Tri Lisse - Correction

Samy Jaziri

1 Introduction

Tri par insertion

```
typedef struct noeud_liste {
    int valeur;
    struct noeud_liste* noeud_suivant;
} noeud_liste;

typedef struct liste {
    noeud_liste* premier_noeud;
} liste;
```

```
liste* creer_liste_vide() {
    liste* l = (liste*) malloc(sizeof(liste));
    l->premier_noeud = NULL;
    return l;
}
```

```
void libere_liste( liste* l ) {
    noeud_liste* noeud;
    while( l->premier_noeud ) {
        noeud = l->premier_noeud->noeud_suivant;
        free(l->premier_noeud);
        l->premier_noeud = noeud;
    }
    free(l);
}
```

```c
// Prérequis : i est un indice valide de la liste
int nth( liste* l, int i ) {
    noeud_liste* noeud_courant = l->premier_noeud;

    for (; i > 0; --i)
        noeud_courant = noeud_courant->noeud_suivant;
    return  noeud_courant->valeur;
}
```

On pense lors de l'insertion à mettre à jour le champ `noeud_suivant` du nœud précédent avec l'adresse du nœud ajouté.

```c
void insere( liste* l, int x ) {
    noeud_liste *noeud_courant = l->premier_noeud, *nouveau_noeud;
    noeud_liste** p_precedent = &(l->premier_noeud);

    nouveau_noeud = (noeud_liste*) malloc(sizeof(noeud_liste));
    nouveau_noeud->valeur = x;
    for (noeud_courant = l->premier_noeud ;
         noeud_courant && noeud_courant->valeur <= x;
         noeud_courant = noeud_courant->noeud_suivant) {
        p_precedent = &(noeud_courant->noeud_suivant);
    }
        nouveau_noeud->noeud_suivant = noeud_courant;
        *p_precedent = nouveau_noeud;
}
```

La fonction de tri insère les valeurs de la liste une part une dans une autre liste initialement vide.

```c
liste* tri_insertion( liste* l ) {
    noeud_liste* noeud_courant;
    liste* l_triee = creer_liste_vide();

    for (noeud_courant = l->premier_noeud ;
         noeud_courant ;
         noeud_courant = noeud_courant->noeud_suivant)
        insere(l_triee, noeud_courant->valeur);

    return l_triee;
}
```

Nombre de comparaisons : Dans une liste triée, `insere` réalise une seule comparaison – avec le premier élément –, si la liste est non vide. Au pire des cas `insere` réalise n comparaison dans une liste de n éléments. Donc `tri_insertion` qui appelle `insere` n fois sur une liste de taille n réalise au minimum n comparaisons, et au pire $\sum_{i=1}^{n-1} i$ comparaisons, *i.e.* $\frac{n(n-1)}{2}$ comparaisons.

Liste aléatoire

Pour le calcul de e_n :

```
liste* e( int u0, int n ) {
    int u = u0;
    liste* e = creer_liste_vide();
    insere(e,u);
    for (int i = 0 ; i < n ; ++i) {
        u = (15091 * u) % 64007;
        insere(e,u);
    }
    return e;
}
```

2 Tas binaires

Type arbre

```
typedef struct noeud_arbre {
    int valeur;
    struct noeud_arbre* fils_gauche;
    struct noeud_arbre* fils_droit;
} noeud_arbre;

typedef struct arbre {
    noeud_arbre* racine;
} arbre;
```

```
arbre* creer_arbre( int x, arbre* fg, arbre* fd ) {
    noeud_arbre* racine = (noeud_arbre*) malloc(sizeof(noeud_arbre));
    arbre* a = (arbre*) malloc(sizeof(arbre));

    racine->valeur = x;
    racine->fils_gauche = fg ? fg->racine : NULL;
    free(fg);
    racine->fils_droit = fd ? fd->racine : NULL;
    free(fd);
    a->racine = racine;
    return a;
}
```

La fonction qui détruit une liste détruit chaque nœud puis la liste :

```c
void libere_noeud_arbre( noeud_arbre* a ) {
    if(!a)
        return;
    libere_noeud_arbre(a->fils_gauche);
    libere_noeud_arbre(a->fils_droit);
    free(a);
}

void libere_arbre( arbre* a ) {
    libere_noeud_arbre(a->racine);
    free(a);
}
```

Les arbres considérés sont binaires et parfaits. On a donc $m_k = 2^k - 1$. On montre le résultat pas récurrence sur la hauteur de l'arbre.

