UNIT - 4

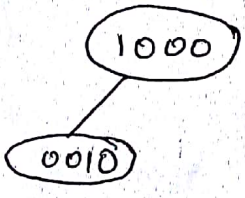
Digital Search Trees:

* A digital Search Tree is a binary tree in which each node contains one element.
* The element to node assignment is determined by the binary representations of the element keys.

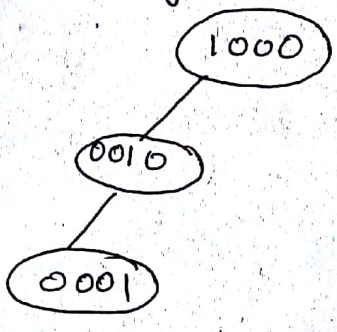
eg: keys are
1000, 0010, 0001, 1100 and 0000

first node will be the root node.

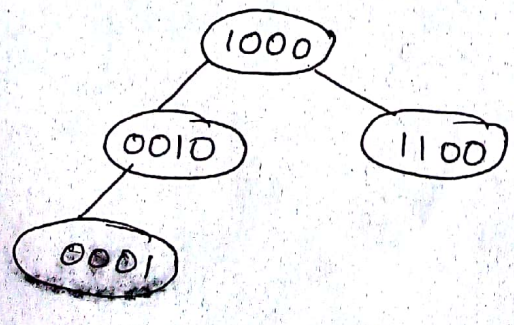
Second key = 0010
↓
node starting bit is zero, so left insertion.



Third key = 0001 (Left)

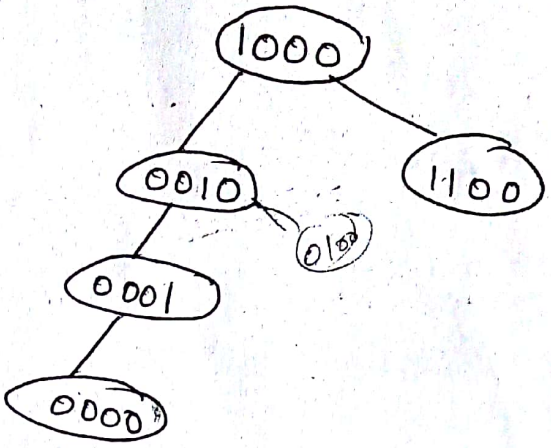


Fourth key = 1100
↓
first bit is one, so right insertion



Scanned by CamScanner

final fifth key = 0000



* Operations Involved in Digital Search Trees:

There are three operations involved in digital Search trees.

1. Search
2. Insert
3. Delete

Search:

* Suppose we are to Search for the key = k = 0000 in the tree

Step 1: * k is first compared with the key in the root. and Since bit of one of k is "0", we move to the left child. (first bit)

Step 2: Now Consider 2nd digit of key k = 0000 (zero) 2nd digit is zero so again move to left.

Step 3: Now consider 3rd digit of key = 0000

Third digit is zero, again Left move.

Scanned by CamScanner

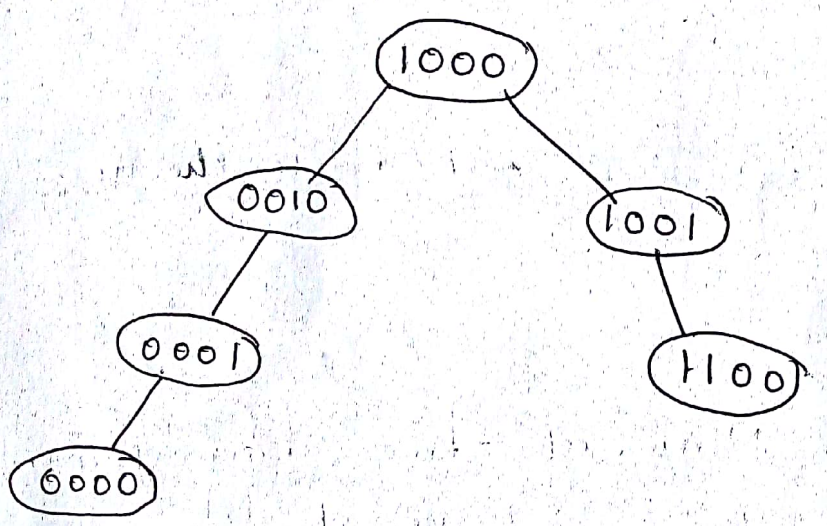
* Step 4: Now consider 4th digit of key k = 0000.
So, 4th digit is zero, so Left move.
There is Left node, so search ends at that node.

* Now compare node data with key k = 0000. if it is same Search is Successful, if element not found i,e an unsuccessful Search.

Insert:-

Before insertion, we should check whether the key is already presented in digital Search tree not.

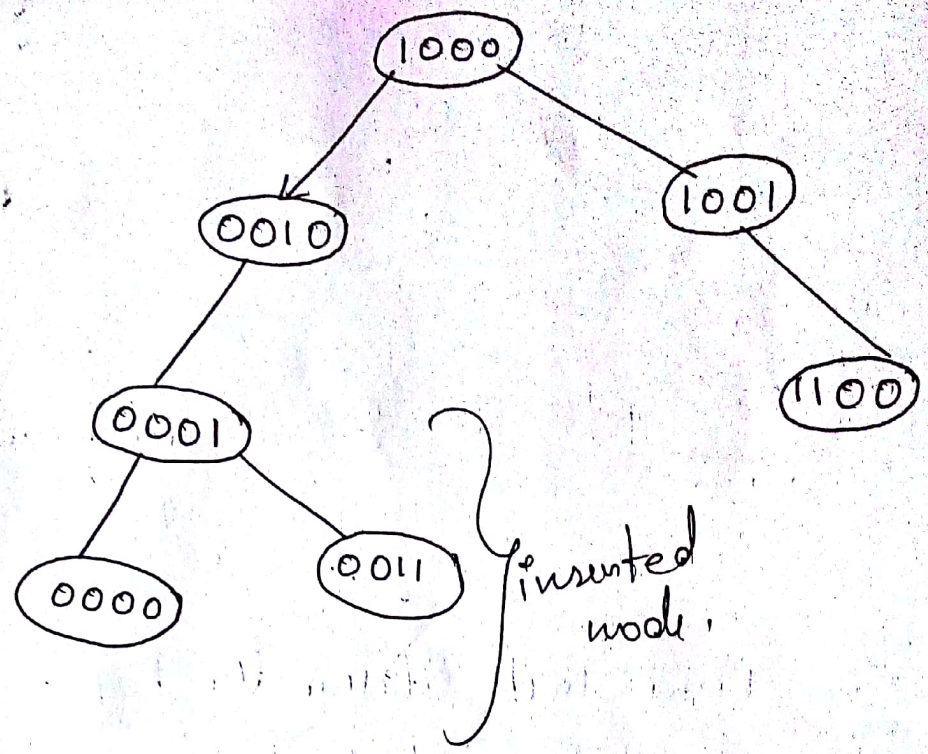
Given digital Search Tree.



* Insert key = 0011, before insertion, we need to check whether "0011" presented in Tree (or) not.

*
Step 1: first digit of the key is "zero", (Left move)
Step 2: Second digit of the key is "zero" (Left Move).
Step 3: Third digit of the key is "one", (right Move).
Now insert element (0011).

Scanned by CamScanner

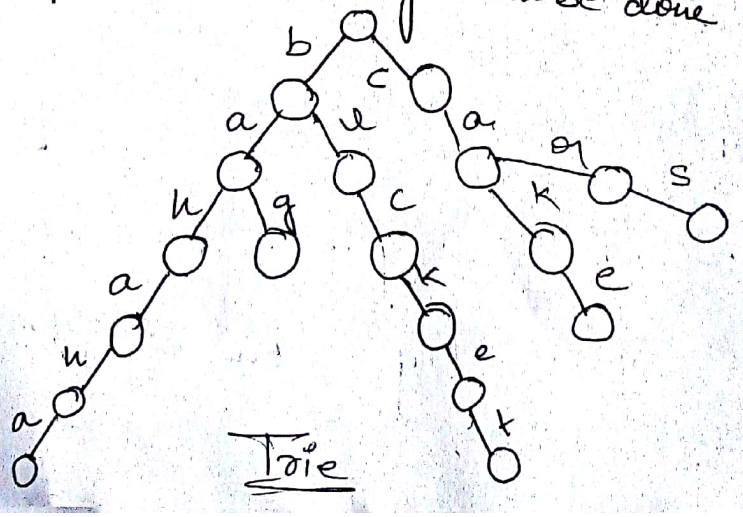


Delete:-

* In order to perform deletion operation, Simply Search the given node.
* If the node is present simply delete the node from the tree. (cases).

