Université de Nice-Sophia Antipolis — Deug MIAS-MI 1
Algorithmique & Programmation — 2003–2004

TP N° 14

Tris anormaux

Buts : L'objectif de cette séance est d'implémenter deux tris anormaux : le tri par base et le tri par calcul d'adresse.

1 Tri par base

1.1 Mise en place

Vous allez implémenter le tri par base tel qu'il est présenté dans le mémento 14, section 3.

Vous êtes déjà familiarisés avec la structure mise en place pour implémenter les tris : l'unité `sort` est disponible sur la page Internet dans le fichier `sort.pas`. Un autre fichier est à votre disposition : `basesort.pas`, qui contient d'ores et déjà des déclarations de type et des procédures de traitement de files d'éléments. Ce fichier doit être complété.

a. La méthode du tri par base nécessite d'accéder aux coefficients des clés dans la base choisie.

Pour rappel, la décomposition d'un entier naturel quelconque dans une base b s'écrit $\sum_{i=0}^{\infty} c_i b^i$ avec des coefficients c_i tels que $\forall i, 0 \leq c_i < b$. Comme nos clés ne sont pas des entiers naturels quelconques mais sont bornées, cette décomposition ne nécessite qu'un nombre borné de termes que nous définissons dans la constante `MaxColumns`. Par exemple, 513 se décompose dans la base 10 en $5 \times 10^2 + 1 \times 10^1 + 3 \times 10^0$.

Le calcul serait coûteux s'il devait être effectué à chaque itération de l'algorithme ; aussi, une solution est de pré-calculer ces coefficients à partir des clés et de les disposer dans des tables. La première étape de l'implémentation est de ce fait l'écriture de deux fonctions permettant de convertir une clé de type `TKey` en un tableau de type `TColumns` de coefficients dans la base `Base` et inversement.

```
const
   Base       = 10; { La base de calcul. }
   MaxColumns = 3;  { Le nombre maximal de colonnes dans cette base. }
type
   { Un coefficient dans la base 'Base'. }
   TCoefficient = 0..Base-1;
   { Les clés sont converties en tableaux de coefficients dans la base. }
   TColumns     = array[1..MaxColumns] of TCoefficient;

{ Retourne un nombre en colonnes à partir de la clé 'k'. }
function keyToColumns (k : TKey): TColumns;
```

```
var
   i : 1..MaxColumns;
   c : TColumns;
   r : TCoefficient;
begin
   for i := MaxColumns downto 1 do begin
      r := k mod Base;
      c[i] := r;
      k := (k - r) div Base
```

```
   end;
   keyToColumns := c
end; { keyToColumns }

.....

{ Retourne une clé correspondant au nombre en colonnes 'c'. }
function columnsToKey (c : TColumns): TKey;
```

```
var
   i : 1..MaxColumns;
   k : Integer;
begin
   k := 0;
   for i := 1 to MaxColumns do
      k := Base * k + c[i];
   columnsToKey := k
end; { columnsToKey }

.....
```

b. Comme le tri par base opère sur des files d'éléments, il est nécessaire de convertir le tableau de données initial de type `TTable` en une file d'éléments de type `TFifo` :

```
type
   { On définit le type file présenté dans le mémento. }
   PFifoNode    = ^TFifoNode;
   TFifoNode    = record
                     object : TColumns;
                     next   : PFifoNode
                  end;
   TFifo        = record
                     head, tail : PFifoNode
                  end;

   { Les casiers de files. }
   TRack        = array[TCoefficient] of TFifo;
```

Comme dans le mémento, la tête de file `head` pointe sur un noeud sans objet, et la queue `tail` sur le dernier noeud de la file, qui a `nil` pour élément suivant (voir la figure 1).

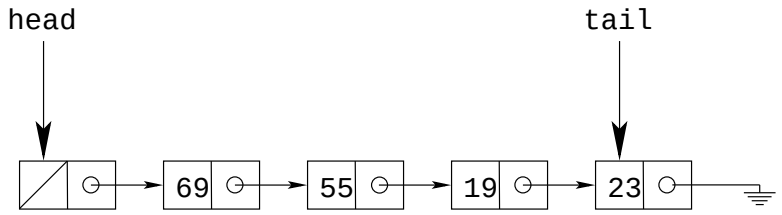

FIG. 1 – Exemple de file.

Le fichier `basesort.pas` contient déjà des procédures utiles, à savoir :
- `initFifo` qui initialise une nouvelle file vide ;
- `tableToFifo` qui remplit une file à partir des éléments de la table passée en argument ;
- `fifoToTable` qui effectue l'opération inverse ;
- `displayFifo` qui permet d'afficher le contenu d'une file et pourra vous aider à déverminer votre programme.

Lisez le code de ces procédures ; vous êtes prêts à attaquer le code du tri par base.

