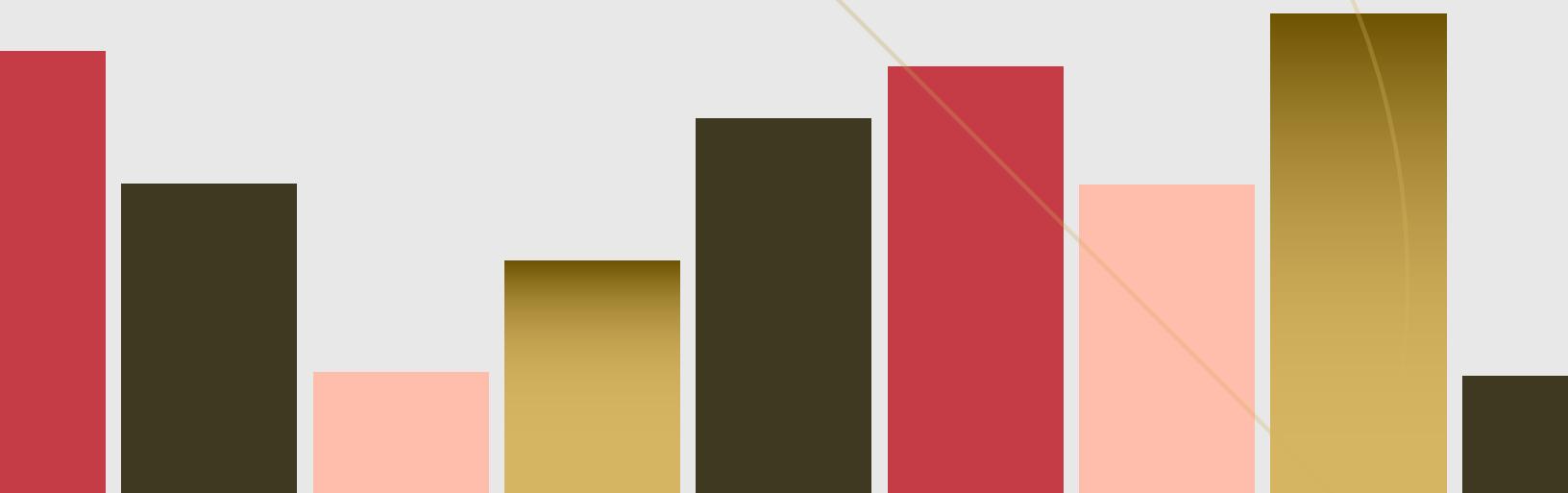


Structure de Données et Algorithmes



SDA1



Introduction

Construction de programmes :

→ Apprendre à résoudre un problème informatique en suivant une démarche concrète de résolution de prob.

1. Spécification informelle, puis formelle
2. décomposition fonctionnelle du problème
3. programme s'appuyant sur la spécification
4. test par des jeux d'essai reprenant tous les cas particuliers

Spécification informelle

→ Point de départ

→ Souvent imprécise, redondante, incomplète

le premier travail est alors d'améliorer l'énoncé en utilisant un formalisme de type mathématique pour obtenir une spécification formelle du problème.

C'est une sorte de mise en équation

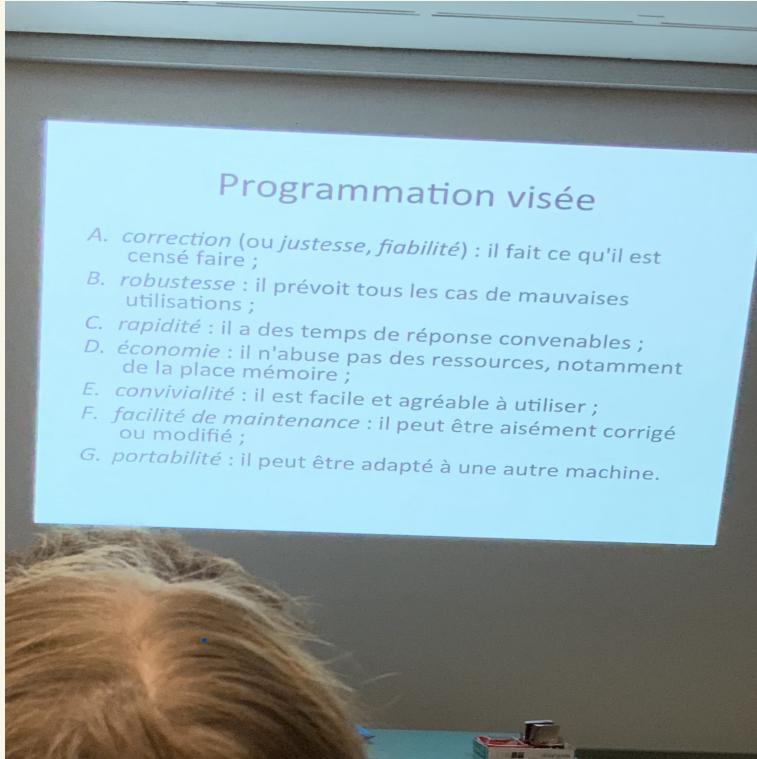
Spécification formelle

- Structure horizontale : classifier les données, objets et opérations qui entrent en jeu.
- Raffinement vertical : suite de spécifications, du tableau des charges au programme, énoncé du problème donné à plusieurs niveaux d'abstraction.

Procédés de résolution

- Décomposition fonctionnelle : décomposer le problème en prob. simple

Programmation visée



Prouver la qualité (A et B)

- En évaluant la qualité d'une spec via des preuves
- En évaluant la validité d'une implémentation par rapport à la spécia des preuves
- En évaluant la terminaison de la spéc.

Evaluer la rapidité (C et D)

Evalue des algo efficaces au terme de complexité en temps et espace .

- Calculer ordre de grandeur de
- Identifier s'il est optimal .

(chap. 4).

SDA 1

- Spécifier un type de données linéaires de ses opérations:
 - liste d'axiomes et préconditions, respecter le type des prototypes
 - identifier les différents pour de. opér.
 - intégrer toutes les conf. possibles et imaginables

Ch.1

Ch.2

- Programmer les variations d'une structure de données linéaires avec ses opérations.
 - distinguer les types de prog.
 - en conservant les arangements / inconc des struct de données et des opérations implément.

- S'appuyer sur la spé pour programmer dans l'algo on doit faire:
 - fonctions spécifiées avec leurs pré-cond. prototypes.
 - alg.

Chapitre 1 : Rappels de programmation et introduction à la spécification

1. Programmation

1.1 Type de base (prédéfinies)

```
base . h
#include <stdbool.h>
...
```

N	entier naturel	1, 2, 3	Nat
	relatif	-1, -2, -3	Ent
	rationnels	1/3	Rat
	réels	π	Reel

```
#define faux false
#define vrai true
typedef unsigned int Nat;
typedef int Ent;
```

Où $2^{16} - 1$

type des float Rat, Reel;

typedef char Car;

2. Spécification

2.1 Un tout petit exemple

Énoncé : définir une fonction calculant x^n avec x est rationnel et n entier

Plutôt que de « programmer discrètement », nous allons la spécifier en raffinant la spécification : tout d'abord par rapport aux spécifications de base, puis de manière récursive directe et enfin de manière itérative.

