ADVANCED DATA STRUCTURES - FINAL PROJECT

Ali Arabyarmohammadi
June 2023

Universitat Politècnica de Catalunya - UPC

Binary Search Trees

Abstract. This work investigates the essential properties, operations, and applications of binary search trees (BSTs). This fundamental data structure provides efficient access to elements, offering operations like insertion, deletion, and search in logarithmic time complexity. The content is primarily based on the study of Chapter 12 of the fourth edition of the book "Introduction to Algorithms"[1]. A series of exercises from the book are solved and discussed in detail, providing a practical understanding of BSTs. The problems cover various aspects such as insertion, deletion, and Querying binary search trees. The project aims to enhance my understanding of the concept, providing a robust foundation for further exploration of more complex tree structures.

Keywords: Binary Search Trees, Data Structures, Algorithms, Insertion, Deletion, Search, Querying a binary search tree

1 INTRODUCTION

The Binary Search Tree (BST) is a pivotal data structure that provides support for a range of dynamic set operations including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. This functionality allows a BST to be utilized effectively as both a dictionary and a priority queue.

Fundamental operations on a BST are dependent on its height. In an ideal scenario, when the BST is a complete binary tree with n nodes, the operations run in $O(\log n)$ worst-case time. However, if the tree degenerates into a linear chain, the same operations escalate to $O(n)$ worst-case time.

The structure of a BST, as the name indicates, is a binary tree. Each node in this structure holds a key, optional satellite data[1], and pointers to its left and right child nodes. Some implementations also include a pointer to the parent node. If a node is missing a child or a parent, the corresponding attribute points to a NIL value.

[1]In the context of data structures, "satellite data" refers to additional data associated with the key in a node. This data does not participate in the operations of the data structure but is merely stored and retrieved along with the key.

2 What is a binary search tree?

$12.1-1$
For the set $\{1,4,5,10,16,17,21\}$ of keys, draw binary search trees of heights 2, 3, 4,5 , and 6 .

Answer:

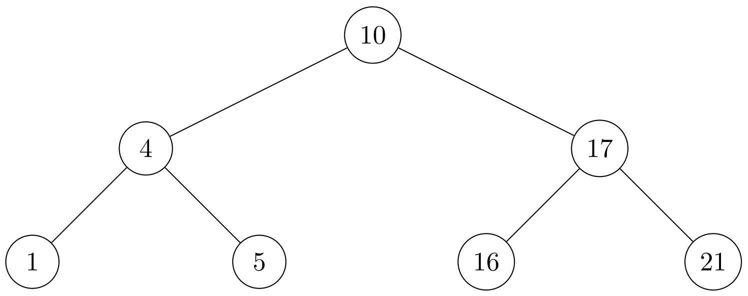
- The tree of height 2 has 10 as root, 4 and 17 as its children, and $\{1,5\}$ and $\{16,21\}$ as the grandchildren.

- The tree of height 3 has 10 as root, 4 and 17 as its children, 1 as the left child of 4,5 as the left child of 17 , and 21 as the right child of 17.

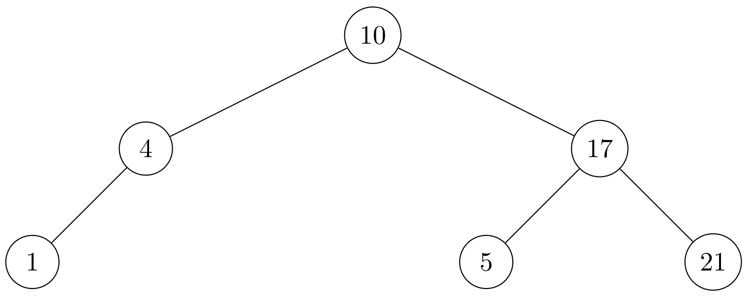
- The tree of height 4 has 10 as root, 4 and 21 as its children, 1 as the left child of 4,5 as the right child of 4 , and 17 as the left child of 21 , with 16 as the left child of 17.

- The tree of height 5 has 10 as root, 5 and 21 as its children, 4 as the left child of 5,1 as the left child of 4 , and 17 as the left child of 21 , with 16 as the left child of 17.

