= Linear probing =

Linear probing is a scheme in computer programming for resolving collisions in hash tables , data structures for maintaining a collection of key ? value pairs and looking up the value associated with a given key . It was invented in 1954 by Gene Amdahl , Elaine M. McGraw , and Arthur Samuel and first analyzed in 1963 by Donald Knuth .
Along with quadratic probing and double hashing , linear probing is a form of open addressing . In these schemes , each cell of a hash table stores a single key ? value pair . When the hash function causes a collision by mapping a new key to a cell of the hash table that is already occupied by another key , linear probing searches the table for the closest following free location and inserts the new key there . Lookups are performed in the same way , by searching the table sequentially starting at the position given by the hash function , until finding a cell with a matching key or an empty cell .
As Thorup & Zhang (2012) write , " Hash tables are the most commonly used nontrivial data structures , and the most popular implementation on standard hardware uses linear probing , which is both fast and simple . " Linear probing can provide high performance because of its good locality of reference , but is more sensitive to the quality of its hash function than some other collision resolution schemes . It takes constant expected time per search , insertion , or deletion when implemented using a random hash function , a 5 @-@ independent hash function , or tabulation hashing .

= = Operations = =

Linear probing is a component of open addressing schemes for using a hash table to solve the dictionary problem . In the dictionary problem , a data structure should maintain a collection of key ? value pairs subject to operations that insert or delete pairs from the collection or that search for the value associated with a given key . In open addressing solutions to this problem , the data structure is an array T (the hash table) whose cells T [i] (when nonempty) each store a single key ? value pair . A hash function is used to map each key into the cell of T where that key should be stored , typically scrambling the keys so that keys with similar values are not placed near each other in the table . A hash collision occurs when the hash function maps a key into a cell that is already occupied by a different key . Linear probing is a strategy for resolving collisions , by placing the new key into the closest following empty cell .

= = = Search = = =

To search for a given key x , the cells of T are examined , beginning with the cell at index h (x) (where h is the hash function) and continuing to the adjacent cells h (x) + 1 , h (x) + 2 , ... , until finding either an empty cell or a cell whose stored key is x . If a cell containing the key is found , the search returns the value from that cell . Otherwise , if an empty cell is found , the key cannot be in the table , because it would have been placed in that cell in preference to any later cell that has not yet been searched . In this case , the search returns as its result that the key is not present in the dictionary .

= = = Insertion = = =

To insert a key ? value pair (x , v) into the table (possibly replacing any existing pair with the same key) , the insertion algorithm follows the same sequence of cells that would be followed for a search , until finding either an empty cell or a cell whose stored key is x . The new key ? value pair is then placed into that cell .
If the insertion would cause the load factor of the table (its fraction of occupied cells) to grow above some preset threshold , the whole table may be replaced by a new table , larger by a constant factor , with a new hash function , as in a dynamic array . Setting this threshold close to

zero and using a high growth rate for the table size leads to faster hash table operations but greater memory usage than threshold values close to one and low growth rates . A common choice would be to double the table size when the load factor would exceed 1 / 2 , causing the load factor to stay between 1 / 4 and 1 / 2 .

= = = Deletion = = =

It is also possible to remove a key ? value pair from the dictionary . However , it is not sufficient to do so by simply emptying its cell . This would affect searches for other keys that have a hash value earlier than the emptied cell , but that are stored in a position later than the emptied cell . The emptied cell would cause those searches to incorrectly report that the key is not present .
Instead , when a cell i is emptied , it is necessary to search forward through the following cells of the table until finding either another empty cell or a key that can be moved to cell i (that is , a key whose hash value is equal to or earlier than i) . When an empty cell is found , then emptying cell i is safe and the deletion process terminates . But , when the search finds a key that can be moved to cell i , it performs this move . This has the effect of speeding up later searches for the moved key , but it also empties out another cell , later in the same block of occupied cells . The search for a movable key continues for the new emptied cell , in the same way , until it terminates by reaching a cell that was already empty . In this process of moving keys to earlier cells , each key is examined only once . Therefore , the time to complete the whole process is proportional to the length of the block of occupied cells containing the deleted key , matching the running time of the other hash table operations .
Alternatively , it is possible to use a lazy deletion strategy in which a key ? value pair is removed by replacing the value by a special flag value indicating a deleted key . However , these flag values will contribute to the load factor of the hash table . With this strategy , it may become necessary to clean the flag values out of the array and rehash all the remaining key ? value pairs once too large a fraction of the array becomes occupied by deleted keys .

= = Properties = =

Linear probing provides good locality of reference , which causes it to require few uncached memory accesses per operation . Because of this , for low to moderate load factors , it can provide very high performance . However , compared to some other open addressing strategies , its performance degrades more quickly at high load factors because of primary clustering , a tendency for one collision to cause more nearby collisions . Additionally , achieving good performance with this method requires a higher @-@ quality hash function than for some other collision resolution schemes . When used with low @-@ quality hash functions that fail to eliminate nonuniformities in the input distribution , linear probing can be slower than other open @-@ addressing strategies such as double hashing , which probes a sequence of cells whose separation is determined by a second hash function , or quadratic probing , where the size of each step varies depending on its position within the probe sequence .