Trie:-

* A trie is a Multiway tree data structure used for storing strings over an alphabet.
* It is used to store a large amount of strings.
* The pattern Matching can be done effectively using tries



Scanned by CamScanner

- preprocessing pattern improves the performance of patt matching algorithm. But if a text is very large then it better to preprocess text instead of pattern for efficeil s

* A trie is a data structure that supports pattern matching queries in time proportional to the pattern size

Advantages of Tries :-

1) In tries the keys are searched using common prefix Hence it is faster.

2) Tries take less space when they contain a large num of short strings. As nodes are shared between the k

3) Tries help with longest prefix matching, when we want to find the key.

Comparison of Tries with Hash Table:

* Looking updata in a trie is faster in worst cas compared to imperfect hash Table.
* There are no collisions of different keys in a trie.
* In trie if singe key is associated with more than value then it resembles buckets in hash table.
* ~~Buckets resembles~~
* There is no hash function in trie.
* Sometimes data retrieval from tries is very much slower than hashing.

Scanned by CamScanner

Binary Tries:

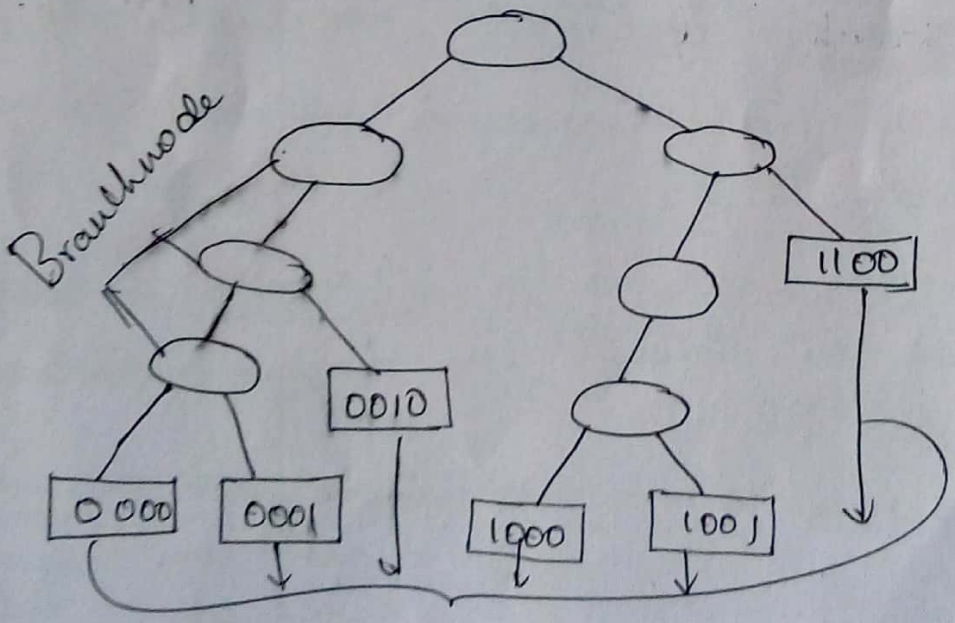
* Binary is Tarie is a binary Tree that has two kinds of nodes.
  1. branch nodes
  2. Element nodes.

* Branch node has the two data members → Left child, Right child.
  * Branch node has no data data member.
* Element node:

  Element node has the single data member "data"

* Branch nodes are used to build a binary tree search structure similar to that of a digital search tree.

Representation:



element Nodes.

* To search for an element with key k = 1000, we use branching pattern determined by the bits of k.

Scanned by CamScanner

* If the bit is zero, the search moves to the left subtree & otherwise it moves to the right subtree.

eg: To search a key K = 1000.,

1. We first follow right child, then left child, again Leftchild. and finally left child.

* The Successful search in a binary tree always ends at an element node.
* Once the Element node is reached, the key in the node is compared with the key we are searching
* If the data and key are same that leads to Successful Search.
* If the data and key are not same that leads to unsuccessful search.

Compressed Binary Trees:

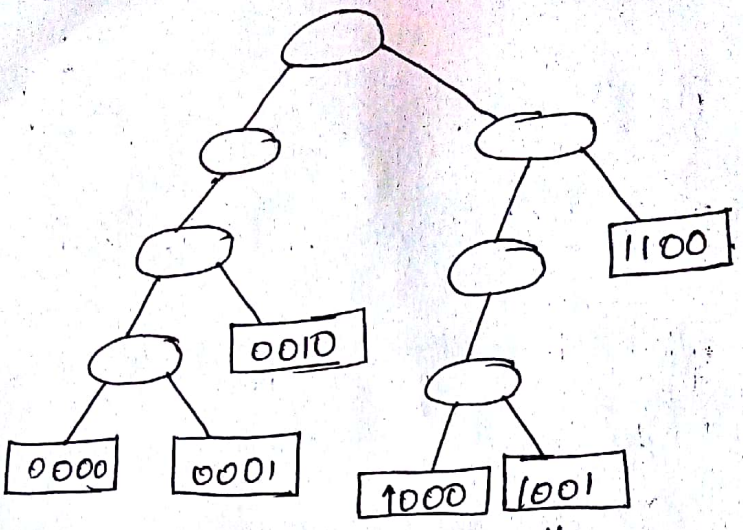
* First, add another data member, bit Number, to each branch node, we can eliminate all degree-one branch nodes from the tree.
* Bit Number (data member) of a branch node gives the bit number of the key that is to be used at this node.

Scanned by CamScanner

Patricia:-

* When we are working with very long keys. (In DST) the cost of a key comparison between pairs of keys,
* Digital search trees are inefficient search structures when the keys are very long.
* We can reduce number of key comparisons done during a search to one by using a related structure called Patricia.
* Patricia (practical algorithm to retrieve information coded in alphanumeric).
  (1) Introduce a structure called binary trie.
  (2) Transform binary tries into compressed tries.
  (3) from the compressed binary tries we obtain patricia.
* compressed Binary tries may be represented using nodes of a single type.
* The new nodes, called augumented branch nodes, are the original branch nodes augumented by the data member data.
* The resulting structure is called Patricia. and is obtained from a compressed binary trie.

Scanned by CamScanner

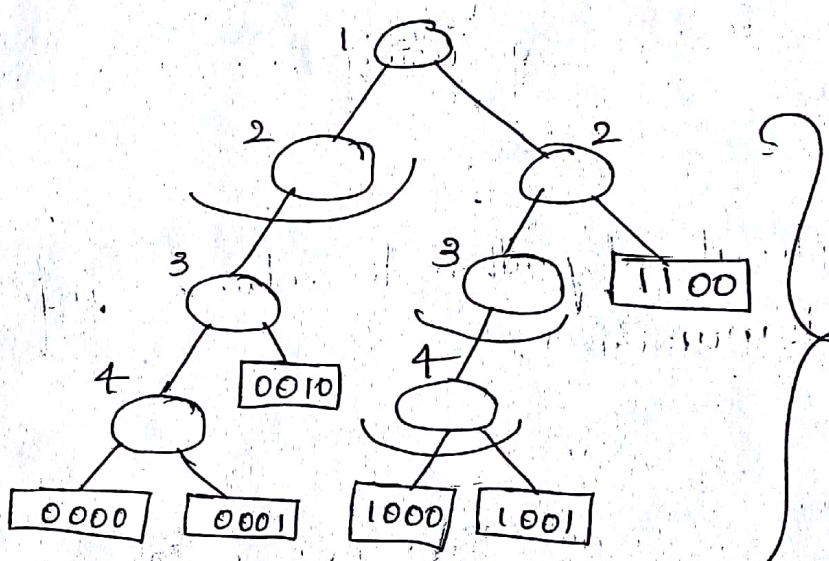


Given Binary Trie

⇓

Now add bit Numbers to each branch node

⇓

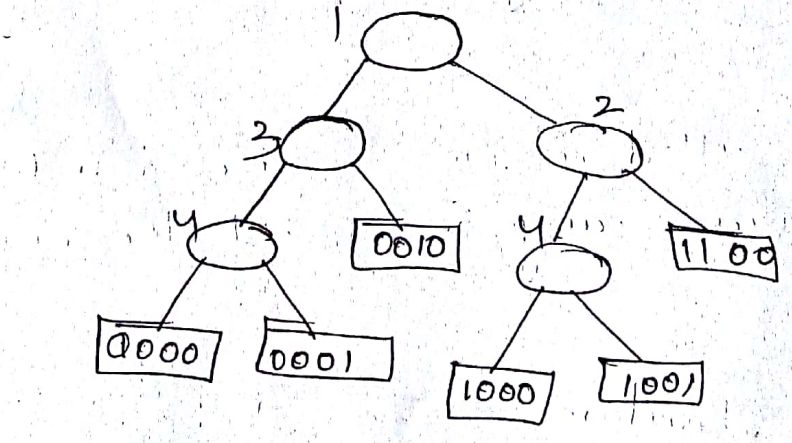


Now identify the branch nodes having Single Child.

