CMPT 280

Topic 13: AVL Trees

Mark G. Eramian

University of Saskatchewan

References

- Textbook, Chapter 13

AVL Trees

- AVL trees are ordered binary trees with a guarantee on how they are balanced.
- Basic insertion/deletion algorithms remain unchanged.
- Insertion/deletion algorithms are augmented to maintain the balance guarantee.
- This makes AVL trees another example of **searchable dispensers**.

AVL Property

AVL Property

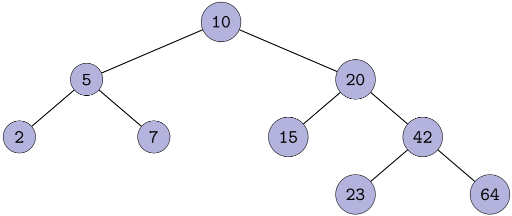
An AVL tree is an ordered binary tree in which the maximum imbalance is 1.

This property, if maintained, keeps the tree "almost" balanced.

Types of Imbalance

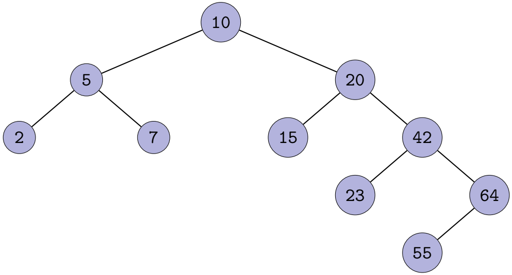
- If we start with an AVL tree, insertion of an element may cause an imbalance.
- Four possible imbalance situations: LL, LR, RL, RR.
- Critical node: node with an imbalance of 2, and all descendants have imbalance ≤ 1.

Exercise 1



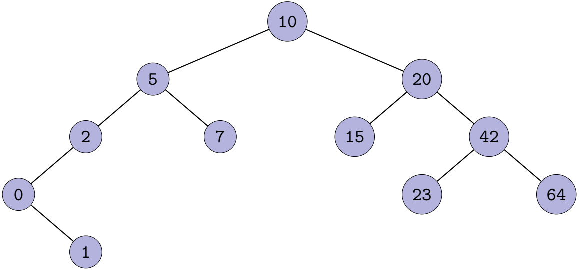
- Is it an AVL tree?
- If not, where is the critical node and what type of imbalance is exhibited?

Exercise 2



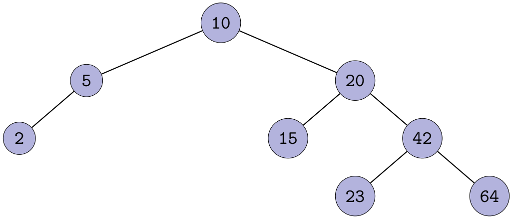
- Insert element 55 starting with tree in Ex. 1.
- Is it an AVL tree?
- If not, where is the critical node and what type of imbalance is exhibited?

Exercise 3



- Insert elements 0 (which doesn't break the AVL property) and then 1 starting with tree in Ex. 1.
- Is it an AVL tree?
- If not, where is the critical node and what type of imbalance is exhibited?

Exercise 4

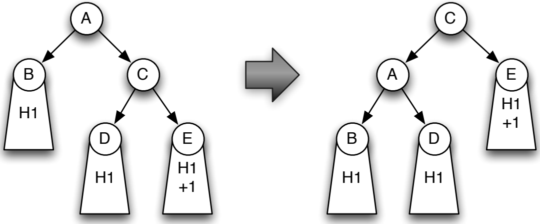


- Starting from the tree in Ex. 1, delete node containing 7.
- Is it an AVL tree?
- If not, where is the critical node and what type of imbalance is exhibited?

Fixing Imbalance After Insertion/Deletion

- We can repair the imbalance at critical nodes caused by insertion or deletion of a single element to/from an AVL tree.
- It can be done using efficient local adjustments to the trees in the vicinity of the critical node.

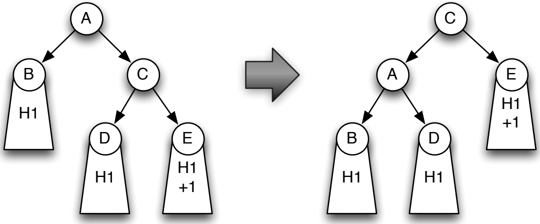
Left Rotation (aka: RR rotation)



A *left rotation* fixes RR imbalance. A is the critical node. After the rotation:

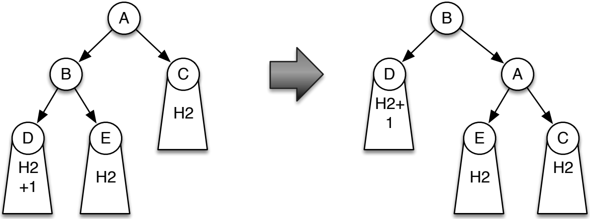
- All nodes in subtrees rooted at B, D, and E still have the AVL property; they did before, and we did not change them.
- A has the AVL property (both child subtrees have height H_1).
- C has the AVL property (both child subtrees have $H_1 + 1$).

