A STUDY AND AN IMPLEMENTATION OF SCAPEGOAT TREES

Honour Project Report
Carleton University, School of Computer Science
Submitted by Phuong Dao, 100337321
Supervised by Dr. Anil Maheshwari

August 15, 2006

ABSTRACT

This report describes the study and implementation of Scapegoat trees, weight and loosely-height-balanced binary search trees, first proposed in the paper:"Scapegoat Trees"[Igal Galperin and Ronald L. Rivest, 1993]. The 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. In this report, the correctness and complexity of operations on Scapegoat trees will be thoroughly and mainly discussed. Also, the test cases from Demo on the operations will be presented. Finally, an application of techniques used in Scapegoat trees will be discussed and conclusions will be drawn.

ACKNOWLEDGEMENTS

I would like to thank very much to Dr. Anil Maheshwari for his guidance and enthusiasm not only in supervising my project but also in inspiring me to further study in Computer Science and the most to my parents and my sister who always support me and be there for me.

Contents

Chapter 1

INTRODUCTIONS

There are two main balanced binary search tree schemes: 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 by Bayer [1], Guibas and Sedgewick [4] 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 subtrees rooted at siblings for every node in the tree approximately equal. Nievergelt and Reingold [6] first introduced a tree using such scheme to implement Search and Update operations in $O(\log n)$ time as well. 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 subtree and rebuilding the subtree, it also ensures weight-balanced after an update operation. Before going deep into operations in Scapegoat trees, let's get ourselves familiar with notations and definitions of balanced binary search trees.

1.1 Binary Search Tree Notations

If n is a node in a Scapegoat tree T then

- **n.key** 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 subtree 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

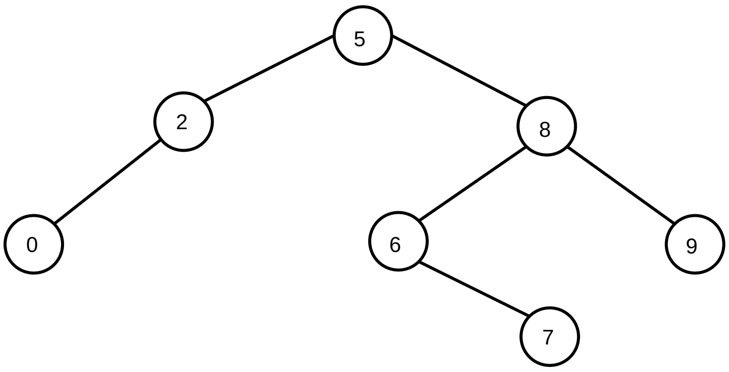


Figure 1.1: A binary search tree with $\alpha = 0.6$

- $n.size$ refers to the number of nodes of subtree rooted at n. This notation is used for proofs of correctness and time complexity
- $n.h_\alpha$ is computed as $n.h_\alpha = \lfloor \log_{1/\alpha}(n.size) \rfloor$
- $n.size()$ is the procedure to compute the size of the binary search tree rooted at node n. This notation is used in the descriptions of operations on Scapegoat trees.

What is in bold font, for example **n.key**, is what is really stored at each node. The others, except for the procedures, are just values used for discussions in proofs of correctness and time complexity of operations on Scapegoat trees. Below are notations related to a binary search tree T:

- **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 $T.size$
- $T.h_\alpha$ is computed as following: $T.h_\alpha = \lfloor \log_{1/\alpha}(T.size) \rfloor$
- $T.height$ refers to the height of T or the longest path in terms of edges from $T.root$ to some leaf in T

Similar to notations of a node, what is in bold font is what is stored in tree T. $T.height$ is just used in concepts and proofs.

1.2 Weight Balanced Binary Search Tree

Definition 1.2.1. *A node n in a binary search tree is α-weight-balanced for some α such that $1/2 \leq \alpha < 1$ if $(n.left).size \leq \alpha \cdot n.size$ and $(n.right).size \leq \alpha \cdot n.size$.*

Table 1: Examples of notations for a binary tree node 8 in Figure 1.1

n.key	n.parent	n.sibling	n.left	n.right	n.height	n.depth	n.size	$n.h_\alpha$	n.size()
8	5	2	6	9	2	1	4	2	4

Table 2: Examples of notations for a binary search tree in Figure 1.1

T.root	T.size	T.height	$T.h_\alpha$
5	7	3	3

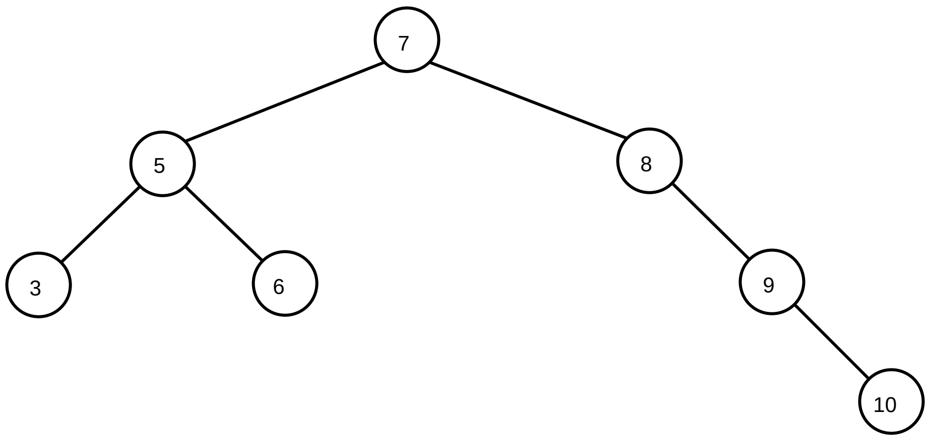


Figure 1.2: A weight-unbalanced binary search tree with $\alpha = 0.6$

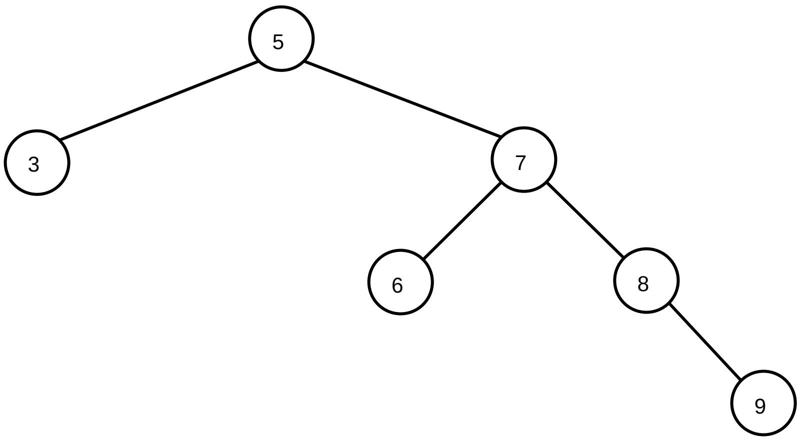


Figure 1.3: A height-unbalanced binary search tree with $\alpha = 0.55$

Example 1.2.1. *Node 8 in Figure 1.2 is not 0.6-weight-balanced because its right subtree has 2 nodes which is greater than* $0.6 * 3 = 1.8$ *where 3 is the total nodes of the subtree rooted at 8.*

Definition 1.2.2. *A binary search tree T is α-weight-balanced for some α such that* $1/2 \leq \alpha < 1$ *if all of its nodes are α-weight-balanced.*

Example 1.2.2. *The tree in Figure 1.2 is not 0.6-weight-unbalanced binary search tree because it contains node 8 which is not 0.6-weight-balanced.*

1.3 Height Balanced Binary Search Tree

Definition 1.3.1. *A binary search tree is α-height-balanced for some α such that* $1/2 \leq \alpha < 1$ *if T.height $\leq T.h_\alpha$.*

Example 1.3.1. *The tree in Figure 1.2 is 0.6-height-balanced binary search tree because it has 7 nodes and its height is* $3 \leq \lfloor \log_{1/0.6}(7) \rfloor = 3$.

