Jean Rouge
Mathis Ranc

CR TP CS353

Introduction :

Nous sommes le responsable informatique d'une startup de la téléphonie mobile, et sommes missionnés pour réaliser le système de facturation.
Le problème est le suivant :
Les équipements du réseau alimentent tout au long du mois un fichier de log, qui trace tous les appels réalisés (une ligne de log contient le numéro de téléphone, la date et le coût de l'appel en centimes d'euros) à la fin du mois, le fichier de log est envoyé au service facturation. Ce fichier doit être traité par un programme, qui doit produire un tableau à deux colonnes : la colonne 1 contient les numéros de téléphone, la colonne 2 contient le montant total des communications. Le tableau doit être trié par ordre croissant de numéro de téléphone.
Votre société dispose de 20 000 clients qui réalisent une centaine d'opérations par mois (une opération peut être un appel ou un envoi de SMS). Le fichier de log contient donc environ 2 millions de lignes.

TP1 : Implémentation naïve - Liste chaînée

Question 1

La structure contenant les informations d'un client, et qui permettra de réaliser la liste chaînée, est la suivante :

```c
typedef struct Client {
    int numero;
    int prixAppel;
    int nbAppel;
    struct Client * suivant;
} client;
```

Fonction créant un nouveau client :

```c
//Creation d'un nouveau client
client *createClient(int numero, int nbAppel, int prixAppel) {
    client *NewClient = malloc(sizeof(client));
    NewClient->numero = numero;
    NewClient->nbAppel = nbAppel;
    NewClient->prixAppel = prixAppel;
    return NewClient;
}
```

Question 2

Insertion d'une ligne de log dans la liste chaînée :

```c
//Update les log
int addLogLine(struct Client **liste, int numero, int prixAppel) {
    client *ptr = *liste;
    client *ptr_prec = NULL;

    while(ptr != NULL && ptr->numero < numero){ //Tant qu'on est "avant" le numero recherche
        ptr_prec = ptr;
        ptr = ptr->suivant; //passage a l'element suivant
    }

    if(ptr_prec == NULL){   //Insertion en tete
        client *NewClient = createClient(numero, 1, prixAppel); //Creation d'un client qui a passe un appel
        *liste = NewClient; //Tete[liste] = NewClient
        NewClient->suivant = ptr;
        return 1;
    }
    if (ptr == NULL || ptr->numero > numero){   //Insertion en milieu ou fin de liste
        client *NewClient = createClient(numero, 1, prixAppel);
        ptr_prec->suivant = NewClient;
        NewClient->suivant = ptr;
    }
    /*if (ptr->numero == numero)*/else{ //le numero existe dans la liste de logs: update du log
        ptr->nbAppel++;
        ptr->prixAppel += prixAppel;
    }
    return 1;
}
```

