

AVL Trees

Data and File Structures Laboratory

<http://www.isical.ac.in/~dfs/lab/2019/index.html>

Type definitions

```
typedef int DATA;
```

```
typedef struct node {  
    DATA data;  
    int left, right, parent, height; // following Weiss,  
    DS & AA in C++, 4ed.  
} AVL_NODE;
```

```
typedef struct {  
    unsigned int num_nodes, max_nodes;  
    int root, free_list;  
    AVL_NODE *nodelist;  
} TREE;
```

API functions

```
extern int init_tree(TREE *);  
extern int search(TREE *, int , DATA);  
extern int insert(TREE *, int , int *, DATA);  
extern int delete(TREE *, int , int *, DATA);  
  
#define DELETE_TREE(tree) free(tree->nodelist);
```

parent index (optional)



Helper functions

```
extern int grow_tree(TREE *);
extern int get_new_node(TREE *);
extern void free_up_node(TREE *, int);
extern int find_successor(TREE *, int);
extern void rotate_on_insert_LL(TREE *, int , int *);
extern void rotate_on_insert_RR(TREE *, int , int *);
extern void rotate_on_insert_LR(TREE *, int , int *);
extern void rotate_on_insert_RL(TREE *, int , int *);
extern void balance(TREE *, int, int *);

extern void inorder(TREE *, int);
extern void print_tree(TREE *, int, int);
extern void print_pstree(TREE *, int);

#define HEIGHT(T, nodeindex) ( ((nodeindex) == -1) ? -1 :
    T->nodelist[nodeindex].height )
```

find_successor() |

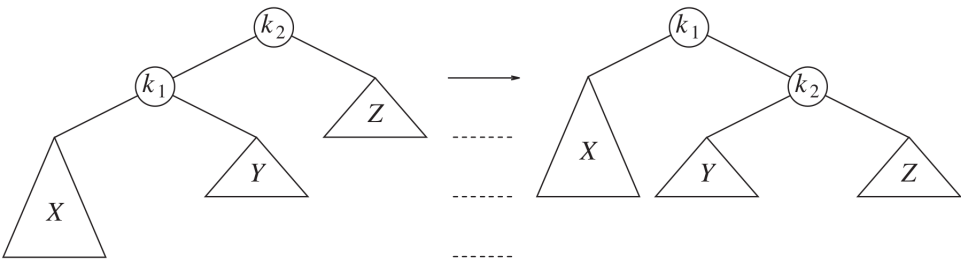
```
int find_successor(TREE *tree, int node) {
    int child;
    assert(node != -1);
    /* Go to right child, then as far left as possible */
    child = tree->nodelist[node].right;
    if (child == -1) /* no successors */
        return -1;
    if (tree->nodelist[child].left == -1) {
        /* Don't do this here for AVL trees */
        /* tree->nodelist[node].right = tree->nodelist[child].right; */
        /* if (tree->nodelist[child].right != -1) */
        /*     tree->nodelist[tree->nodelist[child].right].parent =
node; */
        return child;
    }
    while (tree->nodelist[child].left != -1) {
        node = child;
        child = tree->nodelist[child].left;
    }
}
```

find_successor() ||

```
}  
/* Don't do this here for AVL trees */  
/* tree->nodelist[node].left = tree->nodelist[child].right; */  
/* if (tree->nodelist[child].right != -1) */  
/*     tree->nodelist[tree->nodelist[child].right].parent = node; */  
return child;  
}
```

Rotate LL

- Insertion into **Left** subtree of **Left** subtree causes imbalance
- Rotate right to restore balance



Rotate LL - code I

```
void rotate_on_insert_LL(TREE *tree, int parent, int *node) {
    /* See Weiss, DS & AA in C++, 4 ed., Section 4.4.1, Figure 4.34 */
#ifdef DEBUG
    printf("LL (right) rotation at %d\n",
        tree->nodelist[*node].data);
#endif // DEBUG
    int k2 = *node;
    int k1 = tree->nodelist[k2].left;
    int Z = tree->nodelist[k2].right;
    int X = tree->nodelist[k1].left;
    int Y = tree->nodelist[k1].right;

    /* rotate */
    tree->nodelist[k2].left = Y;
    tree->nodelist[k1].right = k2;

    /* parents (optional) */
    tree->nodelist[k1].parent = parent;
}
```


