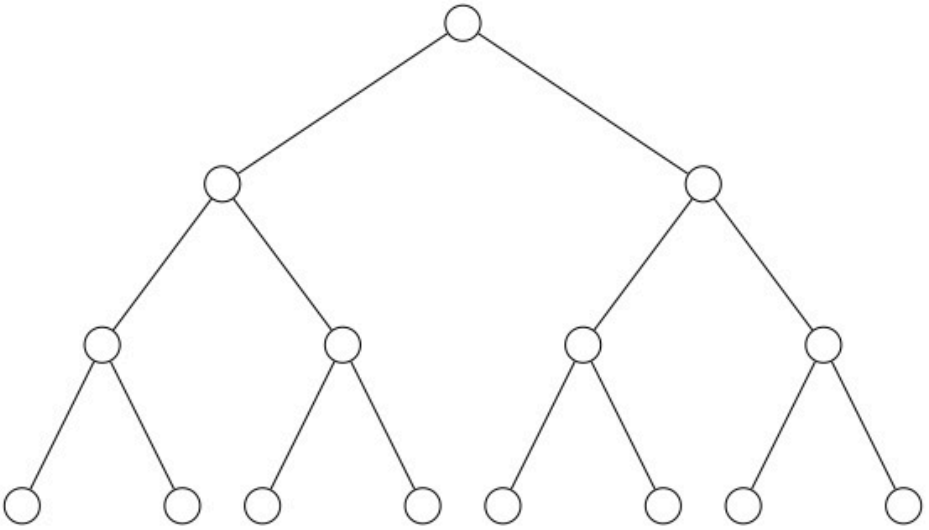
DM : Parcours en largeur d'un arbre binaire



Réalisé par : TANGUY HERREROS
BATUHAN ARIK

SOMMAIRE

I. FONCTIONS IMPLÉMENTÉES

Manipulation de listes:

alloue_cellule(Noeud *n): Alloue dynamiquement une cellule contenant un pointeur vers un nœud.

insere_en_tete(Liste * l, Cellule * c): Insère une cellule en tête d'une liste.

extrait_tete(Liste * l): Extrait et retourne la cellule en tête de liste.

affiche_liste_renversee(Liste lst): Affiche les éléments d'une liste en ordre inverse.

Gestion de file:

initialisation(void): Initialise une file vide.

est_vide(File f): Vérifie si la file est vide.

enfiler(File f, Noeud * n): Ajoute un nœud à la file.

defiler(File f, Noeud ** sortant): Retire un nœud de la fie et met à jour le pointeur fourni.

Création d'arbres:

alloue_noeud(int val, Arbre fg, arbre fd): Crée un noeud avec une valeur et des enfants.

construit_complet(int h, Arbre * a): Construit un arbre binaire complet de hauteur h.

construit_filiforme_aleatoire(int h, Arbre * a, int graine):
Génère un arbre filiforme aléatoire de hauteur h en fonction d'une graine.

Parcours en largeur:

insere_niveau(Arbre a, int niv, Liste * lst): Insère les valeurs des nœuds d'un niveau donné dans une liste.

parcours_largeur_naif(Arbre a, Liste * lst): Effectue un parcours en largeur en utilisant des niveaux successifs.
parcours_largeur(Arbre a, Liste * lst): Parcours en largeur avec une file pour optimiser l'opération.

parcours_largeur_naif_V2(Arbre a, Liste * lst, int * nb_visite): Version améliorée de parcours_largeur_naif(...) comptant le nombre de nœuds visités.

parcours_largeur_V2(Arbre a, Liste * lst, int * nb_visite): Version optimisée du parcours en largeur en comptant le nombre de nœuds visités.

Fonction intermédiaire:

hauteur_arbre(Arbre a): Calcule la hauteur d'un arbre récursivement, utilisé pour les deux fonctions parcours_largeur_naif.

