

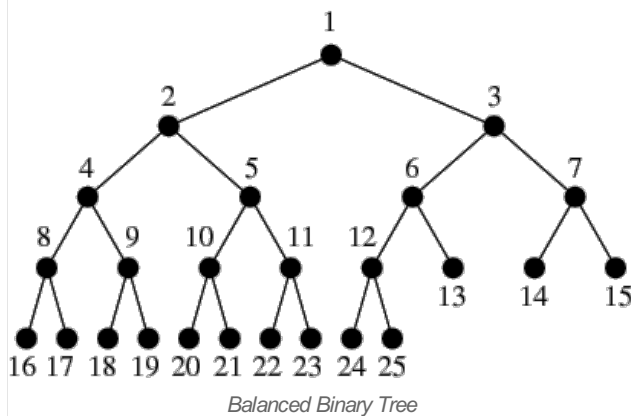


Heavy Light Decomposition

📅 11 April 2014 ✍ anudeep2011 📖 Algorithms, Segment trees 💬 64 Comments 👁 41,209 views

Long post with lot of explanation, targeting novice. If you are aware of how HLD is useful, skip to “Basic Idea”.

Why a Balanced Binary Tree is good?



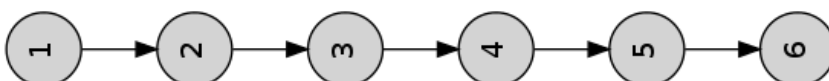
A balanced binary tree with N nodes has a height of $\log N$. This gives us the following properties:

- You need to visit at most $\log N$ nodes to reach **root** node from any other node
- You need to visit at most $2 * \log N$ nodes to reach from any node to any other node in the tree

The \log factor is always good in the world of competitive programming 😊

Now, if a balanced binary tree with N nodes is given, then many queries can be done with $O(\log N)$ complexity. Distance of a path, Maximum/Minimum in a path, Maximum contiguous sum etc etc.

Why a Chain is good?



Subscribe to our Newsletter

SUBSCRIBE!

Categories

[Algorithms](#) (2)
[Data Structures](#) (1)
[Machine Learning](#) (1)
[Segment trees](#) (2)
[SPOJ](#) (1)
[Uncategorized](#) (1)

Recent Posts

[Machine learning everywhere, why not in Competitive programming?](#)
 April 18, 2016

[MO's Algorithm \(Query square root decomposition\)](#)
 December 28, 2014

[When 2 guys talk, its not always about girls/sports ;\)](#)
 July 18, 2014

[Persistent segment trees – Explained with spoj problems](#)
 July 13, 2014

[Heavy Light Decomposition](#)
 April 11, 2014

Recent Comments

ibahadir.altun on [Heavy Light Decomposition](#)

Reinier on [When 2 guys talk, its not always about girls/sports ;\)](#)

Shamit on [Heavy Light Decomposition](#)

A chain is a set of nodes connected one after another. It can be viewed as a simple array of nodes/numbers. We can do many operations on array of elements with $O(\log N)$ complexity using **segment tree / BIT** / other data structures. You can read more about segment trees [here – A tutorial by Utkarsh](#) .

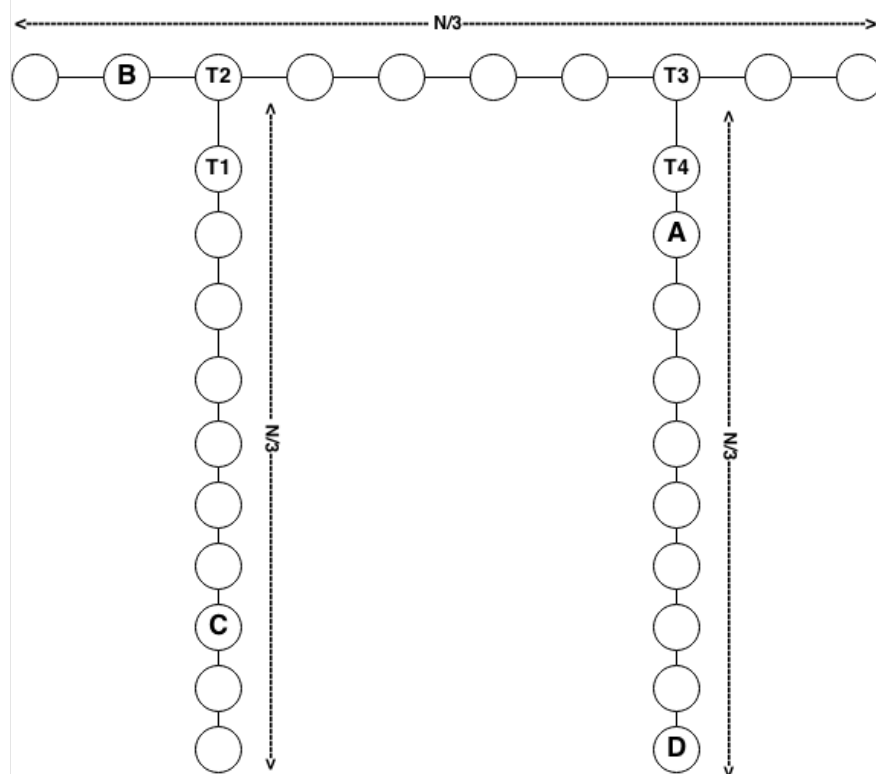
Now, we know that Balanced Binary Trees and arrays are good for computation. We can do a lot of operations with $O(\log N)$ complexity on both the data structures.

