Les structures en C

Les structures sont des définitions de type, et à ce titre ne donnent pas lieu à du code compilé (ce ne sont pas des instructions mais des descriptions, tout comme les prototypes de fonctions). Ces définitions de type se trouvent donc dans des fichiers .h

Syntaxe du C : il suffit de remplacer **structure** par **struct**

```
struct nom_du_type
{
        liste des champs;
};
```

cependant en C, le nom du type comporte le mot struct.

Exemple avec le type complexe, défini dans le fichier complexe.h (pour illustration rédiger complètement ce fichier en y incluant la définition de type ainsi que les prototypes des fonctions saisir, afficher, addition, module, multiplication)

Dans le fichier .h :

```
Struct complexe
{
        double re,im;
};
```

dans une fonction:

```
int main()
{
        struct complexe z; // définition de variable
};
```

donc syntaxe lourde…toujours écrire **struct** !!

pour alléger cela, utiliser l'instruction **typedef** qui permet de créer un raccourci entre le nom complet de la structure et un nom raccourci permettant de ne pas réecrire **struct** à chaque fois.

Attention cette instruction **typedef** porte mal son nom : **typedef** fait penser à type définition : définition de type, ce qui n'est pas vrai ! c'est l'instruction **struct** qui permet de définir un type, **typedef** n'est là que pour créer le raccourci.

Syntaxe d'emploi de typedef :

```
typedef struct s_nom // le nom 'normal' de la structure
{
        liste des champs;
} nom_raccourci;
```

exemple :

```
typedef struct s_complexe
{
	double reel, imag;
} t_complexe;
```

dans toute fonction, pour définir une variable de ce type, on pourra maintenant utiliser le nom raccourci **t_complexe** à la place du nom complet **struct s_complexe**.

Exemple :

```
int main()
{
	t_complexe z1, z2;
	// équivalent à struct s_complexe z1, z2;

	// autres instructions

	return 0;
}
```

les noms raccourcis multiples

avec typedef, il est en fait possible d'associer plusieurs noms raccourcis à une structure. Cela ne semble présenter que peu d'intérêt, car pourquoi définir deux raccourcis pour un même type ? L'intérêt réside dans le fait que l'on peut utiliser ces noms raccourcis pour définir des pointeurs vers les structures.

Exemple :

```
typedef struct s_complexe
{
	double reel, imag;
} t_complexe, *ptr_comp;
```

ce dernier raccourci permet de définir un pointeur vers une structure de type t_complexe en utilisant le nom ptr_comp.

Ainsi, pour définir un pointeur vers une variable de type t_complexe, les 3 écritures suivantes sont équivalentes :

```
int main()
{
	struct s_complexe	*p_z;
	// ou
	t_complexe		*p_z;
	// ou encore
	ptr_comp		 p_z;
}
```

à titre d'illustration : le fichier complexe.h :

```
#ifndef COMPLEX
#define COMPLEX "__COMPLEX"

// définition des types du module

typedef struct s_complexe
{
        double re;
        double im;
} t_complexe;

// prototypes des fonctions du module

void affiche_comp(t_complexe *);
void saisie_comp (t_complexe *);
t_complexe *addition(t_complexe *, t_complexe *);
double module(t_complexe *);

#endif // COMPLEX
```

La récursivité

Définition :

Récursivité : voir récursivité.

Une fonction récursive est une fonction qui s'appelle elle-même : tout à fait possible en informatique, car une fonction peut appeler n'importe quelle autre fonction (donc elle-même). Technique de programmation puissante et élégante mais difficile à maîtriser, car le code écrit n'est plus itératif, peu intuitif avec le genre de langages que nous traitons. Les langages LISP et SCHEME sont des langages basés sur la récursivité.

On rencontre la récursivité dans les fractales, mais aussi dans la nature :

Le chou romanesco est un légume récursif

Le découpage des côtes marines est aussi un exemple de phénomène récursif.

On dit d'un phénomène qu'il est **récursif** lorsque, pour le définir, on se réfère à lui.

Certains problèmes mathématiques se définissent de manière récursive.

Exemple : le calcul de la factorielle

Il existe deux manières de définir le calcul d'une factorielle : une manière itérative "classique" telle que nous l'avons déjà rencontrée :

$n! = \prod_{i=1}^{n} i$, ce type de définition se traite sans peine à l'aide d'une boucle **pour**

une manière récursive :
$n! = n.(n\text{-}1)$!
où $n!$ est définie en fonction de $(n\text{-}1)!$

Ce type de définition se traite à l'aide d'une fonction récursive, point que nous aborderons après avoir parlé d'une chose essentielle en ce qui concerne la récursivité : les conditions de fin de récursivité.

Les conditions de fin de récursivité.

La définition récursive de n! est en fait incomplète, car si l'on cherche à calculer 3! avec cet exemple, on obtient 3! = 3.2!, puis 2! = 2.1! puis 1! = 1.0!, 0! = 0.-1!, et ainsi de suite sans limite. Ce calcul ne s'arrête jamais ! il n'est donc pas correct.
Il ne faut pas oublier de traiter dans ce cas les deux valeurs particulières pour lesquelles la valeur de la factorielle est connue. Dans le cas où n vaut 0 ou 1, n! ne se calcule pas par définition récursive: la valeur de n! est connue. 1! = 1, 0! = 1 par définition.

Ainsi, la définition récursive complète de la factorielle est la suivante :

$$\begin{cases} \text{Si } n=0 \text{ ou } 1,\ n!=1 \\ \text{Sinon } n\ != n.(n\text{-}1)! \end{cases}$$

Où l'on voit apparaître un si…sinon très facilement utilisable dans une fonction !

La rédaction de la fonction facto est alors

```
fonction facto(entrée : entier n→ sortie : entier)
{
      entier res;

      si ((n=0) OU (n=1)) alors // test de la condition de fin
      {
            res ← 1;
      }
      sinon
      {
            res ← n*facto(n-1); // appel récursif de facto
      }

      retourner res;
}
```

Avant de vous lancer dans la rédaction d'une fonction récursive (fortement propice à un mal de crâne), il faut impérativement disposer des conditions d'arrêt et ne pas oublier de les programmer dans la fonction récursive, sinon, elle sera sans fin !

