Table des matières

1 Introduction

Ayant été confronté à plusieurs problèmes sur les arbres auxquels je cherchais une solution, je suis tombé par hasard sur des articles mentionnant des arbres dynamiques. J'ai donc étudié quelques formes de ces derniers dans le but de résoudre un problème, qui fera l'objet de la 3^{eme} partie.

1.1 Énoncé du problème

On considère un arbre enraciné pondéré à N nœuds. On souhaite représenter cet arbre de manière à pouvoir traiter les requêtes suivantes : couper ou ajouter une arête, répondre à une question concernant les pondérations sur un chemin d'un nœud A à un nœud B.

1.2 Simplification

Supposons dans un premier temps que la structure d'arbre soit fixe, et que les requêtes soient uniquement des modifications du poids des arêtes, ou des questions. Nous appellerons ce type d'arbre quasi-statique. Les questions peuvent concerner toute sorte de statistiques sur le chemin, comme une somme, un minimum... Nous étudierons dans cette partie des requêtes de minimum sur un chemin.
Nous présenterons dans ce cas un premier algorithme qui s'appuie sur la technique dite de **décomposition heavy-light**.

1.3 Approche plus générale

Si maintenant la structure d'arbre est dynamique, au sens où il faut désormais maintenir une forêt où l'on peut couper des arêtes ou relier deux arbres entre eux, il faut être capable d'effectuer ces nouvelles opérations, tout en répondant aux questions posées.
On étudiera à ce but la structure de Link-Cut Tree.

2 Décomposition Heavy-light

2.1 Idée

On considère un arbre enraciné à N nœuds.
Dans toute la suite, père(u) désigne le père du nœud u, ou u si u est la racine.
Fils(u) désigne l'ensemble des fils du nœud u.
Pour tout nœud u différent de la racine, Val[u] désigne le poids de l'arête qui relie u à père(u).

Pour tout nœud u, on note W[u] le nombre de nœuds dans le sous-arbre de u.
Ainsi, si u est une feuille, W[u] = 1 ; sinon W[u] = $1 + \sum\limits_{v \in Fils(u)} W[v]$.
Pour chaque nœud u qui n'est pas une feuille, on définit son fils lourd comme étant un nœud v qui vérifie :
W[v] = $\max\limits_{k \in Fils(u)} (W[k])$.
Une arête est dite lourde si elle relie un nœud à son fils lourd, et légère sinon.
Pour toute arête légère on a ainsi $W[v] \leq \frac{1}{2}$ W[père(v)].

Définition 2.1.1. On appelle profondeur légère du nœud u, notée lprof(u), le nombre d'arêtes légères sur le chemin de u à la racine.

Lemme 1. *lprof(v)* $\leq \log n$

Démonstration. On procède par récurrence sur N.
La propriété est vraie pour N=1.
Soit N $\geq$ 1 tel que la propriété soit vraie pour tous les arbres de taille 1,...,N.
On considère un arbre enraciné à (N+1) nœuds. Soit r la racine de cet arbre et v un nœud de l'arbre.
Si v=r, la propriété est vraie. Sinon, soit a le fils de r tel que v appartienne au sous-arbre de a.
Si (a,r) est lourde, alors par hypothèse de récurrence, lprof(v) $\leq \log W[a] < \log W(r)$.
Si (a,r) est légère, par hypothèse de récurrence on a $lprof(v) \leq 1 + \log W[a]$.

Or $W[a] \leq \frac{1}{2}W[r]$
D'où $lprof(v) \leq \log W[r]$. □

On peut donc désormais définir notre décomposition.

Définition 2.1.2. On appelle chemin lourd un chemin constitué uniquement d'arêtes lourdes, qui ne peut être prolongé par aucune de ses extrémités.

Définition 2.1.3. On appelle décomposition heavy-light la décomposition de l'arbre selon ses chemins lourds, qui forment une partition de l'ensemble des nœuds (voir figure 1).

Définition 2.1.4. Si u et v appartiennent au même chemin lourd, on désignera par [u ;v] l'intervalle de nœuds constitué par la partie de chemin lourd entre u et v (inclus).

2.2 Résolution

On applique la décomposition à notre arbre.
Par construction, on a naturellement une fonction qui a un nœud u associe le nœud le plus haut sur le chemin lourd de u. On appellera cette fonction PHSCL(u).
De plus, on construit une fonction qui à un nœud associe père(PHSCL(u)), c'est à dire une fonction qui nous fait remonter une arête légère. On notera cette fonction remonteLeger(u). On supposera par ailleurs fournie une fonction donnePoidsLeger(u) qui renvoie le poids de l'arête légère que l'on pourrait remonter.

2.2.1 Sur un chemin

Supposons que l'on sache répondre à une requête de minimum sur un chemin lourd en temps $O(T(N))$, c'est à dire on suppose donnée une fonction reqMin(x,y) qui renvoie le minimum des poids sur [x ;y]. Comment en déduire la réponse à une requête sur un chemin quelconque, de A à B ?

Soit P le plus petit ancêtre commun à A et B. Le calcul de P peut se faire aisément en $O(\log^2 N)$, en supposant un pré-calcul de $O(N \log N)$.
Il suffit de calculer le minimum de A à P et de B à P pour en déduire le minimum de A à B.
Comment calculer le minimum de A à P ?
Il suffit d'appliquer l'algorithme suivant :

```
pos=A, mini = +inf
tant que pos n'appartient pas au même chemin lourd que P
    mini = min(mini, reqMin(PHSCL(pos),pos))
    mini = min(mini, val[pos]) //ceci correspond à la valeur de l'arête légère qui relie pos à père(pos)
    pos = remonteLeger(pos)
return min(mini, reqMin(P, pos))
```

Ainsi, il s'agit de remonter tout un chemin lourd, puis de prendre une arête légère, et de recommencer.

2.2.2 Sur un chemin lourd

Comment répondre à une requête sur un chemin lourd ?
Soit A et B deux nœuds appartenant à un même chemin lourd. On souhaite connaître le minimum sur le chemin de A à B.
Il suffit pour cela de considérer un arbre binaire minimum posé sur le chemin (voir figure 2). On peut alors répondre à notre requête en $O(\log N)$.

2.3 Complexité

Nous obtenons ainsi une structure pour des arbres quasi-statiques, où la complexité par requête est :
— $O(\log N)$ pour une modification
— $O(\log^2 N)$ pour une requête de minimum.
D'où un algorithme en $O((N + M)\log^2 N)$, où M est le nombre de requêtes.

3 Link-cut trees

3.1 Splay Trees

Nous aurons besoin dans un premier temps d'introduire une nouvelle structure : **les splay trees**. Il s'agit d'un arbre binaire de recherche pour lequel les opérations classiques (insertion, suppression, recherche) sont en $O(\log N)$ amorti. L'idée fondamentale derrière le splay tree est la fonction "splay", qui consiste à déplacer le nœud auquel on vient d'accéder à la racine, en conservant la structure d'arbre binaire de recherche. C'est cette opération qui permet à l'arbre d'être toujours relativement équilibré.

