1 BST Height

**Question:** What is the best case BST height? Worst case? If shuffling, probabilistically, leads to a log n tree height, why don’t we simply shuffle our input data before building our BST based symbol table to avoid worst case behavior?

**Answer:** The best case BST height occurs when the BST is perfectly balanced or balanced (difference of height between subtrees is at most 1). The height in this case is $\log(n)$, where $n$ is the number of nodes in the BST. If the tree was not balanced, then the height will be greater than $\log(n)$ in all cases, since the height of the tree becomes the height of the longest branch.

The worst case height is $n$. Observe when the BST structure leads to a single branch, our height is equal to the number of nodes. Trivially, we cannot do any worse than that.

![Diagram](a) Worst case height (b) Best case height (balanced)

While probabilistically we can obtain a height of $\log(n)$ when constructing our BST-based symbol table, the client using the symbol table can add elements to the BST, altering the structure. We can start from a best case height and shift to a worst case height depending on the elements dynamically added by the client. Therefore, we need a mechanism of maintaining a height of $\log(n)$ (Red-Black trees are a solution).

2 Deleting Nodes from a BST

**Question:** Describe two methods for deleting nodes from a BST. What effect do they have on its running time?

**Answer:**

2.1 Marking Deleted Nodes

Given an input node (value) we want to delete, we can perform search on the BST. Once we find the node, we mark it as deleted (we can have wrappers that add an additional field deleted, or we can replace them with null nodes, which can be harder to manage). This will cost us more space and can potentially increase the runtime of all our operations since the number of nodes will never decrease.
2.2 Standard Search and Delete

Given an input node (value) we want to delete, we perform search on the BST. If we find the node:

- **Case 1 - Node is a leaf**: Remove the node.
- **Case 2 - Node has only one child (one subtree)**: Remove the node and attach the subtree to the parent of the removed node.
- **Case 3 - Node has two children (left subtree and right subtree)**: We will call the node to delete \( x \). To delete \( x \), we will proceed to replace it by its successor \( y \) (the node that is immediately greater than it in the sorted order). Because \( x \) node has a right subtree, then the successor \( y \) must be the minimum node in the right subtree. To understand how to find the successor \( y \) in the right subtree, consider the following algorithm:
  
  - Starting at \( x \), we go down once and to the right. If we find a left branch, we take that left branch and we keep going to the left until we can’t anymore. We have found \( y \). Else, there is no left branch at all, then the successor must be the node immediately to the right of \( x \).

  Once we found the successor, we call delete on it recursively. Note that deleting the successor will never fall in this third case (meaning it can never have two children!). We prove this by contradiction: Suppose the successor of \( x \), which we call \( y \), has two children. Then \( y \) has a left child. Therefore the successor of \( x \) is actually the left child of \( y \), a contradiction.

  After the successor \( y \) is deleted, we replace \( x \) by \( y \).

This method of deleting a node in a BST is proportional to the height of the tree. This will never increase the runtime of our operations since we actually get rid of the node, so the number of nodes will go down by at most 1 (0 when the node is not found). The runtime of our operations are unaffected (and can improve in some cases).

3 Traversals

**Question:** Draw the binary tree for this level-order traversal: \( P \ E \ S \ A \ N \ V \ L \ Y \ I \). Give the in-order traversal, preorder traversal, and postorder traversal of the tree that you drew. For each of these traversals, could you reconstruct the tree and if so, explain how.

**Answer:** Given the list, we can run a level-order traversal on it to generate the following tree:

```
          P
           
        E      S
       /         
      A      N    V
     /     /     
    Y     I    L
```

Now we can run a pre-order (Root, L, R), post-order (L, R, Root), and in-order traversals (L, Root, R) on this tree to obtain the following lists:

- pre-order: \( P, E, A, Y, \) null, null, \( I, \) null, null, \( N, \) null, null, \( S, \) \( V, \) null, null, \( L, \) null, null
- post-order: null, null, \( Y, \) null, null, \( I, A, \) null, null, \( N, E, \) null, null, \( V, \) null, null, \( L, S, P \)
- in-order: null, \( Y, \) null, \( A, \) null, \( I, \) null, \( E, \) null, \( N, \) null, \( P, \) null, \( V, \) null, \( S, \) null, \( L, \) null

Tree reconstruction:
- pre-order:
Algorithm 1 Binary Tree Reconstruction from Pre-Order Traversal

1: **procedure** solution(list)
2:    root ← list.removeFirst
3:    if (root == null) then **return** null
4:    leftSubtree ← solution(list)
5:    rightSubtree ← solution(list)
6:    root.left ← leftSubtree
7:    root.right ← rightSubtree
8: **return** root

- post-order: Reconstructing a binary tree (not necessarily a BST) cannot be done with a post-order traversal only. The list contains the root of the tree at the tail, but the other nodes do not follow a specific order, so there is no obvious way of doing a recursion on the list. If it was a BST, we can find the element that is immediately less than the root and divide the list accordingly for proper recursion.

- in-order: In this type of traversal, we cannot locate the root without further information (i.e. a post-order or a pre-order). We cannot reconstruct a binary tree from an in-order traversal only.

4 BST Property

**Question:** Prove that if a node in a BST has two children, its successor has no left child and its predecessor has no right child.

**Answer:** Consider a node $x$ that has two children. The successor of $x$ must lie in the right subtree of $x$, and must be the minimum element in that right subtree. If the successor $y$ has a left child, then it is not the minimum in the right subtree, a contradiction.

The predecessor case is symmetric (swap minimum for maximum, left with right).

5 Building a BST using a Compare Algorithm

**Question:** Prove that no compare-based algorithm can build a BST using fewer than $\log(N!)$ $N\log N$ compares.

**Answer:** If it was possible to build a BST using a compare-based algorithm that does fewer than $N\log(N)$ compares, then we essentially sort an array in less than $N\log(N)$ compares, because a BST maintains a total ordering over its elements. The universal lower bound for sorting an array is $N\log(N)$. This means we cannot do better than this, as it is mathematically proven. Therefore it is impossible to build such an algorithm.