

# Data Structures and Algorithms

Conf. dr. ing. Guillaume Ducoffe

`guillaume.ducoffe@fmi.unibuc.ro`

# Operations on/with Trees

## Queries on Rooted Trees:

- Least-common ancestors
- Distances

## Tree Decompositions Methods

- Heavy-Path (HP)
- Centroid

## Tree-Based Data Structures:

- Cartesian Trees

# Least common ancestor (LCA) queries

Every two nodes  $x, y$  have a common ancestor (e.g., the root)

## Definition

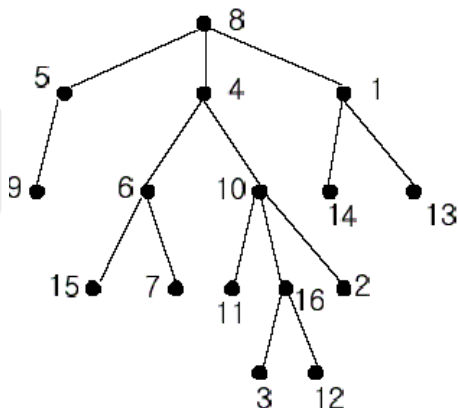
$LCA(x, y) =$  **deepest** common ancestor (furthest from the root)

Examples:

$$LCA(15, 7) = 6$$

$$LCA(15, 12) = 4$$

$$LCA(15, 9) = 8$$



Pre-processing time? Query time?

## Naive resolution

While nodes  $x$  and  $y$  are distinct, replace the deepest node by its parent.

```
node *lca(node *x, node *y) {  
    while(x != y){  
        if(x->level >= y->level)  
            x = x->father;  
        else y = y->father  
    }  
    return x; //==y  
}
```

Pre-processing:  $\mathcal{O}(n)$  – Computation of the levels

Query Complexity: Linear in the **height** of the tree

## Extremal cases

- Stars have height = 1



- But Paths ( $\sim$  Lists) have height =  $n - 1$



- Bounded-degree trees (e.g., Binary) have height in  $\Omega(\log n)$

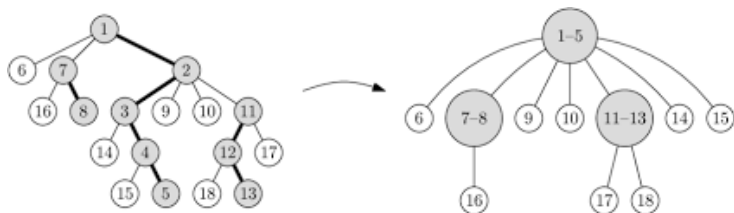


# Heavy-path decomposition

- For each non-leaf node, select a child with maximum number of descendants in its rooted subtree
  - variant: select the unique child  $y$  of  $x$  s.t.  $y.order > x.order/2$ , if any.

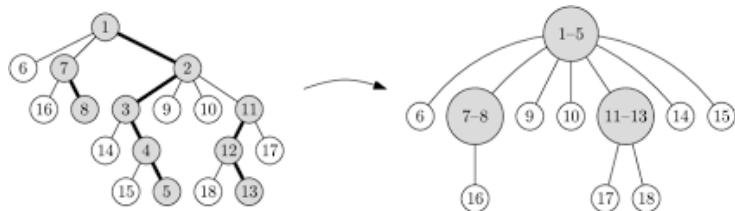
⇒ Partition of the nodes in so-called “heavy” paths.

HP-tree: rooted subtree obtained by contracting each HP in one node.



# Encoding

- We can associate to each node a number (ID of its HP).
- We can store in an array the highest/deepest node of each HP.



- We may also associate, to each node, a pointer to its child in the same HP (possibly, null).

