SCAPEGOAT Tree

November 21, 2021

M.N.S.M.S.Vishnu (2020CSB1097) ,
Gopagoni Sreya (2020CSB1087) ,
D.V.S.Lasyanth (2020CSB1083)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Sarthak Joshi

Summary: A Scapegoat Tree is a self-balancing binary search tree, which is based on the concept of a scapegoat. Typically, a scapegoat is someone who is blamed when something goes wrong. Binary search trees can become unbalanced after insertions and deletions. It is the first kind of balanced binary search trees that do not store extra information at every node maintaining the time 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 .The values kept track the number of nodes in the tree and the maximum number of nodes since the trees was last rebuilt. The name of the trees comes from that locating, deconstructing and rebuilding the tree takes place only after detection of a special node called Scapegoat, whose rooted sub-tree is not weight-balanced.

1. Introduction

scapegoat tree was invented by Arne Anderson (of AA trees) in 1989. This idea was rediscovered by Galperin and Rivest in 1993, who made some refinements and gave it the name “scapegoat tree”. There are two main balanced binary search tree based on: height-balanced and weight balanced. The height-balanced scheme maintains the height of the whole tree in O(log n) where n is the number of nodes in the tree.RED-BLACK trees and AVL trees are examples of this scheme in which the worst-case cost of every operation is O(log n) time. The weight-balanced scheme ensures the size of sub-trees rooted at siblings for every node in the tree approximately equal. The technique used in Scapegoat trees combines those two schemes.By maintaining weight-balanced, it also maintains height-balanced for a Search operation and by detecting a height-unbalanced sub-tree and rebuilding the sub-tree, it also ensures weight-balanced after an update operation.

All of the binary search trees that we have studied achieve balance through the use of rotation operation. Scapegoat trees are unique in that, they do not rely on rotation. This is significant because there exist binary trees that cannot be balanced through the use of rotations. (One such example is the binary space partition tree) As we shall see, the scapegoat tree achieves good balance by “rebuilding” sub-trees that exhibit poor balance. The trick behind scapegoat trees is figuring out which sub-trees to rebuild and when to do this.It does not need to store any additional information in the nodes, other than the key, value, and left and right child pointers. (Additional information, such as parent pointers may be added to simplify coding, however, but these are not needed.) Nonetheless, it height will always be O(log n).

1.1. Notations

If n is a node in a Scapegoat tree T then

- n.value is the value of the key stored at the node n
- n.left refers to the left child of n
- n.right refers to the right child of n

- n.height refers to the height of the sub-tree rooted at n or the longest distance in terms of edges from n to some leaf.
- n.depth refers to the depth of node n or the distance in terms of edges from the root to n
- n.parent refers to the parent node of node n
- n.sibling refers to the other child, other than n, of parent of n
- n.size refers to the number of nodes of sub-tree rooted at n.
- n.h(α) is computed as n.hα= log1/α (n.size)
- n.size() is the procedure to compute the size of the binary search tree rooted at node n
- T.root refers to the root of the tree T
- T.size refers to the current number of nodes in the tree T
- T.maxSize refers to the maximum number of nodes or the maximum number of T.size is or was since T was completely rebuilt because whenever the whole tree T is rebuilt, T.maxSize is set to T.size
- T.h(α) is computed as following: T.hα = log1α (T.size)
- T.height refers to the height of T or the longest path in terms of edges from T.root to some leaf in T

These notations are used in the descriptions of operations on Scapegoat trees.This notation is used for proofs of correctness and time complexity

1.2. Weight Balanced Binary Search Tree

A node n in a binary search tree is α-weight-balanced for some α such that $1/2 <= \alpha < 1$ if (n.left).size $<= \alpha$ $\cdot$ n.size and (n.right).size $<= \alpha \cdot$ n.size.

1.3. Height Balanced Binary Search Tree

A binary search tree is α-height-balanced for some α such that $1/2 <= \alpha < 1$ if T.height <= T.h (α).

2. Algorithms,Figures and Tables

2.1. Algorithms

2.1.1 Search

The Search operation in Scapegoat tree T is done like regular Search in a binary search tree. As the height of a Scapegoat tree T is always loosely-height-balanced in term of the number of nodes T.size after Insert or Delete operations or T.height < ceil(log1/α (T.size)) then worst-case running time is O(log (T.size)) time.

Algorithm 1 Search(k)

```
 1: Here,we declared the tree node "root" globally to avoid some errors.
    Input: root is the root of some tree T to search for an integer key k
    Output: n is a node in T such that n.key = k or null if there is no such n
 2: w=root
 3: while 1 do
 4:    Some instructions
 5:    if $w = null$ then
 6:       return 0
 7:    end if
 8:    if $w.value = k$ then
 9:       return 1
10:    end if
11:    if $w.value < k$ then
12:       w=w.right
13:    end if
14:    if $w.value > k$ then
15:       w = w.left
16:    end if
17: end while
```

2.1.2 Insert

Insertion:

- The key is first inserted just as in a standard (unbalanced) binary tree
- We monitor the depth of the inserted node after each insertion, and if it is too high, there must be at least one node on the search path that has poor weight balance (that is, its left and right children have very different sizes).
- n such a case, we find such a node, called the scapegoat, and we completely rebuild the subtree rooted at this node so that it is perfectly balanced.

