Project one report

Design and Analysis of Algorithms

Alessio Pepe - 0622701463
Dario Zito - 0622701512
Paolo Mansi - 0622701542
Teresa Tortorella - 0622701507

08/11/2020

0 Introduction

0.1 Work division

The team split into two groups, and each group performed two points, using the principles of agile pair programming methodology. A group formed by Dario and Paolo initially developed points 1 and 3, while the second group formed by Alessio and Teresa carried out points 2 and 4.

The first phase was that of an initial interface, to define the times and objectives for development. A development phase followed the initial step in which the points assigned to each sub-team have been developed. Finally, an interfacing process followed, where a merger of the code was carried out, and we collaborated to make improvements and corrections.

0.2 Code Organization

The developed code is in the `project1` package. The class required in exercise 1 is in the `doublehashinghashmap.py` file. The class required in exercise 2 is in `currency.py`. As requested by exercise 3 is in the `project1.abtree` package, which provides the implementation of general `ab-treemap`. A specific version of the latter is for using a `28-tree` with Currency objects inside. The function required in exercise 4 in the file `coin_change.py`.

Each module has a test code inside to verify the functionality of what is required.

0.3 Content organization

Each exercise is provided with chapter explanations on design, algorithms, and everything else necessary to quickly understand the code and the choices.

1 Exercise One

The first exercise asks to implement the class `DoubleHashingHashMap` that provides a hash map with a double hashing collision handling scheme.

1.1 Design choices

It has been decided to extend the `ProbeHashMap` class provided by the `TdP_collections` library.

It was decided to add an attribute `_collisions` that counts the number of collisions that occur during the insertions. The methods `__delitem__` and `__setitem__` have been redefined to count collisions on insertions only, in the specified cases, using a flag.

The polynomial function was used as a hash code, while the MAD compression was used as a compression function. The second hash function is

$$h'(x) = q - k \bmod q$$

where q is the smallest prime number before the length of the table, obtained using the `prevprime` function of the `sympy` module. The second hash function is used in the method `_find_slot`, instead of incremental function.

The resize operation maintains the load factor between 0.2 and 0.5. When the number of elements is above 50% of the table's length, the table's length is doubled and halved when it is under 20%. The `sympy` module is then used to round the new length to the previous or the following prime number.

1.2 Test module

The test module carries out insertions and deletions in a pseudo-random way. It prints the number of collisions that occurred, the actual load factor (provided with the function `get_load_factor`), the size of the map and the actual value inserted or deleted.

2 Exercise Two

The second exercise asks to implement the class `Currency`.

2.1 Design choices

Since operations on the denominations attribute are mostly lookup operations, it was decided to implement this structure as an `AVLTreeMap`. In this way, the operations' complexity is $O(2 \log n)$, except for `clear` and `iter_denomination` that has complexity $O(N)$ and `has_denomintion` and `num_denomination` that have complexity O(1).

It was decided to use the denomination as the map key and 0 as the value. In this way, in addition to guaranteeing the complexities described above, it supports a possible future evolution where you want to store the number of coins in your possession for each specific denomination (e.g., an application for a cash register).

For each requested operation, the algorithms already existing on the structure used to implement that method are called using the adaptation pattern.

2.2 Test module

In the currency.py module, the test code is provided to verify the implementations' functioning following the class's implementations.

3 Exercise Three

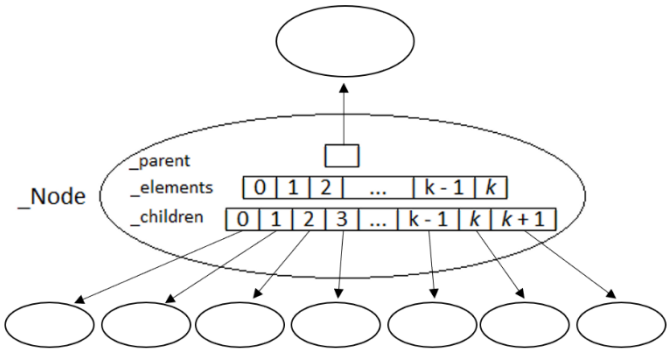
This exercise requires the implementation of a 2-8 Tree meant to store Currency objects.

