Literature Review on AVL Trees

Conducted by:

Navid Galt

April 05, 2017

Department of Software Engineering- INFS 519- Professor Russell

George Mason University

Fairfax, VA 22030

Introduction

Trees are a concept that can be applicable in many other situations other than object-oriented programming. However, in this paper will focus on and stick to trees used in object-oriented programming, specifically AVL trees used in Java. This paper will help to explain the concepts of different operations that distinguish AVL trees from it's brethren according to several tutorial literatures. In the first portion of the analysis will discuss the concept of rebalancing and rotation and the differences in two of the tutorial literatures. In the second portion of the analysis bounded vs. unbounded operations in AVL trees and opaque areas surrounding this topic will be discussed. Finally, this paper will conclude with additions to pseudocode provided by the instructor that will transform a 2/3's tree into a B tree.

Analysis

Rebalancing & Rotations

Rebalancing is a step that occurs subsequently of retracing (when your program checks the parent Nodes in your tree against the invariants/standards of the AVL tree). After this check is done if the tree is unbalanced, as in there is a difference of 2 or more levels on either side of the parent, it must be rebalanced and rotation must occur. This is done without adjusting the order in which the data is sequenced. In the process of rebalancing there are two different overlying situations that could occur. First, where a single/simple rotation is necessary. Second, where a double rotation is necessary. The first situation is labeled as a "right right" rotation or "left left" and the latter situation is labeled as a "left right" or "right left" rotation. These different scenarios all depend on which side is heavier. For instance, let's consider the root of a tree x and it's right child y. In a single rotation if the y is right heavy, then this would call for a left rotation of x. This then sets y as the new root. Vice versa, if y was x's left child and y was left heavy, then this would call for a right rotation of x. The same concept applies for double rotation, except this time we're combining two single rotation's into one. For instance let's consider the same example above, where y is a right child of x, however this time y is left heavy. This instance first requires a right rotation of child y and then a left rotation of parent x. Vice versa, if y were x's left child and was right heavy it would require a left rotation of y and a right rotation of x.

To help completely understand this concept a series of tutorials supplied by the instructor and a supplementary tutorial from YouTube were used. No specific tutorial assisted in complete comprehension of this topic, yet a combination of the tutorials did the trick. The Wikipedia tutorial explained this concept most concisely. This reason for this is because the rotation concept was clearly separated into two different cases which were accompanied with graphs and explanatory text that helped to ease assimilation and understanding. The separation of sub-operations within rotation, made the text more readable and easier to follow than the first tutorial. It also provided code that was easier to follow compared to the first tutorial.

Bounded vs. Unbounded Insertion and Deletion

There are two different approaches to executing insertion and deletion operations inside of AVL Trees, using bounded vs. unbounded. Bounded insertion and deletion considers the height difference for each node also known as the "balancing factor". In these cases the node's ancestors are checked for consistency against the invariants of AVL trees. This operation is called retracing. An unbounded insertion or deletion uses the actual height of the subtree as the balancing factor. This approach is a little less intuitive as you must know which balance factors to replace during a rotation.

This topic is difficult to comprehend from a single tutorial standpoint. In a bounded insertion/deletion you use the difference of the two subtrees when comparing them, however this is not the case for unbounded. It remains a tad bit ambiguous as to what exactly we're comparing the height of the subtree to, in an unbounded insertion or deletion, so that the program will know if a rotation is necessary. The code implementation for this is also a bit unclear in this tutorial. All code examples in the first tutorial are written in C or C++ and therefore the logic supplemented is a bit difficult to follow.

Had the first tutorial's code been written in Java, understanding it would have been more easily surmountable for the reader. The author in the first tutorial leaned heavily on his coding to explain the concepts.

These syntactical differences make it more difficult for a reader, who's comprehension is mostly in Java, understand the variances between bounded vs. unbounded insertion and deletion.

Original Pseudocode Portion

(code sourced from: "Imagine! Java: Programming Concepts in Context" by Frank M. Carrano, First Edition)

Inserting Code

```
insert (ttTree, newItem)
// Inserts newItem into the 2-3 tree ttTree whose items have
// distinct search keys that differ from newItem's search key.
.
.  Let sKey be the search key of newItem
.  Locate the leaf leafNode in which sKey belongs
.  Add newItem to leafNode
.
.  if ( leafNode how has three items )
.     split ( leafNode )

split (n)
// Splits node n, which contains 3 items.  Note: if n is
// not a leaf, it has 4 children
.
.  if ( n is the root )
.     Create a new node p
.  else
.     Let p be the parent of n
.
.  Replace node n with two nodes, n1 and n2, so that p is their parent

.  Give n1 the item in n with the smallest search-key value
.  Give n2 the item in n with the largest search-key value
.
.  if ( n is not a leaf )
.  {
.  .   n1 becomes the parent of n's two leftmost children
.  .   n2 becomes the parent of n's two rightmost children
.  }
.
.  Move the item in n that has the middle search-key value up to p
.
.  if ( p now has three items )
.     split ( p )
```

