RADIX TREE

Overview :

Computers need an efficient mechanism to store, retrieve, manage and process huge amounts of data. In the present scenario, where the amount of data is increasing at an unexpected high rate it becomes necessary to develop a proper algorithm for storing data in an optimal way. When the data is organized efficiently on the storage device then it can be accessed quickly for processing this reduces the access time and thus provides an enhanced functionality. Data Structures, thus when applied in the field of organising data (broadly Database Management) provide a scheme to organise data in such a way that it can be accessed and used quickly.

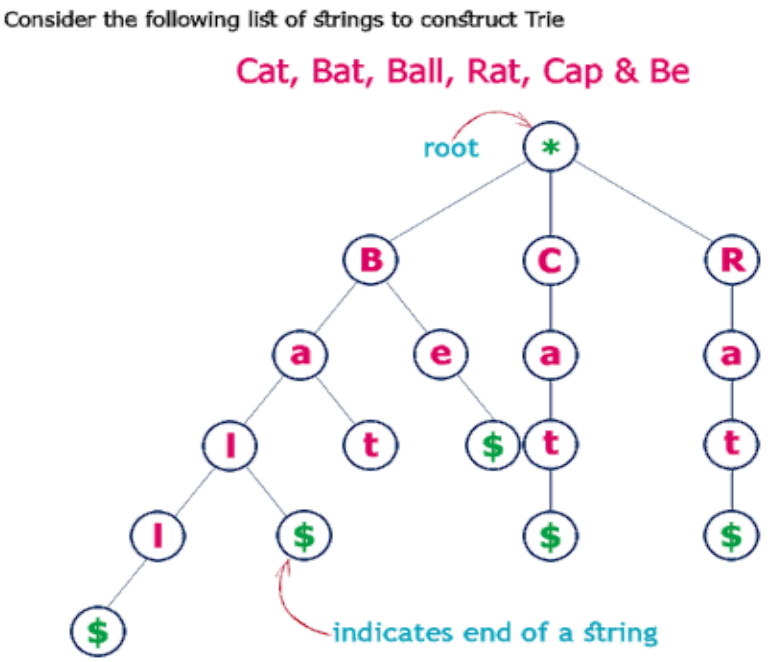
Traditionally to overcome these shortcomings, Balanced Binary Search Trees were used, but they have limited application in present system architecture which basically works on the concept of multithreading and SIMD instruction. Also it is seen that these data structures do not efficiently use Memory Caches. Often Hash Tables are used for memory indexing but they have also limited application in present architecture.

Later, Tries were introduced which are known for their fast and efficient data retrieval functionality. These Tries provide a search scheme where instead of comparing a whole search key with all existing keys , we could also compare each character of the search key. Radix Tree is basically a condensed adaptation of tries which are space optimised. Unlike regular trees, the key at each node is compared chunk-of-bits by chunk-of-bits, where the quantity of bits in that chunk at that node is the radix *r* of the radix trie. Edges in Radix Tree can be labeled with sequences of elements as well as single elements. This makes radix trees much more efficient for storing strings which share large prefixes.

Radix Tree :

A **Radix Tree** is basically a trie-based specialized set data structure that is used to store a set of strings.

