

---

# STRING ALGORITHMS

CHAPTER

15



## 15.1 Introduction

To understand the importance of string algorithms let us consider the case of entering the URL (Uniform Resource Locator) in any browser (say, Internet Explorer, Firefox, or Google Chrome). You will observe that after typing the prefix of the URL, a list of all possible URLs is displayed. That means, the browsers are doing some internal processing and giving us the list of matching URLs. This technique is sometimes called *auto – completion*.

Similarly, consider the case of entering the directory name in the command line interface (in both *Windows* and *UNIX*). After typing the prefix of the directory name, if we press the *tab* button, we get a list of all matched directory names available. This is another example of auto completion.

In order to support these kinds of operations, we need a data structure which stores the string data efficiently. In this chapter, we will look at the data structures that are useful for implementing string algorithms.

We start our discussion with the basic problem of strings: given a string, how do we search a

substring (pattern)? This is called a *string matching* problem. After discussing various string matching algorithms, we will look at different data structures for storing strings.

## 15.2 String Matching Algorithms

In this section, we concentrate on checking whether a pattern  $P$  is a substring of another string  $T$  ( $T$  stands for text) or not. Since we are trying to check a fixed string  $P$ , sometimes these algorithms are called *exact string matching* algorithms. To simplify our discussion, let us assume that the length of given text  $T$  is  $n$  and the length of the pattern  $P$  which we are trying to match has the length  $m$ . That means,  $T$  has the characters from 0 to  $n - 1$  ( $T[0 \dots n - 1]$ ) and  $P$  has the characters from 0 to  $m - 1$  ( $P[0 \dots m - 1]$ ). This algorithm is implemented in  $C++$  as *strstr()*.

In the subsequent sections, we start with the brute force method and gradually move towards better algorithms.

- Brute Force Method
- Rabin-Karp String Matching Algorithm
- String Matching with Finite Automata
- KMP Algorithm
- Boyer-Moore Algorithm
- Suffix Trees

## 15.3 Brute Force Method

In this method, for each possible position in the text  $T$  we check whether the pattern  $P$  matches or not. Since the length of  $T$  is  $n$ , we have  $n - m + 1$  possible choices for comparisons. This is because we do not need to check the last  $m - 1$  locations of  $T$  as the pattern length is  $m$ . The following algorithm searches for the first occurrence of a pattern string  $P$  in a text string  $T$ .

### Algorithm

```
int BruteForceStringMatch (int T[], int n, int P[], int m) {
    for (int i = 0; i <= n - m; i++) {
        int j = 0;
        while (j < m && P[j] == T[i + j])
            j = j + 1;
        if (j == m)
            return i;
    }
    return -1;
}
```

Time Complexity:  $O((n - m + 1) \times m) \approx O(n \times m)$ . Space Complexity:  $O(1)$ .

## 15.4 Rabin-Karp String Matching Algorithm

In this method, we will use the hashing technique and instead of checking for each possible position in  $T$ , we check only if the hashing of  $P$  and the hashing of  $m$  characters of  $T$  give the same result.

Initially, apply the hash function to the first  $m$  characters of  $T$  and check whether this result and  $P$ 's hashing result is the same or not. If they are not the same, then go to the next character of  $T$  and again apply the hash function to  $m$  characters (by starting at the second character). If they are the same then we compare those  $m$  characters of  $T$  with  $P$ .

### Selecting Hash Function

At each step, since we are finding the hash of  $m$  characters of  $T$ , we need an efficient hash function. If the hash function takes  $O(m)$  complexity in every step, then the total complexity is  $O(n \times m)$ . This is worse than the brute force method because first we are applying the hash function and also comparing.

Our objective is to select a hash function which takes  $O(1)$  complexity for finding the hash of  $m$  characters of  $T$  every time. Only then can we reduce the total complexity of the algorithm. If the hash function is not good (worst case), the complexity of the Rabin-Karp algorithm is  $O((n - m + 1) \times m) \approx O(n \times m)$ . If we select a good hash function, the complexity of the Rabin-Karp algorithm complexity is  $O(m + n)$ . Now let us see how to select a hash function which can compute the hash of  $m$  characters of  $T$  at each step in  $O(1)$ .

For simplicity, let's assume that the characters used in string  $T$  are only integers. That means, all characters in  $T \in \{0,1,2,\dots,9\}$ . Since all of them are integers, we can view a string of  $m$  consecutive characters as decimal numbers. For example, string '61815' corresponds to the number 61815. With the above assumption, the pattern  $P$  is also a decimal value, and let us assume that the decimal value of  $P$  is  $p$ . For the given text  $T[0..n - 1]$ , let  $t(i)$  denote the decimal value of length- $m$  substring  $T[i..i + m - 1]$  for  $i = 0, 1, \dots, n - m - 1$ . So,  $t(i) == p$  if and only if  $T[i..i + m - 1] == P[0..m - 1]$ .

We can compute  $p$  in  $O(m)$  time using Horner's Rule as:

$$p = P[m - 1] + 10(P[m - 2] + 10(P[m - 3] + \dots + 10(P[1] + 10 P[0]) \dots))$$

The code for the above assumption is:

```

value = 0;
for (int i = 0; i < m-1; i++) {
    value = value * 10;
    value = value + P[i];
}

```

We can compute all  $t(i)$ , for  $i = 0, 1, \dots, n - m - 1$  values in a total of  $O(n)$  time. The value of  $t(0)$  can be similarly computed from  $T[0..m-1]$  in  $O(m)$  time. To compute the remaining values  $t(0)$ ,  $t(1), \dots, t(n - m - 1)$ , understand that  $t(i + 1)$  can be computed from  $t(i)$  in constant time.

$$t(i + 1) = 10 * (t(i) - 10^{m-1} * T[i]) + T[i + m - 1]$$

For example, if  $T = "123456"$  and  $m = 3$

$$\begin{aligned}
 t(0) &= 123 \\
 t(1) &= 10 * (123 - 100 * 1) + 4 = 234
 \end{aligned}$$

### Step by Step explanation

First : remove the first digit :  $123 - 100 * 1 = 23$

Second: Multiply by 10 to shift it :  $23 * 10 = 230$

Third: Add last digit :  $230 + 4 = 234$

The algorithm runs by comparing,  $t(i)$  with  $p$ . When  $t(i) == p$ , then we have found the substring  $P$  in  $T$ , starting from position  $i$ .

