Implementation And Analysis Of Multi-Splay Trees

November 21, 2022

Ojassvi Kumar ,
Aman Pankaj Adatia ,
Aman Kumar

Summary: Multi-Splay trees are a type of data structure that are composed of multiple splay trees and are conjectured to be Dynamically Optimal. The objective of this project is to develop a Multi-Splay Tree using C++ and evaluate its performance on different test cases. The project also involves a theoretical analysis of the amortized cost of a Multi-Splay Tree. The implementation of the Multi-Splay Tree is based on the principles of Binary Search Trees and Splay Trees.

1. Introduction

The concept of Splay Trees and their practical applications have been a topic of interest in computer science for several decades. Splay Trees, which are a variation of Binary Search Trees, use a splaying operation to bring a node to the root upon access. The splaying operation preserves the BST property and provides several benefits, including improved search times and efficient use of memory.

However, while Splay Trees have been shown to be effective, they are not without limitations. For example, their performance can be sensitive to the order in which nodes are accessed, and their amortized complexity may not be optimal in all scenarios.

To address some of these limitations, Multi-Splay Trees were introduced as an extension of Splay Trees. Multi-Splay Trees are a collection of Splay Trees, each with their own set of properties. The nodes in a Multi-Splay Tree are divided into sets, with each set representing a Splay Tree. These sets can be viewed as preferred paths that represent the order in which nodes are accessed.

One of the key benefits of Multi-Splay Trees is that they provide a competitive bound of O(log log n) and an amortized complexity of O(log n) for accessing elements in a BST, which has been proven using the access lemma. This means that the time complexity of accessing elements in a Multi-Splay Tree is logarithmic in nature, making it a highly efficient data structure for a wide range of applications.

Furthermore, Multi-Splay Trees have several other interesting properties, such as Sequential Access Property, Dynamic Finger Property, Working Set Property, and Dynamic Optimality. These properties help to reduce the amortized cost of accessing elements in the tree, while also improving its overall performance.

Overall, Multi-Splay Trees are a fascinating data structure with numerous practical applications. While they are more complex than traditional Splay Trees, their superior performance and efficiency make them an ideal choice for a wide range of applications, including data caching, database indexing, and network routing.

2. Reference tree

Assuming there are exactly $n = 2^k - 1$ nodes, a perfectly balanced binary search tree consisting of these n nodes is referred to as the reference tree (denoted as P). The depth of any node in P is at most $lg(n+1)$ with the root defined as depth 1. Every node in the reference tree has a preferred child, and the structure of the reference tree is static except that the preferred children will change over time. A chain of preferred children is called a preferred path, and the nodes of the reference tree are partitioned into 2^{k-1} sets, one for each preferred path. Although not an explicit part of the data structure, the reference tree is beneficial in comprehending how it operates.

The multi-splay trees data structure is a binary search tree that undergoes changes over time while maintaining a close relationship with the reference tree. It operates over the same set of n keys as the reference tree. The edges of the multi-splay trees are either solid or dashed. A splay tree is a set of vertices connected by solid edges. The multi-splay trees have a one-to-one correspondence with the preferred paths of the reference tree. The set of nodes in a splay tree is the same as the nodes in its corresponding preferred path. Thus, the multi-splay trees can be derived from the reference tree by viewing each preferred edge as solid and performing rotations on the solid edges.

It's worth remembering that the reference tree and the multi-splay tree convey the same set of nodes in the same symmetric sequence. As a result, the multi-splay tree is a legitimate representation of the binary search tree for the given node set. For the purposes of this discussion, we'll use the notation T to refer to the multi-splay tree.

Every node of the multi-splay tree T has several fields in it, which we enumerate here. First of all, there are the usual keys, left, right, and parent pointers. Although the reference tree P is not explicitly represented in T we do keep several pieces of information related to the reference tree. In each node we keep its depth in P, and its height in P. (The height of a leaf in P is zero, and the depth of the root of P is 1.) Both of these quantities are static. (Note that every node in the same splay tree has a different depth in P.) Another field we store in each node is mindepth. This is the minimum depth of all the nodes in the splay sub-tree rooted there. By splay subtree of x we mean all the nodes in the same splay tree as x that have x as an ancestor (which includes x). Similarly we store *treesize*, which is the number of nodes in the splay sub-tree rooted at this node. To represent the solid and dashed edges, we keep a Boolean variable in each node that indicates if the edge from this node to its parent is dashed. We'll call this the *isroot* bit.

