Lecture 23
Hashing and Hash Tables

Menu

- Hash tables
- Comparison among various search mechanisms
- Table size
- Hash function
- Modular hash function
- Hash function examples
- Collisions

Hash Tables

- another kind of Table
- **O(1)** in average for insert, lookup, and remove
- use an array named T of *capacity* N
- define a hash function that returns an integer int **H(string key)**
- must return an integer between 0 and N-1
- store the key and info at T[H(key)]
- H() must always return the same integer for a given key

Comparison among various search algorithms

– Linear Search

	name	number
0	Parker	12345
1	Davis	43534
2	Harris	32452
3	Corea	46532
4	Hancock	96562
5	Brecker	37811
6	(empty)	
…	…	
N-1	Marsalis	54323

Linear Search = O(N)

Comparison among various search algorithms

– Binary Search

	name	number
0	Brecker	37811
1	Corea	46532
2	Davis	43534
3	Hancock	96562
4	Harris	32452
5	Marsalis	54323
6	Parker	12345
7	(empty)	
…	…	
N-1	(empty)	

Binary Search = O(log(N))

Comparison among various search algorithms

– Hash Table

name	hash
Brecker	6
Corea	2
Davis	2
Hancock	12
Harris	12
Marsalis	2
Parker	8

Hash code	name/number
0	
1	
2	Corea/46532, Davis/43534, Marsalis/54323
3	
4	
5	
6	Brecker/37811
7	
8	Parker/12345
9	
10	
11	
12	Hancock/96562, Harris/32452

Hash = O(1) if no collision

hash function is simply the sum of ASCII codes of characters in a name (considered all in lowercase) computed mod N=13.

Table Size

- Table size is usually *prime* to avoid bias
- overly large table size means wasted space
- overly small table size means more collisions
- what happens as table size approaches 1?

What is a Hash Function?

- A **hash function** is any well-defined procedure or mathematical function for turning data into an index into an array.
- The values returned by a hash function are called **hash values** or simply **hashes**.
- A hash function H is a transformation that
  - takes a variable-size input k and
  - returns a fixed-size string (or int), which is called the ***hash value*** h (that is, h = H(k))
- In general, a hash function may map several different keys to the same hash value.

Example of a Modular Hash Function

- H(k) = **k mod m** (or **k % m**)
- message1 = ‘723416’
- hash function = modulo 11
- Hash $value_1$ = (7+2+3+4+1+6) mod 11 = 1

- message2 = ‘test’ = ASCII ‘74’, ‘65’, ‘73’, ‘74’
- Hash $value_2$ = (74+65+73+74) mod 11 = 0
- another hash function example: **a*k mod m**

Hash Functions

- a good hash function has the following characteristics:

- avoids collisions
- spreads keys evenly in the array
- inexpensive to compute - must be O(1)

Hash Function for Signed Integer Keys

- remainder after division by table length

  ```
  int hash(int key, int N) {
  return abs(key) % N;
  }
  ```
- if keys are positive, you can eliminate the abs

Hash Functions for Strings

- must be careful to cover range from 0 through capacity-1
- some poor choices
  - summing all the ASCII codes
  - multiplying the ASCII codes
- important insight
  - letters and digits fall in range 0101 and 0172 octal
  - so all useful information is in lowest 6 bits
- hash(s) is O(1)

Hash Functions for Integer Keys

- *Mid-square* method
- Squaring the key value first, and then takes out the middle *r* bits of the result, giving a value in the range 0 to 2^r-1.
- This works well because most or all bits of the key value contribute to the result

Mid-square Method

```
        4567
        4567
    --------
       31969
      27402
     22835
    18268
    --------
    20857489

      4567
```

Hash Functions for String Keys

- This function takes a string as input. It processes the string 4 bytes at a time, and interprets each of the four-byte chunks as a single long integer value.
- The integer values for the 4-byte chunks are added together.
- The resulting sum is converted to the range 0 to $M-1$ using the modulus operator.
- There is nothing special about using 4 characters at a time. Other choices could be made.

Dealing with Collisions

- ***open addressing*** - collision resolution
- key/value pairs are stored in array slots
- ***linear probing***
  - hash(k, i) = (hash1(k) + i) mod N
  - increment hash value by a constant, 1, until free slot is found
  - simplest to implement
  - leads to *primary clustering*
- ***quadratic probing***
  - hash(k, i) = (hash1(k) + c_1*i + c_2*i*i) mod N
  - leads to *secondary clustering*
- ***double hashing***
  - hash(k, i) = (hash1(k) + i*hash2(k)) mod N
    - avoids clustering

Dealing with Collisions

- *separate chaining*
- each array slot is a SearchList
- never gets 'full'
- deletions are not a problem

Dealing with A Full Table

- allocate a larger hash table
- rehash each from the smaller into the larger
- delete the smaller

Why hash table can give us O(1) performance?

- It appears most search mechanisms have performance at O(N) or O(logN)
- So why hash table can give us best performance?
- Where does the magic come from?

Summary

- Hash tables
- Comparison among various search mechanisms
- Table size
- Hash function
- Modular hash function
- Hash function examples
- Collisions

Readings

- [Mar07] Read 5.1-5.4, 5.6
- [Mar13] Read 5.1-5.4, 5.6