Hashing

Rayane Laabid, Lauren Zhou

February 25, 2025

This is an augmented transcript of a lecture given by Luc Devroye on the 25th of February 2025 for the Honours Algorithms and Data Structures class (COMP 252, McGill University). The subject covered was hashing.

Hashing

Hashing is a technique which allows us to store a piece of data, x, in a table T. The main goal is to optimize the time it takes to retrieve that data, ideally achieving a time complexity of $O(1)$. To facilitate this, we employ a hashing function $h(\cdot)$ that maps the key related to our data, denoted by $\text{key}[x]$, to an index in the table.

Definition 1. A **hashing function**, $h(\cdot)$: $K \to M$, is a map from a universe of keys, K, to a set of indices $M = \{0, 1, \ldots, m-1\}$.

Example 2. Consider a hash function $h(\cdot)$ that maps a phone number to its last two digits, e.g.,

$$h(562\ \ 562\ \ 363) = 63.$$

The associated index set M is given by $\{0, 1, 2, \ldots, 99\}$.

Hashing has applications such as:

- ADT Dictionary: search, insert, and delete functions.
- Sorting algorithms: Radix sort, bucket sort (we will cover both).
- Rabin–Karp string matching.
- File signatures, i.e., associating $h(x)$ with a file x.

We implement hashing in dictionary data structures using the following techniques:

1. Direct addressing.
2. Hashing with chaining.
3. Open addressing.

If we consider that $h(\cdot)$ can be computed in one-time unit, then the objective is to ensure that all standard dictionary operations can be carried out in $O(1)$ time, making it a formidable competition for search trees.

Direct addressing

Consider a set of data X such that all keys key$[x]$ take values in $M = \{1, ..., m\}$. Let T be a table, implemented as an array, of capacity m. A **direct address** hashing function is the identity function

$$h(\text{key}[x]) = \text{key}[x].$$

We store x, considered as a pointer to a cell, in the table T at the position given by key$[x]$, i.e., $T[\text{key}[x]] = x$ (see Figure 1).

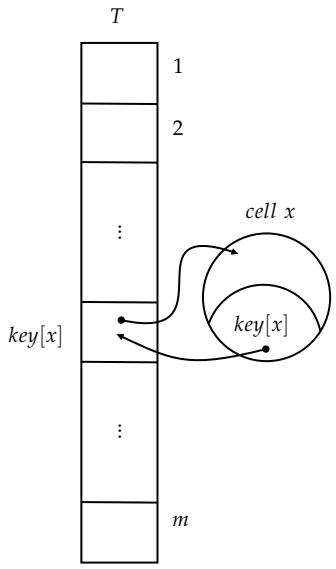


Figure 1: Example of direct addressing

Insert, delete, and search

Using direct addressing, we can implement the following natural algorithms for table management:

INSERT(x, T) :

```
1  T[key[x]] = x
```

DELETE(x, T):

```
1  T[key[x]] = NIL
```

SEARCH(k, T):

```
1  if T[k] == NIL then
2        return NotFound
3  else
4        return T[k]
```

Potential issues

Given n number of data points, a potential issue with implementing a hash table this way comes up when $n \ll m$. In this case, we would waste memory, since many slots would remain unused. In addition, initializing such a large table would take $\Theta(m)$ time, as each slot would need to be set to NIL.

Assume next that we do not initialize the table T, taking it "as is". Then, we use a second table, T^*, along with a stack, S, where $|T^*| = |T|$. Each position in the stack is indexed from 1 to Last$[S]$, and $T^*[k]$ points to a spot in the stack, say i. If $S[i] = k$, and $i \leq$ Last$[S]$, then we are sure that the key at position k in T is legitimate. With the use of forward and backward pointers, we can easily identify "garbage" values, avoiding unnecessary initialization and clean-up. All operations are in $O(1)$ in the RAM model.

