Complexity and Algorithm Analysis of Hash tables and Disjoint sets implementations.

Juan Sebastián Rueda Segura

Data structures

Universidad Nacional de Colombia Sede Bogotá

1/03/2025

Contents

1 Introduction

To access the full code and previous implementations, you can check out this Github repository.

1.1 Goal

This document's objective is to study the implementation of the **Hash tables**, and **Disjoint sets** data structures, analyzing their corresponding algorithms to study the behavior and time complexity using Big O notation. The implementation is presented in *Python*.

1.2 Motivation

These non-linear data structures arise from the need of efficient insert, find and delete times, since the linear structures such as linked lists or array lists had at least a method that required O(N) time complexity. For hash tables we want to create a data structure that has $O(1)$ amortized complexity to find an element and for disjoint sets to know if there are connected components, useful in graph algorithms.

The disjoint set is a rooted-tree implementation, meaning that the leaves now point to the root. Each set is a rooted tree and the whole structure would be the forest containing them. It's array based for easy and constant time access. We have two possible implementations, using rank union and path compression.

For hashing, we will need an array, containing more arrays to handle collisions (this is called chaining) and a hash function for integers and strings. The hash maps implementation will allow insert, find and delete in almost constant time. This will depend on the longest chain but a good hash function will distribute the elements uniformly. We will show fixed size and variable hash table implementations.

2 Code Analysis

In this section we will explain details of the Tree interface in the BST and AVL trees, inspecting the corresponding code, making pertinent comments and finally we will give the time complexity in Big O notation.

2.1 Used Libraries

```
from abc import ABC, abstractmethod
import time
import matplotlib.pyplot as plt
import random
from typing import TypeVar, Generic, Optional, List, Any
```

We will need abc to create the interface of the trees, time and matplotlib to analyze the time measurements at the end of the document and random to generate numbers to be inserted on the trees and heap. Additionally, we will use *matplotlib* to graph the results obtained from the data analysis.

2.2 Interfaces and Templates

```
class DisjointSetsInterface(Generic[T], ABC):
    @abstractmethod
    def find(self, i: T) -> T:
        pass

    @abstractmethod
    def union(self, i: T, j: T) -> None:
        pass

class HashMap(Protocol, Generic[K]):
    @abstractmethod
    def put(self, key: K, value: Any) -> None:
        """Insert or update a key-value pair in the hash map."""
        pass

    @abstractmethod
    def get(self, key: K) -> Optional[Any]:
        """Retrieve a value by key."""
        pass

    @abstractmethod
    def remove(self, key: K) -> bool:
        """Remove a key-value pair by key."""
        pass

    @abstractmethod
    def size(self) -> int:
        """Get the current size of the hash map."""
```

```
29          pass
```

Python doesn't have properly defined *interfaces*, but they can be implemented through abstract classes, as the previous code shows, and then the data structures classes inherit from this class. It defines all the methods that have to be implemented and in case one or them is missing in the child class, the code won't compile and will show an error.

Templates or generics are a way to input parameters to functions or classes so that they can support multiple types of variables. This is useful in case we need to create strings or integer lists without having to change the implementation.

2.3 Disjoint Set (Rank)

When joining two trees in the forest we must ensure that we link the smallest one, so we create an array of the ranks that will be the height of the tree.

2.3.1 Constructor

```
def __init__(self, num: int):
    #makeset
    self.parents = [i for i in range(0, num)]
    self.rank = [0 for _ in range(0, num)]
```

Note that we have implicitly created the Makeset method when creating the parents array. This simplifies the operations, but when needing another data type this must be changed. We need twice the space to store the ranks as well.

Complexity: O(1)

2.3.2 Find

```
def find(self, i):
    while i != self.parents[i]:
        i = self.parents[i]
    return i
```

We iterate traversing the array until the parent of the index is equal to itself, meaning that it is contained in the same set.