Fonction facto ayant oublié les conditions d'arrêt

```
fonction facto(entrée : entier n → sortie : entier)
{
      entier res;

      res ← n*facto(n-1);

      retourner res; // instruction jamais atteinte !
}
```

ou encore

```
fonction facto(entrée : entier n → sortie : entier)
{
      retourner(n*facto(n-1));
}
```

Mise en attente et exemplaires de fonctions

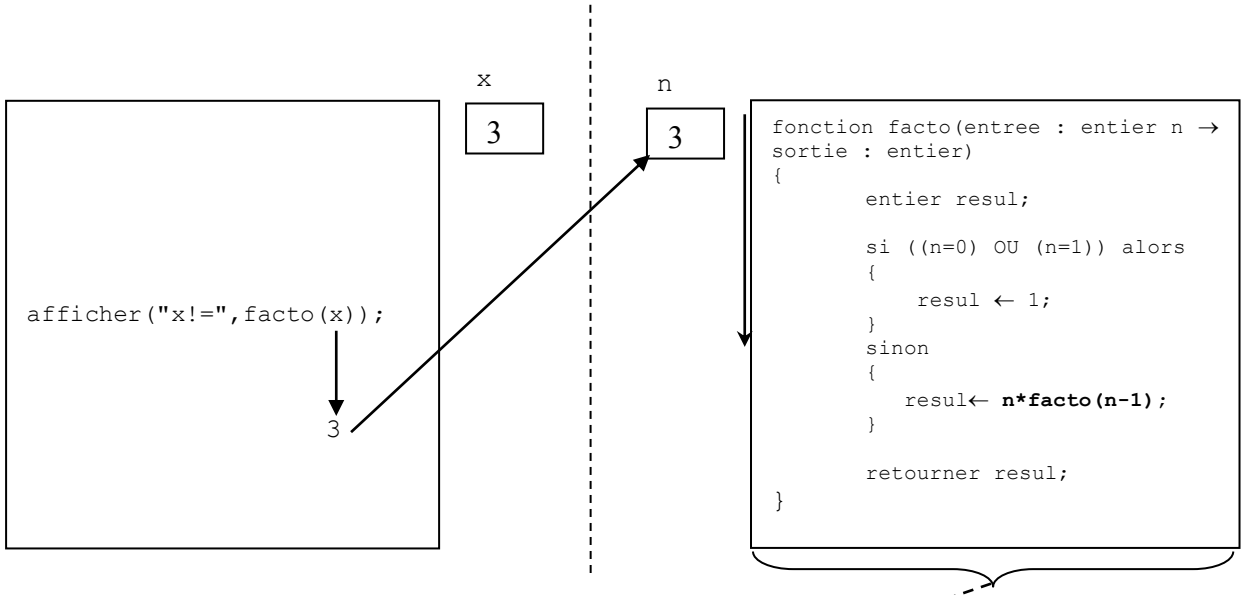
Lorsqu'une fonction est appelée depuis une autre fonction, nous avons déjà vu que la fonction appelante s'arrête, et que la fonction appelée s'exécute. Une fois celle-ci terminée, la fonction appelante reprend juste après l'appel. Qu'en est-il pour une fonction récursive qui s'appelle elle-même ?
Le mécanisme est exactement le même, à cette différence près que lors d'un appel récursif, un nouvel exemplaire de la fonction est créé avec ses propres variables.

Pour illustrer ceci, voyons comment l'ordinateur exécute la fonction facto. Soit le programme complet suivant :

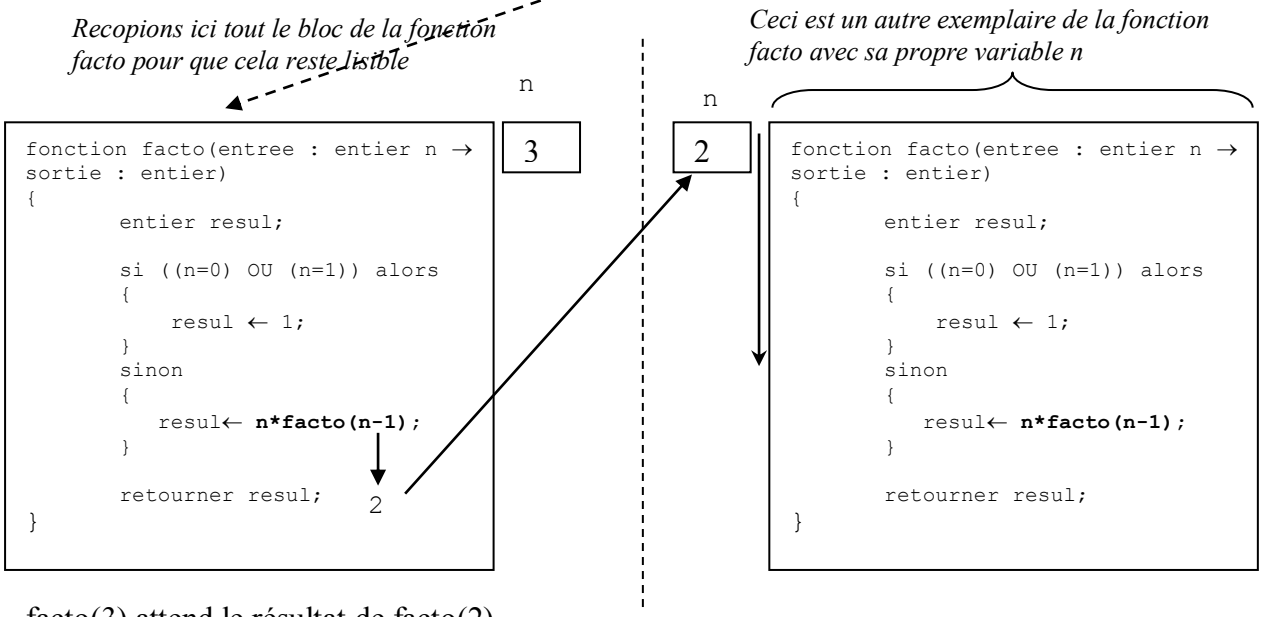
```
fonction facto(entree : entier n → sortie : entier)
{
      entier resul;

      si ((n=0) OU (n=1)) alors
      {
            resul ← 1;
      }
      sinon
      {
            resul ← n*facto(n-1);
      }

      retourner resul;
}

fonction principale()
{
      entier x;
```

```
        afficher("entrez la valeur de x:");
        saisir(x);
        afficher("x!=", facto(x));

        retourner;
}
```

faisons le schéma du premier appel à la fonction facto, en considérant que l'utilisateur a entre 3 comme valeur pour `x` (et vous vous rendrez compte avec soulagement qu'il n'a pas entré la valeur 8 par exemple).



Le test (n=0) ou (n=1) dans la fonction est faux (car n=3) donc l'appel récursif est fait



facto(3) attend le résultat de facto(2)

Sur le même principe, facto(2) fait appel à facto(1) et se met donc en attente.

facto(1) vaut 1 (car on atteint la condition d'arrêt, les test est vrai et il n'y a plus d'appel récursif). Facto(1) se termine et retourne 1 à la fonction qui l'a appelée : c'est la fonction facto(2).
Donc facto(2) reprend et peut calculer 2*facto(1) : 2. facto(2) se termine et retourne la valeur 2 à la fonction qui l'avait appelée : facto(3).
Facto(3) reprend et calcule 3*2 : 6, puis retourne cette valeur à la fonction qui l'avait appelée : afficher() : le résultat 6 s'affiche à l'écran.

On constate bien que l'on obtient le résultat attendu (ce qui est la moindre des choses…) et que l'écriture de la fonction est courte et très fidèle à la définition récursive complète du problème.

Par acquit de conscience, voici la version itérative de la factorielle, basée sur la définition

$$n! = \prod_{i=1}^{n} i$$

```
fonction facto(entrée : entier t → sortie : entier)
{
      entier f, cpt;

      f ← 1;

      pour cpt de 1 à t
      {
            f ← f*cpt;
      }

      retourner f;
}
```

Quid des performances comparées de ces deux versions du calcul de la factorielle ? La performance en terme de temps de calcul et en terme de consommation de la mémoire est meilleure pour la version itérative, car il n'y a pas d'appel de fonctions ni créations de variables locales à chaque appel récursif. Pour ce problème simple, la version itérative est "meilleure" de ce point de vue-là.

