SCAPEGOAT TREE

November 5, 2023

JYOTI (2022CSB1319) ,
RADHA (2022CSB1108) ,
SHAIKH ASRA SWALEH (2022CSB1121)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
E Harshith Kumar Yadav

Summary: The Scapegoat Tree is a specialized form of binary search tree. What makes it stand out is its responsiveness to imbalance. When the tree becomes severely lopsided, the Scapegoat Tree steps in to restructure itself, ensuring that it remains efficient for search and retrieval operations.
This adaptability makes the Scapegoat Tree a powerful choice for scenarios where the dataset may change over time. By maintaining balance, it allows for swift and accurate data access. Understanding the nuances of the Scapegoat Tree can be a valuable asset for programmers and computer scientists, enabling them to make informed decisions about data structure selection in their projects.

---

1. Introduction

In the ever-evolving landscape of computer science, the efficiency of data structures is a critical factor in tackling complex computational tasks. Among the array of available options, the Scapegoat Tree emerges as a compelling contender. What sets the Scapegoat Tree apart is its remarkable space complexity, which outshines even renowned counterparts like AVL and Red-Black Trees. In a world where resources are often constrained, the quest for optimizing both time and space becomes paramount. The challenge lies in finding a solution that not only conserves memory but also executes tasks swiftly. This is where the Scapegoat Tree steps in. Scapegoat trees belong to a wider range of BSTs, called -height-balanced trees.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 (the scapegoat). Once blame is firmly established, we can leave the scapegoat to fix the problem.

2. WHAT IS SCAPEGOAT TREE??

A Scapegoat Tree is a BST that,in addition to keeping track of the number of nodes(n) in the tree also keeps a counter(q),that maintains an upper-bound on the number of nodes.For example taking alpha=3/2.

$$\mathbf{q}/2 \leq \mathbf{n} \leq \mathbf{q} \ .$$

In addition, a ScapegoatTree has logarithmic height,at all times, the height of the scapegoat tree does not exceed

$$\log_{3/2} \mathbf{q} \leq \log_{3/2} 2\mathbf{n} < \log_{3/2} \mathbf{n} + 2 \ .$$

Even with this constraint, a ScapegoatTree can look surprisingly unbalanced.Scapegoat nodes are defined by the following relaxed height balance crietion:

$$size(node.child) \leq a \cdot size(node)$$

The above inequality provides 2 interesting corollaries:
1) None of the 2 childrens' subtrees can own more than (ALPHA)·100 percent of the nodes contained within the parent's tree; it sets a hard cap on both childrens' tree sizes.
2) The height balance factor (ALPHA) is bounded between (0.5, 1).

3. Operations in Scapegoat tree

This section explains basic operation of scapegoat tree along with their time complexity.We tested 3 operations, common to BST implementations: INSERT(x), DELETION,SEARCH(x)and REBUILD(NODE). A Scapegoat Tree is a BST that, in addition to keeping track of the number,of nodes in the tree also keeps a counter,that maintains an upper-bound on the number of nodes.

3.1. Insert

The insertion in the scapegoat trees can be done using following steps:
1)We have to keep a counter to record the depth of the new node during insertion scapegoat.
2)Then we insert the node in tree.
3)Then we have to find the scapegoat node :- Identify the first unbalanced ancestor as the scapegoat during the climb back up to the root.
4)Rebuilding operation comes into action after scapegoat node is found.
5)Insertion operation in the scapegoat tree has an average time complexity of O(log n).

3.1.1 Rebuild

1)Traverse nodes in the subtree to obtain values in sorted order.
2)Recursively select the median value as the new root, achieving perfect balance.
3)Rebuilding the subtree rooted at the scapegoat is accomplished in O(n) time, ensuring efficient restoration of balance.

