LI Yue

FOREST Thomas

NF16 – TP4 : Les arbres binaires de recherche

Objectif du TP :

Le but de ce TP est de se familiariser avec les arbres binaires de recherche et les différentes opérations nécessaires pour les manipuler. Ici, nous allons utiliser un ABR afin d'indexer les fiches médicales des patients d'un médecin. Chaque patient possèdera une liste de consultation représentées sous la forme d'une liste chaînée.

Structure de données :

Afin de représenter correctement notre ABR ainsi que les différents patients, nous utiliserons une structure ***Consultation*** composée de la date de la consultation, de son motif, de son niveau d'urgence ainsi que d'un pointeur vers la consultation suivante. Nous aurons donc besoin d'une seconde structure, nommée ***Patient*** qui sera composée du nom et prénom du patient, de sa liste de consultation ainsi que du nombre de consultation. Evidemment, pour représenter cela sous la forme d'un ABR, chaque patient possèdera également un pointeur vers son fils gauche ainsi que son fils droit. Pour finir, on possèdera un type ***Parbre*** qui est simplement un pointeur sur un ***Patient*** et nous permettra ainsi de garder l'information de la racine de notre arbre. Ci-contre la représentation finale en **C** .

```c
typedef struct Consultation{
    char* date;
    char* motif;
    int niveauUrg;
    struct Consultation* suivant;
}Consultation;

typedef struct Patient{
    char* nom;
    char* prenom;
    Consultation* ListConsult;
    int nbrconsult;
    struct Patient* fils_gauche;
    struct Patient* fils_droit;
}Patient;

typedef Patient* Parbre;
```

Fonctions & Complexité:

1. *Fonctions ajoutées :*

- **afficherMenu()** : affiche le menu à l'utilisateur.
- **getChoix()** : récupère le choix de l'utilisateur pour le menu.
- **getMotif(char* motif, int max_size)** : récupère le motif d'une consultation
- **getNomPrenom(char* str, int max_size, char* type)** : récupère le nom OU le prénom d'un patient.
- **getDate(char* date, int max_size)** : récupère la date d'une consultation
- **getNivu(int* nivu)** : récupère le niveau d'urgence d'une consultation

Ces fonctions permettent simplement de récupérer les entrées de l'utilisateur. Nous avons décider de créer ces fonctions afin de simplifier la lecture du fichier *main.c*.

Complexité :

- **O(1)** pour les trois premières fonctions.
- **O(N)** pour les suivantes car utilisations de boucles while afin de contrôler l'entrée de l'utilisateur.

- **min_abr(Patient* patient)** : renvoie le minimum d'un ABR en le détachant de sa position (utilisé pour la fonction **supprimer_patient**).

Complexité : **O(H)** (H : hauteur de l'ABR).

- **liberer_patient(Patient* patient)** : permet de libérer l'espace mémoire d'un patient ainsi que de ses consultations.

Complexité : **O(N)** (N : nombres de consultations du patient).

- **Supprimer_arbre (Parbre *abr)** : permet de supprimer totalement un ABR en supprimant tous ses patients (appel récursif de **libérer_patient**).

Complexité : **O(N*M)** (N : nombres de consultations d'un patient, M : nombre de nœuds dans l'ABR). En effet, on doit supprimer M patients qui ont potentiellement chacun N consultations.

2. *Fonctions du sujet :*

- **CrerPatient(char* nm, char* pr)** :

Cette fonction permet de créer une fiche médicale pour un patient. On utilise donc un ***malloc*** pour allouer de la mémoire dynamiquement pour notre nouveau patient. Attention, ici on doit également allouer de la mémoire pour le nom et le prénom du patient.

Complexité : **O(1)**

- **inserer_patient(Parbre* abr, char* nm, char* pr)** :

Cette fonction permet d'insérer un patient dans un ABR. Pour cela, on doit parcourir l'arbre en comparant le nom de chaque nœud avec celui que l'on veut insérer grâce à la fonction ***strcmp***. On va donc à droite quand le nom à insérer est supérieur (dans l'ordre alphabétique), sinon on va à gauche. Quand on est arrivé au bon endroit, on peut insérer le nouveau patient en prenant soin de le relier à son éventuel nouveau père. Pour simplifier notre représentation, deux patients ne peuvent pas avoir le même nom

Complexité : **O(H)** (H : hauteur de l'ABR).

