Data Structures and Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Dynamic Trees

Objectives: Techniques for supporting some generic reorganizations of Trees/Tree-based DS.

$\rightarrow$ We mostly considered static trees or insertion/deletion of one node.

1) Restructure of Binary Research Trees

- Join/Split operations
- Implementations for self-balanced BRTs
- Splay trees

2) Dynamic maintenance of a forest of trees:

- Link/cut trees
- Euler tour trees

Join/Split operations

1) **Join**:

- Input: an element i + 2 BRTs T_1, T_2.
- Assumption: $T_1.max < i < T_2.min$.
- Output: a new BRT T that contains all elements from $T_1 \cup T_2 \cup \{i\}$.

2) **Split**

- Input: a BRT T and an element i
- Output: BRTs T_1, T_2 such that all elements $< i$ (resp., $> i$) in T are put in T_1 (resp., in T_2).

Naive implementation: **Join**

- Make a new BRT with root i and left/right subtrees T_1, T_2.

```
typedef node *BRT;

BRT& join(int i, BRT& T1, BRT& T2) {
   node *n = new node;
   n->value = i;
   n->father = nullptr;
   n->left = T1; n->right = T2;
   T1->father = T2->father = n;
   return n;
}
```

Complexity: $\mathcal{O}(1)$.

Drawback: does not preserve most self-balancedness properties (*e.g.*, AVL, path-balanced,etc.)

Naive implementation: **Split**

- Search for i in the BRT. Repeatedly join the subtrees with smaller (resp., larger) values than i.

```
pair<BRT,BRT> split(BRT& T, int i) {
    pair<BRT,BRT> output(nullptr,nullptr);

    if(!empty(T)) {
        if(T->value == i) {
            output->first = T->left; output->second = T->right;
        } else if(T->value < i) {
            output = split(T->right,i);
            output->first = join(T->value,T->left,output->first);
        } else{
            output = split(T->left,i);
            output->second = join(T->value,output->second,T->right);
        }
    }

    return output;
}
```

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

AVL Implementation: **Join**

1) If the heights of both subtrees T_1, T_2 differ by at most 1 (in absolute value) then we do as for the naive implementation.

2) Otherwise, if T_1 has the largest height, then:

- Join with T_1`->right`, i, T_2
- Self-balance T_1 (using at most two rotations).

3) Otherwise, if T_2 has the largest height, then ...

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

finer-grained analysis: linear in $|height(T_1) - height(T_2)|$

AVL Implementation: **Split**

It suffices to apply the naive implementation (but using the AVL implementation for Join).

Analysis:

- The "left" tree T_1 is obtained by repeatedly joining left subtrees $T_1^L, T_2^L, \ldots, T_k^L$ on the search path to value i.
- Telescopic sum for the complexity: $\sum_j |height(T_j^L) - height(T_{j+1}^L)| \leq \sum_j (1 + height(T_j^L) - height(T_{j+1}^L))$.
- Overall complexity in $height(T_1^L) + k = \mathcal{O}(\log n)$.
- Same analysis for the "right" tree T_2.

Red-black tree implementation

Definition (Ranks)

- The rank of a leaf equals 0
- The rank of a black node (other than the root) is one less than the rank of its father node.
- The rank of a red node equals the rank of its father node.

Interpretation: rank in red-black tree = depth of rooted subtree in a corresponding $2-3-4$ tree.

Observation: a node is black if and only if either it is the root or its rank is one less than for its father node.

Red-black tree Implementation: **Join**

Let r_i be the rank value for the root of T_i.

Case 1: $r_1 = r_2 = r$.

$\Longrightarrow$ Do as for the naive implementation. The new (black) root has value i and rank $r + 1$.

Case 2: $r_1 < r_2$.

- Join T_1, i, T_2`->left`.

  (we stop after some recursive calls on a **black node** of T_2 with rank r_1)

- Rebalance if needed (as for insertions).

