Splay tree + Link-cut tree

Matías Hunicken[1]

[1] Universidad Nacional de Córdoba - FaMAF

Training Camp 2020

Contenidos

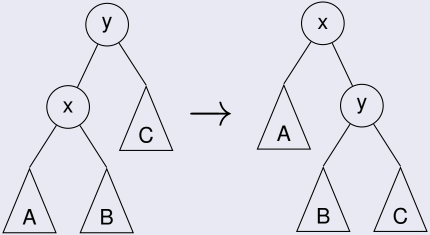
Contenidos

Splay tree - intro

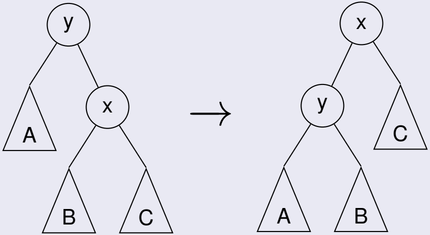
- Splay tree es un árbol binario de búsqueda eficiente.
- La operación base de splay-tree es `splay(x)`, donde `x` es un nodo.
- `splay(x)` lleva el nodo x a la raíz del árbol, manteniendo el orden de los nodos.
- Splay está basada en rotaciones simples, que suben al nodo un nivel.

Rotaciones (`rotate(x)`)

Derecha



Izquierda

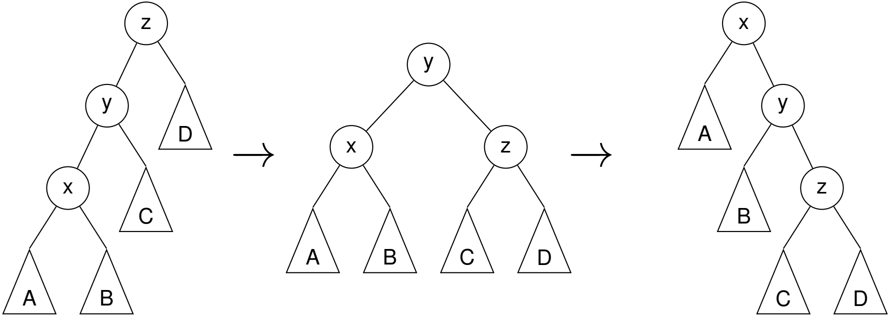


Splay

`splay(x)` levanta a `x` de abajo hacia arriba, de la siguiente forma:

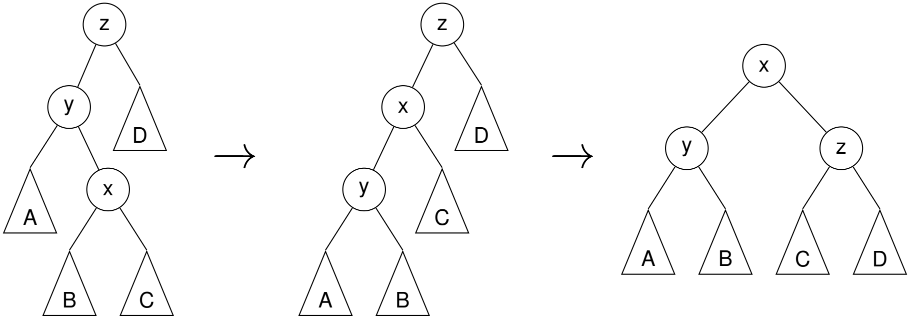
- Mientras `x` no sea la raíz:
  - Sea `p` el padre de `x`.
  - Si `p` es la raíz, rotar `x`.
  - Si no:
    - Si `p` y `x` son ambos hijos izquierdos o ambos hijos derechos de sus respectivos padres (*zig-zig*), rotar `p` y luego rotar `x`.
    - Caso contrario (*zig-zag*), rotar `x` dos veces.

Splay - zig-zig



(Levanto al padre de x, luego a x)

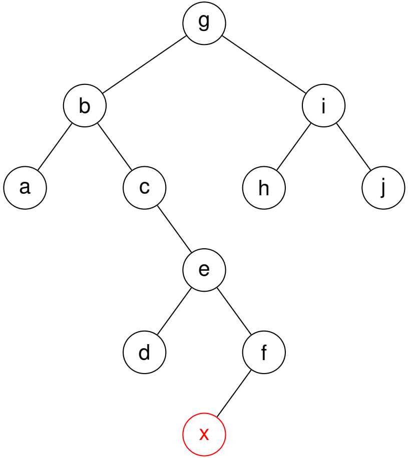
Splay - zig-zag



(Levanto a x dos veces)

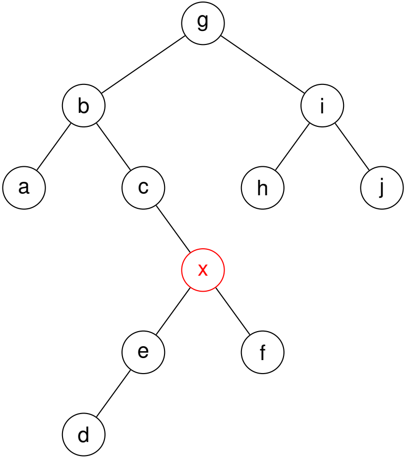
Ejemplo

`splay(x)`



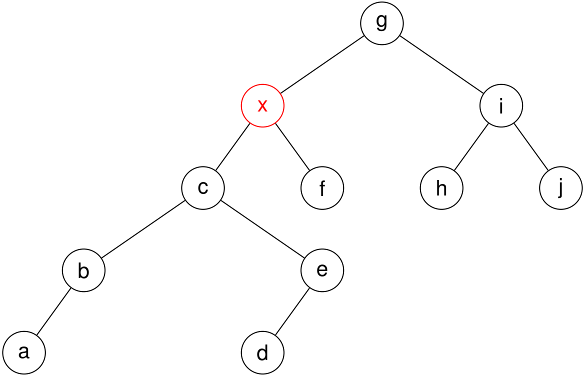
Ejemplo

`splay(x)`



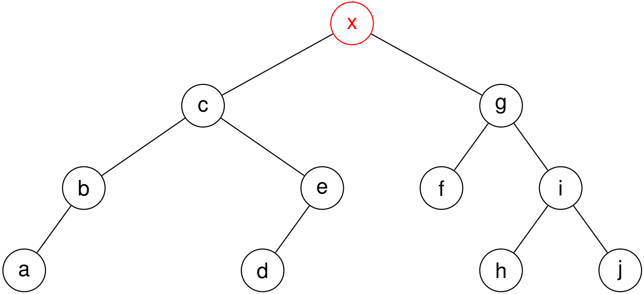
Ejemplo

`splay(x)`



Ejemplo

`splay(x)`



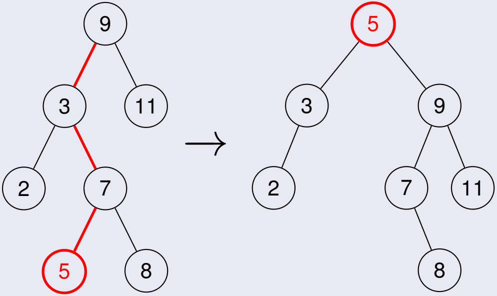
Splay - Complejidad

- El peor caso de splay es $O(n)$ (donde n es el tamaño del árbol).
- Sin embargo splay es $O(log(n))$ amortizado (o sea, el costo de m splays es $O(m \cdot log(n))$).
- Usando splay, se pueden implementar las típicas operaciones de árbol binario de búsqueda.