Il ne faut pas oublier que la simplicité de cet exemple le rend propice à faire une explication détaillée de ce qui se passe. Cela sera très utile pour la suite.
La facilité à écrire un programme d'après l'énoncé du problème fait également partie de la performance globale. Ainsi un programme que l'on écrira simplement de manière récursive mais qui est lent sera considéré comme "meilleur" qu'une version itérative plus rapide de ce programme qui demande un programme compliqué à écrire et dur à mettre au point.

Il n'y a pas de "meilleure" méthode dans l'absolu, mais une méthode adaptée à chaque problème.

Il existe même des programmes dits sous-optimaux traitant des problèmes tellement complexes que l'obtention d'une solution exacte prendrait trop de temps (quelques dizaines de million d'années). On se contentera dans ce cas d'une solution approchée mais calculable en un temps raisonnable (quelques jours ou semaines).

Exemple de problème mis sous forme récursive : l'addition

$$\begin{cases} a+b = a+1 \text{ si } b=1 \\ a+b = (a+1)+(b-1) \text{ sinon} \end{cases}$$

```
fonction addition(entrée : entier a, entier b → sortie : entier)
{
      entier resul;

      si (b=1) alors
      {
            resul ← a+1;
      }
      sinon
      {
            resul ← addition(a+1, b-1);
      }

      retourner resul;
}
```

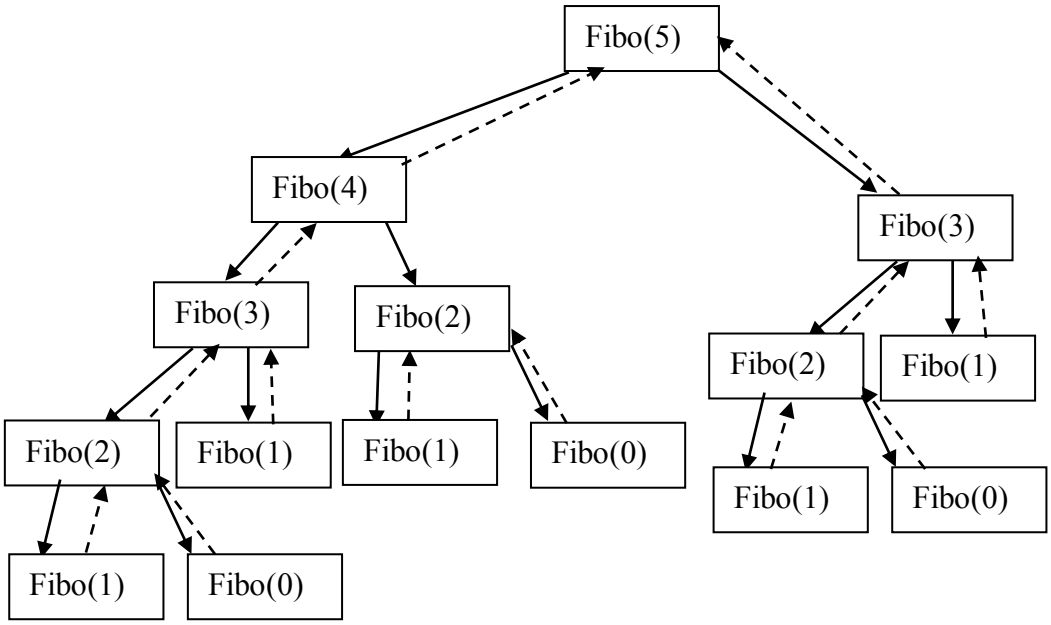
même si ce n'est pas une formulation naturelle, cela fonctionne très bien !

De nombreux problèmes s'écrivent naturellement de manière récursive, nous en traiterons une brève partie dans le cours consacré aux listes chaînées et vous aurez l'occasion d'utiliser la récursivité à maintes reprises en L2 lors du cours de SDD

un commentaire sur l'exercice sur la suite de Fibonacci :

```
fonction fibo(entrée : entier a → sortie : entier)
{
   entier pop;

   si ((n=0) OU (n=1)) alors
   {
         pop ← n;
   }
   sinon
   {
         pop ← fibo(a-1)+fibo(a-2);
   }

   retourner pop;
}
```

exemple d'appel avec fibo(5) (une flèche pleine matérialise un appel, une flèche en pointillés matérialise un retour de fonction



En tout, 15 appels de fonction !

Les listes chaînées

Les listes chaînées sont une alternative aux tableaux pour le stockage d'informations. Dans une liste chaînée, toutes les valeurs stockées ont le même type. Quelle peut être l'utilité d'une telle organisation de données qui n'apporte pas plus d'informations qu'un tableau ?

Cette question est plus importante qu'il n'y paraît, puisqu'elle concerne tout type de données que l'on peut stocker dans un tableau, notamment des structures. Si on se limite uniquement au tableau pour effectuer un stockage de valeurs, on peut en déduire qu'une base de données, quelle qu'elle soit, n'est qu'un ensemble de tableaux…ce qui est bien loin d'être le cas !

En effet, une organisation des données se caractérise par l'efficacité (la complexité) des traitements que l'on cherche à effectuer. Ces traitements sont assez généraux et il est possible d'en citer quelques-uns :
afficher tous les éléments stockés,
rechercher un élément selon un critère,
trier les éléments selon un critère,
insérer un élément,

supprimer un élément.

Considérons un stockage d'éléments sous la forme d'un tableau : parmi les traitements listés ci-dessus, certains sont efficaces, d'autres ne le sont pas du tout !
Par exemple, insérer ou supprimer un élément présent dans un tableau est une tâche simple, mais qui nécessite de décaler de nombreux éléments. Par exemple, pour supprimer le premier élément d'un tableau comportant un million d'éléments, il faudra procéder à 999 999 décalages de valeurs, ce qui peut s'avérer lent, surtout si chaque élément est une structure !

A ce titre, il est primordial de s'intéresser à d'autres organisations. L'une d'entre elles est la liste chaînée.

Le principe d'une liste chaînée est simple : un élément ou cellule (que l'on peut également appeler maillon ou nœud) d'une liste chaînée stocke deux informations de nature distincte : la première information est la valeur de la donnée , la seconde est l'emplacement de la cellule suivante. Ainsi, une valeur n'est plus repérée par un indice (comme dans un tableau), mais par sa position dans la liste.

Qu'est-ce qu'une cellule concrètement ?
Puisqu'une cellule contient deux informations de nature différente, il est bien évident qu'il s'agit d'une structure. Prenons comme exemple une liste chaînée stockant des entiers (cela a l'avantage d'être simple tout en permettant de présenter intégralement l'utilité des liste chaînées. En réalité, le type des valeurs stockées n'a aucune importance).
Chaque cellule devra donc stocker : un entier, ainsi que le moyen d'accéder à la cellule suivante. Ce moyen d'accès est fourni par l'adresse en mémoire de la cellule suivante.

