Formally Verified AVL Trees

Filip Carevic
Natalija Mitic
Radenko Pejic

Formal Verification (CS-550)
EPFL

Overview

- INTRODUCTION
- DEFINITIONS
- OPERATIONS:
  - SEARCH
  - BALANCING OPERATIONS
  - INSERT
  - DELETE
- RESULTS AND FUTURE WORK

Introduction

Definition AVL

- BST where the maximum length of a branch is not greater than $\frac{3}{2}\log_2(N+1)$, where N is the number of nodes.
  An Algorithm For The Organization Of Information - G. M. Adelson-Velskii, E. M. Landis, 1962

- BST where for every node the heights of the left and right subtree differ by not more than 1.
  Sorting and Searching Volume 3 of The art of computer programming - D. E. Knuth, 1997

	Average	Worst Case
search	O(log n)	O(log n)
insert	O(log n)	O(log n)
delete	O(log n)	O(log n)

Definitions

- **AVL = Balanced BST where the heights of right and left subtree differ by not more than 1**

```
isAVL = isBalanced && isBST && right.isAVL && left.isAVL || isEmpty
```

Definitions

- **AVL = Balanced BST where the heights of right and left subtree differ by not more than 1**

```
isAVL = isBalanced && isBST && right.isAVL && left.isAVL                    || isEmpty

isBST = checkGreatest(key, left) && checkSmallest(key, right) && left.isBST && right.isBST || isEmpty
```

Definitions

- **AVL = Balanced BST where the heights of right and left subtree differ by not more than 1**

```
isAVL = isBalanced && isBST && right.isAVL && left.isAVL                                  || isEmpty

isBST = checkGreatest(key, left) && checkSmallest(key, right) && left.isBST && right.isBST || isEmpty

isBalanced = (|left.height - right.height| <= 1) && left.isBalanced && right.isBalanced    || isEmpty
```

Operations

Search

- **regular BST lookup operation:**
  - recursively traverse the tree by utilising the binary search tree property

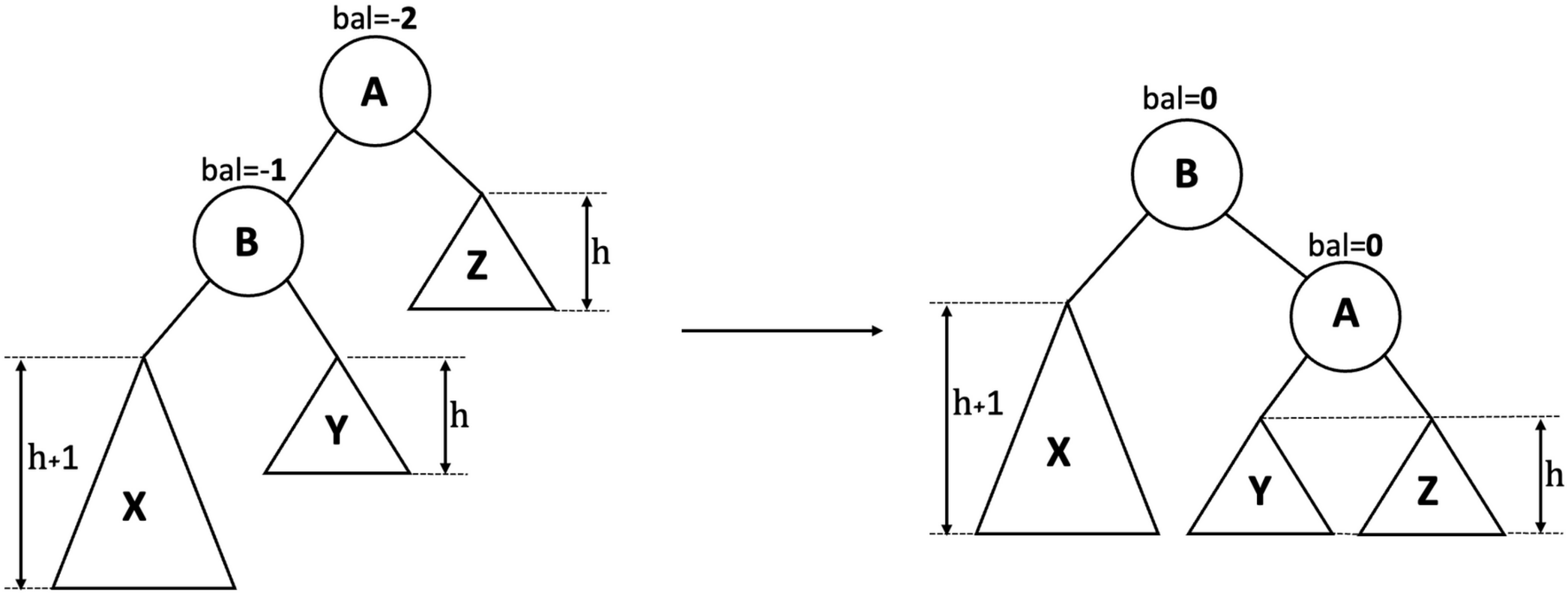
```
def lookupAVL(tree: Tree, searched: BigInt): Boolean = {
  require(tree.isAVL)
}.ensuring(isFound => isFound == tree.toList.contains(searched))
```

Balancing Operations

- **Right rotation:** imbalance due to the left child's left subtree
- **Left-right rotation:** imbalance due to the left child's right subtree
- **Left rotation:** imbalance due to the right child's right subtree
- **Right-left rotation:** imbalance due to the right child's left subtree

