Hashing

Topics

1. Introduction (Hash Tables, Hash Functions, Complexity)
2. Collision Resolution
3. Implementation Details for C++ and Java
4. Problems

Introduction

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want to perfrom 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. Array of phone numbers and records - Sorted/Unsorted
2. Linked List of phone numbers and records - Sorted/Unsorted
3. Balanced binary search tree with phone numbers as keys
4. Direct Access Table

For arrays and linked lists, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

With balanced binary search tree, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in O(logn) time.

Another solution that one can think of is to use a Direct Access Table where we make a big array and use phone numbers as index in the array. An entry in array is NIL 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 extra space required is huge. For example if a phone number is n digits, we need O(m * 10^n) space for the DCT, 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 above limitations Direct Access Table cannot always be used.

Hashing

Hashing is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get O(1) search/inseet/delete time on 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 or any other key to a smaller number and uses the small number as index in a table called Hash Table.

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 or string to a small integer that can be used as index in hash table.

Hash functions accelerate table or 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 HMACs, which provide message authentication.

A good hash function should have following properties:

1. It should be efficiently computable.
2. It should uniformly distribute the keys (Each table position equally likely for each key)

For example, for phone numbers a bad hash function is to take first three digits. A better function is consider last three 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 or 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 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

Since a hash function gets us a small number for a key which is a big integer or string, there is 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 handling technique.

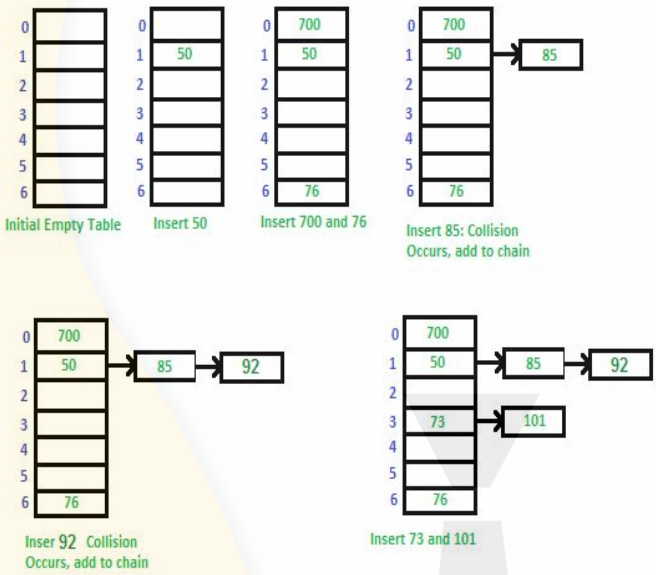
What are the chances of collisions with large table?
Collisions are very likely even if we have big table to store keys. An important observation is Birthday Paradox. With only 23 persons, the probability that two people have same birthday is 50%.

Collisions Handling
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 “key mod 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 chain.
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 hash table are never used)
3. If the chain becomes long, then search time can become O(n) in worst case.
4. Uses extra space for links.

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 $\alpha = n/m$ (keys/slots)
Expected time to search = $O(1 + \alpha)$
Expected time to insert/delete = $O(1 + \alpha)$
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 if needed).

