Tree Algorithms

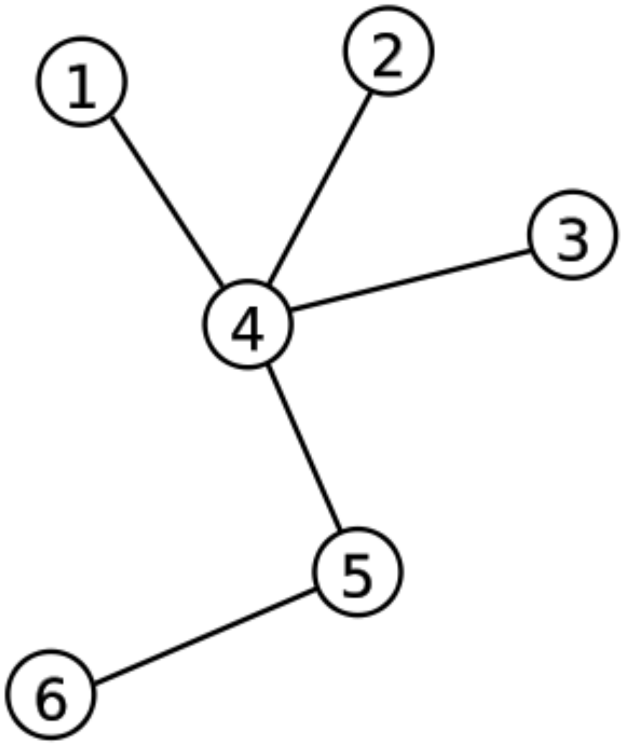
by Paul Wang

Outline

- Introduction
- Euler Tour Technique
- Lowest Common Ancestor
- Heavy-Light Decomposition
- Centroid Decomposition

Introduction

What is a Tree?



Storing a Tree

Treat it as general graph

```
// node numbered from 1 ~ n
// store their adjacency list
vector< Edge > adj[ N ] ;

// traverse neighbors of u
for ( Edge e : adj[ u ] ) {
  int v = e.node ;
  // do anything...
}
```

Trees in Competitive Programming

- DFS Tree, BFS Tree
- Tree Data Structures, e.g. Segment Tree, Treap…
- Minimum Spanning Tree, Shortest Path Tree, Dominator Tree
- Dynamic Programming on Trees
- Update/Query on Trees

Problem (Tree Game)

You are playing a game with Alice.

Given a tree $\boldsymbol{T}$, initially node i has a value A_i on it. Now you put a token on node u, and you move alternatively. Each step, you decrement the value of the current node, and then move the token to a neighbor node. When it's your turn and the current value is $\boldsymbol{0}$, you lose. You go first, who will win?

What we will cover:

Mostly ***Update/Query Problems***, some ***dynamic programming***.

In these problems, we usually care about ***Path*** and ***Subtree***…

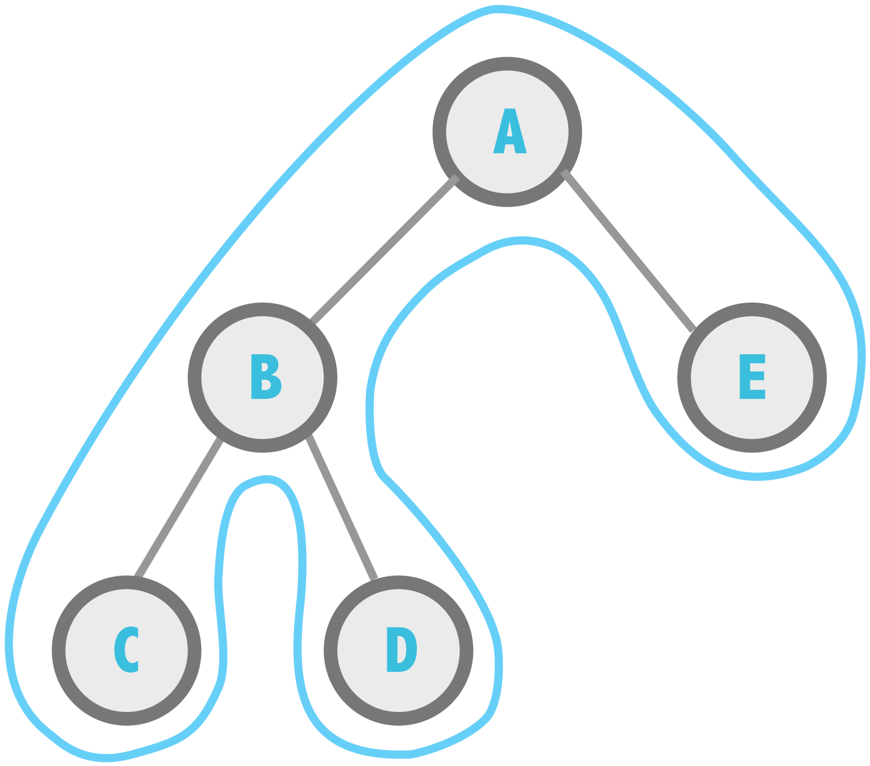
Euler Tour Technique

Depth-First Search

```
void dfs( int now , int p ) {
  for ( int nxt : adj[ now ] ) if ( nxt != p ) {
    dfs( nxt , now ) ;
  }
}

// start from root
dfs( root , -1 ) ;
```

Euler Tour from DFS



ABCBDBAEA

Time Stamp

Store the time ***in*** and ***out*** of a node

```
int timer = 0 ;

void dfs( int u , int p ) {
  time_in[ u ] = ++ timer ;
  for ( int v : adj[ u ] ) if( v != p )
    dfs( v , u ) ;
  time_out[ u ] = ++ timer ;
}

dfs( root , -1 ) ;
```

