CSE 373: Hash Tables

Pete Morcos
University of Washington
4/7/00

http://www.cs.washington.edu/education/courses/cse373/00sp

Why Hash Tables?

- The data structures we've seen so far let us insert and find data in O(N) or O(log N) time
- In practice, programmers often find themselves storing data where N is around, say, 10 to 10000.
  - log N is approx 3 to 14
- It would be ideal to have an O(1) algorithm
  - Speed up that part of the program 3 to 14 times
- Hash tables (sort of) do this

Abstract Model

- We can store structures in arrays
  - A[3] = { "Alan Turing", age 28, height 68" }
  - A[17] = { "Charles Babbage", age 47, height 63" }
- But need to search array to update someone's data
- Hash tables let us effectively say
  - B["Alan Turing"] = {age 28, height 68"}
  - B["Charles Babbage"] = {age 47, height 63"}
- Given a name, we can very quickly get to the data

How?

- We define a *hash function* that converts a *key* (in the example, a string) into an integer
- Then we use the integer (called a *hash code*) to index the array
  - f("Alan Turing") = 79
  - f("Charles Babbage") = 109923
- Keep index within array using modulo arithmetic
  - Usually part of the hash function's definition
  - e.g., if table size is 100, f("Charles Babbage") = 23

Hash Functions

- For integers, could use *i* / table_size
  - Problem if table_size is 100 and inputs are multiples of 1000
- For strings, could use sum of values of chars
  - Problem if table_size is 10000 and inputs are at most 8 chars long—will all hash near beginning of table
- First problem addressed by using prime table size
- Second more difficult; need to pick hash functions that spread input keys evenly through all possible values
- Need to think about possible patterns in the data that could cause these problems

Hashing Strings

- Adding up char values has problems
  - Short strings may not cover whole table
  - Permutations hash to same value
    - "abd", "dab", "bda" all add up to 7
  - Similar strings hash near each other
    - "abc" = 6, "bbe" adds up to 7
- Try treating chars as digits in base 26
  - "abd" = 1 * 26 * 26 + 2 * 26 + 4 = 732
  - "bda" = 2 * 26 * 26 + 4 * 26 + 1 = 1457
  - "dab" = 4 * 26 * 26 + 1 * 26 + 2 = 2732

Designing hash functions

- Be fast: we use hashing only because it speeds things up
- Hash evenly: don't create clumps
- Avoid collisions
- Use whole table: don't use a function that never hashes to some cells
- Be aware of patterns in input keys that might cause problems

Collisions

- Even with a good hash function, sometimes two keys will hash to the same hash code
- Obviously can't store them both in one array cell
- Two main solutions
  - Chaining: use data structure in each cell to hold multiple values
  - Probing: scan array looking for free cells

Chaining

- Most basic way is to have each hash table cell hold a pointer to a linked list
- At each collision, add item to list
- When we search, after using hash code to find proper spot in hash table, need to use list search functions to scan list
- If we already have a list ADT implemented, can just use it here instead of writing code ourselves

Fancier chaining

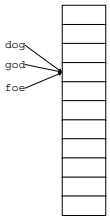
- If a list is good, a binary search tree must be better, right?
  - O(N) reduced to O(log N)
- In practice, rarely done
  - A good hash table is designed to minimize collisions, so not many items will be in each cell
  - log N doesn't pay off until N gets fairly big
  - So, not worth the extra complexity

Load Factor

- The ratio of number of items stored divided by size of table is the *load factor*, λ
- Average length of chained lists will be λ
- Access time is $O(1) + O(\lambda)$
  - So, ideally λ is approx. 1 when we use chaining

Linear Probing

- At collision, scan down array one at a time looking for free cell
- Will always succeed until array is full
  - But, as N approaches table size, insert time approaches O(N)
- General idea of probing is to add to hash function, $i = 1, 2, 3, \ldots$, until a free cell is found
- Linear probing has $F_i = i$

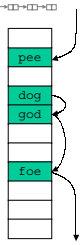


Problems with Linear Probing

- Clusters tend to form
  - Any key that hashes within the cluster will take a while to find a free cell
  - Then it will grow the cluster size, making future additions even costlier
- Thus, even when table is fairly empty, might find some inserts taking several steps to do

Quadratic Probing

- $F_i = i^2$
- This jumps around the table more vigorously
- But, can we be sure it won't jump around forever?
  - If table size is prime, will eventually succeed as long as load factor less than 0.5



About Probing

- Of course, one could define many other probing functions
- Probing is also known as *open addressing*, since the index (address) of a key is no longer a fixed number
- *Must* use lazy deletion with open addressing
  - Why?
- Load factor must be less than 1
  - If we get too many items in a table, must *rehash*
- Main advantage is avoids memory allocation
  - Point of hashing is speed; malloc is slow

Rehashing

- Allocate a bigger table (usually twice the size)
- Copy data from old hash table to new one
  - Note: We could also do this with other array-based ADTs when they fill up
- Unlike other ADTs, can't do a simple copy
  - Hash function depends on table size!
- So, recompute hash value for each key and put into new position in new table

Hash ADT

- As with other ADTs, have Insert and Remove operations
- Unlike other ADTs, we can't have any searching or ordered enumeration
- A hash table is like a mathematical *set*, an unordered collection of elements

Purpose of Hashing

- The only reason to hash is for speed
- Consider a data set of size N, and hash table of size H
  - Time complexity = O(N / H)
  - Space complexity = O(H)
- We are trading space for time. Very common pattern in computer science

Hashing is Pragmatic

- If table size H is roughly N, then time = O(1)
- If table size H is fixed and not a function of N, then time is O(N/H) = O(N)
  - Asymptotically the same!
- Even so, a constant speedup of, say, 1000, is very valuable in practice
- Hash tables extremely common in real programs
- Efficient implementation somewhat more important for hash tables than other ADTs since speed is the only reason to use one

Uses of hash tables

- Compilers store info about variables and functions, and need to access that info repeatedly each time the name appears
  - name (a string) → info about the name
- Game programs see the same board position more than once due to permutations of moves. Want to avoid recomputing the best move.
  - board state (a big structure) → best move
- If you have a string and need to look up something based on the string, you should immediately think of a hash table (as opposed to a bunch of string comparisons)

Hashing Summary

- Ideally performs operations in O(1) time
- Only supports insert/find, no ordering of items
- Parts: table size (prime), hash function (spreads things out), collision strategy
- Chaining collisions allows load factors > 1
- Open addressing must have load factor < 1, but avoidance of malloc speeds things up
- Reason to use: **speed**
- Main cost: **space**

Next time: read 6 - 6.4 (heaps)