Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

Performance Evaluation of Unbalanced Tree[Binary Search] and Balanced Tree[Red Black] based on height and average number of key comparisons as two metrics

Binary Search Tree

It is a node-based binary tree data structure where each node contains a key satisfying binary search tree property. The binary search tree properties are:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- There must not be any duplicate nodes.

Keeping these constraints binary search tree is constructed with key and count values. Key value is basically used to compare with the value entered with previous value and count to keep count of repeated values.

The question specifies for two binary search trees one for 500 random numbers from 1 to 100 and second for 400 random numbers from 1 to 100 and even numbers from 2 to 100 and odd numbers from 1 to 99 making it a total of 500 values.

Since the series may contain repeated values the count field is employed to hold same numbers encountered and not effect the tree.

The program flow of binary search tree is as follows. At first the random numbers 1 to 100 are being inserted in insert function. In the insert function the first random number is always the root. After that a new key is always inserted at leaf. We start searching a key from root till we hit a leaf node according to the binary search tree property. That is if the new element inputed is less than the root we traverse to the left else to the root. The subsequent numbers are inserted accordingly by comparing with keys. Once a leaf node is found, the new node is added as a child of the leaf node. On the other hand if the same number is encountered the count is incremented and element is not inserted. Then to get the average number of key comparisons search is employed where element is first compared with root, if the key is present at root, root is returned. If key is greater than root's key, recur for right subtree of root node. Otherwise recur for left subtree. For search in each node the search gets incremented until the required element is reached.
For example if element to be searched is 6 and it is located at root→left→right then its height is 3 as first it is compared with root then with parent left and then with itself. This height is multiplied by count and then added together with all elements from 1 to 100 each with its height and count multiplied. Finally the total value is divided by 500 to get average key comparisons. For height or maximum depth calculation just the height function with highest number of nodes along the path is returned.

The output obtained through binary search tree is as shown below
[1] A sequence of 500 random numbers each between 1 & 100.

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees

ITCS-6114/002 Data Structures and Algorithms

```
skoralah@smita: /media/skoralah/Smita/data structures/proj
skoralah@smita:/media/skoralah/Smita/data structures/proj$ ./prog1_bst
Element inserted:84
Element inserted:87
Element inserted:78
Element inserted:16
Element inserted:94
Element inserted:36
Element inserted:87
Element inserted:93
Element inserted:50
Element inserted:22
Element inserted:63
Element inserted:28
Element inserted:91
Element inserted:60
Element inserted:64
Element inserted:27
Element inserted:41
Element inserted:27
Element inserted:73
Element inserted:37
Element inserted:12
Element inserted:69
Element inserted:68
```

500 elements are inserted accordingly as shown

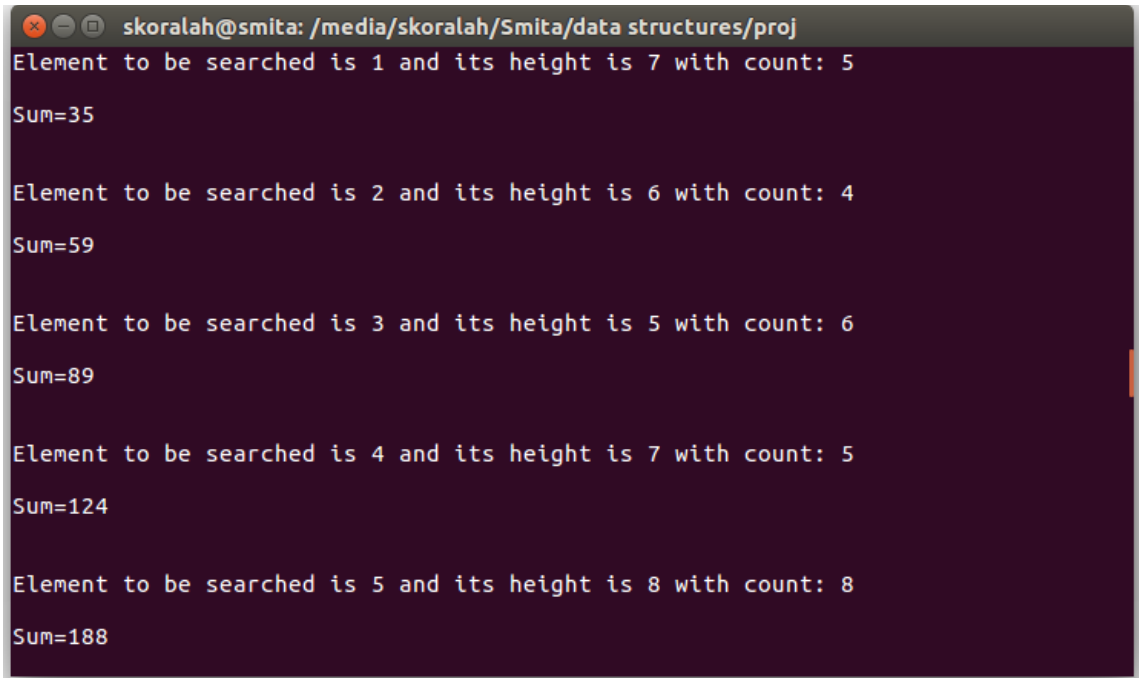
```
skoralah@smita: /media/skoralah/Smita/data structures/proj
Element inserted:33

Inorder Traversal : number - count
1 - 5           2 - 4           3 - 6           4 - 5           5 - 8
6 - 7           7 - 4           8 - 3           9 - 5           10 - 3
11 - 2          12 - 6          13 - 2          14 - 7          15 - 3
16 - 4          17 - 1          18 - 4          19 - 4          20 - 6
21 - 2          22 - 9          23 - 6          24 - 2          25 - 9
26 - 4          27 - 8          28 - 7          29 - 11                 30 - 13
 31 - 4         32 - 2          33 - 5          34 - 4          35 - 4
36 - 5          37 - 10                 38 - 4          39 - 3          40 - 6
41 - 9          42 - 3          43 - 7          44 - 9          45 - 5
46 - 3          47 - 5          48 - 1          49 - 5          50 - 5
51 - 7          52 - 4          53 - 3          54 - 3          55 - 4
56 - 5          57 - 7          58 - 2          59 - 5          60 - 6
61 - 5          62 - 2          63 - 4          64 - 4          65 - 5
66 - 4          67 - 1          68 - 10                 69 - 8          70 - 6
71 - 5          72 - 4          73 - 7          74 - 5          75 - 1
76 - 3          77 - 5          78 - 2          79 - 3          80 - 5
81 - 2          82 - 7          83 - 5          84 - 3          85 - 9
86 - 2          87 - 7          88 - 4          89 - 6          90 - 3
91 - 10         92 - 2          93 - 6          94 - 5          95 - 4
96 - 5          97 - 6          98 - 7          99 - 4          100 - 9
```

In order traversal is done and the element and corresponding count is displayed. As seen element 1 is repeated 5 times.

