**Unit-4**

**1b)B-Trees**

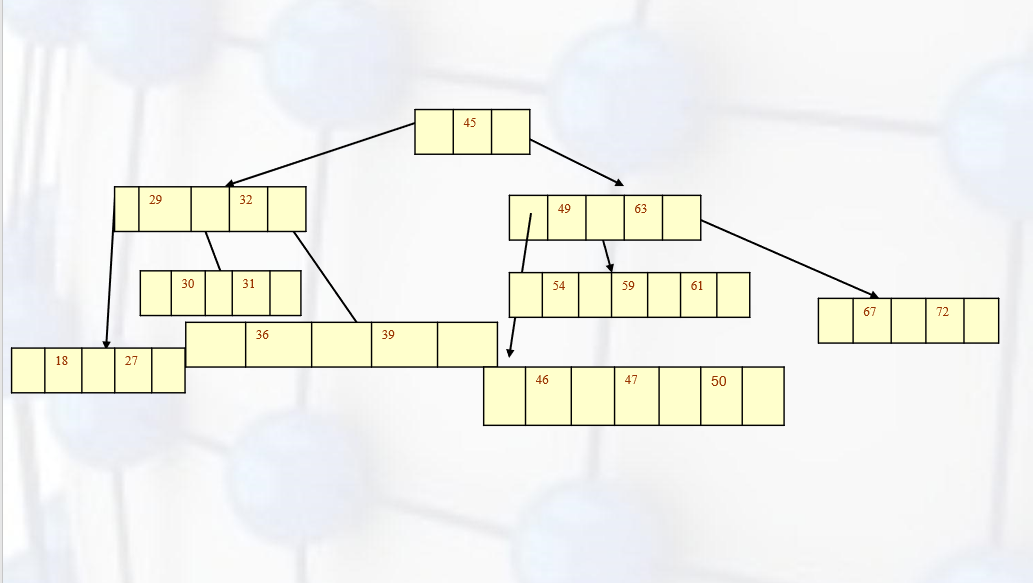
**properties:**

Every node in the B-tree has at most (maximum) *m* children.

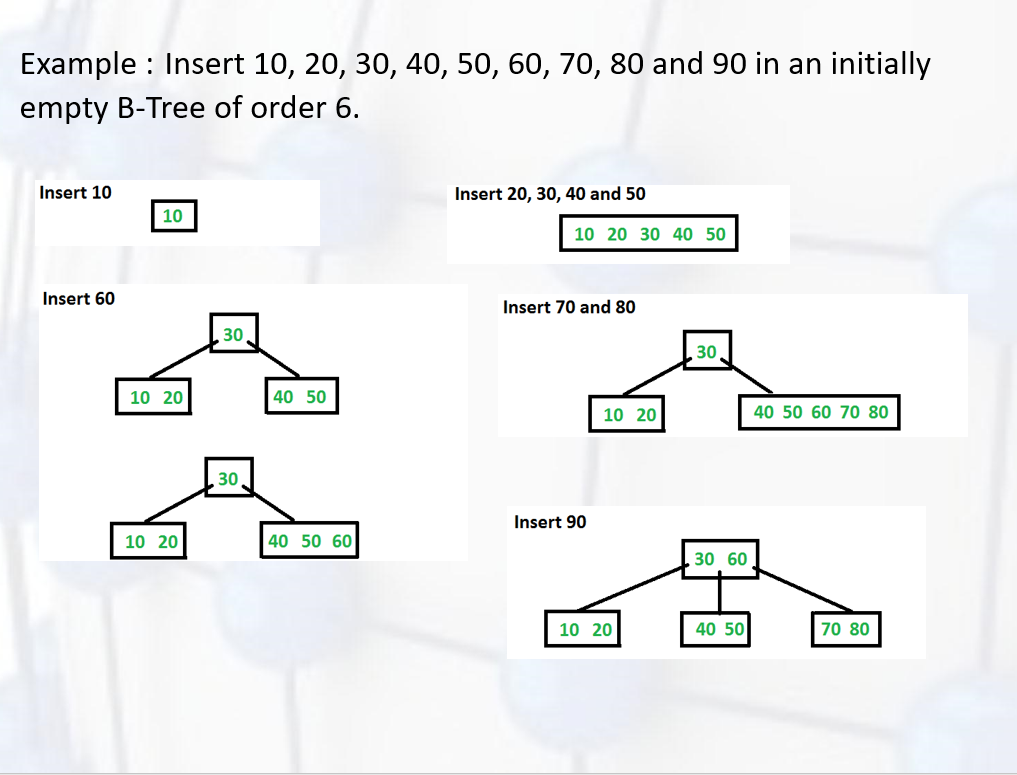
Every node in the B-tree except the root node and leaf nodes have at least (minimum) *m*⁄2 children. This condition helps to keep the tree bushy so that the path from the root node to the leaf is very short even in a tree that stores a lot of data.

