UNIT – 5

Unit-05/Lecture-01

Computation ability

Unit-05/Lecture-02

NP-Hard and NP- Complete Problems

Introduction: NP-hard (Non-deterministic Polynomial-time hard), in computational complexity theory, is a class of problems that are, informally, "at least as hard as the hardest problems in NP". More precisely, a problem H is NP-hard when every problem L in NP can be reduced in polynomial time to H. As a consequence, finding a polynomial algorithm to solve any NP-hard problem would give polynomial algorithms for all the problems in NP, which is unlikely as many of them are considered hard.

A common mistake is to think that the NP in NP-hard stands for non-polynomial. Although it is widely suspected that there are no polynomial-time algorithms for NP-hard problems, this has never been proven. Moreover, the class NP also contains all problems which can be solved in polynomial time.

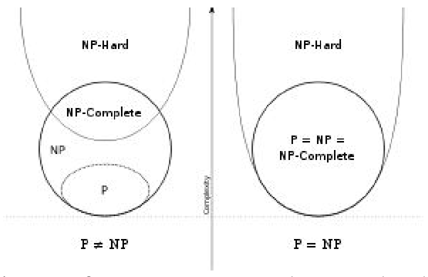


Figure: Euler diagram for P, NP, NP complete, NP hard set of problem.

Definitions :
A decision problem H is NP-hard when for any problem L in NP, there is a polynomial-time reduction from L to H An equivalent definition is to require that any problem L in NP can be solved in polynomial time by an oracle machine with an oracle for H. Informally, we can think of an algorithm that can call such an oracle machine as a subroutine for solving H, and solves L in polynomial time, if the subroutine call takes only one step to compute.

Another definition is to require that there is a polynomial-time reduction from an NP-complete problem G to H.As any problem L in NP reduces in polynomial time to G, Lreduces in turn to H in polynomial time so this new definition implies the previous one. It does not restrict the class NP-hard to decision problems, for instance it also includes search problems, or optimization problems.

Consequences

• If P? NP, then NP-hard problems cannot be solved in polynomial time, while P = NP does not resolve whether the NP-hard problems can be solved in polynomial time;
• If an optimization problem H has an NP-complete decision version L, then H is NP-hard.

Examples
An example of an NP-hard problem is the decision subset sum problem, which is this: given a set of integers, does any non-empty subset of them add up to zero? That is a decision problem, and happens to be NP-complete. Another example of an NP-hard problem is the optimization problem of finding the least-cost cyclic route through all nodes of a weighted graph. This is commonly known as the travelling salesman problem.

There are decision problems that are NP-hard but not NP-complete, for example the halting problem. This is the problem which asks "given a program and its input, will it run forever?" That is a yes/no question, so this is a decision problem. It is easy to prove that the halting problem is NP-hard but not NP-complete. For example, the Boolean satisfiability problem can be reduced to the halting problem by transforming it to the description of a Turing machine that tries all truth value assignments and when it finds one that satisfies the formula it halts and otherwise it goes into an infinite loop. It is also easy to see that the halting problem is not in NP since all problems in NP are decidable in a finite number of operations, while the halting problem, in general, is undecidable. There are also NP-hard problems that are neither NP-complete nor undecidable. For instance, the language of True quantified Boolean formulas is decidable in polynomial space, but not non-deterministic polynomial time (unless NP = PSPACE).

NP-naming convention
NP-hard problems do not have to be elements of the complexity class NP, despite having NP as the prefix of their class name. The NP-naming system has some deeper sense, because the NP family is defined in relation to the class NP and the naming conventions in the Computational Complexity Theory:

- **NP:** Class of computational problems for which a given solution can be verified as a solution in polynomial time by a deterministic Turing machine.
- **NP-hard:** Class of problems which are at least as hard as the hardest problems in NP. Problems in NP-hard do not have to be elements of NP; indeed, they may not even be decidable problems.
- **NP-complete:** Class of problems which contains the hardest problems in NP. Each element of NP-complete has to be an element of NP.
- **NP-easy:** At most as hard as NP, but not necessarily in NP, since they may not be decision problems.
- **NP-equivalent:** Exactly as difficult as the hardest problems in NP, but not necessarily in NP.

Application areas

NP-hard problems are often tackled with rules-based languages in areas such as:

- Configuration
- Data mining
- Selection
- Diagnosis
- Process monitoring and control
- Scheduling
- Planning
- Rosters or schedules
- Tutoring systems
- Decision support
- Phylogenetics

NP- Complete

Introduction: In computational complexity theory, a decision problem is NP-complete when it is both in NP and NP-hard. The set of NP-complete problems is often denoted by NP-C or NPC. The abbreviation NP refers to "nondeterministic polynomial time".

Although any given solution to an NP-complete problem can be verified quickly (in polynomial time), there is no known efficient way to locate a solution in the first place; indeed, the most notable characteristic of NP-complete problems is that no fast solution to them is known. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows. As a consequence, determining whether or not it is possible to solve these problems quickly, called the P versus NP problem, is one of the principal unsolved problems in computer science today.

While a method for computing the solutions to NP-complete problems using a reasonable amount of time remains undiscovered, computer scientists and programmers still frequently encounter NP-complete problems. NP-complete problems are often addressed by using heuristic methods and approximation algorithms.

Overview
NP-complete problems are in NP, the set of all decision problems whose solutions can be verified in polynomial time; NP may be equivalently defined as the set of decision problems that can be solved in polynomial time on a non-deterministic Turing machine. A problem p in NP is NP-complete if every other problem in NP can be transformed into pin polynomial time.

NP-complete problems are studied because the ability to quickly verify solutions to a problem (NP) seems to correlate with the ability to quickly solve that problem (P). It is not known whether every problem in NP can be quickly solved—this is called the P versus NP problem. But if any NP-complete problem can be solved quickly, then every problem in NP can, because the definition of an NP-complete problem states that every problem in NP must be quickly reducible to every NP-complete problem (that is, it can be reduced in polynomial time). Because of this, it is often said that NP-complete problems are harder or more difficult than NP problems in general.