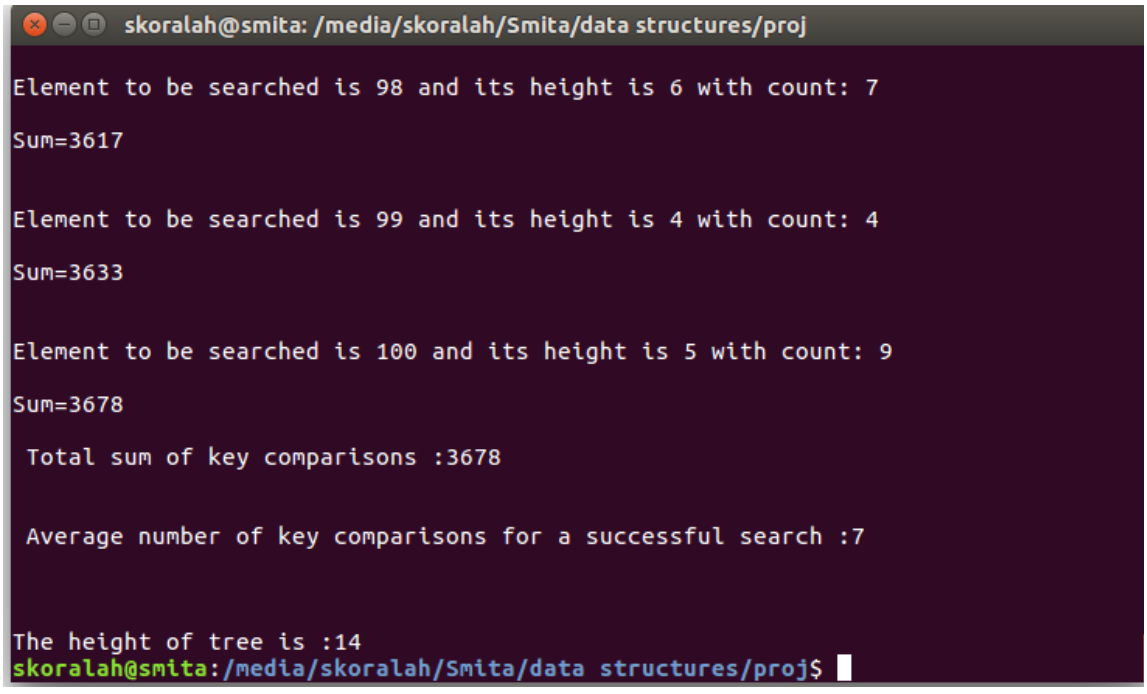
Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees

ITCS-6114/002 Data Structures and Algorithms



Statements are printed accordingly height of each element is nothing but key comparison of each element from the root in a search.



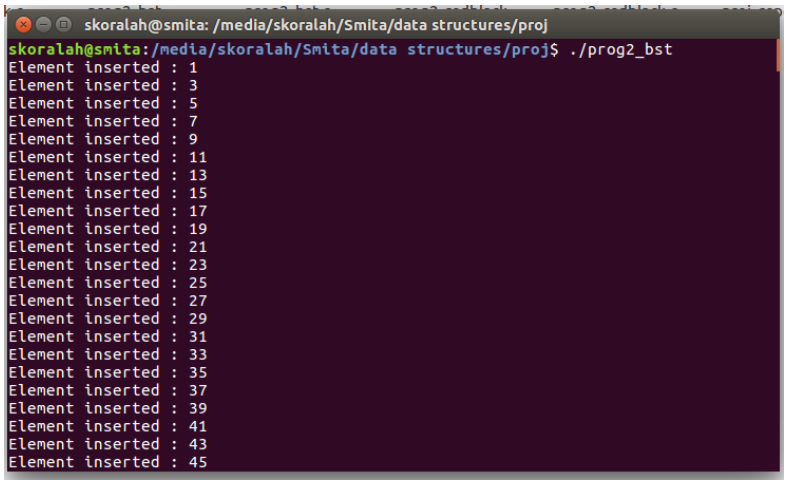
Finally all the elements added and divided by 500 to get average key comparisons as 7 and height that is the maximum depth as 14

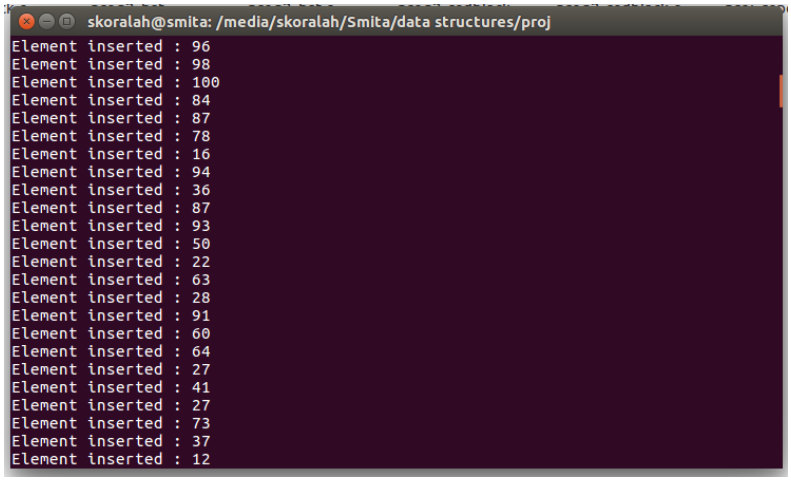
Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

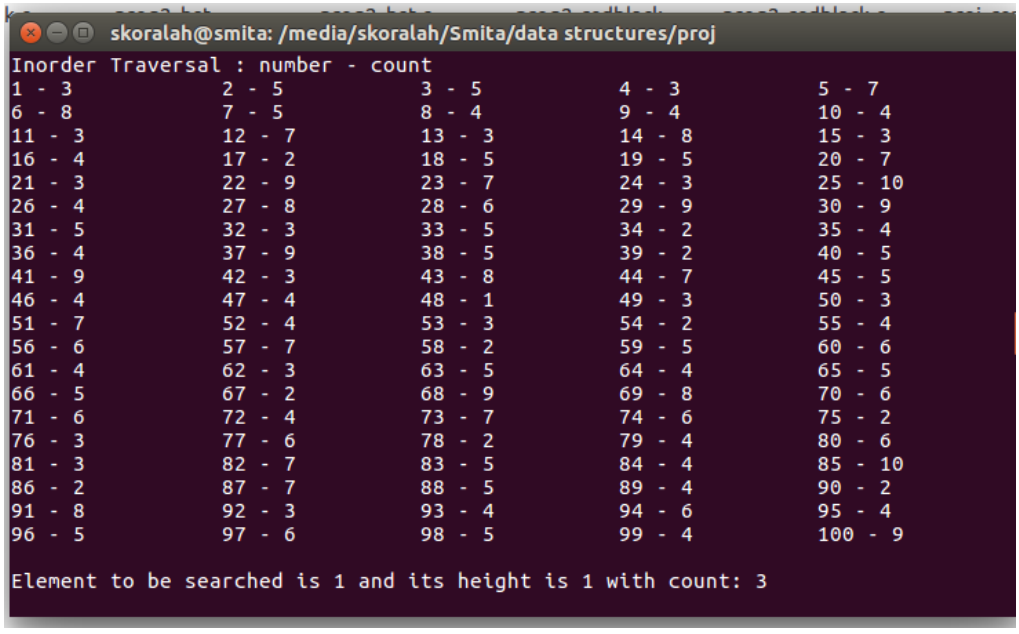
The height ranged from 10 to 15 and average key comparisons from 6 to 8 with **srand(time(NULL))** parameter. It is commented off in program to get same sequence for all 4 trees.

[2] 1, 3, 5,..., 99, 2, 4, 6,..., 100, followed by 400 random numbers between 1 & 100.





As seen first odd numbers and then even numbers are inserted and random numbers are inserted. Random numbers are same in both first and second case since srand(time(NULL)) isn't used.

