Hashing Algorithms

- Hash functions
- Separate Chaining
- Linear Probing
- Double Hashing
Symbol-Table ADT

Records with keys (priorities)

basic operations

• insert
• search
• create
• test if empty
• destroy
• copy

Problem solved (?)

• balanced, randomized trees use
  $O(\lg N)$ comparisons

Is $\lg N$ required?

• no (and yes)

Are comparisons necessary?

• no

ST.h

void STinit();
void STinsert(Item);
Item STsearch(Key);
int STempty();

ST interface in C
“Guaranteed” asymptotic costs for an ST with N items

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>search</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>randomized BST*</td>
<td>lg N</td>
<td>lg N</td>
</tr>
<tr>
<td>red-black BST</td>
<td>lg N</td>
<td>lg N</td>
</tr>
</tbody>
</table>

* assumes system can produce “random” numbers

Can we do better?
Save items in a **key-indexed table** (index is a function of the key)

**Hash function**
- method for computing table index from key

**Collision resolution strategy**
- algorithm and data structure to handle two keys that hash to the same index

**Classic time-space tradeoff**
- no space limitation:
  - trivial hash function with key as address
- no time limitation:
  - trivial collision resolution: sequential search
- limitations on both time and space (the real world)
  - hashing
Goal: random map (each table position equally likely for each key)

Treat key as integer, use prime table size $M$

- hash function: $h(K) = K \mod M$

**Ex: 4-char keys, table size 101**

<table>
<thead>
<tr>
<th>binary</th>
<th>01100001011000100110001101100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>hex</td>
<td>6 1 6 2 6 3 6 4</td>
</tr>
<tr>
<td>ascii</td>
<td>a    b    c    d</td>
</tr>
</tbody>
</table>

Huge number of keys, small table: most collide!

- **abcd** hashes to **11**
  - $0x61626364 = 1633831724$
  - $1633831724 \mod 101 = 11$

- **dcba** hashes to **57**
  - $0x64636261 = 1684234849$
  - $1633831724 \mod 101 = 57$

- **abbc** also hashes to **57**
  - $0x61626263 = 1633837667$
  - $1633837667 \mod 101 = 57$
Goal: **random map** (each table position equally likely for each key)

Treat key as long integer, use **prime** table size $M$

- use **same** hash function: $h(K) = K \mod M$
- compute value with Horner’s method

**Ex:** `abcd` hashes to **11**

$$0x61626364 = 256 \times (256 \times (256 \times 97 + 98) + 99) + 100$$

$$16338831724 \mod 101 = 11$$

**numbers too big?**

**OK to take mod after each op**

- $256 \times 97 + 98 = 24930 \mod 101 = 84$
- $256 \times 84 + 99 = 21603 \mod 101 = 90$
- $256 \times 90 + 100 = 23140 \mod 101 = 11$

... can continue indefinitely, for any length key

How much work to hash a string of length $N$?

$N$ add, multiply, and mod ops

---

**Hash function (long keys)**

```c
int hash(char *v, int M) {
    int h, a = 117;
    for (h = 0; *v != '\0'; v++)
        h = (a*h + *v) % M;
    return h;
}
```

Hash function for strings in C

Uniform hashing: use a different random multiplier for each digit.
Collision Resolution

Two approaches

Separate chaining
  • $M$ much smaller than $N$
  • $\sim N/M$ keys per table position
  • put keys that collide in a list
  • need to search lists

Open addressing (linear probing, double hashing)
  • $M$ much larger than $N$
  • plenty of empty table slots
  • when a new key collides, find an empty slot
  • complex collision patterns
Separate chaining

Hash to an array of linked lists

Hash
- map key to value between 0 and M-1

Array
- constant-time access to list with key

Linked lists
- constant-time insert
- search through list using elementary algorithm

M too large: too many empty array entries
M too small: lists too long

Typical choice M ~ N/10: constant-time search/insert

Theorem (from classical probability theory):
Probability that any list length is > tN/M is exponentially small in t