## 15.5 String Matching with Finite Automata

In this method we use the finite automata which is the concept of the Theory of Computation (ToC). Before looking at the algorithm, first let us look at the definition of finite automata.

### Finite Automata

A finite automaton  $F$  is a 5-tuple  $(Q, q_0, A, \Sigma, \delta)$ , where

- $Q$  is a finite set of states
- $q_0 \in Q$  is the start state
- $A \subseteq Q$  is a set of accepting states
- $\Sigma$  is a finite input alphabet
- $\delta$  is the transition function that gives the next state for a given current state and input

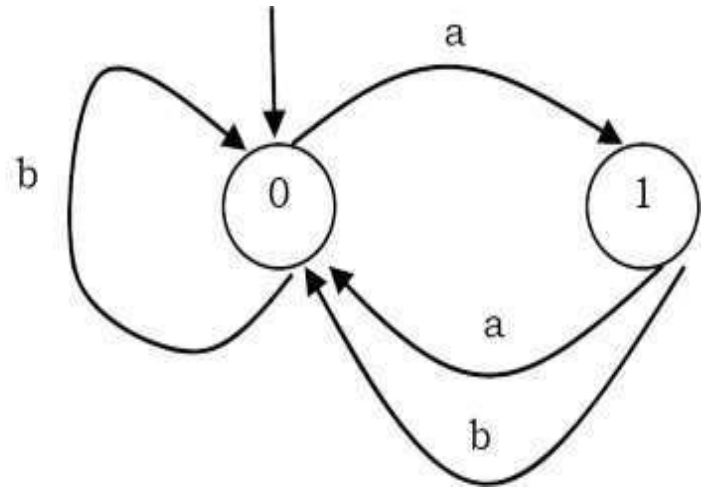
## How does Finite Automata Work?

- The finite automaton  $F$  begins in state  $q_0$
- Reads characters from  $\Sigma$  one at a time
- If  $F$  is in state  $q$  and reads input character  $a$ ,  $F$  moves to state  $\delta(q,a)$
- At the end, if its state is in  $A$ , then we say,  $F$  accepted the input string read so far
- If the input string is not accepted it is called the rejected string

**Example:** Let us assume that  $Q = \{0,1\}, q_0 = 0, A = \{1\}, \Sigma = \{a, b\}$ .  $\delta(q,d)$  as shown in the transition table/diagram. This accepts strings that end in an odd number of  $a$ 's; e.g.,  $abbaaa$  is accepted,  $aa$  is rejected.

Input		
State	a	b
0	1	0
1	0	0

Transition Function/Table



## Important Notes for Constructing the Finite Automata

For building the automata, first we start with the initial state. The FA will be in state  $k$  if  $k$  characters of the pattern have been matched. If the next text character is equal to the pattern character  $c$ , we have matched  $k + 1$  characters and the FA enters state  $k + 1$ . If the next text character is not equal to the pattern character, then the FA goes to a state  $0, 1, 2, \dots$  or  $k$ , depending on how many initial pattern characters match the text characters ending with  $c$ .

## Matching Algorithm

Now, let us concentrate on the matching algorithm.

- For a given pattern  $P[0.. m - 1]$ , first we need to build a finite automaton  $F$ 
  - The state set is  $Q = \{0, 1, 2, \dots, m\}$
  - The start state is 0
  - The only accepting state is  $m$
  - Time to build  $F$  can be large if  $\Sigma$  is large
- Scan the text string  $T[0.. n - 1]$  to find all occurrences of the pattern  $P[0.. m - 1]$

- String matching is efficient:  $\Theta(n)$ 
  - Each character is examined exactly once
  - Constant time for each character
  - But the time to compute  $\delta$  (transition function) is  $O(m|\Sigma|)$ . This is because  $\delta$  has  $O(m|\Sigma|)$  entries. If we assume  $|\Sigma|$  is constant then the complexity becomes  $O(m)$ .

### Algorithm:

```
//Input: Pattern string P[0..m-1],  $\delta$  and F
//Goal: All valid shifts displayed
FiniteAutomataStringMatcher(int P[], int m, F,  $\delta$ ) {
    q = 0;
    for (int i = 0; i < m; i++)
        q =  $\delta(q, T[i])$ ;
    if (q == m)
        printf("Pattern occurs with shift: %d", i-m);
}
```

Time Complexity:  $O(m)$ .

## 15.6 KMP Algorithm

As before, let us assume that  $T$  is the string to be searched and  $P$  is the pattern to be matched. This algorithm was presented by Knuth, Morris and Pratt. It takes  $O(n)$  time complexity for searching a pattern. To get  $O(n)$  time complexity, it avoids the comparisons with elements of  $T$  that were previously involved in comparison with some element of the pattern  $P$ .

The algorithm uses a table and in general we call it *prefix function* or *prefix table* or *fail function*  $F$ . First we will see how to fill this table and later how to search for a pattern using this table. The prefix function  $F$  for a pattern stores the knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern  $P$ . It means that this table can be used for avoiding backtracking on the string  $T$ .

### Prefix Table

```

int F[]; //assume F is a global array
void Prefix-Table(int P[], int m) {
    int i=1,j=0, F[0]=0;
    while(i<m) {
        if(P[i]==P[j]) {
            F[i]=j+1;
            i++;
            j++;
        }
        else if(j>0)
            j=F[j-1];
        else {
            F[i]=0;
            i++;
        }
    }
}

```

As an example, assume that  $P = a\ b\ a\ b\ a\ c\ a$ . For this pattern, let us follow the step-by-step instructions for filling the prefix table  $F$ . Initially:  $m = \text{length}[P] = 7, F[0] = 0$  and  $F[1] = 0$ .

**Step 1:**  $i = 1, j = 0, F[1] = 0$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0					

**Step 2:**  $i = 2, j = 0, F[2] = 1$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1				

**Step 3:**  $i = 3, j = 1, F[3] = 2$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2			

**Step 4:**  $i = 4, j = 2, F[4] = 3$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3		

**Step 5:**  $i = 5, j = 3, F[5] = 1$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	

**Step 6:**  $i = 6, j = 1, F[6] = 1$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

At this step the filling of the prefix table is complete.

