Inf 2B: AVL Trees

Lecture 5 of ADS thread

Kyriakos Kalorkoti

School of Informatics
University of Edinburgh

Dictionaries

A *Dictionary* stores key–element pairs, called *items*. Several elements might have the same key. Provides three methods:

- findElement(k): If the dictionary contains an item with key k, then return its element; otherwise return the special element NO_SUCH_KEY.
- insertItem(k, e): Insert an item with key k and element e.
- removeItem(k): If the dictionary contains an item with key k, then delete it and return its element; otherwise return NO_SUCH_KEY.

Assumption: we have a total order on keys (always the case in applications).

Note: We are concerned entirely with fast access and storage so focus on keys.

ADT *Dictionary* & its implementations

List implementation:
$\Theta(1)$ time for InsertItem(k, e) but $\Theta(n)$ for findElement(k) and removeItem(k).

HashTable implementation (with Bucket Arrays):
Good average-case performance for $n = \Omega(N)$.
Worst-case running time: is InsertItem(k, e) $\Theta(1)$, findElement(k) and removeItem(k) are both $\Theta(n)$.

Binary Search Tree implem. (without Balancing):
Good in the average-case—about $\Theta(\lg n)$ for all operations.
Worst-case running time: $\Theta(n)$ for all operations.

Balanced Binary search trees:
Worst-case is $\Theta(\lg n)$ for all operations.

Binary Search Trees

Abstract definition: A *binary tree* is either empty or has a *root vertex* with a *left* and a *right child* each of which is a tree.

- Recursive datatype definition.

So every vertex v, either:
(i) has two children (v is an *internal vertex*), or
(ii) has no children (v is a *leaf*).

An internal vertex v has a *left* child and a *right* child which might be another internal vertex or a leaf.

A *near leaf* is an internal vertex with one or both children being leaves.

Definition

A tree storing (*key*, *element*) pairs is a Binary Search Tree if for *every* internal vertex v, the key k of v is:

- *greater than or equal to* every key in v's *left subtree*, and
- *less than or equal to* every key in v's right subtree.

Key parameter for runtimes: *height*

- Given any vertex v of a tree T and a leaf there is a unique path form the vertex to the leaf:
  - length of path defined as number of internal vertices.
- The *height* of a vertex is the maximum length over all paths from it to leaves.
- The height of a tree is the height of the root.
- Note that if v has left child l and right child r then

$$\text{height}(v) = 1 + \max\{\text{height}(l), \text{height}(r)\}.$$

- If we insert v_r along the path $v_1, v_2, \ldots, v_r$ then only the heights of $v_1, v_2, \ldots, v_r$ *might* be affected, all other vertices keep their previous height.

Binary Search Trees for *Dictionary*

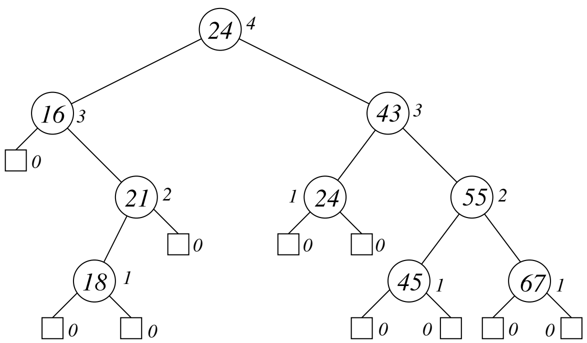
Leaves are kept empty.

Algorithm findElement(k)

```
if isEmpty(T) then return NO_SUCH_KEY
else
    u ← root
    while ((u is not null) and u.key ≠ k) do
        if (k < u.key) then u ← u.left
        else u ← u.right
    od
    if (u is not null) and u.key = k then return u.elt
    else return NO_SUCH_KEY
```

findElement runs in $O(h)$ time, where h is height.

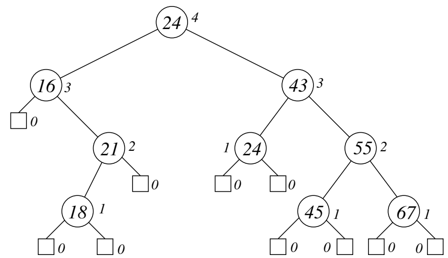
Binary Search Trees



Binary Search Trees for *Dictionary*

Algorithm insertItemBST(k, e)

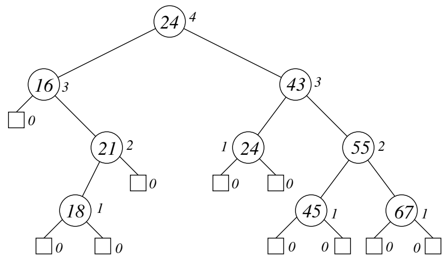
1. Perform findElement(k) to find the “right" place for an item with key k (if it finds k high in the tree, walk down to the “near-leaf” with largest key no greater than k).
2. Neighbouring leaf vertex u becomes internal vertex, $u.key \leftarrow k$, $u.elt \leftarrow e$.



