Université de Sciences & Technologies Houari Boumediene

Faculté d'informatique

Rapport de Mini Projet

TP Algorithmique & Structures de données – Programmation C

Manipulation de Structures dynamiques en C

Réalisé par : Sadi Mohamed Mouaad
Mahdjoub Abderraouf

Responsable du module : Mme Boukhedouma
Enseignant de TP : Mr Toufik Souadia

Année : 2022/2023

1. Introduction

Ce mini-projet a pour but de manipuler les différentes structures de données dynamiques en langage C, nous allons explorer les arbres binaires, et implémenter les différents algorithmes d'opérations de manipulations sur les arbres binaires non ordonnées, et ordonnées (arbre binaire de recherche), mais aussi les piles et les files, nous regrouperons toutes les fonctions du projet sous un menu à choix multiples pour permettre à l'utilisateur d'interagir avec notre programme.

Dans le cadre de notre deuxième année licence en ACAD à l'USTHB, il nous a été demandé de faire un Mini-projet, nous permettant de mettre en pratique nos connaissances en algorithmiques et programmation en langage C sur les ures de données dynamiques (Arbres binaires, Piles & Files).

2. Objectifs du TP :

♣ Manipulation de structures dynamiques en C (arbres binaires, piles et files)
♣ Implémentation d'algorithmes de parcours d'arbre binaire et des opérations de manipulation d'arbres binaires, de piles et de files
♣ Ecriture et appel de fonctions
♣ Utilisation d'un menu à choix multiples

3. Conception :

La réalisation du projet est divisée en deux parties, la première partie est d'écrire les algorithmes des fonctions et procédures relatifs à ce qu'il nous a été demandé de faire.

La deuxième partie serait de finaliser l'algorithme en incluant les fonctions et procédures écrites précédemment, puis d'utiliser un menu pour utiliser les différentes fonctionnalités incluses dans le programme.

On va tout d'abord introduire le menu, il aura la forme suivante :

Menu

1) Partie I
2) Partie II
3) Partie II
0) Quitter

De cette façon, l'utilisateur ait à choisir la partie qui l'intéresse, il faut savoir d'abord que les parties sont liées entre eux, c à d dépendants entre eux, La partie II dépend de la partie I, et la partie III dépend aussi de la partie II, il serait donc impossible à l'utilisateur de choisir la partie II sans avoir traité la partie I, et de traiter la partie III sans avoir traité la partie II.

Lorsque l'utilisateur choisi la partie I, On affichera un autre menu qui a la forme suivante :

PARTIE I

1 : Construire l'arbre avec des données fixe par le programmeur (exemple)
2 : Construire l'arbre manuellement
3 : Afficher en largeur du contenu de l'arbre
4 : Afficher la structure de l'arbre
5 : Recherche d'un mot dans l'arbre
6 : Ajout d'un numéro de page pour un mot donnée dans l'arbre
7 : Suppression d'un numéro de page pour un mot donnée dans l'arbre
0 : Quitter

1. Pour la première option, Il s'agit de la fonction Const_Tree qui prend en paramètre x qui va servir comme variable pour indiquer l'option à faire, sois construire l'arbre manuellement, ou avec des données existantes dans le programme, dans ce cas, ça sera x = 0, elle prend aussi l'arbre qui va être construit, on cite d'abord l'enregistrement de l'arbre & de la liste, pour la construction de l'arbre, on a utilisé un enregistrement données pour la facilité de manipulation et d'écriture de code.
   Voici les enregistrements en langage algorithmique :

```
type donnees enregistrement
   p : ^entier ;
   nmbpage : entier ;
   mot[30] : chaine ;
finenreg ;

type  elt enregistrement
   entier numpage;
    elt ^suivant;
finenreg;
type liste ^elt ;
declaration de l'arbre
type  index enregistrement
   liste page;
   char mot[30];
   succg, succd :^index ;
finenreg;
type index^ livre;
declaration de l'enregistrement donnees
type  donnees enregistrement
   p : ^entier ;
   nmbpage : entier;
   mot[30] : chaîne ;
finenreg;
```

