Topics in Algorithms

PROJECT REPORT

06/08/2017

Simon Kindstroem |2016-82282|
Simeon Varbanov |2017-28499|

Contents

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.

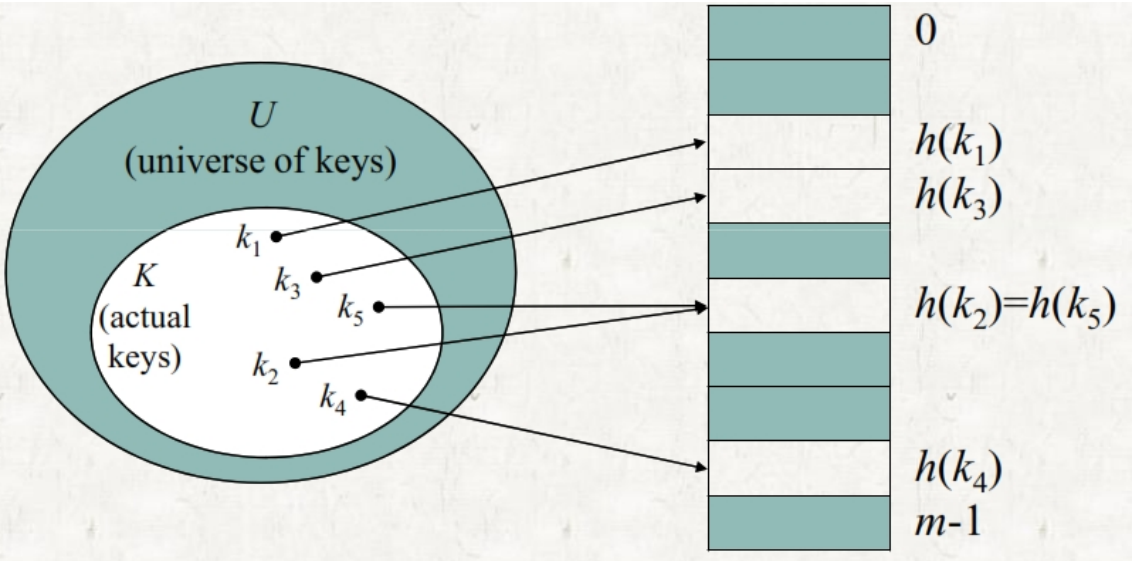

Figure 1 Hashing Collision

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 ϵ U, the number of hash functions h ϵ 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:

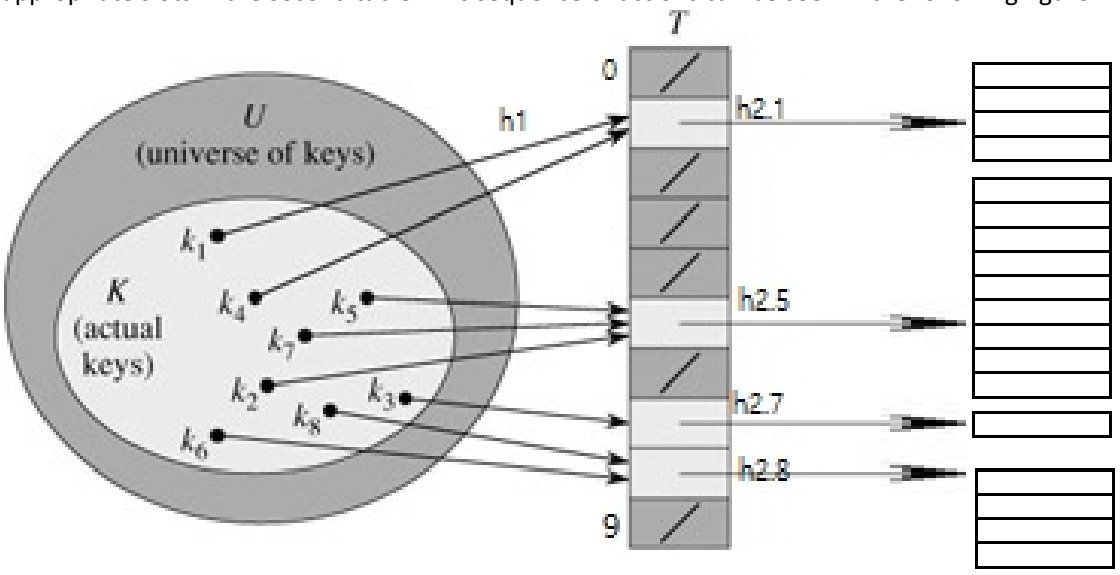

Figure 2Perfect Hashing -FKS

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) (Later on in the implementation stage we discuss why we changed this) .

