Yang XIANG , Kristi TENEQEXHI
5 Décembre 2024
Groupe 2 A

Rapport de NF16

A2024 - TP4 : Arbres binaires

Date : 5 - 12
Benjamin Franklin FB216

Partie 1 La liste des structures

Fonction implémentation	Fonction supplémentaire	Structures

```c
//1.Créer un noeud
T_Noeud *creer_noeud(int id_entr, char *objet, T_inter intervalle);

//2.Ajouter une réservation
void ajouter(T_Arbre *abr, int id_entr, char *objet, T_inter intervalle);

//3.Rechercher une réservation
T_Noeud *rechercher(T_Arbre abr, T_inter intervalle, int id_entr);

//4.Supprimer une réservation
void supprimer(T_Arbre *abr, T_inter intervalle, int id_entr);

//5.Modifier les dates d'une réservation
void modifier(T_Arbre abr, int id_entr, T_inter actuel, T_inter nouveau);

//6.Afficher toutes les réservations présentes dans l'arbre
void afficher_abr(T_Arbre abr);

//7.Afficher les réservations d'une entreprise
void afficher_entr(T_Arbre abr, int id_entr);

//8.Afficher toutes les réservations sur une période
void afficher_periode(T_Arbre abr, T_inter periode);

//9. Détruire tous les nœuds d'un arbre binaire :
void detruire_arbre(T_Arbre *abr);
```

```c
void viderBuffer ();
// fonction qui permet de vider le buffer d'entrée clavier

/******Fonction Supplementaire*****/
int formaterDate(int, int, char);
//Return la date formater MMDD  0 si date non valide

void deFormater_DayMonth(char [3], char [3], int date);
//Recuperer le jour et le mois de format MMDD

char* deFormaterDate_String(int date);
//retourner date deformater dd/mm

void free_noeud(T_Noeud*);

void afficher_noeud(T_Noeud*);
//print les data d'un observation

void printTeteTab();

void readDate(int* debut, int* fin);
//0 for born superiuer, 1 for born inferiuer
```

```c
typedef struct Intervalle{
    int borneInf;
    int borneSup;
} T_inter;

typedef struct Noeud{
    T_inter date;
    int idInter;
    char* descrip;
    struct Noeud* fisGauche;
    struct Noeud* fisDroite;
} T_Noeud;

typedef T_Noeud* T_Arbre;
```

Partie 2 Fonctions supplémentaires

int formaterDate(**int**, **int**, **char**); ~ O(1) – Complexité par rapport à nombre des nœuds.

Pour valider la validité d'une date, la variable char sert de marqueur pour déterminer si l'année est bissextile. La fonction reçoit une date spécifiée et, si cette date respecte les règles de validité, elle retourne la valeur entière correspondant à MMJJ. En revanche, si la date spécifiée est invalide, la fonction retourne 0.

void deFormater_DayMonth(**char** [3], **char** [3], **int** date); ~ O(1) – Complexité par rapport à nombre des nœuds.

Deux pointeurs de type char sont utilisés pour stocker le format humain de la date transmis sous le format MMDD. Ils permettent de convertir une variable entière représentant une date au format MMDD en une chaîne de caractères pour le jour (JJ) et le mois (MM), afin de l'afficher dans un format standardisé.

char* deFormaterDate_String(**int** date); ~ O(1) – Complexité par rapport à nombre des nœuds.

La fonction a une fonctionnalité principale similaire à *deFormater_DayMonth* (elle l'appelle en interne), mais elle renvoie une chaîne de caractères qui concatène directement la date sous forme humaine DD/MM, ce qui facilite la tâche de récupérer les deux chaînes de caractères. La chaîne de caractères est réservée de manière dynamique à l'aide de *malloc*. Ce pointeur vers la chaîne doit être libéré explicitement après l'appel et l'utilisation.

void free_noeud(**T_Noeud***); ~ O(1) – Complexité par rapport à nombre des nœuds.

