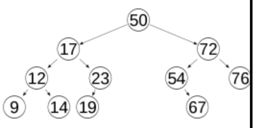
Data Structures & Algorithms 2

Topic 4 – Balanced Binary Trees

Balanced Trees

- We know from our study of Binary Search Trees (BST) that the average search and insertion time is O(log n)

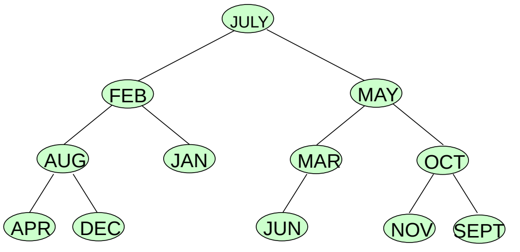


  - If there are n nodes in the binary tree it will take, on average, $\log_2 n$ comparisons/probes to find a particular node (or find out that it isn't there)

However...

- However, this is only true if the tree is 'balanced'
  - The tree usually ends up reasonably balanced when the elements are inserted in random order
  - But how can we guarantee that this will always be the case?

Balanced Trees

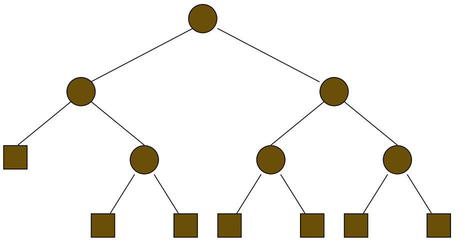


A Balanced Tree for the Months of the Year (alphabetically)

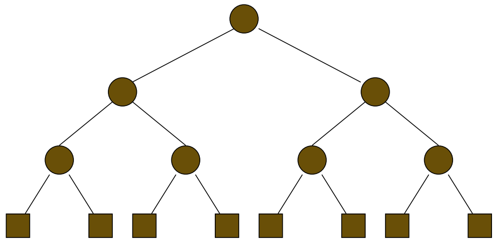
Binary Tree Terminology

- Complete Binary Trees (Fat Trees)
  - the external nodes appear on at most two adjacent levels
- Perfect Trees
  - complete trees having all their external nodes on one level
- Left-complete Trees
  - the internal nodes on the lowest level are in the leftmost possible position

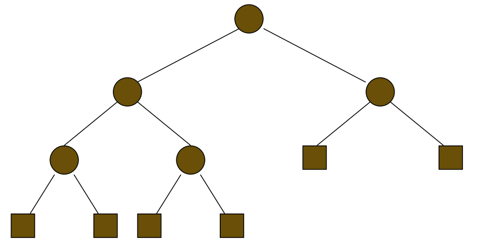
Complete Tree



Perfect Tree



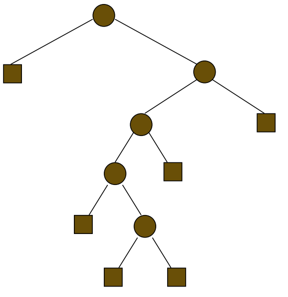
Left-Complete Tree



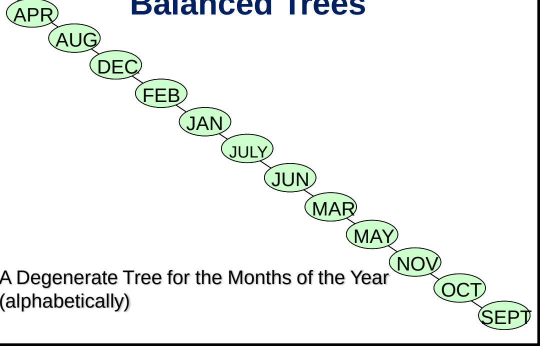
Skinny Trees

- However, if the elements are inserted in sorted order (e.g. lexicographic order) then the tree degenerates into a skinny tree
- In a skinny tree, every internal node has at most one internal child

Skinny Tree



Balanced Trees



A Degenerate Tree for the Months of the Year (alphabetically)

Balanced Trees

- If we are dealing with a dynamic tree nodes are being inserted and deleted over time
  - directory of files
  - index of university students
- We may need to restructure (balance) the tree so that we keep it fat

AVL Trees

- Adelson-Velskii and Landis in 1962 invented the first balanced binary tree algorithm
- Insertions (and deletions) are made such that the tree starts off and remains height-balanced
- The idea is that every node stays balanced with respect to the heights of its subtrees
- The height of one subtree is never allowed to exceed the height of the other by more than 1 level
- If an insertion causes a tree to become unbalanced, a rotation is carried out to fix it

Tree Height

- The height of T is defined recursively as

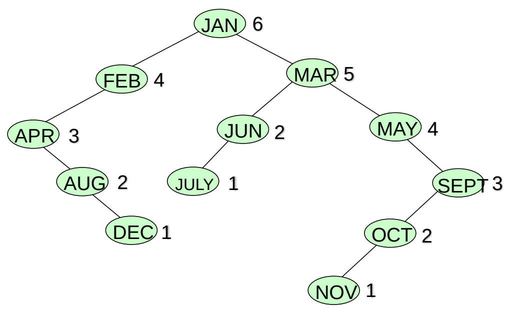
  0 if T is empty and

  1 + $max(height(T_1), height(T_2))$ otherwise, where T_1 and T_2 are the subtrees of the root.
- The height of a tree is the length of a longest chain of descendents

Tree Height

- The height of a node is the height of the subtree rooted at that node
- Height numbering
  - Number all bottom nodes 1
  - Number each internal node to be one more than the maximum of the numbers of its children
  - Then the number of the root is the height of T

