# TRIE Trees

## Introduction

Strings can essentially be viewed as the most important and common topics for a variety of programming problems. String processing has a variety of real world applications too, such as:

- **Search Engines**
- **Genome Analysis**
- **Data Analytics**

All the content presented to us in textual form can be visualized as nothing but just strings.

**Tries:**

Tries are an extremely special and useful data-structure that are based on the *prefix of a string*. They are used to represent the "Re**trie**val" of data and thus the name Trie.

**Prefix : What is prefix:**

The prefix of a string is nothing but any n letters $n \leq |S|$ that can be considered beginning strictly from the starting of a string. For example , the word "abacaba" has the following prefixes:

a
ab
aba
abac
abaca
abacab

A Trie is a special data structure used to store strings that can be visualized like a graph. It consists of nodes and edges. Each node consists of at max 26 children and edges connect each parent node to its children. These 26 pointers are nothing but pointers for each of the 26 letters of the English alphabet A separate edge is maintained for every edge.

Strings are stored in a top to bottom manner on the basis of their prefix in a trie. All prefixes of length 1 are stored at until level 1, all prefixes of length 2 are sorted at until level 2 and so on.

For example , consider the following diagram

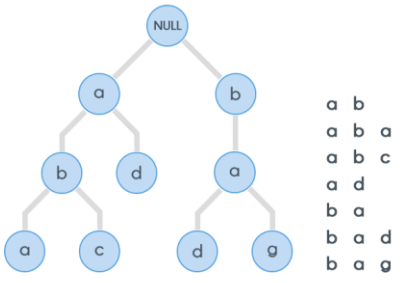



Fig. 1

Now, one would be wondering why to use a data structure such as a trie for processing a single string? Actually, Tries are generally used on groups of strings, rather than a single string. When given multiple strings , we can solve a variety of problems based on them.

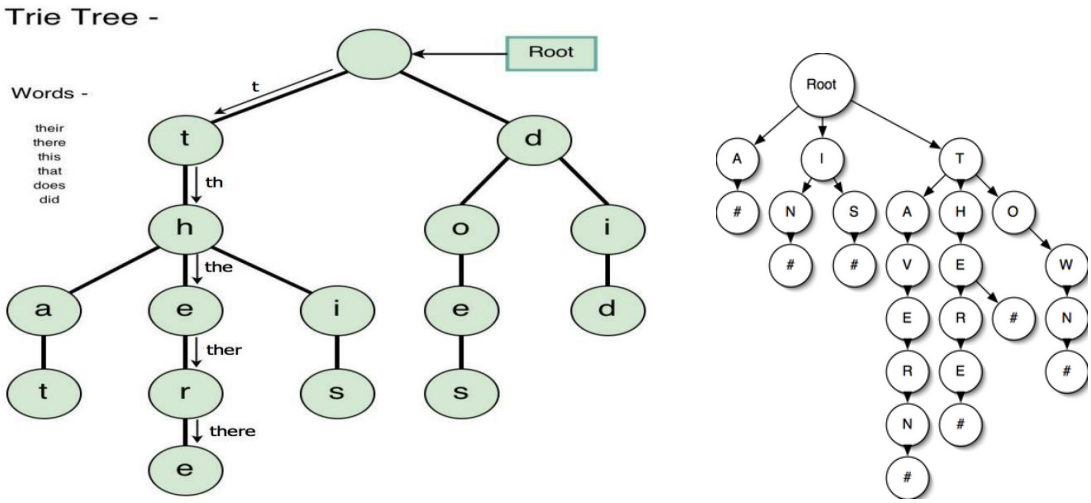
For example, consider an English dictionary and a single string s, find the prefix of maximum length from the dictionary strings matching the string s. Solving this problem using a naive approach would require us to match the prefix of the given string with the prefix of every other word in the dictionary and note the maximum. The is an expensive process considering the amount of time it would take. Tries can solve this problem in much more efficient way.

