Lowest Common Ancestor

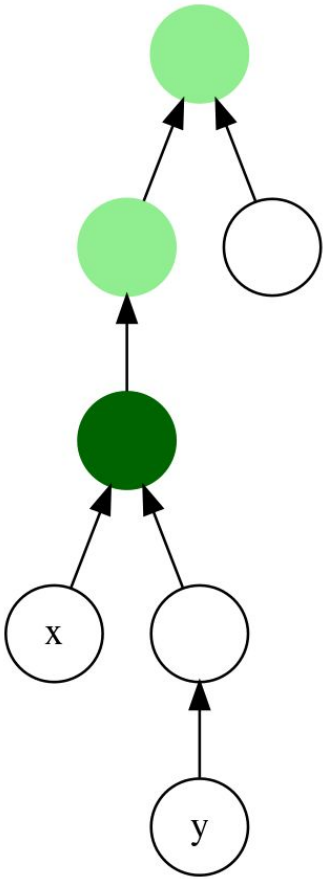
Jeisson Yato Tintaya

¿De qué trata?

Hay que buscar el ancestro común de dos nodos que se encuentra más alejado de la raíz del árbol.

Verde oscuro: El ancestro común más bajo(LCA) de **x** e **y**.

Verde claro: Otros ancestros comunes de **x** e **y**.



Historia

LCA fue definido por Alfred Aho, Jonh Hopcroft y Jeffry Ullman en 1973.

Los primeros en desarrollar una estructura de datos óptima y eficiente para encontrar el ancestro común más bajo fueron Harel Dov y Robert Tarjan. Su algoritmo procesa cualquier árbol en tiempo lineal, usando una descomposición de caminos fuerte, así las preguntas subsiguientes por el ancestro común más bajo pueden ser respondidas en tiempo constante por pregunta. Sin embargo, su estructura de datos es compleja y difícil de implementar.

En 1988 Baruch Schieber y Uzi Vishkin simplificaron la estructura de datos de Harel y Tarjan, consiguiendo una estructura implementable con el mismo preprocesamiento asintótico y rangos de tiempo por query.

En 1993 Omer Berkman y Uzi Vishkin descubrieron una forma completamente nueva para responder preguntas sobre el ancestro común más bajo(LCA), logrando un nuevo preprocesamiento en tiempo lineal con queries en tiempo constante.

Michael Bender y Martin Farach-Colton en el 2000 verificaron el problema de mínimo valor en un rango(RMQ) combinando dos técnicas, una técnica basada en precalcular las querys en intervalos largos que tienen tamaño potencias de dos, y la otra basada en una tabla para buscar en intervalos pequeños.

Otras simplificaciones fueron hechas por Alstrup, Gavoille, Kaplan y Rauhe en 2004 y Fischer y Heun en 2006.

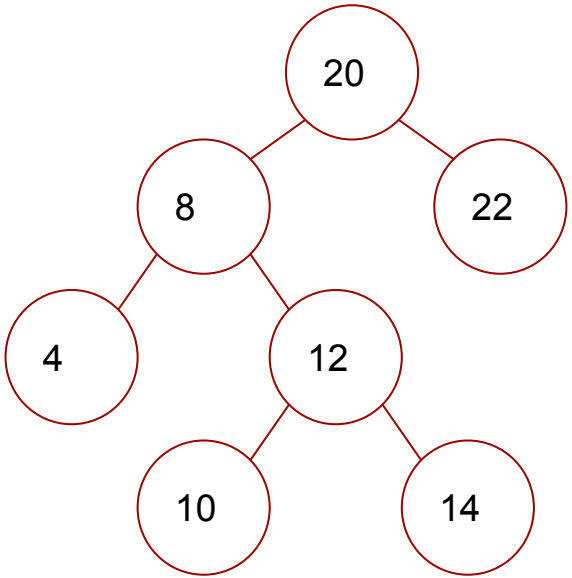
Soluciones

1. Naive
2. SQRT Decomposition
3. Usando Range Minimum Query (RMQ) + Segment Tree

4. Usando Range Minimum Query (RMQ) + Sparse Table

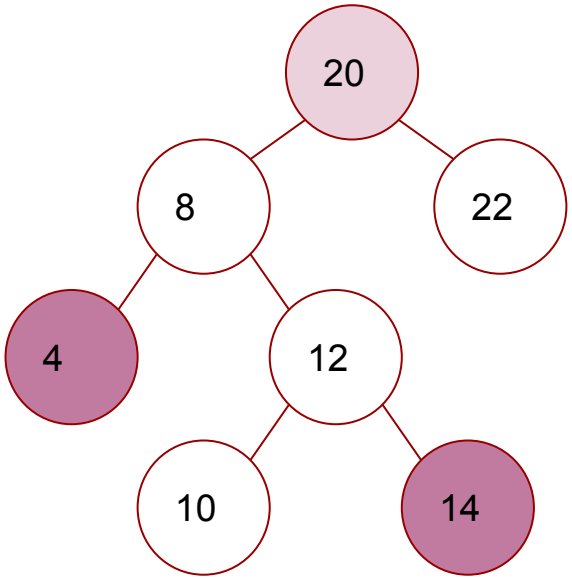
Árbol Binario Ordenado

Complejidad: O(h) h: altura



Árbol Binario Ordenado

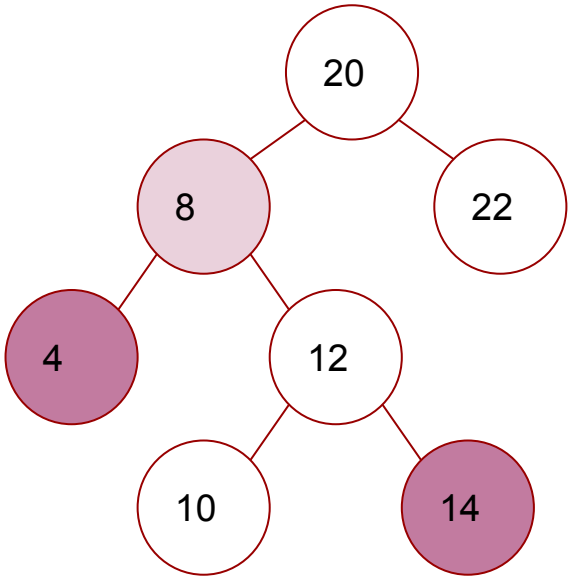
```
struct node *lca(struct node* root, int n1, int n2)
{
    while (root != NULL)
    {
        if(root->data > n1 && root->data > n2){
            root = root->left;
        }
        else if(root->data < n1 && root->data < n2){
            root = root->right;
        }
        else{break;}
    }
    return root;
}
```



n1 = 4 n2= 14

Árbol Binario Ordenado

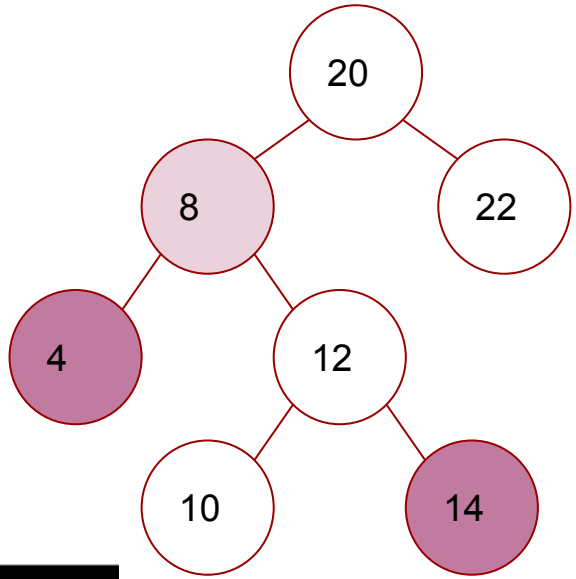
```
struct node *lca(struct node* root, int n1, int n2)
{
    while (root != NULL)
    {
        if(root->data > n1 && root->data > n2){
            root = root->left;
        }
        else if(root->data < n1 && root->data < n2){
            root = root->right;
        }
        else{break;}
    }
    return root;
}
```



n1 = 4 n2= 14

Árbol Binario Ordenado

```
struct node *lca(struct node* root, int n1, int n2)
{
    while (root != NULL)
    {
        if(root->data > n1 && root->data > n2){
            root = root->left;
        }
        else if(root->data < n1 && root->data < n2){
            root = root->right;
        }
        else{break;}
    }
    return root;
}
```

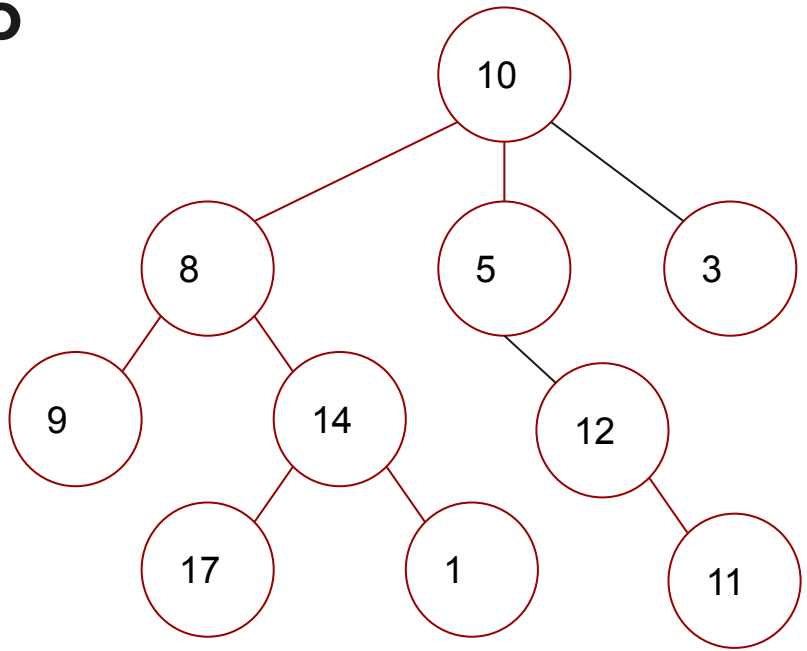


n1 = 4 n2= 14

```
LCA de 4 y 14 es 8
```

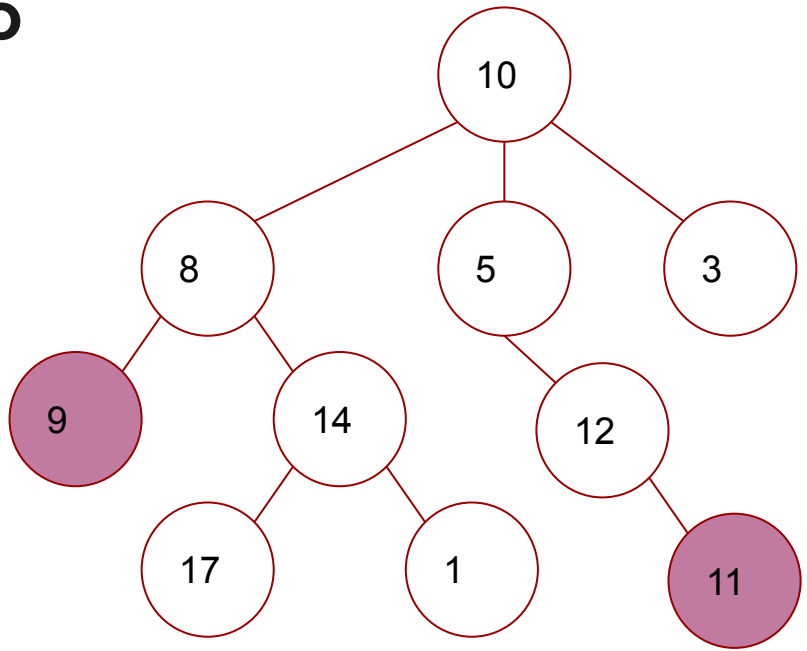
Árbol Binario no Ordenado

Complejidad: O(h) h: altura



Árbol Binario no Ordenado

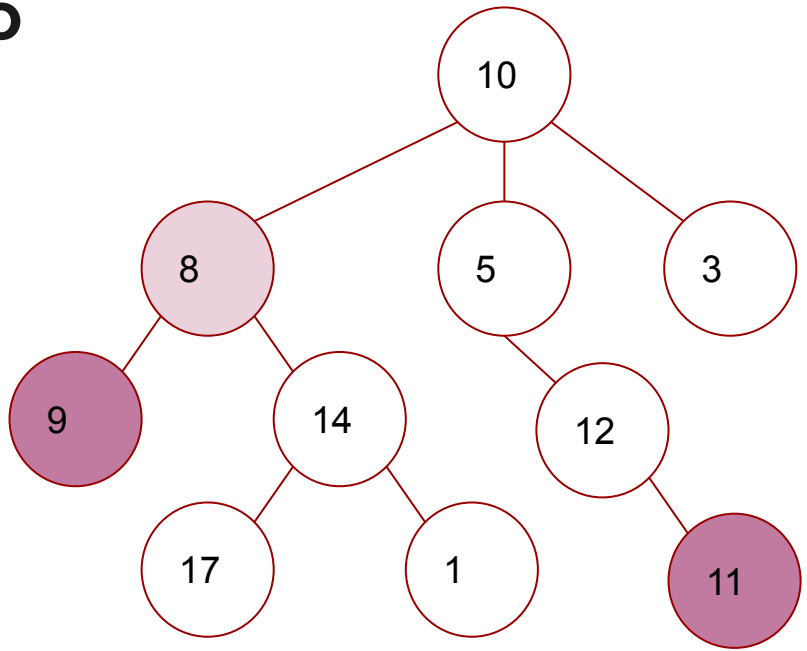
```
struct node{
    int val;
    vector<node*> hijos;
    node* padre;
};
struct node *lca(struct node* n1, struct node* n2){
    set<node*> developed;
    while(n1){
        developed.insert(n1);
        n1 = n1->padre;
    }
    while(n2){
        if(developed.find(n2) != developed.end()){
            break;
        }
        n2 = n2->padre;
    }
    return n2;

};
```



n1 = 9 n2 = 11

Árbol Binario no Ordenado

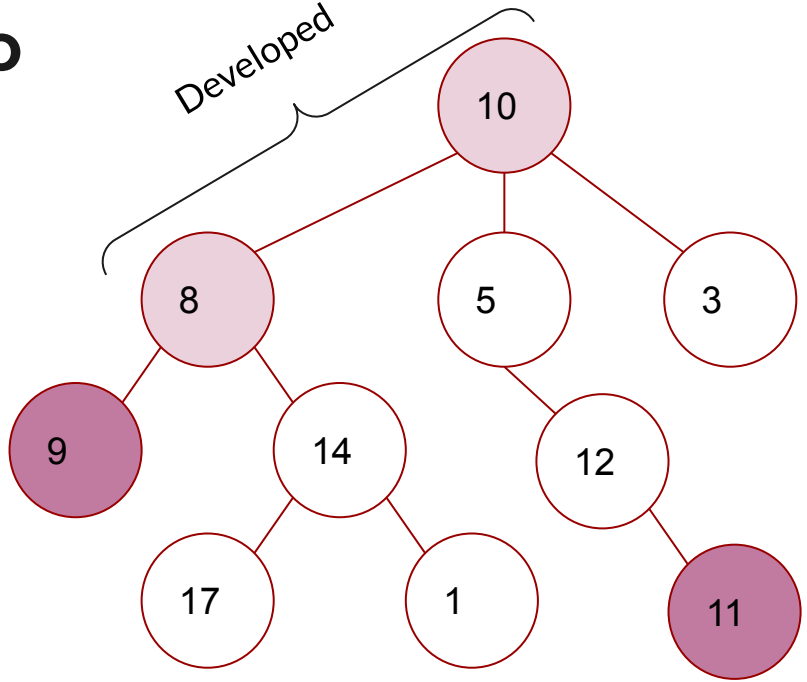
```
struct node{
    int val;
    vector<node*> hijos;
    node* padre;
};
struct node *lca(struct node* n1, struct node* n2){
    set<node*> developed;
    while(n1){
        developed.insert(n1);
        n1 = n1->padre;
    }
    while(n2){
        if(developed.find(n2) != developed.end()){
            break;
        }
        n2 = n2->padre;
    }
    return n2;

};
```



n1 = 8 n2 = 11

Árbol Binario no Ordenado

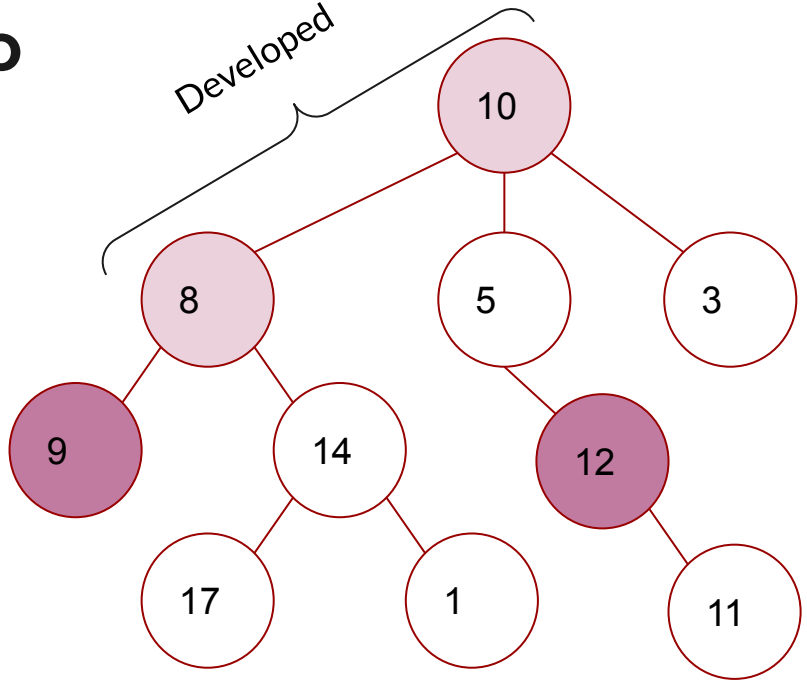
```
struct node{
    int val;
    vector<node*> hijos;
    node* padre;
};
struct node *lca(struct node* n1, struct node* n2){
    set<node*> developed;
    while(n1){
        developed.insert(n1);
        n1 = n1->padre;
    }
    while(n2){
        if(developed.find(n2) != developed.end()){
            break;
        }
        n2 = n2->padre;
    }
    return n2;

};
```



n1 = 10 n2 = 11

Árbol Binario no Ordenado

```
struct node{
    int val;
    vector<node*> hijos;
    node* padre;
};
struct node *lca(struct node* n1, struct node* n2){
    set<node*> developed;
    while(n1){
        developed.insert(n1);
        n1 = n1->padre;
    }
    while(n2){
        if(developed.find(n2) != developed.end()){
            break;
        }
        n2 = n2->padre;
    }
    return n2;

};
```

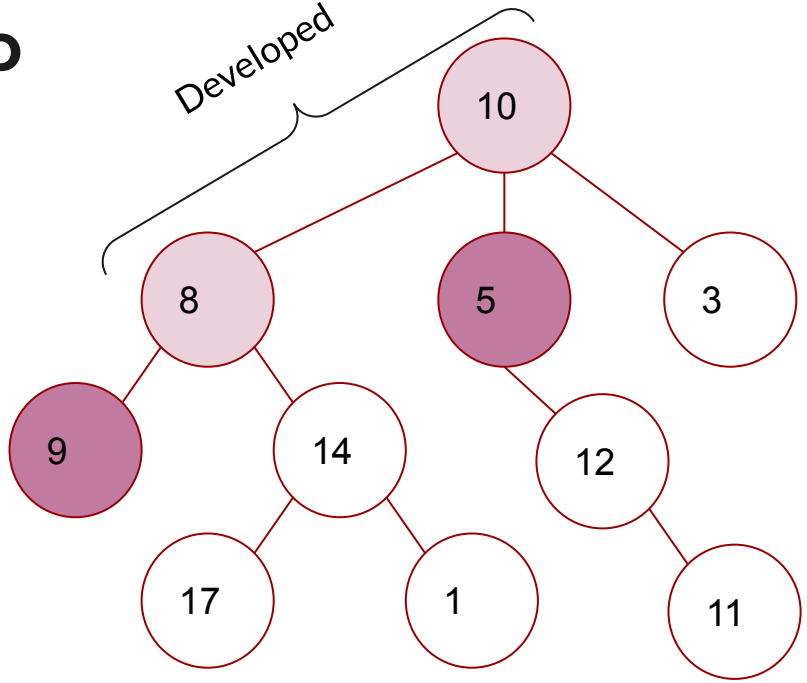


n1 = null n2 = 12

Árbol Binario no Ordenado

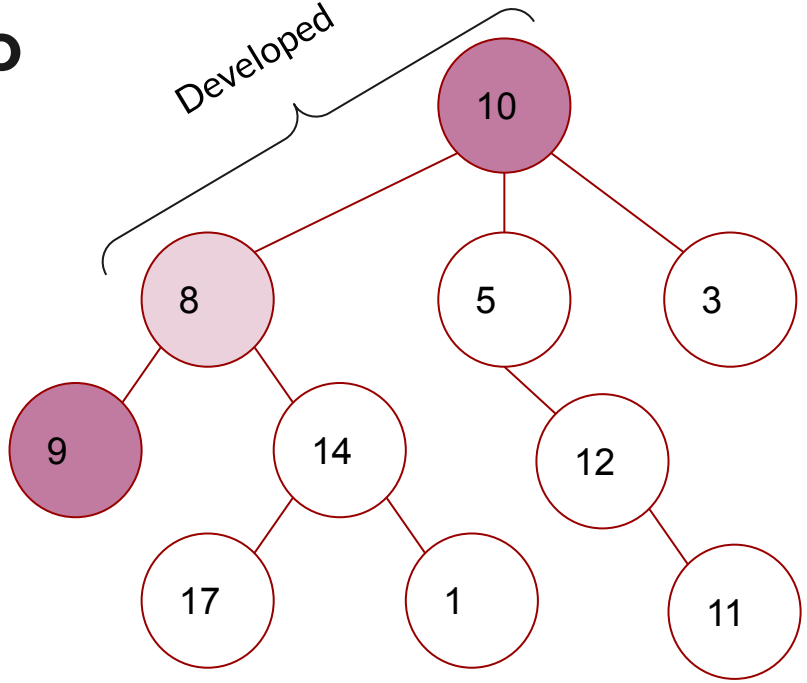
```
struct node{
    int val;
    vector<node*> hijos;
    node* padre;
};
struct node *lca(struct node* n1, struct node* n2){
    set<node*> developed;
    while(n1){
        developed.insert(n1);
        n1 = n1->padre;
    }
    while(n2){
        if(developed.find(n2) != developed.end()){
            break;
        }
        n2 = n2->padre;
    }
    return n2;

};
```

n1 = null n2 = 5



Árbol Binario no Ordenado

```
struct node{
    int val;
    vector<node*> hijos;
    node* padre;
};
struct node *lca(struct node* n1, struct node* n2){
    set<node*> developed;
    while(n1){
        developed.insert(n1);
        n1 = n1->padre;
    }
    while(n2){
        if(developed.find(n2) != developed.end()){
            break;
        }
        n2 = n2->padre;
    }
    return n2;

};
```



n1 = null n2 = 10

```
LCA de 9 y 11 es 10
```

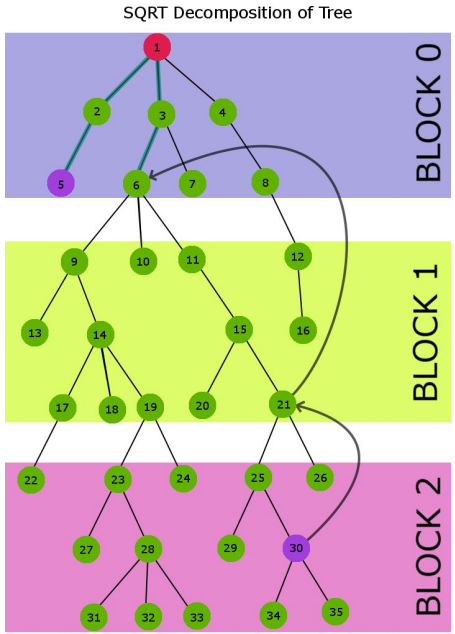
SQRT Decomposition

El truco de la descomposición Sqrt

Se categorizan los nodos del árbol en diferentes grupos de acuerdo a su profundidad. La cantidad de niveles del árbol debe ser un cuadrado perfecto. Por ello, se tendrán sqrt(h) grupos. La división se da de la siguiente manera:

Nodos desde la profundidad 0 hasta la profundidad sqrt(h)-1 en el primer grupo.

