Valued Dictionary Abstract Data Types: Hashing techniques

CS240: Data Structures and Data Management
Slide Set 13

Jérémy Barbay

Feb 28th-March 2nd, 2006

Outline

Algorithms using values

Consider other valued algorithms:

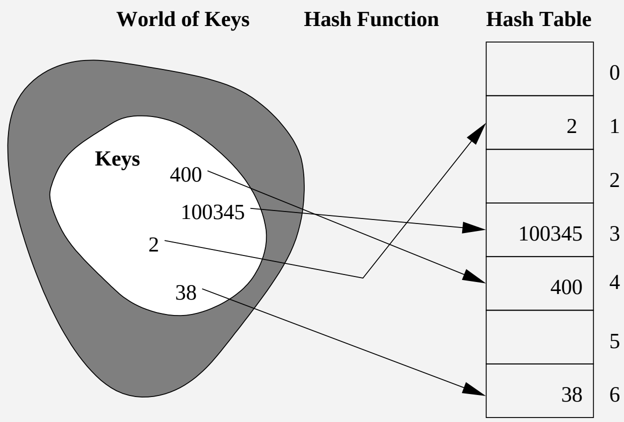
Those algorithms are very good on data following a "well-behaved" distribution, in particular on uniformly distributed data, but can behave badly on a bad distribution.

Principle of Hashing

A projection of the data might be more "well-behaved" (e.g. uniform) than the data itself.

Definition

Use a projection of the key to compute an array index.



Terminology

Definition

A Hash Table is an array of size N,
which elements are often referred to as slots or buckets,
in which n keys are inserted.

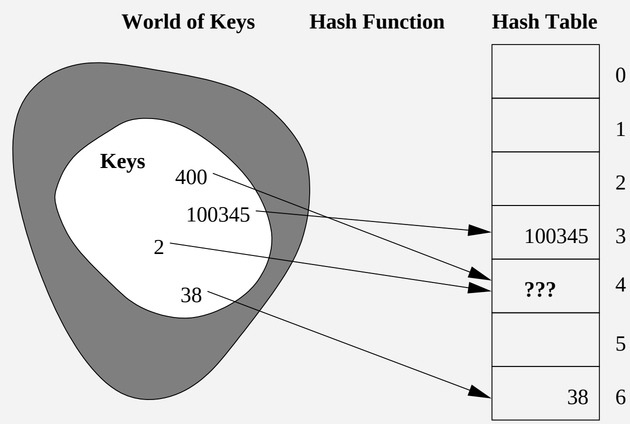
Definition

A Hash Function $h(K)$ maps each key K to an array index $\{0 \ldots N-1\}$

A good hash function distributes keys evenly throughout the table:

$$P[\ h(K) = i\ ] = \frac{1}{N} \quad \text{for all keys } K \text{ and buckets } i$$

Collision



- Two different numbers map to the same range value
- Possible solutions?

Birthday Paradox

- How likely is it in a room with n people that 2 have the same birthday?
- What is the probability of all unique birthdays?

$$P(U) = \frac{\frac{364!}{(365-n)!}}{365^{n-1}}$$

- Likelihood of a shared birthday is $P(S) = 1 - P(U)$

n	$P(S)$
10	
23	
50	
100	

Hashing Techniques

Open Hashing

Associate a linked list with each bucket.
Insert At the begining.

Example

- $h(K) = K \bmod 10$
- Insert Sequence: 52, 18, 70, 22, 44, 38, 62

0 :	70
1 :	
2 :	62, 22, 52
3 :	
4 :	44
5 :	
6 :	
7 :	
8 :	38, 18
9 :	

Open Hashing

Analysis

Assume that h is a uniform hash function,
and let the load factor be $\lambda = \frac{n}{N}$.

