COL106: Data Structures and Algorithms

Ragesh Jaiswal, IIT Delhi

Data Structures: Balanced Binary Search Trees

- Consider the following implementation:

Code

```
class Node{
   public int key;
   public String value;
   public Node leftChild;
   public Node rightChild;
   public Node parent;
}
public class BST{
   public int size;
   public Node root;
   public BST(){
      size = 0;root = null;
   }
   public boolean isLeaf(Node N){//To be written}
   public String get(int k){//To be written}
   public void put(int k, String v){//To be written}
   public void remove(int k){//To be written}
}
```

- Tri-node restructuring for a node x, its parent y, and its grandparent z.

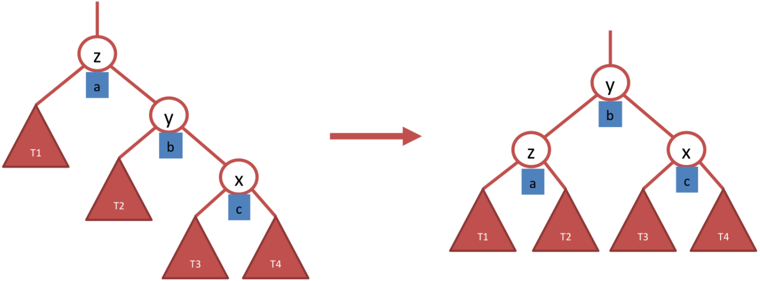


Figure : Case #1

- Tri-node restructuring for a node x, its parent y, and its grandparent z.

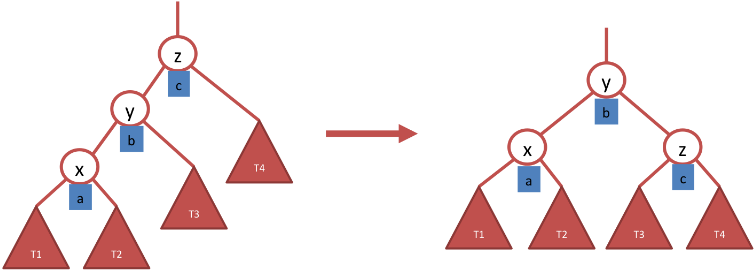


Figure : Case #2

- Tri-node restructuring for a node x, its parent y, and its grandparent z.

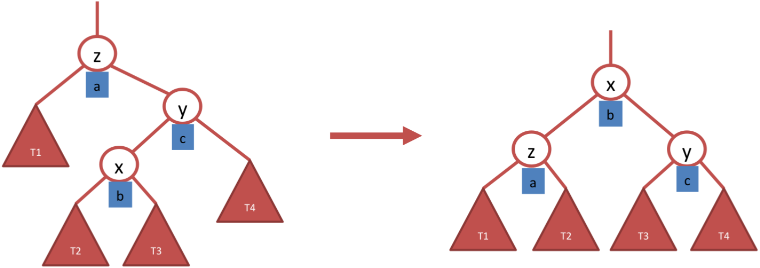


Figure : Case #3

- Tri-node restructuring for a node x, its parent y, and its grandparent z.

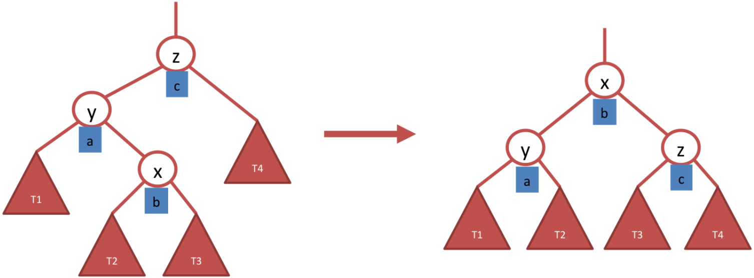


Figure : Case #4

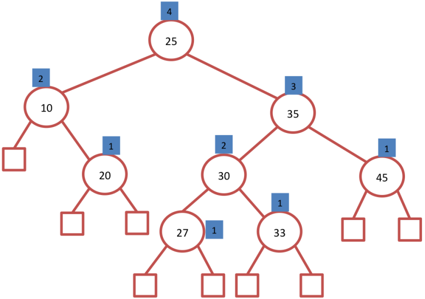
- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
  Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Claim: The height of any AVL tree storing n nodes is $O(\log n)$.

- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
  Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Question: How do we perform `get(k)` operation on an AVL tree?

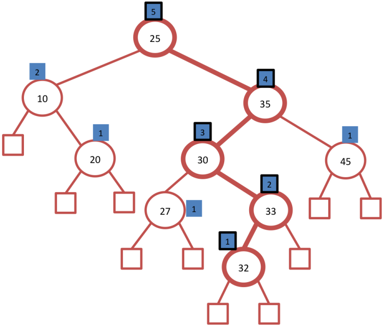
- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
  Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Question: How do we perform `get(k)` operation on an AVL tree?
  The same as BST

- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
  Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Question: How do we perform `put(k, v)` operation on an AVL tree?

- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
  Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Question: How do we perform `put(k, v)` operation on an AVL tree?
  - Same as in BST. However, you also have to make sure that after insertion, the height balance property is maintained.
  - Consider inserting an entry with key 32 in the Tree below.



- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
  Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Question: How do we perform `put(k, v)` operation on an AVL tree?
  - Same as in BST. However, you also have to make sure that after insertion, the height balance property is maintained.
  - Consider inserting an entry with key 32 in the Tree below.



- Question: How do we perform `put(k, v)` operation on an AVL tree?

Algorithm

//p denotes the node that is inserted.

`BalanceAfterPut(Node` p`)`

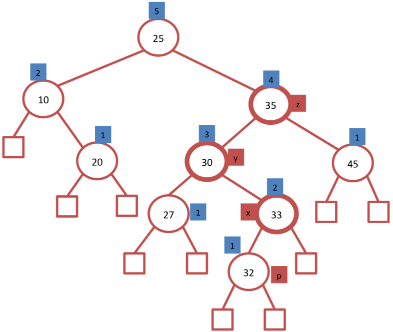
- While going up from p, let z denote the first node for which the height balance property is not satisfied.
- Let y be the child of z with greater height.
- Let x be the child of y with greater height.
- Perform a tri-node restructuring w.r.t. nodes x, y, z.

Algorithm

//p denotes the node that is inserted.

`BalanceAfterPut(Node` p`)`

- While going up from p, let z denote the first node for which the height balance property is not satisfied.
- Let y be the child of z with greater height.
- Let x be the child of y with greater height.
- Perform a tri-node restructuring w.r.t. nodes x, y, z.

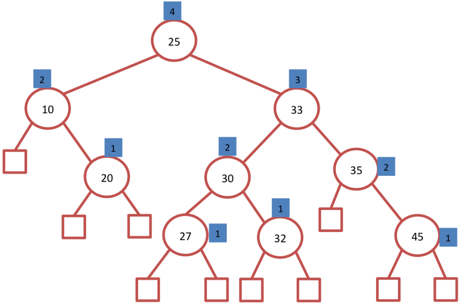


Algorithm

//p denotes the node that is inserted.

`BalanceAfterPut(Node` p`)`

- While going up from p, let z denote the first node for which the height balance property is not satisfied.
- Let y be the child of z with greater height.
- Let x be the child of y with greater height.
- Perform a tri-node restructuring w.r.t. nodes x, y, z.



Algorithm

//p denotes the node that is inserted.

`BalanceAfterPut(Node` p`)`

- While going up from p, let z denote the first node for which the height balance property is not satisfied.
- Let y be the child of z with greater height.
- Let x be the child of y with greater height.
- Perform a tri-node restructuring w.r.t. nodes x, y, z.

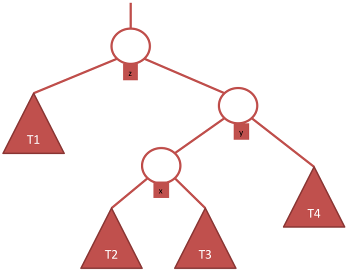


Figure : Suppose the insertion happens in the right sub-tree of node labeled x.

