Rapport TP4 NF16

Introduction

Dans le cadre de NF16, le TP4 porte sur la manipulation et la gestion des structures de données, en particulier les arbres binaires de recherche. Ce rapport présente les différentes structures utilisées, les fonctions implémentées pour gérer ces structures, et les choix de conception réalisés pour optimiser l'efficacité et la performance de ces fonctions. L'objectif principal est de fournir une compréhension approfondie des arbres binaires de recherche, de leur fonctionnement et de leur manipulation en utilisant le langage C. Nous avons fait le choix de représenter graphiquement les structures de données avec GTK3.

Structures de Données

- ***Structure T_Sommet :***

  Cette structure permet de créer un arbre binaire de recherche où chaque nœud peut représenter un intervalle d'éléments.

- ***Structure T_Sommet2 :***

C'est une variante de ***T_Sommet*** conçue pour simuler un sommet classique dans un arbre binaire de recherche..

Sur VSCode, sizeof T_Sommet est de 16 octets et T_Sommet2 de 12 octets, tandis que sur Code::Blocks, les deux structures font 24 octets, en raison de politiques d'alignement différentes appliquées par les compilateurs. L'alignement mémoire est une optimisation qui permet d'accéder plus efficacement aux données en alignant les adresses mémoire sur des limites spécifiques, généralement des multiples de la taille des types de données. Mais pour obtenir les tailles exactes sans alignement du compilateur, on a utilisé #pragma pack(1) désactivant le remplissage automatique, donnant 24 octets pour T_Sommet et 20 octets pour T_Sommet2. Cela est nécessaire pour le bon calcul de tailleMemoire.

```
// Désactivation du rem
#pragma pack(1)
struct T_Sommet {
    int borneInf;
    int borneSup;
    T_Arbre filsGauche;
    T_Arbre filsDroit;
};

// Désactivation du rem
struct T_Sommet2 {
    int element;
    T_Arbre filsGauche;
    T_Arbre filsDroit;
};
#pragma pack()
```

Structure	VSCode (octets)	Code::Blocks (octets)	Sans alignement (#pragma pack(1)) (octets)
T_Sommet	16	24	24
T_Sommet2	12	24	20

Choix de Programmation

Fonctions de base :

- ***T_Sommet *creerSommet(int element) :***

La fonction alloue de la mémoire pour un nouveau sommet, initialise ses attributs borneInf` et borneSup avec la valeur element, et ses pointeurs filsGauche et filsDroit à NULL, puis retourne le pointeur vers ce sommet nouvellement créé. Elle est donc de complexité O(1).

- ***T_Arbre insererElement(T_Arbre abr, int element) :***

La fonction insère un nouvel élément dans un arbre binaire de recherche. Si l'arbre est vide, elle crée un nouveau sommet. Si l'élément est proche des bornes du sommet actuel, elle met à jour ces bornes; sinon, elle insère l'élément dans le sous-arbre gauche ou droit selon sa valeur, puis fusionne les sommets si nécessaire, avant de retourner l'arbre modifié. Elle est dans le pire des cas de complexité O(n).

- ***T_Sommet *rechercherElement(T_Arbre abr, int element) :***

La fonction cherche un élément dans un arbre binaire de recherche. Si l'arbre est vide, elle retourne NULL. Si l'élément se trouve dans les bornes du sommet actuel, elle retourne ce sommet; sinon, elle continue la recherche récursivement dans le sous-arbre gauche ou droit selon la valeur de l'élément. Elle est dans le pire des cas de complexité O(n).

- ***void afficherSommets(T_Arbre abr) :***

La fonction parcourt et affiche les intervalles des sommets d'un arbre binaire de recherche en ordre croissant. Si l'arbre est vide, elle retourne immédiatement. Elle affiche les sommets en visitant récursivement le sous-arbre gauche, affichant le sommet actuel, puis le sous-arbre droit. Elle est dans le pire des cas de complexité O(n) (chaque nœud est visité une fois).

