Program Name/Purpose- Tarjan's off-line lowest common ancestors algorithm

Project Category- Advanced Data Structures, Trees

Programming Paradigm Used- Offline Processing , Colouring, Disjoint Set Union by rank and path compression

Prerequisites-

a) LCA basics- http://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/
b) Disjoint Set Union by Rank and Path Compression- http://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/

Data Structure Used- Disjoint Set Data Structure, Tree data structure

What is the program about?-

Suppose we are given a tree(can be extended to a DAG) and we have many queries of form- ***LCA(u,v),*** i.e.- find LCA of nodes- *'**u**' and '**v**'*.

How will we perform the above queries efficiently.
We can perform those queries in **O(N+QlogN)** time, where **O(N)** time for pre-processing and **O(log N)** for answering the queries, where **N**=number of nodes and **Q**=number of queries to be answered.

See this for implementation- http://www.geeksforgeeks.org/find-lca-in-binary-tree-using-rmq/

Can we do better than this? Can we do in linear(almost) time? Yes.

The article presents an **offline algorithm** which performs those queries in **approximately** - **O(N+Q)** time. Although, this is not ***exactly linear***, as there is an **Inverse Ackermann function** involved in the time complexity analysis. For more details on Inverse Ackermann function see- http://www.gabrielnivasch.org/fun/inverse-ackermann

Just as a summary, we can say that-

" The Inverse Ackermann Function remains less than **4, for any value of input size that can be written in physical inverse.** Thus, we consider this as almost linear. "

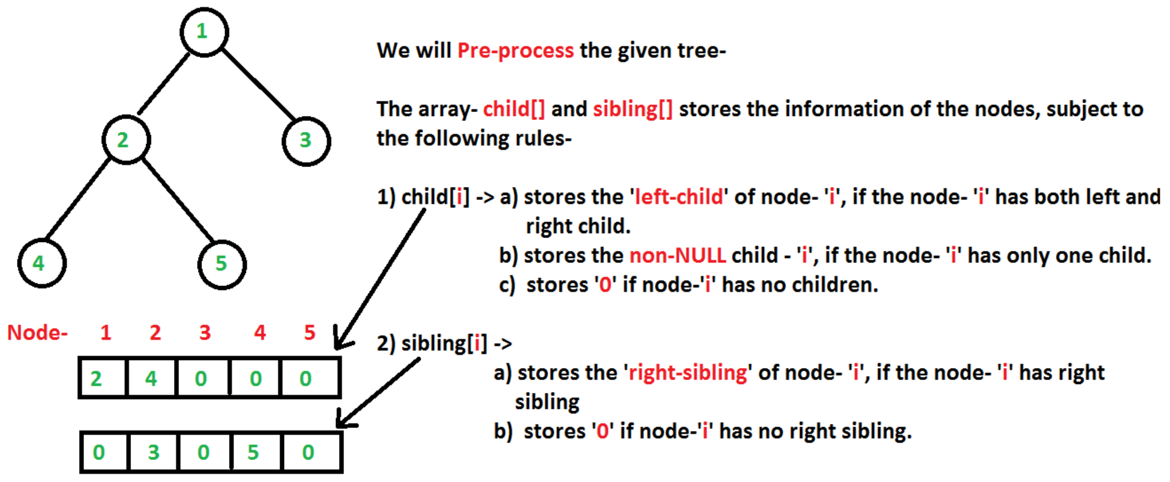
What is an offline algorithm?

When you process input at the time they are given that approach is called online and when you store all input and process that later at once , that comes under offline category.