Question 3

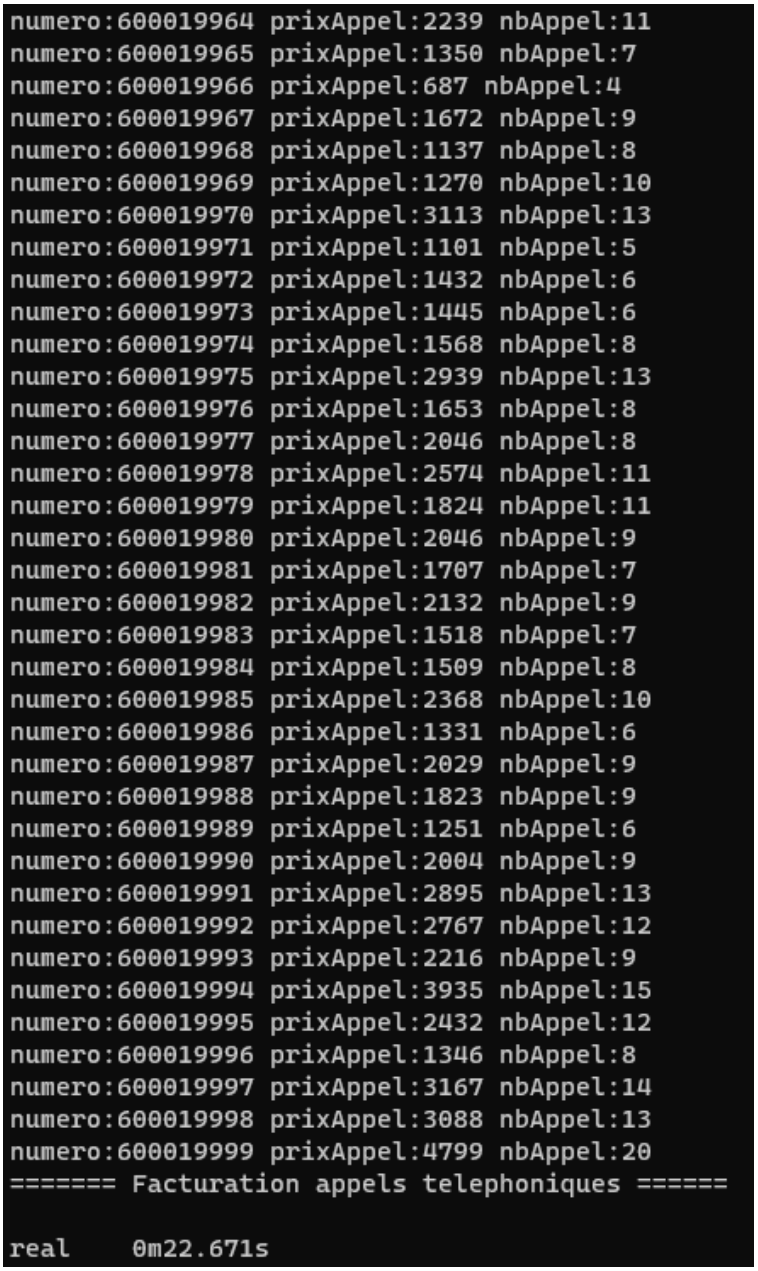
Affichage du contenu de la liste :

```c
//Affich tous la liste de log. Pour chaque client, affiche son numero, le prix total de ses appels et son nombre d'appels
void dumpListe(struct Client *liste) {
    client *ptr = liste;
    while (ptr != NULL){
        printf("numero:%d prixAppel:%d nbAppel:%d\n", ptr->numero, ptr->prixAppel, ptr->nbAppel);
        ptr = ptr->suivant;
    }
}
```

On crée aussi une fonction permettant de "free" tous les clients une fois les tests effectués :

```c
//Free tous les clients créés
void freeEverything(struct Client *liste){
    if (liste==NULL){
        return;
    }
    freeEverything(liste->suivant);
    free(liste);
}
```

A l'aide du code de test fourni dans le sujet (et en ajoutant notre fonction **freeEverything(liste)** à la fin), on calcule le temps d'exécution à l'aide de la commande UNIX time (test effectué sur un ordinateur personnel) :



Pour 200 000 lignes de log, il nous faut environ 22s pour remplir la liste chaînée. Ce temps est déjà trop long, et explosera pour 2 millions de lignes de log (étant donné que le temps de recherche est en O(n), il nous faudra 10 fois plus de temps pour chercher une ligne).
Cette solution n'est donc pas envisageable en situation réelle.

TP2 : Les arbres binaires de recherche standard

1.1 Introduction

En effet les performances avec une liste chainée ne sont pas bonnes.
On utilise donc un arbre binaire de recherche qui à une complexité de recherche O(log(n)).

1.3 Questions

Question 1

Notre structure de Noeud d'arbre

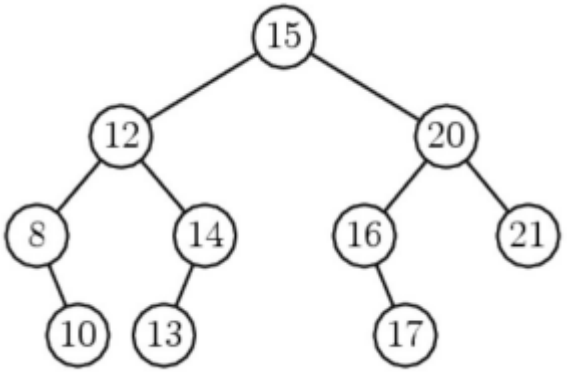
```c
typedef struct Client {
    int numero;
    int prixAppel;
    int nbAppel;
    struct Client * gauche;
    struct Client * droite;

} client;
```

