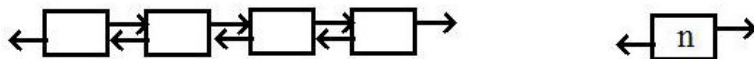


V- Listes chaînées

Nous avons jusqu'à présent utilisé des tableaux. Il s'agit de structures rigides qui doivent être déclarées une fois pour toutes sur une certaine longueur. Quand le nombre des données est variable, la longueur déclarée du tableau doit être surdimensionnée, et seule une partie du tableau est occupée par ces données, ce qui demande de gérer cette longueur occupée. Et si l'on désire ajouter ou enlever un élément d'un tableau, cela demande des déplacements de pans entiers du tableau soit pour faire un trou soit pour en combler un. Aussi dans certains contextes préfère-t-on utiliser des listes chaînées, plus élastiques et malléables.

Une liste chaînée est formée de « wagons » accrochés les uns aux autres. Un wagon contient diverses données et aussi le nom du wagon suivant (et parfois aussi celui du précédent). Voici comment se présente cette structure wagon, que l'on peut aussi appeler cellule, ou atome, ou du nom évocateur de ce qu'elle représente dans un problème concret.



A gauche, une liste doublement chaînée (avec un lien des deux côtés) de quatre cellules, et *à droite* la cellule de base contenant ici par exemple un entier n ainsi que le lien vers la cellule suivante et un lien vers la précédente.

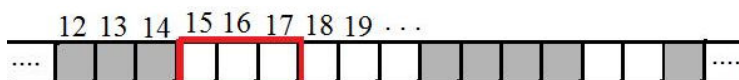
Une telle cellule est déclarée de la façon suivante :

```
struct cell { int n ;      /* on peut éventuellement mettre d'autres données */
              struct cell * s ;
              struct cell * p ;
            } ;
```

On pourrait croire que la cellule (*cell*) se rappelle elle-même deux fois dans sa définition, ce qui nous ferait entrer dans une boucle infinie, en retrouvant le paradoxe du menteur, mais en fait *struct cell ** n'est pas une structure mais un « pointeur » vers une cellule, ou encore l'adresse d'une cellule, comme cela va bientôt s'expliquer. Après cette déclaration de la cellule de base, passons aux actes. Faisons :

```
debut = (struct cell *) malloc (sizeof(struct cell)) ;
```

Comme son nom l'indique la fonction *malloc()* alloue une place dans la mémoire de l'ordinateur pour y placer une cellule *cell*. On peut imaginer que la mémoire de la machine est une très longue succession de cases portant des numéros –des adresses. Le système cherche une place disponible de la taille de la cellule. Et quand il la trouve, il nous ramène l'adresse de la première case de la cellule. C'est cette adresse qui est mise dans *debut*. La fonction *malloc()* place la cellule et nous dit où c'est.



La mémoire de l'ordinateur, la place prise par la cellule, le pointeur *debut* qui contient l'adresse 15, elle-même ramenée par la fonction *malloc()*.

Il reste à remplir les composantes de la cellule. Faisons par exemple :

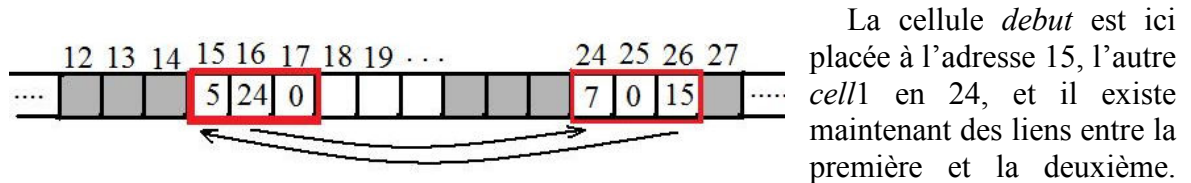
```
debut -> n = 2 ; debut -> s = NULL ; debut -> p = NULL ;
```

Cela signifie que nous avons mis dans la cellule l'entier 2, et qu'il n'y a pour le moment aucune cellule *s* suivante, ni aucune précédente *p*, d'où cette adresse *NULL* qui signifie « rien », et qui peut être remplacée par l'adresse 0 si l'on veut (même si normalement 0 n'a rien à voir avec le vide !).

Faisons maintenant notre première liste doublement chaînée avec deux cellules :

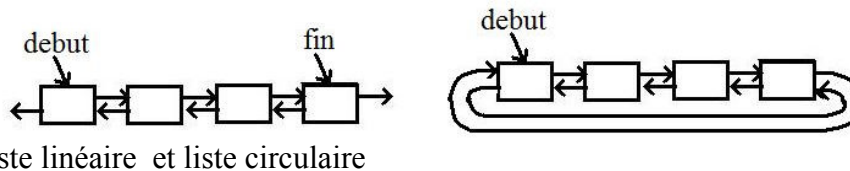
```
debut=(struct cell *) malloc(sizeof(struct cell)) ;
cell1=(struct cell *) malloc(sizeof(struct cell)) ;
debut->n=5 ; debut->s =cell1 ; debut->p=0 ;
cell1->n=7 ; cell1->s=NULL ; cell1->p=debut ;
```

Cela donne la situation suivante en mémoire :



Notamment si l'on accède à *debut*, on pourra ensuite aller vers *cell1* qui est la suivante *s*.