3.1 Design choices

It was decided to extend the `Tree` class provided in the `Tdp_collection.tree` library through the class `abtree` to obtain a general Multiway Search Tree with a number a of minimum children and a number b of maximum children for each node. It was then created the class `abtree_map` that extended both the `abtree` class and the `map-base` class (provided in the `Tdp_collection.map` library) to obtain a data structure organized as a multiway search tree in the form of a map, where the `Item` is still generic. As the last step was created the class `currency_28tree` that specifies the defined requirements of this exercise and implements some methods to help the user accomplish the operation he needs.

As for the internal structure of a single node, it was decided to give it three attributes: a reference to the father of the specific node and two arrays, the former containing the elements sorted in the keys' natural order and the latter containing the references to the node's children. The team was led to this design choice by two reasons:

- the tree is meant to contain Currency object, objects that represent currencies in the standard ISO 4217, which means that there is a limited amount of possible currencies. Given that, searches within the tree are far more probable than insertion and deletion, as the actual currency rarely changes.
- we would obtain a data structure that allowed to maintain a connection between an element of the node and the nodes stored immediately before and after it, remaining compact. In this sense, the arrays offer the best solution in space complexity and connection between indices.



3.2 Algorithms explanation

As described above, the main operation that must be performed on this structure is research.

The search algorithm's time complexity is affected by the two parts it is made of: the search for the node within the tree and the search for the element within the node.

- This data structure respects the property of having a search complexity of the node of $O(\log N / \log a)$, where N is the number of nodes of the whole tree, and a is the minimum number of children the node has. That is obtained since it is an ordered data structure, and so only one node for each level will be inspected, with a maximum of $(\log N / \log a)$ nodes visited. This value also represents the tree's height.
- On the other hand, the array of elements stored in each node is itself an ordered structure, so even here, it is possible to obtain the time complexity of $O(\log b)$, where b is the maximum number of elements stored in the array, by implementing the dichotomic search algorithm (as done in the method `_list_search`). Since the array is sorted, the complexity of inserting and deleting will be linear.

In conclusion, the time complexity of the entire search algorithm is a combination of both the tree and node search complexity, obtaining a total complexity of $O\big((\log N / \log a) \cdot \log b\big)$. For

the same reason, the insertion and deletion operations, considering that an ordered array has time complexity $O(b)$ where b is the maximum number of elements, will have complexity $O\big((\log N / \log a) \cdot b\big)$.

3.3 Test module

In the class `abtree_map.py`, a test code is provided to verify the correct functioning of the methods implemented using the insertion and deletion of integers. As for more straightforward and more direct visualization of the tree's data, the test exploits the tree's representation in a vertical direction. A similar test code is provided in `currency_28tree.py`, but with the insertion and deletion of currency objects.

4 Exercise Four

The fourth exercise asks to solve the change problem using a Priority Queue as a support structure to get the minimum number of coins that sum up a given value.