- **rechercher_patient(Parbre* abr, char* nm)** :

Cette fonction permet de rechercher un patient dans un ABR. Pour cela, on parcours l'ABR de la même manière que si on voulait insérer le nom du patient à trouver (voir fonction précédente). Si l'on tombe sur le nom à trouver alors on renvoie le patient correspondant. Sinon, si l'on a finit le parcours sans trouver le nom, on le signale à l'utilisateur et on retourne ***NULL***.

Complexité : **O(H)** (H : hauteur de l'ABR).

- **afficher_fiche(Parbre* abr, char* nm)** :

Cette fonction permet d'afficher la fiche d'un patient dans un ABR. Pour cela on doit commencer par trouver ce patient en utilisant la fonction **rechercher_patient**. Ensuite, on afficher les informations relatives à ce patient et on doit parcourir la liste de ses consultations affin d'en afficher les informations.

Complexité : **O(H*N)** (H : hauteur de l'ABR, N : nombre de consultations du patient).

- **afficher_patients(Parbre *abr)** :

Cette fonction permet d'afficher tous les patients en parcourant l'ABR de manière préfixée. On utilise ici naturellement une fonction récursive. La condition de sortie sera donc de vérifier si l'arbre est vide ou non. Si l'arbre n'est pas vide, on commence par afficher les informations relatives au patient qui se trouve au nœud actuel, puis on appel successivement **afficher_patients** sur les sous arbre gauche puis droit de notre nœud. Ainsi on obtient un parcours préfixé.

Complexité : **O(N)** (N : nombre de nœuds dans l'ABR). Chaque appel est en **O(1)** et on doit parcourir tous les nœuds.

- **CreerConsult(char* date, char* motif, int nivu)** :

Cette fonction permet de créer une consultation. On retrouve ici le même principe que la fonction **CreerPatient**. On doit tout de même penser à allouer de la mémoire pour la date et le motif de la consultation.

Complexité : **O(1)**

- **ajouter_consultation(Parbre* abr, char* nm, char* date, char* motif, int nivu)** :

Cette fonction permet d'ajouter une consultation à un patient identifié par son nom. Pour cela, on doit commencer par trouver le patient correspondant grâce à la fonction **rechercher_patient**. Si l'on trouve un patient pour le nom passé en paramètre on continue et on parcours les consultations déjà existantes (s'il y en a, sinon on insère directement en tête). On s'arrête lorsque l'on est arrivé à la fin de la liste OU que la date de la consultation a insérer est supérieure à une consultation déjà existante. On peut ensuite insérer la nouvelle consultation (en la créant à l'aide de la fonction **CreerConsult**).

Complexité : **O(H*N)** (H : hauteur de l'ABR, N : nombre de consultations du patient).

- **supprimer_patient(Parbre* abr, char* nm)** :

Cette fonction permet de supprimer un patient de l'ABR passé en paramètre. On commence par parcourir l'ABR afin de trouver le patient à supprimer (de la même manière que la fonction **rechercher_patient** mais en gardant l'information sur le père). Si l'on a trouver le patient alors 3 cas se présentent à nous :

1. Le nœud à supprimer ne possède aucun fils : C'est le cas le plus simple, détache le nœud en modifiant son père (s'il existe).
2. Le nœud à supprimer ne possède qu'un seul fils : On détache le nœud en créant un nouveau lien entre son fils et son père (s'il existe).
3. Le nœud à supprimer possède deux fils : On détache le successeur du nœud (grâce à la fonction **min_abr**) puis, on remplace le nœud à supprimer par le successeur en modifiant tous les liens nécessaires (s'ils existent).

Complexité : **O(H)** (H : hauteur de l'ABR).

- **maj(Parbre* abr1, Parbre* abr2)** :

Cette fonction permet de copier et mettre un jour un arbre. Pour cela, on insère tous les nœuds de abr1 dans abr2 récursivement grâce à la fonction **inserer_patient**. Cela peut donc être très couteux dans certains cas.

Complexité : **O(H*N)** (H : hauteur de l'ABR, N : nombre de nœuds dans l'arbre 1).