

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

→ In general the DST can be 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 $\Rightarrow O(1)$

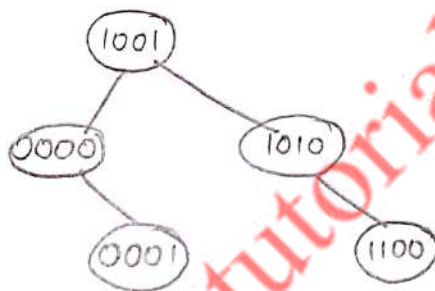
Average case $\Rightarrow O(\log N)$

'N' is height of DST

Worst case $\Rightarrow O(b)$

'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

Step ① :- check whether the tree is empty or not

Step ② :- If the tree is empty then new element is inserted into the DST and which will be treated as root node

Step ③ :-
→ If the tree is not empty then the new element is inserted either left subtree (or) right subtree to the root node 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 ③:- 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 Bit by Bit with the new element.

④ this procedure will be continued at every node in the BST recursively.

Ex:- construct a BST 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

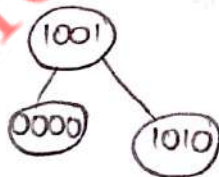
Insert 1001



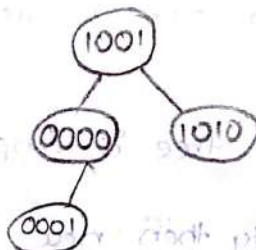
Insert 0000



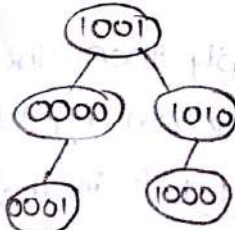
Insert 1010



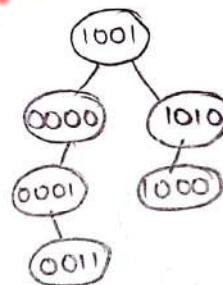
Insert 0001



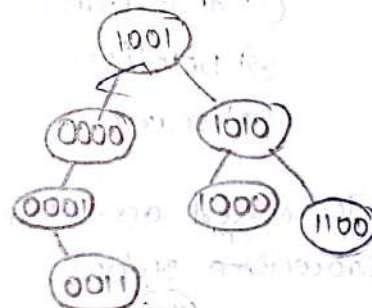
Insert 1000



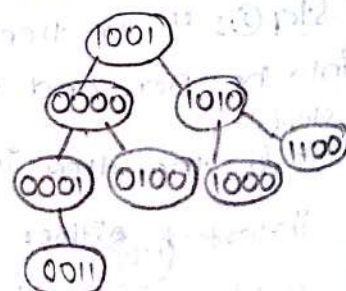
Insert 0011



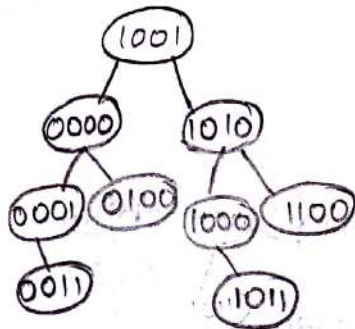
Insert 1100



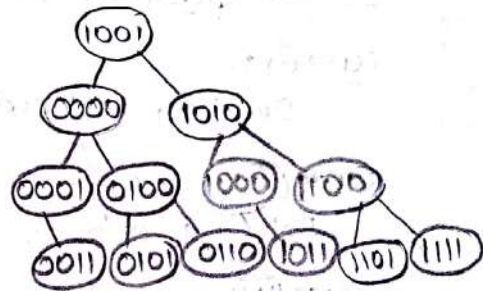
Insert 0100



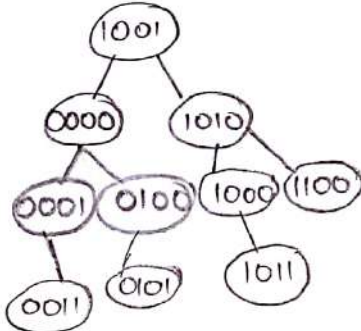
Insert 1011 :-



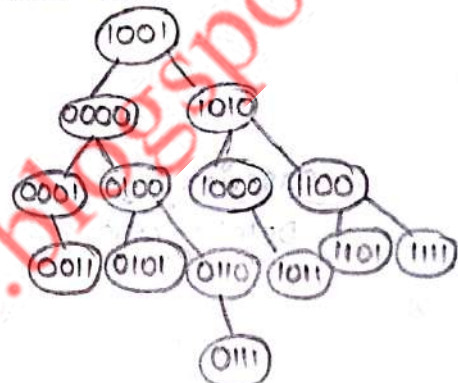
Insert 1111 :-



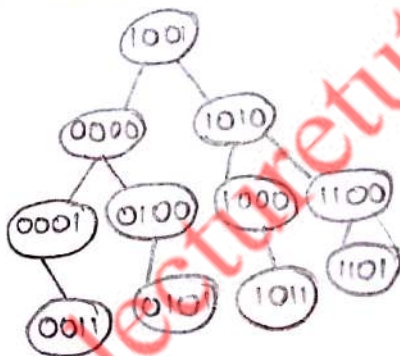
Insert 0101 :-



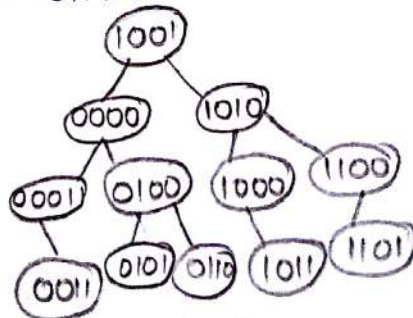
Insert 0111



Insert 1101 :-



Insert 0110 :-



Deletion operation in DST :-

case (i) :-

Deleting leaf node

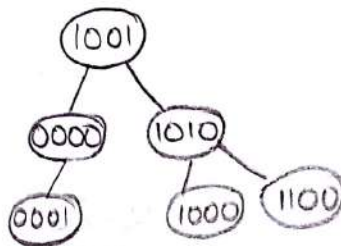
case (ii)

Deleting a node which is having one child

case (iii)

Deleting a node having 2 children

Ex :- Delete the elements from DST



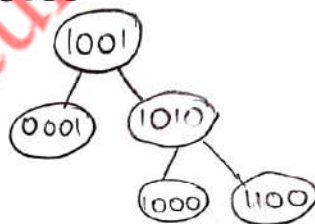
case (i) Eg :- ↓

Delete 0001



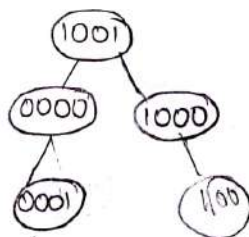
case (ii) Eg :- ↓

Delete 0000



case (iii) Eg :- ↓

Delete 1010



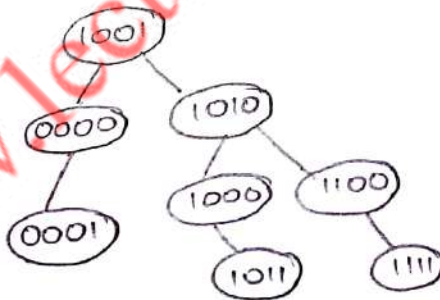
Search operation in DST

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 will be performed recursively on the left subtree of the rootnode. If it is one, the search operation will be performed recursively on the right subtree of the rootnode
- 6) This process will be continued until the leafnode of either left subtree (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

C-C Tsai

P.1

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.

C-C Tsai

P.2

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**.

```
graph TD; A(0110) --> B(0010); B --> C(1001);
```

C-C Tsai P.3

Example

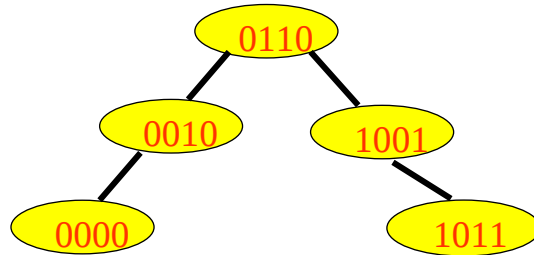
- Now, **insert a pair whose key is 1011**.
- Now, **insert a pair whose key is 0000**.

```
graph TD; A(0110) --> B(0010); A --> C(1001); B --> D(0000); C --> E(1011);
```

C-C Tsai P.4

Search/Insert/Delete

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



C-C Tsai

P.5

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.

C-C Tsai

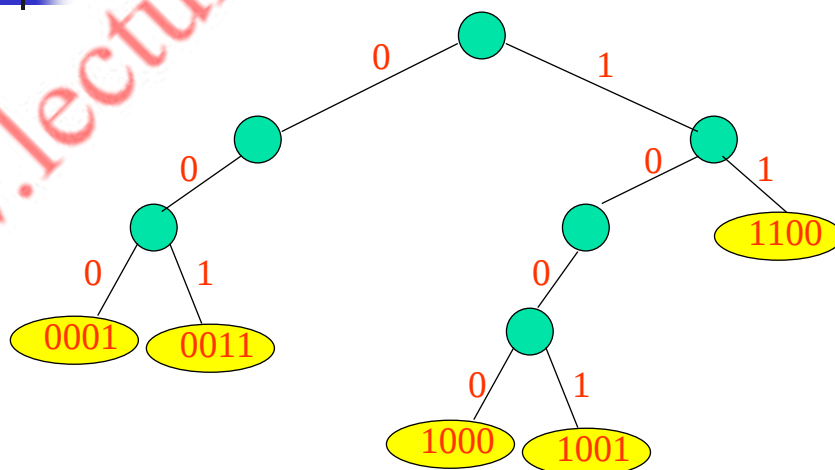
Binary Trie

- 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.

C-C Tsai

P.7

Example of Binary Trie



At most one key comparison for a search.

C-C Tsai

P.8

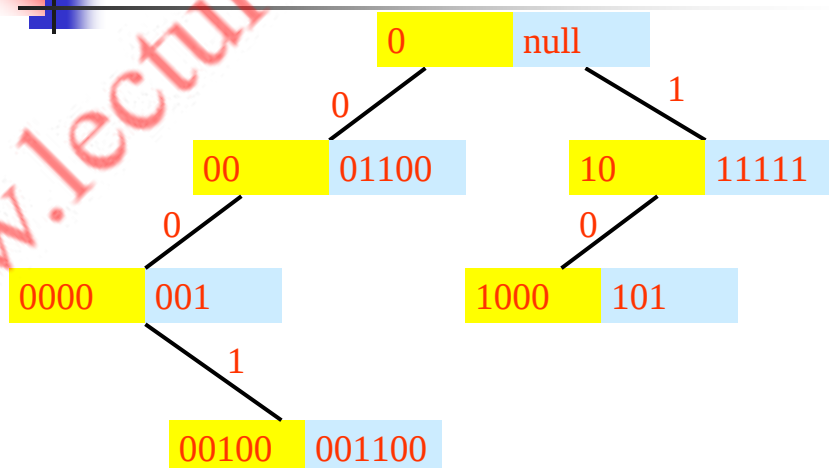
Variable Key Length

- 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.

C-C Tsai

P.9

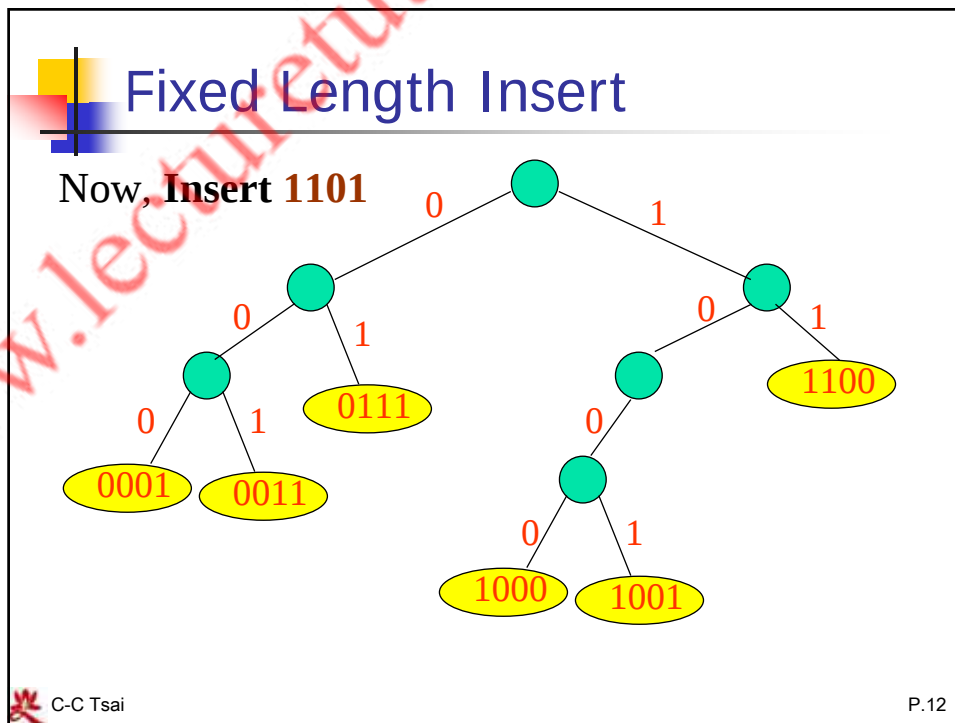
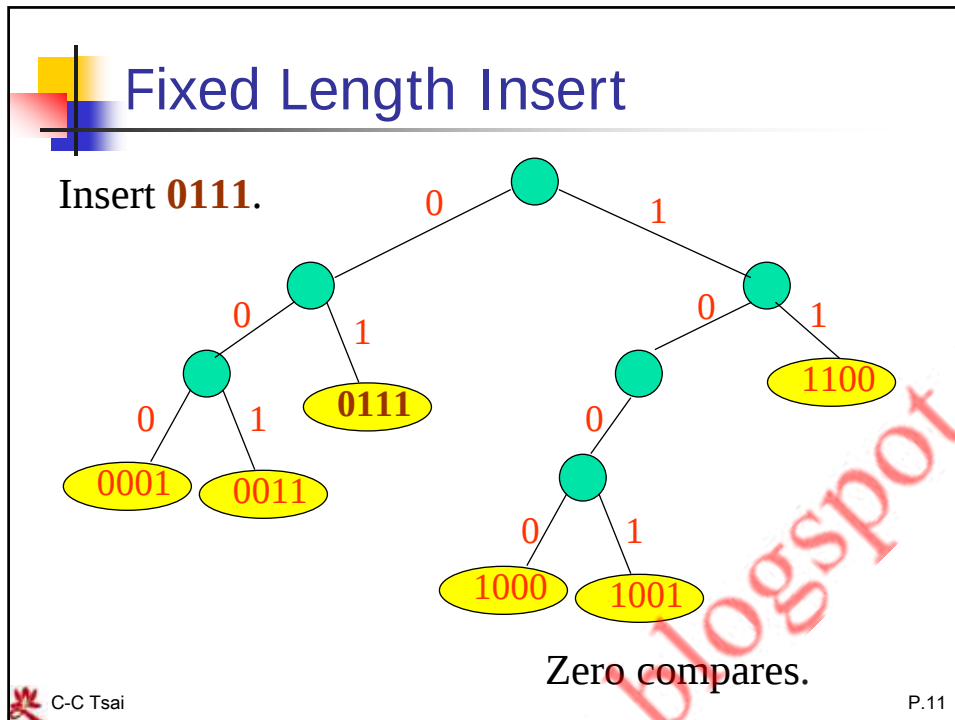
Example of Variable Key Length

