TD7 - Listes chaînées (Suite)

Année 2002

1 Tri par fusion

Le tri par fusion

Pour trier une liste chaînée d'au moins deux éléments, on procède *récursivement* comme suit : on la coupe en deux, on trie chacune des parties (qui sont plus petites) et on les “fusionne” comme on l'explique plus bas, d'où le nom de cette méthode. Lorsque les morceaux de la liste coupée en deux ne sont pas plus petit que la liste de départ, (le cas de base de la récursion) c'est que la liste a au plus un élément, et elle peut être considérées comme déjà triées.

Exercice 1 : Faire le schéma correspondant à l'opération suivante : soit **a** une liste contenant n éléments. On crée une liste **b** qui contient les $\frac{n}{2}$ derniers éléments de a en les détachant de **a**. **a** ne contient donc plus que ses $\frac{n}{2}$ premiers éléments.

Exercice 2 : Écrire une fonction qui partage une liste en deux. Combien de cellules sont visitées pendant cette opération ? En fait on n'est pas obligé de preserver l'ordre des cartes lors du partage. Donner une seconde solution qui ne visite qu'une fois chacune des cellules.

Fusion

La fusion de deux listes triées **a** et **b**, pour obtenir une autre liste triée, s'effectue de la façon suivante : on s'arrange pour que la valeur de tête de **a** soit inférieure à celle de **b** (quitte à permuter les deux). Dans ce cas on écarte le premier élement de **a**, et on fusionne ce qui reste de **a** avec **b**. Finalement on rajoute l'élément écarté au résultat.

Il est à noter que toutes ces opérations se font par manipulations de pointeurs : à aucun moment on ne crée de nouvel élément.

Exercice 3 : Écrire une procédure à deux arguments **a** et**b**, qui fusionne les deux listes dans a.

Exercice 4 : Écrire un programme qui effectue par cette méthode, de façon récursive, le tri de **n** entiers donnés par l'utilisateur.

2 Listes doublement chaînées

Les listes doublement chaînées sont des extensions des listes chaînées. Les cellules contiennent à la fois un pointeur vers la cellule qui la suit (ici `suiv`), et également un pointeur vers la cellule qui la precède. Lorsque l'une des deux n'existe pas, on le signale en lui donnant la valeur `NULL`.

Exercice 5 : On considère une machine constituée d'une tête de lecture et d'un ruban ou elle peut écrire. La tête peut se déplacer vers la gauche ou vers la droite. On représente le ruban à l'aide d'une liste doublement chaînée, et la tête de lecture par un pointeur vers une position sur le ruban.

A chaque fois que la tête change de position, elle augmente de un la valeur de la case sur laquelle elle pointait, avant de la quitter.

Lorsque la tête avance au delà de l'extrémité droite du ruban, elle le rallonge d'une case, et elle y écrit 0.

– définissez la machine
– écrivez les procédures `recule` et `avance`

3 Exercices complémentaires

Exercice 6 : Après avoir trié une liste, on va la mélanger. Dans le monde réel pour mélanger, (c'est l'exemple d'un paquet de cartes), on le coupe d'abord en deux tas à priori non équilibres, puis on intercale les éléments l'un après l'autre. Proposer les procédures qui permettent de faire ces opérations.

Exercice 7 : <u>Listes chaînée CIRCULAIRES</u>

Rien dans la définitions des différents types de liste vus jusqu'à présent n'oblige à ce que la fin d'une liste soit déterminée par le pointeur NULL. On considère ici des listes (dites circulaires) que l'on peut obtenir à partir des listes connues (dites planes), pour lesquelles le successeur du "dernier" est le "premier". (Pour les listes doublement chainees, on dira que le prédécesseur du "premier" élément est le "dernier")