Rotate LL - code II

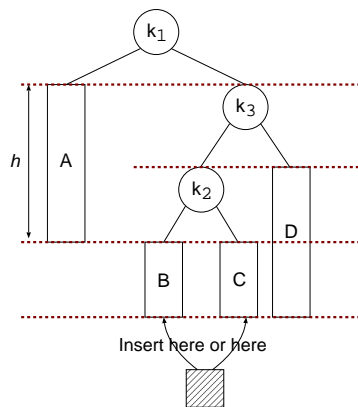
```
tree->nodelist[k2].parent = k1;
if (Y != -1) tree->nodelist[Y].parent = k2;

/* update heights */
tree->nodelist[k2].height = 1 + MAX(HEIGHT(tree, Y), HEIGHT(tree,
Z));
tree->nodelist[k1].height = 1 + MAX(HEIGHT(tree, X), HEIGHT(tree,
k2));

*node = k1;
return;
}
```

Rotate RL

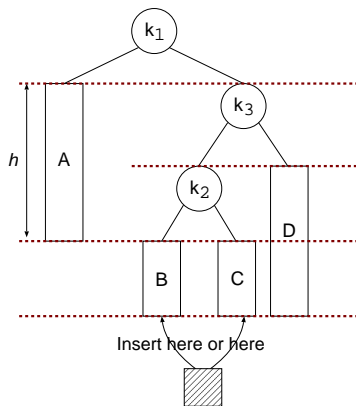
Insertion into **L**eft subtree of **R**ight subtree causes imbalance



h - height

bf - balance factor

Insertion into **L**eft subtree of **R**ight subtree causes imbalance



$$bf_{new}(k_1) = -2, \quad bf_{old}(k_1) = -1$$

$$\Rightarrow h_{new}(k_3) = h + 2, \quad h_{old}(k_3) = h + 1$$

$$\Rightarrow \max(h_{new}(k_2), h(D)) = h + 1$$

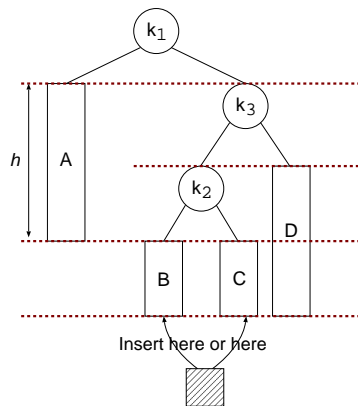
$$\max(h_{old}(k_2), h(D)) = h$$

$$\Rightarrow h_{new}(k_2) = h + 1, \quad h_{old}(k_2) = h$$

h - height

bf - balance factor

Insertion into **L**eft subtree of **R**ight subtree causes imbalance



h - height

bf - balance factor

$$bf_{new}(k_1) = -2, \quad bf_{old}(k_1) = -1$$

$$\Rightarrow h_{new}(k_3) = h + 2, \quad h_{old}(k_3) = h + 1$$

$$\Rightarrow \max(h_{new}(k_2), h(D)) = h + 1$$

$$\max(h_{old}(k_2), h(D)) = h$$

$$\Rightarrow h_{new}(k_2) = h + 1, \quad h_{old}(k_2) = h$$

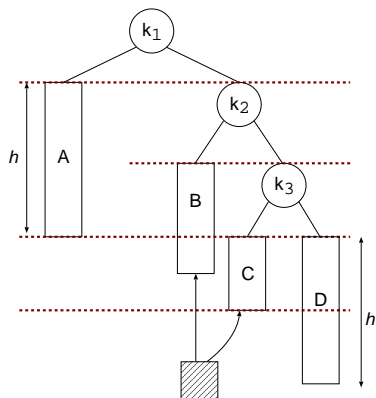
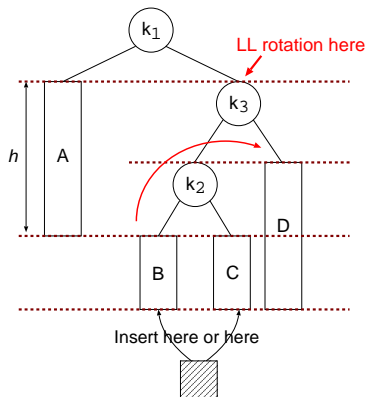
$$h(D) \in \{h, h - 1\}$$

