Unit-VI

Digital Search Tree (DST)

→ A DST is special type of Binary tree in which every node stores Binary number (or) Binary element

→ In general the DST can used mostly in two applications

1) IP routing
2) Security systems (Firewalls)

→ The operations like insertion, deletion & search can be performed very efficiently in DST than AVL Tree & Binary Search Tree

→ The DST occupies less space than AVL & Binary Search Tree

Time complexity

Best case ⇒ O(1)

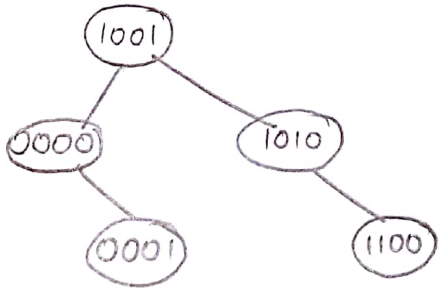
Average case ⇒ O(logN)

'N' is height of DST

worst case ⇒ O(b)

'b' is no of bits of an element

Ex :-



Operations

Three Basic operations performed on DST

1. Insertion
2. Deletion
3. search

→ To insert an element in DST we need to follow the procedure steps

Step① :- check whether the tree is empty or not

Step②:- If the tree is empty then new element is inserted into the DST and which will be treated as rootnode

Step③:-

→ If the tree is not empty then the new element is inserted either left subtree (or) right subtree to the rootnode by checking first bit in the new element. If the

first bit of new element is zero then the new element is to be inserted as the left subtree to the rootnode. Otherwise (1st Bit = 1) the new element is to be inserted as the right subtree to the rootnode

Step ④:- If the node is already existed in either LST (or) RST (Right Subtree) to the rootnode then the next new element is to be inserted either (left subtree (or) Right Subtree to the existed node by comparing next Bit by Bit with the new element

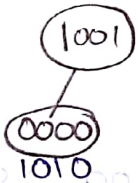
⑤ this procedure will be continued at every node in the DST recursively.

Ex:- construct a DST with four bits by the following elements. 1001, 0000, 1010, 0001, 1000, 0011, 1100, 0100, 1011, 0101, 1101, 0110, 1111, 0111

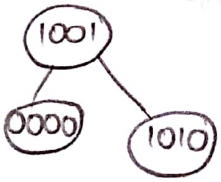
0 - left
1 - right

Insert 1001

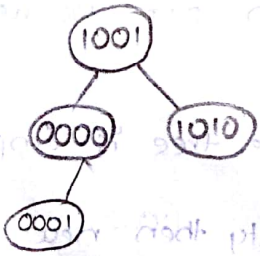
Insert 0000



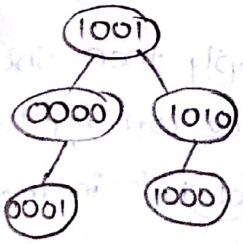
Insert 1010



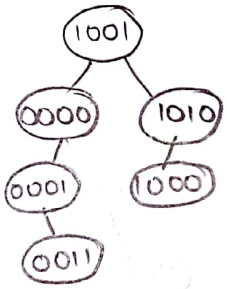
Insert 0001



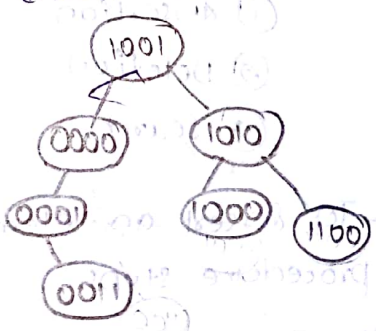
Insert 1000



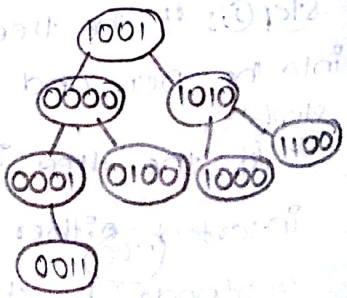
Insert 0011



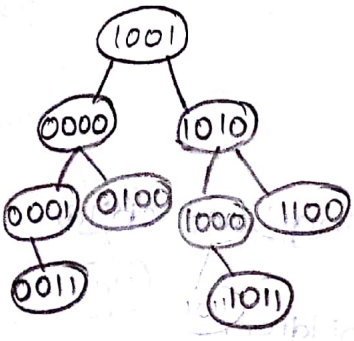
Insert 1100



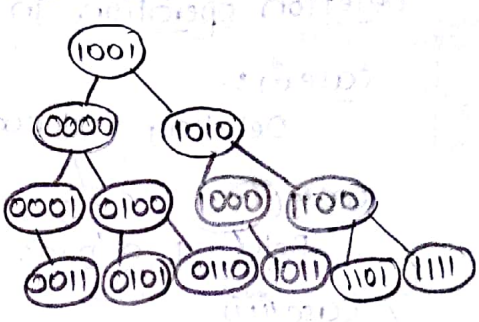
Insert 0100



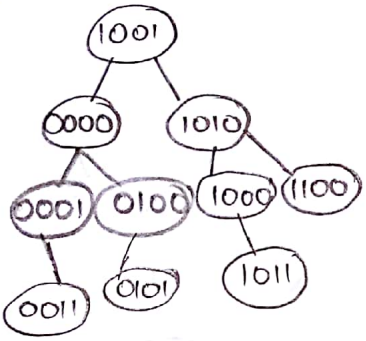
Insert 1011 :-



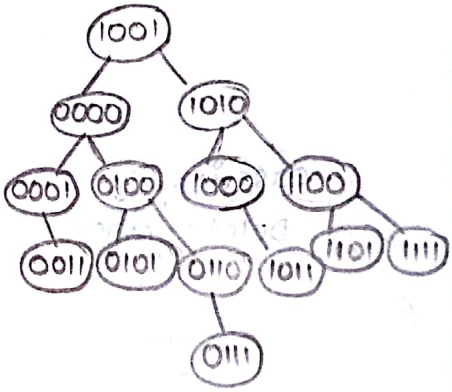
Insert 1111:-



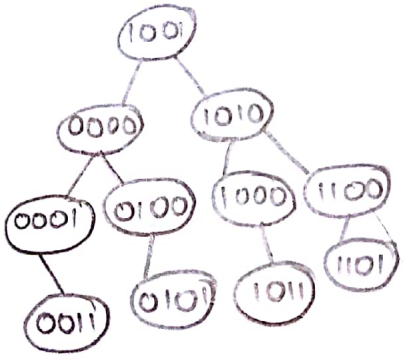
Insert 0101 :-



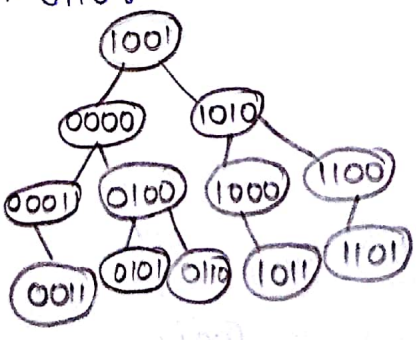
Insert 0111



Insert 1101 :-



Insert 0110 :-



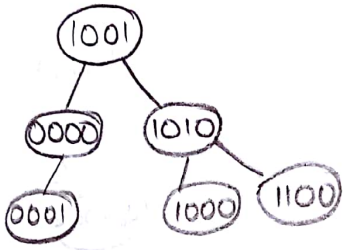
Deletion operation in DST :-

case(i) :-
Deleting leaf node

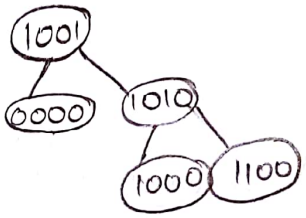
case(ii)
Deleting a node which is having one child

case(iii)
Deleting a node having 2 children

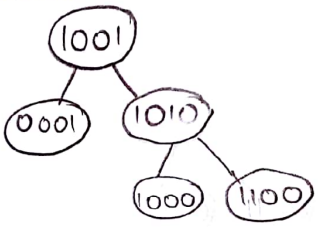
Ex :- Delete the elements from DST



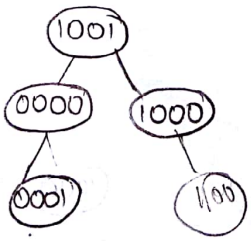
case(i) Eg :-↓
Delete 0001



Case(ii) Eg :-↓
Delete 0000



case(iii) Eg :-↓
Delete 1010



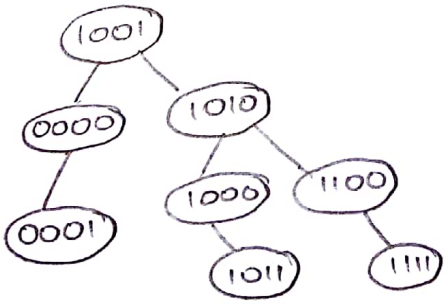
Search operation in DST

To search an element in the DST we have to follow the following procedure steps

procedure steps

1) check whether the tree is empty or not
2) If the tree is empty then the search element cannot be found
3) If the tree is not empty then the search process starts with rootnode onwards
4) If the search element is matched with the rootnode element then search element is found at rootnode
5) If the search element is not matched with the rootnode element then the 1st bit of search element will be checked if it is zero, then the search operation willbe performed recursively on the leftsubtree of the rootnode If it is one, the search operation will be performed recursively on the rightsubtree of the rootnode
6) This process will be continued untill the leafnode of either leftsubtree (or) right subtree of the rootnode

Ex :-



Search the element : 1011

the element 1011 is found at right subtree of rootnode

Chapter 12
Digital Search Structures

- Digital Search Trees
- Binary Tries and Patricia
- Multiway Tries

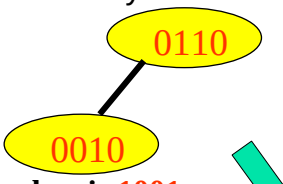
Digital Search Tree

- A digital search tree is a binary tree in which each node contains one element.
- Assume fixed number of bits.
- Not empty
  - Root contains one dictionary pair (any pair).
  - All remaining pairs whose key begins with a 0 are in the left subtree.
  - All remaining pairs whose key begins with a 1 are in the right subtree.
  - Left and right subtrees are digital subtrees on remaining bits.

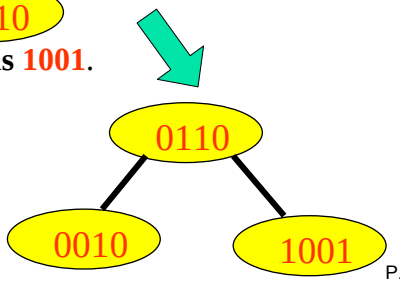
Example of Digital Search Tree

- Start with an empty digital search tree and insert a pair whose key is 0110.

- Now, insert a pair whose key is 0010.

