实验 3 : RedBlackTree && IntervalTree

PB16111485 张劲暾

1. 实验要求

实验 1 : 实现红黑树的基本算法, 分别对整数 n = 20 、 40 、 60 、 80 、 100 , 随机生成 n 个互异的正整数(K_1 , K_2 , K_3 , ..., K_n) , 以这 n 个正整数作为节点的关键字,向一棵初始空的红黑树中依次插入 n 个节点, 然后随机选择其中 n/10 个节点进行删除,统计插入和删除算法运行所需时间, 画出时间曲线。

实验 2 : 实现区间树的基本算法,随机生成 30 个正整数区间,以这 30 个正整数区间的左端点作为关键字构建红黑树,向一棵初始空的红黑树中依次插入 30 个节点,然后随机选择其中 3 个区间进行删除。实现区间树的插入、删除、遍历和查找算法。

2. 实验环境

编译环境: gcc (Ubuntu 7.2.0-8ubuntu3.2) 7.2.0

机器内存: 7.7 GiB

时钟主频: Intel Core™ i7-6500U CPU @ 2.50GHz × 4

3. 实验过程

(1) 实验 1:

a. 编写简单的 Python 程序生成所需的随机整数序列:

```python
import numpy as np
output = open("./PB16111485-project3/inputA/input_integer.txt",'w')
input_integer = []
while len(input_integer) < 100 :
    tmp = np.random.randint(0,65535)
    if tmp not in input_integer:
        input_integer.extend([tmp])
for integer in input_integer :
    output.write(str(integer) + "\n")

output.close()
```

b. 编写 C 语言函数实现对于红黑树相关数据结构定义和左旋、右旋、插入、插入调整、删除、删除调整和遍历函数，详见第 4 部分。

c. 在 main 函数中对不同规模的数据进行逐个插入建树和随机删除测试，并记录相关用时，一个典型的测试单元如下:

```c
/********************* size80 ************************/
InOrderOut = fopen("../outputA/size80/inorder.txt","w");
PreOrderOut = fopen("../outputA/size80/preorder.txt","w");
PastOrderOut = fopen("../outputA/size80/postorder.txt","w");
time1out = fopen("../outputA/size80/time1.txt","w");
deleteout = fopen("../outputA/size80/delete_data.txt","w");
time2out = fopen("../outputA/size80/time2.txt","w");
```

逐个插入建树

```c
sumt = 0;
for(i = 0;i < 8;i++){
    long ten_cost = 0;
    for(j = 0;j < 10;j++){
        RedBlackNode* in = malloc(sizeof(RedBlackNode));
        in->key = keys[i * 10 + j];
        gettimeofday(&start,NULL);
        RB_Insert(T,in);
        gettimeofday(&end,NULL);
        ten_cost += (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec);
    }
    fprintf(time1out,"%d ~ %d time cost: %ld us\n",i * 10 + 1, (i + 1) * 10,ten_cost);
    sumt += ten_cost;
}
fprintf(time1out,"size = 80, sumtime cost: %ld us\n",sumt);
```

前、中、后序遍历

```c
RB_InOrderTraverse(T,T->root,InOrderOut);
RB_ProOrderTraverse(T,T->root,PreOrderOut);
RB_PastOrderTraverse(T,T->root,PastOrderOut);
```

随机删除

```c
for(i = 0;i < 8;i++){
    RedBlackNode* dele = T->nil;
    while(dele == T->nil){
        int choose = rand() % 80;
        dele = RB_Find(T,keys[choose]);
    }
    fprintf(deleteout,"will delete: %d\n",dele->key);
    gettimeofday(&start,NULL);
    RB_Delete(T,dele);
    gettimeofday(&end,NULL);
    long deletime = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec);
    fprintf(time2out,"delete cost = %ld us\n",deletime);
    RB_InOrderTraverse(T,T->root,deleteout);
}
```

```c
fclose(InOrderOut);
fclose(PreOrderOut);
fclose(PastOrderOut);
fclose(time1out);
fclose(deleteout);
fclose(time2out);
Clear_RBTree(T,T->root);
T->root = T->nil;
```

d. **实验数据分析**

对函数用时统计做图分析并解释数量级与理论的异同, 详见第 5 部分 。

(2) 实验 2:

a. **编写简单的 Python 程序生成所需的随机整数区间序列:**

```python
import numpy as np
output = open("./PB16111485-project3/inputB/input_inteval.txt",'w')
interval_left = []
while len(interval_left) < 30 :
    tmp = np.random.randint(0,24)
    if tmp not in interval_left:
        interval_left.extend([tmp])

    tmp = np.random.randint(30,49)
    if tmp not in interval_left:
        interval_left.extend([tmp])

for left in interval_left :
    if(left < 25):
        right = np.random.randint(left + 1,25)
    else:
        right = np.random.randint(left + 1,50)
    output.write(str(left) + ',' + str(right) + "\n")

output.close()
```

b. 编写 C 语言函数实现对于区间树相关数据结构定义和左旋、右旋、插入、插入调整、删除、删除调整和遍历函数，详见第 4 部分。

c. 在 main 函数中进行逐个插入建树和随机删除测试，测试单元如下：

```c
//insert
for(i = 0;i < SIZE;i++){
    IntervalTreeNode* Item = malloc(sizeof(IntervalTreeNode));
    Item->high = h[i];
    Item->low = l[i];
    IT_Insert(T,Item);
}

FILE* InOrder = fopen("../outputB/inorder.txt","w");
FILE* DeleteData = fopen("../outputB/delete_data.txt","w");
FILE* Search = fopen("../outputB/search.txt","w");
//traverse
IT_InOrderTraverse(T,T->Root,InOrder);
int low, high;
IntervalTreeNode* temp;
```

逐个插入建树

中序遍历

```c
//search
/************************************************/
high = rand() % 25 + 1;
low = rand() % high;
temp = IT_Search(T,low,high);
if(temp != T->Nil){
    fprintf(Search,"Search: [%d, %d], Result: [%d, %d]\n",low, high, temp->low, temp->high);
}
else{
    fprintf(Search,"Search: [%d, %d], Result: NULL\n",low, high);
}
/************************************************/
high = rand() % 3 + 27;
low = high - rand() % (high - 26) - 1;
temp = IT_Search(T,low,high);
if(temp != T->Nil){
    fprintf(Search,"Search: [%d, %d], Result: [%d, %d]\n",low, high, temp->low, temp->high);
}
else{
    fprintf(Search,"Search: [%d, %d], Result: NULL\n",low, high);
}
/************************************************/
high = rand() % 20 + 31;
low = high - rand() % (high - 30) - 1;
temp = IT_Search(T,low,high);
if(temp != T->Nil){
    fprintf(Search,"Search: [%d, %d], Result: [%d, %d]\n",low, high, temp->low, temp->high);
}
else{
    fprintf(Search,"Search: [%d, %d], Result: NULL\n",low, high);
}
//delete
for(i = 0;i < 3;i++){
    IntervalTreeNode* dele = T->Nil;
    while(dele == T->Nil){
        int tmp = rand() % 30;
        dele = IT_Search(T,l[tmp],h[tmp]);
    }
    fprintf(DeleteData,"will delete: [%d, %d]\n",dele->low,dele->high);
    IT_Delete(T,dele);
    IT_InOrderTraverse(T,T->Root,DeleteData);
}
```

查找

随机删除

4. **实验关键代码截图（结合文字说明）**

(1) 实验1：

A. 红黑树数据结构：

```c
enum Color{red, black};
typedef struct RedBlackNode{
    struct RedBlackNode* left;
    struct RedBlackNode* right;
    struct RedBlackNode* parent;
    int key;
    enum Color color;
}RedBlackNode;
typedef struct RedBlackTree{
    RedBlackNode* root;
    RedBlackNode* nil;
}RedBlackTree;
```

B. 红黑树左旋右旋：

```c
void RB_LeftRotate(RedBlackTree* T,RedBlackNode* x){
    RedBlackNode* y = x->right;
    x->right = y->left;
    if(y->left != T->nil){
        y->left->parent = x;
    }
    y->parent = x->parent;
    if(x->parent == T->nil){
        T->root = y;
    }
    else{
        if(x == x->parent->left){
            x->parent->left = y;
        }
        else{
            x->parent->right = y;
        }
    }
    y->left = x;
    x->parent = y;
}
```

```c
void RB_RightRotate(RedBlackTree* T,RedBlackNode* x){
    RedBlackNode* y = x->left;
    x->left = y->right;
    if(y->right != T->nil){
        y->right->parent = x;
    }
    y->parent = x->parent;
    if(x->parent == T->nil){
        T->root = y;
    }
    else{
        if(x == x->parent->left){
            x->parent->left = y;
        }
        else{
            x->parent->right = y;
        }
    }
    y->right = x;
    x->parent = y;
}
```