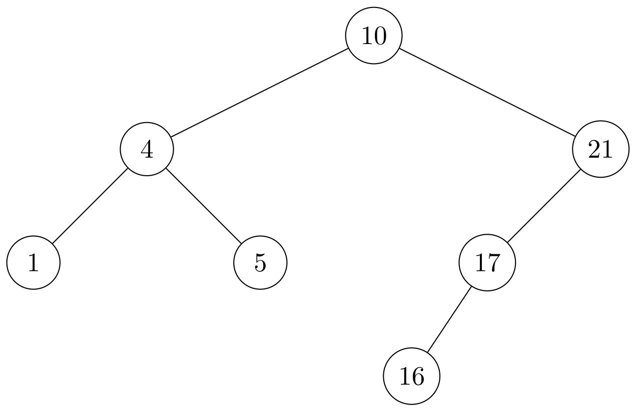
Height 2



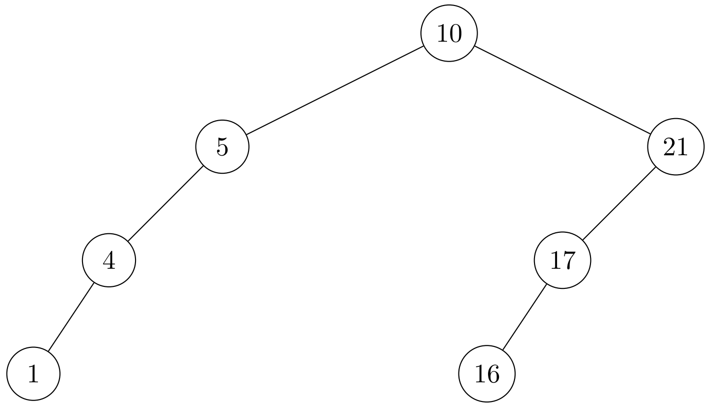
Height 3



Height 4



Height 5



$12.1 - 5$
What is the difference between the binary-search-tree property and the min-heap property on page 163 ? Can the min-heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time? Show how, or explain why not.

Answer:

The binary-search-tree property and the min-heap property are two different properties applied to binary trees.

1. **Binary Search Tree property**: For each node in a binary search tree, all keys in its left subtree are less than the node, and all keys in its right subtree are greater than the node. This property is recursive, applying to all subtrees within the tree.

2. **Min-Heap property**: In a min-heap, each node's key is less than or equal to the keys of its children. Unlike the binary search tree property, this rule applies to parent-child relationships and does not involve ordering between siblings or across different levels of the tree.

Because of the differences in these properties, a min-heap cannot be used to print the keys of an n-node tree in sorted order in $O(n)$ time, unlike a binary search tree. This is because a min-heap only ensures that a parent is smaller than its children but does not specify an order between siblings or across different levels of the tree. Therefore, to print the keys in sorted order from a min-heap, we would need to continually extract the minimum element, an operation that takes $O(\log n)$ time. Consequently, the overall time complexity becomes $O(n \log n)$, not $O(n)$.

12.1 – 3
Give a nonrecursive algorithm that performs an inorder tree walk. (Hint: An easy solution uses a stack as an auxiliary data structure. A more complicated, but elegant, solution uses no stack but assumes that you can test two pointers for equality.)

Answer:

An inorder tree walk can be performed non-recursively using a stack as an auxiliary data structure. The key idea of the algorithm is to use the stack to remember nodes that are yet to be visited. Here is the algorithm:

```
1. Initialize an empty stack, S.
2. Initialize a pointer, P, to the root of the tree.
3. While P is not null or S is not empty, repeat steps 4 and 5.
4. If P is not null, then
     a. Push P onto S.
     b. Move P to its left child.
5. Else
     a. Pop the top node, N, from S.
     b. Visit N.
     c. Move P to the right child of N.
```

The algorithm starts from the root of the tree and goes as far left as possible, pushing nodes onto the stack along the way. When it can go no further, it visits the node at the top of the stack and then goes to its right child, if it has one. If it does not have a right child, it pops nodes from the stack until it finds one that does or the stack becomes empty.

