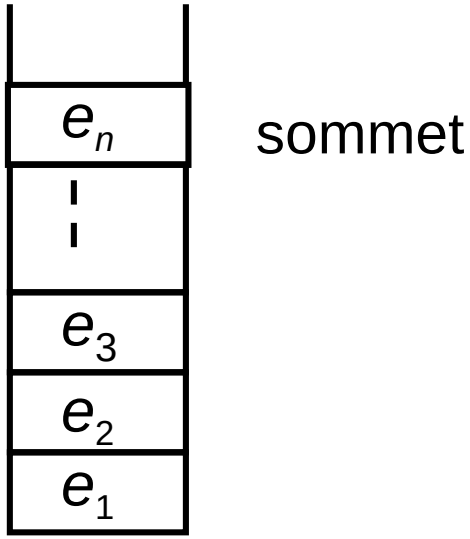
Types de données et structures de données

- Type de données abstrait : description d'un ensemble organisé d'objets et de manipulation sur cet ensemble (spécification, axiomes)

- Structure de données : implémentation explicite d'un type abstrait

Piles

Liste avec accès par une seule extrémité
liste LIFO : « Last In First Out »



Opérations

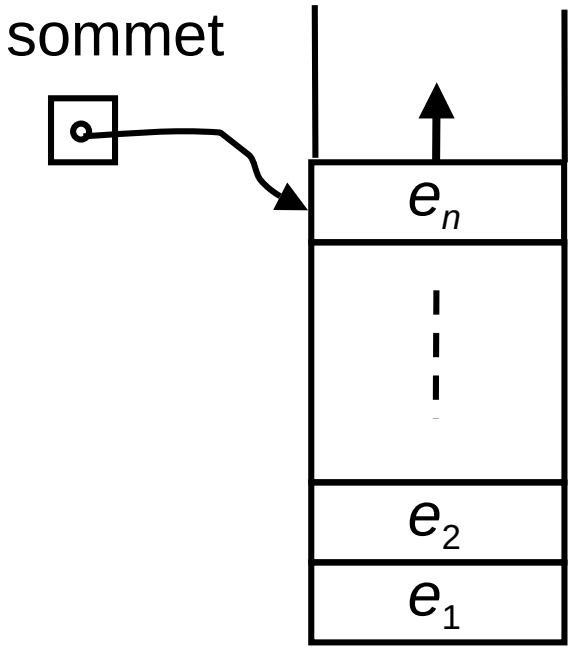
Pile_vide : $\rightarrow$ Pile
Empiler : Pile x Elément $\rightarrow$ Pile
Dépiler : Pile $\rightarrow$ Pile
Sommet : Pile $\rightarrow$ Elément
Vide : Pile $\rightarrow$ Booléen

Axiomes

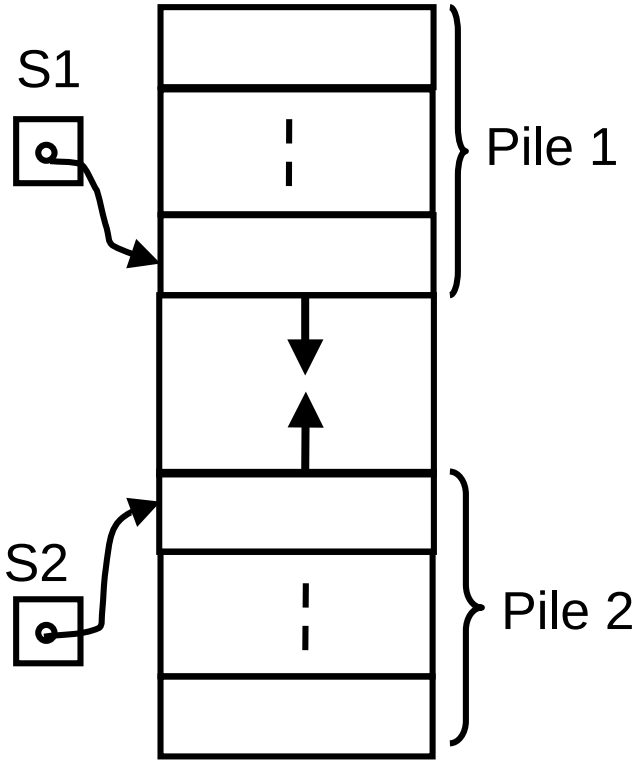
Dépiler (*P*) et Sommet (*P*) défini ssi Vide (*P*) = faux
Dépiler (Empiler (*P*, *e*)) = *P*
Sommet (Empiler (*P*, *e*)) = *e*
Vide (Pile_vide) = vrai
Vide (Empiler (*P*, *e*)) = faux

