LECLERCQ Vivien - CHEN Jian

RAPPORT

TP4 NF16

Liste des structures utilisées:

```c
struct Intervalle{
    int min;
    int max;
};

typedef struct Intervalle T_Inter;

struct Noeud{
    int id_entreprise;
    T_Inter intervalle;
    struct Noeud *fils_gauche;
    struct Noeud *fils_droit;
    int hauteur;
    int equilibre;
};

typedef struct Noeud T_Noeud;

typedef T_Noeud* T_Arbre;
```

T_Inter:

Nous avons choisi d'utiliser la << structure T_Inter >> pour représenter l'intervalle de temps dans la << structure T_noeud >>.

Où << min >> représente la valeur minimale de l'intervalle, c'est-à-dire la date de début, qui est représentée par le type << int >>, les données sont enregistrées sous la forme de MMJJ.

De même, << max >> représente la valeur maximale de l'intervalle, c'est-à-dire la date de fin, et est également représenté par le type << int >>, les données sont enregistrées sous la forme MMJJ.

T_Noeud:

Nous avons utilisé la variable << id_entreprise >> de type int pour représenter l'id de chaque nœud, et différentes valeurs d'id signifient différentes entreprises

Nous avons utilisé des pointeurs de structure << struct Noeud * fils_gauche, * fils_droit >> pour représenter les sous-arbres gauche et droit d'un nœud, et les pointeurs de sous-arbre gauche et droit des nœuds feuilles doivent pointer vers l'adresse du T_Noeud

Chaque nœud a une hauteur, << int >> une variable de type, << hauteur >> enregistre la hauteur de chaque nœud, la hauteur du nœud feuille est 0 par défaut.

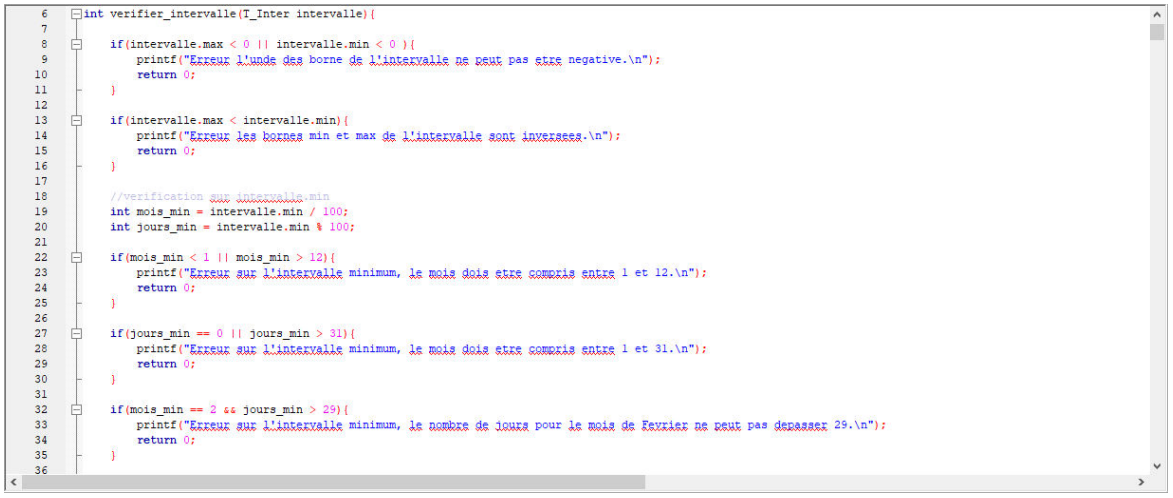
Nous avons utilisé une variable de type << int >>, << equilibre >> pour représenter la différence de hauteur entre les sous-arbres gauche et droit d'un arbre, et elle n'est équilibrée que lorsque la valeur absolue de la différence de hauteur est inférieure à 2.

