CS590 homework 4 – Binary-Search, and Red-black Trees

The due date for this assignment is Friday, Nov 2nd, at 11.59pm. This assignment is worth 10% of your final grade.

Any sign of collaboration will result in a 0 and being reported to the Graduate Academic Integrity Board. Late submission policy described in the syllabus will be applied.

(100 points)

We can use binary-search and red-black trees in order to sort an array of n integers by inserting them into an empty tree, and using a modified INORDER-TREE-WALK algorithm to copy the elements back to the array sorted.

1. You are given an implementation of red-black trees. Implement a binary-search tree with the corresponding functionality. You can omit the delete functionality for binary-search and red-black trees, but you have to update the insertion routine of the binary-search and red-black tree to handle duplicate values. The insertion functions do not insert a value if the value is already in the tree.

2. Modify the INORDER-TREE-WALK algorithm for binary-search and red-black trees such that it traverses the tree in order to copy its elements back to an array, in a sorted ascending order. The number of elements in the tree might be less than n due to the elimination of key duplicates. The function should therefore return the number n' of elements that were copied into the array (number of tree elements).
Notes: The algorithm relies only on the binary-search tree properties which also red-black trees satisfy. Keep in mind that only the first n' elements of your array are afterwards sorted.

3. Implement an insertion function that has an array of integers and the array length as arguments. Modify your insertion routine for binary-search and red-black trees such that it counts the following occurrences over the sequence of insertions.

- Counter for the number of duplicates.
- Counter for each of the insertion cases (case 1, case 2, and case 3) (red-black tree only).
- Counter for left rotate and for right rotate (red-black tree only).

You should have 1 counter for binary-search trees and 6 counters for red-black trees altogether.

4. Develop a test function for red-black trees such that, given a node of the red-black tree, traverses to each of the accessible leaves and counts the number of black nodes on the path to the leave.
Notes: You could use your test function to verify whether or not your red-black tree implementation satisfies red-black property 5.

5. Measure the runtime performance of your "Binary-Search Tree Sort" and "Red-Black Tree Sort" for random, sorted, and inverse sorted inputs of size n = 50000; 100000; 250000; 500000; 1000000; 2500000; 5000000. You can use the provided functions *create random, create sorted, create reverse sorted*.
Repeat each test a number of times (usually at least 10 times) and compute the average running time and the average counter values for each combination of input and size n. Report and comment on your results. How do the counters behave and how is the height of the respective trees in comparison to how the running time behaves.

(You might have to adjust the value for n dependent on your computers speed, but allow each test to take up to a couple of minutes. Start with smaller values of n and stop if one instance of the algorithm takes more than 10 min to complete).

Remarks:

- You are not allowed to use code from online resources. Your submission will be tested against that, and will receive a 0, and a report to the Graduate Academic Integrity Board if it is detected.
- No additional libraries are allowed to be used
- Makefiles are provided for both problems to build the code in LinuxLab.
- The programming, and testing will take some time. Start early.
- Feel free to use the provided source code for your implementation. You have to document your code.

Stevens Institute of Technology

Department of Computer Science

CS590: ALGORITHMS

Homework Assignment 4

Submitted by:
Hadia Hameed
CWID: 10440803

Yijia Tan
CWID: 10427079

Submitted to:
Prof. Iraklis
Tsekourakis

December 6, 2018

1 Problem I

You are given an implementation of red-black trees. Implement a binary-search tree with the corresponding functionality. You can omit the delete functionality for binary-search and red-black trees, but you have to update the insertion routine of the binary-search and red-black tree to handle duplicate values. The insertion functions do not insert a value if the value is already in the tree.

