Votre numéro d'anonymat : | | | |

Programmation et structures de données en C– 2I001

Examen du 6 janvier 2017
2 heures
Aucun document n'est autorisé.

Les calculatrices, baladeurs et autres appareils électroniques sont interdits. Les téléphones mobiles doivent être éteints et rangés dans les sacs. Le barème sur 60 points (11 questions) n'a qu'une valeur indicative.

Les appels à `malloc` seront supposés réussir. Il ne sera pas nécessaire de vérifier leur valeur de retour.

Le mémento qui vous a été distribué est reproduit à la fin de l'énoncé.

L'ensemble des structures et prototypes de fonctions est également rappelé à la fin de l'énoncé, sur une page détachable.

Souriez, vos données sont enregistrées

La vente de produits passe de plus en plus par une connaissance approfondie de l'acheteur et de ses goûts. Il vous a été demandé de mettre en place un système permettant de gérer un ensemble de clients avec tous les achats qu'ils ont réalisés afin de pouvoir réaliser, dans un second temps des études statistiques et des recommandations aux utilisateurs (ce qui ne vous est pas demandé ici).

Un client disposera, dans votre programme, d'un numéro, d'un nom et d'une liste d'achats. Un achat aura une date, un objet (chaîne de caractères) et un montant. Les achats seront représentés sous la forme d'une liste chaînée.

Les structures correspondantes sont données ci-après :

```
typedef struct _Date {
  unsigned char jour;
  unsigned char mois;
  unsigned int annee;
} Date;

typedef struct _Achat {
  Date date;
  char *objet;
  float montant;
  struct _Achat *suivant;
} Achat;

typedef struct _Client {
```

```
  int num;
  char * nom;
  Achat *l_achat; /* liste des achats d'un client */
} Client;
```

1 Gestion d'un client seul

Question 1 (3 points)

Ecrivez une fonction permettant d'allouer un élément de type `Achat`. La chaîne de caractères `objet` passée en paramètre peut être détruite juste après l'appel à cette fonction. Prototype :

```
Achat *creer_achat(Date d,char *objet, float montant);
```

Solution:

```
Achat *creer_achat(Date d,char *objet, float montant) {
  Achat *a=malloc(sizeof(Achat));
  a->date=d;
  a->objet=strdup(objet);
  a->montant=montant;
  a->suivant=NULL;
  return a;
}
```

Question 2 (3 points)

Ecrivez une fonction permettant d'ajouter un achat à un client. L'ordre des achats n'a pas d'importance. Prototype :

```
void ajouter_achat(Client *c, Achat *a);
```

Solution:

```
void ajouter_achat(Client *c, Achat *a) {
  if (!c || !a) {
    printf("WARNING: client ou achat non defini\n");
    return;
  }
  a->suivant=c->l_achat;
  c->l_achat=a;
}
```

Le test initial des valeurs de `c` et `a` n'est pas obligatoire. Il devra être considéré comme un bonus. Ce sera le cas dans toutes les fonctions pour lesquelles une valeur `NULL` n'est pas un cas "normal". Il sera par contre considéré comme une erreur de ne pas tester si une liste ou un arbre ne sont pas vides (en particulier dans le cas de récurrences).

Question 3 (6 points)

Ecrivez une fonction permettant de calculer le montant total des achats d'un client. Prototype :

```
float montant_achats(Client *c);
```

Solution:

```
float montant_achats(Client *c) {
  if (c==NULL) return 0;
  float total=0;
  Achat *a=c->l_achat;
  while (a) {
    total+=a->montant;
    a=a->suivant;
  }
  return total;
}
```

Question 4 (3 points)

Ecrivez une fonction permettant d'écrire un client dans un fichier. Un seul client sera écrit à la fois. Vous devrez définir au préalable une fonction permettant d'écrire un achat seul. Le format retenu sera le suivant :

```
3452: Skywalker
   12/04/5137 Sabre_Laser 5834.12
   01/03/5135 Tenue_Jedi 344.00
   fin achats
```

La dernière ligne sera utilisée par la fonction de lecture pour déterminer la fin de la liste d'achats.

