

12206



By  
Daniel  
Dalati

2023

---

### **Exercice 1 :**

Ecrire un programme qui permet de gérer les notes des étudiants :

- La liste contient l'identifiant et la note de chaque étudiant.
- Remplir la liste avec des éléments de telle sorte à ce que les notes soient triées dans l'ordre croissant . L'utilisateur choisira le nombre d'étudiant à faire rentrer.

**Indice :** Utiliser la fonction d'initialisation à 0 , d'ajout au début tous les éléments et de suppression à la fin pour supprimer l'initialisation.

**Exemple :** 1/8 --> 16/13 --> 24/15 --> 19/15,25 --> 7/17,50 --> 36/18 -->

- Afficher le nombre d'étudiant .
- Afficher la liste .
- Ecrire une fonction qui permet d'ajouter une valeur dans la liste de telle sorte à ce qu'elle reste triée . L'identifiant et la note de l'étudiant est choisi par l'utilisateur .
- Afficher la nouvelle liste .
- Afficher le nouveau nombre d'étudiant .
- Ecrire une fonction qui permet de calculer la moyenne des notes de la classe .

### **Correction :**

#### **Exercice 1 :**

```
#include <stdio.h>
#include <stdlib.h>

typedef struct infos{
    int ID;
    float note;
}infos;

typedef struct element{
    infos data;
    struct element *suivant;
}element;

typedef struct liste{
    element *tete;
}liste;

void creer_liste(liste *l){
```

```
l->tete=NULL;
```

```
}
```

```
void ajouter_tete(liste*l,int id,float note){  
    element *e=(element *)malloc(sizeof(element));  
    e->data.ID=id;  
    e->data.note=note;  
    e->suivant=l->tete;  
    l->tete=e;  
}
```

```
void supp_tete(liste *l){  
    if(l->tete==NULL) return;  
    element *r=l->tete;  
    l->tete=l->tete->suivant;  
    free(r);  
    r=NULL;  
}
```

```
void supp_fin(liste*l){  
    if(l->tete==NULL) return;  
    if(l->tete->suivant==NULL) {supp_tete(l); return;}  
    element *e=l->tete;  
    while(e->suivant->suivant!=NULL){  
        e=e->suivant;  
    }  
    free(e->suivant);  
    e->suivant=NULL;  
}
```

```
void affiche_info(infos d){
    printf("(%d/.2f)\t",d.ID,d.note);
}

void affichage(liste *l){
    element *p=l->tete;
    while(p!=NULL){
        affiche_info(p->data);
        p=p->suivant;
    }
}

void nombre_etudiant(liste *l){
    int nb=0;
    element *p=l->tete;
    while(p!=NULL){
        nb++;
        p=p->suivant;
    }
    printf("\nle nb des etudiant est :%d\n",nb);
}
/*
void ajout_obj(liste *l,int x,float y){
    element *h=(element*)malloc(sizeof(element));
    element *p=l->tete;
    h->data.ID=x;
    h->data.note=y;
    while(r->suivant->data.note<y){
        r=r->suivant;
        h->suivant=r->suivant;
        r->suivant=h;
    }
}
```

```

}*/



void ajout_obj(liste *l,int x,float y){

element *h=(element*)malloc(sizeof(element));

element *p=l->tete;

h->data.ID=x;

h->data.note=y;

while((p->suivant->data.note)<y) {p=p->suivant; }

h->suivant=p->suivant;

p->suivant=h;

}

void moyenne(liste *l){

float s,m;

int n=0;

element *e=l->tete;

while(e!=NULL){

s=s+e->data.note;

e=e->suivant;

n++;

}

m=s/n;

printf("la moyenne des notes de la classe est : %.2f\n",m);

}

void main()

{

liste L1;

int n,i,a;

float b;

creer_liste(&L1);

printf("saisir le nb des etudiant :");

```

```
scanf("%d",&n);
printf("\n");
for(i=0;i<n;i++){
    printf("\Saisir l'identifiant d'une etudiant num %d : ",i+1);
    scanf("%d",&a);
    printf("\nSaisir le note d'une etudiant num %d : ",i+1);
    scanf("%f",&b);
    printf("-----\n");
    ajouter_tete(&L1,a,b);
}
```

```
nombre_etudiant(&L1);
affichage(&L1);
printf("\n\n-----\n\n");
//supp_fin(&L1);
nombre_etudiant(&L1);
//affichage(&L1);
printf("\nSaisir l'identifiant d'un nouveau etudiant : ");
scanf("%d",&a);
printf("\nSaisir le note d'un nouveau etudiant num : ");
scanf("%f",&b);
ajout_obj(&L1,a,b);
affichage(&L1);
moyenne(&L1);
affichage(&L1);
}
```

---

## Exercice 1 :

Dans le but de classifier et exploiter des informations d'un groupe de personne, il faut utiliser la notion d'arbre de recherche qui gère les années de naissances de personnes. L'année 2000 étant la racine, toutes les années qui lui sont supérieures se présentent à sa droite.

Chaque personne est caractérisée par son **année de naissance**, son **initial de nom**, son **initial de prénom** et la **sexé** c'est-à-dire homme « **H** » ou femme « **F** ».

Exemple : « 1995 A O F »

Ecrire un programme qui permet de :

- ❖ Ajouter l'année de naissance 2000 en mettant comme de nom 'O' et de prénom 'A', et sexe 'H'.
- ❖ Ajouter dix informations choisi par le programmeur .
- ❖ Afficher dans l'ordre croissant les éléments de l'arbre.
- ❖ Calculer le nombre de personne dans l'arbre.
- ❖ Calculer nombre d'hommes et de femmes dans l'arbre.

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
typedef struct Infos{  
    int anne;  
    char nom;  
    char prenom;  
    char sexe;  
}Infos;
```

```
typedef struct noeud{  
    Infos data;  
    struct noeud *droite;  
    struct noeud *gauche;  
}noeud;  
typedef struct arbre{  
    noeud *racine;  
}arbre;
```

```

arbre creer_arbre(){
arbre*a=(arbre*)malloc(sizeof(arbre));
a->racine=NULL;
return *a;
}

noeud* creer_feuille(Infos d){
noeud *n=(noeud*)malloc(sizeof(noeud));

n->data=d;

n->gauche=NULL;
n->droite=NULL;
return n;
}

void affich_info(Infos d){
printf("(%d-%c-%c-%c)\t",d.anne,d.nom,d.prenom,d.sex);
}

void afficheCros(noeud *n){
if(n->gauche!=NULL)
afficheCros(n->gauche);
affich_info(n->data);
if(n->droite!=NULL)
afficheCros(n->droite);

}

```

```

void affiche(arbre a){
    if(a.racine!=NULL)
        afficheCROIS(a.racine);
}

int comparer(Infos a1, Infos a2){
    if(a1.anne<a2.anne) return -1;
    if(a1.anne==a2.anne) return 0;
    return 1;
}

void AjouterR(noeud *n , Infos v){
    if(comparer(n->data,v)>0){
        if(n->gauche==NULL)
            n->gauche=creer_feuille(v);
        else AjouterR(n->gauche,v);
    }
    else{
        if(n->droite==NULL)
            n->droite=creer_feuille(v);
        else AjouterR(n->droite,v);
    }
}

void ajouter_main(arbre *a, Infos v){
    if(a->racine==NULL)
        a->racine=creer_feuille(v);
    else AjouterR(a->racine,v);
}

```

```

int compterNoeud(noeud*n){

if(n->gauche==NULL)

    if(n->droite==NULL)

        return 1;

    else 1+compterNoeud(n->droite);

else if(n->droite==NULL)

    return 1+ compterNoeud(n->gauche);

else return 1+compterNoeud(n->gauche)+compterNoeud(n->droite);

}

int compterPersonne(arbre a){

if(a.racine==NULL) return 0;

else return compterNoeud(a.racine);

}

int nbHommeR(noeud *n){

if(n->data.sex=="H") return 1+nbHommeR(n->gauche)+nbHommeR(n->droite);

if(n->gauche==NULL)

    return nbHommeR(n->droite);

else if(n->droite==NULL)

    return nbHommeR(n->gauche);

}

int nbHomme(arbre a){

if(a.racine==NULL)return 0;

return nbHommeR(a.racine);

```

```
}
```

```
int nbFemmeR(noeud *n){
```

```
if(n->data.sex=="F") return 1+nbFemmeR(n->droite)+nbFemmeR(n->gauche);
```

```
return nbFemmeR(n->droite)+nbFemmeR(n->gauche);
```

```
}
```

```
int nbFemme(arbre a){
```

```
if(a.racine==NULL) return 0;
```

```
return nbFemmeR(a.racine);
```

```
}
```

```
Infos LireInfo()
```

```
{
```

```
    Infos *d = (Infos*)malloc(sizeof(Infos));
```

```
    printf("donner anne :");
```

```
    scanf("%d",&d->anne);
```

```
    printf("donner nom :");
```

```
    scanf("%s",&d->nom);
```

```
    printf("donner prenom :");
```

```
    scanf("%s",&d->prenom);
```

```
    printf("donner sexe :");
```

```
    scanf("%s",&d->sexe);
```

```
    printf("\n-----\n");
```

```
    return *d;
```

```
}
```

```
void main()
```

```
{
```

```

int x,i,n;

arbre a=creer_arbre();

for(i=0;i<2;i++){
    ajouter_main(&a,LireInfo());
}
affiche(a);

printf("\n le nombres des personnes : %d ",compterPersonne(a));
// printf("\n le nombres des Homme : %d ",nbHomme(a));
// printf("\n le nombres des Femme : %d ",nbFemme(a));

}


```

---

## Exercice 2 :

Utilisez un arbre binaire de recherche afin de classifier les salaires d'un groupe d'employés. Chaque salarié est caractérisé par son salaire, son initial de nom, initial de prénom et son année d'entrée en entreprise. La classification dans l'arbre se fait par le salaire.

**Exemple :** Si la racine a un salaire de 10.000, tous les salaires qui lui sont inférieurs se présentent à gauche et tous les salaires qui lui sont supérieurs se présentent à droite.

Ecrire un programme qui permet de :

- Initialiser l'arbre par des informations choisies par l'utilisateur.
- Afficher l'arbre.
- Remplir l'arbre. L'utilisateur choisira le nombre d'employés à ajouter et les informations à rentrer .
- Afficher dans l'ordre croissant l'arbre .
- Calculer et afficher le nombre d'employés qui sont entrés en entreprise avant ou égal à l'an 2000.
- Calculer le nombre de salariés.

```

#include <stdio.h>
#include <stdlib.h>
typedef struct infos{
    float salaire;
    char nom[20];
}
```

```
char prenom[20];
int annee;
}infos;

typedef struct noeud{
infos data;
struct noeud*gauche;
struct noeud *droite;
}noeud;

typedef struct arbre{
noeud *racine;
}arbre;

arbre creer_arbre(){
arbre *a=(arbre*)malloc(sizeof(arbre));
a->racine=NULL;
return *a;
}

noeud *creer_feuille(infos d){
noeud *n=(noeud*)malloc(sizeof(noeud));
n->data=d;
n->droite=NULL;
n->gauche=NULL;
return n;
}

void affich_info(infos d){
printf("%s/%s/.2f/%d)\n",d.nom,d.prenom,d.salaire,d.annee);
}
```

```
void affichRec(noeud *n){  
    if(n!=NULL){  
        affichRec(n->gauche);  
        affich_info(n->data);  
        affichRec(n->droite);  
    }  
  
}  
  
void afficher(arbre a){  
    if(a.racine!=NULL)  
        affichRec(a.racine);  
    }  
  
int comparer(infos a1,infos a2){  
    if(a1.salaire<a2.salaire) return -1;  
    if(a1.salaire==a2.salaire) return 0;  
    return 1;  
  
}  
  
void ajouter(noeud *n,infos v){  
    if(comparer(n->data,v)>0)  
    {  
        if(n->gauche==NULL) n->gauche=creer_feuille(v);  
        else ajouter(n->gauche,v);  
    }  
    else {  
  
        if(n->droite==NULL) n->droite=creer_feuille(v);  
        else ajouter(n->droite,v);  
    }  
}
```

```
}

void ajouter_main(arbre *a,infos v){
    if(a->racine==NULL) a->racine=creer_feuille(v);
    else ajouter(a->racine,v);
}
```

```
infos Lire_infos()
{
    infos *d=(infos*)malloc(sizeof(infos));
    printf("donner le nom :");
    scanf("%s",&d->nom);
    printf("donner le prenom :");
    scanf("%s",&d->prenom);
    printf("donner le salaire:");
    scanf("%f",&d->salaire);
    printf("donner l'annee :");
    scanf("%d",&d->annee);
    printf("\n");
    return *d;
}
```

```
/*
int calculer(noeud *n){
    if(n->data.annee<=2000)
    {
        if(n->droite==NULL)
            if(n->gauche==NULL)
                return 1;
            else return 1+calculer(n->gauche);
        else if(n->gauche==NULL)
```

```
    return 1+calculer(n->droite);

else return 1+calculer(n->droite)+calculer(n->gauche);

}

else

{

if(n->droite==NULL)

return calculer(n->gauche);

if(n->gauche==NULL)

return calculer(n->droite);

return calculer(n->droite)+calculer(n->gauche);

}

}

*/
```

```
int calculer(noeud *n){

if(n==NULL) return 0;

else

{



if(n->data.annee<=2000)

return 1+calculer(n->droite)+calculer(n->gauche);





return calculer(n->droite)+calculer(n->gauche);

}

}
```

```
int calcule_main(arbre a){

    return calculer(a.racine);

}

int compter(noeud *n){

    if(n==NULL) return 0;

    return 1+ compter(n->droite)+compter(n->gauche);

}

int compter_main(arbre a){

    if(a.racine==NULL) return 0;

    return compter(a.racine);

}

int main()

{

    arbre a=creer_arbre();

    for(int i=0;i<2;i++)

    {

        ajouter_main(&a,Lire_infos());

    }

    afficher(a);

    printf("\n-----\n");

    int d;

    d=calcule_main(a);

    printf("\nle nombre des personne avant 2000 :%d \n ",d);

    printf("\nle nombre des personne total :%d \n\n",compter_main(a));

    return 0;

