Arbres binaires

Algorithmique et structures de données, 2024-2025

P. Albuquerque (B410) et O. Malaspinas (A401), ISC, HEPIA

2025-03-07

En partie inspirés des supports de cours de P. Albuquerque

Les arbres binaires

L'arbre binaire

- Structure de données abstraite,
- Chaque nœud a au plus deux enfants: gauche et droite,
- Chaque enfant est un arbre.

Comment représenteriez vous une telle structure?

L'arbre binaire

- Structure de données abstraite,
- Chaque nœud a au plus deux enfants: gauche et droite,
- Chaque enfant est un arbre.

Comment représenteriez vous une telle structure?

```
<R, G, D>
    R: racine
    G: sous-arbre gauche
    D: sous-arbre droite
```

Comment cela s'écrirait en C?

L'arbre binaire

- Structure de données abstraite,
- Chaque nœud a au plus deux enfants: gauche et droite,
- Chaque enfant est un arbre.

Comment représenteriez vous une telle structure?

```
<R, G, D>
    R: racine
    G: sous-arbre gauche
    D: sous-arbre droite
```

Comment cela s'écrirait en C?

```
typedef struct _node {
    contenu info;
    struct _node *left, *right;
} node;
```

L'arbre binaire

Que se passerait-il avec

```
typedef struct _node {
    int info;
    struct _node left, right;
} node;
```

L'arbre binaire

Que se passerait-il avec

```
typedef struct _node {
    int info;
    struct _node left, right;
} node;
```

- On ne sait pas quelle est la taille de node, on ne peut pas l'allouer!

Interface minimale

- Qu'y mettriez vous?

L'arbre binaire

Que se passerait-il avec

```
typedef struct _node {
    int info;
    struct _node left, right;
} node;
```

- On ne sait pas quelle est la taille de node, on ne peut pas l'allouer!

Interface minimale

- Qu'y mettriez vous?

```
NULL                -> arbre (vide)
<n, arbre, arbre>   -> arbre
visiter(arbre)      -> nœud (la racine de l'arbre)
gauche(arbre)       -> arbre (sous-arbre de gauche)
droite(arbre)       -> arbre (sous-arbre de droite)
```

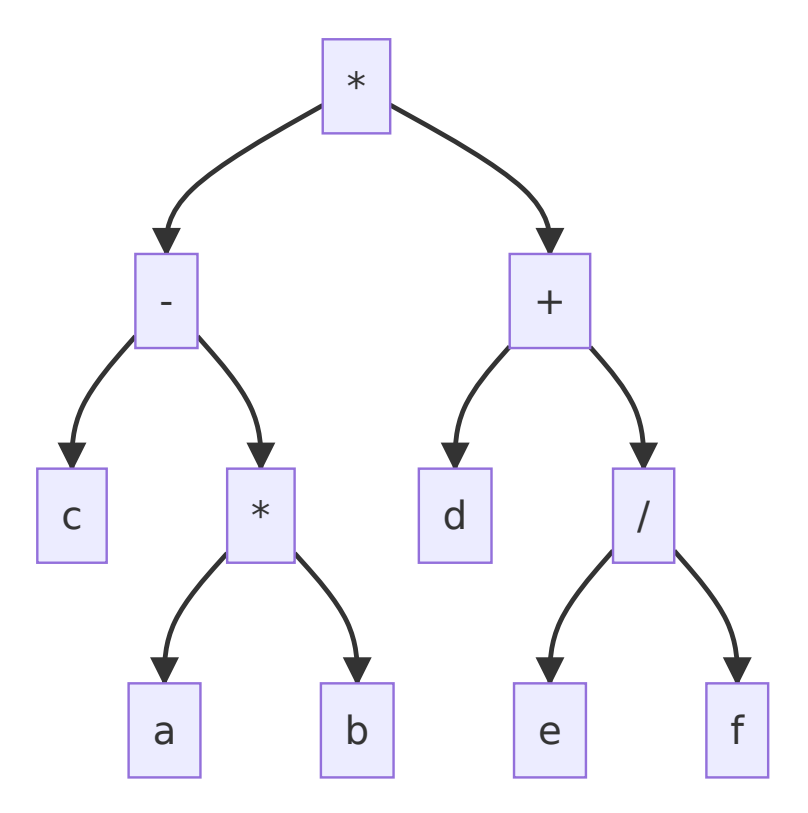
- Les autres opérations (insertion, parcours, etc) dépendent de ce qu'on stocke dans l'arbre.