C. 红黑树插入:

```
void RB_Insert(RedBlackTree* T,RedBlackNode* x){
    RedBlackNode* y = T->nil;
    RedBlackNode* z = T->root;
    while(z != T->nil){
        y = z;
        if(x->key > z->key){
            z = z->right;
        }
        else{
            z = z->left;
        }
    }
    x->parent = y;
    if(y == T->nil){
        T->root = x;
    }
    else{
        if(x->key < y->key){
            y->left = x;
        }
        else{
            y->right = x;
        }
    }
    x->left = x->right = T->nil;
    x->color = red;
    RB_InsertFixup(T,x);
}
```

D. 红黑树插入调整：（注释如图，不再赘述）

```
void RB_InsertFixup(RedBlackTree* T,RedBlackNode* x){
    while(x->parent->color == red){     //破坏红黑树性质：红结点子节点为黑节点，所以需要调整
        if(x->parent == x->parent->parent->left){ //这一步区分主要是叔节点位置以及之后的左右旋
            RedBlackNode* y = x->parent->parent->right;
            if(y->color == red){    //case1: 同层双红，问题上移
                x->parent->color = black;
                y->color = black;
                x->parent->parent->color = red;
                x = x->parent->parent;
            }
            else{   //case2,3:调整之后各节点黑高不变，调整完即结束
                if(x == x->parent->right){  //case2:统一到违反规定的红色子节点在其父左儿子
                    x = x->parent;
                    RB_LeftRotate(T,x);
                }
                //case3:红色父节点变黑上移，黑色爷节点变红下移，各节点黑高不变，红结点子节点为黑节点性质得到恢复
                x->parent->color = black;
                x->parent->parent->color = red;
                RB_RightRotate(T,x->parent->parent);
            }
        }
        else{
            RedBlackNode* y = x->parent->parent->left;
            if(y->color == red){    //case1: 同层双红，问题上移
                x->parent->color = black;
                y->color = black;
                x->parent->parent->color = red;
                x = x->parent->parent;
            }
            else{   //case2,3:调整之后各节点黑高不变，调整完即结束
                if(x == x->parent->left){  //case2:统一到违反规定的红色子节点在其父右儿子
                    x = x->parent;
                    RB_RightRotate(T,x);
                }
                //case3:红色父节点变黑上移，黑色爷节点变红下移，各节点黑高不变，红结点子节点为黑节点性质得到恢复
                x->parent->color = black;
                x->parent->parent->color = red;
                RB_LeftRotate(T,x->parent->parent);
            }
        }
    }
    T->root->color = black;
}
```

E. 红黑树删除:

```
void RB_Delete(RedBlackTree* T,RedBlackNode* x){
    RedBlackNode* y = x;
    int y_original_color = y->color;     //y是真正被删除的节点
    RedBlackNode* z;                     //z是取代y的节点
    if(x->left == T->nil){
        z = x->right;
        RB_TransPlant(T,x,z);
    }
    else{
        if(x->right == T->nil){
            z = x->left;
            RB_TransPlant(T,x,z);
        }
        else{
            y = x->right;
            while(y->left != T->nil){
                y = y->left;
            }
            y_original_color = y->color;
            z = y->right;
            if(y != x->right){
                RB_TransPlant(T,y,z);
                y->right = x->right;
                y->right->parent = y;
            }
            else{
                z->parent = y;
            }
            RB_TransPlant(T,x,y);
            y->left = x->left;
            y->left->parent = y;
            y->color = x->color;
            free(x);
        }
    }
    if(y_original_color == black){
        RB_DeleteFixup(T,z);
    }
}
```

