Plan

Langage C

- struct
- Definition récursive de type
- sizeof
- malloc
- Listes chaînées

Algorithmique

- Listes, piles, files

struct

```
struct Point
{
        int x;
        int y;
};
```

```
struct Rectangle
{
        struct Point p1;
        struct Point p2;
};
```

```
struct Point p;

p.x = 3;
p.y = 5;
```

```
struct Rectangle r;

r.p1.x = 2;
...
```

```
struct Point *q;

q->x = 3;
q->y = 5;
```

sizeof

sizeof objet
donne la taille de l'objet

sizeof(nom de type)
donne la taille du type

Exemple :

sizeof(char) vaut 1
sizeof(short) vaut 2 (en principe...)
sizeof(int) : valeur dépendant de l'implantation.

malloc, calloc

Alloue de la mémoire

```
void TableauDynamique(void)
{
   int *s, *t;

   s = (int *)malloc(50 * sizeof(int));
   t = (int *)calloc(50, sizeof(int));
}
```

De plus, calloc (mais pas malloc !) initialise la zone allouée avec des zéros.

free

free(p) libère la mémoire pointée par p, allouée par malloc ou calloc.

Pièges

Chercher à libérer quelque chose qui n'a pas été alloué par malloc ou calloc est une erreur.

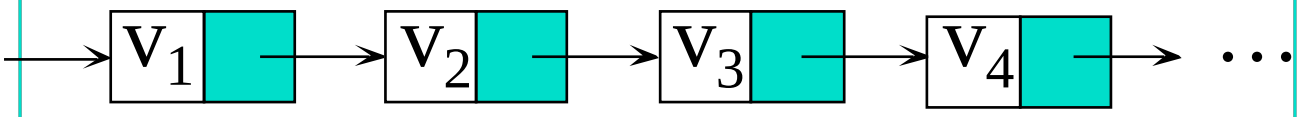
```
struct  Point *p;
```

définit la variable p de type pointeur sur Point, donc réserve la mémoire pour stocker une adresse, mais ne réserve pas la mémoire pour stocker une structure.

```
void Exemple2(void)
{
   struct  Point *p;

   p = (Point *)malloc(sizeof(Point));
}
```

Listes chaînées

```
typedef  int  Element;

struct  Cellule
{
   Element contenu;
   struct  Cellule *suivant;
};

typedef struct   Cellule Cellule, *Liste;
```

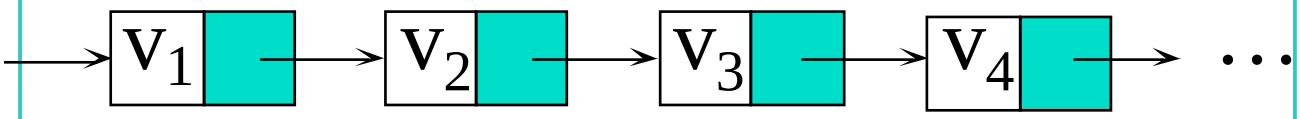


Création

```
Liste FaireLvide(void)
{
   return  NULL ; /* défini dans <stdio.h> */
}
```

Opérations sur les listes

```
int EstLvide(Liste a)
{
   return  a == NULL;
}
```




```
Liste Lajouter(Element x, Liste a)
{
   Liste b;

   b  = (Liste)malloc (sizeof (Cellule));
   b->contenu = x;
   b->suivant = a;
   return  b;
}
```

Initialisation d'une liste

```
Liste LCarres(int  n)      /* Carres */
{
    Liste a;

    a = FaireLvide();
    for  (i = n; i >= 2; --i)
        a = Lajouter(i * i, a);
    return (a);
}
```

Recherche dans une liste

```
int Lrecherche(Element x, Liste a)
{
   while (!EstLvide(a))
   {
      if (a->contenu == x)
         return 1;
      a = a->suivant;
   }
   return 0;
}

int LrechercheR(Element x, Liste a)
{
   if (EstLvide(a))
      return 0;
   else if (a->contenu == x)
      return 1;
   else
      return LrechercheR(x, a->suivant);
}
```

Longueur d'une liste

```
int LlongueurRec(Liste a)
{
   if (EstLvide(a))
      return 0;
   else
      return 1 + LlongueurRec(a->suivant);
}

int LlongueurIter(Liste a)
{
   int longueur = 0;

   while (!EstLvide(a))
   {
      ++longueur;
      a = a->suivant;
   }
   return longueur;
}
```

Suppression dans une liste

```
Liste LsupprimerRec(Element x, Liste a)
{
   Liste b;

   if (!EstLvide(a))
      if (a->contenu == x)
      {
         b = a;
         a = a->suivant;
         free(b);
      }
      else
         a->suivant = LsupprimerRec(x,
                                    a->suivant);
   return a;
}
```

```
Liste LsupprimerIter(Element x, Liste a)
{
   Liste b, c;

   if (!EstLvide(a))
      if (a->contenu == x)
      {
         c = a;
         a = a->suivant;
         free(c);
      }
      else
      {
         b = a ;
         while (!EstLvide(b->suivant) &&
                  b->suivant->contenu != x)
            b = b->suivant;
         if (!EstLvide(b->suivant))
         {
            c = b->suivant;
            b->suivant = b->suivant->suivant;
            free(c);
         }
      }
   return a;
}
```

Piles

Une pile est une liste où les insertions et les suppressions se font toutes du même coté.

```
#define MaxP  100

struct Pile
{
   int          hauteur;
   Element   contenu[MaxP];
};
typedef struct Pile Pile;

void FairePvide(Pile *p);
int EstPvide(Pile *p);
void Pajouter(Element x, Pile *p);
Element Pvaleur(Pile *p);
void Psupprimer(Pile *p);
```