When T is not height-balanced then a Scapegoat node s will be detected and rebuilding will be taken place at the subtree rooted at s. . The Scapegoat s will be the first ancestor of n such that s.height > s.hα. After the Scapegoat s is detected if applicable, the procedure RebuildT ree(size, root) will get called. RebuildT ree will rebuild a new 1/2-weight-balanced subtree from the subtree rooted at Scapegoat node returned. The Insert procedure also makes use of InsertKey(k) which is a modified version of regular insertion in a binary search tree that will return the height for the newly inserted node for comparison with T.hα to detect whether a newly inserted node is a deep node or not

Algorithm 2 Insert(k)

```
 1: Here,we declared the tree node "root" globally to avoid some errors.
    Input: The integer key k
    Output: true if the insertion is successful, false if there exists a node n such that n.key = k
 2: node.value = k;
 3: node.parent,left,right=NULL
 4: $d = BSTInsert(node)$
 5: if $d = -1$ then
 6:    return -1
 7: else if $w.value = k$ then
 8:    p = node.parent
 9:    while $(size(p) <= *size(p- > parent))//findingscapegoatnode$ do
10:       p = p->parent
11:    end while
12:    rebuid(p.parent)
13: end if
```

2.1.3 Delete

- The key is first deleted just as in a standard (unbalanced) binary tree
- Once the number of deletions is sufficiently large relative to the entire tree size,rebuild the entire tree so it is perfectly balanced.

Here, BSTdelete return 1 if deleted else 0

Algorithm 3 Delete(k)

```
1: Input: The integer key k
   Output: There is no node n in T such that n.key = k
2: deleted=BSTDelete(k)
3: if *deleted* then
4:    if $q > 2 * n$ then
5:       Rebuild(T.root)
6:    end if
7: end if
```

2.1.4 Rebuild

A Scapegoat Tree keeps itself balanced by partial rebuilding operations. During a partial rebuilding operation, an entire sub-tree is deconstructed and rebuilt into a perfectly balanced sub-tree. There are many ways of rebuilding a sub-tree rooted at node u into a perfectly balanced tree. One of the simplest is to traverse u's sub-tree, gathering all its nodes into an array, a, and then to recursively build a balanced sub-tree using a.

Algorithm 4 rebuild(u)

```
1: Input:u is the root of sub-tree
   Output: A 1/2-weight-balanced sub-tree built from all the nodes of the sub-tree rooted at u. The
   root of the rebuilt sub-tree will be returned and whole tree is 1/2-weight-balanced
2: do ns = size(u)
3: do p = u.parent
4: storeInArray(u, a, 0) ( a is pointer array of size ns)
5: if ($p == nil$) then
6:    u = Balancedtree(a, 0, ns)
7:    u.parent = NULL
8: else if ($p.right == u$) then
9:    p.right = buildBalanced(a, 0, ns)
10:   p.right.parent = p
11: else
12:   p.left = buildBalanced(a, 0, ns)
13:   p.left.parent = p
14: end if
```

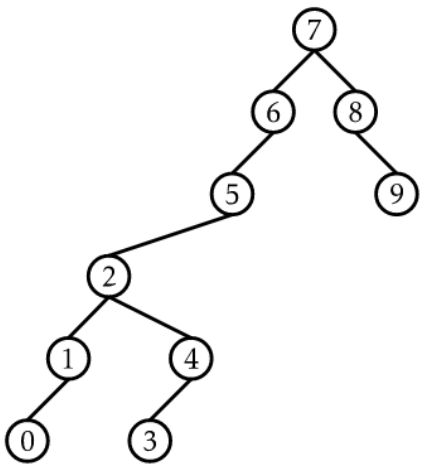
A call to rebuild(u) takes O(size(u)) time. The resulting subtree has minimum height; there is no tree of smaller height that has size(u) nodes.

2.1.5 Balancedtree

Algorithm 5 Balancedtree(a,i, ns)