Why an Unbalanced Tree is bad?

Height of unbalanced tree can be arbitrary. In the worst case, we have to visit $O(N)$ nodes to move from one node to another node. So Unbalanced trees are not computation friendly. We shall see how we can deal with unbalanced trees.

Consider this example..

Consider the following question: Given **A** and **B**, calculate the sum of all node values on the path from **A** to **B**.



Here are details about the given images

1. The tree in the image has **N** nodes.
2. We need to visit **N/3** nodes to travel from **A** to **D**.
3. We need to visit **>N/3** nodes to travel from **B** to **D**.
4. We need to visit **>N/2** nodes to travel from **C** to **D**.

It is clear that moving from one node to another can be up to $O(N)$ complexity.

This is important: What if we broke the tree in to 3 chains as shown in image below. Then

Prashant Yadav on [Machine learning everywhere, why not in Competitive programming?](#)

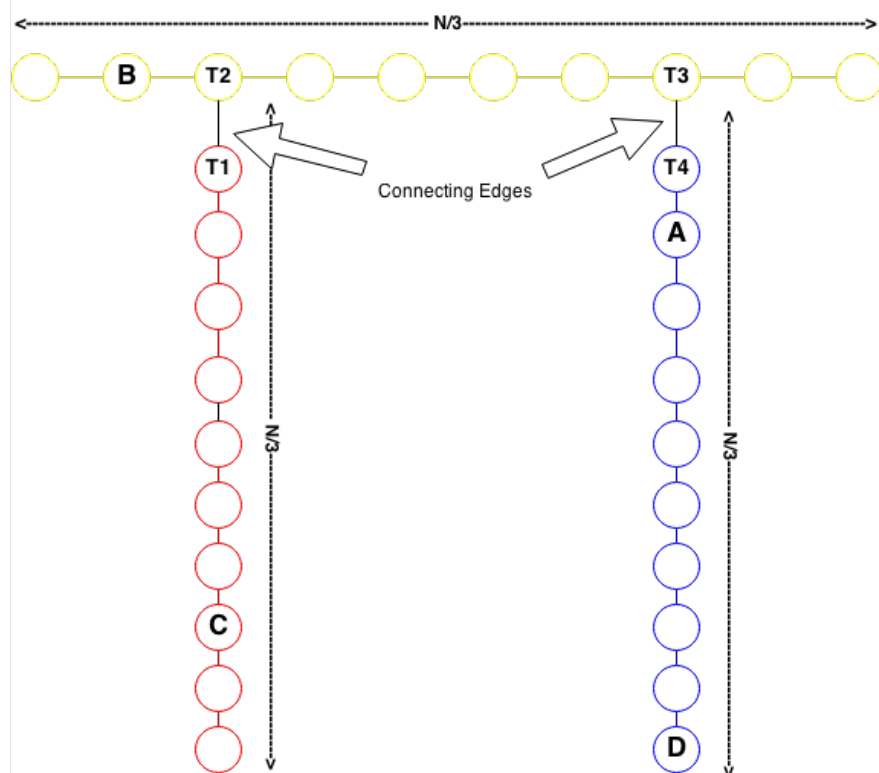
raj454raj on [Machine learning everywhere, why not in Competitive programming?](#)

Contact Me

Ask about something or report me a bug or just say Hi. I will be happy to get back.



we consider each chain as an independent problem. We are dealing with chains so we can use Segment Trees/other data structures that work well on linear list of data.



Here are the details after the trick

1. The tree still has N nodes, but it is **DECOMPOSED** into 3 chains each of size $N/3$. See 3 different colors, each one is a chain.
2. **A** and **D** belong to the same chain. We can get the required sum of node values of path from **A** to **D** in $O(\log N)$ time using segment tree data structure.
3. **B** belongs to yellow chain, and **D** belongs to blue chain. Path from **B** to **D** can be broken as **B** to **T3** and **T4** to **D**. Now we are dealing with 2 cases which are similar to the above case. We can calculate required sum in $O(\log N)$ time for **B** to **T3** and $O(\log N)$ time for **T4** to **D**. Great, we reduced this to $O(\log N)$.
4. **C** belongs to red chain, and **D** belongs to blue chain. Path from **C** to **D** can be broken as **C** to **T1**, **T2** to **T3** and **T4** to **D**. Again we are dealing with 3 cases similar to 2nd case. So we can again do it in $O(\log N)$.

Awesome!! We used concepts of **Decomposition** and **Segment Tree DS**, reduced the query complexity from $O(N)$ to $O(\log N)$. As I said before, competitive programmers always love the **log** factors ☐ ☐

But wait the tree in the example is special, only 2 nodes had degree greater than 2. We did a simple decomposition and achieved better complexity, but in a general tree we need to do something little more complex to get better complexity. And that little more complex decomposition is called **Heavy Light Decomposition**.

Basic Idea

We will divide the tree into **vertex-disjoint chains** (Meaning no two chains has a node in common) in such a way that to move from **any node** in the tree to the **root** node, we will

have to **change at most $\log N$** chains. To put it in another words, the path from **any node** to **root** can be broken into pieces such that the each piece belongs to only one chain, then we will have no more than **$\log N$** pieces.

