A Study of Balanced Search Trees: Brainstorming a New Balanced Search Tree

Anthony Kim

June 17, 2005

1 Abstract

This project investigates three different balanced search trees for their advantages and disadvantages, thus ultimately their efficiency. Run time and memory space management are two main aspects under the study. Statistical analysis is provided to distinguish subtle differences if there is any. A new balanced search tree is suggested and compared with the three balanced search trees under study. Balanced search trees are implemented in C++ extensively using pointers and structs.

2 Introduction

Balanced search trees are important data structures. A normal binary search tree has some disadvantages, specifically from its dependence on the incoming data that significantly affects its tree structure hence its performance. Height of search tree is the maximum distance from the root of the tree to the farthest leaf. An optimal search tree is one that tries to minimize its height given data of some size. To improve its height thus its efficiency, balanced search trees have been developed that self-balance themselves into optimal tree structures that allows quicker access to data stored in the trees. For example red-black tree is a balanced binary tree that balances according to color pattern of nodes (red or black) by rotation functions. Rotation function is a hall mark of nearly all balanced search tree; they "rotate" to adjust subtree heights about a "pivot node." Many balanced trees have been suggested and developed: red-black tree, AVL tree, weight-balanced tree, B tree, and more.

3 Background Information

3.1 Search Tree Basics

This project requires a good understanding of binary trees and general search tree basics. A binary tree has nodes and edges. Nodes are the elements in the tree and edges represent relationship between two nodes. Each node in a binary tree is connected to zero to two nodes by edges. In general search tree, each node can have more than 2 nodes as in the case of B-tree. The node is called a parent and nodes connected by edges from this parent node are called its children. A node with no child is called a leaf node. Easy visualization of binary tree is a real tree put upside down on a paper with roots on the top and branches on the bottom. The grand parent of a binary tree is called root. From the root, the tree branches out to its immediate children and subsequent descendents. Each node's children are designated by left child and right child. One property of binary search tree is that the value stored in the left child is less than or equal to the value stored in parent. The right child's value is, on the other hand, greater than the parent's. ($Left <= Parent, Parent < Right$)

3.2 Search Tree Functions

There are several main functions that go along with binary tree and general search trees: insertion, deletion, search, and traversal. In insertion, a data is entered into the search tree and is compared with the root. If the value is less than or equal to the root's value then the insertion function proceeds to the left child of the root and compares again. Otherwise the function proceeds to the right child and compares the value with the node's. When the function reaches the end of the tree, for example if the last node the value was compared with was a leaf node, a new node is created at that position with the new inserted value. Deletion function works similarly to find a node with the value of interest (by going left and right accordingly). Then the funciton deletes the node and fixes the tree (changing parent children relationship etc.) to keep the property of binary tree or that of general search tree.

Search function or basically data retrieval is also similar. After traversing down the tree (starting from the root), two cases are possible. If there is a value in interest is encountered on the traversal, then the function replys that there is such data in the tree. If the traversal ends at a leaf node with no encounter of the value in search, then the function simply returns the otherwise. There are three kinds of travesal functions to show the structure of a tree: preorder, inorder and postorder. They are recursive functions that print the data in special order. For example in *pre*order traversal, as the prefix "pre" suggests, first the value of node is printed then the recursive repeats to the left subtree and then to the right subtree. Similary, in *in*order traversal, as the prefix "in" suggests, first the left subtree is output, then the node's value, then the right subtree. (Thus the node's value is output "in" the middle of the function.) Same pattern applies to the *post*order transversal.

3.3 The Problem

It is not hard to see that the structure of a binary search tree (or general search tree) that the order of data input is important. In a optimal binary tree, the data are input so that insertion occurs just right which makes the tree "balanced," the size of left subtree is approximately equal to the size of right subtree at each node in the tree. In an optimal binary tree, insertion, deletion, and search functions occur in $O(logN)$ with N as the number of data in the tree. This follows from that whenever data comparison occurs and subsequent traversal (to the left or to the right) the number of possible subset divides in half at each turn. However that's only when the input is nicely ordered and the search tree is balanced. It's also possible that the data are input so that only right nodes are added. $(Root- > right- > right- > right...)$ It's obvious that the search tree now looks like just a linear array and it is. This gives $O(N)$ to do insertion, deletion and search operations. This is not efficient. Thus balanced search trees are developed to perform its functions efficiently regardless of sequence of data input.

