Com S 311: Hashing

September 13, 2019

1 Hashing and Hash Tables

Let S be a set. A *hash function on* S is a function h that maps elements of S to Natural numbers. Given a set S a function $h : S \to \{0, 1, \cdots, T-1\}$ is called a *perfect hash function on* S if for every $x \neq y \in S$, $h(x) \neq h(y)$. Suppose h is a perfect hash function for a set S. We can create a *hash table* of size T for S as follows. Create an array A of size T, initially each cell of the array contains NULL value. To add x into the hash table, compute $h(x)$ and place x at $A[h(x)]$. To remove an element x, compute $h(x)$ and set $A[h(x)]$ to NULL. Finally, to search whether an element y belongs to S or not check if $A[h(y)]$ equals y. Note that the time taken to perform each of these operations if $O(\text{Time taken to compute } h)$. Here is an example: let $S = \{1, 3, 7, 4\}$ and let $h(x) = 3x + 2\%5$. If we store S in a hash table T of size 5, then 1 goes into $T[0]$, 3 goes into $T[1]$, 7 goes into $T[3]$, and 4 goes into $T[4]$.

For this hashing scheme to work, it is critical that h must be a perfect hash function on S. For example, for the above hash function let $S = \{1, 6, 3, 7, 4\}$. Now $h(1) = h(6) = 0$. Thus we should place both 1 and 6 in $T[0]$. We say that there is a *collision* at 1 and 6. In general, for a given a set S, a hash function h and an element x, the *collision set of* h *at* x *with respect to* S is

$$C_h(x) = \{y \in S \mid x \neq y, h(x) = y\}$$

Note that a hash function h is perfect on S if and only if the set $C_h(x) = \emptyset$ for every $x \in S$.

Alternate way to capture collisions is

$$C_h = \{\langle x, y\rangle \mid x \neq y \in S\}$$

This set captures all colliding pairs.

In general, for a given S it is not feasible to find perfect hash functions and we have to deal with collisions. Given a hash function h and a set S, we use *chain hashing* to deal with collosions.

Let S be a set and h be a hash function whose range is $\{0, \cdots m-1\}$. Let T be a table of size m. Initially, each cell of $T[i]$ points to NULL. We add elements to T as follows:

```
Procedure: Add(x);

    compute h(x)
    Add x to the list pointed by T[h(x)]
```

Now to search for an element q: we first compute $h(q)$, and search for q in the list pointed by $T[h(q)]$. To remove an element q from T: we compute $h(q)$ and remove q from the list pointed by $T[h(q)]$.

What is the time taken by each of add, search, or remove? Let us consider search. Say we are searching for q in T. Time taken by this process is bounded by: Time taken to compute $h(q)$ + time taken to search for q in the list pointed by $T[h(q)]$. Equivalently time taken is $O($ time to compute h(x) + size of the set $C_h(q))$. If the size of the list/$C_h(q)$ is large, then this time is high. Otherwise the time is low. While building hash tables a challenge is to pick a hash function h such that the size of the list stored at index i of the hash table is small (ideally constant) for every i.

Suppose T be a hash table of size m that is storing elements from a set S. We introduce a few notions.

Maximum Load of T is the maximum length of lists at $T[0], T[1], \cdots, T[m-1]$. *Average Load* of T is

$$\frac{\sum_{0\leq i\leq m-1, T[i]\neq NULL} \text{Size of list at } T[i]}{\text{Number of Non-Null cells in T}}$$

Finally, we define *load factor* of T as the ratio between numbers of elements added to T and size of T.

Note that the worst-case time perform any of search/add/remove operations is bounded by time taken to compute the hash function plus maximum load. The average/expected time to perform search/add/remove is $O(\text{Time taken to Compute hash function + average load})$. While building hash tables, a goal is pick a hash functions such that the average load is a constant.

1.1 Hash Functions used in Practice

There are two classes of hash functions that are used in practice: *deterministic* and *random*. Random hash functions work as follows. Suppose that S is a set of positive integers and we wish to store S in a hash table of size m. Pick the first prime number p that is at least m. Instead of storing S in a table of size m, we will store in a table of size p. Now, randomly pick $a, b \in \{0, 1, \cdots, p-1\}$. The hash function is defined as $h(x) = (ax + b)\%p$. Thus in addition to the hash table, one needs to store a and b which are two integers. If S is a set of Strings, then we can first convert each string $t \in S$ into an integer by using `hashCode` method of java and then apply the above hash function on the hashcode.

