Compte rendu de la partie B du TP0

Ali Gouarab

15 novembre 2025

Structure des dossiers et du code

Pour la première phase du TP0, j'ai choisi de structurer mon travail en deux dossiers distincts, `liste` pour le TP11 et `arbre` pour le TP12. Dans chacun de ces dossiers, j'ai mis en place une organisation identique basée sur trois sous-dossiers, `src`, `include` et `bin`. Le dossier `include` contient les fichiers d'en-tête (par exemple `liste.h` ou `arbre.h`) regroupant les structures et prototypes. Le dossier `src` contient les fichiers sources, dont `liste.c` ou `arbre.c` pour l'implémentation des fonctions, ainsi que `maListe.c` ou `monArbre.c` pour la fonction `main()`. Enfin, le dossier `bin` est réservé aux exécutables générés après compilation. Cette organisation respecte le principe de compilation séparée et clarifie la structure de chaque projet.

Implémentation de la liste doublement chaînée

Définition et représentation d'une liste doublement chaînée

Une **liste doublement chaînée** est une structure de données linéaire composée d'une suite de cellules, où chaque cellule contient à la fois les données et deux pointeurs : l'un vers la cellule suivante (`suiv`) et l'autre vers la cellule précédente (`prec`). Cette structure permet de parcourir la liste dans les deux sens et facilite l'insertion ou la suppression de cellules à n'importe quelle position.

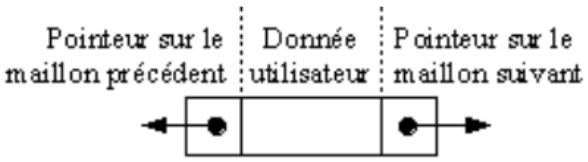

FIGURE 1 – Cellule d'une liste doublement chaînée

Nous souhaitons représenter une forme polygonale par une suite de points (`point`) dans l'espace, stockés dans une liste doublement chaînée. Le travail consiste à définir les structures de données adaptées, à implémenter les fonctions `NouvCel()`, `InsererCellule()`, `SupprimeCellule()` et `Afficher()`, puis à écrire une fonction `main()` permettant de créer la liste, d'y insérer et supprimer des cellules, et d'afficher la forme polygonale.

Fichier `liste.h`

Le fichier `liste.h` contient :

- La définition de la structure `point`, qui représente un couple de coordonnées `x` et `y`.
- La définition de la structure `cellule`, qui contient un `point` et les pointeurs `suiv` et `prec`.
- Les prototypes des fonctions manipulant la liste : `NouvCel()`, `InsererCellule()`, `SupprimeCellule()` et `Afficher()`.

Fichier `liste.c`

Le fichier `liste.c` implémente toutes les fonctions définies dans `liste.h` :

- **NouvCel(point p)** : alloue dynamiquement une nouvelle cellule et initialise ses champs.
- **InsererCellule(int pl, cellule *cel, cellule **liste)** : insère la cellule `cel` à la position `pl`, en mettant à jour les pointeurs `suiv` et `prec` selon les différents cas.
- **SupprimeCellule(int pl, cellule **liste)** : supprime la cellule à la position `pl` et ajuste les pointeurs pour maintenir la cohérence de la liste.
- **Afficher(cellule *liste)** : parcourt la liste et affiche les coordonnées de chaque cellule.

Fichier `maListe.c`

Le fichier `maListe.c` contient la fonction `main()` qui utilise les fonctions de `liste.c` pour manipuler la liste. Il sert à :

- Créer et initialiser la liste.
- Insérer et supprimer des cellules.
- Afficher le contenu de la liste.

Cette séparation en trois fichiers (`.h`, `.c` et `main`) suit le principe de compilation séparée et permet de maintenir le code lisible, modulaire et facilement extensible.

Extension vers un graphe quelconque

Un **graphe** est une structure de données composée d'un ensemble de **noeuds** (ou sommets) et de **lien(s)** (ou arêtes) entre ces noeuds. Chaque noeud peut être connecté à un ou plusieurs autres noeuds. Pour représenter un graphe quelconque, nous utilisons une structure permettant de stocker pour chaque noeud la liste de ses voisins. Cette représentation est particulièrement adaptée aux graphes clairsemés et permet d'ajouter ou de supprimer des arêtes facilement.