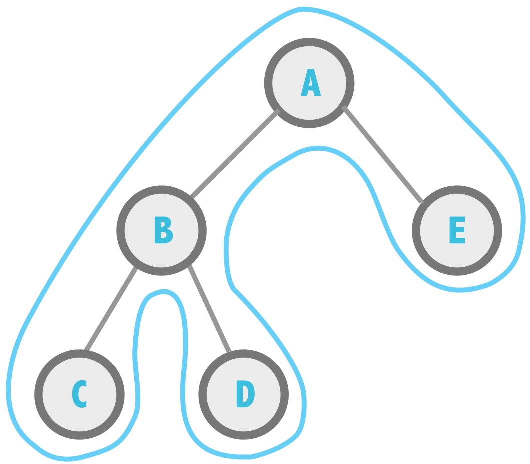
Time Stamp Properties

- Decide if "x" is ancestor of "y" in ***O(1)***

```
bool anc( int x , int y ) {
  return time_in[ x ] <= time_in[ y ]
      && time_out[ y ] <= time_out[ x ];
}
```

- Any ***subtree*** is a ***segment*** - make the tree a ***sequence***!!

A(1, 10) B(2, 7) C(3, 4) D(5, 6) E(8, 9)



In sequence:

A B C C D D B E E A

or

A B C D E

Exercise

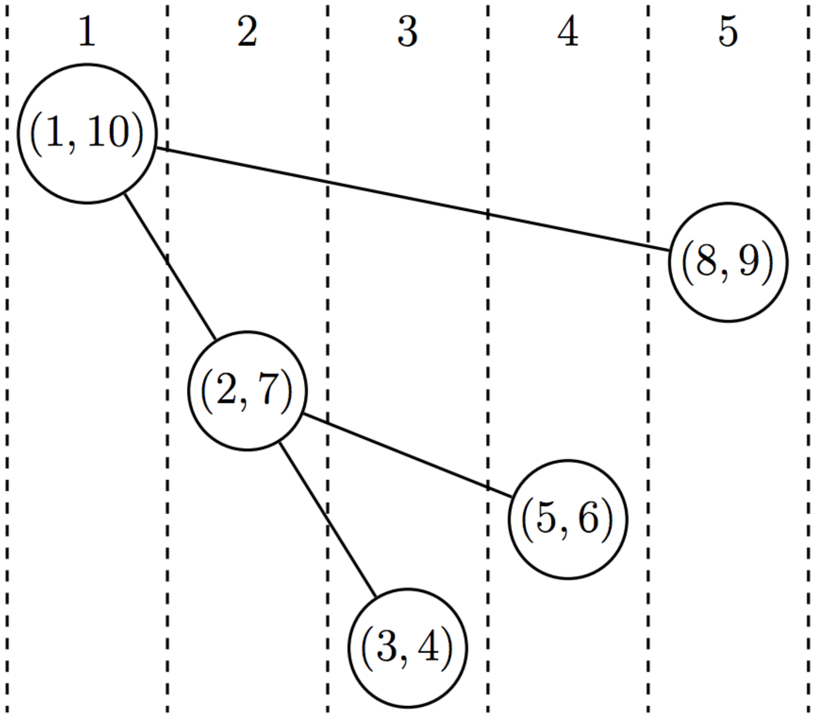
Given tree T, r is the root.

Make the following operations work in $O(\log N)$.

- Add v to all nodes in a subtree
- Query the current value of a node
- Query the sum of a subtree

Time Stamp in 2D

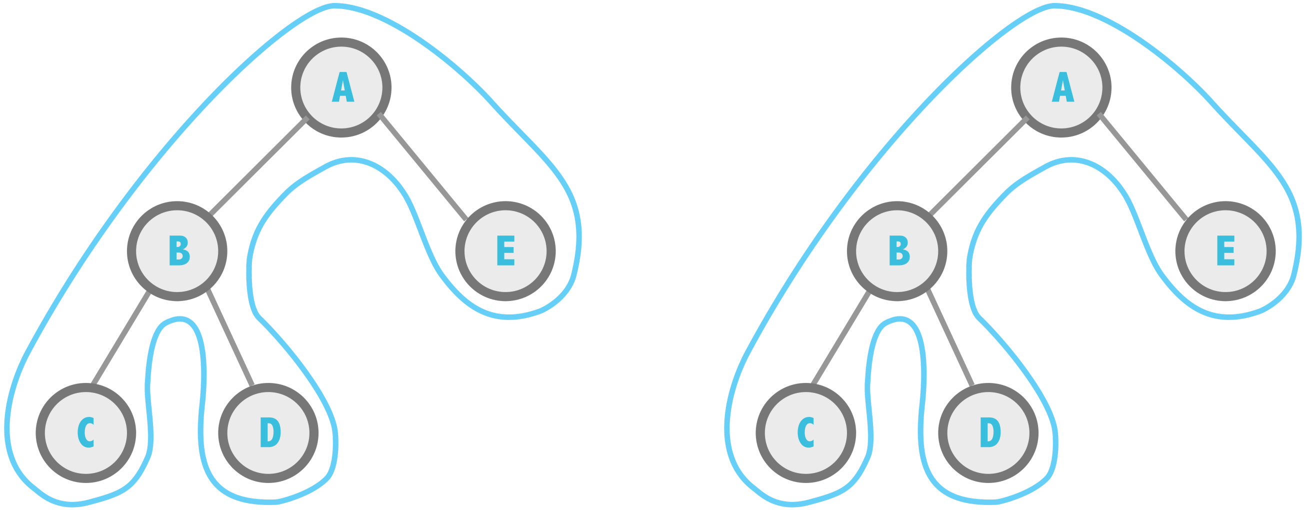
Every path between a node and its ancestors is in a rectangle.