4 Balanced Search Trees

Three major balanced search trees are investigated. They are namely red-black tree, height-balanced tree, and weight-balanced tree, which are binary search trees.

4.1 Red-black tree

Red-black search tree is a special binary with a color scheme; each node is either black or red. There are four properties that makes a binary tree a red-black tree.

(1) The root of the tree is colored black.
(2) All paths from the root to the leaves agree on the number of black nodes.
(3) No path from the root to a leaf may contain two consecutive nodes colored red.
(4) Every path from a node to a leaf (of the descendents) has the same number of black nodes.

The performance of balanced search is directly related to the height of the balanced tree. For a binary, lg(number of nodes) is usually the optimal height. In the case of Red-black tree with n nodes, it has height at most $2lg(n+1)$. The proof is noteworthy, but difficult to understand. In order the prove the assertion that Red-black tree's height is at most $2lg(n+1)$ we should first define bh(x). bh(x) is defined to be the number of black nodes on any path from, but not including a node x, to a leaf. Notice that black height (bh) is well defined under the property 2 of Red-black tree. It is easy to see that black height of a tree is the black height of its root.

First we shall prove that the subtree rooted at any given node x contains at least $2^{bh(x)}-1$ nodes. We can prove this by induction on the height of a node x: The base case is bh(x) = 0, which suggests that x must be a leaf (NIL). This is true so it follows that subtree rooted at x contains $2^0 - 1 = 0$. The following is the inductive step. Let say node x has positive height and has two children. Note that each child has a black-height of either bh(x), if it is a red node, or bh(x)-1, if it is a black node. It follows that the subtree rooted at x contains at least: $2(2^{bh(x)} - 1 - 1) + 1 = 2^{bh(x)} - 1$. The first term refers to the minimum bounded by the sum of black height left and right. and the second term (the 1) refers to the root. Doing some algedra this leades to the right side of the equaiton. Having proved this, the maximum height of Red-black tree is fairly straightforward. Now, let h be the height of the tree. Then by property 3 of Red-black tree, at least half of the nodes on any simple path from the root to a leaf must be black. So then the black-height of the root must be at least h/2.

$n >= 2^{h/2} - 1$ `which is equivalent to` $n >= 2^{bh(x)} - 1$
$n + 1 >= 2^{h/2}$
$lg(n+1) >= lg(2^{h/2}) = h/2$
$h <= 2lg(n+1)$

Therefore we just proved that a red-black tree with n nodes has height at most $2lg(n+1)$.

4.2 Height Balanced Tree

Height-balanced tree is a different approach to bound the maximum height of a binary search tree. For each node, heights of left subtree and right subtree are stored. The key idea is to balance the tree by rotating around a node that has greater than threshold height difference between the left subtree and the right subtree. All boils down to the following property:

(1) At each node, the difference between height of left subtree and height of right subtree is less than threshold value.

Height balanced tree should yield $lg(n)$ height depends on the treshold value. An intuitive, less rigorous and yet valid proof is provided. Imagine a simple binary tree in the worst case scenario, a line of nodes. If the simple binary tree were to be transformed into a height balanced tree, the following process should do it.

(1) Pick some node near the middle of a given strand of nodes so that the threshold property satisfies $(absolutevalue(leftH() - rightH()))$
(2) Define this node as a parent and the resulting two strands (nearly equal in length) as left subtree and right subtree appropriately.
(3) Repeat steps (1) and (2) on the leftsubtree and the rightsubtree.

First, note this process will terminate. It's because at each step, the given strand will be split in two halves smaller than the original tree. So this shows the number of nodes in a given strand will decrease. This will eventually reach a terminal size of nodes determined by the threshold height difference. If a given strand is impossible to divide so that the threshold height difference holds, then that is the end for that sub recursive routine.

Splitting into two halves recursively is analogous to dividing a mass into two halves each time. Dividing by 2 in turn leads to $lg(n)$. So it follows the height of height-balanced tree should be $lg(n)$, or something around that magnitude. It is interesting to note that height balanced tree is roughly complete binary tree. This is because height balancing allows nodes to gather around the top. There is probably a decent proof for this observation, and simple intuition is enough to see this.