The root node has at least two children if it is not a terminal (leaf) node.

All leaf nodes are at the same level.

An internal node in the B tree can have n number of children, where 0 ≤n ≤ m. it is not necessary that every node has the same number of children, but the only restriction is that the node should have at least m/2 children.. 

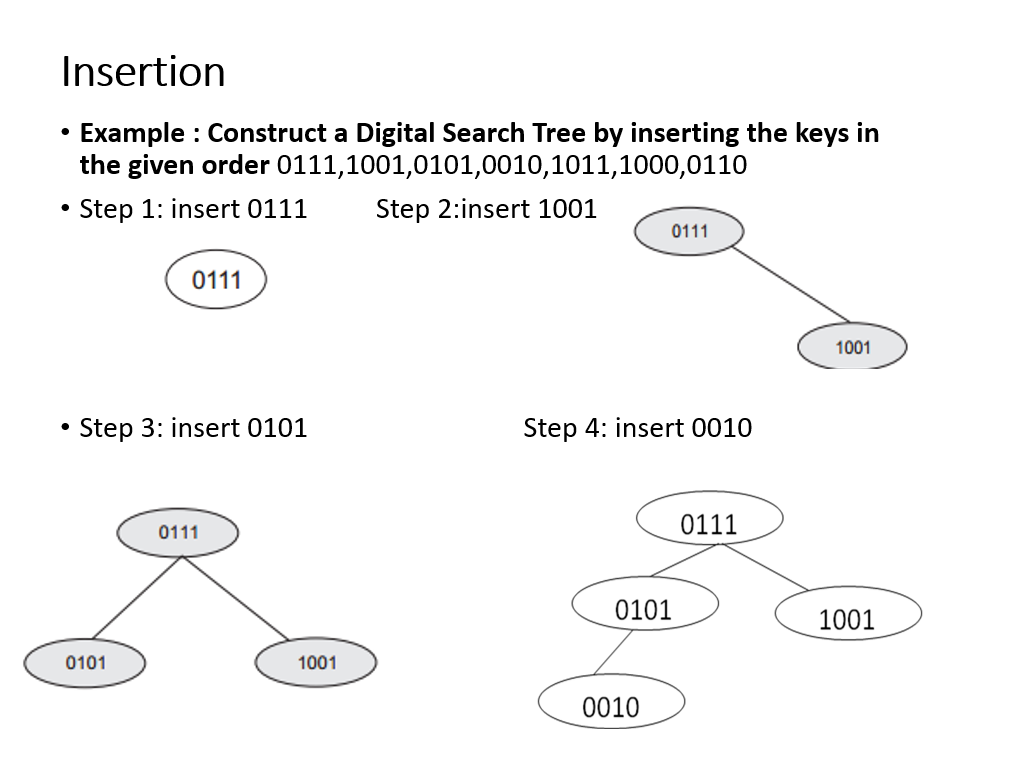
1a)

