# Hash tables

- hashing -- idea
- collision resolution
  - closed addressing (chaining)
  - open addressing techniques
- hash function
- Java **`hashCode()`** for **`HashMap`** and **`HashSet`**
- big-O time bounds
- applications

# Announcements

- Change to Claire's Thurs office hours this Thur 11/2: 12 – 2pm
- Midterm 2
  - Tue. 11/7, THH (room assignments coming soon)
  - closed book, closed note, bring USC ID card
- No lab assignment or lab meetings this week
- Start early on PA4: more design involved (all based on material we have covered already)

# Review: Map ADT

- A map stores a collection of (key,value) pairs
- keys are unique: a pair can be identified by its key

Operations:

- add a new (key, value) pair (called an entry)
- lookup an entry, given its key
- update the value of an entry, given its key
- remove an entry, given its key
- list all the entries
  - (order of visiting depends on the kind of map created)

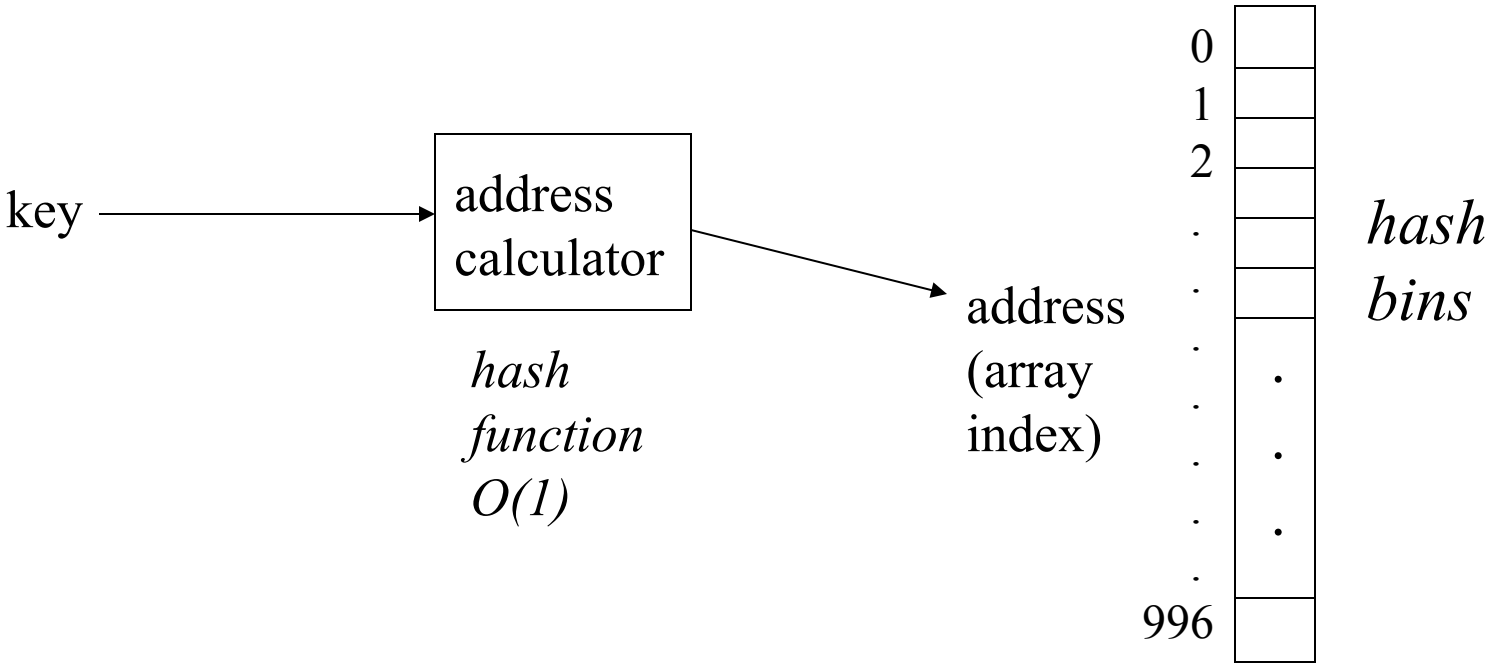
# Hashing idea

- Suppose we had a map where keys were all in the range [0..99].
- What specialized representation could we use?

- Hashing is a generalization of this idea . . .