3.1.1 Rotation d'une arête

On définit l'opération **rotation d'une arête** d'un arbre binaire de la manière suivante : Soit (A,B) une arête, reliant A à l'un de ses deux fils, B.

— Rotation gauche : B est le fils gauche de A. A devient le fils droite de B.

— Rotation droite : B est le fils droite de A. A devient le fils gauche de B.

On se référera à la figure 3 pour clarifier la conservation de la structure.

Définition 3.1.1. La rotation d'une arête (a,b) où b est un fils de a, sera notée rotate((a,b)) ou de manière allégée, rotate(b).
Elle correspond à une rotation gauche ou une rotation droite selon que b est le fils gauche ou droit de a.

Définition 3.1.2. Soit u un nœud de l'arbre différent de la racine, et p = père(u) son père.
On note $p-(G)\to u$ ou $p-(D)\to u$, si u est respectivement le fils gauche ou le fils droit de p.

3.1.2 L'opération splay(v)

Soit v un nœud de l'arbre, p = père(v) et gp = père(p).
Le but est de déplacer récursivement v à la racine de l'arbre.

Si v est la racine, on s'arrête.

— **Étape zig** : Si gp = null, on effectue l'opération rotate(v) (figure 4)

— **Étape zig-zig** : Si les deux arêtes sont dans le même sens, ie. ($gp-(G)\to p$ et $p-(G)\to v$) ou ($gp-(D)\to p$ et $p-(D)\to v$), on effectue rotate(p) puis rotate(v). (figure 5)

— **Étape zig-zag** : Si les deux arêtes sont dans des sens opposés, on effectue rotate(v) puis rotate(v). (figure 6)

3.1.3 Preuve de la complexité

Soit w une fonction de pondération sur l'ensemble des nœuds telle que pour tout u, $w(u) > 0$.
Pour chaque nœud u, on définit sa taille et son rang de la manière suivante :
s(u) = somme des w(k) dans le sous arbre de u.
r(u) = $\lfloor \log(s(u)) \rfloor$.
On définit la fonction potentielle $\phi = \sum_{u\in G} r(u)$.
Du point de vue de la méthode du comptable en analyse amortie, on dispose de r(u) jetons sur chaque nœud u. Il s'agit en fait de crédits alloués au départ qui vont nous permettre de financer une partie des opérations, l'autre partie correspondant à des jetons que l'on doit payer en plus.

Lemme 2. *Lemme de l'accès : Le nombre d'étapes de splay lors du splay d'un nœud u dans un arbre de racine t est au plus 3(r(t) - r(u))+1*

Démonstration. Chaque opération splay coûte 1 jeton, et il faut maintenir le bon nombre de jetons sur chaque nœud (invariant : nombre de jetons sur u = r(u)).
On va montrer que chaque opération zig-zig et zig-zag coûte au plus 3(r(u') - r(u)), et l'opération zig (qui n'est effectuée qu'une seule fois au plus) coûte 3(r(u')-r(u))+1. En sommant, par téléscopage, on aura bien 3(r(t) - r(u))+1.

Considérons une étape de splay(a) qui fait remonter a jusqu'à r.

— Pour zig (figure 7) : le coût est de 1, et on sait que $b \leq r$. Il faut donc payer au plus $1+(r-a)$.

— Pour zig-zig (figure 8) : On sépare en deux cas, et on voit que le coût est $\leq 2 * (r - a)$.
— Pour zig-zag (figure 9) : De même, on sépare en deux cas, et on voit que le coût est $\leq 3 * (r - a)$.

□

Finalement, en prenant pour tout u, w(u) = 1, on a un potentiel initial $\phi \leq N \log N$.
On peut donc conclure le résultat suivant sur un ensemble de M splays :

Théorème. Le coût de M splays sur un arbre binaire de taille N est O((M+N) log N).

Démonstration. Par téléscopage, coût $\leq \phi_{final} - \phi_{initial} \leq M(3 \log N + 1) - N \log N = O((M + N) \log N)$. □

3.2 Link-cut Tree

Considérons à nouveau un arbre enraciné à N nœuds, que nous désignerons comme l'arbre réel.
Pour tout nœud v, on définit son **fils préféré** par :
— **none**, si le dernier accès dans le sous-arbre de v est v.
— **w**, si le dernier accès dans le sous-arbre de v est dans le sous-arbre de w.

De même, on appelle **arête préférée** une arête qui relie un noeud à son fils préféré. Toutes les autres arêtes sont dites **normales**. Les chemins préférés partitionnent l'ensemble des nœuds.

On définit alors les arbres auxiliaires :

Définition 3.2.1. A chaque chemin préféré est associé un splay tree, appelé **arbre auxiliaire**. Ce splay tree est indexé par la profondeur du noeud dans l'arbre réel. De plus, on attribue à chaque splay tree un pointeur, appelé **pointeur-parent**, vers le père du noeud le plus haut du chemin, c'est à dire vers le chemin préféré sur lequel on arrive en remontant par une arête normale.

Notre arbre est donc ainsi représenté par un arbre d'arbres auxiliaires.
On définit sur les nœuds de notre arbre réel les fonctions suivantes :

3.2.1 L'opération accès(u)

On définit l'opération d'accès à un noeud u. Lorsque l'on accède à un noeud, certaines arêtes préférées changent, et le chemin de u à la racine devient préféré. Ainsi, toutes les arêtes sur ce chemin deviennent préférées, et les anciennes arêtes préférées sont supprimées et remplacées par des pointeurs-parent. L'opération accès effectue de plus un splay afin de positionner u à la racine de son arbre auxiliaire ; u est ainsi la racine de l'arbre des arbres auxiliaires (voir figure 10).

```
accès(u) :
    -splay(u)
    -on retire le fils préféré de u :
           v.right.pathparent=u
        v.right.parent=none
        v.right=none
    -tant que u.pathparent<>none
        -w=u.pathparent
        -splay w
        -on change le fils préféré de w en u :
            w.right.pathparent=w
               w.right.parent=none
               w.right=v
               v.parent=w
               v.pathparent=none
            -splay v
```

3.2.2 Les opérations link(v,w) et cut(u)

Pour lier deux noeuds v et w, en supposant que v est la racine de son arbre réel, il suffit d'accéder à v et à w, et de constater qu'alors v n'a pas de sous-arbre gauche (racine de l'arbre réel).

Pour couper l'arête qui relie u à père(u), il suffit d'accéder à u, et de le détacher de son sous-arbre auxiliaire gauche dans lequel se situe l'ensemble de ses ancêtres. Ceci est cohérent avec la représentation des pointeurs-parent (qui pointent vers des noeuds et non des arbres).

```
link(v,w) :
   -accès(v)
   -accès(w)
   -v.left=w
   -w.parent=v

cut(u) :
   -accès(u)
   -u.left.parent=none
   -u.left=none
```

3.3 Analyse de la complexité

Soit N la taille de l'arbre, M le nombre d'opérations link, cut et accès.
On appelle K le nombre de changements de fils préféré.
Nous allons démontrer la complexité en deux étapes, la première permettant de dénombrer K.

