RUTGERS, THE STATE UNIVERSITY OF NEW JERSEY

Word Auto Prediction

Data Structures and Algorithms ECE-573 Final Project

By:

Pranav Shivkumar (ps1029)

Naga Venkata Vihari Vinnakota (nvv13)

Swapnil Shashikant Kamate (sk2181)

Date: 04/28/2020

Contents

Abstract

In this project, we have covered mainly the analysis and complexity of the algorithm along with its application usage.

We have implemented word prediction at the basic level. We have used the trie tree and radix tree data structures for the implementation. This report gives a brief introduction to our project and the motive for the project. It later explains a few terminologies and their definitions which have been used in our project, followed by some of the basic operations on Trie and Radix trees which helps for better understanding of the data structures. We then provide information about the experiments that we have done on these data structures, which gives an insight to the analysis of the results of the experiments performed. After this, we discuss the time complexity calculated using the analysis and finally we throw some light on the real time applications of each tree followed by the conclusions of our experiments and the references we used for achieving the results.

Introduction

Today's world sees the use of the auto-complete feature in a lot of applications. Auto-complete, or auto-prediction, is the feature where a part of the word (substring) is entered and the most probable and sensible word is generated. For example, in Gmail, when a user types in a part of a sentence and based on the sentence he/she enters, the most probable end of the sentence is generated for the user's convenience.

Auto-prediction need not necessarily hold true for sentences only; it is applicable for parts of words as well. For instance, when a user types a letter or two in search engines like Google and Yahoo, related suggestions pop up almost instantaneously to help the user locate what he/she is looking for. The predictions generated by these search engines keep changing continuously based on the input entered by the user, until no match is found. This reduces the labor the user has to put in to manually search for whatever he wants; he just has to enter the first few keys and if a suggestion pops up that matches his requirement, he can click on the suggestion and get results immediately. If no suggestion pops up, he has to enter whatever he wishes to search for.

The feature of auto-prediction comes in handy for the user, since he can use the words that result from the predictions. This reduces the effort to be done by the user while writing a document as the user just needs to type the first few characters and the words starting with this character string will be auto-populated, provided the words are already saved into the structure. The user can then use these auto populated words.

In our project, we will be using the trie and radix trees as the underlying data structures to store the words that need to be predicted.

Definitions

Trie Trees

In simple words, trie trees, also called digital or prefix trees, are special data structures whose nodes store the letters of the alphabet used to store strings so that they can be visualized as graphs. They fall under the "tree" data structure category and are typically used to store a dynamic set or symbol tables, where the keys are usually strings. Using trie trees, search complexities can be brought to the optimal limit.

By structuring the nodes in a particular way, words can be retrieved from the structure by traversing from the root node of the tree down a branch path of the tree, towards a leaf node. Every node of the trie tree consists of multiple branches, where each branch represents a possible key. The last node of every key is marked as the end of the word. The trie tree was

originally used as a computing structure due to the comparatively reduced run time and memory used.

Structure of Trie Trees

The structure of the trie tree is as follows. Each trie has an empty root node, with links to other nodes - one for each possible alphabetic value. The shape and structure of a trie tree is like that of a typical tree data structure; a set of linked nodes connecting back to the root nodes of the tree. An important thing to note is that the number of child nodes in a trie depends completely upon the total number of values possible. For instance, if we are representing the English alphabet, then the total number of child nodes is directly proportional to the total number of letters possible. In the English alphabet, there are 26 letters, so the total number of child nodes will be 26. In this regard, trie trees are often used to represent words in an alphabet.

