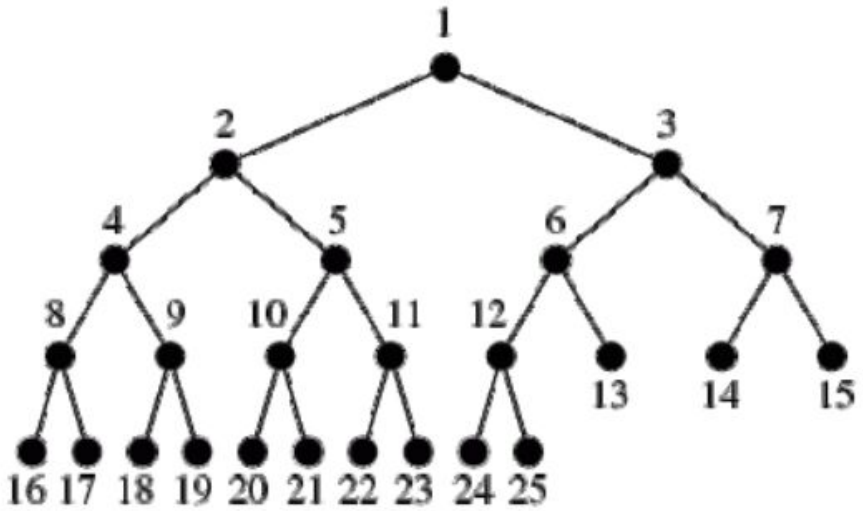
Heavy Light Decomposition

11 April 2014 anudeep2011 Algorithms, Segment trees 96 Comments

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?



Balanced Binary Tree

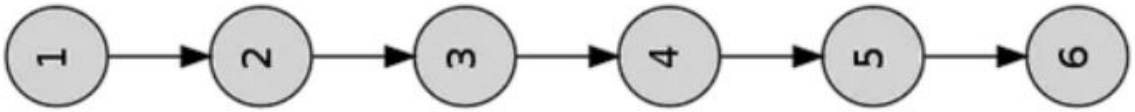
A balanced binary tree with **N** nodes has a height of **log N**. This gives us the following properties:

- 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?



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?

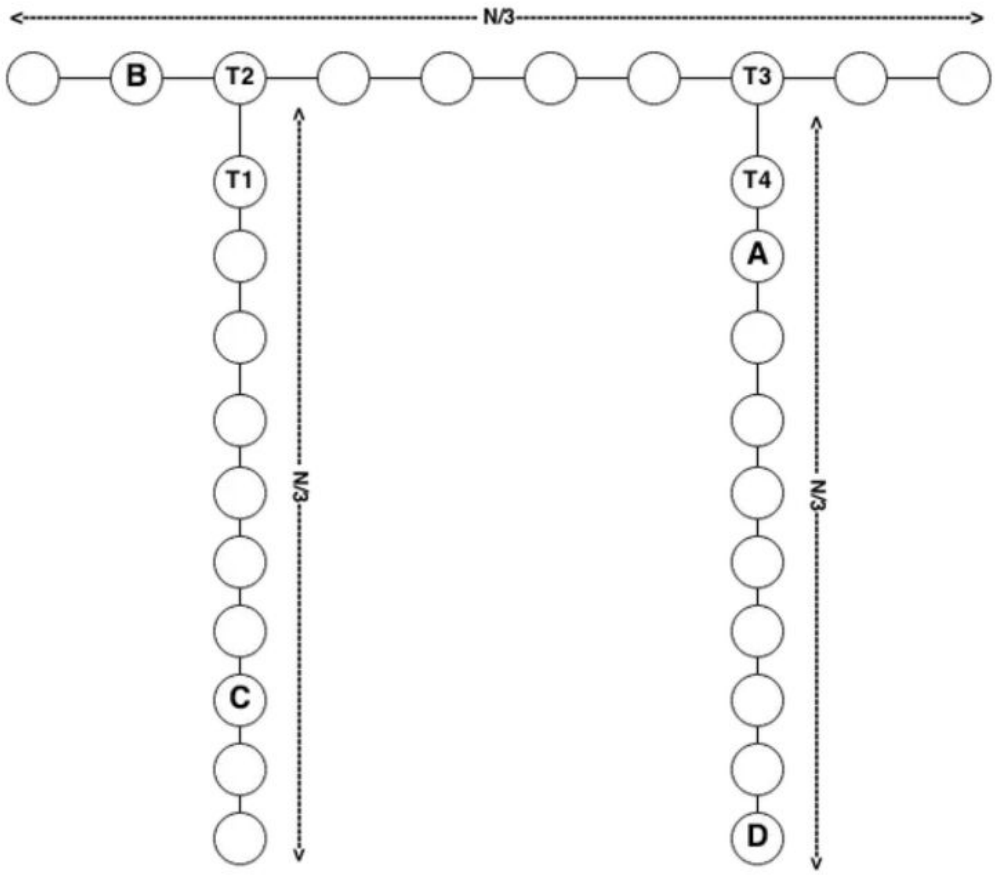
Height of unbalanced tree can be arbitrary. In the worst case, we have to visit **O(N)** nodes to move from one node to another node. So Unbalanced trees are not computation friendly. We shall see how we can deal with unbalanced trees.

