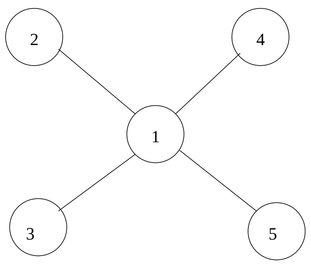
COT5405 Analysis of Algorithms
Homework 3
Solutions

1. Prove or give a counter example:

   (a) In the textbook, we have two routines for graph traversal - DFS(G) and BFS(G,s) - where G is a graph and s is any node in G. These two procedures will create a DFS tree and a BFS tree respectively. If $G = (V, E)$ is a connected, undirected graph then the height of DFS(G) tree is always larger than or equal to the height of any of the BFS trees created by BFS(G, s).
   Answer



   Counter example:
   The DFS can start from any arbitrary node. So we pick the center node. For BFS we pick any of the boundary nodes.
   Then the height of DFS tree is 1, whereas the height of BFS tree is 2.
   For this problem, if you make an assumption that DFS and BFS always start at the same node and prove that the height of the DFS tree is always larger than that of BFS tree, then also you'll get credit – if your proof is good.

   (b) BFS and DFS algorithms on a undirected, connected graph $G = (V, E)$, produce the same tree if and only if G is a tree.
   Answer
   First we show that G is a tree when both BFS-tree and DFS-tree are same.
   If G and T are not same, then there should exist an edge $e(u, v)$ in G, that does not belong to T.
   In such a case:

– in the DFS-tree, one of the u or v, should be an ancestor of the other.

– in the BFS-tree, u and v can differ by only one level.

Since, both DFS-tree and BFS-tree are same tree T, it follows that one of u and v should be an ancestor of the other and they can differ by only one level. This implies that the edge connecting them must be in T.

So, there can not be any edges in G which are not in T.

In the second part of the proof:

Since G is a tree, each node has a unique path from the root. So, both BFS and DFS produce the same tree, and the tree is same as G.

2. When an adjacency-matrix representation is used, most graph algorithms require time $\Omega(V^2)$, but there are some exceptions. Show that determining whether a directed graph G contains a **universal sink** – a vertex with in-degree $|V| - 1$ and out-degree 0 – can be determined in time $O(V)$, given an adjacency matrix for G.

   Answer We are looking for a universal sink i.e. a vertex in degree of $\mid V \mid -1$ and out degree of zero. In the adjacent matrix the elements the element in row i represent the edges that are out-going from node i to other nodes. And the elements in column j represent the edges that are in-coming to node j. The task is to find k, such that row k has all zero's and column k has all $1's$ except for the element in row k (which is a zero).

   Start traversing the first row and stop at the first 1 that is encountered. this will take $O(V)$ steps in the worst case. If no 1 is encountered then this row is the only possible candidate for a universal sink (because it does not have an edge to any other node, no other node can be a universal sink), we can simply check by traversing the first column in $O(V)$ time and see if it has all 1's. if so then node 1 is a universal sink, otherwise the graph has no universal sink. The algorithm terminates once we find a row of all zero's whether that row represents a universal sink or not, thus guaranteeing $O(V)$ running time.

   Now say that while traversing the first row we encounter the first 1 at column k. Then the elements 1 through $k-1$ in the first row have zeros. This means that no node in 1 through $k-1$ can be universal sink because node 1 does not have an edge to any of them. So, we have also eliminated node 1 because its row is not all zero's. We can do this elimination in $O(k)$ steps by examining the first row.

   Now, from the remaining nodes, we begin examining the next available row j and follow a procedure similar to the first row. if no 1 is encountered, we test the $j'th$ column for $1's$ and terminate as explained above. If a 1 is encountered after m zero's then we can eliminate these m nodes from being universal sink. Thus, it can be claimed that we have eliminated $k+m$ nodes in $O(k+m)$ time.

   Repeating the above procedure, examining one row at a time from vertices that have not yet been eliminated, we can find whether a universal sink exist or not. Thus, finding whether a graph has a universal sink or not, can be determined in $O(V)$ time.

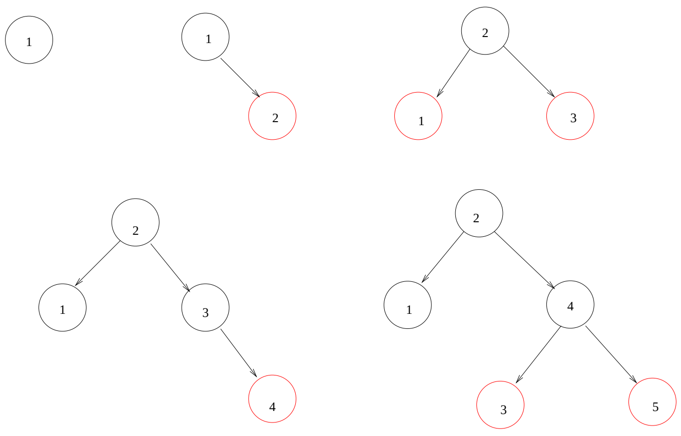
3. From Inorder,Preorder and Postorder traversal of a BST, what are the minimum number of traversals do you need to reconstruct the BST and what are those? Also give an efficient algorithm and

