University of California Los Angeles

Computer Science 32: Introduction to Computer Science II

Unhinged

March 8, 2022

Haoxuan Li

Contents

List of Figures

Declaration

I, Haoxuan Li, hereby honestly declare that all work within this project is my own with the exception of those referenced in the acknowledgements.

Acknowledgements

I would like to thank Professor David Smallberg for designing the specifications as well as Professor Carey Nachenberg for providing pointers during the development process for the Radix Tree. Additionally I would like to acknowledge the Department of Mathematics at the University of California San Diego for providing a foundation to this LaTeX document.

Abstract

Unhinged is a text-based dating app that pairs user data with pre-set attribute combinations to generate matches at the fastest rate possible. The Unhinged program uses a Radix Tree, a type of binary search tree that allows the search process to exceed the speeds of a generic binary search tree. The goal of this project was to build a search and insert process that could rival the speeds of generic sorting algorithms when faced with databases with 10,000+ entries and subentries per entry.

Overview

The Unhinged software collectively compiles user data and searches the created data structure to create matches and output those matches in a readable list of contact information for the user. A person's profile is divided into 4 main parts. The name, email, number of attributes, and a list of attributes that range from jobs to hobbies to traits. A separate translator document matches these traits to each other in order to signify compatibility. The matching program is especially adaptive as additional profiles can be added as long as their personalities are accounted for in the translator document.

When discussing attribute pairs, each member can have dozens of attribute-value pairs. For example, for the pairs "hobby" $\to$ "hair braiding" and "occupation" $\to$ "pet stylist", "hobby" would be an attribute, and "hair braiding" would be its corresponding value. Additionally, if members searching for a compatible pair have an attribute-value pair of "hobby" $\to$ "hair braiding", then this might be translated to a compatible attribute-value pair of "physical attribute" $\to$ "long hair", since they would likely want someone who has long hair as a partner.

The speed of this program is leveraged by a data structure called a radix tree. Compared to other binary trees, the radix tree is especially efficient at long sets of strings. Unlike regular trees, the radix tree instead spreads nodes based on the divgenceny if a word which lessens the depth of the entire tree at the expense of potential sparseness. In other words, it takes less time to traverse the tree as there are generally less paths that the computer must traverse. The radix tree has the benefit of inserting in O(K) time where K is the maximum key length of a new item being inserted into the radix tree and searching in O(K) time where K is the maximum key length of the items in the Radix Tree. It is also important to note that a select deletion function on the radix tree would not be necessary as the tree is constantly being destroyed and rebuilt every time the program is run. Therefore in order to update a specific feature of a profile, the user would rerun the program to compile changes.

Since words are split in a radix tree, inserting into the tree is slightly more complicated than a trie format, or when nodes are only filled with a single letter. For example, say "racecar" was inserted beforehand and someone wanted to insert "racer", the data structure would create a preceding node named race and nodes follow with "car" and "r" with end signifying nodes to remind the program that it has reached the end of a node path. Additionally, if the word "big" was inserted and the word bigger needed to be placed in, then the radix tree would have to split big into a proceeding end node and "ger" node. Likewise, if "bigger" was already inserted and "big" needed to be placed in, then it would have to recognize when to split the larger word and create the previously described end nodes. On top of all this if a pre-preceding word was the split, then the entire tree would have to shift downwards according to that node. This work required here is best accomplished through the use of recursion as finding the right method for insertion at every time would be impossible without the computer knowing what has happened previously on the tree.

Experiment

1 Radix Tree & Edge Cases

The below radix trees can are formatted with a root node and ending in a end character which in this case will be displayed as a *period*. An end character is used in this case as it is able to differentiate when compound words are split up such as firework, fire and work. Without the end character, fatal errors can occur where the program continuously searches for the end of a word after a split addition in Figure 3. Possible issues can occur when inserting into a radix tree. The most effective way to tackle the amount of edge cases that could arise at any moment is through recursion. For example, Figure 2 shows a case in the Radix Tree where a split in test produces another string of nodes. This in turn creates a new period for the other end. In order for the Radix Tree to perform correctly, the program must recognize that it split at test and return ti adding an additional period after producing the *tester* line.

