CS29003 Algorithms Laboratory

Assignment No: 6

Date: 03–March–2020

Binary Trees

In this assignment, you construct a binary tree from user input. Assume that the nodes of the tree store distinct keys. Your task is to convert the input tree *in place* to a binary search tree with respect to these key values. The only operations permitted are child swaps and rotations (explained in Part 3). You are allowed to read keys stored in the nodes, but you are *not* allowed to swap keys in two different nodes. You are also *not* allowed to copy external data (keys) to the nodes. Your conversion process must not create any new node. You need to achieve your objective by making a sequence of permitted operations (child swaps and rotations) on the input binary tree. The final BST you produce should be balanced. Your algorithm should run in $O(n^2)$ time, where n is the number of nodes in the tree.

Part 1: The tree data structure

Write a user-defined data type to store a node in binary trees. Each node should store an integer key, and two child pointers (left and right). If you choose, you may include a parent pointer in each node. Besides these, the nodes must not store any additional data.

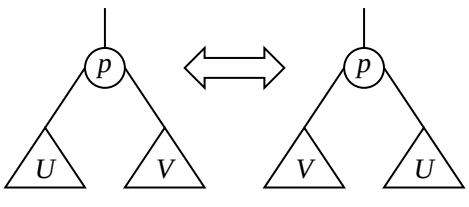
Part 2: Construct the tree from user input

Write a function *readtree* to construct a binary tree from user input. The user first specifies the number n of nodes in the tree. The user then specifies the keys in a preorder fashion. In addition to the key for a node, the user also specifies the numbers of nodes in the left and the right subtrees of that node. The sample output illustrates the user input. Assume that it is the user's duty to supply n distinct key values, and subtree counts consistent with a binary tree with n nodes. Your function may return a pointer to the root of the tree. For a reason that will be clear soon, the function may additionally create a dummy root, make the actual root the right child of the dummy root, and return a pointer to the dummy root. The key of the dummy root will never be consulted, so this field may be kept uninitialized, or set to $-\infty$.

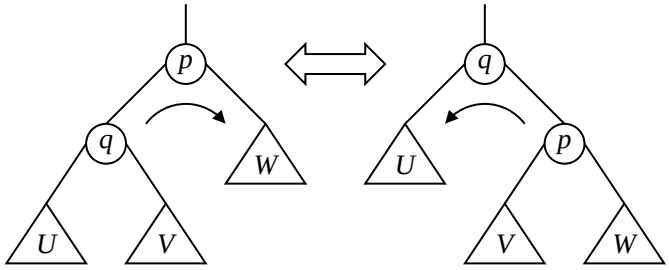
Part 3: Utility functions

Write a function to print the preorder listing of the keys in a binary tree, and another function to print the inorder listing of the keys in a binary tree. You can uniquely construct a tree from these two listings. You may write another function *prntree* that calls these two listing functions.

In later parts, we need two operations on the nodes of a binary tree: (i) swapping the two child pointers of a node, and (ii) left/right rotation at a node. These operations are explained in the figure below. Write three functions *swapchild*, *lrotate* and *rrotate*. Each rotate function should return a pointer to the new root of the rotated subtree. The parent of the old root should change its appropriate child pointer to point to the new root. This is the reason why it is helpful for the root to have a dummy parent.



Swap child nodes



Rotation

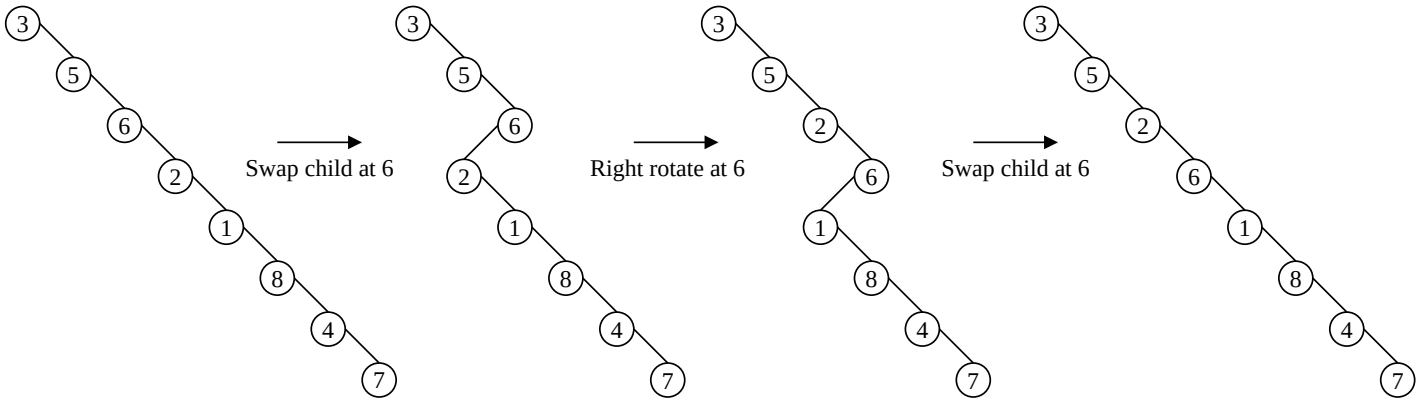
Note that the rotate operations do not change the inorder listing of the tree, but a child swap operation does (unless made on empty subtrees).

