Laboratory Module D

TRIE TREES

1 Preparation Before Lab

Data structures

In computer science a **trie**, or prefix tree, is an ordered multi-way tree data structure that is used to store strings over an alphabet. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. Each node contains an array of pointers, one pointer for each character in the alphabet and all the descendants of a node have a common prefix of the string associated with that node. The root is associated with the empty string and values are normally not associated with every node, only with leaves.

A trie is a tree data structure that allows strings with similar character prefixes to use the same prefix data and store only the tails as separate data. One character of the string is stored at each level of the tree, with the first character of the string stored at the root.

The term trie comes from "re**trie**val." Due to this etymology it is pronounced [tri] ("tree"), although some encourage the use of "try" in order to distinguish it from the more general tree.

For example, in the case of alphabetical keys, each node has an array of (27) pointers to its branches, one for each of the 26 alphabet characters and one for blank (“ ”). The keys are stored in leaf (information) nodes. To access an information node containing a key, we need to move down a series of branch nodes following the appropriate branch based on the alphabetical characters composing the key. All trie_node fields that neither point to a intern node nor to an leaf node are represented using null pointers.

Figure 1 illustrates an example trie for alphabetical keys. The trie stores the keys AEROPLANE, BICYCLE, BIKE, BUS, CAR, CARAVANE, CARRIAGE, TRAIN.

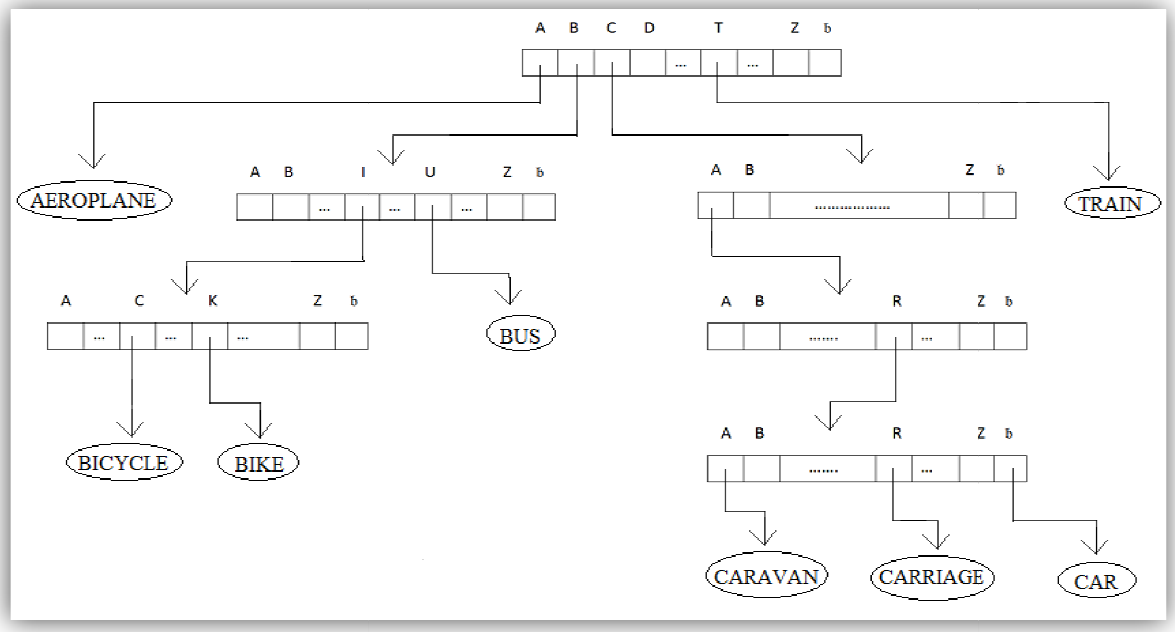


Fig. 1. An example trie

To access these information nodes, we follow a path beginning from a branch node moving down each level depending on the characters forming the key, until the appropriate information node holding the key is reached. Thus the depth of an information node in a trie depends on the similarity of its first few characters (prefix) with its fellow keys. Here, while AEROPLANE and TRAIN occupy shallow levels (level 1 branch node) in the trie, CAR, CARRIAGE, CARAVAN have moved down by 4 levels of branch nodes due to their uniform prefix "CAR". Observe how we move down each level of the branch node with the help of the characters forming the key. The role played by the blank field in the branch node is evident when we move down to access CAR. While the information node pertaining to CAR positions itself under the blank field, those of CARAVAN and CARRIAGE attach themselves to pointers from A to R respectively of the same branch node.