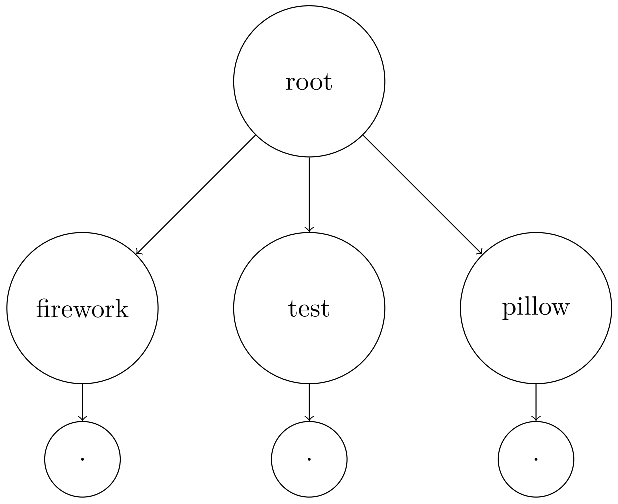


Figure 1: Typical Radix Tree with no edge cases.

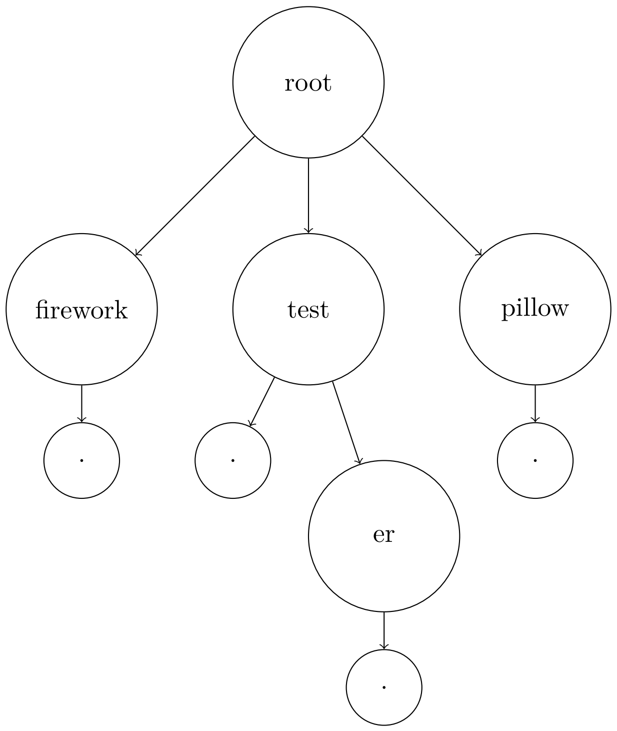


Figure 2: Similar Radix Tree with proceeding additional node.

Figure 4 shows an interesting interpretation of the previous Figure 2 and Figure 3. Instead of breaking off into a end notation, the previously present word pillow has become pilates and pillow. Both of these words share the same initial letters and therefore for the radix tree to be most effective, the split occurs at the point where they differentiate. These discrepancies in insertion become *if-statement* checks within the recursion. This type of split in Figure 4 is the radix tree's advantage over the typical binary search tree. Since many traits and hobbies share similar string formatting, the radix tree is a likely one of the best way to handle information in dating apps.