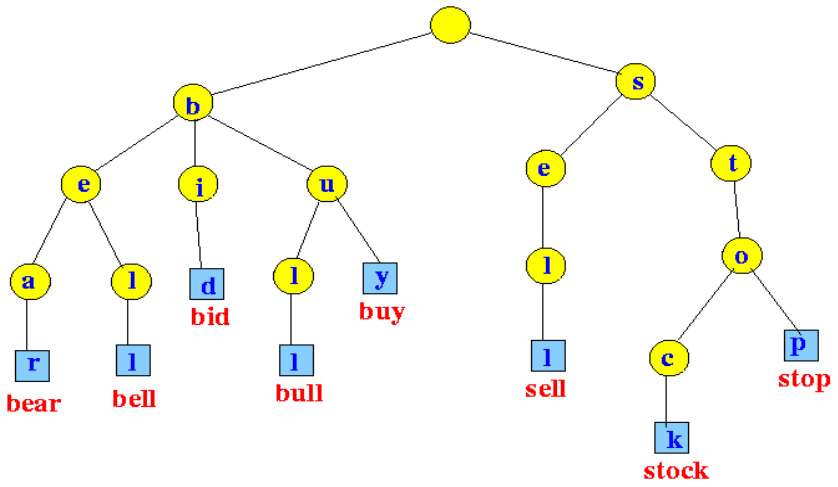


Fig 1: Structure of the Trie Tree

The above diagram depicts the structure of the trie tree. Here, the words *bear*, *bell, bid*, *bull*, *buy, sell, stock* and *stop* are present in the two trie trees. These words are inserted into the tree in such a way that 'b' and 's' are the root nodes for the first and second tree respectively. On traversing down each branch path of the trees, the words can be retrieved using the characters present in each node.

Operations on Trie Trees

The trie tree supports the following operations:

1. Insertion

   As the name suggests, insertion involves adding a new node containing the key into the trie tree. Every character of the input key is inserted as an individual node into the tree. Note that the children is an array of pointers (or references) to nodes in the next level of the trie tree. The key character acts as an index for the array of children. If the input key

is new or an extension of the existing key, we need to construct non-existing nodes of the key and mark the end of the word for the last node. If the input key is a prefix of the existing key in the trie, we simply mark the last node of the key as the end of a word. The key length determines the depth of the trie.

2. Searching

   Searching for a character in the trie tree involves traversing the tree from the root node towards a leaf node of the tree. The search terminates when the end of word character is reached or if the key does not exist in the trie tree.

3. Deleting

   As the name implies, deleting involves removing a node from the trie tree. This is done recursively in a bottom up manner.

Radix Trees

Radix trees, radix trie trees or compact prefix trees, are data structures that represent a spatially optimized trie tree. In these trees, each node is the only child that is merged with its parent. The edges of radix trees can be labeled with sequences of elements as well as with single elements. This makes radix trees much more efficient for smaller sets (especially if the strings are long) and for the sets of strings that share long prefixes. Radix trees are useful for constructing associative arrays (symbol tables) with keys as well as their corresponding values which can be expressed as strings.

The radix tree is easiest to understand as a space-optimized trie where each node with only one child is merged with its child. The result is that every internal node has at least two children. Unlike in regular tries, edges can be labeled with sequences of characters as well as single characters. This makes them much more efficient for small sets (especially if the strings are long) and for sets of strings that share long prefixes. Radix trees are much faster than the binary search tree (BST). There will not be any collisions like hashing in the radix tree so the worst-case time complexity will be reduced.

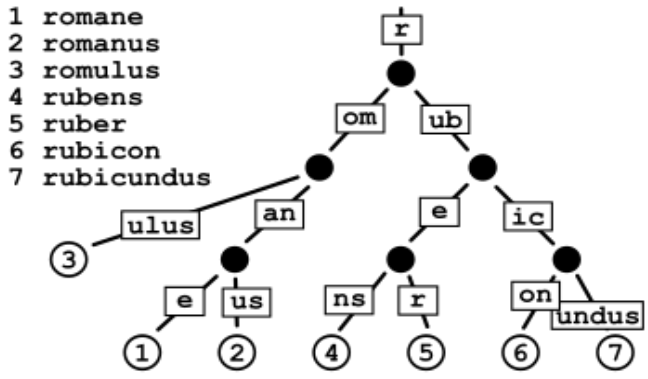


Fig 2: Structure of the Radix Tree

The above figure demonstrates the general structure of the radix tree. Analogous to the trie tree, the radix tree stores the words romane, romanus, romulus, rubens, rubicon, ruber and rubicundus in the tree. In this case, the substrings are stored in the nodes of the tree, so that on traversing down the branch path of the tree, the words can be retrieved.

Operations on Radix Trees

Radix trees support the same operations as the trie tree (since the radix tree is essentially a trie tree), namely insertion, deletion and searching, in addition to a few others:

Lookup: Determines if a string is in the set. This operation is identical to tries except that some edges consume multiple characters.

