A Lecture on Hashing

Aram-Alexandre Pooladian, Alexander Iannantuono

March 22, 2017

This is the scribing of a lecture given by Luc Devroye on the 17th of March 2017 for Honours Algorithms and Data Structures (COMP 252). The subject was hashing and their uses.

Hashing

Definition 1. A **hash function**, $h(\cdot)$, is a map from the space of keys, $\mathcal{K}$, to another space, $\mathcal{M} = \{0, 1, \ldots, m-1\}$ (we note that the cardinality of $\mathcal{M}$, $|\mathcal{M}|$, is m). Hence, we write $h : \mathcal{K} \to \mathcal{M}$, and note that the mapping usually appears to be random and uniformly distributed.

Example 2. Consider a hash function that takes the last two digits of a phone number. Hence, this would map a phone number, say $514.844.1395 \mapsto 95$. Using this hash function, we have that $m = 100$, and thus $M = \{0, 1, \ldots, 99\}$.

There are different methods of using hash functions in data structures that permit dictionary operations. We looked into the following three methods during this lecture:

1. Direct Addressing
2. Hashing with Chaining
3. Open Addressing

Direct Addressing

For a key k, we have $h(k) = k$. The table that the keys are mapped to is defined as $T = T[0], T[1], \ldots, T[m-1]$. This table is represented in the computer as an array, and is typically implemented as an exogenous data structure and has pointers that go to the necessary data (see figure 1). For a data item x, its key is denoted by $key[x]$.

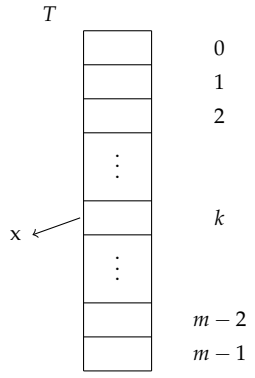

Figure 1: Example of a direct hash table

Operations - Simple

The following dictionary operations work under the assumption that all keys are unique. They all require $O(1)$ time.

INSERT(x, T)

1 $T[h(key[x])] = x$

DELETE(x, T)

1 $T[h(key[x])] = \text{NIL}$

SEARCH(k, T)

1 **if** $T[k] == \text{NIL}$
2 **return** "Not Found"
3 **else**
4 **return** $T[k]$

Avoiding Initialization

A possible issue with implementing the hash table in this specific way occurs when $|\mathcal{K}| << |\mathcal{M}|$. Then we would need to initialize a table that would be very large — arguably larger than how much memory available at our disposal. However, suppose this was possible. We would also be 'wasting' storage space because we would have slots in our table that never get used! Lastly, this could would take $O(m)$ since we initialize every slot to contain NIL. We need to do better. A possible solution would be to simply identify end points (our smallest and largest values in $\mathcal{M}$) and consider everything between these points to be our table. However, this could result in our table T having "garbage" values. It takes $\Theta(m)$ time to delete the "garbage". To avoid initialization, we need a second table, $T^\star$ such that $|T| = |T^\star|$, and a stack S. As we shall see below, with this method, we are able to identify "garbage" by the clever use of forward and backward pointers. The stack S has positions indexed by 1 to $Last[S]$. $T^\star[k]$ points to a place in S where k is a stored key, and $S[i] = k$ as a check. We rewrite the basic dictionary operations:

Operations - Avoiding Initialization

INSERT(x, T)

1 $Last[S] += 1$ // A new element is added in the stack
2 $k = key[x]$ // Note: the keys are still unique
3 $T[k] = x$ // Pointer to x
4 $T^\star[k] = Last[S]$
5 $S[Last[S]] = k$ // Acts as a "back-pointer"

DELETE(x, T)

1 $k = key[x]$
2 $T[k] =$ NIL // Step not required
3 $p = T^\star[k]$ // Want to delete p
4 $q = Last[S]$
5 $Last[S] -= 1$ // Decrement the size of the stack
6 $T^\star[q] = p$
7 $S[p] = q$

SEARCH(k, T)

1 **if** $T[k] \leq Last[S]$ and $S[T^\star[k]] == k$
2 **return** $t[k]$
3 **else**
4 **return** "Not Found"

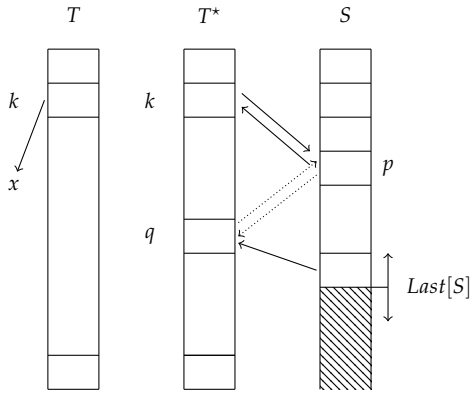

Figure 2: Process of avoiding initialization. The dotted line illustrates the $DELETE$ operation.