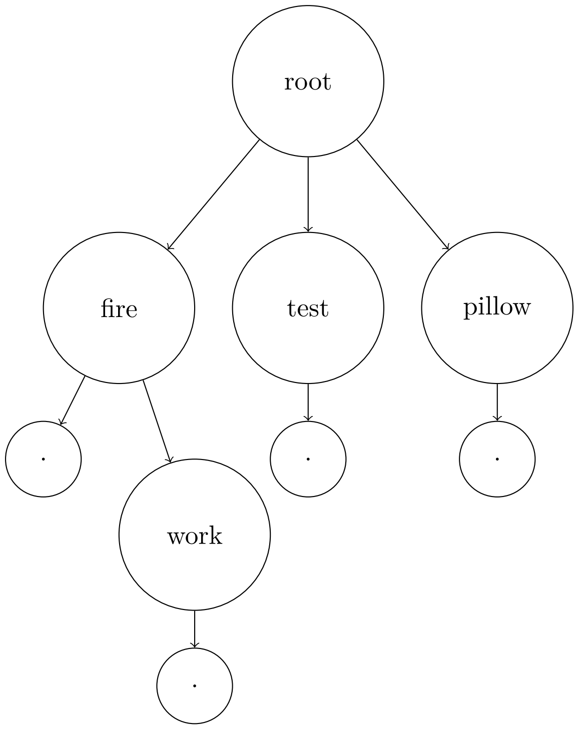


Figure 3: Radix Tree with preceding additional node.

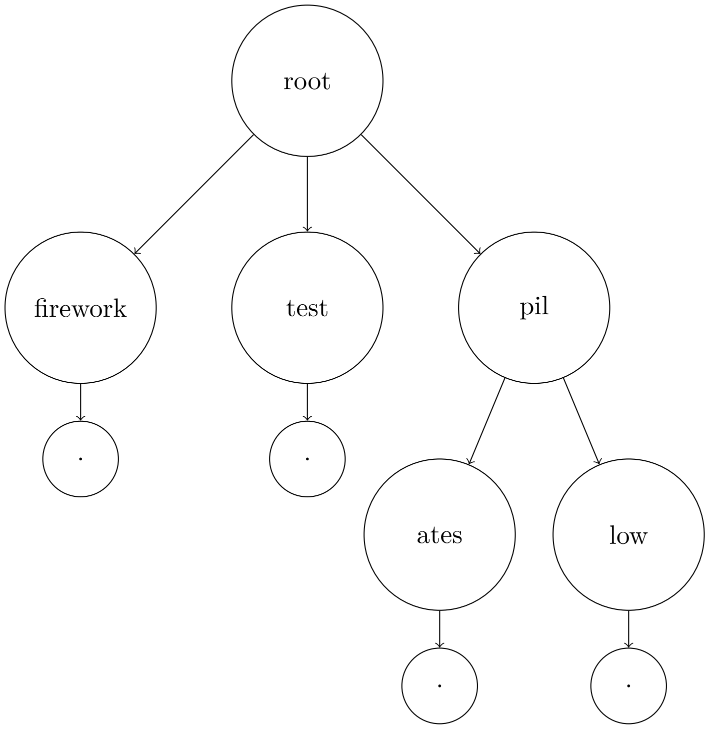


Figure 4: Radix Tree with preceding split additional node.

If Figure 4 existed previously and an additional insertion were to be made with a word that encompassed less than characters than the previous split, then an earlier split would have to be created and the previous split would have to adjusted so that the new split could include as many possible letters as to limit unnecessary node creation. Figure 5 shows how the this complicated split could happen with the addition of *pilot*. As mentioned before, recursion would be the best as traversal of binary trees are best done if the program remembers the situation at which it split.

