Feuille de travaux dirigés n° 2
Types abstraits

Exercice 2.1

On dispose du type « *liste simplement chaînée de* `double`*s* » `dlistp_t` et des fonctions suivantes :

`dlistp_t add_node(double v, dlistp_t next).` Crée un nœud contenant la valeur `v` et le relie à la liste dont la tête est `next` ;

`double get_val(dlistp_t node).` Retourne la valeur stockée dans le nœud identifié par `node` ;

`dlistp_t delete_head(dlistp_t head).` Détruit le nœud pointé par `head` et retourne un pointeur vers le reste de la liste.

1. Définir le type `dstack_t` « *pile de* `double`*s* » en utilisant une représentation par liste chaînée. On fera en sorte de pouvoir connaître le nombre de valeurs dans la pile avec une fonction de complexité temporelle $\mathcal{O}(1)$;
2. Écrire la fonction `dstackp_t new_stack(void)` (ou `dstackp_t` est un type « *pointeur sur pile* `dstack_t` ») créant une nouvelle pile vide ;
3. Écrire la fonction `bool isempty(dstackp_t st)` retournant `true` si la pile est vide et `false` sinon ;
4. Écrire la fonction `void push(double v, dstackp_t st)` empilant la valeur `v` sur la pile `st` ;
5. Écrire la fonction `double pop(dstackp_t st)` retirant la valeur sur le haut de la pile et la retournant ; la fonction devra afficher un message sur la sortie d'erreur et retourner `NAN` si la pile est vide lors de l'appel ;
6. Écrire un programme principal empilant trois valeurs sur une pile, puis les dépilant pour les afficher.

Exercice 2.2

On considère les expressions suivantes :

$$\boxed{\begin{array}{l} 3x + y - 2z \\ 4 + 3 * 7 - 5/(3 + 4) + 6 \\ \log(x + y)^2 - \frac{\sin x}{2z} \end{array}}$$

1. Dessiner l'arbre correspondant à chaque expression ;
2. Réécrire chaque expression en format préfixé.

Exercice 2.3

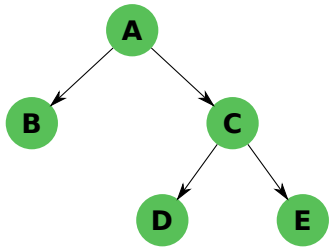
On souhaite manipuler des arbres binaires de caractères.

1. Définir le type nœud d'arbre binaire de caractères `ctree_t` et le type pointeur sur `ctree_t`, `ctreep_t` ;
2. Écrire la fonction `ctreep_t create_ctree(char v, ctreep_t left, ctreep_t right)` créant un arbre avec une racine contenant `v` et des sous-arbres gauche et droit `left` et `right` ;
3. Écrire la fonction `void delete_ctree(ctreep_t root)` détruisant l'arbre de racine `root` ;

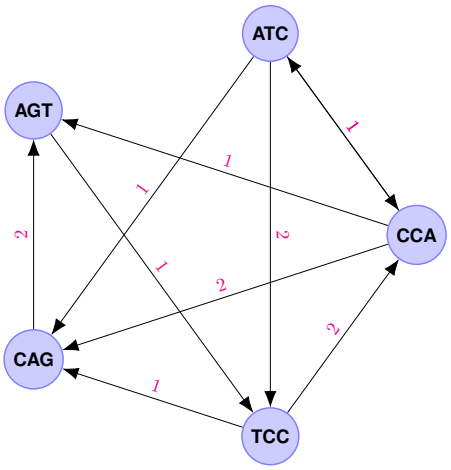
Dans la suite, on considère le type « *pointeur sur fonction prenant en entrée un paramètre de type* `char` *et ne retournant rien* » `visit_node_f`.

4. Écrire la fonction `void prefix(ctreep_t root, visit_node_f f)` visitant l'arbre de racine `root` en ordre préfixe et appliquant la fonction `f` sur la valeur stockée dans chaque nœud ;

5. Écrire la fonction `void infix(ctreep_t root, visit_node_f f)` visitant l'arbre de racine `root` en ordre infixe et appliquant la fonction `f` sur la valeur stockée dans chaque nœud ;
6. Écrire la fonction `void postfix(ctreep_t root, visit_node_f f)` visitant l'arbre de racine `root` en ordre postfixe et appliquant la fonction `f` sur la valeur stockée dans chaque nœud ;
7. Écrire le programme principal créant l'arbre de la figure 1a, l'affichant en ordre préfixe, infixe, puis postfixe et le détruisant.



(a) Arbre de caractères pour l'exercice 2.3.



(b) Graphe pour l'exercice 2.4.

FIGURE 1 – Figures pour les exercices

Exercice 2.4

On souhaite manipuler des graphes orientés avec des poids entiers sur les arcs et une valeur de type « *chaîne de caractères* » dans les nœuds.

1. Définir le type `graph_t` permettant de représenter un tel graphe par sa matrice d'incidence. On définira aussi le type pointeur sur `graph_t`, `graphp_t` ;
2. Définir la fonction `graphp_t graph_create(size_t V)` créant une structure de graphe possédant au maximum `V` nœuds ;
3. Définir la fonction `void graph_delete(graphp_t g)` détruisant un graphe en mémoire ;
4. Définir la fonction `node_t graph_add_node(graphp_t g, const string& s)` ajoutant un nœud au graphe `g` de valeur `s` et retournant un identifiant du nœud créé permettant de le manipuler ultérieurement. La chaîne `s` sera copiée dans le nœud et l'on affichera un message sur la sortie d'erreur si le nombre de nœuds créés est supérieur au maximum défini à la création du graphe ;
5. Définir la fonction `void graph_add_edge(graphp_t g, node_t n1, node_t n2, int w)` ajoutant une arête entre les nœuds `n1` et `n2` de poids `w` ;
6. Définir la fonction `void graph_walk(graphp_t g, node_t n, node_fun f)` effectuant une « marche » dans le graphe `g` à partir du nœud `n` : on part de `n` et l'on se déplace sur le premier nœud accessible à partir de `n` non encore traversé. La marche s'arrête quand on ne peut plus aller sur un autre nœud. À chaque passage sur un nœud, on appelle la fonction `f` en lui passant en paramètre la chaîne de caractères se trouvant dans le nœud ;
7. Écrire un programme principal créant le graphe de la figure 1b et effectuant une *marche* à partir du nœud « **ATC** ».