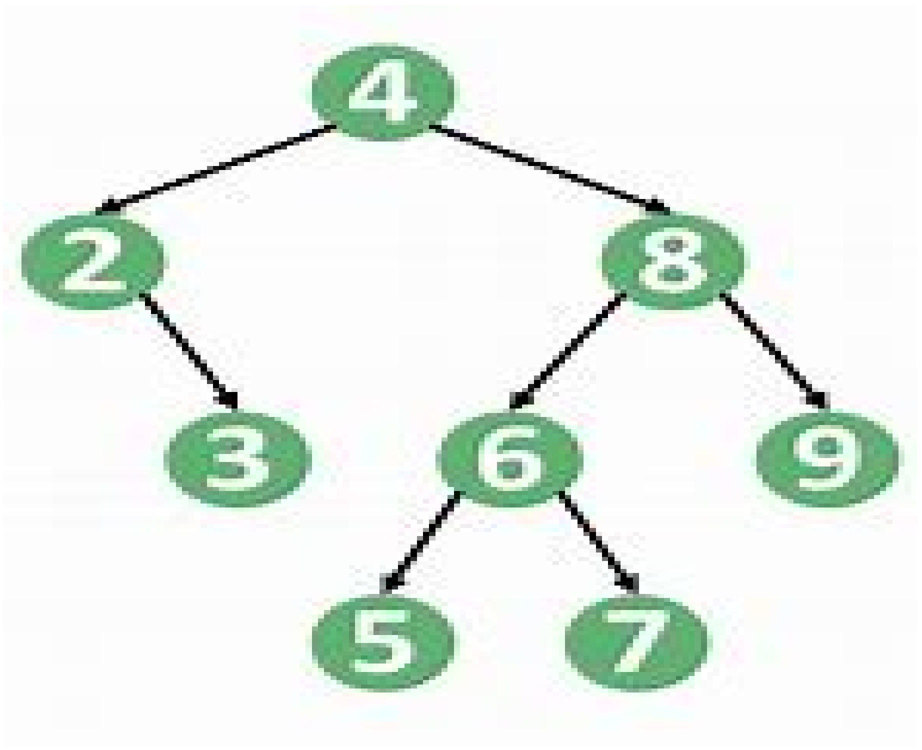
CSC 212: Data Structures and Abstractions

Left-Leaning Red-Black BSTs



Team Members:

Matthew Langton, Jackson Rich, Joey Koumijan

University of Rhode Island

Final Project Report

Fall 2022

0 Table of Contents:

1 Introduction:

Our team was tasked with creating an implementation of a left-leaning red-black BST using C++. Our main objectives were to implement the data structure in a simple effective way so that it may be easily understood by others, and then to create a presentation relating to the data structure that we programmed. We applied principles and knowledge learned from CSC 212 and elsewhere such as:

- Usage of support tools such as Zoom and Github to collaborate and work on the project as a team.

- Usage of C++ as a programming language to implement our solutions.

- Detailed analysis of the data structure using principles such as run time and algorithm performance under different conditions.

Using these ideas, our team has designed a fully operational left-leaning red-black BST algorithm which fulfills all necessary functions that this data structure needs to fulfill. The main goal of our project is to implement a working, well-documented left-leaning red-black BST using C++. The purpose of doing so is to use it as a teaching tool for others to understand the data structure better. This purpose coincides with that of our presentation.

Building off of this, this report will dive deeper into the structure and design of the data structure in question, giving a more detailed explanation to how the structure functions. We will also discuss our implementation of the data structure along with our reasoning for implementing it in this way and our findings from running various tests on our algorithm. The report will then be wrapped up with a conclusion of our findings and a list of contributions from all members.

2 Left-Leaning Red-Black BSTs: Methods and Explanation

A left-leaning red-black BST is a type of binary search tree designed to be both self-balancing, and easier to implement than a standard binary search tree.

