WHY HASHING??

Many applications (like search engines and web pages) deal with lots of data. For these the look-ups are time critical. The typical data structures like arrays and lists, may not be sufficient to handle efficient look-ups. Hashing is a solution to achieve look-ups with near constant time, say O(1).

SEARCHING METHODS: A REVIEW

From among the indexing schemes (used to facilitate faster search) studied -

1. The sequential search algorithm takes time proportional to the data size, i.e, O(n).
2. Binary search improves on liner search reducing the search time to O(log n).
3. BST provides O(log n) search efficiency; but the worst-case is O(n).
4. To guarantee O(log n) search time, BST height balancing is required (AVL trees).

For example, assume that 5,000 transaction records (each with a 5-digit identifier, TID) are stored in a database. To search any record by TID can be summarized as -

1. A linked list implementation would take O(n) time.
2. A height balanced tree would give O(log n) access time.
3. Using an array of size 100,000 would give O(1) access time but will lead to a lot of space wastage.
4. Hashing would lead to O(1) access without wasting a lot of space.

HASH TABLE

A **hash table** [or **scatter table**] is an array of some fixed size, usually a prime number. The **key-value pairs** are stored in a fixed size table called a **hash table.** A hash table is partitioned into many **buckets**. Each bucket has many **slots**. Each slot holds one record. A **hash function, *h(k)*** transforms a key, ***k*** into an address in the hash table

There are two types of hash tables:

1. **Open-Addressed Hash Table**
   It is a one-dimensional array indexed by integer values that are computed by an index function called a hash function.
2. **Separate-Chained Hash Table**
   It is a one-dimensional array of linked lists indexed by integer values that are computed by an index function called a hash function.

Typical hash table operations include – initialize(), insert(), delete(), and search().

Alternate Nomenclature

- OPEN HASHING – Closed Addressing – Separate Chaining
  Inventor: H. P. Luhn, IBM 1953
- CLOSED HASHING – Open Addressing – Linear Probing
  Inventor: Amdahl-Boehme-Rocherster-Samuel, IBM 1953,
  Ershov-Peterson,1957 [independently in Russia]

CATEGORIZATION OF HASHING

1. **Static Hashing**
   The hash function maps search-key values to a fixed set of locations.
2. **Dynamic Hashing**
   The hash table can grow to handle more items. The associated hash function must change as the table grows.

HASH FUNCTION & PROPERTIES

The **load factor** of a hash table is the ratio of the number of keys in the table to the size of the hash table. The higher the load factor, the slower the retrieval.

With open addressing, the load factor cannot exceed 1. While with chaining, the load factor often exceeds 1.

A **HASH FUNCTION**, ***h***, is a function which transforms a key from a set, K, into an index in a table of size **n**:

$$h: K \rightarrow \{0, 1, ..., n\text{-}2, n\text{-}1\}$$

1. A key can be a number, a string, a record etc.
2. The size of the set of keys, |K|, to be relatively very large.
3. It is possible for different keys to hash to the same array location. This situation is called **collision** and the colliding keys are called **synonyms**.

A good hash function should:

1. Minimize collisions.
2. Be easy and quick to compute.
3. Distribute key values evenly in the hash table.
4. Use all the information provided in the key.

COMMON HASHING FUNCTIONS

Division-Remainder (using the table size as the divisor)

1. Computes hash value from key using the % operator.

   h(k) := key % table_size
2. Table size that is a power of 2 should be avoided, for it leads to more collisions.
3. Powers of 10 are not good for table sizes when the keys rely on decimal integers.
4. Prime numbers not close to powers of 2 are better table size values.

Truncation or Digit/Character Extraction

1. Works based on the distribution of digits or characters in the key.
2. More evenly distributed digit positions are extracted and used for hashing purposes.
3. For instance, students IDs or ISBN codes may contain common subsequences which may increase the likelihood of collision.
4. Very fast but digits/characters distribution in keys may not be very even.

Folding

1. This method involves partitioning the key, ***k*** into several parts. All parts, except for the last one have the same length. These parts are then added together to obtain the hash address for ***k***.

   h(k) := **Sum_Partions(*P_i*)**; for $0 < i <$ no_of_partitions+1
2. Very useful if we have keys that are very large.
3. Fast and simple especially with bit patterns.
4. A great advantage is ability to transform non-integer keys into integer values.
5. The addition of parts can be done in two ways -
   (a) **Shift-Folding**: Shift all parts except for the last one, so that least significant bit|digit of each part lines up with corresponding bit|digit of the last part.
   (b) **Folding-at-Boundaries**: Before adding the parts (obtained using procedure for Shift-Folding), reverse every other partition and then add.

   For example, consider the key 761$\underline{234}$510$\underline{012}$14. Assume the segments or partitions of 3 digits in length. The partitions for the key are - P_1=761, P_2=234, P_3=510, P_4=012, and P_5=14.

   Using Shift-Folding, the address of home bucket,

   h(k) := $P_1+P_2+P_3+P_4+P_5$:= 761+234+510+012+14 := **1531**

   Using Folding-at-Boundaries, the address of home bucket,

   h(k) := $P_1+\underline{P_2}+P_3+\underline{P_4}+P_5$:= 761+$\underline{432}$+510+$\underline{120}$+14 := **1927**

   [observe that, P_2 and P_4 are reversed before adding]

Radix Conversion

1. Transforms a key into another number base to obtain the hash value.
2. Use number base other than base 10 & base 2 to calculate the hash addresses.

