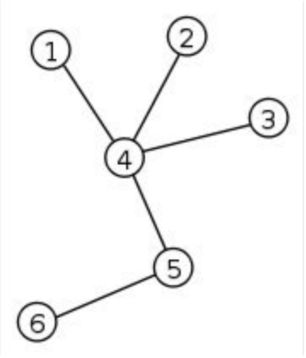
Link Cut Tree

INDEX

- Definición(yerson)
- Heavy Light Decomposition(yerson)
- Splay Tree
- BST(Binary Search Tree) to Balanced Search Trees -> Luis Simple
- Operaciones Luis
  - access()
  - make_tree()
  - link(v,w)
  - cut(v)
  - find_root(v)
  - path_aggregate(v)
- Análisis(complejidad) ->
- Aplicaciones -> Luis
- Referencias

Link-cut Trees



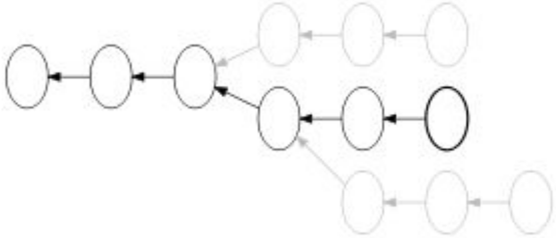
Definition

A link-cut tree is a data structure for representing a forest, a set of rooted trees to

provides a complicated structure but reduces the cost of the operations from amortized O(log n) to worst case O(log n).

The represented forest may consist of very deep trees.

- Represent parent pointer trees.

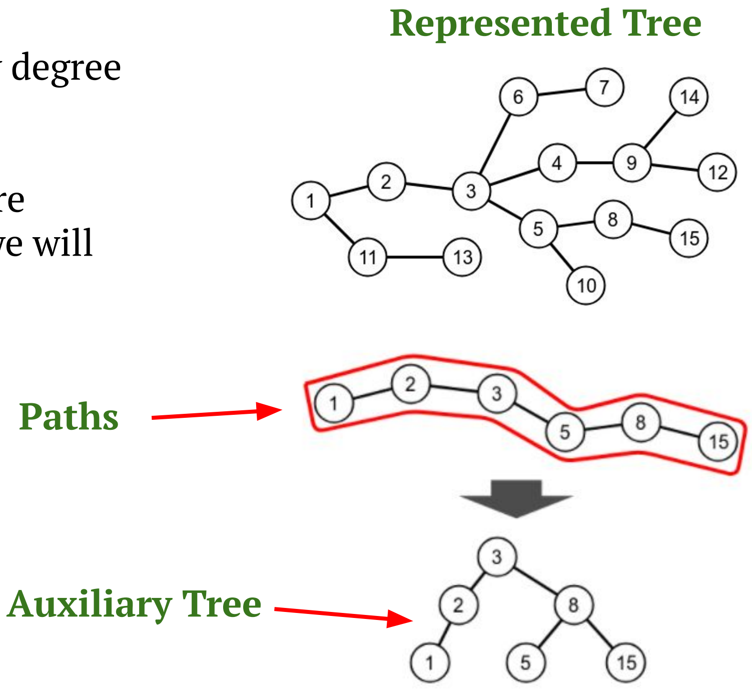


Structure

- We take a tree where each node has an arbitrary degree of unordered nodes and split it into paths

- We call this the represented tree. These paths are represented internally by auxiliary trees (here we will use splay trees)

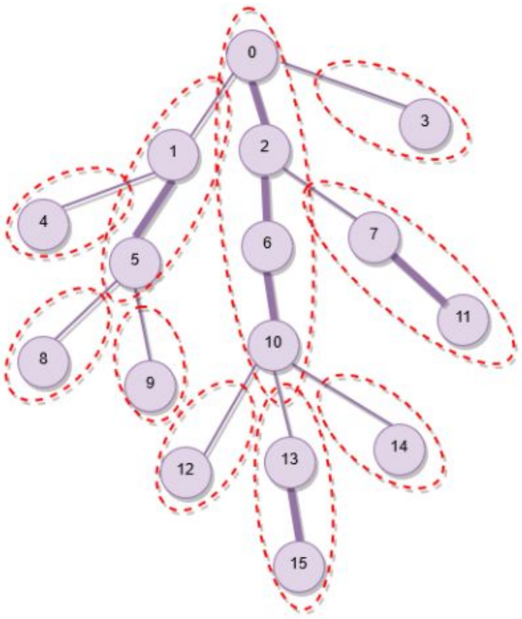
- Operations
  - make tree()
  - link(v,w)
  - cut(v)
  - find root(v)
  - path aggregate(v)



4

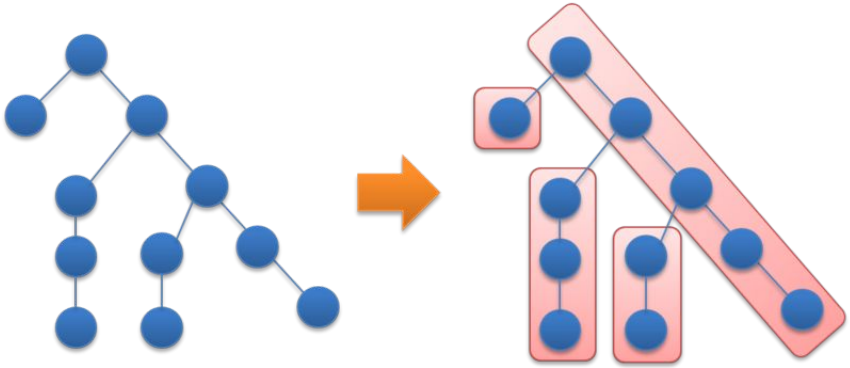
Heavy Light Decomposition

- Is a fairly general technique that allows us to effectively solve many problems that come down to queries on a tree.
- Is one of the most used techniques in competitive programming.
- The essence of this tree decomposition is to **split the tree into several paths**
- we can reach the root vertex from any v by traversing at most log(n) paths.
- In addition, none of these paths should intersect with another.



Heavy Light Decomposition

It is clear that if we find such a decomposition for any tree it will allow us to reduce certain single queries of the form.



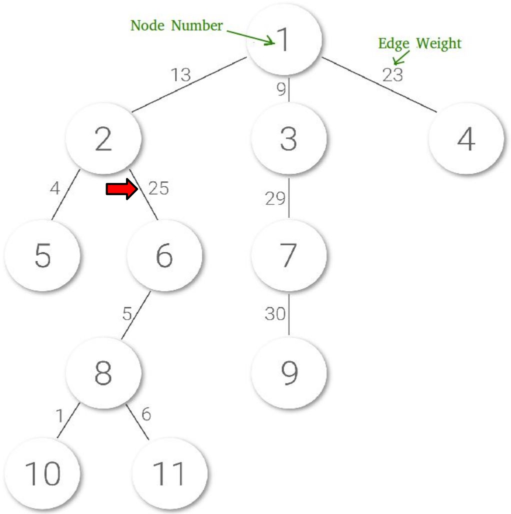
- Calculate something on the path from **a** to **b**.
- Calculate something on the segment **[l,r]** of the kth path.

