Self-Balancing Binary Search Tree

Jackson Gregory
gregoryj17@asmsa.org

Objectives

- Create a generic binary search tree that can add and remove elements that self-balances.
- Make adding elements have a big O of log(n).

Procedure

- The tree was modeled after an AVL tree.
- A binary search tree made in a previous programming class was used, but updated to serve the new purposes.

Adding Elements

- Elements were passed along the tree, directed according to their value, until they reached an empty spot to occupy.
- If this unbalanced the tree, rotations were performed.

Adding Elements (Code Snippet)

```
if(cv.compareTo(t)>0)
{
   if(left==null){
      left=new Node<T>(t);
      left.height=0;
      this.height=Math.max(left.height+1,this.height);
      left.parent=this;
      left.tree=tree;
   }
   else{
      left.add(t);
      this.height=Math.max(left.height+1,this.height);
      while(isUnbalanced()){
         if(left==null||right.height>left.height)rotateLeft();
         else rotateRight();
      }
   }
}
```

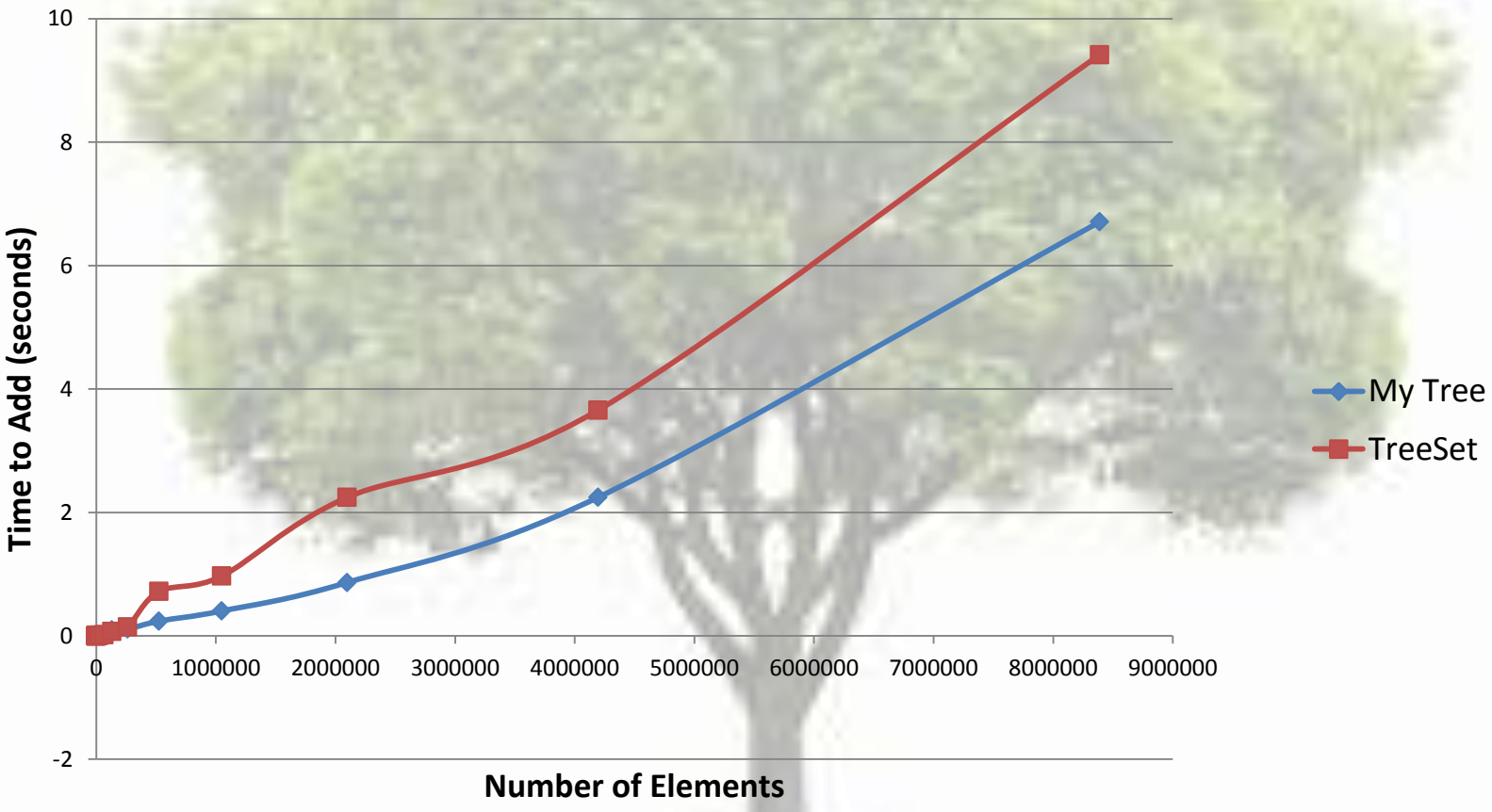
Removing Elements

- When an element was removed, it was replaced when possible. The maximum of the elements to its left was preferred, with the minimum of its right used if the left was not present.
- After a removal, every element from the removed upwards was checked for imbalances. When imbalances were present, rotations were performed.

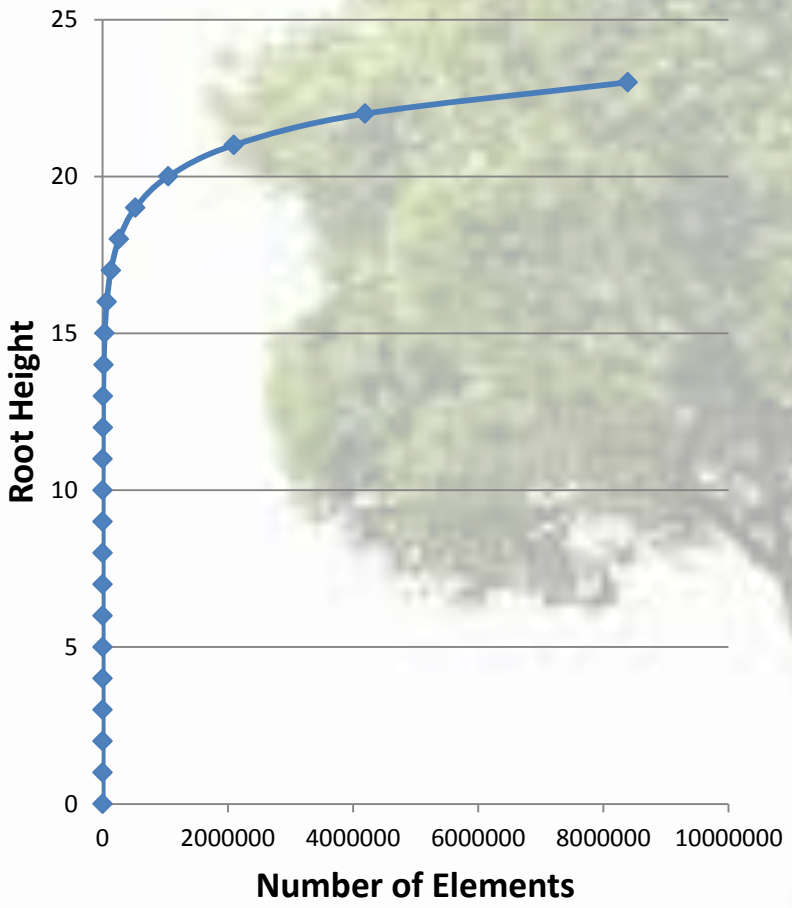
Removing Elements (Code Snippet)