Insertion: Insertion adds a new string to the trie while trying to minimize the amount of data stored. We search the tree until we can make no further progress. At this point we either add a new outgoing edge labeled with all remaining characters in the input string, or if there is already an outgoing edge sharing a prefix with the remaining input string, we split it into two edges and proceed.

Deletion: This operation basically removes a string from the trie.. First, we delete the corresponding leaf. Then, if its parent only has one child remaining, we delete the parent and merge the two incident edges.

Searching: Searching operations include (but are not necessarily limited to) exact lookup, find predecessor, find successor, and find all strings with a prefix.

Find Predecessor: Locates the largest string less than a given string, by lexicographic order.

Find Successor: Locates the smallest string greater than a given string, by lexicographic order.

Experiments

In order to emulate the auto-complete feature, the first task was to insert words into the tree. In order to do this, a source was needed to import the words from into the nodes of the trees. For this, we created text files which had a list of words in such a way that each text file had:

- 10 words with no common prefix
- 20 words with 2 common prefixes
- 30 words with 3 common prefixes
- 40 words with 4 common prefixes
- 50 words with 5 common prefixes
- 60 words with 6 common prefixes

Once we had this set up, the next step was to insert the words into the tree. Initially, since both trees are empty, the first character and substring respectively are linked to the root node of the trie and radix trees respectively. Then, we searched for the prefix of the word to be inserted and if it was present, the characters/strings were inserted in their respective positions. We iterated over each text file and once all the words that were in the text file were inserted into the trees, the time taken for each tree and the number of nodes created for each tree was calculated.

The next task was to search for the word in the trees. To accomplish this, the user is prompted to enter the first few starting characters, or the prefix, of a word. Based on the prefix entered, the words corresponding to that prefix are searched in both trees. The search function traverses throughout the nodes of both trees and checks for the prefix. If found, the words beginning with that prefix are displayed to the user, otherwise the message *"The key or prefix is not present"* is displayed. The time taken to search for the words as well as the number of comparisons was calculated for each tree.

For our convenience, the insertion time, search time, number of nodes, and number of comparisons were written to text files so as to not lose the data as soon as the program execution ended.

Results

Based on the words present in the tree as well as the prefix entered by the user, the output was displayed on the screen. A sample of the output obtained for the trie tree is shown below:

```
Importing data from file 3common_prefix.txt
Time taken for inserting words into the trie tree is:  201884.0 ns.
Number of nodes in the trie tree =  535
Enter the Prefix you want: dis

Words starting with dis
dis
disapprove
dismiss
```

Fig 3: Sample Output for Trie Tree

The same output for the radix tree is shown below:

```
Importing data from file 3common_prefix.txt
Number of nodes in the radix tree is =  107
Enter string: dis
dis
disapprove
dismiss
```

Fig 4: Sample Output for Radix Tree

Clearly, the number of nodes created in the trie tree is much more than the number of nodes created in the radix tree.

The outputs obtained from the text files are shown below:

A. Trie Tree

```
7 18 32 45 60 97
54968.4 106929.0 201884.0 648644.1 532695.1 835522.9
10988.5 12987.0 17991.8 12994.9 18008.8 39993.5
76 232 535 983 1660 2383
```

B. Radix Tree

```
1 3 6 10 15 22
997800 2994400 2997300 8991600 11994800 21007800
49964.7 114932.4 202868.1 255847.8 365780.8 400793.3
11 44 107 202 339 485
```

In the above outputs, the first line corresponds to the number of comparisons done to retrieve the words from both trees, the next 2 lines denotes the insertion and search times in nanoseconds and the last line denotes the number of nodes created while inserting the words into the tree. Comparing the two, we can see that the number of comparisons and number of nodes created in the trie tree is greater than that of the radix tree, and the insertion time and search time for the trie tree is smaller compared to the radix tree.