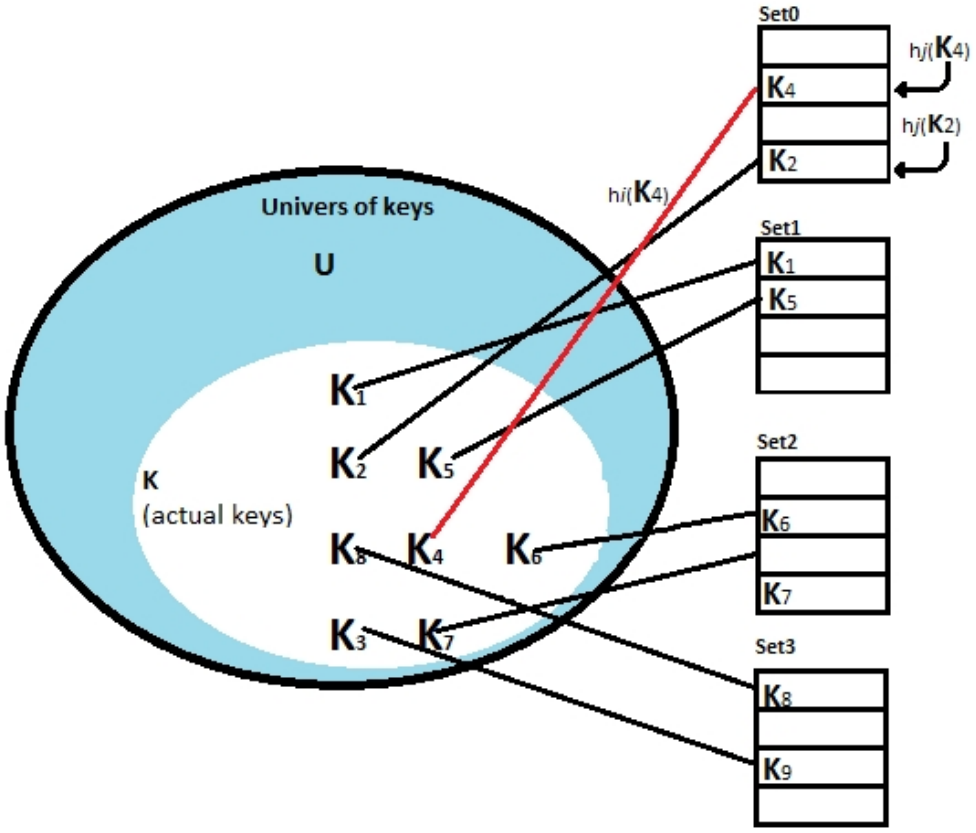

Figure 3 Dynamic Perfect Hashing

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 from 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). Therefore, we prioritized our implementation in the following order RehashAll(x), Locate(x), Delete(x) and Insert(x).

FullRehash(x)

Rehash all is either called by Insert with a parameter x, or by Delete or Initialize without parameters. Rehash all builds a new table for all elements currently in the table (plus x, if given).

In this method, we first go through table T and put all the elements from it into a list L, counting them and subsequently mark the elements as deleted.

