1 Implémentation d'une structure de pile

Exercice 1 (pile à la Python) : Le but de cet exercice est d'implémenter une structure de pile d'entiers en utilisant des tableaux, et dont l'opérateur "empiler" fonctionne comme le `append` des listes Python.

Notre structure `pile` aura donc 3 champs :

- `int *tab` : un tableau d'entiers ;
- `int taille` : la taille de `tab` ;
- `int nb` : le nombre d'éléments dans la pile.

Le principe de cette implémentation est le suivant :

- Les éléments de la pile sont stockés au début du tableau `tab`.
- Pour dépiler, il suffit de décrémenter `nb`.
- Pour empiler :
  - si `tab` n'est pas rempli, on utilise la première case non utilisée ;
  - sinon, on crée un nouveau tableau de taille `2*taille+1` dans lequel on recopie les éléments de `tab`, et on continue de travailler avec ce tableau.

1. Créer deux fichiers `pile.c` et `pile.h`, et créer un type structuré `pile`.
2. Implémenter les opérations sur les piles en utilisant le principe décrit ci-dessus.
   Voici les signatures que devront avoir vos fonctions :

```c
pile *creer_pile(); // crée une nouvelle pile vide
bool pile_est_vide(pile *p); // teste si p est la pile vide
void empiler(pile *p, int val); // empile val au sommet de p
int sommet(pile *p); // renvoie le sommet de p (p doit être non vide)
int depiler(pile *p); // dépile l'élément au sommet de p (non vide) et le renvoie
void detruire_pile(pile *p); // libère p de la mémoire
```

3. Quelle est la complexité de chacune de ces opérations ?

2 Écriture polonaise inversée

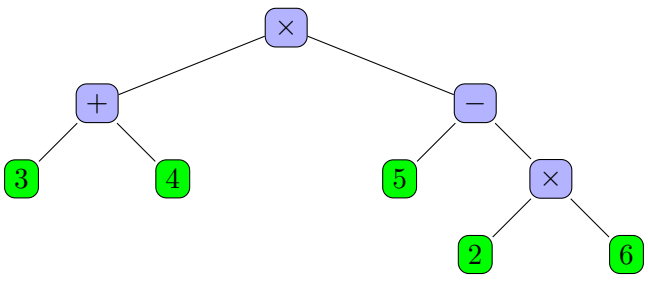
Exercice 2 (Arbre syntaxique) : Pour représenter une expression arithmétique, nous avons l'habitude d'utiliser certains symboles :

$$(3+4)\times(5-(2\times 6))$$

En général, nous utilisons ce qu'on appelle une écriture **infixe**, c'est-à-dire qu'un **opérateur binaire** (opérateur prenant deux arguments) est écrit au milieu de ses deux opérandes (ou arguments).

Pour nos ordinateurs, il n'est pas très pratique d'utiliser la chaîne de caractères "$(3+4)\times(5-(2\times 6))$". Un programme préfèrera utiliser l'**arbre syntaxique** de cette expression, qu'il va devoir commencer par calculer.

L'arbre syntaxique de l'expression ci-dessus est le suivant :



En informatique, un arbre est une structure constituée de **nœuds** reliés entre eux de manière **hiérarchique** par des **branches**.

Sur l'arbre ci-dessus, le nœud "+" possède deux **enfants** : les nœuds "3" et "4". Les opérateurs possèdent tous deux enfants : leurs deux opérandes. On parle de **nœuds** internes. Les nombres ne possèdent pas d'enfants : on parle de **feuilles** (car ils sont au bout d'une branche). Un seul nœudne possède pas de **parent** : c'est la **racine** de l'arbre. Tous les autres nœudspossèdent exactement un parent.

1. Donner l'arbre syntaxique de l'expression $(3 + (4 \times 5)) - (2 \times 6)$.
2. Donner l'arbre syntaxique de l'expression $3 + ((4 \times 5) - (2 \times 6))$.

