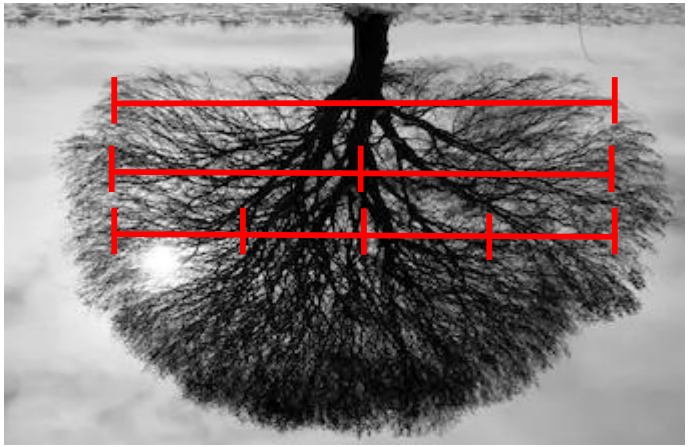
Data Structures III: Segment Tree

Basic Segment Tree

Segment Tree



- Estructura de datos potente que nos permite contestar queries en rango **en O(log n)** bajo algún operador binario o función que cumpla con la propiedad **asociativa.**
- En esta estructura se pueden hacer fácilmente **point updates** en **O(log n)**
- También serán posibles en ciertos casos hacer **range updates** en **O(log n)** con una técnica especial.
- Es una estructura bastante flexible que permite incluso hacer muchos más trucos avanzados
- Está basado en el paradigma de **Divide and Conquer**

Segment Tree – Key Idea

Empecemos con la versión más sencilla del segment tree en donde tenemos **range queries** de algún operador (o función) asociativo ★ y **point updates**.

$$\rightarrow query\,(l,r) = A[l] \bigstar A[l+1] \bigstar A[l+2] \bigstar \cdots \bigstar A[r-1] \bigstar A[r]$$

$$\rightarrow update\,(pos, val) : A[pos] = val$$

Así como en Sparse Table, se hará un **preprocesamiento** de la respuesta de algunos intervalos.

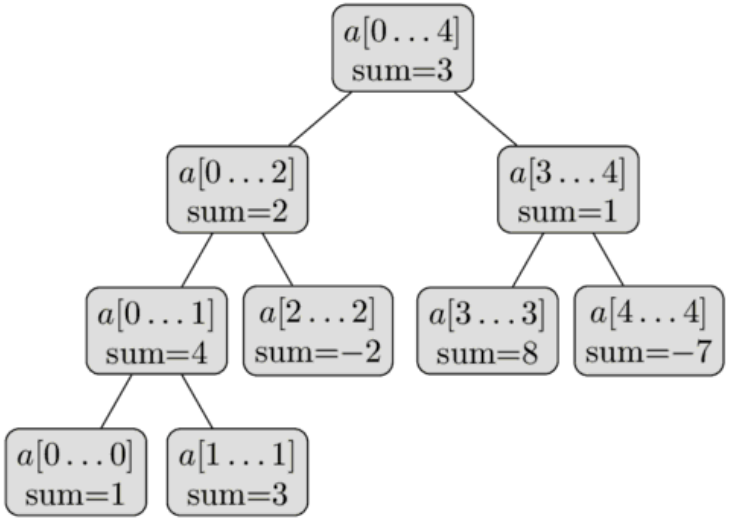
Sin embargo, esta vez lo haremos al estilo Divide and Conquer. Si queremos hallar el resultado del segmento $[0, n-1]$ lo calcularemos a partir de las respuestas de los segmentos $\left[0, \frac{n-1}{2}\right]$ y $[\frac{n-1}{2}+1,\ n-1]$.

De esta forma se irá creando un “árbol de segmentos” con las respuestas precalculadas, la cual usaremos más adelante para responder las queries eficientemente

Segment Tree – Key Idea

Como ejemplo, supongamos que el operación es la suma (+) y el arreglo es el siguiente :

$A = [1, 3, -2, 8, -7]$. Entonces el segment tree sería el siguiente :



Fuente : cp-algorithms

Segment Tree – Construcción

En el segment tree, cada nodo tendrá dos hijos, a excepción de las hojas, y el valor de cada nodo del segment tree se calculará a partir del valor de sus dos hijos.
La primera interrogante que puede surgir es , ¿cuántos nodos podríamos necesitar en el peor caso?

- Un primer límite podría calcularse de la siguiente forma : En el primer nivel hay un solo nodo, en el 2do nivel habrán 2 nodos, en el 3er nivel habrán 4to nodos y así sucesivamente. En cada nivel, la longitud del segmento se divide entre 2, por lo tanto la cantidad de niveles es $\lceil log_2(n) \rceil + 1$.

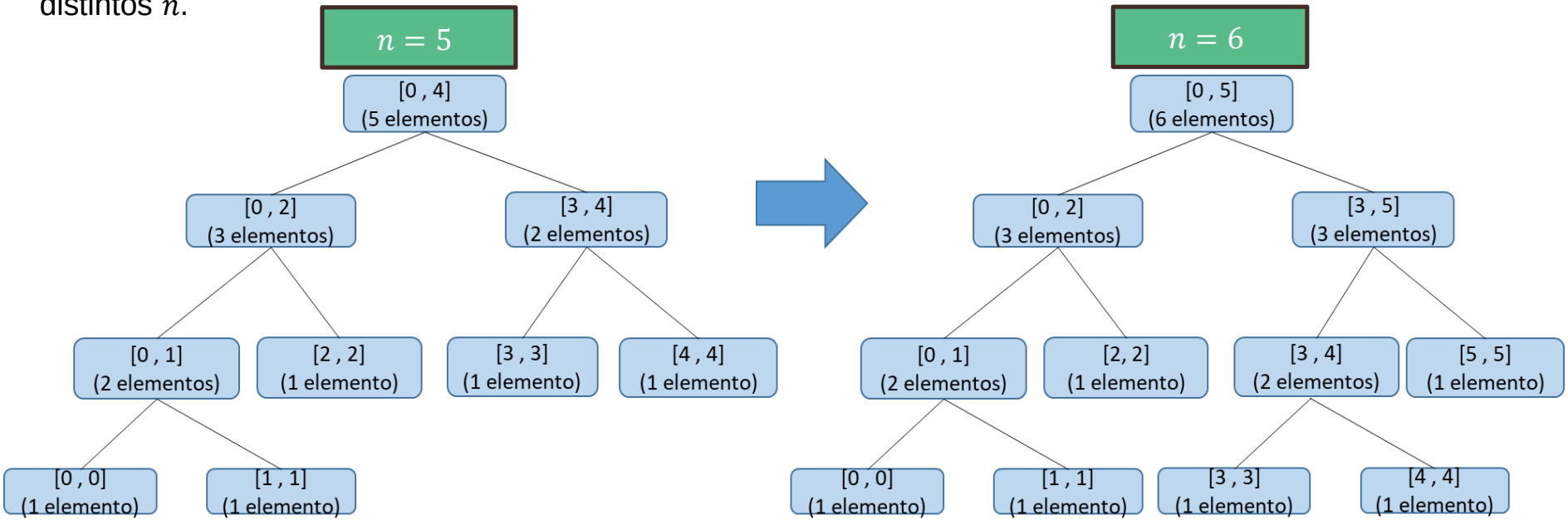
$$\#\ de\ nodos \leq 1 + 2 + 4 + \cdots + 2^{\lceil log_2(n) \rceil} = 2^{\lceil log_2(n) \rceil + 1} < 4n$$

Es decir, tenemos O(n) de memoria.
Sin embargo, esto es solo una cota superior. Pero podemos encontrar un límite más exacto ya que el último nivel del segment tree no siempre está completo (como en el ejemplo de la diapo anterior)

- En el ejemplo anterior vemos que para n = 5, tenemos un total de 9 nodos. Pareciera indicar que la cantidad de nodos fuera $2n - 1$

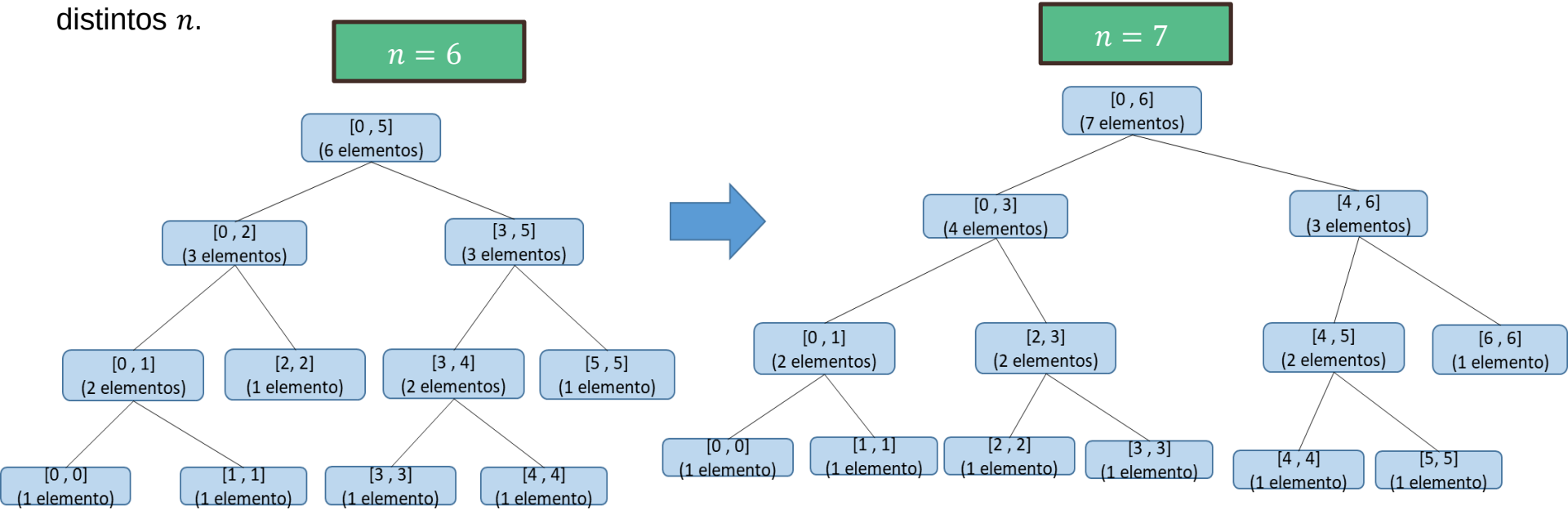
Segment Tree – Construcción

Si el tamaño del arreglo es n, se puede demostrar que la cantidad de nodos es $2n - 1$. Para darnos cuenta de ello primero veamos algunos ejemplos adicionales de la estructura de un segment tree para distintos n.



Segment Tree – Construcción

Si el tamaño del arreglo es n, se puede demostrar que la cantidad de nodos es $2n - 1$. Para darnos cuenta de ello primero veamos algunos ejemplos adicionales de la estructura de un segment tree para distintos n.



Segment Tree – Construcción

En general, cada vez que aumentamos en 1 el tamaño del arreglo. El primer nodo (cuyo segmento engloba a todo el arreglo) aumentará en 1 su tamaño. Luego de ello solo uno de sus hijos (el izquierdo o el derecho) aumentará el tamaño en 1. De igual forma, para ese hijo que aumentó su tamaño, solo 1 de sus hijos aumentará el tamaño en 1. Y así sucesivamente hasta que se llega a un nodo que originalmente era una hoja, pero que al aumentar su tamaño en 1, se vuelva un segmento de tamaño 2 y se vea en la necesidad de crear 2 nodos hijos de tamaño 1.

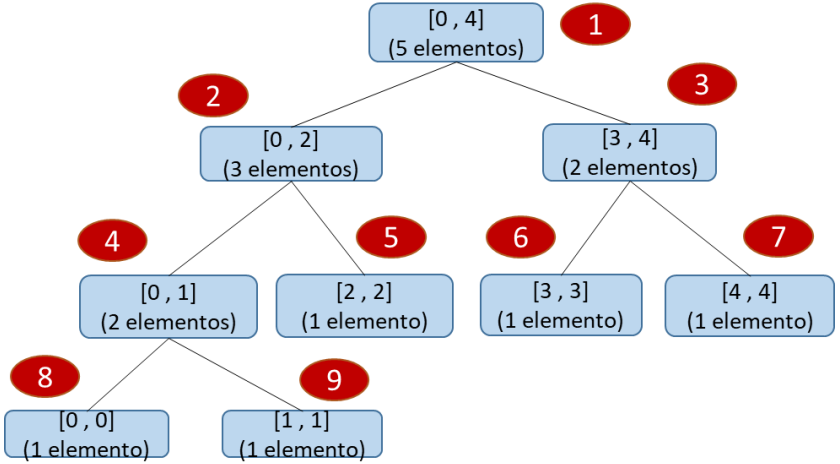
Por lo tanto, cada vez que n aumenta, el # de nodos aumenta en 2. Además sabemos que la cantidad de nodos cuando $n = 1$, es igual a 1.

$\Rightarrow \#\ de\ nodos = 2n\ - 1$

Segment Tree – Construcción

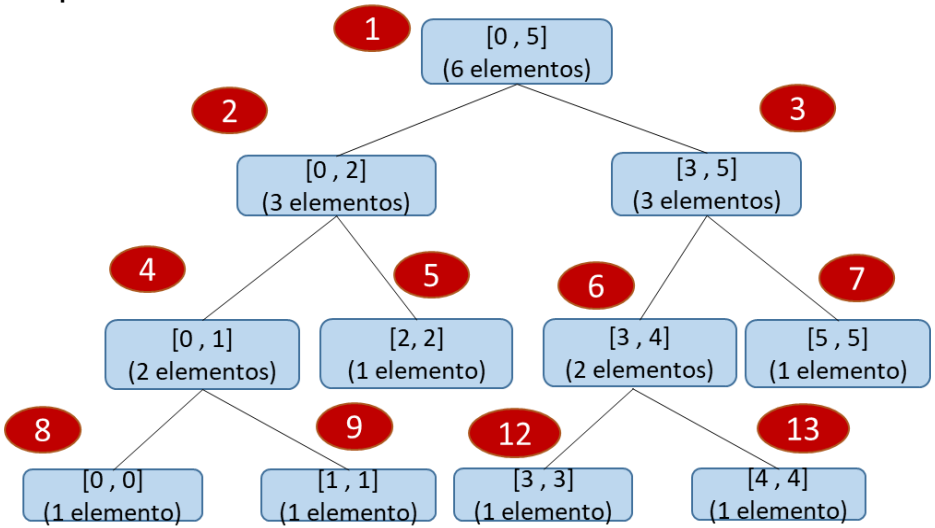
La siguiente interrogante es ¿cómo implementarlo? Podemos hacerlo recursivamente al estilo Divide and Conquer. Pero antes, recordemos que para cada nodo del segment tree, debemos guardar su respuesta. Sabemos que solo son $2n - 1$ nodos, pero también necesitaremos alguna forma de indexar estos nodos para guardarlos en un arreglo.

Puede haber varias formas de hacerlo, pero hay un truco muy usado que nos facilitará mucho las cosas. Al nodo raíz le pondremos índice 1. A partir de ahí, supongamos que un nodo tiene índice i , entonces asignaremos a su hijo izquierdo el índice $2i$ y a su hijo derecho el índice $2i + 1$



Segment Tree – Construcción

Hay un ligero detalle al usar esta clase de indexación. Si bien es cierto que la cantidad de nodos es igual a $2n - 1$, no podemos decir lo mismo de la memoria que utilizaremos para los índices. Vamos a necesitar el doble de memoria $(4n)$, para estar seguros. Tomen por ejemplo el segment tree del n = 6. En el que el mayor índice es $13 > 2 \times 6 - 1$. Esto se debe a que los nodos no necesariamente se crearán secuencialmente de izquierda a derecha.



A pesar de que, en la mayoría de casos, este doble de memoria no es tan crítico, también existen otras indexaciones que solo usan $2n$ de memoria.

Segment Tree – Construcción

La construcción del segment tree para la operación suma quedaría así.

```
Long t[4 * MX];
void build(vector<Long> &a, Long id , Long tl , Long tr) { //O(n)
    if (tl == tr){
        t[id] = a[tl];
    } else{
        Long tm = (tl + tr) / 2;
        build(a, 2 * id, tl, tm);
        build(a, 2 * id + 1, tm + 1, tr);
        t[id] = t[2 * id] + t[2 * id + 1];
    }
}
```

Para llamar a la función se la llamaría de esta forma $build(a, 1, 0, n-1)$

Segment Tree – Construcción

En general podríamos hacer algo así :

```
Long combine(Long x , Long y) {
    return x + y;
}

Long t[4 * MX];
void build(vector<Long> &a, Long id , Long tl , Long tr) { //O(n)
    if (tl == tr){
        t[id] = a[tl];
    } else{
        Long tm = (tl + tr) / 2;
        build(a, 2 * id, tl, tm);
        build(a, 2 * id + 1, tm + 1, tr);
        t[id] = combine(t[2 * id] , t[2 * id + 1]);
    }
}
```

Segment Tree – Construcción

Otro ejemplo de indexación es ir enumerando los nodos consecutivamente con forme se recorren recursivamente (como si fuera el orden de un dfs). Supongamos que estamos en un nodo con índice id Entonces el nodo de la izquierda será $id + 1$. Pero el de la izquierda tendría que ser $id + cantidad\ de\ nodos\ en\ el\ subarbol\ izquierdo + 1$. Podemos usar la fórmula que obtuvimos en diapositivas anteriores : en cualquier árbol cuya longitud de segmento es n , la cantidad de nodos es $2\ (n-1)$. Por lo que el índice del nodo derecho será $id + |rango\ del\ nodo\ de\ la\ izquierda| + 1$
Con esto, reduciríamos la memoria a la cantidad exacta.

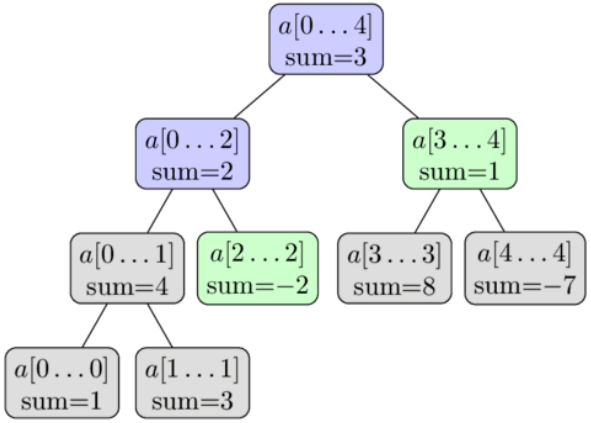
```
Long t[2 * MX];

void build(vector<Long> &a, Long id , Long tl , Long tr) { //O(n)
    if (tl == tr){
        t[id] = a[tl];
    } else{
        Long tm = (tl + tr) / 2;
        Long left = id + 1;
        Long right = id + 2 * (tm - tl + 1) ;
        build(a, left, tl, tm);
        build(a, right, tm + 1, tr);
        t[id] = combine(t[left] , t[right]);
    }
}
```

Segment Tree – Query

Ahora supongamos que queremos responder queries en rango

$$\rightarrow query\,(l,r) = A[l] \bigstar A[l+1] \bigstar A[l+2] \bigstar \cdots \bigstar A[r-1] \bigstar A[r]$$

La idea es dividir este segmento $[l\,,r]$ en segmentos consecutivos que estén presentes en el Segment Tree. Podemos ir haciendo esto recursivamente. Veamos con el ejemplo que pusimos al principio. Definimos el arreglo $A = [1\,,3\,,-2\,,\ 8\,,\ -7]$. Las queries son encontrar la suma de un rango $[l,r]$. Supongamos que tenemos $query(2\,,4) = -2 + 8 - 7 = -1$

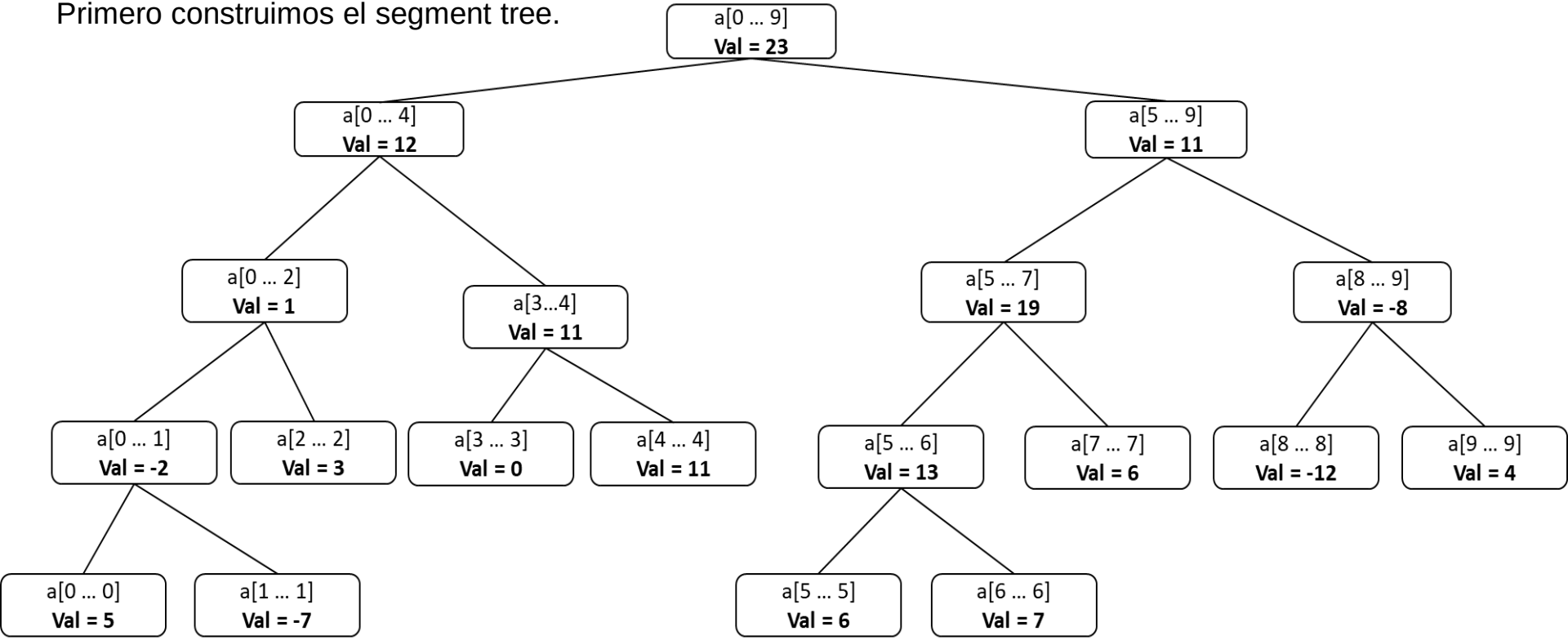


Fuente : cp-algorithms

Segment Tree – Query

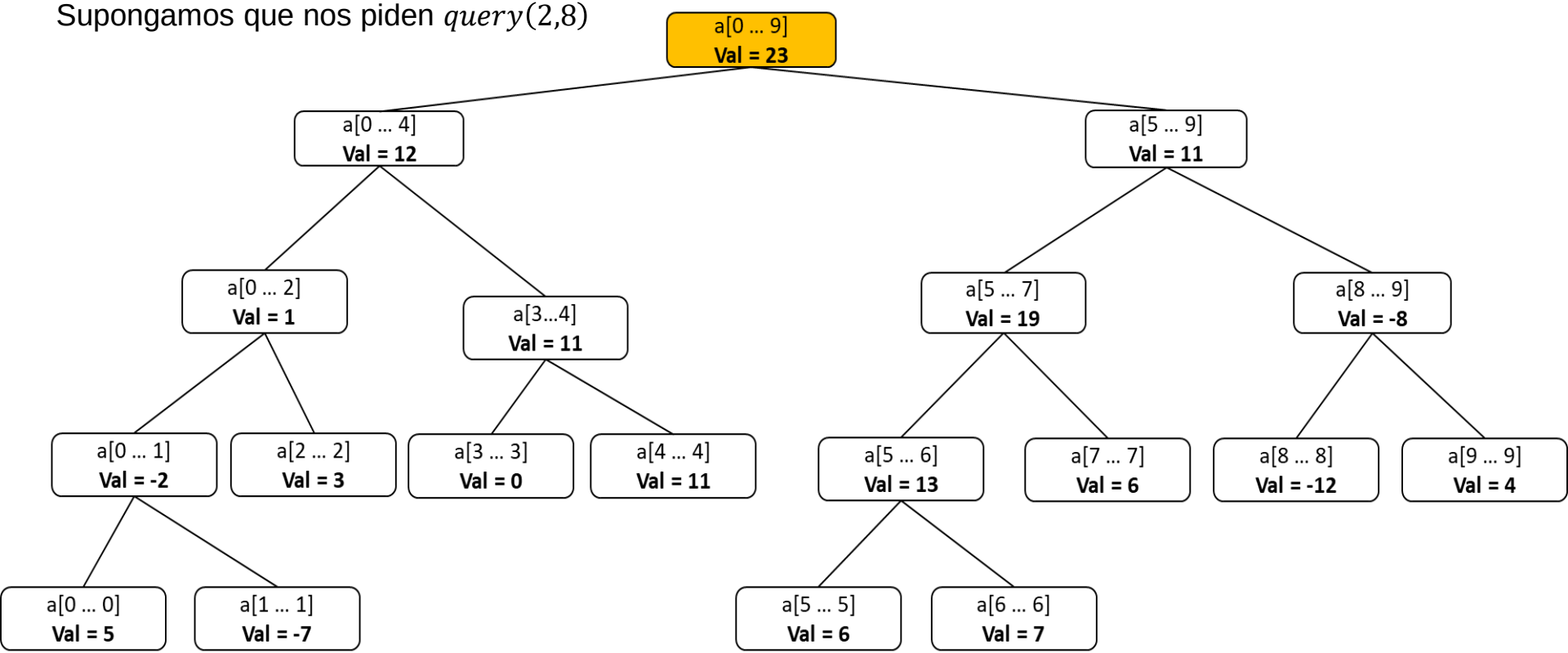
Veamos un ejemplo más complicado, para $n = 10$. $A = [5, -7, 3, 0, 11, 6, 7, 6, -12, 4]$

Primero construimos el segment tree.