— **1ère étape** : $O(\log^2 N)$ amorti

On a vu qu'un splay s'effectuait en $O(\log N)$ amorti.

Pour M opérations, la complexité totale est : $O(\log N) * O(M + K)$.
En effet, l'opération coûteuse est accès, dont la boucle principale s'exécute K fois au total, en effectuant un splay à chaque fois.
Montrons que $K = O(M \log N)$.
Il suffit pour cela d'analyser chaque fonction.
Lemme 3. *$K \leq$ nb créations préférées légères + nb suppressions préférées lourdes + 2*(N-1)*

Démonstration. Considérons une arête intervenant dans un changement. Celle-ci change d'état préférentiel : elle est soit supprimée, soit crée, dans l'ensemble des arêtes préférées. Par ailleurs, cette arête est soit lourde soit légère.
On peut donc dire que K $\leq$ nb créations pref légères + nb suppr pref légères + nb crea pref lourdes + nb suppr pref lourdes.
Donc $K \leq 2 * nbCreaPrefLeg + 2 * nbSupprPrefLourdes + A$, où A est le nombre d'arêtes légères qui ont été supprimées sans êtres crées (donc déjà présentes au début), plus le nombre d'arêtes lourdes qui ont été créées sans être supprimées (donc toujours présentes à la fin).
Ainsi, A $\leq 2 * (N - 1)$. D'où l'inégalité. □

Il suffit désormais de compter séparément le nombre d'arêtes préférées légères crées, noté B, et le nombre d'arêtes préférées lourdes supprimées, noté C, dans chacune des opérations.

- Pour l'opération accès(v), le chemin de v à la racine est rendu préféré. Comme nous l'avons vu dans la partie précédente, ce chemin comporte au plus $\log N$ arêtes légères, donc $B \leq logN$.
  De plus, chaque arêtes lourde supprimée sur des nœuds du chemin correspond donc à une arête légère crée, à part éventuellement pour v. Ainsi, $C \leq 1 + \log N$

- Pour l'opération link(v,w) : il faut compter deux opérations accès, puis certaines arêtes préférées deviennent lourdes, ce qu'on ne compte pas, et donc certaines non préférées deviennent légères, ce qu'on ne compte pas non plus.

— Pour l'opération cut(v) : des arêtes deviennent légères sur un chemin ; on ne peut en avoir plus de $\log N$, donc $B \leq \log N$.

Voir figure 11.

Ainsi, $K = O(N + M \log N)$.

— **2ème étape** : $O(\log n)$ amorti.

On utilise le lemme de l'accès en posant pour tout nœud u la fonction de pondération suivante :

W(u) = 1 + la taille du sous-arbre de u dans l'arbre des aux-trees.

En se plaçant dans l'arbre des aux-trees, on a alors r(u) définie comme précédemment. On a donc l'inégalité du **lemme 2** qui s'applique.

On a ainsi :

— accès(u) exécute des splays jusqu'à la racine, on a donc, pour tout v intervenant dans un changement de fils préféré un coût de $O(\log(W(pere(v))) - \log(W(v)) + 1)$, donc en sommant sur tous les noeuds du chemin induit par accès, on obtient une complexité de :

$\sum_v O(\log(W(pre(v))) - \log(W(v)) + 1) = O(\log(W(racine)) - \log(W(u)) + K)$.

C'est à dire $O(\log N)$ amorti. Nous sommes donc passés d'un facteur K à un terme K grâce à l'analyse avec la méthode du potentiel.

— cut(u) qui ne fait que décroitre la fonction W, donc le potentiel décroit.

— link(u,v) qui augmente uniquement L(v), d'au plus n. Le potentiel augmente donc d'au plus $\log n$.

Conclusion : on obtient bien une complexité amortie en $O(\log n)$.

4 Application

4.1 Énoncé du problème

On se donne un arbre à N nœuds. Chaque nœud est soit noir, soit blanc. Au début, tous les nœuds sont noirs. On doit alors gérer M requêtes présentées sous forme de couple :

— (0, u) $\rightarrow$ Donner la taille de la composante connexe du nœud u. Ici deux nœuds sont reliés si et seulement si tous les nœuds sur le chemin qui les relie sont de même couleur.

— (1, u) $\rightarrow$ Changer la couleur du nœud u.

Notre but est de concevoir un algorithme capable de traiter efficacement ces requêtes.

4.2 Résolution par la décomposition Heavy-light

On enracine notre arbre arbitrairement, puis on applique la décomposition.

On rappelle que PHSCL(u) est la fonction qui au nœud u associe le plus haut nœud sur le chemin lourd de u, et remonteLeger(u) associe père(PHSCL(u)).

Dans chaque nœud u, on stocke deux informations : black[u] correspond à la taille de la composante noire dans le sous-arbre de u, en supposant que u est noir, et white[u] à la taille de la composante blanche dans le sous-arbre de u, en supposant que u est blanc, et ce indépendamment de la couleur réelle de u.

Supposons que l'on dispose d'une fonction TrouvePlusHaut(u) qui renvoie le nœud le plus haut de la même couleur que u.

Alors voici comment traiter les deux requêtes :

— Changer la couleur de u :

Supposons que la couleur de u était noire.

Soit A = père(TrouvePlusHaut(u)). A est le premier nœud blanc au dessus de u.

Pour chaque nœud i sur le chemin de u à A (u exclu), on retranche black[u] à black[i], et on ajoute white[u] à white[i].

— Trouver la réponse pour u.

Si u est noir, il suffit d'afficher black[TrouvePlusHaut(u)].

4.2.1 TrouvePlusHaut(u)

Pour ce faire, considérons sur chaque chemin lourd un arbre binaire somme, de telle sorte que chaque nœud contienne le nombre de nœuds noirs sur l'intervalle qu'il représente. On considère pour simplifier que u est noir.

Alors, en partant de p = u, pPrec = u, on considère la somme sur l'intervalle [PHSCL(p), p], notée $somme(PHSCL(p), p)$. Si elle est égale à la taille de l'intervalle, on pose k = remonteLeger(p), pPrec := p et on réitère avec p := k. D'après le lemme 1, ceci ne peut se produire que $\log N$ fois, et chaque requête prend un temps $O(\log N)$.
Sinon, on peut effectuer une dichotomie pour trouver le nœud le plus haut du chemin lourd v qui vérifie $somme(v, p) = |[v; p]|$, c'est à dire la taille du chemin entre v et p.
La réponse est alors soit ce nœud v, soit le plus haut nœud de l'intervalle précédent, c'est à dire pPrec. En effet, si v est blanc, c'est donc que la réponse est pPrec.