}
```

---



On souhaite implémenter quelques opérations sur les fractions. Une **Lfraction** est formée d'une liste de fractions triées d'une manière décroissante selon les valeurs des numérateurs (après simplification) des fractions. Une fraction est caractérisée par deux entiers : un numérateur (**num**) et un dénominateur (**denom**). On peut utiliser la méthode **pgcd(int a, int b)** pour simplifier la fraction. Les nouvelles valeurs de **num** et **denom**, de la fraction, seront prises en considération.

### **Partie I :**

1. Définir le type **Fraction**.
2. Ecrire la méthode monôme **creerFraction (int n, int d)** qui permet de créer une fraction dont le numérateur est **n** et le dénominateur est **d**. cette méthode retourne le pointeur sur la fraction créée.
3. Ecrire la méthode **evaluer(Fraction f)** qui permet de calculer et de retourner la valeur de la fraction.
4. Ecrire la méthode **additionner(Fraction f1, Fraction f2)** qui permet d'additionner les deux fractions **f1** et **f2** et retourne la fraction résultante.
5. Ecrire la méthode **multiplier(Fraction\* f1, Fraction f2)** qui permet de multiplier les deux fractions **f1** et **f2** et mettre la fraction résultante dans **f1**.
6. Ecrire la méthode **afficher(Fraction f)** qui permet d'afficher la fraction **m** sous la forme **num / denom**.
7. Ecrire la méthode **compare(Fraction f1, Fraction f2)** qui prend en paramètre deux fractions **f1** et **f2** et qui retourne :
  - a. **0** si les valeurs de deux fractions sont égales.
  - b. **1** si la valeur de la fraction **f1** est plus grande que la valeur de la fraction **f2**.
  - c. **-1** si la valeur de la fraction **f1** est plus petite que la valeur de la fraction **f2**.

### **Partie II**

Dans cette deuxième partie, on souhaite traiter le type **LFraction** : définition et opérations sur les fractions en basant sur le type **Fraction** et ses méthodes définies dans la partie I.

1. Définir le type **LFraction** en permettant d'accéder à la fraction précédente à partir de la fraction courante.
2. Définir la méthode *ajoutFraction(LFraction\*l, Fraction f)* qui permet d'ajouter une fraction à **LFraction** en respectant l'ordre des fractions (selon les valeurs). Il faut étudier tous le cas possibles (Dans le cas où une fraction existe déjà, selon sa valeur, on multiplie la fraction existante par 2).
3. Définir la méthode *lireLFraction(LFraction \* l)* qui permet de saisir la **LFraction** en ajoutant plusieurs fractions. Dans le cas où un exposant existe déjà, il ne faut pas créer un nouveau monôme mais il faut le signaler en affichant un message.
4. Définir la méthode *afficheLFraction(LFraction l)* qui affiche la LFraction passée en paramètre.
5. Définir la méthode *additionLfraction(LFraction l1, LFraction l2)* qui effectue l'addition de deux **LFractions** passés en paramètre et retourne la LFraction résultatante.
6. Définir la méthode *multiplicationLfraction(LFraction l1, LFraction l2)* qui effectue l'addition de deux **LFractions** passés en paramètre et retourne la LFraction résultatante.

*Bonne chance*

On souhaite implémenter quelques opérations sur les polynomes. Un **polynome** est formé d'une liste de monomes triés d'une manière décroissante selon les exposants des monomes. Un monome est caractérisé par un entier (exposant) et un réel (coefficients).

### Exercice 1 :

1. Définir le type **monome**.
2. Ecrire la méthode monôme **creerMonome (int e, float c)** qui permet de créer un monome d'exposant **e** et de coefficient **c**. Cette méthode retourne un pointeur sur le monome créé.
3. Ecrire la méthode **evaluer(Monome m, float x)** qui permet de calculer et de retourner la valeur du monome **m** en remplaçant **x** par sa valeur.
4. Ecrire la méthode **additionner(Monome m1, Monome m2)** qui permet d'ajouter les deux monomes **m1** et **m2** et retourne le monome résultant.
5. Ecrire la méthode **multiplier(Monome f1, Monome f2)** qui permet de multiplier les deux monomes **m1** et **m2** et mettre le monome résultant dans **m1**.
6. Ecrire la méthode **afficher(Monome m)** qui permet d'afficher le monome **m** sous la forme **coef\*x^exp**.
7. Ecrire la méthode **compare(Monome m1, Monome m2)** qui prend en paramètre deux monomes **m1** et **m2** et qui retourne :
  - a. **0** si les deux monomes ont le même exposant.
  - b. **1** si l'exposant du monome **m1** est plus grand que l'exposant du monome **m2**.
  - c. **1** si l'exposant du monome **m1** est plus petit que l'exposant du monome **m2**.

### Exercice 2 :

Dans cette deuxième partie, on va traiter le type **polynôme** : définition et opérations sur les polynômes en basant sur le type monôme et ses méthodes définies dans l'**exercice 1**.

1. Définir le type **polynôme** en permettant d'accéder au monôme précédent à partir du monôme actuel.
2. une méthode **ajoutMonome** qui permet d'ajouter un monôme au polynôme en respectant l'ordre des monomes. Il faut étudier tous les cas possibles (Dans le

cas où un exposant existe déjà on doit modifier seulement le coefficient).

3. une méthode ***lirePolynome*** qui permet de saisir le polynôme en ajoutant plusieurs monômes. Dans le cas où un exposant existe déjà, il ne faut pas créer un nouveau monôme mais il faut le signaler en affichant un message.
4. une méthode ***affichePolynome*** qui affiche un polynôme sous la forme:

$$\text{coef}_n * x^n + \text{coef}_{n-1} * x^{n-1} + \dots + \text{coef}_1 * x + \text{coef}_0$$

**exemple : n=3, P(x) = 4\*x^3 - 7 \* x + 1**

5. une méthode ***additionPoly*** qui effectue l'addition de deux polynômes passés en paramètre et retourne le polynôme résultat.
6. une méthode ***multiplicationPoly*** qui permet de multiplier deux polynômes et retourne le polynôme résultat.
7. une méthode ***deriveePoly*** qui calcule la dérivée d'un polynôme passé en paramètre et retourne le polynôme résultat.
8. Dans la méthode ***main***, créer les deux polynomes P et Q et appliquer les méthodes utilisées ci-dessous.

$$P(x) = 4.0 * x^3 - 7.0 * x + 1.0$$

$$Q(x) = 3.0 * x^2 + 7.0 * x + 2.0$$

On s'intéresse aux opérations suivantes :

–Addition de 2 polynômes :  $T(x) = P(x) + Q(x)$ ,

$$T(x) = 4 * x^3 + 3 * x^2 + 3$$

–Multiplication de 2 polynômes :  $T(x) = P(x) * Q(x)$ ,

$$T(x) = 12 * x^5 + 28 * x^4 - 13 * x^3 - 46 * x^2 - 7 * x + 2$$

–Dérivée d'un polynôme :  $P'(x) = 12 * x^2 - 7$

*Bonne chance*

Nous souhaitons mettre en œuvre une structure permettant de représenter et d'évaluer une expression arithmétique bien écrite (correcte). Nous convenons d'appeler "expression arithmétique" toute chaîne de caractères construite à partir de la liste des symboles 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, \*, /. Les expressions considérées ne mettront en jeu que des entiers relatifs (seulement des chiffres).

**Exemple :**

- 1- régulière bien-écrite (correcte):  $3 * 4 + 2 * 7 - 5$
- 2- régulière non bien-écrite (incorrecte):  $3 * 6 +$

Nous allons supposer que l'expression arithmétique (représentée par une chaîne de caractères) est écrite sous forme postfixée se terminant par un « . ».

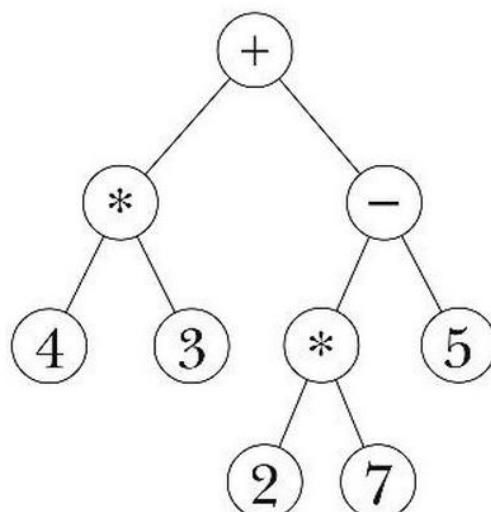
Afin de vérifier que l'expression arithmétique est bien écrite nous allons utiliser une pile de caractères. Pour vérifier qu'une expression arithmétique postfixée est bien écrite, il suffit:

- d'empiler les chiffres successifs que l'on rencontre.
- lorsque l'on détecte un opérateur, il faut dépiler les 2 derniers chiffres ;
- lorsque l'on rencontre un point (la fin de l'expression arithmétique) et la pile est vide nous concluons que l'expression est bien écrite et non dans le cas contraire.

Pour cela, nous supposons qu'on a les fonctions suivantes :

- **int estUnChiffre(char c)** indiquant si le caractère c est un chiffre.
- **int caractèreVersEntier(char c)** qui permet de convertir le caractère c en chiffre.
- **int estOpérateur(char c)** qui permet de tester si le caractère c est un opérateur(+,-,\*,/,%).

Finalement, une expression régulière bien-écrite sera représentée par un arbre binaire dont les nœuds représentent les opérations et les feuilles représentent les opérandes (chiffres). Ex : L'expression régulière bien-écrite  $(3 * 4 + 2 * 7 - 5)$  sera représentée par l'arbre suivant :



## Partie I (30pts)

1. Definir la structure Cellule et la structure Pile.

```
typedef struct Element {
    char car;
    struct Element *suiv;
} Element;

typedef struct Pile {
    Element *tete;
} Pile;
```

2. Definir la methode **creerPile()** qui permet de créer une pile vide et qui retourne un pointeur de type Pile.

```
Pile* creerPile()
{Pile*p=(Pile *) malloc(sizeof(Pile));
p->tete=NULL;
return p;
}
```

3. Definir la methode **empiler(Pile \*p, char c)** qui permet d'empiler le caractere c à la pile.

```
void empiler (Pile *p, char c){
    Element *elt;
    elt= (Element *) malloc(sizeof(Element));
    elt->car=c;
    elt->suiv=NULL;
    if(p->tete==NULL)
    {
        p->tete=elt;
        return;
    }

    elt->suiv = p->tete;
    p->tete = elt;
}
```

4. Definir la methode **depiler(Pile\*p)** qui permet de depiler (si c'est possible) le sommet de la pile et qui retourne le caractere depilé.

```
int depiler (Pile *p) {
    if(p->tete==NULL) return -1;
    Element*tmp=p->tete;
    p->tete=tmp->suiv;
    char c=tmp->car;
    free(tmp);
    return c;
}
```

5. Definir la methode **estVide(Pile P)** qui retourne 1 si la pile est vide et 0 sinon.

```
int estVide(Pile p)
{return (p.tete==NULL) ?1:0;}
```

6. Definir la methode **verifierExpression(char\*expr)** qui retourne 1 si la chaine expr représente bien une expression arithmetique en notation postfixée bien écrite.

```
//expr en format postfixé terminant par un point.
int verifierExpression (Char*expr){
Pile* p= creerPile();
int i, n=strlen(expr);
if(expr[n-1]!='.') return 0;
for(i=0;i<n-1;i++)
{
    if(expr[i]=='.') return 0;
    //Si expr[i] est un opérande, on empile expr[i]
    elseif(estOperande(expr[i])==1) empiler(p,expr[i]);
    //si expr[i] est un opérateur, on dépile les deux opérandes
    else if(estOperateur(expr[i])==1)
        { if(estVide(*p)!=1) depiler(p);
          else return 0;
          if(estVide(*p)!=1) depiler(p);
          else return 0;
        //on suppose que r est le resultat et On
        //l'empile
        empiler(p,'r') ;
        //ou
        // if(depiler(p)==-1) return 0;
        // if(depiler(p)==-1) return 0;
      }
    else return 0;
}
return p->tete->suiv==NULL;
}
```

## Partie II (70pts)

1. Definir la structure **Arbre** qui permet de représenter une expression arithmétique.

```
typedef struct Arbre {
    char info;
    struct Arbre * filsGauche;
    struct Arbre * filsDroite;
} Arbre;
```

2. Ecrire la méthode **creerArbre()** qui permet de créer un arbre vide et qui

retourne un pointeur de type **Arbre**.

```
Arbre* creerArbre () {
    Arbre *a = (Arbre *) malloc(sizeof(Arbre));
    return a;
}
```

3. Ecrire la méthode **creerNoeud(char e)** qui permet de créer un nœud (un arbre est formé des noeuds) en mémoire avec en **e** comme info et retournant un pointeur sur le nœud (Arbre) crée.

```
Arbre* creerNoeud (char e) {
    Arbre *a = (Arbre *) malloc(sizeof(Arbre));
    a->info=e;
    a-> filsGauche=NULL;
    a-> filsDroite=NULL;
    return a;
}
```

4. Écrire une fonction **nombreOperations(Arbre\*racine)** prenant un arbre binaire représentant une expression arithmétique et retourner le nombre de ses opérations.

```
int nombreOperations( Arbre* a ){
    if(a==NULL) return 0;
    Arbre * fg= a->filsGauche;
    Arbre * fd= a->filsDroite;
    if(fg==NULL) return 0;
    if(fd==NULL) return 0;
    return 1+ nombreOperations(fg)+ nombreOperations(fd);
}
```

5. Ecrire la méthode **estFeuille(Arbre\*e)** qui retourne **1** si le nœud **e** passé en parametre est une feuille et **0** dans le cas contraire.

```
int estFeuille( Arbre* a ){
    return (a==NULL ||
            (a->filsGauche==NULL && a->filsDroite==NULL))?1:0;
}
```

6. Ecrire la méthode **afficheExpressionPostfixee(Arbre \*racine)** qui permet d'afficher l'expression arithmetique représentée par l'arbre passé en parametre en notaion postfixée.

```
void afficheExpressionPostfixee ( Arbre *a )
{
    if ( a != NULL ){
        afficheExpressionPostfixee(a->filsGauche);
        afficheExpressionPostfixee(a->filsDroite);
        printf(" %c ",a->info);
    }
}
```

7. Ecrire la méthode **afficheExpressionPrefixee(Arbre \*racine)** qui permet d'afficher l'expression arithmetique représentée par l'arbre passé en paramètre en notation prefixée.

```
void afficheExpressionPrefixee ( Arbre *a ) {
    if(a!=NULL) {
        if(!estFeuille(a))printf("("); //si on a un opérateur
        printf(" %c ",a->info);
        afficheExpressionPrefixee(a->filsGauche);
        afficheExpressionPrefixee(a->filsDroite);
        if(!estFeuille(a))printf(")"); //si on a un opérateur
    }
}
```

8. Ecrire la fonction **evaluation (Arbre\*racine)** qui calcule et retourne la valeur de l'expression représentée par l'arbre passé comme paramètre.

```
int evaluation(Arbre *a) {
    if(a!=NULL) {
        //Si on a un opérateur : on fait le calcul
        if(estOperateur(a->info))
            {
                //opérande gauche
                int opGa = evaluation(a->filsGauche);
                //opérande droite
                int opDr = evaluation(a->filsDroite);
                switch (a->info) {
                    case '+': return opGa + opDr;
                                break;
                    case '-': return opGa - opDr;
                                break;
                    case '*': return opGa * opDr;
                                break;
                    case '/': return opGa / opDr;
                                break;
                    case '%': return opGa % opDr;
                                break;
                }
            }
        //Si on a un opérande : on retourne sa valeur
        else {
            //l'info est de type char et en int à pour
            //valeur son code ASCII
            return caracVersEntier(a->info);
        }
    }
    return -1 ;
}
```

9. Ecrire la méthode **enregistrerArbre(Arbre \* racine, char\*nomFic)** qui permet d'enregistrer l'arbre passé en paramètre dans le fichier appelé nomFic. L'enregistrement d'un arbre revient à enregistrer l'expression arithmétique sous forme postfixée.