Fact

Find and Delete takes time

- *Worst-case:*
- *Average-case*
  - *If search unsuccessful:*
  - *If search successful:*

Fact

Insert takes time

- *if no duplicate check*
- *otherwise on average*

Comments

- Advantages
  - Fast on average, especially
  - easy to implement
  - the dictionary can be larger than the table size.
- Disadvantages
  - Random
  - Good hash functions are data-dependant
  - no guarantee on running time
  - should rehash regularly
  - space (pointers)

Generalization and application

- How else could we store the lists/chains?
  - array (sorted/unsorted)
  - list/chain (sorted / self arranging)
  - trees
  - more generally, any dynamic dictionary structure
- Applications
  - Symbol table in Compilators
  - game playing, to check quickly the reoccurence of positions
  - spell checkers
  - In general, all applications where no deletion is needed.

Hashing Techniques

Closed Hashing (i.e. open addressing)

Does everything within the confines of the hash table array:
If there is a collision, probe a different slot, such that the i'th probe examines the slot

$$(h(K) + f(i)) \bmod N$$

Example

- $f(i) = i$ gives Linear Probing
- $f(i) = i^2$ gives Quadratic Probing

Example
Linear Probing

- $f(i) = i$
- Probe sequence: $h(K) \pmod N$, $h(K)+1 \pmod N$, $h(K)+2 \pmod N$, ...
- $h(K) = K \bmod 11$
- Insert Sequence:

K	48,	67,	9,	53,	21,	75
$h(K)$	4,	1,	9,	9,	10,	9

0 :	21
1 :	67
2 :	75
3 :	
4 :	48
5 :	
6 :	
7 :	
8 :	
9 :	9
10 :	53

Search
Analysis

Fact

Average-case running time of search is

- *if search unsuccessful* $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$;
- *if search successful* $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)}\right)$.

(*see Art of Computer Programming Vol. 3*) (Difficult)

Insertion
Analysis

Insert has same cost as unsuccessful search, but suffers from primary clustering (forming of long sequences).
Solution:

Deletion
Analysis

Delete is much harder – Lazy Delete

- To delete, simply flag the position;
- To insert, search until empty or flagged position found;
- To search, scan until empty bucket found.
- Rehash threshold counts only insertions.

Example

- $f(i) = i$
- probe $h(K), h(K)+1, \ldots \pmod N$
- example: table mod 11

0	
1	67
2	
3	
4	48
5	
6	
7	84
8	
9	9
10	21

- insert 29 ,
  18

Primary Clustering

Linear Hashing has a tendency to for large consecutive blocks.

0 :	399
1 :	5117
2 :	2409
3 :	9000
4 :	1154
5 :	5472
6 :	
7 :	
8 :	8261
9 :	2903
10 :	
11 :	
12 :	4650
13 :	
14 :	6488
15 :	
16 :	871
17 :	7623
18 :	4344

Solution to Primal Clustering

Quadratic Probing

- Example: $f(i) = i^2$
- More generally, i'th probe is to slot:

$$(h(K) + bi + ai^2) \bmod N$$

- Alleviates primary clustering,
  which results in much better performance
- May not hit every cell!
- Suffers from secondary clustering
  - If $h(K_1) = h(K_2)$, the probe sequences are the same

Hashing Techniques

Double Hashing

- Utilize second hash function, $h'(K)$
- $f(i) = i \cdot h'(K)$
- For example $h'(K) = K \pmod{10}$
- Insert Sequence:

	48,	67,	9,	53,	21,	75
pos:	4,	1,	9,	9,	10,	9
				3,		5

0 :	
1 :	67
2 :	
3 :	53
4 :	48
5 :	75
6 :	
7 :	
8 :	
9 :	9
10 :	21

$h'(K)$ should be relatively prime to N,
because otherwise

Hashing Techniques

Ideal Hashing

A hash function generating a seemingly random probe sequence, such that all slots are equally probable to be probed next.