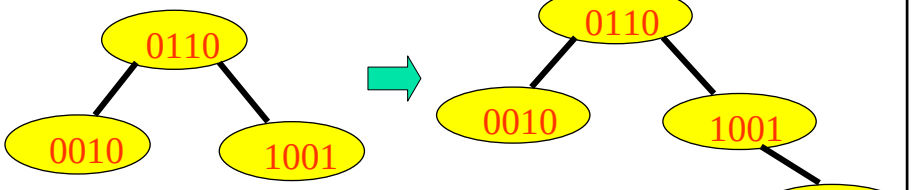


- Now, **insert a pair whose key is 1001**.

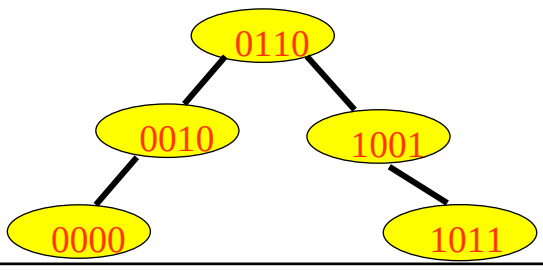


Example

- Now, **insert a pair whose key is 1011**.

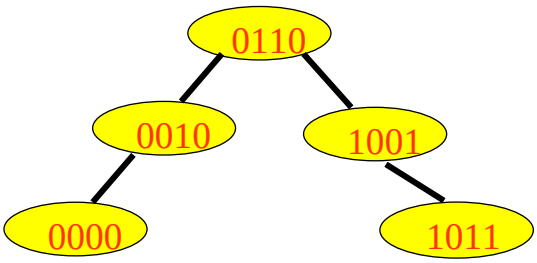


- Now, **insert a pair whose key is 0000**.



Search/Insert/Delete

- Complexity of each operation is **O(#bits in a key).**
- #key comparisons = **O(height).**
- Expensive when keys are very long.



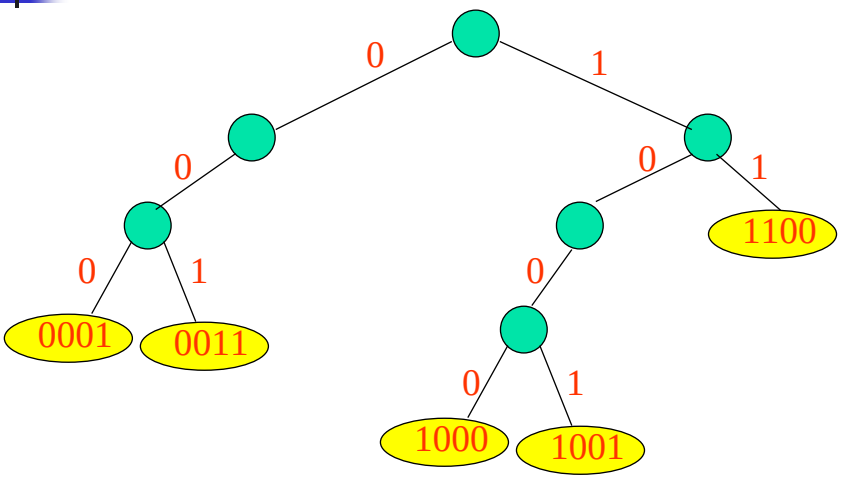
Applications of Digital Search Trees

- Analog of radix sort to searching.
- Keys are binary bit strings.
  - Fixed length – 0110, 0010, 1010, 1011.
  - Variable length – 01, 00, 101, 1011.
- Application – IP routing, packet classification, firewalls.
  - IPv4 – 32 bit IP address.
  - IPv6 – 128 bit IP address.

Binary Trie

- Information Retrieval.
- At most one key comparison per operation, search/insert/delete.
- A Binary trie (pronounced *try*) is a binary tree that has two kinds of nodes: branch nodes and element nodes. For fixed length keys,
  - Branch nodes: Left and right child pointers. No data field(s).
  - Element nodes: No child pointers. Data field to hold dictionary pair.

Example of Binary Trie

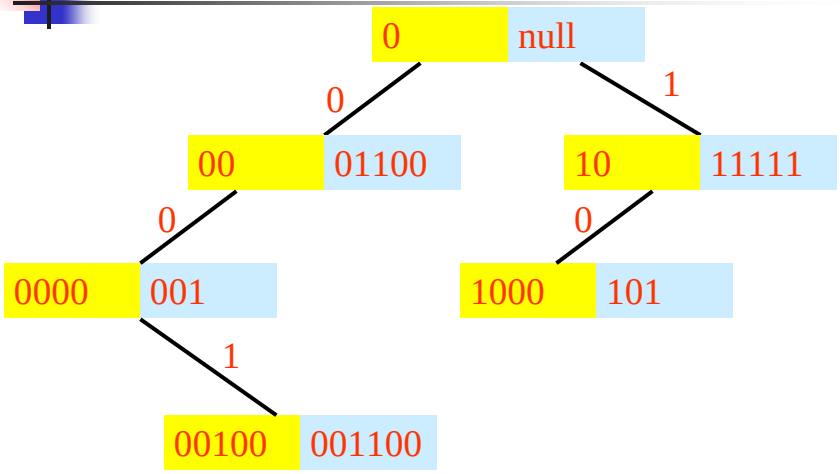


At most one key comparison for a search.

Variable Key Length

- Left and right child fields.
- Left and right pair fields.
  - Left pair is pair whose key terminates at root of left subtree or the single pair that might otherwise be in the left subtree.
  - Right pair is pair whose key terminates at root of right subtree or the single pair that might otherwise be in the right subtree.
  - Field is null otherwise.

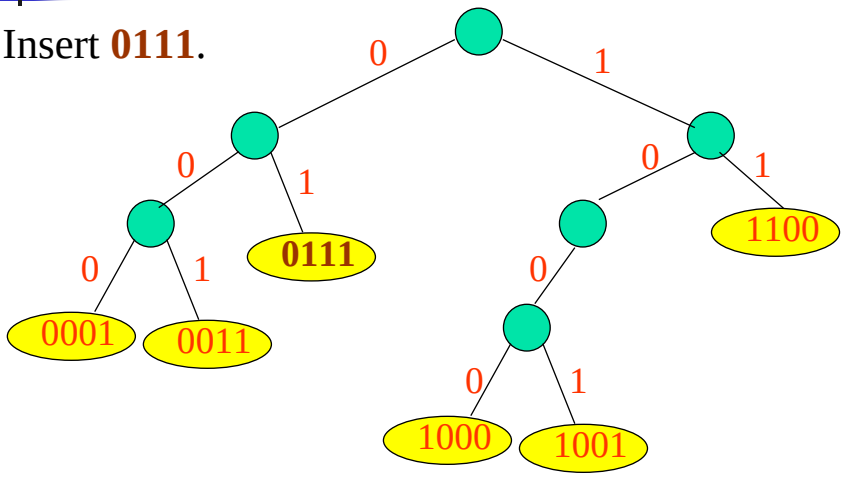
Example of Variable Key Length



At most one key comparison for a search.

Fixed Length Insert

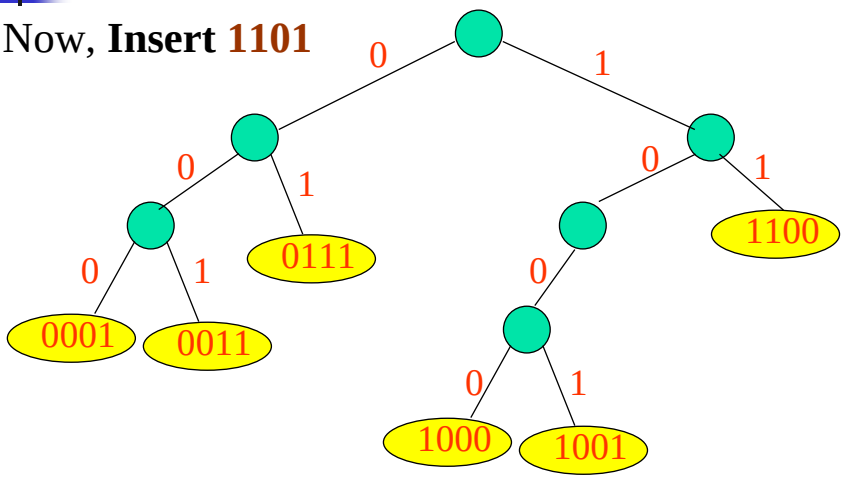
Insert **0111**.



Zero compares.

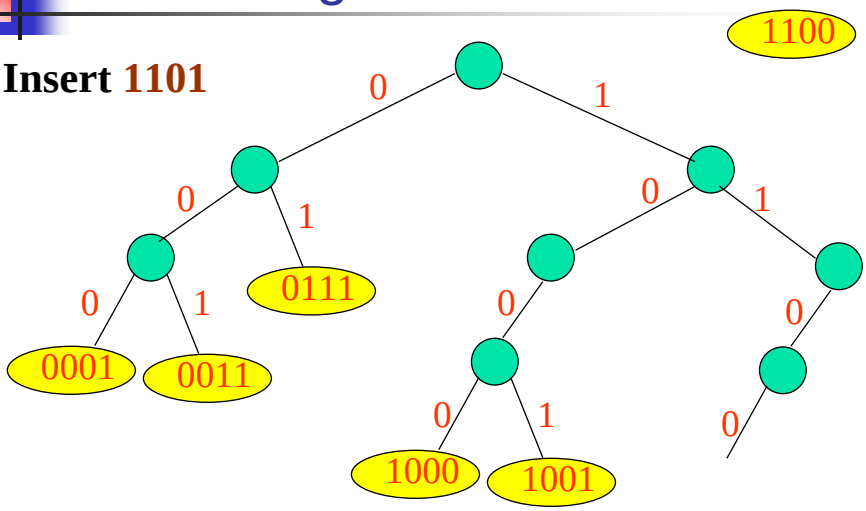
Fixed Length Insert

Now, **Insert 1101**



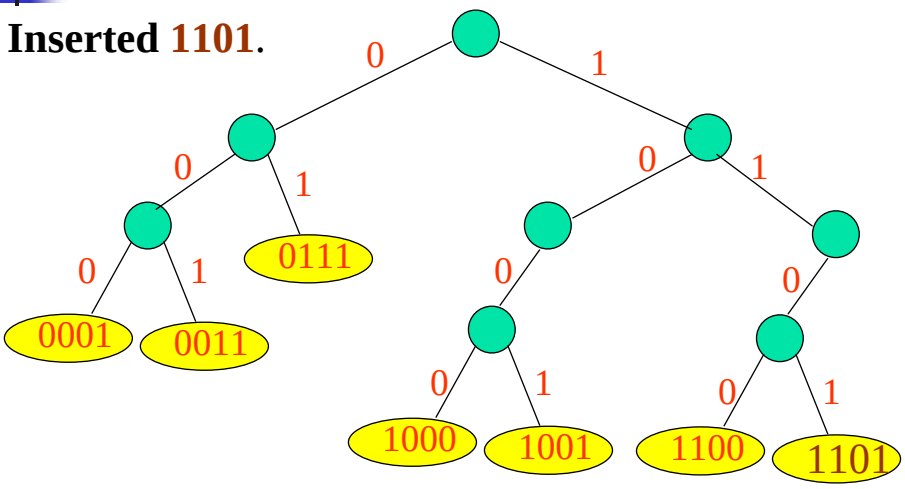
Fixed Length Insert

Insert 1101



Fixed Length Insert

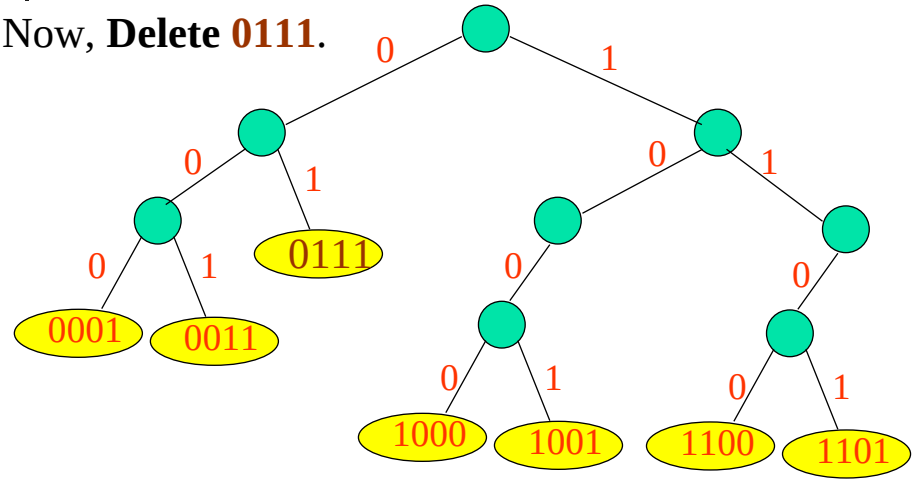
Inserted 1101.