= = Analysis = =

Using linear probing , dictionary operations can be implemented in constant expected time . In other words , insert , remove and search operations can be implemented in O (1) , as long as the load factor of the hash table is a constant strictly less than one .
In more detail , the time for any particular operation (a search , insertion , or deletion) is proportional to the length of the contiguous block of occupied cells at which the operation starts . If all starting cells are equally likely , in a hash table with N cells , then a maximal block of k occupied cells will have probability k / N of containing the starting location of a search , and will take time O (k) whenever it is the starting location . Therefore , the expected time for an operation can be calculated as the product of these two terms , O (k2 / N) , summed over all of the maximal blocks

of contiguous cells in the table . A similar sum of squared block lengths gives the expected time bound for a random hash function (rather than for a random starting location into a specific state of the hash table) , by summing over all the blocks that could exist (rather than the ones that actually exist in a given state of the table) , and multiplying the term for each potential block by the probability that the block is actually occupied . That is , defining Block (i , k) to be the event that there is a maximal contiguous block of occupied cells of length k beginning at index i , the expected time per operation is
<formula>
This formula can be simiplified by replacing Block (i , k) by a simpler necessary condition Full (k) , the event that at least k elements have hash values that lie within a block of cells of length k . After this replacement , the value within the sum no longer depends on i , and the 1 / N factor cancels the N terms of the outer summation . These simplifications lead to the bound
<formula>
But by the multiplicative form of the Chernoff bound , when the load factor is bounded away from one , the probability that a block of length k contains at least k hashed values is exponentially small as a function of k , causing this sum to be bounded by a constant independent of n . It is also possible to perform the same analysis using Stirling 's approximation instead of the Chernoff bound to estimate the probability that a block contains exactly k hashed values .
In terms of the load factor ? , the expected time for a successful search is O (1 + 1 / (1 ? ?)) , and the expected time for an unsuccessful search (or the insertion of a new key) is O (1 + 1 / (1 ? ?) 2) . For constant load factors , with high probability , the longest probe sequence (among the probe sequences for all keys stored in the table) has logarithmic length .

= = Choice of hash function = =

Because linear probing is especially sensitive to unevenly distributed hash values , it is important to combine it with a high @-@ quality hash function that does not produce such irregularities .
The analysis above assumes that each key 's hash is a random number independent of the hashes of all the other keys . This assumption is unrealistic for most applications of hashing . However , random or pseudorandom hash values may be used when hashing objects by their identity rather than by their value . For instance , this is done using linear probing by the IdentityHashMap class of the Java collections framework . The hash value that this class associates with each object , its identityHashCode , is guaranteed to remain fixed for the lifetime of an object but is otherwise arbitrary . Because the identityHashCode is constructed only once per object , and is not required to be related to the object 's address or value , its construction may involve slower computations such as the call to a random or pseudorandom number generator . For instance , Java 8 uses an Xorshift pseudorandom number generator to construct these values .
For most applications of hashing , it is necessary to compute the hash function for each value every time that it is hashed , rather than once when its object is created . In such applications , random or pseudorandom numbers cannot be used as hash values , because then different objects with the same value would have different hashes . And cryptographic hash functions (which are designed to be computationally indistinguishable from truly random functions) are usually too slow to be used in hash tables . Instead , other methods for constructing hash functions have been devised . These methods compute the hash function quickly , and can be proven to work well with linear probing . In particular , linear probing has been analyzed from the framework of k @-@ independent hashing , a class of hash functions that are initialized from a small random seed and that are equally likely to map any k @-@ tuple of distinct keys to any k @-@ tuple of indexes . The parameter k can be thought of as a measure of hash function quality : the larger k is , the more time it will take to compute the hash function but it will behave more similarly to completely random functions . For linear probing , 5 @-@ independence is enough to guarantee constant expected time per operation , while some 4 @-@ independent hash functions perform badly , taking up to logarithmic time per operation .
Another method of constructing hash functions with both high quality and practical speed is

tabulation hashing . In this method , the hash value for a key is computed by using each byte of the key as an index into a table of random numbers (with a different table for each byte position) . The numbers from those table cells are then combined by a bitwise exclusive or operation . Hash functions constructed this way are only 3 @-@ independent . Nevertheless , linear probing using these hash functions takes constant expected time per operation . Both tabulation hashing and standard methods for generating 5 @-@ independent hash functions are limited to keys that have a fixed number of bits . To handle strings or other types of variable @-@ length keys , it is possible to compose a simpler universal hashing technique that maps the keys to intermediate values and a higher quality (5 @-@ independent or tabulation) hash function that maps the intermediate values to hash table indices .

= = History = =

The idea of an associative array that allows data to be accessed by its value rather than by its address dates back to the mid @-@ 1940s in the work of Konrad Zuse and Vannevar Bush , but hash tables were not described until 1953 , in an IBM memorandum by Hans Peter Luhn . Luhn used a different collision resolution method , chaining , rather than linear probing .

Knuth (1963) summarizes the early history of linear probing . It was the first open addressing method , and was originally synonymous with open addressing . According to Knuth , it was first used by Gene Amdahl , Elaine M. McGraw (née Boehme) , and Arthur Samuel in 1954 , in an assembler program for the IBM 701 computer . The first published description of linear probing is by Peterson (1957) , who also credits Samuel , Amdahl , and Boehme but adds that " the system is so natural , that it very likely may have been conceived independently by others either before or since that time " . Another early publication of this method was by Soviet researcher Andrey Ershov , in 1958 .

The first theoretical analysis of linear probing , showing that it takes constant expected time per operation with random hash functions , was given by Knuth . Sedgewick calls Knuth 's work " a landmark in the analysis of algorithms " . Significant later developments include a more detailed analysis of the probability distribution of the running time , and the proof that linear probing runs in constant time per operation with practically usable hash functions rather than with the idealized random functions assumed by earlier analysis .