Ce que l'on a fait avec deux cellules peut être fait avec autant de cellules que l'on veut. Mais si l'on sait où commence la liste, grâce au pointeur *debut*, il faudra aussi savoir où elle finit, en mettant un pointeur *fin* sur la dernière cellule. Aussi préfère-t-on mettre en place une liste chaînée circulaire. En tournant sur la liste, lorsqu'on retombe sur *debut*, cela signifie que l'on a parcouru toute la liste.



Pour parcourir la liste circulaire, on utilise un pointeur *ptr* qui court : il part de *debut* puis avance dans la liste jusqu'à son retour à *debut*. Supposons que nous ayons déjà fabriqué une liste circulaire de *N* cellules contenant les nombres entiers de 1 à *N*. Si nous voulons les voir affichés sur l'écran, il suffit de faire le programme suivant :

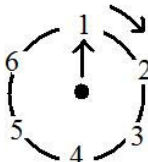
```
déclarer la cellule de base
main()
{ struct cell * debut, *ptr ;
  fabrication de la liste
  ptr=debut ;
  do { afficher ptr ->n ; ptr= ptr ->s ; } while (ptr !=debut) ;
}
```

Dans tous les exemples qui suivent, nous utiliserons des listes doublement chaînées circulaires par souci d'unité, même si cela n'est pas obligatoire. Il s'agit maintenant d'apprendre à créer une telle liste et à procéder à diverses modifications, c'est-à-dire ajouter

des éléments ou en enlever. Le premier exemple qui suit est parfaitement adapté à l'usage d'une liste chaînée circulaire.

1) Le problème de Josephus Flavius

Des prisonniers numérotés de 1 à N sont disposés en cercle. Un garde se trouve au centre, et se dirige vers le numéro 1. Il a pour mission d'éliminer chaque deuxième prisonnier qu'il rencontre en tournant sur le cercle dans le sens des numéros croissants. Et cela jusqu'à ce qu'il ne reste qu'un seul prisonnier, le survivant, dont nous noterons le numéro $S(N)$.



Ici, pour $N=6$, le 2 est éliminé (tout se passe comme si on l'enlevait du cercle), puis le 4, le 6, le 3 et le 1. Il reste le 5 : $S(6)=5$.

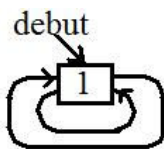
Nous allons programmer cette succession d'éliminations pour trouver finalement $S(N)$. Dans un premier temps il convient de créer la liste doublement chaînée des prisonniers. La cellule de base correspond à un prisonnier avec son numéro n :

```
struct cell { int n ; stuct cell * s ; struct cell * p ; } ;
```

Création de la liste par insertions successives

Cela se fait toujours en deux étapes :

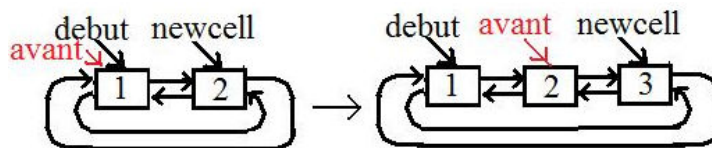
- Construction de l'embryon de la liste circulaire : On commence par créer une liste chaînée avec une seule cellule, celle du prisonnier 1 :



```
debut = (struct cell *) malloc(sizeof(struct cell)) ;
debut->n = 1 ; debut->s = debut ; debut->p = debut ;
```

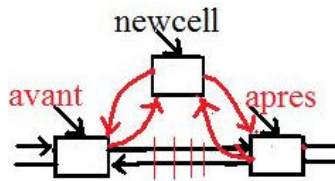
Désormais on pourra toujours accéder à la liste par le pointeur (l'adresse en mémoire) *debut*.

- Construction de la liste par insertions successives



Grâce à une boucle de $numero=2$ à $numero=N$, on ajoute à chaque fois une nouvelle cellule *newcell* correspondant à un prisonnier. Celle-ci vient s'insérer entre celle qui a été insérée le coup d'avant, notée *avant* (c'est un pointeur aussi) et celle d'après qui n'est autre que *debut*. Il y a alors quatre liens à mettre en place ou à changer :

```
avant = debut ;
for(numero=2 ; numero<=N ; numero++)
{ newcell = (struct cell *) malloc (sizeof (struct cell)) ;
  newcell->n = numero ; newcell->s = debut ; newcell->p = avant ;
  avant->s = newcell ; debut->p = newcell ;
}
```



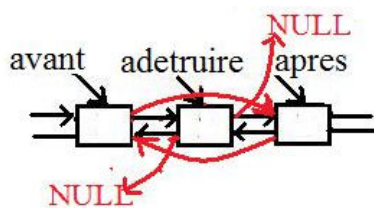
Pour insérer une cellule, on commence par préciser entre quelles cellules cela va avoir lieu, en général entre les cellules *avant* et *apres*, puis on procède à la mise en place des quatre liens, comme indiqué sur le dessin.

Une fois la liste chaînée construite, on a intérêt à procéder à l'affichage des numéros inscrits dans les cellules, pour vérifier que tout s'est bien passé. Pour cela, comme on l'a vu précédemment, on utilise un pointeur *ptr* qui court, de *debut* jusqu'au retour à *debut*, en avançant pas à pas par $ptr = ptr \rightarrow s$.

Destruction de cellules

Maintenant que la liste circulaire est construite, il s'agit de la parcourir comme le faisait le garde, en éliminant de la liste chaque deuxième cellule rencontrée, jusqu'à ce qu'il n'en reste qu'une.

