RAPPORT TP4

LES ARBRES BINAIRES DE RECHERCHE - NF16

ZHAO Ziang / DÉBUREAUX Anaïs - 20 mai 2021

1. FONCTIONS AJOUTÉES

- `T_ABR* initialiserInstance (T_ABR* abr);`

Cette fonction permet d'initialiser un arbre avec les dates et les marques saisies par l'utilisateur.

- `T_ABR* initialiserInstanceFch (T_ABR* abr);`

Cette fonction permet d'initialiser un arbre depuis un fichier saisi par l'utilisateur.

- `int traiterEntree (bool drapo, char dest[100]);`

Cette fonction permet le traitement de la saisie faite par l'utilisateur.

- `void afficherMenu ();`

Cette fonction permet l'affichage du menu intéractif.

- `T_ListeVaccins* creerNoeudVaccin (char* marque, unsigned int nombre_vaccins);`

Cette fonction permet de créer un nouvel élément ABR, à partir d'une date, d'une marque de vaccin et d'un nombre de vaccins, et renvoie un pointeur vers la nouvelle structure.

- `T_ABR* creerNoeudABR (char* date, char* marque, unsigned int nombre_vaccins);`

Cette fonction permet de créer un nouvel élément ABR, à partir d'une date, d'une marque de vaccin et d'un nombre de vaccins, et renvoie un pointeur vers la nouvelle structure.

- `void libererRessrc (T_ABR* abr);`

Cette fonction permet de libérer les ressources d'un arbre.

- `void libererVaccins (T_ListeVaccins* listeVaccins);`

Cette fonction permet de libérer les ressources d'une liste de vaccins.

- `void afficherDates (T_ABR* abr);`

Cette fonction affiche toutes les dates de l'arbre dans l'ordre croissant avec un parcours infixe.

- `void afficherDateMarques (T_ABR* abr, char* date);`

Cette fonction affiche toutes les marques d'une date donnée.

- `bool verifierDate (char* date);`

Cette fonction permet de vérifier le format de la date entrée.

- `void enregistrerMarques (T_ABR* abr, char*** mks, int* nb_mks);`

Cette fonction permet d'enregistrer toutes les marques d'un arbre dans un tableau de pointeurs de marque.

- `bool verifierMarque (char* marque, char** mks, int nb_mks);`

Cette fonction permet de contrôler l'existence de la marque entrée par l'utilisateur. Elle retourner true si la marque existe.

- `void desallouerDate (T_ABR** abr);`

Cette fonction permet de libérer les ressources d'un nœud de l'arbre, dont la date.

- `void desallouerVaccin (T_ListeVaccins** vaccin);`

Cette fonction permet de libérer les ressources d'un vaccin de la liste, dont la marque.

- `T_ABR* smallerDroite (T_ABR* abr);`

Cette fonction trouve le plus petit nœud dans le sous-arbre droit d'un nœud donné.

- `void reformerArbre (T_ABR** abr, char date[]);`

Cette fonction permet de reformer l'arbre binaire après la suppression d'un nœud (une date qui n'a pas de vaccins). Elle permet de respecter les contraintes de l'arbre binaire de recherche : tous les nœuds à gauche d'un nœud père auront tous des clés inférieures à ce dernier, et tous les nœuds à droite auront une clé supérieure.

2. Complexité

Indications :
n : nombre de vaccins
m : nombre de nœuds
h : hauteur de l'arbre
k : nombre de marques enregitrées
C : le nombre de nœuds différents dans lesquels il faut déduire des vaccins

i. Fonctions à implémenter

- `void ajouterVaccinL(T_ListeVaccins** listeVaccins, char* marque, int nb_vaccins);`

Cette fonction fait appel à la fonction `creerNoeudVaccin` qui est de complexité O(1). De plus elle fait appel récursivement à la fonction `ajouterVaccinL`. Soit T(n) le nombre d'appels récursifs. Dans le pire des cas, la marque n'existe pas encore et on parcourt toute la liste de n vaccins donc T(n) = n.
Complexité globale : **O(n)**.

- `void ajouterVaccinA(T_ABR** abr, char*date, char* marque, int nb_vaccins);`

Cette fonction fait appel à la fonction `creerNoeudABR` qui est de complexité O(1). De plus elle fait appel récursivement à la fonction ajouterVaccinA. Soit T(n) le nombre d'appels récursifs. Pour un arbre de hauteur h, dans le pire des cas, après h appels, l'arbre passé en paramètre est vide et l'algorithme se termine : T(n)=h. La fonction fait appel à `ajouterVaccinL` en O(n) pour le nœud recherché (la date demandée).
Complexité globale : **O(h+n)**.

