ÉCOLE NORMALE SUPÉRIEURE DE LYON

Concours d'admission session 2025

Filière universitaire : Second concours

COMPOSITION D'INFORMATIQUE

Durée : 3 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve. L'orthographe ainsi que la propreté de la copie seront prises en compte par les correcteurs.

⋆ ⋆ ⋆

Arbres binaires de recherche et équilibrage

Ce sujet contient des questions de démonstrations mathématiques, ainsi que des questions de pseudo-code. Pour des raisons d'uniformité, on demande à écrire les pseudo-codes dans un langage proche de Python. On évalue la rigueur, la concision et précision.

1 Arbre binaire de recherche

Un **arbre binaire de recherche** est un arbre binaire, c'est-à-dire chaque nœud possède au plus deux fils. Chaque nœud `x` est étiqueté par une clé `x.k`. De plus, étant donné un nœud `x`, les étiquettes des nœuds du sous-arbre enraciné dans le fils gauche sont strictement inférieures à `x.k`, et les étiquettes des nœuds du sous-arbre enraciné dans le fils droit sont strictement supérieures à `x.k`. Une **feuille** est un nœud sans fils. La Figure 1 ci-dessous montre un arbre binaire de recherche avec 8 nœuds et 4 feuilles. La racine est étiquetée par 8.

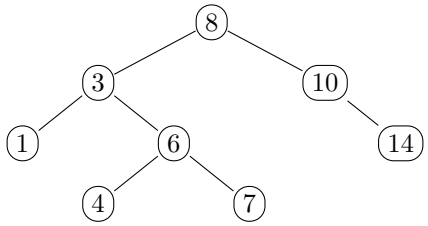


Figure 1: Exemple d'un ABR.

Une **branche** issue de `x` est un chemin simple qui part de `x` jusqu'à une feuille dans le sous-arbre enraciné en `x`. La **hauteur** d'un nœud `x` est la longueur en nombre d'arcs d'une plus longue branche issue de `x`. La hauteur d'un ABR est la hauteur de sa racine. L'ABR de la Figure 1 est de hauteur 3. Souvent on confond un ABR et son nœud racine `x`.

Question 1 Dessiner un arbre binaire de recherche de hauteur 2 et un autre de hauteur 4 pour le même ensemble de clés $\{1, 4, 5, 10, 16\}$.

La recherche d'une clé `k` dans un arbre binaire de recherche `x` s'effectue comme suit : on part de la racine. On compare son étiquette avec `k`. Si son étiquette est égale, on a trouvé la clé `k`. Sinon, si elle est inférieure à `k`, on continue la recherche dans le sous-arbre gauche. Sinon, on continue la recherche dans le sous-arbre droit.

En mémoire, un nœud `x` est un objet avec les attributs suivants :

- `x.k` : la **clé** qui étiquette le nœud `x`;
- `x.g` : le **fils gauche** de `x`. Plus précisément, il s'agit de la racine du fils gauche, ou alors une valeur spéciale, que l'on note `None`, s'il n'y a pas de fils gauche.
- `x.d` : le **fils droit** de `x`. Plus précisément, il s'agit de la racine du fils droit, ou alors une valeur spéciale, que l'on note `None`, s'il n'y a pas de fils droit.
- `x.p` : contient le **parent** de `x`, ou alors `None` si le nœud `x` est la racine de l'ABR.

On dispose d'une fonction `creerNoeud(k)` qui crée un nœud x étiqueté avec la clef `k`, de fils gauche `None`, de fils droit `None` et de parent `None`. Il s'agit du seul moyen de créer un nœud. La Figure 2 (page 3) donne l'exemple de la fonction `peigne(`n`)`, ainsi que l'arbre qu'elle crée et renvoie.

Question 2 Donner le nombre d'appels à `creerNoeud` effectués par un appel à `peigne(n)`. Justifier.