Here, Left side (2)
Right side (3,4)

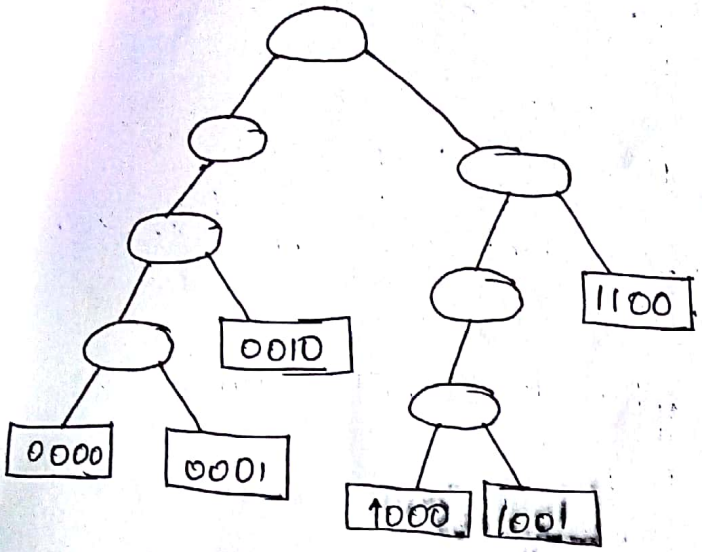
these are the nodes having Single child.

⇓



Compressed trie.

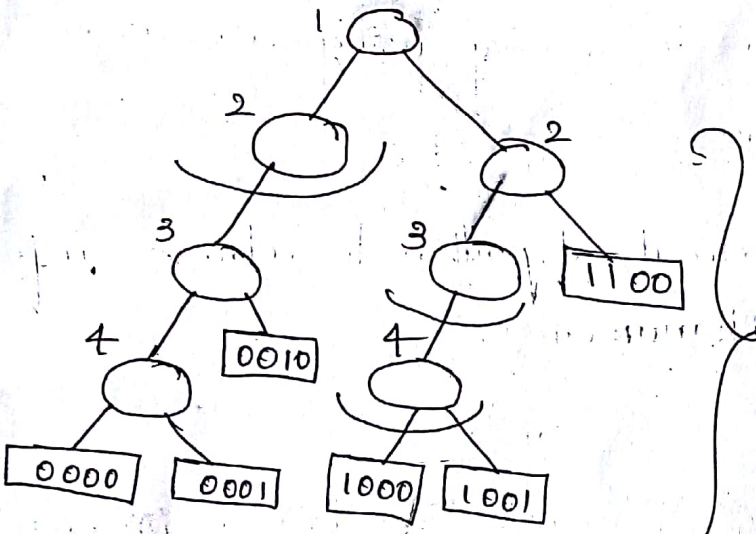
Scanned by CamScanner



Given Binary Trie.

⇓

Now add bit Numbers to each branch node.

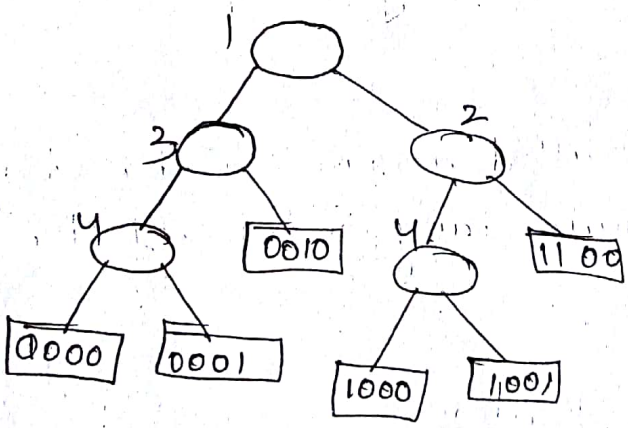


Now identify the branch nodes having Single child.

Here, Left side (2)
Right side (3,4)

these are the nodes having Single child.

⇓

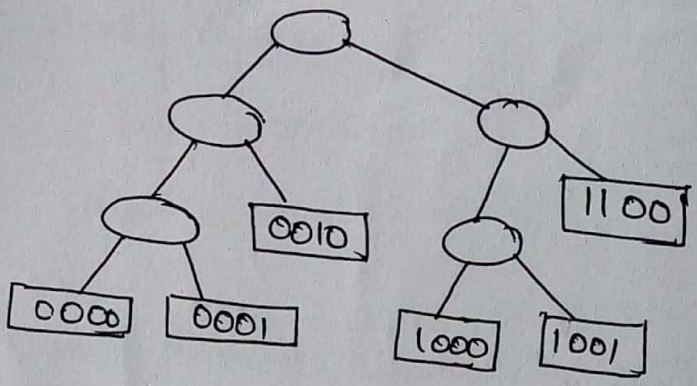


Compressed trie.

Scanned by CamScanner

es to generate patricia:

eg:



(i) Replace each branch node by an augumented branch node.

(ii) Eliminate the element nodes.

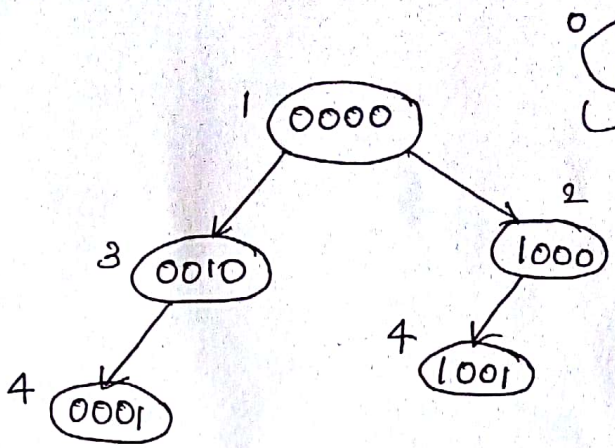
(iii) Store the data previously in the element nodes in the data datamember of the augumented branch nodes.

Here, number of branch nodes in the compressed binary trie are "5", no of element nodes in the compressed trie are "6". So, one branch nodes is less than the element nodes. (i.e so is necessary to add one augumented branch node. ;i.e header node. (The remaining structure is the left subtree of the header node.

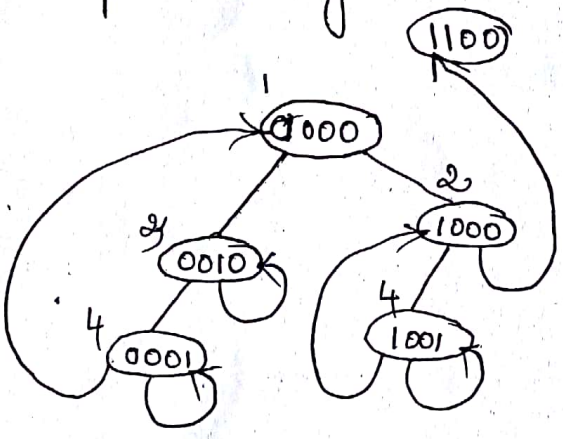
Header Node: Header node has bitNumber equal to zero. and its right child data member is not used.

** The Assignment of data to augumented branch nodes is done in such a way that the bitNumber of in the augumented branch nodes is less than (or) equal to that in the parent of the element node that contained this data.

Scanned by CamScanner



(iv) Replace the original pointer to element nodes by pointers to the respective augmented branch nodes.



Operations of patricia:

1. Searching :-

* To search for a key k = 0000.

1) Start from the header node, and follow the left child pointer to the node with 0000.

2) The bitNumber data member of this node is "1". so check the first digit of your key k. (k = 0), Now take left move.

3) Observe the bit number of the branch node. (Bit no = 3) Now check 3rd digit of your key k. (3rd digit key k = 0 Take left move.

4) Observe the bit number of the branch node (Bit no = 4) Now check 4th digit of your key k (4th digit key k = 0)

Scanned by CamScanner

w. 4th branch node does not contain any branch node
30 Search ends at 1st node. Now Compare 1st node data with key k.