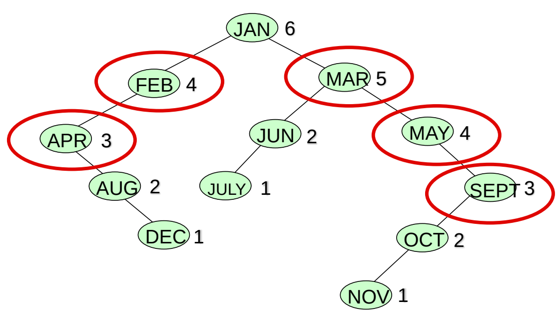
Tree Heights



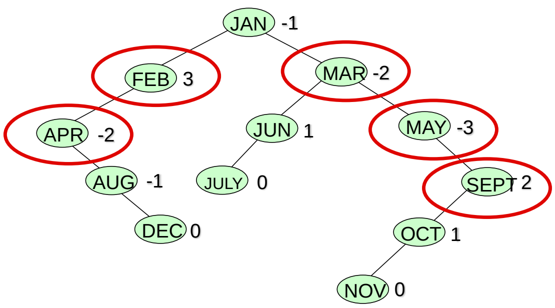
AVL height balancing

- An empty tree is always height-balanced
- If T is a non-empty binary tree with left and right sub-trees T_1 and T_2, then
- T is height-balanced iff (if and only if)
  - T_1 and T_2 are height-balanced, and
  - $|height(T_1) - height(T_2)| \leq 1$
- The balance factor is the difference between the heights of a node's left and right subtrees: $height(T_1) - height(T_2)$
- For balanced AVL trees, balance factors can only be -1,1 or 0

AVL height violations



Balance factors

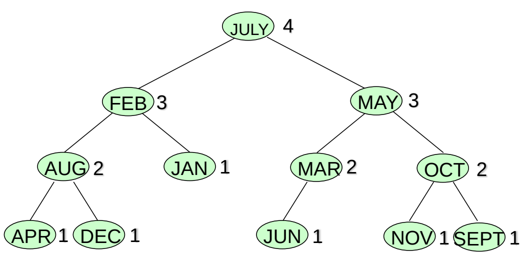


AVL algorithm

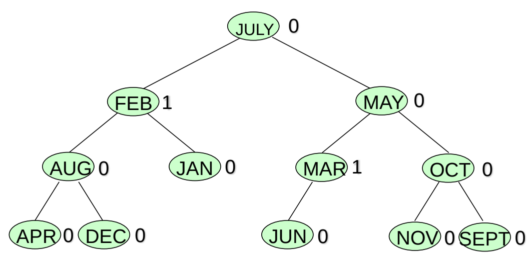
- Every time a new node is inserted check if all nodes are still height balanced
  - Check that the absolute difference in heights between a node's left and right subtrees is no greater than 1

- If any part of the tree is not height balanced, adjust the structure of the tree so that it becomes height balanced again

AVL Heights

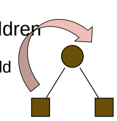


AVL Balance Factors



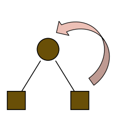
Rotations

- In order to keep the tree balanced following an insertion we use rotations
- A rotation is when a node is promoted to a higher level by twisting the tree at a particular point in either the left or right direction
- Rotations promote either the left or right children

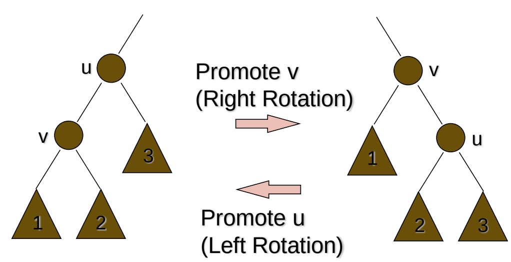


Right rotation promotes the left child

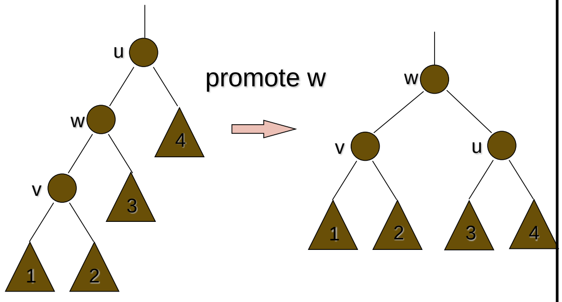
Left rotation promotes the right child



Rotations

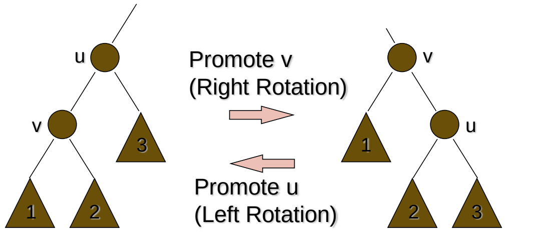


Rotation with child nodes



Binary search preserved

- We can carry out as many rotations as we want because rotations always preserve the binary search feature of the tree
  - Smaller values on the left
  - Bigger values on the right



keys(1) < key(v) < key(u)
key(v) < keys(2) < key(u)
key(u) < keys(3)

keys(1) < key(v)
key(v) < keys(2) < key(u)
key(v) < key(u) < keys(3)

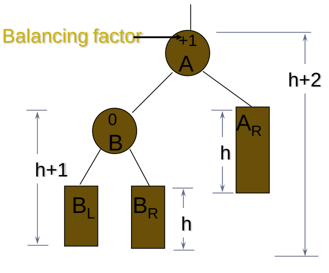
Balancing AVL Trees

- Let's refer to the inserted node as X
- Let's refer to the nearest ancestor having balance factor +2 or -2 as A

1. If X is inserted in the left subtree of the left subtree of A promote A's left child once *(LL rebalancing)*
2. If X is inserted in the right subtree of the left subtree of A promote A's right-left grandchild twice *(RL rebalancing)*
3. If X is inserted in the right subtree of the right subtree of A promote A's right child once *(RR rebalancing)*
4. If X is inserted in the left subtree of the right subtree of A promote A's left-right grandchild twice *(LR rebalancing)*