- Donner un exemple de liste non circulaire et non plane. A quoi cela ressemble t'il si on se pose la même question avec des listes doublement chaînées ?
- Chercher le minimum d'une liste circulaire.
- On veut écrire une fonction qui teste si une liste est plane, pour cela il faut la parcourir et détecter si on est déjà passe par une cellule. Le seul problème est qu'il faut répondre oui ou non, et donc trouver une condition d'arrêt à ce parcourt pour le cas où la liste est non plane. Pour nous aider, on suppose que les cellules comportent un champs supplémentaire `tag`. Ce `booléen`, initialement a faux permet de garder une trace d'un passage. Écrivez la fonction `test_plane`. Attention à remettre la liste dans l'état où vous l'avez trouvé !
- Écrire une fonction qui teste si une liste chaînée est circulaire.
- Comment faire si on n'avait pas sous la main ce `tag` ?

4 Correction

Correction 1 :

Correction 2 : La première solution attendue consiste à compter les éléments : n, puis de recompter en s'arretant juste avant la moitie : n/2-1, pour pouvoir couper.

```
// coupe la liste l en deux partie de meme longueur +- 1
void coupe(liste l,liste &a,liste &b){
   int n=0;
   // calcul de la longueur
   a=l;
   for (liste i=l;l!=NULL;l=l->suiv){n++;}

   // on repere la moitie
   l=a;
   for (int i=1; i < n/2;i++ ) { l=l->suiv;}

   // on detache la branche
   b=l->suiv;
   l->suiv=NULL;
}
```

Une autre solution consiste à distribuer les éléments, alternativement dans une liste puis une autres. Ça vaut le coup de regarder pour cette solution le jeu des passages par référence.

```
// Une autre facon de couper, en distribuant dans des paquets
// initialement vides
void coupe2(liste l,liste &a,liste &b){
   a=NULL;
   b=NULL;
   distribue(l,a,b);
}

// le haut de la liste part dans un paquet, puis le suivant dans
// l'autre paquet, jusqu'a ce que la liste soit vide.
void distribue(liste l,liste &a,liste &b){
   if (l==NULL) return;
   insert(a,l->val);
   distribue(l->suiv,b,a);
}

// ajout d'un element a une liste
void insert(liste &l,int i){
   liste aux=new element;
   aux->val =i;
   aux->suiv = l;
   l=aux;
}
```

Correction 3 :

```
// la fusion de deux listes Triees
// resultat dans a
void fusionneTriee(liste &a,liste &b){
   liste aux,aux2;
   if (a==NULL) {a=b; return;}
   if (b==NULL) {return;}

   if (a->val > b->val) { // echange de a et de b
      aux=a;
      a=b;
      b=aux;
   } // on peut continuer

   fusionneTriee(a->suiv,b); // Remarquer qu'en une etape on fait 2 operations
```

```
   // a) on fusionne les sous listes
   // b) on rattache a la cellule de tete de a
}
```

Correction 4 :

```
// le trie par fusion
void trie(liste &l){
   liste a,b;
   coupe2(l,a,b);
   // si on a pas reduit en coupant on s'arrete
   if (a==NULL) {l=b;return;}
   if (b==NULL) {l=a;return;}
   trie(a);
   trie(b);
   fusionneTriee(a,b);
   l=a;
}
```

Correction 5 :

```
#include <iostream>

typedef struct element{
   int val;
   element * suiv;
   element * pred;
};

typedef element* tete;


void affiche_g(tete t){
   if (t==NULL) { cout << endl; return;}
   cout << t->val << " ";
   affiche_g(t->pred);
}

void affiche_d(tete t){
   if (t==NULL) { cout << endl; return;}
   cout << t->val<< " ";
   affiche_d(t->suiv);
}


void affiche(tete t){
   cout << " partie gauche" << endl;
   affiche_g(t);
   cout << "partie droite" << endl;
   affiche_d(t);
}

void avance(tete &t){
   if(t==NULL){
      t=new element;
      t->suiv=NULL;
      t->pred=NULL;
      t->val = 0;
      return;
   }
   (t->val)++;
   if (t->suiv == NULL){
      t->suiv=new element;
      t->suiv->pred=t;
      t=t->suiv;
```

```
      t->suiv=NULL;
      t->val=0;
   }
   else t=t->suiv;
}

void recule(tete &t){
   if (t == NULL) return;
   if (t->pred==NULL) return;
   (t->val)++;
   t=t->pred;
}



void main(){
    tete t=NULL;
    avance(t);
    avance(t);
    avance(t);
    recule(t);
    recule(t);
    avance(t);
    recule(t);
    avance(t);
    recule(t);

    affiche(t);
}
```