Let us assume that the above is done, So what?. Now the path from any node **A** to any node **B** can be broken into two paths: **A to LCA(A, B)** and **B to LCA(A, B)**. Details about LCA – [Click Here](#) or [Here](#). So at this point we need to only worry about paths of the following format: **Start at some node and go up the tree** because **A to LCA(A, B)** and **B to LCA(A, B)** are both such paths.

What are we up to till now?

- We assumed that we can break tree into chains such that we will have to **change at most $\log N$** chains to move from any node up the tree to any other node.
- Any path can be broken into two paths such both paths start at some node and move up the tree
- We already know that queries in each chain can be answered with **$O(\log N)$** complexity and there are at most **$\log N$** chains we need to consider per path. So on the whole we have **$O(\log^2 N)$** complexity solution. Great!!

Till now I have explained how **HLD** can be used to reduce complexity. Now we shall see details about how **HLD** actually decomposes the tree.

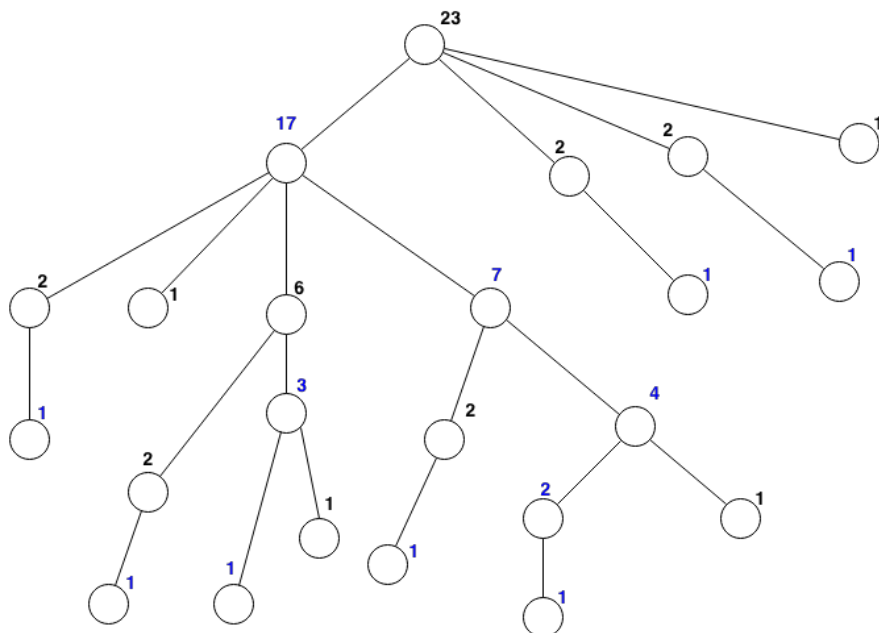
Note : My version of HLD is little different from the standard one, but still everything said above holds.

Terminology

Common tree related terminology can be found [here](#).

Special Child : Among all child nodes of a node, the one with maximum sub-tree size is considered as **Special child**. Each non leaf node has exactly one **Special child**.

Special Edge : For each non-leaf node, the edge connecting the node with its **Special child** is considered as **Special Edge**.



Each node has its sub-tree size written on top.
 Each non-leaf node has exactly one special child whose sub-tree size is colored.
 Colored child is the one with maximum sub-tree size.

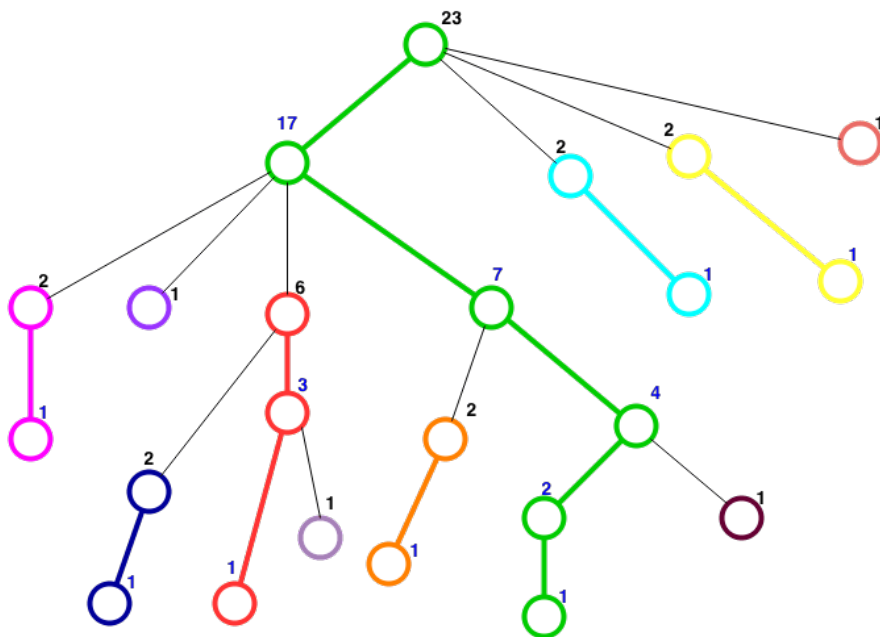
Read the next 3 paras until you clearly understand every line of it, every line makes sense (Hope!). Read it 2 times 3 times 10 times 2 power 10 times .. , until you understand!!