防止子节点是 T->nil 的情况，是由所用数据结构特性导致的，对于操作正确性至关重要，不可随意删除

F. 红黑树删除调整:（注释如图，不再赘述）

```
void RB_DeleteFixup(RedBlackTree* T,RedBlackNode* x){
    while(x != T->root && x->color == black){   //x上有两重黑，所以需要调整
        if(x == x->parent->left){
            RedBlackNode* w = x->parent->right;
            if(w->color == red){    //case1:兄弟节点为红色，统一到兄弟节点为黑色
                w->color = black;
                x->parent->color = red;
                RB_LeftRotate(T,x->parent);
                w = x->parent->right;
            }
            if(w->left->color == black && w->right->color == black){
                //case2:兄弟节点两个子节点都是黑色，则可以安全地脱掉一层黑色到父节点
                w->color = red;
                x = x->parent;
            }
            else{
                if(w->right->color == black){
                    //case3:兄弟节点的左儿子是红色，统一到右儿子是红色
                    w->left->color = black;
                    w->color = red;
                    RB_RightRotate(T,w);
                    w = x->parent->right;
                }
                //case4:修改颜色保持其他节点黑高，重叠的黑色脱到下移的原父节点
                w->color = x->parent->color;
                w->right->color = black;
                x->parent->color = black;
                RB_LeftRotate(T,x->parent);
                x = T->root;    //此时重叠黑高度已经被消化，强行退出循环
            }
        }
```

```
        else{
            RedBlackNode* w = x->parent->left;
            if(w->color == red){    //case1:兄弟节点为红色，统一到兄弟节点为黑色…
            }
            if(w->left->color == black && w->right->color == black){…
            }
            else{
                if(w->left->color == black){
                    //case3:兄弟节点的右儿子是红色，统一到左儿子是红色
                    w->right->color = black;
                    w->color = red;
                    RB_LeftRotate(T,w);
                    w = x->parent->left;
                }
                //case4:修改颜色保持其他节点黑高，重叠的黑色脱到下移的原父节点
                w->color = x->parent->color;
                w->left->color = black;
                x->parent->color = black;
                RB_RightRotate(T,x->parent);
                x = T->root;    //此时重叠黑高度已经被消化，强行退出循环
            }
        }
    }
    x->color = black;
}
```

G. 红黑树遍历：

```
void RB_InOrderTraverse(RedBlackTree* T, RedBlackNode* from, FILE* out){
    if(from == T->nil){
        return;
    }
    RB_InOrderTraverse(T,from->left,out);
    fprintf(out,"%d\n",from->key);
    RB_InOrderTraverse(T,from->right,out);        先序遍历
}
void RB_ProOrderTraverse(RedBlackTree* T, RedBlackNode* from, FILE* out){
    if(from == T->nil){
        return;
    }
    fprintf(out,"%d\n",from->key);                 中序遍历
    RB_ProOrderTraverse(T,from->left,out);
    RB_ProOrderTraverse(T,from->right,out);
}
void RB_PastOrderTraverse(RedBlackTree* T, RedBlackNode* from, FILE* out){
    if(from == T->nil){
        return;
    }
    fprintf(out,"%d\n",from->key);                 后序遍历
    RB_PastOrderTraverse(T,from->left,out);
    RB_PastOrderTraverse(T,from->right,out);
}
```

(2) 实验 2:

A. 区间树数据结构:

```c
enum Color{Red, Black};
typedef struct IntervalTreeNode{
    struct IntervalTreeNode* LeftChild;
    struct IntervalTreeNode* RightChild;
    struct IntervalTreeNode* Parent;
    int low,high,max;
    enum Color color;
}IntervalTreeNode;
typedef struct IntervalTree{
    IntervalTreeNode* Root;
    IntervalTreeNode* Nil;
}IntervalTree;
```

B. 区间树左旋右旋:

```c
void IT_LeftRotate(IntervalTree* T,IntervalTreeNode* x){
    IntervalTreeNode* y = x->RightChild;
    x->RightChild = y->LeftChild;
    if(y->LeftChild != T->Nil){
        y->LeftChild->Parent = x;
    }
    y->Parent = x->Parent;
    if(x->Parent == T->Nil){
        T->Root = y;
    }
    else{
        if(x == x->Parent->LeftChild){
            x->Parent->LeftChild = y;
        }
        else{
            x->Parent->RightChild = y;
        }
    }
    y->LeftChild = x;
    y->max = x->max;
    x->Parent = y;
    x->max = max3(x->high,x->LeftChild->max,x->RightChild->max);
}
```

扩展数据域的维护时机

```c
void IT_RightRotate(IntervalTree* T,IntervalTreeNode* x){
    IntervalTreeNode* y = x->LeftChild;
    x->LeftChild = y->RightChild;
    if(y->RightChild != T->Nil){
        y->RightChild->Parent = x;
    }
    y->Parent = x->Parent;
    if(y->Parent == T->Nil){
        T->Root = y;
    }
    else{
        if(x == x->Parent->LeftChild){
            x->Parent->LeftChild = y;
        }
        else{
            x->Parent->RightChild = y;
        }
    }
    y->RightChild = x;
    y->max = x->max;
    x->Parent = y;
    x->max = max3(x->high,x->LeftChild->max,x->RightChild->max);
}
```