Libérer l'espace mémoire alloué dynamiquement pour décrire l'objet (pointeur de caractères), ainsi que le nœud correspondant. Cette fonction libère un seul nœud. Ainsi, on peut l'appeler dans une opération de suppression ainsi que plusieurs fois avant de quitter le programme, par `void detruire_arbre(T_Arbre *abr);`

void afficher_noeud(**T_Noeud***); ~ O(1) – Complexité par rapport à nombre des nœuds.

Sa fonction est d'afficher les informations du nœud sous une forme standard, utilisée dans toutes les fonctions d'affichage ainsi qu'en insertion, suppression et modification, afin de fournir des informations sur le nœud sélectionné ou celui sur lequel l'action a été effectuée.

void printTeteTab(); ~ O(1) – Complexité par rapport à nombre des nœuds.

Imprimer l'en-tête pour l'affichage. Ici, des couleurs simples de la console ont été ajoutées (la même stratégie est utilisée pour obtenir du rouge en cas de suppression ou de conflit, et du vert en cas d'ajout).

void readDate(int* debut, int* fin); ~ O(1) – Complexité par rapport à nombre des nœuds.

L'interface utilisateur permet de saisir la date de début et la date de fin. La validation des dates est effectuée en faisant appel à ***int*** *formaterDate();*

void viderBuffer (); ~ O(1) – Complexité par rapport à nombre des nœuds.

Cette fonction vide le tampon d'entrée standard en lisant et en supprimant tous les caractères jusqu'à un saut de ligne ou la fin de fichier.

Partie 3 Analyse de Complexité des fonctions Implémentées

Dans cette partie, les notions (***h***, ***n***) sont utilisées pour caractériser la taille du problème, où n - nombre de nœuds ajoutés dans l'arbre et h - hauteur de l'arbre été construit. Pour une arbre binaire complète, nous avons l'équation suivant :

$$log_2 n \simeq h$$

```
T_Noeud *creer_noeud(int id_entr, char *objet, T_inter intervalle);
```

Complexité de O(1) La constructeur *creer_noeud* est utilisée pour créer dynamiquement un *Noeud* et retourne le pointeur vers ce *Noeud*. La date de *T_inter* est validée par légalité et sera directement utilisée pour construire ce *Noeud*. Les paramètres sont utilisés pour l'affectation des valeurs, ce qui rend la complexité de cette fonction O(1).

```
void ajouter(T_Arbre *abr, int id_entr, char *objet, T_inter intervalle);
```

Complexité de O(h) L'idée de cette procédure est de trouver la bonne place pour ajouter le nœud tout en respectant la règle de ne pas chevaucher une autre réservation. Si l'arbre est vide, on insère le nœud à la tête comme racine. Sinon, on parcourt l'arbre tout en gardant une trace du nœud courant (*select*), de son parent et de son successeur (pour tester plus tard la borne supérieure).
On parcourt l'arbre en comparant la borne inférieure comme demandé dans le TP. Si elle est plus petite, on passe à gauche ; sinon, on compare la borne inférieure avec la borne supérieure du nœud courant pour vérifier un éventuel chevauchement avec le prédécesseur candidat (Chevauchement type 1), ce qui pourrait stopper la procédure. Si elle est plus grande, on passe à droite.
Ensuite, une fois la bonne place trouvée, et si aucun chevauchement avec le prédécesseur n'est détecté, on teste la borne supérieure avec le successeur mis à jour lors du parcours pour s'assurer qu'il n'y a pas de conflit (chevauchement type 2) avant de stocker le nœud. Comme pour toutes les procédures, les opérations de comparaison, d'affectation et d'arithmétique ont un comportement constant. Dans le pire des cas, l'ajout se fait au niveau de la feuille, ce qui représente un chemin plus long (pas forcement unique) (son niveau étant égal à la hauteur de l'arbre + 1).

```
T_Noeud *recherche(T_Arbre abr, T_inter intervalle, int id_entr);
```

