Size Balanced Tree

From PEGWiki

A size balanced tree (SBT) is a self-balancing binary search tree (BBST) first published by Chinese student Qifeng Chen in 2007. The tree is rebalanced by examining the sizes of each node's subtrees. Its abbreviation resulted in many nicknames given by Chinese informatics competitors, including "sha bi" tree (Chinese: 傻屄树; pinyin: shǎ bì shù; literally meaning "dumb cunt tree") and "super BT", which is a homophone to the Chinese term for snot (Chinese: 鼻涕; pinyin: bítì) suggesting that it is messy to implement. Contrary to what its nicknames suggest, this data structure can be very useful, and is also known to be easy to implement. Since the only extra piece of information that needs to be stored is sizes of the nodes (instead of other "useless" fields such as randomized weights in treaps or colours in red–black trees), this makes it very convenient to implement the select-by-rank and get-rank operations (easily transforming it into an order statistic tree). It supports standard binary search tree operations such as insertion, deletion, and searching in O(log n) time. According to Chen’s paper, "it works much faster than many other famous BSTs due to the tendency of a perfect BST in practice."[1]

Contents

1 Properties
2 Rotations
   2.1 Left Rotation
   2.2 Right Rotation
3 Maintenance
4 Fundamental Operations
   4.1 Searching
   4.2 Get Max/Min
   4.3 Iteration
   4.4 Insertion
   4.5 Deletion
5 Order Statistics
   5.1 Select
   5.2 Rank
6 References

Properties

The size balanced tree examines each node's size (i.e. the number of nodes below it in the subtree) to determine when rotations should be performed. Each node $T$ in the tree satisfies the following two properties:

\begin{align*}
1. \quad & \text{size}(T.left) \geq \text{size}(T.right.left), \text{size}(T.right.right) \\
2. \quad & \text{size}(T.right) \geq \text{size}(T.left.left), \text{size}(T.left.right)
\end{align*}

In other words, each child node of $T$ is not smaller in size than the child nodes of its sibling. Clearly, we should consider the sizes of nonexistent children and siblings to be 0.

Consider the following example where $T$ is the node in question, $L, R$ are its child nodes, and $A, B, C, D$ are subtrees which also satisfy the above SBT properties on their own.
Then, the node $T$ must satisfy:

- $\text{size}(L) \geq \text{size}(C), \text{size}(D)$
- $\text{size}(R) \geq \text{size}(A), \text{size}(B)$

**Rotations**

The rotations of SBTs are analogous to those in other self-balancing binary search trees.

### Left Rotation

```python
def left_rotate(t):
    k ← t.right
    t.right ← k.left
    k.left ← t
    k.size ← t.size
    t.size ← t.left.size + t.right.size + 1
    t ← k
```

### Right Rotation

```python
def right_rotate(t):
    k ← t.left
    t.left ← k.right
    k.right ← t
    k.size ← t.size
    t.size ← t.left.size + t.right.size + 1
    t ← k
```

**Maintenance**

After insertions and deletions, the new sizes of subtrees may violate the two properties above. Thus, we define a procedure `maintain(T)` to rebalance the SBT rooted at the node $T$. This should be called with the precondition that $T$'s children are already SBTs themselves. Since property 1 and 2 are symmetrical, we will only discuss property 1.

There are 4 cases to consider when rebalancing.

- **Case 1:** $\text{size}(T.left) < \text{size}(T.right.left)$

    Perhaps after inserting a value to $T.right$, the scenario below (figure 1) may occur, leading to $\text{size}(L) < \text{size}(C)$. 
To fix this, we first perform a right-rotate on $T.right$ (figure 2) and then a left-rotate on $T$ (figure 3).

After these operations, the properties of the entire tree in figure 3 becomes unpredictable. Luckily, the subtrees $A, B, D, E, F, L$ are still SBTs. Thus, we can recursively call maintain on subtrees $R$ and $T$ to take care of them.

Now that all of the subtrees are SBTs, we still have to make sure that the root node $C$ satisfies the SBT properties. So, we call maintain one last time on root node $C$.

- **Case 2**: $\text{size}(T.left) < \text{size}(T.right)$

Perhaps after inserting a value to $T.right$, the scenario below (figure 4) may occur, leading to $\text{size}(L) < \text{size}(D)$. This is similar to case 1, except that instead of going below $C, E$ and $F$ instead goes below $D$. We can omit them from the diagram.

Fixing this, we will perform a left-rotate on the root node $T$, obtaining the structure in figure 5.

