# Datastrukturer och algoritmer (Python) 7.5p
Assignment 4

Aladdin Persson (aladdin.persson@umu.se)

Version 1, Username: tfy17api
Supervisors: Thomas Johansson,
Henrik Rosén

## Innehåll

## 1 Introduction

In this report our goal is to implement the abstract data type table and see how it can be implemented in several different ways and discuss advantages, disadvantages and potential improvements. An overview and explanation of the abstract data type table and its operations is given in Section 2. We go into the different implementations of table using arrays and linked list and analyze in Section 3. In this report we also discuss a variant of a table using linked list that we call Move-To-Front table.

## 2 Table

A table is a data structure which gives us the ability to uniquely identify a specific object from a large collection of object. This is done by storing a key and an associated value for each object, contrary to how an array is implemented which only stores a value. This is perhaps best described by an example, and a good intuition of the data structure table is looking at a traditional phone book. We have names and for each name there is an associated telephone number. In this case we can view each name being a key and associated with each key is a value, i.e a telephone number.

The following is an informal specification of the data structure table and its operations.

- **Empty** - Constructs an empty table.
- **Isempty(table)** - Checks if the table currently has no keys and values. Returns True if it is empty and otherwise returns False.
- **Insert(key, value, table)** - Inserts an element with key, value in a table.
- **Lookup(key, table)** - Checks if the key exists in table and if it does then returns True and the associated value.
- **Remove(key, table)** - Checks if key exists in table and if it does then it removes the key and associated value.

A potential problem of this data structure is that we can imagine having several names which are identical, i.e identical keys but with each person having a unique phone number. Most commonly this problem is solved by having a requirement that each key has to be unique, and when inserting an already existing key the value gets overwritten with the value most recently inserted. The ideal run time for an operation with any algorithm is having constant time $\mathcal{O}(1)$, completely independent of the input size $n$. Remarkably this is possible to implement using an implementation referred to as a hash table. Comparing the operations to what would for example be used in a database storing information we can see that the operations for looking up, inserting and removing data and their impressive run times, a hash table implementation can be very useful in practice. This report will not implement a hash table, but rather display that the abstract data type of table can be implemented in various ways, with varying efficiency, but nevertheless containing all necessary operations.

It is important to emphasize that there are different ways of implementing the abstract data type table, for which we stated in the introduction that we will be doing it for array, linked list and look at a variant of linked list implementation. These implementations do not have the same run time for the operations but they all share the same characteristic of a table, i.e they all share the operations described in the informal specification.

## 3 Analysis of implementations

With the usage of directed lists to implement a table it is convenient to have each cell contain a tuple that contains both the key and the value. Similarly for the array the use of a tuple is practical, however it also possible to initialize two arrays, one for the keys and the other for the values. The implementations of a table with array and directed list share many features but there are also fundamental differences between a directed list and array that will impact the performance of the table implementation when using these data structures.

What we can say apriori having analyzed any speed tests or the specific table implementations utilizing these two data structures is that a **considerable advantage** for a linked list is that it's size does not need to be determined at the start. In other words having initialized a large array when only using a small portion of it or having to worry about increasing array size will not be of concern as it will only extend the linked list to the specific amount of cells that are needed. This would save quite a lot of memory and also save compute on initializing the large array. An advantage for the array implementation would be that as there is no need for pointers, hence if a large portion of the initialized array is used it would be more memory sparse.

In general when comparing array and linked list implementations there are some differences to keep in mind. When trying to insert elements at the beginning of an array it is normally necessary to move all the elements one position to the right, making the operation cost $\mathcal{O}(n)$ time, compared to a linked list which gives us $\mathcal{O}(1)$. Inserting at the end gives us the opposite, i.e array having constant time and linked list with linear complexity. Deletion of array element can also be costly because of the same fact that elements needs to be repositioned in contrast to linked list where the pointers can simply be readdressed. Due to the fact that array are stored in a contiguous block of memory and therefore removing the need for pointers they have quick **lookup** time for specific indices, compared to linked lists which have to follow pointers.

