T-106.1227 Data Structures and Algorithms Project

Final Report

Daniel Fogelholm and Hai Phan

1. Introduction

The aim of the report is to pick a programming language and compare the efficiency of operations in data structures to their theoretical estimates. This entails choosing a library from the language to investigate. The report in question will concern a library (Mozman, 2013) of binary search trees in the python programming language. The binary search trees will consist of the normal unbalanced binary search tree, and two balanced binary search trees: AVL trees and Red Black trees. The following operations of the data structures are chosen for testing; insert, search, and delete. The measure for efficiency will only rest on the time complexity of their performance. Memory usage was tested at one point as it is recognized that the augmented trees in our project use more memory to attain their high performance but it turned out not to make any difference for our purposes here.

2. Background

2.1.Python Programming language

The chosen programming language is a scripting language. They are widely used in software engineering as their clearness and ease of use enable higher productivity for software developers. This increase in productivity is paid for by significant worsening in performance. This is in contrast to compiled languages such as C/C++ that have higher performance but are more cumbersome to write code in. To remedy this weakness in Python it is possible to use extension libraries that allow for increased performance. Examples of this are found for applications in scientific computing where special libraries such as Numeric Python (NumPy) are used that are based on the C programming language. There are some drawbacks to this approach in the form of dependencies and the costs in maintaining compatibility between language and libraries. (Mueller & Lumsdaine, 2006)

In empirical comparisons to other programming languages, Python compares well. The design and writing of code takes half the time it takes for languages such as C, C++ and Java. While memory consumption is double that of C/C++, its variability is low which would suggest the automatic memory management in Python is quite good. It is also observed that the normal trade-off between memory and speed for compiled languages doesn't hold for scripted ones, i.e. the more memory used by the script, the slower it is. After loading a test program, the run time, as in searching through a data structure, for python is less than a factor of 2 slower than C, C++ (Prechelt, 2000). This shows that despite Python's script language features, it's still very attractive for computational problems due to its conciseness and the ease of access to its internals from C via the Python/C API (Behnel;Bradshaw;Citro;Dalcin;Seljebotn;& Smith, 2011)

.

2.2.Library

The chosen library is a package that provides binary search trees, Red Black trees and AVL trees in both Python and Cython. The data structures we are testing are written in Python using only the standard library. There are Cython implementations of them that have, according to test run by the library keeper, a large performance advantage over the standard cPython implementation we are testing with. For example, by using a 5000 unique random key AVL tree for search one hundred times, he achieves an improvement of 75 % when using Cython. This is mainly due to the function used to make comparisons the author states. (Mozman, 2013) This quite an interesting feature of the library implementation and goes to show how useful Python as a language can be. We, however, are going to only test the standard implementation in Python for the operations search, delete and insert.

The package has a lot of methods which enables us to perform this task but that will be discussed more thoroughly in the method and testing chapter. One thing that the library lacks is a way to measure the height of the constructed trees, which would've been helpful in testing to distinguish better between AVL and Red Black trees as there are differences in how strict their balance criterion is. We did find another library that had this feature and also featured the splay tree data structure but it did lack a Cython implementation (Sanderson, 2013).

2.3.Time Complexity

What is being measured for the different algorithms and data structures in our tests is their time complexity. There are best, average and worst cases, and we are concentrating on bounding the average case. This is expressed with big O-notation. It gives the upper bound for the growth rate of the time complexity as the problem size becomes larger. Different algorithms have different growth rates. For small problem sizes it's hard to distinguish between the performances of algorithms and one can get away with using inefficient data structures and algorithms. But when the size grows bigger, as is normal in today's computational problem space, the differences become apparent. In the analysis of algorithms the following growth rates have been determined (in order of slowest first): lg(n), n, nlg(n), n^2, 2^n, n!. Why they seem so elegant is because for large enough problem sizes only the leading term matters. (Skiena, 1998)

2.4. Binary Search Tree

Unlike a heap, which is an array you visualize as a tree, a binary search tree has actual pointers: parent (x), left child (x), right child(x). This increases the amount of memory needed but we get the benefit of being able sort the information in this improved tree structure. This invariance, the binary-search-tree property (BST property), entails that for every node x in the tree, if y is to the left of x then the key of y is smaller than x, and vice versa if it's to the right. (Lecture 5: Scheduling and Binary Search, 2011) (Cormen;Leiverson;Rivest;& Stein, 2001)