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

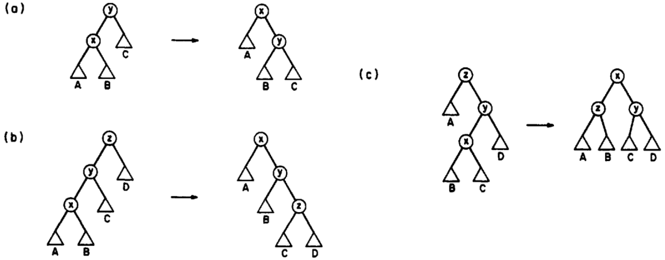
finer-grained analysis: $\mathcal{O}(|r_1 - r_2|)$.

$\Longrightarrow$ Split as before.

The Splay operation

- An heuristic for splitting a BRT T using some value i (assumed to be present in T).

- Algorithm:
  - Locate the node x whose value is equal to i
  - Make of x the root of T by repeated left or right rotations **in pairs** (here called "zig" or "zag").



Complexity: $\mathcal{O}(height)$.

Implementation

```
void splay(BRT& T, int i) {
    //Phase 1: Locate the node x which contains value i
    while(T->value != i)
        if(T->value < i) T = T->right;
        else T = T->left;
    //Phase 2: Zig-zag until the root
    while(T->father != nullptr)
        if(T->father->father == nullptr) //Case (a)
            if(T == T->father->left) rotateRight(T->father);
            else rotateLeft(T->father);
        else if(T == T->father->left && T->father->father->left == T->father){
            rotateRight(T->father->father); rotateRight(T->father);
        } else if(T == T->father->right && T->father->father->right == T->father){
            rotateLeft(T->father->father); rotateLeft(T->father);
        } else { // Case (c)
            if(T->father->left==T) {
                rotateRight(T->father); rotateLeft(T->father->father);
            } else{
                rotateLeft(T->father); rotateRight(T->father->father);
            }
        }
}
```

Potential function Analysis

- Associate a weight $w(x)$ to any node x (for simplicity, $w(x) = 1$).
- Total weight $tw(x)$ = sum of weights $w(y)$ for all descendants y of x.
- Let $rank(x) =^{def} \lfloor \log tw(x) \rfloor$

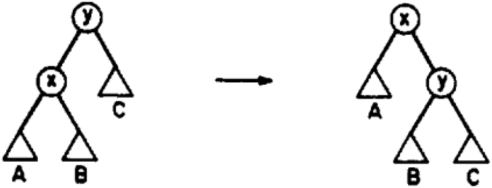
We associate to any BRT the following potential:

$$\Phi = \sum_x rank(x)$$

Observation: If $\forall x,\ w(x) = 1$, then $\Phi = \mathcal{O}(n \log n)$.

Amortized complexity of Splay: Case (a)

(a)



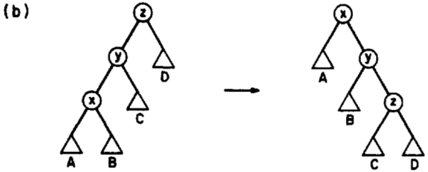
We have $rank'(x) = rank(y)$ and $rank'(y) \leq rank(y)$.

Change of potential:

$$\Phi' - \Phi = rank'(y) - rank(x) \leq rank'(x) - rank(x)$$

Amortized cost: $1 + rank'(x) - rank(x) \leq 1 + 3 \cdot (rank'(x) - rank(x))$

Amortized complexity of Splay: Case (b)



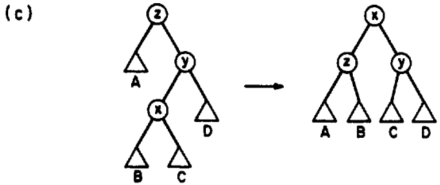
Change of potential:

$$
\begin{aligned}
\Phi' - \Phi &= \mathit{rank}'(x) + \mathit{rank}'(y) + \mathit{rank}'(z) - (\mathit{rank}(x) + \mathit{rank}(y) + \mathit{rank}(z)) \\
&\leq \mathit{rank}'(y) + \mathit{rank}'(z) - \mathit{rank}(x) - \mathit{rank}(y) \leq \mathit{rank}'(y) + \mathit{rank}'(z) - 2 \cdot \mathit{rank}(x) \\
&\leq 2 \cdot (\mathit{rank}'(x) - \mathit{rank}(x))
\end{aligned}
$$

Either $\mathit{rank}'(x) > \mathit{rank}(x)$, or $\mathit{rank}'(x) = \mathit{rank}(x) = \mathit{rank}(z) > \mathit{rank}'(z)$ and then $\Phi' - \Phi < 0$.