Insert, delete, and search with shadow table

INSERT(x):

```
1  LAST[S] = LAST[S] + 1 // new element to add to the stack
2  k = key[x]
3  T[k] = x
4  T*[k] = LAST[S]
5  // shadow table at index k points at the last element of the stack
6  S[LAST[S]] = k // last element of stack points to key
```

DELETE(x):

```
1  k = key[x]
2  T[k] = NIL
3  p = T*[k] // the index where k is placed in the stack
4  q = S[LAST[S]] // we switch places with the key at the last key in the stack
5  LAST[S] = LAST[S] − 1
6  if q ≠ k then
7        T*[q] = p
8        S[p] = q
```

SEARCH(k) :

```
1  // Search for key k
2  if S[T*[k]] == k and T*[k] ≤ LAST[S] then
3        return T[k]
4  else
5        return NotFound
```

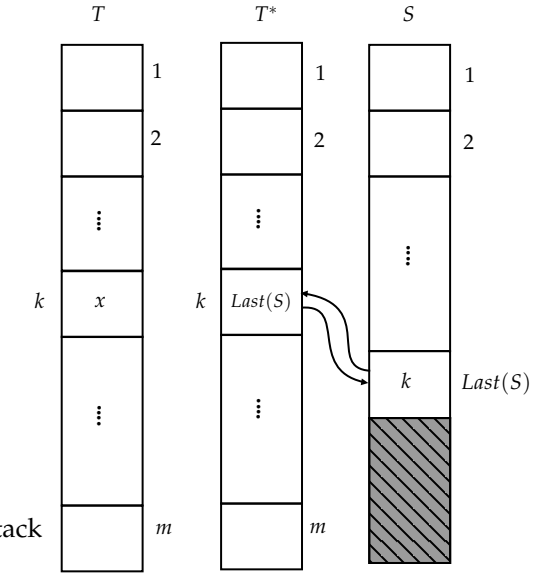


Figure 2: Example of the tables after insert operation

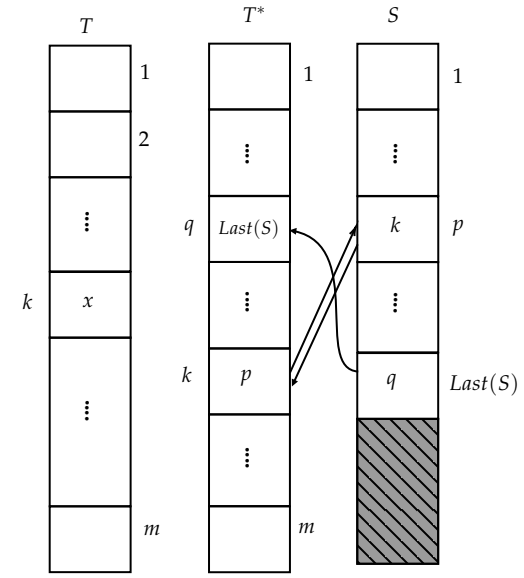


Figure 3: Example of the state of the table before the delete operation.

Hashing with chaining

In direct hashing, we assumed that all of our keys are unique. In particular, we assumed that the hashing map h never mapped two distinct keys to the same integer i. What if this were not the case?

Definition 3. For a hashing function h: $K \to M$, a **collision** between keys k_1 and k_2 occurs when $h(k_1) = h(k_2)$. We say that h has collisions if there exists such a pair $k_1, k_2 \in K$.

A well-established approach to handling collisions in a hash table is to utilize linked lists. In this method, rather than each entry in the table T of size m storing direct pointers to data, each entry $T[k]$ instead stores a pointer to the head of a linked list. Each element x within the linked list at $T[k]$ has a key that hashes to k, meaning $h(\text{key}[x]) = k$. Consequently, the hash table effectively maintains separate chains of elements within linked lists at each index, ensuring that multiple

elements mapping to the same hash value can be efficiently stored and retrieved.

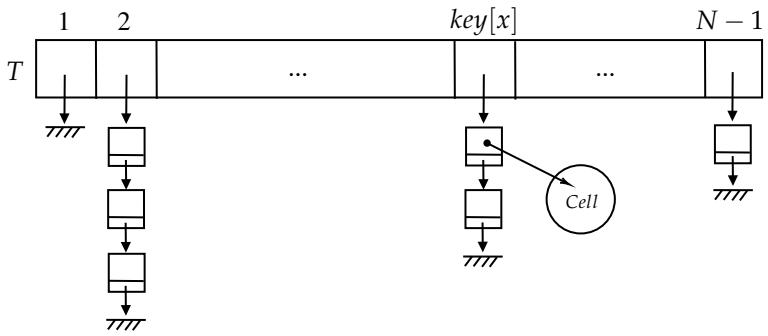


Figure 4: Example of a table for a dictionary

Insert, delete, search 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(x, T) :

1 $k = h(key[x])$
2 **if** $T[k] \neq$ NIL **then**
3 **return** $LinkedListSearch(x, T[k])$

Analysis

Let our table T be of size $|T| = m$ and assume we have n data points stored in the table, and assume furthermore that they are independent and uniformly distributed over T.