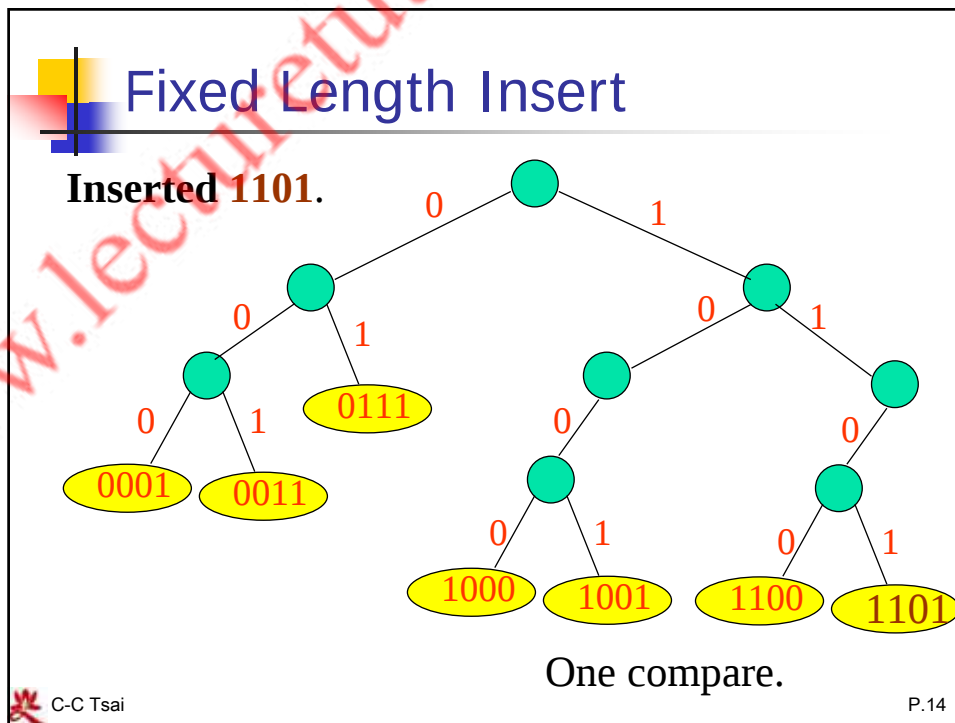
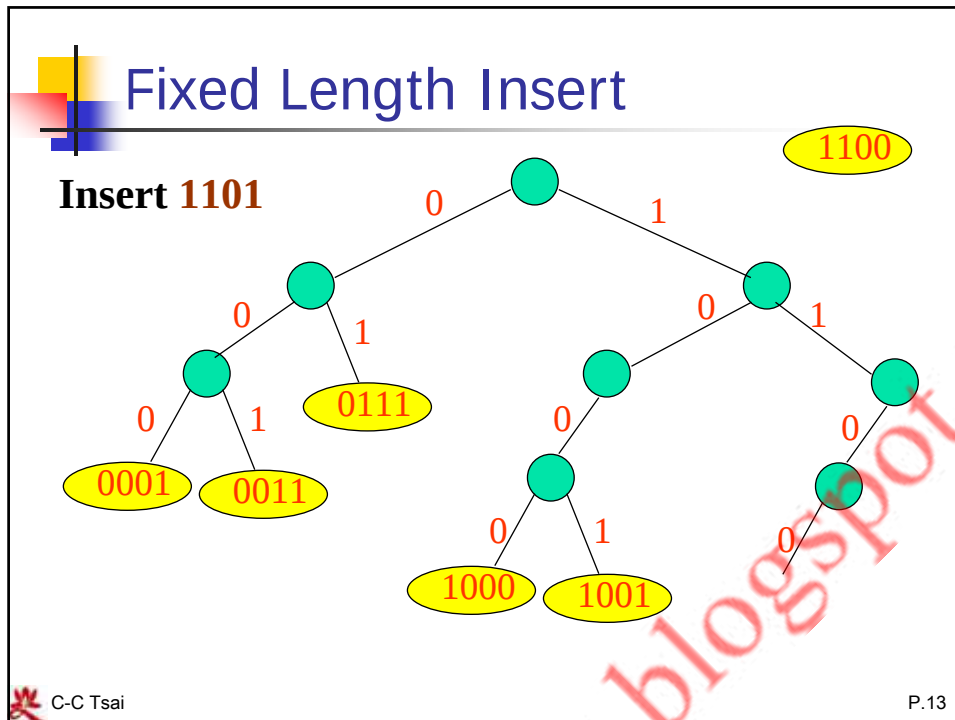


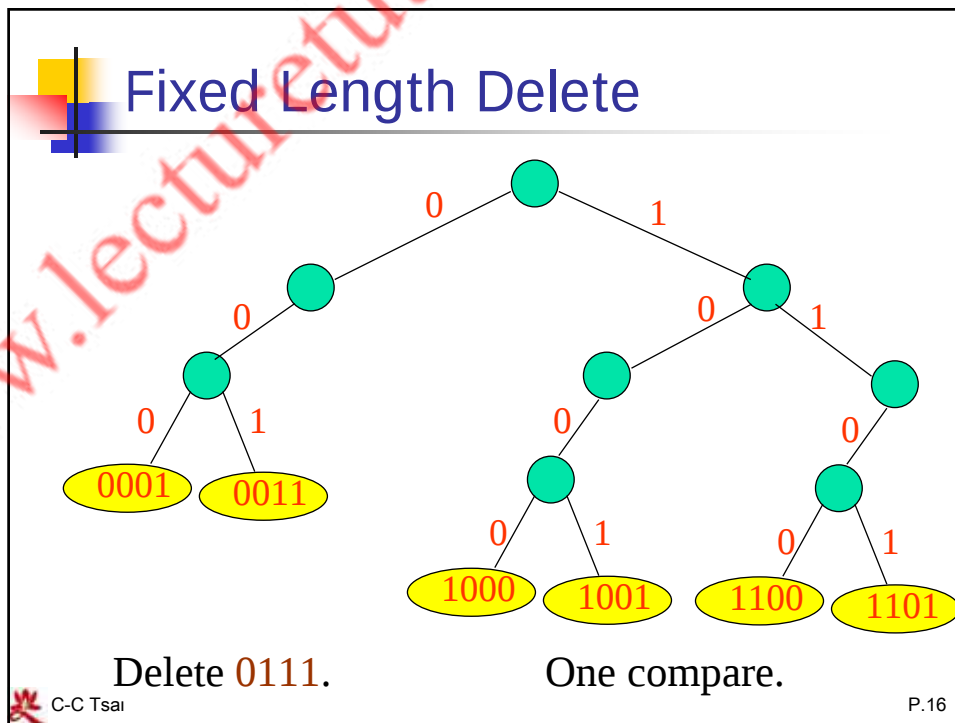
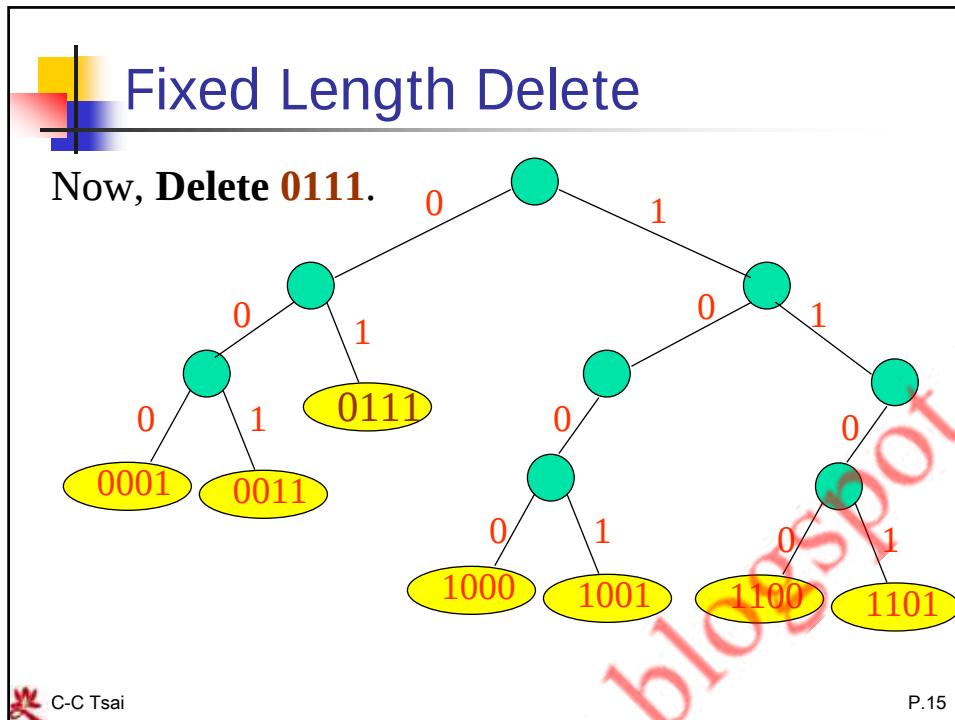
At most one key comparison for a search.

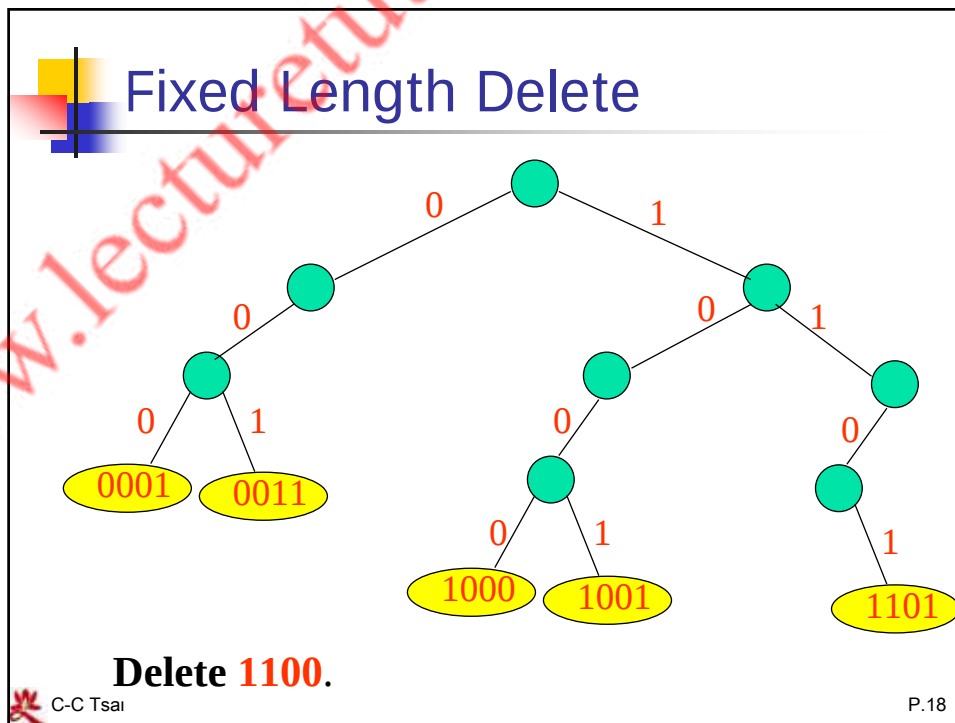
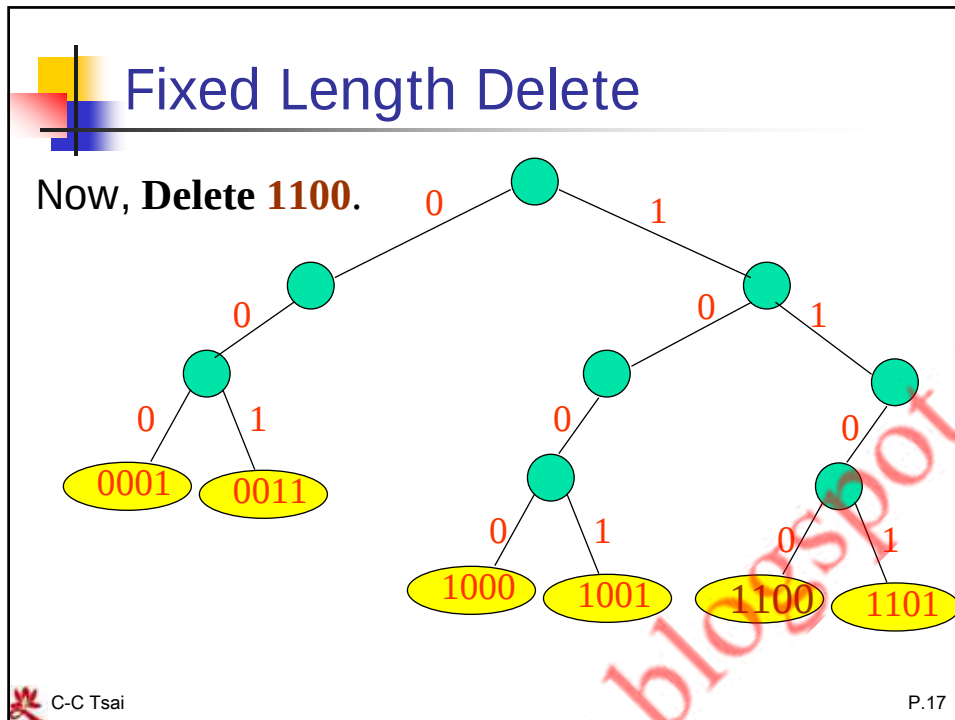
C-C Tsai