1.1 Changing the insert function:

In order to ensure there are no duplicates in the binary search tree, the *insert(bs_tree_node* z)* function is modified in the following way:

```
void bs_tree::insert(bs_tree_node* z)
{
    bs_tree_node* x;
    bs_tree_node* y;

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


1.2 Inserting values in the binary search tree:

```
int main(int argc, char* argv[])
{
  bs_tree BST;
  int input_array [10] = {10,10,12,3,4,5,3,6,5,100};
  BST.insert(input_array,10);
  BST.inorder_output();
  return 0;
}
```

Output:

```
(3,2)
(4,3)
(5,4)
(6,5)
(10,1)
(12,2)
(100,3)
```

2 Problem II

Modify the INORDER-TREE-WALK algorithm for binary-search and red-black trees such that it traverses the tree in order to copy its elements back to an array, in a sorted ascending order. The number of elements in the tree might be less than n due to the elimination of key duplicates. The function should therefore return the number n' of elements that were copied into the array (number of tree elements). Notes: The algorithm relies only on the binary-search tree properties which also red-black trees satisfy. Keep in mind that only the first n' elements of your array are afterwards sorted.

2.1 Adding two protected variables in bs_tree.h and rb_tree.h

2.1.1 bs_tree.h

```
class bs_tree
{
    protected:
        bs_tree_node*   T_root;
    public:
        bs_tree();
        ~bs_tree();
        void insert(int*, int);
```



```
        void insert(int);
        int* inorder_output(int);
         void output()
               { output(T_root, 1); }

    protected:
        void insert(bs_tree_node*);
        void remove_all(bs_tree_node*);
        void inorder_output(bs_tree_node*, int);
        int* array;
        int array_index;
};
```

2.2 Editing inorder_output functions in .cpp files:

```
bs_tree::bs_tree() {
  T_root = new bs_tree_node();
  T_root = nullptr;
  array = new int[1];
}

bs_tree::~bs_tree(){
  remove_all(T_root);
  delete[] array;
}

int* bs_tree::inorder_output(int n) {
   if(array) delete[] array;
   array = new int[n];
   array_index = 0;
   inorder_output(T_root, 1);
   return array;
}

void bs_tree::inorder_output(bs_tree_node* x, int
   level)
{
  if (x != nullptr)
  {
    inorder_output(x->left, level+1);
    cout << "(" << x->key << "," << level << ")" <<
       endl;
    array[array_index] = x->key;
    array_index = array_index + 1;
    inorder_output(x->right, level+1);
  }
```



```
}
```

3 Problem III:

Implement an insertion function that has an array of integers and the array length as arguments. Modify your insertion routine for binary-search and red-black trees such that it counts the following occurrences over the sequence of insertions.

- **Counter for the number of duplicates.**
- **Counter for each of the insertion cases (case 1, case 2, and case 3) (red-black tree only).**
- **Counter for left rotate and for right rotate (red-black tree only).**

You should have 1 counter for binary-search trees and 6 counters for red-black trees altogether.

3.1 Editing the header files:

3.1.1 bs_tree.h

```
class bs_tree
{
    protected:
        bs_tree_node*   T_root;
    public:
        bs_tree();
        ~bs_tree();
        void insert(int*, int);
        void insert(int);
        int* inorder_output(int);
        int get_counter_duplicate() {
              return counter_duplicate;
        }
    protected:
        void insert(bs_tree_node*);
        void remove_all(bs_tree_node*);
        void inorder_output(bs_tree_node*, int);
        int* array;
        int array_index;
        int counter_duplicate;
};
```

3.1.2 rb_tree.h


```
class rb_tree
{
    protected:
        rb_tree_node*   T_nil;
        rb_tree_node*   T_root;
    public:
        rb_tree();
        ~rb_tree();
        void insert(int*, int);
        void insert(int);
        int* inorder_output(int);
        void output()
              { output(T_root, 1); }
        int get_counter_duplicate()
              { return counter_duplicate; }
        int get_counter_right_rotate()
              { return counter_right_rotate; }
        int get_counter_leftt_rotate()
              { return counter_left_rotate; }
        int get_counter_case1()
              { return counter_case1; }
        int get_counter_case2()
              { return counter_case2; }
        int get_counter_case3()
              { return counter_case3; }
    protected:
        void insert(rb_tree_node*);
        void remove_all(rb_tree_node*);
        void inorder_output(rb_tree_node*, int);
        int* array;
        int array_index;
        int counter_duplicate;
        int counter_right_rotate;
        int counter_left_rotate;
        int counter_case1;
        int counter_case2;
        int counter_case3;
};
```

3.2 Editing the cpp files:

3.2.1 bs_tree.cpp