The corresponding graphs obtained are shown below:

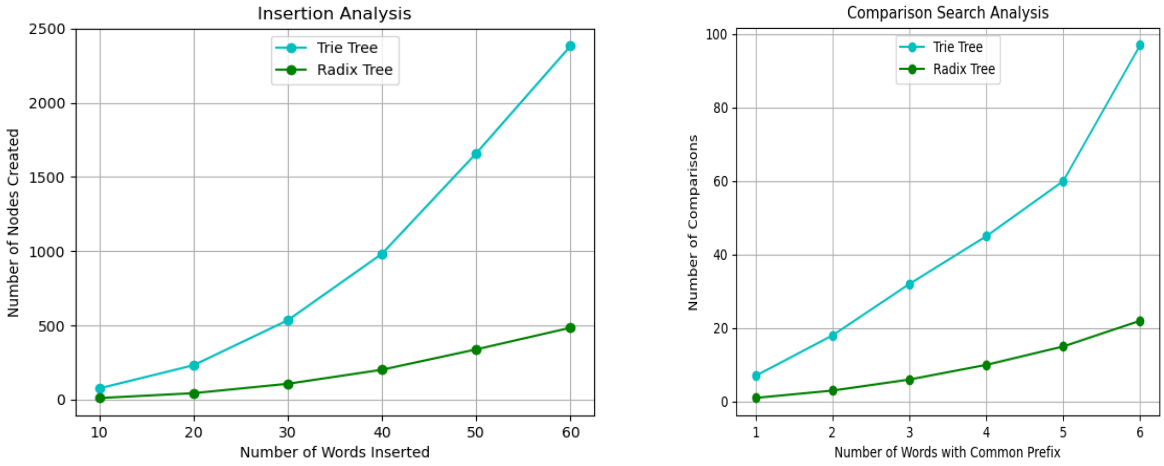


Fig 5: Graphical Results

For Trie trees, we can observe that the number of nodes created increases as the number of words increases. However, this increase is not linear as the insertion is dependent on the number of characters present in the word rather than the number of words inserted. In the case of the Radix tree, we can observe that the nodes inserted have a comparatively less count as the nodes of the tree are substrings and the tree compares these substrings with the string/word that is to be added. This implies that the number of compares is almost equal to the number of substrings present in the tree.

Similarly, we can observe similar results for searching in the Trie tree. The runtime is not dependent on the number of words, but the number of characters present in all the words that have the prefix searched.

From both the graphs in Figure 5, we can see that the radix tree utilizes a much smaller number of nodes in their trees when compared to the trie tree. In the first graph, it can be inferred that as the number of words to be inserted increases, the number of nodes created in the tree also increases exponentially, in the case of the trie tree. This is not surprising, as the trie tree can store only a single character in its nodes. For the case of the radix tree, the number of nodes created also increases but at a much smaller rate than the trie tree. Also, the number of comparisons increases exponentially in the case of the trie tree, while the radix tree increases at a much smaller rate than the trie tree.