Search is an operation of BST that can be done in the average case in O(h) time, where h is the height of the tree. This is equivalent to O(log n) where n is the number of nodes in the tree, i.e. its size. Starting from the root, the searched for value is compared to that in the node. If it is the same, then it stops and reports the location. If it's smaller then it goes to the left child, where all the smaller values are due to the BST property. If it's larger then it goes to the right child. If it does not find a match, it will reach a node with no value (a leaf) and return NIL. It's a recursive procedure that can be written as "while loop" which tends to increase efficiency. (Cormen;Leiverson;Rivest;& Stein, 2001). Apparently, the comparisons made thusly when searching for the value in the tree can be inflated when using a high-level programming language such as Python. The comparisons made are not translated in the compiler into one three-way comparison but are separate and increases time complexity (Andersson, A Note on Searching in A Binary Search Tree, 1991). By using C, as in the Cython implementation, this can be remedied it would seem.

When *inserting* a new value to tree the BST property must continue to hold. Insertion is partly search as the place for the new value has to be found. This is also done in O(log n) time as the previous operation search. If one inserts values in a certain order the tree might start looking like a linked list and become unbalanced. This is pathological for the data structure and makes it unusable for any application that has that data insertion pattern. For this worst case performance deteriorates to O(n) as the search will take linear time through a linked list. For instance inserting numbers in a monotonously increasing or decreasing order will result in a linked list. BST are mostly used when keys are randomly inserted. This allows the balancing to be done in the insertion phase for free and gives an average case for time complexity at O(log n). (Cormen;Leiverson;Rivest;& Stein, 2001)

The opposite operation of insertion is *deletion*. It also involves finding the value. As the node is removed, the tree may have to update pointers around it or simply slice out to uphold the BST property. Its time complexity for the average case is O(log n). (Cormen;Leiverson;Rivest;& Stein, 2001)

2.5. AVL Tree

The balance problems of a binary search tree can be solved by using an AVL tree, a self-balancing binary search tree. It not only has the BST property but it also carries at each node a measurement of the nodes height. This increases the memory usage. By keeping score of every node's height, one can introduce a balancing property and it complements the BST property nicely by avoiding the embarrassing affair of the tree turning into a list and the associated penalties in increased time complexity. The AVL tree keeps itself on an even keel by a balancing factor where the difference in height between the left sub tree and the right sub tree at any node is at most plus or minus one. A perfectly balanced search tree with a balance factor of 0 would of course be ideal but due to the number of values it can be unattainable so we settle for this. (Miller & Ranum)

The worst case for an AVL tree the operations will be done in O(log(n)) time instead of O(n) as in a unbalanced binary search tree. It can be proven with help of the Fibonacci sequence that the upper bound is about 1.440 log(n). (Lecture 6: AVL Trees, AVL Sort, 2011)

Insert for the AVL tree data structure works in similar ways to the operation in the BST structure. It does, however, have to rebalance the tree after a new node has been inserted to keep its height even. The procedure then works its way upwards from the new node to its parent updating the balance factor. If the balance factor criterion is not fulfilled (the tree becomes right, left, right-left-right, left-right-right heavy), the procedure starts to perform rotations. It may sound easy but the actual programming of this procedure is very hard as the order in which the nodes move around is very important. One then has to keep track of how the grandparent's balance factor is affected by the rotations. Doing this effectively is the key for the AVL tree's good performance. After performing a *deletion* in an AVL tree you have to update the tree to keep it balanced which is also quite a challenge to implement. (Miller & Ranum)

An AVL tree performs best when the data inserted into it is in a sorted order. AVL tree's very strict balance criterion starts to work against it when data is random as it does a lot of unnecessary work in the form of more rebalancing. (Pfaff, 2004)

2.6.Red Black Tree

RBTree was invented 1972 by Rudolf Bayer, originally named symmetric Binary B-tree. It was intended to be the binary representation of 2-3-4 tree. The purpose is to make sure every leaf has the same height (Bayer, 1972). RBTree must satisfy a set of conditions besides the binary tree order. Every node in RBTree is colored with either black or red. The following conditions, called red-black properties, define a valid red-black tree.

