ALGORITHMIQUE
ET STRUCTURES DE DONNEES

Pierre Tellier*

17/05/2006

* Département d'Informatique de l'Université Louis Pasteur,
7 rue René Descartes, F67084 STRASBOURG CEDEX,
e-mail : tellier@dpt-info.u-strasbg.fr, Tel : +33 (0) 3 90 240 300

Table des matières

Structures de données

De nombreux objets traités par les programmes ne peuvent pas être représentés à l'aide d'un seul nombre ou d'une chaîne, mais sont constitués naturellement de plusieurs informations :
une date = un jour, un mois, une année ;
un nombre complexe = une partie réelle, une partie imaginaire ;
un tableau = l'adresse de son premier élément, sa taille physique et sa taille effective ;
une sphère = un centre, un rayon ;
une personne = un nom, un prénom ;
une adresse = un numéro, une rue, une ville, un code postal;
...
Une information élémentaire dans un objet est appelée un *champ*. Pour accéder à l'information contenue dans un champ d'un objet composite, il suffit d'ajouter un point au nom de l'objet et de le faire suivre du nom du champ.

Structures simples

Voici un exemple de définition et d'utilisation de structures. Il porte sur les dates, constituées de 3 informations : nom de jour (dans la semaine), jour dans le mois, mois et année.

```
typedef struct { /* ou typedef struct T_Date */
      int jour, mois, annee;
      char lejour[9]; /* "lundi", ...., "dimanche" */
} Date;

Date d1, d2;
d1.jour = 24;
d1.mois = 12;
d1.annee = 1965;
d2.annee = d1.annee+3;
printf("%d-%d-%d\n",d1.jour,d1.mois,d1.annee);}
```

Nous donnons deux versions de fonction de calcul de la veille d'une date. La première délivre un résultat Date (version recommandée, car sans pointeur), la deuxième modifie la date passée en paramètre (à éviter).

```
Date veille(Date d)
{
      Date res = d;
      if (res.jour != 1) res.jour--;
      else if (res.mois != 1) {
            res.mois--;
            res.jour = nbJoursDansMois(res.mois,res.annee);
      }
      else {
            res.jour = 31;
            res.mois = 12;
            res.annee--;
      }
      return res;
}
```

```
void veille(Date *d)
{
      if ((*d).jour != 1) (*d).jour--;
      else if ((*d).mois != 1) {
            (*d).mois--;
            (*d).jour = nbJoursDansMois((*d).mois,(*d).annee);
      }
      else {
            (*d).jour = 31;
            (*d).mois = 12;
            (*d).annee--;
      }
}
```

Remarque : `(*d).jour` est identique à `d->jour`. C'est d'ailleurs la notation que l'on rencontre le plus souvent. En revanche, l'écriture `*f.adresse` est équivalente à `*(f.adresse)`.

Structures imbriquées

Exemple : fiche de renseignement

Nous définissons un type FICHE, destiné à contenir des informations sur des personnes. Ces informations sont : nom, prénom, adresse, date de naissance, sachant qu'une adresse est constituée d'un numéro, d'un nom de rue, d'un code postal et d'une ville, alors que la date de naissance est la réunion de 3 informations : jour, mois et année. Nous définissons notre type ainsi :

```
#include <stdio.h>
#include <string.h>
typedef struct t_date {
      int j, m;
      char a[3]; /* annee = 2 caractères + fin de chaîne = bug an 2000 :) */
} TDate;

typedef struct t_adresse {
      int numero, code;
      char *rue, *ville;
} TAdresse;

typedef struct t_fiche {
      char *nom, *prenom; /* pointeurs sur nom et prenom */
      TDate naissance; /* date de naissance */
      TAdresse *adresse; /* pointeurs sur adresse => pas présente */
} TFiche;
```

Nous écrivons ensuite les fonctions permettant d'initialiser une fiche. Pour réduire le risque d'erreurs, il est impératif de procéder comme dans l'exemple, c'est-à-dire de ne pas tenter d'accéder directement aux sous-champs des champs, mais de passer par des objets intermédiaires qui n'ont qu'un seul niveau de décomposition et qui sont plus simples à initialiser. Certaines fonctions fonctionnent sur le mode des paramètres modifiables, ce qu'il vaut mieux éviter en principe.

```
void saisieAdresse(TAdresse *adr)
{
      char chaine[1024];
      int n;

      printf("Saisie de l'adresse\n");
      printf("Numero : "); scanf("%d", &n);
      adr->numero = n;
      printf("Nom de la rue : "); scanf("%s", chaine);
      adr->rue = strdup(chaine);
      printf("Code Postal : ");scanf("%d",&adr->code); /* ou adr.code (rare) */
      printf("Ville : "); scanf("%s", chaine);
      adr->ville = strdup(chaine);

      return ;
}

TDate saisieNaissance(void)
{
      TDate d;

      printf("Saisie de la date de naissance\n");
      printf("Jour : "); scanf("%d", &d.j);
      printf("Mois (1-12) : "); scanf("%d", &d.m);

      printf("Année (19XX) : "); scanf("%s", d.a);

      return d;
}
```

```
void saisieFiche(TFiche *f)
{
      char chaine[1024];
      TAdresse *adr;
      TDate date;

      printf("Nom : "); scanf("%s", chaine);
      f->nom = strdup(chaine);
      printf("Prénom : "); scanf("%s", chaine);
      f->prenom = strdup(chaine);

      adr=(Tadresse*)malloc(sizeof(TAdresse)); /*allouer une nouvelle adresse*/
      /* il ne faudra pas la libérer */
      f->adresse = adr; /* le champ adresse pointe sur la nouvelle adresse */
      saisieAdresse(adr);
      /* plus facile que : scanf("%d", &f->adresse->numero); !!! */

      date = saisieNaissance();
      f->naissance = date; /* recopie de la date f->naissance=saisieDate(); */
      /* plus facile que : scanf("%d", &f->naissance.j); !!! */
}
```

Nous donnons enfin un exemple d'utilisation de ces fonctions : une fonction qui permet de visualiser le contenu de la structure, et un programme principal qui manipule ces fonctions.

```
void visuAdresse(TAdresse adr)
{
      printf("%d rue %s, %d %s\n", adr.numero, adr.rue, adr.code, adr.ville);
}

void visuDate(TDate d)
{
      printf("%d %d 19%s\n", d.j, f.naissance.m, f.naissance.a);
}

void visuFiche(TFiche f)
{
      TAdresse *adr;
      TDate d;

      adr = f.adresse;
      d = f.naissance;
      printf("%s %s\n", f.nom, f.prenom);
      printf("Adresse : "); visuAdresse(*adr);
      printf("Né(e) le : "); visuDate(d);
}

int main(int argc, char **argv)
{
      TFiche fiche;

      saisieFiche(&fiche);
      visuFiche(fiche);
      exit(0);
}
```

Bonne utilisation des structures

Dans la mesure du possible, on évitera de créer des fonctions qui passent des structures par adresse, car l'écriture de ces fonctions est complexe et conduit à de fréquentes erreurs. Pourtant il est nécessaire de connaître cette possibilité vue précédemment, car elle est un peu plus rapide à l'exécution, et de nombreuses fonctions, en particulier système, que l'on peut être amené à utiliser sont écrites sous cette forme.

Une autre amélioration possible consiste à na pas faire directement référence aux noms des champs, mais à utiliser des fonctions d'accès. L'avantage est qu'en cas de modification de la structure ou des identificateurs

des champs, seule la fonction d'accès doit être mise à jour. Voici l'exemple précédent ré-écrit en tenant compte de ces 2 remarques. On continue bien sur à ne pas utiliser plus d'un niveau de structures à la fois.

```
TAdresse setAdrNum(Tadresse adr, int n)
{
      Tadresse adr2=adr;
      adr2.numero = n;
      return adr2;
}

TAdresse setAdrRue(Tadresse adr, char *ch)
{
      Tadresse adr2=adr;
      if (adr2.rue != NULL) free(adr2.rue);
      adr2.rue = strdup(ch);
      return adr2;
}

TAdresse setAdrCode(Tadresse adr, int c)
{
      Tadresse adr2=adr;
      adr2.code = c;
      return adr2;
}

TAdresse setAdrVille(Tadresse adr, char *ch)
{
      Tadresse adr2=adr;
      if (adr2.ville != NULL) free(adr2.ville);
      adr2.ville = strdup(ch);
      return adr2;
}

TAdresse saisieAdresse(void)
{
      char chaine[1024];
      int n, c;
      TAdresse adr;

      printf("Saisie de l'adresse\n");
      printf("Numero : "); scanf("%d", &n);
      adr = setAdrNum(adr, n);
      printf("Nom de la rue : "); scanf("%s", chaine);
      adr = setAdrRue(adr, chaine);
      printf("Code Postal : "); scanf("%d",&c);
      adr = setAdrCode(adr, c);
      printf("Ville : "); scanf("%s", chaine);
      adr = setAdrVille(adr, chaine);

      return adr;
}

int getAdrNum(Tadresse adr)
{
      return adr.numero;
}

char * getAdrRue(Tadresse adr)
{
      return adr.rue;
}
```

```
int getAdrCode(Tadresse adr)
{
      return adr.code;
}

char * getAdrVille(Tadresse adr)
{
      return adr.ville;
}

void visuAdresse(TAdresse adr)
{
      printf("%d rue %s, %d %s\n",  getAdrNum(adr), getAdrRue(adr),
                                    getAdrCode(adr), getAdrVille(adr));
}
```

Structures récursives et croisées

Tableaux multidimensionnels

Les tableaux statiques à 2 dimensions

Ils sont utilisés pour représenter des matrices, ou toutes autres formes de données rectangulaires (images, feuilles de calcul de tableur, modèles numériques de terrain, etc...). C'est aussi de cette façon que sont organisées certains résultats d'interrogation de bases de données : un tableau d'enregistrements, chaque enregistrement étant un tableau de champs, même si ici les champs n'ont pas tous le même type (texte, valeur etc.).

Matrices

Prenons l'exemple des matrices. En mathématique, on désigne chaque élément d'une matrice par a_{ij} où *i* est le numéro de ligne et *j* celui de colonne. En programmation, il en va de même : si *m* est une matrice de taille *M* lignes et *N* colonnes, on la déclare à l'aide d'un tableau par :

```
float m[M][N];
```

Comme pour les tableaux mono-dimensionnels, la numérotation commence à 0 pour les lignes et pour les colonnes, donc l'élément a_{ij} est obtenu par `m[i-1][j-1]`, ou réciproquement, l'élément `m[i][j]` d'un tableau est celui situé à la *(i+1)*ème ligne et *(j+1)*ème colonne. Pour illustrer nos propos, nous prenons le cas des matrices carrées 3x3, pour lesquels nous définissons un nouveau type, le type `Matrice3x3`.

```
typedef float Matrice3x3[3][3];

Matrice3x3 m1,m2;
Matrice3x3 identite= {  {1.0, 0.0, 0.0},
                        {0.0, 1.0, 0.0},
                        {0.0, 0.0, 1.0}};
```

En mémoire, tous les éléments d'un tableau à 2 dimensions sont contigus : d'abord ceux de la première ligne, puis ceux de la deuxième etc. Les tableaux 2D sont donc des tableaux de tableaux. L'adresse de l'élément `m[i][j]` est celle du premier + (*i**Nombre de colonnes) + *j*. Donc on peut très facilement programmer des matrices avec des tableaux à une dimension.

```
Matrice3x3 m;
float t[3*3];
t[i*3+j]=m[i][j];
```

Tout d'abord, voyons une fonction permettant de fixer les valeurs de chaque élément d'une matrice. Pour accéder à tous les éléments, on imbrique deux boucles :

```
pour toutes les lignes faire
        pour tous les éléments sur chaque colonne de la ligne courante faire
                traitement de l'élément (ligne courante, colonne courante)
        finpour
finpour
```

Dans les exemples suivants, on utilise l'indice *i* pour désigner les lignes, et *j* pour les colonnes.

```
void initMatrice(Matrice3x3 m)
{
      int i,j;

      for(i=0;i<3;i++)
            for(j=0;j<3;j++) {
                  printf("entrez l'élément m[%d][%d] : ",i,j);
                  scanf("%d", &m[i][j]);
            }
}
```

De la même manière, la fonction qui affiche à l'écran une matrice 3x3 est :

```
void afficheMatrice(Matrice3x3 m)
{
      int i,j;

      for(i=0;i<3;i++) {
            for(j=0;j<3;j++)
```

```
                  printf("%6.2f ",m[i][j]);
            printf("\n");
      }
}
```

La matrice identité est celle dont les éléments de la diagonale (même numéro de ligne et de colonne) valent 1, alors que tous les autres valent 0.

```
void initIdentite(Matrice3x3 m)
{
      int i,j;

      for(i=0;i<3;i++)
            for(j=0;j<3;j++)
                  m[i][j]=(float)(i==j);
}
```

La recopie d'une matrice dans une autre est quasiment identique à la recopie d'un tableau dans un autre. On peut la pratiquer élément par élément, ligne par ligne ou d'un seul coup.

```
void copieMatrice(Matrice3x3 m, Matrice3x3 m2)
{
      int i,j;

      for(i=0;i<3;i++)
            for(j=0;j<3;j++)
                  m2[i][j]=m[i][j];
}

void copieMatrice(Matrice3x3 m, Matrice3x3 m2)
{
      int i;

      for(i=0;i<3;i++)
            recopieTableauFloat(m[i], 3, m2[i], 3);
            // ou memcpy(m2[i], m[i], 3*sizeof(float));
}

void copieMatrice(Matrice3x3 m, Matrice3x3 m2)
{
      memcpy(m2, m, sizeof(Matrice3x3));
}
```

La somme de 2 matrices A et B est une matrice C où chaque élément c_{ij} est la somme des éléments correspondants a_{ij} et b_{ij}

```
void sommeMatrices(Matrice3x3 A, Matrice3x3 B, Matrice3x3 C)
{
      int i,j;

      for(i=0;i<3;i++)
            for(j=0;j<3;j++)
                  C[i][j]=A[i][j]+B[i][j];
}
```

La transposée A^t d'une matrice A est obtenue en inversant simplement numéros de ligne et numéros de colonne. Dites ce qui peut poser problème dans la première version, et comparez avec la deuxième.

```
void transposeMatrice(Matrice3x3 A, Matrice3x3 At)
{
      int i,j;
      for(i=0;i<3;i++)
            for(j=0;j<3;j++)
                  At[i][j]=A[j][i];
}

void transposeMatrice(Matrice3x3 A, Matrice3x3 At)
{
      int i,j;
      for(i=0;i<3;i++)
```

```
            for(j=0;j<3;j++)
                  At[i][j]=A[i][j];
      for(i=0;i<3;i++)
            for(j=0;j<i;j++)
                  permuterEntiers(&At[i][j], &At[j][i]);
}
```

La matrice *C* résultant du produit de 2 matrices *A* et *B* (le nombre de colonnes de *A* doit être égal au nombre de lignes de *B*, on le note *n*) est obtenu par la formule : $c_{ij} = \sum_{k=0}^{n} a_{ik} b_{kj}$. Notez que l'on passe par une matrice intermédiaire sinon la fonction rend des résultats erronés lorsqu'on l'utilise avec accumulation : `produitMatrices(A, B, A)` ou `produitMatrices(A, B, B)` .

```
void produitMatrices(Matrice3x3 A, Matrice3x3 B, Matrice3x3 C)
{
      Matrice3x3 Ctmp;
      int i,j,k;

      for(i=0;i<3;i++)
            for(j=0;j<3;j++)
            {
                  Ctmp[i][j]=0.0;
                  for (k=0;k<3;k++)
                        Ctmp[i][j] += A[i][k]*B[k][j];
            }
      copieMatrice(Ctmp,C);
}
```

Enfin un exemple d'utilisation de ces fonctions sur les matrices :

```
main()
{
      Matrice3x3 A,B,C;

      initMatrice(A);
      initMatrice(B);
      produitMatrice(A,B,C);
      afficheMatrice(C);
}
```

Et encore quelques fonctions utiles, comme l'application d'une matrice à une vecteur :

```
/* Application d'une matrice à un vecteur */
void appliqueMatrices(Matrice3x3 m, vecteur v1, vecteur v2)
{
      int i,k;
      vecteur v2t;

      for(i=0;i<3;i++)
      {
            v2t[i]=0.0;
            for (k=0;k<3;k++)
                  v2t[i] += m1[i][k]*v1[k];
      }
      recopieTableauFloat(v2t, 3, v2, 3);
}
```

ou l'inversion d'une matrice 3x3 avec calcul du déterminant :

```
/* Inversion de matrice 3x3. Résultat : succès ou échec */
int inverseMatrice(Matrice3x3 m, Matrice3x3 m2)
{
      float determinant, unSurDeterminant;
      float cf[3][3];
      int   i, j, res=0;

      determinant = m[0][0]*(m[1][1]*m[2][2] - m[1][2]*m[2][1]) -
           m[0][1]*(m[1][0]*m[2][2] - m[1][2]*m[2][0]) +
```

```
            m[0][2]*(m[1][0]*m[2][1] - m[1][1]*m[2][0]);
      if (determinant != 0.0)   {
              res = 1;
              unSurDeterminant = 1.0 / determinant;
              cf[0][0] = m[1][1]*m[2][2] - m[1][2]*m[2][1];
              cf[0][1] = -(m[1][0]*m[2][2] - m[1][2]*m[2][0]);
              cf[0][2] = m[1][0]*m[2][1] - m[1][1]*m[2][0];

              cf[1][0] = -(m[0][1]*m[2][2] - m[0][2]*m[2][1]);
              cf[1][1] = m[0][0]*m[2][2] - m[0][2]*m[2][0];
              cf[1][2] = -(m[0][0]*m[2][1] - m[0][1]*m[2][0]);

              cf[2][0] = m[0][1]*m[1][2] - m[0][2]*m[1][1];
              cf[2][1] = -(m[0][0]*m[1][2] - m[0][2]*m[1][0]);
              cf[2][2] = m[0][0]*m[1][1] - m[0][1]*m[1][0];

              for (i = 0; i < 3; i++)
                  for (j = 0; j < 3; j++)
                        m2[i][j] = unSurDeterminant*cf[j][i];
      }
      return res;
}
```

Matrices dynamiques

Après ces exemples sur les matrices de taille 3x3, nous montrons comment manipuler des informations bidimensionnelles de taille quelconque.

