

# Topic 20

## Data Structure Potpourri: Hash Tables and Maps

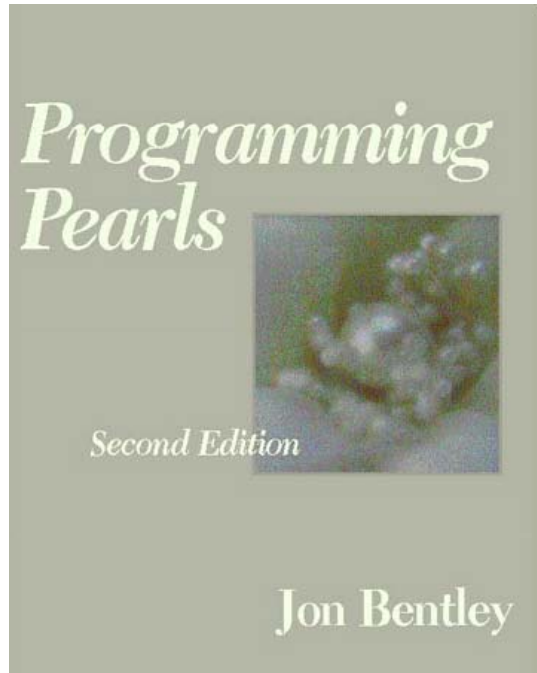
### "hash collision" n.

[from the techspeak] (var. `hash clash') When used of people, signifies a confusion in associative memory or imagination, especially a persistent one (see thinko). True story: One of us was once on the phone with a friend about to move out to Berkeley. When asked what he expected Berkeley to be like, the friend replied: "Well, I have this mental picture of naked women throwing Molotov cocktails, but I think that's just a collision in my hash tables."

## -The Hacker's Dictionary



# Programming Pearls by Jon Bentley



- ▶ Jon was *senior programmer* on a large programming project.
- ▶ Senior programmer spend a lot of time helping junior programmers.
- ▶ Junior programmer to Jon: "I need help writing a sorting algorithm."

# A Problem

## ► From *Programming Pearls* (Jon in Italics)

*Why do you want to write your own sort at all? Why not use a sort provided by your system?*

I need the sort in the middle of a large system, and for obscure technical reasons, I can't use the system file-sorting program.

*What exactly are you sorting? How many records are in the file?*

*What is the format of each record?*

The file contains at most ten million records; each record is a seven-digit integer.

*Wait a minute. If the file is that small, why bother going to disk at all? Why not just sort it in main memory?*

Although the machine has many megabytes of main memory, this function is part of a big system. I expect that I'll have only about a megabyte free at that point.

*Is there anything else you can tell me about the records?*

Each one is a seven-digit positive integer with no other associated data, and no integer can appear more than once.

# Questions

- ▶ When did this conversation take place?
- ▶ What were they sorting?
- ▶ How do you sort data when it won't all fit into main memory?
- ▶ Speed of file i/o?



# A Solution

```
/* phase 1: initialize set to empty */  
for i = [0, n)  
    bit[i] = 0
```

```
/* phase 2: insert present elements into the set */  
for each i in the input file  
    bit[i] = 1
```

```
/* phase 3: write sorted output */  
for i = [0, n)  
    if bit[i] == 1 write i on the output file
```

# Some Structures so Far

## ▶ ArrayLists

- $O(1)$  access
- $O(N)$  insertion (average case), better at end
- $O(N)$  deletion (average case)

## ▶ LinkedLists

- $O(N)$  access
- $O(N)$  insertion (average case), better at front and back
- $O(N)$  deletion (average case), better at front and back

## ▶ Binary Search Trees

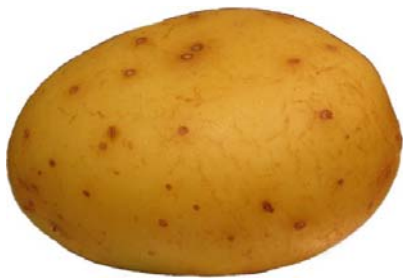
- $O(\log N)$  access if balanced
- $O(\log N)$  insertion if balanced
- $O(\log N)$  deletion if balanced

# Why are Binary Trees Better?

- ▶ Divide and Conquer
  - reducing work by a factor of 2 each time
- ▶ Can we reduce the work by a bigger factor?  
10? 1000?
- ▶ An ArrayList does this in a way when *accessing* elements
  - *but must use an integer value*
  - *each position holds a single element*

# Hash Tables

- ▶ Hash Tables overcome the problems of ArrayList while maintaining the fast access, insertion, and deletion in terms of  $N$  (number of elements already in the structure.)





