Hashing

Topics

1. Introduction (Hashing, Hash Functions, Hash Table)
2. Collisions & Collision Resolution Techniques
3. Implementation Details for C++ and Java
4. Problems & Solutions
5. Important References

Introduction

Suppose we want to design a system for storing employee records *keyed using phone numbers*. And we want to perform the following queries efficiently:

1. Insert a phone number and corresponding information
2. Search a phone number and fetch the information
3. Delete a phone number and related information

We can think of using the following *data structures* to maintain information about different phone numbers:

1. Sorted/Unsorted Array of phone numbers and records.
2. Sorted/Unsorted Linked List of phone numbers and records.
3. Balanced Binary Search Tree[BBST] with phone numbers as keys
4. Direct Access Table [DAT]

For arrays and linked lists, we need to search in a linear fashion, which can be costly in practice[O(n) for each search]. If we use arrays and keep the data sorted, then a phone number can be searched in O($\log_2(n)$) time using Binary Search, but insert and delete operations become costly as we have to maintain the array in the sorted order [For each *insertion/deletion*, it takes "n-1" shifts in worst case, therefore, O(n) time].

With a BBST, we get moderate search, insert and delete times. All of these operations can be guaranteed to be achieved in O($\log_2(n)$) time.

Another solution that one can think of is to use a DAT where we make a big array and use phone numbers as index in the array. An entry in array is `null` if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in O(1) time.

For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table. This solution has many practical limitations:

1. First problem with this solution is that there's a huge space requirement. For example if a phone number is of n digits, we need O($m \times 10^n$) space for the DAT, where m is size of a pointer to record.
2. Another problem is an integer in a programming language may not be able to store n digits.

Due to limitations mentioned above, DAT cannot be used practically, but remains the best solution theoritically.

Hashing

Hashing is the solution that can be used in almost all such situations and performs extremely well compared to aforementioned data structures like Arrays, Linked Lists, BBSTs in practice. With hashing we get O(1) *search/insert/delete* time on an average (under reasonable assumptions) and O(n) in worst case.

Hashing is an *improvement* over Direct Access Table. The idea is to use *hash function* that converts a given *phone number/key* to a *smaller number* and uses that small number as index in a table [table is an array] called Hash Table, *a.k.a* **Associative Arrays**.

Hash Function
A *hash function* is any function that can be used to *map data* of *arbitrary size* to *data of fixed size*. The values returned by a hash function are called *hash values*, *hash codes*, *hash sums*, or simply, *hashes*.

One of the primary uses of hash functions is to create a data structure called a Hash Table, *widely used* in computer software for *rapid data lookup*. In simple terms, a hash function *maps a big number/string to a small integer* that can be used as *index* in the hash table.

Hash functions *accelerate table/database lookup* by detecting *duplicated records* in a large file. An example is finding similar stretches in DNA sequences. They are also useful in *cryptography*. A cryptographic hash function allows one to easily verify that some input data maps to a given hash value, but if the input data is *unknown*, it is *deliberately difficult* to *reconstruct it* (or equivalent alternatives) by knowing the stored hash value. This is used for *assuring integrity of transmitted data*, and is the building block for **HMAC**, which provides message authentication.

A good hash function should have following properties:
1. It should be *efficiently computable i.e.*, in O(1) time.
2. It should *uniformly distribute* the keys (each position in the table should be *equally likely* for each key).

To understand the above properties, if we consider a hash function which takes first 3 digits for phone numbers, it will turn out to be a really bad one, as most numbers of the same service provider, will have same digits to begin with. As such, this will give a lot of collisions. A better hash function would be to consider last 3 digits. Please note that this may not be the best hash function for phone numbers. There may be better ways.

Hash Table
In computing, a Hash Table (Hash Map) is a data structure used to implement an *associative array*, a structure that can map *keys to values*. A hash table uses a hash function to transform an index into an array of *buckets/slots*, from which the desired value can be found.