puissance_deux(int n): Calcule 2^n

II. DIFFICULTÉ RENCONTRÉE

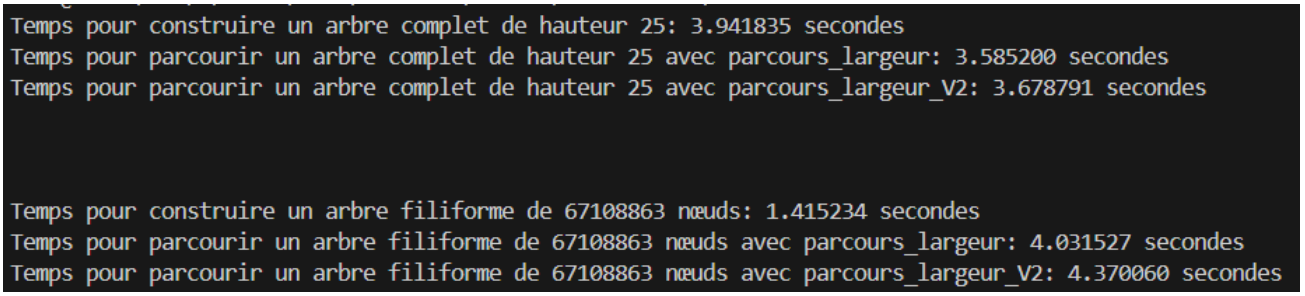
Nous n'avons pas rencontré de grosses difficultés, si ce n'est qu'on a mal compris la gestion des files, ce qui a créé quelques confusions au début. Nous avons donc eu un peu de mal avec la fonction construit_complet. Avec le temps, nous avons mieux saisi leur fonctionnement.

III. MÉTHODE DE TRAVAIL

Nous avons consacré environ 2 à 3 heures par semaine à ce devoir maison et avons échangé régulièrement via Discord. Nous avons aussi fait en sorte d'équilibrer le nombre d'exercices entre nous, afin de répartir les tâches de manière équitable. Parfois, lorsque l'un de nous rencontrait des difficultés sur une fonction, nous nous sommes aidés mutuellement pour trouver des solutions et progresser ensemble.

IV. RÉPONSES AUX QUESTIONS

1. En utilisant les fonctions construit_complet et construit_filiforme_aleatoire, ainsi que d'autres formes d'arbres, écrire un main permettant de tester ces deux dernières fonctions sur différents arbres. Laquelle semble être la plus efficace ?



Pour répondre à cette question, nous avons injecté dans le main des fonctions du module <time.h> permettant de calculer la vitesse d'exécution des fonctions. Nous avons fait en sorte que l'arbre complet et filiforme aient le même nombre de nœuds afin de comparer de manière équitable les deux types d'arbres (en appelant la fonction construit_filiforme_aleatoire avec comme valeur de la hauteur 2^(h) - 1 avec h hauteur de l'arbre complet). Nous pouvons donc voir sur les résultats des calculs qui sont sur l'image ci-dessus que l'arbre complet met **environ 4 secondes** à se créer et que l'arbre filiforme met **environ 1 seconde** à se créer. Nous pouvons aussi voir que parcourir l'arbre complet met **environ 3 secondes** contre **4 secondes** pour l'arbre filiforme. L'efficacité dépend donc de l'utilisation que l'on souhaite en faire : si l'objectif est de créer rapidement un arbre, l'arbre filiforme sera plus adapté, tandis que si l'on cherche à optimiser le parcours de l'arbre, l'arbre complet sera plus performant.

2. Quelle est la complexité de la fonction int parcours_largeur ?

Étapes de la fonction:

-Initialisation de la file: O(1)
-Enfilage de la racine: O(1)
-Boucle principale qui effectue pour chaque noeuds:
    défiler: O(1)
    si le noeud a un fils gauche:
        enfiler: O(1)
    si le noeud a un fils droit:
        enfiler: O(1)
    alloue_cellule: O(1)
    insere_en_tete: O(1)

Etant donné que chaque noeud est traité une seule fois, et que chaque opération sur un noeud a une complexité linéaire, ainsi la complexité de la fonction parcours_largeur est O(n).

3. Quelle est la complexité de la fonction int parcours_naif sur un arbre filiforme ?

Étapes de la fonctions:

-Calcul de la hauteur de l'arbre: O(n)
-Boucle sur la hauteur
    insere_niveau: O(n) (parcours l'arbre entier à chaque appel) et est appelée dans le pire des cas n fois

Ainsi on obtient alors une complexité de $O(n^2)$.

4. Quelle est la complexité de la fonction int parcours_naif sur un arbre complet ?

Étapes de la fonction:

-Calcul de la hauteur de l'arbre: approximativement O(log(n))
-Parcours des niveaux de l'arbre, itère sur chaque niveau de l'arbre, dans un arbre complet, la hauteur est de log(n):
insère_niveau qui parcourt tous les noeuds du niveau N (il y en a 2^N par niveau) , sachant que la hauteur de
l'arbre est d'environ log(n) alors la somme des itérations est approximativement de 2^(log(n))

Ainsi on obtient une complexité finale de O(n).