Briefing Document: Binary Search Trees
Source: Excerpts from "C12-bst.pdf", Chapter 12: Binary Search Trees

Date: October 26, 2023

Overview: This document provides a summary of the key concepts and operations related to Binary Search Trees (BSTs) as described in the provided chapter excerpts. It covers the fundamental properties of BSTs, different traversal methods, common operations (search, minimum, maximum, insertion, successor, predecessor, deletion), and a brief analysis of the average height of a randomly built BST.

Main Themes and Important Ideas/Facts:

1. Definition and Properties of a Binary Search Tree:

A Binary Search Tree is a binary tree with a specific property: "For all nodes x and y, if y belongs to the left subtree of x, then the key at y is less than the key at x, and if y belongs to the right subtree of x, then the key at y is greater than the key at x."
It is assumed that all keys within a BST are pairwise distinct.
Each node in a BST has attributes:
p: pointer to the parent node.
left: pointer to the left child node.
right: pointer to the right child node.
key: the value stored at the node.
2. Traversal of Nodes in a BST:

Traversal refers to visiting all nodes in the tree.
Three main traversal strategies are defined based on the order of visiting the left subtree, the current node, and the right subtree (with the left subtree always visited before the right subtree):
Inorder: Left subtree, Current node, Right subtree.
Preorder: Current node, Left subtree, Right subtree.
Postorder: Left subtree, Right subtree, Current node.
Pseudocode for Inorder Traversal is provided:
Inorder-Walk(x)
1: if x = nil then return
2: Inorder-Walk(left[x])
3: Print key[x]
4: Inorder-Walk(right[x])
Applying these traversals to the example BST yields the following results:
Inorder: 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 19, 20. Notably, "inorder travel on a BST finds the keys in nondecreasing order!"
Preorder: 7, 4, 2, 3, 6, 5, 12, 9, 8, 11, 19, 15, 20.
Postorder: 3, 2, 5, 6, 4, 8, 11, 9, 15, 20, 19, 12, 7.
3. Operations on BST:

Searching for a key:Leverages the BST-property to efficiently search for a given key.
Starts at the root and compares the target key with the current node's key.
If the target key is smaller, the search continues in the left subtree; if larger, in the right subtree.
The algorithm BST-Search(x, k) is provided, which iteratively traverses the tree until the key is found or a nil pointer is encountered (indicating "NOT FOUND").
The Maximum and the Minimum:The minimum key is found by traversing only left branches until a node with no left child is reached (the leftmost node).
The maximum key is found by traversing only right branches until a node with no right child is reached (the rightmost node).
Pseudocode for BST-Minimum(x) and BST-Maximum(x) is provided.
Insertion:A new node z with key k is inserted by finding an appropriate nil position such that the BST-property is maintained.
The BST-Insert(x, z, k) algorithm outlines a process similar to search to find the insertion point.
The Successor and The Predecessor:The successor of a key k is the smallest key in the BST that is strictly greater than k. The predecessor is the largest key strictly less than k.
Algorithm for finding the successor of a node x (BST-Successor(x)):
If x has a right child, the successor is the minimum element in the right subtree.
Otherwise, the successor is the parent of the farthest node reachable from x by following only right branches backward.
Deletion:Deleting a node z involves three cases:
z has no children: Replace z with nil.
z has only one child: Promote the child to z's position.
z has two children:Find z's successor y.
y will either be a leaf or have only a right child.
Promote y to z's position.
Remove y using one of the first two cases.
The BST-Delete(T, z) algorithm provides a detailed implementation of this process, including finding the successor and handling different cases based on the number of children of the node to be removed and its successor.
4. Efficiency Analysis:

Theorem A: "On a binary search tree of height h, Search, Minimum, Maximum, Successor, Predecessor, Insert, and Delete can be made to run in O(h) time." This highlights the importance of the tree's height for the performance of these operations.
5. Randomly Built BST:

The chapter briefly discusses the average height of a BST built by inserting n distinct keys in a random order (where all n! permutations are equally likely).
The height Xn is analyzed using the random variable Yn = 2Xn.
Through mathematical derivation involving expected values and induction, it is shown that the expected value of Yn, E[Yn], has an upper bound.

Applying Jensen's inequality, it is concluded that the average height of a randomly built BST is O(logn). This signifies that, on average, the operations on a randomly built BST will have a logarithmic time complexity with respect to the number of nodes.
Quotes:

"For all nodes x and y, if y belongs to the left subtree of x, then the key at y is less than the key at x, and if y belongs to the right subtree of x, then the key at y is greater than the key at x." (Describing the BST-property)
"inorder travel on a BST finds the keys in nondecreasing order!" (Highlighting the property of inorder traversal)
"On a binary search tree of height h, Search, Minimum, Maximum, Successor, Predecessor, Insert, and Delete can be made to run in O(h) time." (Stating the time complexity of key BST operations)
This briefing document summarizes the fundamental aspects of Binary Search Trees covered in the provided chapter, emphasizing their structure, traversal methods, core operations, and the efficiency of these operations in relation to the tree's height, particularly in the context of randomly built BSTs.