COMP 2012H Honors Object-Oriented Programming and Data Structures

Topic 20: AVL Trees

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Motivation

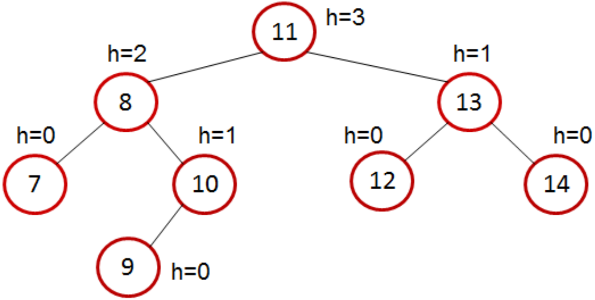
- A binary search trees (BST) supports efficient searching if it is well balanced — its nodes are fairly evenly distributed on both its left and right sub-trees.
- However, this is not always the case as insertions and deletions of tree nodes will generally make the resulting BST unbalanced.
- In the worst case, the tree is de-generated to a sorted linked list and the searching time is linear time.

Target: A balanced binary search tree

A BST with N nodes and a height of the order of $\log N$.

AVL (Adelson-Velsky and Landis) Trees

- An AVL tree is a BST where the height of the two sub-trees of ANY of its nodes may differ by at most one.
- Each node stores a height value, which is used to check if the tree is balanced or not.



AVL Trees

AVL Tree Properties

Every sub-tree of an AVL tree is itself an AVL tree.
(An empty tree is an AVL tree too.)

- With this property, an AVL tree is balanced and it is guaranteed that its height is logarithmic in the number of nodes, N. i.e., order of $\log(N)$.
- Efficiency of its following tree operations can always be guaranteed.
  - Searching: order of log(N) in the worst case
  - Insertion: order of log(N) in the worst case
  - Deletion: order of log(N) in the worst case

AVL Tree Implementation I

```
template <typename T>         /* File: avl.h */
class AVL
{
  private:
    struct AVLnode
    {
        T value;
        int height;
        AVL left;                // Left subtree is also an AVL object
        AVL right;               // Right subtree is also an AVL object
        AVLnode(const T& x) : value(x), height(0) { }
        // AVLnode(const T& x) : value(x), height(0), left(), right() { }
        AVLnode(const AVLnode& node) = default; // Copy constructor
        // AVLnode(const AVLnode& node)         // Equivalent
        //     : value(node.value), height(node.heigjt),
        //       left(node.left), right(node.right) { }
        ~AVLnode() { cout << "delete: " << value << endl; }
    };

    AVLnode* root = nullptr;
```

AVL Tree Implementation II

```
    AVL& right_subtree() { return root->right; }
    AVL& left_subtree() { return root->left; }
    const AVL& right_subtree() const { return root->right; }
    const AVL& left_subtree() const { return root->left; }

    int height() const;       // Find the height of tree
    int bfactor() const;      // Find the balance factor of tree
    void fix_height() const; // Rectify the height of each node in tree
    void rotate_left();       // Single left or anti-clockwise rotation
    void rotate_right();      // Single right or clockwise rotation
    void balance();           // AVL tree balancing

  public:
    AVL() = default;          // Build an empty AVL tree by default
    ~AVL() { delete root; }   // Will delete the whole tree recursively!
    // Shallow AVL copy using move constructor
    AVL(AVL&& avl) { root = avl.root; avl.root = nullptr; }
```

AVL Tree Implementation III

```
    AVL(const AVL& avl)      // Deep copy using copy constructor
    {
        if (avl.is_empty())
            return;

        root = new AVLnode(*avl.root);  // Recursive
    }

    bool is_empty() const { return root == nullptr; }
    const T& find_min() const;       // Find the minimum value in an AVL
    bool contains(const T& x) const; // Search an item
    void print(int depth = 0) const; // Print by rotating -90 degrees

    void insert(const T& x); // Insert an item in sorted order
    void remove(const T& x); // Remove an item
};
```