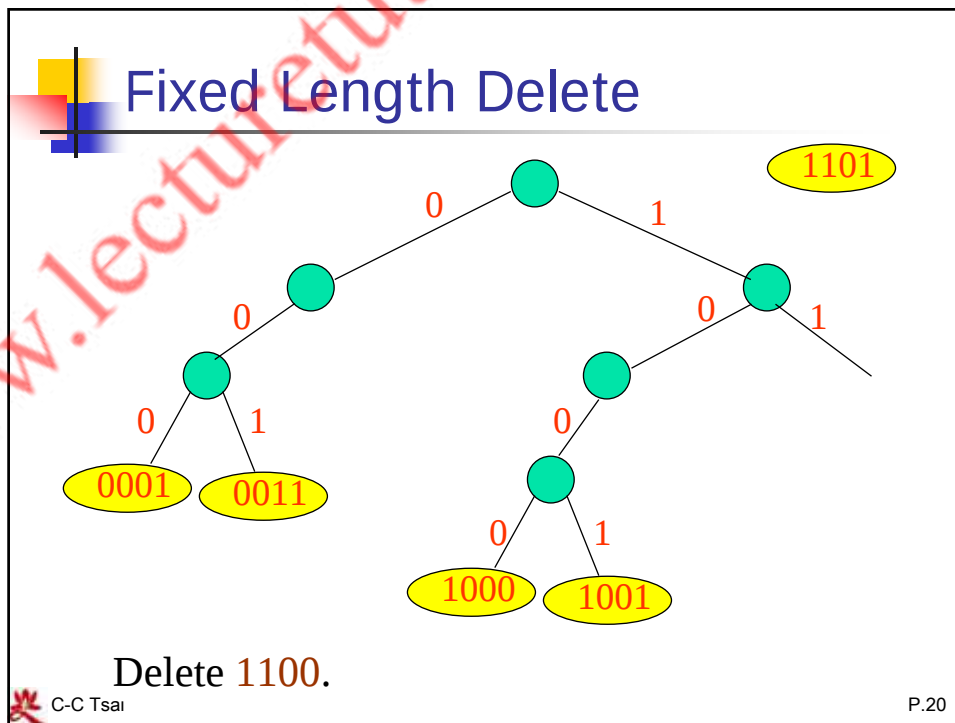
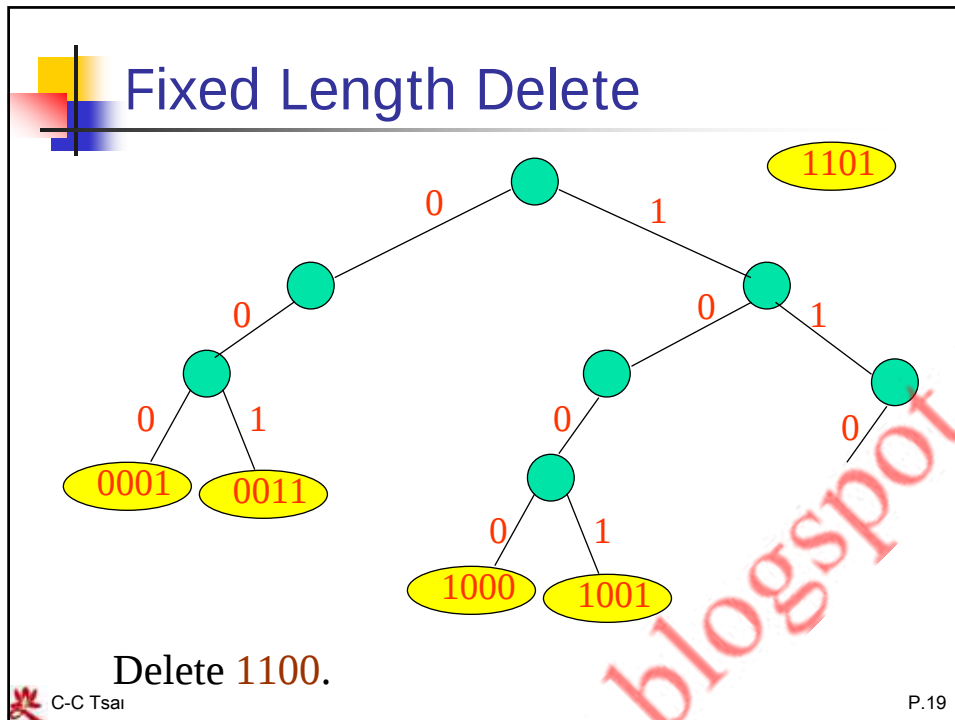
P.10

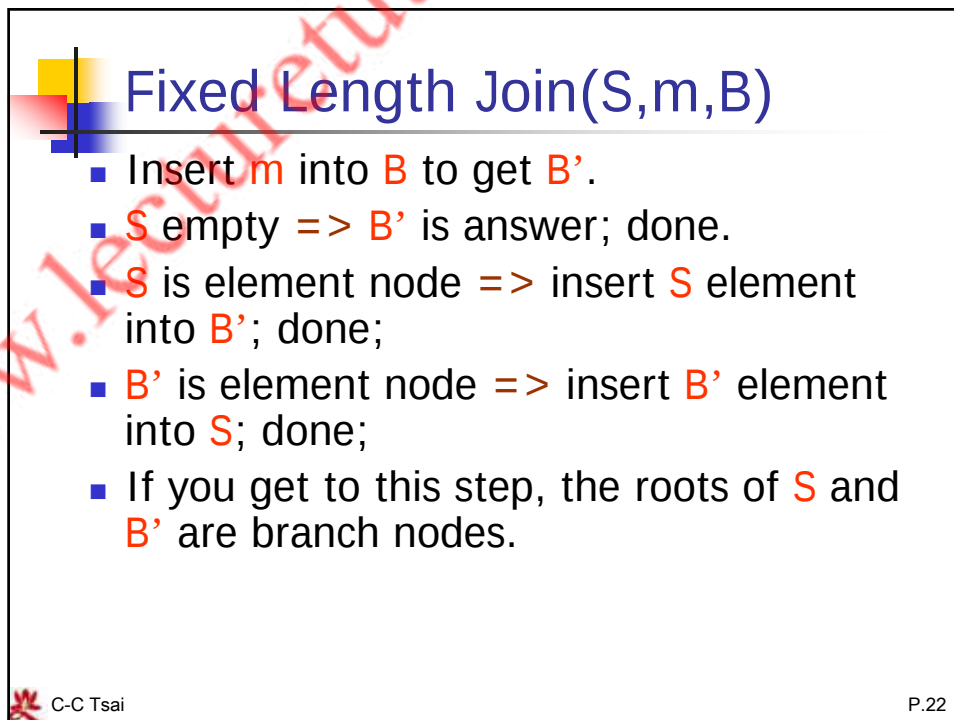
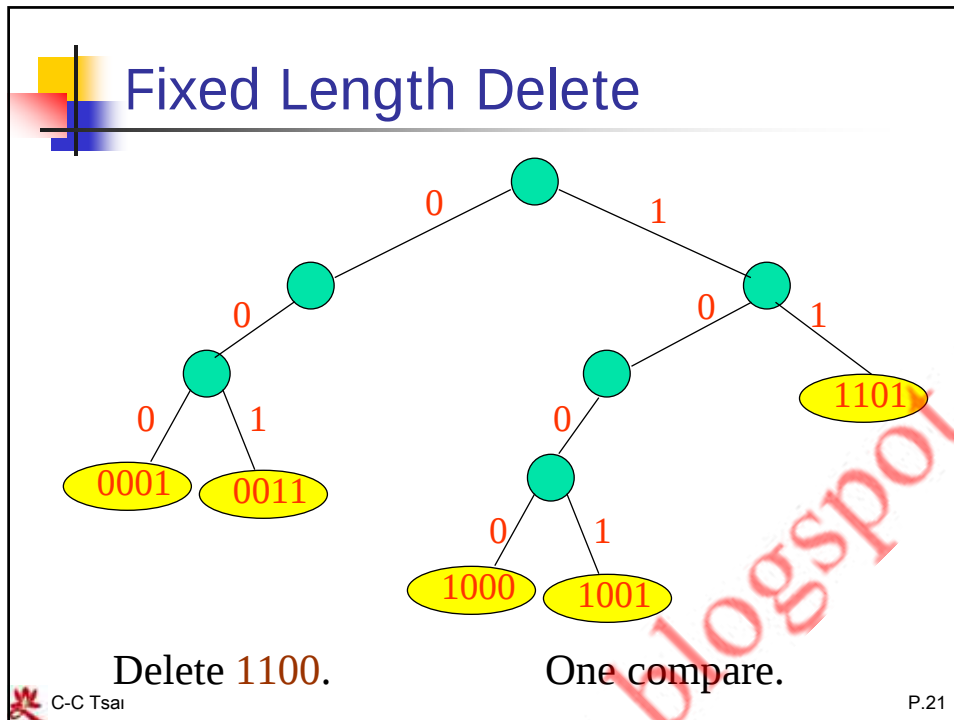




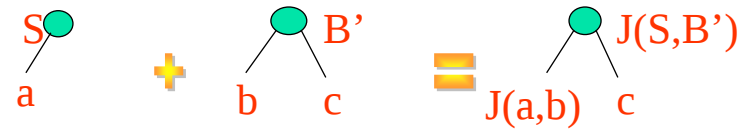


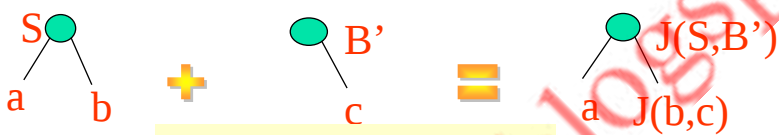






Fixed Length Join(S,m,B)

- S has empty right subtree.
 

$J(X,Y) \Leftrightarrow \text{join } X \text{ and } Y, \text{ all keys in } X < \text{all in } Y.$
- S has nonempty right subtree.
 - Left subtree of B' must be empty, because all keys in B' > all keys in S.

Complexity = $O(\text{height})$.

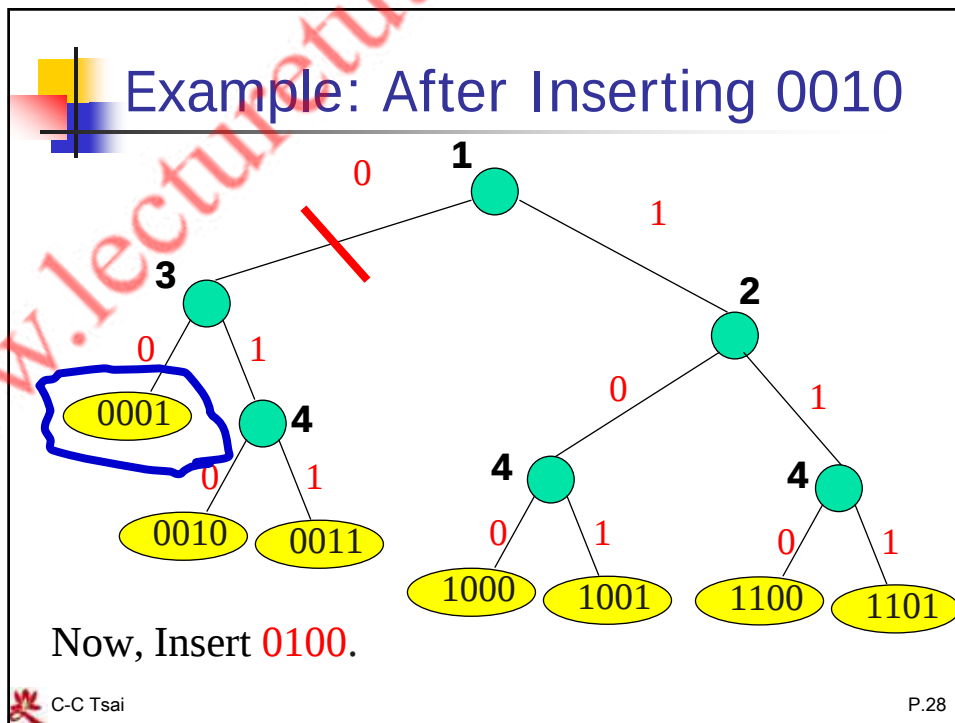
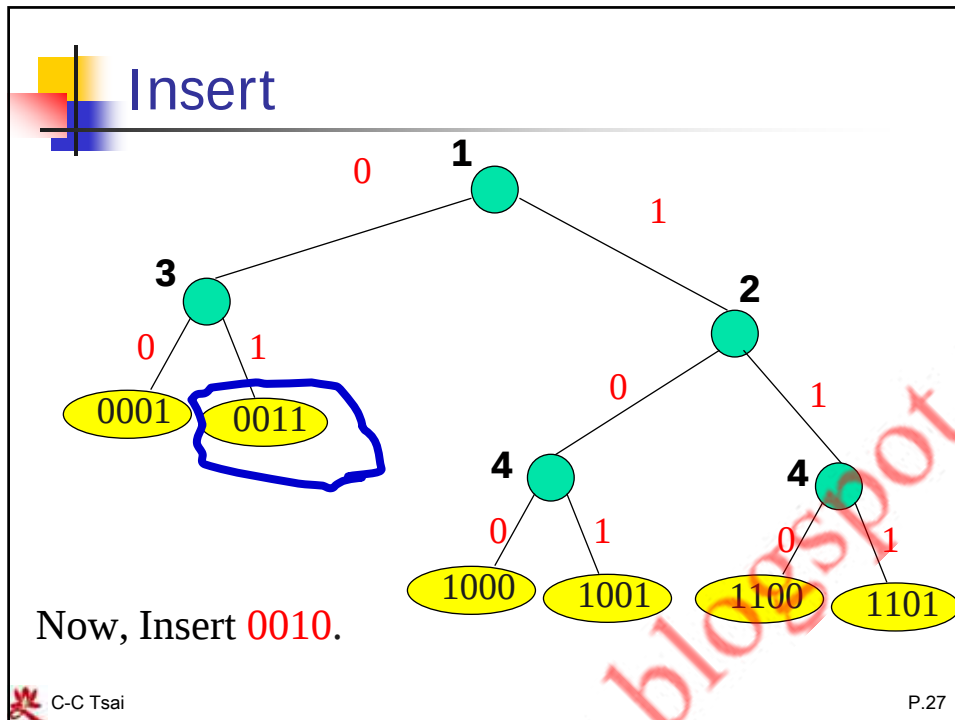
C-C Tsai P.23

