

Acad. Year
2020-21

Self Balancing Binary Tree

Self Balancing Binary Tree



Dehit Trivedi

Department of Electronic and Communication
Institute of Technology, Nirma University
Ahmedabad, India

Special Assignment : 2CSOE52 Data Structure

Self Balancing Binary Trees

SUBMITTED BY

Dehit Trivedi

(18BEC119)

Department of Electronic and Communication, Institute of Technology

Nirma University, Ahmedabad, India

Special Assignment presented to

Prof. Malaram Kumhar

Contents

1	Introduction	2
2	Red Black Trees	2
2.1	Algorithm	3
2.2	Results	3
2.3	Application	4
3	AVL Trees	4
3.1	Algorithm	5
3.2	Results	5
3.3	Application	5
4	Conclusion	5
5	Appendix	6
5.1	CODE FOR RED-BLACK TREE	6
5.2	CODE FOR AVL TREE	13

1 Introduction

Self balancing binary trees are height balanced tree which keeps its height as minimum as possible when a insertion or deletion operations are performed. Typically the height of the tree is kept in range of $\log(n)$ so that all operation cost $O(\log(n))$ average.

This project compares the three major self-balancing BST and determines the specific application of each of the algorithms and its possible real world application. The three self-balancing BST are;

- Red Black Tree
- AVL Tree
- Splay Tree

2 Red Black Trees

It is BST with an extra attribute for each node. The extra attribute is the colour. The colour of the node can either be black or red. So a structure of red black tree could be as shown;

```
1 struct RB_BST_node {  
2     int data;  
3     int color;  
4     struct RB_BST_node* parent;  
5     struct RB_BST_node* right;  
6     struct RB_BST_node* left;  
7 };
```

The properties of the Red-Black trees are;

- Root node is always black
- Every node is either red or black
- Every leaf which is NULL is black
- A red node cannot have red children

- Every simple path from node to descendant(NULL) contains the same number of black nodes.

2.1 Algorithm

Search Algorithm

```

1 search (tree, value)
2   Step #1 :
3   If tree -> data = value OR tree = NULL
4       Return tree
5   Else If value < data
6       Return search(tree -> left , value)
7   Else
8       Return search(tree -> right , value)
9   END

```

Insert Algorithm

```

1 insert(value)
2   If tree = NULL
3       Insert the new value as root with color Black
4   Else
5       Insert new value as leaf with color red
6   If parent of new is black
7       exit
8   else
9       Check the color of parent sibling
10      if color of parent sibling = black
11          perform suitable rotation
12      else
13          change the color and recheck
14  END

```

2.2 Results

The code for the Red Black Tree implementation is in Appendix 5.1.

```

(base) Dehiti-MacBook-Air:~ dehittrivedi$ /var/folders/m1/zy3xgbxs18gbs8sw35g67jhw0000gn/T/18BEC119_DSSA_A ; exit;
Insert elements

Number of key: 4
Enter key: 1
Enter key: 8
Enter key: 4
Enter key: 6
Tree

1      4      6      8
Enter a key to be searched

Enter key: 4
0x7f8834405950
minimum test

MIN: 1
Tree

1      4      6      8
-----
The elapsed time is 0.000482 secondslogout

```

Figure 1: Result for RB tree

2.3 Application

The Major application of the Red Black tree is in Multi-Set, Multi-Map, Map and Set. A recent article also explores the use of Red-Black tree in Wireless sensor network in order to achieve substantial efficiency in network life time and energy efficiency.

3 AVL Trees

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Structure of AVL node

```

1 Struct AVLNode
2 {
3     int data;
4     struct AVLNode *left, *right;
5     int balfactor;
6 };

```

The important feature of a AVL tree is that the difference between the depth of right and left subtrees cannot be more than one. In order to maintain this feature, an implementation of an AVL will include an algorithm to rebalance the tree when adding an additional element would

upset this feature.