A famous offline algorithm is – Mo's algorithm[http://www.geeksforgeeks.org/mos-algorithm-query-square-root-decomposition-set-1-introduction/]

Explanation-

We will consider the input tree as shown below. We will **Pre-Process** the tree and fill two arrays- **child[] and sibling[]** according to the below explanation-



We will Pre-process the given tree-

The array- child[] and sibling[] stores the information of the nodes, subject to the following rules-

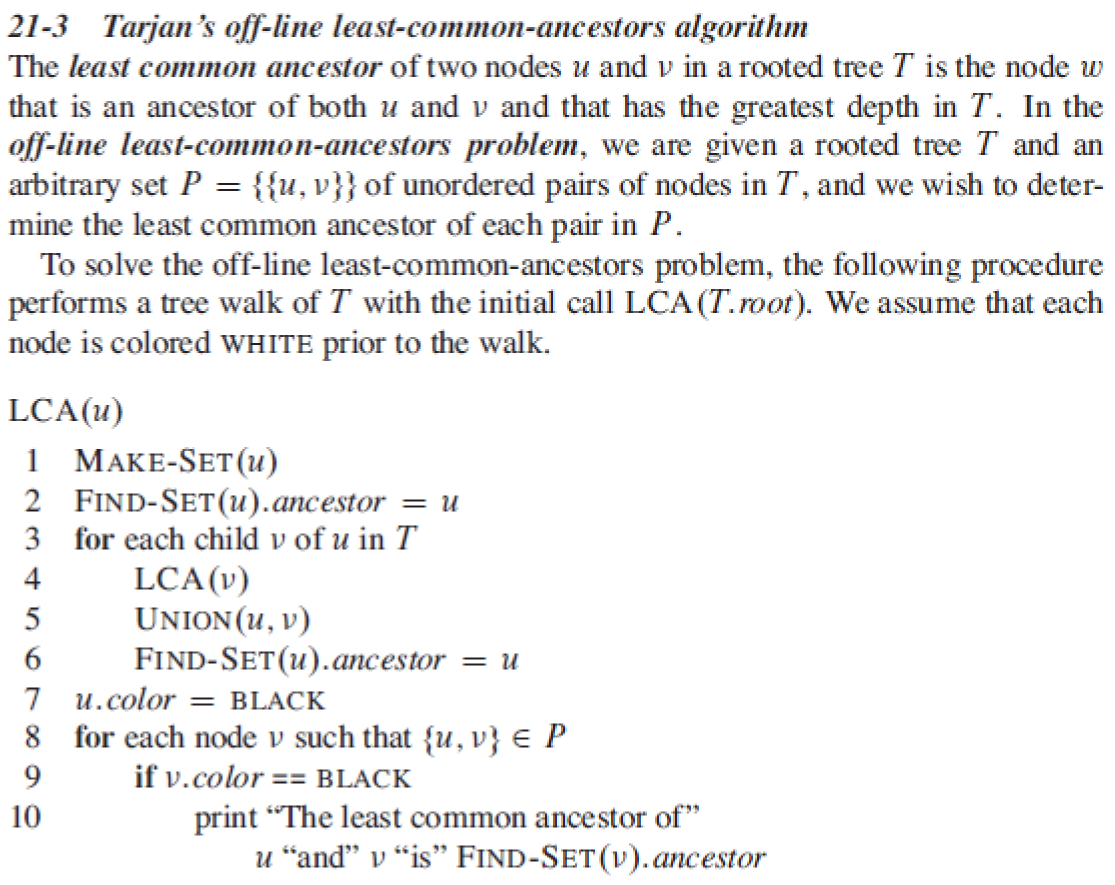
1) child[i] -> a) stores the 'left-child' of node- 'i', if the node- 'i' has both left and right child.
b) stores the non-NULL child - 'i', if the node- 'i' has only one child.
c) stores '0' if node-'i' has no children.

2) sibling[i] ->
a) stores the 'right-sibling' of node- 'i', if the node- 'i' has right sibling
b) stores '0' if node-'i' has no right sibling.

Let we want to process these queries- **LCA(5,4), LCA(1,3), LCA(2,3)**

Now, after **pre-processing,** we will perform a **LCA walk** starting from the **root** of the tree(here-node '1'). But **prior** to the LCA walk**,** we will firstly **colour** all the nodes with **WHITE**. During the whole LCA walk, we will use three disjoint set union functions- **makeSet(), findSet(), unionSet().** These functions uses the technique of union by rank and path compression to improve the running time. During the LCA walk, our queries gets processed and outputted (in a random order). After the LCA walk of the whole tree, all the nodes gets coloured **BLACK.**

Tarjan Offline LCA Algorithm steps-(*Reference- CLRS, Section-21-3, Pg 584, 2nd /3rd edition*)



21-3 *Tarjan's off-line least-common-ancestors algorithm*

The ***least common ancestor*** of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T. In the ***off-line least-common-ancestors problem***, we are given a rooted tree T and an arbitrary set $P = \{\{u, v\}\}$ of unordered pairs of nodes in T, and we wish to determine the least common ancestor of each pair in P.

