Link-Cut Tree

Algorithm Design

YuXin Zeng

Shanghai Jiao Tong University

Dynamic Trees Problem

Introduction

Maintain a tree, consider the following operations:

- Change the path weight between two points.
- Query the path weights between two points.
- Modifies an idea tree weight.
- Query the weights and values of a tree.
- Disconnect and connect some edges to make sure it's still a tree.

This is a Dynamic Trees Problem

Dynamic Trees Problem

What is the Dynamic Trees Problem ?

- Maintain a forest, support delete edge, add edge, ensure that the operation is still the forest, maintain the relevant information.
- Common operations include two-point connectivity, two-point path weights, connecting two points, cutting an edge, modifying information, and so on.

Solution of Dynamic Trees Problem : Link-Cut Tree

We will discuss one data structure called Link-Cut trees that achieves logarithmic amortized time for all operations.

Heavy Path Decomposition

Review Heavy Path Decomposition from the perspective of Link Cut Tree

- Divide the whole tree into sub-tree sizes and re-label it.
- We find that after re-label, some continuous intervals are formed in the tree in the unit of chain, and the interval operation can be carried out with the segment tree.

Dynamic Trees Problem

Now we turn to the Dynamic Trees Problem

- We found that the tree splitting that we just discussed about was conditional on sub-tree size.
- So can we redefine a partition to make it more suitable for our Dynamic Trees Problem?
- Consider what chains are needed for the Dynamic Trees Problem, since we dynamically maintain a forest, obviously we want the chain to be the one we specify so that we can use it to solve.

Real Chain Splitting

Real Chain Splitting

Definition

For the edges that a point connects to all its sons, we choose an edge ourselves and divide it. We call the selected edge the real side and the other edges the imaginary side.

Real Chain Splitting

Definition

For the edges that a point connects to all its sons, we choose an edge ourselves and divide it. We call the selected edge the real side and the other edges the imaginary side.

Definition

For real edges, we call the sons it connects real sons.

Real Chain Splitting

Definition

For the edges that a point connects to all its sons, we choose an edge ourselves and divide it. We call the selected edge the real side and the other edges the imaginary side.

Definition

For real edges, we call the sons it connects real sons.

Definition

For a chain of real edges, we also call it a real chain.

Real Chain Splitting

Definition

Remember the most important reason we choose real chain partitioning: it is what we choose, flexible and mutable.

Real Chain Splitting

Definition

Remember the most important reason we choose real chain partitioning: it is what we choose, flexible and mutable.

Definition

Because of this flexibility, we use Splay Tree to maintain these solid chains.

Link-Cut Tree

- We can simply think of Link-Cut Tree as using some Splay to maintain the dynamic tree chain division, in order to achieve the interval operation of the dynamic tree.
- For each chain, we create a Splay to maintain information for the entire chain.
- Now, we can learn the specific structure of Link-Cut Tree.

Auxiliary Tree

Auxiliary Tree

Auxiliary Tree. Some Splay Tree constitute an Auxiliary Tree, and each Auxiliary Tree maintains a Tree. Some Auxiliary Trees constitute Link Cut Tree, which maintains the whole forest.

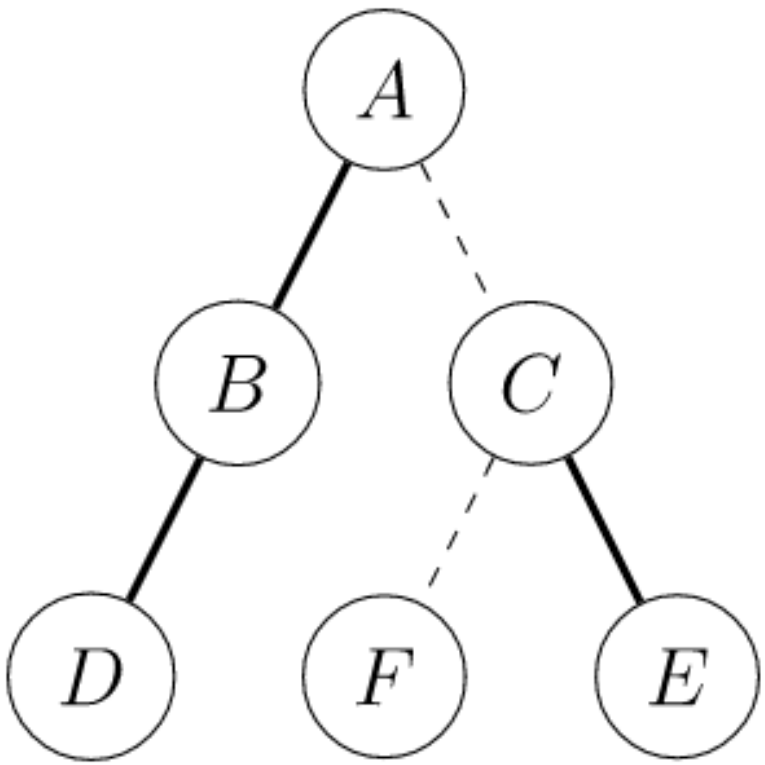
The Auxiliary Tree is composed of multiple Splay Trees. Each Splay Tree maintains a path in the original Tree, and the sequence of points obtained by traversing this Splay Tree in middle order corresponding to a path "from top to bottom" of the original Tree from front to back.

Auxiliary Tree

- Each node in the original tree corresponds to the Splay Tree node in the secondary tree.
- The Splay Tree of the Auxiliary Tree are not independent of each other. The parent node of the root node of each Splay Tree should be empty, but in Link Cut Tree, the parent node of the root node of each Splay Tree points to the parent node of the chain in the original tree (that is, the parent node of the point at the top of the chain). This kind of parent link differs from the usual Splay Tree parent link in that the son recognizes the parent, while the parent does not recognize the son, corresponding to an imaginary edge of the original tree. Therefore, every connected block has exactly one point where the parent node is empty.

