Computing Laboratory

Balanced Search Trees

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

December, 2021

1 Basics

2 Implementation

Balanced search trees

Definition

A balanced search tree is a binary search tree that automatically keeps its height the smallest possible for any arbitrary insertion or deletion.

Main operations

- **Insertion**
- **Search**
- **Deletion**

Auxiliary operations

- **Find successor**
- **Rotate on insertion**
- **Balance**

AVL trees

AVL trees are height balancing trees.

Definition

The balance factor of a node n, say $bf(n)$, in a binary tree is defined to be the height difference between its right subtree and left subtree.

Definition

A binary tree T is defined to be an AVL tree if for every node $n \in T$ the following property holds.

$$\forall n \in T : bf(n) \in \{-1, 0, 1\}$$

Time for insertion / search in AVL trees

Data Structure	Worst case	Average case
Ordinary binary search trees	$O(N)$	$O(\lg N)$
AVL trees	$O(\lg N)$	

Typedefs (avl-alt.h) – Alternative

```
typedef int DATA; // Generic use: typedef void * DATA;

typedef struct node{
    DATA data;
    int left, right, parent, height;
}AVL_NODE;

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

Main functions (avl-alt.h) – Alternative

```
extern int init_tree(TREE *);
extern int search(TREE *, int , DATA); // 2nd argument optional
extern int insert(TREE *, int , int *, DATA); // 2nd argument optional
extern int delete(TREE *, int , int *, DATA); // 2nd argument optional

#define DELETE_TREE(tree) free(tree->nodelist);
```

Note: The optional arguments are representing the parent index.

Auxiliary functions (avl-alt.h) – Alternative

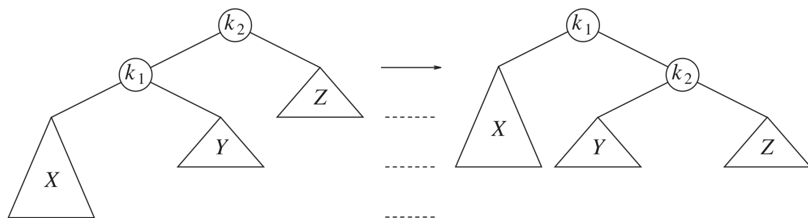
```
extern int grow_tree(TREE *);
extern int get_new_node(TREE *);
extern void free_up_node(TREE *, int);
extern int find_successor(TREE *, int);
extern void rotate_on_insert_LL(TREE *, int , int *);
extern void rotate_on_insert_RR(TREE *, int , int *);
extern void rotate_on_insert_LR(TREE *, int , int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

Finding the 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){
        return child;
    }
    while(tree->nodelist[child].left != -1){
        node = child;
        child = tree->nodelist[child].left;
    }
    return child;
}
```

Rotate LL

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



Rotate LL - The code

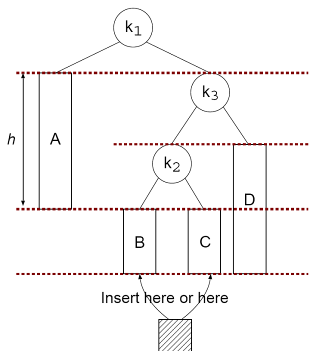
```
void rotate_on_insert_LL(TREE *tree, int parent, int *node){
    #ifdef DEBUG
        printf("LL (right) rotation at %d\n",
                                    tree->nodelist[*node].data);
    #endif // DEBUG
    int k2 = *node;
    int k1 = tree->nodelist[k2].left;
    int Z = tree->nodelist[k2].right;
    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;
    tree->nodelist[k2].parent = k1;
    if(Y != -1)
        tree->nodelist[Y].parent = k2;
```

Rotate LL - The code (contd.)

```
    /* 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

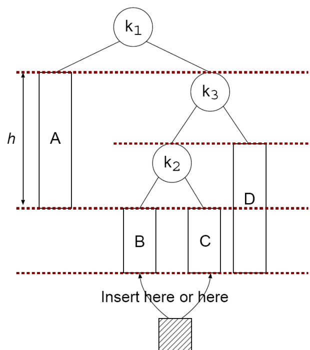
- Insertion into **L**eft subtree of **R**ight subtree causes imbalance



h - height
bf - balance factor

Rotate RL

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



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

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

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

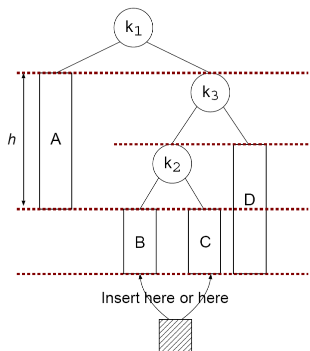
$$\max(h_{old}(k_2), h(D)) = h$$

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

h - height
bf - balance factor

Rotate RL

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



h - height
bf - balance factor

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

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

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

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

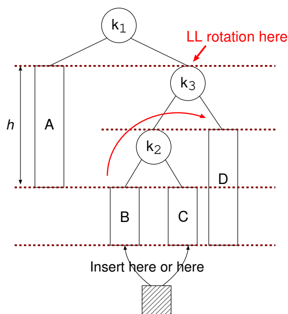
$\implies \quad h_{new}(k_2) = h + 1, h_{old}(k_2) = h$

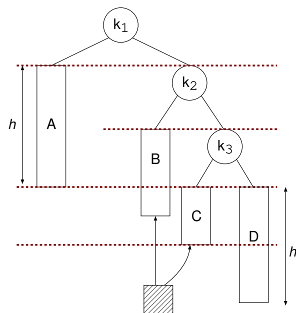
$h(D) \in \{h, h - 1\}$

If $h(D) = h - 1$, imbalance would be observed at k_3 before k_1

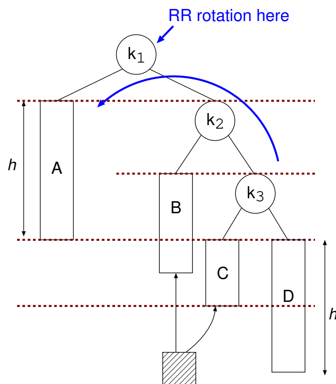
$\implies \quad h(D) = h$

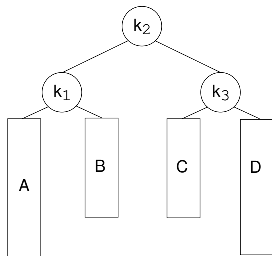
Rotate RL – I





Rotate RL – II





Rotate RL - code

```
void rotate_on_insert_RL(TREE *tree, int parent, int *node){
    #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

```
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 (contd.)

```
        #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;
}
```

Deletion

```
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
```

Deletion (contd.)

```
        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))
```

Deletion (contd.)

```
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
}
```

Deletion (contd.)

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

Problems – Day 16

1. 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.
   The first line of the input is an integer n representing the number of nodes in the binary tree. This will be followed by n more lines (count starts from 1 representing the line where the root appears) corresponding to one node in the tree and will consist of 5 integers: the data (stored in the node), the line number corresponding to the left child (-1 if there is no left child), the line number corresponding to the right child (-1 if there is no right child), the line number corresponding to the parent, and the height of the node.

Problems – Day 16

2. 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 $(8,4),(8,2),(8,1),(4,2),(4,1),(2,1)$).