- ***void afficherElements(T_Arbre abr) :***

La fonction parcourt l'arbre binaire abr de manière récursive. À chaque nœud visité, elle affiche l'élément stocké dans ce nœud, puis répète le processus pour son sous-arbre gauche puis pour son sous-arbre droit. Cela garantit que tous les éléments de l'arbre sont affichés dans l'ordre. La complexité de la fonction est linéaire par rapport au nombre d'éléments de l'arbre. Ainsi, sa complexité est O(n).

- ***T_Arbre supprimerElement(T_Arbre abr, int element) :***

La fonction recherche l'élément à supprimer dans l'arbre, puis effectue des opérations de réorganisation des sous-arbres en fonction de la position de l'élément trouvé. Dans le pire des cas, elle peut parcourir tous les éléments de l'arbre pour trouver l'élément à supprimer. Ensuite, chaque opération d'insertion peut nécessiter jusqu'à n opérations dans le pire des cas. Ainsi, dans le pire des cas, la complexité peut être de $O(n^2)$.

- ***void tailleMemoire(T_Arbre abr) :***

La fonction appelle simplement les deux fonctions ***tailleMemoireIntervalles*** et ***tailleMemoireClassique***, donc sa complexité est celle de la fonction la plus complexe parmi les deux, soit $O(n^2)$ dans le pire des cas.

Fonctions supplémentaires :

- ***T_Arbre fusionnerSommets(T_Arbre abr) :***

La fonction a une complexité de $O(n^2)$ dans le pire des cas. Elle traverse chaque sommet de l'arbre une fois, puis appelle ***ItererFusion*** pour chaque sommet et tente de fusionner le sommet actuel avec tous les autres sommets de l'arbre, potentiellement vérifiant jusqu'à O(n) sommets pour chaque sommet, ce qui entraîne une complexité quadratique au total.

- ***unsigned int tailleMemoireIntervalles(T_Arbre abr) :***

La fonction parcourt tous les nœuds de l'arbre une seule fois de manière récursive, ce qui donne une complexité en temps linéaire. Ainsi, sa complexité est O(n).

- ***unsigned int tailleMemoireClassique(T_Arbre abr) :***

La fonction parcourt également tous les nœuds de l'arbre une seule fois de manière récursive, mais elle effectue également une opération proportionnelle au nombre d'éléments dans chaque intervalle. Dans le pire des cas, chaque nœud peut avoir jusqu'à n éléments, ce qui entraîne une complexité quadratique. Par conséquent, dans le pire des cas, sa complexité est $O(n^2)$.

- ***T_Sommet* rechercherPere(T_Arbre abr, int element) :***

La fonction identifie le nœud parent d'un élément dans l'arbre binaire abr. Elle commence par vérifier si l'arbre est vide ou si le nœud actuel n'a pas de fils gauche ou de fils droit, auquel cas elle retourne NULL. Ensuite, elle examine si l'élément se trouve dans les intervalles des fils gauche ou droit du nœud actuel. Si tel est le cas, elle renvoie le nœud actuel en tant que parent. Sinon, elle effectue une recherche récursive dans le sous-arbre gauche ou droit en fonction de la valeur de l'élément par rapport aux bornes du nœud actuel. La complexité est de O(n) dans le pire des cas.

