mood-book

moodbanao.net

UNIT- 5

Brute - Force Algorithm (Naive algorithm) :

It is a type of algorithm that are extremely simple and straight forward approach to solve a problem. This algorithm uses a large number of patterns to solve a problem and it depends on computing power to achieve results.

Advantages:-

1. This is used by default to solve some problems like sorting, searching, matrix multiplication, binomial expansion, etc.
2. used to Solve Smaller instances of a larger problem.

Disadvantages:-

1. Inefficient when deals with homogeneous problems of higher complexity.
2. Not suitable for hierarchical structured problems and for problems involves logical operations.

Scanned by CamScanner

moodbanao.net

Brute-Force String matching:

Given a String of n characters called text and String of m Characters called pattern, find a Substring of the text that matches the pattern.

This algorithm aligns the pattern against the first m characters of text and Start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match or mismatching pair is encountered. Later, Shifts the position of pattern One position to the right and resume the character Comparisons, Starting again with the first character of the pattern and its counterpart in the text.

Note that the last position in the text that can Still be a beginning of a matching Substring is n-m. Beyond that position, there are not enough characters to match the entire pattern. Hence, the algorithm need not make any Comparisons there.

Scanned by CamScanner

Eg:

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

The given text is "NOBODY_NOTICED_HIM" and the given pattern of string to compare is "NOT".

The algorithm shifts the pattern almost always after a single character comparison. The algorithm may have to make all m comparisons before shifting the pattern, and this happens for each of n-m+1 tries. Thus the algorithm makes m(n-m+1) character comparisons, which produces O(nm) time complexity for worse case.

Algorithm: BruteForceStringmatching(T[0.... n-1], P[0... m-1])

```
for i ← 0 to n-m do
    j ← 0
    while j < m and P[j] = T[i+j] do
        j ← j+1
    if j = m return i
return -1.
```

I/P: T[0.... n-1], represents text
P[0... m-1] represents pattern
O/P: Index of first char in a text that matches substring.

Scanned by CamScanner

moodbanao.net

Boyer-Moore Algorithm:

This is the most efficient String-matching algorithm because it works fastest when the alphabet is moderately sized and the pattern is relatively long.

The algorithm Scans the characters of the pattern from right to left beginning with rightmost character. During the testing of a possible placement of pattern P against text T, a mismatch of text character T[i]=c with the corresponding pattern character P[j] is handled as follows: 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 likely to avoid lot of needless Comparisons by significantly shifting pattern relative to text. For deciding the possible shifts, it uses two preprocessing strategies simultaneously, to reduce the search.

- * Bad character Heuristics
- * Good Suffix Heuristics.

Scanned by CamScanner

moodbanao.net

Bad character Heuristics :-

The Idea is simple that the character of the text which doesn't match with the current character of the pattern is called the bad character.

Upon mismatch, we shift the pattern until,

a) The mismatch becomes a match

b) pattern p move past the mismatched character.

Case 1: Mismatch become match.

Lookup the position of last occurrence of mismatching character in the pattern and if mismatching character exist in pattern then we will shift the pattern such that it get aligned to the mismatching character in text T.

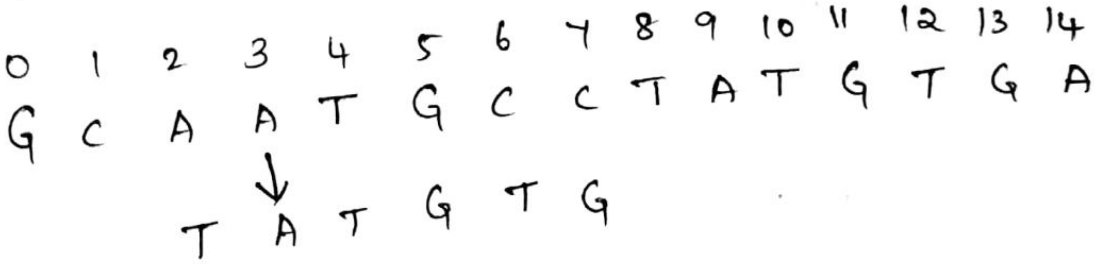
Eg:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	G	C	A	A	T	G	C	C	T	A	T	G	T	G	A
				↯	↓	↓									
	T	A	T	G	T	G									

