Universidad Tecnológica de Pereira

CAPÍTULO: ÁRBOLES ROJINEGROS (RED-BLACK TREES)
CURSO DE ESTRUCTURA DE DATOS - Código IS304

Programa de Ingeniería de Sistemas y Computación
Profesor Hugo Humberto Morales Peña
Semestre Agosto/Diciembre 2021

Árboles rojinegros

Propiedades de los árboles rojinegros

Un árbol rojinegro es un árbol binario de búsqueda, con un campo extra de almacenamiento por nodo: su color, que bien puede ser rojo (`RED`) o negro (`BLACK`). Restringiendo la forma en que los nodos pueden ser coloreados en cualquier camino desde la raíz hasta un nodo hoja, los árboles rojinegros aseguran que no hay camino tal que sea más del doble de largo que cualquier otro, así que, el árbol está aproximadamente balanceado.

Cada nodo del árbol ahora contiene los campos *color*, *key*, $left$, *right* y p. Si un hijo de un nodo no existe, el puntero correspondiente a dicho nodo apunta a un nodo hoja `NIL` (`NILLeaf`). Un nodo hoja `NIL` contiene todos los campos de un nodo normal, en el campo *color* se almacena el color negro, en el campo *key* se almacena el valor de $-\infty$ el cual se reconoce como el valor de `NILKey`, y los campos $left$ y *right* apuntan a `NIL`.

Un árbol binario de búsqueda es un árbol rojinegro si satisface las siguientes propiedades:

1. Cada nodo o es de color rojo o es de color negro.
2. La raíz del árbol es de color negro.
3. Cada nodo hoja `NIL` (`NILLeaf`) es de color negro.
4. Si un nodo es de color rojo, entonces sus dos nodos hijos son de color negro.
5. Para cada nodo, todos los caminos desde el nodo a los nodos hojas `NIL` descendientes contienen el mismo número de nodos de color negro.

Se le llama al número de nodos negros en cualquier camino desde, pero sin incluir, un nodo x hasta un nodo hoja `NIL`, la **altura de nodos negros** denotada por $bh(x)$. Por la propiedad 5, la noción de la altura de los nodos negros está bien definida, desde que todos los caminos descendientes desde el nodo tengan el mismo número de nodos negros. Se define la altura de nodos negros de un árbol rojinegro como la altura de nodos negros de su raíz.

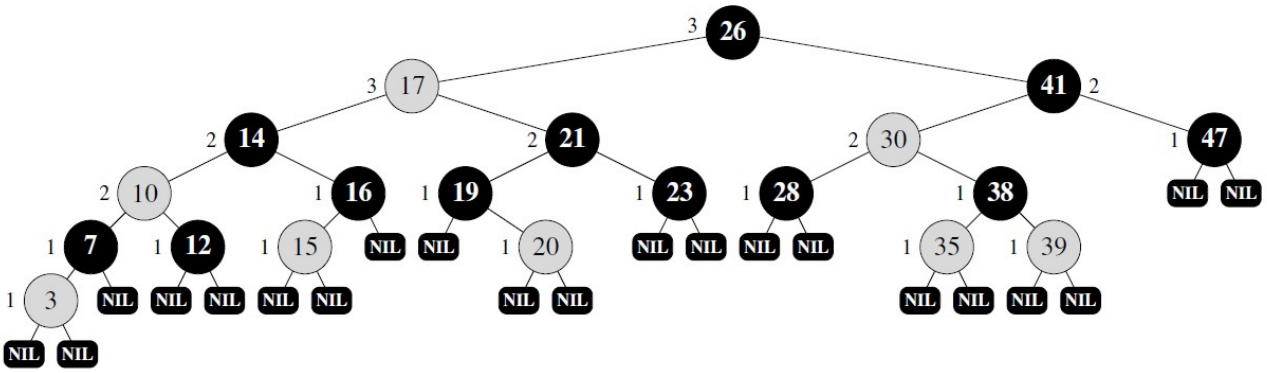

Figura 1: Un árbol rojinegro, con los nodos negros oscurecidos, y los nodos rojos sombreados. Cada nodo en un árbol rojinegro es rojo o negro, los hijos de un nodo rojo son ambos negros, y cada camino simple desde el nodo hasta un nodo hoja `NIL` descendiente contiene el mismo número de nodos negros. Cada nodo que es una hoja `NIL`, mostrado como `NIL`, es de color negro. Todo nodo diferente de hoja `NIL` es marcado con su altura de nodos negros; los nodos que son hojas `NIL` tienen una altura de nodos negros 0.

Rotaciones

Se cambia la estructura de punteros durante la rotación, la cual es una operación local dentro del árbol que preserva la propiedad del árbol binario de búsqueda. La Figura 2 muestra las dos clases de rotación: la rotación a la izquierda, y a la derecha. Cuando se hace una rotación a la izquierda de un nodo x, se asume que su hijo derecho y no es un nodo hoja `NIL` (`NILLeaf`). La rotación a la izquierda pivota alrededor del enlace de x a y. Esto hace que y sea la nueva raíz del subárbol, donde x es el hijo izquierdo de y y el hijo izquierdo de y ahora será el hijo derecho de x.

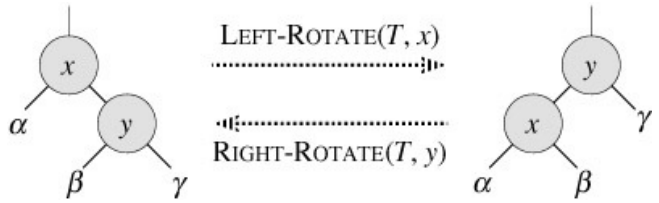

Figura 2: Las operaciones de rotación en un árbol binario de búsqueda. La operación `LEFT-ROTATE`(T, x) transforma la configuración de los dos nodos a la izquierda en la configuración de los de la derecha, cambiando un número constante de punteros. La configuración de la derecha puede ser transformada en la configuración de la izquierda por la operación inversa `RIGHT-ROTATE`(T, y). Las letras α, β y γ representan árboles arbitrarios. Una operación de rotación, preserva la propiedad del árbol binario de búsqueda: Las llaves en α preceden la llave $key[x]$ la cual precede las llaves en β, que a su vez precede a $key[\gamma]$, que precede las llaves en γ.

La Figura 3 muestra como se trabaja la rotación a la izquierda. El código para `RIGHT-ROTATE` es simétrico. Sólo los punteros son cambiados por la rotación; todos los demás campos en un nodo permanecen iguales.

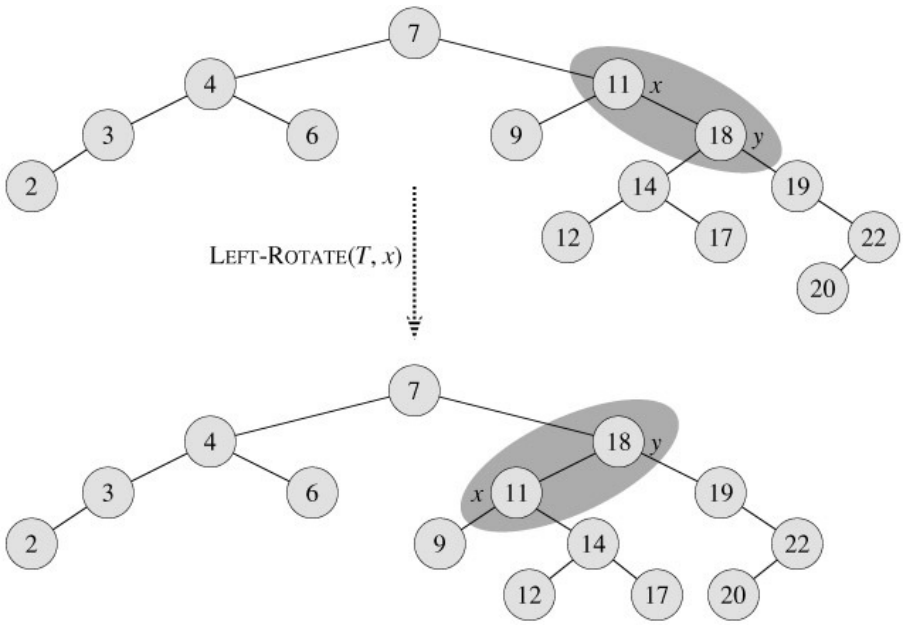

Figura 3: Un ejemplo de como el procedimiento `LEFT-ROTATE`(T, x) modifica un árbol binario de búsqueda. el recorrido inorder del árbol de entrada y el árbol modificado, producen la misma lista de las llaves.

El pseudocódigo para `LEFT-ROTATE` asume que $right[x] \neq$ `NILLeaf` y que el padre de la raíz es el nodo `NILLeaf`.

Rotación a la izquierda

function `LEFT-ROTATE`(T, x)

```
1   y ← right[x]                    ▷ Set y.
2   right[x] ← left[y]              ▷ Turn y's left subtree into x's right subtree.
3   p[left[y]] ← x
4   p[y] ← p[x]                     ▷ Link x's parent to y.
5   if p[x] == NILLeaf then
6       T ← y
7   else
8       if x == left[p[x]] then
9           left[p[x]] ← y
10      else
11          right[p[x]] ← y
12  left[y] ← x                     ▷ Put x on y's left.
13  p[x] ← y
14  return T
```

El pseudocódigo para `RIGHT-ROTATE` asume que $left[x] \neq$ `NILLeaf` y que el padre de la raíz es nodo `NILLeaf`.

Rotación a la derecha

function `RIGHT-ROTATE`(T, x)

```
1   y ← left[x]                    ▷ Set y.
2   left[x] ← right[y]             ▷ Turn y's right subtree into x's left subtree.
3   p[right[y]] ← x
4   p[y] ← p[x]                    ▷ Link x's parent to y.
5   if p[x] == NILLeaf then
6       T ← y
7   else
8       if x == right[p[x]] then
9           right[p[x]] ← y
10      else
11          left[p[x]] ← y
12  right[y] ← x                   ▷ Put x on y's right.
13  p[x] ← y
14  return T
```

Inserción

Se utiliza una versión ligeramente modificada del procedimiento `TREE-INSERT` para insertar un nuevo valor k en un árbol T como si este fuese un árbol binario de búsqueda ordinario. Se crea un nodo z, en el cual se almacena el valor k y se colorea de rojo. Para garantizar que las propiedades de un árbol rojinegro se preservan, entonces se llama un procedimiento auxiliar `RB-INSERT-FIXUP` para volver a colorear los nodos y realizar las rotaciones que sean del caso.