Correction 6 :

```
#include <iostream>
#include <cstdlib>
#include <time.h>

typedef struct element{
   int val;
   element * suiv;
};

typedef element* liste;


// ajout d'un element a une liste
void insert(liste &l,int i){
   liste aux=new element;
   aux->val =i;
   aux->suiv = l;
   l=aux;
}

// creation automatique d'une liste de 10 elements
void liste_10(liste &l){
   l=NULL;
   for (int i=0;i<10;i++) {insert(l,i);}
}

// affichage d'une liste
void affiche(liste l){
   if (l==NULL) {cout << endl; return;}
   cout << " " << l->val;
   affiche(l->suiv);
}
```

```
// taille de la liste
int taille(liste l){
   if (l==NULL) return 0;
   return 1+taille(l->suiv);
}


// coupe en position x, resultat dans a et b
void coupe(liste l,int x,liste &a,liste &b){

   // on repere la position x-1
   a=l;
   for (int i=1; i < x ;i++ ) { l=l->suiv;}

   // on detache la branche
   if (l== NULL) { b=NULL;}
   else {
      b=l->suiv;
      l->suiv=NULL;
   }
}

// on intercale, resultat dans a
void shuffle(liste &a,liste b){
   liste auxa,auxb;
   if (a==NULL) {a=b;return;}
   if (b==NULL) {return;}
   shuffle(a->suiv,b->suiv);
   auxa=a->suiv;
   a->suiv=b;
   b->suiv=auxa;
}

// on coupe au hazard, et on intercale
void melange_1(liste &l){
   liste a,b;
   int x=taille(l);
   x=(rand() % x);
   coupe (l,x,a,b);
   shuffle(a,b);
   l=a;
}

void main(){
   liste l;
   srand(time(NULL));

   liste_10(l);
   affiche(l);
   melange_1(l);
   affiche(l);
}
```

Correction 7 :

- il suffit que le dernier pointe sur un élément du milieu. Pour les doublement chaînée, c'est pire, puisque le prédécesseur est alors une informations incomplète, on peut dire que ça donne des structures plus générales que les arbres.
- je garde un doit sur le début, et je parcours jusqu'à ce que je retombe dessus.

  ```
  int min(liste l){
     liste aux;
     int m;
     if (l==NULL) { cout << ``erreur''; return -1;}
  ```

```
    m=l->val;
    aux=l->suiv;
    while(aux!=l){
      if (aux->val < m) m= aux->val;
      aux=aux->suiv;
    }
}
```

- il ne faut pas oublier de detagger, ça doit se faire avant les return, ce qui m'impose de stocker le résultat.

```
bool est_plane(liste l){
    bool res;
    if (l==NULL) return true;

    if (l->tag) {
      l->tag=false;
      return false;
    }
    l->tag =true;
    res=est_plane(l->suiv);
    l->tag=false;
    return res;
}
```

- Une liste est circulaire, si lorsque je rencontre un état marque c'était mon point de départ. Il faut donc reprendre la fonction précédente, et au lieu de retourner un booléen, retourner le pointeur sur lequel on cycle. Remarquer que NULL est circulaire.

```
bool est_circulaire(liste l){
 return (l==get_cycle(l));
}

liste get_cycle(liste l){
    liste res;
    if (l==NULL) return l;

    if (l->tag) {
      l->tag=false;
      return l;
    }
    l->tag =true;
    res=get_cycle(l->suiv);
    l->tag=false;
    return res;
}
```

- On pourrait utiliser une liste de pointeurs, qui contiendrait les positions déjà visitées.