Complexity: O(Log N)

2.3.3 Find

```
def union(self, i, j):
    i_id = self.find(i)
    j_id = self.find(j)

    if i_id == j_id:
        return

    if self.rank[i_id] > self.rank[j_id]:
        self.parents[j_id] = i_id
    else:
        self.parents[i_id] = j_id
        if self.rank[i_id] == self.rank[j_id]:
            self.rank[j_id] = self.rank[j_id] + 1
```

First we find the sets that contains i and j. If they are on the same set, we are done. Otherwise, we check the rank of the trees that represents the sets to see which is the smallest and append the other one. If they are the same size we update the rank of the one that appended the other. The complexity is O(Log N) because the number of nodes in a tree is at most Log N as we saw on class.

Complexity: O(Log N)

2.4 Disjoint Set (Path compression)

The main difference is that now we don't need the rank array because every element will now point to the element in the root, making the access faster.

2.4.1 Constructor

```
def __init__(self, num: int):
        #makeset
        self.parents = [i for i in range(0, num)]
```

As we said, now we don't need the rank array.

Complexity: O(1)

2.4.2 Find

```
def find(self, i):
    if i != self.parents[i]:
        self.parents[i] = self.find(self.parents[i])
    return self.parents[i]
```

We iterate traversing the array until the parent of the index is equal to itself. The difference with the previous implementation is that now we start to update the elemts that we traverse until we find the root, recursively calling the find method. This operation works in O(Log* N), the iterated logarithm as we saw on class.

Complexity: O(Log* N)

2.4.3 Find

```
def union(self, i, j):
    i_id = self.find(i)
    j_id = self.find(j)

    self.parents[i_id] = j_id
```

It's almost the same as the previous find but in this case we simply make the parent of i be j's. This works because of the find method. The complexity is O(2Log* N) since we call find twice.

Complexity: O(Log* N)

2.5 Fixed Size Hash Table

2.5.1 Constructor

```
def __init__(self, m: int):
    self.m = m
    self.buckets: List[List[Tuple[K, Any]]] = [[] for _ in range(m)
    ]
    self._count = 0

    # For integer hashing (ax + b) mod p mod m
    self.p = self._find_prime_larger_than(m * 100)
    self.a = random.randint(1, self.p - 1)
    self.b = random.randint(0, self.p - 1)

    # For string hashing (polynomial hash)
    self.x = random.randint(1, self.p - 1)
```

We will need m, the maximum size, buckets, the list of lists containing the hash map information, count that is the number of elements on the table, and some parameters to hash integers or strings. To find a prime number larger than m, we implement this function:

```
def _find_prime_larger_than(self, n: int) -> int:
        """Find a prime number larger than n."""
        def is_prime(num: int) -> bool:
            if num <= 1:
                return False
            if num <= 3:
                return True
            if num % 2 == 0 or num % 3 == 0:
                return False
            i = 5
            while i * i <= num:
                if num % i == 0 or num % (i + 2) == 0:
                    return False
                i += 6
            return True

        while not is_prime(n):
            n += 1
        return n
```

Complexity: O(1)

2.5.2 Hash method

```
def _hash_int(self, key: int) -> int:
        return ((self.a * key + self.b) % self.p) % self.m

    def _hash_str(self, key: str) -> int:
```

```
        hash_value = 0
        for char in key:
            hash_value = (hash_value * self.x + ord(char)) % self.p
        return hash_value % self.m

    def _hash(self, key: K) -> int:
        if isinstance(key, int):
            return self._hash_int(key)
        elif isinstance(key, str):
            return self._hash_str(key)
        else:
            raise TypeError("Key must be an integer or string")
```

Depending if is a hash table consisting on integers or strings, we have to select the corresponding method to hash. If it's an integer the hash function will be ((ax + b) mod p) mod m. And if it's a string it will be the polynomial hash as we saw on class.

Complexity: O(1)

2.5.3 Insert

```
def put(self, key: K, value: Any) -> None:
        hash_value = self._hash(key)
        bucket = self.buckets[hash_value]

        # Check if key already exists
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)  # Update value
                return

        # Key not found, append new key-value pair
        bucket.append((key, value))
        self._count += 1
```

We hash the key and search in it's bucket. If a value is found, we update it to the new one. And in case it's not in the bucket, we append it and increase the count attribute by one. The complexity depends on the longest chain c.