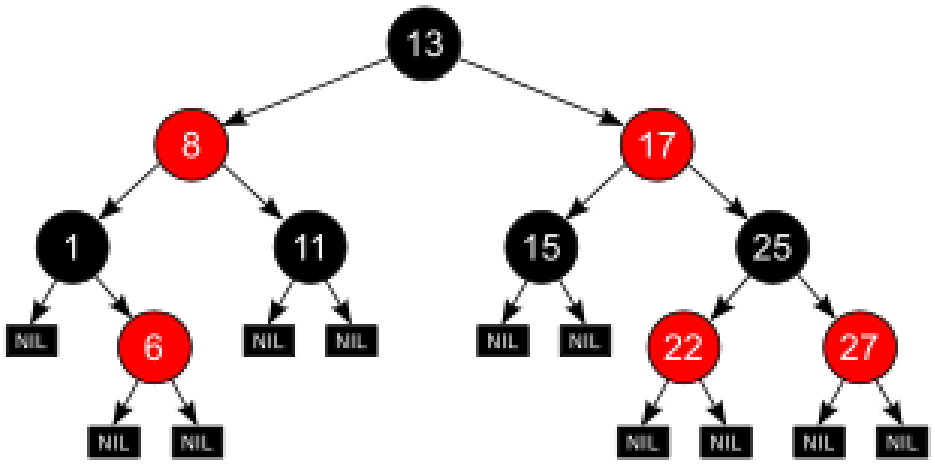


Figure 1: An example of a left-leaning red-black BST with an example data set.

The figure above shows an example of a left-leaning red-black BST.

For a bit of history, in 1972 Rudolf Bayer invented a data structure that was a special order-4 case of a B-tree. These trees maintained all paths from root to leaf with the same number of nodes, creating perfectly balanced trees. However, they were not binary search trees. Bayer called them a "symmetric binary B-tree" in his paper. Later, these trees became popular as 2-3-4 trees or just 2-4 trees. This was then further developed on by Leonidas J. Guibas and Robert Sedgewick in their 1978 paper titled “A Dichromatic Framework for Balanced Trees” where they derived a red-black search tree. Then in 1993, Arne Andersson introduced the idea of a right leaning tree to simplify insert and delete operations and in 1999, Chris Okasaki showed how to make the insert operation purely functional. Its balance function needed to take care of only 4 unbalanced cases and one default balanced case. Then finally, In 2008, Sedgewick proposed the left-leaning red-black tree, leveraging Andersson’s idea that simplified the insert and delete operations.

Left-leaning red-black search trees are simple in nature and similar to other types of search tree algorithms. They follow all of the standard requirements of a usual binary search tree, but with some added conditions:

- All nodes are either red or black.
- All “leaf nodes” and nonexistent nodes in the tree are considered black.
- A red node does not ever have a red child node.
- Every path from a given node to any of its descendant leaf nodes goes through the same number of black nodes.
- The root node of the tree is always black.

Red-black trees allow quite efficient sequential access of their elements. But they also support asymptotically optimal direct access via a traversal from root to leaf, resulting in O(log n) search time.

2.1 Operations on Left-Leaning Red-Black BSTs

Left-leaning red-black BSTs have 3 main operations which can be performed on them, search which will look for an element in the tree to see if it exists, insert which will insert an element into the tree and will reorganize the other existing elements properly, and delete which removes an element from the tree and reorganizes the other existing elements properly.

Operations on left-leaning red-black BSTs are all very similar to the same operations done on regular BSTs, with the main difference being that after an operation is done, the tree must be restructured so that the alternating red-black nature of the tree is maintained.

These operations also make usage of four different rotation cases, left-left, left-right, right-right, and right-left. A left-left rotation is where a single right rotation is performed, a right-right rotation is where a single left rotation is performed, a left-right rotation is where a left rotation is performed, then a right rotation, and a right-left rotation is where a right rotation is performed, then a left rotation.

The time complexity of each operation for the structure is listed below:

Function	Best Case	Worst Case
Search	O(log(n))	O(log(n))
Insert	O(1)	O(log(n))
Delete	O(1)	O(log(n))

Terminology for a tree in graph theory borrows terms from a family tree. A parent node refers to the node located directly above the current node, a child node refers to the node directly below the current node, a grandparent node refers to the node's parent's parent, a sibling node refers to a node's parent's other children, and an uncle node refers to a node's parent node's sibling node.