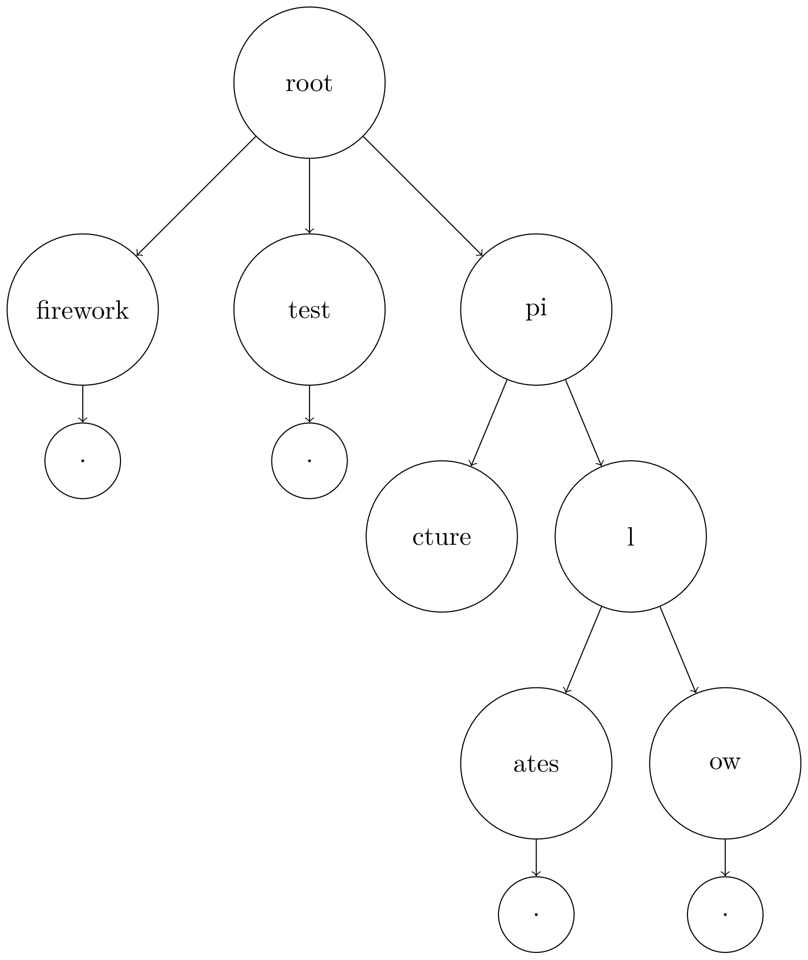


Figure 5: Radix Tree with pre-preceding split additional node.

2 Application of Radix Tree

One example of the Radix Tree use is in Figures 6-8. A person's profile would be stored in a binary tree format. Additionally, the code will also make reference to radix trees in terms of relating a person's additional information such as email, name, etc. to their attribute-value pairs. This in turn will be compared to a separate radix tree where attributes are compared to each other for compatibility. This way, although it could take a little longer to insert information into each separate radix tree, the lookup/search time would be always O(n) regardless of how much more information is placed in by users. Again, the benefit of the radix tree can be found in Figure 8 where *reactive* and *regretful* are split. This eliminates a possible one extra node that could have been create if traits were stored in a normal trie.

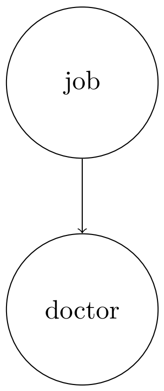


Figure 6: Typical Radix Tree of a profile's job.

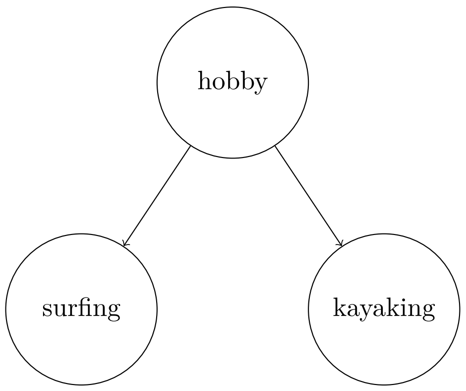


Figure 7: Typical Radix Tree of a profile's hobbies.