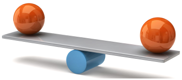
AVL Tree Searching

- Searching in AVL trees is the same as in BST.

```
// Goal: To search for an item x in an AVL tree
// Return: (bool) true if found, otherwise false
template <typename T>
bool AVL<T>::contains(const T& x) const
{
    if (is_empty())               // Base case #1
      return false;

    else if (x == root->value)  // Base case #2
      return true;

    else if (x < root->value)   // Recursion on the left subtree
       return left_subtree().contains(x);

   else                         // Recursion on the right subtree
       return right_subtree().contains(x);
}
```

AVL Tree Insertion and Rotation

- To insert an item in an AVL tree
  - Search the tree and locate the place where the new item should be inserted to.
  - Create a new node with the item and attach it to the tree.
- The insertion may cause the AVL tree unbalanced
  $\Rightarrow$ tree balancing by rotation(s)



- Types of rotation
  - single rotation
  - double rotation (i.e., two single rotations)

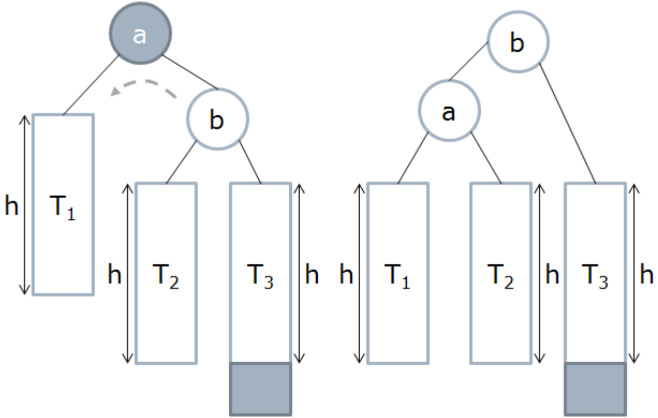
AVL Tree Insertion and Rotation ..

Insertion may violate the AVL tree property in 4 cases:

1. Right-Right (RR)
   Left (anti-clockwise) rotation [single rotation]:
   Insertion into the right sub-tree of the right child of a node
2. Left-Left (LL)
   Right (clockwise) rotation [single rotation]:
   Insertion into the left sub-tree of the left child of a node
3. Left-Right (LR)
   Left-right rotation [double rotation]:
   Insertion into the right sub-tree of the left child of a node
4. Right-Left (RL)
   Right-left rotation [double rotation]:
   Insertion into the left sub-tree of the right child of a node

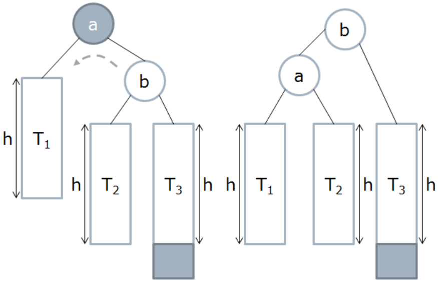
AVL Left (Anti-clockwise) Rotation

Left rotation at node a.



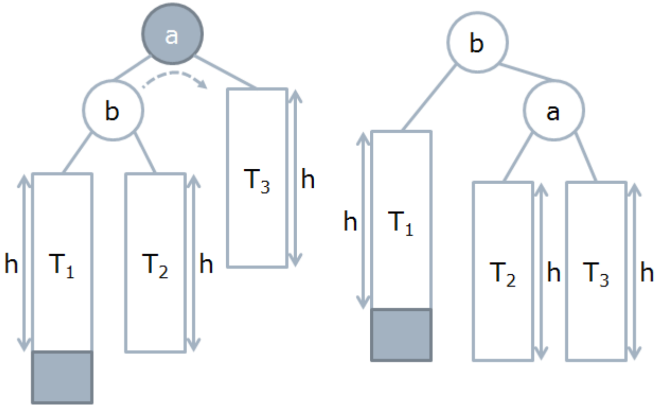
AVL Code: Left Rotation

```
/* Goal: To perform a single left (anti-clocwise) rotation */
template <typename T>
void AVL<T>::rotate_left() // The calling AVL node is node a
{
    AVLnode* b = right_subtree().root; // Points to node b
    right_subtree() = b->left;
    b->left = *this;     // Note: *this is node a
    fix_height();        // Fix the height of node a
    this->root = b;      // Node b becomes the new root
    fix_height();        // Fix the height of node b, now the new root
}
```



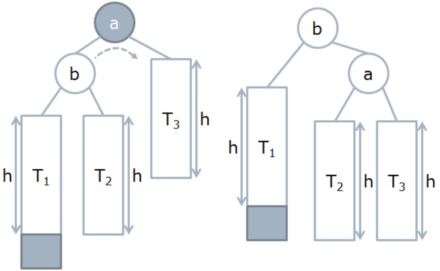
AVL Right (Clockwise) Rotation

Right rotation at node a.