# Hash Functions

- ▶ Hash: "From the French hacher, which means 'to chop'."
- ▶ *to hash* to mix randomly or shuffle (To cut up, to slash or hack about; to mangle)
- ▶ Hash Function: Take a large piece of data and reduce it to a smaller piece of data, usually a single integer.
  - A function or algorithm
  - The input need not be integers!

# Hash Function



5/5/1967

555389085

5122466556

"Mike Scott"

scottm@gmail.net

"Isabelle"

→ 12  
hash  
function

# Simple Example

- ▶ Assume we are using names as our *key*
  - *take 3rd letter of name, take int value of letter (a = 0, b = 1, ...), divide by 6 and take remainder*
- ▶ What does "Bellers" hash to?
- ▶ L -> 11 ->  $11 \% 6 = 5$

# Result of Hash Function

- ▶ Mike =  $(10 \% 6) = 4$
- ▶ Kelly =  $(11 \% 6) = 5$
- ▶ Olivia =  $(8 \% 6) = 2$
- ▶ Isabelle =  $(0 \% 6) = 0$
- ▶ David =  $(21 \% 6) = 3$
- ▶ Margaret =  $(17 \% 6) = 5$  (uh oh)
- ▶ Wendy =  $(13 \% 6) = 1$
- ▶ This is an imperfect hash function. A perfect hash function yields a one to one mapping from the keys to the hash values.
- ▶ What is the maximum number of values this function can hash perfectly?

# More on Hash Functions

- ▶ Normally a two step process
  - transform the key (which may not be an integer) into an integer value
  - Map the resulting integer into a valid index for the hash table (where all the elements are stored)
- ▶ The transformation can use one of four techniques
  - mapping, folding, shifting, casting

# Hashing Techniques

## ► Mapping

- As seen in the example
- integer values or things that can be easily converted to integer values in key

## ► Folding

- partition key into several parts and the integer values for the various parts are combined
- the parts may be hashed first
- combine using addition, multiplication, shifting, logical exclusive OR

# More Techniques

## ► Shifting

- an alternative to folding
- A fold function

```
int hashVal = 0;
int i = str.length() - 1;
while(i > 0){
    hashVal += (int) str.charAt(i);
    i--;
}
```

results for "dog" and "god" ?

# Shifting and Casting

## ► More complicated with shifting

```
int hashVal = 0;
int i = str.length() - 1;
while(i > 0)
{ hashVal = (hashVal << 1) + (int) str.charAt(i);
  i--;
}
```

different answers for "dog" and "god"

Shifting may give a better range of hash values  
when compared to just folding

## Casts

### ► Very simple

- essentially casting as part of fold and shift when working with chars.



# The Java String class hashCode method

```
public int hashCode()  
{  
    int h = hash;  
    if (h == 0)  
    {  
        int off = offset;  
        char val[] = value;  
        int len = count;  
        for (int i = 0; i < len; i++)  
        {  
            h = 31*h + val[off++];  
        }  
        hash = h;  
    }  
    return h;  
}
```



# Mapping Results

- ▶ Transform hashed key value into a legal index in the hash table
- ▶ Hash table is normally uses an array as its underlying storage container
- ▶ Normally get location on table by taking result of hash function, dividing by size of table, and taking remainder

