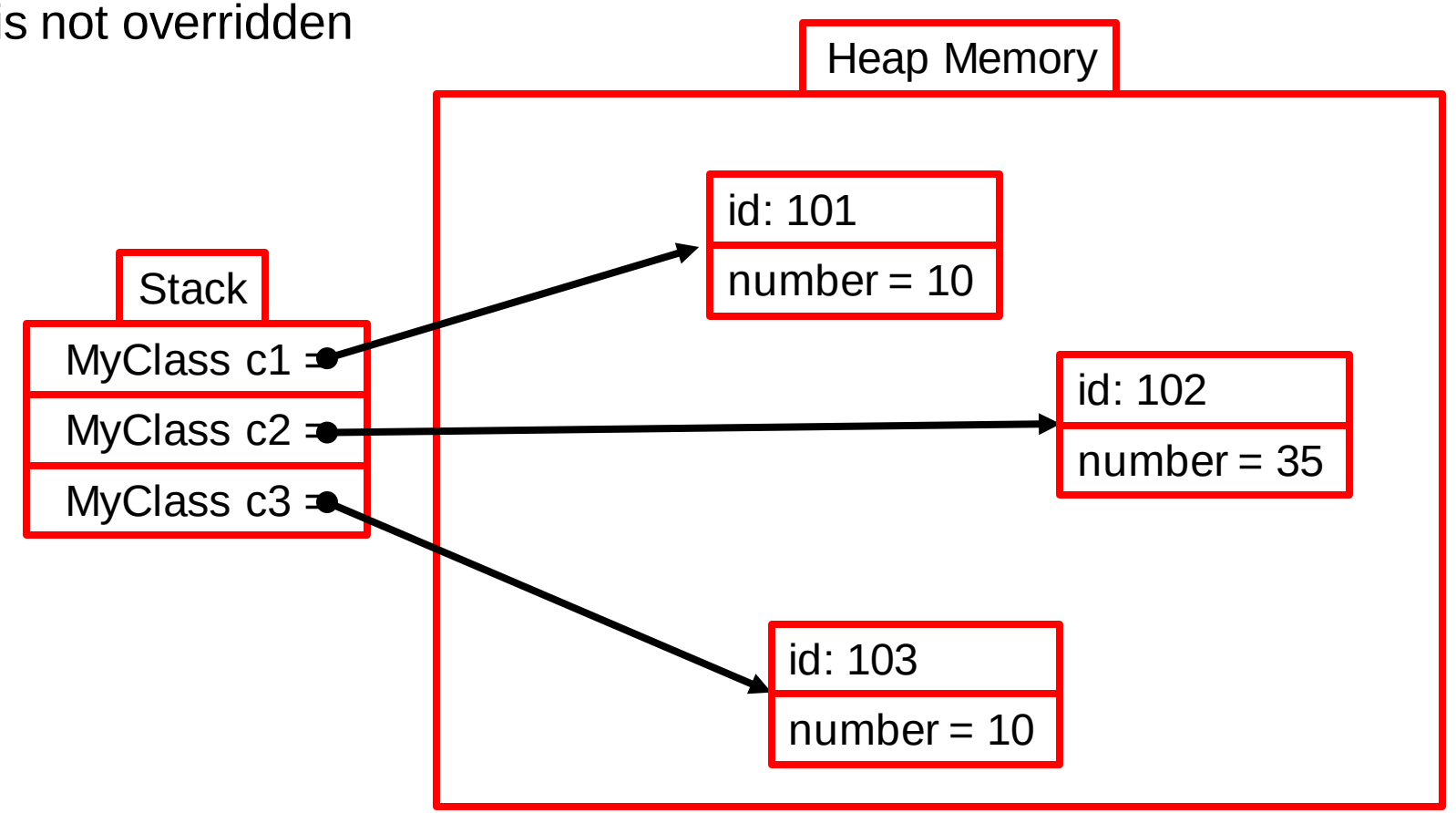
- Making a class Comparable makes it easier to use with sort, TreeSet, TreeMap,…

- There are other methods that may also be needed to make your class easy to use:
  - compareTo(…) [to make it Comparable]
  - toString() [to make it easy to print out objects]
  - equals(…) [to make everything work, needed if two different objects could be considered to be the same, like Strings]
  - hashCode() [to make HashSet and HashMap work]

- Part of making user-defined Classes usable with Collections
  - Note: compareTo, equals, and hashCode should be consistent!