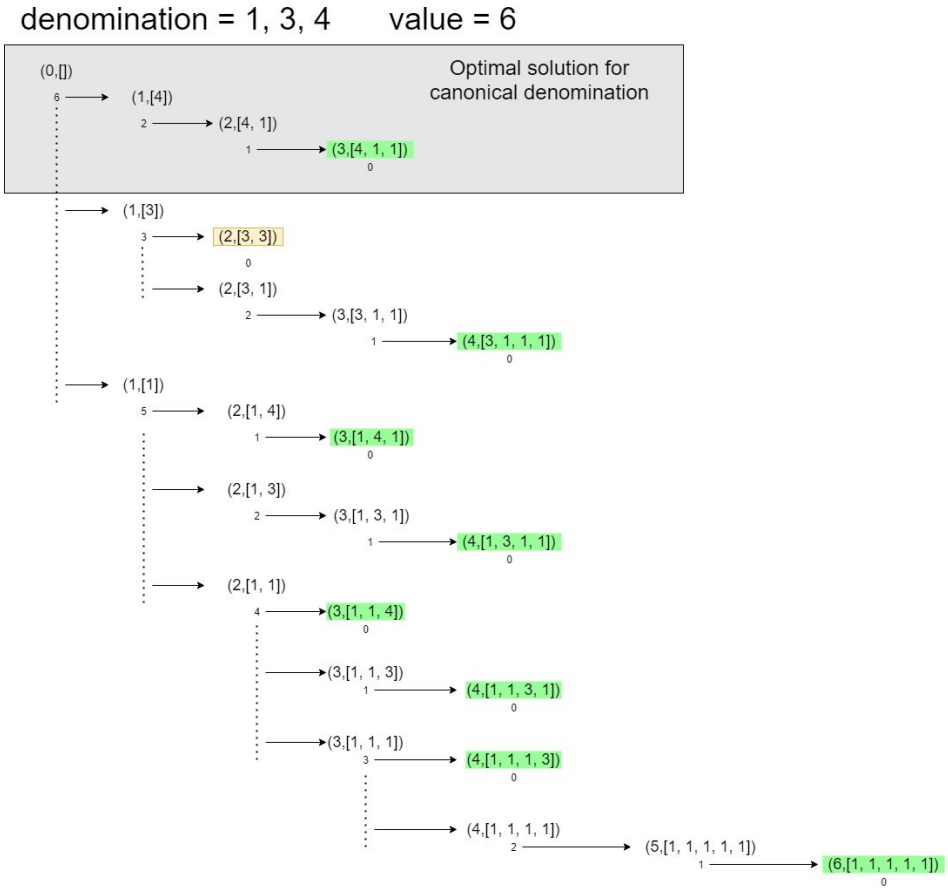
4.1 Design choices

Given the need not to have this structure's variants and the need to use only its essential logic, it was decided to use the `HeapPriorityQueue` queue provided in the `TdP_collections` library. The priority factor used is the minimum number of coins, to which the list of corresponding coins is associated.

Even if there are not many, it was decided to implement the solution to work even with non-canonical denominations.

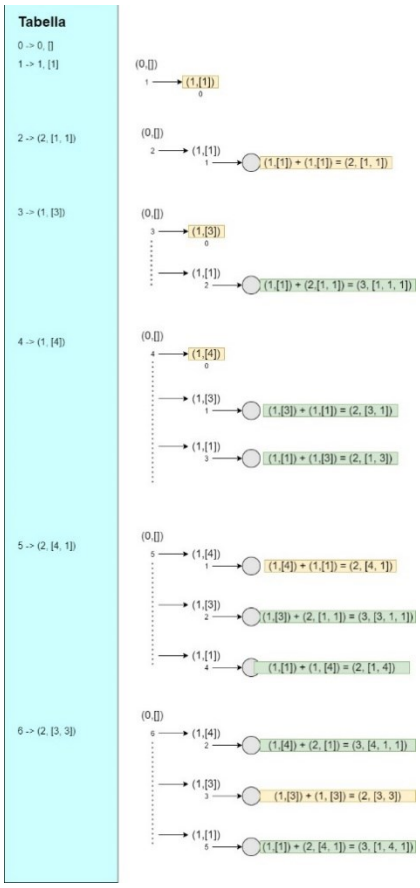
4.2 Algorithms explanation

A recursive version of the algorithm was initially implemented (which is not the final version obtained but is left inside the `change_recurs` function) to explore all possible solutions to the problem. Once a solution was calculated, it was inserted into the priority queue. Once all the solutions were calculated, the one that required the least number of coins was returned (then extracted from the priority queue). As shown in the next figure, this solution often requires the calculation of solutions already found previously, making it inefficient (the time complexity is exponential $O(\#denominations \cdot 2^{(value/0.01)})$.



In the image, you can see the number of coins used up to now, and the corresponding ones are represented, immediately below the rest that still needs to be returned. The solutions are those highlighted in green and yellow (therefore those present in the priority queue), where the latter is the optimal solution. The optimal solution for a canonical coin can be obtained through a greedy approach. For this reason, a canonical parameter has been added to the algorithm to reduce the complexity (which becomes linear) for canonical coins. Note: if there is no possibility to return that value with the given denomination, the queue will be empty.

It was later decided to use an approach that uses an additional support HashMap structure used to store the solutions already found (as shown in the next figure).



The approach is the same, but every time a previously calculated solution is encountered, the latter is used. In this way, the complexity decreases until it becomes polynomial ($O(\#denominations \cdot (value/0.01))$). The implementation of this version of the algorithm is provided in the `change` function.

4.2.1 Test module

In the `coin_change.py` module, following the algorithm's implementations, test code is provided to verify the functioning of the implementations.