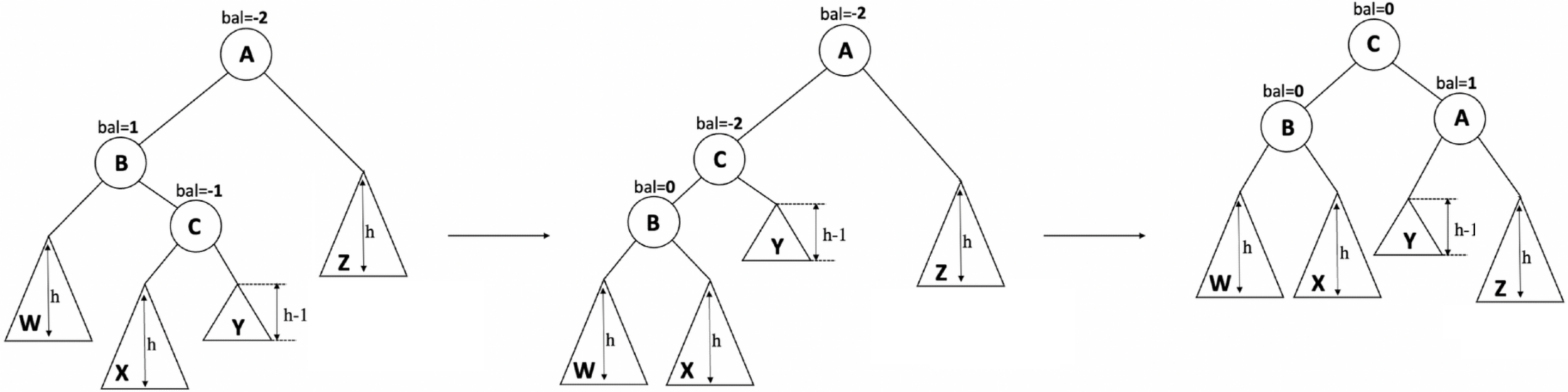
Balancing Operations

- **Right rotation:** imbalance due to the left child's left subtree



Balancing Operations

- **Left-right rotation:** imbalance due to the left child's right subtree



Balancing Operations - Verification

- **Keep Left Balanced = notBalanced ? (Right rotation || Left-right rotation) : unchanged**

```
def balanceLeft(value: BigInt, left: Tree, right: Tree): Tree = {
  require(
    left.isAVL && right.isAVL &&
    checkGreatest(value, left) && checkSmallest(value, right) &&
    ((abs(left.height - right.height) <= 1) || (left.height == right.height + 2))
  )
}.ensuring(tree =>
     tree.isAVL &&
     (tree.size == left.size + right.size + 1)  &&
     union(left.toSet, right.toSet, value) == tree.toSet &&
     (tree.height == max(left.height, right.height) + 1 || tree.height == max(left.height, right.height))
  )
```

Insert

- **regular BST insert operation with balancing operations:**
  - recursively traverse the tree by utilising BST properties
  - create a new node as a leaf node
  - execute balancing operations with a bottom-up approach
- **at most one rotation (single or double) is needed**

Insert

```
def insertAVL(tree: Tree, newKey: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
    newTree.isAVL &&
    ((newTree.size == tree.size + 1) || (newTree.size == tree.size)) &&
    abs(newTree.height - tree.height) <= 1
 )
```

Delete

- **regular BST delete operation with balancing operations:**
  - recursively traverse the tree until node to be deleted found
  - replace node with the largest value in its left subtree (=predecessor of the node)
  - execute balancing operations with a bottom-up approach

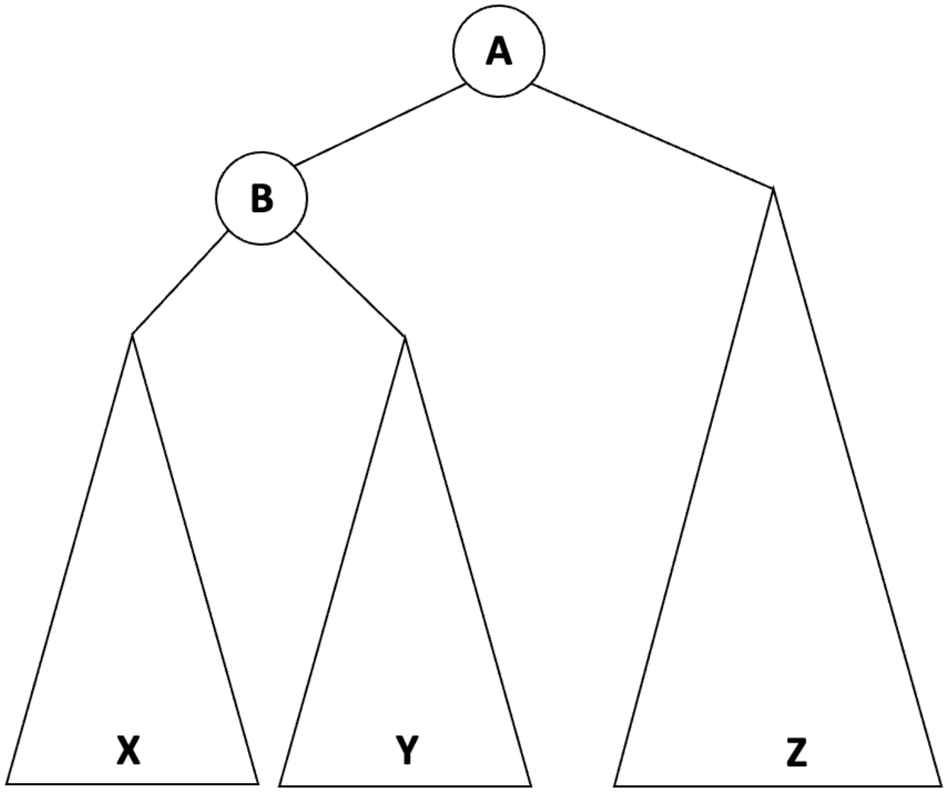
- **number of required balancing operations is O(log N)**