1. Every node is colored black or red.
2. The root, and leaf nodes are black.
3. A red node has black children. This mean a red node cannot have red children or parent.
4. A simple path from node v to its descendant leaves must contain the same number of black nodes. This number is called black-height bh(v). Note: bh(v) does not count v itself.

When an operation such as insert or delete changes the tree structure, the red-black properties must still hold.

RBTree of size n (the emptied leaves not counted) will have height of at most 2log(n+1)=O(logn). Proof: Cormen, p 309

Because of this feature we can guarantee run time for query operations Search, Min, Max, Predecessor bounded by O(logn).

Insertion follows the BST order. The inserted node is then colored red. If one of the red-black properties are violated then proceed to fix it, with a series of rotation and recoloring (Cormen;Leiverson;Rivest;& Stein, 2001). The time to find insert position is O(logn). A single rotation or recolor takes constant time. But in worst case the fixing can go up to the root, which makes it O(logn). Adding all that and we have a single insertion taking O(logn) time, even in worst case in line with the AVL tree.

Deletion follows the same logic as insertion. We have the following cases.

- Case 1: If the node is a leaf just delete it. Replace the leaf with nil.
- Case 2: If it has one child, replace it with that child.
- Case 3: If it has two children, swap its content with its left-most right descendant. Then delete the node at its new location. Keep doing this and we will meet case 1 or case 2.

In case the deleted node is red, we are done. But if it is black, we have to do fixing because the black-height on some path has change. The time to find the node to delete, then possibly finding node to replace it is O(logn). This is because that only depends on height of the tree. A single swapping and deletion takes constant time. Each fixing operation takes constant time O(1). But in worst case the fixing can go up to the root, which makes it O(logn). Adding all that and we have a single deletion taking O(logn) time, even in worst case.

The implementation of an algorithm for balancing a Red Black trees is quite complex as adding a new node to an old one may result in 5 different shapes all with associated rebalancing. However, it is possible to simplify this and reduce the rebalancing cases by turning the tree from a 2-3-4 tree into a 2-3 tree. This eliminates half of the restructuring cases which simplifies the delete operation (Heger, 2004). This is done by adding balance information in order to keep right-edges horizontal. One then uses specific Skew and Split operations. For deletion, in this type of tree, at most 3 skews and two splits would be used. This is quite the contrast to the propagation that might happen in a Red Black Tree (Heger, 2004). This is an AA-tree but it was not included in the library. (Andersson, Balanced Search Trees Made Simple, 1993)

The performance of Red Black Trees is the best of our tree data structures when the items inserted are mainly in random order but do include some pockets of sorted order. This is because it doesn't start to behave pathologically like an unbalanced BST when the items are sorted, nor does it overdo the rebalancing when the items are inserted randomly like an AVL tree would do. (Pfaff, 2004)

3. Testing Methods

The library is bintrees 1.0.1 by mozman. It is not a standard Python library. To use it we have to install from source. We only test the trees implemented in Python. The library also it also support Cython but that is not our focus. The test is written with each function for an operation, taking only tree-size as parameter. (Mozman, 2013)

We test the operations with different tree sizes. The size of the tree is linearly increased. There are 128 data points for each graph. We are able to test with relatively big treesize, about 2^{15}-2^{16}. Also the data are evenly spaced for visibility. The data are chosen randomly. For the trees a node needs a dictionary item {key:value}. In our case only the keys are critical for the operations so we just set all the values to zero for faster performance.

The test was setup and run on a school Linux workstation: CPU Intel(R) Xeon(R) CPU E31230 @ 3.20GHz, RAM 7925 MB, Linux 3.2.0-39-generic, Python 2.7.3. The library was installed on user's space, not on whole system. The test code was written to use the library implementation of the trees. No additional hack was added. To measure the time of each operation we use module timeit. The same operation can be run multiple times and then average is taken as result. We tried to run the test when the workstation was fairly idle to avoid objective factors to affect the test run.