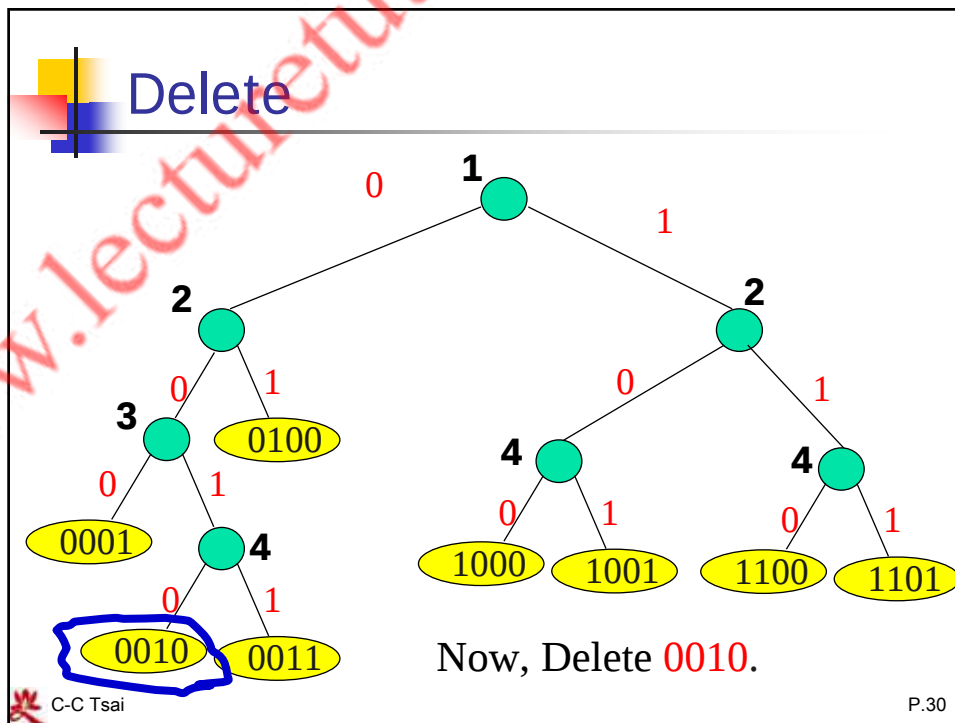
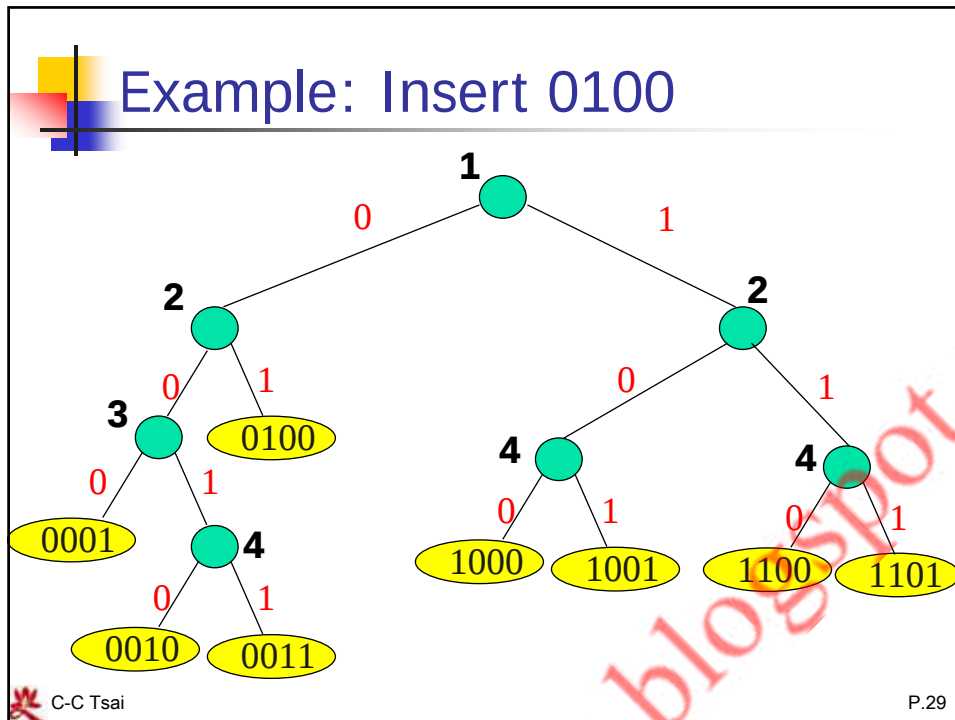
Compressed Binary Tries

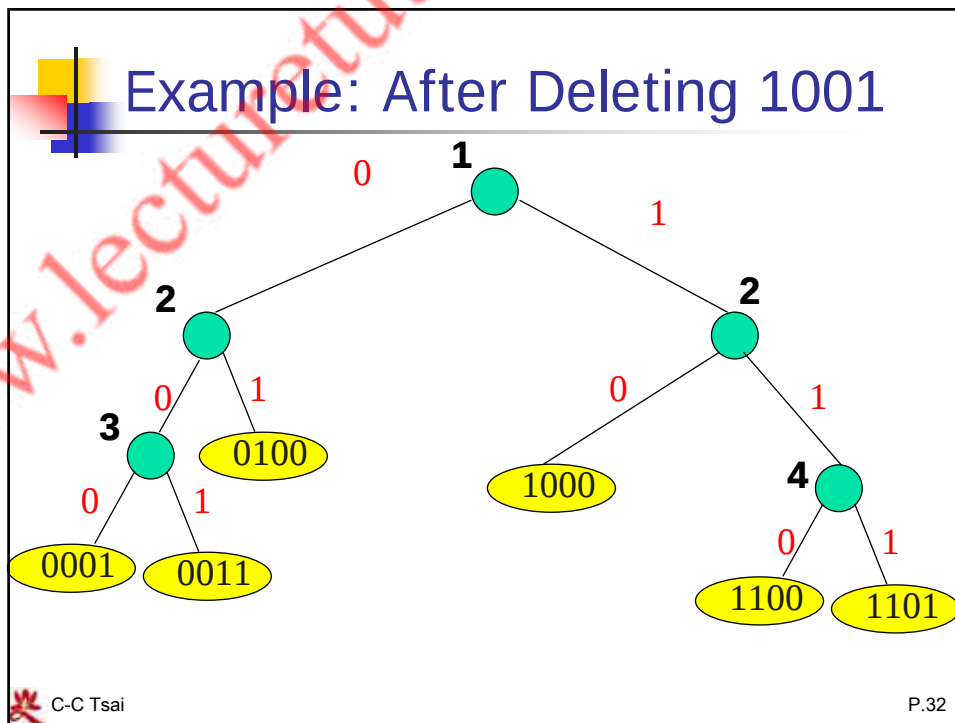
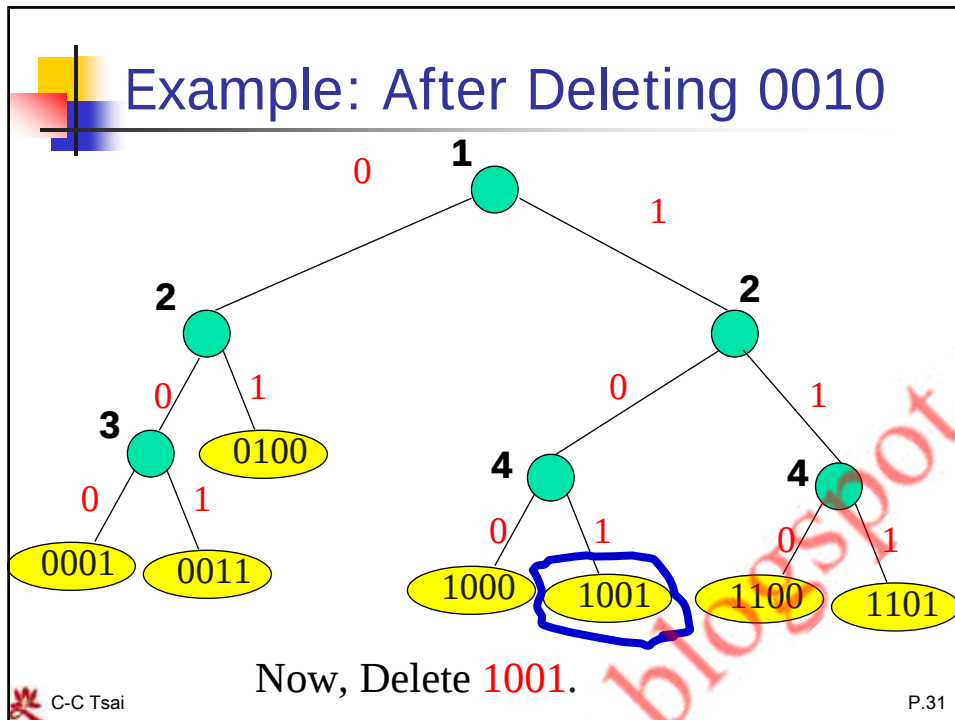
- No branch node whose degree is 1.
- Add a bit# field to each branch node.
- bit# tells you which bit of the key to use to decide whether to move to the left or right subtrie.

C-C Tsai P.24











Patricia

- **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric.
- 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.

C-C Tsai



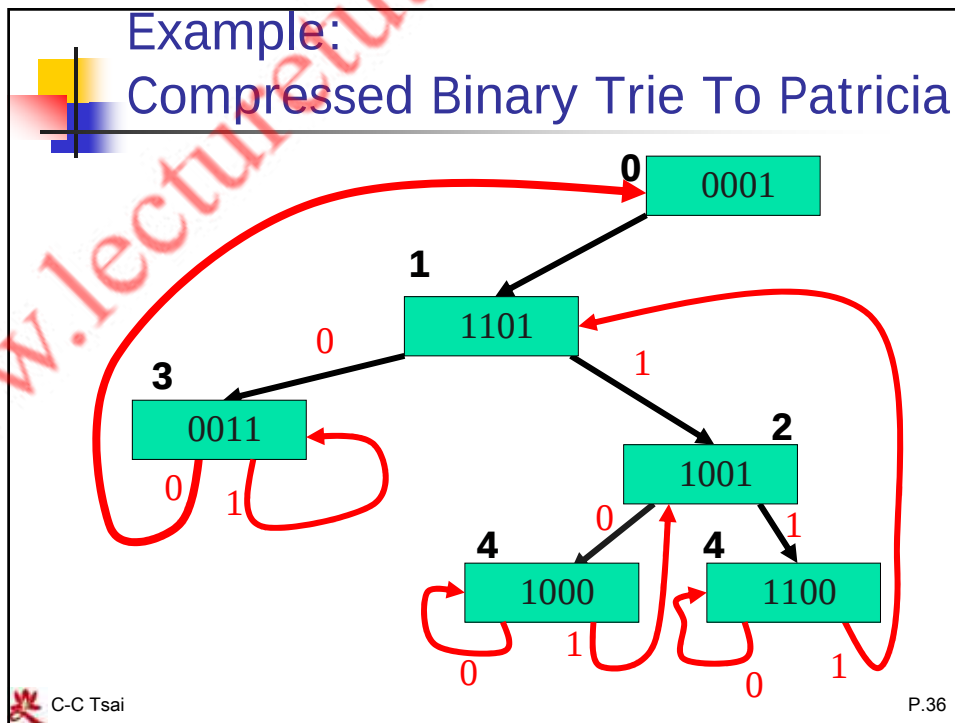
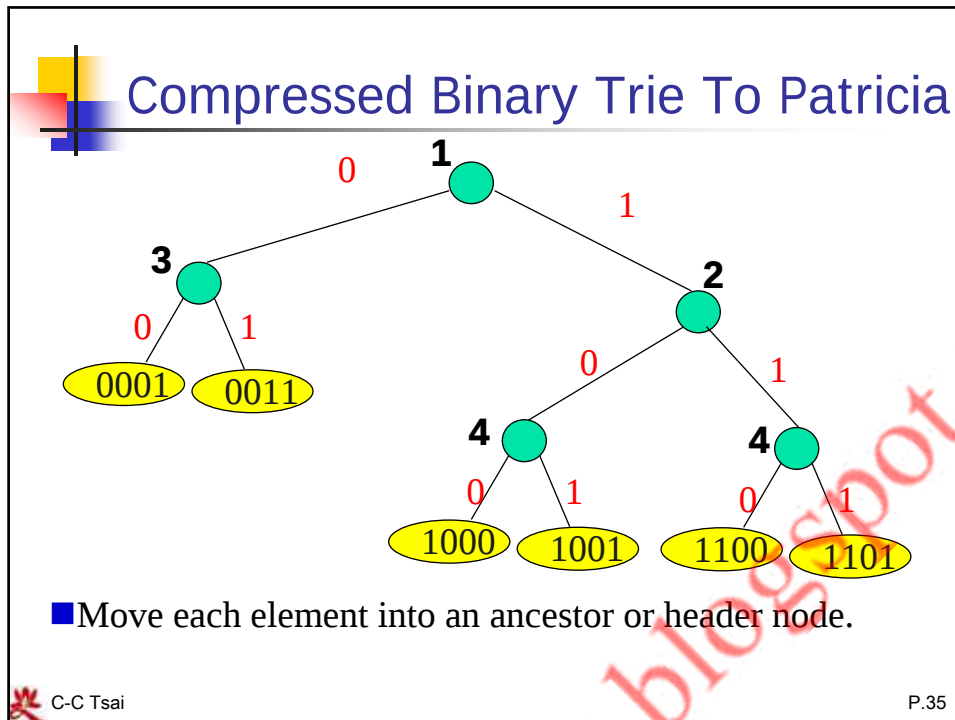
Node Structure

