Program : **B.Tech**

Subject Name: **Analysis and Design of Algorithm**

Subject Code: **CS-402**

Semester: **4th**

Unit-5

Syllabus: Binary search trees, height balanced trees, 2-3 trees, B-trees, basic search and traversal techniques for trees and graphs (In order, preorder, postorder, DFS, BFS), NP-completeness.

BINARY SEARCH TREES:

In a binary tree, every node can have maximum of two children but there is no order of nodes based on their values. In binary tree, the elements are arranged as they arrive to the tree, from top to bottom and left to right. To enhance the performance of binary tree, we use special type of binary tree known as Binary Search Tree.

Binary search tree mainly focus on the search operation in binary tree. Binary search tree can be defined as follows:-

Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.

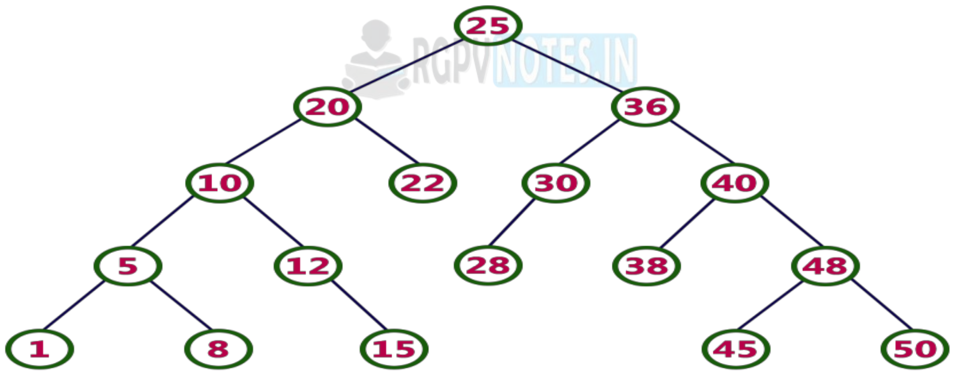

Figure 5.1 Binary Search Tree

The following operations are performed on a binary search tree:-1) Search 2) Insertion 3) Deletion

Search Operation in BST

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows:-

- Step 1: Read the search element from the user
- Step 2: Compare, the search element with the value of root node in the tree.
- Step 3: If both are matching, then display "Given node found!!!" and terminate the function
- Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

- Step 5: If search element is smaller, then continue the search process in left subtree.
- Step 6: If search element is larger, then continue the search process in right subtree.
- Step 7: Repeat the same until we found exact element or we completed with a leaf node
- Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.
- Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows:-
Step 1: Create a newNode with given value and set its left and right to
NULL. Step 2: Check whether tree is Empty.
Step 3: If the tree is Empty, then set set root to newNode.
Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).
Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.
Step 6: Repeat the above step until we reach to a leaf node (e.i., reach to NULL).
Step 7: After reaching a leaf node, then isert the newNode as left child if newNode is smaller or equal to that leaf else insert it as right child.

Deletion Operation in BST
In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree has following three cases:-

Case 1: Deleting a leaf node
We use the following steps to delete a leaf node from BST:-
Step 1: Find the node to be deleted using search operation
Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child
We use the following steps to delete a node with one child from BST:-
Step 1: Find the node to be deleted using search operation
Step 2: If it has only one child, then create a link between its parent and child
nodes. Step 3: Delete the node using free function and terminate the
function.

Case 3: Deleting a node with two children
We use the following steps to delete a node with two children from BST:-
Step 1: Find the node to be deleted using search operation
Step 2: If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.
Step 3: Swap both deleting node and node which found in above step.
Step 4: Then, check whether deleting node came to case 1 or case 2 else goto
steps 2 Step 5: If it comes to case 1, then delete using case 1 logic.
Step 6: If it comes to case 2, then delete using case 2 logic.
Step 7: Repeat the same process until node is deleted from the tree.

Binary Search Tree – Traversal

There are three types of binary search tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

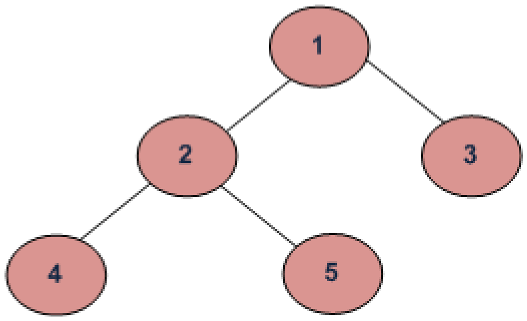

Figure 5.2 Binary Search Tree

1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order

traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

3. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

HEIGHT BALANCED TREES:

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor = heightOfLeftSubtree – heightOfRightSubtree

AVL Tree Rotations: Rotation is the process of moving the nodes to either left or right to make tree balanced.