Left Rotation



Keeping in mind that A might be a subtree of an even larger tree, if implementing this rotation, which nodes have references that need to be changed?

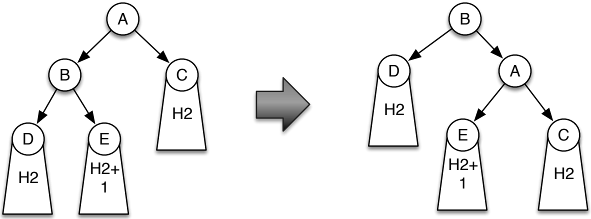
Right Rotation



A *right rotation* fixes LL imbalance. A is a critical node. After the rotation:

- All nodes in subtrees rooted at C, D, and E still have the AVL property; they did before, and we did not change them.
- A has the AVL property (both child subtrees have height H_2).
- B has the AVL property (both child subtrees have $H_2 + 1$).

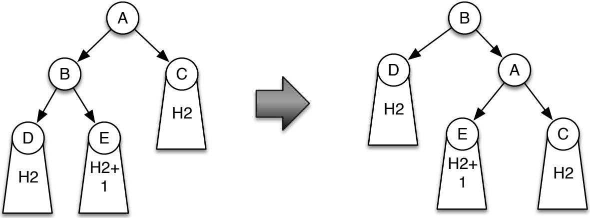
Fixing LR Imbalance



A *right rotation* fails to repair LR imbalance! After the rotation:

- All nodes in subtrees rooted at C, D, and E still have the AVL property; they did before, and we did not change them.
- A has the AVL property (child subtrees have height $H_2 + 1$ and H_2).
- B does not have the AVL property (left subtree has height H_2, right subtree has height $H_2 + 2$).

Fixing LR Imbalance

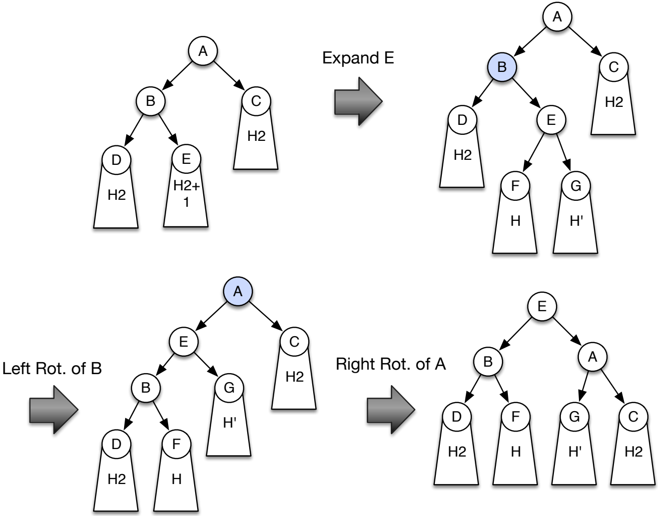


- The problem: B was right heavy before the rotation.
- The solution: make subtree rooted at B left heavy (but still an AVL tree) because we know that doesn't cause problems with right rotations.
- How?

Double Right Rotation

A *double right rotation* (or LR rotation) fixes LR imbalance. It consists of a left rotation of the left subtree of the critical node, followed by a right rotation of the critical node.

Double Right Rotation

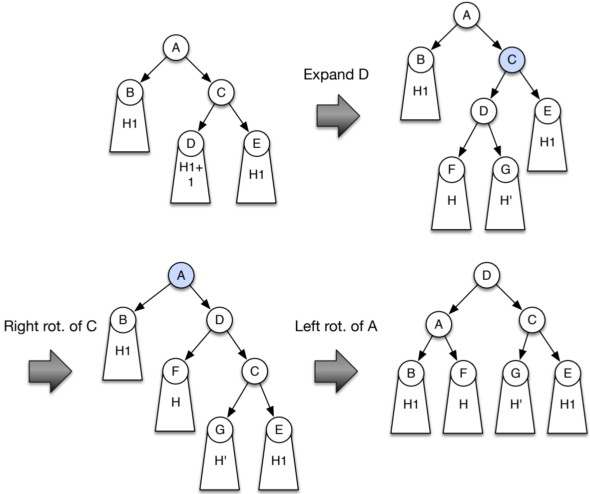


Note: $\max\{H, H'\} = H_2$, $|H - H'| \leq 1$

Double Left Rotation

A *double left rotation* fixes RL imbalance. It consists of a right rotation of the right subtree of the critical node, followed by a left rotation of the critical node.