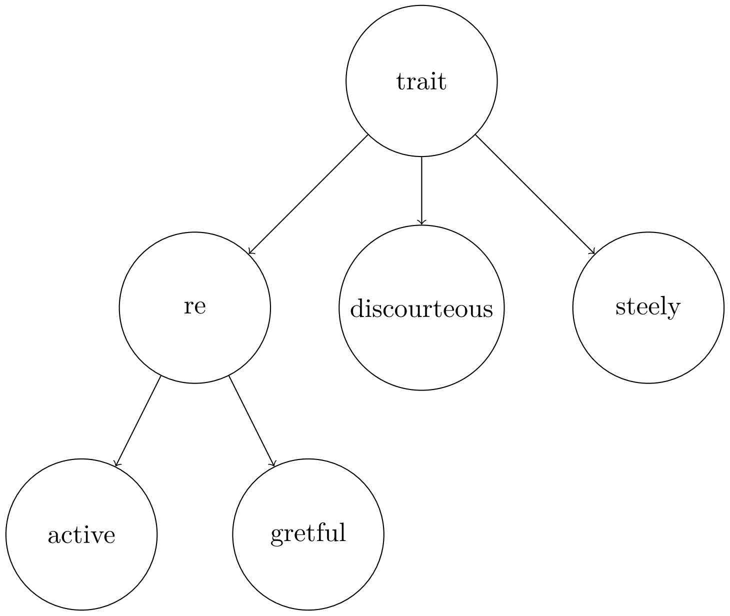


Figure 8: Typical Radix Tree of a profile's traits.

