

Linear Hashing: Key Aspects and a Running Example

Yannis Chronis and Alex Delis
Univ. of Athens,
15703 Athens, Greece

www.alexdelis.eu/LinearHashing-CD21.pdf

June 2021

Introduction

A hash-table is a data structure that maps **keys** to **values** or **memory locations**. The resulting structure allows for the effective look-up of information/record associated with each key [3]. To retrieve the location where a record is stored in a hash-table, we convert every key into a **hash-value** with the help of a *hash-function* [2]. Every hash-value uniquely identifies a **bucket** in which the stored information/record can be found. A hash-table bucket can accommodate **one or more records**.

The size or number of buckets **n** a hash-table features is selected up front and remains fixed while the structure gets populated. As soon as the **load λ** of the hash-table comes closer to 1 or even exceeds it, the structure has to be evidently re-organized while setting a *larger* size n .

Going Dynamic

In deviating from the fixed-number of buckets approach above, hash-tables can adopt and be implemented as more dynamic structures. This means that the **n** number of buckets can grow (or diminish) according to the needs for storing records. The apparent advantage of this dynamic approach is that we do not need to worry a-priori with the selection of **n** and more importantly, we do not use space for the data structure that goes unused.

To accomplish the above objective, a hash-table has to be capable of increasing the number of its buckets and properly adapt the used hash-function(s).

A first approach to offer dynamicity to hashing is that anytime the load λ points out to an expansion, to **double the size** of the hash-table. This is the main idea behind *extensible hashing* [1]. However, when **n** grows to be of certain (large) size, doubling up what is already in place does represent much wasted space in the structure. An alternative approach that is more **incremental** to its work is that of **linear hashing** [4].

Linear Hashing Overview

Through its design, linear hashing is dynamic and the means for increasing its space is by **adding just one bucket at the time**. Any such incremental space increase in the data structure is facilitated by **splitting** the keys between newly introduced and existing buckets utilizing a *new* hash-function.

The mechanism for the expansion of space is enabled through the use of an **ordered family of hash functions**: $h_0, h_1, h_2, h_3, \dots$. It is important to note that this family of functions should be designed so that the range of function h_{i+1} **should be twice as large** as the range of function h_i .

Linear hashing commences its operation with 3 key parameters:

1. **initial size**: this can be $2^i m$ buckets where i represents the **level** or **round** of hashing with $i = 0, 1, 2, \dots$ and m is a **int** designating initial capacity of the structure in terms of number of buckets.
2. **hash function family**: this can be for example:

$$h_i(k) = k \mod 2^i m$$

where k is the *key* and i represents the *level* of hashing we currently work at. Over time, the hash function changes so that it can deal with new buckets that are introduced. In the context of a round, we have to use the corresponding hash function h_i of the level in question *as well as* that of the next round h_{i+1} . Hence, we can accomplish the splitting of keys between a new and an old bucket.

3. *pointer*: a pointer p designates the *next bucket* to be split up. When the number of buckets doubles, all buckets used in the previous level or round have been splitted and a new round is about to start with p returning to the very first bucket.

The indicator that tells us *when to split up* the bucket pointed by p is the value of the hash-table load factor termed λ . The latter is defined as follows:

$$\lambda = \frac{\text{Num. of Keys in Hash Table}}{\text{Key Capacity in Non - Overflow Buckets}}$$

We should note that linear hashing does occasionally use overflow buckets but in calculating the denominator of λ , we consider only bucket slots that can accommodate keys without taking into account the capacity offered by overflow buckets. The check on whether a bucket has to be split *has to occur immediately after a key (or record) has been successfully inserted* in the hash-table.

Looking Up Keys

In order to find whether a key already exists in the linear hash table, we use the hash function h_i with which the structure currently operates. If the result of h_i is *less than* the value of the current location of p , then we have to use the hash function of the *next round* h_{i+1} to possibly find the sought key/record.

An Example

Let's assume the following:

1. The initial number m of buckets used is $m = 2$.
2. Each bucket can store of up to 2 keys; this is also known as the bucket size.
3. The hash-function family adopted is: $h_i(k) = k \bmod 2^i * m$
4. Initial value of $p = 0$ and if $\lambda > 0.75$ bucket splitting will have to occur.

We wish to enter into the hash table the following keys in sequence: 10, 5, 4, 7, 18, 14, 22, 9, 13, 8, 11.

