Advanced Algorithm Design

Red-black Tree

Jingjing Xia

Red-Black Tree

A red-black tree is a binary search tree, and each node contains one extra field: its color, it can be either black or red. There are five fields for each node: ***color, key, left, right, p***. If a node has not got child or parent, the corresponding pointer field of the node should points to the value NIL. We treat these NIL nodes as being pointers to external nodes (leaves) and the normal nodes (key-bearing nodes) as being internal nodes of the binary search tree.

If a binary search tree satisfies all the following red-black properties, it is a red-black tree.
Red-black properties:
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Figure 1 shows an example of a red-black tree.

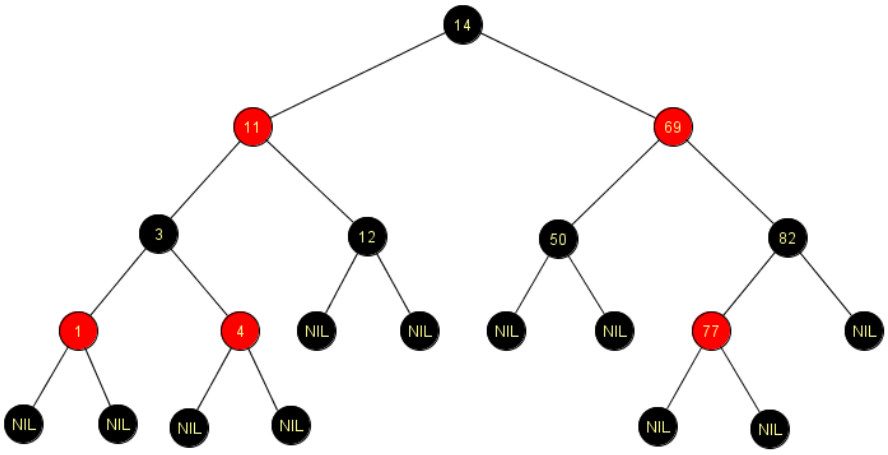


Figure 1

Insertion

Step1: Use RB-INSERT (similar as TREE-INSERT from binary search tree) to inset a node ***z*** into the tree.
Step 2: Color ***z*** red.
Step 3: If there is a violation of red-black tree properties, then use RB-INSERT-FIXUP to fix it.

Algorithm Description:

RB-Insert(*T*, *z*)

```
1.  y ← nil[T]
2.  x ← root[T]
3.  while x ≠ nil[T]
4.      do y ← x
5.          if key[z] < key[x]
```

```
6.              then x ← left[x]
7.              else x ← right[x]
8.    p[z] ← y
9.    if y = nil[T]
10.      then root[T] ← z
11.      else if key[z] < key[y]
12.           then  left[y] ← z
13.           else right[y] ← z
14.   left[z] ← nil[T]
15.   right[z] ← nil[T]
16.   color[z] ← RED
17.   RB-Insert-Fixup (T, z)
```

RB-Insert-Fixup (*T*, *z*)

```
1.    while color[p[z]] = RED
2.       do if p[z] = left[p[p[z]]]
3.          then y ← right[p[p[z]]]
4.               if color[y] = RED
5.                  then color[p[z]] ← BLACK     // Case 1
6.                       color[y] ← BLACk        // Case 1
7.                       color[p[p[z]]] ← RED    // Case 1
8.                       z ← p[p[z]]             // Case 1
9.               else if z = right[p[z]]  // color[y] ≠ RED
10.                 then z ← p[z]                 // Case 2
11.                      LEFT-ROTATE(T, z)        // Case 2
12.                 color[p[z]] ← BLACK           // Case 3
13.                 color[p[p[z]]] ← RED          // Case 3
14.                 RIGHT-ROTATE(T, p[p[z]])      // Case 3
15.         else (if p[z] = right[p[p[z]]]) (same as 3-14
16.                 with "right" and "left" exchanged)
17.   color[root[T ]] ← BLACK
```

Which of red-black properties can be violated when call RB-INSERT-FIXUP?

Property 1: Certainly continues to hold.
Property 2: It will be violated when ***z*** is the root and ***z*** is red.
Property 3: Certainly continues to hold.
Property 4: It will be violated when both ***z*** and ***p[z]*** are red.
Property 5: Certainly continues to hold, because we color ***z*** red.

If there is a violation of red-black tree properties, there is at most one, and it is either property 2 or 4.

The following 3 cases will violate the red-black tree properties and the following steps show how RB-INSERT-FIXUP works to resort the red-black properties. In fact, there are 6 cases in all, but another 3 and the following 3 cases are symmetrical.