Nodos desde la profundidad sqrt(h) hasta 2*sqrt(h)-1 en el segundo grupo. Y así sucesivamente.



Se mantiene un registro del número correspondiente de grupo para cada nodo, y también su profundidad.

El truco de este algoritmo es que ya no se salta(sube) por nodo, se salta por grupo.

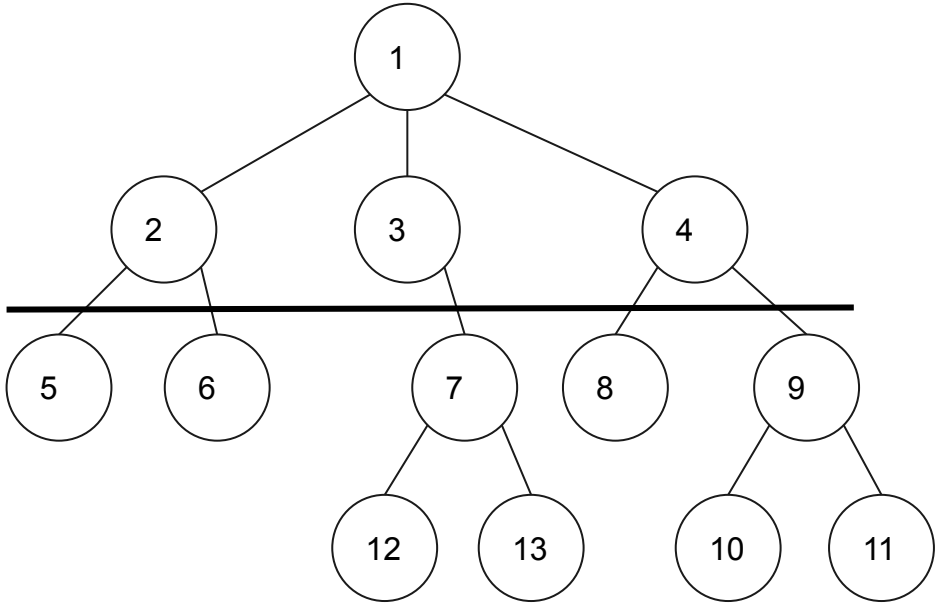
Para ello se tiene tres parámetros para cada nodo: Parent, JumpParent y profundidad.

Parent es el nodo que está encima del nodo actual.

JumpParent es el primer ancestro del nodo en el grupo.

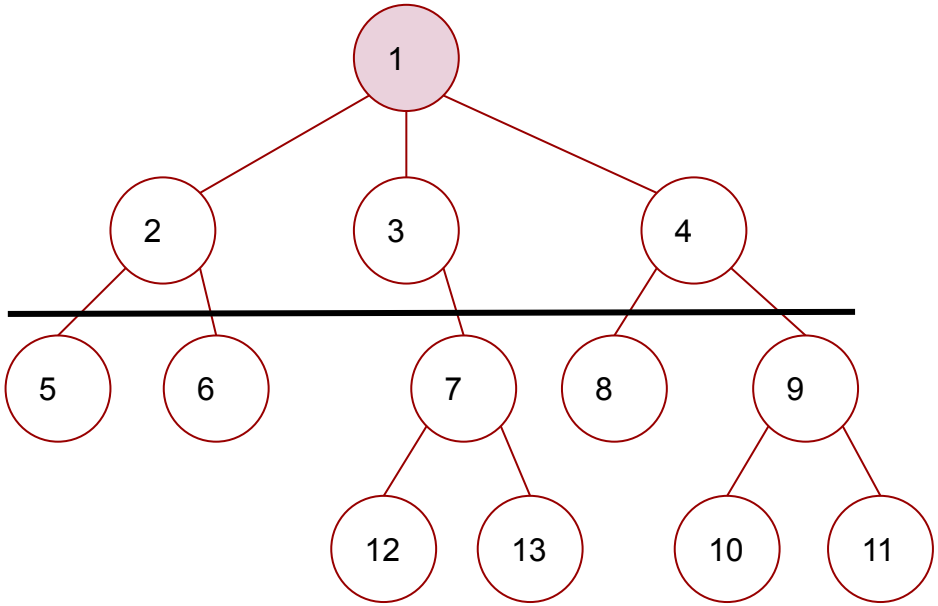
Pre-proceso

```
void preprocess(int height)
{
    block_sz = sqrt(height);
    depth[0] = -1;
    dfs(1, 0);
}
```



block_sz = 2

```
void dfs(int cur, int prev)
{
    depth[cur] = depth[prev] + 1;
    parent[cur] = prev;
    if (depth[cur]%block_sz == 0){
        jump_parent[cur] = parent[cur];
    }
    else{
        jump_parent[cur] = jump_parent[prev];
    }
    for (int i = 0; i<adj[cur].size(); ++i){
        if (adj[cur][i] != prev)
            dfs(adj[cur][i], cur);
    }
}
```



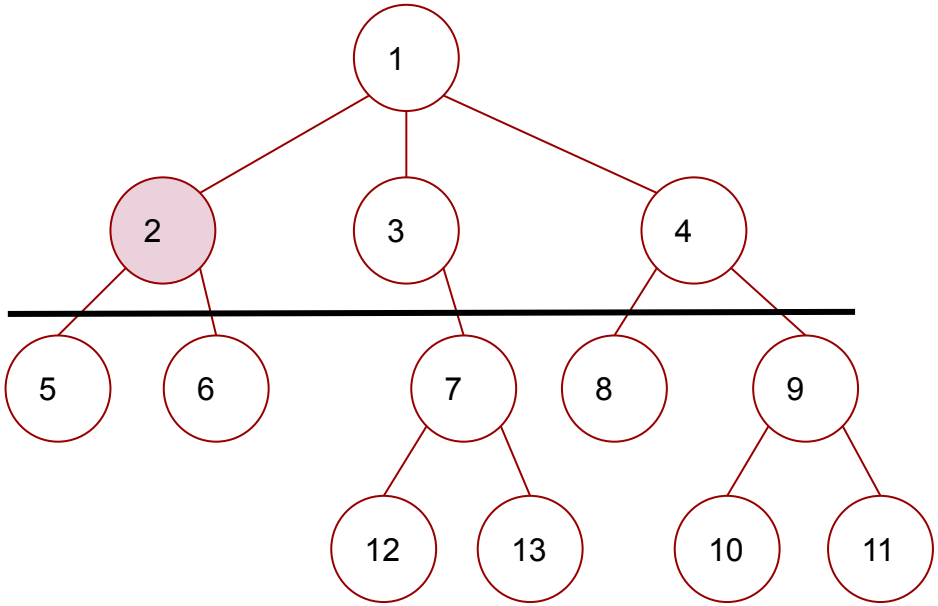
-1	0												
	0												
	0												

depth

parent

jump_parent

```
void dfs(int cur, int prev)
{
    depth[cur] = depth[prev] + 1;
    parent[cur] = prev;
    if (depth[cur]%block_sz == 0){
        jump_parent[cur] = parent[cur];
    }
    else{
        jump_parent[cur] = jump_parent[prev];
    }
    for (int i = 0; i<adj[cur].size(); ++i){
        if (adj[cur][i] != prev)
            dfs(adj[cur][i], cur);
    }
}
```



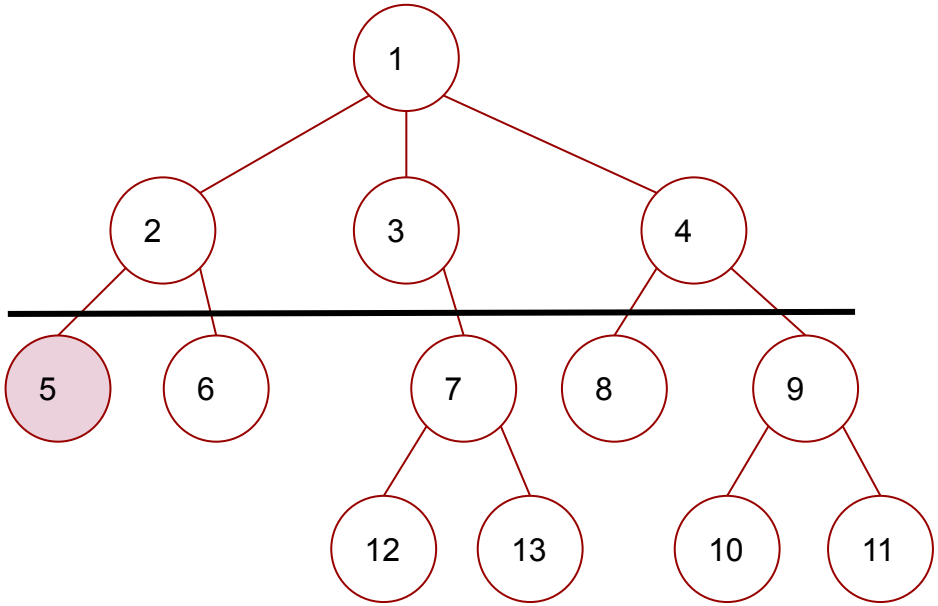
-1	0	1											
	0	1											
	0	0											

depth

parent

jump_parent

```
void dfs(int cur, int prev)
{
    depth[cur] = depth[prev] + 1;
    parent[cur] = prev;
    if (depth[cur]%block_sz == 0){
        jump_parent[cur] = parent[cur];
    }
    else{
        jump_parent[cur] = jump_parent[prev];
    }
    for (int i = 0; i<adj[cur].size(); ++i){
        if (adj[cur][i] != prev)
            dfs(adj[cur][i], cur);
    }
}
```



-1	0	1			2								
	0	1			2								
	0	0			2								

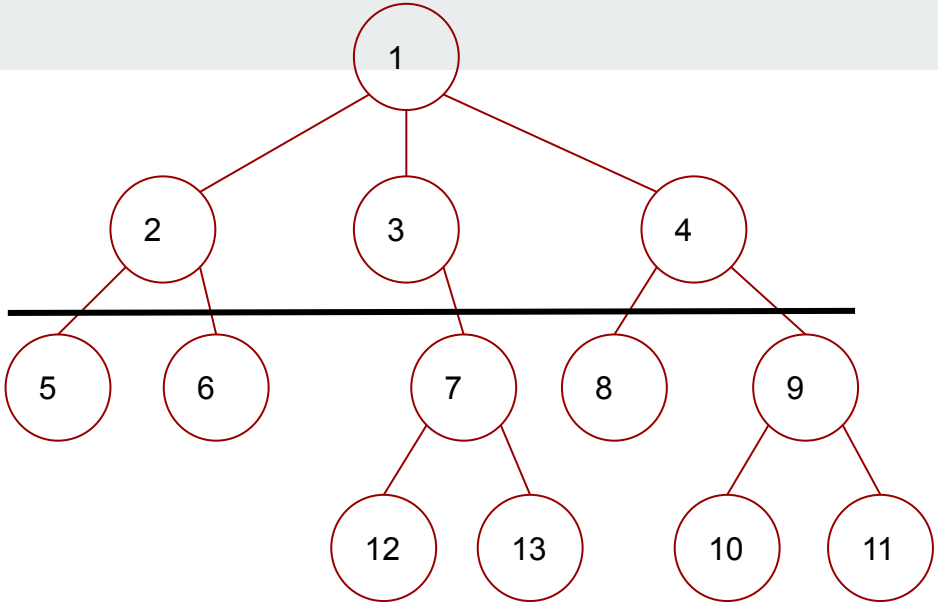
depth

parent

jump_parent

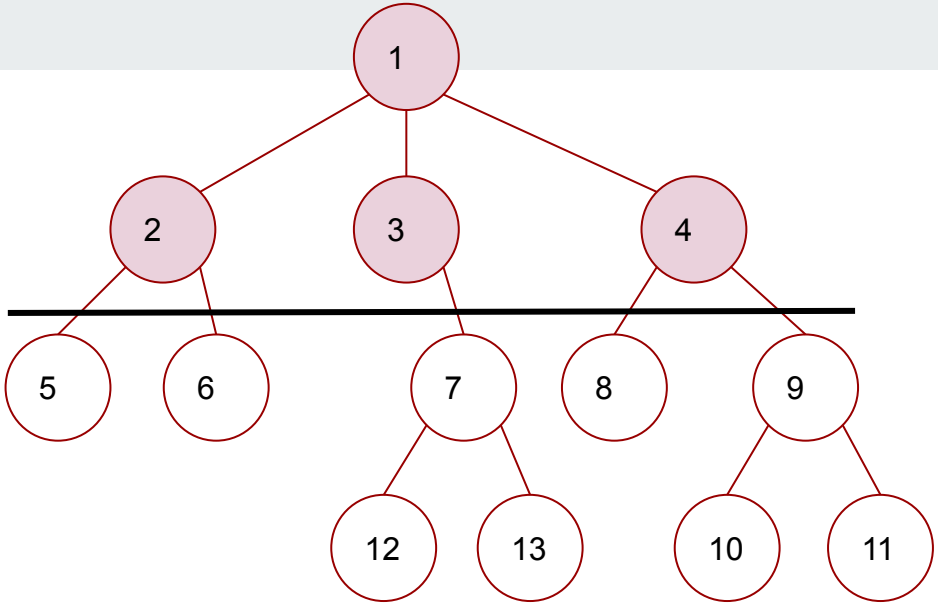
Y así sucesivamente hasta...

```
void dfs(int cur, int prev)
{
    depth[cur] = depth[prev] + 1;
    parent[cur] = prev;
    if (depth[cur]%block_sz == 0){
        jump_parent[cur] = parent[cur];
    }
    else{
        jump_parent[cur] = jump_parent[prev];
    }
    for (int i = 0; i<adj[cur].size(); ++i){
        if (adj[cur][i] != prev)
            dfs(adj[cur][i], cur);
    }
}
```



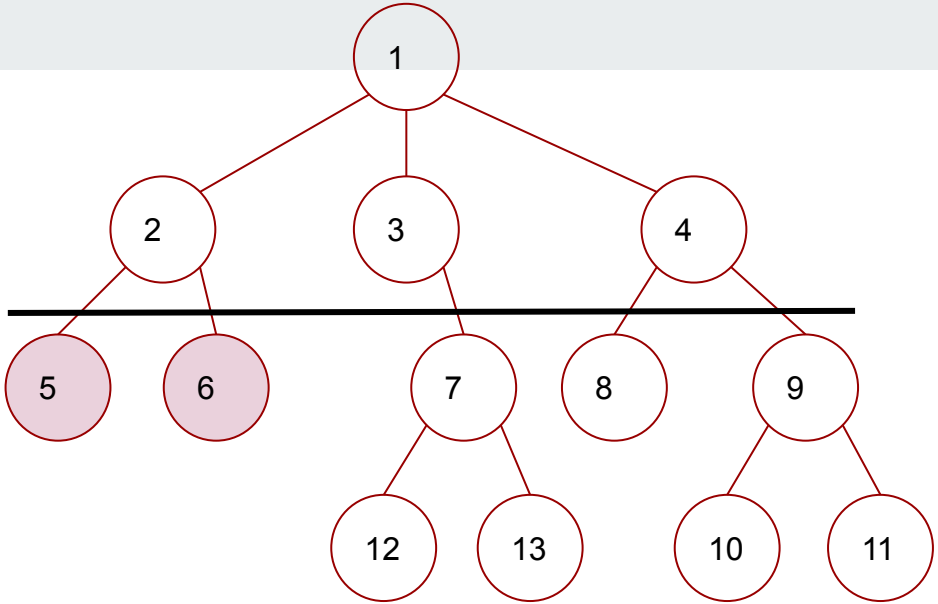
0	1	2	3	4	5	6	7	8	9	10	11	12	13	índice
-1	0	1	1	1	2	2	2	2	2	3	3	3	3	depth
-	0	1	1	1	2	2	3	4	4	9	9	7	7	parent
-	0	0	0	0	2	2	3	4	4	4	4	3	3	jump_parent

```
void dfs(int cur, int prev)
{
    depth[cur] = depth[prev] + 1;
    parent[cur] = prev;
    if (depth[cur]%block_sz == 0){
        jump_parent[cur] = parent[cur];
    }
    else{
        jump_parent[cur] = jump_parent[prev];
    }
    for (int i = 0; i<adj[cur].size(); ++i){
        if (adj[cur][i] != prev)
            dfs(adj[cur][i], cur);
    }
}
```



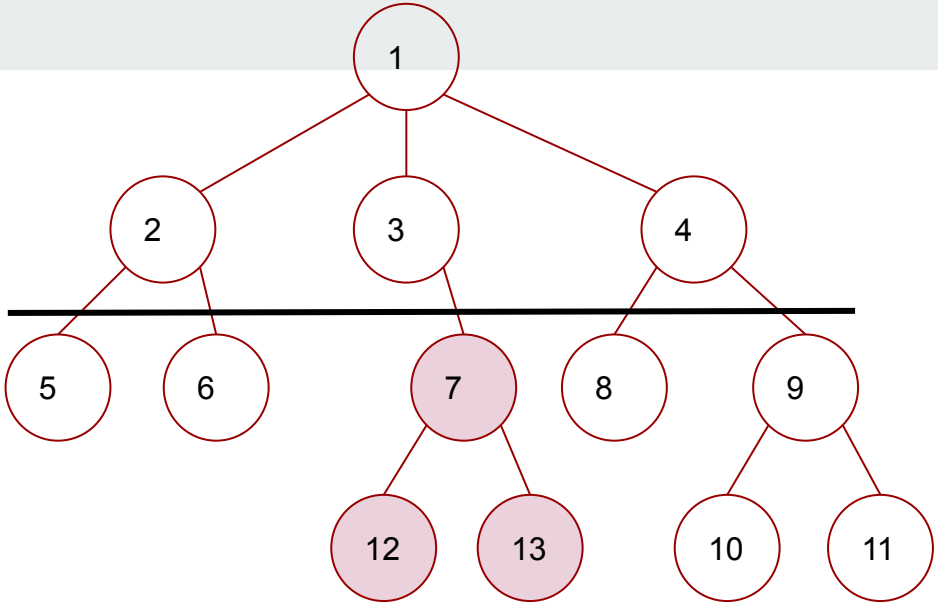
0	1	2	3	4	5	6	7	8	9	10	11	12	13	índice
-1	0	1	1	1	2	2	2	2	2	3	3	3	3	depth
-	0	1	1	1	2	2	3	4	4	9	9	7	7	parent
-	0	0	0	0	2	2	3	4	4	4	4	3	3	jump_parent

```
void dfs(int cur, int prev)
{
    depth[cur] = depth[prev] + 1;
    parent[cur] = prev;
    if (depth[cur]%block_sz == 0){
        jump_parent[cur] = parent[cur];
    }
    else{
        jump_parent[cur] = jump_parent[prev];
    }
    for (int i = 0; i<adj[cur].size(); ++i){
        if (adj[cur][i] != prev)
            dfs(adj[cur][i], cur);
    }
}
```



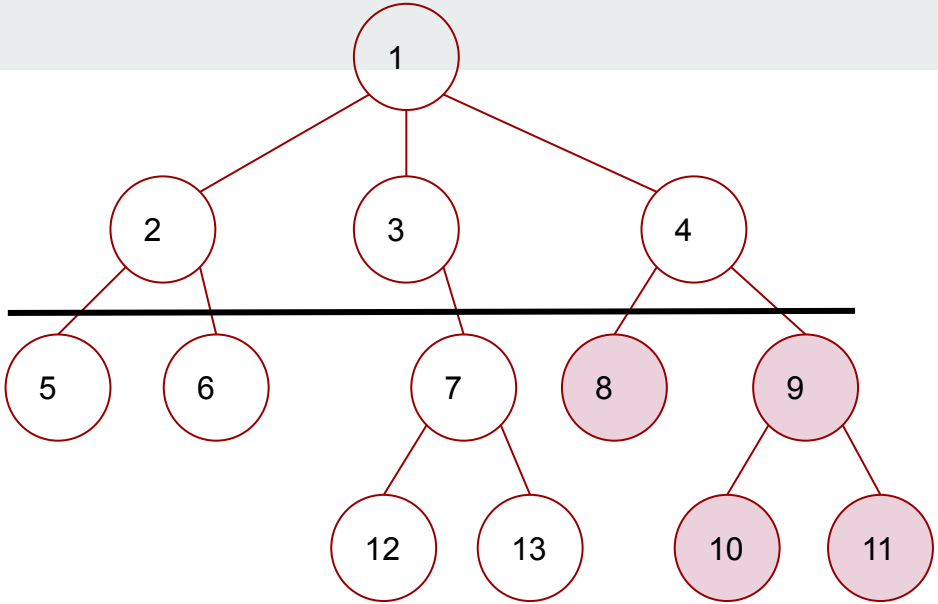
0	1	2	3	4	5	6	7	8	9	10	11	12	13	índice
-1	0	1	1	1	2	2	2	2	2	3	3	3	3	depth
-	0	1	1	1	2	2	3	4	4	9	9	7	7	parent
-	0	0	0	0	2	2	3	4	4	4	4	3	3	jump_parent

```
void dfs(int cur, int prev)
{
    depth[cur] = depth[prev] + 1;
    parent[cur] = prev;
    if (depth[cur]%block_sz == 0){
        jump_parent[cur] = parent[cur];
    }
    else{
        jump_parent[cur] = jump_parent[prev];
    }
    for (int i = 0; i<adj[cur].size(); ++i){
        if (adj[cur][i] != prev)
            dfs(adj[cur][i], cur);
    }
}
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13	índice
-1	0	1	1	1	2	2	2	2	2	3	3	3	3	depth
-	0	1	1	1	2	2	3	4	4	9	9	7	7	parent
-	0	0	0	0	2	2	3	4	4	4	4	3	3	jump_parent

```
void dfs(int cur, int prev)
{
    depth[cur] = depth[prev] + 1;
    parent[cur] = prev;
    if (depth[cur]%block_sz == 0){
        jump_parent[cur] = parent[cur];
    }
    else{
        jump_parent[cur] = jump_parent[prev];
    }
    for (int i = 0; i<adj[cur].size(); ++i){
        if (adj[cur][i] != prev)
            dfs(adj[cur][i], cur);
    }
}
```

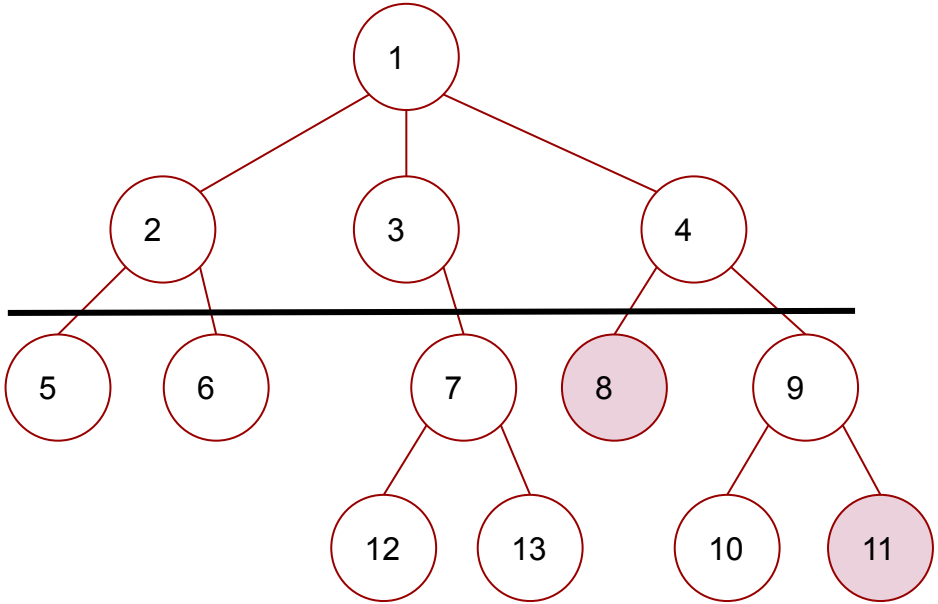


0	1	2	3	4	5	6	7	8	9	10	11	12	13	índice
-1	0	1	1	1	2	2	2	2	2	3	3	3	3	depth
-	0	1	1	1	2	2	3	4	4	9	9	7	7	parent
-	0	0	0	0	2	2	3	4	4	4	4	3	3	jump_parent

EJEMPLO 1

u = 11 v = 8