Before processing each Query of the type where we need to search the length of the longest prefix, we first need to add all the existing words into the dictionary. A Trie consists of a special node called the root node. This node doesn't have any incoming edges. It only contains 26 outgoing edfes for each letter in the alphabet and is the root of the Trie.

So, the insertion of any string into a Trie starts from the root node. All prefixes of length one are direct children of the root node. In addition, all prefixes of length 2 become children of the nodes existing at level one.

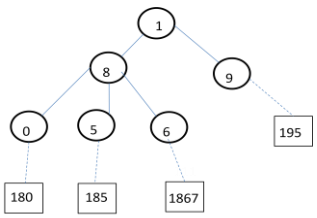
TRIE tree is a digital search tree, need not be implemented as a binary tree.

- •Each node in the tree can contain ‘m’ pointers –corresponding to ’m‘ possible symbols in each position of the key.
- •Generally used to store strings.




A trie, pronounced “try”, is a tree that exploits some structure in the keys
-e.g. if the keys are strings, a binary search tree would compare the entire strings but a triewould look at their individual characters

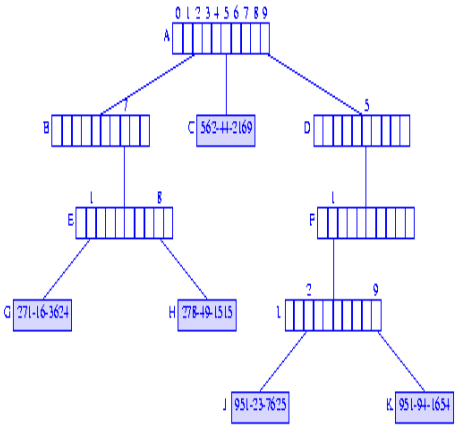
-A trieis a tree where each node stores a bit indicating whether the string spelled out to this point is in the set




- If the keys are numeric, there would be 10 pointers in a node.

- Consider the SSN number as shown.

Name | **Social Security Number (SS#)**
Jack | 951-94-1654
Jill | 562-44-2169
Bill | 271-16-3624
Kathy | 278-49-1515
April | 951-23-7625




Operations on TRIE TREES:

1. Insert a node into a TRIE TREE.

The code for the same is as follows:

// create a trie node using the structure definition as given below with 2 fields:

1. Array of pointers of size 255. Since, the no of characters are 255.

Variables:

**child** - array of pointers to structure trienode.

**endofword** – to see whether it is end of the string or the word.

## Structure of a node in a TRIE Tree :

- A node of a TRIE tree is represented as shown below.
- One field for each alphabet( A – Z), 26 columns.
- Each column is a pointer to another TRIE node or carries NULL and
- One field for end of word (key).

| A | B | C | D | E | F | ..... | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|
| F1 | F2 | F3 | F4 | F5 | F6 | ..... | F23 | F24 | F25 | F26 |
| End of Word / (eok) | | | | | | | | | | |

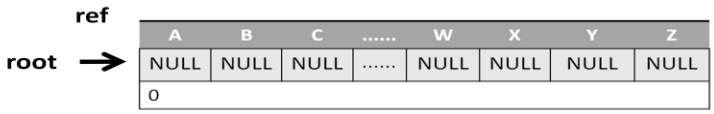
| Address of the next node (reference for us) |
|---|
| Field number – for user's reference no field is created , No memory is allocated |
| End of word / key field |

```
struct trienode {
                    struct trienode *child[255];
                    int endofword;
                  };
```

**// create a trienode – getnode function does the job.**

| A | B | C | D | E | F | ..... | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|
| F1 | F2 | F3 | F4 | F5 | F6 | ..... | F23 | F24 | F25 | F26 |
| End of Word / (eok - $) | | | | | | | | | | |

| |
|---|
| Address of the next node (reference for us) |
| Field number |
| End of word / key field |

