UNIT - VII

PATTERN MATCHING & TRIES

Pattern matching :—

Pattern matching is the act of checking a given sequence of tokens (alphabets or symbols etc) for the presence of constituents of some patterns.

Means., pattern matching is to find a pattern which is relatively small, in a text which is to be very large.

- Pattern & text can be one-dimensional or two-dimensional.

1-dimensional example → text editor & DNA.

Text editor :- have a 26 characters & some special symbols.

DNA :- DNA has 4 characters.

2-dimensional eg.,- computer vision.

- Either 1-dimensional (or) 2-dimensional the text is very large & therefore., a fast algorithm to find the occurance of pattern in it needed.
- In the classic string pattern matching problem we are given text string 'T' of length 'n' and pattern string 'P' of length 'm'.

- To find whether 'P' is a substring of 'T'. The notation of a match is that there is substring of 'T' starting at some index 'i'. That matches, 'P' character by character so that such as.,

  P[0] = T[i], P[1] = T[i+1]; P[2] = T[i+2] ....

  .... P[m-1] = T[i+m-1].

- The output of a pattern matching alg., could put either be some indication. That the pattern P does not exist in 'T' or an integer indicating the starting index in 'T' of a substring matching 'P'.

Applications of pattern matching:—

* Text editor :- In text editor we have lno. of lines of text data for finding required string from the editor we use string pattern match.
* Search engine :- The query submitted by the user in search engine uses the pattern matching.
* Biological search :- Eg: DNA
* Pattern matching is used to find the common ~~personalites~~ functi of different research things.

Pattern matching algorithm:—

The popular pattern matching alg,'s are:-

- Naive pattern matching algorithm.
- Brute force algorithm.
- Boyer_moore algorithm.
- Knuth_morris pratt algorithm [KMP].

· Boyer-moore algorithm:—

Introduction: It is a pattern matching algorithm. It was developed by Robert S Boyer and J S Moore in 1977.

Definition: The boyer-moore algorithm is an sufficient string search alg,. This alg, preprocess the string being searched for the pattern but not the string being searched in the text.

It is well suitable for applications in which pattern is much shorter than the text, does persist (carry on) across multiple searches.

Purpose of Boyer-moore alg,:

It is usually used in text editor & commands substitution.

Text editor :— In text editor for search & substitute commands implementation we use this alg,.

Working Procedure :—

This alg., compare's the pattern P with the sub string of sequence T with in a sliding window in the right to left order.

Means it can scan the characters of pattern from right to left beginning with the right most one.

This alg., uses two heuristics (set of rules)

(1) Bad character rule. (BCR).

(2) Good Suffix rule

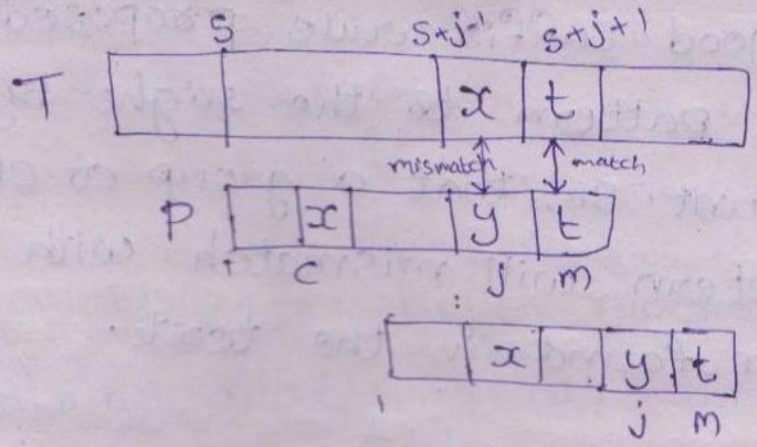
1) BCR :—

Suppose the P_1 is aligned to T_s. Now we perform a pair wise comparing b/w text 'T' & pattern 'P' from right to left.

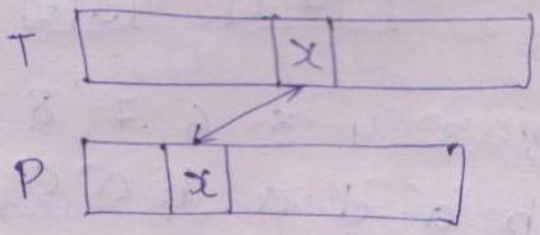
Assume that the first mis-match occurs when comparing T_{s+j-1} with P_j.