Le fichier sera transmis à la fonction au travers d'un FILE* (il aura donc été ouvert avant l'appel à la fonction).

Protopypes :

```
void ecrire_achat(FILE *f, Achat *a);

void ecrire_client(FILE *f, Client *c);
```

Solution:

```
void ecrire_achat(FILE *f,Achat *a) {
  fprintf(f,"␣␣␣%02hhu/%02hhu/%4u␣%s␣%f\n",a->date.jour,a->date.mois
     , a->date.annee, a->objet,a->montant);
}

void ecrire_client(FILE *f, Client *c) {
  if (c!=NULL) {
    fprintf(f,"%d:␣%s\n",c->num, c->nom);
    Achat *a=c->l_achat;
    while (a) {
      ecrire_achat(f,a);
      a=a->suivant;
    }
    fprintf(f,"␣␣␣fin␣achats\n");
  }
}
```

Le format exact à utiliser ici ne figure pas dans le memento. Il est donc demandé de ne pas considérer comme faux l'utilisation d'un `%d` pour un entier, quel qu'il soit. L'utilisation du bon format pourra être considérée comme un bonus.

Question 5 (9 points)

Ecrivez une fonction permettant de lire un client depuis un fichier. Le fichier sera transmis à la fonction au travers d'un `FILE *` (il aura donc été ouvert avant l'appel à la fonction). Le format sera le même que pour la fonction d'écriture de la question précédente. Pour simplifier, il sera fait l'hypothèse que les chaînes de caractères (noms et objets) ne contiennent pas d'espace. L'utilisation d'un `%s` permettra donc de lire d'un coup chacune de ces chaînes. Il sera considéré qu'elles seront de taille inférieure strictement à 40. Les lignes seront considérées comme étant de taille strictement inférieure à 200. Vous prendrez soin de vérifier que le format du fichier à lire est correct. Vous pourrez utiliser des fonctions fournies ou vues précédemment.

Prototype :

```
Client *lire_client(FILE *f);
```

La fonction renverra `NULL` si la lecture a échoué.

Solution:

```
Client *lire_client(FILE *f) {
  int num;
  char nom[40];
  char ligne[200];
  unsigned char jour, mois;
  unsigned int annee;
  char objet[200];
  float montant;

  if (fgets(ligne,200,f)==NULL) {
    return NULL;
  }
  if (sscanf(ligne,"%d: %s",&num,nom)!=2) return NULL;
  Client *c=creer_client(num,nom);
  while (1) {
    if (fgets(ligne,200,f)==NULL) {
      printf("Format incorrect\n");
      liberer_client(c);
      return NULL;
    }
    int res=sscanf(ligne," %hhu/%hhu/%u %s %f", &jour,&mois,&annee,
       objet,&montant);
    if (res!=5) {
      /* fin des achats: lecture de la ligne contenant "fin" (ou d'
         une ligne invalide) */
      break;
    }
    Date d={jour,mois,annee};
    Achat *a=creer_achat(d,objet,montant);
```

```
    ajouter_achat(c,a);
  }
  return c;
}
```

Même remarque que précédemment concernant les formats.

Question 6 (6 points)

Ecrivez une fonction permettant de supprimer les doublons dans la liste d'achats d'un client. Vous pourrez utiliser des fonctions fournies ou vues précédemment, notamment la fonction `chercher_achat` qui permet de trouver un achat dans une liste correspondant à une date, un objet et un montant spécifié. La fonction `chercher_achat` est fournie et n'est pas à écrire. Prototypes :

```
Achat *chercher_achat(Achat *l, Date d, char *objet, float montant);

void supprimer_doublons(Client *c);
```