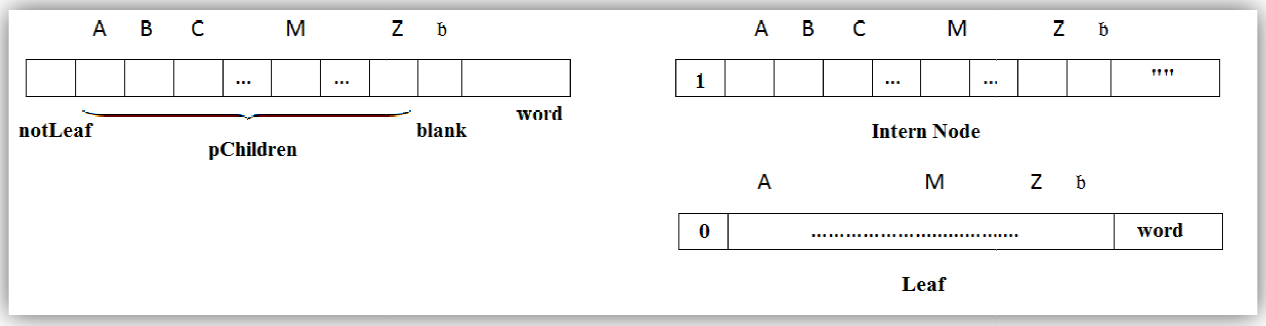
We shall see how to construct a basic TRIE data structure in C++. Each node of the trie needs to store a key (usually a string – here an array of chars) and an array of pointers to its branches. The branches correspond to the 26 results that can be returned by the character position in alphabet with respect to 'a' and the blank character. And each node has a variable called as 'NotLeaf' to mark the end of a word – indicates if the node is an information or an intern one.

The basic element - Node of a TRIE data structure looks like this:

```
typedef struct trie_node{

        bool  NotLeaf;
        trie_node *pChildren[NR];
        var_type word[20];
}node;
```

where :

```
#define NR 27                // the American alphabet(26 letters) plus blank.
typedef char var_type;       // the key is a set of characters
```



The constructors for a trie_node simply sets all pointers in the node to NULL; for intern nodes we have a NULL key and for leaves we store in word[ ] the desired (user input) string. We use as constructors 2 functions:

```
trie_node  *NewIntern() ;
trie_node  *NewLeaf(char word []) ;
```

Algorithms in pseudocode. Examples

The main abstract methods of the TRIE ADT are :

1. bool search (char string[]).
2. void insert (char string[]) ;

Searching a TRIE

To search for a key k in a trie T, we begin at the root which is a branch node. Let us suppose the key k is made up of characters k_1 k_2 k_3 … k_n. The first character of the key K viz. , k_1 is extracted and the pChildren field corresponding to the letter k_1 in the root branch node is spotted. If T->pChildren[k_1 - 'a'] is equal to NULL, then thesearch is unsuccessful, since no such key is found. If T->pChildren[k_1 –'a'] is not equal to NULL. Then the pChildren field may either point to an information node or a branch node. If the information node holds K then the search is done. The key K has been successfully retrieved. Otherwise, it implies the presence of key(s) with a similar prefix. We extract the next character ,k_2 of key K and move down the link field corresponding to k_2 in the branch node encountered at level 2 and so on until the key is found in an information node or the search is unsuccessful. The deeper the search, the more there are keys with similar but longer prefixes.

PSEUDOCODE**.** The search algorithm involves the following steps:

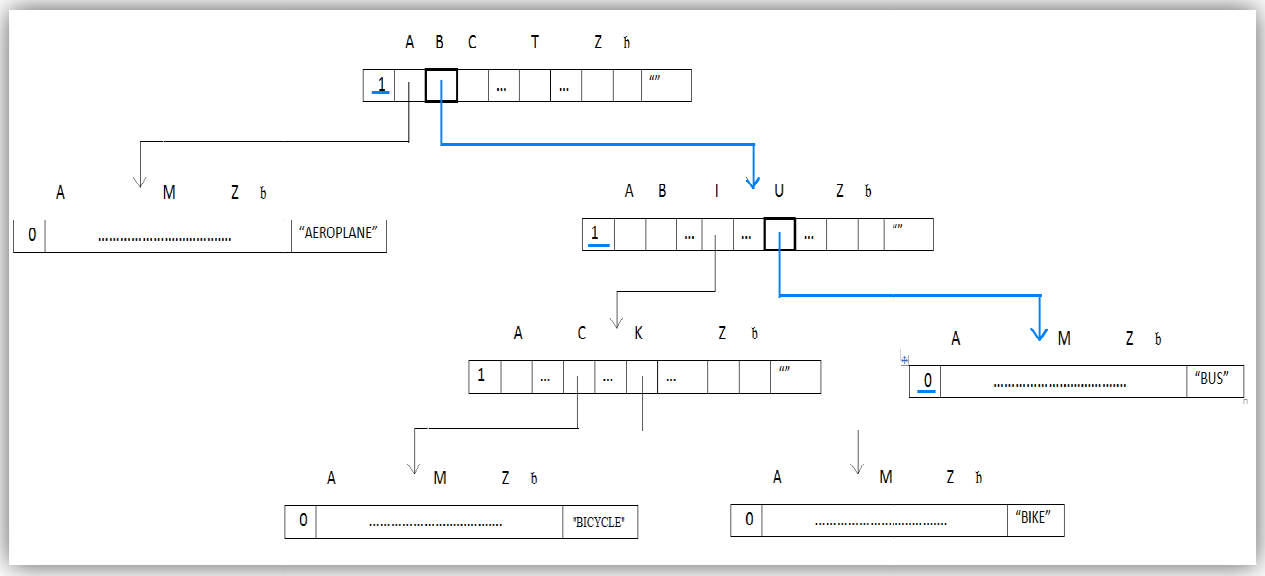
1. For each character in the string, see if there is a child node with that character as the content.
2. If that character does not exist, return false.
3. If that character exist, repeat step 1.
4. Do the above steps until the end of string is reached.
5. When end of string is reached and if the marker (NotLeaf) of the current Node is set to false, return true, else return false.

```
**Procedure FIND(trie, string)
    begin
        if  trie = NULL   then
            return FALSE
        else
            nex <- index <- trie

      count <- 0
      while index->NotLeaf and count < lenght(keyWord) and
                                          index->pChildren[keyWord[count]-'a'] <> NULL   do

            next  <- index->pChildren[keyWord[count]-'a']
           index <- next
           count <- count +1
    end while

    if  next = NULL  then
            return TRUE
    else
       data <- next
        if   data->word = keyWord   then
            return TRUE
       else
            if data->pChildren[26]->word = keyWord  then
                return TRUE
            else
                return NULL
  end
```