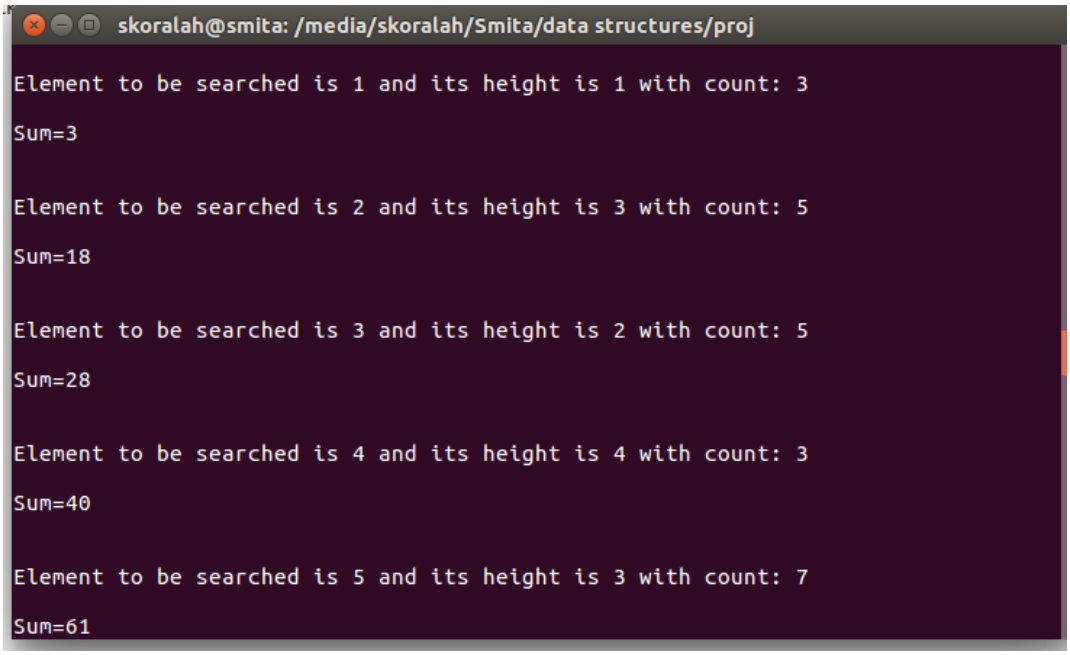


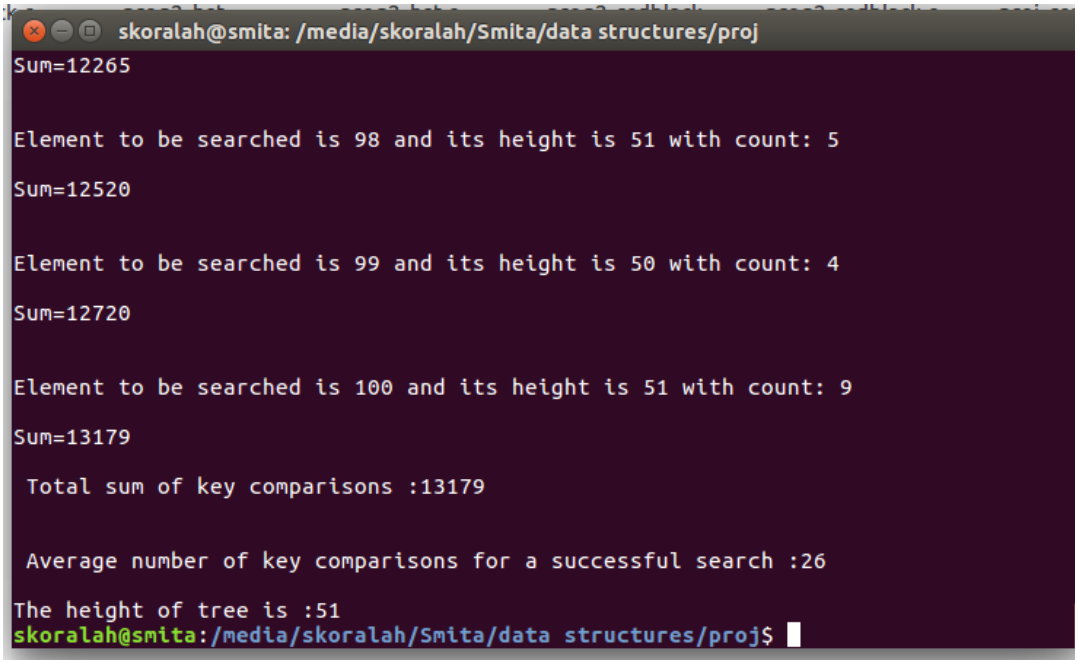
In order traversal and corresponding count display

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees

ITCS-6114/002 Data Structures and Algorithms





Height remained constant as 51 for different numbers since first the even and odd numbers are inserted and then 400 random numbers just increment count. Average key comparisons ranged from 24 to 28.

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

Red Black Tree
It is a self-balancing Binary Search Tree where every node follows following rules:

- Root of tree is always black.
- Every node has a color either red or black.
- There are no two adjacent red nodes.
- Every path from root to a NULL node has same number of black nodes.

Keeping these constraints Red Black Tree is constructed with key, tag and count values. The tag specifies whether the number is either is red or black. Instead of giving as tag as separate parameter the element is specified as **r for red and b for black** explicitly. The question specifies for two Red Black Trees one for 500 random numbers from 1 to 100 and second for 400 random numbers from 1 to 100 and even numbers from 2 to 100 and odd numbers from 1 to 99 making it a total of 500 values.

The program flow of Red Black Tree is as follows. At first the random numbers 1 to 100 are being inserted in insert function. In the insert function the first random number is always the root and color is made as black. After that a new key is always inserted at leaf. We start searching a key from root till we hit a leaf node according to the binary search tree property. That is if the new element inputed is less than the root we traverse to the left else to the root. The subsequent numbers are inserted accordingly by comparing with keys. Once a leaf node is found, the new node is added as a child of the leaf node. New node added is always red node. Then insertfix function is called in order to satisfy red black tree property. In insertfix function two different tasks are done. The parent of new node inserted is checked for color. If black then returned, if red then corresponding sibling is checked. If red sibling then recolored and checked again until property is satisfied, if black or absent sibling tree is rotated and recolored. According to the need left or right rotate is called accordingly.

On the other hand if the same number is encountered the count is incremented and element is not inserted. Then to get the average number of key comparisons search is employed similar to binary tree and other procedures are followed accordingly.

The output obtained through Red Black Tree is as shown below
[1] A sequence of 500 random numbers each between 1 & 100.

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees

ITCS-6114/002 Data Structures and Algorithms

```
skoralah@smita: /media/skoralah/Smita/data structures/proj
skoralah@smita:/media/skoralah/Smita/data structures/proj$ ./prog1_redblack

Element inserted:84

Element inserted:87

Element inserted:78

Element inserted:16

Element inserted:94

Element inserted:36

Element inserted:87

Element inserted:93

Element inserted:50

Element inserted:22

Element inserted:63
```