Amortized cost: $3 \cdot (\mathit{rank}'(x) - \mathit{rank}(x))$.

Amortized complexity of Splay: Case (c)



Change of potential:

$$\begin{aligned}\Phi' - \Phi &= \mathit{rank}'(x) + \mathit{rank}'(y) + \mathit{rank}'(z) - (\mathit{rank}(x) + \mathit{rank}(y) + \mathit{rank}(z)) \\ &\leq 2 \cdot (\mathit{rank}'(x) - \mathit{rank}(x))\end{aligned}$$

Either $\mathit{rank}'(x) > \mathit{rank}(x)$, or $\mathit{rank}'(x) > \min\{\mathit{rank}'(y), \mathit{rank}'(z)\}$ and then $\Phi' - \Phi < 0$.

Amortized cost: $3 \cdot (\mathit{rank}'(x) - \mathit{rank}(x))$.

Splay tree

- *Split is done using the Splay heuristic.*

<u>Amortized complexity</u>: $3 \cdot (rank(root) - rank(x)) + 1 = \mathcal{O}(\log n)$.

Splay tree

- *Split is done using the Splay heuristic.*

Amortized complexity: $3 \cdot (rank(root) - rank(x)) + 1 = \mathcal{O}(\log n)$.

- *Join is done using the naive implementation.*

Amortized complexity: $\mathcal{O}(\log n)$

Splay tree

- *Split is done using the Splay heuristic.*

Amortized complexity: $3 \cdot (rank(root) - rank(x)) + 1 = \mathcal{O}(\log n)$.

- *Join is done using the naive implementation.*

Amortized complexity: $\mathcal{O}(\log n)$

- *After inserting a new element at some (new) node x, we splay.*

The root-to-x path is scanned twice: insertion + splay

Amortized complexity: Increase of Φ after insertion + $\mathcal{O}(\log n)$.

Only nodes with rank $2^r - 1$ can increase their rank after insertion . . . .

$\Longrightarrow \mathcal{O}(\log n)$

Splay tree

- *Split is done using the Splay heuristic.*

Amortized complexity: $3 \cdot (rank(root) - rank(x)) + 1 = \mathcal{O}(\log n)$.

- *Join is done using the naive implementation.*

Amortized complexity: $\mathcal{O}(\log n)$

- *After inserting a new element at some (new) node x, we splay.*

The root-to-x path is scanned twice: insertion + splay

Amortized complexity: Increase of Φ after insertion + $\mathcal{O}(\log n)$.

Only nodes with rank $2^r - 1$ can increase their rank after insertion . . . .

$\Longrightarrow \mathcal{O}(\log n)$

- *Deletion: Reduction to Split and Join*

Amortized complexity: $\mathcal{O}(\log n)$

A more general case: Dynamic forests

- The data is stored in a collection of rooted trees
- Possible queries/operations:
  - `node *findRoot(node *N)`: outputs the root of the tree containing `N`

    $\Longrightarrow$ Allows to check whether two nodes are in the same rooted tree.
  - **void** `cut(node *N)`: disconnects `N` from its parent (thus creating a new tree with root `N`).
  - **void** `link(node *N1, node *N2)`: makes `N1` the parent of `N2`.

    Requires: `N1`,`N2` are in $\neq$ trees and `N2` is a root.

    We may also include an operation **void** `reroot(node *N)`: that makes of node `N` the root of its tree.

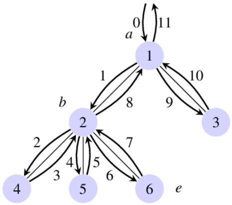
Euler tour

Definition (Euler tour)

Apply a DFS to a rooted tree.

An edge uv is visited each time we go from one of its two end-nodes (u or v) to its other end-node.

We enumerate visited edges one by one.



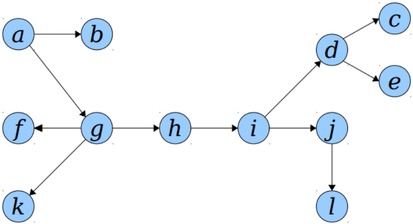
Fundamental Property: Edges uv of the tree are traversed exactly twice: once from u to v, once from v to u.