Scanned by CamScanner

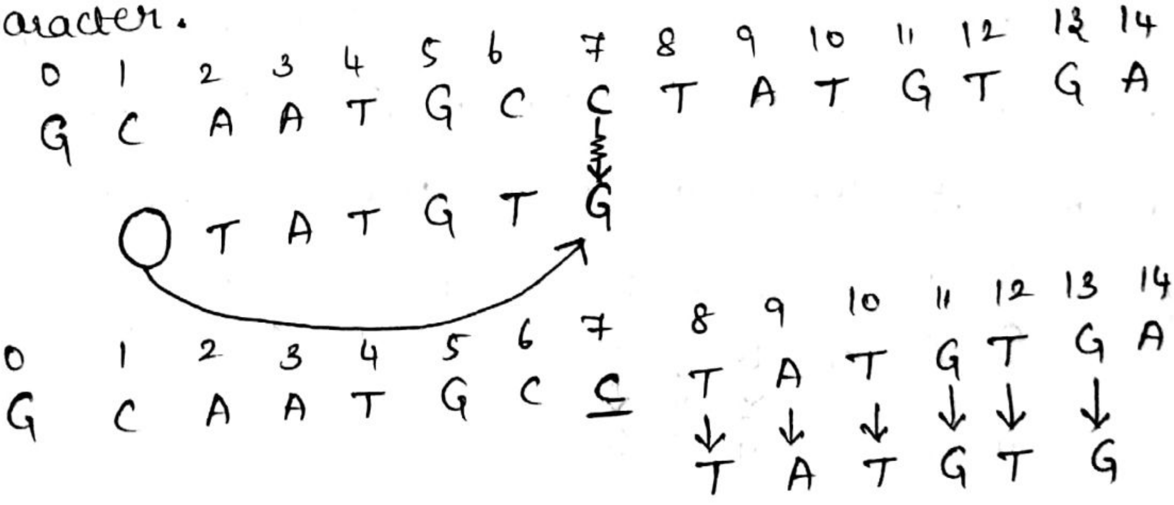
moodbanao.net

We got a mismatch at position 3. Here, the mismatching character is "A". Now, search for the last occurrence of "A" in the pattern. we got "A" at position 1 in pattern, that is the last occurrence of it. Now Shift pattern 2 times so, that "A" in pattern get aligned with "A" in text.



Case 2: pattern move past the mismatch character.

Lookup the position of last occurrence of mismatching character in pattern and if character does not exist we will shift pattern past the mismatching character.



Scanned by CamScanner

Here, mismatch at position 7 and the mismatching character "c" does not exist in the pattern so, shift the pattern past to the position 7 and eventually we got a perfect match of pattern. The bad character heuristic takes $O(n/m)$ time in the best case, and $O(mn)$ time in worst case.

COMPUTE_LAST_OCCURRENCE_FUNCTION (p, m, Σ):

```
1. for each character $a \in \Sigma$
2. do $\lambda[a] = 0$
3. for $j \leftarrow 1$ to m
4. do $\lambda[p[j]] \leftarrow j$
5. Return $\lambda$.
```

Good Suffix Heuristics:

Let t be substring of text T which is matched with substring of pattern p. Now, shift pattern until:

1) Another occurrence of t in p matched with t in T.

2) A prefix of p, matches with suffix to t

3) p moves past t.

Scanned by CamScanner

Case 1: Another occurrence of t in p matched with t in T.

Pattern p might contain few more occurrences of t. In such case, we will try to shift the pattern to align that occurrence with t in text T.

