

Red Black Tree

Afshin Sepehri
afshin@eng.umd.edu

Fall 2002

1. Introduction

In this project, a red black tree is implemented and standard queries *Search*, *Insert* and *Delete* are presented. For each query, the complexity including the number of executed instruction in general case as well as in worst case is calculated. The program is written defensive enough so that *Insert* query can cope with cases that key already exists and *Delete* query considers cases where key is not found in the tree. A graphical view of the tree is generated so that the result tree can be seen.

2. Algorithm

All the algorithms used in this program are classic ones presented in the book [1]. The implemented operations are:

- Finding minimum node of the tree (*Minimum*)
- Finding maximum node of the tree
- Finding successor node of a specific node (*Successor*)
- Traverse the tree using an in-order walk
- Search a desired node in the tree (in case of not finding the key, NIL is returned) (*Search*) (this operation is identical for both regular and red black tree)
- Insert to a regular binary tree (in case of key existence, no operation is performed) (*BinaryTreeInsert*)
- Delete from a regular binary tree (in case of key absence, no operation is performed)
- Left rotate the tree using a specific node as pivot (*LeftRotate*)
- Right rotate the tree using a specific node as pivot (*RightRotate*)
- Insert to a red black tree (some color adjustment is also performed) (*Insert*) (It copes with case the key is already in the tree)
- Delete from a red black tree (some color adjustment is also performed) (*Delete*) (It behaves accordingly in case of key mismatch)
- Fixing up the colors of the nodes after deleting a node (*Fixup*)

3. Graphic

Besides the operation listed in previous section, some special routines for graphical view are developed. The code is smart enough to find the best position and size of the nodes in the screen. A sample view of the initial tree is enclosed.

The graphic is generated in the program. That is all the required operations are built in and no third party application is required. The graphic implementation is in UNIX and compatible with Glue system. However, it is general enough to be transported to a new platform such as Microsoft Windows or MS-DOS. This advantage is obtained using an appropriate level of abstraction such that all the graphical functions are wrapped up with abstract functions. Therefore, by writing the graphic library for a new platform we can use of all the graphical facilities of the program.

4. Initial Tree

As an initial tree, an all-black tree with 127 nodes (4 to 508) is generated in the program. This tree is generated using a linear time algorithm (i.e. $O(n)$). For this purpose nodes are generated from root to leaves and at each point, the key of the desired node is calculated based on the following formula:

If node is left child of its parent:

$$key = ParentKey - 2^{((InitialTreeHeight-2-Level)*InitialTreeStep)}$$

and if node is right child of its parent:

$$key = ParentKey + 2^{((InitialTreeHeight-2-Level)*InitialTreeStep)}$$

where

ParentKet: The key of the parent of the current node

Level: The Level of the current node or the height distance from the root

InitialTreeHeight: The maximum height of the tree (7 in this example)

InitialTreeStep: The difference between the *keys* of two successive nodes (4 in this example)

InitialTreeMin: The key of the most minimum node used in the following code (4 in this example)

The nodes are generated in a recursive Function with the following algorithm:

```
CREATE_INITIAL_TREE()  
    TRedBlackTree *bintree = new TRedBlackTree  
    Key = InitialTreeMin+(2^(InitialTreeHeight-1)-1)*InitialTreeStep  
    root->key = key  
    INSERT_NODE(root, key, 0, 0)  
    INSERT_NODE(root, key, 1, 0)
```

```

return root

INSERT_NODE(Parent, ParentKey, IsRightChild, Level) {
    if(Level<InitialTreeHeight-1) {
        if(IsRightChild=0)
            key = ParentKey-
                2^((InitialTreeHeight-2-Level)*InitialTreeStep)
        else
            key = ParentKey+
                2^((InitialTreeHeight-2-Level)*InitialTreeStep)
        node->key = key
        if(IsRightChild=0)
            Parent->Left = node
        else
            Parent->Right = node
        node->Parent = Parent
        INSERT_NODE(node, key, 0, Level+1)
        INSERT_NODE(node, key, 1, Level+1)
    }
}

```

Since this algorithm meets each node only once (in time of creation) so the order of complexity would be $O(n)$ where n is the number of nodes (127 in this example)

5. Algorithms Complexity Analysis

5.1. General Case Analysis

In the program the number of C instructions, which executed for each query are counted and reported.

We can find the number of operations in worst case using the following analysis:

For each query a number of functions are called and each function may call some other functions. By looking at the codes we find out that the following functions are dealt with in these queries.

The definition operations are mentioned in a former section:

1. *Minimum* (m)
2. *Successor* (u)
3. *Left/RightRotate* (r)
4. *Search* (s)
5. *BinaryTreeInsert* (b)
6. *Insret* (i)
7. *DeleteFixup* (f)
8. *Delete* (d)
9. *SearchQuery*
10. *InsertQuery*
11. *DeleteQuery*

For each of these operations, the number of instructions was counted. Since there are some conditional statements and also some loops in the codes, the number of instructions cannot be determined exactly but using some regular expression rules we can define them. The rules are:

[]: The operation may or may not be performed. This notation is used when we have a conditional statement.