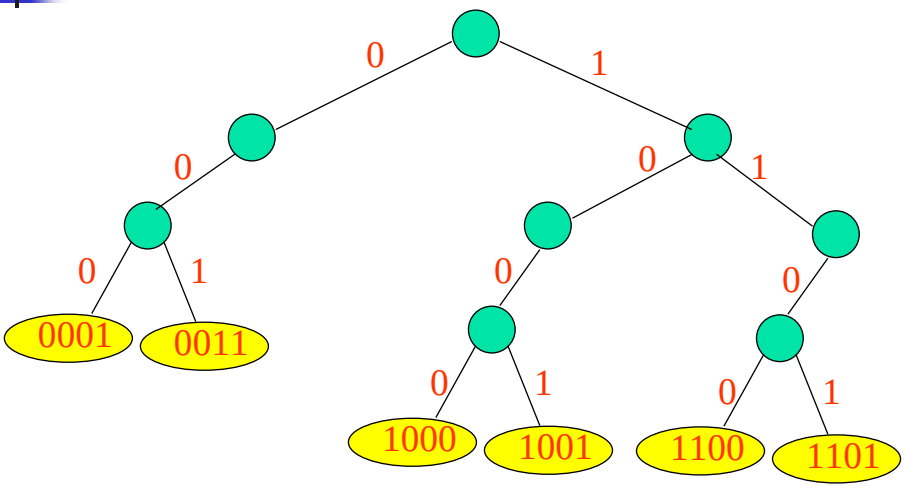
One compare.

Fixed Length Delete

Now, **Delete 0111**.



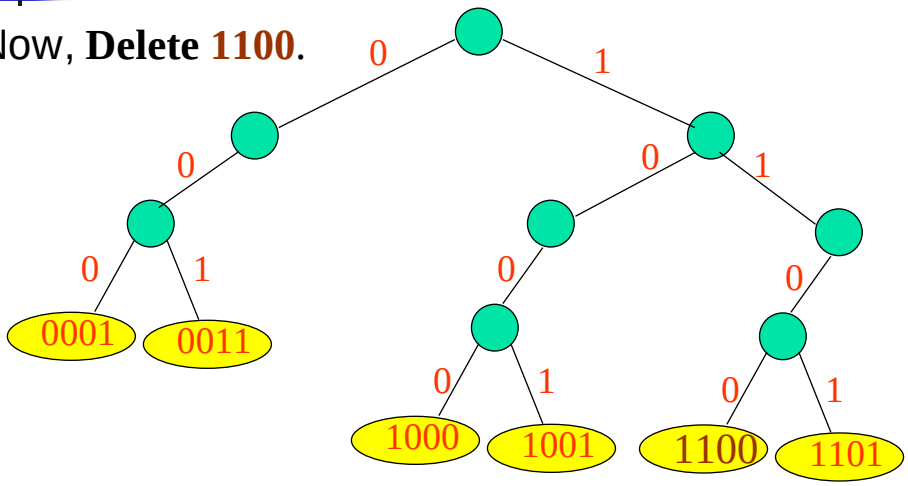
Fixed Length Delete



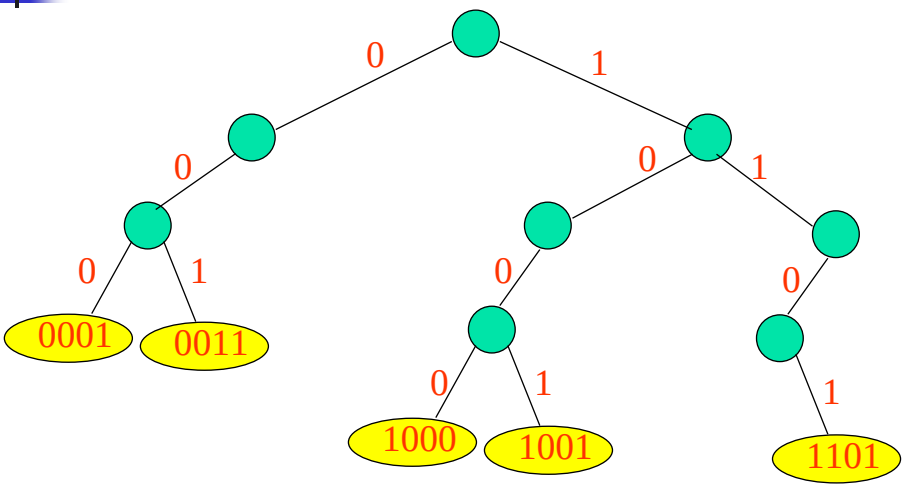
Delete 0111. One compare.

Fixed Length Delete

Now, **Delete 1100**.

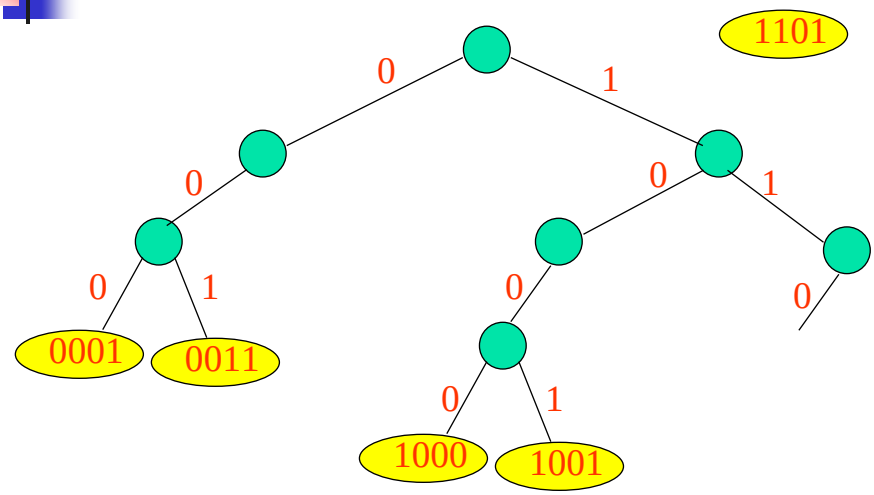


Fixed Length Delete



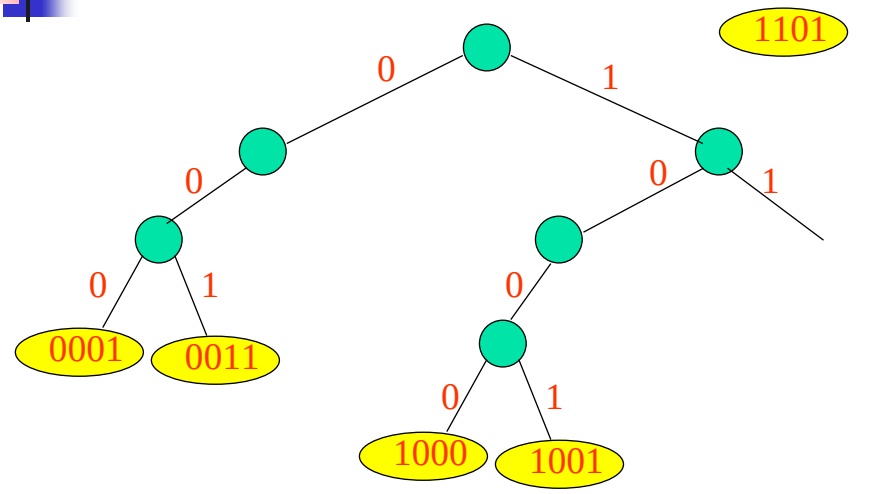
Delete 1100.

Fixed Length Delete



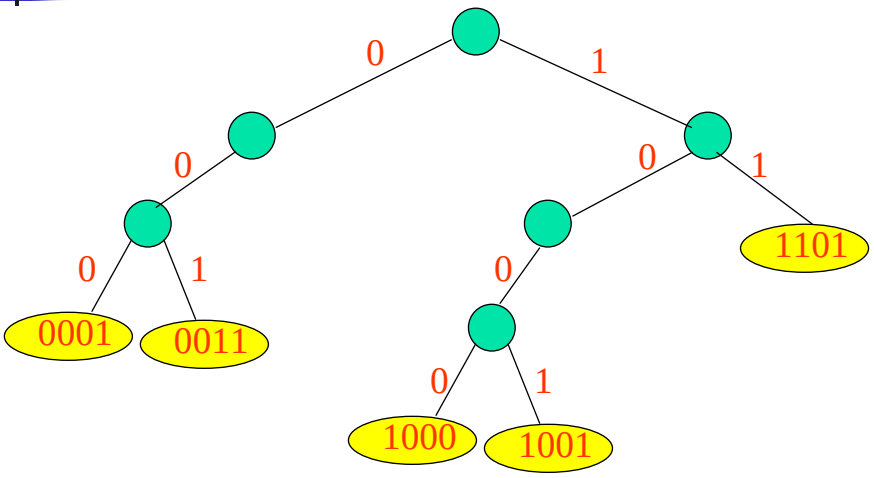
Delete 1100.

Fixed Length Delete



Delete 1100.

Fixed Length Delete



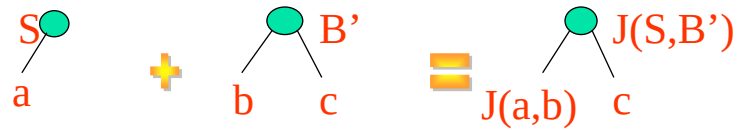
Delete 1100. One compare.