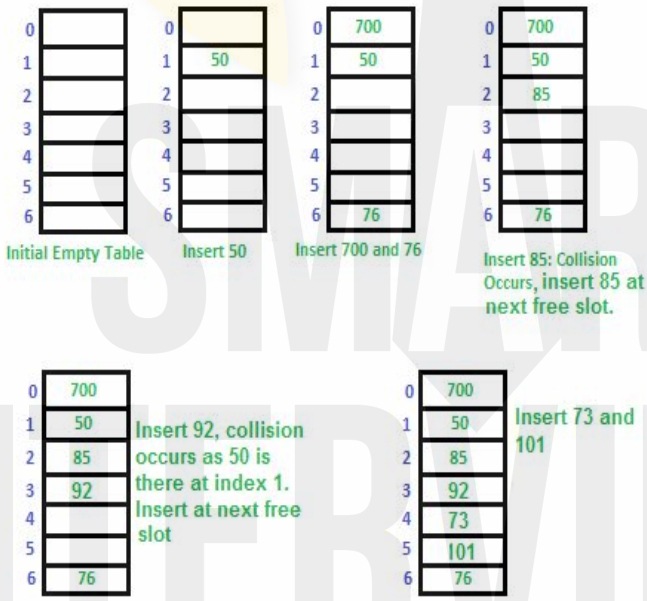
Performing Operations

1. **Insert(k):** Keep probing until an empty slot is found. Once empty slot is found, insert k.
2. **Search(k):** Keep probing until slot's key 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 specially as "deleted". Insert can insert an item in a deleted slot, but search 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 for next slot. For example, typical gap between two probes is 1 as shown in below example as well.

   Example: Let hash(x) be the slot index computed using 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 "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



   Clustering
   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.

2. **Quadratic Probing**
   We look for i^2'th 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 rotation.

   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 adressing 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 has table
Load factor $\alpha = n/m$ (< 1)
Expected time to search/insert/delete $< 1/(1 - \alpha)$

So Search, Insert and Delete take $(1/(1 - \alpha))$ 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 is not 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.

Usage Details for C++ Map

In an ordered map the elements are sorted by the key, insert and access is in O(log 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 hashtable. In the same manner, C++ also has set and unordered_set.

Include

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

Declaration

```
unordered_map<int, int> m; // declares an empty map
map<int, int> m;
```

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. http://www.cplusplus.com/reference/unordered_map/unordered_map/
2. http://www.cplusplus.com/reference/map/map/
3. http://www.cplusplus.com/reference/unordered_set/unordered_set/
4. http://www.cplusplus.com/reference/set/set/

Usage Details for Java

HashMap provides constant-time performance for the basic operation and TreeMap provides guaranteed log(n) time cost for all the operations.

Import
import java.util.Map;
import java.util.TreeMap;

Declaration
HashMap<Integer, Integer> m = new HashMap<Integer, Integer>(); // declares an empty map.
TreeMap<Integer, Integer> m = new TreeMap<Integer, Integer>(); // declares an empty map.

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();
- **Erase from the map:** m.remove(K);

For more details, please refer to the following:

1. http://docs.oracle.com/javase/7/docs/api/java/util/AbstractMap.html
2. http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html
3. http://docs.oracle.com/javase/6/docs/api/java/util/TreeMap.html
4. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentHashMap.html
5. http://docs.oracle.com/javase/6/docs/api/java/util/AbstractSet.html
6. http://docs.oracle.com/javase/6/docs/api/java/util/HashSet.html
7. http://docs.oracle.com/javase/6/docs/api/java/util/TreeSet.html

Problems

1. Length of the largest sub-array with elements that can be arranged in a continuous sequence. (elements may be repeated)
   Solution: Insert elements in a hash set for each sub-array(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 sub-sequence 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(nlogn)
   b. Insert all the elements in hash set. In the hash set, for any element x, if it's the *starting element of a consecutive series*(hash set 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 set for B and check if it has all the elements of A.

4. Check if two given sets are disjoint.
   Solution: Create hash set for 1 set and check if it has any element of the 2nd set.

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 possibe 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 hash map 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 hash map 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 if frequency becomes 0) which is left out and insert the new element(incresae the frequency if already present, otherwise insert with frequency of 1) in the hash map. At every step, size of the hash map is the count of distinct elements in that window.

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

References

1. http://courses.csail.mit.edu/6.006/fall11/lectures/lecture10.pdf
2. http://courses.csail.mit.edu/6.006/fall09/lecture_notes/lecture05.pdf
3. https://www.hackerearth.com/challenge/competitive/code-monk-hashing/problems/