Hashing with Chaining

As before, we have our table T of size m but now instead of containing the pointers to our data, it contains pointers to linked lists that contain the data, i.e., $T[k]$ is a pointer to the head of the linked list. All elements x in that list have keys that hash to k — that is, $h(key[x]) = k$. This is to say that the linked lists are somewhat 'nested' *inside* the array. We will see that this is a convenient way to avoid collisions. That being said, before we continue, we must first formally define a collision.

Definition 3. A **collision** is said to occur when for a given hash function $h(k)$, $\exists k_1, k_2 \in K$ such that $k_1 \neq k_2$ and $h(k_1) = h(k_2)$[1].

[1] In other words, $h(k)$ is not injective.

Operations

INSERT(x, T)

1 $k = h(key[x])$
2 $LinkedListINSERT(x, T[k])$

DELETE(x, T)

1 $k = h(key[x])$
2 $LinkedListDELETE(x, T[k])$

SEARCH(k, T)

1 $k^\star = h(key[x])$
2 **if** $k^\star$ is there
3 **return** $ListSEARCH(k, T[k^\star])$

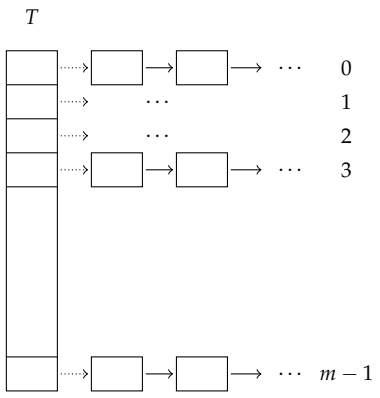

Figure 3: Process of Hashing with chaining. Dotted lines indicate pointers to head of linked list.

Expected Time for Unsuccessful Search

Let $\mathbb{T}_U$ be the expected time for an unsuccessful search. Assume that each key gets mapped uniformly at random to one of the m chains. Let $\mathbb{E}_S$ be the expected list size which is n/m, n being the number of items and m being $|T|$. This ratio is known as the **load factor** and is denoted by α. We have

$$\mathbb{T}_U = 1 + \mathbb{E}_S = 1 + \frac{n}{m} = 1 + \alpha,$$

where the $+1$ is due to overhead.

We also have that the total space, $\mathbb{T}_S$, used is n cells + m headers or,

$$\mathbb{T}_S = n + m = n\left(1 + \frac{m}{n}\right) = n\left(1 + \frac{1}{\alpha}\right).$$

We then notice a trade-off between $\mathbb{T}_U$ and $\mathbb{T}_S$, as illustrated in the figure. It is recommended that $\alpha \simeq 1$ in order to have optimal speed when performing operations.

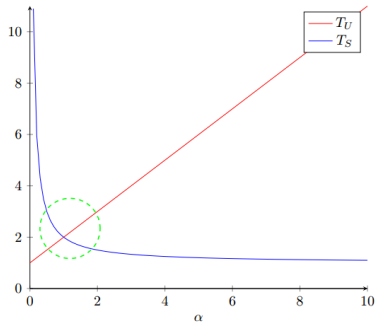

Figure 4: The dotted green circle outlines the desired sweet-spot for the load factor

Discussion of Result

If we keep α fixed, then the expected time of an $INSERT, SEARCH$ or $DELETE$ operation is $O(1)$, regardless of how large n is. This makes hashing a formidable competitor for binary search trees.

Dynamic Hashing

In practice, if we consider the load factor as a function of time, i.e., as dictionary operations are performed on the hash table, we would like it to remain within reasonable bounds. To keep it within bounds, we check if the current load factor is within our bounds before performing the operations $INSERT$ or $DELETE$ and if it is beyond the bound, we do what is known as **rehashing** — a process in which the hash table size is changed to keep the load factor near a target value α and then hashing every element in our table to place it into the new table. Consider the following example:

Example 4. We set our upper and lower bounds to $\frac{1}{2}$ and 2 respectively. Thus, $\alpha \in (\frac{1}{2}, 2)$. If α goes below or reaches $\frac{1}{2}$, we make a new table T' of size $m/2$ and iterate through all of our elements in the table T. For every element, we insert them into T'. We can then relabel T to be T'. On the other hand, if α goes above or reaches 2, we rehash to a table T' of size $2m$. This assures that α remains close to 1. Rehashing takes $O(n)$ time but its amortized cost is $O(1)$.