Tableaux mono-dimensionnels

La manière la plus pratique de représenter notre tableau à 2 dimensions de tailles inconnues à l'avance est incontestablement l'utilisation d'un tableau alloué dynamiquement. Le principal inconvénient est que l'accès à l'élément sur la i-ème ligne et la j-ème colonne d'une matrice *M* de *L* lignes de chacune *C* colonnes devient un peu moins intuitif : `M[i*C+j]` au lieu de `M[i][j]` pour un véritable tableau bidimensionnel. Il devient bien sûr nécessaire de préciser, en paramètres des fonctions de manipulation de tels tableaux, leurs dimensions. Par exemple voici la fonction qui calcule (si possible) le produit de 2 matrices réécrite selon ces conventions. Chaque matrice est un pointeur sur des nombres réels, complétée par son nombre de lignes `nl` et nombre de colonnes `nc`. On suppose la matrice résultat est préalablement allouée, c'est pourquoi on fournit aussi sa taille physique totale en paramètre. Cette version n'est pas optimale, puisqu'on pourrait faire un calcul incrémental des indices, qui éviterait toutes ces multiplications à chaque itération. Toutefois, pour rester à peu près lisible, nous ne présentons pas de version plus optimisée, ce qui est d'ailleurs inutile car aujourd'hui les compilateurs sont capables de pratiquer eux-mêmes ces améliorations.

```
/* calcul du produit m3 = m1 * m2, *nl3, *nc3 : taille de la matrice produit */
int produitMatrices(float *m1, int nl1, int nc1, float *m2, int nl2, int nc2,
                        float *m3, int taille, int *nl3, int *nc3)
{
      float *m3t=NULL;
      int res=0,i,j,k;
      if ((nc1 == nl2) && (nl1*nc2 <= taille)) {
            res=1; *nl3=nl1; *nc3=nc2;
            CALLOC(m3t, float, nl1*nc2);
            for(i=0;i<nl1;i++)
                  for(j=0;j<nc2;j++) {
                        m3t[i*n1+j]=0.0;
                        for (k=0;k<nc1;k++)
                              m3t[i*nc2+j] += m1[i*nc1+k]*m2[k*nc2+j];
                  }
            memcpy((void *)m3, (void *)m3t, nl1*nc2*sizeof(float));
            FREE(m3t);
      }
      return res;
}
```

Tableaux de tableaux

Cette approche est un peu plus lourde que l'approche précédente, néanmoins elle présente 2 avantages : on conserve la notation intuitive d'accès aux données en laissant le soin au compilateur de calculer lui-même l'adresse de chaque élément. Du fait que chaque ligne est représentée par un tableau dynamique, cette méthode est bien adaptée aux cas où les lignes n'ont pas la même longueur. Cela est assez rare, mais se produit par exemple pour les tables à 2 entrées qui sont symétriques, comme un tableau de distance entre villes, pour lequel on n'a pas besoin d'un rectangle puisqu'un triangle suffit. Nous reprenons l'exemple précédent qui consiste à réaliser le produit de 2 matrices. Les matrices seront représentées à l'aide d'un tableau dynamique de tableaux dynamiques. En langage C, cela s'exprime par un double pointeur ou pointeur de pointeur :

```
typedef float **Matrice;
```

Notre fonction de multiplication de matrice est intéressante, car elle procède à l'allocation d'une matrice temporaire. Dans cette version, nous avons supposé que la matrice `m3` est déjà correctement allouée en dehors de la fonction, et avec les bonnes dimensions.

```
int produitMatrices(Matrice m1, int nl1, int nc1, Matrice m2, int nl2, int nc2,
                        Matrice m3)
{
      Matrice m3t=NULL;
      int res=0,i,j,k;

      if (nc1 == nl2) {
            res=1;
            CALLOC(*m3t, float *, nl1); /* allocation du tableau de lignes */
            for(i=0;i<nl1;i++) {
                  CALLOC(m3t[i], float, nc2); /* allocation des lignes */
                  for(j=0;j<nc2;j++) {
                        m3t[i][j]=0.0;
                        for (k=0;k<nc1;k++)
                              m3t[i][j] += m1[i][k]*m2[k][j];
                  }
            }
            for(i=0;i<nl1;i++) {
                  memcpy((void *)m3[i], (void *)m3t[i], nl1*nc2*sizeof(float));
                  FREE(m3t[i]);
            }
            FREE(m3t);
      }

      return res;
}
```

Pour améliorer la lisibilité, nous conseillons vivement d'écrire les fonctions d'allocation et de dés-allocation de ce type de matrice :

```
Matrice alloueMatrice(int nl, int nc)
{
      int i;

      Matrice m=NULL;
      CALLOC(m, float *, nl); /* allocation du tableau de lignes */
      for(i=0;i<nl1;i++) CALLOC(m[i], float, nc); /* allocation des lignes */

      return m;
}

Matrice desAlloueMatrice(Matrice m, int nl, int nc)
{
      int i;
      for(i=0;i<nl;i++) FREE(m[i]);
      FREE(m);
      return m;
}
```

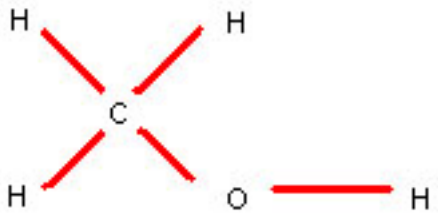
Ce qui donne finalement la fonction de multiplication de matrices :

```
int produitMatrices(Matrice m1, int nl1, int nc1,Matrice m2, int nl2, int nc2,
                        Matrice m3)
{
      Matrice m3t=NULL;
      int res=0,i,j,k;

      if (nc1 == nl2) {
            res=1;
            m3t = allouеMatrice(nl1, nc2);
            for(i=0;i<nl1;i++) {
                  for(j=0;j<nc2;j++) {
                        m3t[i][j]=0.0;
                        for (k=0;k<nc1;k++)
                              m3t[i][j] += m1[i][k]*m2[k][j];
                  }
            }
            for(i=0;i<nl1;i++)
                  memcpy((void *)m3[i], (void *)m3t[i], nl1*nc2*sizeof(float));
            m = desAllouеMatrice(m, nl1, nc2);
      }
      return res;
}
```

Quelques représentations de molécules

Prenons par exemple la molécule CH_3OH, dont les liaisons entre atomes sont illustrées par la figure qui suit :



Voici quelques façons de représenter cette information :

- Atomes et couples d'atomes (+ valence)
  (C, H, H, H, O, H) + ((1,2), (1,3), (1,4), (1,5), (5,6))
- Atomes et tableau de liaisons
  (C, H, H, H, O, H) +

C(1)	H(2)	H(3)	H(4)	0(5)	H(6)	
	x	x	x	x		C(1)
x						H(2)
x						H(3)
x						H(4)
x					x	O(5)
				x		H(6)

C(1)	H(2)	H(3)	H(4)	0(5)	H(6)	
	x	x	x	x		C(1)
						H(2)
						H(3)
						H(4)
					x	O(5)
						H(6)

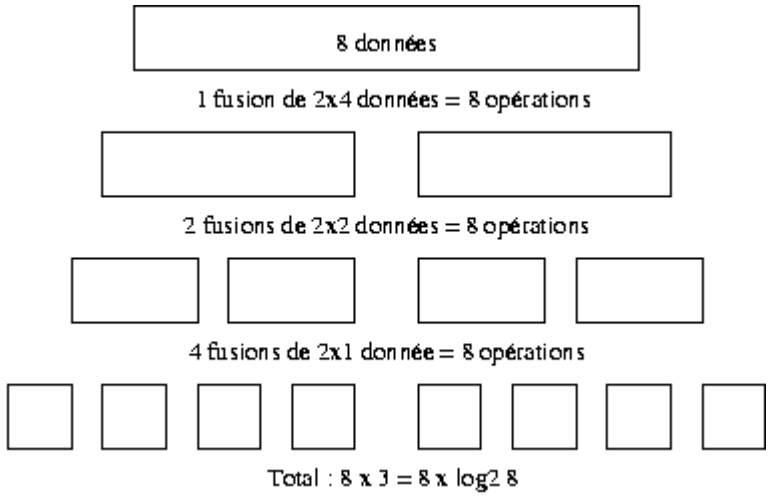
- Atomes et listes de voisins
  - (C, H, H, H, O, H) + (1,(2,3,4,5)), (2,(1)), (3,(1)), (4,(1)), (5,(1,6)), (6,(5))
  - (C, H, H, H, O, H) + ((2,3,4,5), (1), (1), (1), (1,6), (5))
- Graphe avec pointeurs
- SMILE

Tableaux de dimension supérieure à 2

Tris de tableaux

Tri fusion

Le coût du tri naïf (et de ses variantes) étant de l'ordre de n^2, le tri d'un demi tableau sera de l'ordre de $(n/2)^2$, soit $n^2/4$. Le coût du tri des 2 sous-tableaux sera alors de $n^2/2$. Si on applique la même stratégie pour trier les deux sous-tableaux, on obtient un tri donc le coût devient alors de l'ordre de $n.log\ n$.



Il ne reste plus qu'à mettre en place les fonctions qui effectuent la séparation du tableau en sous-tableaux, et qui fusionnent les 2 sous-tableaux triés en un nouveau tableau trié. Un tableau vide ou contenant un seul élément est considéré comme déjà trié. Ce tri par fusion peut être réalisé sans utiliser les tableaux intermédiaires `tab1` et `tab2`.

```
void separe(int tab[], int n, int tab1[], int *n1, int tab2[], int *n2)
{
      int i,j=0;
      *n1= n/2; *n2=n-*n1;
      for(i=0; i<*n1; i++) tab1[i]=tab[i];
      for(i=*n1; i<n; i++) tab2[j++]=tab[i];
}

void fusion(int tab1[], int n1, int tab2[], int n2, int tab[])
{
      int i,j=0,k=0;
      for(i=0; i<n1+n2; i++)
            if (j >= n1) tab[i] = tab2[k++];
            else if (k >= n2) tab[i] = tab1[j++];
            else if (tab1[j] < tab2[k]) tab[i] = tab1[j++];
            else tab[i] = tab2[k++];
}

void triFusion(int tab[], int n)
{
      int tab1[MAXTAILLE], tab2[MAXTAILLE], n1, n2;

      if (n > 1) {
            separe(tab, n, tab1, &n1, tab2, &n2);
            triFusion(tab1, n1);
            triFusion(tab2, n2);
            fusion(tab1, n1, tab2, n2, tab);
      }
}
```

Tri rapide

Le principe de la méthode précédente est très intéressant, puisqu'il permet de réduire de manière significative le nombre d'opérations à effectuer. Néanmoins, le coût des fonctions de séparation et de fusion est encore trop important. En particulier la fusion exige la comparaison des premiers éléments de chaque sous-tableau trié. La méthode dite du tri rapide remédie à cela par sa façon de diviser le tableau tout à fait originale : à partir d'un tableau, et d'une valeur appelée pivot, elle coupe en deux parties le tableau de telle sorte que tous les éléments de la première partie soient inférieurs (ou égaux) au pivot et ceux de la deuxième partie soient supérieurs. De cette manière, la fusion sera grandement simplifiée puisqu'il suffira de mettre bout à bout les 2 sous-tableaux. La difficulté consiste à choisir un pivot, qui donne deux sous tableaux dont les tailles sont le plus proche possible, et à mettre en œuvre le plus efficacement possible la séparation. Cela sera fait de telle sorte que le tri est fait directement dans le tableau, sans tableaux intermédiaires. Des analyses statistiques ont montré que dans la plupart des cas, on peut prendre comme pivot la médiane de 3 valeurs du tableau prises au hasard. Une fois ce pivot trouvé, on parcourt le tableau en partant de ses 2 extrémités, et dans chaque sens, on s'arrête quand on trouve une valeur qui n'est pas dans la bonne partition. Lorsqu'on a deux éléments à la mauvaise place, on les permute. On répète ce calcul jusqu'à ce qu'on se croise.

```
int lePivot(int tab[], int i, int j)
{
      int v[3];
      v[0] = tab[i+rand()%(j-i+1)];
      v[1] = tab[i+rand()%(j-i+1)];
      v[2] = tab[i+rand()%(j-i+1)];
      triBulle(v,3); // ou un  autre tri
      return v[1];
}
int separe(int tab[], int pivot, int deb, int fin)
{
      int i=deb, j=fin, finBoucle=(deb>=fin);
      while(!finBoucle) {
            while(tab[i]<=pivot) i++;
            while(tab[j]>=pivot) j--;
            if (i>=j) finBoucle=1;
            else permuterEntiers(&tab[i], &tab[j]);
      }
      return j;
}
void triRapide(int tab[], int deb, int fin)
{
      int pivot, ppivot;
      if (fin > deb) {
            pivot = lePivot(tab, deb, fin);
            ppivot = separe(tab, pivot, deb, fin);
            triRapide(tab, deb, ppivot);
            triRapide(tab, ppivot+1, fin);
      }
}
```

En langage C il existe une fonction de tri optimisée de tri avec pivot. Elle porte le nom de `qsort` et on l'utilise ainsi : il faut disposer d'une fonction de comparaison de 2 éléments *a* et *b* retournant une valeur négative si *a* doit être avant *b*, positive dans le cas contraire. Cette fonction prend en paramètres les pointeurs sur ces 2 éléments. Comme la fonction `qsort` est générique (prévue pour fonctionner sur n'importe quel type de données), il est nécessaire de se conformer au type `void` = *n'importe quel type*. Voici comment l'employer pour trier un tableau d'entiers :

```
int compare(int *a, int *b)
{
      return *a-*b;
}
int tri(int *tab, int n)
{
   qsort((void *)tab,n,sizeof(int),(int(*)(const void *,const void *))compare);
}
```

Tables

Ce chapitre est reproduit du cours d'Algorithmique Avancé, par Jean-Michel DISCHLER, dischler@dpt-info.u-strasbg.fr, avec son aimable autorisation.

Définition

Une table t est une fonction partielle de E dans S, où E est un type d'entrées (aussi dit index, indice ou clé) et S un type de valeurs.
Exemple : une table *t* : chaîne → chaîne fait correspondre à un nom de département le code postal. Par exemple, pour le Bas-Rhin le code 67.
Les opérations usuelles sur les tables sont :
– ADJONCTION*(t, e, s)* → *t* : adjonction d'un élément à la table, de clé e et de valeur s ;
– SUPPRESSION*(t, e)* → *t* : suppression de l'élément de clé e ;
– ELEMENT*(t, e)* → *s* : accès à la valeur de l'élément de clé e ;

Représentation des tables

Il est important de ne pas confondre table et tableau. Un tableau est un cas particulier de la table avec comme entrée (index ou clé) un entier et comme type de valeur, le type du tableau. Evidemment, le tableau peut servir de moyen de modéliser des tables, mais ce n'est pas la seule approche. Par exemple, dans certains cas une table peut être représentée par une fonction. La table qui associe aux *n* premiers entiers leur carré, est directement donnée sous la forme d'une fonction mathématique du calcul de carré.
Pour modéliser des tables, on passe souvent par un ensemble d'adresses intermédiaires :
E →A→S
i →a(i) →f(a(i))=t(i)
On appelle *a*() la fonction d'adressage et *f* () la fonction d'accès à la valeur recherchée.

fonction d'accès

Accès direct
Dans ce cas *f* donne le contenu d'une cellule dont on connaît l'adresse. Pour une entrée *i*, *a*(*i*) donne l'indice de la case du tableau contenant *t*(*i*) = *f* (*a*(*i*)).
Exemple : La table du nombre des jours dans le mois. On utilise un tableau *v*[12]. Avec *a*(*i*) = *i*-1, on obtient *t*(*i*) = *v*[*i*-1].
Accès indirect
Ici la fonction *f* peut être vue comme donnant le contenu du contenu d'une cellule dont on connaît l'adresse. On peut utiliser un tableau *ta* pour ranger les adresses des valeurs de la table *t*. Celles-ci peuvent être mémorisées dans un autre tableau *v*.

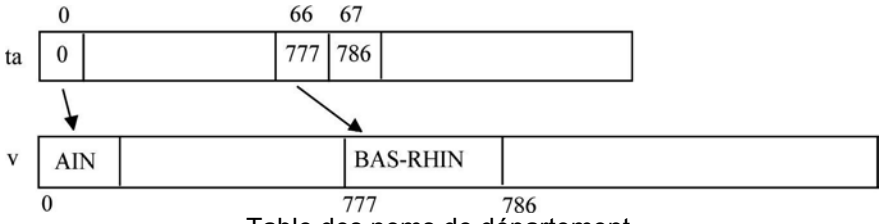


Table des noms de département.

Exemple : Soit *t* la table des noms de département à partir de leurs numéros, allant de 1 à 95. *a*(*i*) = *i*-1. Dans le schéma ci-dessus, on voit que *ta* donne l'indice du caractère dans le tableau *v* des chaînes. Cet exemple est utile dans le cas où le tableau est statique, car les modifications ne sont pas très aisées.

Modes d'adressage

Adressage calculé
Le principe consiste à calculer *a*(*i*) à l'aide d'une fonction injective (c'est une fonction telle que *a*(*i*) = *a*(*j*) implique *i* =*j*). C'était déjà le cas dans les exemples précédents.
Exemple : En langage ADA, un tableau à plusieurs dimensions se déclare de la manière suivante : t : array ($a_1..b_1$,$a_2..b_2$, ... , $a_p..b_p$) of TYPE. Un tableau de ce type peut être vu comme une table associant à des p-uplets une valeur de TYPE. Cette table peut elle même être implémentée sous forme de tableau *v* dont les éléments sont indicés de 0 à *n*-1, avec $n = (b_1-a_1+1)(b_2-a_2+1)...(b_p-a_p+1)$.

Il faut passer d'un p-uplet d'indices $i=(i_1, i_2, ..., i_p)$. à l'indice $a_p(i)$ correspondant dans v. Le calcul se fait par relation de récurrence : $a_p(i) = (b_p-a_p+1)a_{p-1}(i_1, ..., i_{p-1})+i_p-a_p$. En développant cette formule, on obtient une combinaison linéaire des $i_k - a_k$.

Adressage associatif

Dans un adressage associatif, on représente uniquement les entrées effectives par une liste dont les adresses sont associées.

Exemple : Soit une table *t* qui associe à un nom de département le nom du chef-lieu.

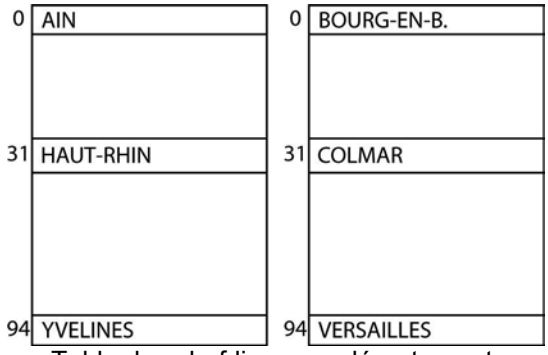


Table des chef-lieux par département.

$a(i)$ est l'indice de la case contenant i. La valeur de $t(i)$ est obtenue par un tableau associé $v[a(i)]$. Dans ce cas, $a(i)$ est obtenu par recherche dichotomique (les départements sont triés par ordre alphabétique). La fonction d'accès est directe. Le problème général de ce type de représentation est que les opérations d'adjonctions ou suppressions sont lourdes. Il est intéressant pour ces traitements d'utiliser une représentation mixte, contiguë et chaînée.

