CSSE 230 Day 18

Hash table basics

Exam 2

- Format same as Exam 1
  - One 8.5x11 sheet of paper (2-sided) for written part
  - Same resources as before for programming part
- Topics: weeks 1-8
  - Reading, programs, in-class, written assignments.
  - Especially
    - OO programming, using various data structures (lists, stacks, queues, sets, maps, priority queues)
    - Binary trees, including AVL, rank, and threaded trees
      - Traversals and iterators, numeric properties
    - Exhaustive search and the Queens problem
    - Mathematical induction
    - Algorithm analysis in general

T
F
IDK

Sample exam on ANGEL may be good practice. **Best practice:** written assignments.

Reminders/Announcements

- See schedule page
- Short quiz over AVL insertion

Questions

Agenda

- Hash table basics
- Collision resolution
- EditorTrees work time

Hashing

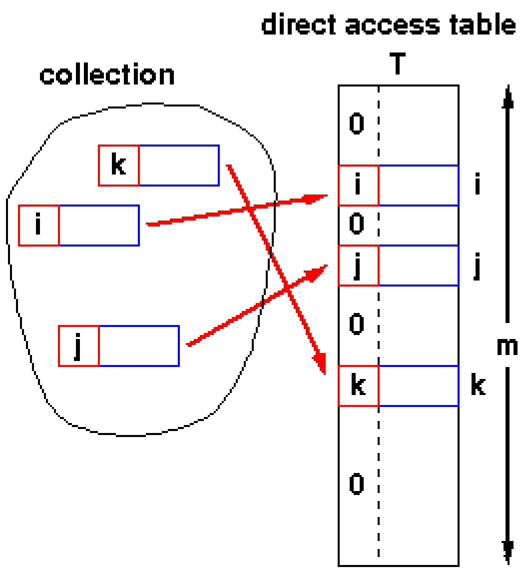
- Efficiently putting 5 pounds of data in a 20 pound bag

A hash table is a very fast approach to dictionary storage

- Provides rapid insertion, retrieval, and deletion of items by *key*
- **HashMap** uses a hash table internally
  - Actual table data is stored in an array
  - **HashSet** uses a **HashMap** internally
- Insertion and lookup are constant time!
  - With a good "hash function"
  - And large enough storage array

On average

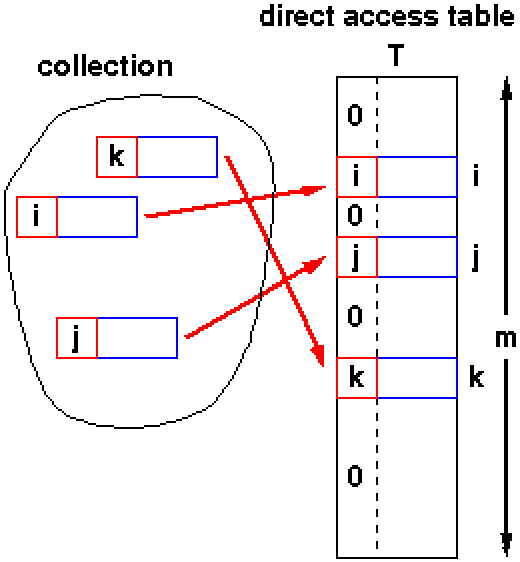
Intro: Direct Address Tables



Contents of this slide are from John Morris, University of Western Australia

- If we have a collection of **n** elements whose keys are unique integers in the range 0 .. **m–1**, where **m** >= **n**,
- then we can store the items in a *direct address* table, **T[m]**,
  - where T_i is either empty or contains one of the elements of our collection
  - Searching a direct address table is clearly an **O(1)** operation:
    - for a key, **k**, we access T_k,
    - if it contains an element, return it,
    - if it doesn't, then return a NULL

Intro: Direct Address Tables



- There are two major constraints:
  1. the keys must be unique
  2. the range of possible keys must be severely bounded

The second constraint is usually impossible to meet

Contents of this slide are from John Morris, University of Western Austrailia

We attempt to create unique keys by applying a hashCode(key) function …

key → hashCode() → integer

A good hashCode() distributes the keys, like:

hashCode("ate")= 48594983
hashCode("ape")= 76849201
hashCode("awe") = 14893202

…and then take it mod the table size (m) to get an index into the array.

