Advanced Data Structures

ISTE-121

Computational Problem Solving In The Information Domain II

Day 11b

Today's activities:

- Sets
- Maps
- Hashcode, Hashtable
- Binary Search Tree

Data Structures overview

- in the preceding classes we encountered two important data structures: array and list
  - keep the elements in the same order in which they are inserted
- Many applications don't care about the order of the elements →can this influence on the data structure performance?
- If the data structure is no longer responsible for remembering the order of element insertion, can it give us better performance for some of its operations?
- In mathematics this is called a **SET**

SET

- Fundamental operations on a set:
  - Adding an element (duplicate is not allowed)
  - Removing an element
  - Locating an element
  - Listing all elements (not necessarily in the order in which they were added)
- How to implement a SET?
  - ArrayList?
  - Linked List?

SET

- There exists to different data structures used for this purpose:
  - Hash table
  - Trees
- The standard Java library provides set implementation based on these structures:
  - **HashSet**
  - **TreeSet**

SET

- **HashSet**
  - Uses „hash table” for storage
  - Used elements must provide a *hashCode* method
- **TreeSet**
  - Uses Binary Search Tree (BTS)
  - The element type should implement the *Comparable* interface

TreeSet<E> & HastSet<E>

TreeSet<E>

- Sets don't allow duplicates
- TreeSet<String> strSet;
- Lists stay sorted, by natural ordering

HashSet<E>

- Sets don't allow duplicates
- HashSet<String> strHash;
- Lists is not sorted in any particular order

Demo: TreeHashSetExp.java

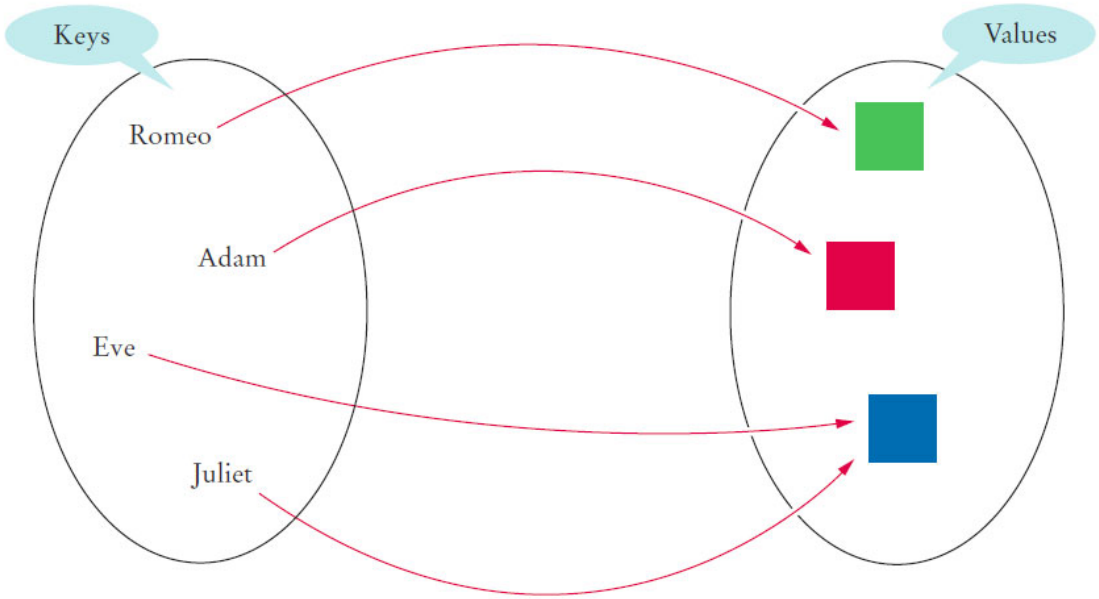
SpellCheck.java

Sets<E> vs. Maps<K,V>

- **Sets** hold one value per entry
  - Example: TreeSet<E>
- **Maps** hold paired (or mapped) values
  - Example: TreeMap<K, V>