Eg:

0	1	2	3	4	5	6	7	8	9	10
A	B	A	A	B	A	B	A	C	B	A
			↓	↓						
C	A	B	A	B						

We have got a substring t of text T matched with pattern P before mismatch at index 2. Now, search for the occurrence of t ("AB") in p. We have found an occurrence starting at position 1. So, right shift of the pattern 2 times to align t in p with t in T, is done.

0	1	2	3	4	5	6	7	8	9	10
A	B	A	A	B	A	B	A	C	B	A
			A	B	A	B				

Case 2: A prefix of P, which matches with suffix of t in T.

Sometimes, there is no occurrence of t in p at all. In such cases we can search for some

moodbanao.net

Suffix of t matching with some prefix of P and try to align them by shifting P.

Eg:

0	1	2	3	4	5	6	7	8	9	10
A	A	B	A	B	A	B	A	C	B	A
		↓	↓	↓						
A	B	B	A	B						

In this, ("BAB") matched with P at index 2-4 before mismatch. As, there exists no occurrences of t in P, search for some prefix of P which matches with some suffix of t. we have found prefix "AB" starting at index 0 which matches not with whole t but the suffix of t "AB" starting at index 3. So, now shift pattern 3 times to align prefix with suffix.

0	1	2	3	4	5	6	7	8	9	10
A	A	B	A	B	A	B	A	C	B	A
			↓	↓						
			A	B	B	A	B			

Case 3: P moves past t.

If two cases above are not satisfied, we will shift the pattern past the t.

Eg:

0	1	2	3	4	5	6	7	8	9	10
A	A	C	A	B	A	B	A	C	B	A
			✗	↓						
C	B	A	A	B						

Scanned by CamScanner

moodbanao.net

0	1	2	3	4	5	6	7	8	9	10
A	B	A	A	B	A	B	A	C	B	A
					C	B	A	A	B.	

Here, we can never find any perfect match before index 4, so we will shift the p past the t.

Knuth - Morris - Pratt algorithm:

KMP algorithm searches for occurrences of a word "w" within a main text "S."

The naive approach doesn't work well in cases where we see many matching characters followed by a mismatching character.

Eg:

Text = A A A A A A A B

pattern = A A A A B.

When the naive approach is applied then, the inner for loop keeps looping till the last to encounter mismatch. Solving a pattern matching problem in linear time was a challenge. Hence, KMP algorithm is used which is known for linear time for exact matching.

Scanned by CamScanner

It compares from left to right. Shifts more than one position. **It** avoids recomputing matches by preprocessing approach of pattern to avoid trivial comparisions.

Case 1: When all the patterns to be matched has all unique characters.

<u>Eg:</u>

0 1 2 3 4 5 6 7 8
C O D E E C O D Y
↓ ↓ ↓ ↓
C O D Y

There is a mismatch at 3rd index in word. As all the characters in the pattern are unique, we can shift the pattern by 3 instead of 1.

C O D E E C O D Y
↓
C O D Y

When the pattern to be matched has unique characters,

* Since all letters in pattern are different, pattern can be shifted to the index where mismatch occurred.

Scanned by CamScanner

moodbanao.net

* As the characters are unique, there won't be any match in between.

case-2:

When all the pattern or parts of pattern have common suffix and prefix.

For a given string, a proper prefix is prefix with whole string not allowed.

For-eg:- "ABC"

Prefixes are: "", "A", "AB", "ABC".

Proper Prefixes are: "", "A", "AB".

Suffixes are: "", "C", "BC", "ABC".

consider the example with common prefixes and suffixes.

Text = C O D C O C O D C O

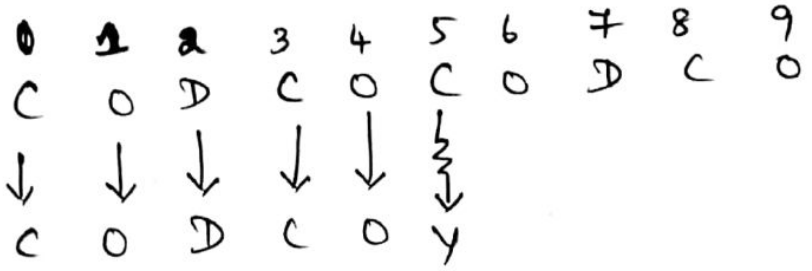
pattern = C O D C O Y.