# Implementation

*//k denotes the ID of the current HP*  
*//first and last store ends of each HP*  
*//hp denotes HP, next\_hp points to next child in HP*

```
void compute_hp(node *n, int& k, vector<node*>& first, vector<node*>& last){
    n->hp = k; last[k] = n;
    if(n->child != nullptr){
        node *c = n->child;
        for(node *p = c->next; p != nullptr; p = p->next)
            if(p->order > c->order) c = p;
        n->next_hp = c; compute_hp(c,k,first,last);
        for(node *p = n->child; p != nullptr; p = p->next)
            if(p != c) { //new HP
                first.push_back(p); last.push_back(p);
                compute_hp(p,++k,first,last);
            }
    }
}
```

Complexity:  $\mathcal{O}(n)$



# LCA queries with HP

## Logarithmic-time version

- Compute levels in the HP-tree.

```
void levels_hp(node *n, vector<int>& hp_lvl) {
    if(first[n->hp] == n) {
        if(n->father == nullptr) hp_lvl[n->hp] = 0;
        else hp_lvl[n->hp] = hp_lvl[n->father->hp] + 1;
    }
    for(node *c = n->child; c != nullptr; c = c->next) { levels_hp(c, hp_lvl); }
}
```

- Simulation of the naive algorithm on the HP-tree.

*//for ease of writing, we assume first and last to be global arrays*

```
node *lca_hp(node *x, node *y) {
    while(x->hp != y->hp){
        if(x->level >= y->level)
            x = ( (x == first[x->hp]) ? x->father : first[x->hp] );
        else y = ( (y == first[y->hp]) ? y->father : first[y->hp] );
    }
    return ( (x->level >= y->level) ? x : y );
}
```

Property: The HP-tree has height in  $\mathcal{O}(\log n)$ .

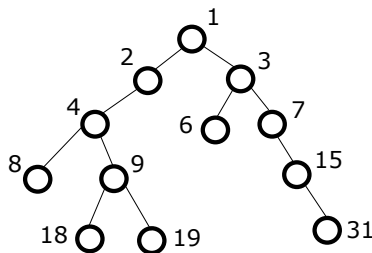
# Improvements: Binary Lifting

Binary tree of height in  $\mathcal{O}(\log n)$ .

1) Label the root with 1. If a node is labelled with  $i$ , then label its left (resp. right) child by  $2i$  (resp.,  $2i + 1$ ).

→ Keep the pairs (label,node) in a hash table

2) A node at level  $i$  has a  $(i + 1)$ -bit label. The  $j^{th}$  closest ancestor of a node  $x$  has label  $x.label \gg j$  (division by  $2^j$ )



Remark: if the LCA of two nodes  $x, y$  is at level  $i$ , then the  $i + 1$  most significant bits of their respective labels are identical.

## Binary Lifting: Application to LCA

**Input:** nodes  $x, y$

s.t.  $x.level \leq y.level$ .

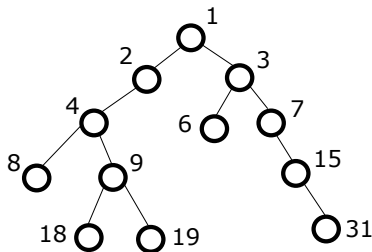
1)  $a = x.label$  and  $b = y.label$

2) If  $i = y.level - x.level$  is  $> 0$ , then  
 $b = b \gg i$  (division by  $2^i$ ).

3) Let  $c = a \oplus b$  (bitwise XOR)

4) Let  $j = \lfloor \log c \rfloor + 1$  (most significant bit)

5)  $LCA(x, y)$  is the node with label  $a \gg j$ .

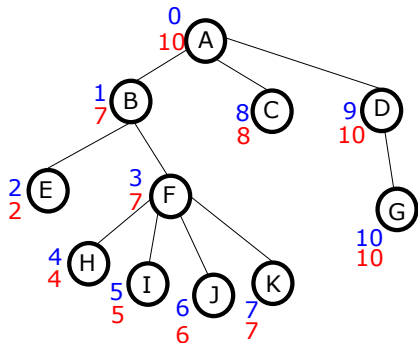


Complexity:  $\mathcal{O}(1)$  assuming bitwise operators (but  $\mathcal{O}(\log n)$  otherwise).

## Intermezzo: A problem of **intervals**

Consider a pre-order of the nodes.