- **change(a, b)**
- **maxEdge(a, b)**

Example

Suppose we have **an unbalanced tree (not necessarily a Binary Tree) of n nodes**, and we have to perform operations on the tree to answer a number of queries, each can be of one of the two types:

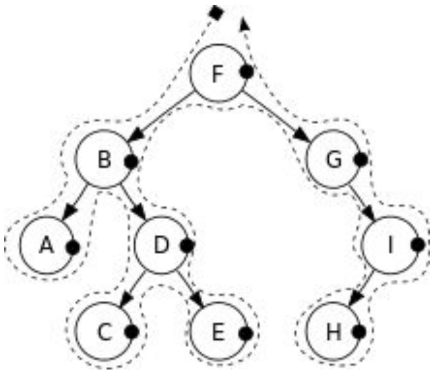
- **change(a, b)**: Update weight of the ath edge to **b**.
- **maxEdge(a, b)**: Print the **maximum edge** weight on the path from node **a** to node **b**.



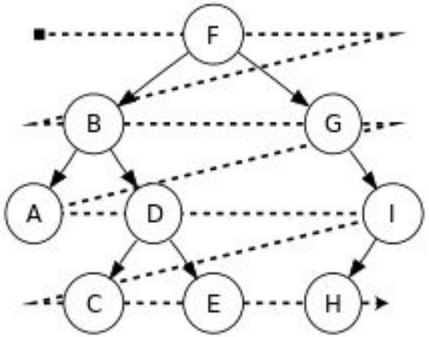
For example maxEdge(5, 10) **should print 25.**

Simple Solution

A Simple solution is to **traverse the complete tree** for any query. Time complexity of every query in this solution is **O(n)**.



PreOrder
InOreder
PostOrder



Transversal

HLD Based Solution

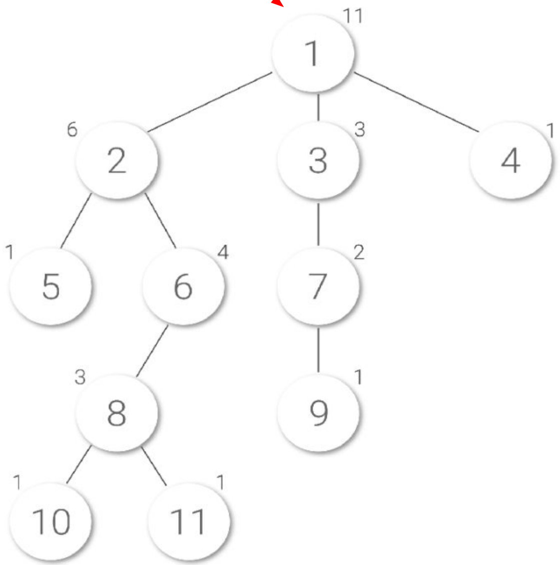
- **Segment Tree**
  - Operations **O(logn)**
  - Input is **[, , ,]** ?

- **Size** of a node x is number of nodes in subtree rooted with the node x.

- **HLD** of a rooted tree is a method of decomposing the vertices of the tree into disjoint chains (**no two chains share a node**)

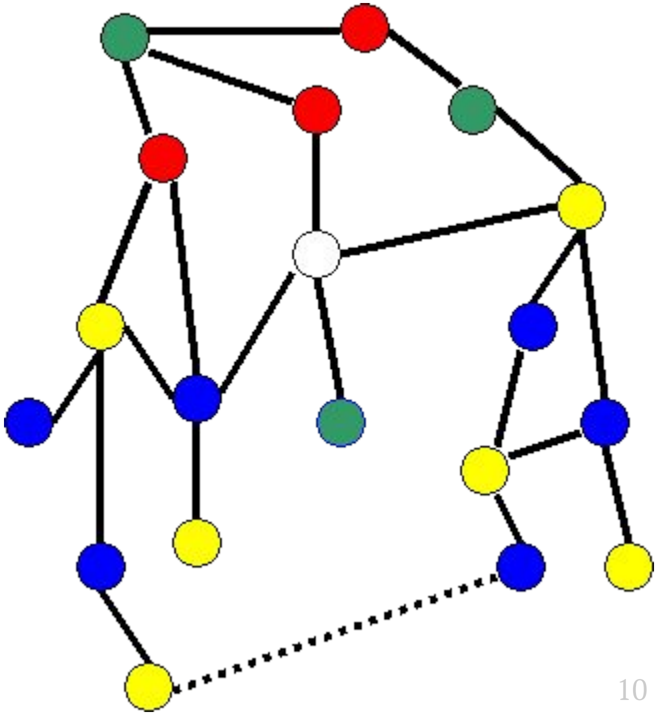
- To achieve important **asymptotic time** bounds for certain problems involving trees.

Size of Subtree rooted with this node.



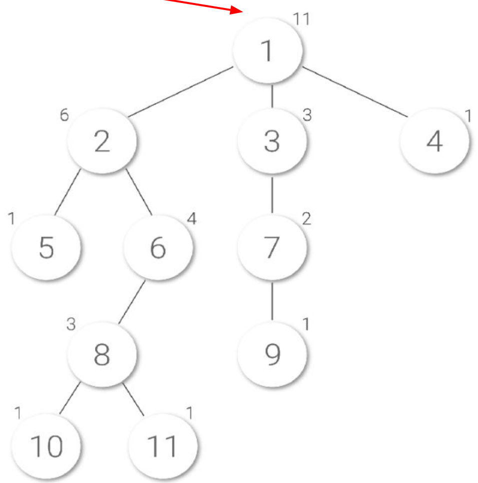
HLD Based Solution

- **HLD** can also be seen as 'coloring' of the tree's edges.
- The '**Heavy-Light**' comes from the way we segregate edges.
- We use **size** of the subtrees rooted at the nodes as our criteria.
- An edge is heavy **if size(v) > size(u)** where **u is any sibling of v.** If they come out to be equal, we pick any one such v as special.

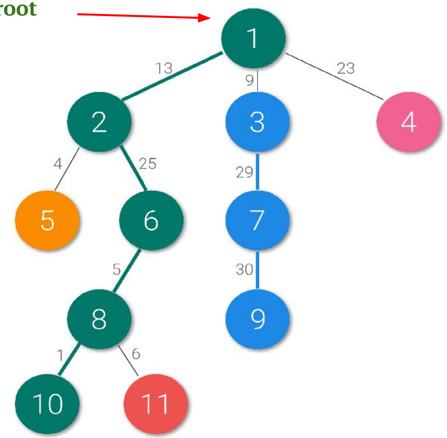


HLD Based Solution

Size of Subtree rooted with this node.