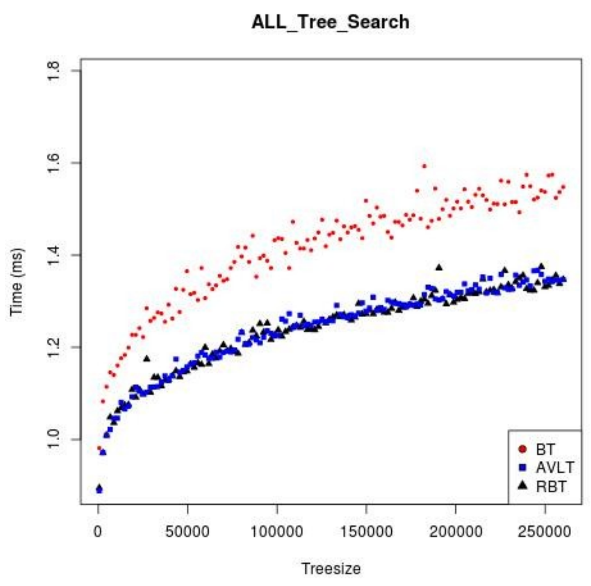
For searching we measure the time to search for 200 keys in trees of increasing sizes. The key for searching are chosen randomly and may not exist in the tree. The library has a method to query the value of a key. In case key is not in the tree, a default value can be returned.

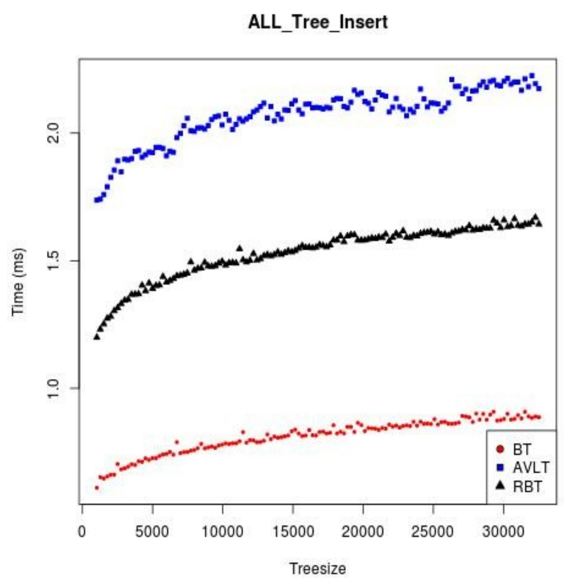
Insertion includes cases where identical keys are used. When inserting a key that already exist in the tree, the tree just replace {key,old_value} with {key,new_value}. The tree size then remains the same. Normally a value should be mapped to a key, probably via some hash function. The position of the dictionary item will only depend on the key. A good mapping function is needed to avoid collision so that different values are not likely to be mapped to the same key. The insertion test is done by first create the tree of certain size and then insert 100 keys one-by-one into the tree.

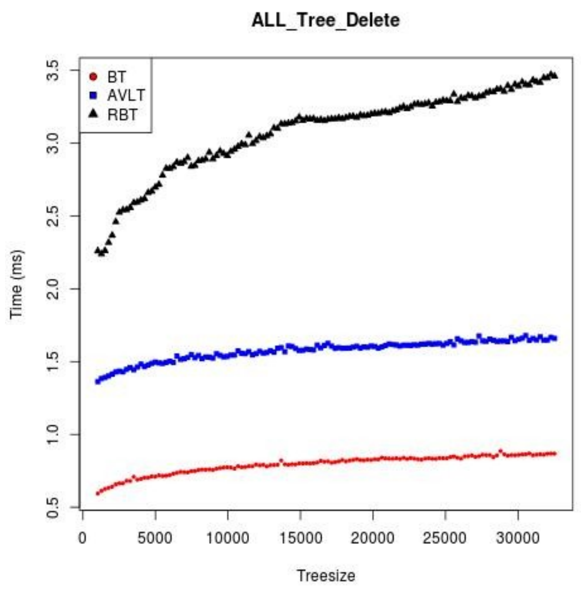
To test deletion we picked out 200 unique keys out of all the existing keys in the tree and remove them one by one. We are using del instead of discard method, which will crash if key does not exist. There is also pop method which will delete the key and return it.

