Homework 6

Tushar Jain, N12753339
CSCI-GA 1170-003 - Fundamental Algorithms

October 13, 2017

Problem 1.a. Give an algorithm that constructs a balanced binary search tree from an array of n elements in time $O(n \log n)$. Briefly justify running time and correctness.

Solution.

We first sort the array in $O(n \log n)$ time (for instance, using MERGE-SORT) and then get the middle of the array, make it the root and do this recursively on the left and right side of the middle while making them left and right children.

Assumption: We assume that the elements are distinct because if we don't, cases having multiple values equal to that of the median would cause problems.

Even if we chose the element having the same value as that of the median with the highest index (as only left subtree can have equal values), there's a chance of building an unbalanced BST (i.e. | left-height - right-height | > 1). This can be clearly be seen with a counter-example such as an array having the exact same value for all its elements (e.g. $[2, 2, 2, 2, 2, 2,]$). I don't see any way of building a balanced BST with only the left sub-tree having equal to or less than elements.

By changing the node structure (adding color for red-black tree or balance-factor for AVL tree), we can handle such situation to a certain extent but we still obviously can't handle extreme cases. However, we were informed we must do this question without changing the node structure on the course forum. Thus, assuming that the elements in the provided array are distinct, the following solution is put forward.

```python
class BSTNode: # Definition for a binary search tree node
  def __init__(self, x):
    self.key = x
    self.left = None
    self.right = None
    self.parent = None

def sorted_array_to_BST(A, parent):
  if not A:
    return None
  mid = len(A) // 2 # Python3 division
  root = BSTNode(A[mid])
  root.left = sorted_array_to_BST(A[:mid], root)
  root.right = sorted_array_to_BST(A[mid+1:], root)
  root.parent = parent
  return root

def array_to_BST(A):
  A_sorted = merge_sort(A) # sorted(A) for python
  return sorted_array_to_BST(A_sorted, None)
```

Correctness:

Assumption: We assume that MERGE-SORT returns the sorted order of array A in $O(n \log n)$ running time.

Proof by Induction:

Base Case: For $n = 1$, there is only 1 element in array A and length of A is 1. Thus, $mid = 0$ and the root is BSTNode with $key = A[0]$. As the subarrays passed to recursive calls are None, the left and right attributes of the root will point to None and this root node will be returned as it is to be expected.

Induction Step: Assuming $n \leq k$, the BST is correctly built. Then for $n = k + 1$, the algorithm chooses the middle element from the sorted array , i.e. the median of the array, and then recursively build the right and left sub-tree using 2 halves on either side of the median respectively, which we know will be built correctly as for them $n \leq k$. Since we choose the median as the root and approximately equal number of elements in its right and left subtree while taking into account if they were smaller or greater than the median, we correctly build a balanced BST of of size $k + 1$. Thus, the above stated algorithm is correct.

Running Time:
The time complexity of MERGE-SORT used in line 19 is $O(n \log n)$. Now looking at the recursive algorithm `sorted-array-to-BST` being invoked at line 20. Following is the recurrence relation for this algorithm:

$$\begin{aligned} T(n) &= 2T(n/2) + c \\ &= O(n) && \text{(Using Master Theorem)} \end{aligned}$$

Therefore, the total time complexity of the above algorithm is $O(n \log n) + O(n) = O(n \log n)$.

□

Problem 1.b. Design an algorithm that takes as input an INORDER-TREE-WALK and POSTORDERTREE-WALK of a binary tree T on n nodes (both as n-elements arrays) and outputs the PREORDERTREE-WALK of T (again, as n-element array). Notice, T is not necessarily a binary search tree. Briefly justify correctness and running time of your algorithm.

Note: you may assume the tree has distinct elements.

Solution.

Assumption: Inorder walk has elements in sorted order.

```
def preorder_from(inorder, postorder):
  # base case
  if len(inorder) == len(postorder) == 0:
    return []
  if len(inorder) == len(postorder) == 1:
    return postorder

  root_idx = binary_search(inorder, postorder[-1])

  left = preorder_from(inorder[:root_idx], postorder[:root_idx])
  right = preorder_from(inorder[root_idx+1:], postorder[root_idx:-1])

  return [postorder[-1]] + left + right
```

Correctness:

Assumption: We assume that BINARY-SEARCH returns the index of the target element in sorted array A in $O(\log n)$ running time.

Proof by Induction:

Base Case: For $n = 1$, there is only 1 element in the arrays and it is returned as expected.

