

Size Balanced Tree

Chen Qifeng (Farmer John)

Zhongshan Memorial Middle School, Guangdong, China

Email:344368722@QQ.com

December 29, 2007

Abstract

This paper presents a unique strategy for maintaining balance in dynamically changing Binary Search Trees that has optimal expected behavior at worst. Size Balanced Tree is, as the name suggests, a Binary Search Tree (abbr. BST) kept balanced by size. It is simple, efficient and versatile in every aspect. It is very easy to implement and has a straightforward description and a surprisingly simple proof of correctness and runtime. Its runtime matches that of the fastest BST known so far. Furthermore, it works much faster than many other famous BSTs due to the tendency of a perfect BST in practice. It supports not only typical primary operations but also Select and Rank.

Key Words And Phrases

Size Balanced Tree

SBT

Maintain

This paper is dedicated to the memory of *Heavens*.

1 Introduction

Before presenting Size Balanced Trees it is necessary to explicate Binary Search Trees and rotations on BSTs, Left-Rotate and Right-Rotate.

1.1 Binary Search Tree

Binary Search Tree is a significant kind of advanced data structures. It supports many dynamic-set operations, including Search, Minimum, Maximum, Predecessor, Successor, Insert and Delete. It can be used both as a dictionary and as a priority queue.

A BST is an organized binary tree. Every node in a BST contains two children at most. The keys for compare in a BST are always stored in such a way as to satisfy the binary-search-tree property:

Let x be a node in a binary search tree. Then the key of x is not less than that in left subtree and not larger than that in right subtree.

For every node t we use the fields of $left[t]$ and $right[t]$ to store two pointers to its children. And we define $key[t]$ to mean the value of the node t for compare. In addition we add $s[t]$, the size of subtree rooted at t , to keep the number of the nodes in that tree. Particularly we call 0 the pointer to an empty tree and $s[0]=0$.

1.2 Rotations

In order to keep a BST balanced (not degenerated to be a chain) we usually change the pointer structure through rotations to change the configuration, which is a local operation in a search tree that preserves the binary-search-tree property.

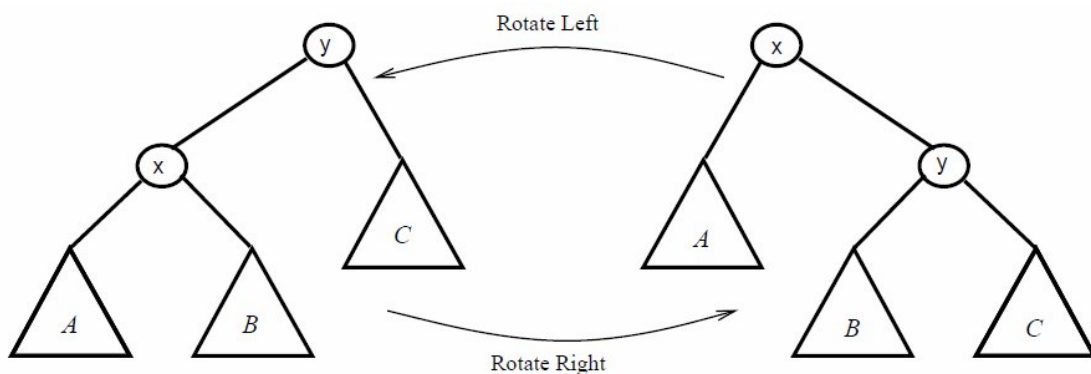


Figure1.1: The operation Left-Rotate(x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The configuration on the left can be transformed into the configuration on the right by the inverse operation, Right-Rotate(y).

1.2.1 Pseudocode Of Right-Rotate

The Right-Rotate assumes that the left child exists.

Right-Rotate (t)

```
1 k ← left[t]
2 left[t] ← right[k]
3 right[k] ← t
4 s[k] ← s[t]
5 s[t] ← s[left[t]] + s[right[t]] + 1
6 t ← k
```

1.2.2 Pseudocode Of Left-Rotate

The Left-Rotate assumes that the right child exists.