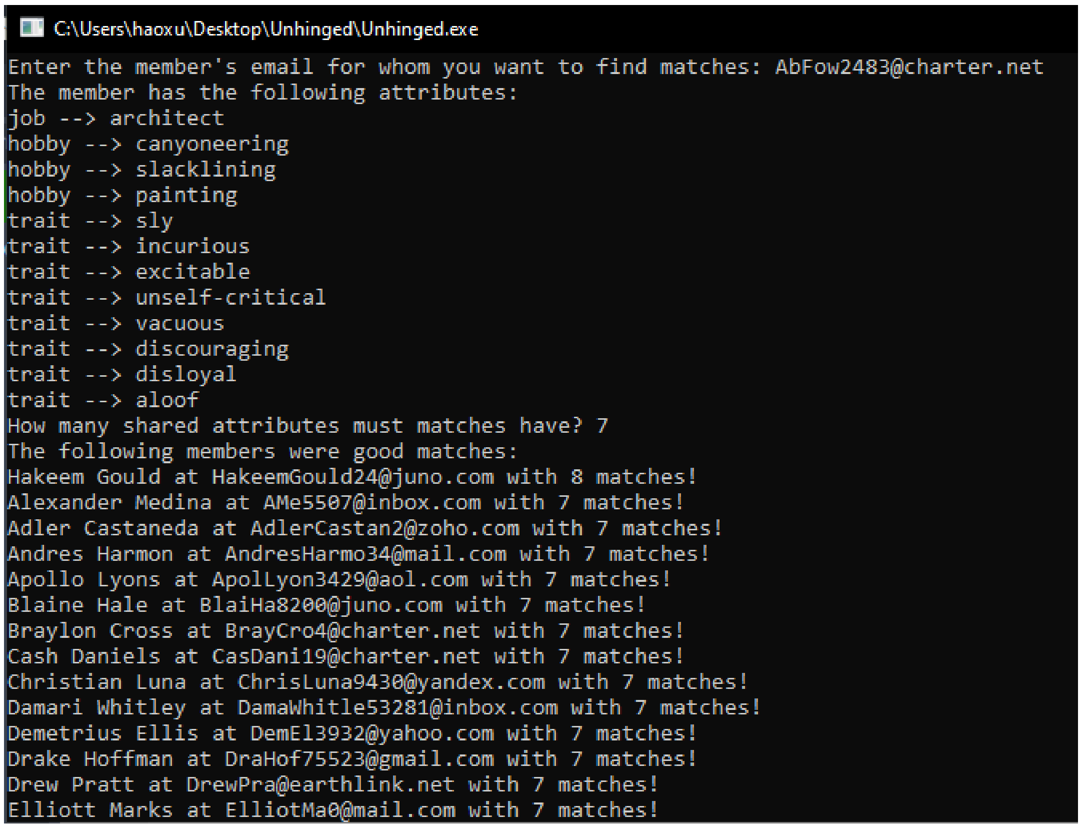
Conclusion

This project involved several significant data management techniques. A member of Unhinged has their data translated into a profile class that stores their attributes in an easily accessible Radix Tree. This type of data structure has profound uses in constructing arrays with keys mostly expressed in the form of string. Furthermore, a more modern use case of this type of data sorting is in the field of IP routing. Since IP addresses are in most cases longer values with few exceptions in the numbering, the radix tree serves as an especially effective way to produce faster look up times as well as space complexity. Like many other data structures, the radix tree is not a one solution that solves all needs for data allocation, instead it has a purpose and therefore should only be used when applications fit the Radix tree sorting methods. For example, one could argue that hashmaps with a time complexity of O(1) in best cases would perform exceptionally better to the recessive methods used in typical binary search trees. However, hashmaps struggle as their buckets begin to fill beyond a certain point (although an optimized hashmap could perform at standard rates). The application of the Radix tree in this sense allows the program to be more versatile and handle even more information. Overall, it is more user friendly and future proofed as it allows back-end developers to allow more user data without worrying about composing speeds.

Future Scope

The raw algorithms part of this project was a complete success. I believe that the file containing the Radix Tree can be reimplemented into several other applications without flaw as it mostly functions as a pre-made binary search tree with the exception of pin-point deletion commands. There is also evidence to believe that the program outside the Radix Tree has potential to fail the deletion of the Radix Tree afterwards and therefore might exhibit memory leaks. It seems it has something to do with how profiles are handled after the program is closed. Disregarding the memory leaks, the program at surface value functions as according to specifications and overall exhibits similar behavior to programs without memory leaks. A future improvement on this project would be to address the potential of constant use breaking the program due to memory leaks.

References



```
C:\Users\haoxu\Desktop\Unhinged\Unhinged.exe
Enter the member's email for whom you want to find matches: AbFow2483@charter.net
The member has the following attributes:
job --> architect
hobby --> canyoneering
hobby --> slacklining
hobby --> painting
trait --> sly
trait --> incurious
trait --> excitable
trait --> unself-critical
trait --> vacuous
trait --> discouraging
trait --> disloyal
trait --> aloof
How many shared attributes must matches have? 7
The following members were good matches:
Hakeem Gould at HakeemGould24@juno.com with 8 matches!
Alexander Medina at AMe5507@inbox.com with 7 matches!
Adler Castaneda at AdlerCastan2@zoho.com with 7 matches!
Andres Harmon at AndresHarmo34@mail.com with 7 matches!
Apollo Lyons at ApolLyon3429@aol.com with 7 matches!
Blaine Hale at BlaiHa8200@juno.com with 7 matches!
Braylon Cross at BrayCro4@charter.net with 7 matches!
Cash Daniels at CasDani19@charter.net with 7 matches!
Christian Luna at ChrisLuna9430@yandex.com with 7 matches!
Damari Whitley at DamaWhitle53281@inbox.com with 7 matches!
Demetrius Ellis at DemEl3932@yahoo.com with 7 matches!
Drake Hoffman at DraHof75523@gmail.com with 7 matches!
Drew Pratt at DrewPra@earthlink.net with 7 matches!
Elliott Marks at ElliotMa0@mail.com with 7 matches!
```

Figure 9: A screenshot of the program after an email and amount of relatable attributes are typed in.

Appendices

To view the raw code, read the project specifications, or download the program for you own viewing pleasure, all details can be found at my GitHub.

`https://github.com/supercoder-hao/Unhinged`