Delete

```
def deleteAVL(tree: Tree, key: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL &&
   !newTree.toList.contains(key) && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size || newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def replaceRootWithMax(tree: Tree): Tree = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def removeMax(tree: Tree): (BigInt, Tree) = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring((maxValue, leftTree) =>
   leftTree.isAVL && checkGreatest(maxValue, leftTree) &&
   (leftTree.size == tree.size - 1) &&
   (leftTree.height == tree.height || leftTree.height == tree.height - 1)  &&
   tree.toList.contains(maxValue) && leftTree.toList.subsetOf(tree.toList)
  )
```

Delete

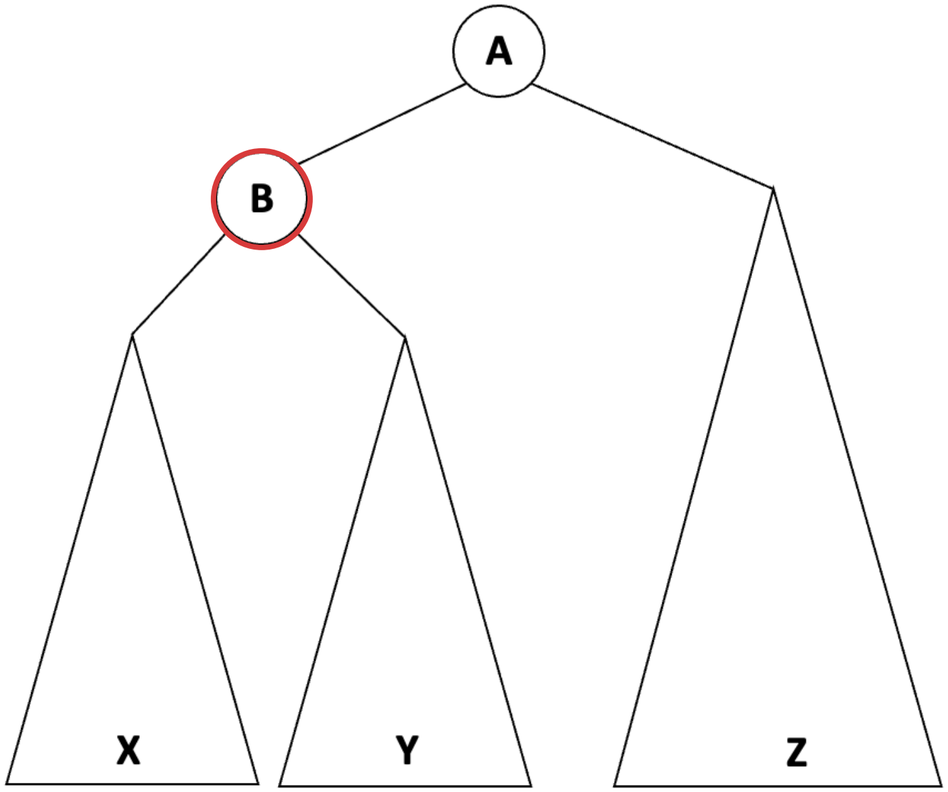
delete B:




```
def deleteAVL(tree: Tree, key: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL &&
   !newTree.toList.contains(key) && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size || newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def replaceRootWithMax(tree: Tree): Tree = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def removeMax(tree: Tree): (BigInt, Tree) = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring((maxValue, leftTree) =>
   leftTree.isAVL && checkGreatest(maxValue, leftTree) &&
   (leftTree.size == tree.size - 1) &&
   (leftTree.height == tree.height || leftTree.height == tree.height - 1)  &&
   tree.toList.contains(maxValue) && leftTree.toList.subsetOf(tree.toList)
  )
```