Key k = 0000.

1st node data = 0000

0000 = 0000. (so Search found).

```
template <class K, class E>
E * patricia <K,E> :: Search (const K& k) const.
{
// Search partiia. Return a pointer to the element whose
   key is k.
// Return Null if no such element
if (!root) return Null; // partricia is empty
patNode <K,E> * y = root -> leftchild; // move to left child
                                       of header node
for (patNode <K,E> * p = root; y -> bitNumber > p -> bitNumber
{
// follow a branch pointer
p = y;
if (bit (k, y -> bit Number)) y = y -> right child;
else y = y -> leftchild;
}
// Check key in y
if (y -> key == k) return & y -> element;
return Null;
}
```

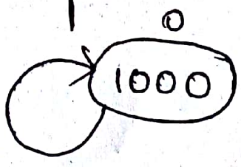

Scanned by CamScanner

Inserting into patricia:

* Before going into insertion, first check the key is already presented in patricia (or) not.

* "1000" (insert node) :

(1) This is the first node "i,e header node (∵ bit Number = 0).



[No right pointer for header Node.]

(2) Now insert key K = 0010.

* First, we search for the key using function.

patricia::search(),

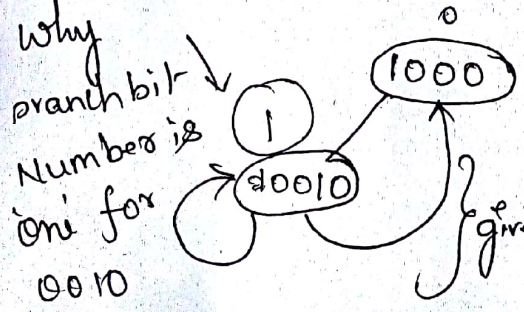
* start from the header node, header node contains left pointer.

* patricia contains only one element ∴ header (left ptr) (self ptr)

∴ Simply, search ends at header node; (q).

Now compare key K = 0010, with headernode.
headernode = 1000.

(∵ key "K" is not equal to "q", nothing but, 0010 not currently in patricia") . so element is inserted.



means → while comparing "q" and "k", at the first ... difference.

Scanned by CamScanner

first of bit of our key = 0, so the node added as Left-child of the header node.

Since, bit one of key (k = 0010) is zero, the Left child data member of this new node points to itself, and its Rights child data member points to header node.

(iii) Now insert node (Key K = 1001), (before Search the element)

1. Start from the header node,
2. Take Left Move (because, header node contains Left child only).
3. Observe the bit Number of the branch node. "1". One means Right Move.
4. So, Right ~~Nor~~ pointer of "1." branch node points to header node. [Whenever the pointer reaches to branch node having bit Number Less than the previous node bit Number " Search ends at that position
5. Here Search ends at header Node,

q = 1000. } Compare both.
Our key K = 1001 } at 4th position, if ~~go~~ difference occured.

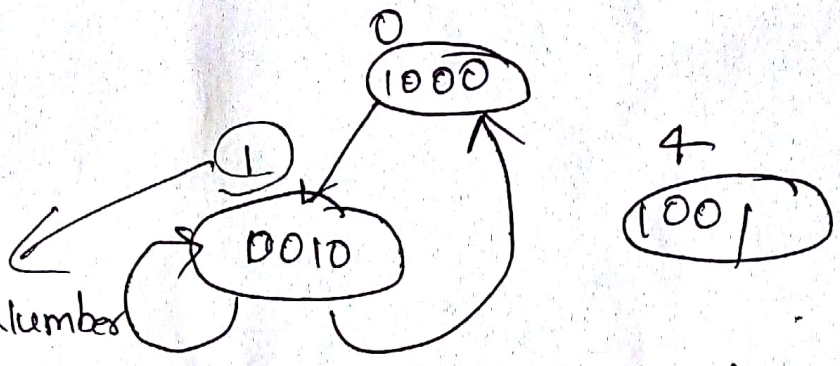
6. so, The newly created branch node bit Number is equal to "4".

(1001) 4. } new node created.

7. Now, check, the newly created node is left of "1" branch node (or) right of "1" branch node

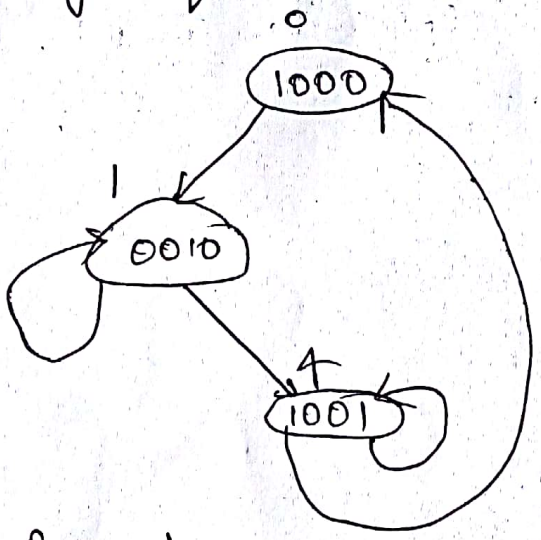
Scanned by CamScanner

The newly created node = 1001

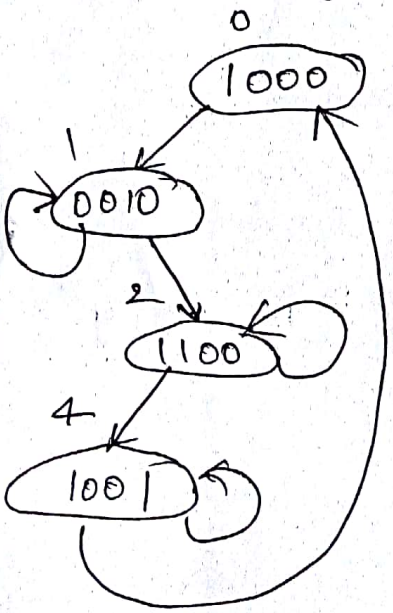


bit Number

"1" → consider first digit of newly created Node (i,e "1"). so insert the newly created node is right of "1" branch node.



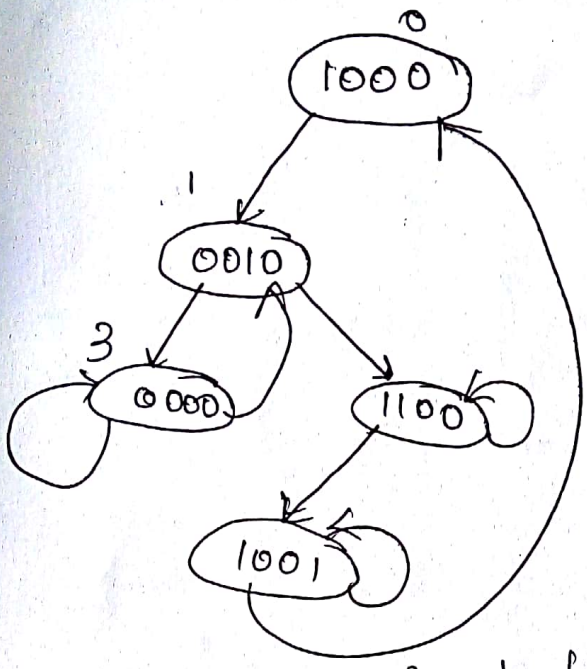
⇒ Now, insert key K = 1100. (Same procedure).



1100 inserted.

Scanned by CamScanner

insert key k = 0000.



0000 inserted.