```

void enregistrerArbreIntermediaire (Arbre *a,FILE *f)
{
    if ( a != NULL ){
        enregistrerArbreIntermediaire(a->filsGauche,f);
        enregistrerArbreIntermediaire(a->filsDroite,f);
        fprintf(f, " %c ",a->info);
    }
}
void enregistrerArbre(Arbre *a,char*nomFic)
{
    FILE* fic=fopen(nomFic, "w");
    if(fic==NULL)
        {printf("Ouverture echoué !!!");
        return ;
    }
    enregistrerArbreIntermediaire(a,fic) ;
    fputc('. ,fic);
    fclose(fic);
}

```

10. Ecrire la fonction **chargerArbre(char\*nomFic)** qui permet de charger l'expression arithmetique sous forme postfixée enregistrée dans le fichier nomFich dans un arbre à retourner son pointeur.

```

//Dans cette reponse, on suppose que la pile est une
//pile d'arbres de caractères (chaque nœud dans
//l'arbre représente un caractère)

Arbre* chargerrArbre (char*nomFic)
{
    FILE* fic=fopen(nomFic, "r");
    if(fic==NULL)
        {printf("Ouverture echoué !!!");
        return NULL;
    }
    //n'importe quelle valeur assez grande il sera mieux
    //de définir une constante #define Max 1000 par exemple

    char expr[1000];
    fgets(expr,1000 ,fic) ;
    fclose(fic);

    Pile*p=creerPile();

```

```

int i,n=strlen(expr);
if(expr[n-1]!='.') return NULL;
for(i=0;i<n-1;i++)
{
    if(expr[i]=='.') return NULL;

//Si expr[i] est un opérande, on le met dans un
//arbre (nœud) et on l'empile.

else if(estOperande(expr[i]))
    empiler(p, creerNoeud(expr[i]));

//Si expr[i] est un opérateur, on dépile les deux premiers
//nœuds de la pile puis on empile le nœud résultant.
    else if(estOperateur(expr[i]))
    {
        // on dépile les deux éléments en haut de pile
        Arbre noeud1= depiler(p);
        if(noeud1==NULL) return NULL ;
        Arbre noeud2= depiler(p);
        if(noeud2==NULL) return NULL ;

        // Création de l'arbre résultant pour l'opérateur
        Arbre *res= creerNoeud(expr[i]);
        res->filsGauche=noeud2;
        res->filsDroite=noeud1 ;

        // On empile le nœud(arbre) résultant
        empile(p,res) ;
    }
else return NULL ;
}

// on vérifie que la pile contient un seul élément
// si c'est le cas, ça signifie que l'expression est
// bien écrite

if(p->tete->suiv !=NULL) return NULL ;

//le pointeur de l'arbre se trouve à la tête de la pile
return p->tete;

}

```

On souhaite implémenter quelques opérations sur les polynômes. Un polynôme sera représenté par un arbre binaire de recherche de monômes (selon l'exposant de monômes). Un monôme est caractérisé par un réel (coefficients) et un entier positif (exposant).

### Exercice 1 : (20points)

1. Définir le type **monome**.

```
typedef struct monome
{
    float coef;
    int exp;
} monome;
```

2. Ecrire la méthode **creerMonome(float coef, int exp)** qui permet de créer un monôme de degré **exp** et de coefficient **coef** et qui retourne le pointeur sur le monôme créé.

```
monome* creerMonome(float coef, int exp)
{monome *m=(monome*)malloc(sizeof(monome));
 m->coef=coef;
 m->exp=exp;
 return m;
}
```

3. Ecrire la méthode **evaluer(monome m, float x)** qui permet d'évaluer le monôme m.

```
float evaluer(monome m, float x)
{float s=1;
 int i;
 for(i=0;i<m.exp ;i++)
 s=s*x;
 return coef*s;
}
```

4. Ecrire la méthode **multiplier(monome \*m1, monome m2)** qui permet de multiplier les deux monômes m1 et m2 et mettre le résultat dans m1.

```
void multiplier(monome *m1, monome m2)
{if(m1==NULL)return;
 m1->coef=m1->coef*m2.coef;
 m1->exp=m1->exp+m2.exp;
}
```

5. Ecrire la méthode **compare** qui prend en paramètre deux monômes et qui retourne si l'exposant du premier monôme est plus grand (retourne 1), égal (retourne 0) ou plus petit (retourne -1) que celui du deuxième.

```
int comparer(monome m1, monome m2)
{if(m1.exp==m2.exp) return 0;
 if(m1.exp>m2.exp) return 1;
 return -1;
}
```

## Exercice 2 : (80points)

Dans cette deuxième partie, on va traiter le type **polynôme** : définition et opérations sur les polynômes en basant sur le type monôme et ses méthodes définies dans l'**exercice 1**.

Nous supposons également que les fonctions suivantes sont définies pour les monômes :

- **monome additionnerDeuxMonomes(monome m1, monome m2)** qui permet d'additionner les deux monômes m1 et m2 (si c'est possible) et retourne le monôme résultant (NULL dans le cas contraire).
- **monome deriveeMonome(monome m)** qui permet de calculer la dérivée du monôme passé en paramètre et qui retourne le monôme résultant.
- **void afficher(monome m)** qui permet d'afficher le monôme m sous la forme coef.x<sup>exp</sup>.

Nous souhaitons représenter un polynôme par un arbre binaire de recherche basé sur l'exposant de monôme.

1. Définir le type **Polynome**.

```
typedef struct noeud {  
    monome m ;  
    noeud* filsg ;  
    noeud* filsdl ;  
} noeud ;
```

```
typedef struct Polynome {  
    noeud* racine ;  
}
```

2. Définir la méthode **ajoutMonome(Polynome \*p, monome m)** qui permet d'ajouter un monôme au polynôme en respectant l'ordre des monômes. Il faut étudier tous les cas possibles (Dans le cas où un exposant existe déjà on doit modifier seulement le coefficient).

```
void ajouter(Polynome*p,monome m){  
    if(p==NULL) {p=creerPolynome(m);  
        return;  
    }  
    p->racine=ajouterRec(p->racine,m);  
}  
noeud* ajouterRec(noeud* n,monome m){  
    if(n==NULL){ Polynome *a= creerPolynome(m);  
        return a->racine;  
    }  
    if(compare(n->m,m)<0)  
        { n->filsg=ajouterRec(n->filsg,m); return n; }  
    if(compare (n->m,m)>0)  
        { n->filsd=ajouterRec(n->filsd,m);return n; }
```

```

n->m.coef=m.coef ;
return n;
}

```

3. Définir la méthode **lirePolynome** qui permet de saisir le polynôme en ajoutant plusieurs monômes. Dans le cas où un exposant existe déjà, il ne faut pas créer un nouveau monôme mais il faut le signaler en affichant un message.

```

Polynome lirePolynome(){
Polynome*p=(Polynome*)malloc(sizeof(Polynome));
int choix,exp;
float coef;
do{
printf("saisir un monome : \n");
printf("donner l'exposant : ");
scanf("%d",&exp);
printf("donner le coefficient : ");
scanf("%f",&coef);
ajoutMonome(p,creerMonome(exp,coef)); //en fait, une version modifiée de la
//fonction ajoutMonome.
printf("Donner votre choix (0 :s'arreter)");
scanf("%d",&choix);
}
while(choix!=0);
}

```

4. Définir la méthode **affichePolynome** qui affiche un polynôme sous la forme:

$$\text{coef}_n * x^n + \text{coef}_{n-1} * x^{n-1} + \dots + \text{coef}_1 * x + \text{coef}_0$$

**exemple :**  $n=3, P(x) = 4*x^3 - 7 * x + 1$

```

void affichePolynome(Polynome p)
{ affichePolynomeRec(p.racine);
}
Void affichePolynomeRec(nœud*n)
{if(n==NULL)return ;
affichePolynomeRec(n-<filsg);
if(n-<filsg!=NULL)printf("+");
afficher(n->m);
if(n-<filsd!=NULL)printf("+");
affichePolynomeRec(n-<filsd);
}

```

5. Définir la méthode **multiplicationPoly** qui permet de multiplier deux polynômes et retourne le polynôme résultat.

```

Polynome multiplicationPoly (Polynome p1,Polynome p2)
{Polynome* result= (Polynome*)malloc(sizeof(Polynome));
If(p1.racine==NULL || p2.racine==NULL) return result;
multiplicationPolyRec(p1.racine,p2.racine,result) ;

```

```

        return * result;
    }
    void multiplicationPolyRec(noeud *n1,noeud*n2, Polynome*result){
        if(n1==NULL) return;
        multiplier(n2,n1->m,result);
        multiplicationPolyRec(n1->fisg,n2,result);
        multiplicationPolyRec(n1->fisd,n2,result);
    }
    void multiplier(noeud*n, monome m, Polynome*result){
        if(n==NULL) return;
        multiplier(&m,n->m);
        ajoutMonome(result,m);
        multiplier(n->filsg,m,result);
        multiplier(n->filsd,m,result);
    }

```

6. Définir la méthode **deriveePoly** qui calcule la dérivée d'un polynôme passé en paramètre et retourne le polynôme résultat.

```

Polynome deriveePoly (Polynome p)
{Polynome* deriv= (Polynome*)malloc(sizeof(Polynome));
deriveePolynomeRec(p.racine, deriv) ;
return *deriv;
}
Void deriveePolynomeRec(nœud*n,Polynome * deriv)
{if(n==NULL) return ;
ajoutMonome(deriv, deriveeMonome(n->m));
deriveePolynomeRec (n->filsg,deriv);
deriveePolynomeRec (n->filsd,deriv);
}

```

7. Définir la méthode **degreePolynome** qui permet de retourner le degré du polynôme passé en paramètre.

```

int degreePolynome (Polynome p)
{ if(p.racine==NULL) return -1;
retutn degreePolynomeRec(p.racine, deriv) ;
}
int degreePolynomeRec (nœud*n)
{if(n->filsg==NULL) return n->m.exp;
return degreePolynomeRec (n->filsg);
}

```

8. Définir la méthode **enregistrerPolynome(Polynome p, char \*nomFichier)** qui permet d'enregistrer le polynôme passé en paramètre dans un fichier nomFichier.txt.
9. Définir la méthode **chargerPolynome(char \*nomFichier)** qui permet de charger le polynôme enregistré dans le fichier nomFichier et qui retourne le

polynome chargé.

10. Dans la méthode **main**, créer les deux polynomes P et Q et appliquer les méthodes utilisées ci-dessous.

$$P(x) = 4.0 * x^3 - 7.0 * x + 1.0$$
$$Q(x) = 3.0 * x^2 + 7.0 * x + 2.0$$

-Multiplication de 2 polynômes :  $T(x) = P(x) * Q(x)$ ,

$$T(x) = 12 * x^5 + 28 * x^4 - 13 * x^3 - 46 * x^2 - 7 * x + 2$$

-Dérivée d'un polynôme :  $P'(x) = 12 * x^2 - 7$

```
int main(){
    Polynome p=creerPolynome();
    ajoutMonome(&p,creerMonome(4,3));
    ajoutMonome(&p,creerMonome(1,0));
    ajoutMonome(&p,creerMonome(-7,1));
    afficherPoy(p);

    Polynome q=creerPolynome();
    ajoutMonome(&q,creerMonome(2,0));
    ajoutMonome(&q,creerMonome(3,2));
    ajoutMonome(&q,creerMonome(7,1));
    afficherPoy(q);

    Polynome t=multiplierPoly(p,q);
    afficherPoy(t);
    Polynome deriveP=deriveePoly(p);
    afficherPoy(deriveP);
    return 0;
}
```

Bonne chance

### مطلوب الإجابة على كل جزء بشكل متواصل ودون تداخل مع الجزء الآخر

On souhaite implémenter quelques opérations sur les fractions. Une **Tfraction** est un arbre de fractions triées d'une manière décroissante selon les valeurs des dénominateurs (après simplification) des fractions. Une **fraction** est caractérisée par deux entiers : un numérateur (**num**) et un dénominateur (**denom**). On peut utiliser la méthode **pgcd(int a, int b)** pour simplifier la fraction. Les nouvelles valeurs de **num** et **denom**, de la fraction, seront prises en considération.

### Partie P (45<sup>m</sup>):

- Définir le type **Fraction**.

```
typedef struct Fraction {
    int num;
    int denom;
} Fraction;
```

- Ecrire la méthode **Fraction creerFraction (int n, int d)** qui permet de créer une fraction dont le numérateur est **n** et le dénominateur est **d**. Cette méthode retourne le pointeur sur la fraction créée (après simplification) si la valeur de **d** est différente de zéro et **null** sinon.

```
Fraction creerFraction(int num,int denom ){
    Fraction *f= (Fraction *) malloc(sizeof(Fraction));
    f->num=num/pgcd(num,denom);
    f->denom=denom/pgcd(num,denom);
    return *f;
}
```

- Ecrire la méthode **evaluer(Fraction f)** qui permet de calculer et de retourner la valeur de la fraction.

```
float evaluer(Fraction f ){
    return ((float)a)/b;
}
```

- Ecrire la méthode **simplifier(Fraction\* f)** qui permet de calculer la forme simplifiée de la fraction **f**. Vous pouvez utiliser la fonction **int pgcd(int a, int b)** qui permet de calculer le pgcd de deux entiers **a** et **b**.

```
void simplifier(Fraction*f){
    int p=pgcd(f->num,f->denom);
    a->num=a->num/p;
    a->denom=denom/p;
}
```

5. Ecrire la méthode **additionner(Fraction f1, Fraction f2)** qui permet d'additionner les deux fractions **f1** et **f2** et retourne la fraction résultante.

```
Fraction additionner(Fraction f1, Fraction f2 ){
```

```
    Fraction *f = (Fraction *) malloc(sizeof(Fraction));
```

```
    f->num=f1.num*f2.denom+f1.denom*f2.num ;
```

```
    f->denom=f1.denom*f2.denom ;
```

```
    int p=pgcd(f->num,f->denom);
```

```
    f->num=f->num/p;
```

```
    f->denom=f->denom/p;
```



Equivalent à simplifier(f);

```
    return *f;
```

```
}
```

6. Ecrire la méthode **multiplier(Fraction \*f1, Fraction f2)** qui permet de multiplier les deux fractions **f1** et **f2** et mettre la fraction résultante dans **f1**.

```
void multiplier(Fraction *f1, Fraction f2 ){
```

```
    f1->num=f1->num*f2.num ;
```

```
    f1->denom=f1->denom*f2.denom;
```

```
    simplifier(f1);
```

```
}
```

7. Ecrire la méthode **afficher(Fraction f)** qui permet d'afficher la fraction **m** sous la forme **num / denom**.

```
void afficher(Fraction f){
```

```
    printf("%d / %d",f.num,f.denom);
```

```
}
```

8. Ecrire la méthode **comparer(Fraction f1,Fraction f2 )** qui prend en paramètre deux fractions **f1** et **f2** et qui retourne :

a. **0** si les valeurs de deux fractions sont égales.

b. **1** si la valeur de la fraction **f1** est plus grande que la valeur de la fraction **f2**.

c. **-1** si la valeur de la fraction **f1** est plus petite que la valeur de la fraction **f2**.