Delete

- **delete B:**
  - locate B

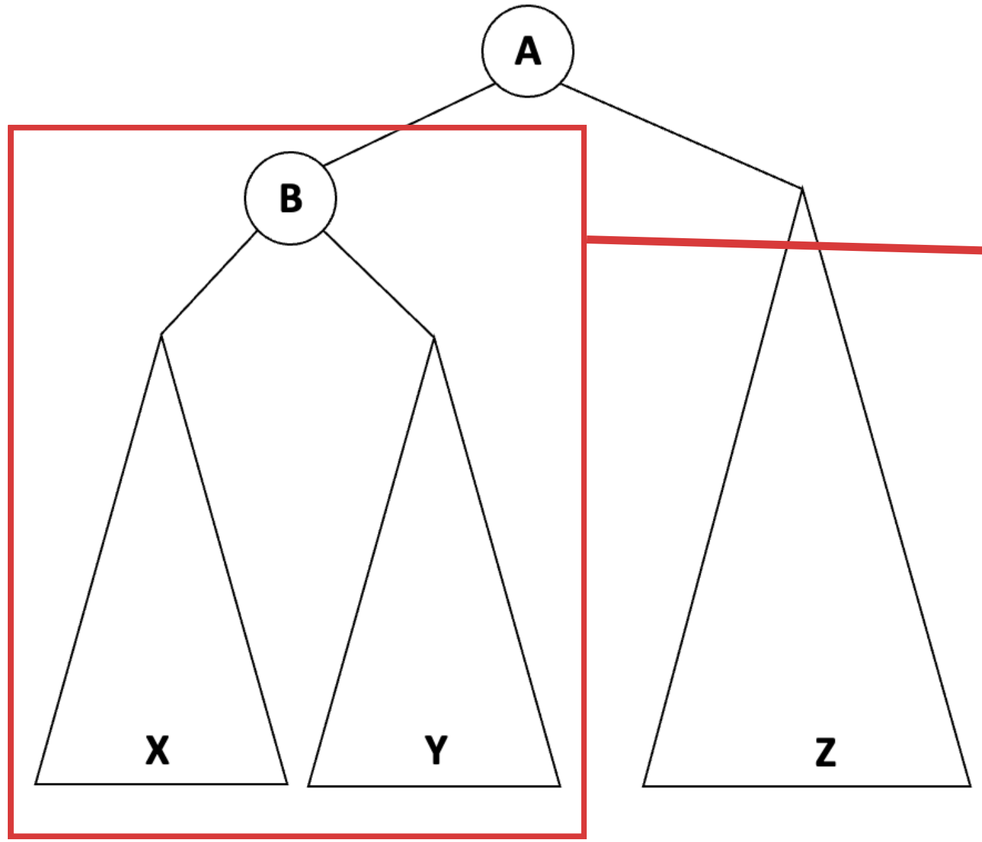



```
def deleteAVL(tree: Tree, key: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL &&
   !newTree.toList.contains(key) && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size || newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def replaceRootWithMax(tree: Tree): Tree = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def removeMax(tree: Tree): (BigInt, Tree) = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring((maxValue, leftTree) =>
   leftTree.isAVL && checkGreatest(maxValue, leftTree) &&
   (leftTree.size == tree.size - 1) &&
   (leftTree.height == tree.height || leftTree.height == tree.height - 1)  &&
   tree.toList.contains(maxValue) && leftTree.toList.subsetOf(tree.toList)
  )
```

Delete

delete B:

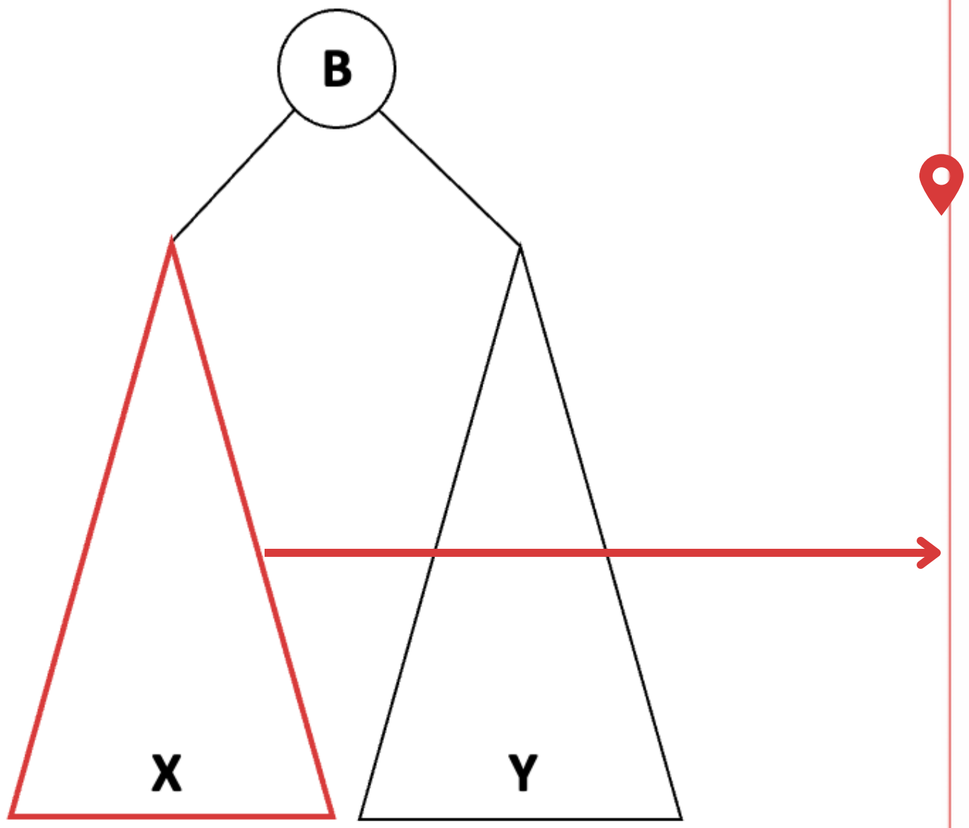
- locate B
- replace B with its predecessor




```
def deleteAVL(tree: Tree, key: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL &&
   !newTree.toList.contains(key) && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size || newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def replaceRootWithMax(tree: Tree): Tree = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def removeMax(tree: Tree): (BigInt, Tree) = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring((maxValue, leftTree) =>
   leftTree.isAVL && checkGreatest(maxValue, leftTree) &&
   (leftTree.size == tree.size - 1) &&
   (leftTree.height == tree.height || leftTree.height == tree.height - 1)  &&
   tree.toList.contains(maxValue) && leftTree.toList.subsetOf(tree.toList)
  )
```