Pour supprimer une cellule, voici comment procéder. Sachant que la cellule à détruire est placée entre deux cellules, celle d'avant et celle d'après, on commence par libérer la cellule à détruire de la chaîne en envoyant ses pointeurs *s* et *p* vers rien (*NULL*), puis on accroche la cellule d'avant à celle d'après. Les trois cellules concernées sont ciblées par les pointeurs *avant*,



adetruiere, *apres*.

On en déduit le programme :

```

adetruiere = debut->s ; avant = debut ; apres = adetruiere->s ; /* première cellule à détruire */
do
  { adetruiere->s = NULL ; adetruiere->p = NULL ;
    avant->s = apres ; apres->p = avant ;
    avant = apres ; adetruiere = avant->s ; apres = adetruiere->s ; /* on prépare l'étape suivante */
  }
while (adetruiere != avant) ;
afficher adetruiere->n /* c'est le survivant */

```

L'arrêt se produit lorsque les trois pointeurs *avant*, *apres*, *adetruiere* sont sur la même cellule, celle du survivant. Signalons que si l'on s'arrêtait dès que $avant = apres$, il resterait encore deux cellules.

Digression théorique

Cherchons la formule qui donne le survivant $S(N)$.

- Cas où N est pair : $N = 2P$. Après un tour complet où tous les numéros pairs sont éliminés, tout se passe comme si le garde recommençait sa tournée comme avant à partir de la cellule 1 en éliminant chaque deuxième rencontré. Les numéros sont maintenant 1, 3, 5, ..., $2P-1$. Renommons-les 1, 2, 3, ..., P . Pour retrouver la numérotation d'origine à partir de la nouvelle, il suffit de doubler le numéro et de lui enlever 1. Avec cette nouvelle numérotation le survivant est $S(P)$. Le numéro correspondant dans la numérotation d'origine est $2S(P) - 1$. D'où $S(2P) = 2S(P) - 1$.

- Cas où N est impair : $N = 2P + 1$. Après un tour, tous les pairs sont éliminés ainsi que le 1. Tout se passe comme si le garde commençait sa tournée à partir du numéro 3. les numéros sont maintenant 3, 5, 7, ... , $2P+1$. Renommons-les 1, 2, 3, P . Dans cette nouvelle numérotation, le survivant est $S(P)$. En revenant à la numérotation d'origine, cela donne : $S(2P+1) = 2 S(P) + 1$.

Le survivant $S(N)$ obéit aux relations de récurrence :

$S(2P) = 2 S(P) - 1$ et $S(2P+1) = 2 S(P) + 1$ avec au départ $S(1) = 1$. Pour trouver la formule explicite, on commence par vérifier, par récurrence évidente, que $S(2^k) = 1$. Puis on constate que la différence de deux termes successifs $S(2P+1) - S(2P)$ vaut 2.

D'où $S(2^{k+1}) = 3$.

Puis $S(2^{k+2}) = 2 S(2^{k+1} + 1) - 1 = 2.3 - 1 = 5$.

Puis $S(2^k + 3) = 5 + 2 = 7$.

Puis $S(2^k + 4) = 2 S(2^{k-1} + 2) - 1 = 2.5 - 1 = 9$.

Et l'on continue ainsi jusqu'à $S(2^k + 2^k - 1) = 2^{k+1} - 1$, avant de retrouver $S(2^{k+1}) = 1$.

Finalement, en écrivant $N = 2^k + p$, où 2^k est la plus forte puissance de 2 contenue dans N , avec p allant de 0 à $2^k - 1$, on a $S(N) = 2 p + 1$.

Avec cette formule à notre disposition, le programme précédent permet de vérifier expérimentalement la validité de la formule. Mais il perd beaucoup de son intérêt, sauf si l'on fait une animation visuelle avec les éliminations progressives. Enfin il est possible de modifier les modalités d'élimination des prisonniers, le garde pouvant par exemple faire deux pas en avant et trois pas en arrière, il suffit alors de légères modifications dans le programme précédent, tandis que l'on chercherait vainement une formule théorique.

2) Quelques exemples

a) Suite du style Fibonacci mais avec retard

Prenons une suite (u_n) obéissant à une relation de récurrence de la forme $u_{n+1} = u_n + u_{n-T}$ comme par exemple $u_{n+1} = u_n + u_{n-5}$ pour $T=5$. Il convient de se donner $T+1$ conditions initiales, de u_0 à u_T pour lancer la récurrence, en calculant u_{T+1} , puis u_{T+2} , ... Par exemple pour $T=5$, on se donne $u_0, u_1, u_2, u_3, u_4, u_5$. On peut alors calculer $u_6 = u_5 + u_0$, puis $u_7 = u_6 + u_1$, etc. Remarquons qu'une fois que l'on a u_6 on n'aura plus jamais besoin de u_0 , et qu'après avoir eu u_7 on n'aura plus besoin de u_1 . Cela va nous donner une méthode de programmation.

