Linear Probing

Advanced Data Structures – Homework 2

Jan Kohlhase

jan.kohlhase@est.fib.upc.edu

June 21, 2020

1 Introduction

Hash tables are one of the most important data structures commonly used in applications like database indexing or in-memory storage. In a hash table data is stored in a manner such that each data has its unique index value. This index value is determined by a hash function. A hash function maps a set of input elements of arbitrary length to a set of elements of fixed length such that the set of hashed values is usually smaller then their keys. To define a good hash function several aspects should be taken into account. First of all the mapping of elements should be as diffuse as possible, meaning elements that are close to each other should map to very different hash values. Also a hash function should not be irreversible and efficiently implementable while having a small probability for collisions. A collision occurs when two different input keys map to the same value. Since the set of possible hash values is usually smaller than the set of possible inputs, such collisions are in principle unavoidable. For this reason applications using hash functions need to take collisions into account. A method to detect and resolve collisions in hash tables is linear probing. Linear probing is a form of open addressing meaning that each slot in a hash table stores a single value (as opposed to seperate chaining where each slot can contain a list of values). When the hash function is causing a collision by mapping a new key to a slot that is already occupied, in linear probing this value will be inserted in the closest following free location.

An example of linear probing can be described by the *Musical chairs* game. Given a set of chairs and persons, each person randomly decides for a chair to take. If the chair is already occupied the person will search for the closest free chair. Figure 1 shows a game with 10 chairs and 10 persons. At start each person randomly choose a chair as follows:

Person	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9
Chair	3	1	4	1	5	9	2	6	5	3

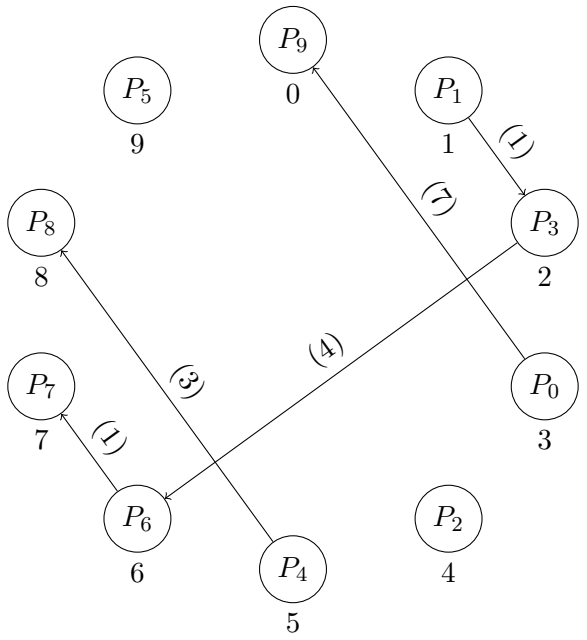

Figure 1: Linear probing example: *Musical chairs* game with persons $P_i, i \in \{0, 1, ..., 9\}$ in their randomly chosen chair $j \in \{0, 1, ..., 9\}$. An arrow between to spots indicates a probing labeled by its probe sequence lengths.

One after another every person tries to take their chosen chair. Each node in the figure is representing a chair identified by the number under the node. The person occupying the chair is shown inside the node. If a persons chosen chair is already taken, the person will move on clockwise until finding an empty chair to take. This procedure is called probing. Arrows in the figure are indicating the probes, pointing from the originally chosen chair to the next free chair. The arrows are labeled with the number of probes required to find a free chair, called the probe sequence lengths.

The previous example describes the standard policy used when inserting keys into a hash table using linear probing to resolve collisions. The following section will furthermore show two more sophisticated insertion policies: LCFS and Robin Hood. The third section discusses some experiments conducted towards the efficiency of the different insertion policies. The efficiency of an insertion policy can be measured regarding the number of operations required to find a given key. The experiments will show that the worst case search time can be drastically reduced by the insertion policy used.

2 Algorithms

When implementing linear probing for hash tables different insertion policies are possible and have been proposed in the literature. When the hash function is causing a collision the insertion policy determines which key, either the one that is already occupying the slot or the new one, needs to move on in the probing sequence. The following subsections will provide a closer look on the standard variant also called first-come-first-served (FCFS), its counterpart variant last-come-first-served (LCFS) [1] and a variant called Robin Hood [2, 3] depending on the probe sequence length.