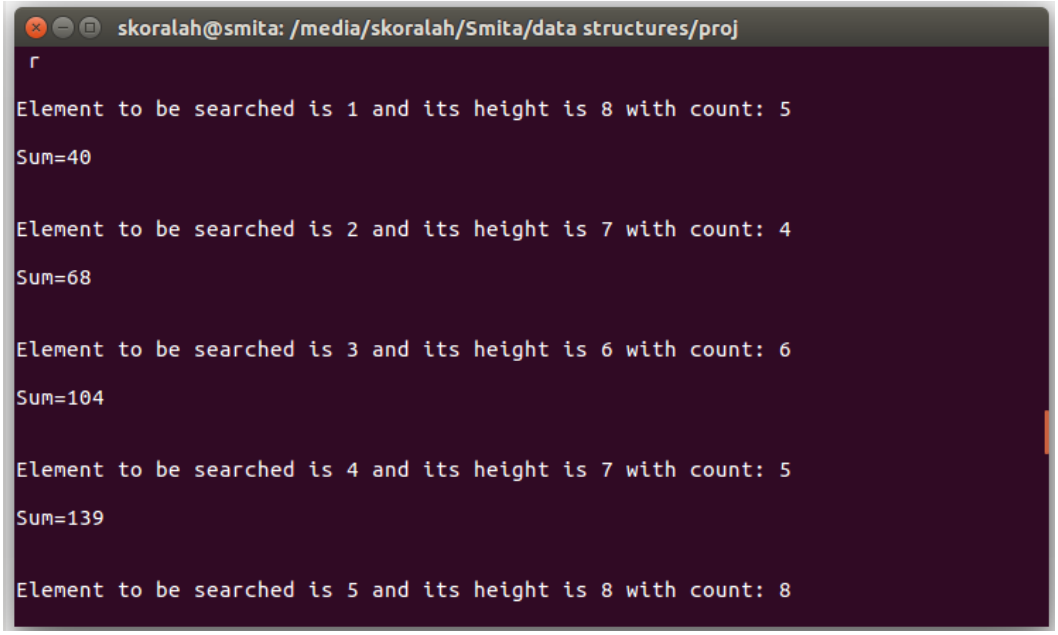
Elements inserted

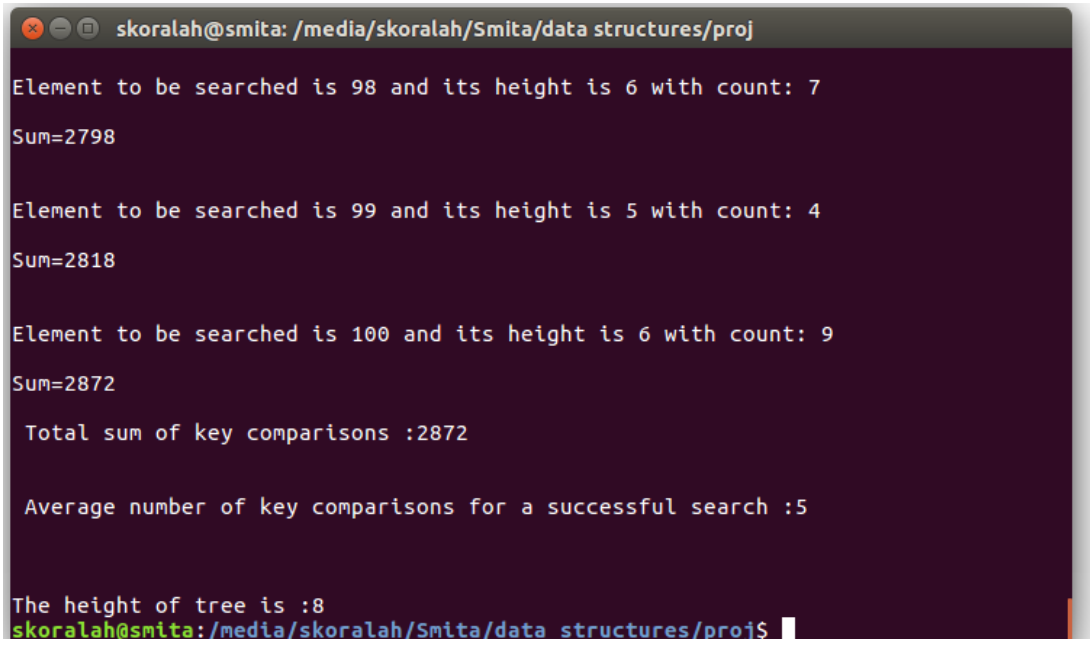
```
skoralah@smita: /media/skoralah/Smita/data structures/proj

Inorder Traversal : number-count-tag
1-5 - r          2-4 - b                3-6 - r                4-5 - b
5-8 - r          6-7 - b                7-4 - b                8-3 - r
9-5 - r          10-3 - b               11-2 - r               12-6 - r
13-2 - r                 14-7 - b               15-3 - r               16-4 -
b                17-1 - r               18-4 - b               19-4 - r
20-6 - b                 21-2 - r               22-9 - b               23-6 -
b                24-2 - b               25-9 - r               26-4 - b
27-8 - r                 28-7 - r               29-11 - r              30-13 -
 b               31-4 - b               32-2 - b               33-5 - r
34-4 - r                 35-4 - b               36-5 - r               37-10 -
 b               38-4 - r               39-3 - r               40-6 - b
41-9 - b                 42-3 - b               43-7 - r               44-9 -
b                45-5 - r               46-3 - b               47-5 - r
48-1 - b                 49-5 - r               50-5 - b               51-7 -
b                52-4 - b               53-3 - b               54-3 - r
55-4 - b                 56-5 - r               57-7 - r               58-2 -
b                59-5 - b               60-6 - r               61-5 - b
62-2 - r                 63-4 - b               64-4 - b               65-5 -
r                66-4 - b               67-1 - r               68-10 - b
69-8 - b                 70-6 - r               71-5 - b               72-4 -
r                73-7 - r               74-5 - b               75-1 - r
76-3 - r                 77-5 - b               78-2 - b               79-3 -
```

Number with its corresponding count and tag fields.

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms





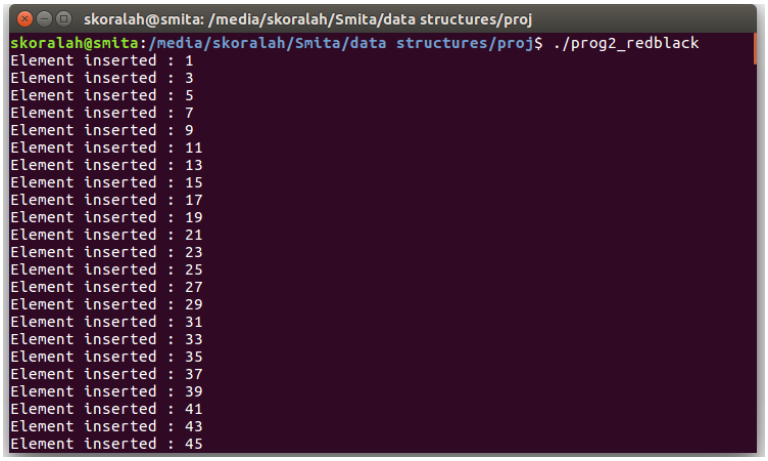
The height remained almost constant and with average number of key comparisons from 5 and 6.

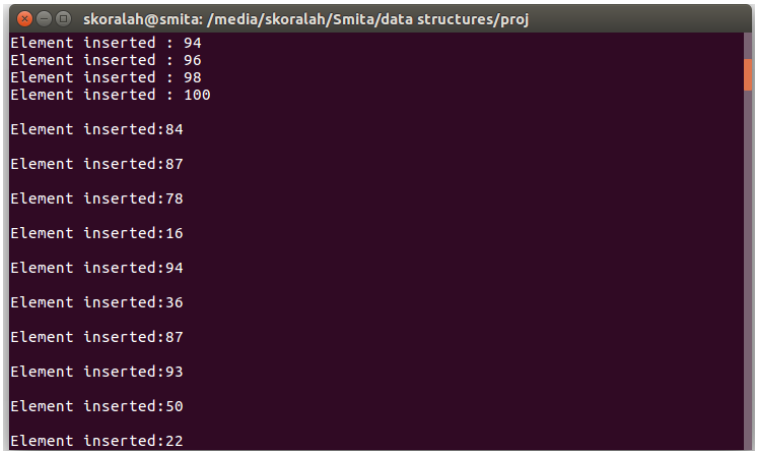
Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees

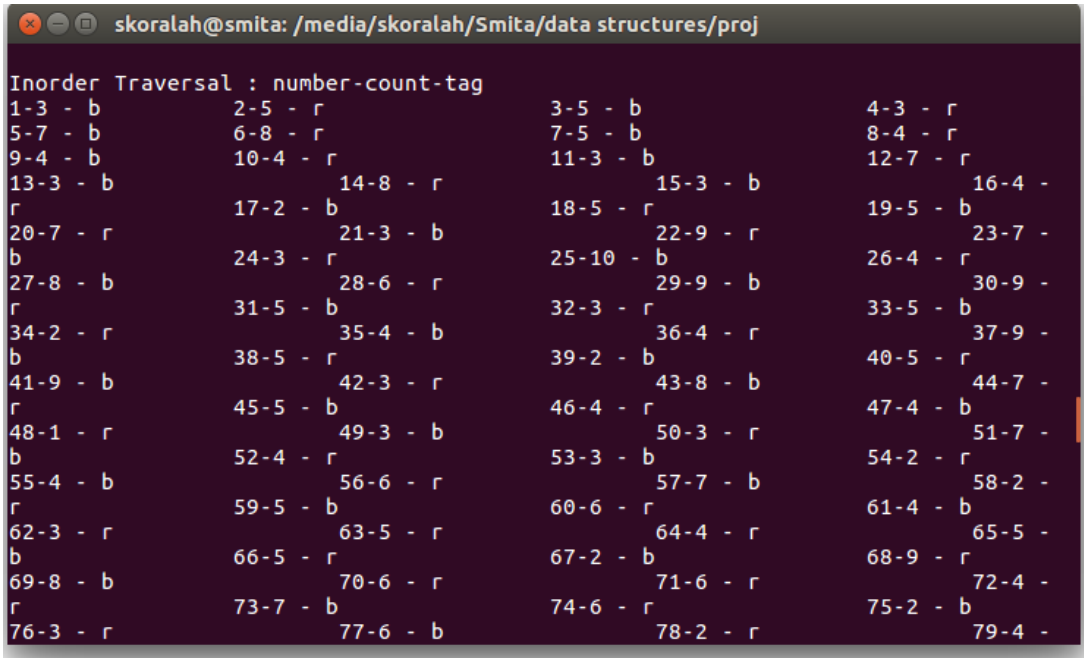
ITCS-6114/002 Data Structures and Algorithms

[2] 1, 3, 5,..., 99, 2, 4, 6,..., 100, followed by 400 random numbers between 1 & 100.





Elements inserted

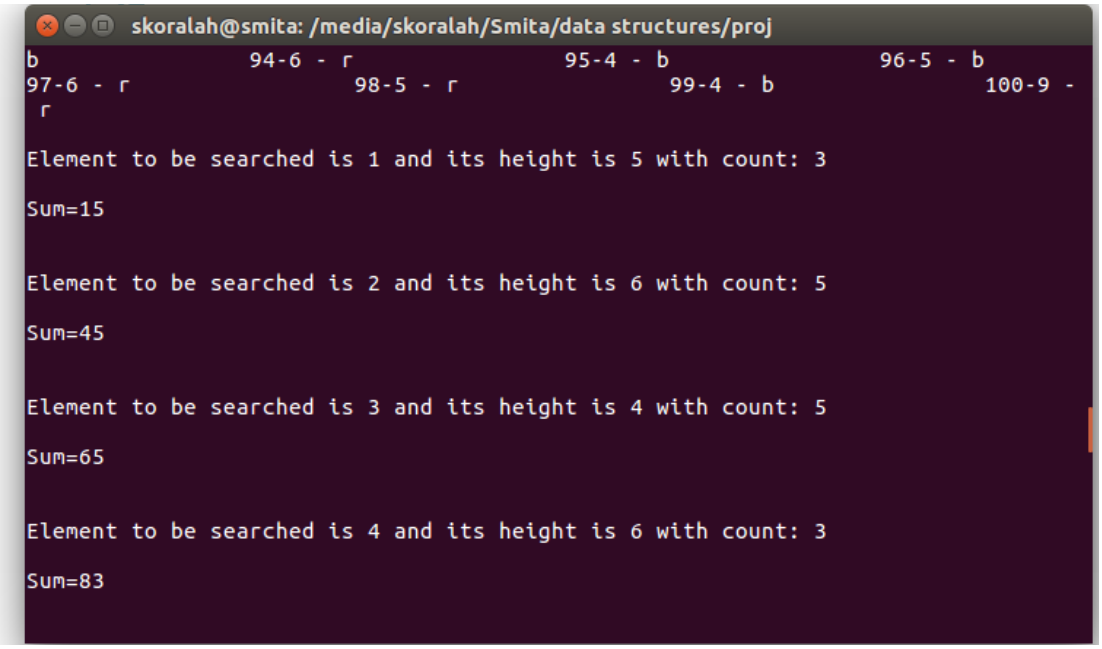


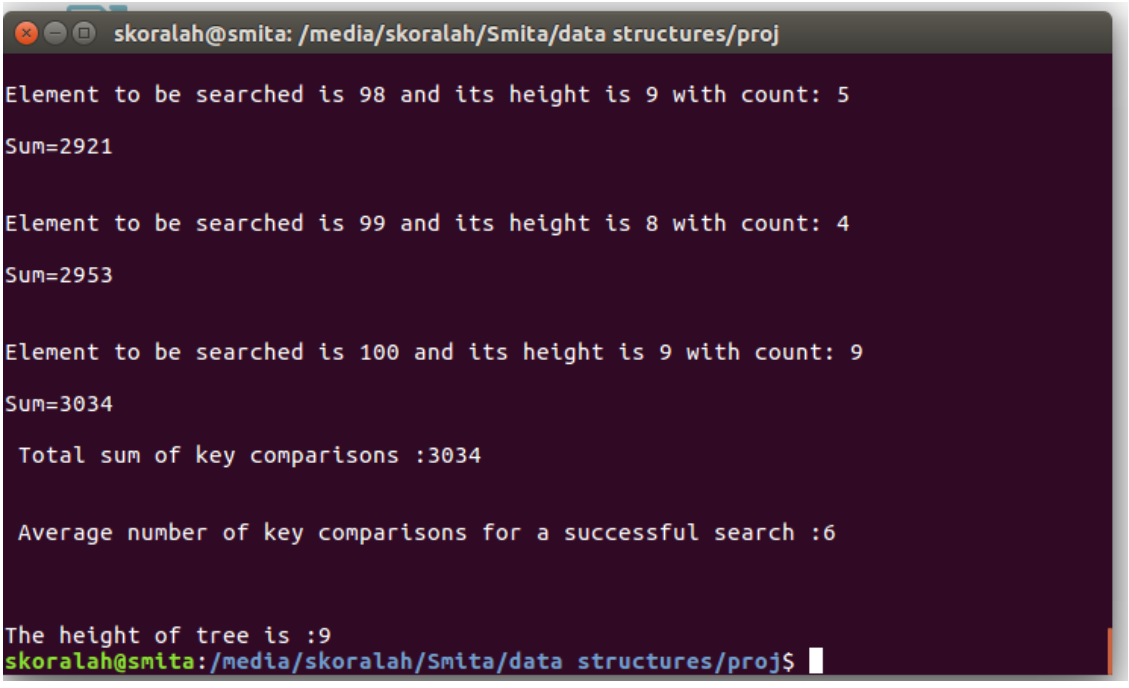
Number with its corresponding count and tag fields.

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees

ITCS-6114/002 Data Structures and Algorithms





The height observed was 9 throughout since even and odd numbers were inserted and average key comparisons ranged from 5 and 6.