On va commencer par construire une liste doublement chaînée circulaire possédant $T+1$ cellules contenant les $T+1$ premiers termes donnés de la suite (u_n) , que nous prendrons tous égaux à 1.¹

Après avoir déclaré la structure de base notée *cell*, et contenant un terme u de la suite, soit :

```
struct cell { double u; struct cell *suivant; struct cell *precedent;};
```

on crée l'embryon de la liste, avec une cellule d'adresse *debut* et contenant u_0 :

```
debut=(struct cell*)malloc(sizeof (struct cell));
```

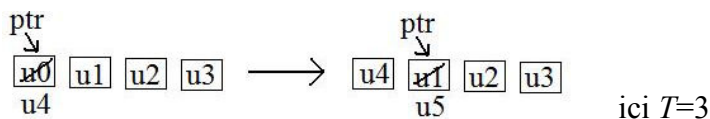
¹ Au lieu de prendre une liste chaînée circulaire, on pourrait aussi bien utiliser un tableau parcouru de façon cyclique, mais nous ne le ferons pas ici.

```
debut->u=1.; debut->suivant=debut; debut->precedent=debut;
```

Puis on insère les T cellules restantes l'une après l'autre, chacune entre celle qui a été insérée le coup d'avant (et appelée *avant*), et *debut*.

```
avant=debut;
for(i=1; i<=T; i++)
{ newc=(struct cell *) malloc(sizeof(struct cell));
  newc->u=1.; newc->suivant=debut; newc->precedent=avant;
  avant->suivant=newc; debut->precedent=newc;
  avant=newc;
}
```

Passons maintenant à l'évolution de la suite. Pour cela on va tourner sur la liste chaînée grâce à un pointeur *ptr*. Au départ celui-ci est placé en *debut*, là où se trouve u_0 . En ajoutant à u_0 ce qui se trouve dans la cellule précédente, à savoir u_T , on obtient u_{T+1} , et on met cette valeur à la place de u_0 dont on n'a plus besoin. Puis on avance le pointeur *ptr* d'un cran. Il tombe sur u_1 qui ajouté à celui qui le précède, à savoir u_{T+1} , donne u_{T+2} qui va écraser u_1 , etc. Autrement dit, quand *ptr* est sur une cellule, c'est là que va être placé le nouveau terme u_{n+1} , alors qu'il y avait jusque-là le terme bien avant (*ubienavant*), à savoir u_{n-T} (dans $u_{n+1}=u_n+u_{n-T}$), et celui-ci est précédé par l'élément juste avant (*ujusteavant*), à savoir u_n dans $u_{n+1}=u_n+u_{n-T}$.



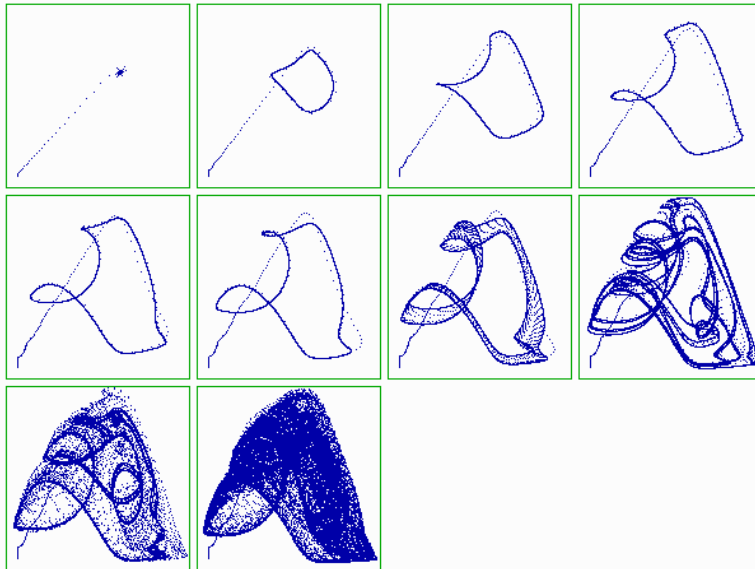
Dans le programme qui suit, nous n'avons pas pris la relation de récurrence $u_{n+1} = u_n + u_{n-T}$ mais une relation plus complexe, à savoir $u_{n+1} = f(u_n, u_{n-T})$, plus précisément :

$$u_{n+1} = (1 - \gamma)u_n + \frac{\lambda a^m u_{n-T}}{a^m + u_{n-T}^m}, \text{ où } u_{n+1} \text{ dépend toujours du terme juste avant et d'un}$$

terme bien avant. Comme valeurs des constantes, on prend : $\lambda=0,2$, $\gamma=0,1$, $m=20$, $a=10$. Ce genre de formule est censé modéliser la synthèse des globules sanguins, ou dans un registre différent, l'évolution d'une population de sauterelles. En effet ces dernières peuvent demeurer sous forme de larves pendant plus d'une dizaine d'années avant de prendre leur envol, d'où l'effet de retard. Plutôt que de visualiser u_n en fonction de n , c'est-à-dire l'évolution de la population en fonction du temps, on préfère se placer dans le repère u_{n-T} , u_n en plaçant les points de coordonnées u_{n-T} et u_n . Ce qui est l'ordonnée à un moment sera l'abscisse à T unités de temps plus tard, et le dessin se restreint à l'intérieur d'un carré.

```
ptr=debut;
for(compteur=T+1; compteur<50000; compteur++)
{ ujusteavant= (ptr->precedent)->u;
  ubienavant=ptr->u;
  dessiner sur l'écran le point (xo+zoom*ubienavant,yo-zoom*ujusteavant);
  ptr->u=f(ujusteavant, ubienavant); /* c'est u_{n+1} */
  ptr=ptr->suivant;
}
```

```
double f (double uja,double uba)
{ double am,resultat;
  am=pow(a,m); resultat=(1.-gamma)*uja+ lambda*am*pba/(q+pow(pba,m)); return resultat;
}
```



Trajectoires des points (u_{n-T}, u_n) pour diverses valeurs de T (entre 1 et 40)