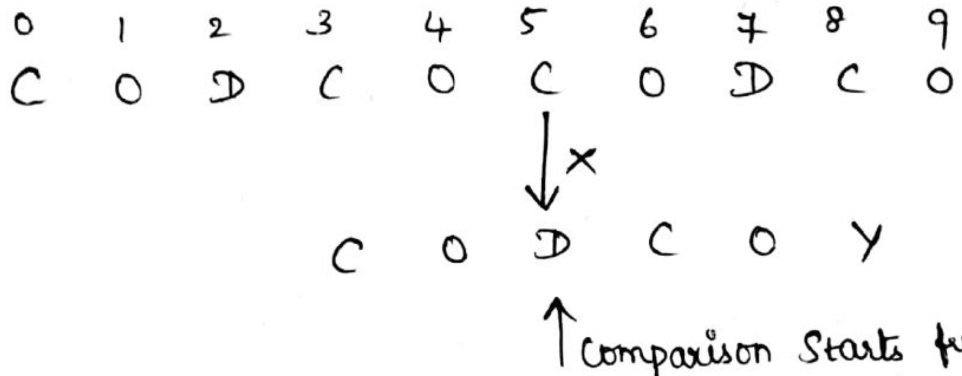
Here, "CO" is the common prefix and suffix of the substring of the pattern till index 4.

Since "CO" is already matched in the current window, it need not be matched again in the next window change.

Scanned by CamScanner

So, instead of shifting the window by 1, we shift by 3.





Comparison starts from here.

So, finding the common prefix and suffix ~~Skit~~ for each and every substring of the pattern will help to skip the unnecessary matches and increase the efficiency of the algorithm.

The LPS[] calculation of the pattern,

Pattern ⇒ C O D C O Y

LPS ⇒ 0 0 0 1 2 0

For, "CODC", "C" is both prefix and suffix. Hence, 1

For, "CODCO", "CO" is both prefix and suffix. Hence, 2.

Scanned by CamScanner

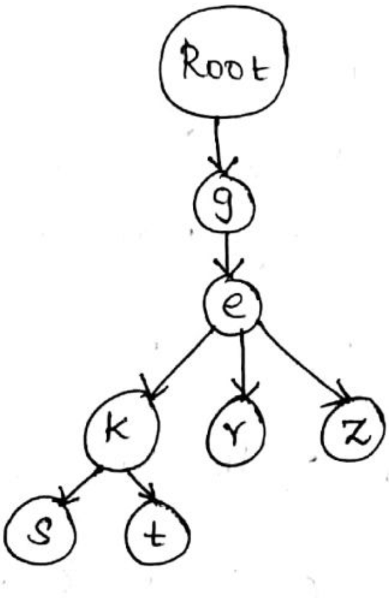
moodbanao.net

Trie :-

A tree like data-Structure wherein the nodes of the tree Store the entire alphabet and strings/words can be retrieved by traversing down a branch path of the tree.

It is an efficient information retrieval data Structure. Search complexity can be brought to optimal limit (key length). Thus, it is O(M) time, where M is the maximum String length.

Eg:



Every node has multiple branches. Each branch represents a possible character of Keys. The last node of every key should be marked as end of word node.

Scanned by CamScanner

Implementation:

If two strings have a common prefix of length n then these two will have a common path in the trie tree till the length n.

```
typedef struct trie_node
{
  bool NotLeaf;
  trie_node *pchildren[NR];
  Var_type word[20];
} node;
```

where, #define NR 27 (26 letters + blank)

Var_type → char (Set of characters).