Algoritmo RB-Insert

function `RB-INSERT`(T, k)

```
1   Make node z
2   color[z] ← RED
3   key[z] ← k
4   left[z] ← ASSIGN-NIL-LEAF( )
5   p[left[z]] ← z
6   right[z] ← ASSIGN-NIL-LEAF( )
```

```
7   p[right[z]] ← z
8   if T ≠ NILLeaf then
9       y ← p[T]
10  else
11      y ← T
12  x ← T
13  while x ≠ NILLeaf do
14      y ← x
15      if key[z] < key[x] then
16          x ← left[x]
17      else
18          x ← right[x]
19  p[z] ← y
20  if y == NILLeaf then
21      T ← z
22  else
23      FREE( x )
24      if key[z] < key[y] then
25          left[y] ← z
26      else
27          right[y] ← z
28  T ← RB-INSERT-FIXUP( T, z )
29  return T
```

Hay 4 diferencias entre los procedimientos `TREE-INSERT` y `RB-INSERT`. Primera, todas las instancias de `NIL` en el `TREE-INSERT` son reemplazadas por el nodo `NILLeaf`. Segunda, se cambia a $left[z]$ y $right[z]$ al nodo `NILLeaf` en las líneas 4 y 6 de `RB-INSERT`, para mantener la estructura adecuada del árbol. Tercera, se colorea de rojo al nodo z en la línea 2. Cuarta, debido a que colorear z puede causar la violación a una de las propiedades del árbol rojinegro, se llama a `RB-INSERT-FIXUP`(T, z) en la línea 28 de `RB-INSERT` para restaurar estas propiedades.

Algoritmo ASSIGN-NIL-LEAF

```
function ASSIGN-NIL-LEAF( )
1   Make node w
2   color[w] ← BLACK
3   key[w] ← −∞
4   left[w] ← NIL
```

```
5   right[w] ← NIL
6   return w
```

Algoritmo RB-INSERT-FIXUP

```
function RB-INSERT-FIXUP( T, z )
1   while color[p[z]] == RED do
2       if p[z] == left[p[p[z]]] then
3           y ← right[p[p[z]]]
4           if color[y] == RED then
5               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
10              if z == right[p[z]] then
11                  z ← p[z]                           ▷ Case 2
12                  T ← LEFT-ROTATE( T, z )            ▷ Case 2
13              color[p[z]] ← BLACK                    ▷ Case 3
14              color[p[p[z]]] ← RED                   ▷ Case 3
15              T ← RIGHT-ROTATE( T, p[p[z]] )         ▷ Case 3
16      else
17          y ← left[p[p[z]]]
18          if color[y] == RED then
19              color[p[z]] ← BLACK                    ▷ Case 1
20              color[y] ← BLACK                       ▷ Case 1
21              color[p[p[z]]] ← RED                   ▷ Case 1
22              z ← p[p[z]]                            ▷ Case 1
23          else
24              if z == left[p[z]] then
25                  z ← p[z]                           ▷ Case 2
26                  T ← RIGHT-ROTATE( T, z )           ▷ Case 2
27              color[p[z]] ← BLACK                    ▷ Case 3
28              color[p[p[z]]] ← RED                   ▷ Case 3
29              T ← LEFT-ROTATE( T, p[p[z]] )          ▷ Case 3
```

30 $color[T] \leftarrow$ `BLACK`

31 **return** T

Para entender como funciona `RB-INSERT-FIXUP` se debe examinar el código en 3 pasos. Primero, se determinan cuales violaciones de las propiedades del árbol rojinegro fueron causadas por `RB-INSERT` cuando el nodo z es insertado y coloreado a rojo. Segundo, se examina el objetivo general del ciclo while en las líneas 1-29. Finalmente, se explora en cual de los 3 casos se rompe el ciclo while y se logra cumplir la meta. La Figura 4 muestra como `RB-INSERT-FIXUP` opera en un árbol rojinegro de ejemplo.

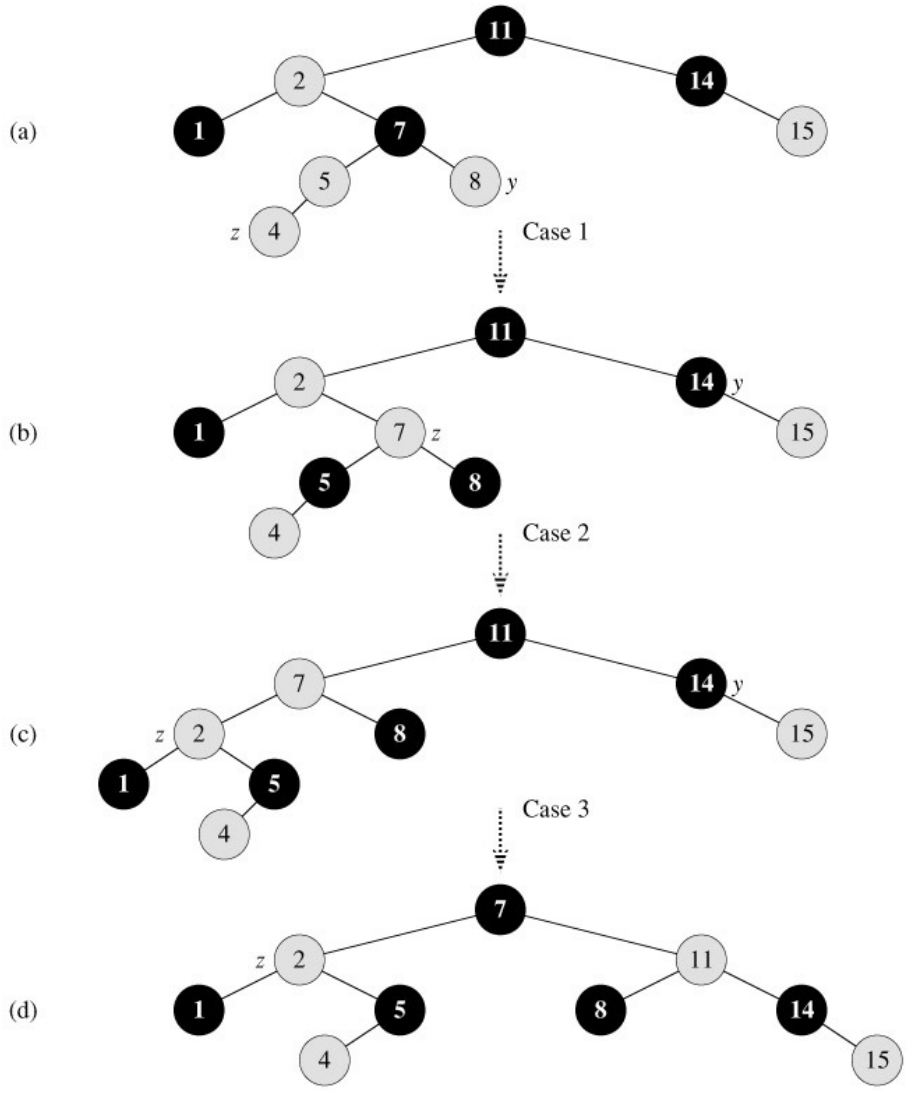

Figura 4: Funcionamiento de la función `RB-INSERT-FIXUP`. (a) Un nodo z después de la inserción. Como z y su padre $p[z]$ son ambos rojos, una violación de la propiedad 4 ocurre. Como el tío y de z es rojo, el caso 1 en el código puede ser aplicado. Los nodos vuelven a ser coloreados y el puntero z se mueve hacia arriba del árbol, resultando en el árbol mostrado en (b). Una vez más, z y su padre son ambos rojos, pero el tío y de z es negro. Como z es el hijo derecho de $p[z]$, el caso 2 puede ser aplicado. Una rotación a la izquierda es desarrollada, y el árbol que resulta es mostrado en (c). Ahora z es el hijo izquierdo de su padre, y el caso 3 puede ser aplicado. Una rotación a la derecha produce el árbol en (d), el cual ya es un árbol rojinegro al cumplir las cinco propiedades.

Ejercicio 1:

Considerar que inicialmente se cuenta con un árbol rojinegro vacío, insertar uno a uno los siguientes elementos respetando el orden siguiente: 1, 2, 3, 4, 5, 6, 7 y 8. Mostrar el gráfico que representa el árbol rojinegro después de insertar uno a uno los elementos pedidos.

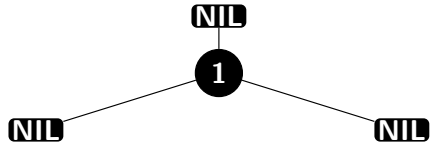

Figura 5: Árbol Rojinegro después de insertar el número 1.

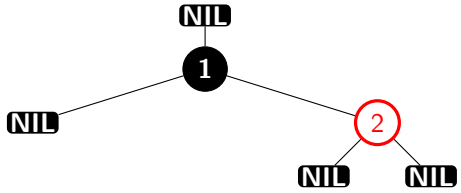

Figura 6: Árbol Rojinegro después de insertar el número 2.

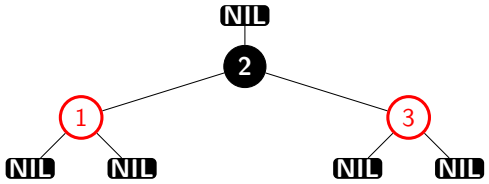

Figura 7: Árbol Rojinegro después de insertar el número 3.

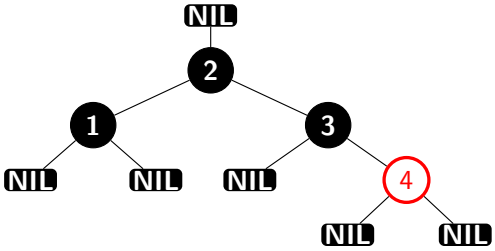

Figura 8: Árbol Rojinegro después de insertar el número 4.