Exemple d'arbre binaire

- Représentez `(c - a * b) * (d + e / f)` à l'aide d'un arbre binaire (matrix)

Exemple d'arbre binaire

- Représentez `(c - a * b) * (d + e / f)` à l'aide d'un arbre binaire (matrix)



Remarques

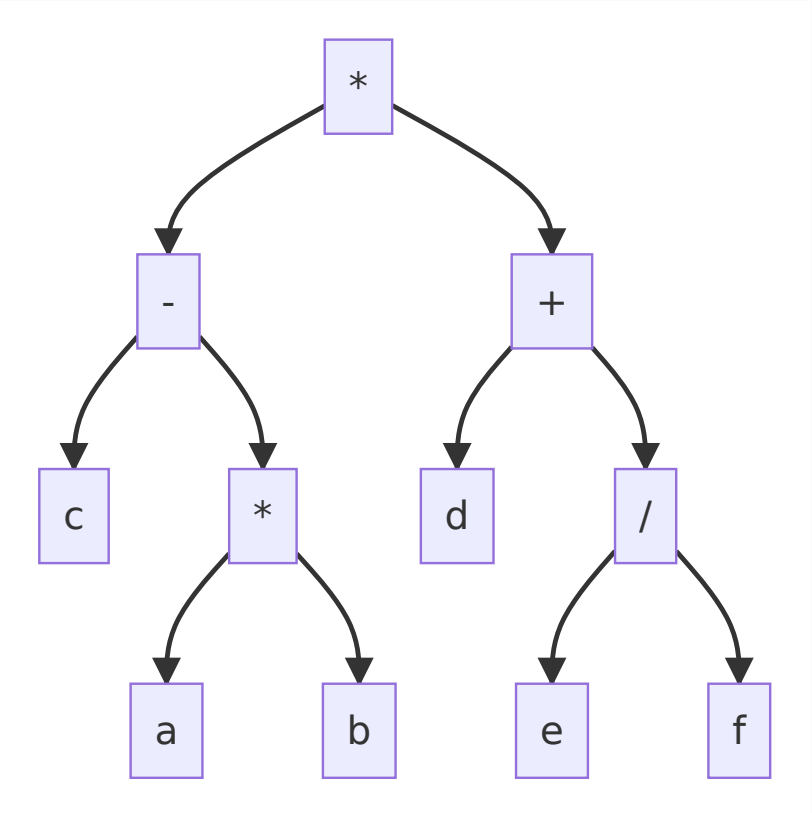
- L'arbre est **hétérogène**: le genre d'info n'est pas le même sur chaque nœud (opérateur, opérande).
  - Les feuilles contiennent les opérandes.
  - Les nœuds internes contiennent les opérateurs.

Parcours d'arbres binaires

- Appliquer une opération à tous les nœuds de l'arbre,
- Nécessité de **parcourir** l'arbre,
- Utiliser uniquement l'interface: visiter, gauche, droite.

Une idée de comment parcourir cet arbre?

- 3 parcours (R: Racine, G: sous-arbre gauche, D: sous-arbre droit):



1. Parcours **préfixe** (R, G, D),
2. Parcours **infixe** (G, R, D),
3. Parcours **postfixe** (G, D, R).

Le parcours infixe (G, R, D)

- Gauche, Racine, Droite:
  1. On descend dans l'arbre de gauche tant qu'il n'est pas vide.
  2. On visite la racine du sous arbre.
  3. On descend dans le sous-arbre de droite (s'il n'est pas vide).
  4. On recommence.