Euler Tour Tree

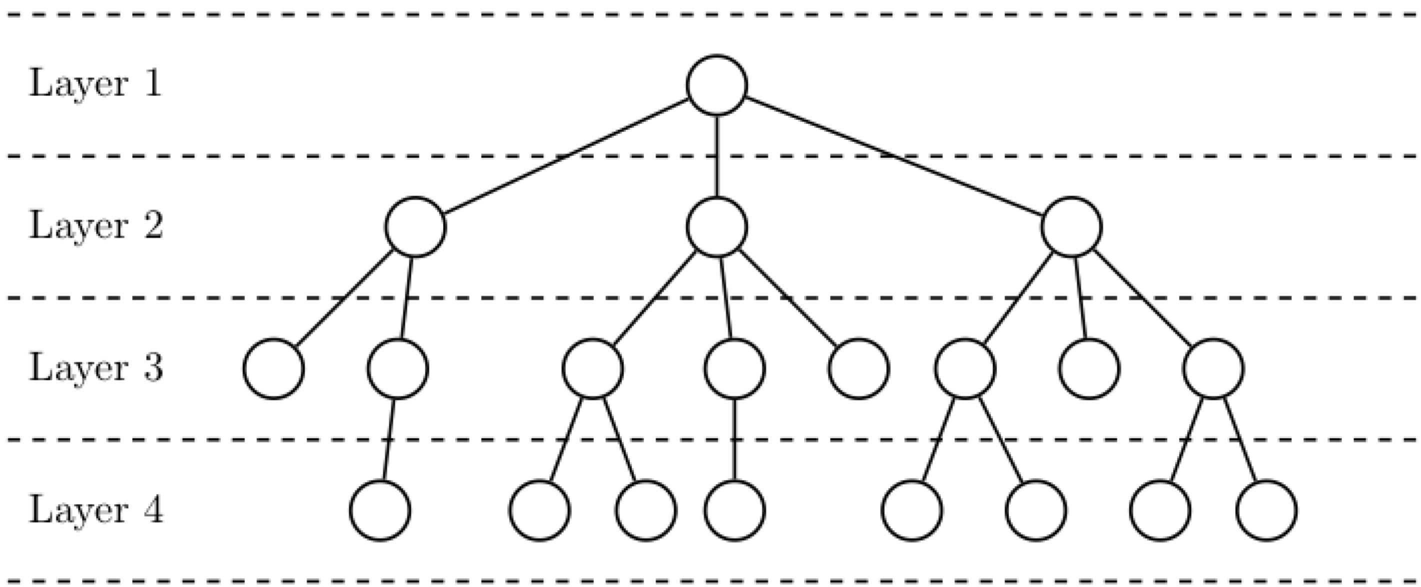
Represent the tree with the Euler Tour sequence. We can ***cut an edge*** / ***link two trees*** / ***change root*** easily. Solves ***dynamic connectivity*** problem on trees.

Implement with BST or Treap.



Lowest Common Ancestor

Definition



How fast can we compute LCA?

Method	Precompute	Query
Brute force	O(1)	O(N)
樹壓平	O(N)	O(1)/O(log N)
倍增法	O(N log N)	O(log N)

倍增法（預處理）

```
// adj[ u ] : adjacency list of u
// par[ u ][ i ] : (2^i)-th parent of u
int timer = 0 ;

void dfs( int u , int p ) {
  par[ u ][ 0 ] = p ;
  time_in[ u ] = ++ timer ;
  for ( int v : adj[ u ] ) if( v != p )
    dfs( v , u ) ;
  time_out[ u ] = ++ timer ;
}

int main() {
  int root = 1 ; // set root node
  dfs( root , root ) ;

  for ( int j = 1 ; j <= LOG ; j ++ ) {
    for ( int i = 1 ; i <= n ; i ++ ) {
      par[ i ][ j ] = par[ par[ i ][ j - 1 ] ][ j - 1 ] ;
    }
  }
}
```

倍增法（查詢）

```
int LOG = 20 ;
bool anc( int x , int y ) {
  return time_in[ x ] <= time_in[ y ]
      && time_out[ y ] <= time_out[ x ];
}

int lca( int x , int y ) {
  if ( anc( y , x ) ) return y;
  for ( int j = LOG ; j >= 0 ; j -- )
    if ( !anc( par[ y ][ j ], x ) ) y = par[ y ][ j ] ;
  return par[ y ][ 0 ] ;
}
```

Exercise

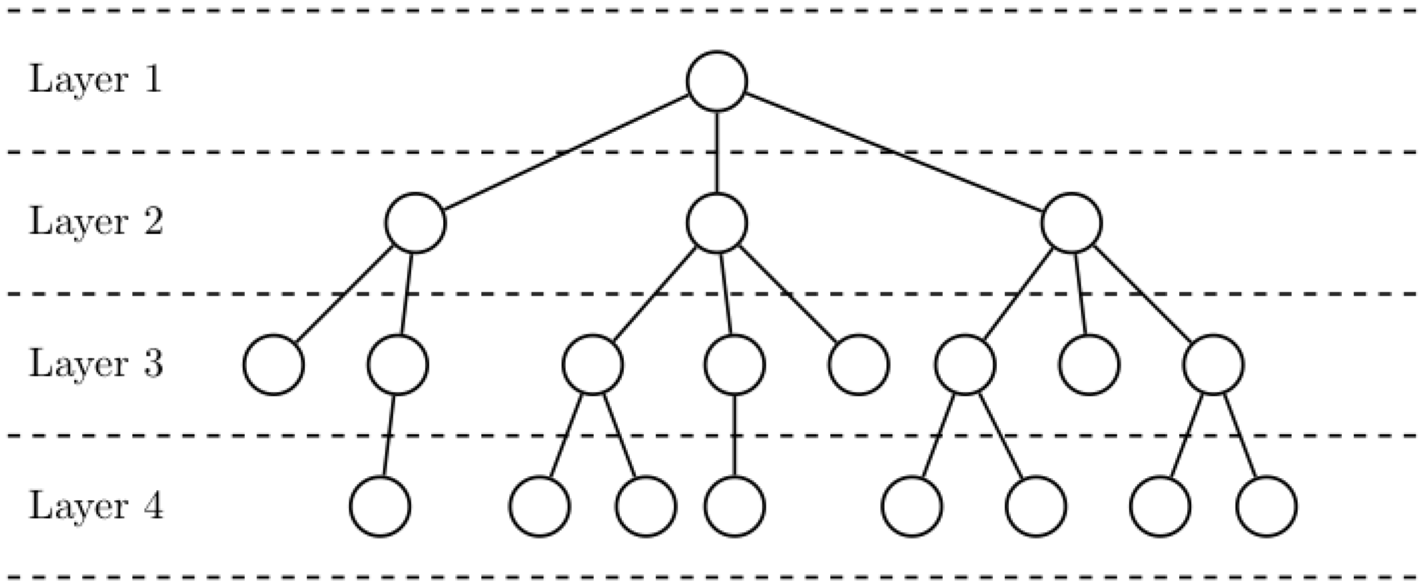
Given a tree T and an empty set S.