Splay tree - Búsqueda

La búsqueda se hace como en cualquier árbol binario de búsqueda, pero se debe hacer splay del último nodo visitado para garantizar buena complejidad amortizada.

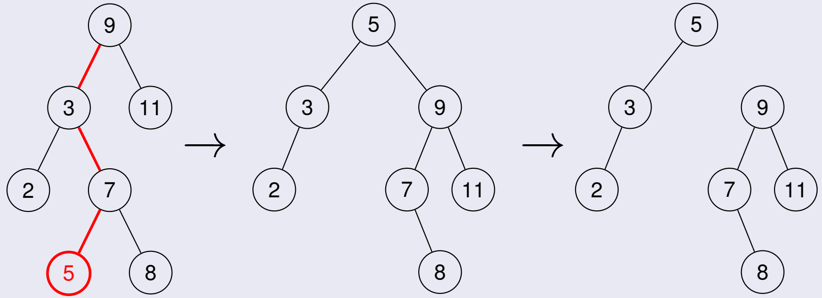
`search(5)`
(también `search(4)` o `search(6)`)



Splay tree - Split

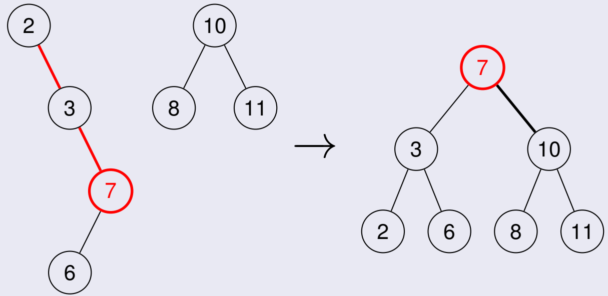
- `split(v)` consiste en dividir el árbol en dos árboles, tal que uno contiene todos los valores menores que `v` y el otro los mayores o iguales.
- Se puede implementar haciendo `search(v)` y borrando la arista de la raíz a su hijo izquierdo o derecho dependiendo de si el valor en la raíz es $\geq v$ o $< v$ respectivamente.

`split(6)`



Splay tree - Merge

- Merge es la inversa de split: Toma dos árboles, tal que los valores del primero son menores que los del segundo, y los combina en un sólo árbol.
- Se puede implementando haciendo splay del nodo más grande del primero, y asignándole como hijo derecho la raíz del segundo.

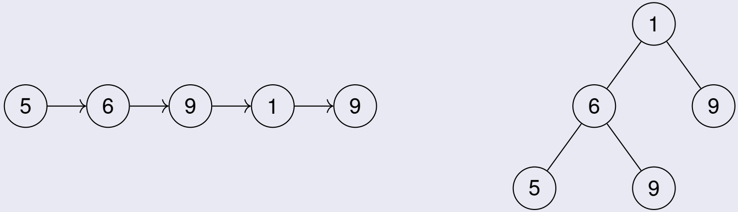


Splay tree - Inserción y borrado

- Inserción y borrado se pueden implementar sencillamente con split y merge.
- Insertar v:
  - Hacer `split(v)` en el árbol, obteniendo dos árboles T_1 y T_2.
  - Si v no es la raíz de T_2, crear un nuevo árbol haciendo merge de T_1, un árbol con un sólo nodo de valor v, y T_2.
  - Caso contrario, hacer merge de T_1 y T_2.
- Borrar v:
  - Hacer `split(v)` en el árbol, obteniendo dos árboles T_1 y T_2.
  - Si v es la raíz de T_2, crear un nuevo árbol haciendo merge de T_1 y el hijo derecho de T_2.
  - Caso contrario, hacer merge de T_1 y T_2.

Splay tree - Listas

- Splay tree se puede usar para representar listas, en lugar de conjuntos (un árbol representa la lista que resulta al recorrer el árbol *in-order*).



- En este caso:
  - Búsqueda se puede ver como llevar un elemento de la lista a la raíz del árbol.
  - Split se puede ver como: Dado un elemento en una lista, partir la lista en los elementos anteriores y posteriores a ese elemento.
  - Merge se puede ver como: Dadas dos listas, concatenarlas.

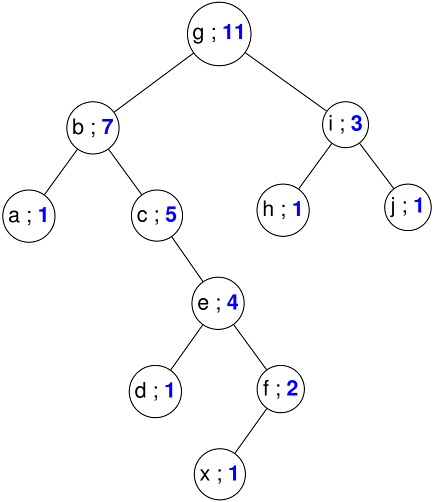
Splay tree - Cantidad de descendientes

Una extensión útil de splay tree es mantener para cada nodo la cantidad de “descendientes”:

- Cuando rotamos un nodo, actualizamos los valores de todos los nodos en el camino de la raíz al nodo.
- Cuando agregamos o quitamos una arista de la raíz a un nodo, actualizamos el valor de la raíz.

