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≤|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

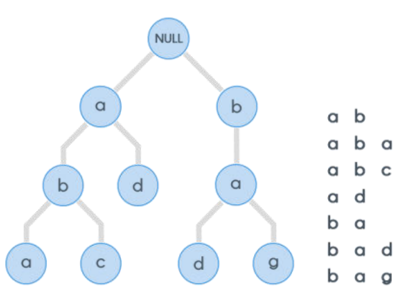


Fig. 1

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.

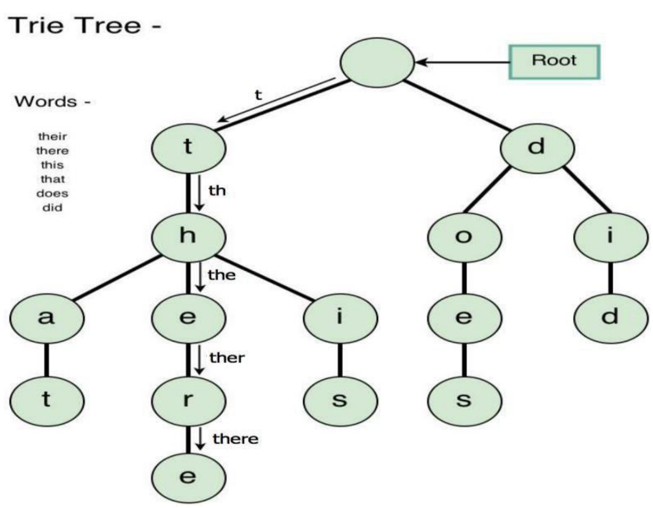
For example, consider an English dictionary and a single string s, find the prefix of maximum length from the dictionary strings matching the string s. Solving this problem using a naive approach would require us to match the prefix of the given string with the prefix of every other word in the dictionary 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.

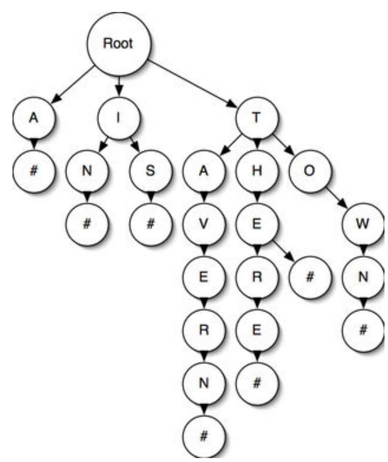
Before processing each Query of the type where we need to search the length of the longest prefix, we first need to add all the existing words into the dictionary. A Trie consists of a special node called the root node. This node doesn't have any incoming edges. It only contains 26 outgoing edfes for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

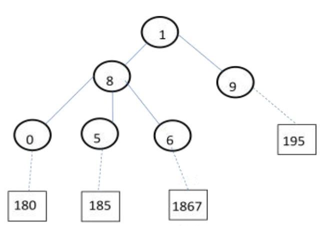
- Each node in the tree can contain ‘m’ pointers –corresponding to ’m‘ possible symbols in each position of the key.
- Generally used to store strings.





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

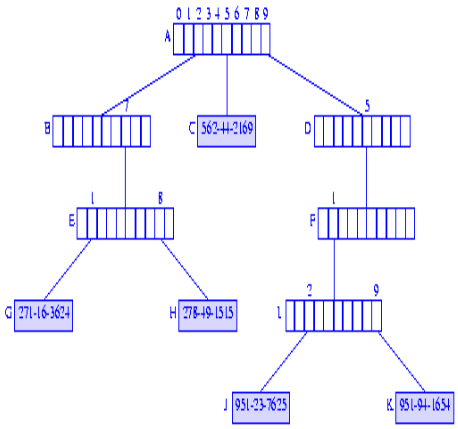
-A trieis a tree where each node stores a bit indicating whether the string spelled out to this point is in the set



- 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)										

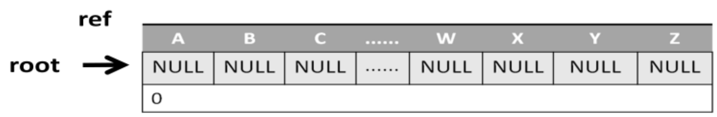
Address of the next node (reference for us)
Field number – for user's reference no field is created , No memory is allocated
End of word / key field

```
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



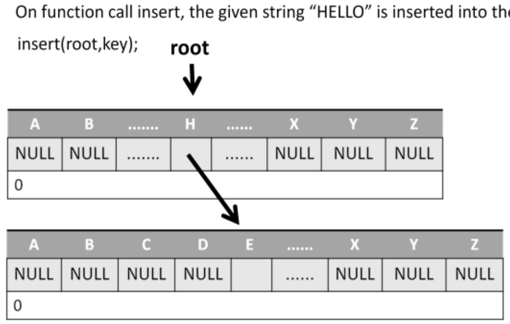
root=getnode();

