**Splay Trees**

Cameron Castillo | Daniel Diaz | Vincent Zhuang | Carl Kakisis

Author | Co-author | Co-author | Co-author

1. Introduction

The purpose and intent of this project was to implement and display a working data structure not covered in course lectures. For this project, our team chose to implement a basic version of a Splay Tree and provide a visualization of the implemented data structure using **SFML**, a multi-language **OpenGL** wrapper used for graphics processing and rendering. Although the project guidelines required a visualization in the form of an exported **DOT** file, our team decided to create the visualization using SFML to allow for active user input in the form of searches, insertions, and deletions. This document is broken down into sections structured similarly to the accompanying PowerPoint presentation, and the sections to be covered, excluding this introduction, are as follows:

1. ***Defining a Splay Tree***
2. ***Balancing a Splay Tree***
3. ***Time complexities & Use cases***
4. ***Implementation***
5. ***Contributions***
6. ***Index***
7. ***Sources***

In order to define a Splay Tree, we will cover the necessary auxiliary data structures used to create the tree, as well as some of its defining properties. Directly following and relating to the definition of the data structure, as a Splay Tree is self-balancing, the types of balancing algorithms used will also be discussed. As follow-ups to these defining topics, this document will cover this data structures’ advantages and disadvantages, as well as examples of real-world uses. Our teams’ implementation of the data structure, as well as the methods and algorithms used to create it, will be the final topic of discussion. Any terms used in this document that may not have immediately apparent definitions will be listed in section 7, Index, and the first appearance of these terms will be bold-faced to indicate that they also appear in the index.

1. Defining a Splay Tree

At a baseline, a Splay Tree is a form of a **binary search tree** in which the tree follows strict **invariants** that allow for accelerated searches by key value. A tree is a data structure composed of nodes and edges that does not allow for parallel edges, meaning that two nodes cannot point to each other with two unique edges, and does not allow for loops on tree nodes, meaning a node cannot point to itself. This structure is expanded upon when implemented as a binary search tree.

Shape

Description automatically generated with medium confidence

Figure 1: Binary Search Tree

As in *Figure 1* to the right, all node key values to the left of the root of a binary search tree are less than the key value at the root, and all node key values to the right of the root are greater than the key value at the root. In the sense that each node in the tree also comprises a subtree of the overall tree, these two invariants apply to internal nodes as well.

Splay Trees expand upon the invariants and structure of a binary search tree to allow for instant access to the most recently searched-for or inserted key value. This is achieved through frequent rebalancing of the tree structure when any operation takes place on the tree, including deletions. The rebalancing methods used in this data structure also preserve the invariants of the binary search tree and can be broken down individually. These balancing methods cause operations performed on a specific node to “bubble” the node up to the top of the tree and make it the root, which is also conducive to reducing the computational cost of operating on nodes near the root as well. The methods are in many ways the crux of the benefit of using this data structure.

1. Balancing a Splay Tree

Balancing a Splay Tree when performing operations on it primarily requires the use of the aforementioned balancing methods, which will be discussed fully in this section. The first of these balancing methods are the Simple Rotations, which refer to both a “zig” and a “zag” rotation.

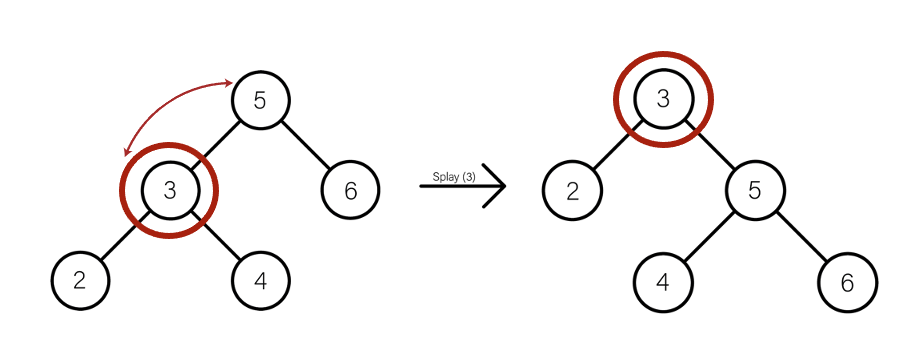
A zig rotation, in its simplest terms, is simply a right rotation from a tree node. In a zig rotation, every node in the tree moves one position to the right from its current position. In Figure 2, when splaying the key value 3, the entire tree is rotated to the right to move the target key value to the root. As one can see, the invariants of a binary search tree are not violated in this operation.