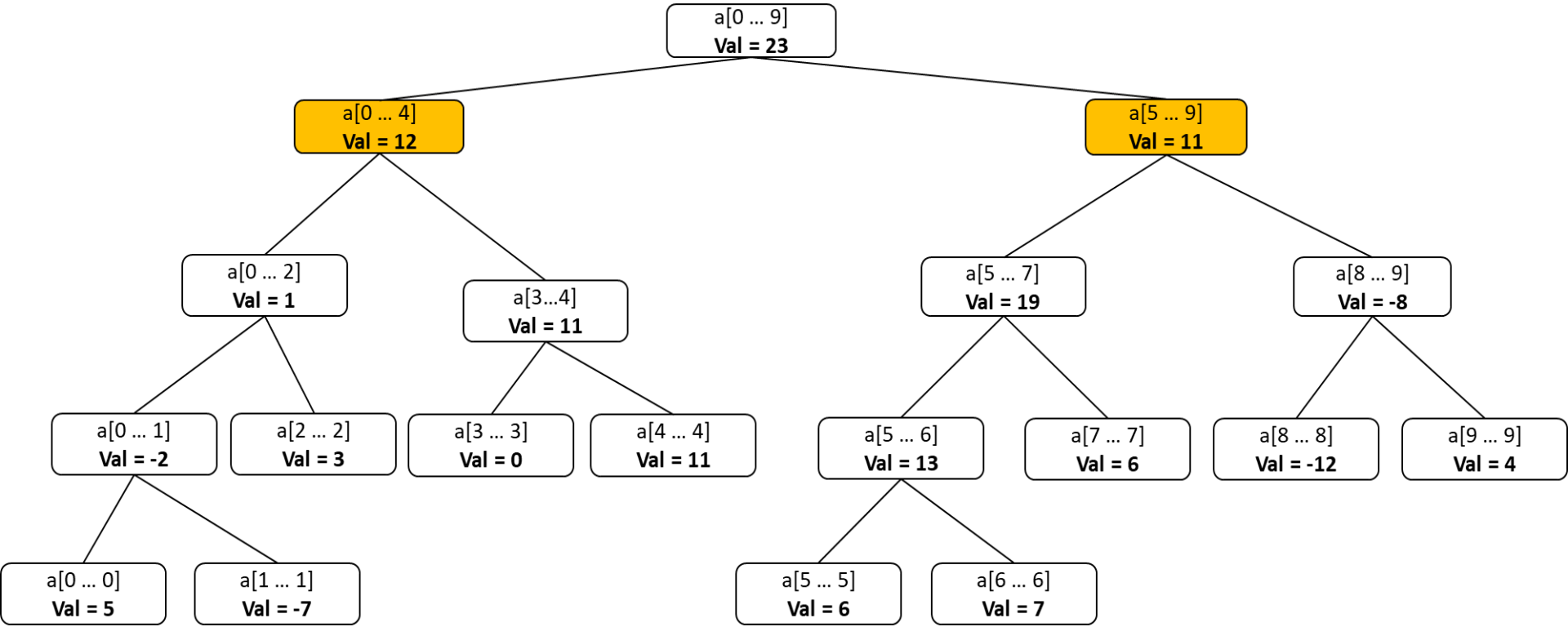
Segment Tree – Query

Supongamos que nos piden $query(2,8)$



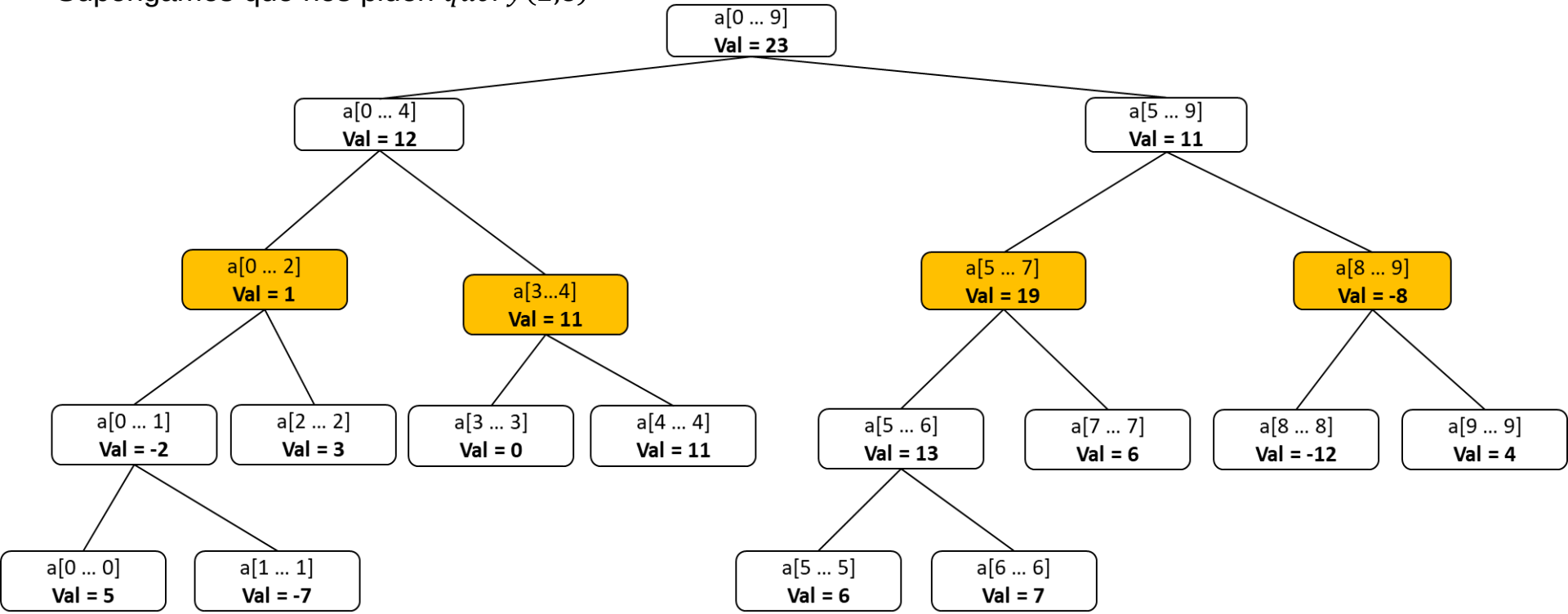
Segment Tree – Query

Supongamos que nos piden $query(2,8)$



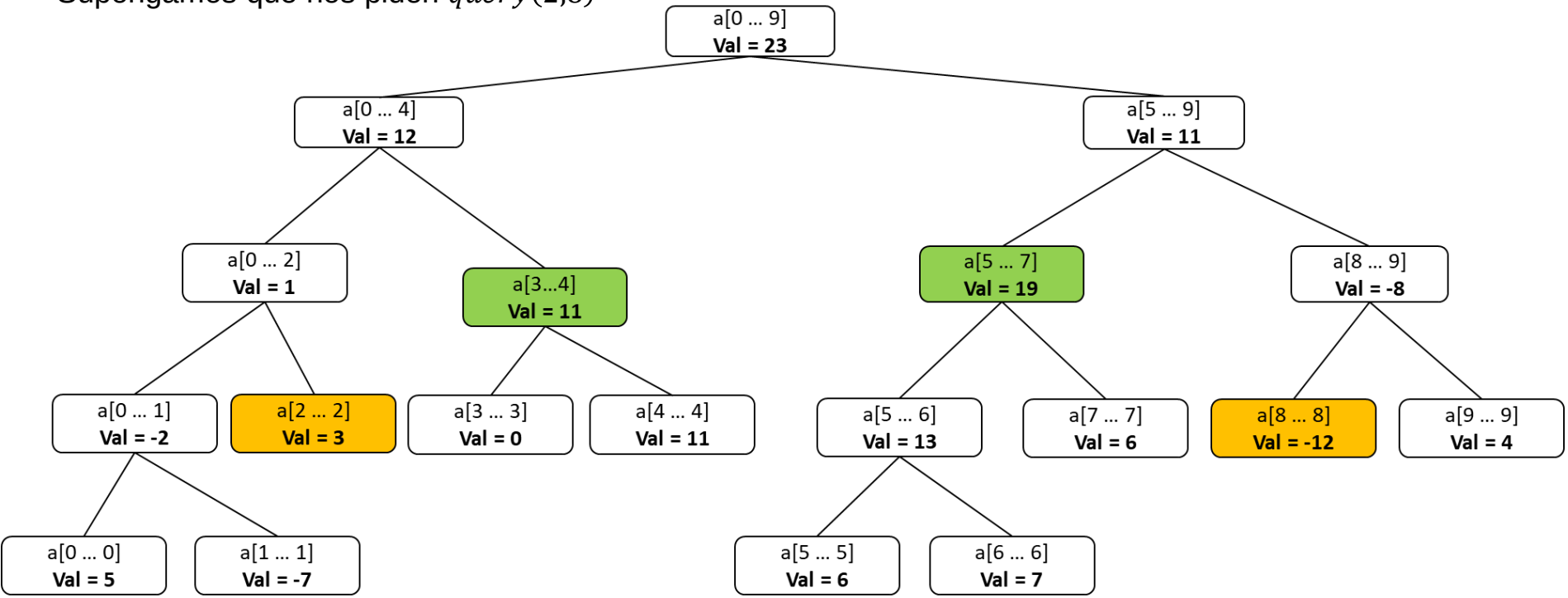
Segment Tree – Query

Supongamos que nos piden $query(2,8)$



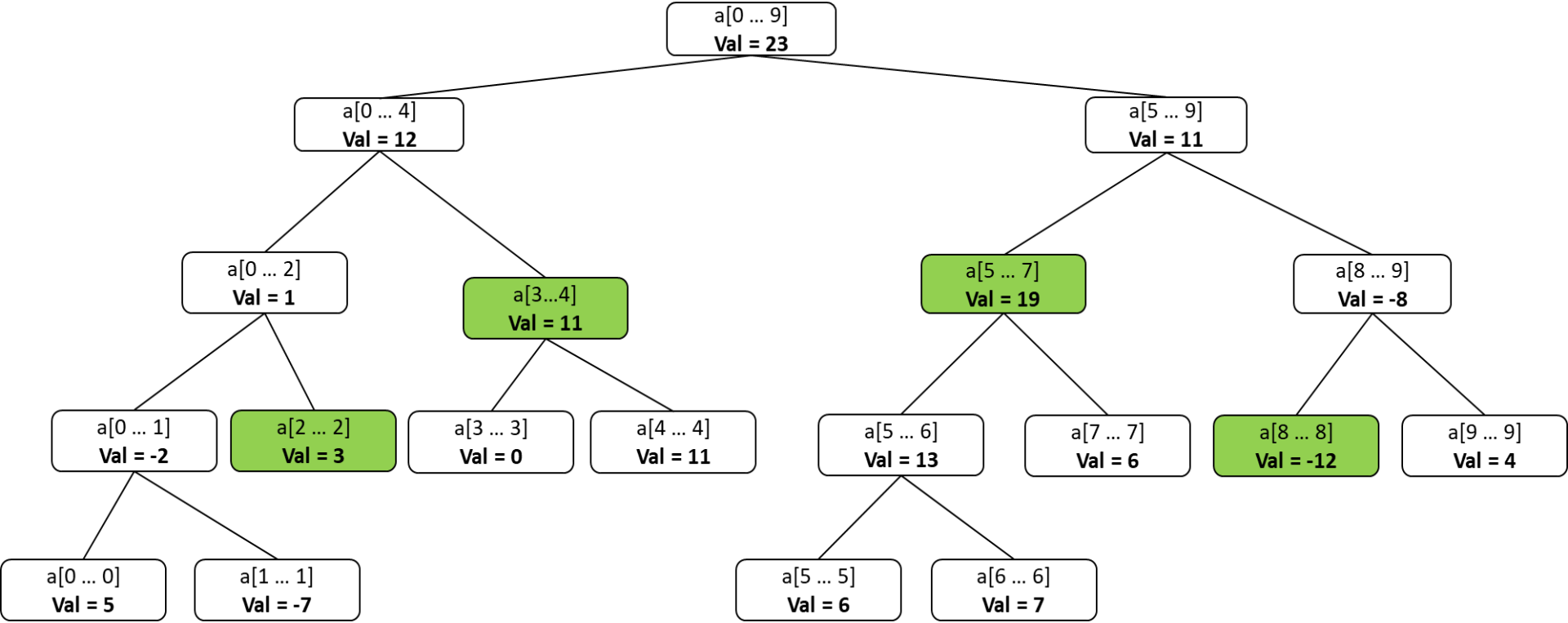
Segment Tree – Query

Supongamos que nos piden $query(2,8)$



Segment Tree – Query

Supongamos que nos piden $query(2,8) = 21$



Segment Tree – Query

Podemos resumir el algoritmo de la query en 3 casos :

1) El intervalo del nodo en el que estamos está totalmente dentro del segmento de la query. En ese caso simplemente retornamos el valor de ese segmento como parte de la respuesta.

2) Caso contrario, solo parte del segmento del nodo está dentro del nodo de la query. Aquí hay 2 casos.

2.1) Solo uno de sus hijos es parte del segmento de la query. Por lo que solo es necesario ir a ese hijo (1 llamada recursiva).

2.2) Ambos hijos son parte del segmento de la query así que debo ir recursivamente a ambos nodos (2 llamadas recursivas).

Segment Tree – Query

Para calcular la complejidad de la query, recordemos que el segment tree tiene $[log_2 n] + 1$ niveles.
A primera vista pareciera que la forma recursiva de la query podría visitar muchos nodos en cada nivel. En el ejemplo que pusimos de $n = 10$ pudimos ver como todo el 3er nivel se visitó.

Afortunadamente la máxima cantidad de nodos que puede visitar la query en cada nivel es 4.
Demostración:
Demostrémoslo por inducción en los niveles. En el primer nivel solo se visita 1 nodo, así que se cumple.
Supongamos que es cierto que en el $\mathrm{i} - ésimo$ nivel se cumple que la cantidad de nodos es ≤ 4
Por lo visto anteriormente , sabemos que hay 3 casos en el algoritmo. En el peor de ellos se hacen 2 llamadas recursivas (cada nodo llama a sus 2 hijos).

- Si el $\#\ de\ nodos$ del nivel i es ≤ 2 , entonces como cada nodo a lo más hace 2 llamadas recursivas, en el siguiente nivel se visitarán a lo más 4 nodos

- Si el $\#\ de\ nodos$ del nivel i es > 2 , note que estos 3 o 4 nodos tienen que corresponder a segmentos consecutivos cuya intersección con el segmento de la query no sea nula. Esto quiere decir que los nodos intermedios tienen que estar **completamente contenidos** en el segmento de la query. Por lo tanto solo los 2 nodos extremos harán llamados a su hijos y la cantidad de nodos visitados en el siguiente nivel será a lo más 4.

Segment Tree – Query

Tenemos entonces que hay $\lceil log_2 n \rceil + 1$ niveles en el segment tree y que la query visita a lo más 4 nodos en cada nivel. Así que en el peor caso, se visitan $4(\lceil log_2 n \rceil + 1)$ nodos.

Entonces la complejidad de la query es $O(4\ log_2 n) = O(\log n)$. La implementación de la query sería de esta forma :

```
Long query(Long l, Long r, Long id , Long tl , Long tr ) { //O(logn)
    if (l <= tl && tr <= r) {
        return t[id];
    }
    Long tm = (tl + tr) / 2;
    Long left = id + 1;
    Long right = id + 2 * (tm - tl + 1) ;
    if(r < tm + 1){
        //only left child
        return query(l , r , left , tl , tm);
    }else if(tm < l){
        //only right child
        return query(l , r, right , tm + 1 , tr);
    } else{
        //both children
        return combine(query(l, r, left, tl, tm) , query(l, r, right, tm + 1, tr));
    }
}
```

Se le llama $query(l\ , r\ , 1\ , 0\ , n\ - 1)$

Segment Tree – Query

Un truco que algunos usan es ahorrarse el caso en donde solo llamas a un solo hijo y llamar siempre a los dos hijos. Pero agregarle la condición de que si el segmento de la query tiene intersección nula con el segmento actual del nodo, entonces se retorna un elemento neutro

```
Long neutral = 0;
Long query(Long l, Long r, Long id , Long tl , Long tr ) { //O(logn)
    if(tr < l || tl > r){
        return neutral;
    }
    if (l <= tl && tr <= r) {
        return t[id];
    }
    Long tm = (tl + tr) / 2;
    Long left = id + 1;
    Long right = id + 2 * (tm - tl + 1) ;
    return combine(query(l, r, left, tl, tm) , query(l, r, right, tm + 1, tr));
}
```

Segment Tree – Update

La forma más simple de update es el **point update**. Es decir, modificar un solo elemento.
Supongamos un update que modifique al elemento de la posición pos. Entonces simplemente nos dirigimos a la hoja del segment tree que contenga al intervalo $a[pos \ldots pos]$. Modificamos ese nodo, y luego a todos sus ancestros.

Como la altura del segment tree (cantidad de niveles) es igual a $[log_2 n]$, entonces solo visitaremos $O(\log n)$ nodos, dándonos esa complejidad.

```
void update(Long pos, Long val, Long id, Long tl , Long tr) { //O(logn)
    if (tl == tr) {
        t[id] = val;
    }else{
        Long tm = (tl + tr) / 2;
        Long left = id + 1;
        Long right = id + 2 * (tm - tl + 1) ;
        if (pos <= tm) {
            update(pos, val, left, tl, tm);
        }else {
            update(pos, val, right, tm + 1, tr);
        }
        t[id] = combine(t[left] , t[right] );
    }
}
```

Segment Tree – Update

En el código anterior, la operación $update(pos, val)$, hacía $A[pos] = val$. Sin embargo, el update en el segment tree podría ser distinto e incluso más complejo sin afectar la exactitud del algoritmo

Por ejemplo

- $update(pos, val)$: $A[pos] \mathrel{+}= val$
- $update(pos, val)$: $A[pos] = \max(A[pos], val)$
- $update(pos, val)$: $A[pos] = numero\ de\ bits\ encendidos\ de\ val$
- $update(pos)$: $A[pos] = A[pos] \times A[pos]$
- $update(pos)$: $A[pos] = \lfloor \sqrt{A[pos]} \rfloor$

Esto proporciona bastante flexibilidad a la estructura.
Lo único que hay que hacer es cambiar la siguiente línea

```
void update(Long pos, Long val, Long id, Long tl , Long tr) { //O(logn)
    if (tl == tr) {
        t[id] = val;
```

Nota: el update cambia, pero las queries siguen siendo bajo el operador ★ (la función $combine$)

Nodos más complejos

Nodos más complejos – Vectores en nodos

Los nodos no solo pueden guardar datos como la suma en un rango o un valor lazy, sino que también pueden guardar datos más complejos como un vector , un set , multiset o incluso otra estructura más compleja como un BIT u otro Segment Tree (produciendo un Segment Tree 2D).

Veamos el caso más sencillo : **Nodos con vectores**. En este caso no guardaremos la respuesta a una query como lo hacíamos anteriormente, sino que guardaremos todo el subarreglo correspondiente a un rango. Por lo que la raíz contendrá todos los elementos del arreglo, los hijos de la raíz contendrán las dos mitades y así sucesivamente.

Hay 2 preguntas que pueden surgir al hacer esto :

1. ¿Cuánta complejidad de memoria se necesitará?
2. ¿Para qué nos serviría?

Nodos más complejos – Vectores en nodos

1. Para contestar a la primera pregunta, podemos fiarnos del hecho de que el segment tree tiene $O(\log n)$ niveles . En cada nivel aparecerán todos los elementos del arreglo en alguno de los nodos, por lo tanto en total solo usaremos $O(n \log n)$
2. Pensemos en el siguiente problema :

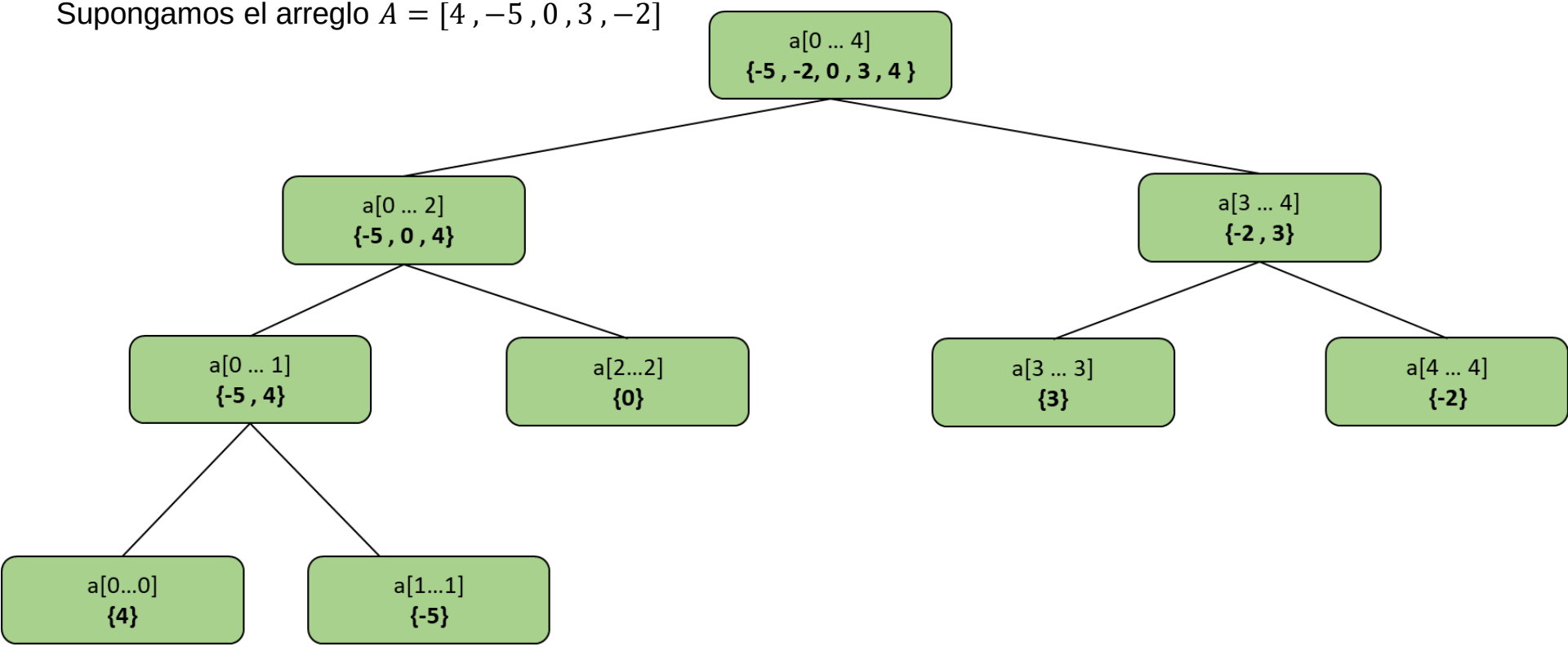
- $query(l, r, x)$: $Encontrar\ el\ menor\ elemento\ que\ sea \geq x\ en\ el\ intervalo\ [l, r]$
- $No\ hay\ updates$

Entonces podemos guardar en cada nodo del segment tree el vector **ordenado** correspondiente al rango para poder aplicar la función STL $lower\_bound$.

Sin embargo, recordemos que una query se divide en varios nodos. En cada nodo hallaremos el $lower\_bound$ del vector de ese nodo y terminaremos el mínimo de todos los $lower\_bound$ hallados.

Nodos más complejos – Vectores en nodos

Supongamos el arreglo $A = [4, -5, 0, 3, -2]$



Nodos más complejos – Vectores en nodos

La construcción será parecida al código de divide and conquer que hicimos para el **merge sort.** Pero para hacer el código más corto, podemos usar la función **merge** de C++ para la parte de juntar dos vectores ordenados.

```
vector<Long> t[2 * MX]; //O(nlogn)
void build(vector<Long> &a, Long id , Long tl, Long tr ) { //O(nlogn)
    if (tl == tr) {
        t[id] = {a[tl]};
    }else {
        Long tm = (tl + tr) / 2;
        Long left = id + 1;
        Long right = id + 2 * (tm - tl + 1) ;
        build(a, left , tl, tm);
        build(a, right, tm + 1, tr);
        merge(t[left].begin(), t[left].end(), t[right].begin(), t[right].end(), back_inserter(t[id]));
    }
}
```

Nodos más complejos – Vectores en nodos

En la query la complejidad será $O(log^2 n)$ debido a la complejidad del $lower\_bound$

```
Long combine(Long x, Long y){
    return min(x, y);
}

Long query(Long l , Long r, Long x, Long id, Long tl , Long tr) { //O(log²n)
    //find the smallest number greater or equal to x
    if(tr < l || tl > r){
        return INF;
    }
    if (l <= tl && tr <= r) {
        auto it = lower_bound(t[id].begin(), t[id].end(), x);
        if (it != t[id].end()) {
            return *it;
        } else{
            return INF;
        }
    }
    Long tm = (tl + tr) / 2;
    Long left = id + 1;
    Long right = id + 2 * (tm - tl + 1) ;
    return combine(query(l, r, x , left, tl, tm) , query(l, r, x , right, tm + 1, tr));
}
```

Nodos más complejos – Multiset en nodos

Ahora intentemos el mismo problema pero con $updates$

- $query(l, r, x)$: $Encontrar\ el\ menor\ elemento\ que\ sea \geq x\ en\ el\ intervalo\ [l, r]$
- $update(pos, val)$: $A[pos] = val$

Imaginemos que queremos mantener nuestra idea anterior de usar un vector. Cuando hagamos un update, vamos a tener que cambiar la hoja, pero todos los vectores de sus ancestros también cambiarán. En todos ellos habrá que quitar el valor antiguo de la hoja, insertar el nuevo y luego volver a ordernar al vector (sino, no se puede aplicar $lower\_bound$).

Esto sería ineficiente, y para ello mejor utilizamos un multiset.