Implémentations possibles

1 pile



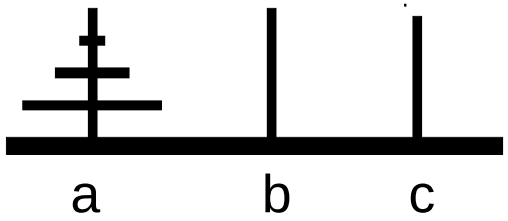
2 piles



> 2 piles

Listes chaînées

Tours de Hanoï



Coup : *x*-*y* = déplacement d ’un disque de *x* vers *y*

H (*n*, *x*, *y*, *z*) = suite de coups pour déplacer *n* disques de *x* vers *y* avec le baton intermédiaire *z*

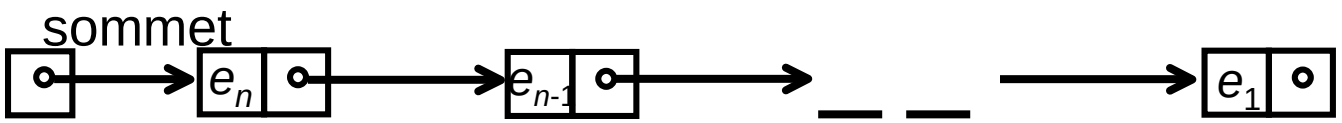
$$H(n, x, y, z) = \begin{cases} x\text{-}y & \text{si } n = 1 \\ H(n\text{-}1, x, z, y) \,.\, x\text{-}y \,.\, H(n\text{-}1, z, y, x) & \text{si } n > 1 \end{cases}$$

```
fonction  H (n, x, y, z);   /*version itérative*/
début
     P  ←  Pile_vide;   Empiler (P, (n, x, y, z));
     tant que non  Vide (P) faire début
          e  ←  Sommet (P);   Dépiler (P);
          si  e = x'-y' ou e = (1, x', y', z') alors
          écrire (x'-y')
          sinon début     /*e = (n, x', y', z')*/
          Empiler (P, (n-1, z', y', x'));
                Empiler (P, x'-y');
          Empiler (P, (n-1, x', z', y'));
          fin
     fin
fin
```

Piles en table

```
typedef struct{
     int sommet;                    /*Element est un type déjà
     Element t[MAX];}Pile;          defini, par exemple avec
                                    typedef int Element*/
Pile Pile_vide (Pile P){
     P.sommet = -1; return P;}

Pile Empiler (Pile P, Element e){
     if(P.sommet == MAX-1) printf(''débordement positif'');
     else{P.sommet++; P.t[P.sommet] = e; return P;}}

Pile Dépiler (Pile P){
     if(Vide(P)) printf(''débordement négatif'');
     else{P.sommet--; return P;}}

Element Sommet (Pile P){
     if(Vide(P)) printf(''pile vide'');
     else return P.t[P.sommet];}

int Vide (Pile P){
     return (P.sommet == -1);}
```

Piles en listes chaînées




```
typedef struct _cellule{
Element elt;
struct _cellule *succ;}cellule, *Liste;

typedef Liste Pile;

Pile Liste_vide (){return NULL;}

Pile Empiler (Pile L, Element e){
    Pile debut = (Liste)malloc(sizeof(cellule));
    debut -> elt = e; debut -> succ = L; return debut;}

Pile Dépiler (Pile L){
    if(L!=NULL) return L->succ;}

Element Sommet (Pile L){
    return L->elt;}

int Vide (Pile L){return (L == NULL);}
```

Files

Liste d'attente, queue
liste FIFO : « First In First Out »