Raffinement de la spécification

Profil puissance: Rat Nat \rightarrow Rat /* fonction puissance */
préconditions: $n : \text{Nat}$, $x : \text{Rat}$

définition $\text{puiss}(x, n) = x \neq 0 \text{ ou } n \neq 0$

$$\begin{cases} \text{puiss}(x, 0) = 1 \\ \text{puiss}(x, n+1) = x \times \text{puiss}(x, n) \end{cases}$$

• définition "directe" $\text{puiss}(x, n) = \begin{cases} \text{si } n = 0 \text{ alors } 1 \\ \text{sinon } x \times \text{puiss}(x, n-1) \end{cases}$ fin

nombre d'appels récursifs

$$\text{puiss}(x, 3) = x \times \text{puiss}(x, 2) = x \times x \times \text{puiss}(x, 1) = x \times x \times x \times \text{puiss}(x, 0)$$

" 1.

En C on "calque" la spécification
programmation récursive

```
Rat puiss (Rat x, Nat n)
{
    return (n == 0) ? 1 : x * puiss(x, n-1);
}
```

• définition itérative /* de n à 0 */
 $\text{puiss}(x, n) = n$
avec $(i, n) = \text{init}(n, 1)$
tq $i \neq 0$
rèp $(i-1, n \times x)$ finp.

Programmation itérative

Rat puiss (Rat x, Nat n)

{ Nat i; Rat x_i;

$i = x$; $n = 1$;

while ($i \neq 0$)

{ $i = i - 1$;

$n = n \times x_i$

} return n;

}

i	3	2	1	0
n	1	$1 \times x$	$1 \times x \times x$	$1 \times x \times x \times x$

3 multiplications

② définition itérative 2 (de 0 à n-1)

puiss (x, n) = r
avec (i, n) = init ($0, 1$)
 $\text{tq } (i \neq n)$
 r \leftarrow r $\times x$
 rép ($i+1, n \times x$) f*rép*

programmation itérative

```
Rat puiss (Rat x, Nat n)
{
    Rat r; Nat i;
    i=0; n=1;
    while (i < n)
    {
        i=i+1;
        r=r*x;
    }
    return r;
}
```

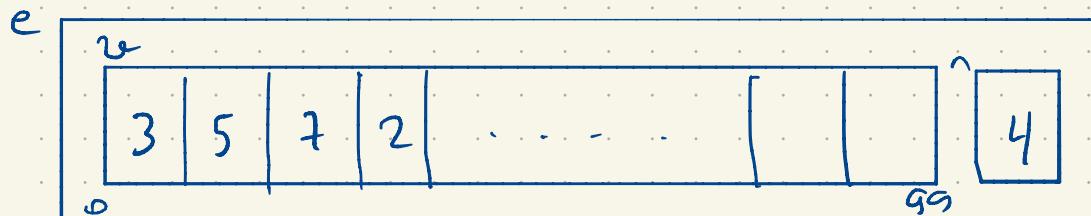
1.2 Structure de données

→ Pour des objets plus compliqués, il faut fabriquer soi-même un ou plusieurs types de données qui soient adaptées, à l'aide de tableaux, de structures, de pointeurs.

Exemple : Fabriquons des ensembles avec au plus 100 entiers naturels

Représenter un ensemble en C :

- Soit comme une structure composée d'un tableau de 100 entiers et un entier n représentant le nombre d'éléments



```
typedef struct stens { Nat a[100]; Nat n; } Ens;
```

2.2 Spécifications des Ensembles

Exemple : spécifier des ensembles finis d'entiers naturels tous distincts (sinon c'est un multi-ensemble) dont le cardinal est borné par 100.

$$\emptyset \quad \{3, 5, 7, 2\}$$

→ Ensemble dont le cardinal est borné par 100.

Spéc

ENS0 étend BASE

Sorte

ENS

Opérations

\emptyset	$- \rightarrow \text{Ens}$
i	$\text{Ens} \text{ Nat} \rightarrow \text{Ens}$
$-e-$	$\text{Nat} \text{ Ens} \rightarrow \text{Bool}$
ν	$\text{Ens} \rightarrow \text{Bool}$
$ _l$	$\text{Ens} \rightarrow \text{Nat}$
$-s$	$\text{Ens} \text{ Nat} \rightarrow \text{Ens}$
m	$\text{Ens} \rightarrow \text{Nat}$

\emptyset	$// \text{Ensemble nide}$	$\} \text{Générateurs de base}$
i	$// \text{insertion}$	$\} \text{« constructeurs »}$
$-e-$	$// \text{appartenance}$	$\} \text{Sélecteurs ou fonctions}$
ν	$// \text{test de vauoir}$	$\} \text{d'accès}$
$ _l$	$// \text{cardinal}$	
$-s$	$// \text{suppression}$	
m	$// \text{minimum}$	$\} \text{Générateurs}$

Préconditions

$e: \text{ens}; i: \text{Nat}$

$$\underline{\text{pré}} \quad i(e, z) = \exists x \in e \text{ et } |e| < 100$$

$$\underline{\text{pré}} \quad \nu(e, x) = x \in e$$

$$\underline{\text{pré}} \quad m(e) = \exists \nu(e)$$

Axiomes

$e: \text{Ens}; x, y: \text{Nat}$

$$(e_0) \quad i(i(e, x), y) = i(i(i(e, y), x))$$

$$\Rightarrow (e_1) \quad x \in \emptyset = \text{fau}$$

$$(e_2) \quad x \in i(e, y) = (x = y \text{ ou } (x \in e)) \rightarrow$$

$$(e_3) \quad \nu(\emptyset) = \text{vrai}$$

$$(e_4) \quad \nu(i(e, y)) = \text{fau}$$

$$(e_5) \quad |\emptyset| = 0$$

$$(e_6) \quad |i(e, y)| = |e| + 1$$

$$(e_7) \quad s(i(e, y), z) = \text{si } x = y \text{ alors } e \text{ sinon } i(s(e, z), y) \text{ fsi}$$

$$(e_8) \quad m(i(e, y)) = \text{si } \nu(e) \text{ alors } y$$

$$\text{sinon si } y < m(e) \text{ alors } y$$

$$\text{sinon } m(e) \text{ fsi fsi}$$

Spécification directe

$x \in e = \text{si } \nu(e) \text{ alors fau}$

sinon si $x = m(e)$ alors

vrai

sinon $x \in m(e, m(e))$ fsi

2.2 Spécification des Ensembles

Exemple : Spécifier des ensembles finis d'entiers naturels tous distincts (sinon c'est un multiensemble) dont le cardinal est borné par 100.

$$\emptyset = \{3, 5, 7, 2\}$$

→ Ensemble dont le cardinal est borné par 100 :

Spéc ENSO étend BASE

sorte ENS

opérations

\emptyset	$- \rightarrow \text{Ens}$	// Ensemble vide
i	$\text{Ens} \cdot \text{Nat} \rightarrow \text{Ens}$	// insertion
\in	$\text{Nat} \cdot \text{Ens} \rightarrow \text{Bool}$	// appartenance
\forall	$\text{Ens} \rightarrow \text{Bool}$	// test de vacuité
$ _1$	$\text{Ens} \rightarrow \text{Nat}$	// cardinal (taille)
$-s$	$\text{Ens} \cdot \text{Nat} \rightarrow \text{Ens}$	// suppression
m	$\text{Ens} \rightarrow \text{Nat}$	// minimum

} Générateurs de base
("constructeurs")

} Sélecteurs ou fonctions d'accès

} Générateurs

préconditions $e: \text{Ens}$ $x: \text{Nat}$

$$\underline{\text{pré}} \quad i(e, x) = \exists x \in e \text{ et } |e| < 100$$

$$\underline{\text{pré}} \quad s(e, x) = x \in e$$

$$\underline{\text{pré}} \quad m(e) = \forall \forall (e)$$

axiomes $e: \text{Ens}$ $x, y: \text{Nat}$

$$(e\emptyset) \quad i(i(e, x), y) = i(i(e, y), x)$$

$$(e1) \quad x \in \emptyset = \text{faux}$$

$$(e2) \quad x \in i(e, z) = (x == z) \vee (x \in e)$$

$$(e3) \quad \forall (\emptyset) = \text{vrai}$$

$$(e4) \quad \forall (i(e, x)) = \text{faux}$$

$$(e5) \quad |\emptyset| = 0$$

$$(e6) \quad |i(e, x)| = 1 + |e|$$

$$(e7) \quad s(i(e, x), y) = \begin{cases} \text{si } (x == y) \text{ alors } e \\ \text{sinon } i(s(e, y), x) \text{ fin} \end{cases}$$

$$(e8) \quad m(i(e, x)) = \begin{cases} \text{si } \forall (e) \text{ alors } x \\ \text{sinon si } y < m(e) \text{ alors } y \\ \text{sinon } m(e) \text{ fin fin} \end{cases}$$

1.3 Programmation Fonctionnelle

→ Sous forme de fonctions laissant intacts leurs arguments après un appel.