Left-Rotate (t)

```
1 k ← right[t]
2 right[t] ← left[k]
3 left[k] ← t
4 s[k] ← s[t]
5 s[t] ← s[left[t]] + s[right[t]] + 1
6 t ← k
```

2 Size Balanced Tree

Size Balanced Tree (abbr. SBT) is a kind of Binary Search Trees kept balanced by size. It supports many dynamic primary operations in the runtime of $O(\log n)$:

Insert(t,v)	Inserts a node whose key is v into the SBT rooted at t.
Delete(t,v)	Deletes a node whose key is v from the SBT rooted at t. In the case that no such a node in the tree, deletes the node searched at last.
Find(t,v)	Finds the node whose key is v and returns it.
Rank(t,v)	Returns the rank of v in the tree rooted at t. In another word, it is one plus the number of the keys which are less than v in that tree.
Select(t,k)	Returns the node which is ranked at the kth position. Apparently it includes the operations of Get-max and Get-min because Get-min is equivalent to Select(t,1) and Get-max is equivalent to Select(t,s[t])
Pred(t,v)	Returns the node with maximum key which is less than v.
Succ(t,v)	Returns the node with minimum key which is larger than v.

Commonly every node of a SBT contains *key*, *left*, *right* and extra but useful updated field: its *size*, which has been defined in the former introduction. The kernel of a SBT is divided into two restrictions on size:

For every node pointed by *t* in a SBT, we guarantee that

Property(a):

$$s[\text{right}[t]] \geq s[\text{left}[\text{left}[t]]], s[\text{right}[\text{left}[t]]]$$

Property(b):

$$s[\text{left}[t]] \geq s[\text{right}[\text{right}[t]]], s[\text{left}[\text{right}[t]]]$$

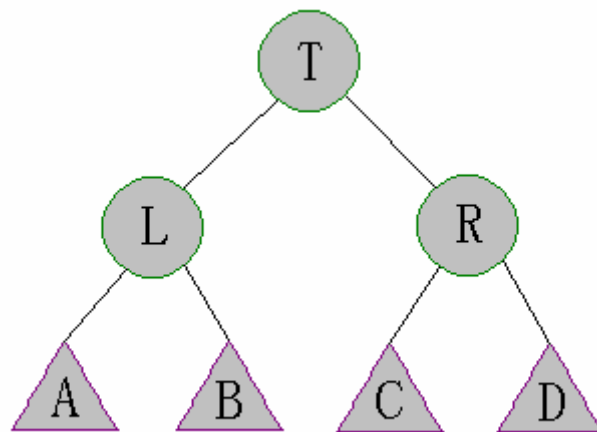


Figure 2.1: The nodes L and R are left and right children of the node T. The Subtrees A and B, C and D are left and right subtrees of the nodes L and R respectively.

Correspond to properties (a) and (b), $s[A], s[B] \leq s[R] \ \& \ s[C], s[D] \leq s[L]$

3 Maintain

Assume that we need to insert a node whose key is *v* into a BST. Generally we use the following procedure to accomplish the mission.

Simple-Insert (*t*,*v*)

```

1  If t=0 then
2    t ← NEW-NODE(v)
3  Else
4    s[t] ← s[t]+1
5    If v<key[t] then
6      Simple-Insert(left[t],v)
7    Else
8      Simple-Insert(right[t],v)

```

After the execution of Simple-Insert properties (a) and (b) are probably not satisfied. In that case we need to maintain the SBT.

The vital operation on a SBT is a unique procedure, Maintain. Maintain(t) is used to maintain the SBT rooted at t . The hypothesis that the subtrees of t are SBT is applied before the performance.

Since properties (a) and (b) are symmetrical we only discuss the case that restriction (a) is violated in detail.

Case 1: $s[\text{left}[\text{left}[t]] > s[\text{right}[t]]]$

In this case that $s[A] > s[R]$ correspond to Figure 2.1 after Insert(left[t], v), we can execute the following instructions to repair the SBT:

- (1) First perform Right-Rotate(T). This operation transforms Figure 2.1 into Figure 3.1;

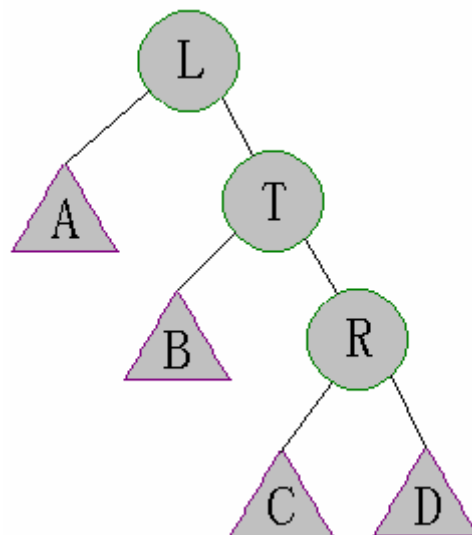


Figure 3.1: All the nodes are defined the same as in Figure 2.1.