This edge is heavy because size of subtree rooted with 2 is six which is more than other children of root

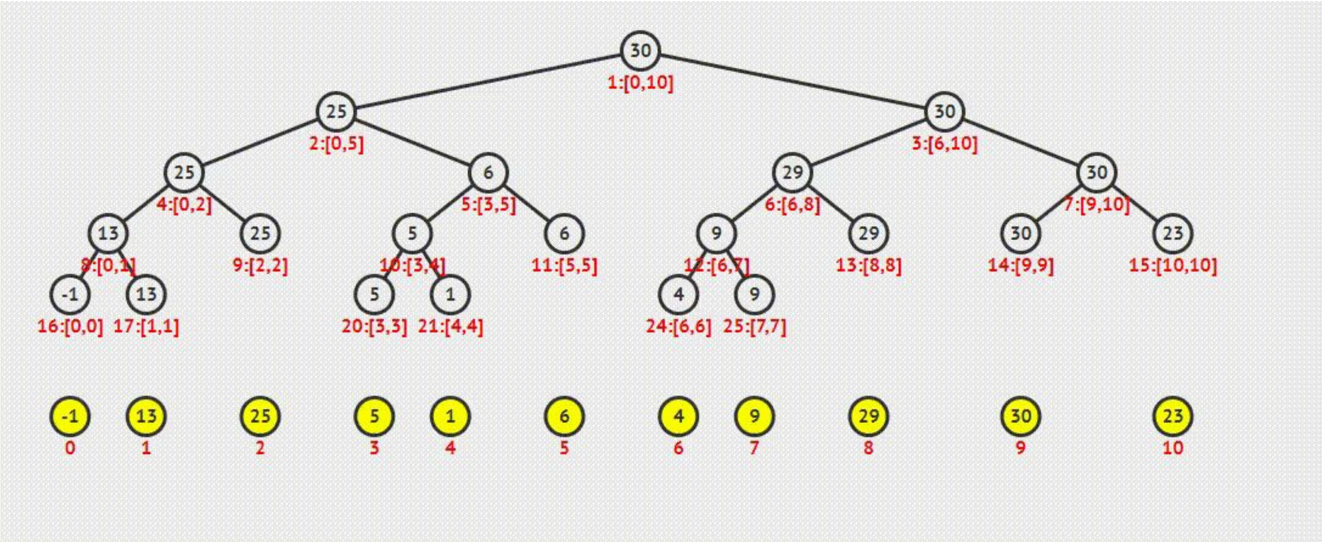


- ❑ Different colors indicate different chains.
- ❑ Edges colores **black** are **light edge.**

input : -1,13,25,5,1,6,4,9,29,30,23

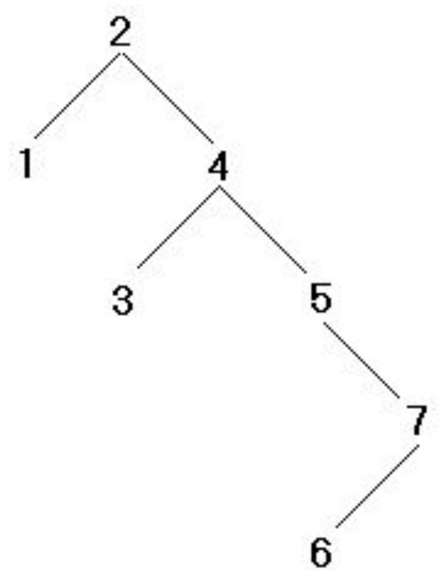
Max Segment tree

input : -1,13,25,5,1,6,4,9,29,30,23



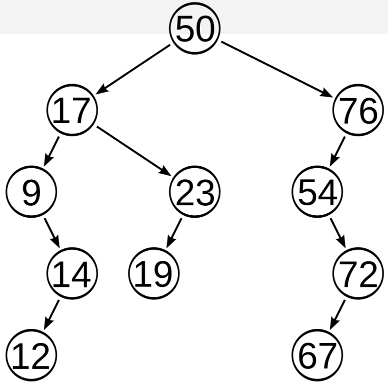
BST: Binary Search Tree

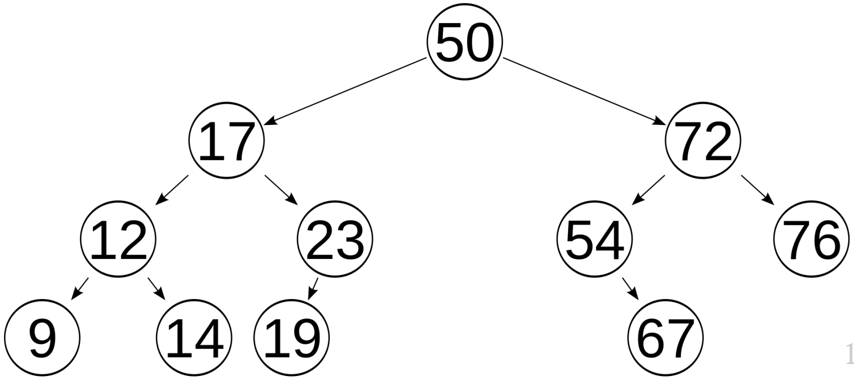
- BST is the Rooted Binary Tree
- Whose internal nodes stored a key and additionally a tree
- Following the properties
  - Subtree to the left of a node contains nodes with lower values
  - Subtree to the left of a node contains nodes with lower values



BBST:balancing binary search tree

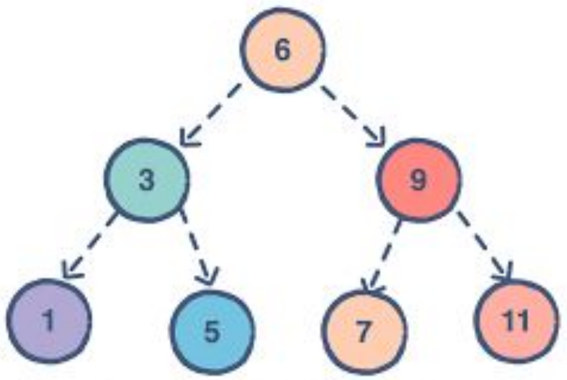
- It is a self-balancing or height-balanced binary search tree
- In simple terms it is any binary search tree that automatically maintains its height
- Examples:
  - AVL,
  - Tree B,
  - Red-black Tree and
  - **Splay Tree** …





Splay Tree

- A splay tree is just a binary search tree that has excellent performance in the cases where some data is accessed more frequently than others.
- The tree self-adjusts after lookup, insert and delete operations
- All the operations in splay tree are involved with a common operation called "Splaying".