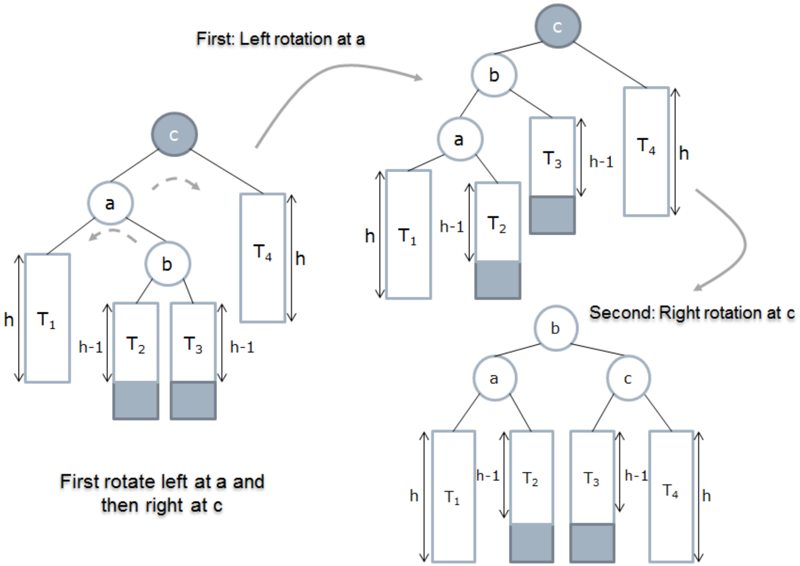


AVL Code: Right Rotation

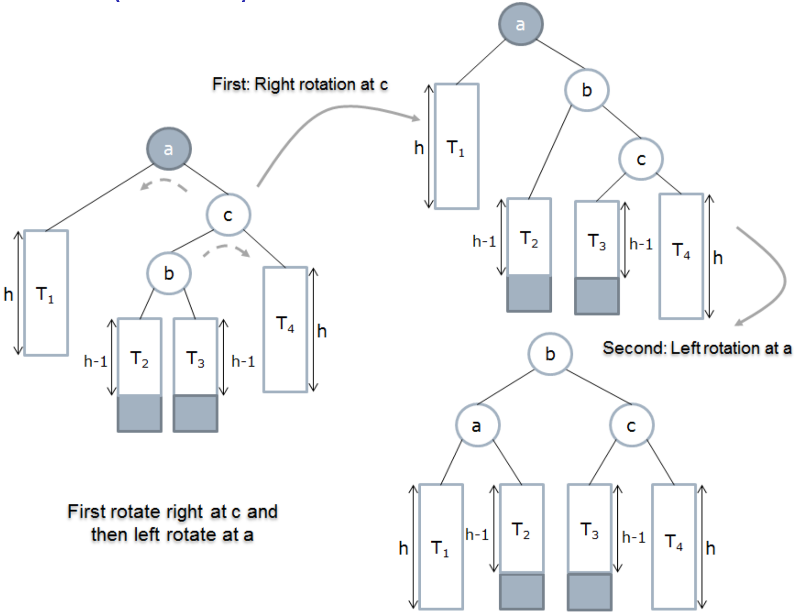
```
/* Goal: To perform right (clockwise) rotation */
template <typename T>
void AVL<T>::rotate_right() // The calling AVL node is node a
{
    AVLnode* b = left_subtree().root; // Points to node b
    left_subtree() = b->right;
    b->right = *this;    // Note: *this is node a
    fix_height();        // Fix the height of node a
    this->root = b;      // Node b becomes the new root
    fix_height();        // Fix the height of node b, now the new root
}
```