- *Initialisation :* $m_0 = 0 = 2^0 - 1$
- *Hérédité :* Soit $k \geq 0$ et supposons $m_k = 2^k - 1$. On considère un arbre binaire de hauteur $k+1$. $k+1 \geq 1$ donc l'arbre possède une racine et deux sous-arbre de hauteur k à gauche et à droite. Ces deux sous-arbres sont des arbres binaires parfaits. On a donc $m_{k+1} = 2*m_k+1 = 2^{k+1}-1$. On peut donc conclure.

Le minimum d'un tas est sa racine par définition.

```c
// Prérequis : a non vide
int min_tas( arbre* a ) {
    return a->racine->valeur;
}
```

Le minimum d'un quasi-tas est le minimum de sa racine et des racines de ses deux fils.

```c
// Prérequis : a non vide
int min_quasi( arbre* a ) {
    int min = a->racine->valeur;
    if (!a->racine->fils_gauche)
        return min;
    if ( a->racine->fils_gauche->valeur < min )
        min = a->racine->fils_gauche->valeur;
    if ( a->racine->fils_droit->valeur < min )
        min = a->racine->fils_droit->valeur;
    return min;
}
```

Transformation d'un quasi-tas en tas

Pour transformer un quasi-tas en tas binaire parfait, on échange la valeur de la racine avec celle de la racine du fils qui possède la plus petite valeur. Puis on procède par induction sur le fils qui a vu sa valeur modifiée.

```
void percole( arbre* a ) {
    if (!a->racine )
        return;

    int min;
    arbre tmp_a = { NULL };
    noeud_arbre* noeud_courant = a->racine ;
    noeud_arbre* noeud_suivant;

    while( noeud_courant->fils_gauche ) {
        tmp_a.racine = noeud_courant;
        min = min_quasi(&tmp_a);
        if (min == noeud_courant->valeur)
            break;

        if (noeud_courant->fils_gauche->valeur == min)
            noeud_suivant = noeud_courant->fils_gauche;
        else
            noeud_suivant = noeud_courant->fils_droit;

        noeud_suivant->valeur = noeud_courant->valeur;
        noeud_courant->valeur = min;
        noeud_courant = noeud_suivant;
    }
}
```

La fonction `percole` procède à des interversions le long d'un chemin de l'arbre. L'interversion, les comparaisons et le calcul de `min_quasi` se font en temps constant. La complexité est donc en $\mathcal{O}(k)$ où k est la hauteur de l'arbre.
Au meilleur cas, `percole` tourne en $\mathcal{O}(1)$ si l'arbre est vide ou si le quasi-tas est déjà un tas.

3 Création d'une liste de tas

Type liste de tas

Le type est le suivante :

```c
typedef struct noeud_liste_tas {
    int taille;
    arbre* tas;
    struct noeud_liste_tas* noeud_suivant;
    struct noeud_liste_tas* noeud_precedent;
} noeud_liste_tas;

typedef struct liste_tas {
    noeud_liste_tas* premier_noeud;
    noeud_liste_tas* dernier_noeud;
} liste_tas;
```

```c
liste_tas* creer_liste_tas() {
    liste_tas* h = (liste_tas*) malloc(sizeof(liste_tas));
    h->premier_noeud = NULL;
    h->dernier_noeud = NULL;
    return h;
}
```

```c
void libere_liste_tas( liste_tas* h ) {
    noeud_liste_tas* noeud;
    while( h->premier_noeud ) {
        noeud = h->premier_noeud->noeud_suivant;
        libere_arbre(h->premier_noeud->tas);
        free(h->premier_noeud);
        h->premier_noeud = noeud;
    }
    free(h);
}
```

```c
void insere_tas(liste_tas* h, arbre* a, int x ) {
    noeud_liste_tas* noeud =
        (noeud_liste_tas*) malloc(sizeof(noeud_liste_tas));
    noeud->taille = x;
    noeud->tas = a;
    noeud->noeud_suivant = h->premier_noeud;
    noeud->noeud_precedent = NULL;
    if(h->premier_noeud)
        h->premier_noeud->noeud_precedent = noeud;
    else
        h->dernier_noeud = noeud;
    h->premier_noeud = noeud;
}
```

```c
// Prérequis h non vide
void extrait_tas(liste_tas* h, arbre** a, int* x) {
    noeud_liste_tas *n1 = h->premier_noeud;
    *a = n1->tas;
    *x = n1->taille;
    h->premier_noeud = n1->noeud_suivant;
    if(h->premier_noeud)
        h->premier_noeud->noeud_precedent = NULL;
    else
        h->dernier_noeud = NULL;
    free(n1);
}
```