To the specific implementation of a table using arrays, the initial advantage of having quick **lookup** time is relatively absent as when finding a specific key there is a need to do a linear search through the array. This is similarly done for a linked list, and except for the need for not following pointers we do not expect to see a large difference in the run time for this operation between the two. For **insertion** the advantage for linked lists which can simply readdress the pointers in the beginning is counteracted by the fact that storing new key-value pairs at the end does not impact the table functionality or speed. Hence insertion of key-value pairs is just as fast for arrays as they are for linked list when using them for implementing a table. The only real difference is that for arrays we do not need to do the operation for assigning the pointers which would save the amount of operations for insertion.

Lastly analyzing the **deletion** operation for the array implementation it becomes apparent after some debugging that after deleting key-value pairs and then inserting at the end we would cause holes in our array and would cause a very inefficient implementation. Fortunately this is easily solved after realizing that it is possible to keep track of the last elements index, and when deleting an element simply assigning the last element to the deleted position and change the last elements index by subtracting one from it. This problem does not exist for linked lists as the only need is to remove the pointers and readdress them to other cells.

As mentioned in the introduction there was an implementation of table using arrays and one using linked list but one can also imagine having a variant of these. For example implementing a **Move-To-Front table** which moves a searched key to the front in hope to save on computation when searching for similar keys often.

| - | TableArray | TableList | MTFTable |
|---|---|---|---|
| 1000 elements Insertion | 0.41 s | 0.48s | 0.48s |
| 2000 elements Random lookups | 0.84s | 0.97s | 1.94s |
| 2000 elements Skewed lookups | 0.84s | 0.97s | 0.84s |
| 2000 elements lookups non-existent key | 1.67s | 1.91s | 1.94s |
| Remove all items | 0.41s | 0.23s | 0.25s |

**Tabell 1** – Comparing the operation time with the implementation of table using array and linked list

As seen in Table (1), many of the discussed characteristics can be seen but some surprising results are also seen. Skewed lookups is not comparably faster for **Move-To-Front table** which is surprising as this was the main intention of this variant. It seems the operations needed to move the key and value when it has been looked up costs more than it gives benefit in this case and is definitely noticeable for random lookups. The deletion operation for arrays discussed previously works fine but when removing all items we notice that it is comparably slower and this might be just because of this reason. When the intention is to remove all items, the need to fill the gaps is no longer necessary and are just wasted operations.

## 4 Conclusions

Depending on the use case there can be different answers to which implementation is the best. In general the array implementation of table performs quite well and the only losing scenario is when removing all items which would rarely happen in practice. As mentioned in our analysis of this implementation is that it is quite inefficient if we have many unused array elements then we have a memory disadvantage and also if we have scenarios where the maximum capacity of the fixed array size has been reached.

The **Move-To-Front table** had unimpressive performance, but I believe some changes to this can be quite useful in practice. Instead of instantly moving a searched key to the front it makes more sense to keep track of the amount of times a key has been searched and therefore the most searched should be on the front. One can imagine scenarios where perhaps there are a few elements in a database that are frequently searched and it therefore makes sense to prioritize having these more accessible than less searched items. This would be a significant advantage to the other implementations in practice I believe and the current tests do not summarize this fairly.

There might be thoughts on how these implementations deals with attempting to delete non-existent keys and the answer is that it will be necessary to search through the entire array and hence this the reason why we have such extensive look-up time for non-existent keys.

Lastly, as mentioned in the introduction of this report there is a more common way of implementing the data structure table using hash tables and would result in powerful speed increases compared to the current implementations. They have disadvantages but in general they perform incredibly well and is the reason they are used the most in practice. This report did not explain them fairly and was only briefly mentioned in the introduction as this report was mainly focused on the point that the abstract data structure table can have several different correct implementations.