Fixed Length Join(S,m,B)

- Insert m into B to get B'.
- S empty => B' is answer; done.
- S is element node => insert S element into B'; done;
- B' is element node => insert B' element into S; done;
- If you get to this step, the roots of S and B' are branch nodes.

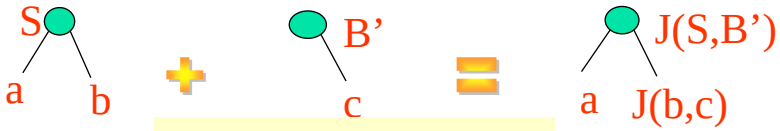
Fixed Length Join(S,m,B)

- S has empty right subtree.



J(X,Y) ⇨join X and Y, all keys in X < all in Y.

- S has nonempty right subtree.
  - Left subtree of B' must be empty, because all keys in B' > all keys in S.

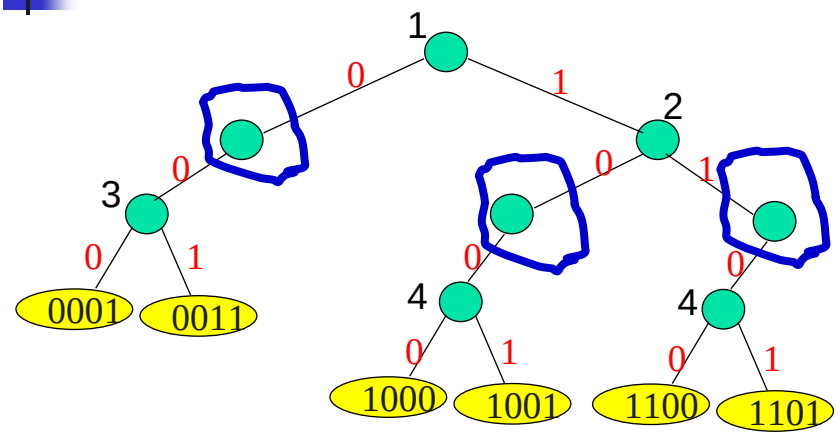


Complexity = O(height).

Compressed Binary Tries

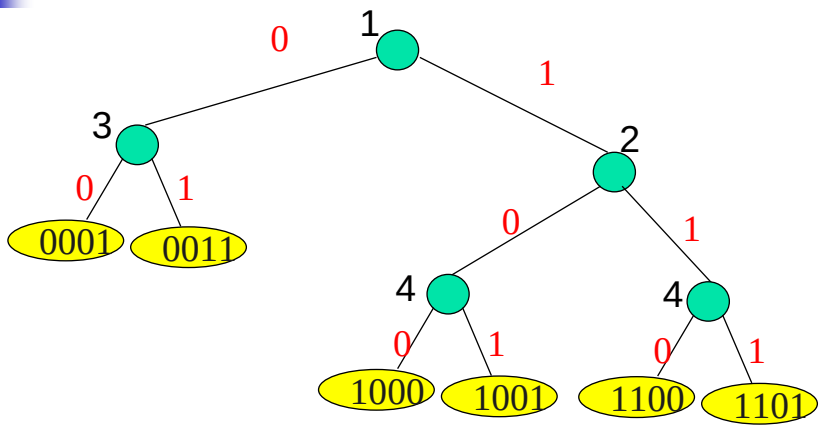
- No branch node whose degree is 1.
- Add a bit# field to each branch node.
- bit# tells you which bit of the key to use to decide whether to move to the left or right subtrie.

Example: Binary Trie



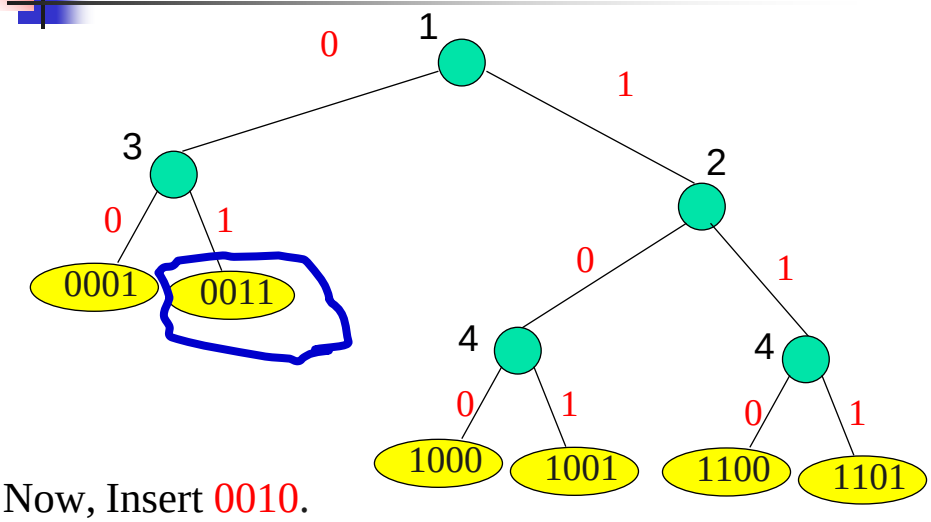
- **bit#** field shown in black outside branch node.

Example: Compressed Binary Trie



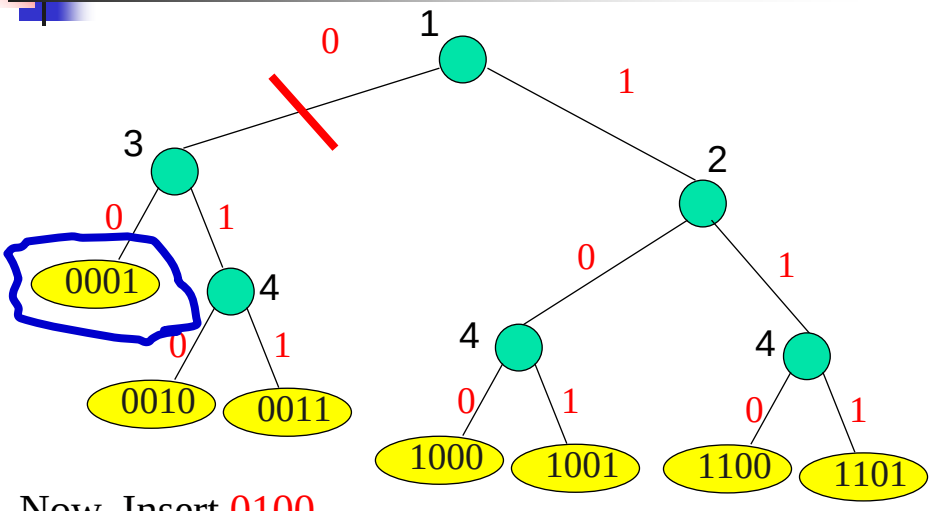
- bit# field shown in black outside branch node.
- #branch nodes = n – 1.

Insert



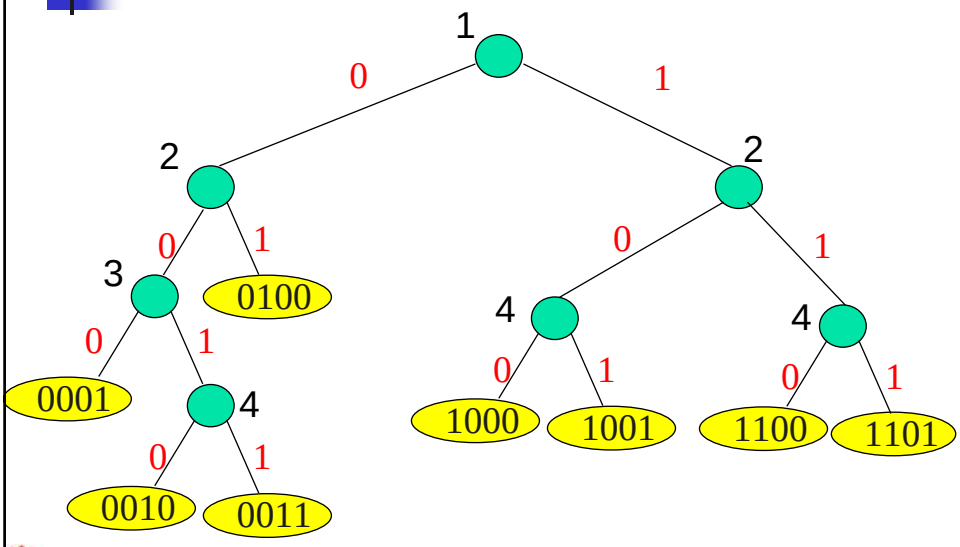
Now, Insert 0010.

Example: After Inserting 0010

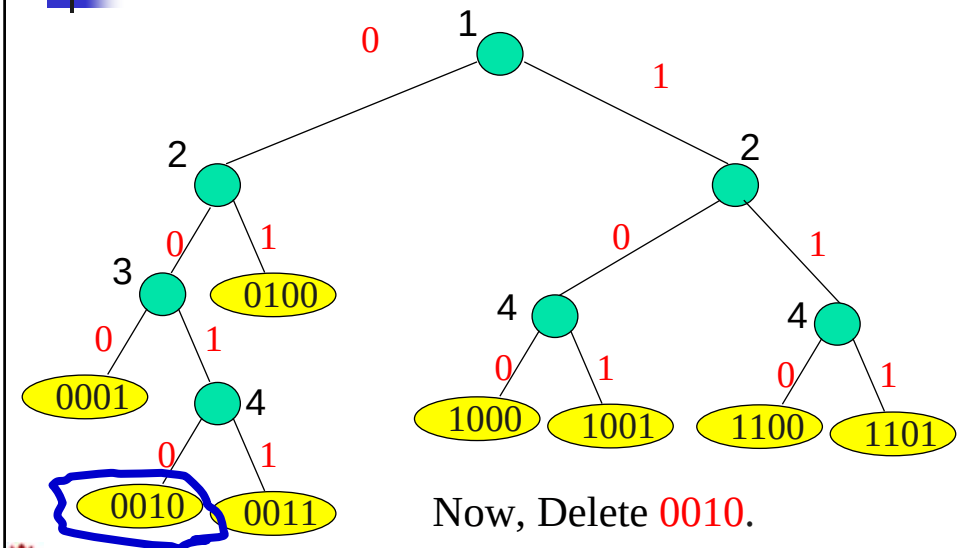


Now, Insert 0100.