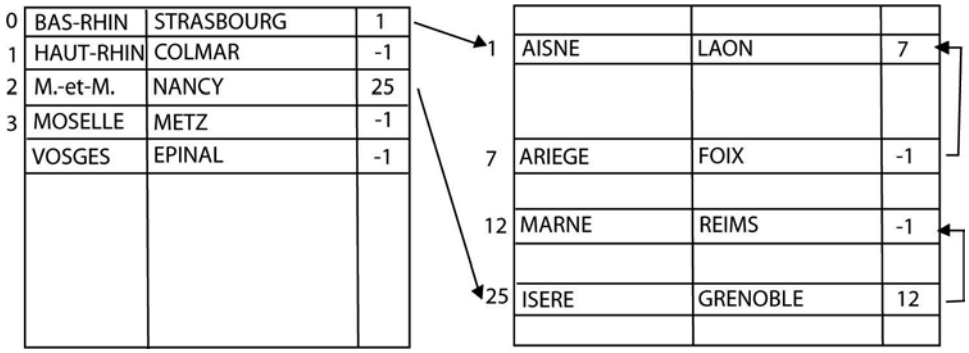


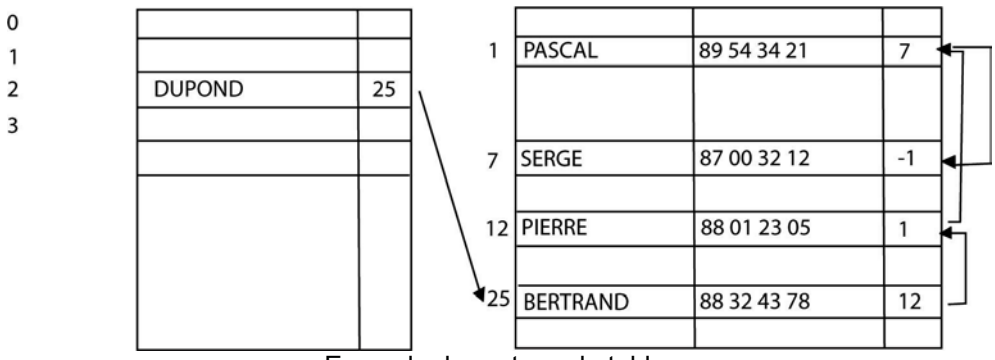
Table des chef-lieux par département, avec une structure mixte contiguë et chaînée.

Dans l'exemple précédent, on utilise deux tableaux de structures pour réaliser la table. Le premier tableau contient les éléments de manière contiguë et triée. Dans le second les éléments sont chaînés. Le chaînage est réalisé par le biais d'indices. Le second tableau peut être considéré comme une zone de débordement. La seule difficulté avec ce type de représentation est la gestion d'un nombre éventuellement important de débordements. En pratique, quand on s'aperçoit que l'on utilise beaucoup d'éléments en débordement, il faut réorganiser les listes en déplaçant des éléments du second tableau vers le premier tableau.

Partage de tables

Il est parfois plus efficace de partager (partitionner) les entrées en sous-ensembles, c'est-à-dire d'avoir des sous-tables à plusieurs niveaux. A deux niveaux, on parle de table majeure et table mineure.

Exemple : Nous voulons réaliser une table qui associe à une personne (nom et prénom) un numéro de téléphone. Il se peut que plusieurs personnes portent le même nom. Nous faisons un partage de table où nous associons toutes les personnes portant le même nom dans la table majeure. Dans la table mineure nous plaçons le prénom et le numéro de téléphone. On parle dans ce cas d'un rangement partitionné, avec adressage associatif.



Exemple de partage de table.

Le rangement dispersé

Le rangement dispersé (hachage) est un rangement où la fonction d'adressage est directement calculée. Le hachage a la particularité de permettre un temps de recherche constant, c'est-à-dire indépendant du nombre de données.

Exemple : On veut gérer une table de personnes indexées par leur prénom. A chaque prénom on associe un entier $h(i)$ de 0 à 12 en procédant comme suit :

– Attribuer aux lettres leur rang dans l'alphabet (A->1, B->2, etc.)

– Ajouter les valeurs des rangs pour chaque lettre composant le nom i.

– Ajouter au nombre obtenu le nombre de lettre de i.

– Prendre le reste de la division de ce nombre par 13.

On obtient pour les prénoms : h("serge")=7, h("odile")=11, h("luc")=0, h("anne")=12, h("basile")=2, h("elise")=3.

On peut placer les prénoms dans un tableau en utilisant comme position la valeur de $h(i)$. Le problème est que si l'on veut insérer paule, on constate que h("paule")=2. On dit qu'il y a collision primaire entre paule et basile.

Toute la difficulté du hachage consiste :

– à trouver une fonction de hachage appropriée pour éviter le plus possible ces collisions

– quoi qu'il en soit il est impossible d'éviter toujours les collisions (surtout lorsque l'ensemble d'entrées est grand), de ce fait il faut trouver un moyen de résoudre les collisions sans pour autant augmenter inconsidérément l'espace de mémorisation.

Les fonctions de hachage

Le choix d'une bonne fonction de hachage dépend de l'ensemble d'entrées sur lequel on travaille réellement. Par exemple la fonction qui associe à une chaîne le rang de son initiale est uniforme sur l'ensemble des chaînes possibles mais pas sur l'ensemble des noms communs français. Il y a par exemple plus de mots qui commencent par B que par Z.

Voici quelques types et techniques :

par extraction

On extrait un certain nombre de bits de la représentation binaire.

Exemple : extraire les bits 1, 2, 7 et 8 en comptant à partir de la droite "ET" => 00101 10100, h("ET")=1000=8, "IL" => 01001 01100, h("IL")=0000=0

Mais ce type de fonction n'est pas vraiment bon, car il ne dépend que d'une partie des données.

Compression

On utilise tous les bits pour obtenir l'indice dans le tableau. On coupe les chaînes de bits en paquets d'égales longueurs et on les additionne. Pour éviter des débordements il arrive de remplacer l'addition par un XOR :

h("ET")=00101 XOR 10100 = 10001 = (17)10, h("IL")= 01001 XOR 01100 = 00101 = (5)10

Pour éviter de hacher de la même façon toutes les permutations d'un même mot, ici h("IL")=h("LI"), on effectue par exemple des décalages circulaires (rotations). Par exemple : en décalant de 1 rang le premier paquet, de deux le deuxième, etc.

Division

On calcule le reste de la division par un entier n (un nombre d'entrée dans le tableau) de la codification binaire :

Exemple avec n=37 : "ET" = 00101 10100 = (180)10, h("ET") = 180 mod 37 = 32

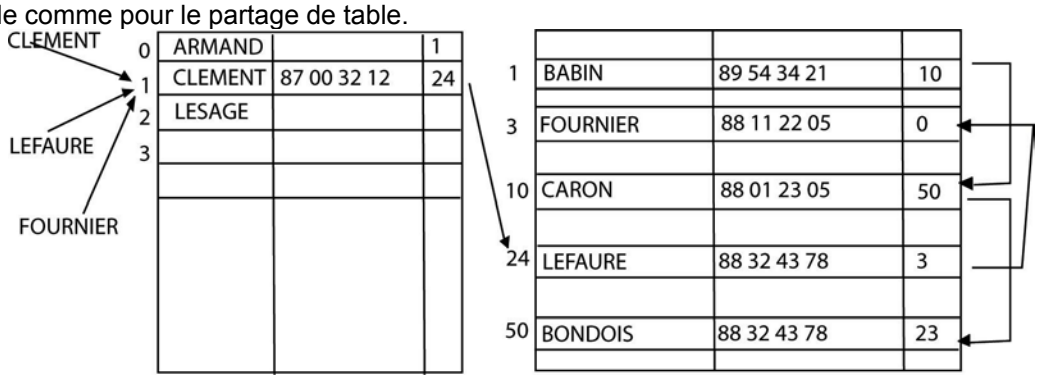
Malheureusement ce type de fonction dépend fortement de n. Si n est pair, alors tous les éléments pairs vont donner des indices pairs. Si en entrée il y plus de pairs que d'impairs cela produit des accumulations. On est alors amené à choisir n premier.

multiplication
Etant donné un nombre réel q, $0 < q < 1$, on pose : $h(i)$ = partie entière de (partie décimale de $(i*q)*n$), avec q = 0.6125423371 et n = 30.
h("ET") = partie entière de (partie décimale de (180*0.6125...)*30) = partie entière de (partie décimale de (110.2576...) * 30) = partie entière de (0.2576... * 30) = partie entière de (7.7...) = 7
On remarque que statistiquement on obtient les meilleurs résultats pour q = (√5-1)/2 ou 1-(√5-1)/2.

Les fonctions de hachage

sans utiliser de table mineure
Lorsqu'un élément j est en collision avec i, c'est-à-dire $h(i) = h(j)$, on essaye de placer j en $h(j)+1$ (à la case suivante). Si celle-ci est également occupée, la suivante etc., jusqu'à trouver un emplacement libre. A présent lorsque l'on recherche un élément x, ce dernier ne se trouve pas nécessairement en $h(x)$. Il peut se trouver plus bas. On cherche donc séquentiellement jusqu'à tomber sur x, ou sur une case vide. Un problème se pose lorsque l'on veut effacer un élément. En effet, marquer simplement la case vide coupe éventuellement "le pont" pour l'accès à d'autres données. Pour cela on utilise un marqueur spécial, de case libérée.

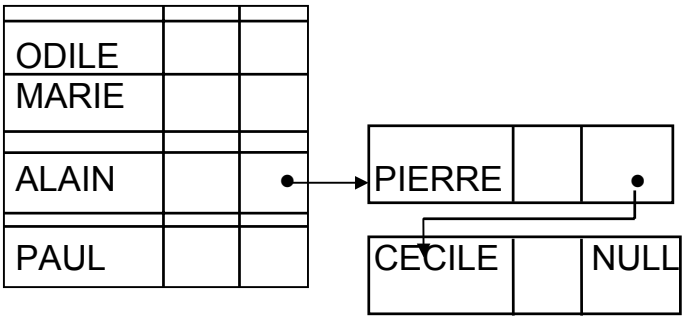
en utilisant une table mineure
On procède comme pour le partage de table.



Exemple de partage de table pour la résolution de collisions.

Complément : résolution des conflits de tables de hachages

Par chaînage séparé



Par chaînage avec collision secondaire

	.....		
1	ODILE		-1
2	MARIE		-1
3	.....		-1
4	ALAIN		7
5	.....		-1
6	PAUL		-1
7	PIERRE		8
8	CECILE		-1

Fichiers et entrées/sorties

Fichiers physiques, fichiers logiques

Les fichiers sont des entités dans lesquelles on regroupe des informations de manière durable, même si on éteint l'ordinateur. Cela peut être un texte, des données, un programme, etc. Par exemple, les données concernant les étudiants inscrits dans notre Université sont écrites sur disque, ce qui évite de tout retaper lors de la ré-inscription d'un étudiant. Les données sur les clients d'une banque (soldes, mouvements) ne peuvent pas toutes être simultanément en mémoire, il est donc nécessaire d'en avoir une partie sur disque, que l'on pourra charger en mémoire en cas de besoin. pas re-saisies à chaque inscription. Un programme est tapé dans un fichier, ce qui évite de le refaire à chaque utilisation. Ce cours a été frappé et sauvé dans un fichier, ce qui permet de le faire évoluer chaque année, sans avoir à tout retaper.

Assignation

On peut manipuler les fichiers par programme. En fait on ne manipule pas directement les fichiers mais des variables de type *fichier* (on appelle ça des fichiers logiques, par opposition au fichier physique, sur le disque dur). Ces variables pourront alors désigner n'importe quel fichier sur disque. L'opération qui associe à une telle variable un fichier physique s'appelle l'assignation. Cela ressemble à l'affectation, sauf qu'ici la variable ne prend pas comme valeur le nom du fichier (paramètre de l'assignation), mais le fichier sur le disque. Dans ce chapitre nous ne parlons que des fichiers à accès séquentiel, présentant une information par ligne (on par aussi de fichiers structurés en ligne). Lorsque nous abordons la traduction en C, nous présentons aussi les fichiers binaires.

```
variables
        Chaîne nomDuFichier;
        Fichier f;
début
        …
        f:=assigner(nomDuFichier);
        …
```

Ouverture

Pour lire des données dans un fichier existant, il faut l'ouvrir en lecture. Au contraire, pour écrire des données dans un fichier, il faut qu'il ait été ouvert en écriture. On réalise ces opérations ainsi :

```
ouvrirLecture(f);
ouvrirEcriture(f);
```

En C, l'assignation et l'ouverture d'un fichier sont réalisées en un seule opération :

```
FILE *f;
f = fopen(nomfichier, "r") ; // Ouverture en lecture
f = fopen(nomfichier, "w") ; // Ouverture en écriture
```

Dans le cas d'une ouverture en écriture, si le fichier existe alors son contenu est perdu.
On peut ajouter `b` au mode d'ouverture, pour préciser qu'il s'agit de données binaires, et éviter les problèmes dus à la différence de codage du caractère retour-chariot (`'\n'`) sur certains systèmes (MSDOS).

- `"rb"` = lire des données binaires.
- `"wb"` = écrire des données binaires.
- Il existe d'autres modes d'ouverture :
- `"a"` = ouverture en allongement.
- `"a+"` = ouverture en allongement, permet aussi la lecture.
- `"r+"` = mise à jour : permet de lire et écrire dans un fichier pour le modifier mais on ne peut pas en augmenter la taille.
- `"w+"` = ouverture en écriture, mais permet aussi la lecture.

`fopen` rend `NULL` en cas d'échec (fichier inexistant, pas de droit d'accès en écriture, etc.)

```
FILE *f ;
char *monfichier="donnees.txt";
f = fopen(nomfichier, "r");
if (f == NULL){
      fprintf(stderr,"erreur ouverture %s en lecture\n", nomfichier) ;
      exit(1);
}
```

On peut écrire une macro pour avoir ce message d'erreur (et l'arrêt du programme) quand il y a échec lors de l'ouverture d'un fichier quelconque :

```
#define FOPEN(f,nf,mode) { f=fopen(nf,mode); \
                  if(f == NULL) { \
                              fprintf(stderr, "erreur ouverture %s\n",nf) ;\
                              exit(1);\
                  }}
```

Remarque : dans une macro tout doit être écrit sur la même ligne, sauf si on utilise un backslash (\ ). C'est lors de l'utilisation de cette macro que l'on spécifie le fichier et le mode, par exemple `FOPEN(f, nomfichier, "r");`

Entrées/Sorties formatées

Lecture

La lecture dans un fichier se fait selon un accès séquentiel : le premier ordre de lecture (*liref*) lit la première donnée présente, le deuxième la suivante et ainsi de suite. Cela implique que l'ordre des instructions de lecture doit respecter dans lequel sont rangées les données. Il y a principalement 2 façons d'organiser la lecture, selon que l'on connaît ou non à l'avance le nombre de données présentes.
Si on connaît le nombre de données, la lecture de toutes les données se fait au moyen d'un boucle *pour*. Dans l'exemple suivant, le nombre de données est la première donnée du fichier. Suivent ensuite autant de nombres entiers qu'indiqué en entête. Pour certaines applications, la taille du fichier est toujours la même, il est donc inutile de le préciser en préambule. Ci suit une fonction d'initialisation de tableau d'entiers à partir de données contenues dans un fichier. En paramètre de cette fonction, nous trouvons le nom du fichier ainsi que la taille maximale du tableau.

```
fonction lireTableauDansFichier(chaîne nomDuFichier; entier taillemax) : (tableau entier, entier)
variables
        Fichier f;
        entier nbDonnées, i, laDonnée;
        tableau entier[TAILLEMEAX] tab;
début
        f := assigner(nomDuFichier);
        ouvrirLecture(f);
        nbDonnées := liref(f);
        si nbDonnées > taillemax alors nbDonnées := taillemax;
        faire pour i depuis 1 jusque nbDonnées
                laDonnée := liref(f);
                tab[i-1] := laDonnée;
        fait
        fermer(f);
        lireTableauDansFichier := (tab, nbDonnées);
fin
```

Si au contraire on l'ignore, on va lire les données jusqu'à atteindre la fin du fichier :

```
fonction lireTableauDansFichier(chaîne nomDuFichier; entier taillemax) : (tableau entier, entier)
variables
        Fichier f;
        Chaîne nomDuFichier;
        entier nbDonnées init 0;
        tableau entier[TAILLEMEAX] tab;
début
        f := assigner(nomDuFichier);
        ouvrirLecture(f)
        tant que NON finDeFichier(f) faire
                tab[nbDonnées] := liref(f);
                nbDonnées := nbDonnées+1;
        fait
        fermer(f);
        lireTableauDansFichier := (tab, nbDonnées);
fin
```

Ecriture

Comme pour la lecture, les données sont écrites les unes après les autres dans le fichier, exactement comme cela se passe lorsque l'on affiche à l'écran.

```
procédure ecrireTableauDansFichier(chaîne nomDuFichier; tableau entier tab[taillemax]; entier taille)
variables
        Fichier f;
        entier nbDonnées;
début
        f := assigner(nomDuFichier);
        ouvrirEcriture(f);
        ecriref(f, taille);
        faire pour i depuis 1 jusque nbDonnées
                ecriref(f,tab[i]);
        fait
        fermer(f);
fin
```

Lecture et écriture formatées en C

Il existe 3 fichiers prédéfinis et déjà ouverts, `stdin` le fichier d'entrée standard, `stdout` le fichier de sortie standard et `stderr` le fichier d'erreur standard. Ce deuxième fichier de sortie permet de séparer les messages qui sont des résultats de ceux qui sont là pour informer l'utilisateur, ou l'avertir d'erreurs. Nous avons déjà utilisé `stdout` et `stdin` implicitement (sauf pour `stdin` que nous avons utilisé explicitement avec `feof(stdin)`) :

`scanf("%d",&x);` est identique à `fscanf(stdin,"%d",&x);`

`printf("%d",x);` est identique à `fprintf(stdout,"%d",x);`

Ainsi, pour lire un entier dans un fichier Ascii `f` ouvert par `fopen`, on utilise : `fscanf(f,"%d",x);`

Pour écrire un entier (sous forme de texte) dans un fichier `f` ouvert par `fopen`, on écrira : `fprintf(f,"%d",x);`

Fermeture d'un fichier

Cette opération est nécessaire avant la fin d'un programme. Une fois fermé, le fichier logique peut être assigné à un autre fichier physique, et donc être ré-ouvert.

```
fermer(f);
```

en C :

```
fclose(f);
```

Fichiers binaires

Lecture et écriture binaires en C

En C, il est possible de manipuler directement les données, sans passer par la conversion en mode texte. Lorsque les données sont très volumineuses, les entrées et sorties deviennent alors beaucoup plus rapides.

Pour recopier tel quel (en binaire) dans un fichier : `fwrite`

Syntaxe : `fwrite((char *) &x, sizeof(int), 1, f);`

Les paramètres de `fwrite` sont un pointeur sur un caractère (si on a un entier ou un autre type, on force la conversion), la taille d'un élément, le nombre d'éléments et le nom du fichier dans lequel on écrit.

```
int tabint[100] ;
fwrite((char *) tabint, sizeof(int), 100, f) ;
```

Pour relire le tableau recopié tel quel : `fread`

```
fread((char *) tabint, sizeof(int), 100, f);
```

Il faut avoir bien auparavant déclaré ou alloué un tableau suffisamment grand.