Complexité de O(h) La fonction permet de rechercher un nœud en fonction de la date et de l'ID spécifiés par l'utilisateur comme mots-clés. Nous avons choisi (afin d'aider l'utilisateur à ne pas entrer nécessairement l'intervalle exact, mais seulement une date ou un sous-intervalle pour chercher un événement) que si la date spécifiée par l'utilisateur **correspond à un sous-intervalle de l'intervalle** d'un événement défini ($I_{utilisateur} \subseteq I_{evenement}$), le nœud correspondant est testé pour l'ID de l'entreprise (si l'ID ne correspond pas, on suggère l'ID de l'événement trouvé et on renvoie NULL) et ensuite renvoyé. Dans le cas où l'intervalle spécifié par l'utilisateur se chevauche avec deux ou plusieurs intervalles ou si l'ID ne correspond pas, la fonction retourne NULL par défaut. Dans le pire des cas, on trouve un élément au niveau d'une feuille plus loin de la racine, ou on ne trouve rien, mais l'intervalle ne chevauche aucun élément c.-à-d. qu'on doit traverser toute la longueur de l'arbre..

```
void supprimer(T_Arbre *abr, T_inter intervalle, int id_entr);
```

Complexité de O(h) La fonction se divise en deux étapes :
Étape 1 : Déterminer la position du nœud à supprimer.
En comparant la borne inférieure (borneinf), on parcourt l'arbre binaire. Lorsque l'intervalle spécifié par l'utilisateur appartient strictement à celui du nœud courant, on enregistre le père du nœud sélectionné (pereSelect) et, en modifiant un indicateur (trouver), on interrompt le parcours de l'arbre binaire.

Étape 2 : Supprimer le nœud.
Cette opération peut se diviser en trois cas :

Cas 1 : Le nœud à supprimer n'a ni fils gauche ni fils droit.
Cas 2 : Le nœud à supprimer a un seul fils.
Pour ces deux cas, il suffit de connecter directement le père au nœud concerné.
Cas 3 : Le nœud à supprimer possède à la fois un fils gauche et un fils droit.
Dans ce cas, on utilise le successeur de ce nœud comme nouvelle connexion. Ensuite, on appelle récursivement la fonction supprimer(successeur) pour réajuster la structure de l'arbre à partir du successeur, réalisant ainsi le remplacement du nœud supprimé. C'est-à-dire qu'en relançant cette fonction deux fois, chaque appel effectue un parcours de l'arbre avec un comportement presque h (le prédécesseur étant à h dans le pire des cas et *selected* en h-1 [faut avoir deux fils] total ~ 2h ~ h).

```
void modifier(T_Arbre abr, int id_entr, T_inter actuel, T_inter nouveau);
```

Complexité de O(h) En utilisant la fonction rechercher() pour localiser un nœud correspondant, l'intervalle et l'ID de l'entreprise servent de clés de recherche. Une fois le nœud identifié, si l'intervalle modifié par l'utilisateur est inclus dans l'intervalle d'origine, la modification est autorisée immédiatement, garantissant ainsi une bonne place et l'absence de chevauchement. Sinon, si les bornes sont en dehors de l'intervalle d'origine (plus grandes ou complètement différentes), une sauvegarde est effectuée en créant un nouveau nœud copie ou en mettant à jour le nouvel intervalle, puis en appelant la fonction supprimer() avec les données actuelles et la fonction ajouter() pour trouver le bon emplacement. La sommes des O(h) + O(h) = O(h).

