Algorithmes et structures de données : TD 8 Corrigé

Tableaux dynamiques - Listes linéaires simplement chaînées - Complexité asymptotique

Rappel :

`SetLength(tableau, n)` est de complexité O(n)
`SetLength(tableau, 1)` est de complexité O(1)
`New(element)` est de complexité O(1) quand `element` est d'un type de taille fixe

Exercice 8.1 *Listes linéaire simplement chaînée*

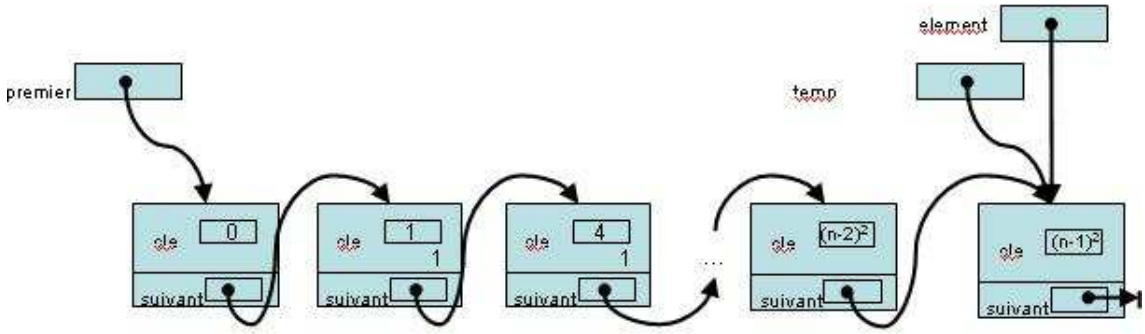
Considérer l'algorithme suivant qui crée une liste de n éléments :

```
{Algorithme 2}
type
        p_t_liste_simple = ^t_liste_simple;
        t_liste_simple = record
                cle     : integer;
                suivant : p_t_liste_simple;
        end;
var element : p_t_liste_simple;
var temp    : p_t_liste_simple;
var premier : p_t_liste_simple;
var i : integer;

début
        New(element);
        element^.suivant := NIL;
        element^.cle := 0;
        premier := element;

        temp := element;
        pour i de 1 à n-1 faire
        début
                New(element);
                element^.cle := i*i;
                element^.suivant := NIL;
                temp^.suivant := element;
                temp := element;
        fin
fin
```

1. Quelle est la complexité de cet algorithme (notation Grand-O) ?
   La complexité de cet algorithme est O(n).
2. Ebaucher l'occupation de la mémoire d'une manière compacte.



3. Ecrire un algorithme qui parcours la liste et affiche la `cle` de chaque élément à l'écran. Qu'est-ce qui est affiché à l'écran ?

```
temp := premier;
        while (NOT (temp = NIL)) do
        begin
                WriteLn(temp^.cle);
                temp := temp^.suivant;
        end;
```

   Il est affiché :

```
0
1
4
..
```

4. Ecrire un algorithme qui rajoute un élément supplémentaire avec la valeur 1000 **au début** de la liste (la liste aura n+1 éléments). Quelle est la complexité de votre algorithme ? Comparer avec l'exercice précédente sur les tableaux dynamiques.

   La complexité de cet algorithme est O(1) comparé à O(n) pour les tableaux dynamiques.

```
New(element);
element^.cle :=1000;
element^.suivant := premier;
premier := element;
```

5. Ecrire un algorithme qui rajoute un élément supplémentaire avec la valeur 1000 **à la fin** de la liste. Quelle est la complexité de votre algorithme ? Comparer avec l'exercice précédente sur les tableaux dynamiques.

```
temp := premier;
si temp = NIL alors
                New(premier);
                premier^.cle := 1000;
                premier^.suivant := NIL;
sinon
        tant que (NOT (temp^.suivant = NIL)) faire
                temp := temp^.suivant;
        fin tant que

        New(element);
        element^.cle := 1000;
        temp^.suivant := element;
        element^.suivant := NIL;
fin si
```

La complexité de cet algorithme est O(n), comme pour les tableaux dynamiques.

6. Que faut-il rajouter à l'algorithme 2 pour réduire la complexité de rajouter un élément supplémentaire avec la valeur 1000 **à la fin** de la liste ? Ecrivez l'algorithme !

Il faut rajouter un pointeur `dernier` **qui pointe toujours vers le dernier élément. On réduit ainsi la complexité pour rajouter un élément à la fin de la liste à O(1).**

```
var dernier : p_t_liste_simple;
{...}
début

{ .. initialiser : }

        New(element);
        element^.suivant := NIL;
        element^.cle := 1111;
        temp := element;
        premier := element;
        for i:=1 to 10 do
        begin
                New(element);
                element^.cle := i*i;
        element^.suivant := NIL;
        temp^.suivant := element;
        temp := element;
                dernier := temp;    { seulement rajouter ça }
        end;
fin

{ .. rajouter en début de liste : }
```

```
        temp := premier;
        New(element);
        element^.cle :=1000;
        element^.suivant := premier;
        premier := element;
        si temp = NIL alors
                dernier := premier;
        fin si
{ .. rajouter en fin de liste : }

        si premier = NIL alors
                New(premier);
                premier^.cle := 1000;
        premier^.suivant := NIL;
                dernier := premier;
        sinon
                New(element);
                element^.cle :=20002;
                element^.suivant := NIL;
                dernier^.suivant := element;
        fin si
```