There are **four** rotations and they are classified into **two** types.

1) Single Rotation

- Left rotation
- Right rotation

2)Double Rotation

- Left-Right rotation
- Right-Left rotation

INSERTION AND DELETION IN AVL TREE:

So time complexity of AVL insert is O(Logn). The AVL tree and other self balancing search trees like Red Black are useful to get all basic operations done in O(Log n) time. The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion.

Insertion Operation:

1. Insert the new Node using recursion so while back tracking you will all the parents nodes to check whether they are still balanced or not.
2. Every node has a field called height with default value as 1.
3. When new node is added, its parent's node height get increased by 1.
4. So as mentioned in step 1, every ancestors height will get updated while back tracking to the root.
5. At every node the balance factor will also be checked. **balance factor = (height of left Subtree — height of right Subtree).**
6. If **balance factor =1** means tree is balanced at that node.
7. If **balance factor >1** means tree is not balanced at that node, left height is more that the right height so that means we need rotation. (Either **Left-Left Case or Left-Right Case**).
8. Say the current node which we are checking is X and If new node is less than the X.left then it will be **Left-Left case**, and if new node is greater than the X.left then it will be **Left-Right case**. see the pictures above.

9. If **balance factor <-1** means tree is not balanced at that node, right height is more that the left height so that means we need rotation. (Either **Right-Right Case or Right– Left Case**)
10. Say the current node which we are checking is X and If new node is less than the X.right then it will be **Right-Right case**, and if new node is greater than the X.right then it will be **Right-Left case**.

Examples:

An important example of AVL trees is the behavior on a worst-case add sequence for regular binary trees:

1, 2, 3, 4, 5, 6, 7

All insertions are **right-right** and so rotations are all **single rotate** from the **right**. All but two insertions require re-balancing:

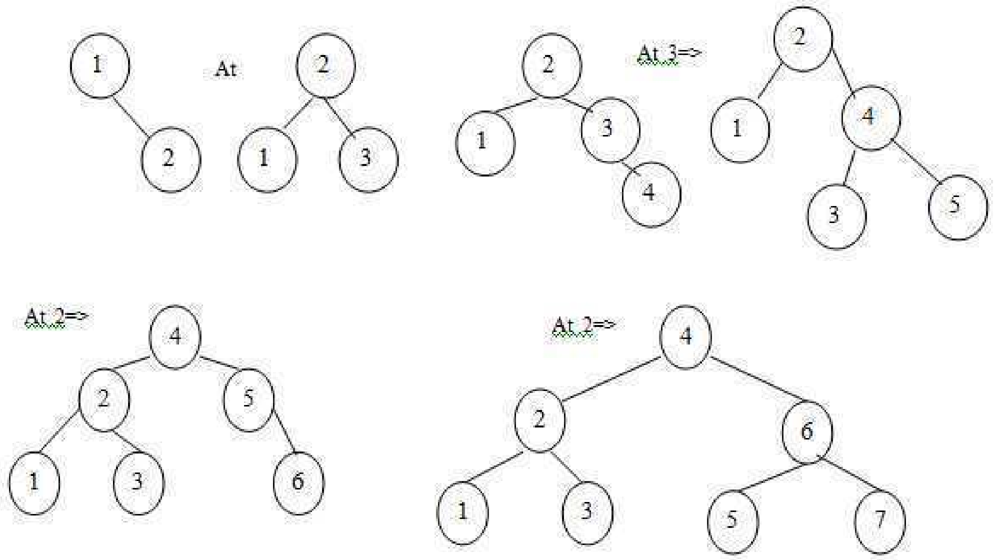

Figure:5.3 AVL Tree Insertion

Deletion in AVL Tree: If we want to delete any element from the AVL Tree we can delete same as BST deletion. for example Delete 30 in the AVL tree from the figure Figure:5.4 .