Pour récupérer l'expression infixe correspondant à un arbre syntaxique, on peut procéder de la manière suivante :

- On démarre à la racine de l'arbre ;
- pour chaque nœudinterne :
  - la première fois qu'on passe par ce nœud, on descend dans son enfant de gauche ;
  - la deuxième fois qu'on passe par ce nœud, on affiche l'opérateur ;
  - la troisième fois qu'on passe par ce nœud, on remonte à son parent (si c'est la racine, on s'arrête).
- pour chaque feuille : on écrit le nombre et on remonte au parent.

3. Effectuer un parcours infixe de l'arbre ci-dessus, ainsi que des deux arbres obtenus lors des questions précédentes.
4. Qu'observez-vous ?

Ce qu'on vient de faire s'appelle un **parcours en profondeur infixe** de l'arbre. Il existe deux autres types de **parcours en profondeur** d'un arbre : les parcours **préfixe** et **postfixe**.

- Parcours préfixe. Pour chaque nœudinterne :on traite d'abord le contenu du nœud, puis le sous-arbre gauche, puis le sous-arbre droit.
- Parcours postfixe. Pour chaque nœudinterne :on traite d'abord le sous-arbre gauche, puis le sous-arbre droit, puis le contenu du nœud.

On peut prouver que, contrairement à l'écriture infixe, nous n'avons jamais besoin de parenthèses pour utiliser l'écriture préfixe ou postfixe. L'écriture postfixe s'appelle aussi l'**écriture polonaise inversée**.

5. Donner l'écriture polonaise inversée des 3 arbres de cet exercice.

Exercice 3 (Écriture polonaise inversée) : L'écriture polonaise inversée possède deux avantages :

- il n'y a jamais besoin de parenthèses ;
- on peut évaluer la valeur d'une expression écrite en polonaise inversée simplement en parcourant cette écriture de gauche à droite et en utilisant une pile.

À cause de cela, certaines vieilles calculatrices fonctionnaient avec l'écriture polonaise inversée, car on pouvait calculer des expressions de manière très efficace (et à l'époque, les calculatrices étaient beaucoup moins puissantes qu'aujourd'hui).

1. Créer un fichier `polonaise.c` dans lequel vous importez le fichier `pile.h` de l'exercice 1.

```c
#include "pile.h"
```

Une calculatrice en écriture polonaise inversée repose sur le principe suivant :

- on démarre le calcul avec une pile vide ;
- on parcours l'expression de gauche à droite ;
- si on tombe sur un nombre, on l'empile ;
- si on tombe sur un opérateur (`op`),
  - on dépile les deux derniers éléments de la pile (notés `a` et `b`),

- on calcule `a op b`,
- et on empile le résultat ;

- si tout va bien (pas d'erreur de syntaxe), à la fin la pile ne contient plus qu'un seul élément : le résultat du calcul.

2. Écrire une fonction `int nb(char n);` prenant en argument le caractère '0', '1', ..., '8' ou '9', et renvoyant sa valeur sous la forme d'un entier.
3. Écrire une fonction `int operation(char op, int a, int b);` qui prend en arguments un caractère `op` (valant `'+'`, `'-'`, `'*'`, ou `'/'`) et deux entiers a et b, et renvoie le résultat de `a op b`.
4. Écrire une fonction `int polonaise(char *expr);` implémentant la méthode décrite ci-dessus.
   Pour simplifier, on supposera que l'expression ne contient pas de nombre supérieur ou égal à 10.
5. Tester votre programme sur les expressions de l'exercice 2.

3 Si vous avez fini

Exercice 4 (Structure de file) : Implémenter une structure de file à l'aide de deux piles.

Exercice 5 (Listes chaînées) : Implémenter une structure de pile reposant sur le principe des listes chaînées.

Exercice 6 (Listes doublement chaînées) : Implémenter une structure de file reposant sur le principe des listes doublement chaînées.