Il existe d'autres fonctions sur les fichiers, parmi lesquelles :

`fseek` qui permet de se placer directement à l'endroit à modifier ou à lire. Pour se placer sur le 100 ème entier du fichier :

```
fseek(f, (100 - 1) * sizeof(int), SEEK_SET) ;
```

`ftell` permet de savoir à quel endroit on se trouve dans le fichier : `ftell(f) ;`

Application : sauvegarde d'un tableau

Cet exemple remplit un tableau et le copie en binaire dans un fichier avec en plus en entête sa taille en clair.

```
#include <stdio.h>
#include <stdin.h>
#define N 1000
main()
{
      int tabint[N] ;
      int nbval ;
      FILE *f ;
      char nomfichier[20] ;
      nbval = initTableau(tabint, N) ;
      fprintf(stdout, "Entrez un nom de fichier de données binaires : ") ;
      fscanf(stdin, "%s", nomfichier) ;
      f = fopen(nomfichier, "w") ;
      fprintf(f, "%d\n", nbval) ;
      fwrite((char *) tabint, sizeof(int), nbval, f) ;
      fclose(f) ;
}
```

Pour lire ce fichier (qui est illisible avec un éditeur de texte), et récupérer la taille en clair :

```
#include <stdio.h>
#define N 1000
main()
{
      int tabint[N] ;
      int nbval ;
      FILE *f ;
      char nomfichier[20] ;
      fprintf(stdout, "Entrez un nom de fichier : ") ;
      fscanf(stdin, "%s", nomfichier) ;
      f = fopen(nomfichier, "r") ;
      fscanf(f, "%d", &nbval) ;
      fread((char *) tabint, sizeof(int), nbval, f) ;
      fclose(f) ;
}
```

Une autre méthode est de lire entièrement la première ligne du fichier, puis de lire dans cette ligne comme dans un fichier.

```
#include <stdio.h>
#define N 1000
#define tailleligne 100
main()
{
      int tabint[N] ;
      int nbval ;
      FILE *f ;
      char nomfichier[20] ;
      char ligne[tailleligne] ;
      fprintf(stdout, "Entrez un nom de fichier : ") ;
      fscanf(stdin, "%s", nomfichier) ;
      f = fopen(nomfichier, "r") ;
      fgets(ligne, tailleligne, f) ; /* lit toute une ligne */
      sscanf(ligne, "%d", &nbval) ; /* lecture dans une chaîne */
      fread((char *) tabint, sizeof(int), nbval, f) ;
      fclose(f) ;
}
```

Introduction à la cinématique des fichiers

Les listes chaînées

Généralités

Les listes offrent un moyen de représenter une collection d'informations de même type avec un certain nombre d'avantages sur les tableaux : leur gestion est plus simple, en particulier lorsque le nombre d'éléments n'est pas connu à l'avance et connaît de nombreuses évolutions pendant la durée de l'exécution du programme, du fait que ces informations ne sont pas nécessairement contiguës en mémoire, au contraire des tableaux. Une liste est une suite d'éléments du même type, et est définie de manière récursive en distinguant les deux cas :

- la liste est vide, ne contient aucun élément ;
- la liste n'est pas vide, elle est formée de 2 composantes :
  - la première composante est le premier élément de la liste ;
  - la deuxième composante est la liste formée de tous les éléments suivant le premier.

D'un point de vue symbolique, on peut représenter la liste comme un couple, la liste vide étant simplement notée par des parenthèses vides : **()**. Toute liste contenant un élément prendra comme forme **(a,())**, une liste à deux éléments ressemblera à **(a,(b,()))** et ainsi de suite. La plupart des algorithmes de manipulation de liste pourront s'exprimer naturellement sous forme récursive, en différenciant les 2 cas :

- si la liste est vide, on n'effectue aucun traitement ;
- si elle n'est pas vide, elle possède 2 composantes, la première étant le premier élément. On applique alors le traitement en question à ce premier élément, et on appelle de manière récursive l'exécution de l'algorithme sur la sous-liste contenue dans la deuxième composante.

Le type liste

Lorsque le type liste est supporté par un langage de programmation, un certain nombre de fonctions élémentaires sont disponibles qui permettent non seulement d'accéder facilement à l'une ou l'autre des composantes, mais aussi de construire une liste, à partir d'un élément et d'une liste. Hélas, le type liste n'existe pas en langage C. On imagine toutefois qu'à l'aide du constructeur de structures, il serait possible de le définir. Il faut toutefois faire très attention, car en C définir un type n'est possible que si on est capable de calculer la taille des informations qu'il décrit. Or on ne connaît pas la taille d'une liste, puisqu'elle dépend de la taille de sa deuxième composante, elle même une liste. C'est pourquoi en C on ne peut définir une structure à l'aide du type qu'on est en train de définir, enfin pas aussi brutalement, sans quoi la structure ainsi décrite aurait une taille infinie. Même en usant du constructeur `union` pour prévoir le cas de la liste vide, on ne résout pas le problème, car le compilateur réserve l'espace mémoire nécessaire pour stocker le plus grand des champs de l'union, donc ne prend jamais en compte le cas de la liste vide. La solution consiste à modéliser la deuxième composante de la liste par un pointeur sur une liste. On résout ainsi le problème de la taille : que cette composante soit vide ou non, elle est toujours décrite par un pointeur, qui s'il est différent de `NULL` référence une liste non vide. On définit alors une liste comme une structure à deux composantes, l'une étant l'information proprement dite, l'autre un pointeur vers la liste suivante, donc l'élément suivant. En fait il est préférable de ne pas embarquer l'information pertinente directement dans la structure, mais plutôt de se contenter d'un pointeur sur l'information, ce qui permet de généraliser plus facilement l'utilisation des listes. La structure suivante décrit une **Cellule** comme un constituant de base de la liste, qui possède 2 champs, l'information utile (ou mieux un pointeur vers cette information) et un pointeur sur la liste suivante. Comme on définit le type `Liste` comme un pointeur sur une telle structure, le champ pointeur sur une cellule est donc bien une liste, comme tout pointeur sur une cellule. La liste vide est simplement réalisée à l'aide la valeur `NULL` : qui ne pointe sur aucune cellule.

```
typedef struct t_liste {
      Info element;
      struct t_liste *suivant;
} Cellule, *Liste;
```

à laquelle on préférera

```
typedef struct t_liste {
      Info *element;
      struct t_liste *suivant;
} Cellule, *Liste;
```

où Info est le type des informations utiles contenues dans la liste.

Opérations élémentaires sur les listes

Les opérations déclaration, test de vacuité, construction, extraction de composantes sont possibles dans de nombreux langages. À partir de ces quelques opérations essentielles, on peut réaliser toutes les opérations possibles sur les listes. Comme en C ces fonctionnalités de base n'existent pas, nous allons les réaliser.

Déclaration et initialisation

On la réalise à l'aide de l'instruction `Liste l = initListeVide() ;` ou `Liste l = uneListe ;`. Attention, dans ce dernier cas, `l` est une liste initialisée à la valeur de la liste `uneListe`. Comme il s'agit de pointeurs, `l` est une copie de l'adresse du premier élément de `uneListe`, et non pas une copie de la liste entière. En conséquence, modifier la liste `l` va entraîner la même modification sur `uneListe`. Attention aux effets de bord !

```
/* ------------------------------------------------------------------------ */
/* Initialisation d'une liste à Vide  */
/* Entrée : Néant, Modifiable : Néant  */
/* Sortie : Liste vide  */
/* Globales lues : Néant, Globales modifiées : Néant  */
/* ------------------------------------------------------------------------ */
Liste initListeVide(void)
{
      return NULL;
}
```

Test de vacuité

Une liste est vide si elle vaut `NULL` :

```
/* ------------------------------------------------------------------------ */
/* Teste si une liste est vide  */
/* Entrée : une liste l  */
/* Modifiable : Néant  */
/* Sortie : entier, nul si la liste n'est pas vide  */
/* Globales lues : Néant, Globales modifiées : Néant  */
/* ------------------------------------------------------------------------ */
int listeVide(Liste l)
{
      return (l==NULL);
}
```

Accès à la première information de la liste : la tête

La fonction suivante délivre l'adresse de la première cellule de la liste. Cette fonction ne doit pas être utilisée avec une liste vide. Il reste ensuite à accéder au champ Info de la cellule, et aux informations qu'il contient.

```
/* ------------------------------------------------------------------------ */
/* Délivre la tête de la liste  */
/* Entrée : une liste l  */
/* Modifiable : Néant  */
/* Sortie : la première cellule (pointeur)  */
/* Globales lues : Néant, Globales modifiées : Néant  */
/* ------------------------------------------------------------------------ */
Cellule *tete(Liste l)
{
      if (listeVide(l)) sortirSurErreur("Tete");
      return l;
}
```

Accès à la deuxième composante : la queue

Ceci n'est possible que si la liste n'est pas vide.

```
/* ------------------------------------------------------------------------ */
/* Délivre la queue de la liste  */
/* Entrée : une liste l  */
/* Modifiable : Néant  */
/* Sortie : la queue de la liste  */
/* Globales lues : Néant, Globales modifiées : Néant  */
```

```
/* ------------------------------------------------------------------------ */
Liste queue(Liste l)
{
      if (listeVide(l)) sortirSurErreur("Queue");
      return l->suivant;
}
```

Ajout d'un élément en tête

L'ajout d'un élément en tête est l'opération essentielle sur les listes, est appelée aussi **constructeur**, délivre une liste, à partir d'un élément et d'une liste. Il suffit de faire pointer le champ suivant du nouvel élément de tête vers l'ancienne liste, qui devient ainsi la queue de la nouvelle.

```
/* ------------------------------------------------------------------------ */
/* Ajout d'une cellule en tête d'une liste  */
/* Entrée : pointeur sur cellule, liste   */
/* Modifiable : Néant  */
/* Sortie : la nouvelle liste obtenue par l'ajout en tête  */
/* Globales lues : Néant, Globales modifiées : Néant  */
/* ------------------------------------------------------------------------ */
Liste constructeur(Cellule *c, Liste l)
{
      if (c != NULL)
            c->suivant = l;
      return c;
}
```

Suppression de l'élément en tête

Une opération importante est la suppression d'informations pouvant être contenues dans une liste. En théorie, cette opération n'est pas indispensable, puisqu'il suffit d'accéder à la queue de la liste pour éliminer *par oubli* la tête : `l = queue (l) ;`

En C, il n'existe pas de ramasse-miettes donc la mémoire occupée par la cellule éliminée, bien qu'elle ne soit plus référencée, n'est pas automatiquement libérée. Pour la rendre réutilisable, il faut libérer (on dit aussi parfois dés-allouer) explicitement cette mémoire. Vous noterez que pour assurer la généralité de ce traitement, la fonction de dés-allocation de l'information utile est passée en paramètre de la fonction de suppression de tête. Évidemment, la suppression du premier élément de la liste n'est possible que si la liste n'est pas vide. Auquel cas, la liste obtenue est celle désignée par le champ `suivant` de la tête de la liste.

```
/* ------------------------------------------------------------------------ */
/* Suppression de la première cellule d'une liste  */
/* Entrée : liste, fonction de destruction de cellule  */
/* Modifiable : Néant  */
/* Sortie : la nouvelle liste obtenue par suppression de la tête  */
/* Globales lues : Néant, Globales modifiées : Néant  */
/* ------------------------------------------------------------------------ */
Liste supprimeTete(Liste l, void(*lc)(Cellule *))
{
      Cellule *t;

      if (!listeVide(l)) {
            t = l;
            l = l->suivant;
            lc(t);
      }
      return l;
}
```

Un exemple de fonction de destruction de cellule correspondant au deuxième type de liste que nous avons défini est donné :

```
/* ------------------------------------------------------------------------ */
/* Fonction de destruction de cellule  */
/* Entrée : pointeur sur une cellule  */
/* Modifiable : Néant  */
/* Sortie : Néant  */
/* Globales lues : Néant, Globales modifiées : Néant  */
/* ------------------------------------------------------------------------ */
```

```
void libereCellule(Cellule *c)
{
      free(c->info);
      free(c);
}
```

Autres opérations utiles sur les listes : notion de type abstrait

Les fonctions suivantes ne nécessitent pas de connaître la structure de données interne de la liste, il suffit de les manipuler à l'aide des fonctions définies ci-dessus. Elles sont données directement en langage C.

Longueur d'une liste

Le nombre d'éléments d'une liste (appelé aussi taille ou longueur) est obtenu par l'algorithme : si la liste est vide, sa longueur est nulle, sinon elle contient au moins un élément. On peut même dire que sa longueur est 1 (on comptabilise le premier élément) plus la longueur de la deuxième composante (la liste des autres éléments).

```
/* ------------------------------------------------------------------------ */
/* Calcul de la longueur d'une liste (nombre d'éléments)  */
/* Entrée : liste  */
/* Modifiable : Néant  */
/* Sortie : nombre entier d'éléments  */
/* Globales lues : Néant, Globales modifiées : Néant  */
/* ------------------------------------------------------------------------ */
int longueurListe(Liste l)
{
      int lg;

      if (listeVide(l)) lg = 0;
      else lg = 1 + longueurListe(queue(l));

      return lg;
}
```

Ajout en queue

Grâce à la fonction constructeur, il est possible de faire d'autres fonctions d'ajout d'éléments dans des listes. En particulier, l'ajout en queue de liste est réalisé en utilisant la stratégie récursive suivante : Si la liste est vide, l'ajout en queue n'est autre qu'un ajout en tête. Sinon on cherchera à réaliser l'ajout de queue de la deuxième composante de la liste en entrée. Attention à ne pas casser la liste, c'est-à-dire rompre le chaînage entre la tête et le corps de la liste :

```
/* ------------------------------------------------------------------------ */
/* Ajout d'une cellule en queue de liste  */
/* Entrée : pointeur sur cellule, liste  */
/* Modifiable : Néant  */
/* Sortie : la nouvelle liste après ajout  */
/* Globales lues : Néant, Globales modifiées : Néant  */
/* ------------------------------------------------------------------------ */
Liste ajoutQueue(Cellule *c, Liste l)
{
      Liste l2;

      if (listeVide(l))
            l2 = constructeur(c, l);
      else
            l2 = constructeur(tete(l), ajoutQueue(c, queue(l)));

      return l2;
}
```

Suppression du dernier élément

La suppression du dernier élément de la liste équivaut à la suppression de la tête, en présence d'un élément unique. On doit alors vérifier si on ce trouve dans ce cas. Dans le cas contraire, on tentera d'ôter le dernier élément de la deuxième composante de la liste.

```
/* ------------------------------------------------------------------------ */
/* Suppression de la dernière cellule d'une liste  */
/* Entrée : liste, fonction de destruction de cellule  */
/* Modifiable : Néant  */
/* Sortie : la nouvelle liste obtenue par suppression   */
/* Globales lues : Néant, Globales modifiées : Néant  */
/* ------------------------------------------------------------------------ */
Liste supprimeQueue(Liste l, void(*lc)(Cellule *))
{
      Liste l2;
      /* si la liste est vide on ne supprime rien */
      if (!listeVide(l)) {
            /* si la liste n'a qu'un seul élément : idem supprimeTete */
            if (listeVide(queue(l)))
                  l2 = supprimeTete(l, lc);
            else
                  l2 = constructeur(tete(l), supprimeQueue(queue(l), lc));
      }
      else l2 = initListeVide();
      return l2;
}
```

Insertion en une position donnée

Pour insérer un élément à une position donnée, on fait là encore appel au constructeur de liste. Si on n'est pas dans le cas idéal d'ajout en position initiale, on tente de s'y ramener en invoquant l'appel de la fonction appliquée à la deuxième composante de la liste, et en faisant diminuer en même temps la position : insérer en position n d'une liste, c'est insérer en position n-1 de sa sous-liste. Dans cette autre version, nous avons rendu possible l'insertion, même si la position préconisée n'est pas valide : en tête si n<1, en queue si n dépasse la taille de la liste.

```
/* ------------------------------------------------------------------------ */
/* Ajout (si possible) d'un élément à une position donnée  */
/* Entrée : liste, pointeur sur cellule, position (entière)  */
/* Modifiable : Néant  */
/* Sortie : la nouvelle liste obtenue par insertion   */
/* Globales lues : Néant, Globales modifiées : Néant  */
/* ------------------------------------------------------------------------ */
Liste ajoutPosition(Liste l, Cellule *c, int n)
{
      Liste l2;
      if (n<1) l2 = l;
      else if (n==1) l2 = constructeur(c, l);
      else if (!listeVide(l))
            l2 = constructeur(tete(l), ajoutPosition(queue(l), c, n-1));
      else l2 = initListeVide();
      return l2;
}
```

```
/* ------------------------------------------------------------------------ */
/* Ajout (toujours possible) d'un élément à une position donnée  */
/* Entrée : liste, pointeur sur cellule, position (entière)  */
/* Modifiable : Néant  */
/* Sortie : la nouvelle liste obtenue par insertion   */
/* Globales lues : Néant, Globales modifiées : Néant  */
/* ------------------------------------------------------------------------ */
Liste ajoutPositionV2(Liste l, Cellule *c, int n)
{
      Liste l2;
      if ( (n<1) || (listeVide(l)) ) l2 = constructeur(c, l);
      else
            l2 = constructeur(tete(l), ajoutPositionV2(queue(l), c, n-1));
      return l2;
}
```

Suppression d'un élément de position donnée

De la même manière, la suppression d'un élément de position donnée n'est simple que s'il s'agit du premier élément. Si tel n'est pas le cas, on tente alors d'opérer la suppression au sein de la deuxième composante, en ayant soin de faire varier correctement la valeur de la position concernée par la suppression.

```
/* ------------------------------------------------------------------------ */
/* Suppression (si possible) d'un élément à une position donnée  */
/* Entrée : liste, position (entière), fonction de destruction de cellule  */
/* Modifiable : Néant  */
/* Sortie : la nouvelle liste obtenue par suppression   */
/* Globales lues : Néant, Globales modifiées : Néant  */
/* ------------------------------------------------------------------------ */
Liste supprimePosition(Liste l, int n, void(*lc)(Cellule *c))
{
      Liste l2;
      if (n<1) l2 = l;
      else if (n==1) l2 = supprimeTete(l, lc);
      else if (!listeVide(l))
            l2 = constructeur(tete(l), supprimePosition(queue(l), n-1, lc));
      else l2 = initListeVide();
      return l2;
}
```

Insertion triée (ou sur critère quelconque)

L'ajout étant trivial dans le cas d'une liste vide, on peut se concentrer sur le cas des listes non vides : si l'ajout en tête permet de maintenir la relation d'ordre (décrite par une fonction passée en paramètre pour assurer un maximum de généralité), on le pratique, sinon on renouvelle la tentative d'insertion, mais cette fois dans la deuxième composante de la liste.

```
/* ------------------------------------------------------------------------ */
/* Insertion cellule dans une liste avec maintien d'une relation d'ordre */
/* Entrée : liste, pointeur sur cellule, relation d'ordre entre 2 cellules */
/* Modifiable : Néant */
/* Sortie : la nouvelle liste obtenue par insertion  */
/* Globales lues : Néant, Globales modifiées : Néant */
/* ------------------------------------------------------------------------ */
Liste ajoutTri(Liste l, Cellule *c, int(*oc)(Cellule *, Cellule *))
{
      Liste l2;
      if (listeVide(l)) l2 = constructeur(c, l);
      else {
            if ((!oc(c, tete(l)))
                  l2 = constructeur(tete(l), ajoutTri(queue(l), c, oc));
            else l2 = constructeur(c, l);
      }
      return l2;
}
```

Si on veut effectuer un tri croissant sur une liste d'entiers, on utilisera une relation d'ordre définie comme suit :

```
int ordreCellule(Cellule *a, Cellule *b)
{
      return ( (*(a->info)).v < (*(b->info)).v );
}
```