T_Arbre:

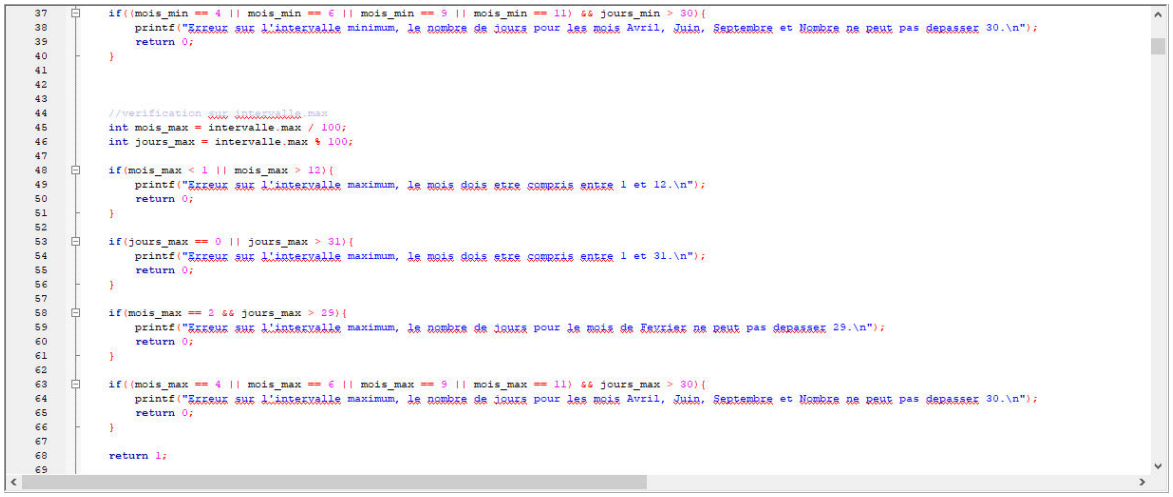
Un pointeur vers la structure T_noeud, pointant vers le nœud racine de l'arbre.

Listes des fonctions non demandées utilisées:

Dans tp4_abr.c:

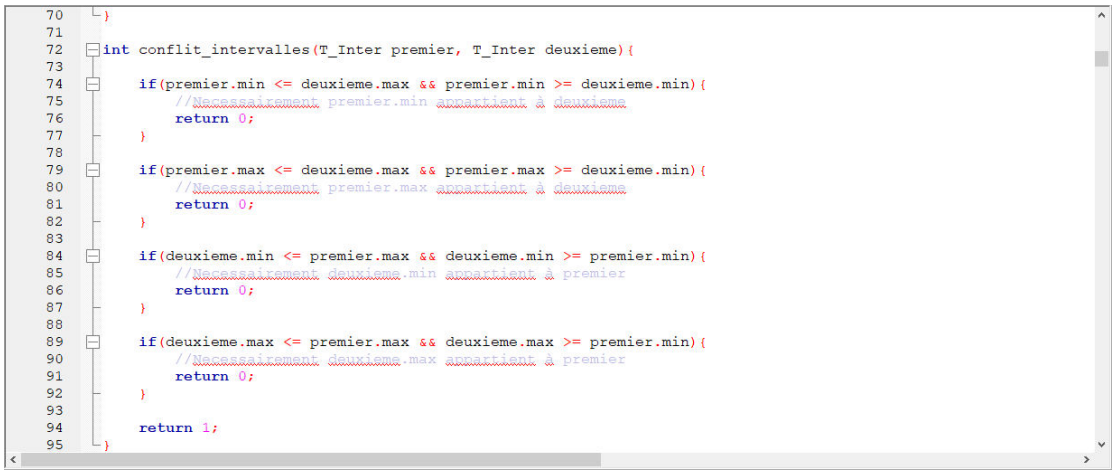


```c
int verifier_intervalle(T_Inter intervalle){

    if(intervalle.max < 0 || intervalle.min < 0 ){
        printf("Erreur l'unde des borne de l'intervalle ne peut pas etre negative.\n");
        return 0;
    }

    if(intervalle.max < intervalle.min){
        printf("Erreur les bornes min et max de l'intervalle sont inversees.\n");
        return 0;
    }

    //verification sur intervalle.min
    int mois_min = intervalle.min / 100;
    int jours_min = intervalle.min % 100;

    if(mois_min < 1 || mois_min > 12){
        printf("Erreur sur l'intervalle minimum, le mois dois etre compris entre 1 et 12.\n");
        return 0;
    }

    if(jours_min == 0 || jours_min > 31){
        printf("Erreur sur l'intervalle minimum, le mois dois etre compris entre 1 et 31.\n");
        return 0;
    }

    if(mois_min == 2 && jours_min > 29){
        printf("Erreur sur l'intervalle minimum, le nombre de jours pour le mois de Fevrier ne peut pas depasser 29.\n");
        return 0;
    }
```



```c
    if((mois_min == 4 || mois_min == 6 || mois_min == 9 || mois_min == 11) && jours_min > 30){
        printf("Erreur sur l'intervalle minimum, le nombre de jours pour les mois Avril, Juin, Septembre et Nombre ne peut pas depasser 30.\n");
        return 0;
    }



    //verification sur intervalle.max
    int mois_max = intervalle.max / 100;
    int jours_max = intervalle.max % 100;

    if(mois_max < 1 || mois_max > 12){
        printf("Erreur sur l'intervalle maximum, le mois dois etre compris entre 1 et 12.\n");
        return 0;
    }

    if(jours_max == 0 || jours_max > 31){
        printf("Erreur sur l'intervalle maximum, le mois dois etre compris entre 1 et 31.\n");
        return 0;
    }

    if(mois_max == 2 && jours_max > 29){
        printf("Erreur sur l'intervalle maximum, le nombre de jours pour le mois de Fevrier ne peut pas depasser 29.\n");
        return 0;
    }

    if((mois_max == 4 || mois_max == 6 || mois_max == 9 || mois_max == 11) && jours_max > 30){
        printf("Erreur sur l'intervalle maximum, le nombre de jours pour les mois Avril, Juin, Septembre et Nombre ne peut pas depasser 30.\n");
        return 0;
    }

    return 1;
```

1. Nous avons utilisé la fonction **verifier_intervalle** pour vérifier que le format de date enregistré est correct.



```
}

int conflit_intervalles(T_Inter premier, T_Inter deuxieme){

    if(premier.min <= deuxieme.max && premier.min >= deuxieme.min){
        //Necessairement premier.min appartient à deuxieme
        return 0;
    }

    if(premier.max <= deuxieme.max && premier.max >= deuxieme.min){
        //Necessairement premier.max appartient à deuxieme
        return 0;
    }

    if(deuxieme.min <= premier.max && deuxieme.min >= premier.min){
        //Necessairement deuxieme.min appartient à premier
        return 0;
    }

    if(deuxieme.max <= premier.max && deuxieme.max >= premier.min){
        //Necessairement deuxieme.max appartient à premier
        return 0;
    }

    return 1;
}
```

2. Nous avons utilisé la fonction **conflit_intervalles** pour nous assurer que deux intervalles de dates ne se chevauchent pas.

```
}

int max(int a,int b){
    if(a>=b){
        return a;
    }
    else{
        return b;
    }
}

int min(int a,int b){
    if(a<=b){
        return a;
    }
    else{
        return b;
    }
}
```

3. On a utilisé la fonction **min** et **max** pour obtenir la valeur minimum et la valeur maximum entre deux nombres.

```
}
T_Arbre* rotDroite(T_Arbre* a, T_Noeud *x, int *delta){

    if(x->fils_gauche != NULL){

        x->hauteur = calculerHauteur(&x);

        T_Noeud *y = x->fils_gauche;
        T_Noeud *C = y->fils_droit;

        T_Noeud *pereX = pere(*a, x);
        if(pereX == x){
            pereX = NULL;
        }

        if(pereX != NULL){
            if(pereX->fils_gauche == x){
                pereX->fils_gauche = y;
            }
            else{
                pereX->fils_droit = y;
            }
        }
        else{
            *a = y;
        }

        y->fils_droit = x;
        x->fils_gauche = C;


        //Mise a jour de la hauteur dans l'ordre

        int h = x->hauteur;
        int hB; //hauteur de B
        int hC; //hauteur de C
        int hD; //hauteur de D

