Lecture Notes on *Data Structures*

M1522.000900

Seoul National University

Fall 2022

Part IV

Hashing

Hashing

Hashing is a popular technique to store and search key values in a table (known as a *hash table*).

1. a hash table with M entries
2. a hash function $h : \{keyvalues\} \longrightarrow [0..M-1]$
3. a collision resolution policy

First described in public literature by Arnold Dumey [1956].

Example 1

For N records with unique keys in [0..N-1], what is your choice of the size of hash table and hash function?

The best will be an identity function. That is, $h(k) = k$. No collision will occur unless duplicate keys are added.

Example 2

To build a hash table for 1,000 records with keys in [0..20,000), can you still use an identity function?

You can create a hash table of 20,000 entries, but it will be too much waste of space. If you reduce the number of entries in the hash table to, say, 2,000, then the range of the hash function must be reduced too. For example, $h(k) = k \mod 2000$. Collisions may occur even for a set of unique keys.

Hash functions & Collisions

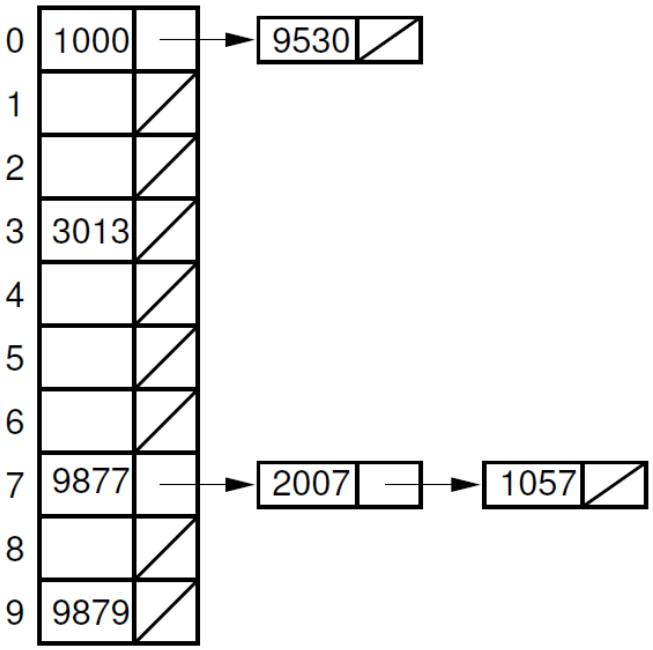
1. data skew vs. hash skew
2. collision resolution: open hashing, closed hashing
3. scalability: static hashing, dynamic hashing
4. Examples of hash functions
   - $h(K) = K \mod 16$
   - $h(K) = (K^2 >> \alpha) \wedge (2^\beta - 1)$ (for $M = 2^\beta$)
   - $h(S) = (\sum_{c \in S} c) \mod M$ (for a string S)

Open Hashing

1. Each slot (or entry) of the hash table is the head of a linked list.
2. A collision is resolved by chaining an overflow entry.
3. Also known as *chained hashing*.

Example 3

Insert the following keys into an initially empty hash table with ten entries ($M = 10$): 9877, 2007, 1000, 9530, 3013, 9879, 1057. Use $h(K) = K \bmod M$ for hash function.



Closed Hashing

Insert: When a collision occurs during insertion, follow a certain *probe sequence* until a free slot or a slot marked as deleted in the hash table is encountered.

Delete: Mark a key as deleted instead of simply removing the key. (Deleting an item in the middle of a probe sequence can cause a subsequent search to stop at the deleted position.)

Search: Follow the *probe sequence* until reaching the search key or a free slot, skipping over slots marked as deleted.

Remark 1

Closed hashing is also known as *open addressing*.

Example 4 (Linear Probing)

Insert the following keys into an initially empty hash table: 9877, 2077, 1000, 9530, 3013, 9879, 1057. Use a closed hashing scheme with $M = 10$, $h(K) = K \bmod M$, and a linear probe sequence $\{(h(K) + i) \bmod M, 0 \leq i < M\}$.

0	1000
1	9530
2	1057
3	3013
4	
5	
6	
7	9877
8	2077
9	9879

- Tends to form clusters. How many collisions?
- The cost of inserting 1057?
- The cost of search for 1057?
- The cost of search for 1058?
- Delete 9879. Can you still find 1057?

Example 5

Insert the following keys into an initially empty hash table: 9877, 2077, 1000, 9530, 3013, 9879, 1057. Use a closed hashing scheme with $M = 10$, $h(K) = K \bmod M$, and a probe sequence $\{(h(K) + c \times i) \bmod M, 0 \leq i < M\}$ for a constant c. Try 2 and 3 for c.