Since, $T_{s+j-1} \neq P_j$, we moves the pattern P to the right. Such that the largest position 'C' in the left of P_j is equal to T_{s+j-1}.

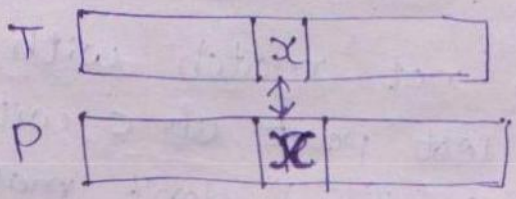
We shift the pattern atleast j-C positions.



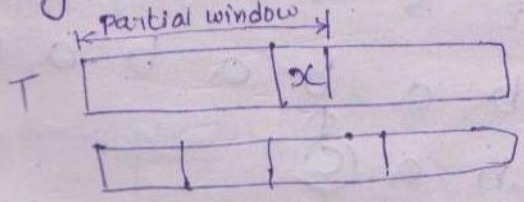
* BCR uses rule 2-1. [character matching rule].
* For any character x in T find the nearest x in P which is to be left of x in T.



Move 'P' so that the two x's are match.



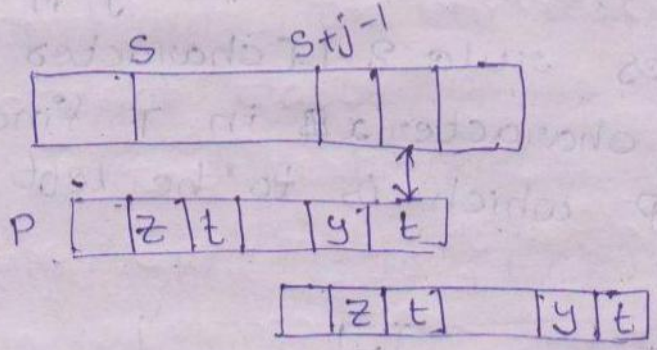
If no such a 'x' in P. consider., the partial window debined by x. In T & the string to the left of it.



2) Good Suffix rule :—

Good suffix rule used when BCR failed.

The good suffix rule proposed to move the pattern to the right by the least amount so that a group of characters in the pattern will mismatch with the good suffix found in the text.



Eg:-1

	0	1	2	3	4	5	6	7	8	9	10
T →	a	b	b	a	d	a	b	a	c	b	a
P →	b	a	b	a	c						

b a b a c

Here 'c' does not match with 'd' then compare the left part of c with left part with d if 'd' don't match with left part of c then shift the 'P' to right part of 'c'.

Eg:-2

	0	1	2	3	4	5	6	7	8	9	10
T →	a	b	b	a	b	a	b	a	c	b	a
P →	b	a	b	a	c						

b a b a c ~~(match)~~.

b a b a c (match)

case-3:

	0	1	2	3	4	5	6	7	8	9	10
Eg: T→	a	b	a	a	b	a	b	a	c	b	a
P→	c	a	b	a	b						
			c	a	b	a	b				

Time analysis for boyer-moore algorithm:-

A string matching algorithm pre-process a pattern P [|P| = n] → size of pattern
For a text T [|T| = m] → no of characters in text.
Find all of the occurance of P in T.

Time complexity is $O(n+m)$

Right to left pattern matching.
Worst case complexity is $O(n \cdot m)$
Best case complexity is $O(n/m)$
(Time complexity is calculated by based on the no. of lines executed).

Knuth-Morris Pratt alg, (KMP):—

Intro: KMP is a pattern matching algorithm.
It was conceived by Donald Knuth,
Vaughan Prat in 1974. and
→ independently J. H. Morris.
→ The three published jointly in 1977.

Definition:—
KMP alg, searches for occurance a word 'w' within a main text 's' by employing (applying) the observation. That when a mismatch occurs the word itself embodies (represent) sufficient information to determine where

the next match could begin, that by passing re-examination of previously matched characters.

Working process:—

It is a tight analysis of naive algorithm. KMP algorithm keeps the information that naive approach wasted gathering during the scan of the text.