Maps<K,V>

- Data type that keeps associations between keys and values




```
Map<String, Color> favoriteColors = new HashMap<String, Color>();
Map<String, Color> favoriteColors = new TreeMap<String, Color>();
Map<String, Color> favoriteColors = new Hashtable<String, Color>();
```

HashMap & Hashtable

- HashMap<K,V> Hashtable<K,V>
  - K = Key
  - V = Value
- Like a dictionary of words and meanings
- If ‘a’ is a key and ‘b’ is value use method **.put(a,b)** to insert / replace values
- Retrieve value by supplying the key **.get(a)**

HashMap<K,V>

- K = Key, can be any object.
  - String, Integer, Employee
- V = Value, can be any value
  - String, ArrayList<String>

HashMap<String, String>

HashMap<String, ArrayList<String>>

Demo: MapDemo.java, WordFrequency.java

For comparison of use

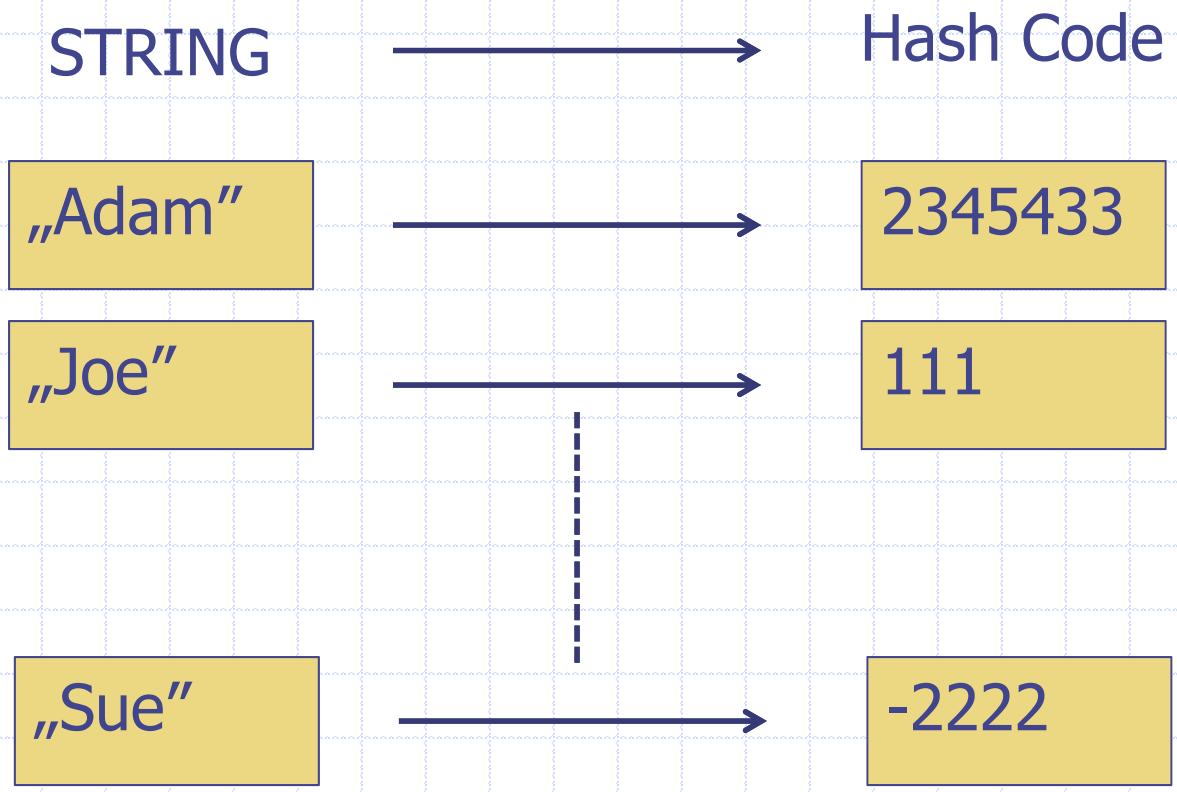
	Values	Duplicates	Ordering	Nulls	Thread Safe
ArrayList	<E>	allowed	None	Allowed	Not safe
HashSet	<E>	Not allowed	None	One allowed	Not safe
Hashtable	<K,V>	Not allowed	None	Not allowed	Thread Safe
HashMap	<K,V>	Not allowed	None	Allowed	Not safe
TreeSet	<E>	Not allowed	Ordering	Not allowed	Not safe
TreeMap	<K,V>	Not allowed	Ordering	Not allowed	Not safe