An example of a binary search tree

15

Rotations in Splay Tree

Zig Rotation

Rotación de Zag

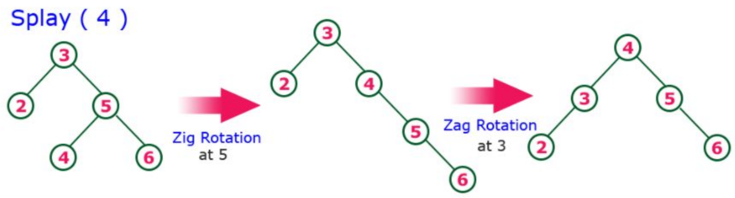
Rotación Zig-Zig

Rotación Zag-Zag

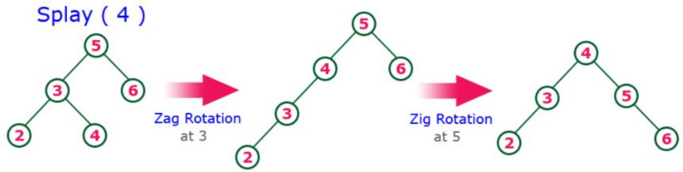
16

Rotations in Splay Tree

Rotación en zig-zag



Rotación Zag-Zig



17

Operations

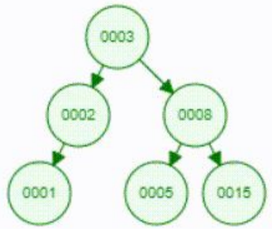
Insert

Step 1 - Check whether tree is Empty.
Step 2 - If tree is Empty then insert the newNode as Root node and exit from the operation.
Step 3 - If tree is not Empty then insert the **newNode** as leaf node using Binary Search tree insertion logic.
Step 4 - After insertion, Splay the newNode

INSERT:1,15,5,2,8,3

Delete

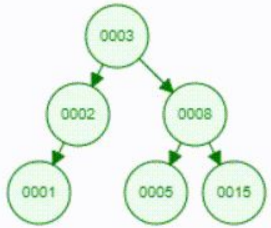
The deletion operation in splay tree is **similar to deletion operation in Binary Search Tree**. But before deleting the element, **we first need to splay that element and then delete it from the root** position. Finally join the remaining tree using binary search tree logic.



DELETE: 5,15,1

Search

similar to search operation in Binary Search Tree



SEARCH: 15,5

Comparison of Search Trees

Search Tree	Average Case			Worst Case		
	Insert	Delete	Search	Insert	Delete	Search
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)
AVL Tree	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
B - Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Red - Black Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Splay Tree	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$

19

Operations

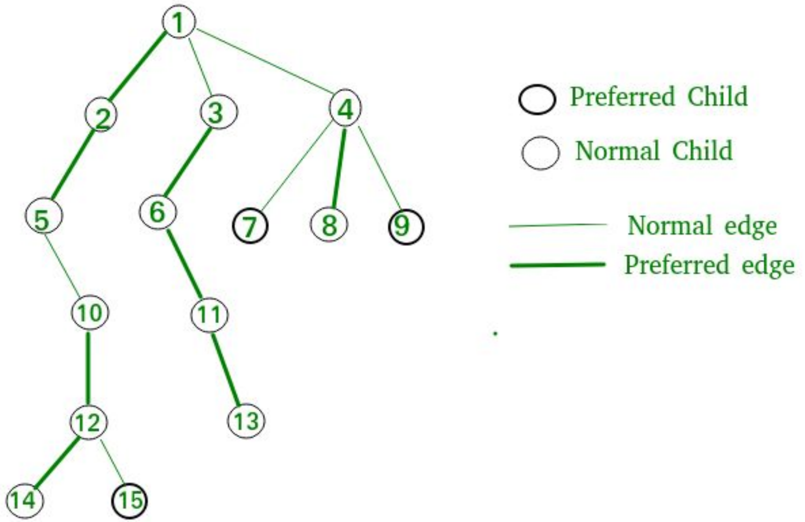
- By using Link-Cut Tree we want to keep a forest rooted
- The data structure must support operations in amortized O (log n) time

link	*O(log n)*
Cut	*O(log n)*
FindRoot	*O(log n)*
Path	*O(log n)*

Consider

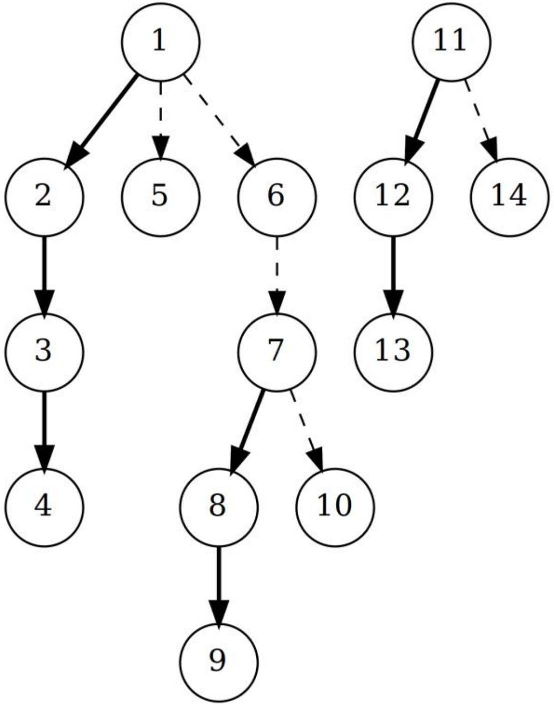
- Link-Cut Tree is similar with Tango Tree

- Use the notion of Preferred Child and preferred path



Remember

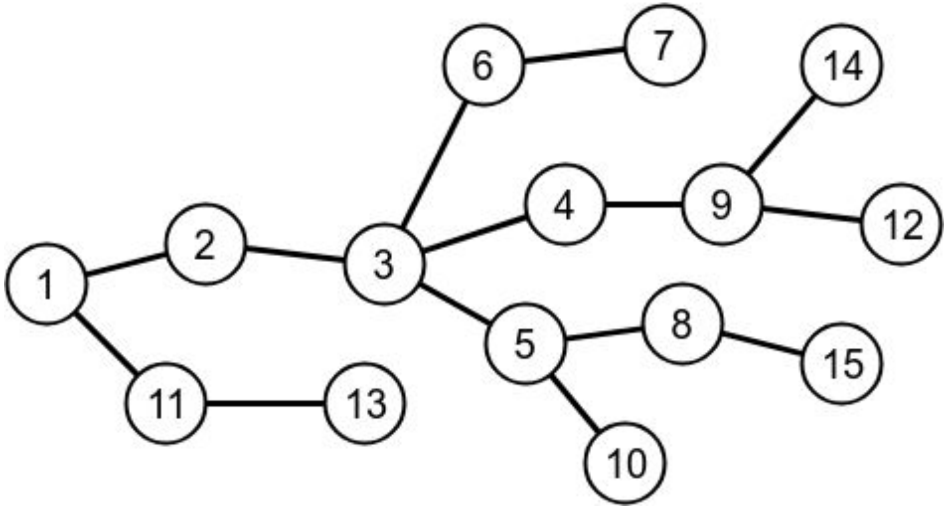
- In general we have an undirected graph
- Use a BBST for internal rendering
- Exactly a Splay Tree
- The represented Tree is divided into Routes
- There is a main path pointer per Helper Tree