- (2) And then sometimes it occurs that the tree is still not a SBT because $s[C] > s[B]$ or $s[D] > s[B]$ by any possibility. So it is necessary to perform Maintain(T).
- (3) In succession the configuration of the right subtree of the node L might be changed. Because of the possibility of violating the properties it is needful to run Maintain(L) again.

Case 2: $s[\text{right}[\text{left}[t]] > s[\text{right}[t]]]$

In this case that $s[B] > s[R]$ correspond to Figure 3.2 after Insert(left[t], v), the maintaining is kind of complicated than that in case 1. We can carry out the following operations for repair.

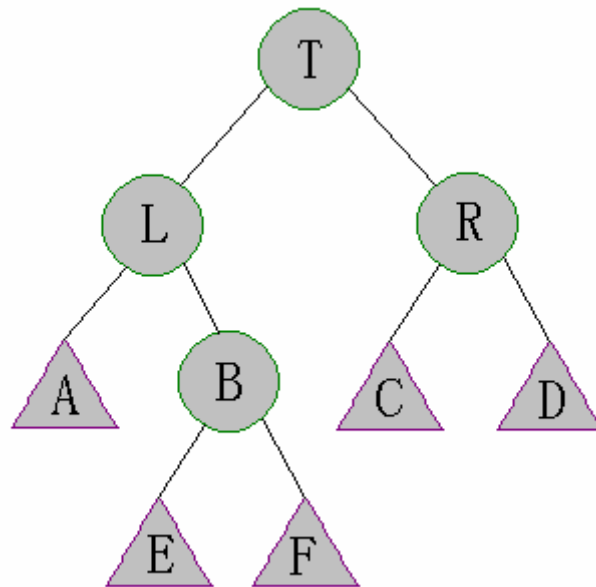


Figure 3.2: All the nodes are defined the same as in Figure 2.1 except E, F and B. E and F are the subtrees of the node B.

- (1) First perform Left-Rotate(L). After that Figure 3.2 was transformed to Figure 3.3 shown below;

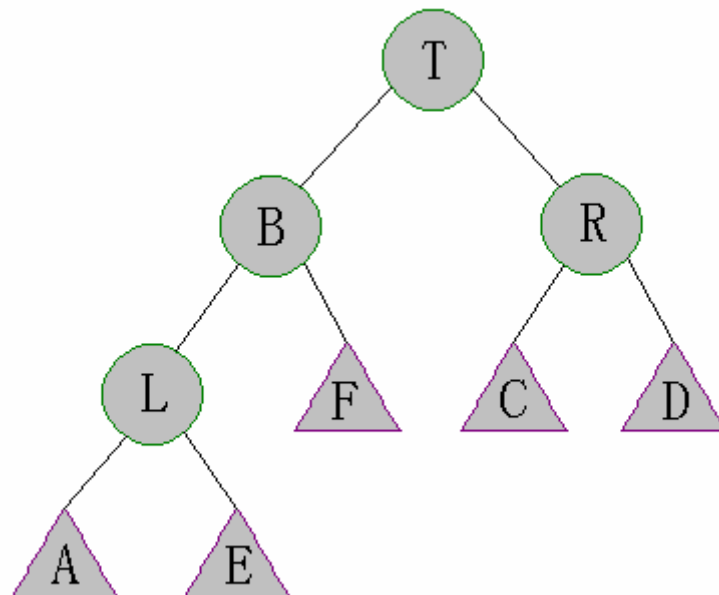


Figure 3.3: All the nodes are defined the same as in Figure 3.2.

- (2) And then perform Right-Rotate(T). This performance results in the transformation from Figure 3.3 to following Figure 3.4.

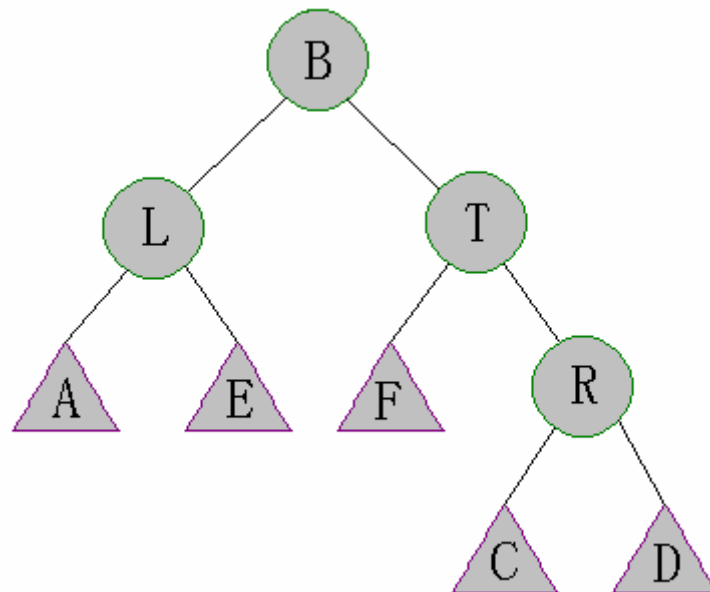


Figure 3.4: All the nodes are defined the same as in Figure 3.2.

