

Arbres binaires

Algorithmes et structures de données, 2025-2026

P. Albuquerque (B410) et O. Malaspinas (A401), ISC, HEPIA
2026-03-18

En partie inspiré des supports de cours de P. Albuquerque

Les arbres binaires

Rappel pour l'insertion

- Les éléments insérés ont une notion d'ordre
- On parcourt l'arbre jusqu'à pouvoir ajouter une nouvelle feuille

Pseudo-code d'insertion (1/4)

- Deux parties :
 - Recherche le parent où se passe l'insertion.
 - Ajout de l'enfant dans l'arbre.

Recherche du parent

```
arbre position(tree, clé)
  si est_non_vide(tree)
    si clé < clé(tree)
      suivant = gauche(tree)
    sinon
      suivant = droite(tree)
  tant que clé(tree) != clé && est_non_vide(suivant)
    tree = suivant
    si clé < clé(tree)
      suivant = gauche(tree)
    sinon
      suivant = droite(tree)

  retourne tree
```

Pseudo-code d'insertion (2/4)

- Deux parties :
 - Recherche de la position.
 - Ajout dans l'arbre.

Ajout de l'enfant

```
rien ajout(tree, clé)
    si est_vide(tree)
        tree = nœud(clé)
    sinon
        si clé < clé(tree)
            gauche(tree) = nœud(clé)
        sinon si clé > clé(tree)
            droite(tree) = nœud(clé)
    sinon
        retourne
```

Recherche du parent (ensemble)

Code d'insertion en C

Recherche du parent (ensemble)

```
node *position(node *tree, key_t key) {
    node * current = tree;
    if (NULL != current) {
        node *subtree = key > current->key
                        ? current->right : current->left;
        while (key != current->key && NULL != subtree) {
            current = subtree;
            subtree = key > current->key
                    ? current->right : current->left;
        }
    }
    return current;
}
```

L'insertion (3/4)

- Deux parties :
 - Recherche de la position.
 - Ajout dans l'arbre.

Ajout du fils (pseudo-code)

```
rien ajout(tree, clé)
  si est_vide(tree)
    tree = nœud(clé)
  sinon
    tree = position(tree, clé)
    si clé < clé(tree)
      gauche(tree) = nœud(clé)
    sinon si clé > clé(tree)
      droite(tree) = nœud(clé)
    sinon
      retourne
```


Ajout du fils (code)

- 2 cas : arbre vide ou pas.
- on retourne un pointeur vers le nœud ajouté (ou NULL)

L'insertion (4/4)

Ajout du fils (code)

- 2 cas : arbre vide ou pas.
- on retourne un pointeur vers le nœud ajouté (ou NULL)

```
node *add_key(node **tree, key_t key) {
    node *new_node = calloc(1, sizeof(*new_node));
    new_node->key = key;
    if (NULL == *tree) {
        *tree = new_node;
    } else {
        node * subtree = position(*tree, key);
        if (key == subtree->key) {
            return NULL;
        } else {
            if (key > subtree->key) {
                subtree->right = new_node;
            } else {
                subtree->left = new_node;
            }
        }
    }
    return new_node;
}
```

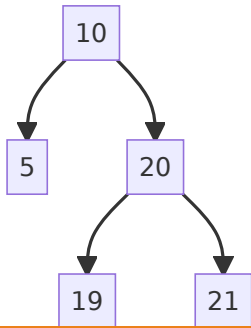
La suppression dans un arbre binaire

La suppression de clé

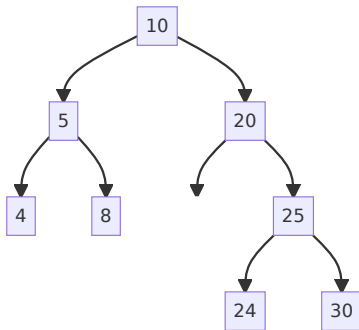
Cas simples :

- le nœud est absent,
- le nœud est une feuille,
- le nœuds a un seul fils.

Une feuille (le 19 p.ex.).



Un seul fils (le 20 p.ex.).



Dans tous les cas

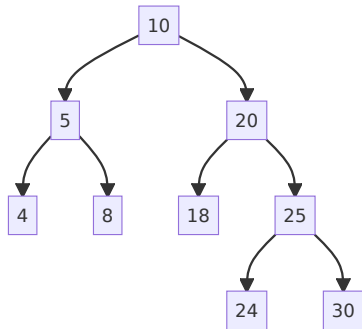
- Chercher le nœud à supprimer : utiliser `position()`.

La suppression de clé

Cas compliqué

- Le nœud à supprimer a (au moins) deux descendants (10).

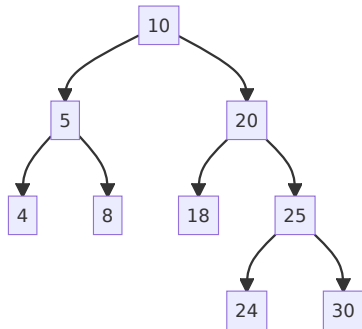
- Si on enlève 10, il se passe quoi ?