Et voilà la fonction pour creerClient

```c
struct Client * creerClient(int numeroTel, int nbAppel,int cout)
{
    client * nouveau = malloc( sizeof(client) );
    nouveau->numero=numeroTel;
    nouveau->nbAppel=nbAppel;
    nouveau->prixAppel=cout;
    nouveau->gauche=NULL;
    nouveau->droite=NULL;
    return nouveau;
}
```

Question 2

On crée une fonction qui construit un arbre de test correspondant à celui-là:



```c
struct Client *CreateSampleTree(){
    client *racine = creerClient(15,0,0);
        racine->gauche = creerClient(12,0,0);
            racine->gauche->gauche = creerClient(8,0,0);
                racine->gauche->gauche->droite = creerClient(10,0,0);
            racine->gauche->droite = creerClient(14,0,0);
                racine->gauche->droite->gauche = creerClient(13,0,0);
        racine->droite = creerClient(20,0,0);
            racine->droite->gauche = creerClient(16,0,0);
                racine->droite->gauche->droite = creerClient(17,0,0);
            racine->droite->droite = creerClient(21,0,0);
    return racine;
}
```

Question 3

Parcours infixe: pour traiter un Noeud, on traite d'abord son fils gauche, puis le Noeud lui-même, puis son fils droit.

```c
void parcourirInfixe(struct Client * abr)
{
    if(abr){
        parcourirInfixe(abr->gauche);
        printf("%d: %d\n",abr->numero, abr->nbAppel);
        parcourirInfixe(abr->droite);
    }
}
```

```c
void test_Q3(){
    printf("-- Test parcours infixe --\n");
    client*root = CreateSampleTree();
    parcourirInfixe(root);
    purgeTree(root);
    printf("-------------------------\n");
}
```

```
-- Test parcours infixe --
8: 0
10: 0
12: 0
13: 0
14: 0
15: 0
16: 0
17: 0
20: 0
21: 0
-------------------------
```

On voit bien que les nœuds sont lus en ordre croissant.

Question 4

Fonction de recherche d'un Noeud dans l'arbre.

```c
struct Client * chercher(struct Client * abr,int numeroTel)
{
    while( !(abr==NULL || numeroTel==abr->numero) ){
        if( numeroTel > abr->numero)
            abr = abr->droite;
        else
            abr = abr->gauche;
    }
    //On dit que la fonction return NULL si le numero n'est pas trouvé
    return abr;
}
```

Cette fonction renvoie un pointeur vers le nœud trouvé, NULL sinon.

```c
void test_Q4(){
    printf("-- Test Chercher Noeud ---\n");
    client *noeud,*root = CreateSampleTree();
    if((noeud = chercher(root,13)))
        printf("Noeud [%d]: %d\n",noeud->numero,noeud->prixAppel);
    purgeTree(root);
    printf("-------------------------\n");
}
```

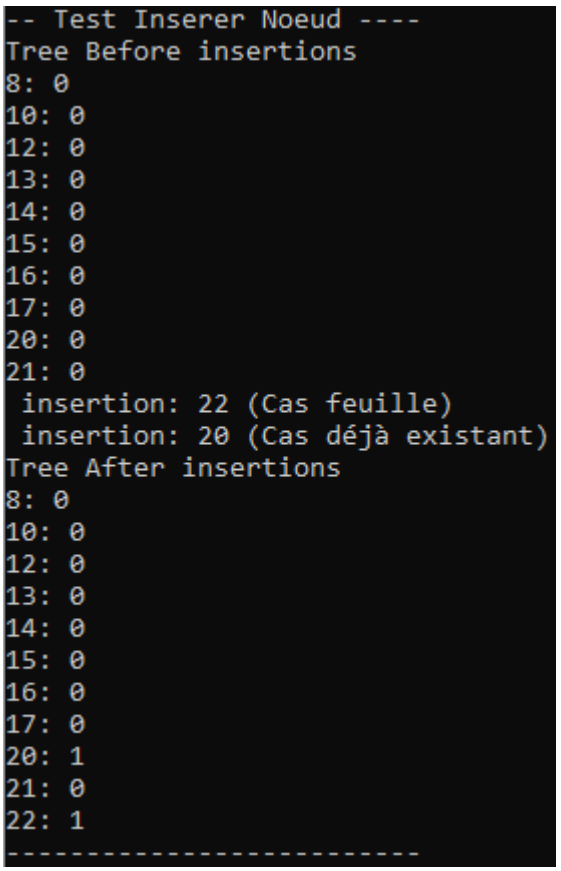
```
-- Test Chercher Noeud ---
Noeud [13]: 0
-------------------------
```

