Hashing

Victor Eijkhout

Notes for CS 594 – Fall 2004

The basic problem

Storing names and information about them:
associative storage

Issues

Issues

- Insertion

Issues

- Insertion
- Retrieval

Issues

- Insertion
- Retrieval
- Deletion

Simple strategies

- List in order of creation

Simple strategies

- List in order of creation
- $\Rightarrow$ Cheap to create, linear search time, linear deletion

Simple strategies

- List in order of creation
- $\Rightarrow$ Cheap to create, linear search time, linear deletion
- Sorted list

Simple strategies

- List in order of creation
- $\Rightarrow$ Cheap to create, linear search time, linear deletion
- Sorted list
- $\Rightarrow$ Creation in linear time, search logarithmic, deletion linear

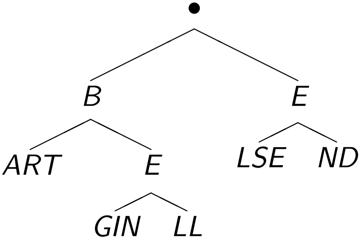
Simple strategies

- List in order of creation
- $\Rightarrow$ Cheap to create, linear search time, linear deletion
- Sorted list
- $\Rightarrow$ Creation in linear time, search logarithmic, deletion linear
- Linear list

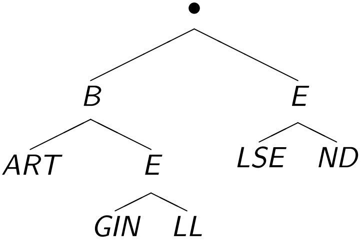
Simple strategies

- List in order of creation
- $\Rightarrow$ Cheap to create, linear search time, linear deletion
- Sorted list
- $\Rightarrow$ Creation in linear time, search logarithmic, deletion linear
- Linear list
- $\Rightarrow$ all linear time

one more strategy



one more strategy



- $\Rightarrow$ all linear in length of string

Hash functions

- Mapping from space of words to space of indices
- Source: unbounded; in practice not extremely large
- Target: array (static/dynamic)

Requirements

Requirements

- Function determined only by input data

Requirements

- Function determined only by input data
- Determined by as much of the data as possible
  `key1`, `key2`,...

Requirements

- Function determined only by input data
- Determined by as much of the data as possible
  `key1`, `key2`,...
- Uniform distribution (clustering bad, collisions really bad)

Requirements

- Function determined only by input data
- Determined by as much of the data as possible
  `key1`, `key2`,...
- Uniform distribution (clustering bad, collisions really bad)
- Similar data, mapped far apart

Good idea: prime numbers

With M size of the hash table:

$$h(K) = K \bmod M, \tag{1}$$

or:

$$h(K) = aK \bmod M, \tag{2}$$

Bad examples:

- M is even, say $M = 2M'$,
  $r = K \bmod M$ say $K = nM + r$ then

$$\begin{aligned} K = 2K' &\quad\Rightarrow\quad r = 2(nM' - K') \\ K = 2K' + 1 &\quad\Rightarrow\quad r = 2(nM' - K') + 1 \end{aligned}$$

  so key even iff number $\Rightarrow$ dependence on last digit
- M multiple of three: anagrams map to same key (sum of digits)
- $\Rightarrow$ M prime, far away from powers of 2

Multiplication instead of division

- $r = K \bmod M = M\big((K/M) \bmod 1\big)$
- $A \approx w/M$, where w maxint
- Then $1/M = A/w$, (A with decimal point to its left).
- from

$$h(K) = \lfloor M \left(\left(\frac{A}{w} K \right) \bmod 1 \right) \rfloor.$$

Example: Bible

- 42,829 unique words,
- into a hash table with 30,241 elements (prime): 76.6% used
- table of size: 30,240 (divisible by 2–9): 60.7% used
- (collisions discussed later)