Ainsi, la définition du type cellule est la suivante :

```
structure cellule
{
      entier valeur;
      cellule *suivante;
};
```

Attention, cette définition n'a que peu de rapport avec la récursivité que nous avons déjà traité. Cette définition n'est pas récursive. En réalité, elle ne peut pas l'être, car une structure

doit avoir une taille fixe en mémoire (cela est la condition sine qua non pour pourvoir utiliser des pointeurs vers des structures).

A ce titre, la définition de cellule suivante est **fausse** et ne peut être réalisée par un ordinateur :

```
structure cellule
{
        entier valeur;
        cellule suivante; // oh que non : on est en train de définir ce type.
};
```

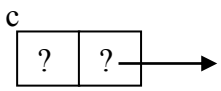
par contre, une adresse peut être stockée dans une structure.

représentation d'une cellule : une représentation simplifiée est adoptée pour les cellules

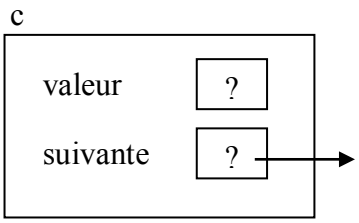
soit la définition de variable suivante :

```
cellule c;
```

la représentation sera alors la suivante :

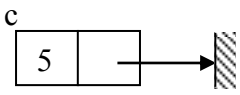


au lieu de



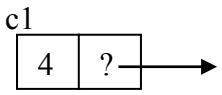
Pour initialiser la cellule c, utilisons l'opérateur .

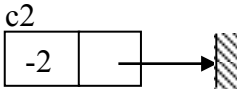
```
c.valeur ← 4; // une valeur arbitraire
c.suivante ← NULL; // pour initialiser le champ suivante qui est un pointeur
```



Comment réunir deux cellules dans une liste ?
Soient c1 et c2 deux cellules. c1 stocke la valeur 4, c2 la valeur –2. On souhaite faire en sorte que ces deux cellules se suivent, qu'elles soient **chaînées**.

```
cellule c1, c2;

c1.valeur ← 4;
c2.valeur ← -2;
```



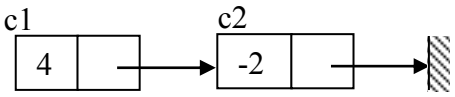


```
c2.suivante←NULL;
```

et maintenant ???

le champs suivante de c1 doit stocker l'adresse de c2. C'est juste ceci qu'il faut écrire en langage algorithmique !

```
c1.suivante←&c2;
```



Et nous obtenons ainsi une liste chaînée comportant 2 cellules.

L'opération consistant à ajouter une cellule se nomme : chaînage.

Tête de liste chaînée

Le principe d'un tableau est de stocker une seule information donnant accès à plusieurs valeurs : un tableau est une adresse à laquelle sont situées les différentes valeurs. Une autre information, implicite, est que les valeurs sont placées successivement en mémoire.

Pour la liste chaînée, on souhaite procéder de même : dans ce cas, l'unique information à garder pour accéder à tous les éléments de la liste est l'adresse de la première cellule (donc un pointeur vers une structure cellule). A partir de cette adresse, on accède à la première cellule (par une simple prise de contenu), puis on peut accéder à la deuxième, à la troisième, et ainsi de suite jusqu'à la dernière cellule de la liste chaînée. Par convention, l'adresse de la première cellule est stockée dans un pointeur dont le nom est très souvent `tete`.

```
fonction principale()
{
      cellule *tete;    // tete est un pointeur vers la première cellule de
                        // la liste

      tete ← NULL;      // pour indiquer que la liste est vide

      retourner;
}
```

construire une liste chaînée au fur et à mesure.

Qu'un tableau soit statique ou dynamique, il a une taille définie par un entier (sa taille utile), et cela peut poser problème lors de l'ajout d'éléments dans le tableau. Si le tableau est plein, il faut créer un nouveau tableau avec une taille augmentée de 1, recopier tous les éléments de l'ancien tableau dans le nouveau, ajouter le nouvel élément, puis détruire l'ancien tableau. La liste chaînée présente un avantage net pour cette opération d'ajout d'élément : il n'est pas nécessaire de créer une nouvelle liste, mais simplement de créer une nouvelle cellule et l'ajouter à la liste.

Pour réaliser ceci, il nous faut tout d'abord pourvoir créer des cellules lorsqu'on en a besoin, puis les ajouter à la liste. Nous délèguerons donc cette création de cellule à une fonction. En entrée, cette fonction nécessitera juste un paramètre permettant d'initialiser le champ valeur de

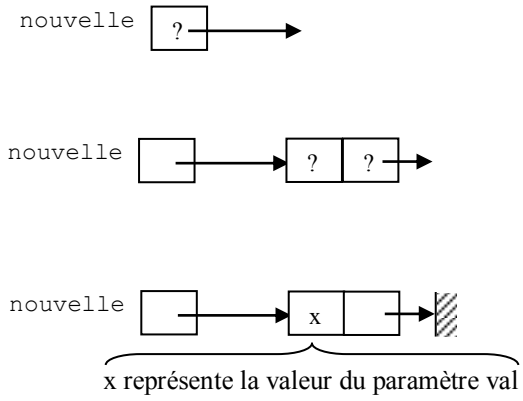
la cellule. Puisque nous illustrons le principe des listes chaînées en utilisant des entiers, la fonction de création de cellule aura un seul paramètre de type entier.
En ce qui concerne le type de la sortie de cette fonction, rappelons que son but est d'obtenir une nouvelle cellule : le type de retour semble donc être : `cellule`. Cependant, puisqu'une cellule est une structure, le type de retour est `cellule *` : la fonction retournera une adresse. Cela est également beaucoup plus souple qu'un type de retour `cellule`. Essayez donc de construire des listes chaînées en employant une fonction dont l'entête est la suivante :

```
fonction cree_cellule(entree : entier val → sortie : cellule)
```

Vous risquez de devoir définir autant de variables que de cellules dans le programme principal !

Voici donc la fonction à utiliser pour créer une cellule de liste chaînée :

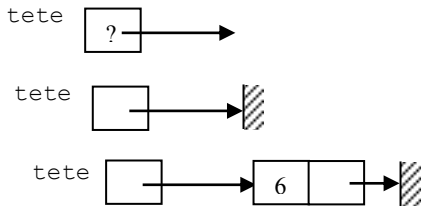
```
fonction cree_cell(entree : entier val → sortie : cellule *)
{
      cellule *nouvelle;

      // une reservation s'impose

      nouvelle ← reservation(1 cellule);

      // initialisation des champs

      nouvelle->valeur ← val;
      nouvelel->suivante ← NULL;

      retourner nouvelle;
}
```



Dans la partie droite de la page figure le schéma indiquant l'évolution de la variable nouvelle et de son contenu.

