DM 1 - Parcours en largeur d'un arbre binaire

Dans ce DM, on se propose d'implémenter deux méthodes pour effectuer un parcours en largeur d'un arbre et de les comparer.

Dans tout ce DM, on considérera le type "arbre binaire d'entiers" suivant :

```
1 typedef struct _noeud {
2     int valeur;
3     struct _noeud *fg, *fd;
4 } Noeud, * Arbre;
```

1 Préliminaires

Le type abstrait de données `File` correspond, de manière imagée, à une file d'attente à la caisse d'un supermarché, vue d'un usager : une personne arrivant se met en début de la queue, patiente et sortira par la fin de la queue une fois arrivée à la caisse.

On rappelle que les éléments à l'intérieur de la file sont chainée par une liste chainée allant de la fin de la file vers son début, afin de pouvoir accéder facilement (*i.e.* en $\mathcal{O}(1)$) au futur prochain élément sortant de la `File`.

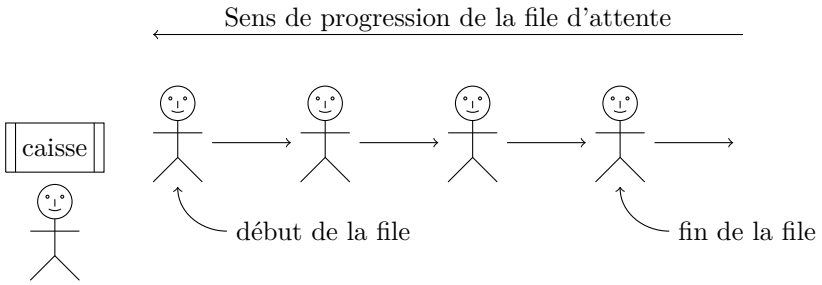

FIGURE 1 – Illustration d'une file d'attente et de sa représentation en machine

Nous utiliserons le type `Liste` suivant :

```
1 typedef struct cell {
2     Noeud * n;
3     struct cell * suivant;
4     struct cell * precedent;
5 } Cellule, * Liste;
```

Dans la modélisation d'une file que nous considérerons ici, une `Queue` sera une sauvegarde de la tête d'une liste chainée et de sa dernière cellule. Une `File` sera alors un pointeur sur une `Queue`.

```
1 typedef struct {
2     Liste debut;
3     Liste fin;
4     int taille;
5 } Queue, * File;
```

Toutes les fonctions demandées dans cet exercice, sauf les fonctions `affiche_liste_renversee`, `construit_complet` et `construit_filiforme_ aleatoire`, doivent être de complexité $\mathcal{O}(1)$.

1. Écrire les fonctions suivantes de manipulation de listes doublement chainées :
   - `Cellule * alloue_cellule(Noeud * n)`, qui alloue sur le tas une `Cellule` contenant le pointeur sur un nœud `n` ;
   - `void insere_en_tete(Liste * l, Cellule * c)`, qui ajoute en tête de `*l` la cellule `*c` ;
   - `Cellule * extrait_tete(Liste * l)`, qui retire le premier maillon de la liste `*l` s'il existe et le renvoie (et qui renvoie `NULL` sinon).
   - `void affiche_liste_renversee(Liste lst)` qui affiche sur la sortie les éléments de la liste `lst` du dernier au premier.
2. — Écrire la fonction `File initialisation(void)` qui initialise une `Queue` vide sur le tas : les attributs de la structure seront mis à `NULL` si possible.
   — Écrire la fonction `int est_vide(File f)` qui renvoie l'entier 1 si la file `f` passée en paramètre est vide, (c'est-à-dire si la liste chainée sous-jacente est vide) et 0 sinon.
   — Écrire la fonction `int enfiler(File f, Noeud * n)` qui fait rentrer le pointeur `n` dans la file `f`. La fonction renverra `1` si l'entier `n` a pu être enfilé et `0` dans le cas contraire.
   — Écrire la fonction `int defiler(File f, Noeud ** sortant)` qui fait sortir de la file `f` un pointeur sur un nœud et le stocke à l'adresse `sortant`. Cette fonction renverra 1 si une adresse a effectivement pu être retiré de la file, 0 sinon.

   On n'oubliera pas de vérifier que la file est non vide, puis de libérer la zone mémoire associée à l'entier sortant de la file.
3. a. Écrire la fonction `Noeud * alloue_noeud(int val, Arbre fg, Arbre fd)` qui alloue sur le tas un `Noeud` contenant la valeur `val` et `fg` et `fd` comme enfant.
   b. Écrire la fonction `int construit_complet(int h, Arbre * a)` qui construit dans `*a` l'arbre complet de hauteur `h` et dont le parcours en largeur est 1, 2, $\cdots$, $2^{\mathtt{h}+1} - 1$.

   La fonction sera de complexité $\mathcal{O}(2^h)$. Elle renverra 1 si l'arbre a été correctement construit et 0 dans le cas contraire. En cas d'échec, la fonction devra libérer la mémoire allouée.
   c. Écrire la fonction `int construit_filiforme_aleatoire(int h, Arbre * a, int graine)` qui construit en mémoire un arbre filiforme, dont la forme sera aléatoire (l'aléatoire sera fixé par le paramètre `graine`) et dont le parcours en profondeur préfixe sera 1, 2, $\cdots$, $\mathtt{h} + 1$.

   La fonction sera de complexité $\mathcal{O}(h)$. Elle renverra 1 si l'arbre a été correctement construit et 0 dans le cas contraire. En cas d'échec, la fonction devra libérer la mémoire allouée.