Let $\mathbb{E}_S$ the average size of a linked list:

$$\mathbb{E}_S = \frac{n}{m} = \alpha.$$

We call $\alpha = \frac{n}{m}$ the **load factor**. Let $\mathbb{T}_U$ denote the expected time of an unsuccessful search. Thus

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

where $+1$ comes from accessing the table at the specific hashed key index.

Finally let $\mathbb{T}_S$ be the space used (n data points + m headers) per data element:

$$\mathbb{T}_S = \frac{n+m}{n} = 1 + \frac{m}{n} = 1 + \frac{1}{\alpha}.$$

Figure 5 shows a trade-off between $\mathbb{T}_S$ and $\mathbb{T}_U$ where the optimal speed/performance balance is reached when $\alpha \approx 1$. In general, it is wise to keep α near 1.

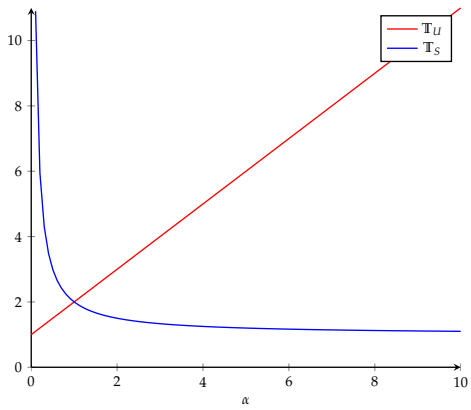


Figure 5: The intersection represents the desired sweet spot for the load factor.

Discussion of result

If we keep α fixed, then the expected time of *insert* , *search* or *delete* operations is $O(1)$. This makes hashing an excellent data structure for memory storage and quick access to data.

Exercise 4. Show that the expected time to search for one of the n data elements is $1 + \frac{\alpha}{2} - \frac{1}{2n}$, assuming that all n data items are equally likely to be searched for.

Open addressing

This method does not require any linked lists but it does require that the number of elements n less than or equal to the size of the table $|T| = m$. The idea behind open addressing is to store a key in the first available slot. We start by checking if $T[k]$ is empty and, if not, we generate some new index every time we find a full slot until we find an empty one. Note that the location of a key 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 hashed values of the same key $k = key[x]$ denoted $h(0,k), h(1,k), \ldots, h(m-1,k)$ that corresponds to a permutation for the slots in Table T, namely a permutation of the set $\{0, 1, 2, \ldots, m-1\}$.

The idea would be to go through this permutation and assign $T[h(i,k)] = x$ for the first empty slot found.

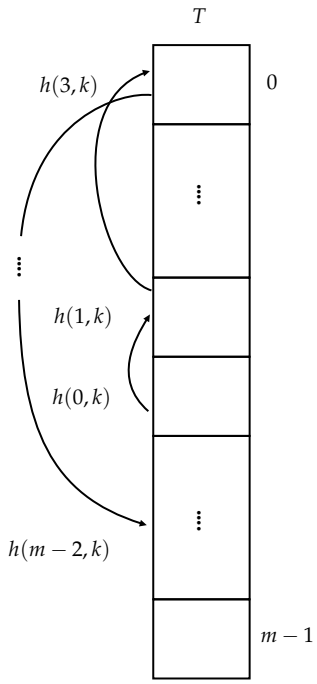


Figure 6: Probe sequence, illustrated.

