Symbol Table, Dictionary
- records with keys
- INSERT
- SEARCH

Balanced trees, randomized trees
- use $O(\lg N)$ comparisons

Is $\lg N$ required?
- (no, and yes)

Are comparisons necessary?
- (no)
Hashing: basic plan

Save keys in a table, at a location determined by the key

KEY-INDEXED TABLE

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

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
    hashing
Hash function for short keys

Treat key as integer, use PRIME table size M
- \( h(K) = K \mod M \)

**Ex:** four-character keys, table size 101

<table>
<thead>
<tr>
<th>bin</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>

Key "abcd" hashes to 11

\[
0x61626364 = 1633831724
\]

\[
1633831724 \mod 101 = 11
\]

Key "dcba" hashes to 57

\[
0x64636261 = 1684234849
\]

\[
1633831724 \mod 101 = 57
\]

Key "abbc" also hashes to 57

\[
0x61626263 = 1633837667
\]

\[
1633837667 \mod 101 = 57
\]

**Obvious point:**
- huge number of keys, small table: most collide!
Hash function for long keys (strings)

Same function: \( h(K) = K \mod M \)

Need multiprecision arithmetic calculation
- Use Horner's method

Ex: (check with 4 chars; works for any length)

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

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

take mod after each multiplication:

\[
\begin{align*}
256 \times 97 + 98 &= 24930 \mod 101 = 84 \\
256 \times 84 + 99 &= 21603 \mod 101 = 90 \\
256 \times 90 + 100 &= 23140 \mod 101 = 11
\end{align*}
\]
int hash(char *v, int M)
{
    int h, a = 117;
    for (h = 0; *v != ' '; v++)
        h = (a*h + *v) % M;
    return h;
}

Scramble by replacing 256 by 117

Uniform hashing:
- use a different random value for each digit
Collisions

N keys, table size M
How many insertions until the first collision?

**BIRTHDAY PARADOX** (classical probability theory)
- Assume hash function "random"
- Expected insertions to first collision (table size M):
  \[ M \approx \sqrt{\pi M/2} \]

<table>
<thead>
<tr>
<th>N</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\sqrt{\pi M/2})</td>
<td>12</td>
<td>40</td>
<td>125</td>
</tr>
</tbody>
</table>

**Option 1:** Allow \( N \gg M \)
- put keys hashing to i in a list
- about \( N/M \) keys per list

**Option 2:** Keep \( N < M \)
- put keys somewhere in table
- complex collision pattern
Collisions (continued)

Experiment 1:
- generate random probes between 0 and 100
  - 84 35 45 32 89 1 58 16 38 69 5 90 16 53 61 ...
- collision at 13th as predicted

Experiment 2:
- use hash function to scatter 4-char keys

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bcba</td>
<td>47</td>
<td>ccad</td>
<td>1</td>
<td>baca</td>
<td>26</td>
</tr>
<tr>
<td>bddc</td>
<td>43</td>
<td>bdac</td>
<td>83</td>
<td>dbcb</td>
<td>24</td>
</tr>
<tr>
<td>dabc</td>
<td>85</td>
<td>dabb</td>
<td>84</td>
<td>dbab</td>
<td>17</td>
</tr>
<tr>
<td>dbdb</td>
<td>78</td>
<td>dcbd</td>
<td>60</td>
<td>dbdd</td>
<td>80</td>
</tr>
<tr>
<td>babb</td>
<td>74</td>
<td>bccc</td>
<td>2</td>
<td>addd</td>
<td>39</td>
</tr>
<tr>
<td>bcbd</td>
<td>50</td>
<td>adbc</td>
<td>31</td>
<td>bcd</td>
<td>55</td>
</tr>
</tbody>
</table>

collision after 20 probes
- still as predicted (standard dev. not small)
Separate chaining

Simple, practical, widely used
Cuts search time by a factor of M over sequential search
Method: M linked lists, one for each table

. 0:  *
. 1:  L A A A A *
. 2:  M X *
. 3:  N C *
. 4:  *
. 5:  E P E E *
. 6:  *
. 7:  G R *
. 8:  H S *
. 9:  I *
. 10:  *
Insert cost: 1
Avg. search cost (successful): N/2M
Avg. search cost (unsuccessful): N/M

Classical balls-and-urns "occupancy" problem
- Probability that some list length is > t(N/M) exponentially small in t
- Long lists unlikely PROVIDED hash is random
- [Analysis doesn't account for bugs or bad hashes]

M large: CONSTANT avg. search time
- independent of how keys are distributed (!)