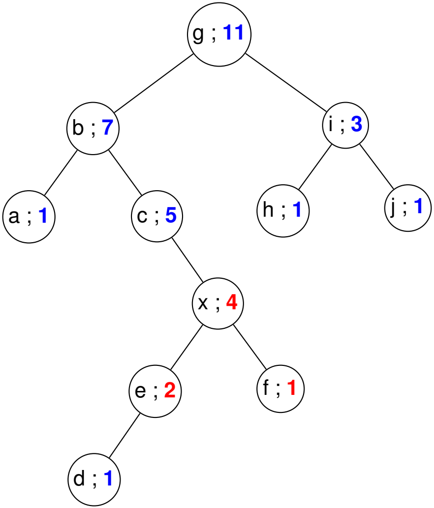
Cantidad de descendientes - ejemplo

`splay(x)`



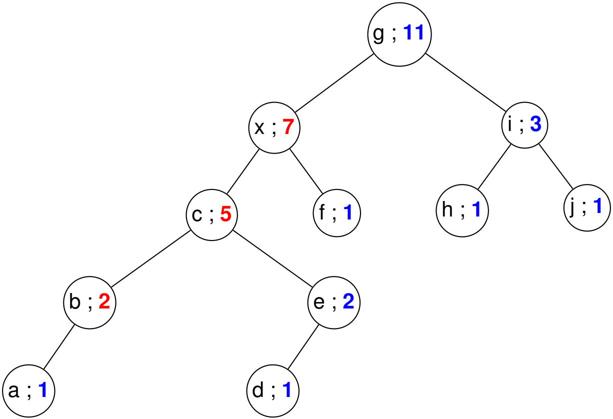
Cantidad de descendientes - ejemplo

`splay(x)`



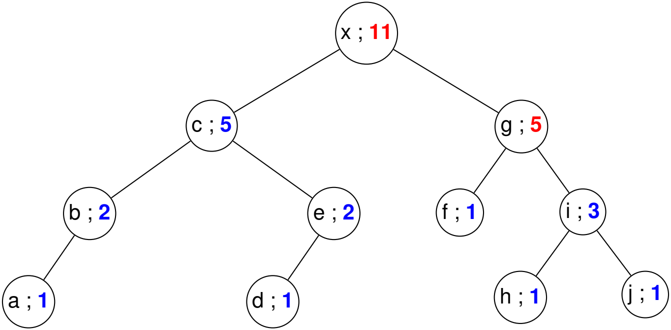
Cantidad de descendientes - ejemplo

`splay(x)`



Cantidad de descendientes - ejemplo

`splay(x)`



Splay tree - Buscar i-ésimo

Usando la cantidad de descendientes de cada subárbol, se puede implementar la operación de buscar el *i*-ésimo nodo en un árbol T de forma recursiva:

- Dado un árbol T y un índice $i \in [0, |T|)$.
- Si $i = |left(T)|$, retornar T.
- Si $i < |left(T)|$, buscar el i-ésimo elemento en $left(T)$.
- Si $i > |left(T)|$, buscar el $(i - |left(T)| - 1)$-ésimo elemento en $right(T)$.

(Nota: cada vez que hacemos esta operación debemos hacer *splay* del nodo resultado).

Splay tree - Agregación

- Así como mantenemos la cantidad de nodos de un subárbol, también se puede mantener el resultado de una operación asociativa realizada en los elementos del subárbol.
- De esta manera se puede utilizar splay tree como un segment tree, donde adicionalmente soportamos las operaciones de partir y concatenar.
  - Para hacer update de un nodo, primero debemos hacer splay del nodo, para garantizar complejidad amortizada.
  - Para hacer query de un rango, hacemos dos splits para quedarnos con ese rango y devolvemos el valor guardado en la raíz.

Splay tree - Lazy propagation

- Splay tree permite realizar ciertas operaciones sobre todos los valores mediante *lazy propagation*.
- Esta técnica consiste en guardar la operación a realizarse en la raíz del árbol inicialmente, y sólo “propagarla” a los hijos en caso de ser necesario, es decir:
  - Cuando hacemos *splay* de un nodo, propagamos los valores de los ancestros del nodo.
  - Cuando agregamos o quitamos una arista de la raíz, propagamos el valor de la raíz antes.

Ejemplo: Mantener mínimo del árbol y sumar un valor a todos los valores del árbol:

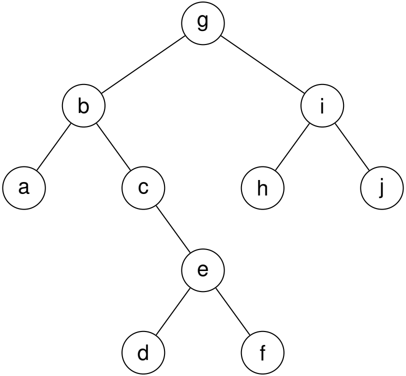
- Mantenemos para cada nodo un valor `add`, que es cuánto falta sumar a todos los descendientes del nodo.
- Cuando propagamos, sumamos `add` al valor del mínimo, sumamos `add` a los valores de `add` de los hijos y seteamos `add` a 0.

Splay tree - `reverse`

- Hay una operación lazy que es de especial interés, que es “dar vuelta” la lista representada por el árbol.
  - En este caso, guardamos para cada nodo un booleano `rev` que representa si debemos dar vuelta el intervalo correspondiente.
  - Cuando propagamos, si `rev` es falso no hacemos nada. Si es verdadero, swapeamos los hijos izquierdo y derecho del nodo e invertimos los valores de `rev` de ambos.

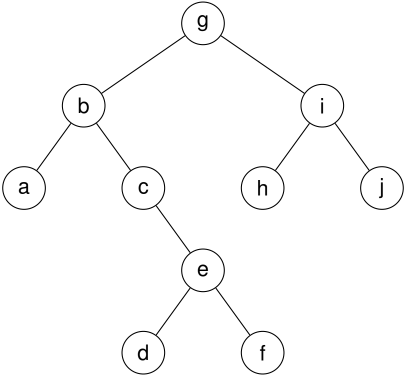
Lazy - ejemplo

```
reverse()
splay(d)
reverse()
splay(g)
```



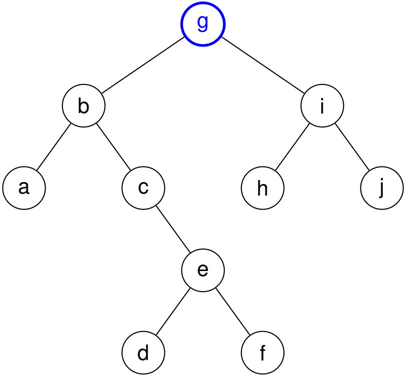
Lazy - ejemplo

`reverse()`
`splay(d)`
`reverse()`
`splay(g)`



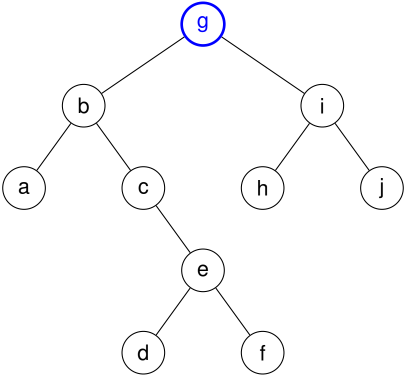
Lazy - ejemplo

`reverse()`
`splay(d)`
`reverse()`
`splay(g)`



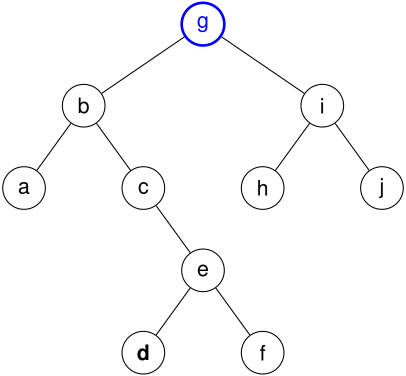
Lazy - ejemplo

```
reverse()
splay(d)
reverse()
splay(g)
```