Structure (sans pointeurs) : `typedef struct strens { Nat v[100]; Nat n; } Ens;`

1) Tant appelle de fonction n'a lieu que lorsque la précondition est vérifiée

```
main ()  
{  
    Ens e1, e2;  
    e1 = ensvide();  
    e2 = i(e1,3);  
}
```

passage par valeur

→ Sous forme de fonction laissant intacts leurs arguments après un appel

```
Ens ensvide () /* Ensemble Vide */  
{  
    Ens e;  
    e.n=0; /* Cardinal à zéro */  
    return e;  
}
```

```
Ens i (Ens e, Nat x) /*Insertion de x dans e */  
{  
    Ens e1 = e; /*recopie de e dans e1 */  
    e1.n = e.n+1 /*cardinal de e1 */  
    e1.v[e.n] = x; /*insertion */  
    return e1;  
}
```

```
Ens i (Ens e, Nat x)  
{  
    e.v[e.n] = x;  
    e.n++;  
    return e;  
}
```

```
Ens app (Ens e, Nat x)  
{  
    if (e.n==0)  
        return false;  
    else if (e.v[e.n-1]==x)  
        return vrai;  
    else  
    {  
        e.n--;  
        return app(e,x);  
    }  
}
```

← Programmation récursive.

• programmation itérative

```

Bool app (Ens e, Nat x)
{
    Nat k; /*position*/
    for (k=e.n; (0<k)&&(e.v[k-1]!=x); k--);
    ...
    return (k!=0)
}

```

• programmation intermédiaire

```

Bool app1 (Ens e, Nat x, Nat k) /* k: position telle que app1 est vraie si x est en position <= k-1 */
{
    if ( k==0)
        return faux;
    else if (e.v[k-1] == x)
        return vrai;
    else
        return app1(e,x,k-1);
}

```

```

Bool app (Ens e, Nat x)
{
    return app1(e,x,e.n);
}

```

2.2 (suite)

"appartenance"

Spec 1

$$\begin{cases} x \in \emptyset = \text{faux} \\ x \in i(e, y) = \begin{array}{l} \text{si } (x = y) \text{ alors vrai} \\ \text{sinon si } x \in e \text{ alors vrai} \\ \text{sinon faux fin fin} \end{array} \end{cases}$$

Spec 2
(recursive directe)

$$\begin{cases} x \in e \text{ si } \forall (e) \text{ alors faut} \\ \text{sinon } i((x \in e), y) \text{ fin} \end{cases}$$

1.3 (suite)

```
Bool v (Ens e) //Test de vacuité
{
    return e.n == 0;
}

Nat card (Ens e) //Cardinal
{
    return e.n
}

Ens s (Ens e, Nat x)
/* pré s(e,x) = app(x,e) */
{
    Nat k;
    for (k = e.n; e.v[k-1] != x; k--);
        e.v[k-1] = e.v[e.n-1]; // remplace x par le dernier élément
    return e;
}
```

→ Recherche le minimum d'un ensemble

```
Nat m (Ens e) // minimum d'un ensemble non-vide
{
    Nat p = e.v[e.n-1];
    Nat k;
    for (k = e.n-1; k>0; k--) // parcours de droite à gauche
        if ( e.v[k] < p )
            p = e.v[k];
    return p;
}
```

2.2 (suite)

axiomes Ens. e; Nat x;

e1 ($x \in \emptyset = \text{faux}$

e2 ($x \in i(e, y) = x = y \text{ ou } x \in e$

e3 ($v(\emptyset) = \text{vrai}$

e4 ($v(i(e, y)) = \text{faux}$

e5 ($| \emptyset | = 0$

e6 ($| i(e, y) | = 1 + | e |$

$i(\emptyset, z)$ contienne à la précondition $z \in e$.

e7 - $= \underline{\text{si}}(x = y) \underline{\text{alors}} e$
sinon $i(s(e, x), y)$

fin

Opérations

m Ens \rightarrow Nat

préconditions

pré m(e) = $\exists v(e)$

axiome

e8 $m(i(e, x)) = \underline{\text{si}} v(e) \underline{\text{alors}} x$

sinon ($\text{Si } m(e) < x \text{ alors } m(e)$
sinon $x = \text{fin}$
fin

Point d'étape énoncé, spécification informelle \leftrightarrow programmation des ensembles finis d'entiers

↓
 spécification formelle \leftrightarrow opérations préconditions

générateur de base \emptyset \in_i profil \equiv

axiomes

structuration horizontale /

raffinement vertical

exemple :

- axiomes

- version récursive

- version itérative

- chaque opération décrite plr aux généralisations de base (\emptyset, i)

- exclu les axiomes contraires aux pré-conditionnés.

pros

1.4 Rappel sur les pointeurs

(A) Simulation du passage par adresse

x comme paramètre dans une fonction peut être remplacé par &x pour réaliser un passage de x par adresse

t x; /* déclaration en C d'une variable x de type t */ , cette variable à une adresse

On peut également définir le type pointeur t * x, où x est de type pointeur.

(B) Allocation dynamique

Pour allouer des zones mémoire la fonction malloc est présente en C.

```
int *x;
x = (int *) MALLOC(sizeof(int));
```

→ pour alléger l'écriture :

```
#define MALLOC(type) ((type*)malloc(sizeof(type)))
```

```
#define MALLOCN(type, n) ((type*)malloc(n * sizeof(type)))
```

```
#define CALLOCN(type, n) ((type*)calloc(n, sizeof(type))) // Initialize les cases à 0
```

```
#define REALLOC(t, type, n) ((type*)realloc(t, n * sizeof(type)))
```

```
#define NULL (int *) 0; // initialization des adresses à NULL
```

```
#define FREE(t) free(t) // permet de rendre la mémoire portée par p
```

(C) Structure de données

...