## Matching Algorithm

The KMP algorithm takes pattern  $P$ , string  $T$  and prefix function  $F$  as input, and finds a match of  $P$  in  $T$ .



```

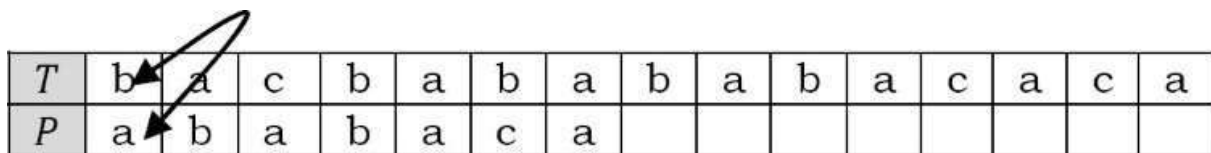
int KMP(char T[], int n, int P[], int m) {
    int i=0,j=0;
    Prefix-Table(P,m);
    while(i<n) {
        if(T[i]==P[j]) {
            if(j==m-1)
                return i-j;
            else {
                i++;
                j++;
            }
        }
        else if(j>0)
            j=F[j-1];
        else
            i++;
    }
    return -1;
}

```

Time Complexity:  $O(m + n)$ , where  $m$  is the length of the pattern and  $n$  is the length of the text to be searched. Space Complexity:  $O(m)$ .

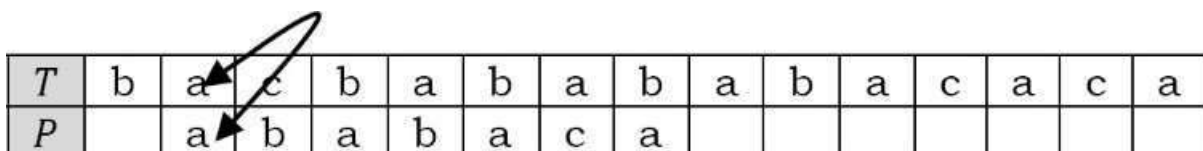
Now, to understand the process let us go through an example. Assume that  $T = b a c b a b a b a b a c a c a$  &  $P = a b a b a c a$ . Since we have already filled the prefix table, let us use it and go to the matching algorithm. Initially:  $n = \text{size of } T = 15$ ;  $m = \text{size of } P = 7$ .

**Step 1:**  $i = 0, j = 0$ , comparing  $P[0]$  with  $T[0]$ .  $P[0]$  does not match with  $T[0]$ .  $P$  will be shifted one position to the right.



T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P	a	b	a	b	a	c	a								

**Step 2 :**  $i = 1, j = 0$ , comparing  $P[0]$  with  $T[1]$ .  $P[0]$  matches with  $T[1]$ . Since there is a match,  $P$  is not shifted.



T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P		a	b	a	b	a	c	a							

**Step 3:**  $i = 2, j = 1$ , comparing  $P[1]$  with  $T[2]$ .  $P[1]$  does not match with  $T[2]$ . Backtracking on  $P$ ,

comparing  $P[0]$  and  $T[2]$ .

$T$	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
$P$		a	b	a	b	a	c	a							

**Step 4:**  $i = 3, j = 0$ , comparing  $P[0]$  with  $T[3]$ .  $P[0]$  does not match with  $T[3]$ .

$T$	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
$P$				a	b	a	b	a	c	a					

**Step 5:**  $i = 4, j = 0$ , comparing  $P[0]$  with  $T[4]$ .  $P[0]$  matches with  $T[4]$ .

$T$	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
$P$					a	b	a	b	a	c	a				

**Step 6:**  $i = 5, j = 1$ , comparing  $P[1]$  with  $T[5]$ .  $P[1]$  matches with  $T[5]$ .

$T$	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
$P$					a	b	a	b	a	c	a				

**Step 7:**  $i = 6, j = 2$ , comparing  $P[2]$  with  $T[6]$ .  $P[2]$  matches with  $T[6]$ .

$T$	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
$P$					a	b	a	b	a	c	a				

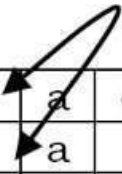
**Step 8:**  $i = 7, j = 3$ , comparing  $P[3]$  with  $T[7]$ .  $P[3]$  matches with  $T[7]$ .

$T$	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
$P$					a	b	a	b	a	c	a				

**Step 9:**  $i = 8, j = 4$ , comparing  $P[4]$  with  $T[8]$ .  $P[4]$  matches with  $T[8]$ .


$T$	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
$P$					a	b	a	b	a	c	a				

**Step 10:**  $i = 9, j = 5$ , comparing  $P[5]$  with  $T[9]$ .  $P[5]$  does not match with  $T[9]$ . Backtracking on  $P$ , comparing  $P[4]$  with  $T[9]$  because after mismatch  $= F[4] = 3$ .



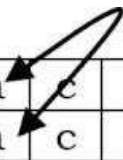
$T$	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
$P$					a	b	a	b	a	c	a				

Comparing  $P[3]$  with  $T[9]$ .




$T$	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
$P$							a	b	a	b	a	c	a		

**Step 11:**  $i = 10, j = 4$ , comparing  $P[4]$  with  $T[10]$ .  $P[4]$  matches with  $T[10]$ .



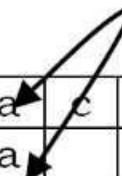
$T$	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
$P$							a	b	a	b	a	c	a		

**Step 12:**  $i = 11, j = 5$ , comparing  $P[5]$  with  $T[11]$ .  $P[5]$  matches with  $T[11]$ .



$T$	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
$P$							a	b	a	b	a	c	a		

**Step 13:**  $i = 12, j = 6$ , comparing  $P[6]$  with  $T[12]$ .  $P[6]$  matches with  $T[12]$ .



$T$	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
$P$							a	b	a	b	a	c	a		

Pattern  $P$  has been found to completely occur in string  $T$ . The total number of shifts that took place for the match to be found are:  $i - m = 13 - 7 = 6$  shifts.

#### Notes:

- KMP performs the comparisons from left to right
- KMP algorithm needs a preprocessing (prefix function) which takes  $O(m)$  space and time complexity
- Searching takes  $O(n + m)$  time complexity (does not depend on alphabet size)

## 15.7 Boyer-Moore Algorithm

Like the KMP algorithm, this also does some pre-processing and we call it *last function*. The algorithm scans the characters of the pattern from right to left beginning with the rightmost character. During the testing of a possible placement of pattern  $P$  in  $T$ , a mismatch is handled as follows: Let us assume that the current character being matched is  $T[i] = c$  and the corresponding pattern character is  $P[j]$ . If  $c$  is not contained anywhere in  $P$ , then shift the pattern  $P$  completely past  $T[i]$ . Otherwise, shift  $P$  until an occurrence of character  $c$  in  $P$  gets aligned with  $T[i]$ . This technique avoids needless comparisons by shifting the pattern relative to the text.