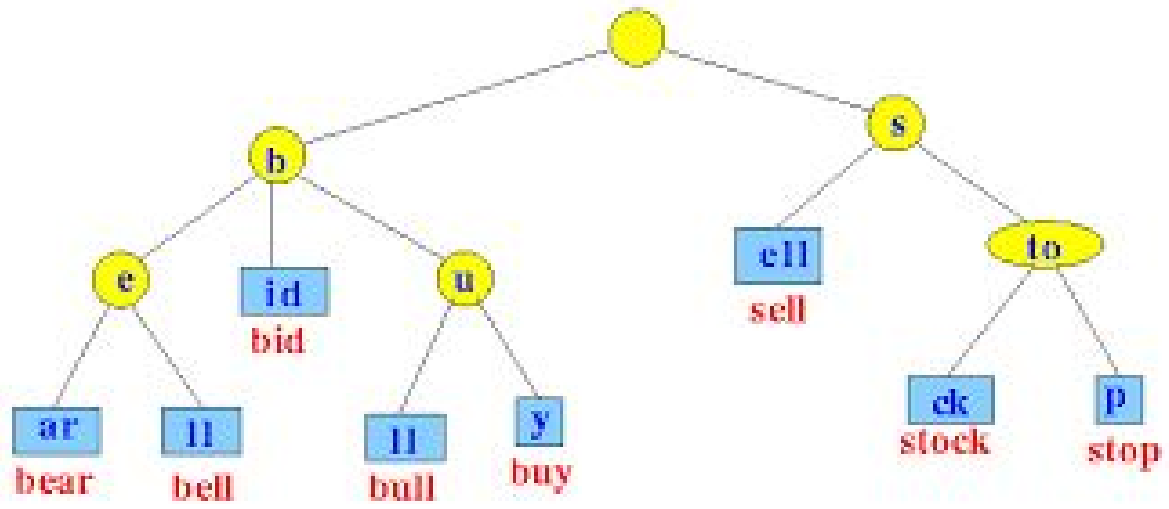
A trie is a tree where edges are labelled with characters. In general, **"Trie"** is an ordered data structure which is used to store associative data structures.It is well suited to string matching algorithms as its implementation is based on the prefix of string. Nodes of Trie can have multiple branches and each branch represents a possible character of keys. Last node of each key is marked with 'EndOfWord' that represents the end of the string. Tries can answer both point and range queries efficiently since keys are stored in lexicographic order. As they also implement associative arrays, tries are often compared to hash tables and moreover they are used at the replacement of Hash Tables at certain instances as it reduces the chance of collision and also there is no need of calculation of hash function that fit into the requirement. However the lookup time required in Trie is more as compared to Hash Table. Also it is observed that a Trie may require more space in memory as memory is allocated to a single character rather than a chunk of characters. This is the major drawback of tries. But despite this drawback they are preferred over Hash Tables as they can be employed when dealing with groups of strings rather than individual strings, enabling them to solve a wide range of problems.



Radix Tree can be seen as a solution to the drawback of Tries! Radix trees are basically and adaptation of Standard Tries with an extension of the idea that each node can store a chunk of characters, determined by certain rules, rather than a single 'character'.Similar to Tries, Radix trees are specialised data structure that are used to store multiple strings. This generally represents a space optimised structure in which all the nodes having one child is merged with its parent node. This leads to the compression and makes tree space optimised. Thus nodes in radix trees do not cover complete keys; instead, nodes in a radix tree represent partial keys, and only the full path from the root to a leaf describes the complete key corresponding to a value.

Radix Trees have two types of nodes: **Inner Node** and **Leaf Nodes**. Radix Tree has several features that distinguish it from various other algorithms:

- The height of a Radix Tree depends on the length of the keys but in general not the number of elements in the tree. The number of children of every internal node is at most the radix *r* of the radix tree, where $\mathbf{r=2^x}$; $x \geq 1$.
- No balancing operations are required in case of insertion.
- Insertion orders do result in the same tree.



Various variations of radix Tree can be observed. **PATRICIA**, which stands for "***Practical Algorithm To Retrieve Information Coded In Alphanumeric***" are the Radix tree with radix=.In the case of Patricia tries, bits are compared individually (since radix = 2). This is not necessarily the case in radix trees. In general, bits are checked in chunks of

size $\log_2 R$ where R is the radix of the trie. A Hat-trie is another variation that is a cache conscious data structure. One variation includes in which leaf node or the node where word ends contain a node containing complete word, but this is against the space optimisation.For optimal performance, the nodes in the radix tree are traditionally used as reference points in children's nodes; when translating a partial key as a reference to a list of children's guides, finding the appropriate branch in the first place serves as access to the same members. However, this representation can lead to excessive memory usage and use of bad cache in the distribution of data which leads to many branches with fewer people, such as random random transfers.

Radix Tree :: A Comparative Study-

Radix Tree usually represents a space optimised trie, and has performance highly comparable to other data structures. In the case of Radix Tree, insertion and deletion strings usually require O(s), where s=size of string whereas in case of other balanced trees it requires approximately O(log n), where n is number of strings. In case of searching or matching the string every case in a balanced tree requires O(k) complexity, which worsens when all strings have large prefixes. But these operations are very fast in radix trees due to its structure. But it is observed that these trees show high performance in case when strings stored have long matching prefixes. Hash tables also show high performance but only when the number of strings are less. In case of many strings there may be a case of one-to-many mapping which leads to collision which itself needs to be handled.