Solution:

```
void supprimer_doublons(Client *c) {
  if (c==NULL) return;
  Achat *l=c->l_achat;
  if (l==NULL) return;
  while (chercher_achat(l->suivant,l->date,l->objet,l->montant)!=
     NULL) {
    c->l_achat=l->suivant;
    liberer_achat(l);
    l=c->l_achat;
  }

  while (l->suivant) {
    if (chercher_achat(l->suivant->suivant,l->suivant->date, l->
       suivant->objet, l->suivant->montant)!=NULL) {
      Achat *tmp=l->suivant;
      l->suivant=l->suivant->suivant;
      liberer_achat(tmp);
    }
    else {
      l=l->suivant;
    }
  }
}
```

2 Gestion d'un ensemble de clients

La société qui envisage d'utiliser ce programme dispose d'une grande quantité de clients. Pour des raisons d'efficacité lors de recherches sur les numéros de clients, vous avez proposé d'ordonner les clients selon un arbre binaire de recherche prenant en compte le numéro de client. Tous les clients qui

se situent dans le sous-arbre gauche d'un noeud `n` auront donc un numéro de client inférieur à celui du client porté par `n`. Les clients qui sont dans le sous-arbre droit auront eux un numéro de client qui sera strictement supérieur (les numéros de client sont tous différents). La recherche dans un arbre binaire de recherche est plus efficace lorsque l'arbre est équilibré, c'est à dire lorsque, pour tout noeud de l'arbre, la hauteur des sous-arbres gauche et droit est identique ou ne diffère que de 1. La construction d'un arbre binaire de recherche ne crée par nécessairement un arbre équilibré. Nous allons nous efforcer de créer un arbre équilibré en suivant une méthode qui requiert dans un premier temps de stocker les données lues dans une liste chaînée triée. Cela permettra de choisir l'ordre le plus approprié pour construire l'arbre. Nous allons nous intéresser à quelques fonctions de manipulation de cette structure d'arbre et de la structure de liste chaînée avant de nous intéresser à la construction d'un arbre équilibré.

Les structures d'arbre et de liste sont les suivantes :

```
typedef struct _Abr_Client {
  Client *data;
  struct _Abr_Client *gauche;
  struct _Abr_Client *droit;
} Abr_Client;

typedef struct _Liste_Client {
  Client *data;
  struct _Liste_Client *suivant;
} Liste_Client;
```

Question 7 (6 points)

Ecrivez une fonction permettant de rechercher un client à partir de son numéro dans un arbre binaire de recherche. Prototype :

```
Client *rechercher_client(Abr_Client *abr, int num);
```

La fonction renvoie `NULL` si aucun client ayant ce numéro n'a été trouvé dans l'arbre.

Solution:

```
Client *rechercher_client(Abr_Client *abr, int num) {
  while (abr!=NULL) {
    if (abr->data->num==num) return abr->data;
    if (num<abr->data->num) abr=abr->gauche;
    else abr=abr->droit;
  }
  return NULL;
}
```

Question 8 (6 points)

Ecrivez une fonction permettant d'afficher l'ensemble des clients d'un arbre binaire de recherche par ordre croissant de leur numéro de client selon le format vu précédemment. Vous pourrez utiliser la fonction `ecrire_client` vue précédemment. Il est rappelé que `stdout` est un `FILE *` correspondant à la sortie standard (l'écran du terminal). Prototype :

```
void afficher_clients(Abr_Client *abr);
```

Solution:

```
void afficher_clients(Abr_Client *abr) {
  if (abr) {
    afficher_clients(abr->gauche);
    ecrire_client(stdout,abr->data);
    afficher_clients(abr->droit);
  }
}
```

Question 9 (6 points)

Ecrivez une fonction permettant d'ajouter un client en place dans une liste triée par ordre croissant des numéros. Cette fonction transmettra la liste par pointeur (cette liste peut être vide). Le client à ajouter est le second paramètre de la fonction. Vous pourrez utiliser des fonctions fournies ou vues précédemment.

