Rapport TP4

ARBRES BINAIRES

DOAN Nhat-Minh / HU Shuohui

Partie 1 : Arbres binaires de recherche

A. Structures

```
typedef struct Noeud{
    int val;
    struct Noeud *gauche;
    struct Noeud *droit;
}T_Noeud;

typedef T_Noeud *T_Arbre;
```

B. Fonctions requises

Dans ce TP, on a utilisé les fonctions suivantes :

1. Création d'un nœud :

```
T_Noeud *abr_creer_noeud(int valeur);
```

Complexité O(1) ;

2. Affichage préfixe d'un arbre binaire de recherche :

```
void abr_prefixe(T_Arbre abr);
```

Complexité :
- meilleur cas : O(1) : il n'y a qu'un seul nœud

- pire cas : O(n) : n est le nombre de nœuds dans l'arbre

3. Insertion d'une valeur dans un arbre binaire de recherche :

```
void abr_inserer(int valeur,T_Arbre *abr);
```

Complexité :

- meilleur cas : O(1) : on appelle une fois la fonction abr_creer_noeud de O(1)

- pire cas : O(k) : k est le nombre de père de la valeur ajoute

4. Suppression d'une valeur dans un arbre binaire de recherche :

Ici, on utilise une fonction supplémentaire pour chercher la valeur a supprimer :

```
T_Noeud *chercher_valeur(int valeur,T_Arbre abr);
```

Complexité :

- pire cas : O(k) : comme la fonction insertion

- meilleur cas : O(1) : la valeur est aussi la racine

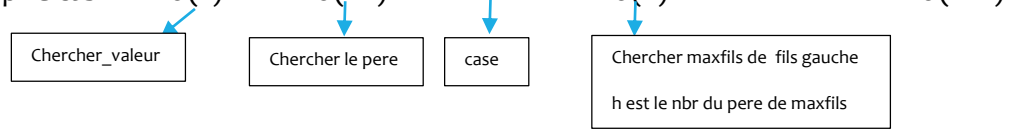
Suppression :

```
void abr_supprimer(int valeur,T_Arbre *abr);
```

Complexité :

- meilleur cas : O(k) : a partir de la fonction recherche et la valeur a supprimer n'existe pas .

- pire cas : O(k) + O(k-1) + 1 + O(h) = O(h+k)



5. Créer une copie

```
void abr_clone(T_Arbre original, T_Arbre *clone, T_Noeud* parent);
```

Complexité :

- meilleur cas : O(1) : on appelle une fois la fonction création

- pire cas : O(n) : n est le nbr de nœuds

Partie 2 : Arbres binaires cousus

A. Structures

```
typedef struct Noeud_C{
    int val;
    struct Noeud_C *gauche;
    struct Noeud_C *droit;
    int gauchePre;// 0 vrai 1 faux
    int droitSuc; // 0 vrai 1 faux
}T_Noeud_C;

typedef T_Noeud_C *T_Arbre_C;
```

B. Fonctions requises

1. Création d'un nœud :

```
T_Noeud_C *cousu_creer_noeud(int valeur);
```

Complexité : O(1)

2. Affichage préfixe d'un arbre binaire cousu :

Ici, on a utilisé les fonctions supplémentaires suivantes :

```
T_Noeud_C *chercher_valeur_C(int valeur,T_Arbre_C abr);
```

Complexité :

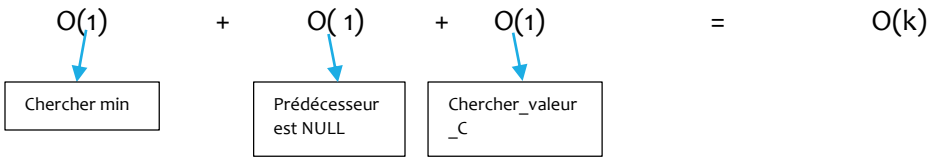
- pire cas : O(k) : comme la fonction insertion

- meilleur cas : O(1) : la valeur est aussi la racine

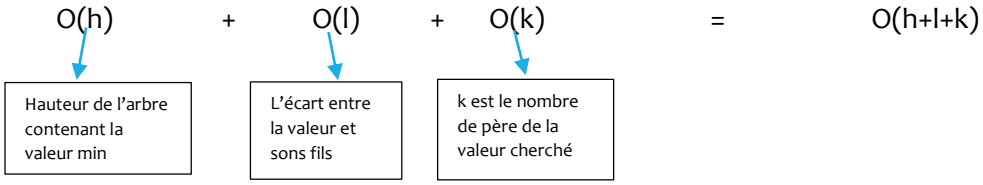
```
int chercherPre(int valeur,T_Arbre_C arbre);// chercher le prédécesseur
int chercherSuc(int valeur,T_Arbre_C arbre);// chercher le successeur
```

Complexité (pour chercherPre et pareil pur chercherSuc) :

- meilleur cas : arbre un nœud :



- pire cas :




```
void sous_cousu_prefixe(T_Arbre_C arbreT,int pre,int suc, T_Arbre_C arbre);
```

Complexité :

- meilleur cas : arbre un nœud : O(1)

- pire cas : O(n) : n est le nombre de nœuds dans l'arbre

```
void cousu_prefixe(T_Arbre_C arbre);
```

- meilleur cas : arbre un nœud : O(1)

- pire cas : O(n) : n est le nombre de nœuds dans l'arbre

3. Insertion d'une valeur dans un arbre binaire cousu :

```
void cousu_inserer(int valeur,T_Arbre_C *arbre);
```

Complexité :

- meilleur cas : O(1) : on appelle une fois la fonction cousu_creer_noeud de O(1)

- pire cas : O(k) : k est le nombre de père de la valeur ajoute

4. Affichage en parcours infixe d'un arbre binaire cousu :

```
void cousu_infixe(T_Arbre_C *arbre);
```

- meilleur cas : O(1) : arbre un nœud

- pire cas : O(n) : nbr de nœuds dans l'arbre

5. Implémenter la fonction qui permet de créer un arbre cousu « cousu » à partir d'un arbre binaire de recherche « abr »

```
void abr_to_cousu(T_Arbre abr, T_Arbre_C *clone, T_Noeud_C* parent) ;
```

Complexité :

- meilleur cas : O(1) : on appelle une fois la fonction création

- pire cas : O(n) : n est le nbr de nœuds