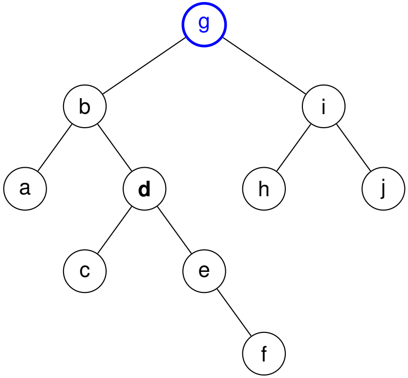
Lazy - ejemplo

```
reverse()
```
`splay(d)`
```
reverse()
splay(g)
```



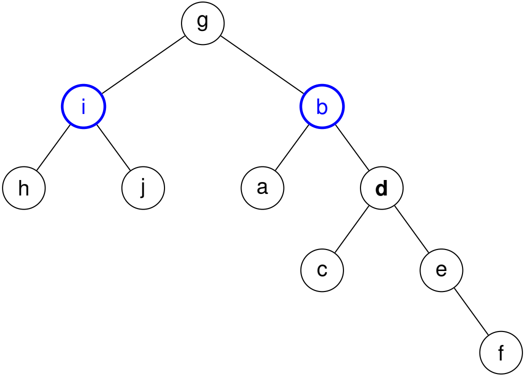
Lazy - ejemplo

```
reverse()
```
`splay(d)`
```
reverse()
splay(g)
```



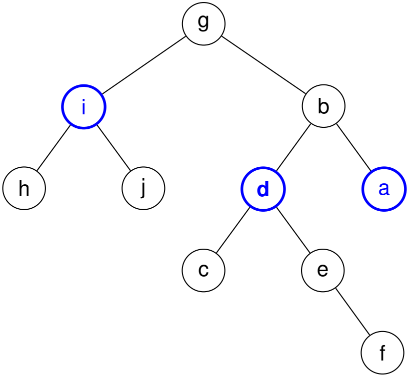
Lazy - ejemplo

```
reverse()
```
`splay(d)`
```
reverse()
splay(g)
```



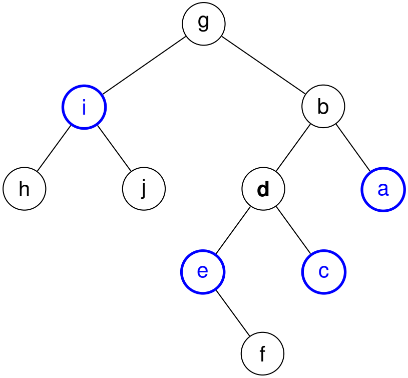
Lazy - ejemplo

```
reverse()
splay(d)
reverse()
splay(g)
```



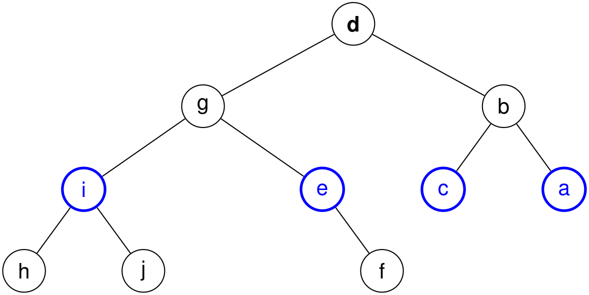
Lazy - ejemplo

```
reverse()
splay(d)
reverse()
splay(g)
```



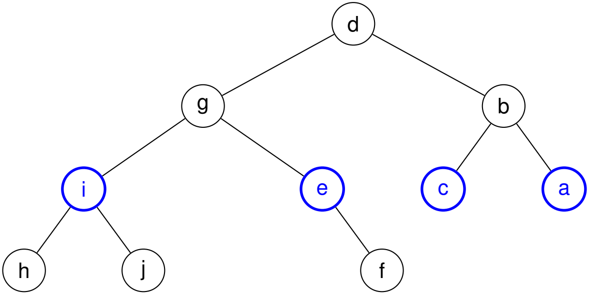
Lazy - ejemplo

```
reverse()
```
`splay(d)`
```
reverse()
splay(g)
```

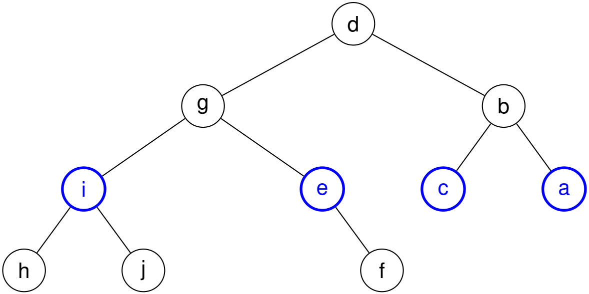


Lazy - ejemplo

```
reverse()
splay(d)
reverse()
splay(g)
```

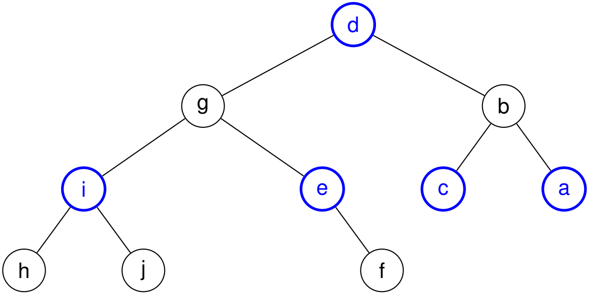


```
reverse()
splay(d)
```
`reverse()`
```
splay(g)
```