4.3 Weight Balanced Tree

Weight-balanced tree is very similar to height balanced tree. It is very the same idea, but just different nuance. The overall data structure is also similar. Instead of heights of left subtree and right subtree, weights of left subtree and right subtree are kept. The ″weight″ of a tree is defined as the number of nodes in that tree. The key idea is to balance the tree by rotating around a node that has greater than threshold weight difference between the left subtree and the right subtree. Rotating around a node shifts the weight balance to a favorable one, specifically the one with smaller difference of weights of left subtree and right subtree. Weight balanced tree has the following main property:

(1) At each node, the difference between weight of left subtree and weight of right subtree is less than the threshold value.

Weight balanced tree is also expected to give $lg(n)$ height depends on the threshold value. Similar approach used to prove height balanced tree is used to show $lg(n)$ of weight balanced tree. The proof uses mostly intuitive argument built on recursion and induction. Transforming a line of nodes, the worst case scenario in a simple binary tree, to a weight balanced tree can be done by the following steps.

(1) Pick some node near the middle of a given strand of nodes so that the threshold property satisfies $(absolutevalue(leftW() - rightW()))$
(2) Define this node as a parent and the resulting two strands (nearly equal in length) as leftsubtree and rightsubtree appropriately.
(3) Repeat steps (1) and (2) on the leftsubtree and the rightsubtree.

It is easy to confuse the first step in height balanced tree and weight balanced tree, but picking the middle node surely satisfies both the height balanced tree property and weight balanced tree. Maybe the weight balanced tree property is well defined, since the middle node presumably has same number of nodes before and after its position.

This process will terminate. It's because at each step, the given strand will be split in two halves smaller than the original strand. So this shows the number of nodes in a given strand will decrease. This will eventually reach a terminal size of nodes determined by the threshold weight difference.

Splitting into two halves recursively is analogous to dividing a mass into two halves each time. Dividing by 2 in turn leads to $lg(n)$. So it follows the height of weight-balanced tree should be $lg(n)$, or something around that magnitude. Like height balanced tree, weight balanced tree is roughly complete binary tree.

5 A New Balanced Search Tree: Median-weight-mix Tree

A new balanced search tree has been developed. The binary tree has no theoretical value to computer science, but probably has some practical value. The new balanced search tree will referred as median-weight-mix tree for each node will have a key, zero to two children, and some sort of weight.

5.1 Background

Median-weight-mix tree probably serves no theoretical purpose because its not perfect. It has no well defined behavior that obeys a set of properties. Rather it serves practical purpose mostly likely in statistics. Median-weight-mix tree is based on following assumption in data processing:

(1) Given lower bound and upper bound of total data input, random behavior is assumed, meaning data points will be evenly distributed around in the interval.

(2) Multiple "bells" is assumed to be present in the interval.

The first property is not hard to understand. This is based on the idea that nature is random. The data points will be scattered about, but evenly since random means each data value has equal chance of being present in the data input set. An example of this physical modeling would be a rain. In a rain, rain drops fall randomly onto ground. In fact, one can estimate amount of rainfall by sampling a small area. Amount of rain is measured in the small sampling area and then total rain fall can be calculated by numerical projection, ratio or whatever method. The total rain fall would be rainfall-in-small-area * area-of-total-area / area-of-small-area. The second assumption is based upon less apparent observation. Nature is not completely random, which means some numbers will occur more often than others. When the data values and the frequency of those data values are plotted on 2D plane, a "wave" is expected. There are greater hits in some range of data values ("the crests") than in other range of data values ("the trough"). A practical example would be height. One might expect well defined bell-shaped curve based on the average height.("People tends to be 5 foot 10 inches.") But this is not true when you look at it global scale, because there are isolated populations around the world. The average height of Americans is not necessarily the average height of Chinese. So this "wave" shaped curve is assumed.