Example: Insert 0100

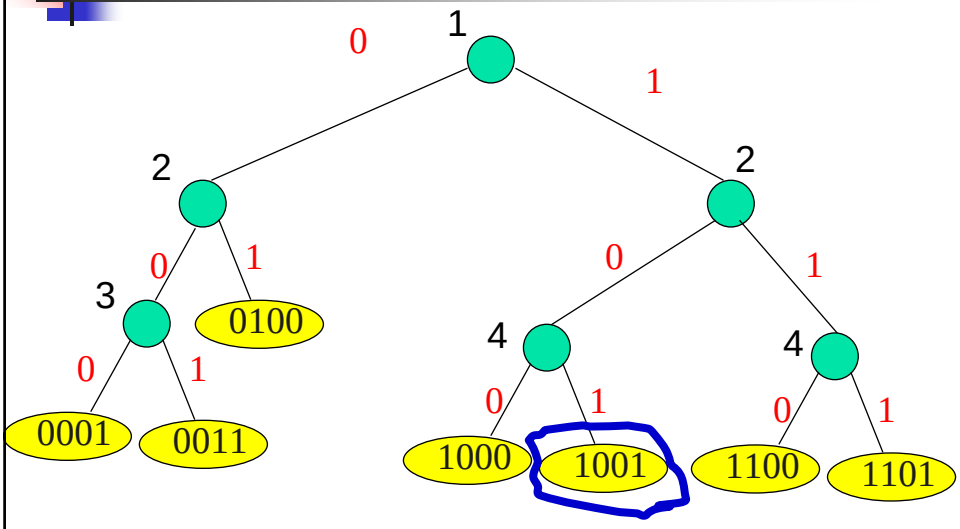


Delete



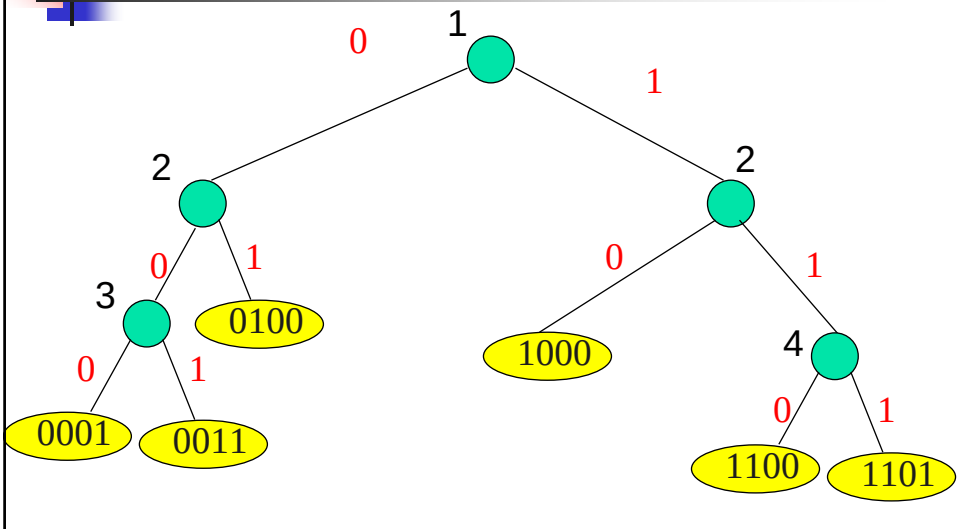
Now, Delete 0010.

Example: After Deleting 0010



Now, Delete 1001.

Example: After Deleting 1001



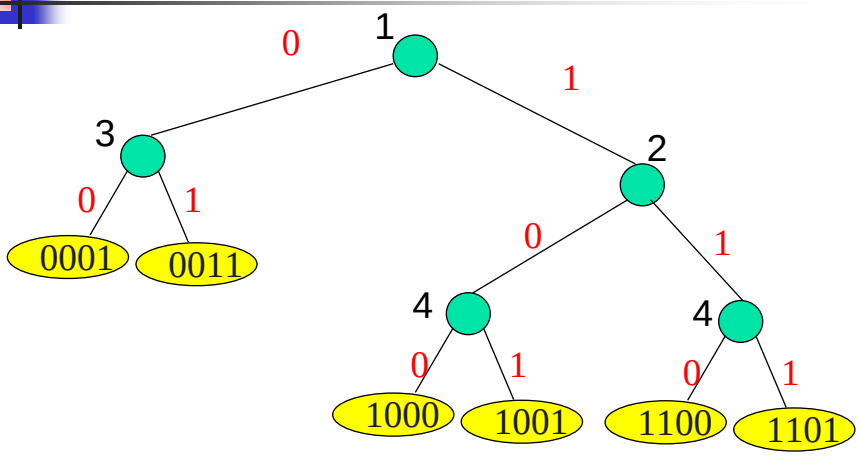
Patricia

- Practical Algorithm To Retrieve Information Coded In Alphanumeric.
- All nodes in *Patricia* structure are of the same data type (binary tries use branch and element nodes).
  - Pointers to only one kind of node.
  - Simpler storage management.
- Uses a header node that has zero bitNumber. Remaining nodes define a trie structure that is the left subtree of the header node. (right subtree is not used)
- Trie structure is the same as that for the compressed binary trie.

Node Structure

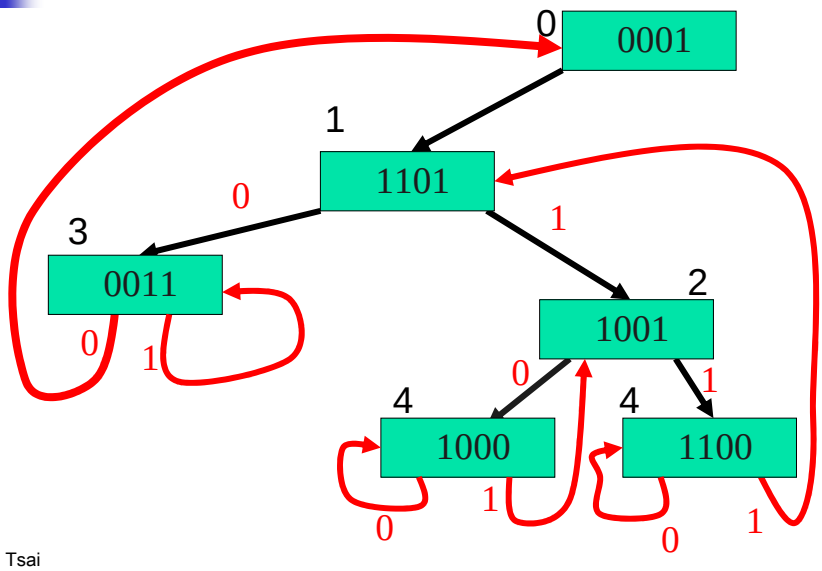
- bit# = bit used for branching
- LC = left child pointer
- Pair = dictionary pair
- RC = right child pointer

Compressed Binary Trie To Patricia



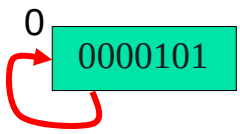
- Move each element into an ancestor or header node.