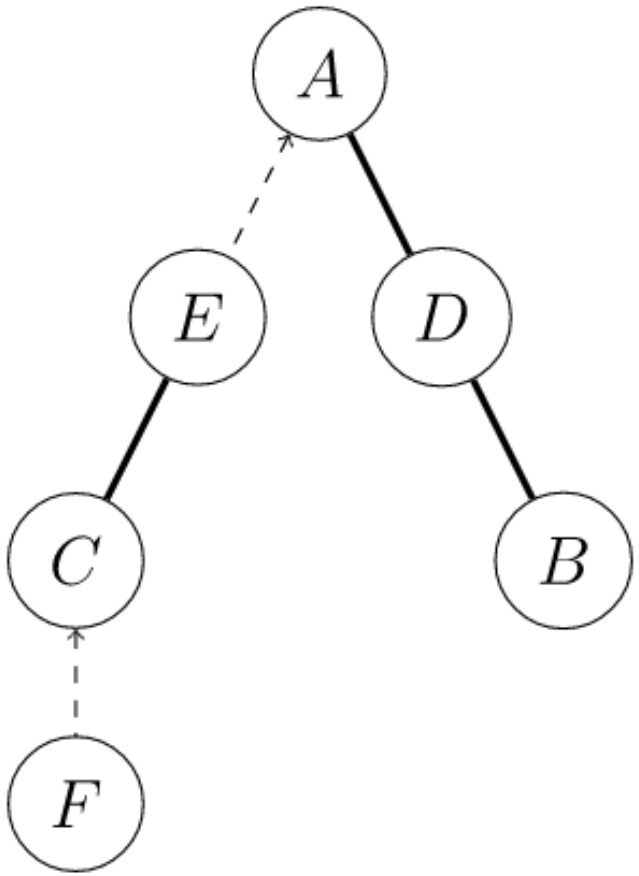
Auxiliary Tree

Now assume we have an original tree. The bold edge is the real edge, and the dashed edge is the imaginary edge.



Auxiliary Tree

As defined above, the structure of the Auxiliary Tree is as follows:



Auxiliary Tree

The relationship between the original tree and Auxiliary Tree

- Real chain in the original tree: Nodes in the Auxiliary Tree are all in a Splay Tree.
- Virtual chain in the original tree: In the Auxiliary Tree, the parent of the Splay Tree of the child points to the parent, but neither of the two sons of the parent points to the child.

Notes:

- The root of the original tree is not equal to the root of the Auxiliary Tree.
- The parent of the original tree is not the same as the parent of the Auxiliary Tree.
- Auxiliary Tree can be arbitrarily changed roots in order to satisfy the properties of Auxiliary Tree and Splay Tree.
- The transformation of virtual chain and real chain can be easily accomplished in the Auxiliary Tree, which is to realize the dynamic maintenance tree chain division.

Link Cut Tree Algorithm

Link Cut Tree Algorithm

Link-Cut Tree is a algorithm that invented by Sleator and Tarjan, to solve the Dynamic Tree Problem. Each operation of this data structure can be implemented in O(logn)

Definition

If in the sub-tree of Node V, the last node to be accessed is in the sub-tree W, where W is the son of V, then w is called Preferred Child of V. If the last node accessed is V itself, it has no Preferred Child. Each point to its Preferred Child is called Preferred Edge. The non-extendable Path connected by Preferred Edge is called Preferred Path.

Link-cut Tree uses Auxiliary Tree to represent each Tree in the forest to be maintained and passes through the Path Parent to connect these Auxiliary trees.

Link Cut Tree Algorithm

Functions in Link Cut Tree (just like Splay Tree):

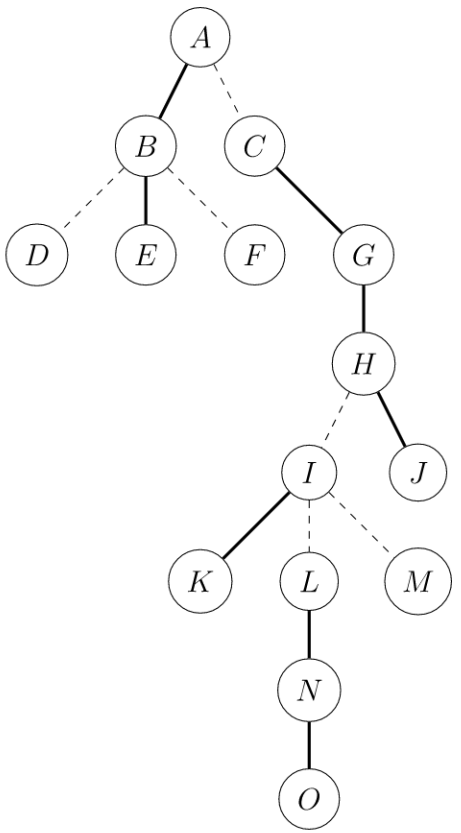
- PushUp(x)
- PushDown(x)
- Get(x): Get parent of x.
- Splay(x): Through the Splay and Rotate operation to make x to be the root of the Splay Tree.
- Rotate(x): Make the x to upper layer.