1.4 Scapegoat Tree Related Definitions

Definition 1.4.1. *A node n in a binary search tree T is a* **deep** *node for some α such that* $1/2 \leq \alpha < 1$ *if n.depth $> T.h_\alpha$.*

Definition 1.4.2. *A node s is called* **Scapegoat** *node of a newly inserted deep node n if s is an ancestor of n and s is not α-weight-balanced.*

Example 1.4.1. *If we have just inserted node 9 into the tree T in Figure 1.3 then node 9 is a deep node because its height is 3 which is greater than $T.h_\alpha = \lfloor \log_{1/0.55}(6) \rfloor = 2$. And in this case, root 5 would be chosen as Scapegoat node.*

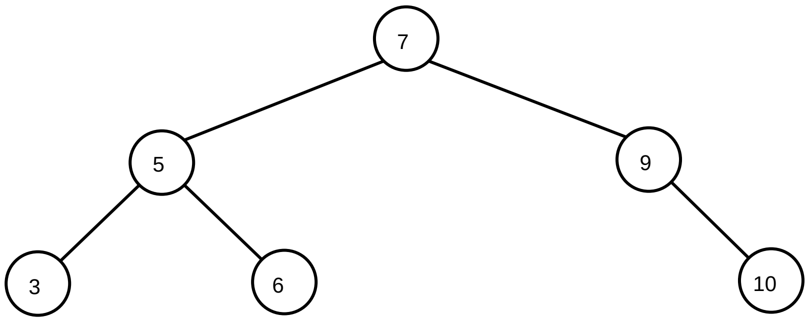


Figure 1.4: An incomplete binary search tree

Definition 1.4.3. *A binary search tree is loosely α-height-balanced for some α such that $1/2 \leq \alpha < 1$ if $T.height \leq (T.h_\alpha + 1)$.*

Example 1.4.2. *The tree T in Figure 1.3 is a good example of loosely 0.55-height-balanced with $3 = T.height \leq (T.h_\alpha + 1) = (\lfloor \log_{1/0.55}(6) \rfloor + 1) = 3$*

Definition 1.4.4. *A binary search tree T is* **complete** *if inserting any node into T increases its height.*

Example 1.4.3. *The tree in Figure 1.4 is incomplete. But after inserting node 8 into the tree, it becomes complete.*

With all the above definitions, we could now envision the notion of a Scapegoat tree. A Scapegoat tree is just a regular binary search tree in which the height of the tree is always loosely α-height-balanced. Such balance is maintained after an Insert operation of a deep node or a Delete operation that leads to $T.size < \alpha \cdot T.maxsize$. In the case of inserting a deep node, a Scapegoat node s will be detected. Both cases will result in rebuilding the subtree rooted at s into a 1/2-weight-balanced binary search one.

Chapter 2

OPERATIONS ON SCAPEGOAT TREES

This chapter will present how to do Search, Insert, Delete operations on a Scapegoat tree. An update i.e. Insert or Delete will come with an example which is taken out from the Demo for this project. So before going deep into the operations, let's take a look at a Scapegoat tree from Demo:

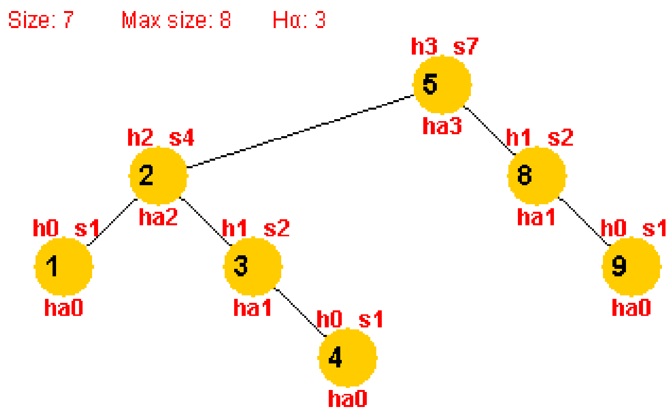


Figure 2.1: A sample Scapegoat tree from the Demo

Figure 2.1 presents a Scapegoat tree T with $\alpha = 0.57$ of size 7, $h_\alpha = 3$ and T had size(maxSize) 8 before node 10 was just deleted. These three indexes are at the top of the tree. Notice that for each node in T, there are also three indexes surrounding it. The index prefixed by s is the size of the subtree rooted at that node while the one prefixed by h is the height of the node in T. The index prefixed by ha below each node is the h_α of that node.

2.1 How to search for a key

The Search operation in Scapegoat tree T is done like regular Search in a binary search tree. As we will prove later that 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 $T.height < \lfloor \log_{1/\alpha}(T.size) \rfloor + 1$ then worst-case running time is $O(\log(T.size))$ time. Below is the simple recursive procedure Search:

Procedure 2.1.1 $Search(root, k)$

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

```
1: if $root$ = null or $root.key = k$ then
2:     return $root$
3: else if $k \leq root.left.key$ then
4:     return $Search(root.left, k)$
5: else
6:     return $Search(root.right, k)$
7: end if
```

2.2 How to insert a new key

The Insert in Scapegoat tree T is also done like the regular Insert operation in a binary search tree when T is still height-balanced. When T is not height-balanced then by Main Theorem (3.1.1), a Scapegoat node s will be detected and rebuilding will be taken place at the subtree rooted at s. Proof of the Main Theorem also shows how to find the Scapegoat given newly inserted node n that makes T height-unbalanced. The Scapegoat s will be the first ancestor of n such that $s.height > s.h_\alpha$. The procedure $FindScapegoat(n)$ implemented in the following will return the Scapegoat s giving the newly inserted deep node n:

After the Scapegoat s is detected if applicable, the procedure $RebuildTree(size, root)$ will get called. $RebuildTree$ will rebuild a new 1/2-weight-balanced subtree from the subtree rooted at Scapegoat node returned. Details of procedure $RebuildTree$ will be presented in the rebuilding section. 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 $T.h_\alpha$ to detect whether a newly inserted node is a deep node or not:

Example 2.2.1. *Figure 2.2 shows a Scapegoat tree T with $\alpha = 0.57$ before inserting node 29. If node 29 is inserted, it would be the right child of node 22 and because the height of the tree is $4 > T.h_\alpha = 3$ then node 29 is a deep node. Scapegoat node 8 will be detected and the whole tree T will be rebuilt into 1/2-weight-balanced binary search tree in Figure 2.3.*

Procedure 2.2.1 *FindScapegoat*(n)

Input: n is a node and $n \neq$ **null**

Output: A node s which is a parent of n and is a Scapegoat node

```
1: $size = 1$
2: $height = 0$
3: while ($n.parent$ <> null) do
4:    $height = height + 1$
5:    $totalSize = 1 + size + n.sibling.size()$
6:    if $height > \lfloor \log_{1/\alpha}(totalSize) \rfloor$ then
7:       return $n.parent$
8:    end if
9:    $n = n.parent$
10:   $size = totalSize$
11: end while
```

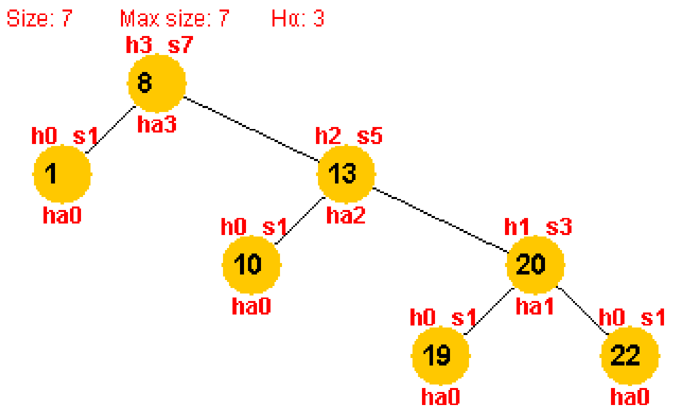


Figure 2.2: A Scapegoat tree before insertion

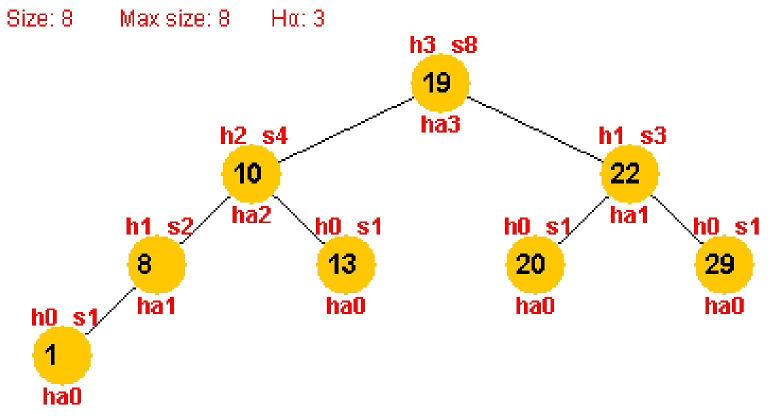


Figure 2.3: A Scapegoat tree after insertion

Procedure 2.2.2 *Insert*(k)

