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

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;

```
struct RB_BST_node {
    int data;
    int color;
    struct RB_BST_node* parent;
    struct RB_BST_node* right;
    struct RB_BST_node* left;
    };
```

The properties of the Red-Black trees are;

- Root note is always black
- Every node is either red of 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

```
search (tree, value)
    Step #1 :
    If tree -> data = value OR tree = NULL
        Return tree
    Else If value < data
        Return search(tree -> left , value)
    Else
        Return search(tree -> righ , value)
    END
```

Insert Algorithm

```
insert(value)
    If tree = NULL
        Insert the new value as root with color Black
    Else
        Insert new value as leaf with color red
    If parent of new is black
        exit
    else
        Check the color of parent sibling
        if color of parent sibling = black
            perform suitable rotation
        else
            change the color and recheck
    END
```

2.2 Results

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

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 the 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

```
Struct AVLNode
{
    int data;
    struct AVLNode *left, *right;
    int balfactor;
};
```

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

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

END
```

3.2 Results

The code for the AVL Tree is in Appendix 5.2

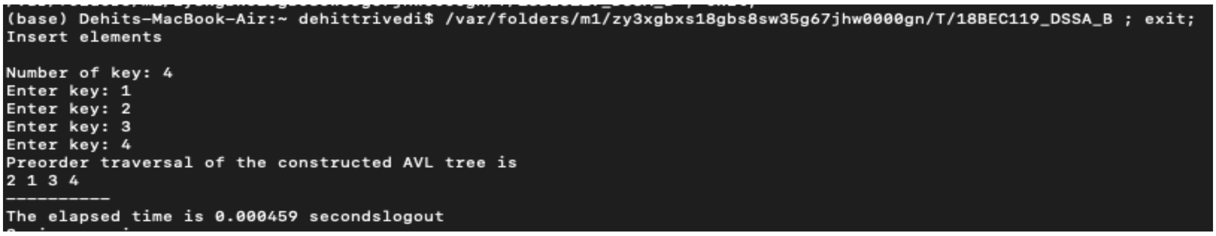

Figure 2: Result for AVL tree

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

```
/*
Data Structure Special assignment
Dehit Trivedi - 18BEC119
RED - BLACK TREE

*/

#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define RED 0
#define BLACK 1

struct node{
 int key;
 int color;
 struct node *parent;
 struct node *left;
 struct node *right;
};

struct node *ROOT;
struct node *NILL;
```

```
void left_rotate(struct node *x);
void right_rotate(struct node *x);
void tree_print(struct node *x);
void red_black_insert(int key);
void red_black_insert_fixup(struct node *z);
struct node *tree_search(int key);
struct node *tree_minimum(struct node *x);
void red_black_transplant(struct node *u, struct node *v);
void red_black_delete(struct node *z);
void red_black_delete_fixup(struct node *x);

int main(){
  double time_spent = 0.0;
  clock_t begin = clock();
 NILL = malloc(sizeof(struct node));
 NILL->color = BLACK;

 ROOT = NILL;

 printf("Insert elements\n\n");

 int i, key;
 printf("Number of key: ");
 scanf("%d", &i);
 while(i--){
  printf("Enter key: ");
  scanf("%d", &key);
  red_black_insert(key);
 }

  printf("Tree\n\n");
 tree_print(ROOT);
 printf("\n");

```

```
 printf("Enter a key to be searhced\n\n");
 printf("Enter key: ");
 scanf("%d", &key);
 printf((tree_search(key) == NILL) ? "NILL\n" : "%p\n", tree_search(key
   ));

 printf("minimum test\n\n");
 printf("MIN: %d\n", (tree_minimum(ROOT))->key);


 printf("Tree \n\n");
 tree_print(ROOT);
 printf("\n");
  printf("\n");
  clock_t end = clock();
  time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
  printf("----------\n");
  printf("The elapsed time is %f seconds", time_spent);

 return 0;
}

void tree_print(struct node *x){
 if(x != NILL){
  tree_print(x->left);
  printf("%d\t", x->key);
  tree_print(x->right);
 }
}

struct node *tree_search(int key){
 struct node *x;

 x = ROOT;
 while(x != NILL && x->key != key){
```

```
   if(key < x->key){
    x = x->left;
   }
   else{
    x = x->right;
   }
  }

  return x;
}

struct node *tree_minimum(struct node *x){
  while(x->left != NILL){
   x = x->left;
  }
  return x;
}

void red_black_insert(int key){
  struct node *z, *x, *y;
  z = malloc(sizeof(struct node));

  z->key = key;
  z->color = RED;
  z->left = NILL;
  z->right = NILL;

  x = ROOT;
  y = NILL;

  /*
   * Go through the tree untill a leaf(NILL) is reached. y is used for
     keeping
   * track of the last non-NILL node which will be z's parent.
   */
```