4.2.2 Modifier les nœuds intermédiaires

Il s'agit d'implémenter une représentation efficace de black[] (white[] se traite de manière analogue). Ceci se fait aisément avec un arbre d'intervalles. Lorsqu'on modifie une couleur, il faut modifier l'ensemble de ses parents ; c'est à dire une réunion d'intervalles sur des chemins lourds. Ainsi, notre structure doit être capable d'ajouter une valeur constante sur un intervalle, ce qui peut être fait avec un arbre d'intervalles (voir figure 12).
Pour lire la valeur d'un nœud u, il suffit donc de sommer les valeurs stockées dans l'arbre binaire B, de u jusqu'à la racine de B (voir figure 12).

4.2.3 Complexité

TrouvePlusHaut(u) s'effectue en $O(\log^2 N)$, de même que modifier les nœuds intermédiaires.
Récupérer la valeur d'un nœud se fait en $O(\log N)$.
D'où une complexité de $O((N + M) \log^2 N)$.

4.3 Résolution avec un link-cut tree

L'idée est simple : on commence par séparer chaque nœud u en deux nœuds intermédiaires, Unoir et Ublanc.
Soit V un nœud de l'arbre et U = pere(V).
Si V est blanc, alors Vblanc doit être relié à Ublanc. Si V est noir, Vnoir doit être relié à Unoir.
Ainsi, une modification de couleur donne lieu à une opération cut et une opération link.

Pour traiter une requête sur u, il suffit donc de connaitre la taille de la composante de même couleur que u.

Plaçons nous dans l'arbre des arbres auxiliaires auquel appartient u.
Si l'on accède à u, il sera placé à la racine de la racine des arbres auxiliaires. On peut alors trouver le noeud le plus haut relié à u en descendant tout à gauche. Appelons le r.
Il suffit alors de répondre à la question : combien de noeuds dans le sous-arbre de r ?
Après une opération splay(r), cette question coïncide avec la suivante : combien de noeuds dans le sous-arbre des arbres auxiliaires enraciné en r ?
(notons qu'il ne faut pas compter r, qui est d'une couleur différente)

Pour ce faire, on appelle pour tout u, taille[u] la taille du sous-arbre splay de u (le nombre de noeuds descendants dans l'arbre des splay trees) et w[u] = somme sur x tel que x→ pathparent → u de taille[x] On a alors la relation taille[u] = taille[gauche(u)] + taille[droite(u)] + w[u] + 1.

On remarque que w[u] est modifié uniquement lors des accès, et que l'on peut facilement le maintenir à jour. Dès lors, il suffit de mettre à jour régulièrement taille[u] grâce à la formule plus haut. En fait, il suffit d'appliquer cette formule à chaque fois que la structure de l'arbre est modifiée, c'est à dire lors des splay et accès.

Ainsi, il suffit d'afficher taille[droite[r]].

4.4 Comparaison des résultats

J'ai testé mes algorithmes sur un site où le problème est disponible, avec une batterie de tests non publics et une limite de temps stricte. Les deux ont obtenu le score maximal, ce qui montre la validité des idées et de leur implémentation. J'ai aussi codé un générateur aléatoire.
Sur les tests ainsi générés, on obtient les résultats suivants (sur une machine à 2.4Ghz) :

Taille de l'entrée (N ; M) \ Algorithme		Exhaustif	Heavy-light	Link-cut
10	10	0.005s	0.023s	0.001s
10	40	0.007s	0.023s	0.001s
100	500	0.07s	0.024s	0.002s
500	1500	0.010s	0.026s	0.003s
1000	5000	0.016s	0.029s	0.007s
3000	10000	0.034s	0.039s	0.013s
5000	15000	0.084s	0.048s	0.018s
10000	50000	0.347s	0.087s	0.048s
30000	50000	3.026s	0.166s	0.066s
50000	50000	5.932s	0.242s	0.080s
80000	100000	19.00s	0.409s	0.154s
100000	100000	23.21s	0.509s	0.182s

5 Conclusion

Comme nous l'avons vu, la décomposition heavy-light ainsi que la structure de link-cut tree permettent de résoudre efficacement le problème. Cependant, cette dernière est plus efficace et permet de modéliser des arbres dont la structure évolue dynamiquement. Les link-cut trees ont été inventés par Sleator et Tarjan dans le but d'optimiser un algorithme de flot maximum, et ont grâce à cela obtenu un algorithme de meilleure complexité que tous ceux précédemment inventés. De nos jours, l'algorithme de flot maximum le plus efficace est un algorithme hybride, qui mélange différentes techniques en fonction des valeurs du problème, notamment celle découverte grâce aux link-cut trees.

La décomposition heavy-light, comme dans notre étude, peut aussi s'avérer très utile pour des calculs de complexité. Le splay tree, inventé peu de temps après et élément central de notre étude, est au coeur de plusieurs questions encore ouvertes, quant à son optimalité en tant qu'arbre binaire de recherche dynamique.
Ces différents algorithmes peuvent être placés dans le cadre plus général des graphes dynamiques, sur lesquels il existe de nombreux résultats intéressants.

Références

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique*, chapter Analyse amortie, chapitre 17. Dunod, 2002.

[2] Erik Demaine (MIT). Cours sur les arbres dynamiques (article). `https://courses.csail.mit.edu/6.851/spring12/scribe/L19.pdf`, Mai 2012.

[3] Erik Demaine (MIT). Cours sur les arbres dynamiques (vidéo). `https://courses.csail.mit.edu/6.851/spring12/lectures/L19.html`, 2012.

[4] Daniel Sleator and Robert Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 1982. `https://www.cs.cmu.edu/~sleator/papers/dynamic-trees.pdf`.

[5] Xiaodao. Page du problème sur codechef. `http://www.codechef.com/problems/QTREE6`, 2013.

A Illustrations

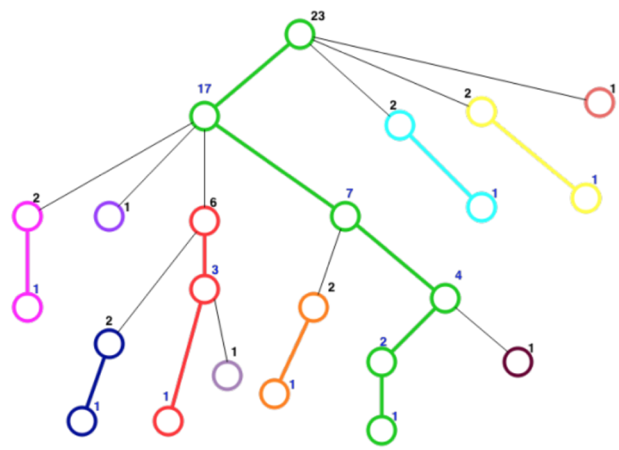

FIGURE 1 – Décomposition d'un arbre selon ses chemins lourds, chacun représenté d'une couleur différente. Les fines arêtes noires représentent les arêtes légères.

