CS121 Data Structures A, C
Balanced Search Trees: AVL Trees

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

Balanced Search Trees

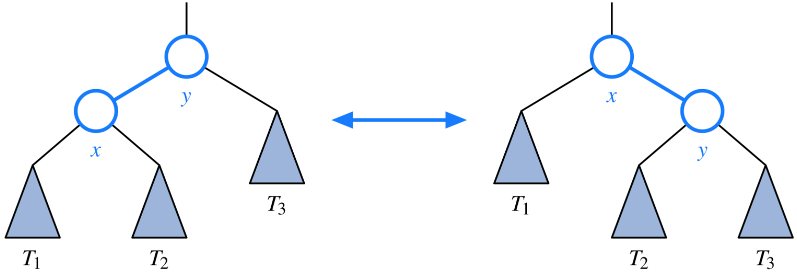
We introduce the intuitive concept of **balanced search trees** which are binary search trees where the depths of the leaves are not much different.

If a binary search tree is not balanced, we will call it **unbalanced search tree**.

AVL trees are an example of balanced search trees. They rely on special operations to reshape the tree and reduce its height.

Rotation

Rotation is a primary operation to rebalance a binary search tree which 'rotates' a child to be above its parent.



A single rotation modifies a constant number of parent-child relationships thus it can be implemented in $O(1)$ time with a linked binary tree representation.

Rotation (cont'd)

To maintain the binary search-tree property through a rotation, we note that if position x was a left child of position y prior to a rotation, then y becomes the *right* child of x after the rotation, and vice versa

Furthermore, we must relink the subtree of entries with keys that lie between the keys of the two positions that are being rotated

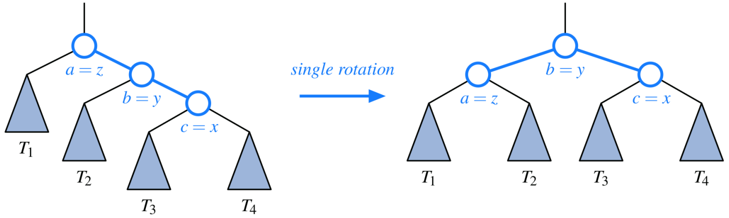
In the context of a tree-balancing algorithm, a rotation allows the shape of a tree to be modified while maintaining the search-tree property

If used wisely, this operation can be performed to avoid highly unbalanced tree configurations

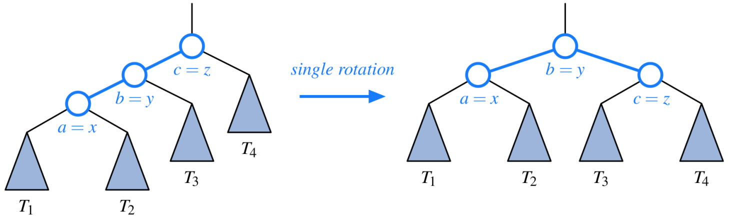
One or more rotations can be combined to provide broader rebalancing within a tree

Trinode Restructuring: Single Rotation

Trinode Restructuring considers a position x, its parent y, and its grandparent z



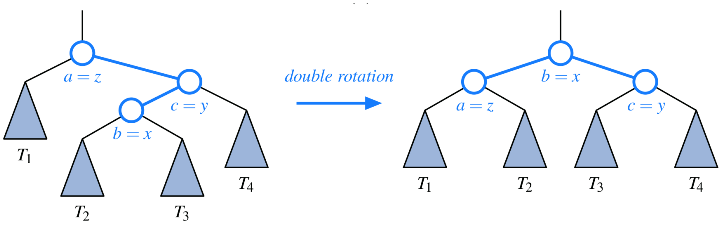
(a)



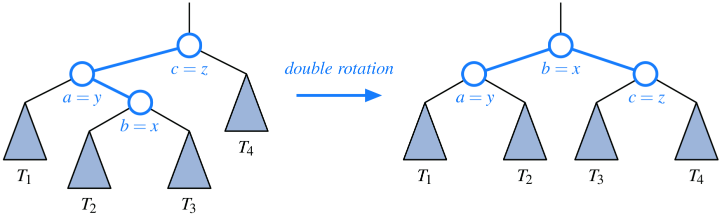
(b)

Trinode Restructuring: Double Rotation

Trinode Restructuring considers a position x, its parent y, and its grandparent z