- ***int niveauDuSommet(T_Arbre racine, T_Sommet *sommet) :***

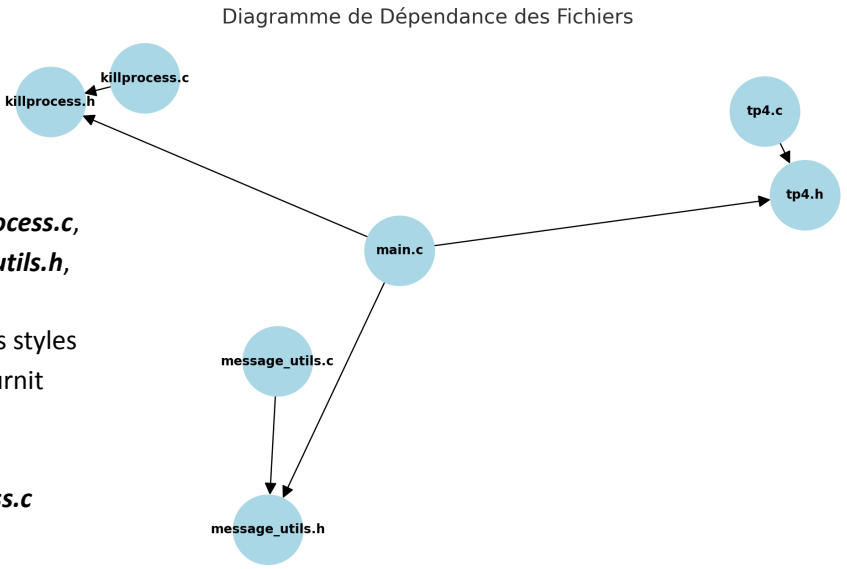
La fonction détermine le niveau d'un sommet donné dans un arbre binaire en parcourant récursivement les ancêtres du sommet jusqu'à la racine. Si le sommet est nul, la fonction retourne -1 pour indiquer que le niveau n'a pas été trouvé. Sinon, elle initialise un compteur de niveau à 0 et recherche récursivement le père du sommet donné jusqu'à ce qu'elle atteigne la racine, en incrémentant le niveau à chaque étape. La complexité de cette fonction est proportionnelle à la hauteur de l'arbre, ce qui est O(n) dans le pire des cas.

- ***int rechercherHauteur(T_Arbre abr) :***

La fonction calcule la hauteur de l'arbre binaire en effectuant un parcours récursif des sous-arbres gauche et droit à partir de chaque nœud. Si l'arbre est vide, sa hauteur est considérée comme -1. Sinon, elle détermine la hauteur maximale entre les sous-arbres gauche et droit, puis ajoute 1

pour inclure le nœud actuel. La complexité de cette fonction est linéaire par rapport au nombre de nœuds dans l'arbre, car chaque nœud est visité une seule fois, ce qui donne une complexité de O(n).

Organisation du projet



L'organisation du projet comprend les fichiers sources ***tp4.c***, ***main.c***, ***message_utils.c***, et ***killprocess.c***, ainsi que les fichiers d'en-tête ***tp4.h***, ***message_utils.h***, et ***killprocess.h***.
De plus, il y a le fichier ***gtkgui.css*** qui contient les styles pour l'interface graphique, et ***readme.txt*** qui fournit des instructions d'utilisation.
Le fichier ***message_utils.c*** gère les messages affichés pour une meilleure lisibilité et ***killprocess.c*** assure la terminaison des processus en cours.

En conclusion,

Le projet a utilisé un ensemble de fonctionnalités robustes pour travailler avec un arbre binaire de recherche, tout en veillant à ce que l'insertion, la recherche, la suppression et l'affichage des éléments respectent une certaine complexité. Des fonctionnalités telles que les décisions de conception et de mise en œuvre sur l'alignement mémoire étaient d'une importance particulière pour cela. D'autres fonctionnalités ajoutent beaucoup plus de souplesse à l'ensemble, créant davantage de potentiel pour interagir avec l'arbre.
En somme, ce projet a abouti à une application solide et fonctionnelle pour la manipulation et la gestion des arbres binaires de recherche. Les choix de conception, les fonctionnalités ajoutées et les performances optimisées en font un exemple notable de développement logiciel dans le cadre de la programmation en C et de la manipulation des structures de données.

Sources/Outils:
https://docs.gtk.org/gtk3/
https://learn.microsoft.com/fr-fr/windows/win32/api/winuser/nf-winuser-showwindow
https://developer.apple.com/documentation/appkit/nswindow
https://www.manpagez.com/html/gtk3/gtk3-3.24.31/GtkStyleContext.php
Wikipedia, Cours de NF16, ChatGPT, stackoverflow, geeksforgeeks, forums.codeblocks.org
VSCode, CodeBlocks, Github, InnoSetup, Process Monitor