⇒ **Round: 0** — utilized functions are: $h_0(k) = k \bmod 2$ and $h_1(k) = k \bmod 4$

p → 0		
1		

Figure 1: The hash-table before any key is inserted

As there is no splitting, the keys 10, 5, 4 and 7 are all inserted using function $h_0(k) = k \bmod 2$. The situation changes to what Fig. 2 depicts.

p → 0	10	4
1	5	7

Figure 2: The hash-table after the insertion of keys 10, 5, 4 and 7

As soon as key 7 is inserted the load λ becomes 1 ($4/4$). This is greater than the threshold set 0.75, an indication that a **new bucket** has to be introduced with index number 2 (i.e., bucket #2).

The bucket currently pointed by p will split and its content will be re-distributed with the new bucket #2 using hash function $h_1(k) = k \bmod 4$. Moreover, the pointer p “moves” one bucket over and will now point into bucket #1.

While re-distributing keys 10 and 4, key 10 will be placed in bucket #2 and key 4 will remain in bucket #0. Fig 3 shows how matters have developed so far with $\lambda = 4/6$.

0	4	
$p \rightarrow 1$	5	7
2	10	

Figure 3: The hash-table after rehashing keys in bucket #0

Should we attempt to insert key 18 and use function h_0 we obtain result 0. This is **less than** the current position of p (i.e., 1). Hence, to identify the correct bucket to insert key 18, we have to utilize function h_1 . The latter helps place 18 into bucket #2 as Fig. 4 shows. However, $\lambda = 5/6$ which is now greater than 0.75.

0	4	
$p \rightarrow 1$	5	7
2	10	18

Figure 4: The hash-table after inserting key 18

This calls for the splitting of bucket #1 whose keys will be redistributed between the old bucket #1 and a newly acquired bucket #3. The hash function to be used is $h_1(k)$ and once the rehashing completes p returns back to point to bucket #0 and we enter **round 1** at this point. Fig. 5 shows the new situation with key 5 staying in bucket #1 and key 7 going into new bucket #3. The value of λ is $5/8$ and as the hash-table

0	4	
1	5	
$p \rightarrow 2$	10	18
3	7	

Figure 5: Redistribution of keys 5 and 7

has double in terms of number of buckets (going from 2 to 4), we set p to point to the 0^{th} bucket and we proceed to **Round: 1**.

⇒ **Round: 1** — utilized functions are **now**: $h_1(k) = k \bmod 4$ and $h_2(k) = k \bmod 8$

Using $h_1(k)$ we insert keys 14 and 22. Both yield value 2 and since this is larger than the current value of $p = 0$ both of these keys end up in an **overflow bucket** off existing bucket #2. Fig. 6 shows the insertion of keys 14 and 22. Immediately after the insertion of 14, λ was $6/8$ which is not greater than 0.75 but the insertion of 22 brings $\lambda = 7/8$ rendering a bucket split required. The bucket #0 pointed by p will be

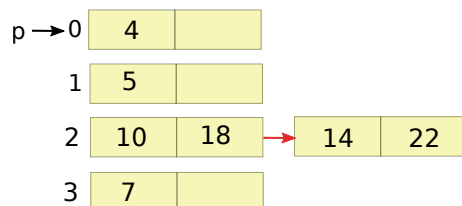


Figure 6: Insertion of 14 and 22 lead to an overflow bucket

split. Subsequently, a new buckets with id #4 is acquired and the sole key 4 has to be redistributed between buckets #0 and #4. This is done with the use of $h_2(k)$ and key 4 now finds itself in bucket #4 whereas bucket #0 remains empty (i.e., Fig. 7). In addition p 's position is advanced to show to bucket #1. The

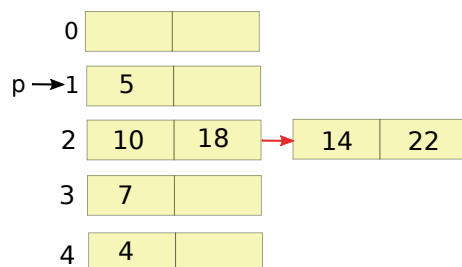


Figure 7: Redistribution of key 4

value of λ becomes $7/12$ which is less than 0.75.

Using $h_1(k)$, we insert key 9 (Fig. 8) and the updated λ is now $8/10$ which is greater than 0.75.

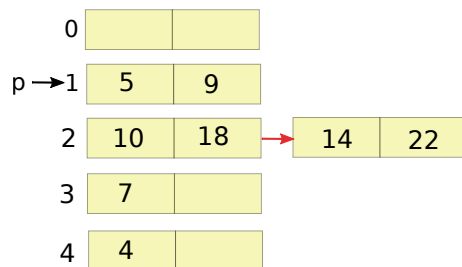


Figure 8: Inserting 9 – $\lambda = 8/10$

The new λ value calls for splitting bucket #1 pointed by p . Function $h_2(k)$ is used to redistribute keys 5 and 9 between buckets #1 and #5 while the position of p is advanced to point bucket #2 (Fig. 9).