```
bs_tree::bs_tree()
{
  T_root = new bs_tree_node();
  T_root = nullptr;
```



```
  counter_duplicate = 0;
}

void bs_tree::insert(bs_tree_node* z)
{
  bs_tree_node* x;
  bs_tree_node* y;
  y = nullptr;
  x = T_root;
  while (x != nullptr)
  {
     y = x;
     if (z->key == x->key){
        counter_duplicate = counter_duplicate + 1;
        return;
     }
     if (z->key < x->key)
         x = x->left;
     else
         x = x->right;
 }
 ...
```

3.2.2 rb_tree.cpp

```
rb_tree::rb_tree()
{
  T_nil = new rb_tree_node();
  T_nil->color = RB_BLACK;
  T_nil->p = T_nil;
  T_nil->left = T_nil;
  T_nil->right = T_nil;
  counter_right_rotate = 0;
  counter_left_rotate = 0;
  counter_duplicate = 0;
  counter_case1 = 0;
  counter_case2 = 0;
  counter_case3 = 0;
  T_root = T_nil;
}

void rb_tree::insert(rb_tree_node* z)
{
  rb_tree_node* x;
  rb_tree_node* y;
  y = T_nil;
```



```
  x = T_root;
  while (x != T_nil)
  {
     y = x;
     if (z->key == x->key){
        counter_duplicate = counter_duplicate + 1;
        return;
     }
     if (z->key < x->key)
         x = x->left;
     else
         x = x->right;
 }
 ...

 void rb_tree::insert_fixup(rb_tree_node*& z)
{
   rb_tree_node* y;
   while (z->p->color == RB_RED)
   {
      if (z->p == z->p->p->left)
      {
         y = z->p->p->right;
         if (y->color == RB_RED)
         {
            counter_case1 = counter_case1 + 1;
            z->p->color = RB_BLACK; // Case 1
            y->color = RB_BLACK;
            z->p->p->color = RB_RED;
            z = z->p->p;
         }
         else
         {
           if (z == z->p->right)
           {
             counter_case2 = counter_case2 + 1;
              z = z->p; // Case 2
              left_rotate(z);
              counter_left_rotate =
                 counter_left_rotate + 1;
           }
           counter_case3 = counter_case3 + 1;
           z->p->color = RB_BLACK;  // Case 3
           z->p->p->color = RB_RED;
           right_rotate(z->p->p);
           counter_right_rotate = counter_right_rotate
```



```
                + 1;
            }
        }
        else
        {
            y = z->p->p->left;
            if (y->color == RB_RED)
            {
              counter_case1 = counter_case1 + 1;
              z->p->color = RB_BLACK;  // Case 1
              y->color = RB_BLACK;
              z->p->p->color = RB_RED;
              z = z->p->p;
            }
            else
            {
                if (z == z->p->left)
                {
                    counter_case2 = counter_case2 + 1;
                    z = z->p;                   // Case 2
                    right_rotate(z);
                    counter_right_rotate =
                        counter_right_rotate + 1;
                }
                counter_case3 = counter_case3 + 1;
                z->p->color = RB_BLACK; // Case 3
                z->p->p->color = RB_RED;
                left_rotate(z->p->p);
                counter_left_rotate = counter_left_rotate +
                    1;
            }
        }
    }
    T_root->color = RB_BLACK;
}
```

4 Problem IV

Develop a test function for red-black trees such that, given a node of the red-black tree, traverses to each of the accessible leaves and counts the number of black nodes on the path to the leave.

4.1 Defining test function and member variable in rb_tree.h:

```
class rb_tree
{
    protected:
        rb_tree_node* T_nil; rb_tree_node* T_root;
    public:
        rb_tree();
        ~rb_tree();
        void insert(int*, int);
        void insert(int);
        int* inorder_output(int);
        void output()
              { output(T_root, 1); }
        int get_counter_duplicate()
              { return counter_duplicate; }
        int get_counter_right_rotate()
              { return counter_right_rotate; }
        int get_counter_leftt_rotate()
              { return counter_left_rotate; }
        int get_counter_case1()
              { return counter_case1; }
        int get_counter_case2()
              { return counter_case2; }
        int get_counter_case3()
              { return counter_case3; }
        int rb_tree_test_function(){
              int black_height = rb_tree_height(T_root
                 ->right);
              return ceil(black_height/2);
         };
    protected:
        void insert(rb_tree_node*);
        void remove_all(rb_tree_node*);
        void inorder_output(rb_tree_node*, int);
        int* array;
        int array_index;
        int counter_duplicate;
        int counter_right_rotate;
        int counter_left_rotate;
        int counter_case1;
        int counter_case2;
        int counter_case3;
        int rb_tree_height(rb_tree_node*);
};
```


4.2 Adding implementation of test function in rb_tree.cpp file:

Using function overriding to create two functions with the same name but different implementation and input arguments.