Input: The integer key k

Output: **true** if the insertion is successful, **false** if there exists a node n such that $n.key = k$

```
1: $height = InsertKey(k)$
2: if $height = -1$ then
3:     return false;
4: else if $height > T.h_\alpha$ then
5:     $scapegoat = FindScapegoat(Search(T.root, k))$
6:     $RebuildTree(n.size(), scapegoat)$
7: end if
8: return true
```

2.3 How to delete a key

Deleting a node in a Scapegoat tree T is done by first deleting the node in a regular binary search tree then compare T's current size($T.size$) to T's maximum size($T.maxSize$) that T obtained since its latest total rebuilding. If $T.size < \alpha * T.maxSize$ (1) then the whole tree T will be rebuilt into 1/2-weight-balanced binary search tree and $T.maxSize = T.size$. The intuition behind that is when (1) is satisfied, T might not be α-weight-balanced so it might not be loosely α-height-balanced either. But we need to maintain T α-height-balanced or loosely α-height-balanced then T needs to be rebuilt.

Procedure 2.3.1 *Delete*(k)

Input: The integer key k

Output: There is no node n in T such that $n.key = k$

```
1: $deleted = DeleteKey(k)$
2: if $deleted$ then
3:     if $T.size < (T.\alpha \cdot T.maxSize)$ then
4:         $RebuildTree(T.size, T.root)$
5:     end if
6: end if
```

Example 2.3.1. *Figure 2.4 shows a Scapegoat tree before a series of deletions of node 9, 10, 16 with $\alpha = 0.57$. As seen, the tree is still 0.57-height-balanced but after the series of deletion it is loosely 0.57-height-balanced but not 0.57-height-balanced in Figure 2.5. But if we remove one more node, node 1, the whole tree would be rebuilt into 1/2-weight-balanced in Figure 2.6.*

2.4 Rebuild Scapegoat Subtree

The procedures below are basically the same as the procedures of rebuilding 1/2-weight-balanced tree in the original paper :"Scapegoat Trees"[Igal Galperin and Ronald L. Rivest, 1993] [3] but with adaptation to my notations and an elimination of a dummy variable.

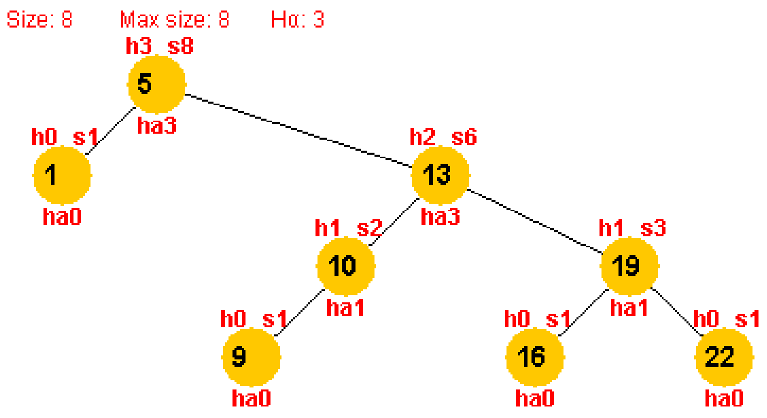


Figure 2.4: A Scapegoat tree before series of deletion

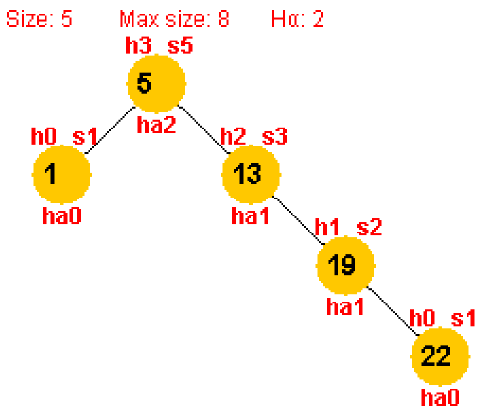


Figure 2.5: A Scapegoat tree after series of deletions

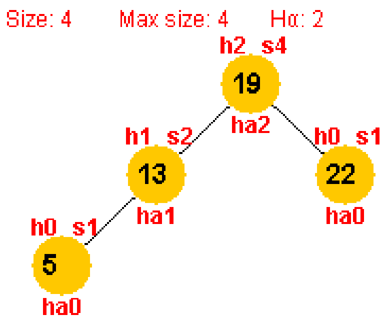


Figure 2.6: A Scapegoat tree after series of deletions and rebuilding

The procedures are presented here because later I will provide proofs of time complexity of rebuilding procedures which are not clearly mentioned in the original paper. The idea of rebuilding 1/2-weight-balanced tree is straight. First flatten tree into the list of nodes in nondecreasing order of their keys. Then divide the list into three parts, among which the list of first half number of nodes and the second half number of nodes will be recursively rebuilt into 1/2-weight-balanced trees and the new root would be the middle node. Its left subtree is the 1/2-weight-balanced tree returned from recursive call for first half list of nodes. Its right subtree is the 1/2-weight-balanced tree returned from recursive call for second half list of nodes. The procedure $Flatten\_Tree(root, head)$ will return the list of nodes of a binary search tree rooted at *root* in nondecreasing order of their keys appending by the list headed by a node *head*. The nodes in the list returned are linked by the right fields:

Procedure 2.4.1 $Flatten\_Tree(root, head)$

Input: *root* is the root of some tree T

Output: The list of all nodes in T in nondecreasing order in terms of their keys headed by node *head*. The node that contains the smallest key in the tree T will be returned.

```
1: if $root$ = null then
2:   return $head$
3: end if
4: $root.right = Flatten\_Tree(root.right, head)$
5: return $Flatten\_Tree(root.left, root)$
```

The procedure $Build\_Height\_Balanced\_Tree(size, head)$ will build a 1/2-weight-balanced tree from the flatten list of all the nodes in a binary search tree T. The procedure will return the last node of the flatten list. Now, the procedure $Rebuild\_Tree(size, scapegoat)$ just makes use of $Flatten\_Tree(root, head)$ procedure to flatten the subtree rooted at *scapegoat* and $Build\_Height\_Balanced\_Tree(size, head)$ to rebuild the flatten list into 1/2-weight-balanced binary search tree T. Because the call $Build\_Height\_Balanced\_Tree(size, head)$ will return the last node of the flatten list so in order to retrieve the root of the 1/2-weight-balanced tree then just traverse the parents of node *head* until we reach the root and this could be done in $O(\log(T.size))$ time because T is 1/2-weight-balanced then T is also 1/2-height-balanced.