Trivial: average list length is N/M
Worst: all keys hash to same list

Guarantee depends on hash function being random map
Linear probing

Hash to a large array of items, use sequential search within clusters

Hash
• map key to value between 0 and M-1

Large array
• at least twice as many slots as items

Cluster
• contiguous block of items
• search through cluster using elementary algorithm for arrays

M too large: too many empty array entries
M too small: clusters coalesce

Typical choice M ~ 2N: constant-time search/insert

Guarantees depend on hash function being random map

Theorem (beyond classical probability theory):

- **insert:** \( \frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right) \)
- **search:** \( \frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right) \)

Trivial: average list length is N/M \( \equiv \alpha \)

Worst: all keys hash to same list
Double hashing

Avoid clustering by using second hash to compute skip for search

Hash
• map key to array index between 0 and M-1

Second hash
• map key to nonzero skip value (best if relatively prime to M)
• quick hack OK
  Ex: 1 + (k mod 97)

Avoids clustering
• skip values give different search paths for keys that collide

Typical choice M ~ 2N: constant-time search/insert
Disadvantage: delete cumbersome to implement

Theorem (deep):
insert: $\frac{1}{1-\alpha}$
search: $\frac{1}{\alpha} \ln(1+\alpha)$

Guarantees depend on hash functions being random map

Trivial: average list length is N/M $\equiv \alpha$
Worst: all keys hash to same list and same skip
Double hashing ST implementation

```c
static Item *st;  // code assumes Items are pointers, initialized to NULL

void STinsert(Item x) {
    Key v = ITEMkey(x);
    int i = hash(v, M);
    int skip = hashtwo(v, M);
    while (st[i] != NULL) i = (i+skip) % M;
    st[i] = x; N++;
}

Item STsearch(Key v) {
    int i = hash(v, M);
    int skip = hashtwo(v, M);
    while (st[i] != NULL)
        if eq(v, ITEMkey(st[i])) return st[i];
        else i = (i+skip) % M;
    return NULL;
}
```
Separate chaining vs. linear probing/double hashing

- space for links vs. empty table slots
- small table + linked allocation vs. big coherant array

Linear probing vs. double hashing

<table>
<thead>
<tr>
<th>Load Factor (%)</th>
<th>Linear Probing</th>
<th>Double Hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>1.5 search</td>
<td>1.4 search</td>
</tr>
<tr>
<td></td>
<td>2.5 insert</td>
<td>1.6 insert</td>
</tr>
<tr>
<td>66%</td>
<td>2.0 search</td>
<td>1.5 insert</td>
</tr>
<tr>
<td></td>
<td>5.0 insert</td>
<td>2.0 insert</td>
</tr>
<tr>
<td>75%</td>
<td>3.0 search</td>
<td>1.8 insert</td>
</tr>
<tr>
<td></td>
<td>8.5 insert</td>
<td>3.0 insert</td>
</tr>
<tr>
<td>90%</td>
<td>5.5</td>
<td>2.6</td>
</tr>
</tbody>
</table>

Hashing vs. red-black BSTs

- arithmetic to compute hash vs. comparison
- hashing performance guarantee is weaker (but with simpler code)
- easier to support other ST ADT operations with BSTs
"Guaranteed" asymptotic costs for an ST with \( N \) items

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>search</th>
<th>delete</th>
<th>find kth largest</th>
<th>sort</th>
<th>join</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>( N )</td>
<td>1</td>
<td>( N )</td>
<td>( N \lg N )</td>
<td>( N )</td>
</tr>
<tr>
<td>BST</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
<td>( N )</td>
</tr>
<tr>
<td>randomized BST</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>( N )</td>
<td>( \lg N )</td>
</tr>
<tr>
<td>red-black BST</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>( N )</td>
<td>( \lg N )</td>
</tr>
<tr>
<td>hashing</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( N )</td>
<td>( N \lg N )</td>
<td>( N )</td>
</tr>
</tbody>
</table>

Not really: need \( \lg N \) bits to distinguish \( N \) keys

* assumes system can produce “random” numbers
* assumes our hash functions can produce random values for all keys

Can we do better?

Tough to be sure....