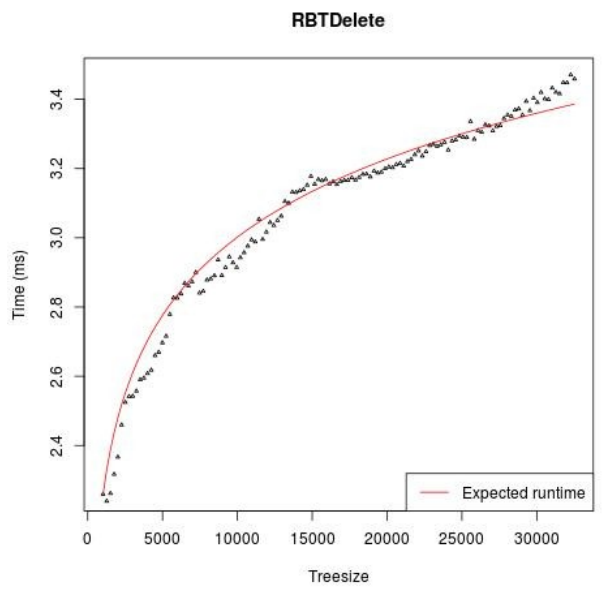
4. Discussion

We put all the trees together on one graph for comparison. We will look at three operations: search, insert and delete.









In general we can see that both AVL tree and Red-Black tree easily beat binary-search-tree for the search operation. This is plainly due to the dynamic balancing both trees perform which makes their height less than that of an unbalanced binary search tree even with random insertion. We can also see little differences in AVL tree and Red-Black tree regarding search. Though the few outliers in Red-Black Tree's case may be due to its less strict balance criteria which would explain the occasional divergence from the stricter AVL tree's performance in this regard. One thing to note is that the performance of binary-tree seems to scatter wider than the other two which might be due to small pockets of order in the randomness which would create isolated growth spurts in the tree's height and make it more unbalanced.

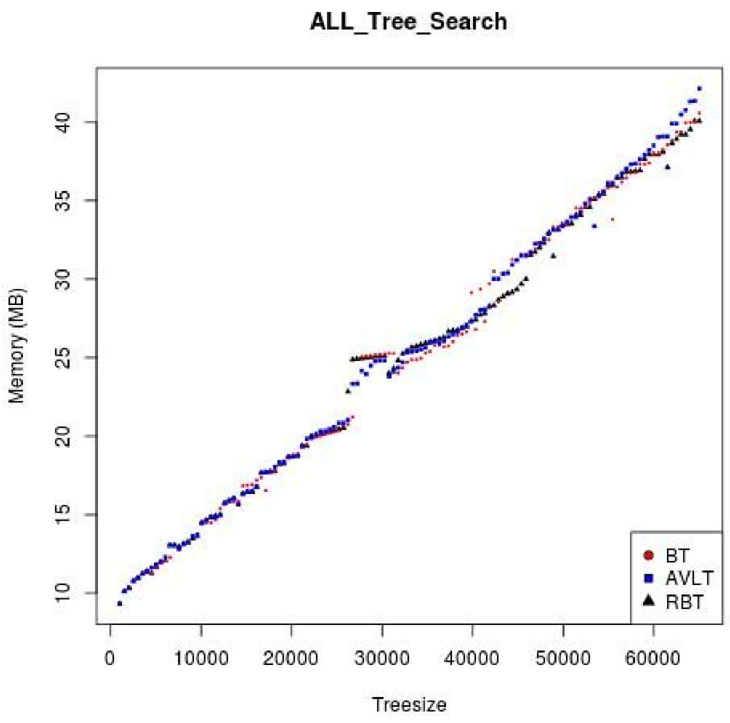
In the *insert* case the three types of tree has distinctively different performance. The binary tree is by far superior due to not having a maintenance function and so has to perform no rotations or anything when adding an item. The tree thus constructed is not worst case for binary tree because we choose the keys totally random which balances it somehow. However, by examining in detail the behavior of the binary tree's insertion it is possible to detect it breaking the O(logn) bound which would entail the height not being log(n) despite the random insertion.