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**

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**.



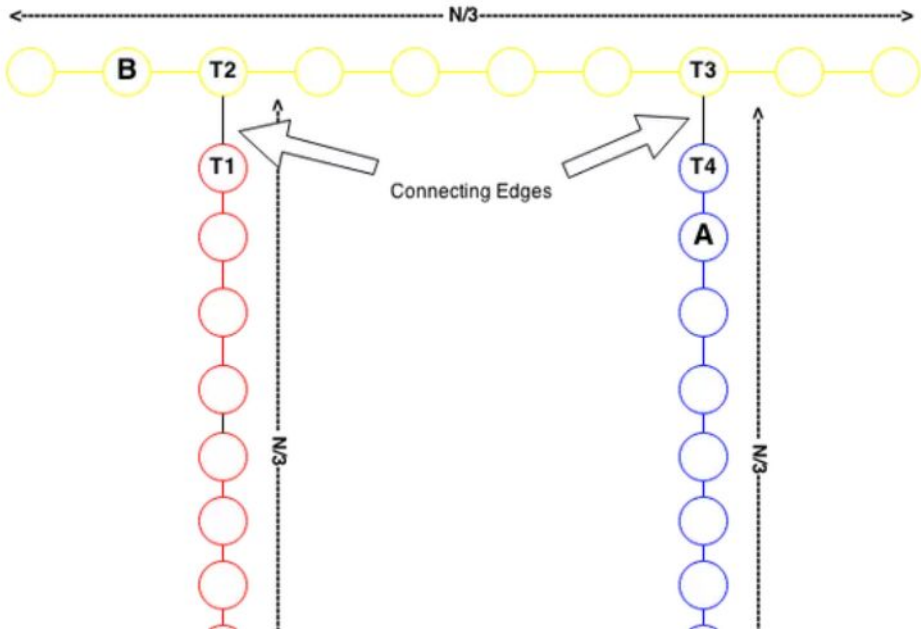
Here are details about the given images

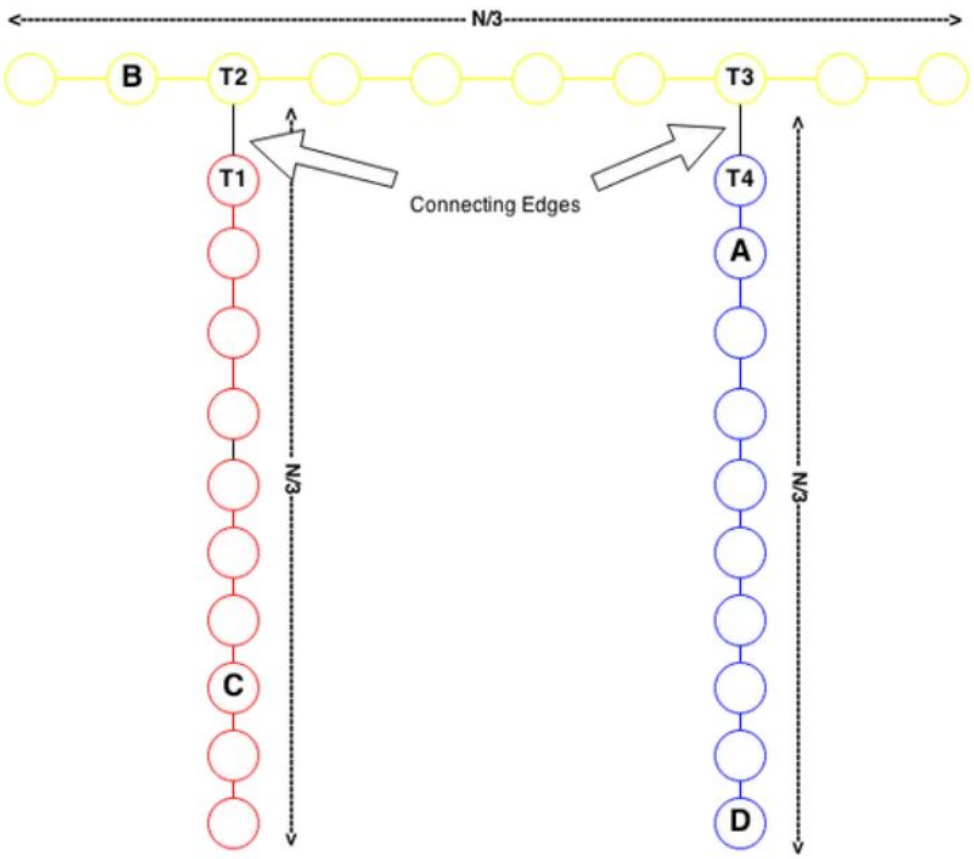
Here are details about the given images

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 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