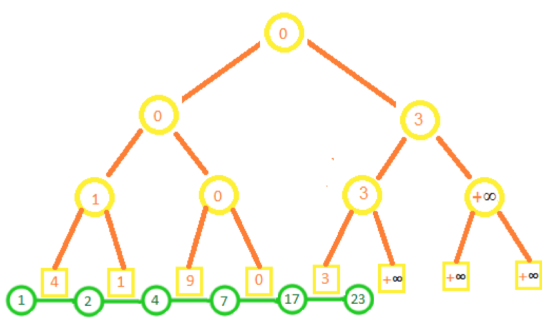

FIGURE 2 – Arbre binaire minimum posé sur le chemin lourd de la figure 1. Les carrés correspondent aux poids des arêtes. Chaque noeud contient alors le minimum de ses deux fils.

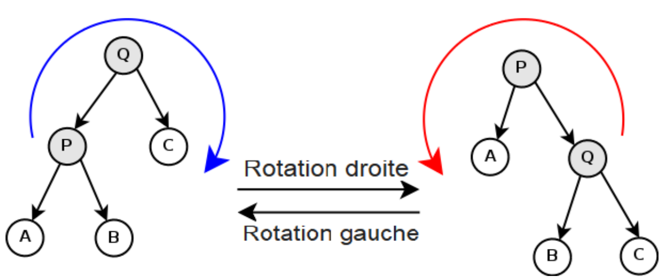

FIGURE 3 – Rotation de l'arête (P,Q)

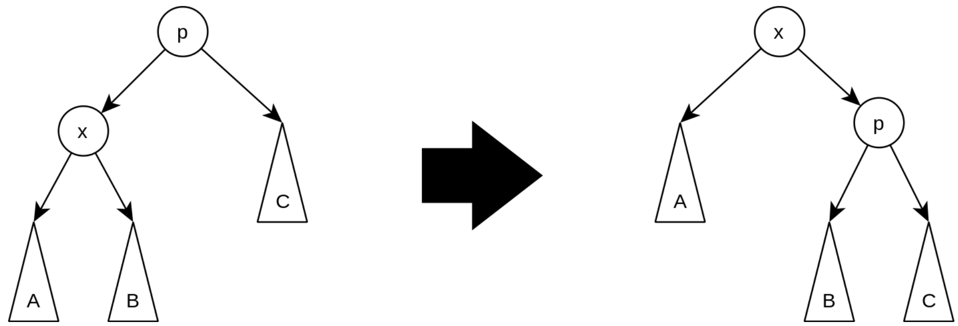

FIGURE 4 – Étape **zig**, $p = p$ère(x) est la racine de l'arbre.

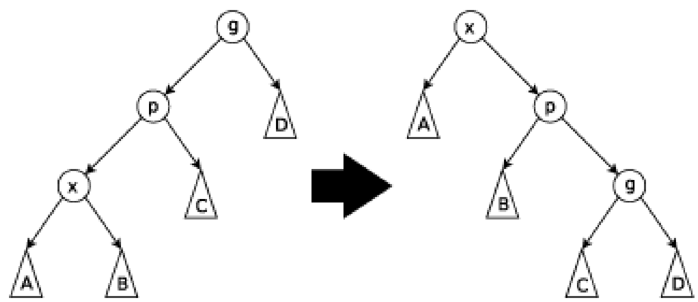

FIGURE 5 – Étape **zigzig**, p = père(x) et g = père(p), la position relative de x-p = celle de p-g.

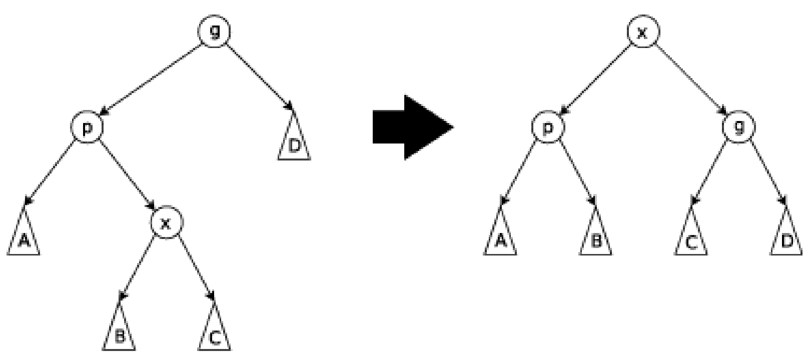

FIGURE 6 – Étape **zigzag**, p = père(x) et g = père(p), la position relative de x-p <> celle de p-g.

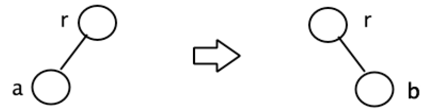

FIGURE 7 – $b \leq r$, le coût est donc majoré par r-a+1

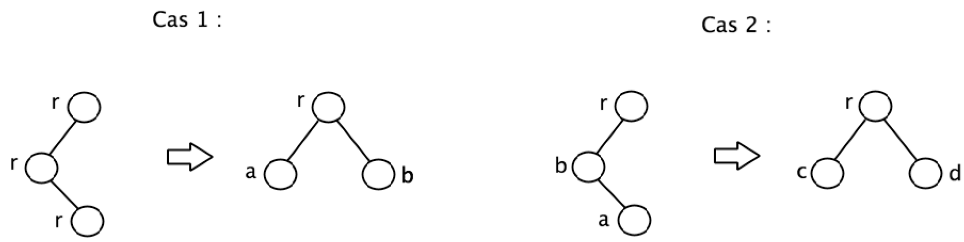

FIGURE 8 – Dans le premier cas, a ou b < r donc 3(r-r)=0 suffit. Dans le second cas, d'une part $b \geq c$ et d'autre part $r > a$, donc on peut utiliser $r - a \geq 1$ pour l'opération et $r - a$ pour d.

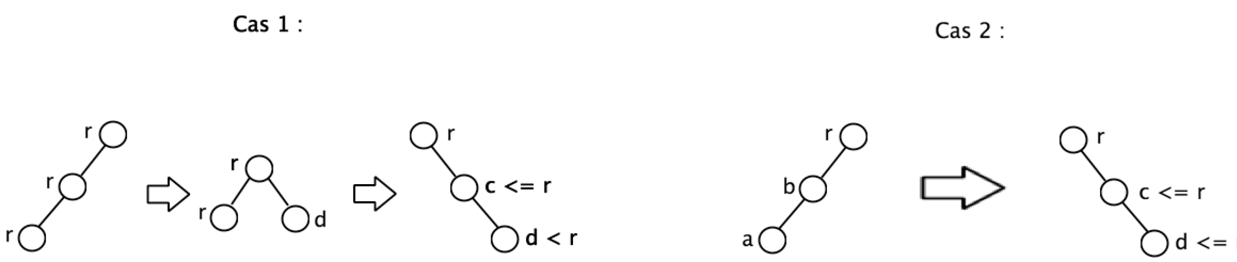

FIGURE 9 – Dans le premier cas, $d < r$ donc on obtient un jeton supplémentaire pour payer l'opération. Dans le second cas, on utilise r-a pour l'opération, puis r-a pour c et r-a pour d.