```
int comparer(Fraction f1,Fraction f2 ){
```

```
    if(evaluer(f1)==evaluer(f2)) return 0 ;
```

```
    if(evaluer(f1)>evaluer(f2)) return 1 ;
```

```
    return -1;
```

```
}
```

## Partie F : (1h45m)

Dans cette deuxième partie, on souhaite traiter le type **TFraction** : définition et opérations sur les fractions en basant sur le type fraction et ses méthodes définies dans la **Partie P**. Une **TFraction** est une somme de plusieurs fractions. Une **TFraction** sera représentée par un arbre binaire où chaque nœud représente une **fraction**.

1. Définir le type **Nœud** qui représente un nœud de fraction.

```

typedef struct Noeud {
    Fraction f;
    struct Noeud * filsGauche;
    struct Noeud * filsDroit;
} Noeud;

```

2. Définir le type **TFraction** qui permet de représenter une.

```

typedef struct TFraction
{ Noeud* racine;} TFraction;

```

3. Ecrire la méthode *ajoutFraction(TFraction t, Fraction f)* qui permet d'ajouter une fraction (simplifiée) à **TFraction** en respectant l'ordre décroissant des fractions (selon les valeurs des dénominateurs). Dans le cas où une fraction existe déjà, selon son dénominateur, on additionne les numérateurs. Si la somme obtenue est égale à 0, on doit supprimer cette fraction. Vous pouvez utiliser la fonction **supprimer(TFraction t, Fraction f)** qui supprime la fraction **f** de la **TFraction t**.

```

Noeud* ajouterRec(Noeud* n,Fraction f){
if(n==NULL){ n=(TFraction*)malloc(sizeof(TFraction));
    n->racine=(Noeud*) malloc(sizeof(Noeud));
    n->racine->f=f;
    return n->racine;
}
if(comparer(f,n->f)<0){n->filsGauche=ajouterRec(n->filsGauche,f);
    return n;
}
if(comparer(f,n->f)>0){n->filsDroite=ajouterRec(n->filsDroite,f);
    return n;
}
n->f.num+=f.num;
if(n->f.num==0){
    TFraction *t=(TFraction*) malloc(sizeof(TFraction));
    t->racine=n;
    supprimer(t,n->f) ;
    n=t->racine ;
    return n;
}
simplifier(&(n->f));
return n;
}
void ajoutFraction (TFraction*a,Fraction f){
if(a==NULL) {a=(TFraction*)malloc(sizeof(TFraction));
    a->racine=(Noeud*) malloc(sizeof(Noeud));
    a->racine->f=f;           return;
}
a->racine=ajouterRec(a->racine,f);
}

```

4. Ecrire la méthode ***lireTFraction(TFraction \* t)*** qui permet d'ajouter plusieurs fractions à Tfraction passée en paramètre. Dans le cas où un denominateur existe déjà, il ne faut pas créer une nouvelle fraction mais il faut le signaler en affichant un message.

```

Noeud* ajouterRec1(Noeud* n,Fraction f){
    if(n==NULL){ Noeud *a = (Noeud *) malloc(sizeof(Noeud));
        a->filsDroite=NULL;
        a->filsGauche=NULL;
        a->f=simplifier(f);
        return a;
    }
    if(comparer(f,n->f)<0){n->filsGauche=ajouterRec1(n->filsGauche,f);
        return n;
    }
    if(comparer(f,n->f)>0){n->filsDroite=ajouterRec1(n->filsDroite,f);
        return n;
    }
    printf("Ce denominateur existe, Merci !! ");
    return n;
}
void ajoutFraction1(TFraction*a,Fraction f){
    if(a.racine==NULL){
        a->racine=(Noeud*) malloc(sizeof(Noeud));
        a->racine->filsGauche=NULL;
        a->racine->filsDroite=NULL;
        a->racine->f=f;
        return;
    }
    a->racine=ajouterRec1(a->racine,f);
}

void lireTFraction(TFraction*f){
    int choix ,num,denom;
    do{printf("Donner votre choix (0 :arreter) : ");
        scanf("%d",&choix);
        if(choix !=0 ){
            printf("Donner le numerateur : ");
            scanf("%d",&num);
            printf("Donner le denominrateur : ");
            scanf("%d",&denom);
            ajoutFraction1(f,creerFraction(num,denom));
        }
    }while(choix !=0 );
}

```

5. Ecrire la méthode ***afficheTFraction(TFraction t)*** qui affiche la TFraction passée en paramètre (n’oubliez pas qu’une **TFraction** est une somme de ses fractions).

```
void afficherRec(Noeud*n){
    if(n==NULL) return;
    afficherRec(n->filsGauche);
    afficher(n->f) ;
    afficherRec(n->filsDroit);
}
void afficher(TFraction t){
    if(t==NULL) return ;
    afficherRec(t.racine);}
```

6. Ecrire la méthode ***additionner(TFraction t1, TFraction t2)*** qui effectue la somme de deux **TFractions** passées en paramètre et retourne la **TFraction** résultante.

```
void additionnerRec(Noeud*n, TFraction *t){
    if(n==NULL) return ;
    ajoutFraction(t,n->f);
    additionnerRec(n->filsGauche,t);
    additionnerRec(n->filsDroite,t);
}

Tfraction additionnerTFraction (TFraction t1,TFraction t2){
    TFraction *t=(TFraction*)malloc(sizeof(TFraction));
    t->racine=NULL ;
    additionnerRec(t1.racine,t) ;
    additionnerRec(t2.racine,t) ;
    return *t ;
}
```

7. Ecrire la méthode ***evaluer(TFraction t)*** qui permet de calculer la valeur de **TFraction t**. Notez que la valeur d’une TFraction est la somme de valeurs de ses fractions.

```
float evaluerRec (Noeud*n){
    if(n==NULL) return 0;
    return evaluer(n->f)+evaluerRec(n->filsGauche) +evaluerRec(n->filsDroite);
}
float evaluerTFraction (TFraction* t){
    if(t==NULL) return 0;
    return evaluerRec(t->racine);
}
```

8. Ecrire la méthode ***conversion(TFraction t)*** qui prend en paramètre une TFraction t et qui la convertit en Fraction. Cette fonction retourne la fraction résultante. Penser à utiliser la fonction **additionner(Fraction f1, Fraction f2)** définie dans la **Partie P**.

```

void conversionRec (Noeud*n, Fraction *f){
    if(n==NULL) return ;
    *f=additionner(f,n->f);
    conversionRec (n->filsGauche,f);
    conversionRec (n->filsDroite,f);
}
Fraction conversion (TFraction t){
    Fraction f=creerFraction(0,1);
    conversionRec(t.racine,&f) ;
    return f ;
}

```

9. Ecrire la méthode *saveTFraction(TFraction t, char\*fichier)* qui permet d'enregistrer la **TFraction** t dans le fichier. Notez que, l'enregistrement d'une **TFraction** revient à enregistrer ses fractions (i.e enregistrer le numérateur et le dénominateur).

```

void saveTFractionRec(Noeud*n, FILE*f){
    if(n==NULL) return ;
    saveTFractionRec (n->filsGauche,f);
    fprintf(f,"%d%d ",n->f.num,n->f.denom) ;
    saveTFractionRec (n->filsDroite,f);
}
void saveTFraction(TFraction t, char*fichier){
FILE*f=fopen(fichier,"w");
if(f==NULL){printf("Erreur d'ouverture!!");}
    return ;
}
saveTFractionRec(t.racine,f) ;
fclose(f) ;
}

```

10. Ecrire la méthode *loadTFraction(char\*fichier)* qui permet charger la **TFraction** enregistré dans le fichier et retourne la **TFraction** chargé.

```

TFraction loadTFraction ( char*fichier){
FILE*f=fopen(fichier,"r");
if(f==NULL){printf("Erreur d'ouverture!!");}
    return NULL;
}
int num,denom ;
TFraction*t=(TFraction*)malloc(sizeof(TFraction));
t->racine=NULL;
while( !feof(f)){
fscanf(f,"%d %d",&num,&denom);
ajoutFraction(t,creerFraction(num,denom));
}
fclose(f) ;
return t ;
}

```

Bonne chance

On souhaite concevoir un système qui permet d'automatiser les processus de gestion des notes des étudiants du Département Informatique à la Faculté des Sciences – Section III à Tripoli.

Un étudiant sera caractérisé par son **numéro** (identifiant), son **nom**, son **prénom**, son **année** d'étude actuelle et un tableau de 10 **notes** (on suppose que l'étudiant doit être inscrit dans 10 matières). Le département Informatique est caractérisé par la liste d'étudiants inscrits au département. Nous allons demander d'implémenter quelques méthodes qui permet la bonne gestion du département.

## **Partie I :**

1. Définir le type **Etudiant**.

```
typedef struct Etudiant{  
    int num ;  
    char*nom ;  
    char*prenom ;  
    int annee ;  
    float notes[10] ;  
}Etudiant ;
```

2. Ecrire la méthode **Etudiant creerEtudiant (int numero, char\*nom, char\*prenom, int annee)** qui permet de créer un étudiant dont les données nécessaires sont passées en paramètre. Les notes de l'étudiant sont initialisées à -1. Cette méthode retourne le pointeur sur l'étudiant créé.

```
Etudiant creerEtudiant(int numero, char*nom,char*prenom,int annee)  
{int i ;  
Etudiant*e=(Etudiant*)malloc(sizeof(Etudiant)) ;  
e->num=numero ;  
e->nom=nom;  
e->prenom=prenom;  
e->annee=annee ;  
for(i=0;i<10;i++)  
e->notes[i]=-1;  
return *e;  
}
```

3. Ecrire la méthode **donnerNote(Etudiant \*e, int m,float n)** qui permet de donner à l'étudiant e une note n sur la matière m. Notez que, une note est acceptable si elle comprise entre 0 et 100.

```
void donnerNote (Etudiant*e, int m,float n)  
{if((m>=0&&m<10)&&(n<=100&&n>=0)) e->notes[m]=n ;  
}
```

4. Ecrire la méthode moyenne(Etudiant e) qui permet de calculer et de retourner la moyenne de l'étudiant e. Notez que, la moyenne de l'étudiant sera calculée seulement pour les notes différentes de -1.

```
float moyenne (Etudiant e)
{int i,nb=0 ;
 float s=0;
for(i=0;i<10;i++)
if(e.notes[i]!=-1){
    s=s+e.notes[i];
    nb++;
}
if(nb==0) return -1;
return s/nb;
}
```

5. Ecrire la méthode afficher(Etudiant e) qui permet d'afficher l'étudiant e sous la forme suivante :

**Num :: Nom :: Prenom :: Annee :: Liste des notes :: Moyenne**

```
void afficher(Etudiant e)
{int i;
printf("%d ::%s ::%s ::%d :: ",e.num,e.nom,e.prenom,e.annee) ;
for(i=0;i<10;i++)
printf("%.1f ",e.notes[i]);
printf("%.1f",moyenne(e));
}
```

6. Ecrire la méthode **compareMoyenne(Etudiant e1, Etudiant e2)** qui prend en paramètre deux étudiants e1 et e2 et qui retourne :

- 0** si les deux étudiants, e1 et e2, ont la même moyenne.
- 1** si la moyenne de e1 est plus grande que celle de e2.
- 1** si la moyenne de e1 est plus petite que celle de e2.

```
int compareMoyenne (Etudiant e1, Etudiant e2)
{ float m1,m2 ;
m1=moyenne(e1) ;
m2=moyenne(e2) ;
if(m1>m2) return 1 ;
if(m1<m2) return -1 ;
return 0;
}
```

7. Ecrire la méthode **compare(Etudiant e1, Etudiant e2)** qui prend en paramètre deux étudiants e1 et e2 et qui retourne :

- 0** si les deux étudiants, e1 et e2, ont le même numéro.
- 1** si le numéro e1 est plus grand que le numéro de e2.
- 1** si le numéro e1 est plus petit que le numéro de e2.

```
int compare (Etudiant e1, Etudiant e2)
{ return (e1.num>e2.num)?1:( e1.num==e2.num?0:-1);
}
```

## **Partie F :**

Dans cette deuxième partie, on souhaite gérer les étudiants du département Informatique. Pour cela, nous allons demander de créer un nouveau type nommé

**DepInfo** qui permet de représenter le département informatique. Un **DepInfo** sera représenté par un arbre binaire où chaque nœud représente un **étudiant**.

1. Définir le type **Nœud** qui représente un nœud de l'arbre.

```
typedef struct Nœud{  
    Etudiant e ;  
    struct Noeud* filsg;  
    struct Noeud* filsdl;  
}Nœud ;
```

2. Définir le type **DepInfo** qui permet de représenter un département.

```
typedef struct DepInfo {  
    Noeud* racine;  
}DepInfo ;
```

3. Définir la méthode **creerDep()** qui permet de créer un nouveau département.

```
DepInfo creerDep() {  
    DepInfo * dep=(DepInfo*)malloc(sizeof(DepInfo));  
    dep->racine=NULL;  
    return *dep;  
}
```

4. Ecrire la méthode **ajouterEtudiant(DepInfo\* dep, Etudiant e)** qui permet d'inscrire l'étudiant **e** au département **dep** en respectant l'ordre croissant des numéros des étudiants. Dans le cas où un étudiant est déjà inscrit au département, le système affiche un message d'alerte.

```
Noeud* ajouterRec(Noeud* n,Etudiant e){  
    if(n==NULL){ n=(Noeud*)malloc(sizeof(Noeud));  
        n ->e=e;  
        n->filsg=NULL;  
        n->filsdl=NULL;  
        return n;  
    }  
    if(compare(e,n->e)<0){n->filsg=ajouterRec(n->filsg,e);  
        return n; }  
    if(compare(e,n->e)>0){n->filsdl=ajouterRec(n->filsdl,e);  
        return n;}  
    printf("L'étudiant est déjà inscrit!!");  
    return n;  
}  
void ajoutetEtudiant(DepInfo* dep,Etudiant e)  
{  
    if(dep==NULL) {dep=(DepInfo*)malloc(sizeof(DepInfo));  
        dep->racine=(Noeud*) malloc(sizeof(Noeud));  
        dep->racine->e=e;  
        dep->racine->filsg=NULL ;  
        dep->racine->filsdl=NULL;  
        return;  
    }  
    dep->racine=ajouterRec(dep->racine,e);
```

}

5. Ecrire la méthode ***afficheDepartement(DepInfo dep)*** qui affiche toutes les données de tous les étudiants du département **dep** passé en paramètre.

```
void afficherRec(Noeud*n){  
    if(n==NULL) return;  
    afficherRec(n->filsg);  
    afficher(n->e) ;  
    printf("\n" );  
    afficherRec(n->filsd);  
}  
void afficherDepartement(DepInfo *dep){  
    if(dep==NULL) return ;  
    afficherRec(dep->racine);}
```

6. Ecrire la méthode ***moyenneDep(DepInfo \*dep)*** qui permet de calculer la moyenne générale des étudiants du département **dep**.