Question 5

Fonction insérer noeud.

```c
struct Client *inserer(struct Client ** abr, int numeroTel, int prixAppel)
{
    if (*abr==NULL){
        (*abr)=creerClient(numeroTel,1,prixAppel);
    } else {
        client *ptr = *abr;
        client *suiv = ptr;
        do {
            ptr = suiv;
            if( numeroTel > ptr->numero)
                suiv = ptr->droite;
            else
                suiv = ptr->gauche;
        }while( !(suiv==NULL || numeroTel==ptr->numero) );

        if(numeroTel==ptr->numero){
            ptr->nbAppel += 1;
            ptr->prixAppel += prixAppel;
        }else{
            suiv = creerClient(numeroTel,1,prixAppel);
            if( numeroTel > ptr->numero)
                ptr->droite = suiv;
            else
                ptr->gauche = suiv;
        }
    }
    return *abr ;
}
```

```c
void test_Q5(){
    printf("-- Test Inserer Noeud ----\n");
    client *root = CreateSampleTree();
    printf("Tree Before insertions\n");
    parcourirInfixe(root);
    printf(" insertion: 22 (Cas feuille)\n");
    inserer(&root,22,0);
    printf(" insertion: 20 (Cas déjà existant)\n");
    inserer(&root,20,0);
    printf("Tree After insertions\n");
    parcourirInfixe(root);
    printf("-------------------------\n");
}
```



On remarque que les nœuds insérés ont leurs nombre d'appels initialisés à 1.
En temps normal, on veut insérer un client seulement si on lit son nom dans les logs d’appels, donc son nombre d'appels est bien égal à 1.

Question 6

Suppression d’un noeud dans l’arbre.

```
75  struct Client *supprimerClient(struct Client ** abr, int numeroTel)
76  {
77      if(!abr) return *abr ;
78
79      client *ptr = *abr;
80      client *preced;
81
82      while( !(ptr==NULL || numeroTel==ptr->numero) ){
83          preced = ptr;
84          if( numeroTel > ptr->numero)
85              ptr = ptr->droite;
86          else
87              ptr = ptr->gauche;
88      }
89
90      // Cas element non trouvé
91      if(ptr==NULL) return *abr;
92
93      if(numeroTel==ptr->numero) {
94          //Cas 0 fils
95          if( !(ptr->gauche || ptr->droite)){
96              //Cas suppr racine
97              if(ptr==preced){
98                  *abr = NULL;
99              } else {
100                 if( numeroTel > preced->numero)
101                     preced->droite=NULL;
102                 else
103                     preced->gauche=NULL;
104             }
105             free(ptr);
106         }
107         //Cas 2 fils
108         else if(ptr->gauche && ptr->droite){
109             client *tmp= ptr->droite;
110             while(tmp->gauche) tmp = tmp->gauche;
111             int num        = tmp->numero;
112             ptr->prixAppel= tmp->prixAppel;
113             ptr->nbAppel   = tmp->nbAppel;
114             supprimerClient(&(ptr),num);
115             ptr->numero    = num;
116         }
117         //Cas 1 fils
118         else {
119             client* tmp;
120             if(ptr->gauche)
121                 tmp = ptr->gauche;
122             else
123                 tmp = ptr->droite;
124             ptr->numero    = tmp->numero;
125             ptr->prixAppel= tmp->prixAppel;
126             ptr->nbAppel   = tmp->nbAppel;
127             ptr->gauche    = tmp->gauche;
128             ptr->droite    = tmp->droite;
129             free(tmp);
130         }
131     }
132     return *abr ;
133 }
```

```c
void test_Q6(){
    printf("-- Test Supprimer Noeud --\n");
    client *root = CreateSampleTree();
    printf("Tree Before suppressions\n");
    parcourirInfixe(root);
    printf(" suppression: 10 (Cas 0 fils)\n");
    supprimerClient(&root,10);
    printf(" suppression: 16 (Cas 1 fils)\n");
    supprimerClient(&root,16);
    printf(" suppression: 20 (Cas 2 fils)\n");
    supprimerClient(&root,20);
    printf(" suppression: 15 (Cas racine)\n");
    supprimerClient(&root,15);
    printf("Tree After suppressions\n");
    parcourirInfixe(root);
    printf("-------------------------\n");
}
```

```
-- Test Supprimer Noeud --
Tree Before suppressions
8: 0
10: 0
12: 0
13: 0
14: 0
15: 0
16: 0
17: 0
20: 0
21: 0
 suppression: 10 (Cas 0 fils)
 suppression: 16 (Cas 1 fils)
 suppression: 20 (Cas 2 fils)
 suppression: 15 (Cas racine)
Tree After suppressions
8: 0
12: 0
13: 0
14: 0
17: 0
21: 0
-------------------------
```