$c = 2$:

0	1000
1	9879
2	9530
3	3013
4	
5	1057
6	
7	9877
8	
9	2077

$c = 3$:

0	2077
1	
2	9879
3	1000
4	
5	1057
6	9530
7	9877
8	
9	3013

Discussions on Linear Probe Sequences

- If c and M are not mutually prime, the probe sequence may not cycle through the entire set of hash entries.
- If $|h(K_1) - h(K_2)| = c$, then the probe sequences of K_1 and K_2 will overlap considerably.
  - Regardless of c and M being mutually prime or not, such pairs of keys are inevitable, and they will increase the cost of search and insertion operations.
  - Fundamental limitation of Linear Probing.

Other Probe Sequences

Example 6 (Quadratic Probing)

Insert the following keys into an initially empty hash table: 9877, 2077, 1000, 9530, 3013, 9879, 1057. Use a closed hashing scheme with $M = 10$, $h(K) = K \bmod M$, and a probe sequence $\{(h(K) + i^2) \bmod M, 0 \leq i < M\}$.

0	1000
1	9530
2	
3	3013
4	
5	
6	1057
7	9877
8	2077
9	9879

Example 7 (Pseudo-Random Probing)

Insert the following keys into an initially empty hash table: 9877, 2077, 1000, 9530, 3013, 9879, 1057. Use a closed hashing scheme with $M = 10$, $h(K) = K \bmod M$, and Pseudo-Random Probing with a probe sequence $\{(h(K) + r_i) \bmod M, 1 \leq i < M\}$, where $\{r_i | 1 \leq i < M\}$ is a random permutation of [1..M-1]. Use {2, 5, 1, 7, 6, 4, 3, 9, 8} for $\{r_i\}$.

0	1000
1	9879
2	9530
3	3013
4	
5	
6	
7	9877
8	1057
9	2077

- How many collisions are caused by inserting 1057?

Example 8 (Double Hashing)

Insert the following keys into an initially empty hash table: 9877, 2077, 1000, 9530, 3013, 9879, 1057. Use a closed hashing scheme with $M = 10$ and $h(K) = K \bmod M$, and Double Hashing with a probe sequence $\{(h(K) + i \times h_2(K)) \bmod M, 0 \leq i < M\}$.

Try $h_2(K) = K^2 \bmod M$.
Note that it should be $h_2(K) \neq 0$. Otherwise, the probe sequence will be empty. So, a better choice would be

$$h_2(K) = \begin{cases} K^2 \bmod M & \text{if } K^2 \bmod M \neq 0 \\ 1 & \text{othewise.} \end{cases}$$

Closed Hashing: Analysis

The cost of insertion

- The cost of a successful search is the same as the cost taken when the key was inserted to the hash table.
- The cost of an unsuccessful search is the same as the cost that would be taken to insert the key to the hash table.
  - If a deletion marker is encountered while inserting a key, the cost of an unsuccessful search may be higher that that of inserting the key.
- So, can we upper-bound the cost of an insertion?
- Assume that
  - no hash entry is probed more than once, and
  - the next hash entry to probe is determined randomly.
- This is known as *uniform hashing*. This is an unrealistic hashing scheme, but we need to assume this to make the analysis tractable in Theorem 1.

Theorem 1

The cost of inserting a key to a closed hash table of load factor $\alpha < 1$ is $\mathcal{O}(1/(1-\alpha))$.

Proof. Let p_i (or q_i) be the probability that a free entry is found after *exactly* (or *at least*) i probes in vain. Assume $\alpha = n/M$ for $n < M$. Then, $p_i = q_i - q_{i+1}$ (for $i \leq n$), $p_i = q_i = 0$ (for $i > n$), and

$$q_1 = \frac{n}{M} = \alpha, \qquad \text{(one probe failed)}$$

$$q_2 = \frac{n(n-1)}{M(M-1)} \leq \alpha^2, \qquad \text{(two probes failed)}$$

$$q_3 = \frac{n(n-1)(n-2)}{M(M-1)(M-2)} \leq \alpha^3, \qquad \text{(three probes failed)}$$

$$\ldots$$

This is because

$$\frac{1}{M-n+1} < \ldots < \frac{n-3}{M-3} < \frac{n-2}{M-2} < \frac{n-1}{M-1} < \frac{n}{M} = \alpha.$$

Therefore, the cost of insertion is

$$\begin{aligned}
& 1 + \sum_{i=1}^{n} i p_i \\
= \; & 1 + \sum_{i=1}^{n} i(q_i - q_{i+1}) \\
= \; & 1 + (q_1 + 2q_2 + 3q_3 + \ldots + nq_n) - (q_2 + 2q_3 + 3q_4 + \ldots + nq_{n+1}) \\
= \; & 1 + \sum_{i=1}^{n} q_i \\
\leq \; & 1 + \sum_{i=1}^{n} \alpha^i = \sum_{i=0}^{n} \alpha^i = \frac{1-\alpha^n}{1-\alpha} \in \mathcal{O}(1/(1-\alpha))
\end{aligned}$$

□

Linear Hashing

Linear hashing is a kind of dynamic, open hashing. For an initially empty hash table of 2^i entries ($i \geq 1$),

- The hash table grows by an entry each time a collision occurs.
  - After 2^i collisions, the hash table will have grown twice large.
  - Some of the keys stored in an old entry are moved to the new entry.
  - No matter where a collision occurs, the i^{th} collision splits the i^{th} entry.
- While the hash table grows from 2^i entries to 2^{i+1} entries,
  - each of the initial 2^i entries is split exactly once,
  - $h_i : K \rightarrow [0..2^i - 1]$ is used for initial lookups.
  - $h_{i+1} : K \rightarrow [0..2^{i+1} - 1]$ is also used for an entry that has been split during this round.