Make the following operations work in $O(\log N)$.

- Insert the path from u to v into S
- Given u and v, how many paths in S intersects with the path from u to v?

Key Insight

- ***path(u,v)*** intersects with ***path(p,q)*** if and only if ***lca(u,v)*** is on ***path(p,q)*** or ***lca(p,q)*** is on ***path(u,v)***



Heavy-Light Decomposition

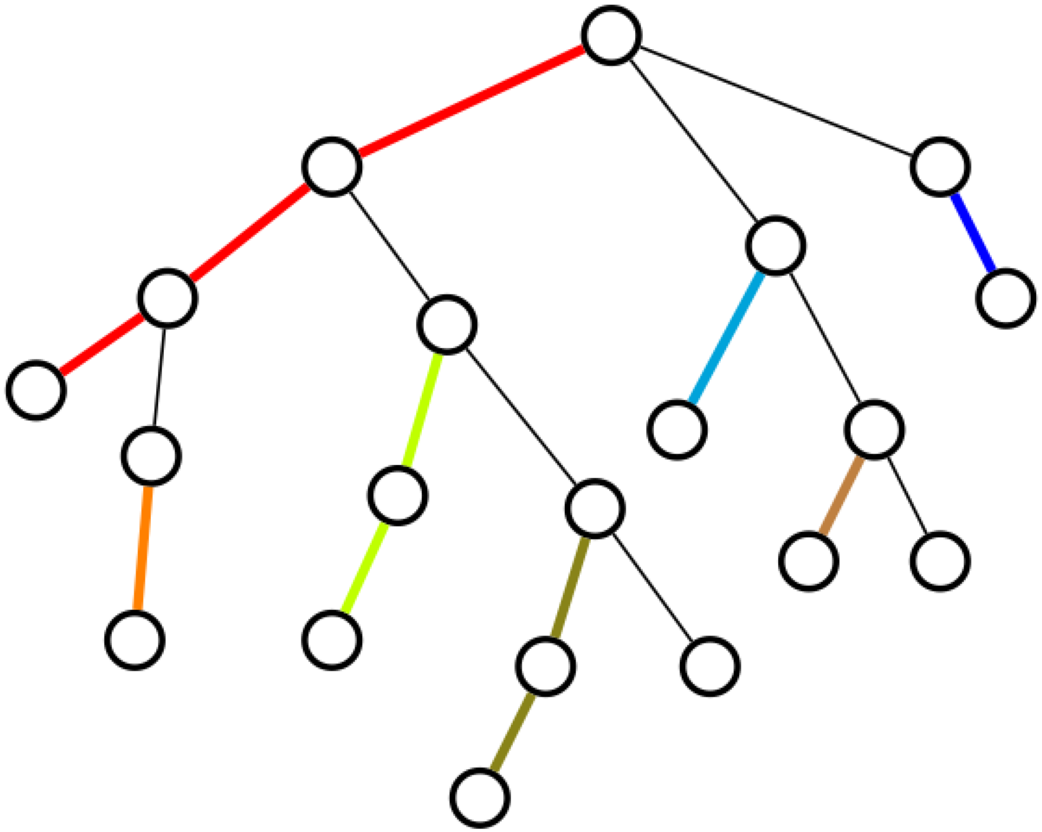
Problem (QTREE)

Given a tree ***T***, make the following operations efficient.

- Update the weight of some edge
- Given u and v, which edge has the maximum weight on ***path(u,v)***