Example: Compressed Binary Trie To Patricia

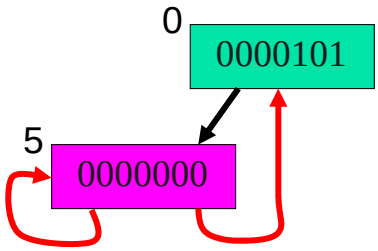


Insert

- Insert 0000101

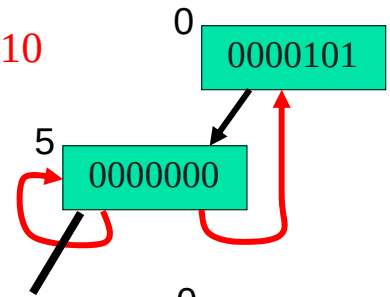


- Insert 0000000

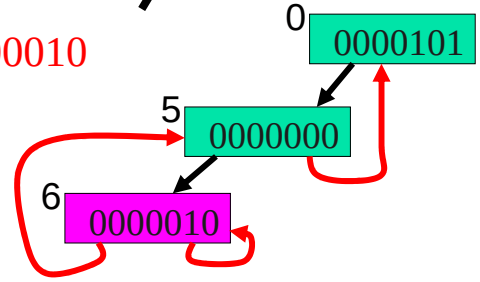


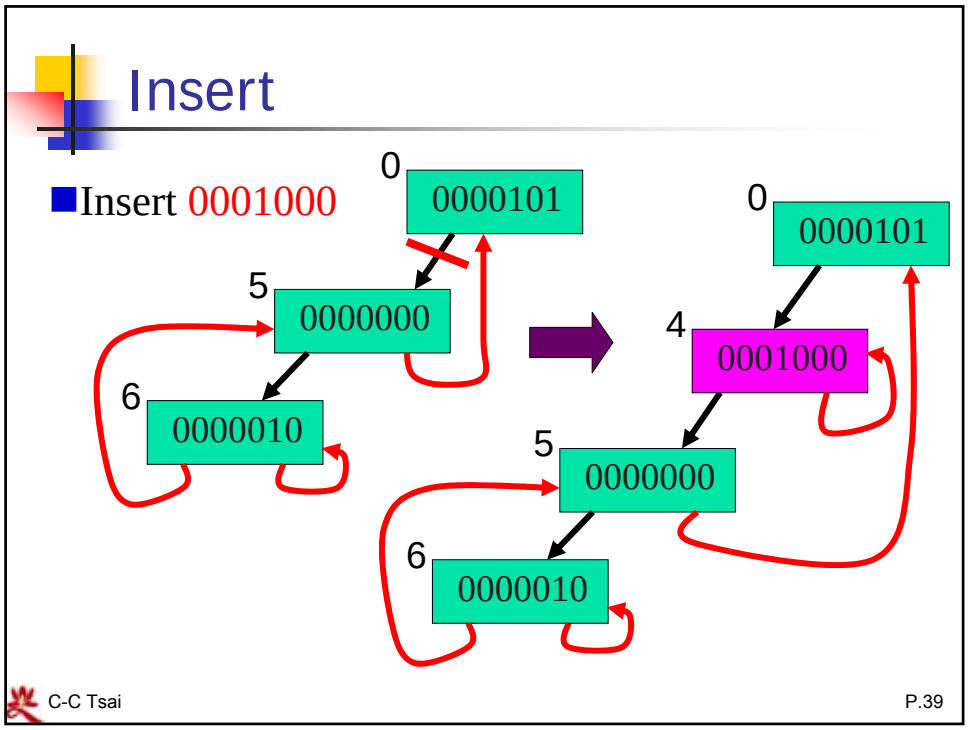
Insert

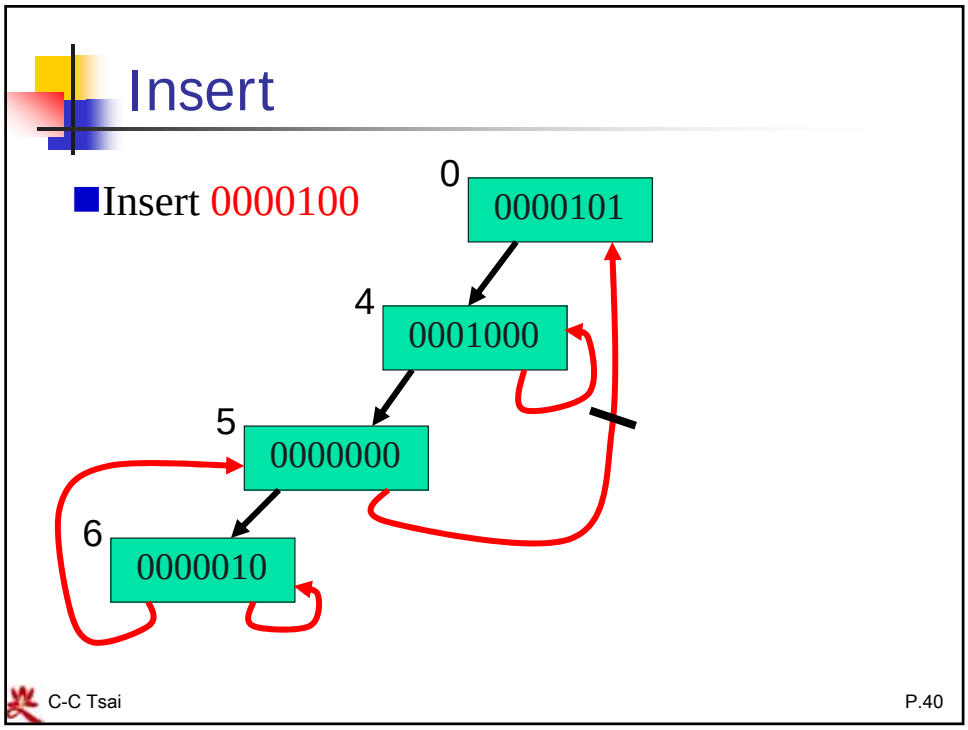
- Now, Insert 0000010



- Inserted 0000010

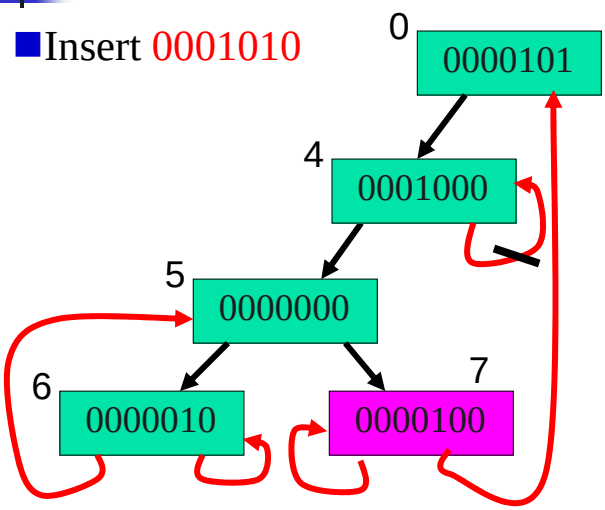



Insert
Insert 0001000
0
0000101
5
0000000
6
0000010
0
0000101
4
0001000
5
0000000
6
0000010
C-C Tsai
P.39


Insert
Insert 0000100
0
0000101
4
0001000
5
0000000
6
0000010
C-C Tsai
P.40

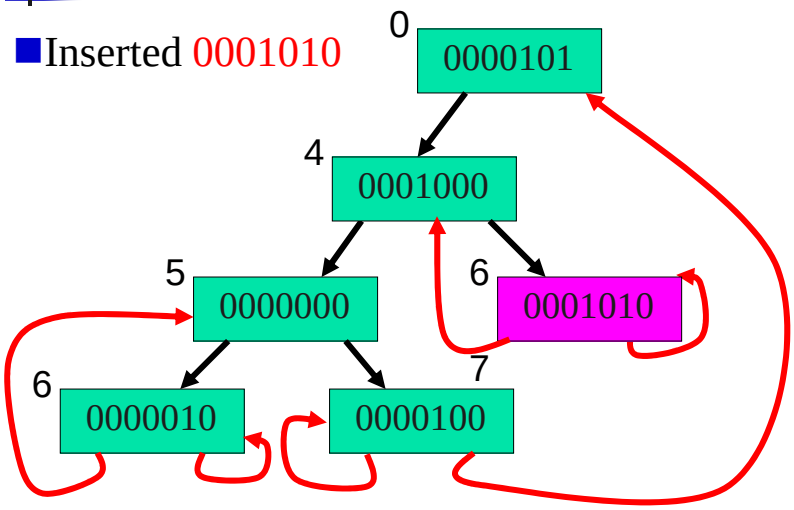
Insert

- Insert 0001010



Insert

- Inserted 0001010

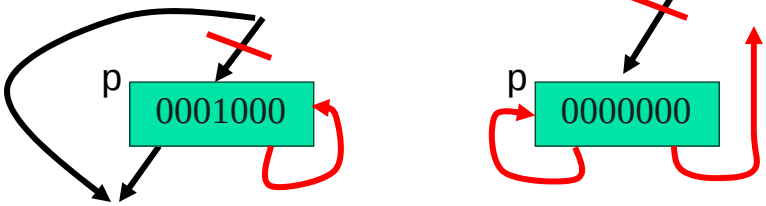


Delete

- Let p be the node that contains the dictionary pair that is to be deleted.
- Case 1: p has one self pointer.
- Case 2: p has no self pointer.

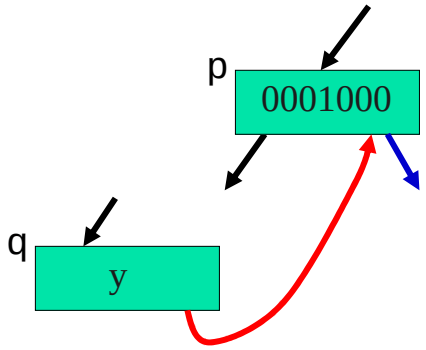
p Has One Self Pointer

- p = header => trie is now empty.
  - Set trie pointer to null.
- p != header => remove node p and update pointer to p.



p Has No Self Pointer

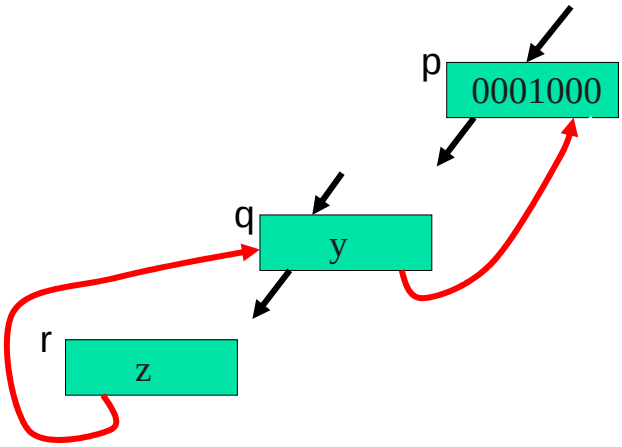
- Let q be the node that has a back pointer to p.
- Node q was determined during the search for the pair with the delete key k.



Blue pointer could be red or black.

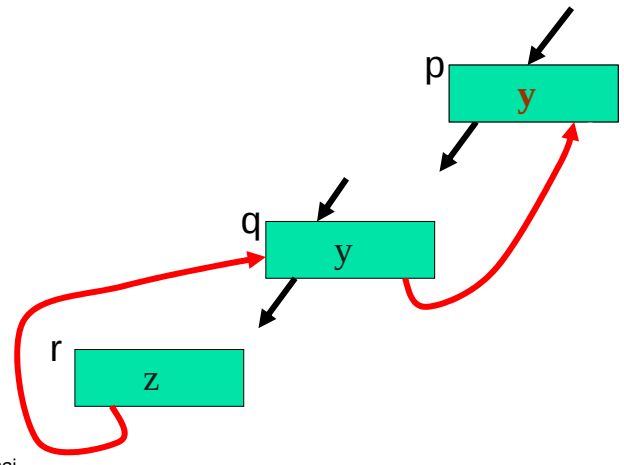
p Has No Self Pointer

- Use the key y in node q to find the unique node r that has a back pointer to node q.



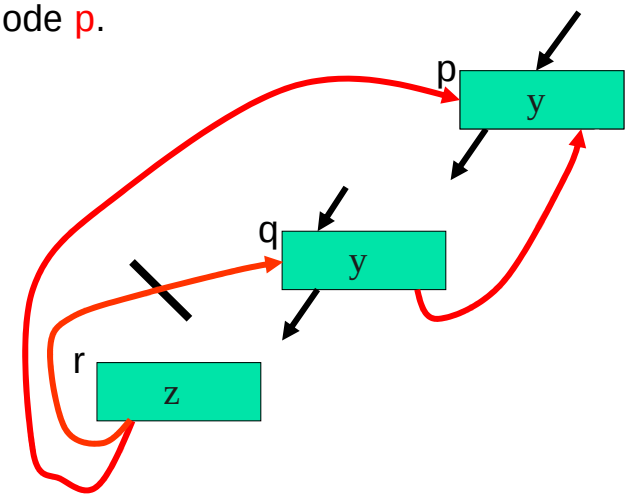
p Has No Self Pointer

- Copy the pair whose key is y to node p.



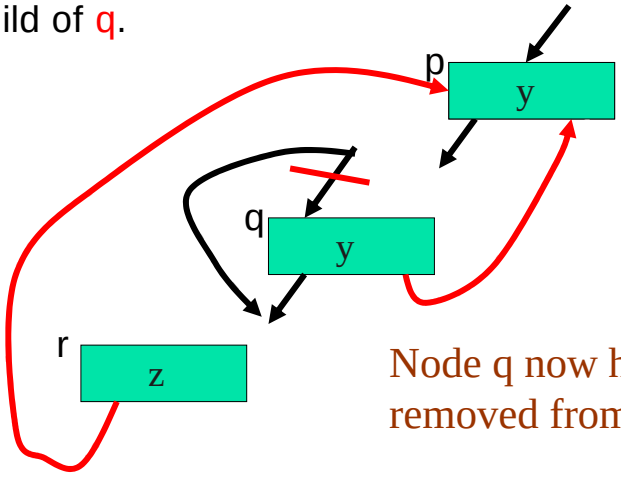
p Has No Self Pointer

- Change back pointer to q in node r to point to node p.



p Has No Self Pointer

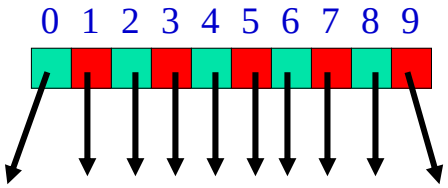
- Change forward pointer to q from parent(q) to child of q.



Node q now has been removed from trie.

Multiway Tries

- Key = Social Security Number.
  - 441-12-1135
  - 9 decimal digits.
- 10-way trie (order 10 trie).



Height <= 10.

Social Security Trie

- 10-way trie
  - Height <= 10.
  - Search: <= 9 branches on digits plus 1 compare.
- 100-way trie
  - 441-12-1135
  - Height <= 6.
  - Search: <= 5 branches on digits plus 1 compare.

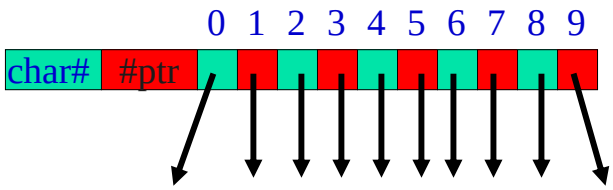
Social Security AVL & Red-Black

- Red-black tree
  - Height <= $2\log_2 10^9 \sim 60$.
  - Search: <= 60 compares of 9 digit numbers.
- AVL tree
  - Height <= $1.44\log_2 10^9 \sim 40$.
  - Search: <= 40 compares of 9 digit numbers.
- Best binary tree.
  - Height = $\log_2 10^9 \sim 30$.