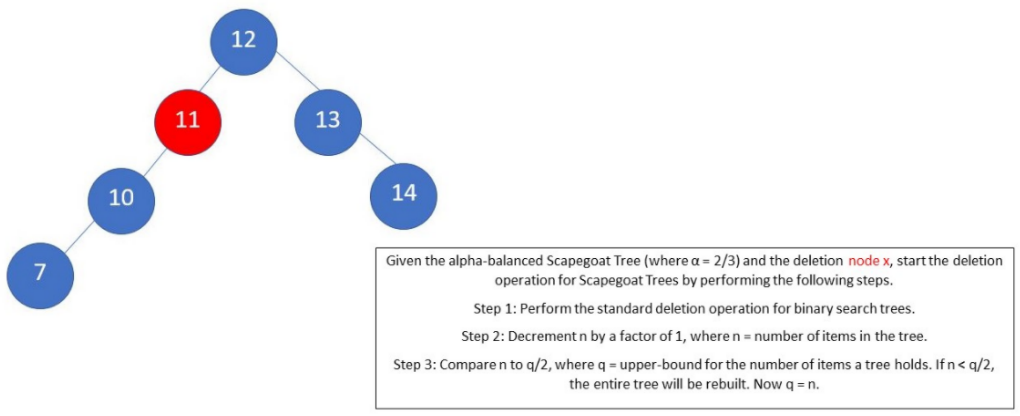
```
Input:n is size,a is pointer array
Output:A 1/2-weight-balanced tree built from the list above. The last node of the list will be returned.
if ($ns == 0$) then
    return NULL
    do m = n / 2
    do a[i+m].left = Balancedtree(a, i, m)
end if
if ($a[i+m].left! = NULL$) then
    a[i+m].left.parent = a[i+m]
    a[i+m].right =Balancedtree(a, i+m+1, n-m-1)
end if
if ($a[i+m].right! = NULL$) then
    a[i+m].right.parent = a[i+m]
    return a[i+m]
end if
```

2.2. Figures



A ScapegoatTree with 10 nodes and height 5.

Figure 1: example of scapegoat tree



Given the alpha-balanced Scapegoat Tree (where α = 2/3) and the deletion node x, start the deletion operation for Scapegoat Trees by performing the following steps.

Step 1: Perform the standard deletion operation for binary search trees.

Step 2: Decrement n by a factor of 1, where n = number of items in the tree.

Step 3: Compare n to q/2, where q = upper-bound for the number of items a tree holds. If n < q/2, the entire tree will be rebuilt. Now q = n.

Figure 2: deletion

2.3. Tables

The complexity of the scapegoat tree is very much like other binary search trees especially very much similar to AVL Tree.

Table 1: scapegoat tree complexity

S.No	Operation	Average	worst-case
1	Space	O(n)	O(n)
2	Search	O(log n)	O(log n)
3	Insert	O(log n)	amortized O(log n)
4	Delete	O(log n)	amortized O(log n)

Table 2: AVL tree complexity

S.No	Operation	Average	worst-case
1	Space	O(n)	O(n)
2	Search	O(log n)	O(log n)
3	Insert	O(log n)	O(log n)
4	Delete	O(log n)	O(log n)

Table 3: Comparison between AVL trees and Scapegoat tree

S.No	AVL Tree	Scapegoat Tree
1	AVL trees check the balance value on every node after insertion/deletion, it is typically stored in each node	scapegoat trees calculate it only when it is needed, which is only when a scapegoat needs to be found.
2	If there are n nodes in AVL tree, minimum height of AVL tree is floor(log2n).If there are n nodes in AVL tree, maximum height can't exceed 1.44*log2n.	If there are n nodes in scapegoat tree,it has restrictions on its height which is always be O(log n)
3	AVL Tress relay completely on rotations and all trees cannot be balanced using rotations	Scapegoat trees are unique that they do not rely on rotation
4	Additional per-node memory overhead. a node additionally stores the balance factor.	no additional per-node memory overhead. A node stores only a key and two pointers to the child nodes.
5	Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where is the number of nodes in the tree prior to the operation	It provides worst-case O(log n) lookup time, and O(log n) amortized insertion and deletion time.
6	Balancing the tree is achieved through the use of many rotation at various nodes until it is balanced.	scapegoat tree achieves good balance by "rebuilding" sub-trees that exhibit poor balance. Once we've found the scapegoat we completely destroy the sub-tree rooted at w and then to recursively rebuild it into a balanced sub-tree.

3. theorems

3.1. Search theorem

Worst-case complexity of any Search operation done in Scapegoat tree T is O(log (T.size)).

3.2. insert theorem

If a Scapegoat tree T is built from a sequence of n Insert operations and m Search or Delete operations starting with an empty tree, then the amortized cost of Insert is O(log n) time

3.3. delete theorem

If a Scapegoat tree T is created from a sequence of n Insert operations and m Search or Delete operations starting with an empty tree, then the amortized cost of Delete is O(log n) time.

4. Conclusions

In this project we implemented scapegoat tree and understood all its operations in depth and compared it with AVL Tree in terms of operations like search, insertion, delete and complexity. a Scapegoat tree T is always loosely -height-balanced after any update operation. Such loosely -height balance is due to rebuilding operation taken place after detecting some Scapegoat node, which is a sign of -height unbalance and -weight unbalance. But the rebuilding operation is shown to happen after enough update operations to pay for cost of rebuilding subtree rooted at the Scapegoat node which is linear in the size. Therefore, a Scapegoat tree T could handle any Search operation in worse-case complexity O(log(T.size)) and the amortized cost of any update operation is O(log(T.size)) time.

5. Bibliography and citations

- **https://opendatastructures.org/newhtml/ods/latex/scapegoat.htm**
- **https://en.wikipedia.org/wiki/Scapegoattree**
- **http://www.domiciaherring.com/lesson−4−what−happen−during−deletion**
- **Igal Galperin and Ronald L. Rivest. Scapegoat trees.**
  In Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms, pages 165 – 174.
  Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 1993.