

Hash Tables

Bernhard von Stengel

Algorithms and Computation
MA407, LSE

see: Cormen, Leiserson, Rivest, Stein [CLRS],
Introduction to Algorithms, 2nd ed., Chapter 11.

Dictionary operations

Given a set of data items, the following **dictionary operations** often suffice:

- **inserting** a key (together with “satellite data”)
- **searching** for a key
- **deleting** a key

Note: no sorted output of the entire set!

Hash tables implement these operations so that they take each $O(1)$, i.e. constant, time, **on average**.

Direct-address tables

Table T provides a place $T[k]$ for each possible key k .

$T[k]$ can be boolean (1 or 0, for “present” or “not”), or be a link to the satellite data (null for nonpresent keys), suppose x is (link to) such data item with key $x.key$.

Direct-address-search (T, k):

return $T[k]$

Direct-address-insert (T, x):

$T[x.key] = x$

Direct-address-insert (T, x):

$T[x.key] = \text{null}$

Terminology

U = set of possible keys

T = table of **size** m with places $T[0], T[1], \dots, T[m-1]$

direct-address table: $m = |U|$, usually too large

hash function $h: U \rightarrow \{0, 1, \dots, m-1\}$

element with key k **hashes to** slot $T[h(k)]$.

Collision: keys x, y with $h(x) = h(y)$.

Collision resolution by chaining

Each table entry is beginning of **linked list** that stores items hashed to that slot.

Idea: these lists are short and so **linear search** is not too expensive on them on average.

Chained-hash-insert (T, x):

insert x at head of the list $T[h(x.key)]$

Chained-hash-search (T, k):

search for item with key k in list $T[h(k)]$

Chained-hash-delete (T, x):

delete x from list $T[h(k)]$

Load factor

Load factor = n/m for table of size **m** with **n** keys

Note: for chained hashing can be greater than 1.

Average length of list per slot is n / m . Why?

Running time analysis

Load factor = n/m for table of size m with n keys

Theorem [11.1 in CLRS, page 227]:

In a hash table in which collisions are resolved by chaining, an **unsuccessful** search takes expected time $\Theta(1 + n/m)$ if there is **simple uniform hashing**.

Simple uniform hashing = every slot is hashed to with equal probability, even for “non-random” set of keys (property of “**good**” hash function).

Note: Time $O(1)$ always needed for computing the hash function.

Running time for successful search

Theorem [11.2 in CLRS, page 227]:

In a hash table in which collisions are resolved by chaining, a **successful** search teakes expected time $\Theta(2 + n/m / 2)$ if there is simple uniform hashing (same order of running time as unsuccessful search).

First 2 steps:

- compute hash function
- search in list of length **at least 1**

$n/m / 2$ instead of n/m : Find element on average in middle of list rather than “not found” at end of list.

Hash functions

Assumption: key = positive integer (less than some N)
Strings like “Hello” can also be seen that way.

Division method: Table size **m** as prime (*not*: power of 2, would only look at last $\log m$ bits of the key),

$$h(k) = k \bmod m.$$

Multiplication method: Take $0 < A < 1$,

$$h(k) = \lfloor m (k A \bmod 1) \rfloor$$

e.g. $A = \text{“Golden Ratio” } (\sqrt{5} - 1) / 2 = 0.618\dots$

Open addressing

Idea: Links used in chaining take space and time to follow them, use space instead to create a larger table where keys are stored **directly** in the table.

Hence the table can “fill up” and the load factor n/m (n keys for m slots) can never be larger than 1.

We will see: load factors near 1 degrade performance, better “re-hash” to larger table size (e.g. double m) instead.

Probe sequence

Extend hash function h with second argument i that says how often key has been tried:

$$h: U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$$

probe sequence of trying to find a free slot

$$h(k, 0), h(k, 1), h(k, 2), \dots, h(k, m-1)$$

is permutation of $0, 1, \dots, m-1$ to guarantee that a free slot will be found.

Inserting into an open-address table

Open-Hash-Insert(T, k):

```
for (i = 0; i < m; i++)  
{  
    j = h(k, i);  
    if ( T[j] == FREE )  
    {  
        T[j] = k;  
        return j;  
    }  
}  
error "hash table overflow"
```

Searching an open-address table

Open-Hash-Search(T, k):

// finds position in table or where to put k

```
for (i = 0; i < m; i++)
```

```
{
```

```
    j = h(k, i);
```

```
    if ( T[j] == FREE or T[j] == k )
```

```
        return j;
```

```
}
```

```
return -1; // table full, no place for k
```

Probe sequences

(recall): $h(k, 0), h(k, 1), h(k, 2), \dots, h(k, m-1)$

Linear probe sequence: $h(k, i) = (H(k) + i d) \bmod m,$

single hash function $H,$

constant d , relatively prime to m (why?)

Double hashing: $h(k, i) = (H(k) + i H'(k)) \bmod m$

two uniform hash functions $H, H'.$

Running time for unsuccessful search

Theorem [11.6 in CLRS, page 241]:

In an open-address hash table with **load factor** n/m , the expected number of probes in an **unsuccessful** search is at most $1 / (1 - n/m)$, assuming uniform hashing.

Running time for successful search

Theorem [11.6 in CLRS, page 241]:

In an open-address hash table with **load factor** $n/m < 1$, the expected number of probes in a **successful** search, assuming uniform hashing, is at most

$$1 / (n/m) \log (1 / (1 - n/m)).$$

(log = ln = natural logarithm)