Practical Assignment

BM40A1500 Data Structures and Algorithms

1. Implementing the Hash Table

1.1 Structure of the hash table

The hash table is practically a list array of length M. When initialized, each node gets value None. Only later when we want to insert a key to an empty node, a linked list node is inserted to the hash table.

```
class HashTable:
    def __init__(self, M):
        self.M = M
        self.T = [None] * M
```

Initializing the hash table

Since we are using linear hashing, each time a new value is inserted with the same index we want to save it as a new node of a linked list. For this I've made Node class, which has attributes value and the pointer to the next node.

```
class Node:
    def __init__(self, key, next):
        self.key = key
        self.next = next
```

Node class for linked list structure

1.2 Hash function

Because the hash table was supposed to work for both integers and strings, in the beginning of the hash function I convert the key value to string, so that the index values inserted will be handled the same way as their corresponding string value (e.g. 123 >> '123').

The first hash function I used was very simple, it calculated the sum of ASCII values for each character in the given key value and returned the remainder when divided by the table size. This hash function however filled the table poorly, leaving a lot of empty slots.

```
# Hash function
def hash(self, data) -> int:
    sum = 0
    data = str(data)
    for i in range(0,len(data)):
        sum = sum + ord(data[i])

    return sum % self.M
```

The original hash function

After the first version I figured more efficient hash function which I found from the course material[1]. This version of the hash function fills the table entirely, thus being much more efficient.

```
# Hash function
def hash(self, data) -> int:
    data = str(data)

    sum = 0
    mul = 1
    for i in range (0,len(data)):
        if i % 4 == 0:
            mul = 1
        else:
            mul = mul * 256
        sum += ord(data[i]) * mul

    return sum % self.M
```

The final hash function

[1] Antti Laaksonen, Tietorakenteet ja algoritmit, 2021

1.3 Methods

Hash function returns an index value in the hash table based on the key value inserted. Because the hash table accepts both integer and string data types, the data is first converted into string.

```
# Hash function
def hash(self, data) -> int:
    data = str(data)

    sum = 0
    mul = 1
    for i in range (0,len(data)):
        if i % 4 == 0:
            mul = 1
        else:
            mul = mul * 256
        sum += ord(data[i]) * mul

    return sum % self.M
```

Hash function

Insert function adds a new value to the hash table. First the index of the hash table is calculated with the hash function. If the first node is not defined, a new node with the given value is inserted there. Otherwise, the function goes through the overflow until a duplicate value, or an empty slot is found, and inserts a new node with the given value there.

```
# Method for inserting a key value to the hash table
def insert(self, value) -> None:
    i = self.hash(value)

    if self.T[i]:
        node = self.T[i]

        while True:
            if node.key == None:
                node.key = value
                break
            elif node.next == None:
                node.next = Node(value, None)
                break
            else:
                node = node.next
    else:
        self.T[i] = Node(value, None)

    return
```

Insert function

Search function searches whether a value exits in the table and returns a boolean value. First the index of the hash table is calculated with the hash function. Then we go through the overflow until the given value is found (return True) or we run out of nodes (return False).

```python
# Method for searching whether the given key value exists in the hash table
def search(self, value) -> bool:
    i = self.hash(value)

    if self.T[i]:
        node = self.T[i]
        while True:
            if node.key == value:
                return True
            elif node.key == None:
                break
            elif node.next == None:
                break
            else:
                node = node.next

    return False
```

Search function

Delete function removes a certain key value from the hash table. It uses the hash function to find the index in the hash table. If a value is found, the linked list will delete the path to this node by replacing the path to the node with the path to the next node (no matter if the next node is empty). The function works iteratively, first we find the index in the hash table and then we loop through the overflow list.

```python
# Method for removing a key value from hash table
def delete(self, value) -> None:
    i = self.hash(value)
    if self.T[i]:
        node = self.T[i]
        if node.key == value:
            self.T[i] = node.next
            return

        while True:
            if not node.next:
                return

            if node.next.key == value:
                node.next = node.next.next
                break
            else:
                node = node.next

    return
```

Delete function

Print function presents the hash table structure. The function goes through each index in the hash table, and for each index the whole linked list structure if there is one in the node. For example, in *hash_2.py* the hash table after the operations is presented as following:

```
Index   Nodes
0       12 -12456
1       hashtable 4328989
2
```

Print in terminal when running *hash_2.py*

```python
# Method for presenting the hash table
def print(self) -> None:
    print("Index\tNodes")
    for i in range(0, self.M):
        node = self.T[i]
        print(f"{i}\t", end="")
        while True:
            if node:
                print(node.key, end=" ")
                if node.next:
                    node = node.next
                    continue
            break
        print()
    return
```

Print function

2. Testing and Analyzing the Hash Table

2.1 Running time analysis of the hash table

Insert: Running time of the hash function is Θ(m), where m is the length of the data (for example the amount of letters in a word). The running time of the insertion depends on whether we are inserting to an empty node or to the overflow. In the best case the running time is Θ(1) if we are inserting to an empty node. In the worst case the entire table is in the same linked list, meaning that the running time for insertion is Θ(n), where n is the total amount of inserted keys. Because we insert the new value at the end of the linked list, we don't need to go through the list more than once: all we do is insert the new value at the end.

If we assume that m < n, the total running time of insertion is **Θ(n).**

Search: Running time of the hash function is Θ(m) like in insertion. Running time of the search depends on whether the searched value is found right away or at the end of the linked list structure. In the best case the running time is Θ(1) if the value is the first node in the linked list. In the worst case the entire table is in the same linked list and the searched value is at the end of this list, meaning that the running time for search is Θ(n), where n is the total amount of keys in the table.