Algorithm

//p denotes the node that is inserted.

```
BalanceAfterPut(Node p)
```

- While going up from p, let z denote the first node for which the height balance property is not satisfied.
- Let y be the child of z with greater height.
- Let x be the child of y with greater height.
- Perform a tri-node restructuring w.r.t. nodes x, y, z.

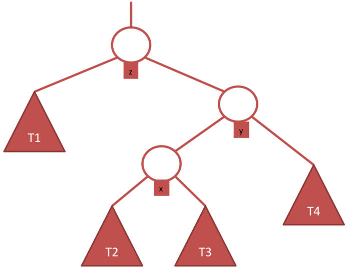


Figure : Suppose the insertion happens in $T3$ and x, y, z are as defined in the pseudocode.

Algorithm

//p denotes the node that is inserted.

`BalanceAfterPut(Node` p`)`

- While going up from p, let z denote the first node for which the height balance property is not satisfied.
- Let y be the child of z with greater height.
- Let x be the child of y with greater height.
- Perform a tri-node restructuring w.r.t. nodes x, y, z.

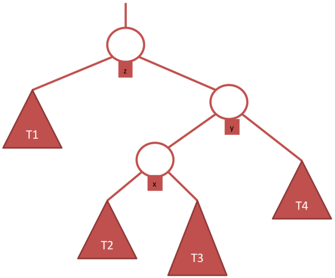


Figure : Suppose the insertion happens in $T3$ and x, y, z are as defined in the pseudocode.

Algorithm

//p denotes the node that is inserted.

`BalanceAfterPut(Node` p`)`

- While going up from p, let z denote the first node for which the height balance property is not satisfied.
- Let y be the child of z with greater height.
- Let x be the child of y with greater height.
- Perform a tri-node restructuring w.r.t. nodes x, y, z.

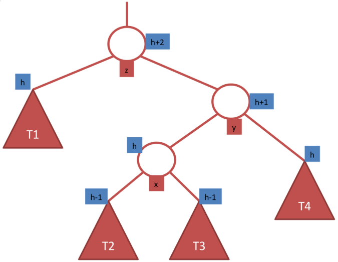


Figure : Suppose the insertion happens in $T3$ and x, y, z are as defined in the pseudocode. For some h the height of the nodes before insertion will be as shown above.

Algorithm

//p denotes the node that is inserted.

`BalanceAfterPut(Node p)`

- While going up from p, let z denote the first node for which the height balance property is not satisfied.
- Let y be the child of z with greater height.
- Let x be the child of y with greater height.
- Perform a tri-node restructuring w.r.t. nodes x, y, z.

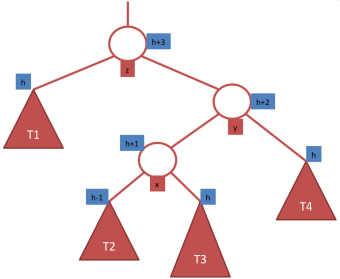


Figure : The height of the nodes after inserting the new node are as shown above.

Algorithm

//p denotes the node that is inserted.

`BalanceAfterPut(Node` p`)`

- While going up from p, let z denote the first node for which the height balance property is not satisfied.
- Let y be the child of z with greater height.
- Let x be the child of y with greater height.
- Perform a tri-node restructuring w.r.t. nodes x, y, z.

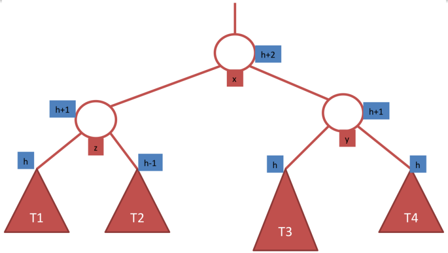
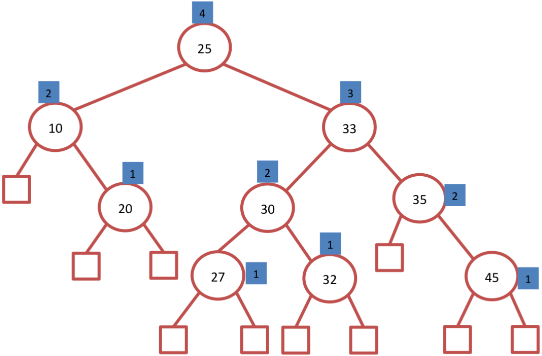


Figure : The height of the nodes after inserting and performing tri-node restructuring.