Euler tour trees

Each tree of the forest is represented by an **Augmented Euler tour**: we insert loops (v, v) for each node.

- We count each loop only once (no repetition).
- Each loop (v, v) is inserted in the tour at some arbitrary visit of node v during the DFS.

$\Longrightarrow$ Preorders/Postorders if we always insert at first/last visit (but difficult to maintain after each operation...).

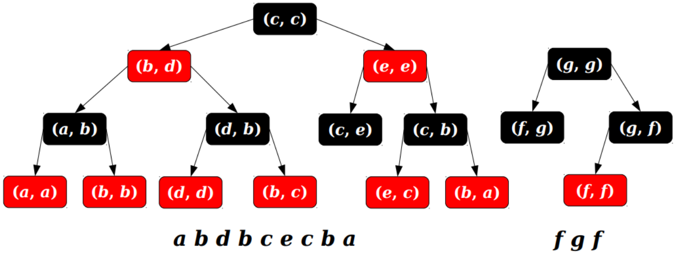


`(a,a),(a,b),(b,b),(b,a),(a,g),(g,g),(g,f),(f,f),(f,g),(g,k),(k,k),.`

Encoding

An augmented Euler tour implicitly defines a total ordering over loops and oriented edges (*i.e.*, traversal order).

$\Longrightarrow$ We can store each tour in a self-balanced binary research tree!



<u>Technical remark</u>: we do <u>not</u> allow explicit comparisons between elements (= loops/edges).

The ordering is implicitly preserved by the operations (*e.g.*, rotations)

Finding the root

- For each node `N`, we store a pointer to the loop `(N,N)`.
- We access to the root of the self-balanced BRT which contains `(N,N)`.
- We repeatedly go left in order to find the minimum

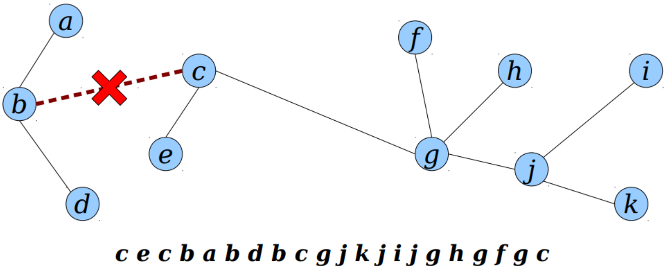
  $=$ the first edge of the tour, whose head equals `root`.

Complexity: $\mathcal{O}(\log n)$ – because we use a self-balanced BRT.

Cutting an edge

- Remove the edges `(N->father,N)` and `(N,N->father)` from the tour.

Requires two Split operations on the BRT

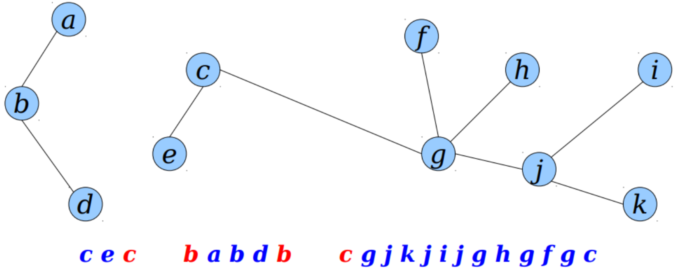


Cutting an edge

- Remove the edges `(N->father,N)` and `(N,N->father)` from the tour.

  Requires two Split operations on the BRT

- We get three BRTs T_1, T_2, T_3.



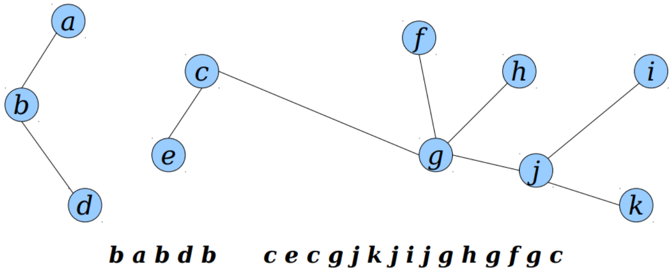
Cutting an edge

- Remove the edges `(N->father,N)` and `(N,N->father)` from the tour.

