Multi Splay Trees: Implementation and Analysis

November 7, 2022

Ayush Raj (2021MCB1346) ,
Vrinda Dua (2021MCB1223) ,
Edgar Aditya Thorpe (2021CSB1169)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Anurag Jaiswal

Summary: Our project entails the implementation and analysis of Multi Splay Trees in C++. We have analysed the run-time on some test cases. Here we will also analyse the amortized cost of Multi-Splay Trees by theoretical approach and also study its different properties such as Static Finger Property, Sequential Access Properties and various other lemmas associated with the topic of Multi-Splay Trees. We have used concepts such as BST(Binary Search Trees) and Splay Trees in the course of this project.

1. Introduction

First of all we need to know what a Splay Tree is, so Splay Tree is basically a binary tree which performs an extra operation which we call splaying, it is like a self adjusting binary tree. Splaying is an algorithm which moves a node or root upon its access such that the Binary Search Tree is still maintained. Splay Trees have various useful properties such which help in reducing the total amortized cost of accessing the elements in the BST. Some of these properties include Sequential Access Property, Working Set Property, Dynamic Optimality etc. A Multi Splay tree is a tree made up of Splay Trees where all the nodes have many different types of properties. We have attached a figure below for reference. All the roots of each individual tree is darkened.

Multi-Splay tree has O(log(log(n))) competitive bound and O(log(n)) amortized complexity for access in a BST, which we have proved by a lemma.For a Multi-Splay Tree we imagine a reference tree denoted by P, which is perfectly balanced tree consisting of n nodes. Each of the node has one preferred child. A path which consists of preferred child of various nodes is called the preferred path. This all makes up a preferrence tree which is only for understanding purposes and not in the scope of our data structure.

We denote our Multi-Splay Tree by T and only contains either dashed or solid edges. The 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 i.e. at any single given point of time the Multi Splay-Trees can be obtained from the refernce tree by viewing each preferred edge as solid and doing rotations on the particular solid edges.

2. Functions Used in Implementation

DispTree() - auxiliary function which displays the whole tree
rotation() - function to perform a standard rotation operation on the tree
Create() - a function to help create the whole tree
splayOp() - a function to perform the basic splaying operation
Transfer() - function to bring the current node to the root of the whole tree
DepthNode() - function to adjust the depths and mindepths values of the nodes
ReferencePar() - function to return the first child whose minDepth value is greater than the depth value
access() - function to access some elements in the tree (multi-splay).