Question 7

Programme de facturation.

```c
int main() {
    client *liste=NULL;

    int i;
    int numeroTel;
    int prixAppel;

    // Aide au calcul du pourcentage d'avancement
    int pas = NBLOGLINE/100;
    for(i=0;i<NBLOGLINE;i++)
    {

        // Génération d'un numéro de telephone portable
        numeroTel = 600000000+(rand() % NBCLIENT);

        // Donne un prix d'appel compris entre 0.01 et 4 euros
        prixAppel = (rand() % 400)+1;

        // Ajout de cette ligne de log dans la liste des clients
    if (inserer(&liste ,numeroTel,prixAppel)==NULL) break;

        // Affichage du pourcentage d'avancement
        if ((i % pas)==0)
        {
            printf("Done  = %02d %%...\r",i/pas);
        }
    }
    printf("\n");

    printf("****** Facturation appels telephoniques ******\n");

    parcourirInfixe(liste);

    printf("****** Suppression de la facturation appels telephoniques ******\n");

    /** J'ai commenté car il y a marqué dans le sujet que supprmierClient
        renvoie un pointeur vers la racine de l'arbre et comme le code
        ci-dessous free cette racine, ça crée des problèmes.
        */
    /*for (i=0;i<NBCLIENT;i++) {
    purgeTree(liste);
    printf("****** Fin Facturation appels telephoniques ******\n");
}
```

On affiche tous les clients dans l’ordre avec parcourirInfixe().

```
****** Suppression de la facturation appels telephoniques ******
****** Fin Facturation appels telephoniques ******

real    0m0.979s
user    0m0.188s
sys     0m0.344s
```

Le programme de facturation s'exécute en 1 seconde, comparé à 22s pour le TP1 avec la liste chaînée. Ce qui correspond à une amélioration de 2 200% !!!!

C’est très rapide.

TP3 : Table de hachage

1.3 Questions

Question 1

```
35  /*----------------------------------------------------------------------------
36   * Cette fonction calcule la valeur de hachage pour le produit itemCode
37   *----------------------------------------------------------------------------*/
38  uint32_t hashkey(uint32_t itemCode,uint32_t nbTry)
39  {
40      return (((itemCode%TABLE_SIZE)+nbTry*(1+itemCode%(TABLE_SIZE -1)))%TABLE_SIZE);
41  }
```