Left-Right (Double) Rotation



Right-Left (Double) Rotation



First rotate right at c and then left rotate at a

AVL Code: Insertion

```
/* To insert an item x to AVL tree and keep the tree balanced */

template <typename T>
void AVL<T>::insert(const T& x)
{
    if (is_empty())                    // Base case
        root = new AVLnode(x);

    else if (x < root->value)
        left_subtree().insert(x);  // Recursion on the left sub-tree

    else if (x > root->value)
        right_subtree().insert(x); // Recursion on the left sub-tree

    balance(); // Re-balance the tree at every visited node
}
```

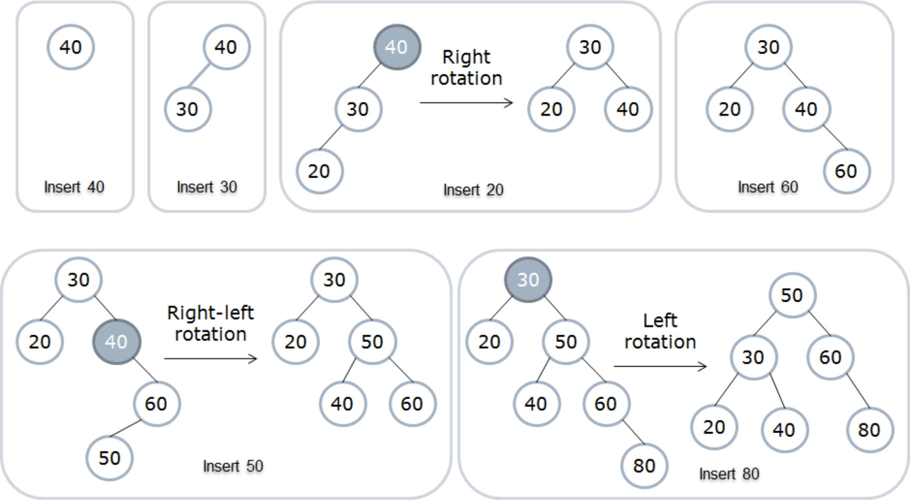
AVL Code: Balancing

```
/* Goal: To balance an AVL tree */
template <typename T>
void AVL<T>::balance()
{
    if (is_empty())
        return;

    fix_height();
    int balance_factor = bfactor();

    if (balance_factor == 2)       // Right subtree is taller by 2
    {
        if (right_subtree().bfactor() < 0) // Case 4: insertion to the L of RT
            right_subtree().rotate_right();
        return rotate_left();      // Cases 1 or 4: Insertion to the R/L of RT
    }
    else if (balance_factor == -2) // Left subtree is taller by 2
    {
        if (left_subtree().bfactor() > 0) // Case 3: insertion to the R of LT
            left_subtree().rotate_left();
        return rotate_right();     // Cases 2 or 3: insertion to the L/R of LT
    }
    // Balancing is not required for the remaining cases
}
```

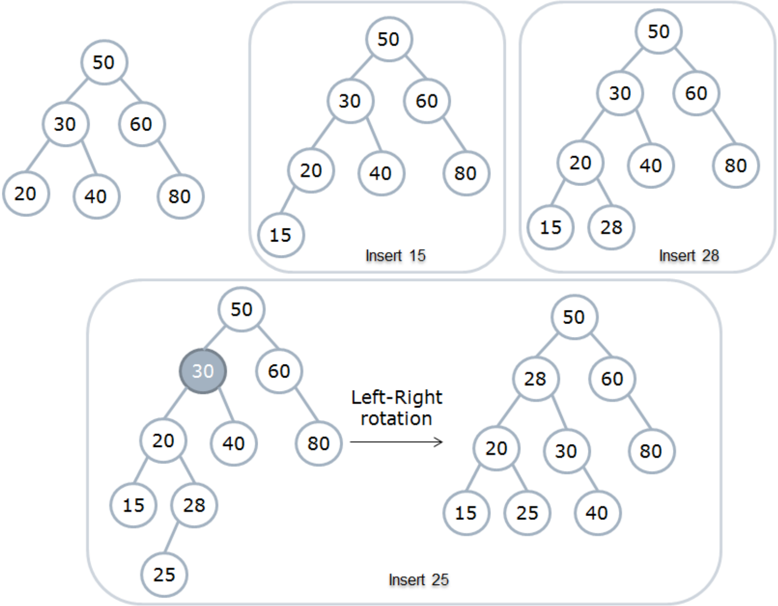
AVL Code: Balancing ..

```
/* To find the height of an AVL tree */
template <typename T>
int AVL<T>::height() const { return is_empty() ? -1 : root->height; }

/* Goal: To rectify the height values of each AVL node */
template <typename T>
void AVL<T>::fix_height() const
{
    if (!is_empty())
    {
        int left_avl_height = left_subtree().height();
        int right_avl_height = right_subtree().height();
        root->height = 1 + max(left_avl_height, right_avl_height);
    }
}

/* balance factor = height of right sub-tree - height of left sub-tree */
template <typename T>
int AVL<T>::bfactor() const
{
    return is_empty() ? 0
        : right_subtree().height() - left_subtree().height();
}
```