Compressed Social Security Trie

- char# = character/digit used for branching.
  - Equivalent to bit# field of compressed binary trie.
- #ptr = # of nonnull pointers in the node.

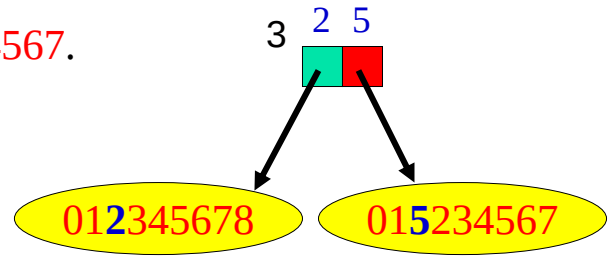
Branch Node Structure



Insert

- Insert 012345678.

- Insert 015234567.

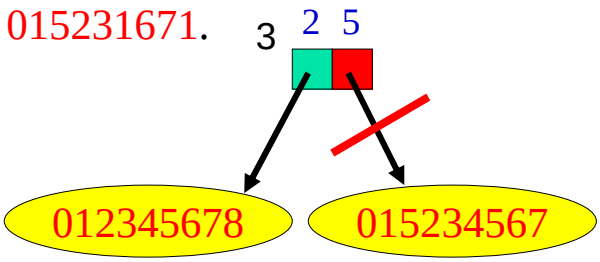


3: The 3rd digit is used for branching

Null pointer fields not shown.

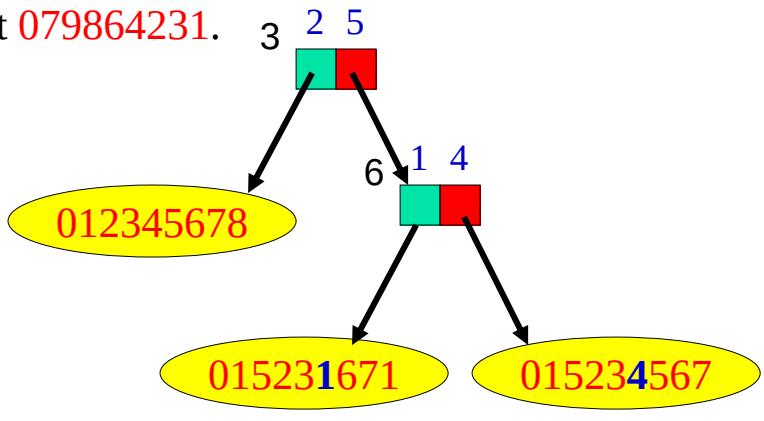
Insert

- Insert 015231671.



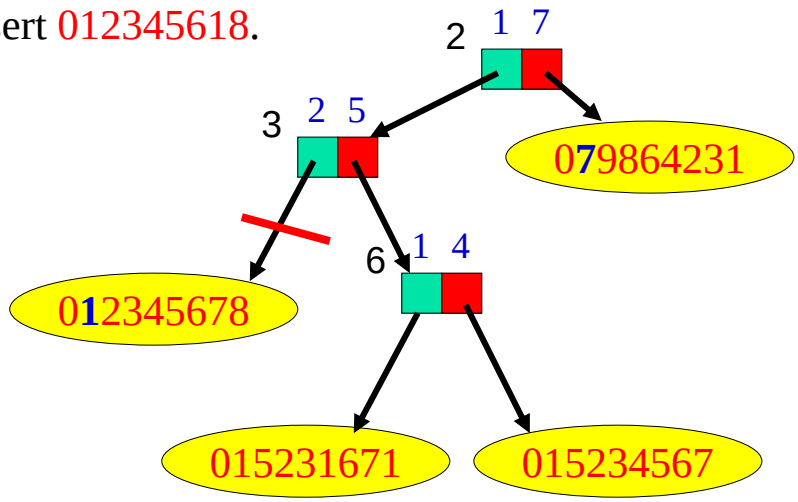
Insert

- Insert 079864231.



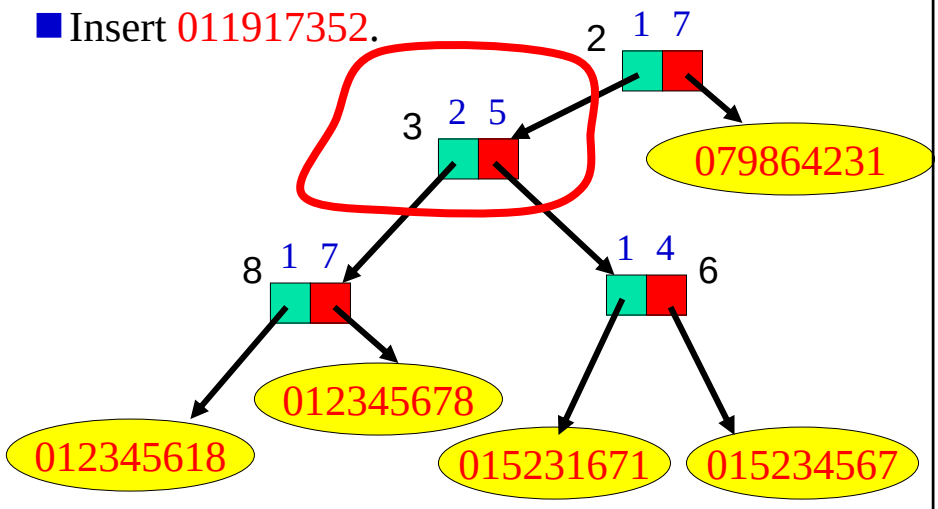
Insert

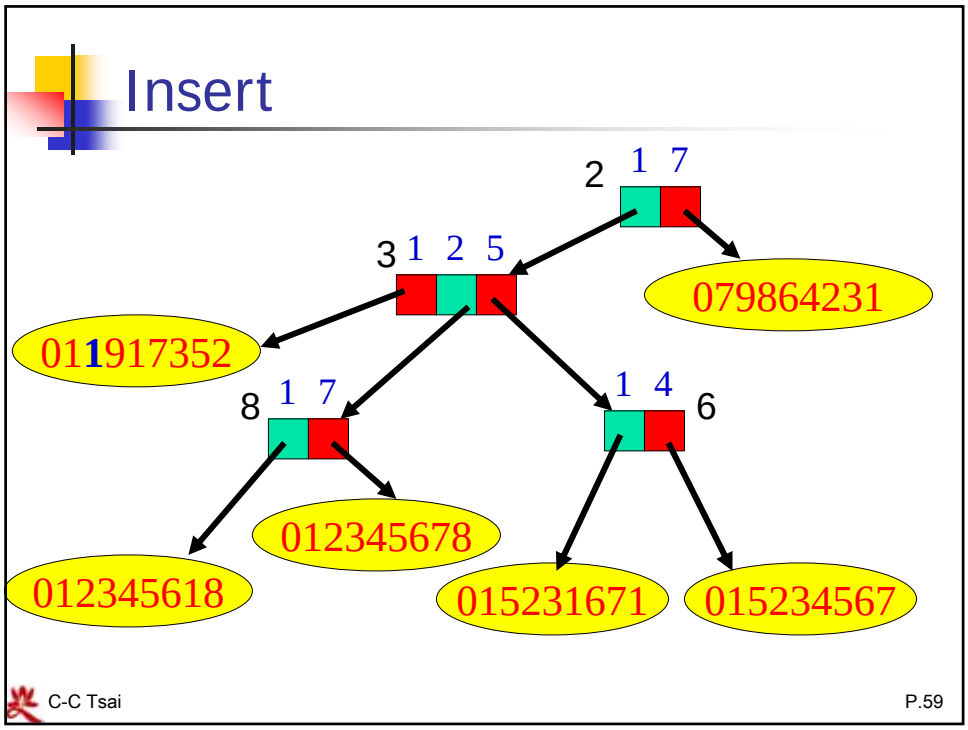
- Insert 012345618.

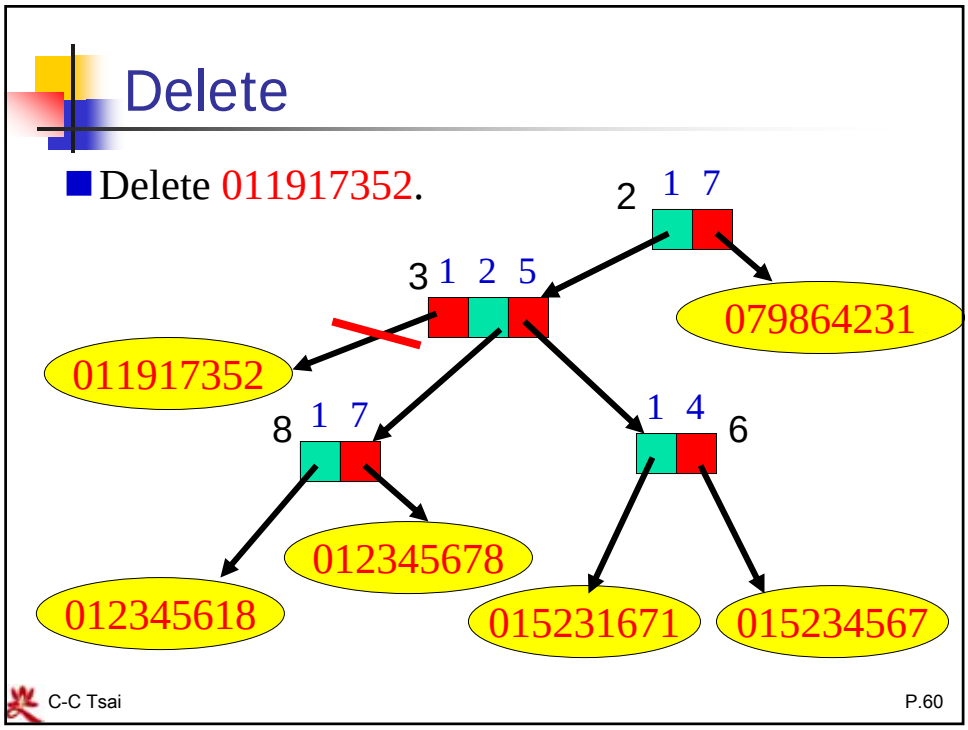


Insert

- Insert 011917352.

