Implementation and Analysis of Multi-Splay Trees

November 21, 2021

Aman Pankaj Adatia (2020CSB1154) ,
Aman Kumar (2020CSB1153) ,
Ojassvi Kumar (2020CSB1187).

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Sravanthi Chede

Summary: This project aims at the implementation of Multi-Splay Trees in C++ and analysing the run-time on some test cases. We also analyse the amortized cost of Multi-Splay Trees theoretically and study the different properties such as Sequential Access property, Static Finger property, and various other Lemmas associated with it. We have used the concepts of Binary Search Trees and Splay Trees. Multi-Splay Trees are conjectured to be Dynamically Optimal.

---

1. Introduction

A Splay Tree is a Binary Search Tree which performs an extra operation called splaying. Splaying is an algorithm which along moves a node to the root upon access, such that the BST property is still maintained. Splay Tree have various useful properties which help us reduce the amortized cost of accessing elements in the BST. Some of these properties are Sequential Access Property, Dynamic Finger Property, Working Set Property, Dynamic Optimality, etc. A Multi-Splay Tree is a tree made up of Splay trees, wherein the nodes have several different types of properties. The root of each individual splay trees is darkened.

Multi-Splay tree has O(log log n) competitive bound and O(log n) amortized complexity for access in a BST, which is proved by an access Lemma. In the Multi-splay tree data structure, we assume a reference tree, denoted by P, which is a perfectly balanced tree consisting n nodes. Each node has a preferred child. A path consisting of preferred child of various node is called preferred path. The preference tree is only for understanding purpose and is not part of our data structure.

Our Multi-Splay tree denoted by T, has either solid or dashed edges. Set of nodes connected by solid edges is our Splay Tree. The set of nodes in a Splay Tree is exactly the same as the nodes in its corresponding preferred path. That is, at any point in time the Multi-Splay Trees can be obtained from the reference tree by viewing each preferred edge as solid, and doing rotations on the solid edges.

Functions used for various operations in our implementation:
explore() - auxiliary function to help display the whole tree.
treeFor() - function to create the whole tree.
rotate() - function to perform the standard rotation operation.
splay() - function to perform the standard splaying operation.
switchPath() - function to adjust the depths and minDepths values of the nodes
refParent() - function to return the first child whose minDepth value is greater than the depth value
expose() - function to bring the current node to the root of the whole tree.
query() - function to access an element in the tree (multi-splay).

2. Figures, Tables and Algorithms

An example of a Multi-Splay Tree along with the reference tree for explanation and better understanding.