2 Deux méthodes pour réaliser un parcours en largeur

2.1 Un parcours en largeur naïf

1. Écrire la fonction `int insere_niveau(Arbre a, int niv„ Liste * lst)` qui ajoute toutes les valeurs des nœuds de l'arbre `a` dont le niveau vaut `niv` à la liste `*lst`.

   La fonction renverra 0 en cas de problème d'allocation., elle laissera alors la liste `*lst` en l'état. Elle renverra 1 lorsque tous les nœuds de l'arbre `a` situés au niveau `niv`, s'il y en a, ont été correctement insérés dans `*lst`.

   Un enfant de gauche d'un nœud sera toujours inséré dans `*lst` *avant l'enfant de droite de ce nœud.*
2. Pour réaliser le parcours en largeur d'un arbre, il suffit alors d'ajouter à une liste initialement vide tous les nœuds dont le niveau vaut d'abord 0, puis 1, puis 2, jusqu'à atteindre la hauteur de l'arbre `a`.

   Écrire la fonction `int parcours_largeur_naif(Arbre a, Liste_Int * lst)` qui réalise le parcourt en largeur d'un arbre `a` en suivant cette idée.

La fonction renverra 1 si le parcours en largeur a pu être réalisé en entier. Elle renverra 0 en cas de problème d'allocation, tout en laissant la liste `*lst` en l'état..

Attention : L'écriture de la fonction `hauteur` n'est pas nécessaire.

2.2 Un parcours en largeur, à l'aide d'une pile

L'algorithme 1 rappelle l'algorithme itératif, vu en cours et en TD, pour réaliser un parcours en largeur

Algorithm 1 Parcours en largeur d'un arbre

```
lst = liste vide
f = file vide
enfiler(f, a)
while f est non vide do
    n = defiler(f)
    if n est non vide then
        enfiler le fils gauche de n dans f
        enfiler le fils droit de n dans f
        ajouter la valeur du nœud a à la liste lst
    end if
end while
return lst
```

Écrire la fonction `int parcours_largeur(Arbre a, Liste * lst)` qui réalise le parcourt en largeur d'un arbre `a` en suivant cet algorithme. La fonction renverra 1 si le parcours en largeur a pu être réalisé en entier et 0 en cas de problème d'allocation mémoire.

3 Comparaison des méthodes

1. Modifier la fonction `int parcours_largeur_naif(Arbre a, Liste_Int * lst)` en la fonction `int parcours_largeur_naif_V2(Arbre a, Liste_Int * lst, int * nb_visite)` qui, en plus de réaliser le parcours en largeur de manière naïve de l'arbre `a`, comptera aussi le nombre de nœuds de `a` visité lors de son parcours.
2. Modifier la fonction `int parcours_largeur(Arbre a, Liste_Int * lst)` en la fonction `int parcours_largeur_V2(Arbre a, Liste_Int * lst, int * nb_visite)` qui, en plus de réaliser le parcours en largeur de l'arbre `a`, comptera aussi le nombre de nœuds de `a` visité lors de son parcours.
3. *(La réponse à l'intégralité de cette question, hors code devra figurer comme une section à part entière dans le rapport ; les réponses devront être justifiée.)*
   a. En utilisant les fonctions `construit_complet` et `construit_filiforme_aleatoire`, ainsi que d'autres formes d'arbres, écrire un `main` permettant de tester ces deux dernières fonctions sur différents arbres. Laquelle semble être la plus efficace ?
   b. Quelle est la complexité de la fonction `int parcours_largeur` ?
   c. Quelle est la complexité de la fonction `int parcours_naif` sur un arbre filiforme ?
   d. Quelle est la complexité de la fonction `int parcours_naif` sur un arbre complet ?

4 Conditions de rendus

Ce devoir est à réaliser en binôme au sein d'un même groupe de TP. Le devoir est à rendre sur le e-learning du cours pour le dimanche 23 février 2025, 23h59.

Le code sera réalisé dans un seul fichier nommé `DM_1.c`. Les structures et prototypes de fonctions proposées dans le sujet, ainsi que dans le fichier `DM_1.h` ne doivent pas être modifiées. En cas de besoin, des fonctions intermédiaires peuvent être ajoutées.

Pour tester votre devoir, un second fichier nommé `main.c` pourra être écrit et rendu. Vous aurez aussi accès sur le e-learning à un fichier nommé `tests_prof.o` de quelques tests semblables à ceux qui seront utilisés pour corriger votre devoir.

Pour compiler votre fichier, vous utiliserez alors les lignes de commande suivantes (voir plus tard en Perf. C pour avoir plus d'information sur la programmation modulaire) :

`clang DM_1.c -c -o DM_1.o`	$\rightsquigarrow$	transforme le fichier `DM_1.c` en un fichier objet
`clang DM_1.o tests_prof.o -o tests`	$\rightsquigarrow$	crée un exécutable `tests` à partir de votre code et de celui comprenant les tests des enseignants.
`clang DM_1.o main.o -o DM_1`	$\rightsquigarrow$	crée un exécutable `DM_1` à partir de votre code et du `main` que vous avez écrit.

Un fichier `.zip` y sera déposé contenant a minima le fichier de code `.c` que vous avez développé ainsi qu'un fichier `rapport.pdf` décrivant succinctement les fonctions implémentées, les difficultés rencontrées, la répartition du travail au sein du binôme et votre réponse aux questions de la fin de la partie "Comparaison des méthodes".

L'archive et le répertoire qu'elle contiendra seront nommés selon la nomenclature `nom1_nom2.zip` où `nom1` et `nom2` désigne les noms de famille des membres du binôme. Le rapport contiendra l'intégralité des noms et prénoms des membres du binômes.

La correction s'effectuera en partie de manière automatique. Il est donc particulièrement important que vous respectiez scrupuleusement l'intégralité de la nomenclature donnée.