Access(v)

- Access(v):
  - splay(v). This will make left sub-tree contains all the elements highter than v and right sub-tree contains all elements lower than v.
  - Remove v's preferred child:
    - path − parent(right(v)) ⟵ v
    - right (v) ⟵ null
  - loop until we reach the root:
    - w ⟵ path − parent (v)
    - splay(w)
    - switch w's preferred child
    - path − parent (right (w)) ⟵ w
    - right (w) ⟵ v
    - path − parent (v) ⟵ null
    - v ⟵ w
  - splay(v) for convenience

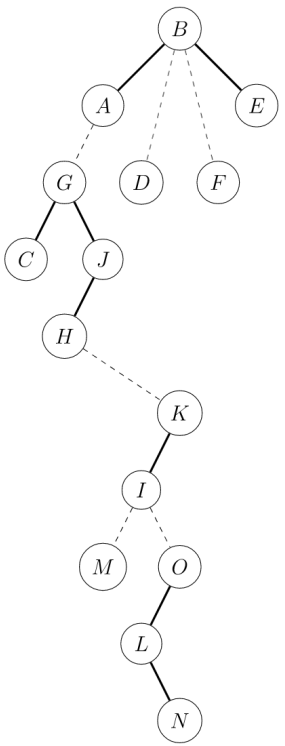
Functions in Link Cut Tree (especially for Link Cut Tree):

- Access(x): Put all nodes from root to x in a real chain so that root to x is a real chain and in the same Splay Tree.
- Assume there is a tree, the solid line is the real chain, the dotted line is the virtual chain.



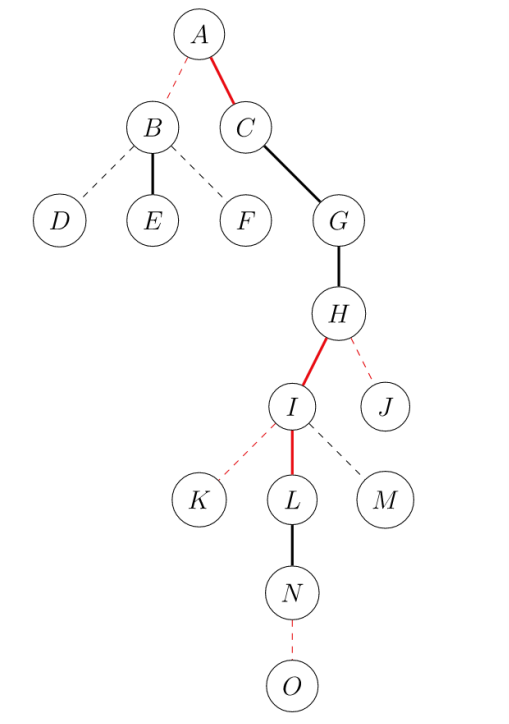
Functions in Link Cut Tree (especially for Link Cut Tree):

- Access(x): Put all nodes from root to x in a real chain so that root to x is a real chain and in the same Splay Tree.
- The Auxiliary Tree of this tree may like this:(different composition way may lead to different structure of Link Cut Tree)



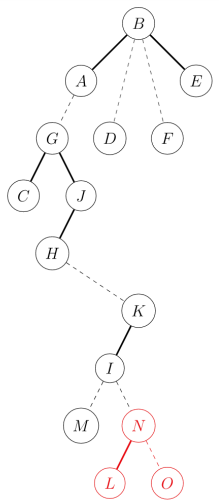
Functions in Link Cut Tree (especially for Link Cut Tree):

- Access(x): Put all nodes from root to x in a real chain so that root to x is a real chain and in the same Splay Tree.
- Now using Access(N)to make path from A to N to be the real chain, make it be the Splay Tree.



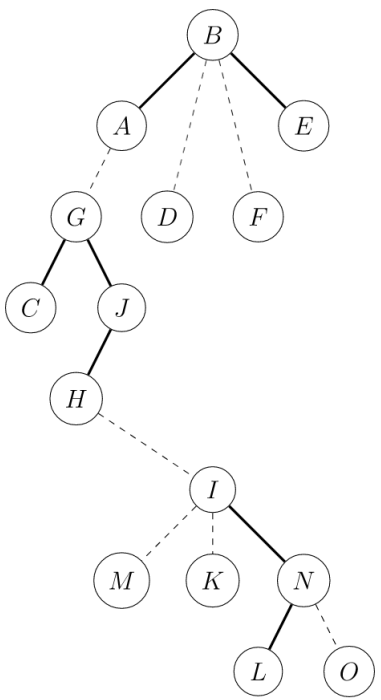
Access(x):

- The realization is updating Splay Tree from the bottom to top by progressively.
- First we need to spin N to the root of the current Splay Tree.
- In order to preserve the Auxiliary Tree feathers, the real chain from N to O need to be changed to virtual chain.
- We can unilaterally change the son of N to Null due to the nature of paternity.
- So the original Auxiliary Tree will be changed to the image below.



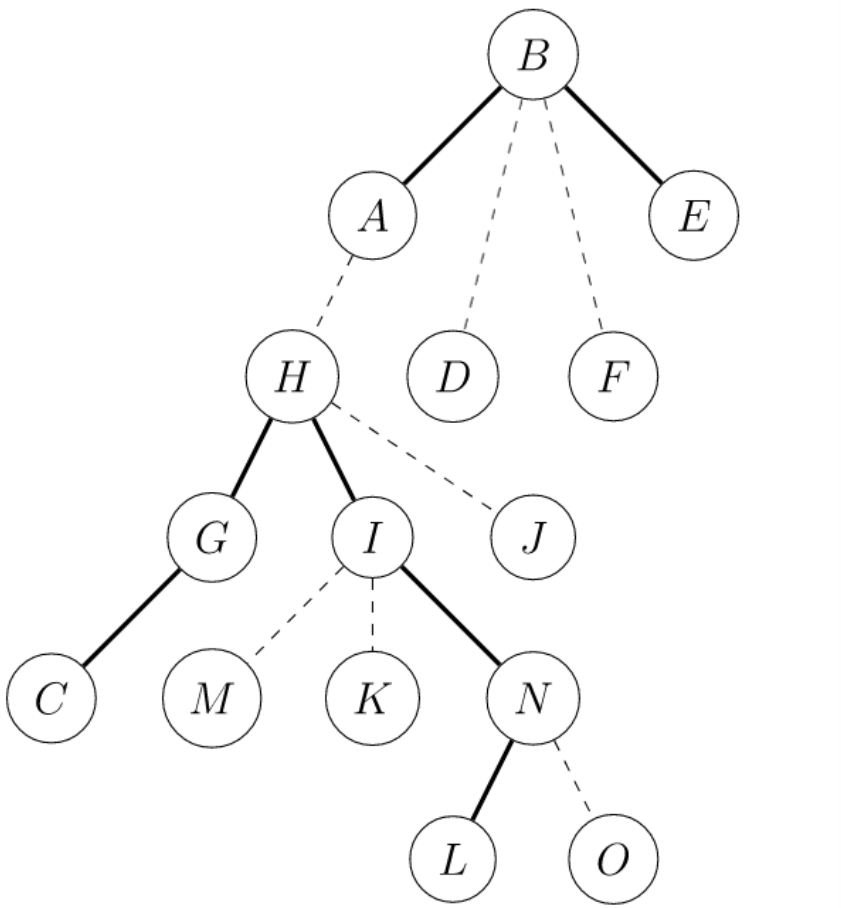
Access(x):

