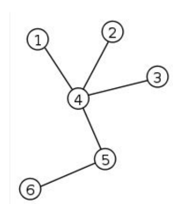
Link Cut Tree

INDEX

Link-cut Trees



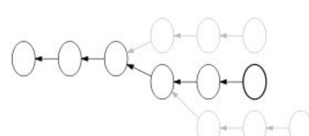
Definition

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).

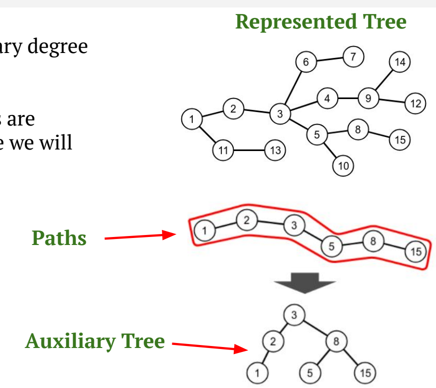
The represented forest may consist of very deep trees.

- Represent parent pointer trees.



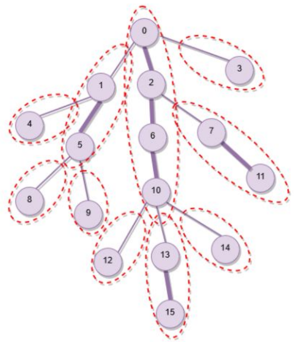
Structure

- 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)



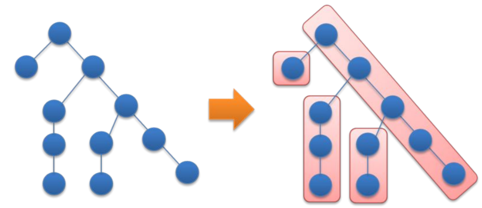
Heavy Light Decomposition

- 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.

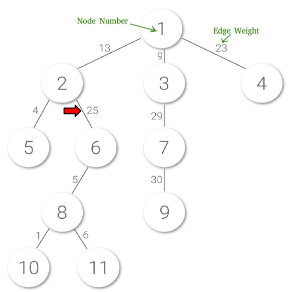
- **change(a, b)**
- **maxEdge(a, b)**

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:

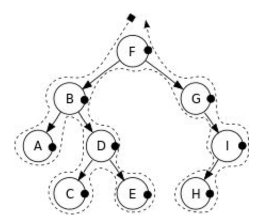
- **change(a, b)**: Update weight of the ath edge to **b**.
- **maxEdge(a, b)**: Print the **maximum edge** weight on the path from node **a** to node **b**.

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

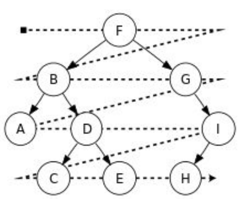


Simple Solution

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



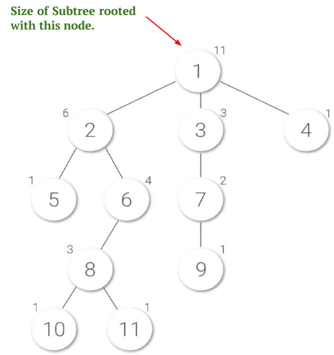
PreOrder
InOreder
PostOrder



Transversal

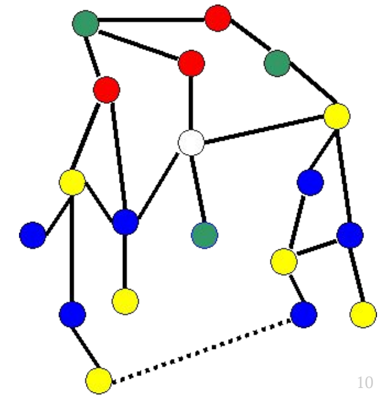
HLD Based Solution

- **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.