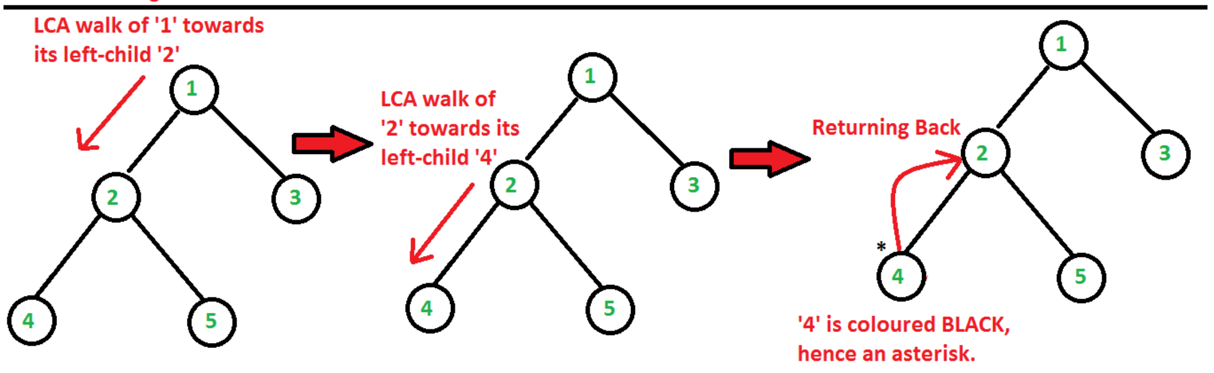
To solve the off-line least-common-ancestors problem, the following procedure performs a tree walk of T with the initial call LCA($T.root$). We assume that each node is colored WHITE prior to the walk.

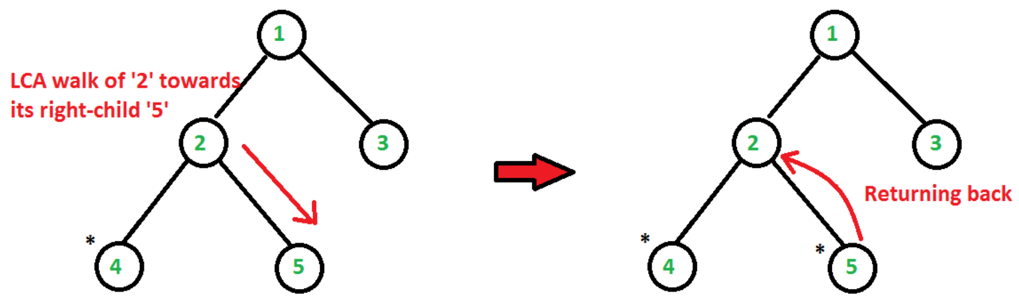
```
LCA(u)
1   MAKE-SET(u)
2   FIND-SET(u).ancestor = u
3   for each child v of u in T
4       LCA(v)
5       UNION(u, v)
6       FIND-SET(u).ancestor = u
7   u.color = BLACK
8   for each node v such that {u, v} ∈ P
9       if v.color == BLACK
10          print "The least common ancestor of"
                u "and" v "is" FIND-SET(v).ancestor
```

Note-The above algorithm is an offline algorithm. Hence, the queries may not be outputted in the original order. We can easily sort them according to the input order.

Since, this is a **recursive function**, so we will explain this algorithm step-by-step on out input tree using these pictures. The below pictures clearly depicts all the steps happening. The red arrow shows the direction of travel of our recursive function.

Initially, prior to the LCA walk, all nodes are coloured WHITE, and during the LCA walk the nodes gets coloured BLACK. We denote a node to be coloured BLACK, by putting an asterisk '*' beside that node, and a WHITE node doesn't has an asterisk '*' beside it. Note that LCA walk of a node consists of two part- towards its left-child and towards its right-child.