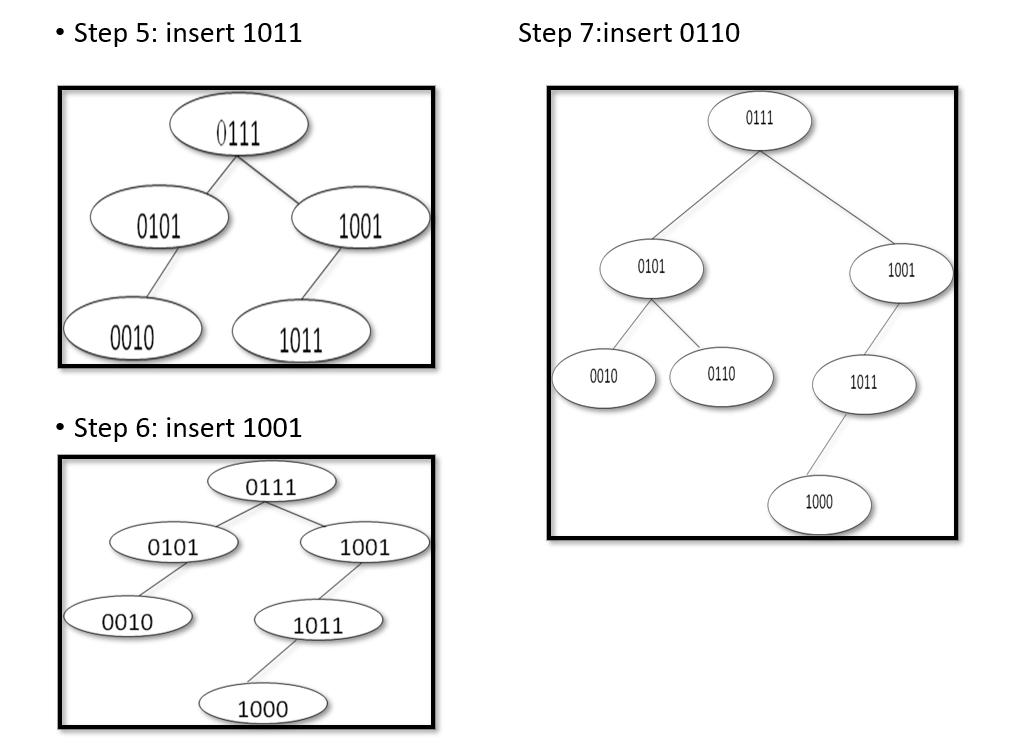


2b)

Deleting an element from a B+ Tree

* Start at root, find leaf node L where the element exists and remove it.
* If L is atleast half-full, exit
* If L underflows (has less than m/2 elements)
  + Try to re-distribute by borrowing from sibling (adjacent node with same parent as L). Take the max from left sibling or min value from right sibling. Update the intervening element in parent with the min element in the second node of the existing two.
  + If re-distribution fails, merge L and sibling. Delete the intervening element in parent of L. Merge could propagate to root, decreasing height.
  + If the interior(index) node underflows, merge the node with sibling and (move down the) intervening element.

3a)



**3b)Algorithm for Deletion**

ELSE IF temp->left ! delete\_dst(tree \*t, key)

Step 1: IF t = = Null

Write “Tree is NULL, deletion not possible”

[END OF IF]

Go to Step 4

Step 2: Use the “dst\_search(t, key)” algorithm and find the position of the node to

be deleted. Let the node to be deleted is represented as “temp” and its parent is

represented as “previous”.

Step 3: IF temp ->left = = NULL && temp ->right = = NULL

IF previous -> left = = temp

SET previous ->left = NULL

FREE temp

ELSE

SET previous ->right = NULL

FREE temp

[END of IF]

= NULL

Replace temp->value with any leaf node value of left sub tree

ELSE

Replace temp->value with any leaf node value of right sub tree

[END of IF]

Step 4: EXIT

* 4a) **Insertion:** **For performing the operation of insertion, the first key inserted into the DST is considered as the root node.**
* **The consecutive keys to be inserted are compared with the root node.**
* **If the first bit of the key is 0 (key starts with bit 0) then the key is inserted as a left child at Level 1 and if the key starts with bit ‘1’ then the key is inserted as a right child at Level 1.**
* **If already a node exists at this level 1, the next bit is considered to decide the position of the key at next level.**
* **The process is repeated until all the keys are inserted into the digital search tree.**