Insertion:

Insertion of a node into a left-leaning red-black BST is very similar to inserting a node into a standard BST, with the main exception being that after insertion checks must be made to ensure that the rules for the data structure are upheld. These are the steps for inserting an element and transferring it into a node "X" into a tree would be as follows:

1. Perform a standard BST insertion and mark the new node as red.

2. If X is the root, change the color of X to black.

3. If X’s parent node is not black and X is not the root:
    a. If X’s uncle node is red:
        i. Change the color of X’s parent and uncle nodes to black.
        ii. Change the color of X’s grandparent node to red.
        iii. Change X to X’s grandparent node and repeat steps 2 and 3 for the new X.
    b. If X’s uncle node is black:
        i. Left-Left Case: Perform a right swap of the grandfather node then swap the colors of the grandparent and parent node.
        ii. Left-Right Case: Perform a left rotation of the parent node then apply the left-left rotation case.
        iii. Right-Right Case: Perform a left rotation of the grandfather node and then swap the colors of the grandparent and parent node.
        iv. Right-Left Case: Perform a right rotation of the parent node then apply the right-right rotation case.

Deletion:

Deletion, like insertion, follows a similar set of steps to both insertion and deletion in standard BSTs with the main difference between the two being that after the delete is performed the red-black nature of the tree must be restored. A list of steps for deleting a node “X” is given below:

1. Perform a standard BST delete.

2. If either X or its parent node is red, change its color to black.

3. If both X and its parent node are black:

    a. Mark X as double black. (A double black node is a node which corresponds to an underflow node in a 2-3-4 tree.)

    b. If X's sibling node is black and at least one of the sibling's children is red, perform rotations. Let the red child of the sibling node be r. This case is divided into the same 4 subcases as insertion: left-left, left-right, right-right, and right-left.

    c. If the sibling node is black and both child nodes are black, perform recoloring, and recur for the parent node if the parent is black.

    d. If the sibling node is red, perform a rotation to move the old sibling node up, then recolor the old sibling and parent. The new sibling is always black. This case can be divided into two subcases: left-left and right-left.
4. If X is the root, make X a single black node, then return.

As a side note, LLRBBST deletion code was impossible to find or make. Professors pointed to Sedgewick's paper, which did have deletion code in Java. However, there were undeclared identifiers in the code that weren't explained in English. Jackson translated this code to C++ and attempted to make it work anyway, but couldn't. Our professors simply pointed to Sedgewick's paper. Online resources such as Stack Overflow, GeeksForGeeks, Wikipedia, etc. only had deletion methods for regular red-black BSTs. Whether or not LLRBBST deletion is the same as red-black BST deletion the internet didn't shed any light on, and on that question we remain unsure.

Search:

Searching for a given key in a left-leaning red-black BST is no different to how one would search for a given key in a standard BST, given that the tree is structured in a very similar manner. The searching method is given as follows:

1. Start with the root and compare the key being searched to the root value.

2. If the value is equal, the search is complete.

3. If the value is smaller than the root, we search the left section of the tree, and vice versa if the value is larger than the root.

4. Step 3 is repeated in subsequent nodes until the correct value is found and returned, or all nodes in the tree have been scanned in which case the value will be reported as not found.

2.2 Usage of Left-Leaning Red-Black BSTs

One of the biggest benefits of this type of BST is that it offers a constant worst-case complexity for operations performed on them. This makes them invaluable for time-sensitive applications of the data structure. The structure is also excellent to pair with other data structures which offer constant worst-case complexity as a building block. For example, many data structures used in computational geometry can be based on red-black trees, and the Completely Fair Scheduler used in current Linux kernels and epoll system call implementation uses red-black trees.

Red-black BSTs are also valuable in functional programming where they are very commonly used to create associative arrays and sets which can retain their previous versions after being changed.

In Java 8 and subsequent versions, HashMaps utilize red-black BSTs replacing the old iteration which used linked lists. This has resulted in an improvement of time complexity to O(log(n)).