Fonction de hachage

Question 2

```
43  /*----------------------------------------------------------------------------
44   * Cette fonction insère le produit indiqué dans la table de hachage.
45   * Si le produit est inséré avec succès, alors la fonction retourne SUCCESS (0)
46   * Si le produit existe déjà dans la table, alors la fonction retourne INSERT_ALREADY_EXIST (-1),
47   * et la table de hachage n'est pas modifiée
48   * Si la table est pleine, alors la fonction retourne TABLE_FULL (-2).
49   *----------------------------------------------------------------------------*/
50  int insertItem(uint32_t itemCode, char* itemName, float itemPrice)
51  {
52    uint32_t i=0;
53    uint32_t key=hashkey(itemCode,i);
54    uint32_t first_deleted=TABLE_SIZE;//impossible index
55    while( hash_table[key].status!=NULL_ITEM && i<TABLE_SIZE ){
56
57      /* Cas où on à trouvé l'objet */
58      if( hash_table[key].status==USED_ITEM && hash_table[key].code==itemCode ){
59        return(INSERT_ALREADY_EXIST);
60      } else
61
62      /* Cas où on trouve le premier objet supprimé dans la chaine */
63      if( hash_table[key].status==DELETED_ITEM && first_deleted==TABLE_SIZE ){
64        first_deleted=key;
65      }
66
67      //Essai suivant
68      i++;
69      key=hashkey(itemCode,i);
70    }
71
72    /* Cas où la table est pleine */
73    if(hash_table[key].status==NULL_ITEM){
74      /* la cle cible le 1er suppr */
75      if(first_deleted<TABLE_SIZE) key=first_deleted;
76
77      /* remplir le [key] */
78          hash_table[key].status = USED_ITEM ;
79          hash_table[key].code   = itemCode  ;
80  strncpy(hash_table[key].name   , itemName  , ITEM_NAME_SIZE);
81          hash_table[key].price  = itemPrice ;

90      return SUCCESS;
91    }
92
93    /* Cas où la table est pleine */
94    return TABLE_FULL;
95  }
```

Fonction d'insertion

Question 3

```
/*-------------------------------------------------------------------------
 * fonction de suppression d'un produit du magasin
 * Si le produit est supprimé avec succès, alors la fonction retourne SUCCESS (0)
 * Si le produit n'existe pas, alors la fonction retourne DELETE_NO_ROW (-4)
 *-------------------------------------------------------------------------*/
int suppressItem(unsigned int itemCode)
{
  uint32_t i=0;
  uint32_t key=hashkey(itemCode,i);
  while( hash_table[key].status!=NULL_ITEM && i<TABLE_SIZE ){
    /* Si on trouve l'objet */
    if( hash_table[key].status==USED_ITEM && hash_table[key].code==itemCode ){
      hash_table[key].status=DELETED_ITEM;
      return(SUCCESS);
    }
    //Sinon essai suivant
    key=hashkey(itemCode,++i);
  }
  return DELETE_NO_ROW;
}
```

Suppression d'un produit

Question 4

```
/*-------------------------------------------------------------------------
 * Pour chaque produit, cette fonction affiche sur une ligne
 * le code du produit
 * son libellé
 * son prix
 * son index dans la table de hashage
 * sa valeur de hash
 *-------------------------------------------------------------------------*/
void dumpItems()
{
  printf("┌──────────┬────────────────────────────────┬───────┬─────┐\n");
  printf("│%-10s│%-32s│PRIX   │INDEX│\n", "CODE", "LIBELLE");
  printf("├──────────┼────────────────────────────────┼───────┼─────┤\n");
  for(int i=0; i<TABLE_SIZE; i++){
    if(hash_table[i].status==USED_ITEM)
    printf("│%-10d│%-32s│%-7.2f│%-5d│\n"
        ,hash_table[i].code
        ,hash_table[i].name
        ,hash_table[i].price
        ,i);
  }
  printf("└──────────┴────────────────────────────────┴───────┴─────┘\n");
}
```

Affichage dans un tableau

Question 5

```c
/*-------------------------------------------------------------------------
 * Cette fonction trouve le produit dont le code est itemCode.
 * Cette fonction retourne NULL si le produit n'existe pas.
 * Cette fonction retourne un pointeur vers le produit si le produit existe.
 *-------------------------------------------------------------------------*/
Item *getItem(unsigned int itemCode)
{
  uint32_t i=0;
  uint32_t key=hashkey(itemCode,i);
  while( hash_table[key].status!=NULL_ITEM && i<TABLE_SIZE ){
    /* Si on trouve l'objet */
    if( hash_table[key].status==USED_ITEM && hash_table[key].code==itemCode )
      return &(hash_table[key]);
    //Sinon essai suivant
    key=hashkey(itemCode,++i);
  }
  return NULL;
}
```

