Hashtables

“Hashtables are arguably the single most important data structure known to humankind.”

- Google.

ZyBook Chapter 5

(5.9 is not actually about hash tables)

High Level **Concept**

- Big array with a *string* for an index rather than an int.

  aInts[1] = 3; // array

  printf(“%d”, aInts[1]);

  htBirthdays[“friend”] = 1992; // **hash table**

  printf(“%d”, htBirthdays[“friend”]);

- Key: string (can by any type)
  Value: int *(or struct)*

Why?

- Fast
  - constant time lookup
  - no rebalancing
- Easy to use
  - store/lookup by any key
- Unordered data
  - trees are ordered

C++: std::map

Java: java.util.HashMap

Python: Dictionary

Hash Function

- Map a key to an array index (int)
  - String to int

```
int sillyHash(char *szKey)
{
  int  i, value = 0;
  for(i = 0; i < strlen(szKey); i++)
  {
    value += szKey[i];
  }
  return value;
}
```

Choosing a *good* hash function can be really hard.

htBirthdays[“friend”] = 1992; // **hash table**

- htBirthdays is really just an array so this really operates like:

  htBirthdays[sillyHash(“friend”)] = 1992;

  printf(“%d”, htBirthdays[sillyHash(“friend”)]);

More formally

- A hash table maps keys to a certain location
  - bucket

- A hash function changes the key into an index value
  - hash value
  - Use case: turn a string into an int

(picture)

Picture

Hash Function

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Collision

- Very difficult to have a hash function that never produces a collision
  - Perfect Hash - each key maps to an empty bucket!
  - very rare

- How should we handle the collision?

Collisions

- Open Addressing
  - find an empty slot
  - Linear probing
  - Quadratic probing
  - re-hash (double hash)
- Separate Chaining
  - each bucket is a linked list!

Open Addressing

- If two keys, map to the same bucket, we have a collision
- Find unoccupied space for the second key
- Must be able to find both again next time!
- Analysis: (sum of the # of probes to locate each key) / # keys in the table

Open Addressing

- Find another open bucket
- bucket =

Linear Probing

- On collision, use the next empty spot
- On collision at h(n) try:
  - (h(n) +1) % tableSize
  - (h(n) +2) % tableSize
  - etc.
  -

Example

f(i) = i

h(Kn) = n % 11

Insert
M13
G7
Q17
Y25
R18
Z26
F6

Bucket	Data
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Primary Clustering

Quadratic Probing

- If h(n) is occupied, try
  - (h(n) + 1^2) mod table-size,
  - (h(n) + 2^2) mod table-size,
  - and so on until an empty cell is found

Example

f(i) = i^2

h(Kn) = n % 11

Insert
M13
G7
Q17
Y25
R18
Z26
F6

Bucket	Data
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Secondary Clustering

Re-Hashing

- Two hash functions
- The second hash function has to be chosen with care:
  - The sequence should be able to visit all slots in the table.
  - The function must be different from the first to avoid clustering.
  - It must be very simple to compute.

Chaining

- Each bucket is a linked list!
- On collision, add item to list
- How does lookup work?

  How can we make chaining “faster”?

Problem

- Hash the keys M13, G7, Q17, Y25, R18, Z26, and F6 using the hash formula h(Kn) = n mod 9 with the following collision handling technique: (a) linear probing, (b) chaining
- Compute the average number of probes to find an arbitrary key K for both methods.
- avg = (summation of the # of probes to locate each key in the table) / # of keys in the table

How to choose a hash function

- Hash function maps keys to indexes
  - h(K) = M
  - indexes (0 to M-1)

- Problems
  - find suitable function
    - distribute keys evenly across the table
    - minimize collisions
  - find suitable M
  - handle collisions

Hash Functions section 5.7

- Modulo or Division hashing
- Midsquare
- Multiplicative string hash

Division Hashing

- bucket = key % N
- N is length of table AND prime number

Multiplicative string hash

MidSquare

- Turn the key into an integer
  - square the key
  - take some bits from the center of the square

https://research.cs.vt.edu/AVresearch/hashing/midsquare.php

Code

```
// get middle 8 bits from an int

// assume 4 byte integers
unsigned int key = 0x1231a456;
unsigned int middle;
middle = (key & 0x000ff000) >> 12;
printf("%08x %08x\n", key, middle);
```