General Operations

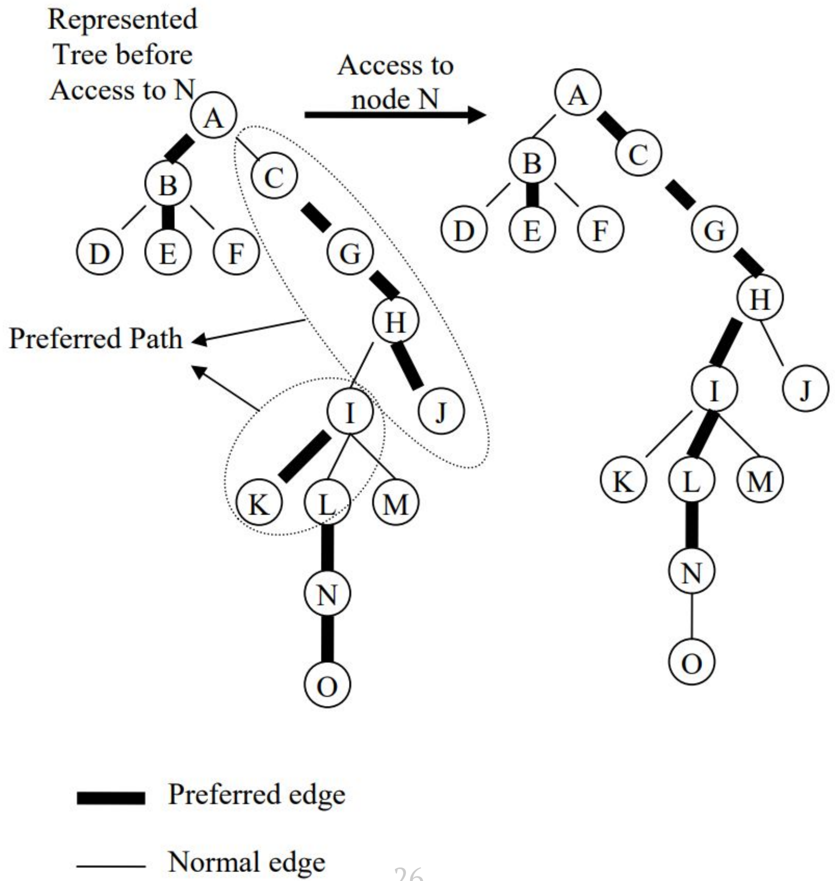
- make_tree: Make a tree with a single node
- Link: Connects a root vertex with an intermediary
- Cut: Unlink the connection with her father
- Find_Root: Returns the root of the corresponding tree
- Path_Agrgregate: Allows aggregation functions

- Consider the undirected graph
- Then consider as a tree with a root of 1 vertex
- We will use heavy light decomposition

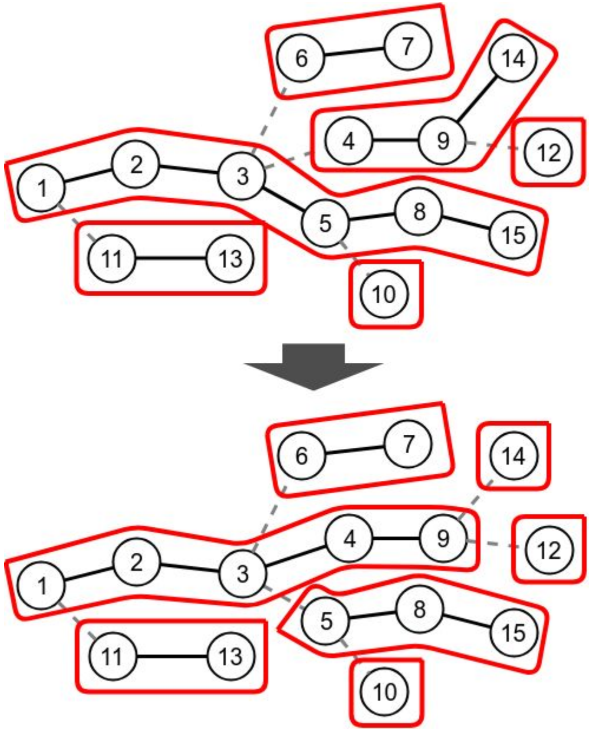


Access(Node v)

- ❏ It is a subroutine for the other operations
- ❏ Restructure the tree T of auxiliary trees that contains the vertex v
- ❏ Connect all the way from root vertex to vertex v
- ❏ Preferred routes change
- ❏ Operation **Splay**

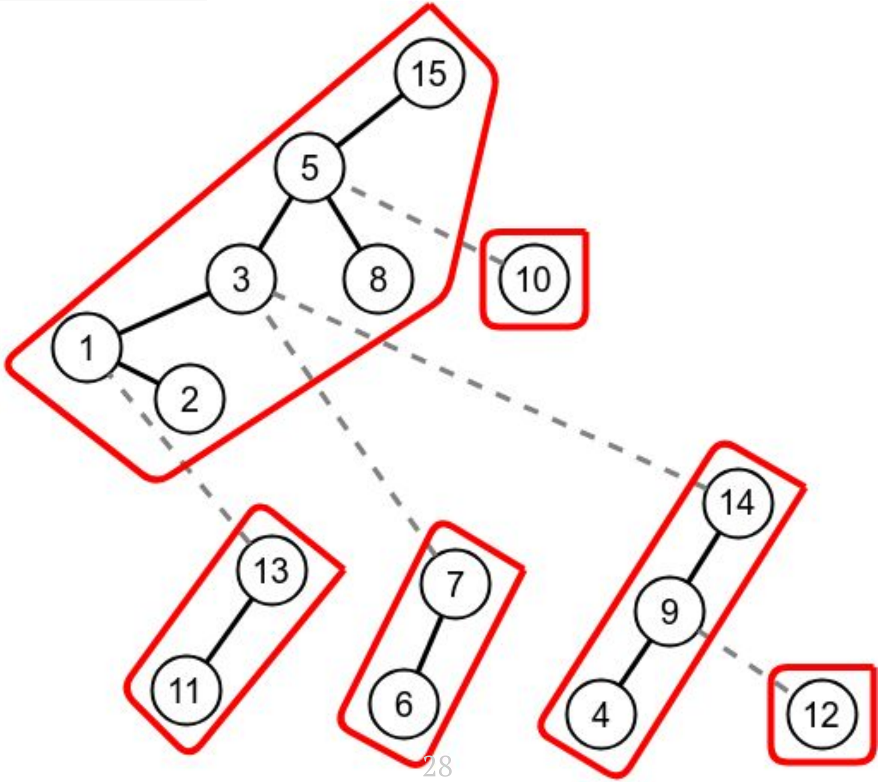

Represented Tree before Access to N
Access to node N
Preferred Path
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
Preferred edge
Normal edge