Binary Search Trees for *Dictionary*

Algorithm removeItemBST(k)

1. Perform findElement(k) on the tree to get to vertex t.
2. **if** we find t with $t.key = k$,
3. **then** remove the item at t, set $e = t.elt$.
4. Let u be “near-leaf" closest to k. Move u’s item up to t.
5. **else return** NO_SUCH_KEY



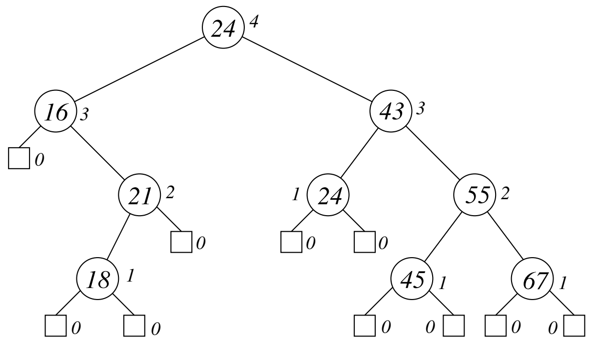
Worst-case running time

Theorem: *For the* *binary search tree* *implementation of Dictionary, all methods (**findElement**, **insertItemBST**, **removeItemBST**) have asymptotic worst-case running time* $\Theta(h)$*, where h is the height of the tree. (can be* $\Theta(n)$*).*

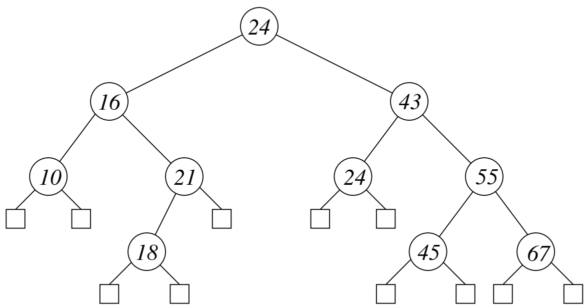
AVL Trees (G.M. Adelson-Velsky & E.M. Landis, 1962)

1. A vertex of a tree is *balanced* if the heights of its children differ by at most 1.
2. An *AVL tree* is a binary search tree in which all vertices are balanced.

Not an AVL tree:



An AVL tree



The height of AVL trees

Theorem: *The height of an AVL tree storing n items is* $O(\lg(n))$.

Corollary: *The running time of the binary search tree methods* **findElement**, **insertItem**, **removeItem** *is* $O(\lg(n))$ *on an AVL tree.*

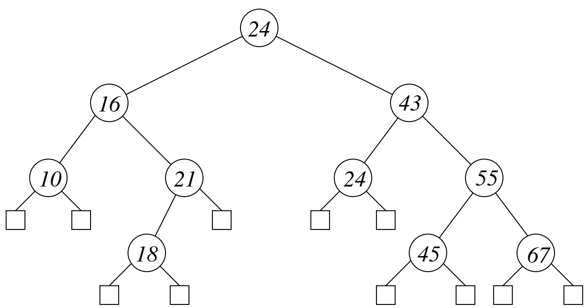
Let $n(h)$ denote minimum number of items stored in an AVL tree of height h. So $n(1) = 1$, $n(2) = 2$, $n(3) = 4$.

Claim: $n(h) > 2^{h/2} - 1$.

$$
\begin{aligned}
n(h) &\geq 1 + n(h-1) + n(h-2) \\
&> 1 + 2^{\frac{h-1}{2}} - 1 + 2^{\frac{h-2}{2}} - 1 \\
&= (2^{-\frac{1}{2}} + 2^{-1})\, 2^{\frac{h}{2}} - 1 \\
&> 2^{\frac{h}{2}} - 1.
\end{aligned}
$$

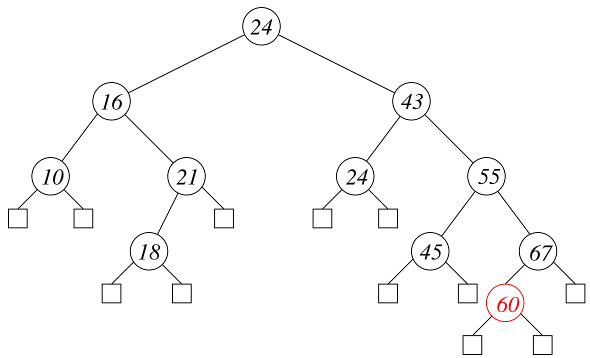
Problem: After we apply **insertItem** or **removeItem** to an AVL tree, the resulting tree might no longer be an AVL tree.

