

TD Algo2 – session 4 – Extension de structure de données

20 janvier 2026

Objectifs d'apprentissage :

- concevoir des algorithmes relatifs aux arbres de rangs ;
- concevoir des algorithmes relatifs aux arbres d'intervalles
- mettre en pratique la technique d'extension de structure de données.

Les 3 premiers exercices sont essentiels.

La méthode pour la conception d'algorithmes est la suivante :

1. comprendre l'énoncé ;
2. faire un exemple non trivial pour résoudre à la main ;
3. en l'absence d'idée d'algorithme, recommencer avec un exemple plus grand ;
4. identifier les étapes de l'algorithme trouvé ;
5. convertir en pseudo-code.

Exercice 1 : Rechercher-Rang-Clé

On considère un arbre de rangs T . Écrire une procédure récursive RECHERCHER-RANG-CLÉ(x, k) qui prend en entrée un nœud x et une clé k , et qui retourne le rang de k dans le sous-arbre de rangs enraciné en x . On supposera que les clés de T sont distinctes et que la clé k est présente dans le sous-arbre enraciné en x .

```
RECHERCHER-RANG-CLÉ( $x, k$ )
r ←  $x.gauche.taille + 1$ 
si  $x.clé = k$  alors
    retourner r
sinon si  $k < x.clé$  alors
    retourner RECHERCHER-RANG-CLÉ( $x.gauche, k$ )
sinon
    retourner  $r + \text{RECHERCHER-RANG-CLÉ}(x.droite, k)$ 
```

Exercice 2 : Successeur

Étant donné un élément x dans un arbre de rangs à n nœuds, et un entier naturel i , comment peut-on déterminer le i -ème successeur de x (c'est-à-dire le nœud dont le rang est celui de x plus i) dans l'arbre avec un temps $O(\log n)$? On suppose que l'élément recherché est dans l'arbre.

```
SUCCESSEUR-RANG( $T, x, i$ )
r ← DÉTERMINER-RANG( $T, x$ )
retourner RÉCUPÉRER-RANG( $T.racine, r + i$ )
```

Exercice 3 : Extrémité minimale

Décrire un algorithme efficace qui, étant donné un intervalle i , retourne l'intervalle recouvrant i dont le début est le plus petit possible, ou qui retourne $T.nil$ si un tel intervalle n'existe pas.

RECHERCHER-INTERVALLE-MINIMAL(T, i)

```

 $x \leftarrow T.\text{racine}$ 
 $y \leftarrow T.\text{nil}$ 
tant que  $x \neq T.\text{nil}$  faire
  si  $i$  recoupe  $x.\text{int}$  faire
     $y \leftarrow x$ 
    si  $x.\text{gauche} \neq T.\text{nil}$  et  $i.\text{début} \leq x.\text{gauche}.max$  alors
       $x \leftarrow x.\text{gauche}$ 
    sinon
       $x \leftarrow x.\text{droite}$ 
  retourner  $y$ 

```

Exercice 4 : Temps constant

Montrer comment mettre en œuvre les requêtes MINIMUM, MAXIMUM, SUCCESEUR et PRÉDÉCESSEUR en temps $\Theta(1)$ dans le cas le plus défavorable. Les performances asymptotiques des autres opérations sur les arbres ne devront pas être affectées. (Conseil : ajouter des pointeurs aux noeuds.)

1. Choix de la structure de données sous-jacente : arbre rouge-noir avec la clé classique.
2. Informations supplémentaires : on ajoute deux champs *pred* et *succ* sur chaque noeud, et deux champs *min* et *max* sur la racine. Ces champs ajoutés forment une liste doublement chaînée sur tous les noeuds.
3. Compatibilité avec l'insertion. À chaque insertion, on appelle l'opération SUCCESEUR sur le noeud inséré dans l'arbre en $O(\log n)$, puis on insère le noeud dans la liste chaînée en $\Theta(1)$. Si le successeur est $T.\text{nil}$, on met à jour $T.\text{max}$ et si le successeur est $T.\text{min}$, on met à jour $T.\text{min}$. Comme l'insertion est en temps $O(\log n)$, sa complexité n'est pas affectée.
4. Nouvelles opérations : les champs ajoutés permettent les requêtes recherchées en $\Theta(1)$.

Exercice 5 : Champ rang

Observons que, chaque fois que le champ taille est utilisé dans RÉCUPÉRER-RANG ou DÉTERMINER-RANG, il ne sert qu'à calculer le rang du noeud dans le sous-arbre issu de ce noeud. Supposons donc que l'on décide de stocker dans chaque noeud son rang dans le sous-arbre dont il est la racine. Montrer comment gérer cette donnée lors de l'insertion. (Ne pas oublier que cette opération peut provoquer des rotations).

