## CS29003 ALGORITHMS LABORATORY

### BALANCED BINARY SEARCH TREE without Extra Information per Node

### Last Date of Submission: 9 – August – 2018

**General Instruction**

1. Please do not use any global variable unless you are explicitly instructed so.
2. Please use proper indentation in your code.
3. Please name your file as `<roll_no>_<assignment_no>`. For example, if your roll number is 14CS10001 and you are submitting assignment 3, then name your file as `14CS10001_3.c or 14CS10001_3.cpp` as applicable.
4. Please write your name and roll number at the beginning of your program.

---

In the class, we have studied **BALANCED BINARY SEARCH TREE** which ensure that the height of the tree is $\mathcal{O}(\log n)$ whenever the tree has $n$ nodes. Popular examples of such **BALANCED BINARY SEARCH TREE** are AVL tree, red black tree, etc. However, such trees usually store extra information in every node of the tree – difference between the height of left and right sub-trees in case of AVL tree and color of a node in case of red black tree. Today, we will implement a "**SUPERB BINARY SEARCH TREE**" which does not store any extra information and still guarantees that any sequence of $k$ tree operations can be performed in $\mathcal{O}(k \log n)$ time. The basic operation of SBST is **LIFT**.

**LIFTING:**

**LIFTING** a node $\mathcal{N}$ is some particular sequence of *rotations* which ensures that $\mathcal{N}$ is placed at the root of the new **SUPERB BINARY SEARCH TREE**. In every step, we perform one of the three sub-steps – `make-root`, `same-orientation`, `opposite-orientation`. Let $p(\mathcal{N})$ and $g(\mathcal{N})$ denote the parent and grand parent of $\mathcal{N}$.

**Make-root step:**

This sub-step is performed only when $p(\mathcal{N})$ is the root node. Figure 1 shows schematic diagram of `make-root` step where $\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}_3$ are arbitrary subtrees (possibly empty).

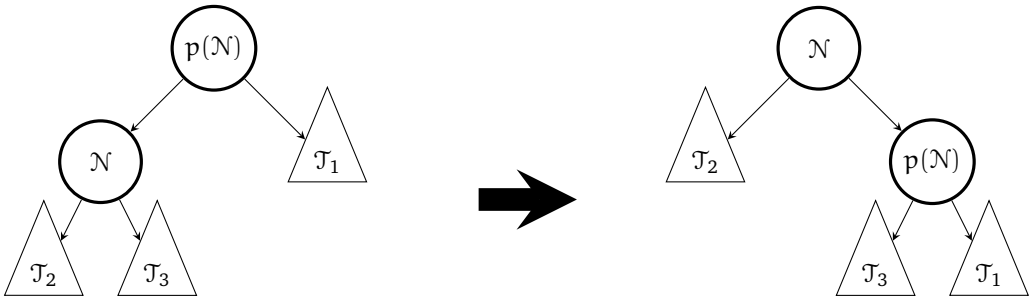


Figure 1: Make-root step.

**Same-orientation step:**

This sub-step is performed only when $\mathrm{p}(\mathcal{N})$ is not the root node and $\mathcal{N}$ and $\mathrm{p}(\mathcal{N})$ are either both left child or both right child of their respective parents. Figure 2 shows schematic diagram of `same-orientation` step.