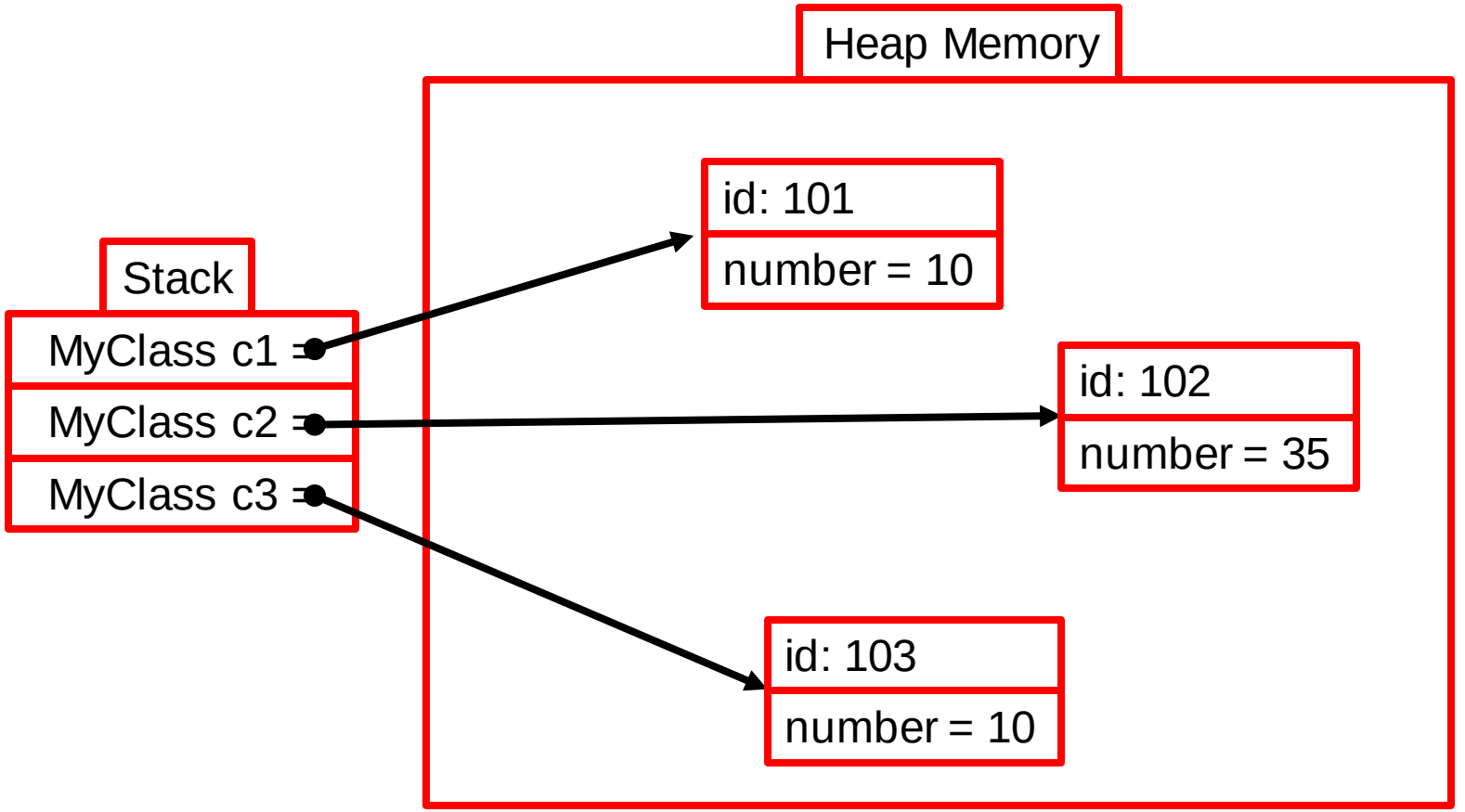
Potential problem with equals(…)

```
public class MyClass {
    private int number;

    //equals is not overridden
}
```