Ideally, the hash function will assign each key to a *unique bucket*, but it is possible that two keys will generate an *identical hash* causing both keys to point to the same bucket. Instead, most hash table designs assume that *hash collisions* (different keys that are assigned by the hash function map to the same bucket) will occur and must be accommodated in some way.

In a *well-dimensioned* hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at constant average cost per operation.

In many situations, hash tables turn out to be *more efficient* than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for *associative arrays*, *database indexing*, *caches*, and *sets*.

Collisions & Collision Resolution Techniques

Since a hash function gets us a *small number* for a *key* which can be an *integer*/*string*, there is a possibility that *two keys* result in *same value*. The situation where a newly inserted key maps to an already *occupied slot* in hash table is called ***collision*** and must be handled using some ***collision resolution technique(s)***.

What are the chances of collisions with large table?

Collisions are *very likely* even if we have a *decently large table* to store keys. An important observation is the *Birthday Paradox*[1]. With only 23 people, the probability that two people have same birthday is 50%. [[1]**More on Birthday Paradox**]

Collisions Resolution

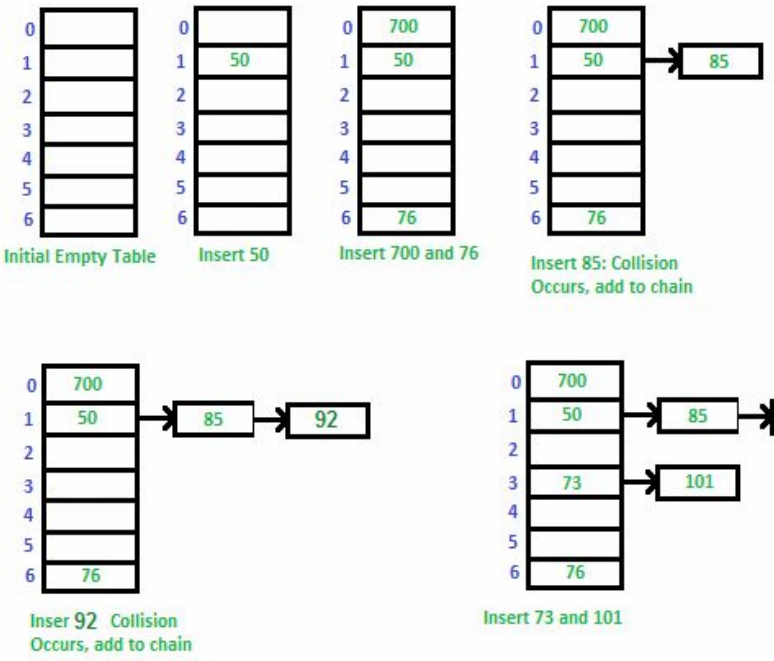
There are mainly two methods to handle collisions:

1. Separate Chaining.
2. Open Addressing.

Separate Chaining

The idea is to make *each cell* of hash table *point to a linked list of records* that have same hash function value.

Let us consider a simple hash function as `hash(key) = key % 7` and sequence of keys as `50, 700, 76, 85, 92, 73, 101`.



Advantages

1. Simple to implement.
2. Hash table never fills up, we can always add more elements to the chain (linked list).
3. Less sensitive to the hash function or *load factors*.
4. It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages

1. Cache performance of chaining is not good, as keys are stored using linked list.
2. Wastage of Space (some parts of the hash table are never used).
3. If the chain becomes long, then the *search time* of a *key* in that chain can become O(*n*) in worst case.
4. Uses extra space for links (since, links are just pointers). [**size of pointers**]

Performance of Separate Chaining

Performance of hashing can be evaluated under the assumption that each *key* is *equally likely* to be hashed to any slot of table (*simple uniform hashing*).

m = *Number of slots* in hash table
n = *Number of keys to be inserted* in hash table
Load factor, α = $\frac{n}{m}$
Expected time to search = O(1 + α)
Expected time to insert/delete = O(1 + α)
Time complexity of search, insert and delete is O(1), if α is O(1).