Then we set M – a constant equal to (1 +c)*size of L-list
n – current number of elements stored in the table
c- is some value bigger than 0

The red bracket from the code below can be interpreted:
Randomly chosen function h ϵ $H_{s(M)}$ partitions S into the blocks W_j, 1 ≤ j ≤ s(M), where h(x) = (k**x** mod p) mod s(M). We will require that the following condition is (nearly) all the time satisfied:

$$\sum_{0 \le j \le s(M)} S \le \frac{32M^2}{s(M)} + 4M$$

```
function FullRehash(x) is
       Put all unmarked elements of T in list L;
       if (x is in U)
          append x to L;
       end if
       count = length of list L;
       M = (1 + c) * max{count, 4};
       repeat
          h = randomly chosen function in H_s(M);
          for all j < s(M)
             form a list L_j for h(x) = j;
             b_j = length of L_j;
             m_j = 2 * b_j;
             s_j = 2 * m_j * (m_j - 1);
          end for
       until the sum total of all s_j = 32 * M² / s(M) + 4 * M
       for all j < s(M)
          Allocate space s_j for subtable T_j;
          repeat
             h_j = randomly chosen function in H_sj;
          until h_j is injective on the elements of list L_j;
       end for
       for all x on list L_j
          store x in position h_j(x) of T_j;
       end for
end
```

The blue bracket represents the second sub level where we again rehash until there is no collisions.

Locate(x)

We set j to be a universal hash function. If the element in sub table T_j contains x and not deleted then this function will return true else will return false.

```
function Locate(x) is
       j = h(x);
       if (position h_j(x) of subtable T_j contains x (not deleted))
          return (x is in S);
       end if
       else
          return (x is not in S);
       end else
end
```

Delete(x) – Lazy delete

```
function Delete(x) is
        count = count + 1;
        j = h(x);
        if position hj(x) of subtable Tj contains x
            mark x as deleted;
        end if
        else
            return (x is not a member of S);
        end else
        if (count >= M)
            FullRehash(-1);
        end if
end
```

Deletion of x simply flags x as deleted without removal and increments count[1] (Later, in the Implementation stage we discuss about changing this to decrement). Only when a new level-one hash function h or a new hash function hj for the sub table Tj is chosen, we drop the elements with a tag "deleted" from Tj.

In the case of both insertions and deletions, if count reaches a threshold M the entire table is rebuilt, where M is some constant multiple of the size of S at the start of a new phase. Here phase refers to the time between full rebuilds. Note that here the -1 in "Delete(x)" is a representation of an element which is not in the set of all possible elements U.

Insert(x)

During the insertion of a new entry x at j, the global operations counter, count, is incremented.

If x exists at j, but is marked as deleted, then the mark is removed.

If x exists at j or at the subtable Tj, and is not marked as deleted, then a collision is said to occur and the jth bucket's second-level table Tj is rebuilt with a different randomly selected hash function hj.

```
function Insert(x) is
       count = count + 1;
       if (count > M)
          FullRehash(x);
       end if
       else
          j = h(x);
          if (Position hj(x) of subtable Tj contains x)
             if (x is marked deleted)
                remove the delete marker;
             end if
          end if
          else
             bj = bj + 1;
             if (bj <= mj)
                if position hj(x) of Tj is empty
                   store x in position hj(x) of Tj;
                end if
                else
                   Put all unmarked elements of Tj in list Lj;
                   Append x to list Lj;
                   bj = length of Lj;
                   repeat
                      hj = randomly chosen function in Hsj;
                   until hj is injective on the elements of Lj;
                   for all y on list Lj
                      store y in position hj(y) of Tj;
                   end for
                end else
             end if
             else
                mj = 2 * max{1, mj};
                sj = 2 * mj * (mj - 1);
                if the sum total of all sj ≤ 32 * M^2 / s(M) + 4 * M
                   Allocate sj cells for Tj;
                   Put all unmarked elements of Tj in list Lj;
                   Append x to list Lj;
                   bj = length of Lj;
                   repeat
                      hj = randomly chosen function in Hsj;
                   until hj is injective on the elements of Lj;
                   for all y on list Lj
                      store y in position hj(y) of Tj;
                   end for
                end if
                else
                   FullRehash(x);
                end else
             end else
          end else
       end else
end
```

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

Implementation was done strictly by the book and we did not have any major changes except the followings:

Delete – decrementing the counter and remove unnecessary check

```
def Delete(self, element):
    '''Deletion of x simply flags x as deleted without removal and increments count.'''
1   self.count -= 1
    j = self.universal_hash_function.hash(element)

2   if self.table_list[j] != None:

    table = self.table_list[j]

    location = table.hash_function.hash(element)

3       if table.elements[location] != None:
    if table.elements[location].deleted:
        raise ValueError('Element does not exist')
    else:
        table.elements[location].deleted = True

    if self.count >= self.M:
        self.RehashAll(-1)
```

