CHAPITRE 2

PILES, FILES ET LISTES LINÉAIRES

Notre étude systématique de la spécification et de l'implantation de structures de données débute par les *structures linéaires*, qui sont parmi les plus utilisées.

Pour toutes les structures, nous adoptons le même style de présentation. D'abord, quand cela est justifié, nous donnons une brève présentation mathématique des objets concernés. Nous en faisons ensuite une spécification algébrique avec un noyau d'opérateurs de base.

Puis, nous décrivons les grands modes d'implantation de ces objets, leurs avantages et leurs inconvénients. Nous expliquons comment programmer et détaillons certains points intéressants. Nous ne pouvons pas aller au fond de toutes les implantations ou de leurs variantes, car cela serait lassant et sans grand intérêt.

Nous donnons dans les différents domaines et applications de l'informatique quelques indications sur l'utilisation des structures étudiées. Enfin, nous présentons des problèmes, des opérations et algorithmes élaborés qui les mettent en jeu.

Pour un problème donné, il n'y a en général pas de solution toute faite qu'il suffirait de recopier, en termes de structures de données. Mais il est bon de connaître certaines grandes classes de solutions qu'il faudra adapter. Ainsi, la *réutilisation* est un maître mot, et avec lui la modularité et l'abstraction.

L'usage des *piles*, *files* et *listes* est très fréquent en informatique. Les piles servent à gérer la traduction et l'exécution, notamment récursive, des programmes et sous-programmes. Les files sont utilisées pour modéliser les fichiers ou les tampons. Enfin, les listes sont la structure de base de certains langages fonctionnels ou logiques.

1 PILES

1.1 Spécification algébrique

Intuitivement, une pile est un *conteneur* d'objets que l'on peut manipuler avec trois opérations fondamentales : placer un objet sur une pile, ou *empiler*, accéder à l'objet qui est au *sommet* d'une pile, et retirer cet objet, ou *dépiler*. La gestion d'une pile est du type DAPS, c'est-à-dire *dernier arrivé-premier servi*, en anglais LIFO, ou *last in-first out*.

Nous prenons l'exemple classique des *piles génériques non bornées* d'objets de la sorte S, déclarée dans la théorie TRIV notée aussi ELEM.

spéc TRIV **étend** BASE
sorte S
fspéc

La spécification de base BASE contient les spécifications supposées *prédéfinies* des booléens, entiers naturels et relatifs, rationnels, chaînes, etc.

Une spécification algébrique PILE0(TRIV) des piles, paramétrée par TRIV où des objets génériques de type S sont définis, s'écrit :

spéc PILE0(TRIV) **étend** BASE
/* version simplifiée : **spéc** PILE **étend** TRIV
plutôt que de noter *et* le paramétrage par TRIV (préférable pour les langages fonctionnels) *et* l'extension du module BASE (préférable pour les langages objets), on note juste l'extension de TRIV qui étant lui même BASE */

sorte Pile /* piles non bornées d'objets de sorte S */
opérations

pilenouv : → Pile	/* constante pile vide */
empiler : Pile S → Pile	/* empilement d'un élément */
dépiler : Pile → Pile	/* dépilement du sommet */
remplacer : Pile S → Pile	/* remplacement du sommet */
sommet : Pile → S	/* élément au sommet */
vide : Pile → Bool	/* test de vacuité */
hauteur : Pile → Nat	/* nombre d'éléments */

préconditions p : Pile ; x : S
pré sommet(p) = ⌝ vide(p)
pré dépiler(p) = ⌝ vide(p)
pré remplacer(p, x) = ⌝ vide(p)

axiomes p : Pile ; x : S
(p1) vide(pilenouv) = vrai
(p2) hauteur(pilenouv) = 0
(p3) vide(empiler(p, x)) = faux
(p4) hauteur(empiler(p, x)) = hauteur(p) + 1
(p5) sommet(empiler(p, x)) = x
(p6) dépiler(empiler(p, x)) = p
(p7) remplacer(p, x) = empiler(dépiler(p), x)
fspéc
La spécification PILE0(TRIV) étend la spécification BASE. La signification et le comportement des opérateurs définis ci-dessus sont immédiats.

Comme nous l'avons fait pour les ensembles, il est facile de contraindre la sorte Pile pour obtenir des *piles bornées*. Il suffit de définir la borne supérieure comme étant strictement positive et bornée. Cette borne supérieure qui est un paramètre de la spécification sera ensuite utilisée dans une précondition sur la fonction empiler. L'impact de cette contrainte est à vérifier pour l'ensemble de la spécification.

spéc BSUP **étend** BASE /* pour le paramètre borne supérieure */
opération
  N : → Nat /* borne supérieure */
axiome
  (p0) 0 < N = vrai
fspéc

spéc PILE1(TRIV ; BSUP) **étend** BASE
...
pré empiler(p) = hauteur(p) < N
...

Pour obtenir une spécification effective des piles bornées (PILE1) ou non bornées (PILE0) de n'importe quelle sorte d'objets, il suffit de substituer au paramètre formel TRIV une spécification avec un paramètre effectif ayant les mêmes propriétés.

spéc PILENAT0 **étend** PILE0(TRIV → NAT **avec** S → Nat) **fspéc**
spéc PILENAT1 **étend** PILE1(TRIV → NAT **avec** S → Nat ;
  BSUP → NAT **avec** N → 100) **fspéc**

1.2 Implantation contiguë et programmation fonctionnelle

Pour une pile concrète contenant des objets du type concret `S`, nous pouvons prendre par exemple une structure contenant (Fig. 1) un *tableau* `v` de `N` éléments de type `S` et un *entier* h pour mémoriser la hauteur de la pile. `N` est supposé prédéfini.

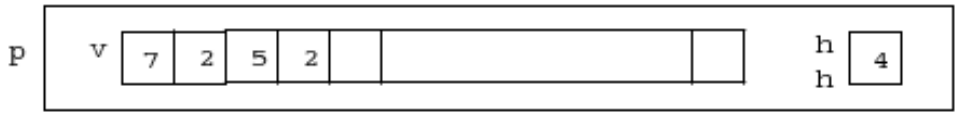


Figure 1 : Implantation contiguë d'une pile de 4 entiers (le sommet contient 2)

Le tableau v peut être avec allocation statique, déclaré `S v[N]`, ce qui ne convient que pour une pile bornée par N. Il peut aussi être avec allocation dynamique, déclaré `S *v`, ce qui convient pour une pile bornée ou non, à

condition de faire une allocation (en C, par `MALLOC`) au début de l'utilisation de la pile, et des réallocations à chaque fois que l'espace-mémoire est insuffisant pour empiler (en C, par `REALLOC`). Dans ce cas, le nouvel espace-mémoire peut être complètement distinct de l'ancien, ce qui oblige à des recopies. Ce procédé est très lourd et sera utilisé avec parcimonie dans le cadre d'une programmation avec mutation cf sections suivantes.