If we assume that m < n, the total running time of searching is **Θ(n).**

Delete: Running time of the hash function is Θ(m) like in insertion and search. Running time of the removal depends (again) on whether the searched value is found right away or at the end of the linked list structure. In the best case the running time is Θ(1) if the value is the first node in the linked list. In the worst case the entire table is in the same linked list and the searched value is at the end of this list, meaning that the running time for search is Θ(n), where n is the total amount of keys in the table. When deleting the value, we replace the path of the previous node with the path to the next node, and the running time of this process is Θ(1).

If we assume that m < n, the total running time of deleting is **Θ(n).**

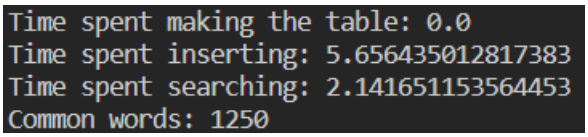
3. The Pressure Test

Step	Time (s)
Initializing the hash table	0.00
Adding the words	5.65
Finding the common words	2.14

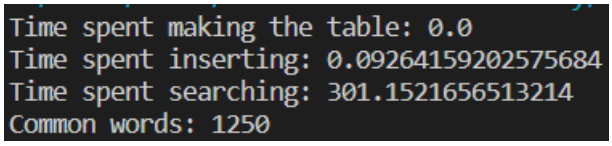
Results of the pressure test

3.1 Comparison of the data structures

Time comparison between two different data structures is presented in the pictures below:



Hash table



Linear array (list structure)

When using the list structure, I first added all the English words to a list, and then went through all the Finnish words and for each one of them compared whether or not they were in the list.

Linear array was much faster than the hash table when adding words. This is because we always added the new value at the end of the list. When we insert the value to the hash table, we first find the index on the table and then go through the linked list structure until an empty slot is found.

Hash table was significantly faster than linear array when searching for words. This is because the hash table is balanced based on the index value calculated by the hash function, so instead of going through the entire table when comparing values, we only go through the linked list structure for a specific index. In the linear array we always go through the entire list structure unless a match is found earlier.

3.2 Further improvements

By increasing the size of the hash table, I was able to improve the running time from the default size of 10 000.

```
Time spent making the table: 0.0
Time spent inserting: 2.968876361846924
Time spent searching: 0.5721848011016846
Common words: 1250
```

Running time analysis for a hash table with a size of 100 000

```
Time spent making the table: 0.008188724517822266
Time spent inserting: 2.346669912338257
Time spent searching: 0.42278456687927246
Common words: 1250
```

Running time analysis for a hash table with size of 1 000 000

```
Time spent making the table: 0.033376455307006836
Time spent inserting: 2.5116055011749268
Time spent searching: 0.405198335647583
Common words: 1250
```

Running time analysis for a hash table with size of 10 000 000

Increasing the size of the table only helps until a certain point, because the bigger the table, the more memory is used. In this exercise the number of words is less than 400 000 so having a table larger than this will guaranteed have empty slots.

When the size of the table is 10 000, there are no empty slots in the table. If the table size is increased to 100 000, only 97321 of the slots are filled which leaves 2619 slots completely empty. However, this is only 2.6% of the entire hash table, so relatively speaking not that bad.

If we want to determine how well the data is distributed, obviously we want to do some statistical analysis for the data. The data in this case is the length of the linked lists inside the hash table. For this I wrote an analysis function, which calculates the mean of the lengths, the standard deviation of the lengths and the number of empty nodes in the hash table.

```python
def analyse(self) -> None:
    lengths = []
    empty = 0
    for node in self.T:
        length = 0
        while True:
            if node:
                length += 1
                node = node.next
            else:
                break
        lengths.append(length)
        if length == 0:
            empty += 1

    sum_of_lengths = sum(lengths)
    nodes = len(lengths)

    mean = sum_of_lengths/nodes
    help_sum = 0
    for length in lengths:
        help_sum += (length-mean)**2
    std = sqrt(help_sum/nodes)

    print(f"\nAnalysis for data distribution ( M = {self.M} ):\n")
    print("Analysing the lengths of the linked lists inside the hash table")
    print(f"Mean: {mean}")
    print(f"Standard deviation: {std}\n")
    print(f"Empty nodes: {empty}\n")
    return
```

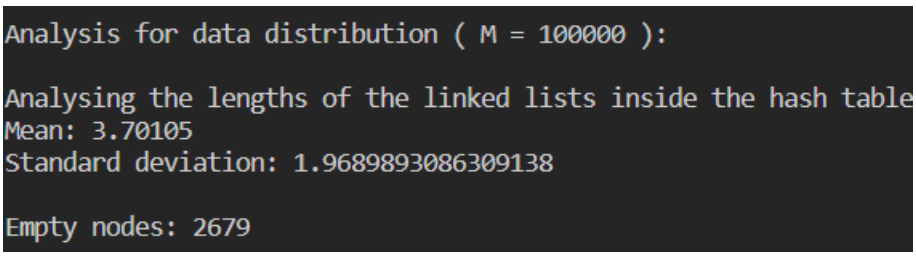
Analysis function

```
Analysis for data distribution ( M = 10000 ):

Analysing the lengths of the linked lists inside the hash table
Mean: 37.0105
Standard deviation: 6.515672624526202

Empty nodes: 0
```

Analysis when M = 10 000



Analysis when M = 100 000

As the result of the analysis, we can see that the data is well distributed in the hash table. The average length of a linked list is ~37 nodes and the standard deviation is ~7 nodes.

List of references

1. Antti Laaksonen, Tietorakenteet ja algoritmit, 2021