Example: AVL Tree Insertion



Example: AVL Tree Insertion ..



AVL Tree Deletion

To delete an item from an AVL tree.

1. Search and locate the node with the required key.
2. Delete the node like deleting a node in BST.
3. A node deletion may result in a unbalanced tree
   $\Rightarrow$ Re-balance the tree by rotation(s).
   - single rotation
   - double rotation (i.e. two single but different rotations)

AVL Tree Deletion ..

- Similar to node deletion in BST, 3 cases need to be considered
  1. The node to be removed is a leaf node

     $\Rightarrow$ Delete the leaf node immediately
  2. The node to be removed has 1 child

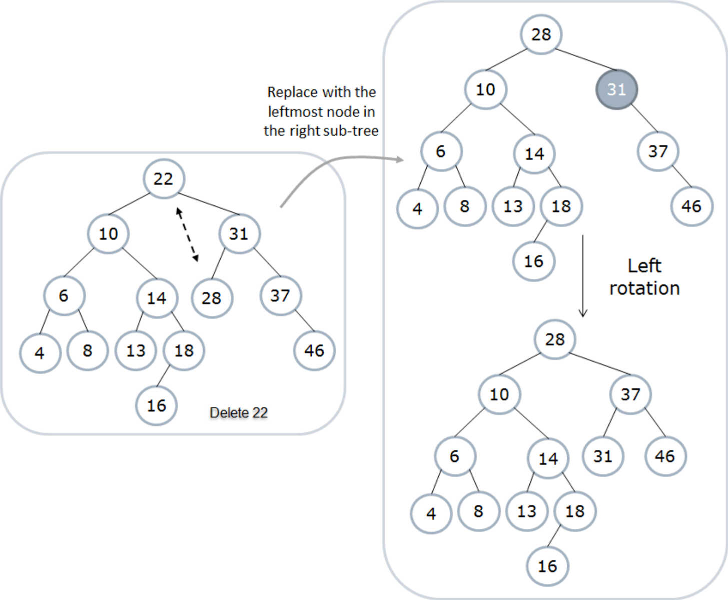
     $\Rightarrow$ Adjust a pointer to bypass the deleted node
  3. The node to be removed has 2 children

     $\Rightarrow$ Replace the node to be removed with either the

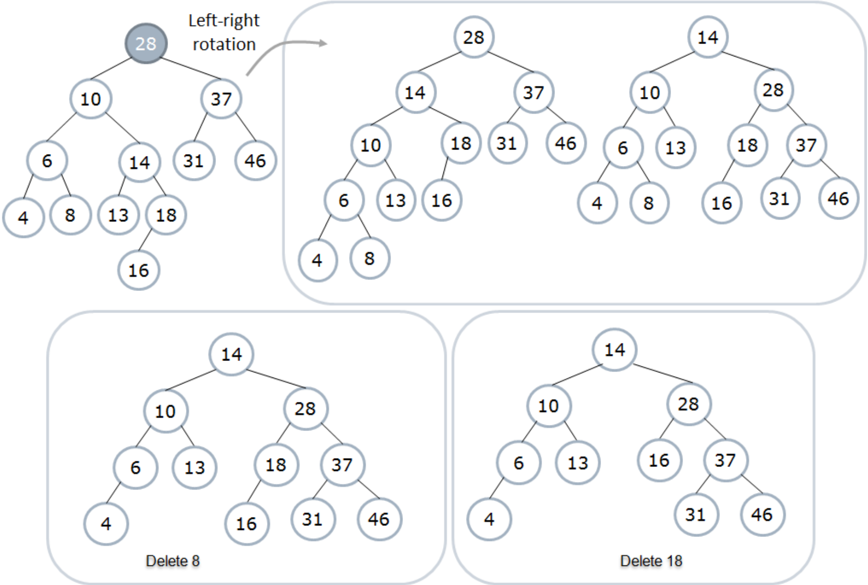
     • maximum node in its left sub-tree, or

     • minimum node in its right sub-tree Then remove the max/min node depending on the choice above.
- Removing a node can render multiple ancestors unbalanced
  $\Rightarrow$ every sub-tree affected by the deletion has to be re-balanced.