5.2 Algorithm

Each node will have a key (data number), an interval (with lower and upper bounds of its "assigned" interval) and weights of left subtree and right subtree. The weights of each subtree are calculated is based on constants R and S. Constant R represents the importance of focusing frequency heavy data points. Constant S represents the importance of focusing frequency weak data points. So the ratio R/S consequently represents the relative importance of frequency heavy vs. frequency weak data points. Then tree will be balanced to adjust to a favorable R/S ratio at each node by means of rotating, left rotating and right rotating. Notice that each node is approximately at the middle of the interval (determined by one or two children) because the algorithm will try to balance the left and right subtrees.

An analogy is a rod balance and two kinds of weights which are placed various place along the rod. Balance would be achieved when the total torques (sum of weight * distance from the balacing point) on the left side and the right side are equal. This algorithm approximates this process by selecting a node with data that exists in the tree.

5.3 A Little Bit of Code

The following is a part of median-weight-mix tree, specifically updating weight part of the right rotating function.

```
...
//fix weight by formula
   if(pivot -> left != NULL)
      pivot -> leftw = pivot -> left -> leftw + pivot -> left -> rightw +
                          pivot -> left -> freq*R/S;
   else pivot->leftw = 0;
   if(pivot->right != NULL)
      pivot -> rightw = pivot -> rightw + pivot -> right -> leftw +
                          pivot -> right -> freq*R/S;
   leftchild -> rightw = pivot -> leftw + pivot -> rightw + pivot -> freq*R/S;
...
```

6 Methodology

6.1 Idea

Evaluating binary search trees can be done in various ways because they can serve a number of purposes. For this project, a binary search tree was developed to take some advantage of random nature of statistics with some assumption. Therefore it is reasonable to do evaluation on this basis. With this overall purpose, several behaviors of balanced search trees will be examined. Those are:

(1) Time it takes to process a data set
(2) Average retrieval time of data
(3) Height of the binary tree
(4) Average retrieval depth of data

The above properties are the major ones that outline the analysis. Speed is important and each binary tree is timed to check how long it takes to process input data. But average retrieval time of data is also important because it is the best indication of efficiency of the data structures. What is the use when you can input a number quick but retrieve it slow? The height of the binary tree is check to see if how theoretical idea works out in practical situation. Lastly, the average retrieval depth of data is measured for the same purpose behind the average retrieval time.

6.2 Detail

It is worthwhile to note how each behaviors are measured in C++. For measuring time it takes to process a data set, the starting time and the ending time will be recorded by function *clock*() under *time.h* library. Then the time duration will be (End-Time - Start-Time) / CLOCKS PER SEC. The average time retrieval of data will be calculated by first summing up time it takes to check each data points in the tree and dividing this sum by the number of data points in the binary tree. Height of the binary tree, the third behavior under study, is calculated by tree traversal, pre-, in- or post-order, by simply taking the maximum height or depth visited as each node is scanned. Average retrieval depth is just the sum of depths of all nodes * frequency of that data divided by the number of data in the input. This should be directly proportional to average retrieval time, assuming there is constant time between each recursive step.

There will be several test cases (identical) to check red-black binary tree, height-balanced tree, weight-balanced tree, and median-weight-mix tree. The first category of test run will be test cases with gradually increasing number of randomly generated data points. The second category of test run will be real life data points such as heights, ages, and others. Due to immense amount of data, some proportional scaling might be used to accommodate the memory capability of the balanced binary trees.

There are 14 randomly generated test cases. There are 4 real-life test cases. The following chart summarizes the test data used to test the four balanced search trees. The first and the second real life data are from USA Computing Olympiad contest scores. The third and fourth real life data are from American Mathematics Competition 12 and American Invitational Mathematics Exam scores respectively.