Delete

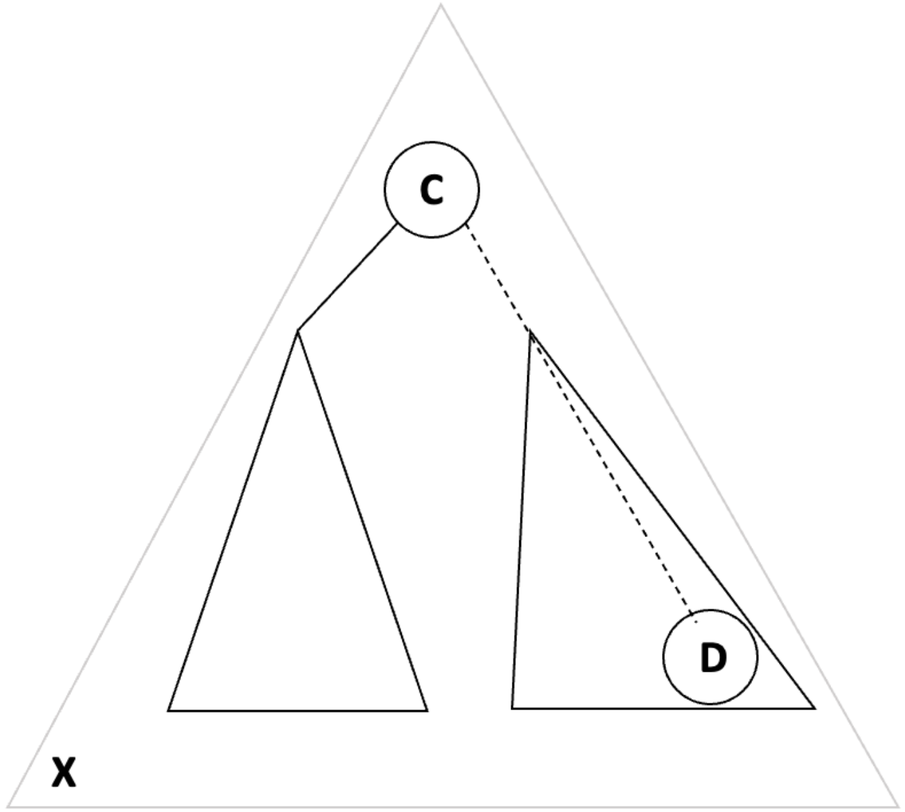
- **delete B:**
  - locate B
  - replace B with its predecessor
    - remove max value in left subtree




```
def deleteAVL(tree: Tree, key: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL &&
   !newTree.toList.contains(key) && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size || newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def replaceRootWithMax(tree: Tree): Tree = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def removeMax(tree: Tree): (BigInt, Tree) = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring((maxValue, leftTree) =>
   leftTree.isAVL && checkGreatest(maxValue, leftTree) &&
   (leftTree.size == tree.size - 1) &&
   (leftTree.height == tree.height || leftTree.height == tree.height - 1)  &&
   tree.toList.contains(maxValue) && leftTree.toList.subsetOf(tree.toList)
  )
```

Delete

- **delete B:**
  - locate B
  - replace B with its predecessor
    - remove max value in left subtree

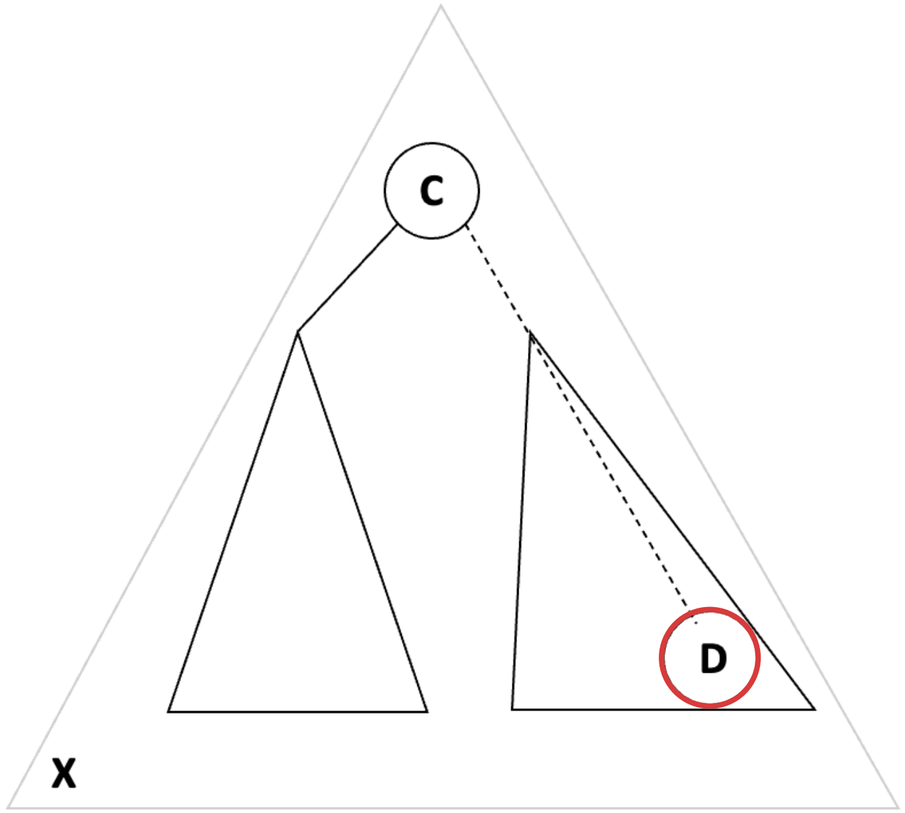



```
def deleteAVL(tree: Tree, key: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL &&
   !newTree.toList.contains(key) && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size || newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def replaceRootWithMax(tree: Tree): Tree = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def removeMax(tree: Tree): (BigInt, Tree) = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring((maxValue, leftTree) =>
   leftTree.isAVL && checkGreatest(maxValue, leftTree) &&
   (leftTree.size == tree.size - 1) &&
   (leftTree.height == tree.height || leftTree.height == tree.height - 1)  &&
   tree.toList.contains(maxValue) && leftTree.toList.subsetOf(tree.toList)
  )
```