C. 区间树插入:

```
void IT_Insert(IntervalTree* T,IntervalTreeNode* x){
    IntervalTreeNode* y = T->Nil;
    IntervalTreeNode* z = T->Root;
    while(z != T->Nil){
        y = z;
        if(x->low < z->low){
            z = z->LeftChild;
        }
        else{
            z = z->RightChild;
        }
    }
    x->Parent = y;
    if(y == T->Nil){
        T->Root = x;
    }
    else{
        if(x->low > y->low){
            y->RightChild = x;
        }
        else{
            y->LeftChild = x;
        }
    }
    x->LeftChild = x->RightChild = T->Nil;
    x->max = x->high;
    x->color = Red;
    while(y != T->Nil){
        y->max = max3(y->high,y->LeftChild->max,y->RightChild->max);
        y = y->Parent;
    }
    IT_InsertFixup(T,x);
}
```

自底向上维护扩展数据域

D. 区间树插入调整：（不需要在这里维护扩展数据域，所以几乎与红黑树插入调整相同，这里不再赘述）

E. 区间树删除：

```
void IT_Delete(IntervalTree* T,IntervalTreeNode* x){
    IntervalTreeNode* y = x;
    int y_origin_color = y->color;
    IntervalTreeNode* z;
    if(x->LeftChild == T->Nil){
        z = x->LeftChild;
        IT_TransPlant(T,x,z);
    }
    else{
        if(x->RightChild == T->Nil){
            z = x->RightChild;
            IT_TransPlant(T,x,z);
        }
        else{
            y = x->RightChild;
            while(y->LeftChild != T->Nil){
                y = y->LeftChild;
            }
            y_origin_color = y->color;
            z = y->RightChild;
            if(y != x->RightChild){
                IT_TransPlant(T,y,z);
                y->RightChild = x->RightChild;
                y->RightChild->Parent = y;
            }
            else{
                z->Parent = y;
            }
            IT_TransPlant(T,x,y);
            y->LeftChild = x->LeftChild;
            y->LeftChild->Parent = y;
            y->color = x->color;
            free(x);
        }
    }
    y = z;
    while(y != T->Nil){
        y->max = max3(y->high,y->LeftChild->max,y->RightChild->max);
        y = y->Parent;
    }
    if(y_origin_color == Black){
        IT_DeleteFixup(T,z);
    }
}
```

自底向上维护扩展数据域

F. 区间树删除调整：（不需要在这里维护扩展数据域，所以几乎与红黑树删除调整相同，这里不再赘述）

G. 区间树遍历：（不需要在这里维护扩展数据域，所以几乎与红黑树遍历相同，这里不再赘述）

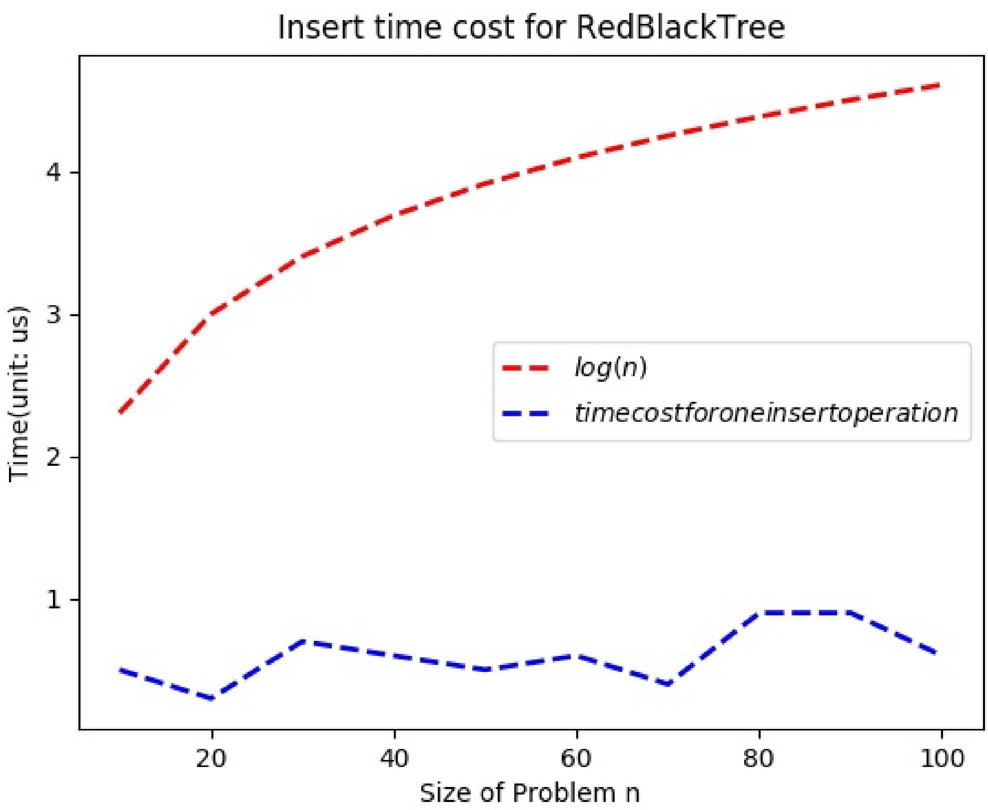
H. 区间树查找：

```
IntervalTreeNode* IT_Search(IntervalTree* T,int low,int high){
    IntervalTreeNode* x = T->Root;
    while(x != T->Nil && !(low <= x->high && high >= x->low)){
        if(x->LeftChild != T->Nil && x->LeftChild->max >= low){
            x = x->LeftChild;
        }
        else{
            x = x->RightChild;
        }
    }
    return x;
}
```