**Observation:** a rooted subtree = an interval



$LCA(x,y)$ ,  $x < y \implies$  **shortest** interval that contains  $[x,y]$

## DFS computation

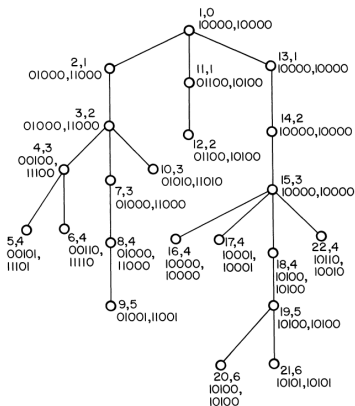
```
void compute_intervals(node *n, int& num){  
    n->start = num++;  
    for(node *c = n->child; c != nullptr; c = c->next)  
        compute_intervals(c,num);  
    n->end = num;  
}
```

Complexity:  $\mathcal{O}(n)$

Remark: Post-ordering = nodes ordered by increasing end value (break ties by decreasing height...)

# Binary Lifting + HP-tree

## Pre-processing



1) Compute a preorder + preorder-intervals [start,end] for each node

2) For each node  $v$ , let its **inlabel** denote the node in its subtree whose preorder number has maximum number of rightmost 0s in its binary representation.

3) **HP decomposition using inlabels**  
(same HP = same inlabel)

Pre-processing time:  $\mathcal{O}(n)$  + computation of inlabels

## Computation of inlabels

- We compute, for each node  $v$ , the number of rightmost 0s in  $v.preorder$ .

→ For that, we simulate the increment of a binary counter with a stack.

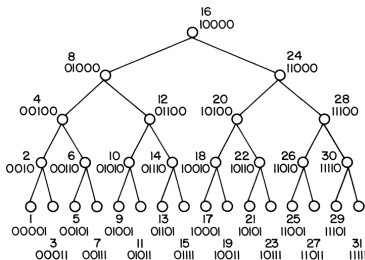
- We consider the nodes in preorder
- At the time we consider the  $i^{th}$  node, a stack  $S$  memorizes the position of all 1's in decreasing order (from right to left) in the binary representation of  $i - 1$ .

Incrementation: pop until we find two nonconsecutive 1's or  $S$  becomes empty. **Amortized complexity**  $\mathcal{O}(1)$

- Number of rightmost zeros = position of the rightmost zero (access to top element)
- Inlabels by dynamic programming on the tree.  $\implies \mathcal{O}(n)$

# Binary Lifting + HP-tree

## Properties



1) Identify the inlabels with the **inorder** numbers of some nodes in the smallest **complete binary tree**  $B$  with  $> n$  nodes.

2) **Descendance-preservation property**: if  $x$  is a descendant of  $y$ , then  $x.inlabel$  is a descendant of  $y.label$  (the converse is false in general)

3) In a complete binary rooted tree of order  $2^{h+1} - 1$ , for a node  $x$  at level  $i$ , we can compute its  $j^{th}$  closest ancestor by suppressing the  $i, i-1, \dots, i-j+1$  most significant bits and adding 0s to the right.

Consequences: if  $z = LCA(x, y)$ , then  $z.inlabel$  is an ancestor of  $w = LCA(x.inlabel, y.inlabel)$  in the complete binary rooted tree  $B$ .

Furthermore, we can compute  $w$  using binary lifting.



# Binary Lifting + HP-tree

Query (1/2)

**Input:** nodes  $x, y$  with  $x.inlabel \neq y.inlabel$ .

1) Compute  $LCA(x.inlabel, y.inlabel)$  in  $B$

- Compute the difference of level between  $x.inlabel$  and  $LCA(x.inlabel, y.inlabel)$   
(from the most significant bit of  $x.inlabel \text{ XOR } y.inlabel$ )
- Binary lifting: Compute  $a = (x.inlabel \ll j) \gg j$  (least significant bits),  $b = (x.inlabel - a)$  then  $c = b - ((b \ll k) \gg k)$  (suppression of the bit subsequence). Output  $c + (a \ll k)$ .