On met à jour le rang de tous les ancêtres dont la clé est supérieure.

RN-INSÉRER(T, z)

```

...
 $y \leftarrow z$ 
tant que  $y.\text{parent} \neq T.\text{nil}$  faire
  si  $y = y.\text{parent}.\text{gauche}$  alors
     $y.\text{parent}.\text{rang} \leftarrow y.\text{parent}.\text{rang} + 1$ 
   $y \leftarrow y.\text{parent}$ 

```

RN-INSÉRER-CORRECTION(T, z)

Il faut déterminer combien de prédécesseurs apparaissent dans les rotations. Pour la rotation gauche, y aura comme prédécesseurs supplémentaires : x et α .

ROTATION-GAUCHE(T, x)

```

...
 $y.\text{rang} \leftarrow y.\text{rang} + x.\text{rang}$ 

```

Exercice 6 : Rotation gauche

Écrire un pseudo code pour ROTATION-GAUCHE agissant sur les noeuds d'un arbre d'intervalles et capable de mettre à jour les champs max en $\Theta(1)$.

```
ROTATION-GAUCHE( $T, x$ )
...
 $x.max \leftarrow \max(x.int.fin, x.gauche.max, x.droite.max)$ 
 $y.max \leftarrow \max(x.max, y.max)$ 
```

Exercice 7 : Récupérer-rang

Écrire une version non récursive de RÉCUPÉRER-RANG.

```
RÉCUPÉRER-RANG( $T, i$ )
 $x \leftarrow T.racine$ 
tant que  $x \neq T.nil$  faire
   $r \leftarrow x.gauche.taille + 1$ 
  si  $i = r$  alors
    retourner  $x$ 
  sinon si  $i < r$  alors
     $x \leftarrow x.gauche$ 
  sinon
     $x \leftarrow x.droite$ 
     $i \leftarrow i - r$ 
retourner  $x$ 
```

Exercice 8 : Tous les intervalles

Étant donné un arbre d'intervalles T et un intervalle i , décrire comment tous les intervalles de T recouvrant i peuvent être recensés en $O(\min(n, k \log n))$, où k est le nombre d'intervalles présents dans la liste de sortie. Trouver une solution qui ne modifie pas l'arbre.

```
RECHERCHER-TOUS-INTERVALLES( $x, i$ )
 $y \leftarrow \emptyset$ 
si  $x$  recoupe  $x.int$  faire
   $y \leftarrow \{x\}$ 
  si  $x.gauche \neq T.nil$  et  $i.\text{début} \leq x.gauche.max$  alors
     $y \leftarrow y \cup \text{RECHERCHER-TOUS-INTERVALLES}(x.gauche, i)$ 
  si  $x.droite \neq T.nil$  et  $x.clé \leq i.fin$  alors
     $y \leftarrow y \cup \text{RECHERCHER-TOUS-INTERVALLES}(x.droite, i)$ 
retourner  $y$ 
```

Exercice 9 : Rechercher-Intervalle-Exact

Suggérer des modifications aux procédures d'arbre d'intervalles permettant de supporter l'opération RECHERCHER-INTERVALLE-EXACT(T, i), qui retourne un pointeur sur un noeud x de l'arbre d'intervalles T tel que $x.int.\text{début} = i.\text{début}$ et $i.int.fin = i.fin$, ou qui retourne $T.nil$ si T ne contient pas un tel noeud. Toutes les opérations, y compris RECHERCHER-INTERVALLE-EXACT, devront s'exécuter dans un temps en $O(\log n)$ sur un arbre à n noeuds.

On suppose que la clé selon laquelle les éléments s'insèrent dans l'arbre n'est plus simplement $x.int.\text{début}$, mais $(x.int.\text{début}, x.int.fin)$. C'est-à-dire que les noeuds sont triés suivant l'ordre lexicographique sur les extrémités des intervalles.

```
RECHERCHER-INTERVALLE-EXACT( $T, i$ )
```

```
 $x \leftarrow T.\text{racine}$ 
tant que  $x \neq T.\text{nil}$  et ( $i.\text{début} \neq x.\text{int.}\text{début}$  ou  $i.\text{fin} \neq x.\text{int.}\text{fin}$ ) faire
    si  $i.\text{début} < x.\text{int.}\text{début}$  ou ( $i.\text{début} = x.\text{int.}\text{début}$  et  $i.\text{fin} < x.\text{int.}\text{fin}$ ) alors
         $x \leftarrow x.\text{gauche}$ 
    sinon
         $x \leftarrow x.\text{droite}$ 
retourner  $x$ 
```

Exercice 10 : Hauteurs noires

On souhaite gérer les hauteurs noires des noeuds d'un arbre rouge-noir en tant que champs des nœuds de l'arbre sans que les performances asymptotiques des opérations d'arbre rouge-noir soient affectées. Comment procéder ?