Formal definition of NP-completeness

A decision problem C is NP-complete if:

1. C is in NP, and
2. Every problem in NP is reducible to C in polynomial time.

C can be shown to be in NP by demonstrating that a candidate solution to C can be verified in polynomial time.

Note that a problem satisfying condition 2 is said to be NP-hard, whether or not it satisfies condition

A consequence of this definition is that if we had a polynomial time algorithm (on a UTM, or any other Turing-equivalent abstract machine) for$C$, we could solve all problems in NP in polynomial time.

NP-complete problems

An interesting example is the graph isomorphism problem, the graph theory problem of determining whether a graph isomorphism exists between two graphs. Two graphs are isomorphic if one can be transformed into the other simply by renaming vertices. Consider these two problems:

- Graph Isomorphism: Is graph G1 isomorphic to graph G2?
- Sub graph Isomorphism: Is graph G1 isomorphic to a sub graph of graph G2?

The Sub graph Isomorphism problem is NP-complete. The graph isomorphism problem is suspected to be neither in P nor NP-complete, though it is in NP. This is an example of a problem that is thought to be hard, but is not thought to be NP-complete.

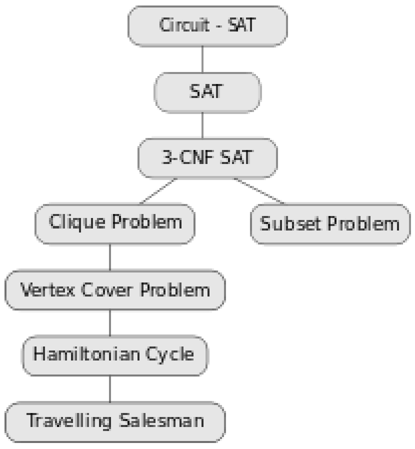


Figure: Some NP complete problem, indicating the reductions typically used to prove their NP completeness

The easiest way to prove that some new problem is NP-complete is first to prove that it is in NP, and then to reduce some known NP-complete problem to it. Therefore, it is useful to know a variety of NP-complete problems.

The list below contains some well-known problems that are NP-complete when expressed as decision problems.

- Boolean satisfiability problem (Sat.)
- Knapsack problem

- Hamiltonian path problem
- Travelling salesman problem
- Sub graph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Dominating set problem
- Graph colouring problem

To the right is a diagram of some of the problems and the reductions typically used to prove their NP-completeness? In this diagram, an arrow from one problem to another indicates the direction of the reduction. Note that this diagram is misleading as a description of the mathematical relationship between these problems, as there exists a polynomial-time reduction between any two NP-complete problems; but it indicates where demonstrating this polynomial-time reduction has been easiest.

There is often only a small difference between a problem in P and an NP-complete problem. For example, the 3-satisfiability problem, a restriction of the Boolean satisfiability problem, remains NP-complete, whereas the slightly more restricted 2-satisfiability problem is in P (specifically, NL-complete), and the slightly more general max. 2-sat. Problem is again NP-complete. Determining whether a graph can be colored with 2 colors is in P, but with 3 colors is NP-complete, even when restricted to planar graphs. Determining if a graph is a cycle or is bipartite is very easy (in L), but finding a maximum bipartite or a maximum cycle sub graph is NP-complete. A solution of the knapsack problem within any fixed percentage of the optimal solution can be computed in polynomial time, but finding the optimal solution is NP-complete.

Unit-05/Lecture-03

AVL Tree

Introduction
The AVL tree is named after its two Soviet inventors, Georgy Adelson-Velsky and E. M. Landis, who published it in their 1962 paper "An algorithm for the organization of information".

In computer science, an AVL tree (Georgy Adelson-Velsky and Landis' tree, named after the inventors) is a self-balancing binary search tree. It was the first such data structure to be invented. In an AVL tree, the heights of the two child sub trees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

AVL trees are often compared with red-black trees because both support the same set of operations and take O(log n) time for the basic operations. For lookup-intensive applications, AVL trees are faster than red-black trees because they are more rigidly balanced.[3] Similar to red-black trees, AVL trees are height-balanced. Both are in general not weight-balanced nor μ-balanced for any ; that is, sibling nodes can have hugely differing numbers of descendants.

Operations

Basic operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications are followed by zero or more operations called tree rotations, which help to restore the height balance of the sub trees.

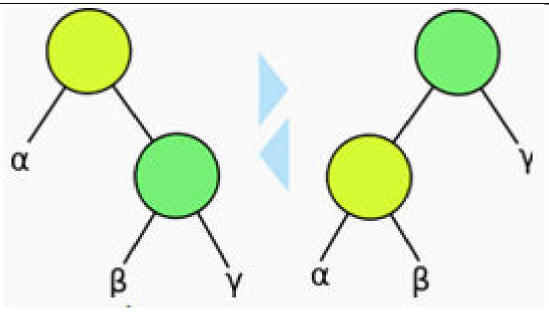


Figure: Tree rotations.