En nous limitant au **cas de l'allocation statique**, et avec `N = 100`, nous avons le type de piles :

```
   #define N 100
   typedef S Stab[N];
   typedef struct {S v[N]; Nat h;} Pile;

Pile pilenouv()
{
   Pile p;
   p.h = 0;
   return p;
}
```

Deux versions de l'opération `empiler` sont données en correction.

```
/* version 1 */
Pile empiler(Pile p, S x)*/
{
   Pile p1 = p;
   p1.v[p.h] = x;
   p1.h = p.h + 1;
   return p1;
}
```

```
/* version 2      */
Pile empiler(Pile p, S x)

{
      p.v[p.h] = x;
      p.h = p.h + 1;
      return p;
}
```

Lors de l'appel à la fonction version 1, la pile `p` est donc recopiée sur une zone temporaire, puis une nouvelle fois sur la pile `p1`. La version 2 est plus efficace puisqu'elle travaille directement sur la zone temporaire.
En effet, en C, les arguments d'une fonction sont passés *par valeur* ce qui signifie que lors de l'appel de cette fonction les paramètres sont recopiés dans une zone de mémoire temporaire qui est détruite en quittant la fonction.

```
Pile depiler(Pile p)
   {
      p.h = p.h - 1;

      return p;
   }
```

```
Pile remplacer(Pile p, S x)
{
   p.v[p.h - 1] = x;

   return p;
}
```

```
S sommet(Pile p){return p.v[p.h - 1];}
Bool vide(Pile p){return p.h == 0;}
Nat hauteur(Pile p){return p.h;}
```

1.3 Implantation contiguë et programmation avec mutation

Pour éviter les recopies multiples de structures, une implantation d'opérations avec mutation est intéressante. On privilégie le choix de cette implantation avec mutation lorsqu'il n'est pas nécessaire de garder l'ancienne version des objets manipulés. Pour la réaliser en C, il faut simuler le passage *par adresse*, qui n'a pas été prévu dans le langage, à l'aide d'un pointeur sur l'argument que l'on veut modifier par *effet de bord*.

Pour la structure de données des piles, on prévoie alors d'avance un pointeur sur la structure ce qui est préférable car ainsi, la structure de données ne conditionnera pas le profil des fonctions qui ne changent pas.

```
#include "base.h"
#define N 100
typedef struct strpile {S v[N]; Nat h;} Strpile, * Pile;

Pile pilenouv()/* mutation sans changement de prototype */
{
   Pile p = MALLOC(Strpile);
   p->h = 0;
   return p;
}
```

Ce type d'implantation est plus efficace puisque lors de l'appel d'une fonction, seule la case mémoire contenant l'adresse de la pile est recopiée.

```
Pile empiler(Pile p, S x)
{
   p->v[p->h] = x;
   p->h = p->h + 1;
   return p;
}
Pile depiler(Pile p)
{
   p->h = p->h - 1;

   return p;
}
```

```
Pile remplacer(Pile p, S x)
{
   p->v[p->h - 1] = x;

   return p;
}
```

1.4 Implantation contiguë et programmation avec mutation des piles non bornées

Pour des piles non bornées, des tableaux dynamiques peuvent être utilisés, avec ou sans passage par adresse de la structure `struct strpile`, mais dans tous les cas, le tableau contenant les éléments de la pile étant dynamique, la programmation sera avec mutation de la pile :

```
typedef struct strpile {S* v; Nat h; Nat taille;} Pile;
typedef struct strpile {S* v; Nat h; Nat taille;} Strpile, * Pile;
#define TAILLE 100
```

```
Pile pilenouv()
   {
      Pile p
      p.v = MALLOCN(S,TAILLE);
      p.h = 0;
      p.taille = TAILLE;
      return p;
   }
```

```
Pile pilenouv()
{
   Pile p = MALLOC(Strpile);
   p->v = MALLOCN(S,TAILLE);
   p->h = 0;
   p->taille = TAILLE;
   return p;
}
```

```
Pile empiler(Pile p, S x)
   {
      if (p->h == p->taille)
      {
            S* temp = p->v;
            p->v = REALLOC(p->v, S, p->taille + TAILLE);
            if (p->v == NULL)
            {
                  printf();   p->v = temp;
            }
            else
            {
                  p->taille = p->taille + TAILLE;
                  p->v[p->h] = x;
                  p->h = p->h + 1;
            }
      }
      else
      {
            p->v[p->h] = x;
            p->h = p->h + 1;
      }
      return p;
   }
```

Pour bien faire il vaudrait mieux refaire la spécification pour ensuite réallouer en dehors de la fonction empiler lorsque le tableau dynamique est plein.

1.5 Implantation par chaînage

Nous insistons davantage sur ce mode de représentation. Une pile bornée ou non bornée peut être implantée comme une *liste chaînée*, c'est-à-dire une suite de cellules de mémoire à deux places, la première contenant une valeur, la deuxième un pointeur vers la cellule suivante (Fig. 2). Une pile est alors par exemple représentée par un pointeur sur la première cellule, qui est celle du sommet. La dernière cellule contient l'adresse fictive `NULL`, notée aussi `null`.

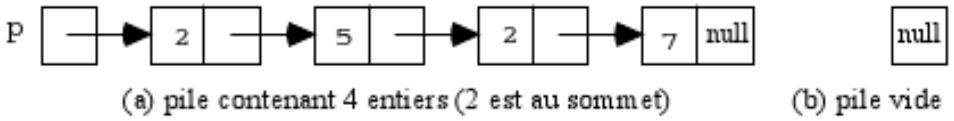


Figure 2 : Implantation chaînée de piles d'entiers

En C, les cellules sont des structures du type `struct strpile`, noté aussi `Strpile`, avec le champ `valeur` pour la valeur et `suivant` pour le pointeur sur le successeur : `[Attention à la confusion S et s à remplacer par suivant]`

```
typedef struct strpile {S valeur; struct strpile *suivant;}
                  Strpile, *Pile;


      Pile pilenouv(){return (Pile) NULL;}
```

Avec les piles, nous avons une chance exceptionnelle de pouvoir conserver un point de vue rigoureusement fonctionnel en partageant des cellules entre différentes piles. En acceptant un tel partage, nous programmons rapidement les différentes opérations, parfois avec plusieurs formes :

```
      Pile empiler(Pile p, S x)
      {
             Pile p1 = MALLOC(Strpile);
             p1->valeur = x;
             p1->suivant = p;
             return p1;
      }

      Pile depiler(Pile p)    /* version sans free … */
      {return p->suivant;}
```

Avec cette implantation chaînée des piles nous obtenons une programmation fonctionnelle grâce au partage de cellule, l'opération `remplacer` utilise `empiler` et `depiler` pour conserver l'aspect fonctionnel :

```
Pile remplacer(Pile p, S x)
{
      return empiler (depiler (p), x);
}

S sommet(Pile p)
{return p->valeur;}

Bool vide(Pile p)
{return p == (Pile) NULL;}

Nat hauteur(Pile p)     /* forme récursive */
{return vide(p) ? 0 : hauteur(p->suivant) + 1;}

Nat hauteur(Pile p)     /* forme itérative */
{
      Nat h;
      for(h = 0; !vide(p); h++, p = p->suivant);
      return h;
}
```

A noter. Dans l'écriture récursive de hauteur, nous utilisons une *expression conditionnelle* C, et dans la boucle for de la version itérative l'*opérateur virgule* «,» qui offre une remarquable concision.

A cause du partage de cellules de mémoire, cette programmation des piles est fonctionnelle. Du coup, pour une programmation avec mutation, les seules différences résident dans les fonctions `depiler`, qui doit effectivement désallouer la cellule de mémoire correspondant au sommet de la pile par utilisation de la procédure à *effet de bord* `FREE`, et `remplacer`, qui doit modifier la valeur au sommet.

```
Pile depiler(Pile p)    /* version avec free */
{
      Pile p1 = p->suivant;
      FREE(p);
      return p1;
}
```

```
Pile remplacer(Pile p, S x)
{
      p->valeur = x;
      return p;
}
```

Les autres opérations sont identiques à celles de la version fonctionnelle.