Open Addressing

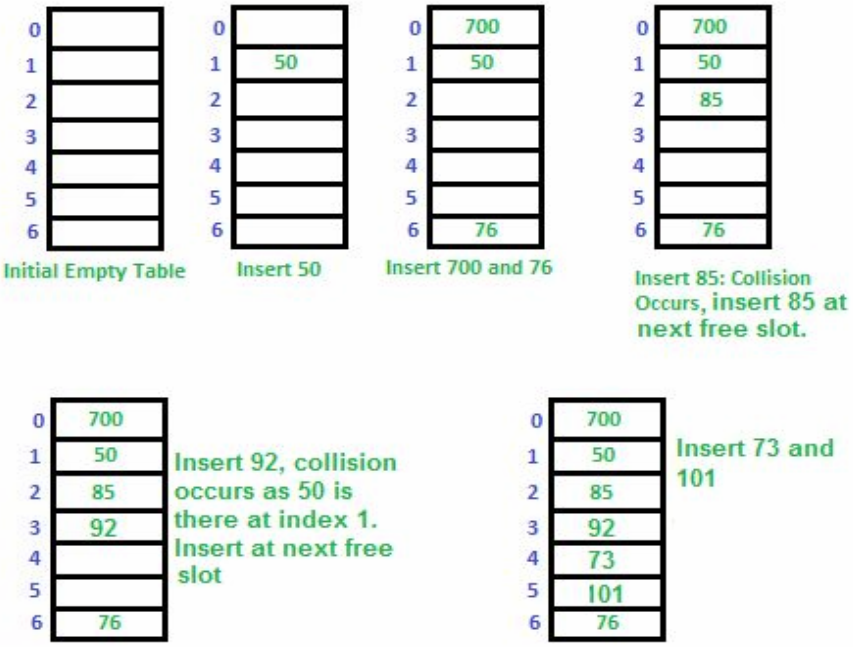
Like separate chaining, *open addressing* is a method for handling collisions. In Open Addressing, all *elements are stored in the hash table itself*. So at any point, *size of table* must be *greater than or equal to total number of keys* (Note that we can increase table size by copying old data into a new table by rehashing, if needed).

Performing Operations

1. `Insert(k)`: Keep probing until an empty slot is found. Once an empty slot is found, insert the `k` into that empty slot.
2. `Search(k)`: Keep probing until *slot's key is equal to* `k` or *an empty slot is reached*.
3. `Delete(k)`: Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked especially, as `deleted`. `Insert(k)` can insert an item in a deleted slot, but `Search(k)` doesn't stop at a deleted slot.

Open Addressing can be done in the following ways:

1. **Linear Probing:**
   In linear probing, we *linearly probe/explore* for the next empty slot. For example, typical gap between two probes is 1 as shown in the example below.
   Example: Let `hash(x)` be the *slot index* in the hash table computed using the hash function, and `S` be the table size. If slot `hash(x)%S` is *full*, then we try `(hash(x)+1)%S`. If `(hash(x)+1)%S` is also *full*, then we try `(hash(x)+2)%S`. If `(hash(x)+2)%S` is also *full*, then we try `(hash(x)+3)%S` and so on.
   Let us consider a simple hash function as `hash(key) = key % 7` and sequence of keys as `50, 700, 76, 85, 92, 73, 101`.



Clustering [Disadvantage]
The main problem with linear probing is clustering, where many consecutive elements form groups and it starts taking time to find a free slot or to search an element, whenever a collision occurs.

2. **Quadratic Probing**
   We look for i^2th slot in i^{th} iteration.
   Example: Let `hash(x)` be the *slot index* computed using hash function. If slot `hash(x)%S` is *full*, then we try `(hash(x) + 1*1)%S`. If `(hash(x) + 1*1)%S` is also *full*, then we try `(hash(x) + 2*2)%S`. If `(hash(x) + 2*2)%S` is also *full*, then we try `(hash(x) + 3*3)%S` and so on.