Para poder obtener el antiguo valor de la hoja mantendremos un arreglo en donde guardaremos ese dato y lo iremos actualizando en cada update.

Nodos más complejos – Multiset en nodos

Para la construcción, hay que tener en cuenta que para cada nodo tenemos que juntar los 2 multiset de sus 2 hijos en el nodo actual. Una forma es iterando sobre los elementos de los multisets hijos e insertarlos en el padre. Pero podemos hacerlo en menos líneas de código aprovechando que la función $insert$ también puede insertar varios valores de otro contenedor. Debido a que juntar dos multisets tiene complejidad $O(sz \log sz)$ (donde sz es el tamaño de los 2 multisets) , la complejidad total del build tendrá un log adicional, y finalmente sería $O(nlog^2 \mathrm{n})$.

```
multiset<Long> t[2 * MX]; //O(nlogn)
Long curVal[MX]; //current value of the leaf

void build(vector<Long> &a , Long id , Long tl , Long tr) { //O(nlog²n)
    if (tl == tr){
        t[id] = {a[tl]};
        curVal[tl] = a[tl];
    }else{
        Long tm = (tl + tr) / 2;
        Long left = id + 1;
        Long right = id + 2 * (tm - tl + 1) ;
        build(a , left, tl, tm);
        build(a , right, tm + 1, tr);
        t[id] = t[left];
        t[id].insert(t[right].begin(), t[right].end());
    }
}
```

Nodos más complejos – Multiset en nodos

Las queries no cambiarán mucho

```
Long combine(Long x, Long y){
    return min(x , y);
}

Long query(Long l, Long r, Long x , Long id , Long tl , Long tr) { //O(log²n)
    //find the smallest number greater or equal to X
    if(tr < l || tl > r){
        return INF;
    }

    if (l <= tl && tr <= r) {
        auto it = t[id].lower_bound(x);
        if (it != t[id].end()) {
            return *it;
        } else{
            return INF;
        }
    }
    Long tm = (tl + tr) / 2;
    Long left = id + 1;
    Long right = id + 2 * (tm - tl + 1) ;
    return combine(query(l, r, x , left, tl, tm) , query(l, r, x , right, tm + 1, tr));
}
```

Nodos más complejos – Multiset en nodos

En los updates hay que eliminar el valor antiguo e insertar el nuevo en todos el recorrido hasta la hoja.

```
void update(Long pos, Long val , Long id , Long tl, Long tr ) { //O(log²n)
    t[id].erase(t[id].find(curVal[pos]));
    t[id].insert(val);
    if(tl == tr){
        curVal[pos] = val;
    }else{
        Long tm = (tl + tr) / 2;
        Long left = id + 1;
        Long right = id + 2 * (tm - tl + 1) ;
        if (pos <= tm) {
            update(pos, val, left, tl, tm);
        }else {
            update(pos, val , right, tm + 1, tr);
        }
    }
}
```

Lazy Propagation

Point Query ,Range Update

Hasta ahora hemos trabajado **range queries** y **point updates**. Ahora supongamos que tenemos el siguiente problema :

- $query(pos) \rightarrow Retornar\ A[pos]$
- $update(l, r, val) \rightarrow \forall i \in [l, r],\ A[i] \mathrel{+}= val$

Es decir, sumar un valor a todo un rango.

Por un momento puede parecer que el segment tree ya no es útil. Después de todo el objetivo de cada de nodo del segment tree era contener el valor de la query para un rango. Sin embargo, la idea es que ahora cada nodo contenga la información del update para el rango de ese nodo.

Point Query ,Range Update

Un punto importante es que en problemas que requieren de **range updates**, tenemos que ver la forma de como **acumular** varios updates. Esta “acumulación” no siempre es de la misma forma.

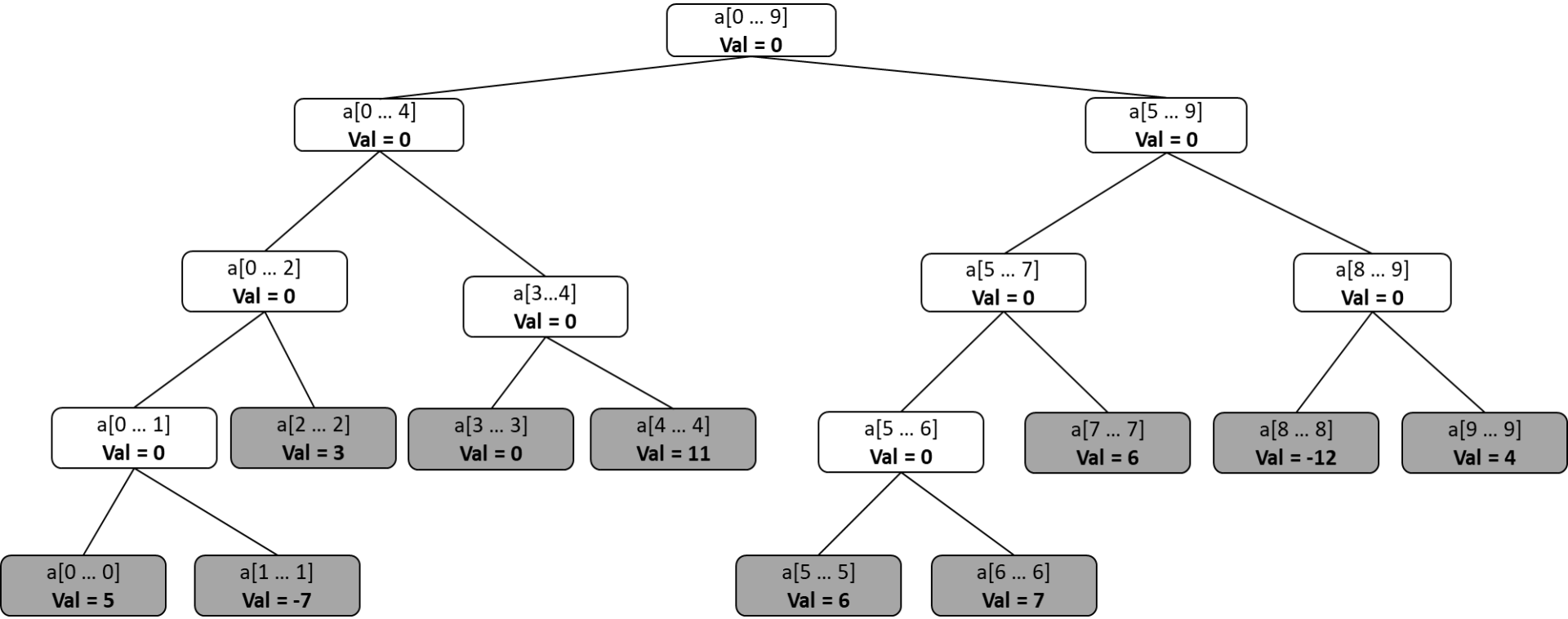
Por ejemplo, para nuestro problema actual, imaginemos que tenemos un update de sumar +5 en todo un rango y luego otro update de sumar +7 en ese mismo rango. Entonces podemos acumularlos y decir que en total acumularán +12. Pero tengan cuidado que no siempre se pueden acumular los updates, y en ese caso quizás sea mejor pensar en otra estructura de datos distinta al segment tree.

Entonces , así como antes dividíamos un **range query** en nodos con segmentos precalculados del segment tree, de igual forma dividiremos los **range update** en nodos del segment tree donde acumularemos los updates.

Pero para contestar las queries, necesitaríamos obtener el valor de la hoja habiendo sido actualizada luego de todos los updates. Sin embargo, **cuando el orden de los updates no importan** podemos usar el truco de aplicar los updates conforme recorremos el camino hacia la hoja.

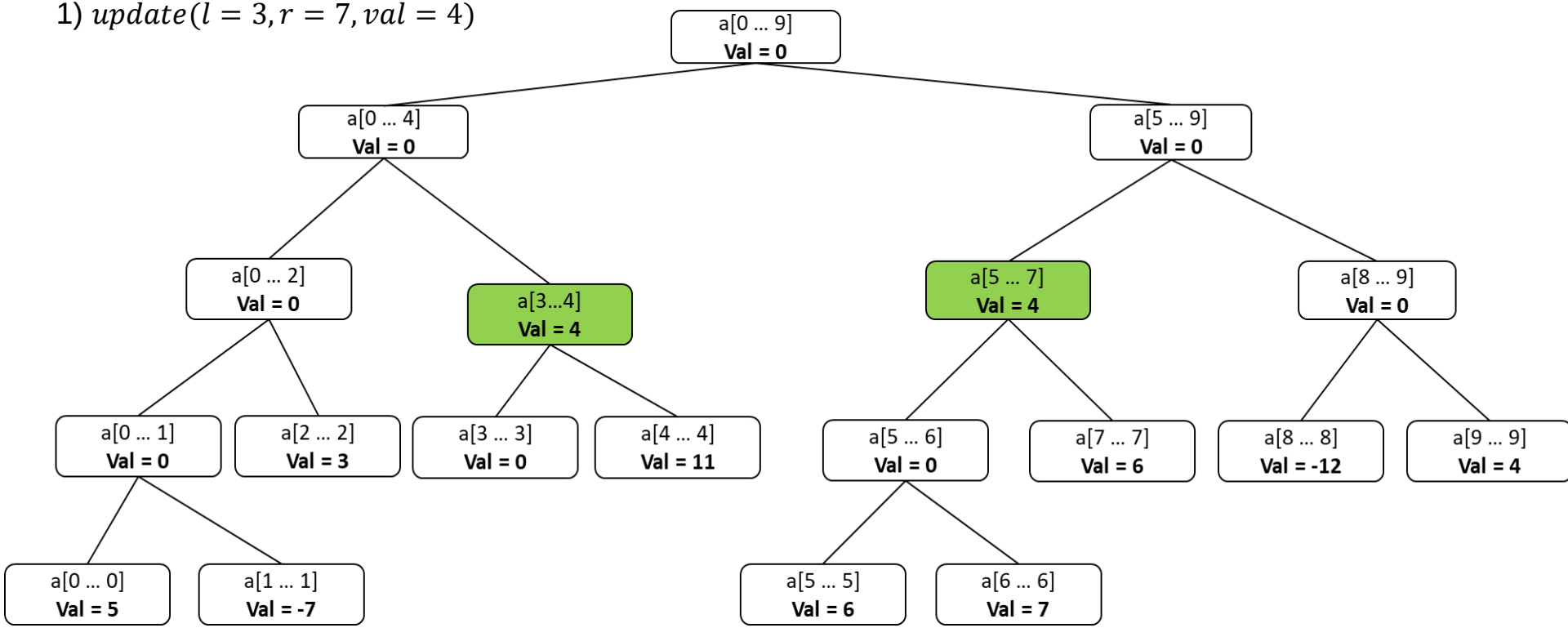
Point Query ,Range Update – Approach 1

Tomemos como ejemplo un arreglo de tamaño $n = 10$ $A = [5, -7, 3, 0, 11, 6, 7, 6, -12, 4]$



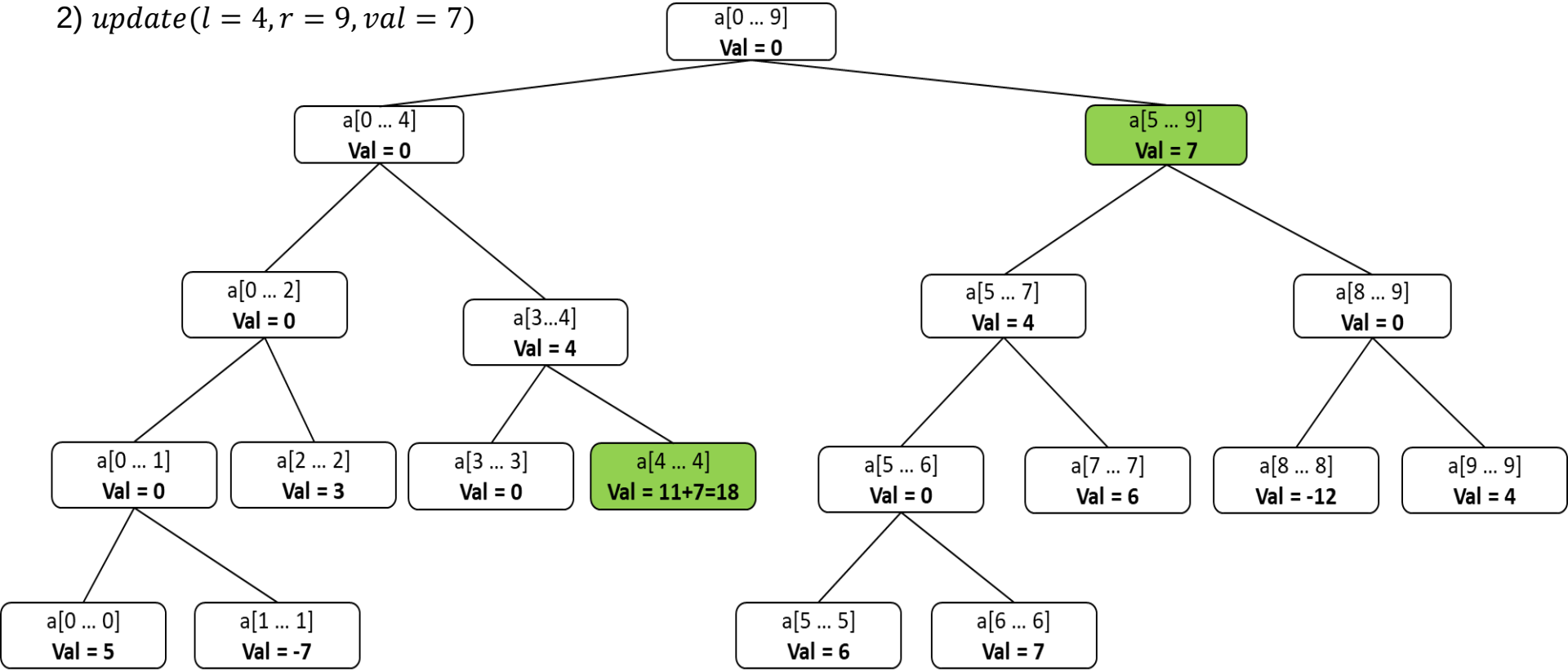
Point Query ,Range Update – Approach 1

1) $update(l = 3, r = 7, val = 4)$



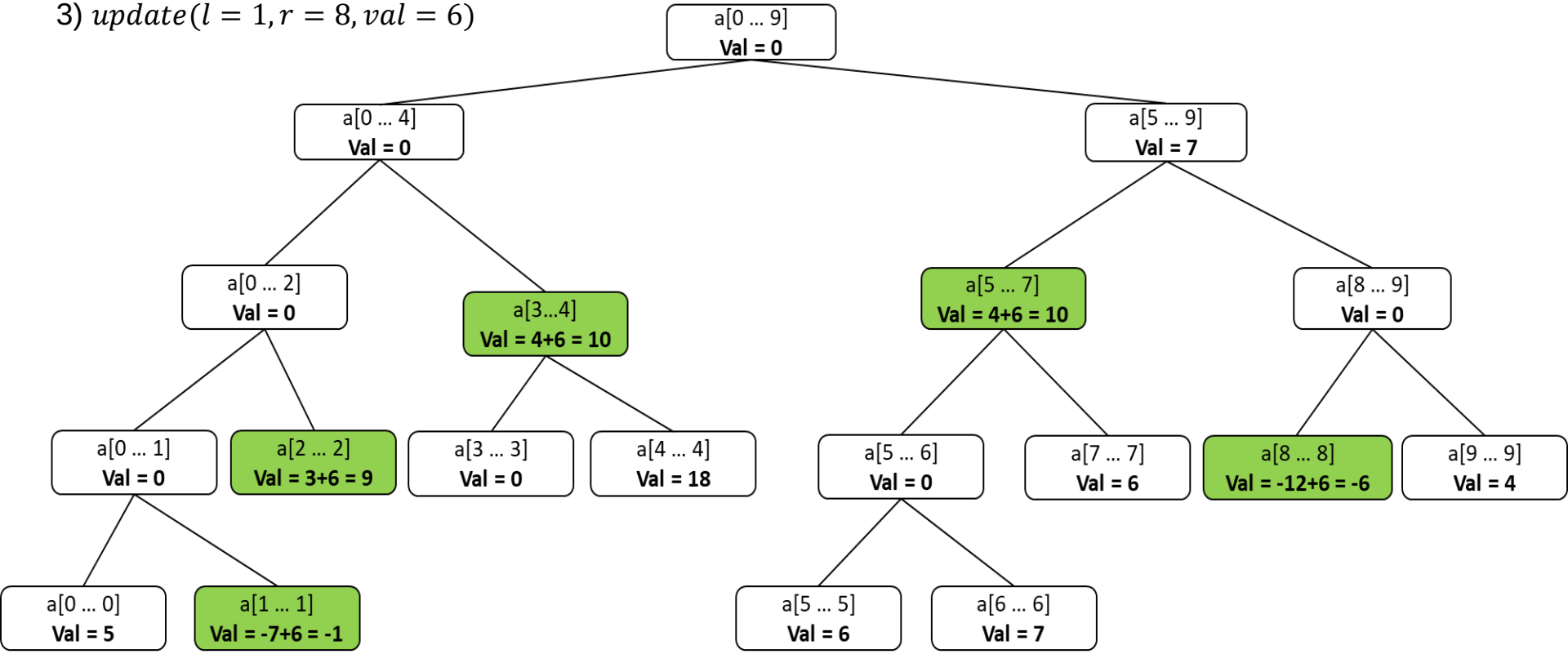
Point Query ,Range Update – Approach 1

2) $update(l = 4, r = 9, val = 7)$



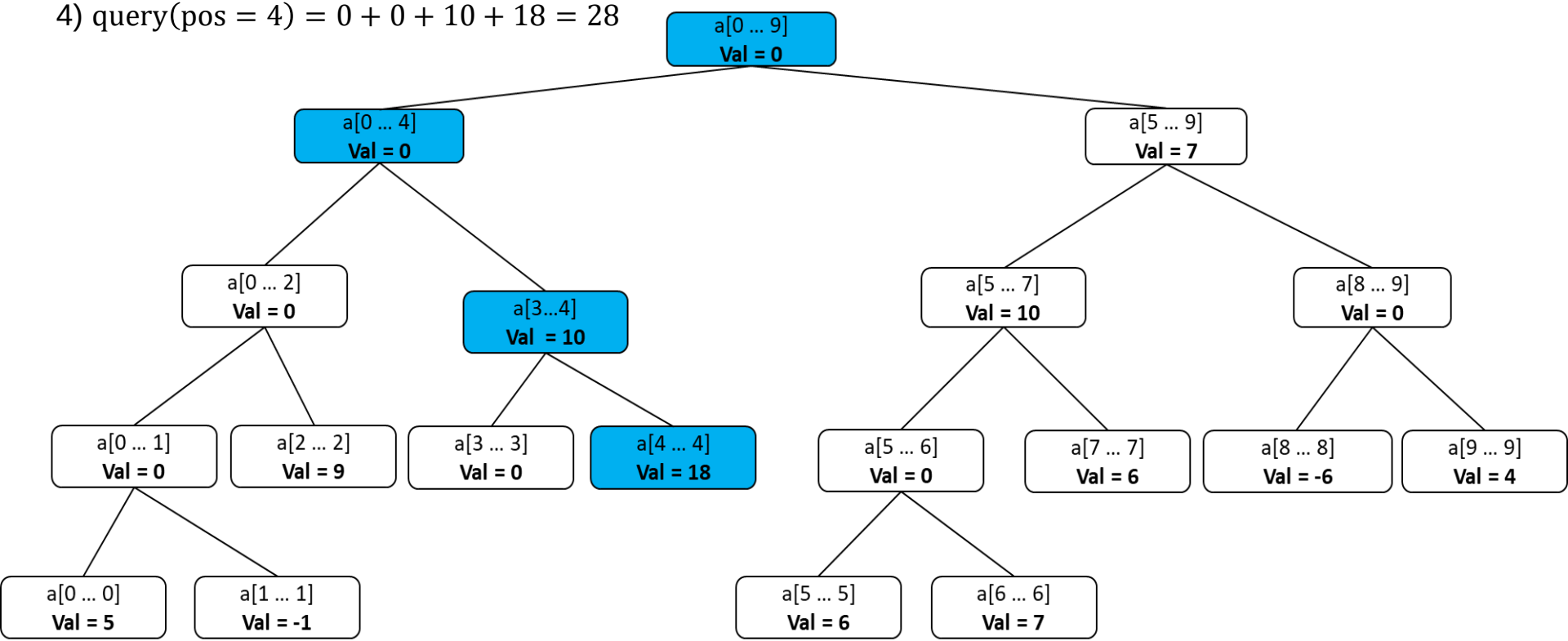
Point Query ,Range Update – Approach 1

3) $update(l = 1, r = 8, val = 6)$



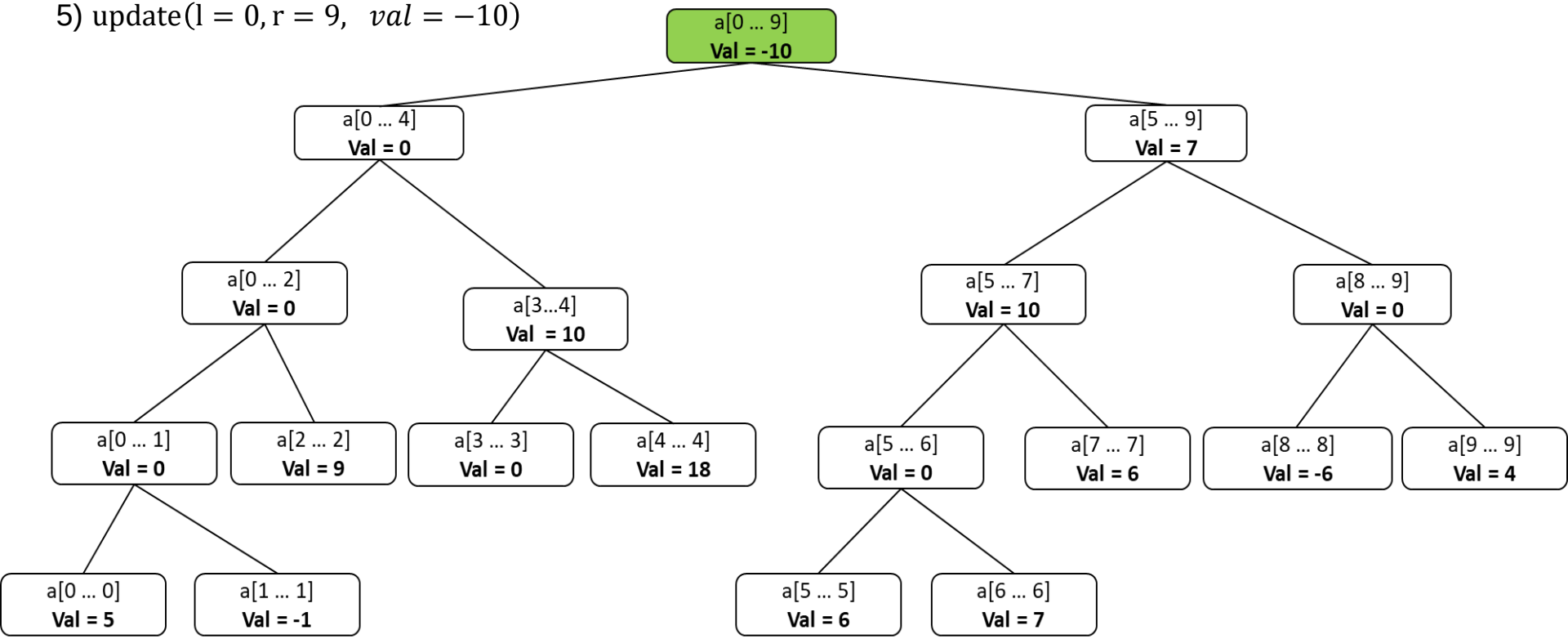
Point Query ,Range Update – Approach 1

4) $\text{query}(\text{pos} = 4) = 0 + 0 + 10 + 18 = 28$



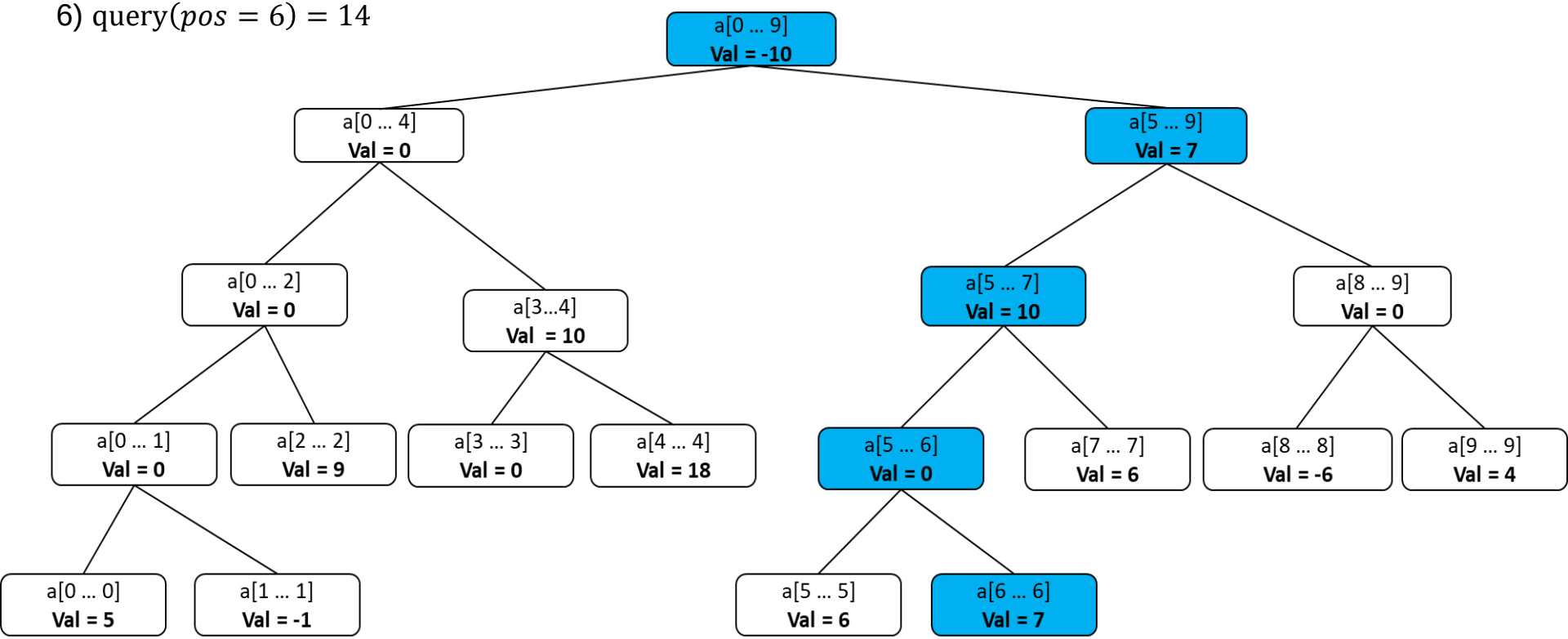
Point Query ,Range Update – Approach 1

5) update$(\mathrm{l} = 0, \mathrm{r} = 9,\ \ val = -10)$



Point Query ,Range Update – Approach 1

6) query$(pos = 6) = 14$



Point Query ,Range Update – Approach 1

Código para el build:

```
void build(vector<Long> &a, Long id, Long tl, Long tr ) { //O(n)
    if (tl == tr) {
        t[id] = a[tl];
    }else {
        Long tm = (tl + tr) / 2;
        Long left = id + 1;
        Long right = id + 2 * (tm - tl + 1) ;
        build(a, left, tl, tm);
        build(a, right, tm + 1, tr);
        t[id] = 0;
    }
}
```

Point Query ,Range Update – Approach 1

Código para el update:

```
void update(Long l, Long r, Long add, Long id, Long tl , Long tr ) { //O(logn)
    if(tr < l || tl > r){
        return;
    }
    if (l <= tl && tr <= r) {
        t[id] += add;
    }else {
        Long tm = (tl + tr) / 2;
        Long left = id + 1;
        Long right = id + 2 * (tm - tl + 1) ;
        update(l, r, add , left, tl, tm);
        update(l, r, add , right, tm + 1, tr);
    }
}
```

Point Query ,Range Update – Approach 1

Código para la query:

```
Long query(Long pos, Long id, Long tl, Long tr) { //O(logn)
    if (tl == tr) {
        return t[id];
    }

    Long tm = (tl + tr) / 2;
    Long left = id + 1;
    Long right = id + 2 * (tm - tl + 1) ;
    if (pos <= tm) {
        return t[id] + query(pos, left, tl, tm);
    }else {
        return t[id] + query(pos, right, tm + 1, tr);
    }
}
```

Point Query ,Range Update – Approach 2 : Lazy Propagation

No siempre es tan fácil utilizar esta técnica, sobre todo cuando **importa el orden** de los updates. Incluso se complicará más cuando veamos los problemas del tipo **range query, range update**. Es por eso que existe otra técnica bastante útil llamada **Lazy Propagation.**

En el caso de point query, range update, la idea es que los updates que se hacen en un nodo del segment tree, finalmente deberían llegar a las hojas del subárbol de ese nodo. Sin embargo, en vez de hacer esta **propagación** de forma abrupta, mejor seamos **lazy** y hagámoslo solo cuando sea realmente necesario. Es decir, cuando una query o update nos obligue a propagar hacia los hijos.