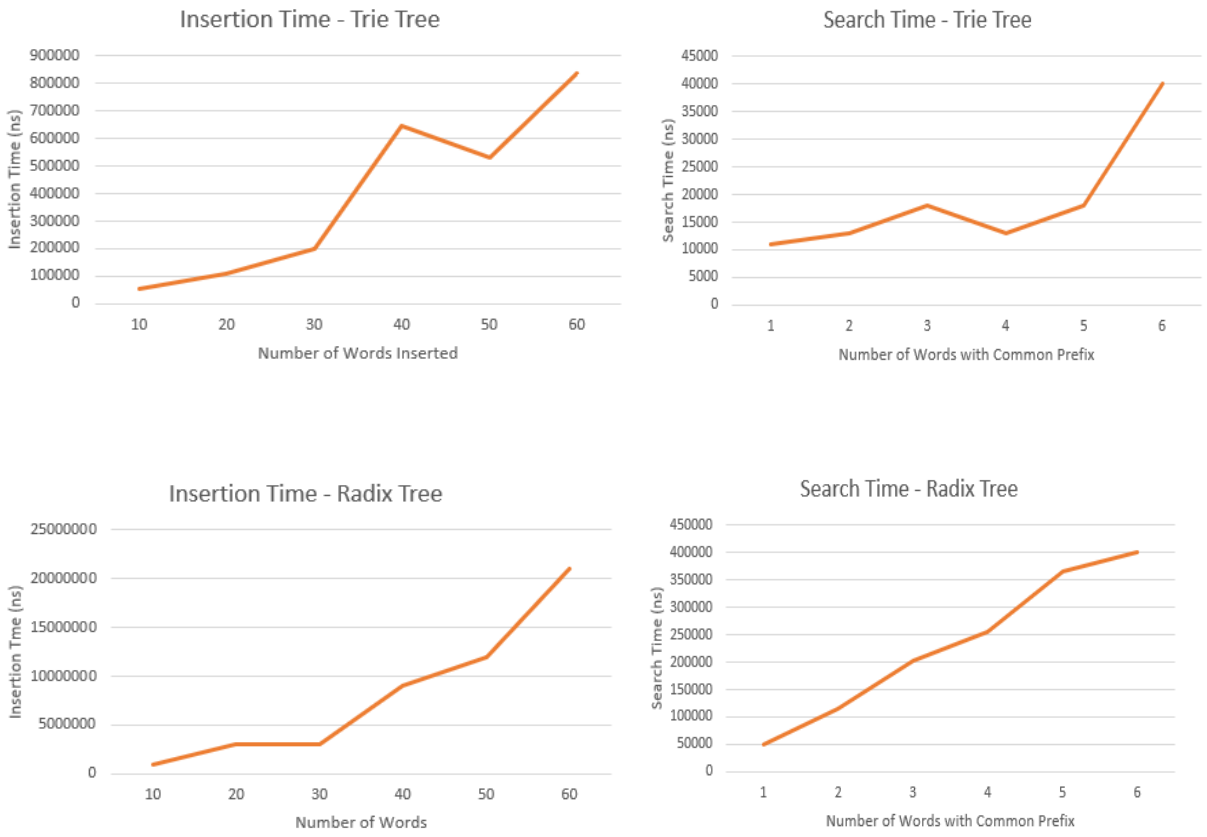


Fig 6: Insertion and Search Time for Trie and Radix Trees

In the above graphs, we plotted the insertion time (taken in nanoseconds) vs the number of words inserted into both trees. As can be seen for both trees, the insertion time increases with the increase in the number of words inserted. This is because the trie tree insertion is dependent on the number of characters present in the words inserted, and for radix tree, it is dependent on the number of substrings. Similarly, in the case of the search time for both trees, the time taken to search for the words based on the prefix increases with the number of words with a common prefix. From the graphs, it can be seen that the radix tree takes more time compared to the trie tree, for both insertion and search operations.

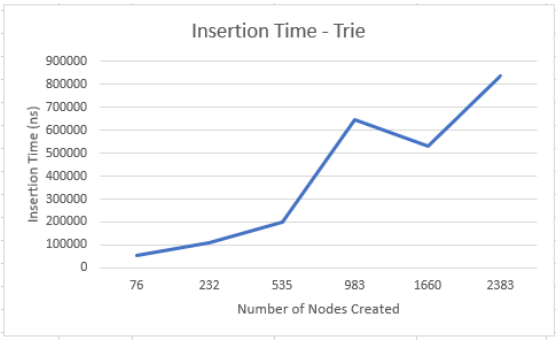
Analysis

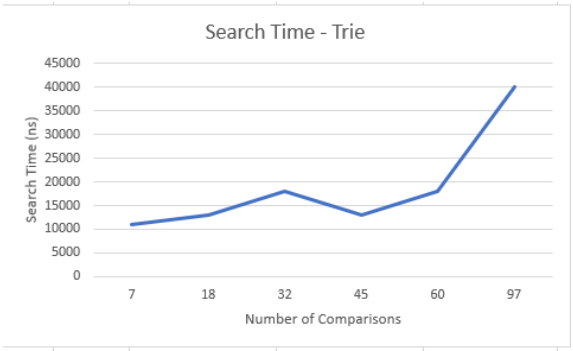
By inserting the same words for both trees, we were able to understand and compare the runtime of insertion functions in both the cases. We tried to search for a prefix of the word(s) we want and tried to print all the words starting with that prefix. This allowed us to analyze the time taken

for searching in the trees. By using the values obtained from the analysis, we were able to compare the runtime for search in both trees.