2.1. Figures

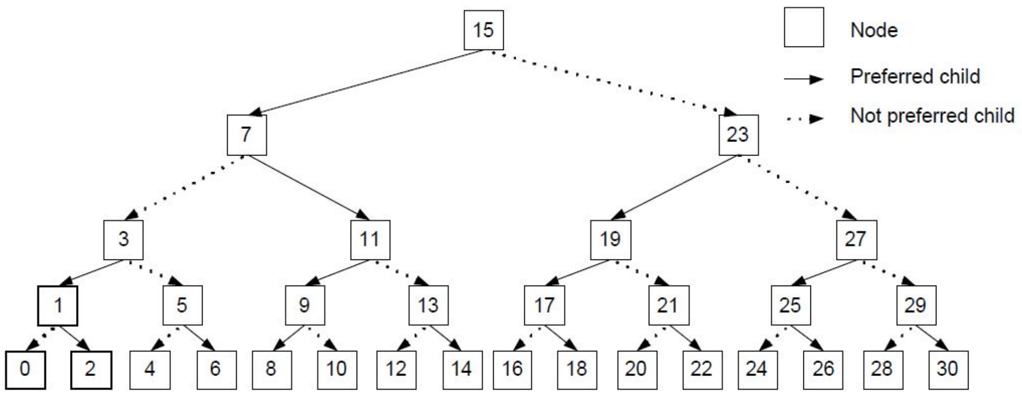

Figure 1: Reference Tree P

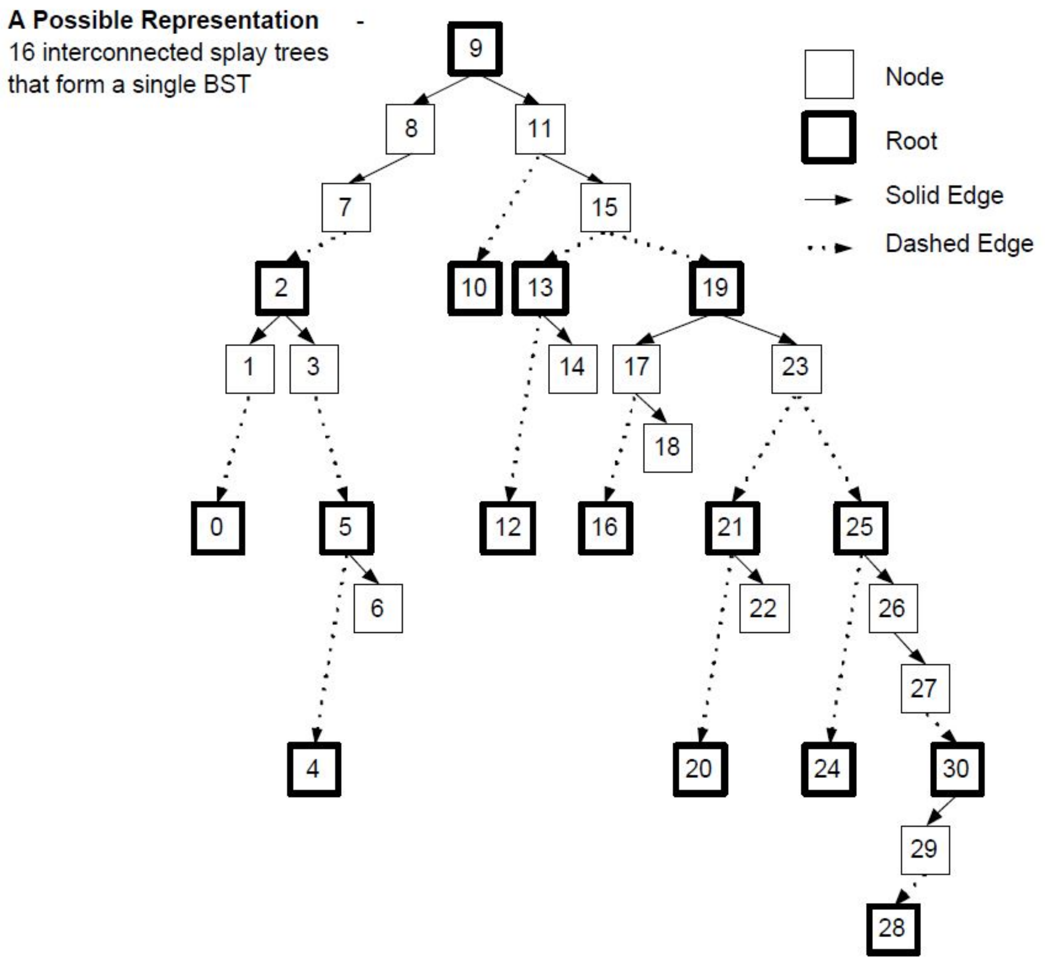

Figure 2: 16 inter-connected Splay Trees to form a single BST

3. Functions employed

treeFor(): Function to construct the whole tree.
query(): Function to access an element in the tree.
rotate(): Function to perform the standard rotation operation.
splay(): Function to perform the standard splaying operation.
expose(): Function to bring the current node to the root of the whole tree.
switchPath(): Function to adjust the depth and minDepth values of the nodes.
refParent(): Function to return the first child whose minDepth value is greater than the depth value.
explore(): Auxiliary function to help display the whole tree.