- Next, we rotate parent I of N to the Splay Tree root of I.
- We remove the real chain I – K, and then we point the right child of I to K, and we get the Splay Tree of I – L.



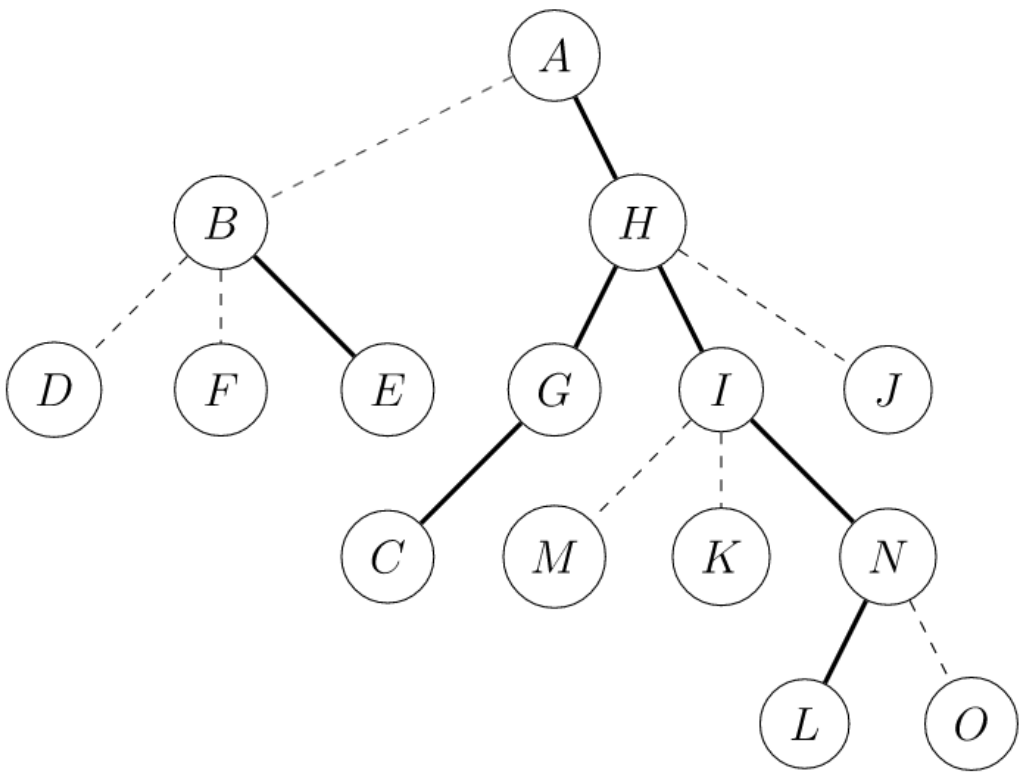
Access(x):

- Next, as the parent of I points to H, we rotate H to the root of the Splay Tree and set right child of H to I.
- And then the tree looks like this.



Access(x):

- Similarly, we Splay(A), and pint the right child of A to H.
- Then we got this Auxiliary Tree as below, and find that the entire path of A – N is already in the same Splay.



Link Cut Tree Algorithm

We find Access(x) is very easy, with just a few steps:

- Roll the current node to the root.
- Replace the son with the previous node.
- Updates information about the current point.
- Change the current point to the current point's parent and continue.

Link Cut Tree Algorithm

We find Access(x) is very easy, with just a few steps:

- Roll the current node to the root.
- Replace the son with the previous node.
- Updates information about the current point.
- Change the current point to the current point's parent and continue.

There is a return value for the Access(x), which is equivalent to the number of the virtual chain's parent node when the last virtual-real chain transformation was performed. This value has two meanings:

- For two consecutive Access operations, the return value of the second Access operation is equal to the LCA of the two nodes.
- Represents the root of the Splay Tree where the chain from x to the root. The node must have been rotated to the root node and the parent must be empty.

Link Cut Tree Algorithm

isRoot(x) Determine x is the root of the tree or not.

- Cause Link Cut Tree has the feather that if a child is not a real child then his parent cannot find it, if a node is neither its parent's left child nor its parent's right child, it will be the root of the current Splay Tree.

update(x) After an Access operation, the information is recursively pushed down from top to bottom.

- Just make pushDown operation from top to bottom.

fix(x, v) Change the weight of x to v.

- Just access(x), then splay(x) and make the weight of x be v, pushUp(x).

makeRoot(x) Make x to be the root of the Tree.