```
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])
            swap(u,v);

        v = jump_parent[v];
    }
    return LCANaive(u,v);
}
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}
```



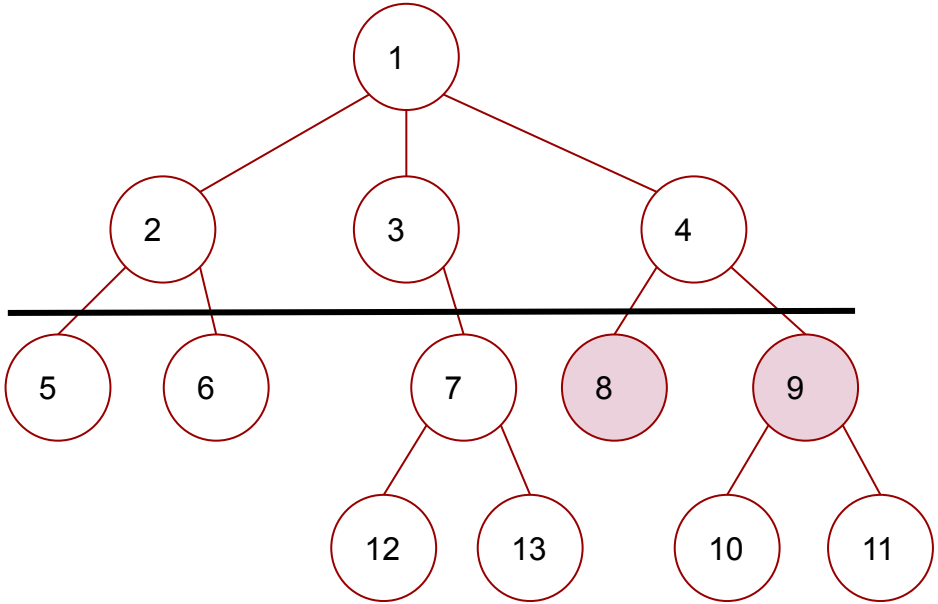
0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	0	0	0	0	2	2	3	4	4	4	4	3	3

índice

jump_parent

u = 8 v = 11 -> 9

```
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])
            swap(u,v);

        v = jump_parent[v];
    }
    return LCANaive(u,v);
}
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}
```



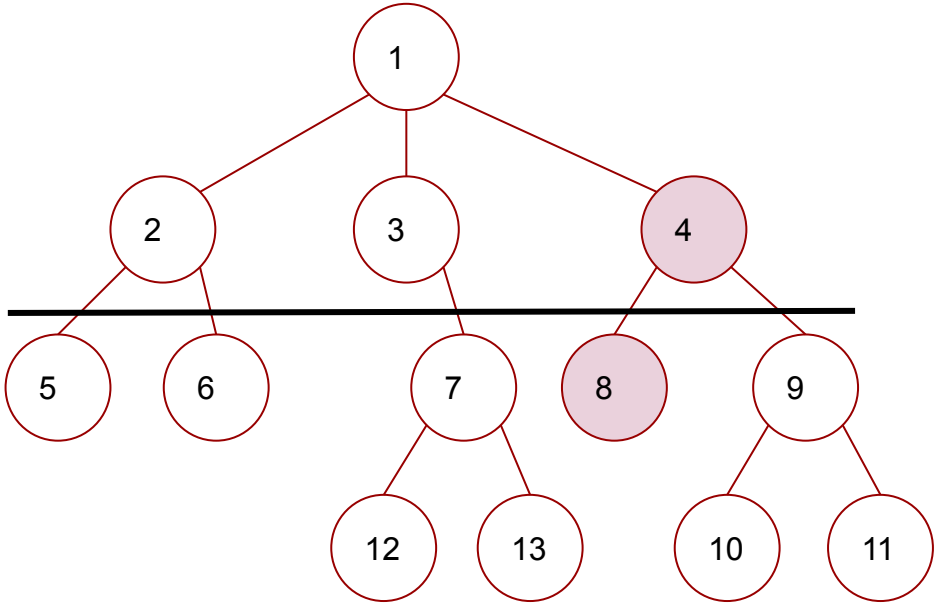
0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	0	0	0	0	2	2	3	4	4	4	4	3	3

índice

jump_parent

u = 8 v = 9->4

```
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])
            swap(u,v);

        v = jump_parent[v];
    }
    return LCANaive(u,v);
}
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}
```



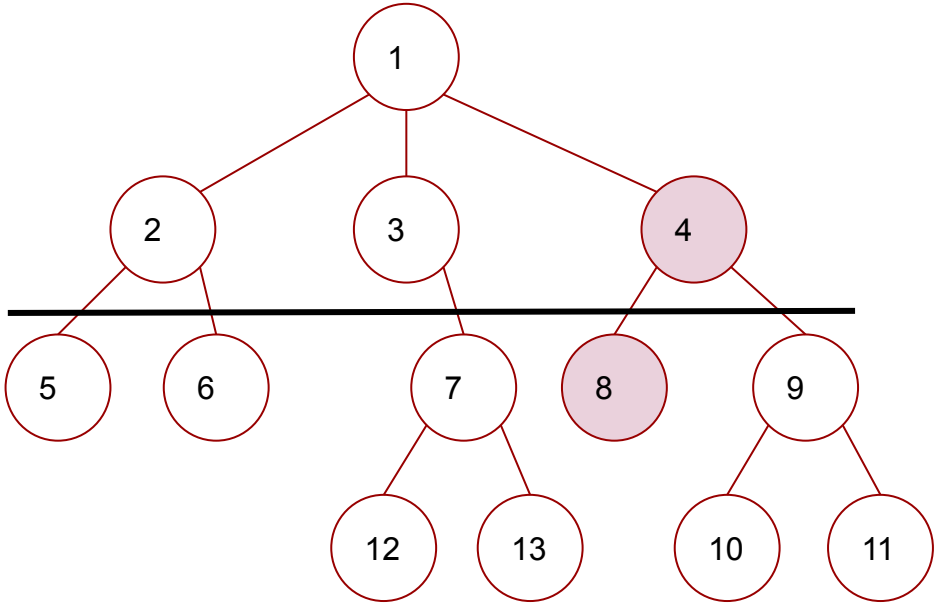
0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	0	0	0	0	2	2	3	4	4	4	4	3	3

índice

jump_parent

u = 8 v = 4

```
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])
            swap(u,v);

        v = jump_parent[v];
    }
    return LCANaive(u,v);
}
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}
```



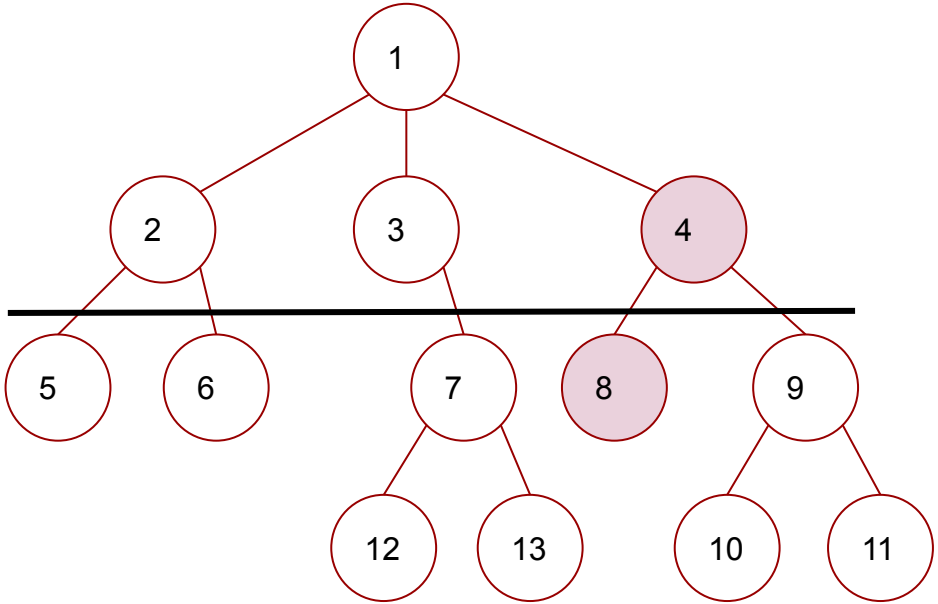
0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	0	0	0	0	2	2	3	4	4	4	4	3	3

índice

jump_parent

u = 4 v = 8

```
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])
            swap(u,v);

        v = jump_parent[v];
    }
    return LCANaive(u,v);
}
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}
```



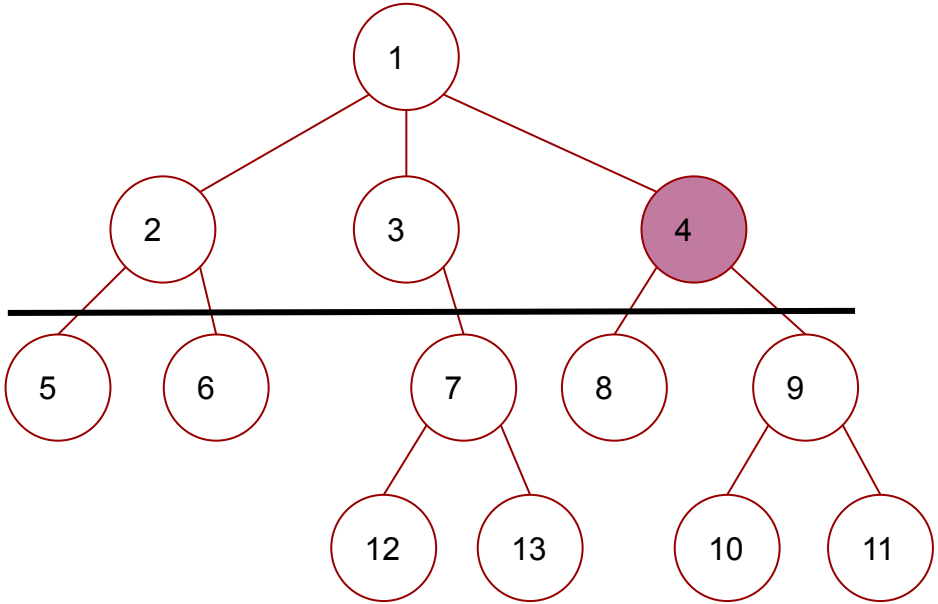
0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	0	0	0	0	2	2	3	4	4	4	4	3	3

índice

jump_parent

u = 4 v = 4

```
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])
            swap(u,v);

        v = jump_parent[v];
    }
    return LCANaive(u,v);
}
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}
```




```
LCA(11,8) : 4
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	0	0	0	0	2	2	3	4	4	4	4	3	3

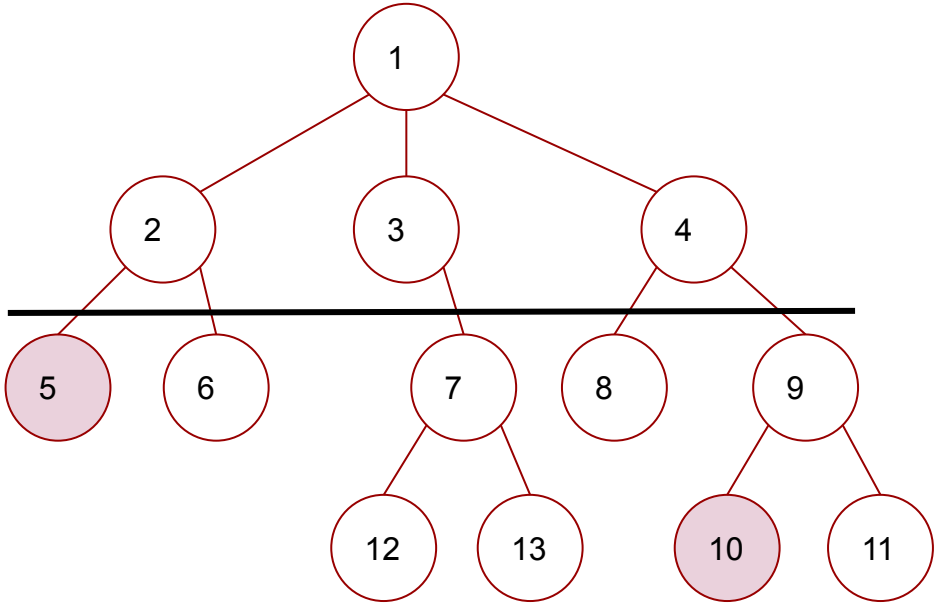
índice

jump_parent

EJEMPLO 2

u =5 v = 10

```
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])
            swap(u,v);

        v = jump_parent[v];
    }
    return LCANaive(u,v);
}
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	0	0	0	0	2	2	3	4	4	4	4	3	3

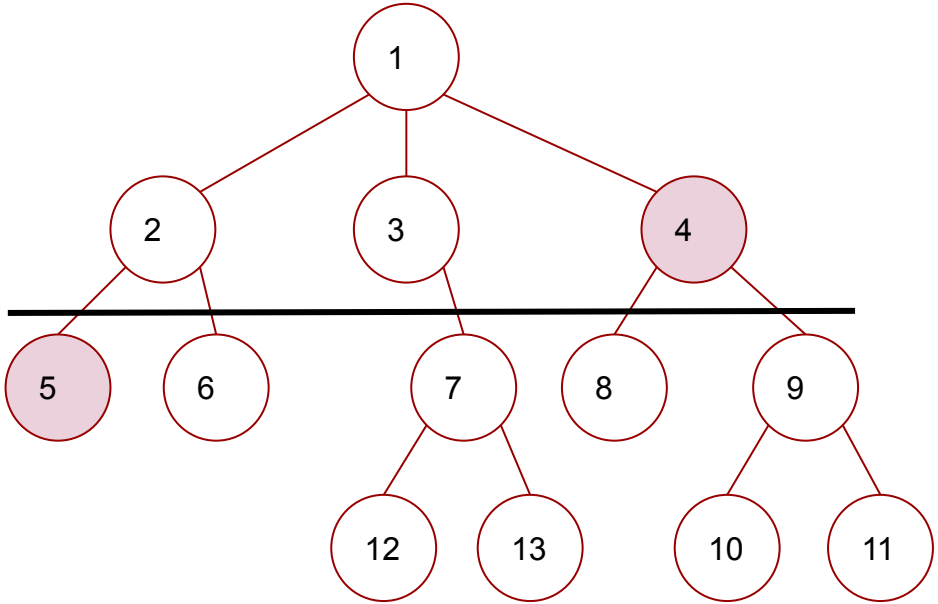
índice

jump_parent

EJEMPLO 2

u =5 v = 4

```
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])
            swap(u,v);

        v = jump_parent[v];
    }
    return LCANaive(u,v);
}
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	0	0	0	0	2	2	3	4	4	4	4	3	3

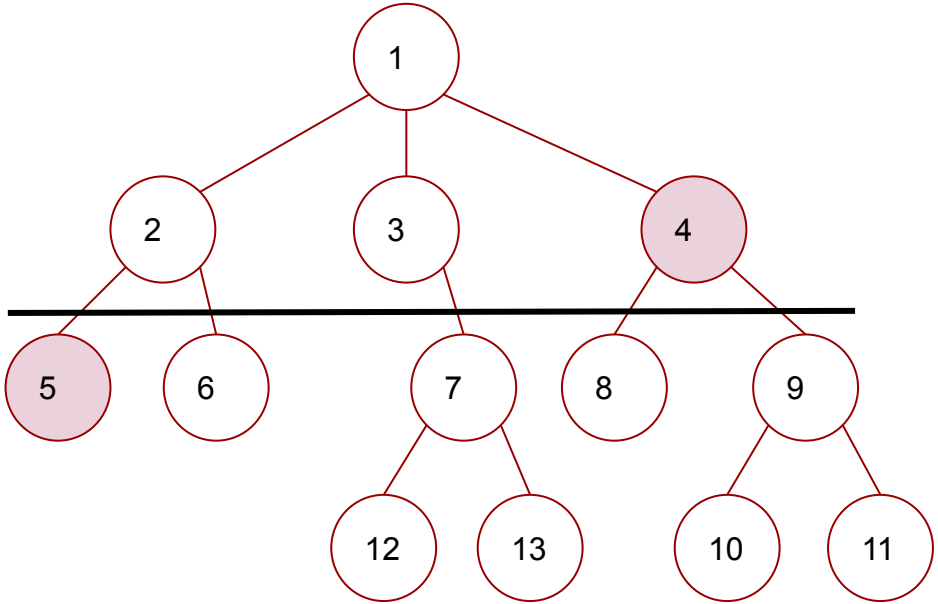
índice

jump_parent

EJEMPLO 2

u = 4 v = 5

```
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])
            swap(u,v);

        v = jump_parent[v];
    }
    return LCANaive(u,v);
}
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	0	0	0	0	2	2	3	4	4	4	4	3	3

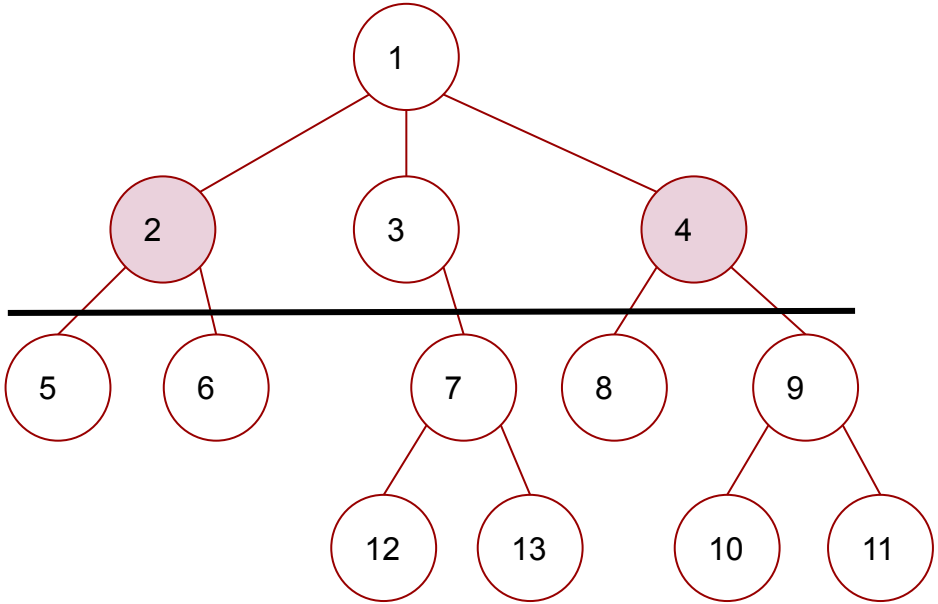
índice

jump_parent

EJEMPLO 2

u = 4 v = 2

```
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])
            swap(u,v);

        v = jump_parent[v];
    }
    return LCANaive(u,v);
}
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}
```



y se hace el mismo proceso LCANaive que en el primer ejemplo

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	0	0	0	0	2	2	3	4	4	4	4	3	3

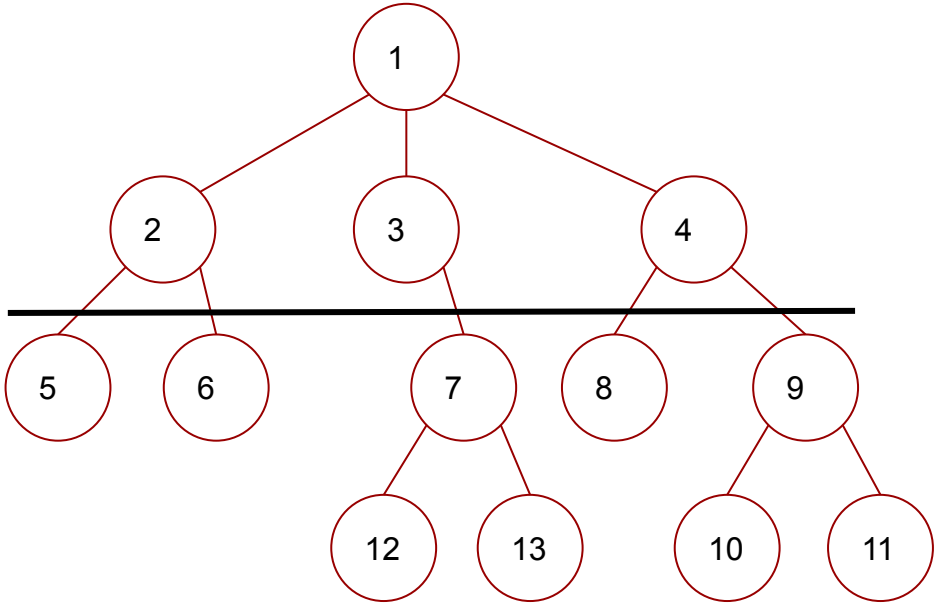
índice

jump_parent

EJEMPLO 2

u = 4 v = 5

```
int LCASQRT(int u, int v)
{
    while (jump_parent[u] != jump_parent[v])
    {
        if (depth[u] > depth[v])
            swap(u,v);

        v = jump_parent[v];
    }
    return LCANaive(u,v);
}
int LCANaive(int u,int v)
{
    if (u == v)  return u;
    if (depth[u] > depth[v])
        swap(u,v);
    v = parent[v];
    return LCANaive(u,v);
}
```




```
LCA(5,10) : 1
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
-	0	0	0	0	2	2	3	4	4	4	4	3	3

índice

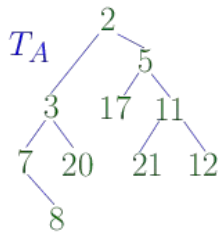
jump_parent

Range Minimum Query(RMQ) usando Segment Tree

Range Minimum Query(RMQ)

RMQ resuelve el problema de encontrar el valor mínimo en un subarreglo de un arreglo de objetos; es decir, para encontrar la posición de un elemento con el valor mínimo entre dos índices especificados.

$$A = [8, 7, 3, 20, 2, 17, 5, 21, 11, 12]$$



$$min(A, 6, 10) = \quad \text{LCA}(17,12) = 5$$

Range Minimum Query(RMQ)

Se desarrollará con un enfoque basado en árbol de segmentos. Lo que da:

n : Tamaño del array Euler

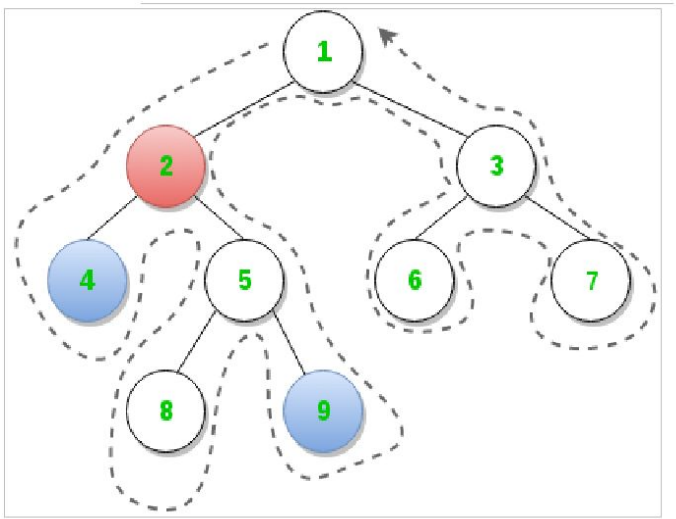
Tiempo de construcción: O (n)

Tiempo de consulta: O (Logn).