3.2. Deletion

For deletion operation in scapegoat trees:-
1)A property is introduced called MaxNodeCount.
2)MaxNodeCount signifies the highest recorded NodeCount.
3)After a full tree rebalance, MaxNodeCount is initially set to NodeCount.
4)Following an insertion, MaxNodeCount is updated to the maximum of its current value and NodeCount.
5)Deletion in Scapegoat Trees involves standard removal of a node.
6)If NodeCount is less than or equal to times MaxNodeCount, trigger a full tree rebalance at the root.
7)After a rebalance, set MaxNodeCount to the current NodeCount.
8)Deletion operation in the scapegoat tree has an average time complexity of O(log n).

3.3. Search

1)In a scapegoat tree, the search operation is akin to the search performed in a standard binary search tree. It follows the properties of a binary search tree, where for a given node:
2)If the target value is less than the current node's value, the search continues in the left subtree.
3)If the target value is greater than the current node's value, the search continues in the right subtree.
4)If the target value matches the current node's value, the search operation successfully finds the node.
5)Search operation in the scapegoat tree has an average time complexity of O(log n).[1].

4. Figures, Amortized analysis and Algorithms

4.1. Figures

[2].

4.1.1 Insertion figures

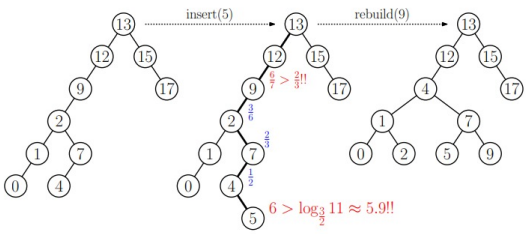

Figure 1: Insertion.

4.1.2 Deletion figures

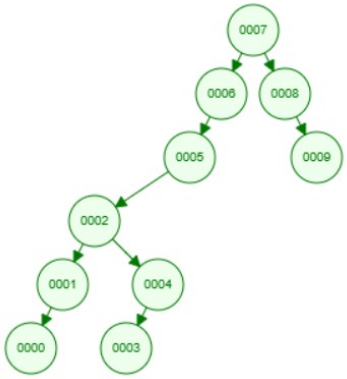

Figure 2: Initially n=10,q=10.

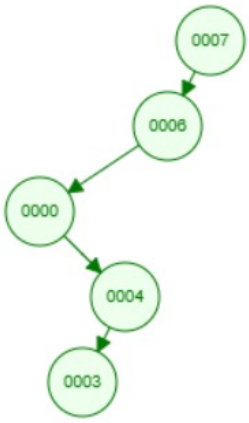

Figure 3: After a series deletion of 8,5,1,2 and 9, n=5 and q=10

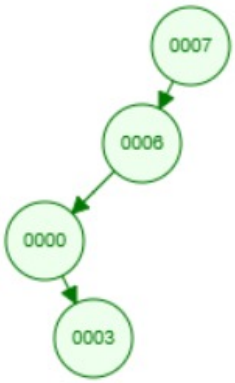

Figure 4: After deletion of 4 , n=4 and q=10 so 2*n<q

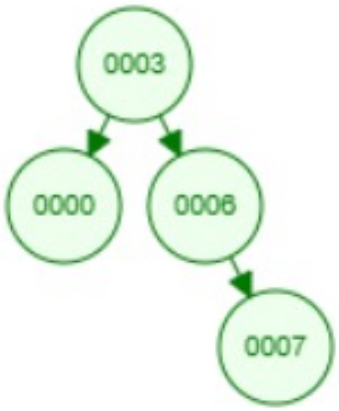

Figure 5: After the rebuilding of whole tree

4.2. Amortized analysis

[5]