En general hay 3 acciones que debemos de pensar:

1. **Dividir** un update en 2
2. **Acumular** los updates
3. **Reiniciar** el lazy update de un determinado nodo luego de propagar.

   Es decir establecer una especie **update neutro**

Point Query ,Range Update – Approach 2 : Lazy Propagation

Para nuestro ejemplo anterior :

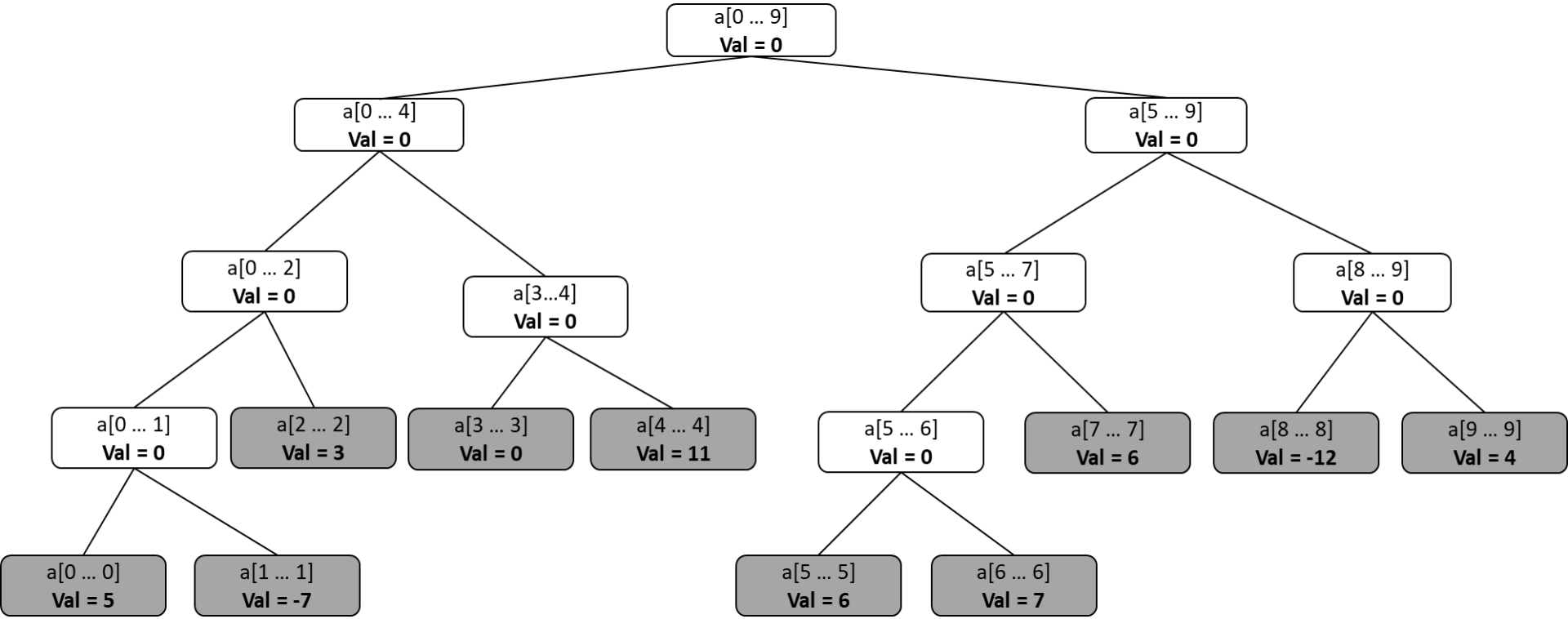
- $query(pos) \rightarrow Retornar\ A[pos]$
- $update(l, r, val) \rightarrow \forall i \in [l, r],\ A[i] \mathrel{+}= val$

Serían las siguientes acciones

1. **Dividir** : Puedes dividir $update(l, r, val)$ en $update(l, p, val), update(p+1, r, val)$
2. **Acumular** los updates : Si hago $update(l, r, x)$ y luego hago $update(l, r, y)$ esto es equivalente a $update(l, r, x+y)$
3. **Reiniciar** el lazy update con el elemento neutro 0.

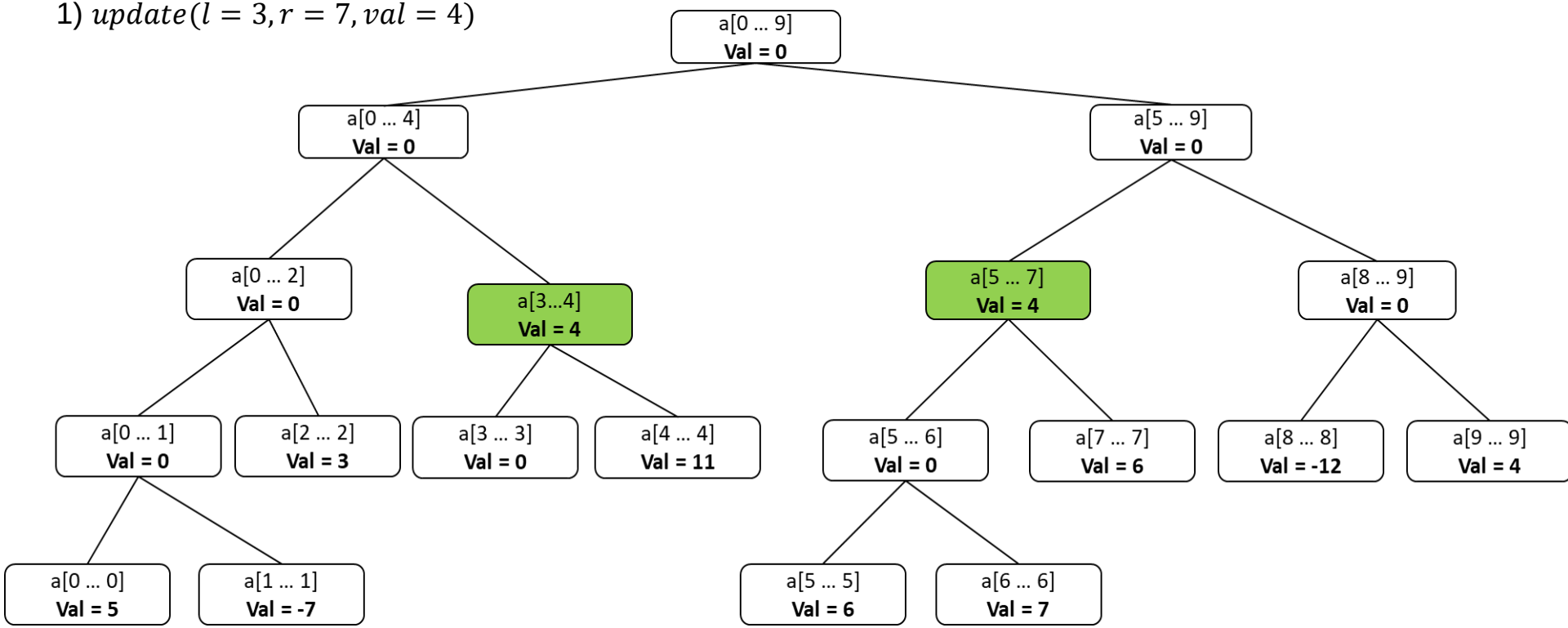
Point Query ,Range Update – Lazy Propagation

Tomemos como ejemplo un arreglo de tamaño $n = 10$ $A = [5, -7, 3, 0, 11, 6, 7, 6, -12, 4]$



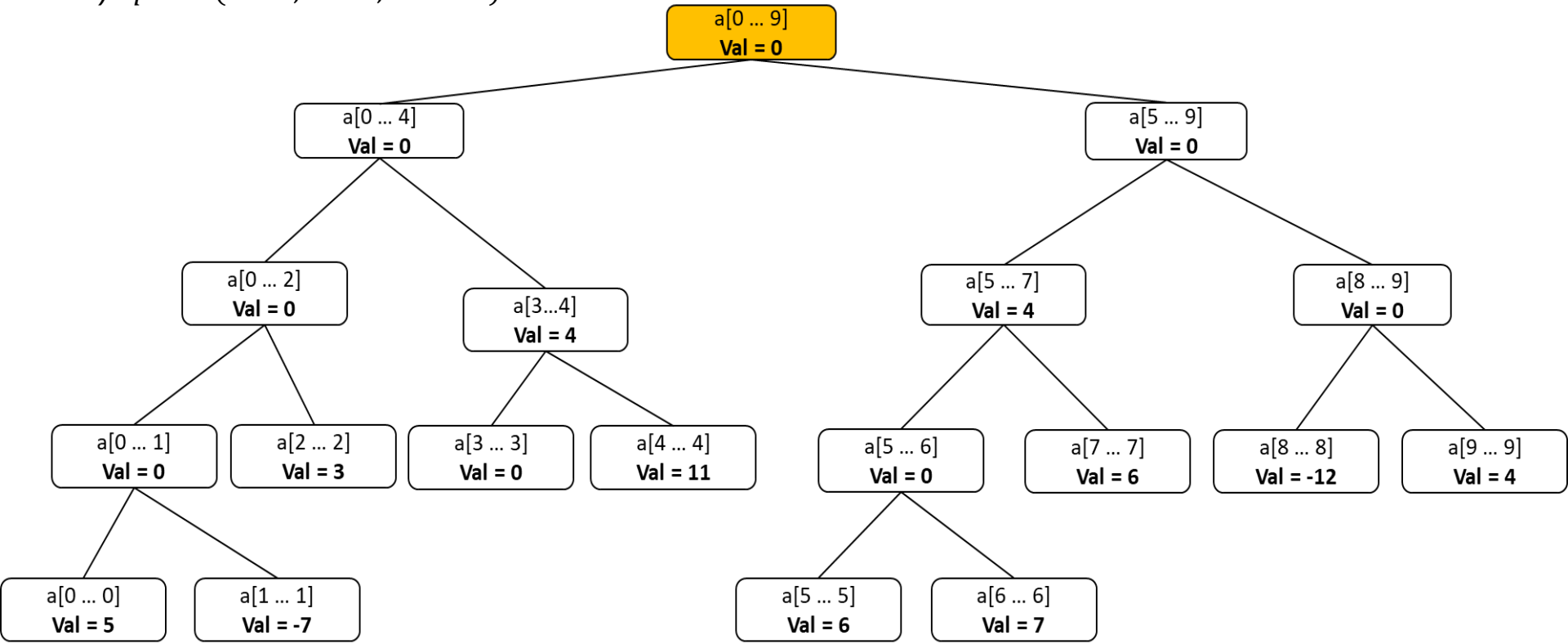
Point Query ,Range Update – Lazy Propagation

1) $update(l = 3, r = 7, val = 4)$



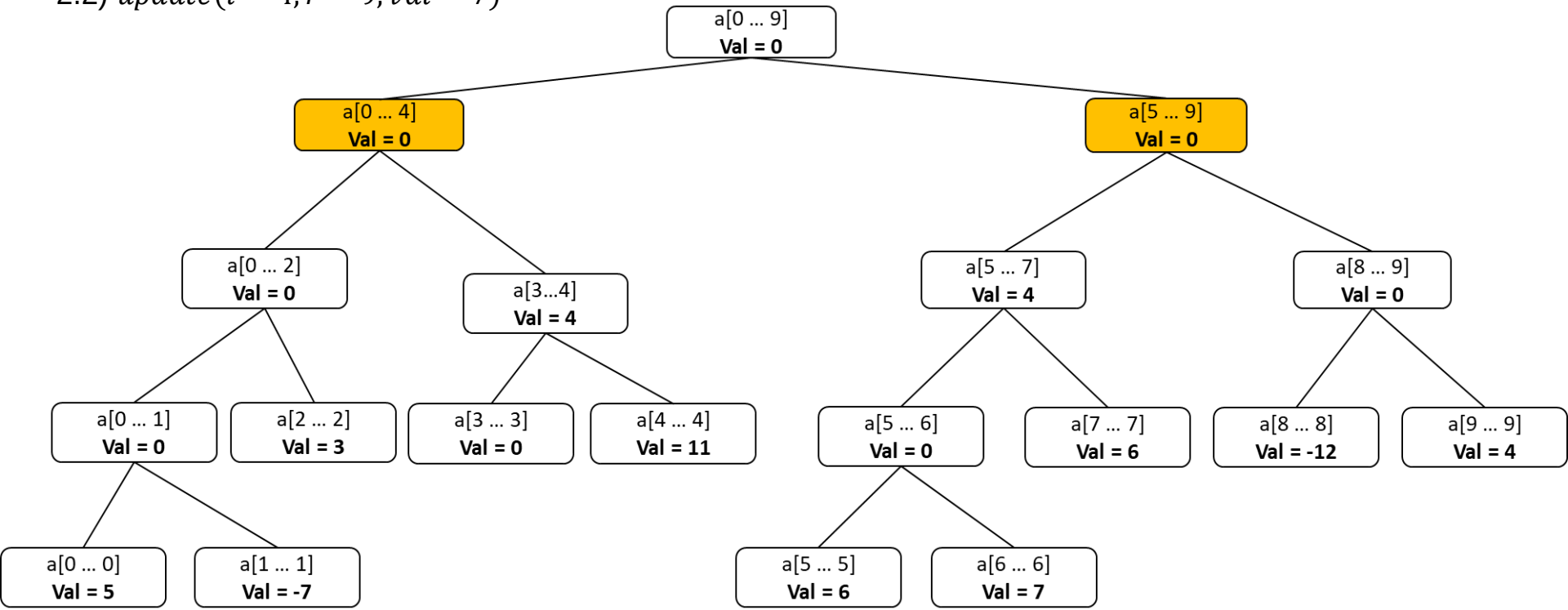
Point Query ,Range Update – Lazy Propagation

2.1) $update(l = 4, r = 9, val = 7)$



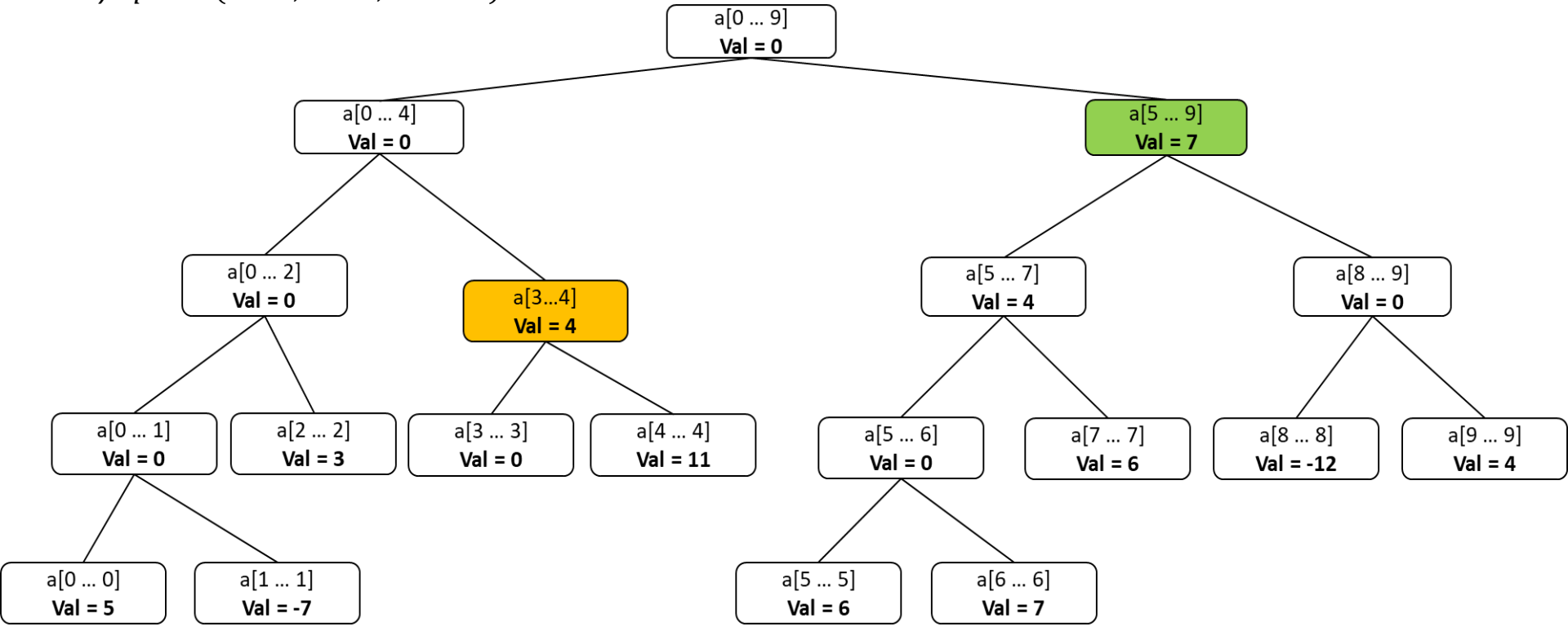
Point Query ,Range Update – Lazy Propagation

2.2) $update(l = 4, r = 9, val = 7)$



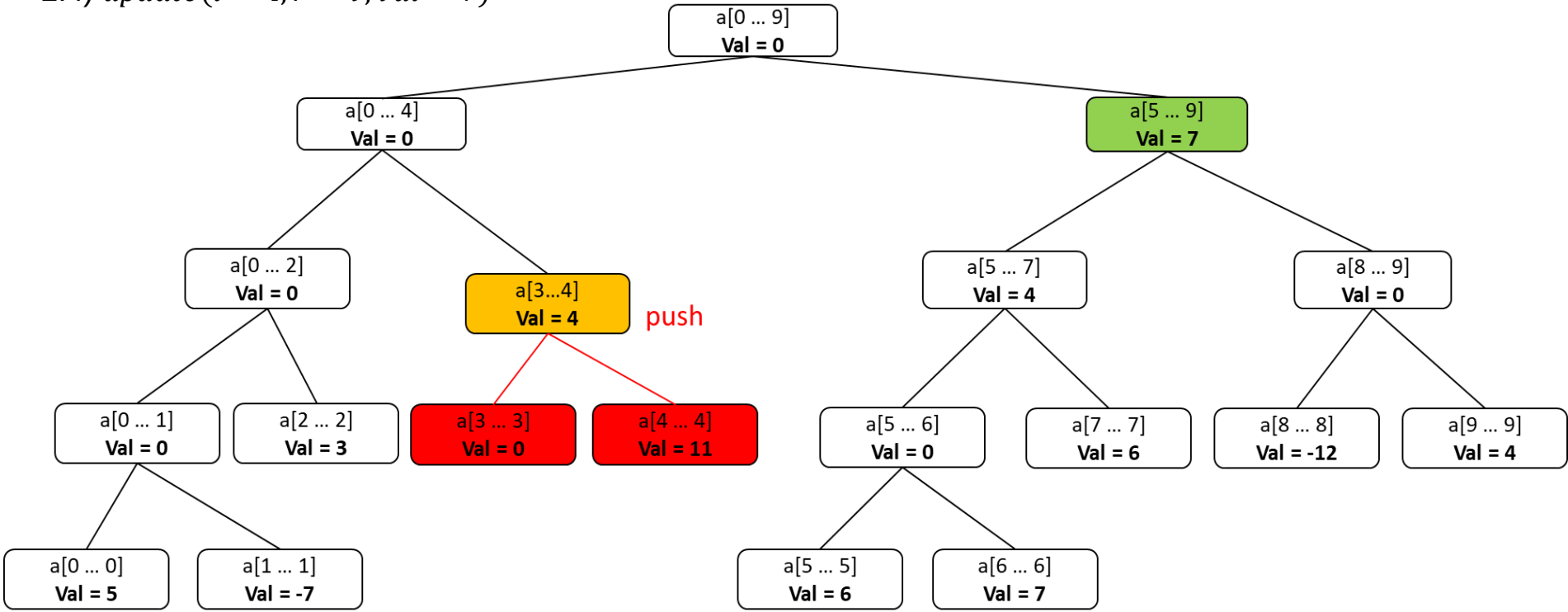
Point Query ,Range Update – Lazy Propagation

2.3) $update(l = 4, r = 9, val = 7)$



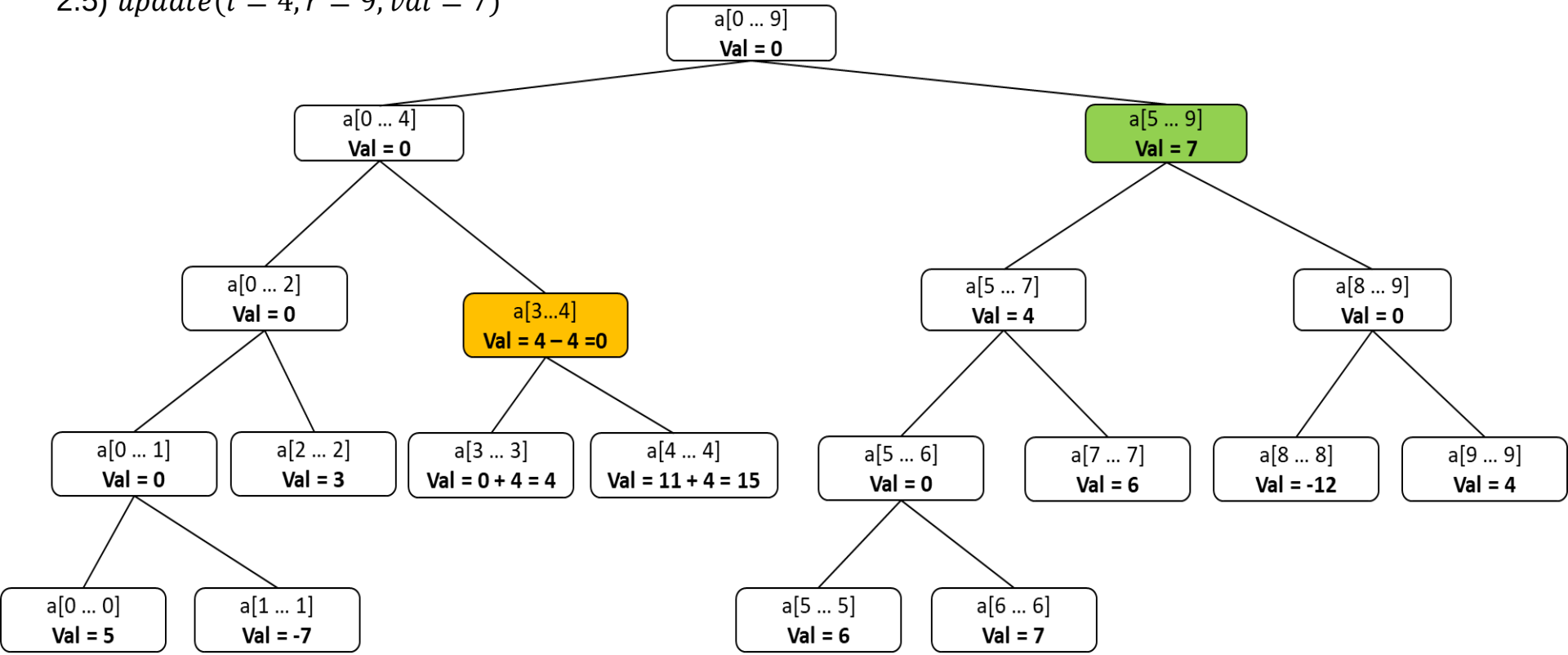
Point Query ,Range Update – Lazy Propagation