The *last* function takes  $O(m + |\Sigma|)$  time and the actual search takes  $O(nm)$  time. Therefore the worst case running time of the Boyer-Moore algorithm is  $O(nm + |\Sigma|)$ . This indicates that the worst-case running time is quadratic, in the case of  $n == m$ , the same as the brute force algorithm.

- The Boyer-Moore algorithm is very fast on the large alphabet (relative to the length of the pattern).
- For the small alphabet, Boyer-Moore is not preferable.
- For binary strings, the KMP algorithm is recommended.
- For the very shortest patterns, the brute force algorithm is better.

## 15.8 Data Structures for Storing Strings

If we have a set of strings (for example, all the words in the dictionary) and a word which we want to search in that set, in order to perform the search operation faster, we need an efficient way of storing the strings. To store sets of strings we can use any of the following data structures.

- Hashing Tables
- Binary Search Trees
- Tries
- Ternary Search Trees

## 15.9 Hash Tables for Strings

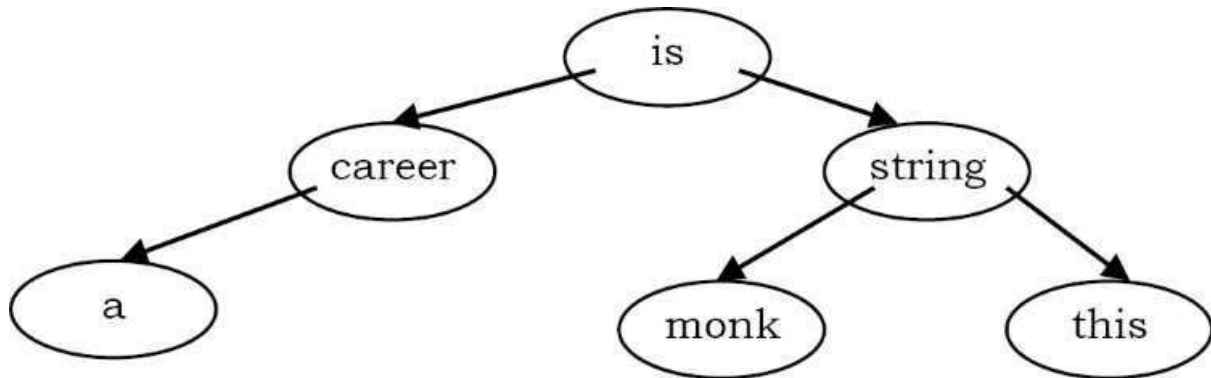
As seen in the *Hashing* chapter, we can use hash tables for storing the integers or strings. In this case, the keys are nothing but the strings. The problem with hash table implementation is that we lose the ordering information – after applying the hash function, we do not know where it will map to. As a result, some queries take more time. For example, to find all the words starting with the letter “K”, with hash table representation we need to scan the complete hash table. This is because the hash function takes the complete key, performs hash on it, and we do not know the location of each word.

## 15.10 Binary Search Trees for Strings

In this representation, every node is used for sorting the strings alphabetically. This is possible because the strings have a natural ordering: *A* comes before *B*, which comes before *C*, and so on. This is because words can be ordered and we can use a Binary Search Tree (BST) to store and retrieve them. For example, let us assume that we want to store the following strings using BSTs:

*this is a career monk string*

For the given string there are many ways of representing them in BST. One such possibility is shown in the tree below.



## Issues with Binary Search Tree Representation

This method is good in terms of storage efficiency. But the disadvantage of this representation is that, at every node, the search operation performs the complete match of the given key with the node data, and as a result the time complexity of the search operation increases. So, from this we can say that BST representation of strings is good in terms of storage but not in terms of time.

## 15.11 Tries

Now, let us see the alternative representation that reduces the time complexity of the search operation. The name *trie* is taken from the word re"trie".

### What is a Trie?

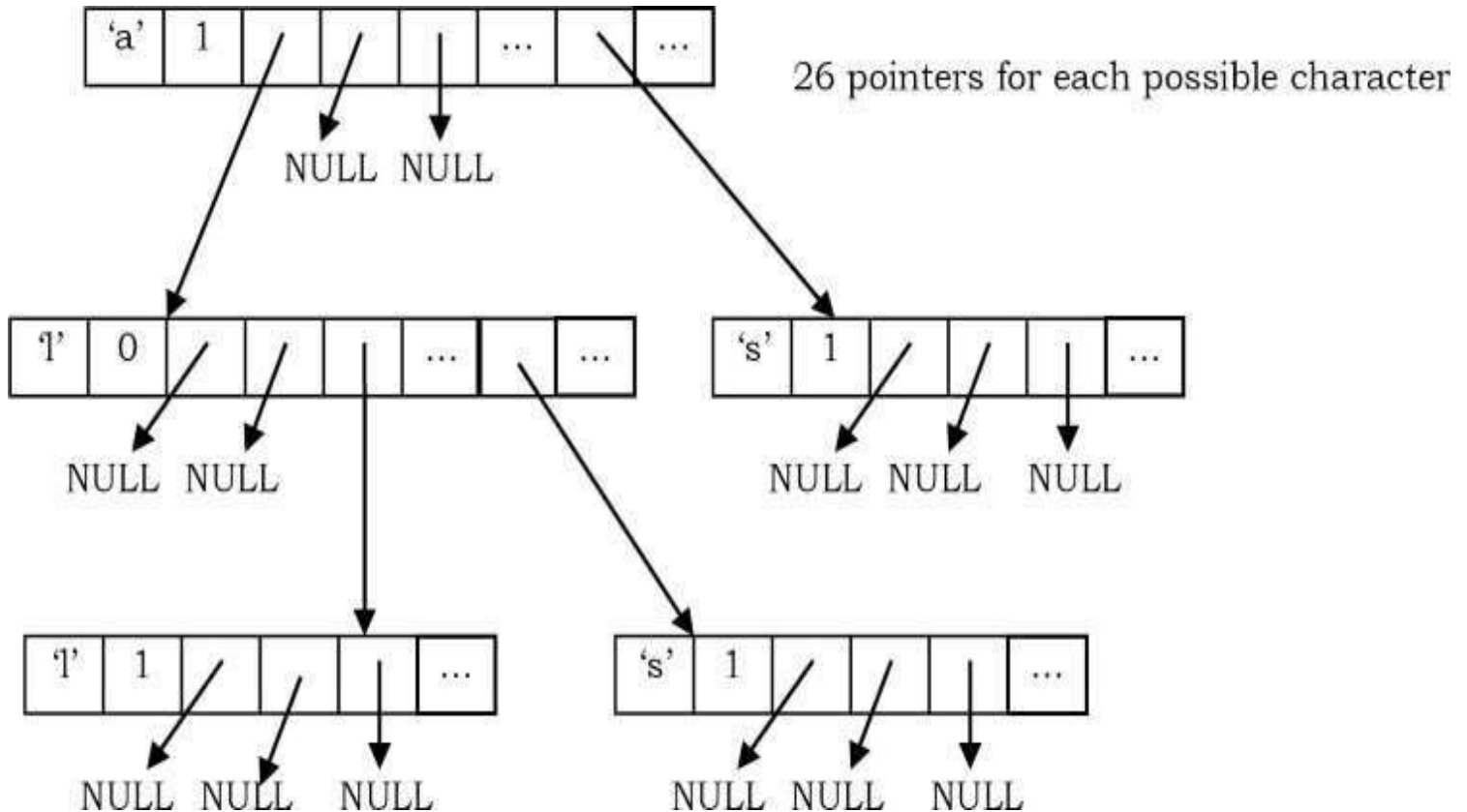
A *trie* is a tree and each node in it contains the number of pointers equal to the number of characters of the alphabet. For example, if we assume that all the strings are formed with English alphabet characters "a" to "z" then each node of the trie contains 26 pointers. A trie data structure can be declared as:

```

struct TrieNode {
    char data;           // Contains the current node character.
    int is_End_Of_String; // Indicates whether the string formed from root to
                        // current node is a string or not
    struct TrieNode *child[26]; // Pointers to other tri nodes
};