12.1 – 4
Give recursive algorithms that perform preorder and postorder tree walks in $\Theta(n)$ time on a tree of n nodes.

Answer:

Preorder Tree Walk: The preorder tree walk visits the root, then performs a preorder walk on the left subtree, followed by a preorder walk on the right subtree. Here is the recursive algorithm:

```
PREORDER-TREE-WALK(x)
1. if x != NIL
2.     print x.key
3.     PREORDER-TREE-WALK(x.left)
4.     PREORDER-TREE-WALK(x.right)
```

Postorder Tree Walk: The postorder tree walk performs a postorder walk on the left subtree, followed by a postorder walk on the right subtree, and then visits the root. Here is the recursive algorithm:

```
POSTORDER-TREE-WALK(x)
1. if x != NIL
2.     POSTORDER-TREE-WALK(x.left)
3.     POSTORDER-TREE-WALK(x.right)
4.     print x.key
```

Both these algorithms take $\Theta(n)$ time on a tree of n nodes because they visit each node exactly once.

$12.1-5$
Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

Answer:

In the comparison model, any algorithm that sorts n elements requires a lower bound of $\Omega(n \lg n)$ comparisons in the worst-case scenario. Constructing a binary search tree from an arbitrary list of n elements essentially requires sorting these elements because the inorder traversal of the constructed binary search tree yields the elements in sorted order.

This implies that if there were a comparison-based algorithm for constructing a binary search tree that could do so in less than $\Omega(n \lg n)$ time, it could also be used to sort n elements in less than $\Omega(n \lg n)$ time. This would violate the $\Omega(n \lg n)$ lower bound for comparison-based sorting algorithms. Therefore, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements must take $\Omega(n \lg n)$ time in the worst case.

---

3 Querying a binary search tree

12.2-1 You are searching for the number 363 in a binary search tree containing numbers between 1 and 1000. Which of the following sequences cannot be the sequence of nodes examined?

a. $2, 252, 401, 398, 330, 344, 397, 363$.

b. $924, 220, 911, 244, 898, 258, 362, 363$.

c. $925, 202, 911, 240, 912, 245, 363$.

d. $2, 399, 387, 219, 266, 382, 381, 278, 363$.

e. $935, 278, 347, 621, 299, 392, 358, 363$. **Answer:**

Answer:

In a BST, for every node, all the keys in its left subtree are less than the node and all the keys in its right subtree are greater than the node.

We use this property to determine which of the sequences given cannot be the sequence of nodes examined while searching for the number 363 in a BST.

(a) $2, 252, 401, 398, 330, 344, 397, 363$: This sequence is possible because each move to the right is to a node with a greater value, and each move to the left is to a node with a smaller value.

(b) $924, 220, 911, 244, 898, 258, 362, 363$: This sequence is possible because each move to the right is to a node with a greater value, and each move to the left is to a node with a smaller value.

(c) $925, 202, 911, 240, 912, 245, 363$: This sequence is **not** possible. After visiting node 911 and going left to node 240, it would be impossible to visit a node (912) greater than 911 in the left subtree.

(d) $2, 399, 387, 219, 266, 382, 381, 278, 363$: This sequence is possible because each move to the right is to a node with a greater value, and each move to the left is to a node with a smaller value.

(e) $935, 278, 347, 621, 299, 392, 358, 363$: This sequence is **not** possible. After visiting node 621 and going left to node 299, it would be impossible to visit a node (392) greater than 621 in the left subtree.

So, sequences (c) and (e) cannot be the sequence of nodes examined while searching for the number 363 in a BST.

$12.2 - 2$
Write recursive versions of TREE-Minimum and Tree-MaXimum.

Answer:

In a BST, the minimum and maximum values can be found by following left and right children from the root, respectively, until a leaf is encountered.

The recursive versions of TREE-Minimum and TREE-Maximum are as follows:

TREE-Minimum Recursive Function

```
FUNCTION TREE-Minimum(x):
    IF x.left != NIL THEN
        RETURN TREE-Minimum(x.left)
    RETURN x
```

This function starts at a node x and recursively goes to the left child until there is no more left child (i.e., $x.left$ equals NIL). At that point, the node x contains the minimum key.