What happens if you go to each node, find the special child and special edge and mark all special edges with green color and other edges are still black? Well, what happens is **HLD**. What would the graph look like then? Colorful yes. Not just colorful. Green edges actually forms vertex disjoint chains and black edges will be the connectors between chains. Let us explore one chain, start at root, move to the special child of root (there is only one special child, so easy pick), then to its special child and so on until you reach a leaf node, what you just traveled is a chain which starts at root node. Let us assume that root node has **m** child nodes. Note that all **m-1** normal child nodes are starting nodes of some other chains.

What happens if you move from a node to a normal child node of it. This is the **most important part**. When you move from a node to any of its normal child, the sub-tree size is at most half the sub-tree size of current node. Consider a node **X** whose sub-tree size is **s** and has **m** child nodes. If **m=1**, then the only child is special child (So there is no case of moving to normal child). For **m>=2**, sub-tree size of any normal child is $\leq s/2$. To prove that, let us talk about the sub-tree size of special child. What is the least sub-tree size possible for special child? Answer is **ceil(s/m)** (what is ceil? [click here](#)). To prove it, let us assume it is less than **ceil(s/m)**. As this child is the one with maximum sub-tree size, all other normal child nodes will be at most as large as special child, **m** child nodes with each less than **ceil(s/m)** will not sum up to **s**, so with this counter-intuition. We have the following: The minimum sub-tree size possible for special child is **ceil(s/m)**. This being said, the maximum size for normal child is **s/2**. **So when ever you move from a node to a normal child, the sub-tree size is at most half the sub-tree size of parent node.**

We stated early that to move from **root** to any node (or viceversa) we need to change at most **log N** chains. Here is the proof; Changing a chain means we are moving for a node to a normal child, so each time we change chain we are at least halving the sub-tree size. For a

tree with size **N**, we can halve it at most **log N** times (Why? Well, take a number and keep halving, let me know if it takes more than **ceil(log N)** steps).



Each Chain is represented with different color.
Thin Black lines represent the connecting edges. They connect 2 chains.

At this point, we know what **HLD** is, we know why one has to use **HLD**, basic idea of **HLD**, terminology and proof. We shall now see implementation details of **HLD** and few related problems.

Implementation

Algorithm

```
HLD(curNode, Chain):
Add curNode to curChain
If curNode is LeafNode: return //Nothing left
to do
sc := child node with maximum sub-tree size //sc is the
special child
HLD(sc, Chain) //Extend
current chain to special child
for each child node cn of curNode: //For normal
childs
if cn != sc: HLD(cn, newChain) //As told above,
for each normal child, a new chain starts
```

Above algorithm correctly does **HLD**. But we will need bit more information when solving **HLD** related problems. We should be able to answer the following questions:

1. Given a node, to which chain does that node belong to.
2. Given a node, what is the position of that node in its chain.

3. Given a chain, what is the head of the chain
4. Given a chain, what is the length of the chain

So let us see a **C++** implementation which covers all of the above

```
int chainNo=0,chainHead[N],chainPos[N],chainInd[N],chainSize[N];
void hld(int cur) {
    if(chainHead[chainNo] == -1) chainHead[chainNo]=cur;
    chainInd[cur] = chainNo;
    chainPos[cur] = chainSize[chainNo];
    chainSize[chainNo]++;

    int ind = -1,mai = -1;
    for(int i = 0; i < adj[cur].sz; i++) { if(subsize[ adj[cur][i] ]
    > mai) {
        mai = subsize[ adj[cur][i] ];
        ind = i;
    }
    }

    if(ind >= 0) hld( adj[cur][ind] );

    for(int i = 0; i < adj[cur].sz; i++) {
        if(i != ind) {
            chainNo++;
            hld( adj[cur][i] );
        }
    }
}
```

Initially all entries of chainHead[] are set to -1. So in line 3 when ever a new chain is started, chain head is correctly assigned. As we add a new node to chain, we will note its position in the chain and increment the chain length. In the first for loop (lines 9-14) we find the child node which has maximum sub-tree size. The following if condition (line 16) is failed for leaf nodes. When the if condition passes, we expand the chain to special child. In the second for loop (lines 18-23) we recursively call the function on all normal nodes. chainNo++ ensures that we are creating a new chain for each normal child.

Example

Problem : [SPOJ – QTREE](#)

Solution : Each edge has a number associated with it. Given 2 nodes **A** and **B**, we need to find the edge on path from **A** to **B** with maximum value. Clearly we can break the path into **A** to **LCA(A, B)** and **B** to **LCA(A, B)**, calculate answer for each of them and take the maximum of both. As mentioned above as the tree need not be balanced, it may take upto **O(N)** to travel from **A** to **LCA(A, B)** and find the maximum. Let us use **HLD** as detailed above to solve the problem.

Solution Link : [Github – Qtree.cpp](#) (well commented solution)

I will not explain all the functions of the solution. I will explain how query works in detail

main() : Scans the tree, calls all required functions in order.

dfs() : Helper function. Sets up depth, subsize, parent of each node.

LCA() : Returns Lowest Common Ancestor of two node

make_tree() : Segment tree construction

update_tree() : Segment tree update. Point Update

query_tree() : Segment tree query. Range Update

HLD() : Does HL-Decomposition, similar to one explained above

change() : Performs change operation as given in problem statement

query() : We shall see in detail about the query function.

```
int query(int u, int v) {
    int lca = LCA(u, v);
    return max( query_up(u, lca), query_up(v, lca) );
}
```

we calculate LCA(u, v). we call query_up function twice once for the path u to lca and again for the path v to lca. we take the maximum of those both as the answer.

query_up() : This is **important**. This function takes 2 nodes u, v such that v is ancestor of u. That means the path from u to v is like going up the tree. We shall see how it works.