```
void moyenneDepRec(Noeud*n, float *somme, int *nb){  
    if(n==NULL) return;  
    float m=moyenneEtudiant(n->e);  
    if(m>=0){  
        *somme=*somme+m;  
        *nb=*nb+1 ;  
    }  
    moyenneDepRec (n->filsg,somme,nb);  
    moyenneDepRec (n->filsd,somme,nb);  
}  
float moyenneDep(DepInfo *dep){  
    if(dep==NULL) return -1 ;  
    float somme=0;  
    int nb=0;  
    moyenneDepRec (dep->racine,&somme,&nb);  
    if(nb==0) return -1 ;  
    return somme/nb ;  
}
```

7. Ecrire la méthode ***afficheDepartementAnnee(DepInfo dep, int a)*** qui affiche toutes les données des étudiants inscrits à l'année **a** du département **dep** passé en paramètre.

```
void afficheDepartementAnneeRec (Noeud*n,int a){  
    if(n==NULL) return;  
    if(n->e.annee==a) {afficher(n->e); printf("\n");}  
    afficheDepartementAnneeRec (n->filsg,a);  
    afficheDepartementAnneeRec (n->filsd,a);  
}  
void afficheDepartementAnnee (DepInfo *dep,int a){  
    if(dep==NULL) return ;  
    afficheDepartementAnnee Rec(dep->racine,a);}
```

8. Ecrire la méthode **moyenneAnnee**(**DepInfo dep, int a**) qui permet de calculer la moyenne générale des étudiants de l'année **a** du département **dep**.

```

void moyenneAnneeRec (Noeud*n, int a,float *somme, int *nb){
    if(n==NULL) return;
    if(n->e.annee==a) {
        float m=moyenneEtudiant(n->e);
        if(m>=0){
            *somme=*somme+moyenneEtudiant(n->e);
            *nb=*nb+1 ;
        }
    }
moyenneAnneeRec (n->filsg,a,somme,nb);
moyenneAnneeRec (n->filsd,a,somme,nb);
}
float moyenneAnnee(DepInfo *dep, int a){
    if(dep==NULL) return -1 ;
    float somme=0;
    int nb=0;
    moyenneAnneeRec (dep->racine,a,&somme,&nb);
    if(nb==0) return -1 ;
    return somme/nb ;
}

```

9. Nous souhaitons afficher la liste des étudiants selon leurs années d'études. Proposer une solution qui permet de réaliser cette tache.

```

Noeud* ajouterAnneeRec(Noeud* n,Etudiant e){
    if(n==NULL){ n=(Noeud*) malloc(sizeof(Noeud));
        n->e=e;
        n->filsd=NULL;
        n->filsg=NULL;
        return n;
    }
    if(e.annee<=n->e.annee){n->filsg= ajouterAnneeRec (n->filsg,e);
        return n; }
    n->filsd= ajouterAnneeRec (n->filsd,e);
    return n;
}
void ajouterAnnee (DepInfo* dep,Etudiant e){
    if(dep==NULL) {dep=(DepInfo*)malloc(sizeof(DepInfo));
        dep->racine=(Noeud*) malloc(sizeof(Noeud));
        dep->racine->e=e;
        dep->racine ->filsg=NULL;
        dep->racine ->filsd=NULL;
        return;
    }
    dep->racine=ajouterAnneeRec(dep->racine,e);}

```

```

void copierRec(Nœud*source,DepInfo*depdestination){
if(source==NULL) return ;
ajouterAnnee(depdestination,source->e) ;
copierRec(source->filsg, depdestination) ;
copierRec(source->filsd, depdestination) ;
}
void copier(DepInfo *depsource,DepInfo *depdestination)
{if(depsource==NULL) return ;
copierRec(depsource->racine,depdestination) ;
}
void afficherEtudiantAnnee(DepInfo *dep)
{
if(dep==NULL) return;
DepInfo *depa=(DepInfo*)malloc(sizeof(DepInfo));
depa->racine=NULL;
copier(dep,depa) ;
afficherDepartement(depa) ;
}
Autre methode : on suppose qu'on a, par exemple, 5 année :
void afficherEtudiantAnneeSimple(DepInfo dep)
{int i;
for(i=1;i<=5;i++)
afficheDepartementAnnee(dep, i);
}

```

10. Proposer une solution qui permet d'obtenir la liste des étudiants du département informatique triée selon leurs moyennes.

```

Noeud* ajouterMoyenneRec(Noeud* n,Etudiant e){
if(n==NULL){ n=(Noeud*) malloc(sizeof(Noeud));
n->e=e;
n->filsd=NULL;
n->filsg=NULL;
return n;
}
if(compareMoyenne(e,n->e)<=0){n->filsg= ajouterMoyenneRec (n->filsg,e);
return n; }
n->filsd= ajouterMoyenneRec (n->filsd,e);
return n;
}
void ajouterMoyenne (DepInfo* dep,Etudiant e){
if(dep==NULL) {dep=(DepInfo*)malloc(sizeof(DepInfo));
dep->racine=(Noeud*) malloc(sizeof(Noeud));
dep->racine->e=e;
dep->racine->filsg=NULL;

```

```

        dep->racine->filsd=NULL;
        return;
    }
    dep->racine= ajouterMoyenneRec (dep->racine,e);
}
void copierMoyenneRec(Nœud*source,DepInfo*depdestination){
if(source==NULL) return ;
ajouterMoyenne(depdestination,source->e) ;
copierMoyenneRec (source->filsg, depdestination) ;
copierMoyenneRec (source->filsd, depdestination) ;
}
void copierMoyenne(DepInfo *depsource,DepInfo *depdestination)
{if(depsource==NULL) return ;
 copierMoyenneRec(depsource->racine,depdestination) ;
}
void afficherEtudiantMoyenne(DepInfo *dep)
{
if(dep==NULL) return;
DepInfo *depa=(DepInfo*)malloc(sizeof(DepInfo));
depa->racine=NULL;
copierMoyenne(dep,depa) ;
afficherDepartement(depa) ;
}

```

11. Ecrire la méthode ***saveDepartement(DepInfo dep, char\*fichier)*** qui permet d'enregistrer les informations des étudiants du département **dep** dans le **fichier**.

```

void saveDepartementRec (Nœud*n, FILE*f){
int i;
if(n==NULL) return ;
saveDepartementRec (n->filsg,f);
fprintf(f,"%d %s %s %d",n->e.num,n->e.nom,n->e.prenom,n->e.annee) ;
for(i=0;i<10;i++)
fprintf(f, "%f",n->e.notes[i]);
saveDepartementRec (n->filsd,f);
}
void saveDepartement (DepInfo dep, char*fichier){
FILE*f=fopen(fichier,"w");
if(f==NULL){printf("Erreur d'ouverture!!");}
return ;
saveDepartementRec(dep.racine,f) ;
fclose(f) ;
}

```

12. Ecrire la méthode ***loadDepartement (char\*fichier)*** qui permet de créer un nouveau département et de charger les étudiants à partir du **fichier** et de les ajouter au département créé.

```

DepInfo loadDepartement (char*fichier){
    FILE*f=fopen(fichier,"r");
    if(f==NULL){printf("Erreur d'ouverture!!");}
        return NULL; }

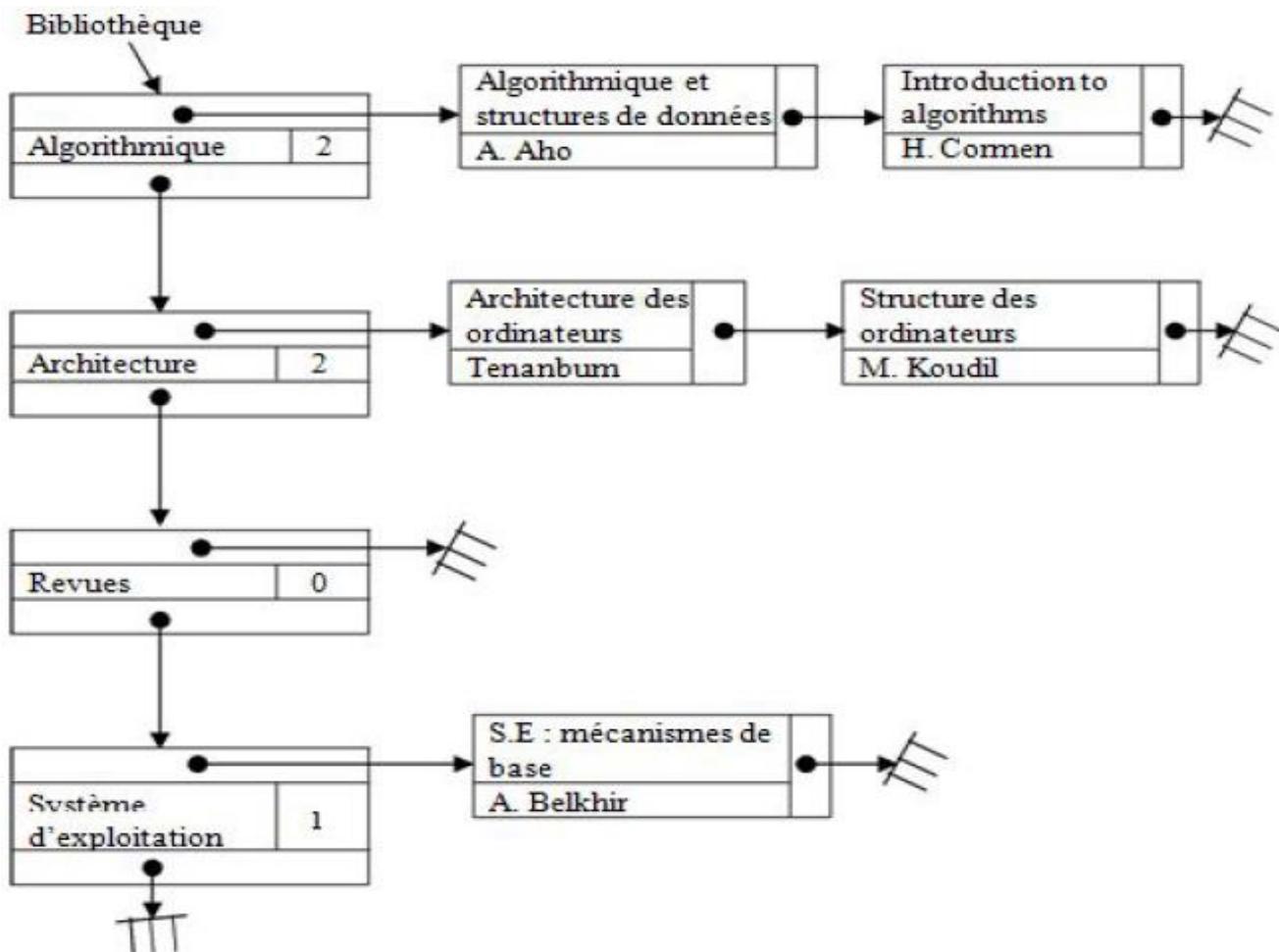
Etudiant e;
DepInfo *dep=( DepInfo *)malloc(sizeof(DepInfo));
dep->racine=NULL;
int num,annee,i;
float note;
char*nom,*prenom;
while( !feof(f)){
    fscanf(f,"%d %s %s %d ",&num,nom,prenom,&annee);
    e=creerEtudiant(num,nom,prenom,annee);
    for(i=0;i<10;i++)
        fscanf(f,"%f ", &note);
        donnerNote(&e,i,note);
        ajouterEtudiant(dep,e);
    }
    fclose(f) ;
    return dep ;
}

```

*Bonne chance*

### Exercice 1: (45 pts)

Dans cet exercice, on souhaite représenter la bibliothèque d'une ville donnée en utilisant une structure dynamique. La structure proposée est représentée sur la figure suivante :



### Structure de la bibliothèque

La liste verticale contient les catégories des livres avec le nombre de livres dans chacune, tandis que les listes horizontales contiennent les titres des livres avec leurs auteurs (pour simplifier le travail on suppose qu'on a un auteur par livre) dans chaque catégorie.

- a. Donner la déclaration des structures de données nécessaires (Bibliothèque, Catégorie et Livre) à l'implémentation de cette bibliothèque.

```

typedef struct Livre {
    char titre[200]; // char*titre ;
    char auteur[100] ; // char*auteur;
} Livre;

typedef struct CellLivre {
    Livre liv ;
    CellLivre *suiv;
} CellLivre;

typedef struct Categorie {
    int nbLivres ;
    char* nomCateg // char nomCateg[100]
    CellLivre *livres;
} Categorie;

typedef struct CellBiblio {
    Categorie categ ;
    CellBiblio *suiv;
} CellBiblio;

typedef struct Bibliotheque {
    CellBiblio * bib;
} Bibliotheque;

```

- b. Ecrire la fonction **ajouterCategorie(...)** qui permet d'ajouter une nouvelle catégorie à la bibliothèque (l'ajout se fait toujours à la fin de la liste).

```

void ajouterCategorie (Bibliotheque *b, char* c){
    CellBiblio*cb=(CellBiblio*)malloc(sizeof(CellBiblio));
    cb->categ=(Categorie *)malloc(sizeof(Categorie));
    cb->categ.nbLivres=0;
    Strcpy(Cb->categ.nomCateg,c);
    cb->categ.livres=NULL;
    cb->suiv=NULL ;

    if(b->bib=NULL)
    {
        bb->bib=cb ;
        return;
    }

    CellBiblio *tmp=b->bib;
    while(tmp->suiv!=NULL)
    {tmp=tmp->suiv;}
    tmp->suiv=cb;
}

```

- c. Ecrire la fonction **insererLivre(...)** qui permet l'insertion d'un nouveau livre à la bibliothèque (l'ajout se fait au début de la liste selon sa catégorie).

```
void insererLivre(Bibliotheque* b, Livre l, char* cat)
{
    CellBibio *tmp=b->bib;
    while(tmp!=NULL && strcmp(cat,tmp->categ.nomCateg)!=0)
    {tmp=tmp->suiv;
    }
    if(tmp ==NULL)
        ajouterCategorie (b,cat) ;
    tmp=b->bib;
    while(tmp!=NULL && strcmp(cat,tmp->categ.nomCateg)!=0)
    {tmp=tmp->suiv;
    }
    CellLivre* cl=( CellLivre *)malloc(sizeof(CellLivre));
    CellLivre->livre=liv ;
    CellLivre->suiv=tmp->bib ;
    tmp->bib=cellLivre ;
}
```

- d. Ecrire la fonction **afficherCategorie(...)** qui permet d'afficher les livres d'une catégorie donnée.

```
void afficherCategorie(Bibliotheque b, char* cat)
{
    CellBibio *tmp=b->bib;
    while(tmp!=NULL && strcmp(cat,tmp->categ.nomCateg)!=0)
    {tmp=tmp->suiv;
    }
    if(tmp ==NULL){printf("Categorie iconnue!!");
    return ;
    }
    CellLivre*cl=tmp->categ.livres;
    while(cl!=NULL)
    {afficherLivre(cl->liv);
    cl=cl->suiv;
    }
}
```

- e. Ecrire la fonction **nombreLivres(...)** qui retourne le nombre total de livre dans la bibliothèque.