Espacio requerido adicional por el árbol de segmentos: O(n)

Reducir RMQ a LCA

La idea es atravesar el árbol a partir de la raíz mediante un recorrido de Euler (recorrido sin levantar el lápiz), que es un recorrido de tipo DFS con características de recorrido en preorden.



Euler Tour

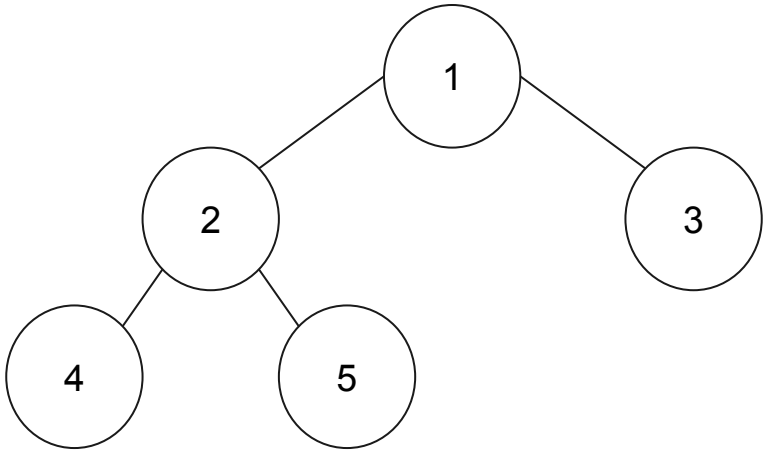
Para la construcción, se requiere tres arrays para la implementación:

- Nodos visitados en orden del tour Euler -> Euler
- Nivel de cada nodo visitado en Euler tour -> Nivel
- Índice de la primera ocurrencia de un nodo en el recorrido de Euler -> PrimeraOcurrencia

Algoritmo

- Completar un recorrido Euler en el árbol y completar los arrays de euler, nivel y primera ocurrencia.
- Construir el segment tree a partir de los niveles y el euler.
- Usando array de primera ocurrencia, obtener los índices correspondientes a los nodos que serán los límites del rango en el arreglo de nivel usado por el algoritmo RMQ para el valor mínimo.
- Una vez que el algoritmo devuelve el índice de nivel mínimo en el rango, se lo usa para determinar el LCA usando el array de euler.

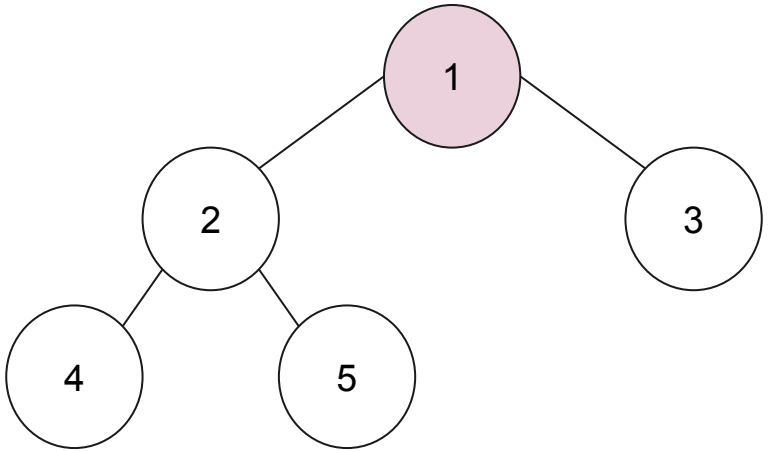
```
int LCA(Node *root, int u, int v)
{
    memset(primeraOcurrencia, -1, sizeof(int)*(V+1));
    ind = 0;
    eulerTour(root, 0);
    int *st = constructST(nivel, 2*V-1);

    if (primeraOcurrencia[u]>primeraOcurrencia[v])
       swap(u, v);

    int qs = primeraOcurrencia[u];
    int qe = primeraOcurrencia[v];

    int index = RMQ(st, 2*V-1, qs, qe);
    return euler[index];
}
```



euler								
nivel								

primera_ocurrencia	-1	-1	-1	-1	-1	-1

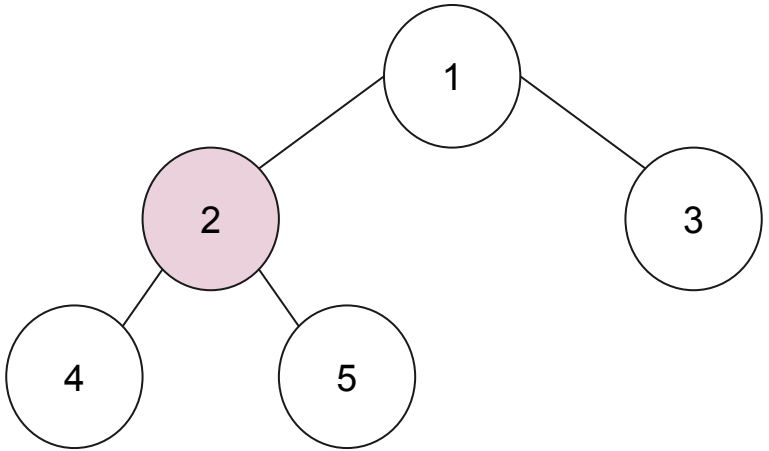
```
void eulerTour(Node *actual, int l)
{
    if (actual)//nodo actual
    {
        euler[ind] = actual->key; // insertar en eu
        nivel[ind] = l;           // insertar l en ni
        ind++;                    // incrementar inde

        if (primeraOcurrencia[actual->key] == -1){
            primeraOcurrencia[actual->key] = ind-1;
        }
        if (actual->left)
        {
            eulerTour(actual->left, l+1);
            euler[ind] = actual->key;
            nivel[ind] = l;
            ind++;
        }

        if (actual->right)
        {
            eulerTour(actual->right, l+1);
            euler[ind]=actual->key;
            nivel[ind] = l;
            ind++;
        }
    }
}
```



euler	1								
nivel	0								

primera_ocurrencia	0				

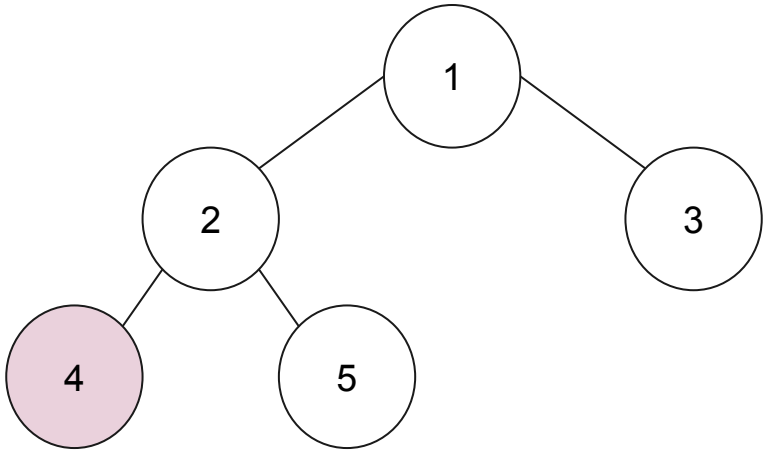
```
void eulerTour(Node *actual, int l)
{
    if (actual)//nodo actual
    {
        euler[ind] = actual->key; // insertar en eu
        nivel[ind] = l;           // insertar l en ni
        ind++;                    // incrementar inde

        if (primeraOcurrencia[actual->key] == -1){
            primeraOcurrencia[actual->key] = ind-1;
        }
        if (actual->left)
        {
            eulerTour(actual->left, l+1);
            euler[ind] = actual->key;
            nivel[ind] = l;
            ind++;
        }

        if (actual->right)
        {
            eulerTour(actual->right, l+1);
            euler[ind]=actual->key;
            nivel[ind] = l;
            ind++;
        }
    }
}
```



euler	1	2							
nivel	0	1							

primera_ocurrencia	0	1			

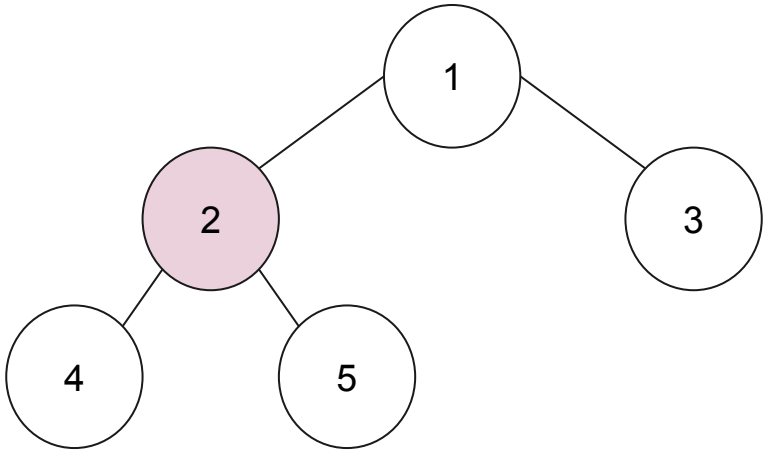
```
void eulerTour(Node *actual, int l)
{
    if (actual)//nodo actual
    {
        euler[ind] = actual->key; // insertar en eu
        nivel[ind] = l;           // insertar l en ni
        ind++;                    // incrementar inde

        if (primeraOcurrencia[actual->key] == -1){
            primeraOcurrencia[actual->key] = ind-1;
        }
        if (actual->left)
        {
            eulerTour(actual->left, l+1);
            euler[ind] = actual->key;
            nivel[ind] = l;
            ind++;
        }

        if (actual->right)
        {
            eulerTour(actual->right, l+1);
            euler[ind]=actual->key;
            nivel[ind] = l;
            ind++;
        }
    }
}
```



euler	1	2	4						
nivel	0	1	2						

primera_ocurrencia	0	1		2	

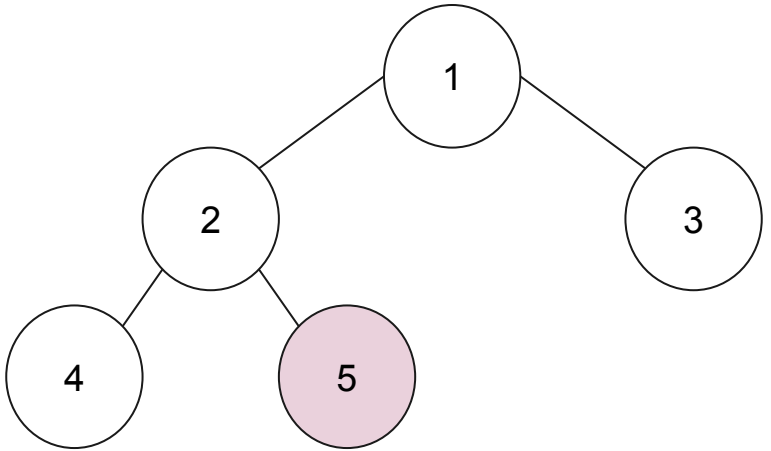
```
void eulerTour(Node *actual, int l)
{
    if (actual)//nodo actual
    {
        euler[ind] = actual->key; // insertar en eu
        nivel[ind] = l;           // insertar l en ni
        ind++;                    // incrementar inde

        if (primeraOcurrencia[actual->key] == -1){
            primeraOcurrencia[actual->key] = ind-1;
        }
        if (actual->left)
        {
            eulerTour(actual->left, l+1);
            euler[ind] = actual->key;
            nivel[ind] = l;
            ind++;
        }

        if (actual->right)
        {
            eulerTour(actual->right, l+1);
            euler[ind]=actual->key;
            nivel[ind] = l;
            ind++;
        }
    }
}
```



euler	1	2	4	2				
nivel	0	1	2	1				

primera_ocurrencia	0	1		2	

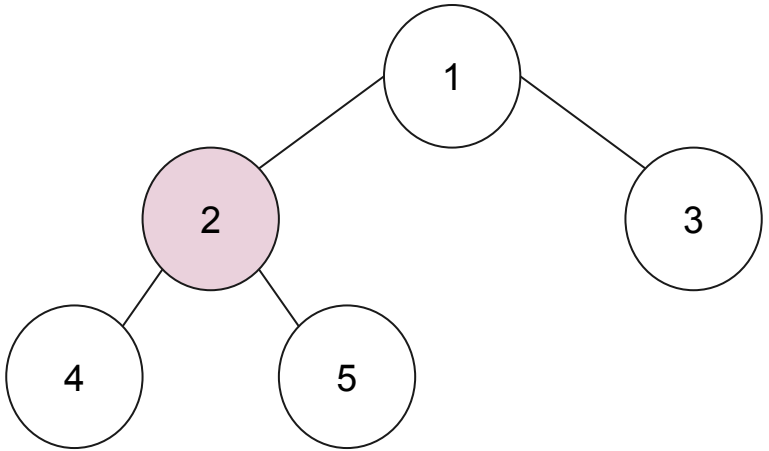
```
void eulerTour(Node *actual, int l)
{
    if (actual)//nodo actual
    {
        euler[ind] = actual->key; // insertar en eu
        nivel[ind] = l;           // insertar l en ni
        ind++;                    // incrementar inde

        if (primeraOcurrencia[actual->key] == -1){
            primeraOcurrencia[actual->key] = ind-1;
        }
        if (actual->left)
        {
            eulerTour(actual->left, l+1);
            euler[ind] = actual->key;
            nivel[ind] = l;
            ind++;
        }

        if (actual->right)
        {
            eulerTour(actual->right, l+1);
            euler[ind]=actual->key;
            nivel[ind] = l;
            ind++;
        }
    }
}
```



euler	1	2	4	2	5				
nivel	0	1	2	1	2				

primera_ocurrencia	0	1		2	4

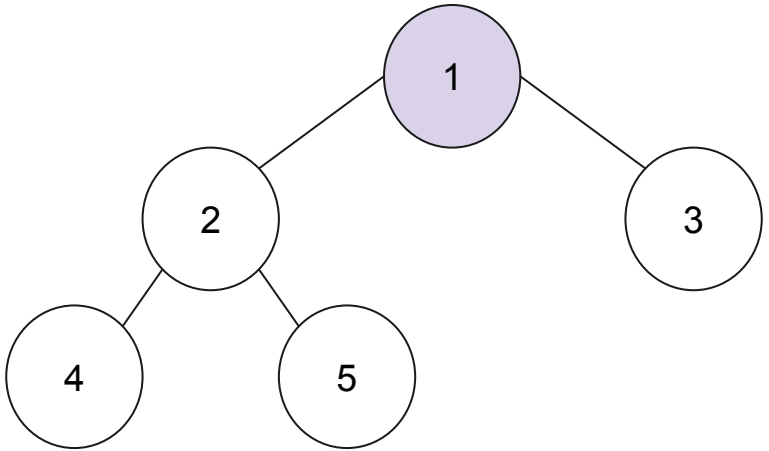
```
void eulerTour(Node *actual, int l)
{
    if (actual)//nodo actual
    {
        euler[ind] = actual->key; // insertar en eu
        nivel[ind] = l;           // insertar l en ni
        ind++;                    // incrementar inde

        if (primeraOcurrencia[actual->key] == -1){
            primeraOcurrencia[actual->key] = ind-1;
        }
        if (actual->left)
        {
            eulerTour(actual->left, l+1);
            euler[ind] = actual->key;
            nivel[ind] = l;
            ind++;
        }

        if (actual->right)
        {
            eulerTour(actual->right, l+1);
            euler[ind]=actual->key;
            nivel[ind] = l;
            ind++;
        }
    }
}
```



euler	1	2	4	2	5	2			
nivel	0	1	2	1	2	1			

primera_ocurrencia	0	1		2	4

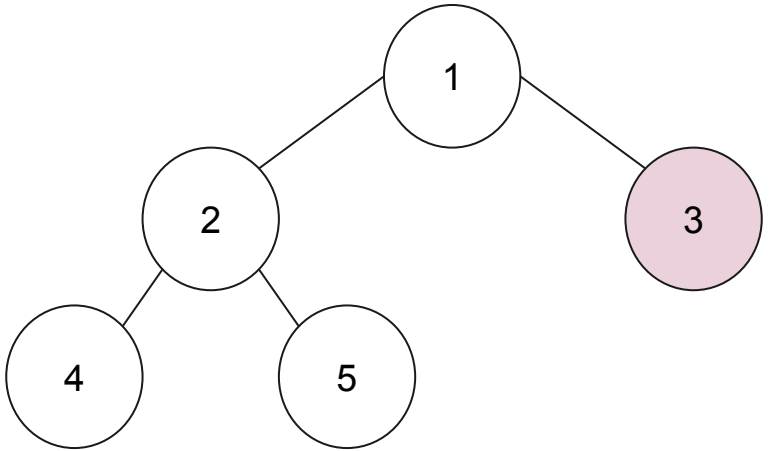
```
void eulerTour(Node *actual, int l)
{
    if (actual)//nodo actual
    {
        euler[ind] = actual->key; // insertar en eu
        nivel[ind] = l;           // insertar l en ni
        ind++;                    // incrementar inde

        if (primeraOcurrencia[actual->key] == -1){
            primeraOcurrencia[actual->key] = ind-1;
        }
        if (actual->left)
        {
            eulerTour(actual->left, l+1);
            euler[ind] = actual->key;
            nivel[ind] = l;
            ind++;
        }

        if (actual->right)
        {
            eulerTour(actual->right, l+1);
            euler[ind]=actual->key;
            nivel[ind] = l;
            ind++;
        }
    }
}
```



euler	1	2	4	2	5	2	1		
nivel	0	1	2	1	2	1	0		

primera_ocurrencia	0	1		2	4

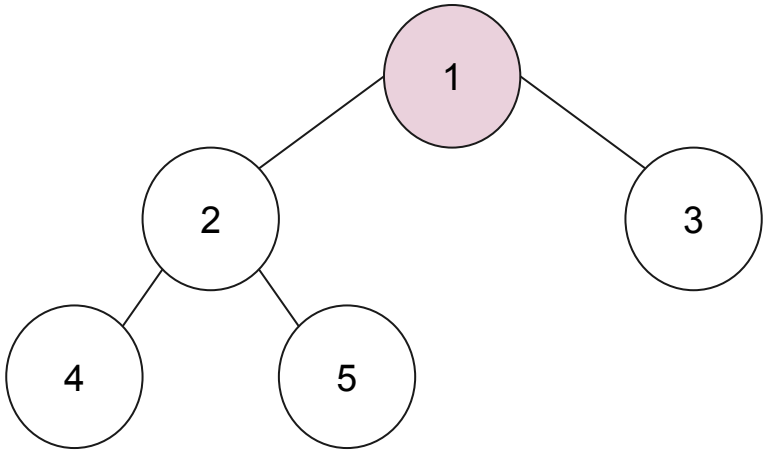
```
void eulerTour(Node *actual, int l)
{
    if (actual)//nodo actual
    {
        euler[ind] = actual->key; // insertar en eu
        nivel[ind] = l;           // insertar l en ni
        ind++;                    // incrementar inde

        if (primeraOcurrencia[actual->key] == -1){
            primeraOcurrencia[actual->key] = ind-1;
        }
        if (actual->left)
        {
            eulerTour(actual->left, l+1);
            euler[ind] = actual->key;
            nivel[ind] = l;
            ind++;
        }

        if (actual->right)
        {
            eulerTour(actual->right, l+1);
            euler[ind]=actual->key;
            nivel[ind] = l;
            ind++;
        }
    }
}
```



euler	1	2	4	2	5	2	1	3	
nivel	0	1	2	1	2	1	0	1	

primera_ocurrencia	0	1	7	2	4

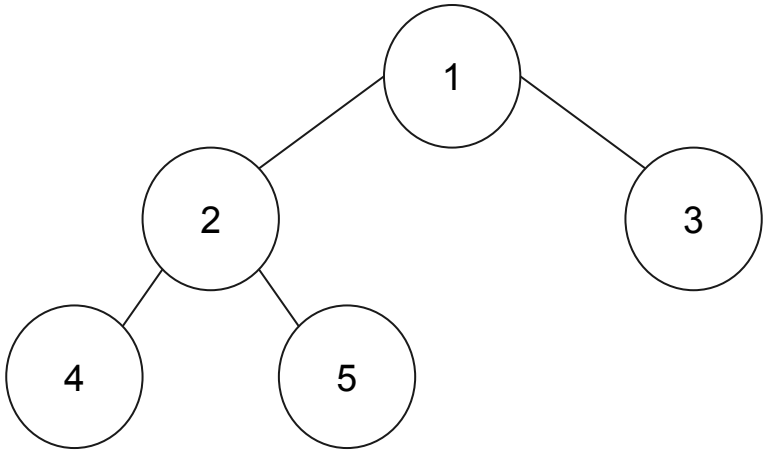
```
void eulerTour(Node *actual, int l)
{
    if (actual)//nodo actual
    {
        euler[ind] = actual->key; // insertar en eu
        nivel[ind] = l;           // insertar l en ni
        ind++;                    // incrementar inde

        if (primeraOcurrencia[actual->key] == -1){
            primeraOcurrencia[actual->key] = ind-1;
        }
        if (actual->left)
        {
            eulerTour(actual->left, l+1);
            euler[ind] = actual->key;
            nivel[ind] = l;
            ind++;
        }

        if (actual->right)
        {
            eulerTour(actual->right, l+1);
            euler[ind]=actual->key;
            nivel[ind] = l;
            ind++;
        }
    }
}
```



euler	1	2	4	2	5	2	1	3	1
nivel	0	1	2	1	2	1	0	1	0

primera_ocurrencia	0	1	7	2	4

```
int LCA(Node *root, int u, int v)
{
    memset(primeraOcurrencia, -1, sizeof(int)*(V+1));
    ind = 0;
    eulerTour(root, 0);
    int *st = constructST(nivel, 2*V-1);

    if (primeraOcurrencia[u]>primeraOcurrencia[v])
       swap(u, v);

    int qs = primeraOcurrencia[u];
    int qe = primeraOcurrencia[v];

    int index = RMQ(st, 2*V-1, qs, qe);
    return euler[index];
}
```



euler	1	2	4	2	5	2	1	3	1
nivel	0	1	2	1	2	1	0	1	0

primera_ocurrencia	0	1	7	2	4

Construir el Segment Tree a partir de los niveles

```
int *constructST(int nivel_[], int n)
{
    int x = Log2(n)+1; //4
    int max_size = 2*(1<<x) - 1; //31   2*pow(2,


    int *st = new int[max_size];
    constructSTUtil(0, 0, n-1, nivel_, st);
    return st;
}
```