Operations

INSERT(x, T) :

```
1  k = key[x]
2  set j = 0 // jump counter
3  while j < m and T[h(j,k)] ≠ NIL :
4        j = j + 1
5  if j == m then
6        return 'Table_Full'
7  else
8        T[h(j,k)] ← x
```

SEARCH(k, T) :

```
1  // returns the position in the table if found
2  set j = 0
3  while j < m and T[h(j,k)] ≠ NIL and key[T[h(j,k)]] ≠ k
4        j = j + 1
5  if j == m then
6        return 'Not_Found'
7  else if T[h(j,k)] = NIL then
8        return 'Not_Found'
9  else
10       return T[h(j,k)] // points to a cell
```

Examples of hash functions

Linear probing
Linear probing uses the probe sequence generated by:

$$h(k, j) \ = \ \Big(h(k) \ + \ j \ \Big) \ mod(m).$$

or

$$h(k, j) \ = \ \Big(h(k) \ + \ cj \Big) \ \ mod(m),$$

where h is a given hash function and $c \ \geq \ 1$ is an integer with $gcd(c, m) = 1$. Often, $c = 1$.

Remark:
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 uses the probe sequence generated by:

$$h(k,j) \;=\; \Big(h(k) \;+\; d_j\Big)\; mod(m).$$

where $(d_0, ..., d_{m-1})$ is a permutation of $\{0, 1, \ldots, m-1\}$ that 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 $d_0, d_1, \ldots, d_{m-1}$ is a permutation of $\{0, 1, ..., m-1\}$ if and only if $a - 1$ is a multiple of every prime that divides m where "4" is considered a prime. For example, if $m \;=\; 100 \;=\; 5^2 \cdot 4$ then $a - 1$ must be a multiple of 20. Thus the possible values for a are $\{1, 21, 41, 61, 81\}$.

Double probing
Let h^* and h^{**} be two hash functions, and assume that m is prime. Then, one can use the probe sequence

$$h(i,k) = \Big(h^*(k) + ih^{**}(k)\Big)\; mod(m),\; 0 \le i \le m-1.$$

Analysis and speed comparison with chaining method

Consider a table of m elements storing n data points using an open addressing method. Assume that our resulting probe sequence $h(0,k), h(1,k), \ldots, h(m-1,k)$ are *i.i.d.*, i.e., independent and identically distributed for each key $k = key[x]$. Let us examine the operation *Insert* in order to compare it with the chaining method. We have

$$\mathbb{P}[\textit{Finding an empty space in 1st attempt}] = \frac{m-n}{m} = 1 - \alpha.$$

and thus,

$$\mathbb{P}[\textit{Need at least one attempt}] = 1 - (1-\alpha) = \alpha.$$

Defining

$$\mathbb{P}[O_i] = P[\textit{Need at least i attempts}],$$

We have

$$\mathbb{P}[O_i] = \left(\frac{n}{m}\right) \cdot \left(\frac{n-1}{m-1}\right) \cdot \left(\frac{n-2}{m-2}\right) \ldots \left(\frac{n-i+1}{m-i+1}\right) \le \left(\frac{n}{m}\right)^i = \alpha^i.$$

The expected number of attempts for an unsuccessful search is

$$\sum_{i=1}^{n} \mathbb{P}[O_i] \le \; \alpha^1 + \alpha^2 + \cdots + \alpha^n \le \sum_{i=1}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

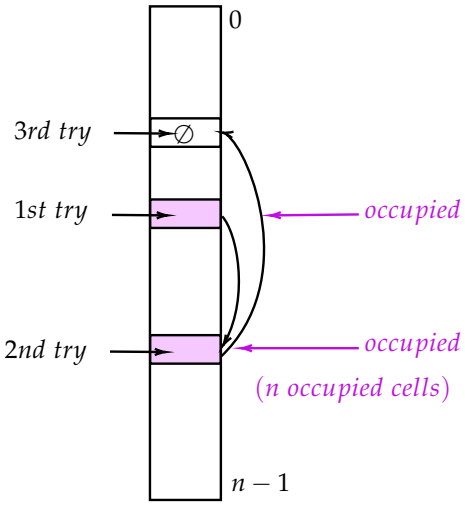


Figure 7: Example

This is worse than chaining (since $1 + \alpha \leq \frac{1}{1-\alpha}$) but can get close when α is small enough, say less than 0.5. If we maintain that then just like chaining, we would have a good and efficient data structure able to compete with binary search trees.