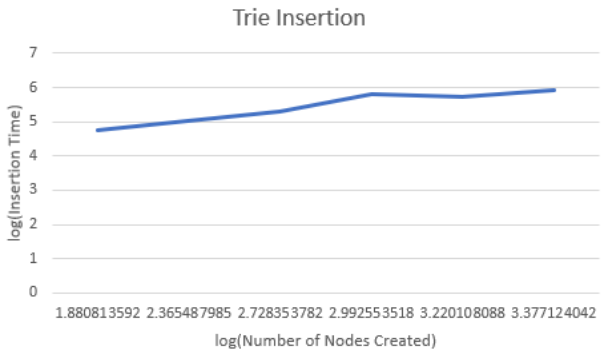
Using the graphs obtained by the experiments, we plotted the log-log plots for the insertion runtime vs the number of nodes created for both trees. The plots are shown below:

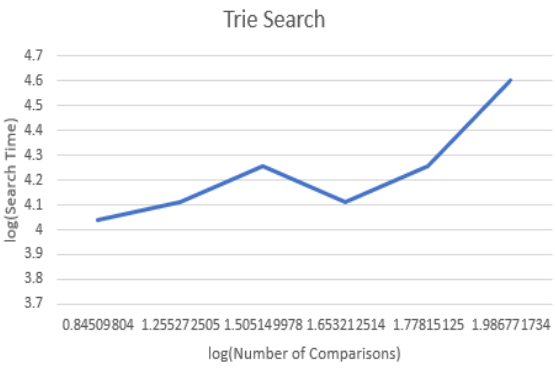
A. Trie Tree



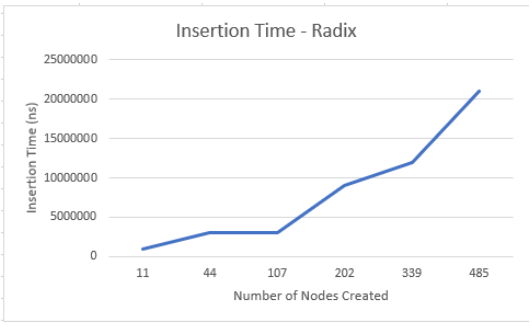


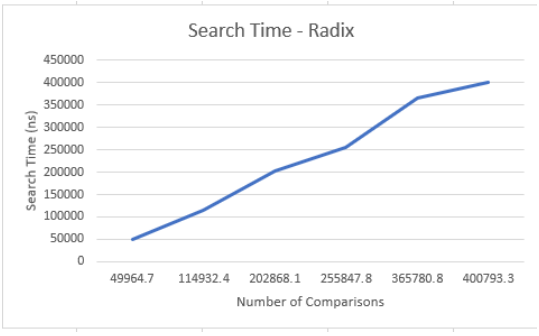
The log log plots are shown below:



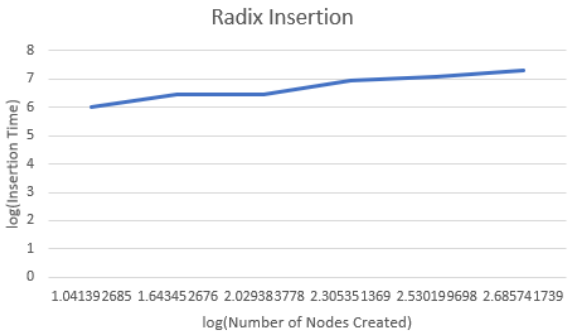


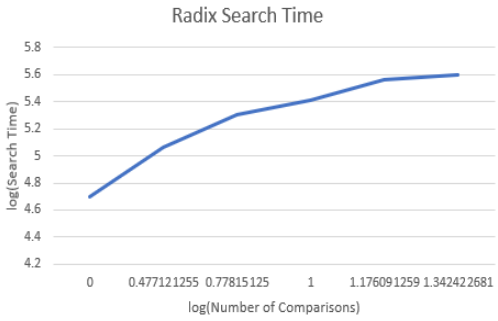
B. Radix Tree





The log log plots are shown below:





From these plots, the slopes of the lines were calculated to determine the time complexity, which were tabulated for the insertion and search operations as shown below:

Operation	Trie Tree	Radix Tree
Insertion	O(N^0.823)	O(N^0.944)
Search	O(N^0.39)	O(N^0.69)

Complexity

From the results obtained from the previous section, analysis, we were able to get the time complexity of both the trees for insertion and searching. For both the insertion and searching a word in a trie tree is of order O(k), where k is the number of characters in the words that are inserted or searched.