3.1 Algorithm

Insert Algorithm

```
1 Insert the element using BST insertion
2 Check for the balance factor of each node
3 If balance factor = [-1,0,1]
4     Proceed to next operation
5 Else
6     The tree is imbalanced, perform suitable rotations to balance
7
8 END
```

3.2 Results

The code for the AVL Tree is in Appendix 5.2

```
(base) Dehitis-MacBook-Air:~ dehittrivedi$ /var/folders/ml/zy3xgbxs18gbs8sw35g67jhw0000gn/T/18BEC119_DSSA_B ; exit;
Insert elements

Number of key: 4
Enter key: 1
Enter key: 2
Enter key: 3
Enter key: 4
Preorder traversal of the constructed AVL tree is
2 1 3 4
-----
The elapsed time is 0.000459 secondslogout
```

Figure 2: Result for AVL tree

3.3 Application

The major use of AVL tree in database transactions and in memory sorts of set and dictionaries.

4 Conclusion

This project helps to understand different types of self balancing binary tree. The implementation of 2 common self-balancing BST was demonstrated in the project. The splay tree are also

Self-balancing BST however due to its complexity and lack of resources its implementation was not covered in the project.

The AVL and Red-Black both of them are complex algorithm however this algorithm help to achieve better time and space complexity. This algorithm brings the average complexity to $O(\log_2(n))$.

5 Appendix

5.1 CODE FOR RED-BLACK TREE

```
1  /*
2  Data Structure Special assignment
3  Dehit Trivedi - 18BEC119
4  RED - BLACK TREE
5
6  */
7
8  #include <stdio.h>
9  #include <time.h>
10 #include <stdlib.h>
11
12 #define RED 0
13 #define BLACK 1
14
15 struct node{
16     int key;
17     int color;
18     struct node *parent;
19     struct node *left;
20     struct node *right;
21 };
22
23 struct node *ROOT;
24 struct node *NIL;
```



```

25
26 void left_rotate(struct node *x);
27 void right_rotate(struct node *x);
28 void tree_print(struct node *x);
29 void red_black_insert(int key);
30 void red_black_insert_fixup(struct node *z);
31 struct node *tree_search(int key);
32 struct node *tree_minimum(struct node *x);
33 void red_black_transplant(struct node *u, struct node *v);
34 void red_black_delete(struct node *z);
35 void red_black_delete_fixup(struct node *x);
36
37 int main(){
38     double time_spent = 0.0;
39     clock_t begin = clock();
40     NILL = malloc(sizeof(struct node));
41     NILL->color = BLACK;
42
43     ROOT = NILL;
44
45     printf("Insert elements\n\n");
46
47     int i, key;
48     printf("Number of key: ");
49     scanf("%d", &i);
50     while(i--){
51         printf("Enter key: ");
52         scanf("%d", &key);
53         red_black_insert(key);
54     }
55
56     printf("Tree\n\n");
57     tree_print(ROOT);
58     printf("\n");
59

```

```

60 printf("Enter a key to be searhced\n\n");
61 printf("Enter key: ");
62 scanf("%d", &key);
63 printf((tree_search(key) == NULL) ? "NULL\n" : "%p\n", tree_search(key
    ));
64
65 printf("minimum test\n\n");
66 printf("MIN: %d\n", (tree_minimum(ROOT))->key);
67
68
69 printf("Tree \n\n");
70 tree_print(ROOT);
71 printf("\n");
72 printf("\n");
73 clock_t end = clock();
74 time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
75 printf("-----\n");
76 printf("The elapsed time is %f seconds", time_spent);
77
78 return 0;
79 }
80
81 void tree_print(struct node *x){
82     if(x != NULL){
83         tree_print(x->left);
84         printf("%d\t", x->key);
85         tree_print(x->right);
86     }
87 }
88
89 struct node *tree_search(int key){
90     struct node *x;
91
92     x = ROOT;
93     while(x != NULL && x->key != key){