Case 1: ***z***'s uncle ***y*** is red.

1. Color ***p[z]*** and ***y*** black.
2. Color ***p[p[z]]*** red.
3. Move pointer ***z*** two levels up to ***p[p[z]]***. If new ***z*** is red, the while loop repeats.

Case 2: ***z***'s uncle ***y*** is black and ***z*** is a right child.

1. Do left rotate on ***z***.
2. Go to case 3.

Case 3: ***z***'s uncle ***y*** is black and ***z*** is a left child.

1. Color ***p[z]*** black.
2. Color ***p[p[z]]*** red.
3. Do right rotate on ***p[p[z]]***.

Time Analysis

Since the height of a red-black tree of n nodes is ***O*(lg*n*)**, the total cost of RB-INSERT without call to RB-DELETE-FIXUP runs in ***O*(lg*n*)** time.

Within RB-INSERT-FIXUP, case 2 and case 3 each terminate after performing a constant number of color changing and at most two rotations. The while loop repeats only if case 1 is executed, and then the pointer ***z*** moves two levels up the tree. The worst case is pointer ***z*** moves up the tree at most ***O*(lg*n*)** times. Therefore the total number of times the while loop can be executed is ***O*(lg*n*)**.

Thus, RB-INSERT takes total of ***O*(lg*n*)** time.

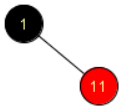
Note: There are at most two rotations at all if case 2 or case 3 is executed.

Implementation Details

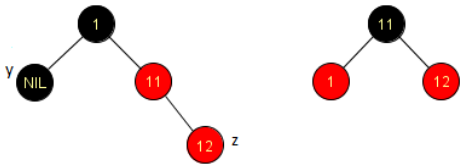
We stored 10 elements in an array A then used RB-INSERT to construct a red-black tree from A[0] to A[9].
A[0]=1; A[1]=11; A[2]=12; A[3]=69; A[4]=4; A[5]=14; A[6]=82; A[7]=50; A[8]=77; A[9]=3;

Insert A[0]=1: 1 becomes the root of the red-black tree and its color is black.

Insert A[1]=11: 11 becomes the right child of 1and its color is red.



Insert A[2]=12: ***z***'s uncle ***y*** is black and ***z*** is the right child, so case 3 can be applied(Figure a). We color 11 black and color 1 red then do left rotate on 1(Figure b).

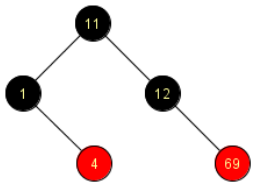


(a) (b)

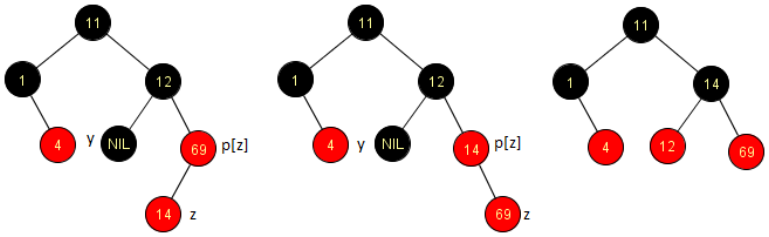
Insert A[3]=69: ***z***'s uncle **y** is red, so case 1 can be applied (Figure a). We color ***y*** and ***p[z]*** black, color 11 red, move pointer z two levels up to the root then color root black(Figure b).

(a) (b)

Insert A[4]=4:

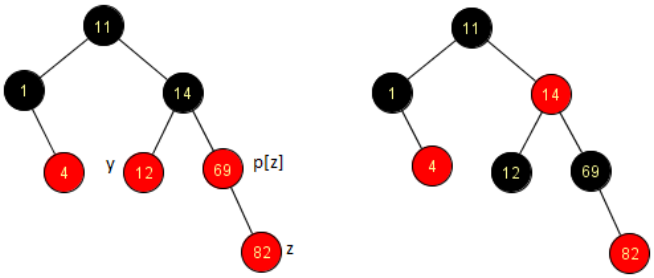


Insert A[5]=14: ***z***'s uncle ***y*** in black and ***z*** is the left child, so case 2 can be applied (Figure a). Do right rotate on **z** then 69 becomes new ***z*** and it's the right child, so case 3 can be applied (Figure b). Color ***p[z]*** black and 12 red then do left rotate on 12(Figure c).