Algorithms:

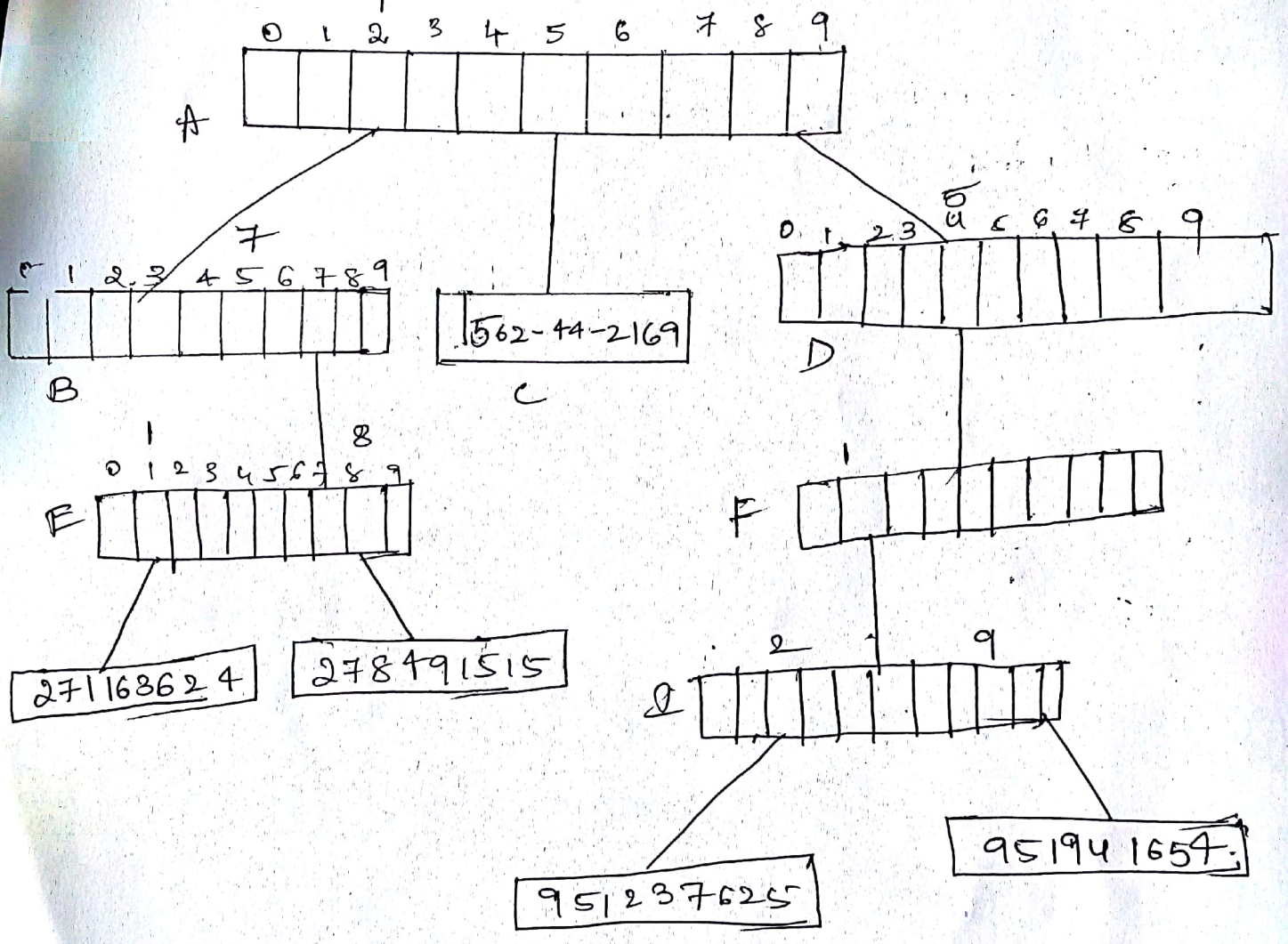
```
template <class K, class E>
void patricia<K,E>::insert(const K& k, const E& e)
{// insert e into the patricia. k is the key.
  if (!root) // patricia is empty
  {
  root = new patNode <K,E> (0,K,e);
// create a patNode and set its bitNumber, key and element fields
  root->leftchild = root; return;
  }
patNode <K,E> *y = Nsearch(K);
// Nsearch returns pointer to last node seen in search for k.
if (y->key == K) { y->element = e; return; }
```

Scanned by CamScanner

```
// update old element.
// New element, A new node with e is to be inserted.
// Find first bit where k and y->key differ.
for(int j=1; bit(k,j) == bit(y->key,j); j++);
// search Patricia using first j-1 bits of 'k.
patNode <K,E> *s = root->leftchild, *p = root;
While ((s->bitNumber > p->bitNumber) && (s->bitNumber < j))
{
    p=s;
    if(!(bit(k,s->bitNumber))) s = s->leftchild;
    else s = s->rightchild;
}
// insert 'x' as a child of p.
patNode <K,E> * z = new patNode <K,E>(j,k,e);
if (bit(k,j)){z->leftchild=s; z->rightchild=z;}
else
{z-> leftchild = z; z-> rightchild = s;}
if(s==p->leftchild) p->leftchild = z;
    else p->rightchild = z;
    return;
}
```

Scanned by CamScanner

Name	Social Security Number.
Jack	951-94-1654
Jill	562-44-2169
Bill	271-16-3624
Kathy	278-49-1515
April	951-23-7625.



Searching :-

Searching a trie for an element whose key is 'k' requires breaking up 'k' into its constituent characters / digits. and following branches determined by these characters.

Scanned by CamScanner

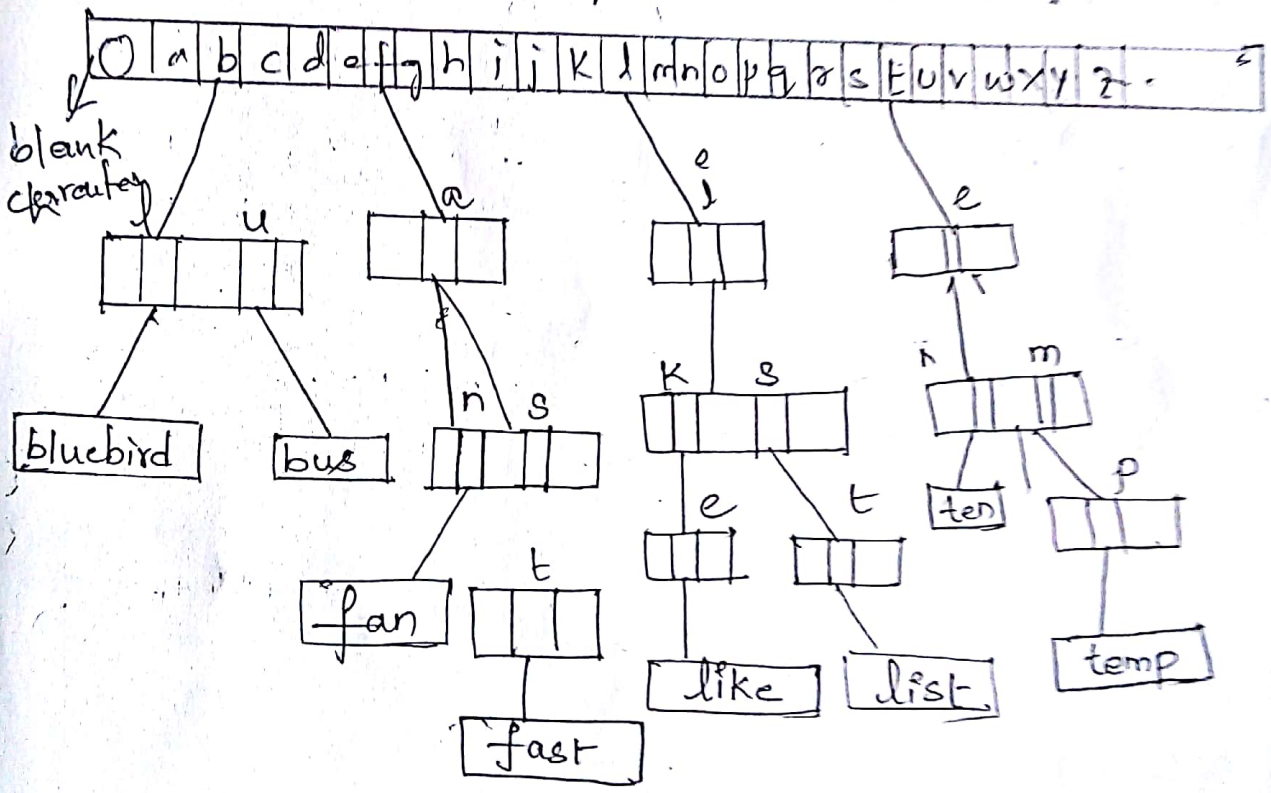
Multiway Tries:

* Multiway tries is a structure that is particularly useful when key values are of varying size.
* Multiway tries data structure is a generalization of the binary tries
* The tries contains two types of nodes: element and branch.

Element nodes are shaded while branch nodes are not shaded.

Element node has only one data field.

Branch node contains pointers to subtrees.



eg: An another Example of a trie Suppose that we have a collection of student records that contain fields such as student name, roll Social Security number and name

Scanned by CamScanner

To search for a key K = 27849151515

First digit is "2"

(i) Now check the "A" branch node contains any pointer from "2".

(ii) from "2" there is a pointer to "B" branch node.