- (3) After (1) and (2), the tree becomes to be very unpredictable. But luckily, in Figure 3.4 subtrees A, E, F and R are still SBTs. So we can perform Maintain(L) and Maintain(T) to repair the subtrees of the node B.
- (4) After step (3), those subtrees are already SBTs. But properties (a) and (b) may be still violated on the node B. Thus we need perform Maintain(B) again.

Case 3: $s[\text{right}[\text{right}[t]]] > s[\text{left}[t]]$

This case is symmetrical with case 1.

Case 4: $s[\text{left}[\text{right}[t]]] > s[\text{left}[t]]$

This case is symmetrical with case 2.

3.1 Pseudocode Of Standard Maintain

According to the previous analysis it is easy to implement a normal Maintain.

Maintain (t)

- 1 **If** $s[\text{left}[\text{left}[t]]] > s[\text{right}[t]]$ **then**
- 2 Right-Rotate(t)
- 3 Maintain(right[t])
- 4 Maintain(t)
- 5 Exit
- 6 **If** $s[\text{right}[\text{left}[t]]] > s[\text{right}[t]]$ **then**
- 7 Left-Rotate(left[t])

```

8   Right-Rotate(t)
9   Maintain(left[t])
10  Maintain(right[t])
11  Maintain(t)
12  Exit
13  If s[right[right[t]]>s[left[t]] then
14      Left-Rotate(t)
15      Maintain(left[t])
16      Maintain(t)
17      Exit
18  If s[left[right[t]]>s[left[t]] then
19      Right-Rotate(right[t])
20      Left-Rotate(t)
21      Maintain(left[t])
22      Maintain(right[t])
23      Maintain(t)

```

3.2 Pseudocode Of Faster And Simpler Maintain

The standard pseudocode is a little complicated and slow. Generally we can ensure that property (a) or (b) has been satisfied. Therefore we only need to check cases 1 and 2 or case 3 and 4 to speed up. In that case we can add a boolean variant, *flag*, to avoid meaningless checking. If *flag* is false cases 1 and 2 will be checked, otherwise cases 3 and 4 will be checked.

```

Maintain (t,flag)
1  If flag=false then
2      If s[left[left[t]]>s[right[t]] then
3          Right-Rotate(t)
4      Elseif s[right[left[t]]>s[right[t]] then
5          Left-Rotate(left[t])
6          Right-Rotate(t)
7      Else exit
8  Elseif s[right[right[t]]>s[left[t]] then
9      Left-Rotate(t)
10     Elseif s[left[right[t]]>s[left[t]] then
11         Right-Rotate(right[t])
12         Left-Rotate(t)
13     Else exit
14  Maintain(left[t],false)
15  Maintain(right[t],true)
16  Maintain(t,false)
17  Maintain(t,true)

```


Why $\text{Maintain}(\text{left}[t], \text{true})$ and $\text{Maintain}(\text{right}[t], \text{false})$ are expelled? What is the runtime of Maintain ? You can find the answers in part 6, analysis.

4 Insertion

The insertion on a SBT is very simple. Here's the pseudocode of Insert on a SBT.

4.1 Pseudocode Of Insert

```
Insert (t,v)
1  If t=0 then
2    t←NEW-NODE(v)
3  Else
4    s[t] ←s[t]+1
5    If v<key[t] then
6      Insert(left[t],v)
7    Else
8      Insert(right[t],v)
9    Maintain(t,v ≥ key[t])
```

5 Deletion

I augment the deletion for convenience. If no such a value to delete in a SBT we delete the node searched at last and record it. Here's the standard pseudocode of Delete on a SBT.

5.1 Pseudocode Of Standard Delete

```
Delete (t,v)
1  s[t] ←s[t]-1
2  If (v=key[t])or(v<key[t])and(left[t]=0)or(v>key[t])and(right[t]=0) then
3    Delete ←key[t]
4    If (left[t]=0)or(right[t]=0) then
5      t ←left[t]+right[t]
6    Else
7      key[t] ←Delete(left[t],v[t]+1)
8  Else
9    If v<key[t] then
10   Delete(left[t],v)
```

```

11 Else
12   Delete(right[t],v)
13 Maintain(t,false)
14 Maintain(t,true)

```

5.2 Pseudocode Of Faster And Simpler Delete

Actually this is the simplest deletion without other functions. Delete(t,v) here is a function that returns the value deleted. It can result in a destroyed SBT. But with the insertion above, a BST is still kept at the height of $O(\log n^*)$ where n^* is the total number of insertions, not the current size!