7. Ecrire un algorithme qui parcours la liste et libère la mémoire occupé par la liste en utilisant `Dispose`.

```
var ancien : p_t_liste_simple;
{...}
début

        temp := premier;
        while (NOT (temp = NIL)) do
        begin
                ancien := temp;
                temp := temp^.suivant;
                Dispose(ancien);
        end;

fin
```

Exercice 8.2 *Pointeurs*

Rappeler vous de l'exercice sur les musiciens de la dernière séance (TD 6.3), et considérer maintenant l'algorithme suivant :

```
type p_t_musicien = ^t_musicien;
     t_musicien = record
                        cle     : integer;
```

```
                        nom     : string;
                        annee   : integer;
                        suivant : p_t_musicien;
                    end;

var guitariste  : p_t_musicien;
var bassiste    : p_t_musicien;
var chanteur    : p_t_musicien;
var musicien    : p_t_musicien;

var i : integer;

begin

  New(guitariste);
  New(bassiste);
  New(chanteur);

  guitariste^.cle := 1;
  guitariste^.nom := 'Robert';
  guitariste^.annee := 1982;
  guitariste^.suivant := NIL;

  bassiste^.cle := 2;
  bassiste^.nom := 'Sebastian';
  bassiste^.annee := 1979;
  bassiste^.suivant := NIL;

  chanteur^.cle := 3;
  chanteur^.nom := 'Rainer';
  chanteur^.annee := 1984;
  chanteur^.suivant := NIL;

  guitariste^.suivant := bassiste;
  bassiste^.suivant := chanteur;
  chanteur^.suivant := guitariste;

  musicien := guitariste;

  { Endroit 1 }

  for i := 1 to 10 do
  begin
    write('nom ', musicien^.nom);
    writeln('  annee  ', musicien^.annee);
    musicien := musicien^.suivant;
  end;
```

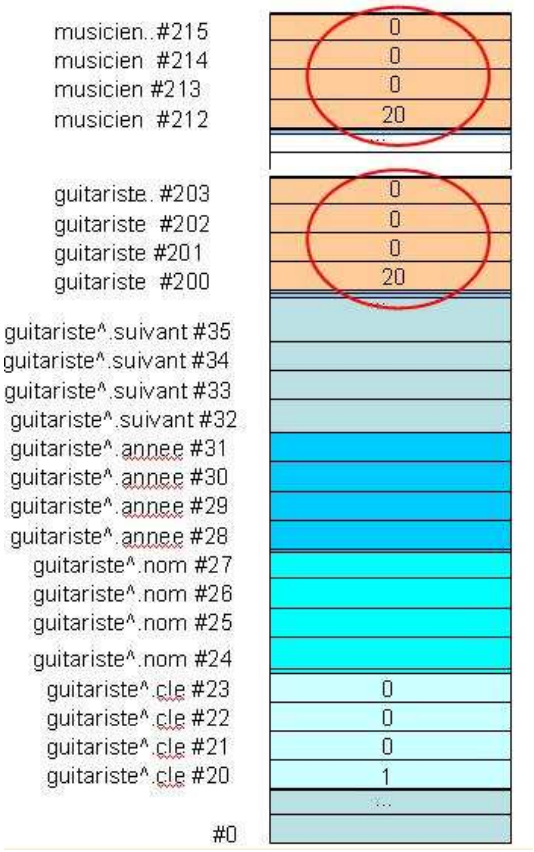
```
  readln;

  Dispose(guitariste);
  Dispose(bassiste);
  Dispose(chanteur);

end.
```

1. Ebaucher l'occupation de la mémoire à l'endroit 1.

   Un extrait de l'occupation de la mémoire est comme suit :



2. Quelle est la différence entre cet algorithme et celui de la dernière séance ?

   Cette fois-ci, chaque musicien est stocké en tant que pointeur vers un enregistrement, et ne plus par un enregistrement tout court.

3. Comparé à l'algorithme de la dernière séance (TD 6.3), lequel des deux algorithme occupe moins de la place dans la mémoire et est donc plus efficace en terme de mémoire ?

   Regardons d'abord l'occupation de la mémoire : dans le TD 6.3, les 4 enregistrements occupaient 4*16 = 64 octets. Dans ce TD, les 4 pointeurs et les trois enregistrements occupent 4*4+3*16 = 64 octets également.

   Par contre, l'algorithme de ce TD est beaucoup plus efficace, car à chaque affectation de type `musicien := musicien^ .suivant`**, il faut uniquement affecter**

les 4 octets du pointeur au lieu de copier toute la mémoire de 16 octets du musicien.

4. Faites tourner cet algorithme dans un tableau.
   Un extrait est comme suit :

i	musicien^ .nom
..	..
	Robert (avant l'entrée dans la boucle
1	
	Sebastian (après l'éxecution de `musicien := musicien^ .suivant;`)
2	
	Rainer
3	
	Robert
4	
	Sebastian
5	
	Rainer
6	
	Robert
7	
	Sebastian
8	
	Rainer
9	
	Robert
10	
	Sebastian

5. Qu'est-ce qui affiche cet algorithme à l'écran ?

```
nom Robert  annee  1982
nom Sebastian  annee  1979
nom Rainer  annee  1984
nom Robert  annee  1982
nom Sebastian  annee  1979
nom Rainer  annee  1984
nom Robert  annee  1982
nom Sebastian  annee  1979
nom Rainer  annee  1984
nom Robert  annee  1982
```