2.4) $update(l = 4, r = 9, val = 7)$



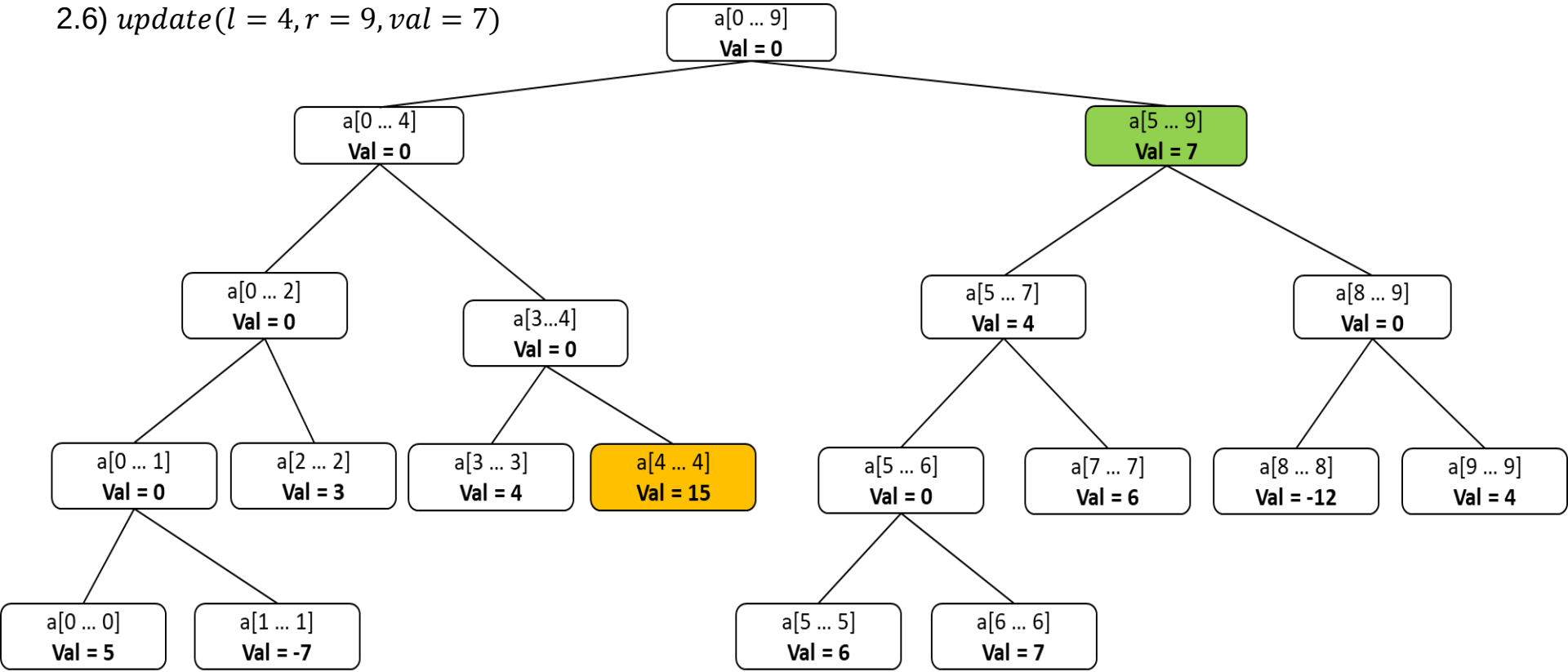
Point Query ,Range Update – Lazy Propagation

2.5) $update(l = 4, r = 9, val = 7)$



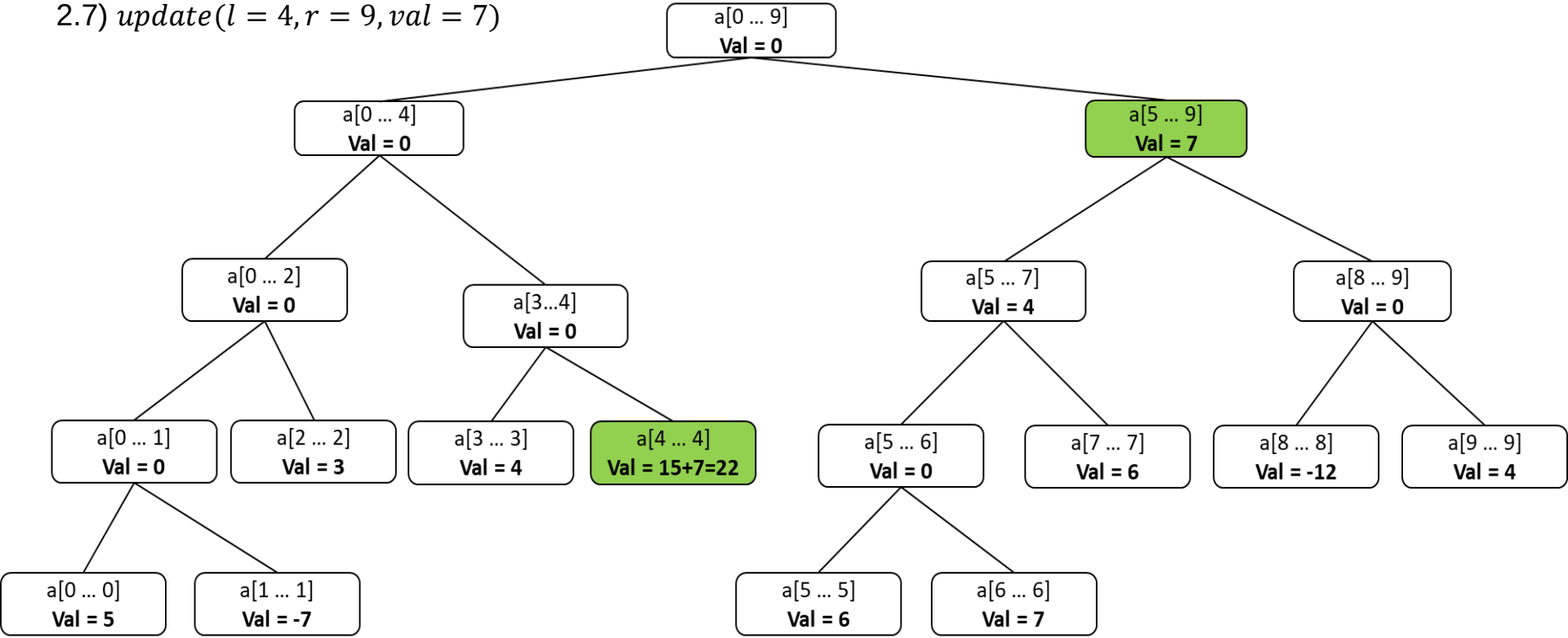
Point Query ,Range Update – Lazy Propagation

2.6) $update(l = 4, r = 9, val = 7)$



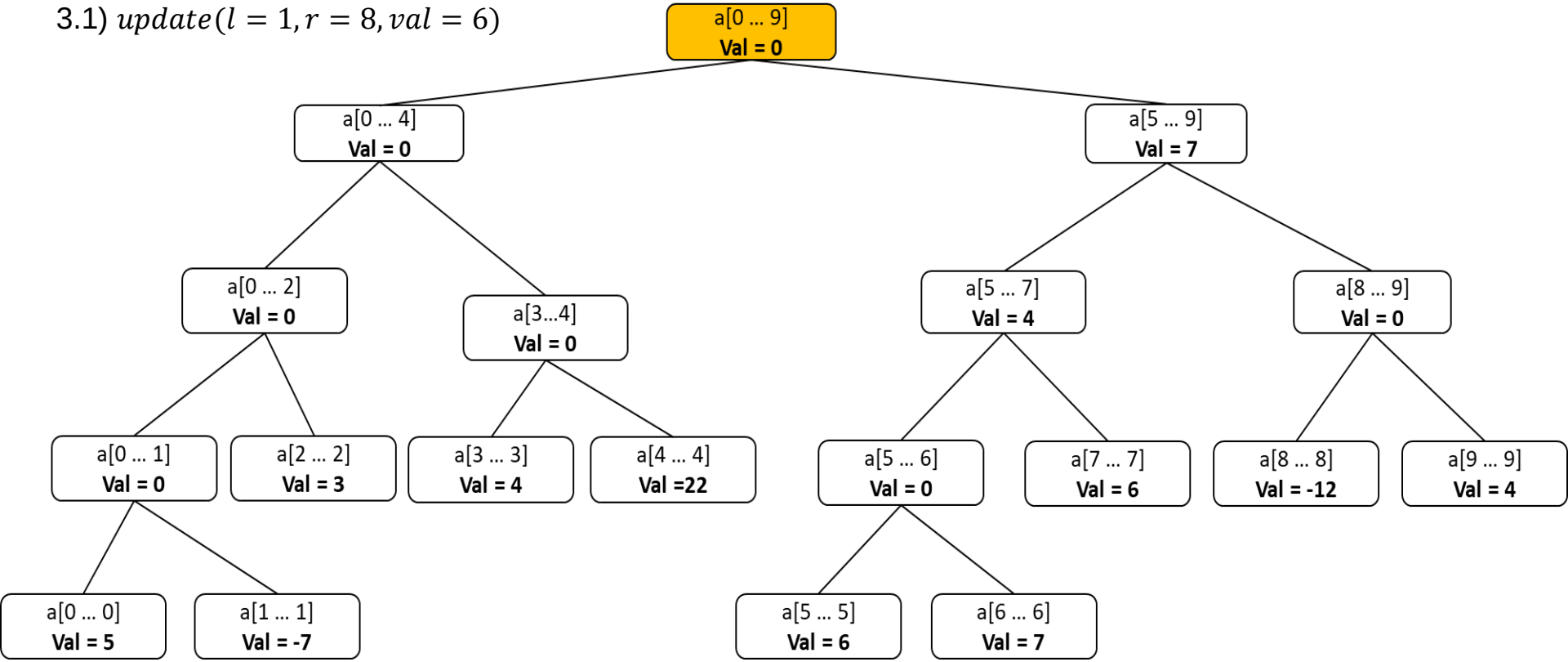
Point Query ,Range Update – Lazy Propagation

2.7) $update(l = 4, r = 9, val = 7)$



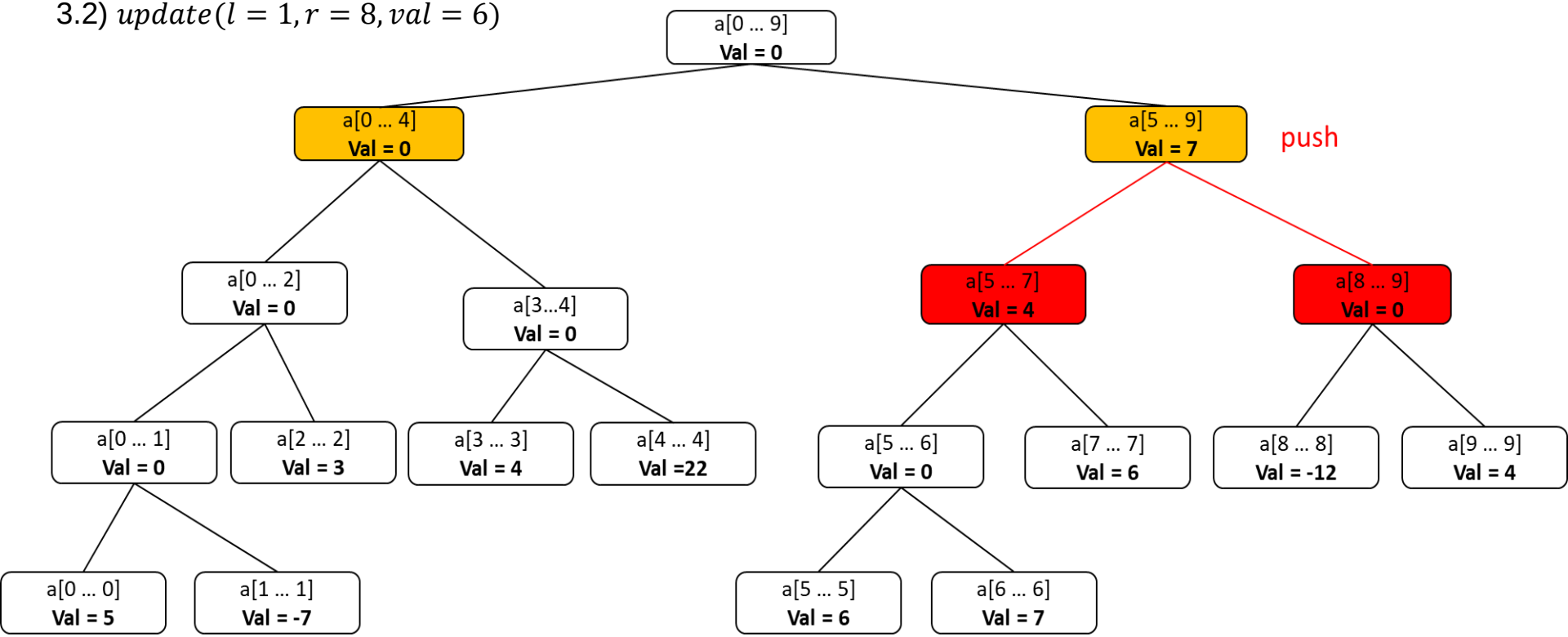
Point Query ,Range Update – Lazy Propagation

3.1) $update(l = 1, r = 8, val = 6)$



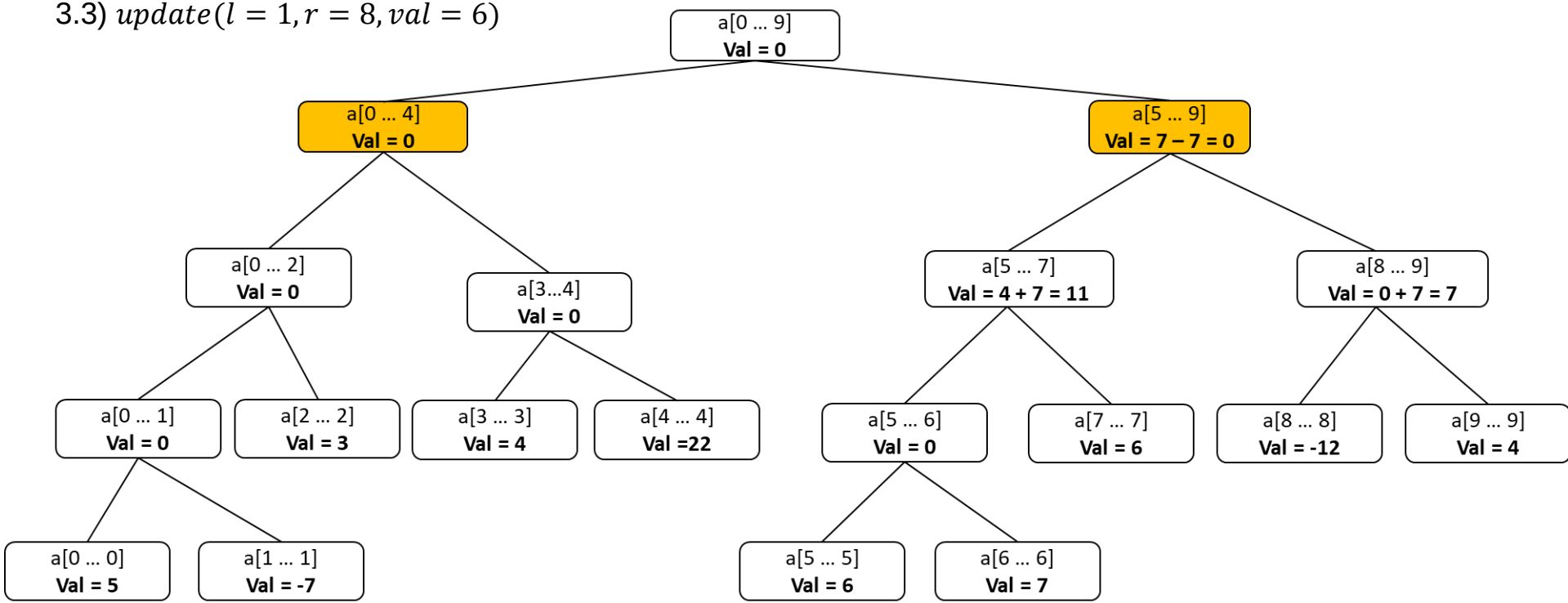
Point Query ,Range Update – Lazy Propagation

3.2) $update(l = 1, r = 8, val = 6)$



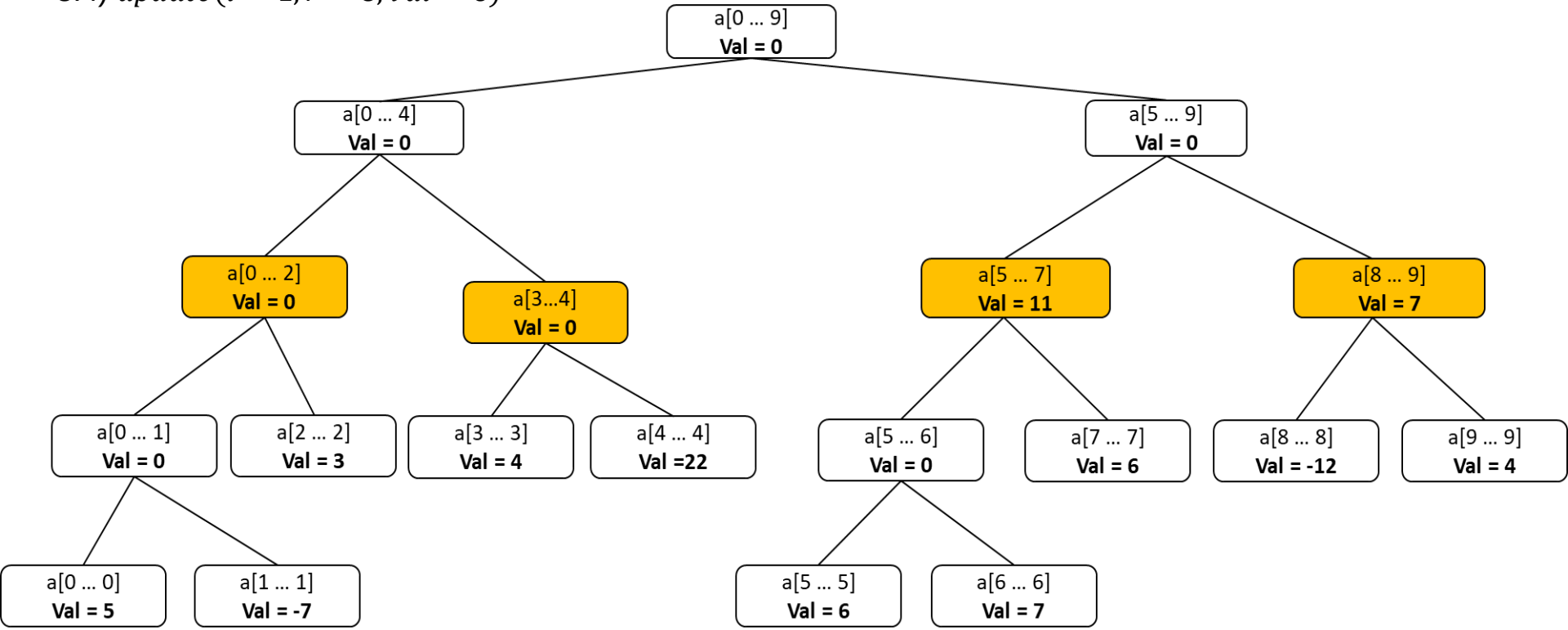
Point Query ,Range Update – Lazy Propagation

3.3) $update(l = 1, r = 8, val = 6)$



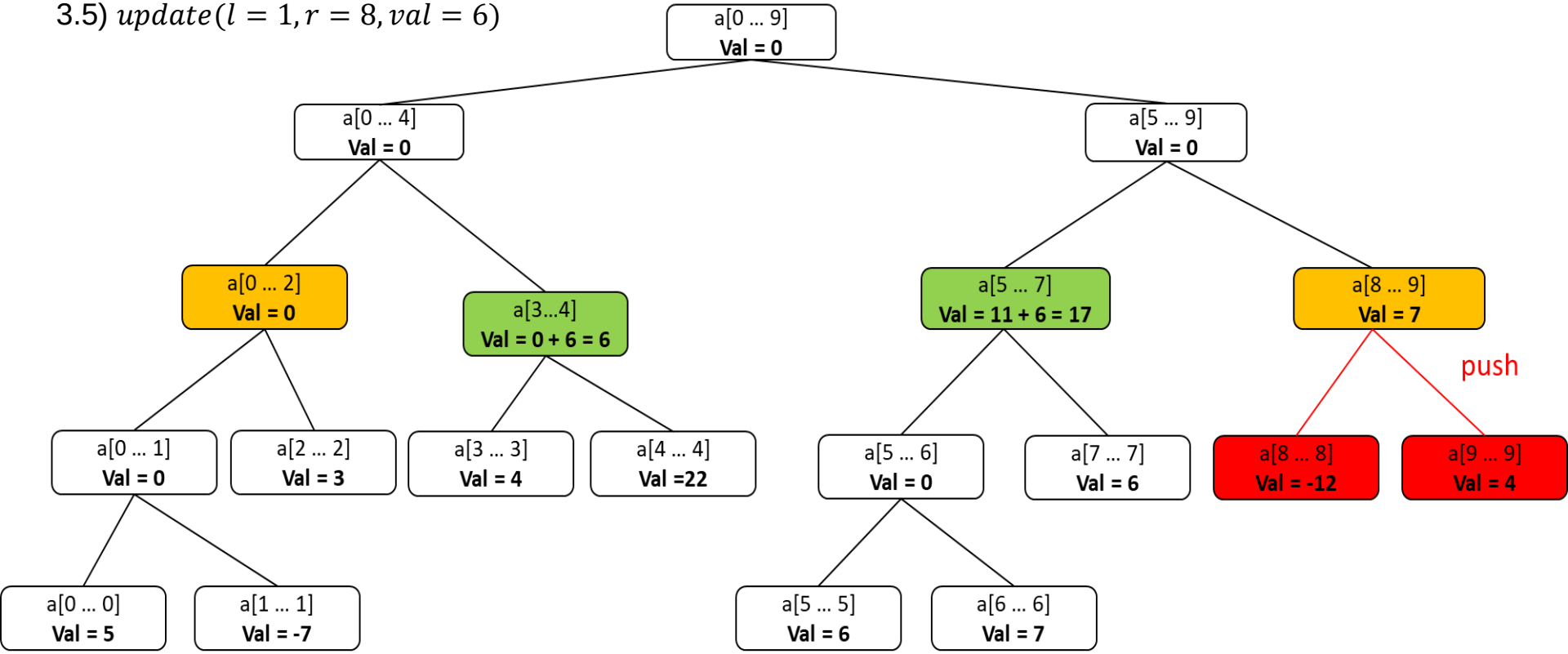
Point Query ,Range Update – Lazy Propagation

3.4) $update(l = 1, r = 8, val = 6)$



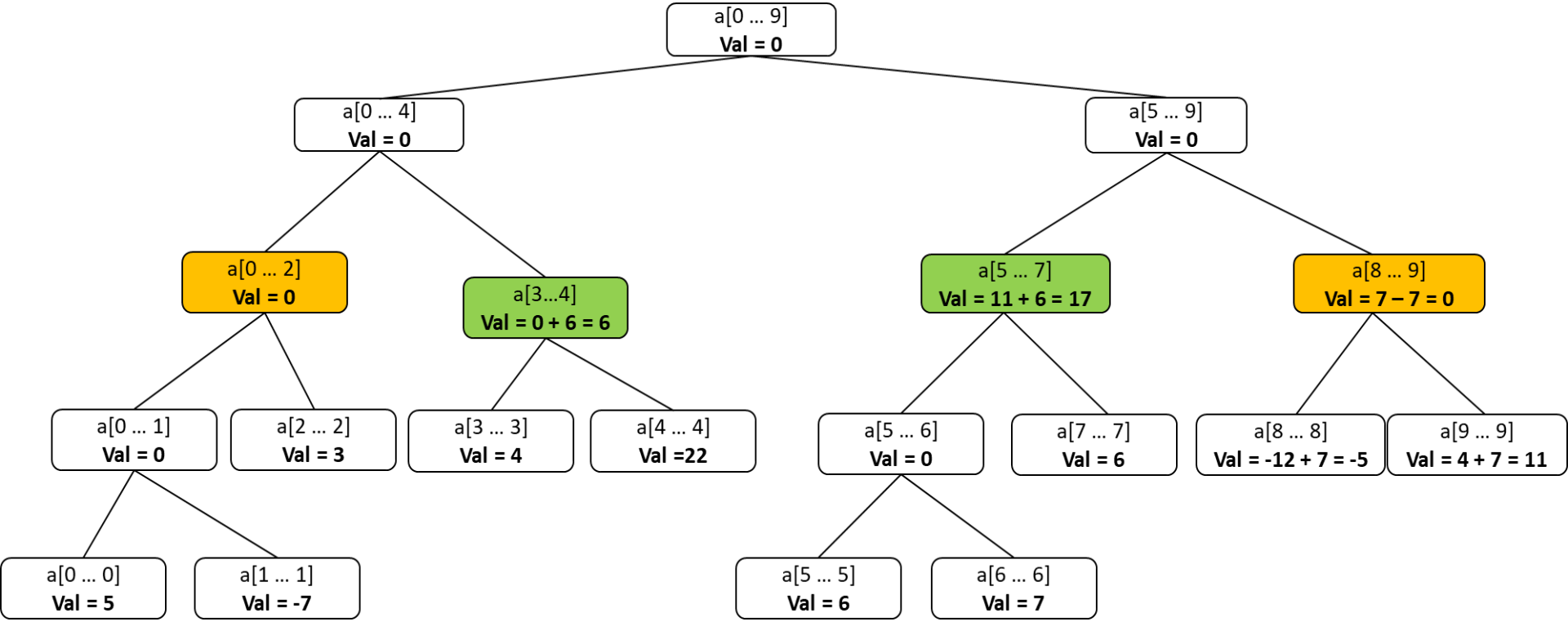
Point Query ,Range Update – Lazy Propagation