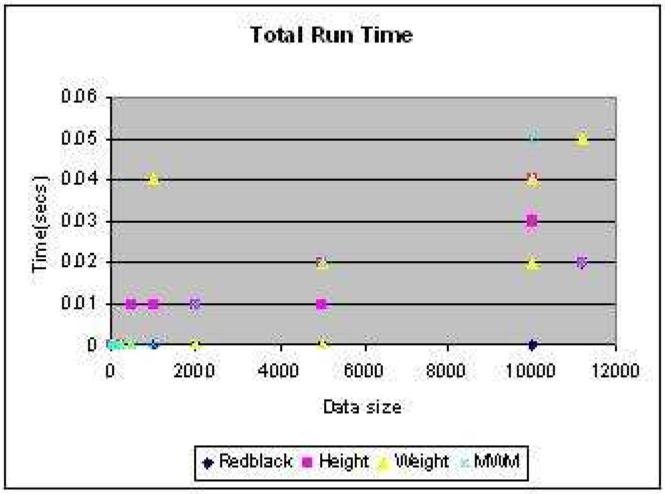
Test Cases		
Test Case Number	Number of Data Points	Range of Data Points
Randomly Generated		
1	20	1 - 50
2	50	1 - 200
3	100	1 - 400
4	500	1 - 2000
5	1000	1 - 10000
6	2000	1 - 20000
7	5000	1 - 20000
8	5000	1 - 20000
9	5000	1 - 20000
10	10000	1 - 100000
11	10000	1 - 100000
12	10000	1 - 100000
13	10000	1 - 100000
14	10000	1 - 100000
Real Life Data		
1	206	0 - 1000
2	87	0 - 1000
3	127947	0 - 150.0
4	11197	0 - 15

7 Result and Analysis

7.1 Total Run Time

Each balanced search tree was run against 18 test cases for its total run time. The total run time is pretty much process time that takes for a tree to get the input, insert the data, and balance its structure. This measures the strength of a balanced search tree against a large amount of data.

Total Run Time				
Test Case Number	RedBlack	Height	Weight	WeightMix
Randomly Generated				
1	0	0	0	0
2	0	0	0	0
3	N/A	.01	0	0
4	N/A	.01	0	0
5	N/A	.01	.04	0
6	N/A	.01	0	.01
7	N/A	.01	0	.02
8	N/A	.01	.02	.02
9	N/A	.02	.02	.02
10	N/A	.03	.02	.05
11	N/A	.03	.04	.05
12	N/A	.03	.04	.05
13	N/A	.04	.04	.05
14	N/A	.03	.040	.05
Real Life Data				
1	0	0	0	0
2	0	0	0	0
3	N/A	.08	0	N/A
4	0	.02	.05	.02

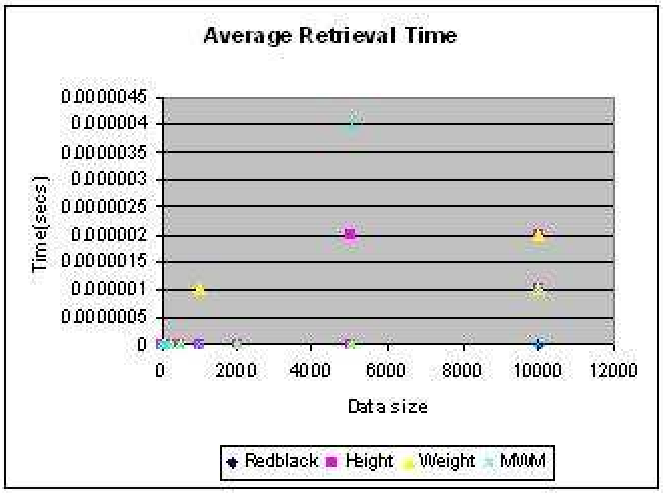


The general trend is that the total run time increases slowly as the size of test case increases. The testing shows that the height-balanced tree is strong in processing all inputs. It generally performs faster than weight-balanced tree and median-weight-mix tree. This is surprising that height-balanced tree outperforms weight-balanced tree even though the theoretical idea behind both trees are very similar. This is probably due to the difference in minimizing strategy of height of tree. Height-balanced tree tries to minimize its height by rotation functions. On the other hand, weight-balanced tree tries to make two subtrees nearly equal in weights. The rotatation functions for two trees are structured differently. Each recursive run in weight balanced tree cumulates weights of each subtrees while the recursive run in height-balanced doesn't necessary adds up heights. This could have been the main difference. Meanwhile, median-weight-mix tree did not perform as well as I hoped. This is probably because there are a lot of balancing going on within after each input is put in. Median-weight-mix tree calculates frequency and special weights based on R/S ratio. It is possible that this necessary functions slow down the binary search tree. Red black tree was not available for testing because it segmentation faulted for a large amount of data.

7.2 Average Retrieval Time

Each balanced search tree was run against 18 test cases for its average retrieval time. The average retrieval time is average time that takes for a tree to access each data in the input. The general sense is that the smaller the average time, the faster the tree accesses all of its data.