Two-step hashing

- Mix up characters of the key
- then modulo with table size

Character based hashing

```
h = <some value>
for (i=0; i<len(var); i++)
  h = h + <byte i of string>;
```

prevent anagram problem:

```
h = <some value>
for (i=0; i<len(var); i++)
  h = Rand( h + <byte i of string> );
```

with table of random numbers; also function possible

ELF hash

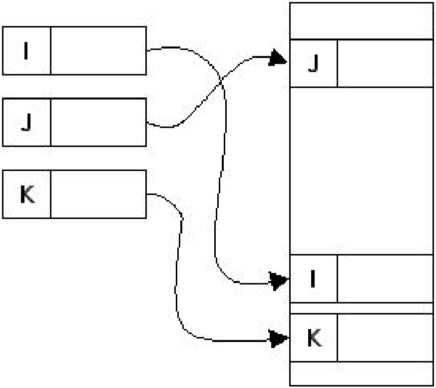
```
/* UNIX ELF hash
 * Published hash algorithm used in the UNIX ELF format
 * for object files
 */
unsigned long hash(char *name)
{
    unsigned long h = 0, g;

    while ( *name ) {
        h = ( h << 4 ) + *name++;
        if ( g = h & 0xF0000000 )
          h ^= g >> 24;
        h &= ~g;
    }
}
```

Another hash function

```
/* djb2
 * This algorithm was first reported by Dan Bernstein
 * many years ago in comp.lang.c
 */
unsigned long hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;
    while (c = *str++) hash = ((hash << 5) + hash) + c;
    return hash;
}
```

Hash tables: collisions

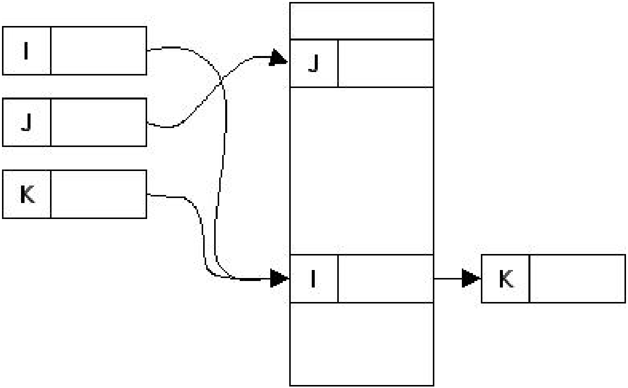
So far so good



Collisions

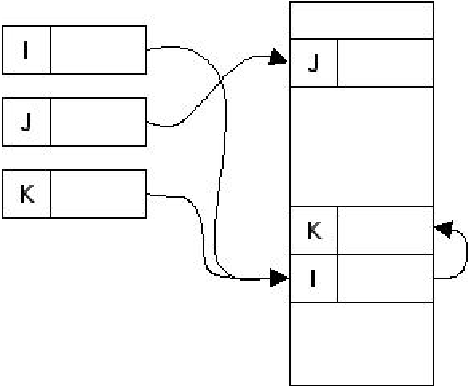
- $k_1 \neq k_2$, $h(k_1) = h(k_2)$
- several strategies; all analysis statistical in nature
- open hash table: solve conflict outside the table
- closed hash table: solve by moving around in the table

Separate chaining



- Pro: no need for searching through hash table
- Con: dynamic storage
- Also: M large to prevent collisions $\Rightarrow$ wasted space

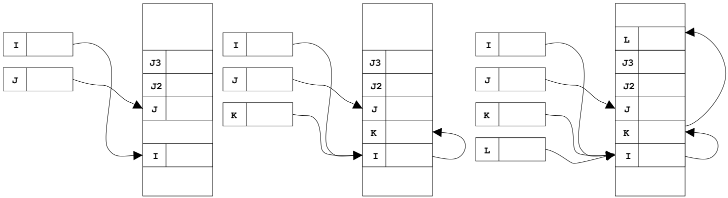
Linear probing



Location occupied: search linearly from first hash

```
addr = Hash(K);
if (IsEmpty(addr)) Insert(K,addr);
else {
    /* see if already stored */
  test:
    if (Table[addr].key == K) return;
    else {
      addr = Table[addr].link; goto test;}
    /* find free cell */
    Free = addr;
    do { Free--; if (Free<0) Free=M-1; }
    while (!IsEmpty(Free) && Free!=addr)
    if (!IsEmpty(Free)) abort;
    else {
      Insert(K,Free); Table[addr].link = Free;}
}
```

