CS 261 – Data Structures

AVL Trees
AVL Implementation

```
struct AVLNode {
    TYPE val;
    struct AVLNode *left;
    struct AVLNode *rght;
    int hght; /* Height of node*/
};
```
Add – Insert at the Leaf Level

Add Adam

Before:
- Alex
  - Abner
    - Abigail
    - Adela
  - Angela

After:
- Alex
  - Abner
    - Adam
  - Angela

Height:
- Before: height = 0
- After: height = 2
Add – Insert at the Leaf Level

Rotation right

Is it double rotation?
Add – Insert at the Leaf Level

left child is heavy on the right

rotation right

height = 0
height = 2
rotation right
height = 1
height = 0
Add – Insert at the Leaf Level

Double rotation right
Add – Insert at the Leaf Level

Double rotation right
Add – Insert at the Leaf Level

Double rotation right

height = 1
height = 1
Add – Recursive Function

struct AVLNode * _addAVLNode(struct AVLNode* current, TYPE e) {
    ...
    if (current == 0) { /* stop recursion */
        ...
        /* allocate memory for newnode; add newnode to tree*/
        return newnode;
    }
    else {
        /* recursively move down the tree */
        }
    return /* ????? */;
}
struct AVLNode *_addAVLNode(struct AVLNode* current, TYPE e) {
    ...
    if (current == 0) { /* stop recursion */
        ...
        /* allocate memory for newnode; add newnode to tree*/
        return newnode; /* why not call balance() here? */
    }
    else {
        /* recursively move down the tree */
    }
    return balance(current);
}
Add – Recursive Function

```c
struct AVLNode *addAVLNode(struct AVLNode* current, TYPE e)
{
    ...
    if (current == 0) { /* stop recursion */
        ... /* allocate memory for newnode; add newnode to tree*/
        return newnode;
    }
    else { /* recursively move down the tree */
        if( LT(e, current->value) )
            current->left = addAVLNode(current->left, e);
        else
            current->right = addAVLNode(current->right, e);
    }
    return balance(current);
}
```
Rotation

Rotate left

Rotate right
Double Rotation

heavy on right

Rotate left the left child

Rotate right
Balance

struct AVLNode * balance (struct AVLNode * current){

    int rotation = _height(current->right) - _height(current->left);

    if (rotation < -1) {
        /* (double) rotation right */
    } else if (rotation > 1) {
        /* (double) rotation left */
    }

    _setHeight(current);

    return current;
}
Balance – (Double) Rotation Right

...  

if (rotation < -1) { /* (double) rotation right */
    int drotation = _height(current->left->right)
                   - _height(current->left->left);

    if (drotation > 0){ /* double rotation */
        /* left child is heavy on the right */
        current->left = _rotateLeft(current->left);
    }

    return _rotateRight(current);
}

else { ...
Balance – (Double) Rotation Left

...}

} else if (rotation > 1) { /* (double) rotation left */
    int drotation = _height(current->right->right)
                 - _height(current->right->left);
    if (drotation < 0) { /* double rotation */
        /* right child is heavy on the left */
        current->right = _rotateRight(current->right);
    }
    return _rotateLeft(current);
} ...
Rotation Left

struct AVLNode * _rotateLeft (struct AVLNode * current) {
    struct AVLNode * newtop = current->right;
    ...
}
Rotation Left

```c
struct AVLNode * _rotateLeft (struct AVLNode * current) {
    struct AVLNode * newtop = current->right;
    current->right = newtop->left;
    newtop->left = current;
    ...
}
```
struct AVLNode * _rotateLeft (struct AVLNode * current) {
    struct AVLNode * newtop = current->right;
    current->right = newtop->left;
    newtop->left = current;
    _setHeight(current);
    _setHeight(newtop);
    return newtop;
}
struct AVLNode * _rotateRight (struct AVLNode * current) {
    struct AVLNode * newtop = current->left;
    current->left = newtop->right;
    newtop->right = current;
    _setHeight(current);
    _setHeight(newtop);
    return newtop;
}
Remove: Who fills the hole in the tree?

Answer:

the leftmost child of the right child
(smallest element in right subtree)
void removeAVLTree(struct AVLTree *tree, TYPE val) {
    if (containsAVLTree(tree, val)) {
        tree->root = _removeNode(tree->root, val);
        tree->cnt--;
    }
}
struct AVLNode * _removeNode(struct AVLNode *current, TYPE e) {
    struct AVLNode *temp;
    assert(current);
    if (EQ(e, current->val)) {
        /* replace current with the leftmost descendant of the right child */
    }
    else if (LT(e, current->val))
        current->left = _removeNode(current->left, e);
    else
        current->rght = _removeNode(current->rght, e);

    return balance(current);
}
AVL Trees: Sorting

• An AVL tree can sort a collection of values:

1. Copy data into the AVL tree: $O(??)$

2. Copy them out using the ?? traversal: $O(??)$
AVL Trees: Sorting

• An AVL tree can sort a collection of values:
  
  Copy data into the AVL tree:
  
  \[ O(n \log_2 n) \]

  Copy them out using the **in-order traversal**:
  
  \[ O(n) \]
AVL Trees: Sorting

• Execution time $\rightarrow O(n \log n)$:
  – Matches that of quick sort in benchmarks
  – Unlike quick sort, AVL trees don’t have problems if data is already sorted or almost sorted (which degrades quick sort to $O(n^2)$)

• However, requires extra storage to maintain both the original data buffer (e.g., a DynArr) and the tree structure