Hash Tables

- To hash means calculate a value based on data.
- Values are placed at the location – Hash table
- Hash function – a function that computes an integer value, the **hash code,** from an object data → different object, different code.?
- @Override hashCode() method

```
int h = x.hashCode();
```

String and Hash codes #1



#1 – conversion String to Integer ?? Sum of ASCII?

docs.oracle.com

hashCode

```
public int hashCode()
```

Returns a hash code for this string. The hash code for a `String` object is computed as

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

using `int` arithmetic, where `s[i]` is the *i*th character of the string, `n` is the length of the string, and `^` indicates exponentiation. (The

Overrides:

`hashCode` in class `Object`

Returns:

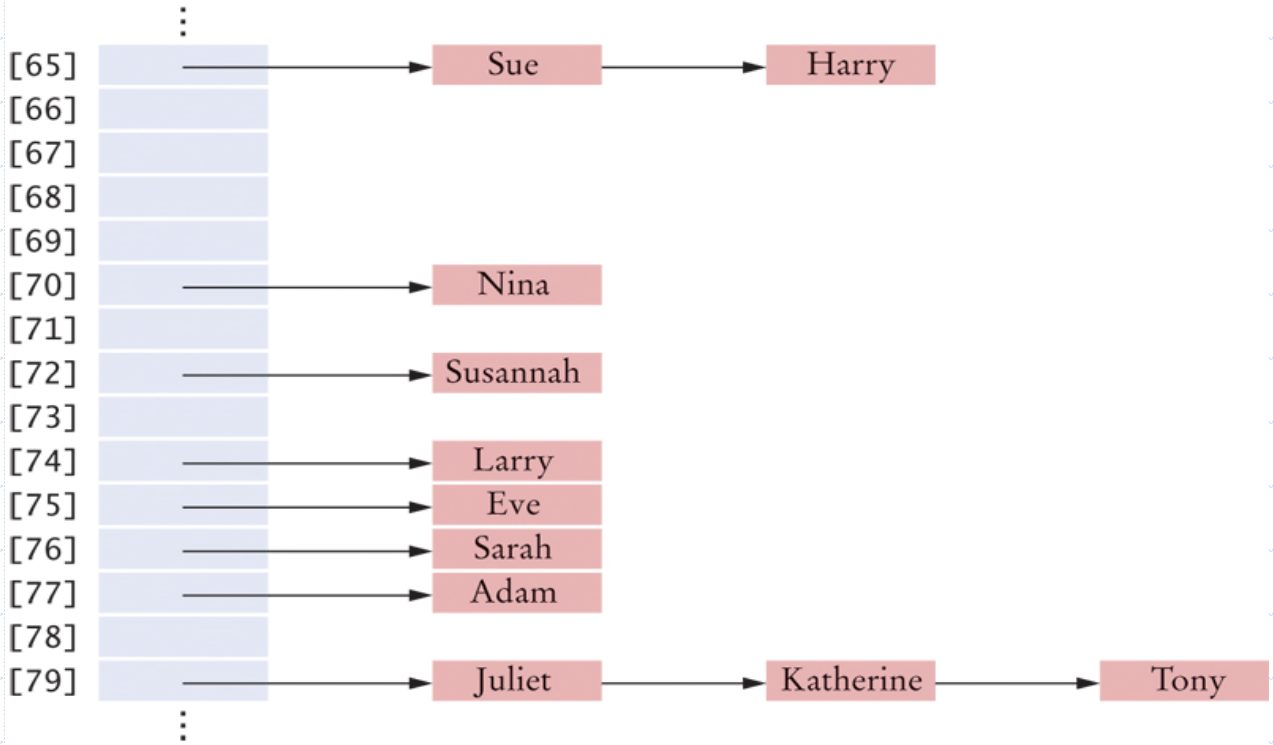
a hash code value for this object.