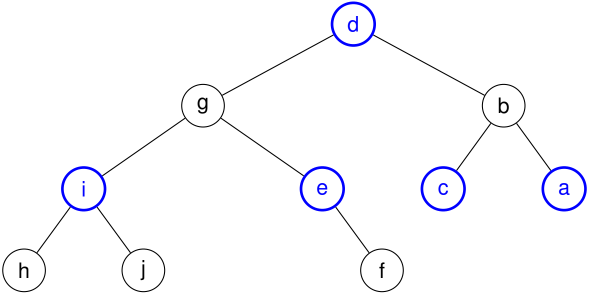
Lazy - ejemplo

```
reverse()
splay(d)
```
`reverse()`
```
splay(g)
```



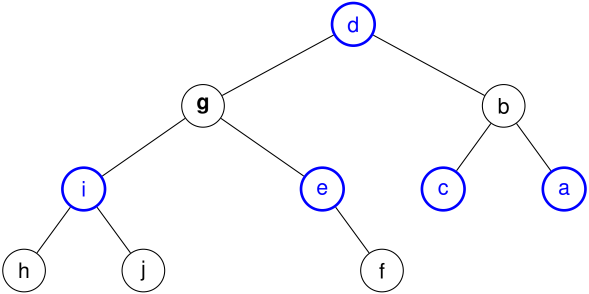
Lazy - ejemplo

```
reverse()
splay(d)
reverse()
splay(g)
```



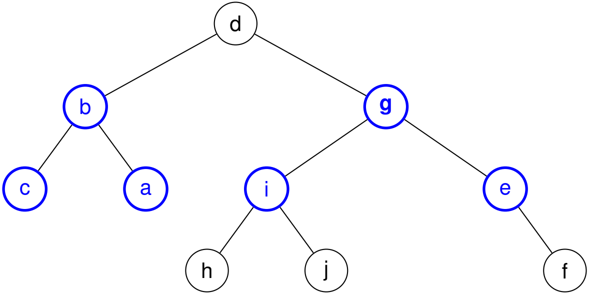
Lazy - ejemplo

```
reverse()
splay(d)
reverse()
```
`splay(g)`



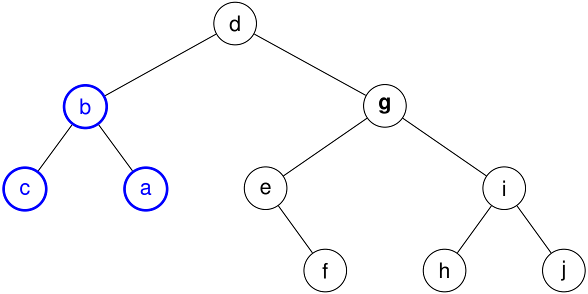
Lazy - ejemplo

```
reverse()
splay(d)
reverse()
```
`splay(g)`

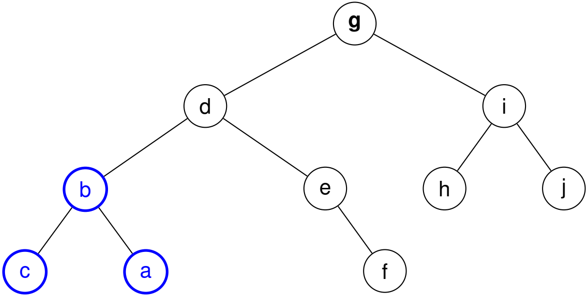


Lazy - ejemplo

```
reverse()
splay(d)
reverse()
```
`splay(g)`

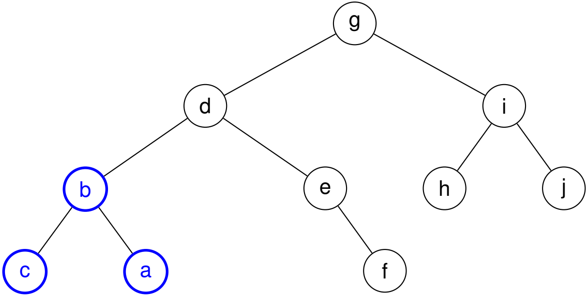


```
reverse()
splay(d)
reverse()
```
`splay(g)`



Lazy - ejemplo

```
reverse()
splay(d)
reverse()
splay(g)
```



Splay tree - conclusión

- *Splay tree* puede ser utilizado como un árbol binario de búsqueda extendido, o como un segment tree con operaciones de cortar, concatenar y dar vuelta.
- Hay otros árboles binarios de búsqueda que permiten las mismas operaciones, como *Treap*, que es más sencillo de implementar y soporta persistencia.
- Sin embargo, *Splay tree* funciona mejor que otros como base para implementar *Link-cut tree*.

Contenidos

Link-cut tree - Definición

Link-cut tree es una estructura que mantiene un bosque de árboles con raíz. Soporta las siguientes operaciones en $O(log(n))$ amortizado:

- `link(X,Y)`: Conectar los nodos X e Y, haciendo a X hijo de Y (X e Y deben estar en distintos árboles y X debe ser raíz de su árbol).
- `cut(X)`: Desconectar a X de su padre.
- `makeRoot(X)`: Hacer a X la raíz de su árbol.
- `getRoot(X)`: Devolver la raíz del árbol de X.
- `lca(X,Y)`: Dado que X e Y están en el mismo árbol, devolver el ancestro común más bajo de X e Y.
- `lift(X,k)`: Devolver el k-ésimo ancestro de X (`lift(X,1)` es el padre de X, `lift(X,2)` el abuelo, etc.).

Link-cut tree - Definición (cont.)

Además, si cada nodo tiene asociado un valor, se puede soportar:

- `aggregate(X,Y)`: Devolver el resultado de una operación asociativa en los nodos del camino de X a Y.
- `update(X,v)`: Cambiar a v el valor asociado a X.
- Operaciones lazy sobre los valores de un camino (del mismo tipo que las operaciones de segment-tree).

Link-cut tree - “Acceder” e hijos preferidos

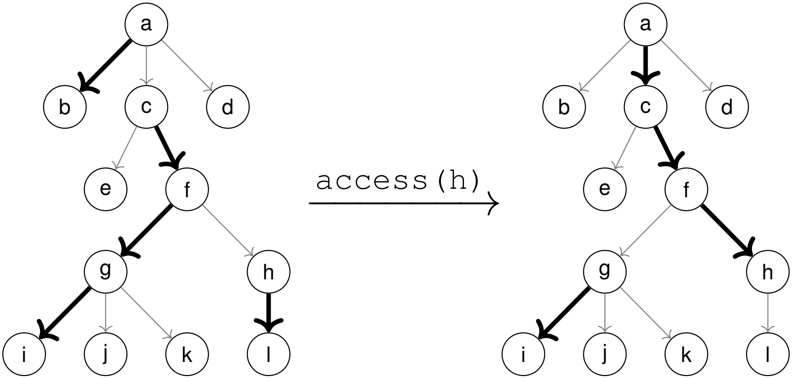
- La operación básica de *Link-cut tree* es “acceder” a un nodo, lo cuál se hace cada vez que hacemos una operación sobre ese nodo.
- Llamamos a esta operación `access(x)`.