Le parcours infixe (G, R, D)

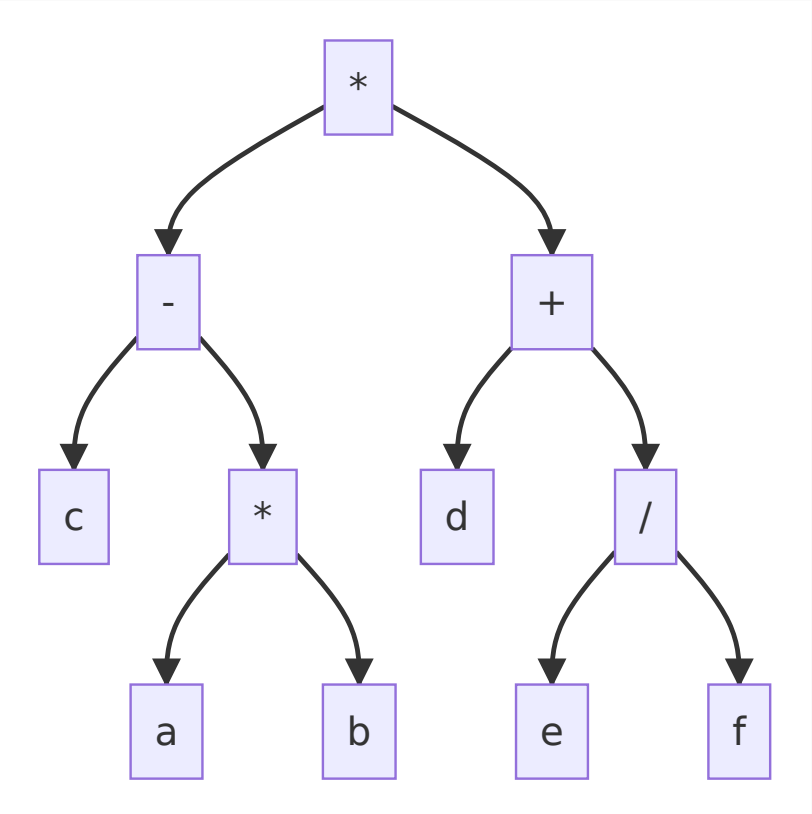
- Gauche, Racine, Droite:
  1. On descend dans l'arbre de gauche tant qu'il n'est pas vide.
  2. On visite la racine du sous arbre.
  3. On descend dans le sous-arbre de droite (s'il n'est pas vide).
  4. On recommence.

Incompréhensible?

- La récursivité, c'est la vie.

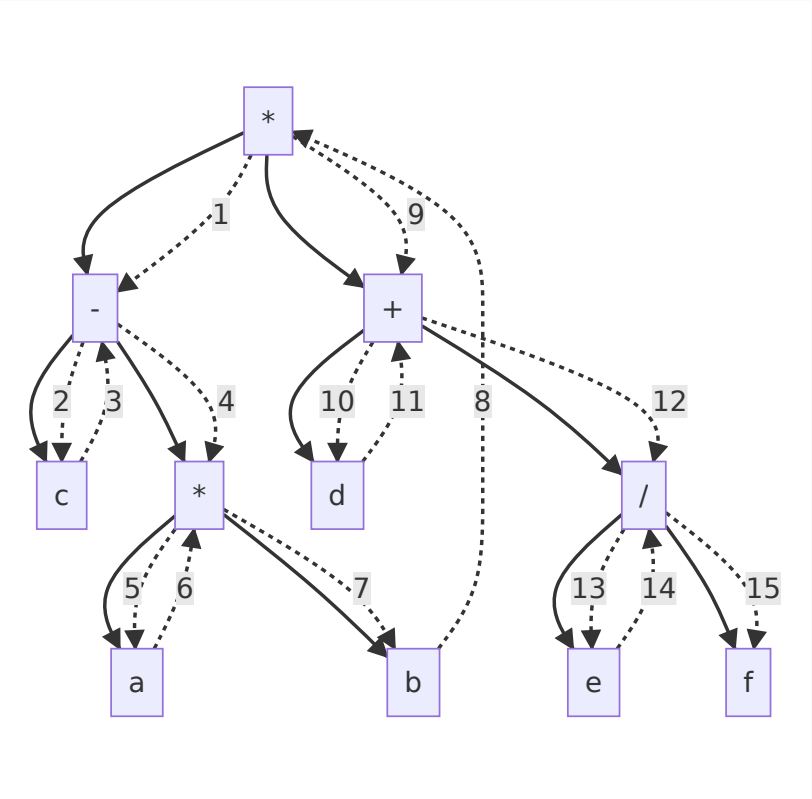
```
parcours_infixe(arbre a)
   si est_pas_vide(gauche(a))
      parcours_infixe(gauche(a))
   visiter(a)
   si est_pas_vide(droite(a))
      parcours_infixe(droite(a))
```