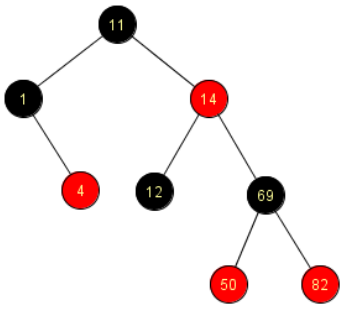
(a) (b) (c)

Insert A[6]=82: ***z***'s uncle **y** is red, so case 1 can be applied (Figure a). We color ***y*** and ***p[z]*** black, color 14 red, move pointer ***z*** two levels up to 14,because new ***p[z]*** is black, go out of the while loop (Figure b).

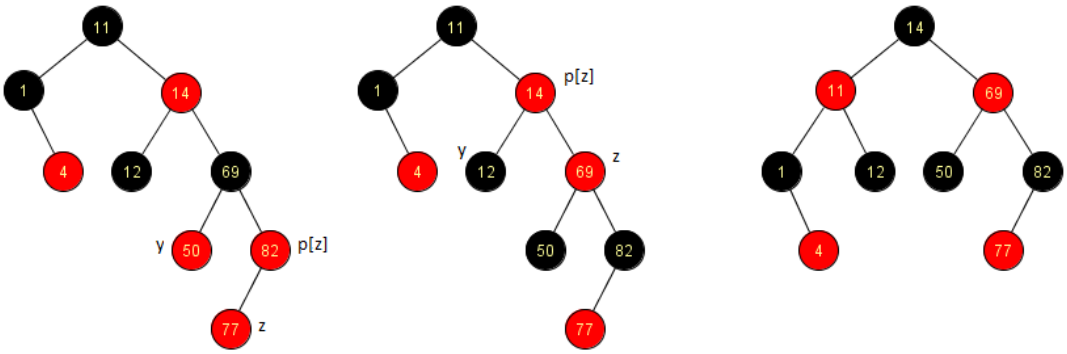


(a) (b)

Insert A[7]=50:

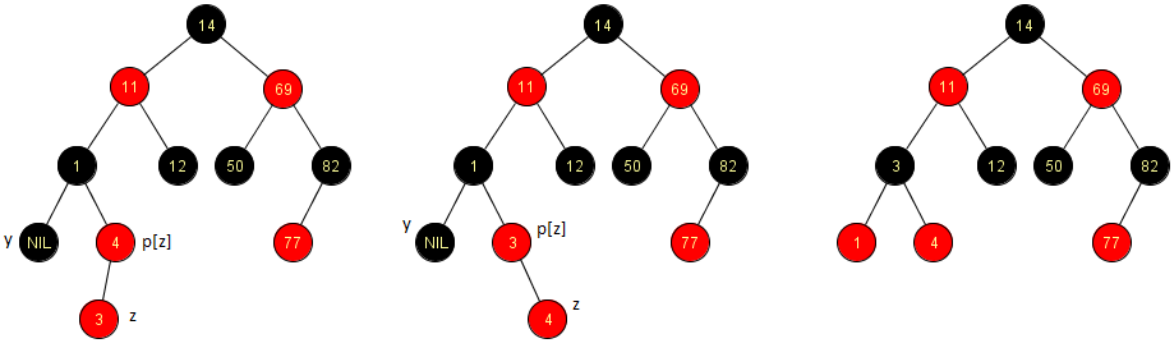


Insert A[8]=77: ***z***'s uncle **y** is red, so case 1 can be applied (Figure a). We color ***y*** and ***p[z]*** black, color 69 red, move pointer ***z*** two levels up to 69. Because new ***p[z]***is red and new ***y*** is black, so case 3 can be applied(Figure b). Color ***p[z]*** black and 11red, do left rotate on ***p[z]*** (Figure c).