Figure 2: Zig Rotation

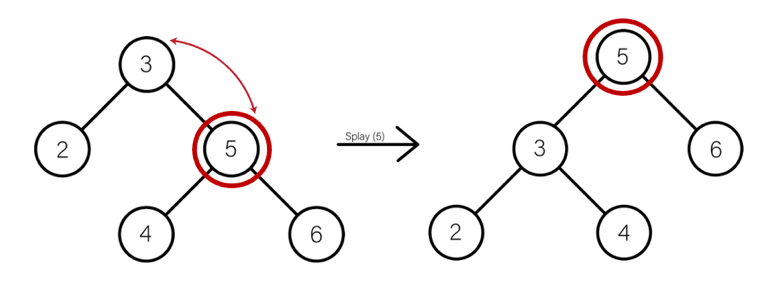
A zag rotation is the opposite of a zig rotation in terms of its rotation direction. In a zag rotation, every node in the tree moves one position to the left from its current position. In Figure 3, when splaying the key value 5, the entire tree is rotated to the left to move the target key value to the root. Once again, the invariants of a binary search tree are not disobeyed by this operation.

Figure 3: Zag Rotation

These basic balancing methods are expounded upon into more complex forms that involve multiple rotations in order to rebalance the tree and prevent its invariants from being violated. Despite that, these additional forms of balancing rely on these simple rotations, so balancing always takes the form of sequential left or right rotations on a node.

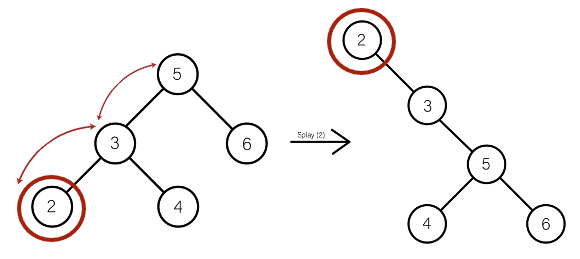
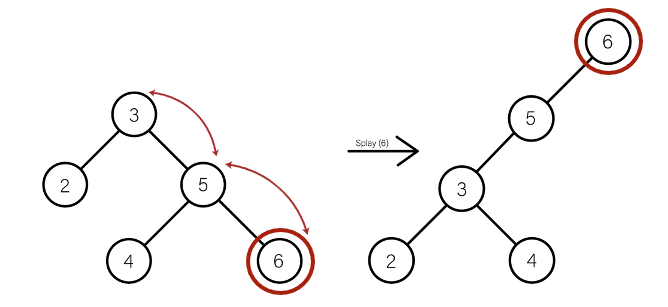


Figure 4: Zig-zig Rotation

The first of these sequential rotations is the “zig-zig” rotation. As its name suggests, and with the prior definition of zig in mind, a zig-zig rotation is simply a double-right rotation. In this method, every node in the tree moves two positions to the right from its current position. In Figure 4, when splaying the key value 2, the entire tree is rotated two positions to the right to move the target key value to the root.

The next sequential rotation is, as expected, the “zag-zag” rotation, which involves a double-left rotation. In this method, every node in the tree moves two positions to the left from its current position. In Figure 5, when splaying the key value 6, the entire tree is rotated two positions to the left to move the target key to the root.