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 some thing 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

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.

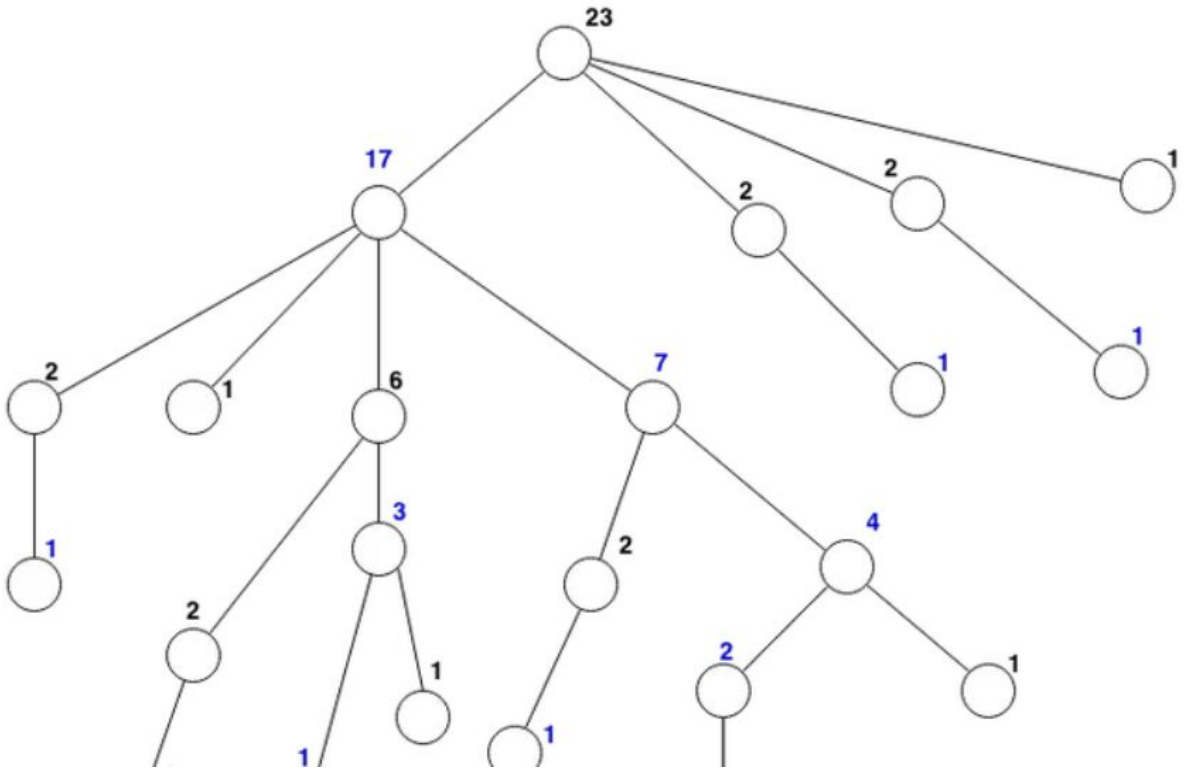
Note : My version of HLD is little different from the standard one, but still everything said above holds.