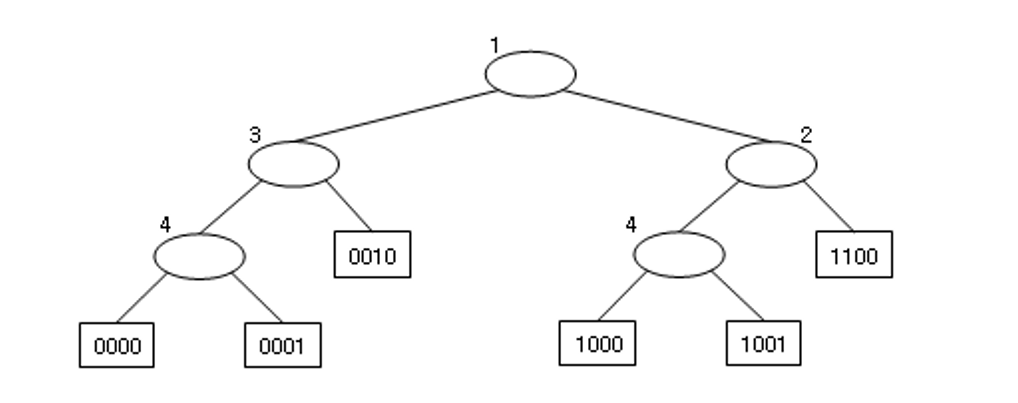
Ex:3a answer is example.

4b) **Search Operation**

* Searching for an element in a B-trees is similar to that in m-way search trees.
* **Example1:** To search for 59, we begin at the root node. The root node has a value 45 which is less than 59. So, we traverse in the right sub-tree. The right sub-tree of the root node has two key values, 49 and 63. Since 49 ≤ 59 ≤ 63, we traverse the right sub-tree of 49, that is, the left sub-tree of 63. This sub-tree has three values, 54, 59, and 61. On finding the value 59, the search is successful.
* **Example 2:** To search for 9, we traverse the left sub-tree of the root node. The left sub-tree has two key values — 29 and 32. Again, we traverse the left sub-tree of 29. We find that it has two key values, 18 and 27. There is no left sub-tree of 18, hence the value 9 is not stored in the tree.

5-unit

1a) **Compressed Binary Trie**

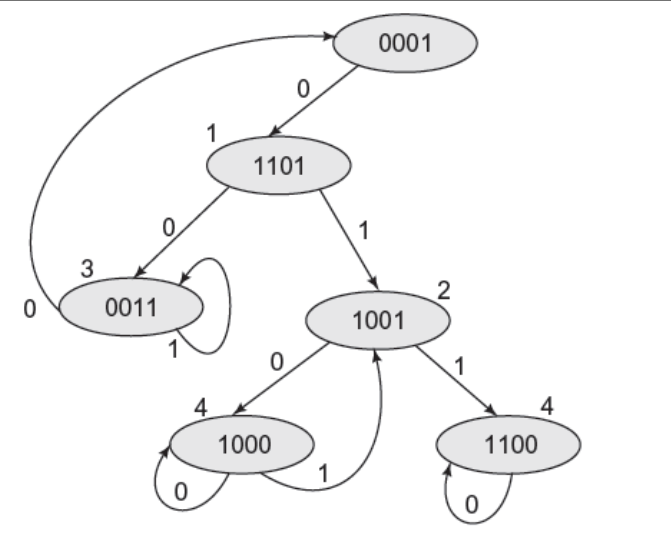
* A Compressed binary trie is a binary trie that is modified by removing all the nodes with degree one.
* There is no branch node whose degree is 1
* A BitNumber is added to each branch node, so it consists three fields Left child, Right child and bit field.
* The BitNumber tells which bit of the key to be used to decide whether to move to the left or right subtrie.
* 
* Example: Search the key 1001 in the Compressed Trie.