Now goto key K = 27849151515

second digit = "7"

(iii) Now check is there any pointer from "7" in "B" branch node.

(iv) from "7" there is a pointer to "E" branch node.

Now goto key K = 27849151515.

Third digit = 8

(v) Now check is there any pointer from "8" in "E" branch node.

(vi) From "8" there is a ~~branch node~~ pointer to Element Node.

{ if the pointer reaches to element Node means Search ends at that position }

Now compare Element data with key "K"

Key K = 27849151515

Element Node = 27849151515 } Same

⇓

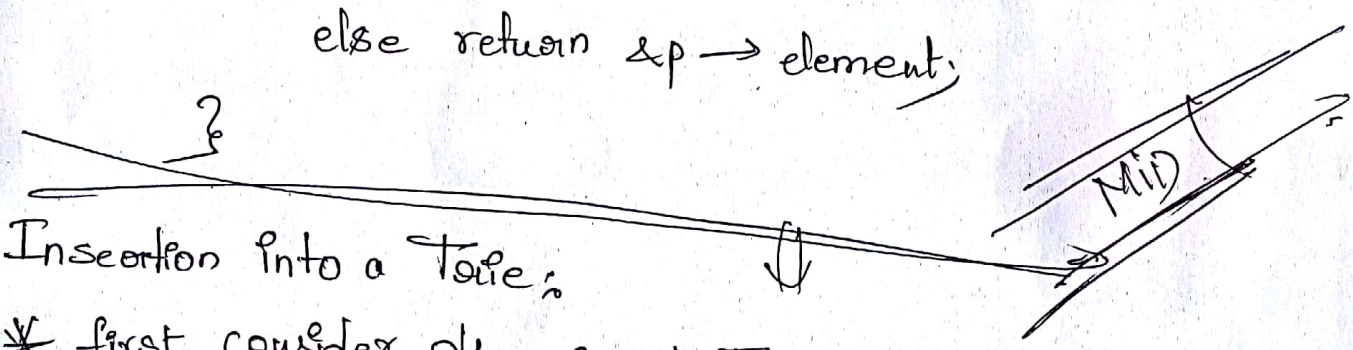
Nothing but

Search/found element.

Scanned by CamScanner

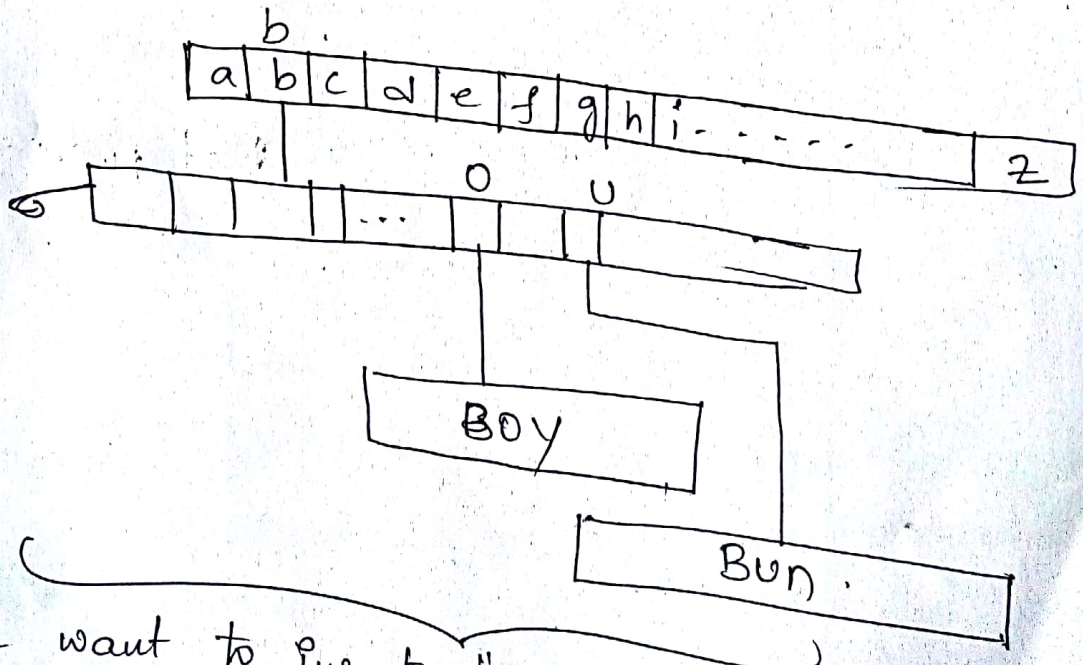
**Alg: template <class K, E>

```
E * Trie<K,E>::search (const K&k) const
{ // Search a trie. Return a pointer to the element
  // whose key is k
  // Return NULL if no such element
  TrieNode <K,E> *p = root;
  for (int i=1; p is a branch node; i++)
     p = p->child[digit(k,i)];
  if (p==NULL || p->key!=k) return NULL;
     else return &p->element;
}
```



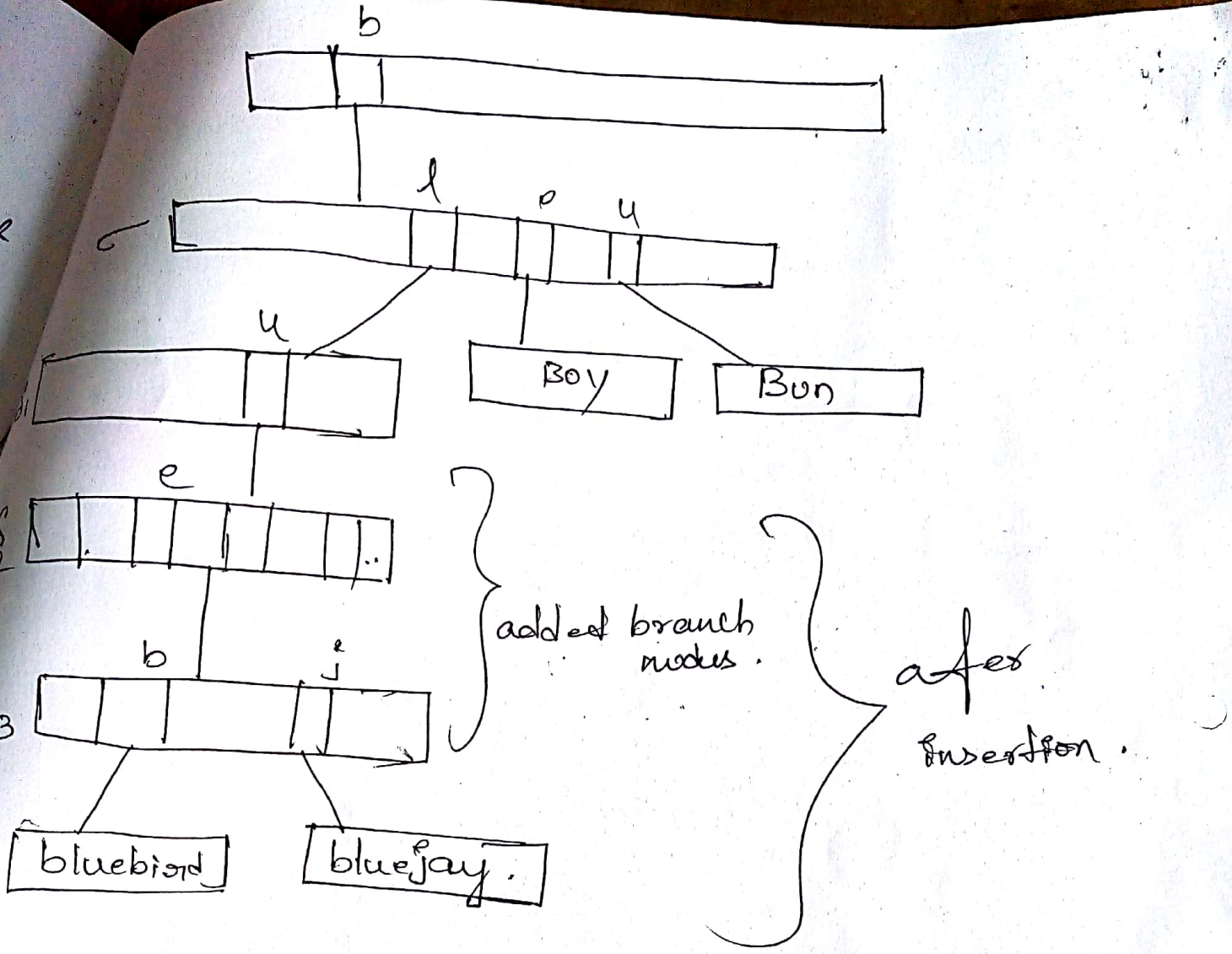
Insertion into a Trie:

* first consider the sample Trie.