Example



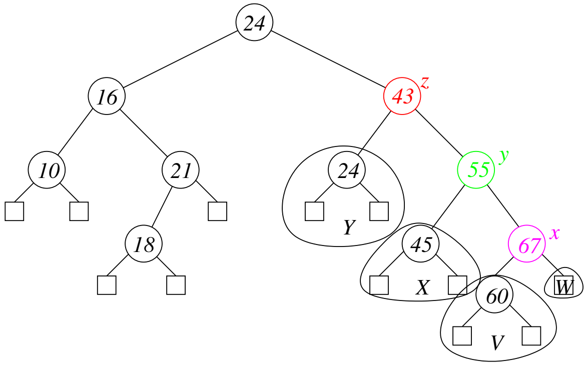
AVL tree. INSERT 60

Example (cont'd)



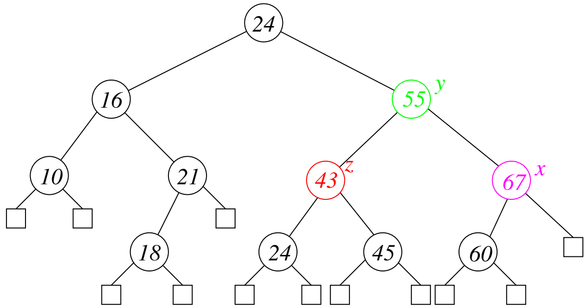
not AVL now . . .

Example (cont'd)



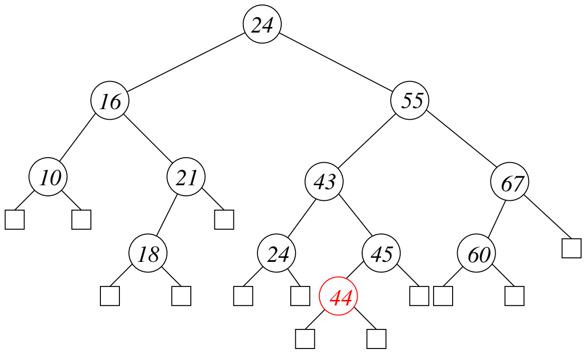
We can rotate . . .

Example (cont'd)



Now is AVL tree. INSERT 44

Example (cont'd)



AVL tree.

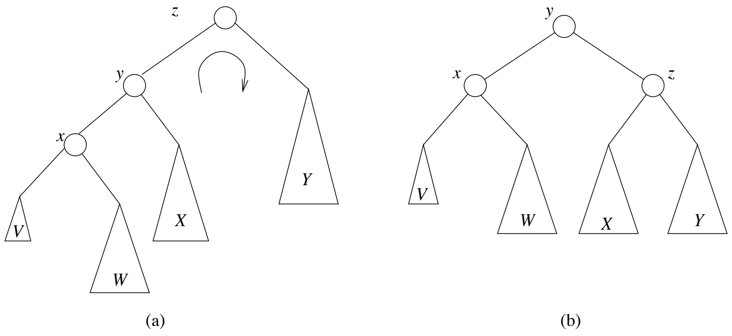
Restructuring

- z unbalanced vertex of minimal height
- y child of z of larger height
- x child of y of larger height (exists because 1 ins/del unbalanced the tree).
- V, W subtrees rooted at children of x
- X subtree rooted at sibling of x
- Y subtree rooted at sibling of y

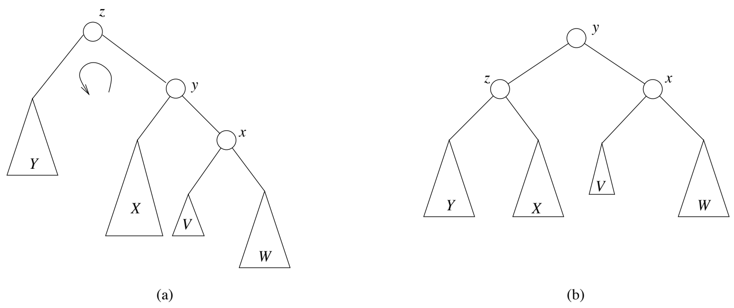
Then

$\text{height}(V) - 1 \leq \text{height}(W) \leq \text{height}(V) + 1$
$\max\{\text{height}(V), \text{height}(W)\} = \text{height}(X)$
$\max\{\text{height}(V), \text{height}(W)\} = \text{height}(Y)$.