Upon the j^{th} ($1 \leq j \leq 2^i$) collision,

- Let a variable *splitindex* point to the hash entry to split next. (Reset to zero at the beginning of each round.)
- A new empty slot is appended and becomes the $(2^i + j)^{th}$ entry.
- Split (rehash) the keys in the j^{th} entry between this (*i.e.*, "stay") and the new (*i.e.*, "move") entries using the extended hash function h_{i+1}.
- Do *splitindex*++.
- Invariant: *splitindex* $== j$ when dealing with the j^{th} collision has completed.

Remark 2

No matter where the j^{th} collision occurs, the j^{th} entry is the one that is split.

Upon the $(2^i)^{th}$ collision, do the following actions additionally.

- h_i is discarded, and h_{i+1} becomes the default hash function, and a new function h_{i+2} becomes the extended one.
- Reset `splitindex` to zero. This starts a new round of hash table expansion.
- The size of hash table is now 2^{i+1}.

Remark 3

During the round, all the 2^i hash entries are split in a predefined order, namely, from top to bottom of the hash table.

Linear Hashing: Insertion

To insert a key K:

- If the hash entry $h_i(K)$ has not been split in the current round (*i.e.*, $h_i(K) \geq splitindex$), then insert k to the entry.
- Otherwise (*i.e.*, $h_i(K) < splitindex$), insert K to the hash entry pointed to by $h_{i+1}(K)$. This entry is either $h_i(K)$ or $h_i(K) + 2^i$.
- If a collision occurs, then append a new entry, split an existing entry pointed to by `splitindex` and do `splitindex++`.

Remark 4

- A collision is detected only when a new key is inserted. Nothing that happens while splitting an entry is considered a collision.
- The default and extended hash functions, h_i and h_{i+1}, must satisfy the following condition.

$$h_{i+1}(K) = h_i(K) \text{ or } h_{i+1}(K) = h_i(K) + 2^i.$$

Example 9

Insert 20, 15, 5, 9, 16, 8, 13, 4 into an initially empty hash table of 4 entries. Use the linear hashing scheme and hash functions $h_i(K) = K \bmod 2^i$ for $i \geq 2$.

0	20 16 8
1	5 9
2	
3	15
4	20 4
5	5 13
6	
7	15

- Initially, *splitindex* $= 0$.
- $h_i(20) = 0 \geq$ *splitindex*. Insert 20 to $h_i(20) = 0$.
- Insert 15 and 5 to $h_i(15) = 3$ and $h_i(5) = 1$, respectively.
- $h_i(9) = 1 \geq$ *splitindex*. It causes the first collision and split, *splitindex* $= 1$.
- Entry 0 is split, and 20 moves to $h_{i+1}(20) = 4$.
- $h_i(16) = 0 <$ *splitindex*. Insert 16 to $h_{i+1}(16) = 0$.
- $h_i(8) = 0 <$ *splitindex*. Insert 8 to $h_{i+1}(8) = 0$, causing the second collision and split, *splitindex* $= 2$.
- Entry 1 is split, and 5 moves to $h_{i+1}(5) = 5$.
- $h_i(13) = 1 <$ *splitindex*. Insert 13 to $h_{i+1}(13) = 5$, causing the third collision and split, *splitindex* $= 3$.
- Entry 2 is split, but it is empty.
- $h_i(4) = 0 <$ *splitindex*. Insert 4 to $h_{i+1}(4) = 4$, causing the fourth collision and split, *splitindex* $= 0$.
- Entry 3 is split, and 15 moves to $h_{i+1}(15) = 7$.
- Ready for the next round.

Linear Hashing: Search

To search for a key K:

- If the hash entry $h_i(K)$ has not been split in the current round (*i.e.*, $h_i(K) \geq$ *splitindex*), then probe the entry.
- Otherwise (*i.e.*, $h_i(K) <$ *splitindex*), probe the hash entry $h_{i+1}(K)$.

Example 10

Stop after inserting 20, 15, 5, 9, 16, 8 into an initially empty linear hash table of 4 entries, and then search for 5 and 9. Assume the hash functions used in Example 9 are used here too.

More hashing algorithms

- Extendible hashing
- Perfect hashing
- Cuckoo hashing
- Universal hashing
- Fibonacci hashing
- Cryptographic hashing
- ...