& want to insert "bluebird" & "bluejay".

Scanned by CamScanner



Deletion from a Trie:

* Before deleting an node, we need to check/or to perform search operation to know whether the element is present (or) not.
* If there is direct link from main/root branch node to element node simply delete the element node.
* eg: if you want to delete "bluejay" from multiway trie.
  (i) first search whether the trie contains bluejay (or) not.

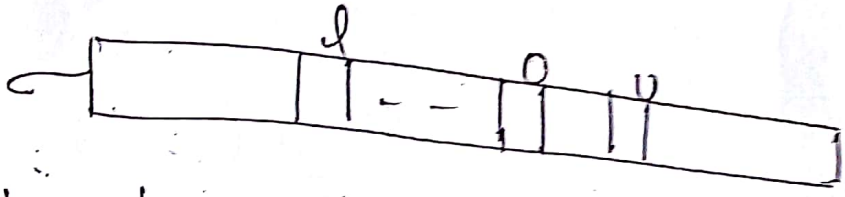
Scanned by CamScanner

* (i) While insertion, first observe the first Letter of the two words "b" ← common for both words.

(ii) In given Multiway trie there is a "σ" Sigma branch node, the branch node contains different pointers.

(iii) What is the second Letter of our new words ("e" "l").

So, insert "l" in "σ" branch node.



(iv) Now observe third Letter in each branch nodes [new words]

"u" ↔ common for both words.

So now add S_1 branch node - contains pointer "u".

(v) Now observe 4th Letter in each nodes [new node]

"e" → common for both

So now add S_2 branch node, contains only one pointer "e".

(vi) Now observe 5th Letter in each nodes [new nodes]

bluebird ↔ bluejay.

So, here bluebird contains "b"
bluejay contains "j"

} Now insert S_3 branch node with two pointers.

Scanned by CamScanner

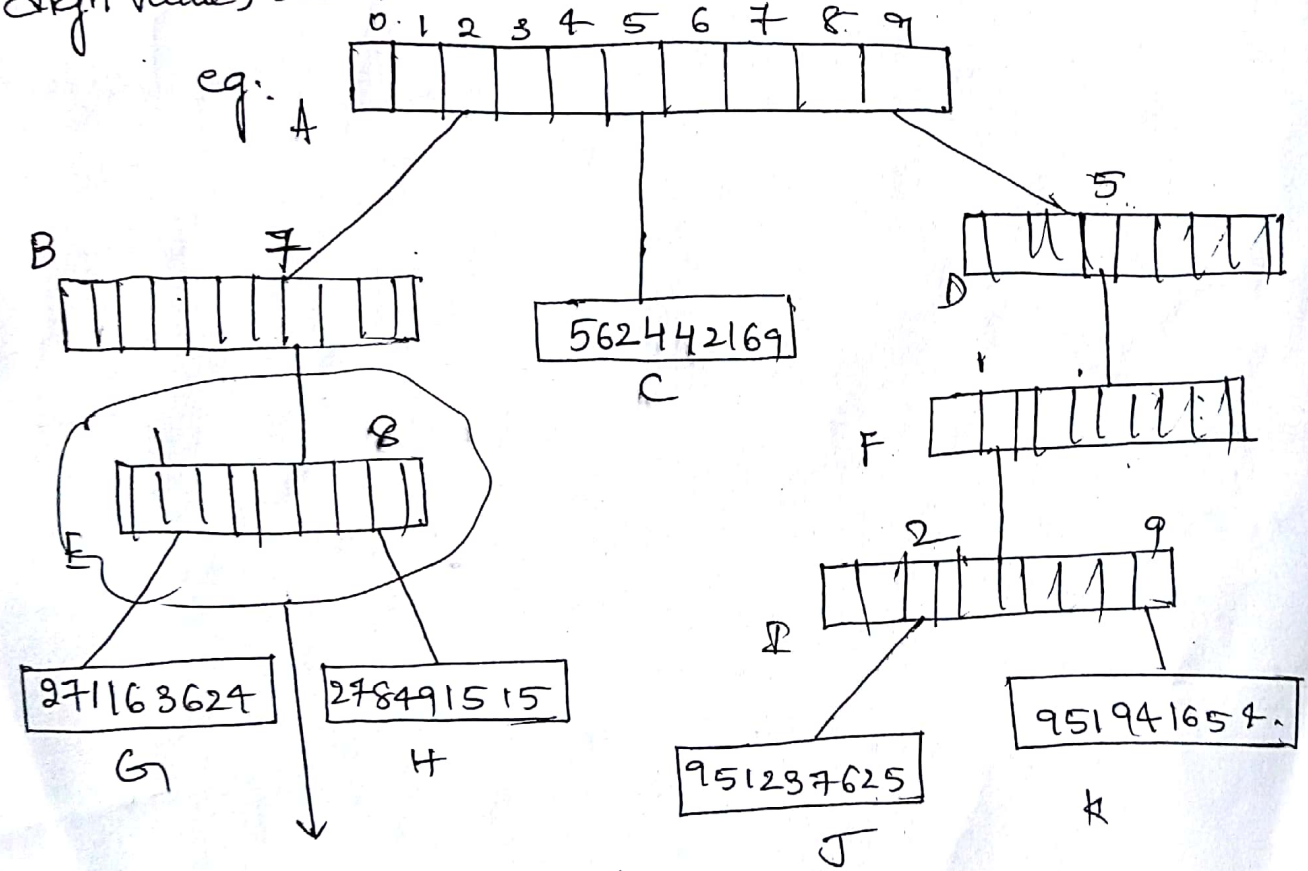
Required and Alternative Node Structures:

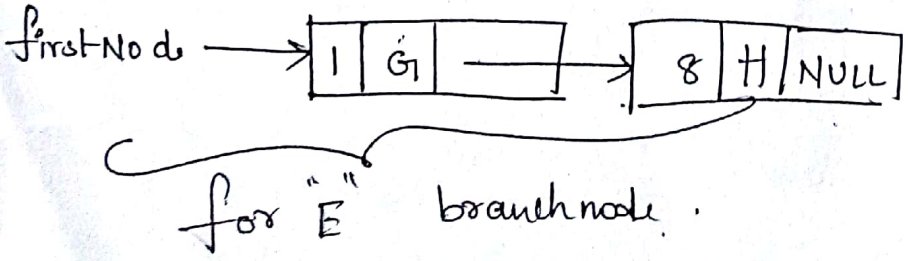
The use of branch nodes that have as many fields as the radix of the digits. Each time we are inserting branch node we are not using all child fields.

We can Reduce the space requirements, at the Expense of increased search time, by changing the node structure. Some of the possible alternative structures for the branch node of a trie.

A chain of Nodes:

Each node of the chain has three fields digitValue, child, and next.