Prototype :

```
void ajouter_liste(Liste_Client **l,Client *c);
```

Solution:

```
void ajouter_liste(Liste_Client **l,Client *c) {
  Liste_Client *lc=creer_element(c);
  if (!(*l)) {
    *l=lc;
    return;
  }
  if ((*l)->data->num>c->num) {
    lc->suivant=(*l);
    *l=lc;
    return;
  }
  Liste_Client *ltmp=*l;
  while(ltmp->suivant) {
    if (ltmp->suivant->data->num>c->num) {
      lc->suivant=ltmp->suivant;
      ltmp->suivant=lc;
      return;
    }
    ltmp=ltmp->suivant;
  }
  ltmp->suivant=lc;
}
```

Question 10 (6 points)

La création de l'arbre binaire de recherche équilibré se fera par dichotomie : les clients contenus dans une liste triée `l` seront divisés en deux listes `l1` et `l2` et une valeur pivot. Le pivot correspond au centre de la liste. La liste `l1` correspond aux éléments avant le pivot (les numéros inférieurs à celui du pivot) et la liste `l2` correspond aux éléments après le pivot (les numéros supérieurs au pivot). Cette procédure

garantira un découpage équilibré à chaque étape entre les sous-arbres gauche et droit. Les éléments de liste chaînée portant les clients devront être détruits au fur et à mesure de la construction de l'arbre.

Un début de code a été écrit par un de vos collègues. Complétez ce code pour créer un arbre binaire de recherche équilibré. Vous pourrez utiliser des fonctions fournies ou vues précédemment.

```
Abr_Client *creer_abr_equilibre(Liste_Client *l) {
  if (!l) return NULL;
  int len=longueur_liste(l);
  if (len==1) {
```

Solution:

```
    Abr_Client *abr=creer_abr(l->data,NULL,NULL);
    free(l);
    return abr;
```

```
  }
  int i=1;
  Liste_Client *l1=l;
  while (i<len/2) {
    l=l->suivant;
    i++;
  }
  Client *pivot=l->suivant->data;
  Liste_Client *l2=l->suivant->suivant;
```

Solution:

```
  free(l->suivant);
  l->suivant=NULL;
  return creer_abr(pivot,creer_abr_equilibre(l1),creer_abr_equilibre
     (l2));
```

```
}
```

Question 11 (6 points)

Ecrivez une fonction `main` permettant de tester ces fonctions. Cette fonction lira le contenu d'un fichier *clients.txt* et ajoutera les clients lus à une liste de clients (triée par ordre croissant des numéros). Elle créera ensuite un arbre binaire équilibré et l'affichera à l'écran, sous la forme de la liste des numéros par ordre croissant, puis sous la forme d'un arbre. Vous prendrez, naturellement, soin de libérer toute la mémoire allouée et vous pourrez utiliser des fonctions fournies ou vues précédemment.

Solution:

```
#include "client.h"

int main(void) {

  Abr_Client *abr=NULL;

  FILE *f=fopen("clients.txt","r");
```

```
  Liste_Client *lc=NULL;
  Client *c=lire_client(f);
  while (c)  {
    ajouter_liste(&lc,c);
    c=lire_client(f);
  }

  abr=creer_abr_equilibre(lc);
  afficher_clients(abr);
  printf("\n");
  afficher_abr(abr);
  printf("\n");
  liberer_abr(abr);
  fclose(f);
  return 0;
}
```

Mémento de l'UE 2I001

Ce document a pour vocation de présenter un bref descriptif des fonctions offertes par les bibliothèques standards et qui sont susceptibles d'être utilisées dans l'UE.

Entrées - sorties

Prototypes disponibles dans `stdio.h`.