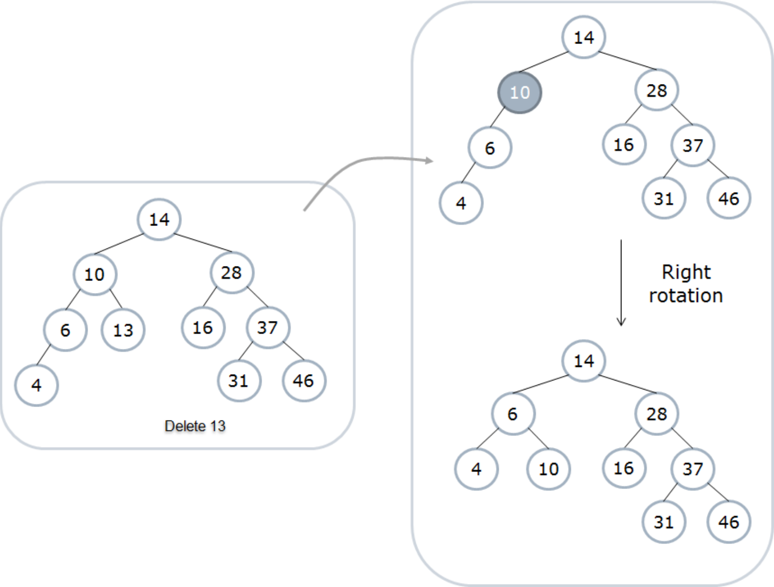
Example: AVL Tree Deletion



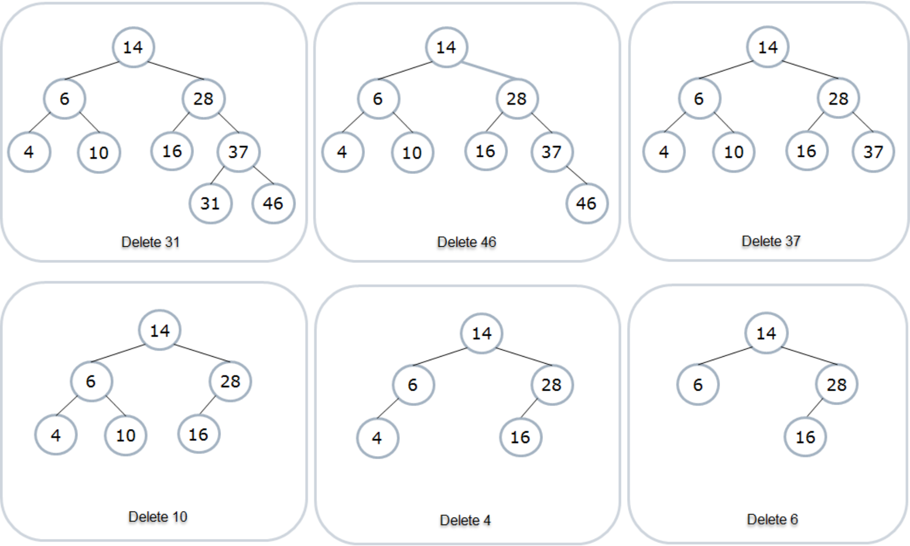
Example: AVL Tree Deletion ..