  Requires two Split operations on the BRT

- We get three BRTs T_1, T_2, T_3.

- T_2 represents the tree containing `N`. The tree containing `N->father` is represented by the **join** of T_1, T_3.



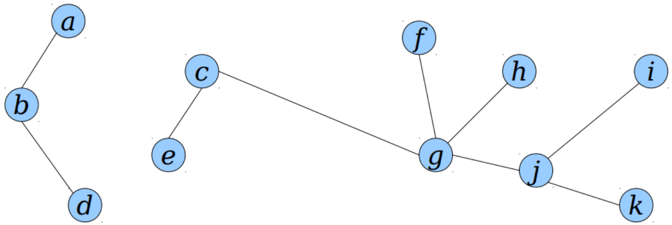
Cutting an edge

- Remove the edges `(N->father,N)` and `(N,N->father)` from the tour.

  Requires two Split operations on the BRT

- We get three BRTs T_1, T_2, T_3.

- T_2 represents the tree containing `N`. The tree containing `N->father` is represented by the **join** of T_1, T_3.

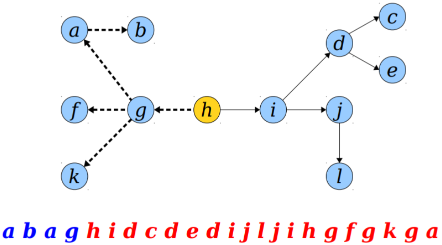


b a b d b **c e c g j k j i j g h g f g c**

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

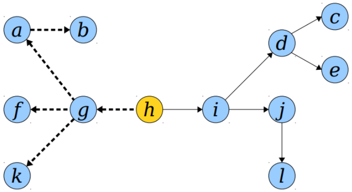
Rerooting

- Split the BRT using `(N->father,N)` – `N` is the intended new root.



Rerooting

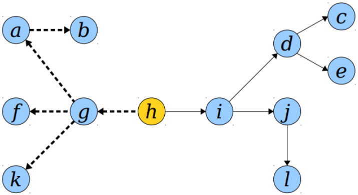
- Split the BRT using `(N->father,N)` – `N` is the intended new root.
- We obtain two BRTs T_1, T_2



a b a g h i d c d e d i j l j i h g f g k g a

Rerooting

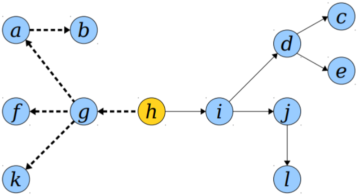
- Split the BRT using `(N->father,N)` – `N` is the intended new root.
- We obtain two BRTs T_1, T_2
- Join of T_2, T_1 (we revert left and right) then re-insertion of `(N->father,N)`



h i d c d e d i j l j i h g f g k g a b a g h

Rerooting

- Split the BRT using `(N->father,N)` – `N` is the intended new root.
- We obtain two BRTs T_1, T_2
- Join of T_2, T_1 (we revert left and right) then re-insertion of `(N->father,N)`



h i d c d e d i j l j i h g f g k g a b a g h

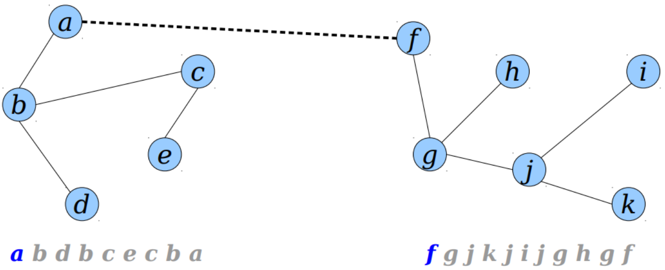
Complexity: $\mathcal{O}(\log n)$.