- makeRoot() is no less important than Access(), when we need to maintain path information, the path depth will certainly not increase strictly. According to the feather of Auxiliary Tree, such path cannot appear in a Splay Tree. So we need makeRoot().
- The effect of makeRoot() is to make the specified node the root of the original tree. If the return value of Access(x) is y, then the path from x to the current root constitutes a Splay Tree, and the root of this Splay is y.

Link Cut Tree Algorithm

makeRoot(x) Make x to be the root of the Tree.

- Consider representing the tree as a directed graph, giving each chain a direction from child to father, it is easy to find that swapping roots is equivalent to reversing all the chains of the path to the root.
- So just flip the path x to the current root.
- Since y is the root of the Splay Tree represented by the path from x to the current root, the Splay Tree with y as the root can be flipped between sections.

Link Cut Tree Algorithm

link(x, y) Have an edge between x and y.

- Link two nodes is very easy, first makeRoot(x), then let the parent of x point to y. Obviously, this operation cannot take place in the same tree, so it should be judge first.

split(x, y) Extract the path between x and y, make it convenient to do interval operation.

- Just take out a Splay Tree to maintain the path between x and y.
- First makeRoot(x), then Access(y). If y need to be root, just Splay(y).

Link Cut Tree Algorithm

cut(x, y) Remove the edge between x and y. It must meet the following three conditions:

- x and y are connected.
- The path between x and y have no other chain.
- x don't have the right child.

find(x) Find the number of the root node of the tree in which x located.

- find() means to find the root of current Auxiliary Tree.
- First access(p), then Splay(p) will make the root be the smallest. Go all the way to the left child and pushDown until there is no left child.
- Attention that the node splay corresponding to the queried answer should be added after each query to ensure the complexity.

Notes:

- Remember pushUp and pushDown.
- The rotate in Link Cut Tree should make if(...) first.
- Splay in Link Cut Tree is rotating to the root, not to the child.
- When creating a tree, all edges can be virtual by default and only the parent relationship can be specified.

Time Complexity Analysis for Link Cut Tree

- It can easily find the amortized time complexity is at most O(log n) for all operations except ACCESS.
- In the analysis of ACCESS operation, we divide the ACCESS time into two parts. In the first part, we prove that the time of switching Preferred Child is amortized O(log n). In the second part we prove that the total time of all Splay operations in ACCESS is amortized O(log n).

Heavy-Light Decomposition

Heavy-Light Decomposition is an analytical technique applicable to any tree. It divides the edges of the tree into two types, one of which is either Heavy or Light.

Let size(i) be the number of nodes in the tree whose root is i and u be the largest son of size(u) of v's sons. We call the edge (v, u) heavy and the edge (v, w) from v to its other son w light. If a node has sons, there is a single heavy edge that starts from it. According to this definition, it can be obtained directly:

- Property 3.1 If u is a son of v and (v, u) is a light edge, then

$$\text{size(u)} < \text{size(v)}/2$$

Proof: Assuming $\text{size(u)} \geq \text{size(v)}/2$, since (v, u) is a light edge, there exists a heavy edge (v, w) from v. Depending on the heavy edge definition,$\text{size(v)} \geq 1 + \text{size(w)} + \text{size(u)} \geq 2\text{size(u)} + 1 \geq 1 + \text{size(v)}$. Which is Contradiction.

Heavy-Light Decomposition

Let size(i) be the number of nodes in the tree whose root is i and u be the largest son of size(u) of v's sons. We call the edge (v, u) heavy and the edge (v, w) from v to its other son w light. If a node has sons, there is a single heavy edge that starts from it. According to this definition, it can be obtained directly:

Let light-depth(v) be the number of light edges passing from v to the root, then

Lemma

Lemma 3.1

$$light - depth(v) \leq logn$$

Proof of Lemma 3.1

Lemma

Lemma 3.1

$$light - depth(v) \leq logn$$

Proof: According to property 3.1, if a light edge is passed, the number of nodes in the current sub-tree is at most half of the original one. The number of nodes in the original sub-tree is n, and the sub-tree of v is at least 1, so it has passed log n light edges at most.

Proof of the number of changes of Preferred Child

In order to calculate the number of changes of the Preferred Child, we first perform Heavy-Light Decomposition on the tree T.

Except the disappearance of the Preferred Child of v, every change of the Preferred Child will inevitably generate a new Preferred Edge. So We just need to count the number of newly created Preferred Edges.

Proof of the number of changes of Preferred Child

According to Lemma 3.1, at most log n light Preferred Edge will be generated during ACCESS operation.We know the number of times each heavy edge becomes Preferred Edge is at most the number of times the heavy edge changes from Preferred Edge to normal edge plus the total number of edges.And a heavy edge changes from Preferred Edge to normal edge must accompany with a light edge changes to Preferred Edge, so the the number of times each heavy edge becomes Preferred Edge is at most the number of times the light edge changes to Preferred Edge plus the total number of edges. So, on average, the number of times that the heavy edges in each ACCESS operation become Preferred Edge is O(log n).

To sum up, the number of changes of Preferred Child are evenly amortized O(log n).

Proof of the amortization of the total time of Splay operation

We commonly use potential energy analysis method for the time complexity analysis of Splay operation.

Assign a positive weight w(u) to each node u in the Splay Tree, and let s(u) be the sum of the weights of the nodes in the sub-tree of u.Define the potential energy function $\Phi=\sum_u \lg s(u)$.