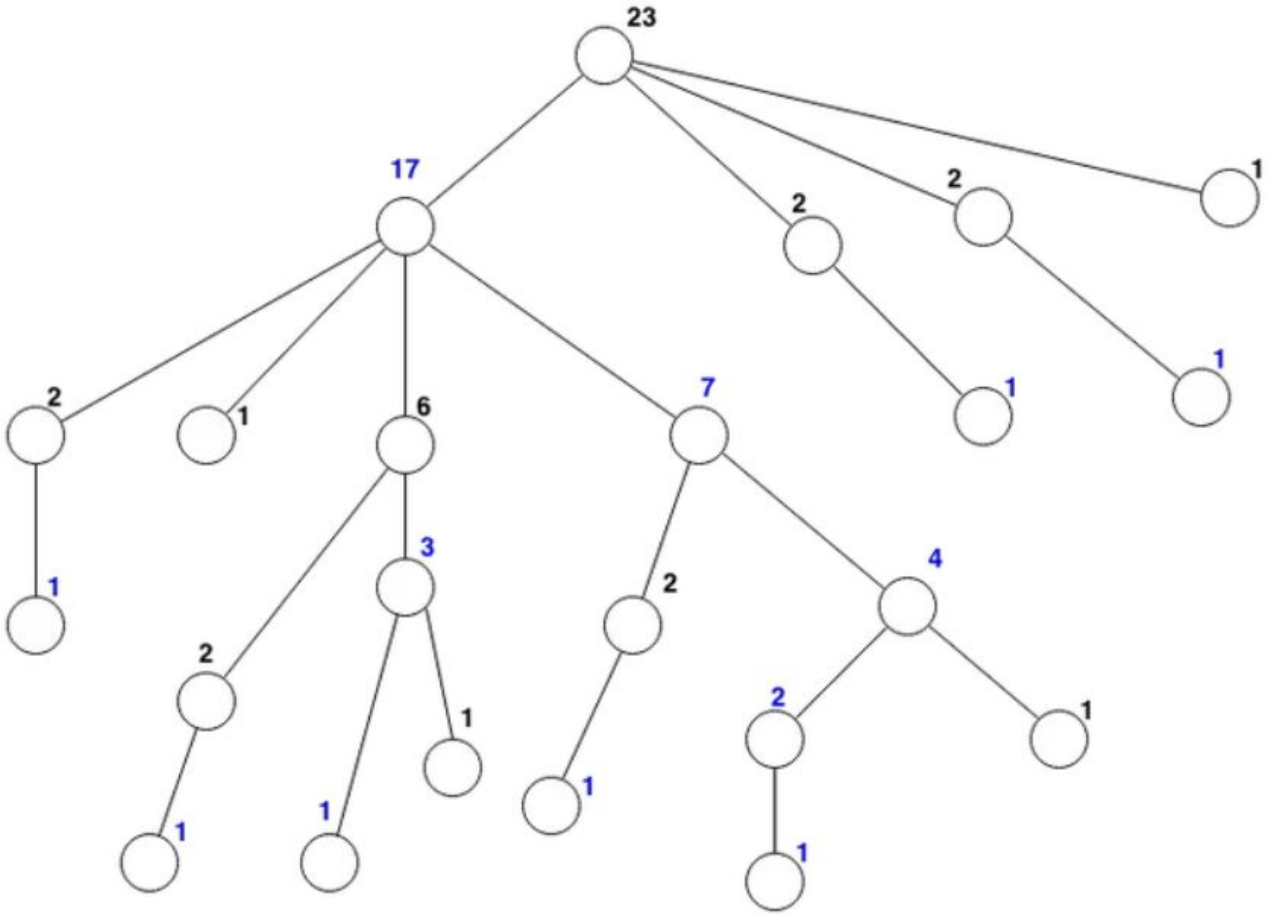
Terminology

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.