Procedure 2.4.2 *Build_Height_Balanced_Tree*$(size, head)$

Input: The list of *size* nodes in nondecreasing order in terms of their keys headed by node *head*

Output: A 1/2-weight-balanced tree built from the list above. The last node of the list will be returned.

```
if $size = 1$ then
    return $head$
else if $size = 2$ then
    $(head.right).left = head$
    return $head.right$
end if
$root = (Build\_Height\_Balanced\_Tree(\lfloor (size-1)/2 \rfloor, head)).right$
$last = Build\_Height\_Balanced\_Tree(\lfloor (size-1)/2 \rfloor, root.right)$
$root.left = head$
return $last$
```

Procedure 2.4.3 *Rebuild_Tree*$(size, scapegoat)$

Input: A scapegoat node *scapegoat* that is the root of a subtree of *size* nodes

Output: A 1/2-weight-balanced subtree built from all the nodes of the subtree rooted at *scapegoat*. The root of the rebuilt subtree will be returned

```
$head = Flatten\_Tree(scapegoat,$ null$)$
$Build\_Height\_Balanced\_Tree(size, head)$
while head.parent!=null do
    $head = head.parent$
end while
return $head$
```

Chapter 3

CORRECTNESS

3.1 Main Theorem

Theorem 3.1.1. *If T is an α-weight-balanced binary search tree, then T is α-height-balanced as well.*

Proof. We could prove the other way around: If T is not α-height-balanced then T is not α-weight-balanced either. If T is not α-height-balanced then there is node n_0 such that $n_0 > T.h_\alpha$ (2). Denote the deepest ancestor of n_0 which is not α-height-balanced as n_i or let $n_1, ..., n_i$ be ancestors of n_0 such that n_1 is the parent of n_0 and so on, then n_i is the first ancestor of n_0 that satisfies (2). Such n_i always exists because $T.root$ always satisfies (2). By the way of choosing such n_i, the following inequalities are satisfied:

$$\begin{array}{rrcl} & i & > & n_i.h_\alpha \\ \Rightarrow & i & > & \lfloor \log_{1/\alpha}(n_i.size) \rfloor \\ \Rightarrow & i & > & \log_{1/\alpha}(n_i.size) \end{array} \tag{3}$$

and

$$\begin{array}{rrcl} & i-1 & \leq & n_{i-1}.h_\alpha \\ \Rightarrow & i-1 & \leq & \lfloor \log_{1/\alpha}(n_{i-1}.size) \rfloor \\ \Rightarrow & i-1 & \leq & \log_{1/\alpha}(n_{i-1}.size) \\ \Rightarrow & 1-i & \geq & -\log_{1/\alpha}(n_{i-1}.size) \end{array} \tag{4}$$

(3) + (4) gives

$$\begin{array}{rrcl} & 1 & > & \log_{1/\alpha}(n_i.size) - \log_{1/\alpha}(n_{i-1}.size) \\ \Rightarrow & 1 & > & \log_{1/\alpha} \dfrac{n_i.size}{n_{i-1}.size} \end{array}$$

Due to $\frac{1}{2} \leq \alpha < 1$ or $\dfrac{1}{\alpha} > 1$ and $n_i.size > n_{i-1}.size$
Then

$$n_{i-1}.size > \alpha \cdot n_i.size$$

Therefore, the node n_i is not α-weight-balanced or T is not α-weight-balanced. □

The proof above shows us not only how to find the Scapegoat node s for rebuilding the subtree rooted at s when a deep node n is detected but also such node s always exists. Let d be the distance in term of edges from n to s, s is the deepest ancestor of n such that $d > s.h_\alpha$.

3.2 Insert

Lemma 3.2.1. *A 1/2-weight-balanced binary search tree T has the smaller than or equal height of any binary search tree T' of the same size.*

Proof. First notice that

$$\begin{array}{rcl} \lfloor \log_2(T'.size) \rfloor & \leq & T'.height \\ \Rightarrow \lfloor \log_2(T.size) \rfloor & \leq & T'.height \end{array}$$

Due to $T.size = T'.size$
And by Main Theorem (3.1.1), T is 1/2-height-balanced:

$$T.height \leq \lfloor \log_2(T.size) \rfloor$$

Therefore,

$$T.height \leq T'.height$$

□

Lemma 3.2.2. *If $T.root$ is not α-weight-balanced node then its heavy subtree contains at least 2 more nodes than its light subtree.*

Proof. Denote the sizes of the heavy, light subtrees of the root, and the whole tree by h,l, and t, respectively. We have:

$$h + l + 1 = t$$

and

$$\begin{array}{rcl} h & > & \alpha \cdot t \\ \Rightarrow h & > & \alpha \cdot (h + l + 1) \\ \Rightarrow h \cdot (1 - \alpha) & > & \alpha \cdot l + \alpha \\ \Rightarrow h \cdot \dfrac{1 - \alpha}{\alpha} & > & l + 1 \end{array}$$

Since $1/2 \leq \alpha < 1$ or $1 < \frac{1}{\alpha} \leq 2$ and $0 < 1 - \alpha \leq 1/2$ then $\frac{1-\alpha}{\alpha} < 1$.
Therefore,

$$h \; > \; h \cdot \frac{1 - \alpha}{\alpha} \; > \; l + 1$$

and h and l are positive integers then $h \geq l + 2$ □

Lemma 3.2.3. *If T is not α-weight-balanced and T contains only one node at depth $T.height$ then rebuilding T decreases its height.*

Proof. Let n be the only node at depth $T.height$. Let T'_l, T'_h be the light and heavy subtrees of a subtree T' rooted at Scapegoat node s which is α-weight-unbalanced and is an ancestor of n, by proof of Main Theorem (3.1.1), s always exists. If $n \in T'_l$ then T' after removing n is not complete tree of height $T'.height - 1$ because T'_h contains at least 2 more nodes than T'_l by Lemma(3.2.2) but $T'_h.height < T'_l.height$. In the other case $n \notin T'_l$ or $n \in T'_h$, T' is still not complete subtree of height $T'_l.height - 1$ after removing n because T'_h has at least 2 more nodes than T'_l but $T'_h.height > T'_l.height$. So rebuilding tree T results in rebuilding T' into 1/2-weight-balanced binary search tree that causes it decrease its height by Lemma (3.2.1) or T decreases its height. □

Theorem 3.2.4. *If a Scapegoat tree T was created from a 1/2-weight-balanced tree by a sequence of Insert operations then T is α-height-balanced.*

Proof. This proof is done by induction. If there are no Insert operations then by Main Theorem (3.1.1), the theorem follows. Suppose that it is true that if a Scapegoat tree T was created from a 1/2-weight-balanced tree by any sequence of n Insert operations then T is α-height-balanced. We have to prove it also holds for an extra Insert done after those sequences. If the extra Insert does not cause T to rebuild, T is still α-height-balanced because it was and the height of newly inserted node is not over $T.h_\alpha$. If the extra Insert does cause T to rebuild or the depth of the newly inserted node n: $n.depth > T.h_\alpha$ and n is the only node at depth $T.h_\alpha + 1$ then by Lemma (3.2.3), the theorem is followed. □

3.3 Delete

Lemma 3.3.1. *Let T be a α-weight-balanced binary search tree and let T' be the tree after inserting a node n into T then $T'.height \leq max(T'.h_\alpha, T.height)$*

Proof. If rebuilding is not triggered after the insertion of n, then the depth of n is at most $T'.h_\alpha$ or $T'.height \leq T'.h_\alpha$ or $T'.height \leq max(T'.h_\alpha, T.height)$. If the insertion of n causes T to rebuild or $n.depth > T'.h_\alpha$, there are still two cases to take care. In first case where there were already some other nodes at $n.depth$, since rebuilding tree into 1/2-weight-balanced does not make the tree deeper by Lemma (3.2.1) or $T'.height \leq T.height \leq max(T'.h_\alpha, T.height)$. In the other case where n is the only node at $n.depth > T'.h_\alpha$, by Lemma (3.2.3), rebuilding tree does not make the tree deeper, we're done. □

Lemma 3.3.2. *Let T be a loosely α-height-balanced binary search tree such that $T.height = T.h_\alpha + 1$ and let T' be the tree after insert a node n into T such that $T'.h_\alpha = T.h_\alpha + 1$ then T' is α-height-balanced.*

Proof. We have

$$\begin{aligned} T.height &= T.h_\alpha + 1 \\ T'.h_\alpha &= T.h_\alpha + 1 \end{aligned}$$

then

$$T.height = T'.h_\alpha$$

Then Lemma (3.3.1) gives

$$\begin{array}{rrcl} & T'.height & \leq & max(T'.h_\alpha, T.height) \\ \Rightarrow & T'.height & \leq & max(T'.h_\alpha, T'.h_\alpha) \\ \Rightarrow & T'.height & \leq & T'.h_\alpha \end{array}$$