```

```

        T_Noeud *B = y->fils_gauche;
        if(B != NULL){
            hB = B->hauteur;
        }
        else{
            hB = -1;
        }

        if(C != NULL){
            hC = C->hauteur;
        }
        else{
            hC = -1;
        }

        T_Noeud *D = x->fils_droit;
        if(D != NULL){
            hD = D->hauteur;
        }
        else{
            hD = -1;
        }

        //x->hauteur = max(hC, hD) + 1;
        y->hauteur = (max(hB, calculerHauteur(&(y->fils_droit))) + 1);

        delta = y->hauteur - h;

        T_Noeud *i = y;
        T_Noeud *v = pere(*a, i);

        T_Noeud *j = NULL;
        int hJ; //hauteur de J

        while(v != NULL && delta != 0){

            if(v->fils_gauche == i){
                j = v->fils_droit;
```

```
            }
            else{
                j = v->fils_gauche;
            }

            if(j != NULL){
                hJ = j->hauteur;
            }
            else{
                hJ = -1;
            }

            if(hJ > i->hauteur || (hJ == i->hauteur && delta == 1)){
                v->hauteur = v->hauteur + *delta;
                i = v;
                v = pere(*a, v);
            }
            else{
                delta = 0;
            }

        }

        //Equilibrage des equilibres
        int eqX = x->equilibre;
        int eqY = y->equilibre;

        x->equilibre = eqX - 1 - max(0, eqY);

        if(x->equilibre >= 0){
            y->equilibre = eqY - 1;
        }
        else{
            y->equilibre = eqX -2 + min(0, eqY);
        }

        delta = 0;
        if(y->equilibre >= 0){
            delta = -1;
```

```
252             }
253             else if(eqX <= 0){
254                 delta = 1;
255             }
256 
257             a = reequilibreArbre(a, pereX, delta);
258 
259             return y;
260         }
261 
262         return x;
263     }
264 
```

4. Nous avons utilisé la fonction **rotDroite** pour faire tourner un arbre à droite autour d'un nœud. Nous avons utilisé << delta >> pour représenter le changement de hauteur de cet arbre avant et après rotation.

Dans la fonction, nous terminons d'abord l'opération de rotation, c'est-à-dire que le nœud pivoté devient l'enfant droit de son enfant gauche et le droit de son enfant gauche. Le sous-arbre devient le sous-arbre gauche du nœud tourné. Ensuite, mettez à jour la hauteur de rotation. Enfin, nous devons nous assurer que l'arbre est toujours équilibré après la rotation.

Le même principe s'applique à la fonction **rotGauche.**

```
T_Arbre* rotGaucheDroite(T_Arbre* abr, T_Noeud *x, int *delta){
    if(x->fils_gauche != NULL && x->fils_gauche->fils_droit != NULL){
        x->fils_gauche = rotGauche(abr, x->fils_gauche, *delta);
        x->equilibre = x->equilibre + *delta;
        T_Arbre *x2 = rotDroite(abr, x, *delta);
        return x2;
    }
}

T_Arbre* rotDroiteGauche(T_Arbre* abr, T_Noeud *x, int *delta){
    if(x->fils_droit != NULL && x->fils_droit->fils_gauche != NULL){
        x->fils_droit = rotDroite(abr, x->fils_droit, delta);
        x->equilibre = x->equilibre + delta;
        T_Arbre *x2 = rotGauche(abr, x, delta);
        return x2;
    }
}
```

5. Dans certains cas, la rotation du nœud une fois peut ne pas rééquilibrer l'arbre, il doit donc être pivoté deux fois. Nous avons donc écrit deux fonctions **rotGaucheDroite** et **rotDroiteGauche** afin de résoudre ce problème.