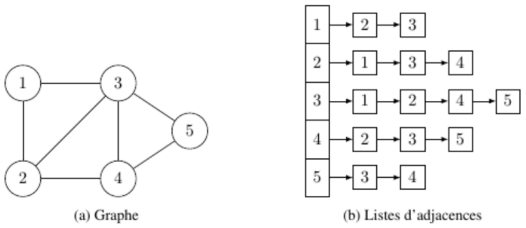

FIGURE 2 – Représentation d'un graphe à l'aide d'une liste d'adjacence

Structures de données

Nous avons défini les structures suivantes dans le fichier `graphe_liste.h` :

- **voisin** : élément d'une liste chaînée représentant un voisin d'un noeud. Contient un pointeur vers le noeud voisin et un pointeur vers le prochain voisin.
- **noeud** : représente un sommet du graphe. Contient un nom et un pointeur vers la liste chaînée de voisins.

Fonctions principales

Les principales fonctions manipulant le graphe sont :

- **creerNoeud(const char *nom)** : crée et initialise un nouveau noeud avec le nom donné.
- **ajouterArete(noeud *n1, noeud *n2)** : ajoute une arête entre deux noeuds en mettant à jour leurs listes de voisins respectives.
- **afficherVoisins(noeud *n)** : affiche la liste des voisins d'un noeud donné.
- **afficherGraphe(noeud **graphe, int nb_noeuds)** : parcourt l'ensemble du graphe et affiche tous les noeuds et leurs voisins.

- **degre(noeud *n)** : retourne le nombre de voisins d'un noeud.
- **libererGraphe(noeud **graphe, int nb_noeuds)** : libère toute la mémoire allouée pour le graphe.

Note importante

Cette amélioration sera implémentée lors du TP4.

Implémentation de l'arbre binaire

Définition et représentation d'un arbre binaire

Un **arbre binaire** est une structure de données hiérarchique composée de **nœuds**, où chaque nœud possède au maximum deux enfants, un **fils gauche** et un **fils droit**. Cette structure permet d'organiser les données de manière arborescente et de réaliser facilement des parcours préfixe, infixe ou suffixe.

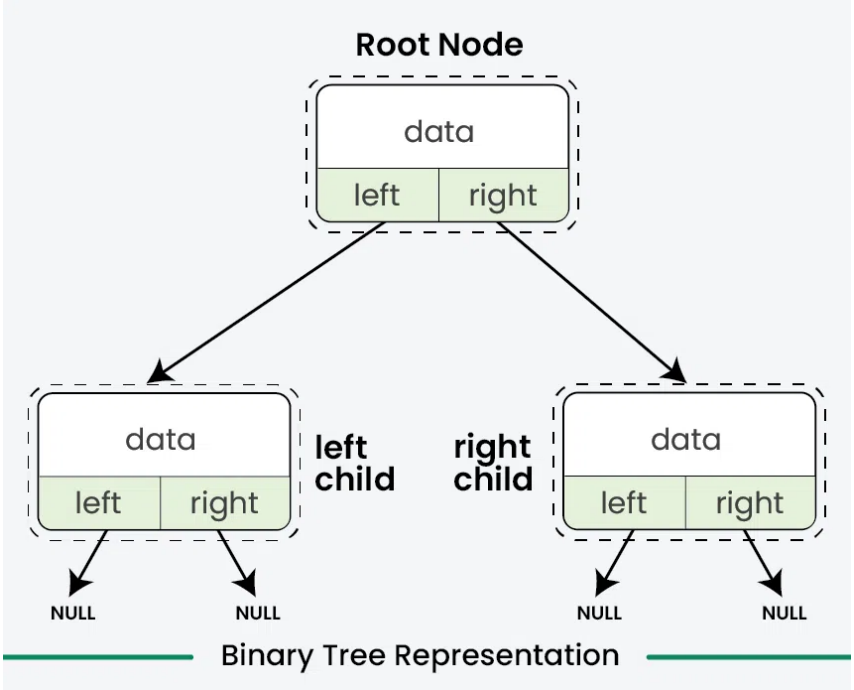

FIGURE 3 – Représentation d'un arbre binaire

Fichier `arbre.h`

Le fichier `arbre.h` contient :

- La définition de la structure `noeud`, contenant un caractère `val`, un identifiant `num`, et les pointeurs `filsGauche` et `filsDroit`.
- Les prototypes des fonctions manipulant l'arbre : `nouvNoeud()`, `rechercheNoeud()`, `insererFG()`, `insererFD()` et `parcoursPrefixe()`.