1.5 Programmation par mutation

→ Reprenons la structure définissant les ensembles (1.3)

typedef struct strens {Nat v[100]; Nat x; } Ens;

→ Discutons des différentes formes d'écriture pour une programmation par mutation

La **Forme 1** qui remplace les Ens par des pointeurs Ens * est écrite avec des procédures
(= fonction qui ne renvoie rien : "void")

```
void ensvide(Ens *e)
{
    e->n = 0;
}
```

```
void i(Ens *e, Nat x)
{
    e->v[e->n] = x;
    (e->n)++;
}
```

```
void s(Ens *e, Nat x)
{
    Nat k;
    (e->n)--;
    for (k = e->n; e->v[k] != x; k --);
    e->v[k] = e->v[e->n];
}
```

```
main()
{
    Ens e;
    ensvide(&e);
    i(&e, 3);
    i(&e, 5);
    [...]
}
```

/+ K = place de
x dans
e → n = x/

La **Forme 2** travaille également avec des pointeurs sur Ens*, mais avec un aspect fonctionnel, comme aussi par mutation = modification des paramètres par effet de bord

```
Ens * Ensvide ()
{
    Ens *e; /* allocation dynamique */
    e = MALLOC (Ens);
    e->n = 0;
    return e;
}
```

```
Ens *i( Ens *e, Nat x) /* passage par adresse pour l'ensemble */
{
    e->v[e->n] = x;
    e->n++;
    return e;
}
```

```

Ens s (Ens *e, Nat x)
{
    Nat k;
    (e -> n) --;
    for (k= e-> n; e-> v[k] != x ; k --)
        e -> v[k] = e -> v[ e->n ];
    return e;
}

main()
{
    Ens *e;
    e= i(i(ensvide(),3),5);
    return 0;
}

```

/ même qu'avant sauf pour le return */*

Pour éviter de modifier les types entre l'usage “ fonctionnel ou par mutation”, dans la **Forme ③, le pointeur cette fois est dans la structure de données elle-même**

```
typedef struct strEns { Nat v[100]; Nat x;} StrEns, *Ens;
```

```

Ens ensvide()          /* Profil de la spécification */
{                      /* Programmation par mutation des ensembles */
    Ens e;
    e = MALLOC (StrEns);
    e -> n=0;
    return e;
}

Ens i (Ens e, Nat x)
{
    e-> v [e->n] = x;
    e -> n++;
    return e;
}

Ens s (Ens e, Nat x)
{
    Nat k;
    (e ->n) --;
    for (k = e->n ; e->v[k] != x ; k --);      /* k= place de x en sortie de for */
        e-> v[k] = en -> v [e ->n];
    return e;
}

main()
{
    Ens e;
    e = s( i( i( ensvide(), 3), 5), 7);
}      //affiche 3, 5

```

Bilan

Spécification d'un objet

ex: $\text{app} \left\{ (x \in e) \right.$

- quel(s) sont (les) générateurs de base permettant construire cet objet?

- imaginez le noyau d'opérations à définir P/x aux générateurs de base

- (a) spécification aux générateurs de base
- (b) spécification « direct »
prog.
- (c) spécification intermédiaires où on fait apparaître une variable du parcours (position)
lprog
- (d) spécification itérative
lprog-

La spécification est FONCTIONNELLE

passage naturellement à la PROG FONCTIONNELLE



prog avec MUTATION

forme 3: Pour ne pas modifier les profils par rapport à la forme fonctionnelle.

Chapitre 2: Piles, Files et Listes

→ Les structures linéaires

1. Piles

1.1 Énoncé formel ou spécification

Une pile est un contenu avec 3 opérations fondamentales :

- placer un objet sur la pile (empiler)
- accéder à l'objet au sommet de la pile
- retirer cet objet

Gestion d'une pile DAPS : "dernier Arrivé, Premier Sorti" / LIFO "Last In, First Out"

Pour la spécification, considérons de piles non bornées d'objets de sorte S (générique)

Rappel: La spécification de BASE contient des booléens, des entiers naturels, des rationnels, des réels, chaînes...

Bool Ent Nat Rat

Réel Char ...

Spec PILEO (TRIV) étend BASE

ou bien:

spec PILEO étend TRIV /* version simplifiée */

sous PILE /* pile non bornée d'objets de sorte S */

opérations

- pilenouv : $_ \rightarrow \text{Pile}$ /* création d'une nouvelle pile */
- empiler : $\text{Pile } S \rightarrow \text{Pile}$ /* empilement d'un él */
- sommet : $\text{Pile} \rightarrow S$ /* él du sommet */
- dépiler : $\text{Pile} \rightarrow \text{Pile}$ /* retire le sommet */
- vide : $\text{Pile} \rightarrow \text{Bool}$ /* Test de vacuité */
- hauteur : $\text{Pile} \rightarrow \text{Nat}$ /* hauteur de la pile */
- remplace : $\text{Pile } S \rightarrow \text{Pile}$ /* remplace le sommet */

préconditions: p: Pile ; xc : S

pré : sommet(p) = \neg vide(p)

pré : dépiler(p) = \neg vide(p)

pré : remplacer(p) = \neg vide(p)

axiomes: p: Pile, xc : S

(p1) : vide(pilenouv()) = vrai

avec
pilenouv()

(p2) : hauteur(pilenouv()) = 0

avec
pilenouv()

(p3) : vide(empiler(p, xc)) = faux

(p4) : hauteur(empiler(p, xc)) = hauteur(p) + 1

(p5) : sommet(empiler(p, xc)) = xc

(p6) : dépiler(empiler(p, xc)) = p

(p7) : remplacer(empiler(p, xc), y) = empiler(p, y)

(p8) : remplacer(p, y) = empiler(dépiler(p), y) /* variante de spéc directe */

fspec

Pile: Opération renvoyant une pile ; c'est un générateur ou constructeur
les autres sont des sélecteurs ou fonction d'accès.
Générateurs de base? : pilenav et empiler: empiler(empiler(pilenav(), a), b) c

→ Discussion de variantes

- Piles bornées: Il nous faut une borne strictement positive

Spéc BSAP étend BASE

opérations

N: Nat / \neq borne \times /

axiomes

(p0) $0 < N \leq 100 = \text{vrai}$

fspéc

Et je change la spéfic.

Spéc PILE1 (TRIV) étend BSAP

préconditions

pré: empiler (p, z) = hauteur (p) $< N$

- Piles d'entiers: plutôt génériques que de sorte

Spéc PILENAT étend PILE0 ($\text{TRIV} \rightarrow \text{NAT}$ avec $S \rightarrow \text{NAT}$) fspéc

1.2 Implantation contigüe et programmation fonctionnelle

Je considère des piles bornées implantées sur une structure contenant un tableau v des éléments de sorte S et la hauteur h de la pile.

```
# define N 100
```

```
typedef struct { S v[N]; Nat h; } Pile;
```

"tableau statique"

On choisit un tableau "statique" qui convient pour des piles bornées.
On réfléchira à des tableaux "dynamiques" pour des piles non bornées.

→ Piles bornées

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

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

```
S sommet (Pile p) { return p.v[ p.h -1 ]; }
```

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

```
Nat hauteur (Pile p) { return p.h; }
```

```
Pile depile(Pile p)  
{  
    p.h = p.h-1;  
    return p;  
}
```

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

Variante qui semble plus fonctionnelle :

```
Pile empiler ( Pile p, S x)  
{  
    Pile p1 = p; ← Copie des arguments  
    p1.v[ p.h ] = x;  
    p1.h = p1.h + 1;  
    return p1;  
}
```

Faites automatiquement
en langage C.

```
main ()  
{  
    pile p0 = pilenouv();  
    pile p1 = empiler(p0,2);  
    // ...  
}
```

Version dynamique :

CF. TD

- Déclaré $S *v;$
- Alloué dynamiquement avec MALLOC et REALLOC.
- Faut-il stocker la taille du tableau?

1.3 Implantation contigüe et Programmation par mutation

→ simuler le passage par adresse pour éviter la réécriture de la structure.

FORME 1 : Par Mutation

```
void pilenouv(Pile *p)
{
    p->h = 0;
}
```

```
void emplier(Pile *p, S x)
{
    p->v[ p->h ] = x;
    p->h++;
}
```

à finir pour les autres...