Delete

delete B:

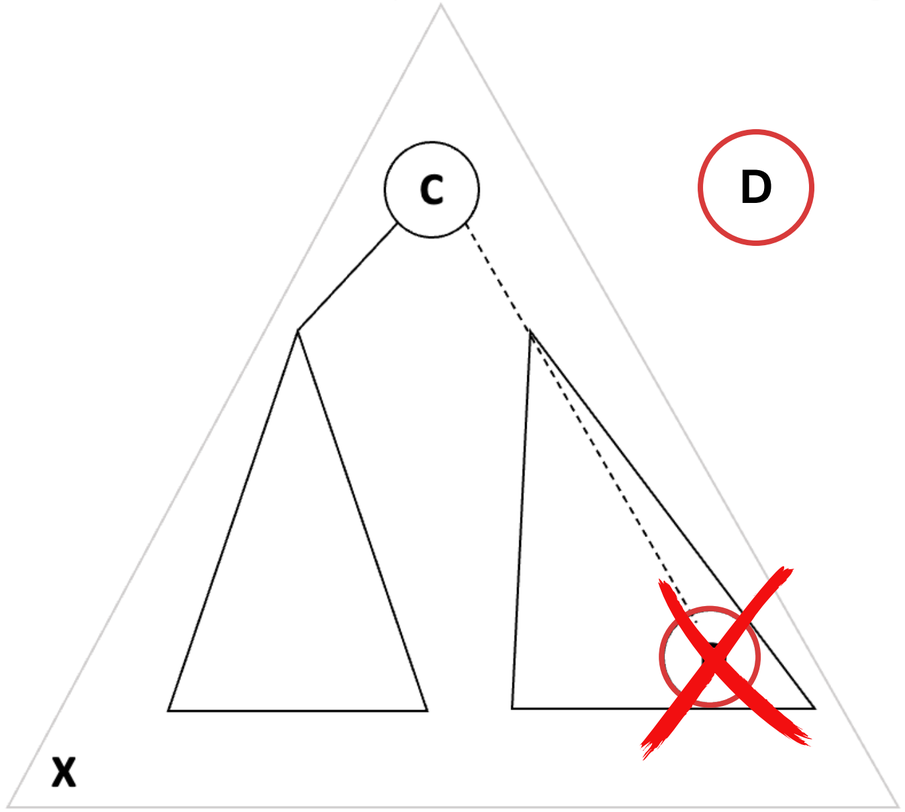
- locate B
- replace B with its predecessor
  - remove max value in left subtree




```
def deleteAVL(tree: Tree, key: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL &&
   !newTree.toList.contains(key) && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size || newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def replaceRootWithMax(tree: Tree): Tree = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def removeMax(tree: Tree): (BigInt, Tree) = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring((maxValue, leftTree) =>
   leftTree.isAVL && checkGreatest(maxValue, leftTree) &&
   (leftTree.size == tree.size - 1) &&
   (leftTree.height == tree.height || leftTree.height == tree.height - 1)  &&
   tree.toList.contains(maxValue) && leftTree.toList.subsetOf(tree.toList)
  )
```

Delete

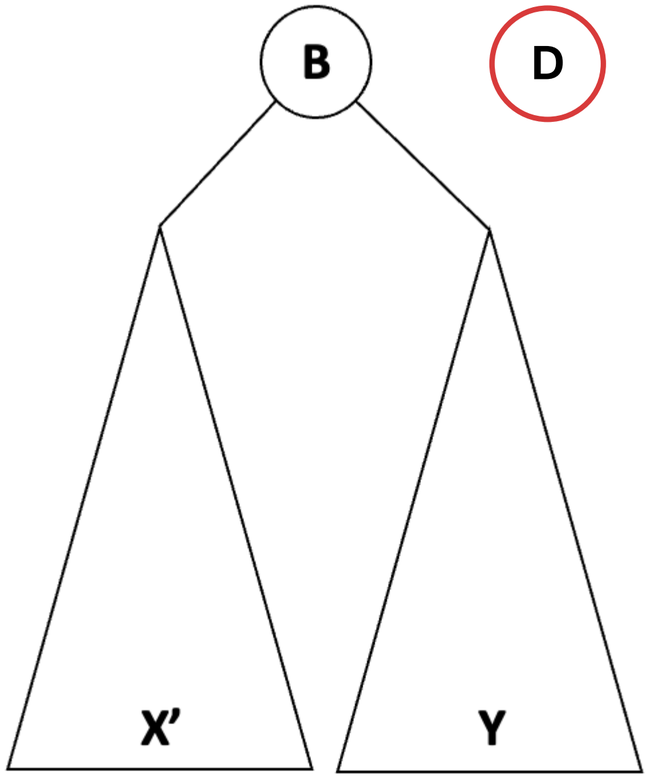
- **delete B:**
  - locate B
  - replace B with its predecessor
    - remove max value in left subtree
      - keep max and do balancing




```
def deleteAVL(tree: Tree, key: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL &&
   !newTree.toList.contains(key) && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size || newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def replaceRootWithMax(tree: Tree): Tree = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def removeMax(tree: Tree): (BigInt, Tree) = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring((maxValue, leftTree) =>
   leftTree.isAVL && checkGreatest(maxValue, leftTree) &&
   (leftTree.size == tree.size - 1) &&
   (leftTree.height == tree.height || leftTree.height == tree.height - 1)  &&
   tree.toList.contains(maxValue) && leftTree.toList.subsetOf(tree.toList)
  )
```