- Choix de la structure de données sous-jacente : arbre rouge-noir.
- Information supplémentaire : le champs *hauteur*, avec $T.\text{nil}.\text{hauteur} = 0$.
- Compatibilité avec l'insertion. On ne modifie le champ que lorsque les couleurs d'un nœud changent :
 - Cas 1 : le grand-parent distribue sa noirceur à ses enfants, donc son champ *hauteur* est incrémenté.
 - Cas 3 : les couleurs sont échangées lors de la rotation et on échange aussi les valeurs des champs *hauteur*.
 - La racine qui était devenue rouge est remise à noire : on incrémente son champs *hauteur*.
- Nouvelle opération : HAUTEUR-NOIRE récupère le champ *hauteur* de n'importe quel nœud en $\Theta(1)$.

Exercice 11 : Profondeur

Peut-on utiliser un champ supplémentaire dans les nœuds d'un arbre rouge-noir pour gérer efficacement la profondeur des nœuds. Dire pourquoi.

On ne peut pas conserver la complexité $O(\log n)$ car lors d'une rotation sur la racine suite à une insertion, la moitié des nœuds peuvent changer de profondeur et il faut donc mettre à jour $O(n)$ profondeurs, ce qui est inefficace.

Exercice 12 : Distance-Min

Montrer comment gérer un ensemble dynamique Q de nombres pouvant supporter l'opération DISTANCE-MIN, qui donne la longueur de la différence entre les deux nombres les plus proches dans Q . Par exemple, si $Q = \{1, 5, 9, 15, 18, 22\}$, alors DISTANCE-MIN(Q) retourne $18 - 15 = 3$, puisque 15 et 18 sont les nombres les plus proches dans Q . Rendre les opérations INSÉRER, SUPPRIMER, RECHERCHER et DISTANCE-MIN les plus efficaces possibles, et analyser leur temps d'exécution.

- Choix de la structure de données sous-jacente : arbre rouge-noir dont les clés sont les nombres.
- Informations supplémentaires : on ajoute les champs $x.\text{distmin}$, $x.\text{min}$ et $x.\text{max}$ qui représentent respectivement la distance minimale entre chaque paire de nombres, le minimum et le maximum pour le sous-arbre enraciné en x . Les valeurs pour $T.\text{nil}$ sont $-\infty$ pour les 2 premiers et ∞ pour le dernier.
- Compatibilité avec l'insertion : quand on insère un élément, on met à jour les nouveaux champs des nœuds sur le chemin du nœud inséré à la racine. Les champs se calculent ainsi :

- $x.distmin \leftarrow \min(x.clé - x.gauche.min, x.droite.max - x.clé, x.gauche.distmin, x.droite.distmin);$
 - $x.min \leftarrow \min(x.clé, x.gauche.min, x.droite.min);$
 - $x.max \leftarrow \max(x.clé, x.gauche.max, x.droite.max).$
- Ce parcours se fait en temps $O(\log n)$. Pour chaque rotation, on réalise à nouveau ces calculs pour x et y , donc en temps $\Theta(1)$.
- Nouvelle opération : DISTANCE-MIN récupère le champ *distmin* de la racine en $\Theta(1)$.

Exercice 13 : Rang

On souhaite un algorithme qui calcule le rang d'une valeur, c'est-à-dire, le nombre de noeuds dont les valeurs sont inférieures ou égales à celle qui est donnée à l'algorithme. Comment faudrait-il procéder pour avoir un algorithme en $O(h)$? Proposer en pseudo-code cet algorithme.

Pour que la méthode soit en $O(h)$, il faut rajouter un champ à chaque noeud et le mettre à jour à chaque insertion ou suppression : la taille du sous-arbre.

```
RANG( $T, v$ )


---


rang  $\leftarrow 0$ 
 $x \leftarrow T.racine$ 
tant que  $x \neq NIL$  faire
    si  $x.clé \leq v$  alors
        rang  $\leftarrow$  rang + 1
        si  $x.gauche \neq NIL$  alors
            rang  $\leftarrow$  rang +  $x.gauche.taille$ 
             $x \leftarrow x.droite$ 
        sinon
             $x \leftarrow x.gauche$ 
    retourner rang
```