The AVL Tree performs the worst in comparison when it comes to insertion and also seems to have more variation than other two. This may be due to its balancing criteria being very strict (height difference of only one allowed) which will have it perform a lot of rotations, maybe unnecessarily so, for the random insertion which deteriorates its performance. However, when examined in detail one notices that it doesn't break the O(logn) bound. The Red-Black-Tree outperforms AVL trees when we are inserting random keys. This is because of its less strict balancing criteria that will allow for a bit more unbalance which the random insertion will take care of thus leaving it doing less unnecessary rotations. Had the keys been inserted in a sorted order then the AVL tree would've outperformed both Red-Black-Tree and Binary-search-tree (Pfaff, 2004).

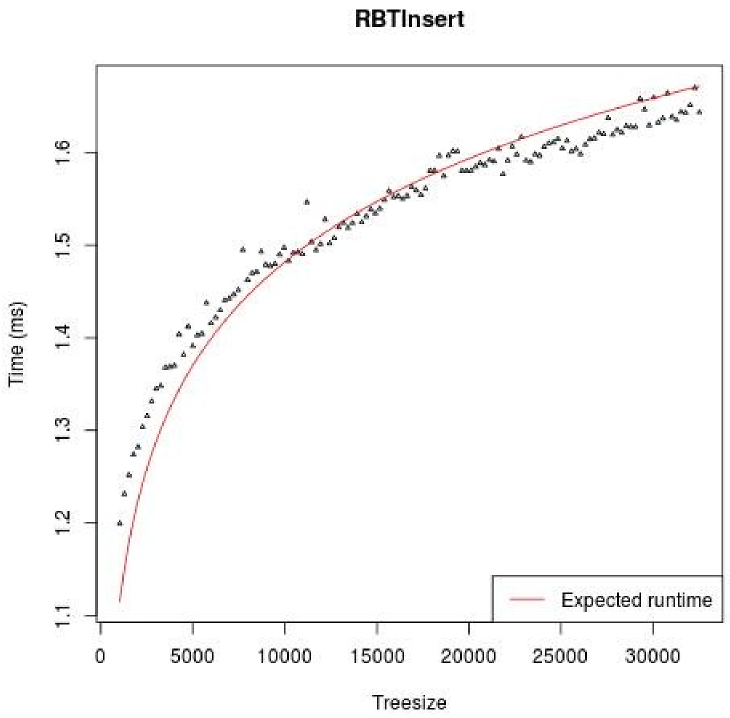
Although at local level some unusual variation can be spotted, in long term the runtime increases slowly as treesize approach infinity. In fact the increment rate seems to smoothly flatten near the tail for all the graph, except for *deletion* on Red-Black-Tree. The trouble the Red Black tree has for the *deletion* operation can be explained by its complicated rebalancing operations. As we mentioned in the background section of this tree, there are a larger number of cases the maintenance operations have to take into account while rebalancing the tree. In the worst case this can propagate up the tree in a recursive manner. So even though theoretically it is bounded by O(log n) due to implementation complexity this bound is broken. In cases where deletion is a frequent operation it might be better to use the mentioned AA-trees which, while using a bit more memory, drastically cuts down on the complicated rebalancing (Heger, 2004).

Delete for Binary-search-tree has the least time complexity as minimal updating of the tree data structure is done and the tree is almost balanced so searching grows logarithmically. For the AVL tree we find that it performs better than the Red-Black-Tree which is undoubtedly because of its limited maintenance operations.

We suspected at one point that maybe the memory of the trees may have an impact on the performance as some of our trees (the balanced ones) use more information. We ran tests for a couple of operations that showed that memory increases linearly with size for all trees. There were some jumps and patterns in the amount of memory used as the size of the trees increased but it was undoubtedly due to python's automatic memory manager performing its duties. As previously discussed in the section on the programming language, the automatic memory management in Python is quite good so it was felt this was not worth exploring in more detail.



Because of the way keys are generated. All these performance are measured for average case. Theoretically all the operations search/insert/delete are bounded in O(logn) for average case. We assume that the runtime is bounded by a line C*log(n_k) with n_k the size of the tree. If we assume at a point n_k, t_k is exactly at the bound we can calculate CC, the array of potential C values. And then try each one to find which has the tightest bound.