Delete (t,v)

```

1  s[t] ← s[t]-1
2  If (v=key[t])or(v<key[t])and(left[t]=0)or(v>key[t])and(right[t]=0) then
3    Delete ← key[t]
4    If (left[t]=0)or(right[t]=0) then
5      t ← left[t]+right[t]
6    Else
7      key[t] ← Delete(left[t],v[t]+1)
8  Else
9    If v<key[t] then
10     Delete(left[t],v)
11   Else
12     Delete(right[t],v)

```

6 Analysis

Obviously Maintain is a recursive procedure. Maybe the question about whether it can stop or not has appeared in your mind. Not to worry: it has been proved that the amortized runtime for Maintain is only $O(1)$ at worst.

6.1 Analysis Of Height

Let $f[h]$ be the minimum number of the nodes of a SBT whose height is h . We have

$$f[h] = \begin{cases} 1 & (h = 0) \\ 2 & (h = 1) \\ f[h-1] + f[h-2] + 1 & (h > 1) \end{cases}$$

6.1.1 Proof

- (1) It is easy to work out that $f[0] = 1$ and $f[1] = 2$.
- (2) First of all, $f[h] \geq f[h-1] + f[h-2] + 1 (h > 1)$. For each $h > 1$, let's assume that t points to a SBT whose height is h . And then this SBT contains a subtree at the height of $h-1$. It doesn't matter to suppose it as the left subtree. According to the previous definition of $f[h]$, we have that $s[left[t]] \geq f[h-1]$. And there is an $h-2$ -tall subtree of the left subtree. In another word there is a subtree whose size is at least $f[h-2]$ of the left subtree. Owing to the property (b) we have that $s[right[t]] \geq f[h-2]$. Therefore we draw the conclusion that $s[t] \geq f[h-1] + f[h-2] + 1$.

On the other hand, $f[h] \leq f[h-1] + f[h-2] + 1 (h > 1)$. We can construct a SBT with exact $f[h]$ node(s) and its height is h . We call this kind of SBT $tree[h]$.

$$tree[h] = \begin{cases} \text{a BST with one node} & (h = 0) \\ \text{any BST with two nodes} & (h = 1) \\ \text{a BST containing } tree[h-1] \text{ and } tree[h-2] \text{ as two subtrees} & (h > 1) \end{cases}$$

Hence that $f[h] = f[h-1] + f[h-2] + 1 (h > 1)$ is obtained by summing up two upper aspects.

6.1.2 The Worst Height

In fact $f[h]$ is exponential function. Its precise value can be calculated from the recursion.

$$f[h] = \frac{\alpha^{h+3} - \beta^{h+3}}{\sqrt{5}} - 1 = \left\| \frac{\alpha^{h+3}}{\sqrt{5}} \right\| - 1 = Fibonacci[h+2] - 1 = \sum_{i=0}^h Fibonacci[i]$$

$$\text{where } \alpha = \frac{1+\sqrt{5}}{2} \text{ and } \beta = \frac{1-\sqrt{5}}{2}$$

Some usual values of $f[h]$

H	13	15	17	19	21	23	25	27	29	31
F[h]	986	2583	6764	17710	46367	121392	317810	832039	2178308	5702886

Lemma

The worst height of an n -node SBT is the maximum h subjected to $f[h] \leq n$.

Assume that $\max h$ is the worst height of a SBT at the size of n . According to the lemma above, we have

$$\begin{aligned}
 f[\max h] &= \left\lfloor \frac{\alpha^{\max h+3}}{\sqrt{5}} \right\rfloor - 1 \leq n \Rightarrow \\
 \frac{\alpha^{\max h+3}}{\sqrt{5}} &\leq n + 1.5 \Rightarrow \\
 \max h &\leq \log_{\alpha}^{\sqrt{5}(n+1.5)} - 3 \Rightarrow \\
 \max h &\leq 1.44 \log_2^{n+1.5} - 1.33
 \end{aligned}$$

Now it is clear that the height of a SBT is $O(\log n)$.

6.2 Analysis Of Maintain

We can easily prove that Maintain works quite efficiently using the previous conclusions.

