

Topics in Algorithms

Project report

Simon Kindstroem |2016-82282| 05/25/2017

Simeon Varbanov |2017-28499| 05/25/2017

Contents

[Abstract 3](#_Toc484431355)

[Introduction 3](#_Toc484431356)

[Background 3](#_Toc484431357)

[Dictionary problem 3](#_Toc484431358)

[Hash table 3](#_Toc484431359)

[Open addressing 4](#_Toc484431360)

[Chaining 4](#_Toc484431361)

[Theorem 4](#_Toc484431362)

[Perfect Hashing 5](#_Toc484431363)

[Dynamic Perfect Hashing 5](#_Toc484431364)

[Analysis and Design 6](#_Toc484431365)

[FullRehash(x) 7](#_Toc484431366)

[LookUp(x) 7](#_Toc484431367)

[Delete(x) – Lazy delete 7](#_Toc484431368)

[Insert(x) 8](#_Toc484431369)

[Others 8](#_Toc484431370)

[Implementation 9](#_Toc484431371)

[Testing 9](#_Toc484431372)

[Results 9](#_Toc484431373)

[Conclusion, Evaluation and Further Work 9](#_Toc484431374)

[Appendices 9](#_Toc484431375)

[Process Report Proposal 9](#_Toc484431376)

[Log Report 9](#_Toc484431377)

[GitHub repository overview 9](#_Toc484431378)

[Bibliography 9](#_Toc484431379)

# Abstract

Dynamic perfect hashing is an algorithm which gives O(1) worst case time for lookup and O(1) amortized expected time for insertion and deletion for the dictionary problem. This task was given to our group as a final project of the “Topics in in Algorithms” class. In this project, we decided to implement this algorithm and analyze deeply the given pseudocode and the outcome from our code. As a result, this project should produce high quality implementation in Python (programming language). The solution will be well documented and all the step from the begging of the project until the final stage will be documented in Log Report.

# Introduction

The group consist two members and we are both people with compute science background. However, Simon’s knowledge of the Python is much better, therefore he has a major role in the project implementation. However, this project will be done equally but both members and description of the work done by each member can be found in the Log Report or can be seen from the GitLog file attached as Appendices to this file.

For this project, we were not given so much time therefore we had to quickly come with a plan and analyze all possible problems that can arise during the implementation. First major problem that we found was understanding the pseudocode and prioritize the methods by their importance. Another issue was how can we compare our results and what kind of testing should be done to determine if the implementation is successful or not. Finally, since our coding styles are different best way of implementing this project is working together – this way misunderstandings can be avoided and the process flow can go easier, this was making our team work less flexible and due to schedule conflicts the number of possible group meetings was not big.

We decided to meet every Tuesday and Thursday from 9 until 11 a.m. for making the major project decisions together. During this meeting, we implemented most of the code but due to the time limitations we had to split some work and do some it at home. Simeon was more involved with documenting the choices we made and Simon about the code quality and real implementation. A better idea of each member duties can be seen in the GitLog file or in the Log Report.

# Background

## Dictionary problem

To get a better overview of what this algorithm solves we should introduce you to the problem that the Dynamic perfect hashing solves. Dictionary is an Abstract Data Type (ADT) are also known as associative arrays or maps. Every element in the dictionary has a key and an associated value for this key. Both the key and the value represent a pair. The analogy with the real-world dictionary comes from the fact, that in every dictionary, for every for word (key), we also have a description related to this word (value). Dictionary ADT has insert, remove and look up set of operations. The two major solutions to the dictionary problem are a hash table or a search tree. In some cases, it is also possible to solve the problem using directly addressed arrays, binary search trees, or other more specialized structures.

## Hash table

Hashing tables is a data structure invented 1953. In computing, a hash table (hash map) is a data structure which implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash collisions where the hash function generates the same index for more than one key (example of collision is k5 and k2 from the picture bellow). Such collisions must be accommodated in some way.