La suppression de clé

Cas compliqué

- Le nœud à supprimer a (au moins) deux descendants (10).

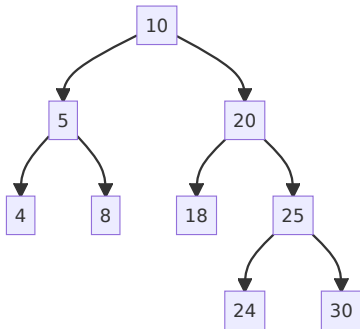


- Si on enlève 10, il se passe quoi ?
- On ne peut pas juste enlever 10 et recoller...
- Proposez une solution !

La suppression de clé

Cas compliqué

- Le nœud à supprimer a (au moins) deux descendants (10).



- Si on enlève 10, il se passe quoi ?
- On ne peut pas juste enlever 10 et recoller...
- Proposez une solution !

Solution

- Échange de la valeur à droite dans le sous-arbre de gauche ou ...
- de la valeur de gauche dans le sous-arbre de droite !
- Puis, on retire le nœud.

Le pseudo-code de la suppression

Pour une feuille ou absent (ensemble)

```
arbre suppression(arbre, clé)
    sous_arbre = position(arbre, clé)
    si est_vide(sous_arbre) ou clé(sous_arbre) != clé
        retourne vide
    sinon
        si est_feuille(sous_arbre) et clé(sous_arbre) == clé
            nouvelle_feuille = parent(arbre, sous_arbre)
            si est_vide(nouvelle_feuille)
                arbre = vide
            sinon
                si gauche(nouvelle_feuille) == sous_arbre
                    gauche(nouvelle_feuille) = vide
                sinon
                    droite(nouvelle_feuille) = vide
        retourne sous_arbre
```


Il nous manque le code pour le parent

Pseudo-code pour trouver le parent (5min -> matrix)

Il nous manque le code pour le parent

Pseudo-code pour trouver le parent (5min -> matrix)

```
arbre parent(arbre, sous_arbre)
    si est_non_vide(arbre)
        actuel = arbre
        parent = actuel
        clé = clé(sous_arbre)
        faire
            si (clé != clé(actuel))
                parent = actuel
                si clé < clé(actuel)
                    actuel = gauche(actuel)
                sinon
                    actuel = droite(actuel)
            sinon
                retour parent
        tant_que (actuel != sous_arbre)
    retourne vide
```

Le pseudo-code de la suppression

Pour un seul enfant (5min \rightarrow matrix)

Le pseudo-code de la suppression

Pour un seul enfant (5min -> matrix)

```
arbre suppression(arbre, clé)
    sous_arbre = position(arbre, clé)
    si est_vide(gauche(sous_arbre)) ou est_vide(droite(sous_arbre))
        parent = parent(arbre, sous_arbre)
        si est_vide(gauche(sous_arbre))
            si droite(parent) == sous_arbre
                droite(parent) = droite(sous_arbre)
            sinon
                gauche(parent) = droite(sous_arbre)
        sinon
            si droite(parent) == sous_arbre
                droite(parent) = gauche(sous_arbre)
            sinon
                gauche(parent) = gauche(sous_arbre)
    retourne sous_arbre
```

Le pseudo-code de la suppression

Pour au moins deux enfants (ensemble)

```
arbre suppression(arbre, clé)
    sous_arbre = position(arbre, clé) # on vérifie pas que c'est bien la clé
    si est_non_vide(gauche(sous_arbre)) et est_non_vide(droite(sous_arbre))
        max_gauche = position(gauche(sous_arbre), clé)
        échange(clé(max_gauche), clé(sous_arbre))
    suppression(gauche(sous_arbre), clé)
```

Exercices (poster sur matrix)

1. Écrire le pseudo-code de l'insertion purement en récursif.

Exercices (poster sur matrix)

1. Écrire le pseudo-code de l'insertion purement en récursif.

```
arbre insert(tree, clé)
    si est_vide(tree)
        retourne nœud(clé)

    si (clé < tree(clé))
        gauche(tree) = insert(gauche(tree), clé)
    sinon
        droite(tree) = insert(droite(tree), clé)
    retourne tree
```

Exercices (poster sur matrix)

2. Écrire le pseudo-code de la recherche purement en récursif.

Exercices (poster sur matrix)

2. Écrire le pseudo-code de la recherche purement en récursif.

```
booléen recherche(tree, clé)
    si est_vide(tree)
        retourne faux // pas trouvée
    si clé(tree) == clé
        retourne vrai // trouvée
    si clé < clé(tree)
        retourne recherche(gauche(tree), clé)
    sinon
        retourne recherche(droite(tree), clé)
```

Exercices (à la maison)

3. Écrire une fonction qui insère des mots dans un arbre et ensuite affiche l'arbre.