The zag-zag and zig-zig rotations address balancing issues when a node is the left child of a node that is the left child of another node, and the same for a right child, respectively. With these rotations, however, there are still cases to be addressed in terms of rebalancing the tree. These additional cases are solved using the algorithms below:

Figure 5: Zag-zag Rotation

The first multi-direction rotation is a “zig-zag” rotation, which involves a right rotation followed by a left rotation. In this balancing method, each node is moved one position to the right on the zig step, and then one position to the left on the zag step, from its current position. In Figure 6a, when splaying the key value 4, the entire tree is initially moved one position to the right. Since the target node is not yet the root, this is followed-up by moving the entire tree one position to the left in Figure 6b, effectively moving the target node to the root, and preserving the balance of the tree.



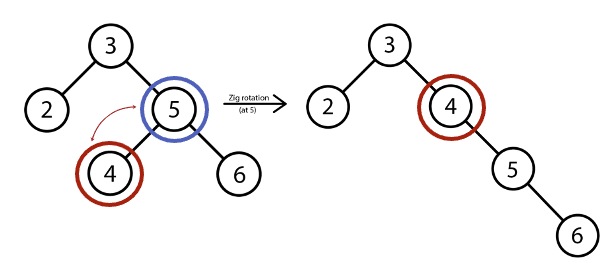
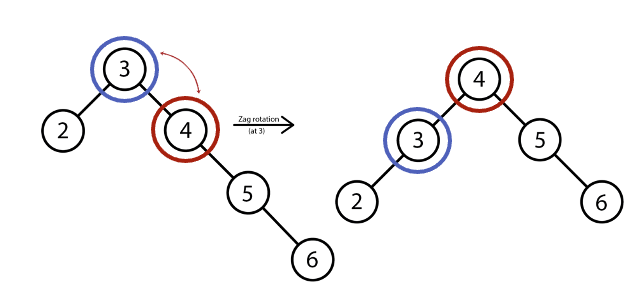


Figure 6b: Zig-zag (Zag step)



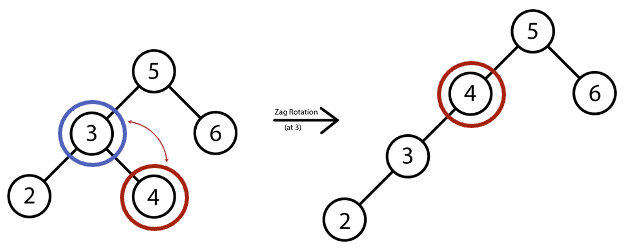
Figure 7a: Zig-zag (Zig step)

A picture containing watch, clipart

Description automatically generatedThe final rotation is a “zag-zig” rotation, which involves a left rotation followed by a right rotation. In this method, each node is moved one position to the left on the zag step, and then one position to the right on the zig step, from its current position. In Figure 7a, when splaying the key value 4, every node in the tree is initially moved one position to the left. Since the target node is not yet the root, this is followed-up by moving every node in the tree one position to the right in Figure 7b, which moves the target node to the root, and keeps the balance of the tree.

Figure 7b: Zag-zig (Zig step)

Figure 7a: Zag-zig (Zag step)



These balancing methods are essential in not only keeping the invariants of a binary search tree intact for this data structure but are also important in the “splaying” step of tree operations, which is unique to this data structure. Without these methods, it would be impossible to bubble up nodes to the root and would also result in a highly imbalanced tree that may violate its inherent qualities. However, that is not to say that this data structure is always perfectly balanced, which is a topic to be explored in the next section.

1. Time complexities & Use cases

Chart

Description automatically generatedAs with many data structures including trees, the efficiency of the data structure is the primary reason why it is used. The data structure itself is well and good on its own, but if it does not have any reasonable uses, then there does not seem to be a reason to implement it at all. The Splay Tree benefits from an amortized Θ(log n) average and O(log n) worst-case performance with operations such as searches, insertions, and deletions[1]. Figure 8 below provides the growth rate of data structures and algorithms based on their time complexity:

Figure 8: Big O Time Complexities (https://www.bigocheatsheet.com/)

As one can see, although the number of rotations involved in rebalancing a Splay Tree may seem

particularly computationally expensive, its operations sit reasonably below the linear time cost, O(n).