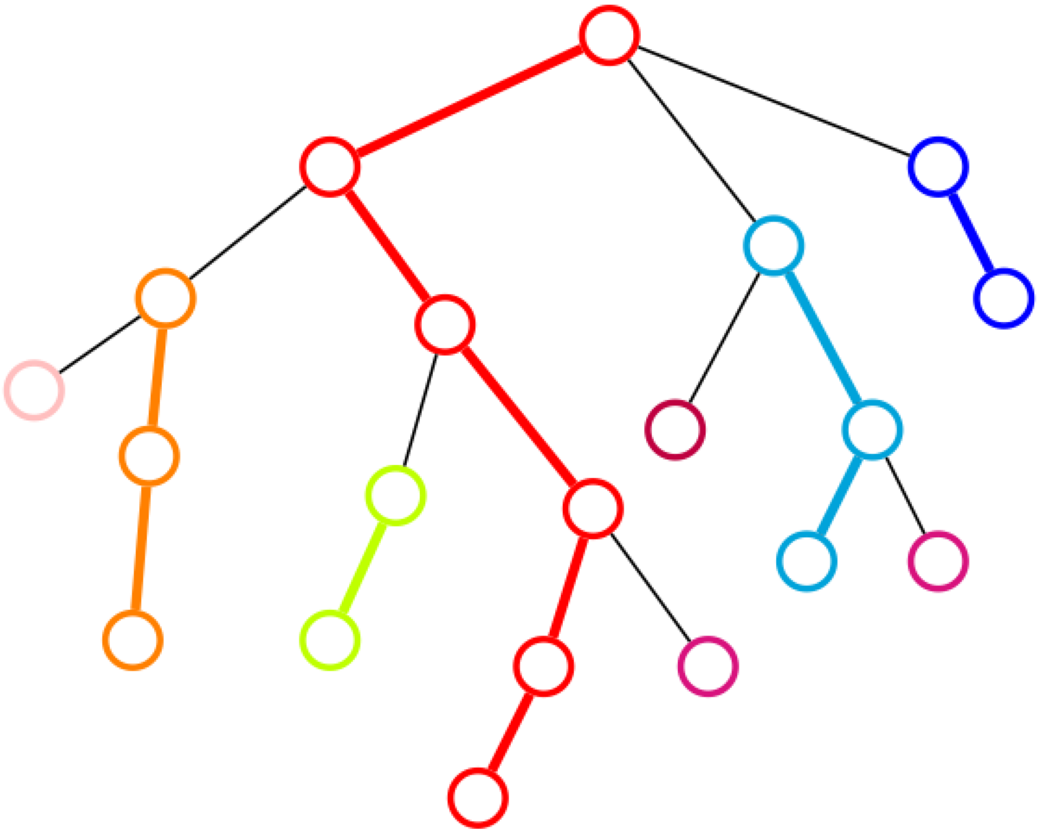
Sequence?

- Path does not form a continuous segment
- Still OK sometimes



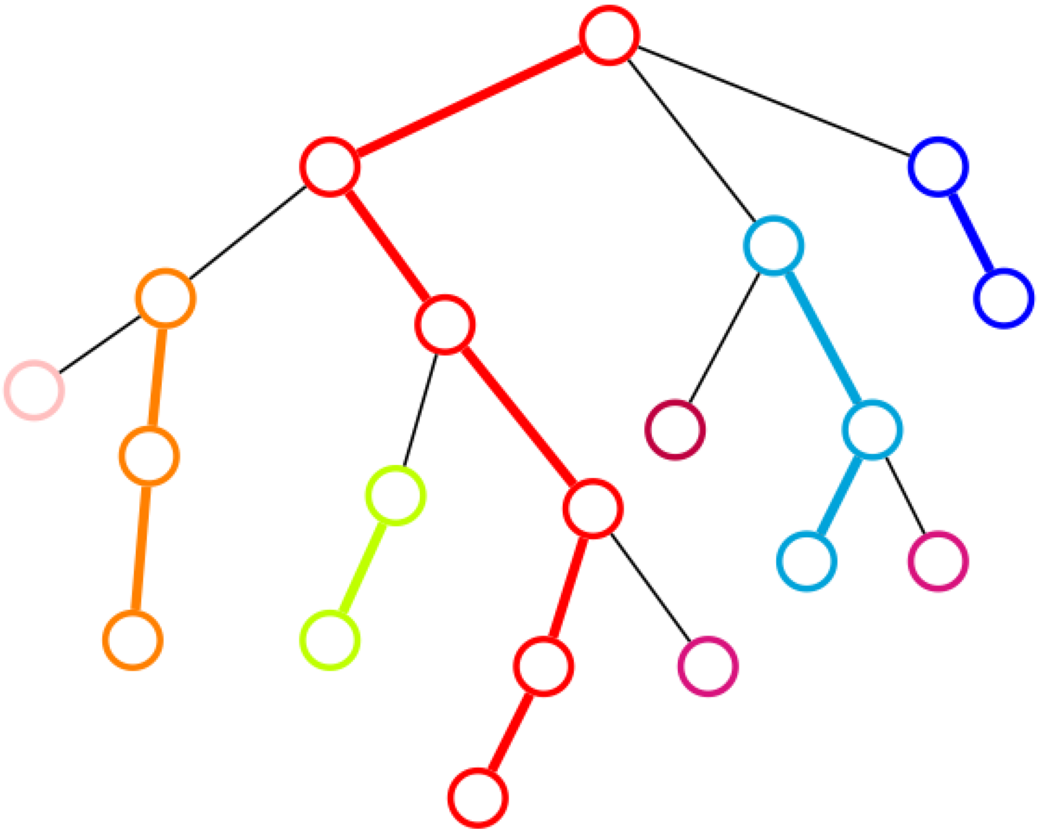
Partition to Chains

- Find some good partitions
- What partitions are good?