If $h(D) = h - 1$, imbalance would be observed

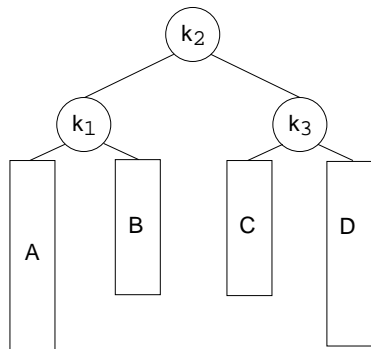
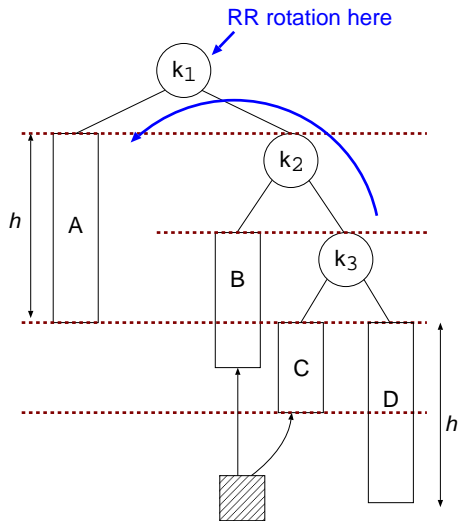
at k_3 before k_1

$$\Rightarrow h(D) = h$$

Rotate RL - I



Rotate RL - II



Rotate RL - code

```
void rotate_on_insert_RL(TREE *tree, int parent, int *node) {
    /* See CMSC 420 Lecture Notes by David M. Mount, UMCP, pg. 39. */
#ifdef DEBUG
    printf("RL (double) rotation at %d\n",
        tree->nodelist[*node].data);
#endif // DEBUG
    int k1 = *node;
    rotate_on_insert_LL(tree, k1, &(tree->nodelist[k1].right));
    rotate_on_insert_RR(tree, parent, node);
    return;
}
```

balance() I

```
void balance(TREE *tree, int parent, int *node) {
    int thisnode = *node;
    int left = tree->nodelist[thisnode].left;
    int right = tree->nodelist[thisnode].right;

    if (HEIGHT(tree, left) - HEIGHT(tree, right) > 1) {
#ifdef DEBUG
        printf("Left sub-tree too high at %d\n",
            tree->nodelist[thisnode].data);
#endif // DEBUG

        if (HEIGHT(tree, tree->nodelist[left].left) >= HEIGHT(tree,
            tree->nodelist[left].right))
            rotate_on_insert_LL(tree, parent, node);
        else
            rotate_on_insert_LR(tree, parent, node);
    }
    else if (HEIGHT(tree, right) - HEIGHT(tree, left) > 1) {
```


balance() ||

```
#ifdef DEBUG
    printf("Right sub-tree too high at %d\n",
        tree->nodelist[thisnode].data);
#endif // DEBUG
    if (HEIGHT(tree, tree->nodelist[right].right) >= HEIGHT(tree,
        tree->nodelist[right].left))
        rotate_on_insert_RR(tree, parent, node);
    else
        rotate_on_insert_RL(tree, parent, node);
}

thisnode = *node;
left = tree->nodelist[thisnode].left;
right = tree->nodelist[thisnode].right;
tree->nodelist[thisnode].height = 1 +
    MAX(HEIGHT(tree, left), HEIGHT(tree, right));
return;
}
```