Remark: requires level of  $x.inlabel$  in  $B$ , that can be deduced from the number of rightmost 0s (already computed)

In what follows, let  $w = LCA(x.inlabel, y.inlabel)$

## The **ascendant** numbers

Reminder: for every node  $r$ , we know the level of  $r.inlabel$  in  $B$

- We define  $r.ascendant$  so that its  $j^{th}$  bit is set to 1 if and only if it has an ancestor  $s$  in  $T$  whose  $inlabel$  is at level  $j$  in  $B$ .

**We can compute the ascendant numbers by dynamic programming on the tree  $T$ .**

*Sketch*: Let  $p$  denote the parent of  $r$ . We obtain  $r.ascendant$  from  $t.ascendant$  by setting one new bit to 1 (namely, the level of  $r.inlabel$  in  $B$ ). This can be done using bitwise operators:

$$r.ascendant = t.ascendant + (1 \ll i)$$

Additional pre-processing time:  $\mathcal{O}(n)$

# Binary Lifting + HP-tree

## Query (2/2)

**Input:** nodes  $x, y$  with  $x.inlabel \neq y.inlabel$ .

2) Let  $z = LCA(x, y)$  (we do not know  $z$  at this point). In order to compute  $z.inlabel$  from  $w$ , we compute the level of  $z.inlabel$  in  $B$ .

→ XOR on  $x.ascendant, y.ascendant$

3) To compute  $z$ , we need to find the closest ancestors of  $x, y$  in the heavy-path corresponding to  $z.inlabel$ . Let us detail for  $x$ . We assume  $x.inlabel \neq z.inlabel$ . Let  $P_x, P_z$  denote the two HPs of  $x, z$ . We just need to find the neighbour of  $P_z$  on the unique  $P_z P_x$ -path in the HP-tree.

→ The inlabel of this HP can be retrieved from  $x.ascendant$ ! (again using bitwise operators).

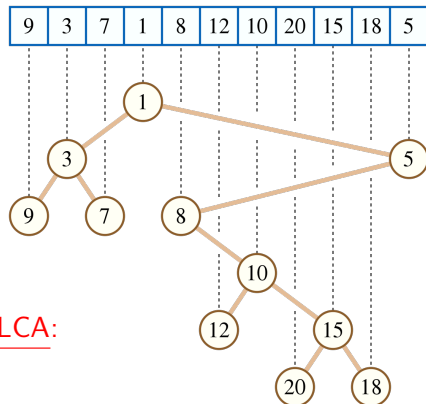
Query time:  $\mathcal{O}(1)$  using bitwise operators, and  $\mathcal{O}(\log n)$  otherwise.

## An application to Range queries: Cartesian trees

**Input:** an  $n$ -size vector  $v[]$

**Output:** an  $n$ -node binary rooted tree  $T$  whose nodes are the elements of  $v[]$ , such that:

- The root is a min/max element
- the left/right subtrees are Cartesian trees for the left/right subvectors to the root.



Reduction of min/max range queries to LCA:

The min/max elements between indices  $i$  and  $j$  is the element in position  $LCA(i,j)$ .

# Construction

- It suffices to compute the vector `father` encoding the parent of each node.
- We scan the vector from left to right and maintain potential candidates for parent nodes in a stack. We repeatedly pop out smaller/bigger elements.  $\implies$  **left neighbour**

Interpretation: if the father of a  $i$  is at an earlier position  $j < i$ , then it must be its left neighbour.

- Compute the **right neighbours** in the same way by scanning from right to left.
- Father of a node: the largest/least value amongst its left and right neighbours.