3. Figures, Tables and Algorithms

3.1. Figures

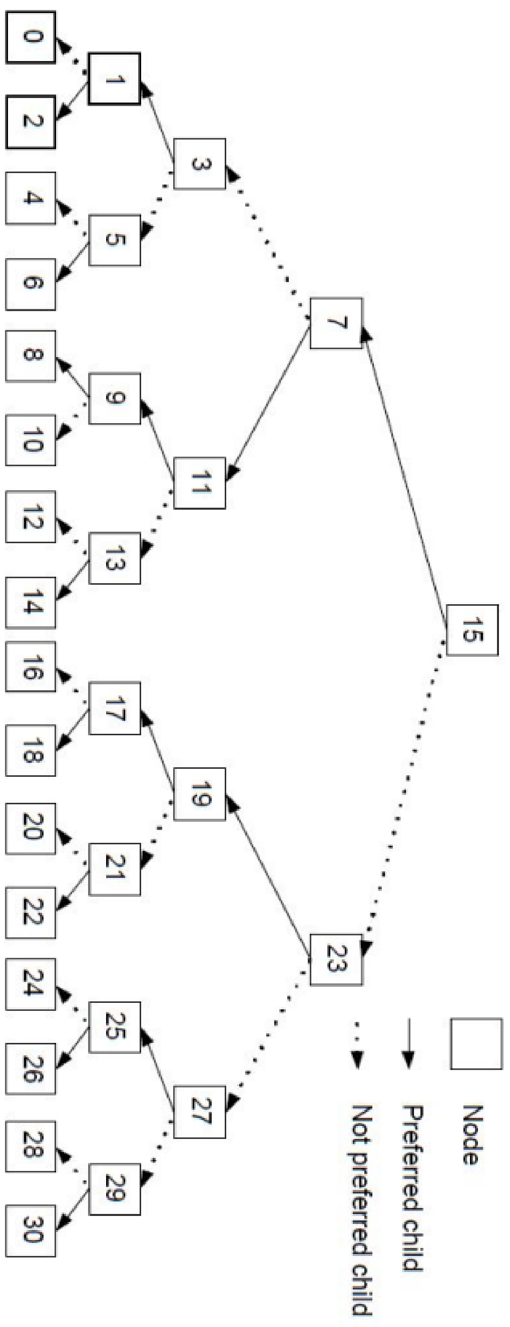

Figure 1: Reference Tree P

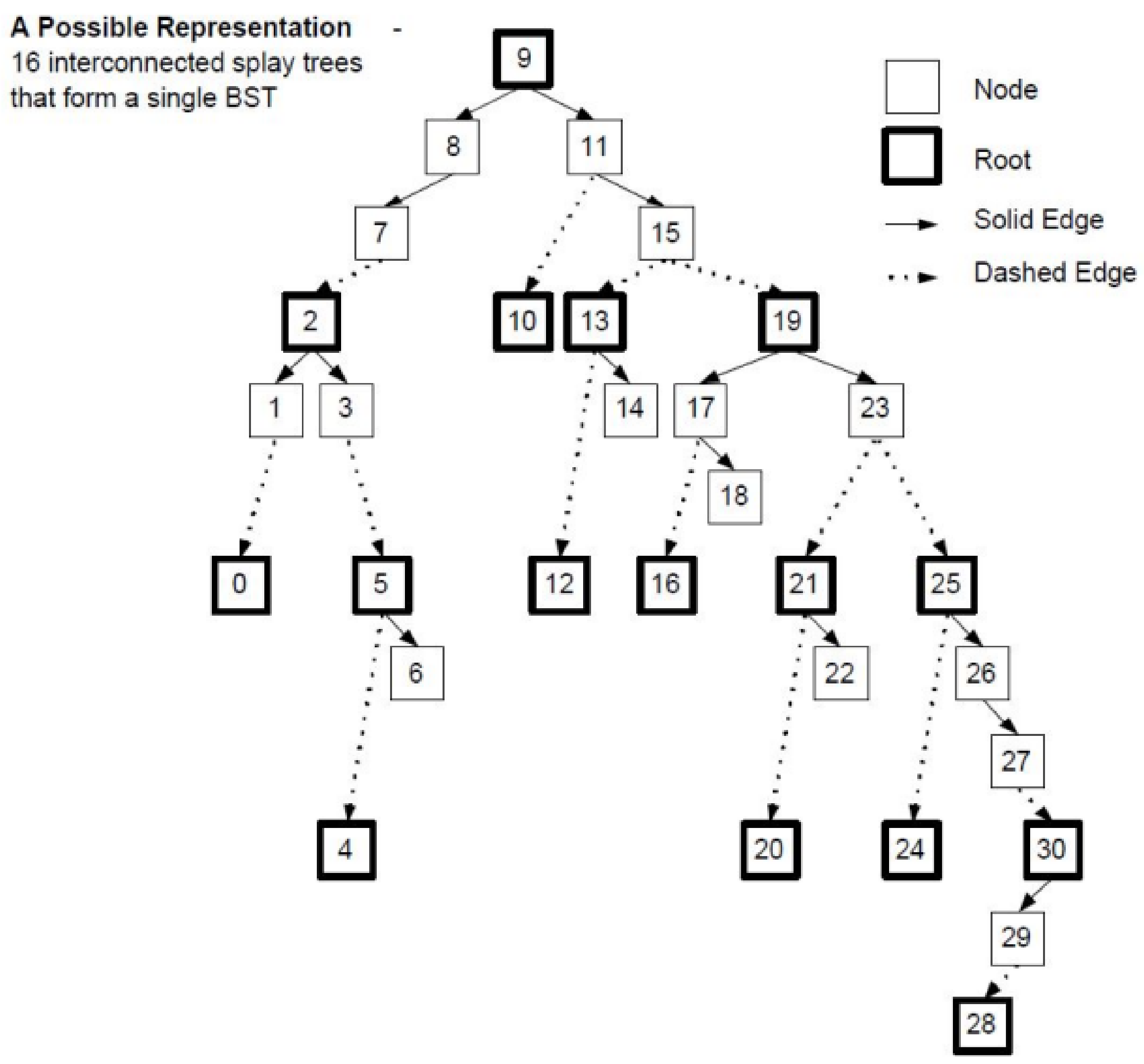

Figure 2: 16 inter-connected Splay Trees to form a single BST

3.2. Build Query Time Tables

We have made a table which displays the approximated Build Time and Query Time taken by the Multi-Splay Tree in many different cases it is being used. For all the different test cases, different access sequence were used on the tree having size ranging for mere 30 elements to even 300000 elements.

TABLE

	Build Time	**Query Time**
Sequential		
Size: 30	0ns	0ns
Size: 3000	900ns	3500ns
Size: 300000	200000ns	650000ns

	Build Time	**Query Time**
Random		
Size: 30	0ns	0ns
Size: 3000	900ns	1000ns - 10000ns
Size: 300000	40000ns	500000ns-1000000ns
Reversal		
Size: 30	0ns	0ns
Size: 3000	900ns	2500ns
Size: 300000	35000ns	600000ns

Table 1: Runtime Anlaysis

3.3. Algorithms

We have made function named access() which can be used to access any node in the Multi-Splay Tree. Its basic working is that it splays the element until it reaches the root of entire BST. So now 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. After this the ReferencePar() and DepthNode(), the further splaying procedure is taken place by adjusting 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. access() uses the Transfer() function which then uses the splayOp(), rotation(), ReferencePar(), and DepthNode() functions.
Pseudo Code for the access() Algorithm is given below:

Algorithm 1 bool access(key)

```
1: Search until we find either the key or NULL
2: The search operation is similar to Binary Search
3: curr = current node
4: prev = parent of curr
5: if $curr == NULL$ then
6:     Transfer prev
7:     return FALSE
8: end if
9: Transfer(curr)
10: return TRUE
```

Pseudo Code for the Transfer() Algorithm is given below:

Algorithm 2 void Transfer(node)

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

Here we have given pseudo codes of only two of the functions from our Data Structure

4. Some further useful suggestions

Some useful theorems for cost calculations

Theorem 4.1. *Multi-Splaying is amortized O(log n).*

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

Theorem 4.2. *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 4.3. *For any query in a multi-splay tree, the worst-case cost is O(log2 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($\log^2 n$).

5. 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. It excels in operating on many complex levels of splay trees.

6. Bibliography and citations

[1] [2] [3] [4] [5]
github.com
google.com

Acknowledgements

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

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.