As can be seen on the picture above:

1. We decided that makes more sense to decrement the counter, than incrementing it since the we are deleting an element and count is responsible for keeping the number of elements. We found out that this topic has different paper variations of the pseudocode. In the paper published in 1988 they decrement but in the revised paper in January 1990 they decrement. There is no explanation why this change was made, but we adopt the idea of the initial publication.
2. We removed this check since is useless since of course if the table list is not empty there will be table.
3. As explained before we had to unify the deletion check and from checking if the element location is not equal to None we just paraphrase it.

Locate – remove unnecessary check

```
def Locate(self,element):
    '''Return true if the element exists'''
    j = self.universal_hash_function.hash(element)
1   if self.table_list[j] != None:
    table = self.table_list[j]
    location = table.hash_function.hash(element)


        if table.elements[location] != None:
            return true
    else :
        return false
    if not table.elements[location].deleted and table.elements[location].value == element:
        return True


    return False
```

Just as in the previous method – Delete, in Locate we also remove this unnecessary check that could be avoided since the table will have a list.

During the implementation, we tried to split the methods and into smaller sub problems therefore to avoid unnecessary complexity. Methods like: calculate_m, is_injective, calculate_prime etc. to make the code more readable and easy to test later.

```
def calculate_prime(self, universe_size):
    '''Calculates prime larger than the universe_size'''
    # base cases
    if universe_size == 0:
        return 1
    elif universe_size == 1:
        return 2

    primes = []
    value = 3
    is_prime = True
    while True:
        # check if it's divisible by prime
        for prime in primes:
            if value % prime == 0:
                is_prime = False
                break

        # return if bigger than universe size
        if value > universe_size and is_prime:
            return value

        # add to prime list because not divisible by prime
        primes.append(value)
        value += 2
        is_prime = True
```

Testing

To test this project, we used Unit testing. Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation.

Initial Simple test

As can be seen in the Log Report in day 06.06.2017 we created a simple initial test and run it in the debugger to see that everything was displayed as expected. As can be seen from the picture bellow our initial test are very simple we just insert an element and using debugger we check if the tables are created correctly, if the element is inserted correctly. This initial test is just a small confirmation that we were working correctly. From them we had to change a few minor details like not fetching the element value but instead the whole element, there was not major mistakes detected.

```python
#!/usr/bin/env python3
'''Unit tests for dynamic-perfect-hashing'''

import unittest

# use weird import because of invalid module name (my bad)
dynamicperfecthashing = __import__('dynamic-perfect-hashing')

class TestDynamicPerfectHashing(unittest.TestCase):
    def test_calculate_prime(self):
        dynperf = dynamicperfecthashing.DynamicPerfectHashing(0)
        self.assertEqual(dynperf.prime, 1)

        dynperf = dynamicperfecthashing.DynamicPerfectHashing(2)
        self.assertEqual(dynperf.prime, 3)

        dynperf = dynamicperfecthashing.DynamicPerfectHashing(500)
        self.assertEqual(dynperf.prime, 503)

    def test_insert(self):
        dynperf = dynamicperfecthashing.DynamicPerfectHashing(2)

        dynperf.Insert(1)
        self.assertTrue(dynperf.Locate(1))

    def test_delete(self):
        dynperf = dynamicperfecthashing.DynamicPerfectHashing(2)

        dynperf.Insert(1)
        self.assertTrue(dynperf.Locate(1))

        dynperf.Delete(1)
        self.assertFalse(dynperf.Locate(1))

if __name__ == '__main__':
    unittest.main()
```

More in-depth testing

test_calculate_prime

```python
def test_calculate_prime(self):
    dynperf = DynPerf(0)
    self.assertEqual(dynperf.prime, 1)

    dynperf = DynPerf(2)
    self.assertEqual(dynperf.prime, 3)

    dynperf = DynPerf(500)
    self.assertEqual(dynperf.prime, 503)
```