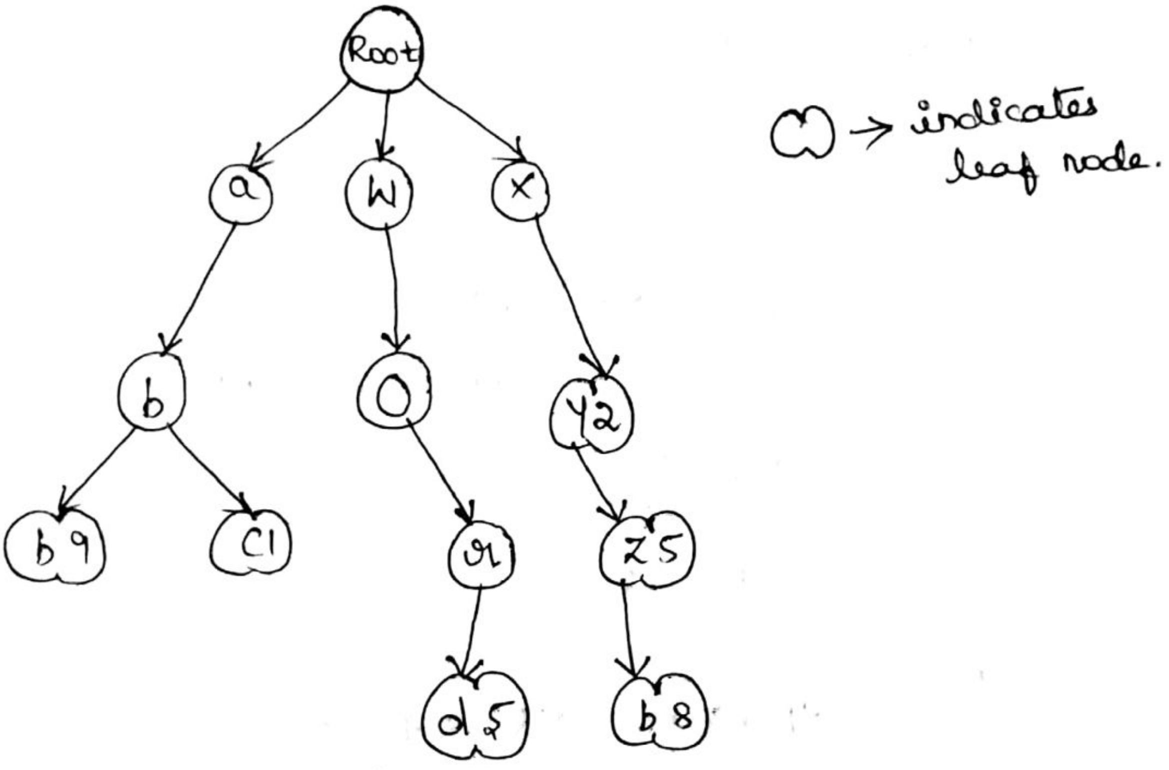
Operations:

- Insertion
- Deletion
- Searching
- Traversing.

Scanned by CamScanner

Trie data structure for key, value pairs,

("abc", 1), ("xy", 2), ("xyz", 5), ("abb", 9), ("xyzb", 8), ("word"), 5).



Insertion:- Insert ("abb", 9) into the trie.

1. Go to root node.
2. Get the index of first character of "abb" i.e., 0 in our alphabet system.
3. Go to 0th child of node root, which is not null. Mark 'a' as current node
4. Get the index of 'b' which would be 1. Go to first child of current node "a".
5. 1st child of "a" is not null. Hence, node 'b' as current node.
6. Get index of last character 'b' which is again 1. Now, create a new trienode at 1st child of 'b' which is null.
7. once read, mark current node as leaf and store the value 9 in it

Scanned by CamScanner

Every character of the input key is inserted as an individual trie node. Note that the children is an array of pointers to next level trie nodes.

If the input key is new or an extension of existing key, we need to construct non-existing nodes of key and mark end of word for the last node. If the input key is a prefix of the existing key, simply mark the last node of the key as the end of the word. The key length determines the trie depth.

Searching is similar to insert operation, where we only compare the characters and move down. The search can terminate due to end of string or lack of key in the trie.