Nous avons donc $h = [(a_1, t_1), \dots, (a_r, t_r)]$ et

- $\forall 1 \leq i \leq r, \exists k_i, t_i = m_{k_i}$,
- (k_1,...,k_r)$ est QSC
- $|h| = t_1 + \cdots + t_r$.

Supposons $r > 1$. Notons que comme $(k_1, \dots, k_r)$ est QSC, $r - 1 \leq k_r$. Or $2^{k_r} - 1 \leq |h|$ donc $2^{r-1} - 1 \leq |h|$ et $r - 1 \leq \log(|h| + 1)$. On peut donc conclure que $\mathcal{L}(h) = \mathcal{O}(\log(|h|))$.
Par ailleurs toujours comme $(k_1, \dots, k_r)$ est QSC, et comme k_i est la hauteur du $i^{\text{ème}}$ tas de la liste, $\mathcal{H}(h) = k_r = \mathcal{O}(\log(|h|)$ par le même raisonnement que ci-dessus.

Ajout dans une liste de tas

Ajouter une racine isolée en tête d'une liste dont les deux premiers éléments n'ont pas la même taille ne viole pas la propriété TC.
Si toutefois les deux premiers éléments n'ont pas la même taille, il n'est plus suffisant d'ajouter la racine isolée en tête de la liste. La formule $(\star)$ nous dit qu'il faut dans ce cas là construire un arbre

de taille $2 * t_1 + 1$ qui remplacera les deux premiers arbres de taille t_1. Une racine dont les deux sous-arbres sont de taille t_1 convient.
Pour construire un tas binaire parfaits regroupant les valeurs de la racine isolée a à ajouter et des deux tas binaires parfaits t_g et t_d, on crée un quasi-tas dont la racine est a et les fils gauche et droit sont respectivement t_g et t_d. Puis on appelle `percole` sur le quasi-tas ainsi formé.

```
void ajoute( liste_tas* h, int x ) {
    arbre *ax, *a1, *a2;
    int t1, t2;
    if(!h->premier_noeud
       || !h->premier_noeud->noeud_suivant
       || h->premier_noeud->taille != h->premier_noeud->noeud_suivant->taille) {
        ax = creer_arbre(x,NULL,NULL);
        insere_tas(h, ax, 1);
    } else {
        extrait_tas(h, &a1, &t1);
        extrait_tas(h, &a2, &t2);
        ax = creer_arbre(x,a1,a2);
        insere_tas(h, ax, t1 + t2 + 1);
        percole(h->premier_noeud->tas);
    }
}
```

La fonction `ajoute` tourne au meilleur cas en temps constant. Ce cas est atteint lorsque `h` est vide, `h` est réduit à un élément, les deux premiers tas de `h` ont une taille différente ou si `percole` tourne en $\mathcal{O}(1)$, donc si `x` est plus petits que tous les éléments de `a_1` et `a_2`.
Au pire cas, elle appelle la fonction `percole` qui tourne au pire en $\mathcal{O}(k+1)$ la hauteur de `ax`.