Double Left Rotation



Note: $\max\{H, H'\} = H_2$, $|H - H'| \leq 1$

Computing the Height of a Tree

```
Algorithm height(R)
R is the root of a binary tree

if R == null
     return 0
else
     leftHeight = height(R.left);
     rightHeight = height(R.right);
     return 1 + max(leftHeight, rightHeight);
```

Time Complexity?

Algorithm: Restore AVL Property of Critical Node

```
Algorithm signed_imbalance(N)
N is a node.

  return height(N.left) - height(N.right)

Algoirthm restoreAVLProperty(R)
R is a (possibly critical) node

  imbalanceR = signed_imbalance(R);
  if abs(imbalanceR) <= 1 return;      // imbalance is 1 or 0
                                       // node is not critical

  if imbalanceR == 2                   // R is left heavy
    if signed_imbalance(R.left) >= 0   // R.left is left heavy
       RightRotate(R)                  // LL imbalance!  Do right rotation.
    else
       LeftRotate(R.left)              // LR imbalance! Do double right rotation.
       RightRotate(R)
  else            // (imbalanceR == -2): R is right heavy
    if signed_imbalance(R.right) <= 0 // R.right is right heavy
       LeftRotate(R)                   // RR imbalance!  Do left rotation.
    else
       RightRotate(R.right)            // RL imbalance!  Do double-left rotation.
       LeftRotate(R)
}
```

Time Complexity?

Checks for null omitted for brevity.

AVL Tree: Insertion Algorithm

```
// Recursively insert data into the tree rooted at R
Algorithm insert(data, R)
data is the element to be inserted
R is the root of the tree in which to insert 'data'

  if data < R.item() and R.left is not null
     insert (data, R.left)
  else if data >= R.item() and R.right is not null
     insert (data, R.right)
  else {
     // do normal ordered binary tree
     // insertion of data as child of R
  }
  restoreAVLProperty(R)
```

AVL Tree: Insertion Algorithm Complexity

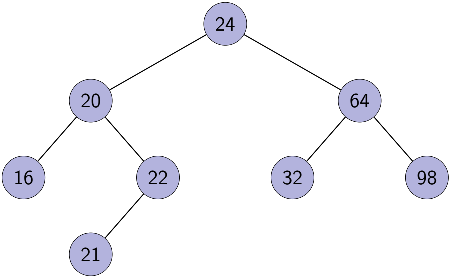
- The tree height is $\Theta(\log n)$. Thus, `restoreAVLProperty()` is called on at most $O(\log n)$ nodes.
- `restoreAVLProperty()` is $\Theta(\texttt{signed\_imbalance()}) + \Theta(1) = \Theta(\texttt{signed\_imbalance()})$
- `signed_imbalance()` is $2\Theta(\texttt{Height()}) = \Theta(\texttt{Height()})$
- The naive implementation of `Height()` (traversal of entire tree) is $\Theta(n)$, but it can be made $\Theta(1)$ if we store the heights of N.left and N.right in node N (and keep them updated by modifying insert & delete).
- Thus, insertion into an AVL tree is either $\Theta(n \log n)$ or $\Theta(\log n)$, depending on implementation.

Exercise 5

Starting with an empty AVL tree, insert the following values into the tree in the order given. Show the resulting tree after each step.

16, 32, 64, 24, 20, 22, 21, 98

If done correctly, you should have the following tree at the end.



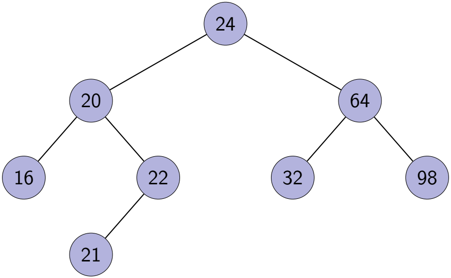
AVL Tree: Deletion

```
Algorithm delete(data, R)
data - element to be deleted
R - root of tree from which 'data' should be deleted

  if R is null
     // throw exception, data is not in tree.
  if data == R.item()
     if R has zero or one children delete R trivially
    else
     replace R.item() with the in-order successor item
          delete(in-order successor item, R.right)
  else
    if data <= R.item()
      delete(data, R.left);
    else
      delete(data, R.right);
  restoreAVLProperty(R);
}
```

Complexity of this deletion algorithm is also identical to that of the AVL tree insertion algorithm.

Exercise 6



Starting with the AVL tree, above, delete the following values from the tree in the order given.

$$64, 24, 20, 16, 21$$

Show the resulting tree after each step.

Next Class

- Next class reading: Chapter 14: Dictionaries