3 Implementation of the Data Structure

Our team created this application of the data structure with the main goal in mind being simple understanding and education. A left-leaning red-black BST can be complicated to get a firm grasp on, and understanding of a data structure such as this in the computer science field is highly important. This is why our implementation of the structure has been aimed at being primarily a learning tool. The program's goal is to provide the user with a simple, clean interface which runs through all major functions which are typically used in a left-leaning red-black BST so they may be visualized and understood better.

We decided the best way to go about doing this is to create a working iteration of the data structure which visualizes what it's doing as it runs. The program will present the user with a list of options for the different functions (insert, delete, and search) and perform the action on a value given by the user whilst also utilizing animated visual aids showing the user what is going on as it runs.

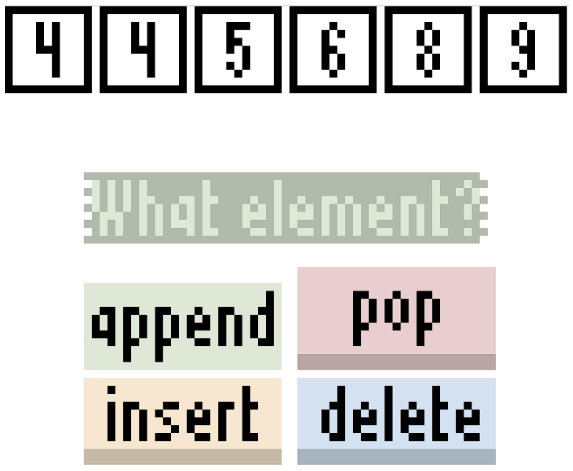


Figure 2: An image of our program's main menu with options for available data structure functions.

Once the user loads up the program they will be met with a set of options to choose from, as seen in the figure above. Choosing one of the available options will cause the program to do the specified action on the value given and play an animation to show what the action is doing to the BST internally, as shown in the image below.

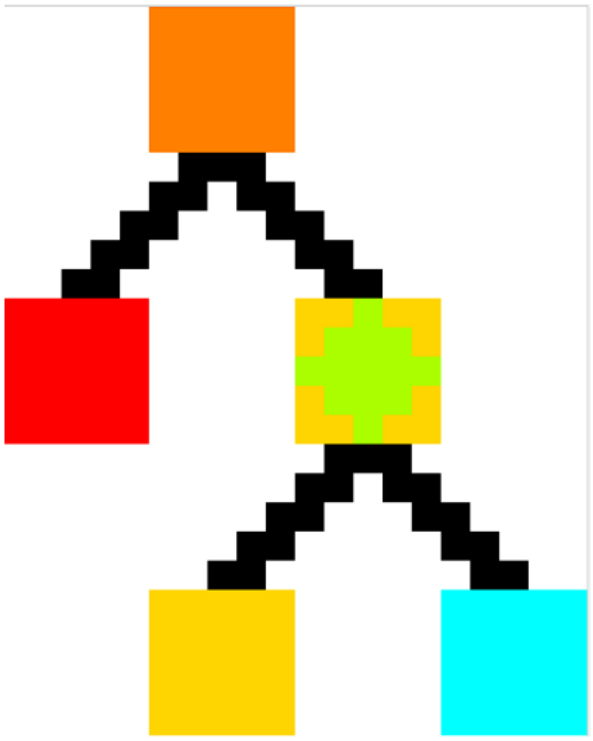
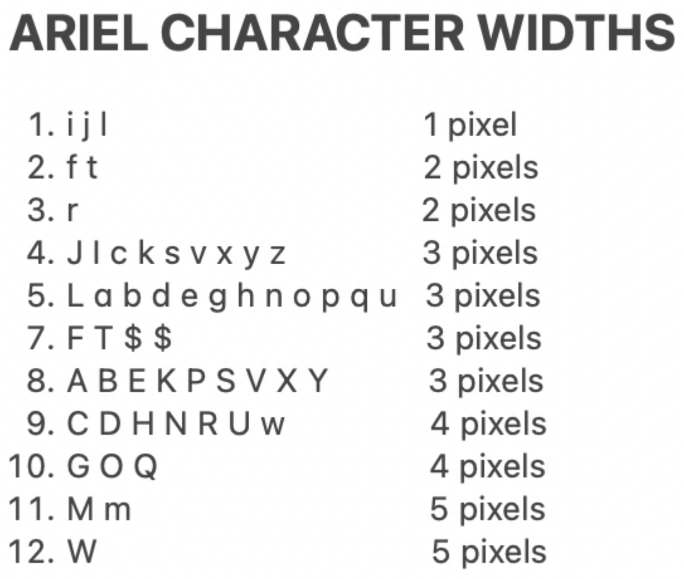