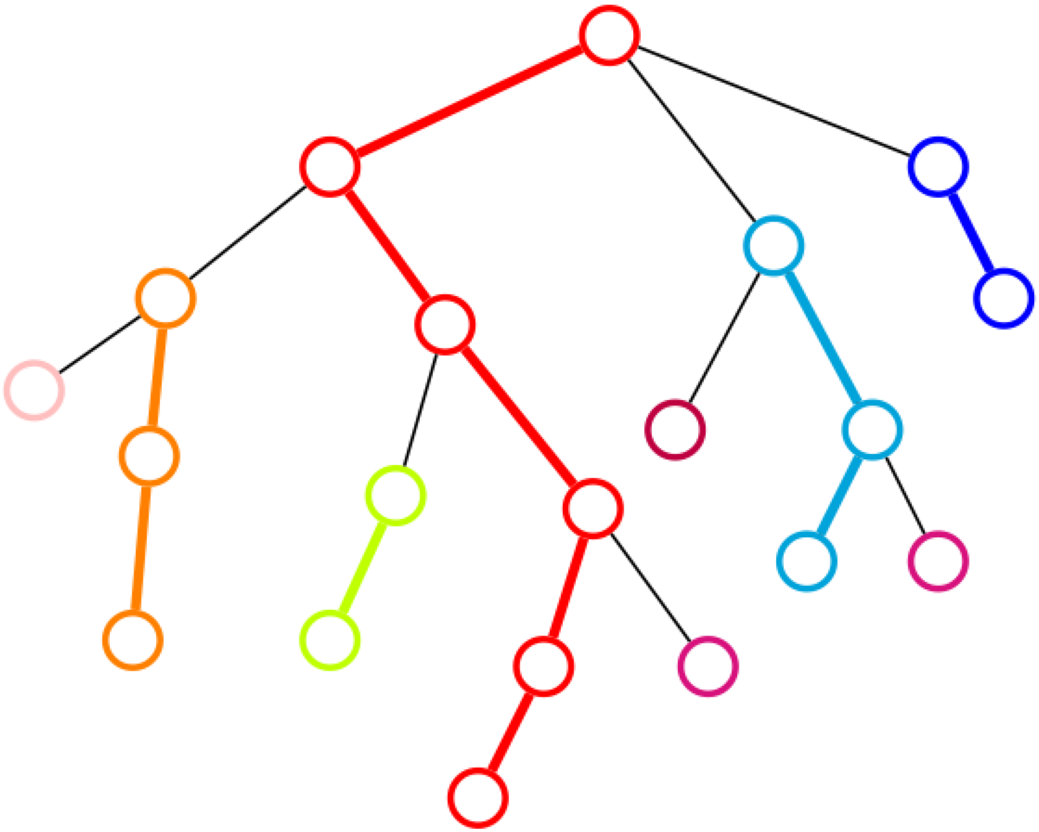
Heavy-Light Decomposition

- Heavy edges and light edges
- Heavy edges form heavy chains



So what?

- A path is composed of at most ***log N*** chains
- Subtree can still be a segment!



Implementation (precompute subtree size)

```
void dfssz( int u, int p ) {
  // precompute the size of each subtree
  par[ u ][ 0 ] = p ;
  sz[ u ] = 1 ;
  head[ u ] = u ;
  for( int v : g[ u ] ) if( v != p ) {
    dep[ v ] = dep[ u ] + 1 ;
    dfssz( v , u ) ;
    sz[ u ] += sz[ v ] ;
  }
}

// inside main()
int root = 1 ; // set root node
dep[ root ] = 1 ;
dfssz( root , root ) ;
```

Implementation (heavy-light decomposition)

```
void dfshl( int u ){
  tin[ u ] = ++ ts ;
  sort( g[ u ].begin() , g[ u ].end() ,
    [&]( int a , int b ){ return sz[ a ] > sz[ b ] ; } ) ;
  bool flag = 1 ;
  for ( int v : g[ u ] ) if ( v != par[ u ][ 0 ] ) {
    if ( flag ) head[ v ] = head[ u ] , flag = 0 ;
    dfshl( v ) ;
  }
  tout[ u ] = ts ;
}

// inside main()
ts = 0 ;
dfshl( root ) ;
```

Implementation (decompose a path)

```
vector< ii > getPath( int u , int v ) {
  // u must be ancestor of v
  // returns a list of intervals from v to u
  vector< ii > res ;
  while( tin[ u ] < tin[ head[ v ] ] ) {
    res.push_back( ii( tin[ head[ v ] ] , tin[ v ] ) ) ;
    v = par[ head[ v ] ][ 0 ] ;
  }
  if ( tin[ u ] + 1 <= tin[ v ] )
    res.push_back( ii( tin[ u ] + 1 , tin[ v ] ) );
  return res;
}
```

Implementation (answer querys)

```
// query operation
int vx[ 2 ] ;
scanf( "%d%d" , &vx[ 0 ] , &vx[ 1 ] ) ;
int z = lca( vx[ 0 ] , vx[ 1 ] ) ;
int ans = -INF ;
vector< ii > path ;
for ( int u : vx ) {
  path = getPath( z , u ) ;
  for ( ii pr : path ) {
    if ( pr.first > pr.second ) swap( pr.first , pr.second ) ;
    ans = max( ans , query( 1 , 1 , n , pr.first , pr.second ) ) ;
  }
}
printf( "%d\n" , ans ) ;
```

Centroid Decomposition

Problem

Given a tree $\boldsymbol{T}$, all nodes are white initially, make the following operations efficient.

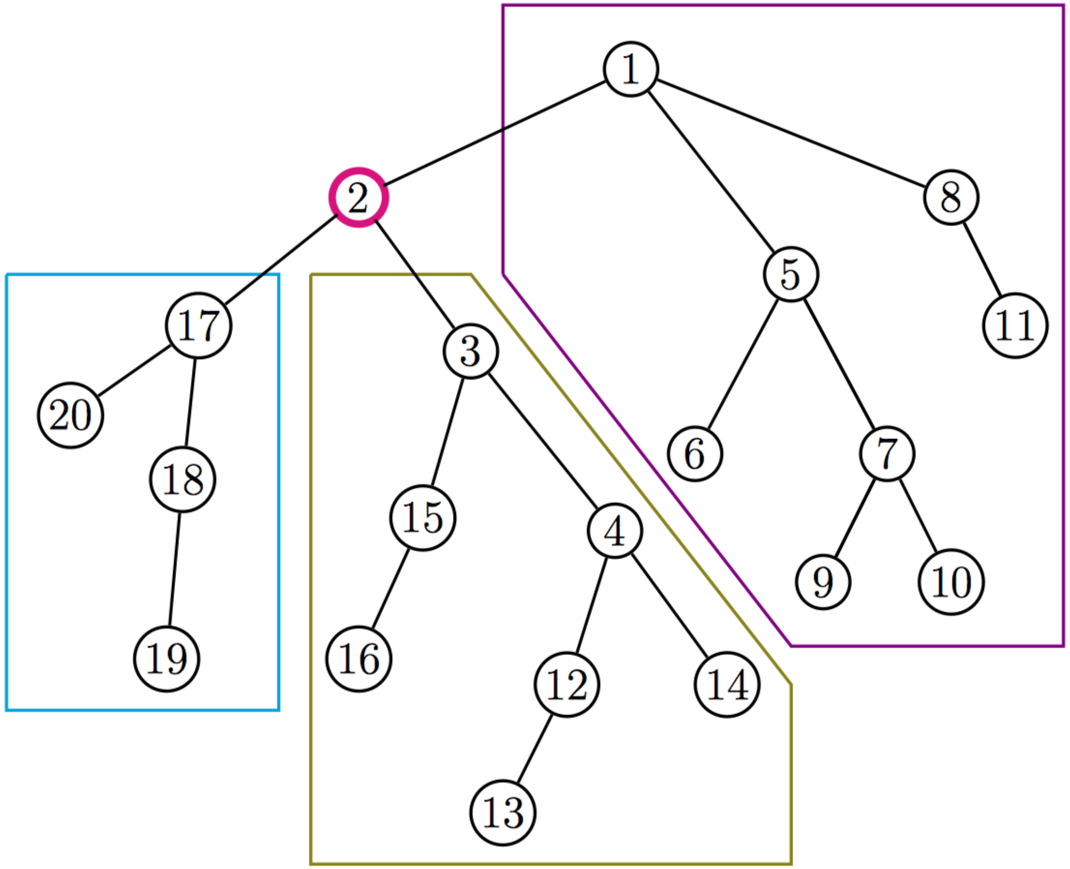
- Color a node black
- Given u, calculate the sum of distance from node u to all black nodes.