The following definition is considered for all variants. Let H denote the hash table with elements $h_i, i \in \{0, 1, ..., n\}$. Further x, y denote keys from a set of keys S and the hash function $f : S \to H$ maps key x to a slot h_i which is either occupied by y or empty.

2.1 FCFS

By default when a collision occurs, the new key will be inserted in the closest following free slot. This policy is called first-come-first-served as the key that maps first to a free slot stays and all following colliding keys need to move on. The FCFS policy is the one used in the introductory example of the *Musical chairs* game (see section 1) and will be used as baseline.

Given the key x the algorithm calculates $h_i = f(x)$. If h_i is empty the algorithm stores x in h_i. While h_i is already occupied by a key y, y remains in h_i and the algorithm continues by trying to place x in $h_i = h_{i+1}$.

2.2 LCFS

Contrary to the FCFS policy, last-come-first-served resolves a collision by kicking out the present key in favour of the new key. The insertion algorithm then moves on always swapping the keys until finding a free slot. This policy was introduced in [1] where the authors showed how it reduces the variance of search probes drastically compared to the standard variant.

Given the key x the algorithm calculates $h_i = f(x)$. If h_i is empty the algorithm stores x in h_i. While h_i is already occupied by a key y it swaps x and y (placing x in h_i and setting $x = y$) and continues by trying to place x in $h_i = h_{i+1}$.

2.3 Robin Hood

The Robin Hood policy [2, 3] is based on the notion of probe sequence lengths (PSL). The PSL of a key is the number of probes required to find the key during lookup. A key with a low PSL can be thought of as rich, and a key with a high PSL can be thought of as poor. When inserting a new key the algorithm moves the rich in favor of the poor, which gives this policy its name ("taking from the rich and giving to the poor"). In [2, 3] the authors showed that their insertion policy reduces the variance of search probes drastically and they also proposed a new search procedure which requires only a constant number of probes on average.

Let $p(x)$ denote the probe sequence length of a key x. Given the key x the algorithm calculates $h_i = f(x)$. If h_i is empty the algorithm stores x in h_i. While h_i is already occupied by a key y it compares $p(x)$ and $p(y)$. If $p(x) < p(y)$ it swaps x and y by placing x in h_i and setting $x = y$, otherwise y remains in h_i. The algorithm continues by trying to place x in $h_i = h_{i+1}$.

3 Experiments

The efficiency of an insertion policy can be measured regarding the search time S_n (the number of operations required to find a given key). Given a hash table H of size $n = 1000$, experiments were conducted for different load factors $\alpha \in \{0.05, 0.1, ..., 0.95\}$. The load factor of a hash table is the amount of keys stored in it devided by its capacity. Therefore a set S of n random numbers were drawn. Subsequently each subset $S' \subset S$ of size $\frac{n}{20}$ were inserted into H using the modulo operator ($mod\ n$) as hash function. S_n were calculated for each existing value in the table (successful search). Each experiment were repeated 10 times. The plot in figure 2 shows the maximum search time S_n^{max} out of all trials for different loading factors α required when searching the hash table filled with different insertion policies. The source code of the implementation is available for authenticated users at the FIBs internal GitLab[1].

The plot shows that in worst case LCFS and Robin Hood are outperforming FCFS especially when the the hash table has a load factor $\alpha \geq 0.5$, saying when the hash table is at least half full. As LCFS and Robin Hood are showing a similar maximum search time, we will have a closer look on the distribution of their search time. Figure 3 shows boxplots comparing the probe sequence lengths for values inserted with LCFS or Robin Hood. While the median search time is the same for both insertion variants, one can see that the maximum and the variance of the probing for values inserted with the Robin Hood strategy is lower than the ones inserted with LCFS. This result reflects the outcome

