Types Abstraits & Base de la POO - TD 5

‡ ‡ ‡

Types de données Abstraits Hiérarchiques

I - Types de données hiérarchiques : les arbres

Si vous maitrisez la section précédente, vous pouvez maintenant vous frotter à la mise en œuvre des structures hiérarchiques. Commençons par la plus simple : l'arbre.

Exercice I.1

Arbre binaire

1. Réaliser l'implémentation de l'API d'un arbre binaire générique.
2. Réaliser un programme principal réalisant la création d'un arbre binaire contenant des entiers et affichant sa taille et sa hauteur.

Notre TDA arbre binaire maintenant en place, attaquons nous aux parcours ...
Comme nous l'avons vu en cours, les parcours itératifs (profondeur et largeur) nécessite de remplacer la pile d'appels des versions récursives par une structure dynamique adéquate en fonction du parcours ; la *pile* pour le parcours en profondeur et la *file* pour le parcours en largeur.
Tant qu'à faire les choses correctement, nous allons rendre notre module d'arbre indépendant des changements potentiels des modules pile et file !
L'exercice suivant va nous apprendre à créer une librairie dynamique. L'utilisation de cette dernière permettra de prendre en compte les futures éventuelles modifications de celle-ci dans le programme appelant sans avoir besoin de recompiler ce dernier ! magique non ? ?

Exercice I.2

Librairies dynamiques pile & file

Construction d'une librairie dynamique :

Une bibliothèque est une collection de fonctions compilées pouvant être utilisées par un programme. Pour une bibliothèque statique, ce code est intégré à l'exécutable généré pendant l'étape de compilation. Pour une bibliothèque dynamique, le lien entre l'exécutable et la bibliothèque n'est réalisé qu'au moment de l'exécution par le système d'exploitation. Cela permet à un code compilé d'être intégré à plusieurs programmes sans être dupliqué, mais aussi de mettre à jour une bibliothèque sans avoir à recompiler les programmes qui l'utilisent.

Sous Linux, les bibliothèques partagées doivent être nommées `lib<nom>.so`, ce qui donne `libpile.so` pour une bibliothèque `pile`. Les conventions sont dépendantes du système d'exploitation ; le nom de la bibliothèque doit ainsi être `lib<nom>.dylib` sous MacOS et `<nom>.dll` sous Windows.
Ainsi,par exemple, la compilation des fonctions de manipulation d'une pile `pile.c`) sous la forme d'une bibliothèque partagée est réalisée avec la commande suivante :

```
$ gcc -Wall -fPIC -shared -o libtree.so tree.c
```

L'option `-fPIC` (position-independent code) permet de générer du code adapté à une bibliothèque partagée, où les adresses mémoires sont gérées dynamiquement pour rendre les instructions indépendantes de l'emplacement où le code exécutable est chargé en mémoire.
L'option `-shared` indique au compilateur de générer une bibliothèque partagée.

Pour utiliser une bibliothèque lors de la compilation d'un programme, il faut mentionner explicitement cette dernière à l'aide de l'option `-l` (L minuscule). L'option `-L`, optionnelle, donne l'emplacement des bibliothèques si celles-ci ne se trouvent pas dans les répertoires standards – `/usr/local/lib` et `/usr/lib` sous Linux.
Par exemple, la commande suivante compile les un programme `test_pile.c` avec la bibliothèque `libpile.so` nouvellement créée et se trouvant dans le répertoire courant :

```
$ gcc -o test_pile.exe test_pile.c -L. -lpile
```

1. dans le répertoire deque, créer la librairie dynamique correspondant au module `deque.h` en utilisant le nom et l'extension adaptée à votre système. Déplacer cette librairie dans le répertoire contenant le programme `test-deque.c`.
2. Reconstruire l'exécutable `test-deque.exe` en utilisant la nouvelle librairie. Tester.
3. Modifier le code [1] de `deque.c`. Reconstruire la librairie.
4. Exécuter de nouveau `test-deque.exe` *sans le reconstruire*. Alors ? ? Elle est pas belle la vie ? ?

Maintenant que vous êtes convaincu des avantages indéniables de cette techniques, nous allons préparer nos librairies pour l'implémentation des différents parcours itératifs.

5. Construire les bibliothèques dynamiques du module pile et du module file (qui utilise elles-mêmes la librairie du module deque).
6. Copier ou déplacer ces librairies dans le répertoire contenant le module `tree.h`.

Exercice I.3

Parcours d'un arbre binaire

1. Ajouter à votre API d'arbre binaire générique l'implémentation des parcours préfixe, infixe et suffixe en version récursive.
2. Tester ces trois parcours.
3. Ajouter à votre API d'arbre binaire générique l'implémentation itérative des parcours en profondeur.
4. Tester de nouveau vos parcours.
5. Ajouter à votre API d'arbre binaire générique l'implémentation itérative du parcours en largeur.
6. Tester le parcours en largeur.

Exercice I.4

Arbre binaire de recherche

Nous touchons presque au but !

1. Proposer un arbre binaire de recherche complet stockant les valeurs : `1 3 4 6 7 8 10 13 14`
2. En vous basant sur votre API d'arbre binaire, créer une API d'arbre binaire de recherche proposant 4 primitives : recherche, insertion, suppression et parcours infixe.