Entrées, sorties formatées

```
int printf(const char *format, ...);
```

La fonction `printf` écrit sur la sortie standard (par défaut le terminal), un message correspondant au texte spécifié dans la chaîne `format`. Le texte affiché peut contenir des portions *variables* spécifiées par des codes de conversion précédés par le caractère `%`. Les arguments suivant `format` doivent correspondre (ordre et type) aux codes de conversion présents. Voici quelques codes de conversion courants :

- `%d` : entier
- `%c` : caractère
- `%s` : chaîne de caractères
- `%f` : nombre réel

`printf` renvoie le nombre de caractères imprimés et la chaîne de format peut contenir des codes de contrôle permettant le formatage comme `\n` pour forcer le passage à la ligne et `\t` pour insérer une tabulation.

```
int scanf (const char *format, ...);
int sscanf(const char *entree, const char *format, ...);
```

Les fonctions `scanf` et `sscanf` permettent de saisir et analyser un texte saisi sur l'entrée standard, par défaut le clavier (`scanf`) ou depuis une chaîne de caractères passée en argument (`sscanf`). Le texte devra respecter le `format` spécifié et les arguments suivants doivent correspondre à des pointeurs sur des variables de type appropriés. Les codes de conversion son identiques à ceux de `printf`.

Entrées, sorties caractères

```
int getchar(void);
```

Lit un caractère dans le flux d'entrée standard (par défaut le clavier). La fonction retourne le code **EOF** en cas d'erreur, le caractère lu dans le cas contraire.

```
int putchar(int c);
```

Affiche un caractère dans le flux de sortie standard (par défaut le terminal). La fonction retourne le code **EOF** en cas d'erreur, le caractère lu dans le cas contraire.

```
int puts(const char *s);
```

Affiche une chaîne de caractères dans le terminal et passe à la ligne, renvoie **EOF** en cas d'erreur.

Fichiers

Prototypes disponibles dans `stdio.h`.

```
FILE *fopen(const char *path, const char *mode);
```

Ouvre un fichier dont le chemin est spécifié par la chaîne `path` et retourne un pointeur de type `FILE *` (`NULL` en cas d'échec). L'argument `mode` permet de spécifier le type d'accès à réaliser sur le fichier :

- `[r]` pour un accès en lecture,
- `[w]` pour un accès en écriture et le contenu précédent du fichier est écrasé,
- `[a]` pour un accès en écriture, le contenu du fichier est préservé et les écritures sont effectuées à la suite des contenus déjà présents.

En cas d'erreur la fonction retourne le code **NULL** sinon un pointeur vers le fichier ouvert.

```
int fclose(FILE *fp);
```

Cette fonction provoque la fermeture du fichier pointé par `fp`. En cas d'erreur la fonction retourne le code **EOF** sinon `0`.

```
int fprintf(FILE *stream, const char *format, ...);
```

Identique à `printf` mais l'argument `stream` permet de spécifier le flux de sortie.

```
int fscanf(FILE *stream, const char *format, ...);
```

Identique à `scanf` mais l'argument `stream` permet de spécifier le flux d'entrée.

```
int fputc(int c, FILE *stream);
```

Identique à `putchar` mais l'argument `stream` permet de spécifier le flux de sortie.

```
int fputs(const char *s, FILE *stream);
```

Identique à `puts` mais l'argument `stream` permet de spécifier le flux de sortie et il n'y a pas d'ajout de passage à la ligne.

```
int getc(FILE *stream);
```

Identique à `getchar` mais l'argument `stream` permet de spécifier le flux d'entrée.

```
char *fgets(char *s, int size, FILE *stream);
```

Lit au plus `size` octets dans le flux `stream`. La lecture s'arrête dès qu'un passage à la ligne est rencontré. Les octets lus sont stockés dans `s`. La fonction retourne `s` en cas de succès et `NULL` en cas d'erreur.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Lecture binaire de `nmemb` éléments de `size` octets dans le fichier `stream`. Les données lues sont stockées en mémoire à partir de l'adresse `ptr`. La fonction retourne le nombre d'éléments effectivement lus.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

Écriture de `nmemb` éléments de `size` octets dans le fichier `stream`. Les données à écrire sont lues en mémoire à partir de l'adresse `ptr`. La fonction retourne le nombre d'éléments effectivement écrits.