EXAMPLE.

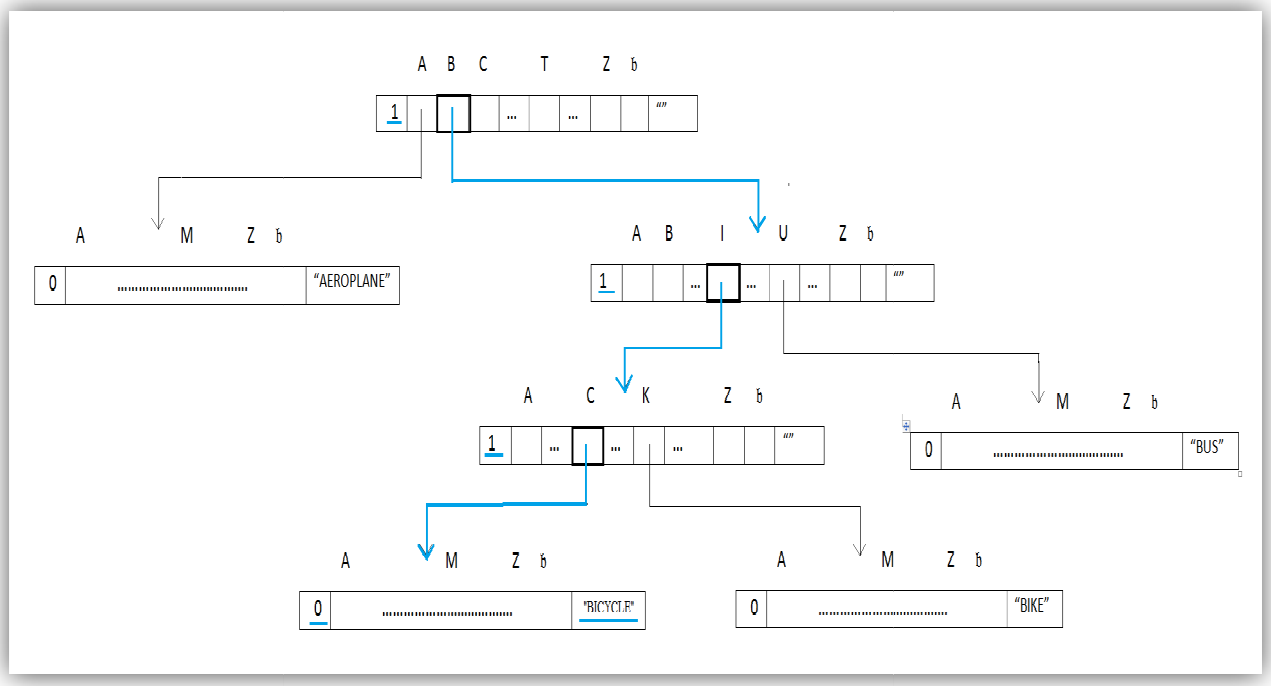
Using the above algorithm, let's perform a search for the key "BU".

1. See whether "B" is present in the current node's children. Yes, it's present, so set the current node to child node which is having character "B".
2. See whether "U" is present in the current node's children. Yes, it's present, so set the current node to child node which is having character "U".
3. The current node it's a leaf, se we compare the searched key "BU" with the data contained in the current node, which is "BUS". The words are different, so the function returns false.



Using the above algorithm, let's perform a search for the key "BICYCLE".

1. See whether "B" is present in the current node's children. Yes its present, so set the current node to child node which is having character "B".
2. See whether "I" is present in the current node's children. Yes its present, so set the current node to child node which is having character "I".
3. See whether "C" is present in the current node's children. Yes its present, so set the current node to child node which is having character "C".
4. The current node it's a leaf, se we compare the searched key "BICYCLE" with the data contained in the current node, which is "BICYCLE". It's the same word, so the function returns true.



Insertion into a Trie

To insert a key K into a trie we begin as we would to search for the key K, possibly moving down the trie, following the appropriate pChildren fields of the branch nodes, corresponding to the characters of the key. At the point where the pChildren of the branch node leads to NULL, the key K is inserted as an information node.