Delete

- **delete B:**
  - locate B
  - replace B with its predecessor
    - remove max value in left subtree
      - keep max and do balancing

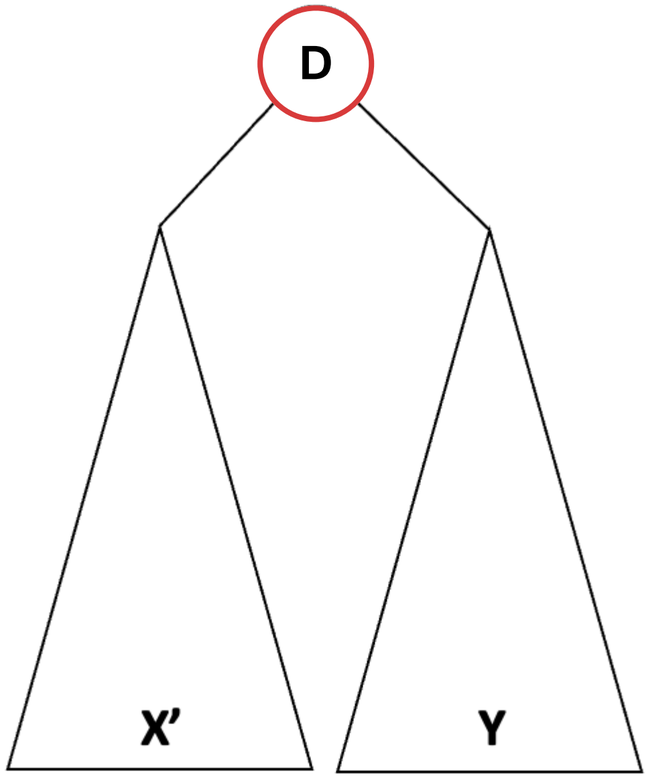



```
def deleteAVL(tree: Tree, key: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL &&
   !newTree.toList.contains(key) && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size || newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def replaceRootWithMax(tree: Tree): Tree = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def removeMax(tree: Tree): (BigInt, Tree) = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring((maxValue, leftTree) =>
   leftTree.isAVL && checkGreatest(maxValue, leftTree) &&
   (leftTree.size == tree.size - 1) &&
   (leftTree.height == tree.height || leftTree.height == tree.height - 1)  &&
   tree.toList.contains(maxValue) && leftTree.toList.subsetOf(tree.toList)
  )
```

Delete

delete B:

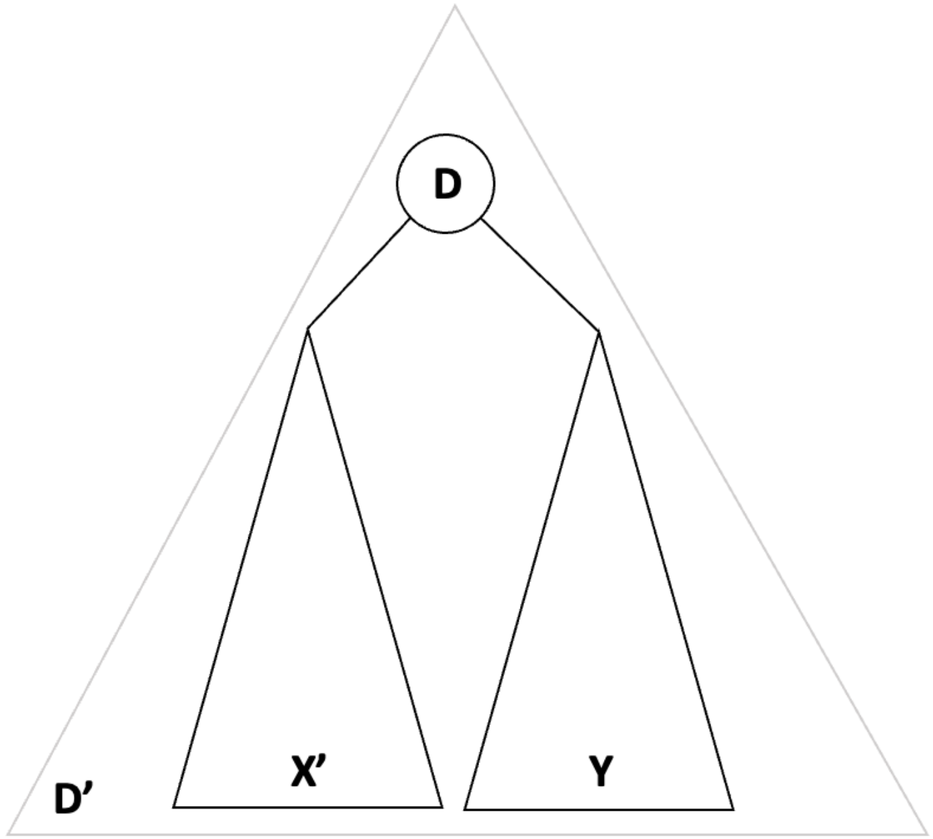
- locate B
- replace B with its predecessor
- perform balancing operations




```scala
def deleteAVL(tree: Tree, key: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL &&
   !newTree.toList.contains(key) && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size || newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def replaceRootWithMax(tree: Tree): Tree = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def removeMax(tree: Tree): (BigInt, Tree) = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring((maxValue, leftTree) =>
   leftTree.isAVL && checkGreatest(maxValue, leftTree) &&
   (leftTree.size == tree.size - 1) &&
   (leftTree.height == tree.height || leftTree.height == tree.height - 1)  &&
   tree.toList.contains(maxValue) && leftTree.toList.subsetOf(tree.toList)
  )
```