   For example, to map key 55354 in the range 0 to 9999 using base **11**, first radix conversion is done as : $(55354)_{10} = (38652)_{11}$
   Finally, the high-order digit may be truncated to yield **8652** (h(k) in [0 .. 9999])

Mid-Square

1. The key is squared and the middle part of the result taken as the hash value.

   For Example, to map the key 3121 into a hash table of size 1000,
   - Square the key, $(3121)^2 = 9740641$
   - Extract ***m*** middle digits (as required), say **406** as the hash value.

2. Works well if the keys do not contain a lot of leading or trailing zeros.
3. Non-integer keys have to be preprocessed to obtain corresponding integer values.
4. The middle of square hash function is frequently used in symbol table application.

Use of a Random-Number Generator

1. Given a seed as parameter, the method generates a random number.
2. The algorithm must ensure that it always generates the same random value for a given key. It is unlikely for two keys to yield the same random value.
3. The random number produced can be transformed to produce a valid hash value.

APPLICATIONS OF HASH TABLES

1. Database Systems – situations necessitating efficient random access.
2. Symbol Tables (in Assemblers, Compilers to access symbols frequently)
3. Dictionary Lookup (Spelling checkers, Natural language understanding - word sense)
4. Network Processing (route lookup, packet classification, network monitoring)
5. Browser Cashes
6. Heavily used in text processing languages - Perl, Python, Java*

APPROACHES TO COLLISION RESOLUTION

Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys.

SEPARATE CHAINING [Open Hashing]

- Collisions can be resolved by creating a list of keys that map to the same value .
- **Chaining**: An array is used to hold the key and a pointer to a **linked list** (either singly or doubly linked) or a **tree**.
- Each node in the chain is large enough to hold one entry.
- The entries are inserted as nodes in a linked list.

Advantages of Chaining:

1. Insertion can be carried out at the head of the list at the index.
2. The array size is not a limiting factor on the size of the table.
3. To locate a key one only need to search i^{th} chain.

Disavantages of Chaining:

- Higher memory overhead if the table size is small. (also due to pointers)
- Cost is proportional to length of linked list.
- Longer linked lists could negatively impact performance.

LINEAR PROBING [Closed Hashing]

- The hash table remains a simple array of size N
- On insert(k), compute h(k) mod N, if the cell is full, find another by sequentially searching for the next available slot.
- Go to h(k)+1, h(k)+2 so on the place the key. Otherwise hash table is full.
- The Linear Probing function for a probe $\boldsymbol{i}$ is gven by -

$$\boldsymbol{h(k,\ i) = (f(x) + i)\ \%\ N} \text{ (for i = 1, 2, 3, 4, ...)}$$

Advantages of Linear Probing:

- Minimal transformation and easy to implement.

Disavantages of Linear Probing:

- It suffers from **primary clustering**.
- For any load factor, $\lambda < 1$, linear probing will find an empty slot .
- As the loading factor tends to $\lambda > 1/2$, the performance degrades quickly.
- Probe sequences can get longer with time.

<u>**Primary Clustering**</u> : Keys tend to cluster in one part of table. Keys that hash into cluster will be added to the end of the cluster (making it even bigger). Other keys could also get affected if it maps to a crowded neighborhood.

QUADRATIC PROBING

- Another open addressing method
- Resolve collisions by examining certain cells (1, 4, 9, 16, 25, ...) away from the original probe point.
- Collision Policy: Define $h_0(k)$, $h_1(k)$, $h_2(k)$, $h_3(k)$, ...

$$h_i(k) = (\text{hash}(k) + i^2)\ \%\ \text{size}$$

Advantages of Quadratic Probing:

- Easy to implement.
- It is s less likely to encounter Primary Clustering problem.

Disavantages of Quadratic Probing:

- May not find a vacant cell (even when vacant slots exist).
- The hash table must be less than half full [$\lambda < 1/2$] for better performance.
- It suffers from Secondary Clustering.
- If size is prime and $\lambda < 1/2$, then quadratic probing will find an empty slot in size/2 probes or fewer.
- Linear probing always finds a cell (if one exists).
- Difficult to analyze. Ensure table is never half full: if closer, then expand it.

REHASHING

- It is done in anticipation that probes (collisions) would tends to be higher. It is applied across all collision handling schemes.
- When the hash table gets too full, create a bigger table (usually 2x size)
- Hash all the items from the original table into the new table.
- Rehash when??
  – hash table is half full ($\lambda = 0.5$)
  – when an insertion fails on the hash

OPEN ADDRESSING

Advantages [over chaining]

- No need for list or tree structures.
- No need to allocate/deallocate memory during insertion/deletion (slow).

Disadvantages

- Slower insertion – May need several attempts to find an empty slot.
- Hash table needs to be bigger (than chaining-based table) to achieve average case constant time performance.

PERFORMANCE OF HASHING

There are three factors the influence the performance of hashing:

Hash function

- Should distribute the keys and entries evenly throughout the entire table
- Should minimize collisions

Collision resolution strategy

- Open Addressing: store the key/entry in a different position
- Separate Chaining: chain several keys/entries in the same position

Table size

- Too large a table, will cause a wastage of memory
- Too small a table will cause increased collisions and eventually force rehashing (creating a new hash table of larger size and copying the contents of the current hash table into it)
- The size should be appropriate to the hash function used and should typically be a prime number.

Note:

Creation of Hashes and Collision Resolution Methods will be practiced in scheduled classes.