Transformation d'une liste en liste de tas

```
liste_tas* constr_liste_tas( liste* l ) {
    liste_tas* h = creer_liste_tas();

    for (noeud_liste* noeud_courant = l->premier_noeud ;
         noeud_courant ;
         noeud_courant = noeud_courant->noeud_suivant )
        ajoute(h,noeud_courant->valeur);
    return h;
}
```

La complexité dans le cas d'une liste triée est $\mathcal{O}(n)$, car `ajoute` tourne alors en $\mathcal{O}(1)$ puisque l'élément ajouté est plus petit que tous les éléments présents dans la liste de tas.
La complexité dans le cas pire de `ajoute` est en $\mathcal{O}(k)$ avec k la hauteur du premier tas de la liste dans laquelle on ajoute la valeur. Or $k \leq \mathcal{H}(h) = \mathcal{O}(\log(|h|))$.

4 Tri des racines

On utilise un `xor` pour éviter l'utilisation d'une variable intermédiaire.

```
//Prerequis : a et b sont non vides
void echange_racine( arbre* a, arbre* b ) {
    a->racine->valeur ^= b->racine->valeur;
    b->racine->valeur ^= a->racine->valeur;
    a->racine->valeur ^= b->racine->valeur;
}
```

Notons $m = \min_A(a)$ et $m_1 = \min_A(a_1)$. Si $m \leq m_1$ après l'appel à `percole`, la racine du premier élément de la liste sera m qui est bien inférieur à $\min_A(a_1)$ donc la condition RO est vérifiée. Sinon $m_1 < m$ et m_1 est plus petit que la valeur de chaque noeud de a. Le minimum de l'arbre a dont on a remplacé la racine par m_1 est donc m_1. a où la racine est remplacé par m_1 est donc un tas binaire parfait. a_1 lui devient un quasi-tas.
Il suffit ensuite de réitérer l'opération en partant du quasi-tas a_1 et en comparant sont minimum au minimum du tas suivant s'il existe. On s'arrête lorsque le minimum du quasi-tas est plus petit que le minimum du tas lui succédant ou si il n'y a pas de tas ensuite. Dans les deux cas on appelle `percole` et le tas vérifie bien RO.

```
// Prérequis : h vérifie RO et t est la taille de a
void insere_tas_ro ( liste_tas* h, arbre* a, int t ) {
    noeud_liste_tas *noeud_suivant, *noeud_courant;

    insere_tas(h, a, t);

    noeud_courant = h->premier_noeud;
    noeud_suivant = noeud_courant->noeud_suivant;
    while (noeud_suivant
        && min_quasi(noeud_courant->tas) > noeud_suivant->tas->racine->valeur ) {
        echange_racine(noeud_courant->tas, noeud_suivant->tas);
        noeud_courant = noeud_suivant;
        noeud_suivant = noeud_courant->noeud_suivant;
    }
    percole(noeud_courant->tas);
}
```

La fonction `insere_tas` procède au maximum à autant d'échange que la longueur de la liste de tas, un $\mathcal{O}(\log(n))$, et un appel à `percole`, aussi un $\mathcal{O}(\log(n))$. Elle tourne donc en $\mathcal{O}(\log(n))$ au pire cas et en $\mathcal{O}(1)$ si l'élément ajouté est plus petit que tous les éléments de la liste de tas.
On remarque que la fonction `insere_tas` ne modifie pas les tailles des tas de la liste. On insère les tas en partant de la fin dans une liste de tas initialement vide.

```
// Prérequis : h vérifie TC
void tri_racine( liste_tas** h ) {
    liste_tas* h_triee = creer_liste_tas();

    for (noeud_liste_tas* noeud_courant = (*h)->dernier_noeud ;
         noeud_courant ;
         noeud_courant = noeud_courant->noeud_precedent )
        insere_tas_ro(h_triee, noeud_courant->tas, noeud_courant->taille);
   free(*h);
   *h = h_triee;
}
```

La fonction `tri_racine` appelle `insere_tas_ro` pour chaque élément de la liste de tas, donc $\mathcal{O}(\log(n))$ fois. La complexité finale est donc $\mathcal{O}(\log(n)^2)$ au pire.