3. **Double Hashing**
   We use another hash function `hash2(x)` and look for `i*hash2(x)` slot in i^{th} iteration.
   Example: Let `hash1(x)` be the *slot index* computed using hash function. If slot `hash1(x)%S` is *full*, then we try `(hash1(x) + 1*hash2(x))%S`. If `(hash1(x) + 1*hash2(x))%S` is also *full*, then we try `(hash1(x) + 2*hash2(x))%S`. If `(hash1(x) + 2*hash2(x))%S` is also *full*, then we try `(hash1(x) + 3*hash2(x))%S` and so on.

Comparison of above three methods for hashing using Open Addressing Technique

1. *Linear probing* has the *best cache performance*, but *suffers from clustering*. The main advantage of Linear probing is it's *easy to compute*.
2. *Quadratic probing* lies *between the two* in terms of cache performance and clustering.
3. *Double hashing* has *poor cache performance* but *no clustering*. Double hashing *requires more computation time* as *two hash functions* needs to be *computed*.

Performance of Open Addressing

Like chaining, performance of open addressing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing).

m = *Number of slots* in hash table

n = *Number of keys to be inserted* in the hash table

Load factor $\alpha = \frac{n}{m}$ (< 1)

Expected time to *search*/*insert*/*delete* $< \frac{1}{1-\alpha}$

So *search*, *insert* and *delete* takes $\frac{1}{1-\alpha}$ time.

Open Addressing vs Separate Chaining

Advantages of Chaining

1. Chaining is *simple* to implement.
2. In chaining, hash table *never fills up*, we can always *add more elements* to chain. In open addressing, table may become *full*.
3. Chaining is *less sensitive* to the hash function or load factors.
4. Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.
5. Open addressing requires *extra care* to avoid *clustering* and load factor.

Advantages of Open Addressing

1. Cache performance of chaining isn't good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
2. Wastage of Space (some parts of hash table in chaining are never used). In Open addressing, a slot can be used even if an input doesn’t map to it.
3. Chaining uses extra space for links.

Implementation Details for C++ & Java

Usage Details of C++ Map

In an ordered map the elements are sorted by the key. Insert and access is in O($\log_2(n)$). Usually the **STL**[*] internally uses **Red-black trees** for ordered maps. But this is just an implementation detail. In an unordered map, insert and access is in O(1). It is just another name for a Hash Table. In the same manner, C++ also has set and unordered_set.
[[*]**Advanced details on STL** & **Video Playlist on STL**]

Libraries to Include

```
#include<map>
#include<unordered_map>
```

Declaration

```
unordered_map<int, int> m; //declares an empty unordered map (Hashtable)
map<int, int> m;           //declares an empty ordered_map (BBST)
```

Functions

- **Inserting a <key, value> pair:**
```
m.insert(key, value);
```
- **Search for a key:**
```
if(m.find(k) != m.end())
      return m[k];
```
- **Size (number of elements) in the map:**
```
m.size();
```
- **Erase a key:**
```
m.erase(k);
```

For more details, please refer to the following:

1. **Documentation on unordered_map<> in C++**
2. **Documentation on map<> in C++**
3. **Documentation on unordered_set<> in C++**
4. **Documentation on set<> in C++**
5. **Video link on Associative Containers (map<> and set<>)**
6. **Video link on Unordered Containers (unordered_map<> and unordered_set<>)**

Usage Details for Java
HashMap provides constant-time performance [O(1)] for the search, insert, and delete.
TreeMap provides guaranteed O($\log_2(n)$) time cost for all the operations.