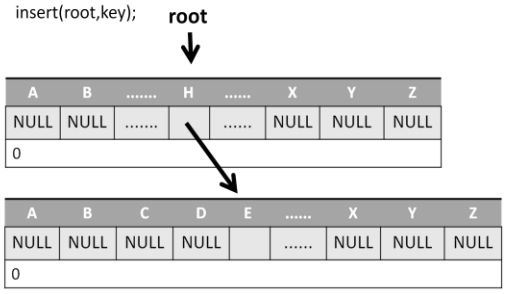



root=getnode();

```
struct trienode* getnode()
{
  struct trienode* temp;
  int i;
  temp=(struct trienode *)(malloc(sizeof( struct trienode)));
  for(i=0;i<255;i++)
    temp->child[i]=NULL; // initially all the variables are assigned NULL
  temp->endofword=0;    //  end of word is assigned to 0.
  return temp;
}
```

**Function to insert a node / character into the trie tree using the function insert.**

On function call insert, the given string "HELLO" is inserted into the TRIE tree as shown below.




**For every character read getnode function and store the link in the child variable**

```
void insert(struct trienode* root, char *key)
{
  struct trienode *curr;
  int i, index;

  curr = root;
  for(i=0;key[i]!='\0';i++)
   { index=key[i];
       if(curr->child[index]==NULL)
           curr->child[index]=getnode();
     curr=curr->child[index];
```

```
    }
  curr->endofword=1;
}
```

// to display the trie tree, the function display is used.

```
void display(struct trienode *curr)
{int i,j;

      for(i=0;i<255;i++)
      {
            if(curr->child[i]!=NULL)
            {
                  word[length++]=i;
                  if(curr->child[i]->endofword==1)
                  {
                        //print the word
                        printf("\n");
                        for(j=0;j<length;j++)
                              printf("%c",word[j]);
                  }
                  display(curr->child[i]);
            }
      }
      length--;
      return;
}
```

**To search for a given string, use the function search as shown below.**

```
int search(struct trienode * root, char *key)
{
 int i,index;
 struct trienode *curr;
 curr=root;
  for(i=0;key[i]!='\0';i++)
   { index=key[i];
        if(curr->child[index]==NULL)
              return 0;
        curr=curr->child[index];
   }
   if(curr->endofword==1)
         return 1;
   return 0;
}
```

**To, delete a given string, use the function delete_trie as shown below.**
**The function searches for a given string in the tree. If the string does not exist then it displays string not found. Otherwise, the word has to be deleted with respect to the following cases:**
**Case 1: As the word is searched character by character in the trie, the index and the addresses of the nodes are stored on the stack if a match is found. At the end, endofword is set to 0.**

**Now, to delete the word, first pop the top of the stack, if it has -1 as the index it does nothing as it is the end of the word. Otherwise, it does nothing if it is a root node of the trie tree. Otherwise it ill delete the node if the node doesnot have any descendents (child nodes).**

```
void delete_trie(struct trienode *root,char *key)
{
 int i,k,index;
 struct trienode *curr;
 struct stack x;

 curr =root;
 for(i=0;key[i]!='\0';i++)
   { index=key[i];
       if(curr->child[index]==NULL)
       {
            printf("Word not found..");
         return;
       }
       push(curr,index);
      curr=curr->child[index];
     }
     curr->endofword=0;
     push(curr,-1);

     while(1)
     {
           x=pop();
           if(x.index!=-1)
                x.m->child[x.index]=NULL;
           if(x.m==root)//if root
           break;
           k=check(x.m);
           if((k>=1)||(x.m->endofword==1))break;
           else free(x.m);
     }
     return;
}
```

**The function checks whether it has any descendents or not. If a node has descendents then it returns count of the number of descendents otherwise returns 0.**

```
int check(struct trienode *x)
{
     int i,count=0;
     for(i=0;i<255;i++)
     {
           if(x->child[i]!=NULL) count++;
     }
     return count;
}
```