Rayane KACHBI TP0

Introduction

Ce document fait suite aux travaux pratiques sur les listes doublement chaînées (TP11) et les arbres binaires (TP12). L'objectif est d'explorer conceptuellement comment étendre les structures de données implémentées pour représenter des structures plus complexes : l'arbre générique (ou n-aire) et le graphe quelconque.

Partie 1 : Extension vers un Arbre Générique

Problématique

La structure de nœud implémentée dans le TP12 (`arbre.c`) est spécifiquement binaire. Chaque nœud possède un nombre fixe de pointeurs enfants : un fils gauche et un fils droit.

```c
typedef struct{
    char val;
    int num_crea;
    struct noeud *gauche;
    struct noeud *droit;
} noeud;
```

Listing 1: Structure du nœud binaire (TP12)

Cette structure ne permet pas à un nœud de posséder trois enfants ou plus, ce qui est la définition d'un arbre générique (n-aire).

Solution : Représentation "Fils Gauche, Frère Droit"

Pour permettre à un nœud d'avoir un nombre variable d'enfants sans allouer une liste ou un tableau pour chaque nœud, la solution la plus courante est la représentation "Fils Gauche, Frère Droit".

La structure du nœud est modifiée comme suit :

- Un pointeur (que l'on peut nommer `premierEnfant`) pointe vers le premier enfant du nœud.
- Un second pointeur (que l'on peut nommer `frereSuivant`) pointe vers le "frère" suivant du nœud, c'est-à-dire le prochain enfant du **même** parent.

Les enfants d'un nœud forment ainsi une simple liste chaînée, reliée par les pointeurs `frereSuivant`.

```c
typedef struct noeud_generique {
    char val;
    int num_crea;

    // Pointeur vers le premier enfant
    struct noeud_generique *premierEnfant;

    // Pointeur vers son frere suivant
    struct noeud_generique *frereSuivant;

} noeud_generique;
```

Listing 2: Structure d'un nœud pour arbre générique

Cette approche conserve une taille de structure constante par nœud (toujours deux pointeurs) tout en offrant la flexibilité de représenter n'importe quel nombre d'enfants.

Partie 2 : Extension vers un Graphe Quelconque

Problématique

Un graphe est une structure encore plus générale. Il n'y a plus de notion de hiérarchie (parent/enfant) ni de racine unique. Les relations peuvent être multidirectionnelles et des cycles (revenir à un nœud déjà visité) sont possibles.

Solution : Liste d'Adjacence

La représentation par matrice d'adjacence est une solution, mais elle est très coûteuse en mémoire si le graphe est "creux" (peu d'arêtes).

Une solution plus flexible, qui réutilise les concepts de nos TPs, est la **liste d'adjacence**. Cette méthode combine l'idée d'un tableau (pour les sommets) et de listes chaînées (pour les arêtes/voisins), s'inspirant directement de notre TP11.

Le concept est le suivant :

1. On définit une structure pour un **Sommet** (un nœud du graphe).
2. On définit une structure pour une **Cellule** de liste chaînée, qui servira à lister les voisins.
3. Chaque structure `Sommet` contiendra un pointeur vers la tête d'une liste chaînée de ses "voisins" (les sommets auxquels il est connecté).

```c
// Déclaration anticipée pour permettre la référence croisée
typedef struct Sommet Sommet;

// Structure pour la cellule de la liste des voisins
// (Inspirée du TP11)
typedef struct CelluleVoisin {
    Sommet *voisin; // Pointeur vers le sommet voisin
    struct CelluleVoisin *suivant;
} CelluleVoisin;

// Structure pour le Sommet (le noeud du graphe)
// (Inspirée du TP12)
struct Sommet {
    char val; // Donnée du sommet
    int num;  // Identifiant

    // Tete de la liste chainee de ses voisins
    CelluleVoisin *listeVoisins;
};
```

Listing 3: Structures pour une liste d'adjacence

Pour créer un graphe, on gère un tableau ou une liste de tous les `Sommets`. Pour ajouter une arête (un lien) du `Sommet A` vers le `Sommet B`, il suffit de créer une `CelluleVoisin` pointant vers `B` et de l'insérer (en utilisant la logique de `InsererCellule` du TP11) dans la `listeVoisins` du `Sommet A`.

Cette méthode est efficace en mémoire pour les graphes peu denses et réutilise les principes de listes chaînées vus en TP.