In some cases, a Splay Tree may end up leaning very heavily to the right or left depending on the nature of the operations performed on it as it does not rely on internal node data for balancing like red-black trees do, among other examples. In cases such as these, the tree may end up looking like a linked list in a visualization, and the worst-case height of these types of trees will be O(n), where n is the number of nodes in the tree. Overall, this is the main disadvantage of this data structure, as it will cause the real-time cost of accessing nodes far from the node to be increasingly expensive. Despite its amortized cost, in general and more so in a poorly balanced splay tree, individual operations on a tree can end up being rather computationally expensive, which is a main drawback for its implementation in real-time applications.

As mentioned previously, Splay Trees are a particularly useful data structure when it is known that the same nodes will be accessed multiple times in a short period of time. This also extends to node accesses near the root, which can be retrieved and bubbled to the top of the tree very quickly due to their location. Since Splay Trees rely solely on rotations to preserve balance, they do not rely on internal balance data. As such, it could be argued that they are easier to implement than red-black trees, which not only require internal balance data, but that data is frequently altered and flipped.

Despite its drawbacks, Splay Trees do have real-world applications. For example, a Splay tree might be used in the creation of a cache, where a cache keeps track of the contents of the memory locations that were recently requested by a processor [2]. Like a binary search tree, the main intent of implementing a Splay Tree is to quicken node accesses and data retrieval, while also reducing the amount of memory used to implement a storage system like this.

1. Implementation

Our implementation of a Splay Tree utilizes all of the prior balancing and construction methods mentioned earlier to create a working back-end data structure, as well as a visualization of the structure. Since the structure of many trees, including Splay Trees, lends itself to recursion, our team has created a variety of **private** tree functions that are utilized with all operations performed on the tree. A table of all of the private functions used is below, with their uses:

| **Function Name** | **(Parameters) / Return Type** | **Description** |
| --- | --- | --- |
| Insert | (*int* data, *Node\** root) / *Node\** | The insert function is the main driver function for inserting new data into the tree. We have included cases for when the tree does not have a root (construct a new root node), and when the data being inserted already exists (increment a counter). |
| Left Rotate | (*Node\** root) / *Node\** | The left rotate function is a left rotation, or a “zag” rotation in its simplest form. As more complex rotations involve sequential simple rotations, this function can be called within other recursive functions to perform other rotations. |
| Right Rotate | (*Node\** root) / *Node\** | The right rotate function is a right rotation, or a “zig” rotation in its simplest form. This function can be called, like the left rotate function, in other recursive functions to perform complex rotations. |
| Splay Tree | (*int* data, *Node\** root) / *Node\** | This is the primary function that will execute a search on the tree and returns a pointer to the found node if it exists. The function has two branching paths that are called recursively for the left and right sides of the tree. As each node in the tree also comprises a subtree, the branch traversal will follow the same rules as if the user was searching for the left or right child of the existing root. |
| Height | (*Node\** root) / *int* | This function is used for retrieving the overall height of the tree, which is the number of edges on the longest path from the root to the furthest **leaf** node. The function recursively traverses the left and right branches of the tree and compares the height of both branches. |
| InOrder, PreOrder, PostOrder, LevelOrder | (*Node\** root) / *void* | These functions are purely to display the contents of the tree in a particular order. |
| Destroy | (*Node\** root) / *void* | This function will recursively destroy an existing Splay Tree, and also free the memory occupied by the tree as the recursive call stack resolves. |
| Delete Node | (*int* data, *Node\** root) / *Node\** | This function is used to delete specific nodes by their value. The function will traverse the tree by utilizing the Splay Tree function above to search for the node containing the target data, delete the node, and then rebalance the tree to maintain its invariants. |

*(Declarations: “SplayTree.h”, Definitions: “SplayTree.cpp”)*

One might notice that all of these private functions accept a pointer to a node as one or all of their parameters. This is entirely intentional, as the actual utilization of these functions is done using **public** methods. This allows users to make use of the private functions and data members without actually directly interfacing with them. Our team believes that this separation of functions was also necessary to make sure that when using private functions, the functions are guaranteed to start at the root node of the tree, whether it already exists. The separation also has the added benefit of preventing bugs and oddities in cases where a user might try to begin an operation at a node other than the root.