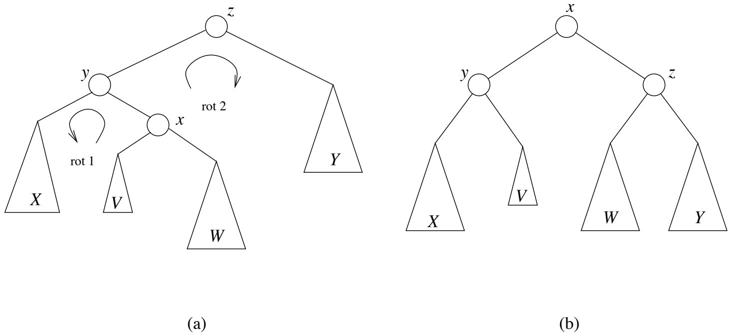
A clockwise single rotation



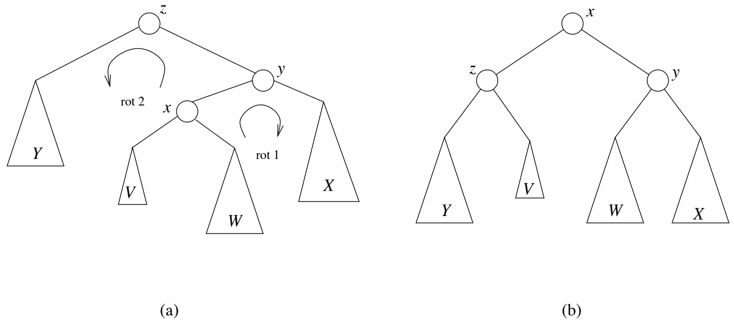
An anti-clockwise single rotation



An anti-clockwise clockwise double rotation



A clockwise anti-clockwise double rotation



Rotations

After an InsertItem():
We can always rebalance using just one *single rotation* or one *double rotation* (only 2x2 cases in total).

single rotation:
We make y the new root (of rebalancing subtree), z moves down, and the X subtree crosses to become 2nd child of z (with X as sibling).

double rotation:
We make x the new root, y and z become its children, and the two subtrees of x get split between each side.

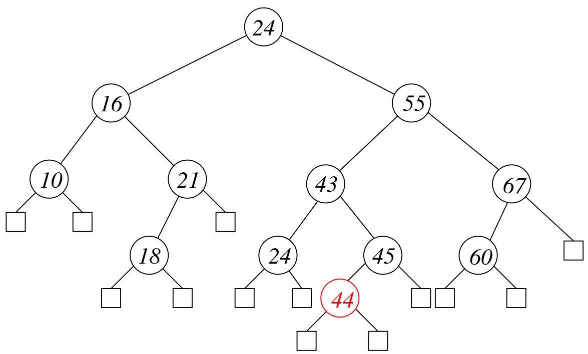
$\Theta(1)$ time for a single or double rotation.

The insertion algorithm

Algorithm insertItem(k, e)

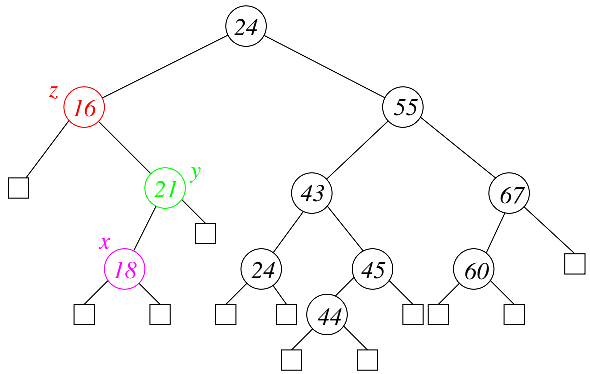
1. Insert (k, e) into the tree with insertItemBST.
   Let u be the newly inserted vertex.
2. Find first unbalanced vertex z on the path from u to root.
3. **if** there is no such vertex,
4. **then return**
5. **else** Let y and x be child, grandchild of z on $z \to u$ path.
6. Apply the appropriate rotation to x, y, z. **return**