```
int nombreLivres(Bibliotheque b)
{int cpt=0;
CellBibio *tmp=b->bib;
while(tmp!=NULL)
{cpt=cpt+tmp->categ.nbLivres;
tmp=tmp->suiv;
}
return cpt;
}
```

- f. Ecrire la fonction **supprimerCategorie(...)** qui permet de supprimer une catégorie avec tous ses livres.

```
void supprimerCategorie(Bibliotheque*b, char* cat)
{
    if(b->bib==NULL) return;
    if(strcmp(cat,b->bib.categ.nomCateg)==0)
        supprimerTete(b,cat);

    CellBiblio *tmp=b->bib;
    while(tmp->suv!=NULL &&
          strcmp(cat,tmp->suv->categ.nomCateg)!=0)
    {tmp=tmp->suv;
    }
    if(tmp->suv ==NULL) {printf("Categorie inconnue!!");
    return ;
    }

    CellBiblio*cl=tmp->suv ;
    tmp->suv=tmp->suv->suv ;
    free(cl) ;
}
```

- g. Ecrire la fonction **enregistrer(...)** qui permet d'enregistrer les informations disponibles dans la bibliothèque.

```
void enregistrer(Bibliotheque b, char* file)
{
    FILE*fic=fopen(file,"w+");
    if(fic==NULL){printf("Erreur d'ouverture");exit(0) ;

    int nCat=nbCategorie(b);
    fprintf(fic,"%d",nCat);
    CellBiblio *tmp=b.bib;
    while(tmp!=NULL)
    {
        fprintf(fic,"%s",tmp->categ.nomCateg);
        fprintf(fic,"%d",tmp->categ.nbLivres);

        CellLivre*cl=tmp->categ.livres;
        while(cl!=NULL)
        {
            fprintf(fic,"%s",tmp->cl.liv.titre);
            fprintf(fic,"%s",tmp->cl.liv.auteur);
            cl=cl->suv ;
        }
        tmp=tmp->suv;
    }
    fclose(fic);
```

## **Exercice 2:** (55 pts)

Dans cet exercice, nous allons travailler sur la liste des bibliothèques de chaque ville au Liban. Ces bibliothèques seront représentées par un arbre binaire de recherche (selon les noms des villes). Chaque nœud de cet arbre est composé des champs suivants :

- Le nom de la ville
- La bibliothèque de livres de cette ville

Définir les fonctions suivantes :

- a. Définir la structure **nœud** et la structure **arbre** nécessaires.

```
typedef struct noeud {
    char* nomVille ;
    Bibliotheque b ;
    struct noeud *filsG ;
    struct noeud *filsD ;
}

} noeud;

typedef struct Arbre {
    noeud* racine ;
}
```

- b. Définir la fonction **creerArbre()** qui retourne un pointeur sur l'arbre créé.

```
Arbre* creerArbre()
{
    Arbre *a=(Arbre*)malloc(sizeof(Arbre*)) ;
    a->racine=NULL;
    return a;
}
```

- c. Définir la fonction **afficheBibliothequeVille(...)** qui affiche la liste des livres de la bibliothèque d'une ville donnée (si la ville existe).

```
Void afficherBibliothequeVille(Arbre a, char*ville)
{
    Nœud*n=a->racine ;
    while(n !=NULL && strcmp(n->nomVille,ville) !=0)
        if{ strcmp(n->nomVille,ville)>0)n=n->filsG ;
        if{ strcmp(n->nomVille,ville)<0)n=n->filsD ;
    }
    if(n==NULL) return ;
    afficherBiblio(n->b) ;
}
```

- d. Définir la fonction **insererLivreVille(...)** qui permet d'ajouter un livre à la bibliothèque d'une ville donnée suivant sa catégorie.

```
Void insererLivreVille(Arbre* a,
    char*ville,char*categ,Livre l)
{
```

```

Nœud*n=a->racine ;
while(n !=NULL && strcmp(n->nomVille,ville) !=0)
    if{ strcmp(n->nomVille,ville)>0)n=n->filsg ;
    if{ strcmp(n->nomVille,ville)<0)n=n->filsd ;
    }
if(n==NULL) return ;
insererLivre(n->b,l,categ);
}

```

- e. Définir la fonction **afficheLivresCategorie(...)** qui permet d'afficher la liste de tous les livres d'une catégorie donnée dans toutes les bibliothèques du pays.

```

Void afficherLivresCategorie(Arbre a, char*categ)
{
afficherLivresCategorieRec(a.racine,categ) ;
}

Void afficherLivresCategorieRec(Nœud*n, char *cat)
{
if(n==NULL) return ;
afficherCategorie(n->b, cat) ;
afficherLivresCategorieRec(n->filsg,cat) ;
afficherLivresCategorieRec(n->filsd,cat) ;

}

```

- f. Définir la fonction **nombreLivres(...)** qui retourne le nombre de livres disponibles dans toutes les bibliothèques du pays.

```

int nombreLivres(Arbre a)
{
return nombreLivresRec(a.racine,categ) ;
}

int nombreLivresRec (Nœud*n, char *cat)
{
if(n==NULL) return 0 ;
return nombreLivres(n->b)+nombreLivresRec(n->filsg) +
        nombreLivresRec (n->filsd) ;
}

```

- g. Définir la fonction **laPlusgrandeBibliotheque(...)** qui retourne le nom de la ville dont sa bibliothèque possède le plus grand nombre de livres.

```

Char* laPlusgrandeBibliotheque(Arbre a)
{return laPlusgrandeBibliothequeRec(a->racine) ;}

```

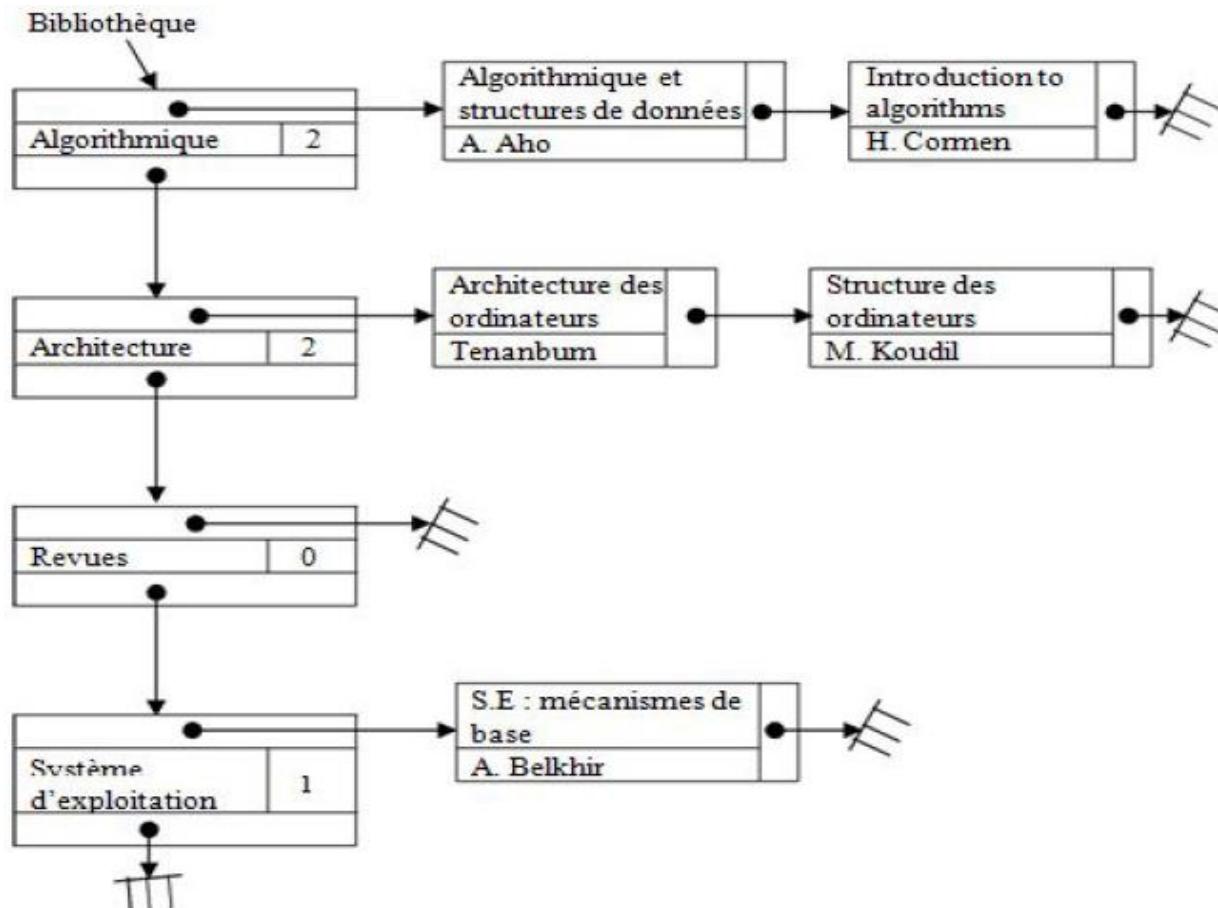
```
char* laPlusgrandeBibliothequeRec (Nœud*n) {  
    if (n==NULL) NULL ;  
}
```

**Vous pouvez utiliser les fonctions définies dans l'exercice précédent.**

*Bonne chance*

### **Exercice 1:** (45 pts)

Dans cet exercice, on souhaite représenter la bibliothèque d'une ville donnée en utilisant une structure dynamique. La structure proposée est représentée sur la figure suivante :



## **Structure de la bibliothèque**

La liste verticale contient les catégories des livres avec le nombre de livres dans chacune, tandis que les listes horizontales contiennent les titres des livres avec leurs auteurs (pour simplifier le travail on suppose qu'on a un auteur par livre) dans chaque catégorie.

- a. Donner la déclaration des structures de données nécessaires (**Livre**, **CellLivre**, **Catégorie**, **CellCatégorie** et **Bibliothèque**) à l'implémentation de cette bibliothèque.

- b. Ecrire la fonction **ajouterCategorie(Bibliotheque\* b, char\* nomCategorie)** qui permet d'ajouter une nouvelle catégorie à la bibliothèque (l'ajout se fait toujours à la fin de la liste).
- c. Ecrire la fonction **insererLivre(Bibliotheque\*b, char\* nomCategorie, Livre l)** qui permet l'insertion d'un nouveau livre à la bibliothèque (l'ajout se fait au début de la liste selon sa catégorie).
- d. Ecrire la fonction **afficherCategorie(Categorie c)** qui permet d'afficher les livres d'une catégorie donnée.
- e. Ecrire la fonction **nombreLivres(Bibliotheque b)** qui retourne le nombre total de livre dans la bibliothèque.
- f. Ecrire la fonction **supprimerCategorie(Bibliotheque\*b, char\*nomCategorie)** qui permet de supprimer une catégorie avec tous ses livres.
- g. Ecrire la fonction **enregistrer(Categorie c)** qui permet d'enregistrer les livres disponibles dans une catégorie donnée.

### **Exercice 2:** (55 pts)

Dans cet exercice, nous allons travailler sur la liste des bibliothèques de chaque ville au Liban. Ces bibliothèques seront représentées par un arbre binaire de recherche (selon les noms des villes). Chaque noeud de cet arbre est composé des champs suivants :

- Le nom de la ville
- La bibliothèque de livres de cette ville

Définir les fonctions suivantes :

- a. Définir la structure **Nœud** et la structure **Arbre** nécessaires.
- b. Définir la fonction **creerArbre()** qui retourne un pointeur sur l'arbre créé.
- c. Définir la fonction **afficheBibliothqueVille(Arbre a, char\*nomVille)** qui affiche la liste des livres de la bibliothèque d'une ville donnée (si la ville existe).
- d. Définir la fonction **insererLivreVille(Arbre\*a, char\*nomCategorie, Livre l)** qui permet d'ajouter un livre à la bibliothèque d'une ville donnée suivant sa catégorie.
- e. Définir la fonction **afficheLivresCategorie(Arbre a, char\*nomCategorie)** qui permet d'afficher la liste des livres d'une catégorie donnée dans toutes les bibliothèques du pays.
- f. Définir la fonction **nombreLivres(Arbre a)** qui retourne le nombre de livres disponibles dans toutes les bibliothèques du pays.

**Vous pouvez utiliser les fonctions définies dans l'exercice précédent et la fonction strcmp de la librairie string.h.**

***Bonne chance***

**Exercice 1** ( pts )

Soit la liste chaînée, représentant les données de personnes. Pour chacune d'elles on mémorise le nom et la taille de la personne



- 1) Définissez les types de données nécessaires pour définir un élément de la liste.

```

struct personne
{char nomP[15];
float taille;
structpersonne*next;};
  
```

- 2) Ecrivez une fonction qui permet de créer la liste chaînée des personnes **L1**. La saisie s'arrête lorsque l'utilisateur fournit le nom « vide » comme nom de la personne.

```

struct personne*saisie(){
    struct personne*P,*liste=NULL;
    char nom[15];
    do { printf("donner lenom de la personne et vide sinon:");
    scanf("%os",nom);
    if(strcmp(nom,"vide")==0) break;
    else
    {P=(structpersonne*)malloc(sizeof(structpersonne));
    strcpy(P->nomP,nom);
    printf("donner la taille de la personne:");
    scanf("%f",&P->taille);
    P->next=liste;
    liste=P;}
    }while(strcmp(nom,"vide")!=0);
    returnP;
  
```

```

void display(struct personne*liste){
    struct personne*p=liste;
    do{
        printf("Name of personne is %s\t size %f\n",p->nomP,p->taille);
        p=p->next;
    }
    while(p!=NULL);
}

```

- 3) Ecrivez une fonction qui reçoit la tête de la liste et renvoie la moyenne des tailles des personnes de la liste.

```

float moyenne(structpersonne*liste){
    structpersonne*P=liste;
    floatM=0;
    intn=0;
    while(P!=NULL)
    {M+=P->taille;
    P=P->next;
    n=n+1;}
    M=M/n;
    returnM;
}

```

- 4) Ecrivez une fonction qui à partir de la liste des personnes **L1**, on crée une deuxième liste **L2** telle que les personnes dont la taille est inférieure à la taille moyenne soient en tête de **L2**, suivis par les personnes dont la taille est supérieure à la taille moyenne.

```

struct personne*insertHead(structpersonne*L2,structpersonne*L1){
    structpersonne*temp=(structpersonne*)malloc(sizeof(structpersonne));
    strcpy(temp->nomP,L1->nomP);
    temp->taille=L1->taille;
    temp->next=L2;
    L2=temp;
    Return L2;
}
struct personne*insertTail(structpersonne*L2,structpersonne*L1){
    structpersonne*temp,*p=L2,temp=(structpersonne*)malloc(sizeof(structpersonne));

```

```

strcpy(temp->nomP,L1->nomP);
temp->taille=L1->taille;
temp->next=NULL;
if(L2==NULL)L2=temp;
else{while(p->next!=NULL)p=p->next;
p->next=temp;
}
Return L2;
}
struct personne*creation(structpersonne*L1){
struct personne*P=L1,*temp,*nouv,*L2=NULL;
floatt=moyenne(P);
while(P!=NULL)
{if(P->taille<t)L2=insertHead(L2,P);
elseL2=insertTail(L2,P);
P=P->next;
}
return
L2;
}

```

- 5) Ecrire le programme principal (fonction main) dans lequel vous
- Créez les deux listes **L1** et **L2**,
  - Affichez les tailles qui sont supérieures à la taille moyenne,
  - Affichez le nom de la personne qui a la plus grande taille
  - Créez un fichier (dont le nom est à saisir par l'utilisateur) contenant le nom et la taille des personnes sauvegardés dans la liste **L1**,
  - Affichez le contenu du fichier créé.