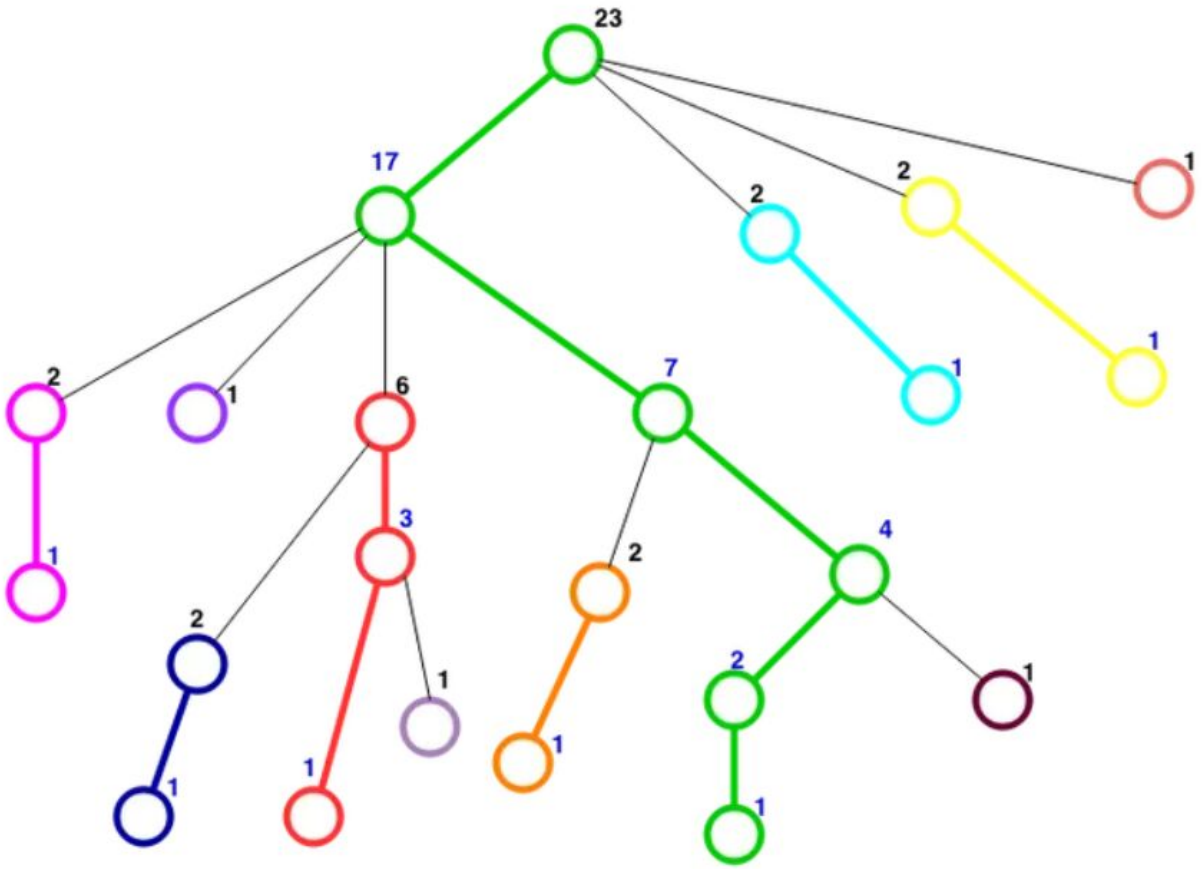
Read the next 3 paras until you clearly understand every line of it, every line makes sense (Hope!).
Read it 2 times 3 times 10 times 2 power 10 times .. , until you understand!!

