0.1 Introduction

This section was studied by L.N.Quang. Some most common operation performed on a binary search tree are searching algorithms,insert algorithm and delete algorithm.

0.2 Algorithm

In this section we shall first present the search operation and show that this operation can be supported in time $O(h)$ on a binary search tree of height h. Then we will discuss how the operations of insertion and deletion cause the dynamic set of a binary search tree change but still remain its properties.As we shall see, modifying the tree to insert a new element is relatively straightforward, but handling deletion is somewhat more intricate.

0.2.1 Searching algorithm

Given a pointer to the root of the tree and a key k, searching algorithm will return a pointer to a node with key k if one exists, otherwise it returns nothing (or $NULL$). The procedure begins its search at the root and traces a path downward in the tree, as shown in Figure 1.

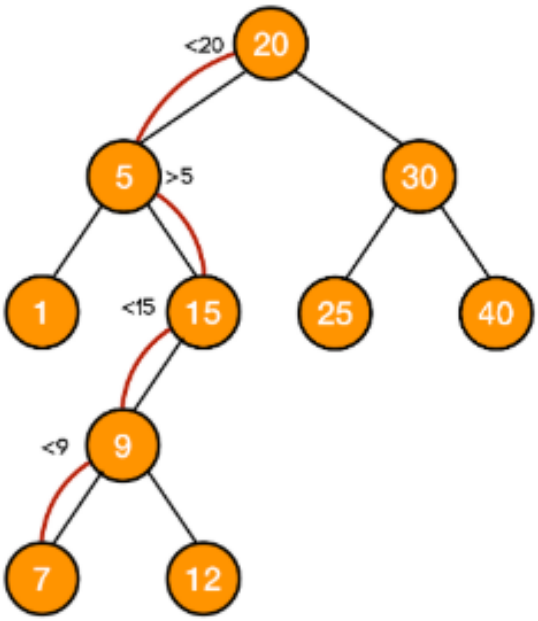

Figure 1: Searching in BST

For example in Figure 1 one has to search for key 7.We start at the root node as current node. Since the search key's value (7) is smaller than current node's key(20)

and the current node has *a left child*, we will search for the value in the left subtree left. Next, we search to the right since the search key's value(7) is greater than current node' key(5).This procedure will continue recursively in the subtree until the search key's value matches the current node's key.

In general, for each node x it encounters, it compares the key k with $key[x]$. If the two keys are equal, the search terminates. If k is smaller than $key[x]$, the search continues in the left subtree of x, since the binary-search-tree property implies that k could not be stored in the right subtree. Symmetrically, if k is larger than $key[x]$, the search continues in the right subtree.The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of this search operation is $O(h)$, where h is the height of the tree.

0.2.2 Insert algorithm

We can't insert any new node anywhere in a binary search tree because the tree after the insertion of the new node must follow the binary search tree property.Therefore,to insert an element, we first search for that element and if the element is not found, then we insert it.

Let us take an example similar to the previous search operation, but instead of finding the key 7 we will insert it into the tree in Figure 2 .We will use a temporary pointer and go to the place where the node is going to be inserted.Here, we are starting from the root of the tree and then moving to the $left$ subtree if the data of the node to be inserted is less than the current node. Otherwise, we are moving $right$.

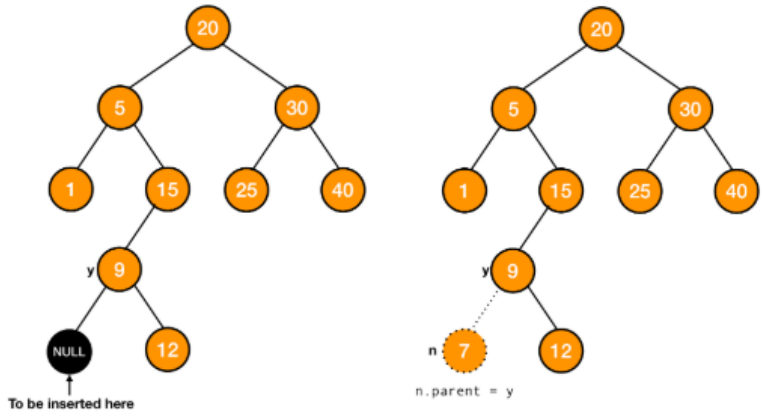

Figure 2: Inserting in BST

Hence the general way to explain the insert algorithm is:

(i) Always insert new node as leaf node

(ii) Start at root node as current node

(iii) If new node's key $<$ current's key

- If current node has a $left$ child, search $left$
- Else add new node as current's $left$ child

(iv) If new node's key $>$ current's key

- If current node has a *right* child, search *right*
- Else add new node as current's *right* child

(v) This process continues, until the new node is compared with a leaf node

Remark. *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.*

0.2.3 Delete algorithm

The last operation we need to do on a binary search tree to make it a full-fledged working data structure is to delete a node.