Some applications :

1. Spell checking
2. Data compression
3. Storing/Querying XML documents etc.,

Scanned by CamScanner

moodbanao.net

Deletion:

1. If key 'k' is not found, then should not modify trie.
2. If key 'k' is not prefix or suffix of any other key and nodes of key 'k' are not part of any other key then all nodes from root to leaf of key 'k' should be deleted. Eg: delete key - "word".

3. If key 'k' is a prefix of some other key, then leaf node corresponding to key 'k' should be marked as 'not a leaf node'. No node should be deleted in this case. Eg: delete key - "xyz".

4. If key 'k' is a suffix of some other key 'k₁' then all nodes of key 'k' which are not part of key 'k₁' should be deleted. Eg: delete key - "xyzb". Only node "b" is deleted here.

5. If key 'k' is not a prefix or suffix of any other key but some nodes of 'k' are shared with other some other keys, then the nodes which are not common to any other keys should be deleted.
   Eg: delete key - "abc". Here only 'c' should be deleted.

Scanned by CamScanner

moodbanao.net

Standard trie uses O(n) space and O(dm) time to find, insert and delete.

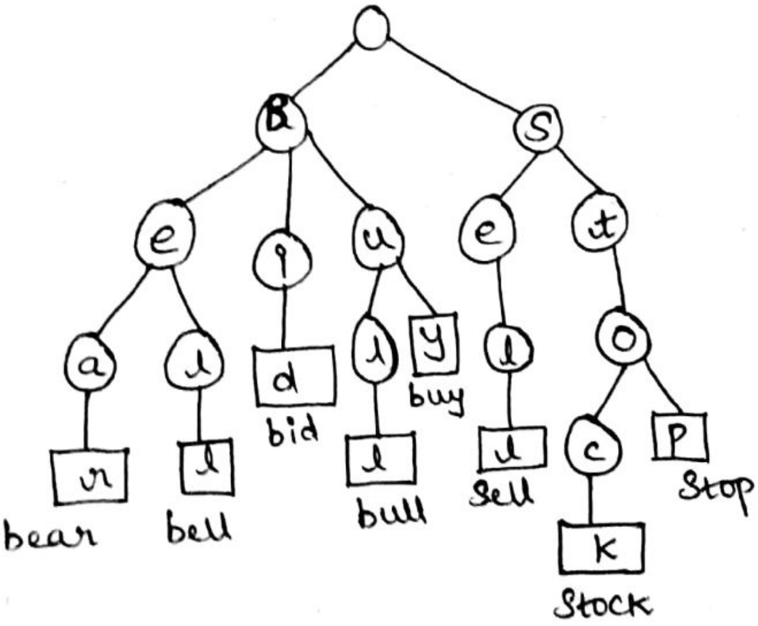
d→ alphabet size, m→ size of string parameter.

Compressed Tries:

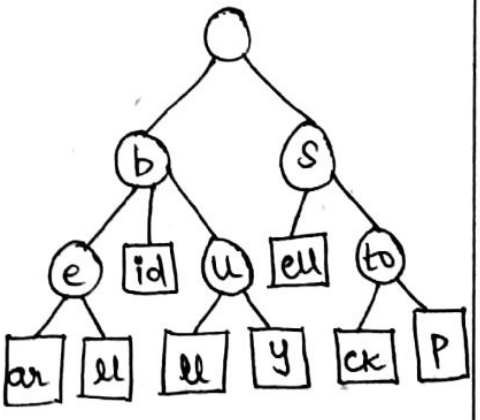
To overcome the disadvantage of Standard trie - space requirement, compressed tries are formed by compressing the chains of redundant nodes in Standard tries.

Eg.