TREE-Maximum Recursive Function

```
FUNCTION TREE-Maximum(x):
    IF x.right != NIL THEN
        RETURN TREE-Maximum(x.right)
    RETURN x
```

This function starts at a node x and recursively goes to the right child until there is no more right child (i.e., $x.right$ equals NIL). At that point, the node x contains the maximum key.

$12.2 - 3$
Write the TREE-PREDECESSOR procedure.

Answer:

The predecessor of a node x in a binary search tree is the node with the largest key less than $x.key$. The procedure to find the predecessor of a node in a binary search tree varies based on whether or not the node has a left child.

If the node has a left child, the predecessor is the maximum key in the left subtree. If the node does not have a left child, the predecessor is the lowest ancestor of x whose right child is also an ancestor of x.

The TREE-PREDECESSOR procedure can be defined as follows:

```
FUNCTION TREE-PREDECESSOR(x):
    IF x.left != NIL THEN
        RETURN TREE-MAXIMUM(x.left)
    y = x.p
    WHILE y != NIL AND x == y.left
        x = y
        y = y.p
    RETURN y
```

The function begins by checking if the node x has a left child. If it does, the function calls the TREE-MAXIMUM procedure on $x.left$ to find the maximum value in the left subtree.

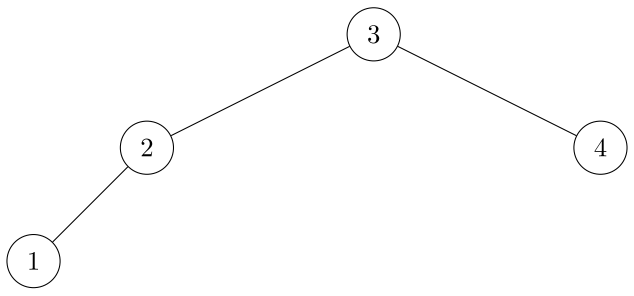
If x does not have a left child, the function finds the predecessor by going up the tree from x until it encounters a node y such that x is not the left child of y. This node y is the predecessor of x.

$12.2-4$
Professor Kilmer claims to have discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up at 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 Kilmer 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:

Professor Kilmer's claim is incorrect when considering keys to the left and right of the search path. His claim only holds when considering keys in the subtrees of a node encountered during the search.

A counterexample to the professor's claim can be provided by considering a binary search tree with just three nodes. We search for a key not in the tree, ending the search at a leaf node. Here's an example binary search tree:



In this example, we search for the key 5. Our search path includes the keys 3 and 4, so $B = \{3, 4\}$. The keys to the left of the search path are $A = \{2, 1\}$ and there are no keys to the right of the search path, hence $C = \{\}$.

In this case, if we choose $a = 2 \in A$, $b = 3 \in B$, and $c = 4 \in B$, we have $a \leq b \leq c$. But this contradicts the professor's claim that $a \leq b \leq c$ for any $a \in A$, $b \in B$, and $c \in C$ since we have $c \in B$, not C.

Therefore, Professor Kilmer's claim is not always true.

12.2 – 5
Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

Answer:

Suppose we have a node x in a binary search tree and node x has two children.

1. Successor of a node in BST is the node with the smallest key greater than x. The successor of a node can be found by following two steps starting from this node:

- If the right subtree of node x is nonempty, then the successor of x is just the node with the minimum key in the right subtree of x. - If the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

From the first case, it is clear that the successor of x (if it exists) does not have a left child because it is the minimum element in the right subtree of x.

2. Predecessor of a node in BST is the node with the largest key less than x. The predecessor of a node can be found by following two steps starting from this node:

- If the left subtree of node x is nonempty, then the predecessor of x is just the node with the maximum key in the left subtree of x. - If the left subtree of node x is empty and x has a predecessor y, then y is the lowest ancestor of x whose right child is also an ancestor of x.

From the first case, it is clear that the predecessor of x (if it exists) does not have a right child because it is the maximum element in the left subtree of x.

Therefore, we conclude that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

12.2 – 6
Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

Answer:

If a node x in a binary search tree has an empty right subtree and has a successor y, then y must be a node that has a key greater than x but is the smallest among all such nodes.