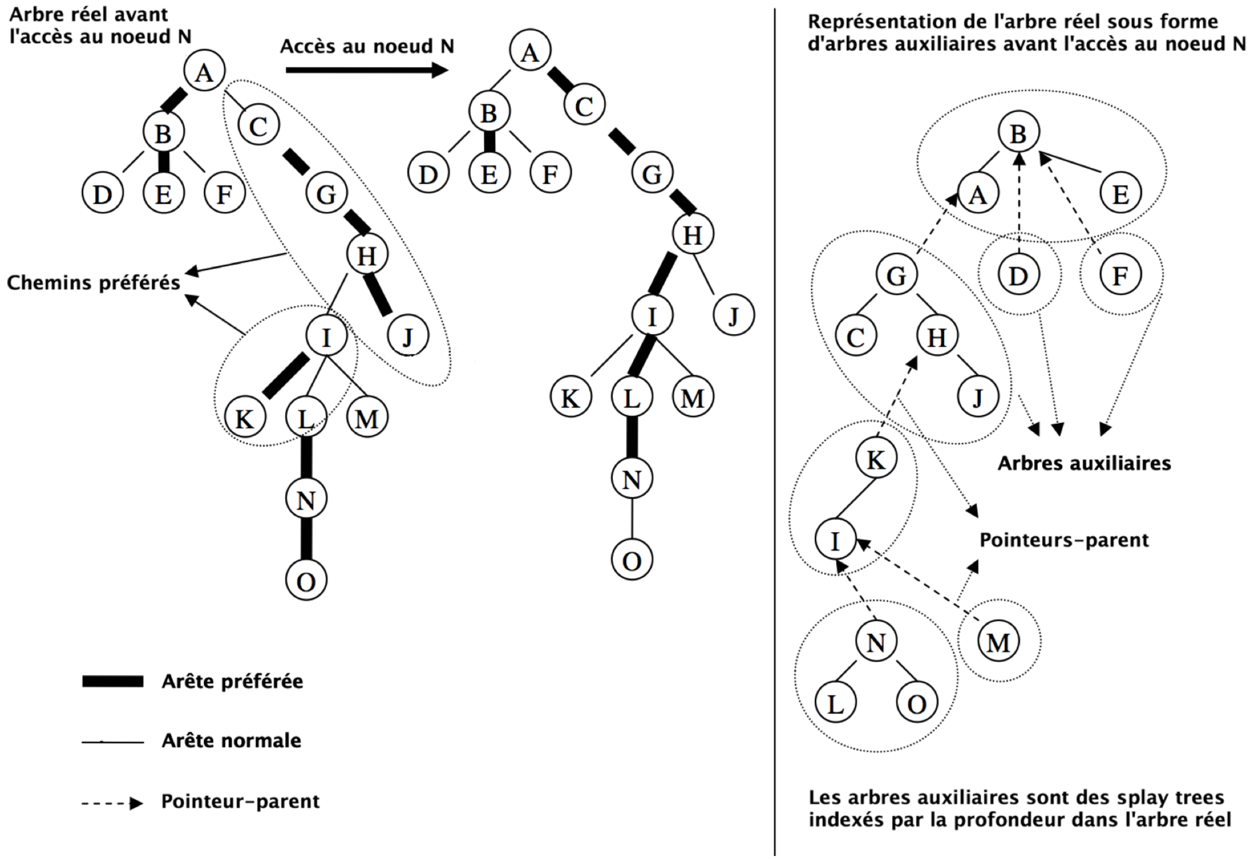

FIGURE 10 – Schéma explicatif de l'opération accès(u)

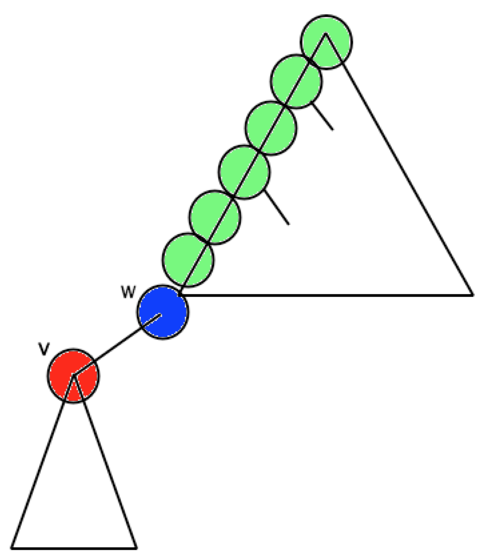

FIGURE 11 – Coût des opérations link et cut. Lorsque l'on relit v à w, les traits noirs représentent les arêtes qui étaient lourdes et deviennent légères. Ces arêtes ne sont pas préférées, elles ne sont donc pas comptées. Lorsque l'on coupe l'arête entre v et w, cependant, certaines arêtes du chemin de w à la racine deviennent légères. Celles-ci se situent toutes sur un chemin de l'arbre, donc par le lemme 1, il y en a au plus $\log N$.

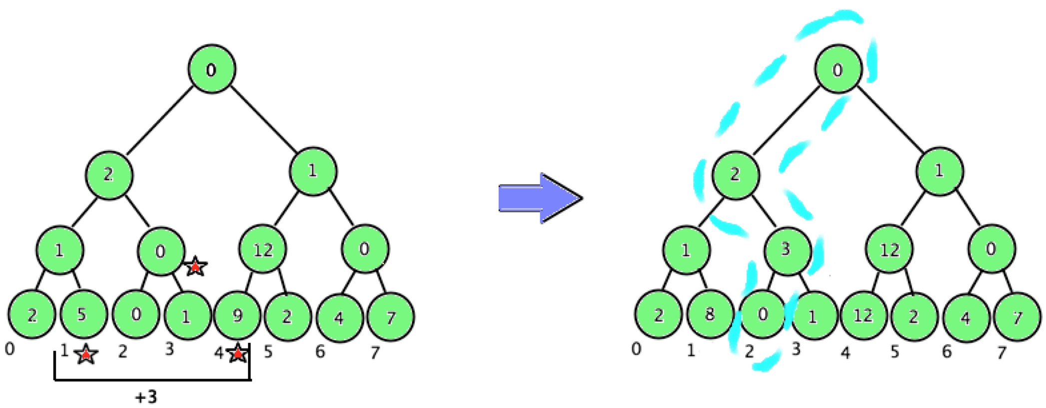

FIGURE 12 – Exemple d'arbre d'intervalle. Chaque nœud contient le bonus à ajouter à l'intervalle qu'il représente.