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

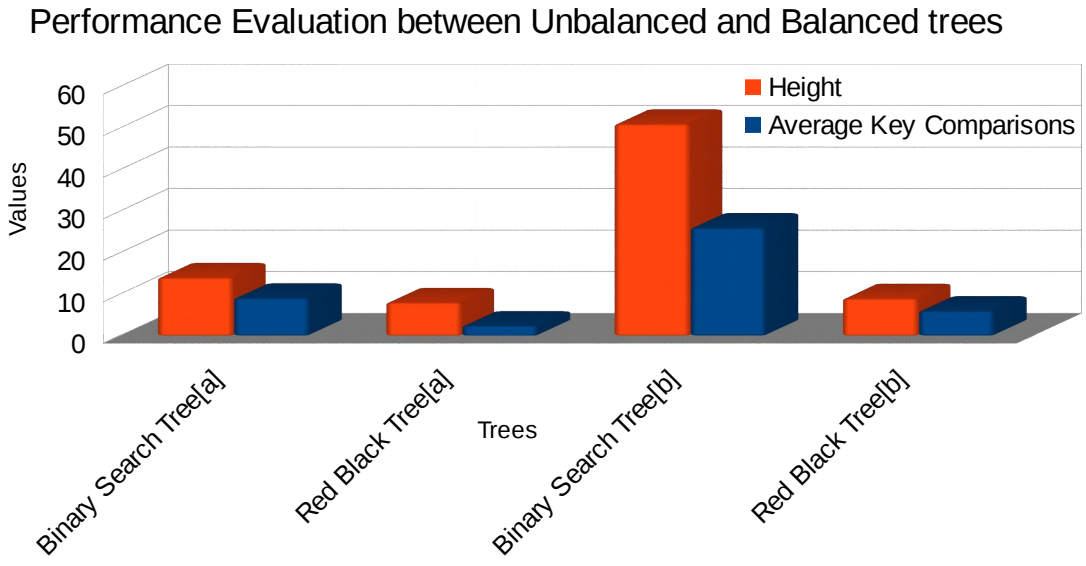
Project 1-Red Black Trees

ITCS-6114/002 Data Structures and Algorithms

Table of Comparison

Trees	Sequence of 500 random numbers	
	Height Observed	Average key comparisons
Binary Search Tree	14	7
Red Black Tree	8	5

Trees	Sequence of odd, even followed by 400 random numbers	
	Height Observed	Average key comparisons
Binary Search Tree	51	26
Red Black Tree	9	6



Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

Results Obtained

- As observed Red Black Tree performed better than binary search tree in both the cases. Additionally Red Black Tree with the sequence of **500 random numbers** with height 8 and average key comparisons of 5 **has the best performance** among all trees.

- The reason being Red Black Tree is a self balancing tree which makes the search to take logarithmic time whereas a binary search tree is able to form long chains of nodes that can cause searches to take linear time.

- That is as and when the new node/element is inserted in a Red Black Tree it fixes the newly inserted node, continuing up along the path to the root node and fixing nodes along that path with recoloring some nodes and performing rotation. As a result the tree becomes balanced after each insertion and is done in constant time.

- Thus the total running time both in the **worst case and best case** of the insertion process in **Red Black Tree** is **O(log N)** where N is the number of nodes in the tree with height 2log(N+1) whereas the **Binary Search Tree** has height of ceil(log(N+1) – 1) with **worst** case running time as **O(N)** when the elements are inserted in a reverse order. That is the reason the average key comparison and height **were substantially large** for binary search tree in the **second case** as at first the odd numbers are inserted till 99 and again even numbers are inserted all the way from beginning which caused traversing from 99 to 2 in reverse order. Thus the tree with 500 random numbers performed better than the second case.

Appendices
Source Code:

Binary Search Tree

[1] A sequence of 500 random numbers each between 1 & 100.

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

int match;                                  //key comparisons//
int sum=0;

typedef struct node {
  int data;
  struct node *left;
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
  struct node *right;
  int count;                                  //count for repeating values//
}node;


/*********Insertion Function******************/
struct node *insert ( struct node *tree, int key)
{

  if ( tree == NULL )
  {

        tree = malloc ( sizeof *tree );       //Initialize tree//
        if ( tree == NULL )
      return tree;

        tree->data = key;


        tree->left = tree->right = NULL;
        tree->count = 1;
  }

        else if (key == tree->data)           //Same element increment count//

      {
      (tree->count)++;
      return tree;
      }

      else if ( key < tree->data )            //Compare data inserted with key if less insert to
left//
        {
      tree->left = insert ( tree->left, key );

      }
        else
        {
      tree->right = insert ( tree->right, key);  //if greater insert to right//

        }

      return tree;
  }
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
/*********Search Function******************/
struct node *search(struct node* tree, int key)
{


   if(tree == NULL)
   {
        printf("No element found\n");
   }
   else if(key==tree->data)
   {


        printf("\n\nElement to be searched is %d and its height is %d with count: %d\n",tree->data,
match, tree->count);
      sum+=tree->count*match;                    // Calculation of total key comparisons
      printf("\nSum=%d\n",sum);
      return tree;
   }

   else if(key < tree->data)                    //Increment the comparisons as each element is compared//
   {
        match++;
      return search(tree->left,key);


   }
   else if(key > tree->data)
   {
        match++;
      return search(tree->right,key);


   }

   return tree;


}

/*********Height******************/
int bst_height(struct node *tree)
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
{
   int tmp1=0;
   int tmp2=0;

   if (tree)
   {

        tmp1 = bst_height(tree->left);                    // Maximum Depth//
      tmp2 = bst_height(tree->right);

        if (tmp1>tmp2)
        return tmp1+1;

      else
        return tmp2+1;

         }
}

/**********In-Order Traversal*******************/
void inorderTree(struct node *tree)                       //left, root, right//
{
        struct node* temp = tree;
        if (temp != NULL)
        {

                inorderTree(temp->left);

                printf("%d - %d\t\t ",temp->data, temp->count);

                inorderTree(temp->right);
        }
        return;
}


int main ( void )
{
  struct node *tree = NULL;
  int i;
  int b=1;

//function for different random numbers for each iteration; commented for same sequence for both
binary search tree and red black tree//
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
  //srand(time(NULL));
  for ( i = 0; i < 500; i++ )                    // For 500 numbers
  {
        int x=rand() % 100+1;                    //Random Function for generation of 100
numbers
        printf("Element inserted:%d\n",x);
        tree = insert ( tree, x);

  }
  printf("\nInorder Traversal : number - count\n");
  inorderTree(tree);

  while(b<=100)                                  //Search for all 100 numbers
  {
        match=1;
        search(tree,b);
        b++;

  }
  printf("\n Total sum of key comparisons :%d\n",sum);
  printf("\n\n Average number of key comparisons for a successful search :%d\n\n", sum/500);
  printf("\n\nThe height of tree is :%d\n",bst_height(tree));