Une variante de la représentation ci-dessus consiste à munir la pile d'une cellule d'en-tête contenant avec le pointeur sur le sommet de pile et divers renseignements à maintenir. L'information trouvée couramment est le nombre d'éléments dans la pile, comme dans la représentation contiguë, où elle permettait d'accéder rapidement au sommet. Ici, l'enregistrement de la hauteur évite un recalcul quand elle doit être obtenue. La contrepartie est qu'il faut la mettre à jour lors des empilements et dépilements.

```
typedef struct strpile {S valeur; struct strpile *suivant;}
                  Strpile;
typedef struct strpile {Nat hauteur; Strpile *sommet;} Pile;
```

Un pointeur peut être placé sur cet en-tête mais l'économie est négligeable :

```
typedef struct strpile {Nat hauteur; Strpile *sommet;} *Pile;
```

1.5 Utilisation des piles

Les piles sont très utiles en informatique, notamment dans les techniques de gestion de mémoire liées à la compilation, à la gestion des blocs lors de l'exécution d'un programme et à la récursivité, dans les problèmes de parcours de structures comme arbres, graphes, etc. Les piles sont aussi câblées au niveau des processeurs physiques pour exécuter rapidement les appels de sous-programmes.

2 FILES

2.1 Spécification algébrique

Intuitivement, une file est un conteneur d'objets que l'on peut manipuler avec trois opérations fondamentales : *ajouter* un objet *en queue* de file, accéder à l'objet qui est *en tête* de la file, et *supprimer* l'objet *en tête* de la file. Sa gestion est donc du type PAPS, c'est-à-dire *premier arrivé-premier servi*, en anglais FIFO, ou *first in-first out*.

Nous prenons l'exemple des *files génériques non bornées* d'objets de la sorte S, apparaissant dans la théorie paramètre triviale TRIV. Une spécification algébrique de ces objets est :

spéc FILE0(TRIV) **étend** BASE
/* version simplifiée : **spéc** FILE **étend** TRIV */
/* version bornée : **spéc** FILE1(TRIV) **étend** BSUP */
sorte File /* files non bornées d'objets de sorte S */
opérations

filenouv : → File	/* constante file vide */
adjq : File S → File	/* adjonction en queue */
supt : File → File	/* suppression en tête */
tête : File → S	/* élément en tête*/
vide : File → Bool	/* test de vacuité */
lgr : File → Nat	/* longueur ou nombre d'éléments */

préconditions
pré tête(f) = ⌉ vide(f)
pré supt(f) = ⌉ vide(f)

axiomes f : File ; x : S
(f1) vide(filenouv) = vrai
(f2) lgr(filenouv) = 0
(f3) vide(adjq(f, x)) = faux
(f4) lgr(adjq(f, x)) = lgr(f) + 1
(f5) tête(adjq(f, x)) = **si** vide(f) **alors** x **sinon** tête(f) **fsi**
(f6) supt(adjq(f, x)) = **si** vide(f) **alors** filenouv
sinon adjq(supt(f), x) **fsi**
(f7) supt(adjq(filenouv, x)) = filenouv /* variante */
(f8) supt(adjq(adjq(f,y), x)) = adjq(supt(adjq(f,y)), x)
fspéc

Nous pourrions aussi ajouter une contrainte de taille pour avoir des files bornées, que l'on appelle aussi *tampons*, ou *buffers*.

2.2 Représentation par contiguïté

Une file peut être représentée (Fig. 3) par un *tableau* `v` de `N` éléments du type concrétisant `S`, et deux *entiers naturels*, `h` pour l'indice de tête, et `l` pour la longueur de la file. A la place de la longueur, il serait également possible de noter `q` l'indice de queue. L'entier `N` est supposé prédéfini. Le tableau peut être alloué statiquement, c'est-à-dire déclaré `S v[N]`, ce qui ne convient que pour une file bornée. Il peut être alloué dynamiquement, c'est-à-dire déclaré `S *v`, ce qui convient pour une file bornée ou non.

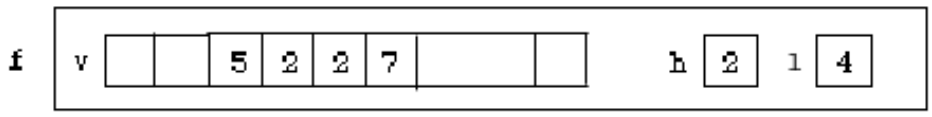


Figure 3. Implantation contiguë d'une file de 4 entiers (5 est en tête)

Avec une allocation statique, nous pouvons définir en C le type File :

```
#define N 100
typedef struct strfile {S v[N]; Nat h; Nat l;} File;
```

Avec une allocation dynamique, nous pourrions définir une variante avec la taille en mémoire du tableau dynamique :

```
typedef struct strfile {S*v; Nat h; Nat l; Nat taille;} File;
```

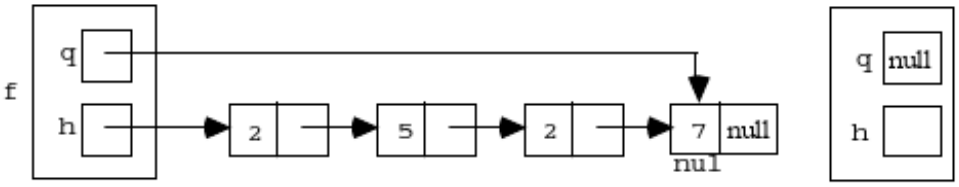
La programmation des opérations est du même ordre que celle des piles ou des ensembles. Notons toutefois qu'une suppression en tête *avec mutation* fait progresser `h` de `1`, ce qui a pour effet de rendre disponible une case de tableau pouvant être réutilisée lors de nouvelles adjonctions. Ainsi, une file peut être scindée en deux parties : le début de file occupant la fin du tableau, et la fin de file le début du tableau. Il faut donc entendre ici contiguïté des cellules *modulo* `N`. Pour des manipulations en queue de liste, notamment des adjonctions, nous devons récupérer l'indice `q` de l'élément en queue par :

```
q = (h + l - 1) % N;
```

où `%` est mis pour *modulo*, puisqu'en C les indices de `v` vont de `0` à `N - 1`. L'indice de la première case libre est lui donné par la valeur `(h + l) % N`. Enfin, la file vide est telle que `l == 0` et `h` quelconque mais pour que la première adjonction en queue se fasse à l'indice 0 du tableau, `h == 0`.

2.3 Représentation par chaînage et programmation avec mutation

Une file peut être composée (Fig. 4) de cellules de mémoire chaînées, comme les piles, avec un en-tête comportant deux pointeurs, `h` et `q`, sur la tête et la queue de la file.