By avoiding the waste of information it acheives a running time of text is $O(m+n)$ (size pattern, size text)

Eg: W = ABCDABD

S = ABC ABCDAB ABCDABCDABDE.

At given time., the algorithm is in a stage determined by two integers.

1st: m → which denotes the position in text which is the begining of a perspective (future/potential) match for W.

2nd: i → The index in W denoting the character currently under consideration.

In each step we compare S[m+i] with W[i] advance if they are equal.

Eg: 1

M: 0 1 2 3 4 5 6 7 8 9 0 12 3 4 5 6 7 8 9 0 12

S: ABC ABCDAB ABCDABCDABDE

W: A B C D A B D
i: 0 1 2 3 4 5 6

S[3] = Space } mismatch.
W[3] = D }

We start at S[0] but it fail. Now we start from S[1]. But we note that no 'A' occurs b/w positions 0 & 3 expect at 0. having checked all those characters previously we know that there is no chance of finding the beginning of a match.

∴ We move onto the next character. M=4, i=0

II) M: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
S: A B C A B C D A B A B C D A B C D A B D E
W: A B C D A B D
i: 0 1 2 3 4 5 6.

S[10] = Space } mismatch.
W[6] = D }

III)
M: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
S: A B C A B C D A B A B C D A B C D A B D E
W: A B C D A B D
i: 0 1 2 3 4 5 6.

S[10] = space ? } mismatch.
W[2] = B C }

IV) M: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
S: A B C A A B C D A B A B C D A B C D A B D E
W: A B C D A B D
i: 0 1 2 3 4 5 6

S[17] = C
W[6] = D } mismatch.

v)

M: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
S: A B C A B C D A B A B C D A B C D A B D G
W: A B C D A B D
i: 0 1 2 3 4 5 6

This time we are able to complete the match
· Choose 1st character is S[15].

Algorithm:-

```
Alg KMP-Search ( S[], w[] )
{
// I/P :- S → array of characters (The text.
                                   to be searched)
       w → array of characters.
           (The text to be sought (looking for)

// O/P :- an integer.
       The zero placed position in S
       at which w is found.
Integer: M←0 // The begining of the current
                match in S.
Integer: i←0 // The position of the current
                character in w.
Array integer : T // the table compute else where.
while ( m+i < length(S) do
 if W[i] = S[m+i] then
    if i = length(w) - 1 then
```

return m
i ← i+1
else
m ← m+i-T[i]
if T[i] > -1 then
i ← T[i]
else
i ← 0
end while.

If above alg., return m value ∴ w found in S at position m. If above alg., did not return m. we say we haved searched all of 'S' unsuccessful.

KMP - time analysis :-

For finding 'P' of size 'n' (|P| = n).
In text T of size 'm' (|T| = m).
By using naive alg., time complexity → O(n.m)

The KMP makes use of information gained by previous symbol comparisions.

It never recompares a text symbol that has matched a pattern symbol. Time complexity for text is O(m) & pattern is O(n).

Overall time complexity is O(m+n)

Trie :-

Trie is an efficient information retrieval ordered tree data structure.

- It is also called multi way tree d.s

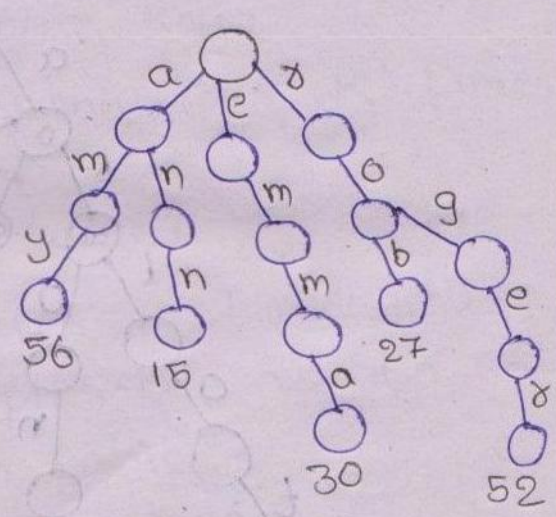
[Multiway tree: It is a tree d.s. In this if the root node contains 'x' elements then the subtrees are (x+1)]

- The name trie comes from its use of retrieval.
- We use trie to store piece of data that have a key K (which is used to identify a data) & possibly a value.