- `void afficherStockL(T_ListeVaccins* listeVaccins);`

Cette fonction fait appel récursivement à la fonction `afficherStockL`. Soit T(n) le nombre d'appels récursifs. Dans tous les cas, on parcourt toute la liste de n vaccins donc T(n) = n.
Complexité globale : **O(n)**.

- `void afficherStockA(T_ABR* abr);`

Cette fonction fait appel à la fonction `afficherStockL` qui est de complexité O(n). De plus elle fait deux appels récursifs à la fonction `afficherStockA`. Soit T(m) le nombre d'appels récursifs. Pour un arbre de m noeuds, dans tous les cas, il faut m appels pour parcourir chaque nœud de l'arbre avant que l'arbre passé en paramètre soit vide et que l'algorithme se termine : T(m)=m.
Complexité globale : **O(n*m)**.

- `int compterVaccins(T_ABR* abr, char* marque);`

Cette fonction fait deux appels récursifs à la fonction `compterVaccins`. Soit T(m) le nombre d'appels récursifs. Pour un arbre de m noeuds, dans tous les cas, il faut m appels pour parcourir chaque nœud de l'arbre avant que l'arbre passé en paramètre soit vide et que l'algorithme se termine : T(m)=m. De plus une boucle itère pour chaque nœud sur les éléments de la liste de vaccins. Soit n le nombre de vaccins par liste. Pour une liste de n vaccins, il faut itérer n fois pour parcourir la liste. Les opérations de la boucle sont simples : O(n).
Complexité globale : **O(n*m)**.

- `void deduireVaccinL(T_ListeVaccins** listeVaccins, char* marque, int nb_vaccins);`

Cette fonction fait appel récursivement à la fonction `deduireVaccinL`. Soit T(n) le nombre d'appels récursifs. Dans le pire des cas, la marque se trouve en dernière position et on parcourt toute la liste de n vaccins donc T(n) = n. De plus, cette fonction fait appel à la fonction `desallouerVaccin` qui est de complexité O(1).
Complexité globale : **O(n)**.

- `void deduireVaccinA(T_ABR** abr, char* marque, int nb_vaccins);`

Soit C le nombre de nœuds différents dans lesquels il faut déduire des vaccins pour déduire `nb_vaccins`. On appelle C fois la fonction `deduireVaccinA` et 3C fois la fonction `compterVaccins` de O(n*m) dans le `main.c`. Au pire des cas, C = m si on déduit des vaccins dans chaque nœud : O(m + n*m) ~ O(m).

Cette fonction fait appel récursivement à la fonction `deduireVaccinA`. Soit T(m) le nombre d'appels récursifs. Pour un arbre de m noeuds, dans le pire des cas, on parcourt tous les noeuds donc T(m) = m.

Cette fonction fait appel 3 fois à `compterVaccins` : O(3*n*m) ~ O(n*m).

Cette fonction fait appel à `reformerArbre` de O(h) et `deduireVaccinL` de O(n) pour le nœud où il faut déduire des vaccins.

Complexité globale : O(m*(m*(n*m + n) + h))) ~ **$O(m^3*n)$**

ii. Fonctions ajoutées

- `T_ABR* initialiserInstance (T_ABR* abr);`

Cette fonction itère sur le nombre de dates O(m), le nombre de marque par date O(n) et le nombre de vaccin par marque à ajouter à une date de l'arbre O(n+h).
Complexité globale : **O(m*n*(n+h))**.

- `T_ABR* initialiserInstanceFch (T_ABR* abr);`

Cette fonction est de même complexité que initialiserInstance.
Complexité globale : **O(m*n*(n+h))**.

- `int traiterEntree (bool drapo, char dest[100]);`

Cette fonction itère sur la chaîne de caractère (<=100) entrée par l'utilsateur. Il n'y a que des instructions simples. O(100)~O(1)
Complexité globale : **O(1)**.

- `void afficherMenu ();`

Il n'y a que des instructions simples, pas de boucle ni d'appel récursif.
Complexité globale : **O(1)**.

- `T_ListeVaccins* creerNoeudVaccin (char* marque, unsigned int nombre_vaccins);`

Il n'y a que des instructions simples, pas de boucle ni d'appel récursif.
Complexité globale : **O(1)**.