# 5 Appendix

## 5.1 ArrayTable.py

Source code for `ArrayTable.py`.

```
# -*- coding: utf-8 -*-

# Purpose of this code is to implement the abstract datatype table using
# datatype array.

# Programmed by Aladdin Persson <aladdin.persson@umu.se>
#    2019-02-13 Initial programming

from Array import Array

class ArrayTable:

    def __init__(self, highval=1000):
        """
            Syfte: Skapar en tom tabell med hjälp av en array
            Returvärde: -
            Kommentarer: -
        """

        # Sätt alltid undre index till 0. Övregräns kan ändras.
        self._table = Array(lo = (0, ), hi = (highval, ))
        self.lastpos = -1

    def insert(self, key, obj):
        """
            Syfte: utökar eller omdefinierar tabellen så att nyckeln key kopplas
                   till värdet obj
            Returvärde: -
            Kommentarer: -
        """

        # Försök alltid ta bort key:n först och sedan lägg in. Vill ej ha flera
        # element med samma key!
        self.remove(key)

        self._table.setValue((self.lastpos + 1, ), (key, obj))
        self.lastpos += 1

    def isempty(self):
        """
            Syfte: Testar om tabellen är tom
            Returvärde: Returnerar sant om tabellen är tom, annars falsk
            Kommentarer:
        """
        return self.lastpos == -1

    def lookup(self, key):
        """
            Syfte: Ser efter om tabellen innehåller nyckeln key och returnerar
                   i så fall värdet som är kopplat till nyckeln
            Returvärde: Returnerar en tuppel (True, obj) där obj är värdet som
                   är kopplat till nyckeln om nyckeln finns och annars (False, Non
            Kommentarer: -
        """
        i = 0
        found = False

        while found != True and i <= self.lastpos:
            val = self._table.inspectValue((i,))

            # First runs first part of if statement, if its not None then tuple
            if val != None and val[0] == key:
                (newKey, newValue) = val
                return (True, newValue)

            i += 1

        return (False, None)

    def remove(self, key):
        """
            Syfte: Tar bort nyckeln key och dess sammankopplade värde.
            Returvärde: -
            Kommentarer: Om nyckeln inte finns så händer inget med tabellen
        """
        i = 0
        found = False


        while found != True and i <= self.lastpos:
            val = self._table.inspectValue((i,))

            # First runs first part of if statement, if its not None then tuple
            if val != None and val[0] == key:
                last_tuple = self._table.inspectValue((self.lastpos,))
                self._table.setValue( (i,), last_tuple)
                self.lastpos -= 1

            i += 1
```

## 5.2 TableList.py

Source code for TableList.py.

```
# -*- coding: utf-8 -*-

#Written by Lena Kallin Westin <kallin@cs.umu.se>.
#Bug fixed 2012-08-24 by Johan Eliasson
#May be used in the course Datastrukturer och Algoritmer (Python)
#at Umeå University.
#Usage exept those listed above requires permission by the author.

from DirectedList import DirectedList


# Datatypen Tabell enligt definitionen på sidan 117 i Lars-Erik Janlert,
# Torbjörn Wiberg Datatyper och algoritmer 2., [rev.] uppl.,Lund,
# Studentlitteratur, 2000, x, 387 s. ISBN 91-44-01364-7
#
# Variabler och funktioner som inleds med ett enkelt underscore "_" är privata
# för klassen och ska inte användas av de som använder denna klass.
#
# Denna klass implementerar tabell med hjälp av en riktad lista

class TableList:

    def __init__(self):
        # Syfte: Skapar en tom tabell med hjälp av en riktad lista
        # Returvärde: -
        # Kommentarer: I boken heter denna funktion Empty.


        self._table = DirectedList()

    def insert(self, key, obj):

        # Syfte: utökar eller omdefinierar tabellen så att nyckeln key kopplas
        #        till värdet obj
        # Returvärde: -
        # Kommentarer: Det krävs att key är en typ som kan jämföras med
        #         likhet. Om det är en egen klass måste man överladda
        #         funktionen __eq__

        if self.isempty():
            self._table.insert(self._table.first(), (key, obj))
        else:
            found = False
            pos = self._table.first()
            while (not found) and (not self._table.isEnd(pos)):
                (newKey, newObj) = self._table.inspect(pos)
                if newKey == key:
                    found = True
                    pos = self._table.remove(pos)
                    pos = self._table.insert(self._table.first(), (key, obj))
                pos = self._table.next(pos)
            if not found:
                self._table.insert(self._table.first(), (key, obj))

    def isempty(self):

        # Syfte: Testar om tabellen är tom
        # Returvärde: Returnerar sant om tabellen är tom, annars falsk
        # Kommentarer:

        return self._table.isempty()
    def lookup(self, key):

        # Syfte: Ser efter om tabellen innehåller nyckeln key och returnerar
        #         i så fall värdet som är kopplat till nyckeln
        # Returvärde: Returnerar en tuppel (true, obj) där obj är värdet som
        #         är kopplat till nyckeln om nyckeln finns och annars (false, None)
        # Kommentarer: Om kön är tom returneras (false, None)

        pos = self._table.first()
        while not self._table.isEnd(pos):
            (newKey, newObj) = self._table.inspect(pos)
            if newKey == key:
                return (True, newObj)
            pos = self._table.next(pos)
        return (False, None)

    def remove(self, key):

        # Syfte: Tar bort nyckeln key och dess sammankopplade värde.
        # Returvärde: -
        # Kommentarer: Om nyckeln inte finns så händer inget med tabellen

        if not self.isempty():
            found = False
            pos = self._table.first()
            while (not found) and (not self._table.isEnd(pos)):
                (newKey, newObj) = self._table.inspect(pos)
                if newKey == key:
                    found = True
                    pos = self._table.remove(pos)
                else:
                    pos = self._table.next(pos)
```

