



# ESTRUCTURA DE DATOS: RANGE TREE. LAZY PROGATION

**Autores:**

Colectivo de Entrenadores ACM-ICPC UM

Marzo / 2019

ENTRENAMIENTO PARA CONCURSANTES  
ACM-ICPC DE LA UNIVERSIDAD DE  
MATANZAS

# Objetivos

- ▶ Caracterizar la estructura de datos *Range Tree*.
- ▶ Caracterizar la estructura de datos *Range Tree + Lazy Propagation*.
- ▶ Implementar estructura de datos *Range Tree*.

# Objetivos

- ▶ Implementar estructura de datos *Range Tree + Lazy Propagation*.
- ▶ Identificar problemas donde la solución sea aplicando la estructura de datos *Range Tree*.
- ▶ Identificar problemas donde la solución sea aplicando la estructura de datos *Range Tree + Lazy Propagation*.

# Bibliografía

- ▶ *Manual de preparación para concursantes ACM-ICPC de la Universidad de Matanzas.* Publicado en el Entorno Virtual de Aprendizaje de la Universidad de Matanzas ( [eva.umcc.cu](http://eva.umcc.cu) ). Sección *Bibliografía*.

# Problema a resolver

Dado una colección de elementos de hasta  $10^5$  elementos sobre la cual se realiza dos operaciones básicas:

- ▶ Actualización: Dado un índice y un valor modificar el valor del elemento en el índice especificado por el nuevo valor.

# Problema a resolver

## Continuación ...

Dado una colección de elementos de hasta  $10^5$  elementos sobre la cual se realiza dos operaciones básicas:

- ▶ Consulta: Dado un rango definido por dos índices ver en ese rango el resultado de una determinado operación (mínimo, máximo, suma, ect).

# Problema a resolver

## Continuación ...

La cantidad de operaciones que se realizan sobre la colección es no supera  $10^5$  operaciones.

[6, 15, 17, 21, 24, 23, 42,  
51, 52, 57, 65, 73, 78]

# *Range Tree*

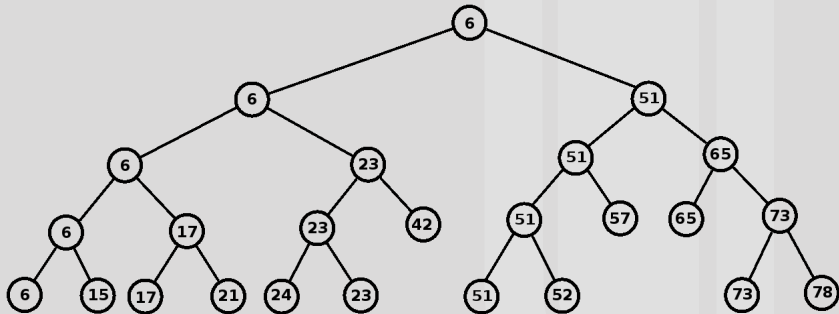
Un árbol de rango *Range Tree* en una colección de elementos de una dimensión es un árbol de búsqueda binario equilibrado en esos elementos.



# *Range Tree*

Los elementos de la colección son almacenados en el árbol en las hojas del árbol; cada nodo interno almacena el valor la respuesta de la operación de consulta en sus subárboles.

# Range Tree



# *Range Tree*

## Operaciones

- ▶ Construcción  $O(N \log^d N)$
- ▶ Actualización  $O(\log N)$
- ▶ Consulta  $O(\log N)$

# *Range Tree*

## Implementación: Variables

```
#define MID (left+right)/2
#define MAX_N 1000001
#define MAX_TREE (MAX_N << 2)
#define INF 987654321
using namespace std;
typedef long long lld;

int n;
int niz[MAX_N];
int RT[MAX_TREE];
```

# *Range Tree*

## Implementación: Construcción

```
void InitTree(int idx, int left, int right){  
    if(left == right){  
        RT[idx] = niz[left];  
        return;  
    }  
    InitTree(2*idx, left, MID);  
    InitTree(2*idx+1, MID+1, right);  
    RT[idx] = min(RT[2*idx], RT[2*idx+1]);  
}
```