FORME 2 : Par Mutation avec Forme fonctionnelle

```
Pile *pilenouv ()
{
    Pile *p = MALLOC (Pile);
    p->h = 0;
    return p;
}
```

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

FORME 3 : Par mutation sans changement du profil de la spécification . Je modifie la SD pour cacher le pointeur

```
typedef struct { S v[N]; Nat h} Pile; F 1.
typedef struct { S v[N]; Nat h} StrPile; F 2.
typedef struct { S v[N]; Nat h} StrPile, *Pile; F 3.
```

```
Pile pilenouv()
{
    Pile p= MALLOC(StrPile);
    p->h = 0;
    return p;
}
```

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

/* Effet de bord lors des appels de fonctions génératrices ou constructrices */

1.4 Implémentation par chaînage

Pour ranger les éléments d'une pile dans chaque emplacement, il nous faut une place pour l'élément de sorte S et une place pour le chaînage.

```
typedef struct strpile { S v; struct strpile *s; } StrPile, *Pile;
```

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

```
Pile empiler (Pile p, S x)
{
    Pile p1 = MALLOC (StrPile);
    p1->v = x;
    p1->s= p;
    return p1;
}
```

```
S Sommet (Pile p)
{
    return p->v;
}
```

```
Pile depiler_fonctionnel(Pile p)
{
    return p->s;
}
```

```
Pile depiler_mutation (Pile p)
{
    Pile t = p;
    p = p->s;
    free (t);
    return p;
}
```

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

```
Nat hauteur (Pile p)
{
    return vide(p)?0: hauteur (p->s+1);
}
```

```
/* Variante itérative */
```

```
Nat hauteur (Pile p)
{
    Nat h;
    for (h=0; !vide(p); h++, p=p->s);
    return h;
}
```

A éviter!

2. Files

2.1 Formalisation ou spécification

FILES : c'est un conteneur d'objets avec 3 opérations fondamentales:

- ajouter un élément en queue de file
- { - accéder à l'objet en tête de file
- Supprimer "

Premier arrivé, premier sorti = PAPS

First In, First Out = FIFO

forme des files:

adjq(... adjq (filenouv(), x0) ...)

Exemple de files génériques (de sorts) non bornées

spec FILE0 (TRIV) étend BASE

sorte FILE /t files non bornées */

Opérations:

générateurs { filenouv : _ → file /* file vide */
 de base } adjq : File S → File /* ajout en queue */
 supt : File → File /* suppression en tête */
 tête : File → S /* él^t en tête */
 vide : File → Bool /* vacuité */
 lgr : File → Nat /* longueur ou nb d'él^ts */

préconditions f: File

pr_e tête (f) = \neg vide (f)

pr_e supt (f) = \neg vide (f) /* variante possible sans */

axiomes f:file ; x:S

(f₁) : vide (filenouv()) = vrai

(f₂) : lgr (filenouv()) = 0

(f₃) : vide (adjq(f, x)) = faux

(f₄) : lgr (adjq(f, x)) = lgr (f) + 1

(f₅) : tête (adjq(f, x)) = si vide (f) alors x (récursivité)
sinon tête (f) f si

(f₆) : supt (adjq(f, x)) = si vide (f) alors filenouv()

sinon adjq(supt(f), x) f si

Variante:

supt (adjq (filenouv, x)) = filenouv

supt (adjq (adjq (f, g), x)) = adjq (supt (adjq (f, g), x))

Espéc

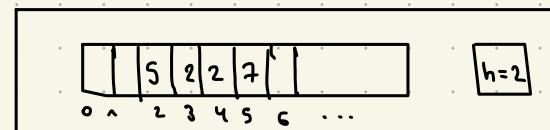
2.2 Représentation par contiguous.

- Représentation de la file par un tableau de N éléments et 2 entiers :

- h pour l'indice de tête
- l pour la longueur de la file
(variante possible avec q indice de queue)

$$q = \begin{cases} (h+l-1) \bmod N & \text{ou bien} \\ 1^{\text{ère}} \text{ case libre } (h+l) \bmod N. \end{cases}$$

- Pour une succession d'adj q et de supt,
La file peut être scandée en deux parties :
 - le début de la file en fin de tableau.
 - la fin de la file en début de tableau.



* allocation statique :

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

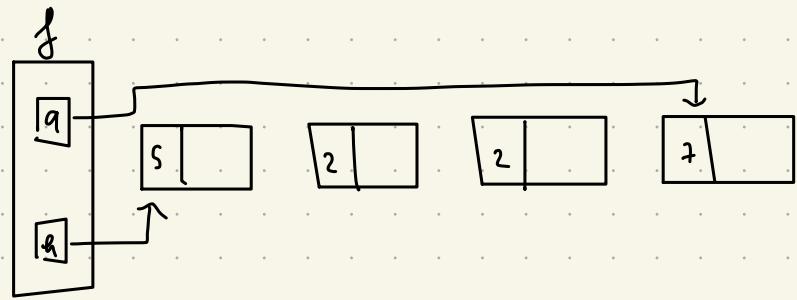
* allocation dynamique :

"file non bornée"

```
typedef struct strfile { S v[]; Nat h; Nat l; nat taille; } File;
```

2.3 Représentation par chaînage

Une file est composée de cellules mémoire chaînées avec par exemple, deux pointeurs : h (head) et q (queue) sur la tête et la queue de la file.

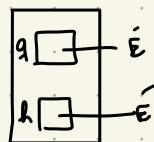


```
typedef struct strfile { S v; struct strfile * s;} Strfile;
typedef struct file { Strfile *h, Strfile *q;} File;
```

Méthodologie pour les opérations :

1. dessin en couleurs des actions pour chaque opération
2. écriture de la fonction en numérotant les actions qui deviennent des instructions
3. vérification du C par le dessin et par une troisième personne.

• Filenouv

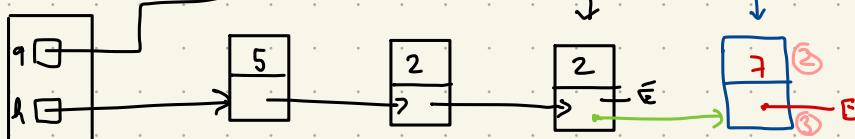


File filenouv()

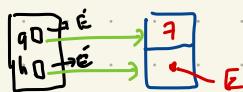
```
{
  File f;
  f.q = NULL;
  f.h = NULL;
  return f;
}
```

• Adjq (F, 7)

①



cas particulier :
filenew()



② File adjq(File f, S x)

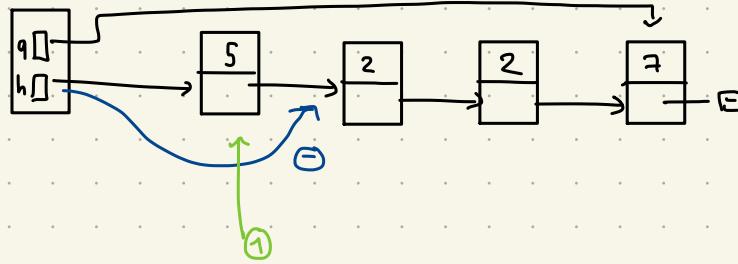
```

{
    Strfile * f1 = MALLOC (Strfile);
    f1 -> v = x;
    f1 -> s = NULL;
    if ( f.h == NULL) ③
    {
        f.h = f1;
        f.q = f1;
    }
    else
    {
        f.q -> s = f1; ④
        f.q = f1; ⑤
    }
    return f;
}

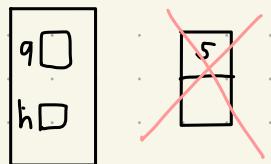
```