[1] `https://gitlab.fib.upc.edu/jan.kohlhase/data-structures`

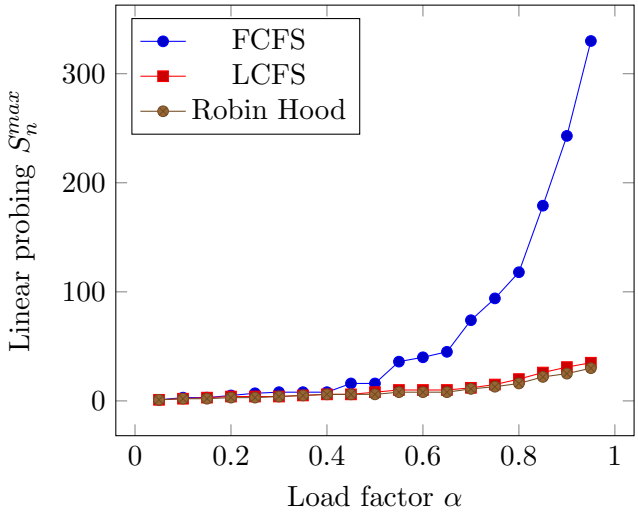

Figure 2: Probing plot. The plot shows the maximum number of probing shifts S_n^{max} needed to search for all values in a hash table of size $n = 1000$ for different load factors α of the table.

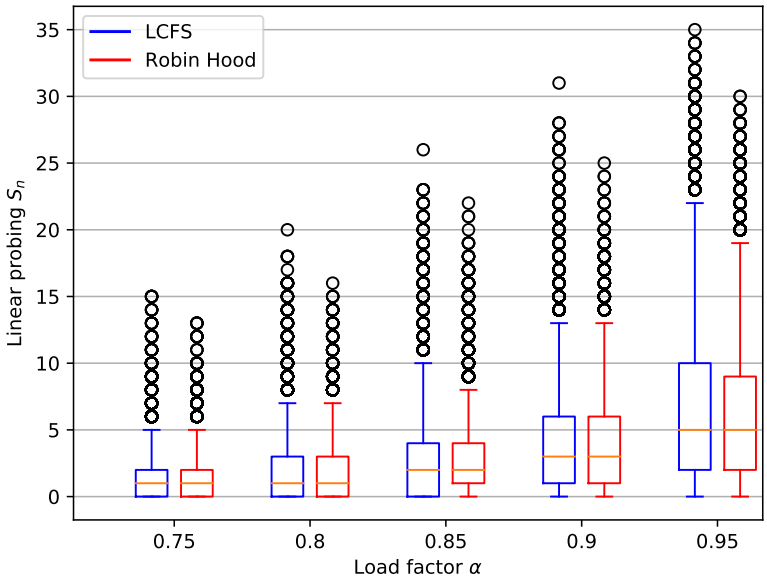

Figure 3: Boxplot comparing LCFS and Robin Hood probing for different load factors.

of [2, 3] where the authors stated: *"... insertion procedure which (in comparison to the standard approach) has the effect of dramatically reducing the variance of the number of probes required for a search"*. In addition to the insertion procedure, they also proposed

a search procedure making use of the way values get inserted in the hash table. So far the experiments concentrated on successful searches, meaning that all values searched in the table were existing values. When searching a value that is not present in the hash table the algorithm can only be sure that it is not present if it either comes across an empty slot or searched the whole table. With the Robin Hood insertion policy the search algorithm can also make use of the probe sequence length. When it comes across a value with a lower PSL it can be sure the searched value is not present.

4 Conclusion

Linear probing is a widely used scheme to resolve collisions in hash tables. The analysis of different insertion strategies showed a way to speed up the lookup time. Furthermore the more sophisticated insertion strategies also providing more constant search times. This analysis mostly concentrated on the insertion policies impact on the search time. Other operations worthy to analyse are the construction and deletion costs for hash tables resolving collisions with linear probing. There are interesting works regarding deeper analysis as [4, 5]. Regarding the construction for example an interesting feature of using the Robin Hood insertion policy is that the hash table will always have the same order independent on the order of insertion. Deletions with linear probing need to maintain the validity of searches as just emtpying cells will affect searches for keys probed above the deleted cell.

References

[1] Patricio V Poblete and J Ian Munro. Last-come-first-served hashing. *Journal of Algorithms*, 10(2):228–248, 1989.

[2] Pedro Celis, Per-Ake Larson, and J. Ian Munro. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 281–288. IEEE, 1985.

[3] Pedro Celis. Robin hood hashing. *PhD thesis, Computer Science Department, University of Waterloo*, Technical Report CS-86-14, 1986.

[4] Philippe Flajolet, Patricio Poblete, and Alfredo Viola. On the analysis of linear probing hashing. *Algorithmica*, 22(4):490–515, 1998.

[5] Rosa M Jiménez and Conrado Martínez. On deletions in open addressing hash-

ing. In *2018 Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 23–31. SIAM, 2018.