```

void main(){
struct personne*L1=NULL,*L2=NULL,*T;
float m;
char nom[15]; L1=saisie();
display(L1);
m=moyenne(L1);
printf("\nmoyenne=%f\n",m);
L2=creation(L1);
display(L2);
printf("\nTaille>moyenne:");
T=L2;
while(T!=NULL&&T->taille<m)
T=T->next;
if(T!=NULL)
{while(T!=NULL)
{printf("%f\t",T->taille);
T=T->next;
}
}

```

```

}
T=L1;
m=T->taille;
while(T!=NULL)
{if(T->taille>m)
{m=T->taille;
strcpy(nom,T->nomP);
}
T=T->next;
}
printf("\nnom delapersonnedont la taille est max est %s\n",nom);
}

```

## **Exercice 2** ( pts )

On souhaite gérer les enseignants à l'université d'un département donné. Pour cela, nous allons demander d'utiliser un arbre binaire de recherche où les nœuds contiennent les informations concernant les enseignants. Un enseignant est caractérisé par son nom, sa spécialité et le nombre d'heures effectuées par semaine. L'arbre doit être trié d'une manière croissante par rapport au nombre d'heures effectuées.

1. Définissez le type **Enseignant**.

```

typedef struct Enseignant {
    char nom[20];
    char spec[20];
    int nbHeures;
}Enseignant;

```

2. Définir le type **Departement** qui permet de représenter le département.

<pre> typedef struct Noeud{     Enseignant e ;     struct Noeud*filsGauche ;     struct Noeud*filsDroite ; }Noeud ; </pre>	<pre> typedef struct Departement{     Nœud*racine ; }Departement ; </pre>
--	---

3. Ecrire la méthode **ajouterEnseignant(Departement \*dep, Enseignant e)** qui permet d'ajouter un enseignant au département **dep** en respectant l'ordre fixé. Dans le cas où l'enseignant fait déjà un membre du département, le système affiche un message d'alerte.

```

void ajouterEnseignantRec(Noeud *n , Enseignant e)
{

```

```

if(e.nbHeures<n->e.nbHeures)
{
    if(n-> filsGauche == NULL )
    {
        noeud *a = (noeud*)malloc(sizeof(noeud));
        a->e = e;
        a-> filsDroite = NULL;
        a-> filsGauche = NULL;
        n-> filsGauche = a
    }
    else ajouterEnseignantRec (n-> filsGauche,e);
}
else //<=
{
    if(n-> filsDroite == NULL)
    {
        noeud *a = (noeud*)malloc(sizeof(noeud));
        a->e = e;
        a-> filsDroite = NULL;
        a-> filsGauche = NULL;
        n-> filsDroite = a
    }
    else ajouterEnseignantRec (n-> filsDroite ,e);
}
}
}

```

```

void ajouterEnseignant (Departement*dep, Enseignant e)
{
/* if(a->racine == NULL)
   dep->racine = CreerNoeud(e);*/
if( !membre(dep->racine,e))
    ajouterEnseignantRec(dep->racine,e);
else printf("Alerte :L'enseignant est deja un membre du departement") ;
}

int membre (Node*n,Enseignant){
if(n==NULL) return 0 ;
if(strcmp(n->e,e)==0) return 1;
if( membre(n->filsGauche)) return 1 ;
return membre(n->filsDroite);
}

```

4. Ecrire la méthode ***afficheDepartement(Departement dep)*** qui affiche toutes les données de tous les enseignants du département **dep** passé en paramètre.

```
void afficheDepartementRec(Noeud *a)
```

```

    {   if(a-> filsGauche !=NULL) afficheDepartementRec (a-> filsGauche);
        printf("%s\t%s\t%d\n",a->e.nom,a->e.spec,a->e.nbHeures);
        if(a-> filsGauche!=NULL) afficheDepartementRec (a-> filsGauche);
    }

void afficheDepartement (Departement a)
{   if(a.racine!=NULL)
    return afficheDepartementRec (a.racine);
}

```

5. Ecrire la méthode **moyenneHeures(Departement dep)** qui permet de calculer la moyenne générale des heures effectuées par les enseignants par semaine du département **dep**.

```

int sommeHeuresRec (Noeud *a)
{   if(a==NULL) return 0 ;
    return sommeHeuresRec(a->filsGauche)+sommeHeuresRec(a->filsDroite)+ a->e.nbHeures ;
}

int nbEnseignantsRec (Noeud *a)
{   if(a==NULL) return 0 ;
    return nbEnseignantsRec (a->filsGauche)+ nbEnseignantsRec (a->filsDroite)+ 1 ;
}

float moyenneHeures (Departement dep)
{   if (dep.racine!=NULL)
    return sommeHeuresRec (dep.racine)*1.0/nbEnseignantsRec(dep.racine);
    return -1 ;
}

```

6. Ecrire la méthode **enregistrer(Departement dep, char\*f)** qui permet d'enregistrer les informations des enseignants du département **dep** dans le fichier **f**.

```

void enregistrerRec(noeud*n,FILE **f){
    if(n==NULL) return;
    enregistrerRec (n-> filsGauche,f);
    fprintf(*f,"%s %s %d\n",d.nom,d.spec,d.nbHeures);
    enregistrerRec(n-> filsDroite,f);
}

void enregistrer(Departement a,char*fileName){
FILE*file;
file=fopen(fileName,"w");
if(file==NULL)
    {printf("echec d'ouverture!!");exit(-1);
}
enregistrerRec (a.racine,&file);
fclose(file);
}

```

7. Ecrire la méthode ***charger (char\*f)*** qui permet de charger la liste des enseignants enregistrés dans le fichier **f** et retourne et retourne le département formé de ces enseignants.

```
Departement charger (char*fileName){
    FILE*file;
    file=fopen(fileName,"r");
    Departement *a = (Departement*)malloc(sizeof(Departement));
    a->racine = NULL;

    while(!feof(file)){
        Enseignant *d=(Enseignant*)malloc(sizeof(Enseignant));
        char nom[20],spec[20];
        int nbh;
        fscanf(file,"%s %s %d\n",nom,spec,&nbh);
        strcpy(d->nom,nom);
        strcpy(d->spec,spec);
        d->nbHeures=nbh;
        ajouterEnseignant (a,*d);
    }
    fclose(file);
    return *a;
}
```

***Bonne Chance***

# I2206 Session 1 (2020 – 2021)

## Exercise 1 ( 55 pts )

The objective of this exercise is to create a linked list, representing the data of racehorses. Each horse characterized by his name and his rank that represents its classification in the race.

- 1) Give the necessary data types to define an item of list.
- 2) Write a function that takes as parameter a linked list, and adds a horse at the tail of this list. The information, concerning the horse to add, should be given, by the user, in the function.
- 3) Write a function that takes as parameter the number of horses  $n$ , and creates a linked list of  $n$  horses.
- 4) Write a function that returns the horse having the highest rank.
- 5) Write a function that deletes the horse having the highest rank.
- 6) Write a function that takes as parameter a linked list, and displays this linked list.
- 7) Write a function that takes as parameter a linked list as well as the name of a file, and saves the linked list in the file.
- 8) Write a function that takes as parameter the name of a file, and display the content of this file.

## Solution

```
#include <stdio.h>
#include<stdlib.h>
/* Question 1 */
typedef struct horse {
    char name[30];
    int rank;
    struct horse * next;
} horse ;

/* Question 2 */
horse * addTail(horse * head) {
    horse * new = (horse *)malloc(sizeof(horse));
    printf("Give the name of the horse : "); scanf("%s", new->name);
    printf("Give the rank of the horse : "); scanf("%d", &new->rank);
    new->next = head;
    head = new;
    return head;
}
```

```

if (head == NULL) {
    new->next = head;
    head = new;
}
else {
    new->next = NULL;
    horse * liste = head;

    while(liste->next != NULL) liste = liste->next;

    liste->next = new;
}
return head;
}

```

### */\* Question 3 \*/*

```

horse * create(int n) { // insertion at tail of list
    horse * liste = NULL;
    int i;
    for (i = 1; i <= n; i++)
        liste = addTail(liste);
    return liste;
}

```

### *Other solution question 3:*

```

horse * create(int n) { // insertion at head of list
    horse * liste = NULL , *new = NULL;
    int i;
    for(i = 1; i <= n; i++) {
        printf("for horse %d, give\n", i);
        new = (horse *)malloc(sizeof(horse));
        printf("\this name : "); scanf("%s", new->name);
        printf("\this rank : "); scanf("%d", &new->rank);
        new->next = liste;
        liste = new;
    }
    return liste;
}

```

*/\* Question 4 \*/*

```
horse * maxRank(horse * head) {  
    if (head == NULL) return NULL;  
    else {  
        horse * max = head, *p = head->next;  
        while(p != NULL) {  
            if(max->rank < p->rank) max = p;  
            p = p->next;  
        }  
        return max;  
    }  
}
```

*/\* Question 5 \*/*

```
horse * deleteMaxRank(horse * head) {  
    horse *max = maxRank(head);  
    horse *before, *after, *liste = head;  
    if (max != NULL) {  
        if (max == head)  
            head = head->next;  
        else {  
            after = max->next;  
            while(liste->next != max) liste = liste->next;  
            before = liste;  
            before->next = after;  
        }  
        return head;  
    }  
}
```

*/\* Question 6 \*/*

```
void display(horse * head) {  
    horse * liste = head;  
    while(liste != NULL) {  
        printf("horse %s has rank %d\n", liste->name, liste->rank);  
        liste = liste->next;  
    }  
}
```

*/\* Question 7 \*/*

```
void saveFile(horse * head, char * file) {  
    FILE * pf;  
    pf = fopen(file, "w");  
    if(pf == NULL) printf("\nImpossible to open file\n");  
    else {  
        horse * liste = head;  
        while(liste != NULL) {  
            fprintf(pf, "%s\t%d\n", liste->name, liste->rank);  
            liste = liste->next;  
        }  
        fclose(pf);  
    }  
}
```

*/\* Question 8 \*/*

```
void displayFile(char * file) {  
    FILE *pf;  
    pf = fopen(file, "r");  
    if(pf == NULL) printf("\nImpossible to open file\n");  
    else {  
        char nameHorse[30];  
        int r;  
        while(!feof(pf)) {  
            fscanf(pf, "%s%d", nameHorse, &r);  
            if(!feof(pf))  
                printf("Horse %s has rank %d\n", nameHorse, r);  
        }  
        fclose(pf);  
    }  
}
```

*/\* Supplementary Questions ( to see results ) \*/*

```
void displayMaxRank(horse * liste) {  
    if (liste != NULL)  
        printf("horse %s has rank %d\n", liste->name, liste->rank);  
}
```

```

void main() {
    printf("---- Creation of 5 horses ----\n");
    horse * H = create(5);
    printf("\n---- List of horses ----\n");
    display(H);
    printf("\n----Add a horse at end of liste ----\n");
    H = addTail(H);
    printf("\n---- Horse having max rank ----\n");
    horse * max = maxRank(H);
    displayMaxRank(max);
    printf("\n---- After deletion of the horse ----\n");
    H = deleteMaxRank(H);
    display(H);
    printf("\n---- Creation of File ----\n");
    saveFile(H, "HorsesFile.txt");
    printf("\n---- Display File ---\n");
    displayFile("HorsesFile.txt");
}

```

```

---- Creation of 5 horses ----
for horse 1, give
    his name : horse1
    his rank : 3
for horse 2, give
    his name : horse2
    his rank : 1
for horse 3, give
    his name : horse3
    his rank : 6
for horse 4, give
    his name : horse4
    his rank : 2
for horse 5, give
    his name : horse5
    his rank : 4

```

Using Insertion at head method

```
---- List of horses ----  
horse horse5 has rank 4  
horse horse4 has rank 2  
horse horse3 has rank 6  
horse horse2 has rank 1  
horse horse1 has rank 3
```

After Insertion at head

```
----Add a horse at end of liste ----  
Give the name of the horse : horse6  
Give the rank of the horse : 5
```

```
---- Horse having max rank ----  
horse horse3 has rank 6
```

```
---- After deletion of the horse ----  
horse horse5 has rank 4  
horse horse4 has rank 2  
horse horse2 has rank 1  
horse horse1 has rank 3  
horse horse6 has rank 5
```

```
---- Creation of File ----
```

main.c	HorsesFile.txt :
1	horse5 4
2	horse4 2
3	horse2 1
4	horse1 3
5	horse6 5
6	

```
---- Display File ---  
Horse horse5 has rank 4  
Horse horse4 has rank 2  
Horse horse2 has rank 1  
Horse horse1 has rank 3  
Horse horse6 has rank 5
```

## Exercise 2

**( 45 pts )**

The objective of this exercise is to manage, through a binary search tree, the grades of students in I2206 course. Each student characterized by his university number (integer), his name (string) and his grade (real). The tree must be sorted in ascending order according to the student's grade.

- 1) Give the necessary data types to define an element of the binary tree.
- 2) Write a function that takes as parameter a student, and adds this student to course I2206.
- 3) Write a function that display the student's information of course I2206. The display should be according to the student's grade (from highest grade to lowest grade).
- 4) Write a function that calculates the number of students who passed in I2206
- 5) Write a function that returns the student having the highest grade.
- 6) Write a program in which, you
  - a) Asks the user to give an integer  $n$  that should be between 3 and 10. Then, Adds  $n$  students to the course I2206.
  - b) Display the students registered in course I2206.
  - c) Display the number of students who passed in course I2206.
  - d) Display the university number, name and grade of the student having the highest grade.

### **Solution**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* Question 1 */
typedef struct student {
    int num;
    char name[30];
    float grade;
    struct student *left, *right;
} stud;

typedef struct tree {
    stud * root;
} tree;

/* supplementary function */
tree createEmpty() {
    tree * t = (tree *)malloc(sizeof(tree));
    t->root = NULL;
    return * t;
}
```

```

/* Question 2 */

stud * createLeaf(int num, char *name, float grade) {
    stud *new = (stud *)malloc(sizeof(stud));
    new->num = num;
    strcpy(new->name, name);
    new->grade = grade;
    new->left = NULL;
    new->right = NULL;
    return new;
}

```

```

void insertStud(stud * S, int num, char *name, float grade) {
    if(S->grade > grade) {
        if(S->left == NULL) S->left = createLeaf(num, name, grade);
        else insertStud(S->left, num, name, grade);
    }
    else {
        if(S->right == NULL) S->right = createLeaf(num, name, grade);
        else insertStud(S->right, num, name, grade);
    }
}

```

```

void insertTree(tree * a, int num, char *name, float grade) {
    if (a->root != NULL) insertStud(a->root, num, name, grade);
    else a->root = createLeaf(num, name, grade);
}