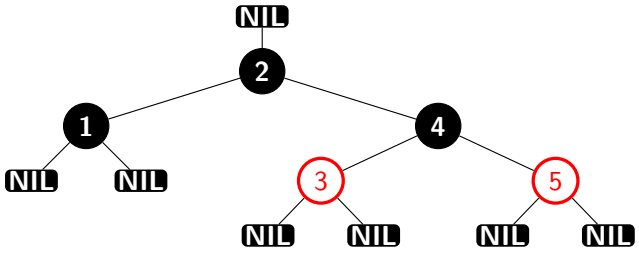

Figura 9: Árbol Rojinegro después de insertar el número 5.

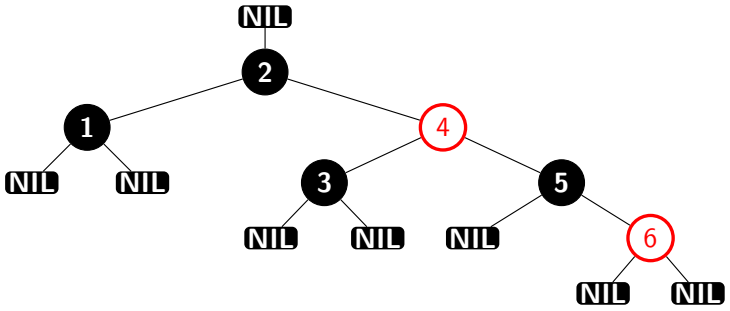

Figura 10: Árbol Rojinegro después de insertar el número 6.

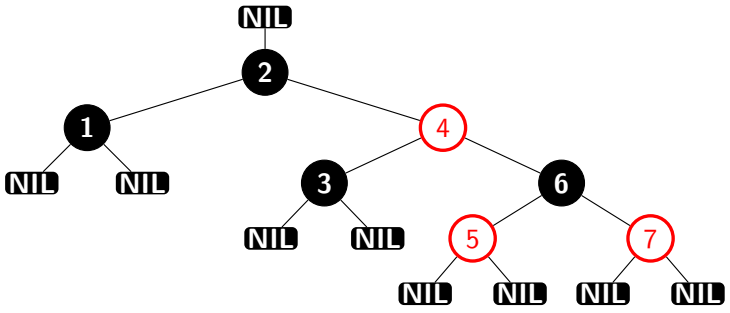

Figura 11: Árbol Rojinegro después de insertar el número 7.

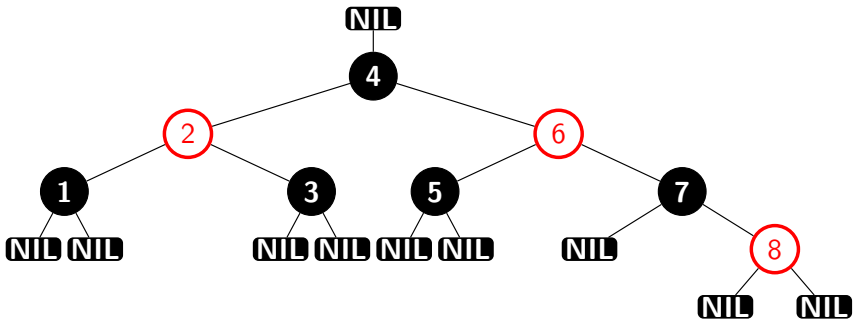

Figura 12: Árbol Rojinegro después de insertar el número 8.

Borrado

El procedimiento `RB-DELETE` es una pequeña modificación del procedimiento `TREE-DELETE`. Después de borrar un nodo, es llamado el procedimiento auxiliar `RB-DELETE-FIXUP` que ajusta los colores de los nodos y realiza tantas rotaciones como sean necesarias para restaurar las propiedades de árboles rojinegros.

Algoritmo RB-DELETE

function `RB-DELETE`(T, z)

1 **if** $left[z]$ == `NILLeaf` **or** $right[z]$ == `NILLeaf` **then**
2 $y \leftarrow z$
3 **else**
4 $y \leftarrow$`TREE-SUCCESSOR`(z)
5 **if** $left[y] \neq$ `NILLeaf` **then**
6 $x \leftarrow left[y]$
7 **else**
8 $x \leftarrow right[y]$
9 $p[x] \leftarrow p[y]$
10 **if** $p[y]$ == `NILLeaf` **then**
11 $T \leftarrow x$
12 **else**
13 **if** $y == left[p[y]]$ **then**
14 $left[p[y]] \leftarrow x$
15 **else**
16 $right[p[y]] \leftarrow x$
17 **if** $y \neq z$ **then**
18 $key[z] \leftarrow key[y]$
19 Copy all information fields from y to z

```
20  if color[y] == BLACK then
21      T ←RB-DELETE-FIXUP( T, x )
22  if y == p[left[y]] then
23      FREE( left[y] )
24  if y == p[right[y]] then
25      FREE( right[y] )
26  FREE( y )
27  return T
```

Hay 3 diferencias entre los procedimientos `TREE-DELETE` y `RB-DELETE`. La primera, todas las referencias a `NIL` en `TREE-DELETE` son reemplazadas por las referencias al nodo `NILLeaf` en `RB-DELETE`. Segunda, la prueba de que si x es `NIL` en la línea 9 de `TREE-DELETE` es removida, y la asignación $p[x] \leftarrow p[y]$ es realizada sin condiciones en la línea 9 de `RB-DELETE`. Por lo tanto, si x es el nodo `NILLeaf`, el apuntador a su padre, apunta al padre del nodo y. Tercera, un llamado a `RB-DELETE-FIXUP` es realizado en las líneas 20-21 si y es negro. Si y es rojo, las propiedades se mantienen cuando y es borrado, debido a las siguientes razones:

- Ninguna altura negra en el árbol ha cambiado.
- No hay nodos rojos adyacentes, y
- Desde que y no sea la raíz, si este es rojo, la raíz permanecerá negra.

El nodo x utilizado en el llamado a `RB-DELETE-FIXUP` es uno de dos nodos: o el nodo que era hijo único de y antes de que y fuese borrado si y tuviese un hijo que no fuese el nodo `NILLeaf`, o si y no tiene hijos, x es el nodo `NILLeaf`. En éste último caso, la asignación incondicional en la línea 9 garantiza que el padre de x es ahora el nodo que antes fue el padre de y.

Algoritmo RB-DELETE-FIXUP

```
function RB-DELETE-FIXUP( T, x )
1   while x ≠ T and color[x] == BLACK do
2       if x == left[p[x]] then
3           w ← right[p[x]]
4           if color[w] == RED then
5               color[w] ← BLACK                          ▷ Case 1
6               color[p[x]] ← RED                         ▷ Case 1
7               T ← 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 then
10              color[w] ← RED                            ▷ Case 2
11              x ← p[x]                                  ▷ Case 2
12          else
```

```
13              if color[right[w]] == BLACK then
14                  color[left[w]] ← BLACK                  ▷ Case 3
15                  color[w] ←RED                           ▷ Case 3
16                  T ← 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              T ← LEFT-ROTATE( T, p[x] )                  ▷ Case 4
22              x ← T                                       ▷ Case 4
23      else
24          w ← left[p[x]]
25          if color[w] == RED  then
26              color[w] ← BLACK                            ▷ Case 1
27              color[p[x]] ← RED                           ▷ Case 1
28              T ← RIGHT-ROTATE( T, p[x] )                 ▷ Case 1
29              w ← left[p[x]]                              ▷ Case 1
30          if color[right[w]] == BLACK  and color[left[w]] == BLACK then
31              color[w] ← RED                              ▷ Case 2
32              x ← p[x]                                    ▷ Case 2
33          else
34              if color[left[w]] == BLACK then
35                  color[right[w]] ← BLACK                 ▷ Case 3
36                  color[w] ←RED                           ▷ Case 3
37                  T ← LEFT-ROTATE( T, w )                 ▷ Case 3
38                  w ← left[p[x]]                          ▷ Case 3
39              color[w] ← color[p[x]]                      ▷ Case 4
40              color[p[x]] ← BLACK                         ▷ Case 4
41              color[left[w]] ← BLACK                      ▷ Case 4
42              T ← RIGHT-ROTATE( T, p[x] )                 ▷ Case 4
43              x ← T                                       ▷ Case 4
44  color[x] ← BLACK
45  return T
```

Si el nodo a borrar y del árbol en `RB-DELETE` es negro, 3 problemas se pueden generar. Primero, si y es la raíz y un hijo rojo de y se vuelve la nueva raíz, entonces se está violado la propiedad 2. Segundo, si x y $p[y]$ (el cual ahora es $p[x]$) son rojos, entonces se está violado la propiedad 4. Tercero, remover a y causa que cualquier camino que previamente haya contenido a y tenga un nodo menos de color negro. Por lo tanto, la propiedad 5 se ha violado por algún ancestro de y en el árbol.

Ejemplo 1:

Para el árbol rojinegro de la Figura 13 borrar el nodo de información 1.

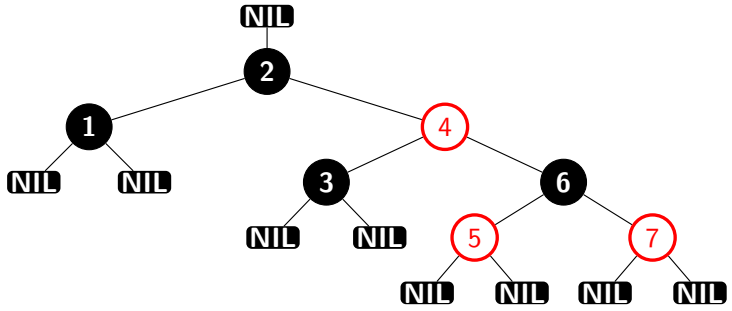

Figura 13: Árbol Rojinegro el cual tiene almacenados los números del 1 al 7.

Después de borrar el número 1 se obtiene el árbol rojinegro de la Figura 14.