nivel	0	1	2	1	2	1	0	1	0

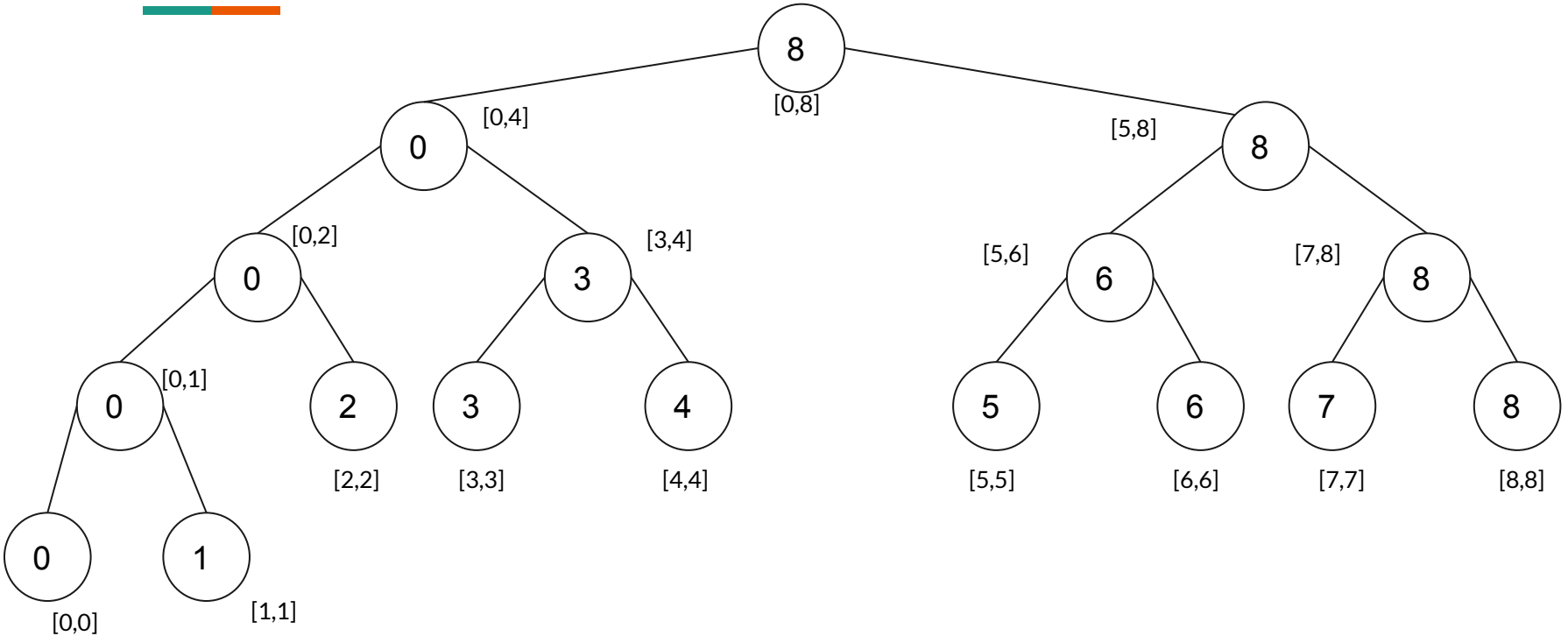
Construir el Segment Tree a partir de los niveles

```
void constructSTUtil(int si, int ss, int se, int nivel_[], int *st)
{
    if (ss == se)st[si] = ss;

    else
    {
        int mid = (ss + se)/2;
        constructSTUtil(si*2+1, ss, mid, nivel_, st);
        constructSTUtil(si*2+2, mid+1, se, nivel_, st);

        if (nivel_[st[2*si+1]] < nivel_[st[2*si+2]]){
            st[si] = st[2*si+1];
        }
        else{
            st[si] = st[2*si+2];
        }
    }
}
```

nivel	0	1	2	1	2	1	0	1	0

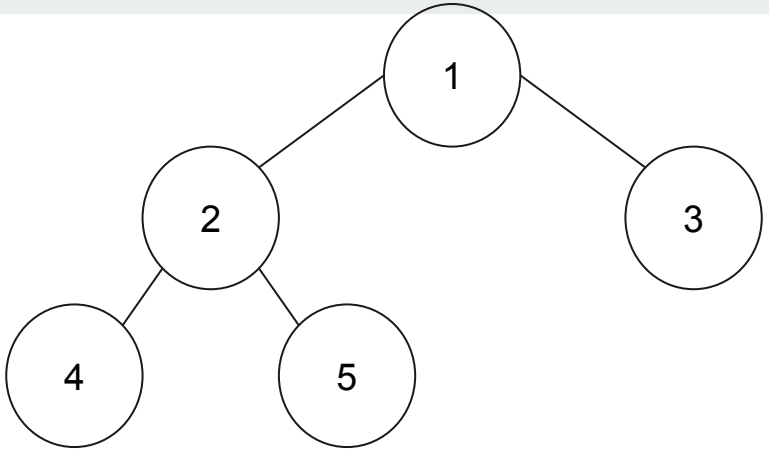
nivel	0	1	2	1	2	1	0	1	0

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1



```
int LCA(Node *root, int u, int v)
{
    memset(primeraOcurrencia, -1, sizeof(int)*(V+1));
    ind = 0;
    eulerTour(root, 0);
    int *st = constructST(nivel, 2*V-1);

    if (primeraOcurrencia[u]>primeraOcurrencia[v])
       swap(u, v);

    int qs = primeraOcurrencia[u];
    int qe = primeraOcurrencia[v];

    int index = RMQ(st, 2*V-1, qs, qe);
    return euler[index];
}
```

u = 4 v = 3



euler	1	2	4	2	5	2	1	3	1
nivel	0	1	2	1	2	1	0	1	0

qe qs

primera_ocurrencia	0	1	7	2	4

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

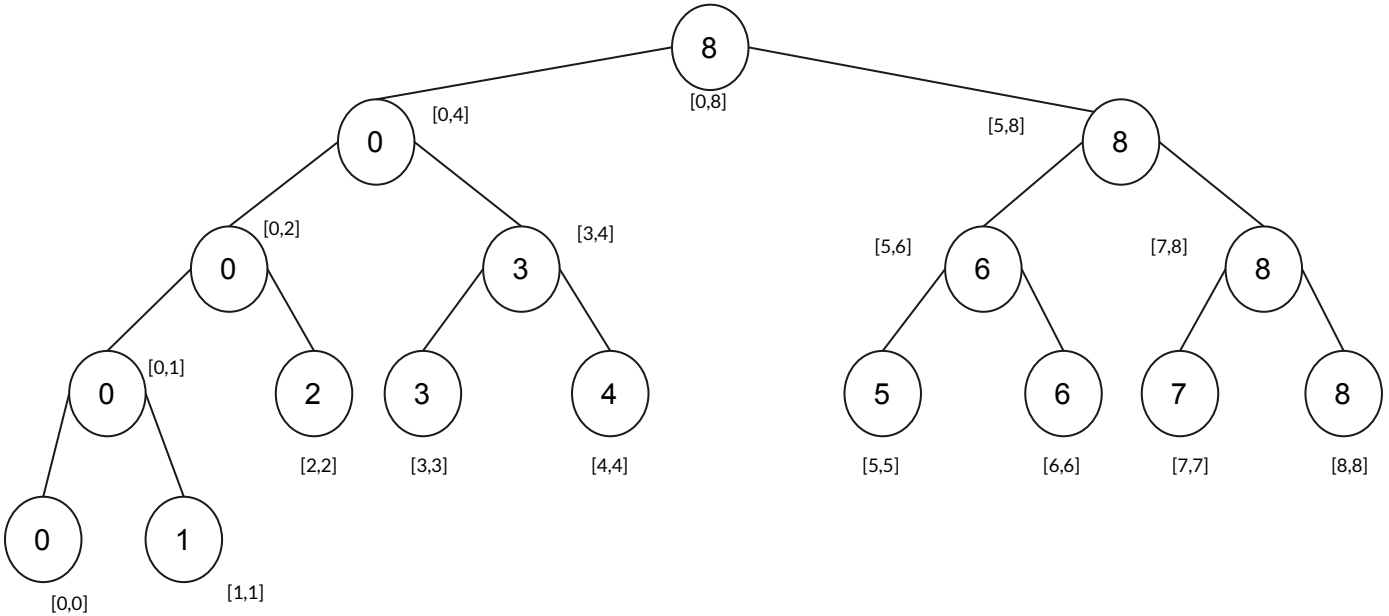
```
int RMQ(int *st, int n, int qs, int qe)
{
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Entrada invalida");
        return -1;
    }
    return RMQUtil(0, 0, n-1, qs, qe, st);
}
```

```
int RMQUtil(int index, int low, int high, int qlow, int qhigh, int *st)
{
    if (qlow <= low && qhigh >= high)///total overlap
        return st[index];
    else if (high < qlow || low > qhigh)///no overlap
        return -1;
    ///partial overlap
    int mid = (low + high)/2;
    int q1 = RMQUtil(2*index+1, low, mid, qlow, qhigh, st);
    int q2 = RMQUtil(2*index+2, mid+1, high, qlow, qhigh, st);
    if (q1==-1) return q2;
    else if (q2==-1) return q1;
    return (nivel[q1] < nivel[q2]) ? q1 : q2;
}
```

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

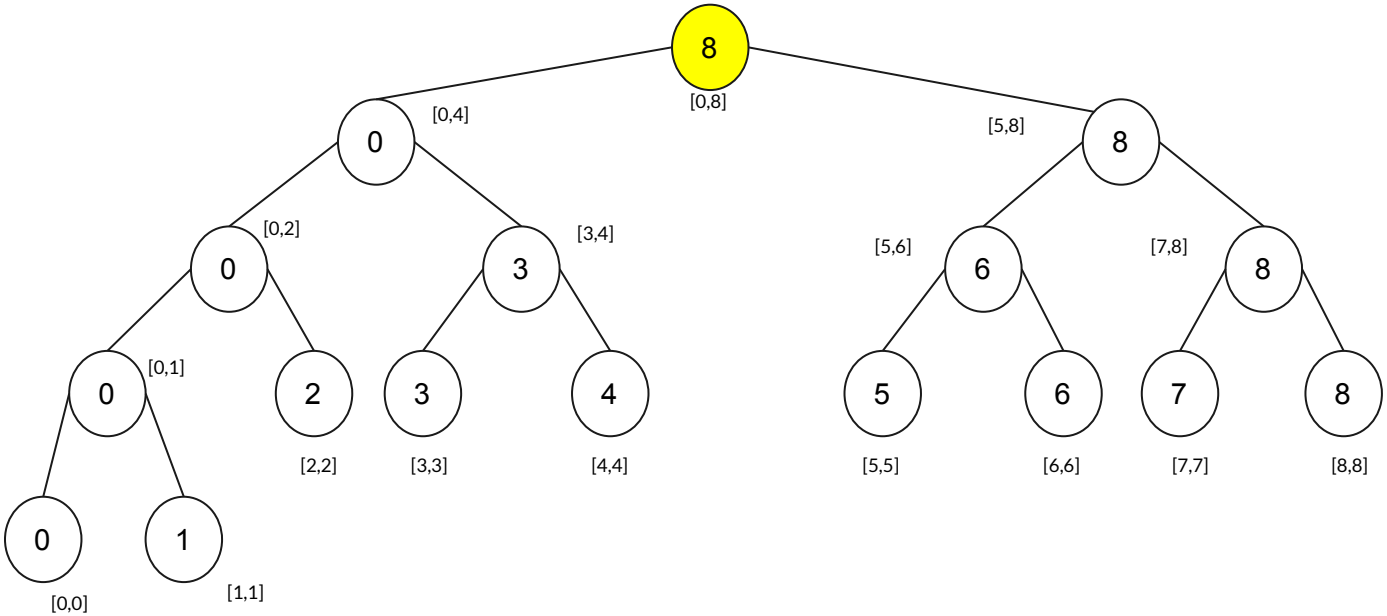


- Partial Overlap
- Total Overlap
- No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

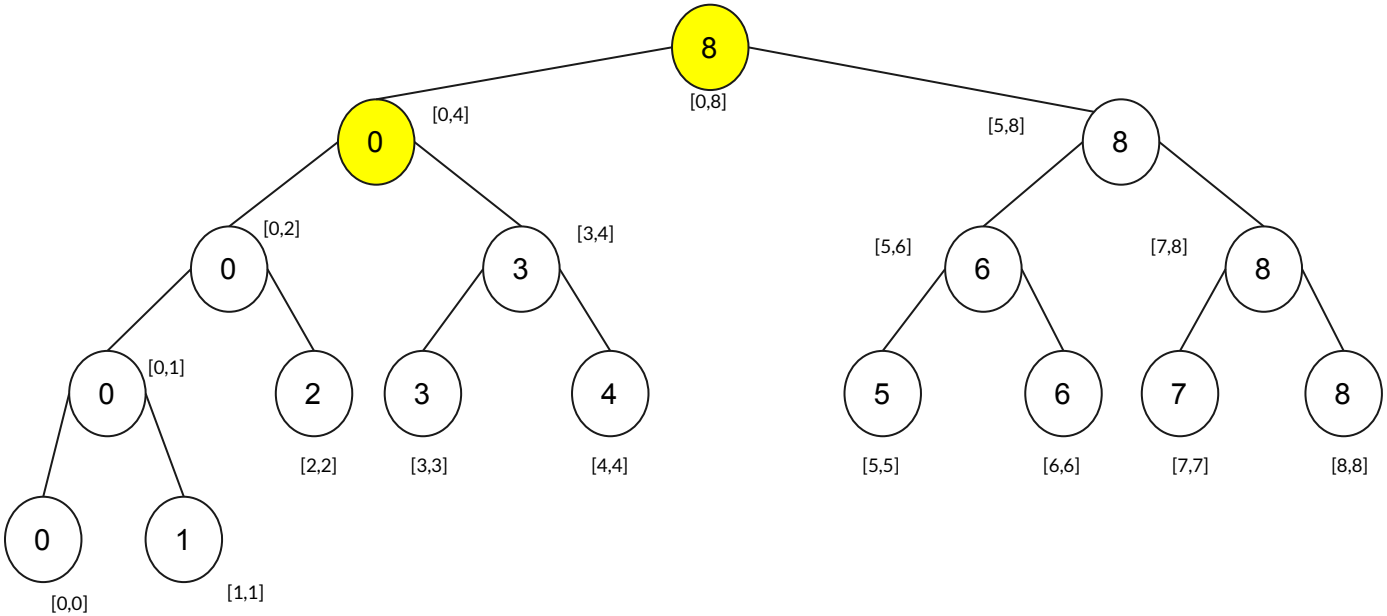


- Partial Overlap
- Total Overlap
- No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

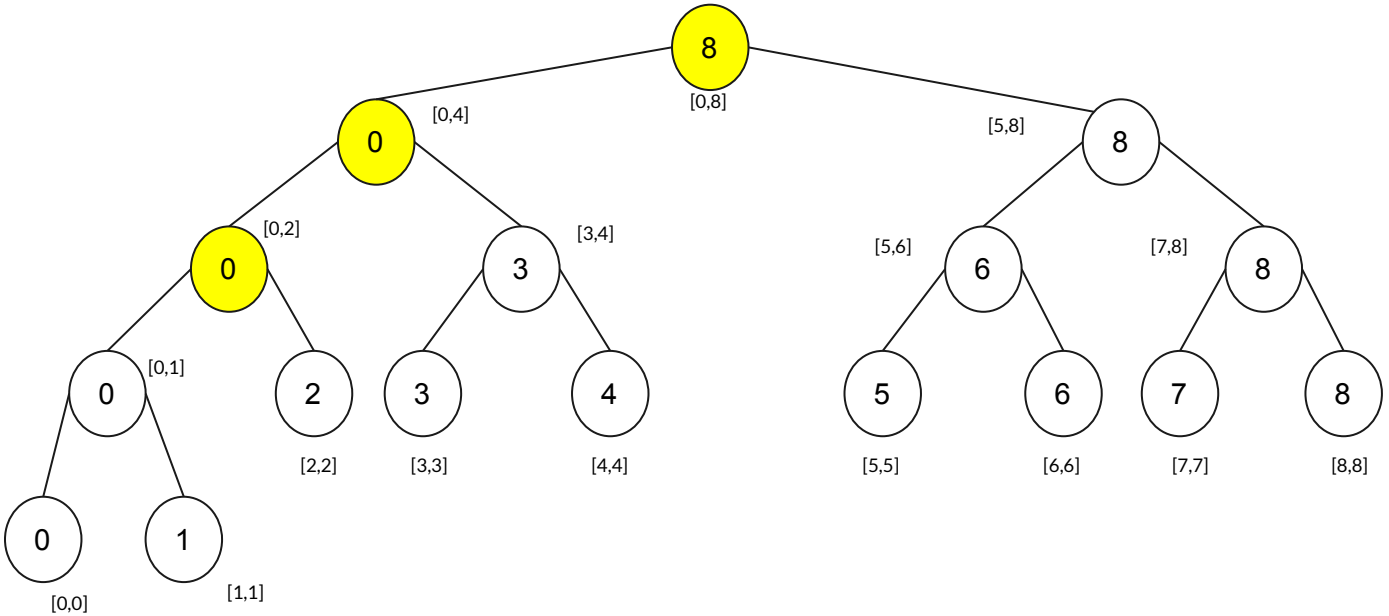


- Partial Overlap
- Total Overlap
- No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

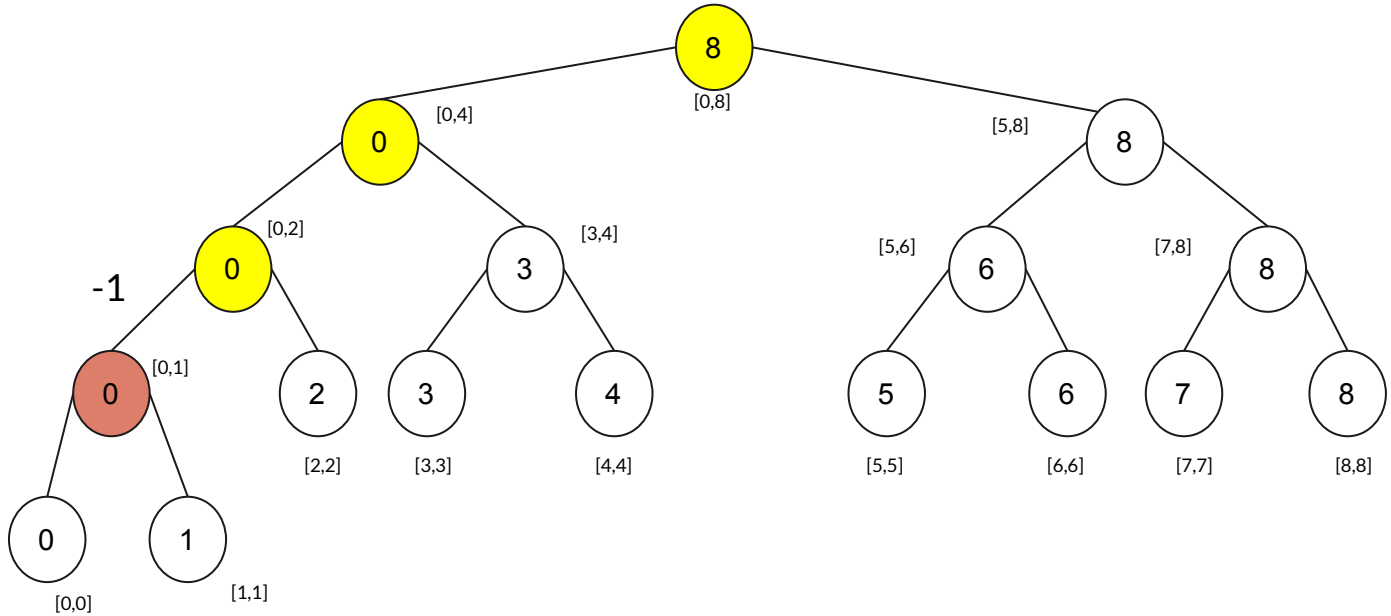


- Partial Overlap
- Total Overlap
- No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

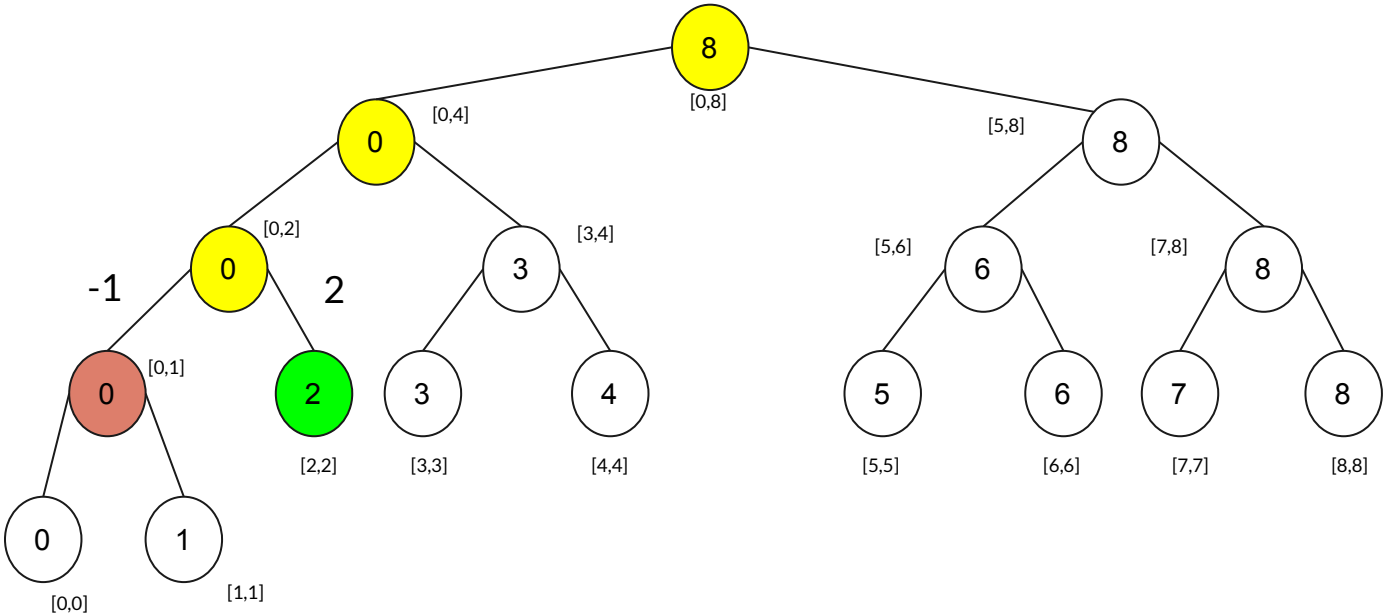


- Partial Overlap
- Total Overlap
- No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

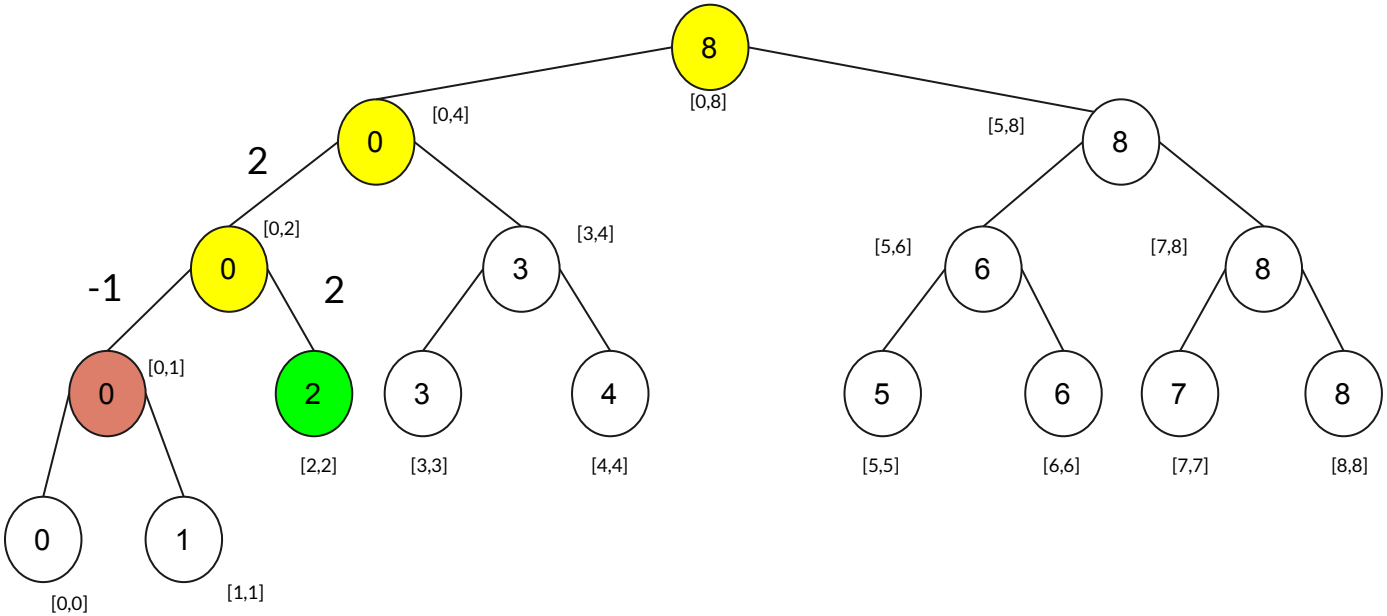


Partial Overlap
Total Overlap
No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