After this, the tree rooted at R is still not yet a SBT because $\text{size}(C) < \text{size}(A)$ or $\text{size}(C) < \text{size}(B)$ may be true. So, we continue to call maintain on $T$.

Now that we have satisfied the precondition of making $R$'s subtrees SBTs, we may call maintain on $R$ itself.

- **Case 3**: $\text{size}(T.right) < \text{size}(T.left.right)$

Symmetrical to case 1.

- **Case 4**: $\text{size}(T.right) < \text{size}(T.left.left)$

Symmetrical to case 2.

With this casework being taken care of, it becomes straightforward to actually implement maintain.

```python
def maintain(t):
    if t.left.size < t.right.left.size:         //case 1
        right-rotate(t.right)
        left-rotate(t)
        maintain(t.left)
        maintain(t.right)
```
This pseudocode is slightly slow and redundant. Since we know that the two SBT properties will usually be satisfied, the following is an optimization. Simply add an extra boolean flag to the maintain parameters, indicating whether cases 1/2 or their symmetrical cases are being examined. If the flag is TRUE, then we examine cases 1 and 2, otherwise we examine cases 3 and 4. Doing so will eliminate many unnecessary comparisons.

```python
def maintain(t, flag):
    if flag:
        if t.left.size < t.right.left.size:        //case 1
            right-rotate(t.right)
            left-rotate(t)
        else if t.left.size < t.right.right.size:  //case 2
            left-rotate(t)
        else:
            done
    else:
        if t.right.size < t.left.right.size:       //case 1'
            left-rotate(t.left)
            right-rotate(t)
        else if t.right.size < t.left.left.size:   //case 2'
            right-rotate(t)
        else:
            done
    maintain(t.left, FALSE)   //maintain the left subtree
    maintain(t.right, TRUE)   //maintain the right subtree
    maintain(t, TRUE)         //maintain the whole tree
    maintain(t, FALSE)        //maintain the whole tree
```

The proof for why `maintain(t.left, TRUE)` and `maintain(t.right, FALSE)` are unnecessary can be found in section 6 of Chen's paper. Furthermore, the running time of `maintain` is O(1) amortized (which means that you do not have to worry about it not terminating).

**Fundamental Operations**

**Searching**

Searching in SBTs is exactly the same as searching in other binary search trees. The following iterative implementation will return a pointer to the node in the SBT rooted at \( t \) which has key \( k \).

```python
def search(t, k):
    x ← t
    while x is not NIL:
        if k < x.key then x ← x.left
        else if x.key < k then x ← x.right
```

Get Max/Min

The size of the SBT is already stored. These operations can thus be handled trivially by the `select` operation implemented in the section below.

Iteration

Iterating a SBT is exactly the same as iterating a normal binary search tree (by repeatedly finding nodes' predecessors/successors).

Insertion

Inserting into a SBT is very simple. The only difference from normal binary search trees is that it has an extra call to `maintain` at the end. The following recursive version will insert the node `x` into the SBT rooted at `t`.

```python
def insert(t, x):
    if x is NIL:
        t ← x
    else:
        t.size ← t.size + 1
        if x.key < t.key:
            insert(t.left, x)
        else:
            insert(t.right, x)
        maintain(t, x.key ≥ t.key)
```

Deletion

Deletion is exactly the same as in normal binary search trees. It is not even necessary to call `maintain` afterwards. The proof for this is as follows: A SBT will have all of its properties before deletion. Even though we cannot guarantee that the SBT will retain its balanced properties after the insertion, we know for sure that its height (and thus, its running time) will not increase. Given this, it is clear that calling `maintain` after deleting is extraneous.

Order Statistics

Since SBTs already conveniently store the `size` field to maintain balance, nothing else is needed to transform it into a fully-fledged order statistics tree.

Select

The following function returns a pointer to the `i`th smallest element in the SBT rooted at `t`, where `i` is zero-indexed. To make this one-indexed, simply change "`r ← t.left.size`" to "`r ← t.left.size + 1`" and "`i - (r + 1)`" to "`i - r`".

```python
def select(t, i):
    r ← t.left.size
    if i = r:
        return t
    else if i < r:
        return select(t.left, i)
    else:
        return select(t.right, i - (r + 1))
```
Rank

Determining the rank of an element in a SBT is exactly the same as doing so for a regular binary search tree.

References


- This page was last modified on 8 September 2014, at 20:33.
- Content is available under Attribution 3.0 Unported unless otherwise noted.