Searching

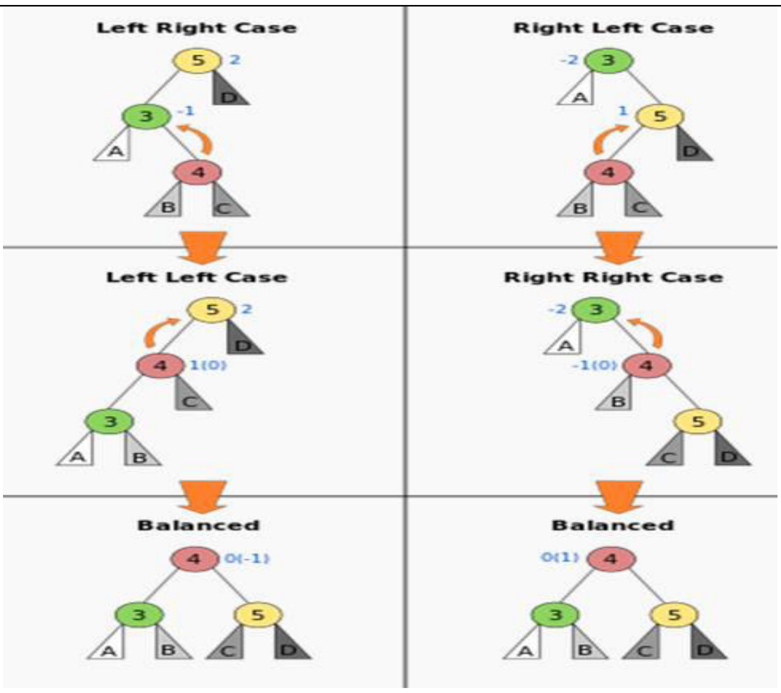
Searching for a specific key in an AVL Tree can be done the same way as that of a normal unbalanced Binary Search Tree.

Traversal

Once a node has been found in a balanced tree, the next or previous nodes can be explored in amortized constant time. Some instances of exploring these "nearby" nodes require traversing up to log (n) links (particularly when moving from the rightmost leaf of the root's left sub tree to the root or from the root to the leftmost leaf of the root's right sub tree; in the example AVL tree, moving from node 14 to the next but one node 19 takes 4 steps). However, exploring all n nodes of the tree in this manner would use each link exactly twice: one traversal to enter the sub tree rooted at that node, another to leave that node's sub tree after having explored it. And since there are n-1 links in any tree, the amortized cost is found to be 2× (n-1)/n, or approximately 2.

Insertion

Pictorial description of how rotations rebalance an AVL tree. The numbered circles represent the nodes being rebalanced. The lettered triangles represent sub trees which are themselves balanced AVL trees. A blue number next to a node denotes possible balance factors (those in parentheses occurring only in case of deletion).



After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL ("retracing"). The balance factor is calculated as follows:

Balance Factor = height (left sub tree) - height (right sub tree)

Since with a single insertion the height of an AVL sub tree cannot increase by more than one, the temporary balance factor of a node will be in the range from -2 to +2. For each node checked, if the balance factor remains in the range from -1 to +1 then only corrections of the balance factor, but no rotations are necessary. However, if the balance factor becomes less than -1 or greater than +1, the sub tree rooted at this node is unbalanced.

Description of the Rotations

Let us first assume the balance factor of a node P is 2 (as opposed to the other possible unbalanced value -2). This case is depicted in the left column of the illustration with P: =5. We then look at the left sub tree (the higher one) with root N. If this sub tree does not lean to the right - i.e. N has balance factor 1 (or, when deletion also 0) - we can rotate the whole tree to the right to get a balanced tree. This is labelled as the "Left Left Case" in the illustration with N: =4. If the sub tree does lean to the right - i.e. N: =3 has balance factor -1 - we first rotate the sub tree to the left and end up the previous case. This second case is labelled as "Left Right Case" in the illustration.

If the balance factor of the node P is -2 (this case is depicted in the right column of the illustration P: =3) we can mirror the above algorithm. I.e. if the root N of the (higher) right sub tree has balance factor -1 (or, when deletion also 0) we can rotate the whole tree to the left to get a balanced tree. This is labelled as the "Right Right Case" in the illustration with N: =4. If the root N: =5 of the right sub tree has balance factor 1 ("Right Left Case") we can rotate the sub tree to the right to end up in the "Right Right Case".

After a rotation a sub tree has the same height as before, so retracing can stop. In order to restore

the balance factors of all nodes, first observe that all nodes requiring correction lie along the path used during the initial insertion. If the above procedure is applied to nodes along this path, starting from the bottom (i.e. the inserted node), then every node in the tree will again have a balance factor of -1, 0, or 1.
The time required is O (log n) for lookup, plus a maximum of O (log n) retracing levels on the way back to the root, so the operation can be completed in O (log n) time.

Deletion

Let node X be the node with the value we need to delete, and let node Y be a node in the tree we need to find to take node X's place, and let node Z be the actual node we take out of the tree.

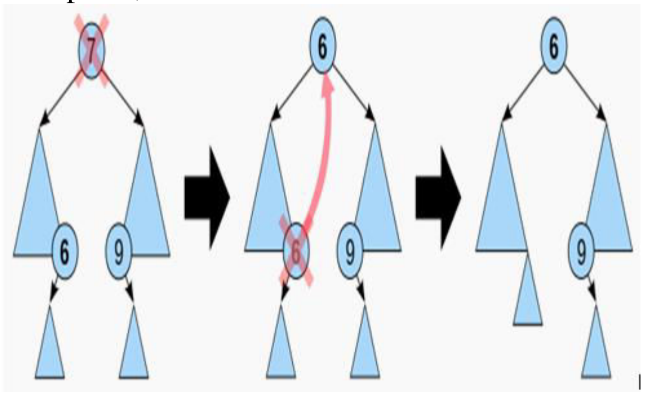


Figure: Deleting a node with two children from a binary search tree using the in-order predecessor (rightmost node in the left sub tree, labelled 6).

Steps to consider when deleting a node in an AVL tree are the following:

1. If node X is a leaf or has only one child, skip to step 5 with Z: =X.
2. Otherwise, determine node Y by finding the largest node in node X's left sub tree (the in-order predecessor of X - it does not have a right child) or the smallest in its right sub tree (the in-order successor of X - it does not have a left child).
3. Exchange all the child and parent links of node X with those of node Y. In this step, the in-order sequence between nodes X and Y is temporarily disturbed, but the tree structure doesn't change.
4. Choose node Z to be all the child and parent links of old node Y = those of new node X.
5. If node Z has a sub tree (which then is a leaf) attach it to Z's parent.
6. If node Z was the root (its parent is null), update root.
7. Delete node Z.
8. Retrace the path back up the tree (starting with node Z's parent) to the root, adjusting the balance factors as needed.