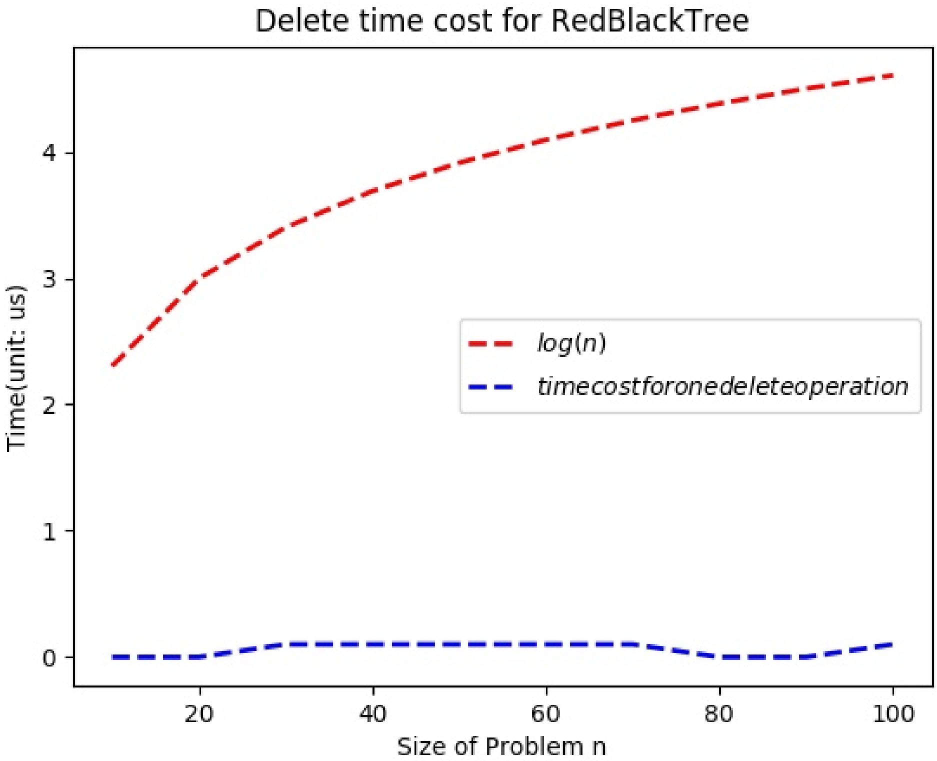
5. 实验结果、分析（结合相关数据图表分析）

(1) 实验1:

红黑树插入时间复杂度（每十次插入平均一次），可以看到满足 O(lgN)理论复杂度，而且实际表现远低于紧致上界，体现了红黑树数据结构的优越性。



红黑树删除时间复杂度（随机删除十分之一的节点），可以看到满足 O(lgN)理论复杂度，而且实际表现远低于紧致上界，体现了红黑树数据结构的优越性。



6. 实验心得

(1) 加深了对于红黑树和区间树数据结构和算法的认识
(2) 理解了红黑树删除算法中由数据结构导致的特性
(3) 具体了解了区间树扩展数据域的维护时机
(4) 增强了算法实现和编程能力