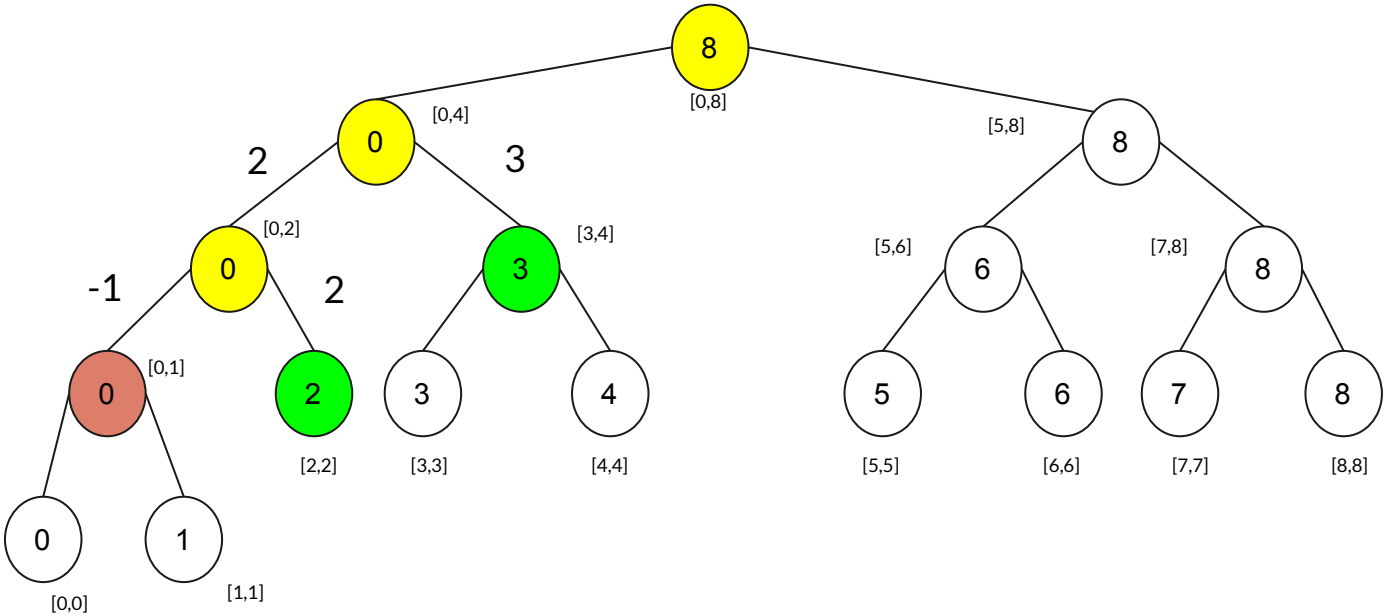


- Partial Overlap
- Total Overlap
- No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

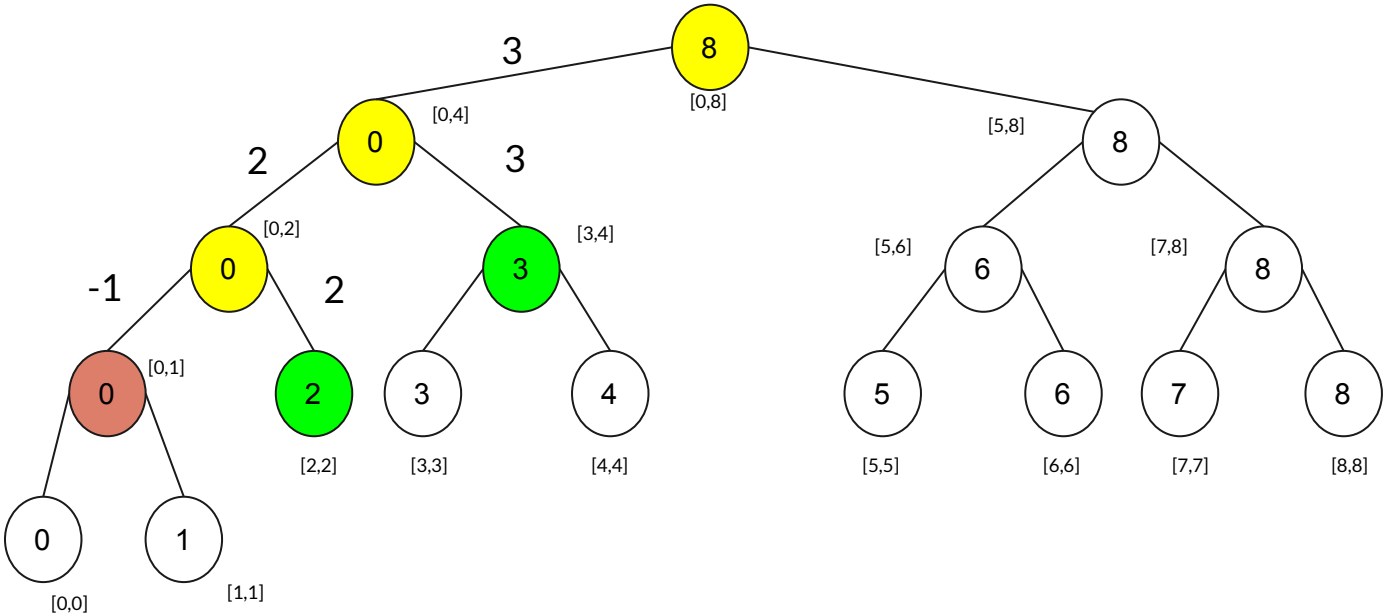


- Partial Overlap
- Total Overlap
- No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

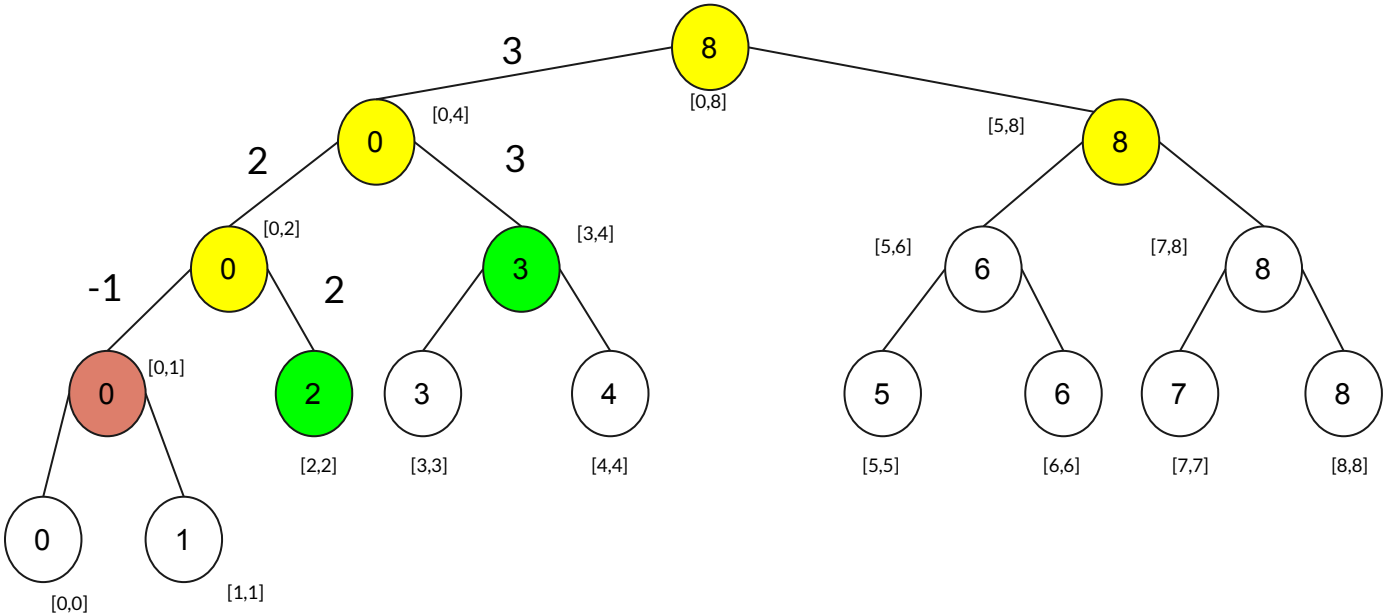


- Partial Overlap
- Total Overlap
- No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

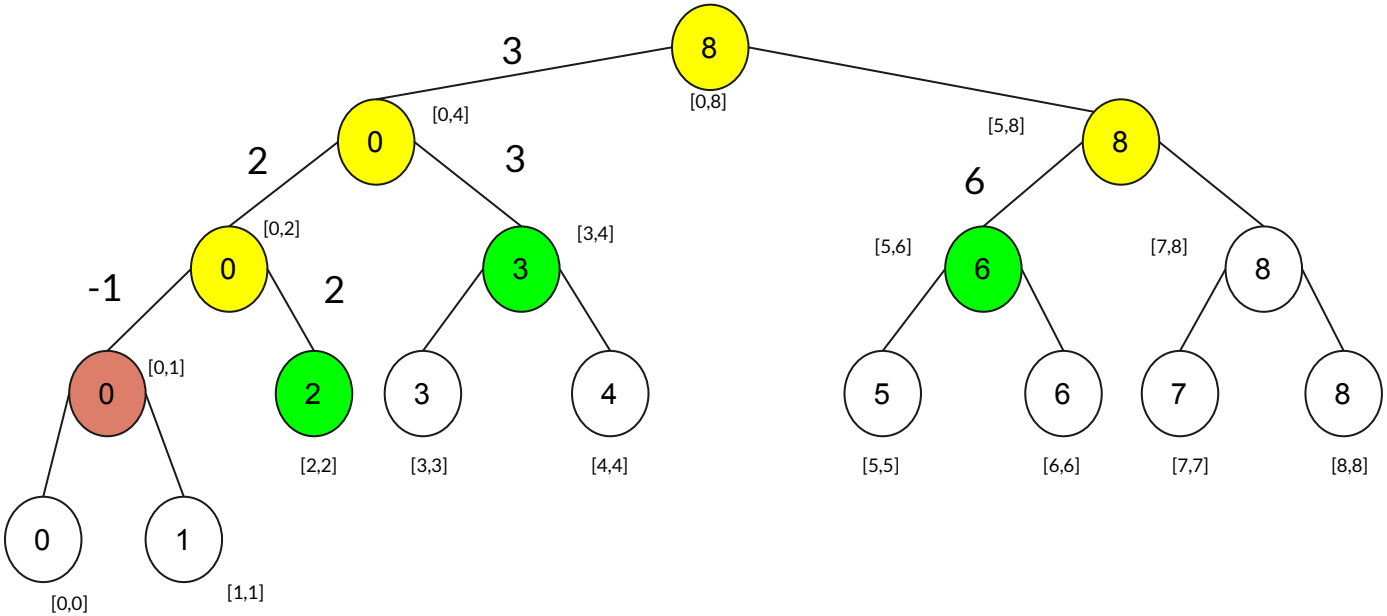


- Partial Overlap
- Total Overlap
- No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

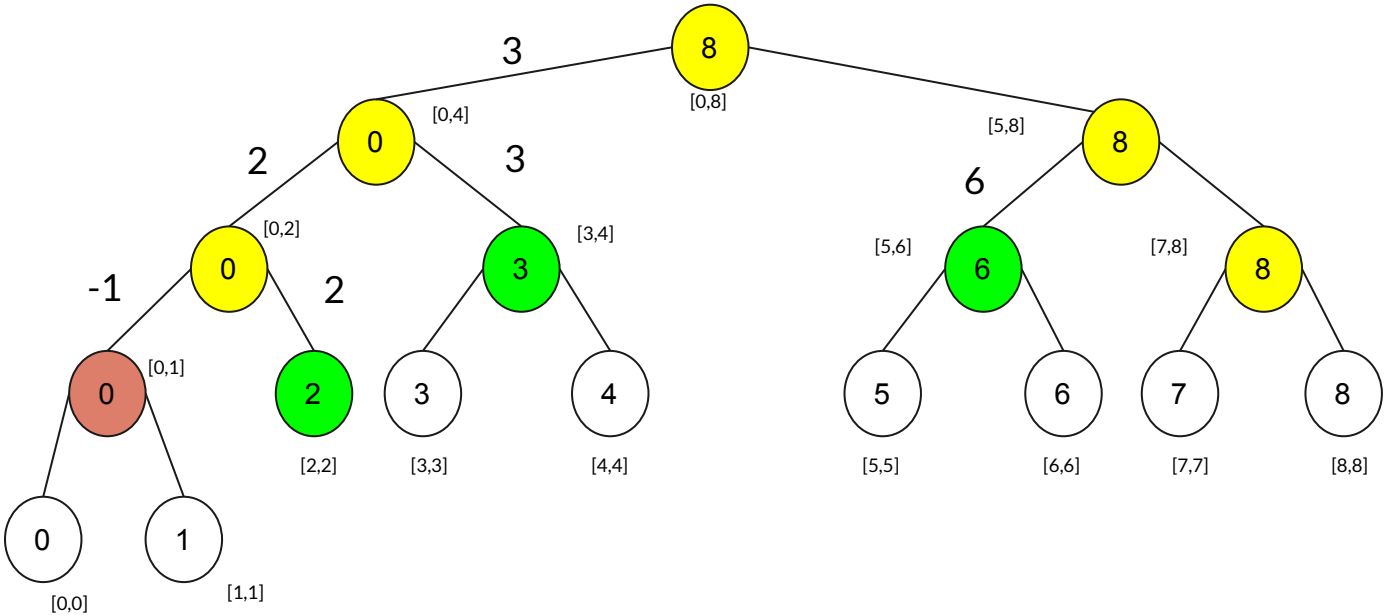


- Partial Overlap
- Total Overlap
- No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

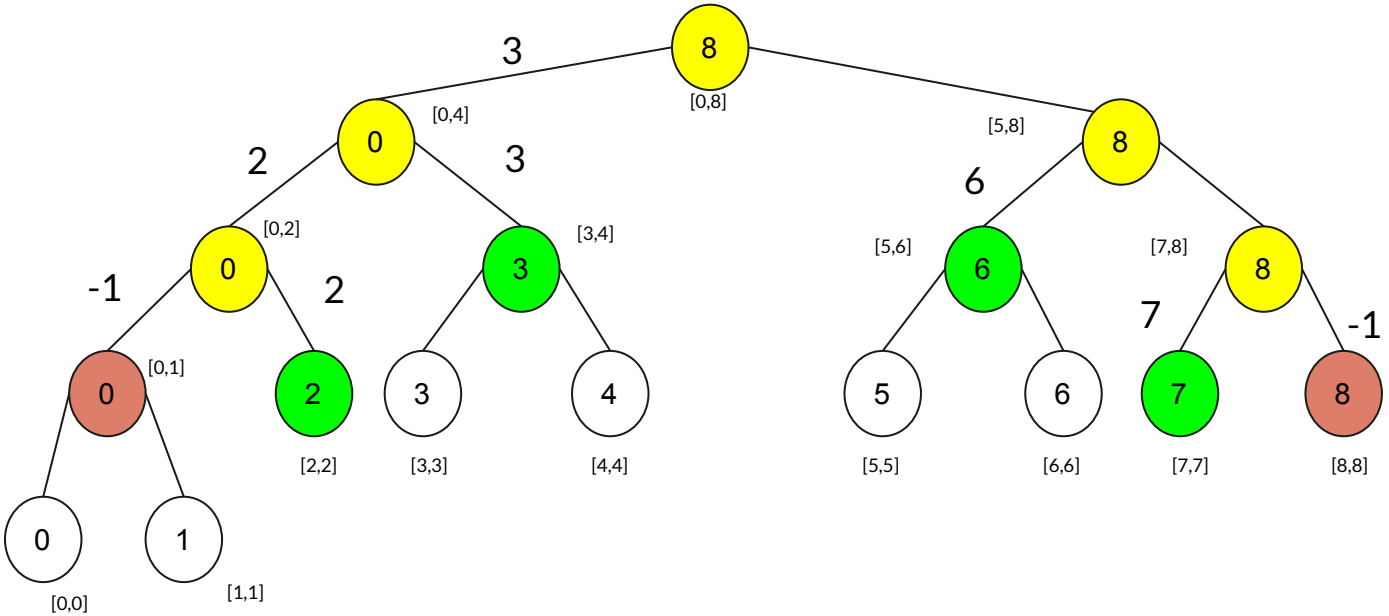


- Partial Overlap
- Total Overlap
- No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1

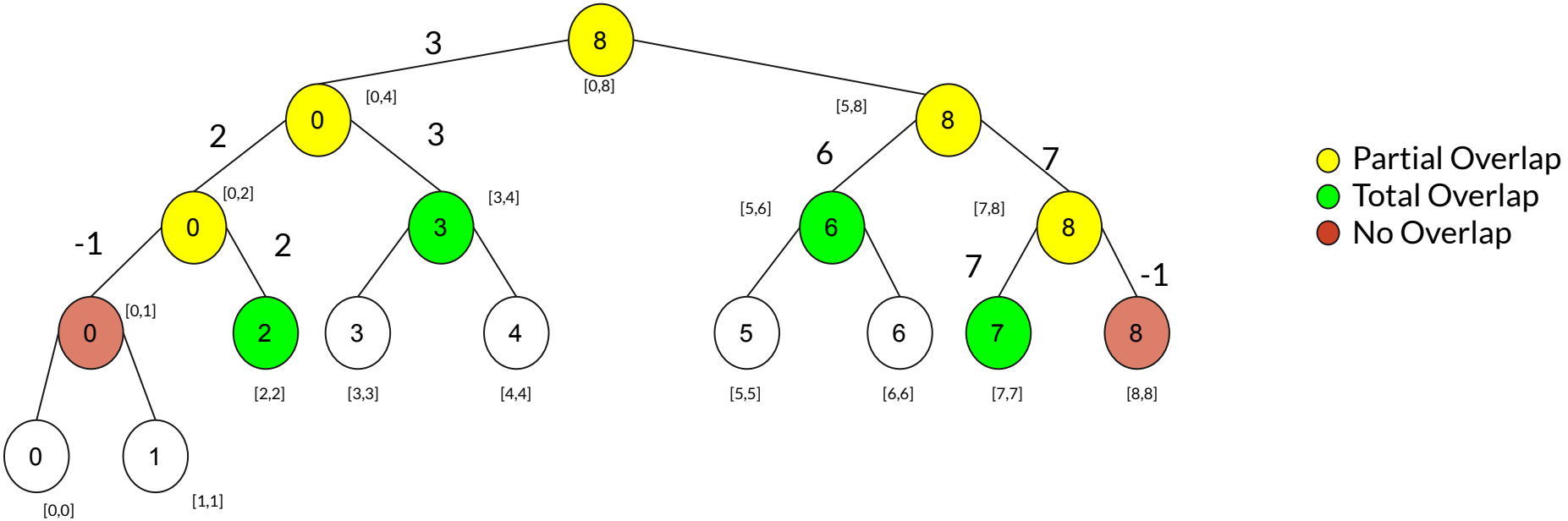


Partial Overlap
Total Overlap
No Overlap

nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

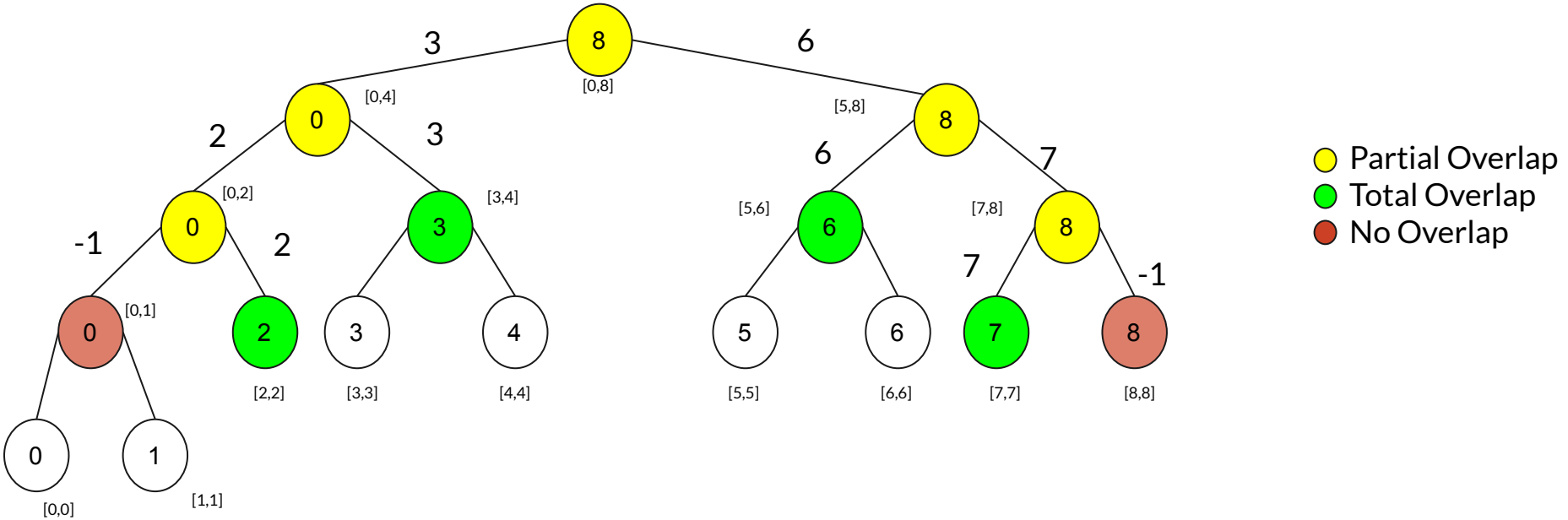
STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1



nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

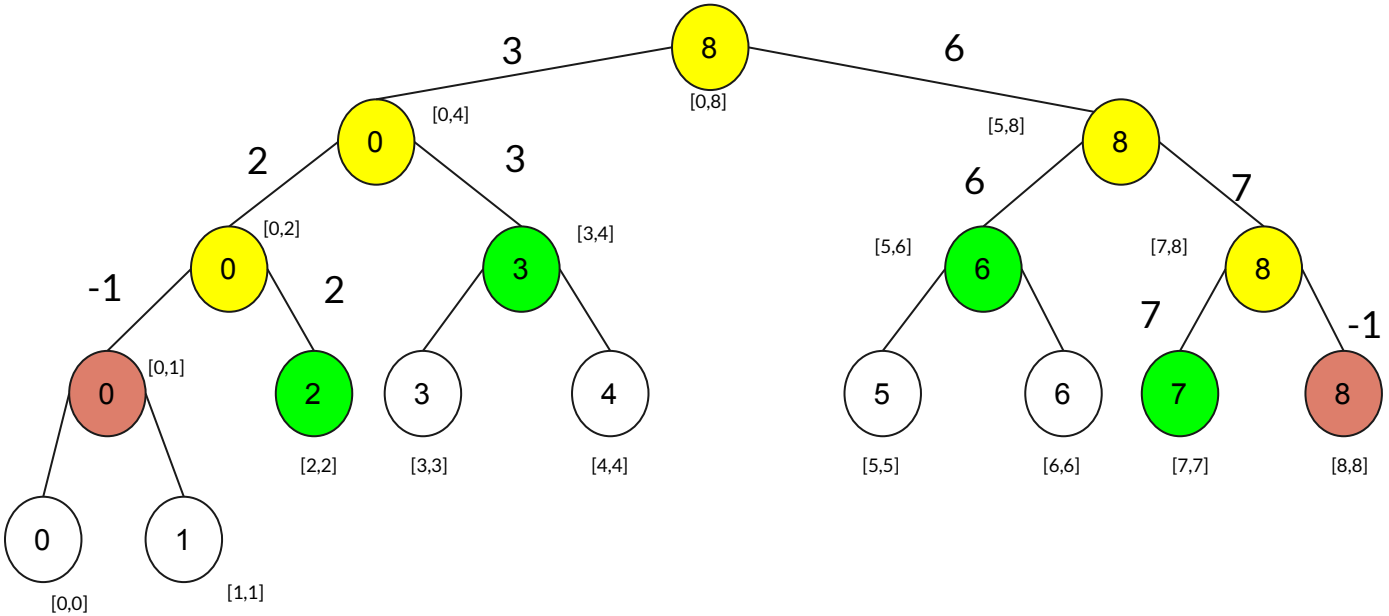
STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1



nivel	0	1	2	1	2	1	0	1	0

qs = 2
qe = 7

STree	8	0	8	0	3	6	8	0	2	3	4	5	6	7	8	0	1



index = 6

- Partial Overlap
- Total Overlap
- No Overlap

```
int RMQ(int *st, int n, int qs, int qe)
{
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Entrada invalida");
        return -1;
    }
    return RMQUtil(0, 0, n-1, qs, qe, st);
}
```