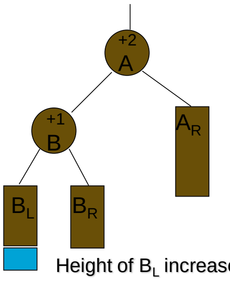
AVL Trees

Balanced Subtree



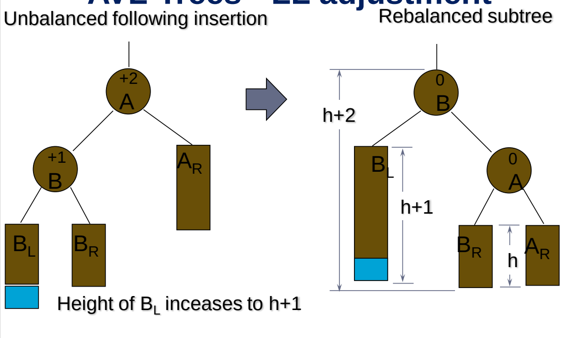
AVL Trees

Unbalanced following LL insertion

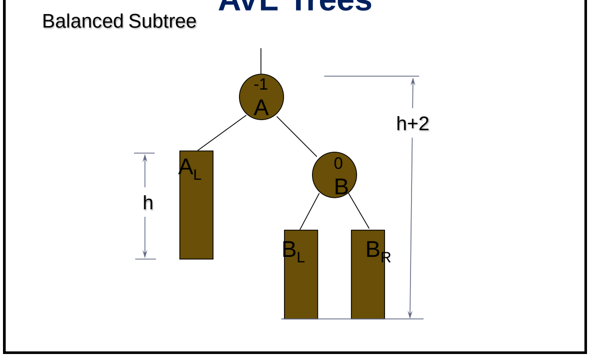


Height of B_L increases to h+1

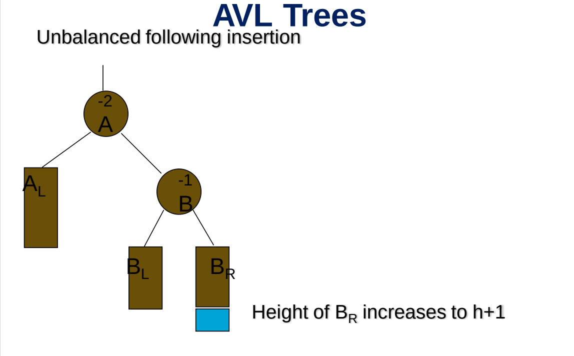
AVL Trees - LL adjustment



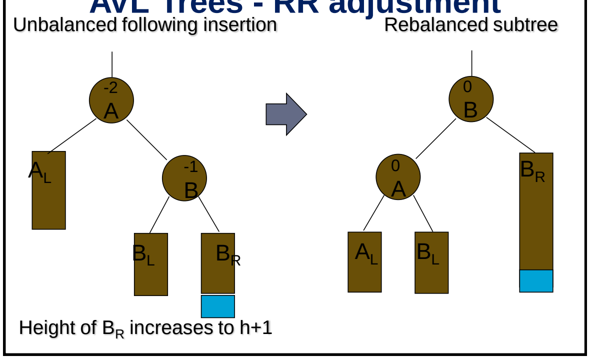
AVL Trees



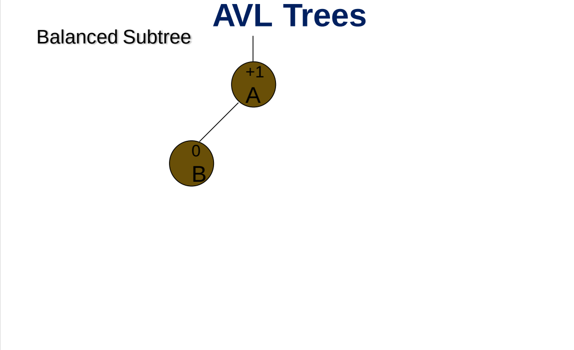
AVL Trees



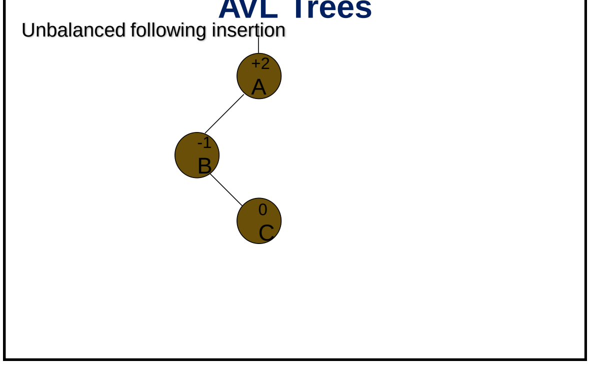
AVL Trees - RR adjustment



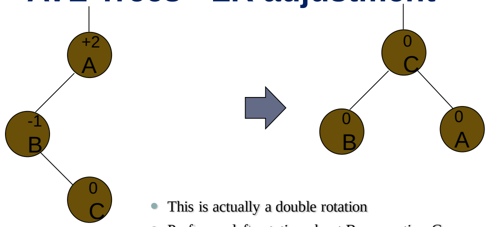
AVL Trees



AVL Trees

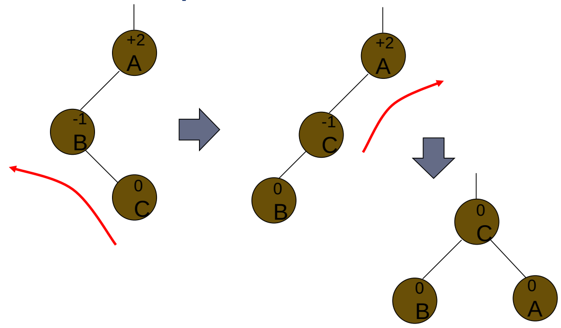


AVL Trees - LR adjustment



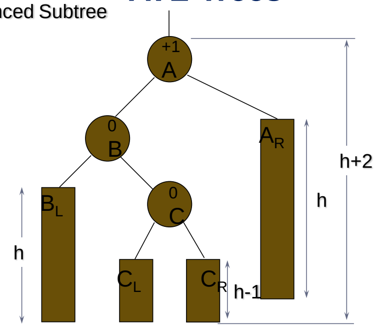
- This is actually a double rotation
- Perform a left rotation about B, promoting C
- Now perform a right rotation around A, promoting C again

