Introduction to HashTables

Steve Cutchin

Boise State University

February 24, 2021

What Problem Do They Solve?

Why not use arrays for everything?

1. Arrays can be very wasteful:
    - Example Social Security Numbers
    - 999-99-9999 1 billion entries
    - only 319 million people in US as of 2014.
    - Array would be 68 % empty.
2. Non-numeric objects:
    - Store strings in the array. 'LLAPLN19312015'
    - Need to map string to a numeric index.
    - This map is very similar to a hash function.

Hash Tables: The Basics

Hash Table

A[]	Array to hold keys,values.	
insert(A, key, value);	Save key and value in A.	$O(1)$
int findA, key);	Find the key.	$O(1)$
delete(A,key);	Delete Key/Value.	$O(1)$
int hash(key);	Compute Hash Value.	$O(1)$

Hash Function

Two Basic Hash Approaches

- **division** $h(k) = k \mod m$. Use the remainder of k/m. m comes from the size of what object?
- **multiplication** $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ where $0 < A < 1$. This equation will range from zero to m.

Hash Function

What makes a good hash function:

Uniform Hashing: Keys are equally like to go to any hash value.

Fully Utilize: Can generate a hash value for any table entry.

Fast: Hash function computes in $O(1)$

Minimizes Collisions: For a set of keys, keeps collisions to a minimum.

Collisions Since the hash function is applied to **unbounded** keys there are going to be keys that generate the same hash value. These are called **Collisions**.

*It is not the size of the hash table that causes the collision but the nature of the hash function.

Hash Tables

Solving Collisions: There are two approaches:

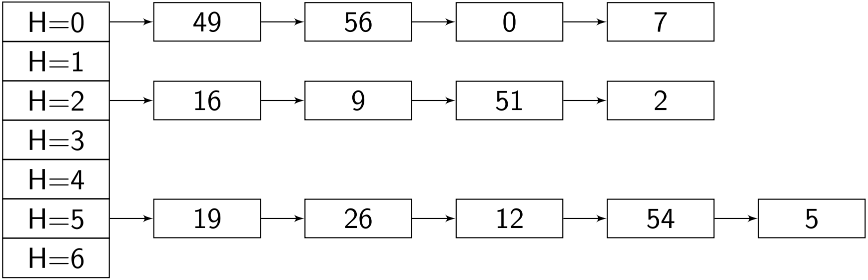
1. Separate Chaining: store the keys in a linked list anchored at the **hash value**.
2. Open Addressing: **rehash** repeatedly to find an empty space in the hash table.

Separate Chaining

Separate Chaining

Hash $= k \mod 7$

Keys = [7, 0, 56, 2,51,49,5,54, 12, 26, 9, 19, 16]



Separate Chaining

Separate Chaining: set method $O(1)$

```
int insert(A,key,value)
    hi = hash(key); // get the hash value.
    listp = A[hi];  // get list linked list
    listp.insert(key,value);  // insert at front.
    return 1; // always room with separate chaining.
```

Separate Chaining: get method $O(n)$

```
int find(A,key)
    hi = hash(key); // get the hash value.
    listp = A[hi];  // get list linked list
    do { if (llistp.key == key) return listp.value;
         listp=listp.next; } while (list);
    return ERROR; // key not present
```

Separate Chaining: delete method $O(n)$

```
 delete(A,key)
    hi = hash(key); // get the hash value.
    listp = A[hi];  // get list linked list
    listp.delete(key);  // assume built in method
```

Open Addressing

- If their is a **collision** we **rehash** the key to generate a new key.
- How many times do we **rehash** until we give up?

Hash functions with rehashing

linear
: We use the hash function $h(k) = (h'(k) + i) \mod m$, specific case $h(k) = (k \mod 7 + i) \mod m$.

quadratic
: We use the hash function $h(k) = (h'(k) + c_1 * i + c_2 * i^2) \mod m$ $h(k) = (k \mod 7 + c_1 * i + c_2 * i^2) \mod 7$, first rehash matches linear when $c_1 = c_2 = 1/2$.

double
: We use the hash function $h(k) = (h_1(k) + i * h_2(k)) \mod m$, $h(k) = (k \mod 7 + i * (k \mod 2)) \mod 7$

General
: Linear and Quadratic are specializations of double hashing, linear $h_2(k) = 1$, quadratic $i * h_2(k) = c_1 * i + c_2 * i^2$.

