TEA Structures De Données Avancées Automne 2024

Par Amadou Tidiane ANNE, Killian DAMASCENO, Abdel Ouakil KERRAF, Ehoussoud N'GOUAN

Explication des fonction développées

Fonction `insertFixUp(struct noeud **root, struct noeud *z)` :

Cette fonction est utilisée pour **rééquilibrer un arbre Rouge-Noir** après l'insertion d'un nouveau nœud. Elle vérifie et rétablit les **propriétés spécifiques des arbres Rouge-Noir** :

1. Chaque nœud est soit rouge, soit noir.
2. La racine est toujours noire.
3. Les feuilles (`NULL`) sont noires.
4. Aucun chemin ne peut contenir deux nœuds rouges consécutifs.
5. Tous les chemins d'un nœud à ses feuilles contiennent le même nombre de nœuds noirs.

Fonctionnement:

- **Boucle principale** :
  - Tant que le nœud actuel (`z`) n'est pas la racine et que son parent est rouge (condition problématique), il faut rééquilibrer.
- **Récupération de l'oncle (`y`)** :
  - Selon que le parent de `z` est le fils gauche ou droit de son grand-parent, on identifie l'oncle comme étant le sous-arbre opposé.
- **Cas 1 : L'oncle est rouge** :
  - Si l'oncle est rouge, cela signifie que le parent, l'oncle, et le grand-parent de `z` doivent être rééquilibrés :
    - L'oncle et le parent deviennent noirs.
    - Le grand-parent devient rouge.
    - `z` est déplacé vers le grand-parent pour continuer le traitement.
- **Cas 2 : L'oncle est noir (ou inexistant)** :
  - Cela nécessite des rotations pour rééquilibrer :
    - **Sous-cas 1 : Parent à gauche, `z` est un fils gauche** :
      - Une **rotation droite** est effectuée sur le grand-parent.
    - **Sous-cas 2 : Parent à gauche, `z` est un fils droit** :
      - Une **rotation gauche sur le parent**, suivie d'une **rotation droite sur le grand-parent**.
    - **Sous-cas 3 : Parent à droite, `z` est un fils droit** :
      - Une **rotation gauche** est effectuée sur le grand-parent.
    - **Sous-cas 4 : Parent à droite, `z` est un fils gauche** :
      - Une **rotation droite sur le parent**, suivie d'une **rotation gauche sur le grand-parent**.
- **Finalisation** :
  - Une fois l'arbre rééquilibré, la racine est toujours remise en **noir**.

Fonction `insert(struct noeud **root, int data)`

Cette fonction insère un nouveau nœud avec une valeur donnée dans l'arbre Rouge-Noir et utilise `insertFixUp` pour garantir que l'arbre conserve ses propriétés après l'insertion.

Fonctionnement

- **Création du nœud** :
  - Un nouveau nœud (`z`) est créé avec la valeur donnée.
  - Ce nœud est initialement rouge pour respecter la règle que tous les chemins vers les feuilles contiennent le même nombre de nœuds noirs.
- **Insertion dans l'arbre** :
  - Si l'arbre est vide, le nœud est inséré comme **racine** et devient automatiquement noir.
  - Sinon, l'algorithme recherche la **position correcte** pour insérer le nœud, en suivant la règle des arbres binaires de recherche (valeurs inférieures à gauche, supérieures à droite).
  - Une fois inséré, son parent est mis à jour.
- **Rééquilibrage** :
  - Après l'insertion, l'arbre peut devenir déséquilibré (par exemple, deux nœuds rouges consécutifs peuvent apparaître).
  - La fonction `insertFixUp` est appelée pour corriger cela.