(a) file de 4 entiers (b) file vide

Figure 4 : Implantation chaînée de files avec 2 pointeurs

D'où les déclarations de types en C :

```
typedef struct strfile {S v; struct strfile *s;} Strfile;
typedef struct sfile {Strfile *h; Strfile *q;} File;
```

Méthodologie:

1. dessin en couleur des actions pour chaque fonction
2. fonction C en numérotant instructions / actions
3. vérification du C par le dessin (par une tierce personne)

La file vide est caractérisée par un en-tête avec `h == q == NULL`.

```
File filenouv()
{
   File f;
   f.h = NULL;
   f.q = NULL;
   return f;
}
```

Cette représentation permet de réaliser facilement les opérations adjq et supt, l'une en queue et l'autre en tête.

```
File adjq(File f, S x)
{
   Strfile * f1 = MALLOC(Strfile);
   f1->v = x;
   f1->s = NULL;
   if (f.h == NULL) { f.h = f1; f.q = f1; }
   else { f.q->s = f1; f.q = f1; }
   return f;
}
```

```
File supt(File f)
{
   Strfile * f1 = f.h;
   if (f.h == f.q) {f.h = NULL; f.q = NULL; }
   else f.h = f.h->s;
   FREE(f1);
   return f;
}

S tete(File f){return f.h->v;}
Bool vide(File f){return f.h == NULL;}
```

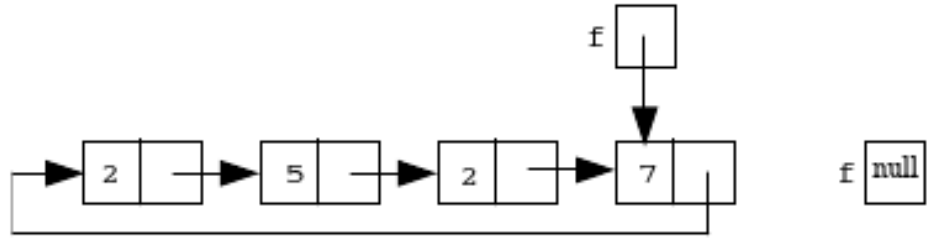
Compte tenu des différences de type entre `f` et `f.h` , il faut recourir à une fonction intermédiaire :

```
Nat lgrb(Strfile *f1, Strfile *f2)
{return f1 == f2 ? 1 : 1 + lgrb(f1->s, f2);}
Nat lgr(File f){return vide(f) ? 0 : lgrb(f.h, f.q);}
```

Mais nous avons mieux. Si, à la place de NULL, nous plaçons en queue l'adresse de la cellule de tête, nous n'avons plus besoin que du pointeur de queue, qui *est* la file `f` elle-même, et la seule déclaration est :

```
typedef struct strfile {S v; struct strfile *s;}
   Strfile, *File;

File filenouv(){return NULL;}
```



(a) file de 4 entiers (b) file vide

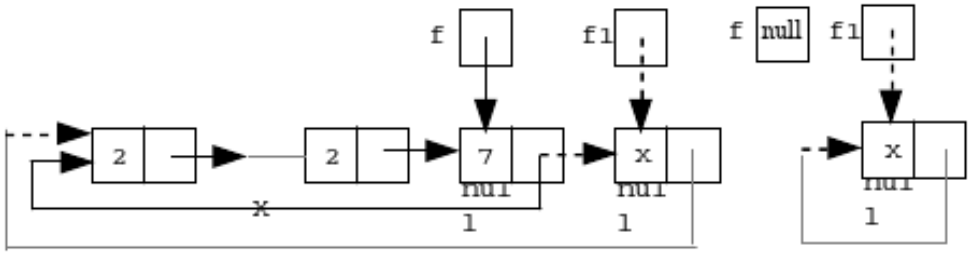
Figure 5 : Implantation chaînée d'une file avec un seul pointeur sur la queue

Comme dans le cas des piles, une variante consiste à munir une file d'un en-tête. Les opérations sur les files sont proches de celles sur les piles. Une des différences réside toutefois en l'absence de la particularité que nous avions signalée pour les piles. Pour les files en effet, si l'on veut garantir un point de vue fonctionnel pur, c'est-à-dire préserver l'intégrité des arguments files, il faut faire des copies des parties qui ne sont pas affectées pour constituer la nouvelle file. Programmons les opérations avec mutation. Pour l'opération

`adjq`, *avec mutation*, dans la représentation des files avec un seul pointeur sur la queue, nous distinguons les cas illustrés à la figure 6, où la file initiale est vide ou non.

A effectuer en 3 étapes:

1. dessin couleur avec numérotation des actions,
2. écriture en C en numérotant les instruction,
3. vérification du C sur le dessin.



(a) Cas de file non vide (b) Cas de file vide

Figure 6 : Adjonction de `x` en queue de `f` avec mutation
(Les ajouts de pointeurs sont en pointillés et la seule suppression marquée d'une croix.)

```
File adjq(File f, S x)
{
   File f1;
   f1 = MALLOC(Strfile);
   f1->v = x;
   if (vide(f)) f1->s = f1;
   else {f1->s = f->s; f->s = f1;}
   return f1;
}
File supt(File f)
{
   File retour;
   Strfile * f2;
   if (f == f->s) {FREE(f); retour = NULL; }
   else { f2 = f->s; f->s = f->s->s;  FREE(f2); retour = f; }
   return retour;
}
S tete(File f){return f->s->v;}
Bool vide(File f){return f == NULL;}

Nat lgrb(Strfile *f1, Strfile *f2)
{return f1 == f2 ? 1 : 1 + lgrb(f1->s, f2);}
Nat lgr(File f){return vide(f) ? 0 : lgrb(f->s, f);}
```

2.5 Utilisation

Elles servent aussi à modéliser des systèmes dynamiques pour des études quantitatives de comportement. A une file est associé un *serveur* dont les objets enregistrés dans la file sont les *clients*. Les clients se présentent dans la file avec une certaine fréquence. Le serveur sert le premier client dans la file pendant un certain temps et les autres clients attendent. Quand le premier client est servi, il sort de la file, et le serveur passe au suivant. Arrivée et service suivent des lois de probabilité plus ou moins complexes.

On tente de déterminer, en fonction de ces lois, le temps d'attente moyen des clients, le temps où le serveur est actif, la longueur moyenne de la file, etc. Dans les cas simples, il est possible de calculer exactement ces valeurs ; sinon, il faut procéder par *simulation*, en reproduisant dans un programme la vie de la file.

Dans des problèmes complexes, il faut faire intervenir des *réseaux* de files d'attente interconnectées, comme au projet 6.19. Enfin, la politique de gestion PAPS n'est pas toujours assez réaliste. Il peut exister des clients prioritaires, des phénomènes de lassitude qui font quitter une file sans être servi, ou de préemption qui obligent à interrompre un service au profit d'un client ultraprioritaire, des services avec *tourniquets*, c'est-à-dire en temps partagé du serveur entre les clients, etc. Il faut parfois des structures de données plus complexes que la simple file PAPS, pour gérer convenablement ces cas : listes générales, files avec priorités, arbres, etc.

3 LISTES LINÉAIRES

3.1 Spécification algébrique

Définition 4.1 Une *liste linéaire* sur un ensemble E est une suite finie x_1, x_2, ..., x_k d'éléments de E. Elle est souvent notée $(x_1, x_2, ..., x_k)$.

Mais cette définition ne dit rien des opérations que peuvent subir les listes. Au moins au début, celle des *listes génériques non bornées* peut calquer la spécification précédente des piles ou des files. En variante, il est facile de définir des *listes bornées* avec des contraintes de taille comme pour les piles bornées.