Fonction de recherche d'un produit donné

Question 6

```c
/*-------------------------------------------------------------------------
 *  fonction de mise à jour d'un produit :
 * Si le produit est mis à jour avec succès, alors la fonction retourne SUCCESS (0)
 * Si le produit n'existe pas, alors la fonction retourne UPDATE_NO_ROW (-5)
 *-------------------------------------------------------------------------*/
int updateItem(unsigned int itemCode, char *itemName, float itemPrice)
{
    Item *item = getItem(itemCode);
    int ret = UPDATE_NO_ROW;
    if(item){
        item->code  = itemCode ;
strncpy(item->name  , itemName , ITEM_NAME_SIZE);
        item->price = itemPrice;
        ret = SUCCESS;
    }
    return ret;
}
```

Modification du prix d'un item existant

Question 7

```
/*-------------------------------------------------------------------------
 * la fonction de réorganisation in situ:
 *-------------------------------------------------------------------------*/
void rebuildTable()
{
  // On salit toutes les entrées de la table
  for(int i=0; i<TABLE_SIZE; i++) hash_table[i].dirty = DIRTY;
  uint32_t j,key,done;
  Item to_find,tmp;
  for(int i=0; i<TABLE_SIZE; i++) {

    if(hash_table[i].status==USED_ITEM){
      to_find = hash_table[i];
      //clean
      hash_table[i].status = NULL_ITEM;
      done=0;
      while(!done){
        j=0;
        key=hashkey(to_find.code,j);
        while( hash_table[key].dirty!=DIRTY && j<TABLE_SIZE )
           key=hashkey(to_find.code,++j);

        /* Si on trouve l'objet */
        if( hash_table[key].dirty==DIRTY ){
          tmp=hash_table[key];
          hash_table[key].status=NULL_ITEM;
          /* remplir le [key] */
         hash_table[key]=to_find;
          hash_table[key].dirty = CLEAN;
        } else printf("ERREUR in rebuild table");

        // Si il y a un nouvel item à placer
        if(tmp.status==USED_ITEM){
          to_find=tmp;
        } else
          done = 1;
      }

    }
  }
}
```

Rebuild in situ

TP4 : Table de hachage (suite)

Question 8

```
Result *findItem(char* itemName){
    Result * liste = malloc(sizeof(Result));
    Result *preced = NULL;
    Result *ptr = liste;

    for(int i=0; i<TABLE_SIZE; i++)
    if(hash_table[i].status==USED_ITEM && !strcmp(itemName,hash_table[i].name)){
        ptr->item=&hash_table[i];
        ptr->next=malloc(sizeof(Result));
        preced = ptr;
        ptr=ptr->next;
    }

    if(ptr==liste)
        liste=NULL;
    else
        preced->next=NULL;

    free(ptr);

    return liste;
}
```

Recherche par libellé naïve
La complexité est en O(n).

Question 9

```
Result *findItemWithIndex(char* itemName){
    Result * liste = malloc(sizeof(Result));
    Result *preced = NULL;
    Result *ptr = liste;
    unsigned int Code = hashIndex(itemName, strlen(itemName));

    uint32_t i=0;
    uint32_t key=hashkey(Code,i);
    while( item_hash_table[key]->status!=NULL_ITEM && i<TABLE_SIZE ){
        if(item_hash_table[key]->status==USED_ITEM && !strcmp(itemName,item_hash_table[key]->name)){
            ptr->item=item_hash_table[key];
            ptr->next=malloc(sizeof(Result));
            preced = ptr;
            ptr=ptr->next;
        }
        key=hashkey(Code,++i);
    }

    if(ptr==liste)
        liste=NULL;
    else
        preced->next=NULL;

    return liste;
}
```

Recherche par libellé avec une deuxième table de hachage (naïve aussi)
(Comme l'exemple remplis entièrement la table de hachage, ce n'est pas vraiment plus rapide)