Question 3 Donner un algorithme `appartient(A, k)` qui renvoie `True` (vrai) si la clé `k` est présente dans l'ABR `A`, et renvoie `False` sinon.

```
1  def peigne(n):
2      if n <= 0:
3          return None
4
5      A = creerNoeud(n)
6      A.d = None
7      A.g = peigne(n-1)
8      A.g.p = A
9
10     return A
```

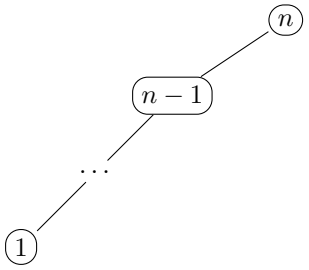


Figure 2: La fonction `peigne(`n`)` (à gauche), et l'arbre qu'elle renvoie (à droite).

Question 4 Donner la complexité temporelle asymptotique pire cas de la recherche `appartient(A, k)` en fonction de la hauteur h de l'ABR `A`.

Question 5 Donner la complexité temporelle asymptotique pire cas de la recherche `appartient(A, k)` en fonction du nombre n de nœuds dans l'ABR `A`. Justifier.

Question 6 Donner un algorithme `ajouter(A, k)` qui ajoute une clé `k` à un ABR `A`. On rappelle qu'un ABR vide est représentée par `None`. La fonction renvoie (un pointeur sur) la racine de l'arbre. On exige que la fonction fasse au plus un appel à `creerNoeud`.

2 Arbres rouge-noir

Afin de garantir une meilleure complexité pour la recherche et l'insertion, on introduit la notion d'arbres rouge-noir. Un **arbre rouge-noir** est un ABR dans lequel chaque nœud est colorié en rouge ou en noir, en respectant les contraintes suivantes :

1. La racine de l'arbre est noire.
2. Si un nœud est rouge, alors ses enfants sont noirs.
3. Pour tout nœud `x`, tous les chemins simples reliant `x` à une feuille située plus bas contiennent le même nombre de nœuds noirs.

Pour cela, on stocke la couleur d'un nœud x dans l'attribut $x.couleur$. La valeur de $x.couleur$ est soit noire ou rouge.

Question 7 Montrer que l'exemple de la Figure 3 ci-dessous est un arbre rouge-noir.

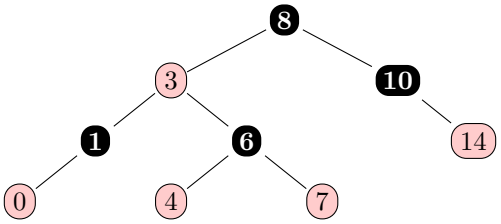


Figure 3: Exemple d'un arbre rouge-noir.

2.1 Propriétés structurelles

Soit $bh(\mathtt{x})$ la **hauteur noire** du nœud `x`, il s'agit du nombre de nœuds noirs le long d'une branche reliant `x` à une feuille située plus bas. Par exemple, un arbre rouge-noir avec un seul nœud est de hauteur 0 et de hauteur noire 1 (car la racine est noire).

Question 8 Montrer par récurrence sur la hauteur de `x` que tout arbre rouge-noir enraciné en `x` contient au moins $2^{bh(\mathtt{x})} - 1$ nœuds.

Question 9 En déduire qu'un arbre de n nœuds a une hauteur au plus égale à $2\log_2(n+1)$.

Question 10 Montrer que le nombre de nœuds dans une plus longue branche issue de `x` est au plus égale à deux fois le nombre de nœuds dans celle d'une plus courte branche issue de `x`.

2.2 Rotations

La **rotation droite** est l'opération qui consiste intuitivement à faire pivoter vers la droite un arbre `y`, comme le montre la Figure 4 ci-dessous. Elle suppose que le fils gauche de `y` existe. Les deux fils du fils gauche de `y` et le fils droit de `y` peuvent être `None`.