See Also:

`Object.equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

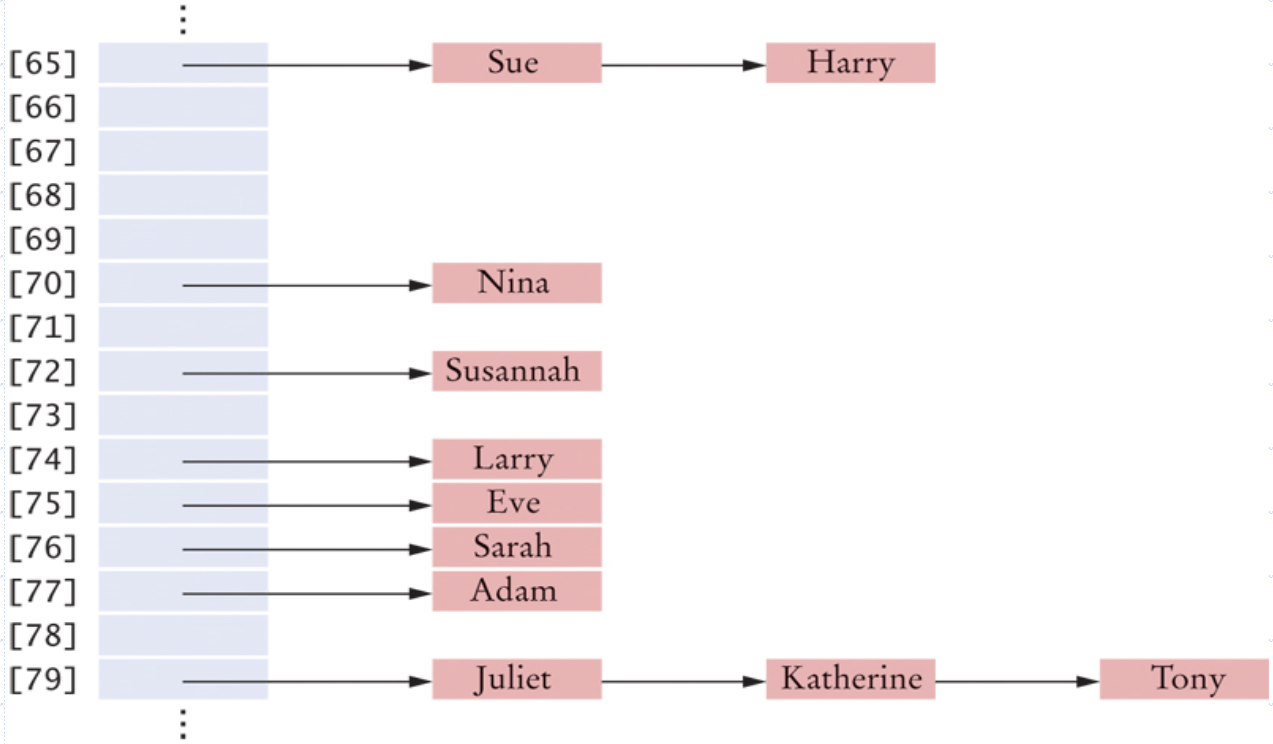
String and Hash codes #2

- Create a table, insert String into a bucket at index of the HashCode value
- limited table size (%n) → possible collision



Here is the algorithm for finding an object x in a hash table

- Compute the hash code and reduce it modulo the table size. This gives an index h into the hash table.
- Iterate through the elements of the bucket at position h. For each element of the bucket, check whether it is equal to x.
- If a match is found among the elements of that bucket, then x is in the set. Otherwise, it is not.



Binary search tree

- https://www.youtube.com/watch?v=qYo8BVxtoH4