Algorithms - I Project

Universal Phonebook

MOTIVATION

Telephones and phone books were in use from the 20th Century. The last couple of decades saw a rise in the use of mobiles and the storage of contacts virtually. Nowadays, contacts directory of a mobile contains contacts stored in different sources. Our proposed algorithm inserts, deletes, searches a contact or merges contacts stored in two different sources into one.

PROBLEM DESCRIPTION

There are various places in a contact book in which contacts are stored, for example: sim card contacts, phone memory contacts, google id contacts etc.

We propose an efficient way to organize these contacts in this project. The user can add, delete, edit contacts in different memory locations and can search contacts by name. The user can also merge contacts present in different memory locations, and duplicates can be deleted, to have an organized contacts list.

Disjoint set data structure and hash tables can be used to implement the same in an efficient way.

PROBLEM STATEMENT

Given a randomly generated contacts directory, the user is provided with the following menu:

- Click 1 to insert a new contact
- Click 2 to delete an existing contact
- Click 3 to search for a contact
- Click 4 to merge contacts from 2 sources
- Click 5 to display the phonebook
- Click 6 to exit the phonebook

A test run is provided in the subsequent slides for a better understanding of the problem.

```
****Menu****
Click 1 to insert a new contact
Click 2 to delete an existing contact
Click 3 to search for a contact
Click 4 to merge contacts from 2 sources
Click 5 to display the phonebook
Click 6 to exit the phonebook
Your Input: 1
Enter contact name to be inserted: Cole
Enter contact number to be inserted: 9876543211
Enter the source number in which the contact is to be inserted: 1
Contact inserted successfully

Your input: 3
Enter contact name to be searched: Cole
Contact found
Cole: 9876543211
```

```
Your input: 2
Enter contact name to be deleted: Cole
Contact deleted successfully

Your input: 3
Enter contact name to be searched: Cole
Contact not found

Your input: 4
Enter two source numbers to be merged: 2 3
Merged successfully

Your input: 6
-----------------------------------------------
Thanks for using the Universal Phonebook
Developed by: Akshay Jain and Md Areeb Hussain
-----------------------------------------------
```

PSEUDO CODE

```
// There is an array of hash maps (phone_book) where each element of the array
contains contacts of 1 source.
// Some randomly generated data is initially present to populate the phone_book

insertContact(name, number, src) {
      //inserts a new contact in the specified source
      src = getParent(src)
      //getParent()uses the path compression heuristic and returns the parent of
the source.
      phone_book[src].insert(name, number)
}

deleteContact(name) {
      //deletes the contact with given name
      for element in phone_book {
            if(element.search(name)) {
                  element.delete(name)
                  return
            }
      }
}
```

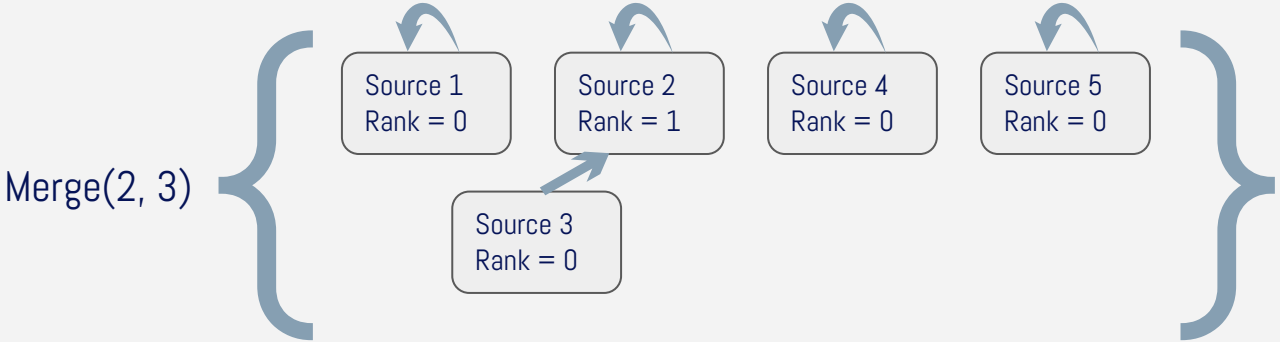
PSEUDO CODE

```
search(name) {
      //searches for a contact with the given name
      for element in phone_book {
            if(element.search(name))
                  return element[name]
      }
}

merge(src, des) {
      //merges two disjoint sets
      src=getParent(src)
      des=getParent(des)
      if(src != des) {
            if(phone_book[src].rank > phone_book[des].rank)
                  phone_book[src].parent=des
            else {
                  phone_book[des].parent=src
                  if(phone_book[src].rank == phone_book[des].rank)
                        phone_book[src].rank += 1
            }
      }
}
```