- Example: if m = 100:

hashCode(“ate”)= 48594983
hashCode(“ape”)= 76849201
hashCode(“awe”) = 1489036

mod

→83
→01
→36

Index calculated from the object itself, not from a comparison with other objects

- Every Java object has a **hashCode** method that returns an integer H
  - It uses H % m as the index into the array

"ate" → hashCode() → 48594983 → mod → 83

…	
82	
83	ate
84	
…	

  - Unless this position is already occupied

a "collision"

Object implements a default *hashCode* method

- Should we inherit it?
- JDK classes override the **hashCode()** method
- If you plan to use instances of your class as keys in a hash table, you probably should too!

hashCode method

- Should be fast to compute
- Should distribute keys as evenly as possible
- These two goals are often contradictory; we need to achieve a balance

A simple hash function for strings is a function of every character

```
// This could be in the String class
public static int hash(String s) {
 int total = 0;
 for (int i=0; i<s.length(); i++)
   total = total + s.charAt(i);
 return Math.abs(total);
}
```

- Advantages?
- Disadvantages?

A better hash function for Strings uses place value with a base that's prime

```
// This could be in the String class
public static int hash(String s) {
 int total = 0;
 for (int i=0; i<s.length(); i++)
   total = total*23 + s.charAt(i);
 return Math.abs(total);
}
```

- Spreads out the values more, and anagrams not an issue.
- We can't entirely avoid collisions. Why?
- What about overflow during computation?
- Note: **String** already has a reasonable **hashCode()** method; we don't have to write it ourselves.

Hash Table Caveats

- Objects that are equal (based on the *equals* method) **MUST** have the same *hashCode* values
- Different objects should have different *hashCodes* if possible
- Beware of mutable objects!
- Hash tables don't maintain sorted order
  - So what's cost to find min or max element?

Collisions are Inevitable

- A hash table implementation (like *HashMap*) provides a "collision resolution mechanism"
- There are a variety of approaches to this
- Fewer collisions lead to faster performance

Collision Avoidance

- Just make hashCode unique?

- Possible key values >> capacity of table
  - Example: A key may be an array of 16 characters
  - How many different values could there be?