# *Range Tree*

## Implementación: Actualización

```
void Update(int idx, int x, int val, int left, int
    right){
    if(left == right){
        RT[idx] = val;
        return;
    }
    if(x <= MID)Update(2*idx,x,val,left,MID);
    else Update(2*idx+1,x,val,MID+1,right);
    RT[idx]=min(RT[2*idx],RT[2*idx+1]);
}
```

# *Range Tree*

## Implementación: Consulta

```
int Query(int idx, int l, int r, int left, int
right){
    if(l<=left && right<=r) return RT[idx];
    int ret = INF;
    if(l<=MID)
        ret=min(ret,Query(2*idx,l,r,left,MID));
    if(r>MID)
        ret=min(ret,Query(2*idx+1,l,r,MID+1, right));
    return ret;
}
```

# *Range Tree*

## Implementación: Uso

```
int main() {  
    n = 6;  
    niz[1] = 4;niz[2] = 2;niz[3] = 5;  
    niz[4] = 1;niz[5] = 6;niz[6] = 3;  
  
    InitTree(1, 1, n);  
    printf("%d\n",Query(1, 1, 3, 1, n));  
  
    Update(1, 4, 10, 1, n);  
    Update(1, 5, 0, 1, n);  
    printf("%d\n",Query(1, 4, 6, 1, n));  
  
    return 0;  
}
```



# *Range Tree*

## Problemas

- ▶ 2249 - Curious Robin Hood
- ▶ 3632 - Toby and Query
- ▶ 3484 - Work for Ten
- ▶ 3954 - The Evarista Store

# Problema a resolver

## Nueva variante

Hagamos una pequeña modificación al problema inicial en la operación de actualización:

- ▶ Actualización: Dado un rango de índices y un valor, modificar todos los elementos comprendidos en el rango con el nuevo valor.

# Problema a resolver

## Nueva variante

Por ejemplo, modifique por 10 a todos los valores en los índices de 2 a 7 en la colección.

Una solución pudiera ser invocarse la función de actualización para cada número de 2 a 7. Esto puede ser una solución pero no es óptima.

# Problema a resolver

## Nueva variante

Entonces esta vez nuestro problema radica en como realizar una función que permita actualizar en un rango dentro de la colección con un tiempo  $\log N$ .

# *Lazy Proagation*

**Propagación diferida o perezosa (Lazy proagation):** Es una optimización para hacer que las actualizaciones de rango sean más rápidas.

# *Lazy Proagation*

Cuando hay muchas actualizaciones en un rango, podemos posponer algunas actualizaciones y hacer esas actualizaciones solo cuando sea necesario.

# *Lazy Propagation*

Recuerde que un nodo en el árbol de rangos almacena o representa el resultado de una consulta para un rango de índices. Y si el rango de este nodo se encuentra dentro del rango de operación de actualización, todos los descendientes del nodo también deben actualizarse.

# *Lazy Propagation*

La idea es inicializar todos los elementos de *lazy* [] con valor  $X$ . Un valor  $X$  en *lazy* [ $i$ ] indica que no hay actualizaciones pendientes en el nodo  $i$  en el árbol de rangos.



# *Lazy Propagation*

Un valor distinto de  $X$  en `lazy[i]` significa que esta cantidad debe agregarse al nodo  $i$  en el árbol de rangos antes de realizar cualquier consulta al nodo.

# *Lazy Proagation*

¿Hay algún cambio en la función de consulta también?

# *Lazy Proagation*

Como hemos cambiado la actualización para posponer sus operaciones, puede haber problemas si se realiza una consulta a un nodo que aún no se ha actualizado. Por lo tanto, debemos actualizar nuestro método de consulta.

# *Lazy Proagation*

En el método de consulta ahora primero comprueba si hay una actualización pendiente y si la hay, luego actualiza el nodo. Una vez que se asegure de que la actualización pendiente esté completa, funciona igual que el método de consulta de un *Range Tree* sin *Lazy proagation*.