```

Suppose we want to store the strings “a”, ”all”, ”als”, and “as” : trie for these strings will look like:



## Why Tries?

The tries can insert and find strings in  $O(L)$  time (where  $L$  represents the length of a single word). This is much faster than hash table and binary search tree representations.

## Trie Declaration

The structure of the TrieNode has data (char), is\_End\_Of\_String (boolean), and has a collection of child nodes (Collection of TrieNodes). It also has one more method called subNode(char). This method takes a character as argument and will return the child node of that character type if that is present. The basic element - TrieNode of a TRIE data structure looks like this:

```

struct TrieNode {
    char data;
    int is_End_Of_String;
    struct TrieNode *child[];
};

struct TrieNode *TrieNode subNode(struct TrieNode *root, char c){
    if(root != NULL){
        for(int i=0; i < 26; i++){
            if(root.child[i]→data == c)
                return root.child[i];
        }
    }
    return NULL;
}

```

Now that we have defined our TrieNode, let's go ahead and look at the other operations of TRIE. Fortunately, the TRIE data structure is simple to implement since it has two major methods: insert() and search(). Let's look at the elementary implementation of both these methods.

## Inserting a String in Trie

To insert a string, we just need to start at the root node and follow the corresponding path (path from root indicates the prefix of the given string). Once we reach the NULL pointer, we just need to create a skew of tail nodes for the remaining characters of the given string.

```

void InsertInTrie(struct TrieNode *root, char *word) {
    if(!*word) return;
    if(!root) {
        struct TrieNode *newNode = (struct TrieNode *) malloc (sizeof(struct TrieNode *));
        newNode→data=*word;
        for(int i =0; i<26; i++)
            newNode→child[i]=NULL;
        if(!*(word+1))
            newNode→is_End_Of_String = 1;
        else newNode→child[*word] = InsertInTrie(newNode→child[*word], word+1);
        return newNode;
    }
    root→child[*word] = InsertInTrie(root→child[*word], word+1);
    return root;
}

```

Time Complexity:  $O(L)$ , where  $L$  is the length of the string to be inserted.

**Note:** For real dictionary implementation, we may need a few more checks such as checking whether the given string is already there in the dictionary or not.

## Searching a String in Trie

The same is the case with the search operation: we just need to start at the root and follow the pointers. The time complexity of the search operation is equal to the length of the given string that want to search.

```
int SearchInTrie(struct TrieNode *root, char *word) {
    if(!root)
        return -1;
    if(!*word) {
        if(root->is_End_Of_String)
            return 1;
        else return -1;
    }
    if(root->data == *word)
        return SearchInTrie(root->child[*word], word+1);
    else return -1;
}
```

Time Complexity:  $O(L)$ , where  $L$  is the length of the string to be searched.

## Issues with Tries Representation

The main disadvantage of tries is that they need lot of memory for storing the strings. As we have seen above, for each node we have too many node pointers. In many cases, the occupancy of each node is less. The final conclusion regarding tries data structure is that they are faster but require huge memory for storing the strings.

**Note:** There are some improved tries representations called *trie compression techniques*. But, even with those techniques we can reduce the memory only at the leaves and not at the internal nodes.

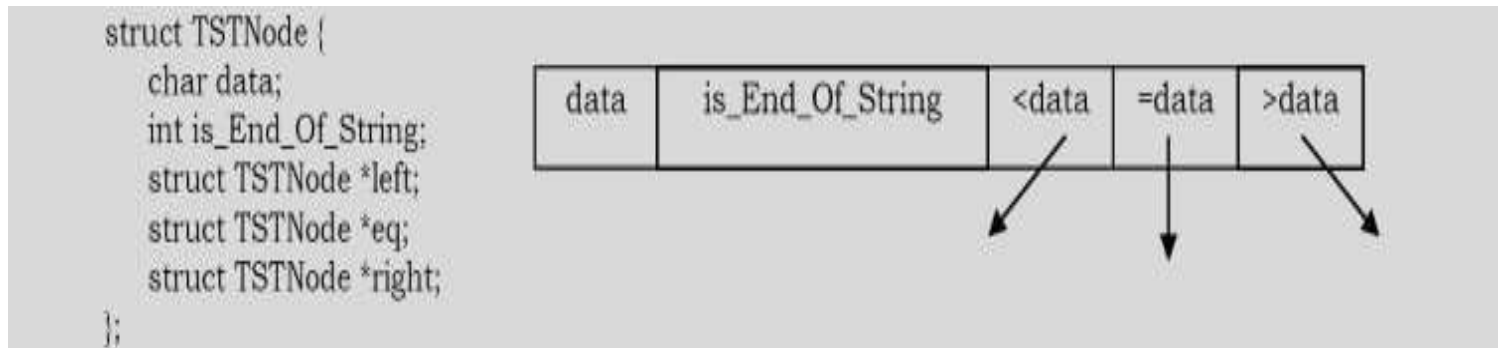
## 15.12 Ternary Search Trees

This representation was initially provided by Jon Bentley and Sedgewick. A ternary search tree takes the advantages of binary search trees and tries. That means it combines the memory



efficiency of BSTs and the time efficiency of tries.