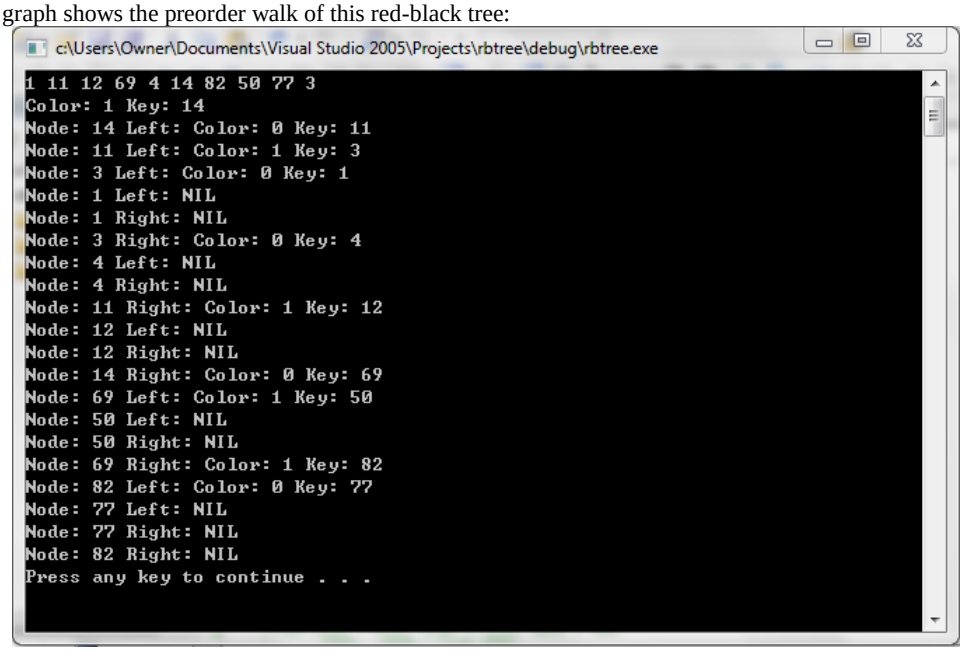
(a) (b) (c)

Insert A[9]=3: ***z***'s uncle ***y*** in black and ***z*** is the left child, so case 2 can be applied (Figure a). Do right rotate on **z** then 4 becomes new ***z*** and it's the right child, so case 3 can be applied (Figure b). Color ***p[z]*** black and 1 red then do left rotate on 1 (Figure c). The red-black tree with these 10 elements is showed by Figure c.



(a) (b) (c)

The following graph shows the preorder walk of this red-black tree:



Deletion

Step 1: Use RB-DELETE (similar as TREE-DELETE from binary search tree) to delete a node ***z*** into the tree *T*.
Step 2: If there is any violation of red-black tree properties, then use RB-DELETE-FIXUP to fix it.

Algorithm Description

RB-Delete(*T, z*)

```
1.  if left[z] = nil[T] or right[z] = nil[T]
2.     then y ← z                                  // Case 1
3.     else y ← TREE-SUCCESSOR(z)                  // Case 2
4.  if left[y] ≠nil[T ]
5.     then x ← left[y]
6.     else x ← right[y]
7.  p[x] ← p[y]   // Do this, even if x is nil[T]
8.  if p[y] = nil[T ]
9.    then root[T ] ← x
10.   else if y = left[p[y]]  (*if y is a left child.*)
11.         then left[p[y]] ← x
12.         else right[p[y]] ← x  (*if y is a right
13. if y ≠ z                                  child.*)
14.   then key[z] ← key[y]
15.   copy y's satellite data into z
16. if color[y] = BLACK
17.    then RB-Delete-Fixup(T, x)
18. return y
```

RB-Delete-Fixup(*T, x*)

```
1.  while x ≠ root[T] and color[x] = BLACK
2.     do if x = left[p[x]]
3.        then w ← right[p[x]]
4.             if color[w] = RED                          // Case 1
5.               then color[w] ← BLACK                    // Case 1
6.                    color[p[x]] ← RED                   // Case 1
7.                    LEFT-ROTATE(T, p[x])                // Case 1
8.                  w ← right[p[x]]                       // Case 1
9.          if color[left[w]] = BLACK and color[right[w]] = BLACK
10.           then color[w] ← RED                         // Case 2
11.                x ← p[x]                               // Case 2
12.                p[x] ← p[p[z]]                         // Case 2
13.           else if color[right[w]] = BLACK             // Case 3
14.                  then color[left[w]] ← BLACK          // Case 3
15.                       color[w] ← RED                 // Case 3
16.                       RIGHT-ROTATE(T,w)              // Case 3
17.                       w ← right[p[x]]                // Case 3
18.                color[w] ← color[p[x]]                // Case 4
19.                color[p[x]] ← BLACK                   // Case 4
20.                color[right[w]] ← BLACK               // Case 4
21.                LEFT-ROTATE(T, p[x])                  // Case 4
22.                x ← root[T ]                          // Case 4
23.     else (same as 3 - 21 with "right" and "left" exchanged)
24. color[x] ← BLACK
```

The code for RB-DELETE considers the following three cases:

Case 1: ***z*** has no child.

We modify ***p[z]*** to replace ***z*** with *nil*[*T*] as its child.