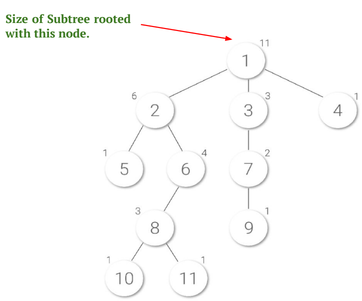


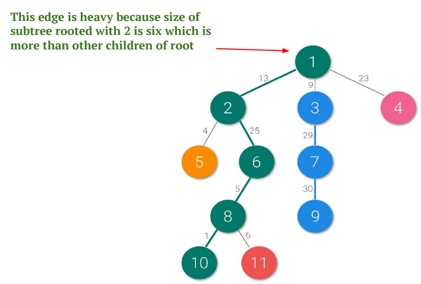
HLD Based Solution

- **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



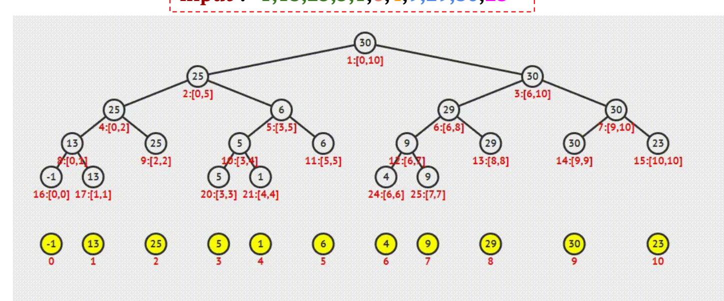


- Different colors indicate different chains.
- Edges colores **black** are **light edge.**

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

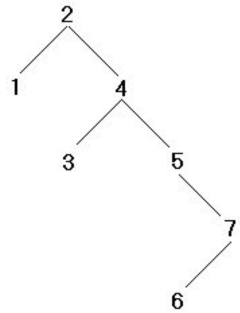
Max Segment tree

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

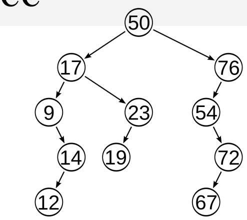


BST: Binary Search Tree

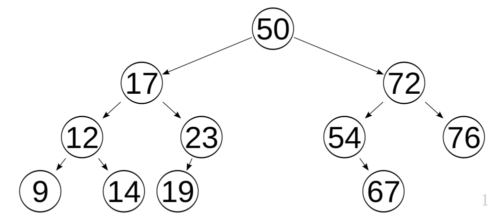
- BST is the Rooted Binary Tree
- Whose internal nodes stored a key and additionally a tree
- Following the properties
  - Subtree to the left of a node contains nodes with lower values
  - Subtree to the left of a node contains nodes with lower values



BBST:balancing binary search tree

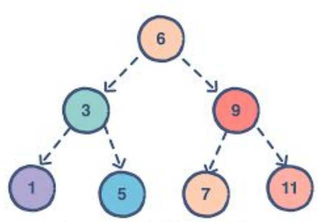


- It is a self-balancing or height-balanced binary search tree
- In simple terms it is any binary search tree that automatically maintains its height
- Examples:
  - AVL,
  - Tree B,
  - Red-black Tree and
  - **Splay Tree** ...



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".

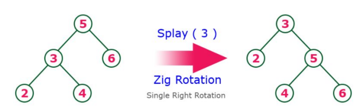


An example of a binary search tree

15

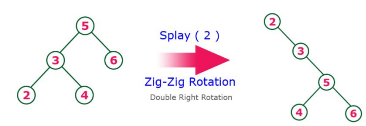
Rotations in Splay Tree

Zig Rotation



Rotación de Zag

Rotación Zig-Zig

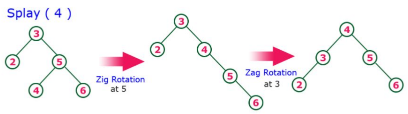


Rotación Zag-Zag

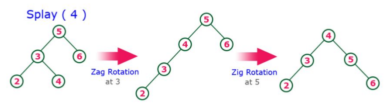
16

Rotations in Splay Tree

Rotación en zig-zag



Rotación Zag-Zig



17

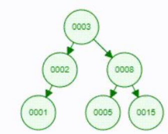
Operations

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

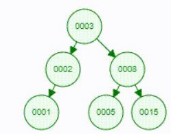
Delete

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.



Search

similar to search operation in Binary Search Tree



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

DELETE: 5,15,1

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