4.2.1 Insertion

Defining the imbalance of a node v to be the absolute value of the difference in size between its left node ans right node minus 1,or 0, whichever is greater. In other words: I(v):max(|left(v)-right(v)|-1,0) Immediately after rebuilding a subtree rooted at v, I(v) = 0.
let m be the root of a subtree immediately after rebuilding. h(m)=log(|m|+1).If there are $\Omega(|m|)insertions.I(v) \in \Omega(|m|)$,
$h(v) = h(m) + \Omega(|m|)and$
$log(|v|) \geq$ log(|m|+1) +1.
Since I(v) $\in \Omega(|v|)$
before rebuilding,there were $[\Omega(|v|)$
inesrtions into the subtree rooted at v that did not result in rebuilding .Each of these insertions can be performed in O(logn) time. The final insertion that causes rebuilding costs O(|v|). Using **aggregate analysis** it becomes clear that the amotized cost of insertion is O(logn)

$$\frac{\Omega(|v|)O(\log n) + O(|v|)}{\Omega(|v|)} = O(\log n)$$

4.2.2 Deletion

Suppose a scapegoat tree is just rebuild and there are n nodes in scapegoat tree. Then at most n/2-1 deletions can be performed before the tree must be rebuild.Each of this deletion takes O(logn) time.Then the n/2 deletion causes the tree to be rebuilt and takes O(logn)+O(n) time.Using aggregate analysis it becomes clear it becomes clear that the amortized cost of a deletion is O(logn).

$$\frac{\sum_1^{n/2} O(\log n) + O(n)}{n/2} = \frac{\frac{n}{2} O(\log n) + O(n)}{n/2} = O(\log n)$$

4.3. Algorithms

[4]. 1)INSERTION

```
//Algorithm: Insertion in a Scapegoat Tree
If the tree is empty (root is null), set `root` to `newNode` and return.
Initialize a counter `depth` to 1 and a reference `current` to the root of the tree.
Traverse the tree to find the appropriate position for `newNode`:
   a. While `current` is not null:
      i. If `newNode` < `current`, move to the left child: `current = current.left`
      ii. Else, if `newNode` > `current`, move to the right child: `current = current.right`
      iii. Increment `depth` by 1.
Once the appropriate position is found, insert `newNode`:
   a. If `newNode` < `current`, set `current.left` to `newNode`
   b. Else, if `newNode` > `current`, set `current.right` to `newNode`
Increment `depth` by 1 to account for the new node.
Check if the subtree rooted at `current` violates the α-height-balance property:
   - If the height of the right subtree of `current` is more than α times the height of the left
   subtree, or vice versa, proceed to the next step.
Find the scapegoat ancestor:
   a. Start from `newNode` and move up to the root:
      i. If the height of the right subtree of the current node is more than α times the height of
      the left subtree, set scapegoat` to the current node.
      ii. Move to the parent node.
If a scapegoat is found, rebalance the subtree rooted at the scapegoat:
   - Rebuild the subtree to achieve balance. This can be done by finding values in sorted order
   and recursively selecting the median as the new root.
Update MaxNodeCount if necessary:
   - If the new node count exceeds MaxNodeCount, update MaxNodeCount with the current node count.
Return the updated tree.
//End of Algorithm.
```

Figure 6: Insertion

2)Deletion

```
//Algorithm: Deletion in a Scapegoat Tree
Start at the root of the tree: `current = root`, and initialize a parent reference `parent` to
null.
Traverse the tree to find the node with value `targetValue` to be deleted:
   a. While `current` is not null and `current.value` is not equal to `targetValue`:
      i. Set `parent` to `current`.
      ii. If `targetValue` < `current.value`, move to the left child: `current = current.left`
      iii. Else, if `targetValue` > `current.value`, move to the right child: `current = current.
      right`
If `current` is null, the target value was not found in the tree. Exit the algorithm.
Determine the type of deletion based on the number of children of the node to be deleted
(`current`):
   a. Case 1: `current` is a leaf node (has no children):
      - Set the corresponding child pointer of `parent` to null.
   b. Case 2: `current` has one child:
      - Replace `current` with its child in the appropriate position relative to `parent`.
   c. Case 3: `current` has two children:
      - Find the in-order predecessor (the largest value in the left subtree of `current`) or
      in-order successor (the smallest value in the right subtree of `current`).
      - Replace `current.value` with the value of the in-order predecessor or successor.
      - Recursively delete the node containing the in-order predecessor or successor.
Check if the tree needs to be rebalanced after deletion:
   - If `NodeCount <= α * MaxNodeCount`, trigger a full tree rebalance at the root.
Return the updated tree.
//End of Algorithm.
```