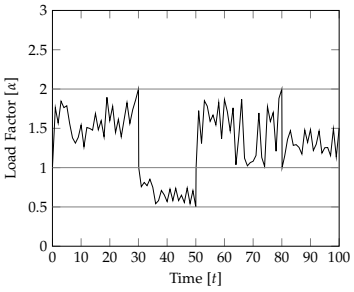

Figure 5: Load factor as a function of time with dynamic hashing.

Radix Sort

Radix sort can be regarded as a version of hashing with chaining. We demonstrate how radix sort works for integers, given in a decimal base. As an example, take

$$319 \;\; 329 \;\; 707 \;\; 729 \;\; 321 \;\; 327 \;\; 322.$$

We first create ten linked lists (so $|\mathcal{M}| = 10$) and perform what is called "bucketing" using the following hash function

$$h(x) = x \mod 10,$$

where x is simply the integer we want to sort. We place them in their respective "buckets" as follows:

0	1	2	3	4	5	6	7	8	9
	321	322					707		319
							327		329
									729

Then we concatenate the linked lists from left to right

$$321 \;\; 322 \;\; 707 \;\; 327 \;\; 319 \;\; 329 \;\; 729,$$

and repeat the process with the following function

$$h(x) = (x \div 10) \mod 10.$$

Their buckets are

0	1	2	3	4	5	6	7	8	9
707	319	321							
		322							
		327							
		329							
		729							

Doing a third and final round would place them all in order. We note that this procedure takes time $n \cdot \#$ of rounds , so in this small example, the time and space are indeed $O(n)$.

Assume now that we wish to sort n numbers from the space $\{1, 2, \ldots, n^{17}\}$ in $O(n)$ time and space (in RAM). The number of rounds would be $\log_{10} n^{17} = 17 \log_{10} n$, leading to time complexity $\Theta(n \log_{10} n)$, which is not good! A solution is to change the base from 10 to something else ... like n. Then any number in the range of $[0, n^{17})$ can be written as follows

$$x = x_0 + x_1 \cdot n^1 + x_2 \cdot n^2 + \cdots + x_{16} n^{16},$$

with the coefficients being:

$$x_0 = x \mod n, x_1 = \frac{x - x_0}{n} \mod n, x_2 = \frac{x - x_0 - x_1 \cdot n}{n^2} \mod n,$$

and so on. Writing the integers in this fashion and using radix sort gives us hash tables of size n and time complexity $O(n)$, since only 17 rounds are required.

Open Addressing

This method does not require any linked lists but it does need that the number of elements n be less than or equal to $|T| = m$. The idea behind open addressing is to store a key in the first available slot. We start by checking if $T[k]$ is free to be filled and if not, we generate some new index every time we find a full slot until we find an empty one. An interesting difference in open addressing, in comparison to the previous methods we saw, is that a key's location in the table is affected by the order in which the keys were inserted. To consider examples of hashing functions, we must first define the concept of a probe sequence.

Definition 5. A **probe sequence** is a sequence of indices of the slots checked in attempt to insert a key into the table T.

We use the notation $h(k,0), h(k,1), \ldots, h(k,m-1)$ for the probe sequence, where it is understood that this must form a permutation of $0, 1, \ldots, m-1$.

```
INSERT(x, T)
1  k = key[x]
2  j = 0 // Counts the number of attempts
3  while j < m and T[h(k,j)] ≠ NIL
4       j+ = 1
5       if j = m // Used all possible options
6            return "Table is Full"
7       else
8            T[h(k,j)] = x
```

```
SEARCH(k, T)
1  j = 0
2  while j < m and T[h(k,j)] ≠ NIL
3       if key[T[h(k,j)]] = k
4            return T[h(k,j)]
5       j = j + 1
6  return "Not Found"
```

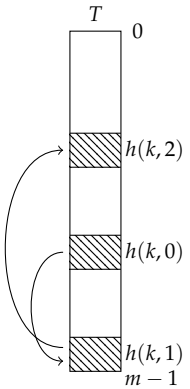

Figure 6: Probe sequence, illustrated. The hatched lines indicate filled slots.