PSEUDOCODE**.** Any insertion would ideally be following the below algorithm:

1. Find the place of the item by following bits
2. If there is nothing, just insert the item there as a leaf node
3. If there is something on the leaf node, it becomes a new intern node. Build a new subtree or subtrees to that inner node depending how the item to be inserted and the item that was in the leaf node differs.
4. Create new leaf nodes where you store the item that was to be inserted and the item that was originally in the leaf node.

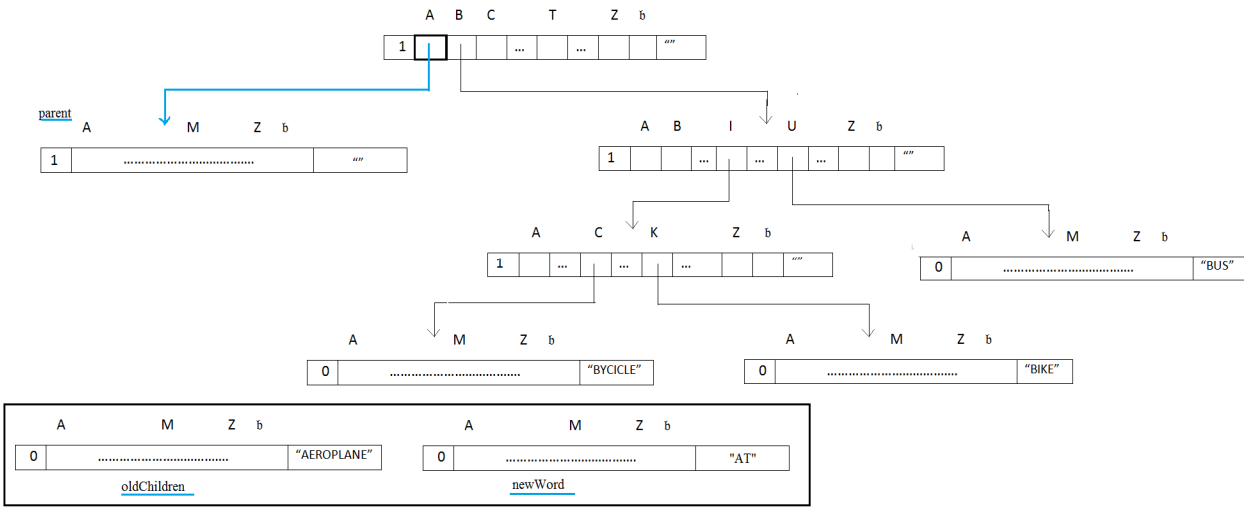
```
*** PROCEDURE B-TREE-SEARCH (x , k)
begin i<=1
while i <= n[x] and k > keyi[x]
do
i <- i + 1
 if i <= n[x] and k = keyi[x] then
 return (x, i)
if leaf[x] then
return NIL
else
Disk-Read(ci[x])
return B-Tree-Search(ci[x], k)
end;
```

```
**Procedure Insert(trie, keyWord)
    begin

        lenght <- length(keyWord)

        next <- trie;
        //
        if  trie = NULL then
                    trie          =  create empty internal node
                    new_leaf  = create leaf with keyWord
                    trie->pChildren[keyWord[0]-'a'] <- new_leaf
                     exit
        else
                index  <- next
         end if

        inWordIndex <- 0//
        while  inWordIndex < lenght  and  index->NotLeaf = true  and
                index->pChildren[keyWord[inWordIndex]-'a']  <>  NULL))   do

                  parent  <- next;
                  next     <- index->pChildren[keyWord[inWordIndex]-'a'];
                  index   <- next;

                  inWordIndex <- inWordIndex + 1
        end while


       if   inWordIndex < lenght and index->pChildren[keyWord[inWordIndex]-'a'] = NULL and
              index->NotLeaf = true     then

                  new_index  <- NewLeaf(keyWord)
                  index->pChildren[keyWord[inWordIndex]-'a']  <- new_index

                  exit
        else
                  data <- next


        if data->word = keyWord   then
                   print "Word already exists in trie !!!"
        else
                  oldChildren <- parent->pChildren[keyWord[inWordIndex-1]-'a']
                  newWord <- NewLeaf(keyWord)

                  prefixLenght <- lenght(keyWord)
                  if data->word[0] <> '\0'     then
                      if lenght(data->word) <  prefixLenght    then
                                  prefixLenght = lenght(data->word)

        createIntern <- false

       while  inWordIndex <= prefixLenght and (data->word[0] <> '\0'  and
                  (data->word[inWordIndex-1] = keyWord[inWordIndex-1])  or  (data->word[0] == '\0' )     do

                  intern <-NewIntern()
```

