TP1 Méthodes Numériques

Ali Ghammaz Jérémy Pucci Eloi Charra

21/02/2022

1 Pile

Lors de l'initialisation d'une pile, on place le sommet à `0`. Pour tester qu'une pile est vide on regarde donc si son sommet est à `0`. Le dépilement vérifie premièrement que la pile n'est pas vide, ensuite nous récupérons l'élément au sommet de la pile que nous stockons pour le retourner après avoir décrémenté le sommet. L'empilement vérifie que la pile n'est pas pleine, pour ensuite ajouter l'élément au niveau du sommet et incrémenter le sommet de `1`.

2 File

Nous avons choisi de créer une file circulaire. Nous positionnons la tête et la queue de la file à `-1` à la création. Tester si une file est vide consiste à regarder si la tête est positionnée à `-1`. La tête est à `-1` lors de la création de la file ou lorsqu'elle est devant la queue après un défilement (il n'y a plus d'élément dans la file). Enfiler un élément consiste à mettre la tête à `0` si elle était à `-1` pour ensuite insérer l'élément à l'emplacement suivant où pointe actuellement la queue. Comme c'est une file circulaire, nous ajoutons `% MAX_FILE_SIZE` pour revenir au "début" si nous avons atteint le "bout" de la file. Défiler un élément s'assure que la file n'est pas vide pour ensuite récupérer l'élément pointé par la tête. Nous incrémentons la tête de la même manière que la queue et nous testons si la tête est supérieure à la queue. Si c'est le cas, nous mettons la tête et la queue à `-1` (la file est vide).

3 ABR

3.1 Parcourir arbre largeur

Nous utilisons une file pour cet algorithme. Nous insérons la racine de l'arbre dans la file. Ensuite tant que la file n'est pas vide, on défile l'élément que nous affichons pour ensuite enfiler ses fils droit et gauche s'ils existent.

3.2 Hauteur arbre

3.2.1 Récursif

La version récursive consiste à retourner le plus grand sous-arbre parmi les deux fils de l'arbre courant plus `1`. La hauteur est donc calculée pour tous les noeuds, si un arbre est `NULL` on retire `1` au résultat pour compenser la descente sur l'arbre `NULL` en question.

3.2.2 Non récursif

Pour la version non récursive nous utilisons deux files. On commence par enfiler la racine de l'arbre dans la première file. Pour chaque élément de cette file (tant qu'elle n'est pas vide), on enfile ses deux fils dans la seconde file. Ensuite, on enfile dans la première tous les éléments de la seconde et on rajoute `1` à la variable qui compte la hauteur. La première file va donc contenir tous les éléments d'un niveau à la fois.

3.3 Nombre de clés

3.3.1 Récursif

Cet algorithme se rapproche de celui de la hauteur sauf qu'on ne prend pas le nombre le plus grand entre les deux fils mais on les additionne.

3.3.2 Non récursif

Nous utilisons une file que nous parcourons jusqu'à qu'elle soit vide. Pour chaque élément de la file, nous enfilons ses deux fils s'ils existent. Pour chaque élément défilé, nous incrémentons le compteur de clé de `1`.

3.4 Liste clés triées

3.4.1 Récursif

Nous faisons ici un simple parcours infixe puisque c'est comme cela que les clés sont triées.

3.4.2 Non récursif

Nous utilisons une pile pour parcourir l'arbre ainsi qu'un arbre temporaire. Tant que notre variable temporaire n'est pas `NULL` (fin d'arbre), on empile cet arbre et nous passons à son fils gauche. Ensuite, nous dépilons un élément que nous affichons et nous passons à son fils droit. Ceci est répété tant que la pile n'est pas vide ou que l'arbre actuellement parcouru n'est pas `NULL`.

3.5 Arbre plein

Nous vérifions que l'arbre possède $2^{hauteur-1} - 1$ noeuds.

3.6 Arbre parfait

Nous vérifions que l'arbre possède $\lfloor \log_2(noeuds) \rfloor$ comme hauteur.

3.7 Recherche clé supérieure

Si la valeur est supérieure à la racine de l'arbre nous rapellons cette fonction avec le fils droit et la même valeur. Sinon, elle se situe dans l'arbre gauche ou c'est la racine. Nous rapellons cette fonction sur le fils gauche de la racine et nous comparons les clés de la racine et du noeud sortie. Si la clé du noeud obtenu est égale à la valeur, nous nous arrêtons ici. Sinon, nous sélectionnons la clé la plus proche de la valeur.

3.8 Recherche clé inférieure

Si la valeur est inférieure à la clé de la racine, nous rapellons cette fonction sur l'arbre gauche. Sinon, on récupère la clé directement inférieure à la valeur dans le sous-arbre droit. Et nous la comparons avec la clé de la racine.

3.9 Intersection

Pour l'intersection nous récupérons toutes les clés des deux arbres dans deux tableaux. Nous utilisons ensuite deux boucles `for` imbriquées pour comparer chacunes des valeurs des deux tableaux entre elles. Si une valeur est dans les deux tableaux à la fois nous ajoutons la clé dans un arbre résultat. Nous n'avons pas besoin de nous soucier des doublons ici puisque la fonction `ajouter_cle` le fait déjà.

3.10 Union

L'union est la même base que l'intersection. La différence est que nous n'utilisons pas de boucles imbriquées pour réaliser une comparaison mais deux boucles séparées pour inclure tous les élements des deux tableaux.

3.11 Destruction d'une clé

Nous cherchons premièrement le noeud portant la clé voulant être suprimée. Une fois trouvée, nous avons trois cas :

- Le noeud est une feuille : suppression du noeud sans conséquence
- Le noeud possède un fils : remplaçement du noeud retiré par son fils
- Le noeud possède deux fils : on récupère la clé la plus petite dans son fils droit qu'on met à la place du noeud qui sera détruit. Ensuite, nous appliquons encore l'algorithme sur le nouveau noeud qui est maitenant une feuille ou un noeud à 1 fils.

4 AVL

Nous avons ajouté un champ `balance` dans la structure qui permet de décrire un AVL. Cette balance indique l'équilibrage de ses sous-arbres.

```
typedef struct n_AVL {
  int cle;
  int bal;
  struct n_AVL *fgauche, *fdroite;
} noeud_AVL_t, *pnoeud_AVL_t;
```

Nous avons créé une fonction `convertToAVL` qui permet de convertir un ABR en AVL. Cette fonction rajoute juste le champ `bal` en calculant la balance pour chaque noeud.

4.1 Rotations simples

4.1.1 Rotation gauche

Nous crééons deux arbres. L'un est le sous-arbre droit de l'arbre originel, l'autre est le sous-arbre gauche de celui que nous venons de créer. Ensuite, le sous-arbre gauche du premier est l'arbre original dont sa partie droite a été remplacée par le second arbre créé. Il suffit enfin d'actualiser les balances.

4.1.2 Rotation droite

La rotation droite est le même principe que la rotation gauche. La différence est qu'il faut inverser le choix des arbres (`gauche <-> droit`)

4.2 Double rotations

La double rotation gauche consiste à remplacer le sous-arbre droit par une rotation droit de lui-même pour enfin effectuer une rotation gauche sur le tout.
C'est le même principe pour la rotation droite en inversant les arbres.

4.3 Rééquilibrage

Le rééquilibrage consiste à regarder si la balance est équilibrée. Si elle ne l'est pas, on applique la bonne rotation (à gauche ou à droite).

4.4 Ajout clé

L' ajout de clé se base sur le même principe que l'ajout de clé pour un ABR. Il suffit seulement d'ajouter les balances et rééquilibrer le tout si besoin.