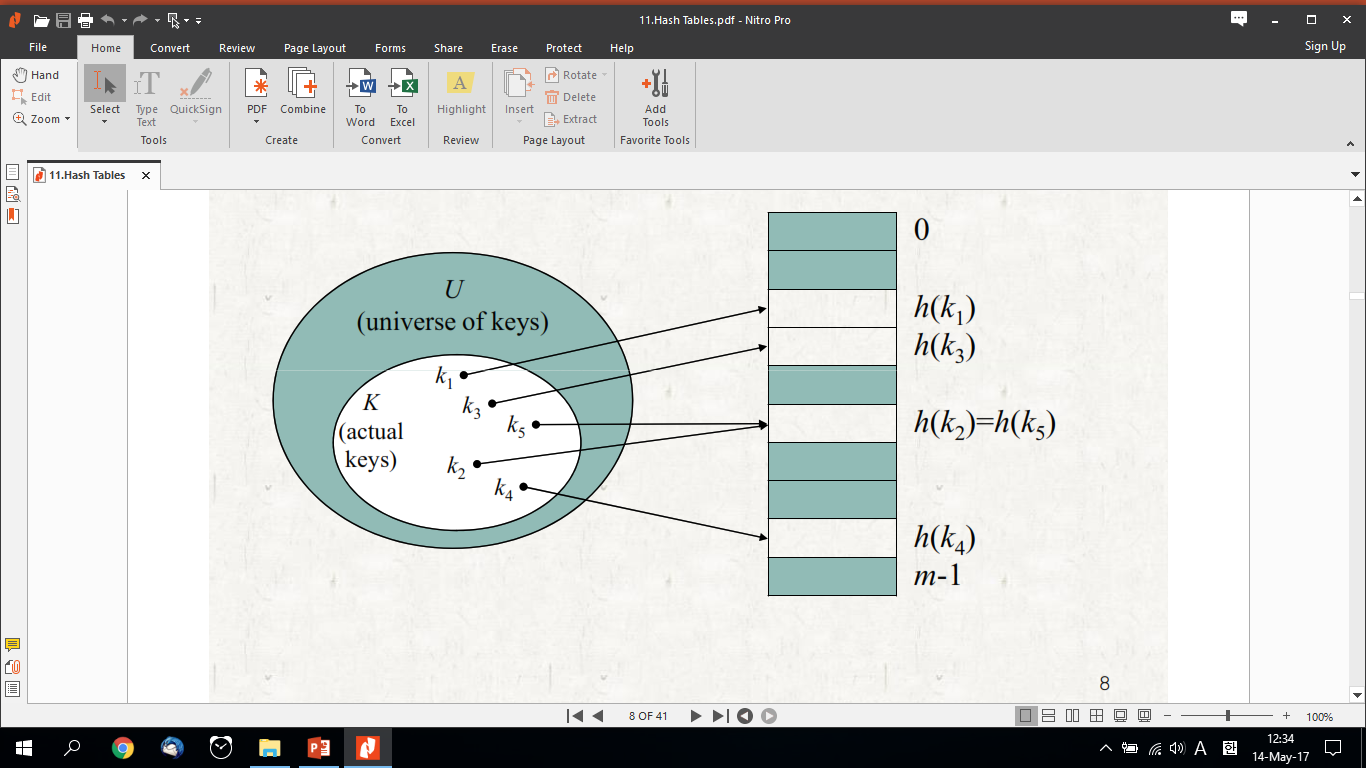


Figure 1 Hashing Collision

Popular collision resolving techniques are:

### Open addressing

To insert: if slot is full, try another slot, and another, until an open slot is found (probing). To search, follow same sequence of probes as would be used when inserting the element.

### Chaining

Keep linked list of elements in slots. Upon collision, just add new element to list (downside of this technique is that if a bad hash function is selected this can lead to a Linked List data structure which leads to much worst performance.).

Therefore, we need to pick our hash function very carefully. A conflict free solution can be universal hashing. We pick a hash function randomly when the algorithm begins (not upon every insert!), this guarantees good performance on average, no matter what keys adversary chooses, but for this we need a family of hash functions to choose from. This lead us to the following theorem expaling what a hash family is and more specifically a universal hash family:

### Theorem

If H is universal, then for any set K ⊆ U of size N, for any k ∈ U if we construct h at random according to H, the expected number of collisions between k and other elements in K is at most N/M.