Radix Tree :: Algorithm -

Data Structure:

```
{
string data;            //to store the data in the node
Node *child[26];    //to store the address to the child corresponding to the alphabet
bool isEndOfString;         //to store whether or not the node is end of an string
};
```

A.

Create-New-Node (string s)

```
1.  Node *newnode = new Node;
2.  newnode.data = s;
3.  newnode.isEndOfString = true;
4.  for i = 0 upto 26:
5.          newnode.child_i = NULL;
6.  return newnode;
```

Working: The function takes a string as parameter and stores it in variable s. It dynamically allocates memory of size and type Node to a pointer newnode of type Node and stores the string s in the member data of newnode and makes the member isEndOfString true. It then starts a loop from i = 0 to 26 and assign NULL value to the member $child_i$ of the newnode. Finally, it returns the newnode.

B.

Count-Node-Child (Node *node)

```
1.  int count=0;
2.  for i =0 upto 26:
3.          if node.child_i != NULL:
4.                  count++;
5.  return count;
```

Working: The function takes a pointer to a Node and stores it in the variable node. Initialising integer count as 0. Then it starts a loop from i=0 to 26 and then it checks whether the member child of the node is NULL or not and if it is not NULL then it increments the count. Finally, it returns the count.

C.

Return-First-Child (Node *node)

```
1.  for i = 0 upto 26:
2.      if node->child_i != NULL:
3.          return index;
```

Working: The function takes a pointer to a Node and store it in the variable node. Then it starts a loop from i=0 to 26 and the it checks whether the member $child_i$ of the node is NULL or not and if it is not NULL then it returns the value of i.

D.

Prefix-Search (string s1, string s2)

```
1.  int lenS1 = length(s1);
2.  int lenS2 = length(s2);
3.  int minLen;
4.  if lenS1 > lenS2:
5.      minLen = lenS2;
6.  else
7.      minLen = lenS1;
8.  for i=0 upto minLen:
9.      if (s1.at(i) == s2.at(i))
10.         continue;
11.     else
12.         return i;
```

Working: The function takes two strings and stores them in s1 and s2 respectively. It then stores length of s1 in lenS1 and length of s2 in lenS2. Then it stores the minimum of lenS1 and lenS2 in integer minLen. Now it starts a loop from i=0 to minLen and check if character at i^{th} index of string s1 is equal to i^{th} index of string s2 and if not it returns i

E.

Split-String (string *s1, string *s2, int index, string s)

```
1.  (*s1).append(s.begin(), s.begin()+index);
2.  (*s2).append(s.begin() + index, s.end());
```

Working: The function takes pointer to two string s1 and s2 whose values should be empty, an integer index and a string s. It now appends characters from beginning of string s to the $(index-1)^{th}$ character to the value of s1. Then it appends characters from $index^{th}$ of the string s to the end to the value of s2. Appending is done using the (**".append"**) method of string class and beginning and end of the string s is recognised using (.begin) and (.end) method of the string class.

F.

Traversal-For-Insert (string *s, Node *node, int *index)

```
1.  *index = Prefix-Search(node.data, *s);
2.  string s1, s2;
3.  Split-String(&s1, &s2, *index, *s);
4.  *s = s2;
5.  if string s is empty:
6.      return node;
7.  else if length(node.data) > *index:
8.      return node;
9.  else if node.child[(*s)[0]-65] != NULL:
10.     node = node.child[(*s)[0]-65];
11.     node = Traversal-For-Insert(s, node, index);
```

```
12.         return node;
13.  else:
14.         return node;
```

Working: The function takes a pointer to string(s), a pointer to Node(node), and a pointer to an integer(index). It calls the function Prefix-Search by passing member data of the node and the value at s and store the returned integer in value at index. Then it creates two strings s1 and s2 with no values and calls Split-String by passing the address of s1, address of s2, value at index and value at s. Then it stores the string in s2 in value at s. Now it checks, if string s is empty it returns the node else if the length of the member data of the node is greater than the value at index it returns the node else if the member child[(*s)[0] – 65] of node(i.e. the child corresponding to the first character of the string s) is not equal to NULL it assigns the member child[(*s)[0] – 65] of node to node and recursively calls itself by passing the parameters s , node and index and then it returns the node else if none of the above cases are found it simply returns the node.