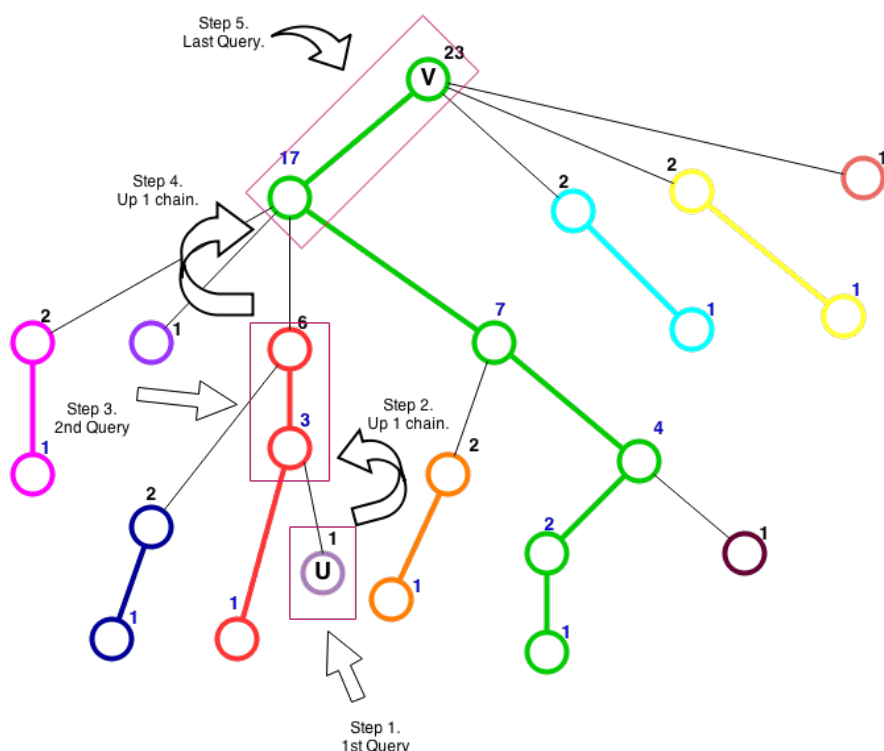
```
int query_up(int u, int v) {
    int uchain, vchain = chainInd[v], ans = -1;

    while(1) {
        if(uchain == vchain) {
            int cur = query_tree(1, 0, ptr, posInBase[v]+1, posInBase[u]+1);
            if( cur > ans ) ans = cur;
            break;
        }
        int cur = query_tree(1, 0, ptr, posInBase[chainHead[uchain]],
            posInBase[u]+1);
        if( cur > ans ) ans = cur;
        u = chainHead[uchain];
        u = parent(u);
    }
    return ans;
}
```

uchain and **vchain** are chain numbers in which **u** and **v** are present respectively. We have a while loop which goes on until we move up from **u** till **v**. We have 2 cases, one case is when both **u** and **v** belong to the same chain, in this case we can query for the range between **u** and **v**. We can stop our query at this point because we reached **v**. Second case is when **u** and **v** are in different chains. Clearly **v** is above in the tree than **u**. So we need to completely move up the chain of **u** and go to next chain above **u**. We query from **chainHead[u]** to **u**, update the answer. Now we need to change chain. Next node after

current chain is the parent of **chainHead[u]**.

Following image is the same tree we used above. I explained how query works on a path from **u** to **v**.



Consider the path from U to V.
Step 1 : Query for chain 1 in image.
Step 2 : Move up to chain 2.
Step 3 : Query for chain 2 in image. Update Answer.
Step 4 : Move up to chain 3.
Step 5 : Query for chain 3 in image. Update answer.

Problems

SPOJ – QTREE – <http://www.spoj.com/problems/QTREE/>
SPOJ – QTREE2 – <http://www.spoj.com/problems/QTREE2/>
SPOJ – QTREE3 – <http://www.spoj.com/problems/QTREE3/>
SPOJ – QTREE4 – <http://www.spoj.com/problems/QTREE4/>
SPOJ – QTREE5 – <http://www.spoj.com/problems/QTREE5/>
SPOJ – QTREE6 – <http://www.spoj.com/problems/QTREE6/>
SPOJ – QTREE7 – <http://www.spoj.com/problems/QTREE7/>
SPOJ – COT – <http://www.spoj.com/problems/COT/>
SPOJ – COT2 – <http://www.spoj.com/problems/COT2/>
SPOJ – COT3 – <http://www.spoj.com/problems/COT3/>
SPOJ – GOT – <http://www.spoj.com/problems/GOT/>
SPOJ – GRASSPLA – <http://www.spoj.com/problems/GRASSPLA/>
SPOJ – GSS7 – <http://www.spoj.com/problems/GSS7/>
CODECHEF – GERALD2 – <http://www.codechef.com/problems/GERALD2>
CODECHEF – RRTREE – <http://www.codechef.com/problems/RRTREE>
CODECHEF – QUERY – <http://www.codechef.com/problems/QUERY>
CODECHEF – QTREE – <http://www.codechef.com/problems/QTREE>
CODECHEF – DGCD – <http://www.codechef.com/problems/DGCD>

All codechef problems have Editorials.