LCA walk of '1' towards its left-child '2'
1
2
3
4
5
LCA walk of '2' towards its left-child '4'
1
2
3
4
5
Returning Back
1
2
3
*
4
5
'4' is coloured BLACK, hence an asterisk.



Steps happening-

1) Union of 2 and 4
2) ancestor[2]=2

2
/
4

Steps happening-

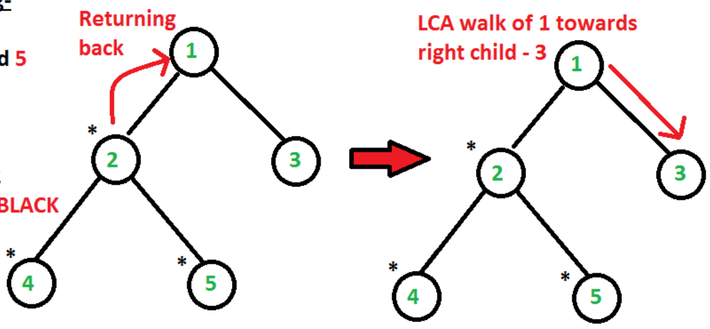
1) 5 is coloured BLACK
2) LCA(5,4) = ancestor[find(4)] = ancestor[2] = 2

Steps Happening-

1) Union of 2 and 5

2
4 5

2) ancestor[2]=2
3) 2 is coloured BLACK

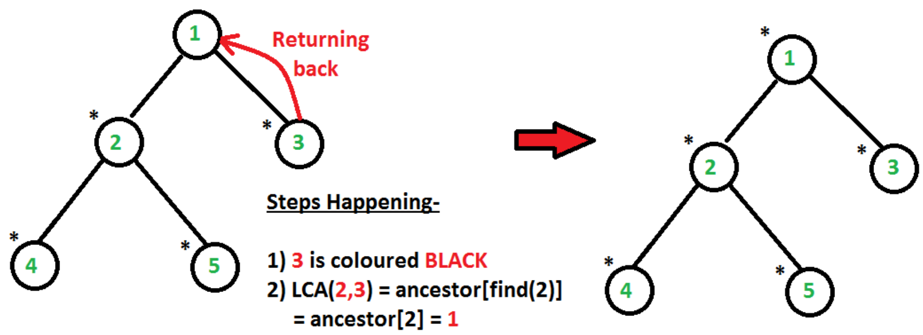


Steps Happening-

1) Union of 1 and 2

2
4 5 1

2) ancestor[2] = 1



Steps Happening-

1) 3 is coloured BLACK
2) LCA(2,3) = ancestor[find(2)]
= ancestor[2] = 1

Steps Happening-

1) Union of 1 and 3

2
4 5 1
3

2) ancestor[2] = 1
3) 1 is coloured BLACK
4) LCA(1,3) = ancestor [find(3)] = ancestor[2] = 1

As, we can clearly see from the above pictures, the queries are outputted in the following order- **LCA(5,4), LCA(2,3), LCA(1,3)** which is not in the same order as the input(**LCA(5,4), LCA(1,3), LCA(2,3)**).

Time Complexity-

Super-linear, i.e- barely slower than **linear. O(N+Q) time,** where **O(N)** time for pre-processing and almost **O(1) time** for answering the queries, where **N**=number of nodes and **Q**=number of queries to be answered.

Space Complexity-

We will use a many arrays- **parent[], rank[], ancestor[]** which will be used in Disjoint Set Union Operations each with the size equal to the number of nodes. We will also use the arrays- **child[], sibling[], color[]** which is useful in this offline algorithm. Hence, we use **O(N)**

For convenience, all these arrays are put up in a structure- **struct subset** to hold these arrays.

References-

https://en.wikipedia.org/wiki/Tarjan%27s_off-line_lowest_common_ancestors_algorithm

CLRS, Section-21-3, Pg 584, 2nd /3rd edition

http://wcipeg.com/wiki/Lowest_common_ancestor#Offline