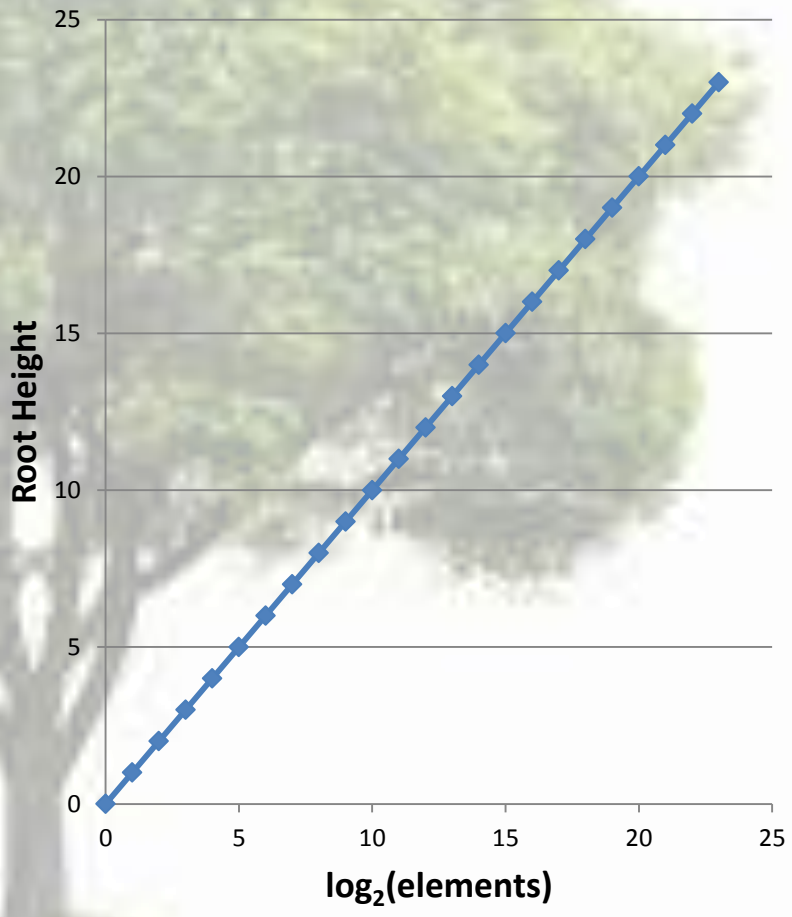
```
if(left==null&&right==null){
   if(parent!=null){
      if(this==parent.left)parent.left=null;
      else parent.right=null;
      Node<T> node=parent;
      parent.recalculateHeight();
      while(node!=null){
         while(node.isUnbalanced()){

if(node.left==null||node.right.height>node.left.height)node.rotateLeft();
            else node.rotateRight();
         }
         node=node.parent;
      }
   }
   else tree.rootRemove();
}
```

Adding Elements (Data)