There is a very important value to estimate how well a BST is, the average of all nodes' depths. It is the quotient obtained by SD, the sum of the depths of all nodes, dividing n . Generally the less it is, the better a BST is. Because of the constant n , SD is expected to be as small as possible.

Now we need to concentrate on SD. Its significance is the ability to restrict the times of Maintain. Recalling the conditions to perform rotations in Maintain it is surprising that SD always decreases after rotations.

In case 1, for example, comparing Figure 2.1 with Figure 3.1, SD increases a negative value, $s[\text{right}[t]] - s[\text{left}[\text{left}[t]]]$.

And for instance comparing Figure 3.2 to Figure 3.4, SD increases a value less than -1 , $s[\text{right}[t]] - s[\text{right}[\text{left}[t]]] - 1$.

Due to the height of $O(\log n)$ SD is always kept $O(n \log n)$. And SD just increases $O(\log n)$ after an insertion on a SBT. Therefore

$$\begin{aligned}
 SD &= n \times O(\log n) - T = O(\log n) \Rightarrow \\
 T &= O(n \log n)
 \end{aligned}$$

where T is the number of Maintains running rotations. The total number of Maintains is T plus the number of Maintains without rotations. Since the latter is $O(n \log n) + O(T)$, the amortized runtime for Maintain is

$$\frac{O(T) + O(n \log n) + O(T)}{n \log n} = O(1)$$

6.3 Analysis Of Each Operation

Now that the height of SBT is $O(\log n)$ and Maintain is $O(1)$, the runtime for all primary operations are $O(\log n)$.

6.4 Analysis Of Faster And Simpler Delete

We call the statement $P(n^*)$ that a BST with the faster and simpler Delete and n^* standard Inserts is at the height of $O(\log n^*)$. We prove $P(n^*)$ is true for arbitrary positive integer n^* by mathematical induction.

6.4.1 Proof

Here I just give a rough proof.

Assume that a node t is checked by $\text{Maintain}(t, \text{false})$, we have

$$\begin{aligned} \frac{s[\text{left}[t]] - 1}{2} &\leq s[\text{right}[t]] \Rightarrow \\ s[\text{left}[t]] &\leq \frac{2s[t] - 1}{3} \end{aligned}$$

Therefore if all nodes on the path from a node to the root are checked by Maintain the depth of this node is $O(\log n)$.

- (1) For $n^*=1$, $P(n^*)$ is true apparently;
- (2) Assume that $P(n^*)$ is true for $n^* < k$. For $n^*=k$, after the last consecutive insertions, the nodes checked by Maintain must be connected together to form a tree. For every leaf of this tree, the subtree pointed by it does not be changed by Maintain. So the depths of the nodes in this subtree is not larger than $O(\log n^*) + O(\log n) = O(\log n^*)$
- (3) Therefore $P(n^*)$ is true for $n^*=1, 2, 3, \dots$

In this way the amortized runtime of Maintain is still $O(1)$.

Analysis Of Faster And Simpler Maintain

Here's the discussion about why $\text{Maintain}(\text{left}[t], \text{true})$ and $\text{Maintain}(\text{right}[t], \text{false})$ can be expelled.

In case 1 in Figure 3.2 we have

$$\begin{aligned}
s[L] &\leq 2s[R] + 1 \Rightarrow \\
s[B] &\leq \frac{2s[L] - 1}{3} \leq \frac{4s[R] + 1}{3} \Rightarrow \\
s[E], s[F] &\leq \frac{2s[B] - 1}{3} \leq \frac{8s[R] + 3}{9} \Rightarrow \\
s[E], s[F] &\leq \left\lfloor \frac{8s[R] + 3}{9} \right\rfloor \leq s[R]
\end{aligned}$$

Thus $\text{Maintain}(\text{right}[t], \text{false})$, correspond to $\text{Maintain}(T, \text{false})$ in Figure 3.1, can be expelled. And $\text{Maintain}(\text{left}[t], \text{true})$ is unnecessary apparently.

In case 2 in Figure 3.2 we have

$$\begin{cases} s[A] \geq s[E] \\ s[F] \leq s[R] \end{cases}$$

These inequations also mean that the size of subtrees of E is less than $s[A]$ and the size of subtrees of F is less than $s[R]$. Thus $\text{Maintain}(\text{right}[t], \text{false})$ and $\text{Maintain}(\text{left}[t], \text{true})$ can be expelled.