L'enregistrement contient le mot, le nombre de numéro de page, et un tableau dynamique d'entiers, de cette façon, la fonction Const_Tree va générer un nombre de numéro de page aléatoire pour chaque mot, l'utilisateur va d'abord introduire le nombre de pages qu'il y a dans le livre, ensuite,

grâce à la bibliothèque #include <time.h>, et la fonction rand(), on génère le nmbpage pour chaque mot par cette formule : donnees .nmbpage = rand()%taille, avec taille qui représente le nombre

Bien sûr, pour que les valeurs soient réellement aléatoire, on engraine (seed) le générateur avec le temps avec srand(time(0)), de cette façon, rand() va générer des valeurs aléatoire à chaque fois.

Dans la fonction Const_Tree, on créée un tableau de l'enregistrement données de taille 10, puis, on génère les valeurs aléatoire de pages, ensuite on les trie, après on fait appel à la fonction ajouter_noeud dans une boucle qui tourne 10 fois et qui va ajouter un nœud contenant le mot et la liste de page ou il apparaît à chaque fois.

2. On fait la même chose pour la saisie manuelle sauf qu'on va prendre x = 1, et on demande à l'utilisateur à chaque fois d'entrer le mot et les pages où il apparaît
   En utilisant une variable k qu'on multiplie par -1 à chaque fois pour alterner l'insertion du nœud afin qu'il soit symétrique, parfois on ajoute dans le fils gauche, et parfois dans le fils droit^

3. Pour afficher l'arbre en largeur, on a utilisé une file qui va enfiler la racine, ensuite utiliser une file pour le défiler et l'afficher au même temps, on enfile le fils gauche ou le fils droit s'ils existent (unpacking), voici l'enregistrement de la file modifié et adapté selon nos besoins écrit en algorithmiques :

```
type eltfile enregistrement
  livre info;
  ^suivant : eltfile ;
finenreg
type file enregistrement
 tete, queue :^ eltfile ;
finenreg
```

   Bien sûr, on a écrit les fonctions primitives de manipulation de cette file, avec quelques modifications pour qu'elle soit compatible avec cet enregistrement

4. Pour afficher la structure de l'arbre, c'est le même principe que celui de la question 3, sauf qu'on a précisé le fils gauche et le fils droit pour chaque nœud.

Pour 5. 6. 7. On a utilisé une seule fonction qui va prendre en paramètre l'arbre et x pour indiquer quoi faire, 0 pour rechercher, 1 pour insérer et 2 pour supprimer une page, bien évidemment, on fait entrer en paramètre la page à insérer/supprimer/rechercher, on compare le mot qui nous intéresse avec toutes les nœuds de l'arbre en utilisant la fonction strcmp() de la bibliothèque #include <string.h> qui retourne 0 dans le cas où ils sont égaux.

Ensuite, on réalise les opérations nécessaires relatives à l'option x.

0. Ce choix permet à l'utilisateur de revenir au menu principal.

Pour la partie II voici la forme de son menu :

PARTIE II

1 : A partir d'un arbre non ordonne déjà construit, construire un arbre ordonne et équilibré
2 : Afficher le contenu de l'index en respectant l'ordre croissant des mots et des pages pour chaque mot
3 : Recherche un mot donne dans l'arbre et retourner son niveau (si il existe)
4 : Ajouter un mot donne dans l'arbre avec un numéro de page. L'arbre reste ordonnée
5 : Supprimer un mot de l'index avec tous les numéros de pages associées au mot.
0 : Quitter

1. Pour le premier choix, on va construire un arbre équilibré et ordonnée à partir de l'arbre construit auparavant dans la partie 1 (ce qui justifie la dépendance entre la partie 1 et 2), pour cela, on a utilisé un tableau, auquel on l'a rempli avec les informations de l'arbre précédent en utilisant la fonction RemplirInfo().
   Pour ensuite, trier le tableau en utilisant la fonction tri_tableau(), et par la fin, créer un arbre équilibré et naturellement ordonné en utilisant une fonction récursive Creer_Arbre_Equilibre qui prends le millieu du tableau comme racine, et calculer à chaque fois un nouveau milieu pour chaque sous tableau afin de lier les fils gauches & droits de l'arbre.