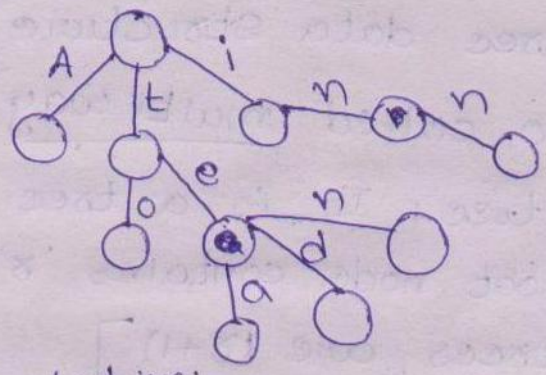
In trie we use data whoose keys are strings.

Ex:- Name & age for set of people.

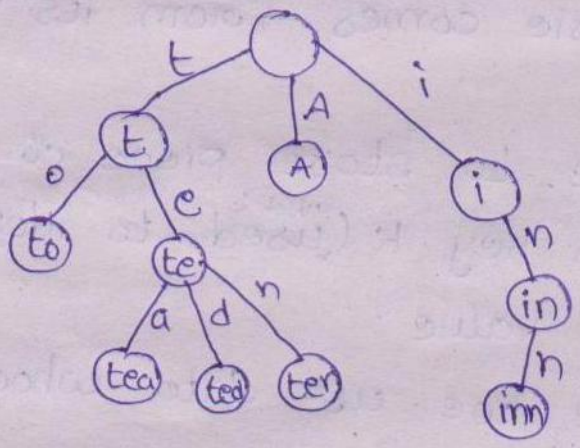
Name	age
amy	56
ann	15
emma	30
rob	27
roger	52



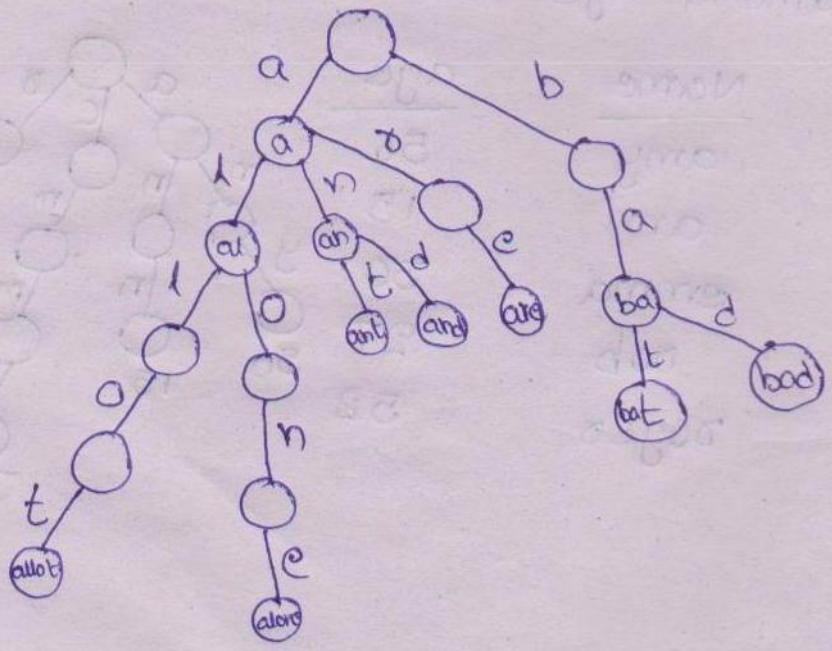
Ex:- A trie for keys "A" "to" "tea" "ted" "i" "ten" "in" "inn"



Another representation:-



Ex:- "allot", alone, ant, and, are, bat, bad.



Unlike., a binary search tree, no node in the tree stores the key associated with that node. Instead., its position in the tree defines the key with whict it is associated.

- All the descendats of a node have a common prefix of the string associated with that node.
- Root is associated with empty string.
- Values are not associated with every node.
- Only with leaf nodes & some inner nodes that corresponds to keys of insert.