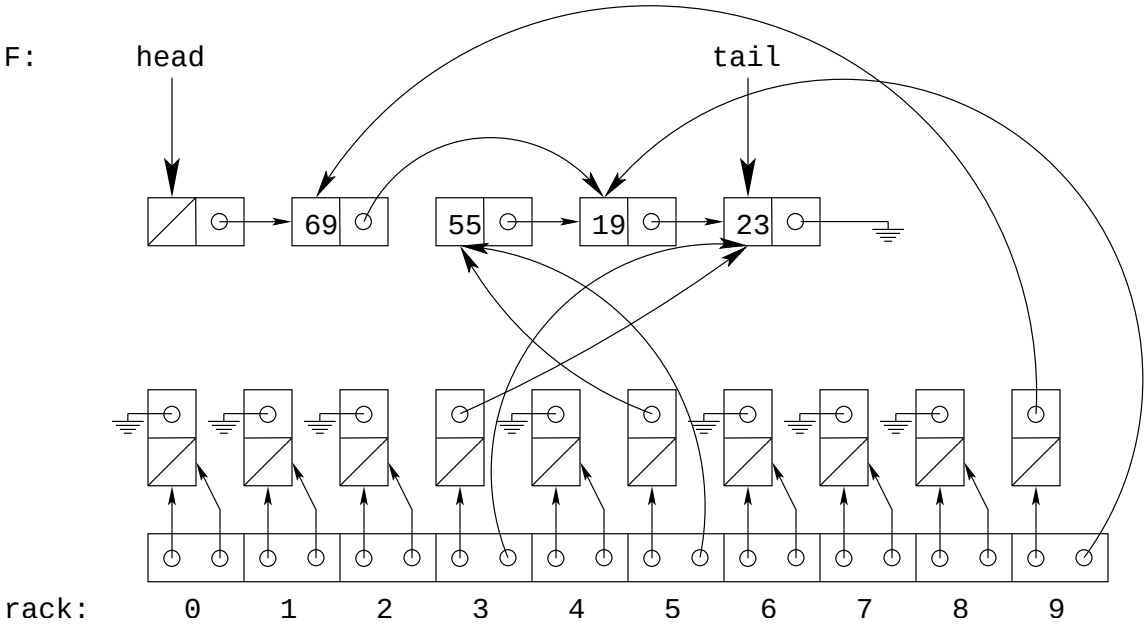

FIG. 2 – Résultat de la dispersion selon la colonne des unités des éléments de la file de la figure 1. Les deux pointeurs dans chaque casier sont dans l'ordre la tête et la queue de chaque file.

1.2 Le tri

Le mémento présente un algorithme optimisant les opérations sur les files d'éléments lors des opérations de dispersion et de concaténation. Celles-ci vont être implémentées au moyen de deux procédures internes à la procédure de tri `baseSort`. Le tri s'effectuant sur des files, une dernière procédure interne de tri devra être écrite.

c. La dispersion des éléments de la file à trier dans les casiers est l'objet de la procédure `spread` (voir la figure 2) :

```
{ Disperse les éléments de 'F' dans les casiers de 'rack'
  selon leurs coefficients dans la colonne 'c'. }
{ Conséquents : pour tout 'i', 'rack[i].tail' pointe sur le dernier élément
  du casier 'i'. }
procedure spread (var F : TFifo; var rack : TRack; c : TCoefficient);
```

```
var
   i : TCoefficient;
   q : PFifoNode;
begin
   q := F.head^.next;
   while q <> nil do begin
      i := q^.object[c];
      { Ajout de 'q' dans 'rack[i]'. }
      rack[i].tail^.next := q;
      rack[i].tail := q;

      q := q^.next
   end
end; { spread }
```

.....

d. Une fois cette répartition dans les casiers effectuée, il faut concaténer les casiers dans la file principale (voir la figure 3) ; c'est l'objet de la procédure `concatenate` :