Comment utiliser cette fonction pour construire une liste ? la première étape consiste à partir d'une liste vide et à y ajouter une première cellule. A titre d'illustration, nous considèrerons que la valeur à stocker dans cette première cellule est 6.

```
fonction principale()
{
      // définition de la liste chaînée par l'adresse de la première
      // cellule

      cellule *tete;

      tete ←NULL;

      tete ← cree_cell(6);

      retourner;
}
```



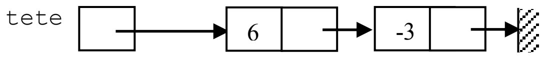
passons maintenant à l'étape suivante : ajouter à la liste une nouvelle cellule, dont la valeur est –3 par exemple.

Pour ce faire, il ne suffit pas d'écrire `tete ← cree_cell(-3);` car dans ce cas, l'ancienne valeur de tete (c'est à dire l'adresse de la cellule stockant la valeur 6) serait écrasée.

Il est tentant d'écrire la chose suivante :

```
tete->suivante ← cree_cell(-3);
```

Ce qui donnerait :



En effet, `tete->suivante` s'écrit également `(*tete).suivante`, c'est donc le champ `suivante` du contenu de `tete`, qui est de type `cellule`. Ce champ suivante est bien de type `cellule *`.

Jusqu'ici, tout se passe bien, mais que se passerait-il si l'on souhaitait ajouter une troisième cellule à la liste (avec la valeur 0 par exemple ?).

Suivant le même principe, on devrait écrire :

```
tete->suivante->suivante ← cree_cell(0);
```

(A titre d'exercice, quel est le type de `tete->suivante->suivante` ?)

Le constat est donc le suivant : pour ajouter une cellule directement dans une liste chaînée, l'instruction à écrire dépend de la position de cette cellule. Imaginez donc ce qu'il faudrait écrire pour ajouter une cellule en 100ème position d'une liste !

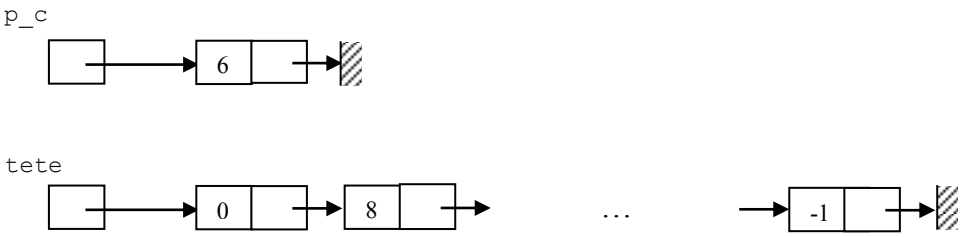
Il faudrait écrire…ceci !

```
tete->suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante->suivante->suivante->suivante->suivante->suivante-
>suivante->suivante ← cree_cell(12);
```

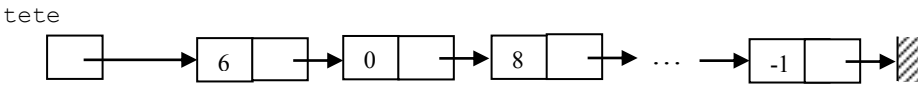
Intéressant, n'est-ce-pas ?

Ajout en tête de liste :

Il est par contre possible, contrairement à un tableau , d'ajouter simplement une cellule avant toutes les autres. Dans ce cas, l'opération à faire pour l'ajout reste toujours la même. Soit `p_c` un pointeur vers la cellule à ajouter, et `tete` le pointeur vers la liste à laquelle on doit ajouter la nouvelle cellule.



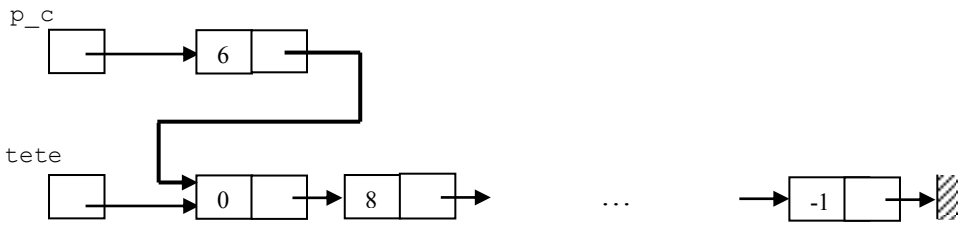
L'état final à obtenir est celui-ci :



Quelles instructions vont nous permettre d'obtenir cet état ?

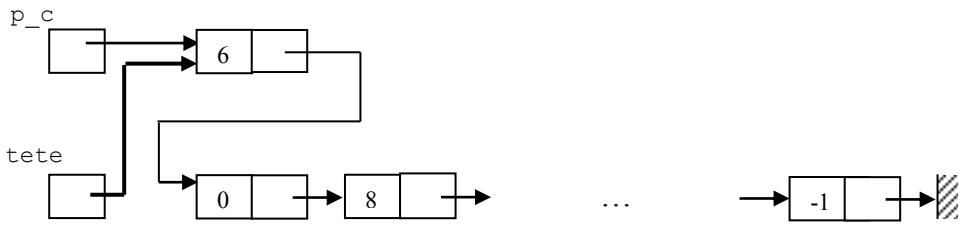
1ère étape : chaîner la tete de liste derrière la cellule à ajouter :

```
p_c->suivante ← tete; // égalité des pointeurs : une adresse est recopiée
```



2ème étape : remettre la tete de liste sur la première cellule

```
tete ← p_c;
```



Et voilà !

L'ordre de ces deux opérations est primordial : si les deux instructions sont interverties, que se passe-t-il ?

Si l'on souhaite ajouter une nouvelle cellule avec la valeur 123, il suffit d'écrire :

```
p_c ← cree_cell(123); // on peut réutiliser le pointeur p_c puisque son
                      // ancienne valeur a été dupliquée dans le pointeur tete

// de nouveau un chainage

p_c->suivante ←tete;
tete ← p_c;
```

Ce type de chaînage, où la nouvelle cellule a été mise avant celles déjà présentes dans la liste, s'appelle **chaînage en tête de liste**.

Ce sont les même instructions qui ont été utilisées ! Il est donc utile d'en faire…une fonction.

Cette fonction doit évidemment prendre en entrée les deux pointeurs représentant respectivement l'adresse de la cellule à chaîner en tête, et celle de la première cellule de la liste. En sortie, cette fonction fournira la nouvelle valeur de la tête de liste, car cette tete de liste est modifiée par le chaînage.

Ainsi, cette fonction s'écrit :

```
fonction chaine_tete(entree : cellule *tete, cellule *nouv ← sortie :
cellule *)
{
      nouv->suivante ← tete;
      tete ← nouv;

      retourner tete;
}
```

voire même :

```
fonction chaine_tete(entree : cellule *tete, cellule *nouv ← sortie :
cellule *)
{
      nouv->suivante ← tete;

      retourner nouv;
}
```

comment faut-il appeler cette fonction ?

dans un programme dans lequel on souhaite chaîner en tête de liste, cette fonction doit être appelée de la manière suivante :

```
fonction principale()
{
      cellule *tete; // définition de la tête de liste

      // initialisation de la liste

      // ajout d'une cellule en tete, avec la valeur 2 par exemple
```

```
      tete ← chaine_tete(tete,cree_cell(2));

      retourner;
}
```

regardons de plus près cet appel :

les arguments sont :

- `tete`, l'adresse de la cellule en tête de la liste
- `cree_cell(2)`, qui est un appel à la fonction de création de cellule, retournant un pointeur vers une nouvelle cellule, ce qui est le type attendu par la fonction de chaînage.