Standard trie



Compressed trie



Scanned by CamScanner

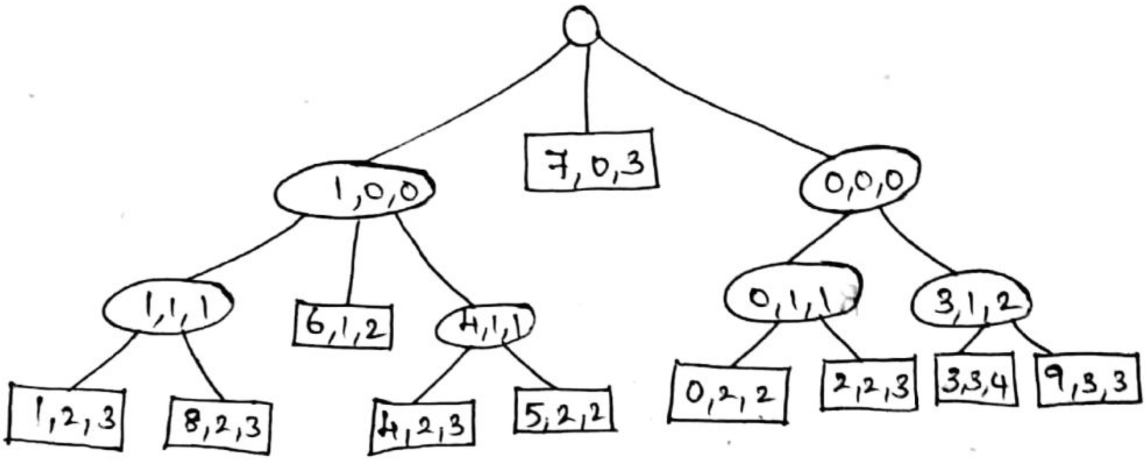
Compact representation of a compressed trie for an array of strings.

- * Uses O(S) space, where S is the number of strings in the array.
- * Serves as an auxiliary index structure.

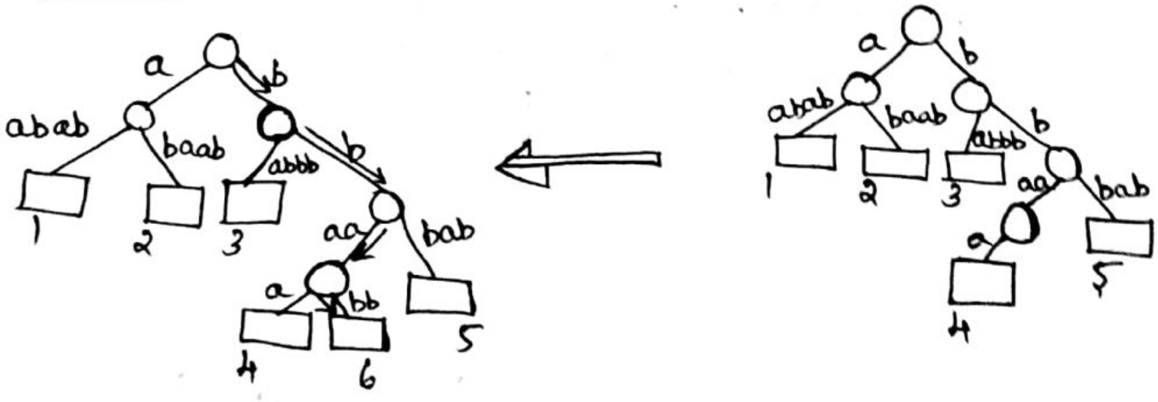
0 1 2 3 4
S[0] = s e e
S[1] = b e a r
S[2] = s e l l
S[3] = s t o c k