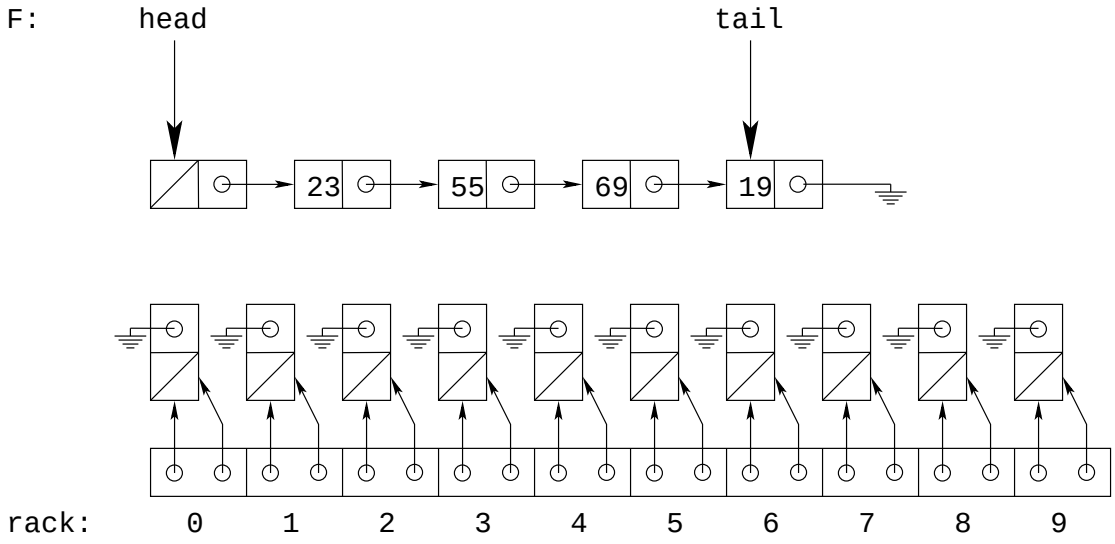

FIG. 3 – Résultat de la concaténation des casiers de la figure 2.

```
{ Concatène les éléments des casiers de 'rack' dans la file 'F'. }
{ Conséquents : pour tout 'i', 'rack[i]' est vide et 'f.tail^.next = nil'. }
procedure concatenate (var rack : TRack; var F : TFifo);
```

```
var
   i : TCoefficient;
begin
   F.tail := F.head;
   for i := 0 to Base-1 do begin
      { Ajout du casier à 'F' s'il n'est pas vide. }
      if rack[i].head^.next <> nil then begin
         F.tail^.next := rack[i].head^.next;
         F.tail := rack[i].tail;
      end;
      { On vide le casier. }
      rack[i].tail := rack[i].head;
      rack[i].head^.next := nil;
   end;
   F.tail^.next := nil
end; { concatenate }
```

.....

e. Le tri utilise enfin une procédure interne `sort` qui trie les éléments d'une file. Implémentez cette procédure en adaptant l'algorithme présenté dans le mémento pour utiliser `spread` et `concatenate`. N'oubliez pas de libérer la mémoire allouée aux casiers lors de leur initialisation.

```
{ Trie la file 'F'. }
procedure sort (var F : TFifo);
```

```
var
   rack : TRack;
```

```
   c    : 1..MaxColumns;
   i    : TCoefficient;
begin
   { Initialisation des casiers. }
   for i := 0 to Base-1 do
      initFifo(rack[i]);

   for c := MaxColumns downto 1 do begin
      { Dispersion des éléments dans les casiers. }
      spread(F, rack, c);
      { Concaténation des casiers dans 'F' et vidage de ceux-ci. }
      concatenate(rack, F)
   end;

   { Destruction des casiers. }
   for i := 0 to Base-1 do
      dispose(rack[i].head)
end; { sort }
```

.....

f. Le tri ignore les compteurs de comparaisons et d'affectations de clés puisque le tri par base n'est pas un tri par comparaison. La procédure `baseSort` se borne donc à convertir la table fournie en une file, appeler `sort` pour la trier, et reconvertir cette file en une table.

Écrivez le code principal de la procédure `baseSort` :

```
{ Trie la table 'T'. 'nc' et 'na' ne sont pas mis à jour. }
procedure baseSort (var T : TTable; var nc, na : Integer);
```

```
var
   F : TFifo;
begin
   tableToFifo(T, F);
   sort(F);
   fifoToTable(F, T)
end; { baseSort }
```

.....

Vous êtes prêts à tester votre tri. Compilez votre programme avec l'option `--automake` pour inclure l'unité `sort.pas` et testez-le.

2 Tri par calcul d'adresse

Vous allez implémenter un tri par calcul d'adresse simple tel qu'il est présenté dans le TD 14 section 2. Comme dans la section précédente, vous allez utiliser l'unité `sort`. Celle-ci définit en particulier une valeur maximale de clé `MaxKey`. N'oubliez donc pas d'importer cette unité dans votre programme.

g. Implémentez le tri par calcul d'adresse ; comme ce n'est pas un tri par comparaison, les compteurs de comparaisons et d'affectations sont simplement ignorés.

```
{ Trie la table 'T'. 'nc' et 'na' restent inchangés. }
procedure adressSort (var T : TTable; var nc, na : Integer);
```

```
var
   count : array[TKey] of 0..TableSize;
   v     : TKey;
```

```
   i, c  : TIndex;
begin
   { Initialisation du tableau de comptage. }
   for v := 1 to MaxKey do
      count[v] := 0;

   { Comptage des occurrences de chaque clé. }
   for i := 1 to T.size do
      count[T.table[i]] := count[T.table[i]] + 1;

   { Remplissage du tableau. }
   i := 1;
   for v := 1 to MaxKey do
      for c := 1 to count[v] do begin
         T.table[i] := v;
         i := i + 1
      end;
end; { adressSort }
```

h. Utilisez la procédure `testSort` définie dans l'unité `sort` pour écrire le corps du programme principal. Compilez avec l'option `--automake` et testez votre programme.

```
program adresssort;

uses sort;

procedure adressSort (var T : TTable; var nc, na : Integer);
   ...
end; { adressSort }

begin
   testSort(adressSort)
end.
```