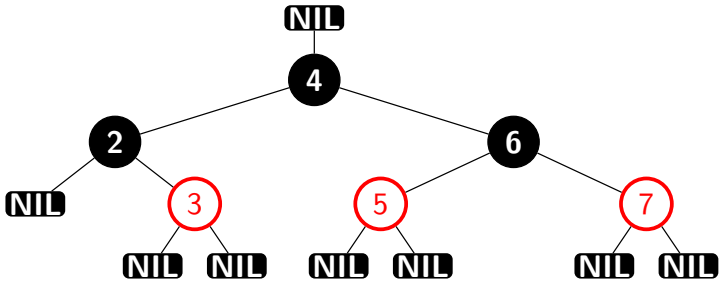

Figura 14: Árbol Rojinegro que se obtiene como resultado después de borrar el número 1.

Ejercicio 2:

Para el árbol rojinegro de la Figura 15 borrar el 2 (nodo que es la raíz del árbol).

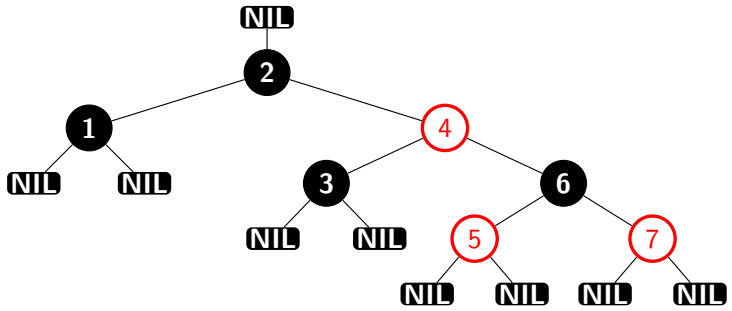

Figura 15: Árbol Rojinegro el cual tiene almacenados los números del 1 al 7.

Ejercicio 3:

Sobre el árbol rojinegro de la Figura 16 borrar el nodo de información 47.

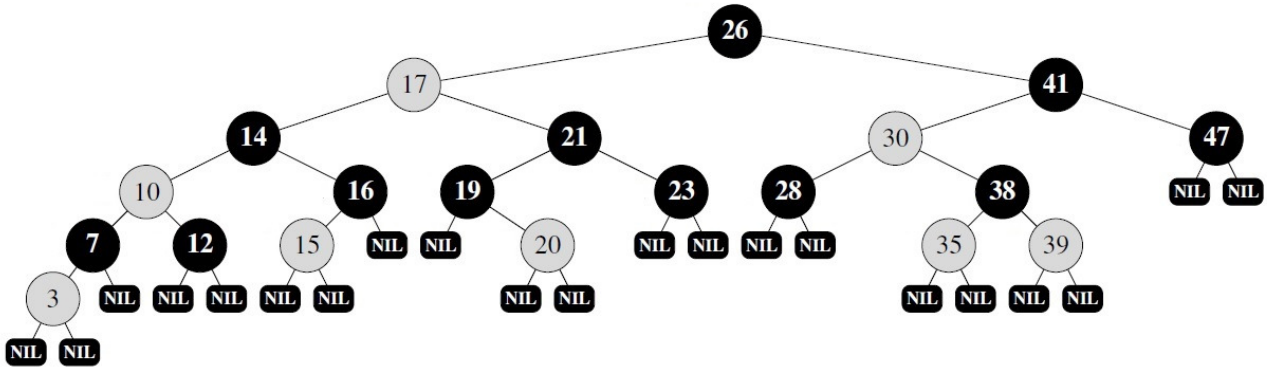

Figura 16: Árbol rojinegro sobre el cual se pide borrar el nodo de información 47.

Ejercicio 4:

Sobre el árbol rojinegro de la Figura 17 borrar el nodo raíz (nodo de información 26).

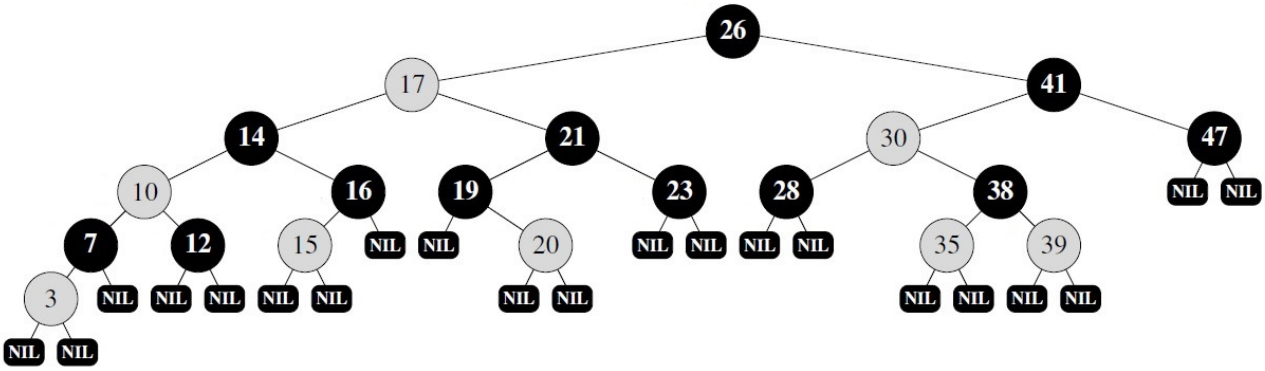

Figura 17: Árbol rojinegro sobre el cual se pide borrar la información del nodo que se encuentra en la raíz.

Implementación Completa de Árboles Rojinegros

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define RED 1
#define BLACK 0
#define NILKey -2147483647

struct nodeRBTree
{
    int key;
    int color;
    struct nodeRBTree *left;
    struct nodeRBTree *right;
    struct nodeRBTree *p;
};

void InorderTreeWalk(struct nodeRBTree *x)
{
    if(x->key != NILKey)
    {
        InorderTreeWalk(x->left);
        if(x->color == BLACK)
            printf("(%d, BLACK) ", x->key);
        else
            printf("(%d, RED) ", x->key);
        InorderTreeWalk(x->right);
    }
}

struct nodeRBTree *TreeSearch(struct nodeRBTree *x, int k)
{
    while((x->key != NILKey) && (k != x->key))
    {
        if(k < x->key)
            x = x->left;
        else
            x = x->right;
    }
```

```
    return x;
}

struct nodeRBTree *TreeMinimum(struct nodeRBTree *x)
{
    while(x->left->key != NILKey)
        x = x->left;

    return x;
}

struct nodeRBTree *TreeMaximum(struct nodeRBTree *x)
{
    while(x->right->key != NILKey)
        x = x->right;

    return x;
}

struct nodeRBTree *TreeSuccessor(struct nodeRBTree *x)
{
    struct nodeRBTree *y;

    if(x->right->key != NILKey)
        return TreeMinimum(x->right);

    y = x->p;
    while((y->key != NILKey) && (x == y->right))
    {
        x = y;
        y = y->p;
    }

    return y;
}

struct nodeRBTree *TreePredecessor(struct nodeRBTree *x)
{
    struct nodeRBTree *y;

    if(x->left->key != NILKey)
        return TreeMaximum(x->left);

    y = x->p;
    while((y->key != NILKey) && (x == y->left))
    {
        x = y;
        y = y->p;
    }

    return y;
}

struct nodeRBTree *LeftRotate(struct nodeRBTree *T, struct nodeRBTree *x)
{
    struct nodeRBTree *y;
    y = x->right;
    x->right = y->left;
    y->left->p = x;
    y->p = x->p;

    if(x->p->key == NILKey)
        T = y;
    else
    {
        if(x == x->p->left)
            x->p->left = y;
```

```
        else
            x->p->right = y;
    }

    y->left = x;
    x->p = y;

    return T;
}

struct nodeRBTree *RightRotate(struct nodeRBTree *T, struct nodeRBTree *x)
{
    struct nodeRBTree *y;
    y = x->left;
    x->left = y->right;
    y->right->p = x;
    y->p = x->p;

    if(x->p->key == NILKey)
        T = y;
    else
    {
        if(x == x->p->right)
            x->p->right = y;
        else
            x->p->left = y;
    }

    y->right = x;
    x->p = y;

    return T;
}

struct nodeRBTree *AssignNilLeaf()
{
    struct nodeRBTree *w;
    w = (struct nodeRBTree *) malloc(sizeof(struct nodeRBTree));
    w->color = BLACK;
    w->key = NILKey;
    w->left = NULL;
    w->right = NULL;

    return w;
}

struct nodeRBTree *RB_InsertFixup(struct nodeRBTree *T, struct nodeRBTree *z)
{
    struct nodeRBTree *y;

    while(z->p->color == RED)
    {
        if(z->p == z->p->p->left)
        {
            y = z->p->p->right;
            if(y->color == RED)
            {
                z->p->color = BLACK;
                y->color = BLACK;
                z->p->p->color = RED;
                z = z->p->p;
            }
            else
            {
                if(z == z->p->right)
                {
                    z = z->p;
                    T = LeftRotate(T, z);
```

```
                }
                z->p->color = BLACK;
                z->p->p->color = RED;
                T = RightRotate(T, z->p->p);
            }
        }
        else
        {
            y = z->p->p->left;
            if(y->color == RED)
            {
                z->p->color = BLACK;
                y->color = BLACK;
                z->p->p->color = RED;
                z = z->p->p;
            }
            else
            {
                if(z == z->p->left)
                {
                    z = z->p;
                    T = RightRotate(T, z);
                }
                z->p->color = BLACK;
                z->p->p->color = RED;
                T = LeftRotate(T, z->p->p);
            }
        }
    }

    T->color = BLACK;

    return T;
}

struct nodeRBTree *RB_Insert(struct nodeRBTree *T, int k)
{
    struct nodeRBTree *x, *y, *z;

    z = (struct nodeRBTree *) malloc(sizeof(struct nodeRBTree));
    z->color = RED;
    z->key = k;
    z->left = AssignNilLeaf();
    z->left->p = z;
    z->right = AssignNilLeaf();
    z->right->p = z;

    if(T->key != NILKey)
        y = T->p;
    else
        y = T;

    x = T;

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

    z->p = y;
    if(y->key == NILKey)
        T = z; /* Empty tree. */
    else
    {
```

```
        free(x);
        if(z->key < y->key)
            y->left = z;
        else
            y->right = z;
    }