Advantages :—

* The pattern matching can be done efficiently.
* In tries, the keys are searched using common prefix.
  - ↳ It takes O(K) lookup (searching) time. where k is the size of a key.
* Lookup can takes less than (<) 'K' times if it is not there.
* Comparing with hash table: —
  - Look up can be faster in time ., in worst case time complexity as to compared with hash table.
  - There is no collision in trie.
  - There is no hash function in trie.
* comparing with BST : —

Disadvantages:

* Some times data retrieval of trie is very much slower than hash table.
* Representation of keys a string is complex
Eg:- Representation of floating point numbers using string is really complicated in tries.
* It always takes more space.
* It is not available in programming tool.

Applications :—

- Tries has an ability to insert, delete, or search for the entries. Hence., there are used in building dictionaries. Such as., English words, telephone numbers.
- These are also used in spelling check s/w
- These are well suited for approximate matching algorithm.

Digital Search tree (DST) :—

- A DST is a binary tree in which each node contains one element.
- The element to node assignment is

determined by the binary representation of element key.

- DST represents one possible data structure which allow us to store, search, & delete data using the key.
- DST is similar to BST but, the main difference is instead of comparing key values, they make use of the digital representation of the keys.
- If the key can be represented as binary number it makes ~~seen~~ sense to refer the b^{th} bit of a key, where the bits are numbered from left to right.
- To insert a record (K, data) with key K into a DST, we ~~said~~ set 'x' to point to root and "b" to 1.

Steps for DST:—

*Step-1: If x is nill, then store (K, data) in new node and terminate.

Step-2: If Key(x) = K, then terminate (K is already stored in the tree).

Step-3: Otherwise,

↳ If b^{th} bit of K is 0 then set x to left(x).

↳ If b^{th} bit of K is 1 then set x to right(x).

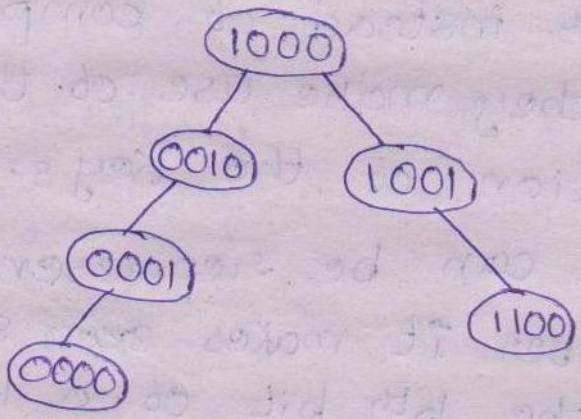
Step-4: Set b to (b+1)

Step-5: Go to Step-1

Example :—

Create a DST by using keys,

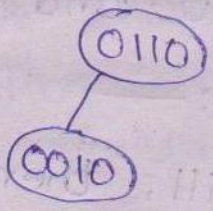
1000, 0010, 1001, 0001, 1100, 0000



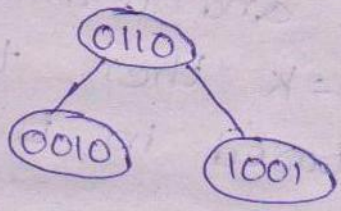
Ex-2)