Exemple d'utilisation

Voici un petit programme qui illustre l'usage de fonctions de manipulation de listes, tout en étant aussi générique que possible. Nous définissons le type des éléments de la liste (dans le fichier infoT.h)

infoT.h

```
typedef struct t_info {
      int v;
} Info;
```

La définition de notre type Liste dans le fichier **listeT.h** reste inchangée, seul le type Info ayant besoin d'être modifié.

listeT.h

```
#include "infoT.h"

typedef struct t_cellule {
      Info *info;
      struct t_cellule *suivant;
} Cellule, *Liste;
```

Quant au fichier listeP.h, il contient les prototypes des fonctions de manipulation de liste. :

listeP.h

```
extern Liste initListeVide(void);
extern int listeVide(Liste l);
extern Liste queue(Liste l);
extern Liste constructeur(Cellule *c, Liste l);
extern Liste supprimeTete(Liste l, void(*lc)(Cellule *));
extern int longueurListe(Liste l);
extern Liste ajoutQueue(Cellule *c, Liste l);
extern Liste supprimeQueue(Liste l, void(*lc)(Cellule *));
extern Liste ajoutPosition(Liste l, Cellule *c, int n);
extern Liste ajoutPositionV2(Liste l, Cellule *c, int n);
extern Liste supprimePosition(Liste l, int n, void(*lc)(Cellule *c));
extern Liste ajoutTri(Liste l, Cellule *c, int(*oc)(Cellule *, Cellule *));
```

Voici enfin le petit applicatif qui manipule des listes :

appliListe.c

```
#include <stdio.h>
#include <stdlib.h>

#include "infoT.h"
#include "listeT.h"
#include "listeP.h"

Cellule * nouvelleCellule(int v)
{
      Cellule *c;

      c = (Cellule *)malloc(sizeof(Cellule));
      c->info = (Info *)malloc(sizeof(Info));
      (*(c->info)).v = v;
      return c;
}

int ordreCellule(Cellule *a, Cellule *b)
{
      return ( (*(a->info)).v < (*(b->info)).v );
}

void libereCellule(Cellule *c)
{
      free(c->info); free(c);
}

void afficheListe(Liste l)
{
      if (!listeVide(l)) {
            fprintf(stderr,"%d ",(*(l->info)).v);
            afficheListe(queue(l));
      }
      else fprintf(stderr,"\n");
}
```

```
/* PROGRAMME PRINCIPAL */
/* ################### */

int main(int argc, char **argv)
{
      Liste l;
      Cellule *c;

      l = initListeVide();
      c = nouvelleCellule(4);
      l = ajoutTri(l, c, ordreCellule);
      c = nouvelleCellule(1);
      l = ajoutTri(l, c, ordreCellule);
      c = nouvelleCellule(2);
      l = ajoutTri(l, c, ordreCellule);
      l = supprimeTete(l, libereCellule);
      c = nouvelleCellule(3);
      l = ajoutTri(l, c, ordreCellule);
      c = nouvelleCellule(5);
      l = ajoutPosition(l, c, 5);
      afficheListe(l);
}
```

Structures récursives et croisées

Chaînages multiples

Les piles

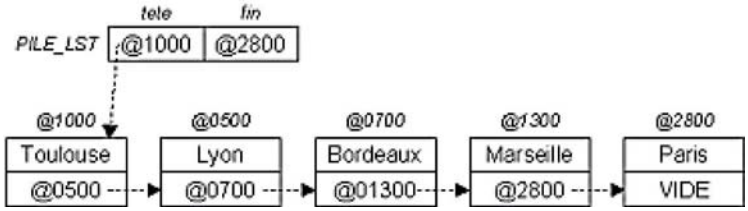
Modélisation de la structure

Introduction

Une pile est une structure qui stocke de manière ordonnée des éléments, mais rend accessible uniquement un seul d'entre eux, appelé le sommet de la pile. Quand on ajoute un élément, celui-ci devient le sommet de la pile, c'est-à-dire le seul élément accessible. Quant on retire un élément de la pile, on retire toujours le sommet, et le dernier élément ajouté avant lui devient alors le sommet de la pile. Pour résumer, le dernier élément ajouté dans la pile est le premier élément à en être retiré. Cette structure est également appelée une liste LIFO (*Last In, First Out*).
Généralement, il y a deux façons de représenter une pile. La première s'appuie sur la structure de liste chaînée vue précédemment. La seconde manière utilise un tableau.

Modélisation par liste chaînée

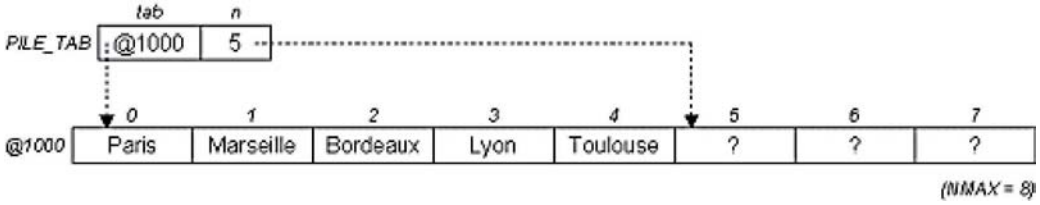
La première façon de modéliser une pile consiste à utiliser une liste chaînée en n'utilisant que les opérations *ajouterTete* et *retirerTete*. Dans ce cas, on s'aperçoit que le dernier élément entré est toujours le premier élément sorti. La figure suivante représente une pile par cette modélisation. La pile contient les chaînes de caractères suivantes qui ont été ajoutées dans cet ordre: "Paris", "Marseille", "Bordeaux", "Lyon" et "Toulouse".



Pour cette modélisation, la structure d'une pile est celle d'une liste chaînée.

Modélisation par tableau

La deuxième manière de modéliser une pile consiste à utiliser un tableau. L'ajout d'un élément se fera à la suite du dernier élément du tableau. Le retrait d'un élément de la pile se fera en enlevant le dernier élément du tableau. La figure suivante représente une pile par cette modélisation. Elle reprend la même pile que pour l'exemple de modélisation par liste chaînée.



Voici la structure de données correspondant à une pile représentée par un tableau.

```
typedef struct {
      ELEMENT tab[NMAX];
      ENTIER n;
} PILE_TAB ;
```

NMAX est une constante représentant la taille du tableau alloué. *n* est le nombre d'éléments dans la pile.

Opérations sur la structure

Introduction

A partir de maintenant, nous allons employer le type PILE qui représente une pile au sens général, c'est-à-dire sans se soucier de sa modélisation. PILE représente aussi bien une pile par liste chaînée (PILE_LST) qu'une liste par pointeurs (PILE_PTR). Voici les opérations que nous allons détailler pour ces deux modélisations.

- *fonction initialiserPile() : PILE*

Initialise une pile vide.

- *fonction pileVide(PILE p) : BOOLEEN*

Indique si la pile *p* est vide.

- *fonction sommet(PILE p) : ELEMENT*

Retourne l'élément au sommet de la pile *p*

- *fonction empilerElement(PILE p, ELEMENT e) : PILE*

Empile l'élément *e* au sommet de la pile *p*.

- *fonction depilerElement(PILE p) : (PILE, ELEMENT)*

Dépile et délivre l'élément au sommet de la pile *p*. La nouvelle pile est aussi délivrée.

Les prototypes de ces opérations (paramètres et type de retour) sont les mêmes quelque soit la modélisation choisie.

Opérations pour la modélisation par liste chaînée

Initialiser la pile

Cette fonction initialise les valeurs de la structure représentant la pile, afin que celle-ci soit vide. Dans le cas d'une représentation par liste chaînée, il suffit d'initialiser la liste chaînée qui représente la pile.

```
fonction initialiserPile():PILE
début
        initialiserPile := initialiserListe();
fin
```

Pile vide ?

Cette fonction indique si la pile *p* est vide. Dans le cas d'une représentation par liste chaînée, la pile est vide si la liste qui la représente est vide.

```
fonction pileVide(PILE p): BOOLEEN
début
        pileVide := listeVide(p);
fin
```

Sommet d'une pile

Cette fonction retourne l'élément au sommet de la pile *p*. Dans le cas d'une représentation par liste chaînée, cela revient à retourner la valeur de l'élément en tête de la liste. A n'utiliser que si la pile *p* n'est pas vide.

```
fonction sommet(PILE p): ELEMENT
variables
        ELEMENT res init NIL;
début
        si (non pileVide(p)) alors res := teteListe(p); finsi
        sommet := res ;
fin
```

Empiler un élément sur une pile

Cette fonction empile l'élément *e* au sommet de la pile *p*. Pour la représentation par liste chaînée, cela revient à ajouter l'élément *e* en tête de la liste.

```
fonction empilerElement(PILE p, ELEMENT e): PILE
début
        empilerElement := ajouterTete(p,e);
fin
```

Dépiler un élément d'une pile

Cette fonction dépile l'élément au sommet de la pile *p* et stocke sa valeur dans *e*. Pour la représentation par liste chaînée, cela revient à récupérer (si possible) la valeur de l'élément en tête de liste avant de le supprimer de cette dernière.

```
fonction depilerElement(PILE p): (PILE, ELEMENT)
variables
        ELEMENT e init NIL;
début
        si (non pileVide(p)) alors e := sommet(p) ; p := retirerTete(p); finsi
        depilerElement := (p, e) ;
fin
```

Opérations pour la modélisation par tableau

Initialiser la pile

Cette fonction initialise les valeurs de la structure représentant une pile, afin que celle-ci soit vide. Dans le cas d'une représentation par tableau, il suffit de rendre *n* nul.

```
fonction initialiserPile():PILE
variables
        PILE p ;
début
        p .n := 0;
        initialiserPile := p ;
fin
```

Pile vide ?

Cette fonction indique si la pile *p* est vide. Dans le cas d'une représentation par tableau, la pile est vide si le champ *n* est nul.

```
fonction pileVide(PILE p): BOOLEEN
début
        pileVide := (p.n = 0);
fin
```

Sommet d'une pile

Cette fonction retourne l'élément au sommet de la pile *p*. Dans le cas d'une représentation par tableau, cela revient à retourner la valeur du *n*ième élément du tableau (i.e. l'élément d'indice *n* - 1). A n'utiliser que si la pile *p* n'est pas vide.

```
fonction sommet(PILE p): ELEMENT
variables
        ELEMENT e init NIL ;
début
        si (non pileVide(p)) alors e := (p.tab[p.n - 1]); finsi
        sommet := e ;
fin
```

Empiler un élément sur une pile

Cette fonction empile l'élément *e* au sommet de la pile pointée par *p*. Pour la représentation par tableau, cela revient à ajouter l'élément *e* à la fin du tableau. S'il reste de la place dans l'espace réservé au tableau, l'empilement peut avoir lieu.

```
fonction empilerElement(PILE p, ELEMENT e): PILE
variables
        PILE res init p ;
début
        si (res.n < NMAX) alors res.tab[res.n] :=e; res.n := res.n + 1; finsi
        empilerElement :=res ;
fin
```

```
fonction empilerElement(PILE p, ELEMENT e): PILE
début
        si (p.n < NMAX) alors p.tab[p.n] :=e; p.n := p.n + 1; finsi
        empilerElement :=res ;
fin
```

Dépiler un élément d'une pile

Cette fonction dépile l'élément au sommet de la pile *p* et stocke sa valeur dans *e*. Pour la représentation par tableau, cela revient à diminuer d'une unité le champ *n*, et à renvoyer l'élément d'indice *n* du tableau. Si la pile n'est pas déjà vide, on peut dépiler.

```
fonction depilerElement(PILE p) : (PILE, ELEMENT)
variables
        ELEMENT e init NIL;
```

```
début
        si (non (pileVide(p))) alors e := sommet(p);  p.n := p.n - 1; finsi
        depilerElement := (p, e) ;
fin
```

Conclusion

Comme la pile ne permet l'accès qu'à un seul de ses éléments, son usage est limité. Cependant, elle peut être très utile pour supprimer la récursivité d'une fonction. La différence entre la modélisation par liste chaînée et la modélisation par tableau est très faible. L'inconvénient du tableau est que sa taille est fixée à l'avance, contrairement à la liste chaînée qui n'est limitée que par la taille de la mémoire centrale de l'ordinateur. En contrepartie, la liste chaînée effectue une allocation dynamique de mémoire à chaque ajout d'élément et une libération de mémoire à chaque retrait du sommet de la pile. En résumé, la modélisation par liste chaînée sera un peu plus lente, mais plus souple quant à sa taille.

Les files

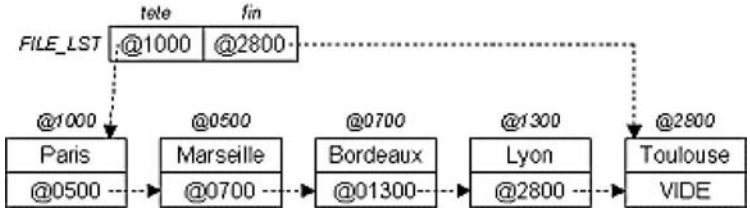
Modélisation de la structures

Introduction

Une file d'attente est une structure qui stocke de manière ordonnée des éléments, mais rend accessible uniquement un seul d'entre eux, appelé la tête de la file. Quant on ajoute un élément, celui-ci devient le dernier élément qui sera accessible. Quant on retire un élément de la file, on retire toujours la tête, celle-ci étant le premier élément qui a été placé dans la file.
Pour résumer, le premier élément ajouté dans la pile est le premier élément à en être retiré. Cette structure est également appelée une liste FIFO (*First In, First Out*). Généralement, il y a deux façons de représenter une file d'attente. La première s'appuie sur la structure de liste chaînée vue précédemment. La seconde manière utilise un tableau d'une façon assez particulière que l'on appelle modélisation par "tableau circulaire".

Modélisation par liste chaînée

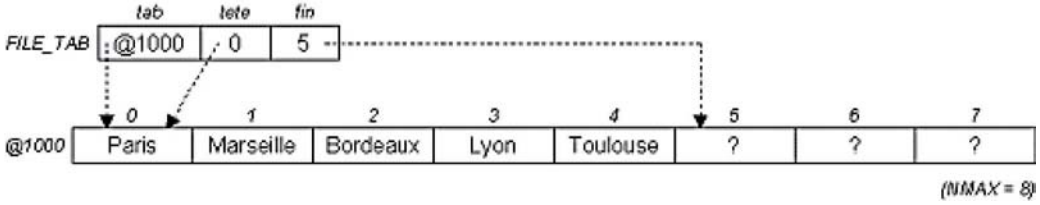
La première façon de modéliser une file d'attente consiste à utiliser une liste chaînée en n'utilisant que les opérations ajouterQueue et retirerTete. Dans ce cas, on s'aperçoit que le premier élément entré est toujours le premier élément sorti. La figure suivante représente une file d'attente par cette modélisation. La file contient les chaînes de caractères suivantes qui ont été ajoutées dans cet ordre: "Paris", "Marseille", "Bordeaux", "Lyon" et "Toulouse".



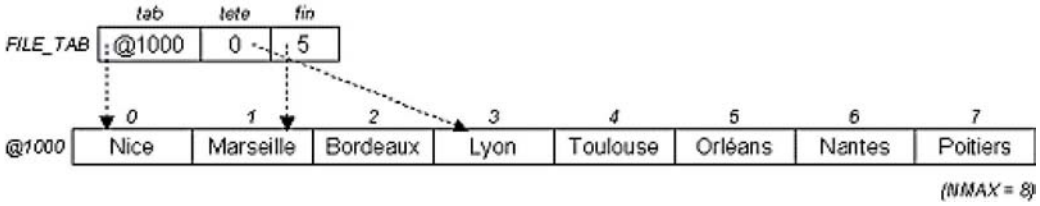
Pour cette modélisation, la structure d'une file d'attente est celle d'une liste chaînée (avec tête et queue).

Modélisation par tableau circulaire

La deuxième manière de modéliser une file d'attente consiste à utiliser un tableau. L'ajout d'un élément se fera à la suite du dernier élément du tableau. Le retrait d'un élément de la file se fera en enlevant le premier élément du tableau. Il faudra donc deux indices pour ce tableau, le premier qui indique le premier élément de la file et le deuxième qui indique la fin de la file. La figure suivante représente une file d'attente par cette modélisation. Elle reprend la même pile que pour l'exemple de modélisation par liste chaînée.



On peut noter que progressivement, au cours des opérations d'ajout et de retrait, le tableau se déplace sur la droite dans son espace mémoire. A un moment, il va en atteindre le bout de l'espace. Dans ce cas, le tableau continuera au début de l'espace mémoire comme si la première et la dernière case étaient adjacentes, d'où le terme "tableau circulaire". Ce mécanisme fonctionnera tant que le tableau n'est effectivement pas plein. Pour illustrer, reprenons l'exemple précédent auquel on applique trois opérations de retrait de l'élément de tête. On ajoute également en queue les éléments suivants et dans cet ordre: "Orléans", "Nantes", "Poitiers", "Nice".



La file d'attente contient alors, et dans cet ordre: "Lyon", "Toulouse", "Orléans", "Nantes", "Poitiers" et "Nice". On s'aperçoit que, ayant atteint la fin du tableau, la file redémarre à l'indice 0.
Voici la structure de données correspondant à une file d'attente représentée par un tableau circulaire.

```
typedef struct {
      ELEMENT tab[NMAX];
      ENTIER tete;
      ENTIER fin;
} FILE_TAB;
```

`NMAX` est une constante représentant la taille du tableau alloué. `tete` est l'indice de tableau qui pointe sur la tête de la file d'attente. fin est l'indice de tableau qui pointe sur la case suivant le dernier élément du tableau, c'est-à-dire la prochaine case libre du tableau.

Opérations sur la structure

Introduction

A partir de maintenant, nous allons employer le type FILE qui représente une file d'attente au sens général, c'est-à-dire sans se soucier de sa modélisation. FILE représente aussi bien une file d'attente par liste chaînée (FILE_LST) qu'une file d'attente par tableau circulaire (FILE_TAB). Voici les opérations que nous allons détailler pour ces deux modélisations.

- *fonction initialiserFile() : FILE*

Initialise une file vide.

- *fonction fileVide(FILE f) : BOOLEEN*

Indique si la file *f* est vide.

fonction teteFile(FILE f) : ELEMENT

Retourne l'élément en tête de la file *f*.

- *fonction entrerElement(FILE f, ELEMENT e) : FILE*

Entre l'élément *e* dans la file *f*.

- *fonction sortirElement(FILE f) : (FILE, ELEMENT)*

Sort l'élément en tête de la file *f*.

Les prototypes de ces opérations (paramètres et type de retour) sont les mêmes quelque soit la modélisation choisie.

Opérations pour la modélisation par liste chaînée

Initialiser une file

Cette fonction initialise les valeurs de la structure représentant la file *f* pour que celle-ci soit vide. Dans le cas d'une représentation par liste chaînée, il suffit d'initialiser la liste chaînée qui représente la file d'attente.

```
fonction initialiserFile():FILE
début
      initialiserFile := initialiserListe();
fin
```

File vide ?