Case 2: ***z*** has one child.

We splice out z by making a new link between it child and its parent.

Case 3: ***z*** has two children.

We splice out ***z***'s successor y, which has no left child and replace z's key and satellite data with y's key and satellite data.

If y is black, we call RB-DELETE-FIXUP in line 16-17. If y is red, we just return y and all red-black properties still hold.

Which of red-black properties can be violated when call RB-DELETE-FIXUP?

Property 2: It will be violated when ***y*** is the root and a red child of y will become the new root.
Property 4: It will be violated when both ***x*** and ***p[y]*** (which is also ***p[x]***) are red.
Property 5: It will be violated by any ancestor of y in the tree. Because any path containing y has 1 fewer black node.

The following 4 cases will violate the red-black tree properties and the following steps show how RB-DELETE-FIXUP works to resort the red-black properties. In fact, there are 8cases in all, but another 4 and the following are symmetrical.

Case 1: ***x***'s brother ***w*** is red.

1. Color ***w*** black (Line 5).
2. Color ***p[x]*** red (Line 6). (***p[x]*** must be black, because ***w*** is red.)
3. Left rotate on ***p[x]*** (Line7).
4. Move pointer ***w*** to ***right[p[x]]*** (Line 8). New ***w*** was child of ***w*** before rotation and it must be black.
5. Go to case 2.

Case 2: ***x***'s brother ***w*** is black and both of ***w***'s children are black.

1. Color ***w*** red (Line 10).
2. Move pointer ***x*** one level up to ***p[x]*** (Line 11).
3. Move pointer ***p[x]*** one level up to ***p[p[x]]*** (Line 12).
4. Do the next iteration with the new ***x***. If entered this case from case 1, the while loop must terminate (because the new ***x*** was ***p[x]***, it was colored red in case 1). Then color this ***x*** black (Line 24).

Case 3: ***x***'s brother ***w*** is black, ***w***'s left child is red, and ***w***'s right child is black.

1. Color ***w***'s left child black (Line 14).
2. Color ***w*** red (Line 15).
3. Right rotate on ***w*** (Line 16).
4. Move pointer ***w*** to ***p[x]***'s right child (Line 17). (New ***w*** must be black with a red right child.)
5. Go to case 4.

Case 4: ***x***'s brother ***w*** is black, and ***w***'s right child is red.

1. Color w to be the same color as ***p[x]*** (Line 18).
2. Color ***p[x]*** black (Line 19).
3. Color ***w***'s right child black (Line 20).
4. Left rotate on ***p[x]*** (Line 21).
5. Move pointer ***x*** to ***root[T]*** (Line 22). It causes the while loop to terminate.

Time Analysis

Since the height of a red-black tree of n nodes is $O(\lg n)$, the total cost of the procedure RB-DELETE without call to RB-DELETE-FIXUP takes $O(\lg n)$ time.

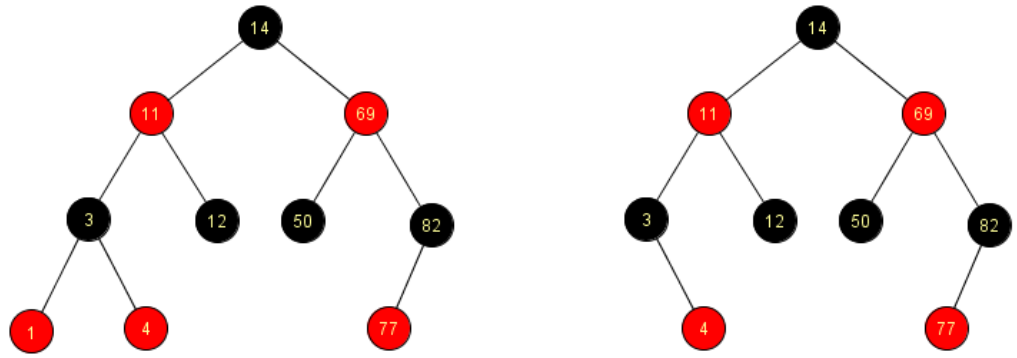
Within RB-DELETE-FIXUP, case 1, case 3 and case4 each terminate after performing a constant number of color changes and at most three rotations. The while loop can be repeated only in case 2, and the pointer ***x*** moves up the tree at most ***O*(lg*n*)** times and no rotation are performed. Therefore, RB-DELETE-FIXUP takes ***O*(lg*n*)** time and performs at three rotations.

Thus, the overall time for RB-DELETE is also ***O*(lg*n*)**.