G.

Split (Node **node, int index)

```
1.   string s1, s2;
2.   Split-String(&s1, &s2, index, (*node).data);
3.   (*node).data = s1;
4.   Node *temp;
5.   temp = Create-New-Node(s2);
6.   temp.isEndOfString = (*node).isEndOfString;
7.   (*node).isEndOfString = false;
8.   for i = 0 upto 26:
9.        temp.child_i = (*node).child_i;
10.  for i = 0 upto 26:
11.       (*node).child_i = NULL;
12.  (*node).child[s2[0] - 65] = temp;
```

Working: The function takes a pointer to pointer to Node(node) and an integer index. Creates two strings s1 and s2 which contains nothing and calls Split-String by passing address of s1, address of s2, index, member data of the (*node). Then is assigns string s1 to the member data of (*node). Creates a pointer to Node temp, calls the function Create-New-Node by passing string s2 and stores the returned value in temp. Then it assigns the value of member isEndOfString of (*node) to the member isEndOfString of temp. Then it starts a loop from i=0 to i=26 and assigns the value at i^{th} child of (*node) to

the i^{th} child of temp. Then it starts a loop from i=0 to i=26 and assigns the NULL value to the i^{th} child of (*node). It then assigns the temp to the $(s2[0]-65)^{th}$ child of (*node).

H.

Insert-Node (Node **root, string inputString)

```
1.    if *root == NULL:
2.        *root = Create-New-Node(inputString);
3.    else:
4.        Node *node;
5.        int index;
6.        node = Traversal-For-Insert(&inputString, *root, &index);
7.        if inputString is empty:
8.            if length(node.data) == index:
9.                node.isEndOfString = true;
10.           else:
11.               Split(&node, index);
12.               Node.isEndOfString = true;
13.       else:
14.           Node *temp;
15.           temp = Create-New-Node(inputString);
16.           if length(node.data) == index:
17.               node.child[int(inputString.at(0) - 65)] = temp;
18.           else:
19.               Split(&node, index);
20.               node.child[int(inputString.at(0) - 65)] = temp;
```

Working: The function takes the pointer to a pointer of type Node(root) which is basically the pointer to the pointer of the root node of the tree and a string inputString. It checks whether the value at root is NULL or not.

If it is NULL then it assigns the returned value of the function Create-New-Node by passing inputString.

If it is not NULL it creates a node of type Node* and index of type int. It then calls Traversal-For-Insert by passing reference of inputString, value at root and reference of

index and stores the returned value in node. It then checks if the inputString is empty or not. If it is empty it checks if length of member data of node is equal to index and if it is equal it assigns true to the member isEndOfString of node and if it is not equal it first calls the function Split by passing reference to node and index and then it assigns true to the member isEndOfString of node. If the inputString is not empty it first creates a pointer to Node temp and calls Create-New-Node by passing inputString and stores the returned value in temp. Then it checks whether the length of member data of the node is equal to the index or not. If it is equal it assigns temp to the member child at (inputString[0] – 65)th index of node and if it is not equal it first calls Split by passing reference to node and index and then it assigns temp to the member child at (inputString[0] – 65)th index of node.

I.

void Print-Tree (Node *temp, string s1)

```
1.  if temp.isEndOfString:
2.      s1.append(temp.data);
3.      cout<<s1<<endl;
4.      for j = 0 upto length(temp.data):
5.          s1.pop_back();
6.  for i = 0 upto 26:
7.      if(temp.child_i != NULL)
8.          s1.append(temp.data);
9.          Print-Tree(temp.child_i, s1);
10.         for j = 0 upto length(temp.data):
11.             s1.pop_back();
```

Working: The function takes a pointer to Node temp and a string s1 as parameters. It now checks if the member isEndOfString of temp is true and if it is it appends the member data of temp to s1 using (.append) method of string class, prints s1 and sends the cursor to newline and starts a loop from j=0 to length of member of data of temp and pops a character from s1 till the loop runs. Now it is out of if condition and a new loop starts from i=0 to 26 and it check if the ith child of the temp is NULL or not and if it is not it appends the member data of temp to s1, then calls itself recursively by passing ith child of the temp and s1, then it starts a loop from j=0 to length of temp.data and pops a character from s1 till the loop continues.