Since with a single deletion the height of an AVL sub tree cannot decrease by more than one, the temporary balance factor of a node will be in the range from -2 to +2.
If the balance factor becomes ±2 then the sub tree is unbalanced and needs to be rotated. The various cases of rotations are depicted in section "Insertion".

Unit-05/Lecture-04

B-Trees

Introduction

In computer science, a B-tree is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children (Comer 1979, p. 123). Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write large blocks of data. It is commonly used in databases and file systems.

Definition
According to *Knuth's definition*, a B-tree of order m is a tree which satisfies the following properties:
1. Every node has at most m children.
2. Every non-leaf node (except root) has at least? M/2? Children.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with k children contains k-1 keys.
5. All leaves appear in the same level

Each internal node's keys act as separation values which divide its sub trees. For example, if an internal node has 3 child nodes (or sub trees) then it must have 2 keys: a1 and a2. All values in the leftmost sub tree will be less than a1, all values in the middle sub tree will be between a1 and a2, and all values in the rightmost sub tree will be greater than a2.

Variants

The term B-tree may refer to a specific design or it may refer to a general class of designs. In the narrow sense, a B-tree stores key in its internal nodes but need not store those keys in the records at the leaves. The general class includes variations such as the B+ tree and the B*.

•In the B+ tree, copies of the keys are stored in the internal nodes; the keys and records are stored in leaves; in addition, a leaf node may include a pointer to the next leaf node to speed sequential access (Comer 1979, p. 129).

•The B*-tree balances more neighbouring internal nodes to keep the internal nodes more densely packed (Comer 1979, p. 129). This variant requires non-root nodes to be at least 2/3 full instead of 1/2 (Knuth 1998, p. 488). To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with a node next to it. When both nodes are full, then the two nodes are split into three. Deleting nodes is somewhat more complex than inserting however.

•B-trees can be turned into order statistic trees to allow rapid searches for the Nth record in key order, or counting the number of records between any two records, and various other related operations.

In particular, a B-tree:
- keeps keys in sorted order for sequential traversing
- uses a hierarchical index to minimize the number of disk reads
- uses partially full blocks to speed insertions and deletions

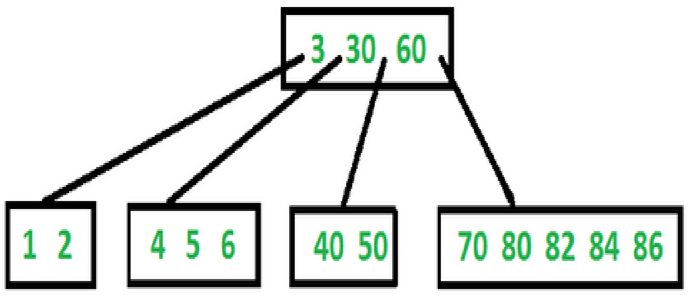
- keeps the index balanced with an elegant recursive algorithm

In addition, a B-tree minimizes waste by making sure the interior nodes are at least half full. A B-tree can handle an arbitrary number of insertions and deletions.

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 key.
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).

Following is **an example B-Tree of minimum degree 3**.



Note that in practical B-Trees, the value of minimum degree is much more than 3.

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.

Traverse

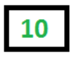
Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

Let us understand the algorithm with an example tree of minimum degree ‘t’ as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.

Insertion

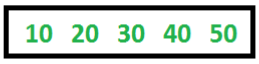
Initially root is NULL. Let us first insert 10.

Insert 10



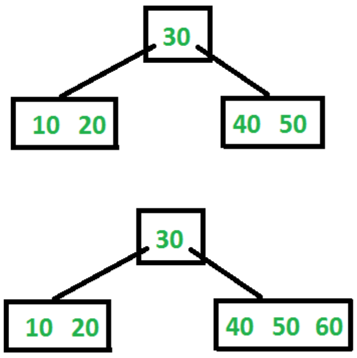
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because maximum number of keys a node can accommodate is 2*t – 1 which is 5.

Insert 20, 30, 40 and 50



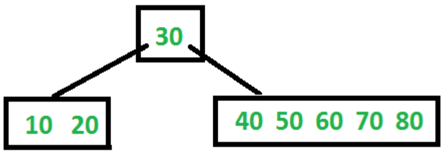
Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.

Insert 60



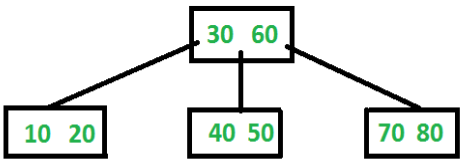
Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.