Figure 3: An image of one of our visual aid animations playing as the algorithm runs.

Once the program has been run and the algorithm has done its job, the action taken onto the tree will have been done, as this is a functioning left-leaning red-black BST, and the program will redirect the user to the main menu. The importance of making sure the data structure was implemented in a completely functional manner was of high importance as correct implementation of the data structure was considered important for the sake of creating a tool for learning about how the given data structure works.



ARIEL CHARACTER WIDTHS

1. i j l	1 pixel
2. f t	2 pixels
3. r	2 pixels
4. J I c k s v x y z	3 pixels
5. L a b d e g h n o p q u	3 pixels
7. F T $ $	3 pixels
8. A B E K P S V X Y	3 pixels
9. C D H N R U w	4 pixels
10. G O Q	4 pixels
11. M m	5 pixels
12. W	5 pixels

Figure 4: Chart of widths of different characters in the Ariel font, and the pixel widths of their pixel-art counterparts. This was used in the making of a custom pixel-art font to use for text fields in interactive diagrams.

Our program also utilized text fields in the different slides to display instructions and results to the user. Text was created using different fonts imported using the setFont class and displayed in set text fields on each displayed slide.

3.1 Background Program Statistics - Number of Nodes

Aside from all of the reasons behind our implementation of the data structure, we also made sure to keep track of what was going on as the program was running, not only for data collection, but for being sure that everything was working correctly.

Below is a graph showcasing the difference between Wikipedia's height formula and the actual data. Our findings indicated that the average number of nodes in the tree (assuming x elements) was given by the formula $\log_2(x) * 16/11 - 1$. This differs from the formula of $2 \ln(x)$ given by the creator of the data structure, Robert Sedgewick, and from the Wikipedia formula of $2\log_2(x)$. We believe that this difference is due to a higher degree of precision we pursued that may have been lost in the ambiguity of complexity calculations.

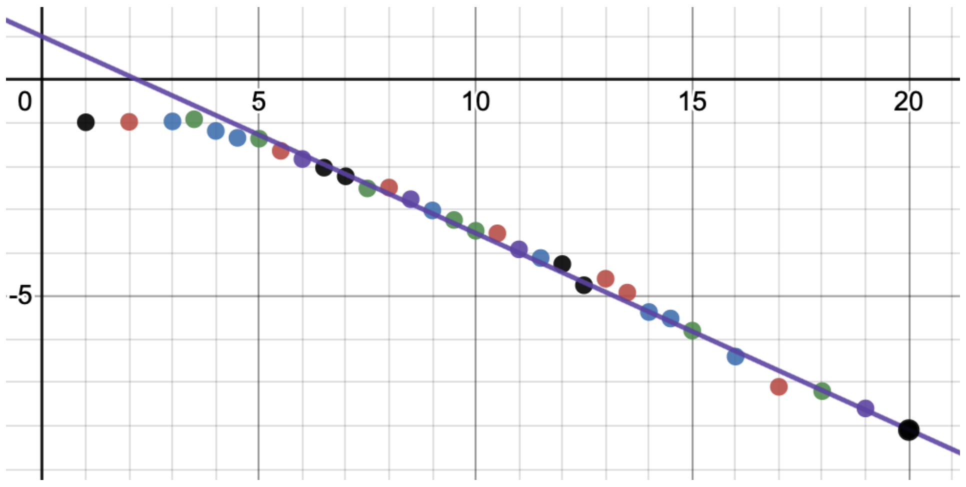


Figure 4: A graph where n = the number of nodes in a LLRBBST, x = log$_2$(n) and y = the difference between the average tree height and log$_2$(n).