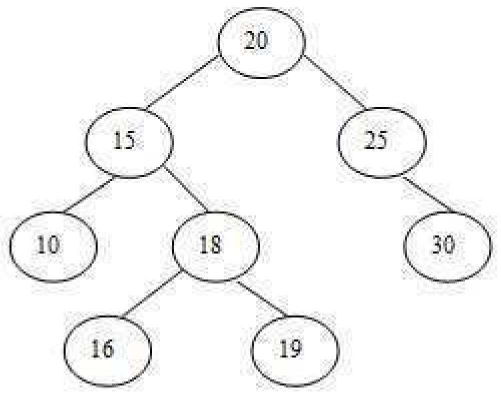

Figure:5.4 AVL Tree

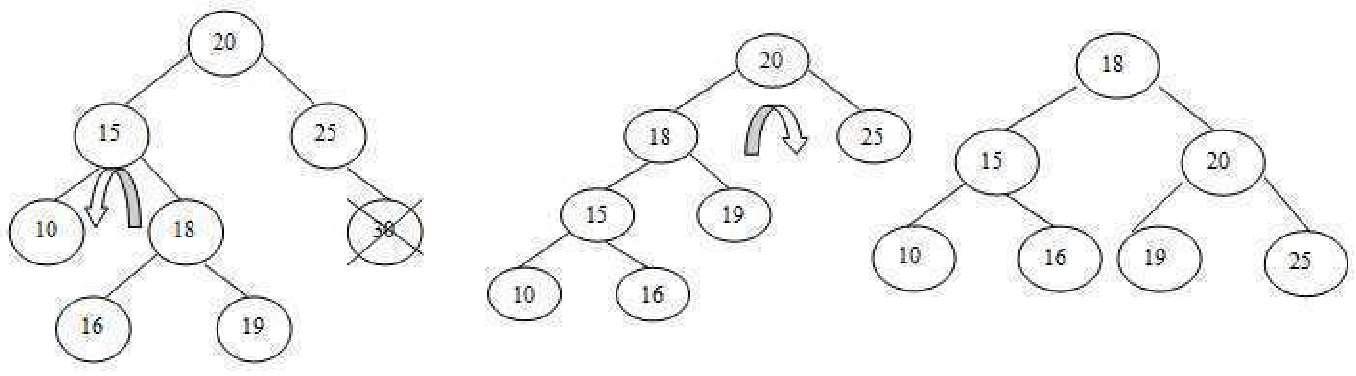

Figure:5.5 AVL Tree Deletion

A node with value 30 is being deleted in figure 5.5. After deleting 30, we travel up and find the first unbalanced node which is 18. We apply rotation and shift 18 to up for balanced tree .Again we have to move 18 up, so we perform left rotation.

2-3 TREES:

A 2-3 Tree is a specific form of a B tree. A 2-3 tree is a search tree. However, it is very different from a binary search tree.
Here are the properties of a 2-3 tree:

- each node has either one value or two value
- a node with one value is either a leaf node or has exactly two children (non-null). Values in left subtree < value in node < values in right subtree
- a node with two values is either a leaf node or has exactly three children (non-null). Values in left subtree < first value in node < values in middle subtree < second value in node < value in right subtree.
- all leaf nodes are at the same level of the tree

Insertion algorithm

Into a two-three tree is quite different from the insertion algorithm into a binary search tree. In a two-three tree, the algorithm will be as follows:

1. If the tree is empty, create a node and put value into the node
2. Otherwise find the leaf node where the value belongs.
3. If the leaf node has only one value, put the new value into the node
4. If the leaf node has more than two values, split the node and promote the median of the three values to parent.
5. If the parent then has three values, continue to split and promote, forming a new root node if necessary

The lookup operation

Recall that the lookup operation needs to determine whether key value k is in a 2-3 tree T. The lookup operation for a 2-3 tree is very similar to the lookup operation for a binary-search tree. There are 2 base cases:
1. T is empty: return false
2. T is a leaf node: return true iff the key value in T is k
And there are 3 recursive cases:
1. k <= T.leftMax: look up k in T's left subtree
2. T.leftMax < k <= T.middleMax: look up k in T's middle subtree
3. T.middleMax < k: look up k in T's right subtree
It should be clear that the time for lookup is proportional to the height of the tree. The height of the tree is O(log N) for N = the number of **nodes** in the tree. You may think this is a problem, since the actual values are only at the leaves. However, the number of leaves is always greater than N/2 (i.e., more than half the nodes in

the tree are leaves). So the time for lookup is also O(log M), where M is the number of key values stored in the tree.

The delete operation

Deleting key k is similar to inserting: there is a special case when T is just a single (leaf) node containing k (T is made empty); otherwise, the parent of the node to be deleted is found, then the tree is fixed up if necessary so that it is still a 2-3 tree.

Once node n (the parent of the node to be deleted) is found, there are two cases, depending on how many children n has:

case 1: n has 3 children

- Remove the child with value k, then fix n.leftMax, n.middleMax, and n's ancestors' leftMax and middleMax fields if necessary.

case 2: n has only 2 children

- If n is the root of the tree, then remove the node containing k. Replace the root node with the other child (so the final tree is just a single leaf node).
- If n has a left or right sibling with 3 kids, then:
  - remove the node containing k
  - "steal" one of the sibling's children
  - fix n.leftMax, n.middleMax, and the leftMax and middleMax fields of n's sibling and ancestors as needed.
- If n's sibling(s) have only 2 children, then:
  - remove the node containing k
  - make n's remaining child a child of n's sibling
  - fix leftMax and middleMax fields of n's sibling as needed
  - remove n as a child of its parent, using essentially the same two cases (depending on how many children n's parent has) as those just discussed

The time for delete is similar to insert; the worst case involves one traversal down the tree to find n, and another "traversal" up the tree, fixing leftMax and middleMax fields along the way (the traversal up is really actions that happen after the recursive call to delete has finished).