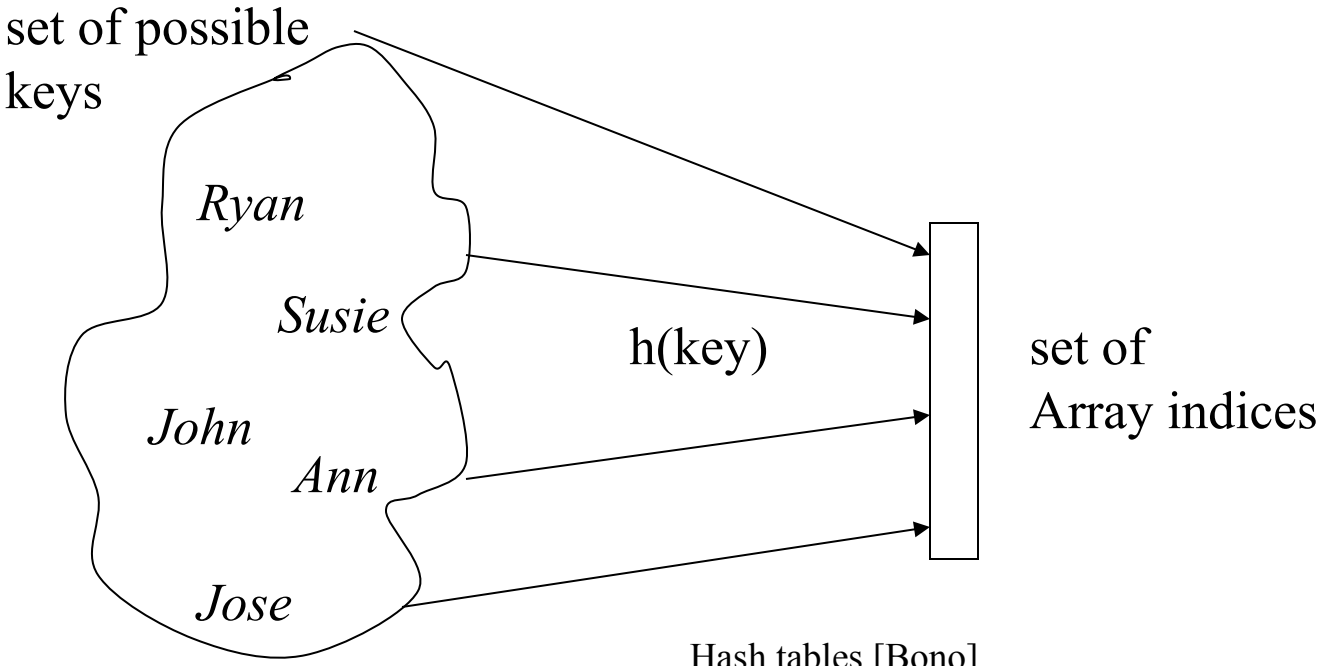
# Hashing idea (cont.)

- In Map ADT, keys might be strings, ints, …



# Problems

- More key values than indices in the array.
- How big is the set of one-word names of up to 20 characters?



# Collisions

- Even if set of actual key values is small, the set of possible key values is large.
- The hash function maps the set of *possible* values, so you need to worry about collisions.

- But if the set of actual values is small, how soon do we need to really worry about this?

# The Birthday Problem

How many people in a room before chance that any two have the same birthday is > 50%?

# What's the implication?

- In general…
  - No matter what the table size
  - No matter what the hash function
  - No matter how many actual entries currently stored (>1)
- …the hash address for key does ***not*** uniquely identify the map entry <key, value>.
  - i.e., collisions can happen
- hash address just identifies a hash bin
  - assume you can store multiple entries in a bin

# Collision resolution

- Recall: hash value does not uniquely identify the entry.

- Collision resolution:

  Where to put the entries when two of them map to the same location (i.e., $\text{hash}(\text{key}_1) = \text{hash}(\text{key}_2)$)?

# Collision resolution strategies

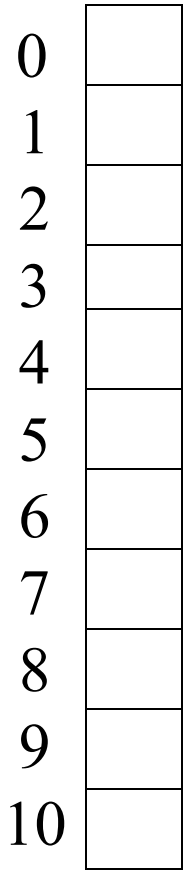
- Where to put the entries when two of them map to the same location?
  - Open addressing – put them in different hash buckets
    - linear probing: put in next empty bucket
  - Closed addressing – each bucket can hold multiple entries.
    - chaining: use a linked list for all entries in a bucket.
- We'll only discuss chaining in detail.

# Collision resolution by chaining

- Each bucket stores multiple items -- using linked list.