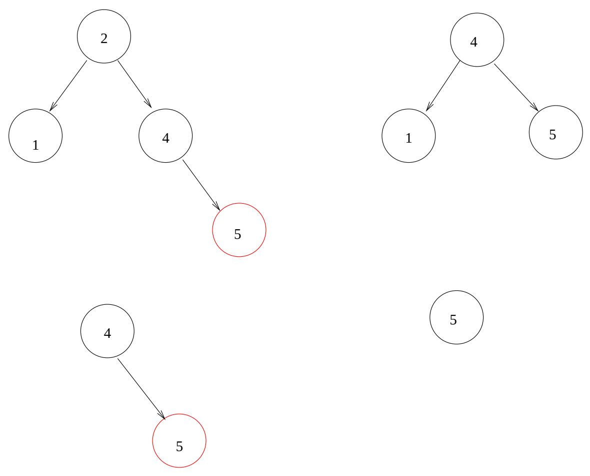
its complexity analysis for building the BST from those traversals.

Answer
For general binary tree, we need at least 2 traversals to reconstruct the binary tree. But here we have Binary Search Tree, so the Inorder is actually the sorted order of the keys at the nodes.
So for BST, sorted order is the Inorder indeed. So minimum number of traversal you need to reconstruct is 1.
So as Inorder is the sorted order, you need just another one tranversal along with it - either Preorder or Postorder.
Lets say you have the preorder traversal.
For the given Preorder you can sort these number - $O(n\ lg\ n)$. Then from the Preorder find the root, which will be the first node and search it in the Inorder. In the Inorder, the numbers left to that number form the left subtree and numbers right to it form the right subtree. Do it recursively to build the whole tree.
Recurrence for the algorithm:
$T(n)\ =\ T(n-1)\ +\ O(lg\ n)$
Solving the above recurrence, we have a solution of $O(n\ lg\ n)$.
If Postorder is used instead, then we will have a similar algorithm, but you have to start from the opposite end, because Postorder contains the root at the end.

4. Beginning with an empty Red-Black tree, insert 1,2,3,4,5 (in that particular order), and then delete 3,2,1,4,5 (in that particular order). Show the tree after each insertion or deletion.

   Answer





5. (a) Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

   Answer

   Deletion from a binary search tree is not commutative. Here is a counter example:

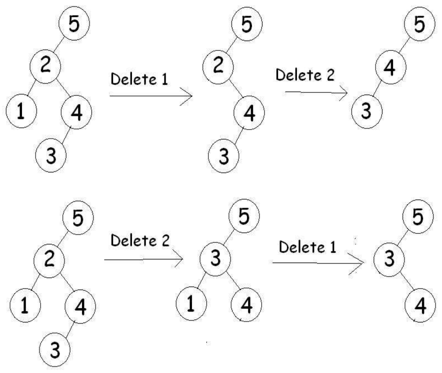


Figure 1: Counter Example to show that delete from binary search tree is not commutative

   (b) Show that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $O(n)$ rotations.

   Answer We can convert any binary search tree with n nodes into a right going chain of length n using at most $O(n)$ right rotation operations. If a node in the tree has a left subtree then we perform a right rotation on that tree node. There can be $O(n)$ such nodes, so we need at most n right rotations.

   By using similar argument we can prove that a right going chain of length n can be converted into any binary search tree with n nodes using as most n left rotations. Combining these two we observe that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $O(n)$ rotations.

(c) Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

Answer

Following figure is the counter example to disprove Professor's claim. Here, we assume that the search ended at 4. so, $A = \{2\}$, $B = \{1, 5, 3, 4\}$ and $C = \{6\}$.

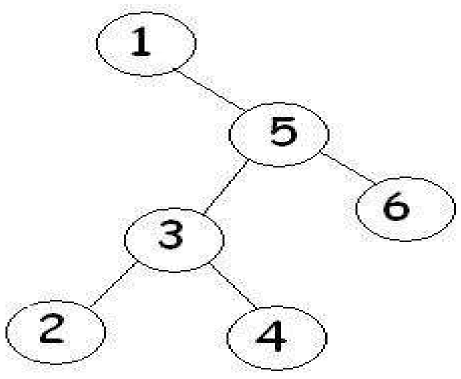


Figure 2: Counter Example to disprove professor's claim.

6. Describe a general way to construct red-black trees with the property that the height of the tree should be as close as to the number of nodes in the tree. Give a lower bound for height in terms of the number of internal nodes.