So the total time is 2 * height-of-tree = O(log N).

Complexity Analysis

- keys are stored only at leaves, ordered left-to-right
- non-leaf nodes have 2 or 3 children (never 1)
- non-leaf nodes also have leftMax and middleMax values (as well as pointers to children)
- all leaves are at the same depth
- the height of the tree is O(log N), where N = # nodes in tree
- at least half the nodes are leaves, so the height of the tree is also O(log M) for M = # values stored in tree
- the lookup, insert, and delete methods can all be implemented to run in time O(log N), which is also O(log M)

<u>B-TREE:</u>

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red Black Trees), it is assumed that everything is in main memory. To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, ..etc.

Properties of B-Tree

1) All leaves are at same level.

2) A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.

3) Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.

4) All nodes (including root) may contain at most 2t – 1 keys.

5) Number of children of a node is equal to the number of keys in it plus 1.

6) All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in range from k1 and k2.

7) B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.

8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(logn).

BASIC SEARCH AND TRAVERSAL TECHNIQUES FOR TREES AND GRAPHS:

Search

Search is similar to search in Binary Search Tree. Let the key to be searched be k. We start from root and recursively traverse down. For every visited non-leaf node, if the node has key, we simply return the node. Otherwise we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

Traversal

Tree traversal (also known as tree search) is a form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.

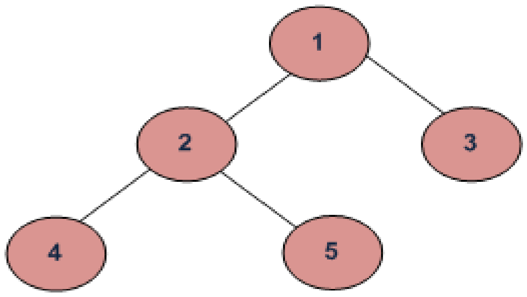

Figure: 5.6 Binary Tree

Depth First Traversals:

(a) Inorder (Left, Root, Right): 4 2 5 1 3

(b) Preorder (Root, Left, Right): 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal: 1 2 3 4 5

Graph Traversal Techniques:

Depth First Search (DFS)

The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node. Stack is used in the implementation of the depth first search.

Algorithmic Steps

Step 1: Push the root node in the Stack. Step 2: Loop until stack is empty.

Step 3: Peek the node of the stack.

Step 4: If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.

Step 5: If the node does not have any unvisited child nodes, pop the node from the stack.

Breadth First Search (BFS)

This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search.
Algorithmic Steps
Step 1: Push the root node in the Queue. Step 2: Loop until the queue is empty.
Step 3: Remove the node from the Queue.
Step 4: If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

NP-COMPLETENESS:

We have been writing about efficient algorithms to solve complex problems, like shortest path, Euler graph, minimum spanning tree, etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed.
Can all computational problems be solved by a computer? There are computational problems that cannot be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source Halting Problem).Status of NP Complete problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.
What are NP, P, NP-complete and NP-Hard problems?
P is set of problems that can be solved by a deterministic Turing machine in Polynomial time.
NP is set of decision problems that can be solved by a Non-deterministic Turing Machine in Polynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time) figure 5.1.
Informally, NP is set of decision problems which can be solved by a polynomial time via a “Lucky Algorithm”, a magical algorithm that always makes a right guess among the given set of choices (Source Ref 1).

NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:

1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is NP-Hard if it follows property 2 mentioned above, doesn’t need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.

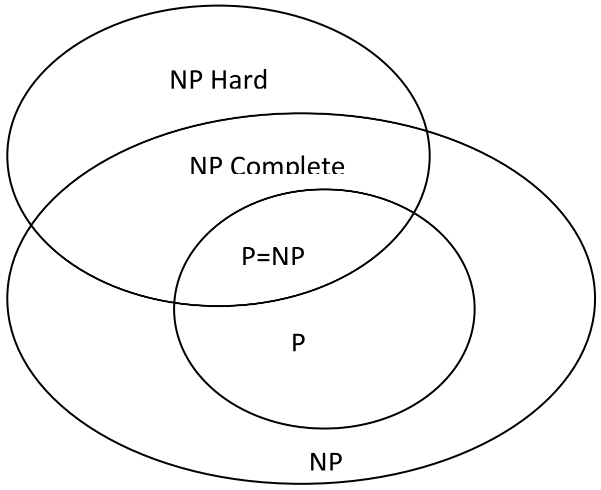

Figure: 5.7: Relationship between P,NP, NP Complete & NP Hard