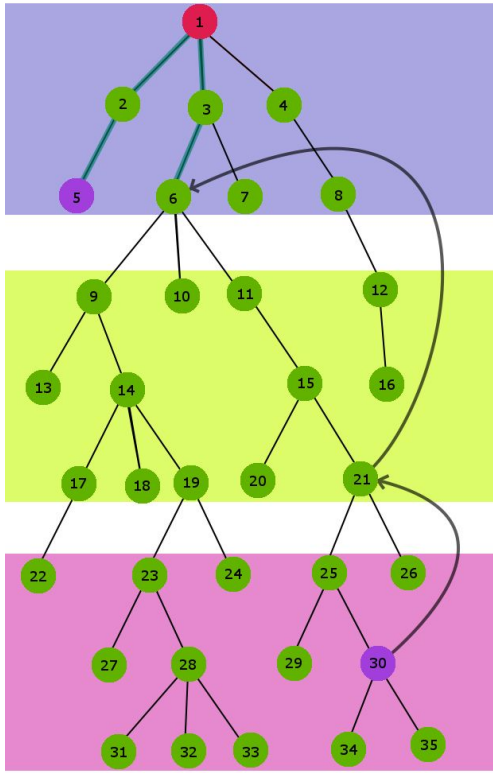
```
int RMQUtil(int index, int low, int high, int qlow, int qhigh, int *st)
{
    if (qlow <= low && qhigh >= high)///total overlap
        return st[index];
    else if (high < qlow || low > qhigh)///no overlap
        return -1;
    ///partial overlap
    int mid = (low + high)/2;
    int q1 = RMQUtil(2*index+1, low, mid, qlow, qhigh, st);
    int q2 = RMQUtil(2*index+2, mid+1, high, qlow, qhigh, st);
    if (q1==-1) return q2;
    else if (q2==-1) return q1;
    return (nivel[q1] < nivel[q2]) ? q1 : q2;
}
```

```
int LCA(Node *root, int u, int v)
{
    memset(primeraOcurrencia, -1, sizeof(int)*(V+1));
    ind = 0;
    eulerTour(root, 0);
    int *st = constructST(nivel, 2*V-1);

    if (primeraOcurrencia[u]>primeraOcurrencia[v])
       swap(u, v);

    int qs = primeraOcurrencia[u];
    int qe = primeraOcurrencia[v];

    int index = RMQ(st, 2*V-1, qs, qe);
    return euler[index];
}
```

index = 6

euler	1	2	4	2	5	2	1	3	1

```
El LCA del nodo 4 y el nodo 3 es el nodo 1.
```



2*35 -1 = 69

2*35 -1 = 69
st_size = 2*2^(log(69))-1 = 127
7 niveles

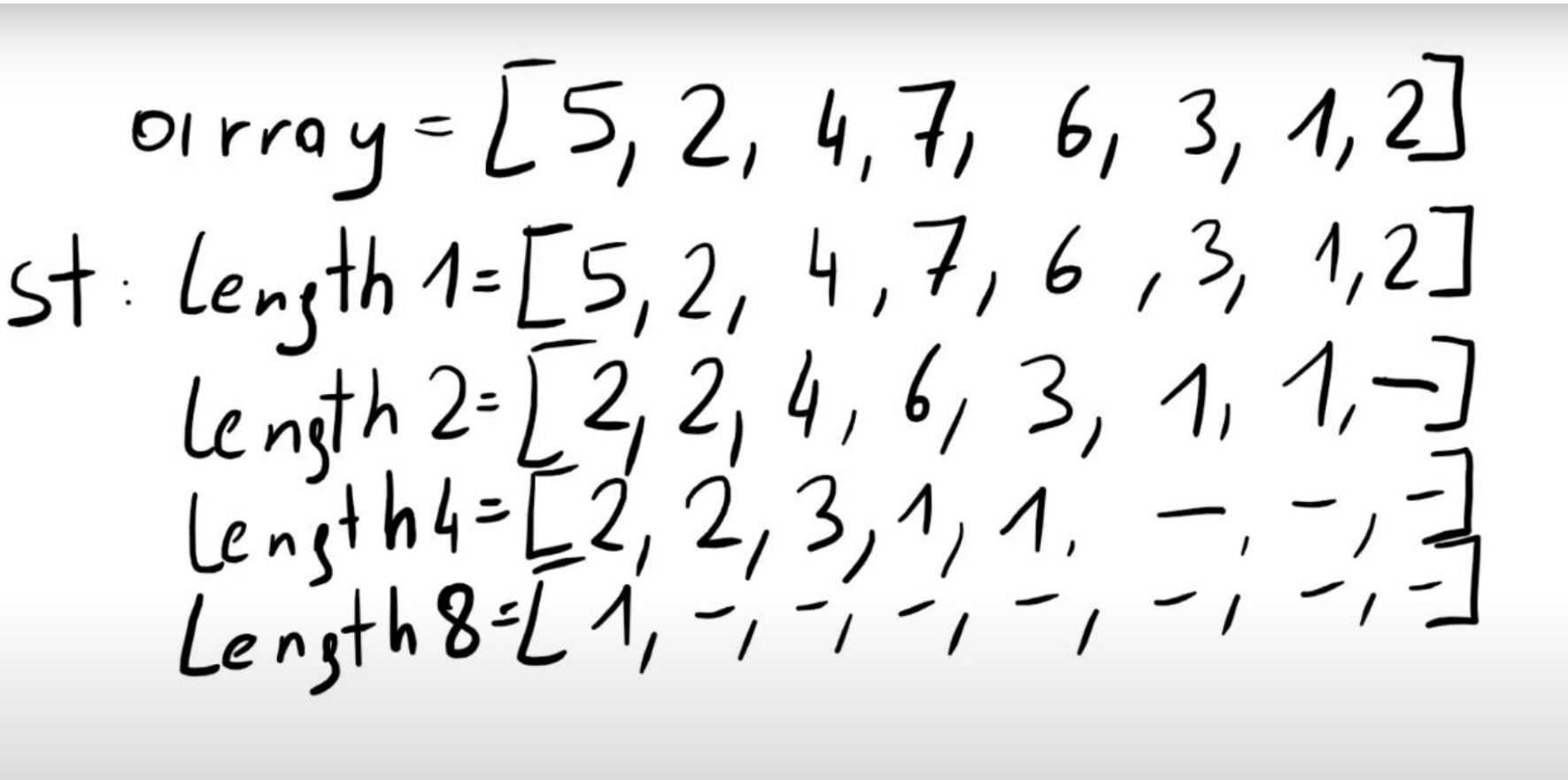
RMQ + SPARSE TABLE

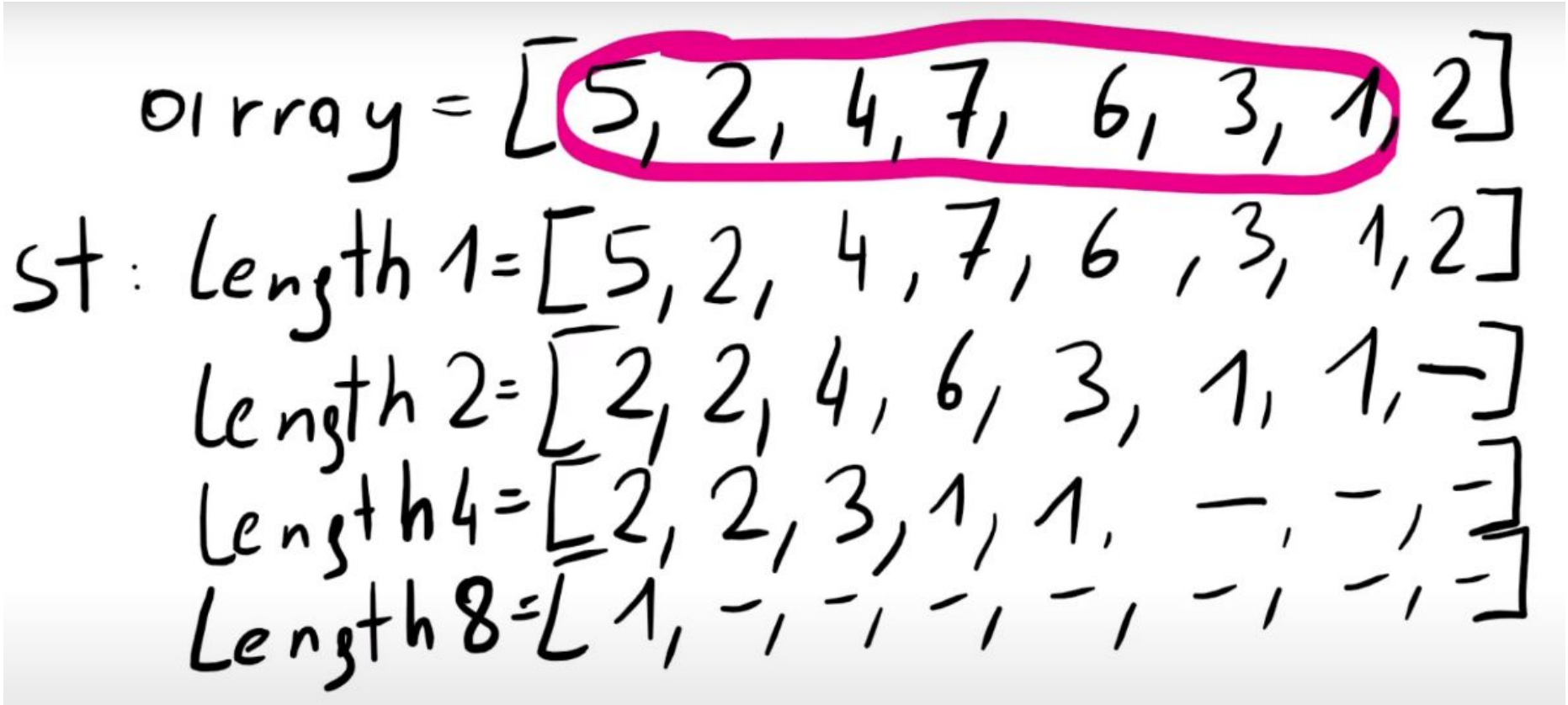
¿Cómo funciona el Sparse Table?

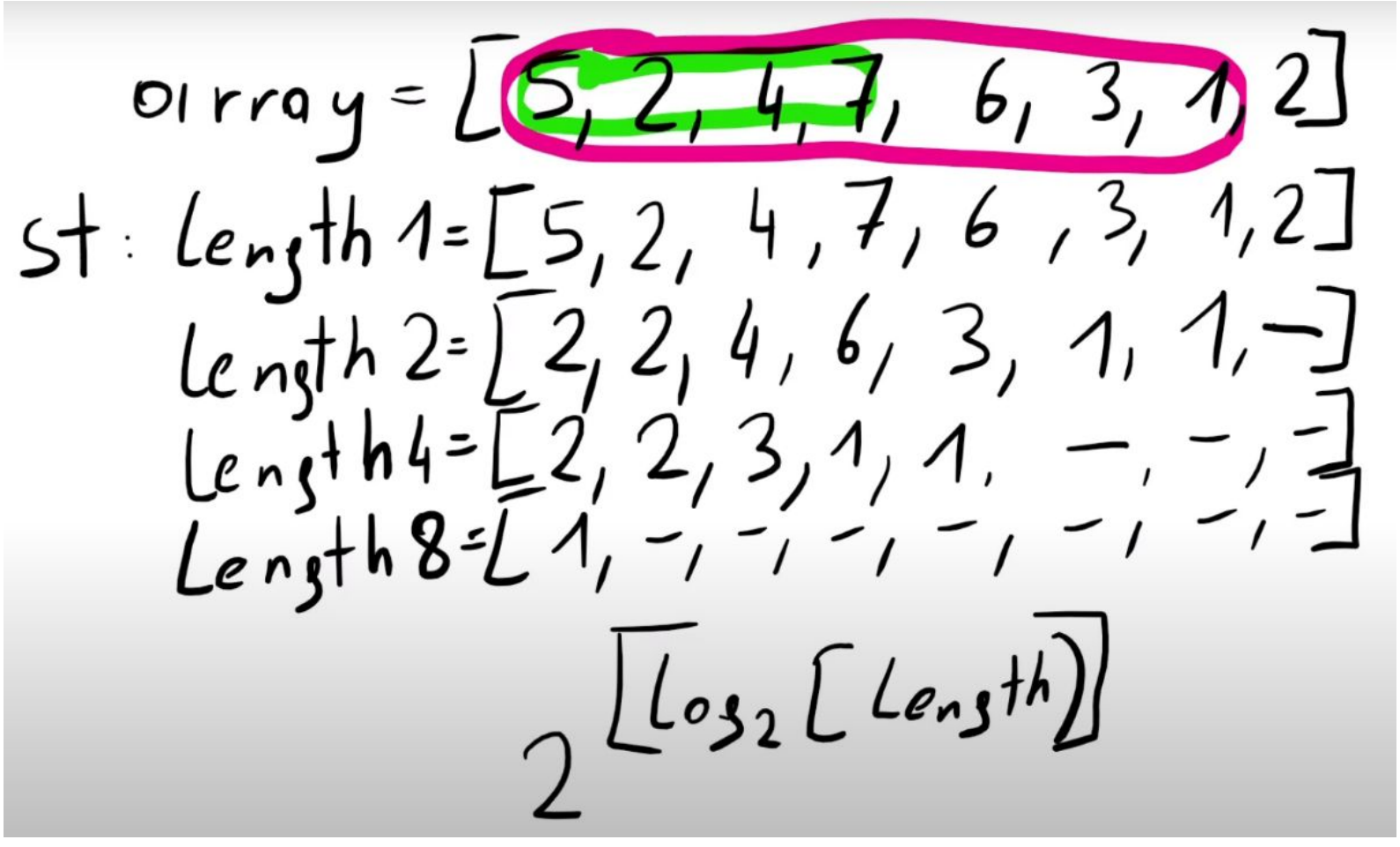
Sirve para almacenar el mínimo o máximo de un rango de elementos de un arreglo.

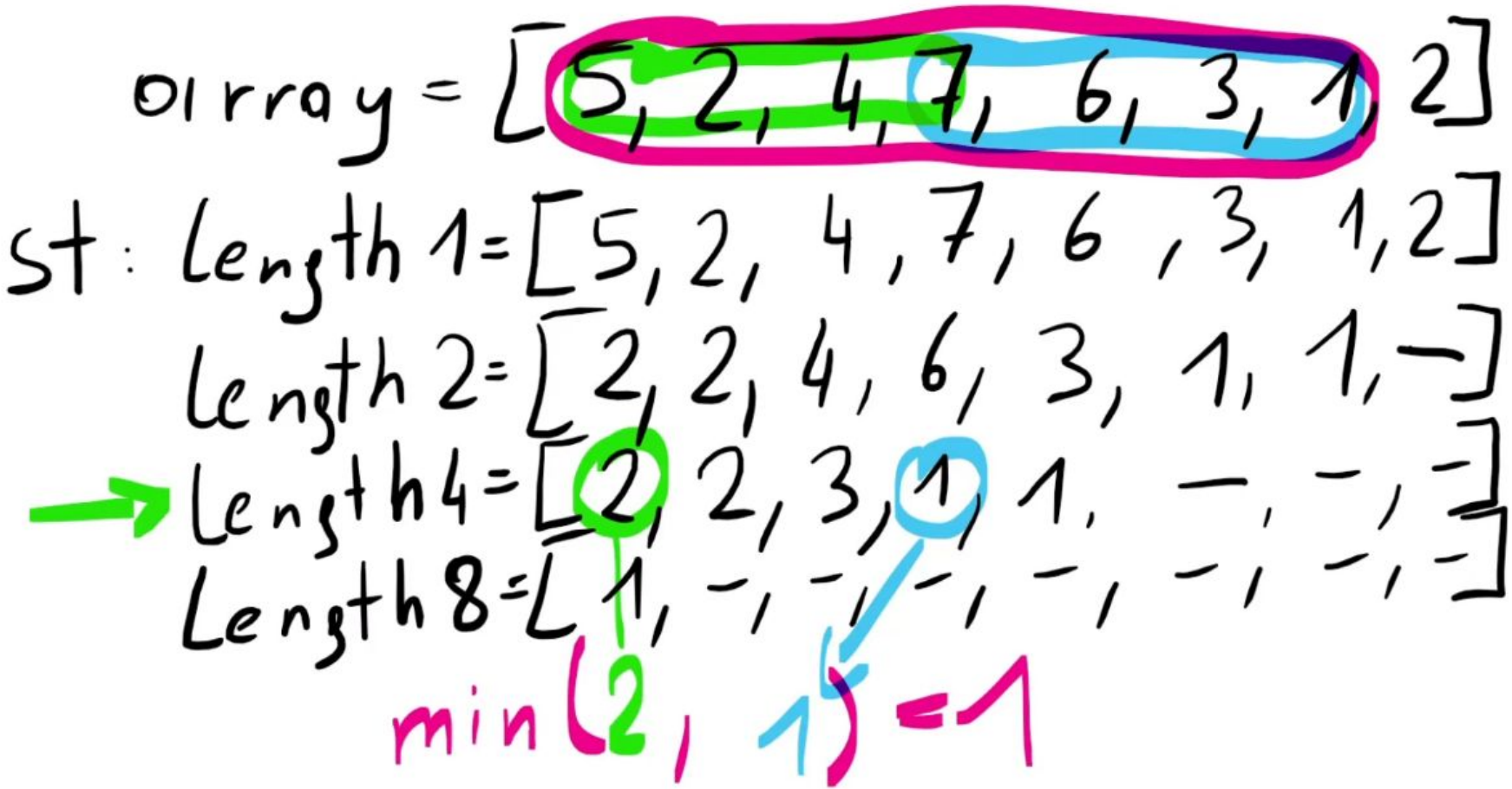
Se almacena los subrangos en log(n) + 1 arreglos.

Se extrae de acuerdo al correspondiente logaritmo base 2 del rango solicitado.

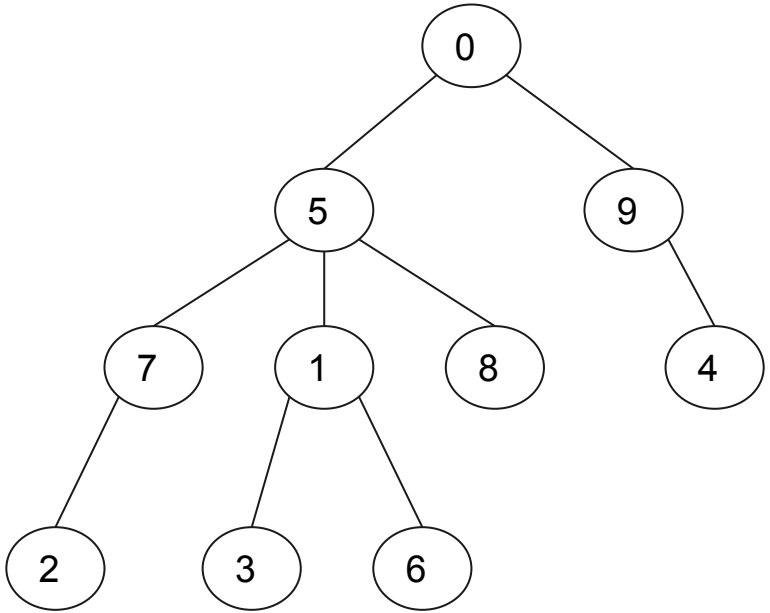

array = [5, 2, 4, 7, 6, 3, 1, 2]
st: length 1 = [5, 2, 4, 7, 6, 3, 1, 2]
length 2 = [2, 2, 4, 6, 3, 1, 1, -]
length 4 = [2, 2, 3, 1, 1, -, -, -]
length 8 = [1, -, -, -, -, -, -, -]


array = [5, 2, 4, 7, 6, 3, 1, 2]
st: length 1 = [5, 2, 4, 7, 6, 3, 1, 2]
length 2 = [2, 2, 4, 6, 3, 1, 1, -]
length 4 = [2, 2, 3, 1, 1, -, -, -]
length 8 = [1, -, -, -, -, -, -, -]

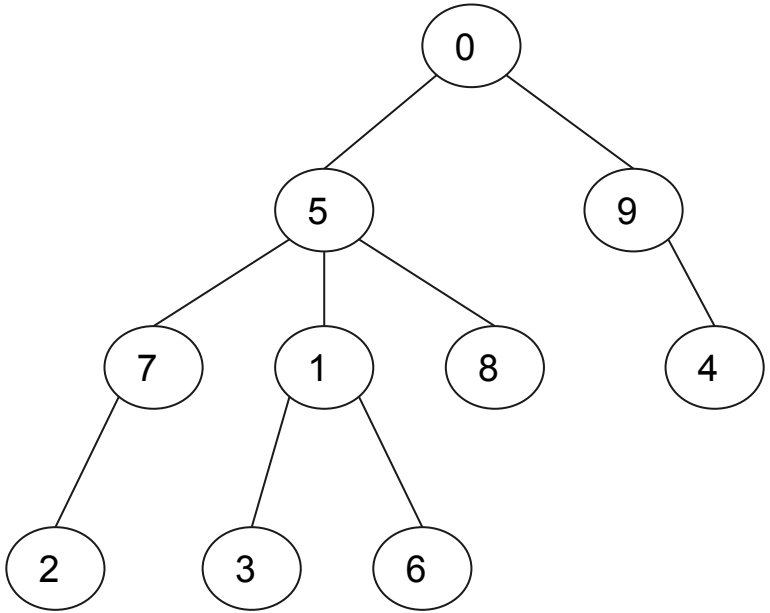

array = [5, 2, 4, 7, 6, 3, 1, 2]
st: length 1 = [5, 2, 4, 7, 6, 3, 1, 2]
length 2 = [2, 2, 4, 6, 3, 1, 1, -]
length 4 = [2, 2, 3, 1, 1, -, -, -]
length 8 = [1, -, -, -, -, -, -, -]
$2^{\lceil \log_2 \lceil length \rceil \rceil}$


array = [5, 2, 4, 7, 6, 3, 1, 2]
st: Length 1 = [5, 2, 4, 7, 6, 3, 1, 2]
Length 2 = [2, 2, 4, 6, 3, 1, 1, –]
Length 4 = [2, 2, 3, 1, 1, –, –, –]
Length 8 = [1, –, –, –, –, –, –, –]
min (2, 1) = 1

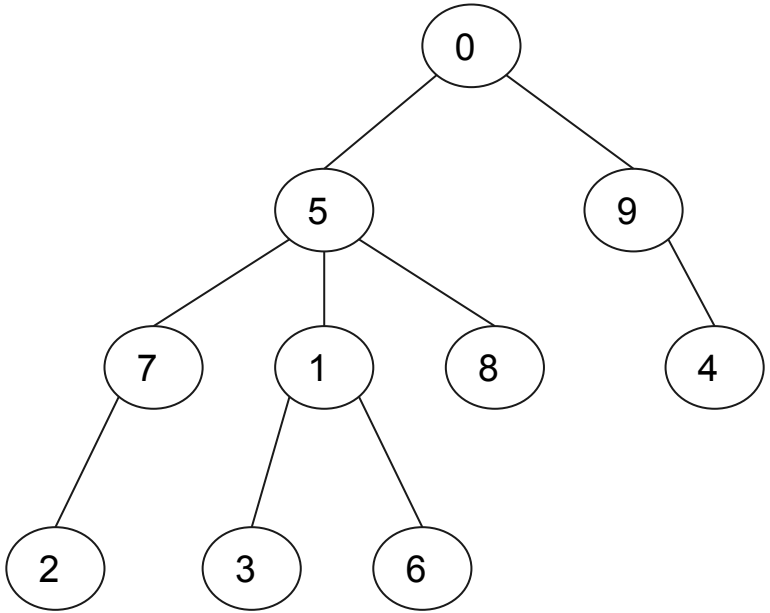
```
vector<pair<int, int>> edges = {{0, 5}, {5, 7}, {7, 2}, {5, 1},
{1, 3}, {1, 6}, {5, 8}, {0, 9}, {9, 4}};

adj.resize(nodes);
for (auto edge : edges) {
    adj[edge.first].push_back(edge.second);
    adj[edge.second].push_back(edge.first);
}
```



```
first_encounter.resize(nodes);
dfs_euler_tour(0, -1);
RMQ rmq(euler_tour);
```