Therefore, T' is α-height-balanced by Definition (1.3.1), $\square$

Lemma 3.3.3. *A Scapegoat tree T built from a sequence of Insert and Delete operations from an empty tree is always loosely α-height-balanced.*

Proof. Divide the sequence of Insert and Delete operations into subsequences of consecutive operations $o_1, ..., o_k$ such that at the end of each sequence the tree has to be rebuilt and during the executions of operations, no rebuilding is taken place. So we need to show that during such sequence of operations, the tree is α-height-balanced or loosely α-height-balanced.
In the case that during any sequence of Insert and Delete operations, if $T.h_\alpha$ does not change then $max(T.h_\alpha, T.height)$ is not increased and the tree is still at least loosely α-height-balanced. This is because Delete operation could not increase $max(T.h_\alpha, T.height)$ nor the Insert operation:
Let T' be the tree after inserting a node into T which is a tree after an operation o_i $(1 \leq i \leq k)$ in the sequence. Now we have to prove that $max(T'.h_\alpha, T'.height) \leq max(T.h_\alpha, T.height)$. By Lemma (3.3.1):

$$\begin{array}{rlcl} & T'.height & \leq & max(T'.h_\alpha, T.height) \\ \Rightarrow & max(T'.h_\alpha, T'.height) & \leq & max(T'.h_\alpha, T.height) \\ \Rightarrow & max(T'.h_\alpha, T'.height) & \leq & max(T.h_\alpha, T.height) \end{array}$$

In the other case, $T.h_\alpha$ is changed during sequence of Insert and Delete operations. Denote $o'_1, ..., o'_l$ be the sequence of operations that change $T.h_\alpha$ and let T' be the tree after inserting a node into T which is a tree before some operation o'_i $(1 \leq i \leq l)$. For an Insert operation, if T is α-height-balanced then T' is loosely α-height-balanced because $T.h_\alpha$ is increased at most 1 and the same for $T.height$. Otherwise if T is loosely α-height-balanced but not α-height-balanced, by Lemma (3.3.2), T' is α-height-balanced. For a Delete operation, notice that there are no two consecutive Delete operations in that sequence. This is due to if a Delete operation change $T.h_\alpha$, it will decrease $T.h_\alpha$ by 1 or the cutting size is $\alpha \cdot T.maxSize$. If there is another Delete operation right, and we know the second will again cause $T.h_\alpha$ decrease by 1 then rebuilding should be taken between those two Delete operations. This violates our assumption. So a Delete operation should be performed on α-height-balanced tree. As we know a Delete operation does not increase $T.height$ nor T_α then the tree after will be at most loosely α-height-balanced. This completes the proof. $\square$

Chapter 4

TIME COMPLEXITY

4.1 Find Scapegoat and Rebuild

Theorem 4.1.1. *The time to find the Scapegoat node s is $O(s.size)$ time.*

Proof. The $FindScapegoat(n)$ procedure is triggered when n, a newly inserted node, has $n.depth > T.h_\alpha$. Let $n_0, n_1, ..., n_i$ be the sequence of accessors of n the procedure examined where n_i is s, then all nodes $n_0, n_1, ...n_i$ are examined once. Moreover, all nodes in the other subtrees of $n_0, n_1, ...n_i$ are examined once by $size()$ procedure. Therefore, all nodes in the tree rooted at s is examined once or the time to find the Scapegoat node s is $O(s.size)$ time. □

Lemma 4.1.2. *The call $Flatten\_Tree(root, head)$ takes $O(root.size)$ time.*

Proof. Notice that every node of the tree will be visited at most one because $Flatten\_Tree$ is recursively called for two children of $root$ and will not come back to the parent nodes. Therefore, the worst-case complexity is $O(root.size)$ time. □

Lemma 4.1.3. *The call $Build\_Height\_Balanced\_Tree(size, head)$ takes $O(size)$ time.*

Proof. It is similar to the procedure $Flatten\_Tree(root, head)$ where every node of the tree will be visited at most one. Procedure $Build\_Height\_Balanced\_Tree(size, head)$ is recursively called for 2 halves of the original list and will not be called to process the visited nodes in the list. Therefore, the worst-case complexity is $O(size)$ time. □

Theorem 4.1.4. *The worst-case complexity of $Rebuild\_Tree(size, scapegoat)$ is $O(size)$ time.*

Proof. The procedure makes use of two procedures $Flatten\_Tree$ and $Build\_Height\_Balanced\_Tree$ whose worst-case complexity is $O(size)$ time and at the end, traversing parents of last node returned by procedure $Flatten\_Tree$ to find the root of newly rebuilt 1/2-weight-balanced tree which could be done in $O(\log(size))$ time because the tree is also 1/2-height-balanced by Main Theorem (3.1.1). Therefore, the worst-case complexity of $Rebuild\_Tree(size, scapegoat)$ is $O(size)$ time. □

4.2 Search

Theorem 4.2.1. *Worst-case complexity of any Search operation done in Scapegoat tree T is $O(\log(T.size))$.*

Proof. By Theorem (3.3.3), the Scapegoat tree T is loosely α-height-balanced i.e. $T.height \leq T.h_\alpha + 1$ or $T.height = O(\log(T.size))$ therefore, the worst-case complexity of Search operation is $O(\log(T.size))$ time. □

4.3 Insert

Theorem 4.3.1. *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.*

Proof. The proof is done using accounting method in the amortized analysis chapter in [2] starting with accounting function defined for each node in a Scapegoat tree:

$$\Phi_n = \begin{cases} 0 & \text{if} \quad |(n.left).size - (n.right).size| \leq 1 \\ |(n.left).size - (n.right).size| & \text{otherwise} \end{cases}$$

By that way, we have $\Phi_n = 0$ when n is 1/2-weight-balanced node. When n is not α-weight-balanced node or a Scapegoat node and suppose that $(n.left).size > \alpha \cdot n.size$, we have:

$$\begin{aligned} & \Phi_n &=& \ |(n.left).size - (n.right).size| \\ \Rightarrow & \Phi_n &=& \ |(n.left).size - (n.size - (n.left).size - 1)| \\ \Rightarrow & \Phi_n &=& \ |2 \cdot (n.left).size - n.size + 1| \\ \Rightarrow & \Phi_n &=& \ 2 \cdot (n.left).size - n.size + 1 \\ \Rightarrow & \Phi_n &\geq& \ (2 \cdot \alpha - 1) \cdot n.size + 1 \end{aligned}$$

and

$$\begin{aligned} & \Phi_n &=& \ |(n.left).size - (n.right).size| \\ \Rightarrow & \Phi_n &=& \ |(n.size - (n.right).size - 1) - (n.right).size| \\ \Rightarrow & \Phi_n &=& \ |(n.size - 2 \cdot (n.right).size + 1| \\ \Rightarrow & \Phi_n &\leq& \ n.size \end{aligned}$$

Therefore, $\Phi_n = O(\log(n.size))$. Now, in the first case that Insert operation i^{th} of node n did not trigger rebuilding:

$$\begin{aligned} A_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= c \cdot \log(T.size) + \sum_{m \text{ is an ancestor of } n} \Phi_m + \sum_{p \text{ is not an ancestor of } n} \Phi_p \\ &\quad + \Phi_n - \sum_{m \text{ is an ancestor of } n} \Phi'_m - \sum_{p \text{ is not an ancestor of } n} \Phi'_p \end{aligned}$$

By Theorem 3.3.3, the Scapegoat tree after operation i^{th} is loosely α-weight-balanced. Therefore, $C_i = O(\log(T.size))$. The difference between Φ_m and Φ'_m is at most 1 after the insertion and there are at most $O(\log(T.size))$ ancestors like that. Therefore:

$$\sum_{m \text{ is an ancestor of } n} \Phi_m - \sum_{m \text{ is an ancestor of } n} \Phi'_m = O(\log(T.size))$$