**Solution:**

* Compare the search key with root node (branch node). It consists of BitNumber 1, which indicates the branching is to be done depending on the first bit of the data. Since the first bit in the search key is 1, move to the right child of the root.
* The right child is the branch node with BitNumber 2, which indicates that the branching is done depending on the second bit of the data.
* In the search key, the second bit is 0 and so move to the left branch. Observe that this branch node has BitNumber 4.
* The 4th bit in the search key is 1, so move to the right branch.
* Finally the element node is reached. Compare the data in the element node with the search key.
* The search key is matched with the data in the element node. Hence, the search is successful.
* Note: The number of moves for searching 1001 are reduced from 4 in binary trie to 3 in compressed binary trie.

1b) **Searching Patricia**

* To search for a key in Patricia, we start at the root and proceed down the tree, using the BitNumber in each node to tell us which bit to examine in the search key.
* We proceed left if the bit is 0 and right if it is 1.
* The keys in the nodes are not examined at all on the way down the tree.
* Eventually, when an element pointer is encountered i.e pointer to node with a lesser BitNumber, the node value is checked with the search key.
* Thus, if the key at the node pointed to by this element pointer is equal to the search key, then the search is successful; otherwise, it is unsuccessful.
* Example: Search for 1001 in the Patricia



**Algorithm for Searching Patricia**

**Patricia\* patricia\_ search(Patricia \*t, key)**

**// returns a pointer to node whose data is checked with key**

**Patricia p, y;**

**Step 1: IF t = = NULL**

**Return t // empty tree**

**[END of IF]**

**Step 2: SET y = t->left\_child; // move to left child**

**SET p = t;**

**Step 3:**

**WHILE y->bit\_number > p->bit\_number**

**do**

**SET p = y;**

**If key[y->bit\_number]= =0**

**SET y = y->left\_child;**

**ELSE**

**SET y = y->right\_child;**

**[END of IF]**

**[END of WHILE]**

**Step 4: IF key = = y ->data**

**Write “Element Found”**

**ELSE**

**Write “Element Not Found”**

**[END of IF]**

**Return y**

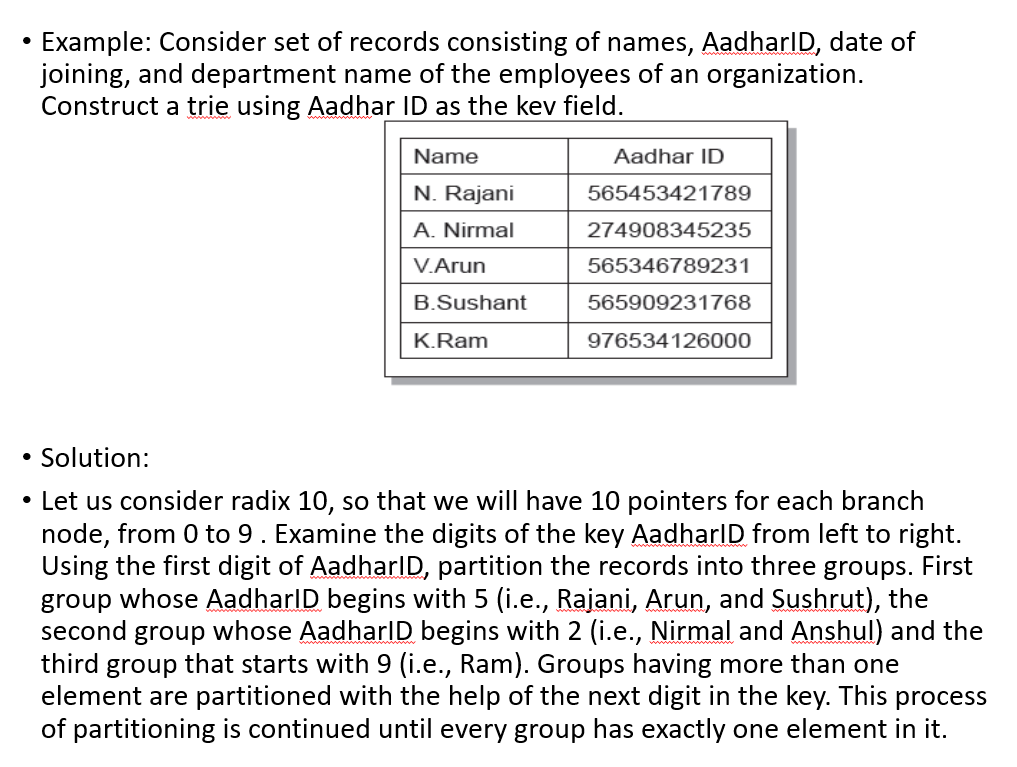
**Step 5: EXIT**

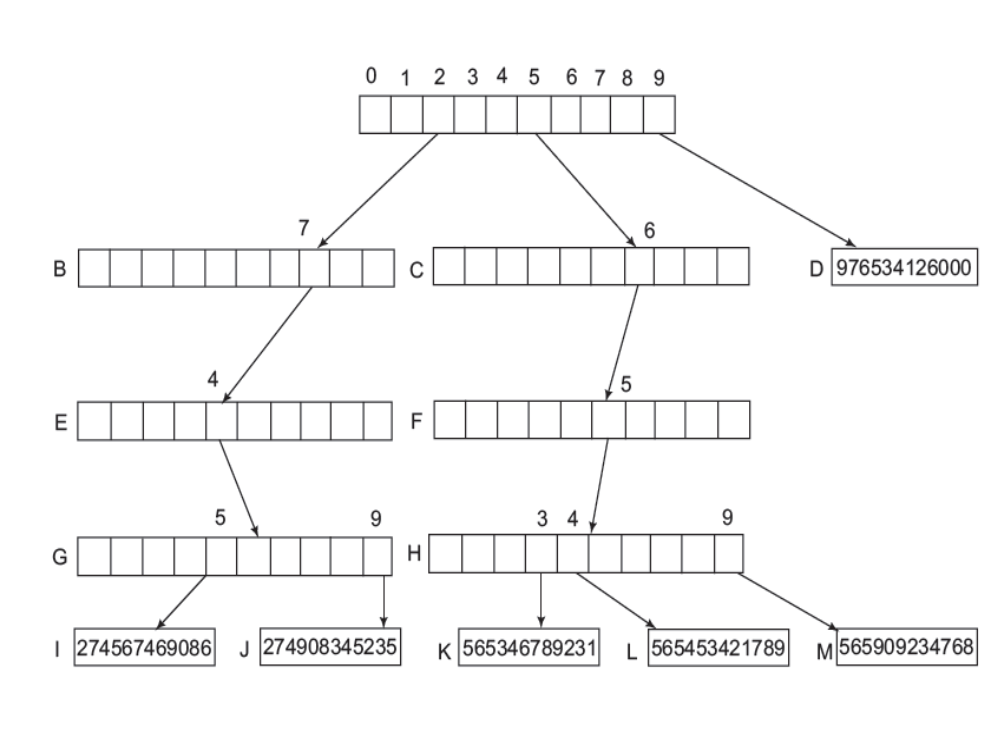
2a) MULTI-WAY TRIES

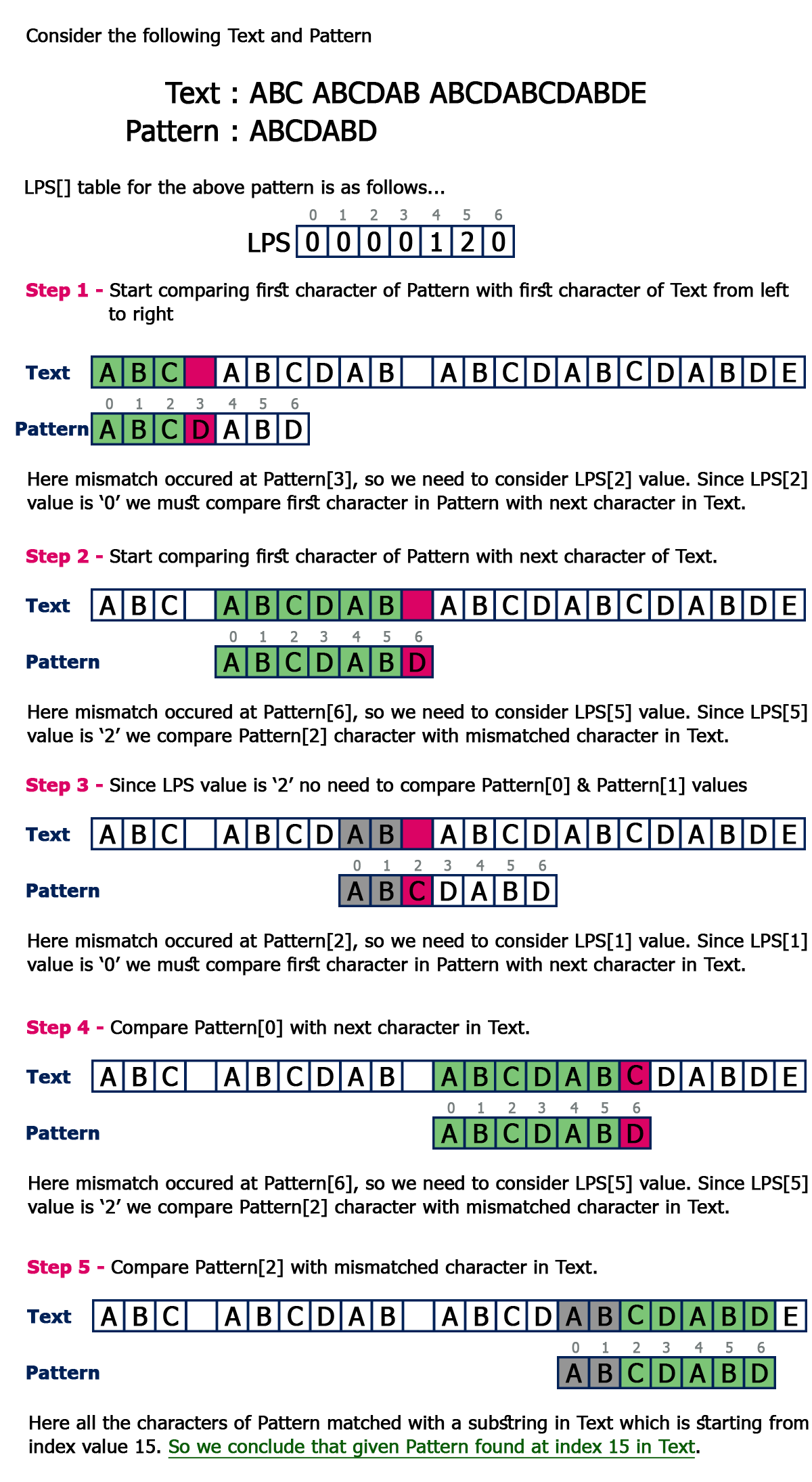
* A *multi-way trie* (or simply trie) is a tree data structure used to store strings of varying length.
* The word ‘TRIE’ is extracted from the word ‘RETRIEVAL’.
* A trie is used for efficient retrieval of the data, i.e., for performing efficient search on the data.
* A trie is a tree of degree m>=2 in which the branching at any level of the tree is determined not by entire key value, but by only a portion of it.
* The trie consists of two types of nodes, i.e., element nodes and branch nodes.
* The element node has only a data field which consists of the key which is being stored in the trie.
* The branch node consists of the pointers to other sub-trees which may again contain pointers to other sub-trees or pointers to element nodes.

The elements or keys are stored in the leaf nodes

* Example: Consider the trie which stores English words of different lengths. In this trie, each branch node contains 27 pointers, 26 pointers pointing to English alphabets and an extra pointer field which stores a blank character that is used to terminate the keys.
* Operations:
  + Searching
  + Insertion
  + Deletion





**2b) Let us see a working example of KMP Algorithm to find a Pattern in a Text..**

4a)