Merging blocks in linear probing



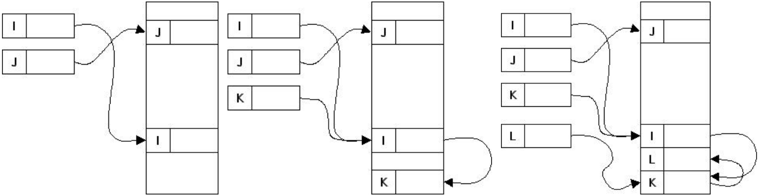
Linear probing analysis

- Clusters forming
- Particularly bad: merging clusters
- Ratio occupied/total: $\alpha = N/M$
  expected search time

$$T \approx \begin{cases} \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right) & \textit{unsuccessful} \\ \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) & \textit{successful} \end{cases}$$

- $\Rightarrow$ increasing as table fills up

Chaining



If location occupied, search from top of table

```
addr = Hash(K); Free = M-1;
if (IsEmpty(addr)) Insert(K,addr);
else {
    /* see if already stored */
  test:
    if (Table[addr].key == K) return;
    else {
      addr = Table[addr].link; goto test;}
    /* find free cell */
    do { Free--; }
    while (!IsEmpty(Free)
    if (Free<0) abort;
    else {
      Insert(K,Free); Table[addr].link = Free;}
}
```

Chaining analysis

- No clusters merging
- Coalescing lists
- Search time (α occupied fraction)

$$T \approx \begin{cases} 1 + (e^{2\alpha} - 1 - 2\alpha)/4 & \textit{unsuccessful} \\ 1 + (e^{2\alpha} - 1 - 2\alpha)/8\alpha + \alpha/4 & \textit{successful} \end{cases}$$

Nonlinear rehashing

- 'Random probing': Try $(h(m) + p_i) \bmod s$, where p_i is a sequence of random numbers (stored)
  prevent secondary collisions
- 'Add the hash': Try $(i \times h(m)) \bmod s$. (s prime)
- Pro: scattered hash keys
- Con: more calculations, worse memory locality

Deleting keys

- Simple in direct chaining
- Very hard in closed hash table methods: can only mark 'unused'

Search in chess programs

- Problem: evaluation board positions
- if position arrived in two ways, no two calculations
- Solution: hash the board, use as key in table of evaluations
- Collisions?

String searching

- Problem: does string (length M) occur in document (length N)
- naive: N comparisons, giving $O(MN)$ complexity
- solution: hash the strings, compare hash values
- (hash function does not distinguish between anagrams)

$$h(k) = \left\{ \sum_i k[i] \right\} \bmod K$$

- string comparison in $O(1)$, $\Rightarrow$ total cost $O(M+N)$

String searching

- Problem: does string (length M) occur in document (length N)
- naive: N comparisons, giving $O(MN)$ complexity
- solution: hash the strings, compare hash values
- (hash function does not distinguish between anagrams)

$$h(k) = \left\{ \sum_i k[i] \right\} \bmod K$$

- cheap updating of the document hash key:

$$h(t[2 \ldots n+1]) = h(t[1 \ldots n]) + t[n+1] - t[1]$$

(with addition/subtraction modulo K)

- string comparison in $O(1)$, $\Rightarrow$ total cost $O(M+N)$

Discussion

Hash table vs trees

Hash table vs trees

- Best case search time can be equal: harder to implement in trees

Hash table vs trees

- Best case search time can be equal: harder to implement in trees
- Trees can become unbalanced: considerable time and effort to balance

Hash table vs trees

- Best case search time can be equal: harder to implement in trees
- Trees can become unbalanced: considerable time and effort to balance
- Threes have dynamic storage: harder to code optimally; worse memory locality

Open vs closed hash tables

Open vs closed hash tables

- Approximately equal performance until the table fills up

Open vs closed hash tables

- Approximately equal performance until the table fills up
- Open: much simpler storage management, especially deletion