Cette fonction indique si la file *f* est vide. Dans le cas d'une représentation par liste chaînée, la file est vide si la liste qui la représente est vide.

```
fonction fileVide(FILE f): BOOLEEN
début
      fileVide := listeVide(f);
fin
```

Tête d'une file

Cette fonction retourne l'élément en tête de la file *f*. Dans le cas d'une représentation par liste chaînée, cela revient à retourner la valeur de l'élément en tête de la liste. A n'utiliser que si la file *f* n'est pas vide.

```
fonction teteFile(FILE f) : ELEMENT
variables
      ELEMENT e init NIL;
début
      si (non (fileVide(f))) alors e := teteListe(f); finsi
      teteFile := e ;
fin
```

Entrer un élément dans une file

Cette fonction place un élément *e* en queue de la file *f*. Pour la représentation par liste chaînée, cela revient à ajouter l'élément *e* en queue de la liste.

```
fonction entrerElement(FILE  f, ELEMENT e): FILE
début
        entrerElement := ajouterQueue(f,e);
fin
```

Sortir un élément d'une file

Cette fonction retire l'élément en tête de la file pointée par *f* et stocke sa valeur dans *e*. Pour la représentation par liste chaînée, cela revient à récupérer la valeur de l'élément en tête de liste avant de le supprimer de cette dernière.

```
fonction sortirElement(FILE f) : (FILE, ELEMENT)
variables
        ELEMENT e init NIL ;
début
        si (non (fileVide(f))) alors e := teteFile(f); f := queue(f) ;finsi
        sortirElement := (f,e) ;
fin
```

Opérations pour la modélisation par tableau circulaire

Initialiser une file

Cette fonction initialise les valeurs de la structure représentant la file pointée par *f* pour que celle-ci soit vide. Dans le cas d'une représentation par tableau circulaire, on choisira de considérer la file vide lorsque *f.tete = f.fin*. Arbitrairement, on choisit ici de mettre ces deux indices à 0 pour initialiser une file d'attente vide.

```
fonction initialiserFile() : FILE
variables
        FILE f ;
début
        f.tete := 0;
        f.fin := 0;
        initialiserFile := f ;
fin
```

File vide ?

Cette fonction indique si la file *f* est vide. Dans le cas d'une représentation par tableau circulaire, la file est vide lorsque *f.tete = f.fin*.

```
fonction fileVide(FILE f): BOOLEEN
début
        fileVide := (f.tete = f.fin);
fin
```

Tête d'une file

Cette fonction retourne l'élément en tête de la file *f*. Dans le cas d'une représentation par tableau circulaire, il suffit de retourner l'élément pointé par l'indice de tête dans le tableau. A n'utiliser que si la file *f* n'est pas vide.

```
fonction teteFile(FILE f): ELEMENT
variables
        ELEMENT e init NIL;
début
        si (non fileVide(f)) alors e := (f.tab[f.tete]); finsi
        teteFile := e ;
fin
```

Entrer un élément dans une file

Cette fonction place un élément e en queue de la file *f*. Pour la représentation par tableau circulaire, l'élément est placé dans la case pointée par l'indice *fin*. Ce dernier est ensuite augmenté d'une unité, en tenant compte du fait qu'il faut revenir à la première case du tableau si il a atteint la fin de celui-ci. D'où

l'utilisation de l'opérateur *mod* qui retourne le reste de la division entre ses deux membres. Ceci ne peut être exécuté que si le tableau n'est pas plein au moment de l'insertion.

```
fonction entrerElement(FILE  f, ELEMENT e): FILE
début
        si ((f.fin + 1) mod NMAX  <>  f.tete) alors
                f.tab[res.fin] := e;
                f.fin := (f.fin + 1) mod NMAX;
        finsi
        entrerElement := f ;
fin
```

On détecte que le tableau est plein si l'indice *fin* est juste une case avant l'indice *tete*. En effet, si on ajoutait une case, fin deviendrait égal à *tete*, ce qui est notre configuration d'une file vide. La taille maximale de la file est donc *NMAX* - 1, une case du tableau restant toujours inutilisée.

Sortir un élément d'une file

Cette fonction retire l'élément en tête de la file *f* et stocke sa valeur dans e. Pour la représentation par tableau circulaire, cela revient à récupérer la valeur de l'élément en tête de file avant de le supprimer en augmentant l'indice *tete* d'une unité. Il ne faut pas oublier de ramener *tete* à la première case du tableau au cas où il a atteint la fin de ce dernier.

```
fonction sortirElement(FILE  f) : (FILE, ELEMENT)
variables
        ELEMENT e init NIL ;
début
        si (non(fileVide(f))) alors
                e := teteFile(f);
                f.tete := (f.tete + 1) mod NMAX;
        finsi
        sortirElement := (f, e) ;
fin
```

Conclusion

Ce genre de structure de données est très utilisée par les mécanismes d'attente. C'est le cas notamment d'une imprimante en réseau, où les tâches d'impressions arrivent aléatoirement de n'importe quel ordinateur connecté. Les tâches sont placées dans une file d'attente, ce qui permet de les traiter selon leur ordre d'arrivée. Les remarques concernant les différences entre la modélisation par liste chaînée et la modélisation par tableau circulaire sont les mêmes que pour la structure de file. Très peu de différences sont constatées. En résumé, la modélisation par liste chaînée sera un peu plus lente, mais plus souple quant à sa taille.

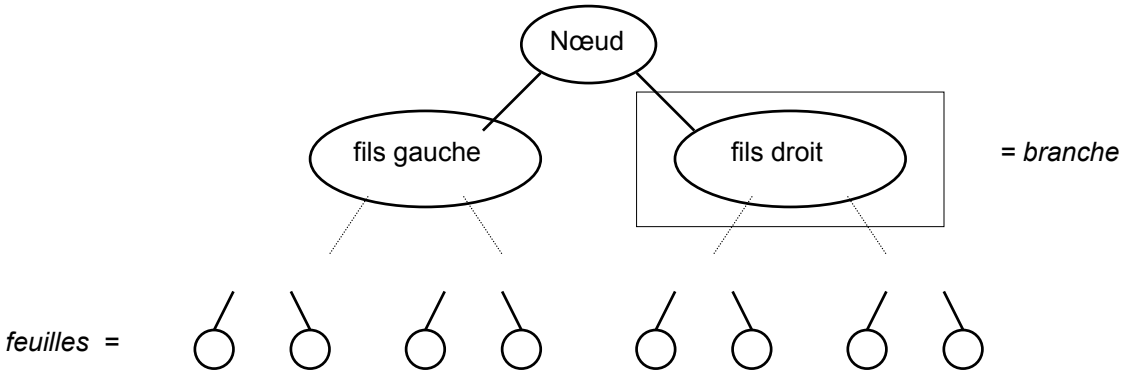
Les arbres

Généralités

Dans ce chapitre nous nous intéressons aux structures de données arborescentes. Ces structures dynamiques récursives constituent un outil important qui est couramment utilisé dans de nombreuses applications : codage de Huffman, interfaces, SGBD, expressions arithmétiques, intelligence artificielle, génomique, etc. Nous distinguerons dans un premier temps les arbres binaires, avant d'étendre et de généraliser cette structure, qu'on peut définir simplement comme une disposition hiérarchique de nœuds, dans laquelle le nœud situé au dessus de chaque fils est son père.

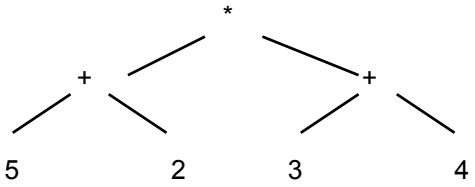
Le type Arbre Binaire

Dans un arbre binaire, chaque nœud a au plus 2 fils.



Les feuilles sont des nœuds sans fils.

Exemple : (5 + 2) (3 + 4)
* + 5 2 + 3 4



Dans certaines applications, seules les feuilles contiennent de l'information utile.

Opérations de base

Déclaration

ArbreBinaire Info a init ARBREVIDE ;

```
typedef struct t_arbre {
      Info *info;
      struct t_arbre *filsG, *filsD;
} *Arbre, Nœud;

Arbre a = NULL;
```

Test de vacuité

Fonction vide (Arbre a) : Booléen
début
vide := a = ARBREVIDE ;
fin

```
int vide(Arbre a)
{
	return a==NULL;
}
```

Accès au fils gauche

```
Arbre filsGauche(Arbre a)
{
	Arbre res = NULL;
	If ( !vide(a) ) res = a->filsG;
	return res;
}
```

Initialisation

info(a) := info; filsG(a) :=ARBREVIDE; filsD(a) := ARBREVIDE;

```
Nœud *nouveauNoeud(Info *info)
{
	Nœud *n = malloc(sizeof(Nœud));
	n->info = info;
	n->filsG = n->filsD = NULL;
	return n ;
}
```

Parcours préfixé

```
procédure parcours(ArbreBinaire Info a)
début
	si NON vide(a) alors
		traiter(info(a));
		parcours(filsG(a));
		parcours(filsD(a));
	finsi
fin
```

Parcours infixé

```
procédure parcours(ArbreBinaire Info a)
début
	si NON vide(a) alors
		parcours(filsG(a));
		traiter(info(a));
		parcours(filsD(a));
	finsi
fin
```

Parcours postfixé

```
procédure parcours(ArbreBinaire Info a)
début
	si NON vide(a) alors
		parcours(filsG(a));
		parcours(filsD(a));
		traiter(info(a));
	finsi
fin
```

Parcours par niveau

Ce type de parcours, en largeur d'abord, nécessite l'emploi de structures de données particulières, comme des piles ou des liens vers les frères, si l'on souhaite optimiser le parcours. En revanche, il est possible de le faire facilement, en multipliant les parcours, et en ne traitant que le niveau concerné à chaque passage.

Destruction d'un arbre

```
procédure detruireArbre(ArbreBinaire Info a)
début
        si NON vide(a) alors
                détruireArbre(filsG(a));
                détruireArbre(filsD(a));
                détruireNoeud(a);
        finsi
fin
```

Ajout d'un fils

À gauche

```
fonction insertionG(ArbreBinaire Info a, ArbreBinaire Info n) : ArbreBinaire
début
        si NON vide(a) filsG(a) := n;
        insertionG := a;
fin
```

Enracinement

A partir de 2 arbres et d'une valeur, on construit un nouvel arbre. Pour cela on doit créer un nouveau nœud.

```
fonction enracinement(Info v, ArbreBinaire Info a, ArbreBinaire Info b) : ArbreBinaire
variables
        ArbreBinaire Info n init ARBREVIDE ;
Début
        Info(n) := v ;
        filsG(n) := a;
        filsD(n) := b;
        enracinement := n;
fin
```

```
Arbre enracinement(Arbre a, Arbre b)
{
      Nœud *n = malloc(sizeof(Nœud));
      n->info = NULL;
      n->filsG = a;
      n->filsD = b;
      return n ;
}
```

Feuille

```
fonction estFeuille(ArbreBinaire Info a):Booléen
variables
        Booléen res init FAUX;
début
        si vide(filsG(a)) ET vide(filsD(a)) alors res := VRAI;      finsi
        estFeuille := res;
fin
```

Comptage

```
fonction compte(ArbreBinaire Info a)
variables
        entier res init 0;
début
        si NON vide(a) alors
                res:=1+compte(filsG(a))+compte(filsD(a));
        finsi
        compte := res;
fin
```

hauteur d'un arbre

```
fonction hauteur(ArbreBinaire a) : entier
variables
        entier res init 0 ;
début
        si NON vide(a) alors res := max(hauteur(filsG(a), hauteur(filsD(a))) + 1 ; finsi
        hauteur := res ;
fin
```

Présence d'un élément dans un arbre

```
fonction appart(Info v, ArbreBinaire a)
variables
        Booléen res init FAUX ;
début
        si NON vide(a)  alors
                si v = info(a) alors res := VRAI ;
                sinon   res := appart(v, filsG(a)) OU| appart(v, filsD(a)) ;
                finsi
        finsi
        apprt := res ;
fin
```

Suppression d'un élément dans un arbre

S'il n'y a pas d'ordre, on remplace l'élément supprimé par n'importe quel élément. Par convention, on prend le 1^{ier} du fils gauche de l'élément supprimé, etc. S'il y a un ordre, il faut remplacer l'élément supprimé par un élément qui vérifie cette condition d'ordre. Cette fonction supprime toutes les occurrences d'un élément dans un arbre non ordonné.

```
fonction supparbre (info v, ArbreBinaire a) : ArbreBinaire
variables
        ArbreBinaire res = ARBREVIDE ;
début
  si  NON vide(a) alors
    si v = info(a) alors
      si  vide(filsG(a))   alors res := supparbre (v, filsD (a)) ;
      sinon res:=enraciner(info(filsG(a)), suppdarbre(info(filsG(a)),filsG(a)), suppdarbre(v, filsD(a)));
      finsi
    sinon res := enraciner(info(a), suppdarbre(v,supparbre(info(filsG(a)),filsG(a)))), suppdarbre(v, filsD (a))) ;
    finsi
  finsi
  supparbre := res ;
fin
```

Arbres binaires ordonnés

Suppression d'un élément dans un arbre ordonné

Lorsqu'on supprime le sommet, il faut le remplacer par le minimum de fils droit pour respecter la condition d'ordre. Il faut donc d'abord écrire une fonction qui trouve le minimum d'un arbre.

```
fonction minArbre(ArbreBinaire a) : Info
variables
        Info res ;
début
        si  NON vide (a) alors
                si  vide(filsG(a))   alors res :=info(a) ;
                sinon   res := minArbre(filsG(a)) ;
                finsi
        finsi
        minArbre := res ;
fin
```

```
fonction suppordrearbre(Info v, ArbreBinaire  a) : ArbreBinaire
variables
        ArbreBinaire res init ARBREVIDE ;
début
  si NON vide(a) alors
    si  v = info(a) alors
      si  vide(filsD(a))  alors res := filsG (a) ;
      sinon res := enraciner(minArbre(filsD(a)), filsG (a), suppordrearbre(minArbre(filsD(a)),filsD (a))) ;
      finsi
    sinonsi v < sommet  res := enraciner (sommet(a), suppordrearbre(v, filsG (a)), filsD (a)) ;
    sinon res := enraciner (info(a), filsG(a), suppordrearbre(v, filsD(a))) ;
    finsi
  finsi
  suppordrearbre := res ;
fin
```

Insertion d'un élément dans un arbre ordonné

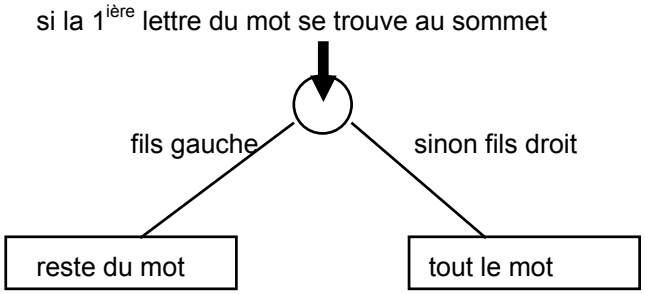
Le but est d'insérer une valeur à la bonne place. L'algorithme est le suivant : on compare la valeur au nœud. Si elle est inférieure, on explore le fils droit. Si elle est supérieure, on parcourt le fils gauche. On recommence pour le fils suivant jusqu'aux feuilles.

```
fonction insererordonné (ArbreBinaire  a, Info v) : ArbreBinaire
variables
        ArbreBinaire res;
début
        si vide(a)  alors res := nouveauNoeud(v) ;
        sinonsi info(a) ≤ v alors res := enraciner(info(a), insererordonné(filsG(a), v), filsD(a)) ;
        sinon res := enraciner(info(a), filsG(A), insererordonné(filsD(a), v) ;
        finsi
        insererordonné := res ;
fin
```

```
procédure afficheArbreOrdonné(ArbreBinaire Info a)
début
        si NON vide(a) alors
                afficheArbreOrdonné (filsG(a));
                afficher(info(a));
                afficheArbreOrdonné (filsD(a));
        finsi
fin
```

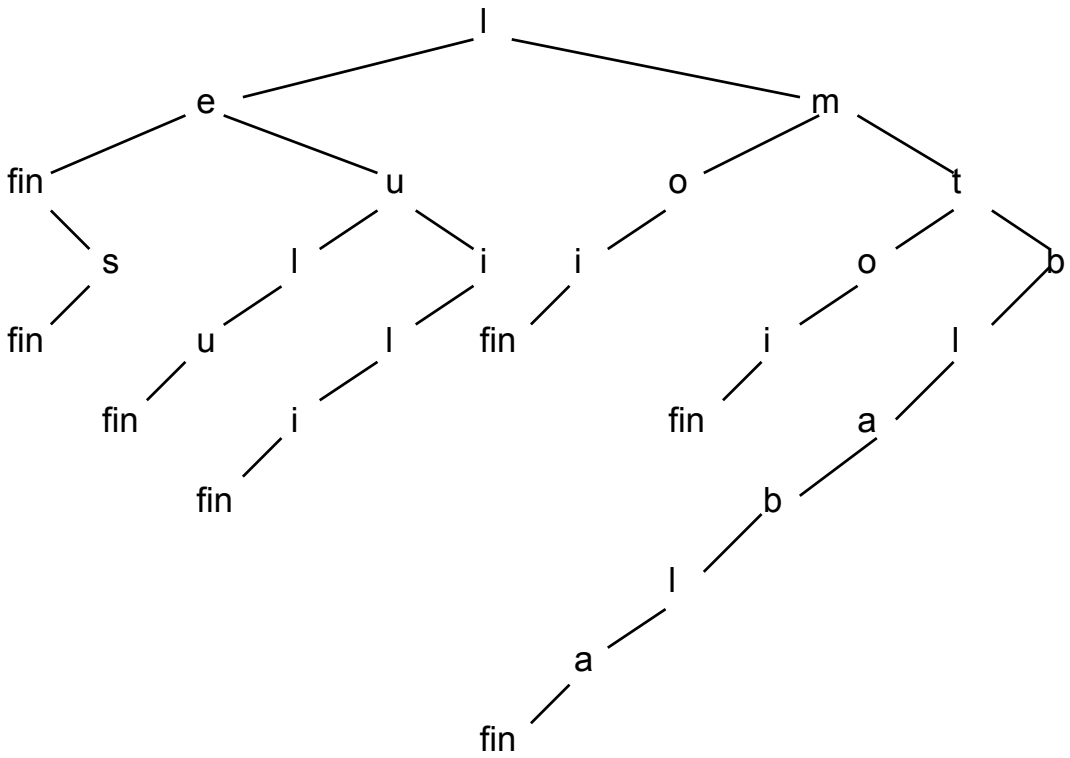
Application : les arbres dictionnaires

pas seulement pour les mots



Remarque : même si lettre suivante avant m, on va dans fils gauche.