/: It shows choice. It is being used in cases when an *if/else* statement exists and based upon the condition one of the two paths is followed.

xName: number of iterations of a loop in function Name. (e.g. *xi* is the number of iterations of a loop in function insert)

Name': When function Name is called recursively (e.g. *s'* is the number of instructions executed in the recursive call of function Search)

By using these rules, we can define the number of instructions of each mentioned function as following:

<i>Minimum</i> :	$m = 2 + xm * 2$
<i>Successor</i> :	$u = 2 + ((m) (2 + xu * 3))$
<i>Left/RightRotate</i> :	$r = 8 + [1] + [1]$
<i>Search</i> :	$s = 2 + [1 + s']$
<i>BinaryTreeInsert</i> :	$b = 6 + xb * 4 + [1]$
<i>Insert</i> :	$i = 4 + b + xi * 8 + [r + [2 + r]]$
<i>DeleteFixup</i> :	$f = 2 + (xf - 1) * 7 + (7 + [4 + r] + [4 + r + [4 + r]])$
<i>Delete</i> :	$d = 10 + [u] + [1] + [1] + [1 + f]$
<i>SearchQuery</i> :	$sq = s$
<i>InsertQuery</i> :	$iq = s + [i]$
<i>DeleteQuery</i> :	$dq = s + [d]$

As it can be seen, in some of these formulas we see name of some others. This is because we have some of the functions being called in others. For example to find successor of a node, we may need to find the minimum node. In this case, we call Minimum and we see *m* in formula of *s*.

Some of the formulas may need more explanations. So in the following, I briefly explain more:

Minimum: In this function we have a loop, which is repeated for *xu* number of times and 2 instructions are executed for each iteration. Two other instructions are executed unconditionally.

Successor: There is an *if/else* statement. We may go through *if* which requires us to call function *Minimum* or go to *else* part, which is a loop with *xu* number of iterations.

Left/RightRotate: Execution time of these two functions are identical, so we can look at one of them. We execute 8 instructions unconditionally and 1 or 2 more instructions may be executed according to *if* conditions. Note that since we consider execution time of all operations same, so if we have an *if/else* statement with same number of instructions in their bodies, we can consider it an unconditional statement.

Search: In case we do not hit the key, we go through a recursive call so we write it s' .

BinaryTreeInsert: From the formula we see that a loop runs for xb number of iterations and one instruction may or may not get executed.

Insert: We have a large *while* loop, which runs for xi number of times. We consider three cases for execution of this loop [1]. The important issue here is that if we enter cases 2 or 3, then the boolean condition of the loop is violated and loop is terminated. So we can consider that cases 2 or 3 can be called only in the last iteration and for the previous iterations we had only case 1. So by writing the formulas such that case one executed for $(xi-1)$ times and case 1 or 2 or 3 executed for the last iteration we come to the presented formula.

DeleteFixup: We have something similar to *Insert* function in this function. There are four different cases in the book [1]. We can see that if both case 1 and 2 or case 3 or 4 is entered, this would be the last iteration of the loop because the condition gets false. So the only way of surviving in the loop is having case 2 individually running. It means we can have functions Rotation at most three times. On the other iterations of the loop we have only fixed number of instructions (7) executed.

SearchQuery: This query is nothing but calling function Search and report the result, so executing time is similar to function search.

InsertQuery: For inserting a node, first we should make sure that the key does not exist in the tree. So this requires us to call Search first. Then based on the result of Search, we may or may not call insert function.

DeleteQuery: If the desired key cannot be located in the tree, we should not go to delete function. Then we call Search function unconditionally and consequently *Delete* function conditionally.

Delete: In this function we have a number of *if/else* statements and based on the conditions we may call function Successor and/or function *Fixup*.

5.2. Worst Case Analysis

To see what is the worst case of the algorithms, we should find out the maximum number of iterations for each loop. As it can be seen from the codes, all of the loops but one (insert) can be executed as many as the height of the tree, because in each iteration we

may go one level up or down on the tree. For function *Insert*, in each iteration we go two level up (parent of parent) so the number of iterations in worst case would be $height/2$. Therefore we can replace the constant used in the above formulas with these values:

$$\begin{aligned}x_m &= x_u = x_b = x_f = h \\ x_i &= h/2\end{aligned}$$

where h is the height of the tree.

So by using this maximum value and replacing the formulas used in the other ones, we find the following formulas:

$$\begin{aligned}m &= 2+2h \\ u &= 2 + ((2+2h) | (2+3h)) = 3h+4 \\ r &= 10 \\ s &= 3 + s' = 2 + h*3 = 3h+2 \\ b &= 4h+7 \\ i &= 4 + b + 8*(h/2) + 2*r + 2 = 4 + (4h+7) + 4h + 20 + 2 = 8h+33 \\ f &= 2 + (h-1)*7 + 7 + 4 + r + 4 + r + 4 + r = 7h+44 \\ d &= 13 + (3h+4) + (7h+44) = 10h+61 \\ sq &= 3h+2 \\ iq &= (3h+2) + (8h+33) = 11h+35 \\ dq &= (3h+2) + (10h+61) = 13h+63\end{aligned}$$

Also, according lemma 14.1 in the book [1], we know that the maximum height of a red black tree is $2lg(n+1)$ where n is the number of nodes in the tree. So we can find the maximum number of required instruction for the queries as following:

$$\begin{aligned}\text{Search Query: } & \mathbf{6lg(n+1)+2} \\ \text{Insert Query: } & \mathbf{22lg(n+1)+35} \\ \text{Delete Query: } & \mathbf{26lg(n+1)+63}\end{aligned}$$

5.3. Counted Instructions

At this point, some comments about the counted instructions are required. I tried to count all the possible instructions, so I counted all the following instructions:

1. *return* instruction
2. Function calls (besides the body of the functions)
3. *else if* instructions
4. Extra instruction of a loop in last iteration (just to test the condition)
5. Initialization of the variables

6. Sample Tree

As described earlier, program starts with a sample initial tree as defined. (Numbers 4 to 508 in 127 nodes) This tree can be seen in next page. More trees can be seen after applying the queries (next part of the project)

Finally, the source code of the program can be found as an attachment.

Reference

[1] Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., "Introduction to Algorithms", The MIT Press, 1997

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.