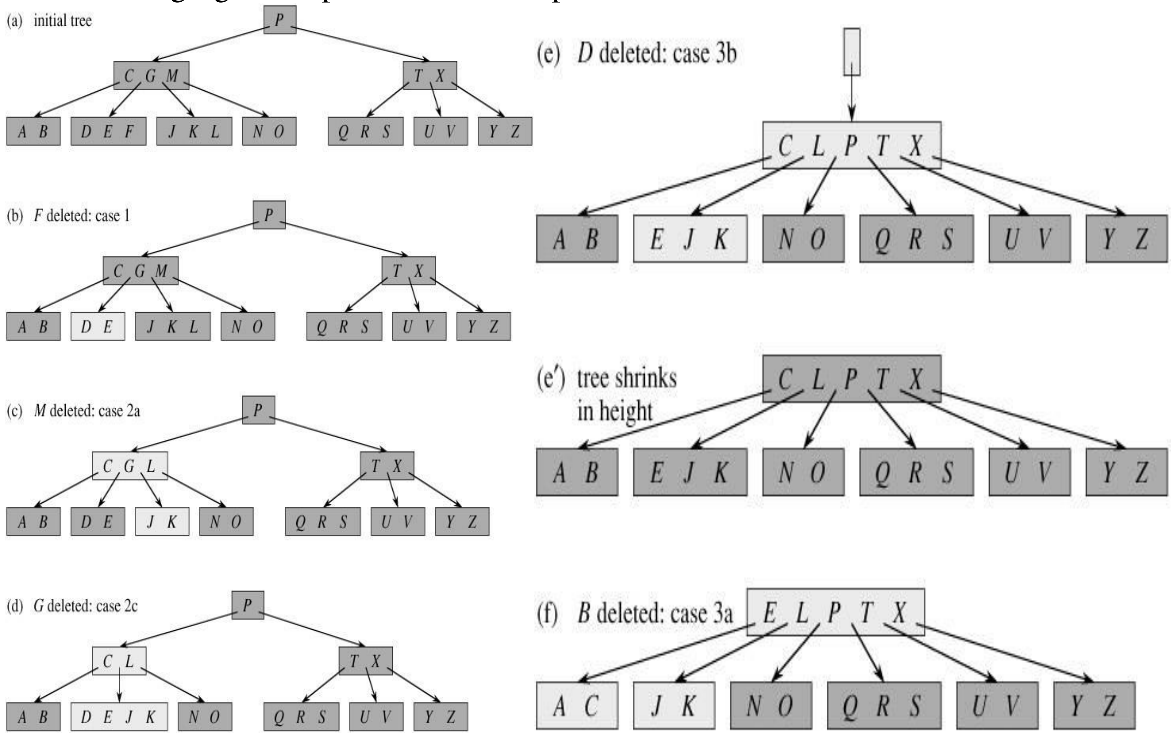
Insert 90



Deletion process

Deletion from a B-tree is more complicated than insertion, because we can delete a key from any node-not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children.

The following figures explain the deletion process.



S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			
Q.3			

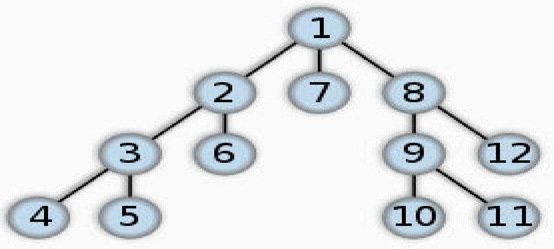
Unit-05/Lecture-05

Depth-first search (DFS)

Introduction : Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.
A version of depth-first search was investigated in the 19th century by French mathematician Charles Pierre Trémaux as a strategy for solving mazes.
Depth first search is another way of traversing graphs, which is closely related to preorder traversal of a tree. Recall that preorder traversal simply visits each node before its children. It is most easy to program as a recursive routine:

```
Preorder (node v)
{
visit(v);
for each child w of v
    preorder(w);
}
```

Depth-first search



Order in which the nodes are visited

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(\lvert E\rvert)$ for explicit graphs traversed without repetition, $O(b^d)$ for implicit graphs with branching factor *b* searched to depth *d*
Worst case space complexity	$O(\lvert V\rvert)$ if entire graph is traversed without repetition, O(longest path length searched) for implicit graphs without elimination of duplicate nodes

To turn this into a graph traversal algorithm, we basically replace "child" by "neighbor". But to prevent infinite loops, we only want to visit each vertex once. Just like in BFS we can use marks to keep track of the vertices that have already been visited, and not visit them again. Also, just like in BFS, we can use this search to build a spanning tree with certain useful properties.

```
dfs(vertex v)
{
visit(v);
for each neighbor w of v
    if w is unvisited
    {
    dfs(w);
    add edge vw to tree T
    }
}
```

The overall depth first search algorithm then simply initializes a set of markers so we can tell which vertices are visited, chooses a starting vertex x, initializes tree T to x, and calls dfs(x). Just like in breadth first search, if a vertex has several neighbours it would be equally correct to go through them in any order. I didn't simply say "for each unvisited neighbour of v" because it is very important to delay the test for whether a vertex is visited until the recursive calls for previous neighbours are finished.

The proof that this produces a spanning tree (the depth first search tree) is essentially the same as that for BFS, so I won't repeat it. However while the BFS tree is typically "short and bushy", the DFS tree is typically "long and stringy".

Just like we did for BFS, we can use DFS to classify the edges of G into types. Either an edge vw is in the DFS tree itself, v is an ancestor of w, or w is an ancestor of v. (These last two cases should be thought of as a single type, since they only differ by what order we look at the vertices in.) What this means is that if v and w are in different sub trees of v, we can't have an edge from v to w. This is because if such an edge existed and (say) v were visited first, then the only way we would avoid adding vw to the DFS tree would be if w were visited during one of the recursive calls from v, but then v would be an ancestor of w.

As an example of why this property might be useful, let's prove the following fact: in any graph G, either G has some path of length at least k. or G has O(kn) edges.

Proof: look at the longest path in the DFS tree. If it has length at least k, we're done. Otherwise, since each edge connects an ancestor and a descendant, we can bound the number of edges by counting the total number of ancestors of each descendant, but if the longest path is shorter than k, each descendant has at most k-1 ancestors. So there can be at most (k-1)n edges.

Applications

Algorithms that use depth-first search as a building block include:

- Finding connected components.
- Topological sorting.
- Finding 2-(edge or vertex)-connected components.
- Finding 3-(edge or vertex)-connected components.
- Finding the bridges of a graph.
- Generating words in order to plot the Limit Set of a Group.
- Finding strongly connected components.
- Planarity testing.
- Solving puzzles with only one solution, such as mazes.
  (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
- Maze generation may use a randomized depth-first search.
- Finding biconnectivity in graphs.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			
Q.3			

Unit-05/Lecture-06

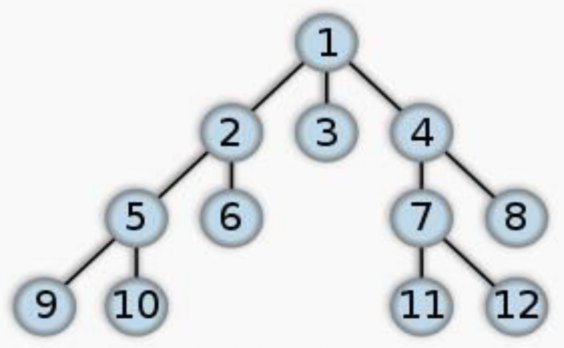
BFS

Introduction: Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a `search key') and explores the neighbour nodes first, before moving to the next level neighbours. Compare BFS with the equivalent, but more memory-efficient Iterative deepening depth-first search and contrast with depth-first search.