Piles

```
void FairePvide(Pile *p)
{
   p->hauteur = 0;
}

int EstPvide(Pile *p)
{
   return p->hauteur == 0;
}

void Pajouter(Element x, Pile *p)
{
   p->contenu[p->hauteur] = x;
   ++p->hauteur;
}
```

Manipulation des piles

```
Element Pvaleur(Pile *p)
{
   int i;

   i = p->hauteur -1;
   return  p->contenu[i];
}

void  Psupprimer(Pile *p)
{
   --p->hauteur;
}
```

Initialisation d'une pile

```
Pile *PCarres(int n)      /* Carres */
{
    Pile *p = (Pile *)malloc (sizeof (Pile));
    int  i;

    FairePvide(p);
    for  (i = n; i >= 1; --i)
        Pajouter(i * i, p);
    return (p);
}
```

Files

Une file est une structure où les insertions se font en queue et les suppressions en tête. La valeur de la file est par convention l'élément de tête.

On peut l'implanter à l'aide d'un tableau circulaire (cf poly) ou à l'aide d'une liste munie de deux pointeurs fin et début.

Files

```
typedef struct   Fil
{
    Liste debut;
    Liste fin;
} Fil;

Fil FaireFvide(void)
{
    Fil f;

    f.debut = (Liste)malloc (sizeof (Cellule));
    f.fin = f.debut;
    return  f;
}

Liste FSuccesseur(Liste a)
{
    return  a->suivant;
}
```

X, Petite classe 5

```
int EstFvide(Fil f)
{
   return  f.debut == f.fin;
}

Element Fvaleur(Fil f)
{
   Liste b = FSuccesseur(f.debut);
   return  b->contenu;
}

Fil Fajouter(Element x, Fil f)
{
   Liste a = (Liste)malloc (sizeof (Cellule));

   a->contenu = x;
   a->suivant = NULL ;
   f.fin->suivant = a;
   f.fin = a;
   return  f;
}
```

Suppression dans une file

```
Fil Fsupprimer(Fil f)
{
    Liste a = f.debut;

    f.debut = FSuccesseur(f.debut);
    free (a);
    return  f;
}
```

Evaluation des expressions arithmétiques

• Expressions arithmétiques en notation préfixée :

- n (n entier naturel) est une expression préfixée
- Si e et f sont des expressions préfixées, alors

+ e f - e f * e f

sont des expressions préfixées

Exemple

* * + 5 3 - 2 3 * 2 2

[(5 + 3) * (2 - 3)] * (2 * 2)

ExprPrefixees.h

```
enum Nature {Symbole, Nombre};

struct Element
{
   enum Nature   nature;
   int           valeur;
   char          valsymb;
};
typedef struct Element Element;
typedef Expression Element[MaxP];


int Calculer (char a, int x, int y);
void Inserer (Element x, Pile *p);
int Evaluer (Expression u, int n);
```

ExprPrefixees.c

```
int Calculer(char a, int x, int y)
{
   switch (a)
   {
      case '+':
         return x + y;
      case '*':
         return x * y;
   }
}
```

X, Petite classe 5

```
void Inserer(Element x, Pile *p)
{
   Element y, z;

   if (EstPvide(p) || x.nature == Symbole)
      Pajouter(x, p);
   else
   {
      y = Pvaleur(p);
      if (y.nature == Symbole)
        Pajouter(y, p);
      else
      {
         Psupprimer(p);
         z = Pvaleur(p);
         Psupprimer(p);
         x.valeur = Calculer(z.valsymb,
                             x.valeur, y.valeur);
         Inserer(x,p);
      }
   }
}
```

Evaluation des expressions arithmétiques

```
int Evaluer(Expression u, int n)
{
    int i;
    Pile p;

    FairePvide(&p);
    for (i = 1; i <= n ; ++i)
        if (u[i].nature == Symbole ||
    u[i].valsymb == '+' || u[i].valsymb == '*')
            Inserer(u[i], &p);
    return (Pvaleur(&p)).valeur;
}
```

Manipulation des listes

```
Liste Tail(Liste a)     /* Suppression */
{                       /* du 1er élément */
   if (EstLvide(a))
   {
      printf ("Tail d'une liste vide.\n");
      exit (1);
   }
   else
      return  a->suivant;
}

Liste Append(Liste a, Liste b)
{                    /* concaténation */
   if (EstLvide(a))
      return  b;
   else
      return  Lajouter(a->contenu,
                       Append(a->suivant, b));
}
```

Manipulation des listes

```
Liste Nconc(Liste a, Liste b)
{
    Liste c;

    if (EstLvide(a))
        return b;
    else
    {
        c = a;
        while (!EstLvide(c->suivant))
            c = c->suivant;
        c->suivant = b;
    return a;
    }
}
```

X, Petite classe 5

```
void Nreverse(Liste a)          /* - */
{
   Liste b, c;

   b = FaireLvide();
   while (!EstLvide(a))
   {
      c = a->suivant;
      a->suivant = b;
      b = a;
      a = c;
   }
}

Liste Reverse(Liste a)
{
   if (EstLvide(a))
     return  a;
   else
     return Append(Reverse(a->suivant),
                   Lajouter(a->contenu, NULL));
}
```

Insertion dans une liste triée

On utilise ici une liste circulaire gardée. Le contenu de la première cellule contient le nombre d'éléments de la suite (ou autre chose) et le champ suivant de la dernière cellule contient l'adresse de la première.

```
Liste Insert(Element v, Liste a)
{
   Liste b;

   b = a->suivant;
   while (b->suivant != a && v >
                    b->suivant->contenu)
      b = b->suivant;
   b->suivant = Cons(v, b->suivant);
   ++a->contenu;
   return  a;
}
```