Mettre au clair le fait que dans adjt(l, x), l est une variable muette. Par exemple, la précondition sur supt(l) s'applique sur une liste qui peut ensuite être remplacée par adjt(l, x). La notation l= adjt(l', x) permettrait d'éclaircir la présentation ...

spéc LISTE0(TRIV) **étend** BASE
/* version simplifiée : **spéc** LISTE **étend** TRIV */
sorte Liste /* listes d'objets de la sorte S */
opérations
    listenouv : → Liste /* liste vide, notée aussi Λ */
    adjt : Liste S → Liste /* adjonction en tête*/
    supt : Liste → Liste /* suppression en tête */
    tête : Liste → S /* élément en tête */
    vide : Liste → Bool /* test de vacuité */
    lgr : Liste → Nat /* nombre d'éléments */
préconditions l : Liste ; x : S
    pré tête(l) = ⌝ vide(l)
    pré supt(l) = ⌝ vide(l)
/* l'axiome supt(listenouv) = listenouv peut remplacer cette précond. */
axiomes l : Liste ; x : S
    (l1) vide(listenouv) = vrai
    (l2) lgr(listenouv) = 0
    (l3) vide(adjt(l, x)) = faux
    (l4) lgr(adjt(l, x)) = lgr(l) + 1
    (l5) tête(adjt(l, x)) = x
    (l6) supt(adjt(l, x)) = l
fspéc

A l'aide de ces opérateurs, beaucoup d'autres peuvent être définis. Mais pour la sorte S en paramètres, nous avons besoin d'une théorie plus riche en opérations et propriétés. Si une relation binaire notée ==, interprétée comme une égalité, est demandée sur S, alors nous prenons la théorie EG qui étend la théorie dite triviale appelée TRIV, qui elle étend les booléens et les entiers.

spéc TRIV
étend BOOL
sorte S
fspéc

spéc EG **étend** TRIV
opérations
    _==_ , _≠_ : S S → Bool /* égalité, différence dans S */
axiomes x, y, z : S
    (e1) x == x = vrai /* réflexivité de == */
    (e2) x ≠ y = ⌝ (x == y) /* définition de ≠ */
fspéc

Voici quelques-unes des opérations utilisant l'égalité dans l'ébauche de spécification LISTE1 suivante.

spéc LISTE1(EG) **étend** LISTE0(TRIV → EG) /* utilise == pour S */
/* rappel: *étend* est plutôt pour les langages objet
et *paramétrage* plutôt pour les langages fonctionnels */
/* version simplifiée : **spéc** LISTE **étend** EG */

app : liste S → Bool /* appartenance d'un élément */
conc : Liste Liste → liste /* concaténation */
axiomes l : Liste ; x : S ; i : Nat
app(listenouv, x) = faux
app(adjt(l, y), x) = x == y ˅ app(l, x)
conc(listenouv, l2) = l2
conc(adjt(l1, x), l2) = adjt(conc(l1, l2), x)
ou bien
conc(l, listenouv) = l
conc(l, adjt(l1, x)) = conc(adjq(l, x), l1)

/* opérations sur le dernier élément */
queue : Liste → S /* dernier élément */
adjq : Liste S → Liste /* adjonction en queue */
supq : Liste → Liste /* suppression en queue */

/* opérations sur la première occurrence d'un élément */
sup1 : Liste S → Liste/* suppression 1ère occ. d'un élt */
chg1 : Liste S S → Liste /* remplacement 1ère occ. d'un élt */
rech1 : Liste S → Nat /* plus petit rang d'un élément */
rech2 : Liste S → Liste /* liste commençant à l'élément */

/* opérations sur le ième élément */
_[_] : Liste Nat → S /* ième élément, noté aussi él(l,i) */
ieme : Liste Nat → Liste /* sous-liste débutant au ième élt */
/* sans préconditions */
ins : Liste Nat S → Liste /* insertion en ième position */
sup : Liste Nat → Liste /* suppression du ième élément */
chg : Liste Nat S → Liste /* remplacement du ième élément */
lig : Liste Nat → Liste /* sous-liste terminant au ième élt */
lid : Liste Nat → Liste /* sous-liste débutant au ième élt */
/* sans le ième élément */
...
/* opérations sur le dernier élément */
préconditions l : Liste ;
pré queue(l) = ℸ vide(l)
pré supq(l) = ℸ vide(l)
/* l'axiome supq(listenouv) = listenouv peut remplacer cette précond. */

axiomes l : Liste ; x : S

queue(adjt (l, x)) = **si** vide (l) **alors** x **sinon** queue(l) **fsi**

ou bien spécifier «directement»

queue(l) = l[lgr (l)]

adjq(listenouv, x) = adjt(listenouv, x)

adjq(adjt (l, y), x) = adjt(adjq(l, x), y)

supq(adjt (l, x)) = **si** vide (l) **alors** listenouv
sinon adjt(supq(l),x) **fsi**

cet axiome peut être remplacé par 2 axiomes séparant les 2 cas

supq(adjt(listenouv, x)) = listenouv

supq(adjt(adjt(l, x), y)) = adjt(supq(adjt(l, x)), y)

Une variante pour définir adjq et supq est d'utiliser des opérateurs définis auparavant et non les générateurs de base pour une spécification «directe», pour retrouver les axiomes précédents, remplacer l par l' et poser l'= adjt(l,y)

adjq(l, x) = **si** vide(l) **alors** adjt(listenouv, x)
sinon adjt(adjq(supt(l), x), tête(l)) **fsi**

supq(l) = **si** lgr(l) == 1 **alors** listenouv
sinon adjt(supq(supt(l)), tête(l)) **fsi**

Une programmation à l'identique conduirait à des exécutions d'une extrême inefficacité, où l'on passerait son temps à parcourir, à faire, à défaire et à refaire des listes inutilement. Rappelons que nous sommes en train de *spécifier* et non d'implanter. Dans cette deuxième phase, il faudra être économe en temps et en espace. Cette question sera rediscutée.

/* opérations sur la première occurrence d'un élément */

sup1(listenouv, x) = listenouv

sup1(adjt(l, x), y) = **si** (x == y) **alors** l **sinon** adjt(sup1(l, y), x) **fsi**

chg1(listenouv, x, y) = listenouv

chg1(adjt(l, x), y, z) = **si** (x == y) **alors** adjt(l, z)
sinon adjt(chg1(l, y, z), x) **fsi**

rech1(listenouv, x) = 0

rech1(adjt(l, x), y) = si (x == y) alors 1 sinon
si rech1(l, y)==0 alors 0 sinon 1+ rech1(l, y) fsi fsi

rech2(listenouv, x) = listenouv

rech2(adjt(l, x), y) = si (x == y) alors adjt(l, x) sinon rech2(l, y) fsi

/* opérations sur le $i^{ème}$ élément */

préconditions l : Liste ; x, y : S ; i : Nat

pré l[i] = $1 \leq i \leq$ lgr(l)

pré ieme(l, i) = $1 \leq i$

axiomes l : Liste ; x, y : S ; i : Nat

```
        adjt(l, x)[i] = si i == 1 alors x sinon l[i-1] fsi
        ieme(listenouv, i) = listenouv
        ieme(adjt(l, x), i) = si i == 1 alors adjt(l, x) sinon ieme(l, i - 1) fsi

        ins(listenouv, i, y) = adjt(listenouv, y)
        ins(adjt(l, x), i, y) =  si (i < 1) ou i == 1  alors adjt(adjt(l, x), y)
/* cas inutile */                sinon si i == 2 alors adjt(adjt(l, y), x)
/* cas inutile */                sinon si (lgr(l) + 1 < i) alors adjt(adjq(l, y), x)
                                 sinon adjt(ins(l, i - 1, y), x) fsi fsi fsi
        sup(listenouv, i) = listenouv      /* 2e test inutile */
        sup(adjt(l, x), i) =     si (i < 1) ou (lgr(l) + 1 < i) alors adjt(l, x)
                                 sinon si i == 1 alors l
                                 sinon adjt(sup(l, i - 1), x) fsi fsi
        chg(listenouv, i, y) = listenouv   /* 2e test inutile */
        chg(adjt(l, x), i, y) =  si (i < 1) ou (lgr(l) + 1 < i) alors adjt(l, x)
                           sinon si i == 1 alors adjt(l, y)
                           sinon adjt(chg(l, i - 1, y), x) fsi fsi
        lig(listenouv, i) = listenouv
        lig(adjt(l, x), i) = si i ≤ 1 alors listenouv
/* cas inutile */        sinon si i == 2 alors adjt(listenouv, x)
/* cas inutile */        sinon si lgr(l) + 1 < i alors adjt(l, x)
                         sinon adjt(lig(l, i - 1), x) fsi fsi fsi
        lid(listenouv, i) = listenouv
        lid(adjt(l, x), i) = si i == 0 alors adjt(l, x)
/* cas inutile */        sinon si i == 1 alors l
/* cas inutile */        sinon si lgr(l) + 1 ≤ i alors listenouv
                         sinon lid(l, i - 1) fsi fsi fsi
fspéc
```