- Table size << possible hashCode values

- Object hashCodes are taken **mod** the current table size

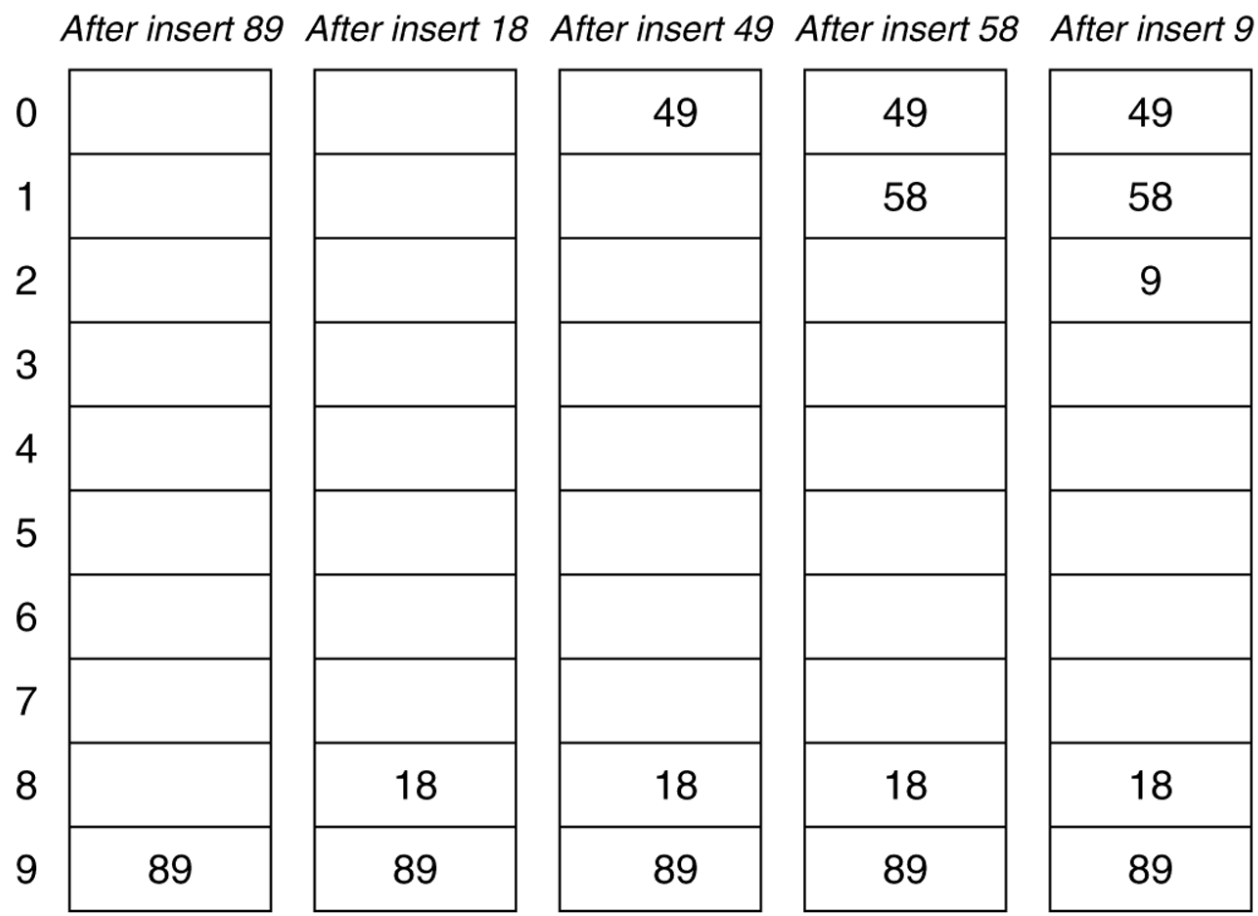
Collision Resolution: Linear Probing

- Collision? Use the next available space:
  - Try H+1, H+2, H+3, …
  - Wraparound at the end of the array
- Problem: Clustering
- Animation:
  - http://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html

Figure 20.4

Linear probing hash table after each insertion

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```



Linear Probing Efficiency

- Depends on Load Factor, λ:
  - Ratio of the number of items stored to table size
  - $0 \leq \lambda \leq 1$.

- For a given λ, what is the expected number of probes before an empty location is found?

Rough Analysis of Linear Probing

- For a given λ, what is the expected number of probes before an empty location is found?
- Assume all locations are equally likely to be occupied, and equally likely to be the next one we look at.
- Then the probability that a given cell is full is λ and probability that a given cell is empty is $1-\lambda$.
- What's the expected number?

$$\sum_{p=1}^{\infty} \lambda^{p-1}(1-\lambda)p = \frac{1}{1-\lambda}$$

Better Analysis of Linear Probing

- "Equally likely" probability is not realistic
- Clustering!
  - Blocks of occupied cells are formed
  - Any collision in a block makes the block bigger
- Two sources of collisions:
  - Identical hash values
  - Hash values that hit a cluster
- Actual average number of probes for large λ:

$$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

For a proof, see Knuth, The Art of Computer Programming, Vol 3: Searching Sorting, 2nd ed, Addision-Wesley, Reading, MA, 1998.

Why consider linear probing?

- Easy to implement
- Simple code has fast run time per probe
- Works well when load is low
  - It could be more efficient just to rehash using a bigger table once it starts to fill.
  - Typically done in practice: rehash to an array that is double in size once the load goes over 0.5
- What about other fast, easy-to-implement strategies?

Quadratic Probing

- Linear probing:
  - Collision at H? Try H, H+1, H+2, H+3,...
- Quadratic probing:
  - Collision at H? Try H, $H+1^2$. $H+2^2$, $H+3^2$, ...
  - Eliminates primary clustering, but can cause "secondary clustering"

Quadratic Probing Tricks (1/2)

- Choose a prime number p for the array size
- Then if $\lambda \leq 0.5$:
  - Guaranteed insertion
    - If there is a "hole", we'll find it
  - No cell is probed twice
- See proof of Theorem 20.4:
  - Suppose that we repeat a probe before trying more than half the slots in the table
  - See that this leads to a contradiction
    - Contradicts fact that the table size is prime

Quadratic Probing Tricks (2/2)