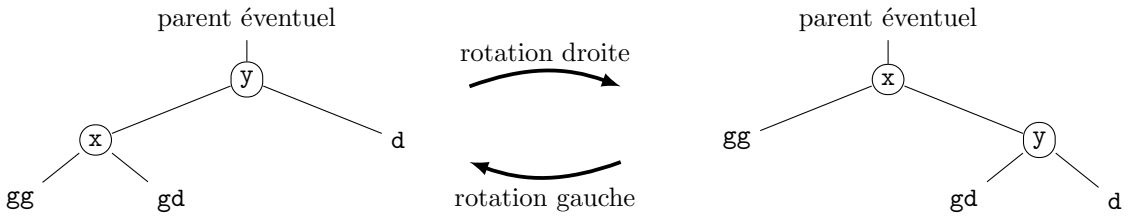


Figure 4: Rotation droite : l'arbre de racine `y` est transformé en un arbre de racine `x`. Les sous-arbres `gg`, `gd` et `d` peuvent être `None`.. La rotation gauche est l'opération qui transforme l'arbre de racine `x` en l'arbre de racine `y`. La fonction respecte le parent du sous-arbre : le parent de `y` devient le parent de `x` (ce parent pouvant être `None` si `y` était la racine).

Question 11 Écrire le pseudo-code de `rotationDroite(y)` qui prend en entrée un nœud `y`, modifie en place les objets existants pour effectuer la rotation droite en la racine `y` (cf. Figure 4), et renvoie `x`.

Question 12 Quelle est la complexité de la rotation droite ?

De la même façon, on définit la **rotation gauche** sur x qui prend en entrée un arbre de racine x comme celui de droite sur la Figure 4 et qui renvoie l'arbre de gauche enraciné en y.

Question 13 Soit A et A' deux arbres binaire de recherche contenant les même clefs. Montrer que l'on peut transformer A en A' en effectuant une suite de rotations appliquées en différents nœuds de l'arbre.

2.3 Insertion d'une clé

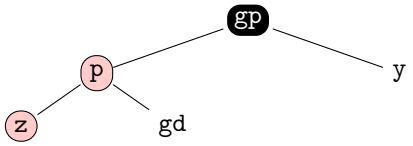
L'insertion naïve dans un ABR ne garantit pas de respecter les propriétés d'un arbre rouge-noir. On propose donc l'algorithme suivant pour insérer une clé `k`.

1. Insérer `k` dans `A` en tant qu'une feuille z avec `z.k` = `k`.
2. Si `A` est `None` alors `A` := `z`
3. Peindre la feuille `z` en rouge
4. Appeler la fonction `correction(A, z)` donnée en Figure 5 (page 5) où l'appel initial est effectué sur cette feuille z. Cette fonction renvoie la racine de l'arbre corrigé.

Question 14 Montrer que la variable `gp` en ligne 14 dans la Figure 5 n'est pas affectée à `None`.

On peut dessiner l'effet de la correction au niveau du nœud `z` courant. Pour cela, on peut dessiner la situation initiale et la situation finale. Par exemple, le cas '`y` $\neq$ `None` et `y` est rouge' (ligne 19) peut se dessiner comme suit :

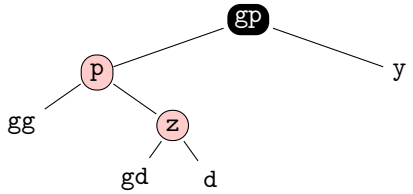
Question 15 Dessiner la situation finale dans le cas **`else`** (ligne 24 dans la Figure 5) quand `z` est un fils gauche, i.e. depuis la situation initiale :



```
def correction(A, z):
    # arguments :
    #   A est le noeud racine
    #   z est le noeud où on effectue la correction
    # renvoie le noeud racine après correction

    if z == A:
        peindre z en noir
        return A
    p = z.p
    if p est noir:
        return A

    gp = parent de p

    if p est un fils gauche
        y = gp.d

        if y != None and y est rouge:
            peindre p en noir
            peindre y en noir
            peindre gp en rouge
            return correction(gp)
        else:
            if z est un fils droit:
                z = z.p
                rotationGauche(z)

            peindre z.p en noir
            peindre z.p.p en rouge
            r = rotationDroite(z.p.p)
            return r if r.p == None else A
    else:
        # même pseudo-code en échangeant droite et gauche
```

Figure 5: Fonction de correction.