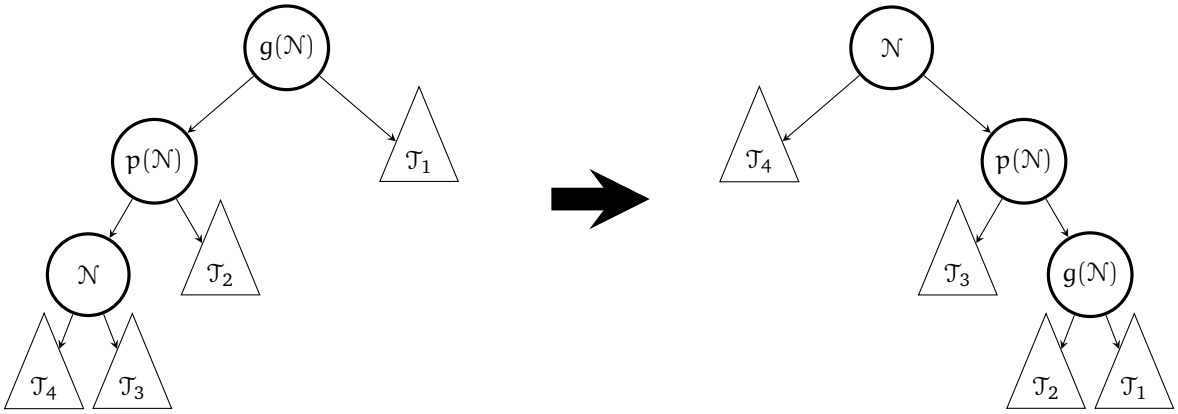


Figure 2: Same-orientation step.

**Opposite-orientation step:**

This sub-step is performed only when $\mathrm{p}(\mathcal{N})$ is not the root node and $\mathcal{N}$ is a left child and $\mathrm{p}(\mathcal{N})$ is a right child of their respective parents or vice versa. Figure 3 shows schematic diagram of `opposite-orientation` step.

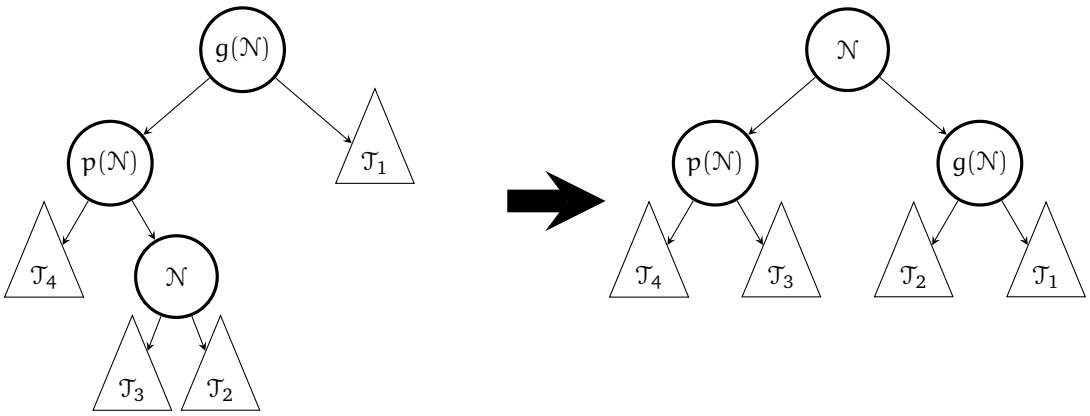


Figure 3: Opposite-orientation step.

Please use the following struct to define a node of a binary tree.

```
typedef struct node{
    int value;
    struct node *left;
    struct node *right;
    struct node *parent;
}NODE, *NODEPTR;
```