As can be seen on the picture above the purpose of this test is to test the correct calculation of prime number. The prime number is connected to the size of the universe since the prime number should

be bigger than the universe. The calculate_prime function clearly states this connection. If universe is 0 then prime is equal to one. XXX

```
def calculate_prime(self, universe_size):
    '''Calculates prime larger than the universe_size'''
    # base cases
    if universe_size == 0:
        return 1
    elif universe_size == 1:
        return 2

    primes = []
    value = 3
    is_prime = True
    while True:
        # check if it's divisible by prime
        for prime in primes:
            if value % prime == 0:
                is_prime = False
                break

        # return if bigger than universe size
        if value > universe_size and is_prime:
            return value

        # add to prime list because not divisible by prime
        primes.append(value)
        value += 2
        is_prime = True
```

test_is_injective

Using a construct called "lambda" creation of anonymous functions (i.e. functions that are not bound to a name) we create a test that forces collision with a predictable hash.

```
def test_is_injective(self):
    '''Force collision with predictable hash'''
    dynperf = DynPerf(2)
    self.assertFalse(dynperf.is_injective([1, 1], PredictableHash(lambda _: 0)))
    self.assertTrue(dynperf.is_injective([0, 1], PredictableHash(lambda x: x)))
```

test_simple

In a similar manner the initial test before, this test is checking the Insert and Delete functionality.

```
def test_simple(self):
    dynperf = DynPerf(2)

    dynperf.Insert(1)
    self.assertEqual(dynperf.count, 1)
    self.assertEqual(len(dynperf.table_list), 1)
    self.assertTrue(dynperf.Locate(1))

    dynperf.Delete(1)
    self.assertEqual(dynperf.count, 0)
    self.assertEqual(len(dynperf.table_list), 1)
    self.assertFalse(dynperf.Locate(1))
```

test_insert_multiple

In this test we check if multiple inserts can be located and after that deleted successfully.

```
def test_insert_multiple(self):
    dynperf = DynPerf(2)

    dynperf.Insert(1)
    self.assertEqual(dynperf.count, 1)
    self.assertTrue(dynperf.Locate(1))

    dynperf.Insert(2)
    self.assertEqual(dynperf.count, 2)
    self.assertTrue(dynperf.Locate(1))
    self.assertTrue(dynperf.Locate(2))

    dynperf.Delete(1)
    self.assertEqual(dynperf.count, 1)
    self.assertFalse(dynperf.Locate(1))

    dynperf.Delete(2)
    self.assertEqual(dynperf.count, 0)
    self.assertFalse(dynperf.Locate(2))
```

test_outside_universe

Checking if the ValueError message will be triggered if an invalid insert is made (outside universe).

```
def test_outside_universe(self):
    dynperf = DynPerf(300)

    dynperf.Insert(0)
    self.assertEqual(dynperf.count, 1)
    self.assertRaises(ValueError, dynperf.Insert, -1)
    self.assertRaises(ValueError, dynperf.Insert, 301)

    self.assertRaises(ValueError, dynperf.Delete, -1)
    self.assertRaises(ValueError, dynperf.Delete, 301)

    self.assertRaises(ValueError, dynperf.Locate, -1)
    self.assertRaises(ValueError, dynperf.Locate, 301)
```

test_insert_duplicate

Duplicate check since there have not been mentioned in the paper how they should be handled when a duplicate inserted no exception or message is triggered. Similarly if the duplicate is deleted the value is deleted as it normally would.

```
def test_insert_duplicate(self):
    '''There's no documentation about how to handle duplicates, but IMO no change should occur'''
    dynperf = DynPerf(50)

    dynperf.Insert(37)
    self.assertEqual(dynperf.count, 1)
    self.assertTrue(dynperf.Locate(37))

    dynperf.Insert(37)
    self.assertEqual(dynperf.count, 1)
    self.assertTrue(dynperf.Locate(37))

    dynperf.Delete(37)
    self.assertEqual(dynperf.count, 0)
    self.assertFalse(dynperf.Locate(37))
```