Average Retrieval Time				
Test Case Number	RedBlack	Height	Weight	WeightMix
Randomly Generated				
1	0	0	0	0
2	0	0	0	0
3	N/A	0	0	0
4	N/A	0	0	0
5	N/A	0	0	0
6	N/A	0	0	0
7	N/A	0	0	0
8	N/A	0	0	0
9	N/A	0	0	0
10	N/A	0	0	
11	N/A	0	0	0
12	N/A	0	0	0
13	N/A	0	0	0
14	N/A	0	0	0
Real Life Data				
1	0	0	0	0
2	0	0	0	0
3	N/A	0	0	N/A
4	0	0	0	0

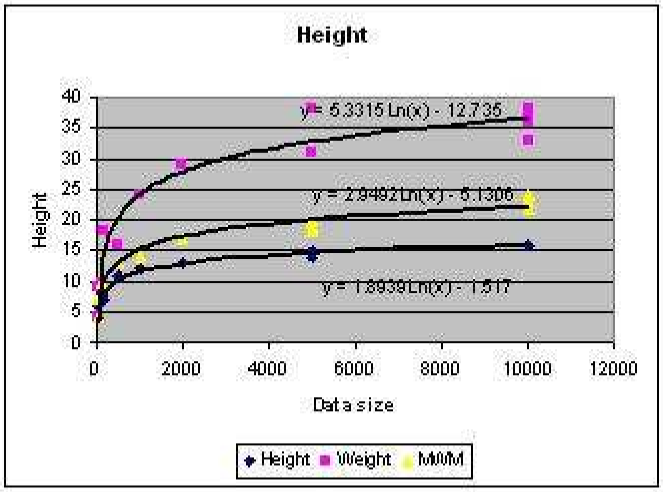


Unfortunately, the average retrieval time anaylsis did not go well. This is because the computer is too fast. Whatever it does, it does in within one second. So unless I have very large test data, it is not possible to get reasonable average retrieval time. The predicted general trend is that it increases as the size of input data increases. Red black tree was not available for testing because it segmentation faulted for a large amount of data.

7.3 Height

Each balanced search tree was run against 18 test cases for its height. The height basically shows how big a tree is. The bigger tree is, the longer it takes for the tree to search through the entire tree. The smaller the tree is, the shorter it takes for the tree to search through the entire tree.

Height				
Test Case Number	RedBlack	Height	Weight	WeightMix
Randomly Generated				
1	7	4	4	4
2	12	6	9	7
3	N/A	8	10	10
4	N/A	11	16	13
5	N/A	12	24	14
6	N/A	13	29	17
7	N/A	14	31	20
8	N/A	14	31	19
9	N/A	15	38	18
10	N/A	16	38	23
11	N/A	16	37	22
12	N/A	16	37	22
13	N/A	16	36	23
14	N/A	16	33	24
Real Life Data				
1	101	7	18	9
2	38	6	8	7
3	N/A	16	42	N/A
4	5596	13	24	12



It is interesting to note that each tree shows lograithmically increasing curve as the data size increased. This logarithmic trendline was expected because the height of a perfect binary search should be $lg(n)$. This characteristic was proved rigorously using induction for red-black tree. Height-balanced tree and weight-balanced tree should yield logarithmic curve because they are self-balancing trees after all. Self-balancing allows trees to obtain the optimal structure with "filled" branches and thus all nodes closer to the root. Median-weight-mix tree also shows logarithmic curve. Though its special R/S ratio makes it deviates from its mother the weight balanced tree. It seems to stablize the original weight-balanced tree, interestingly. This is probably caused by R/S ratio. I think R/S ratio skews the data points little bit so that in a way it behaves like interval balanced, or height balanced. As result, the logarithmic curve of median-weight-mix tree lies between the logarithmic curve of weight-balanced tree and the curve of height-balanced tree. The red black tree should give some insight into this analysis, but unfornately, the red black tree fails to run on large test cases due to a seemigly minor bug in the program.

Using Microsoft Spreadsheet, the best fit logarithmic linear regression lines and corresponding logarithm bases are found to be. For mathematics involved, see appendix A.