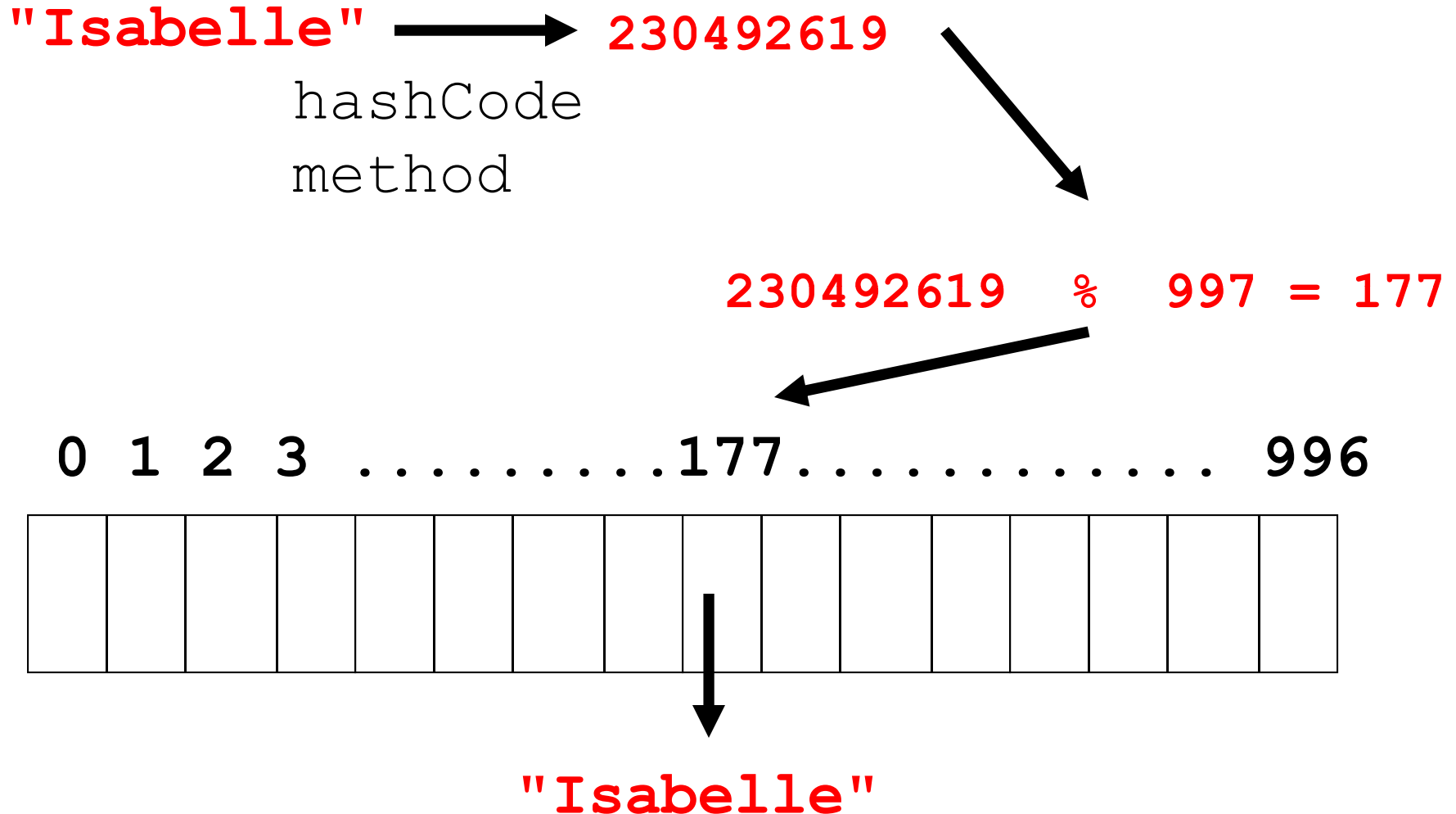
$\text{index} = \text{key} \bmod n$

$n$  is size of hash table

empirical evidence shows a prime number is best

1000 element hash table, make 997 or 1009 elements

# Mapping Results



# Handling Collisions

- ▶ What to do when inserting an element and already something present?



# Open Address Hashing

- ▶ Could search forward or backwards for an open space
- ▶ Linear probing:
  - move forward 1 spot. Open?, 2 spots, 3 spots
  - reach the end?
  - When removing, insert a blank
  - null if never occupied, blank if once occupied
- ▶ Quadratic probing
  - 1 spot, 2 spots, 4 spots, 8 spots, 16 spots
- ▶ Resize when *load factor* reaches some limit



# Chaining

- ▶ Each element of hash table be another data structure
  - linked list, balanced binary tree
  - More space, but somewhat easier
  - everything goes in its spot
- ▶ Resize at given load factor or when any chain reaches some limit: (relatively small number of items)
- ▶ What happens when resizing?
  - Why don't things just collide again?