Exemple : le les moi toi blabla lulu lili



Après chaque mot, il y a un caractère fin qui va dans le fils gauche de la dernière lettre du mot.
Si la première lettre du mot est dans la racine alors le reste du mot est à gauche.
Si la première lettre du mot n'est pas dans la racine alors le mot est à droite.
Il faut commencer cet arbre avec la première lettre la plus fréquente.

```
fonction  rechercher_mot(Liste m, ArbreBinaire a) : Booléen
variables
        Booléen res init FAUX ;
début
        si NON vide (a) alors
                si vide(m) alors
                        si  info(a) = fin   res := VRAI;
                        sinon  res := FAUX ;
                        fsi
                sinonsi tete(m) = info(a)  res := rechercher_mot(suivant(m), filsG(a)) ;
                sinon res := rechercher_mot(m, filsG(a)) ;
                finsi
        finsi
        rechercher_mot := res ;
fin
```

```
fonction  inserer_mot(Liste m, ArbreBinaire a) : ArbreBinaire
variables
        ArbreBinaire res;
début
        ii vide(a) alors res := construire(m) ;
        sinonsi videliste(m) alors
                si  info(a) = fin   alors res := a ;
                sinon  res := enraciner(fin, ARBREVIDE, a) ;
                finsi
        sinonsi tete(m) = info(a) alors res : enraciner(tete(m), inserer_mot(suivant(m), filsG(a)), filsD(a)) ;
        sinon   res := enraciner(tete(m), filsG(a), inserer_mot(m, filsD(a))) ;
        finsi
        inserer_mot := res ;
fin
```

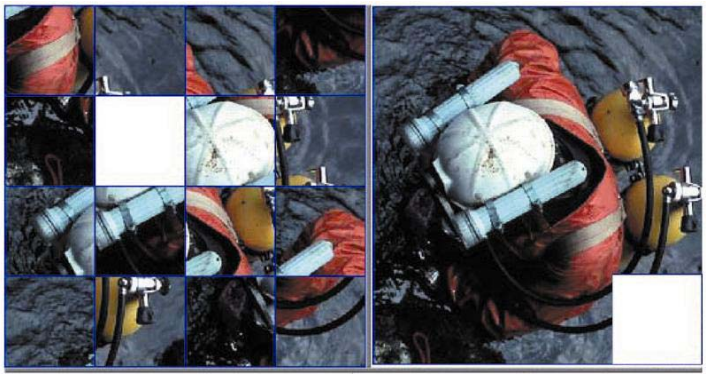
```
fonction construire(Liste m) : ArbreBinaire
variables
        ArbreBinaire res ;
début
        si  videliste(m)  alors res := enraciner(fin, ARBREVIDE, ARBREVIDE) ;
        sinon res := enraciner(tete(m), enraciner(suivant(m)), ARBREVIDE) ;
        finsi
        construire := res ;
fin
```

Parcours heuristique d'arbres

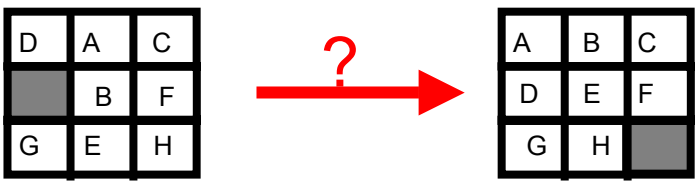
Un arbre est une structure particulièrement adaptée pour représenter l'exploration des solutions d'un problème, dans le cas de la résolution par essais successifs. Toutefois, l'efficacité de la recherche de la solution dépend souvent de l'ordre dans lequel elles ont explorées. On peut alors avoir recours à un autre type de parcours des arbres, dit heuristique, qui permet d'explorer des branches à la demande, et non de manière systématique comme le parcours en profondeur d'abord.

Exemple : le jeu du taquin

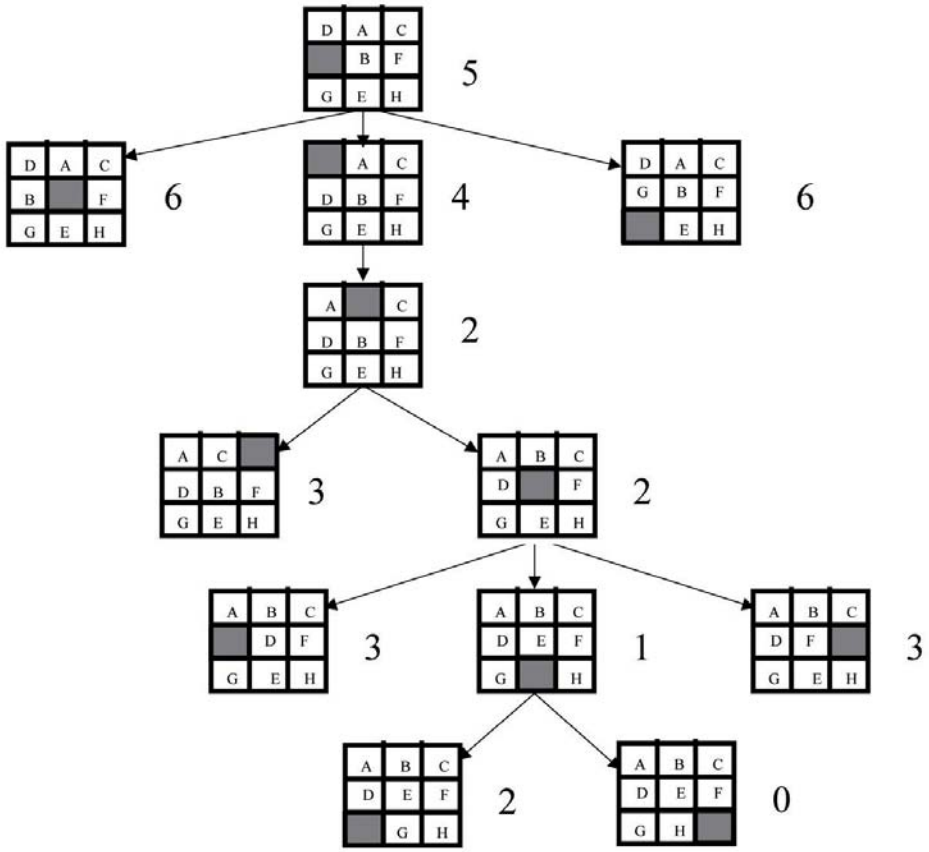
C'est un jeu dont le but est de reconstituer à l'aide de permutations une image découpée en morceaux et disposés sur une grille carrée. Les permutations sont uniquement possible avec l'unique emplacement vide. Comment faire pour revenir à la configuration idéale ?



Tout d'abord il faut modéliser le jeu. On utilisera alors un tableau à 2 dimensions, qui contient les pièces repérées par un numéro ou un caractère. Une valeur servira à représenter la case libre.
Toute action de jeu se résume à inverser les contenus de la case vide et d'une de ses voisines. Ces permutations sont au nombre de 2, 3 ou 4 en fonction de la position du trou.
Pour revenir à la configuration idéale, on peut être tenté de tester tous les coups possibles, et de refaire ceci à partir de chaque configuration obtenue, jusqu'à obtenir la solution. Le problème est qu'on retrouve des états déjà connus, ce qui ne garantit pas que l'on va finir par tomber sur la configuration finale. Pour éviter de retrouver des configurations déjà visitées (bouclage), on se propose de les mémoriser. On a pour cela un grand choix de structures, comme les listes, ou mieux les arbres, qui sont bien adaptés pour décrire le cheminement qui a conduit au résultat.
Parmi les possibilités, il est intéressant essayer d'abord celles qui "semblent" les meilleures. Ainsi, on ne souhaite pas parcourir tout l'arbre des configurations, mais seulement développer à la demande certains nœuds. On introduit alors la notion d' « heuristique », qui est une valeur associée à chaque configuration et qui est une estimation de sa proximité de la solution. Pour ce qui est du jeu du taquin, une heuristique efficace est par exemple la somme des distances de chaque portion d'image à sa position idéale. La distance entre 2 cases est la somme des valeurs absolues des différences de lignes et de colonnes. Sur la figure suivante, le trou est à une distance 3 de sa position idéale (1 ligne et 2 colonnes).
Enfin, il faut s'assurer que la configuration de départ à bien été obtenue par « mélange » de l'image idéale, et qu'on peut donc réobtenir cette image, ce qui n'est pas forcément possible depuis une configuration quelconque. Ceci pourrait être vérifié à l'aide d'une fonction de détection de bouclage.



La figure suivante montre l'arbre de parcours heuristique des solutions : toutes les configurations ne sont pas développées, seulement celles qu'on estime être les plus favorables.




```
typedef unsigned char Jeu[MAXSIZE][MAXSIZE];
```

```
typedef struct t_arbre
{
      Jeu jeu;
      int distance;
      int mt, nt;
      int marque;
      struct t_arbre *fils[4];
} Noeud, *Arbre;
```

```
void afficheJeu(Jeu jeu, int h, int l) ;
void initTaquin(Jeu jeu, Jeu ref, int *H, int *L, char *nf) ;
void chercherVal(unsigned char val, Jeu ref, int h, int l, int *m, int *n) ;
int valJeu(Jeu jeu, Jeu ref, int h, int l) ;
Noeud *meilleureConfig(Arbre arbreJeu) ;
int pasDansArbre(Jeu jeu, Arbre arbreJeu, int h, int l) ;
Arbre enraciner(Jeu jeu, Jeu ref, Arbre arbre, int numFils, int h, int l) ;
void jouer(Jeu ref, int h, int l, Arbre arbreJeu) ;
```

Arbres équilibrés (bien balancés)

Arbres n-aires

Il est relativement aisé de manipuler des arbres dont les nœuds comportent plus de 2 fils. Toutefois cela devient plus complexe lorsque ce nombre peut être quelconque.
Dans un arbre n-aire, un nœud peut posséder un nombre quelconque de fils. La structure la mieux adapté à la gestion d'une collection d'informations étant la liste, un nœud d'arbre n-aire pourra alors être décrit par sa valeur et une liste de ses fils, qui sont eux aussi des arbres…
Il est tout à fait possible de représenter cette liste par un tableau dynamique, ou par une liste chaînée.

Modélisation des arbres n-aires

Arbres n-aires avec des tableaux dynamiques

```
typedef struct t_arbreN {
      Info *info;
      int nbFils;
      struct t_arbreN *fils[]; // tableau
} *ArbreN, Nœud;

ArbreN a = NULL;
```

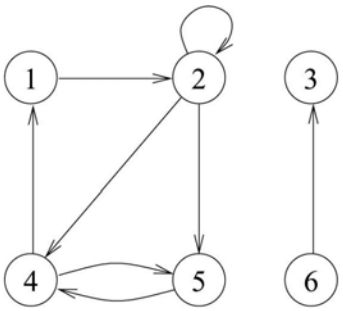
Arbres n-aires mis en œuvre avec des listes chaînées

```
typedef struct t_arbre; // réf. avant
typedef struct t_liste {
      struct t_arbre *a;
      struct t_liste *suivant;
} *Liste, Cellule; // Liste d'arbres
typedef struct t_arbreN {
      Info *info;
      Liste listeFils;
} *ArbreN, Nœud; // Arbre n-aire
```

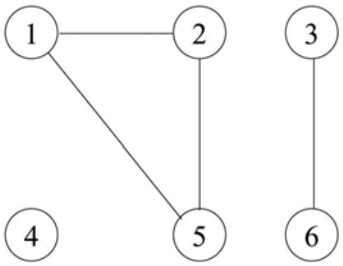
Les graphes

Définition

Un **graphe orienté** *G* est représenté par un couple (*S*, *A*) où *S* est un ensemble fini et *A* une relation binaire sur *S*. L'ensemble *S* est **l'ensemble des sommets** de *G* et *A* est **l'ensemble des arcs** de *G*. La figure suivante à gauche est une représentation graphique du graphe orienté *G* = (*S*,*A*) avec l'ensemble de sommets *S* = {1,2,3,4,5,6} et l'ensemble d'arcs *A* = {(1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3)} ; les sommets étant représentés par des cercles et les arcs par des flèches. On notera que les **boucles**—une boucle étant un arc qui relie un sommet à lui-même — sont ici possibles.



– Exemple de graphe orienté.



– Exemple de graphe non orienté.

Dans un **graphe non orienté** *G*=(*S*,*A*), l'ensemble des **arêtes** *A* n'est pas constitué de *couples* mais de *paires* de sommets—une paire étant non ordonnée contrairement à un couple. Par convention, on représente l'arête entre les sommets *u* et *v* non par la notation {*u*,*v*} mais, indifféremment, par les notations (*u*,*v*) ou (*v*,*u*). Dans un graphe non orienté les boucles sont interdites et chaque arête est donc constituée de deux sommets distincts. La figure précédente, à droite, est une représentation graphique du graphe non orienté *G*=(*S*,*A*) avec l'ensemble de sommets *S*={1,2,3,4,5,6} et l'ensemble d'arcs ={(1,2), (2,5), (5,1), (6,3)}.
Si (*u*,*v*) est un arc d'un *graphe orienté G* = (*S*,*A*), on dit que (*u*,*v*) **part** du sommet *u* et **arrive** au sommet *v*.
Si (*u*,*v*) est une arête d'un graphe *non orienté G* = (*S*,*A*), on dit que l'arête (*u*,*v*) est **incidente** aux sommets *u* et *v*.
Dans un *graphe non orienté*, le **degré** d'un sommet est le nombre d'arêtes qui lui sont incidentes. Si un sommet est de degré 0, comme le sommet 4 de la figure à droite, il est dit **isolé**. Dans un *graphe orienté*, le **degré sortant** d'un sommet est le nombre d'arcs qui en partent, le **degré (r)entrant** est le nombre d'arcs qui y arrivent et le **degré** est la somme du degré entrant et du degré sortant.
Un ***chemin*** est la suite ***d'arcs*** qui permettent de relier un sommet i à un sommet j.
Dans un *graphe orienté G* = (*S*,*A*), un **chemin** de **longueur** *k* d'un sommet *u* à un sommet *v* est une séquence (u_0,u_1, ...,u_k) de sommets telle que $u = u_0$, $v = u_k$ et (u_{i-1},u_i) $\in A$ pour tout *i* dans {1, ...,*k*}. Un chemin est **élémentaire** si ces sommets sont tous distincts. Dans la figure du graphe orienté, le chemin (1,2,5,4) est élémentaire et de longueur 3, mais le chemin (2,5,4,5) n'est pas élémentaire. Un **sous-chemin** *p*' d'un chemin *p* = (u_0,u_1, ...,u_k) est une sous-séquence contiguë de ses sommets. Autrement dit, il existe *i* et *j*, 0 <= *i* <= *j* <= *k*, tels que *p*' = (u_i,u_{i+1}, ...,u_j). On définit dans les *graphes non orientés* la notion correspondante de **chaîne**.
Un ***circuit*** est un chemin dont les extrémités coïncident.
Dans un *graphe orienté G* = (*S*,*A*), un chemin (u_0,u_1, ...,u_k) forme un **circuit** si $u_0 = u_k$ et si le chemin contient au moins un arc. Ce circuit est **élémentaire** si les sommets u_1, ..., u_k sont distincts. Une boucle est un circuit de longueur 1. Dans un *graphe non orienté G* = (*S*,*A*), une chaîne (u_0,u_1, ...,u_k) forme un **cycle** si *k* >= 3 et si $u_0 = u_k$. Ce cycle est **élémentaire** si les sommets u_1, ..., u_k sont distincts. Un graphe sans cycle est dit **acyclique**.
Un *graphe non orienté* est **connexe** si chaque paire de sommets est reliée par une chaîne. Les **composantes connexes** d'un graphe sont les classes d'équivalence de sommets induites par la relation « est accessible à partir de ». Le graphe non orienté de la figure contient trois composantes connexes : {1,2,5}, {3,6} et {4}.
Un *graphe orienté* est **fortement connexe** si chaque sommet est accessible à partir de n'importe quel autre. Les **composantes fortement connexes** d'un graphe sont les classes d'équivalence de sommets induites par la relation « sont accessibles l'un à partir de l'autre ». Le graphe de la figure gauche contient trois composantes connexes : {1,2,4,5}, {3} et {6}.
On dit qu'un graphe G' = (S',A') est un sous-graphe de G = (S,A) si S' $\subset$ S et si A' $\subset$ A.

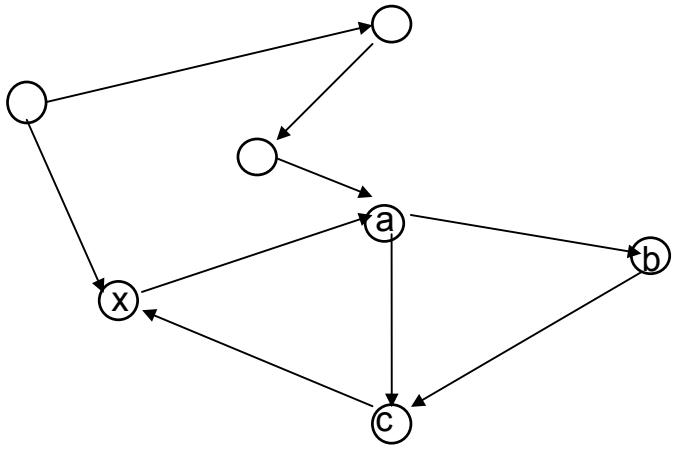
Les graphes ***pondérés*** / ***valués*** ont des poids : chaque arc a une valeur.

Parcours des graphes

Le parcours des graphes se révèle être un peu plus compliqué que celui des arbres. En effet, les graphes peuvent contenir des cycles et nous voulons éviter de parcourir indéfiniment ces cycles ! Pour éviter cet écueil on colorie les sommets des graphes : initialement les sommets sont tous blancs ; lorsqu'il est rencontré pour la première fois un sommet est peint en gris ; lorsque tous ses successeurs dans l'ordre de parcours ont été visités, un sommet est repeint en noir.

Parcours en profondeur

```
fonction  PP (sommet x) : ens
variables
        ensemble Y init {} :
début
        pour tout j ∈ succ(x) faire
                Y := {j} ∪ PP(j) ∪ Y ;
        finpour
        PP := Y ;
fin
```



Les sommets à partir de x sont tous les successeurs de x et les successeurs des successeurs de x. Attention, cet algorithme ne fonctionne que s'il n'y a ni boucle ni cycle. Pour éviter cela, il faut marquer les éléments :

```
fonction  PP (Sommet x) : ens
variables ensemble Y init {} :
début
        pour tout j ∈ succ(x) faire
                si non marqué(j) alors
                        marqué(j) := VRAI ;
                        Y := {j} ∪ PP(j) ∪ Y ;
                finsi
        finpour
        PP := Y ;
fin
```

```
fonction PP(Graphe G) : Graphe
variables
        Tableau Couleur couleur[] ;
        Sommet u ;
début
        pour chaque sommet u de G faire couleur[u] := BLANC ; finpour
        pour chaque sommet u de G faire
                si couleur[u] = BLANC alors (G, couleur) := VISITER-PP(G, u, couleur) ;
                finsi
        finpour
        PP := G ;
```

```
fin

fonction VISITER-PP(Graphe G, Sommet s, Tableau Couleur couleur[]) : (Graphe, Tableau Couleur)
variables
        Sommet v ;
début
        couleur[s] := GRIS ;
        pour chaque voisin v de s faire
                si couleur[v] = BLANC alors (G, couleur) := VISITER-PP(G, v, couleur) ; finsi
        finpour
        couleur[s] := NOIR ;
        VISITER-PP := (G, couleur) ;
fin
```

Parcours en largeur

```
fonction PL(Graphe G, Sommet s) : Ensemble
variables
        Sommet u, v ;
        Tableau Couleur couleur[] ;
        Ensemble F ;
début
        couleur[s] := GRIS ;
        pour chaque sommet u de G, u <> s, faire couleur[u] := BLANC ; finpour
        F := {s} ;
        tant que F <> {} faire
                (u, F) := SUPPRESSION(F) ;
                pour chaque voisin v de u faire
                        si couleur[v] = BLANC
                        alors   couleur[v] := GRIS ;
                                F := INSERTION(F,v) ;
                        finsi
                finpour
                couleur[u] := NOIR ;
        fintantque
        PL := F ;
fin
```

Représentation des graphes

On représente les graphes sous forme matricielle ou sous forme de listes. Comment trouver les cycles ?