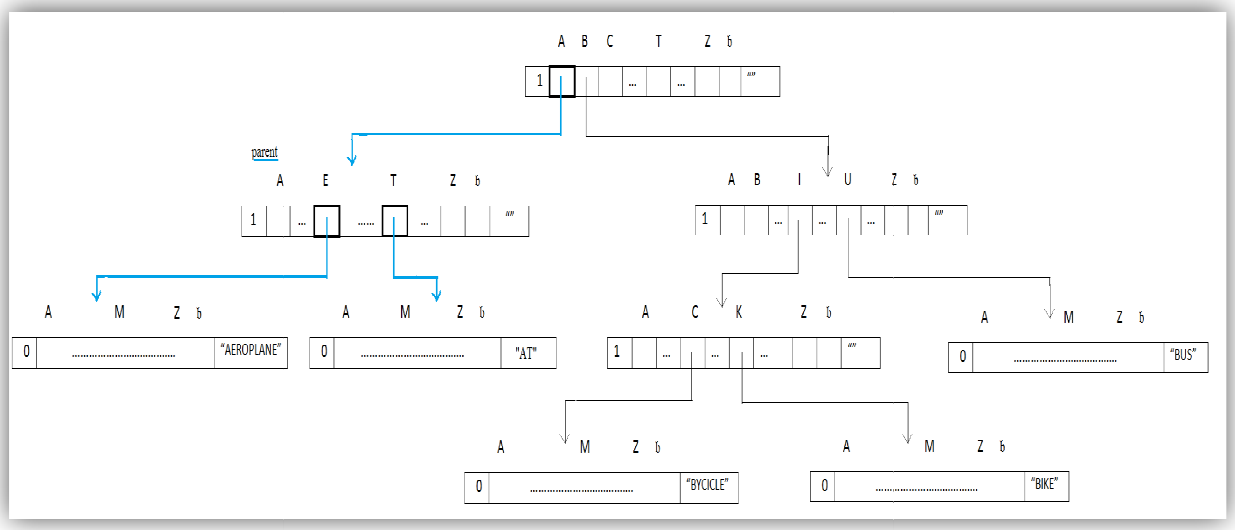
```
                parent->pChildren[keyWord[inWordIndex-1]-'a'] <- intern
                parent->NotLeaf <- true
                parent <- intern
                inWordIndex <- inWordIndex +1

                createIntern  = true
        end while

        if createIntern   then
              inWordIndex <- inWordIndex -1

        if  inWordIndex <> prefixLenght or (inWordIndex = prefixLenght and
              length(keyWord) = length(data->word))        then

                parent->pChildren[data->word[inWordIndex]-'a'] <- oldChildren
                parent->pChildren[keyWord[inWordIndex]-'a']    <- newWord
        else
                if data->word[0] <> '\0'     then
                     if lenght(data->word) <= prefixLenght   then

                          parent->pChildren[26] = oldChildren
                          parent->pChildren[keyWord[prefixLenght]-'a'] = newWord
                     else
                          parent->pChildren[26] = newWord
                          parent->pChildren[data->word[prefixLenght]-'a'] = oldChildren
                      end if
                else
                          for (int count = 0 ; count < 27;count++)
                               parent->pChildren[count] = oldChildren->pChildren[count]
                          parent->pChildren[26] = newWord
                end if
         end if

         exit
end
```

Now let's see how the word "AT" is getting inserted:

- Current node not a leaf. See whether "A" is present in the current node's children (which is root). Yes it's present, so set the current node to the child node which is having the character "A".
- The pChildren field corresponding to 'A' in the root branch points to an information node holding "AERPLANE". This implies that there is already a key with a uniform prefix available in the trie. We now remove "AERPLANE" and store it in oldChildren and create a new leaf for our word. We open a branch node to accommodate both "AERPLANE " and "AT".

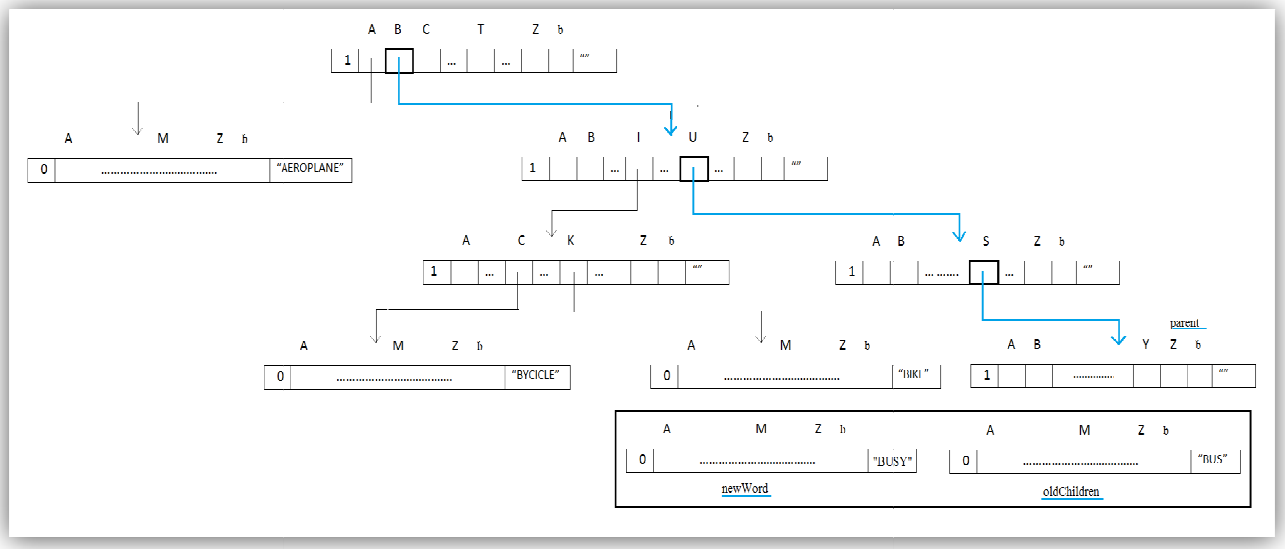


- The second character of the two words doesn't match, so "AERPLANE" and "AT" can be inserted as information nodes corresponding to the pChildren fields of 'E' and 'T'.