Figure 7: Deletion

3)search

```
//Algorithm: Searching in a Scapegoat Tree
Start at the root of the tree: `current = root`.
Traverse the tree to find the node with value `targetValue`:
   a. While `current` is not null and `current.value` is not equal to `targetValue`:
      i. If `targetValue` < `current.value`, move to the left child: `current = current.left`
      ii. Else, if `targetValue` > `current.value`, move to the right child: `current = current.
      right`
If `current` is null, the target value was not found in the tree.
If `current.value` is equal to `targetValue`, the value was found in the tree.
Return `current` (which may be either the node with the target value or null if the value was not
found).
//End of Algorithm.
```

Figure 8: Search

4)Rebuild

```
1  //Algorithm: Rebuilding a Scapegoat Tree
2  Traverse the entire tree in-order and store the values in a sorted array.
3  Find the median value in the sorted array. This will be the new root of the subtree.
4  Create left and right subtrees from the elements to the left and right of the median value,
   respectively.
5  Recursively repeat steps 2 and 3 for the left and right subtrees, using the median values in their
   respective sub-arrays as the new roots.
6  Assemble the tree with the new root and balanced subtrees.
7  Return the balanced tree.
8  //End of Algorithm.
```

Figure 9: Rebuild

5. APPLICATION

Applications of scapegoat tree includes that it is faster than other BST trees operations.
We have compared our Scapegoat tree with AVL and plotted the graph of Insert(x),Delete(x),Search(x) and Average Depth.
1)INSERT(x): Performance analysis Figure 10 demonstrates add(x)'s different performance for each BST implementation and integer set: Both Ordered and Unordered insertion for AVL is considerably better than SG tree. It achieves that by performing 1 left rotation in every insertion. That essentially forms a perfect tree, which in turn minimizes insertion times. SG insertion appears to behave much better, as ALPHA moves towards the two higher values; that is due to the following reason:
a)The higher ALPHA becomes, the less frequent subtree rebalancing operations become to reduce the depth(which can be related with Figure 1 graph) ,hence the decreasing times.
GRAPH

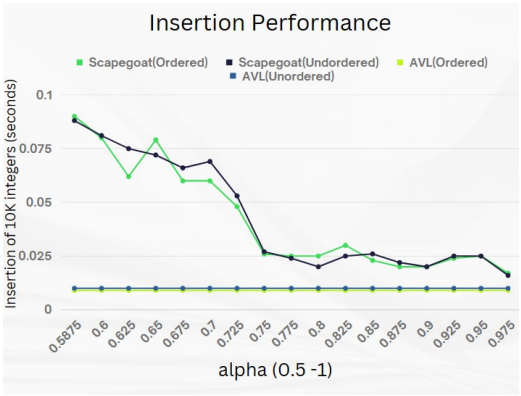

Figure 10: INSERT

2)DELETE(x): performance analysis Figure 3 presented compare how two tree implementations, SG and AVL, perform when items are removed in the same order they were inserted. In Figure 3, during ordered removal, the SG tree demonstrates superior performance over the AVL tree. It excels in both ordered and unordered initial insertions. The reason for the same is explained in the amortised time analysis.
GRAPH

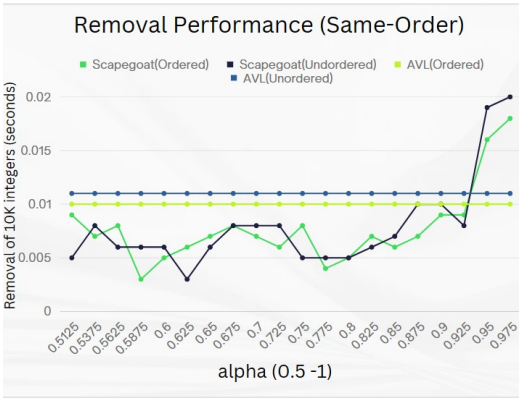

Figure 11: Deletion

3)Search(x): performance analysis Increasing ALPHA exponentially increases average search time for SG trees,

for ordered insertions. In spite of that, at low ALPHA values, the SG implementation outperforms the AVL one, for ordered insertions.
GRAPH