    T = RB_InsertFixup(T, z);

    return T;
}

struct nodeRBTree *RB_DeleteFixup(struct nodeRBTree *T, struct nodeRBTree *x)
{
    struct nodeRBTree *w;

    while((x != T) && (x->color == BLACK))
    {
        if(x == x->p->left)
        {
            w = x->p->right;
            if(w->color == RED)
            {
                w->color = BLACK;
                x->p->color = RED;
                T = LeftRotate(T, x->p);
                w = x->p->right;
            }
            if((w->left->color == BLACK) && (w->right->color == BLACK))
            {
                w->color = RED;
                x = x->p;
            }
            else
            {
                if(w->right->color == BLACK)
                {
                    w->left->color = BLACK;
                    w->color = RED;
                    T = RightRotate(T, w);
                    w = x->p->right;
                }
                w->color = x->p->color;
                x->p->color = BLACK;
                w->right->color = BLACK;
                T = LeftRotate(T, x->p);
                x = T;
            }
        }
        else
        {
            w = x->p->left;
            if(w->color == RED)
            {
                w->color = BLACK;
                x->p->color = RED;
                T = RightRotate(T, x->p);
                w = x->p->left;
            }
            if((w->right->color == BLACK) && (w->left->color == BLACK))
            {
                w->color = RED;
                x = x->p;
            }
            else
            {
                if(w->left->color == BLACK)
                {
```

```
                    w->right->color = BLACK;
                    w->color = RED;
                    T = LeftRotate(T, w);
                    w = x->p->left;
                }
                w->color = x->p->color;
                x->p->color = BLACK;
                w->left->color = BLACK;
                T = RightRotate(T, x->p);
                x = T;
            }
        }
    }

    x->color = BLACK;

    return T;
}

struct nodeRBTree *RB_Delete(struct nodeRBTree *T, struct nodeRBTree *z)
{
    struct nodeRBTree *x, *y;

    if((z->left->key == NILKey) || (z->right->key == NILKey))
        y = z;
    else
        y = TreeSuccessor(z);

    if(y->left->key != NILKey)
        x = y->left;
    else
        x = y->right;

    x->p = y->p;

    if(y->p->key == NILKey)
        T = x;
    else
    {
        if(y == y->p->left)
            y->p->left = x;
        else
            y->p->right = x;
    }

    if(y != z)
    {
        z->key = y->key;
        /* Copy all information fields from y to z. */
    }

    if(y->color == BLACK)
        T = RB_DeleteFixup(T, x);

    if(y == y->left->p)
        free(y->left);

    if(y == y->right->p)
        free(y->right);

    free(y);

    return T;
}

int main()
{
    int operation, element;
```

```
    struct nodeRBTree *T, *z;
    T = AssignNilLeaf();

    while(scanf("%d %d", &operation, &element) != EOF)
    {
        if(operation == 1) /* Insert */
        {
            T = RB_Insert(T, element);
            InorderTreeWalk(T);
            printf("\n");
            printf("key[T]: %d\n", T->key);
        }
        else
        {
            if(operation == 2) /* Delete */
            {
                z = TreeSearch(T, element);
                if(z->key == NILKey)
                {
                    printf("The %d is not in the Red-Black Tree\n", element);
                    InorderTreeWalk(T);
                    printf("\n");
                }
                else
                {
                    T = RB_Delete(T, z);
                    InorderTreeWalk(T);
                    printf("\n");

                    if(T->key != NILKey)
                        printf("key[T]: %d\n", T->key);
                }
            }
            else
                printf("Bad use. \n 1 ... Insert \n 2 ... Delete \n");
        }
    }

    return 0;
}
```

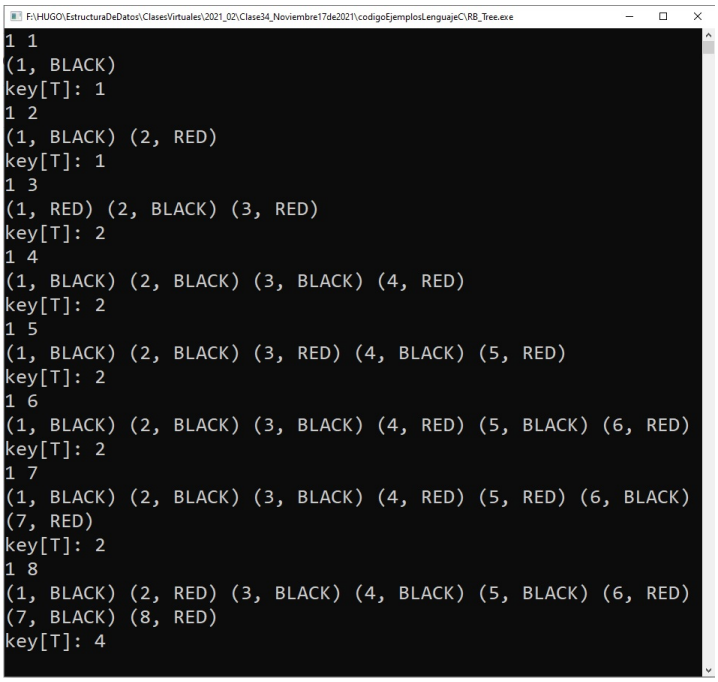

Figura 18: Salida del programa para la inserción de números en un árbol rojinegro.

Programming Challenge: "Add All - Sumar Todos"[1]

Si!, el nombre del problema refleja el trabajo a realizar; es justamente sumar un conjunto de números. Usted puede sentirse motivado a escribir un programa en su lenguaje de programación favorito para sumar todo el conjunto de números. Tal problema debe ser algo muy sencillo para su nivel de programación. Por este motivo le vamos a poner un poco de sabor y vamos a volver el problema mucho más interesante.

Ahora, la operación de suma tiene un costo, y el costo es el resultado de sumar dos números. Por lo tanto, el costo de sumar 1 y 10 es 11. Por ejemplo, si usted quiere sumar los números 1, 2 y 3, hay tres formas de hacerlo:

- Forma 1:
  $1 + 2 = 3$, costo $= 3$
  $3 + 3 = 6$, costo $= 6$
  Costo total $= 9$
- Forma 2:
  $1 + 3 = 4$, costo $= 4$
  $2 + 4 = 6$, costo $= 6$
  Costo total $= 10$
- Forma 3:
  $2 + 3 = 5$, costo $= 5$
  $1 + 5 = 6$, costo $= 6$
  Costo total $= 11$

Yo creo que usted ya ha entendido su misión, sumar un conjunto de números enteros tal que el costo de la operación suma sea mínimo.

Input specification:

La entrada contiene múltiples casos de prueba. Cada caso de prueba debe comenzar con un número entero positivo N ($2 \leq N \leq 5000$), seguido por N números enteros positivos (todos son mayores o iguales a 1 y menores o iguales a 100000). La entrada es finalizada por un caso donde el valor de N es igual a cero. Este caso no debe ser procesado.

Output specification:

Para cada caso de prueba imprimir una sola línea con un número entero positivo que representa el costo total mínimo de la operación suma después de sumar los N números.

Example input:

```
3
1 2 3
4
1 2 3 4
5
5 4 3 2 1
0
```

Example output:

```
9
19
33
```

[1] https://www.hackerrank.com/contests/utp-open-2018/challenges/add-all-2-1

Solución utilizando árboles rojinegros del reto: “Add All”

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define RED 1
#define BLACK 0
#define NILKey -2147483647

struct nodeRBTree
{
    int key;
    int color;
    struct nodeRBTree *left;
    struct nodeRBTree *right;
    struct nodeRBTree *p;
};

void InorderTreeWalk(struct nodeRBTree *x)
{
    if(x->key != NILKey)
    {
        InorderTreeWalk(x->left);
        if(x->color == BLACK)
            printf("(%d, BLACK) ", x->key);
        else
            printf("(%d, RED) ", x->key);
        InorderTreeWalk(x->right);
    }
}

struct nodeRBTree *TreeSearch(struct nodeRBTree *x, int k)
{
    while((x->key != NILKey) && (k != x->key))
    {
        if(k < x->key)
            x = x->left;
        else
            x = x->right;
    }

    return x;
}

struct nodeRBTree *TreeMinimum(struct nodeRBTree *x)
{
    while(x->left->key != NILKey)
        x = x->left;

    return x;
}

struct nodeRBTree *TreeMaximum(struct nodeRBTree *x)
{
    while(x->right->key != NILKey)
        x = x->right;

    return x;
}

struct nodeRBTree *TreeSuccessor(struct nodeRBTree *x)
{
    struct nodeRBTree *y;

    if(x->right->key != NILKey)
        return TreeMinimum(x->right);
```

```
    y = x->p;
    while((y->key != NILKey) && (x == y->right))
    {
        x = y;
        y = y->p;
    }

    return y;
}

struct nodeRBTree *TreePredecessor(struct nodeRBTree *x)
{
    struct nodeRBTree *y;

    if(x->left->key != NILKey)
        return TreeMaximum(x->left);

    y = x->p;
    while((y->key != NILKey) && (x == y->left))
    {
        x = y;
        y = y->p;
    }

    return y;
}

struct nodeRBTree *LeftRotate(struct nodeRBTree *T, struct nodeRBTree *x)
{
    struct nodeRBTree *y;
    y = x->right;
    x->right = y->left;
    y->left->p = x;
    y->p = x->p;

    if(x->p->key == NILKey)
        T = y;
    else
    {
        if(x == x->p->left)
            x->p->left = y;
        else
            x->p->right = y;
    }

    y->left = x;
    x->p = y;

    return T;
}


struct nodeRBTree *RightRotate(struct nodeRBTree *T, struct nodeRBTree *x)
{
    struct nodeRBTree *y;
    y = x->left;
    x->left = y->right;
    y->right->p = x;
    y->p = x->p;

    if(x->p->key == NILKey)
        T = y;
    else
    {
        if(x == x->p->right)
            x->p->right = y;
        else
            x->p->left = y;
```

```
    }

    y->right = x;
    x->p = y;

    return T;
}

struct nodeRBTree *AssignNilLeaf()
{
    struct nodeRBTree *w;
    w = (struct nodeRBTree *) malloc(sizeof(struct nodeRBTree));
    w->color = BLACK;
    w->key = NILKey;
    w->left = NULL;
    w->right = NULL;

    return w;
}

struct nodeRBTree *RB_InsertFixup(struct nodeRBTree *T, struct nodeRBTree *z)
{
    struct nodeRBTree *y;

    while(z->p->color == RED)
    {
        if(z->p == z->p->p->left)
        {
            y = z->p->p->right;
            if(y->color == RED)
            {
                z->p->color = BLACK;
                y->color = BLACK;
                z->p->p->color = RED;
                z = z->p->p;
            }
            else
            {
                if(z == z->p->right)
                {
                    z = z->p;
                    T = LeftRotate(T, z);
                }
                z->p->color = BLACK;
                z->p->p->color = RED;
                T = RightRotate(T, z->p->p);
            }
        }
        else
        {
            y = z->p->p->left;
            if(y->color == RED)
            {
                z->p->color = BLACK;
                y->color = BLACK;
                z->p->p->color = RED;
                z = z->p->p;
            }
            else
            {
                if(z == z->p->left)
                {
                    z = z->p;
                    T = RightRotate(T, z);
                }
                z->p->color = BLACK;
                z->p->p->color = RED;
                T = LeftRotate(T, z->p->p);
```

```
            }
        }
    }

    T->color = BLACK;

    return T;
}

struct nodeRBTree *RB_Insert(struct nodeRBTree *T, int k)
{
    struct nodeRBTree *x, *y, *z;

    z = (struct nodeRBTree *) malloc(sizeof(struct nodeRBTree));
    z->color = RED;
    z->key = k;
    z->left = AssignNilLeaf();
    z->left->p = z;
    z->right = AssignNilLeaf();
    z->right->p = z;

    if(T->key != NILKey)
        y = T->p;
    else
        y = T;

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

    z->p = y;
    if(y->key == NILKey)
        T = z; /* Empty tree. */
    else
    {
        free(x);
        if(z->key < y->key)
            y->left = z;
        else
            y->right = z;
    }

    T = RB_InsertFixup(T, z);

    return T;
}

struct nodeRBTree *RB_DeleteFixup(struct nodeRBTree *T, struct nodeRBTree *x)
{
    struct nodeRBTree *w;

    while((x != T) && (x->color == BLACK))
    {
        if(x == x->p->left)
        {
            w = x->p->right;
            if(w->color == RED)
            {
                w->color = BLACK;
                x->p->color = RED;
                T = LeftRotate(T, x->p);
                w = x->p->right;
```

```
            }
            if((w->left->color == BLACK) && (w->right->color == BLACK))
            {
                w->color = RED;
                x = x->p;
            }
            else
            {
                if(w->right->color == BLACK)
                {
                    w->left->color = BLACK;
                    w->color = RED;
                    T = RightRotate(T, w);
                    w = x->p->right;
                }
                w->color = x->p->color;
                x->p->color = BLACK;
                w->right->color = BLACK;
                T = LeftRotate(T, x->p);
                x = T;
            }
        }
        else
        {
            w = x->p->left;
            if(w->color == RED)
            {
                w->color = BLACK;
                x->p->color = RED;
                T = RightRotate(T, x->p);
                w = x->p->left;
            }
            if((w->right->color == BLACK) && (w->left->color == BLACK))
            {
                w->color = RED;
                x = x->p;
            }
            else
            {
                if(w->left->color == BLACK)
                {
                    w->right->color = BLACK;
                    w->color = RED;
                    T = LeftRotate(T, w);
                    w = x->p->left;
                }
                w->color = x->p->color;
                x->p->color = BLACK;
                w->left->color = BLACK;
                T = RightRotate(T, x->p);
                x = T;
            }
        }
    }

    x->color = BLACK;

    return T;
}

struct nodeRBTree *RB_Delete(struct nodeRBTree *T, struct nodeRBTree *z)
{
    struct nodeRBTree *x, *y;

    if((z->left->key == NILKey) || (z->right->key == NILKey))
        y = z;
    else
        y = TreeSuccessor(z);
```

```
    if(y->left->key != NILKey)
        x = y->left;
    else
        x = y->right;

    x->p = y->p;

    if(y->p->key == NILKey)
        T = x;
    else
    {
        if(y == y->p->left)
            y->p->left = x;
        else
            y->p->right = x;
    }

    if(y != z)
    {
        z->key = y->key;
        /* Copy all information fields from y to z. */
    }

    if(y->color == BLACK)
        T = RB_DeleteFixup(T, x);

    if(y == y->left->p)
        free(y->left);

    if(y == y->right->p)
        free(y->right);

    free(y);

    return T;
}


int main()
{
    int n, element, index;
    long long int result;
    struct nodeRBTree *T, *z;
    T = AssignNilLeaf();

    while(scanf("%d", &n) && (n > 0))
    {
        for(index = 1; index <= n; index++)
        {
            scanf("%d", &element);
            T = RB_Insert(T, element);
        }

        result = 0;

        for(index = 1; index < n; index++)
        {
            z = TreeMinimum(T);
            element = z->key;
            T = RB_Delete(T, z);

            z = TreeMinimum(T);
            element += z->key;
            T = RB_Delete(T, z);

            result += element;
            T = RB_Insert(T, element);
```

```
        }

        T = RB_Delete(T, T);
        printf("%lld\n", result);
    }

    return 0;
}
```

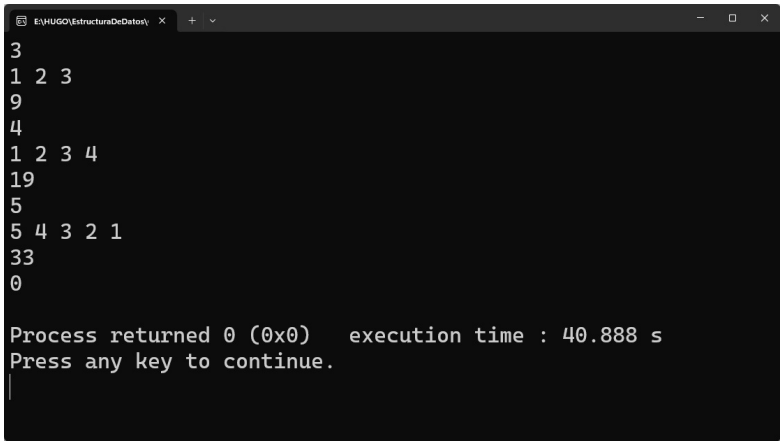

Figura 19: Salida del programa para el reto “Add All”.

Programming Challenge: "Interval"[2]

Let's take a list L containing n ($1 \leq n \leq 10^6$) positive integer numbers in the interval $[1, 10^5]$, and assume you have three operations to work on it: Query, Insert and Delete.

The format for each operation is: p y, p indicates the type of the operation, 1 = Query, 2 = Insert, and 3 = Delete. y represents the value where the operation will be applied.

The **Query** operation must print a pair of numbers x z, such that $x = \text{máx}(\{v \in L \mid v < y\})$, and $z = \text{mín}(\{v \in L \mid y < v\})$

The **Insert** operation adds y to the list.

The **Delete** operation removes one occurrence of the element y in L, if the element is present.

Note 1: For the **Query** and **Delete** the element y might not exist.

Nota 2: There can be several occurrences of y inside the list.

Input specification:

Input begins with an integer t ($1 \leq t \leq 10$), the number of test cases. The first line of the test contains a positive integer n ($1 \leq n \leq 10^6$), with the lenght on the list L. The second line contains exactly n space-separated positive integer numbers L_1, L_2, L_3, ..., L_n ($1 \leq L_i \leq 10^5$, for $1 \leq i \leq n$), that is the list L. The test case continues with a line containing a number q ($1 \leq q \leq 10^5$), the number of operations to apply over the list. In each one of the next q lines two integers p y, ($p \in \{1,\ 2,\ 3\}$, $1 \leq y \leq 10^5$) are presented.

Output specification:

For each query of the type 1, you should print one single line containing the two integers x z. If there is not such value x then $x = -1$, if there is not such value z then $z = 100001$.

Example input:

```
2
5
4 4 4 4 4
8
1 4
1 3
1 5
2 1
2 8
1 4
3 4
1 3
5
10 1 8 3 5
7
1 5
3 5
2 6
1 5
3 8
```

[2] https://www.hackerrank.com/contests/utp-open-2018/challenges/interval-2

```
2 12
1 8
```

Example output:

```
-1 100001
-1 4
4 100001
1 8
1 4
3 8
3 6
6 10
```

Solución utilizando árboles rojinegros del reto: "Interval"

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define RED 1
#define BLACK 0
#define NILKey -2147483647

struct nodeRBTree
{
    int key;
    int color;
    int frecuency;
    struct nodeRBTree *left;
    struct nodeRBTree *right;
    struct nodeRBTree *p;
};

void InorderTreeWalk(struct nodeRBTree *x)
{
    if(x->key != NILKey)
    {
        InorderTreeWalk(x->left);
        if(x->color == BLACK)
            printf("(%d, BLACK) ", x->key);
        else
            printf("(%d, RED) ", x->key);
        InorderTreeWalk(x->right);
    }
}

struct nodeRBTree *TreeSearch(struct nodeRBTree *x, int k)
{
    while((x->key != NILKey) && (k != x->key))
    {
        if(k < x->key)
            x = x->left;
        else
            x = x->right;
    }

    return x;
}

struct nodeRBTree *TreeMinimum(struct nodeRBTree *x)
{
    while(x->left->key != NILKey)
        x = x->left;
```