QTREE : Explained above as example

QTREE2 : Segment trees this time should support sum operation over range. There is no update operation. Note that you can also use **BIT**. For finding the Kth element on a path, we can use **LCA** style jumping, it can be done in **$O(\log N)$** .

QTREE3 : Simple. Segment tree should answer the following: index of left most black node in a range.

QTREE4 : You need the maximum distance. Clearly we can take the left most white node and right most white node. Distance between them will be the required answer. So our segment tree should be able to answer left most white node index as well as range sum.

QTREE5 : Bit hard. Read editorial of **QTREE6** to get idea. (Correct me if I am wrong, I did not solve it yet)

QTREE6 : Editorial – [Click Here](#)

QTREE7 : Hard. Read editorial of **MONOPOLY** and **GERALD2**.

MONOPOLY : Editorial – [Click Here](#)

GERALD2 : Editorial – [Click Here](#)

The End!

Finallyyyy!!! Long one!! When I was learning various algorithms and doing programming, the topic that bugged me the most was HLD, I did not understand it for a long time, also there was not any good tutorial. So I decided to start with HLD, I hope it will help a lot 😊 This is my first attempt in writing a post/tutorial. Do share your views on it. Also I took a lot of time setting up this blog on EC2 (3 days, well I am new to this stuff), you can use contact me to write anything related to this blog, about cautions or bugs or will-do-good changes.

Share this post. Learn and let learn! ☐

[Persistent segment trees – Explained with spoj problems](#) ➔

64 Comments



Sarthak Taneja

March 22, 2016 at 5:40 PM

Link:

<http://ideone.com/CTgDs3>

➔ Reply



Sarthak Taneja

March 22, 2016 at 5:40 PM

Hey, I am getting TLE with the code written by me though it almost same as that whose link you have shared.

If possible please tell me mistake....

Link:

[↩ Reply](#)



Donglin Zhuang

March 8, 2016 at 10:58 AM

it's the best guidance i have ever seen, thx very much!

[↩ Reply](#)



PrudhviRaj

March 5, 2016 at 8:20 PM

Nice one

[↩ Reply](#)



abhishek

February 27, 2016 at 11:03 PM

no explanation can be better than your's.

[↩ Reply](#)



ibahadir.altun

January 28, 2016 at 10:16 PM

I've never seen such a awesome tutorial before. Everything has explained clearly, thank you very much 😊

[↩ Reply](#)



bholagabbar

January 24, 2016 at 5:43 PM

Great tutorial. I think there are a couple of things I would like to clarify. I think it could help clear concepts for others too.

1. We are using the terminology 'special child' and 'special edge'. AFAIK, in the original Heavy-Light Decomposition, an edge is called 'Heavy' IFF the *subtree rooted at the child vertex is $\geq (1/2)$ size of subtree rooted at parent. It is entirely possible that a node has no 'heavy children'.

However, here in this tutorial, I believe we are replacing the word 'heavy' with 'special' and are removing the $1/2$ constraint. Every parent node has a special child, it should be the one with max subtree size that's it. Am I correct in stating this?

2. In your implementation,

(<https://github.com/anudeep2011/programming/blob/master/qtree.cpp#L194>), can you please explain line 84? Why are we querying `posInBase[v]+1`, `posInBase[u]+1`? Why the +1? I'm not sure but it is because the edges are stored at a position +1 from the vertex?

The rest seems pretty clear. Thanks

@anudeep, I'd love if you could clarify my doubts 😊

➡ Reply



bholagabbar

January 24, 2016 at 8:24 PM

Anyone looking for a cleaner and slightly more lucid (1-based indexing) implementation, can refer to: <http://ideone.com/9XpM4q>

➡ Reply



bholagabbar

January 25, 2016 at 5:02 AM

More problems:

1. <https://www.codechef.com/problems/TAQTREE/> (must try. v. similar to QTREE)
2. http://www.lightoj.com/volume_showproblem.php?problem=1348 (must try. Here variation is nodes instead of edges)
3. <https://www.codechef.com/problems/CHEFTREE> (again, try to solve this. I didn't give it enough thought)
4. <https://www.codechef.com/problems/IDOLS/>
5. <https://www.codechef.com/problems/QTREE>
6. <https://www.codechef.com/problems/TRIPS>

➡ Reply



Karen Mkrtumyan

December 26, 2015 at 3:21 PM

Ty for the tutorial.

I have a question. Can we keep the answer of each chain and for the whole chain answer the query in $O(1)$? We would have $O(\log(N))$ for each given query then.

Waiting for a reply.

➡ Reply



Jaswant Singh

December 16, 2015 at 3:02 PM

nice tutorial .. 😊 was trying for an generalized code in Java , i found one at Github :

<https://github.com/indy256/codelibrary/blob/master/java/src/HeavyLight.java>

but unable to understand functions [modifyOperation() , deltaEffectOnSegment() ,

getNeutralDelta()).

.....please provide some explanation for these functions so . It will be helpful.

Thanks .

[↩ Reply](#)



Vidit

September 26, 2015 at 6:18 PM

i couldn't understand the function of otherEnd array and how change function works, please help me understand it clearly.

[↩ Reply](#)



Vidit

September 22, 2015 at 6:39 PM

In the code for qtree i can't understand the use of otherEnd array

[↩ Reply](#)



rtheman

August 27, 2015 at 2:07 PM

Excellent explanation! Loved it!!