Complexity: O(c + 1)

2.5.4 Find

```
def get(self, key: K) -> Optional[Any]:
        hash_value = self._hash(key)
        bucket = self.buckets[hash_value]

```

```
        for k, v in bucket:
            if k == key:
                return v

        return None
```

We repeat the process in insert but if no key in the bucket matches the key to find, we return None.

Complexity: O(c + 1)

2.5.5 Find

```
def remove(self, key: K) -> bool:
        hash_value = self._hash(key)
        bucket = self.buckets[hash_value]

        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket.pop(i)
                self._count -= 1
                return True

        return False
```

We repeat the process in find. When the key is found we delete from the bucket the tuple value, key, subtract 1 to count and return True. If no key in the bucket matches the key to find, we return False.

Complexity: O(c + 1)

2.6 Variable Size Hash Table

We will use a fixed size table to use some operations, but will rebalance after an insertion.

2.6.1 Constructor

```
def __init__(self, initial_size: int, load_factor_threshold: float)
    :
        self.load_factor_threshold = load_factor_threshold
        self.count = 0
        self.capacity = initial_size
        self.map = FixedSizeHashMap[K](initial_size)
```

We set a load factor that represents $\frac{n}{m}$, the number of elements in the table and the maximum size. If we cross that balance factor, we rebalance.

Complexity: O(1)

2.6.2 Insert

```
def put(self, key: K, value: Any) -> None:
        # Check if key already exists
        old_value = self.map.get(key)
        if old_value is not None:
            self.map.put(key, value)
            return

        # New key, check load factor
        current_load_factor = (self.count + 1) / self.capacity
        if current_load_factor > self.load_factor_threshold:
            self._resize()

        self.map.put(key, value)
        self.count += 1
```

We insert it in the fixed table size and then we check if the load factor is still satisfied. Otherwise we resize:

```
def _resize(self) -> None:
        """Resize the hash map by doubling its size."""
        old_map = self.map
        new_capacity = self.capacity * 2
        new_map = FixedSizeHashMap[K](new_capacity)

        # Rehash all entries
        for bucket in old_map.buckets:
            for key, value in bucket:
                new_map.put(key, value)
```

```
11
12          self.map = new_map
13          self.capacity = new_capacity
```

We double the capacity and create a new map. We must then rehash all the elements present in the old table so we iterate through all the list accessing each list element and inserting it in the new table. Finally we update the pointer to new map and new capacity. Similarly to the array variable size, resizing takes lineal time but amortized constant time.

Complexity: O(c + 2)

2.6.3 Find and Delete

```
1       def get(self, key: K) -> Optional[Any]:
2           return self.map.get(key)
3
4       def remove(self, key: K) -> bool:
5           if self.map.remove(key):
6               self.count -= 1
7               return True
8           return False
```

We simply call the methods in the map attribute. In the delete we decrease the count.

Complexity: O(c + 1) each

3 Time measurements

3.1 Path compression vs Union by Rank

```
if __name__ == "__main__":
    sizes = [1000, 10000, 100000, 1000000]
    times = []
    times2 = []
    for size in sizes:
        start = time.time()
        ds = DisjointSets(size)
        for i in range(0, size - 1):
            ds.union(random.randint(0, size - 1), random.randint(0,
     size - 1))
        end = time.time()
        times.append(end - start)

        start = time.time()
        ds2 = DisjointSets2(size)
        for i in range(0, size - 1):
            ds2.union(random.randint(0, size - 1), random.randint
    (0, size - 1))
        end = time.time()
        times2.append(end - start)

    plt.plot(sizes, times2, marker='o', linestyle='-', label="Union
    -Find with Path Compression")


    plt.plot(sizes, times, marker='o', linestyle='-', label="Union-
    Find with Rank")
    plt.xscale("log")
    plt.yscale("log")
    plt.grid(True, which="both", linestyle="--", alpha=0.7)
    plt.xlabel("Size (n)")
    plt.ylabel("Total Time (s)")
    plt.legend()
    plt.show()
```

We compare different sizes and create two disjoint sets, one with path compression and the other with rank implemented. We see that they behave very similarly with the only exception being that the rank works better for small sizes.