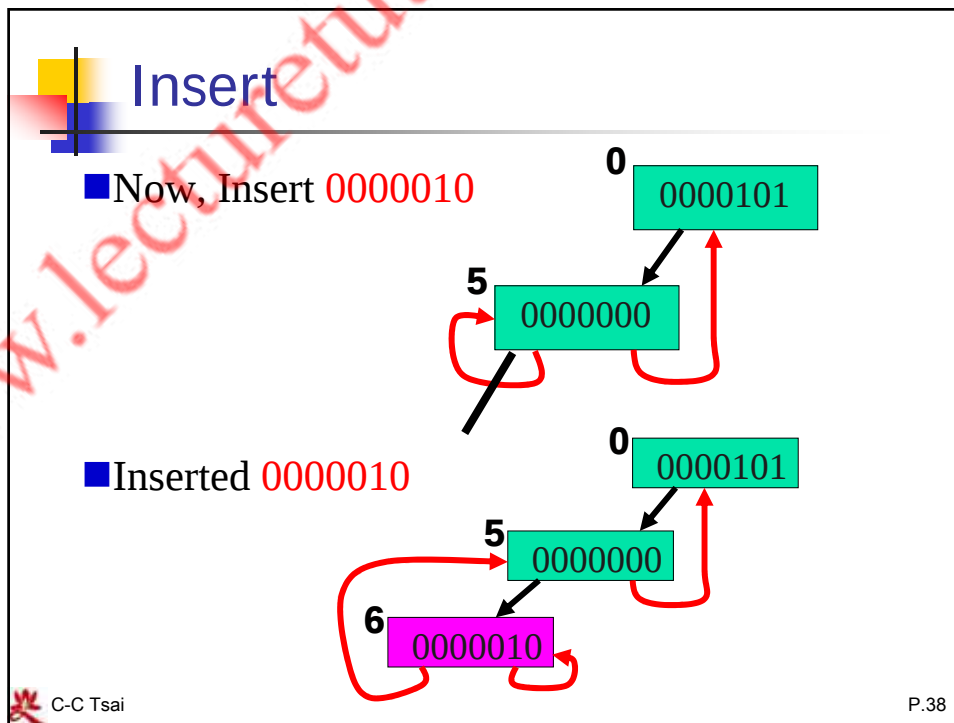
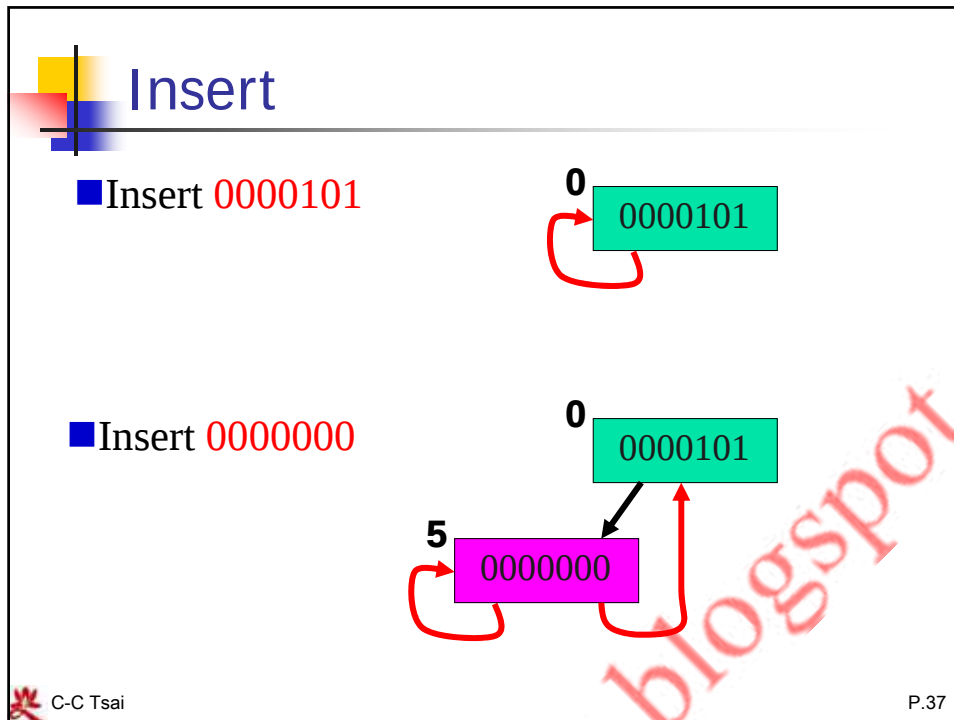
bit#	LC	Pair	RC
------	----	------	----

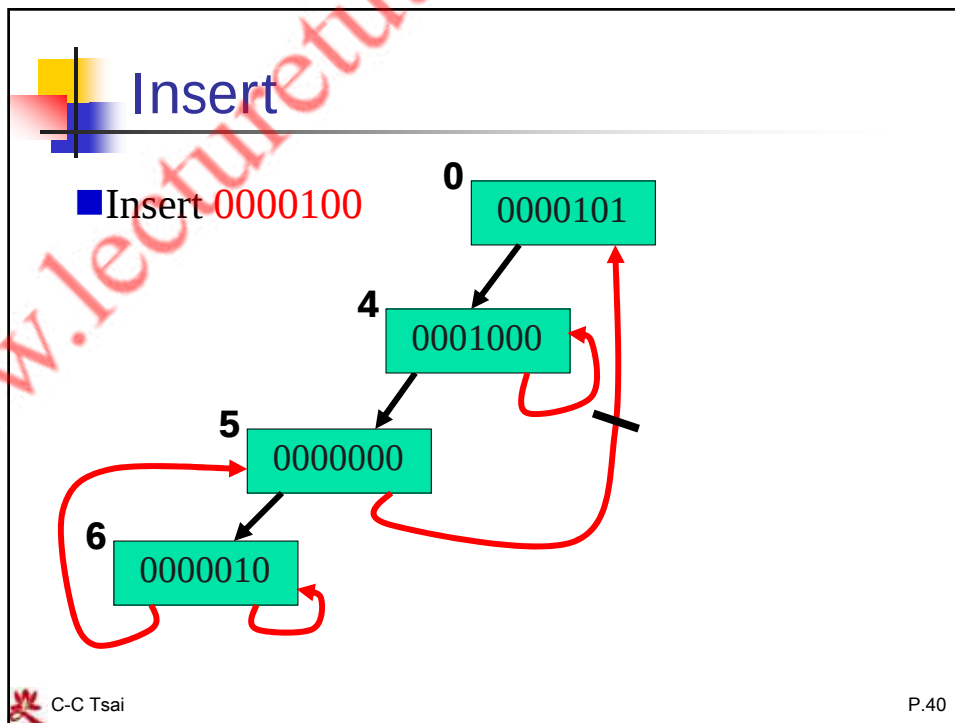
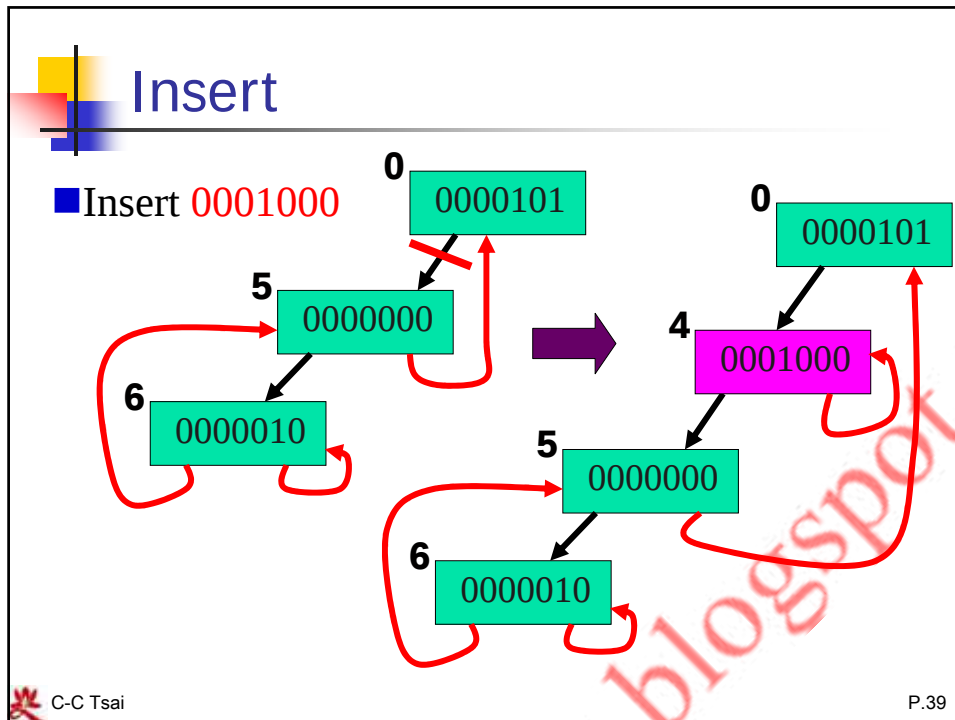
- **bit#** = bit used for branching
- **LC** = left child pointer
- **Pair** = dictionary pair
- **RC** = right child pointer

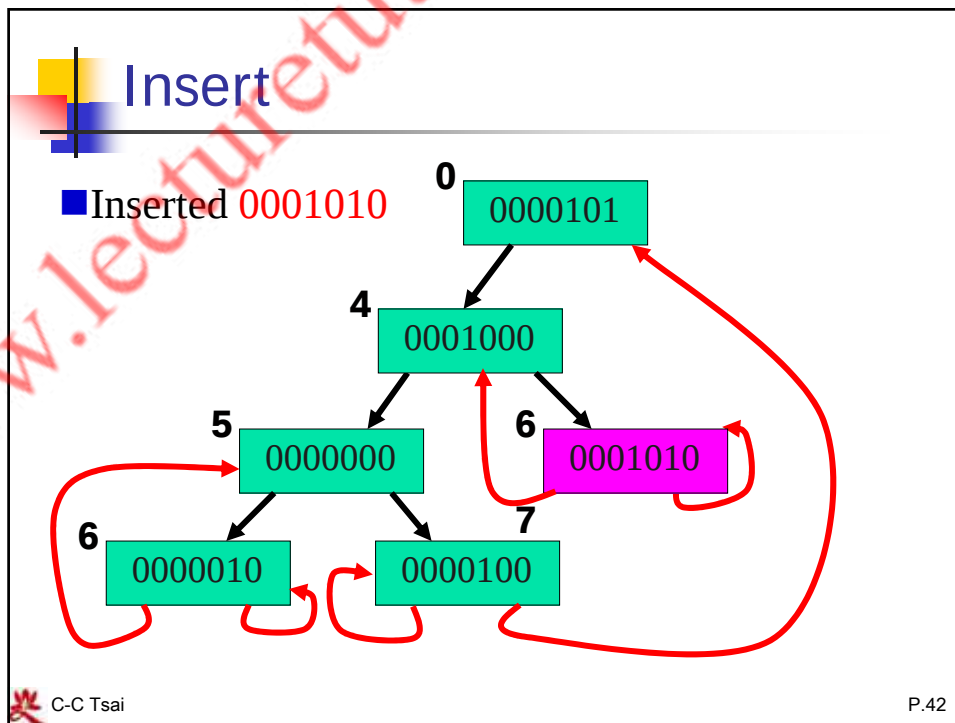
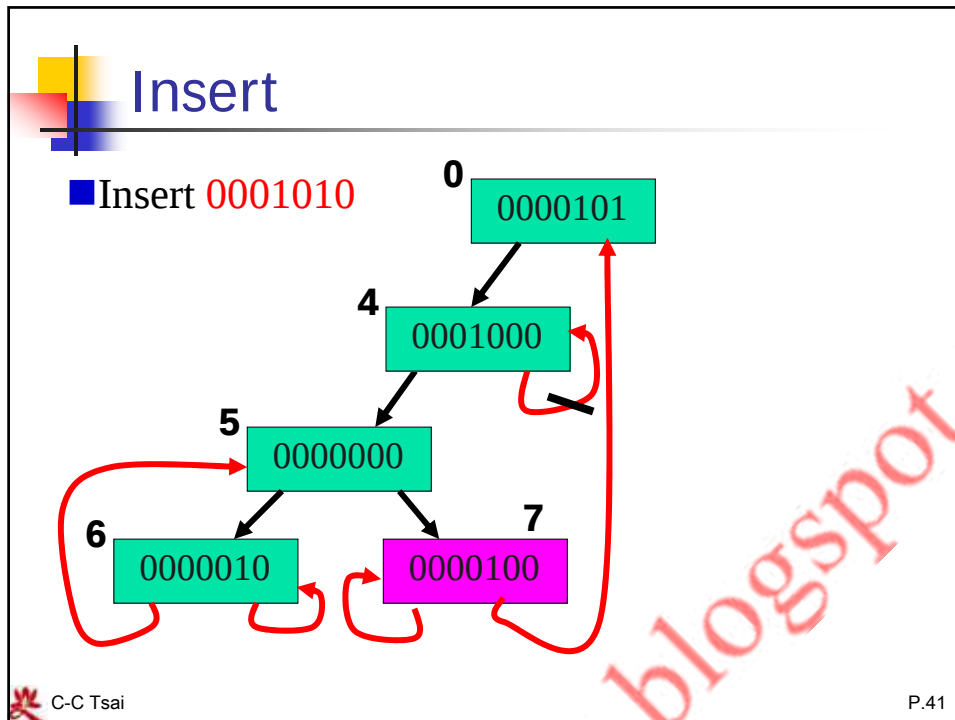
C-C Tsai

P.34









Delete

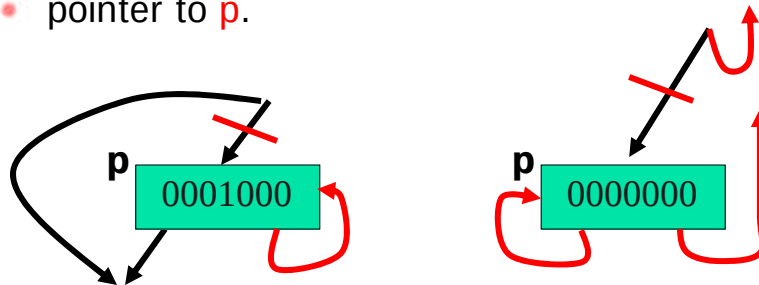
- Let **p** be the node that contains the dictionary pair that is to be deleted.
- Case 1: **p** has one self pointer.
- Case 2: **p** has no self pointer.