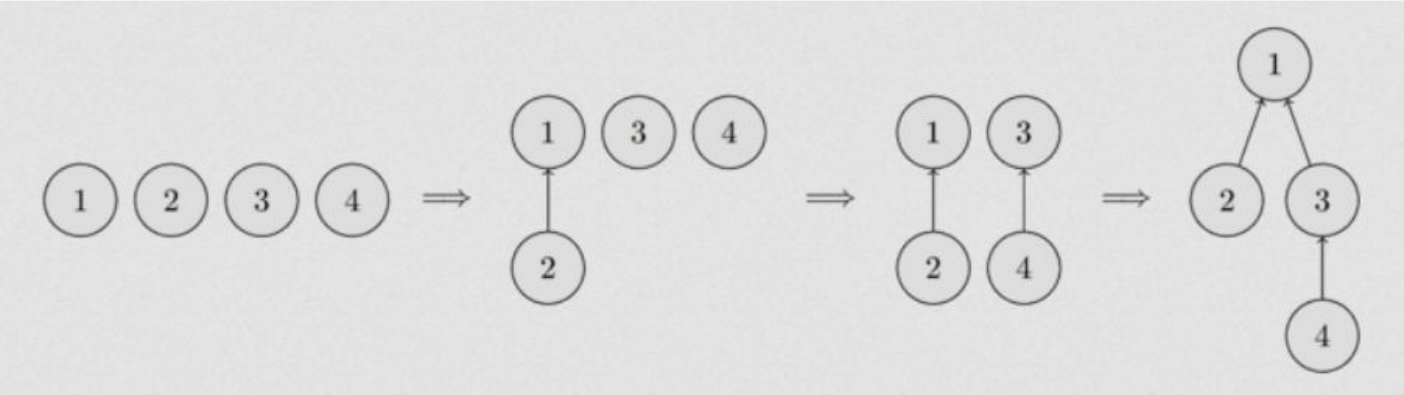
DATA STRUCTURES USED

We have an array of disjoint sets of size 'k' where each set represents a hashmap of a particular source. Initially the rank of each set is 0 and every set is the parent of itself.
To merge two disjoint sets, we update the parent pointer of the set having lower rank to the one having a higher rank. If the ranks of both the sets to be merged is equal, then the parent pointer of one set is updated to the other and the rank of the parent is increased by 1. This method is called the rank heuristic method in disjoint set union



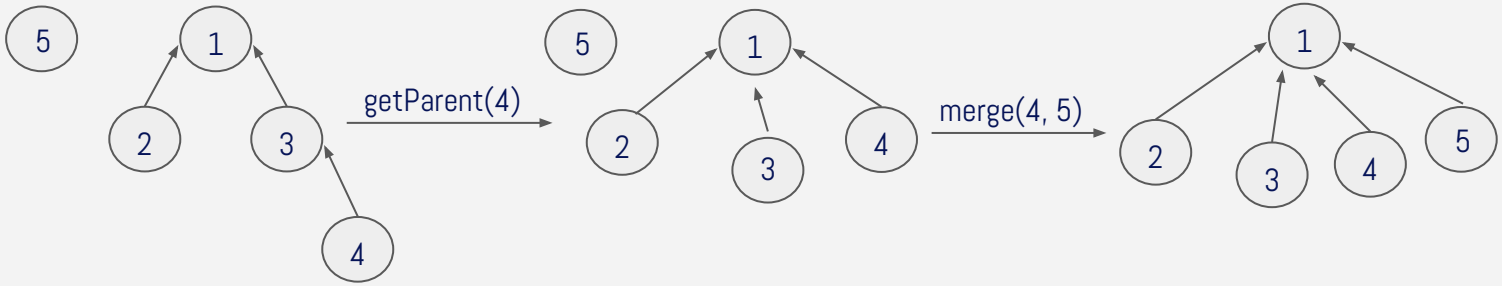
DATA STRUCTURES USED

Along with the rank heuristic, a path compression heuristic is also implemented to make the operations on disjoint sets efficient (in particular, the union operation)
Some merge operations have been performed on the disjoint sets:



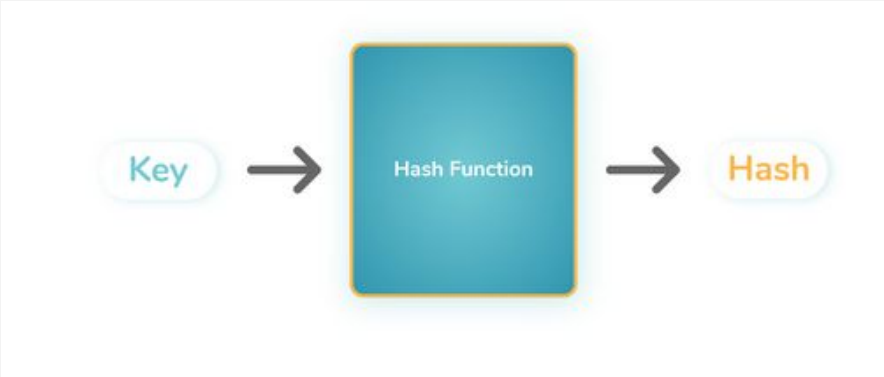
Now if we call getParent(4) for the 4th set, it finds the root of the tree which 4 is a part of and simultaneously updates the parent of 4 to be 1. This effectively looks like path compression and it reduces the subsequent time complexity to a great extent. In this way, the depth of any tree in the forest will not be large, making the subsequent merges efficient.

In continuation to the previous example, let us see how path compression will make the program more efficient.
Let's say we want to merge the 4th set with a new set say, 5



It can be seen that the path from 4 to its root has been compressed. Now, the subsequent calls to getParent(4) (if any) will be faster as the whole depth of the tree will not have to be traversed to reach the root node, leading to faster union operations.

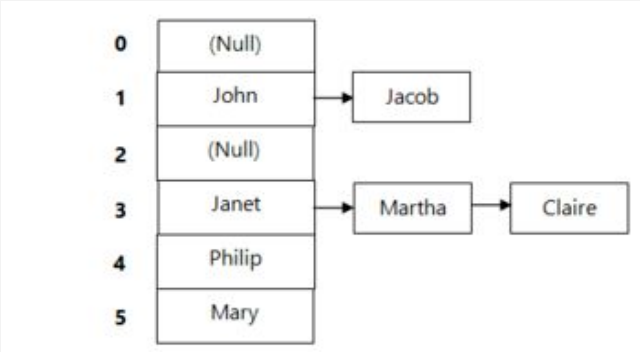
Now, let's discuss about the hashmap / hash table. Each disjoint set is an individual hashmap which stores key value pairs, where name maps to the number of the contact.
A hashmap is a data structure which maps a key to its corresponding value using a hash function. A hash function takes as input the key and generates an integer in a desired range at which the value is stored. A good hash function is such that the probability of getting same hash value for different keys is very low, or the probability of collisions is low. Such a function enables us to search, insert and delete elements in the hashmap in nearly constant time (amortized constant time complexity).



There are mainly two methods to handle collision:

- Separate Chaining
- Open Addressing

In this project, we have used the separate chaining method. In this method we maintain an array of linked lists at each index. Whenever two different keys generate a same hash value, we add the key value pair to the linked list of that particular hash index.
Thus, it can be seen that less collisions will ensure small chain length and hence will ensure constant time operations in the average case.



In this project, the following hash function is used which generates an integer hash value on passing a string (name of the person):

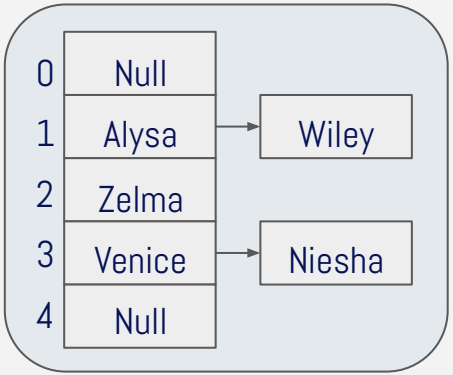
Let the string name be of size n.

hash value = $\sum_{i=0}^{n-1} name[i] \times p^i \ mod \ m$

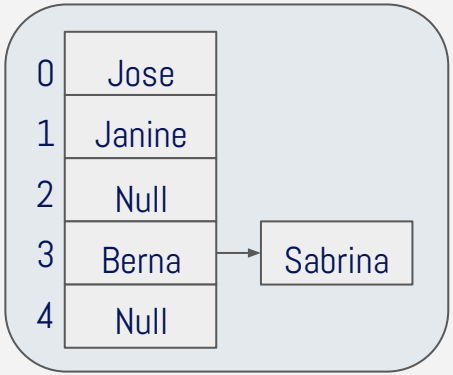
where, name[i] is the value corresponding to each letter defined as follows:
First convert each letter in lowercase and then,
a = 1, b = 2, c = 3 .... and so on till z = 26

p is a prime number. In our case, we have set p = 31
m is the size of the hashmap which should also be a large prime number to avoid collisions as the probability of collision is nearly 1/m. Here we have set m as 101 as the dataset is small.

