|  |  |
| --- | --- |
|  | **Department of Electrical and Computer Engineering** |
| **North South University** |

**CSE225.2 Project Report**

Submitted to Tamanna Motahar (TMM)

By

Sams Uddin Ahmed(1620129042)

&

Najmus Sakib Borson(1610538642)

21 August,2017

Abstract

This report describes the study and implementation of Scapegoat trees. The Scapegoat trees guarantee the amortized complexity of Insert and Delete O(log n) time while the worst case complexity of Searching is O(log n) time where n is the number of nodes in the tree. It is the first kind of balanced binary search trees that do not store extra information at every node while still maintaining the complexity as above. The values kept track are the number of nodes in the tree and the maximum number of nodes since the trees were last rebuilt. The name of the trees comes from that rebuilding the tree only is taken place only after detection of a special node called Scapegoat, whose rooted subtree is not weight-balanced.

|  |
| --- |
|  |
| **Table of Contents** |

1. Introduction,Objective……………………..4
2. Functions………………………………………….5
3. Insert………………………………………….…….6
4. Rebuild……………………………………………..7
5. Delete,Other Functions……………………..8
6. Time Complexity……………………….……….9
7. References………………………………….…...10

***Introduction***

The scapegoat tree is an implementation of a self-balancing binary tree. This structure is based on the common wisdom that when something goes wrong, the first thing people tend to do is find someone to blame as known as scapegoat. Once blame is firmly established, we can leave the scapegoat to fix the problem.

***Objective***

The objective of Scapegoat trees are to let us use a more efficient BST which gets automatically balanced after every operations like INSERT and DELETE. As a result the SEARCH operation takes less time to find a value because the tree is balanced everytime.

***Functions***

All the functions are similar like BST. But insertion and deletion are bit different. We added some new functions like

**addwithDepth(Node\*)**

**Rebuild(Node\*)**

**packintoArray(Node\*,Node\*[],int)**

**buildbalancedfromArray(Node\*\*,int,int)**

Except this functions all other functions like **Search(float),IsEmpty(),SizeOfTree(Node\*)** etc are same.

***Insert***

Here we just created a newly added tree’s object and pass it to addwithDepth(Node\*) function. In this addwithDepth(Node\*) function we just simply add the new node, count the depth of that node and also locate the parent node of that newly added node. This functions returns the counted depth of last node. Then we check if that depth is greater then log32(NoOfNodes). If it is then we need to find out the scapegoat from newly added node by size(w)/size(w->parent)>2/3 (Using loop). After finding the scapegoat node, we just rebuild that scapegoat node by calling the rebuild function.

***Rebuild***

Here we just created a new Node type array who’s size is the size of that scapegoat node. Then we just pass that array,that scapegoat node and array size onto packintoArray(Node\*,Node\*[],int). This packintoArray function is a recursive function. On this function we store inorder traversal on BST in that array.

After allthis process, we just call the buildbalancedfromArray(Node\*\*,int,int) function where we pass that inordered array, index of array and size of that array. This is also a recursive function which returns Node\* type. Here we build a new BST from array by recursively dividing it into two halves.

***Delete***

We simply delete a node by standard deletion of BST. Then we decrement the total number of nodes. After decrementing we check if the tree is unbalanced or not by following condition:

(NoOfNodes<MaxNode/2)

Where MaxNode is overestimation of NoOfNodes.

If this condition fulfills then we rebuild the entire tree by calling rebuild function. Then we set MaxNode = NoOfNodes.

***Other Functions***

Search function is exactly same as BST’s search function. SizeOfTree(Node\*) function returns the total number of Nodes.

***Time Complexity***

|  |
| --- |
| **Scapegoat tree** |
| [**Time complexity**](https://en.wikipedia.org/wiki/Time_complexity)**in**[**big O notation**](https://en.wikipedia.org/wiki/Big_O_notation) |
| |  |  |  |  | | --- | --- | --- | --- | | **Algorithm** |  | **Average** | **Worst Case** | | **Space** |  | O(n) | O(n) | | **Search** |  | O(log n) | O(log n) | | **Insert** |  | O(log n) | amortized O(log n) | | **Delete** |  | O(log n) | amortized O(log n) | |

Starting with an empty scapegoat tree, a sequence of m

insertions and deletions takes O(mlog n) .

***References***

-https://en.wikipedia.org/wiki/Scapegoat\_tree/

- <http://www.geeksforgeeks.org/scapegoat-tree-set-1-introduction-insertion/>

- <https://prezi.com/y4d1jembgnxu/scapegoat-trees/>

- <https://brilliant.org/wiki/scapegoat-tree/#operations/>

- http://opendatastructures.org/versions/edition-0.1g/ods-python/8\_Scapegoat\_Trees.html/