LR and RL adjustments involve two promotions



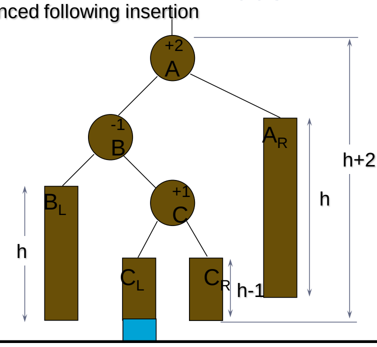
AVL Trees

Balanced Subtree

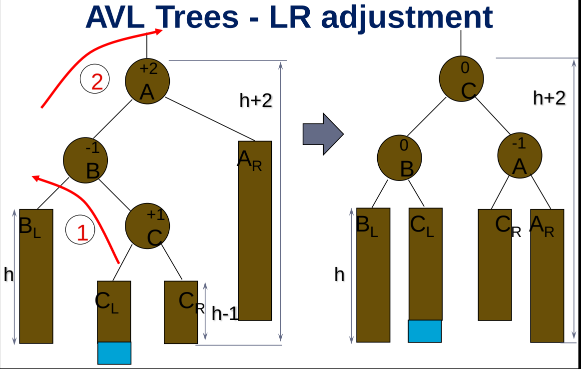


AVL Trees

Unbalanced following insertion

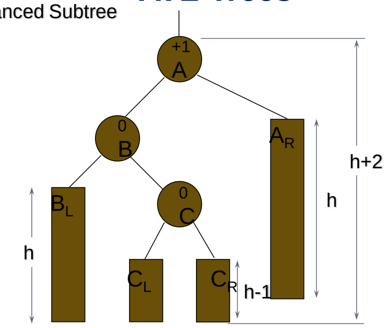


AVL Trees - LR adjustment



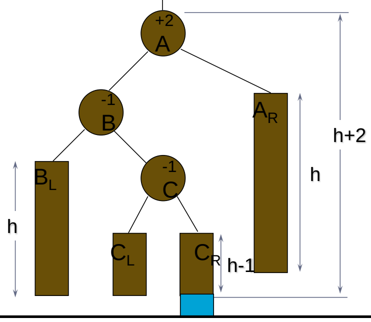
AVL Trees

Balanced Subtree

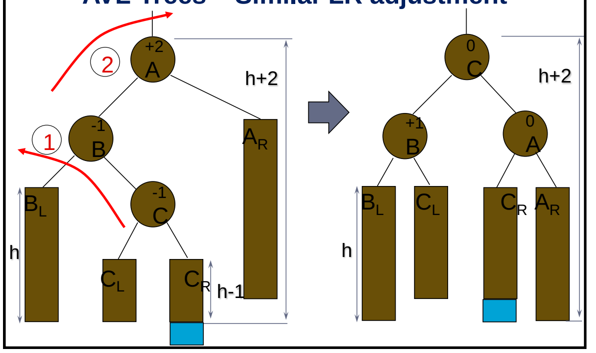


AVL Trees

Unbalanced following insertion

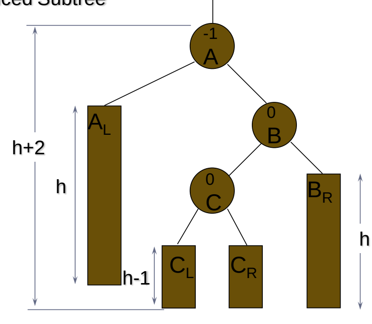


AVL Trees – Similar LR adjustment



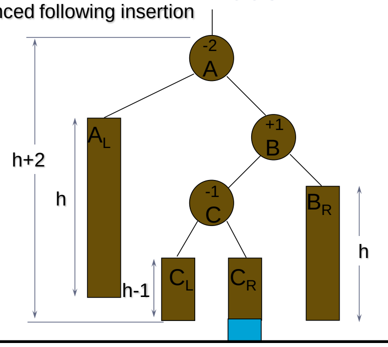
AVL Trees RL adjustment

Balanced Subtree

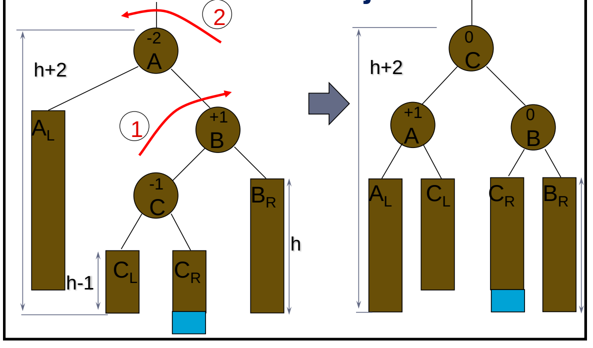


AVL Trees

Unbalanced following insertion

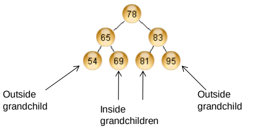


AVL Trees - RL adjustment



Rule of thumb

- Just remember, if your insertion is an inside grandchild descendant of the problem node, promote the inside grandchild twice
- If your insertion is an outside grandchild descendant of the problem node, promote the outside child once

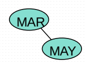


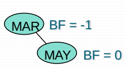
AVL Tree Algorithm

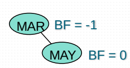
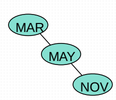
- When a node is inserted you check back up the path of insertion looking for nodes with a BF of +-2
- End the search when you reach a node with a BF of either 0 or a BF of +-2
- If you reach a node with a BF of 0, no adjustments are needed
  - None of the BFs above this will be affected because the height of that node is the same as before
- If you reach a node with a BF of +-2, rebalance that subtree
  - After rebalancing, BF at A (the problem node) will end up as 0
  - No further adjustment needed

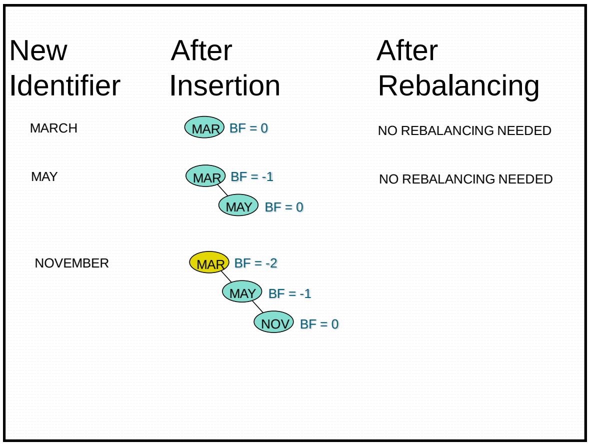
New Identifier	After Insertion	After Rebalancing
MARCH	MAR	

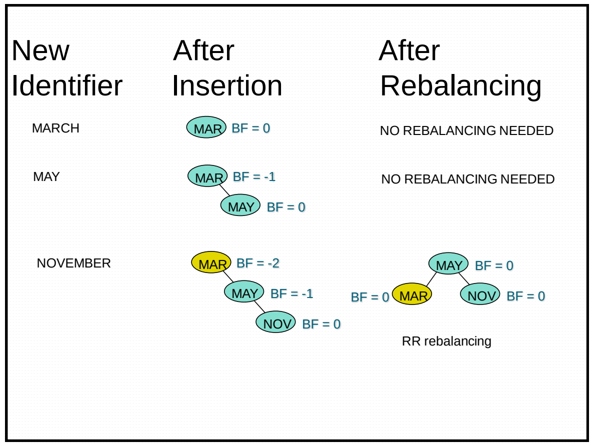
New Identifier	After Insertion	After Rebalancing
MARCH	MAR BF = 0	NO REBALANCING NEEDED