For searching and printing the words with common prefixes, similar logic holds true. The tree prints the words with common prefixes, which means that it traverses and prints all the nodes until an end of the word is reached. So, the runtime here will be a function of the number of words that contain the same prefix.

Hence, the insertion has a runtime of order O(S), where S is the number of the substrings present in the tree. The printing of the words with the searched prefix also has the same runtime as the insertion i.e. O(S).

Applications

Trie Trees

Some of the important applications of trie and radix trees are listed below.

Auto Complete: - The auto complete functionality is widely used in mobile applications and text editors. Trie trees are efficient data structures widely used for its implementation since they provide easy ways to search for the possible dictionary words to complete the sentence.

Spell Checkers: - Spell checking is a three-step process. Check if a word is in a dictionary, generate potential suggestions, and then sort the suggestions–hopefully with the intended word on top. Tries can also be used to store that dictionary and by searching the words over the data structure one can easily implement a spell checker in the most efficient way. Using trie, the lookup for a word into the dictionary not only becomes easier, but an algorithm to provide the list of valid words or suggestions can be easily constructed.

Longest Prefix Matching: - Also called as the maximum prefix length match, this refers to an algorithm used by routers in Internet protocol(IP) networking to select an entry from a routing table. One of the first IP lookup techniques to employ tries is the radix trie implementation in the BSD kernel. Optimizations requiring contiguous masks bind the worst-case lookup time to O(W) where W is the length of the address in bits. In order to speed up the lookup process ,multi-bit trie schemes were developed which perform a search using multiple bits of the address at a time.

Auto Command Completion: When using an operating system such as Unix, we type in system commands to accomplish certain tasks. For example, to see the list of commands like ls /usr/bin/ps* having prefix ps can be auto suggested by just pressing tab. We can simplify the task of typing in commands by providing a command completion facility which automatically types in the command suffix once the user has typed in a long enough prefix to uniquely identify the command.

Network Browser History: - A network browser keeps a history of the URLs of sites that you have visited. By organizing this history as a trie, the user needs only type the prefix of a previously used URL and the browser can complete the URL.

Radix Trees

Radix trees are efficient data structures, in terms of runtime as well as the memory consumed for implementation. Due to this, they find extensive applications in the following areas:

Searching Words in Dictionaries: Like trie trees, radix trees can also be used to search for words in dictionaries.

IP Routing: When a new IP address needs to be added to the tree, the tree which matches the prefix of the IP address is traversed.The new node can then be inserted which will contain the new IP address. In addition, radix trees are very useful for representing binary or hexadecimal numbers.

Information Retrieval: for searching words in dictionaries. In addition, they are used in IP routing, since these trees can be used to store the IP addresses efficiently in hierarchical order. Also, the most important application, which plays a very important role in our project, is the prefix search feature. This implies that all the words starting with a prefix can be retrieved.

Conclusion

From the results obtained from the experiments, we can conclude that both the radix tree and the trie tree exhibit a linear time complexity i.e. of the order around O(N). It was also seen that the time complexity obtained for the radix tree was slightly higher than the trie tree. As far as the memory requirements are considered, the radix tree utilizes less memory for its implementation compared to the trie tree implementation. The height of the Radix tree after inserting 'N' words is less than that of a Trie tree containing the same 'N' words. We can also see that these results are better than other data structures like arrays and linked lists which will take more time either for insertion or searching. This is because they consider the factors of random insertion and sorted insertion. In this regard, trees always give better results in these applications which has been proved by our results.

References

1. http://blog.xebia.in/index.php/2015/09/28/applications-of-trie-data-structure/
2. https://en.wikipedia.org/wiki/Radix_tree
3. https://en.wikipedia.org/wiki/Trie
4. http://cglab.ca/~morin/teaching/5408/notes/strings.pdf
5. https://xlinux.nist.gov/dads/HTML/patriciatree.html
6. https://www.geeksforgeeks.org/trie-insert-and-search/
7. https://www.geeksforgeeks.org/trie-delete/