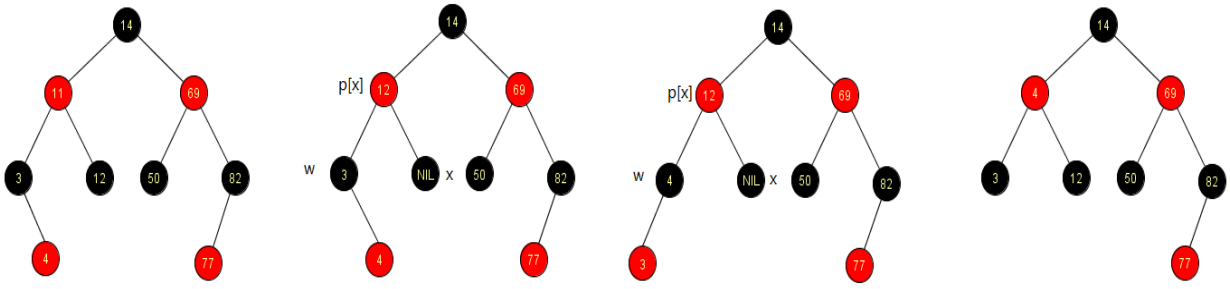
Implementation Details

We used RB-INSERT to construct a red-black tree, as the following I will show how does RB-DELETE destroy the red-black tree.

Delete A[0]=1: do not violate any property of red-black tree.

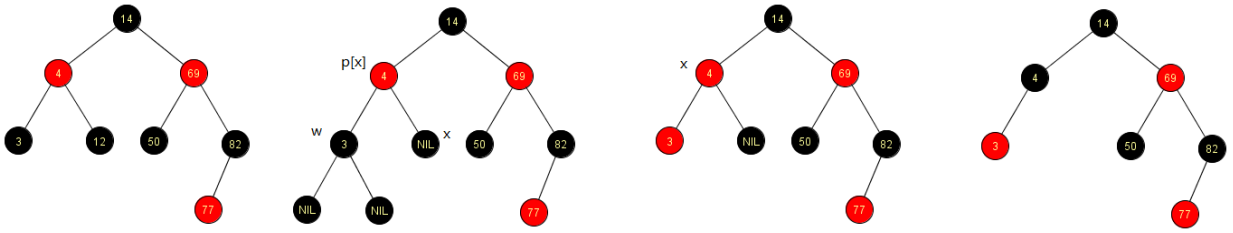


Delete A[1]=11: we splice out 11's successor 12(which is black). Now ***x*** is a NIL node, ***w*** is 3 and ***w***'s right child is red, so case 3 can be applied (Figure b). So we color ***w*** red and color 4 black, do left rotate on ***w*** then case 4 can be applied(Figure c) and 4 becomes the new ***w***. We color ***w*** red, color ***p[x]*** and 3 black then do right lotate on ***p[x]***(Figure d).



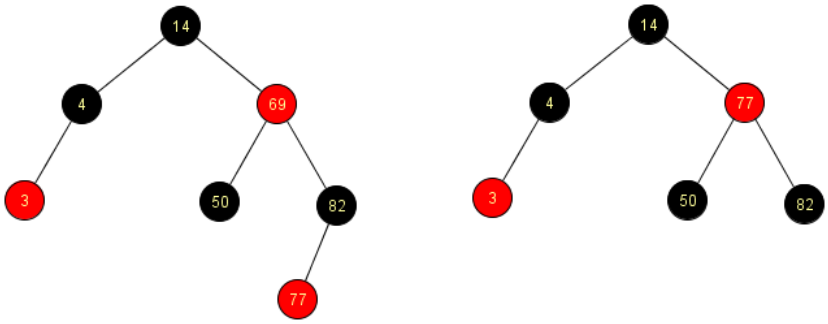
(a) (b) (c) (d)

Delete A[2]=12: We modify ***p[z]*** to replace ***z*** with *NIL* as its child. Now ***x*** is a NIL node, ***w*** is 3 and both of ***w***'s children are black, so case 2 can be applied (Figure b). We color ***w*** red and move pointer ***x*** to 4, now new ***x*** is 4(Figure c). Because ***x*** is red now, we can get out of the while loop and color ***x*** black(Figure d).