La valeur renvoyée par la fonction est bien stockée dans `tete` pour que la modification soit prise en compte.

Appeler une fonction pour obtenir la valeur d'un argument n'est pas toujours aisé à comprendre, voici donc une deuxième version du même programme utilisant un pointeur supplémentaire :

```
fonction principale()
{
      cellule *tete; // définition de la tête de liste
      cellule *dev;  // pointeur vers la cellule à ajouter

      // initialisation de la liste

      // ajout d'une cellule en tete, avec la valeur 2 par exemple

      dev ← cree_cell(2);

      tete ← chaine_tete(tete,dev);

      retourner;
}
```

Parcours de liste : ne perdons pas la tête !

Maintenant que nous savons construire une liste chaînée, intéressons-nous à des traitement classiques. Celui qui vient immédiatement à l'esprit est l'affichage des valeurs stockées dans une liste.

Au vu de l'organisation de la liste, la méthode est simple :

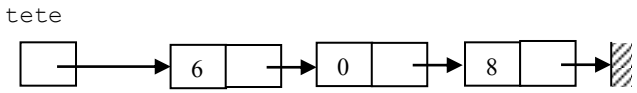
Si il y a une cellule alors
{
      afficher la valeur de cette cellule;
      passer à la cellule suivante;
}

que nous pouvons traduire par :

```
tant que  (tete ≠ NULL) alors
{
        afficher(tete->valeur);
        tete ← tete->suivante;
}
```

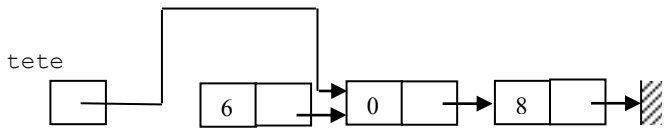
et cela fonctionne sans problème. Mais une seule fois ! en effet, si l'on modifie la valeur du pointeur tete, l'accès à la première cellule de la liste est perdu, et il est impossible de passer d'un cellule à la précédente, en quelque sorte de faire marche arrière !

illustration avec la liste suivante (chaque étape est matérialisée)



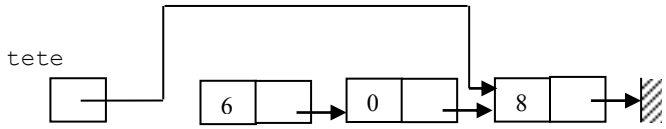
`tete` est différent de NULL, affichage de `tete->valeur` (6)

`tete ← tete->suivante;`



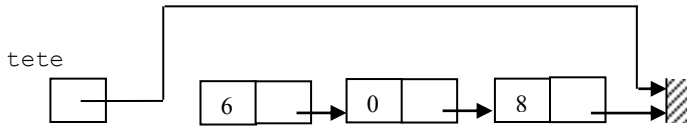
`tete` est toujours différent de NULL, affichage de `tete->valeur` (0)

`tete ← tete->suivante;`



`tete` est toujours différent de NULL, affichage de `tete->valeur` (8)

`tete ← tete->suivante;`



`tete` vaut NULL, la boucle s'arrête, les valeurs 6, 0 et 8 ont bien été affichées, les cellules se trouvent toujours en mémoire mais sont inaccessibles !

Pour pallier à cela, il existe deux solutions pour ne pas perdre la tête :

Utiliser un autre pointeur pour effectuer le parcours de la liste, par exemple un pointeur nommé cour que l'on utilise ainsi :

```
cellule *cour;

cour ← tete; // recopie de la valeur de tete dans cour

tant que  (cour ≠ NULL) alors
{
      afficher(cour->valeur);
      cour ← cour->suivante;
}
```

seconde solution : utiliser une fonction pour effectuer l'affichage : cette fonction aura en entrée un pointeur vers la tete de liste, mais ce paramètre sera une variable locale, copie de la tete de la fonction appelante : on peut donc le modifier à loisir dans la fonction

```
fonction  affiche_liste(entree : cellule *liste)
{
      tant que (liste ≠ NULL)
      {
            afficher(liste->valeur);
            liste ←liste ->suivante;
      }

      retouner;
}
```

l'appel depuis le programme principal se fait alors de cette manière :

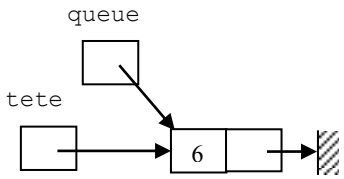
```
fonction principale()
{
      cellule *tete;

      // initialisation de la liste, ajout de cellules

      affiche_liste(tete);    // pas de souci, on transmettra la valeur
                              // numérique de tete
                              // tete ne sera pas modifiée
}
```

chaînage en queue

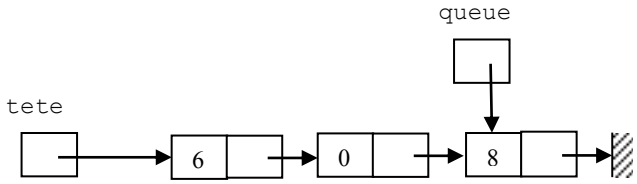
Il est également possible de réaliser un chaînage en fin ou queue de liste, lorsque l'on désire par exemple que les éléments soient rangés dans un certain ordre.
La méthode la plus basique pour réaliser ce chaînage ne fait qu'utiliser des algorithmes que nous venons de rencontrer : parcourir la liste depuis le début, puis ajouter un élément. Cependant, parcourir une liste prend du temps ! A l'inverse du tableau, on ne connaît dans une liste que l'adresse du premier élément et l'on est obligé de la parcourir depuis le début à chaque traitement.

Il devient alors avantageux de disposer d'un deuxième pointeur pour se repérer dans la liste, un pointeur désignant la dernière cellule de la liste. Lorsque la liste est vide ou ne contient qu'une seule cellule, les pointeurs tête et queue ont la même valeur.

Illustration, avec une liste à une cellule :

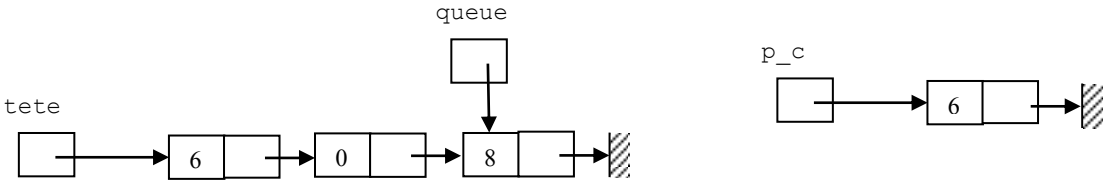


Avec une liste à plus d'une cellule :

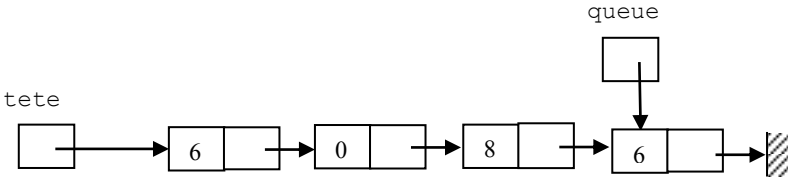


Une insertion en queue d'une cellule se fait donc assez simplement, à partir d'un pointeur `p_c` sur la cellule à ajouter.