C-C Tsai

P.43

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**.



C-C Tsai

P.44

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**.

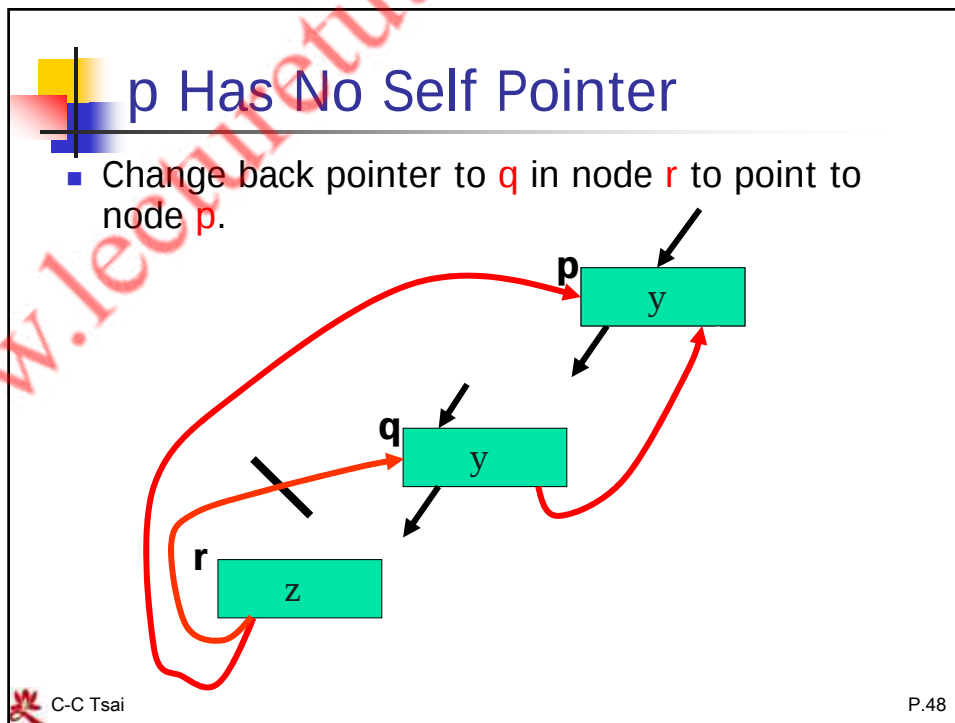
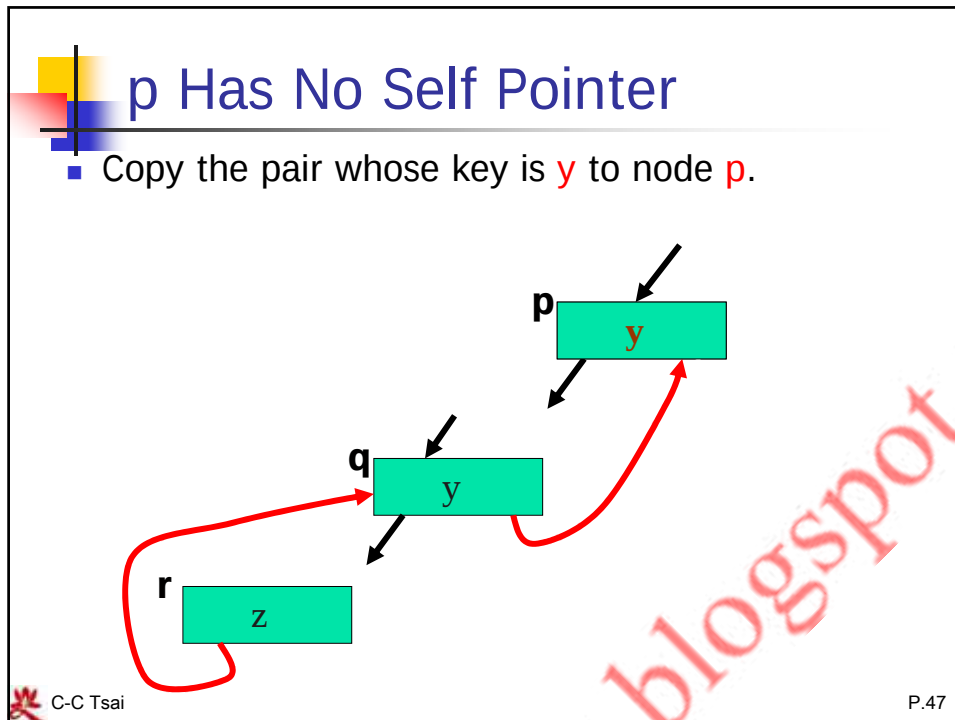
Blue pointer could be red or black.

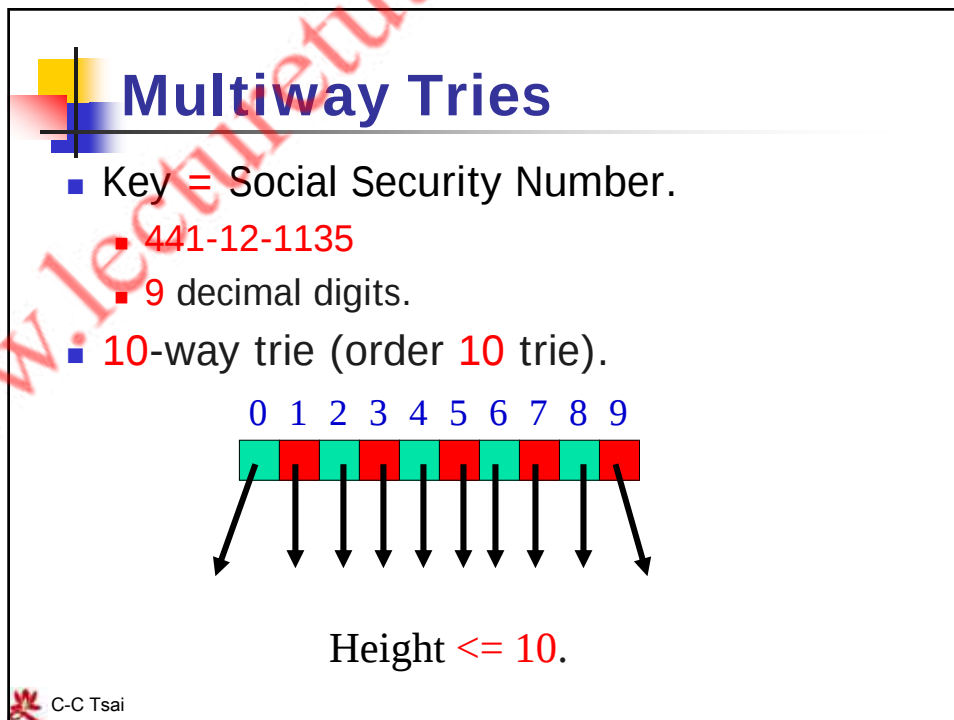
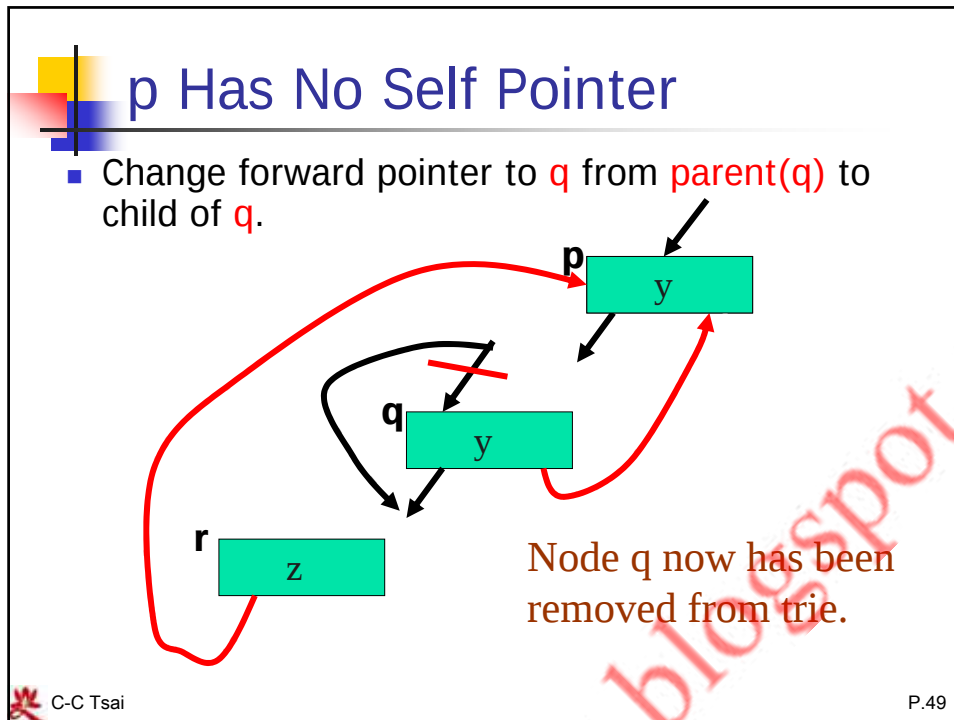
C-C Tsai P.45

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**.

C-C Tsai P.46





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.

C-C Tsai

P.51

Social Security AVL & Red-Black

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

C-C Tsai

P.52

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

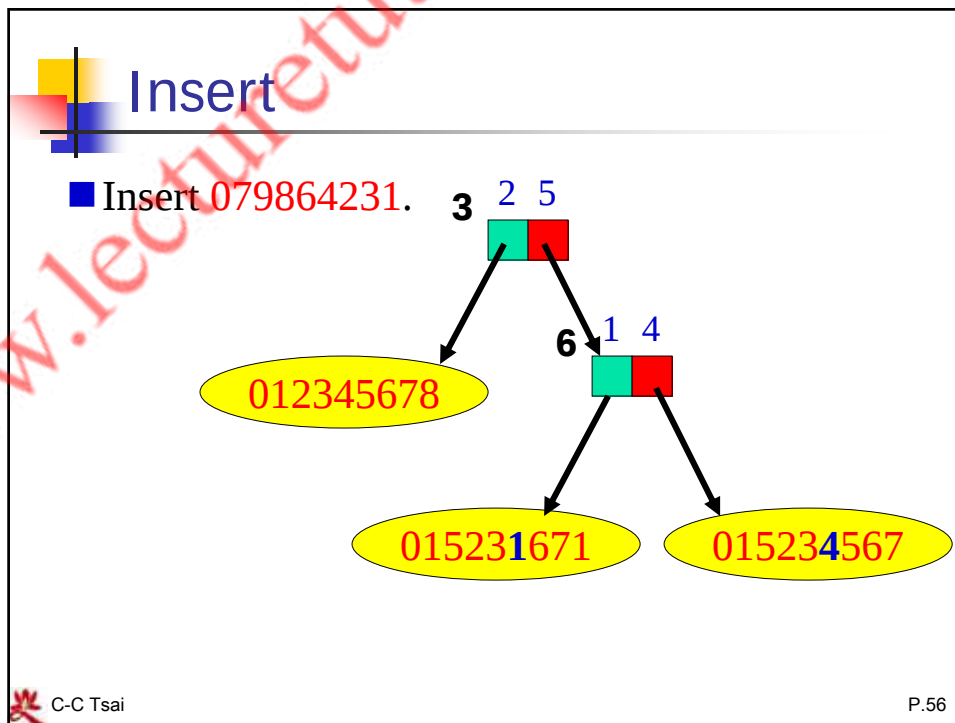
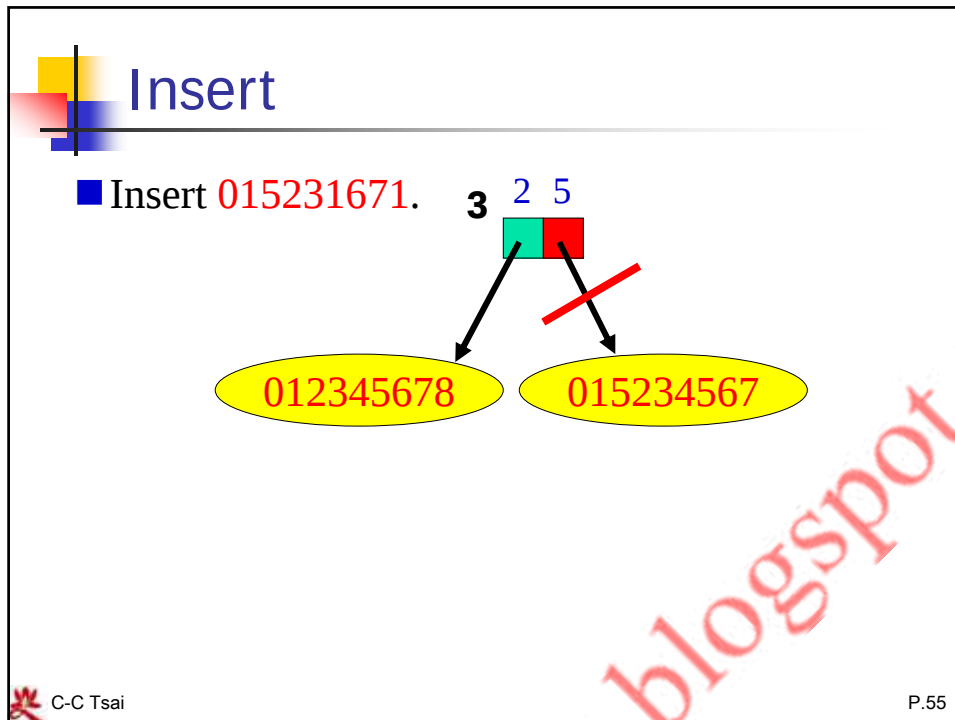
C-C Tsai P.53