Induction Step: Assuming $n \leq k$, the algorithm is correct. Then for $n = k + 1$, the algorithm chooses the last element from postorder array, finds its corresponding index in the inorder array , and then recursively do these steps on the right and left sub-array after correctly identifying them for which the algorithm works correctly as we know their size $n \leq k$. Then we return the recursively found left and right pre-order sub-arrays concatenated with the root element found before in the right order $(root|left|right)$. Thus, the above stated algorithm is correct.

Running Time:

The above algorithm visits every node in the array. For every visit, it calls binary-search which takes $O(n \log n)$ time. Therefore, overall time complexity of the algorithm is $O(n \log n)$ □

Problem 1.c. Now assume that the tree T is a binary search tree. Modify your algorithm in part (b) so that it works given only the POSTORDER-TREE-WALK of T.

Solution. To convert from *postorder* to *preorder* walks of a BST, we look into the structure of each carefully.

- *postorder* : $left|right|root$
- *preorder* : $root|left|right$

Thus, we simply need to find find the boundary of left and right subtrees in the postorder walk by comparing to the root value which we know as it's the last value in the postorder walk. We do this recursively and stitch together the arrays to form the preorder walk for the same BST.

```python
def preorder_from(postorder):
  # base case
  if len(postorder) == 0:
    return []
  if len(postorder) == 1:
    return postorder

  root = postorder[-1]

  i = len(postorder) - 2
  while root < postorder[i]:
    i = i - 1

  left = preorder_from(postorder[:i+1])
  right = preorder_from(postorder[i+1:-1])

  return [root] + left + right
```

Correctness: Proof by Induction in a very similar manner to the previous question.

Running Time:

The above algorithm visits every node in the array. For every visit, it loops till it finds an element greater than the root element which takes at most $O(n)$ time. Therefore, overall time complexity of the algorithm is $O(n^2)$

□

Problem 2. Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. For each of the following sequences, say whether or not it could be the sequence of nodes examined, and justify.

Solution:

Using the below algorithm we should inspect each sequence. The target, k, will be 363. Simply if 2 consecutive values are both greater than than the target and are in monotonically increasing order, the sequence is not from a BST. Similarly, if 2 consecutive values are both less than than the target and are in decreasing order, the sequence is not from a BST.

This can be clearly seen from the algorithm below because if the node examined has a value greater than the target, the algorithm goes to the left child of the node, which by BST property should equal to to less than the parent node. Also, the other case of the target value being greater than the examined node can be considered similarly where the algorithm goes to the right child of the examined node which must be greater than its parent by BST property.

Also, whenever there is an increase in the values of nodes examined, all future nodes must be greater than the origin (node before the increase) of increase ,because we are then examining the right subtree of the origin. And similarly, whenever there's a decrease all future elements must be smaller or equal to the origin, because we are then examining the left subtree of the origin.

Pseudo-Code:

```
def BST_search(x, k):
  if x == None or k == x.key:
    return x
  if k < x.key:
    return BST_search(x.left, k)
  else:
    return BST_search(x.right, k)
```

- 2,252,401,398,330,344,397,363.
  Yes, it could be from a BST
- 924, 220, 911, 244, 898, 258, 362, 363.
  Yes, it could be from a BST
- 925, 202, 911, 240, 912, 245, 363.
  No, it could **not** be from a BST as 912 can not be a part of 911's left subtree as $912 > 911$.
- 2,399,387,219,266,382,381,278,363.
  Yes, it could be from a BST
- 935, 278, 347, 621, 299, 392, 358, 363.
  No, it could **not** be from a BST as 299 can not be a part of 347's right subtree as $299 < 347$.

□

Problem 3. Professor Donald 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 Donald claims that any three keys $a \in A, b \in B$,and $c \in C$ must satisfy $a \leq b \leq c$. Give the smallest possible counterexample to the professors claim.

Solution:

For the smallest possible counter-example, we need at least 4 nodes. Suppose we search for the number 9 in the following tree:

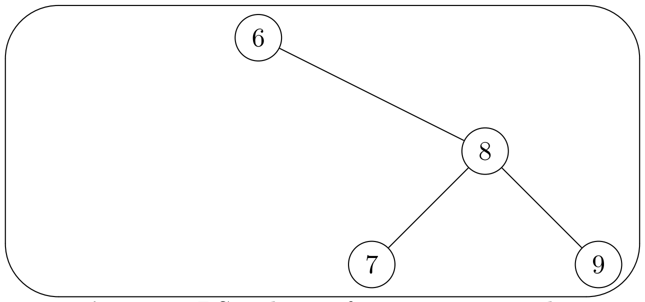

Fig 6.1: BST chosen for counterexample.

Then A = 7, B = 6, 8, 9 and C = ϕ, and Professor Donalds claim fails as 6 ¡ 7 because $6 \in B$ while $7 \in A$.