Tableau à 2 dimensions :

C'est la représentation la plus simple et la plus volumineuse : n sommets = tableau n²

	1	2	3	4	5	
1	0	0	1	0	0	V_1
2	0	1	0	0	0	V_2
3	0	0	0	1	0	V_3
4	0	0	0	0	1	V_4
5	1	0	1	0	0	V_5

Il faut choisir et préciser une orientation de lecture : les successeurs sont ici les lignes et les prédécesseurs les colonnes. Si le graphe est valué, au lieu de 1 on met la valeur des arcs. Si les sommets sont valués, on

prévoit une colonne supplémentaire pour la valeur des sommets. On trouve les boucles en regardant la diagonale du tableau.

Les matrices d'incidence aux arcs :

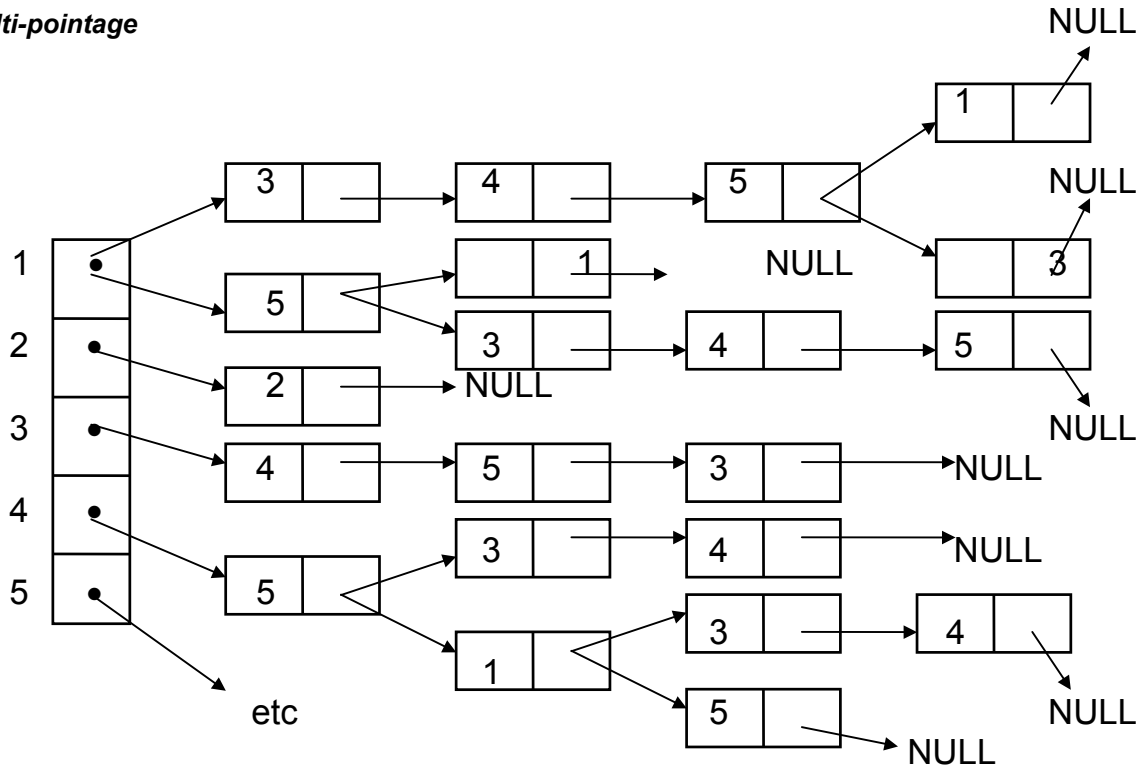
pour un graphe orienté non valué.

i \ j (arcs)	1	2	3	4	5	6	7
1	1	-1	1	0	0	0	0
2	0	0	0	0	0	0	V
3	0	0	-1	0	1	0	0
4	0	0	0	0	1	-1	0
5	-1	1	0	1	-1	0	0

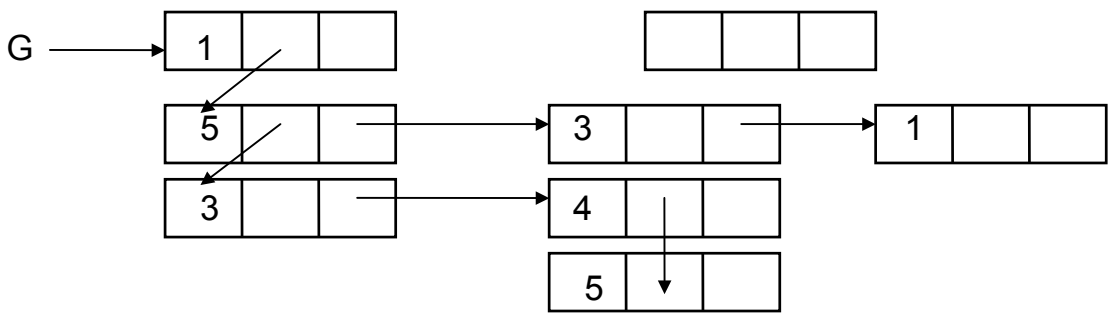
sommets

a(i, j) = -1 si j entrant, = 1 si j sortant, = 0 sinon, = V si boucle

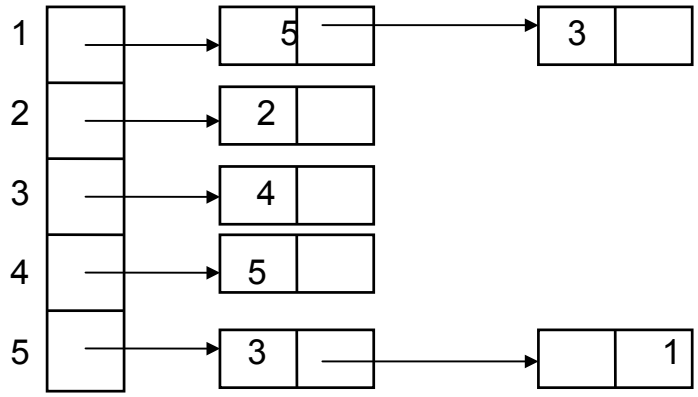
Le multi-pointage



Les têtes chaînées



Les têtes contiguës



Fonctions de manipulation des graphes

Graphe sans valeur

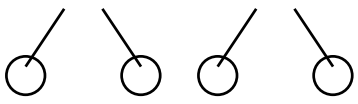
estvidegraphe → graphe ⇒ Bool
créegraphevide → rien ⇒ graphe
adjsomm → graphe + sommet ⇒ graphe
supsomm → graphe + sommet ⇒ graphe
adjarc → graphe + arc ⇒ graphe
suparc → graphe + arc ⇒ graphe
succsomm → graphe + sommet ⇒ sommet
predsomm → graphe + sommet ⇒ sommet
degint → graphe + sommet ⇒ entier
adjext → graphe + sommet ⇒ graphe
Enssucc → graphe + sommet ⇒ E(s) (ensemble de sommets)
Enspred → graphe + sommet ⇒ E(s)

Graphe avec valeur

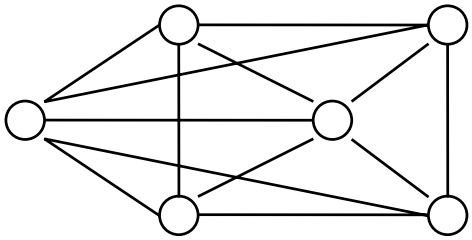
idem sauf :

adjsomm → graphe + sommet + valeur ⇒ graphe
supsomm → graphe + sommet + valeur ⇒ graphe
adjarc → graphe + arc + valeur ⇒ graphe
suparc → graphe + arc + valeur ⇒ graphe

Les arbres sont utilisés pour traiter des conditions.



Les graphes sont utilisés pour traiter des boucles.



Application : graphe de représentation d'une molécule

Plus courts chemins (à origine unique)

Dans un problème de plus courts chemins, on possède en entrée un graphe orienté pondéré $G = (S,A)$ de fonction de pondération $w : A \rightarrow \mathbb{R}$. Le poids du chemin $p = (v_0, v_1, ..., v_k)$ est la somme des poids de ses arcs :

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Le poids $\delta(u,v)$ d'un plus court chemin d'un sommet u à un sommet v est bien évidemment le minimum des poids des chemins de u à v (si celui-ci est défini, ce qui peut ne pas être le cas si le graphe contient un circuit de poids strictement négatif). Un **plus court chemin** d'un sommet u à un sommet v est alors un chemin de u à v de poids $\delta(u,v)$.
On souhaite dans cette section trouver les plus courts chemins depuis un sommet origine s et vers n'importe quel autre sommet. Dans la suite, $\pi[u]$ désignera le prédécesseur de u dans l'estimation du plus court chemin de s à u et $d[u]$
désignera la longueur de ce chemin.

Algorithme de Dijkstra

L'algorithme de Dijkstra résout le problème de la recherche d'un plus court chemin à origine unique pour un graphe orienté pondéré $G=(S,A)$ dans le cas où **tous les arcs ont un poids positif ou nul** : $\forall(u,v)\in A$, $w(u,v)>=0$.
L'algorithme de Dijkstra maintient à jour un ensemble E des sommets de G dont le plus court chemin à partir de l'origine s est connu et calculé. À chaque itération, l'algorithme choisit parmi les sommets de $S \setminus E$ — c'est-à- dire parmi les sommets dont le plus court chemin à partir de l'origine n'est pas connu — le sommet u dont l'estimation de plus court chemin est minimale. Cet algorithme est donc **glouton**. Une fois un sommet u choisi, l'algorithme met à jour, si besoin est, les estimations des plus courts chemins de ses successeurs (les sommets qui peuvent être atteint directement à partir de u).
SOURCE-UNIQUE-INITIALIZATION initialise les valeurs de $\pi[u]$ et de $d[u]$ pour chaque sommet u : initialement, il n'y a pas de chemin connu de s à u (si $u <> s$) et u est estimé être à une distance infinie de s.
RELÂCHER(u, v, w) compare le plus court chemin de s à v connu avec une nouvelle proposition (chemin estimé de s à u puis arc de u à v), et met les différentes données à jour si besoin est.
Voici l'algorithme de Dijkstra pour le calcul des plus courts chemins.

```
fonction SOURCE-UNIQUE-INITIALIZATION(Graphe G, Sommet s) : (Tableau valeur, Tableau valeur)
variables
        Tableau valeur d[],π[] ;
        Sommet v ;
début
        pour chaque sommet v de G faire
                d[v] := +∞;
                π[v] := NIL;
        finpour
        d[s] := 0;
        SOURCE-UNIQUE-INITIALIZATION := (d, π);
fin

fonction RELACHER(Sommet u, v; Poids w; Tableau valeur d[],π[]) : (Tableau valeur, Tableau valeur)
début
        si d[v] > d[u]+w(u,v) alors
                d[v] := d[u]+w(u,v);
                π[v] := u;
        finsi
        RELACHER := (d, π);
fin

fonction DIJKSTRA(Graphe G, Poids w, Sommet s) : Ensemble
variables
        Tableau valeur d[],π[] ;
        Ensemble E init {}, F ;
        Sommet u, v ;
début
```

```
        (d, π ) := SOURCE-UNIQUE-INITIALIZATION(G, s);
        F := S ;
        tant que F <> {} faire
                (u, F) := EXTRAIRE-MIN(F) ;
                E := E ∪ {u} ;
                pour chaque arc (u,v) de G faire
                        (d, π ) := RELÂCHER(u, v, w , d, π);
                finpour
        fintantque
        DIJKSTRA := E ;
fin
```

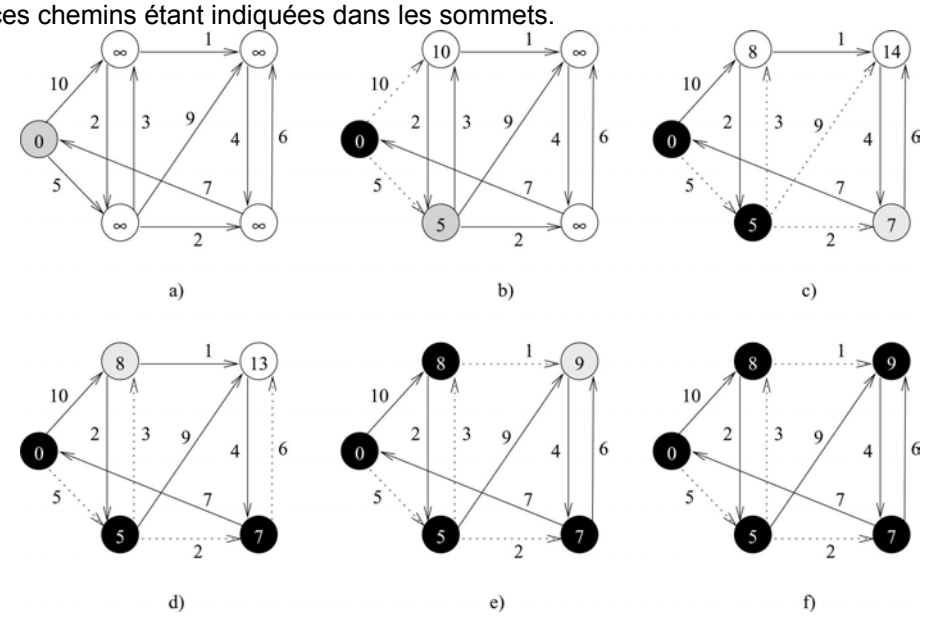
Cet algorithme fournit effectivement les plus courts chemins. L'algorithme glouton fonctionne uniquement parce que les poids sont positifs. On montre la correction de l'algorithme par récurrence.
– Le premier sommet ajouté à *E* est *s* car *d*[*s*] vaut alors 0 quand toutes les autres distances estimées sont infinies.
– Supposons qu'à un instant donné pour chaque sommet u de E, d[u] est bien la longueur du plus court chemin de s à u. On rajoute alors un sommet v à E. d[v] est alors minimale parmi les sommets de S\E. Montrons que d[v] = δ (s,v).
Soit p un plus court chemin de s à v Soit y le premier sommet de p n'appartenant pas à E. Par minimalité de d[v] on a : d[v]<=d[y]. De plus on a d[y] = δ(s,y) : parce que p contient le plus court chemin de s à x et donc de s au prédécesseur z de y, parce que d[z] = δ (s, z) par hypothèse de récurrence, et finalement parce que z a été relâché.
Par positivité des poids, δ(s,y)<=δ(s,v). Donc d[v]<=d[y] = δ(s,y)<= δ(s,v) et d[v] = δ(s,v).
La figure suivante présente un exemple d'exécution de l'algorithme de Dijkstra.

Complexité

La complexité de l'algorithme dépend de la complexité de l'opération EXTRAIRE-MIN. Dans le cas (défavorable) où on implémente l'ensemble *F* au moyen d'un simple tableau, la recherche du minimum coûte à chaque fois $\Theta(|F|) = O(|S|)$. La boucle « tant que » s'exécutant exactement $|S|$ fois, et chaque arc étant visité une unique fois, la complexité de l'algorithme est $O(|S|^2+|A|) = O(|S|^2)$.

Exemple

La figure suivante présente un exemple d'exécution de l'algorithme de Dijkstra : l'origine est le sommet le plus à gauche ; dans chaque graphe, les sommets noirs sont éléments de *E*, le sommet grisé est celui qui va être rajouté à *E* et les arcs en pointillés sont ceux utilisés pour les estimations des plus courts chemins, les longueurs de ces chemins étant indiquées dans les sommets.



Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford résout le problème des plus courts chemins avec origine unique dans le cas général où le poids des arcs peut être négatif. Appelé sur un graphe *G* = (*S*,*A*), l'algorithme de Bellman-Ford renvoie un booléen indiquant si le graphe contient ou non un circuit de poids strictement négatif accessible à partir de l'origine. Voici l'Algorithme de Bellman-Ford pour le calcul des plus courts chemins.

```
fonction  BELLMAN-FORD(Graphe G, Poids w, Sommet s) : Booléen
variables
        Tableau valeur d[],π[] ;
        Sommet u, v ;
        Booléen res init VRAI
début
        (d, π ) := SOURCE-UNIQUE-INITIALIZATION(G, s);
        pour i depuis 1 jusque |S|-1 faire
                pour chaque arc (u,v) ∈ A faire
                        (d, π ) := RELACHER(u, v, w , d, π);
                finpour
        finpour
        pour chaque arc (u,v) ∈ A faire
                si d[v] > d[u]+w(u,v) alors res := FAUX ;  (+ arrêt boucle) finsi
        finpour
        BELLMAN-FORD := res;
fin
```

Correction

La correction de l'algorithme de Bellman-Ford peut se montrer par récurrence sur le nombre d'arcs des plus courts chemins : à la fin de la $i$ème itération de la première boucle, les plus courts chemins contenant au plus i arcs sont connus, à la condition que le graphe ne contienne aucun circuit de poids strictement négatif. $|S|$-1 itérations suffisent car un plus court chemin est élémentaire (sans perte de généralité) et contient donc au plus $|S|$-1 arcs.

Vu ce qui précède, l'algorithme renvoie VRAI s'il n'y a pas de circuit de poids strictement négatif. Montrons qu'il renvoie FAUX sinon. Pour s'en convaincre, prenons un circuit c de sommets $u_0,u_1, ...,u_p,u_pu_0$. Si l'algorithme renvoie VRAI, alors pour tout $i \in [1, p]$, on a $d(u_i) <= d(u_{i-1})+w(u_{i-1},u_i)$. Par sommation on obtient :

$$\sum_{i=1}^{p} d(u_i) \le \sum_{i=1}^{p} d(u_{i-1}) + \sum_{i=1}^{p} w(u_{i-1},u_i) \Leftrightarrow \sum_{i=1}^{p} d(u_i) \le \sum_{i=1}^{p} d(u_i) + w(c) \Leftrightarrow d(u_p) \le d(u_0) + w(c) \Leftrightarrow 0 \le w(c)$$

Donc, si l'algorithme renvoie VRAI le graphe ne contient pas de circuit de poids strictement négatif.

Complexité

Cet algorithme est en $\Theta(|S|.|A|)$ car l'initialisation et la vérification de la non-existence d'un circuit de poids strictement négatif sont en $\Theta(|S|)$ et $\Theta(|A|)$ respectivement, et car la boucle « pour » s'exécute exactement $(|S|-1)$ fois et que chaque itération visite chaque arc exactement une fois ce qui nous coûte $\Theta(|S|.|A|)$.

Exemple

La figure qui suit est un exemple d'exécution de l'algorithme de Bellman-Ford : l'origine est le sommet le plus à gauche ; dans chaque graphe les arcs en pointillés sont ceux utilisés pour les estimations des plus courts chemins, les longueurs de ces chemins étant indiquées dans les sommets. Les arcs sont considérés dans l'ordre lexicographique : (u,v), (u,x), (u,y), (v,u), (x,v), (x,y), (y,v), (z,u) et (z,x).