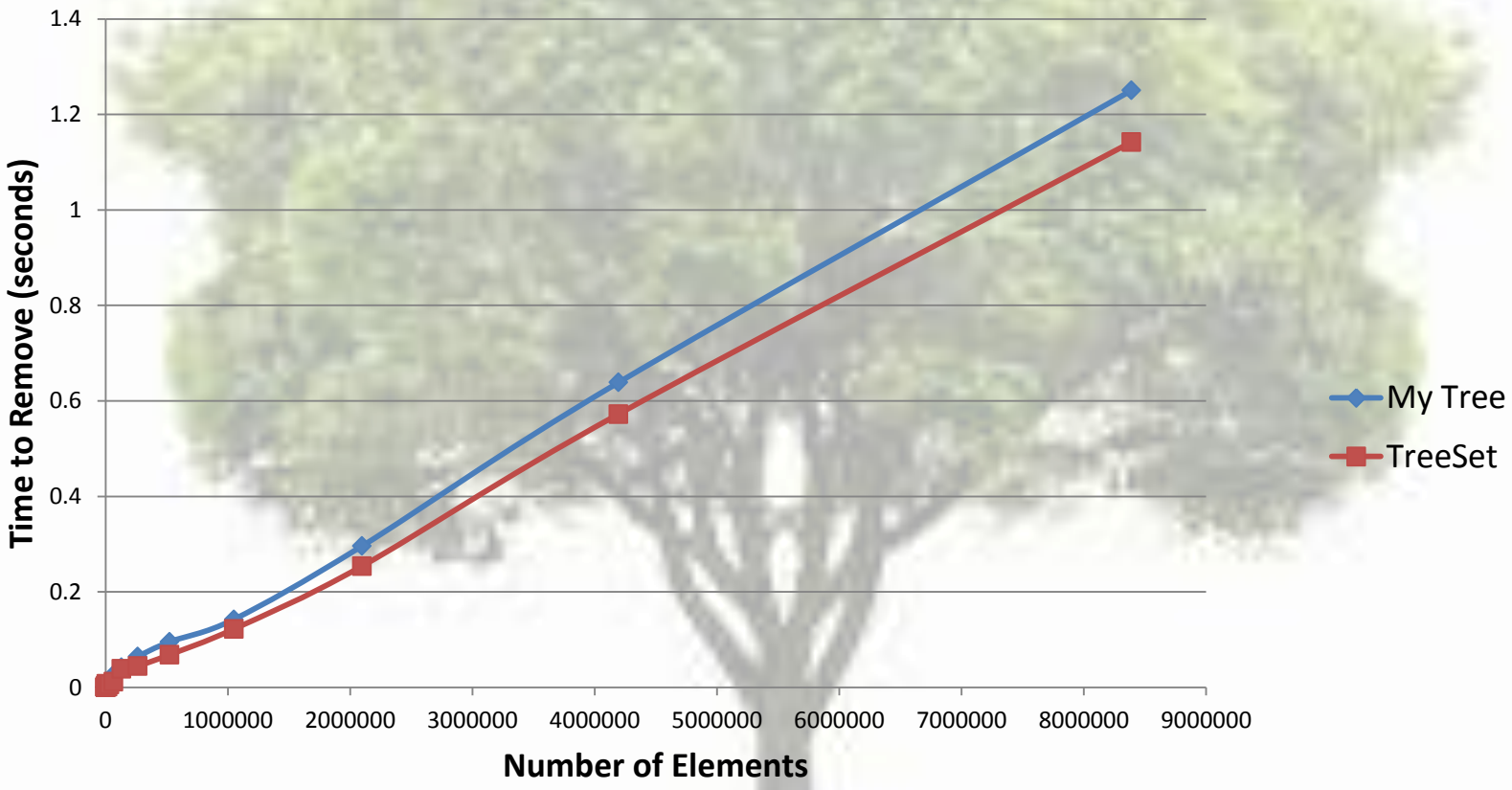


Height (Data)


25
20
15
10
5
0
Root Height
0
2000000
4000000
6000000
8000000
10000000
Number of Elements


25
20
15
10
5
0
Root Height
0
5
10
15
20
25
log2(elements)

Removal (Data)



Time Complexity

- Adding: When an element is added, since the tree is balanced, its paths will be approximately even size. Thus the big-O time complexity is O(log(n)), where n is the number of elements already in the tree.
- Height: Since height is cached, retrieving height has a big-O time complexity O(1).

Time Complexity

- Removal: When an element is removed, it must be found. It must also retrieve a node from lower in the tree. Overall, this scans the height of the tree, resulting in a big-O time complexity of O(log(n)), where n is the number of elements in the tree.

Graphical Time Complexity

- Adding: Java's TreeSet has a known time complexity of O(log(n)). The created tree has similar time scaling. The time seems to increase faster than O(log(n)) because in addition to the additions, each tree must perform additional rotations as the number of elements increases.

Graphical Time Complexity

- Removal: Similar to addition, TreeSet has a known time complexity of O(log(n)). The created tree scales similarly with the number of elements. As with addition, higher numbers of removed elements cause more rotations, creating a time graph which appears to have faster growth than O(log(n)), even though each individual removal scales at a logarithmic rate.

Self-Balancing Binary Search Tree

Jackson Gregory
gregoryj17@asmsa.org