Hash Tables

CS211
Fall 2000

Goal: Design a *Dictionary*

- Operations
  - void insert (key,value)
  - void remove (key)
  - Object get (key)

Array implementation:
Uses an array of (key,value) pairs

	Unsorted	Sorted
insert	O(1)	O(n)
remove	O(n)	O(n)
get	O(n)	O(log n)

n is the number of items currently held in the array

Direct Address Table

- An easy version of a Hash Table
- Assumes the key set is from a small *Universe*
- Example: Addresses on my street
  - Start at 1, go to 40
  - A few lots don't have houses
- For a *Direct Address Table*, we make an array as large as the *Universe*
- To find an entry, we just index to that entry of the array
- Dictionary operations all take O(1) time

What if the Universe is large?

- Idea is to re-use table entries via a *hash function* h
- h: U $\rightarrow$ [0,…,m-1] where m = table size
- h must
  - Be easy to compute
  - Cause few *collisions*
  - Have equal probability for each table position

Typical situation:
U = all legal identifiers

Typical hash function:
h converts each letter to a number and we compute a function of these numbers

A Hashing Example

- Suppose each word below has the following hashCode

jan	7
feb	0
mar	5
apr	2
may	4
jun	7
jul	3
aug	7
sep	2
oct	5

- How do we resolve collisions?
- We'll use chaining: each table position is the head of a list
- For any particular problem, this *might* work terribly
- In practice, using a good hash function, we can assume each position is equally likely

Analysis for Hashing with Chaining

- Analyzed in terms of *load factor* λ = n/m = (items in table)/(table size)
- We count the expected number of *probes* (key comparisons)
- Goal: Determine U = number of probes for an *unsuccessful* search
- Claim U is the same as the average number of items per table position = n/m = λ
- Now we want to determine S = number of probes for a *successful* search (shown on blackboard)

Table Doubling

We know each operation takes time $O(\lambda)$ where λ=n/m

But isn't $\lambda = \Theta(n)$?

What's the deal here? It's still linear time!

Table Doubling:

- Set a bound for λ (call it λ_0)
- Whenever λ reaches this bound we
  - Create a new table, twice as big and
  - Re-insert *all* the data
- Easy to see operations *usually* take time O(1)

Analysis of Table Doubling

- Suppose we reach a state with n items in a table of size m and that we have just completed a table doubling

	Copying Work
Everything has just been copied	n inserts
Half were copied previously	n/2 inserts
Half of those were copied previously	n/4 inserts
…	…
Total work	n + n/2 + n/4 + … = 2n

Analysis of Table Doubling, Cont'd

- Total number of insert operations needed to reach current table = copying work + initial insertions of items
  = 2n + n = 3n inserts
- Each insert takes expected time $O(\lambda_0)$ or O(1), so total expected time to build entire table is O(n)
- Thus, expected time per operation is O(1)
- Disadvantages of table doubling:
  - Worst-case insertion time of O(n) is definitely achieved (but rarely)
  - Thus, not appropriate for time critical operations

Java Hash Functions

- Most Java classes implement the hashCode() method
- hashCode() returns an *int*
- Java's HashMap class uses h(X) = X.hashCode() mod m
- h(X) in detail:

```
int hash = X.hashCode();
int index = (hash & 0x7FFFFFFF) % m;
```

What hashCode() returns:

Integer: uses the int value

Float: converts to a bit representation and treats it as an int

Short **String**s: 37∗previous + value of next character

Long **String**s: sample of 8 characters; 39∗previous + next value

Hash Tables in Java

java.util.HashMap
java.util.HashSet
java.util.Hashtable (legacy)

- Use chaining
- Initial (default) size = 101
- Load factor = λ_0 = 0.75
- Uses table doubling (2∗previous+1)

A node in the *chain* looks like this:

hashCode	key	value	next

original hashCode (before mod m)
[Allows faster rehashing and (possibly) faster key comparison]

Hashing Application: Spell Checking

- We want to create a "spelling dictionary" containing 10,000 words
  - A spelling query should be fast
  - Should return true iff word is contained in dictionary
- Basic idea:
  - Use a Hashtable consisting only of bits (say 100K bytes or about 800,000 bits)
  - Compute a hash value for each word and turn on the corresponding bit in the table
  - What's the probability of a false positive? (It's too high!)
  - Fix: Use more hash functions