Example (cont'd)



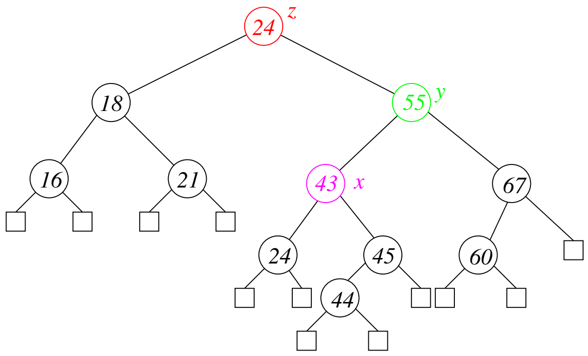
AVL tree. REMOVE 10.

Example (cont'd)



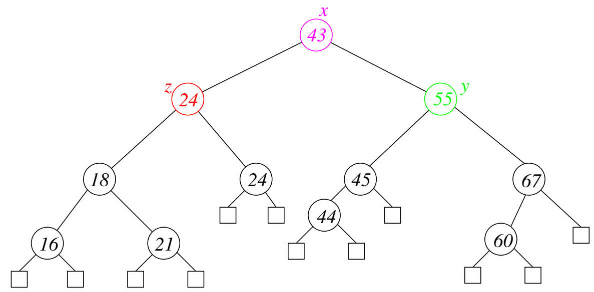
Not AVL tree ... We rotate

Example (cont'd)



Still not AVL . . . We rotate again.

Example (cont'd)



AVL tree again.

Rotations

After a removeItem():
We may need to re-balance "up the tree".

This requires $O(\lg n)$ rotations at most, each takes $O(1)$ time.

The removal algorithm

Algorithm removeItem(k)

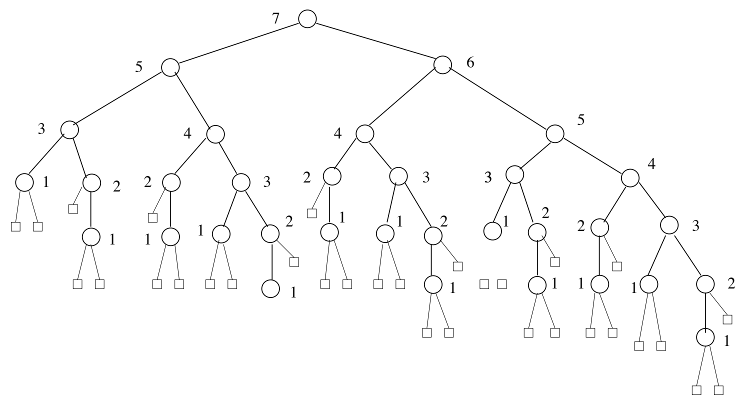
1. Remove item (k, e) with key k from tree using removeItemBST. Let u be leaf replacing removed vertex.
2. **while** u is not the root **do**
3.   let z be the parent of u
4.   **if** z is unbalanced **then**
5.   do the appropriate rotation at z
6.   let u be the parent of u
7. **return** e

Question on heights of AVL trees

- By definition of an AVL tree, for every internal vertex v, the difference between the *height* of the *left child* of v and the *right child of* v is at most 1.
- How large a difference can there be in the heights of *any two vertices at the same "level"* of an AVL tree?
  - 1.
  - 2.
  - At most $\lg(n)$.
  - Up to n.

Answer: At most $\lg(n)$.

Example of "globally-less-balanced" AVL tree



For this example, $n = 33$, $\lg(n) > 5$.

Ordered Dictionaries

The *OrderedDictionary* ADT is an extension of the *Dictionary* ADT that supports the following additional methods:

- closestKeyBefore(k): Return the key of the item with the largest key less than or equal to k.
- closestElemBefore(k): Return the element of the item with the largest key less than or equal to k.
- closestKeyAfter(k): Return the key of the item with the smallest key greater than or equal to k.
- closestElemAfter(k): Return the element of the item with the smallest key greater than or equal to k.

Range Queries

findAllItemsBetween(k_1, k_2): Return a list of all items whose key is between k_1 and k_2.
Binary Search Trees support Ordered Dictionaries AND Range Queries well.

Reading and Resources

- If you have [GT]:
  The Chapter on “Binary Search Trees" has a nice treatment of AVL trees. The chapter on “Trees" has details of tree traversal etc.
- If you have [CLRS]:
  The balanced trees are Red-Black trees, a bit different from AVL trees.