Speed Comparison with Chaining Method

Recall that we have a table of size m with $n \leq m$ items stored. Assume that each item is inserted using an independent, uniform random permutation for $h(k,0),\ldots,h(k,m-1)$ (this is unrealistic but nevertheless insightful). Let us look at the operation *INSERT* in order to compare with the chaining method. We have

$$\mathbb{P}[\text{Finding an empty space in 1st attempt}] = (m-n)/m = 1-\alpha,$$

and thus,

$$\mathbb{P}[\text{Finding an occupied space in 1st attempt}] = 1-(1-\alpha) = \alpha.$$

Clearly, $\mathbb{P}[\text{Finding an empty space in 2nd attempt}]$ is

$$\mathbb{P}[O_2] = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right) \leq \left(\frac{n}{m}\right)^2 = \alpha^2,$$

and, in general, the $\mathbb{P}[\text{Finding an empty space in } k\text{th attempt}]$

$$\mathbb{P}[O_k] = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-k}{m-k}\right) \leq \left(\frac{n}{m}\right)^k = \alpha^k.$$

Thus we can say that the expected time to insert, $\mathbb{E}[T_{INSERT}]$, is

$$\begin{aligned}
\mathbb{E}[T_{INSERT}] &= 1\cdot\mathbb{P}[O_1] + 2\cdot\mathbb{P}[O_2] + 3\cdot\mathbb{P}[O_3] + \cdots \\
&= \underbrace{\mathbb{P}[O_1] + \mathbb{P}[O_2] + \mathbb{P}[O_3] + \cdots}_{=1} \\
&\qquad \underbrace{+\mathbb{P}[O_2] + \mathbb{P}[O_3] + \cdots}_{\leq\alpha^1} \\
&\qquad\qquad \underbrace{+\mathbb{P}[O_3] + \cdots}_{\leq\alpha^2} \\
&\cdots \\
&\leq \sum_{j=0}^{\infty} \alpha^j = \frac{1}{1-\alpha}.
\end{aligned}$$

This is worse than chaining (since $1+\alpha \leq 1/(1-\alpha)$), but can get close when α is small. It is recommended to keep α small preferably less than 0.5. If we maintain that, then, just like chaining, we have a formidable competition for binary search trees in dictionary applications.

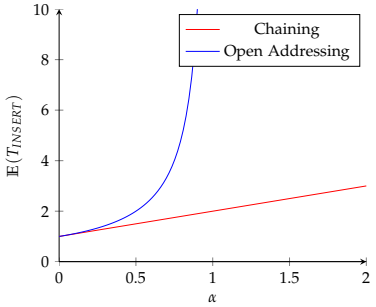

Figure 7: The Open Addressing asymptotically approach infinity as $\alpha \to 1$ whereas chaining does not have this behavior

Examples of Hashing Functions

Linear Probing

Linear Probing is a method of probing whose probe sequence is generated by the equations below:

$$h(k, j) = h(k) + j \pmod{m}$$

or

$$h(k, j) = h(k) + c \cdot j \pmod{m} \qquad \text{where } gcd(c, m) = 1.$$

Remark 6. Linear probing, while easy to implement, has a flaw in it known as **primary clustering**. Clustering begins to reveal itself as the slots in the table T fill up, thus negatively affecting the search and insert times.

Random Probing

Random Probing is another method of probing whose probe sequence is generated by the equations below:

$$h(k, j) = h(k) + d_j \pmod{m}$$

where d_j is a sequence with $j \in [0, m-1]$ that forms a permutation of $\{0, 1, \ldots, m-1\}$ that appears to be, or is close to a, truly random uniform permutation. An example is the **linear congruential sequence**: $d_0 = 0$ and $d_{i+1} = (a \cdot d_i + 1) \mod m$, where a is an integer. It is well known that $d_0, \ldots, d_{m-1}$ is a permutation of $\mathcal{M}$ if and only if $a - 1$ is a multiple of every prime that divides m, where "4" is considered a prime. If m is prime, then all values of a are good. If $m = 100$, then a must be $1, 4, 41, 61$ or 81.

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. 2009. Cambridge, MA.

[2] Kaylee Kutschera, Pavel Kondratyev, and Ralph Sarkis. *A Lecture on Cartesian Trees*. 2017. Montreal, QC. `http://luc.devroye.org/Kutschera-Kondratyev-Sarkis-CartesianTreesLectureNotes-McGillUniversity-2017.pdf`