- Use an algebraic trick to calculate next index
  - Replaces mod and general multiplication
  - Difference between successive probes yields:
    - Probe i location, $H_i = (H_{i-1} + 2i - 1) \% M$
  - Just use bit shift to “multiply” i by 2
  - Don’t need mod, since i is at most M/2, so
    - ```
      probeLoc= probeLoc+ (i << 1) - 1;
      if (probeLoc >= M)
              probeLoc -= M;
      ```
      ```

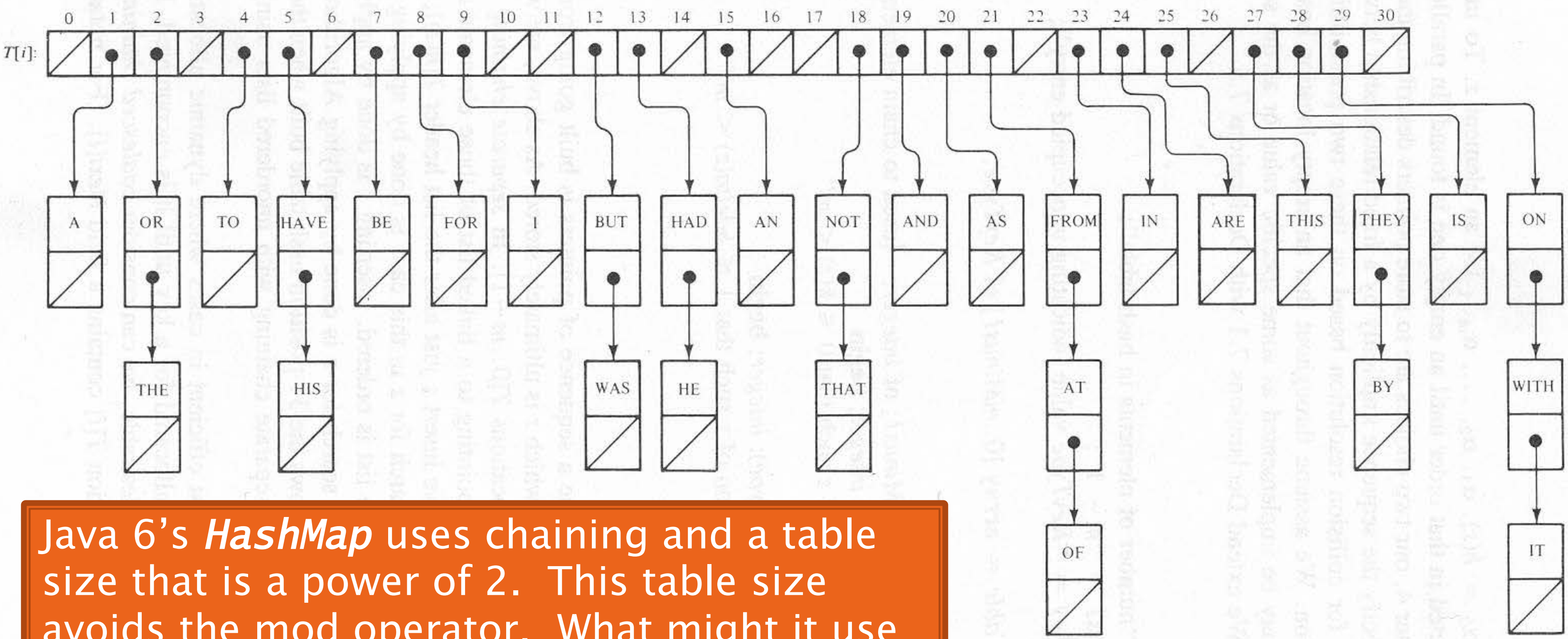
Quadratic probing analysis

- No one has been able to analyze it!
- Experimental data shows that it works well
  - Provided that the array size is prime, and is the table is less than half full

Another Approach: Separate Chaining

- Use an array of **linked lists**
- How would that help resolve collisions?

Hashing with Chaining



Java 6's ***HashMap*** uses chaining and a table size that is a power of 2. This table size avoids the mod operator. What might it use instead to make hashCodes() point to table locations?
(http://www.javaspecialists.eu/archive/Issue054.html)

Editor Trees

- Immersion in tree manipulation