```

```

94     if(key < x->key){
95         x = x->left;
96     }
97     else{
98         x = x->right;
99     }
100 }
101
102 return x;
103 }
104
105 struct node *tree_minimum(struct node *x){
106     while(x->left != NULL){
107         x = x->left;
108     }
109     return x;
110 }
111
112 void red_black_insert(int key){
113     struct node *z, *x, *y;
114     z = malloc(sizeof(struct node));
115
116     z->key = key;
117     z->color = RED;
118     z->left = NULL;
119     z->right = NULL;
120
121     x = ROOT;
122     y = NULL;
123
124     /*
125      * Go through the tree untill a leaf(NULL) is reached. y is used for
126      * keeping
127      * track of the last non-NULL node which will be z's parent.
128      */

```

```

128 while(x != NULL){
129     y = x;
130     if(z->key <= x->key){
131         x = x->left;
132     }
133     else{
134         x = x->right;
135     }
136 }
137
138 if(y == NULL){
139     ROOT = z;
140 }
141 else if(z->key <= y->key){
142     y->left = z;
143 }
144 else{
145     y->right = z;
146 }
147
148 z->parent = y;
149
150 red_black_insert_fixup(z);
151 }
152
153 void red_black_insert_fixup(struct node *z){
154     while(z->parent->color == RED){
155
156         /* z's parent is left child of z's grand parent*/
157         if(z->parent == z->parent->parent->left){
158
159             /* z's grand parent's right child is RED */
160             if(z->parent->parent->right->color == RED){
161                 z->parent->color = BLACK;
162                 z->parent->parent->right->color = BLACK;

```

```

163     z->parent->parent->color = RED;
164     z = z->parent->parent;
165 }
166
167 /* z's grand parent's right child is not RED */
168 else{
169
170     /* z is z's parent's right child */
171     if(z == z->parent->right){
172         z = z->parent;
173         left_rotate(z);
174     }
175
176     z->parent->color = BLACK;
177     z->parent->parent->color = RED;
178     right_rotate(z->parent->parent);
179 }
180 }
181
182 /* z's parent is z's grand parent's right child */
183 else{
184
185     /* z's left uncle or z's grand parent's left child is also RED */
186     if(z->parent->parent->left->color == RED){
187         z->parent->color = BLACK;
188         z->parent->parent->left->color = BLACK;
189         z->parent->parent->color = RED;
190         z = z->parent->parent;
191     }
192
193     /* z's left uncle is not RED */
194     else{
195         /* z is z's parents left child */
196         if(z == z->parent->left){
197             z = z->parent;

```

```

198     right_rotate(z);
199 }
200
201     z->parent->color = BLACK;
202     z->parent->parent->color = RED;
203     left_rotate(z->parent->parent);
204 }
205 }
206 }
207
208 ROOT->color = BLACK;
209 }
210
211 void left_rotate(struct node *x){
212     struct node *y;
213
214     /* Make y's left child x's right child */
215     y = x->right;
216     x->right = y->left;
217     if(y->left != NULL){
218         y->left->parent = x;
219     }
220
221     /* Make x's parent y's parent and y, x's parent's child */
222     y->parent = x->parent;
223     if(y->parent == NULL){
224         ROOT = y;
225     }
226     else if(x == x->parent->left){
227         x->parent->left = y;
228     }
229     else{
230         x->parent->right = y;
231     }
232

```

```

233  /* Make x, y's left child & y, x's parent */
234  y->left = x;
235  x->parent = y;
236  }
237
238  void right_rotate(struct node *x){
239      struct node *y;
240
241      /* Make y's right child x's left child */
242      y = x->left;
243      x->left = y->right;
244      if(y->right != NULL){
245          y->right->parent = x;
246      }
247
248      /* Make x's parent y's parent and y, x's parent's child */
249      y->parent = x->parent;
250      if(y->parent == NULL){
251          ROOT = y;
252      }
253      else if(x == x->parent->left){
254          x->parent->left = y;
255      }
256      else{
257          x->parent->right = y;
258      }
259      y->right = x;
260      x->parent = y;
261  }