Graphiquement (dessinons)




```
parcours_infixe(arbre a)
   si est_pas_vide(gauche(a))
      parcours_infixe(gauche(a))
   visiter(a)
   si est_pas_vide(droite(a))
      parcours_infixe(droite(a))
```

Graphiquement (`mermaid` c'est super)




```
parcours_infixe(arbre a)
   si est_pas_vide(gauche(a))
      parcours_infixe(gauche(a))
   visiter(a)
   si est_pas_vide(droite(a))
      parcours_infixe(droite(a))
```

Remarque

Le nœud est visité à la **remontée**.

Résultat

`c - a * b * d + e / f`

Et en C?

Live code

Et en C?

Live code

```
typedef int data;
typedef struct _node {
    data info;
    struct _node* left;
    struct _node* right;
} node;
void tree_print(node *tree, int n) {
    if (NULL != tree) {
        tree_print(tree->left, n+1);
        for (int i = 0; i < n; i++) {
            printf(" ");
        }
        printf("%d\n", tree->info);
        tree_print(tree->right, n+1);
    }
}
```

Question

Avez-vous compris le fonctionnement?

Question

Avez-vous compris le fonctionnement?

Vous en êtes sûr · e · s?

Question

Avez-vous compris le fonctionnement?

Vous en êtes sûr · e · s?

OK, alors deux exercices:

1. Écrire le pseudo-code pour le parcours R, G, D (matrix).
2. Écrire le pseudo-code pour la parcours G, D, R (matrix).

Rappel

```
parcours_infixe(arbre a)
    si est_pas_vide(gauche(a))
       parcours_infixe(gauche(a))
    visiter(a)
    si est_pas_vide(droite(a))
       parcours_infixe(droite(a))
```

Correction

- Les deux parcours sont des modifications **triviales** de l'algorithme infixe.

Le parcours postfixe

```
parcours_postfixe(arbre a)
    si est_pas_vide(gauche(a))
       parcours_postfixe(gauche(a))
    si est_pas_vide(droite(a))
       parcours_postfixe(droite(a))
    visiter(a)
```

Le parcours préfixe

```
parcours_préfixe(arbre a)
    visiter(a)
    si est_pas_vide(gauche(a))
        parcours_préfixe(gauche(a))
    si est_pas_vide(droite(a))
        parcours_préfixe(droite(a))
```

Correction

- Les deux parcours sont des modifications **triviales** de l'algorithme infixe.

Le parcours postfixe

```
parcours_postfixe(arbre a)
    si est_pas_vide(gauche(a))
       parcours_postfixe(gauche(a))
    si est_pas_vide(droite(a))
       parcours_postfixe(droite(a))
    visiter(a)
```

Le parcours préfixe

```
parcours_préfixe(arbre a)
    visiter(a)
    si est_pas_vide(gauche(a))
        parcours_préfixe(gauche(a))
    si est_pas_vide(droite(a))
        parcours_préfixe(droite(a))
```

Attention: L'implémentation de ces fonctions en C sont **à faire** en exercice (inspirez vous de ce qu'on a fait avant)!