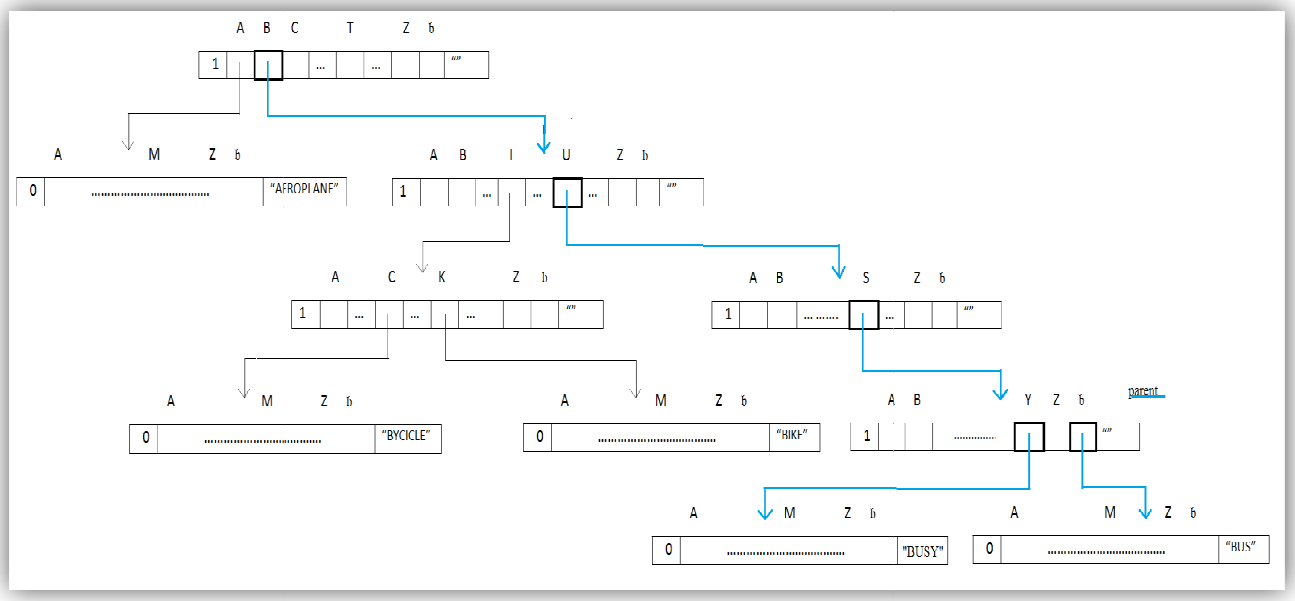


Now let's see how the word "BUSY" is getting inserted:

- Current node not a leaf. See whether "B" is present in the current node's children (which is root). Yes it's present, so set the current node to the child node which is having the character "B".
- Current node not a leaf. See whether "U" is present in the current node's children (which is root). Yes it's present, so set the current node to the child node which is having the character "U".
- The pChildren field corresponding to 'U' points to an information node holding "BUS". We now remove "BUS" and store it in oldChildren and create a new leaf for "BUSY". We open a branch node to accommodate both "BUS" and "BUSY".
- The second character of the 2 keys matches and so does the third ! Since the matching prefixes of "BUS" and "BUSY" ("BUS") is of length 3, the situation now calls for opening three levels of branch nodes other than the root node



- It is only at level 4 ,where the end of "BUS" is reached, that "BUS " and "BUSY"can be inserted as information nodes corresponding to the pChildren fields of 'Y' and blank character, being given the fact that "BUS" represents a prefix for "BUSY".



Complexity analysis

Now that we've seen the basic operations on how to work with a TRIE, we shall now see the space and time complexities in order to get a real feel of how good a TRIE data structure is. Let's take the two important operations INSERT and SEARCH to measure the complexity.

INSERT operation first. Let's always take into account the worst case timing first and later convince ourselves of the practical timings. For every Node in the TRIE we had something called as pChildren where the pChildren can be either an Array or a List. If we choose Array, the order of whatever operation we perform over that will be in O(1) time, whereas if we use a Linked List the number of comparisons at worst will be 26 (the number of alphabets). So for moving from one node to another, there will be at least 26 comparisons will be required at each step.

Having these in mind, for inserting a word of length 'k' we need (k * 26) comparisons. By Applying the Big O notation it becomes O(k) which will be again O(1). Thus insert operations are performed in constant time irrespective of the length of the input string (this might look like an understatement, but if we make the length of the input string a worst case maximum, this sentence holds true).