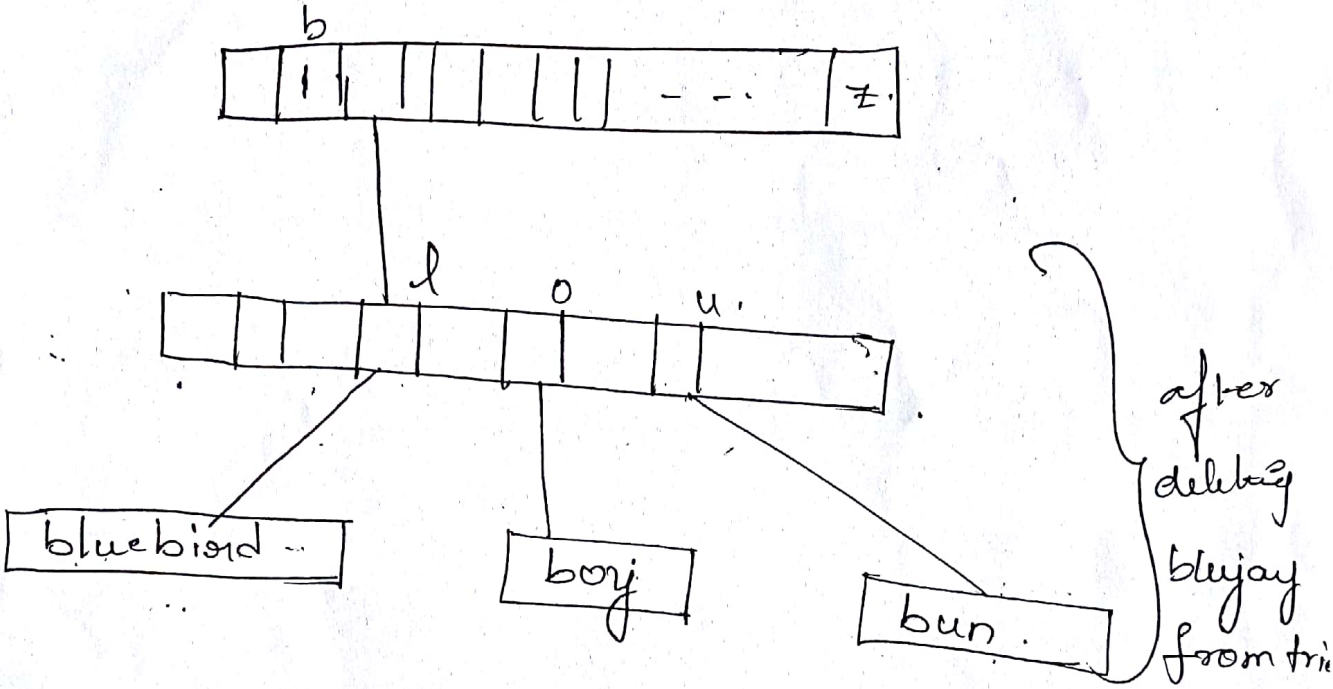




Scanned by CamScanner

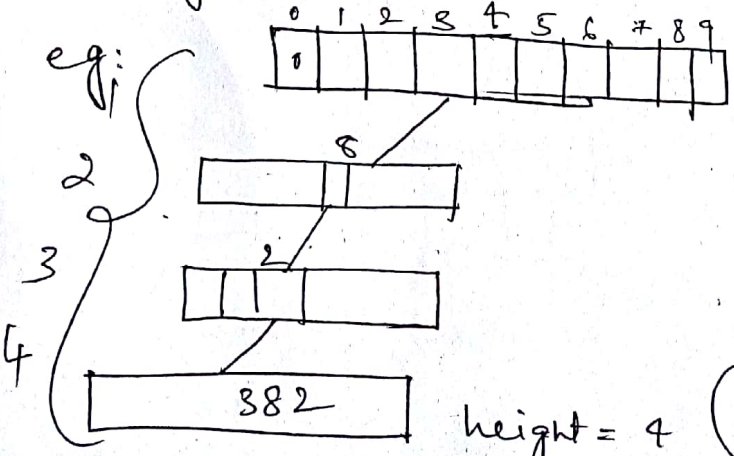
(2) Now perform delete operation, after deleting the node, we can apply (perform) roll up operation. because the trie contains now only word bluebird. No need of using S_1, S_2, S_3 Levels.

(3) Perform roll up operation upto "—". Now give pointer from 'l' to element node (bluebird).



Height of a trie:

* Root node to element node has a branch node for every digit in a key.
* The Height of a trie is almost (no of digits + 1)

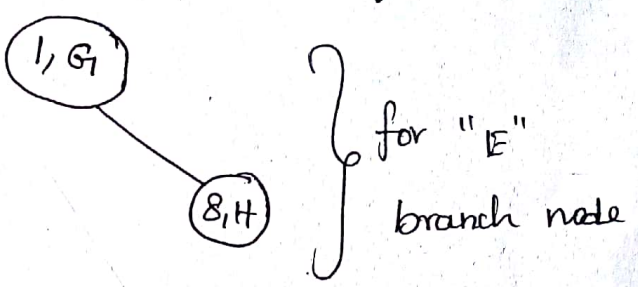


382

Scanned by CamScanner

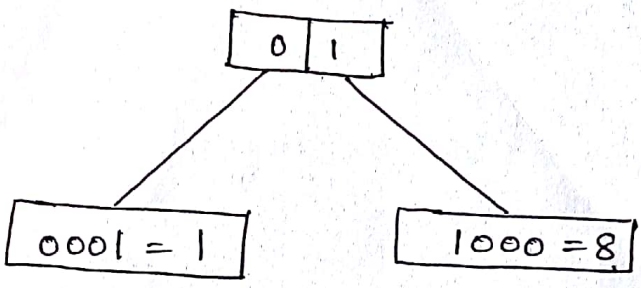
(Unbalanced) binary Search Tree:

Each node of the binary Search Tree has a digit value and a pointer to the Subtrie for that digit value.



A binary Trie:

binary trie for node "E".



A Hash Table:

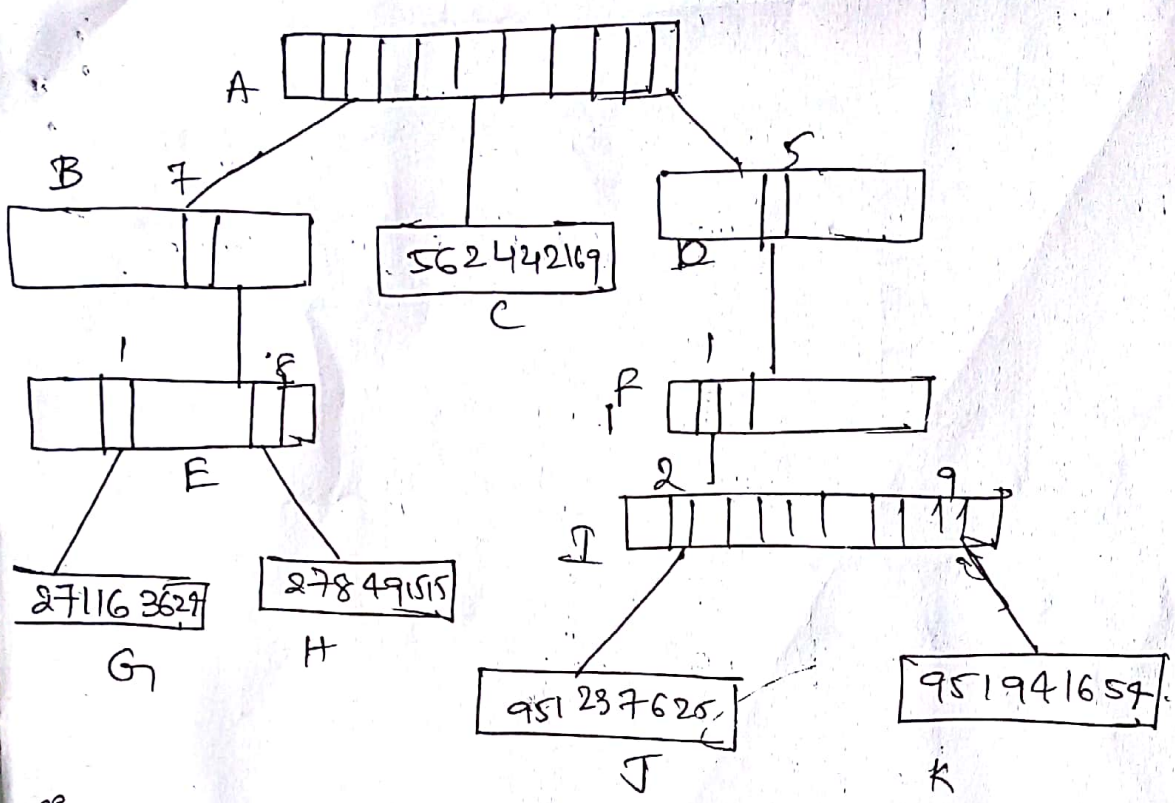
* When a hash table with a sufficiently small loading density is used, [Load factor = No of elements/size].

* Null child fields in a branch node vary from node to node also to increase as we go down the trie, Maximum Space efficiency is obtained by consolidating all of the branch nodes into a Single hash table.

⇒ (i) Each node in the trie is assigned a number, and each parent to child pointer is replaced by a triple of the form (currentNode, digit value, childNode)

// The numbering scheme is used to distinguish branchnodes and element nodes.

Scanned by CamScanner



(ii) Now write down the tuples for each and every doors, before we need to assign numbers to branch nodes.

Branch nodes:	A	B	C	D	E	F	G	H	I	J	K

Suppose, we have "10" element nodes.
(0-9) for element nodes.
from "10" onwards assign numbers to branch nodes

branch node:	A	B	C	D	E	F	G	H	I	J	K
	10	11	0	12	13	14	1	2	15	3	4

(iii) Tuples for "A" → A contains 3 childrens so 3-tuples.

A → (10, 2, 11)
→ (10, 5, 0)
→ (10, 9, 12).

Scanned by CamScanner