3.5) $update(l = 1, r = 8, val = 6)$



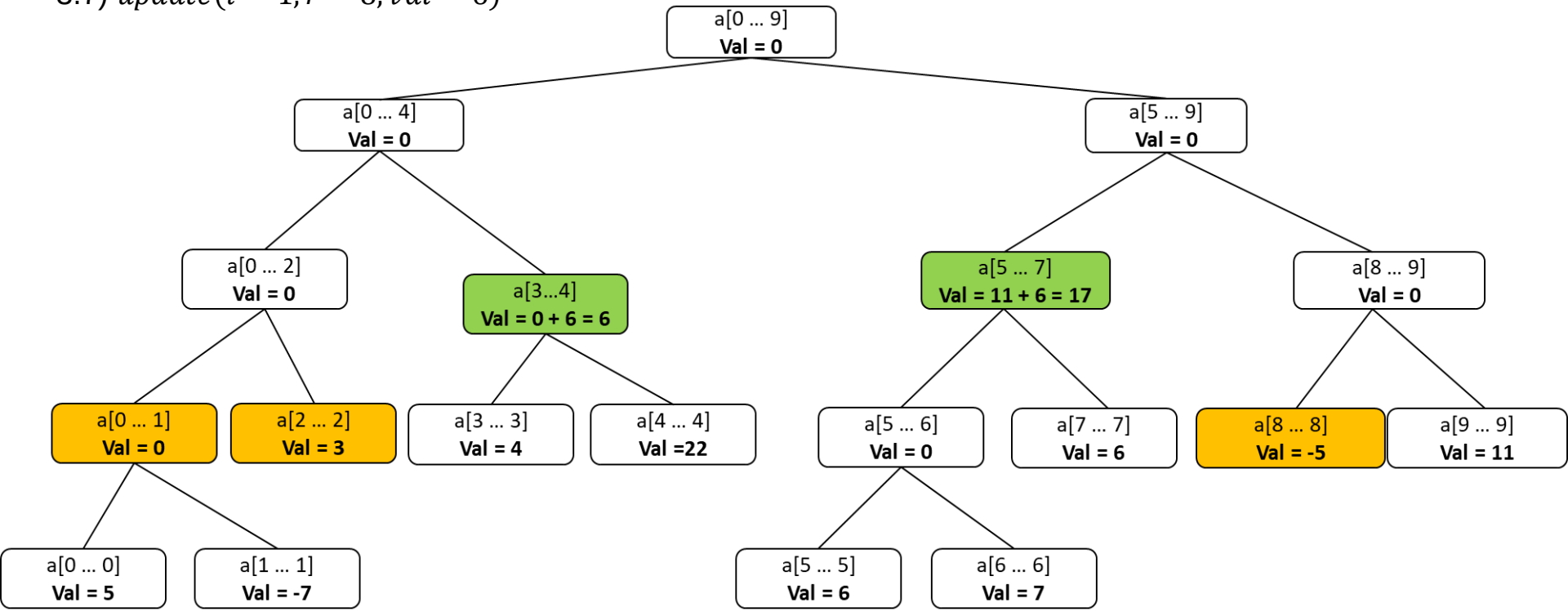
Point Query ,Range Update – Lazy Propagation

3.6) $update(l = 1, r = 8, val = 6)$



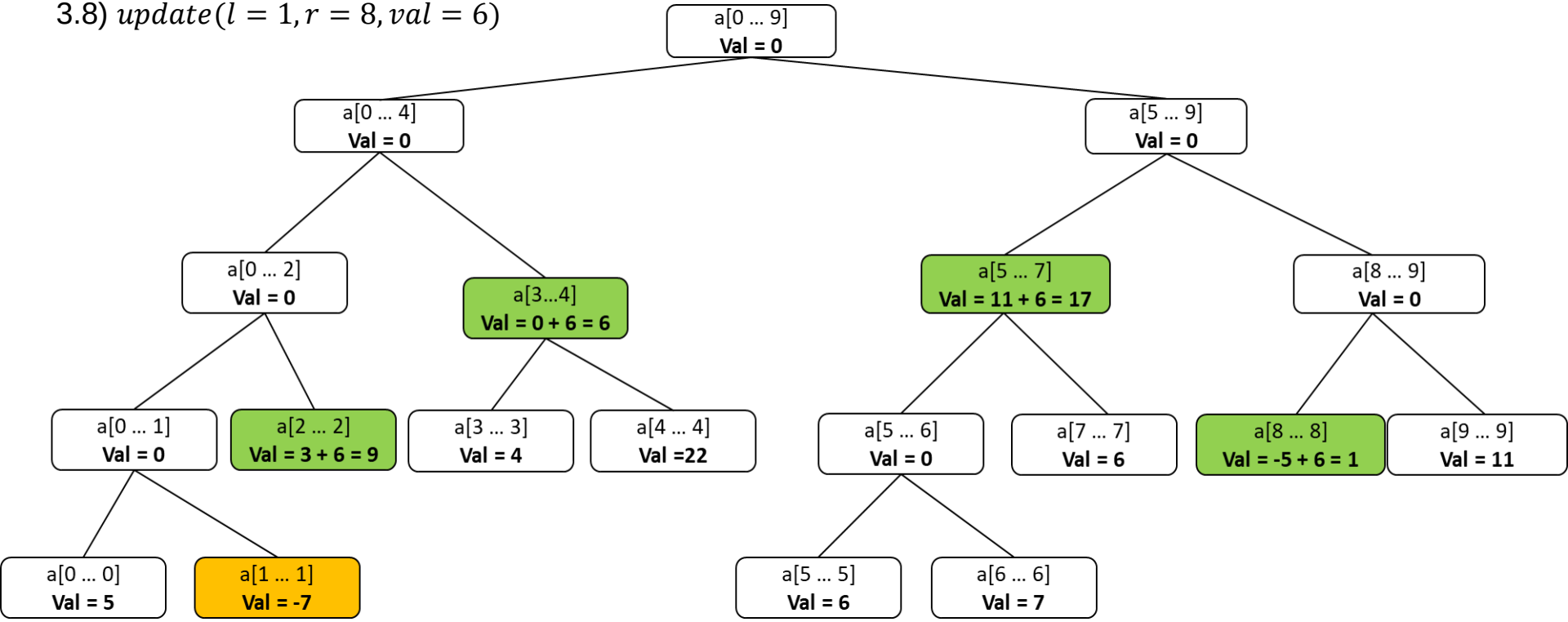
Point Query ,Range Update – Lazy Propagation

3.7) $update(l = 1, r = 8, val = 6)$



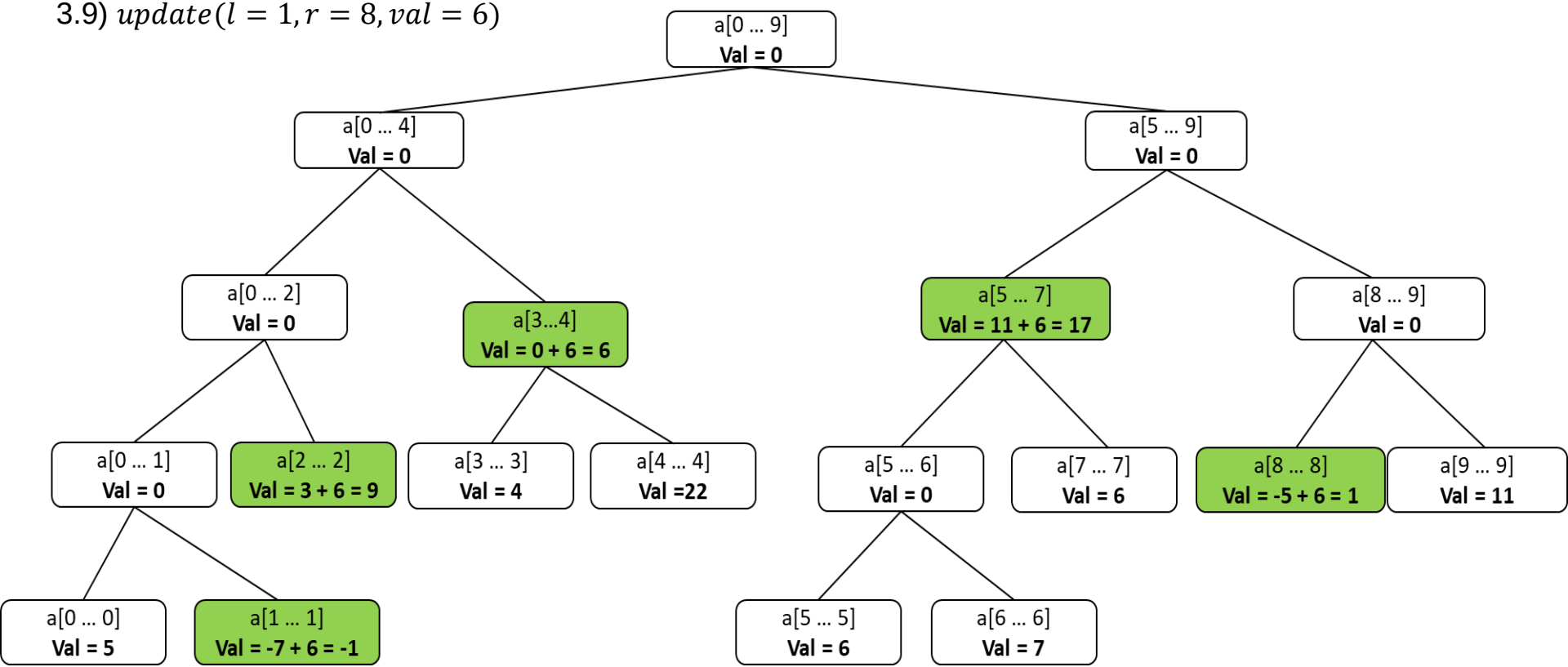
Point Query ,Range Update – Lazy Propagation

3.8) $update(l = 1, r = 8, val = 6)$



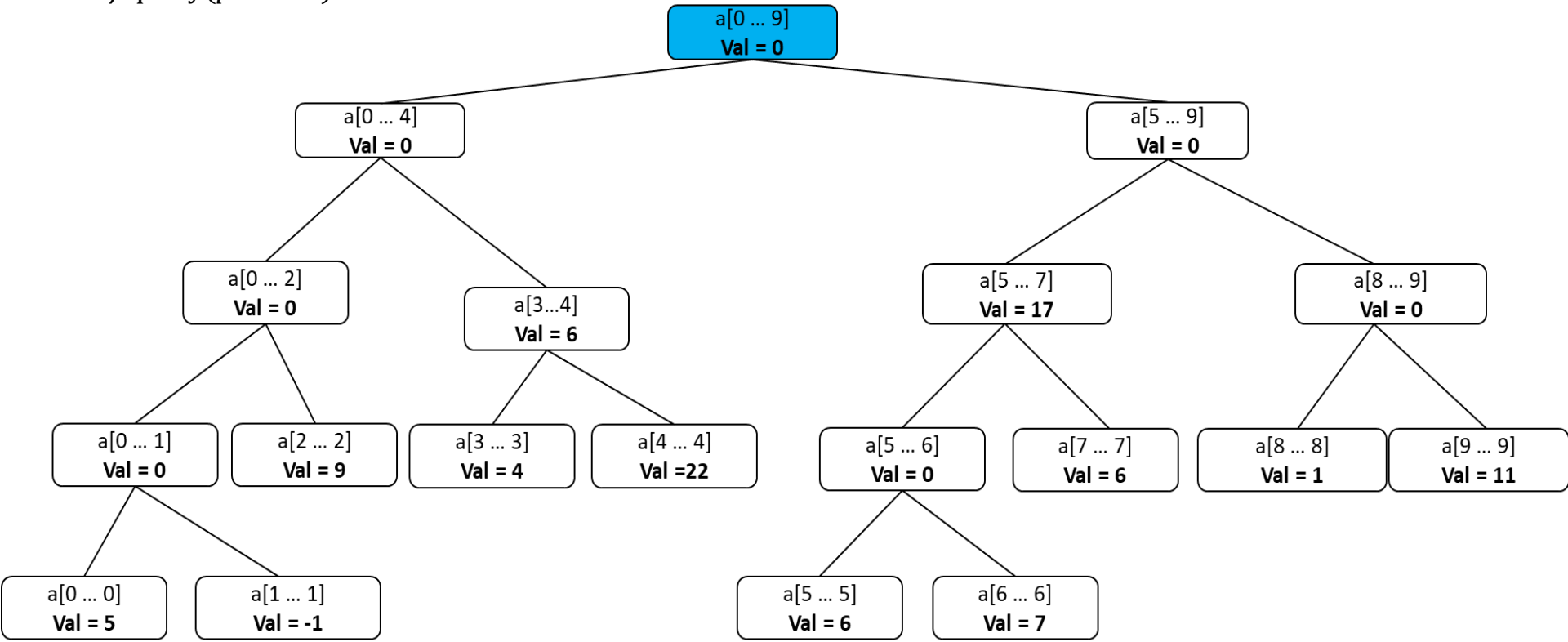
Point Query ,Range Update – Lazy Propagation

3.9) $update(l = 1, r = 8, val = 6)$



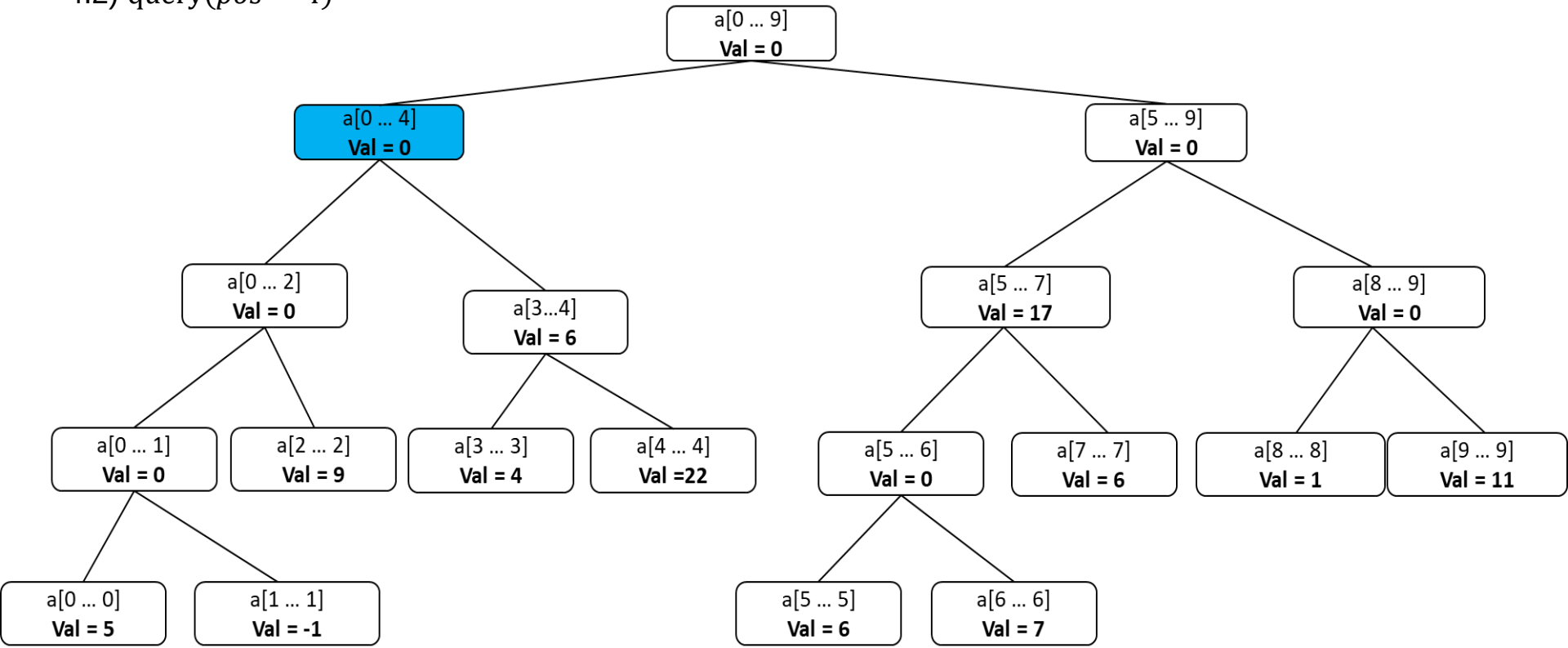
Point Query ,Range Update – Lazy Propagation

4.1) query($pos = 4$)



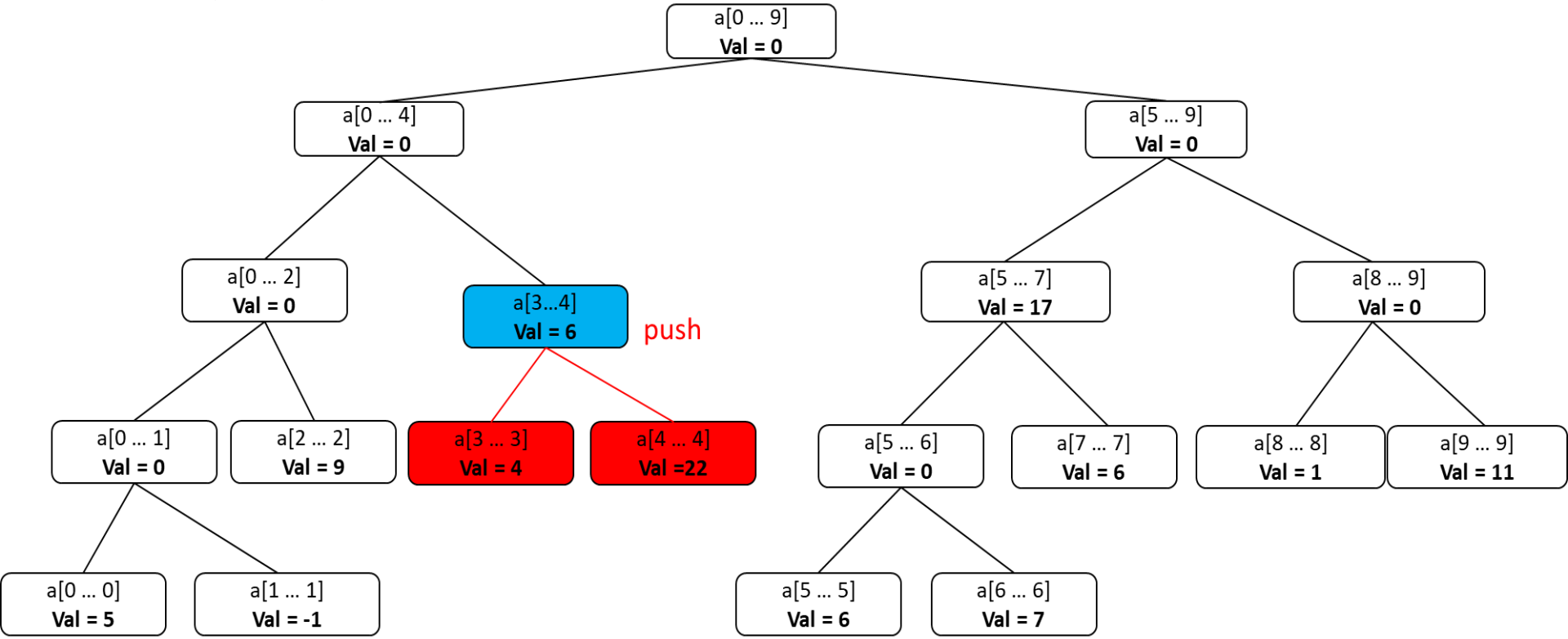
Point Query ,Range Update – Lazy Propagation

4.2) query($pos = 4$)



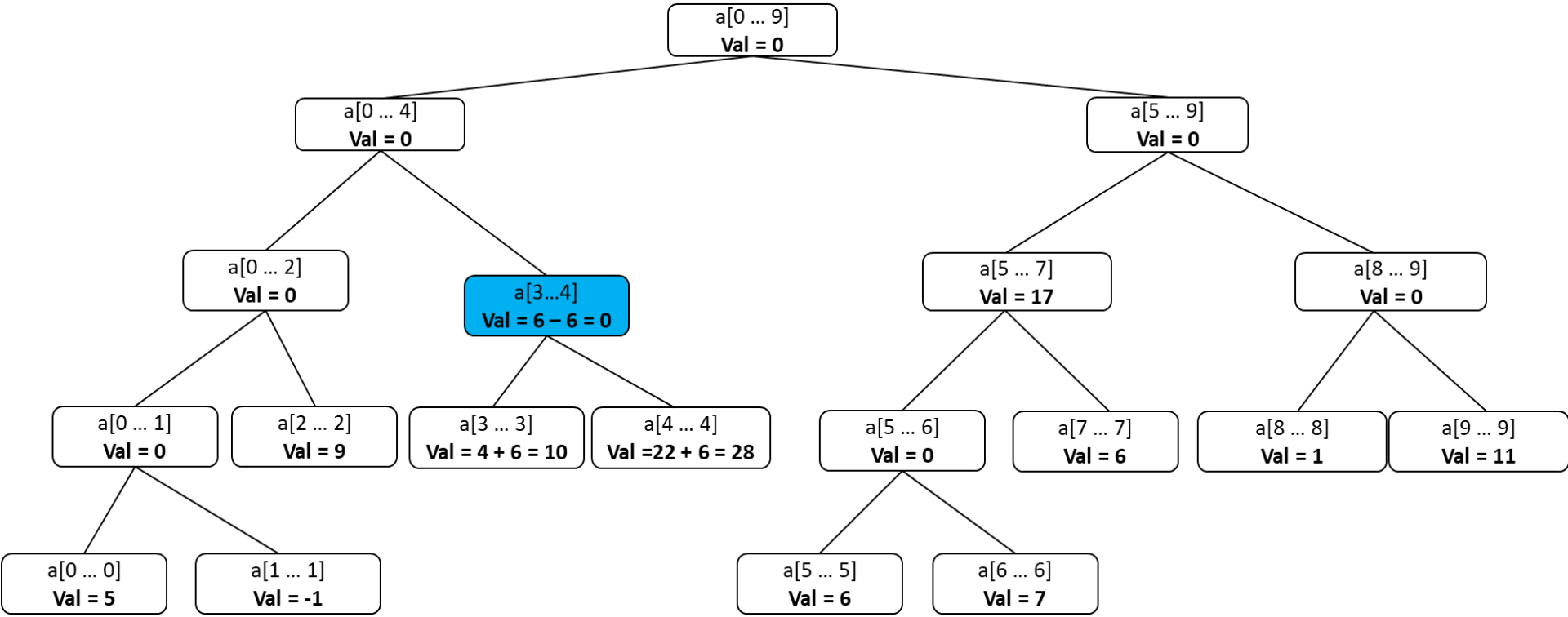
Point Query ,Range Update – Lazy Propagation

4.3) query($pos = 4$)



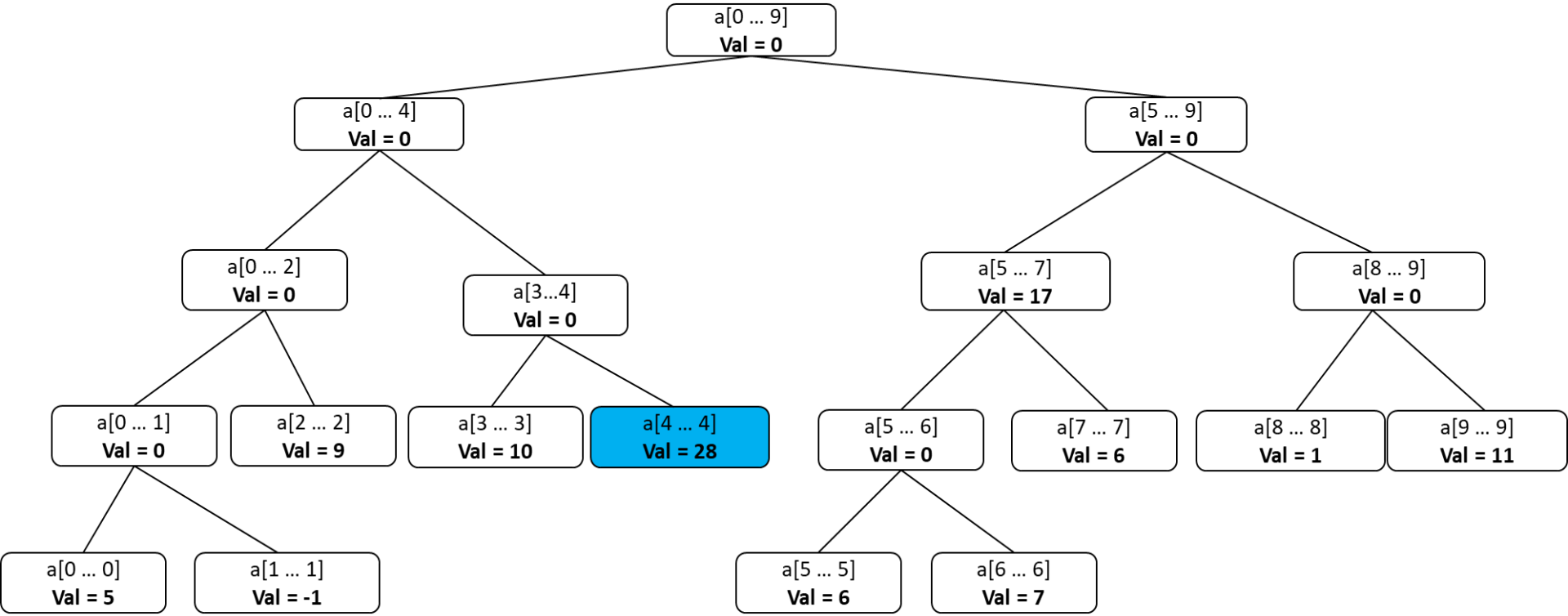
Point Query ,Range Update – Lazy Propagation

4.4) query($pos = 4$)



Point Query ,Range Update – Lazy Propagation

4.5) query($pos = 4$)



Point Query ,Range Update – Lazy Propagation

Para la propagación, crearemos una nueva función que lo haga. Usaremos un arreglo $lazy[\,]$ en lugar del arreglo $t[\,]$ (reservemos $t[\,]$ para las queries)


```
void push(Long id){
    Long left = 2 * id;
    Long right = 2 * id + 1;

    //aggregate the lazy value of the node to the lazy value of the children
    lazy[left] += lazy[id];
    lazy[right] += lazy[id];

    //restart the lazy value
    lazy[id] = 0;
}
```

Point Query ,Range Update – Lazy Propagation

```
void build(vector<Long> &a, Long id, Long tl, Long tr ) { //O(n)
    if (tl == tr) {
        lazy[id] = a[tl];
    }else {
        Long tm = (tl + tr) / 2;
        Long left = 2 * id;
        Long right = 2 * id + 1 ;
        build(a, left, tl, tm);
        build(a, right, tm + 1, tr);
        lazy[id] = 0;
    }
}
```

Point Query ,Range Update – Lazy Propagation

```
Long query(Long pos, Long id, Long tl, Long tr) { //O(logn)
    if (tl == tr) {
        return lazy[id];
    }
    Long tm = (tl + tr) / 2;
    Long left = 2 * id;
    Long right = 2 * id + 1 ;
    push(id);
    if (pos <= tm) {
        return query(pos, left, tl, tm);
    }else {
        return query(pos, right, tm + 1, tr);
    }
}
```

Point Query ,Range Update – Lazy Propagation

```
void update(Long l, Long r, Long add, Long id, Long tl , Long tr ) { //O(logn)
    if(tr < l || tl > r){
        return;
    }
    if (l <= tl && tr <= r) {
        //aggregate update
        lazy[id] += add;
    }else {
        Long tm = (tl + tr) / 2;
        Long left = 2 * id;
        Long right = 2 * id + 1 ;
        push(id);
        update(l, r, add , left, tl, tm);
        update(l, r, add , right, tm + 1, tr);
    }
}
```

Point Query ,Range Update – Lazy Propagation

Otro problema que también se puede resolver con esta técnica es el siguiente

- $query(pos) \rightarrow Retornar\ A[pos]$
- $update(l, r, val) \rightarrow \forall i \in [l, r],\ A[i] = val$

Es decir, asignar un valor a todo un rango. Repasemos las 3 cosas que debemos hacer

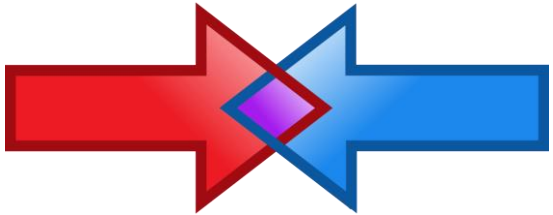
1. **Dividir :** Puedes dividir $update(l, r, val)$ en $update(l, p, val), update(p + 1, r, val)$
2. **Acumular :** Supongamos que tenemos un update en donde asignamos a todo un rango el valor $val1$ y luego viene otro update en donde asignamos al mismo rango el valor $val2$. En el update acumulado, quedará la asignación más reciente, es decir $val2$.
3. **Reiniciar:** Para indicar que un nodo ya no debe propagar podemos usar un arreglo booleano que estará en true, si el nodo necesita propagar.