Question 16 Dessiner la situation intermédiaire après la ligne 27 dans la Figure 5 puis la situation finale, dans le cas **`else`** (ligne 24) quand `z` est un fils droit, i.e. depuis la situation initiale :



Question 17 Montrer que `z` est un fils gauche à partir de la ligne 29 dans la Figure 5.

Question 18 Donner une borne sur le nombre de rotations (appels à `rotationGauche` et `rotationDroite`) lors d'une insertion dans un arbre-rouge noir.

Question 19 Montrer que `correction(A, z)` termine.

Question 20 Montrer que l'on a un arbre rouge noir après insertion d'une clef `k`. Indice : écrire une propriété vérifiée par l'entrée `(A, z)` donnée à la fonction `correction`.

Question 21 Donner la complexité asymptotique de l'ajout d'un élément dans un arbre rouge-noir en fonction du nombre n d'éléments dans la structure.

2.4 Application : tri

Question 22 Donner un algorithme `trier(T)` pour trier un tableau `T` de n éléments par comparaison en temps $O(n \log n)$. On demande à ce que la fonction `trier(T)` renvoie un tableau avec les même éléments que le tableau `T` initial. On demande à ce que la **fonction utilise un arbre rouge-noir**.

3 Rangs dynamiques

Un **arbre de rang** est un arbre rouge-noir dans lequel, en tout nœud `x`, on ajoute l'information `x.taille` qui est le nombre de nœuds du sous-arbre enraciné en `x` (`x` compris). La Figure 6 ci-dessous montre un exemple d'un arbre de rang.

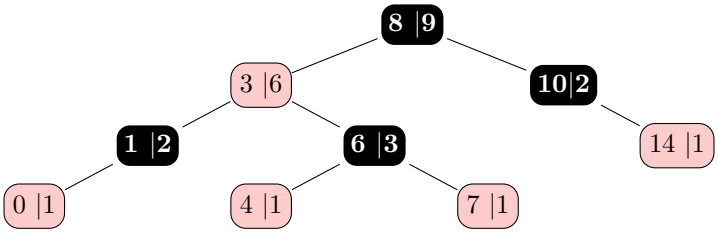


Figure 6: Exemple d'un arbre rang. En plus des clés, on indique la taille en chaque nœud. Par exemple, en la racine, on inscrit '|9' car il y a 9 nœuds dans l'arbre.

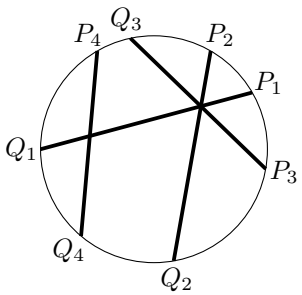
Le **rang** d'un élément `k` dans un ensemble E est l'indice `i` tel que `k` soit le `i`-ème plus petit élément dans E. Par extension, le **rang** d'une clé `k` dans un arbre de rang `A` est le rang de `k` dans l'ensemble des clefs apparaissant dans `A`. Par exemple, le rang de 6 dans l'arbre de la Figure 6 est 5 car il est en 5-ième position dans la suite triée des éléments 0, 1, 3, 4, 6, 7, 8, 10, 14.

3.1 Quelques manipulations

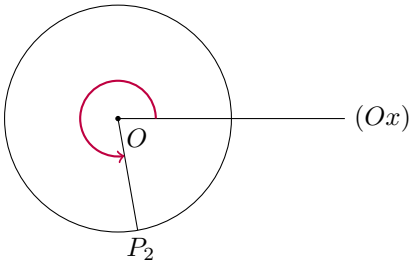
Question 23 Écrire le pseudo-code d'une fonction `donnerieme(A, i)` qui donne la `i`-ème plus petite clé dans l'arbre `A` (i.e. qui donne l'élément `k` de rang `i` dans l'arbre `A`).

Question 24 Écrire une fonction `rang(A, k)` qui renvoie le rang de la clé `k` dans l'arbre de rang `A`. On suppose que `k` se trouve dans `A`. (on rappelle que les clefs dans l'arbre sont distinctes).