Pour préparer à une implantation efficace, nous proposons de travailler par raffinement de la spécification.

Exemple Travaillons à la transformation de l'opération donnant le $i^{ème}$ élément en en donnant plusieurs variantes se rapprochant de plus en plus de l'implantation. Après la version par rapport aux générateurs de base des entiers :

```
        adjt(l, x)[i] = si i == 1 alors x sinon l[i-1] fsi
```

nous pouvons écrire une spécification directe, tout d'abord récursivement :

```
        l[i] = si i == 1 alors tête(l) sinon supt(l)[i-1] fsi
        l[i] = si i == lgr(l) alors queue(l) sinon supq(l)[i] fsi
```

puis avec une itération dans un sens de parcours ou dans l'autre :

```
l[i] = tête (l1)
avec (l1, k) = init (l, 1) tant que (k < i) rép (supt(l1), k+1) frép

l[i] = queue (l1)
avec (l1, k) = init (l, lgr(l))  tant que (k > i)
                                 rép (supq(l1), k-1) frép
```

3.2 Ordre sur les éléments des listes triées

Dans les listes triées sur lesquelles nous voulons travailler maintenant, nous pouvons ranger les éléments de manière croissante ou décroissante mais il nous faut une relation d'ordre sur les éléments. Nous commençons par quelques rappels mathématiques.

Définition Un *préordre* sur un ensemble E est une relation binaire dans E notée habituellement ≤, réflexive et transitive. La relation notée ~ définie par : $x \sim y = x \leq y$ et $y \leq x$ est une relation d'équivalence.

Si un préordre partiel large ≤ et strict < est exigé, nous utilisons la théorie PREORD où, de plus, sont définis le préordre partiel strict noté < et l'équivalence notée ~ :

```
spéc PREORD étend TRIV              /* préordre partiel */
opérations
        _≤_, _<_ , _~_ : S S → Bool
        /* préordre large, strict, équivalence */
axiomes x, y, z : S
        (o1) x ≤ x = vrai                                  /* réflexivité de ≤ */
        (o2) (x ≤ y et y ≤ z) =>  x ≤ z = vrai             /* transitivité de ≤ */
        (o3) x < y = x ≤ y et ¬ y ≤ x                      /* préordre strict < */
        (o4) x ~ y = x ≤ y et y ≤ x            /* définition de l'équivalence */
fspéc
```

Définition Un *ordre* est un préordre antisymétrique c'est à dire que l'antisymétrie est vérifiée : ((x ≤ y et y ≤ x) => x == y) = vrai. Dans ce cas, ~ et l'égalité dans E, notée == coïncident.

S'il s'agit d'un ordre partiel, nous prenons ORD, où l'équivalence et l'égalité sont confondues :

spéc ORD **étend** PREORD, EG /* ordre partiel */
axiomes x, y, z : S
  (o5) x ~ y = x == y /* l'équivalence est l'égalité */
fspéc

Pour ranger des éléments de manière croissante ou décroissante, un préordre suffit : il n'est pas nécessaire d'avoir l'antisymétrie. Donnons un exemple.

Exemple Prenons pour *E* le type `Personne`, déclaré en C :

```
typedef struct {Chaine nom; Nat num;} Personne;
```

Nous pouvons y définir une relation de préordre ≤ : pour tout couple `(x,y)` de personnes :

$$x \leq y \text{ } ssi \text{ } x.\texttt{nom} \leq_{\texttt{ch}} y.\texttt{nom}$$

où $\leq_{\texttt{ch}}$ est l'ordre habituel des chaînes de caractères, ou *ordre lexicographique*. Cette relation est souvent utilisée pour constituer des listes de personnes rangées par ordre alphabétique.
Il est facile de vérifier que c'est un préordre. Mais ce n'est pas un ordre.
En effet, avec `x = ("Dupont", 123)` et `y = ("Dupont", 456)` nous avons bien $x \leq y$ et $y \leq x$ sans que `x == y` soit vrai. La relation définie par $x \sim y = x \leq y$ et $y \leq x$ est telle que $x \sim y = x.\texttt{nom} ==_{\texttt{ch}} y.\texttt{nom}$,
où $==_{\texttt{ch}}$ est la comparaison des chaînes de caractères.
C'est une relation d'équivalence.

Définition Une relation binaire ≤ sur *E* est dite *totale* si, pour tout couple (*x*, *y*) de $E \times E$, $x \leq y$ ou $y \leq x$ (il ne s'agit pas du ou *exclusif).* Alors 2 éléments de E sont toujours comparables.

S'il s'agit d'un préordre total, nous retenons PREORDT :
spéc PREORDT **étend** PREORD /* préordre total */
axiomes x, y, z : S
  (o6) x ≤ y **ou** y ≤ x = vrai
fspéc

Enfin, pour un ordre total, nous utilisons ORDT :
spéc ORDT **étend** ORD /* ordre total */
axiomes x, y, z : S
  (o7) x ≤ y **ou** y ≤ x = vrai
fspéc

3.3 Listes triées

Définition Une liste $(x_1, x_2, ..., x_k)$ sur E muni d'un préordre total ≤ est *triée* de manière croissante si $x_i \leq x_j$ pour $1 \leq i < j \leq k$.

Nous supposons alors la sorte S munie d'un opérateur booléen _≤_ satisfaisant les axiomes d'un *préordre total*. Nous voulons en plus == et ≠ figurant dans EG. Ceci est offert par la spécification PREORDTEG.

