Hashing

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Data Structures (CS 223)

1 The Dictionary Problem

Given a set $\mathcal{X}$ of n integers from $\{0, 1, 2, \ldots, U-1\}$, the problem is to design a data structure using which we can quickly verify whether $\mathcal{X}$ contains an input integer *key* or not. Thus, for the set $\mathcal{X} = \{5, 9, 19, 1, 24, 27, 32\}$, the data structure must report *true* for $key = 19$ (or for any $key \in \mathcal{X}$) and *false* for $key = 200$ (or for any $key \notin \mathcal{X}$).

Of course the data structure has to be fast, as without such a restriction, we can simply scan through $\mathcal{X}$ and answer the query in $O(n)$ time. Additionally, the data structure should ideally support insertion of new numbers and deletion of existing numbers. Thus, we seek for a data structure over $\mathcal{X}$ that supports the following operations:

- $search(key)$: detect if *key* is present in $\mathcal{X}$ or not
- $insert(x)$: insert x into $\mathcal{X}$
- $delete(x)$: delete x from $\mathcal{X}$

We have already seen binary search trees can support all these operations. In fact, binary search trees discussed in class can be improved to support these operations in worst case $O(\log n)$ time.

Can we do even better? Using hashing, we can support these operations in $O(1)$ time. However, we will look at an extremely simplified form.

2 The Naive Solution

Let us look at the most basic approach. We create a table $A[U]$. (Implementation wise, A can be thought of as an array.) Now, we set $A[x] = 1$ if $x \in \mathcal{X}$ and $A[x] = 0$, otherwise. Thus, we can easily support the following operations in $O(1)$ time as follows:

- $search(key)$: by checking the value of $A[key]$
- $insert(x)$: by setting $A[x] = 1$
- $delete(x)$: by setting $A[x] = 0$

Refer to Figure 1. The set $\mathcal{X}$ contains 9 and $A[9] = 1$, whereas $\mathcal{X}$ does not contain 4 and $A[4] = 0$. To insert 10, we simply set $A[10] = 1$ and to delete 14, we simply set $A[14] = 0$.

Unfortunately, although very simplistic, the solution has a major disadvantage – it occupies too much space. Suppose $U \gg n$, i.e., the maximum value in $\mathcal{X}$ is much higher than the size of $\mathcal{X}$. Then to store a small number of values, we create a gigantic array.

0	0	0	1	0	1	0	1	0	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Figure 1: A naive hash table for the set $\mathcal{X} = \{14, 3, 9, 11, 5, 7\}$ where the universe is $\{0, 1, 2, \ldots, 14\}$ of size $U = 15$

3 A Space-Efficient Solution

To alleviate the space issue, we take the following two-pronged approach:

- reduce the size of table A to some upper bound, say $TABLE\_SIZE$. Initialize each cell of the array to -1.
- use a hash function, say $h(x) = x\%TABLE\_SIZE$, where % denotes the modulo operator

3.1 Simple Scenario

Suppose there does not exist two integers x and y in $\mathcal{X}$, which satisfy $h(x) = h(y)$. Under this assumption, we can can slightly modify our technique:

$$\text{set } A[h(x)] = x \text{ if } x \in \mathcal{X} \text{ and } A[h(x)] = -1, \text{ otherwise}$$

Under this scenario, we can again support the above operations in $O(1)$ time as follows:

- $search(key)$: return *true* if $A[key] >= 0$ else return $false$
- $insert(x)$: set $A[h(x)] = x$
- $delete(x)$: set $A[h(x)] = -1$

So if $TABLE\_SIZE$ is not too large, we have a good solution.

Refer to Figure 2. The set $\mathcal{X}$ contains 14 and $A[14\%10] = A[4] = 14$, whereas $\mathcal{X}$ does not contain 26 and $A[26\%10] = A[6] = -1$. To insert 10, we simply set $A[10\%10] = A[0] = 10$ and to delete 14, we simply set $A[14\%10] = A[4] = -1$.

-1	11	-1	3	14	15	-1	7	-1	9
0	1	2	3	4	5	6	7	8	9

Figure 2: A hash table for the set $\mathcal{X} = \{14, 3, 9, 11, 15, 7\}$ where $TABLE\_SIZE = 10$ and hash function $h(x) = x\%TABLE\_SIZE$. Note that 14, 11, and 15 get hashed to positions 4, 1, and 5, because $14\%10 = 4$, $11\%10 = 1$, $15\%10 = 5$.

3.2 Collision

Unfortunately, the assumption above is rather too stringent. So, let us remove it.

Immediately we run into a problem – if two numbers x and y are such that $h(x) = h(y)$, then they *collide* into the same position in the table A. There is now no way of telling whether $\mathcal{X}$ contains x, or y, or both, since we can keep only value in the array cell.

One of the standard ways of removing these collisions is called *separate chaining*.

3.3 Separate Chaining

The idea is to modify the structure of table A. Instead of $A[i]$ being a standard array cell, we let it be a linked list. (Implementation wise, we can imagine A to be an array of linked lists.)

Now, we can support the operations as follows:

- $search(key)$: go to the linked list $A[h(key)$ and scan it to detect if it contains key.
- $insert(x)$: first $search(x)$ and make sure that x is not already present. If not present, add x at the end of the linked list $A[h(x)]$.
- $delete(x)$: scan the linked list $A[h(x)]$ and remove x if present. To implement this, you can search the linked list to find the position that contains x, and then do a deleteAfter on the node at the previous position.

Refer to Figure 3. The set $\mathcal{X}$ contains 14 and the linked-list $A[14\%10] = A[4]$ contains 14, whereas $\mathcal{X}$ does not contain 26 and $A[26\%10] = A[6]$ does not contain 6. To insert 10, we simply add 10 to the linked list $A[10\%10] = A[0]$ and to delete 14, we simply delete 14 from the linked list $A[14\%10] = A[4]$.

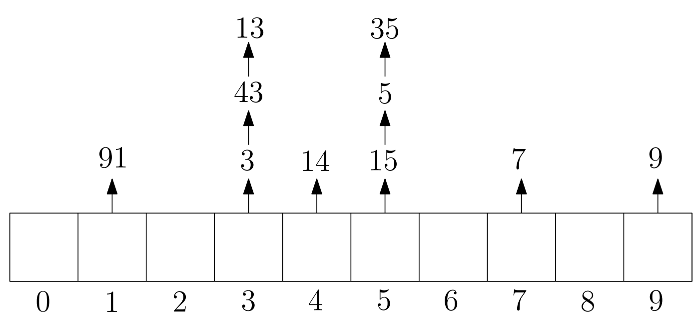

Figure 3: A hash table with separate for the set $\mathcal{X} = \{14, 3, 9, 15, 7, 5, 35, 43, 13, 91\}$ where $TABLE\_SIZE = 10$ and hash function $h(x) = x\%TABLE\_SIZE$. Note that 13, 43, and 3 get hashed to the linked list at index 3, because $13\%10 = 3$, $43\%10 = 3$, $3\%10 = 3$.

Of course, our $O(1)$ time complexity is no longer guaranteed. Because, if we are really unlucky a large number of values may get hashed into the same linked list. However, with good choice of hash functions and if $TABLE\_SIZE$ is a large prime number close to n, then collisions are usually not too high and the data structure is fast.

3.4 Which hash function is better?

Usually, given a set $\mathcal{X}$ and two hash functions, the one which reduces the number of collisions is a better choice. The number of collisions can be computed by subtracting the number of non-empty linked lists from n.