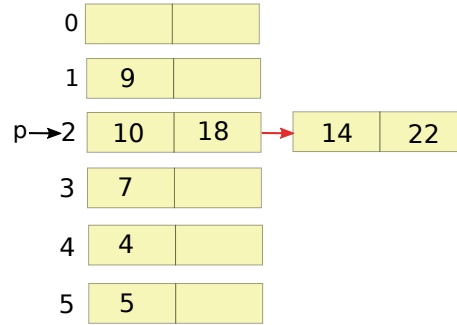


Figure 9: Redistribution of keys 5 and 9

We move on to insert key 13. We attempt with $h_1(k) = 1$ which is **less** than the value of the current position of p and so, we proceed to hash key 13 with $h_2(k) = 5$. Fig. 10 shows the updated content of the hash-table with $\lambda = 9/12$.

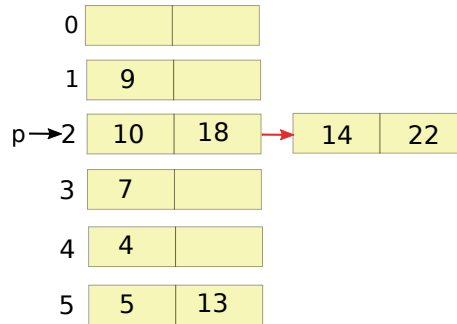


Figure 10: Inserting key 13

Next is the insertion of key 8 with $h_1(k) = 0$ which is smaller than the current position of p and so we proceed to hashing with $h_2(k) = 0$ Fig. 11 shows the updated content of the hash-table with $\lambda = 10/12$. This last value is greater than 0.75 and so, redistribution of all keys pointed by p with a newly acquired

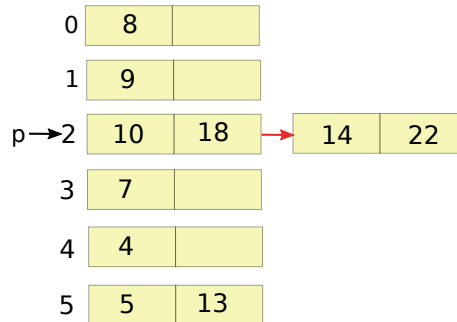


Figure 11: Inserting key 8

bucket #6 has to take place. Fig. 12 presents the resulting hash-table while function $h_2(k)$ is being used; in this realigned hash-table λ is 10/14.

0	8	
1	9	
2	10	18
p→3	7	
4	4	
5	5	13
6	14	22

Figure 12: Redistribution of keys 10, 10, 14, 22

If we are to insert the only remaining key 11 from our sequence of numbers, we initially attempt to hash with $h_1(11) = 3$ where there is space. However, $\lambda = 11/14$ which is greater than 0.75 so a bucket split is required. As we need to split bucket #3, a new bucket #8 is being acquired, and keys 7 and 11 are re-distributed.

0	8	
1	9	
2	10	18
p→3	7	11
4	4	
5	5	13
6	14	22

Figure 13: Inserting key 11 and creating $\lambda = 0.78(> 0.75)$

Fig. 14 shows how matters have progressed.

The total number of buckets is 2^3 the double of what used to be (i.e., 2^2). This means that once redistribution occurs, we move on to the **next round 2**, p returns to point back to bucket #0 and the next phase hash functions to be used are: $h_2(k) = k \bmod 8$ and $h_3(k) = k \bmod 16$.

⇒ **Round: 2** — utilized functions are: $h_2(k) = k \bmod 8$ and $h_2(k) = k \bmod 16$

...

p→0	8	
1	9	
2	10	18
3	11	
4	4	
5	5	13
6	14	22
7	7	

Figure 14: Going into **round 2**

References

- [1] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible Hashing - A Fast Access Method for Dynamic Files. *ACM Trans. on Database Systems*, 4(3):315–344, September 1979.
- [2] G. D. Knott. Hashing Functions. *The Computer Journal*, 18(3):265–278, March 1975.
- [3] K. Loudon. *Mastering Algorithms in C*. O'Reilly, Sebastopol, CA, 1999.
- [4] L. Witold. Linear Hashing: A New Tool for File and Table Addressing. In *Proc. 6th Conf. on Very Large Databases*, 1980.