On constate que la trajectoire des points obtenus converge vers une certaine forme, que l'on appelle un attracteur, car si l'on change les conditions initiales, dans certaines limites, la trajectoire finit toujours par s'enrouler sur ce même attracteur. Lorsque T augmente, de 1 à 40, cet attracteur est d'abord un point fixe, ce qui signifie que la population se stabilise, puis il devient circulaire, ce qui veut dire que la population ne cesse d'osciller entre deux extrêmes, puis l'attracteur se déforme, se dédouble, s'enroule sur lui-même et zigzague en devenant de plus en plus complexe, ce que l'on appelle un *attracteur étrange*. En augmentant encore la valeur de T , l'attracteur finit par s'éparpiller pour donner un nuage de points.

b) Séquences de Farey

Il s'agit d'une méthode qui permet d'obtenir toutes les fractions irréductibles comprises entre 0 et 1, dont le dénominateur est inférieur ou égal à un nombre donné, et cela dans l'ordre croissant. Pour cela on a besoin de définir ce que l'on appelle la médiane de deux fractions irréductibles $\frac{a}{b}$ et $\frac{c}{d}$, qui est la fraction $\frac{a+c}{b+d}$. On démontre que cette médiane est aussi irréductible et qu'elle s'intercale entre les deux fractions : $\frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d}$.

Voici comment on procède. On part des deux fractions 0/1 et 1/1, ce qui correspond à l'étape 1. Puis lors du passage de l'étape $n-1$ à l'étape suivante n , on intercale entre deux fractions successives de l'étape $n-1$ leur médiane, et cela tout au long de la séquence des fractions que l'on avait déjà à l'étape $n-1$, sous réserve que les médiantes aient un dénominateur inférieur ou égal au numéro de l'étape n . Cela donne :

- Etape 2 : 0/1 1/2 1/1 (on a intercalé la médiane 1/2 entre les deux fractions de l'étape 1 puisque son dénominateur est inférieur ou égal au numéro de l'étape, à savoir 2.
- Etape 3 : 0/1 1/3 1/2 2/3 1/1.
- Etape 4 : 0/1 1/4 1/3 1/2 2/3 3/4 1/1. On a intercalé 1/4 et 3/4 mais pas les médiantes 2/5 et 3/5 dont le dénominateur est trop grand.
- Etape 5 : 0/1 1/5 1/4 1/3 2/5 1/2 3/5 2/3 3/4 4/5 1/1.

On démontre –nous ne le ferons pas ici, que toutes les fractions irréductibles de dénominateur inférieur ou égal à n se trouvent dans la séquence des fractions à l'étape n .

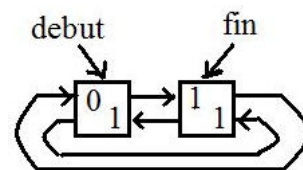
On veut programmer la fabrication de ces séquences de fractions jusqu'à l'étape N , N étant donné. Pour cela on va utiliser une liste doublement chaînée dont les cellules sont les fractions, avec leur numérateur et leur dénominateur.

Définition de la structure fraction :

```
struct f { int n;      /* numérateur */
          int d;      /* dénominateur */
          struct f * s;
          struct f * p;
        };
```

Création de l'embryon de la liste, avec les deux fractions de l'étape 1:

```
etape=1;
debut= (struct f *) malloc (sizeof (struct f));
fin= (struct f *) malloc (sizeof (struct f));
debut->n=0;debut->d=1;debut->s=fin; debut->p=fin;
fin->n=1;fin->d=1;fin->s=debut; fin->p=debut;
afficher la liste chainée
```



Remarquons qu'avec la présence des deux cellules *debut* et *fin*, entre lesquelles tout va se passer, le fait de rendre la liste circulaire n'est pas important. On aurait pu aussi bien faire *debut->p=NULL* et *fin->s=NULL*.

La boucle des étapes:

A chaque étape, on parcourt la liste grâce à un pointeur courant *ptr* et, le cas échéant, on insère une nouvelle cellule *newf* correspondant à la médiane, en choisissant de la placer avant la cellule sur laquelle pointe *ptr* (ainsi *ptr* commence sur *debut->s* et on intercale la nouvelle cellule entre la cellule *avant* et *ptr*).

```
for(etape=2;etape<=N;etape++)
{ ptr=debut->s;
  while(ptr!=debut)
  { avant=ptr->p;
    if (avant->d + ptr->d <=etape)
    { newf=(struct f*)malloc(sizeof(struct f));
      newf->n=avant->n + ptr->n; newf->d=avant->d + ptr->d;
      newf->p=ptr->avant; newf->s= ptr;
      avant->s=newf; ptr->p=newf;
    }
    ptr=ptr->s;
  }
  afficher la liste à l'étape où l'on est
}
```

c) Evolution d'une organisation fortement hiérarchisée

On part d'un groupement de N personnes, toutes de grade 0 à l'instant initial (*temps*=0). A chaque étape de temps (*temps*=1, *temps*=2, ..., chaque année si l'on veut), chaque individu a deux possibilités :

- Soit il réussit à faire entrer un nouvel adhérent dans le groupe, et son grade augmente de 1, tandis que le nouvel adhérent est mis au grade 0,

- Soit il est exclu définitivement du groupe, puisqu'il n'a pas réussi à faire adhérer un nouveau membre.

On constate au fil du temps que la première éventualité a 4 chances sur 10 de se produire, et la deuxième 6 chances sur 10. Avec de telles probabilités, le groupe est condamné à disparaître en un temps fini (cela se démontre).