What's wrong with brute force?

- Each update takes constant time ***O(1)***
- For each query, there is ***O(N)*** other nodes, so need to consider ***O(N)*** paths

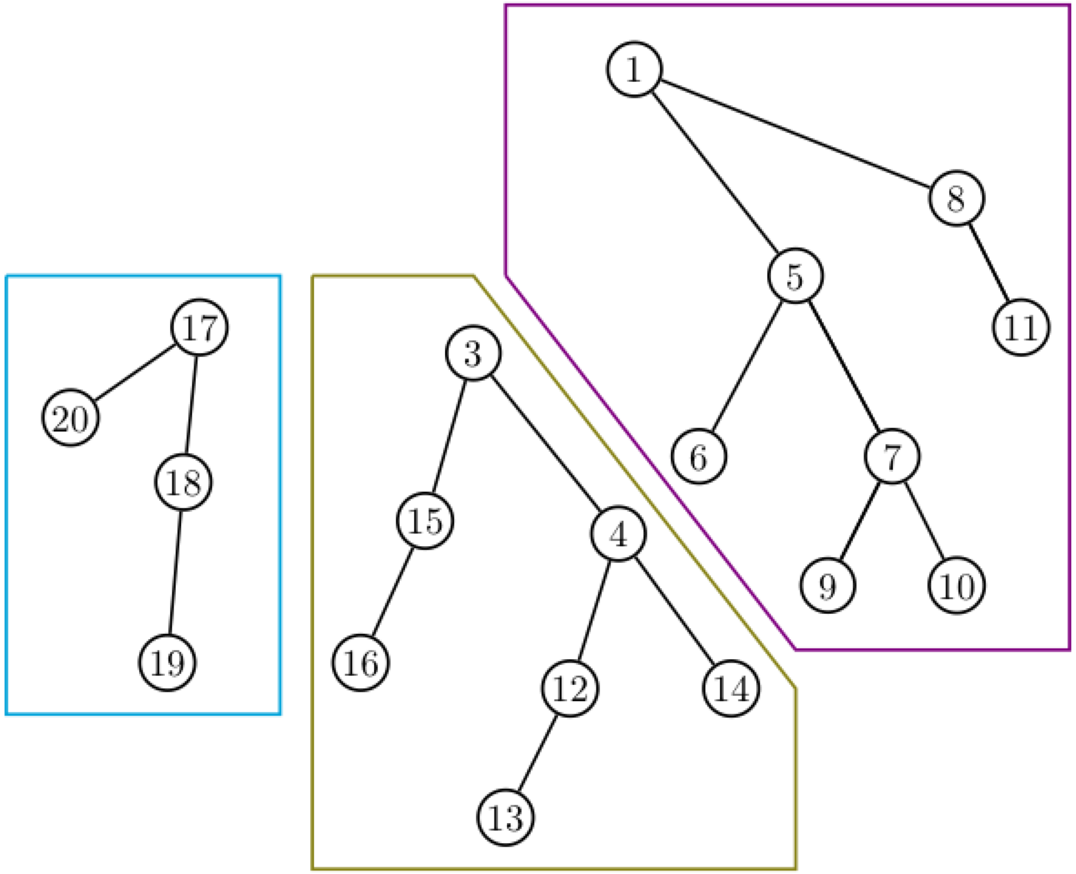
Centroid

The node such that removing it results in no tree of size $> \frac{n}{2}$



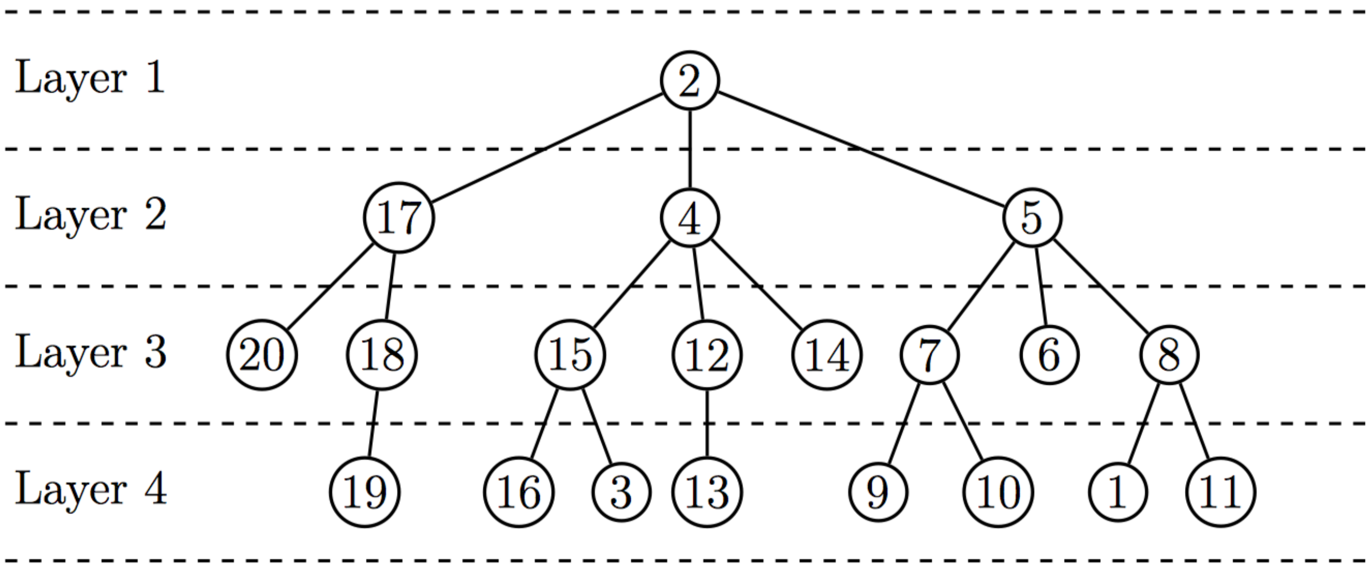
Divide and Conquer

Remove the centroid and repeat the same process on each resulting tree



Centroid Tree

Built through the process of Divide and Conquer



Properties from Divide and Conquer

- ***path(u,v)*** can be partitioned into ***path(u, lca(u,v))*** and ***path(lca(u,v), v)*** , where ***lca(u,v)*** is the LCA in the Centroid Tree
- So we only care about ***path(u,v)*** , where ***u*** is the ancestor of ***v*** in the Centroid Tree

Properties from Centroid

- In total, we only care about ***O(N log N)*** paths, ***O(log N)*** for each node

Implementation (decomposition-1)

```
int centroidDecomp( int x ) {
  // decompose the subtree and return the centroid
  vector< int > q ;
  { // bfs from arbitrary point to get bfs order for
    // later computation of subtree size and M(u)
    size_t pt = 0 ;
    q.push_back( x ) ;
    p[ x ] = -1 ;
    while ( pt < q.size() ) {
      int now = q[ pt ++ ] ;
      sz[ now ] = 1 ;
      M[ now ] = 0 ;
      for ( auto &pr : adj[ now ] ) {
        int nxt = pr.first ;
        if ( !vis[ nxt ] && nxt != p[ now ] ) {
          q.push_back( nxt ) , p[ nxt ] = now ;
        }
      }
    }
  }
```

Implementation (decomposition-2)

```
// calculate subtree size in reverse order
reverse( q.begin() , q.end() ) ;
for ( int& nd : q ) if ( p[ nd ] != -1 ) {
  sz[ p[ nd ] ] += sz[ nd ] ;
  maxify( M[ p[ nd ] ] , sz[ nd ] ) ;
}
for ( int& nd : q )
  maxify( M[ nd ] , (int)q.size() - sz[ nd ] ) ;

// find centroid
int centroid ;
for ( int &nd : q )