# Hash Tables in Java

- ▶ `hashCode` method in `Object`
- ▶ `hashCode` and `equals`
  - "If two objects are equal according to the `equals` (`Object`) method, then calling the `hashCode` method on each of the two objects must produce the same integer result. "
  - if you override `equals` you need to override `hashCode`

# Hash Tables in Java

- ▶ `HashTable` class
- ▶ `HashSet` class
  - implements `Set` interface with internal storage container that is a `HashTable`
  - compare to `TreeSet` class, internal storage container is a Red Black Tree
- ▶ `HashMap` class
  - implements the `Map` interface, internal storage container for keys is a hash table



# Maps (a.k.a. Dictionaries)

A → 65

Dictionary

**gouge** |gouj|  
noun

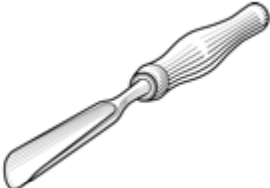
1 a chisel with a concave blade, used in carpentry, sculpture, and surgery.  
2 an indentation or groove made by gouging.

verb [ trans. ]

1 make (a groove, hole, or indentation) with or as if with a gouge : *the channel had been **gouged out** by the ebbing water.*

- make a rough hole or indentation in (a surface), esp. so as to mar or disfigure it : *he had wielded the blade inexpertly, gouging the grass in several places.*
- ( **gouge something out** ) cut or force something out roughly or brutally : *one of his eyes had been gouged out.*

2 informal overcharge; swindle : *the airline ends up gouging the very passengers it is supposed to assist.*



gouge 1

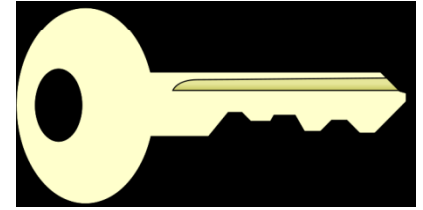
# Maps

- ▶ Also known as:
  - table, search table, dictionary, associative array, or associative container
- ▶ A data structure optimized for a very specific kind of search / access
  - with a *bag* we access by asking "is X present"
  - with a *list* we access by asking "give me item number X"
  - with a *queue* we access by asking "give me the item that has been in the collection the longest."
- ▶ In a *map* we access by asking "give me the *value* associated with this *key*."

# Keys and Values

- ▶ Dictionary Analogy:

- The *key* in a dictionary is a word:  
*foo*



- The *value* in a dictionary is the definition:  
*First on the standard list of metasyntactic variables used in syntax examples*

- ▶ A key and its associated value form a pair that is stored in a map
- ▶ To retrieve a value the key for that value must be supplied
  - A List can be viewed as a Map with integer keys

# More on Keys and Values

- Keys must be unique, meaning a given key can only represent one value
    - but one value may be represented by multiple keys
    - like synonyms in the dictionary.
- Example:
- factor: n. See coefficient of X*
- *factor* is a key associated with the same value (definition) as the key *coefficient of X*

# The Map<K, V> Interface in Java

- ▶ `void clear()`
  - Removes all mappings from this map (optional operation).
- ▶ `boolean containsKey(Object key)`
  - Returns true if this map contains a mapping for the specified key.
- ▶ `boolean containsValue(Object value)`
  - Returns true if this map maps one or more keys to the specified value.
- ▶ `Set<K> keySet()`
  - Returns a Set view of the keys contained in this map.

# The Map Interface Continued

- ▶ `V get(Object key)`
  - Returns the value to which this map maps the specified key.
- ▶ `boolean isEmpty()`
  - Returns true if this map contains no key-value mappings.
- ▶ `V put(K key, V value)`
  - Associates the specified value with the specified key in this map

# The Map Interface Continued

- ▶ `V remove(Object key)`
  - Removes the mapping for this key from this map if it is present
- ▶ `int size()`
  - Returns the number of key-value mappings in this map.
- ▶ `Collection<V> values()`
  - Returns a collection view of the values contained in this map.

# Implementing a Map

- ▶ Two common implementations of maps are to use a binary search tree or a hash table as the internal storage container
  - HashMap and TreeMap are two of the implementations of the Map interface
- ▶ `HashMap` uses a hash table as its internal storage container.
  - keys stored based on hash codes and size of hash tables internal array



# TreeMap implementation

- ▶ Uses a Red - Black tree to implement a Map
- ▶ relies on the `compareTo` method of the keys
- ▶ somewhat slower than the HashMap
- ▶ keys stored in sorted order

# Sample Map Problem

Determine the frequency of words in a file.

```
File f = new File(fileName);
Scanner s = new Scanner(f);
Map<String, Integer> counts =
    new Map<String, Integer>();
while( s.hasNext() ){
    String word = s.next();
    if( !counts.containsKey( word ) )
        counts.put( word, 1 );
    else
        counts.put( word,
            counts.get(word) + 1 );
}
```