Une telle évolution va être programmée grâce à une liste doublement chaînée circulaire. Chaque cellule de la liste correspond à un individu du groupe avec son grade.

- 1) Au départ ($temps=0$), le nombre de cellules de la liste est $nombre=N$. A chaque étape de temps, la variable $nombre$ évolue. Quel est le test d'arrêt du processus ?

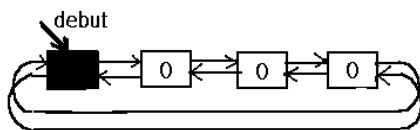
Le test d'arrêt est $nombre=0$.

- 2) Donner la structure de la cellule de base de la liste, seul le grade de l'individu correspondant à la cellule nous intéresse.

```
struct cell { int grade ; struct cell * s ; struct cell * p ; } ;
```

- 3) Programmer pour fabriquer la liste doublement chaînée initiale, avec ses N individus tous au grade 0. On rajoutera une cellule factice (aucun individu n'y est associé) correspondant au pointeur $debut$ (si $debut$ pointait sur un individu, et que cet individu soit éliminé, il faudrait raccrocher $debut$ sur un autre individu de la liste pour pouvoir accéder à celle-ci). Cette cellule factice indique le point de départ de la liste et restera jusqu'à la fin.

```
debut=(struct cell *) malloc(sizeof(struct cell)) ;
avant=debut ;
for(i=0 ; i<N ; i++)
{ cellule= (struct cell *) malloc(sizeof(struct cell)) ;
cellule->s=debut ; cellule->p = avant ; cellule->grade=0 ;
avant->s=cellule ;debut->p=cellule ;
avant=avant->s ;
}
nombre=N ;
```



Cas où $N=3$

- 4) A chaque étape de temps, on parcourt la liste circulaire grâce à un pointeur courant ptr . Pour chaque individu à tour de rôle, on fait un tirage au sort du style :

$nombreauhasard = rand() \% 10$, ce qui donne un chiffre aléatoire entre 0 et 9.

Si $nombreauhasard < 6$, on supprime la cellule de la liste (cela correspondant à 6 chances sur 10). Sinon on garde la cellule en faisant monter le grade de l'individu, et l'on adjoint à ses côtés un nouvel individu (une nouvelle cellule) au grade 0. Programmer cette suppression ou cette insertion. Puis donner le programme total, jusqu'à l'arrêt définitif, de façon à obtenir la durée de vie du groupe, en affichant à chaque étape le nombre des individus ainsi que la liste de leur grade.

```
temps=0 ;
while(nombre !=0)
```

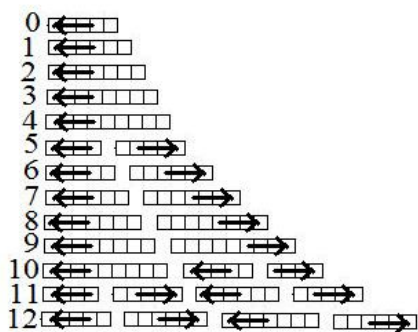
```

{ temps++ ;
  ptr=debut->s ;
  while(ptr !=debut)
  { nbhasard=rand()%10 ;
    if (nbhasard<6)
      { nombre-- ;avant=ptr->p ;apres=ptr->s ;avant->s=apres ;apres->p=avant ; free(ptr) ;
        ptr=apres ;
      }
    else{ (ptr->grade)++ ; apres=ptr->s ;
      newcell==(struct cell *) malloc(sizeof(struct cell)) ;
      newcell->grade=0 ;newcell->s=apres ; newcell->p=ptr ;
      pr->s=newcell ; apres->p = newcell ;
      nombre++ ; ptr=apres ;
    }
  }
}

```

d) Division cellulaire de l'algue *Anabaena Catenula*

Il s'agit d'une algue microscopique qui grossit et se divise, et dont on étudie l'évolution en ligne. Elle a la particularité de présenter une orientation à gauche ou à droite dans sa forme. A l'étape 0, on part d'une cellule orientée à gauche et de taille 4. Celle-ci grandit d'une longueur unité à chaque étape de son évolution. A l'étape 5, elle atteint la taille 9, qui est sa dimension maximale. Alors, à l'étape 6, elle se divise en deux, donnant naissance à une cellule orientée à gauche de taille 4 suivie d'une cellule orientée à droite de taille 5. A leur tour ces cellules grossissent de 1 à chaque étape jusqu'à la taille maximale 9. On a déjà vu ce qu'il arrivait à une cellule orientée à gauche. Une cellule orientée à droite ayant atteint la taille 9 donne naissance en se divisant en deux, à une cellule orientée à gauche de taille 5 suivie d'une cellule orientée à droite de taille 4. Remarquons que lorsqu'une cellule se divise, c'est toujours celle de même sens qui est la plus petite, de taille 4. Remarquons aussi que lors de cette division, la cellule « de gauche » est toujours à gauche de la cellule « de droite ».



On veut réaliser la simulation d'une telle évolution sur ordinateur en utilisant une liste chaînée circulaire. On notera le sens gauche par -1 et le sens droit par 1. Chaque cellule possède une taille et un sens, et la situation à l'étape 12 par exemple s'écrira :

(5,-1)(6,1)(7,-1)(6,1).