Fichier `arbre.c`

Le fichier `arbre.c` implémente toutes les fonctions définies dans `arbre.h` :

- **nouvNoeud(char carac)** : crée dynamiquement un nouveau nœud avec un caractère et un identifiant unique.
- **rechercheNoeud(noeud *n, int num_noeud)** : recherche un nœud par son identifiant dans l'arbre.

- **insererFG(noeud *noeudAInserer, noeud *arbre, int num_parent)** : insère un nœud comme fils gauche du nœud parent spécifié.
- **insererFD(noeud *noeudAInserer, noeud *arbre, int num_parent)** : insère un nœud comme fils droit du nœud parent spécifié.
- **parcoursPrefixe(noeud *n)** : parcourt l'arbre en ordre préfixe (Nœud → Gauche → Droit) et affiche les valeurs.

Fichier `monArbre.c`

Le fichier `monArbre.c` contient la fonction `main()` qui utilise les fonctions de `arbre.c` pour manipuler l'arbre. Il sert à :

- Créer et initialiser l'arbre.
- Insérer des nœuds comme fils gauche ou droit.
- Parcourir et afficher l'arbre en préfixe.

Cette organisation en trois fichiers (`.h`, `.c` et `main`) suit le principe de compilation séparée et permet de maintenir le code lisible, modulaire et facilement extensible.

Extension vers un arbre quelconque

Un **arbre générique** est une structure hiérarchique composée de **nœuds**, où chaque nœud peut avoir un nombre arbitraire d'enfants. Pour représenter efficacement cet arbre, nous utilisons la technique dite **first-child / next-sibling** :

- Chaque nœud contient une valeur (`val`) et un identifiant unique (`num`).
- Le pointeur `premierFils` pointe vers le premier enfant du nœud.
- Le pointeur `frèreSuivant` pointe vers le frère suivant du nœud.

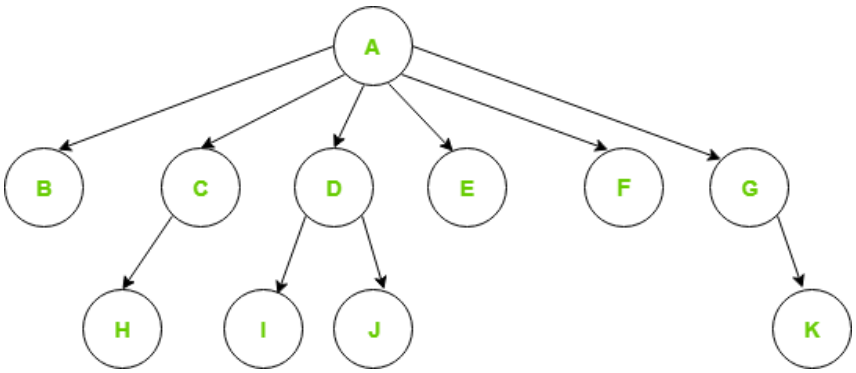

FIGURE 4 – Représentation d'un arbre générique

Structures de données

Le fichier `arbre.h` contient la structure suivante :

- **noeud** : représente un nœud de l'arbre générique. Contient une valeur (`val`), un identifiant unique (`num`) et les pointeurs permettant de relier ses enfants et frères (`premierFils` et `frèreSuivant`).

Fonctions principales

Les principales fonctions manipulant l'arbre générique sont :

- **nouvNoeud(char carac)** : crée et initialise un nouveau nœud avec une valeur donnée et un identifiant unique.
- **rechercheNoeud(noeud *n, int num_noeud)** : recherche un nœud dans l'arbre à partir de son identifiant.

- **ajouterEnfant(noeud *parent, noeud *enfant)** : ajoute un nœud comme enfant du nœud parent donné.
- **parcoursPrefixe(noeud *n)** : effectue un parcours préfixe (Nœud → Premier fils → Frères suivants) et affiche les valeurs des nœuds.

Note importante

Cette structure sera implémentée lors du TP4.