- Question: How do we perform `remove(k)` operation on an AVL tree?
  - Same as in BST. However, you also have to make sure that after deletion, the height balance property is maintained.
  - Consider deleting the entry with key 20 in the Tree below.



- Question: How do we perform `remove(k)` operation on an AVL tree?
  - Same as in BST. However, you also have to make sure that after deletion, the height balance property is maintained.
  - Consider deleting the entry with key 20 in the Tree below.

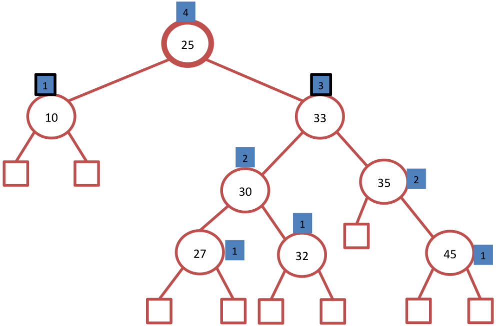


Figure : The tree needs to be balanced.

- Question: How do we perform `remove(k)` operation on an AVL tree?

Algorithm Sketch

//Initially p denotes the parent of the removed node

`BalanceAfterRemove(Node` p`)`

- Let z be the first unbalanced node going up from p
- If no such z exists then return
- Let y be the child of z of greater height.
- Let x be the child of y defined as follows:
  If one child of y is taller than the other then x is the taller child, otherwise x is the child of y with the same side as y is of z.
- Perform Tri-node restructuring w.r.t. x, y, z
- Let b denote the tallest node (among the nodes involved in restructuring) after the restructuring.
- If b is not the root, then `BalanceAfterRemove(b.parent)`

Algorithm Sketch

//Initially p denotes the parent of the removed node

`BalanceAfterRemove(Node` p`)`

- Let z be the first unbalanced node going up from p
- If no such z exists then return
- Let y be the child of z of greater height.
- Let x be the child of y defined as follows:
  If one child of y is taller than the other then x is the taller child, otherwise x is the child of y with the same side as y is of z.
- Perform Tri-node restructuring w.r.t. x, y, z
- Let b denote the tallest node (among the nodes involved in restructuring) after the restructuring.
- If b is not the root, then `BalanceAfterRemove(b.parent)`

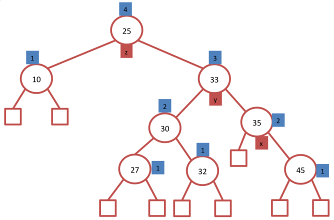


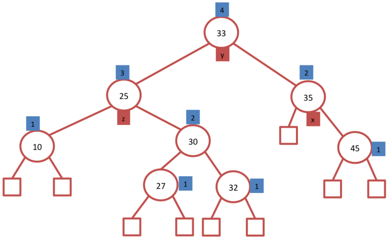
Figure : The tree needs to be balanced.

Algorithm Sketch

//Initially p denotes the parent of the removed node

`BalanceAfterRemove(Node p)`

- Let z be the first unbalanced node going up from p
- If no such z exists then return
- Let y be the child of z of greater height.
- Let x be the child of y defined as follows:
  If one child of y is taller than the other then x is the taller child, otherwise x is the child of y with the same side as y is of z.
- Perform Tri-node restructuring w.r.t. x, y, z
- Let b denote the tallest node (among the nodes involved in restructuring) after the restructuring.
- If b is not the root, then `BalanceAfterRemove(b.parent)`



Algorithm Sketch

//Initially p denotes the parent of the removed node

`BalanceAfterRemove(Node` p`)`

- Let z be the first unbalanced node going up from p
- If no such z exists then return
- Let y be the child of z of greater height.
- Let x be the child of y defined as follows:
  If one child of y is taller than the other then x is the taller child, otherwise x is the child of y with the same side as y is of z.
- Perform Tri-node restructuring w.r.t. x, y, z
- Let b denote the tallest node (among the nodes involved in restructuring) after the restructuring.
- If b is not the root, then `BalanceAfterRemove(b.parent)`

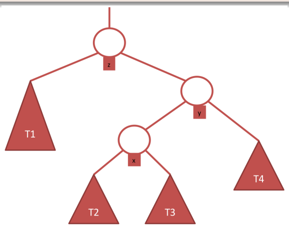


Figure : Suppose a node is deleted from $T1$.