J.

Capitalise (string *str)

```
1.  for i = 0 upto length(*str):
2.      if (*str)_i >= 'a' and (*str)_i <= 'z' :
3.          (*str)_i = (*str)_i - 32;
```

Working: The function takes a pointer to string and stores it in str. It starts a loop from i=0 to length of value at str. It checks if i^{th} character of string is between 'a' and 'z' and if it is it subtracts 32 from that character (as the difference between lowercase alphabets and uppercase alphabets is 32.)

K.

Space-Remover (string *s)

```
1.  for i = 0 upto length(*s):
2.      if (*s)_i == ' ':
3.          (*s).erase((*s).begin()+i);
```

Working: The function accepts a pointer to a string and stores it in s. It now starts a loop from i=0 to the length of the string stored at s and checks whether the i^{th} character of the string stored at s is ' ' (space) and if it is it removes that character using the **(".earse")** method of string class.

L.

Modify-String (string *s)

```
1.  Space-Remover(s);
2.  Capitalise(s);
```

Working: This function takes a pointer to string as a parameter and stores it in s and calls the functions Space-Remover and Capitalise by passing s in both of them which in return modify the string at s to have no spaces and all alphabets in uppercase.

M.

Search (Node *node, string stringToBeFind)

```
1.  Node * currNode;
2.  int index;
3.  currNode = Traversal-For-Insert(&stringToBeFind, node, &index);
4.          if stringToBeFind is empty and index == length(currNode.data) and
    currNode.isEndOfString == true:
5.      return true;
6.  else:
```

```
7.      return false;
```

Working: This function takes a pointer to Node(node) and a string stringToBeFind. It creates a variable currNode of type Node* and index of type int. It now calls Traversal-For-Insert function by passing reference to stringToBeFind and reference to index and stores the returned value in currNode. Now it checks whether or not the stringToBeFind is empty, index is equal to length of member data of currNode and member isEndOfString is true. If all three conditions are true it returns true (bool value) else it returns false (bool value).

N.

Merge-Node (Node *node)

```
1.   string s="";
2.   int childIndex = Return-First-Child(node);
3.   s.append(node.data);
4.   s.append(node.child[childIndex].data);
5.   node.data = s;
6.   Node *temp = node.child[childIndex];
7.   node.isEndOfString = temp.isEndOfString;
8.   node->child[childIndex] = NULL;
9.   for i = 0 upto 26:
10.     node->child[i] = temp->child[i];
11.  delete temp;
```

Working: This function takes a pointer to Node and stores it in node. It creates a string s and initialises it to "" (empty). It calls the function Return-First-Child by passing node and stores the returned value in an int variable childIndex. It appends the data from node to s and then it appends the data from the childIndexth child of the node. It then assigns the value of s to the member data of node (overwrites). It then creates a Node pointer named temp and stores the child at childIndexth place of node in it. It then copies the value of isEndOfString of temp to isEndOfString of node. It then assigns NULL value to childIndexth child of node. Now it starts a loop from 0 to 26 and assigns the value in ith child of temp to ith child of node. Finally, it deletes temp.

O.

Delete-Node (Node* node, string stringToBeDeleted)

```
1.   Node * currNode, *prevNode;
2.   int index1, index2;
3.   int count;
```

```
4.    string modifiedString = stringToBeDeleted;
5.    currNode = Traversal-For-Insert(&stringToBeDeleted, node, &index1);
6.              if stringToBeDeleted is and index1 == length(currNode.data) &&
    currNode.isEndOfString == true:
7.        for i=0 upto length(currnode.data):
8.            modifiedString.pop_back();
9.        prevNode = Traversal-For-Insert(&modifiedString, node, &index2);
10.     count = Count-Node-Child(currNode);
11.     currNode.isEndOfString = false;
12.     if count == 0:
13.        prevNode.child[currNode->data.at(0) - 65] = NULL;
14.        delete currNode;
15.        int count1 = Count-Node-Child(prevNode);
16.        if count1 == 1 && prevNode.isEndOfString == false:
17.           mergeNode(prevNode);
18.        else if count == 1:
19.           Merge-Node(currNode);
20.        return true;
21. else
22.     cout<<"\nString Not Found!!!\n";
23.     return false;
```