Keep lists sorted?
- increases insert time to N/2M
- cuts unsuccessful search time to N/2M
Linear Probing

No links, keep everything in table

**Method:** start linear search at hash position
  - (stop when empty position hit)

Still get $O(1)$ avg. search time if table sparse

Very sparse table: like separate chaining
As table fills up: CLUSTERING occurs
  - (infinite loop on full table)
Linear probing code

```c
void STinit(int max)
{
    int i;
    N = 0; M = 2*max;
    st = malloc(M*sizeof(Item));
    for (i = 0; i < M; i++) st[i] = NULLItem;
}

void STinsert(Item item)
{
    Key v = key(item);
    int i = hash(v, M);
    while (!null(i)) i = (i+1) % M;
    st[i] = item; N++;
}

Item STsearch(Key v)
{
    int i = hash(v, M);
    while (!null(i))
        if eq(v, key(st[i])) return st[i];
    else i = (i+1) % M;
    return NULLItem;
}
```
Linear probing analysis

CLUSTERING
- bad phenomenon: items clump together
- long clusters tend to get longer
- avg. search cost grows to M as table fills

Precise analysis very difficult.

THM (Knuth):
- Insert cost: approx. \( \frac{1 + \frac{1}{(1-N/M)^2}}{2} \)
- Search cost (hit): approx. \( \frac{1 + \frac{1}{(1-N/M)}}{2} \)
- Search cost (miss): same as insert

Too slow when table gets 70%-80% full
Double Hashing

Avoid clustering by using 2nd hash to compute skip for search
Double Hashing analysis

Extremely difficult

**THM:** (Guibas-Szemeredi) Nearly equivalent to random probe ideal
- Insert cost: approx. $1/(1-N/M)$
- Search cost (hit): approx. $\ln(1+N/M)/(N/M)$
- Search cost (miss): same as insert

Not too slow until table gets 90%-95% full
Amortized analysis of algorithms

Measure running time for $X$ operations by
- $(\text{total cost of all } X \text{ operations})/ X$

Ex:
- insert $N$ elements in a heap:
  \[(\lg 1 + \lg 2 + \ldots + \lg N) / N = \lg N + O(1)\]

Ex:
- insert $N$ elements in a binomial queue:
  \[(1\times N/2 + 2\times N/4 + 3\times N/8 + \ldots)/N < 2\]

Worst case for a SEQUENCE of operations
- guarantee bound on TOTAL
  (same as cost per operation)
- individual operation may be slow
Dynamic hashing

Hashing:
- grow table while keeping search cost \( O(1) \)
- when number of keys in table doubles
  rebuild to double the size of the table

**Ex:** separate chaining
- avg search cost < 2
- 4M keys in table of size M
- proof by induction: amortized cost < 2
  cost to build: \( x \times 4M \)
  cost to rebuild to new table size 2M: 4M
  amortized cost of first 8M insertions:
  \[
  \frac{(x \times 4M + 4M + 4M)}{8M}
  \]
  \[
  \frac{x}{2} + 1 < x
  \]

Same argument works for other basic ADTs!

**Ex:** stacks, queues in arrays, double hashing
Separate chaining vs. double hashing

Space for separate chaining w/ rehashing
- 4M keys (or links to keys)
- M table links (approx same size as keys)
- 4M links in nodes
- Total space: 9M words for 4M items
- Avg search time: 2

Double hashing in same space
- 4M items, table size 9M
- Avg search time: \(\frac{1}{1-(4/9)} = 1.8\) (10% faster)

Double hashing in same time
- 4M items, avg search time 2
- Space needed: 8M words \(\frac{1}{1-(4/8)} = 2\) (11% less)

Separate chaining advantages
- idiot-proof (doesn’t break)
- no large chunks of memory (is that good?)
**Other ST ADT operations**

**DELETION**
- Separate chaining: trivial
- Linear probing: rehash keys in cluster or use indirect method (see below)
- Double hashing: no easy direct method
  - mark deleted nodes as “deadwood”
  - rebuild periodically to clear deadwood

**SORT, FIND kth largest**
- Separate chaining w/ sorted lists
- Linear probing/double hashing
  - have to do full sort

**JOIN**
- Separate chaining: easy
- Linear probing/double hashing:
  - rehash whole table
Reasons not to use hashing

Hashing achieve ST ADT implementation goal

- search and insert in constant time.

Why use anything else?

- no performance guarantee
- too much arithmetic on long keys
- takes extra space
- doesn't support all ADT ops efficiently
- compare abstraction works for partial order (searching without keys)
Other hashing variants

Perfect hashing
- fixed set of keys
- hash function with no collisions
- good hack for small tables
- not practical for large tables
- totally static

Coalesced hashing
- properly account for link space
- mix hash table, storage allocation

Ordered hashing
- cut costs in half as with ordered lists

Brent's variation
- guarantee constant search cost
- up to M insert cost