• Supt (F)

cas général



cas particuliers



File supt (File f)

```

{
    Strfile * f1 = f.h; ①
    if ( f.h == f.q ) /* file à 1 élmt */
    {
        f.h = NULL;
        f.q = NULL;
    }
    else
        f.h = f.h->s; ②
    free (F1); ③
    return f
}

```

• Autres fonctions :

S tête (File f) { return f.h -> v; }

Bool vide (File f) { return f.h == NULL; }

→ Opération intermédiaire pour la fonction longueur compte tenu de la différence de type entre f et f.h.

```

Nat lgr (File f) {
    return vide (f)?0 : lgrb (f.h, f.q);
}

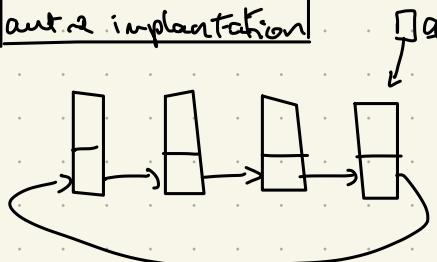
```

```

Nat lgrb (Strfile * f1, Strfile * f2){
    return f1 == f2 ? 1 : 1 + lgrb (f1 -> s, f2);
}

```

→ autre implantation



→ Ou à la place de NULL, on place en queue l'adresse de la cellule de tête et du coup, on a plus besoin que du pointeur de queue, qui est la file f, elle-même et la seule déclaration est :

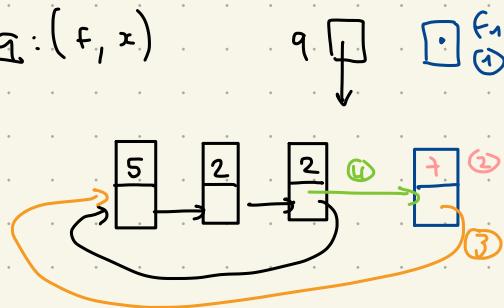
```
typedef struct strfile { S v; struct strfile * s; } StrFile, * File;
```

File filenou ()

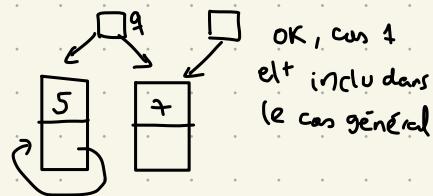
```
{
    return NULL;
}
```

→ Variante si on veut stocker la longueur de la file
→ en exercice.

- Adjq : (f, x)



Ces particuliers :



Fle adjq (File f , S x)

```
{
    File f1 = MALLOC (StrFile);
    f1 -> v = x;
    if ( vide (f) )
        f1 -> s = f1;
    else
    {
        f1 -> s = f -> S;
        f -> s = f1;
    }
    return f1;
}
```

- Autres fonctions :

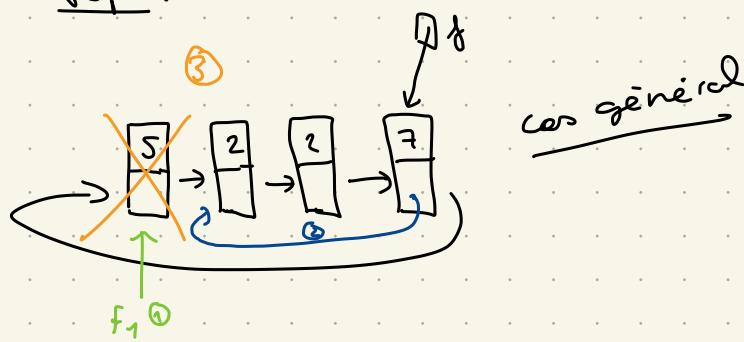
S tête (File f) { return f -> s -> v; }

Bool vide (File f) { return f == NULL; }

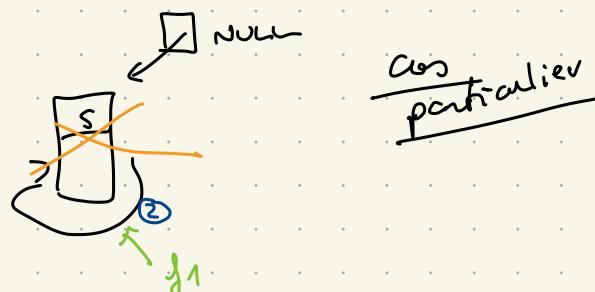
Nat lgrb = idem

```
Nat lgr (File f) {
    return vide (f) ? 0 : lgrb (f ->s ; f );
}
```

Supr :



cas général



cas particulier

File supt (File f)

```

File f1;
if (f == f -> s)
{
    FREE (f);
    f = NULL;
}
else
{
    f1 = f -> s;
    f -> s = f -> s -> s;
    FREE (f1);
}
return f;
/* cas 1 élément */

```

3. Listes

IS-OCT-19

3.1. Spécification

Déf: Une liste linéaire sur un ensemble E d'éléments est une suite finie x_1, x_2, \dots, x_k d'éléments de E notée (x_1, x_2, \dots, x_k) .

Ce sont des opérations qui précisent l'usage de ces listes.

exemple: listes génériques non bornées de sorte S.

spéc LISTEO (triv) étend BASE

sous LISTE

opérations

gen. de base {
 listenouv : $__ \rightarrow$ Liste /* Liste nulle, notée aussi \wedge */
 adjt : Liste S \rightarrow Liste /* ajout d'un él. en tête */
 supt : Liste \rightarrow Liste /* suppr 1^{er} tête */
 tête : Liste \rightarrow S /* él. tête */
 vide : Liste \rightarrow Bool /* test de vacuité */
 lgr : Liste \rightarrow Nat /* longueur */.
préconditions : l:liste ; x:S

pré tête : \neg vide (l)

pré supt : \neg vide (l)

/* l'axiome supt (listenouv) = listenouv peut remplacer la précond. sur supt */.

axiomes : l: liste ; x:S

(l1) vide (listenouv) = vrai

(l2) lgr (listenouv) = 0

(l3) vide (adjt(l,x)) = faux

$$(14) \text{ lgr}(\text{adjt}(l, x)) = \text{lgr}(l) + 1$$

$$(15) \text{ fetc}(\text{adjt}(l, z)) = z$$

$$(16) \text{ supt}(\text{adjt}(l, x)) = l.$$

fspec

Nous allons enrichir la sorte S générique pour pouvoir compléter ce noyau d'opérations. En premier lieu, ajoutons l'égalité $=$ sur les élts de sorte S.

spéc EG étend TRIV

opérations

$$=: S \times S \rightarrow \text{bool} \quad /* \text{égalité} */$$

$$!= S \times S \rightarrow \text{bool} \quad /* \text{différence} */$$

axiomes

$$e_1 (x == x) = \text{vrai} \quad /* \text{reflexivité de } == */$$

$$e_2 (x != y) = \neg(x == y)$$

Fspec

On enrichit la spécification LISTE0 avec des opérations utilisant l'égalité sur les élts de sorte S.

spéc LISTE1 (EG) étend LISTE0 (TRIV \rightarrow EG)

$/*$ paramétrage pour la sorte S définie par EG.