1) Définir la structure d'une cellule.

```

struct cell { int taille;
              int sens;

```

```

struct cell * suivant;
struct cell * precedent; };

```

2) Programmer les conditions initiales (étape 0)

```

debut=(struct cell *) malloc(sizeof (struct cell));
debut->taille=4; debut->sens=-1;debut->suivant=debut;debut->precedent=debut;
printf("etape 0: (%d %d)",debut->taille,debut->sens);

```

3) On traite maintenant l'évolution étape par étape jusqu'à l'étape N , N étant donné. Pour passer d'une étape à la suivante, un pointeur ptr fait le tour de la liste en faisant les modifications au passage. Lors de la division en deux, une seule nouvelle cellule est ajoutée, l'autre nouvelle prenant la place de l'ancienne de même sens. Si une cellule de sens -1 se divise, la nouvelle cellule insérée se place après, tandis que si la cellule est de sens 1, la nouvelle cellule à insérer (de sens -1) se place devant. Faire le programme.

```

for( etape=1; etape<=N; etape++) /* la grande boucle des étapes */
{ ptr=debut;
do
{ if (ptr->taille <9) ptr->taille++; /* on augmente la taille */
else /* division en deux */
{ ptr->taille=4;
newcell=(struct cell *) malloc( sizeof (struct cell));
newcell->taille=5;
if (ptr->sens==1) newcell->sens=1; else newcell->sens=-1;
if (ptr->sens==1)
{ newcell->suivant=ptr->suivant; newcell->precedent=ptr;
ptr->suivant->precedent=newcell; ptr->suivant=newcell;
ptr=ptr->suivant;
}
else
{ newcell->suivant=ptr; newcell->precedent=ptr->precedent;
ptr->precedent->suivant=newcell;
ptr->precedent=newcell;
}
}
ptr=ptr->suivant;
}
while (ptr!=debut);
afficher la liste des cellules avec leur taille et leur sens
}

```

e) Tri par compartiments

Supposons que l'on ait N nombres à trier et que chacun s'écrive avec K chiffres au maximum : ces nombres sont compris entre 0 et $10^K - 1$. Mieux vaut que K soit nettement inférieur à N . Par exemple $N=10000$ et $K=3$, les mêmes nombres se retrouvant alors plusieurs fois. Il existe une méthode de tri très simple : on commence par classer les nombres selon leur chiffre le plus à gauche (celui des centaines si $K=3$), comme on fait quand on cherche un mot dans un dictionnaire. Puis dans chacun des 10 paquets ainsi formés on classe les nombres selon leur deuxième chiffre. Et ainsi de suite jusqu'au chiffre des unités. C'est là une méthode récursive. Pour la programmation, on fait plutôt l'inverse.

On définit 10 compartiments numérotés de 0 à 9. En parcourant la liste des données, mises dans un tableau $a[N]$, on commence par placer au fur et à mesure les nombres dans les compartiments correspondants suivant leur chiffre des unités, ce qui donne une liste dans chaque compartiment. Puis on concatène ces dix listes pour obtenir une nouvelle liste que l'on met dans le tableau $a[]$. A partir du parcours de cette liste, on recommence en plaçant les nombres dans les compartiments suivant leur chiffre des dizaines, puis on concatène... Et ainsi de suite, soit K fois. A la fin, la liste est triée.

Prenons un exemple avec $N=10$ et $K=2$. On prend la liste :

27 59 33 61 32 13 22 63 25 57.

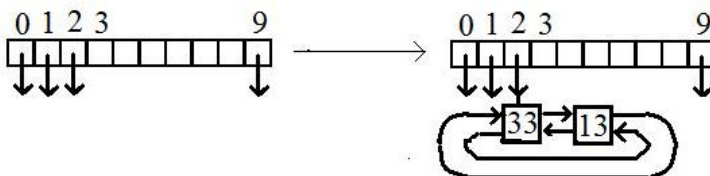
Le classement suivant le chiffre des unités donne les 10 compartiments {}, {61}, {32 22}, {33 13 63}, {}, {25}, {}, {27 57}, {}, {59}. Après concaténation on a la liste 61 32 22 33 13 63 25 27 57 59. On recommence avec les chiffre des dizaines : {}, {13}, {22 25 27}, {32 33}, {}, {57 59}, {61 63}, {}, {}, {}. Après concaténation, la liste est triée :

13 22 25 27 32 33 57 59 61 63.

Le programme se déroule en K étapes, et à chaque fois on remplit les 10 compartiments puis on les concatène.

Mais d'abord comment dégager le chiffre des unités, puis celui des dizaines, etc., d'un nombre ? Prenons par exemple le nombre 397. Son chiffre des unités est obtenu en faisant $397\%10 = 7$ (le reste de la division par 10). Pour avoir le chiffre des dizaines, on commence par diviser le nombre par 10 (division euclidienne), soit ici 39 puis on prend le chiffre des unités, soit $39\%10=9$. Et pour le chiffre des centaines, on divise par 100 et avec le nombre obtenu, on prend son reste en le divisant par 10, soit $(397/100)\%10=3$. Dans tous les cas on fait $(397/pas)\%10$, avec pas qui vaut 1 puis 10 puis 100, ... c'est-à-dire que pas est multiplié par 10 à chaque étape.

Pour le programme on va fabriquer à chaque étape 10 listes chaînées circulaires, une pour chaque compartiment. Au début de chaque étape, on crée un tableau de pointeurs $debut[]$ qui pointent vers rien ($NULL$). Puis en parcourant le tableau $a[N]$ on fabrique les 10 listes chaînées en construisant d'abord l'embryon de la liste avec la mise en place de la première cellule, puis en insérant les autres cellules toujours entre le précédent de $debut[]$ et $debut[]$.