3.2 Fixed vs Variable size

```
if __name__ == "__main__":
    sizes = [1000, 10000, 100000, 1000000]
    trials = 5
    times = []
    times2 = []
```

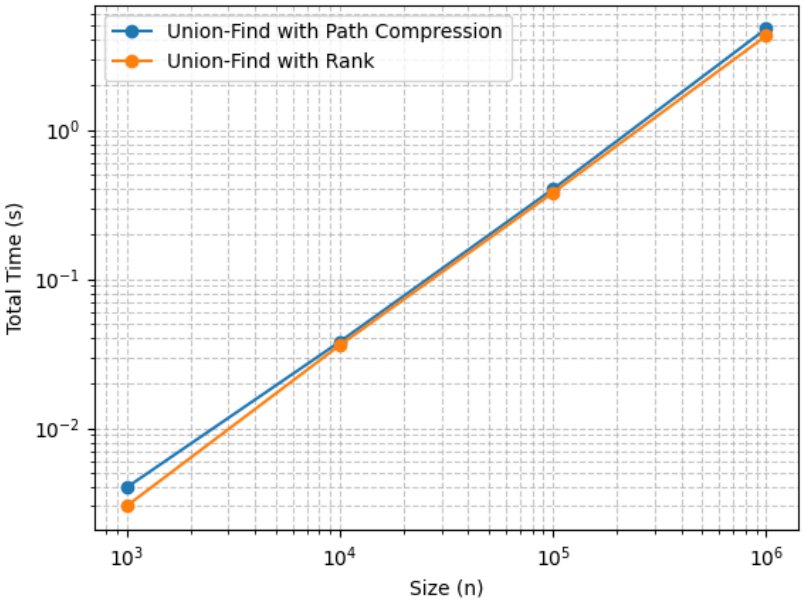

Figure 1: Comparison between Rank and Path compression

```
    for size in sizes:
        fhm_times = []
        rhm_times = []

        for _ in range(trials):
            random.seed(42)  # Ensure the same random values for both hash maps

            # FixedSizeHashMap timing
            start = time.time()
            fhm = FixedSizeHashMap(size)
            for i in range(size):
                fhm.put(random.randint(0, size - 1), random.randint(0, size - 1))
            end = time.time()
            fhm_times.append(end - start)

            random.seed(42)
            # ResizableHashMap timing
            start = time.time()
            rhm = ResizableHashMap()
            for i in range(size):
                rhm.put(random.randint(0, size - 1), random.randint(0, size - 1))
            end = time.time()
```

```
            rhm_times.append(end - start)

        times.append(sum(fhm_times) / trials)
        times2.append(sum(rhm_times) / trials)

    plt.plot(sizes, times2, marker='o', linestyle='-', label="
    Resizable Hash Map")
    plt.plot(sizes, times, marker='o', linestyle='-', label="Fixed
    Size Hash Map")
    plt.xscale("log")
    plt.yscale("log")
    plt.grid(True, which="both", linestyle="--", alpha=0.7)
    plt.xlabel("Size (n), Load Factor = 0.75")
    plt.ylabel("Total Time (s)")
    plt.legend()
    plt.show()
```

We compare different sizes and create two tables, one with path fixed size and the other with variable size. We set the initial size to be the default, 16 and the load factor to be 0.75 and we initialize the fixed size with the number of elements. We ran 5 tests using the same random seed, to ensure that the items are the same for each test. We see that the resizeable hash map performs better. I thought that the static had the advantage to get the same size as the total of elements, but it seems that the resize outperforms this.