Exercice: parcours

Comment imprimer l'arbre ci-dessous?

```
                    f
          /
                    e
    +
          d
*
          c
    -
                    b
          *
                    a
```

Exercice: parcours

Comment imprimer l'arbre ci-dessous?

```
                    f
          /
                    e
    +
          d
*
          c
    -
                    b
          *
                    a
```

Bravo vous avez trouvé!

- Il s'agissait du parcours D, R, G.

Implémentation

Vous avez 5 min pour implémenter cette fonction et la poster sur matrix!

Implémentation

Vous avez 5 min pour implémenter cette fonction et la poster sur matrix!

```
void pretty_print(node *tree, int n) {
    if (NULL != tree) {
        pretty_print(tree->right, n+1);
        for (int i = 0; i < n; ++i) {
            printf(" ");
        }
        printf("%d\n", tree->info);
        pretty_print(tree->left, n+1);
    }
}
```

Exercice supplémentaire (sans corrigé)

Écrire le code de la fonction

```
int depth(node *t);
```

qui retourne la profondeur maximale d'un arbre.

Indice: la profondeur à chaque niveau peut-être calculée à partir du niveau des sous-arbres de gauche et de droite.

La recherche dans un arbre binaire

- Les arbres binaires peuvent retrouver une information très rapidement.
- À quelle complexité? À quelle condition?

La recherche dans un arbre binaire

- Les arbres binaires peuvent retrouver une information très rapidement.
- À quelle complexité? À quelle condition?

Condition

- Le contenu de l'arbre est **ordonné** (il y a une relation d'ordre (<, > entre les éléments).

Complexité

- La profondeur de l'arbre (ou le $\mathcal{O}(\log_2(N))$)

La recherche dans un arbre binaire

- Les arbres binaires peuvent retrouver une information très rapidement.
- À quelle complexité? À quelle condition?

Condition

- Le contenu de l'arbre est **ordonné** (il y a une relation d'ordre (<, > entre les éléments).

Complexité

- La profondeur de l'arbre (ou le $\mathcal{O}(\log_2(N))$)

Exemple: les arbres lexicographiques

- Chaque nœud contient une information de type ordonné, la **clé**.
- Par construction, pour chaque nœud N:
  - Toute clé du sous-arbre à gauche de N est inférieure à la clé de N.
  - Toute clé du sous-arbre à droite de N est inférieure à la clé de N.

Algorithme de recherche

- Retourner le nœud si la clé est trouvée dans l'arbre.

```
tree recherche(clé, arbre)
    tant_que est_non_vide(arbre)
        si clé < clé(arbre)
            arbre = gauche(arbre)
        sinon si clé > clé(arbre)
            arbre = droite(arbre)
        sinon
            retourne arbre
    retourne NULL
```

Algorithme de recherche, implémentation (live)

Algorithme de recherche, implémentation (live)

```c
typedef int key_t;
typedef struct _node {
    key_t key;
    struct _node* left;
    struct _node* right;
} node;
node *search(key_t key, node *tree) {
    node *current = tree;
    while (NULL != current) {
        if (current->key > key) {
            current = current->left;
        } else if (current->key < key){
            current = current->right;
        } else {
            return current;
        }
    }
    return NULL;
}
```

Exercice (5-10min)

Écrire le code de la fonction

```
int tree_size(node *tree);
```

qui retourne le nombre total de nœuds d'un arbre et poster le résultat sur matrix.

Indication: la taille, est 1 + le nombre de nœuds du sous-arbre de gauche additionné au nombre de nœuds dans le sous-arbre de droite.

Exercice (5-10min)

Écrire le code de la fonction

```
int tree_size(node *tree);
```

qui retourne le nombre total de nœuds d'un arbre et poster le résultat sur matrix.