Same holds true for the search operation as well. The search operation exactly performs the way the insert does and its order is O(k*26) = O(1).

2. Sample coding

```
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <ctype.h>
#include <conio.h>
#include <iostream>

using namespace std;

#define NR 27                  // the American alphabet(26 letters) plus blank.
typedef char var_type;       // the key is a set of characters

typedef struct trie_node{
        bool NotLeaf;          // indicates if the trie_node struct is a leaf or an intern node
        trie_node *pChildren[NR]; // a list of pointers corresponding to the used alphabet
        var_type word[20];       // the string stored in node
}node;


//function for creating a leaf node
trie_node *NewLeaf(char keyWord[20])
{
        trie_node *t_node;
        int count;

            //allocating the necessary memory
        t_node = (trie_node *)malloc(sizeof(trie_node));

        for(count = 0; count < 27; count++)           //the terminal nodes don't have children
                        t_node->pChildren[count] = NULL;
        t_node->NotLeaf = false;      // the node is a leaf
   strcpy(t_node->word,keyWord); //store in the structure(node->word) the string

        return t_node;
}
```

```
//function for creating a intern node
trie_node *NewIntern()
{
	trie_node *t_node;
	int count;

	   //allocating the necessary memory
	t_node = (trie_node *)malloc(sizeof(trie_node));

	for(count = 0; count < 27; count++)    // initial the intern node don't have children
				t_node->pChildren[count] = NULL;
	t_node->NotLeaf = true;      //it isn't a leaf
	t_node->word[0] = 0;      //so we store the null string in node

	return t_node;
}

//function performs a search in the TRIE when a string or key is passed.
void Find(trie_node *trie, char keyWord[20])
{
	trie_node *next, *index, *data;
	int count;

	next = trie;  //start searching from the trie root

	if(next == NULL)  //trie is empty
	{
		cout << "Word not found in trie !!!!" << endl ;
		exit(1);
	}
	else
		index = next;// index - the current node from trie

	count = 0; // start searching for the first letter of the word(index of letter in word  is 0)
	while((index->NotLeaf == true) && (count < strlen(keyWord)) && (index->pChildren[keyWord[count]-
'a'] != NULL))
	{
		next  = index->pChildren[keyWord[count]-'a'];
		index = next;
		count ++ ;
	}

	if(next == NULL)
		cout << "Word not found in trie !!!!" << endl;
	else
	{
		data = next;
		   //the string is in a leaf
		if(!strcmp(data->word,keyWord))
			cout << "Key exists --- Word found in trie !!!!" << endl;
		else//the string is in the blank pointer(prefix for others words stored in trie)
			if((data->pChildren[26]) && !strcmp(data->pChildren[26]->word,keyWord))
				cout << "Key exists --- Word found in trie !!!!" << endl ;
			else
				cout << "Word not found in trie !!!!" << endl ;
   }
```

```
}

//function for inserting a string into the trie
trie_node *Insert(trie_node *trie, char keyWord[20])
{
	trie_node *next, *index, *parent;
	trie_node *new_leaf, *data, *new_index;
	trie_node *oldChildren, *newWord, *intern;

	int inWordIndex, prefixLenght, lenght = strlen(keyWord);

	next = trie;
	if(next == NULL) //trie empty
	{
		trie     =  NewIntern();
		new_leaf =  NewLeaf(keyWord);
		trie->pChildren[keyWord[0]-'a'] = new_leaf;

		return trie;
	}
	else
		index = next;

	inWordIndex = 0;
	while((inWordIndex < lenght) &&(index->NotLeaf == true)&&(index-
>pChildren[keyWord[inWordIndex]-'a'] != NULL))
	{
		parent = next;
		next   = index->pChildren[keyWord[inWordIndex]-'a'];
		index  = next;

		inWordIndex++;
	}


	if((inWordIndex < lenght) && (index->pChildren[keyWord[inWordIndex]-'a'] == NULL) && (index-
>NotLeaf == true))
	{
		new_index = NewLeaf(keyWord);
		index->pChildren[keyWord[inWordIndex]-'a'] = new_index;

		return trie;
	}
	else
		data=next;

	if(!strcmp(data->word,keyWord))
		cout << "Word already exists in trie !!!" << endl;
	else
	{
		oldChildren = parent->pChildren[keyWord[inWordIndex-1]-'a'];
		newWord = NewLeaf(keyWord);

		prefixLenght= strlen(keyWord);
		if(data->word[0] != '\0')
		  if(strlen(data->word) < prefixLenght)
```

```
                                        prefixLenght = strlen(data->word);
        }

        bool createIntern = false;
        while((inWordIndex <= prefixLenght)&&(((data->word[0] != '\0' )&& (data->word[inWordIndex-1] ==
keyWord[inWordIndex-1])) || (data->word[0] == '\0')))
        {
                intern = NewIntern();
                parent->pChildren[keyWord[inWordIndex-1]-'a'] = intern;
                parent->NotLeaf = true;
                parent = intern;
                inWordIndex++;

                createIntern  = true;
        }
        if(createIntern)
              inWordIndex--;

        if((inWordIndex != prefixLenght) || ((inWordIndex == prefixLenght)&&(strlen(keyWord) == strlen(data-
>word))))
        {
                parent->pChildren[data->word[inWordIndex]-'a'] = oldChildren;
                parent->pChildren[keyWord[inWordIndex]-'a']    = newWord;
        }
        else
                if(data->word[0] != '\0')// doar un cuv care il are ca prefix pe keyWord sau invers
                  if(strlen(data->word) <= prefixLenght)
                {
                        parent->pChildren[26] = oldChildren;
                        parent->pChildren[keyWord[prefixLenght]-'a'] = newWord;
                }
                else
                {
                        parent->pChildren[26] = newWord;
                        parent->pChildren[data->word[prefixLenght]-'a'] = oldChildren;
                }
                else// 2 sau mai multe cuv care au acelasi prefix
                {
                        for (int count = 0 ; count < 27;count++)
                             parent->pChildren[count] = oldChildren->pChildren[count];
                        parent->pChildren[26] = newWord;
                }

                return trie;
}

//function for displaying the words stored in the trie
void DisplayTrie(trie_node *trie, int nivel)
{
        int count;

        if(trie)
        {
                if (trie->NotLeaf != true) // if trie_node is a leaf(a word is stored in)
                {   // display the string at his level
                        for (count = 0; count <= nivel; count++)
```

```
                                        cout << " ";
                        cout << trie->word << endl;
                }
                 // display all the words stored through trie children
                for (count = 26; count >= 0; count--)
                        DisplayTrie(trie->pChildren[count], nivel + 4);
        }

}

int main()
{
        trie_node *trie;
        char UserInputWord[20], cont_insert=' ';
        int option = 0;  //stores the user's input(the chosen option)

        trie = NULL;

label_menu:
        while( option != 5)
        {
                //display menu
                cout << endl << "                Menu: " << endl;
                cout <<         "_____________________________________________" << endl;
           cout << "     1. Create tree\n     2. Insert node\n     3. Search for node\n     4. Display tree\n     5. Exit\n";

                //get user input
                cout << "\n\n\nInput choice: ";
                cin  >> option;


                switch (option)
                {
                case 1:    //Create tree
                        while(cont_insert != 'n')
                        {
                                // get user input string
                                cout << endl << "Insert word :";
                                cin  >> UserInputWord;

                                trie  = Insert(trie,UserInputWord);

                                cout << "\n Continue ? <y/n>";
                                cont_insert = getch();
                        }

                        break;
                case 2:    //Insert node
                        cout << endl << "Insert word :";
                        cin  >> UserInputWord;

                        Insert(trie,UserInputWord);

                        break;
                case 3:    //Search for node
                        cout << endl << "Searched word :";
```

```
                    cin  >> UserInputWord;

                    Find(trie,UserInputWord);

                    break;
            case 4:  //Display tree
                    DisplayTrie(trie,0);

                    break;
            case 5:  //Exit
                    break;
            default:
                     cout << "Choose from the displayed options (1-5) !!!";
                     goto label_menu;
          }//end switch
        }//end while

 system("PAUSE");
 return 0;
}
```