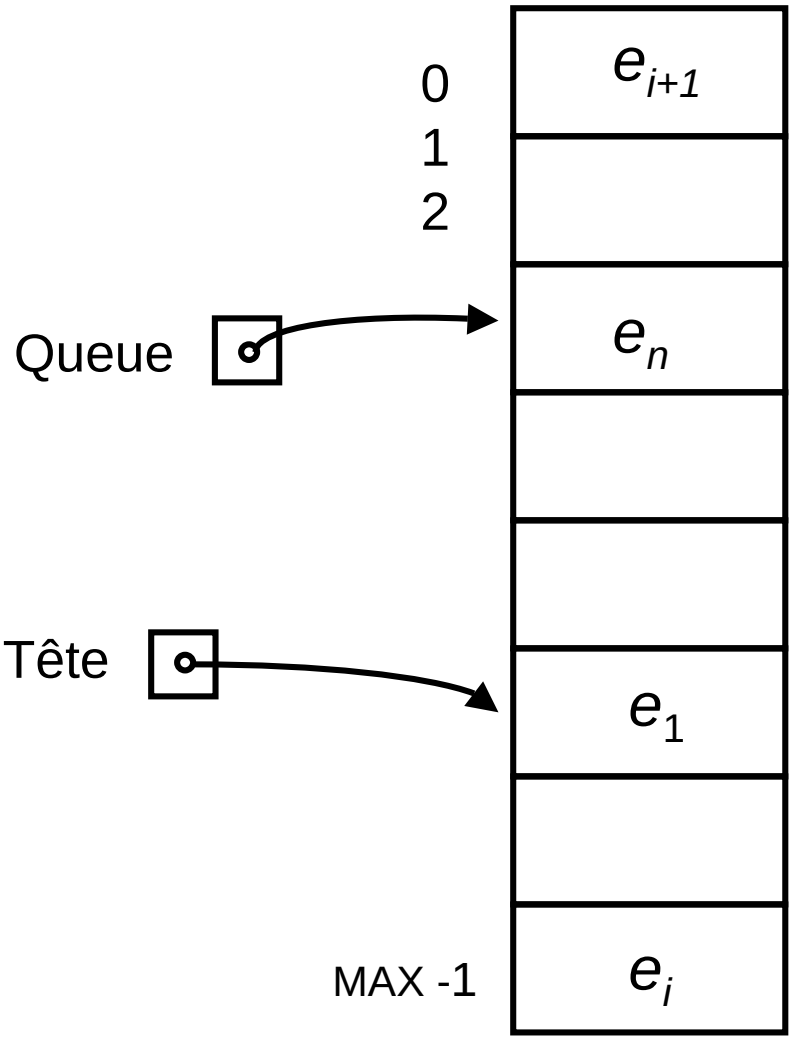
Opérations

File_vide : $\rightarrow$ File
Enfiler : File x Element $\rightarrow$ File
Défiler : File $\rightarrow$ File
Tête : File $\rightarrow$ Element
Vide : File $\rightarrow$ Booléen

Axiomes

Défiler (*F*) et Tête (*F*) définis ssi Vide (*F*) = faux
Vide (*F*) = vrai $\Rightarrow$ Tête (Enfiler (*F, e*)) = *e*
Vide (*F*) = faux $\Rightarrow$ Tête (Enfiler (*F, e*)) = Tête (F)
Vide (*F*) = vrai $\Rightarrow$ Défiler (Enfiler (*F, e*)) = file_vide
Vide (*F*) = faux $\Rightarrow$ Défiler (Enfiler (*F, e*)) = Enfiler (Défiler (*F*), *e*)
Vide (file_vide) = vrai
Vide (Enfiler (*F,e*)) = faux

Files en table



Succ (i) = i + 1 mod MAX

Condition :

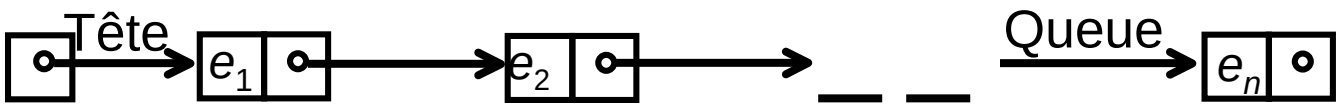
MAX ≥ max | F | + 1
F

```
typedef struct{
        int tete, queue;
        Element t[MAX];}File;

File File_vide (File F){
        F.tete = 0;
        F.queue = -1;
        return F;}
```

Files en table


```
File Enfiler (File F, Element e){
     if(Succ(F.queue)==F.tete && !Vide(F))
          printf(‘‘débordement positif’’);
     else{F.queue = Succ(F.queue);
          F.t[F.queue] = e;
          return F;}}

File Défiler (File F){
     if(Vide(F)) printf(‘‘débordement négatif’’);
     else
       if(F.tete == F.queue)return File_vide(F);
       else F.tete = Succ(F.tete); return F;}}

Element Tête (File F){
     if(Vide(F)) printf(‘‘file vide’’);
     else return F.t[P.tete];}

int Vide (File F){
     return (F.queue == -1);}
```