BFS was invented in the late 1950s by E. F. Moore, who used it to find the shortest path out of a maze, and discovered independently by C. Y. Lee as a wire routing algorithm (published 1961).

This can be thought of as being like Dijkstra's algorithm for shortest paths, but with every edge having the same length. However it is a lot simpler and doesn't need any data structures. We just keep a tree (the breadth first search tree), a list of nodes to be added to the tree, and markings (Boolean variables) on the vertices to tell whether they are in the tree or list.

Breadth-first search


Order in which the nodes are expanded

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(\lvert E\rvert) = O(b^d)$
Worst case space complexity	$O(\lvert V\rvert) = O(b^d)$

breadth first search:

```
unmark all vertices
choose some starting vertex x
mark x
list L = x
tree T = x
while L nonempty
choose some vertex v from front of list
visit v
for each unmarked neighbor w
    mark w
    add it to end of list
    add edge vw to T
```

It's very important that you remove vertices from the other end of the list than the one you add them to, so that the list acts as a queue (FIFO storage) rather than a stack (LIFO). The "visit v" step would be filled out later depending on what you are using BFS for, just like the tree traversals usually involve doing something at each vertex that is not specified as part of the basic algorithm. If a vertex has several unmarked neighbours, it would be equally correct to visit them in any order. Probably the easiest method to implement would be simply to visit them in the order the adjacency list for v is stored in.

Let's prove some basic facts about this algorithm. First, each vertex is clearly marked at most once, added to the list at most once (since that happens only when it's marked), and therefore removed from the list at most once. Since the time to process a vertex is proportional to the length of its adjacency list, the total time for the whole algorithm is O(m).

Next, let's look at the tree T constructed by the algorithm. Why is it a tree? If you think of each edge vw as pointing "upward" from w to v, then each edge points from a vertex visited later to one visited earlier. Following successive edges upwards can only get stopped at x (which has no edge going upward from it) so every vertex in T has a path to x. This means that T is at least a connected subgraph of G. Now let's prove that it's a tree. A tree is just a connected and acyclic graph, so we need only to show that T has no cycles. In any cycle, no matter how you orient the edges so that one direction is "upward" and the other "downward", there is always a "bottom" vertex having two upward edges out of it. But in T, each vertex has at most one upward edge, so T can have no cycles. Therefore T really is a tree. It is known as a breadth first search tree.

We also want to know that T is a spanning tree, i.e. that if the graph is connected (every vertex has some path to the root x) then every vertex will occur somewhere in T. We can prove this by induction on the length of the shortest path to x. If v has a path of length k, starting v-w-...-x, then w has a path of length k-1, and by induction would be included in T. But then when we visited w we would have seen edge vw, and if v were not already in the tree it would have been added.

Breadth first traversal of G corresponds to some kind of tree traversal on T. But it isn't preorder, postorder, or even inorder traversal. Instead, the traversal goes a level at a time, left to right within a level (where a level is defined simply in terms of distance from the root of the tree). For instance, the following tree is drawn with vertices numbered in an order that might be followed by breadth first search:

```
    1
   /|\
  2 3 4
   / \  |
  5    6 7
  |   /|\
  8   9 10 11
```

The proof that vertices are in this order by breadth first search goes by induction on the level number. By the induction hypothesis, BFS lists all vertices at level k-1 before those at level k. Therefore it will place into L all vertices at level k before all those of level k+1, and therefore so list those of level k before those of level k+1. (This really is a proof even though it sounds like circular reasoning.)

Breadth first search trees have a nice property:

Every edge of G can be classified into one of three groups. Some edges are in T themselves. Some connect two vertices at the same level of T. And the remaining ones connect two vertices on two adjacent levels. It is not possible for an edge to skip a level.

Therefore, the breadth first search tree really is a shortest path tree starting from its root. Every vertex has a path to the root, with path length equal to its level (just follow the tree itself), and no path can skip a level so this really is a shortest path.

Breadth first search has several uses in other graph algorithms, but most are too complicated to explain in detail here. One is as part of an algorithm for matching, which is a problem in which you want to pair up the n vertices of a graph by n/2 edges. If you have a partial matching, pairing up only some of the vertices, you can extend it by finding an alternating path connecting two unmatched vertices; this is a path in which every other edge is part of the partial matching. If you remove those edges in the path from the matching, and add the other path edges back into the matching, you get a matching with one more edge. Alternating paths can be found using a version of breadth first search.

A second use of breadth first search arises in certain pattern matching problems. For instance, if you're looking for a small sub graph such as a triangle as part of a larger graph, you know that every vertex in the triangle has to be connected by an edge to every other vertex. Since no edge can skip levels in the BFS tree, you can divide the problem into sub problems, in which you look for the triangle in pairs of adjacent levels of the tree. This sort of problem, in which you look for a small graph as part of a larger one, is known as sub graph isomorphism. In a recent paper, I used this idea to solve many similar pattern-matching problems in linear time.

Applications
Breadth-first search can be used to solve many problems in graph theory, for example:

- Copying Collection, Cheney's algorithm
- Finding the shortest path between two nodes u and v (with path length measured by number of edges)
- Testing a graph for bipartiteness
- (Reverse) Cuthill–McKee mesh numbering
- Ford–Fulkerson method for computing the maximum flow in a flow network
- Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.
- Construction of the failure function of the Aho-Corasick pattern matcher.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			
Q.3			

Unit-05/Lecture-07

Binary search Tree

Introduction

In computer science, binary search trees (BST), sometimes called ordered or sorted binary trees, are a class of data structures used to implement lookup tables and dynamic sets. They store data items, known as keys, and allow fast insertion and deletion of such keys, as well as checking whether a key is present in a tree.