B Code source Heavy-light

```cpp
#include <cstdlib>
#include <cstdio>
#include <algorithm>
#include <vector>

using namespace std;

int N;
int M;

const int MAXN = 100001;

//3 arbres, 0 = black[], 1 = white[], 2=pour trouver upper
int color[MAXN];
vector<int> arcs[MAXN];
int pere[MAXN];
int heavy[MAXN];
int parent[MAXN];
vector<int> tree[3][MAXN];
int id[MAXN][2];
vector<int> lourd[MAXN];
int nbFeuilles[MAXN];

int dfs(int noeud, int ppere=1)
{
    int pb=0;
    int maxi=0;
    int tt = 1;
    int maxj=0;
    pere[noeud]=ppere;
    for(int i = 0; i < arcs[noeud].size(); i++)
    {
        if(arcs[noeud][i]==ppere){pb=i;continue;}
        int nb = dfs(arcs[noeud][i], noeud);

        tt+=nb;
        if(nb>maxi)
            maxi=nb,maxj=i;
    }
    heavy[noeud]=arcs[noeud][maxj];
    if(arcs[noeud].size()>0 && arcs[noeud][pb]==ppere)
    {
        swap(arcs[noeud][pb], arcs[noeud].back());
        arcs[noeud].pop_back();
    }

    return tt;
}

int oc=1;

void dfs2(int noeud, int pos=0)
{
    //curr représente le numéro du chemin lourd
```

```
    int curr=0;
    if(noeud!=1)
        curr=id[pere[noeud]][0];
    if(noeud!=1&&heavy[pere[noeud]]!=noeud)
    {
        curr=oc++;
        parent[curr]=pere[noeud];
        pos=0;
    }

    id[noeud][0]=curr;
    id[noeud][1]=pos;
    tree[0][curr].push_back(0);
    tree[1][curr].push_back(0);
    tree[2][curr].push_back(0);
    lourd[curr].push_back(noeud);

    for(int i = 0; i < arcs[noeud].size(); i++)
    {
        dfs2(arcs[noeud][i], pos+1);
    }
}

void updateTree(int col, int arbre, int gauche, int droite, int val)
{
    if(gauche>droite)
    {
        return;
    }
    if(gauche==droite)
        tree[col][arbre][gauche]+=val;
    else
    {
        if(gauche&1)
        {
            tree[col][arbre][gauche]+=val;
            gauche++;
        }
        if(!(droite&1))
        {
            tree[col][arbre][droite]+=val;
            droite--;
        }
        updateTree(col, arbre, gauche/2, droite/2, val);
    }
}

int somme(int arbre, int gauche, int droite)
{
    if(gauche>droite)return 0;
    if(gauche==droite)
        return tree[2][arbre][gauche];
    if(gauche&1)
        return tree[2][arbre][gauche]+somme(arbre, (gauche+1), droite);
    if(!(droite&1))
        return tree[2][arbre][droite]+somme(arbre, gauche, (droite-1));
```

```c
    return somme(arbre, gauche/2, droite/2);
}

void afficheQuiArbre()
{
    for(int i = 1; i <= N; i++)
    {
        printf("noeud %d : arbre=%d\n", i, id[i][0]);
    }
}

int findUpper(int u, int col) //trouve le plus haut noeud avec tout de même couleur sur le
{
    int nbF = nbFeuilles[id[u][0]];
    int gauche = nbF;
    int droite = id[u][1];

    int s = somme(id[u][0], gauche, droite);

    if((col==0 && s==droite-gauche+1) || (col==1 && s==0))
    // tous les noeuds de l'intervalle de u sont de la couleur col
    {
        if(id[u][0]==0)
            return lourd[id[u][0]][gauche-nbF];
        int res = findUpper(parent[id[u][0]],col);
        if(color[res]==col)
            return res;
        return lourd[id[u][0]][gauche-nbF];
    }
    else // la réponse se situe ici
    {
        //NB : on peut se permettre un log^2 car il est final.

        gauche = nbFeuilles[id[u][0]]-1; //gauche exclu droite inclus
        droite = id[u][1];

        while(gauche < droite-1)
        {
            int mid=(gauche+droite)/2;

            int ok = col==1?0:id[u][1]-mid+1;

            if(somme(id[u][0], mid, id[u][1]) == ok)
            {
                droite=mid;
            }
            else
                gauche=mid;
        }
        return lourd[id[u][0]][droite-nbF];
    }
}

void update(int node, int lim, int val, int col)
{
    int gauche = nbFeuilles[id[node][0]];
```

```
    if(id[node][0] == id[lim][0])
        gauche=id[lim][1];

    updateTree(col, id[node][0], gauche, id[node][1], val);

    if(id[node][0] != id[lim][0])
        update(parent[id[node][0]], lim, val, col);
}

void maj(int arbre, int noeud)
{
    if(noeud==0)return;

    tree[2][arbre][noeud]=tree[2][arbre][noeud*2] + tree[2][arbre][noeud*2+1];

    maj(arbre, noeud/2);
}

int sumToRoot(int col, int arbre, int noeud)
{
    if(noeud==0)return 0;

    return tree[col][arbre][noeud] + sumToRoot(col, arbre, noeud/2);
}

int main()
{
    scanf("%d", &N);

    for(int i = 1; i < N; i++)
    {
        int a,b;
        scanf("%d%d",&a,&b);

        arcs[a].push_back(b);
        arcs[b].push_back(a);
    }

    //enracine l'arbre et crée la structure heavy-light
    dfs(1);
    dfs2(1);

    for(int i = 0; i < N; i++)
    {
        int p2=1;
        while(p2<tree[0][i].size())p2<<=1;
        nbFeuilles[i]=p2;
        p2<<=1;
        tree[0][i].resize(p2);
        tree[1][i].resize(p2);
        tree[2][i].resize(p2);
    }

    for(int i = 1; i <= N; i++)
    {
        id[i][1]+=nbFeuilles[id[i][0]];
```

```
        tree[2][id[i][0]][id[i][1]] = 1;
    }
    for(int i = 1; i <= N; i++)
    {
        maj(id[i][0], id[i][1]/2);

        update(i, 1, 1, 0);
        tree[1][id[i][0]][id[i][1]]=1;
    }

    //afficheQuiArbre();

    scanf("%d", &M);

    for(int i = 0; i < M; i++)
    {
        int t,u;

        scanf("%d%d", &t,&u);

        int upperSB = findUpper(u, color[u]);
        if(t==0)
        {
            printf("%d\n", sumToRoot(color[u], id[upperSB][0], id[upperSB][1]));
        }
        else
        {
            int maValeur=sumToRoot(color[u], id[u][0], id[u][1]); //val[u]
            if(u!=1)
                update(pere[u], pere[upperSB], -maValeur, color[u]); //on soustrait à tous

            color[u]=1-color[u];

            tree[2][id[u][0]][id[u][1]]^=1;
            maj(id[u][0], id[u][1]/2);

            upperSB = findUpper(u,color[u]);

            maValeur=sumToRoot(color[u], id[u][0], id[u][1]);

            if(u!=1)
                update(pere[u], pere[upperSB], +maValeur, color[u]);
        }
    }

    return 0;
}
```