.

### 5.3 MTFTable.py

Source code for `MTFTable.py`.

```
# -*- coding: utf-8 -*-

#Written by Lena Kallin Westin <kallin@cs.umu.se>.
```

```
#Bug fixed 2012-08-24 by Johan Eliasson
#May be used in the course Datastrukturer och Algoritmer (Python)
#at Umeå University.
#Usage exept those listed above requires permission by the author.

# Modified to MTFTable by Aladdin Persson 2019-02-22

from DirectedList import DirectedList


class MTFTable:

    def __init__(self):
        # Syfte: Skapar en tom tabell med hjälp av en riktad lista
        # Returvärde: -
        # Kommentarer: I boken heter denna funktion Empty.

        self._table = DirectedList()

    def insert(self, key, obj):
        # Syfte: utökar eller omdefinierar tabellen så att nyckeln key kopplas
        #        till värdet obj
        # Returvärde: -
        # Kommentarer: Det krävs att key är en typ som kan jämföras med
        #          likhet. Om det är en egen klass måste man överladda
        #          funktionen __eq__

        if self.isempty():
            self._table.insert(self._table.first(), (key, obj))
        else:
            found = False
            pos = self._table.first()
            while (not found) and (not self._table.isEnd(pos)):
                (newKey, newObj) = self._table.inspect(pos)
                if newKey == key:
                    found = True
                    pos = self._table.remove(pos)
                    pos = self._table.insert(self._table.first(), (key, obj))
                pos = self._table.next(pos)
            if not found:
```

```
            self._table.insert(self._table.first(), (key, obj))

    def isempty(self):
        # Syfte: Testar om tabellen är tom
        # Returvärde: Returnerar sant om tabellen är tom, annars falsk
        # Kommentarer:

        return self._table.isempty()

    def lookup(self, key):
        # Syfte: Ser efter om tabellen innehåller nyckeln key och returnerar
        #        i så fall värdet som är kopplat till nyckeln
        # Returvärde: Returnerar en tuppel (true, obj) där obj är värdet som
        #        är kopplat till nyckeln om nyckeln finns och annars (false, None)
        # Kommentarer: Om kön är tom returneras (false, None)

        pos = self._table.first()
        while not self._table.isEnd(pos):
            (newKey, newObj) = self._table.inspect(pos)

            if newKey == key:

                # Add the searched key and value by removing
                # it at current position and adding it to the front!
                self.remove(newKey)
                self._table.insert(self._table.first(), (newKey, newObj))
                return (True, newObj)
            pos = self._table.next(pos)
        return (False, None)

    def remove(self, key):
        # Syfte: Tar bort nyckeln key och dess sammankopplade värde.
        # Returvärde: -
        # Kommentarer: Om nyckeln inte finns så händer inget med tabellen

        if not self.isempty():
            found = False
            pos = self._table.first()
            while (not found) and (not self._table.isEnd(pos)):
                (newKey, newObj) = self._table.inspect(pos)
                if newKey == key:
                    found = True
                    pos = self._table.remove(pos)
                else:
                    pos = self._table.next(pos)
```

.