H is said to be universal if:

for each pair of distinct keys x, y є U, the number of hash functions h є H for which h(x) = h(y) is |1|/m. In other words: With a random hash function from H, the chance of a collision between x and y is exactly 1/m (x ≠ y).

## Perfect Hashing

This lead us to Perfect Hashing Perfect Hashing or also known as FKS - Fredman, Komlós and Szémeredi – 1984. The characteristics for FKS are:

* Worst case for O(1) time for search.
* O(n) linear space in the worst case.
* Polynomial (nearly linear) to build this data structure.

The Idea behind FKS is two level hashing. We first hash the element to a slot of a table, which reference us to another table. In the second table, we use another hash function that hashes the element to the appropriate slots in the second table. This sequence of actions can be seen in the following figure:

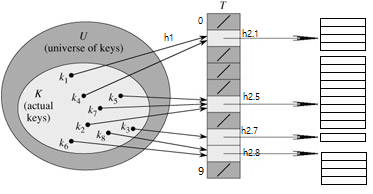


Figure 2Perfect Hashing -FKS

FKS this is a collision free solution, but has one main downside – the data that must be hashed in the table cannot be modified (this is a static solution for the dictionary problem). Therefore in 1993 Dynamic perfect hashing was introduced.

## Dynamic Perfect Hashing

In the dynamic case, when a key is inserted into the hash table, if its entry in its respective sub table is occupied, then a collision is said to occur and the sub table is rebuilt based on its new total entry count and randomly selected hash function. Because the load factor of the second-level table is kept low (1/k), rebuilding is infrequent, and the amortized expected cost of insertions is O(1). Similarly, the amortized expected cost of deletions is O(1).

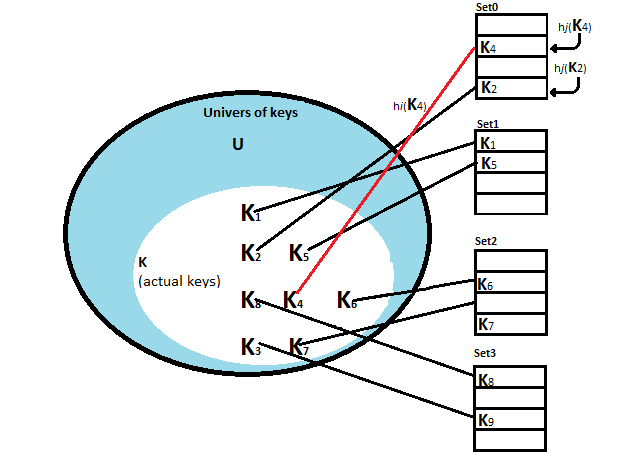


Figure 3 Dynamic Perfect Hashing

Additionally, the ultimate sizes of the top-level table or any of the sub tables is unknowable in the dynamic case. One method for maintaining expected O(n) space of the table is to prompt a full reconstruction when a sufficient number of insertions and deletions have occurred. By results due to Dietzfelbinger et al., as long as the total number of insertions or deletions exceeds the number of elements at the time of last construction, the amortized expected cost of insertion and deletion remain O(1) with full rehashing taken into consideration.

# Analysis and Design

For our project, we strictly followed the design given to us from the paper. We analyzed the pseudocode carefully and came up with the conclusion that the most vital method is RehashAll(x). Therebefore, we prioritized our implementation in the following order RehashAll(x), LookUp(x), Delete(x) and Insert(x).

## FullRehash(x)

In this method we first



## LookUp(x)

## Delete(x) – Lazy delete



## Insert(x)



## Others

During the implementation, we will create variables and methods that we will need for each one of these methods. These methods and variables will be explained in the Implementations next part of our report.

# Implementation

Xxx

# Testing

Xxx

# Results

Xxx

# Conclusion, Evaluation and Further Work

Xxx

# Appendices

## Process Report Proposal

## Log Report

## GitHub repository overview

# Bibliography

**There are no sources in the current document.**