On souhaite passer de :



à :



Cela s'obtient par les instructions suivantes :

```
queue->suivante ← p_c;
queue ← p_c;
```

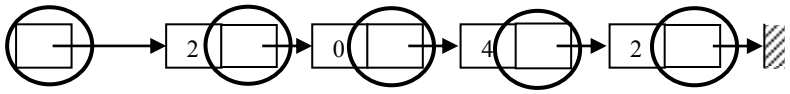
Une fois de plus, que se passe-t-il si ces deux instructions sont interverties ?

Le pointeur de queue n'est utile que pour le chaînage en queue de liste, car il ne permet que d'accéder à la dernière cellule de la liste.

Récursivité et liste chaînées

Une liste chaînée est une organisation récursive qui se prête volontiers à une utilisation par des fonctions récursives. En effet, une liste est donnée par un pointeur vers la cellule située en tête. Or cette cellule stocke un pointeur vers la cellule suivante. On peut légitimement considérer que ce pointeur est la tête d'une liste chainée.

Une liste chaînée est donc un pointeur vers une cellule stockant une valeur et une liste chaînée.



Sur ce schéma, il y a en tout 5 listes chaînées, repérées par les ellipses en gras

Un exemple de traitement récursif d'une liste chaînée : l'affichage des valeurs stockées dans la liste.

Nous avons rencontré un algorithme itératif effectuant cette tâche à base d'une boucle tant que.

La définition récursive de cet affichage est :

Si la liste est vide, l'affichage est terminé : *condition d'arrêt*

Sinon, afficher la valeur de la tête de liste et afficher la liste commençant par la cellule pointée par `tete->suivante`. En effet, la liste pointée par `tete->suivante` est la liste privée de l'élément pointé par `tete`. : *partie récursive*

```
fonction affich_liste(entree : cellule *liste)
{
      si (liste ≠ NULL) alors
      {
            afficher(liste->valeur);
            affich_list(liste->suivante);
      }
      // sinon rien à faire !

      retourner;
```

```
}
```

Autres traitements :

Compter les éléments d'une liste chaînée

Nous exposerons deux méthodes permettant de réaliser ce décompte : une méthode itérative et une méthode récursive.

Version itérative avec une boucle tant que :

```
fonction compte(entree : cellule *liste → sortie : entier)
{
	entier nb;

	nb ← 0;

	tant que (liste ≠ NULL)
	{
		nb ← nb+1;
		liste ← liste->suivante;
	}

	retourner nb;
}
```

version récursive :

```
fonction compte_rec(entree: cellule *liste → sortie : entier)
{
	entier nb;

	si (liste = NULL) alors
	{
		nb ← 0;
	}
	sinon
	{
		nb ← 1+compte_rec(liste->suivante);
	}

	retourner nb;
}
```

Détruire une liste chaînée

Il est possible de "détruire" une liste chaînée dans la mesure où chacune des cellules est obtenue par une réservation de mémoire. L'algorithme pour détruire une liste chaînée peut être expliqué de manière itérative ou récursive, nous allons traiter les deux cas. Dans les deux cas, il faudra faire attention au fait qu'une cellule stocke entre autres l'adresse de la cellule suivante. Cela signifie que lorsque l'on libère une cellule, l'adresse de la cellule suivante sera

effacée : une précaution s'impose donc : sauvegarder l'adresse de la cellule suivante avant de libérer la cellule concernée.

Algorithme itératif :

Tant qu'il reste au moins une cellule dans la liste
{
sauvegarder l'adresse de la cellule suivante;
libérer la cellule concernée;
passer à la cellule suivante;
}

qui se traduit par (`tete` est le pointeur vers la première cellule, comme d'habitude):

```
cellule *sauv; // pour sauvegarder l'adresse de la suivante

tant que (tete ≠ NULL)
{
      sauv ← tete->suivante; // sauvegarde de l'adresse de la prochaine
                             // cellule
      liberer(tete);    // un p'tit coup de balai dans la mémoire
      tete ← sauv;      // tete repointe sur la cellule suivante
                        // de la liste
}

// en sortie de cette boucle, tete vaut NULL
```

Dans ce cas, le parcours de la liste en modifiant la valeur de `tete` ne pose plus de problème, car les cellules sont détruites les unes à la suite des autres. La valeur de `tete` à la fin de cet algorithme est NULL.

Version récursive de cet algorithme :

Si `tete` vaut NULL, on ne fait rien : condition d'arrêt
Sinon, détruire la liste pointée par `tete` revient à détruire la liste pointée par la cellule suivante puis libérer `tete` (l'ordre des deux actions est primordial pour tenir compte du fait que l'on doit sauvegarder l'adresse de la cellule suivante).

```
fonction detruit_liste(entree : cellule *tete)
{
      si (tete ≠ NULL) alors
      {
            detruit_liste(tete->suivante);
            liberer(tete);
}

retourner;
}
```

Insérer un élément dans une liste chaînée : à faire en exercice

Supprimer un élément dans une liste chaînée : à faire en exercice

Trier une liste chaînée :

Le tri d'une liste chaînée, contrairement aux opérations d'insertion ou de suppression, est assez simple et a la même complexité (efficacité) que le tri d'un tableau, car aucune cellule n'est ajoutée ou retirée de la liste, il suffit juste dans ce cas d'intervertir les valeurs de deux cellules consécutives. Par contre, une donnée manquante est le nombre de cellules de la liste, qui est utile pour effectuer par exemple le tri à bulles. Pour parer à cette information manquante, l'algorithme utilisé est légèrement modifié :

Tant que la liste n'est pas triée
{
parcourir toute la liste : si deux cellules consécutives stockent des valeurs qui ne sont pas dans le bon ordre de tri, échanger les valeurs de ces cellules;
}

une précaution supplémentaire s'impose ici : puisque la comparaison porte sur deux cellules consécutives, les deux cellules doivent exister.
Voici l'algorithme de tri d'une liste par ordre croissant :

```
fonction trie_liste(entree : cellule *tete)
{
      entier est_triee, temp;
      cellule *courante;

      faire
      {
            est_triee ← 1;
            courante ← tete;

            // parcours de la liste

            tant que ((courante ≠ NULL) ET (courante->suivante ≠ NULL))
            {
                  si((courante->valeur > courante->suivante->valeur) alors
                  {
                        temp ← courante->valeur;
                        courante->valeur ← courante->suivante->valeur;
                        courante>suivante->valeur ←temp;
                        est_triee ← 0;
                  }

                  courante ← courante->suivante; // cellule suivante
            }

      } tant que (est_triee = 0);

      retourner;
```

```
}
```

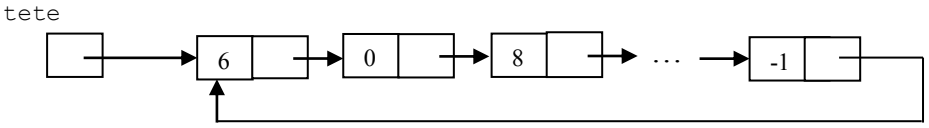
La structure de double boucle du tri à bulles est ici modifiée : plutôt que deux boucles pour, ce sont des boucles tant que et faire…tant que qui sont utilisées, car la taille de la liste n'est pas connue au moment de l'écriture de la boucle.