Hijo preferido - definición

Para cada nodo x, definimos como “hijo preferido” de x, al hijo de x en cuyo subárbol fue el último acceso entre los descendientes de x (o ninguno si el último acceso en el subárbol de x fue a x).

Asimismo, llamamos “arista preferida” a la arista de un nodo a su hijo preferido.

Acceder e hijos preferidos - ejemplo



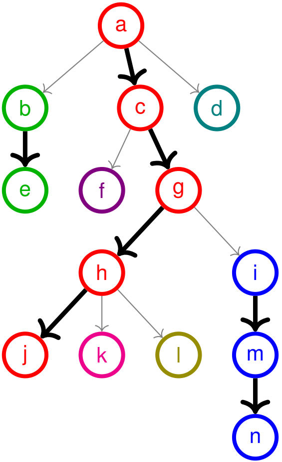
Link-cut tree - Representación

- Las aristas preferidas particionan los nodos en caminos descendientes, que llamamos “caminos preferidos”.
- La operación `access(x)` convierte al camino entre x y la raíz en un camino preferido.

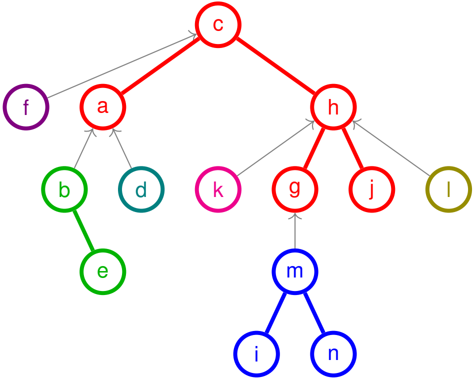
- La idea de *Link-cut tree* es representar cada camino preferido con un splay tree.
- Además, mantenemos en la raíz de cada splay tree un puntero al padre del nodo más alto del camino preferido (llamamos a ese nodo “padre del camino”).

Link-cut tree - Representación (ejemplo)

Árbol representado:



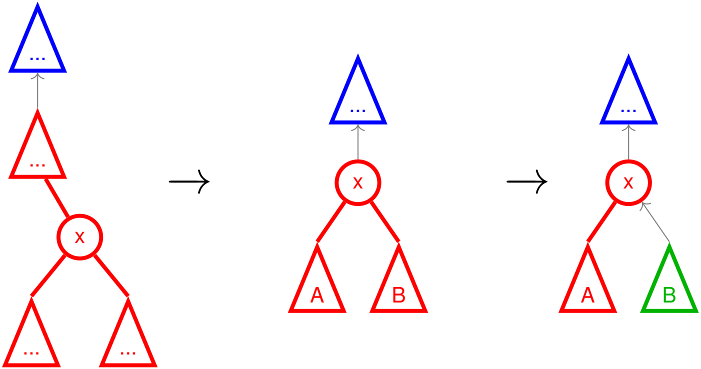
Árbol auxiliar (lo que guardamos):



Link-cut tree - Access

Con esta representación, podemos implementar `access(x)` de la siguiente manera:

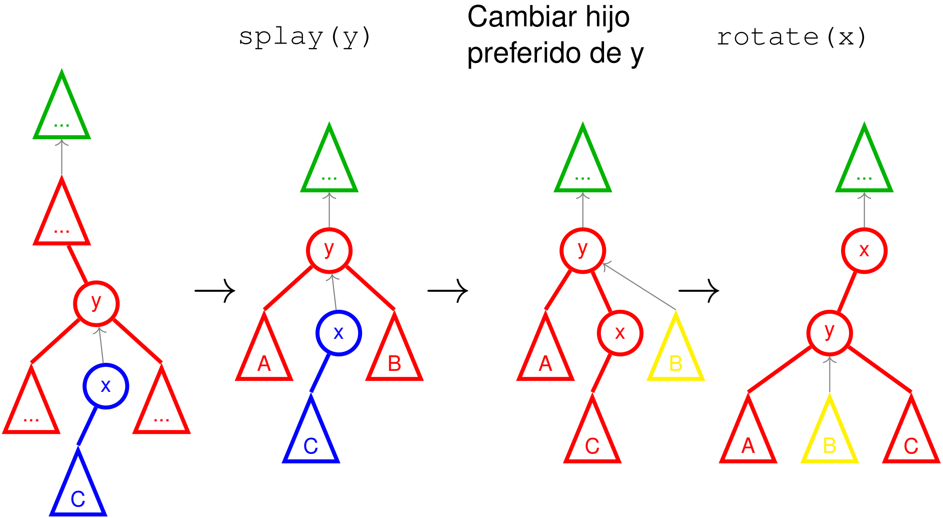
- Para hacer que x no tenga más hijo preferido, hacemos `splay(x)` y desconectamos a x de su hijo derecho en el splay tree, asignando x al "puntero a padre del camino" de su hijo derecho.



Link-cut tree - Access (cont.)

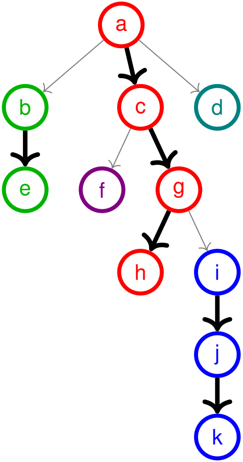
- (Mantenemos el invariante de que x es raíz de su splay tree).
- Iteramos mientras x tenga un puntero a “padre del camino de x” (lo llamamos y).
  - Hacemos `splay(y)`, llevando a y a la raíz de su splay tree.
  - Para hacer que el hijo preferido de y deje de serlo, desconectamos a y de su hijo derecho en el splay tree, en su lugar asignando y al “puntero a padre de camino” del hijo derecho de y.
  - Concatenamos los splay trees de x e y, haciendo hijo derecho de y a x.
  - Rotamos x para mantener el invariante.

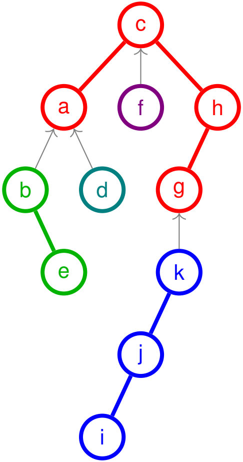
Link-cut tree - Access (cont.)