Since x does not have a right child, y cannot be in the right subtree of x. Also, since y has a key greater than x, it cannot be in the left subtree of x. Therefore, y must be an ancestor of x.

Furthermore, if y were the right child of its parent, then the parent would be another node with a key greater than x but smaller than y. This contradicts the assumption that y is the smallest key greater than x.

Therefore, y must be the left child of its parent. And since y is the smallest key greater than x, there cannot be any other ancestor of x that is a left child of its parent and has a key smaller than y. Hence, y is the lowest ancestor of x whose left child is also an ancestor of x.

12.2 – 7
An alternative method of performing an in-order tree walk of an n-node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making $n - 1$ calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

Answer:

We know that the TREE-MINIMUM operation on a binary search tree takes $O(h)$ time, where h is the height of the tree, because it traverses a path from the root to a leaf.

The TREE-SUCCESSOR operation, when called for a node x, could take $O(h)$ time in the worst-case scenario. This worst-case scenario happens when x is the rightmost node of the tree (i.e., the node with the maximum key), in which case the operation would need to traverse from x to the root.

However, when we're calling TREE-SUCCESSOR $n-1$ times in a sequence, each node is visited exactly once. This is because each call to TREE-SUCCESSOR returns the node in the tree that has the next higher key compared to the current node. Therefore, the total time for $n-1$ calls to TREE-SUCCESSOR is $O(n)$, since each call involves a constant amount of work (checking and updating a few pointers) plus an amount of work proportional to the number of edges traversed (and each edge in the tree is traversed at most twice – once upwards and once downwards).

Therefore, the overall time complexity for this alternative in-order tree walk algorithm is $O(h)+O(n) = O(n)$, where h is the height of the tree. We can say that this algorithm runs in $\Theta(n)$ time in all cases, not just for balanced binary search trees.

12.2 – 8
Prove that no matter what node you start at in a height- h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k + h)$ time.

Answer:

The k successive calls to TREE-SUCCESSOR are essentially an in-order traversal of k elements of the tree. As analyzed in the previous exercise, the time complexity of in-order traversal is linear in the number of nodes visited, regardless of the shape of the tree. Therefore, k calls to TREE-SUCCESSOR will take $O(k)$ time.

However, before we can start the in-order traversal, we may need to descend from the starting node to the minimum element in the tree. In the worst case, the starting node is the root of the tree and the tree is a degenerate tree (essentially a linked list) with height h. Thus, we might need to traverse h edges to reach the minimum element.

Therefore, no matter where we start in a height-h binary search tree, k successive calls to TREE-SUCCESSOR will take $O(k+h)$ time in total: $O(k)$ for the in-order traversal of k elements and $O(h)$ for the initial descent to the minimum element.

$12.2-9$
Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $y.key$ is either the smallest key in T larger than $x.key$ or the largest key in T smaller than $x.$ key.

Answer:

Assume $y.key > x.key$, meaning x is a left child. We know that for any binary search tree, if a node has a right child, then the minimum key larger than the node's key is the smallest key in the right subtree. If the node does not have a right child, then the minimum key larger than the node's key is one of the node's ancestors: specifically, the lowest ancestor for which the node is in the left subtree. Since x is a leaf node, it has no right child. Therefore, the smallest key larger than $x.key$ must be one of x's ancestors. The parent y is the lowest such ancestor, so $y.key$ is the smallest key in T larger than $x.key$.

Similarly, assume $y.key < x.key$, meaning x is a right child. If a node has a left child, then the maximum key smaller than the node's key is the largest key in the left subtree. If the node does not have a left child, then the maximum key smaller than the node's key is one of the node's ancestors: specifically, the lowest ancestor for which the node is in the right subtree. Since x is a leaf node, it has no left child. Therefore, the largest key smaller than $x.key$ must be one of x's ancestors. The parent y is the lowest such ancestor, so $y.key$ is the largest key in T smaller than $x.key$.

4 Insertion and deletion

12.3-1
Give a recursive version of the TREE-INSERT procedure.