En prenant la fonction **rotGaucheDroite** comme exemple, nous faisons d'abord tourner le nœud enfant gauche << y >> du paramètre passé << x >> vers la gauche, puis tournons le nœud passé << x >> vers la droite pour que l'arbre atteigne l'équilibre.

Le même principe s'applique à la fonction **rotDroiteGauche.**

```
410  T_Arbre* reequilibreArbre(T_Arbre* abr, T_Noeud *x, int *delta){
411
412      if(x != NULL){
413          //on recalcule l'equilibre ici
414          x->equilibre = calculerHauteur(&x->fils_gauche) - calculerHauteur(&x->fils_droit);
415
416          if(x->equilibre == 2){
417              T_Noeud *y = x->fils_gauche;
418
419              y->equilibre = calculerHauteur(&y->fils_gauche) - calculerHauteur(&y->fils_droit);
420
421              if(y->equilibre == 1){
422                  //printf("rotDroite\n");
423                  return rotDroite(abr, x, delta);
424              }
425              else if(y->equilibre == -1){
426                  //printf("rotGaucheDroite\n");
427                  return rotGaucheDroite(abr, x, delta);
428              }
429          }
430          else if(x->equilibre == -2){
431              T_Noeud *y = x->fils_droit;
432
433              y->equilibre = calculerHauteur(&y->fils_gauche) - calculerHauteur(&y->fils_droit);
434
435              if(y->equilibre == -1){
436                  //printf("rotGauche\n");
437                  return rotGauche(abr, x, delta);
438              }
439              else if(y->equilibre == 1){
440                  //printf("rotDroiteGauche\n");
441                  return rotDroiteGauche(abr, x, delta);
442              }
443          }
444      }
445      return NULL;
446  }
```

6. Nous avons utilisé la fonction **reequilibreArbre** pour ramener l'arbre enraciné par le nœud de référence << x >> à nouveau. Tout d'abord, nous calculons d'abord l'équilibre de cet arbre. Il y a quatre cas, respectivement, où l'équilibre du nœud << x >> est 2 et l'équilibre du nœud enfant gauche << y >> du nœud << x >> est 1 et -1, et l'équilibre de << x >> est -2 et Un cas où le nœud enfant droit << y >> du nœud << x >> a un équilibre de 1 et -1.

```
int calculerHauteur(T_Noeud** noeud){
    if(*noeud == NULL){
        return -1;
    }
    else{
        return  1 + max( calculerHauteur(&(*noeud)->fils_gauche), calculerHauteur(&(*noeud)->fils_droit));
    }
}
```

7. Dans **calculerHauteur**, si le noeud passé << noeud >> est NULL, il retourne -1, sinon il retourne la valeur maximale des hauteurs de sous-arbre gauche et droite du noeud passé +1.

```c
T_Noeud* maximum(T_Arbre *abr){

    if(*abr == NULL){
        return NULL;
    }

    T_Noeud *noeud = *abr;

    while(noeud->fils_droit != NULL){
        noeud = noeud->fils_droit;
    }

    return noeud;
}

T_Noeud* minimum(T_Arbre *abr){

    if(*abr == NULL){
        return NULL;
    }

    T_Noeud *noeud = *abr;

    while(noeud->fils_gauche != NULL){
        noeud = noeud->fils_gauche;
    }
    return noeud;
}
```

8. Nous avons utilisé les fonctions **maximale** et **minimale** pour trouver les nœuds les plus grands et les plus petits dans un arbre avec abr comme nœud racine, c'est-à-dire les nœuds les plus à gauche et à droite de l'arbre.

```c
T_Noeud* pere(T_Arbre arb, T_Noeud* noeud){
    if(arb == NULL){
        printf("Erreur l'arbre passe en parametre est NULL.\n");
        return NULL;
    }

    T_Noeud* Pere = NULL;
    Pere = arb;

    while(arb != NULL){
        if((arb->intervalle.min == noeud->intervalle.min && arb->intervalle.max == noeud->intervalle.max) && noeud->id_entr
            return Pere;
        }
        else if(noeud->intervalle.max < arb->intervalle.min){
            Pere=arb;
            arb = arb->fils_gauche;
        }
        else{
            Pere=arb;
            arb = arb->fils_droit;
        }
    }

    return NULL;
}
```

9. Nous avons utilisé la fonction **pere** pour obtenir le nœud parent du nœud souhaité

Complexité:

Puisqu'on équilibre systématiquement les arbres la complexité O(h) est équivalente à $O(\log_2(n))$.