spéc PREORDTEG **étend** EG,PREORDT **fspéc**

Une spécification générique de listes de la sorte Listet contenant des objets de la sorte S triés de manière croissante s'écrit :

```
spéc LISTET(PREORDTEG) étend LISTE1(EG → PREORDTEG)
/* version simplifiée : spéc LISTET étend LISTE, PREORDTEG   */
sorte Listet   /* sous-sorte des listes triées */
sous-sorte Listet ≤ Liste    /* facultatif */
opérations
        triée : Liste → Bool          /* test de tri d'une liste quelconque */
        listenouv : → Listet          /* liste triée vide */
        adjt : Listet S → Listet      /* nouvelle précondition */
        insav : Listet S → Listet     /* adjonction juste avant les élts ≥ */
        insap : Listet S → Listet     /* adjonction juste après les élts ≤  */
        rech : Listet S → Nat         /* plus petit rang d'un élément */
préconditions lt : Listet ; x : S
        pré adjt(lt, x) =  si vide(lt) alors vrai sinon x ≤ tête (lt) fsi
        /* expliquer la précondition */
axiomes l : Liste ; lt : Listet ; x, y : S
        (lt0) triée(lt) = vrai          /* invariant des listes triées */
        (lt1) triée(listenouv) = vrai   /* définition de triée */
        (lt2) triée(adjt(listenouv, x)) = vrai
        (lt3) triée(adjt(adjt(l, y), x)) = x ≤ y ^ triée(adjt(l, y))
        (lt4) insav(listenouv, x) =   adjt(listenouv, x)
        (lt5) insav(adjt(lt, y), x) =   si x ≤ y alors adjt(adjt(lt, y), x)
                                        sinon adjt(insav(lt, x), y) fsi
        (lt6) insap(listenouv, x) =   adjt(listenouv, x)
        (lt7) insap(adjt(lt, y), x) =   si x < y alors adjt(adjt(lt, y), x)
                                        sinon adjt(insap(lt, x), y) fsi
        (lt8) rech(listenouv, x) = 0
        (lt9) rech(adjt(lt, y), x) =    si x < y alors 0
                                        sinon si  x == y alors 1
                                        sinon avec i = rech(lt, x)
                                                si i == 0 then 0 else i+1 fsi
                                        fsi fsi
fspéc
```

Ainsi, insav(l, x) insère x dans l juste avant les éléments y de l tels que x ≤ y, et insap(l, x) juste après les éléments y de l tels que y ≤ x.

Puisqu'elle en est une extension, la théorie PREORDTEG satisfait bien toutes les propriétés de la théorie EG, ce qui justifie l'instanciation LISTE1(EG → PREORDTEG)**.** L'opérateur triée teste si une liste quelconque de sorte Liste est triée de manière croissante.

La sorte des listes triées est en fait définie par spécialisation des listes générales grâce à l'*invariant* (lt0) triée(lt) = vrai. Si l'on désire uniquement des listes triées, il vaut mieux *cacher* l'opérateur adjt conduisant à des listes générales, c'est-à-dire permettre la construction des listes triées uniquement en utilisant listenouv, insav, insap, ou leurs dérivés. Bien sûr supt laisse triée une liste triée. La propriété suivante peut être obtenue par induction.

Proposition Une liste construite par composition des opérateurs listenouv, insav, insap et supt est une liste triée de manière croissante.

3.4 Implantation des listes linéaires par contiguïté

En *mémoire centrale*, nous avons les mêmes implantations que pour les piles ou les files contiguës, dans un tableau dont les éléments sont alloués de manière statique ou dynamique. Une liste occupe un ensemble de places contiguës et sa représentation peut, comme pour les files, comporter par exemple un tableau, la longueur et l'indice de la tête, car elle peut commencer n'importe où dans le tableau. Les listes triées sont implantées de même, avec un *invariant de représentation* imposant que les éléments soient rangés dans le tableau en ordre croissant. La déclaration C correspondante est :

```
typedef S Stab[N];
typedef struct strliste {Stab v; Nat h; Nat l;}
      Liste, Listet;
```

Nous ne détaillons pas plus cette implantation contiguë, sans particularité notable par rapport à celles des piles et des files, hormis le fait que, si les mises à jour en tête ou queue y sont aisées, il n'en va pas de même pour les mises à jour à l'intérieur. Ainsi, ajouter ou supprimer un élément à l'intérieur d'une liste, parce que l'on veut la garder triée par exemple, oblige à faire des décalages, ce qui est coûteux en temps de calcul. Une autre solution pour supprimer des éléments est de marquer les places libérées et de les réutiliser par la suite. Ceci oblige à une gestion un peu compliquée.

3.5 Représentations par chaînage

Avec une allocation dynamique de cellules de mémoire, nous pouvons définir un type de listes de manière analogue à ce que nous avons vu pour les piles et les files. Avec un pointeur sur la tête de chaque liste, nous écrivons en C :

```
typedef struct strliste {S v; struct strliste *s;}
                                        Strliste, *Liste;
```

Comme précédemment, nous pouvons ajouter un en-tête avec un entier représentant la longueur de la file, un pointeur sur la queue, etc. La programmation des opérations en dérive immédiatement.

Une liste chaînée *circulaire* (Fig. 7) est telle que le dernier élément pointe sur le premier, ce que nous avons déjà vu pour les files. Dans ce cas, on a besoin seulement du pointeur de queue. L'intérêt est de permettre d'effectuer facilement à la fois les opérations en tête, ainsi que des *permutations circulaires* vers la gauche.

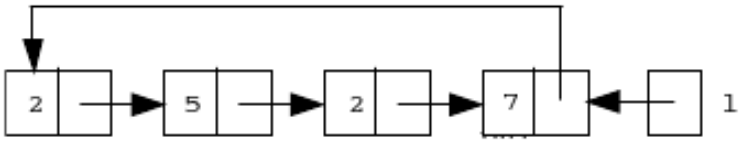


Figure 7 : Liste circulaire

Une liste *doublement chaînée* (Fig. 8), dite aussi *bilatère*, *symétrique* ou *bi-directionnelle*, est une liste circulaire chaînée dans les deux sens. En plus des données déjà vues pour une liste circulaire simple, chaque cellule comporte un pointeur vers la cellule précédente. Ceci facilite les opérations en queue et en tête, le parcours de liste dans les deux sens, et les permutations circulaires vers la gauche et vers la droite. Les exercices 6.11 à 6.14 reprennent toutes ces questions, notamment avec la légende de Flavius Josèphe, ainsi que celle des listes *pointées*, comparables aux files pointées.

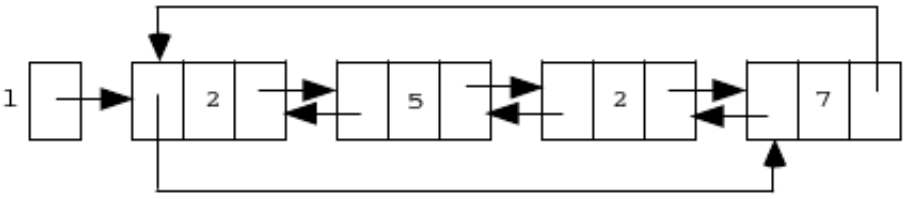


Figure 8 : Liste bilatère

Une bonne implantation chaînée des listes doit valider la spécification LISTE1. Les confusions sont faciles à éviter. En revanche, ces implantations peuvent présenter des rossignols, dus à des erreurs dans la gestion des pointeurs, comme celles que nous avons vues pour les piles. On les évite en ne travaillant jamais directement sur la représentation et en utilisant toujours les fonctions adéquates issues de la spécification.