Answer:

Algorithm 1 Recursive TREE-INSERT procedure

```
 1: procedure RecursiveTreeInsert(node, key)
 2:     if node ← NIL then
 3:         return NewNode(key)
 4:     else
 5:         if key < node.key then
 6:             node.left ← RecursiveTreeInsert(node.left, key)
 7:         else
 8:             node.right ← RecursiveTreeInsert(node.right, key)
 9:         end if
10:     end if
11:     return node
12: end procedure
```

12.3 – 5
Suppose that you construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is 1 plus the number of nodes examined when the value was first inserted into the tree.

Answer:

When we first insert a value into a binary search tree, we traverse from the root of the tree to the location where we insert the node, examining each node along the path. This forms the search path for that value.

When we subsequently search for that same value, we follow the same path from the root to the node. Since the binary search tree property ensures that all values to the left of a node are smaller and all values to the right are larger, no other path could lead to that node.

However, the difference between the search and the insertion process is that when we search, we also examine the node containing the value itself, while during insertion, we stop at the parent of the new node.

Thus, when searching for a value, we examine 1 node more than when the value was first inserted. Hence, the number of nodes examined in searching for a value in the tree is 1 plus the number of nodes examined when the value was first inserted into the tree.

12.3 – 3
You can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worstcase and best-case running times for this sorting algorithm?

Answer:

The running time of this sorting algorithm depends on the time it takes to build the binary search tree (BST) and then perform an in-order tree walk.

To build the BST, we use TREE-INSERT n times. In the worst case, if the numbers are inserted in increasing or decreasing order, the BST becomes skewed and the time to insert each number is proportional to the number of numbers already in the tree. Thus, the worst-case time to build the BST is $1 + 2 + \cdots + n = \Theta(n^2)$.

In the best case, if the numbers are inserted in an order such that the BST is perfectly balanced, the height of the BST is $\log n$ and the time to insert each number is $O(\log n)$. Thus, the best-case time to build the BST is $O(n \log n)$.

An in-order tree walk takes $O(n)$ time, as it visits each node once.

Thus, the worst-case running time of this sorting algorithm is $\Theta(n^2)$, and the best-case running time is $O(n \log n) + O(n) = O(n \log n)$.

12.3 – 4
When TREE-DELETE calls TRANSPLANT, under what circumstances can the parameter v of TRANSPLANT be NIL?

Answer:

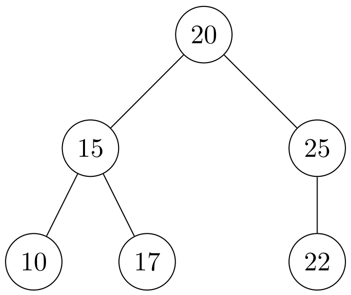
The parameter v of TRANSPLANT in TREE-DELETE can be NIL only when the node to be deleted z is a leaf node (i.e., it has no children). In this case, z is replaced with NIL.

12.3 – 5
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.

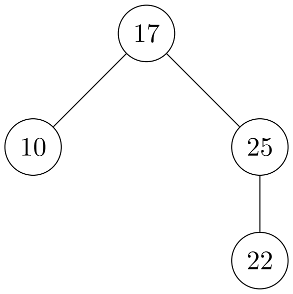
Answer:

No, the operation of deletion is not commutative in a binary search tree. The order of deletions can affect the final structure of the tree. Here is a counterexample to illustrate this:

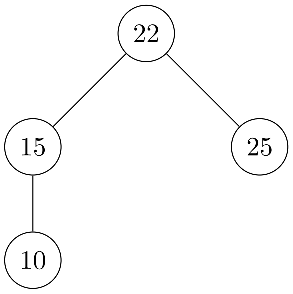
Suppose we have a binary search tree with the nodes 20, 15, 25, 10, 17, and 22, where 20 is the root. The tree would look like this:



If we delete 15 first and then 20, the tree would look like this:



But if we delete 20 first and then 15, the tree would look like this:



As you can see, the two resulting trees are different, so the operation of deletion is not commutative.

12.3 – 6
Suppose that instead of each node x keeping the attribute x.p, pointing to x 's parent, it keeps x.succ, pointing to x 's successor. Give pseudocode for TREESEARCH, TREE-INSERT, and Tree-Delete on a binary search tree T using this representation. These procedures should operate in $O(h)$ time, where h is the height of the tree T. You may assume that all keys in the binary search tree are distinct. (Hint: You might wish to implement a subroutine that returns the parent of a node.)