[↩ Reply](#)



parijat

July 4, 2015 at 12:56 AM

could you tell me why there is +1 in ->query_tree(1, 0, ptr, posInBase[v]+1, posInBase[u]+1);

[↩ Reply](#)



parijat

July 4, 2015 at 6:35 PM

got it thanks.

amazing tutorial. please continue this good work

[↩ Reply](#)



Shamit

April 19, 2016 at 11:17 PM

Hi

Can you explain why +1 is used. I understood +1 for v because v is the ancestor node and the cost of edge v-x (where v is parent of x) is stored in node x. But why is +1 used for "u"?

[↩ Reply](#)



ibahadir.altun

April 21, 2016 at 8:37 PM

Hi,

I have been unstable at that point, too. But then I realized that, in make_tree the base-case is as follows :

```
if(s == e-1) {  
    st[cur] = baseArray[s];  
    return;  
}
```

So, e equals to (end_of_current_segment + 1). That's the reason behind why we used (posInBase[u] + 1). So you can easily build tree like, make_tree(1, 0, ptr - 1), and change the query_tree function to the query_tree(1, 0, ptr - 1, posInBase[v]+1, posInBase[u]). There's is no need to use +1 if you change functions as above.

[↩ Reply](#)



I_am_looser

May 19, 2015 at 11:20 AM

I have read post on topcoder for LCA. In that post they are checking for condition $P[p][i] \neq -1$...but in your code you ain't why?

[↩ Reply](#)



gaurav chandel

April 6, 2015 at 12:01 PM

Brilliant explanation...You have done a lot of hardwork....Its easy and its fun.....Thank you very much.....and please , dont stop writing such tutorials ...

[↩ Reply](#)



Deepak Kumar Singh

March 24, 2015 at 11:53 AM

Please can you tell me how you are calculating the LCA of two nodes using DP and that recursive function ?

[↩ Reply](#)



shubham

March 22, 2015 at 11:09 AM

nice tutorial

[↩ Reply](#)



Parshant garg

March 21, 2015 at 10:57 PM

awesome tutorial. I need more tutorial on some topics trie, suffix array.

[↩ Reply](#)



sai krishna

March 18, 2015 at 7:36 AM

colored one are different chains am i right right?

[↩ Reply](#)



Saurabh Verma

March 13, 2015 at 3:47 AM

You made it too easy for me 😊

[↩ Reply](#)



Akhil Appana

March 12, 2015 at 11:07 PM

there are many algos for lca, what algorithm would you prefer to use for finding lca ?

[↩ Reply](#)



Abu Asif Khan Chowdhury

February 18, 2015 at 3:22 PM

What is the purpose of this array otherEnd[] ? I'm stuck in here.

[↩ Reply](#)



zuizui123

February 9, 2015 at 8:58 PM

I want more super-easy-to-understand posts like this. Thank you so much!

[↩ Reply](#)



Saurabh Hota

February 1, 2015 at 3:22 PM

In QTREE4 explanation you have not taken care of the negative weights

[↩ Reply](#)



ankit

January 24, 2015 at 7:56 PM

hi anudeep..can u plz explain the dynamic programming approach u used for computing lca..

[↩ Reply](#)



Nikhil Gupta

January 20, 2015 at 2:43 PM

Fantastic tutorial!

[↩ Reply](#)



Shubham

January 7, 2015 at 11:57 AM

Hi, In query_up function, why in this line "query_tree(1, 0, ptr, posInBase[v]+1, posInBase[u]+1);" posInBase[v]+1 & posInBase[u]+1 is used ?

Shouldn't be posInBase[v] & posInBase[u]?

Thnx. for the great tutorial !!!

[↩ Reply](#)



Aditya

December 25, 2014 at 12:59 PM

Anudeep, in your explanation of QTREE4 can you what you mean by the left most and right most white nodes? In the question it is stated that the distance between nodes can be negative.

[↩ Reply](#)



★ anudeep2011

December 28, 2014 at 8:38 AM

Note that the solution is not trivial.

We need a, b such that a and b are white and distance(a, b) is maximum.

Whatever the a, b be, for node LCA(a, b) one will be in its left subtree, other will be in its right subtree.

For each node x, if you have the left farthest white node and right farthest white node, you can get the answer.

[↩ Reply](#)



★ Rishit

December 7, 2015 at 2:06 PM

Anudeep, can you elaborate more on this? I am trying to solve using above mentioned technique, but unable to proceed further. I am stuck in finding out the information to store in segment tree base array for answering query in logarithmic time.

[↩ Reply](#)



Vladimir

December 22, 2014 at 12:18 AM

"Each non leaf node has exactly one Special child." And what if:

4-2-1

|

2-1

Who is special here?

[↩ Reply](#)



★ **anudeep2011**

December 26, 2014 at 9:25 PM

It can be either of them.

[↩ Reply](#)



Karen Mkrtumyan

December 26, 2015 at 3:39 PM

Isn't that 4 actually 5?

[↩ Reply](#)



Vivek Rai

December 13, 2014 at 11:44 AM

Thats really great 😊 =D !!!!! Thanks !!

[↩ Reply](#)



Tom

November 29, 2014 at 3:24 AM

"The mininum sub-tree size possible for special child is $\text{ceil}(s/m)$. This being said, the maximum size for normal child is $\text{ceil}(s/m)$ and $m > 2$." – I think the first sentence is true, but the second one is not.