```
    return x;
}

struct nodeRBTree *TreeMaximum(struct nodeRBTree *x)
{
    while(x->right->key != NILKey)
        x = x->right;

    return x;
}

struct nodeRBTree *TreeSuccessor(struct nodeRBTree *x)
{
    struct nodeRBTree *y;

    if(x->right->key != NILKey)
        return TreeMinimum(x->right);

    y = x->p;
    while((y->key != NILKey) && (x == y->right))
    {
        x = y;
        y = y->p;
    }

    return y;
}

struct nodeRBTree *TreePredecessor(struct nodeRBTree *x)
{
    struct nodeRBTree *y;

    if(x->left->key != NILKey)
        return TreeMaximum(x->left);

    y = x->p;
    while((y->key != NILKey) && (x == y->left))
    {
        x = y;
        y = y->p;
    }

    return y;
}

struct nodeRBTree *LeftRotate(struct nodeRBTree *T, struct nodeRBTree *x)
{
    struct nodeRBTree *y;
    y = x->right;
    x->right = y->left;
    y->left->p = x;
    y->p = x->p;

    if(x->p->key == NILKey)
        T = y;
    else
    {
        if(x == x->p->left)
            x->p->left = y;
        else
            x->p->right = y;
    }

    y->left = x;
    x->p = y;

    return T;
```

```
}

struct nodeRBTree *RightRotate(struct nodeRBTree *T, struct nodeRBTree *x)
{
    struct nodeRBTree *y;
    y = x->left;
    x->left = y->right;
    y->right->p = x;
    y->p = x->p;

    if(x->p->key == NILKey)
        T = y;
    else
    {
        if(x == x->p->right)
            x->p->right = y;
        else
            x->p->left = y;
    }

    y->right = x;
    x->p = y;

    return T;
}

struct nodeRBTree *AssignNilLeaf()
{
    struct nodeRBTree *w;
    w = (struct nodeRBTree *) malloc(sizeof(struct nodeRBTree));
    w->color = BLACK;
    w->key = NILKey;
    w->left = NULL;
    w->right = NULL;

    return w;
}

struct nodeRBTree *RB_InsertFixup(struct nodeRBTree *T, struct nodeRBTree *z)
{
    struct nodeRBTree *y;

    while(z->p->color == RED)
    {
        if(z->p == z->p->p->left)
        {
            y = z->p->p->right;
            if(y->color == RED)
            {
                z->p->color = BLACK;
                y->color = BLACK;
                z->p->p->color = RED;
                z = z->p->p;
            }
            else
            {
                if(z == z->p->right)
                {
                    z = z->p;
                    T = LeftRotate(T, z);
                }
                z->p->color = BLACK;
                z->p->p->color = RED;
                T = RightRotate(T, z->p->p);
            }
        }
        else
        {
```

```
            y = z->p->p->left;
            if(y->color == RED)
            {
                z->p->color = BLACK;
                y->color = BLACK;
                z->p->p->color = RED;
                z = z->p->p;
            }
            else
            {
                if(z == z->p->left)
                {
                    z = z->p;
                    T = RightRotate(T, z);
                }
                z->p->color = BLACK;
                z->p->p->color = RED;
                T = LeftRotate(T, z->p->p);
            }
        }
    }

    T->color = BLACK;

    return T;
}

struct nodeRBTree *RB_Insert(struct nodeRBTree *T, int k)
{
    struct nodeRBTree *x, *y, *z;

    z = (struct nodeRBTree *) malloc(sizeof(struct nodeRBTree));
    z->color = RED;
    z->key = k;
    z->frecuency = 1;
    z->left = AssignNilLeaf();
    z->left->p = z;
    z->right = AssignNilLeaf();
    z->right->p = z;

    if(T->key != NILKey)
        y = T->p;
    else
        y = T;

    x = T;

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

    z->p = y;
    if(y->key == NILKey)
        T = z; /* Empty tree. */
    else
    {
        free(x);
        if(z->key < y->key)
            y->left = z;
        else
            y->right = z;
    }
```

```
    T = RB_InsertFixup(T, z);

    return T;
}

struct nodeRBTree *RB_DeleteFixup(struct nodeRBTree *T, struct nodeRBTree *x)
{
    struct nodeRBTree *w;

    while((x != T) && (x->color == BLACK))
    {
        if(x == x->p->left)
        {
            w = x->p->right;
            if(w->color == RED)
            {
                w->color = BLACK;
                x->p->color = RED;
                T = LeftRotate(T, x->p);
                w = x->p->right;
            }
            if((w->left->color == BLACK) && (w->right->color == BLACK))
            {
                w->color = RED;
                x = x->p;
            }
            else
            {
                if(w->right->color == BLACK)
                {
                    w->left->color = BLACK;
                    w->color = RED;
                    T = RightRotate(T, w);
                    w = x->p->right;
                }
                w->color = x->p->color;
                x->p->color = BLACK;
                w->right->color = BLACK;
                T = LeftRotate(T, x->p);
                x = T;
            }
        }
        else
        {
            w = x->p->left;
            if(w->color == RED)
            {
                w->color = BLACK;
                x->p->color = RED;
                T = RightRotate(T, x->p);
                w = x->p->left;
            }
            if((w->right->color == BLACK) && (w->left->color == BLACK))
            {
                w->color = RED;
                x = x->p;
            }
            else
            {
                if(w->left->color == BLACK)
                {
                    w->right->color = BLACK;
                    w->color = RED;
                    T = LeftRotate(T, w);
                    w = x->p->left;
                }
                w->color = x->p->color;
                x->p->color = BLACK;
```

```
                w->left->color = BLACK;
                T = RightRotate(T, x->p);
                x = T;
            }
        }
    }

    x->color = BLACK;

    return T;
}

struct nodeRBTree *RB_Delete(struct nodeRBTree *T, struct nodeRBTree *z)
{
    struct nodeRBTree *x, *y;

    if((z->left->key == NILKey) || (z->right->key == NILKey))
        y = z;
    else
        y = TreeSuccessor(z);

    if(y->left->key != NILKey)
        x = y->left;
    else
        x = y->right;

    x->p = y->p;

    if(y->p->key == NILKey)
        T = x;
    else
    {
        if(y == y->p->left)
            y->p->left = x;
        else
            y->p->right = x;
    }

    if(y != z)
    {
        z->key = y->key;
        z->frecuency = y->frecuency;
        /* Copy all information fields from y to z. */
    }

    if(y->color == BLACK)
        T = RB_DeleteFixup(T, x);

    if(y == y->left->p)
        free(y->left);

    if(y == y->right->p)
        free(y->right);

    free(y);

    return T;
}

int main()
{
    int totalCases, idCase, n, idElement, element, q, idQuery, p, y;
    struct nodeRBTree *T, *z, *left, *right;

    scanf("%d", &totalCases);
    for(idCase = 1; idCase <= totalCases; idCase++)
    {
        T = AssignNilLeaf();
```

```
        scanf("%d", &n);
        for(idElement = 1; idElement <= n; idElement++)
        {
            scanf("%d", &element);
            z = TreeSearch(T, element);
            if(z->key == NILKey)
                T = RB_Insert(T, element);
            else
                z->frecuency++;
        }

        T = RB_Insert(T, -1);
        T = RB_Insert(T, 100001);
        scanf("%d", &q);

        for(idQuery = 1; idQuery <= q; idQuery++)
        {
            scanf("%d %d", &p, &y);
            if(p == 1)
            {
                z = TreeSearch(T, y);
                if(z->key != NILKey)
                {
                    left = TreePredecessor(z);
                    right = TreeSuccessor(z);
                    printf("%d %d\n", left->key, right->key);
                }
                else
                {
                    T = RB_Insert(T, y);
                    z = TreeSearch(T, y);
                    left = TreePredecessor(z);
                    right = TreeSuccessor(z);
                    printf("%d %d\n", left->key, right->key);
                    T = RB_Delete(T, z);
                }
            }
            else
            {
                if(p == 2)
                {
                    z = TreeSearch(T, y);
                    if(z->key == NILKey)
                        T = RB_Insert(T, y);
                    else
                        z->frecuency++;
                }
                else
                {
                    if(p == 3)
                    {
                        z = TreeSearch(T, y);
                        if(z->key != NILKey)
                        {
                            if(z->frecuency == 1)
                                T = RB_Delete(T, z);
                            else
                                z->frecuency--;
                        }
                    }
                }
            }
        }
    }

    return 0;
}
```

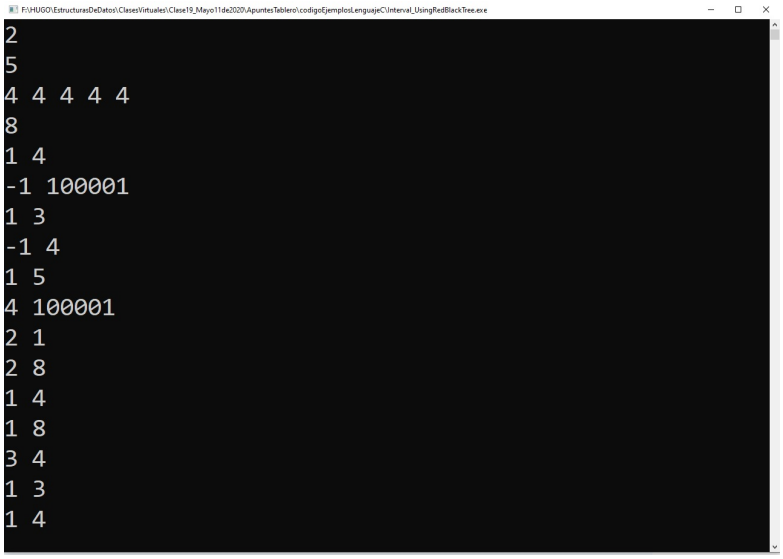

Figura 20: Salida del programa para el reto “Interval”.

Reto de Programación: "How Many Sub Sets?"[3]

Se tiene un conjunto A de números enteros positivos con cardinalidad N ($|A| = N$, recordar que la cardinalidad de un conjunto es el total de elementos distintos que este tiene). Se quiere averiguar cuántos subconjuntos hay de tamaño 2 que tengan una suma de sus elementos menor o igual a S.

Por ejemplo, si se tiene el siguiente conjunto $A = \{6,\ 5,\ 1,\ 4,\ 2,\ 3\}$ y $S = 7$, entonces hay un total de 9 subconjuntos de tamaño dos cuya suma de sus elementos es menor o igual a 7, dichos subconjuntos son: $\{6,\ 1\}$, $\{5,\ 1\}$, $\{5,\ 2\}$, $\{1,\ 4\}$, $\{1,\ 2\}$, $\{1,\ 3\}$, $\{4,\ 2\}$, $\{4,\ 3\}$ y $\{2,\ 3\}$.

Su misión, si decide aceptarla es el de contar el total de subconjuntos de tamaño dos que tienen una suma de sus elementos menor o igual a S.

Entrada:

La entrada del problema consiste de un único caso de prueba. El caso de prueba contiene tres líneas, la primer línea contiene dos números enteros positivos n ($1 \leq n \leq 5 \cdot 10^5$) y q ($1 \leq q \leq 50$), que representan respectivamente la cardinalidad del conjunto A y el total de consultas que se van a realizar sobre el conjunto A. La siguiente línea contiene exactamente n números enteros positivos (separados por espacio en blanco) A_1, A_2, A_3, ..., A_n ($1 \leq A_i \leq 10^8$, para $1 \leq i \leq n$), obviamente se garantiza que los n elementos del conjunto A son diferentes. La siguiente línea contiene exactamente q números enteros positivos (separados por espacio en blanco) S_1, S_2, S_3, ..., S_q ($1 \leq S_j \leq 2 \cdot 10^8$, para $1 \leq j \leq q$), para las consultas.

Salida:

Su programa debe imprimir q líneas (una línea de respuesta por consulta), cada una de ellas conteniendo un único valor que representa el resultado total de subconjuntos de tamaño dos que la suma de sus elementos es menor o igual a S_j.

Ejemplo entrada:

```
6 3
6 5 1 4 2 3
7 8 12
```

Ejemplo salida:

```
9
11
15
```

Solución utilizando árboles rojinegros del reto: "How Many Sub Sets?"

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define BLACK 0
#define RED 1
#define NILKey -2147483647

struct nodeRBTree
{
    int key;
    int color;
    int position;
    struct nodeRBTree *p;
    struct nodeRBTree *left;
```

[3] https://www.hackerrank.com/contests/utp-open-2018/challenges/how-many-sub-sets

```
    struct nodeRBTree *right;
};

int positionInOrder = 1;

void InorderTreeWalk(struct nodeRBTree *x)
{
    if(x->key != NILKey)
    {
        InorderTreeWalk(x->left);
        if(x->color == BLACK)
            printf("(%d, %d, BLACK) ", x->position, x->key);
        else
            printf("(%d, RED) ", x->position, x->key);
        InorderTreeWalk(x->right);
    }
}

void AssignPostions(struct nodeRBTree *x)
{
    if(x->key != NILKey)
    {
        AssignPostions(x->left);
        x->position = positionInOrder;
        positionInOrder++;
        AssignPostions(x->right);
    }
}

struct nodeRBTree *TreeSearch(struct nodeRBTree *x, int k)
{
    if((x->key == NILKey) || (k == x->key))
        return x;
    else
    {
        if(k < x->key)
            return TreeSearch(x->left, k);
        else
            return TreeSearch(x->right, k);
    }
}

struct nodeRBTree *TreeMinimum(struct nodeRBTree *x)
{
    while(x->left->key != NILKey)
        x = x->left;

    return x;
}

struct nodeRBTree *TreeMaximum(struct nodeRBTree *x)
{
    while(x->right->key != NILKey)
        x = x->right;

    return x;
}

struct nodeRBTree *TreeSuccessor(struct nodeRBTree *x)
{
    struct nodeRBTree *y;

    if(x->right->key != NILKey)
        return TreeMinimum(x->right);
    else
    {
        y = x->p;
        while((y != NULL) && (x == y->right))
```

```
        {
            x = y;
            y = y->p;
        }
        return y;
    }
}

struct nodeRBTree *TreePredecessor(struct nodeRBTree *x)
{
    struct nodeRBTree *y;

    if(x->left->key != NILKey)
        return TreeMaximum(x->left);
    else
    {
        y = x->p;
        while((y != NULL) && (x == y->left))
        {
            x = y;
            y = y->p;
        }
        return y;
    }
}

struct nodeRBTree *AssignNilLeaf()
{
    struct nodeRBTree *z;

    z = (struct nodeRBTree *) malloc(sizeof(struct nodeRBTree));

    z->color = BLACK;
    z->key = NILKey;
    z->left = NULL;
    z->right = NULL;

    return z;
}

struct nodeRBTree *LeftRotate(struct nodeRBTree *T, struct nodeRBTree *x)
{
    struct nodeRBTree *y;
    y = x->right;
    x->right = y->left;
    y->left->p = x;
    y->p = x->p;
    if(x->p->key == NILKey)
        T = y;
    else
    {
        if(x == x->p->left)
            x->p->left = y;
        else
            x->p->right = y;
    }
    y->left = x;
    x->p = y;
    return T;
}

struct nodeRBTree *RightRotate(struct nodeRBTree *T, struct nodeRBTree *x)
{
    struct nodeRBTree *y;
    y = x->left;
    x->left = y->right;
    y->right->p = x;
    y->p = x->p;
```

```
    if(x->p->key == NILKey)
        T = y;
    else
    {
        if(x == x->p->right)
            x->p->right = y;
        else
            x->p->left = y;
    }
    y->right = x;
    x->p = y;
    return T;
}

struct nodeRBTree *RB_InsertFixup(struct nodeRBTree *T, struct nodeRBTree *z)
{
    struct nodeRBTree *y;

    while(z->p->color == RED)
    {
        if(z->p == z->p->p->left)
        {
            y = z->p->p->right;
            if(y->color == RED)
            {
                z->p->color = BLACK;
                y->color = BLACK;
                z->p->p->color = RED;
                z = z->p->p;
            }
            else
            {
                if(z == z->p->right)
                {
                    z = z->p;
                    T = LeftRotate(T, z);
                }
                z->p->color = BLACK;
                z->p->p->color = RED;
                T = RightRotate(T, z->p->p);
            }
        }

        else
        {
            y = z->p->p->left;
            if(y->color == RED)
            {
                z->p->color = BLACK;
                y->color = BLACK;
                z->p->p->color = RED;
                z = z->p->p;
            }
            else
            {
                if(z == z->p->left)
                {
                    z = z->p;
                    T = RightRotate(T, z);
                }
                z->p->color = BLACK;
                z->p->p->color = RED;
                T = LeftRotate(T, z->p->p);
            }
        }
    }

    T->color = BLACK;
```

```
    return T;
}

struct nodeRBTree *RB_TreeInsert(struct nodeRBTree *T, int k)
{
    struct nodeRBTree *x, *y, *z;

    z = (struct nodeRBTree *) malloc(sizeof(struct nodeRBTree));

    z->color = RED;
    z->key = k;
    z->left = AssignNilLeaf();
    z->left->p = z;
    z->right = AssignNilLeaf();
    z->right->p = z;

    if(T->key != NILKey)
        y = T->p;
    else
        y = T;

    x = T;

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

    z->p = y;
    if(y->key == NILKey)
        T = z;
    else
    {
        free(x);
        if(z->key < y->key)
            y->left = z;
        else
            y->right = z;
    }

    T = RB_InsertFixup(T, z);
    return T;
}

struct nodeRBTree *RB_DeleteFixup(struct nodeRBTree *T, struct nodeRBTree *x)
{
    struct nodeRBTree *w;

    while((x != T) && (x->color == BLACK))
    {
        if(x == x->p->left)
        {
            w = x->p->right;
            if(w->color == RED)
            {
                w->color = BLACK;
                x->p->color = RED;
                T = LeftRotate(T, x->p);
                w = x->p->right;
            }
            if((w->left->color == BLACK) && (w->right->color == BLACK))
            {
                w->color = RED;
                x = x->p;
```

```
            }
            else
            {
                if(w->right->color == BLACK)
                {
                    w->left->color = BLACK;
                    w->color = RED;
                    T = RightRotate(T, w);
                    w = x->p->right;
                }
                w->color = x->p->color;
                x->p->color = BLACK;
                w->right->color =BLACK;
                T = LeftRotate(T, x->p);
                x = T;
            }
        }
        else
        {
            w = x->p->left;
            if(w->color == RED)
            {
                w->color = BLACK;
                x->p->color = RED;
                T = RightRotate(T, x->p);
                w = x->p->left;
            }
            if((w->left->color == BLACK) && (w->right->color == BLACK))
            {
                w->color = RED;
                x = x->p;
            }
            else
            {
                if(w->left->color == BLACK)
                {
                    w->right->color = BLACK;
                    w->color = RED;
                    T = LeftRotate(T, w);
                    w = x->p->left;
                }
                w->color = x->p->color;
                x->p->color = BLACK;
                w->left->color =BLACK;
                T = RightRotate(T, x->p);
                x = T;
            }
        }
    }
    x->color = BLACK;
    return T;
}

struct nodeRBTree *RB_TreeDelete(struct nodeRBTree *T, struct nodeRBTree *z)
{
    struct nodeRBTree *x, *y;
    if((z->left->key == NILKey) || (z->right->key == NILKey))
        y = z;
    else
        y = TreeSuccessor(z);

    if(y->left->key != NILKey)
        x = y->left;
    else
        x = y->right;

    x->p = y->p;
    if(y->p->key == NILKey)
```

```
        T = x;
    else
    {
        if(y == y->p->left)
            y->p->left = x;
        else
            y->p->right = x;
    }

    if(y != z)
    {
        z->key = y->key;
        // Copy all information fields from y to z.
    }

    if(y->color == BLACK)
        T = RB_DeleteFixup(T, x);

    if(y == y->left->p)
        free(y->left);

    if(y == y->right->p)
        free(y->right);

    free(y);
    return T;
}

int main()
{
    int n, q, s, element, index, idQuery, element2;
    long long int result;
    struct nodeRBTree *T, *actualNode, *maximumNode, *z;
    T = AssignNilLeaf();

    scanf("%d %d", &n, &q);
    for(index = 1; index <= n; index++)
    {
        scanf("%d", &element);
        T = RB_TreeInsert(T, element);
    }

    AssignPostions(T);

    for(idQuery = 1; idQuery <= q; idQuery++)
    {
        result = 0;
        scanf("%d", &s);

        actualNode = TreeMinimum(T);
        maximumNode = TreeMaximum(T);

        while(actualNode != maximumNode)
        {
            element2 = s - actualNode->key;
            if(element2 > actualNode->key)
            {
                z = TreeSearch(T, element2);
                if(z->key == NILKey)
                {
                    if(z == z->p->left)
                        index = z->p->position - 1;
                    else
                        index = z->p->position;
                }
                else
                    index = z->position;
                result += index - actualNode->position;
```

```
            }
            actualNode = TreeSuccessor(actualNode);
        }
        printf("%lld\n", result);
    }
    return 0;
}
```