7 Advantage

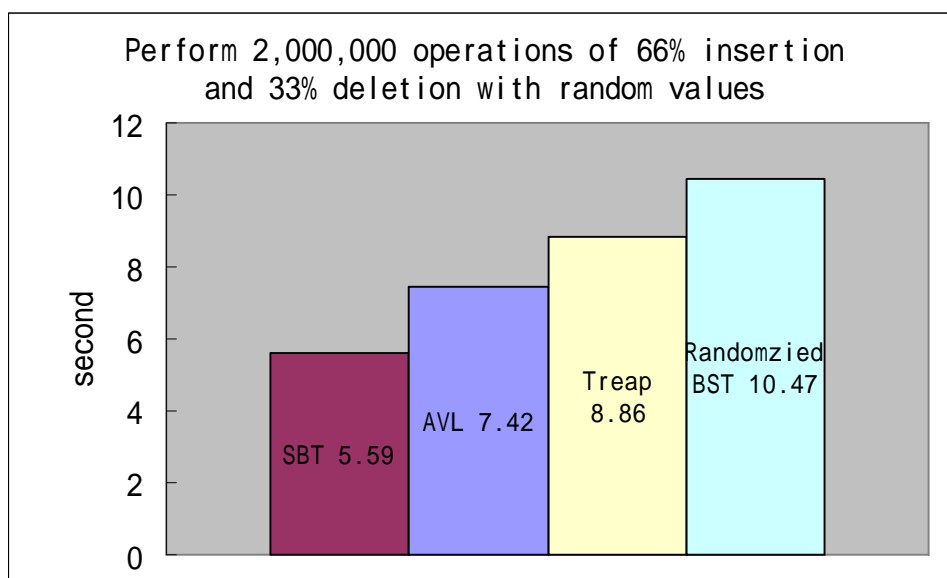
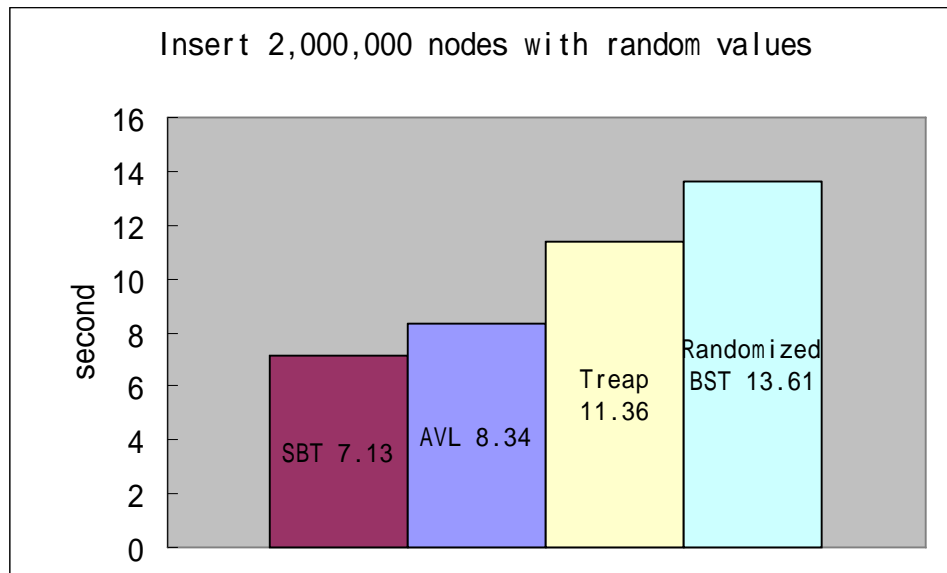
7.1 How Fast A SBT Runs