[↩ Reply](#)



★ **anudeep2011**

December 26, 2014 at 9:38 PM

Corrected 😊

[↩ Reply](#)



Mohammad Samiul Islam

November 26, 2014 at 11:58 PM

I liked the way you chained a node with random node when there was no heavy edge connected to it. Nice optimization.

[↩ Reply](#)



James Porcelli

November 23, 2014 at 8:59 AM

Thank you very much!

[↩ Reply](#)



Sharad

November 2, 2014 at 6:23 PM

Hi, anudeep i think there is some problem with the 2nd para in which you said that the sub-tree size at least becomes half and on the other hand you are saying that it's size should be $\leq s/2$ which indicates that instead of at-least it should be at-most which is ambiguous in your description .

[↩ Reply](#)



★ **anudeep2011**

December 26, 2014 at 9:32 PM

Made it clear. Thank you!

[↩ Reply](#)



Aravind Srinivas

November 2, 2014 at 1:29 AM

Nice tutorial.

[↩ Reply](#)



Ahmed Morst

October 21, 2014 at 1:34 AM

your post is really helpful thnx alot ☐

[↩ Reply](#)



sunil

October 15, 2014 at 12:04 AM

anudeep can u please add a driver main for which this fuction hld will run

thankyou

[↩ Reply](#)



greendragons

October 9, 2014 at 6:47 AM

Really beautiful technique.. and explanation made it so easy to grab, which otherwise would have been tough to get at once... 😊

If you have time, please write about $O(n)$ suffix array and LCP construction.. there

are lots of $O(n \lg^2 n)$ material on web, but for $O(n)$ either mostly links are redirected to research papers or explained in very cryptic style..

 Reply



★ anudeep2011

December 26, 2014 at 2:32 PM

Thank you.

About the $O(N)$ suffix array,

- 1) I never tried to learn it, for competitive programming it was not worth it.
- 2) Almost always a $O(N * \log N)$ is good enough.

I will anyway add that to my TODO list (To learn it) and then decide about writing a blog or not 😊

 Reply



Niloy D Rocker

September 23, 2014 at 11:19 AM

Great tutorial man, you make great hard work to understand this, but make a way for us to learn the thing only in 10 mins! 😊 Hats off to you buddy! Carry on! 😊

 Reply



Jules

August 30, 2014 at 12:39 AM

"This being said, the maximum size for normal child is $\text{ceil}(s/m)$ and $m > 2$."

This is false, right?

You can only say it's $s/2$ maximum, but for example if $m=3$, there can be one child with 5, another with 5 and the third one with 2 ; and $\text{ceil}(s/m) = 4$.

(What you actually proved is that not all normal child can be greater than $\text{ceil}(s/m)$)

That being said, thank you for the tutorial ! 😊

 Reply



★ anudeep2011

December 26, 2014 at 9:35 PM

Corrected! Thank you

 Reply



Pratik kumar

August 18, 2014 at 5:20 PM

Hey Anudeep nice tutorial.....easy to understand and to the point.

Please look my code for errors:

<http://ideone.com/foVgD4>

I used the same concept but kept edge cost at the child side of the edge so as to avoid ambiguity as you did and hence queried using the child sided node.....but still it returns sigsegy error on submission on spoj.

↩ Reply



★ anudeep2011

August 19, 2014 at 2:14 AM

Looking into code and finding bug is just very hard. I would not want to do that.

↩ Reply



Sikander

August 12, 2014 at 12:40 AM

Had been postponing learning HLD since months because of it being a hard topic, learned it in less than an hour thanks to your amazing explanation.

↩ Reply



virat sharma

August 3, 2014 at 2:40 PM

Can u explain How can we solve <http://www.spoj.com/problems/COT/> using HLD bcz i didn't figure out.

↩ Reply



Karen Mkrtumyan

December 26, 2015 at 3:33 PM

You need to use persistent segment tree too. Trying to code right now ☐

↩ Reply



Saumye

July 24, 2014 at 6:22 PM

Wow! Just landed up on your blog and it's very informative.
You explain stuff very well. Should leave Google and start conducting workshops in India. ☐

PS. If you can please do a blog article on different uncommon data structures(not the common ones). 😊

↩ Reply



★ anudeep2011

July 27, 2014 at 1:08 AM

Thank you! Yes, I always try to blog about stuff that has least resources online,

or the ones that are not common with sub-continent programmers!

 Reply



rahul kumar

July 23, 2014 at 1:23 PM

hi anudeep ,thank u for this greate tutorial ..finally i learnt how the hld works...
i implemented QTREE but it gives TLE ...i used similar code for HLD as of yours
here is the link <https://github.com/princerk/code/blob/master/.gitignore>
i uesd segment tree discussd in utkarsh's blog and calculated LCA using HLD
(im sure it wont take more than 5 min of urs)

 Reply



rohit verma

July 14, 2014 at 2:32 AM

nice tutorial 😊

 Reply



Mriganka Basu Roy Chowdhury

July 13, 2014 at 8:12 PM

This is great. ☐ Yes, totally agreed, HLD has very less number of tutorials; so
thanks a lot. Could you please refer me / write one (this is even better :D) tutorial on
2D segtrees? This year I went to IOITC, and there was a problem on 2D BIT; in IOI
2013 there was one on 2D seg trees, so I thought knowing 2D segtrees would help.

 Reply



★ **anudeep2011**

July 13, 2014 at 8:54 PM

Thank you. I will surely consider that topic.

 Reply

Leave a Reply

Enter your comment here...