- A probe sequence may hit the same slot twice
- Identical keys follow the same probe sequence
- We will assume no deletions ever take place

Analysis

Ideal Hashing

Theorem

Given an open-address hash table with load factor $\lambda = j/N$, the expected number of probes in an unsuccessful search is at most $1/(1-\lambda)$, assuming uniform hashing.

Proof.

Let u_j be the expected cost of an unsuccessful search with j keys in the table. A slot is empty with probability $1-\lambda$.

$\square$

Analysis

Ideal Hashing

Corollary

Inserting an element into an open-address hash table with load factor λ requires at most $1/(1-\lambda)$ probes on average, assuming uniform hashing.

Analysis

Ideal Hashing

Fact

Given an open-address hash table with load factor $\lambda < 1$, the expected number of probes in a succesful search is at most $\frac{1}{\lambda} \ln \frac{1}{1=\lambda}$

Comments

- Double hashing performs close to ideal if h and h' are uniform
- Quadratic probing also performs reasonably well
- What is the worst-case performance of double hashing?

Summary

- Even when data is not uniformly distributed, a suitable projection of it might be.
- To deal with collisions, use a second hashing function.
- Hashing permits to reach constant time on average, even though worst case is linear.

References:

- Algorithm Design, by Goodrich & Tamassia : pp. 114-124
- Introduction to Algorithms, by Cormen, Leisersen, Rivest & Stein: pp. 221-232, 237-244

Outline

Hash Functions

- Hash functions generally operate on numeric keys
- How do we handle **Strings**?
  - Need to convert an L character string to a number
  - Let c_i by the numeric value of the i'th character

$$K = \sum_{i=0}^{L-1} c_i \cdot r^i$$

Division Hash Functions

- $h(K) = K \pmod{N}$
- Fast
- Generally use a prime table size

Multiplication Hash Functions

- Allows non-prime table sizes
- Choose an A such that $0 < A < 1$ (e.g. $\frac{\sqrt{5}-1}{2}$)
- $h(K) = \lfloor N \cdot \mathit{frac}(K \cdot A) \rfloor$
- $\mathit{frac}(X)$ is the fractional part of a real number X

Universal Hash functions

(see *CLRS*)

- $h(K) = (K \bmod p) \bmod N$
- p is a prime number greater than N
- Universal hashing uses a more sophisticated function:
  - $h(K) = ((aK + b) \bmod p) \bmod N$
  - a is in range $[1..p-1]$
  - b is in range $[0..p-1]$
- a and b chosen randomly at startup
- Also allows non-prime table sizes

Outline

Motivation

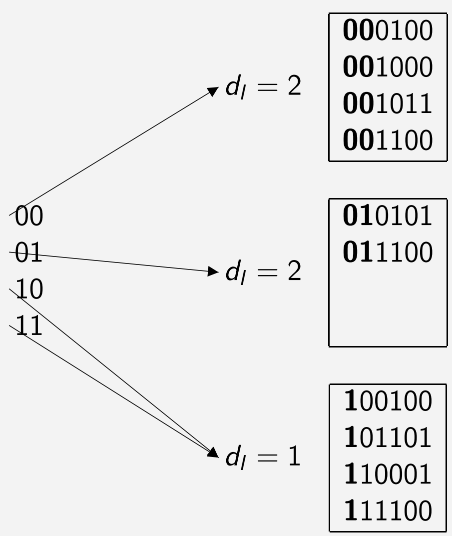
- What if hash table does not fit in main memory?
- Idea: Emulate a
  - Organize data using hash values.
  - Keep a directory at the root node.
  - Only use **part** of the hash value to determine subtree.
  - Require the directory to **extend** as more data is inserted.

Description

- B – the nb of items that fit on one page
  Each leaf page stores at most B key-data pairs
- h – hash function which maps keys to range $[0..2^k - 1]$
  (for some $k > 0$, where k the nb of digits in the hash function.)
- D – the global depth where $D \leq k$
  Root has 2^D pointers to leaf pages
- d_l – the local depth of each leaf page l:
  - Hash values in l have leading d_l bits in common
  - There are 2^{D-d_l} pointers to leaf l

Example

$B = 4$, $k = 6$, $D = 2$



The values in the nodes represent the hash value of the key.