Question 25 Donner les complexités temporelles `donnerieme(A, i)` et `rang(A, k)` en fonction du nombre n d'éléments dans `A`.



(a) Un cercle et quatre cordes. Il y a ici 4 paires de cordes qui se croisent.



(b) Coordonnées polaires. On repère un point P_i par l'angle formé entre l'axe (Ox) et (OP_i). De même pour un point Q_i.

Figure 7: Application géométrique.

```
1    def trier(P, Q):
2        L = P + Q # union des deux listes P et Q
3        L.sort(key=lambda x: x.angle) # tri selon les angles
4        return L
```

Figure 8: Fonction auxiliaire qui prend deux tableaux P et Q en entrée, et renvoie une permutation des éléments contenues dans P et Q, triée selon les angles.

Dans la suite, on suppose que l'on dispose de **fonctions d'insertion et de suppression** dans un arbre de rang qui s'exécutent en $O(\log n)$ où n est le nombre de nœuds dans l'arbre. Plus précisément, on dispose d'une fonction `inserer(A, k)` qui insère la clé `k` dans l'arbre de rang enraciné en `A` et renvoie la racine de l'arbre après insertion. On dispose aussi d'une fonction `supprimer(A, k)` qui supprime la clé `k` de l'arbre `A` et renvoie la racine de l'arbre après suppression.

3.2 Application géométrique

On considère un cercle de centre O. On note (Ox) l'axe des abscisses. Une **corde** $[P_iQ_i]$ est un segment qui relie deux points P_i et Q_i du cercle. La Figure 7a ci-dessus montre un cercle et $n = 4$ cordes $[P_1Q_1], \dots, [P_4Q_4]$. On représente un ensemble de n cordes par deux listes $P = (P_1, \dots, P_n)$ et $Q = (Q_1, \dots, Q_n)$ de points sur le cercle. On suppose que les points $P_1, \dots, P_n, Q_1, \dots, Q_n$ sont deux à deux distincts.

Étant donné n cordes, l'objectif est de calculer efficacement le **nombre de paires de cordes** qui se croisent à l'intérieur du cercle. Sur l'exemple donné en Figure 7a, il y a 4 paires de cordes qui se croisent : $[P_1Q_1]$ et $[P_4Q_4]$, $[P_1Q_1]$ et $[P_3Q_3]$, $[P_1Q_1]$ et $[P_2Q_2]$, et $[P_2Q_2]$ et $[P_3Q_3]$.

On utilise les coordonnées polaires. On repère un point P_i (resp. Q_i) par l'angle entre (Ox) et (OP_i) (resp. (OQ_i)), cf. Figure 7b ci-dessus. L'angle est notée P_i.`angle` (resp. Q_i.`angle`). Chaque angle est un nombre dans $[0, 2\pi[$.

On suppose que pour tout i, P_i.`angle` $< Q_i$.`angle`. Par convention, P_i est le "début" d'une corde et Q_i sa "fin". On suppose que cette indication est écrite dans les objets P_i et Q_i de la façon suivante : P_i.`type` = `debut` et Q_i.`type` = `fin`. On suppose aussi que les extrémités de fin ont un attribut `angleDebut` qui contient l'angle de l'extrémité de début. Autrement dit, Q_i.`angleDebut` = P_i.`angle`.

La Figure 8 ci-dessus donne la fonction auxiliaire suivante qui place les extrémités dans une seule et même liste `L`. On suppose que `L.sort` trie `L` en place et en temps $O(n \log n)$.

Question 26 Écrire un algorithme `nbcroisements`(P, Q) en temps $O(n \log n)$ qui prend en entrée deux listes P et Q d'angles représentants les extrémités de n cordes, et qui renvoie le nombre de paires de cordes $[P_iQ_i]$ qui se croisent à l'intérieur du cercle.

Question 27 Démontrer que l'algorithme `nbcroisements`(P, Q) est en temps $O(n \log n)$.

FIN DU SUJET.