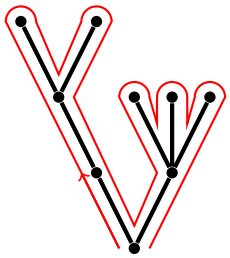
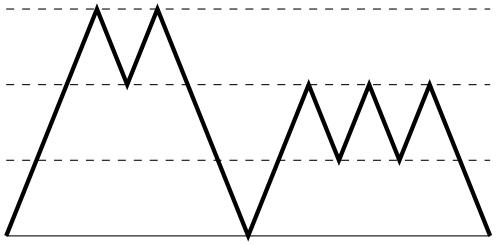
Université Pierre et Marie Curie — **Master M1 "Mathématiques et Applications"**
Période 2 — **Programmation en C/C++**

TD 4 : Pointeurs, arbres planaires enracinés dynamiques, programmation récursive

Le but de ce TD est de construire et manipuler des arbres planaires enracinés. Un arbre enraciné est un graphe sans boucle, dont on a particularisé un nœud. On peut donc le représenter (voir les schémas ci-dessous) par couche ou *génération* selon la distance d'un nœud à la racine. Le père d'un nœud est l'unique nœud voisin plus proche de la racine, les enfants sont les nœuds voisins plus éloignés de la racine. Un arbre planaire est un arbre dont les enfants de chaque nœud sont ordonnés de gauche à droite. La hauteur d'un nœud est la distance d'un nœud à la racine.
Le processus de contour associe à tout arbre planaire enraciné une fonction positive de $\{1, \ldots, 2n_{\text{arêtes}}\} \to \mathbb{N}$ telle que $f(i+1) - f(i) \in \{-1, 1\}$ et telle que $f(i)$ soit égal à la hauteur dans l'arbre du i-ème nœud visité par le contour :





L'arbre binaire enraciné sera codé par la structure suivante pour chaque nœud :

```
typedef struct noeud {
    int    value;
    int        height;
    struct noeud *  pere;
    int nbfils ;
    struct noeud ** fils;
    int rang;
} t_noeud;
```

où le champ `valeur` permettra plus tard d'associer une variable à chaque nœud de l'arbre. Le champ `height` contient la hauteur d'un nœud, le champ `pere` l'adresse du nœud père, le champ `nbfils` le nombre d'enfants, le champ `fils` le tableau dynamique des adresses des enfants et enfin le champ `rang` d'un individu désigne l'indice qui le représente dans la liste `fils` de son nœud père.
Le processus de contour sera décrit par une liste chaînée de valeurs de hauteurs :

```
typedef struct path {
        int height;
        struct path * next;
} t_path;
```

Une fois arrivé au dernier point du contour, le pointeur `next` doit pointer vers la valeur `NULL`. Pour qu'un contour corresponde au contour d'un arbre, il faut que les hauteurs soient positives, que la première et la dernière soit nulles et qu'entre deux points consécutifs, la différence de hauteur soit ± 1.

1 *Écrire la fonction de création* `cree_racine` *de la racine d'un arbre : elle crée un noeud simple avec* `malloc` *et renvoie l'adresse de l'objet créé :*

```
t_noeud * cree_racine(void);
```

Écrire la fonction `cree_enfants` *qui ajoute n enfants à un nœud* `individu` *en complétant le code suivant :*

```
_____ cree_enfants( ________ individu, int const n) {
                t_noeud * courant ;
                int i;
                individu->nbfils= ________ ;
                individu->fils = malloc(__________);
                for (i = 0; i < n; i++)
                {
                        courant = _____________);
                        individu->fils[i] = courant;
```

```
10                          courant->rang=i;
                            courant->height=___________ ;
12                          courant->value=0;
                            courant->pere=_________;
14                          courant->nbfils=0;
                            courant->fils=____________;
16                  }
                    return ______;
18  }
```

Écrire la fonction `frere_gauche` *(resp.* `frere_droit`*) qui donne l'adresse du frère gauche (resp. droit) d'un nœud s'il existe (ces fonctions ne sont pas strictement nécessaires pour la suite mais elles permettent de se familiariser avec la navigation dans l'arbre).*

2 *Écrire les fonctions récursives* `hauteur` *qui donne la hauteur de l'arbre (hauteur maximale des feuilles) et la fonction* `taille` *qui donne le nombre total de nœuds d'un arbre.*

3 *Écrire la fonction récursive de prototype*

```
void arbre_n_aire(t_noeud * root, int n, int h)
```

qui crée à partir d'un nœud racine un arbre régulier n-aire (chaque nœud a exactement n enfants) de profondeur h. ***Indice :*** *la fonction pour la hauteur h doit appeler un certain nombre de fois la même fonction pour la valeur* $h-1$ *si* $h>1$.