II - Types de données hiérarchiques : les graphes

Nous allons mettre en œuvre la structure de graphe afin de modéliser le problème des 7 ponts de Königsberg. Ce problème, datant de 1736, est à l'origine de la théorie des graphes et de la topologie.

Exercice II.1

Les 7 ponts de Königsberg

La ville de Königsberg (aujourd'hui Kaliningrad) est construite autour de deux îles situées sur le Pregel et reliées entre elles par un pont.
Six autres ponts relient les rives de la rivière à l'une ou l'autre des deux îles, comme représentés sur le plan ci-contre.
Le problème consiste à déterminer s'il existe ou non une promenade dans les rues de Königsberg permettant, à partir d'un point de départ au choix, de passer une et une seule fois par chaque pont, et de revenir à son point de départ, étant entendu qu'on ne peut évidemment traverser le Pregel qu'en passant sur les ponts.

Commençons tout d'abord par définir une implémentation d'un graphe non orienté. Tout d'abord, la structure de données :

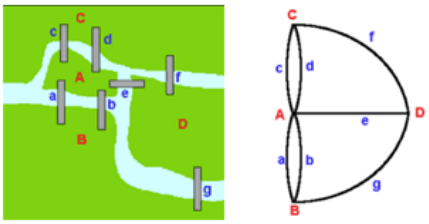

Figure 1 – Modélisation du problème des 7 ponts de Königsberg

1. Par exemple la fonction d'affichage de la deque.

```c
typedef struct so sommet; /* définitions préliminaires */
typedef struct ar arc;
typedef struct nso *l_sommet; /* liste de sommets */
typedef struct nar *l_arc; /* liste d'arcs */

typedef struct so /* sommet */
{
  char valeur; /* l'information associée au sommet */
  l_arc arcs; /* la liste des arcs */
} ;

typedef struct nso /* noeud de la liste des sommets */
{
  sommet s;
  l_sommet suivant;
} ;

typedef struct ar /* arc */
{
  char valeur; /* l'information associée à l'arc */
  l_sommet s1; /* 1er sommet de l'arc */
  l_sommet s2; /* 2ème sommet de l'arc (ordre utile si graphe ordonné) */
} ;

typedef struct nar /* noeud de la liste des arcs */
{
  arc a;
  l_arc suivant;
} ;

typedef l_sommet graphe; /* le graphe contient sa liste de sommets
 chaque sommet contient la liste de ses arcs (entrant ou sortant) */
```

Listing 1 – Structure de données

On associe à cette structure de données un ensemble de primitives élémentaires :

```c
/* Crée un graphe vide */
graphe grapheVide();

/* ajoute le sommet 'valeur' dans g
 pré : le sommet ne doit pas exister */
void ajouterSommet ( char valeur, graphe *g);

/* ajoute l'arc 'valeur' entre les sommets 's1' et 's2' dans g
 pré : l'arc 'valeur' ne doit pas exister */
void ajouterArc ( char valeur, char s1, char s2, graphe *g);

void afficher ( graphe g);
```

Listing 2 – Primitives du module.

Ainsi, le programme principal permettant de supporter le problème des 7 ponts peut alors s'écrire :

```c
...
graphe g=grapheVide();
ajouterSommet('A',&g);ajouterSommet('B',&g);ajouterSommet('C',&g);
ajouterSommet('D',&g);
ajouterArc('a','A','B',&g);ajouterArc('b','A','B',&g);
ajouterArc('c','A','C',&g);ajouterArc('d','A','C',&g);
ajouterArc('e','A','D',&g);
ajouterArc('f','C','D',&g);
ajouterArc('g','B','D',&g);
afficher(g);
...
```

Listing 3 – Extrait du programme principal.

Les ajouts se faisant en tête de liste, voici le résultat de l'exécution :

```
sommet D
arc g =(B,D)
arc f =(C,D)
arc e =(A,D)
```

```
sommet C
arc f =(C,D)
arc d =(A,C)
arc c =(A,C)
sommet B
arc g =(B,D)
arc b =(A,B)
arc a =(A,B)
sommet A
arc e =(A,D)
arc d =(A,C)
arc c =(A,C)
arc b =(A,B)
arc a =(A,B)
```

1. Réaliser l'implémentation de l'API `libgraphe` définie précédemment.
2. Réaliser le programme principal permettant de construire le graphe support du problème des 7 ponts.
3. Créer une deuxième API proposant les mêmes primitives mais se basant sur une implémentation par matrice d'adjacence.

Si le temps le permet, nous nous proposons d'analyser le parcours d'un graphe de sommet en sommet. Pour ce faire, nous allons réaliser une action récursive qui construit les chemins partant d'un sommet donné (sans revenir sur un sommet déjà parcouru) :

4. Comment définir la notion de chemin ?
5. Spécifier les primitives utiles sur ce type de donnée pour notre problème :
   (a) Construction de chemin
   (b) Test d'un sommet dans le chemin
6. Spécifiez l'action qui construit le chemin en parcourant le graphe.
7. Analysez le critère d'arrêt, puis programmez récursivement l'action.
8. Testez votre programmation sur l'exemple des ponts de Königsberg.

‡ ‡ ‡