Working: This function takes a pointer to Node(node) and a string stringToBeDeleted. It creates variables currNode and prevNode of type Node* and index1, index2 and count all of type int. It then creates a variable modifiedString of type string and copies the string from stringToBeDeleted to it. It then calls Traversal-For-Insert by passing reference to stringToBeDeleted, node and reference to the index1 and stores the returned value in currNode. Then it checks whether or not the stringToBeDeleted is empty, index1 is equal to the length of the data of currNode and isEndOfString of currNode is true.

If all three are true, it starts a loop from i=0 to length of data of currNode and pops characters from modifiedString till the loop runs. It then calls Traversal-For-Insert by passing reference to modifiedString, node and reference to the index2 and stores the returned value in prevNode. It then calls the Count-Node-Child function by passing currNode and stores the returned value in count and then it assigns false Boolean value to the isEndOfString of currNode. Now, if count is 0 it assigns NULL value to the child at (0^{th} index character fo data of currNode - 65)th index of prevNode. It then deletes currNode and calls Count-Node-Child by passing prevNode and stores the returned value in int count1. And if count1 is 1 and isEndOfString of prevNode is false it calls

Merge-Node by passing prevNode. Else if count is equal to 1 it calls merge node by passing currNode. Then it returns Boolean value true.

Else it prints “String Not Found” on the console and returns Boolean value false.

Radix Tree :: Analysis -

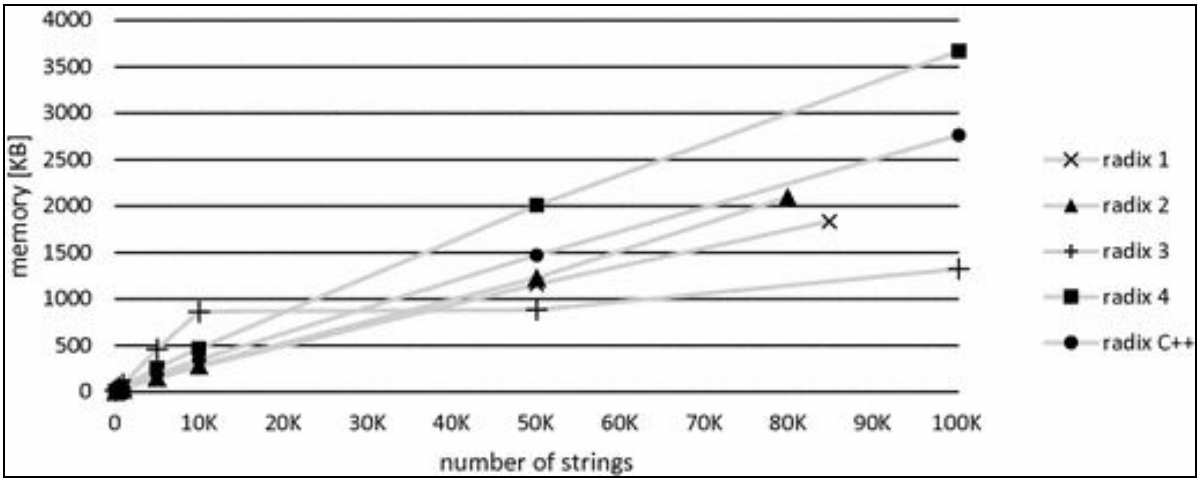
Time Complexity:

Assuming the maximum height of the tree is ‘h’ and the length of the string to be inserted is ‘s’ and they both are comparable and the total number of nodes in the tree is ‘n’.
Time complexity of Prefix-Search(), Split-String() and Split() is O(s) and time complexity of Traversal-For-Insert is **O(h.s).**
Hence The time complexity of the Insert-Node() can be given as O(h.s).
The time complexity of Print-Tree() can be given as O(n.s).
The time complexity for Merge-Node() is O(1) and hence The time complexity of the Delete-Node() is O(h.s).