insert 0110

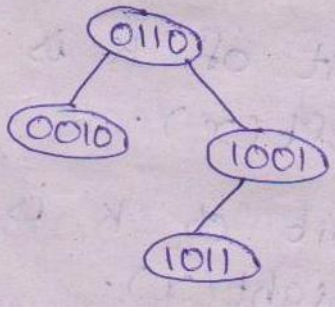
insert 0010



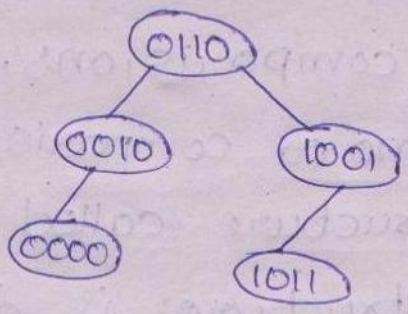
insert 1001



insert 1011



insert 0000

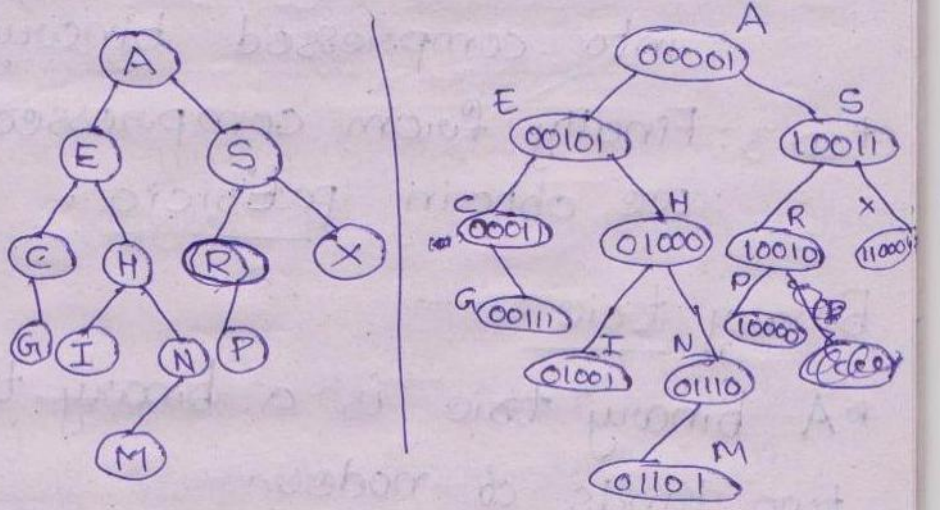


Eg-3) create DST.,

A-00001 , S-10011 , E-00101 , R-10010

C-00011 , H-01000 , I-01001 , N-01110

G-00111 , X-11000 , M-01101 , P-10000



Advantages of DST :—

In DST we compare keys bitwise, by this we avoid full length of key which is not suitable for searched element key. By this we eliminate time for searching full length of key.

Drawbacks of DST :—

* When we dealing with very large key, the cost of a key comparision is very high.

* To Solving this problem, we have to reduce no.of comparisions.
* For reducing no.of comparisions we using another data structure called "Patricia".
* This patricia structure is developed in three steps.

Step-1: First we introduce a structure called binary trie.

Step-2: Then we transform binary-trie into compressed binary-tries.

Step-3: Finally from compressed binary tries we obtain patricia.

Binary trie :—

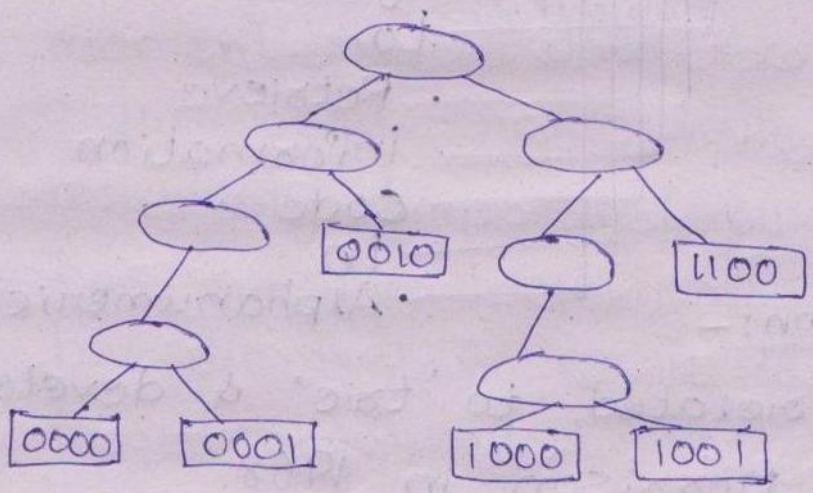
* A binary trie is a binary tree that has two kinds of nodes.
  (i) Branch Node
  (ii) Element Node.

(i) Branch Node: Branch node has two data members. * Left child
* Right child

(ii) Element Node:
Element node has single data member called Data.

+ Branch nodes are used to build a binary tree search structure. Similary to the DST This leads to element nodes.