Search

Find(K)
  $x \leftarrow$ *first D bits of* $h(K)$
  Read in page referenced by Root[x]
  Scan page for K

- We naively use the hash function $h(K) = K$
- On the previous slide:
  - `Find( 110001 )`
  - `Find( 001111 )`

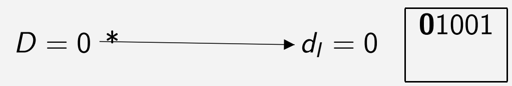
Insert

Read in the leaf page, and three possible cases

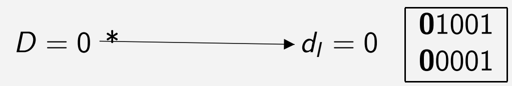
1. There is some room in the leaf page: add the key to it.
2. No room and $d_l < D$
   - Split page (more than once possibly) incrementing d_l
   - Add the key.
3. No room and $d_l = D$
   - Double the size of the directory (incrementing D)
   - Update leaf pointers
   - Split page (more than once possibly) incrementing d_l
   - Add the key.

Example

- We will use $B = 2$ and $k = 5$
- Start with an initially empty extendible hash table
- `Insert( 01001 )`

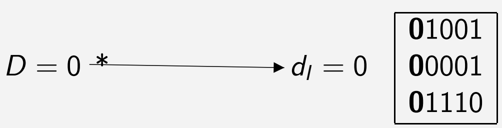


- `Insert( 00001 )`



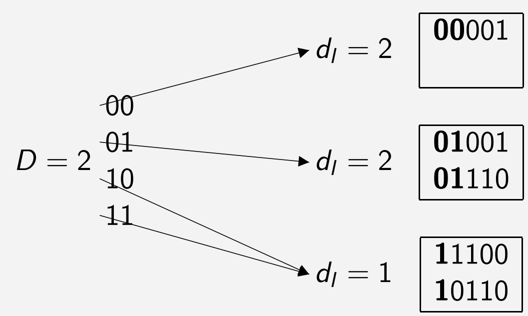
Example

`Insert( 01110 )`



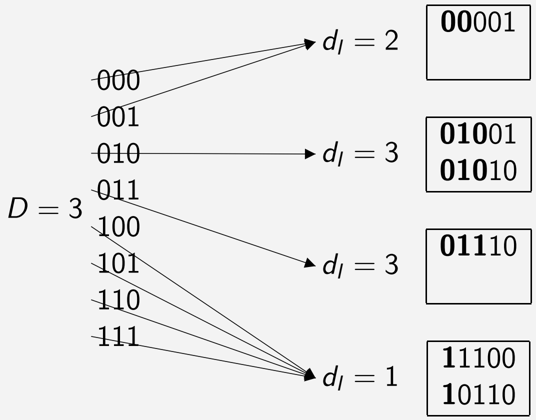
Example

- `Insert( 11100 )`
- `Insert( 10110 )`



Example

- `Insert( 01010 )`

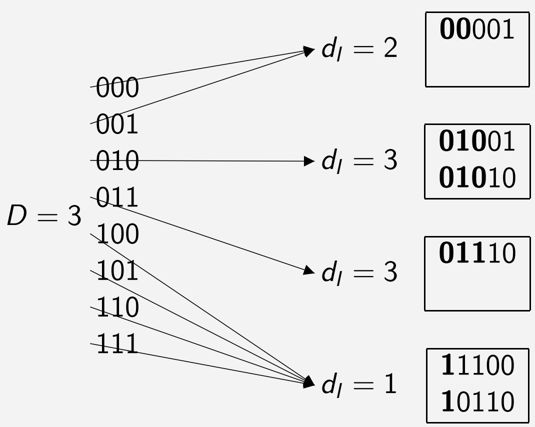


- **Exercise**: `Insert( 11101 )` and then `Insert( 01011 )`

Delete

- Search and remove entry from leaf
- Try and merge with "buddy"
  - Buddy is a leaf with same local depth
  - Buddies agree in first $d_l - 1$ bits
- Several merges may occur
- Shrink root directory if possible