Moreover, we have $\displaystyle\sum_{p \text{ is not an ancestor of } n} \Phi_p = \sum_{p \text{ is not an ancestor of } n} \Phi'_p$. That concludes:

$$\begin{aligned} A_i &= c \cdot \log(T.size) + d \cdot \log(T.size) \\ &= O(\log(T.size)) \end{aligned}$$

Now, in other case that Insert operation i^{th} of node n trigger rebuilding a subtree rooted at Scapegoat node s:

$$\begin{aligned} A_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= c \cdot (\log(T.size) + s.size) + \sum_{m \in \text{ the rebuilt subtree rooted at } s} \Phi_m + \sum_{p \notin \text{ the rebuilt subtree rooted at } s} \Phi_p \\ &\quad + \Phi_n - \sum_{m \in \text{ the rebuilt subtree rooted at } s} \Phi'_m - \sum_{p \notin \text{ the rebuilt subtree rooted at } s} \Phi'_p \end{aligned}$$

The real cost here is involved searching for a place to insert in $O(\log(T.size))$ time and finding and rebuilding the subtree rooted at Scapegoat node s in $O(s.size)$ time. The difference between Φ_m and Φ'_m is $O(m.size)$ nodes because $\Phi_m = 0$ and $\Phi'_m = O(m.size)$ due to m was not α-weight-balanced node. For any node p that is not in rebuilt subtree and p is not an ancestor of newly inserted node n, we have $\Phi_p - \Phi'_p = 0$. If p is not in the rebuilt subtree and p is an ancestor of newly inserted node n (there are at most $O(\log(T.size))$ nodes like that because T is always loosely α-weight-balanced), therefore:

$$\sum_{p \notin \text{ the rebuilt subtree rooted at } s} \Phi_p - \sum_{p \notin \text{ the rebuilt subtree rooted at } s} \Phi'_p = O(\log(T.size))$$

That concludes the proof:

$$\begin{aligned} A_i &= c \cdot \log(T.size) + c \cdot s.size + d \cdot \log(T.size) - c \cdot s.size \\ &= O(\log(T.size)) \end{aligned}$$

□

4.4 Delete

Theorem 4.4.1. *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.*

Proof. The total cost of after a Delete operation with rebuilding is $\Omega(T.size + \log(T.size))$ or $\Omega(n + \log n)$ due to $T.size \leq n$. Notice that the rebuilding the whole tree T is taken after a Delete operation if $T.size < \alpha \cdot T.maxSize$ (5). And (5) is satisfied if there were at least $maxSize \cdot (1 - \alpha)$ operations taken place since latest rebuilding or starting empty tree. Moreover, $maxSize \leq n$ or there are $\Omega(n)$ operations to pay up for the cost of rebuilding tree. Therefore, the amortized cost of a Delete operation is $\Omega(\log n)$. □

4.5 Generalized Theorem

The theorem presented in this section is the generalized version of these theorems 4.2, 4.3.1 and 4.4.1 of complexity of Search, Insert, and Delete operations. As we know whenever a Scapegoat tree or a subtree of a Scapegoat tree is rebuilt, it is going to be rebuilt into a 1/2-weight-balanced binary search tree or subtree. Now 1/2 will be replaced by any constant $\alpha_{balanced}$ such that $1/2 \leq \alpha_{balanced} < 1$. Similarly, whenever the height of a newly inserted node n satisfies $n.height > \lfloor \log_{1/\alpha_{trigger}}(n.size) \rfloor$ where $\alpha_{balanced} \leq \alpha_{trigger} < 1$, then some parent p of n will be detected as Scapegoat node by Main Theorem 3.1.1 and the subtree rooted at p will be rebuilt into $\alpha_{balanced}$-weight-balanced subtree. And we assume that we already have two algorithms: rebuilding a tree of n nodes into a $\alpha_{balanced}$-weight-balanced binary search tree in $O(n \cdot F(n))$ time and finding the node that is not $\alpha_{trigger}$-weight-balanced in $O(n \cdot F(n))$ time where $F(n) = \Omega(1)$ and $F(c \cdot n) = O(F(n))$ for any constant c. Then the following theorem holds and is restated without proof from the main paper [3]:

Theorem 4.5.1. *A relaxed Scapegoat tree can handle a sequence of n Insert and m Search or Delete operations, beginning with an empty tree, with an amortized cost $O(F(n)\log_{1/\alpha_{trigger}} n)$ per Insert or Delete and $O(\log_{1/\alpha_{trigger}} k)$ worst-case time per Search, where k is the size of the tree the Search is performed on.*

Chapter 5

APPLICATION

5.1 Orthogonal range queries

An orthogonal query of d dimensions given a range of each dimension is a query for the number of d-dimensional vectors such that each component of a dimension falls in the range of that dimension. Figure 5.1 illustrates an othorgonal query in two dimensions. The query asks for the number of points whose x coordinate ranges from 3 to 11 and y coordinate ranges from 2 to 6. And the expected output is 3 as only three points (4,4), (8,3) and (9,5) fall into the specified ranges. In the general case of d dimensions, Leuker in [5] proposed

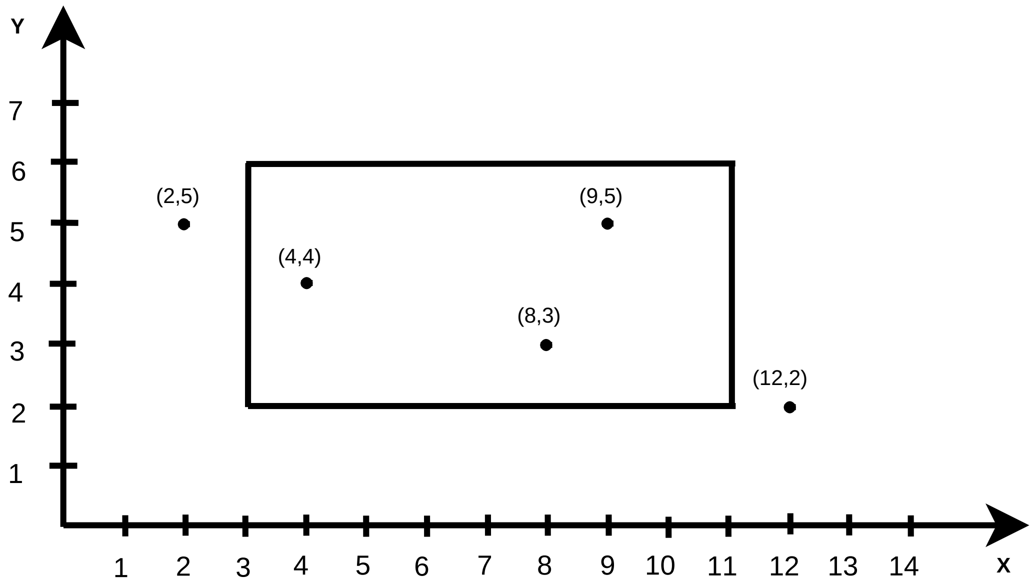


Figure 5.1: An orthogonal query in two dimensions

that orthogonal queries could be done in $O(\log^d n)$ worst-case time where n is the number of nodes in the tree and an update operation is done with an amortized cost of $O(n \log^d n)$. Moreover, also in the paper, it was proved that 1/3-balanced tree of n keys could be built in $O(n \log^{min(1,d-1)} n)$ time. Now we will use that fact to improve the time complexity of operations of orthogonal range querries. It is cleary that if T is a 1/3-balanced tree then T

is 2/3-weight-balanced. So in order to apply Generalized Theorem 4.5.1, we will need to find out what are function F, $\alpha_{balanced}$, $\alpha_{trigger}$. Due to the fact that 1/3-balanced tree of n keys could be built in $O(n \log^{min(1,d-1)} n)$ time, we could determine that $F(n) = \log^{min(1,d-1)} n$, $\alpha_{balanced} = 1/2$ and $\alpha_{trigger} = 2/3$. Then the following theorem follows from Generalized Theorem 4.5.1:

Theorem 5.1.1. *A relaxed Scapegoat tree can handle a sequence of n Insert and m Search or Delete operations, beginning with an empty tree, with an amortized cost $O(F(n) \log_{1/\alpha_{trigger}} n)$ per Insert or Delete and $O(\log_{1/\alpha_{trigger}} k)$ worst-case time per Search, where k is the size of the tree the Search is performed on.*

Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

As shown in this report, a Scapegoat tree T is always loosely α-height-balanced after any update operation. Such loosely α-height balance is due to rebuilding operation taken place after detecting some Scapegoat node, which is a sign of α-height unbalance and α-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.