Ce n'est pas la solution la plus optimisée, mais c'est la plus lisible et elle permet de recycler en réutilisant les fonctions précédentes. La validation du nouvel intervalle est effectuée afin de ne pas supprimer le nœud si un chevauchement est détecté (l'ajout échouera). Nous validons cela en utilisant une recherche avec l'ID de l'entreprise -1, ce qui permet de contourner le test de l'entreprise, afin de vérifier s'il existe un événement qui chevauche avec notre nouvel intervalle. Cette validation est effectuée dans le main().

```
void afficher_abr(T_Arbre abr);
```

Complexité de O(n) L' algorithme récursif est utilisé pour parcourir cet arbre binaire. La fonction adopte le mode de parcours infixe, garantissant que les intervalles sont affichés dans l'ordre chronologique. La fonction vérifie d'abord si le nœud est valide (pas NULL), puis effectue un appel récursif pour parcourir le sous-arbre gauche de ce nœud, affiche ensuite les informations du nœud lui-même, et enfin effectue un appel récursif pour parcourir le sous-arbre droit. Étant donné que la fonction doit accéder à tous les nœuds de l'arbre binaire, elle génère un appel récursif, ce qui donne une complexité de O(n).

//7.Afficher les réservations d'une entreprise

```
void afficher_entr(T_Arbre abr, int id_entr);
```

Complexité de O(n) En adoptant le même mode de parcours que la fonction *afficher_abr();* , ajoutant une comparaison entre la variable du nœud id_entr et l'id spécifié par l'utilisateur. Cela permet d'afficher les intervalles correspondant à id spécifié par l'utilisateur. Étant donné qu'il est nécessaire de vérifier les valeur clés de tous les nœuds, la complexité de cette fonction est de O(n).

//8.Afficher toutes les réservations sur une période

```
void afficher_periode(T_Arbre abr, T_inter periode);
```

Complexité de O(n) En adoptant le même mode de parcours que la fonction *afficher_abr()*, Lors du parcours, ajouter une comparaison avec l'intervalle spécifié par l'utilisateur. Cette optimisation permet d'améliorer l'algorithme récursif. Dans le pire cas, où l'ensemble de l'arbre est parcouru, la complexité reste de O(n).

Partie 4 Amélioration après la séance et Analyse

- **Changer pour revenir au prototype original (avant l'utilisation de T_Arbre* abr) :**

void modifier(T_Arbre **abr**, int id_entr, T_inter actuel, T_inter nouveau)

abr est un pointeur, c'est-à-dire une adresse, bien qu'il soit en réalité un pointeur vers l'arbre et non un nœud. Pour résoudre le besoin d'avoir un pointeur sur l'arbre lui-même afin de le passer comme argument aux fonctions appelées à l'intérieur de supprimer() et ajouter(), on passe l'adresse de l'arbre (T_Arbre*) vers le paramètre (T_Arbre), puis on effectue une conversion de type (type casting) (T_Arbre* <- T_Arbre). Cela permet de le stocker dans le bon type [T_Noeud** ≡ T_Arbre*].

- **Fixed bug dans modification :**

Nous effectuons une suppression sans vérifier si le nouvel intervalle est valide avant de procéder au réajout, et le résultat est que nous perdons cette réservation au lieu de la modifier (ce qui revient à une simple suppression). Ce que nous voulons faire, c'est ne pas effectuer de changement si le nouvel intervalle n'est pas éligible pour être ajouté. Les trois cas à vérifier sont :

1. La borne inférieure à l'intérieur d'une réservation existante.
2. La borne supérieure à l'intérieur d'une réservation existante.
3. Le nouvel intervalle contient une réservation entière à l'intérieur.

Pour cela, nous avons ajouté dans main() le contrôle de la recherche du nouvel intervalle sans l'ID de l'entreprise (nous avons modifié rechercher() pour ajouter la fonctionnalité de contourner la vérification de l'ID en utilisant -1). Cela permet de vérifier si le nouvel intervalle se chevauche dans l'un des trois cas mentionnés. Si un chevauchement est trouvé, la fonction renvoie une adresse non nulle et affiche celle-ci. La fonction de modification n'est alors pas appelée du tout, elle n'est appelée que si le nouvel intervalle est éligible.