## Ternary Search Trees Declaration

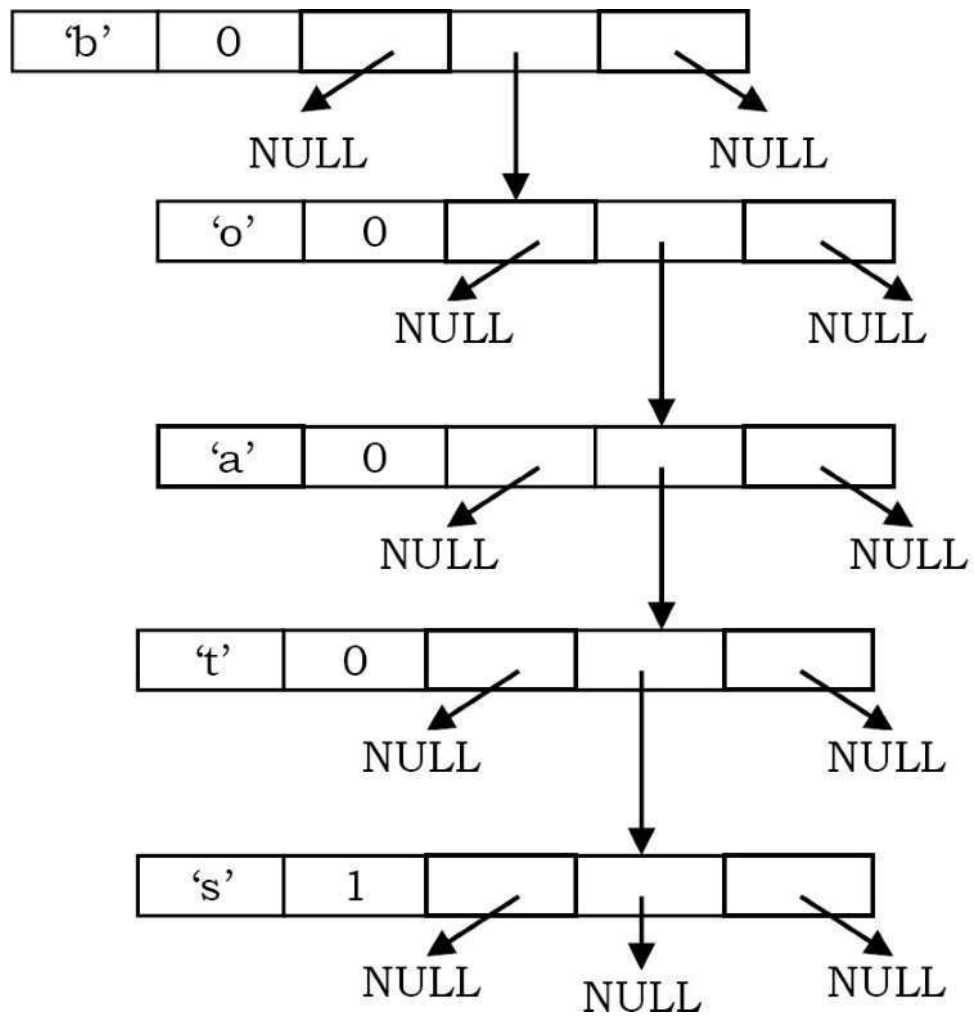


The Ternary Search Tree (TST) uses three pointers:

- The *left* pointer points to the TST containing all the strings which are alphabetically less than *data*.
- The *right* pointer points to the TST containing all the strings which are alphabetically greater than *data*.
- The *eq* pointer points to the TST containing all the strings which are alphabetically equal to *data*. That means, if we want to search for a string, and if the current character of the input string and the *data* of current node in TST are the same, then we need to proceed to the next character in the input string and search it in the subtree which is pointed by *eq*.

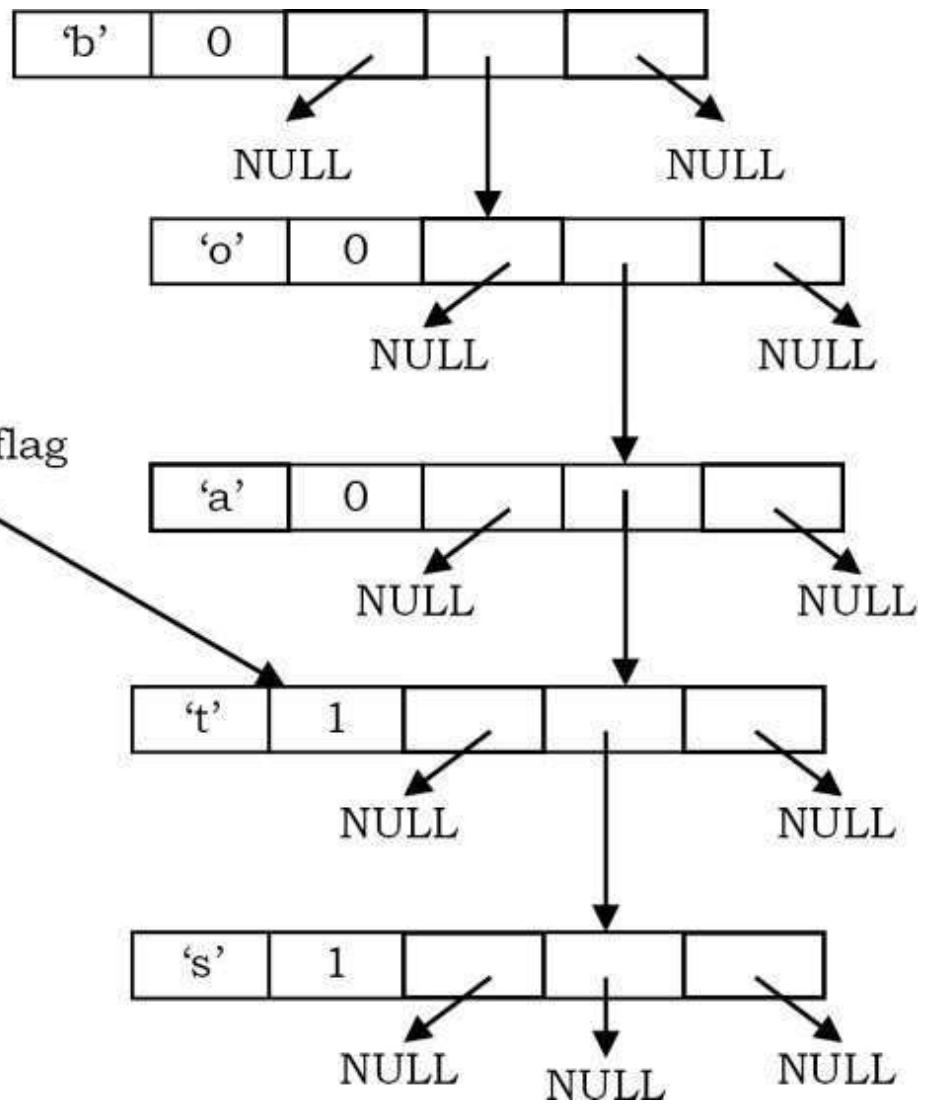
## Inserting strings in Ternary Search Tree