}
```

[2] 1, 3, 5,..., 99, 2, 4, 6,..., 100, followed by 400 random numbers between 1 & 100.

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

int match;                                       //key comparisons//
int sum=0;

typedef struct node {
  int data;
  struct node *left;
  struct node *right;
  int count;                                     //count for repeating values//
}node;


/*********Insertion Function******************/
struct node *insert ( struct node *tree, int key)
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
{

 if ( tree == NULL )
 {

        tree = malloc ( sizeof *tree );                 //Initialize tree//
        if ( tree == NULL )
      return tree;

        tree->data = key;


        tree->left = tree->right = NULL;
        tree->count = 1;
 }

        else if (key == tree->data)                     //Same element increment count//

      {
      (tree->count)++;
      return tree;
      }

      else if ( key < tree->data )                      //Compare data inserted with key if less insert to
left//
        {
      tree->left = insert ( tree->left, key );

      }
        else
        {
      tree->right = insert ( tree->right, key);  //if greater insert to right//

        }

      return tree;
 }


/*********Search Function******************/
struct node *search(struct node* tree, int key)
{
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
    if(tree == NULL)
    {
        printf("No element found\n");
    }
    else if(key==tree->data)
    {


        printf("\n\nElement to be searched is %d and its height is %d with count: %d\n",tree->data,
match, tree->count);
       sum+=tree->count*match;                    // Calculation of total key comparisons
       printf("\nSum=%d\n",sum);
       return tree;
    }

    else if(key < tree->data)                    //Increment the comparisons as each element is compared//
    {
        match++;
       return search(tree->left,key);


    }
    else if(key > tree->data)
    {
        match++;
       return search(tree->right,key);


    }

    return tree;


}

/*********Height*******************/
int bst_height(struct node *tree)
{
    int tmp1=0;
    int tmp2=0;

    if (tree)
    {
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
        tmp1 = bst_height(tree->left);                  // Maximum Depth//
      tmp2 = bst_height(tree->right);

        if (tmp1>tmp2)
        return tmp1+1;

      else
        return tmp2+1;

         }
}

/*********In-Order Traversal*******************/
void inorderTree(struct node *tree)                     //left, root, right//
{
        struct node* temp = tree;
        if (temp != NULL)
        {

                inorderTree(temp->left);

                printf("%d - %d\t\t ",temp->data, temp->count);

                inorderTree(temp->right);
        }
        return;
}

int main ( void )
{
  struct node *tree = NULL;

  int i;
  int odd,even;
  int b=1;

//function for different random numbers for each iteration; commented for same sequence for both
binary search tree and red black tree//
  //srand(time(NULL));


/******** Odd number generation ***********/
        for(odd=1; odd<=99; odd+=2)
        {
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
        tree=insert(tree,odd);
        printf("Element inserted : %d\n",odd);
        }


/******** Even number generation **********/
        for(even=2; even<=100; even+=2)
        {

      tree=insert(tree,even);
      printf("Element inserted : %d\n",even);
        }



        for ( i = 0; i < 400; i++ )                    // For 400 numbers
        {
        int x=rand() % 100+1;                          //Random Function for generation of
100 numbers
        printf("Element inserted : %d\n",x);
      tree = insert ( tree, x);

        }


        printf("\nInorder Traversal : number - count\n");
        inorderTree(tree);

        while(b<=100)                                  //Search for all 100 numbers
        {
                match=1;
                search(tree,b);
                b++;

        }

  printf("\n Total sum of key comparisons :%d\n",sum);
  printf("\n\n Average number of key comparisons for a successful search :%d\n\n", sum/500);
  printf("The height of tree is :%d\n",bst_height(tree));



}
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

Red Black Tree

[1] A sequence of 500 random numbers each between 1 & 100.

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


int match;                                          //key comparisons//
int sum=0;

typedef struct node{
int key;
char color;                                         //count for repeating values//
struct node *left;
struct node *right;
struct node *parent;
int count;
}node;

struct node* root = NULL;

/**************Left Rotate*************/
void leftRotate(struct node *x)
{

        struct node *y;
        y = x->right;                               //Set y
        x->right = y->left;                         // Initialize y's left subtree into x's right
subtree

        if( y->left != NULL)
        {
                y->left->parent = x;                //Connect the y's left sublink
        }

        y->parent = x->parent;                              //Connect x's old parent and y's
parent

        if( x->parent == NULL)
        {
                root = y;
        }
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
        else if((x->parent->left!=NULL) && ( x->key == x->parent->left->key))
        {
                x->parent->left = y;                    //Connect x's old parent's left or right child
        }

        else x->parent->right = y;
        y->left = x;                                    //put x on y's left
        x->parent = y;                                          // x's parent

        return;
}


/**************Right Rotate************/
void rightRotate(struct node *y)
{
        struct node *x;
        x = y->left;                                    //set x
        y->left = x->right;                             //Turn x's right subtree into y's left subtree

        if ( x->right != NULL)
        {
                x->right->parent = y;
        }

        x->parent = y->parent;                                  //Connect y's old parent and x's
parent
        if( y->parent == NULL)
        {
                root = x;
        }

        else if((y->parent->left!=NULL) && ( y->key == y->parent->left->key))
        {
                y->parent->left = x;                    //Connect y's old parent's left or right child
        }

        else y->parent->right = x;
        x->right = y;                                   //put y on x's right
        y->parent = x;                                          //Take care of y's parent

        return;

}
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
/***********Perform operations to fix the node***************/
//If parent black exit//
//If parent red
//a.Red sibling: Change color//
//b.Black or absent sibling: Rotate and recolor//

void insertFix(struct node *z)
{
	struct node *y=NULL;
	while((z->parent!=NULL) && (z->parent->color == 'r'))
	{
		if ((z->parent->parent->left!=NULL) && (z->parent->key == z->parent->parent->left->key))
		{
			if(z->parent->parent->right!=NULL)
		y = z->parent->parent->right;
			if((y!=NULL) && (y->color == 'r'))
			{
				z->parent->color = 'b';
				y->color = 'b';
			z->parent->parent->color = 'r';
				if(z->parent->parent!=NULL)
				z = z->parent->parent;
		}

			else
			{
				if ((z->parent->right!=NULL) &&(z->key == z->parent->right->key))
				{
					z = z->parent;
					leftRotate(z);
			}

				z->parent->color = 'b';
			z->parent->parent->color = 'r';
			rightRotate(z->parent->parent);
			}
		}

		else
		{
		if(z->parent->parent->left!=NULL)
		y = z->parent->parent->left;
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
                if ((y!=NULL) && (y->color == 'r'))
                        {
                                z->parent->color = 'b';
                        y->color = 'b';
                                z->parent->parent->color = 'r';
                                if(z->parent->parent!=NULL)
                                z = z->parent->parent;
                }

                        else
                {
                                if ((z->parent->left!=NULL) && (z->key == z->parent->left->key))
                                {
                                        z = z->parent;
                                        rightRotate(z);

                                }
                        z->parent->color = 'b';
                        z->parent->parent->color = 'r';
                        leftRotate(z->parent->parent);
                        }
                }
          }

        root->color = 'b';
}




/***********Insert operation*********************/
void insert(int val)
{
        struct node *x, *y;
        struct node *z = (struct node*)malloc(sizeof(struct node));
        z->key = val;
        z->left = NULL;
        z->right = NULL;
        z->color = 'r';
        x = root;



        if ( root == NULL )                     // root insertion and setting color to black//
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees

ITCS-6114/002 Data Structures and Algorithms

```
        {

                root = z;
                root->count=0;
                root->color = 'b';

                return;
        }


      else if (root!=NULL)                    // same element found in root just increment count//
        {
        if(z->key==root->key){
                root->count++;
                return;}
        }




        while ( x != NULL)                    // subsequent node insertion and increment//
        {

                y = x;


           if ( z->key < x->key)
                {
                        x = x->left;
                }

                else if (z->key > x->key)
                {
                        x = x->right;
                }

                else
                {
                        x->count++;
                        return;
                }

        }
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
        z->parent = y;

        if ( y == NULL)
        {
                root = z;

        }

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

        else if (z->key > y->key)
        {
                y->right = z;
        }

        else
        {
                y->count++;
                return;
        }




        insertFix(z);



        return;
}



/***********Search operation*********************/

int search(int val)
{
        struct node* temp=root;
        int diff;
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
        while(temp!=NULL)
        {
                diff=val - temp->key;                   //calculate diff with key and value and traverse//

                if(diff>0)
                {
                        temp=temp->right;
                        match++;
                }

                else if (diff<0)
                {
                        temp=temp->left;
                        match++;
                }

                else
                {

                        printf("\n\nElement to be searched is %d and its height is %d with count: %d\n",temp->key, match, temp->count+1);
                sum+=(temp->count+1)*match;
                printf("\nSum=%d\n",sum);
                return 1;
                }
                }

                return 0;
}


/*********Height*******************/
int rbt_height(struct node *root)
{
   int tmp1=0;
   int tmp2=0;
   if (root)
   {
      tmp1 = rbt_height(root->left);
      tmp2 = rbt_height(root->right);
      if (tmp1>tmp2)
         return tmp1+1;
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
        else
            return tmp2+1;


    }
}

/**********In-Order Traversal*******************/
void inorderTree(struct node* root){
struct node* temp = root;

if (temp != NULL){
    inorderTree(temp->left);
    printf("%d-%d - %c\t\t ",temp->key, temp->count+1, temp->color);
    inorderTree(temp->right);
}
return;
}


int main ( void )
{
  struct node *tree = NULL;
  int i;
  int b=1;