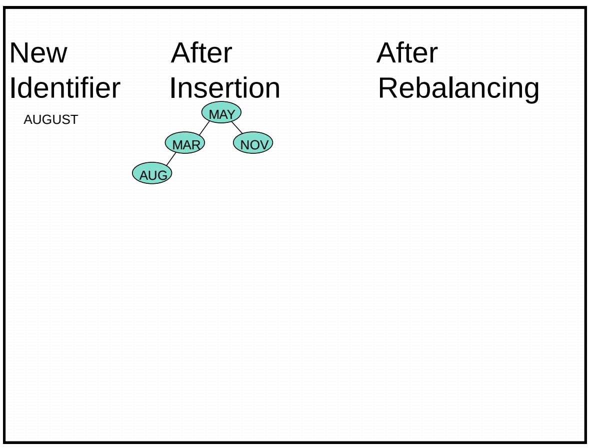
New Identifier	After Insertion	After Rebalancing
MARCH	MAR BF = 0	NO REBALANCING NEEDED
MAY		

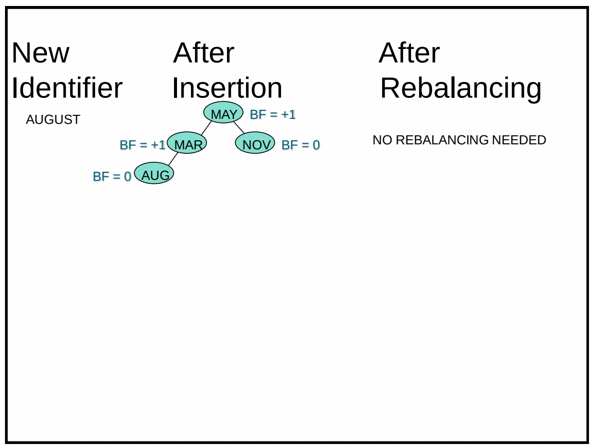
New Identifier	After Insertion	After Rebalancing
MARCH	MAR BF = 0	NO REBALANCING NEEDED
MAY		NO REBALANCING NEEDED

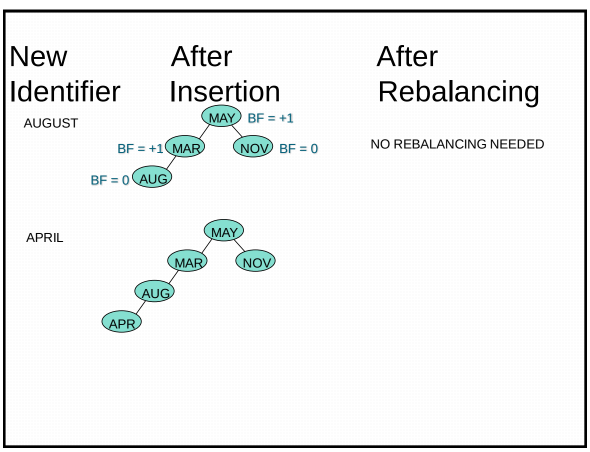
New Identifier	After Insertion	After Rebalancing
MARCH	MAR BF = 0	NO REBALANCING NEEDED
MAY		NO REBALANCING NEEDED
NOVEMBER		

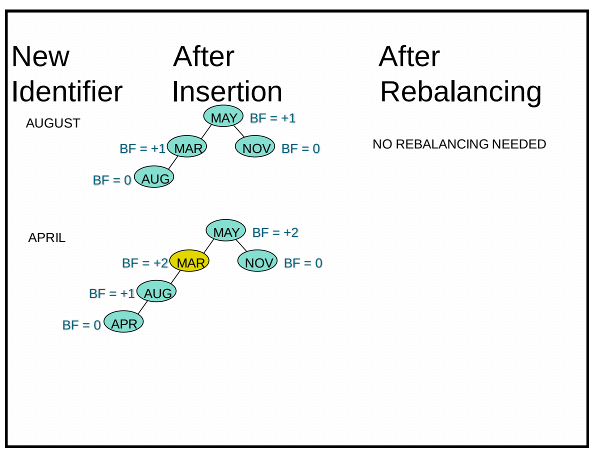

New Identifier
After Insertion
After Rebalancing
MARCH
MAR BF = 0
NO REBALANCING NEEDED
MAY
MAR BF = -1
MAY BF = 0
NO REBALANCING NEEDED
NOVEMBER
MAR BF = -2
MAY BF = -1
NOV BF = 0


New Identifier
After Insertion
After Rebalancing
MARCH
MAR BF = 0
NO REBALANCING NEEDED
MAY
MAR BF = -1
MAY BF = 0
NO REBALANCING NEEDED
NOVEMBER
MAR BF = -2
MAY BF = -1
NOV BF = 0
MAY BF = 0
BF = 0 MAR
NOV BF = 0
RR rebalancing


New Identifier
After Insertion
After Rebalancing
AUGUST
MAY
MAR
NOV
AUG

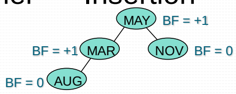

New Identifier
After Insertion
After Rebalancing
AUGUST
MAY BF = +1
BF = +1 MAR
NOV BF = 0
BF = 0 AUG
NO REBALANCING NEEDED


New Identifier
After Insertion
After Rebalancing
AUGUST
MAY BF = +1
BF = +1 MAR
NOV BF = 0
BF = 0 AUG
NO REBALANCING NEEDED
APRIL
MAY
MAR
NOV
AUG
APR


New Identifier
After Insertion
After Rebalancing
AUGUST
MAY BF = +1
BF = +1 MAR
NOV BF = 0
BF = 0 AUG
NO REBALANCING NEEDED
APRIL
MAY BF = +2
BF = +2 MAR
NOV BF = 0
BF = +1 AUG
BF = 0 APR

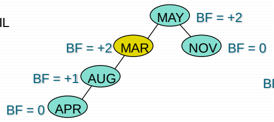
New Identifier | After Insertion | After Rebalancing