.
As illustrated here the expected bound is the closest estimation of the runtime, with the assumption that it follows a logarithmic trend. In this figure at the end of the curve, the data points seems to flatten out and settle just below the estimated line. There are still a few exceptions though.

Projected runtime for each tree and operation. Expected constant C in t(n)=C*log(n). Error rate is computed for the the last 20% of data points. It is the rate that runtime exceeds this expected threshold. Lower error rate means that using this C value can better estimate runtime for big n.

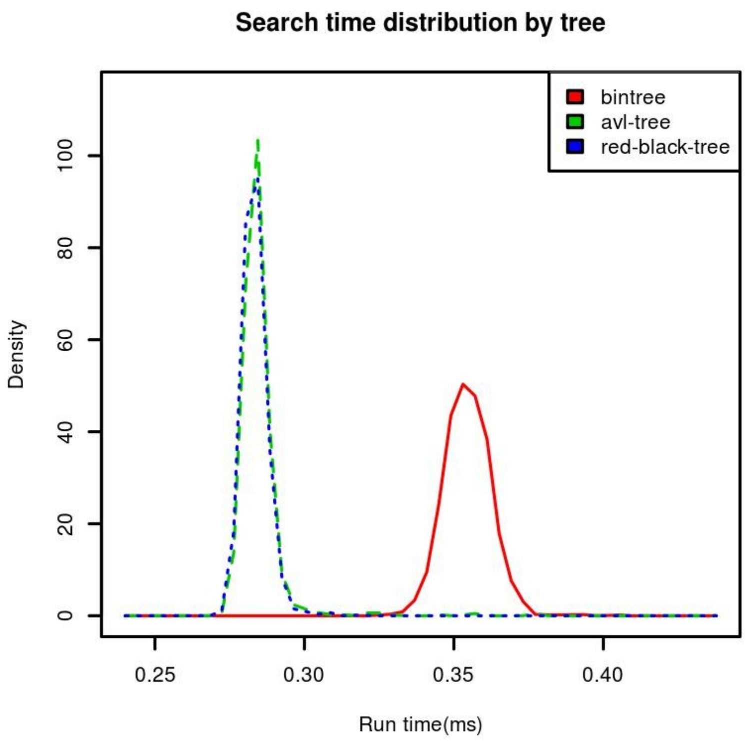
Trees	Search		Insert		Delete	
	C	error rate(%)	C	error rate(%)	C	error rate(%)
binTree	0.0855	56	0.0592	52	0.0579	44
avlTree	0.0750	44	0.1508	0	0.1146	0
rbTree	0.0748	48	0.1115	12	0.2248	96

We see that indeed as n gets to infinity searching in AVL-tree is as efficient as in Red-Black tree. Inserting into an AVL-tree takes longer undoubtedly due to the stricter balancing criteria and the subsequent increase in rotations as previously mentioned here.

Besides the average runtime we are interested in knowing the reliability of this library implementation. Because in reality searching is the most used operation, we decided to test searching on each type of trees. The treesize is fixed at 2^{15}, and we want to create trees with same set of keys. We just repeatedly call search for 100 random keys on the trees and calculate the median absolute deviation of the run time. This value tells us how far the real runtime can be from expected value. We chose MAD over standard deviation because we wanted a more robust measurement for statistical dispersion that is less susceptible to outliers.

Search 100 keys	mean	median absolute deviation
BT	0.3546	0.00760
AVLT	0.2849	0.00318
RBT	0.2836	0.00318

This distribution graph shows how the trees fare against each other in search operation. We compare kernel density distribution of the trees.



We can see immediately that the search time for avl tree and red-black tree are quite predictable. Also at this level they seem to have about the same performance. It is easy to see that for binary tree the lookup speed is visibly slower and the spread wider. We can explain this behavior by tree-height. Because for binary tree there is no guarantee that every path from root to a leaf is equal, some branches may have significantly bigger height than others. And the bound O(logn) can be broken. Because both avl-tree and red-black tree have mechanism to balance all the branches, they can have height bounded by O(logn). If we look closely the peak of AVL tree is a little higher. The dispersion for both Red Black Tree and AVL tree is the same, at least to accuracy we are able to get. Intuitively one would think that AVL would have smaller dispersion due to the stricter balancing criteria. The time scales are so small that comparison becomes very difficult.