D'autres types de listes chaînées :

Les listes chaînées circulaires.

Une liste chaînée circulaire est une liste qui n'a pas de fin : le pointeur de la dernière cellule de la liste, plutôt que d'avoir la valeur NULL, repointe sur la tete de liste. Cette organisation particulière est parfois utile pour des aspects relatifs à l'ordonnancement des processus dans un système d'exploitation, qui est un mécanisme sans fin.

La structure d'une telle liste est alors la suivante :



Dans ce cas, les algorithmes se basant sur la détection de fin de liste ou de dernière cellule ne fonctionnent plus, car plus aucune cellule de la liste n'a de champ `suivante` égal à NULL.

La dernière cellule de la liste est maintenant caractérisée par le fait que la suivante a pour adresse la valeur stockée dans `tete`, le pointeur de début de liste.
Ainsi le test `courante ≠ NULL` devient `courante ≠ tete` (attention dans ce cas à bien pouvoir démarrer l'algorithme qui commence souvent par `courante ← tete` !) et les test `courante->suivante≠ NULL` devient : `courante->suivante ≠ tete`.

Un exemple : la distribution d'un paquet de cartes en temps constant.

Voici un exemple simple d'application des listes chaînées circulaires : comment effectuer de manière efficace la distribution de cartes d'un paquet de 52 cartes.
La première idée qui vient à l'esprit pour stocker un paquet de cartes est d'utiliser un tableau de structures nommée carte qui pourrait être défini de la manière suivante :

```
structure carte
{
        entier couleur; // avec un code
        entier hauteur; // avec un code
};
```

Le souci se pose pour la distribution de cartes de faire en sorte que chaque carte ne soit distribuée qu'une seule fois. Comment faire pour ne choisir dans le tableau que des cartes non encore distribués ? la solution la plus immédiate est d'associer à chaque carte un entier valant 0 ou 1 pour indiquer si cette carte fait encore partie du paquet ou non. On aurait ainsi, pour la distribution d'un paquet de cartes, le programme suivant, dont le but est de distribuer 5 cartes depuis un paquet de 52 cartes.

```
structure carte
{
	entier couleur;
	entier hauteur;
};

fonction principale()
{
	carte paquet[52];
	entier distri[52] ← {0};

	carte main[5];

	entier cpt, tirage;

	// initialisation des cartes
	// la division entière nous simplifie bien les choses…

	pour cpt de 0 à 51
	{
		paquet[cpt].couleur ← cpt/13;
		paquet[cpt].hauteur ← cpt%13;
	}

	// distribution : on tire un indice au hasard et on doit regarder
	// si la carte correspondante est toujours dans le paquet

	pour cpt de 0 à 4
	{
		faire
		{
			tirage ← hasard()%52;
		} tant que (distri[hasard] = 1);

		main[cpt] ← paquet[hasard]; // la carte est mise dans la main
		distri[hasard] ← 1; // on note qu'elle a été distribuée
	}

	// on peut afficher les cartes en main maintenant
}
```

Avec 5 cartes à distribuer, cette méthode reste rapide, mais d'autres exemples vont montrer le défaut de cette méthode : si l'on souhaite distribuer 40 cartes parmi les 52, le nombre de cartes disponibles dans le paquet va diminuer eu fur et à mesure, alors que les tirages au hasard ne tiendront pas compte de l'état (distribué ou non) de la carte; En moyenne, s'il reste N cartes disponibles parmi les 52 cartes initiales, il faudra $52/N$ tirages pour obtenir l'indice d'une carte restant disponible. En globalité, il faudra donc, pour distribuer P cartes depuis un paquet de 52 : $\sum_{N=1}^{N=P} \frac{52}{N}$. Pour tirer les 52 cartes, 240 tirages (en moyenne) seront nécessaires, donc 188 tirages redondants : le pourcentage de tirages utile est de : 21,7 %

Plus le nombre de cartes du paquet initial est élevé, plus cette valeur augmente.

Avec une liste chaînée circulaire, l'algorithme devient beaucoup plus simple :
Il suffit de construire une liste chaînée circulaire de 52 cartes, de tirer un nombre au hasard entre 0 et 100 (par exemple), de se déplacer dans la liste ce nombre de fois et de retirer la

carte atteinte : il s'agit d'une suppression, ce qui est beaucoup plus efficace avec une liste chaînée qu'avec un tableau. Il suffit de recommencer la même méthode avec la carte suivante, puisque dans la liste chaînée circulaire, il n'y a plus que les 51 cartes restant disponibles pour la distribution. Voici l'ensemble des fonctions nécessaires à réaliser ceci, mis à part la `fonction principale()` qu'il vous appartient d'écrire !

```
structure carte
{
      entier couleur;
      entier hauteur;
};

structure cellule
{
      carte c;
      cellule *suivante;
};

fonction cree_cellule(entree : carte *p_carte → sortie : cellule *)
{
      cellule *nouv;

      nouv ←reservation(1 cellule);

      nouv->c ← *p_carte;
      nouv->suivante ← NULL;

      retourner nouv;
}

fonction chaîne(entree : cellule *liste, cellule *prem→sortie : cellule *)
{
      prem->suivante ← liste;

      retourner prem;
}


fonction referme(entree : cellule *tete) // pour faire en sorte que la
                                         // liste soit circulaire
{
      cellule *cour;

      cour ← tete;

      si (cour ≠ NULL) alors
      {
            tant que (cour->suivante ≠NULL)
            {
                  cour ← cour->suivante;
            }

            cour->suivante ← tete;
      }

      retourner;
}
```

```
fonction distrib(entree : cellule *tete → sortie : cellule *)
{
      entier tirage, cpt;

      cellule *choisie;

      tirage ← hasard()%100;

      pour cpt de 0 à tirage
      {
            tete ← tete->suivante;
      }

      choisie ← tete->suivante;

      tete->suivante ← tete->suivante->suivante;

      retourner choisie;
}
```

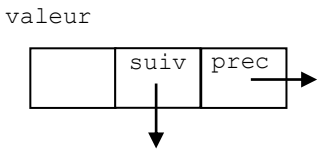
avant et arrière : les listes doublement chaînées.

Utilité : parcourir la liste dans les deux sens. Dans ce cas, chaque cellule doit permettre l'accès aux cellules précédentes et suivantes. La structure à utiliser est alors légèrement modifiée :

```
structure cellule
{
      entier valeur;
      cellule *suiv;
      cellule *prec;
};
```

Il faut dans ce cas être prudents sur les manipulations à faire,

En ce qui concerne la schématisation, une cellule de liste doublement chaînée ressemble à ceci :



Les traitements à appliquer aux listes doublement chaînées seront vus en exercice :

Chaînage en tête, chaînage en queue / insertion / suppression