Libraries to Import
```
import java.util.Map;
import java.util.TreeMap;
```

Declaration
```
/* declares an empty hashmap */
HashMap<Integer, Integer> m = new HashMap<Integer, Integer>();

/* declares an empty treemap */
TreeMap<Integer, Integer> m = new TreeMap<Integer, Integer>();
```

Functions
- **Inserting a `<key, value>` pair:** `m.put(key, value);`
- **Search for a `key`:**
  - `m.containsKey(k); //tells whether the key 'k' is present`
  - `m.get(k);  //returns null if the key 'k' is not present`
- **Size (number of elements) of the map:** `m.size();`
- **Remove `key` from the map:** `m.remove(k);`

For more details, please refer to the following:
1. **Official Documentation for `AbstractMap`**
2. **Official Documentation for `HashMap`**
3. **Official Documentation for `TreeMap`**
4. **Official Documentation for `ConcurrentHashMap`**
5. **Official Documentation for `AbstractSet`**
6. **Official Documentation for `HashSet`**
7. **Official Documentation for `TreeSet`**
8. **Video Playlist link on Collections Framework in Java (`HashMap`, `TreeMap`, etc)**

Problems & Solutions

1. Length of the *largest subarray* with elements that can be arranged in a continuous sequence. (elements may be repeated)
   Solution: Insert elements in a hash table for each subarray(*i*..*j*) and maintain `min/max` as well. If for a subarray from `[i,j]`, `max-min == (j-i+1)` and `sizeOf(hash table) == j-i+1`, this means `[i,j]` is a *valid* subarray. As a small *optimization step*, we should `break` from the inner loop of subarrays as soon as we find a *duplicate element*.

2. Given an array of integers, find the length of the *longest subsequence* such that elements in the subsequence are *consecutive* integers, the consecutive numbers can be in *any order*.
   Solution:
   a. Sort and find longest consecutive set of elements: $O(n \times \log_2(n))$
   b. Insert all the elements in hash table. In the hash table, for any element `x`, if it's the *starting element of a consecutive series* (hash table shouldn't have `x-1`), find the longest sequence starting from `x` using the hash table. It means simply *search for* `x+1`, `x+2` and so on, until your search `returns` `false`.

3. Check whether an array `A[]` is subset of another array `B[]`.
   Solution: Create hash table for `B[]` and check if it has all the elements of `A[]`.

4. Check if two given sets are disjoint.
   Solution: Create hash table for 1 set and check if it has any element of the 2nd set. If there are no elements in common, then `return true`.

5. Find four elements `a`, `b`, `c` and `d` in an array such that `a+b = c+d` (all elements are *distinct* in the array).
   Solution: Iterate on all possible *pair sums* and maintain a hashset with sum as `key`. While iterating for all the pairs of the array, first search if the pair sum already exists in the hashset. If *found*, `return true`, otherwise *insert* the current pair sum in the hashset.

6. Given an array of strings, return all groups of strings that are *anagrams*.
   Solution: Create hashmap with sorted form of `string as key` and `value as list` of `actual strings`.

7. Count distinct elements in every window of size ***k***
   Solution: Create hashmap for the 1st window of size ***k***, with `key` as the *array element* and `value` as the *frequency* of that element. When you move to the next window, remove the element (decrease the frequency, erase the element if the frequency of that element becomes 0) which is left out and insert the new element (increase the frequency if already present, otherwise, insert with frequency of 1) in the hashmap. At every step, size of the hashmap is the count of distinct elements in that window.

8. Given an array of `0`'s and `1`'s, find the length of the largest subarray with equal number of `0`'s and `1`'s.
   Solution: Convert `0`'s to `-1`'s and calculate `Prefix Sum in P[]`.
   Now, if `P[i] == P[j]`, that means subarray `P[i+1,j]` has equal number of `0`'s and `1`'s. Now use hashmap to find `min/max` index for every unique value in `P[]`.

Important References

1. **Detailed MIT material on Hashing & Chaining**
2. **Detailed MIT material on Open Addressing**
3. **Problems on Hashing, from Hackerearth**
4. **Video Lecture on Hashing with Chaining**
5. **Video Lecture on Hashing with Open Addressing**
6. **Video Lecture on Hash Functions used in Cryptography (Advanced Hashing)**