Since our team has decided to provide a visualization in the form of a viewable SFML program, it is pertinent to discuss the functions used to create that visualization as well. The visualization is preceded by a menu wrapper that is defined in the “Menu.cpp” file. In the compiled definition file, “Splay.cpp,” we have implemented an intentionally large function that is the driver function for creating a window to display the created tree. The function is broken down primarily into keyboard and mouse events, all of which are listed below:

| **Event** | **Description** |
| --- | --- |
| Key: *I* | This event handles user input for inserting a new node into the tree via the terminal. The event can handle either individual entries, or a string of entries separated by white space. There are safeguards in place to make sure that the program does not crash if the function is attempting to draw a node outside of the viewable window. |
| Key: *S* | This event allows the user to search for a specific node in the tree. The search function will highlight the nodes it is examining as it traverses the tree. Once the node has been found, or if the node does not exist in the tree, the tree will be rebalanced with the found node or closest node to the searched value at the root. |
| Key: *N* | This event allows the user to incrementally zoom in to the viewable window. It is particularly useful for examining individual node values in the tree, and for viewing selected subtrees in the tree. |
| Key: *M* | This event allows the user to incrementally zoom out from the viewable window. It is particularly useful for trees with many nodes, or for trees with nodes that branch a considerable distance in the cardinal directions from the root. |
| Key: *Down, Up, Left, Right* | These four key events allow the user to “scroll” around the viewable window in the four cardinal directions. Tree node positions and sizes are preserved when executing these events. |
| Key: *J*  + Mouse: *Left Click* | This event allows the user to delete individual nodes. Toggles exist that are utilized in this function to make sure that the intended operation is performed. |

1. Contributions

All team members contributed to the creation of this data structure in a variety of different ways. Larger chunks of this project are broken down into individual contributions below.

1. **Back-end Implementation**
   1. Back-end implementation was completed primarily by ***Daniel Diaz*** and ***Vincent Zhuang.***
   2. Individual smaller contributions were made by ***Cameron Castillo*** for code reorganization, debugging, and commenting/documentation.
2. **Front-end Implementation (SFML)**
   1. Front-end implementation was completed primarily by ***Vincent Zhuang*** and ***Cameron Castillo***. This includes declaration and definition of front-end functions and data members, back-end implementation of the visualization of the tree (since the front-end relies on the recursive back-end function), debugging, and documentation.
   2. The menu wrapper that precedes the visualization of the tree was created solely by ***Daniel Diaz***.
3. **Presentation**
   1. The presentation in the form of a PowerPoint was created primarily by ***Carl Kakisis*** and co-authored by ***Cameron Castillo***.
   2. Information retrieval/source examination and slide authoring were performed by ***Carl Kakisis***.
   3. Creation of visualization assets for slides was completed by ***Cameron Castillo***. Some assets were reused multiple times but can be found in the “SPLAY\_ASSETS.7z” archive in our team’s GitHub repository.
4. **Project Report**
   1. The project report was authored by ***Cameron Castillo***. This includes source examination, implementation of figures and visualizations, and code breakdown.
   2. Critiques and changes were contributed by all other members of the team.

Records of all contributions are available in our team’s GitHub repository, located [here](https://github.com/danielediazp/Splay-tree). A direct link to the history of repository commits is available [here](https://github.com/danielediazp/Splay-tree/commits/main).