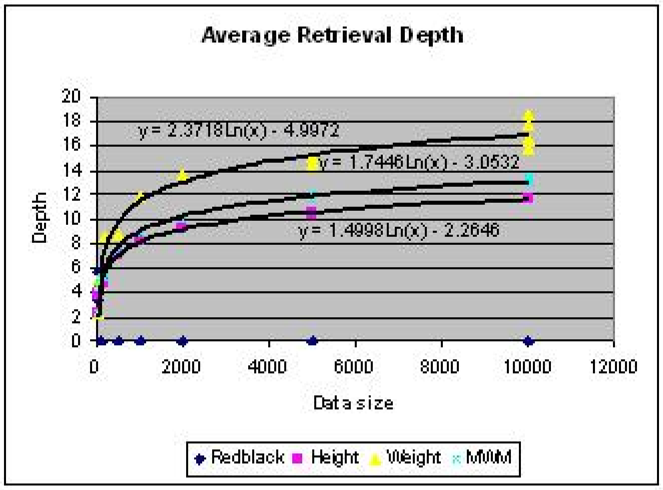
Weight-balanced : y = 5.3315 $ln(x)$ - 12.735, base = 1.2063
Median-weight-Mix: y = 2.9492 $ln(x)$ - 5.1306, base = 1.4036
Height-balanced : y = 1.8939 $ln(x)$ - 1.5170, base = 1.6956

The greater the base, the slower the logarithmic curve increases. So as the calculation of bases shows height-balanced tree performs the best among the three balanced search trees.

7.4 Average Retrieval Depth

Each tree was run against 18 test cases for average retrieval depth which should be proportional to the average retrieval time.

Average Retrieval Depth				
Test Case Number	RedBlack	Height	Weight	WeightMix
Randomly Generated				
1	3.5	2.35	2.35	2.35
2	5.68	3.9	5.08	4.42
3	N/A	4.85	5.44	5.31
4	N/A	7.01	8.68	7.42
5	N/A	8.22	11.82	8.82
6	N/A	9.24	13.72	9.96
7	N/A	10.38	14.67	12.02
8	N/A	10.48	14.52	11.65
9	N/A	10.47	15.16	11.65
10	N/A	11.57	18.59	13.22
11	N/A	11.59	15.68	13.02
12	N/A	11.65	16.43	13.12
13	N/A	11.67	16.57	13.37
14	N/A	11.54	17.77	13.13
Real Life Data				
1	49.03	4.87	8.51	5.19
2	18.56	4.4	4.55	4.74
3	N/A	0	42	N/A
4	2368.98	1.69	2.7	2.63



As mentioned earlier, average retrieval depth is proportional to average retrieval time, assuming constant time between each recursive steps. Fortunately, the average retrieval depth data is more readable than the average retrieval time. It's interesting that the average retrieval depth data shows logarithmic regression line. This is not obvious, but probably the fact that height of search tree follows logarithmic curve somehow forces the average retrieval depth to behave similarly. The average retrieval depth data also supports the observation made based on the height data. The median-weight-mix tree's curve lies between height-balanced tree's and weight-balanced tree's.

Weight-balanced : y = 2.3718 $ln(x)$ - 4.9972, base = 1.5244
Median-weight-Mix: y = 1.7446 $ln(x)$ - 3.0532, base = 1.7739
Height-balanced : y = 1.4998 $ln(x)$ - 2.2646, base = 1.9479

The greater the base, the slower the logarithmic curve increases. So as the calculation of bases shows height-balanced tree performs the best among the three balanced search trees. Note that height-balanced tree's base is experimentally shown to be 1.9479 which is pretty close to 2, the base number predicted theoretically.

8 Conclusion

This independent study of four balanced binary search trees yields some insight into the data structure. First, it has experimentally verified that balanced search trees' logarithmic characteristics. The logarithmic characteristics include the height of the tree and the average retrieval depth. Although, no reliable experimental data was produced, but the average retrieval time and the total run time are also expected to behave logarithmically. The average retrieval time should be logarithmic because it should be proportional to the average retrieval depth. Total run time's logarithmic characteristic is totally speculation. More research is needed to study the total run time. However, as computer gets faster, the possibility of such experiments is questionable.