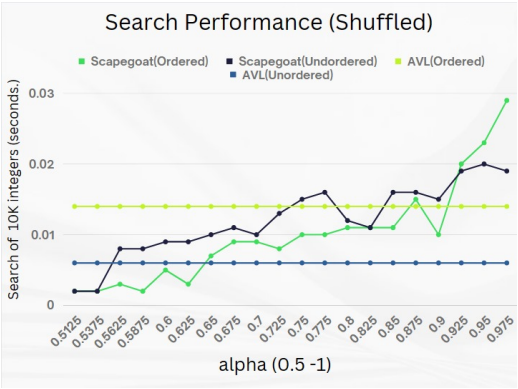

Figure 12: SEARCH

4)AVL depth and rotations At first, we computed the average depth of AVL and SG trees after inserting 10K integers, both ordered and Unordered sets. The AVL tree achieves a depth 14 with both ordered and unordered sets of integers. The SG tree acts differently, based on the ALPHA coefficient. At low ALPHA, the SG tree essentially behaves like an AVL (both have the same depths '14' at ALPHA = 0.5125) , where every node's child contains, on average, half of the node's subtree , while at high ALPHA, it behaves like an unbalanced BST.
GRAPH [3].

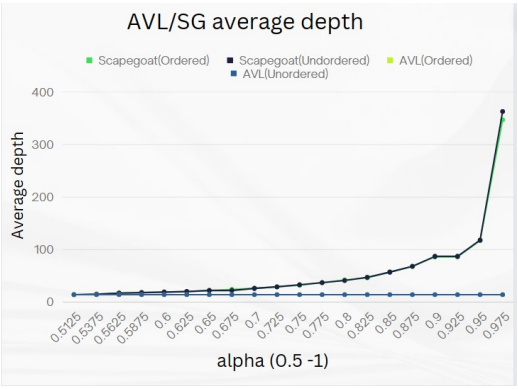

Figure 13: DEPTH

6. Conclusions

Scapegoat Trees offer an efficient solution for maintaining balance in a binary search tree during insertions and deletions. By incorporating the -height-balance property and the concept of scapegoats, these trees provide a reliable data structure for dynamic operations.
Key takeaways from Scapegoat Trees include:
1) Insertion and Deletion Efficiency: While deletion is generally easier than insertion in Scapegoat Trees, both operations are handled effectively. The additional property MaxNodeCount ensures that rebalancing occurs when necessary.
2) Balancing Strategy: The use of scapegoats allows for rebalancing a subtree when a node violates the ALPHA-height-balance property. This ensures that the tree maintains a balanced structure over time.
3) Rebuilding Process: The rebuilding algorithm, which involves sorting and recursively selecting median values, restores perfect balance to subtrees rooted at scapegoats.
4) Time Complexity: While worst-case scenarios for insertion and deletion are O(n), amortized analysis demonstrates that these operations typically perform in O(log n) time, making Scapegoat Trees suitable for dynamic data sets.

In conclusion, Scapegoat Trees strike a balance between simplicity and efficiency, providing an elegant solution for maintaining balance in binary search trees. Their design considerations make them a valuable tool in scenarios where dynamic data operations are prevalent.

Acknowledgements

To ANIL SHUKLA Sir
TA:E HARSHIT KUMAR Sir

References

[1] PHUONG DAO. A study and implementation of scapegoat trees.

[2] JUSTIN WYSS GALLIFENT. Cmsc 420:scapegoat trees.

[3] Igal Galperin and Ronald L Rivest. Scapegoat trees. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 165–174, 1993.

[4] PAT MORIN. Scapegoat trees. *libreTexts*, 2006.

[5] VASILOS I. VENIERIS. Scapegoat trees:a comparative performance assessment.