# *Lazy Proagation*

## Implementación: Variables

```
#define MID (left+right)/2
#define MAX_N 1000001
#define MAX_TREE (MAX_N << 2)
#define INF 987654321
#define LL unsigned long long
using namespace std;
typedef long long lld;

int c;
int tree[MAX_TREE];
int lazy[MAX_TREE];
```

# *Lazy Proagation*

## Implementación: Construcción

```
void buildRangeTree(int treeIndex, int left, int
    right){
    if(left == right){
        tree[treeIndex] = 0;
        lazy[treeIndex]=-1;
        return;
    }
    buildSegTree( 2 * treeIndex , left, MID);
    buildSegTree( 2 * treeIndex+1, MID + 1, right);
    tree[treeIndex]=tree[2*treeIndex+1]+tree[2*
        treeIndex+2];
    lazy[treeIndex]=-1;
}
```

# *Lazy Propagation*

## Implementación: Actualización

```
void updateLazyRangeTree(int treeIndex, int left, int
    right, int i, int j, int val){
    if (lazy[treeIndex] != -1) {
        tree[treeIndex] = (right - left + 1) * lazy[
            treeIndex];
        if (left != right){
            lazy[2 * treeIndex] = lazy[treeIndex];
            lazy[2 * treeIndex + 1] = lazy[treeIndex];
        }
        lazy[treeIndex] = -1;
    }
    ...
}
```

# *Lazy Proagation*

## Implementación: Actualización

```
...
if(left>right || left>j || right<i)
    return;
if(i<=left && right<=j){
    tree[treeIndex]=(right-left+1)*val;
    if(left!=right){
        lazy[2*treeIndex]=val;
        lazy[2*treeIndex+1]=val;
    }
    return;
}
...

```



# *Lazy Propagation*

## Implementación: Actualización

```
...  
updateLazyRangeTree(2*treeIndex, left, MID, i, j,  
    val);  
updateLazyRangeTree(2*treeIndex+1, MID+1, right, i  
    , j, val);  
tree[treeIndex]=tree[2*treeIndex]+tree[2*  
    treeIndex+1];  
}
```

# *Lazy Propagation*

## Implementación: Consulta

```
int queryLazyRangeTree(int treeIndex, int left, int
    right, int i, int j){
    if(left>j || right<i)
        return 0;
    if(lazy[treeIndex]!=-1){
        tree[treeIndex]=
            (right-left+1)*lazy[treeIndex];
        if(left!=right){
            lazy[2*treeIndex]=lazy[treeIndex];
            lazy[2 * treeIndex + 1] = lazy[treeIndex];
        }
        lazy[treeIndex]=-1;
    }
    ...
}
```

# *Lazy Propagation*

## Implementación: Consulta

```
...  
if (i <= left && j >= right)  
    return tree[treeIndex];  
if (i > MID)  
    return queryLazyRangeTree(2*treeIndex+1, MID  
        +1, right, i, j);  
else if (j <= MID)  
    return queryLazyRangeTree(2*treeIndex, left,  
        MID, i, j);  
...
```

# *Lazy Propagation*

## Implementación: Consulta

```
...  
int leftQuery=queryLazyRangeTree(2*treeIndex ,  
    left,MID,i,MID);  
int rightQuery=queryLazyRangeTree(2*treeIndex  
    +1,MID+1,right,MID+1,j);  
return leftQuery + rightQuery;  
}
```

# *Lazy Proagation*

## Problemas

- ▶ 3200 - High Frequency

# Conclusiones

Los problemas donde se pudiera aplicar las estructuras estudiadas son sencillos de identificar. Existen pistas como:

- ▶ Colección de elementos de hasta  $10^5$  elementos.
- ▶ Dos operaciones se realizan sobre la colección.
- ▶ El total de operaciones puede llegar hasta  $10^5$  operaciones.

# UNIVERSIDAD DE MATANZAS

cosechando el saber

FIN