Assuming c_i is the actual cost of i-th operation, then we can describe the average cost of i-th operation as $c_i + \Phi_i - \Phi_{i-1}$. Then the average cost of n operations:

$$a_n = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0$$

Proof of the amortization of the total time of Splay operation

Let $\Phi_0 = 0$,then $a_n = \sum_{i=1}^{n} c_i + \Phi_n \geq \overline{cost}$
Let $\mu(x) = \log_2 s(x)$. Then $\Phi_i(x) = \sum_{i=1}^{n} \mu_i(x)$

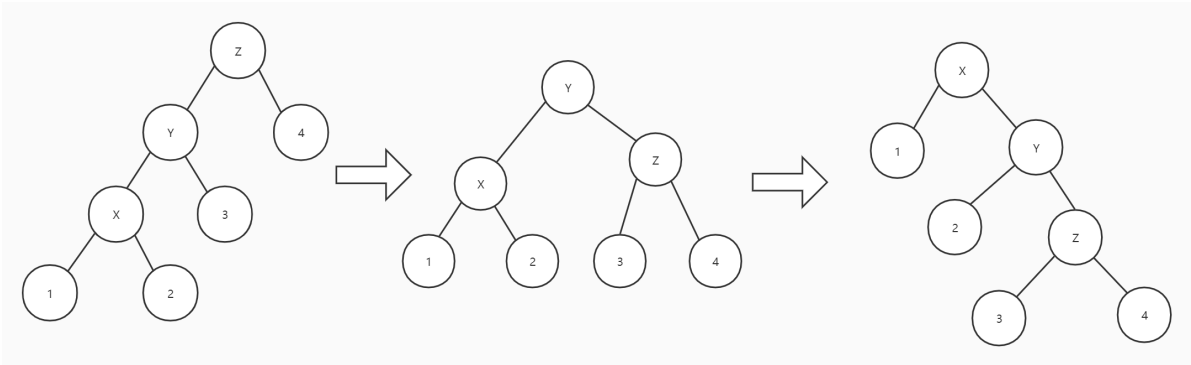
Lemma

Lemma 3.2 zig-zig, zag-zag average time spent in operation Splay(v)

$$\overline{cost} \leq 3(\lg s'(v) - \lg s(v))$$

Proof of Lemma 3.2

Let's see an example as following graph:



$$a_n = 2 + \mu(x') + \mu(y') + \mu(z') - \mu(x) - \mu(y) - \mu(z)$$

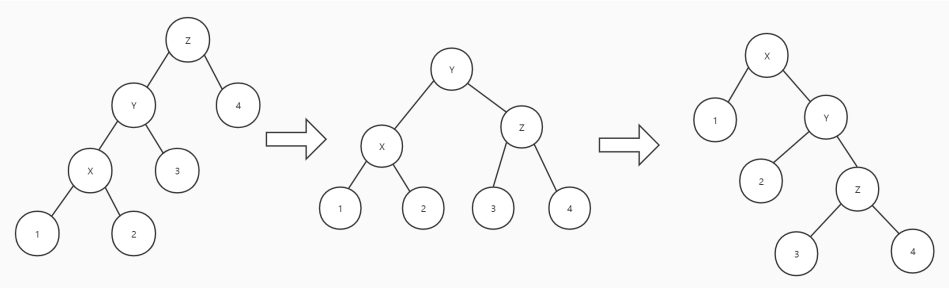
It's obvious that $\mu(x') = \mu(z)$

Thus $a_n = 2 + \mu(y') + \mu(z') - \mu(x) - \mu(y)$

It's obvious that $\mu(y) > \mu(x)$ and $\mu(x') \geq \mu(y')$

Thus $a_n \leq \mu(y') + \mu(z') - 2\mu(x) + 2 \leq \mu(x') + \mu(z') - 2\mu(x) + 2$

Proof of Lemma 3.2



$$s(x) + s(z') = 2 + s(1) + s(2) + s(3) + s(4) \leq s(x')$$

Thus $s(x) \cdot s(z') \leq \frac{1}{4} s^2(x')$

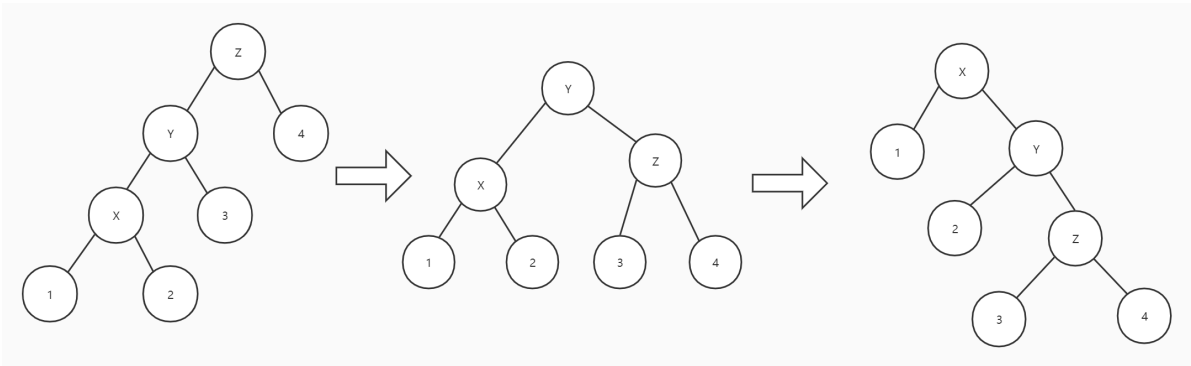
Thus $\mu(x) + \mu(z') - 2\mu(x') \leq -2$