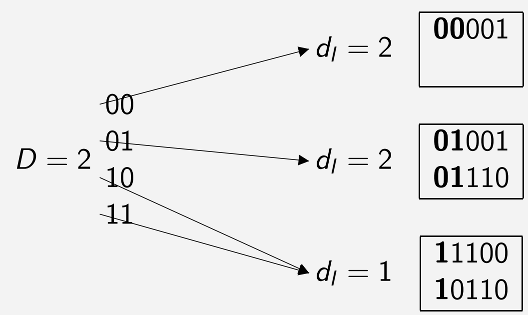
Example



`Delete( 01010 )`

Example

After merging and reducing the dictionary:



Analysis

- Search:
  - If the directory fits in one page:
  - Otherwise:
- Expected number of pages, P, to store n keys is

$$\frac{n}{B \ln 2} \approx 1.44 \frac{n}{B}$$

Hence pages are about 69% full.

Summary

- Hashing can be combined with other techniques, here B-trees.

References:

- Data Structures and Algorithm Analysis, by Mark Allen Weiss: pp. 204-212

Outline

Counting Sort

We want to sort n integer numbers from interval $[0, k]$.

```
clear array count[0..k];

for i:=1 to n
| count[A[i]]++;

pos[0]:=1;
for i:=1 to k
| pos[i]:=pos[i-1]+count[i-1];
// now pos[i] is the first position where
// integer i will come in the sorted array B

for i:=1 to n
| B[pos[A[i]]]:=A[i];
| pos[A[i]]++;
```

Running time: $\Theta(n + k)$

Radix Sort

We want to sort n integer numbers which have at most d digits (can be straightforwardly adapted for strings that have at most d characters).

```
for i:=1 to d
| use counting sort to sort A[1..n] by
| the d-th least significant digit (i.e. k=10)
| // inv: array is sorted by last i digits
```

The algorithm is demonstrated in Figure 8.3 of CLRS.
Running time: $\Theta(nd)$

Bucket Sort

We want to sort n real numbers that are chosen independently randomly uniformly from interval $[0, 1)$.

```
bucket[0..n-1] is an array of empty lists
for i:=1 to n
| insert A[i] into bucket[floor(A[i]*n)]

for i:=0 to n-1
| sort list bucket[i] by insertion sort

concatenate bucket[0], bucket[1], ..., bucket[n-1]
```

The algorithm is demonstrated in Figure 8.4 in CLRS.

Theorem
Expected running time for bucket sort is $O(n)$.

Summary for Valued based sorting

- The principle of hashing can be applied to algorithms as well.
- Radix sort has worst case complexity $\Theta(nd)$ to sort n integers of d digits.
- Bucket sort has expected complexity $O(n)$ to sort elements of bounded value.

Outline

Specific Dictionary ADTs and their DS

Reading Materials

	Topic	GT	CLRS
Unordered	MTF	114-115, 28-30	Not covered.
Ordered	Arrays	pp. 140-151	pp. 253-264
	BST		
	Skiplists	pp. 195-202	Not covered.
	AVL	pp. 152-158	pp. 296 (poorly covered)
	$(2, 4)$-trees	pp. 159-169	pp, 434-452 (indirectly)
	B-Trees	pp. 649-653	pp. 434-452
Valued	Hashing	pp. 114-124	pp. 221-232, 237-244

- GT = Algorithm Design, by Goodrich & Tamassia
- CLRS = Introduction to Algorithms, by Cormen, Leisersen, Rivest & Stein

Additional reference for Extendible Hashing: Data Structures and Algorithm Analysis, by Weiss, pp. 204-212.