# Implementation

Min. version

```
vector<int> compute_cartesian_tree(const vector<int>& v) {  
    vector<int> left(v.size());  
    stack<int> candidates;  
    for(int i = 0; i < v.size(); i++) {  
        while(!candidates.empty() && v[candidates.top()] > v[i])  
            candidates.pop();  
        left[i] = (candidates.empty()) ? -1 : candidates.top();  
        candidates.push(i);  
    }  
    /* right neighbours */  
    vector<int> father(v.size());  
    for(int i = 0; i < v.size(); i++) {  
        father[i] = (left[i] <= right[i]) ? right[i] : left[i];  
    }  
    return father;  
}
```

Complexity:  $\mathcal{O}(n)$  (Potential: size of the stack)

## Beyond LCA: Distance queries

- Distance between two nodes  $x$  and  $y$ : number of edges on the (unique) path in the tree between  $x$  and  $y$
- Application: Routing in Tree Networks (or in general networks using a spanning trees)
- All distances can be computed in  $\mathcal{O}(n^2)$  time by varying the root and applying BFS.

*Can we do better?*

## Reduction to LCA

**Input:** nodes  $x, y$ .

1) Let  $z = LCA(x, y)$ .

2) Node  $z$  must be on the  $xy$ -path

$$\implies d(x, y) = d(x, z) + d(z, y).$$

3) Since  $z$  is an ancestor of  $x$  (resp.,  $y$ ), we have

$$d(x, z) = x.height - z.height \text{ (resp., } d(y, z) = y.height - z.height).$$

Pre-processing:  $\mathcal{O}(n)$  (compute the heights + LCA pre-processing)

Query time:  $\mathcal{O}(1)$



# Centroids in trees

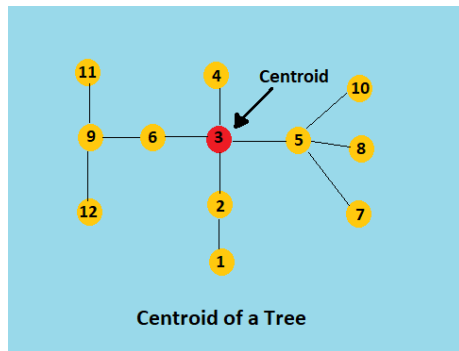
## Definition

A centroid in an  $n$ -node tree  $T$  is a node  $c$  such that every subtree of  $T \setminus \{c\}$  has order  $\leq n/2$ .

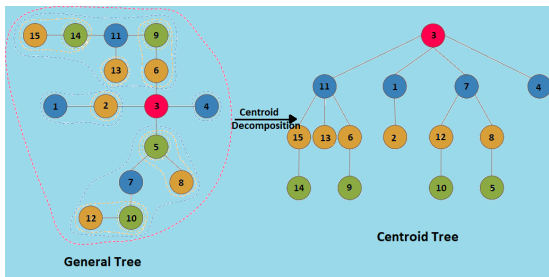
A centroid always exists.

Proof:

- 1) There are  $n - 1$  edges (all nodes but the root have a parent)
- 2) For each node  $n$ , choose a heaviest subtree of  $T \setminus \{n\}$  and orient the edge from  $n$  to this subtree.
- 3) One edge is oriented in both directions!



# Centroid decomposition



- 1) Compute a centroid  $c$
- 2) Compute a centroid decomposition for each subtree of  $T \setminus \{c\}$   
(output = rooted tree)
- 3) Merge all decompositions in one rooted tree with  $c$  as root.

The result is a rooted tree  $T'$  with same node-set as the original tree  $T$ .

**Property:** the centroid decomposition outputs a tree of height in  $\mathcal{O}(\log n)$

Application to LCA/Distance queries: store for each node its path-to-root in the centroid decomposition + distances in  $T$ .

# Computation

- A centroid can be computed in  $\mathcal{O}(n)$  time.
- Pre-compute the order of each rooted subtree.
- **Local search.** Start from any node. If each subtree of  $T \setminus \{c\}$  has order  $\leq n/2$  then output  $c$ . Otherwise, go to a neighbour in a heaviest subtree.

Remark: for each child  $c'$  of  $c$  we know the order of its rooted subtree. If  $c'$  is the parent of  $c$ , then the subtree of  $T \setminus \{c\}$  that contains  $c'$  has order  $n - c - > \text{order}$ .

- There are  $\mathcal{O}(\log n)$  recursive steps
- $\implies$  Centroid decomposition in  $\mathcal{O}(n \log n)$  time.

# Questions