1. Index

| **Term** | **Definition** |
| --- | --- |
| SFML | Simple and Fast Multimedia Library (SFML) is a cross-platform software development library designed to provide a simple application programming interface to various multimedia components in computers.[3] |
| OpenGL | OpenGL is a cross-language, cross-platform application programming interface for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit, to achieve hardware-accelerated rendering.[4] |
| DOT | DOT is a graph description language used to represent directed and undirected graphs.[5] |
| Binary Search Tree | In computer science, a binary search tree, also called an ordered or sorted binary tree, is a rooted binary tree data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree.[6] |
| Invariant | In mathematics, an invariant is a property of a mathematical object which remains unchanged after operations or transformations of a certain type are applied to the objects.[7] |
| Child | A node extending from another node which is its parent node.[8] |
| Amortized | In computer science, amortized analysis is a method for analyzing a given algorithm's complexity, or how much of a resource, especially time or memory, it takes to execute. The motivation for amortized analysis is that looking at the worst-case run time can be too pessimistic.[9] |
| Red-black Tree | In computer science, a red–black tree is a kind of self-balancing binary search tree. Each node stores an extra bit representing "color" ("red" or "black"), used to ensure that the tree remains balanced during insertions and deletions.[10] |
| Private | Private is a keyword that specifies access level and provides programmers with some control over which variables and methods are hidden in a class. Variables and methods defined with the private keyword may be accessed only by other methods within the class and cannot be accessed by derived classes.[11] |
| Public | In object-oriented programming, a public class is any part of the program that accesses or updates its members using the member name to be accessed. This type of class is useful for information or methods that need to be available in any part of the program code.[12] |

1. Sources

Anand, “Splay tree,” astikanand.github.io. [Online]. Available: https://astikanand.github.io/techblogs/advanced-data-structures/splay-tree. [Accessed: 22-Jul-2022].

[9] “Amortized analysis,” Wikipedia, 24-Apr-2022. [Online]. Available: https://en.wikipedia.org/wiki/Amortized\_analysis. [Accessed: 22-Jul-2022].

[6] “Binary search tree,” Wikipedia, 21-Jul-2022. [Online]. Available: https://en.wikipedia.org/wiki/Binary\_search\_tree. [Accessed: 22-Jul-2022].

[5] “Dot (graph description language),” Wikipedia, 12-Jul-2022. [Online]. Available: https://en.wikipedia.org/wiki/DOT\_(graph\_description\_language). [Accessed: 22-Jul-2022].

[7] “Invariant (mathematics),” Wikipedia, 27-Mar-2022. [Online]. Available: https://en.wikipedia.org/wiki/Invariant\_(mathematics). [Accessed: 22-Jul-2022].

[8] “Node (computer science),” Wikipedia, 13-Jul-2022. [Online]. Available: https://en.wikipedia.org/wiki/Node\_(computer\_science). [Accessed: 22-Jul-2022].

[4] “OpenGL,” Wikipedia, 10-Jul-2022. [Online]. Available: https://en.wikipedia.org/wiki/OpenGL. [Accessed: 22-Jul-2022].

[10] “Red–Black Tree,” Wikipedia, 22-Jul-2022. [Online]. Available: https://en.wikipedia.org/wiki/Red%E2%80%93black\_tree. [Accessed: 22-Jul-2022].

[3] “Simple and fast multimedia library,” Wikipedia, 04-Jun-2022. [Online]. Available: https://en.wikipedia.org/wiki/Simple\_and\_Fast\_Multimedia\_Library. [Accessed: 22-Jul-2022].

[1] “Splay tree - Indiana State University.” [Online]. Available: https://cs.indstate.edu/~rcheruku/splaytree.pdf. [Accessed: 22-Jul-2022].

[2] “Splay Trees,” Splay trees. [Online]. Available: https://www.cs.cornell.edu/courses/cs3110/2013sp/recitations/rec08-splay/rec08.html. [Accessed: 22-Jul-2022].

[11] Techopedia, “What is private? - definition from Techopedia,” Techopedia.com, 11-Dec-2011. [Online]. Available: https://www.techopedia.com/definition/24015/private. [Accessed: 22-Jul-2022].

[12] “What is public?,” What is Public?, 08-Aug-2017. [Online]. Available: https://www.computerhope.com/jargon/p/public.htm. [Accessed: 22-Jul-2022].