5 Extraction des éléments d'une liste de tas

```
int extraire( liste_tas* h) {
    arbre *a, *a1, *a2;
    int x, t;

    extrait_tas(h,&a,&t);
    x = a->racine->valeur;
    if(a->racine->fils_gauche) {
        a1 = (arbre*) malloc(sizeof(arbre));
        a1->racine = a->racine->fils_gauche ;
        a2 = (arbre*) malloc(sizeof(arbre));
        a2->racine = a->racine->fils_droit ;
        t = ( t - 1 ) / 2;
        insere_tas_ro(h, a2, t );
        insere_tas_ro(h, a1, t );
    }

    free(a->racine);
    free(a);
    return x;
}
```

La fonction `extraire` fait deux appels à `insere_tas_ro`, le reste des opérations ont un coup constant. La complexité finale est en $\mathcal{O}(\log(n))$ au pire et en $\mathcal{O}(1)$ si les racines de a, $a1$ et $a2$ sont plus petites que tous les éléments du tas et rangées dans cet ordre.

6 Tri lisse

```
liste* tri_lisse( liste* l) {
    liste* l_triee = creer_liste_vide();
    liste_tas* h = constr_liste_tas(l);
    tri_racine(&h);
    while(h->premier_noeud)
        insere(l_triee, extraire(h));
    free(h);
    return l_triee;
}
```

Soit $n = |l|$. La création de `h` est en $\mathcal{O}(n)$ si la liste est triée, et en $O(nlog(n)) au pire cas. Le tri des racines est en $\mathcal{O}(\log(n)^2)$ et peut être négligé. L'extraction est au pire en $\mathcal{O}(\log(n))$ et est réalisée une fois pour chaque élément. La complexité au pire cas est donc en $\mathcal{O}(n\ \log(n))$. Si la liste initiale est triée, on remarque que les tas sont des arbres binaires de recherche parfait, et `extraire` tourne systématiquement en $\mathcal{O}(1)$, donc la complexité du tri dans ce cas est $\mathcal{O}(n)$.

7 Affichage des résultats de la fiche réponse

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

void print_tailles( liste_tas* h ) {
    printf("[ ");
    for (noeud_liste_tas* noeud_courant = h->premier_noeud ;
         noeud_courant ;
         noeud_courant = noeud_courant->noeud_suivant )
        printf("(%i, %i) ", min_tas(noeud_courant->tas), noeud_courant->taille);
    printf(" ]");
}
```

```c
int main( int argc, char** argv ) {
    if (argc != 2) {
        fprintf(stderr, "Usage : %s u0\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int u0 = atoi(argv[1]);
    liste *e100 = e(u0,100), *e1000 = e(u0,1000), *e10000 = e(u0,10000);
    liste *t_e100 = tri_insertion(e100), *t_e1000 = tri_insertion(e1000),
        *t_e10000 = tri_insertion(e10000);
    liste_tas *h100 = constr_liste_tas(e100), *h1000 = constr_liste_tas(e1000),
        *h10000 = constr_liste_tas(e10000);

    printf("Question 4 : a) %i  b) %i  c) %i \n",
           nth(t_e100,21), nth(t_e1000,144), nth(t_e10000,1212));
    printf("Question 12 : a) ");
    print_tailles(h100);
    printf(" b) ");
    print_tailles(h1000);
    printf(" c) ");
    print_tailles(h10000);
    printf("\n");

    tri_racine(&h100);
    tri_racine(&h1000);
    tri_racine(&h10000);
    printf("Question 16 : a) %i  b) %i  c) %i \n",
           h100->dernier_noeud->tas->racine->valeur,
           h1000->dernier_noeud->tas->racine->valeur,
           h10000->dernier_noeud->tas->racine->valeur );

    libere_liste(e100);
    libere_liste(e1000);
    libere_liste(e10000);
    libere_liste(t_e100);
    libere_liste(t_e1000);
    libere_liste(t_e10000);
    libere_liste_tas(h100);
    libere_liste_tas(h1000);
    libere_liste_tas(h10000);
}
```