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 **<=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 mininum 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

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 speci
    for each child node cn of curNode:                //For normal childs
        if cn != sc: HLD(cn, newChain)                //As told above, for each norma
```

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;
```

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

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 statemenm

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) {
```

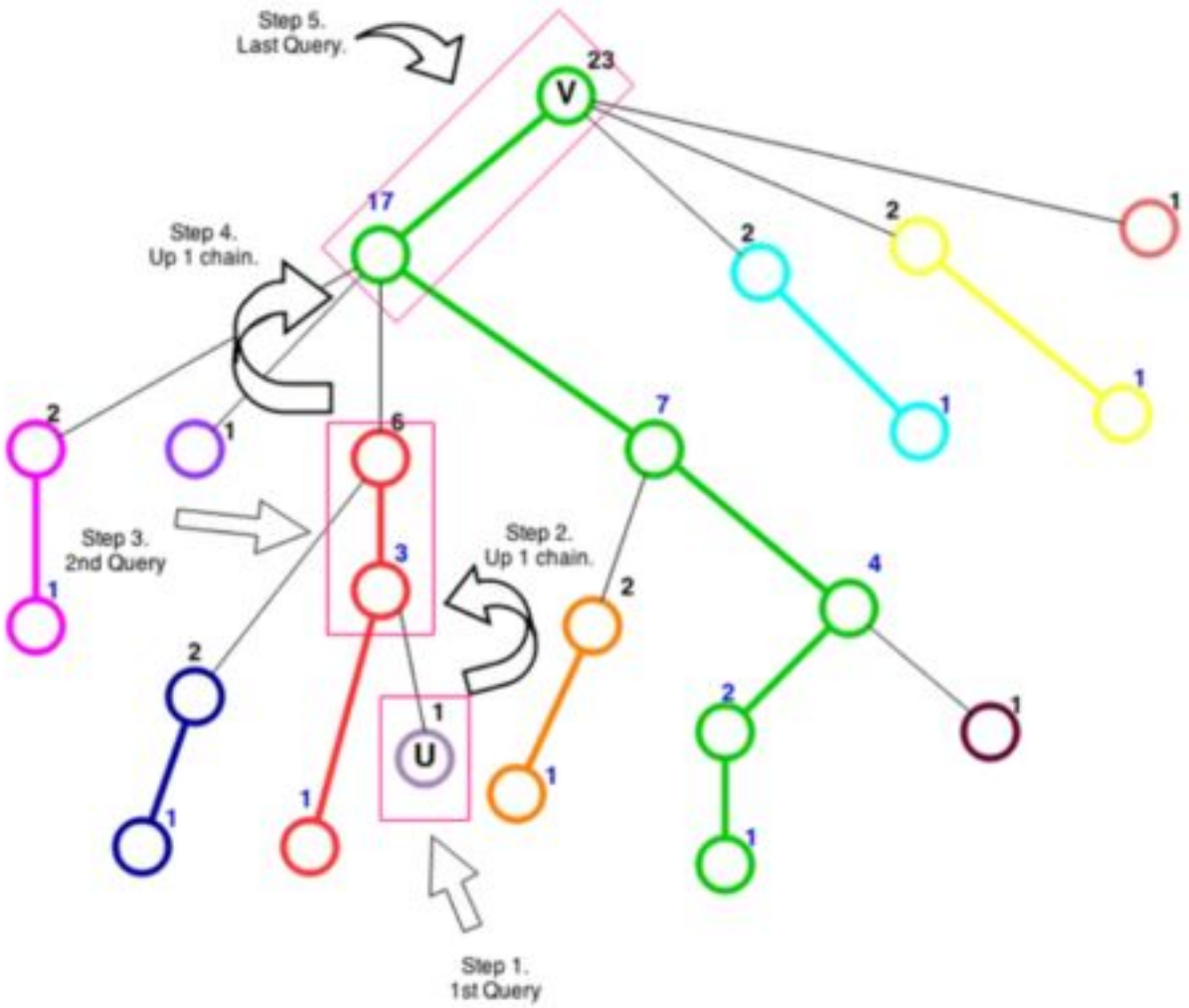