That's to say, $-2 - \mu(x) - \mu(z') + 2\mu(x') \geq 0$

Add it to a_n, we can get

$$a_n \leq 3\mu(x') - 3\mu(x)$$

Proof of Lemma 3.2



Thus, we can prove that

$$\overline{\text{cost}} \leq 3\left(\lg s'(v) - \lg s(v)\right)$$

Proof of the amortization of the total time of Splay operation

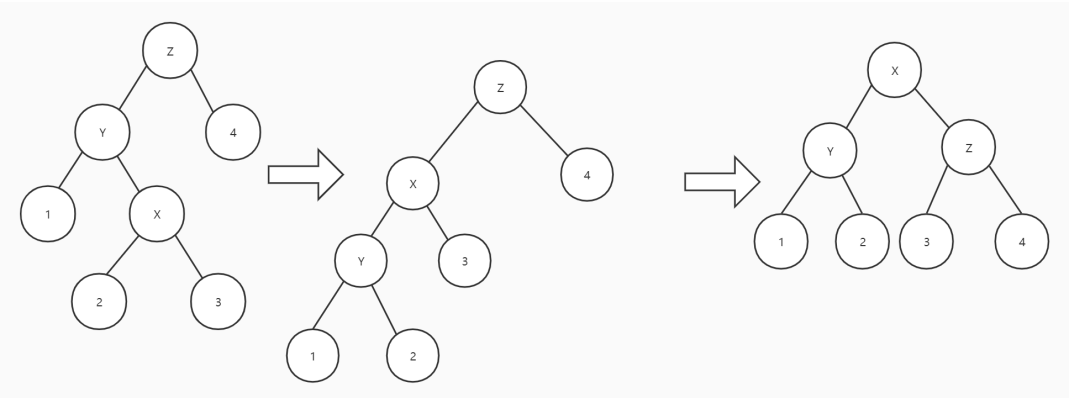
Lemma

Lemma 3.3 zig-zag, zag-zig average time spent in operation Splay(v)

$$\overline{\text{cost}} \leq 2\left(\lg s'(v) - \lg s(v)\right) \leq 3\left(\lg s'(v) - \lg s(v)\right)$$

Proof of Lemma 3.3

Let's see an example as following graph:



$$a_n = 2 + \mu(x') + \mu(y') + \mu(z') - \mu(x) - \mu(y) - \mu(z)$$

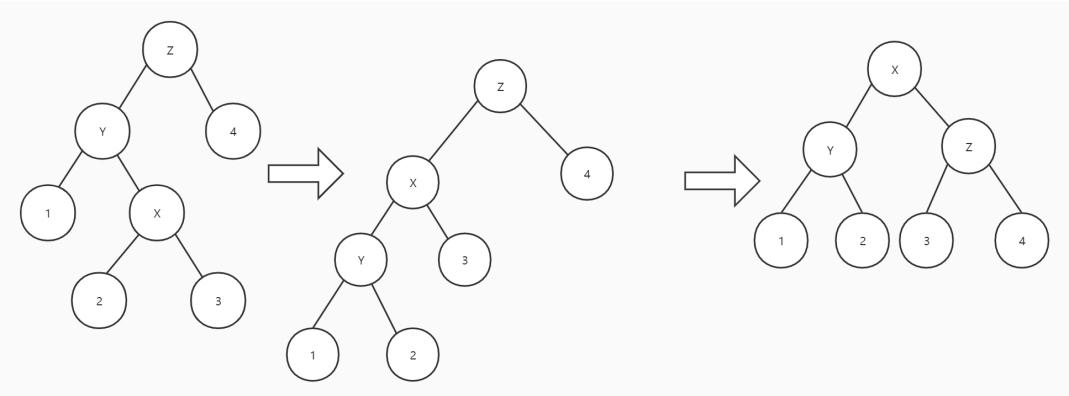
Because of $\mu(x') = \mu(z)$

$$a_n = 2 + \mu(y') + \mu(z') - \mu(x) - \mu(y)$$

Similar to Lemma 2, we can get $a_n \leq 2 + \mu(y') + \mu(z') - 2\mu(x)$

Similar to Lemma 2, we can get $-2 - \mu(y') - \mu(z') + 2\mu(x') \geq 0$

Proof of Lemma 3.3



Thus

$$a_n \leq 2\mu(x') - 2\mu(x) \leq 3\mu(x') - 3\mu(x)$$

Thus, we can prove that

$$\overline{cost} \leq 2(\lg s'(v) - \lg s(v)) \leq 3(\lg s'(v) - \lg s(v))$$

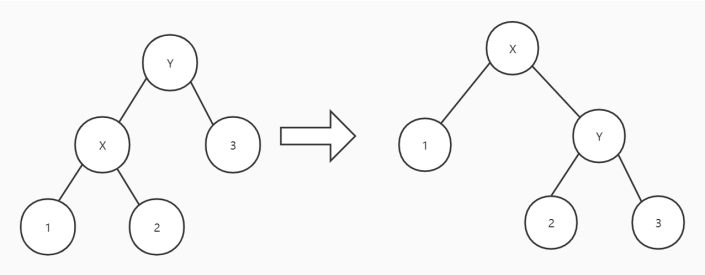
Proof of the amortization of the total time of Splay operation

Lemma

Lemma 3.4 zig, zag average time spent in operation Splay(v)

$$\overline{\text{cost}} \leq \lg s'(v) - \lg s(v) + 1 \leq 3(\lg s'(v) - \lg s(v)) + 1$$

Proof of Lemma 3.4



$$a_n = 1 + \mu(x') + \mu(y') - \mu(x) - \mu(y)$$

Because $\mu(x') = \mu(y)$ and $\mu(x') \geq \mu(y')$

We can get

$$a_n = 1 + \mu(y') - \mu(x) \leq 1 + \mu(x') - \mu(x) \leq 1 + 3\mu(x') - 3\mu(x)$$

Thus

$$\overline{cost} \leq \lg s'(v) - \lg s(v) + 1 \leq 3(\lg s'(v) - \lg s(v)) + 1$$

Proof of the amortization of the total time of Splay operation

Lemma

Lemma 3.5 (Access Theorem) The average time spent Splay(v)

$$\overline{\text{cost}} \leq 3\left(\lg s'(v) - \lg s(v)\right) + 1$$

Proof of Lemma 3.5

This can be proved easily by Lemma2, Lemma 3 and Lemma4.