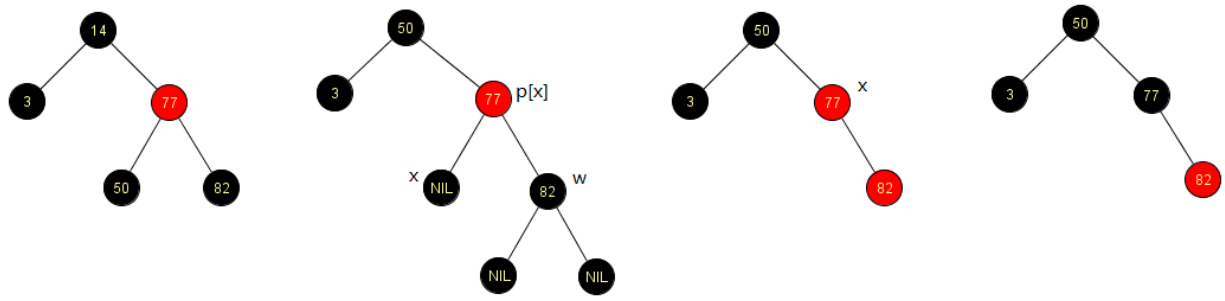
(a) (b) (c) (d)

Delete A[3]=69: we splice out 69's successor 77.



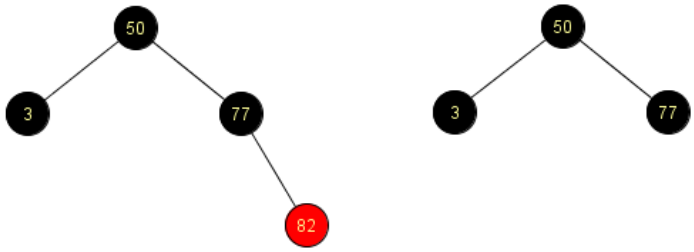
Delete A[4]=4: We splice out by making a new link between it child and its parent.

Delete A[5]=14: we splice out 14's successor 50. Now ***x*** is a NIL node, ***w*** is 82 and both of ***w***'s children are black, so case 2 can be applied (Figure b). We color w red and move pointer ***x*** to 77, now new ***x*** is 77(Figure c). Because ***x*** is red now, we can get out of the while loop and color ***x*** black(Figure d).

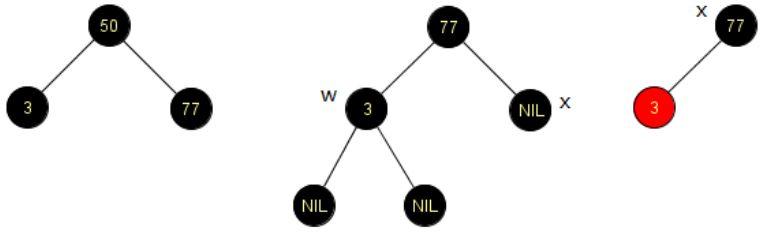


(a) (b) (c) (d)

Delete A[6]=82: do not violate any property of red-black tree.



Delete A[7]=50: we splice out 50's successor 77. Now ***x*** is a NIL node, ***w*** is 3 and both of ***w***'s children are black, so case 2 can be applied (Figure b). We color ***w*** red and move pointer ***x*** to 77, now new ***x*** is 77(Figure c). Because ***x*** is root now, we can get out of the while loop.



(a) (b) (c)

Delete A[8]=77: 3 becomes the root. Delete A[9] then the red-black tree is destroyed.

Test Results

It is mentioned before, the running time of RB-INSERT and RB-DELETE are both ***O*(lg*n*)**. The worst running time of constructing and destroying a red-black tree by using RB-INSERT and RB-DELETE are ***O*(lg*n!*)**.

Code for constructing a red-black tree:

```
for( int i = 0; i < n; i ++)
        {
        RBT * pZ = new RBT;
        pZ->key=A[i];
        rbInsert(root, pZ);
        }
```

From the code we can conculde: if the numeber of nodes is n, the worst running time of construct the tree is:
O(lg1+lg2+lg3+…+lgn)=O(lg(1*2*3*…*n))=O(lgn!)