Le tableau des pointeurs $debut[]$ au démarrage de chaque étape puis l'accrochage des listes chaînées par insertions successives (on n'en a dessiné qu'une)

Programme (complet pour une fois !)

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#define K 3
#define N 200
/* la cellule de base appelée ici w */
struct w { int n; struct w * s; struct w * p;}; struct w * debut[10],* wa, *ptr, * oldptr, * avant;
int a[N];
```

```

main()
{
int i,etape,pas,chiffre,k;
clrscr(); randomize(); printf("LISTE INITIALE:\n");
/* on remplit le tableau a[N] avec des nombres pris au hasard entre 0 et 10^K -1 */
for(i=0;i<N; i++) { a[i]=random((int)pow(10.,(double)K)); printf("%d ", a[i]); }
pas=1;
for(etape=0;etape<K;etape++) /* les K grandes étapes du tri */
{ for(i=0;i<10;i++) debut[i]=NULL;
for(i=0;i<N;i++)
{ chiffre=(a[i]/pas)%10; wa=(struct w *) malloc(sizeof(struct w)); wa->n=a[i];
/* création de l'embryon de la liste */
if(debut[chiffre]==NULL) { debut[chiffre]=wa; debut[chiffre]->s=debut[chiffre];
debut[chiffre]->p=debut[chiffre]; }
/* insertion de la cellule entre le précédent de début [chiffre] et début [chiffre] */
else { avant=debut[chiffre]->p ; wa->s=debut[chiffre];
wa->p=avant; avant->s=wa;debut[chiffre]->p=wa ; }
}
k=0; /* concatenation des 10 listes chaînées sauf celles réduites à rien */
for(i=0;i<10;i++) if (debut[i]!=NULL)
{ ptr=debut[i];
do { a[k++]=ptr->n; oldptr=ptr; ptr=ptr->s; free(oldptr); } while(ptr!=debut[i]);
/* noter qu'on libère de la place en mémoire à chaque étape, pour avoir N cellules
occupées à chaque étape, et éviter d'en avoir KN à la fin du programme */
}
printf("\n\n"); if (etape==K-1) printf("LISTE FINALE:\n");
for(i=0;i<N;i++) printf("%d ",a[i]); /* affichage des listes obtenues à chaque étape */
pas=pas*10;
}
getch();
}

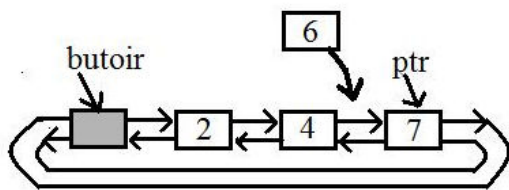
```

Performance de ce tri

A chaque étape, on fait en quelque sorte deux parcours de la liste, soit un total de $2KN$ opérations élémentaires. Dès que N est grand par rapport à K , ce tri a une performance de l'ordre de N . On ne peut guère faire mieux en matière de tri, mais cela suppose que $K \ll N$.

f) Tri par insertion au coup par coup

Nous avons déjà vu le tri par insertion. Nous en donnons ici une variante : les cartes sont distribuées une par une et à chaque fois, on range la carte parmi celles que l'on a déjà en main. Par exemple on commence par recevoir la carte 7, puis on obtient le 2, que l'on range devant le 7 pour avoir 2 7, puis on a le 4 que l'on range de façon à avoir en main 2 4 7, etc. Le processus d'insertion nous invite à utiliser une liste chaînée pour la programmation. Comme le début de la liste peut changer au cours des insertions, on va rajouter une case butoir factice en début de liste.



Pour insérer une nouvelle carte, on fait circuler un pointeur *ptr* à partir de la première carte, soit *butoir->s*, et on le fait avancer tant que la carte sur laquelle il pointe est inférieure à la carte à insérer, comme dans l'exemple ci-contre. Quand le pointeur se bloque, on doit insérer la nouvelle carte juste avant lui.

Mais le pointeur s'arrête-t-il toujours ? Oui, à condition de mettre dans la case *butoir* une grande valeur. Ainsi lorsque la carte à insérer est plus grande que toutes les autres, le pointeur se bloque sur le *butoir*, et on insère la carte juste avant, en dernière position. Et lorsqu'on part de l'embryon de la liste circulaire qui est réduit au *butoir*, *butoir->s* n'est autre que *butoir*, *ptr* se bloque sur *butoir*, et l'insertion de la première carte se fera avant le *butoir*, c'est-à-dire en première place (à noter que si l'on mettait une valeur petite, nulle par exemple, dans le *butoir*, on entrerait dans une boucle infinie, où *ptr* ne cesserait de se mettre sur *butoir*, sans jamais s'arrêter). Le programme en découle.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 200
int a[N];
struct cell { int n; struct cell * s; struct cell * p;};
struct cell * butoir, *ptr, *cell1, *newcell, *avant;

int main()
{ int i;
  srand(time(NULL)); for(i=0;i<N;i++) a[i]=rand()%100;
  butoir=(struct cell *) malloc(sizeof(struct cell));
  butoir->n=100000; butoir->s=butoir; butoir->p=butoir;
  for(i=0;i<N;i++)
  { ptr=butoir->s;
    while( a[i]>ptr->n) ptr=ptr->s;
    avant=ptr->p;
    newcell=(struct cell *) malloc(sizeof(struct cell));
    newcell->n=a[i]; newcell->s=ptr; newcell->p=avant;
    avant->s=newcell; ptr->p=newcell;
    afficher la liste
  }
  return 0;
}
```