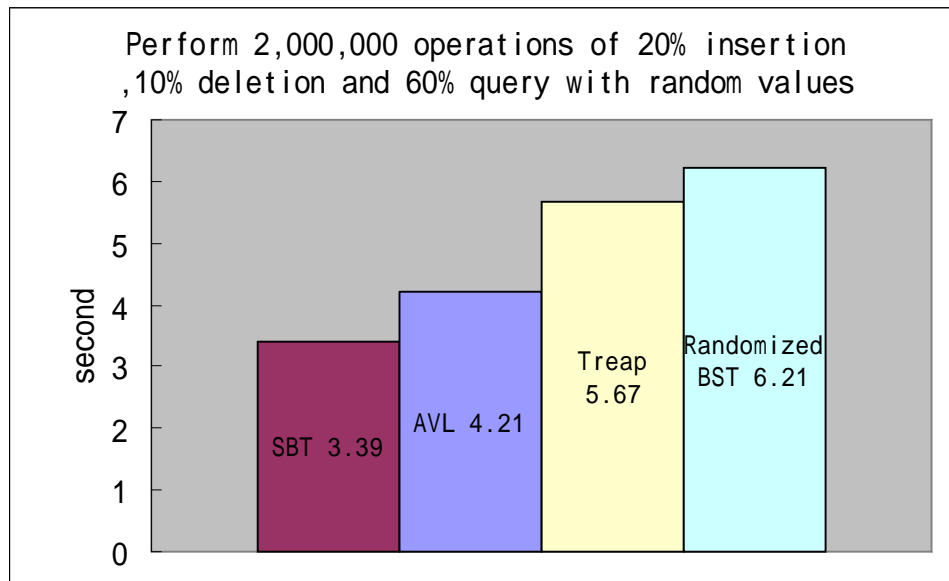
7.1.1 Typical Problem

Write a program to perform n operations given from the input. They are

- 1) Insert a given number into the set;
- 2) Delete a given number from the set;
- 3) Return if a given number is in the set;
- 4) Return the rank of a given number in the set;
- 5) Return the kth number in the set;
- 6) Return the previous number of a given number in the set;
- 7) Return the succeeding number of a given number in the set.

7.1.2 Statistic





In practice SBTs work excellently. From the upper charts we see that SBTs run much faster than other balanced BSTs while the data are random. Moreover the more ordered the data are, the unexpectedly faster SBTs run. It takes only 2 seconds to insert 2,000,000 ordered nodes into a SBT.

7.2 How Efficiently A SBT Works

The average of all nodes' depths nonincreases while Maintain is running. For this reason a SBT always tends to be a perfect balanced BST.

Insert 2,000,000 nodes with random values

Type	SBT	AVL	Treap	Randomized BST	Splay	Perfect BST
Average Depth	19.2415	19.3285	26.5062	25.5303	37.1953	18.9514
Height	24	24	50	53	78	20
Times of Rotation	1568017	1395900	3993887	3997477	25151532	?

Insert 2,000,000 nodes with ordered values

Type	SBT	AVL	Treap	Randomized BST	Splay	Perfect BST
Average Depth	18.9514	18.9514	25.6528	26.2860	999999.5	18.9514
Height	20	20	51	53	1999999	20
Times of Rotation	1999979	1999979	1999985	1999991	0	?

7.3 How Easy Debugging Is

At first we can merely implement a simple BST to guarantee that the main of the code is correct. After that we add Maintain into the Insert and then debug. If an error is checked out we only need to debug Maintain. Moreover a SBT doesn't base on random so debugging is much stabler comparing to Treap, Skip list, Randomized BST and so on.

7.4 How Simple A SBT Is

A SBT is almost the same as a simple BST. Removing the only additional Maintain in Insert the former turns to be the latter. And Maintain is quite simple too.

7.5 How Compact A SBT Is

Lots of balanced BSTs such as SBT, AVL, Treap, Red-Black BST and so on need extra fields to be kept balanced. But many of them are useless like *height*, *random factor* and *color*. On the contrary a SBT contains a useful extra field, *size*. In possession of it we can augment BSTs to support selection and ranking.

7.6 How Versatile A SBT Is

Now that the height of a SBT is $O(\log n)$, we can perform Select in $O(\log n)$ in the worst case. But Splay can't support it very well because the height can be degenerated to be $O(n)$ easily, which is exposed by the char above.

Acknowledgments

The author thanks his English teacher, Fiona, for enthusiastic helps.

References

- [1] G.M.Adelson-Velskii and E.M.Landis, "An algorithm for the Organization of Information", Soviet.Mat.Doklady (1962)
- [2] L.J.Guibas and R.Sedgewick, "A dichromatic Framework for Balanced Trees", Proceedings of the Nineteenth Annual IEEE Symposium on Foundations of Computer Science (1978)

- [3] D. D. Sleator and R. E. Tarjan, 'Self-adjusting binary search trees', *JACM*, **32**, 652–686 (1985).
- [4] S.W.Bent and J.R.Driscoll, Randomly balanced searchtrees. Manuscript (1991).
- [5] Raimund Seidel and Cecilia R. Aragon, Randomized Search Trees.(1996).
- [6] M.A.Weiss, "Data Structures and Algorithm Analysis in C++", Third Edition, Addison Wesley, 2006.
- [7] R.E. Tarjan, "Sequential access in splay trees takes linear time", *Combinatorica* 5(4), 1985, pp. 367-378.
- [8] J. Nievergelt and E. M. Reingold, 'Binary search trees of bounded balance', *SIAM J. Computing*, 2, 33–43 (1973).
- [9] K.Mehlhorn, and A.Tsakalidis, "Data Structures," in *Handbook of Theoretical Computer Science*, Chapter 6, Vol.A, pp.303–341, Elsevier Science Publishers,(1990)