In this problem, since an ACCESS operation will perform multiple (equalized $O(\log n)$ times) SPLAY operations, we should find a suitable weight function instead of simply let w(u) = 1. Let's imagine the edges in Auxiliary Tree as real edges, and think of Path Parent is imagined as a virtual edge.The total number of nodes of a sub-tree is equal to 1 add the total number of nodes of all its sub-trees (including the sub-trees connected by the virtual edge). Let w(u) equal to 1 add the total number of nodes in the sub-tree with u as the Path Parent, then s(u) is the total number of nodes in the sub-tree of u.

Proof of Lemma 3.5

Assuming in the process of ACCESS(v), we respectively enter $v_0 = v$, $v_1 = \text{path} - \text{parent}[v_0]$,..., $v_k = \text{path} - \text{parent}[v_{k-1}]$ to SPLAY operation. Then, the average time of ACCESS is

$$\begin{aligned}
\overline{\text{cost}} &\leq \sum_{i=0}^{k} 3\left(\lg s'(v_i) - \lg s(v_i)\right) + 1 \\
&\leq 3\left[\sum_{i=1}^{k}\left(\lg s'(v_i) - \lg s'(v_{i-1})\right) + \lg s'(v_0) - \lg s(v_0)\right] + k \\
&= 3\left(\lg s'(v_k) - \lg s(v_0)\right) + k \\
&\leq 3\lg n + \text{ the number of changes of Preferred Child}
\end{aligned}$$

In 5.2, we have proved that the number of changes of Preferred Child equalized $O(\log n)$ times, so the amortized time complexity of ACCESS is $O(\log n)$.

In the process of a single rotation, not only must the local form of the splay tree be modified, but also the maxcost attribute of the vertex must be maintained. At the same time, because of the usual writing of splay tree, it is necessary to check whether the node is the root node or the father. This also took a lot of time. Thus there exists a concise and efficient way to write SPLAY, ROTATE, UPDATE as follows

Optimize of Link Cut Tree Algorithm

UPDATE

- UPDATE(node):
  - k $\longleftarrow$ cost [node]
  - x $\longleftarrow$ maxcost [child [node] [0]]
  - y $\longleftarrow$ maxcost [child [node] [1]]
  - if x $<$ y then
    - x $\longleftarrow$ y
  - end if
  - if k $<$ x then
    - k $\longleftarrow$ x
  - end if
  - maxcost [node] $\longleftarrow$ k

ROTATE

- ROTATE(node):
  - p $\longleftarrow$ parent [node]; y $\longleftarrow$ nodetype [p]; tmp $\longleftarrow$ parent [p]
  - parent [node] $\longleftarrow$ tmp; nodetype [node] $\longleftarrow$ y
  - if y $\neq$ 2 then child [tmp] [y] $\longleftarrow$ node
  - y $\longleftarrow$ 1 $-$ x
  - tmp $\longleftarrow$ child [node] [y]
  - child [p] [x] $\longleftarrow$ tmp
  - parent [tmp] $\longleftarrow$ p
  - nodetype [tmp] $\longleftarrow$ x
  - parent [p] $\longleftarrow$ node
  - nodetype [p] $\longleftarrow$ y
  - child [node] [y] $\longleftarrow$ p
  - UPDATE(p)

Optimize of Link Cut Tree Algorithm

SPLAY

- SPLAY(node):
  - repeat
    - a ⟵ nodetype [node]
    - if a = 2 then break
    - p ⟵ parent [node];
    - b ⟵ nodetype [p]
    - if a = b then ROTATE(p,a)
    - else ROTATE(node,a)
    - if b=2 then break
    - ROTATE(node,b)
  - until FALSE
  - UPDATE(node)

Optimize of Link Cut Tree Algorithm

In the process of SPLAY (node), the usual wording of SPLAY will perform many times of UPDATE on some points. In fact, this is unnecessary. We only need to update the parent node of the rotated node in the ROTATE operation. And at the end of the SPLAY, perform an UPDATE on the node. Since UPDATE is a time-consuming operation, this optimization removes about half of the redundant UPDATE operations, so the performance of the program has been greatly improved.

Applications of Link Cut Tree Algorithm

Maintain tree chain information

- Through split operation can extract the path between x and y to a Splay Tree, and the modification and statistics of tree chain information are transformed into the operation of Balanced Tree, which gives Link Cut Tree an advantage in maintaining tree chain information.
- In addition, the on-chain dichotomy with the aid of Splay Tree has one less log complexity than the heavy or heavy chain splitting.
- To change the tree path (u,v), split(u,v) first. Everything else is done on Splay Tree.

Maintain sub-tree information

- Link Cut Tree is not good at maintaining sub-tree information.

Applications of Link Cut Tree Algorithm

Maintenance connectivity property

- Through find operation can maintain the connectivity property of a dynamic forest. If find(x) == find(y), then it means that x, y are in the same tree and they are connective.

Maintenance edge weight

- Link Cut Tree cannot deal with edge weights directly, so virtual points are needed to maintain edge weights.
- Specifically, each edge establishes a virtual point, virtual point with two edges to connect its original starting point and end point.

Conclusion

Link-Cut Tree only adds a few lines of code on the basis of the existing Splay, which can perfectly solve the problems related to Dynamic Trees. With the powerful tool of Link-Cut Tree, many tree or ring-plus extroverted trees, and even dynamic problems on cacti can be easily solved.

In ACCESS(v), if there are only normal edge, the operation will jump from vertex v to root, which will take $O(n)$. But the time complexity of Link-Cut Tree is $O(n \lg n)$, which was proved by Tarjan.

References


D. D. Sleator, R. E. Targan, A Data Structure for Dynamic Trees, Jornal. Comput. Syst. Sci., 28(3):362-391, 1983.


R. E. Tarjan, Data Structure and Network Algorithms, Society for Industrial and Applied Mathmatics, 1984.