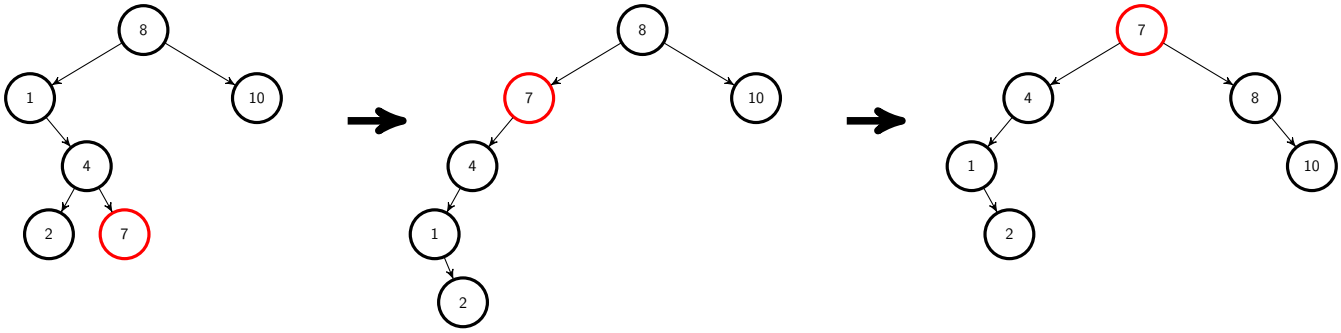


Figure 4: Sequence of rotations after inserting 7.

## Part I: Implement Make-root, Same-orientation, and Opposite-orientation

Implement Make-root, Same-orientation, and Opposite-orientation using the following prototype.

**NODEPTR makeRoot(NODEPTR root, NODEPTR N)**

**NODEPTR sameOrientation(NODEPTR root, NODEPTR N)**

**NODEPTR oppositeOrientation(NODEPTR root, NODEPTR N)**

All the three functions above return the pointer to the root of the modified tree.

## Part II: Insert into SUPERB BINARY SEARCH TREE

To insert a new key $x$, we first insert $x$ into **SUPERB BINARY SEARCH TREE** as we do in usual **BINARY SEARCH TREE**. Then we perform **LIFT** on the node $\mathcal{X}$ corresponding to $x$ to make $\mathcal{X}$ the root node of the **SUPERB BINARY SEARCH TREE**. Please use the following prototype to insert a key into **SUPERB BINARY SEARCH TREE** which returns a pointer to the root node of the new tree. Insert the key in your **SUPERB BINARY SEARCH TREE** only if it is not already present in your **SUPERB BINARY SEARCH TREE**. If the key is already present, then do nothing. Figure 4 depicts an example the sequence of rotations one needs to perform after an insertion.

**NODEPTR insert(NODEPTR root, int key)**

## Part III: Delete from SUPERB BINARY SEARCH TREE

Deleting a key $x$ is also exactly same as deletion in usual **BINARY SEARCH TREE** except we perform **LIFT** on the parent of the deleted node. Please use the following prototype to insert a key into **SUPERB BINARY SEARCH TREE**. Delete the key in the **SUPERB BINARY SEARCH TREE** only if it is already present in the **SUPERB BINARY SEARCH TREE**. If the key is not already present in the **SUPERB BINARY SEARCH TREE**, then do nothing.

**NODEPTR delete(NODEPTR root, int key)**

## main()

In main() make the following declarations.

**NODEPTR root=NULL;**

This creates an empty **SUPERB BINARY SEARCH TREE**. The pointer variable root is meant to contain the address of the root node of the **SUPERB BINARY SEARCH TREE**, and is initialized to NULL.

Next, take as input the number of numbers $n$ to insert. Then, read $n$ numbers one by one and call insert $n$ times to build a **SUPERB BINARY SEARCH TREE**. After each call, assign the return values back to root.

Next print the preorder and inorder traversals of your **SUPERB BINARY SEARCH TREE**. Please use the following prototype for implementing preorder and postorder traversals.

**void preOrder(NODEPTR root), void inOrder(NODEPTR root)**

Next, take as input the number $n$ of numbers to delete. Then, read $n$ numbers one by one and call delete $n$ times. Print the preorder and inorder traversals of your **SUPERB BINARY SEARCH TREE** after each deletion.

## Sample Output

Enter number of numbers to insert: 6

Enter numbers to insert: 8, 1, 4, 2, 7, 10

preorder traversal: 8, 1, 4, 2, 7, 10
preorder traversal: 1, 2, 4, 7, 8, 10

Enter number of numbers to delete: 2

Enter key to delete: 4

Preorder traversal: 1, 8, 2, 7, 10
Inorder traversal: 1, 2, 7, 8, 10

Enter key to delete: 2

Preorder traversal: 8, 1, 7, 10
Inorder traversal: 1, 7, 8, 10