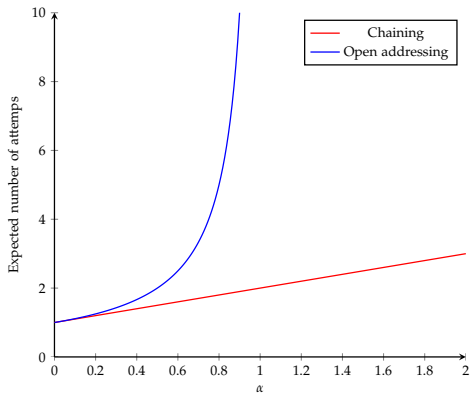


Figure 8: The open addressing graph will asymptotically approach infinity as $\alpha \to 1$, unlike the chaining graph.

Dynamic hashing, rehashing

In open addressing and hashing with chaining, the load factor plays an important role in the access time of the operation (insert, delete, search), thus keeping it near a target value is essential for the efficiency of the data structure. Before performing an operation such as insert or delete, one can monitor the load factor. If α is no longer within a given range (e.g., $[a, b]$), we **re-hash**, which involves creating a new table of an appropriate size to bring the load factor α back within range. We then transfer all keys from the old table to the new one.

Example 6. Let us try to keep $\alpha \in (\frac{1}{2}, 2)$. If α goes below or reaches $\frac{1}{2}$, then make a new table T' of size $\frac{m}{2}$ and rehash all elements to T' using a new hash function. When α reaches 2, we create a table T' of size $2m$ and proceed the same way. This method ensures that α remains in $[\frac{1}{2}, 2]$.

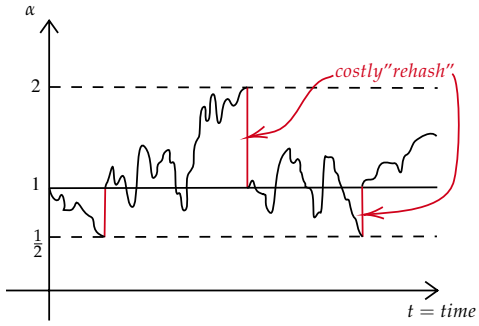


Figure 9: Load factor as a function of time with dynamic hashing.

Analysis of the time complexity

In figure 10, t_k represents the time at which rehashing occurs, and Δ_k denotes **the time elapsed** between two consecutive rehashing events, specifically between t_{k-1} and t_k. Assume that the table contains n data items at time t_{k-1}. We continue the setting of example 5.

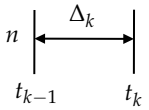
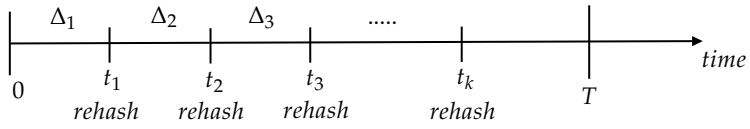


Figure 10: Example of time elapsed between two rehashing instances.



We know that the cost of rehashing at moment t_k is at most $\leq$ to $2n$ since at the rehashing stage we can at most double the size of the table T.

We also know that between two rehashing instances we need to at least access the table $\frac{n}{2}$ times. Thus we can conclude that $\Delta_k \geq \frac{n}{2}$. Thus, the cost of rehashing at time t_k is at most $2n \leq 4\Delta_k$.

The total cost of rehashing between zero and T is:

$$\sum_k \text{cost of rehashing at } t_k \leq 4 \sum_k \Delta_k = 4T.$$

We conclude from our analysis that the total cost of rehashing between the first and last rehashing instance is always $\leq$ four times the total time elapsed between the first and last rehashing instances. In other words, it is linear in the number of operations.

Radix sort

Radix sort is a sorting algorithm that implements hashing with chaining. Let us illustrate the algorithm through an example.