Chaînes de caractères

Prototypes disponibles dans `string.h`.

Une chaîne de caractères correspond à un tableau de caractère et doit contenir un marqueur de fin `\0`.

```
size_t strlen (const char *s);
```

Renvoie la longueur d'une chaîne de caractères (marqueur de fin `\0` non compris).

```
int strcmp (const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

Comparaison entre chaînes de caractères éventuellement limité aux `n` premiers caractères. La valeur retournée est :

- 0 si les deux chaînes sont identiques,
- négative si `s1` précède `s2` dans l'ordre lexicographique (généralisation de l'ordre alphabétique),
- positive sinon.

```
char *strcpy (char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

Copie le contenu de la chaîne `src` dans la chaîne `dest` (marqueur de fin `\0` compris). La chaîne `dest` doit avoir précédemment été allouée. La copie peut être limitée à `n` caractères et la valeur retournée correspond au pointeur de destination `dest`.

```
void *memcpy(void *dest, const void *src, size_t n);
```

Copie `n` octets à partir de l'adresse contenue dans le pointeur `src` vers l'adresse stockée dans `dest`. `dest` doit pointer vers une zone mémoire préalablement allouée et de taille suffisante. `memcpy` renvoie la valeur de `dest`.

```
size_t strlen(const char *s);
```

Retourne le nombre de caractères de la chaîne `s` (marqueur de fin `\0` non compris).

```
char *strdup(const char *s);
```

Cette fonction permet de dupliquer une chaîne de caractères, elle retourne un pointeur vers la chaîne nouvellement allouée. La nouvelle chaîne pourra être libérée avec la fonction `free`.

```
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

Ajoute la chaîne `src` à la suite de la chaîne `dst`. La chaîne `dest` devra avoir été allouée et être de taille suffisante. La fonction retourne `dest`.

```
char *strstr(const char *haystack, const char *needle);
```

La fonction renvoie un pointeur sur la première occurrence de la sous-chaîne `needle` rencontrée dans la chaîne `haystack`. Si la chaîne recherchée n'est pas présente, la fonction retourne `NULL`.

Conversion de chaînes de caractères

Prototypes disponibles dans `stdlib.h`.

```
int atoi(const char *nptr);
```

La fonction convertit le début de la chaîne pointée par `nptr` en un entier de type `int`.

```
double atof(const char *nptr);
```

Cette fonction convertit le début de la chaîne pointée par `nptr` en un `double`.

```
long int strtol(const char *nptr, char **endptr, int base);
```

Convertit le début de la chaîne `nptr` en un entier long. l'interprétation tient compte de la `base` et la variable pointée par `endptr` est affectée avec l'adresse du premier caractère invalide (au sens de la conversion).

Allocation dynamique de mémoire

Prototypes disponibles dans `stdlib.h`.

```
void *malloc(size_t size);
```

Alloue `size` octets de mémoire et retourne un pointeur générique correspondant à l'adresse du premier octet de la zone, renvoie `NULL` en cas d'échec.

```
void *realloc(void *ptr, size_t size);
```

Permet de modifier la taille d'une zone de mémoire allouée dynamiquement. `ptr` doit correspondre à l'adresse du premier octet de la zone précédemment allouée par `malloc` ou `realloc`. `size` correspond à la taille en octet de la nouvelle zone allouée. `realloc` garantie que la nouvelle zone contiendra les données présentes dans la zone initiale.

```
void free(void *ptr);
```

Libère une zone mémoire allouée dynamiquement. `ptr` doit correspondre à l'adresse du premier octet de la zone précédemment allouée par `malloc` ou `realloc`.

La liste des fonctions du programme considéré est indiquée ci-après. Certaines fonctions ne sont pas à écrire. Ces fonctions peuvent tout de même être utilisées et considérées comme disponibles.