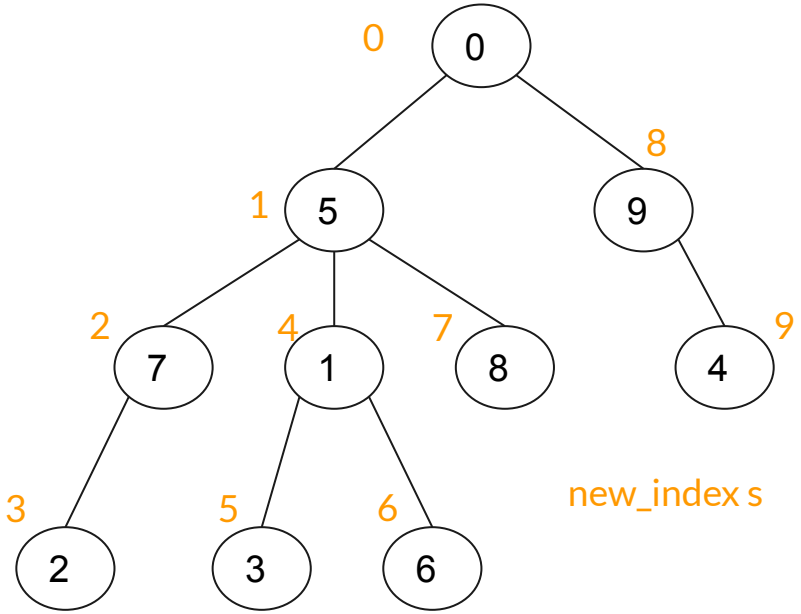


```
void dfs_euler_tour(int v, int p) {
    int new_index = new_to_old.size();
    new_to_old.push_back(v);
    first_encounter[v] = euler_tour.size();
    euler_tour.push_back(new_index);

    for (int u : adj[v]) {
        if (u == p)
            continue;

        dfs_euler_tour(u, v);
        euler_tour.push_back(new_index);
    }
}
```



new_to_old:{0 5 7 2 1 3 6 8 9 4}

```
void dfs_euler_tour(int v, int p) {
    int new_index = new_to_old.size();
    new_to_old.push_back(v);
    first_encounter[v] = euler_tour.size();
    euler_tour.push_back(new_index);

    for (int u : adj[v]) {
        if (u == p)
            continue;

        dfs_euler_tour(u, v);
        euler_tour.push_back(new_index);
    }
}
```



Euler	0	1	2	3	2	1	4	5	4	6	4	1	7	1	0	8	9	8	0

First_encounter	0	6	3	7	16	1	9	2	12	15

```
first_encounter.resize(nodes);
dfs_euler_tour(0, -1);
RMQ rmq(euler_tour);
```

```
RMQ(vector<int> euler_) {
    log_table.assign(euler_.size() + 1, 0);
    for (int i = 2; i < log_table.size(); i++)
        log_table[i] = log_table[i/2] + 1;

    sparse_table.assign(log_table.back() + 1, vector<int>(euler_.size()));
    sparse_table[0] = euler_;
    for (int row = 1; row < sparse_table.size(); row++) {
        for (int i = 0; i + (1 << row) <= euler_.size(); i++) {
            sparse_table[row][i] = min(sparse_table[row-1][i], sparse_table[row-1][i+(1<<(row-1))]);
        }
    }
}
```

```
RMQ(vector<int> euler_) {
    log_table.assign(euler_.size() + 1, 0);
    for (int i = 2; i < log_table.size(); i++)
        log_table[i] = log_table[i/2] + 1;

    sparse_table.assign(log_table.back() + 1, vector<int>(euler_.size()));
    sparse_table[0] = euler_;
    for (int row = 1; row < sparse_table.size(); row++) {
        for (int i = 0; i + (1 << row) <= euler_.size(); i++) {
            sparse_table[row][i] = min(sparse_table[row-1][i], sparse_table[row-1][i+(1<<(row-1))]);
        }
    }
}
```

log_table

-	0	1	1	2	2	2	2	3	3	3	3	3	3	3	4	4	4	4

Euler	0	1	2	3	2	1	4	5	4	6	4	1	7	1	0	8	9	8	0

len 1	0	1	2	3	2	1	4	5	4	6	4	1	7	1	0	8	9	8	0
len 2	0	1	2	2	1	1	4	4	4	4	1	1	1	0	0	8	8	0	
len 4	0	1	1	1	1	1	4	4	1	1	1	0	0	0	0	0			
len 8	0	1	1	1	1	1	1	0	0	0	0	0							
len 16	0	0	0	0															

```
RMQ(vector<int> euler_) {
    log_table.assign(euler_.size() + 1, 0);
    for (int i = 2; i < log_table.size(); i++)
        log_table[i] = log_table[i/2] + 1;

    sparse_table.assign(log_table.back() + 1, vector<int>(euler_.size()));
    sparse_table[0] = euler_;
    for (int row = 1; row < sparse_table.size(); row++) {
        for (int i = 0; i + (1 << row) <= euler_.size(); i++) {
            sparse_table[row][i] = min(sparse_table[row-1][i], sparse_table[row-1][i+(1<<(row-1))]);
        }
    }
}
```

log_table

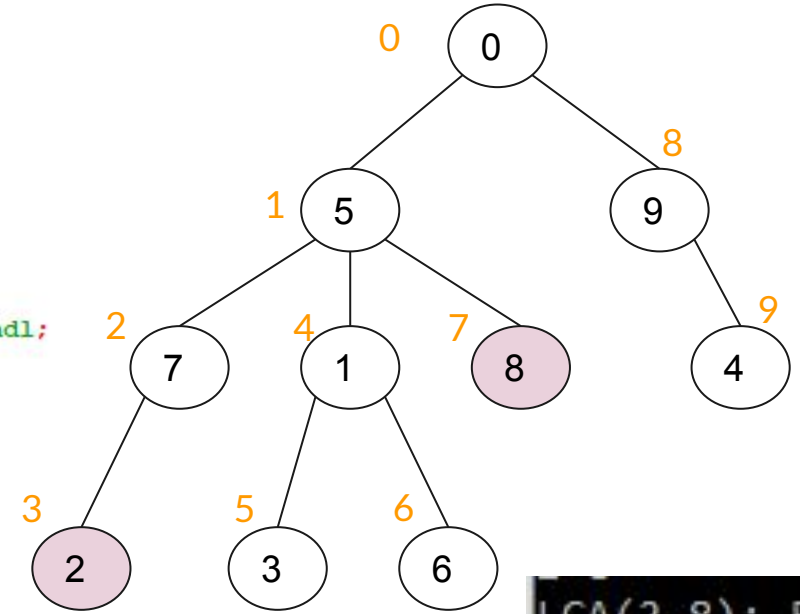
-	0	1	1	2	2	2	2	3	3	3	3	3	3	3	4	4	4	4

Euler	0	1	2	3	2	1	4	5	4	6	4	1	7	1	0	8	9	8	0

len 1	0	1	2	3	2	1	4	5	4	6	4	1	7	1	0	8	9	8	0
len 2	0	1	2	2	1	1	4	4	4	4	1	1	1	0	0	8	8	0	
len 4	0	1	1	1	1	1	4	4	1	1	1	0	0	0	0	0			
len 8	0	1	1	1	1	1	1	0	0	0	0	0							
len 16	0	0	0	0															

```
int u, v;
cin >> u >> v;
int fe1 = first_encounter[u];
int fe2 = first_encounter[v];
if (fe1 > fe2)
    swap(fe1, fe2);
int LCA_new_index = rmq.minimum(fe1, fe2);
int LCA_old_index = new_to_old[LCA_new_index];
//cout << "LCA_new_index: " << LCA_new_index << endl;
cout << "LCA(" << u << "," << v << "): " << LCA_old_index << endl;
```

u = 2, v= 8




```
LCA(2,8): 5
```

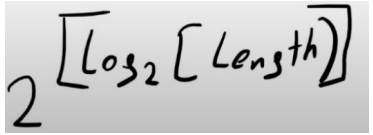
Se usan los nuevos índices

First_encounter	0	6	3	7	16	1	9	2	12	15

Euler	0	1	2	3	2	1	4	5	4	6	4	1	7	1	0	8	9	8	0

```
int minimum(int l, int r) {
    int log = log_table[r - l];
    return min(sparse_table[log][l], sparse_table[log][r - (1 << log)]);
}
```

r = 3, L = 12

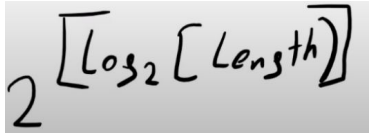


=8

log_table	-	0	1	1	2	2	2	2	3	3	3	3	3	3	3	3	4	4	4	4
Euler	0	1	2	3	2	1	4	5	4	6	4	1	7	1	0	8	9	8	0	
len 1	0	1	2	3	2	1	4	5	4	6	4	1	7	1	0	8	9	8	0	
len 2	0	1	2	2	1	1	4	4	4	4	1	1	1	0	0	8	8	0		
len 4	0	1	1	1	1	1	4	4	1	1	1	0	0	0	0	0				
⇨ len 8	0	1	1	1	1	1	1	0	0	0	0	0								
len 16	0	0	0	0																

```
int minimum(int l, int r) {
    int log = log_table[r - l];
    return min(sparse_table[log][l], sparse_table[log][r - (1 << log)]);
}
```

r = 3, L = 12

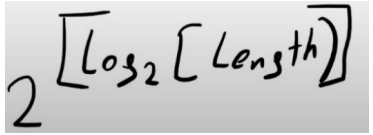


=8

log_table	-	0	1	1	2	2	2	2	3	3	3	3	3	3	3	4	4	4	4
Euler	0	1	2	3	2	1	4	5	4	6	4	1	7	1	0	8	9	8	0
len 1	0	1	2	3	2	1	4	5	4	6	4	1	7	1	0	8	9	8	0
len 2	0	1	2	2	1	1	4	4	4	4	1	1	1	0	0	8	8	0	
len 4	0	1	1	1	1	1	4	4	1	1	1	0	0	0	0	0			
len 8	0	1	1	1	1	1	1	0	0	0	0	0							
len 16	0	0	0	0															

```
int minimum(int l, int r) {
    int log = log_table[r - l];
    return min(sparse_table[log][l], sparse_table[log][r - (1 << log)]);
}
```

r = 3, L = 12

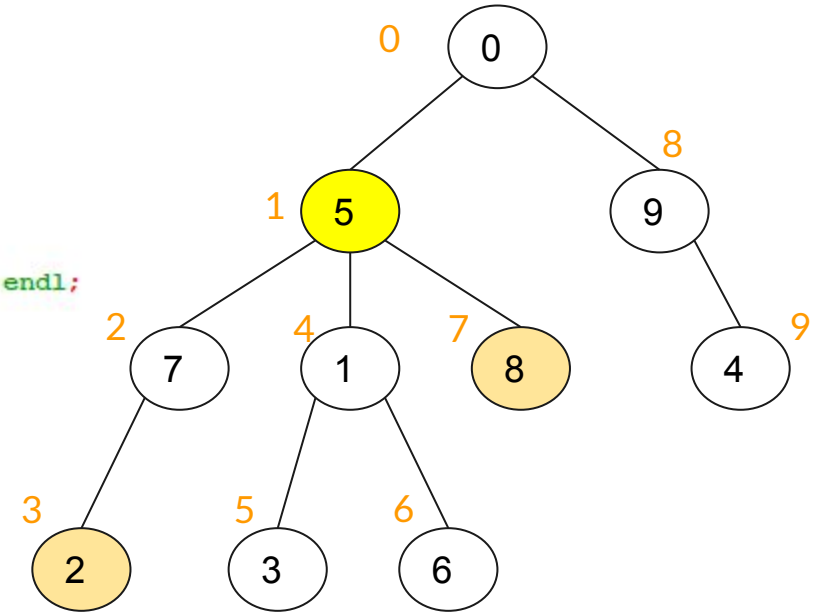


=8

log_table	-	0	1	1	2	2	2	2	3	3	3	3	3	3	3	3	4	4	4	4
Euler	0	1	2	3	2	1	4	5	4	6	4	1	7	1	0	8	9	8	0	
len 1	0	1	2	3	2	1	4	5	4	6	4	1	7	1	0	8	9	8	0	
len 2	0	1	2	2	1	1	4	4	4	4	1	1	1	0	0	8	8	0		
len 4	0	1	1	1	1	1	4	4	1	1	1	0	0	0	0	0				
⇨ len 8	0	1	1	1	1	1	1	0	0	0	0	0								
len 16	0	0	0	0																

```
int u, v;
cin >> u >> v;
int fe1 = first_encounter[u];
int fe2 = first_encounter[v];
if (fe1 > fe2)
    swap(fe1, fe2);
int LCA_new_index = rmq.minimum(fe1, fe2);
int LCA_old_index = new_to_old[LCA_new_index];
//cout << "LCA_new_index: " << LCA_new_index << endl;
cout << "LCA(" << u << "," << v << "): " << LCA_old_index << endl;
```

new_to_old:{0 5 7 2 1 3 6 8 9 4}

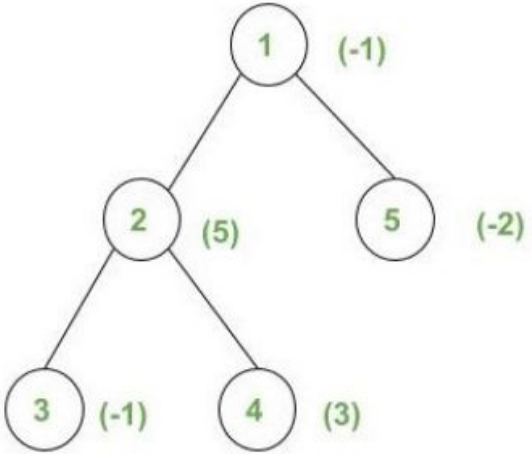



```
LCA(2,8): 5
```

	Construcción	Query
Naive	-	O(h)
SQRT	O(V+E)	O(sqrt(h))
RMQ + Segment Tree	O(n)	O(logn)
RMQ + Sparse Table	O(nlogn)	O(1)

Aplicación

Consulta para encontrar el peso máximo y mínimo entre dos nodos en el árbol dado usando LCA



Query=[{1, 3}, {2, 4}, {3, 5}]

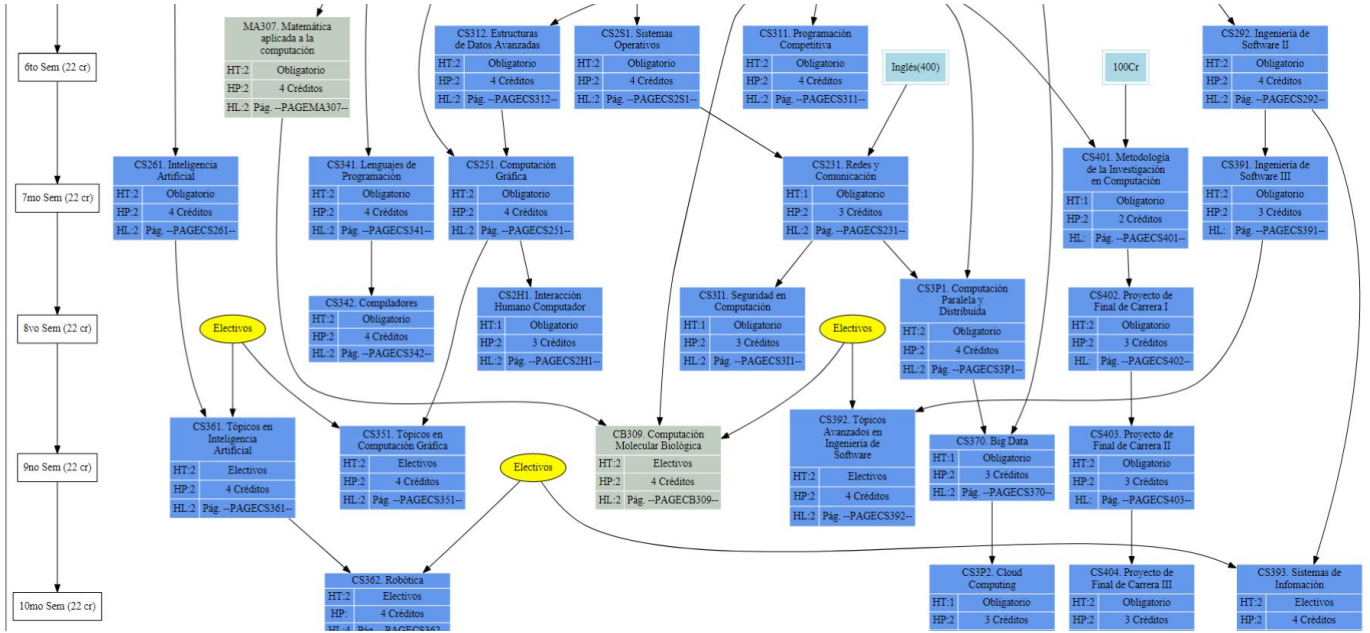
Output:

-1 5

3 5

-2 5

Aplicación

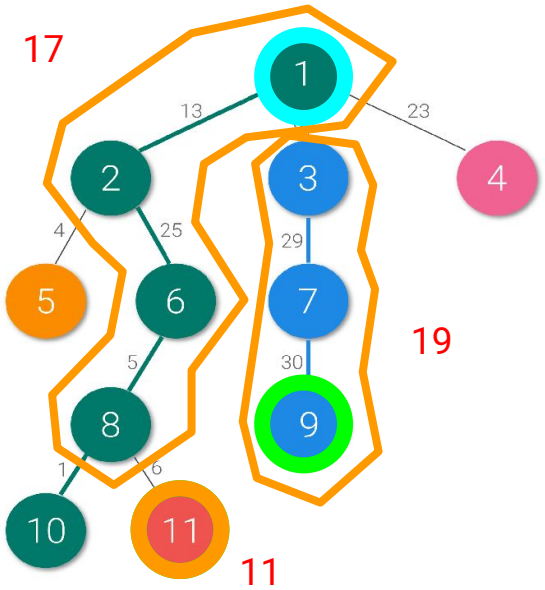

6to Sem (22 cr)
7mo Sem (22 cr)
8vo Sem (22 cr)
9no Sem (22 cr)
10mo Sem (22 cr)
MA307. Matemática aplicada a la computación
HT:2 Obligatorio
HP:2 4 Créditos
HL:2 Pág. --PAGEMA307--
CS312. Estructuras de Datos Avanzadas
HT:2 Obligatorio
HP:2 4 Créditos
HL:2 Pág. --PAGECS312--
CS2S1. Sistemas Operativos
HT:2 Obligatorio
HP:2 4 Créditos
HL:2 Pág. --PAGECS2S1--
CS311. Programación Competitiva
HT:2 Obligatorio
HP:2 4 Créditos
HL:2 Pág. --PAGECS311--
Inglés(400)
100Cr
CS292. Ingeniería de Software II
HT:2 Obligatorio
HP:2 4 Créditos
HL:2 Pág. --PAGECS292--
CS261. Inteligencia Artificial
HT:2 Obligatorio
HP:2 4 Créditos
HL:2 Pág. --PAGECS261--
CS341. Lenguajes de Programación
HT:2 Obligatorio
HP:2 4 Créditos
HL:2 Pág. --PAGECS341--
CS251. Computación Gráfica
HT:2 Obligatorio
HP:2 4 Créditos
HL:2 Pág. --PAGECS251--
CS231. Redes y Comunicación
HT:1 Obligatorio
HP:2 3 Créditos
HL:2 Pág. --PAGECS231--
CS401. Metodología de la Investigación en Computación
HT:1 Obligatorio
HP:2 2 Créditos
HL: Pág. --PAGECS401--
CS391. Ingeniería de Software III
HT:2 Obligatorio
HP:2 3 Créditos
HL: Pág. --PAGECS391--
CS342. Compiladores
HT:2 Obligatorio
HP:2 4 Créditos
HL:2 Pág. --PAGECS342--
CS2H1. Interacción Humano Computador
HT:1 Obligatorio
HP:2 3 Créditos
HL:2 Pág. --PAGECS2H1--
CS3I1. Seguridad en Computación
HT:1 Obligatorio
HP:2 3 Créditos
HL:2 Pág. --PAGECS3I1--
CS3P1. Computación Paralela y Distribuida
HT:2 Obligatorio
HP:2 4 Créditos
HL:2 Pág. --PAGECS3P1--
CS402. Proyecto de Final de Carrera I
HT:2 Obligatorio
HP:2 3 Créditos
HL: Pág. --PAGECS402--
Electivos
Electivos
Electivos
CS361. Tópicos en Inteligencia Artificial
HT:2 Electivos
HP:2 4 Créditos
HL:2 Pág. --PAGECS361--
CS351. Tópicos en Computación Gráfica
HT:2 Electivos
HP:2 4 Créditos
HL:2 Pág. --PAGECS351--
CB309. Computación Molecular Biológica
HT:2 Electivos
HP:2 4 Créditos
HL:2 Pág. --PAGECB309--
CS392. Tópicos Avanzados en Ingeniería de Software
HT:2 Electivos
HP:2 4 Créditos
HL:2 Pág. --PAGECS392--
CS370. Big Data
HT:1 Obligatorio
HP:2 3 Créditos
HL:2 Pág. --PAGECS370--
CS403. Proyecto de Final de Carrera II
HT:2 Obligatorio
HP:2 3 Créditos
HL: Pág. --PAGECS403--
CS362. Robótica
HT:2 Electivos
HP: 4 Créditos
CS3P2. Cloud Computing
HT:1 Obligatorio
HP:2 3 Créditos
CS404. Proyecto de Final de Carrera III
HT:2 Obligatorio
HP:2 3 Créditos
CS393. Sistemas de Infomación
HT:2 Electivos
HP:2 4 Créditos

Aplicación

Suma de los vértices que conforman el camino de 11 a 9

LCA(11,9)=1

Inicio y fin de la trayectoria



Referencias

https://en.wikipedia.org/wiki/Lowest_common_ancestor

https://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-search-tree/

https://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-tree-set-2-using-parent-pointer/

https://www.geeksforgeeks.org/sqrt-square-root-decomposition-set-2-lca-tree-osqrth-time/

https://www.slideshare.net/ekmett/skewbinary-online-lowest-common-ancestor-search

https://www.geeksforgeeks.org/find-lca-in-binary-tree-using-rmq/?ref=rp

https://www.geeksforgeeks.org/lca-n-ary-tree-constant-query-o1/

https://www.geeksforgeeks.org/lca-n-ary-tree-constant-query-o1/

https://www.geeksforgeeks.org/sqrt-square-root-decomposition-set-2-lca-tree-osqrth-time/

https://github.com/jakobkogler/Algorithm-DataStructures/blob/master/Graphs/lca.cpp

https://cp-algorithms.com/graph/hld.html

https://www.youtube.com/watch?v=ZBHKZF5w4YU

https://www.youtube.com/watch?v=9FLPwDn6L08&t=943s

https://www.geeksforgeeks.org/query-to-find-the-maximum-and-minimum-weight-between-two-nodes-in-the-given-tree-using-lca/?ref=rp

¿Alguna pregunta?