et étend les LISTE0 en remplaçant TRIV par EG $*/$

EG = Égalité

opération:

$$\text{app} : \text{Liste } S \rightarrow \text{Bool} \quad /* \text{app. d'un elt */}$$

$$\text{conc} : \text{liste liste} \rightarrow \text{liste} \quad /* \text{concaténation */}$$

précond:

axiomes $/*$ Par rapport aux gér. de base $*/$ $l, l_1, l_2 : \text{liste}; x, y : S;$

$$\text{app}(\text{listenouv}, x) = \text{faux}$$

$$\text{app}(\text{adjt}(l, y), x) = x == y \vee \text{app}(l, x) \quad ? /* \text{ si } x == y \text{ alors vrai sinon app}(l, x) */$$

première version

$$\text{conc}(\text{listenouv}, \text{listenouv}) = \text{listenouv}$$

$$\text{conc}(\text{adjt}(l_1, x), \text{listenouv}) = \text{adjt}(l_1, x)$$

$$\text{conc}(\text{listenouv}, \text{adjt}(l_2, y)) = \text{adjt}(l_2, y)$$

$$\text{conc}(\text{adjt}(x, l_1), \text{adjt}(l_2, y)) = \text{adjt}(\text{conc}(l_1, \text{adjt}(l_2, y)), x).$$

$$\boxed{x} \left(\boxed{l_1} \boxed{y} \boxed{l_2} \right)$$

deuxième version

$$\text{conc}(\text{listenouv}, l_3) = l_3$$

$$\text{conc}(\text{adjt}(l, x), l_3) = \text{adjt}(\text{concat}(l, l_3), x)$$

troisième version

(à compléter plus tard)

$$\text{conc}(l, \text{listenouv}) = l$$

$$\text{conc}(l, \text{adjt}(l_2, y)) = \text{conc}(\text{adjt}(l, y), l_2)$$

$$\boxed{l} \quad \boxed{y} \quad \boxed{l_2}$$

↑ on a besoin d'opérations en queue de liste.

/* OPÉRATIONS SUR LE DERNIER ÉLÉMENT */

opérations:

queue: Liste \rightarrow S /* dernier élément */
adjt : Liste \rightarrow S /* ajout en queue */
Supq : Liste \rightarrow Liste /* suppression du dernier élément */

préconditions:

pré : queue (l) = \neg vide (l)
pré : Supq (l) = \neg vide (l)

} la précondition ou l'axiome

axiomes:

(a1) supq (listenouv) = listenouv

(a2) queue (adjt (l, x)) = si vide (l) alors x sinon queue (l) fin
// variante de spécification "directe":

(a2') queue (l) = l [lgr (l)] ou ième (l, lgr (l)) (accès au ième él^e)



(a3) adjq (listenouv, x) = adjt (listenouv, x)

(a4) adjq (adjt (l, y), x) = adjt (adjq (l, x), y)

(a5) supq (adjt (l, x)) = si vide (l) alors listenouv
sinon adjt (supq (l), x) fin

// variante à deux axiomes à la place du précédent:

(a5') supq (adjt (listenouv, x)) = listenouv

(a5'') supq (adjt (adjt (l', y), x)) = adjt (supq (adjt (l', y)), x)

// ou bien, changement de variables amènes:

(a5''') supq (adjt (adjt (l, x), y)) = adjt (supq (adjt (l, x)), y)

// variante de la spécification directe:

(a3' & a4') adj (l, x) = si vide (l) alors adjt (listenouv, x)
sinon adjt (adjq (supq (l), x), tête (l)) fin

(a5''') supq (l) = si lgr (l) == 1 alors listenouv

sinon adjt (supq (supq (l), tête (l))) fin

remarque: Une programmation à l'identique est possible, mais inefficace en temps de mémoire (cf.. Chapitre 4).

* OPÉRATIONS SUR LE i ÈME ÉLÉMENT *

opérations

- $[-]$: $\text{Liste Nat} \rightarrow S$ /* i -ème élmt noté aussi i ème (l, i) ou encore $\text{el}(l, i)$ */
- lième : $\text{Liste Nat} \rightarrow \text{liste}$ /* sous-liste débuteant au i -ème élmt */
- ins : $\text{Liste Nat } S \rightarrow \text{liste}$ /* insertion en i -ème pos */
- sup : $\text{Liste Nat} \rightarrow \text{liste}$ /* suppression de l'élmt en i -ème position */
- chg : $\text{Liste Nat } S \rightarrow \text{liste}$ /* remplacement du i -ème élmt */
- lig : $\text{Liste Nat} \rightarrow \text{liste}$ /* sous-liste terminant au i -ème élmt sans le i -ème */
- lid : $\text{Liste Nat} \rightarrow \text{Liste}$ /* sous-liste commençant au i -ème élmt - $l[i]$ */

préconditions $l: \text{liste}, i: \text{Nat}$

pré $l[i] = \emptyset \wedge i < \text{lgr}(l)$ // ceci implique $\emptyset < \text{lgr}(l)$ et donc $l \neq \emptyset$

pré $\text{lième}(l, i) = i \geq 1$ // possible aussi sans précondition et si jamais $i = \emptyset$, on retourne l

axiomes

(a1) $\text{adjt}(l, x)[i] = \begin{cases} x & \text{si } i = 1 \\ \text{lième}(l, i-1) & \text{sinon} \end{cases}$ fsi

// variante par rapport au générateur de base de Nat

(a1') $\text{adjt}(l, x)[\text{succ}(\emptyset)] = x$

(a1'') $\text{adjt}(l, x)[\text{succ}(\text{succ}(i))] = l[\text{succ}(i)]$

min 2

// variante de la spécification "directe" de $[]$:

(a1''') $l[i] = \begin{cases} \text{tête}(l) & \text{si } i = 1 \\ \text{supr}(l)[i-1] & \text{sinon} \end{cases}$ fsi

ou bien

(a1''') $l[i] = \begin{cases} \text{queue}(l) & \text{si } i = \text{lgr}(l) \\ \text{supr}(l)[i] & \text{sinon} \end{cases}$ fsi

// version itérative

(a1^{5x'}) $l[i] = \text{tête}(l_1)$

avec $(l_1, k) = \text{init}(l, 1)$ tant que $(k < i)$ rép $(\text{supr}(l_1), k+1)$ frép

ou bien

(a1^{6x'}) $l[i] = \text{queue}(l_1)$

avec $(l_1, k) = \text{init}(l, \text{lgr}(l))$ tant que $(k > i)$ rép $(\text{supr}(l_1), k-1)$ frép

(a2) $\text{lième}(\text{listenouv}, i) = \text{listenouv}$ // nécessaire comme cas d'arrêt récursivité lorsque $i > \text{lgr}(l)$
 $\text{lième}(\text{adjt}(l, x), i) = \begin{cases} \text{adjt}(l, x) & \text{si } i = 1 \\ \text{lième}(\text{lième}(l, i-1), i) & \text{sinon} \end{cases}$ fsi

Exemple: • $\text{lième}((x_1, x_2, x_3), 5) = \text{lième}((x_2, x_3), 4) = \text{lième}((x_3), 3)$
= $\text{lième}((), 2)$
= $()$

+ imaginer des opérations sur la 1^{ère} occurrence d'un élément.

3.2 Ordre sur les éléments des listes triées

Déf: Un préordre sur un ensemble E est une relation binaire dans E notée \leq , réflexive et transitive.

La notation notée \sim définie par $x \sim y = x \leq y \text{ et } y \leq x$ est une relation d'équivalence

Spéc PREORD étend TRIV

opérations

$\leq, <, \sim : S \times S \rightarrow \text{Bool}$ /* préordre large: \leq , strict: $<$, équivalence: \sim */

axiomes:

- (a1) $x \leq x = \text{vrai}$ /* réflexivité */
- (a2) $(x \leq y \text{ et } y \leq z) \Rightarrow (x \leq z = \text{vrai})$ /* transitivité */
- (a3) $x < y = x \leq y \text{ et } \forall z \leq x \quad z \leq y$ /* définition de $<$ */
- (a4) $x \sim y = x \leq y \text{ et } y \leq x$ /* équivalence */

- Exemple:

→ Prenons pour E le type Personne, déclarée en C :

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

→ Nous définissons le préordre \leq pour tout couple (x, y) :

$$x \leq y \text{ si } x.\text{nom} \leq y.\text{nom}$$

où \leq est l'ordre habituel des chaînes caractères

Permet de constituer une liste triée de Personne.

exemple (suite):

$$x = ("Dupont", 123)$$

$$y = ("Dupont", 456)$$

→ On a $(x \leq y)$ et $(y \leq x)$ et pourtant, $(x == y)$ est faux, c'est bien un préordre mais pas un ORDRE



- Définition: Un ordre est un préordre antisymétrique c.à.d que l'antisymétrie est vérifiée : $(x \leq y \text{ et } y \leq x \Rightarrow x = y) = \text{vrai}$.