(c)



(d)

Trinode Restructuring Algorithm

```
Algorithm restructure($x$):
  Input: A position $x$ of a binary search tree $T$ that has both a parent $y$ and a grandparent $z$
  Output: Tree $T$ after a trinode restructuring (which corresponds to a single or double rotation) involving positions $x$, $y$, and $z$

  1: Let $(a, b, c)$ be the left-to-right (inorder) listing of $x, y, z$ and let $(T_1, T_2, T_3, T_4)$ be a left-to-right (inorder) listing of the four subtrees of $x, y$, and $z$ not rooted at $x, y$, or $z$

  2: Replace the subtree rooted at $z$ with a new subtree rooted at $b$

  3: Let $a$ be the left child of $b$ and let $T_1$ and $T_2$ be the left and right subtrees of $a$, respectively

  4: Let $c$ be the right child of $b$ and let $T_3$ and $T_4$ be the left and right subtrees of $c$, respectively
```

What is the running time of this algorithm?

Trinode Restructuring Algorithm

```
Algorithm restructure($x$):
  Input: A position $x$ of a binary search tree $T$ that has both a parent $y$ and a grandparent $z$
  Output: Tree $T$ after a trinode restructuring (which corresponds to a single or double rotation) involving positions $x$, $y$, and $z$

  1: Let $(a, b, c)$ be the left-to-right (inorder) listing of $x, y, z$ and let $(T_1, T_2, T_3, T_4)$ be a left-to-right (inorder) listing of the four subtrees of $x, y$, and $z$ not rooted at $x, y$, or $z$

  2: Replace the subtree rooted at $z$ with a new subtree rooted at $b$

  3: Let $a$ be the left child of $b$ and let $T_1$ and $T_2$ be the left and right subtrees of $a$, respectively

  4: Let $c$ be the right child of $b$ and let $T_3$ and $T_4$ be the left and right subtrees of $c$, respectively
```

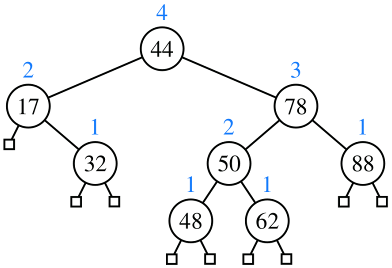
What is the running time of this algorithm? $O(1)$

AVL Tree

Our goal is to maintain a logarithmic height for a binary search tree

Height-Balance Property: For every internal position p of T, the heights of the children of p differ by at most 1

Any binary search tree T that satisfies the height-balance property is said to be an **AVL tree** (inventors: Adel'son-Vel'skii and Landis)



Consequence: a subtree of an AVL tree is itself an AVL tree.

Height of an AVL Tree

Proposition: The height of an AVL tree storing n entries is $O(\log n)$

Proof (by induction): Let us bound $n(h)$: the minimum number of *internal* nodes of an AVL tree of height h

- We easily see that $n(1) = 1$ and $n(2) = 2$
- For $h > 2$, an AVL tree of height h contains the root node, one AVL subtree of height $h-1$ and another of height $h-2$
- That is, $n(h) = 1 + n(h-1) + n(h-2)$
- Knowing $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$. So

$$n(h) > 2n(h-2) > 4n(h-4) > 8n(h-6), \ldots, \text{(by induction)}$$
$$n(h) > 2^i n(h-2i)$$

- Solving the base case we get: $n(h) > 2^{h/2-1}$
- Taking logarithms: $h < 2\log n(h) + 2$
- Thus the height of an AVL tree is $O(\log n)$

Balanced Positions

Given a binary search tree T, a position p in T is **balanced** if $|T.height(T.left(p)) - T.height(T.right(p))| \leq 1$

Otherwise, it is **unbalanced**

The height-balance property characterizing AVL trees is equivalent to saying that every position is balanced

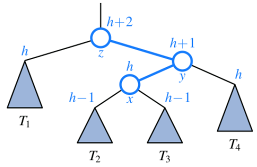
AVL Insertion

Insertion into an AVL tree is as in a binary search tree: always done by expanding an external position p

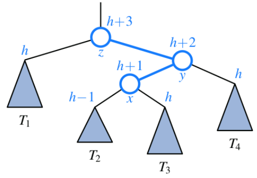
This action may violate the height-balance property