Point Query ,Range Update – Lazy Propagation

```
bool marked[4 * MX];
void push(Long id){ //O(1)
    if(marked[id]){
        Long left = 2 * id;
        Long right = 2 * id + 1 ;

        //aggregate the lazy value of the node to the lazy value of the children
        lazy[left] = lazy[right] = lazy[id];
        marked[left] = marked[right] = true;

        //restart the lazy value
        marked[id] = false;
    }
}

void update(Long l, Long r, Long val, Long id, Long tl, Long tr ){ //O(logn)
    //...
    if (l <= tl && tr <= r) {
        //aggregate update
        lazy[id] = val;
        marked[id] = true;
    }else {
        //...
    }
}
```

Range Query ,Range Update – Lazy Propagation

Por último, tenemos los problemas más complicados, del tipo **range query , range update**.
Veremos que aquí también podemos aplicar **lazy propagation**.



Veamos el siguiente problema :

- $query(l, r) \rightarrow Retornar \max(A[l], A[l+1], \ldots, A[r])$
- $update(l, r, val) \rightarrow \forall i \in [l, r],\ A[i] \mathrel{+}= val$

La idea será juntar las ideas que hemos hecho en **range query – point update** y en **point query – range update**.
Cuando hagamos el update, dividiremos el rango del update en nodos del segment tree. Pero supongamos que nos ubicamos en un nodo que corresponde al intervalo $[tl, tr]$, y el update nos dice que debemos añadir val a todo ese rango. Al hacer eso, ¿cómo varía la respuesta de ese nodo del segment tree? (Es decir como varía el $t[id]$). Siempre tenemos que pensar cómo haremos esto en O(1) o similar para cualquier problema de este tipo en que se usa **lazy propagation**.

Range Query ,Range Update – Lazy Propagation

Cuando se haga lazy propagation tenemos que pensar cómo haremos las siguientes 4 acciones:

1. **Dividir** el update en 2
2. **Aplicar** el update en un determinado rango (nodo del segment tree) y actualizar eficientemente el valor actual del operador $\bigstar$ aplicado en el rango de un nodo.
3. **Acumular** los updates
4. **Reiniciar** el lazy update de un determinado nodo. Es decir establecer una especie **update neutro**

La idea es hacer todos estos pasos en $O(1)$ o similar para no perjudicar la performance del segment tree

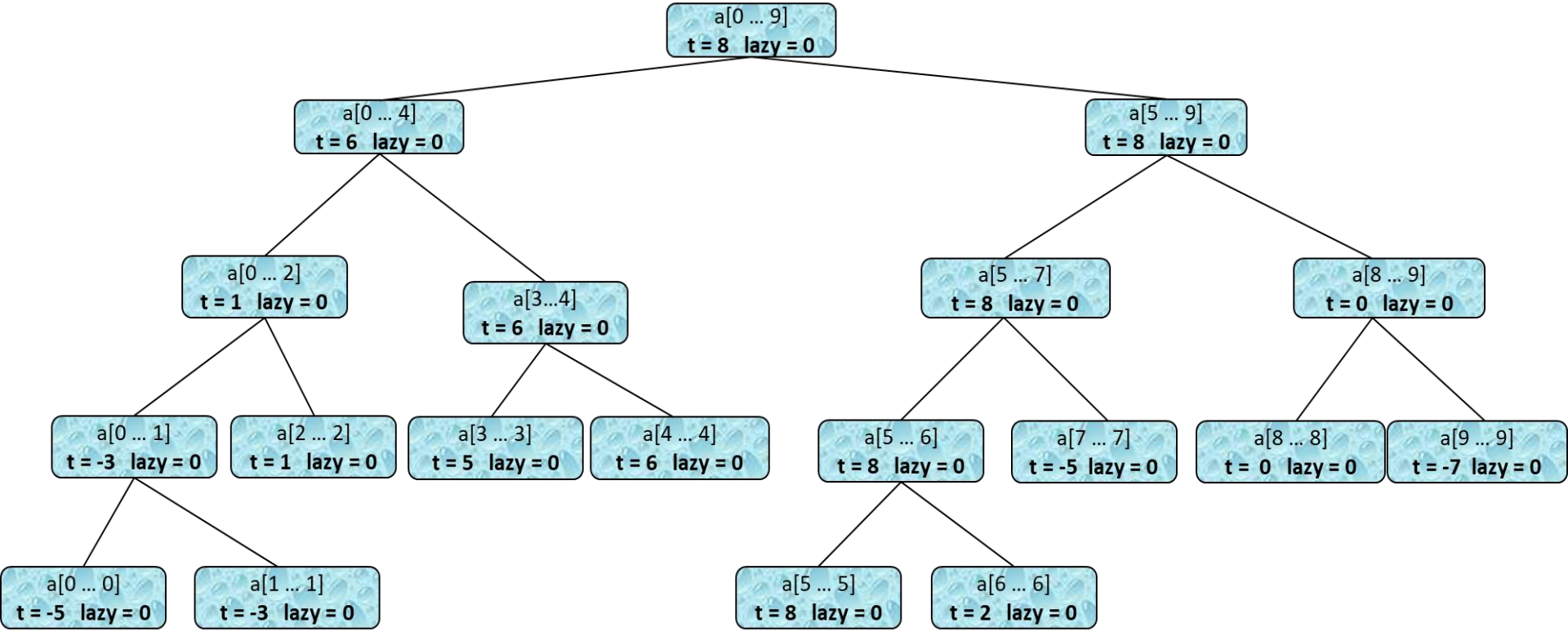
Range Query ,Range Update – Lazy Propagation

Para nuestro problema de queries en rango por el máximo y updates donde se agrega un valor a todo un rango, podemos hacer lo siguiente :

1. **Dividir :** Puedes dividir $update(l\ ,\ r\ ,val)$ en $update(l,p,val), update(p+1\ ,r,val)$
2. **Aplicar :** Si en todo un rango el valor de todos los números aumenta en val entonces el valor del máximo también aumenta en val
3. **Acumular :** Si tenemos 2 updates que tenemos que juntar, en donde en el 1ero se aumentó $val1$ a todo un rango y en el 2do se aumentó $val2$ a todo ese mismo rango, entonces podemos acumularlos y decir que tendremos que aumentar $val1 + val2$ en todo el rango
4. **Reiniciar:** Cuando reiniciamos el $lazy$ de un nodo, simplemente debemos de ponerlo en 0, simbolizando que en ese nodo se debe agregar 0 (es decir no se hace nada).

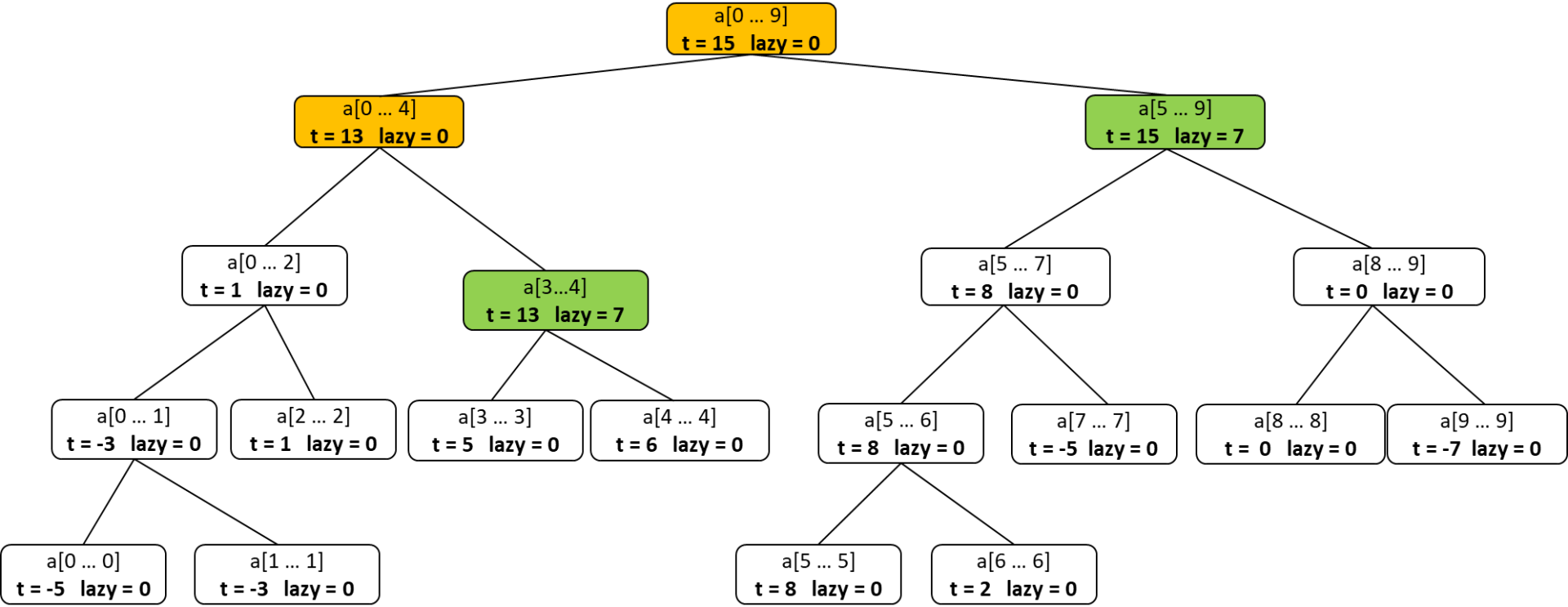
Range Query ,Range Update – Lazy Propagation

Tomemos como ejemplo $A = [-5, -3, 1, 5, 6, 8, 2, -5, 0, -7]$.



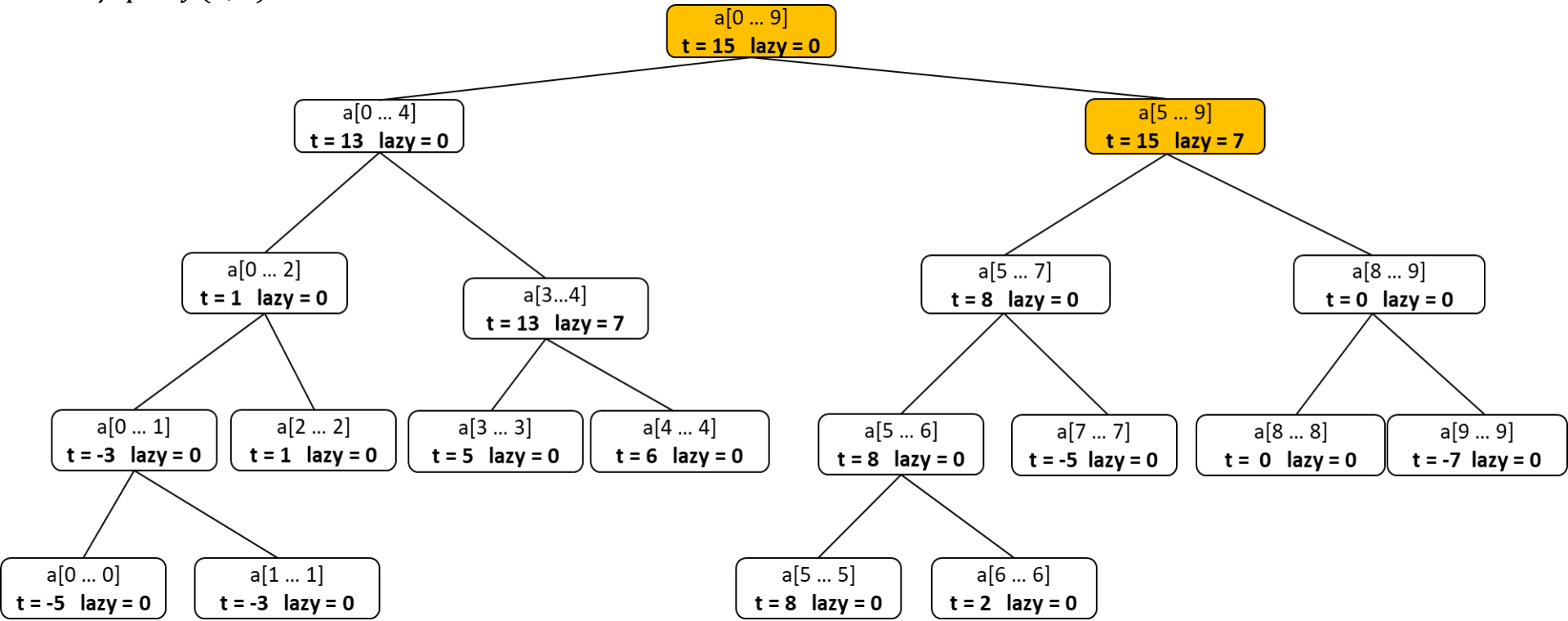
Range Query ,Range Update – Lazy Propagation

1) $update(3\,,9\,,7)$



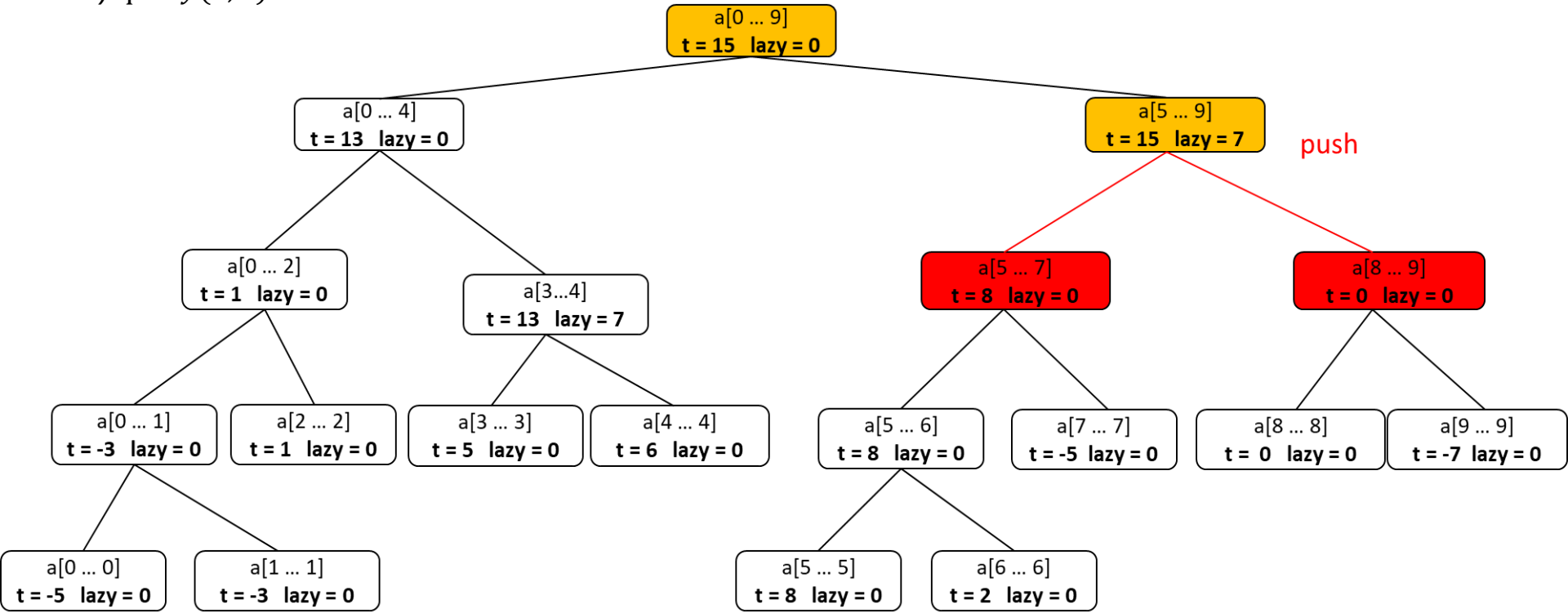
Range Query ,Range Update – Lazy Propagation

2.1) $query(7, 9)$



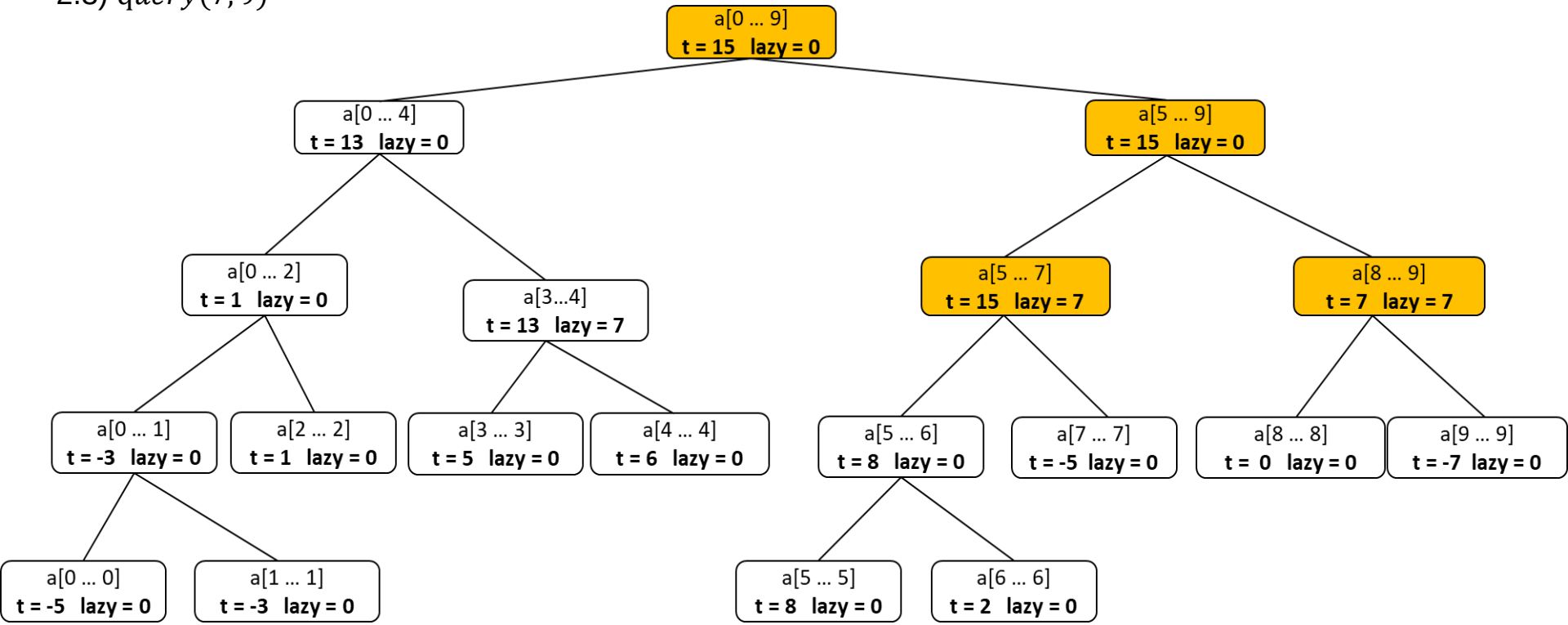
Range Query ,Range Update – Lazy Propagation

2.2) $query(7, 9)$



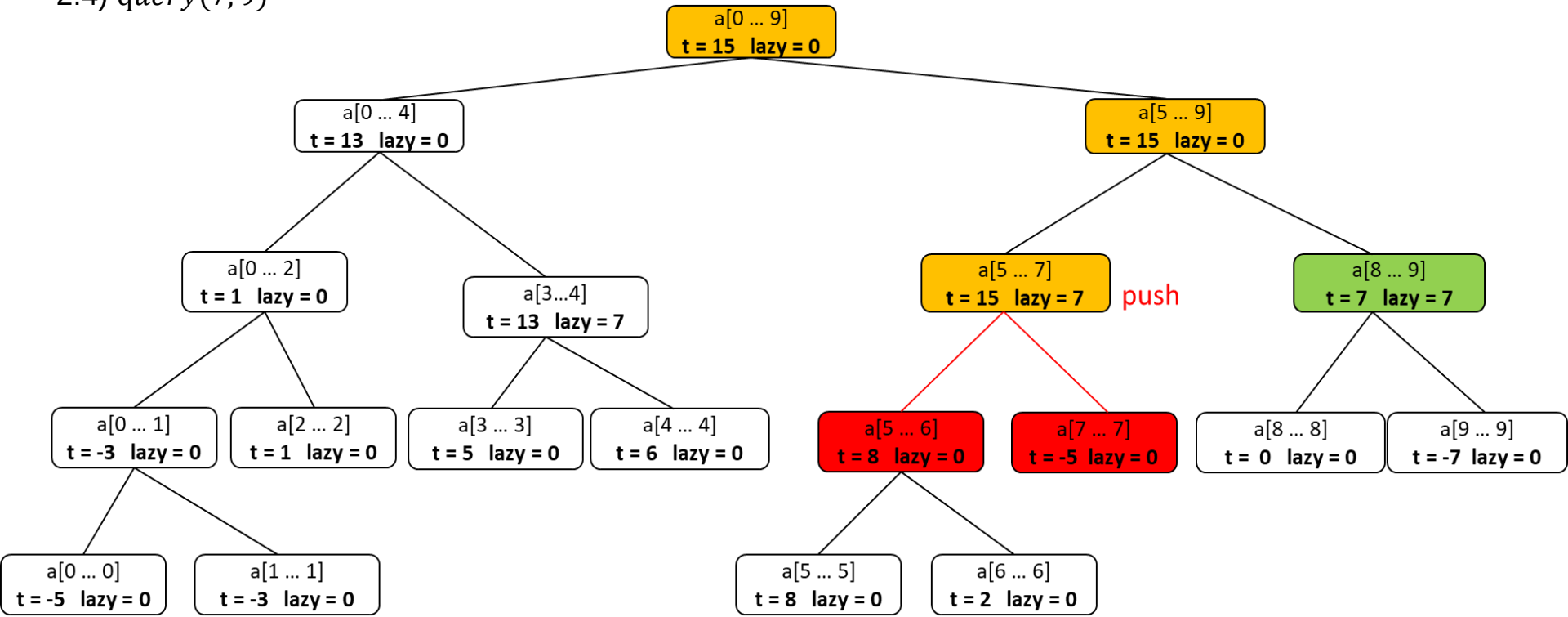
Range Query ,Range Update – Lazy Propagation