```
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.

insert(root,key);



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;
}
```

UE19CS202: DATA STRUCURES AND ITS APPLICATIONS (4-0-0-4)
Department of Computer Science & Engineering
PES UNIVERSITY

# of Hours : 56

Class #	Chapter Title/Reference Literature	Topics Covered	Reference Chapter	Page Numbers
38.	Unit-5: Suffix Tree , Hashing Techniques T1: Chapters 7(7.3,7.4) R1: Chapter 8(8.1,8.6) 10(10.2,10.3)	Suffix Trees: Definition, Introduction of Trie Trees, suffix trees	T1 : Chapter 7	465, 467
39.		Implementation of Trie trees-insert operations,	T2 : Chapter 10	457 - 461
40.		Implementation of Trie trees-delete and search operations.	T2 : Chapter 10	457-461
41.		Application: URL decoding		
42.		Hash: definition, hash function, hash table.	T1: Chapter 7 T2 Chapter 8	T1: 468-470 T2: 329, 345-353
43.		Collision Handling: Separate Chaining		T1:488-491 T2:345-353
44.		Collision Handling: Open Chaining		T1:471-473 T2:345-353
45.		Double hashing, Rehashing		T1:473 T2:345-353
46.		Application of hashing in Cryptography.		
47.		Summary of Data Structures.		

Literature

Book Type	Code	Title & Author	Publication Information		
			Edition	Publisher	Year
Text Book	T1	Data Structures using C & C++, YedidyahLangsam, Moshe J. Augenstein, Aaron M. Tenenbaum	2nd	Pearson Education	2015

Reference Book	R1	Data Structures and Program Design in C, Robert L. Kruse, Clovis L. Tando, Bruce P. Leung	2nd	Pearson Education	2007

Hash Function, Hash Table creation:

A function that transforms a key into a table index is called a hash function. If h is a hash function and
key is a key, h(key) is called the hash of key and is the index at which a record with the key key should he placed. If r is record whose key hashes into hr, hr is called the hash key of r. The hash function in the preceding example is h(k) = key/ 1000. The values that h produces should cover the entire set of indices in the table. For example, the function x%1000 can produce any integer between 0 to 999, depending on the value of x. It is a good idea for the table size to be somewhat larger than the number of records that are to be inserted.

- A good hash function is one that distributes keys evenly among all slots / index (locations).
- Design of a hash function is an art more than science.

- Consider key elements as 34, 46, 72, 15, 18, 26, 93
- Hash function is **key mod 5.**
- Index value for the keys are generated using the given hash function
- 34 mod 5 = 4, 34 is stored at index 4.
- 46 mod 5 = 1, 46 is stored at index 1.
- 72 mod 5 = 2 , 72 is stored at index 2.
- 15 mod 5 = 0, 15 is stored at index 0.
- 18 mod 5 = 3, 18 is stored at index 3.

This technique is called **closed hashing. It is shown as below:**

Hash Table

Index / hash	DATA
0	15
1	46
2	72
3	18
4	34

Consider, additional elements 26, 93.

Since, the capacity of the hash table is full, it cannot insert the next element 26 and 93.

Let us say if the capacity is increased, it can accommodate 26 in the hash table.

Now calculate the hash value for 26 = 26 % 5 =1.

This results in collision as the location is not empty. Then search for the first empty location. Say it is at index 5. Now, 26 can be stored in it.

Later for 93, the location 3 is non empty. Searching for the empty location, it can find at location 6. 93 is stored at location with index 6.

Coming to the previous case of only 5 locations, collision occurs. This problem can be resolved by

- Increasing the Memory Capacity.
- Overcoming Collision using
    - Open Addressing / Separate Chaining
    - Closed Addressing :
        - Linear Probing
        - Quadratic Probing
        - Double Hashing

Open Addressing / Separate Chaining :

Initially the hash table contains all NULL values in the address field. This is as shown in the figure below.

Hash Table

Index	address
0	NULL
1	NULL
2	NULL
3	NULL
4	NULL

Let us consider again the same key elements as 34, 46, 72, 15, 18. If the hash function is

key mod 5. The series of locations generated is as shown.

- 34 mod 5 = 4, 34 is stored at index 4.
- 46 mod 5 = 1, 46 is stored at index 1.
- 72 mod 5 = 2 , 72 is stored at index 2.

- 15 mod 5 = 0,   15 is stored at index 0.
- 18 mod 5 = 3,   18 is stored at index 3.

The same is as shown below.