We rebalance the tree by finding the first unbalanced ancestor z of p with the *trinode restructuring* method

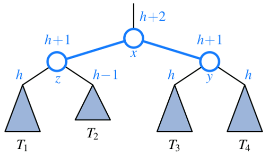
This restructuring restores the height-balance property *globally*



(a)



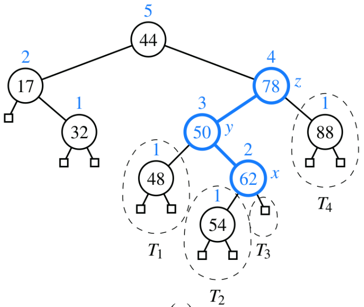
(b)



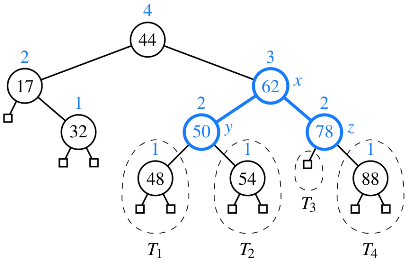
(c)

AVL Insertion Example

After insertion of 54: nodes of 78 and 44 are unbalanced



(a)



(b)

The restructuring operation rebalances the whole tree

Correctness of Rebalancing During AVL Insertion

Only ancestors of p may become unbalanced, because

Correctness of Rebalancing During AVL Insertion

Only ancestors of p may become unbalanced, because those are the only positions whose subtrees have changed

z is the nearest ancestor of p that became unbalanced after the insertion of p

It must be that the height of y increased by one due to the insertion and that it is now 2 greater than its sibling

Since y remains balanced, its subtrees formerly had equal heights, and the subtree containing x has increased its height by one

That subtree increased either because $x = p$, and thus its height changed from 0 to 1, or because x previously had equal-height subtrees and the height of the one containing p has increased by 1

After the trinode restructuring, each of x, y, and z is balanced, and the root of the subtree has height $h + 2$, same as the height of z before the insertion

AVL Removal

Deletion from an AVL tree is as in a binary search tree: resulting in the removal of a node having either zero or one internal children

This change may violate the height-balance property in an AVL tree

Let position p represent a (possibly external) child of the removed node in tree T

We rebalance the tree by finding the only unbalanced ancestor z of p with the *trinode restructuring* method

This restructuring restores the height-balance property *locally*

We continue walking up T looking for unbalanced positions and performing the restructuring on those all the way to the root

Since the height of T is $O(\log n)$, then $O(\log n)$ trinode restructurings are sufficient to restore the height-balance property

AVL Removal Restructuring Details

As with insertion, we use trinode restructuring to restore balance in the tree T

Let z be the first unbalanced position encountered going up from p toward the root of T

Let y be that child of z with greater height (y will not be an ancestor of p)

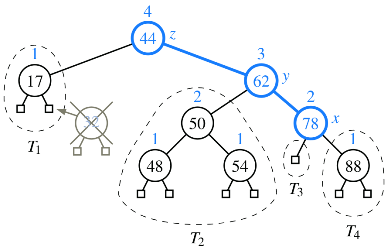
Let x be the child of y defined as follows:

- if one of the children of y is taller than the other, let x be the taller child of y
- otherwise, let x be the child of y on the same side as y

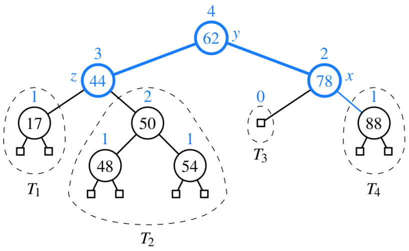
We then perform a `restructure(`x`)` operation

AVL Removal Example

After deletion of 32: the root is unbalanced



(a)



(b)

One restructuring operation rebalances the tree (for the given example)

Sorted Map Implementation Using an AVL Tree

The height of an AVL tree with n entries is guaranteed to be $O(\log n)$

Method	Running Time
size, isEmpty	$O(1)$
get, put, remove	$O(\log n)$
firstEntry, lastEntry	$O(\log n)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$
subMap	$O(s + \log n)$
entrySet, keySet, values	$O(n)$

Summary

Reading

Section 11.2 Balanced Search Trees

Section 11.3 AVL Trees

Questions?