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right sub trees. On average, this means that each comparison allows the operations to skip over half of the tree, so that each lookup/insertion/deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an unsorted array, but slower than the corresponding operations on hash tables

Definition
A binary search tree is a node-based binary tree data structure where each node has a comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left sub-tree and smaller than the keys in all nodes in that node's right sub-tree. Each non-leaf node has exactly two child nodes. Each child must either be a leaf node or the root of another binary search tree. BSTs are also dynamic data structures, and the size of a BST is only limited by the amount of free memory in the operating system. The main advantage of binary search trees is that it remains ordered, which provides quicker search times than many other data structures. The common properties of binary search trees are as follows:

- One node is designated the root of the tree.
- Each internal node contains a key and has two sub trees.
- The leaves (final nodes) of the tree contain no key. Leaves are commonly represented by a special leaf or nil symbol, a NULL pointer, etc.
- Each sub tree is itself a binary search tree.
- The left sub tree of a node contains only nodes with keys strictly less than the node's key.
- The right sub tree of a node contains only nodes with keys strictly greater than the node's key.

Operations

Searching
Searching a binary search tree for a specific key can be a recursive or an iterative process.
We begin by examining the root node. If the tree is null, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is null. If the searched key is not found before a null subtree is reached, then the item must not be present in the tree. This is easily expressed as a recursive algorithm:

```
function Find-recursive(key, node):  // call initially with node = root
   if node = Null or node.key = key then
      return node
   else if key < node.key then
      return Find-recursive(key, node.left)
   else
      return Find-recursive(key, node.right)
```

The same algorithm can be implemented iteratively:

```
function Find(key, root):
   current-node := root
   while current-node is not Null do
      if current-node.key = key then
         return current-node
      else if key < current-node.key then
         current-node ← current-node.left
      else
         current-node ← current-node.right
   return Null
```

Because in the worst case this algorithm must search from the root of the tree to the leaf farthest from the root, the search operation takes time proportional to the tree's height(see tree terminology). On average, binary search trees with n nodes have O(log n) height. However, in the worst case, binary search trees can have O(n) height, when the unbalanced tree resembles a linked list (degenerate tree).

Insertion

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right sub trees as before. Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'new Node') as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left sub tree if its key is less than that of the root, or the right sub tree if its key is greater than or equal to the root.

Here's how a typical binary search tree insertion might be performed in a binary tree in C++:

```
void insert(Node*& root, int data)
 {
  if (!root)
    root = new Node(data);
  else if (data < root->data)
    insert(root->left, data);
  else if (data > root->data)
    insert(root->right, data);
}
```

The above destructive procedural variant modifies the tree in place. It uses only constant heap space (and the iterative version uses constant stack space as well), but the prior version of the tree is lost. Alternatively, as in the following Python example, we can reconstruct all ancestors of the inserted node; any reference to the original tree root remains valid, making the tree a persistent data structure:

```
 def binary_tree_insert(node, key, value):
     if node is None:
         return TreeNode(None, key, value, None)
     if key == node.key:
         return TreeNode(node.left, key, value, node.right)
     if key < node.key:
         return TreeNode(binary_tree_insert(node.left, key, value), node.key, node.value, node.right)
     else:
         return TreeNode(node.left, node.key, node.value, binary_tree_insert(node.right, key, value))
```

The part that is rebuilt uses O(log n) space in the average case and O(n) in the worst case (see big-O notation).

In either version, this operation requires time proportional to the height of the tree in the worst case, which is O(log n) time in the average case over all trees, but O(n) time in the worst case. Another way to explain insertion is that in order to insert a new node in the tree, its key is first compared with that of the root. If its key is less than the root's, it is then compared with the key of the root's left child. If its key is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its key.

There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

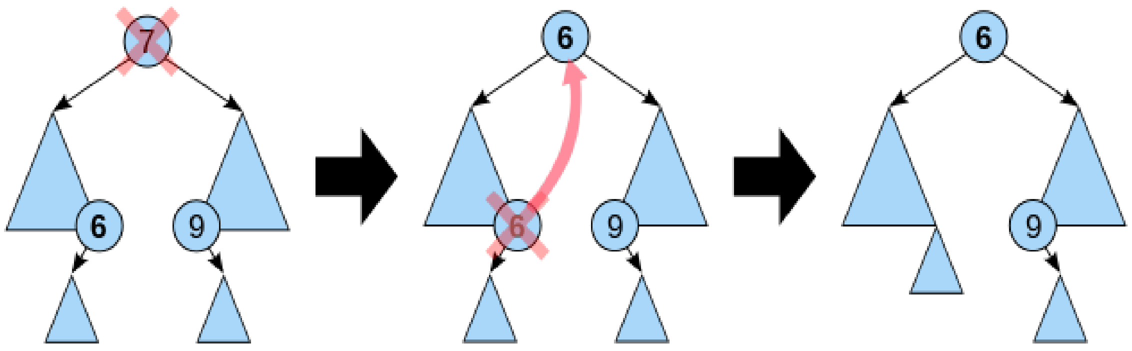
Deletion

There are three possible cases to consider:

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted N. Do not delete N.

Instead, choose either its in-order successor node or its in-order predecessor node, R. Copy the value of R to N, then recursively call delete on R until reaching one of the first two cases. If you choose in-order successor of a node, as right sub tree is not NIL (Our present case is node has 2 children), then its in-order successor is node with least value in its right sub tree, which will have at a maximum of 1 sub tree, so deleting it would fall in one of first 2 cases.

Broadly speaking, nodes with children are harder to delete. As with all binary trees, a node's in-order successor is its right subtree's left-most child, and a node's in-order predecessor is the left subtree's right-most child. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.