□

Problem 4. We 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 worst-case and best-case running times for this sorting algorithm?

Solution:

- *Worst Case:* $O(n^2)$

  The worst case will occur when the constructed BST is completely skewed which would happen when the given array is in sorted order. And into some time for an element into a BST is of the order of O(h), where h is the height of the BST. In the case of sorted array, the insertion time for an element would be in order, $O(i)$ of the position ,i, of the element in the array as that would be the height of the intermediate BST. In total the order of formation of the BST would be:

  $$O(1) + O(2) + ... + O(n) = O(n^2)$$

  And the time taken for printing them in an in-order tree walk would take $O(n)$. Thus, in the worst case, the running time for the total algorithm would be $O(n^2) + O(n) = O(n^2)$.

- *Best Case:* $O(n \log n)$

  The best case would be when the constructed BST is at most balanced. One such possibility is when the sorted array is a pre-order tree walk. Having a balanced BST implies that the height of the formed

BST would be of the order of $O(\log_2 n)$ as the maximum number of elements a BST can hold is $2^{h+1} - 1 = O(2^h)$. Thus, the insertion time for an element would be in order, $O(\log i)$ of the logarithm of its position ,i, of the element in the array as that would be the height of the intermediate BST. In total the order of formation of the BST would be:

$$O(\log 1) + O(\log 2) + ... + O(\log n) = O(n \log n)$$

And the time taken for printing them in an in-order tree walk would take $O(n)$. Thus, in the worst case, the running time for the total algorithm would be $O(n \log n) + O(n) = O(n \log n)$.

□

Problem 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.

Solution:

No, deletion is not "commutative" in the above stated sense. Consider the following tree for purpose of demonstrating a counterexample.

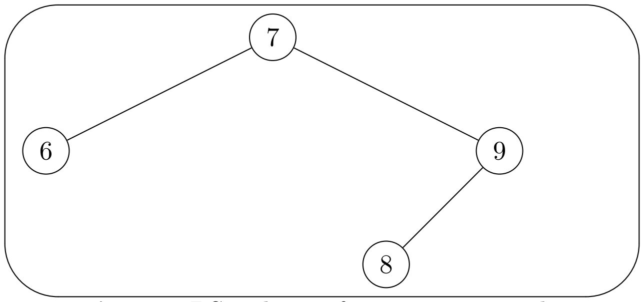

Fig 6.2: BST chosen for counterexample.

- *Case 1:* Deleting "6" and then "7".

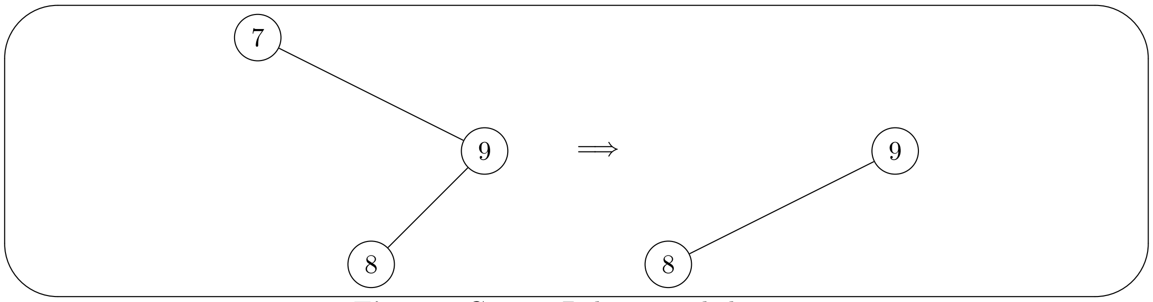

Fig 6.3: Case 1: Delete 6 and then 7.

- *Case 2:* Deleting "7" and then "6".

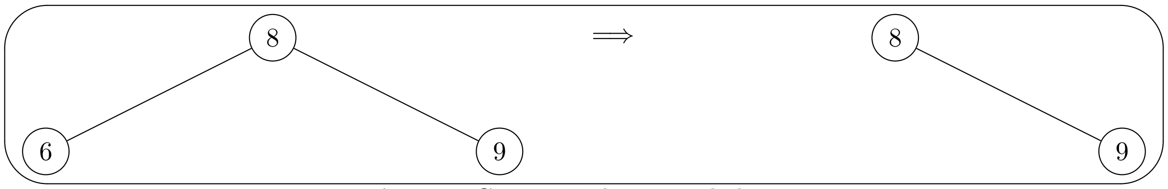

Fig 6.4: Case 2: Delete 7 and then 6.

And we can clearly see the 2 BSTs we arrived at in the 2 cases are different.

□