```
 while(x != NILL){
  y = x;
  if(z->key <= x->key){
   x = x->left;
  }
  else{
   x = x->right;
  }
 }

 if(y == NILL){
  ROOT = z;
 }
 else if(z->key <= y->key){
  y->left = z;
 }
 else{
  y->right = z;
 }

 z->parent = y;

 red_black_insert_fixup(z);
}

void red_black_insert_fixup(struct node *z){
 while(z->parent->color == RED){

  /* z's parent is left child of z's grand parent*/
  if(z->parent == z->parent->parent->left){

   /* z's grand parent's right child is RED */
   if(z->parent->parent->right->color == RED){
    z->parent->color = BLACK;
    z->parent->parent->right->color = BLACK;
```

```
    z->parent->parent->color = RED;
    z = z->parent->parent;
   }

   /* z's grand parent's right child is not RED */
   else{

    /* z is z's parent's right child */
    if(z == z->parent->right){
     z = z->parent;
     left_rotate(z);
    }

    z->parent->color = BLACK;
    z->parent->parent->color = RED;
    right_rotate(z->parent->parent);
   }
  }

  /* z's parent is z's grand parent's right child */
  else{

   /* z's left uncle or z's grand parent's left child is also RED */
   if(z->parent->parent->left->color == RED){
    z->parent->color = BLACK;
    z->parent->parent->left->color = BLACK;
    z->parent->parent->color = RED;
    z = z->parent->parent;
   }

   /* z's left uncle is not RED */
   else{
    /* z is z's parents left child */
    if(z == z->parent->left){
     z = z->parent;
```

```
     right_rotate(z);
    }

    z->parent->color = BLACK;
    z->parent->parent->color = RED;
    left_rotate(z->parent->parent);
   }
  }
 }

 ROOT->color = BLACK;
}

void left_rotate(struct node *x){
 struct node *y;

 /* Make y's left child x's right child */
 y = x->right;
 x->right = y->left;
 if(y->left != NILL){
  y->left->parent = x;
 }

 /* Make x's parent y's parent and y, x's parent's child */
 y->parent = x->parent;
 if(y->parent == NILL){
  ROOT = y;
 }
 else if(x == x->parent->left){
  x->parent->left = y;
 }
 else{
  x->parent->right = y;
 }

```

```
 /* Make x, y's left child & y, x's parent */
 y->left = x;
 x->parent = y;
}

void right_rotate(struct node *x){
 struct node *y;

 /* Make y's right child x's left child */
 y = x->left;
 x->left = y->right;
 if(y->right != NILL){
  y->right->parent = x;
 }

 /* Make x's parent y's parent and y, x's parent's child */
 y->parent = x->parent;
 if(y->parent == NILL){
  ROOT = y;
 }
 else if(x == x->parent->left){
  x->parent->left = y;
 }
 else{
  x->parent->right = y;
 }
  y->right = x;
 x->parent = y;
}
```

5.2 CODE FOR AVL TREE

```
/*
Data Structure Special assignment
Dehit Trivedi - 18BEC119
```

```
AVL TREE : Insertion
*/

#include<stdio.h>
#include <time.h>
#include<stdlib.h>

struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

int max(int a, int b);
int height(struct Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

int max(int a, int b)
{
    return (a > b)? a : b;
}

struct Node* newNode(int key)
{
    struct Node* node = (struct Node*)
                        malloc(sizeof(struct Node));
    node->key   = key;
    node->left   = NULL;
    node->right  = NULL;
```

```
    node->height = 1;
    return(node);
}

struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    return x;
}
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    return y;
}

int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
```

```c
    return height(N->left) - height(N->right);
}

struct Node* insert(struct Node* node, int key)
{

    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node;

    node->height = 1 + max(height(node->left),
                           height(node->right));

    int balance = getBalance(node);

    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    if (balance > 1 && key > node->left->key)
    {
        node->left =  leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && key < node->right->key)
    {
```

```
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

void preOrder(struct Node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

int main()
{
  double time_spent = 0.0;
  clock_t begin = clock();
  struct Node *root = NULL;

  printf("Insert elements\n\n");

  int i, key;
 printf("Number of key: ");
 scanf("%d", &i);
 while(i--){
  printf("Enter key: ");
  scanf("%d", &key);
  root = insert(root,key);
 }
  /*
  root = insert(root, 10);
```

```
  root = insert(root, 20);
  root = insert(root, 30);
  root = insert(root, 40);
  root = insert(root, 50);
  root = insert(root, 25);
  */
  printf("Preorder traversal of the constructed AVL"
         " tree is \n");
  preOrder(root);
  printf("\n");
  clock_t end = clock();
  time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
  printf("----------\n");
  printf("The elapsed time is %f seconds", time_spent);
  return 0;
}
```