Genealogy Data Representation

Deddy Jobson

To store the genealogy tree effectively, there are some constraints which one has to consider. They are:

- Space complexity
- Time complexity

There are many implementations, each with their own strengths and weaknesses. But before I discuss some of them, I must make the following assumptions.

- If a person sees atleast one (not all) of his great grandchildren, it means he has lived to see his grandchildren.
- There is only one original ancestor given in the input file.
- There is no person in the genealogy who isn't a parent or a child of any other person in the genealogy. (I'm looking at you, Sarah)

Binary Tree Implementation

A binary tree can be used, provided every parent's children are stored in an array of strings. This, however, is a weak method of implementation because you will not have instant access to a person's relatives. The person's name should be used to decide his position in the binary tree as any person's memory's access would only be done through his name. So, the time taken to access a person from his name is of $O(\log n)$ Also, for the worst case, the tree could become equivalent to a linked list which has very bad implications. In fact, using a linked list is the worst possible way to store the genealogy.

2-3 Tree Implementation

A 2-3 tree is similar to a binary tree, the main difference being that it will always be balanced and so it will never be equivalent to a linked list. However, It requires lots of extra space as the people are only stored in the leaves. There are better implementations which take up less space and take less time.

General Tree

This is the most intuitive method to store a genealogy as the parent child relations are actually made in this graph. This allows us to go down a generation instantly and would fill most of our needs. However, finding a random person with a name would take $O(n)$ time which is fine for small inputs like in the input file, but consider for an extreme example, the genealogy of Confucius. It has 86 generations and over 2 million members living today. For a genealogy like that, one would require a faster way to access a randomly chosen person.

General Graph Implementation

This is one of the better implementations if space is not a constraint. Basically, it is a combination of a binary tree and a genealogy tree. This allows a parent to access his children instantly while also allowing access to a person from his name in $O(\log n)$ time as long as the binary tree is balanced.

My Implementation

The best implementation would be a general graph with a hash table so that one can find a person from his name in $O(1)$ time. This method takes about as much space as the previous implementation but is much faster, especially if you consider the worst case complexity. I have made a combination of a general tree with a binary tree and a hash table to see which is faster.

In my implementation, I have stored the edge information, which is age of parent when the child was born with the child itself. To determine the original ancestor, I need to look for the one person whose age of parent variable has no data. This take $O(n)$ time, whether I use hash or binary tree.

To find out number of descendants of a person, one has to go through every single descendant. If the total number of people is n and the number of generations is k, this algorithm would have a time complexity of $O(n)$. To put it in terms of k, we need to assume the average number of children per person is c. Now we get

$$\begin{aligned} \log_c n &= k \\ n &= c^k \\ T &= O(c^k) \end{aligned}$$

Simply put, finding the number of descendants for a person takes exponential time with respect to number of generations. For huge genealogies, this can be very costly. So I used dynamic programming and stored the number of descendants for all people in $O(n)$ time in an array which is linked to a person through his ID. So for future function calls, the only time spent is in accessing a person's ID from his name which takes $O(\log n)$ time with a balanced binary tree or $O(1)$ time with a hash table.

To find out the number of people who live to see their grandchildren, I must traverse only along the genealogy to avoid any overlap. Each person has direct access to his children which means he will also have direct access to to all of his descendants. As each person can be a great grandchild of at most one person, there will be no overlap. For each person, the time complexity for checking if he lives to see his great grandchildren T is

$$\begin{aligned} T &= O(c^3) \\ n &= c^k \quad => \quad c = n^{\frac{1}{k}} \\ T &= O(n^{\frac{3}{k}}) \end{aligned}$$

But to calculate the total number of people who see their descendants we have to traverse the entire graph. I made a breadth first traversal and so the total complexity T_{tot} is

$$T_{tot} = O(n^{\frac{3+k}{k}})$$

To get the list of a person's great grandchildren, one must pass through all of his grandchildren. This is similar to checking if a person lives to see his great grandchildren which has been already discussed. So we get the same time complexity of

$$T = O(n^{\frac{3}{k}})$$

.

0.0.1 Comparison

The following table shows how the implementations fare against each other in certain tasks which were asked.

Task	Binary or 2-3	Genealogy tree	Graph graph	Hash table
Original ancestor	$O(n)$	$O(n)$	$O(n)$	$O(n)$
# descendants first time	$O(n \log n)$	$O(n)$	$O(n)$	$O(n)$
# descendants later	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
Person's details	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
# see great grandchildren	$O(n^{\frac{3+k}{k}} \log n)$	$O(n^{\frac{3+k}{k}})$	$O(n^{\frac{3+k}{k}})$	$O(n^{\frac{3+k}{k}})$
List of great grandchildren	$O(n^{\frac{3}{k}} \log n)$	$O(n)$	$O(n^{\frac{3}{k}})$	$O(n^{\frac{3}{k}})$

From the above table, we can clearly see that the hash table is the fastest implementation for all of the tasks mentioned. The general graph method comes a close second losing only in random access of a person from his name. For smaller genealogies though, the $\log n$ factor won't matter much, and the general graph's speed can be comparable to hashes. Using a binary tree alone or a genealogy tree alone is not a very good implementation strategy.

I have compared the general graph and hash table implementations experimentally by recording the run time of some of the methods.

Task	Graph	Hash
# descendants 1st time James	2.05μs	2.13μs
# descendants 1st time Steven	0.70μs	0.64μs
# descendants average James	0.06μs	0.15μs
# descendants average Steven	0.19μs	0.15μs
Person's details James	0.07μs	0.15μs
Person's details Steven	0.20μs	0.15μs
List descendants James	0.43μs	0.51μs
List descendants Lisa	0.51μs	0.52μs
List descendants Susan	0.23μs	0.16μs

According to the table, the graph seems to be faster than the hash, especially for James. This is because James happens to be the root of the binary tree part of the graph and so it's access time will be faster than even the hash. However, the hash dominates when the task only requires person access and someone down the genealogy, like Steven is chosen. The hash's constant time can be clearly seen in the time taken to fetch a person's details. So to summarise, if space is your biggest constraint and you don't mind about speed, a binary tree is a good option. If you don't mind a little additional space, a general graph which consists of a genealogy tree and a binary tree is a very good option as it is much faster than a binary tree. But if you want the fastest implementation and don't mind about space, a genealogy tree coupled with a hash table is the best option.