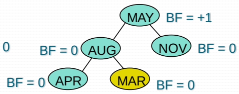
AUGUST



NO REBALANCING NEEDED

APRIL

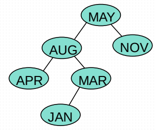




LL rebalancing

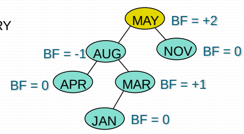
New Identifier | After Insertion | After Rebalancing

JANUARY



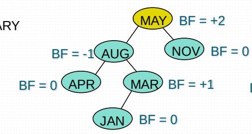
New Identifier | After Insertion | After Rebalancing

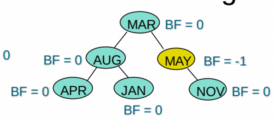
JANUARY



New Identifier | After Insertion | After Rebalancing

JANUARY

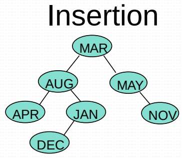




LR rebalancing

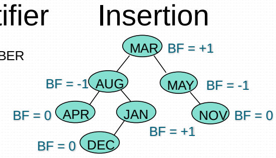
New Identifier | After Insertion | After Rebalancing

DECEMBER



New Identifier | After Insertion | After Rebalancing

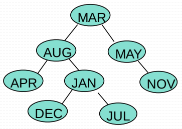
DECEMBER



NO REBALANCING NEEDED

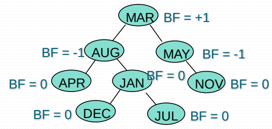
New Identifier | After Insertion | After Rebalancing

JULY



New Identifier | After Insertion | After Rebalancing

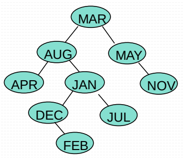
JULY



NO REBALANCING NEEDED

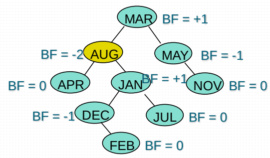
New Identifier | After Insertion | After Rebalancing

FEBRUARY



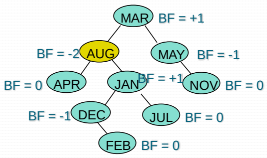
New Identifier | After Insertion | After Rebalancing

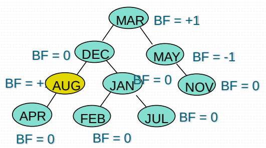
FEBRUARY



New Identifier | After Insertion | After Rebalancing

FEBRUARY

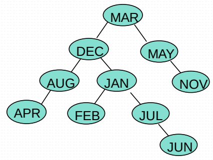




RL rebalancing

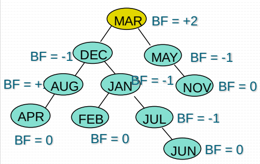
New Identifier | After Insertion | After Rebalancing

JUNE



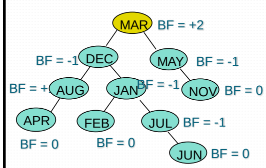
New Identifier | After Insertion | After Rebalancing

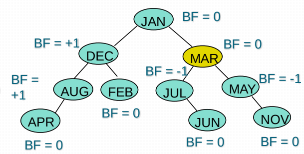
JUNE



New Identifier | After Insertion | After Rebalancing

JUNE

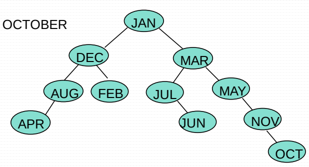




LR rebalancing

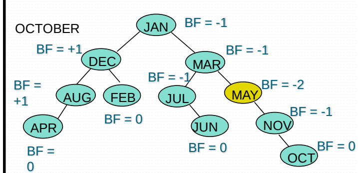
New Identifier | After Insertion | After Rebalancing

OCTOBER



New Identifier | After Insertion | After Rebalancing

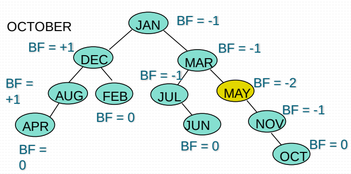
OCTOBER

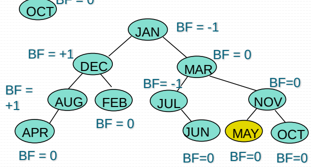


New Identifier | After Insertion | After Rebalancing

OCTOBER

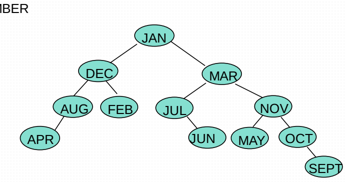
RR rebalancing





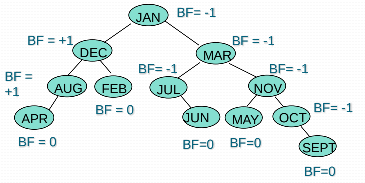
New Identifier | After Insertion | After Rebalancing

SEPTEMBER



New Identifier | After Insertion | After Rebalancing

SEPTEMBER | NO REBALANCING NEEDED



Red-Black Trees

- Red-black trees are less rigidly balanced than AVL
- Although they still guarantee a search time of O(logN), they are somewhat skinnier than AVL trees
  - max height 2*logN as opposed to 1.4*logN
- The original structure was invented in 1972 by Rudolph Bayer
- Keeping a red-black tree balanced requires only an average of one rotation per insertion and deletion
  - AVL requires O(logN) rotations per deletion
- A small disadvantage is that each node must store its colour

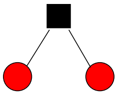
Red-Black Trees

- A red-black tree is a binary tree whose nodes can be coloured either red or black to satisfy the following conditions:
  - Black condition: Each root-to-leaf path contains exactly the same number of black nodes (black height)
  - Red condition: You can't have two red nodes together
  - The root is always black
  - Inserted nodes are always red

Red-Black Trees

Red-black tree (root must be black)

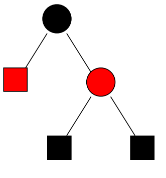
Red-Black Trees



Red-black tree

Root is black
Black height is maintained

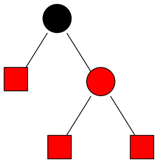
Red-Black Trees



Rule violation!

Black height not preserved

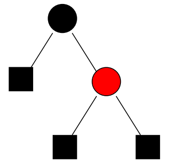
Red-Black Trees



Rule violation!