4 *Tester ces fonctions sur le cas de l'arbre n-aire avec quelques valeurs de n et de la hauteur.*

5 (arbre de Galton-Watson de degré au plus 2) *Le but de cette question est de créer un arbre aléatoire de profondeur au plus h, tel que chaque nœud (sauf les derniers) a un nombre* $m \in \{0,1,2\}$ *aléatoire d'enfants indépendant des autres nœuds, de loi* $\mathbb{P}(m=1)=p_1$ *et* $\mathbb{P}(m=2)=p_2$ *avec* $p_1+p_2<1$.
Nous proposons le code suivant à compléter :

```
 int galton_watson_2( ______ racine, double p1,double p2, int h)
2 {
                    if (p1+p2 > 1) { printf{"Mauvaises probabilites !\n");return 0};
4                   if (h<0) return ___ ;
                    int i;
6                   int n[2];
                    double r;
8                   r=rand()/((_____) _______);
                    if (r<p2)
10                  {
                            cree_enfants(_____,____);
12                          for (i = 0; i < ____; i++)
                            {
14                                  n[i]=___________(racine->fils[i],p1,p2,__________);
                            }
16                          return 1+n[0]+n[1];
                    }
18                  if (r<p2+p1)
                    {
20                          cree_enfants(_____,_____);
                            n[0] = galton_watson_2(racine->fils[0],p1,p2,__________);
22                          return 1+n[0];
                    }
24                  return 1;
}
```

Préciser ce que renvoie cette fonction (justifier ce choix) et compléter le code.

6 *Écrire la fonction* `cree_path` *qui crée un chemin réduit à un seul point de hauteur donnée et renvoie son adresse et la fonction de prototype*

```
void print_path(t_path * const chemin, FILE * nom_du_fichier)
```

qui écrit dans un fichier pour chaque site visité une ligne qui donne la hauteur du nœud. Écrire également la fonction `vide_path` *qui détruit un chemin complet.*

7 *Pour écrire la fonction* `contour_process` *qui correspond au processus de contour de l'arbre décrit dans l'introduction, nous proposons le code récursif à trous suivant :*

```
t_path * contour_process(t_noeud * const racine, t_path * chemin)
{
        t_path * courant;
        int i;
        chemin->height = racine->height ;
        if (racine->nbfils==0)
        {
                chemin->next=______;
                _______ chemin;
        }
        courant=chemin;
        for (i = 0; i < (racine->nbfils); i++)
        {
                courant->next = cree_chemin(____________ ->height);
                courant=courant->next;
                courant=contour_process(racine->______,________);
                courant->next= ___________(racine->height);
                courant=courant->next;
        }
        return _______;
}
```

L'argument `racine` *donne l'adresse de la racine de l'arbre, l'argument* `chemin` *donne l'adresse du chemin vide à partir duquel est construit le contour de l'arbre.*
Comprendre, compléter et implémenter cette fonction.

8 *Puisque les commandes de création utilisent* `malloc`*, il faut prévoir une commande qui vide (avec* `free`*) les arbres après utilisation ou élimine des sous-arbres d'un arbre. Compléter le code de la la fonction* `vide_descendance` *qui détruit* toute *la descendance stricte d'un nœud* `individu` *et qui a le prototype suivant :*

```
void vide_descendance( t_noeud *  individu)
```

On écrira cette fonction de manière récursive en s'inspirant de la question **??**.

9 *Écrire un programme qui testera ces fonctions en écrivant dans deux fichiers distincts les contours obtenus à partir d'un arbre n-aire régulier et d'un arbre de Galton-Watson aléatoire de paramètres* $p_1 = 1/2$ *et* $p_2 = 1/4$ *(de taille au moins 100 pour avoir un beau dessin : pour cela régénérer l'arbre autant que nécessaire et vider l'arbre précédent pour ne pas encombrer la mémoire).*

On pourra visualiser le contour obtenu avec le logiciel *gnuplot* en tapant `gnuplot` dans l'invite de commande puis en tapant `plot "nom_du_fichier" with lines`. On peut sortir de ce programme en tapant `exit`.

10 *On veut à présent utiliser le champ* `value` *laissé libre jusqu'à présent pour décrire des marches aléatoires branchantes. Le champ* `value` *d'un nœud est obtenu en ajoutant* $+1$ *ou* -1 *avec probabilité* $1/2$ *à la valeur* `value` *du nœud père indépendamment des autres enfants. Écrire une fonction de prototype*

```
void marches_branchantes( t_noeud * racine, int n0)
```

qui remplit les champs `value` *d'un arbre avec ce processus initialisé à la valeur* `n0` *pour la racine.*
Bonus : *écrire une fonction récursive d'élagage qui détruit toute la descendance d'un nœud dès que son champ* `value` *est négatif.*

11 ***(à faire chez soi)*** *Écrire la fonction* `int tempslocal(t_path * chemin,int h)` *qui compte le nombre de points d'un chemin qui ont une hauteur* `h` *donnée avant que le chemin ne se termine ou bien qu'il ne touche la hauteur* `(chemin->height)-1`*. Utiliser cette fonction pour reconstruire un arbre à partir d'un chemin de Dyck donné en argument (réciproque de la fonction qui donne le processus de contour).*