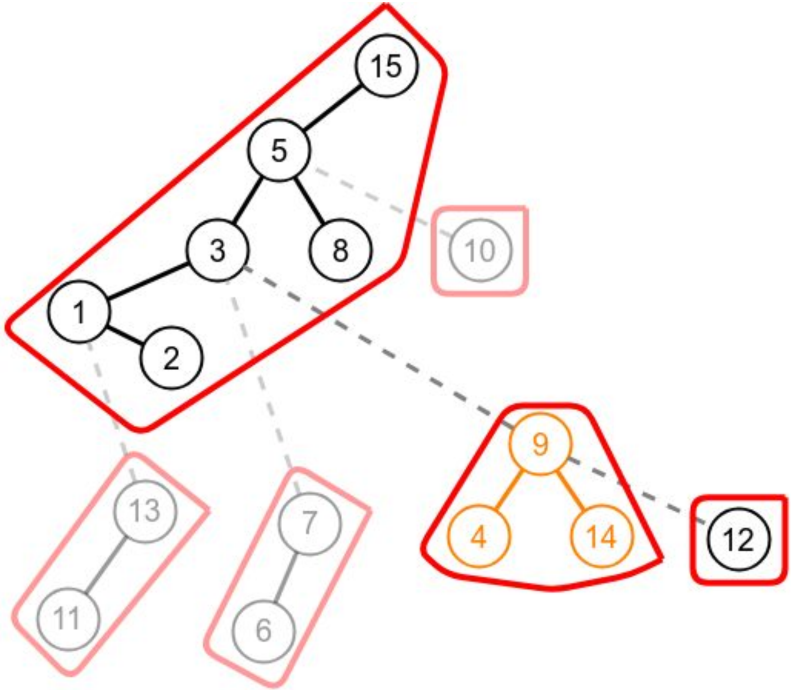
Access (9) : General idea



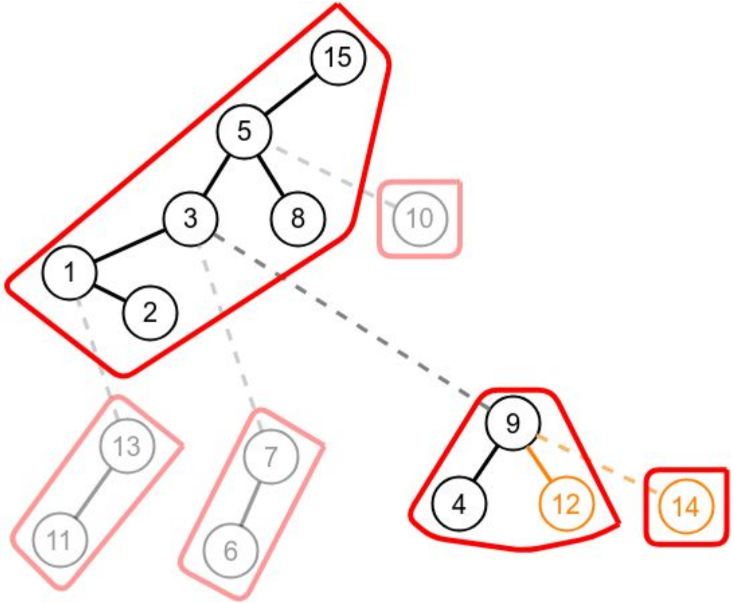
Access(12): Steps Complete

Splay(12)

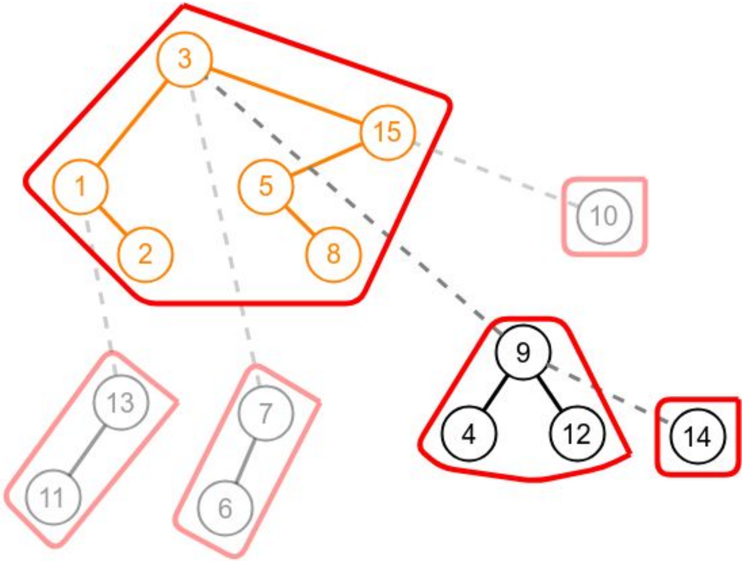




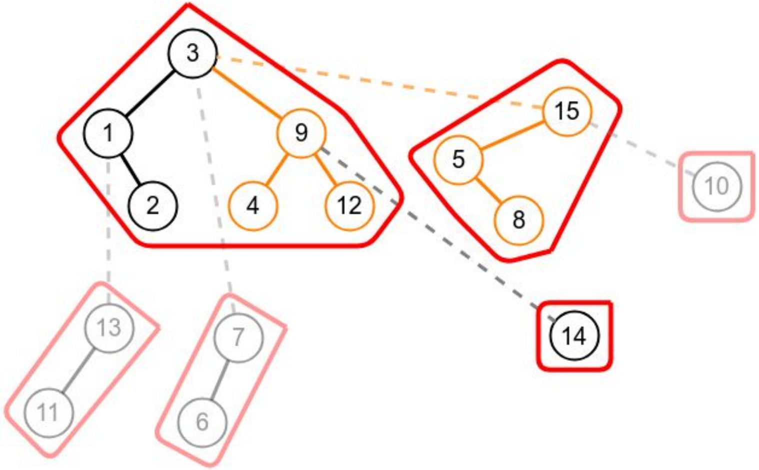
Splay (9)