```
int rb_tree::rb_tree_height(rb_tree_node* x){
   if (x == T_nil) {
      return 0;
   }
   else {
     return max(rb_tree_height(x->left),rb_tree_height
        (x->right))+1;
  }
}

int max(int a, int b){
  if (a>=b){
    return a;
  }
  else {
    return b;
 }
}
```

4.3 Test case:

4.3.1 Finding black height of the root:

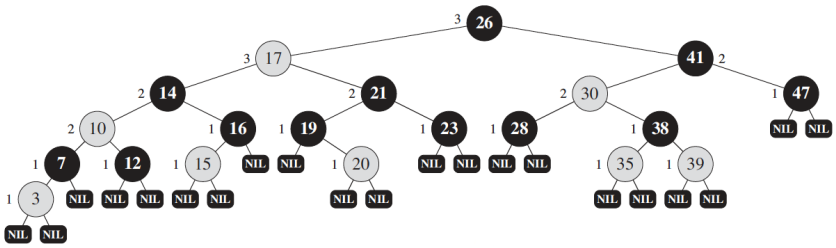


Figure 1: An example of red balck tree from *Cormen, Thomas H., et al. Introduction to algorithms. MIT press, 2009*

```
int input_array [20] =
   {26,17,41,14,21,30,47,10,16,19,23,28,38,7,12,15,20,35,39,3};
```



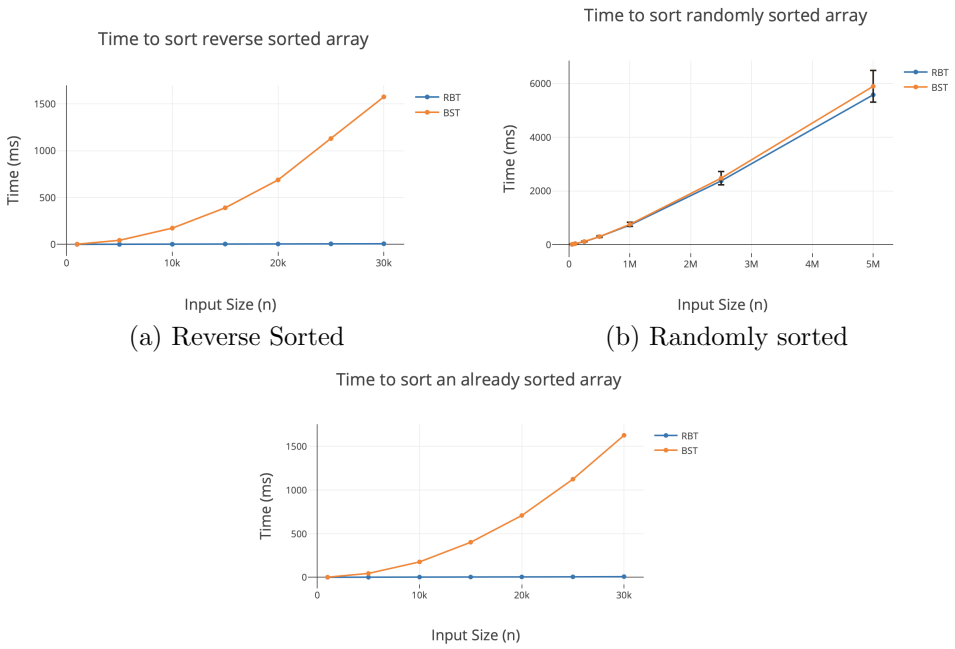
```
rb_tree* RBT = new rb_tree();
RBT->insert(input_array,20);
int bh = RBT->rb_tree_test_function();
cout<<``Black Height of the root is: ''<<bh<<endl;
```

4.3.2 Output:

```
Printing Sorted array:
3 , 7 , 10 , 12 , 14 , 15 , 16 , 17 , 19 , 20 , 21 , 23 , 26 , 28 , 30 , 35 , 38
 , 39 , 41 , 47

Testing Property 5: Black Height
---------------------------------
Black Height of the root is: 3
```

5 Problem V



(a) Reverse Sorted

(b) Randomly sorted

(c) Sorted

Figure 2: Time performance for Binary Search Trees (BST) and Red Black Trees (RBT) for each combination of input and value of n

	Time (ms)	
	Direction = 0	
	BST	RBT
50,000	18	14
100,000	40	44
250,000	121	114
500,000	303	301
1,000,000	760	733
2,500,000	2476	2374
5,000,000	5891	5573

Figure 3: Sort time using Binary search trees and Red Black trees for randomly sorted arrays.

	Time (ms)			
	Direction = -1		Direction = 1	
	BST	RBT	BST	RBT
1,000	1	0	1	0
5,000	43	0	44	0
10,000	173	1	176	2
15,000	390	3	401	3
20,000	688	4	708	4
25,000	1130	5	1124	5
30,000	1575	6	1627	7

Figure 4: Sort time using Binary search trees and Red Black trees for reverse sorted and sorted arrays.

	Average values of counters for Direction = -1						
	counter_duplicates_BST	counter_duplicates_RBT	counter_case 1	counter_case 2	counter_case 3	counter_left_rotate	counter_right_rotate
1,000	0	0	978	0	983	0	983
5,000	0	0	4973	0	4978	0	4978
10,000	0	0	9971	0	9976	0	9976
15,000	0	0	14969	0	14975	0	14975
20,000	0	0	19969	0	19974	0	19974
25,000	0	0	24968	0	24973	0	24973
30,000	0	0	29967	0	29973	0	29973

(a) Reverse Sorted

	Average values of counters for Direction = 1						
	counter_duplicates_BST	counter_duplicates_RBT	counter_case 1	counter_case 2	counter_case 3	counter_left_rotate	counter_right_rotate
1,000	0	0	978	0	983	983	0
5,000	0	0	4973	0	4978	4978	0
10,000	0	0	9971	0	9976	9976	0
15,000	0	0	14969	0	14975	14975	0
20,000	0	0	19969	0	19974	19974	0
25,000	0	0	24968	0	24973	24973	0
30,000	0	0	29967	0	29973	29973	0

(b) Sorted

	Average values of counters for Direction = 0						
	counter_duplicates_BST	counter_duplicates_RBT	counter_case 1	counter_case 2	counter_case 3	counter_left_rotate	counter_right_rotate
50,000	1	1	25680	9703	19388	14567	14524
100,000	1	1	51309	19509	38903	29259	29153
250,000	18	10	128166	48353	97182	72734	72801
500,000	52	52	256882	97381	194646	145873	146154
1,000,000	223	239	513805	193900	388250	291072	291078
2,500,000	1498	1453	1282427	484257	969396	726482	727171
5,000,000	5944	5767	2564083	969900	1939640	1454312	1455228

(c) Randomly created array

Figure 5: Average values of counters for Binary Search Trees (BST) and Red Black Trees (RBT) for each combination of input and value of n

6 Conclusion:

The results show that for randomly sorted arrays, BST and RBT show roughly the same performance since a randomly created BST with n nodes, has height *O(lgn)*, the same as a RBT.

However, reverse sorted and sorted arrays are the worst-case scenarios for BST since the tree is unbalanced and IN-ORDER-TREE-WALK becomes in *O(n)* where n are the number of nodes in the BST. The sorting time complexity for the corresponding RBT would remain *O(lgn)* since RBT is balanced. This is shown in Fig. 4 where the sort time for BST is rising exponentially as the input size is increased for sorted and reverse sorted arrays, whereas there is little change in the sort time for RBT.

Fig 5. shows the average values of the counters for different combinations of input and size n. There are no instances of case II and left rotation for RBT when the array is reverse sorted. This is because since the array is already reverse sorted, there are more chances for case 1 or 3 to take place as these involve the left subtree. Similarly, when the array is sorted, there are no instances of case II and right rotation.

Also, intuitively, there are more chances to have duplicates when the array is created randomly as compared to when the array is already sorted or reverse sorted as shown in Fig 5.