We can proceed in a similar manner to delete any node that has one child (or no children), but what can we do to delete a node that has two children? We are left with two links, but have a place in the parent node for only one of them. An answer to this dilemma, first proposed by *T. Hibbard* in 1962, is to delete a node x by replacing it with its *successor*.

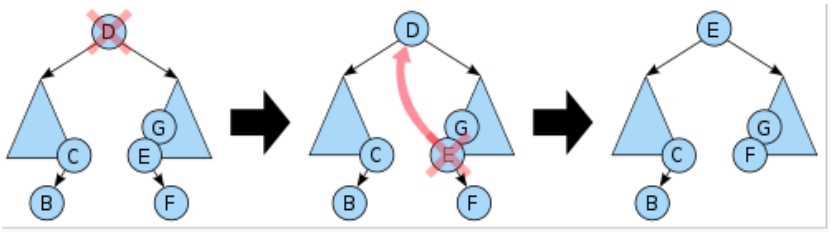

Figure 3: Deleting a node with 2 children in BST

There are 3 possible cases:

(i) Deleting a node with no children: simply remove the node from the tree.

(ii) Deleting a node with one child: remove the node and replace it with its child

(iii) Deleting a node with two children: call the node to be deleted D. Do not delete D. Instead, choose either its in-order predecessor node or its in-order successor node as replacement node E as Figure 3 . Copy the user values of E to D.If E does not have a child simply remove E from its previous parent G. If E has a child, say F, it is a right child. Replace E with F at E's parent.

Remark. *In all cases, when D happens to be the root, make the replacement node root again.*

Here is another example to better understanding 3 cases, see Figure 4:

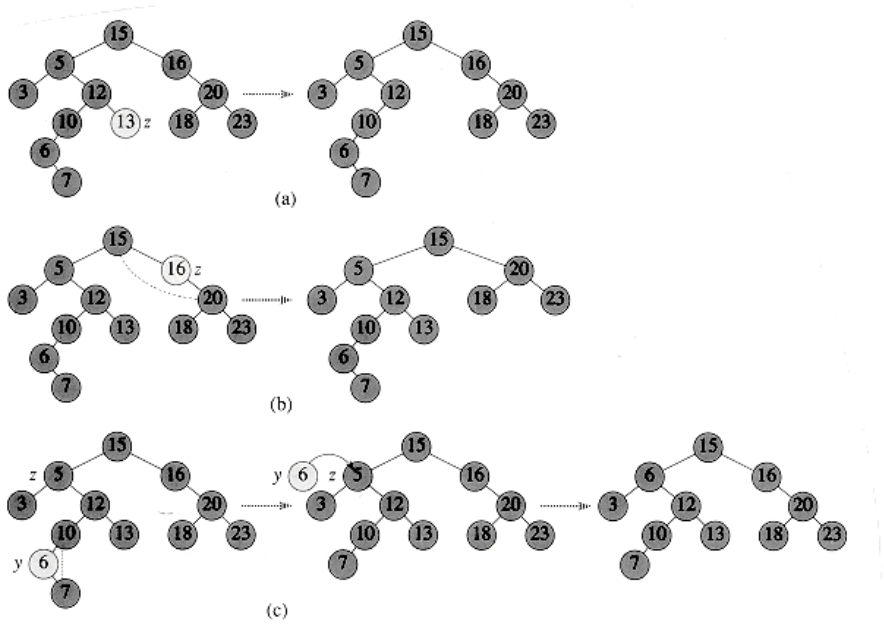

Figure 4: Deleting a node z from a binary search tree

In case (a) we can see that if z has no children, we just remove it. For case (b) since z has only one child, we splice out z. And in (c) z has two children, we splice out its successor , which has at most one child, and then replace the contents of z with the contents of the successor.

While this method does the job, it has a flaw that might cause performance problems in some practical situations. The problem is that the choice of using the successor is arbitrary and not symmetric.

Bibliography

[1] ttps://www.codesdope.com/course/data-structures-binary-search-trees/

[2] ttp://staff.ustc.edu.cn/ csli/graduate/algorithms/book6/chap13.htm

[3] ttps://en.wikipedia.org/wiki/Binary_search_tree

[4] ttps://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BST.html