- `T_ABR* creerNoeudABR (char* date, char* marque, unsigned int nombre_vaccins);`

Cette fonction fait appel à la fonction `ajouterVaccinL` qui est de complexité O(n).
Complexité globale : **O(n)**.

- `void libererRessrc (T_ABR* abr);`

Cette fonction fait appel à la fonction `libererVaccins` qui est de complexité O(n). De plus elle fait deux appels récursifs à la fonction `libererRessrc`. Soit T(m) le nombre d'appels récursifs. Pour un arbre de m noeuds, dans tous les cas, il faut m appels pour parcourir chaque nœud de l'arbre avant que l'arbre passé en paramètre soit vide et que l'algorithme se termine : T(m)=m.
Complexité globale : **O(n*m)**.

- `void libererVaccins (T_ListeVaccins* listeVaccins);`

Cette fonction fait appel récursivement à la fonction `libererVaccins`. Soit T(n) le nombre d'appels récursifs. Dans tous les cas, on parcourt toute la liste de n vaccins donc T(n) = n.
Complexité globale : **O(n)**.

- `void afficherDates (T_ABR* abr);`

Cette fonction fait deux appels récursifs à la fonction `afficherDates`. Soit T(m) le nombre d'appels récursifs. Pour un arbre de m noeuds, dans tous les cas, il faut m appels pour parcourir chaque nœud de l'arbre avant que l'arbre passé en paramètre soit vide et que l'algorithme se termine : T(m)=m.
Complexité globale : **O(m)**.

- `void afficherDateMarques (T_ABR* abr, char* date);`

Cette fonction fait deux appels récursifs à la fonction `afficherDateMarques`. Soit T(m) le nombre d'appels récursifs. Pour un arbre de m noeuds, au pire des cas, il faut m appels pour parcourir chaque nœud de l'arbre avant que l'arbre passé en paramètre soit vide ou corresponde à la date recherchée et que l'algorithme se termine : T(m)=m. De plus une boucle itère pour le nœud recherché sur les éléments de la liste de vaccins. Soit n le nombre de vaccins par liste. Pour une liste de n vaccins, il faut itérer n fois pour parcourir la liste. Les opérations de la boucle sont simples : O(n).
Complexité globale : **O(n+m)**.

- `bool verifierDate (char* date);`

Cette fonction fait des opérations simples pour tester la validité de la date. Une boucle itère 10 fois sur les caractères de date. Les opérations sont simples. O(10)~O(1)
Complexité globale : **O(1)**.

- `void enregistrerMarques (T_ABR* abr, char*** mks, int* nb_mks);`

Soit k le nombre de marques enregistrées. Cette fonction fait itère sur le nombre de marques enregitrées : O(k) et cela pour chaque vaccin de la liste d'un nœud : O(n*k). De plus elle fait deux appels récursifs à la fonction `enregistrerMarques`. Soit T(m) le nombre d'appels récursifs. Pour un arbre de m noeuds, dans tous les cas, il faut m appels pour parcourir chaque nœud de l'arbre avant que l'arbre passé en paramètre soit vide et que l'algorithme se termine : T(m)=m.
Complexité globale : **O(k*n*m)**.

- `bool verifierMarque (char* marque, char** mks, int nb_mks);`

Cete fonction itère sur le nombre de marques existantes. Soit n le nombre de marques existantes.
Complexité globale : **O(n)**.

- `void desallouerDate (T_ABR** abr);`

Il n'y a que des instructions simples, pas de boucle ni d'appel récursif.
Complexité globale : **O(1)**.

- `void desallouerVaccin (T_ListeVaccins** vaccin);`

Il n'y a que des instructions simples, pas de boucle ni d'appel récursif.
Complexité globale : **O(1)**.

- `T_ABR* smallerDroite (T_ABR* abr);`

Cette fonction fait appel récursivement à la fonction `smallerDroite`. Soit T(n) le nombre d'appels récursifs. Pour un arbre de hauteur h, dans le pire des cas, après h appels, l'arbre passé en paramètre est vide et l'algorithme se termine : T(n)=h.
Complexité globale : **O(h)**.

- `void reformerArbre (T_ABR** abr, char date[]);`

Cette fonction fait appel à la fonction `desallouerDate` qui est de complexité O(1). De plus elle fait appel récursivement à la fonction `reformerArbre`. Soit T(n) le nombre d'appels récursifs. Pour un arbre de hauteur h, dans le pire des cas, après h appels, l'arbre passé en paramètre est vide et l'algorithme se termine : T(n)=h.
Complexité globale : **O(h)**.