Two red nodes together

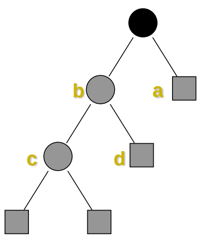
Red-Black Trees



Valid Red-Black Tree

Black height equal
Root is black
Red nodes have black children

Red-Black Trees



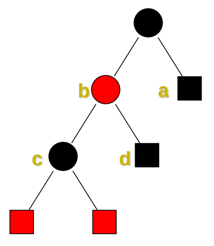
To satisfy black condition →

If b is black then black height cannot be preserved

Node b is red and node c and d are therefore black

a has to be black to preserve black height

Red-Black Trees

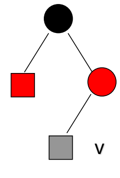


Solution

Red-Black Trees

- For all n >= 1, every red-black tree of size n has height $O(\log_2 n)$
- Thus, red-black trees provide a guaranteed worst-case search time of $O(\log_2 n)$
- However a best case path could involve all black nodes
- A worst case path could involve a red-black red-black path which would be at worst twice as long as the best case scenario
- Worst possible search time is $2 * \log_2 n$

Red-Black Trees



insertion at v

If new node is red, is the tree red-black?
If the new node is black, is the tree red-black?
v cannot be either, therefore we need to change the structure

Red-Black Trees

- Just as with ordinary trees, we perform the insertion by
  - first searching the tree until an external node is reached (if the key is not already in the tree)
  - then inserting the new (internal) node
- Insertions and deletions can cause red and black conditions to be violated
- Trees then have to be restructured
  - We can use rotations
  - We can also flip colours

red ←→ black

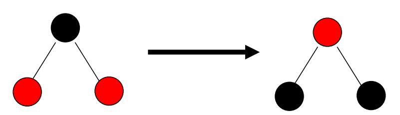
Terminology

- X is the node that has caused a rule violation
- P is the parent of X
- G is the grandparent of X (parent of P)

Complete Algorithm

- To insert a node you go left and right, searching for the place where the node should go
- On the way down the tree perform a colour flip whenever you find a black node with two red children
- This flip can cause a red-red conflict (the child node is denoted X)
- Conflict is fixed by a single or double rotation, depending on whether X is an outside or inside grandchild of G
- When you get to the insertion point insert your new node

Colour Flips all the way down

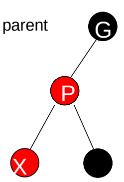


Make your way down the tree from the root

If you encounter a black node with two red children flip the colours

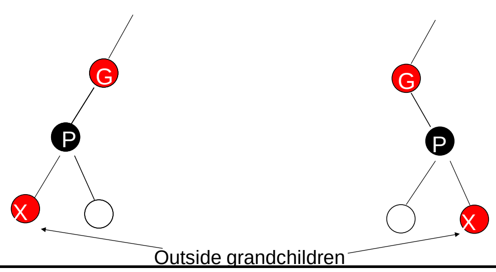
Problems

- A colour flip can't cause the black height rule to be violated
  - There are the same number of black nodes on any path
- The red rule might be violated because of a red-red conflict
  - One of the nodes you turned red already has a red parent
  - Let this node you turned red be called X
  - It's parent is P and grandparent is G
- Use rotations and colour flips to solve the problem

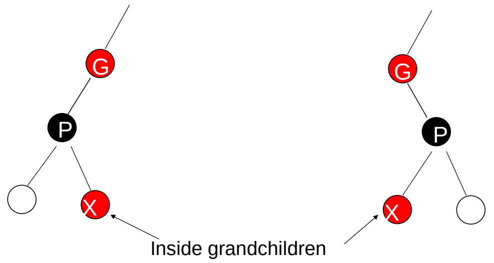


Outside Grandchildren

- We need some new terminology



Inside Grandchildren

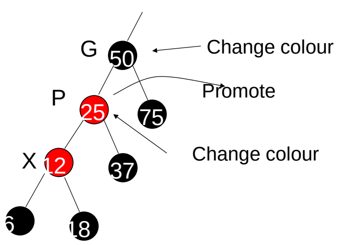


Eliminating red-red violations

- If X is an outside grandchild:
  - Switch the colour of X's grandparent G
  - Switch the colour of X's parent P
  - Promote the parent P by right rotating around the grandparent G
- If X is an inside grandchild:
  - Change the colour of the grandparent G
  - Change the colour of X
  - Promote X by rotating about P
  - Finally, promote X again by rotating about G

Outside Grandchild

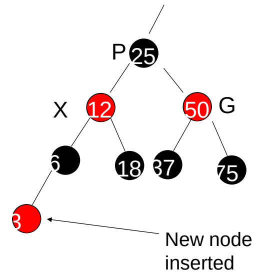
- Let X be the child in the red-red conflict



This tree has been created by inserting 25, 75, 12, 37, 6, 18

Now we want to insert 3

Solution

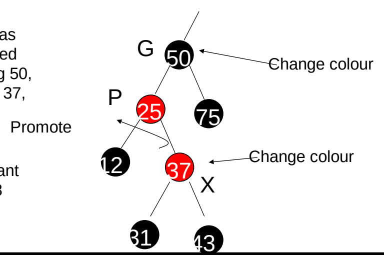


Inside Grandchild

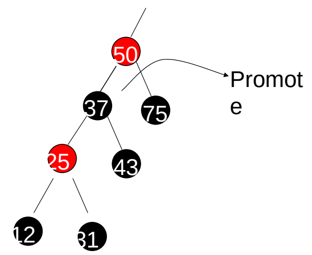
- Again, let X be the child in the red-red conflict

This tree has been created by inserting 50, 25, 75, 12, 37, 31, 43

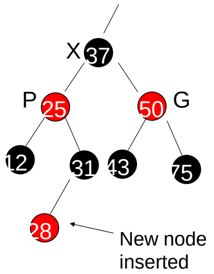
Now we want to insert 28



Second promotion



Insertion



Insertion

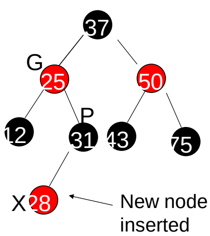
- When the colour flips all the way down have been completed we are ready to insert
- A newly inserted node is always red
- There are 3 possibilities:

1. P is black	(no problem)
2. P is red and X is an outside grandchild	(red-red violation)
3. P is red and X is an inside grandchild	(red-red violation)

