# A Logarithmic Random Access Tree

Tom Morgan

March 30, 2007

**Abstract**

Making a data structure that performs like a dynamic array but functions in logarithmic time for all operations is the goal, but is by no means a trivial one. The obvious solution is to use a tree of some sort, but how?

By using a binary tree in which values are stored at the leaf nodes and each node keeps track of how many leaves there are below it, we can quickly achieve logarithmic random access, insertion and deletion in the average case but all operations are linear on the worst case. To balance the tree, a self-balancing binary search tree like a Red Black Tree or a Splay Tree is used. These trees are always balanced and by using their balancing mechanisms and changing them to use the random access system described above, the goal is reached.

**Keywords:** data structure, algorithmic efficiency, binary tree, red black tree, splay tree

## Background

Dynamic arrays (such as vectors and array lists) are commonly used among programs to fill the places of arrays when the size is unknown and intermediate insertions and deletions are necessary. They are generally implemented as arrays whose data is shifted around when necessary and copied to a larger array when extra space is needed. Tradition dynamic arrays have O(1) random access, O(N) insertion (except to the end where it is amortized O(1)) and O(N) deletion (except from the end where it is amortized O(1)).

Red-Black Trees have been extended to allow for logarithmic access of the nth element, however although this is similar to random access it is not identical. Random access uses sequential integral keys and will change the keys of the other elements as elements are added or removed.

## Scope

I intend to both design and implement a fully functioning data structure. I will have it implemented in both C++ and Java and apply it to different possible problems. Also, the theory will be fully worked through and the efficiency proven.

## Type of Research

My project falls under pure basic research. I am devising an algorithm to fundamentally expand the knowledge we have of data structures and algorithmic efficiency.

## Procedure and Methodology

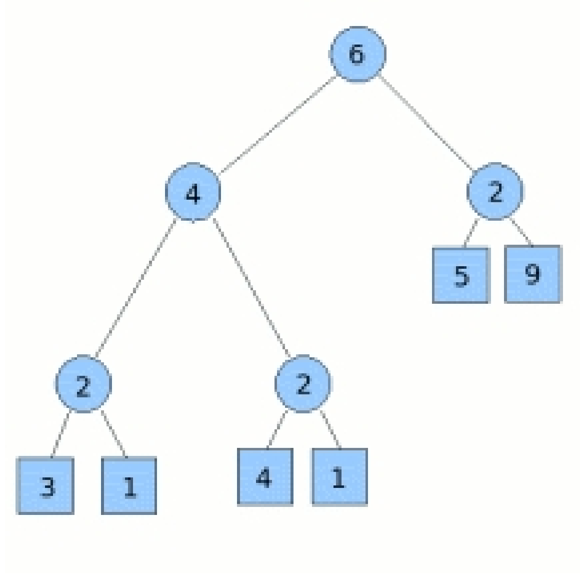
The primary work in devising an algorithm is conceptual, however putting it into code greatly helps in fleshing out the ideas as well as identifying flaws in the process. Similarly, when put into code the efficiency of the general case can be tested empirically.

The various attempts at randomly accessible trees are implemented as classes in Java, and a separate Java program is used for testing the classes. The testing program allows for both randomly inserting and deleting data as well as sequential insertion and deletion so both random and extreme cases are testable.

The original procedure was to make a randomly accessible tree and then balance it but the focus was later shifted to modifying an already balanced tree to allow for random access. First this was done upon a Red Black Tree, but due to a series of setbacks this work was postponed in favor of work on a Splay Tree.

## Results

Currently, I have made both a randomly accessible data structure that is guaranteed balanced and logarithmic in the general case and have worked on a standard Red Black Tree which logarithmic for insertion, deletion and lookup but is not randomly accessible at all. As I reached a snag while working on the Red Black Tree I have begun working instead on the much simpler Splay Tree, which I will shortly work to make randomly accessible.



## Value

This data structure will be usable in a wide variety of applications. Just like any other data structure it will be potentially usable in any program. For example, say one wanted to put a bunch of values into a data structure and then later remove then randomly from it. If one used a linked list or a dynamic array the adding of elements would be $O(1)$ per element however the random removal would be $O(N)$ per element. If one used a binary search tree, they would have to keep a list of the keys and would have the original problem once again with this list. The data structure described here however, would be easily useable and would run in $O(log(N))$ per element for each step. If it is found to be usable enough, it may even find its way into standard libraries like dynamic arrays have.

## References

[1] Dúnlaing, C. . (2003, October). Inorder traversal of splay trees. Electronic Notes in Theoretical Computer Science, 74, 1-24. Retrieved January 2, 2007, from Science Direct database.

[2] Park, H., & Park, K. (n.d.). Parallel algorithms for redblack trees. Theoretical Computer Science, 262(1-2), 415-435. Retrieved January 2, 2007, from Science Direct database.

[3] Tarjan, R. E. (1983, June 10). Updating a balanced search tree in O(1) rotations. Information Processing Letters, 16(5), 253-257. Retrieved January 2, 2007, from Science Direct database.