The Demo for this project is implemented successfully. Even though any update operation in the implementation is done in linear time in term of the size and that's because all the nodes shown in UI needs to be redrawn after the operation. So the coordinates of all the nodes need to be recomputed and all of them need to be redrawn. Aside from the UI, the Search, Insert, Delete operations that are implemented in the Scapegoat tree class truly follow the complexity of theoretical part. Through the implementation of and experiments with the Demo, I have found out some helpful cases, for example, some Scapegoat tree after some number of Insert and Delete operations is loosely α-height-balanced but not α-height-balanced. The case helps me to understand thoroughly the idea of Scapegoat trees to prove theorems and explain it in this report.

6.2 Future Work

There could be many improvements to the current project. First, I could investigate how Scapegoat tree techniques could apply for quad trees in the original paper [3] or even in other applications not mentioned in the paper as well. Another improvement could come from the performance of the Demo. Another method of drawing nodes could be explored so that when rebuilding the subtree rooted at Scapegoat is taken place, the whole tree does not need to be redrawn or just the rebuilt subtree needs to be redrawn. Finally, an alternative way to find the Scapegoat node might be investigated. Finding Scapegoat node in the implementation of this project is done in the time of linear in the size of the subtree rooted at Scapegoat. The alternative way might reduce that kind of complexity into the logarithmic time in the size of the whole tree because the search for a place to insert might give us some information to detect the Scapegoat node.

REFERENCES

[1] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[2] Thomas H. Cormen. *Introduction to Algorithms.* The MIT Press, second edition edition, 2001.

[3] 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.

[4] Leo J. Guibas and Robert Sedgewick. A diochromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Sciences*, pages 28–34. IEEE Computer Soceity, 1978.

[5] George S. Leuker. A data structure for orthogonal range queries. In *In Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 28– 34. IEEE Computer Society, 1978.

[6] I. Nievergelt and E. M. Reinfold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2:33–43, 1973.

Appendix A

HOW TO USE THE DEMO

A.1 How to run

The Demo is already compiled and put in the folder Demo in the enclosed CD. It was written and compiled in Java version 1.5.0_06. To run the Demo, do the following steps if you don't have any IDE to open those java files:

- Enter Command Prompt mode in Windows or Terminal mode in Linux
- Change the current folder to Demo on the CD
- Simply enter command **java ScapegoatTreeDemo**

A.2 How to do Search, Insert, and Delete

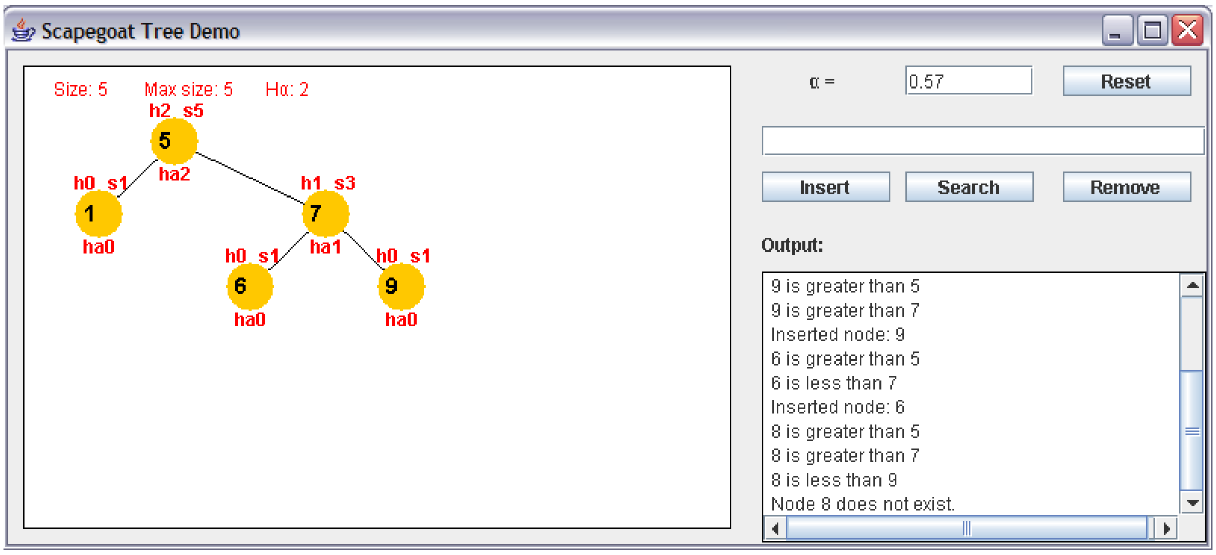
Important notice: For the purpose of demo, all input keys should be positive integers.

Now when you would like to search, insert, or delete a node, just enter the key of a node in the text box below the text box of α then press the corresponding buttons. If you search for or delete a node , for example, node 9 that is not in the tree, you will get the error message "Node 9 does not exist" in the output text box like in Figure A.1.

While inserting a node already in the tree for example node 5, you will get the error message "Node 5 already exists" like in Figure A.2.
If inserting a node results in rebuilding a subtree rooted at Scapegoat node , for example, node 8, you will get the prompting messages: "Scapegoat Node 8 is detected" and "The tree rooted at Scapegoat is going to be rebuilt." like in Figure A.3 and A.4.

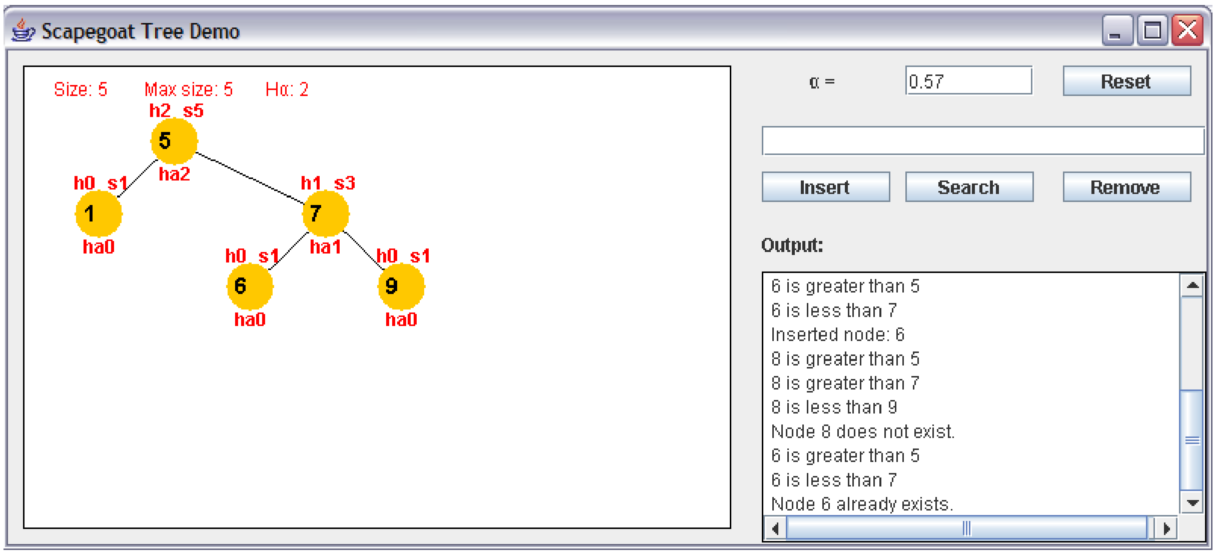
A.3 How to reset tree with a new α

Sometimes, if you would like to experience a new value of α, just simply enter a new value in the input text box of *alpha* and press Reset button. The current tree is deleted and you