Nous pouvons réaliser notre propre mécanisme d'allocation en gérant nous-même un espace-mémoire de `M` cellules. Celui-ci peut être vu comme un tableau `t` de structures du type Elem avec les déclarations :

```
typedef struct selem {S v; Ent s;} Elem;
Elem t[M];
```

Les éléments successifs d'une liste sont alors chaînés avec des pointeurs qui sont en fait des indices du tableau `t`. Le pointeur null est représenté ici par -1. Une liste est alors simplement l'indice de la première cellule :

```
typedef Nat Liste;
```

Exemple Nous considérons une représentation chaînée de la liste de caractères l = ('B', 'A', 'D', 'A', 'B'), avec M = 12 (Fig. 9). ∎

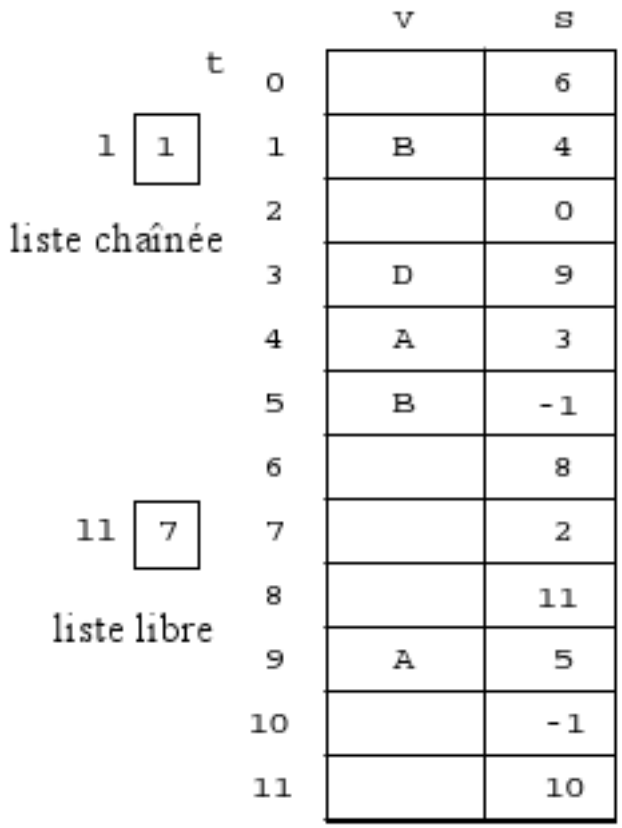


Figure 9 : Liste chaînée et liste libre dans un tableau

Il est commode de chaîner les places vides dans une liste ll appelée habituellement *liste libre*. Ainsi, lors de l'allocation d'une cellule à la liste l, il faut retirer cette cellule de ll ; lors de la restitution d'une cellule de l inutilisée, il faut la remettre dans ll. Le plus efficace est toujours de faire ces opérations en tête de ll, en fait de gérer la liste libre comme une pile.

Bien sûr plusieurs listes peuvent cohabiter dans le même espace, mais avec une seule liste libre. Il est préférable que cette gestion reste transparente aux utilisateurs de listes et que, dans la programmation, elle soit réalisée par *effet de bord* sur un tableau t global.

Enfin, signalons que les listes linéaires sont une structure de données de base dans Prolog (1973), premier langage de *programmation logique*, dans ML (1960), langage *fonctionnel* d'ordre supérieur, et dans Simula (1966), premier langage *orienté objets* et aussi dans *Caml.*

3.6 Recherche dans une liste triée (implantation contiguë)

Pour simplifier, nous supposons que (S, ≤) est un *ordre total*. L'opérateur rech de recherche du *plus petit rang* d'un élément x dans une liste l, ici non vide (pour simplifier) *triée*, ou bien 0 s'il ne s'y trouve pas, a été spécifié par :

```
/* plus petit rang d'un élément dans une liste non vide */
        rech : Listet S → Nat
        pré rech(l, x) = ¬ vide(l)
        rech(adjt(lt, y), x) =  si x < y alors 0
                                sinon si  x == y alors 1
                                sinon avec i = rech(lt, x)
                                        si i == 0 then 0 else i+1 fsi
                                fsi fsi
```

Cet axiome exprime que, dans le cas où x appartient à l, i = rech(l, x) est le plus petit rang tel que l[i] == x, sinon i == 0. Mais il ne renseigne pas sur la façon dont i peut être calculé. Il en est une *caractérisation implicite*. Une variante encore plus *implicite* (sans précondition) :

```
rech(l, x) = i
avec
( ¬ app(l, x) => i == 0 )
et
(app(l, x) => l[i] == x et (1 < i => l[i - 1] < x ))
= vrai
```

Il nous faut en donner une définition *explicite* pour une programmation efficace. Nous étudions à cet effet les recherches séquentielle et dichotomique. Nous verrons que ces définitions utilisent abondamment l'accès direct aux éléments d'une liste par leur position. Cet accès doit être très rapide, ce qui exclut le cas d'une représentation chaînée.

En revanche, une *représentation contiguë* convient bien, avec un tableau contenant les éléments de la liste en ordre croissant ou décroissant.

a *Recherche séquentielle*

Dans la recherche séquentielle *croissante*, les éléments de l sont examinés dans l'ordre des positions i croissantes depuis 1, jusqu'à trouver x ou atteindre lgr(l). Une spécification explicite de rech utilisant une fonction auxiliaire rseq définie récursivement est :

```
rech(l, x) = rseq(l, x, 1, lgr(l))
avec rseq(l, x, i, n) = si i == n ou x ≤ l[i]
                        alors si x == l[i] alors i sinon 0 fsi
                        sinon rseq(l, x, i + 1, n) fsi
```

La version itérative correspondante sera obtenue immédiatement :

```
rech(l, x) = si x == l[i] alors i sinon 0 fsi
avec i = init 1 ttq i < n et x > l[i] rep i + 1 frep
avec n = lgr(l)
```

b *Recherche dichotomique*

Si les éléments de l sont tous *distincts*, une spécification explicite de rech utilisant une fonction rdich définie récursivement est :

```
rech(l, x) = rdich(l, x, 1, lgr(l))
avec rdich(l, x, a, b) = si b < a alors 0
        sinon si l[m] == x alors m
        sinon si l[m] < x alors rdich(l, x, m + 1, b)
        sinon rdich(l, x, a, m - 1) fsi fsi
                avec m = [(a + b)/2] fsi
```

où [x] désigne la fonction plancher *plancher* de x, c'est-à-dire le plus grand entier inférieur ou égal à x, opération que nous supposons définie. Quand x est positif, comme ici, [x] est aussi la partie entière de x. Dans le cas où certains éléments peuvent figurer plusieurs fois dans la liste, nous avons :

```
rech(l, x) = rdich(l, x, 1, lgr(l))
```

```
avec rdich(l, x, a, b) = si a == b
        alors si l[a] == x alors a sinon 0 fsi
        sinon si l[m] < x alors rdich(l, x, m + 1, b)
                sinon rdich(l, x, a, m) fsi avec m = ⌊(a + b)/2⌋
                /* si  x < l[m] alors la recherche continue à gauche
                car on cherche la 1ère occurrence de x */
        fsi
```

Ici encore la *dérécursivation* est sans problème.

CONCLUSION

Les piles, files et listes sont des structures de base ayant de nombreuses applications. On peut voir les piles et les files comme des cas particuliers de listes où l'on s'interdit certaines opérations. La tentation existe donc de prendre des listes pour tous les cas où de simples piles ou listes suffisent.

Nous pensons qu'il vaut mieux résister à cette tentation pour deux raisons. La première est d'ordre logique. Pour poser et résoudre un problème, il faut choisir les objets et opérations les mieux adaptés. Trop de richesse risque d'en cacher la véritable nature et d'alourdir sa formulation. La deuxième est d'ordre opératoire. En choisissant la généralité des listes, on risque de passer à côté d'implantations agréables à manipuler et efficaces en temps et en espace qui suffisent à résoudre le problème.

Pour prendre un exemple, on peut implanter des listes avec un chaînage double, alors qu'une simple pile bornée implantée de manière contiguë est suffisante. Si la deuxième implantation oblige à surdimensionner un peu la réservation statique de mémoire, elle évite les coûteuses allocations et désallocations à répétitions de cellules à trois places, avec une gestion complexe de pointeurs.