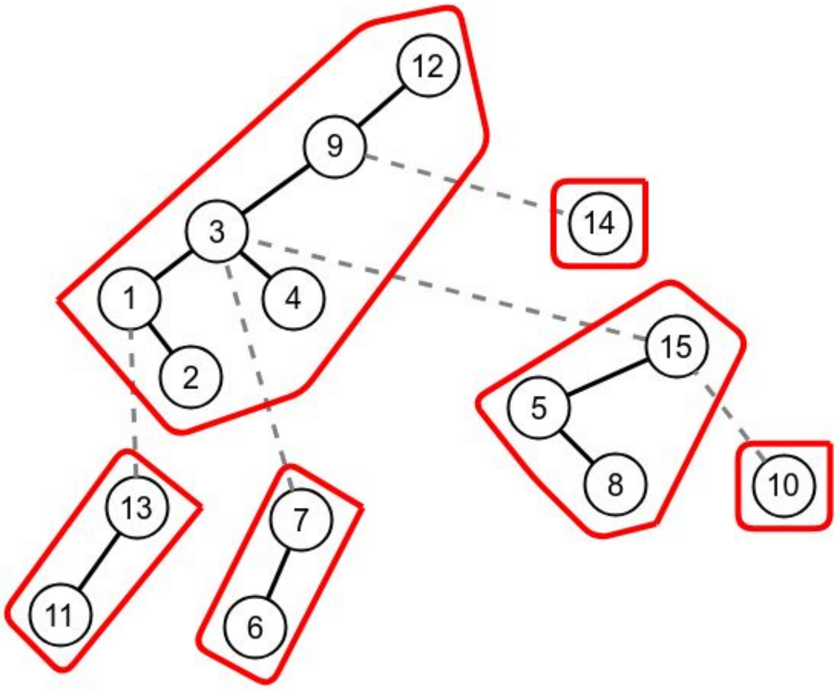
New Connection



Splay (3)



New Connection



Splay(1)

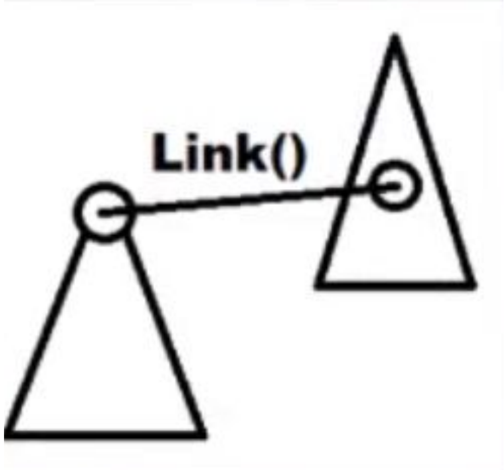
Pseudocode:

ACCESS(v)

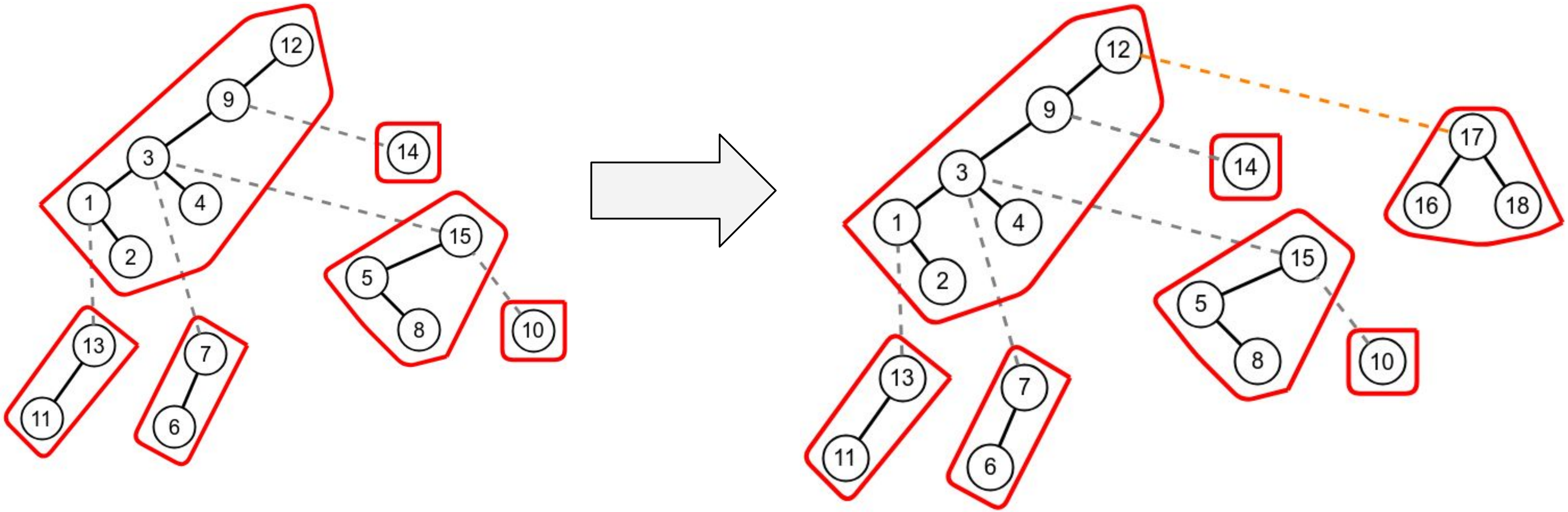
- Splay v within its auxiliary tree, i.e. bring it to the root. The left subtree will contain all the elements higher than v and right subtree will contain all the elements lower than v
- Remove v's preferred child.
  - path-parent(right(v)) $\leftarrow v$
  - right(v) $\leftarrow$ $null$ (+ symmetric setting of parent pointer)
- loop until we reach the root
  - $w \leftarrow$ path-parent(v)
  - splay w
  - switch w's preferred child
    - path-parent(right(w)) $\leftarrow w$
    - right(w) $\leftarrow v$ (+ symmetric setting of parent pointer)
    - path-parent(v) $\leftarrow$ $null$
  - $v \leftarrow w$
- splay v just for convenience

Link (Node v , Node w)

- Link 2 rendered trees
- **Access** both v and w
- Add the edge between v and w
- The right vertex v connects with w
- v will be the son of w



Link (12 , 17)



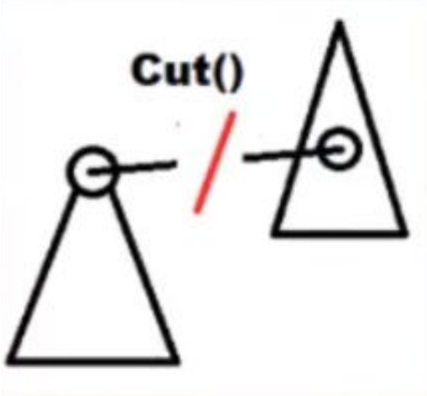
Pseudocode:

$\mathsf{LINK}(v, w)$

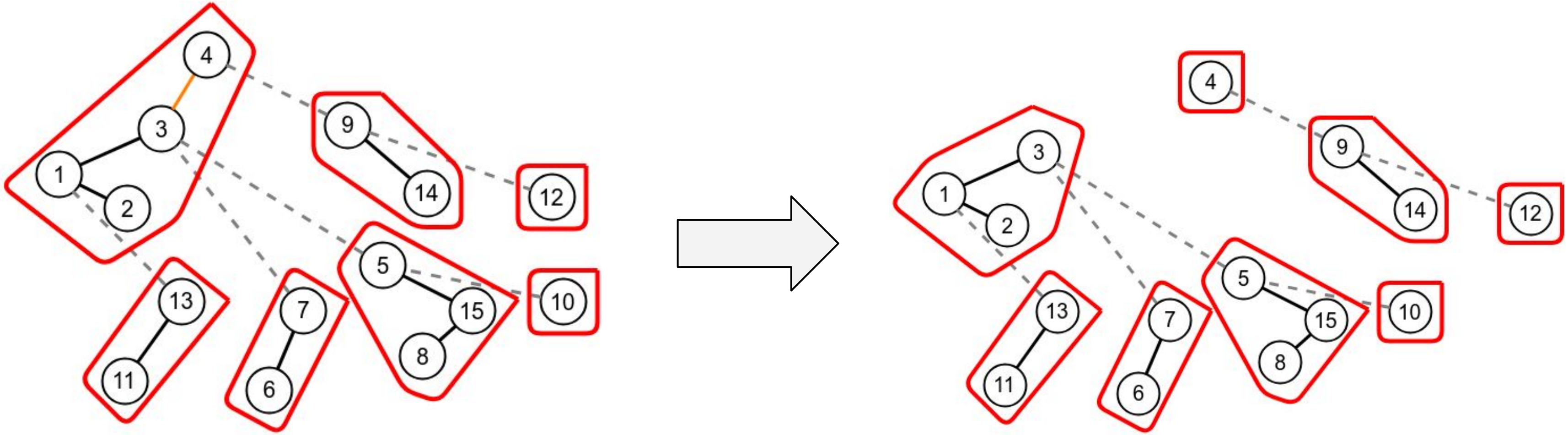
- $\mathsf{access}(v)$
- $\mathsf{access}(w)$
- $\mathsf{left}(v) \leftarrow w$ (+ symmetric setting of parent pointer)

Cut(Nodo v)

- Cut the edge between v and parent of v in the represented tree
- Separate nodes in subtree v from the tree of auxiliary trees
- Access (v)
- Spawn 2 auxiliary trees



Cut (4)



Pseudocode:

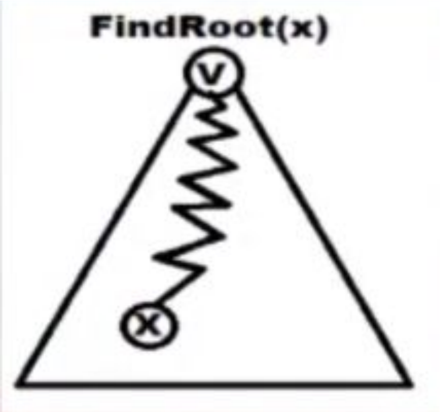
CUT(v)

– access(v)

– left(v) $\leftarrow null$ (+ symmetric setting of parent pointer)

Find_Root(Nodo v)

- Returns the root where v is a node of this tree
- Helps us determine if 2 nodes are connected



Pseudocode:

FIND_ROOT(v)

- access(v)
- Set v to the smallest element in the auxiliary tree, i.e. to the root of the represented tree
  - $v \leftarrow$ left(v) until left(v) is $null$
- access r
- return r

Path_aggregate(Node v)

- Auxiliary function
- For this operation we want to do some aggregate function on all the nodes
- Returns an aggregate set as (max / min / sum) of edge weights
- **Access (v)**

Pseudocode:

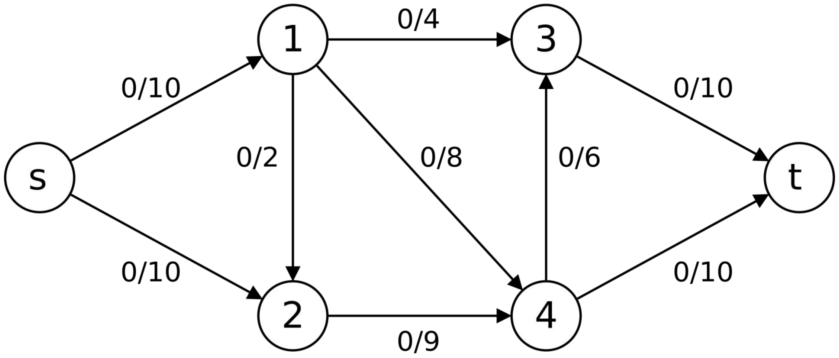
PATH-AGGREGATE(v)

- access(v)
- return v.subtree-sum (augmentation within each aux. tree)

Análisis

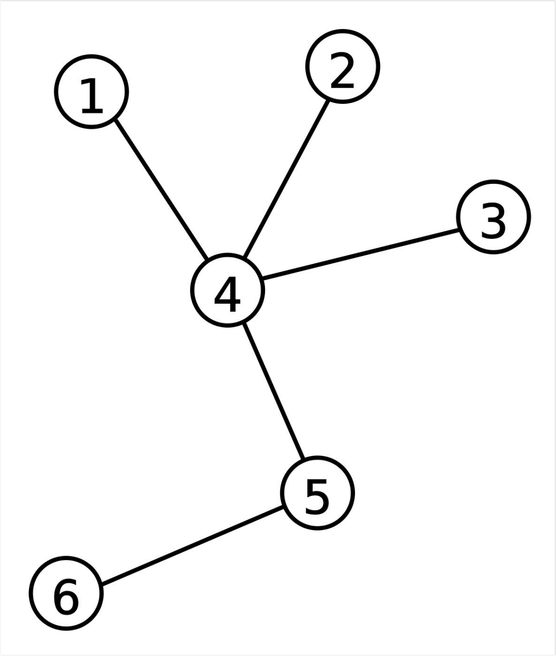
Applications

- It can be used to speed up the Dinic algorithm



$$O(V^2E) \Rightarrow O(EV \log V)$$

- Can be used to solve dynamic connectivity problems for acyclic graphs



References

- Akiba, T., 2020. *Link-Cut Treeの実装メモ - 日々drdrする人のメモ* . [online] 日々drdrする人のメモ . Available at: <https://smijake3.hatenablog.com/entry/2018/11/19/085919> [Accessed 6 December 2020].
- Holmgren, J., Jian, J. and Stepanenko, M., 2012. *Advanced Data Structures*. [ebook] Available at: <http://courses.csail.mit.edu/6.851/spring12/scribe/L19.pdf> [Accessed 6 December 2020].
- En.wikipedia.org. 2020. *Self-Balancing Binary Search Tree*. [online] Available at: <https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree> [Accessed 6 December 2020].
- Weinstein, O., 2019. *Dynamic Graph Data Structures*. [ebook] Available at: <https://posobin.com/advancedDS/advancedDS_lec13.pdf> [Accessed 6 December 2020].
- Heavy-Light Decomposition. (2014, 24 junio). Heavy-Light Decomposition. https://math314.hateblo.jp/entry/2014/06/24/220107
- Splay Tree datastructure. (2015, 20 marzo). Splay Tree Datastructure. http://www.btechsmartclass.com/data_structures/splay-trees.html
- Varyani, Y. (2017, 15 mayo). Heavy Light Decomposition | Set 1 (Introduction). Heavy Light Decomposition | Set 1 (Introduction). https://www.geeksforgeeks.org/heavy-light-decomposition-set-1-introduction/