Another graph pictured below shows the average values of the number of nodes (also assuming x elements) over a larger scale. This graph shows the average value as x → infinity. The average value shown in the graph appears to be an oscillating logarithmic function where the oscillations begin to stretch more and more the closer x gets to infinity.

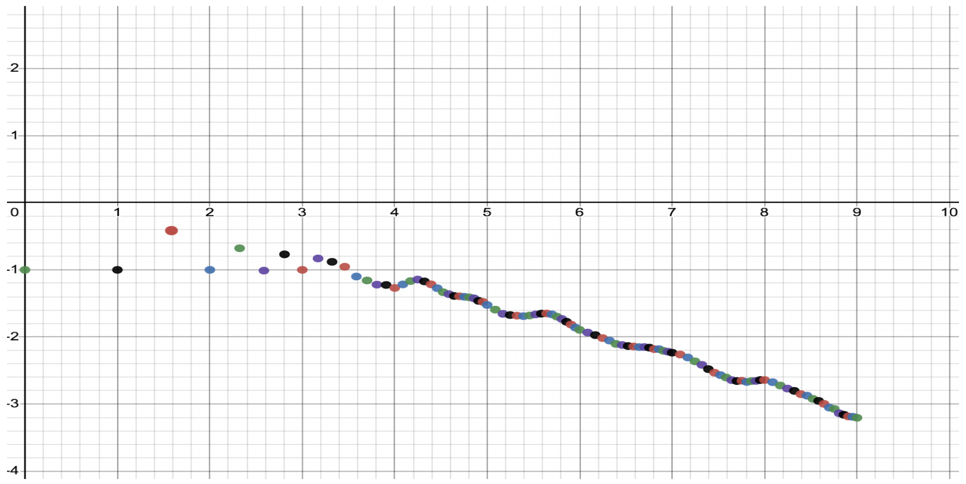


Figure 5: This graph displays the same graph shown in figure 4 on a logarithmic scale.

4 Contributions

Over the several weeks given and used for working on this project, all team members ensured their part was done. Regular group meetings were held to discuss, work on, and critique our own and each other's work to ensure that it was done accurately, effectively, and of a high quality. Research was done to ensure that all group members fully understood the data structure being implemented and a group effort was made to ensure that this was the case. Furthermore, additional, specific contributions were made by group members:

Matthew Langton

- Responsible for main report creation.
- Tested pieces of code to ensure they ran properly.
- Did research on the data structure for both project, presentation, and coding purposes.
- Assisted with ideas and proofreading presentation slides.
- Wrote code responsible for timing slides correctly to ensure that they functioned correctly.

Joey Koumijan

- Wrote code responsible for visual aid implementation into the program.
- Assisted with creation of animations used in code and the presentation.
- Assisted with proofreading both presentation slides and the report.
- Utilized CLion tools to create visuals in the program.

Jackson Rich

- Responsible for main presentation creation.
- Created majority animations for use in code and presentation.
- Assisted with proofreading report.
- Wrote code responsible for main implementation of the data structure.

5 Works Cited

"Red–Black Tree." *Wikipedia*, Wikimedia Foundation, 19 Nov. 2022, https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#:~:text=%20In%202008%2C%20Sedgewick%20proposed%20the%20left-leaning%20red%E2%80%93black,added%2C%20making%20new%20trees%20more%20like%202%E2%80%933%20trees.

"Searching and Inserting with Red-Black Trees." *Searching and Inserting with Red-Black Trees*, http://math.oxford.emory.edu/site/cs171/redBlackTreeSearchingAndInserting/.

"Red-Black Tree: Visual Algorithms and Data Structures." *Codetube.vn*, 2020, https://codetube.vn/visual/redblacktree/.

Soodan, Ashpreet. "Left Leaning Red Black Tree (Insertion)." *GeeksforGeeks*, 21 July 2022, https://www.geeksforgeeks.org/left-leaning-red-black-tree-insertion/.

"Red-Black Tree: Set 3 (Delete)." *GeeksforGeeks*, 23 Dec. 2021, https://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/.