Link operation

- Let T_1, T_2 be the BRTs representing the trees of `N1,N2`.
- Split T_1 w.r.t. `(N1,N1)`. We obtain two subtrees T_1^L, T_1^R.
- Join T_2, `(N2,N1)`, T_1^R and insert `(N1,N2)` as min. element

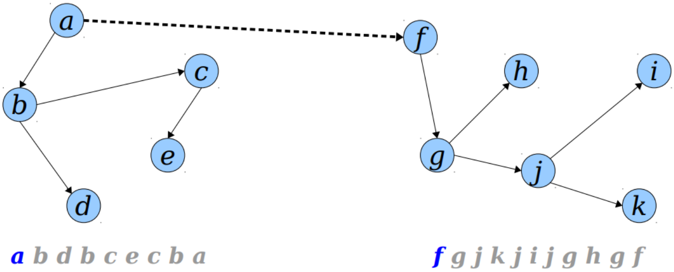
$\rightarrow T_3$

- Join T_1^L, `(N1,N1)`, T_3.



Link operation

- Let T_1, T_2 be the BRTs representing the trees of `N1,N2`.
- Split T_1 w.r.t. `(N1,N1)`. We obtain two subtrees T_1^L, T_1^R.
- Join T_2, `(N2,N1)`, T_1^R and insert `(N1,N2)` as min. element $\rightarrow T_3$
- Join T_1^L, `(N1,N1)`, T_3.

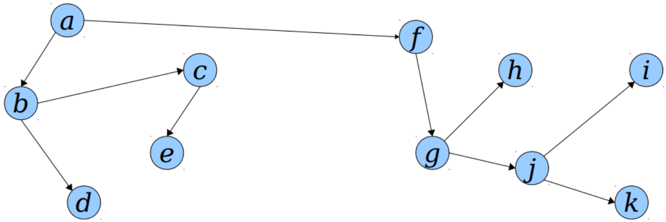


Link operation

- Let T_1, T_2 be the BRTs representing the trees of `N1,N2`.
- Split T_1 w.r.t. `(N1,N1)`. We obtain two subtrees T_1^L, T_1^R.
- Join T_2, `(N2,N1)`, T_1^R and insert `(N1,N2)` as min. element

$\rightarrow T_3$

- Join T_1^L, `(N1,N1)`, T_3.



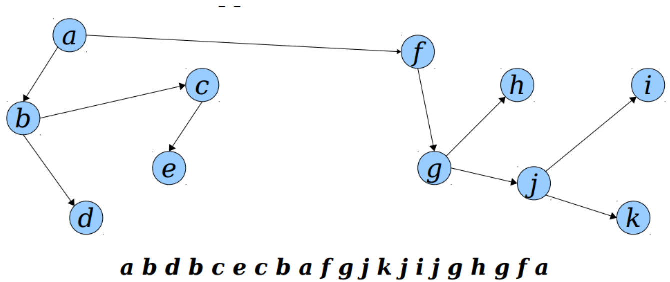
a b d b c e c b a f g j k j i j g h g f a

Link operation

- Let T_1, T_2 be the BRTs representing the trees of `N1,N2`.
- Split T_1 w.r.t. `(N1,N1)`. We obtain two subtrees T_1^L, T_1^R.
- Join T_2, `(N2,N1)`, T_1^R and insert `(N1,N2)` as min. element

$\rightarrow T_3$

- Join T_1^L, `(N1,N1)`, T_3.



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

Link-cut Tree

An alternate DS for dynamic forests which also supports the computation of several **aggregates**, *e.g.*:

- The minimum value stored on a path between a given node and its root.
- The maximum value stored on a path between a given node and its root.
- The sum of all values stored on a path between a given node and its root.
- ...

Informally, link-cut trees allow us to perform all the classic queries that we know how to do on static trees!

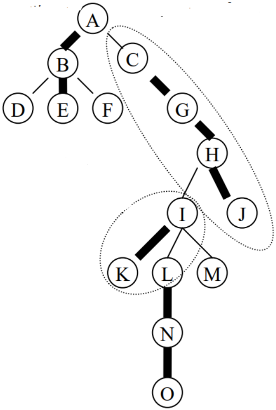
Drawback: Operations only have *amortized* complexity in $\mathcal{O}(\log n)$. This can be made worst-case also in $\mathcal{O}(\log n)$ but at the expense of a more complicated implementation.

Encoding

- The nodes of each tree are partitioned in a set of paths, called “preferred paths”.

- These paths are changing after each operation (their maintenance is explained in the next slides).