Consider the following list of numbers consisting of three decimal digits:

$$320,\ 219,\ 118,\ 380,\ 220, 109,\ 707,\ 415,\ 655,\ 315.$$

We construct a table with $|M| = 10$ buckets, where each bucket contains a linked list. As a result, the table consists of a total of ten linked lists. We then proceed by applying the following hash function:

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

We insert each number into its corresponding bucket. This results in the following table:

0	1	2	3	4	5	6	7	8	9
320					415		707	118	219
380					655				109
220					315				

We then reorganize the array by emptying the buckets from left to right:

$$320,\ 380,\ 220,\ 415,\ 655,\ 315,\ 707,\ 118,\ 219,\ 109.$$

Next, we apply the process again with a new hash function, the second digit from the right, preserving the order in which the elements were stored as we move from left to right and top to bottom across the table:

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

Example:

$$h(320) = (320\ div\ 10)\ mod(10) = 32\ mod(10)\ = 2.$$

We then store these values in the same table:

0	1	2	3	4	5	6	7	8	9
707	415	320		655				380	
109	315	220							
	118								
	219								

We regroup the array again by the same method:

$$707,\ 109,\ 415,\ 315,\ 118,\ 219,\ 320,\ 220,\ 655,\ 380.$$

and for the last time use the following hash function and store the result into the table:

$$h(x) = (x\ div\ 10^2)\ mod(10).$$

We thus obtain:

0	1	2	3	4	5	6	7	8	9
	109	219	315	415		655	707		
	118	220	320						
			380						

By regrouping one last time we obtain our sorted list.

$$109,\ 118,\ 219,\ 220,\ 315,\ 320,\ 380,\ 415,\ 655,\ 707.$$

Note that this procedure runs in time equal to n times the number of rounds, which in this case is the number of digits in the numbers. Consider the case where we take n numbers from the set $\{1, 2, \ldots, n^{20}\}$. The worst-case number of rounds in this scenario would be $\log_{10}(n^{20}) = 20 \cdot \log_{10}(n)$. A good alternative would be to change the base from 10 to n and rewrite any number in $\{1, 2, \ldots, n^{20}\}$ in the form of $x = x_0 + x_1 \cdot n + \cdots + x_{20} \cdot n^{20}$. We then obtain a hash table with n buckets of linked lists and it only takes 20 rounds to sort the list.

Bucket sort

Consider a monotone hashing function $h(\cdot)$ such that $x \leq y$ implies $h(x) \leq h(y)$. Then assign each element to its corresponding bucket in the table T for hashing with chaining, and finally sort each linked list in all the linked lists (also called buckets) and regroup.

Example 7. Assemble the following array of size 10:

$$[0.78,\ 0.17,\ 0.39,\ 0.26,\ 0.72,\ 0.94,\ 0.21,\ 0.12,\ 0.23,\ 0.68]$$

with the following hash function :

$$h(x) = \lfloor 10 \cdot x \rfloor.$$

Finally consider a hash table T with size 10 then we can assign each value to its corresponding bucket:

0	1	2	3	4	5	6	7	8	9
	0.17	0.26	0.39			0.68	0.78		0.94
	0.12	0.21					0.72		
		0.23							

Sorting each linked list in each bucket we thus obtain:

0	1	2	3	4	5	6	7	8	9
	0.12	0.21	0.39			0.68	0.72		0.94
	0.17	0.23					0.78		
		0.26							

We finally end by regrouping each linked List from each bucket from left to right to form our final solution:

$$[0.12,\ 0.17,\ 0.21,\ 0.23,\ 0.26,\ 0.39,\ 0.68,\ 0.72,\ 0.78,\ 0.94].$$

Property
One of the interesting properties of bucket sorting is the following. If the hash function h distributes the points randomly and uniformly, then:

$$\mathbb{E}[Time] = O(n) \quad if\ n = m.$$

Bloom Filters

Context and goal: Given a data structure D_n holding n items labeled $(x_1, ..., x_n)$ that all live in a space X, the goal is to quickly check whether a new $x \in X$ is not in D_n.

Definition 8. Consider the following defined **design parameters**:

- $\varepsilon > 0$: An upper bound on the probability of making a false positive error, i.e., declaring $x \in D_n$ when it is not.
- k: The number of hash functions. The hash functions are denoted as $h_1, ..., h_k$, mapping $X \to \{1, ..., N\}$.
- N: The size of the Bloom filter, measured as a number of bits.