   Answer

   Consider the 2 element red-black tree with a black root and a single red left child. This will be the base case of our construction. As a general extension, we add another base case as the left child of the leftmost branch, and a single black node as a child everywhere there is no child. This clearly makes the black height of the tree one higher.

   Remember that in general the relation is $h \leq 2lg(n+1)$. Consider the base case. It has 3 sites at which we can add a child, and under our construction, we will add a black child at 2 of them and a base case at 1 of them. In general, if we have k sites for children, we add k-1 black children and 1 base case, thus adding k+1 new nodes. Each of these black children has 2 sites available for children, and the one new base case has 3 , thus giving us 2(k-1)+3, or 2k+1 sites at which we can attach children on the next iteration.

   This gives us the following recursuve definitions for height and number of nodes (with $h_0 = 2$, $k_0 = 3$, $n_0 = 2$):

$$h_{i+1} = h_i + 2$$

$$k_{i+1} = 2k_i + 1$$

$$n_{i+1} = n_i + k_i + 1$$

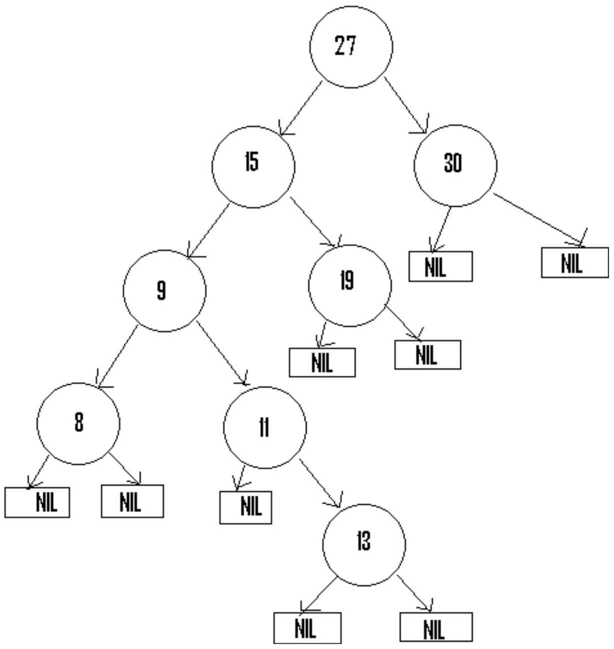
   Clearly $h(i) = 2i + 2$, $k(i) = 2^{i+2} - 1$. We wish to show that $n(i) = 2^{i+2} - 2$. We will do this inductively.
   Base case: $i = 0$. $n_0 = 2$; $2^{0+2} - 2 = 2$.
   Inductively:

$$\begin{aligned} n(i+1) &= n(i) + k(i) + 1 \\ &= (2^{i+2} - 2) + (2^{i+2} - 1) + 1 \\ &= 2.2^{i+2} - 2 \\ &= 2^{i+3} - 2 \end{aligned}$$

   Now, the question becomes: how does h(i) relate to n(i)? Well, $lg(n(i) + 2) = i + 2$, so $h(i) = 2lg(n(i) + 2) - 2$.

7. Is it possible to color the below binary tree as red black tree? Explain why.

   Answer



Proof by contradiction. Suppose that a valid coloring exists. In a red-black tree, all paths from a node to descendant leaves contain the same number of black nodes. The path (17, 19, NIL) can contain at most three black nodes. Therefore the path (17, 11, 3, 5, 7, NIL) must also contain at most three black nodes and at least three red nodes. By the red-black tree properties, the root 17 must be black and the NIL node must also be black. This means that there must be three red nodes in the path (11, 3, 5, 7), but this would mean there are two consecutive red nodes, which violates the red-black tree properties. This is a contradiction, therefore the tree cannot be colored to form a legal red-black tree.

8. A directed graph $G = (V, E)$ is said to be *semiconnected* if, for all pairs $u, v \in V$,we have $u \leadsto v$ or $v \leadsto u$. Give an efficient algorithm to determine whether or not G is semiconnected. Prove that your algorithm is correct, and analyze its running time.
Answer

In order to determine whether a graph $G = (V, E)$ is semiconnected, we apply following steps:

(a) Call STRONGLY-CONNECTED-COMPONENTS
(b) Form the component graph. (By Exercise 22.5-5).
(c) Topologically sort the component graph. It is possible to topologically sort the component graph, because component graph is always a DAG (Directed Acyclic Graph).
(d) Verify that the sequence of vertices $(v_1, v_2, ..., v_k)$ given by topological sort forms a linear chain in the component graph. That is, verify that the edges (v_1, v_2), (v_2, v_3),...,(v_{k-1}, v_k) exist in the component graph. If the vertices form a linear chain, then the original graph is semiconnected; otherwise it is not.

Because we know that all vertices in each SCC are mutually reachable from each other, it suffices to show that the component graph is semiconnected if and only if it contains a linear chain.

Total running time is $\theta(V + E)$.