test_insert_sub_table_no_size_increase_rehash

```
def test_insert_sub_table_no_size_increase_rehash(self):
    '''Create collision by forcing the elements into the same subtable and the same location'''
    dynperf = DynPerf(30)

    dynperf.global_hash_function = PredictableHash(lambda _: 0)
    dynperf.Insert(0)

    dynperf.table_list[0].hash_function = PredictableHash(lambda _: 0)
    dynperf.Insert(1)

    self.assertEqual(dynperf.count, 2)
    self.assertTrue(dynperf.Locate(0))
    self.assertTrue(dynperf.Locate(1))
```

test_insert_sub_table_size_increase_rehash

```
def test_insert_sub_table_size_increase_rehash(self):
    '''Create collision by forcing the elements into the same subtable and the same location'''
    dynperf = DynPerf(30)

    dynperf.global_hash_function = PredictableHash(lambda _: 0)
    dynperf.Insert(0)

    dynperf.table_list[0].hash_function = PredictableHash(lambda _: 0)

    dynperf.Insert(1)
    # max element count is 2, will force a rehash on subtable
    dynperf.Insert(2)

    self.assertEqual(dynperf.count, 3)
    self.assertTrue(dynperf.Locate(0))
    self.assertTrue(dynperf.Locate(1))
    self.assertTrue(dynperf.Locate(2))
```

test_create_another_subtable

```
def test_create_another_subtable(self):
    dynperf = DynPerf(100)

    for i in range(6):
        dynperf.Insert(i)

    self.assertTrue(dynperf.M, 6)
    self.assertTrue(dynperf.count, 6)

    # 6 elements are allowed until more tables are created
    dynperf.insert(6)

    self.assertTrue(dynperf.count, 7)

    for i in range(7):
        self.assertTrue(dynperf.Locate(i))

    for i in range(7):
        dynperf.Delete(i)

    for i in range(7):
        self.assertFalse(dynperf.Locate(i))
```

Results

Our test ran successfully and gave us the desired results. As we stated at the beginning of this project we successfully implemented Dynamic Perfect Hashing in Python programming language and ran the necessary test to assure that the implementation works and we do not have bugs or misinterpretation of the pseudocode. We strictly followed the instructions from the paper and the given pseudocode, as a result we successfully applied the knowledge that we got from “Dynamic Perfect Hashing: Upper and Lower Bounds”.

Conclusion

We implemented Dynamic perfect hashing using the pseudocode provided from the paper. It was challenging to understand completely the logic behind the pseudocode that is why we went many times trough analyzing the paper and understanding the meaning behind each variable. In this implementation, we tried to select our functions names and variables names precisely to avoid any confusion. Some of the group members tried for the first-time coding in Python programming language. We believe this project was a successful and was very educational for all members of the team.

Appendices

Process Report Proposal

N.B.

Since until recently both group members were occupied with preparing and submitting their presentation the progress is about the final project at the current stage is not big. However, in the next week we are expecting to be able to deliver more results.

Current Progress

Until this moment, we agreed what should be the goal of our project. The team decided that most interesting will be an implementation of the Dynamic Perfect Hashing and testing it is performance. Since such a job is already done in Java, we decided to write our solution in Python. To the extent of our knowledge such an implementation has not been done therefore will be interesting and challenging.

Our first job should be first setting clear goals what this Project should show and clear process how we will do that. We also need to come up with a good plan to document the whole process in a clean and understandable manner. Therefore, we will create a version control repository – GitHub so we can keep track on our changes.

When implementing the code focus will be that on readability instead of performance, making sure not to care about micro-optimization. Readability is important to help other people understand the algorithm better and implement it in their language of choice.

Functions

As we already know from the pseudocode we have the following functions:

1. Locate(x)
2. Insert(x)
3. Delete(x)
4. FullRehash(x)

Most time demanding and important functions are Insert(x) and FullRehash(x).

*For the moment, this is an initial plan and can go over the time can undertake some changes.

Log Report

31.05.2017

Today we had a meeting in which we discussed the code ident style and looked and analyze the pseudo code. We also decide that is a good idea to keep a log of our meetings to keep a better track of our progress. Today we started implementing the most important function RehashAll and we choose date and time for the next meeting (from 09:30 a.m. until 11 a.m 01/06/2017). Since Simeon does not have experience with Python programming language for the next meeting he should prepare and implement the simplest function - Delete, this way he can get a better feeling about the language syntax and get more involve with the project. In the next meeting the group will go over the implementation and discuss if there are any problems with this function. Also, the group will proceed with the rest of the implementation.