Part 4: Make the tree completely right-skew

By making a sequence of right rotations, make the tree completely right skew. Start at the root, and keep on applying right rotations there until the left subtree becomes empty. Then proceed to the right child, and repeat. This process takes $O(n)$ time. Write a function *makeskew* for this part.

Part 5: Bubble sort the skew tree

After Part 4, your tree is essentially a linked list connected by the right-child links. For a general binary tree, this list is not sorted in general. Write a function *bsort* to bubble sort this list. Bubble sort requires swapping of data in consecutive locations. However, changing the key values of nodes is not permitted. A child swap, a right rotation, and finally another child swap is a sequence that achieves swapping of two nodes in the linked list. The following figure illustrates this process. The running time is $O(n^2)$.



Part 6: Rebalance the skew tree

Finally, we want a balanced tree in the sense that at each node v, we must have $|\text{left}(v)| - |\text{right}(v)| \in \{0,1\}$, where $|u|$ is the number of nodes in the subtree rooted at u. Write a function *rebalance* that works as follows. After Part 5, the tree is fully right-skew. Locate the middle node in the list. Keep on applying left rotations at the root until that middle element reaches the root position. At this point, the left subtree of the root is fully left-skew, and the right subtree of the root is fully right-skew. Again, locate the middle nodes in the two subtrees. Keep on applying rotations (right rotations at the left child of the root, and left rotations at the right child of the root) until the middle nodes are placed in the desired locations. Recurse. This process takes $O(n \log n)$ time.

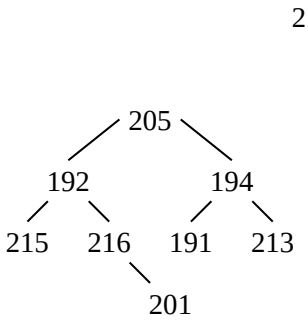
The *main*() function

- Call *readtree* to construct the input binary tree T. Print T.
- Call *makeskew* on T. Print T.
- Call *bsort* on the right-skew tree T. Print T.
- Call *rebalance* on the sorted right-skew tree. Print T.

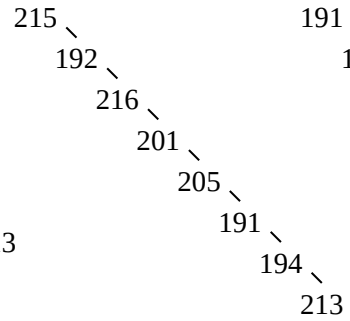
Submit a single C/C++ source file. Do not use global/static variables.

Sample output

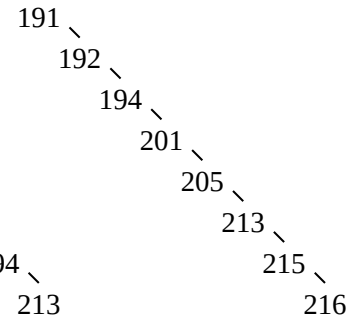
The sample output corresponds to the trees given below.



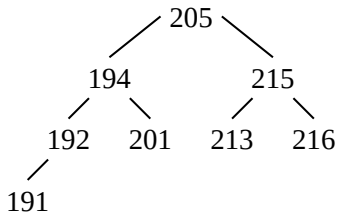
The initial binary tree



After right skewing



After bubble sort



After rebalancing

```
n = 8

205     4   3
192     1   2
215     0   0
216     0   1
201     0   0
194     1   1
191     0   0
213     0   0

+++ Initial tree
--- Preorder listing
    205   192   215   216   201   194   191   213
--- Inorder listing
    215   192   216   201   205   191   194   213

+++ Tree made skew
--- Preorder listing
    215   192   216   201   205   191   194   213
--- Inorder listing
    215   192   216   201   205   191   194   213

+++ Tree after sorting
--- Preorder listing
    191   192   194   201   205   213   215   216
--- Inorder listing
    191   192   194   201   205   213   215   216

+++ Tree after rebalancing
--- Preorder listing
    205   194   192   191   201   215   213   216
--- Inorder listing
    191   192   194   201   205   213   215   216
```