1	0	0	1	0	1	1	1	0	0	0	0	0	1	0
⋮	1	0	1	1	1	0	0	0	0	0	1	0	1	0
k	0	0	0	1	1	0	0	0	0	1	1	1	1	1
	1	2					...							N

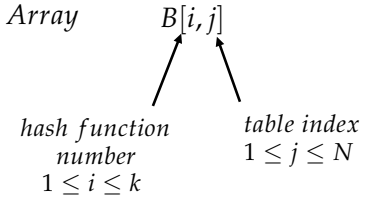


Figure 11: For the Bloom filter, we maintain a $k \times n$ matrix B containing bits. $B[i, j]$ refers to the i-th hash function and j-th data entry.

Operations

Consider the following operations defined for Bloom filters:

- *Makenull* (initialize):

```
MAKENULL(B):
1  for i = 1 to k
2      for j = 1 to N
3          B[i, j] ← 0
```

- Add x to B:

```
ADD(x, B):
1  for i = 1 to k
2      B[i, h_i(x)] ← 1
```

- Check if x is in D_n: We have the following cases, if

$$\prod_{i=1}^{k} B[i, h_i(x)] = \begin{cases} 0 & \text{then declare "}x \notin D_n\text{" (this is always correct).} \\ 1 & \text{then declare "}x \in D_n\text{" (there could be some false positives).} \end{cases}$$

Probability of a false positive

We assume that all values $h_i(x_j)$, $1 \leq i \leq k$, $1 \leq j \leq n$, are independent and uniformly distributed on $\{1, ..., N\}$.

If x is a new item not in D_n, then the probability of getting a false positive is:

$$\begin{aligned} P\{\text{false positive}\} &= P\left\{ \bigcap_{i=1}^{k} B[i, h_i(x)] = 1] \right\} \\ &= \left(1 - \left(1 - \frac{1}{N}\right)^n\right)^k. \end{aligned}$$

Let us now pick $k = \lceil \log_2 \frac{1}{\varepsilon} \rceil$, and $N = \lceil \frac{n}{\ln 2} \rceil$. Then

$$P\{\text{false positive}\} \leq \left(1 - \left(1 - \frac{\ln 2}{n}\right)^n\right)^k \overset{n\to\infty}{\longrightarrow} \frac{1}{2^k} \leq \frac{1}{2^{\log_2 \frac{1}{\varepsilon}}} = \varepsilon.$$

Example and uses

Example 9. Consider the example with $\varepsilon = \frac{1}{128} = \frac{1}{2^7}$, so $k = 7$, and $m = k \times N \leq \frac{7}{\ln 2}(n+1)$ which is around $10n$ bits. The expected time complexity for searching with and without Bloom filters for each operation can be summarized as follows, where $f(n)$ depends upon the data structure D_n.

Operation	Without Bloom filter	With Bloom filter
Search for x in D_n when $x \in D_n$	$f(n)$	$k + f(n)$
Search for x in D_n when $x \notin D_n$	$f(n)$	$k + \varepsilon f(n)$

When ε is small, one can considerably reduce the expected time complexity for searching. In addition, only $O(n)$ bits of extra storage are required.

For large D_n, many applications used Bloom filters to improve performance. For example, in Google Chrome (to identify malicious URLs), in Bing's bit funnel search index, in Bitcoin (for wallet synchronization - now discontinued), in Ethereum (to quickly find logs on the Ethereum blackchain), in database software like BigTable, Apache HBase, Apache Cassandra, ScyllaDB and PostgreSQL, (to reduce disk look-ups for non-existence of rows or columns) and in Akamai (to prevent one-hit wonders, request by users just once, and cache web objects only upon second request).

Exercises

Exercise 10. If $k = 1$, how large should $m = N$ be to have an error probability of at most ε?

Exercise 11. If the table size ($m = k \times N$) is restricted to be at most n for technical reasons, then show that no matter how k is picked, the probability of error is at least $1 - \frac{1}{e} + O(1)$.

References

B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, vol.13(7):p422–426, 1970.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2009.

Aram-Alexandre Pooladian and Alexander Iannantuono. A lecture on hashing, 2017. URL `https://luc.devroye.org/AramPooladian+AlexIannantuono-HashingLectureNotes-McGillUniversity-2017.pdf`.

Wikipedia. Bloom filters. URL `https://en.wikipedia.org/wiki/Bloom_filter`.