Link-cut tree - Access (ejemplo)

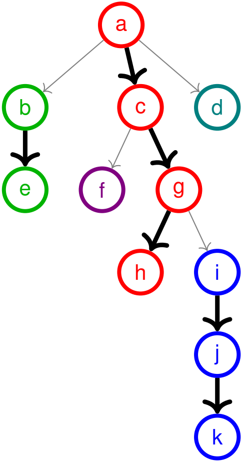
`access(j)`

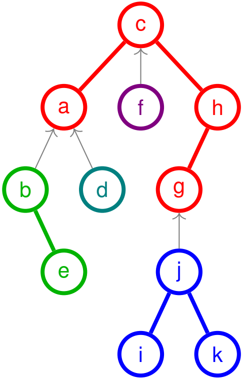




Link-cut tree - Access (ejemplo)

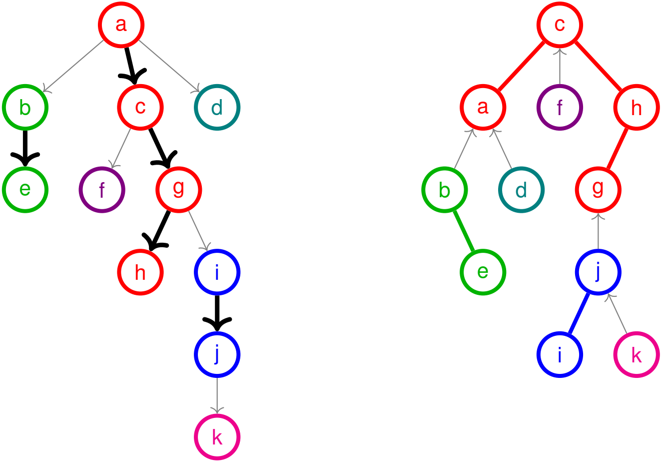
`access(j)`





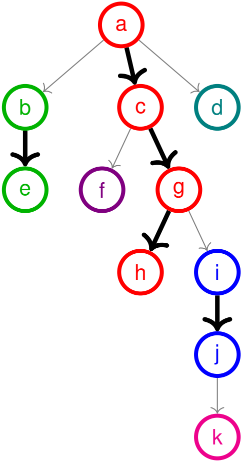
Link-cut tree - Access (ejemplo)

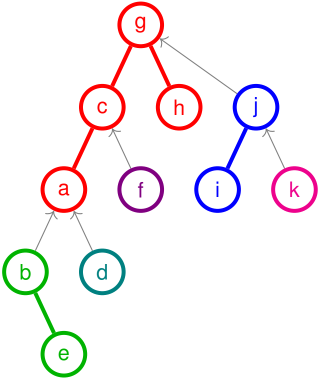
`access(j)`



Link-cut tree - Access (ejemplo)

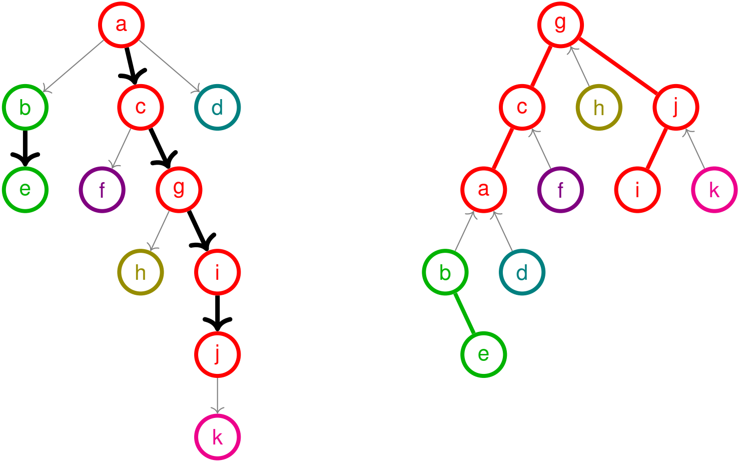
`access(j)`



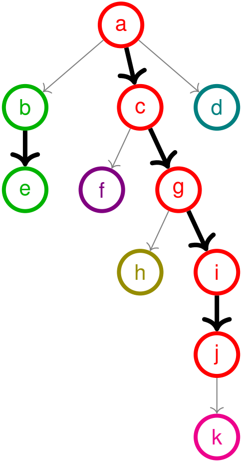


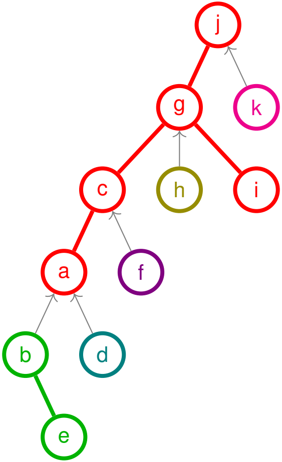
Link-cut tree - Access (ejemplo)

```
access(j)
```



`access(j)`





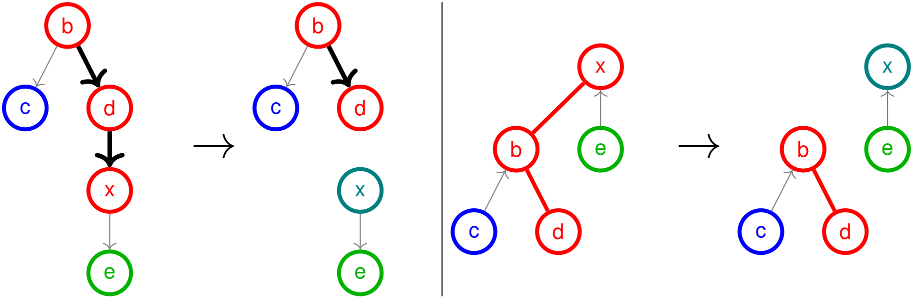
Access - complejidad

- Aunque no lo parezca, la cantidad de cambios de arista preferida a no preferida y viceversa es $O(log(n))$ por acceso (amortizado).
- Esto prueba una cota superior de $O(log^2(n))$ amortizado por acceso.
- Usando propiedades de splay trees, se puede demostrar que la complejidad amortizada es en realidad $O(log(n))$.
- Las operaciones de link-cut tree se pueden implementar fácilmente usando `access`, y su complejidad también resulta $O(log(n))$ amortizado.