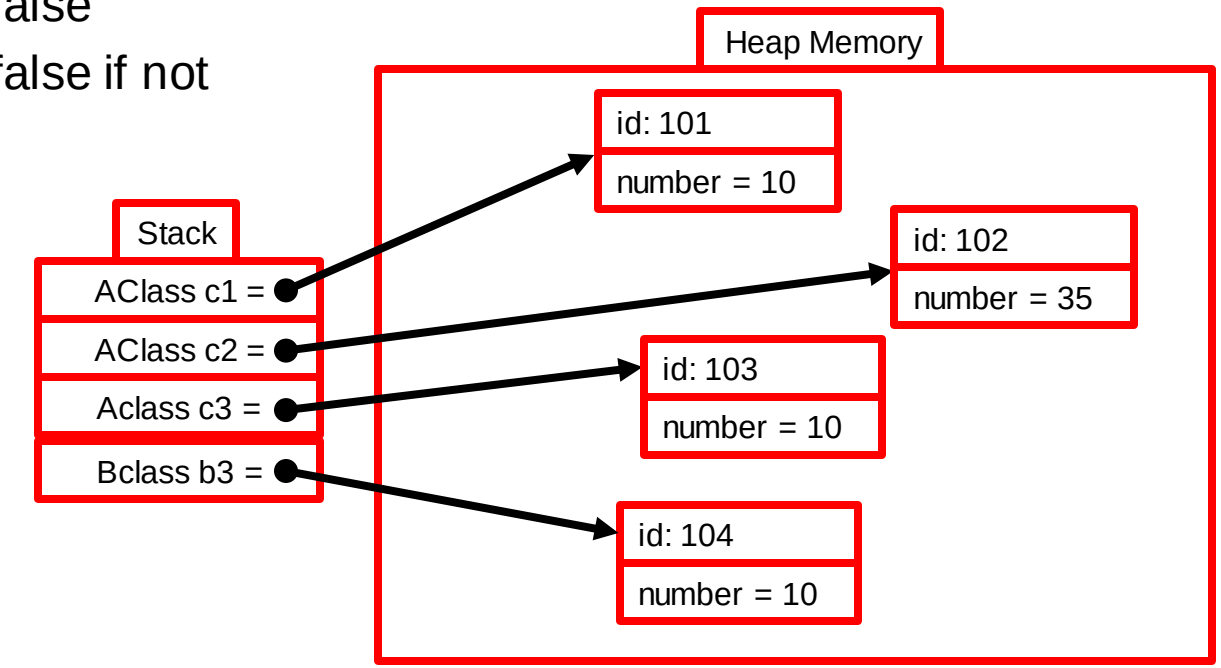
Potential problem with equals(…)

Default equals() checks id of objects. Not the value(s) stored within the object.



public boolean equals(Object obj){…

- Standard way to override the equals method:
  - Check if obj is the same object as this. → true
  - Check if obj is null → false
  - Check if obj is the same type → false if not
  - Cast obj to this type
  - Check if the fields are the same




```
public class AClass{
    .
    .
    public boolean equals(Object obj){
        if (this==obj)                    { return true; }
        if (obj == null)                  { return false; }
        if (! obj instanceof AClass )     { return false; }
        AClass other = (AClass) obj;
        // check if field values are the same
        return  (this.number==other.number && this.n.equals(other.n) && …) ;
    }
}
```