01.06.2017

Today we finished implementation of the RehashAll(x). Simon checked the Delete(x) function and made some minor changes. There is two more function to be implemented and we need to make create a table of hash functions. Today we made some structural changes too:

1. A class called Table was made in which will represent the second layer of the table list. Like showed in the diagram below we have a hashing function h_i that hashes to a list of tables and h_j hash function which rehashes this key in the specified by the h_i table in the specified table slot.
   class Table has the following parameters:
   elements – a list with all elements
   element_count – number of elements in this specified table
   max_element_count – as shown in the pseudocode represent 'M' the maximum number of element that the table supports before being rehashed
   allocated_space – representing the total space of the table
   hash_function – holding the chosen hash function
2. The list variable **tables** was renamed to **table_list** more appropriate name because was confused with the other list variable **table** which holds the list of the specific table's elements.

Simeon will should implement until the next meeting the Insert(x) functionality and the team will discuss the implementation on their next meeting. Next meeting will be scheduled on Tuesday 6th until then we should be done with the implementation. Run a simple tests and see if the code works well.

06.06.2017

Today Simon implemented the Insert function and we together went through the code to analyze if we have any mistakes. We found out that there was an ambiguous function in our code we were both using deleted and None if one element is deleted therefore we unified to just deleted.

Today we did some other minor changes in the code that can be seen in commit "f1832911". In this commit we also implemented a simple test which Insert (1) and Locate (1) after that Delete (1) and Locate (1). We implemented this basic test to assure that the code is good for now.

As an individual work, each member was assigned a task:

Simon should work over the other test cases.

Simeon should work on finalization of the report.

For our next meeting, we are expecting all the documentation and implementation part to be done.

08.06.2017

We finalized the report with agreeing that even if the report is quite big, we should keep it that way. We made the final version of the report in PDF and submit it together with the implementation.

GitHub repository overview

For a better overview of the in involvement of each member the following commit log can be seen. Each member contributed fairly for the completion of this project.

```
$ git shortlog
Simon Kindström (19):
      Added my student id and reordered our names
      Start of RehashAll
      Cleaned up delete a bit
      Rehash supposedly working
      Added tests for creating a prime
      Fixed create prime function
      Added some comments and fixes to the report
      Implementation of Insert
      Simple insert / delete is successful
      Updated tests
      Make sure value is inside universe
      Add tests for duplicates
      Wrote test on count
      Fixed duplicate values
      Updated initial list count to 1
      Rehash subtable tests and fix
      Added tests for rehashing subtables
      Added test for inserting until more tables are created
      Some updates to the report

simeonvar (19):
      Initial commit
      Initial Draft Process Report
      Make PDF File
      Make PDF v 1.1
      Add Log Report file
      Add Delete method
      Add Delete function
      Remove a comment
      Make a Reports directory.
      Upadate Log Report
      Final Report structure.
      Write Abstract, Introduction, Background and part of Analysis and Design section.
      Merge reports and add Analysis and Design, Conclusion and Bibliography section.
      Implement Locate function.
      Update .gitignore
      Fix comments, write Implementation and first half of Testing.
      Finish writting Testing part.
      Finish writting Result section.
      Finalize the report and make a PDF version of the report.
```

Figure 4 Git log

Bibliography

" Project Report Structure." Project Report Structure. N.p., n.d. Web. 06 June 2017.

"Dynamic Perfect Hashing." Wikipedia. Wikimedia Foundation, 22 Mar. 2017. Web. 05 June 2017.

Dietzfelbinger, M., A. Karlin, K. Mehlhorn, F.m. Auf Der Heide, H. Rohnert, and R.e. Tarjan. "Dynamic Perfect Hashing: Upper and Lower Bounds." [Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science (1988): n. pag. Web.

Mjn. "Mjn/dynperfhash-java." GitHub. N.p., 10 May 2012. Web. 05 June 2017.

Cormen, Thomas H., and Charles E. Leiserson. Introduction to Algorithms, 3rd Edition. Place of Publication Not Identified: n.p., 2009. Print.