Ex, ~~box~~ binary trie:



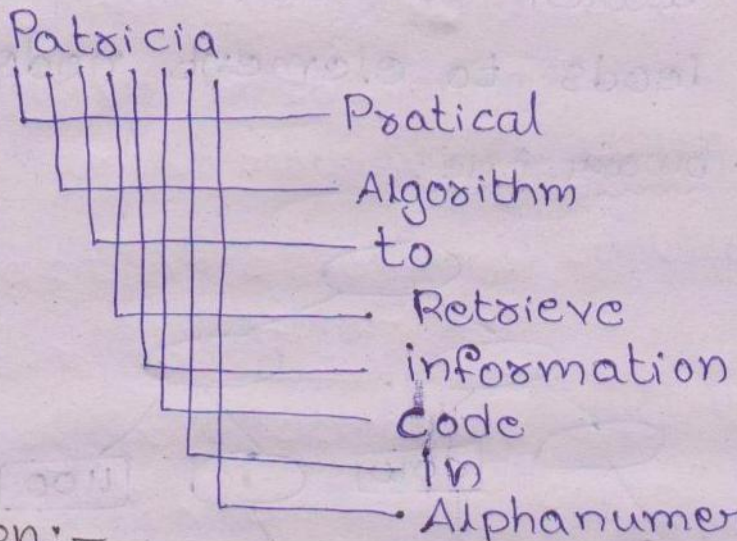
Compressed binary trie:-

*The binary trie's contains branch whoose degree is one.

*By adding another data member bit-number, to each branch node We can eleminate all degree one branch node from the trie

* The bit-~~member~~ num member data ~~no~~. of a branch nodes gives the bit-number of the key. i.e., to be used at this node.

Patricia :—



Introduction :—

It is related to 'trie' & developed by D.R. Morrison in 1968

Purpose :—

- This is for solving the comparision problem in DST.
- It is also called radix tree.
  [Radix / base : is the no. of unique digits including zero].

Definition :—

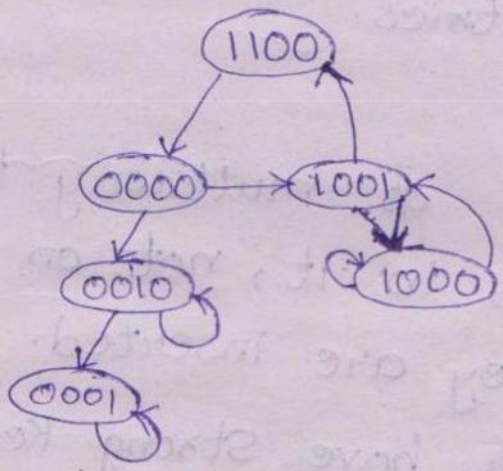
Patricia is a space optimized trie d.s. where each node with only one child is merged with its parent.
This obtained from compressed binary trie in the following way.,

Step-1: Replace each branch node by a argumented branch node.

Step-2: Eleminate the element nodes.

Step-3: Store the data previously in the element node in the data members of argumented branch node since every non-empty compressed binary tries has one less branch node then it has element nodes. It is necessary to add 1 argument branch node. This node is called head node.

Step-4: Replace the original pointers to element node by pointers to the respective argumented branch nodes



Multiway tries :—

Multiway tries is a one of the trie d.s for fast data retrieving.

. It is for over come's the drawbacks for binary trie.

* Binary trie uses radix search with radix 2
* Coming to multiway tries, coc uses radix search with $R > 2$
* Multiway tries sometimes called as R-ary tries.
* In each digit in a key has 'r'-bits., then the radix is $R = 2^r$.
* If the key have atmost B-bits, the worst case time for the no.of comparisions would be "B/r".

Ex-1) Keys are words made up of lower case letters in english. There are 26 different lower case letters in english. So, R-ary tries with $R = 26$ could hold these keys. This type of tries sometimes refferes as alphabet tries.

Properties :——

* The structure of multiway trie depends only on keys in it, not on the order in which they are inserted.
* Multiway-tries have strong key ordering property.
  - At a node 'x', all keys in 'x' left most sub tree are smaller than keys in 'x'.
  - At a node x, all keys in x right most

Sub tree are larger than keys in x.

- So, tree order traversal of a multiway trie visits keys in ~~sorted~~ sorted order.

* The worst case time for the no. of comparisions would be B/δ.

- ⊙ worst case height in BST contains N
- ⊙ worst case height in DST contains $\log_R N$

Drawbacks :—

There is a space cost disadvantage in multiway trie. So, To solve
To solve this problem we go for another data stoucture ternary trie.