delete() I

```
int delete(TREE *tree, int parent, int *root, DATA d) {
    int thisnode = *root;
    if (thisnode == -1)
        return 0;
    if (d < tree->nodelist[thisnode].data) {
#ifdef DEBUG
        printf("Deleting recursively from left subtree ");
        PRINT_NODE(tree, tree->nodelist[thisnode].left);
#endif
        if (FAILURE == delete(tree, thisnode,
            &(tree->nodelist[thisnode].left), d))
            return FAILURE;
    }
    else if (d > tree->nodelist[thisnode].data) {
#ifdef DEBUG
        printf("Deleting recursively from right subtree ");
        PRINT_NODE(tree, tree->nodelist[thisnode].right);
#endif
    }
}
```

delete() II

```
    if (FAILURE == delete(tree, thisnode,
&(tree->nodelist[thisnode].right), d))
        return FAILURE;
}
else {
    /* DELETE THIS NODE */
    if (tree->nodelist[thisnode].left != -1 &&
        tree->nodelist[thisnode].right != -1) {
        int successor = find_successor(tree, thisnode);
        assert(successor != -1);
        tree->nodelist[thisnode].data =
tree->nodelist[successor].data;
#ifdef DEBUG
        printf("Replacing "); PRINT_NODE(tree, thisnode);
        printf(" by successor "); PRINT_NODE(tree, thisnode);
#endif
        if (FAILURE == delete(tree, thisnode,
&(tree->nodelist[thisnode].right),
                        tree->nodelist[successor].data))
```

delete() III

```
        return FAILURE;
    }
    else {
        /* EITHER LEAF or ONLY ONE CHILD */
#ifdef DEBUG
        printf("Deleting "); PRINT_NODE(tree, thisnode);
#endif

        if (tree->nodelist[thisnode].left != -1) {
            *root = tree->nodelist[thisnode].left;
            tree->nodelist[*root].parent = parent;
#ifdef DEBUG
            printf(" replacing by "); PRINT_NODE(tree, *root);
#endif
        }
        else if (tree->nodelist[thisnode].right != -1) {
            *root = tree->nodelist[thisnode].right;
            tree->nodelist[*root].parent = parent;
#ifdef DEBUG
            printf(" replacing by "); PRINT_NODE(tree, *root);
#endif
        }
    }
}
```

delete() IV

```
#endif
    }
    else {
#ifdef DEBUG
        printf(" (leaf)\n");
#endif

        *root = -1;
    }
    free_up_node(tree, thisnode);
    tree->num_nodes--;
    if (parent != -1) {
        int left = tree->nodelist[parent].left;
        int right = tree->nodelist[parent].right;
        tree->nodelist[parent].height = 1 +
            MAX(HEIGHT(tree, left), HEIGHT(tree, right));
    }
}
}
```

delete() V

```
    balance(tree, parent, root);  
    return 0;  
}
```

Problems I

1. Complete the missing parts (marked TODO) in the provided implementation of AVL trees.
2. Given a tree that is supposed to be an AVL tree, write a function to check whether it is indeed an AVL tree, and whether all fields have correct / consistent values.

You will need to check whether the `left`, `right`, and `parent` fields are consistent, whether the `height` field is correct (if not, fill in the field with the correct value), and finally whether imbalances (if any) are within the permissible limit.

For this problem, the tree will be given to you via an input file, using a format similar to the one used in Lab Test 2. The name of the input file will be given as a command-line argument.

3. Use the function in problem 2 to test (and debug if necessary) the AVL tree implementation in program 1.

4. Consider an array A that contains elements from an ordered set. Two elements $A[i]$ and $A[j]$ of the array are said to form an *inversion* if $A[i] > A[j]$ and $i < j$. Write a program to count the number of inversions in a given array.

Note that the inversion count indicates how far (or close) the array is from being sorted in ascending order. If the array is already sorted then the inversion count is 0. If the array is sorted in reverse order, then the inversion count is the maximum. For example, if the given array is 8 4 2 1, your program should output 6 (the six inversions are (8, 4), (8, 2), (8, 1), (4, 2), (4, 1), (2, 1)).