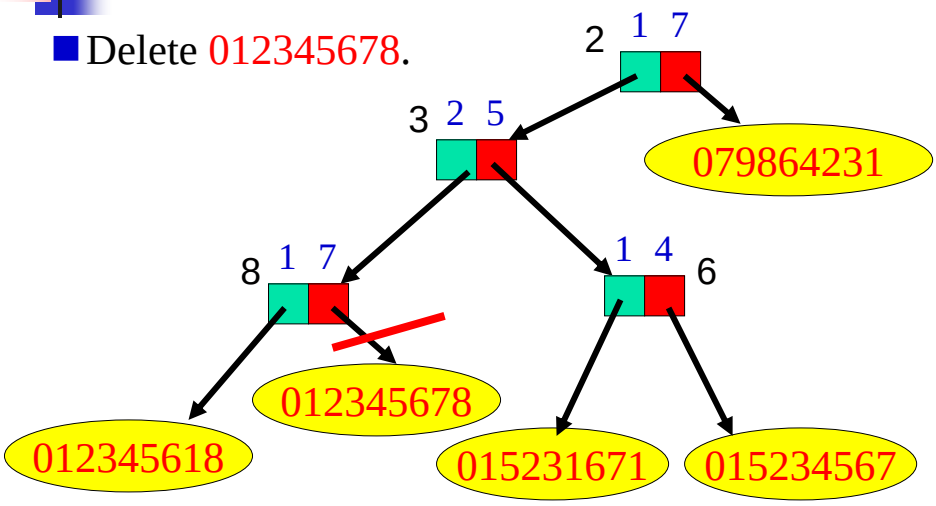



Insert
2 1 7
3 1 2 5
079864231
011917352
8 1 7
1 4 6
012345678
012345618
015231671
015234567
C-C Tsai
P.59


Delete
Delete 011917352.
2 1 7
3 1 2 5
079864231
011917352
8 1 7
1 4 6
012345678
012345618
015231671
015234567
C-C Tsai
P.60

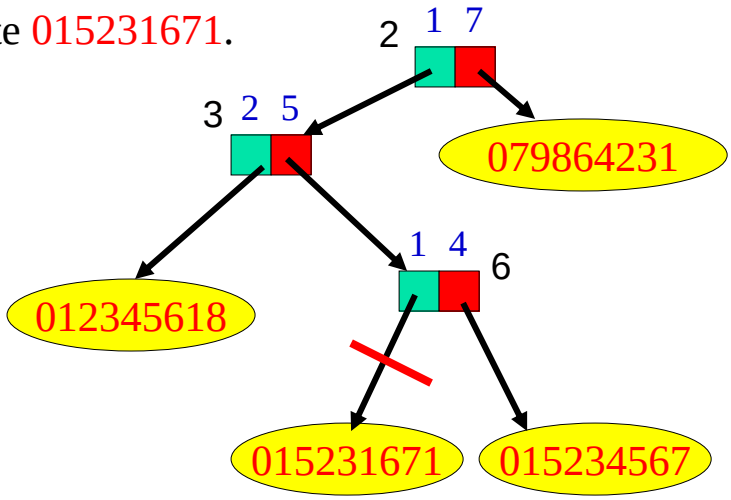
Delete

- Delete 012345678.

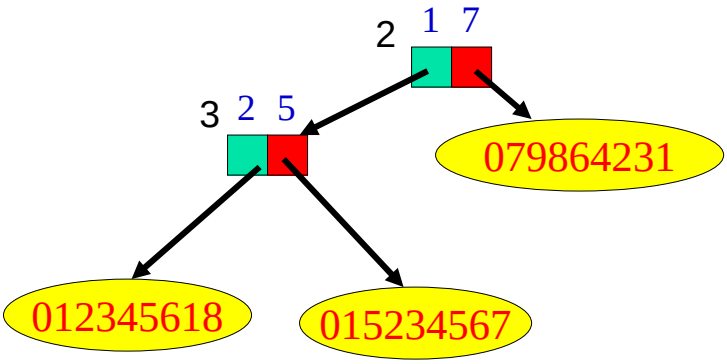


Delete

- Delete 015231671.

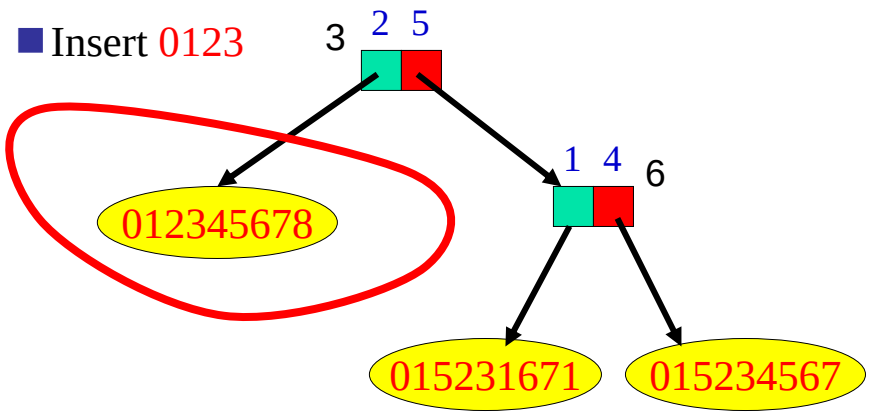


Delete



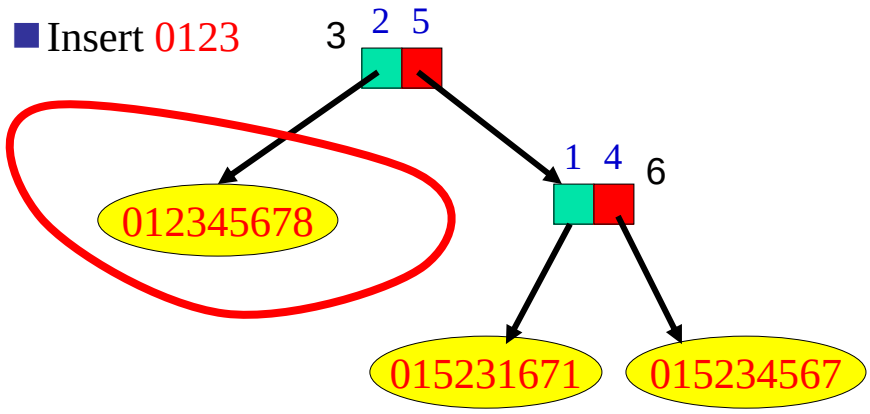
Variable Length Keys

- Problem arises only when one key is a (proper) prefix of another.
- Insert 0123



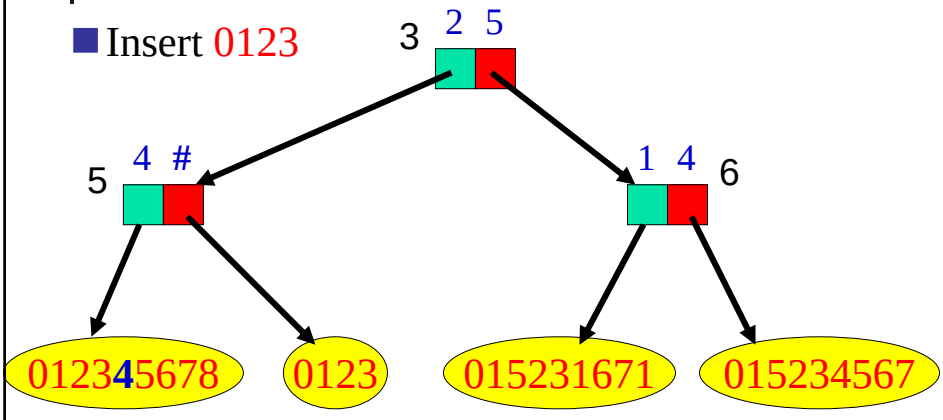
Variable Length Keys

- Add a special end of key character (#) to each key to eliminate this problem.
- Insert 0123



Variable Length Keys

- Insert 0123

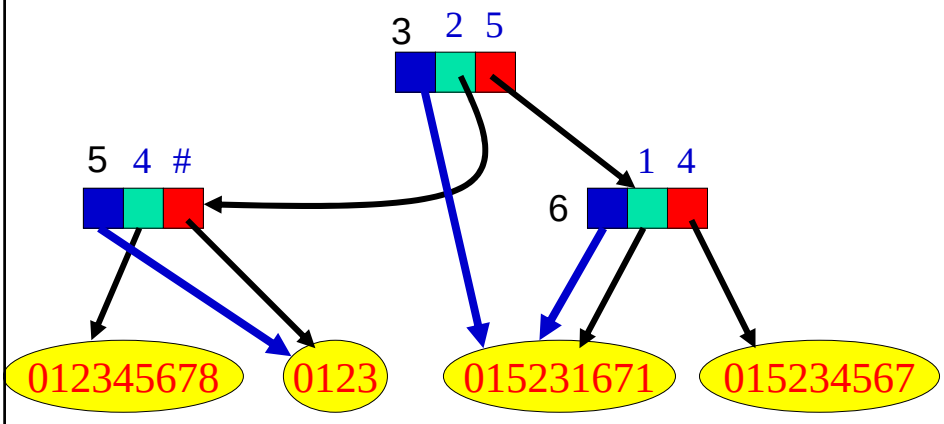


End of key character (#) not shown.

Tries With Edge Information

- Add a new field (element) to each branch node.
- New field points to any one of the element nodes in the subtree.
- Use this pointer on way down to figure out skipped-over characters.

Example



- element field shown in blue.

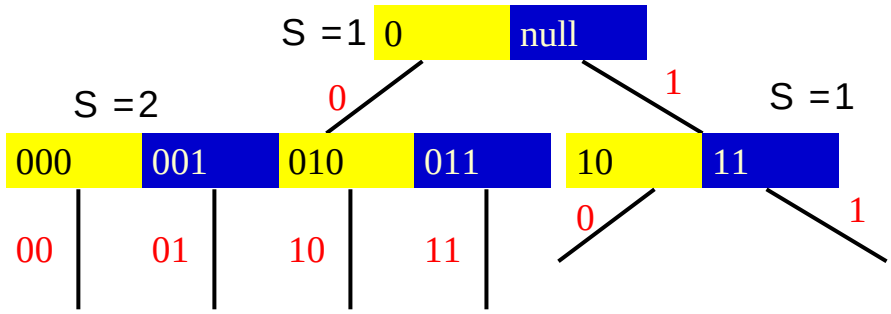
Trie Characteristics

- Expected height of an order m trie is ~$\log_m n$.
- Limit height to h (say 6). Level h branch nodes point to buckets that employ some other search structure for all keys in subtrie.
- Switch from trie scheme to simple array when number of pairs in subtrie becomes <= s (say s=6).
  - Expected # of branch nodes for an order m trie when n is large and m and s are small is n/(s ln m).
- Sample digits from right to left (instead of from left to right) or using a pseudorandom number generator so as to reduce trie height.

Multibit Tries

- Variant of binary trie in which the number of bits (stride) used for branching may vary from node to node.
- Proposed for Internet router applications.
  - Variable length prefixes.
  - Longest prefix match.
- Limit height by choosing node strides.
  - Root stride = 32 => height = 1.
  - Strides of 16, 8, and 8 for levels 1, 2, and 3 => only 3 levels.

Multibit Trie Example



Multibit Tries

- Node whose stride is s uses s bits for branching.
  - Node has 2^s children and 2^s element/prefix fields.
  - Prefixes that end at a node are stored in that node.
  - Short prefixes are expanded to length represented by node.
  - When root stride is 3, prefixes whose length is < 3 are expanded to length 3.
  - P = 00* expands to P0 = 000* and P1 = 001*.
  - If Q = 000* already exists P0 is eliminated because Q represents a longer match for any destination.