Examples of deterministic hash functions are `hashCode` in Java, FNV, Murmer, Jenkins etc. They work by "exploiting randomness" that is present in the data.

Java uses following hash function (hashCode) for String. Let $x = c_1 \cdots c_m$ be a m-character String. Fix α.

$$h(x) = c_m + c_{m-1}\alpha + c_{m-2}\alpha^2 + \cdots + c_2\alpha^{m-2} + c_1\alpha^{m-1}$$

Java takes 31 for α. Here we view each character c_m as an integer. This can be easily done by converting the ASCII representation of a character to an integer.

Hash Tables in Java. Java creates and maintains hash tables *dynamically*. Java always ensures that the size of the hash table is a power of 2. Let m denote the current hash table size. Java uses a combines hash code with a secondary hash function g: The secondary hash function works as follows: Given an int x, the value of $g(x)$ is the value returned by the following code.

```
h = x^ ^ (x >>>16);

return h
```

Given an object x, then the $h(x) = g(x.hashCode())\%m$, where m is the current size of the hash table. When the load factor of the hash table approaches 0.7, then Java will double the hash table size and re-hashes the elements to the new hash table.

2 Applications of Hashing

Consider the following problem: Given two integer arrays A and B (let us assume that both of them are size n). Compute the set of elements that appear in both A and B. A naive algorithm is the following:

```
For i in the range 1 to n {
   x = A[i];
   Search for x in the array B
}
```

If we use linear search to search for x in the array B, then the time taken to search is $O(n)$, and thus the total time taken by the algorithm is $O(n^2)$. However, we could sort the array B, and use binary search to search for x in the array B. The time taken to sort B is $O(n \log n)$ and binary search takes $O(\log n)$. Thus the time taken by the following algorithm is $O(n \log n)$.

```
Sort $B$.
For i in the range 1 to n {
   x = A[i];
   Binary Search for x in the array B
}
```

We can further reduce the time using hash tables. Build a hash table for B and search for x in the hash table. Since the (expected/average) time to search in hash tables is $O(1)$, the time taken by the algorithm is $O(n)$.

```
Create a hash table T for elements in $B$.
For i in the range 1 to n {
   x = A[i];
  Search for x in the hash table T
}
```

Here is another problem: Given an array A of integers find the longest sub-array of A whose elements sum to 0. For example if A is $[4, 3, -7, 8, 1, 5, 7, -1, -5, -3, -2, -1, 18]$. It has two sub arrays that sum to 0: $[4, 3, -7]$ and $[1, 5, 7, -1, -5, -3, -2, -1]$ and the second is the longer subarray. A naive algorithm for this problem is the following:

```
longest = 0;
for i in the range 1 to n
  for j in the range i to n
     find the sum of elements a[i], a[i+1], ... a[j]
     if the sum equals 0, then  longest  = max{longest, j-i+1]
```

It can be seen that this algorithm takes $O(n^3)$ time.

We can arrive at a more efficient algorithm. Let us try to calculate the prefix sums. I.e., let us create an array P where $P[i] = \sum_{j=1}^{i} A[i]$ (it is assumed the array is indexed from 1). This $P[i]$ is the sum of the first i elements of the array. Suppose that $P[3] = 25$ and $P[8] = 25$, then it must be the case that sum of the elements $A[4], A[5], A[6], A[7]$ and $A[8]$ must be zero. This suggests the following algorithm.

```
longest = 0;
left = 0; right  = 0;
P[1] = A[1];
for i in the range 2 to n
  P[i] = P[i-1] + A[i];
  Search for P[i] among P[1], P[2], ...P[i-1];
  Let j be the smallest index at which P[i] appears.
  if (j-i) > longest {
    longest = i-j
    left = j;
    right = i;
  }

  Return the sub array A[left], A[left+1] ...A[right]
```

What is the time taken to search for $P[i]$ among $P[1], P[2], \cdots P[i-1]$? Naively, this can be done in $O(i)$ time and this leads to $O(n^2)$ time algorithm. We can store $P[i]$'s in a hash table and attempt to reduce the time for search to $O(1)$. However, we need to be little careful with the add procedure (as we would like to find the smallest index at which $P[i]$ appears). We create a hash table T consisting of key-value pairs. When would like to add a pair $\langle k, v \rangle$ to the table, we compute $h(k) = x$. And search at $T[x]$ is there is a tuple whose key is k. If such a tuple exists, then we do not add $\langle k, v \rangle$ to the table. This ensures that (expected) time for search is $O(1)$. Thus the time taken by the algorithm is $O(1)$.