For simplicity let us assume that we want to store the following words in TST (also assume the same order): *boats*, *boat*, *bat* and *bats*. Initially, let us start with the *boats* string.

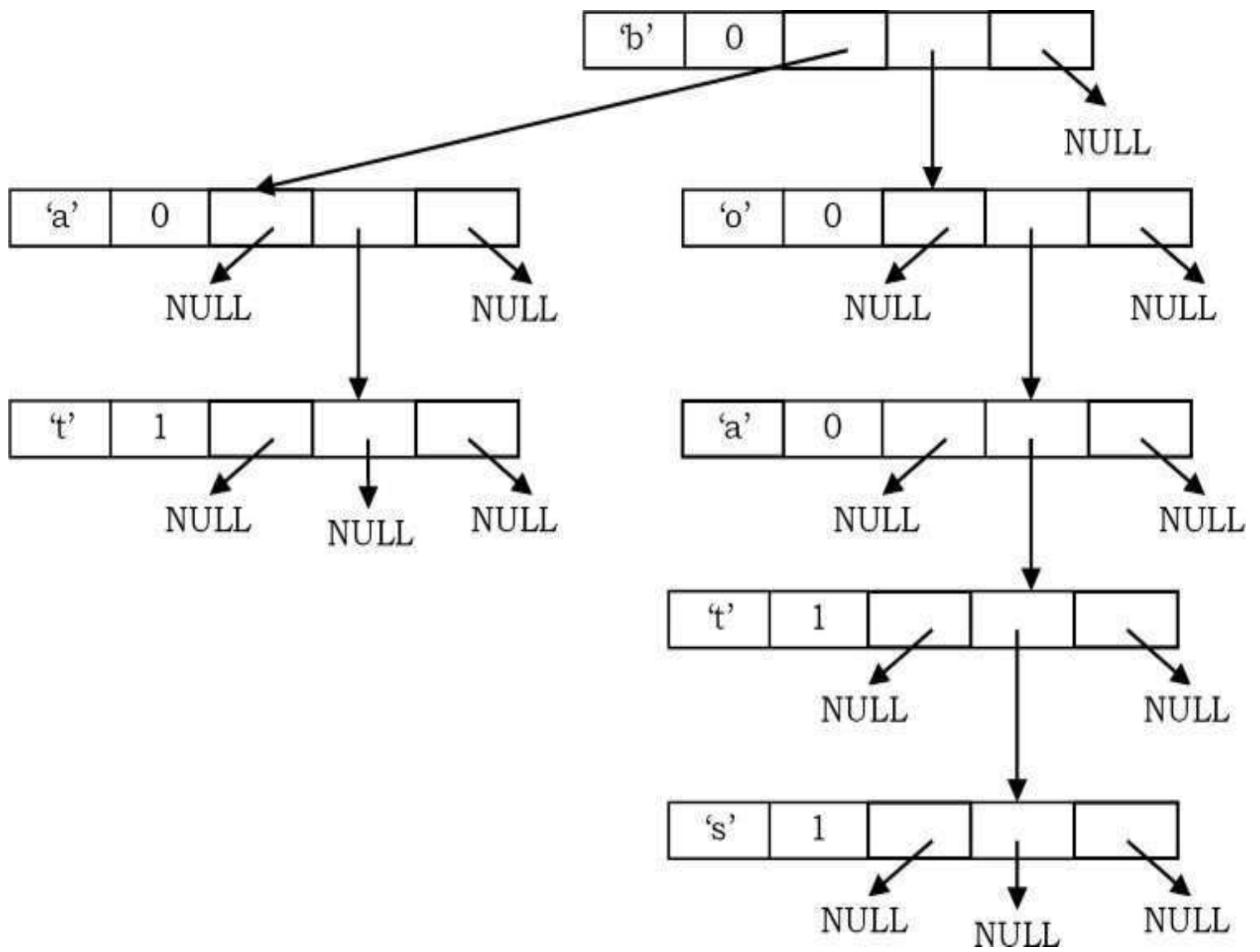


Now if we want to insert the string *boat*, then the TST becomes [the only change is setting the *is\_End\_Of\_String* flag of “t” node to 1]:

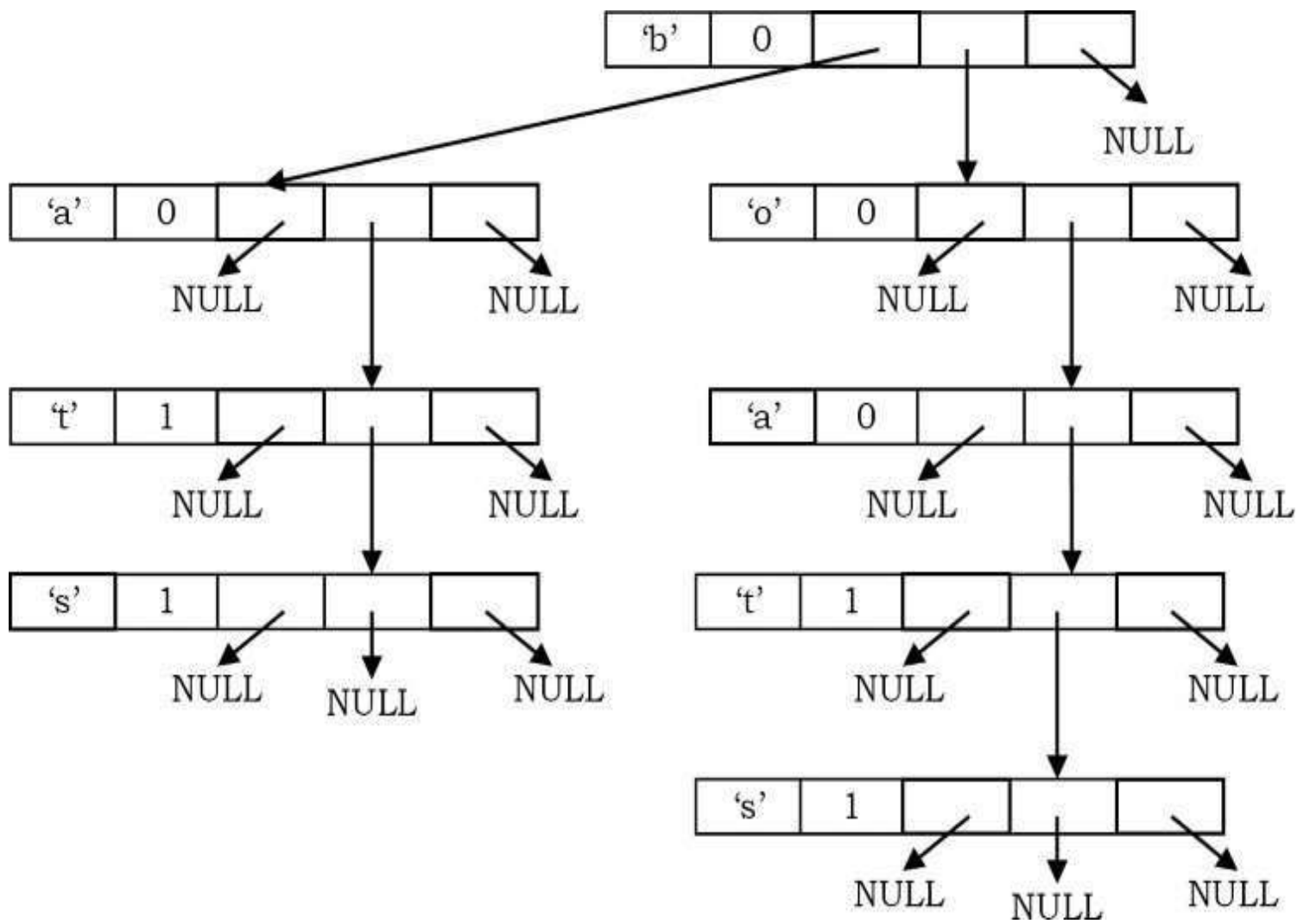
Set the *is\_End\_Of\_String* flag to 1.



Now, let us insert the next string: *bat*



Now, let us insert the final word: *bats*.



Based on these examples, we can write the insertion algorithm as below. We will combine the insertion operation of BST and tries.

```

struct TSTNode *InsertInTST(struct TSTNode *root, char *word) {
    if(root == NULL) {
        root = (struct TSTNode *) malloc(sizeof(struct TSTNode));
        root->data = *word;
        root->is_End_Of_String = 1;
        root->left = root->eq = root->right = NULL;
    }

    if(*word < root->data)
        root->left = InsertInTST (root->left, word);
    else if(*word == root->data) {
        if(*(word+1))
            root->eq = InsertInTST (root->eq, word+1);
        else root->is_End_Of_String = 1;
    }
    else root->right = InsertInTST (root->right, word);
    return root;
}

```

Time Complexity:  $O(L)$ , where  $L$  is the length of the string to be inserted.

## Searching in Ternary Search Tree

If after inserting the words we want to search for them, then we have to follow the same rules as that of binary search. The only difference is, in case of match we should check for the remaining characters (in *eq* subtree) instead of return. Also, like BSTs we will see both recursive and non-recursive versions of the search method.

```

int SearchInTSTRecursive(struct TSTNode *root, char *word) {
    if(!root)
        return -1;
    if(*word < root->data)
        return SearchInTSTRecursive(root->left, word);
    else if(*word > root->data)
        return SearchInTSTRecursive(root->right, word);
    else {
        if(root->is_End_Of_String && *(word+1)==0)
            return 1;
        return SearchInTSTRecursive(root->eq, ++word);
    }
}

int SearchInTSTNon-Recursive(struct TSTNode *root, char *word) {
    while (root) {
        if(*word < root->data)
            root = root->left;
        else if(*word == root->data) {
            if(root->is_End_Of_String && *(word+1) == 0)
                return 1;
            word++;
            root = root->eq;
        }
        else root = root->right;
    }
    return -1;
}

```

Time Complexity:  $O(L)$ , where  $L$  is the length of the string to be searched.

## Displaying All Words of Ternary Search Tree

If we want to print all the strings of TST we can use the following algorithm. If we want to print them in sorted order, we need to follow the inorder traversal of TST.

```

char word[1024];
void DisplayAllWords(struct TSTNode *root) {
    if(!root)
        return;
    DisplayAllWords(root->left);
    word[i] = root->data;
    if(root->is_End_Of_String) {
        word[i] = '\0';
        printf("%c", word);
    }
    i++;
    DisplayAllWords(root->eq);

    i--;
    DisplayAllWords(root->right);
}

```

## Finding the Length of the Largest Word in TST

This is similar to finding the height of the BST and can be found as:

```

int MaxLengthOfLargestWordInTST(struct TSTNode *root) {
    if(!root)
        return 0;
    return Max(MaxLengthOfLargestWordInTST(root->left),
               MaxLengthOfLargestWordInTST(root->eq)+1,
               MaxLengthOfLargestWordInTST(root->right));
}

```

## 15.13 Comparing BSTs, Tries and TSTs

- Hash table and BST implementation stores complete the string at each node. As a result they take more time for searching. But they are memory efficient.
- TSTs can grow and shrink dynamically but hash tables resize only based on load factor.
- TSTs allow partial search whereas BSTs and hash tables do not support it.
- TSTs can display the words in sorted order, but in hash tables we cannot get the sorted order.
- Tries perform search operations very fast but they take huge memory for storing the string.