3. Assignments

Prefix trees are a bit of an overlooked data structure with lots of interesting possibilities. TRIE is an interesting data-structure used mainly for manipulating with Words in a language. TRIE has a wide variety of applications in :

- Spell checking. Word completion
- Data compression
- Computational biology
- Routing table for IP addresses
- Storing/Querying XML documents etc.

As a dictionary

Looking up if a word is in a trie takes O(n) operations, where n is the length of the word. Thus - for array implementations - the lookup speed doesn't change with increasing trie size. It has been used to store large dictionaries of English (say) words in spelling-checking programs and in natural-language "understanding" programs.

Simple spell checkers operate on individual words by comparing each of them against the contents of a dictionary, possibly performing stemming on the word. If the word is not found it is considered to be an error, and an attempt may be made to suggest a word that was likely to have been intended.

Word completion is straightforward to implement using a trie: simply find the node corresponding to the first few letters, and then collapse the subtree into a list of possible endings. This can be used in autocompleting user input in text editors.

Tries and Web Search Engines

The index of a search engine(collection of all searchable words) is stored into a compressed trie. Each leaf of the trie is associated with a word and has a list of pages (URLs) containing that word, called occurrence list.
The trie is kept in internal memory. The occurrence lists are kept in external memory and are ranked by relevance. Boolean queries for sets of words (e.g. Java and coffe) correspond to set operations (e.g. intersection) on the occurrence lists . Additional information retrievel techniques are used, such as:

- stopword elimination (e.g ignore “the” ,“a” ,“is”).

- Stemming (e.g. identify “add”, “adding”, “added”).
- Link analysis(recognize authoritative pages).

Tries an Internet Routers

Computers on the internet(hosts) are identified by a unique 32-bit IP(internet protocol) address, usually written in “dotted-quad-decimal” notation. E.g. www.google.com is 62.233.189.104. An organization uses a subset of IP addresses with the same prefix, e.g. IIDT uses 10.*.*.*

Data is sent to a host by fragmenting it into packets. Each packet carries the IP address of its destination.A router forwards packets to its neighbours using IP prefix matching rules. Routers use tries to do prefix matching.