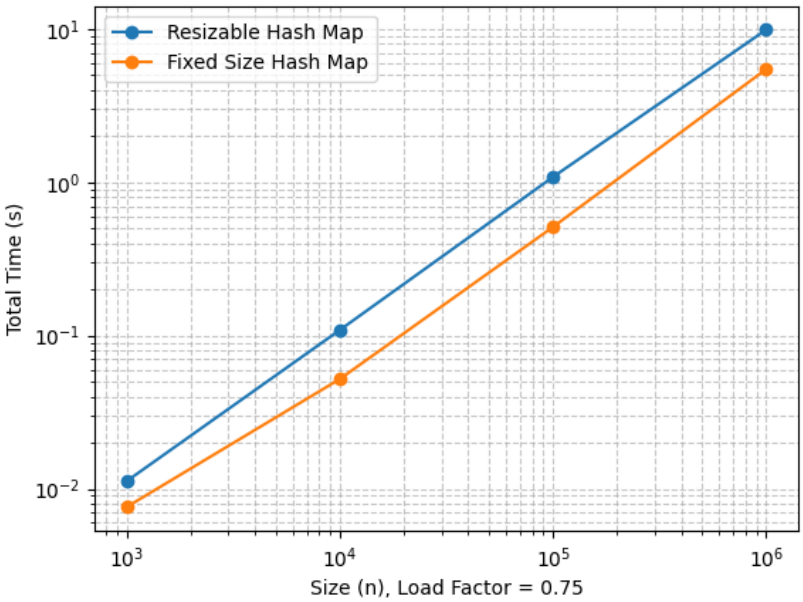

Figure 2: Comparison between Rank and Path compression

3.3 Load Factor

```
if __name__ == "__main__":
    sizes = [1000, 10000, 100000, 1000000]
    trials = 1
    load_factors = [0.1, 0.25, 0.5, 0.75, 0.9, 1.2, 1.5]
    times = {lf: [] for lf in load_factors}

    for size in sizes:
        rhm_times = {lf: [] for lf in load_factors}

        for _ in range(trials):
            for lf in load_factors:
                random.seed(42)  # Reset the seed for consistent
    values

                # ResizableHashMap timing
                start = time.time()
                rhm = ResizableHashMap(load_factor_threshold=lf)
                for i in range(size):
                    rhm.put(random.randint(0, size - 1), random.
    randint(0, size - 1))
                end = time.time()
                rhm_times[lf].append(end - start)

        for lf in load_factors:
            times[lf].append(sum(rhm_times[lf]) / trials)

    for lf in load_factors:
        plt.plot(sizes, times[lf], marker='o', linestyle='-', label
    =f"Resizable Hash Map (Load Factor = {lf})")

    plt.xscale("log")
    plt.yscale("log")
    plt.grid(True, which="both", linestyle="--", alpha=0.7)
    plt.xlabel("Size (n)")
    plt.ylabel("Total Time (s)")
    plt.legend()
    plt.show()
```

We compare different sizes and different load factors, following the same considerations of the last test. We can see that the smallest values, 0.1 and 0.2 don't behave stably and fluctuate depending the size. The loading factors from 0.5 to 1 behave the same and the 1.5 load factor works better with larger input sizes. I conclude that the loading factor is very situational and choosing it from 0.5 to 1.5 is better.