- Solve the red-red violation in the same way

1. P is black

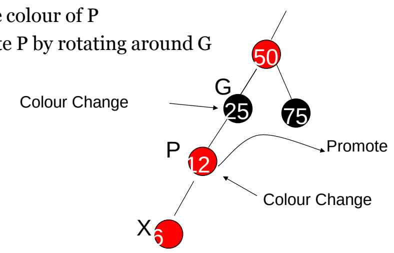
- The newly inserted node is red
- If P is black then there is no problem
- Just insert



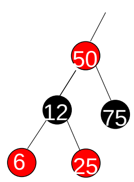
2. P is red, X is outside grandchild

- Change colour of G
- Change colour of P
- Promote P by rotating around G

This tree was created by inserting 50, 25, 75, 12 and finally 6



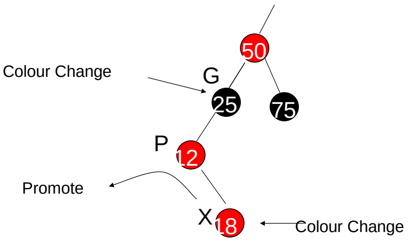
Final Tree with Insertion



3. P is red, X is inside grandchild

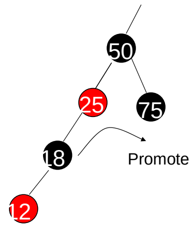
- 2 rotations, 2 colour changes
- Flip the colour of the grandparent G
- Flip the colour of X
- Promote X by rotating around P
- Promote X again by rotating around its new parent (its original grandparent G)

3. P is red, X is inside grandchild

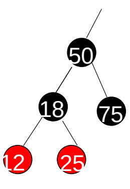


This tree was created by inserting 50, 25, 75, 12 and finally 18

Second Promotion



Finally

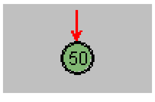


An example

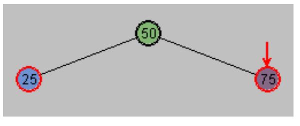
- As an example of a red black tree construction consider the following
- Insert the following integers into a red black tree structure:

50 25 75 12 37 31 28

Insert 50



Insert 25 and 75

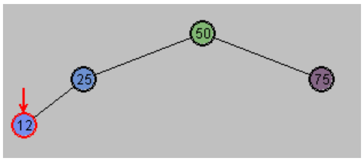


A black node with two red children…

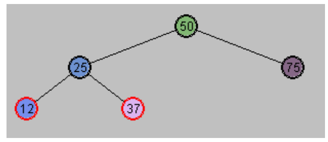
Insert 12 - First problem

As we go down we find a black node with two red children – flip their colours

Root always stays black



Insert 37

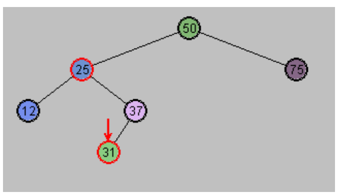


Again here is a situation where a black node has two red children

The colours will be flipped at the next insertion

Insert 31

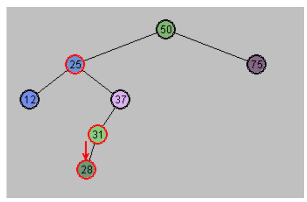
We flip the colours of 25, 12 and 37 on the way down and then insert 31



Insert 28

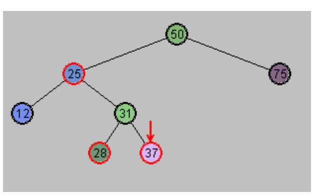
We now have our first double red situation

Change the colour of 31 and 37 and promote 31



Insert 37

Just keep following the steps!



Deletions

- Deletion for an ordinary binary tree is complicated enough - deletion for a red-black tree is even more complicated!
- One way to avoid the problem is to use a boolean value to mark a node as deleted without actually deleting it
- Disadvantage is that the tree starts to fill up with deleted nodes, increasing search times
- This can be acceptable if deletions are uncommon

Efficiency

- Like any binary search tree, red black trees allows for searching, insertion and deletion in O(logn) time
- Insertion and deletion have the same order because in order to insert or delete a node you have to find it first – the adjustment processes are also O(logn) at worst
- Insertion and deletion will only be a slightly slower O(logn) than with a binary search tree but having a balanced tree far outweighs this cost
- The red-black structure of the tree is irrelevant during searching
- The only memory penalty for having this structure is having to store a boolean with each node (is it red or black?)

Implementation

- Include an extra boolean variable in the node class to record node colour
- Adapt the insertion routine from the ordinary binary tree so that it checks on the way down to the insertion point if the current node is black with two red children
- If so, colour flip then check for red-red violations
- When you get to the insertion point, insert a red node and check for red-red violations
- Write methods for doing colour flips and for resolving red-red violations
- Write a method for promoting a node

Implementation

- Rules for removing red-red violation

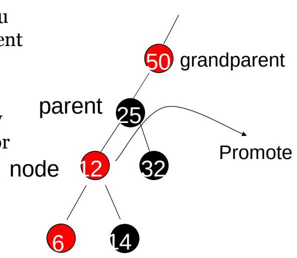
```
if inside grandchild {
 change colour of grandparent and child
 and promote child twice
}

if outside grandchild {
 change colour of parent and grandparent
 and promote parent once
}
```

Implementation

- If ever we come across a red-red violation then we need to know whether X is an inside or outside grandchild which means we need to know X's grandparent
- If we're doing a rotation about this grandparent then we need to know the great-grandparent
- A simple solution is just to track the last four nodes that you've come across on your path down the tree and the relationships between them
- This way you will always have access to child, parent, grandparent and great-grandparent

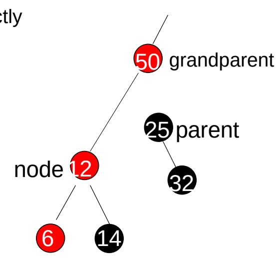
Step 1

- To promote a node you need the node, its parent and its grandparent
- You also need to know whether they are left or right descendants



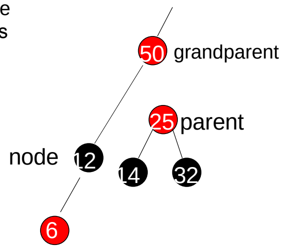
Step 2

- Connect the node directly to its grandparent
- It replaces its parent



Step 3

- The parent takes on the node's left/right child as its own left/right child



Step 4

- The parent can now be connected to the node as one of its children
- The node has been promoted