0 1 2 3
S[4] = b u l l
S[5] = b u y
S[6] = b i d

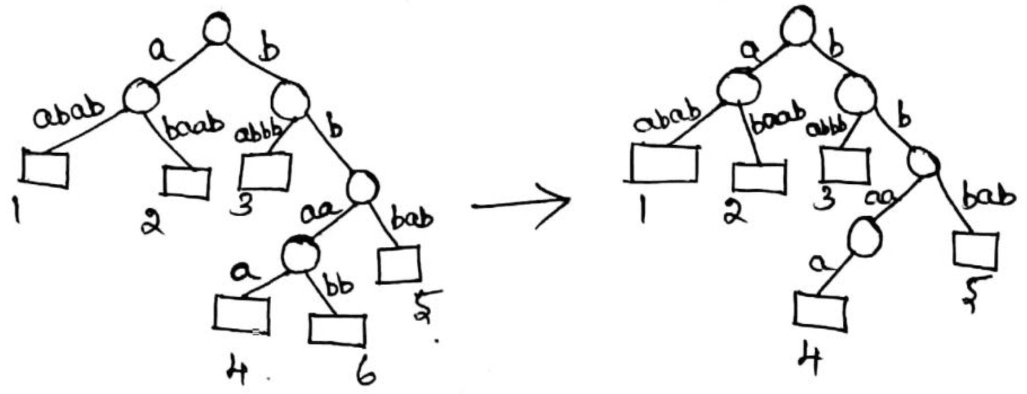
0 1 2 3
S[7] = h e a r
S[8] = b e l l
S[9] = s t o p



Insert (bbaabb)



Delete (bbaabb) :



Suffix Trees :-

The suffix trie of a String x is the compressed trie of all the suffixes of x. It helps in solving a lot of String related Problems like pattern matching, finding distinct Substrings in a given String, finding longest palindrome etc.,.

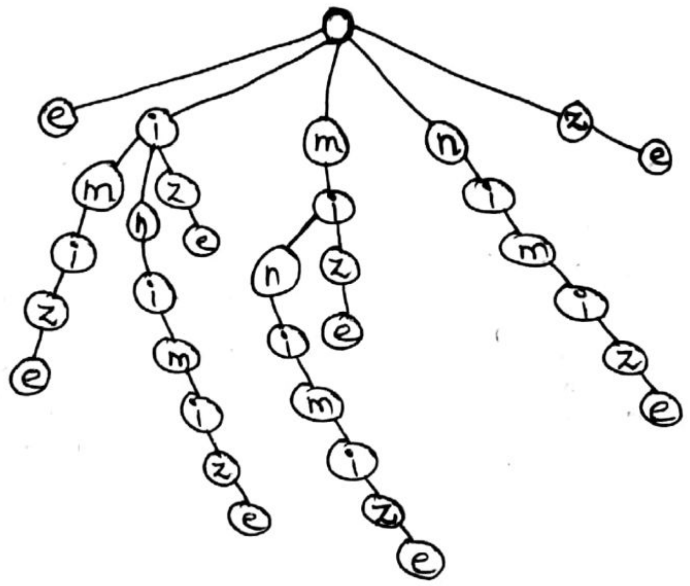
Eg:

minimize

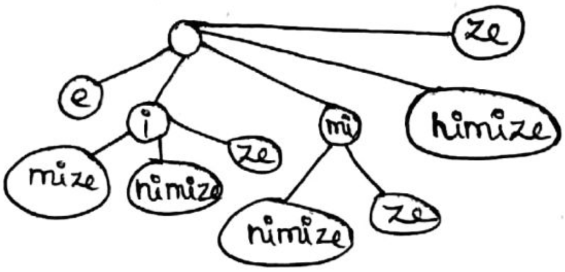
Identify all suffixes,

$$\left.\begin{array}{r} e \\ ze \\ ize \\ mize \\ imize \\ nimize \\ inimize \\ minimize \end{array}\right] \&$$

moodbanao.net

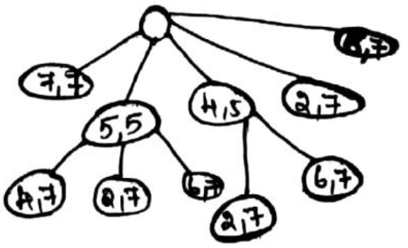


↓ compressed tree



Represent the compressed tree using numbers.

0	1	2	3	4	5	6	7
m	i	n	i	m	i	z	e



Hence, the suffix trie representation of a given word is done.

Scanned by CamScanner