2.3) $query(7, 9)$



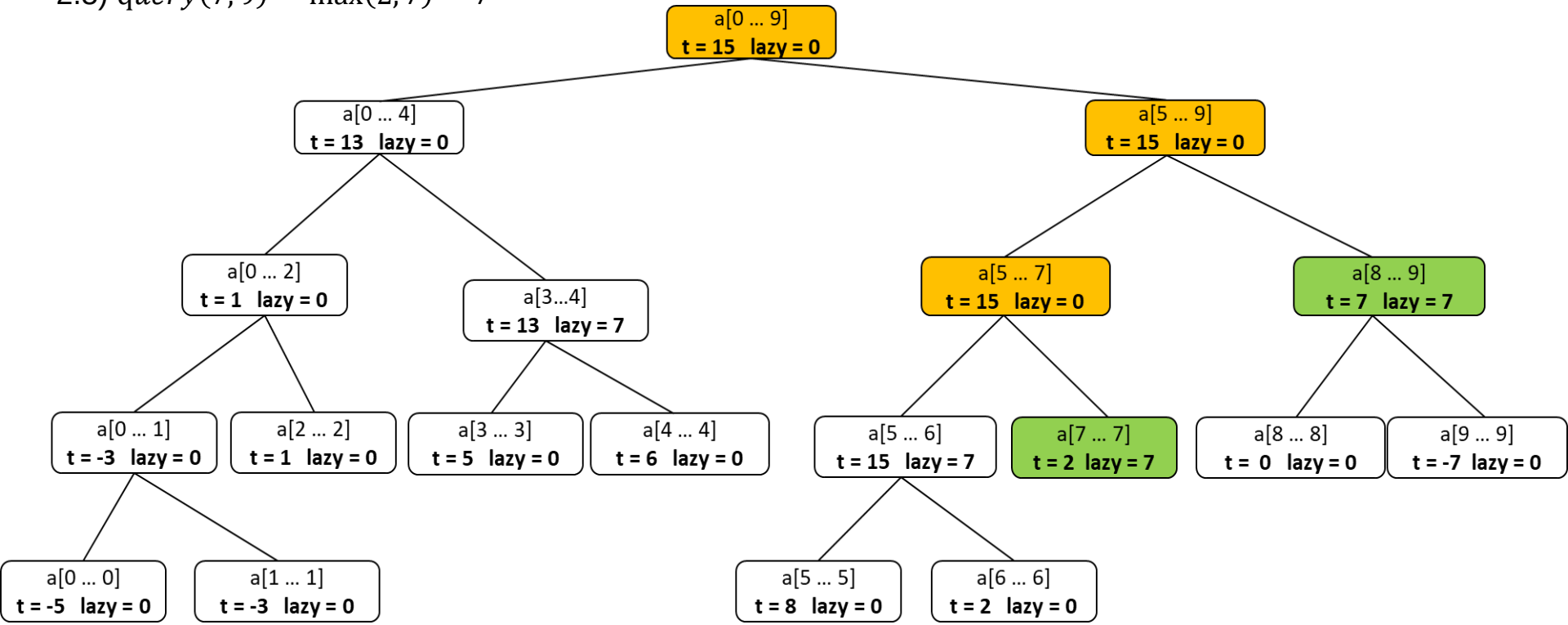
Range Query ,Range Update – Lazy Propagation

2.4) $query(7, 9)$



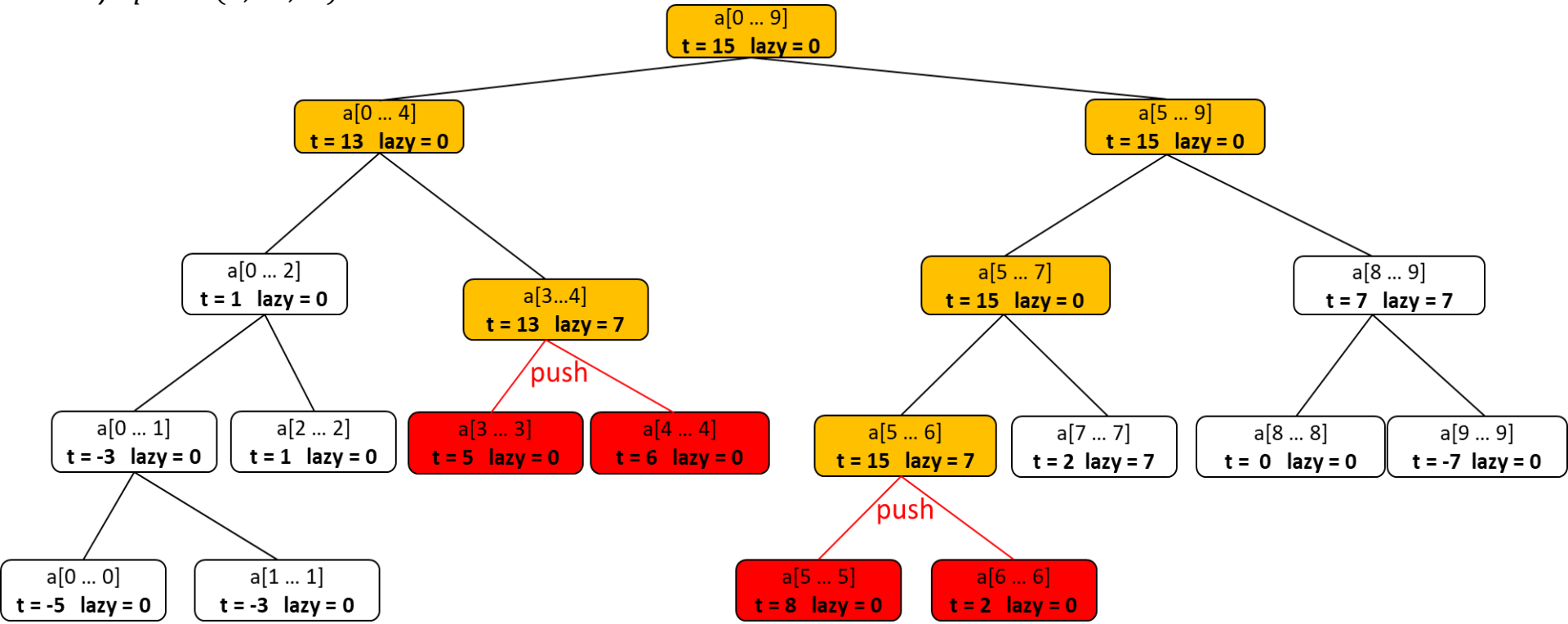
Range Query ,Range Update – Lazy Propagation

2.5) $query(7, 9) = \max(2, 7) = 7$



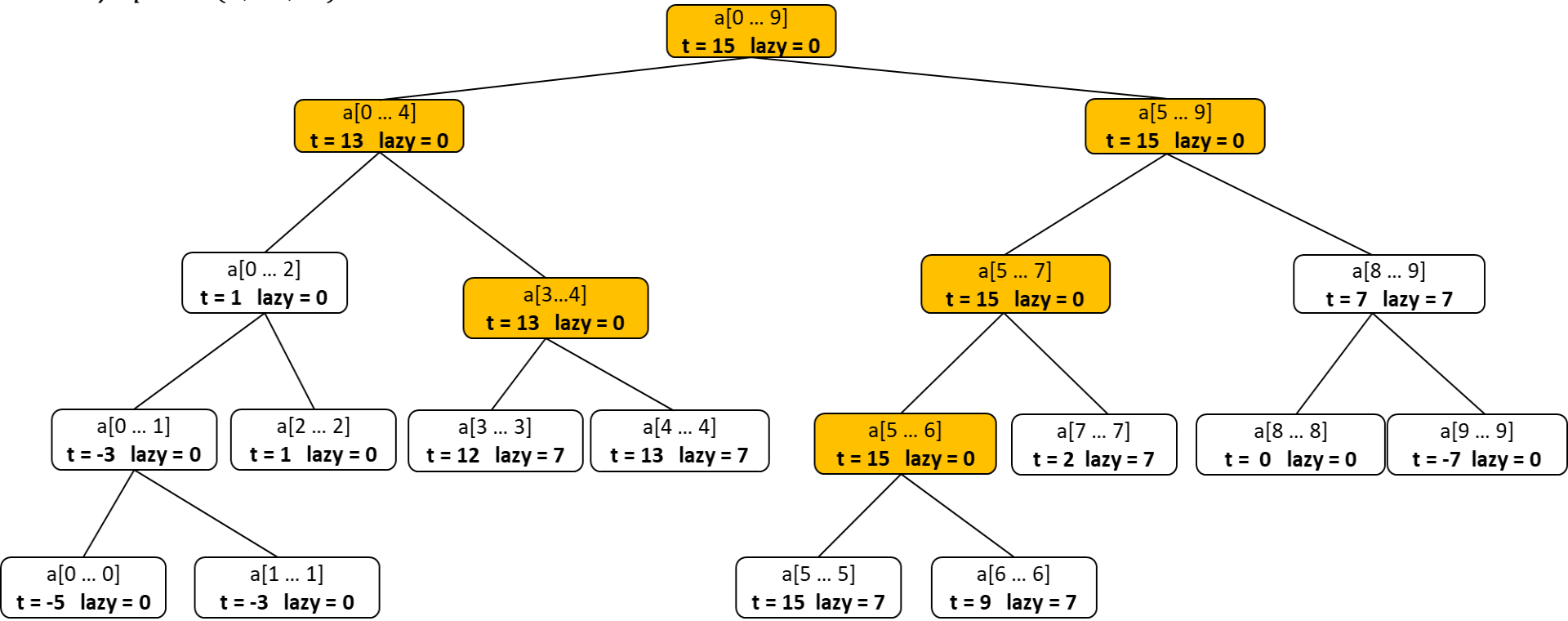
Range Query ,Range Update – Lazy Propagation

3.1) $update(4,\ 5\ ,\ 3)$



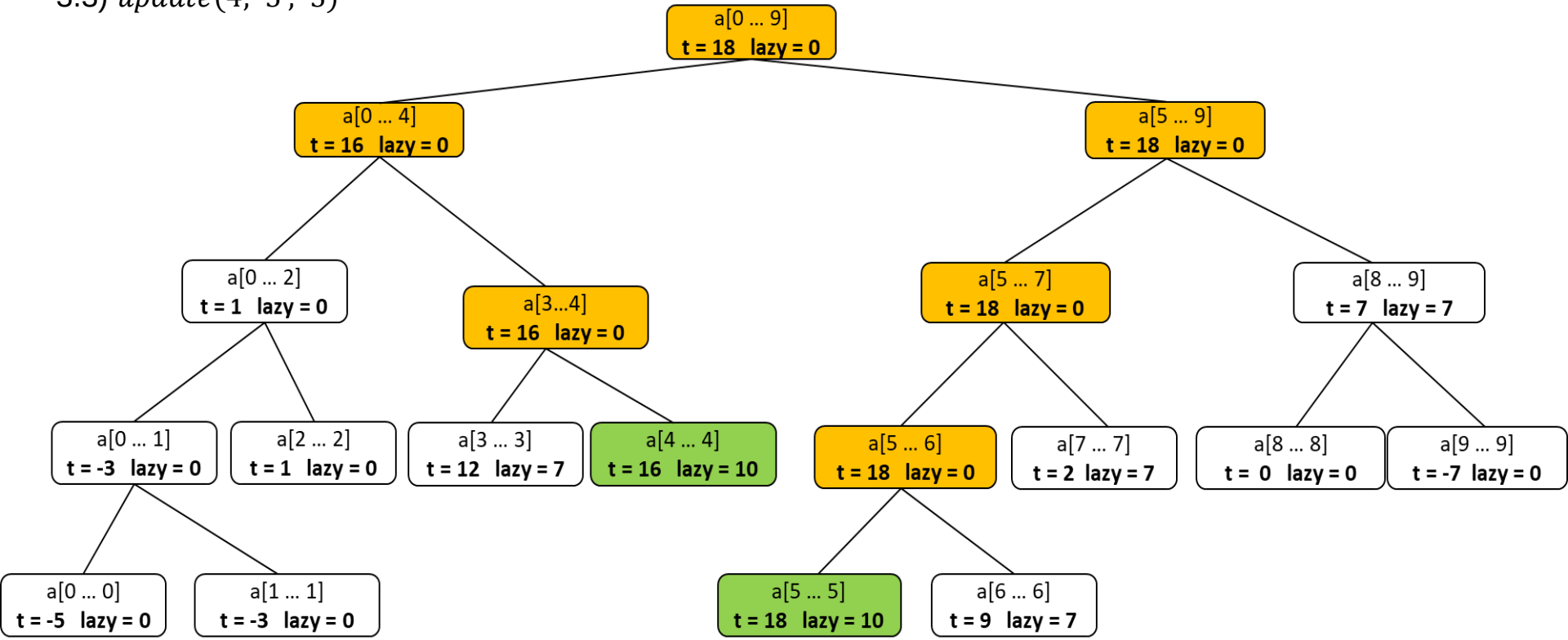
Range Query ,Range Update – Lazy Propagation

3.2) $update(4,\ 5\ ,\ 3)$



Range Query ,Range Update – Lazy Propagation

3.3) $update(4,\ 5\ ,\ 3)$



Range Query ,Range Update – Lazy Propagation

En el código , el build casi no cambia nada, solo hay que inicializar los valores del lazy.

```
Long combine(Long x, Long y){
    return max(x, y);
}


Long t[4 * MX];
Long lazy[4 * MX];

void build(vector<Long> &a, Long id , Long tl , Long tr ) {
    lazy[id] = 0;
    if (tl == tr) {
        t[id] = a[tl];
    }else{
        Long tm = (tl + tr) / 2;
        Long left = 2 * id;
        Long right = 2 * id + 1;
        build(a, left, tl, tm);
        build(a, right, tm + 1, tr);
        t[id] = combine(t[left], t[right]);
    }
}
```

Range Query ,Range Update – Lazy Propagation

Código para el **push**


```
void push(Long id) { //O(1)
    Long left = 2 * id;
    Long right = 2 * id + 1;
    //Apply the lazy value of the node to the children
    t[left] += lazy[id];
    t[right] += lazy[id];

    //aggregate the lazy value of the node to the lazy value of the children
    lazy[left] += lazy[id];
    lazy[right] += lazy[id];

    //restart the lazy value
    lazy[id] = 0;
}
```

Range Query ,Range Update – Lazy Propagation

El código de las query es muy similar a códigos anteriores. Solo no hay que olvidarse de propagar siempre.

```
Long query(Long l, Long r, Long id , Long tl, Long tr ) { //O(logn)
    if (l <= tl && tr <= r) {
        return t[id];
    }
    Long tm = (tl + tr) / 2;
    Long left = 2 * id;
    Long right = 2 * id + 1;
    push(id);
    if(r < tm + 1){
        return query(l , r , left , tl , tm);
    }else if(tm < l){
        return query(l , r, right , tm + 1 , tr);
    } else{
        return combine(query(l, r, left, tl, tm) , query(l, r, right, tm + 1, tr));
    }
}
```

Range Query ,Range Update – Lazy Propagation

En el update, también hay que propagar. Adicionalmente, cuando se llegue a un nodo que esté contenido en el rango del update, tenemos que **aplicar** el update al nodo actual y **“acumular”** el valor de su lazy. Es como hacer el paso 1 y 2 del **lazy propagation** pero por primera vez y sin ningún padre.

```
void update(Long l, Long r, Long add, Long id , Long tl , Long tr) { //O(logn)
    if(tr < l || tl > r){
        return;
    }
    if (l <= tl && tr <= r) {
        //apply the update to the node
        t[id] += add;

        //agregate the lazy value
        lazy[id] += add;
    }else{
        Long tm = (tl + tr) / 2;
        Long left = 2 * id;
        Long right = 2 * id + 1 ;
        push(id);
        update(l, r, add , left, tl, tm);
        update(l, r, add , right, tm + 1, tr);
        t[id] = combine(t[left], t[right]);
    }
}
```

Range Query ,Range Update – Lazy Propagation

Esto se puede aplicar a muchos otros problemas. Por ejemplo:

- $query(l, r) \rightarrow Retornar\ A[l] + A[l+1] + \cdots + A[r]$
- $update(l, r, val) \rightarrow \forall i \in [l, r],\ A[i] \mathrel{+}= val$

Lo único que cambiará será el push y una pequeña parte del update (y obviamente la función combine). Solo tenemos que tener en cuenta los 4 pasos vistos anteriormente.

1. **Dividir :** Puedes dividir $update(l, r, val)$ en $update(l, p, val), update(p+1, r, val)$
2. **Aplicar :** Si en todo un rango el valor de todos los números aumenta en val entonces y ese nodo tiene rango tamaño sz , entonces la suma de ese nodo aumentará en $val * sz$
3. **Acumular :** Si tenemos 2 updates que tenemos que juntar, en donde en el 1ero se aumentó $val1$ a todo un rango y en el 2do se aumentó $val2$ a todo ese mismo rango, entonces podemos acumularlos y decir que tendremos que aumentar $val1 + val2$ en todo el rango
4. **Reiniciar:** Cuando reiniciamos el $lazy$ de un nodo, simplemente debemos de ponerlo en 0.

Range Query ,Range Update – Lazy Propagation

```
void push(Long id, Long tl, Long tr) { //O(1)
    Long left = 2 * id;
    Long right = 2 * id + 1;
    Long tm = (tl + tr) / 2;
    Long szLeft = tm - tl + 1;
    Long szRight = tr - tm;
    //Apply the lazy value of the node to the children
    t[left] += lazy[id] * szLeft;
    t[right] += lazy[id] * szRight;

    //aggregate the lazy value of the node to the lazy value of the children
    lazy[left] += lazy[id];
    lazy[right] += lazy[id];

    //restart the lazy value
    lazy[id] = 0;
}
```

Range Query ,Range Update – Lazy Propagation

```
void update(Long l, Long r, Long add, Long id , Long tl , Long tr) { //O(logn)
    //...
    if (l <= tl && tr <= r) {
        Long sz = tr - tl + 1;
        //apply the update to the node
        t[id] += add * sz;

        //agregate the lazy value
        lazy[id] += add;
    }else{
        //...
    }
}
```

Range Query ,Range Update – Lazy Propagation

Sin embargo no siempre es posible hacerlo eficientemente…

- $query(l, r) \rightarrow Retornar\ A[l] \times A[l+1] \times \cdots \times A[r]$
- $update(l, r, val) \rightarrow \forall i \in [l, r],\ A[i] \mathrel{+}= val$

1. **Dividir :** Puedes dividir $update(l,\ r, val)$ en $update(l, p, val), update(p+1, r, val)$
2. **Aplicar :** ¿Se te ocurre alguna forma para poder aplicar esto en $O(1)$? ¿Cómo variará el producto en rango si todos los elementos aumentan en val ? A simple vista no parece algo simple.

Tenemos que $prod = A[l] \times A[l+1] \times \cdots \times A[r]$

Luego $newProd = (A[l] + val) \times (A[l+1] + val) \times \cdots \times (A[r] + val)$, lo cual se vuelve un polinomio y posiblemente no se pueda o sea difícil hacer con Segment Tree.

Lazy creation

Rangos muy grandes

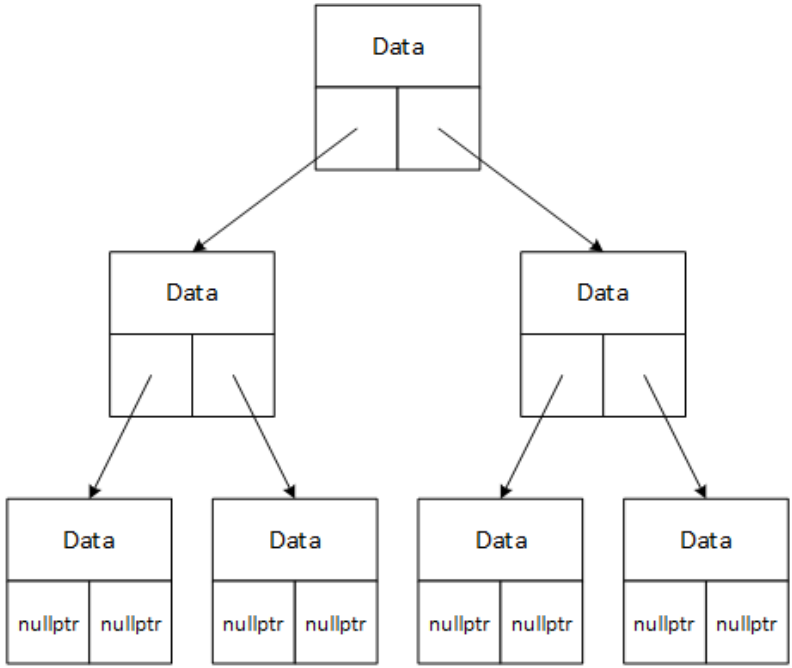
Existen ciertos problemas en donde el rango de las queries / updates es muy grande. Por ejemplo rangos que pueden llegar a ser de tamaño 10^{10}. Usar un arreglo estático excedería la memoria y el tiempo.
Sin embargo , cuando esto sucede, existen ciertas opciones :

- **Comprensión de coordenadas** : Se comprimen todos los rangos y se usa un segment tree normal. Sin embargo esto no siempre es posible ya que la comprensión puede alterar la realización correcta de las queries o updates. También está la posibilidad de que el problema fuerce a que las queries sean contestadas de forma **online**, evitando que se pueda usar esta técnica.

- **Lazy creation con map :** Podemos usar la técnica de solo crear los nodos cuando sean necesarios. Para ello , en vez de un arreglo estático podeos usar un map (o un unordered_map) haciendo que en vez de usar $O(|rango|)$ de memoria, usemos $O((Q+U)\log|rango|)$. Es decir solo se crea $O(\log|rango|)$ de memoria por cada query o update. Sin embargo, la desventaja es que a la complejidad en tiempo se le agrega un factor logarítmico por el map.

Rangos muy grandes

- **Lazy creation con punteros:** Se puede hacer la misma técnica de lazy creation pero en lugar de usar un map, se pueden usar punteros. Lo cual evita el factor logarítmico adicional que daba el map , dando como resultado la misma complejidad de un segment tree normal (solo con un poco de constante adicional por el overhead de los punteros)



Esta es la solución óptima. A esta técnica también se le llama **Implicit Segment Tree** o **Dynamic Segment Tree.**

La idea es crear una estructura que represente al nodo de segment tree y que sea un puntero, y que este a su vez posea 2 punteros que representen a sus 2 hijos (en caso el nodo sea hoja, los punteros hijos serian nulos).

Lazy Creation con Punteros

Supongamos que tenemos el siguiente problema:

- $query(l, r) \rightarrow Retornar\ A[l] + A[l+1] + \cdots + A[r]$
- $update(l, r, val) \rightarrow \forall i \in [l, r],\ A[i] \mathrel{+}= val$
- $l \leq r \leq 10^9$
- $queries\ online$
- $\#\ queries + \#\ updates \leq 5 \times 10^5$

Ya lo hemos resuelto antes, pero le hemos agregado la condición de que los rangos pueden llegar a números muy lejanos y que las queries son online (para evitar la compresión de coordenadas). El número de queries también es demasiado grande como para usar map. La única vía es usar punteros.

Lazy Creation con Punteros

Cuando hagamos el código, lo primero que haremos es crear nuestra estructura $Node$ que debe contener toda la información necesaria del nodo del segment tree, incluyendo la respuesta a la query, los lazys , los dos nodos hijos, etc. Adicionalmente, un constructor también será bastante útil.

```
struct Node{
    Long sum;
    Long lazy;
    Node *left;
    Node *right;

    Node(){
        sum = lazy = 0;
        left = right = nullptr;
    }
};
```

Lazy Creation con Punteros

Como los rangos son muy grandes, todos los cambios se harán por medio de updates y NO será necesario una función $build$.

En nuestro segment tree, tenemos que definir el node raíz desde el cual empezaremos cada update y query. Podemos crear nuestra estructura Segment Tree de la siguiente forma :

```
struct SegmentTree {
    Node *root;
    Long maxN;

    SegmentTree(){}

    SegmentTree(Long n){
        root = new Node();
        maxN = n;
    }

    void push(/.../){
        //...
    }

    Long query(/.../){
        //...
    }

    void update(/.../){
        //...
    }
};
```

Lazy Creation con Punteros

En la función $push$ agregaremos la funcionalidad de crear los nodos hijos en caso sean nulos.

```
void push(Node *node, Long tl, Long tr) { //O(1)
    //Create children if they are NULL
    if(!node->left){
        node->left = new Node();
        node->right = new Node();
    }
    Long tm = (tl + tr) / 2;
    Long szLeft = tm - tl + 1;
    Long szRight = tr - tm;
    //Apply the lazy value of the node to the children
    node->left->sum += node->lazy * szLeft;
    node->right->sum += node->lazy * szRight;

    //aggregate the lazy value of the node to the lazy value of the children
    node->left->lazy  += node->lazy;
    node->right->lazy += node->lazy;

    //restart the lazy value
    node->lazy = 0;
}
```

Lazy Creation con Punteros

```
Long query(Long l, Long r, Node *node ,  Long tl, Long tr ) { //O(logn)
    if (l <= tl && tr <= r) {
        return node->sum;
    }
    Long tm = (tl + tr) / 2;
    node->push(tl , tr);
    if(r < tm + 1){
        return query(l , r, node->left  , tl , tm);
    }else if(tm < l){
        return query( l , r , node->right , tm + 1 , tr);
    } else{
        return combine(query(l, r, node->left, tl, tm) , query(l, r, node->right, tm + 1, tr));
    }
}

Long query(Long l , Long r) {
    assert(maxN > 0);
    return query(l , r  ,root ,  0 , maxN - 1);
}
```

Lazy Creation con Punteros

```
void update(Long l, Long r, Long val, Node *node , Long tl , Long tr) { //O(logn)
    if(tr < l || tl > r){
        return;
    }
    if (l <= tl && tr <= r) {
        Long sz = tr - tl + 1;
        node->sum += val * sz;
        node->lazy += val;
    }else{
        Long tm = (tl + tr) / 2;
        push(node, tl , tr);
        update(l, r, val , node->left, tl, tm);
        update(l, r, val , node->right, tm + 1, tr);
        node->sum = combine(node->left->sum, node->right->sum);
    }
}

void update(Long l , Long r, Long val) {
    assert(maxN > 0);
    update(l, r , val , root , 0 , maxN - 1);
}
```

Referencias

- cp-algorithms: https://cp-algorithms.com/data_structures/segment_tree.html
- Comentarios del blog : https://codeforces.com/blog/entry/44478