Space Complexity:

Since it is a compressed version of trie it can not be defined where the string will be splitted or where the node will be merged and hence the space complexity cannot be specifically defined. It depends on the strings inserted and the worst case of a radix tree will be when there is no string with the same prefix and hence n different nodes will be assigned. Assuming the size of a single node is m,

The space complexity of the tree then can be defined as $O(n^2)$.

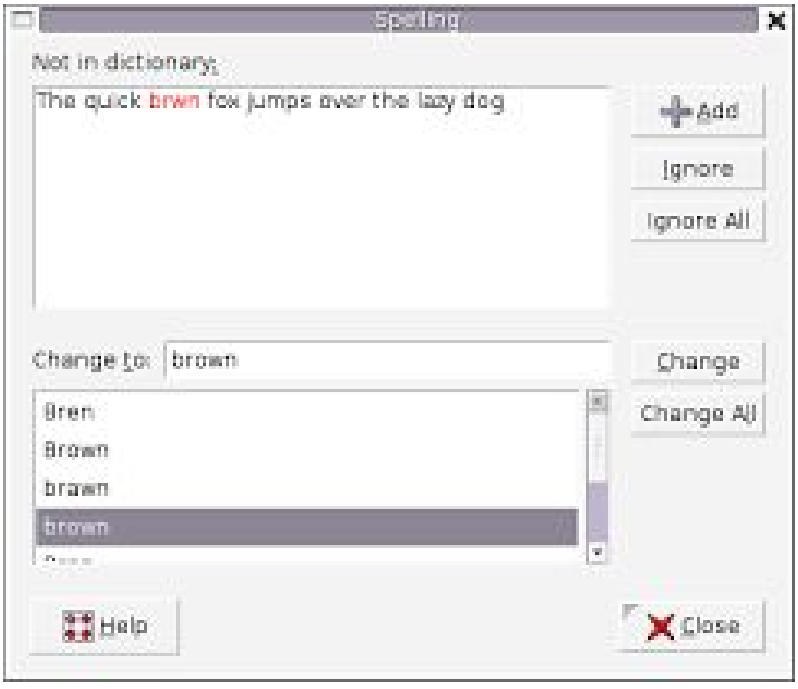


Radix Tree :: Applications -

Radix Trees have a wide range of applications in the case of Data Science and Text processing. These trees prove to be a boon in case when strings have large matching prefixes.

- **IN AUTOCOMPLETION & SPELL CHECKER :**

  This structure can be used in case of text processing or as an aid for users when product designing is taken into concern. These algorithms can be used in case of designing autofill and spell check tools. Basically string matching algorithm is used for spell check but with certain algorithms in combination it is possible to provide the facility of autocompletion. This generally works on the principle that most matching word or broadly saying more correlated word with the word input is generally displayed or used for auto-completion. This technique uses prefix matching which is traditionally implemented using Radix Tree prefix matching Algorithm.In a Radix Trie the lookup is O(k), where k is the length of the key. While this is very fast, unfortunately this holds only true for a single term lookup. The lookup of all terms (or the top-k most relevant) for a given prefix is much more expensive — and this is the operation required for auto-complete.



This algorithm can also help in designing a search box mechanism for implementing a suggestion box or search box which displays all the words with matching prefixes. This generally requires greater time, so it is generally not taken into use. But traditionally it can be used. Nowadays, generally an improved version is used where each substring range containing prefix is linked with some hash table and rank. Generally this is implemented on the client side where the application server keeps a Patricia tree with the valid entries, searches for the strings starting with what was inserted so far and, that's a valid prefix, returns a certain number of entries in this string's subtree.

• IN GETTING WORDS WITH SOME PREFIX :

Suppose in some application if it is required to get all the strings with a certain prefix then it can be best achieved by Radix Trees. This algorithm works on matching prefixes and then visiting all the nodes attached to that node. This terminates when all the nodes are visited and the end of word is found.