4. Algorithms

The function `query()` allows us to access any node in the Multi-Splay Tree. When called, it splays the node until it reaches the root of the entire BST. The splaying process starts by moving the node to the root of the splay tree to which it belongs, using necessary rotations. Then, the function uses `refParent()` and `switchPath()` to further splay the node and adjust its depths and minDepths until the parent of the accessed node is NULL, indicating that the accessed node has reached the root of the BST.

To carry out its task, `query()` relies on the `expose()` function, which in turn uses `splay()`, `rotate()`, `refParent()`, and `switchPath()` to perform the necessary splaying and adjustment procedures.

Algorithm 1 **bool query(key)**

```
1: Search until we find either the key or NULL
2: curr = current node
3: par = parent of curr
4: if curr == NULL then
5:     expose(par)
6:     return false
7: end if
8: expose(curr)
9: return true
```

Algorithm 2 **void expose(node)**

```
1: while node->parent != NULL do
2:     splay(node)
3:     update depth and minDepth values
4: end while
5: return
```

5. Observations

The following table presents the approximate Build Time and Query Time taken by the Multi-Splay Tree in various test cases. These test cases were conducted using different access sequences on trees of varying sizes, ranging from 30 to 300,000 elements. The purpose of these tests was to evaluate the performance of the Multi-Splay Tree under different conditions, and to identify any patterns or trends in its behavior.

It is important to note that the times reported in the table are approximate, and may vary depending on factors such as the hardware used. Nevertheless, they provide a useful benchmark for comparing the performance of the Multi-Splay Tree across different test cases and contexts.

	Build Time	**Query Time**
Sequential		
1. Size: 30	3ns	4ns
2. Size: 3000	100ns	700ns
3. Size: 300000	10500ns	245000ns
Reverse		
1. Size: 30	3ns	5ns
2. Size: 3000	50ns	800ns
3. Size: 300000	7500ns	290000ns
Random		
1. Size: 30	3ns	9ns
2. Size: 3000	50ns	2200ns
3. Size: 300000	6200ns	520000ns

Table 1: Run-time Analysis

6. Some useful results without proof

Theorem 6.1. *Multi-Splaying is amortized O(log n).*

The amortized time taken to access elements in a Multi-Splay Tree is O(log n).

Theorem 6.2. *Multi-Splaying is competitive O(log log n).*

Multi-Splay Tree is a competitive BST having a time complexity of O(log log n).

Theorem 6.3. *For any query in a multi-splay tree, the worst-case cost is* $O(\log^2 n)$.

This follows from the fact that to query a node, we visit at most O(height(P)) splay trees. Because the size of each splay tree is O(log n), the total number of nodes we can possibly touch is $O(\log^2 n)$.

7. Conclusions

The Multi-Splay Tree algorithm is remarkable for its competitiveness in a Binary Search Tree, as it is the only one known to have a competitiveness of O(log log n). This is an impressive feat, as it ensures that the search times for the Multi-Splay Tree algorithm will grow at a slower rate than other algorithms as the number of nodes in the tree increases. This makes it a desirable choice for applications that require quick and efficient search times.

Moreover, the theorems and results obtained from the analysis of the Multi-Splay Trees algorithm demonstrate that it satisfies many important properties of a Dynamically Optimal BST algorithm. Dynamically Optimal BST algorithms are those that achieve the best possible search times in a BST given a sequence of search queries.

Overall, the Multi-Splay Trees algorithm is an impressive example of an efficient and effective BST algorithm that achieves excellent search times and satisfies many of the key properties of a Dynamically Optimal BST algorithm.

8. Bibliography and citations

[1] [2] [3] [4] [5]

Acknowledgements

Papers published by researchers at the School of Computer Science, Carnegie Mellon University in Pittsburgh.

References

[1] Amy Chou. Tango tree and multi-splay tree. *Github*.

[2] Jonathan Derryberry Daniel Sleator and Chengwen Chris Wang. Properties of multi-splay trees. *CS. CMS*, 2009.

[3] Parker J. Rule and Christian Altamirano. Multi-splay trees and tango trees in c++. *Github*.

[4] Daniel Dominic Sleator and Chengwen Chris Wang. Dynamic optimality and multi-splay trees. *CS, CMS*, 2004.

[5] Chengwen Chris Wang. Multi-splay trees. *CS, CMS*, 2006.

Hyperlink:
Research Papers: [2] [4] [5]