```

5.2 CODE FOR AVL TREE

```

1  /*
2  Data Structure Special assignment
3  Dehit Trivedi - 18BEC119

```

```

4 AVL TREE : Insertion
5 */
6
7 #include<stdio.h>
8 #include <time.h>
9 #include<stdlib.h>
10
11 struct Node
12 {
13     int key;
14     struct Node *left;
15     struct Node *right;
16     int height;
17 };
18
19 int max(int a, int b);
20 int height(struct Node *N)
21 {
22     if (N == NULL)
23         return 0;
24     return N->height;
25 }
26
27 int max(int a, int b)
28 {
29     return (a > b)? a : b;
30 }
31
32 struct Node* newNode(int key)
33 {
34     struct Node* node = (struct Node*)
35         malloc(sizeof(struct Node));
36     node->key    = key;
37     node->left   = NULL;
38     node->right  = NULL;

```



```

39     node->height = 1;
40     return(node);
41 }
42
43 struct Node *rightRotate(struct Node *y)
44 {
45     struct Node *x = y->left;
46     struct Node *T2 = x->right;
47
48     x->right = y;
49     y->left = T2;
50
51     y->height = max(height(y->left), height(y->right))+1;
52     x->height = max(height(x->left), height(x->right))+1;
53
54     return x;
55 }
56 struct Node *leftRotate(struct Node *x)
57 {
58     struct Node *y = x->right;
59     struct Node *T2 = y->left;
60
61     y->left = x;
62     x->right = T2;
63
64     x->height = max(height(x->left), height(x->right))+1;
65     y->height = max(height(y->left), height(y->right))+1;
66
67     return y;
68 }
69
70 int getBalance(struct Node *N)
71 {
72     if (N == NULL)
73         return 0;

```

```

74     return height(N->left) - height(N->right);
75 }
76
77 struct Node* insert(struct Node* node, int key)
78 {
79
80     if (node == NULL)
81         return(newNode(key));
82
83     if (key < node->key)
84         node->left = insert(node->left, key);
85     else if (key > node->key)
86         node->right = insert(node->right, key);
87     else
88         return node;
89
90     node->height = 1 + max(height(node->left),
91                           height(node->right));
92
93     int balance = getBalance(node);
94
95     if (balance > 1 && key < node->left->key)
96         return rightRotate(node);
97
98     if (balance < -1 && key > node->right->key)
99         return leftRotate(node);
100
101     if (balance > 1 && key > node->left->key)
102     {
103         node->left = leftRotate(node->left);
104         return rightRotate(node);
105     }
106
107     if (balance < -1 && key < node->right->key)
108     {

```

```

109     node->right = rightRotate(node->right);
110     return leftRotate(node);
111 }
112
113     return node;
114 }
115
116 void preOrder(struct Node *root)
117 {
118     if(root != NULL)
119     {
120         printf("%d ", root->key);
121         preOrder(root->left);
122         preOrder(root->right);
123     }
124 }
125
126 int main()
127 {
128     double time_spent = 0.0;
129     clock_t begin = clock();
130     struct Node *root = NULL;
131
132     printf("Insert elements\n\n");
133
134     int i, key;
135     printf("Number of key: ");
136     scanf("%d", &i);
137     while(i--){
138         printf("Enter key: ");
139         scanf("%d", &key);
140         root = insert(root, key);
141     }
142     /*
143     root = insert(root, 10);

```

```

144 root = insert(root, 20);
145 root = insert(root, 30);
146 root = insert(root, 40);
147 root = insert(root, 50);
148 root = insert(root, 25);
149 */
150 printf("Preorder traversal of the constructed AVL "
151        " tree is \n");
152 preOrder(root);
153 printf("\n");
154 clock_t end = clock();
155 time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
156 printf("-----\n");
157 printf("The elapsed time is %f seconds", time_spent);
158 return 0;
159 }

```