Tirgul 9

Binary Search Trees

sample questions

Q. Write a pseudo code for computing the height of a binary search tree (BST), what is its time complexity?

A. The height of a binary tree is the maximum between the height of its children + 1. We can write a simple recursive algorithm for calculating the height:

```
int height(node) {
  if (node == null)
        return -1;
  return (1 + max(height(node.left),
                  height(node.right)));
}
```

The algorithm visits all of the tree nodes

➲ its time complexity is $O(n)$

Q. You are provided with a nearly BST, a binary tree in which for every node, either the key of its left child is smaller than its key and the key of its right child is bigger, or vice versa. Write a pseudo code for fixing the tree, what is its complexity?

A. The solution is very simple, we should check each node and swap its children if necessary, pay attention that fixing (or not fixing) a node doesn't say we shouldn't fix its children.

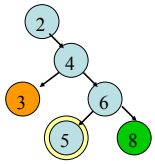
```
void fix(node) {
  // should check for nulls
  if (node.left.key > node.right.key)
        swap(node.left,node.right)
  fix(node.left);
  fix(node.right);
}
```

The algorithm visits all of the tree nodes

➲ its time complexity is $O(n)$

Q. Your colleague tells you he has found a remarkable BST property: 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. He claims that any three keys $a \subset A, b \subset B$ and $c \subset C$ must satisfy *a* < *b* < *c*. Is the claim true?

A. No:



Q. Write an algorithm for building a sorted circular doubly linked list containing the elements of a BST, what is its complexity?

A. We should iterate through all the tree nodes, starting with the minimum and moving to the successor and add them to the list (remembering to link the last node to the first one)

`--- no pseudo code ---`

Traversing through all the nodes takes $O(n)$ (previous tirgul).
Adding n elements to the list takes $O(n)$ as well

➲ the total time complexity is therefore $O(n)$

<u>One drawback:</u>

We need an extra $O(n)$ space complexity (for the list)
can we avoid the extra space?

Q. Write an algorithm for the same purpose without using the extra space (at the same time complexity)

A. We exploit the fact that the keys of all the members to the left of a node are smaller than that of the node while all those to its right are larger.
Intuitively, we would like to convert the left subtree into a list, convert the right subtree into a list and concatenate the lists [LEFT, node, RIGHT].
The algorithm is recursive and does exactly so
It uses the tree nodes pointers (NO EXTRA SPACE!). In the result list, left means previous while right means next
The algorithm performs a constant amount of work for each node and its time complexity is therefore linear

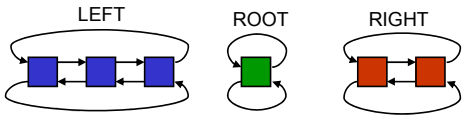
```
List TreeToList(root) {
  if (root == null)
       return null;
  List LEFT  = TreeToList(root.left);
  List RIGHT = TreeToList(root.right);
  List ROOT = makeList(root);
  return concatenate(LEFT, ROOT, RIGHT);
}
```

`makeList` - takes a single tree node and builds a circular doubly linked list (containing one element) of it.

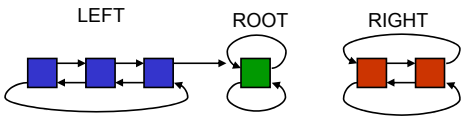
`concatenate` - takes three circular doubly linked lists and concatenates them into one list (assume it can handle null lists)

```
List concatenate(LEFT, ROOT, RIGHT) {
  // should handle null lists
  LEFT.left.right = ROOT;
  ROOT.left = LEFT.left;
  ROOT.right = RIGHT;
  RIGHT.left.right = LEFT;
  LEFT.left = RIGHT.left;
  RIGHT.left = ROOT;
  return LEFT;
}
```

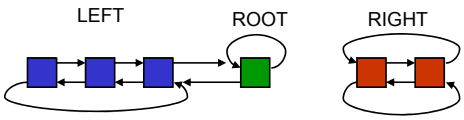

Concatenate example



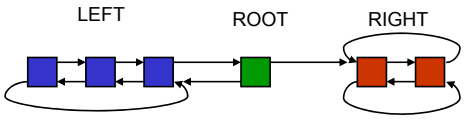
`LEFT.left.right = ROOT;`



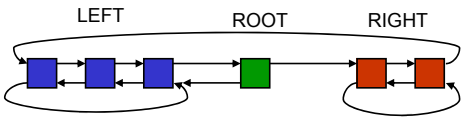
```
ROOT.left = LEFT.left;
```



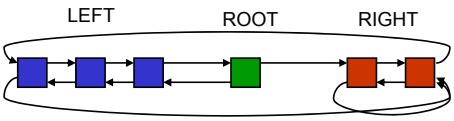

```
ROOT.right = RIGHT;
```




```
RIGHT.left.right = LEFT;
```




```
LEFT.left = RIGHT.left;
```




```
RIGHT.left = ROOT;
```