Indication: la taille, est 1 + le nombre de nœuds du sous-arbre de gauche additionné au nombre de nœuds dans le sous-arbre de droite.

```
int tree_size(node *tree) {
    if (NULL == tree) {
        return 0;
    } else {
        return 1 + tree_size(tree->left)
            + tree_size(tree->right);
    }
}
```

L'insertion dans un arbre binaire

- C'est bien joli de pouvoir faire des parcours, recherches, mais si on ne peut pas construire l'arbre....

Pour un arbre lexicographique

- Rechercher la position dans l'arbre où insérer.
- Créer un nœud avec la clé et le rattacher à l'arbre.

Exemple d'insertions

- Clés uniques pour simplifier.
- Insertion de 5, 15, 10, 25, 2, -5, 12, 14, 11.
- Rappel:
  - Plus petit que la clé courante => gauche,
  - Plus grand que la clé courante => droite.
- Faisons le dessins ensemble

Exercice (3min, puis matrix)

- Dessiner l'arbre en insérant 20, 30, 60, 40, 10, 15, 25, -5

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

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

Pseudo-code d'insertion (2/4)

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

Ajout de l'enfant

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

Code d'insertion en C

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

L'insertion (4/4)

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)

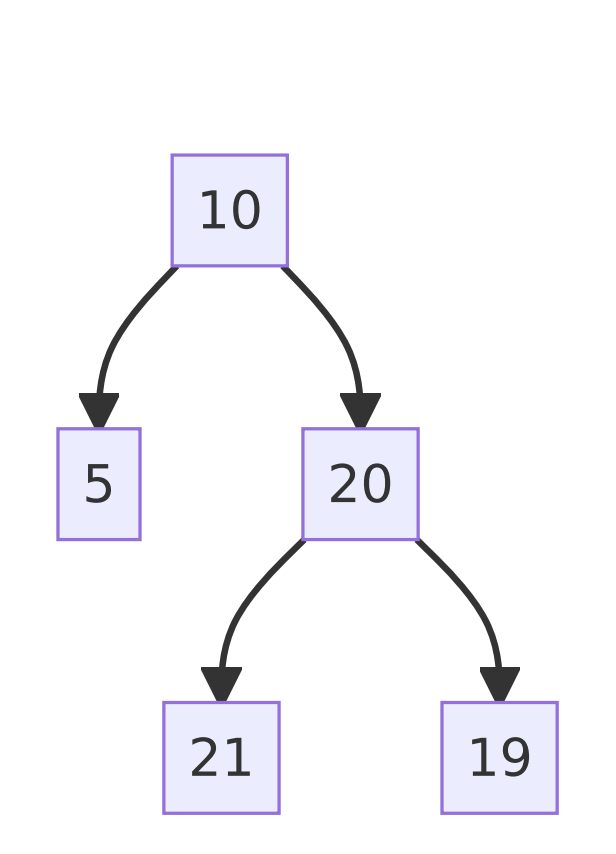
```
node *add_key(node **tree, key_t key) {
    node *new_node = calloc(1, sizeof(*new_node));
    new_node->key = key;
    if (NULL == *tree) {
        *tree = new_node;
    } else {
        node * subtree = position(*tree, key);
        if (key == subtree->key) {
            return NULL;
        }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

La suppression de clé

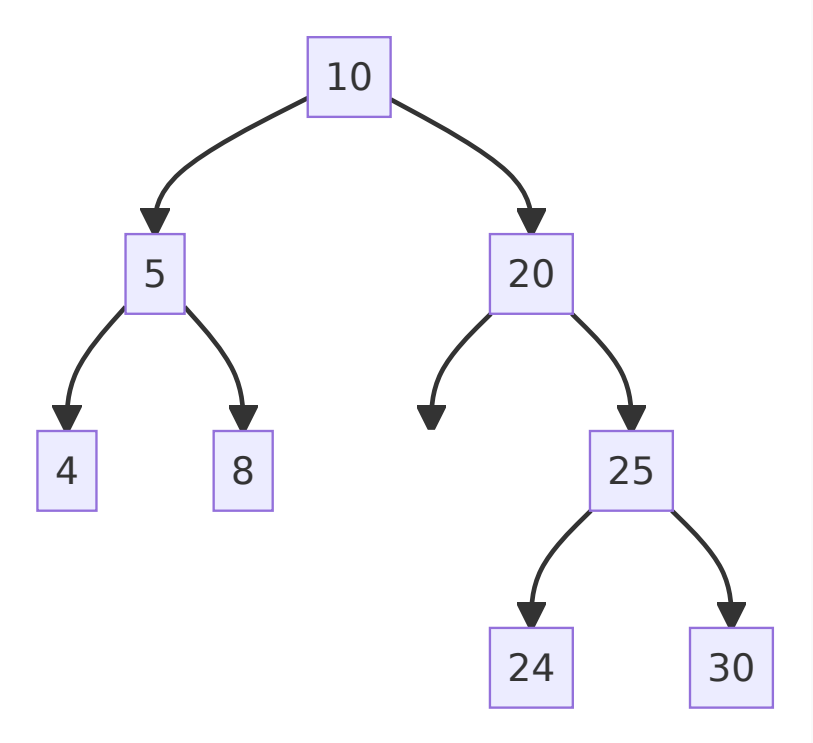
Cas simples:

- 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.).



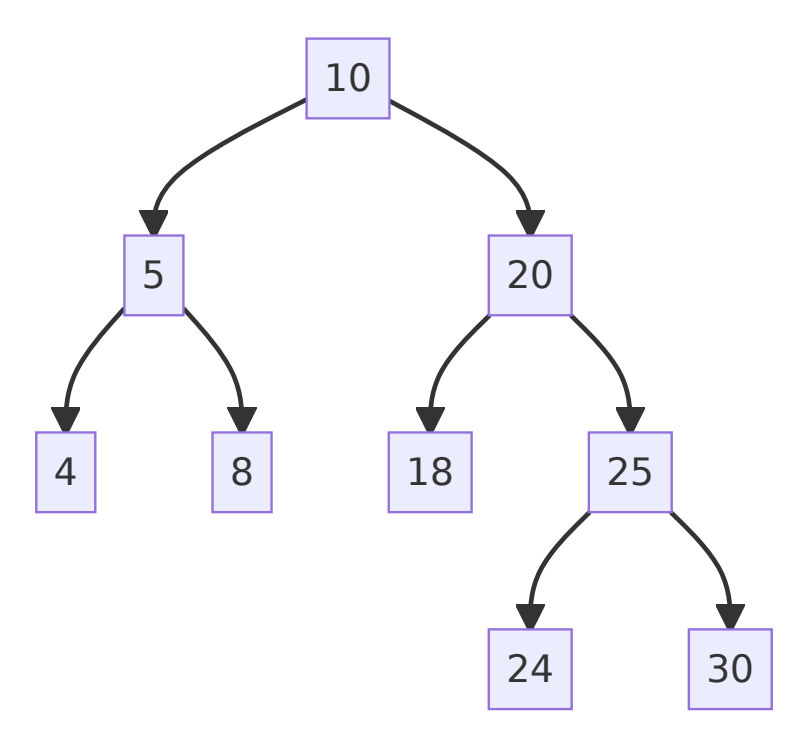
Dans tous les cas

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

La suppression de clé

Cas compliqué

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

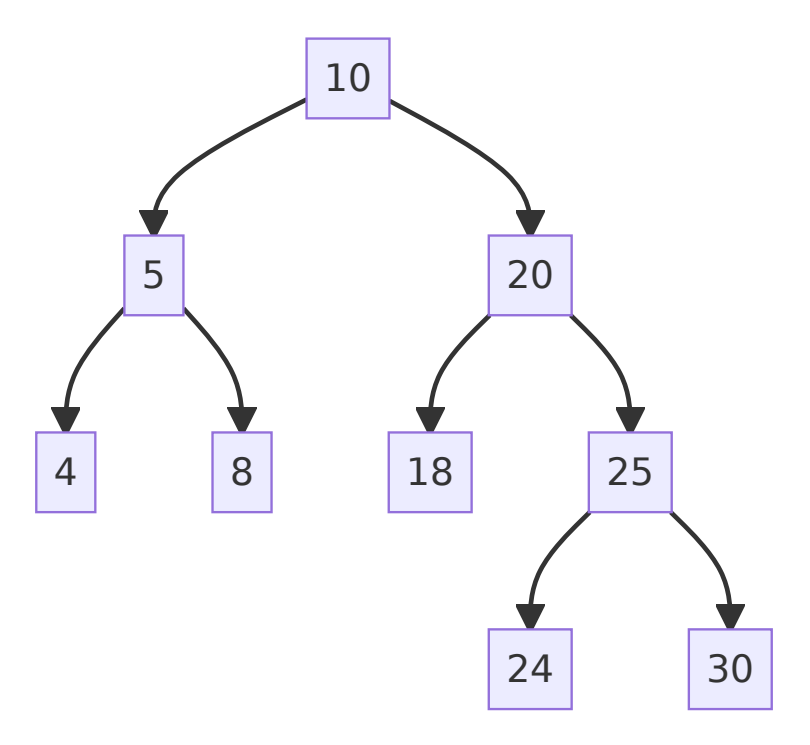


- Si on enlève 10, il se passe quoi?