2. Pour ce choix, on a utilisé le parcours infixé puisqu'on manipule un arbre ordonnée, donc naturellement, l'affichage du contenu de l'index respectera l'ordre croissant des mots, quant aux pages, ils sont de base trié de façon croissante.

3. Pour la recherche d'un mot dans l'arbre, on a quasiment utilisé la même fonction que celle de la partie I, pour éviter les fonctions redondantes et optimiser donc le programme.

4. Pour l'ajout d'un mot dans l'arbre, on a créé une fonction récursive ajouterMot() qui va comparer le mot en utilisant toujours la fonction strcmp(), et parcourir l'arbre selon le résultat de strcmp(), c à d, parcourir le fils gauche de l'arbre, ou le fils droit.

5. On vérifie d'abord en utilisant la fonction recherche_insere_supprime_page() de la partie 1 l'existence du mot, ensuite, dans le cas où il existe, on utilise la fonction récursive SupprimerMot(), toujours en utilisant la fonction strcmp().

   Cette fonction va parcourir l'arbre en comparant à chaque fois le mot du nœud avec le mot à supprimer, après, une fois le nœud trouvé, si le fils gauche du nœud à supprimer est nul, on fait le chaînage avec le fils droit, et on libère le nœud.
   Sinon si le fils droit est nul, on fait la même chose sauf avec le fils gauche, sinon les deux nœuds existent, on va dans le fils droit, ensuite on prend le nœud le plus à gauche, et on le chaîne à la place du nœud supprimé en utilisant la fonction supprimerNoeud() qui fonctionne avec le même principe.

Pour la dernière partie voici la forme de son menu :

Partie III
1 : Construire les différentes listes contenant les chemins de l'arbre. 2 : Afficher tous les chemins en commençant par le plus à gauche.

1. Pour le premier choix, on a créé une fonction itérative branches() en utilisant une pile, c'est à peu près le même principe que celui du parcours en largeur, sauf que cette fois si, c'est un parcours en profondeur, on empile tout d'abord la racine, puis on entre dans une boucle, à chaque fois on empile le fils de plus gauche, sinon on empile le fils droit.
   Lorsqu'on atteint une feuille, on sauvegarde le chemin dans une file, puis on insère cette file dans une liste qui contient la file, ensuite, on dépile en sauvegardant dans un arbre intermédiaire, puis en cherche le fils le plus à droite qui n'a pas été dans un chemin parcouru, on saura grâce à l'arbre intermédiaire, lorsqu'on trouve un fils droit, on l'empile puis on refait le même processus jusqu'à ce que la pile devient vide.
   Il faut noter qu'à chaque fois qu'on empile, on doit enfiler aussi, et vice versa.

   Bien sûr, on a donc créé de nouveaux enregistrements pour effectuer cette opération et des primitives relatifs aux nouvelles structures de données, voici les enregistrements :

```
type struct eltpile
  livre info;
  struct eltpile ^suivant;
 finenreg ;
type eltpile ^pile;

 declaration d'une file qui contient des mots (utilisé pour les chemins)
type eltfilechemin enregistrement
  char mot[30];
  struct eltfilechemin ^suivant;
 eltfilechemin;
type filechemin enregistrement
  eltfilechemin ^tete, ^queue;
 finenreg ;

 declaration d'une liste qui contient les chemins d'un arbre (les files)
type eltlistechemin enregistrement
  filechemin info;
  eltlistechemin ^suivant;
finenreg ;
type eltlistechemin ^listechemin;
```

Pour le menu, on a utilisé une bibliothèque pour rendre le programme agréable à voir pour l'utilisateur, cette bibliothèque est la suivante : #include <unistd.h> de cette façon, le menu s'afficheras de façon dynamique et plaisante à l'œil.

5. Conclusion :

On considère ce mini-projet de manipulation de structures dynamiques en C nous a permis de mettre en pratique nos connaissances en implémentant des algorithmes de parcours d'arbre binaire et des opérations de manipulation de ces structures de données. Ce mini-projet a été une opportunité pour améliorer nos compétences en programmation et en résolution de problèmes et mettre en évidence l'importance de la planification et de l'organisation lors de la réalisation d'un projet informatique.