```

```

/* Question 3 */

void displayStud(stud * S) {
    if(S->right != NULL) displayStud(S->right);
    printf("%d\t%s\t%f\n", S->num, S->name, S->grade);
    if(S->left != NULL) displayStud(S->left);
}

```

```

void displayTree(tree a) {
    if (a.root != NULL) displayStud(a.root);
}

```

*/\* Question 4 \*/*

```
int nbStud(stud * S) {
    int nb = 0;
    if(S != NULL) {
        if (S->grade >= 50) nb++;
        nb += nbStud(S->left);
        nb += nbStud(S->right);
    }
    return nb;
}
```

**int nbTree(tree a) {**

```
    if (a.root == NULL) return 0;
    else return nbStud(a.root);
}
```

*/\* Question 5 \*/*

```
stud * studMax(stud *S) {
    if (S == NULL) return NULL;
    else {
        while(S->right != NULL) S = S->right;
        return S;
    }
}
```

**stud \* maxTree(tree a) {**

```
    if (a.root == NULL) return NULL;
    else return studMax(a.root);
}
```

*/\* Question 6 \*/*

```
void main() {
    int n, i, num;
    float grade;
    char name[30];
    tree a = createEmpty();
    stud * max;
```

```
/* Question 6.a */
do {
    printf("give nb stud between 3 and 10 : ");
    scanf("%d", &n);
} while(n < 3 || n > 10);

printf("\n---- Add %d stud ----\n", n);
for (i = 1; i <= n; i++) {
    printf("for stud %d, give \n", i);
    printf("\this num : "); scanf("%d", &num);
    printf("\this name : "); scanf("%s", name);
    printf("\this grade : "); scanf("%f", &grade);
    insertTree(&a, num, name, grade);
}
```

#### /\* Question 6.b \*/

```
printf("\n---- Stud in Course I2206 ----\n");
displayTree(a);
```

#### /\* Question 6.c \*/

```
printf("\nnb of students passed exam is %d\n", nbTree(a));
```

#### /\* Question 6.d \*/

```
printf("\n---- Stud having max grade ----\n");
max = maxTree(a);
printf("%d\t%s\t%f\n", max->num, max->name, max->grade)
}
```

```
give nb stud between 3 and 10 : 4  
---- Add 4 stud ----  
for stud 1, give  
    his num : 123  
    his name : stud1  
    his grade : 45  
for stud 2, give  
    his num : 223  
    his name : stud2  
    his grade : 67  
for stud 3, give  
    his num : 323  
    his name : stud3  
    his grade : 23.75  
for stud 4, give  
    his num : 423  
    his name : stud4  
    his grade : 56
```

```
---- Stud in Course I2206 ----  
223      stud2      67.000000  
423      stud4      56.000000  
123      stud1      45.000000  
323      stud3      23.750000
```

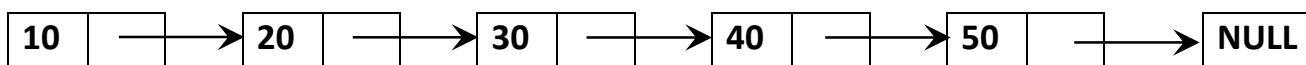
```
nb of students passed exam is 2
```

```
---- Stud having max grade ----  
223      stud2      67.000000
```

## Exercise 1 ( 50 pts ~ 40 min )

The objective of this exercise is to create a singly linked list of integer numbers.

- 1) Give the necessary data types to define an item of list.
- 2) Write a function that takes as parameter an integer  $n$  and, creates a linked list of  $n$  integers.
- 3) Write a function that takes as parameter a linked list and **returns** this list in its reversed order. For example, if we have the following linked list



The reversed linked list is



- 4) Write a function that takes as parameter a linked list and displays its elements.
- 5) Write a function that takes as parameter a linked list and returns the total number of nodes in this linked list.
- 6) By using previous functions, write the main program in which,
  - You create a singly linked list of five integer numbers, then
  - Display the total number of nodes of the linked list,
  - Display the reversed linked list.

## Exercise 2 ( 50 pts ~ 40 min )

We want to classify the cities of Lebanon in a binary search tree. We characterize each city by its name (string of 40 characters) and its population number (integer); Tree is sorted in ascending order according to the population number of the city.

- 1) Give the necessary data types to define an element of the tree.

- 2) Write a function that verify the existence, in the tree, of a city given as a parameter of the function.  
The function returns 1 if the city exists and 0 otherwise.
- 3) Write a function that reads the information of a city (name and population number) and returns this city.
- 4) Write a function that adds, to the tree, a city given as a parameter of the function.
- 5) Write a function that returns the city having the smallest number of population.
- 6) Write a function that removes (i.e., deletes), from the tree, the city having the smallest number of population, while retaining the property of a binary search tree.
- 7) Using the previous functions, write the main program in which you
- Add 5 cities to the tree; **Be careful not to add a city that already exists in the tree;**
  - Remove, from the tree, the city having the smallest number of population.

## Lab Exam ( ~ 10 min )

( لِجَابَةٌ عَلَى نَفْسِ الْكِرَاسِ مَعَ Exercice 1 & Exercice 2 )

In a hospital, a nurse can be defined by his name and his number of experience years.

1. Give the necessary data types to define the structure nurse.
2. Write a program that allows to
  - Creates a file (whose name should be given by the user) containing the information of 10 nurses,
  - Displays the created file.

**Good Luck**

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

Ex I -
typedef struct item
{
    int n;
    struct item *next;
} Item;

Item *createList(int length)
{
    Item *List = NULL, *prev = NULL;
    for (int i = 0; i < length; i++)
    {
        Item *new = (Item *)malloc(sizeof(Item));
        printf("Give a number: ");
        scanf("%d", &new->n);
        new->next = prev;
        prev = new;
        List = new;
    }

    return List;
}

Item *reverseList(Item *I)
{
    Item *Reversed = NULL, *prev = NULL, *p = I;
    while (p != NULL)
    {
        Item *new = (Item *)malloc(sizeof(Item));
        new->n = p->n;
        if (Reversed == NULL)
        {
            new->next = NULL;
            Reversed = new;
        }
        else
        {
            new->next = Reversed;
            Reversed = new;
        }
        p = p->next;
    }
    return Reversed;
}

void display(Item *I)
{
    Item *p;
    while (p != NULL)
    {
        printf("%d\t", p->n);
        p = p->next;
    }
}

int *sumList(Item *I)
{
    Item *p = I;
    int sum = 0;
    while(p != NULL)
    {

```

```

        sum += p->n;
        p = p->next;
    }
    return sum;
}

int main()
{
    Item *List, *Reversed;
    List = createList(5);
    display(List);
    printf("\n");
    Reversed = reverseList(List);
    display(Reversed);
    printf("\nThe total number is %d", sumList(List));
}

// Ex 2

typedef struct city
{
    char name[20];
    int population;
    struct city *left;
    struct city *right;
} City;

int exist(City *C, City *C2)
{
    if (C != NULL)
    {
        if (C2->population == C->population)
        {
            return 1;
        }
        else
        {
            if (C2->population < C->population)
                return exist(C->left, C2);
            else
                return exist(C->right, C2);
        }
    }
    return 0;
}

City *CreateCity()
{
    City *new = (City *)malloc(sizeof(City));
    printf("Give the city's name: ");
    scanf("%s", new->name);
    printf("Give the population number: ");
    scanf("%d", &new->population);
    new->left = NULL;
    new->right = NULL;
    return new;
}

City *addCity(City *C, City *toAdd)
{
    if (C == NULL)
    {
        C = toAdd;
    }
    else if (toAdd->population < C->population)
    {

```

```

    if (C->left == NULL)
    {
        C->left = toAdd;
    }
    else
    {
        C->left = addCity(C->left, toAdd);
    }
}
else
{
    if (C->right == NULL)
    {
        C->right = toAdd;
    }
    else
    {
        C->right = addCity(C->right, toAdd);
    }
}
return C;
}

void display(City *C)
{
    if (C != NULL)
    {
        printf("%s,%d\t", C->name, C->population);
        display(C->left);
        display(C->right);
    }
}

City *smallestCity(City *Tree)
{
    if (Tree->left != NULL)
    {
        return smallestCity(Tree->left);
    }
    else
    {
        return Tree;
    }
}

City *removeSmallest(City *Tree)
{
    City *smallest = smallestCity(Tree), *ToDelete;
    if (smallest == Tree)
    {
        ToDelete = Tree;
        Tree = Tree->right;
    }
    else
    {
        Tree->left = removeSmallest(Tree->left);
    }
    return Tree;
}

int main()
{
    City *Tree = NULL;
    for(int i=0; i<5; i++)
    {
        printf("\ncity number %d\n", i+1);
    }
}

```

```
City *c = CreateCity();
if(exist(Tree, c))
{
    printf("this city is already exist");
    i--;
}
else
{
    Tree = addCity(Tree, c);
}
display(Tree);
City *smallest = smallestCity(Tree);
printf("\nThe smallest city is: %s\n", smallest->name);
Tree = removeSmallest(Tree);
display(Tree);
}
//@M.F
```

## Exercise 1 (Final Exam)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Question 1 */

typedef struct course {
    char code[7];
    int credit;
    struct course * next;
} course;

typedef struct category {
    char name[20];
    int nbCourses;
    course * listCours;
    struct category * next;
} category;
```

---

### **/\* Question 2 \*/**

```
course * insertTail(course *head) {
    course * new = (course*)malloc(sizeof(course));
    printf("\t\t give the name of the course %d: "); scanf("%s", new->code);
    do {
        printf("\t\t give the credit nb (between 3 and 6): ");
        scanf("%d", &new->credit);
    } while (new->credit < 3 || new->credit > 6);
    if (head == NULL) {
        new->next = head;
        head = new;
    }
}
```

```
else {
    new->next = NULL;
    course *p = head;
    while (p->next != NULL)
        p = p->next;
    p->next = new;
}
return head;
}
```

---

### ***/\* Question 3 \*/***

```
course *createCourses (int n) {
    course *p = NULL;
    int i;
    for (i = 1; i <= n; i++)
        p = insertTail(p);
    return p;
}
```

---

### ***/\* Question 4 \*/***

```
int sumCredit (course *Cours) {
    int sumCredit = 0;
    course *p = Cours;
    while (p != NULL) {
        sumCredit += p->credit;
        p = p->next;
    }
    return sumCredit;
}
```

---

### ***/\* Question 5 \*/***

```
category * createCategory ( ) {
    int n, i, nb;
    category *p = NULL, *new = NULL;
```

```

do {
    printf("give nb of categories (between 2 and 8) : ");
    scanf("%d", &n);
} while(n < 2 || n > 8);
for (i = 1; i <= n; i++) {
    printf("\n for category %d, give \n", i);
    new = (category*)malloc(sizeof(category));
    printf("\t its name : "); scanf("%s", new->name);
    printf("\t nb of its courses : "); scanf("%d", &new->nbCourses);
    new->listCours = createCourses(new->nbCourses);
    int s = sumCredit(new->listCours);           /* not required */
    printf("\t ==> sum of credits = %d\n", s);   /* not required */
    new->next = p;
    p = new;
}
return p;
}

```

---

### ***/\* Question 6 \*/***

```

void displayCours(course * cours) {
    course * p = cours;
    while (p != NULL) {
        printf("\n\t\t code: %s\t nb credits: %d", p->code, p->credit);
        p = p->next;
    }
}

void displayCategory(category *C) {
    category *p = C;
    course *cours = NULL;
    int i = 1;
    while (p != NULL) {
        printf("\nCategory %d has : \n", i);
        printf("\tname : %s ", p->name);

```

```

printf("\tnb of courses : %d ", p->nbCourses);
int s = sumCredit(p->listCours);
printf("\ttotal sum of credits = %d", s);
displayCours(p->listCours);
p = p->next;
i++;
}
}

```

---

### **/\* Question 7 \*/**

```

category *belongs (category *C, char *name) {
    category *p = C, *trouve = NULL;
    int exist = 0;
    while (p != NULL) {
        if (strcmp(p->name, name) == 0) {
            exist = 1; trouve = p; break;
        }
        p = p->next;
    }
    if (exist == 0) printf("category %s does not exist in the list\n", name);
    else printf("category %s exist in the list", name);
    return trouve;
}

```

### **Other solution :**

```

category *belongs (category *C, char *name) {
    category *p = C;
    while (p != NULL) {
        if (strcmp(p->name, name) == 0) return p;
        p = p->next;
    }
    return NULL;
}

```

---

**/\* Question 8 \*/**

```
category *deleteCategory(category * C, char *name) {
    category *search = belongs(C, name);
    if (search == NULL) printf("category %s cannot be deleted ", name);
    else {
        category *before, *after, *p = C;
        if (search == C) C = C->next;
        else {
            after = search->next;
            while(p->next != search)
                p = p->next;
            before = p;
            before->next = after;
        }
    }
    return C;
}
```

---

## **Exercise 2 (Lab Exam)**

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

**/\* Question 1 \*/**

```
typedef struct employee {
    char name[20];
    float salary; /* or int salary; */
    struct employee *left, *right;
} emp;
```

```
typedef struct tree {  
    emp * root;  
} tree;
```

---

### /\* Question 2 \*/

```
emp * createLeaf(char * name, float salary) {  
    emp * new = (emp *)malloc(sizeof(emp));  
    strcpy(new->name, name);  
    new->salary = salary;  
    new->right = NULL;  
    new->left = NULL;  
    return new;  
}  
  
void insertEmp(emp * E, char *name, float salary) {  
    if (E->salary > salary) {  
        if (E->left == NULL)  
            E->left = createLeaf(name, salary);  
        else insertEmp(E->left, name, salary);  
    }  
    else {  
        if (E->right == NULL)  
            E->right = createLeaf(name, salary);  
        else insertEmp(E->right, name, salary);  
    }  
}  
  
void insertTree(tree *a, char *name, float salary) {  
    if (a->root != NULL) insertEmp(a->root, name, salary);  
    else a->root = createLeaf(name, salary);  
}
```

---

### /\* Question 3 \*/

```
void displayPostEmp(emp * E) { /* Postfixed => left, right, root */  
    if (E->left != NULL) displayPostEmp(E->left);  
    if (E->right != NULL) displayPostEmp(E->right);  
    printf("Name : %s\tSalary : %f\n", E->name, E->salary);  
}
```

```
void displayTree(tree a) {  
    if (a.root != NULL) displayPostEmp(a.root);  
}  
/* or  
void displayTree(tree *a ) {  
if (a->root != NULL) displayPostEmp(a->root);  
}  
*/
```

---

#### /\* Question 4 \*/

```
int nbEmployee(emp * E) {  
    int nb = 0;  
    if (E != NULL) {  
        nb++;  
        nb += nbEmployee(E->left);  
        nb += nbEmployee(E->right);  
    }  
    return nb;  
}  
  
int nbTree(tree a) {  
    if (a.root == NULL) return 0;  
    else return nbEmployee(a.root);  
}  
/* or  
int nbTree(tree *a) {  
    if (a->root == NULL) return 0;  
    else return nbEmployee(a->root);  
}  
*/
```