Fonction **verifier_intervalle**:

Cette fonction n'utilise pas de boucles ni d'appels récursifs. Elle sert uniquement à vérifier que le format de la date enregistrée est correct.

Sa complexité est en O(1)

Fonction **conflit_intervalles**:

Cette fonction n'utilise pas de boucles ou d'appels récursifs, elle est uniquement utilisée pour s'assurer qu'il n'y a pas de chevauchement entre les dates.

Sa complexité est en O(1)

Fonction **creer_noeud**:

Dans cette fonction, nous malloc un nouveau nœud et l'initialisons, sans boucles et récursivité.

Sa complexité est en O(1)

Fonction **max** et **min**:

Il n'y a pas de boucle et de récursivité, seulement une comparaison et retourne une valeur.

Sa complexité est en O(1)

Fonction **maximum** et **minimum:**

La complexité de cette fonction dépend de la hauteur de l'arborescence du nœud passé, car nous devons à chaque fois prendre le nœud enfant gauche ou le nœud enfant droit du nœud jusqu'à NULL.

Sa complexité est en $O(\log_2(n))$

Fonction **pere**:

Cette fonction consiste à trouver le nœud parent d'un nœud, de sorte qu'il commence à partir du nœud racine jusqu'en bas, donc une boucle while est utilisée. La complexité dépend de la différence de hauteur entre le nœud et le nœud racine.

Sa complexité est en $O(\log_2(n))$

Fonction **recherche**:

Cette fonction utilise un appel récursif. La terminaison récursive consiste à rechercher le nœud souhaité à partir du nœud racine, qui correspond à la hauteur de la recherche descendante.

Sa complexité est en $O(\log_2(n))$

Fonction **calculerHauteur**:

Cette fonction utilise un appel récursif. La terminaison récursive est que NULL apparaît sur les nœuds gauche et droit d'un nœud, et le nombre d'appels est la hauteur maximale des sous-arbres gauche et droit du nœud.

Sa complexité est en $O(\log_2(n))$

Fonction **rotDroite** et **rotGauche**:

Dans cette fonction, deux fonctions, calculerHauteur et reequilibreArbre, et une boucle while sont utilisées. La relation entre les trois est parallèle et non intégrée, et la complexité des trois

Sa complexité est en $O(\log_2 n)$

Fonction **reequilibreArbre**:

Il y a plusieurs fonctions récursives dans cette fonction, car il n'y a pas d'imbrication, donc la complexité de cette fonction est la complexité maximale de la fonction récursive.

Sa complexité est en $O(\log_2 n)$

Fonction **rotGaucheDroite** et **rotDroiteGauche**:

Cette fonction utilise deux fonctions récursives de rotation, donc la complexité est la même qu'une rotation

Sa complexité est en $O(\log_2 n)$

Fonction **ajouter_noeud:**

Cette fonction utilise une fonction récursive et deux boucles while, qui ont toutes la même complexité.

Sa complexité est en $O(\log_2 n)$

Fonction **Supp_noeud:**

Nous utilisons cette méthode pour rechercher, supprimer, puis équilibrer.

Sa complexité est en $O(\log_2 n)$

Fonction **modif_noeud**:

Nous utilisons cette méthode pour rechercher, supprimer, ajouter et équilibrer.

Sa complexité est en $O(\log_2 n)$

Fonction **afficher_abr, afficher_entr, detruire_arbre**:

Les trois fonctions utilisent une méthode de traversée récursive, chaque nœud n'a besoin de traverser qu'une seule fois.

Sa complexité est en O(n)