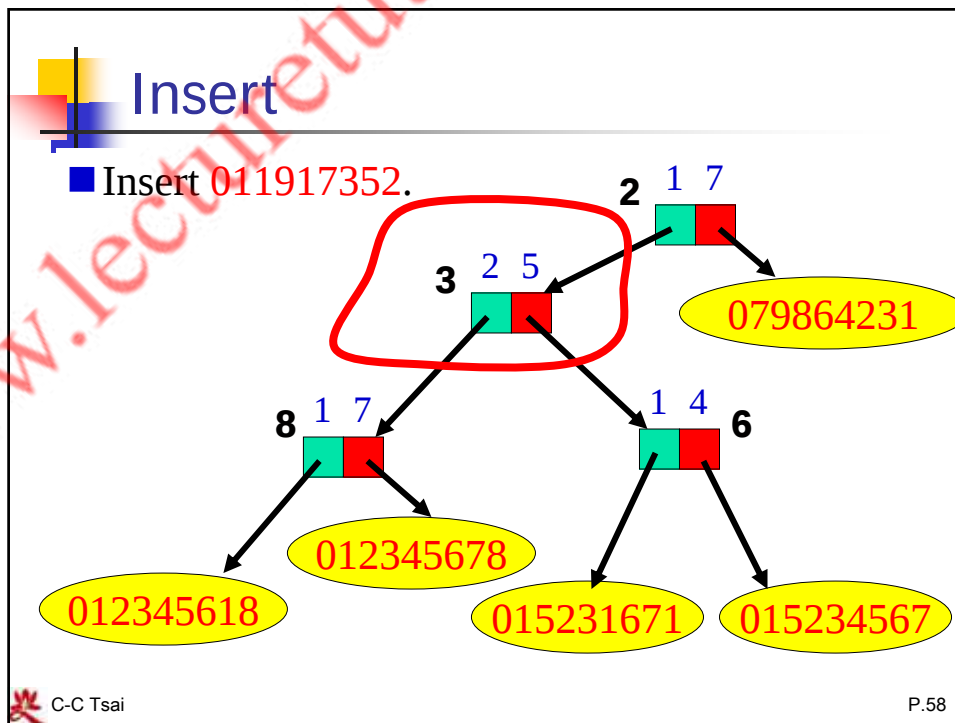
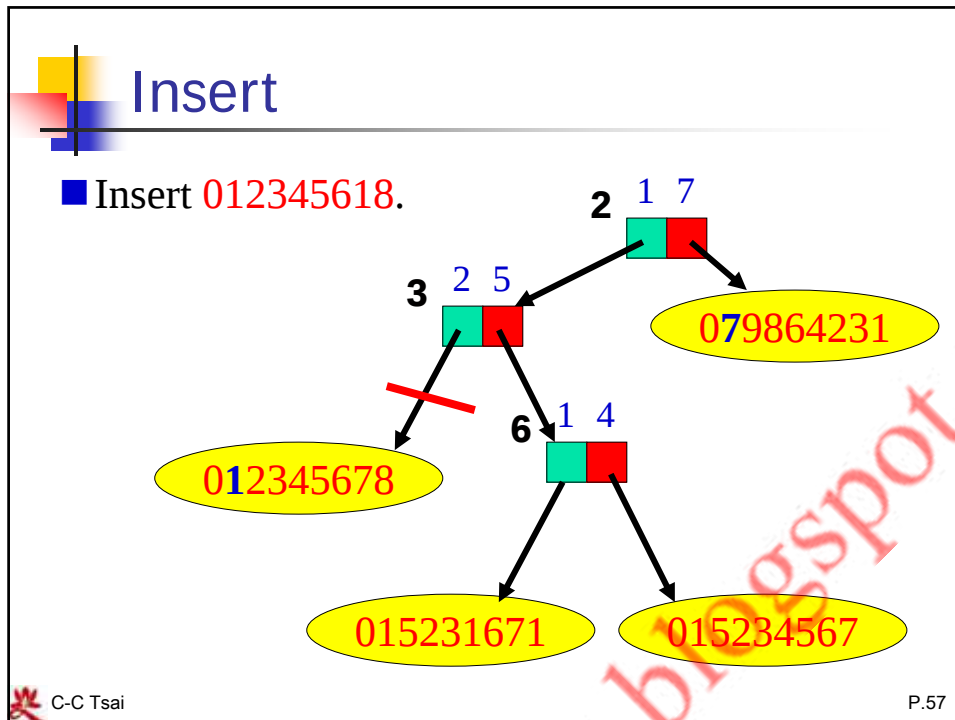
Insert

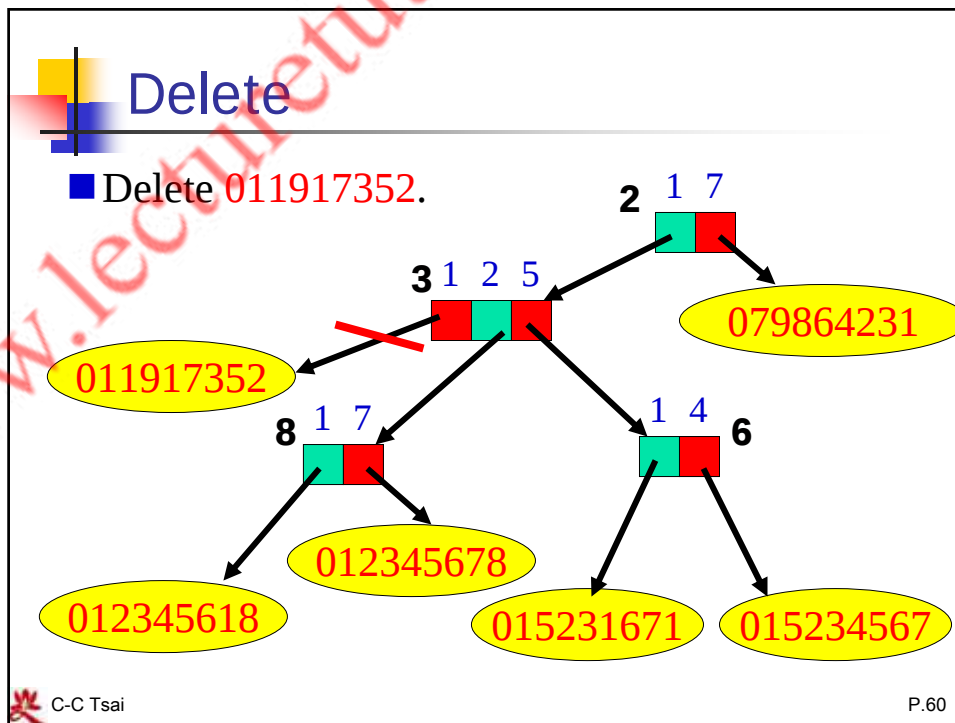
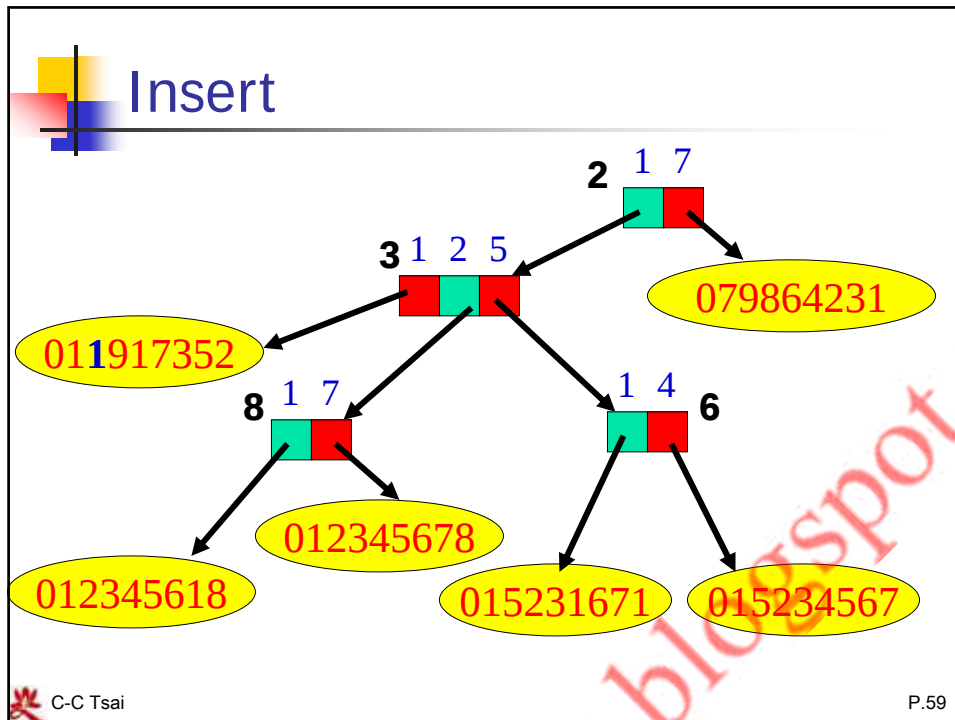
- Insert 012345678.
- Insert 015234567.

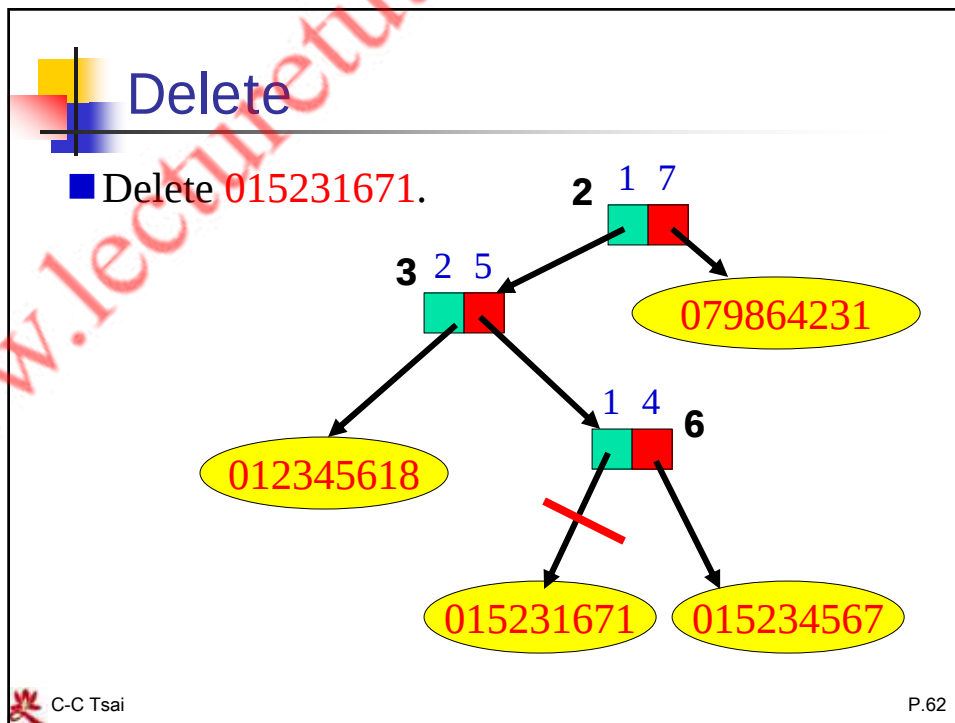
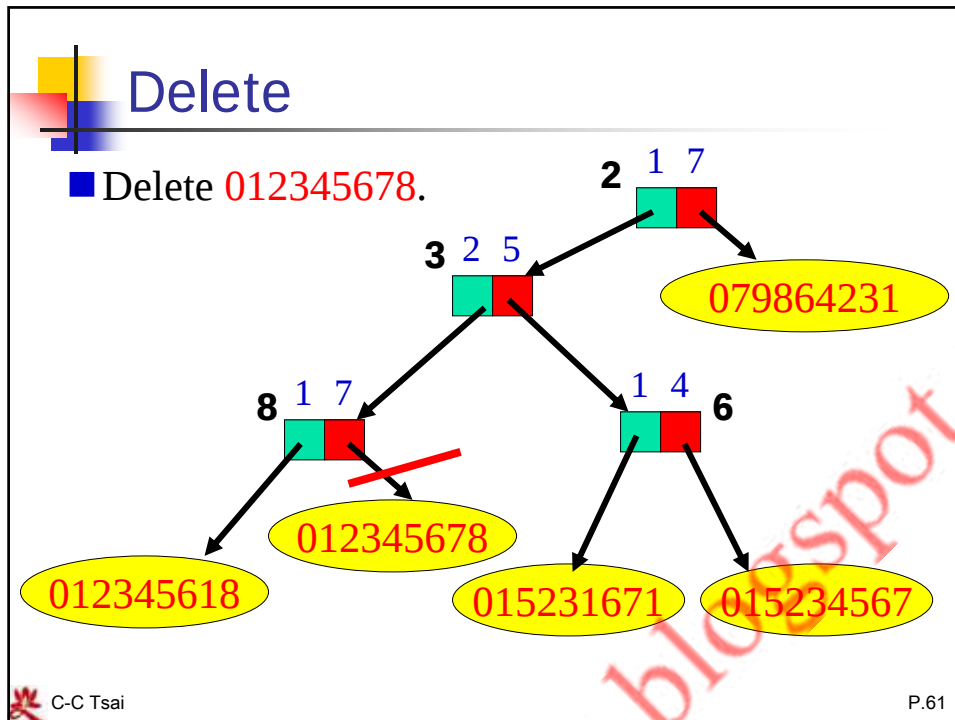
3: The 3rd digit is used for branching
Null pointer fields not shown.