3.4 Initial size

```
if __name__ == "__main__":
    sizes = [1000, 10000, 100000, 1000000]
    trials = 1
```

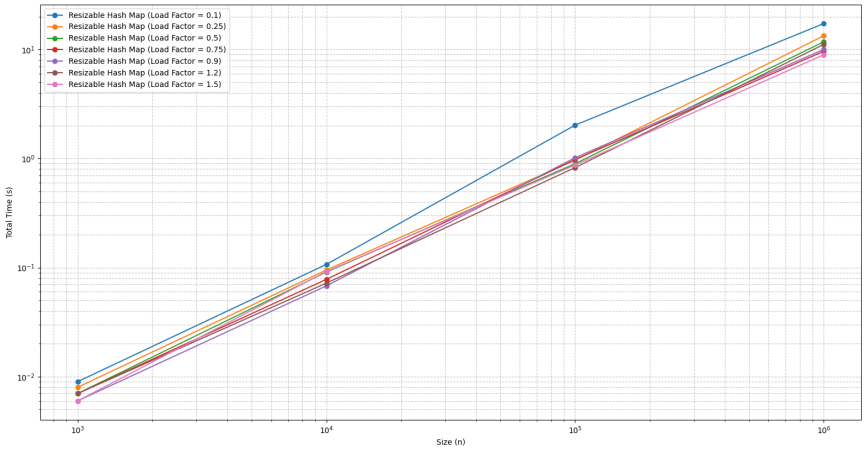

Figure 3: Comparison between Rank and Path compression

```
    initial_sizes = [0.5, 0.75, 1.0, 1.5]  # Multipliers for the initial size
    times = {isize: [] for isize in initial_sizes}

    for size in sizes:
        fhm_times = {isize: [] for isize in initial_sizes}

        for _ in range(trials):
            random.seed(42)  # Ensure the same random values for all hash maps

            for isize in initial_sizes:
                m = int(size * isize)
                start = time.time()
                fhm = FixedSizeHashMap(m)
                for i in range(size):
                    fhm.put(random.randint(0, size - 1), random.randint(0, size - 1))
                end = time.time()
                fhm_times[isize].append(end - start)

        for isize in initial_sizes:
            times[isize].append(sum(fhm_times[isize]) / trials)

    for isize in initial_sizes:
        plt.plot(sizes, times[isize], marker='o', linestyle='-', label=f"Fixed Size Hash Map (m = {isize} * size)")

    plt.xscale("log")
    plt.yscale("log")
    plt.grid(True, which="both", linestyle="--", alpha=0.7)
```

```
    plt.xlabel("Size (n)")
    plt.ylabel("Total Time (s)")
    plt.legend()
    plt.show()
```

We compare different sizes and different initial sizes for the fixed hash table, following the same considerations of the previous test. Here we see that if we assign the size of the list as m we have faster insertion in small cases but then it becomes the worst. The most stable is 1.5 times the size of the inputs and when the size is big the smallest size, 0.5, is good, which means that the hash function distributes uniformly.

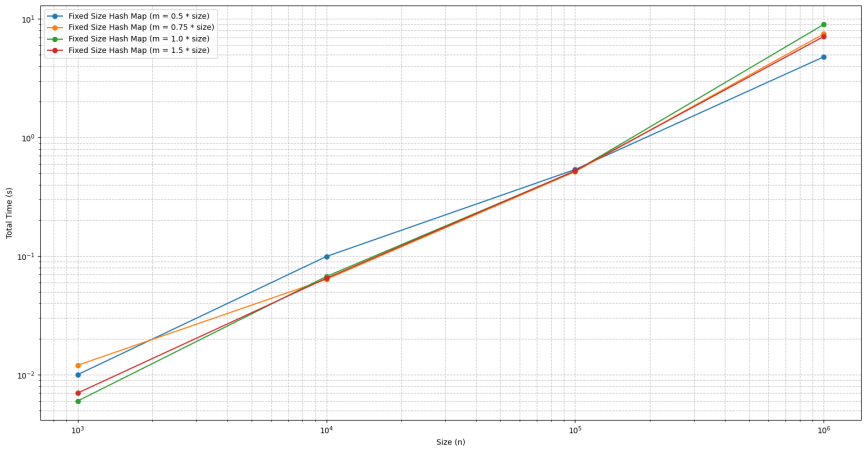

Figure 4: Comparison between Rank and Path compression

4 Conclusions

This document summarized two non-linear data structures implementations, analyzing the code, time complexity and obtained results given by testing the code and plotting the time vs size graphics. One thing to note is the faster results given by implementing these data structures. Note that Python is an interpreted language and therefore is slower than if we had implemented these data structures on a language like C or C++.

The results obtained were as I expected and as we saw on class: the best loading factor is between 0.5 and 0.9, if we assign a good hash function then the time is linear, which is represented on the graphs since we call N times the insert function, the resize is O(1) amortized time or else it would have showed worse results and finally the disjoint sets implementation using path compression is as efficient as using the rank but better in memory usage.