//function for different random numbers for each iteration; commented for same sequence for both binary search tree and red black tree//
  //srand(time(NULL));
  for ( i = 0; i < 500; i++ )
  {
          int x=rand() % 100+1;
          printf("\nElement inserted:%d\n",x);
          insert (x);

  }
     printf("\nInorder Traversal : number-count-tag\n");
     inorderTree(root);

  while(b<=100)
  {
          match=1;
          search(b);
          b++;
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
  }

  printf("\n Total sum of key comparisons :%d\n",sum);
  printf("\n\n Average number of key comparisons for a successful search :%d\n\n", sum/500);
  printf("\n\nThe height of tree is :%d\n",rbt_height(root));

}
```

[2] 1, 3, 5,..., 99, 2, 4, 6,..., 100, followed by 400 random numbers between 1 & 100.

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


int match;                                          //key comparisons//
int sum=0;

typedef struct node{
int key;
char color;                                         //count for repeating values//
struct node *left;
struct node *right;
struct node *parent;
int count;
}node;

struct node* root = NULL;

/**************Left Rotate************/
void leftRotate(struct node *x)
{

        struct node *y;
        y = x->right;                               //Set y
        x->right = y->left;                         // Initialize y's left subtree into x's right
subtree

        if( y->left != NULL)
        {
                y->left->parent = x;                //Connect the y's left sublink
        }
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
	y->parent = x->parent;				//Connect x's old parent and y's
parent

	if( x->parent == NULL)
	{
		root = y;
	}
	else if((x->parent->left!=NULL) && ( x->key == x->parent->left->key))
	{
		x->parent->left = y;			//Connect x's old parent's left or right child
	}

	else x->parent->right = y;
	y->left = x;				//put x on y's left
	x->parent = y;					// x's parent

	return;
}


/**************Right Rotate************/
void rightRotate(struct node *y)
{
	struct node *x;
	x = y->left;				//set x
	y->left = x->right;			//Turn x's right subtree into y's left subtree

	if ( x->right != NULL)
	{
		x->right->parent = y;
	}

	x->parent = y->parent;				//Connect y's old parent and x's
parent
	if( y->parent == NULL)
	{
		root = x;
	}

	else if((y->parent->left!=NULL) && ( y->key == y->parent->left->key))
	{
		y->parent->left = x;			//Connect y's old parent's left or right child
	}
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
        else y->parent->right = x;
        x->right = y;                                   //put y on x's right
        y->parent = x;                                          //Take care of y's parent

        return;

}

/***********Perform operations to fix the node***************/
//If parent black exit//
//If parent red
//a.Red sibling: Change color//
//b.Black or absent sibling: Rotate and recolor//

void insertFix(struct node *z)
{
        struct node *y=NULL;
        while((z->parent!=NULL) && (z->parent->color == 'r'))
        {
                if ((z->parent->parent->left!=NULL) && (z->parent->key == z->parent->parent->left-
>key))
                {
                        if(z->parent->parent->right!=NULL)
                y = z->parent->parent->right;
                        if((y!=NULL) && (y->color == 'r'))
                        {
                                z->parent->color = 'b';
                                y->color = 'b';
                        z->parent->parent->color = 'r';
                                if(z->parent->parent!=NULL)
                                z = z->parent->parent;
                }

                        else
                        {
                                if ((z->parent->right!=NULL) &&(z->key == z->parent->right->key))
                                {
                                        z = z->parent;
                                        leftRotate(z);
                        }

                                z->parent->color = 'b';
                        z->parent->parent->color = 'r';
                        rightRotate(z->parent->parent);
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
                }
        }

        else
        {
        if(z->parent->parent->left!=NULL)
        y = z->parent->parent->left;
        if ((y!=NULL) && (y->color == 'r'))
                {
                        z->parent->color = 'b';
                y->color = 'b';
                        z->parent->parent->color = 'r';
                        if(z->parent->parent!=NULL)
                        z = z->parent->parent;
        }

                else
        {
                        if ((z->parent->left!=NULL) && (z->key == z->parent->left->key))
                        {
                                z = z->parent;
                                rightRotate(z);

                        }
                z->parent->color = 'b';
                z->parent->parent->color = 'r';
                leftRotate(z->parent->parent);
                }
        }
    }

  root->color = 'b';
}




/***********Insert operation*********************/
void insert(int val)
{
    struct node *x, *y;
    struct node *z = (struct node*)malloc(sizeof(struct node));
    z->key = val;
    z->left = NULL;
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees

ITCS-6114/002 Data Structures and Algorithms

```
z->right = NULL;
z->color = 'r';
x = root;



if ( root == NULL )                    // root insertion and setting color to black//
{

        root = z;
        root->count=0;
        root->color = 'b';

        return;
}


    else if (root!=NULL)                    // same element found in root just increment
count//
        {
        if(z->key==root->key){
                root->count++;
                return;}
        }




        while ( x != NULL)                    // subsequent node insertion and increment//
        {

                y = x;


           if ( z->key < x->key)
                {
                        x = x->left;
                }

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

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
                else
                {
                        x->count++;
                        return;
                }

        }

        z->parent = y;

        if ( y == NULL)
        {
                root = z;

        }

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

        else if (z->key > y->key)
        {
                y->right = z;
        }

        else
        {
                y->count++;
                return;
        }



        insertFix(z);



        return;
}
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
/**********Search operation*********************/

int search(int val)
{
        struct node* temp=root;
        int diff;

        while(temp!=NULL)
        {
                diff=val - temp->key;                   //calculate diff with key and value and
traverse//

                if(diff>0)
                {
                        temp=temp->right;
                        match++;
                }

                else if (diff<0)
                {
                        temp=temp->left;
                        match++;
                }

                else
                {

                        printf("\n\nElement to be searched is %d and its height is %d with count:
%d\n",temp->key, match, temp->count+1);
                sum+=(temp->count+1)*match;
                printf("\nSum=%d\n",sum);
                return 1;
                }
                }

                return 0;
}


/*********Height*******************/
int rbt_height(struct node *root)
{
   int tmp1=0;
   int tmp2=0;
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
    if (root)
    {
       tmp1 = rbt_height(root->left);
       tmp2 = rbt_height(root->right);
       if (tmp1>tmp2)
          return tmp1+1;

       else
          return tmp2+1;


    }
}

/**********In-Order Traversal*******************/
void inorderTree(struct node* root){
struct node* temp = root;

if (temp != NULL){
    inorderTree(temp->left);
    printf("%d-%d - %c\t\t ",temp->key, temp->count+1, temp->color);
    inorderTree(temp->right);
}
return;
}


int main ( void )
{
  struct node *tree = NULL;
  int i;
  int odd, even;
  int b=1;

//function for different random numbers for each iteration; commented for same sequence for both
binary search tree and red black tree//
  //srand(time(NULL));


/********* Odd number generation ***********/
          for(odd=1; odd<=99; odd+=2)
          {
          insert(odd);
          printf("Element inserted : %d\n",odd);
```

Smita Koralahalli Channabasappa skoralah@uncc.edu
Student ID: 801008161

Project 1-Red Black Trees
ITCS-6114/002 Data Structures and Algorithms

```
        }


/******** Even number generation ***********/
        for(even=2; even<=100; even+=2)
        {

        insert(even);
        printf("Element inserted : %d\n",even);
        }


        for ( i = 0; i < 400; i++ )
        {
                int x=rand() % 100+1;
                printf("\nElement inserted:%d\n",x);
                insert (x);

        }
      printf("\nInorder Traversal : number-count-tag\n");
      inorderTree(root);

        while(b<=100)
        {
                match=1;
                search(b);
                b++;

        }

  printf("\n Total sum of key comparisons :%d\n",sum);
  printf("\n\n Average number of key comparisons for a successful search :%d\n\n", sum/500);
  printf("\n\nThe height of tree is :%d\n",rbt_height(root));

}
```