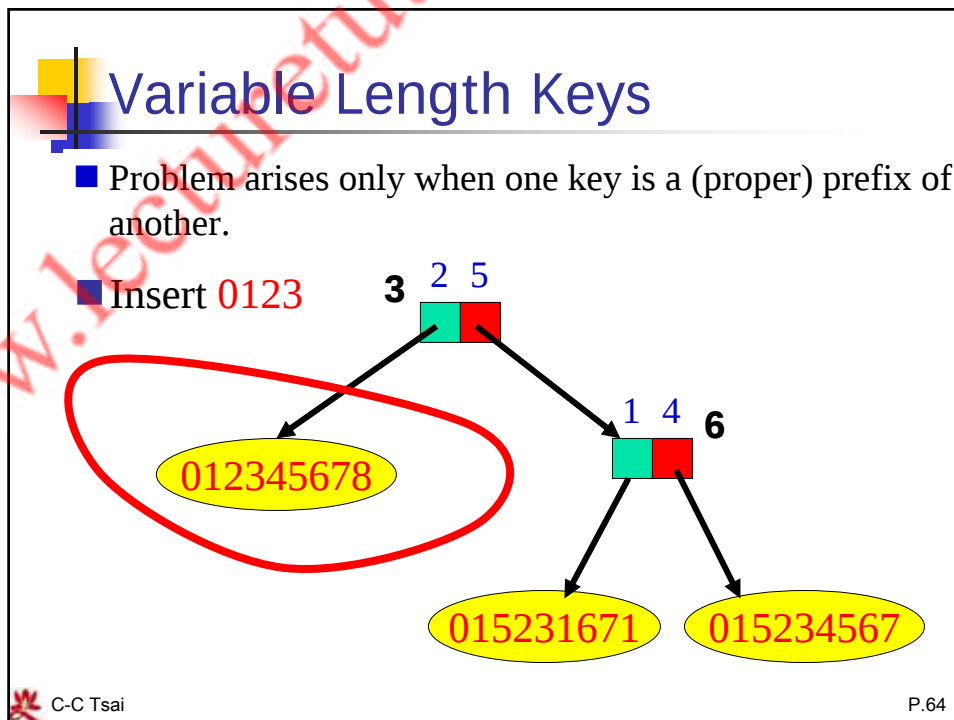
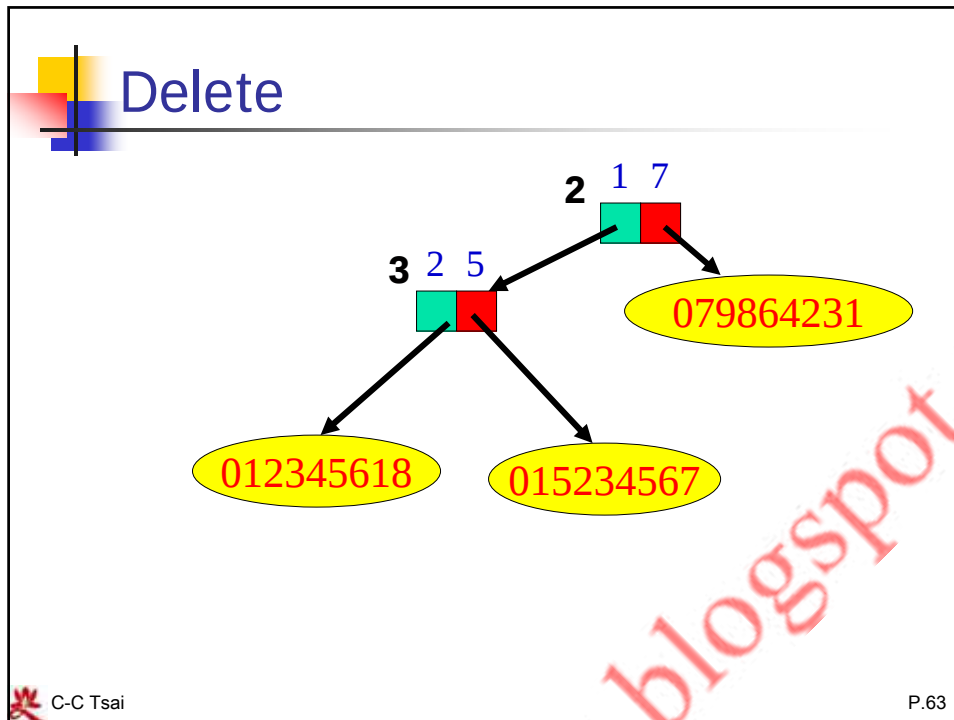
C-C Tsai P.54

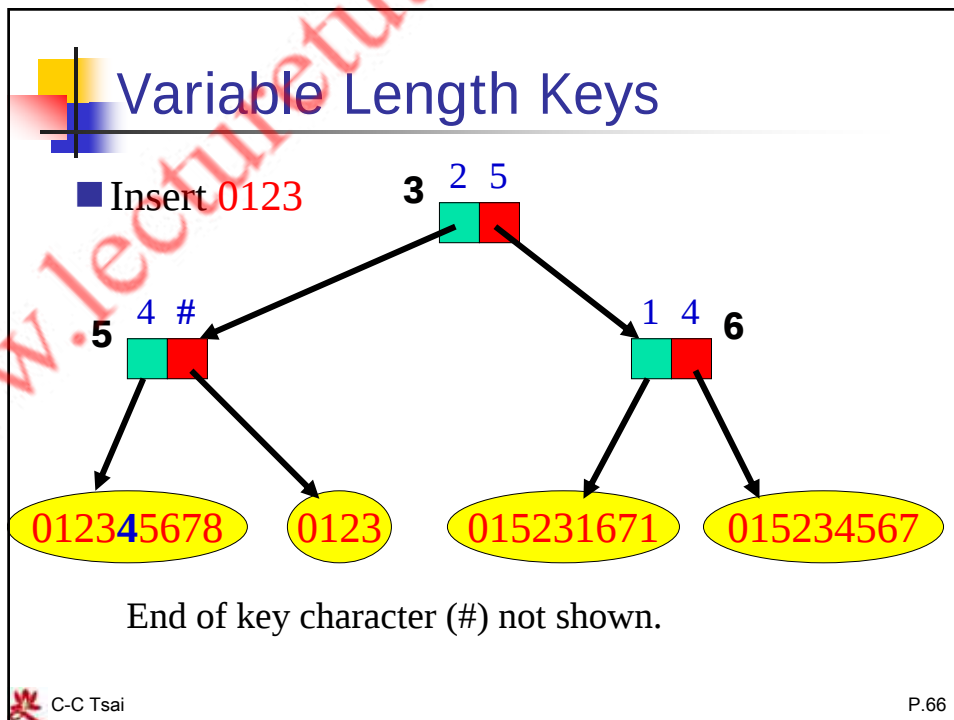
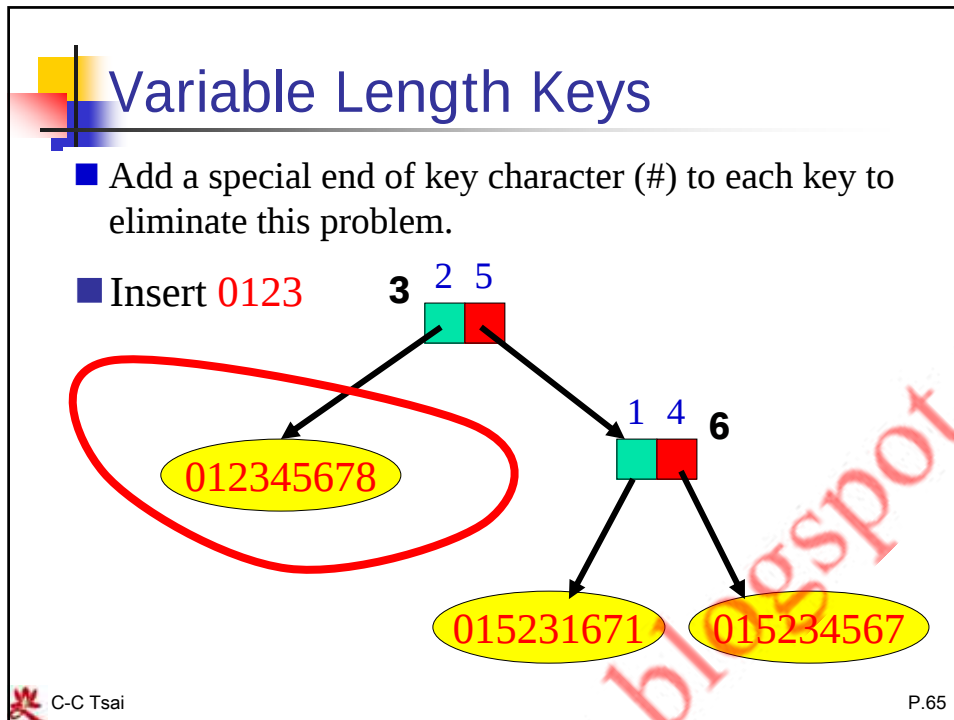














Tries With Edge Information

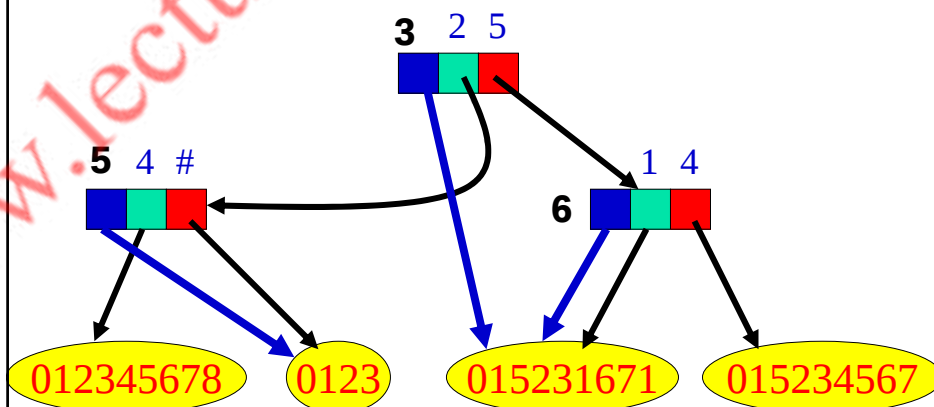
- Add a new field (**element**) to each branch node.
- New field points to any one of the element nodes in the subtree.
- Use this pointer on way down to figure out skipped-over characters.

C-C Tsai

P.67



Example



■ **element** field shown in blue.

C-C Tsai

P.68

Trie Characteristics

- Expected height of an order m trie is $\sim \log_m n$.
- 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 $\leq s$ (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.

C-C Tsai

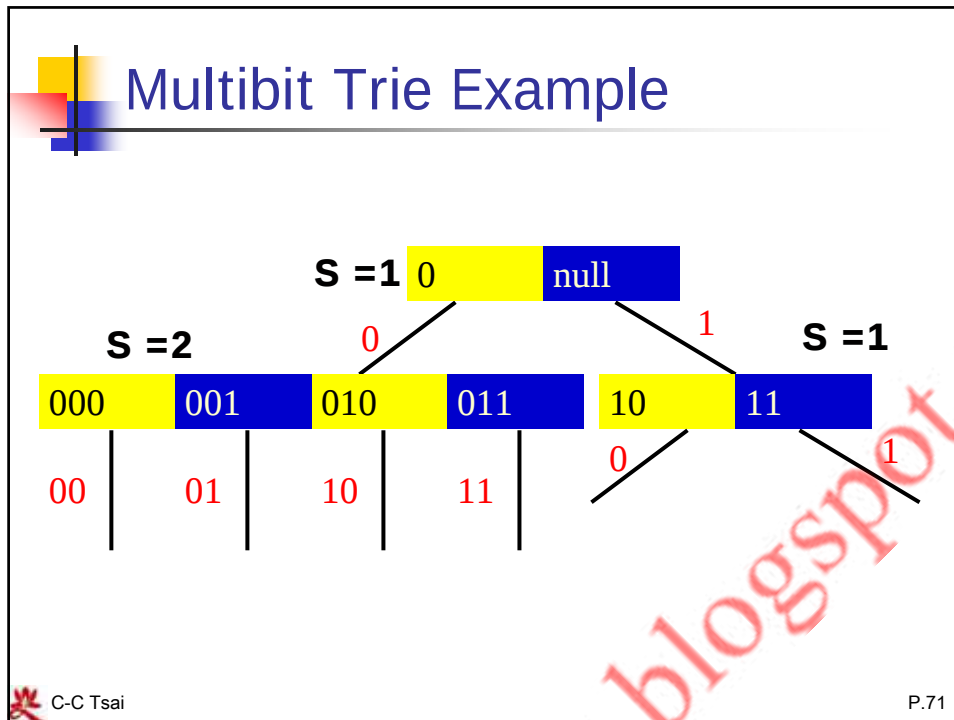
P.69

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 $\Rightarrow$ height = 1.
 - Strides of 16, 8, and 8 for levels 1, 2, and 3 $\Rightarrow$ only 3 levels.

C-C Tsai

P.70



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.

C-C Tsai P.72