- Each path is stored in a splay tree (we order nodes by levels).

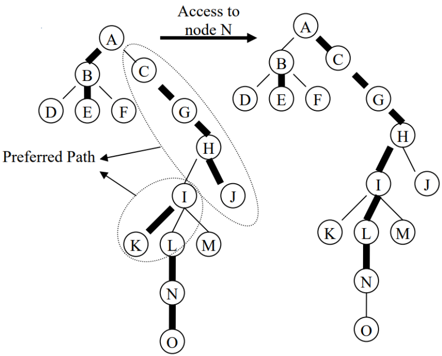


Preferred-path decomposition

• Any operation on a link-cut tree involves one node (cut,find root, aggregate) or two nodes (link).

• A node is accessed if it is the input of one such operation.

• Whenever we access a node `N`, we create a new preferred path, that contains all nodes on the path between `N` and its root. This effectively splits the former preferred paths to which these nodes were belonging.



Access to a node: implementation

Let N_0 be a pointer to the node which we want to access.

- While N_i is a valid (not null) pointer, we proceed as follows:
  - We keep a pointer to N_i in the splay tree of its current preferred path.
  - We split this splay tree at N_i
  - We find the root R_i of the current preferred path (min. in the splay tree).
  - Set N_{i+1} as a pointer to the parent node of R_i.
- We end up joining the splay trees of all nodes $N_0, N_1, \ldots, N_i, \ldots$.

Complexity (First estimation): $\mathcal{O}(\log n)$ time per splay tree.

$\Rightarrow \mathcal{O}(\log n)$ time per new preferred edge.

The number of new preferred edges

1) Let us consider a **heavy-path decomposition** for each tree in the DS.

Disclaimer: HP $\neq$ Preferred paths

We do *not* compute this HP (it's just for the analysis).

2) Let us consider a sequence of m consecutive accesses.

- At each access there are at most $\mathcal{O}(\log n)$ "light edges" (*i.e.*, not in a HP) that become preferred.
- If a "heavy edge" becomes preferred, then either it was *never* preferred before, or it *cancels* a preferred light edge.

$\Rightarrow$ at most $n - 1 + 2m \log n$ preferred edges.

The (amortized) number of new preferred edges is in $\mathcal{O}(\log n)$ (if the number of accesses goes large enough)

$\Rightarrow$ Access in amortized $\mathcal{O}(\log^2 n)$

An improved analysis

- Let $N_0 = N, N_1, \ldots, N_p = root$ be the nodes on the new preferred path.
- Let $s(u_i)$ be the size of the subtree rooted at u_i (rank in the splay tree).
  - Splaying at u_i in amortized $3 \cdot (\log s(u_i) - \log s(u_{i+1})) + 1$.
  - Amortized cost in $\mathcal{O}(\log s(root)) +$ new preferred edges $= \mathcal{O}(\log n)$.

Remark: The analysis breaks down for other choices of self-balance binary search trees.

(Worst-case implementation in $\mathcal{O}(\log n)$ is also possible but at the expense of a more complicated implementation...)

Operations

- `FindRoot` Can be reduced to accessing node `N`.

Amortized complexity: $\mathcal{O}(\log n)$.

- `Cut` an edge between a node `N` and its parent `M` consists in:
  - Access to `N`
  - Removing the edge between `N` and `M`. It splits their preferred path. We split the splay tree at either `N` or `M`.
  - We now have two trees: T_1, T_2 (that contains `M` and `N`, resp.).

Amortized complexity: $\mathcal{O}(\log n)$.

Operations cont'd

- `Link` a root `N` to a node `M` (in another tree) consists in:
  - Access to `M`
  - Adding an edge between `M` and `N`.

Amortized complexity: $\mathcal{O}(\log n)$.

Operations cont'd

- `Link` a root `N` to a node `M` (in another tree) consists in:
  - Access to `M`
  - Adding an edge between `M` and `N`.

Amortized complexity: $\mathcal{O}(\log n)$.

- Computation of aggregates (min/max/sum)
  - Access to a node `N`
  - Each node stores the aggregate (max/min/sum of all elements) for all its descendants *in the splay tree*.
  - This information can be updated "for free" during each join/split operation.

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

Questions