John --> 3
Sam --> 6
Mary --> 3
Beth --> 4
Joe --> 4
Bob --> 3

**`HSIZE = 11`**




- Insert, lookup, delete

# Hash functions

- Required properties:
  - deterministic: value only depends on key
  - has to map to the indices of the array:
    - achieve with : **`hashValue % HASH_SIZE`**

- Desirable properties
  - easy to compute (i.e., fast)
  - promotes uniform distribution of key values over whole range of addresses

# Hash functions

- General methods
  - truncation (e.g., key is a SSN)
  - folding
  - modulus (%)

    `% HASHSIZE` (prime)

# Hash functions: example

- Simple example of folding technique:
  - for a key that is a string: $c_1..c_n$

    $(\Sigma c_i) \% HASHSIZE$
- How many distinct values (addresses) can we get with this hash function? (before modulus)

# Better string hash function

- Takes into account the position of the character.

  $\alpha$ is a constant (prime better)

  $h_0 = 0$

  $h_i = \alpha\, h_{i-1} + c_i \qquad 1 <= i <= k$
- Implement with a loop
- but to see what's happening, here's the whole computation
  - for three chars:

    $\alpha^2 c_1 + \alpha c_2 + c_3$
- To find good hash functions:

  The Art of Computer Programming, volume on Sorting and Searching, by Don Knuth

# Using Java **`HashMap`** (or **`HashSet`**)

- Recall: for **`HashMap<KeyType, ValueType>`**
  - **`KeyType`** must have
    **`equals`** and **`hashCode`** defined
  - Already defined for Java types such as **`String`**, **`Integer`**, **`Double`**

- How to use our own type as the **`KeyType`**?

# Java `hashCode()` method

- Java **`HashMap`** / **`HashSet`** uses **`hashCode()`** method as the hash function.
- Many Java classes override it to do the right thing.
- For example **`String`**'s **`hashCode()`** uses the algorithm from the previous slide ($\alpha = 31$)
- Also defined for **`LinkedList`** and **`ArrayList`**, for example (see documentation for details)

# Writing your own `hashCode()`

- Can be overridden from **`Object`**
- Doesn't depend on size of hash table:
  - **`%`** **`HASH_SIZE`** is handled by HashMap/HashSet class
- Contract for **`hashCode()`**
  - if a.equals(b) then a.hashCode() == b.hashCode()
- Warning: Object version does not do the right thing – don't rely on inherited one.
  - object version returns a code based on the *address* of the object (i.e., object identity)
  - We want it to use the *value* of the object

# Ex: Student class

```
public class Student {
  private String theName;
  private int score;
  public boolean equals(Object other) {
   // returns true iff they have the same name
   // and the same score
  }
  public int hashcode() {
    return theName.hashCode() + score;
  }
  . . .
}
```

# Danger of inheriting `hashCode()`

- Consider a **`Term`** class: (coefficient, exponent).
- Suppose we *override* **`equals()`** to be that the exponents and coefficients must be the same.
- but *inherit* **`hashCode()`** from **`Object`**.
- Reminder: Contract for hashCode()
  - if a.equals(b) then a.hashCode() == b.hashCode()

```
Term t = new Term(3,4);
Term t2 = new Term(3,4);
```

- `t.equals(t2)` // true or false?
- `t.hashCode() == t2.hashCode()` // true or false?

# Why does this matter?

- What happens if you use these terms in a HashSet?

```
Term t = new Term(3,4);
. . .
hashSet.add(t);
. . .
Term target = new Term(3,4);
if (hashSet.contains(target)) . . .
```

- `t.equals(target)` – true or false?
- `t.hashCode() == target.hashCode()` – true or false?

# Converse to contract not true

- Reminder: Contract for **`hashCode()`**
  - if **`a.equals(b)`** then **`a.hashCode() == b.hashCode()`**

- But, if **`a.hashCode() == b.hashCode()`**
  - what do we know about **`a`** and **`b`**?

# Big-O time bounds

- Suppose chaining

  n = number of items in table
  b = size of table
  $\lambda = n / b$  average length of a list (uniform distr)
- $O(\lambda)$ search time
- On average, O(1)
- Inserts and deletes also take $O(\lambda)$

# Performance in practice

- Performance depends on hash function, and load factor.
- Good hash function:
  - uniform distribution of keys over hash addresses
- Bad hash function:
  - worst case, all values could be in one bucket.
- Load factor:
  - HashMap / HashSet make the array bigger if it gets above 0.75 load factor. (numEntries / HASH_SIZE)
  - Involves rehashing everything.

# Big-O for traversal

- Traversal (in no special order)
  takes O(*n*) -- but really *n* + size of the array
- Ordered traversal involves sorting.
  - Best sorts are O(nlogn)
- hashing is an excellent Map representation,
  providing that you aren't going to be
  doing traverseInOrder operation much.
  - What representations are well suited for ordered traversal?

# Applications of hash tables

- First: properties desired:
  - lookup
  - insert
  - (remove)
  - don’t care about order of keys
- Examples of applications:
  - Database (master customer file)
  - Compilers: symbol tables
  - Games: look up board configuration to find the move that goes with it (e.g., chess, tic-tac-toe)
  - UNIX shell: quick command lookup.