Algorithm Sketch

//Initially p denotes the parent of the removed node

`BalanceAfterRemove(Node` p`)`

- Let z be the first unbalanced node going up from p
- If no such z exists then return
- Let y be the child of z of greater height.
- Let x be the child of y defined as follows:
  If one child of y is taller than the other then x is the taller child, otherwise x is the child of y with the same side as y is of z.
- Perform Tri-node restructuring w.r.t. x, y, z
- Let b denote the tallest node (among the nodes involved in restructuring) after the restructuring.
- If b is not the root, then `BalanceAfterRemove(b.parent)`

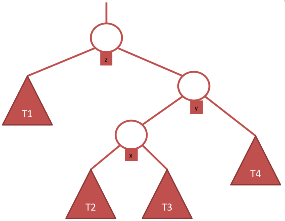


Figure : Suppose a node is deleted from $T1$.

Algorithm Sketch

//Initially p denotes the parent of the removed node

`BalanceAfterRemove(Node` p`)`

- Let z be the first unbalanced node going up from p
- If no such z exists then return
- Let y be the child of z of greater height.
- Let x be the child of y defined as follows:
  If one child of y is taller than the other then x is the taller child, otherwise x is the child of y with the same side as y is of z.
- Perform Tri-node restructuring w.r.t. x, y, z
- Let b denote the tallest node (among the nodes involved in restructuring) after the restructuring.
- If b is not the root, then `BalanceAfterRemove(b.parent)`

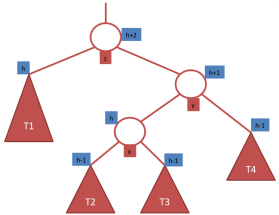


Figure : Suppose a node is deleted from $T1$. One possible scenario for heights before deletion.

Algorithm Sketch

//Initially p denotes the parent of the removed node
`BalanceAfterRemove(Node` p`)`
- Let z be the first unbalanced node going up from p
- If no such z exists then return
- Let y be the child of z of greater height.
- Let x be the child of y defined as follows:
  If one child of y is taller than the other then x is the taller child, otherwise x is the child of y with the same side as y is of z.
- Perform Tri-node restructuring w.r.t. x, y, z
- Let b denote the tallest node (among the nodes involved in restructuring) after the restructuring.
- If b is not the root, then `BalanceAfterRemove(b.parent)`

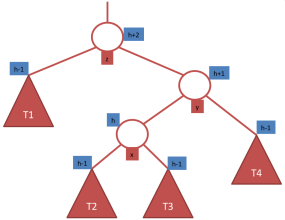


Figure : Suppose a node is deleted from $T1$. Heights after deletion.

Algorithm Sketch

//Initially p denotes the parent of the removed node

`BalanceAfterRemove(Node` p`)`

- Let z be the first unbalanced node going up from p
- If no such z exists then return
- Let y be the child of z of greater height.
- Let x be the child of y defined as follows:
  If one child of y is taller than the other then x is the taller child, otherwise x is the child of y with the same side as y is of z.
- Perform Tri-node restructuring w.r.t. x, y, z
- Let b denote the tallest node (among the nodes involved in restructuring) after the restructuring.
- If b is not the root, then `BalanceAfterRemove(b.parent)`

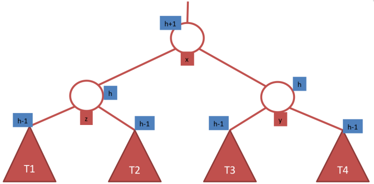


Figure : Suppose a node is deleted from $T1$. Heights after tri-node restructuring.

End