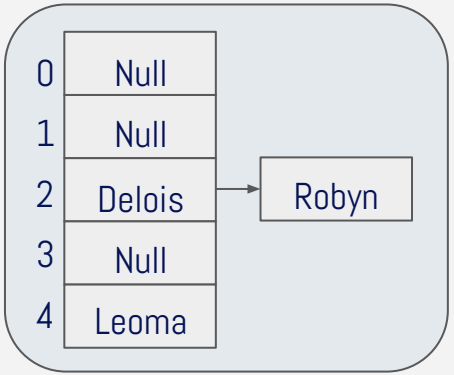
Thus, the finalized data structures used in developing the Universal Phonebook is an array of hashmaps where each element of the array is itself a disjoint set data structure which supports efficient set union operation.



Disjoint Set 1



Disjoint Set 2



Disjoint Set 3

hashFunction() :

// Here we are assuming that the string length which is to be hashed is at max 20.

hash value = $\sum_{i=0}^{n-1} name[i] \times p^i \ mod \ m$

Since the string length is a small constant, the time complexity of calculating the hash value for a particular string is almost constant.

∴ Time Complexity ≈ O(1) in the average case and the worst case

insertContact() :

// This is a standard hashmap insert function which inserts the pair (name, number) at an index specified by its hash value.

Time Complexity = Time Complexity of calculating hash value + inserting a node in a linked list

// Since the probability of collision is very less, the chain length (length of a linked list) is small thus, inserting a node in a linked list is nearly a constant time operation

∴ Time Complexity ≈ O(1) in the average case

TIME COMPLEXITY ANALYSIS

search() :

//This is a standard hashmap function to search an entry with the given key

Time Complexity = (Time Complexity of calculating hash value + searching a node in a linked list) for each disjoint set / source

// Searching a node is nearly a constant time operation for small chain lengths

Time Complexity ≈ (O(1) + O(1))*k (where k is the number of sources)

∴ Time Complexity ≈ O(k) in the average case

deleteContact() :

// This is a standard hashmap function which deletes an entry with the given key (contact name).

Time Complexity = Time Complexity of search() + deleting a node in a linked list

// Deleting a node is nearly a constant time operation for small chain lengths

∴ Time Complexity ≈ O(k) in the average case

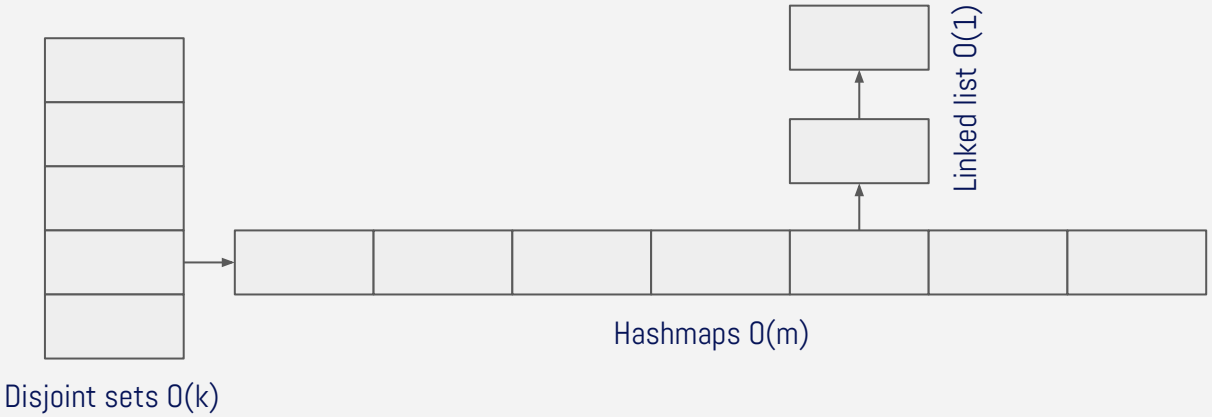
merge() :

//This is a standard disjoint set union function which uses rank and path compression heuristic

∴ Time Complexity ≈ O(1) in the average case

// We have used an array of size k (where k is the number of sources).
// Each element of the array is itself a hashmap implemented using an array of size m
// Each element of the hashmap array is a linked list itself, but since we have ensured less number of collisions by using a good hash function, the length of the chain (linked list) can be said to be approximately constant.

∴ Space Complexity = O(m*k) in the average case



MD AREEB HUSSAIN 19EC10043
AKSHAY JAIN 19EC10003

THANKS!