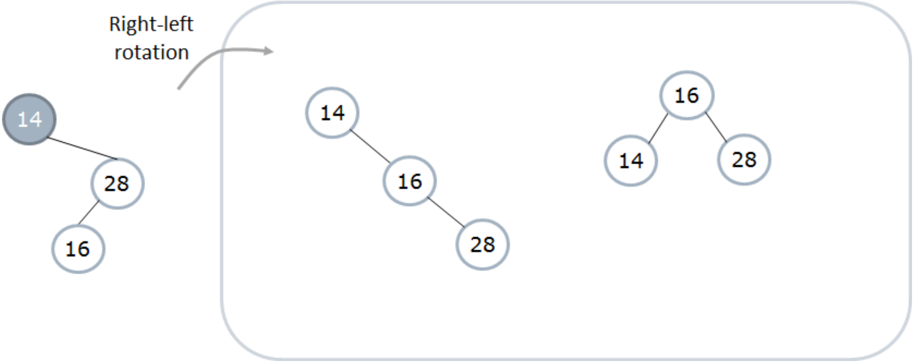
Example: AVL Tree Deletion ...



Example: AVL Tree Deletion ....



Example: AVL Tree Deletion .....



AVL Code: Deletion I

```
/* To remove an item x in AVL tree and keep the tree balanced */

template <typename T>
void AVL<T>::remove(const T& x)
{
    if (is_empty())                  // Item is not found; do nothing
        return;

    if (x < root->value)
        left_subtree().remove(x);   // Recursion on the left sub-tree

    else if (x > root->value)
        right_subtree().remove(x); // Recursion on the right sub-tree

    else
    {
        AVL& left_avl = left_subtree();
        AVL& right_avl = right_subtree();
```

AVL Code: Deletion II

```
        // Found node has 2 children
        if (!left_avl.is_empty() && !right_avl.is_empty())
        {
            root->value = right_avl.find_min(); // Copy the min value
            right_avl.remove(root->value); // Remove node with min value
        }

        else  // Found node has 0 or 1 child
        {
            AVLnode* node_to_remove = root; // Save the node first
            *this = left_avl.is_empty() ? right_avl : left_avl;

            // Reset the node to be removed with empty children
            right_avl.root = left_avl.root = nullptr;
            delete node_to_remove;
        }
    }

    balance(); // Re-balance the tree at every visited node
}
```

AVL Code: Find the Minimum Value

```
/* To find the minimum value stored in an AVL tree. */

template <typename T>
const T& AVL<T>::find_min() const
{
    // It is assumed that the calling tree is not empty
    const AVL& left_avl = left_subtree();

    if (left_avl.is_empty())    // Base case: Found!
        return root->value;

    return left_avl.find_min(); // Recursion on the left subtree
}
```

AVL Testing Code

```
/* File: avl.tpp
 *
 * It contains template header and all the template functions
 */

#include "avl.h"
#include "avl-balance.cpp"
#include "avl-bfactor.cpp"
#include "avl-contains.cpp"
#include "avl-find-min.cpp"
#include "avl-fix-height.cpp"
#include "avl-height.cpp"
#include "avl-insert.cpp"
#include "avl-print.cpp"
#include "avl-remove.cpp"
#include "avl-rotate-left.cpp"
#include "avl-rotate-right.cpp"
```

AVL Testing Code .. I

```
#include <iostream>     /* File: test-avl.cpp */
using namespace std;
#include "avl.tpp"

int main()
{
    AVL<int> avl_tree;
    while(true)
    {
        char choice; int value;
        cout << "Action: f/i/m/p/q/r (end/find/insert/min/print/remove): ";
        cin >> choice;

        switch(choice)
        {
            case 'f':
                cout << "Value to find: "; cin >> value;
                cout << boolalpha << avl_tree.contains(value) << endl;
                break;

            case 'i':
                cout << "Value to insert: "; cin >> value;
                avl_tree.insert(value);
                break;
```

AVL Testing Code .. II

```
        case 'm':
            if (avl_tree.is_empty())
                cerr << "Can't search an empty tree!" << endl;
            else
                cout << avl_tree.find_min() << endl;
            break;

        case 'p':
            avl_tree.print();
            break;

        case 'q': default:
            return 0;

        case 'r':
            cout << "Value to remove: "; cin >> value;
            avl_tree.remove(value);
            break;
    }
  }
}
```

AVL Trees: Pros and Cons

Pros:

- Time complexity for searching is in the order of $log(N)$ since AVL trees are always balanced.
- Insertion and deletions are also in the order of $log(N)$ since the operation is dominated by the searching step.
- The tree re-balancing step adds no more than a constant factor to the time complexity of insertion and deletion.

Cons:

- A bit more space for storing the height of an AVL node.

That's all!

Any questions?