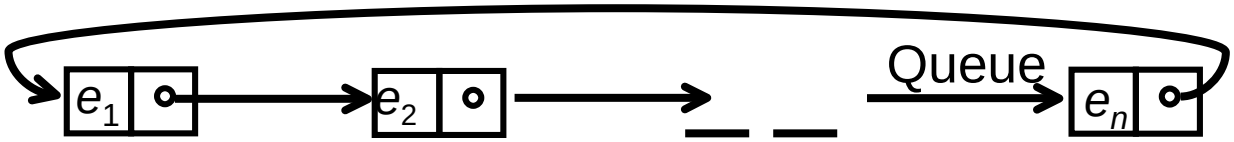
Files en listes chaînées




```
typedef struct {Liste tete; Liste queue;}File;

File File_vide(File F){F.tete = F.queue = NULL; return F;}

File Enfiler(File F, Element e){
    Liste N = (Liste)malloc(sizeof(cellule));
    N->elt = e; N->succ = NULL;
    if(Vide(F)) F.tete = F.queue = N;
    else{F.queue->succ = N; F.queue = F.queue->succ;}
    return F;}

File Defiler(File F){
    if(!Vide(F)){
      F.tete = F.tete->succ;
      if(F.tete == NULL) F.queue = NULL;}
    return F;}

Element Tete(File F){if(!Vide(F)) return (F.tete)->elt;}

int Vide(File F){return (F.tete==NULL && F.queue==NULL);}
```

Files en listes chaînées : implémentation par listes circulaires

UMLV ©




```
typedef Liste File;
File File_vide(){return NULL;}
File Enfiler(File F, Element e){
    Liste N = (Liste)malloc(sizeof(cellule)); N->elt = e;
    if(Vide(F)){F = N; N->succ = F;
    else{F->succ = N; N->succ= F->succ;}
    return N;}
File Defiler(File F){
    if(!Vide(F)){
      F->succ = F->succ->succ;
      if(F == F->succ) F = NULL;}
    return F;}
Element Tete(File F){if(!Vide(F)) return F->succ->elt;}
int Vide(File F){return F == NULL;}
```

Listes

Une liste est une suite finie d'éléments d'un même type

E ensemble

Liste : $(e_1, e_2, \ldots, e_n)$ où $n \geq 0$, $e_i \in$ E pour $i = 1, \ldots, n$

longueur $((e_1, e_2, \ldots, e_n)) = |(e_1, e_2, \ldots, e_n)| = n$

liste vide = ( ) = ε

position de e_i dans $(e_1, e_2, \ldots, e_n) = i$

accès séquentiel :

p = place ou adresse de e_i

Succ (p) = adresse de e_{i+1} si elle existe

Tête (L) = adresse de e_1

Axiomes

p adresse d'un élément de $L \Rightarrow \exists k \geq 0$ $p = \text{Succ}^k$ (Tête(L))

$k \geq$ longueur (L) $\Rightarrow \text{Succ}^k$ (Tête (L)) non défini

Opérations de base

Liste_vide : $\rightarrow$ Liste

Tête : Liste $\rightarrow$ Adresse

Fin : Liste $\rightarrow$ Liste

Cons : Elément x Liste $\rightarrow$ Liste

Premier : Liste $\rightarrow$ Elément

Elt : Adresse $\rightarrow$ Elément

Succ : Adresse $\rightarrow$ Adresse

Axiomes (exemples)

Tête (L), Fin (L), Premier (L) définis ssi $L \neq \varepsilon$

$L \neq \varepsilon \Rightarrow$ Premier (L) = Elt (Tête (L))

Fin (Cons (e, L)) = L Premier (Cons (e, L)) = e

$L \neq \varepsilon \Rightarrow$ Succ (Tête(L)) = Tête (Fin(L))

Quelques implémentations possibles

- Tables

  avantages : simple, économique en espace

  inconvénients : concaténation lente, opérations de modifications lentes

- Pointeurs

  - Adresse = adresse mémoire
  - Successeur explicite,
  - *L* peut être identifié à Tête (*L*)

  avantages : nombreuses opérations rapides, gestion souple, autorise plusieurs listes, accès à tout l'espace mémoire disponible

  inconvénients : pas d 'accès direct, un peu gourmand en mémoire

- Index

  - mémoire = tableau de MAX cellules
  - une cellule est un couple constitué d'un élément et d'un entier qui est l'indice du tableau mémoire où est stocké l'élément suivant de la liste

Extensions

Concaténation

Produit : Liste x Liste → Liste

$(e_1, \ldots, e_n) . (f_1, \ldots, f_m) = (e_1, \ldots, e_n, f_1, \ldots, f_m)$

Modifications

Ajouter : Elément x Adresse x Liste → Liste

ajouter *e* après l'élément d'adresse *p*

Supprimer : Adresse x Liste → Liste

Elément : Elément x Liste → Booléen

Autres

Place : Elément x Liste → Adresse

Position : Elément x Liste → Entier

Ième : Entier x Liste → Elément

Tri : Liste → Liste

etc.