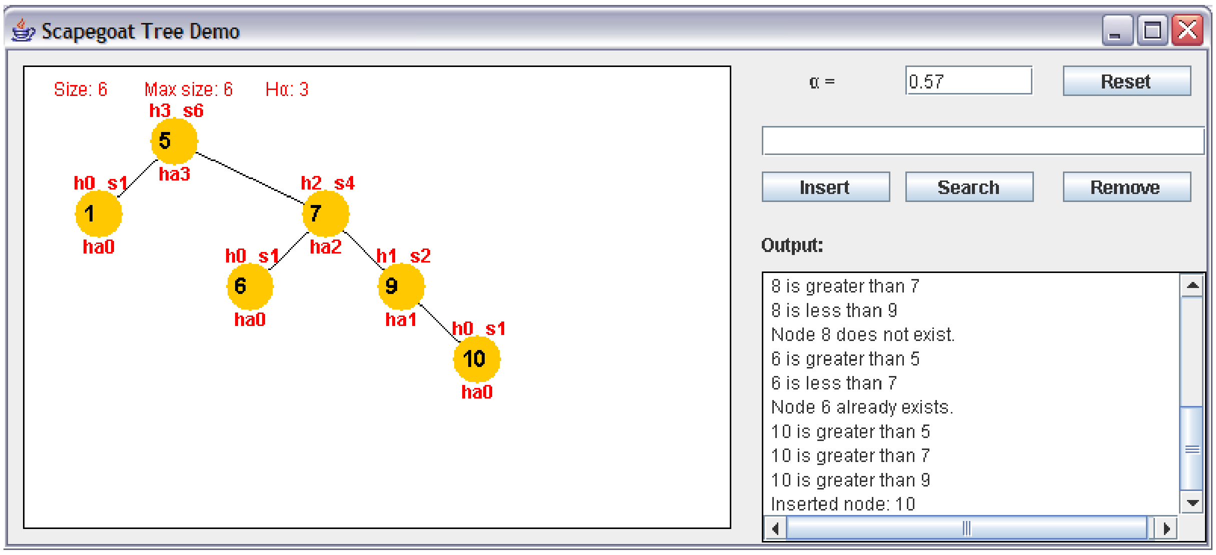
```
9 is greater than 5
9 is greater than 7
Inserted node: 9
6 is greater than 5
6 is less than 7
Inserted node: 6
8 is greater than 5
8 is greater than 7
8 is less than 9
Node 8 does not exist.
```

Figure A.1: Removing a node that does not exist in a Scapegoat tree



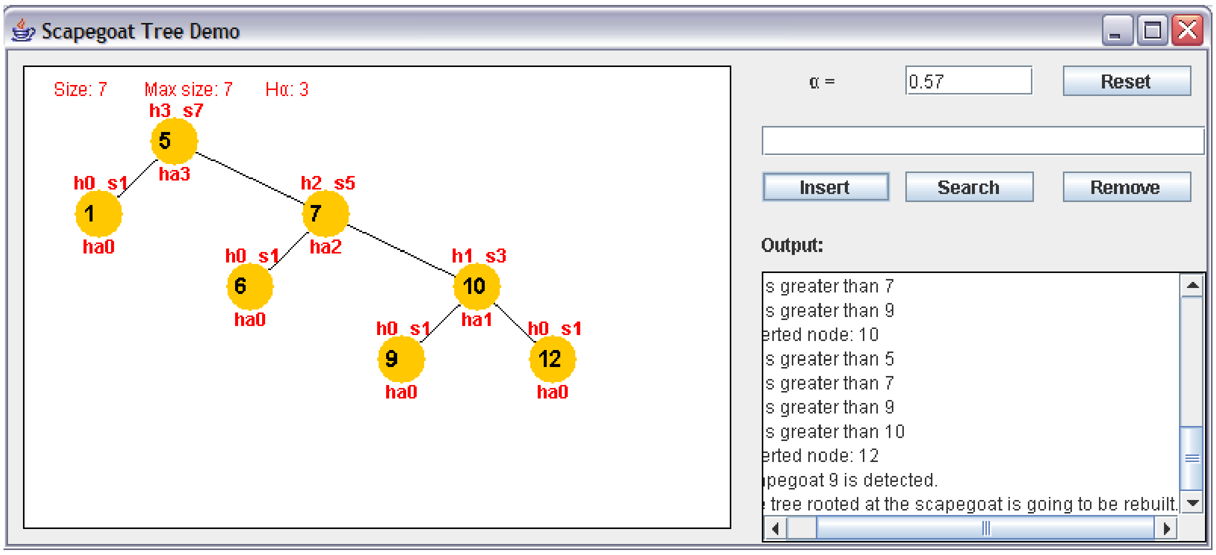
```
6 is greater than 5
6 is less than 7
Inserted node: 6
8 is greater than 5
8 is greater than 7
8 is less than 9
Node 8 does not exist.
6 is greater than 5
6 is less than 7
Node 6 already exists.
```

Figure A.2: Inserting a node that already exists in a Scapegoat tree



```
8 is greater than 7
8 is less than 9
Node 8 does not exist.
6 is greater than 5
6 is less than 7
Node 6 already exists.
10 is greater than 5
10 is greater than 7
10 is greater than 9
Inserted node: 10
```

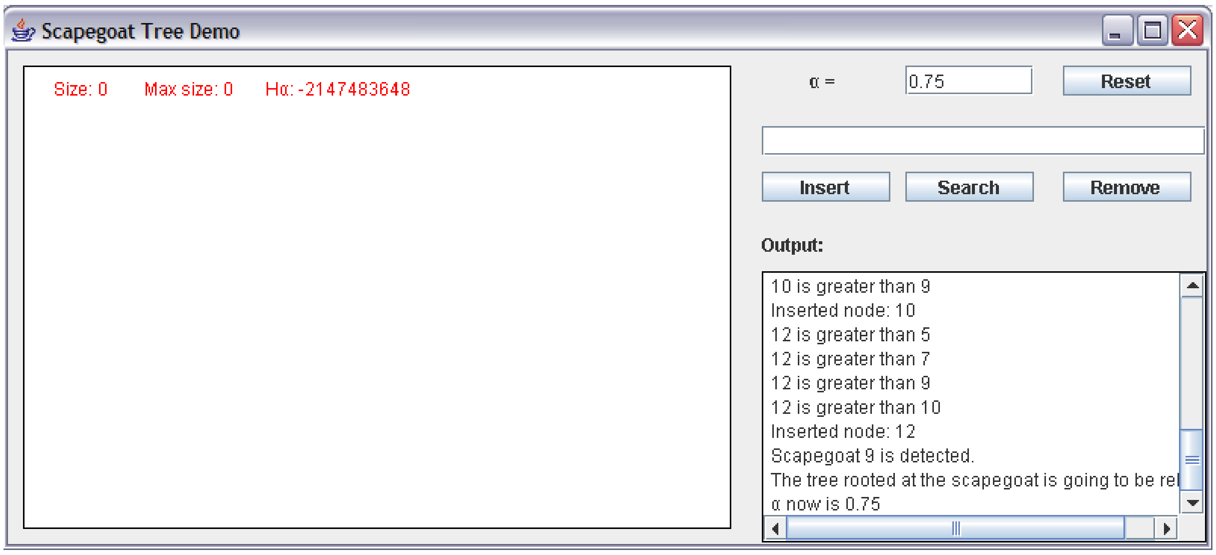
Figure A.3: A Scapegoat tree before inserting a node that leads to rebuilding



```
s greater than 7
s greater than 9
erted node: 10
s greater than 5
s greater than 7
s greater than 9
s greater than 10
erted node: 12
pegoat 9 is detected.
 tree rooted at the scapegoat is going to be rebuilt.
```

Figure A.4: A Scapegoat tree after inserting a node with rebuidling subtree rooted at Scapegoat

will need to build the new tree from scratch like in Figure A.5



```
10 is greater than 9
Inserted node: 10
12 is greater than 5
12 is greater than 7
12 is greater than 9
12 is greater than 10
Inserted node: 12
Scapegoat 9 is detected.
The tree rooted at the scapegoat is going to be rel
α now is 0.75
```

Figure A.5: After resetting the new value of α

Appendix B

LIST OF FIGURES AND TABLES

List of Figures

List of Tables