Points communs et liens entre les deux fonctions

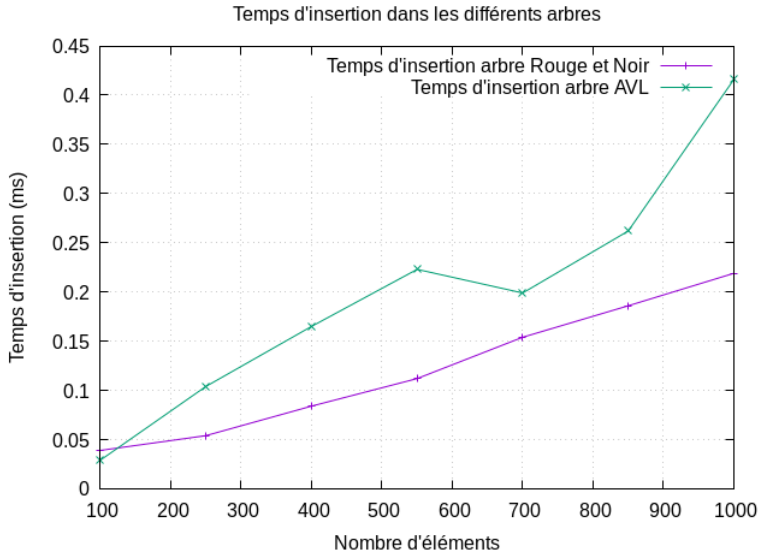
- `insertFixUp` est une fonction auxiliaire appelée par `insert`.
- `insert` gère la logique principale de l'insertion d'un nouveau nœud.
- `insertFixUp` est responsable de restaurer les propriétés spécifiques des arbres Rouge-Noir si elles sont violées après l'insertion.

Résumé des complexités

- La recherche de la position d'insertion est $O(logn)$, car l'arbre est équilibré.
- La correction de l'arbre (fonction `insertFixUp`) est également $O(logn)$, car elle nécessite au maximum un chemin de corrections depuis la feuille jusqu'à la racine.

  O(logn)O(\log n)
- **Complexité totale :** $O(\log n)$**.**

Graphe du temps d'exécution et comparaison avec les arbres AVL



Graphe réalisé avec `gnuplot`, du temps d'exécution par nombre d'éléments

Constats graphiques :

1. **Tendance générale** :
   - Les deux courbes montrent une croissance positive du temps d'insertion en fonction du nombre d'éléments (de 100 à 1000).
   - La courbe représentant l'AVL (bleu-vert avec des croix) croît plus rapidement que celle de l'arbre Rouge et Noir (violet avec des traits).
2. **Disparités de performance** :
   - **Pour un faible nombre d'éléments (100 à 400)** :
     - Les deux arbres ont des performances proches.
     - L'arbre Rouge et Noir est légèrement plus rapide.
   - **Pour un grand nombre d'éléments (> 400)** :
     - La courbe de l'AVL commence à s'éloigner significativement de celle de l'arbre Rouge et Noir.
     - L'arbre Rouge et Noir reste plus stable et performant.
3. **Croissance du temps** :
   - L'arbre Rouge et Noir semble croître de manière plus linéaire et régulière.
   - L'arbre AVL montre des variations plus importantes, probablement dues à des rééquilibrages fréquents.

Explications techniques :

1. **Arbre Rouge et Noir** :
   - L'arbre Rouge et Noir utilise un système de **balancement relâché** (avec une tolérance de hauteur entre les branches). Cela entraîne moins de rotations dans la majorité des cas, donc un temps d'insertion global plus faible.
   - C'est un choix optimal pour des scénarios où les insertions massives sont fréquentes.
2. **Arbre AVL** :
   - L'arbre AVL est **plus strictement équilibré**, ce qui garantit une recherche très rapide mais au coût d'un **nombre plus élevé de rotations** lors des insertions.
   - Pour chaque insertion, il rééquilibre immédiatement, ce qui explique le temps d'insertion plus élevé à mesure que la taille augmente.
3. **Complexité théorique** :
   - Les deux structures ont une complexité d'insertion de $O(\log n)$. Cependant, l'AVL a une surcharge liée aux rotations fréquentes nécessaires pour maintenir son invariant d'équilibre strict.

Conclusion :

- Pour des insertions massives, **l'arbre Rouge et Noir** est un meilleur choix grâce à son approche plus flexible du balancement.

- L’arbre AVL, bien qu’ayant un coût d’insertion plus élevé, est idéal dans des cas où **les opérations de recherche** sont prioritaires et doivent être aussi rapides que possible.