Link-cut tree - `cut(x)`

`cut(x)` corta la arista de x a su padre:

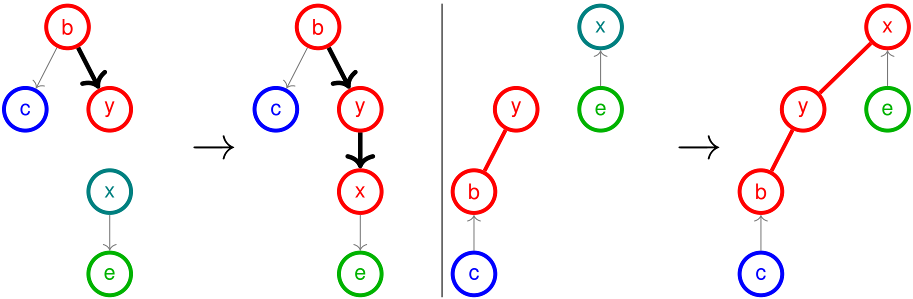
- Hacer `access(x)`
- Eliminar la arista de x a su hijo izquierdo en el splay tree.



Link-cut tree - `link(x,y)`

`link(x,y)` hace a x hijo de y (x debe ser raíz de su árbol):

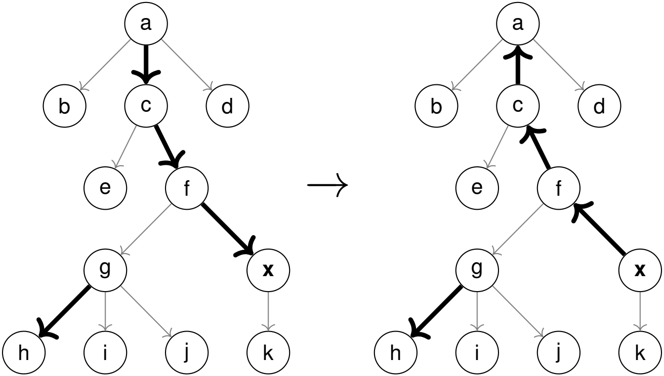
- Hacer `access(x)` y `access(y)`
- Hacer a y hijo izquierdo de x en el splay tree de x.



Link-cut tree - `makeRoot(x)`

`makeRoot(x)` transorma a x en la raíz de su árbol (representado):

- Hacer `access(x)`.
- Hacer `reverse` lazy del splay tree de x.



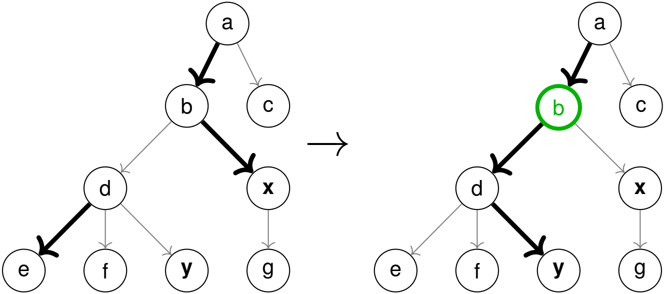
Link-cut tree - `getRoot(x)`

`getRoot(x)` devuelve la raíz del árbol de x:

- Hacer `access(x)`.
- Encontrar el primer elemento del splay tree de x, moviéndose a la izquierda mientras pueda.
- Hacer splay del elemento (para garantizar complejidad amortizada) y devolverlo.

`lca(x,y)` devuelve el ancestro común más bajo de x e y (en el árbol representado).

- Hacer `access(x)` y luego `access(y)`.
- Si al hacer `access(y)` no hubo ninguna iteración de “cambiar hijo preferido”, es porque y es ancesstro de x. Devolver y.
- Si no, devolver el último nodo al cuál se le cambió el hijo preferido al hacer `access(y)`.



Link-cut tree - `depth(x)`

`depth(x)` devuelve la profundidad de x (en el árbol representado).

- Hacer `access(x)`.
- Asumiendo que fuimos manteniendo la cantidad de descendientes en cada nodo de splay tree, el valor para x contiene la cantidad de nodos entre x y la raíz del árbol representado (inclusive).
- Esa cantidad menos 1 es la profundidad de x.

Link-cut tree - `lift(x,k)`

`lift(x,k)` devuelve el k-ésimo ancestro de x.

- Hacer `access(x)`.
- Sea `i=depth(x)-k`.
- Devolver el i-ésimo elemento del splay tree visto como lista (antes hacer `splay` sobre el mismo para garantizar complejidad amortizada).

Link-cut tree - aggregate, update y lazy

- Link-cut tree permite hacer operaciones sobre caminos de un árbol de forma similar a como segment tree permite hacer operaciones sobre segmentos.
- Para eso, implementamos las operaciones en el splay tree, como explicamos antes.
- Para hacer `update(x)`, simplemente hacemos `access(x)` y actualizamos el valor y la agregación en x.
- Para hacer `aggregate(x,y)` o `lazy(x,y)`:
  - Hacemos `makeRoot(x)` y `expose(y)` para tener un splay tree con el camino de interés.
  - Luego realizamos la operación sobre ese splay tree.
  - Opcionalmente (si nos importa mantener las raíces originales) hacemos `z=getRoot(x)` antes y `makeRoot(z)` después.

Link-cut tree - aggregate, update y lazy (sobre aristas)

Si queremos mantener valores en las aristas y realizar operaciones sobre las aristas de un camino (en lugar de los nodos):

- Creamos un nodo adicional para cada arista, y guardamos el valor ahí (este nodo tiene dos vecinos que son los extremos de la arista “real”).
- En los “nodos que representan nodos” guardamos un valor neutro.
- Cuando hacemos `link` o `cut`, necesitamos hacer un `link` o `cut` adicional (respectivamente).

Más recursos

- Implementación compacta de link-cut tree[1]: `https://github.com/mhunicken/icpc-team-notebook-el-vasito/blob/master/data_structures/linkcut2.cpp`
- Algunos problemas para testear link-cut tree:
  - `https://www.spoj.com/problems/DYNACON1/`
  - `https://www.spoj.com/problems/DYNALCA/`
  - `https://codeforces.com/gym/102059/problem/A`
- Extensión de link-cut tree para hacer queries sobre sub-árboles: `https://codeforces.com/blog/entry/67637`
- Notas de la clase sobre link-cut tree del MIT (para quien le interese el análisis de la complejidad): `https://courses.csail.mit.edu/6.851/spring12/scribe/L19.pdf`

[1] Recomiendo de todos modos intentar hacer una implementación propia para entender bien la estructura

Temas relacionados

- *Euler-tour tree*: Es otra representación de árboles (más sencilla) que conviene para hacer queries sobre subárboles (pero no soporta queries sobre caminos).
- *Heavy-light decomposition*: Permite hacer queries y updates sobre caminos de subárboles (lo mismo que link-cut tree pero sin modificar la estructura del árbol). Es bueno saberlo porque la implementación es más sencilla y corta.
- *Treap*: Árbol binario de búsqueda randomizado. Por lo general más sencillo de usar que splay tree, y soporta persistencia muy fácilmente.

¡Gracias!