Deleting a node with two children from a binary search tree. First the rightmost node in the left sub tree, the inorder predecessor 6, is identified. Its value is copied into the node being deleted. The inorder predecessor can then be easily deleted because it has at most one child.
The same method works symmetrically using the inorder successor labelled 9.

Consistently using the in-order successor or the in-order predecessor for every instance of the two-child case can lead to an unbalanced tree, so some implementations select one or the other at different times.

Runtime analysis: Although this operation does not always traverse the tree down to a leaf, this is always a possibility; thus in the worst case it requires time proportional to the height of the tree. It does not require more even when the node has two children, since it still follows a single path and does not visit any node twice.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			
Q.3			

Unit-05/Lecture-08

2-3 Tree

Introduction

In computer science, a 2–3 tree is a tree data structure, where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements. Nodes on the outside of the tree (leaf nodes) have no children and one or two data elements.

2−3 trees were invented by John Hopcroft in 1970.

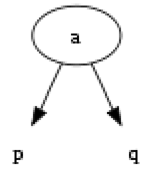


Figure : 2 Node

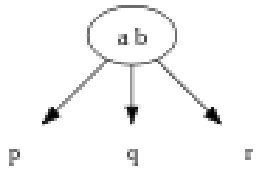


Figure : 3 Node

2–3 trees are an isometry of AA trees, meaning that they are equivalent data structures. In other words, for every 2–3 tree, there exists at least one AA tree with data elements in the same order. 2–3 trees are balanced, meaning that each right, center, and left subtree contains the same or close to the same amount of data.

Properties

- Every internal node is a 2-node or a 3-node.
- All leaves are at the same level.
- All data is kept in sorted order.

Operations

Searching

Searching for an item in a 2-3 tree is similar to searching for an item in a binary search tree. Since the data element in each node is ordered, a search function will be directed to the correct sub tree and eventually to the correct node which contains the item.

1. Let T be a 2-3 tree and d be the data element we want to find. If T is empty, then d is not in T and we're done.

2. Let r be the root of T.

3. Suppose r is a leaf. If d is not in r, then d is not in T. Otherwise, d is in T. In particular, d can be found at a leaf node. We need no further steps and we're done.

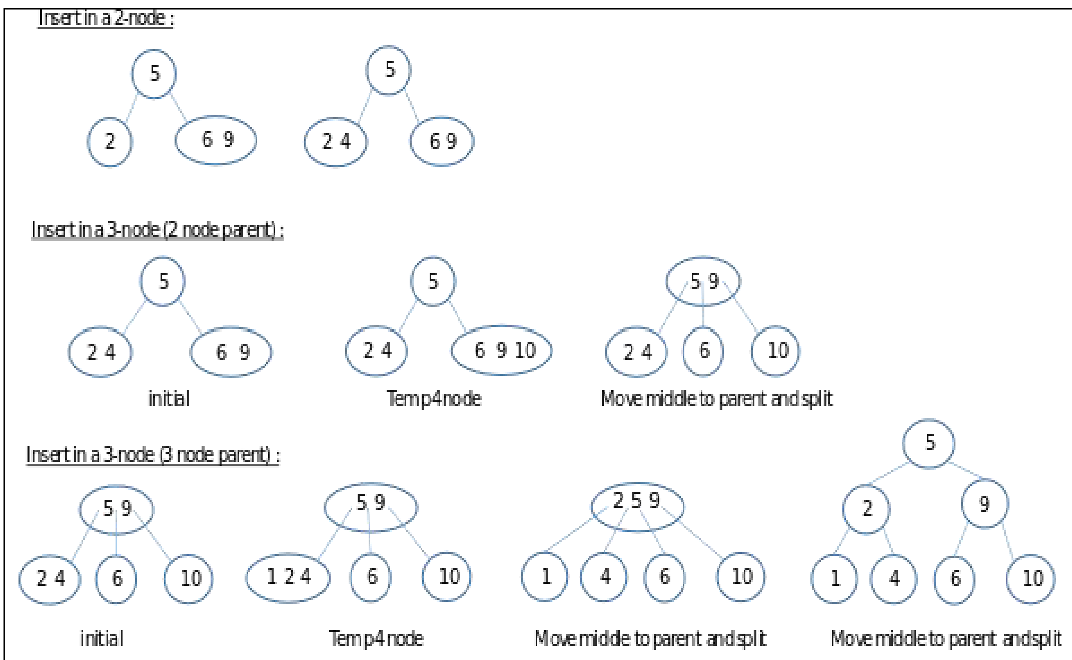
4. Suppose r is a 2-node with left child L and right child R. Let e be the data element in r. There are three cases: If d is equal to e, then we've found d in T and we're done. If, then set T to L, which by definition is a 2-3 tree, and go back to step 2. If, then set T to R and go back to step 2.

5. Suppose r is a 3-node with left child L, middle child M, and right child R. Let a and b be the two data elements of r, where. There are four cases: If d is equal to aor b, then d is in T and we're done. If , then set T to L and go back to step 2. If, then set T to M and go back to step 2. If, then set T to R and go back to step 2.

Insertion

Insertion works by searching for the proper location of the key and adds it there. If the node becomes a 4-node then the node is split from two 2-nodes and the middle key is moved up to the parent.

The diagram illustrates the process.


Insert in a 2-node :
5
2
6 9
5
2 4
6 9
Insert in a 3-node (2 node parent) :
5
2 4
6 9
initial
5
2 4
6 9 10
Temp4node
5 9
2 4
6
10
Move middle to parent and split
Insert in a 3-node (3 node parent) :
5 9
2 4
6
10
initial
5 9
1 2 4
6
10
Temp4 node
2 5 9
1
4
6
10
Move middle to parent and split
5
2
9
1
4
6
10
Move middle to parent and split

Insertion of a number in a 2-3 tree for the 3 possible cases.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			
Q.3			