Retrieving an Item Code

```
retrieveItem (ttTree, searchKey)
// Returns from the nonempty 2-3 tree ttTree the
// item whose search key equals searchKey.  The operation
// fails and returns null if no such item exists.
.
.  if ( searchKey is in ttTree's root node r )
.  {  // the item has been found
.  .  return the data portion of r
```

```
. }
.
. else if ( r is a leaf )
.    return null   // failure
.
. else if ( r has two data items )
. {   if ( searchKey < smaller search key of r )
. .     return retrieveItem(r's left child, searchkey)
. .  else if ( searchKey < larger search key of r )
. .     return retrieveItem(r's middle child, searchkey)
. .  else
. .     return retrieveItem(r's right child, searchkey)
. }
.
. else
. {   if ( searchKey < smaller search key of r )
. .     return retrieveItem(r's left child, searchkey)
. .  else
. .     return retrieveItem(r's right child, searchkey)
. }
```

Pseudocode Changes (Marked all in red)

Inserting Code

```
insert (ttTree, newItem)
// Inserts newItem into the B tree ttTree whose items have
// distinct search keys that differ from newItem's search key.

Let sKey be the search key of newItem
Locate the leaf leafNode in which sKey belongs
Add newItem to leafNode

//Edit the amount of items/elements possible inside of one node
if ( leafNode now has "L" items )
{
        split ( leafNode )
}

method split (n)
// Splits node n, which contains "L" items.  Note: if n is
// not a leaf, it has "m" + 1 children
//"m" is designated as a variable for the amount of children possible in a B tree, "L" is designated as a variable for
the amount of items a node can contain

//these statements help set a reference variable for the parent node
if ( n is the root )
{
        Create a new node reference variable p
}
else
{
```

```
        Let p be a reference variable to the parent of n
}

//Splitting n child into specified m amount of nodes
Replace node n with "m" nodes, n1 - nm, so that p is their parent

//According to child's search-key value this line will separate the items into the respective children of the parent node
Give n1- nm the item(s) respectively according to search-key value

//if n is a parent then split children accordingly
if ( n is not a leaf )
{
        n1- nm become the parents of n's "L" children, respectively
}

Move the item in n that has the middle search-key value up to p

//if parent node exceeds limit then split it and do the same
if ( p over-exceeds "L" items )
{
        split ( p )
}
```

Retrieving an Item Code

```
retrieveItem (ttTree, searchKey)
// Returns from the nonempty B ttTree the
// item whose search key equals searchKey. The operation
// fails and returns null if no such item exists.

if ( searchKey is in ttTree's root node r )
{
        // the item has been found
        return the data portion of r
}

else if ( r is a leaf )
{
        return null   // failure
}
// this else statement will search through the whole tree based on the search-key using recursion
else
{
        for(every item in r node)
        {
                if(searchKey < the current data item)
                {
                        return retrieveItem(r's current data item's child node, searchkey)
                }
        }
        if(searchKey > the last element in the data item's array)
        {
```

```
        Return retrieveItem(r's rightmost child, searchkey)
}
```

Conclusions

In this paper a thorough examination of several literatures provided by the instructor, was conducted. From the analysis and from the gathered knowledge an explanation of several operations used in AVL trees was provided. Included in this paper was pseudocode additions in converting functions from use in a 2/3 tree to a B tree. Trees have their many uses including providing us with oxygen, but in object-oriented programming we like to use them for efficiently sorting and storing data!

References

[1] "AVL Trees." AVL Trees. N.p., n.d. Web. 05 Apr. 2017. <http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_avl.aspx>.

[2] "AVL Tree." Wikipedia. Wikimedia Foundation, 04 Apr. 2017. Web. 05 Apr. 2017. <https://en.wikipedia.org/wiki/AVL_tree>.

[3] "AVL Trees." AVL Trees. N.p., n.d. Web. 05 Apr. 2017. <http://www.cs.wcupa.edu/rkline/ds/avl-trees.html#rebalance-operations>.

[4] Pfenning, Frank. "Lecture Notes on AVL Trees." (n.d.): n. pag. Lecture Notes on AVL Trees. 22 Mar. 2011. Web. 05 Apr. 2017. <http://www.cs.cmu.edu/~fp/courses/15122-s11/lectures/18-avl.pdf>.

[5] MIT. "6. AVL Trees, AVL Sort." YouTube. YouTube, 14 Jan. 2013. Web. 04 Apr. 2017. <https://www.youtube.com/watch?v=FNeL18KsWPc>.