Code for destroying a red-black tree:

```
for( int i = 0 ; i < n; i ++)
        {
        RBT * pZ = treeSearch(root, A[i]);
        rbDelete(root, pZ);}
```

```
RBT * treeSearch(RBT **root, int key)
{
        RBT * pX = * root;
        while(pX != NULL && pX->key != NILVALUE)
        {
        if( pX->key == key)
                return pX;
        else if(pX->key < key)
                pX = pX->ptRight;
        else
                pX = pX->ptLeft;
        }
        return NULL;
}
```

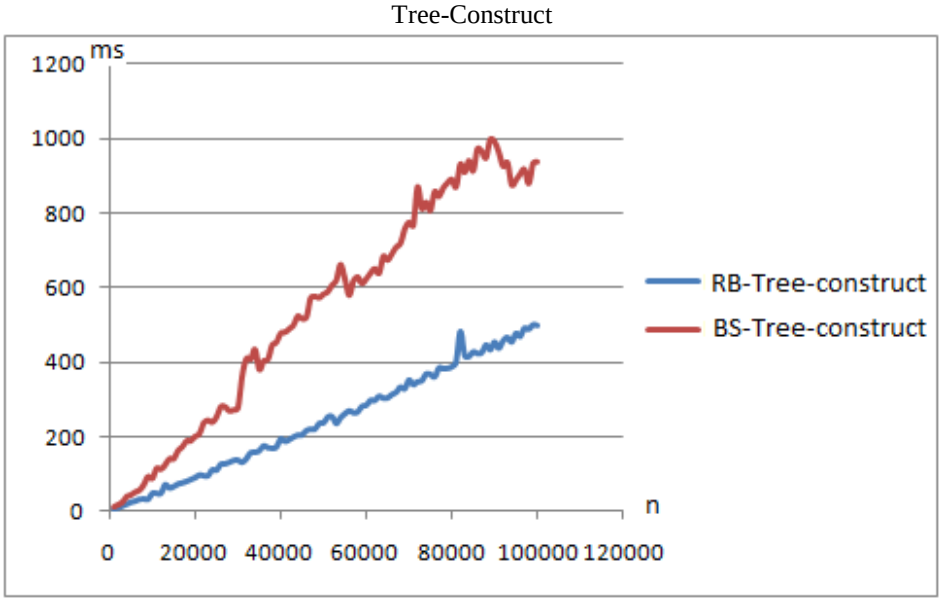
To destroy the red-black tree, firstly we need to search the tree to find the node then use RB-DELETE to delete it. The code of treeSearch() runs from the root then go down the tree to find the correct node. So the running time of treeSearch() is ***O*(lg*n*).** Thus, the overall time for destroying the tree is:

O(2(lg1+lg2+lg3+…+lgn))=O(2(lg1*2*3*…*n))=O(2lgn!)=O(lgn!)

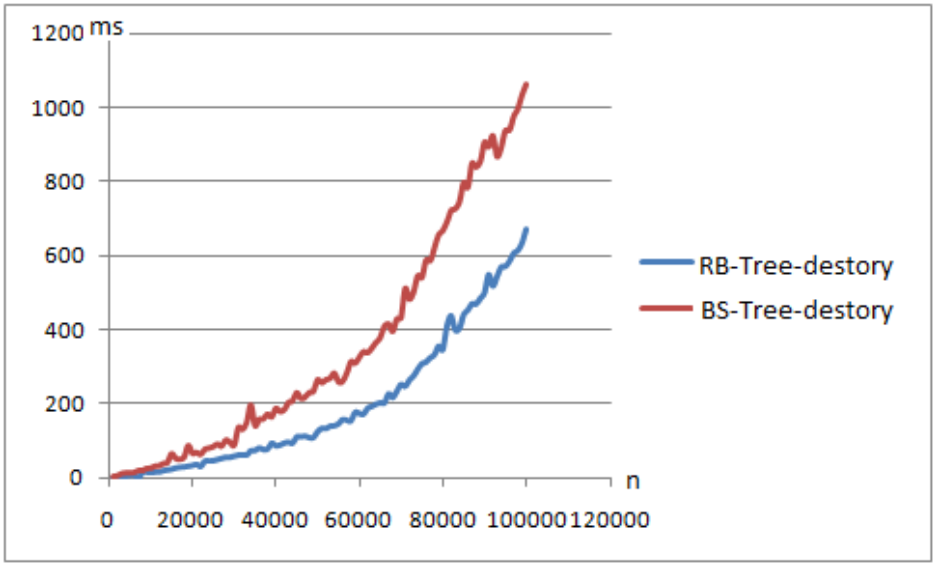
Compared with binary search tree, the worst running time of insertion and deletion is ***O(n),*** it occurs when the binary search tree is a linear chain of nodes. So the worst running time to construct and destroy a binary search tree is :

O(1+2+3+…+n)=O((1+n)n/2)=O((n^2+n)/2)=O(n^2)

The following graphs shows the results of the running time of constructing and destroying on these two kinds of trees.



Tree-Destroy



From the above graphs, we can conclude the operation of red-black tree runs better than binary search tree.