Open Addressing

Open Addressing: insert method $O(m)$

```
int insert(A,key,value)
    i = 0; hi = key mod A.length; // get the hash value
    while (i < A.length && A[hi] != EMPTY)
     {
          i = i + 1;
          hi = (key mod A.length + i) mod A.length);
    }
    if (i >= A.length) return FALSE; // Table is full.
    A[hi] = value; // assumes fully utilized.
    return TRUE;
```

Open Addressing

```
int find(A, key)
    i = 0; hi = hash(key); // get the hash value.
    while (i < A.length && A[hi] != key && A[hi] != EMPTY) {i = i + 1; hi = (key mod A.length + i) mod A.length);
    if (i >= A.length) return FALSE; // no key
    if (A[hi] == EMPTY) return FALSE; // no key
    return A[hi]; // found the key
```

```
delete(A, key)
    i = 0;
    hi = hash(key); // get the hash value.
    while (i < A.length && A[hi] != EMPTY && A[hi] != key) {i = i + 1; hi = (key mod A.length + i) mod A.length);
    if (i >= A.length) return; // no such key - table full
    if (A[hi] == EMPTY ) return; // no such key
    A[hi] = EMPTY; // what is wrong with this?
```

Hash Tables

Key Issues in Hash Table Performance

Uniform Hashing For any random key the probability of hashing to any specific location in the table is equal to $1/|A|$. Or alternatively, the probability for table locations are the same. This is known as **uniform hashing**.

Fully Utilize For Open Addressing, a rehashing scheme is said to **fully utilize** the table if given all entries in the table are full, for some finite number of steps the rehashing scheme will have inspected every location in the table.

Load Factor The **Load Factor** for a hash table is equal to n/m where n is the number of keys in the table and m is the size of the table.

Hash Tables

What makes a good Uniform Hashing function?
Let's look at the mod function. For example $k \mod 7$. For any series of numbers, say 0 through 21, it will cycle through the numbers 0 through 6 exactly 3 times. What is the probability of a random key falling at given index i:

- Given a key k what is the probability that it $k \mod 7 = i$ for a fixed i?
- How many numbers in 0 to 21 mod to i? **3**.
- What is the probability that k is one of those numbers? $\mathbf{3/21 = 1/7}$.
- So the probability that k fall at any specific index i is $1/7$.
- So the mod function provides uniform hashing.

Full Utilization

Fully Utilize a table: definition: - example problem showing failure.

index	linear	quadratic	double
0	23	23	40
1	47	40	23
2	31	47	31
3	40		
4			47
5		31	
6			
7	63	63	63

Hash Functions:

- Linear: $h(k) = (k \mod 8 + i) \mod 8$
- Quad: $h(k) = (k \mod 8 + c_1 * i + c_2 * i^2)$ mod 8, $c_1 = c_2 = 1/2$
- Double: $h(k) = (k \mod 8 + i * (k \mod 7)) \mod 8$

Load Factor

Load Factor is the ratio of the number of elements to the size of the table: Load Factor =