C Code source Link-cut Tree

```
#include <cstdlib>
#include <cstdio>
#include <algorithm>
#include <vector>

using namespace std;
```

```
const int NB_NOEUDS = 100005;
int N,M;

struct Noeud
{
    int id;
    int w;
    int sz;
    Noeud* fa;
    Noeud* gauche;
    Noeud* droite;
    Noeud* pere;
    Noeud* pathparent;

    Noeud() : id(0), w(0), sz(0), fa(NULL), gauche(NULL), droite(NULL), pere(NULL), pathpar
    {
    }
};

Noeud arbre[NB_NOEUDS][2];
vector<int> arcs[NB_NOEUDS];

void update(Noeud *noeud)
{
    int d=0,g=0;
    if(noeud->droite)
        d=noeud->droite->sz;
    if(noeud->gauche)
        g=noeud->gauche->sz;
    noeud->sz = g + d + noeud->w + 1;
}

int col[NB_NOEUDS];

Noeud* findRoot(Noeud* noeud)
{
    while(noeud->gauche)
        noeud=noeud->gauche;
    return noeud;
}

void setPere(Noeud *noeud, Noeud* pere)
{
    if(noeud)
        noeud->pere=pere;
}

/*WIKI*/

void left_rotate( Noeud *x ) {
    Noeud *y = x->droite;
    if(y) {
      x->droite = y->gauche;
      if( y->gauche ) y->gauche->pere = x;
      y->pere = x->pere;
```

```

      y->pathparent=x->pathparent;
    x->pathparent=NULL;
    }

    if( !x->pere ) ;//root = y;
    else if( x == x->pere->gauche ) x->pere->gauche = y;
    else x->pere->droite = y;
    if(y) y->gauche = x;
    x->pere = y;
    update(x);
  }

  void right_rotate( Noeud *x ) {
    Noeud *y = x->gauche;
    if(y) {
      x->gauche = y->droite;
      if( y->droite ) y->droite->pere = x;
      y->pere = x->pere;

      y->pathparent=x->pathparent;
        x->pathparent=NULL;
    }
    if( !x->pere ) ;//root = y;
    else if( x == x->pere->gauche ) x->pere->gauche = y;
    else x->pere->droite = y;
    if(y) y->droite = x;
    x->pere = y;
    update(x);
  }

  void splay( Noeud *x ) {
    while( x->pere ) {
      if( !x->pere->pere ) {
        if( x->pere->gauche == x ) right_rotate( x->pere );
        else left_rotate( x->pere );
      } else if( x->pere->gauche == x && x->pere->pere->gauche == x->pere ) {
        right_rotate( x->pere->pere );
        right_rotate( x->pere );
      } else if( x->pere->droite == x && x->pere->pere->droite == x->pere ) {
        left_rotate( x->pere->pere );
        left_rotate( x->pere );
      } else if( x->pere->gauche == x && x->pere->pere->droite == x->pere ) {
        right_rotate( x->pere );
        left_rotate( x->pere );
      } else {
        left_rotate( x->pere );
        right_rotate( x->pere );
      }
    }
    update(x);
  }
/* WIKI */

/*void rotate(Noeud *noeud)
{
```

```
    Noeud *pere = noeud->pere;

    //màj pp
    noeud->pathparent = pere->pathparent;
    pere->pathparent  = NULL;
    ////////

    if(pere->gauche==noeud)
    {
        if(pere->droite)pere->droite->pere=noeud;
        pere->gauche=noeud->droite;
        noeud->droite=pere;
    }
    else
    {
        if(pere->gauche)pere->gauche->pere=noeud;
        pere->droite=noeud->gauche;
        noeud->gauche=pere;
    }
    setPere(pere->gauche, pere);
    setPere(pere->droite, pere);
    setPere(noeud, pere->pere);
    setPere(pere, noeud);

    update(noeud);
}

void splay(Noeud *noeud)
{
    while(noeud->pere)
    {
        Noeud *p = noeud->pere;
        Noeud *gp= p->pere;

        if(!gp)
            rotate(noeud); //zig
        else if((p->gauche==noeud) ^ (gp->droite==noeud))
            rotate(p), rotate(noeud); //zig-zig
        else
            rotate(noeud), rotate(noeud); //zig-zag
    }
    update(noeud);
}*/

void access(Noeud *noeud)
{
    splay(noeud);
    if(noeud->droite)
    {
        noeud->w += noeud->droite->sz;

        //lct
        noeud->droite->pathparent=noeud;
        noeud->droite->pere=NULL;
        noeud->droite=NULL;
        update(noeud);
```

```
    }
    while(noeud->pathparent)
    {
        Noeud *pp = noeud->pathparent;
        splay(pp);
        if(pp->droite)
        {
            pp->w += pp->droite->sz; //on ajoute celui là qui maintenant est un pp

            //lct
            pp->droite->pathparent=pp;
            pp->droite->pere=NULL;
        }

        pp->w -= noeud->sz; //celui là n'est plus un pp mais bien un fils direct

        pp->droite=noeud;
        noeud->pere=pp;
        noeud->pathparent=NULL;
        update(pp);
        splay(noeud);

        //le pathparent est màj dans le splay
    }
}

void link(Noeud* noeud, Noeud* pere=0)
{
    if(!pere) pere = noeud->fa;

    access(pere);
    access(noeud);
    pere->droite=noeud;
    noeud->pere=pere;
    update(pere);
}

void cut(Noeud *noeud)
{
    access(noeud);
    //noeud->
    if(noeud->gauche){
    noeud->gauche->pere=NULL;
    noeud->gauche=NULL;
    update(noeud);}
}

void dfs(int noeud, int pere=0)
{
    for(int i = 0; i < arcs[noeud].size(); i++)
    {
        if(arcs[noeud][i]!=pere)
        {
            arbre[arcs[noeud][i]][0].fa=&arbre[noeud][0];
            arbre[arcs[noeud][i]][1].fa=&arbre[noeud][1];
            arbre[arcs[noeud][i]][0].pathparent=&arbre[noeud][0];
```

```
            //link(&arbre[arcs[noeud][i]][0], &arbre[noeud][0]);

            dfs(arcs[noeud][i], noeud);
            arbre[noeud][0].w += arbre[arcs[noeud][i]][0].sz;
        }
    }
    update(&arbre[noeud][0]);
    update(&arbre[noeud][1]);
}

int requete(Noeud* noeud)
{
    access(noeud);
    Noeud *rac = findRoot(noeud);
    splay(rac);

    return rac->droite->sz; //on est obligé d'avoir un fils droit car le noeud est à exclur
}

int main()
{
    scanf("%d", &N);

    for(int i = 1; i < N; i++)
    {
        arbre[i][0].id=arbre[i][1].id=i;
        int a,b;
        scanf("%d%d",&a,&b);
        arcs[a].push_back(b);
        arcs[b].push_back(a);
    }
    arbre[N][0].id=arbre[N][1].id=N;

    //sentinelle
    arbre[1][0].fa = &arbre[N+1][0];
    arbre[1][1].fa = &arbre[N+1][1];
    arbre[1][0].pathparent = &arbre[N+1][0];

    dfs(1);
    scanf("%d", &M);

    for(int i = 0; i < M; i++)
    {
        int type, noeud;
        scanf("%d%d", &type, &noeud);

        if(type==1)
        {
            cut(&arbre[noeud][col[noeud]]);
            col[noeud] ^= 1;
            link(&arbre[noeud][col[noeud]]);
        }
        else
            printf("%d\n", requete(&arbre[noeud][col[noeud]]));
    }
```

```
    return 0;
}
```