Delete

- **delete B:**
  - locate B
  - replace B with its predecessor
  - perform balancing operations

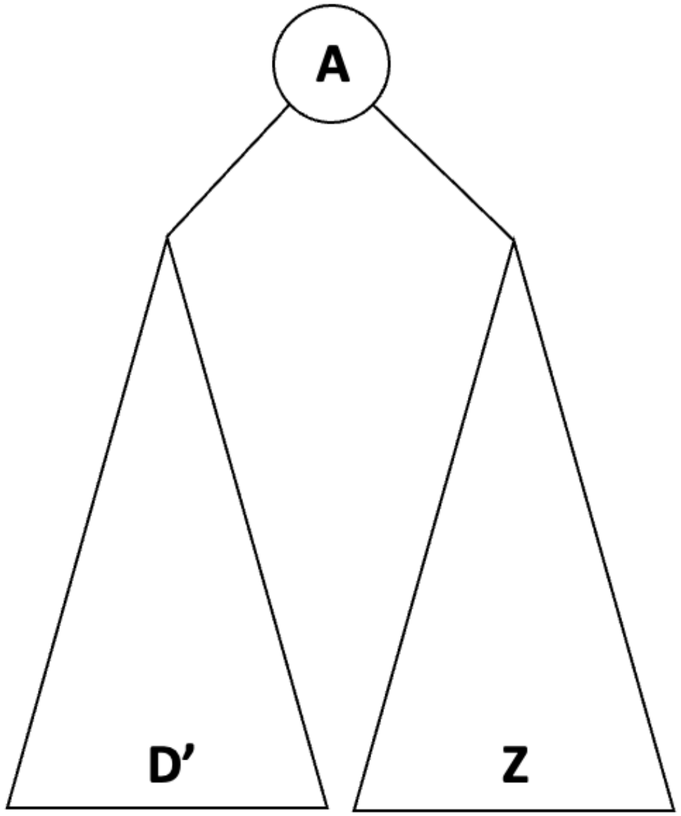



```scala
def deleteAVL(tree: Tree, key: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL &&
   !newTree.toList.contains(key) && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size || newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def replaceRootWithMax(tree: Tree): Tree = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def removeMax(tree: Tree): (BigInt, Tree) = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring((maxValue, leftTree) =>
   leftTree.isAVL && checkGreatest(maxValue, leftTree) &&
   (leftTree.size == tree.size - 1) &&
   (leftTree.height == tree.height || leftTree.height == tree.height - 1)  &&
   tree.toList.contains(maxValue) && leftTree.toList.subsetOf(tree.toList)
  )
```

Delete

delete B:

- locate B
- replace B with its predecessor
- perform balancing operations




```
def deleteAVL(tree: Tree, key: BigInt): Tree = {
  require(tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL &&
   !newTree.toList.contains(key) && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size || newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def replaceRootWithMax(tree: Tree): Tree = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring(newTree =>
   newTree.isAVL && newTree.toList.subsetOf(tree.toList) &&
   (newTree.size == tree.size - 1) &&
   (newTree.height == tree.height || newTree.height == tree.height - 1)
  )

def removeMax(tree: Tree): (BigInt, Tree) = {
  require(tree.size > 0 && tree.isAVL)
}.ensuring((maxValue, leftTree) =>
   leftTree.isAVL && checkGreatest(maxValue, leftTree) &&
   (leftTree.size == tree.size - 1) &&
   (leftTree.height == tree.height || leftTree.height == tree.height - 1)  &&
   tree.toList.contains(maxValue) && leftTree.toList.subsetOf(tree.toList)
  )
```

Results and Future Work

- **Completed:**
  - verified *Search* operation
  - verified *Balancing* operations
  - verified *Insert* operation
  - verified *Delete* operation

Results and Future Work

Completed:

- verified *Search* operation
- verified *Balancing* operations
- verified *Insert* operation
- verified *Delete* operation

Work in progress:

- implement and verify the *JOIN* operation

  Just Join for Parallel Ordered Sets - G. Blelloch, D. Ferizovic, Y. Sun, 2016

```
def joinAVL(treeLeft: Tree, k: BigInt, treeRight: Tree): Tree = {
  require(treeLeft.isAVL && treeRight.isAVL &&
          checkGreatest(k, treeLeft) && checkSmallest(k, treeRight))
}.ensuring(tree =>
    tree.isAVL && tree.size == treeLeft.size + treeRight.size + 1 &&
    union(treeLeft.toSet, treeRight.toSet, k) == tree.toSet
  )
```

Results and Future Work

Completed:

- verified *Search* operation
- verified *Balancing* operations
- verified *Insert* operation
- verified *Delete* operation

Work in progress:

- implement and verify the *JOIN* operation

  Just Join for Parallel Ordered Sets - G. Blelloch, D. Ferizovic, Y. Sun, 2016

Future work:

- verify the *SPLIT* operation
- implement a generic for Ordering
- write report

```
def joinAVL(treeLeft: Tree, k: BigInt, treeRight: Tree): Tree = {
  require(treeLeft.isAVL && treeRight.isAVL &&
          checkGreatest(k, treeLeft) && checkSmallest(k, treeRight))
}.ensuring(tree =>
    tree.isAVL && tree.size == treeLeft.size + treeRight.size + 1 &&
    union(treeLeft.toSet, treeRight.toSet, k) == tree.toSet
  )
```

Thank you!