5. Conclusions

For searching AVL tree and Red-Black tree gains remarkable advantage over Binary tree as the size increases. This is due to the fact that height of the binary tree is not bounded by O(logn). Although in average case binary tree is faster for insertion and deletion, there is no guarantee that it can avoid the worst case. Thus despite the good (average) performance of BST due to the random key insertion, it does not fare so well in comparison with its balanced cousins in searching tests. Red-Black-Tree has very heavy task to check every node if the rules still hold so it requires very careful implementation in order to help reduce the fix the tree after insert or delete.

For binary tree the worst case cannot be generated because of the implementation using recursion. Therefore to be safe it is recommended to either use AVL Tree or Red-Black Tree. Red-Black tree seems to fare better for building large trees, thus a good candidate if writing to the structure is done often. A good hash function is also needed because the query is based on keys. But this should be trivial since there are a lot of good hash functions. There is a problem with deletion performance on Red-Black-Tree, so another alternative may be needed in case it needs to do a lot of deletions.

There are also other auto-balance trees such as B-tree, AATree, Splay and Treap. This library however only implements these three types of binary search tree. In general the implementation is very close to theoretical descriptions. The way we conduct the test may have some impact on the result. But we use the same procedure for all the trees. So if one anomaly happen to only one tree, we are sure that it is mainly due to its implementation. Because we use public machine in school for testing, other external factor could have impact on the result as well. In general the library performs according to the trees' specifications, except some faults as discussed above. Such basic data structure should be implemented on low level language such as C for better performance and easy control of the environment.

Timetable

Participant\Phase	Defining	Solution	Final
Hai Phan	10	24	12
Daniel Fogelholm	14	16	18

Time spent by tasks

Participant\Phase	Meeting	Academic research	Code and Test	Report writing
Hai Phan	8	12	16	10
Daniel Fogelholm	8	20	5	15

Works Cited

Lecture 5: Scheduling and Binary Search. (2011). Retrieved 4 1, 2013, from MIT OpenCourseWare: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec05.pdf

Lecture 6: AVL Trees, AVL Sort. (2011). Retrieved April 1, 2013, from MIT OpenCourseWare: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec06.pdf

Andersson, A. (1991). A Note on Searching in A Binary Search Tree. *Software-Practice and Experience*, 1125-1128.

Andersson, A. (1993). Balanced Search Trees Made Simple. *In proc. Workskop on Algorithms and Data Structures*, 60-71.

Bayer, R. (1972). Symmetric binary B-trees: Data structure and maintenance algorithms.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D., & Smith, K. (2011). Cython: The best of both worlds. *IEEE Computing in Science and Engineering*, 1-9.

Cormen, T. H., Leiverson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to Algorithms, second edition.* Cambridge: The MIT Press.

Heger, D. A. (2004). A Disquisition on the Performance Behaviour of Binary Search Tree Data Structures. *UPGRADE CEPIs*, 67-73.

Miller, B., & Ranum, D. (n.d.). *Balanced Binary Search Trees.* Retrieved April 2, 2013, from Problem Solving with Algorithms and Data Structures using Python: http://interactivepython.org/courselib/static/pythonds/Trees/balanced.html

Mozman. (2013, February 24). *bintrees 0.3.0.* Retrieved February 21, 2013, from Python Programming Language – Official Website: https://pypi.python.org/pypi/bintrees/0.3.0

Mueller, C., & Lumsdaine, A. (2006). Runtime Synthesis of High-Performance Code from Scripting languages. *OOPSLA research paper*, 954-963.

Pfaff, B. (2004). Performance Analysys of BSTs in System Software. *SIGMETRICS/performance*, 1-12.

Prechelt, L. (2000). An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl. *IEEE Computer.*

Sanderson, T. (2013, January 9). *pybst 1.0.* Retrieved April 29, 2013, from Python Software Foundation: https://pypi.python.org/pypi/pybst/1.0

Skiena, S. S. (1998). *The Algorithm Design Manual.* New York: Springer Science+ Business Media.