Another observation is that height-balanced tree performs better than weight-balanced tree in almost all aspects. This is not expected because I thought they would behave similary with their similar divide and conquer strategy. I have an explanation for the unexpected result. I think the answer lies in the recursive functions of the two search trees. Even though both are structured in the same way, recursive balancing function of weight-balanced tree is cumulative. This is because weight of each node is the sum of the weight of left subtree, the weight of right subtree plus one. On the other hand, the recursive balancing function of height-balanced tree is somewhat conservative. The height at each node is the greater of heights of left subtree and right subtree plus one. Conservative is probably right descripter because this tends to keep the value as low as possible by taking just the greater value plus one.

The median-weight-mix tree showed an interesting performance. Though with specialized weighting system, the tree nevertheless shows logarithmic characteristics like red-black tree, height-balanced tree and weight-balanced tree. This experimentall proves that median-weight-mix tree is one of many of balanced search trees. More interestingly, the median-weight-mix tree's logarithmic characteristics lies between height-balanced tree's and weight-balanced tree's. It is as if the special weighting system deviated the orignal weight-balanced tree toward height-balanced tree. However, median-weight-mix tree did not necessarily performed better than other balanced search trees against real life data. But it was not that bad.

More studies can be done on balanced search trees. One suggestion is to expand the study to other balanced search trees such as B-tree etc. Also most test cases of various type can be generated and tested to give more accurate result. One thing I wanted to do but did not have time to is varying R/S ratio. Varying R/S ratio changes median-weight-mix tree and ultimately its performance. Varying R/S ratio is probably linked to logarithmic characteristics of the balanced search tree. It is possible that varying R/S ratio will change the logarithmic curve in the range defined by height-balanced tree's curve and weight-balanced tree's curve.

9 Reference

The following is the list of works I used for the project.

AVL Tree Insertions, Dublin City University <www.compapp.dcu/ie/ aileen/balance/>

Binary Trees; Monash Univeristy Information Technology <www.csse.monash.edu.au/ lloyd/tildeAlgDS

B-tree Algorithms <www.semaphorecorp.com/btp/algo.html>

Cormen Thomas H. Introduction to Algorithms MIT Press; 2nd edition, 2001

Data Structures and Algorithms, University of Western Austrailia CIIPS
<ciips.ee.uwa.edu.au/ morris/Year2/LDS210/ds(underscore)ToC.html>

Dictionary of Algorithms and Data Structures; National Institute of Standards and Technology;
<www.nist.gov/dads/>

Dynamic Binary Search Trees; ICS 165: Project in Algorithms and Data Structures, UC at Irvine, 2005

Introduction to Balanced Binary Search Trees CS212 Lectures
<http://newds.zefga.net/snips/Docs/BalancedBSTs.html>

Red-Black and B trees, CS 660: Combinatorial Algorithms, San Diego State University
<www.eli.sdsu.edu/courses/fall95/cs/660/notes/RedBlackTrees/RedBlack.html>

Red/Black Tree Demo; C321 Data Structures at University at Cincinnati;
<www.ececs.uc.edu/ franco/C321/html/RedBlack/redblack.html>

Red Black Trees, Winter 1997 Class Notes at McGill University;
<www.cs.mcgill.ca/ cs251/OldCourses/1997/topic18/>

Yanovksy, Vladimir; Visualization of Splay Tree,
<www.cs.technion.ac.il/ itai/ds2/framesplay/splay.html>

10 Appendix A: Some Mathematical Details

This section explains how to derive the bases of logarithm involved in balanced search trees from Microsoft Spreadsheet logarithmic linear regression lines. Note that each regression line is in form of "y = A $ln(x)$ + B." We use the following logarithm rule to find the base b.

$$\frac{\ln x}{\ln b} = \log_b x$$

Then it follows that

$$A = \frac{1}{\ln b}$$

$$\ln b = \frac{1}{A}$$

Therefore,

$$b = e^{\frac{1}{A}}$$

11 Appendix B: Other Balanced Search Trees

The following is a list of some balanced search trees.

2,3 Tree, 2, 3, 4 Tree, AVL tree, B-tree, Height-balanced Tree, Red black Tree, Splay Tree, Weight-balanced Tree.

12 Appendix C: Codes