query_up() : This is **important.** This function takes 2 nodes u, v such that v is ancestor of u. That means the path from u to v is like going up the tree. We shall see how it works.

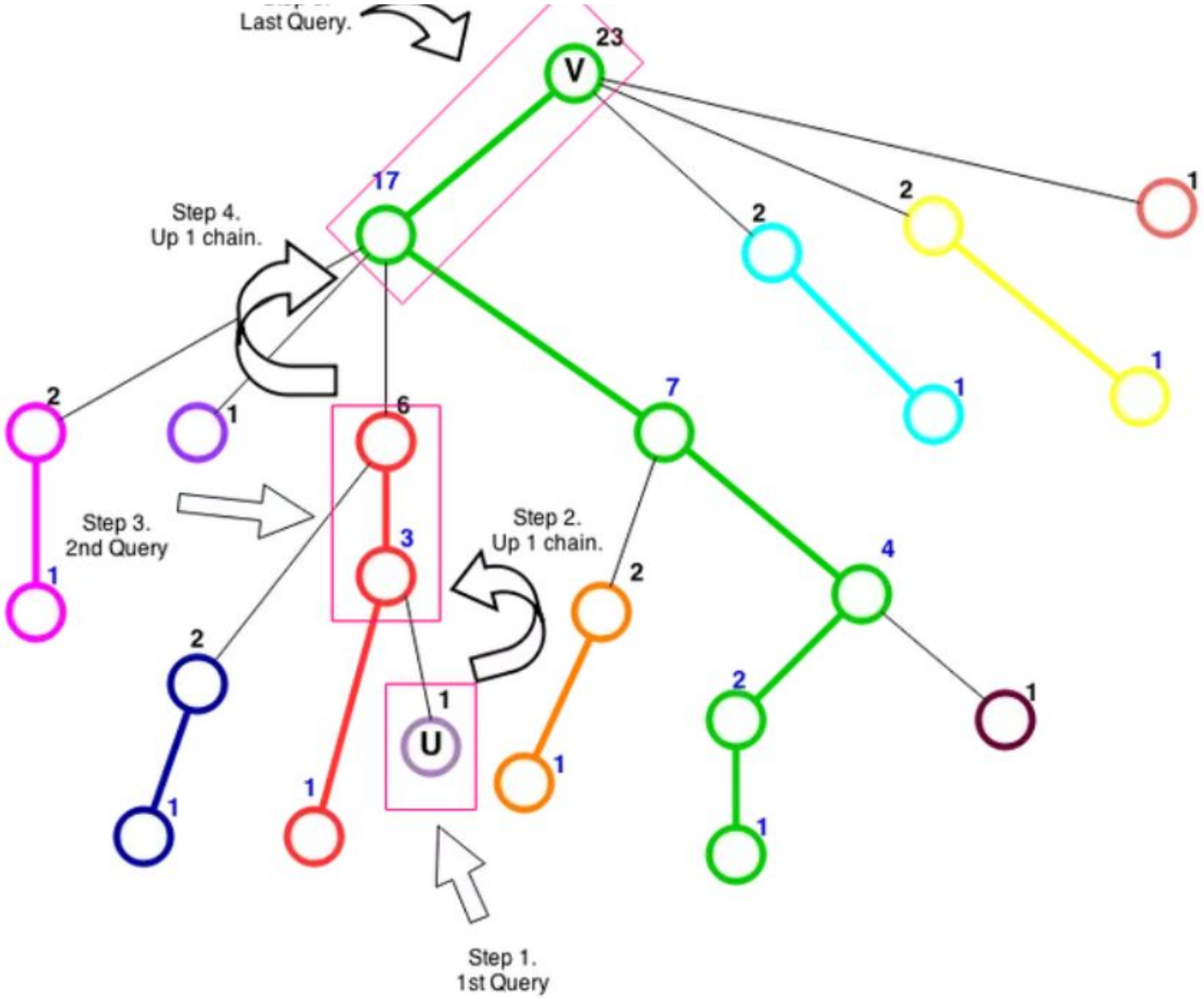
```
int query_up(int u, int v) {
    int uchain, vchain = chainInd[v], ans = -1;

    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

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
CODECHEF – MONOPLOY – http://www.codechef.com/problems/MONOPLOY

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)**

CODECHEF MONOPLOY http://www.codechef.com/problems/MONOPLOY

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!

Finallyyy!!! 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! 😀