$$\alpha = \frac{n}{m}$$

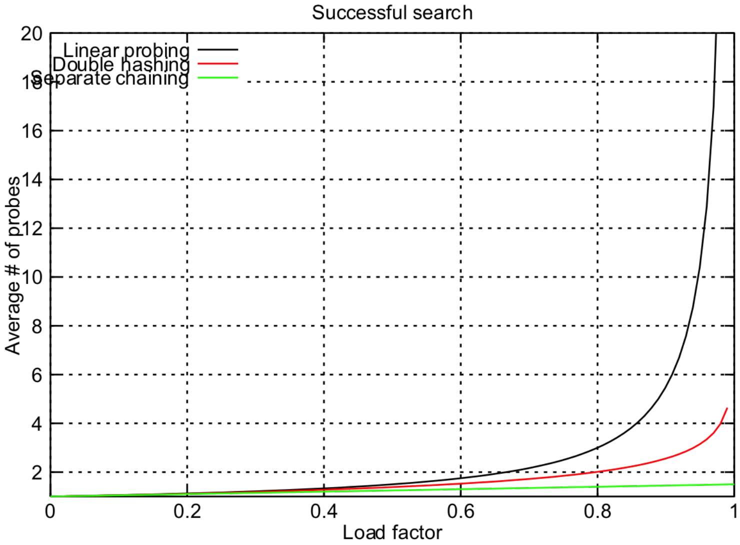
Bounds for Load Factor

Separate Chaining In separate chaining the Load Factor is unbounded because the table can hold more list elements than indices in the table.

Open Addressing In Open Addressing the Load Factor is bounded to the range $0 <= \alpha <= 1$.

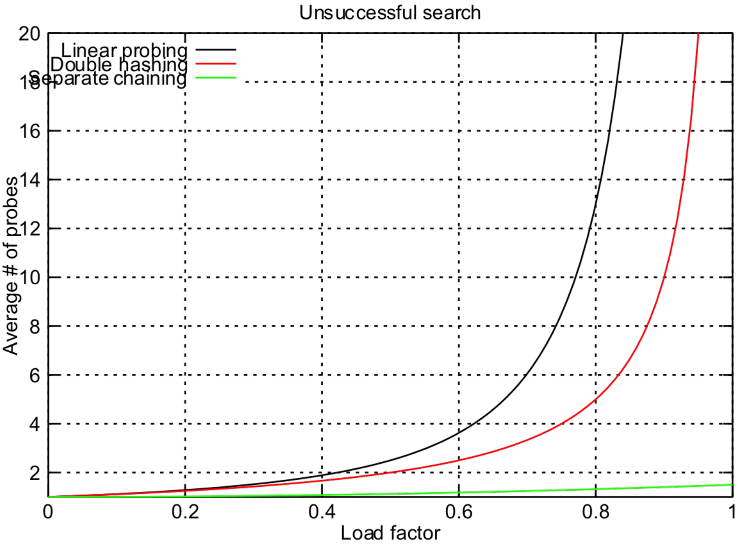
Good Load Factors What are good values for load factors for each method?

Load Factor



Successful search
Linear probing
Double hashing
Separate chaining
Average # of probes
20
18
16
14
12
10
8
6
4
2
0
0.2
0.4
0.6
0.8
1
Load factor

Load Factor


Unsuccessful search
Linear probing
Double hashing
Separate chaining
Average # of probes
Load factor
20
18
16
14
12
10
8
6
4
2
0
0.2
0.4
0.6
0.8
1

Expected Probes for an Unsuccessful Search using linear probing.

HT	8	9		15			17	18		
COUNT	3	2	1	2	1	1	3	2	1	1

Expected Probes =

$$\frac{\sum_{i=1}^{|A|} p(i)}{|A|}$$

Sum $= 3+2+1+2+1+1+3+2+1+1 = 17, |A| = 10$

Expected Probes $= 17/10 = 1.7$. Load Factor $= 5/10 = 0.5$.
Challenge Problem: How to calculate probes in a successful search?

Expected Number of Probes for Open Addressing

- A probe is any inspection of the table. Minimum probe for insert, find, or delete is 1 probe.
- For Unsuccessful search expected number of probes is at most $\frac{1}{(1-\alpha)}$ where $\alpha = n/m < 1$.
- Inserting an element takes at most $\frac{1}{(1-\alpha)}$ where $\alpha = n/m < 1$ on average.
- The number of probes in a successful search is at most $\frac{1}{\alpha} \ln \frac{1}{(1-\alpha)}$ where $\alpha = n/m < 1$.

Cuckoo Hashing

A type of Open Addressing.

Cuckoo Hashing Uses two (or more) hash functions. If the first hash fails, the second is used. If the second hash fails, the old key/value in the Table is replaced with the new one and the old value is rehashed using alternating rehashing, swapping old values as they occur. This technique leads to very space efficient tables and very rapid table lookups. What fundamental computer science principal is exhibited here?