2.1. Figures

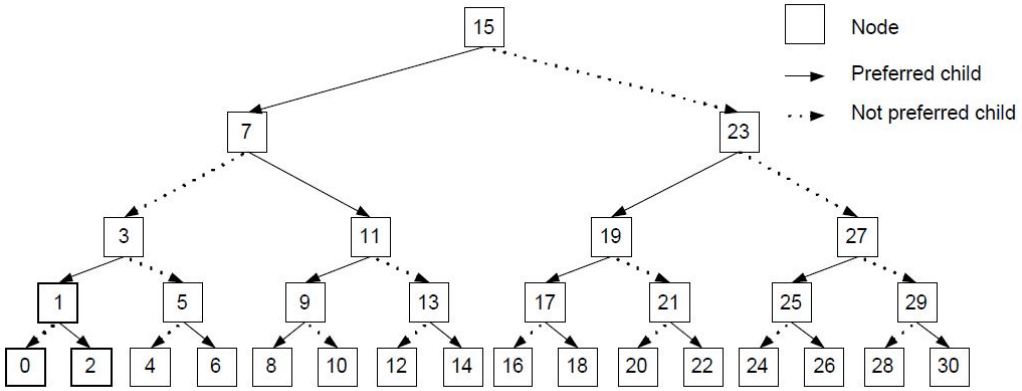

Figure 1: Reference Tree P

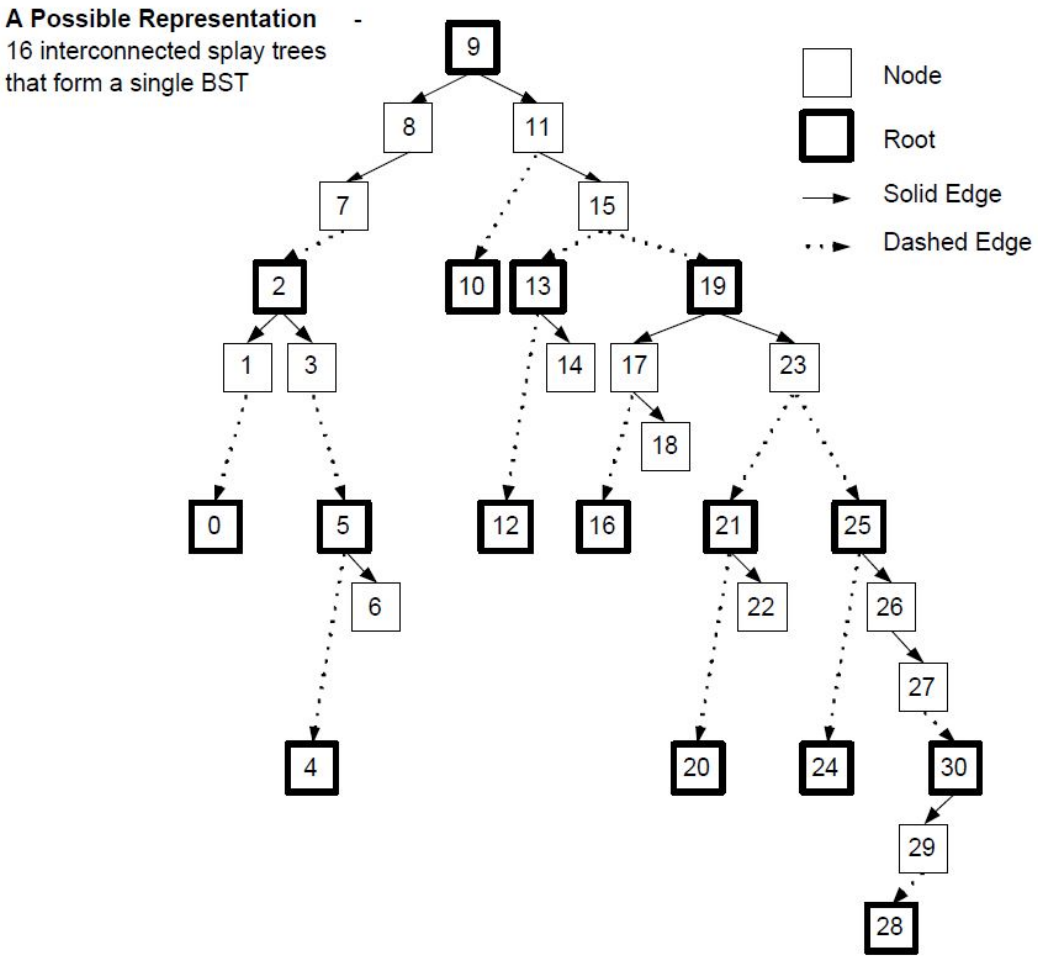

Figure 2: 16 inter-connected Splay Trees to form a single BST

2.2. Tables

This table contains the approximate Build Time and Query Time taken by the Multi-Splay Tree in the different test cases used. For the different test cases, different access sequences were used on the tree having size ranging from 30 to 300000 elements.

	Build Time	Query Time
Sequential		
1. Size: 30	0ns	0ns
2. Size: 3000	900ns	3500ns
3. Size: 300000	200000ns	650000ns
Reverse		
1. Size: 30	0ns	0ns
2. Size: 3000	900ns	2500ns
3. Size: 300000	35000ns	600000ns
Random		
1. Size: 30	0ns	0ns
2. Size: 3000	900ns	1000ns-10000ns
3. Size: 300000	40000ns	500000ns-1000000ns

Table 1: Runtime Analysis

2.3. Algorithms

We call the `query()` function to access any node in the Multi-Splay Tree. Its functionality is that it splays the element until it reaches to the root of the entire BST. So, when splayed for the first time, the node is moved to the root of the splay tree in which it belongs by performing necessary rotations. Following, using `refParent()` and `switchPath()`, the further splaying procedure is taken place by adjusting the depths and minDepths of the node. It ends when the parent of the accessed node is NULL, i.e. the accessed node reaches to the root of the BST. `query()` uses `expose()` function, which in turn uses `splay()`, `rotate()`, `refParent()` and `switchPath()` function.

Pseudo code of the `query()` algorithm:

Algorithm 1 bool query(key)

```
Search until we find either the key or NULL
The search operation is similar to Binary Search
curr = current node
prev = parent of curr
if curr == NULL then
    expose(prev)
    return false
end if
expose(curr)
return true
```

Pseudo code of the `expose()` algorithm:

Algorithm 2 void expose(node)

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

3. Some further useful suggestions

Theorem 3.1. *Multi-Splaying is amortized O(log n).*
The amortized time taken to access elements in a Multi-Splay Tree is O(log n).

Theorem 3.2. *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 3.3. *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 $O(\log^2 n)$.

4. Conclusions

It is the only algorithm with O(log log n) competitiveness in a BST. In addition, the above theorems and results show that Multi-Splay Trees satisfy many of the important properties of a Dynamically Optimal BST algorithm.

5. Bibliography and citations

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

Acknowledgements

Research Papers published by researchers at 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]