La suppression de clé

Cas compliqué

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

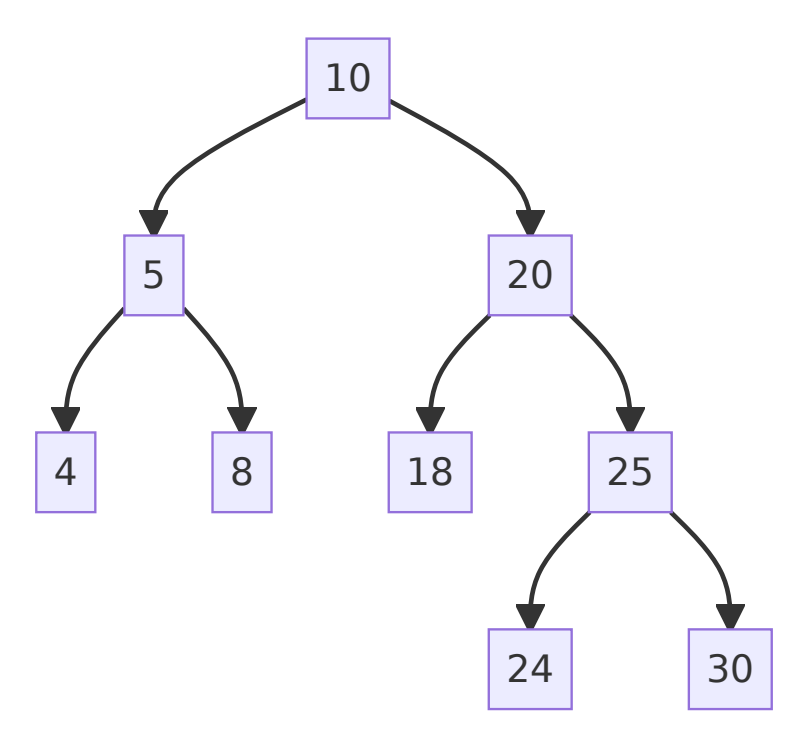


- Si on enlève 10, il se passe quoi?
- On ne peut pas juste enlever 10 et recoller...
- Proposez une solution !

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)

```
tree 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)

```
tree 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 -> matrix)

Le pseudo-code de la suppression

Pour un seul enfant (5min -> matrix)

```
tree 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)

```
tree suppression(arbre, clé)
    sous_arbre = position(arbre, clé) # on revérifie 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.

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

    si (clé < arbre->clé)
        gauche(arbre) = insert(gauche(arbre), clé)
    sinon
        droite(arbre) = insert(droite(arbre), clé)
    retourne arbre
```

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

Exercices (à la maison)

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