```
typedef struct _Date {
  unsigned char jour;
  unsigned char mois;
  unsigned int annee;
} Date;

typedef struct _Achat {
  Date date;
  char *objet;
  float montant;
  struct _Achat *suivant;
} Achat;

typedef struct _Client {
  int num;
  char * nom;
  Achat *l_achat; /* liste des achats d'un
     client */
} Client;

/* Structure pour stocher un ensemble de
   clients sous la forme d'un arbre */
typedef struct _Abr_Client {
  Client *data;
  struct _Abr_Client *gauche;
  struct _Abr_Client *droit;
} Abr_Client;

/* Structure pour stocher un ensemble de
   clients sous la forme d'une liste */
typedef struct _Liste_Client {
  Client *data;
  struct _Liste_Client *suivant;
} Liste_Client;

/* Cree un achat et initialise ses champs a
   partir les valeurs passees en argument (a
   ecrire)*/
Achat *creer_achat(Date d,char *objet, float
   montant);

/* Cree un client et initialise ses champs a
   partir des valuers passees en argument */
Client *creer_client(int num, char *nom);

/* Ajouter un achat a la liste d'achats d'un
   client  (a ecrire) */
void ajouter_achat(Client *c, Achat *a);

/* Calcule le montant des achats d'un client  (
   a ecrire)*/
float montant_achats(Client *c);

/* Ecrit un achat (seul, sans ecrire les
   suivants) dans un flux f  (a ecrire)*/
void ecrire_achat(FILE *f, Achat *a);

/* Ecrit un client et tous ses achats dans un
   flux f  (a ecrire)*/
void ecrire_client(FILE *f, Client *c);

/* Lit un client depuis un fichier. Renvoie
   NULL si la lecture a echoue  (a ecrire)*/
Client *lire_client(FILE *f);

/* Cree un arbre binaire de recherche et
   initialise ses champs a partir des arguments
```

```
    */
Abr_Client *creer_abr(Client *c, Abr_Client *
   gauche, Abr_Client *droit);

/* Recherche un client a partir de son numero
   dans un arbre binaire de recherche  (a
   ecrire)*/
Client *rechercher_client(Abr_Client *abr, int
   num);

/* Affiche l'arbre des numeros des clients d'un
    arbre binaire de recherche de la facon
   suivante (racine (gauche ...) (droit ...))
   */
void afficher_abr(Abr_Client *abr);

/* Affiche les clients par ordre croissant de
   leur numero  (a ecrire)*/
void afficher_clients(Abr_Client *abr);

/* Libere toute la memoire associee a un client
    */
void liberer_client(Client *c);

/* Libere un achat (seul) */
void liberer_achat(Achat *a);

/* Libere une liste d'achats */
void liberer_achats(Achat *l);

/* Libere un arbre binaire de recherche (
   clients inclus) */
void liberer_abr(Abr_Client *abr);

/* Cree un element de liste chainee */
Liste_Client *creer_element(Client *c);

/* Ajoute un client dans une liste triee par
   ordre croissant des numeros  (a ecrire) */
void ajouter_liste(Liste_Client **l,Client *c);

/* Determine la longueur d'une liste de clients
    */
int longueur_liste(Liste_Client *l);

/* Cree un arbre binaire de recherche equilibre
    a partir d'une liste  (a ecrire) */
Abr_Client *creer_abr_equilibre(Liste_Client *l
   );

/* Cherche dans la liste l un achat ayant les
   date, objet et montant indiques. Renvoie
   NULL si aucun achat correspondant n'a ete
   trouve. */
Achat *chercher_achat(Achat *l, Date d, char *
   objet, float montant);

/* Supprime les doublons dans une liste d'
   achats associee a un client c  (a ecrire)*/
void supprimer_doublons(Client *c);
```