Answer:

Algorithm 2 TREE-INSERT(T, z)

```
procedure TreeInsert($T$, $z$)
    $y \leftarrow NIL$
    $x \leftarrow T.root$
    while $x \neq NIL$ do
        $y \leftarrow x$
        if $z.key < x.key$ then
            $x \leftarrow x.left$
        else
            $x \leftarrow x.right$
        end if
    end while
    if $y == NIL$ then
        $T.root \leftarrow z$
    else if $z.key < y.key$ then
        $y.left \leftarrow z$
        $z.succ \leftarrow y$
    else
        $z.succ \leftarrow y.succ$
        $y.right \leftarrow z$
        $y.succ \leftarrow z$
    end if
end procedure
```

Algorithm 3 TREE-DELETE(T, z)

```
1: procedure TreeDelete($T$, $z$)
2:     if $z.left == NIL$ then
3:         Transplant($T$, $z$, $z.right$)
4:     else if $z.right == NIL$ then
5:         Transplant($T$, $z$, $z.left$)
6:     else
7:         $y \leftarrow z.succ$
8:         if $y \neq z.right$ then
9:             Transplant($T$, $y$, $y.right$)
10:            $y.right \leftarrow z.right$
11:            $y.right.succ.left \leftarrow y$
12:        end if
13:        Transplant($T$, $z$, $y$)
14:        $y.left \leftarrow z.left$
15:    end if
16:    if $z.succ \neq NIL$ and $z.succ.left == z$ then
17:        $z.succ.left \leftarrow y$
18:    end if
19: end procedure
```

Algorithm 4 TREE-SEARCH(T, k)

```
1: procedure TreeSearch($T$, $k$)
2:     $x \leftarrow T.root$
3:     $prev \leftarrow NIL$
4:     while $x \neq NIL$ and $x.key \neq k$ do
5:         if $x.key > k$ then
6:             $x \leftarrow x.left$
7:         else
8:             $prev \leftarrow x$
9:             $x \leftarrow x.right$
10:        end if
11:    end while
12:    if $x \neq NIL$ then
13:        return $x$
14:    else
15:        return $prev$
16:    end if
17: end procedure
```

12.3 – 7
When node z in TREE-DELETE has two children, you can choose node y to be its predecessor rather than its successor. What other changes to TREE-DELETE are necessary if you do so? Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might TREE-DELETE be minimally changed to implement such a fair strategy?

Answer:

Algorithm 5 TREE-DELETE(T, z)

```
procedure TreeDelete($T$, $z$)
    Generate a random number $r \in \{0, 1\}$
    if $r == 0$ then                                ▷ The original case, use successor
        if $z.left == NIL$ then
            Transplant($T$, $z$, $z.right$)
        else if $z.right == NIL$ then
            Transplant($T$, $z$, $z.left$)
        else
            $y \leftarrow$ minimum of right subtree of $z$
            if $y.parent \neq z$ then
                Transplant($T$, $y$, $y.right$)
                $y.right \leftarrow z.right$
                $y.right.parent \leftarrow y$
            end if
            Transplant($T$, $z$, $y$)
            $y.left \leftarrow z.left$
            $y.left.parent \leftarrow y$
        end if
    else                                            ▷ New case, use predecessor
        if $z.right == NIL$ then
            Transplant($T$, $z$, $z.left$)
        else if $z.left == NIL$ then
            Transplant($T$, $z$, $z.right$)
        else
            $y \leftarrow$ maximum of left subtree of $z$
            if $y.parent \neq z$ then
                Transplant($T$, $y$, $y.left$)
                $y.left \leftarrow z.left$
                $y.left.parent \leftarrow y$
            end if
            Transplant($T$, $z$, $y$)
            $y.right \leftarrow z.right$
            $y.right.parent \leftarrow y$
        end if
    end if
end procedure
```

References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*, 4th Edition, The MIT Press, Cambridge, Massachusetts, 2022, ISBN: 9780262046305.