  if ( M[ nd ] + M[ nd ] <= (int)q.size() )
    centroid = nd ;

// path[ nd ] stores the nodes on the path from the root
// to "nd" on the centroid tree
// struct node( now , nxt , dis )
for ( int &nd : q ) {
  if ( path[ nd ].size() )
    path[ nd ].back().nxt = centroid ;
  path[ nd ].emplace_back( centroid , -1 , 0 ) ;
}
```

Implementation (decomposition-3)

```
  { // bfs from centroid to compute distance from all
    // nodes to the centroid, can also be done with LCA
    q.clear() ;
    size_t pt = 0 ;
    q.push_back( centroid ) ;
    p[ centroid ] = -1 ;
    while ( pt < q.size() ) {
      int now = q[ pt ++ ] ;
      long long ndis = path[ now ].back().dis ;
      for ( auto &pr : adj[ now ] ) {
        int nxt = pr.first ;
        long long cdis = pr.second ;
        if ( !vis[ nxt ] && nxt != p[ now ] ) {
          q.push_back( nxt ) , p[ nxt ] = now ;
          path[ nxt ].back().dis = ndis + cdis ;
        }
      }
    }
  }

  // decompose the tree recursively
  // set vis[ centroid ] = 1 to break the tree into forest
  vis[ centroid ] = 1 ;
  for ( auto &pr : adj[ centroid ] ) {
    int nxt = pr.first ;
    if ( !vis[ nxt ] ) centroidDecomp( nxt ) ;
  }
  return centroid ;
}
```

Implementation (update & query)

Need some math to avoid counting the same path twice

```
long long sum[ N ] ;
int tot[ N ] , cnt[ N ] ;
void mark( int x ) {
  for ( auto& nd : path[ x ] ) {
    int now = nd.now , nxt = nd.nxt ;
    long long dis = nd.dis ;
    sum[ now ] += dis ;
    tot[ now ] ++ ;
    if ( nxt != -1 ) {
      sum[ nxt ] -= dis ;
      cnt[ nxt ] ++ ;
    }
  }
}

long long query( int x ) {
  long long ret = 0 ;
  for ( auto& nd : path[ x ] ) {
    int now = nd.now , nxt = nd.nxt ;
    long long dis = nd.dis ;
    ret += sum[ now ] + dis * ( tot[ now ] - cnt[ nxt ] ) ;
  }
  return ret ;
}
```

Questions?

THE END