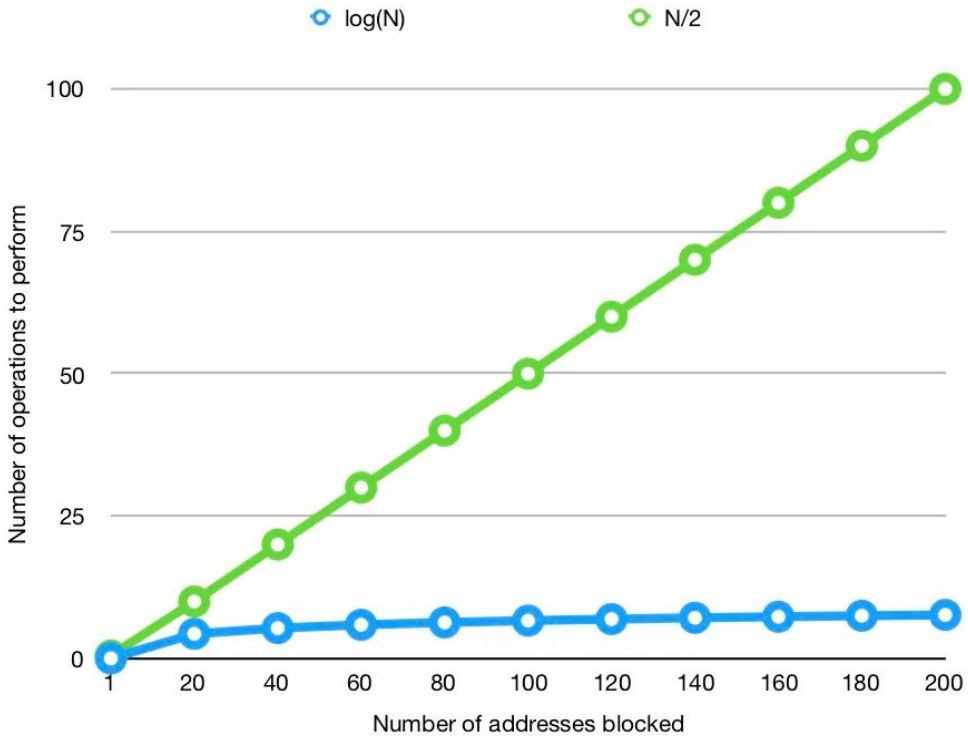
```
function searchNodeWithPrefix(node, s):
    if s == "" then
        return node
    (edge, commonPrefix, sSuffix, edgeSuffix) ← matchEdge(node, s)
    if edge == null then
        return null
    else if edgeSuffix == "" then
        return searchNodeWithPrefix (edge.destination, sSuffix)
    else if sSuffix == null then
        return edge.destination
    else
        return null
```

This is the general algorithm to search all the words with a given prefix. This further is used in various applications including text processing.

● IP BLOCKING MECHANISM :

Almost every organisation encounters a lot of DoS attacks everyday. **Denial-of-Service** (**DoS**) **attack** is an **attack** meant to shut down a machine or network, making it inaccessible to its intended users. Thus it is necessary to figure out some way to block some specific IP addresses from accessing the server. So in order to achieve this several methods are adopted. One of such approaches is designing a blacklist mechanism that itself blocks certain users.

Although the IP addresses are represented as a sequence of decimal (127.0.0.1, IPv4) or hexadecimal (e3e7:682:c209:4cac:629f:6fbe:d82c:7cd, IPv6) numbers, they are internally a single number (encoded using 32 bits, 2130706433 for IPv4, or 128 bits, 302934307671667531413257853548643485645 for IPv6). And these can fairly be represented in binary format ie., 0 and 1. The result is that they can be represented easily through radix trees. Also it is observed that these IP addresses are quite long and consist of only two digits so they usually have long matching prefixes. Thus it becomes a space optimised tool for storing such long data. Also it is observed that since lookup time Radix Trees is O(k), where k=length of string , so they generally work faster. So numbers of operations are much lower and grow as a function of O(log(N)) whereas in other structures it grows as a function of O(N).



- **I.P. ROUTING :**

  IP Routing describes the process of determining the path for data to follow in order to navigate from one computer or server to another.Presently, IP router database consists of a number of address prefixes. When an IP router receives a packet, it must compute which of the prefixes in its database has the longest match when compared to the destination address in the packet. The packet is then forwarded to the output link associated with that prefix.The use of best matching prefix in forwarding has allowed IP routers to accommodate various levels of address hierarchies, and has allowed different parts of the network to have different views of address hierarchy

- Another application may include determining Longest Common Prefix, which itself acts as an input or aid for other algorithms.