Dans ce cas, les opérations \sim et $=$ sont identiques :

spec ORD étend PREORD, EG

axiomes

(a5) $x \sim y = x = y$ /* équivalence = égalité */

fspéc

- Définition: Une relation binaire \leq sur E est totale si, pour tout couple (x, y) de $E \times E$, $x \leq y$ ou $y \leq x$.
→ Alors 2 éléments de E sont toujours comparables.

→ Pour définir un préordre total,

Spéc PREORDT étend PREORD

axiomes

(a6) $x \leq y \vee y \leq x = \text{vrai}$

fspéc

→ Pour définir un ordre total

Spéc ORD étend ORD

axiomes

(a6) $x \leq y \vee y \leq x = \text{vrai}$

fspéc

→ le plus utile en général :

Spéc PREORDT EG étend PREORD

fspéc

À retenir, avant de trier des élts de sorte S (Ensemble E), définir les opérations de comparaisons disponibles.

3.3 Les listes triées

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

Pour la spécification, nous voulons les opérateurs suivants: \leq satisfaisant un préordre total. Nous voulons également $=$ et \neq . Ceci est offert par la spécification PREORDTEG

Spéc LISTET (PREORDTEG) étend LISTE1 ($E \rightarrow \text{PREORDTEG}$)

sous-sorte: Listet

sous-sous-sorte: Listet \leq Listé

préconditions: lt : Listé ; $x : S$

pré: adjt (lt, x) = si vide (lt) alors vrai sinon tête (lt) fin

// Attention: ajout d'autres précond. pour utiliser certaines opérations sur des listes triées conc, adjt, ...

opérations:

triée : Liste \rightarrow Bool /* test de tri d'une liste normale */

Extension des 2 générateurs des listes aux listes triées

listenouv : _ \rightarrow Listé

adjt : Liste S \rightarrow Listet

insav : Liste S \rightarrow Listé

insap : Liste S \rightarrow Listet

rech : Liste S \rightarrow Nat

/* adjt juste avant les éléments \geq */

/* adjt juste après les éléments \leq */

/* plus petit rang d'1 élément */

Axiomes: lt : Listet ; l : Listé ; $x, y : S$.

(lt0) triée (lt) = vrai /* invariante des listes triées */

(lt1) triée (listenouv) = vrai

(lt2) triée (adjt (listenouv, x)) ?

(lt3) triée (adjt ((l, y), x)) = $x \leq y \wedge$ triée (adjt (l, y)) * voir technique

(lt4) insav (listenouv, x) = adjt (listenouv, x)

(lt5) insav (adjt (lt, y), x) = si $x \leq y$ alors adjt (adjt (lt, y), x)
sinon adjt (insav (lt, x), y) fin

(lt6) insap (listenouv, x) = adjt (listenouv, x)

(lt7) insap (adjt (lt, y), x) = si $x < y$ alors adjt (adjt (lt, y), x)
sinon adjt (insap (lt, x), y) fin

/* rang de 1 à lgr(lt) et \emptyset si l'elt n'y est pas */

(lt8) rech (listenouv, x) = 0

(lt9) rech (adjt (lt, y), x) = si $x < y$ alors \emptyset
sinon si $x = y$ alors 1

sinon 1 + rech (lt, x)

avec $i = \text{rech} (\text{lt}, x)$

si $i = \emptyset$ alors \emptyset sinon $i + 1$ fin

fin

fin

Remarque sur les listes triées: Si on veut utiliser uniquement des listes triées, il vaut mieux cacher l'adjt et n'utiliser que listenouv, insav et insap.

4. Implantation des Listes

4.1 Par contiguïté

→ Comme pour les files, il nous faut 2 paramètres car la liste peut commencer n'importe où dans le tableau, et finir aussi n'importe où dans le tableau.

- indice tête de la liste
- indice queue ou bien longueur

→ Les opérations en tête et en queue sont aisées, qu'en est-il des opérations en milieu de liste ?

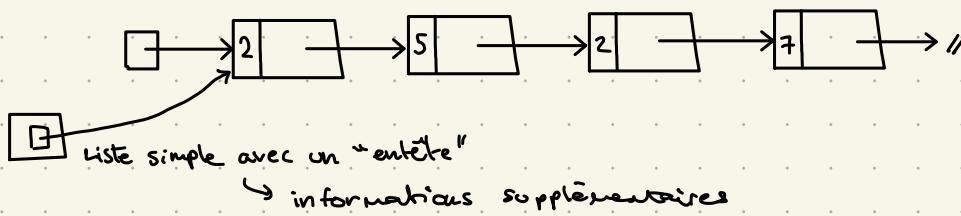
2 solutions:

- décalage : coûteux en temps de calcul
- marquage des cases libérées et les réutiliser ensuite : ce qui rend la gestion plus compliquée

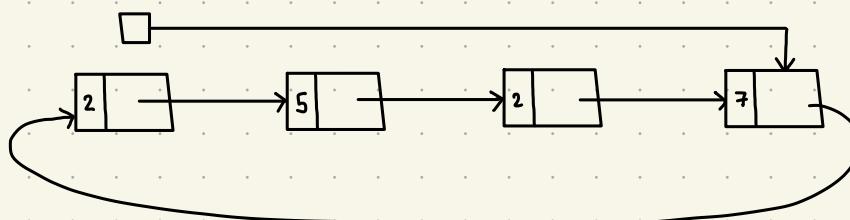
4.2 Par chaînage

```
typedef struct Strliste { Sx; struct Strliste *s; } Strliste, *Liste;
```

- Liste simple avec un pointeur sur la tête : {2,5,2,7} :

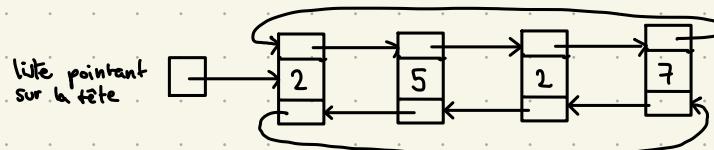


- Liste circulaire:



Les opérations adjg sont facilitées, mais Δ supq nécessite un parcours..

- Liste doublement circulaire



Liste circulaire chaînée dans les deux sens.
"Liste symétrique, bidirectionnelle, bilatérale"

5. Recherche dans une liste triée

a) Recherche séquentielle

• De gauche à droite :

$\text{rech}(l, x) = \text{rsqg}(l, x, 1, \text{lgr}(l))$
avec $\text{rsqg}(l, x, i, n) = \begin{cases} \text{si } (i == n) \vee (x \leq l[i]) \\ \text{alors} \\ \quad \text{si } x == l[i] \text{ alors } i \text{ sinon } \emptyset \text{ fin} \\ \text{sinon} \\ \quad \text{rsqg}(l, x, i+1, n) \text{ finrsqg} \end{cases}$

• La version itérative :

$\text{rech}(l, x) = \begin{cases} \text{si } i == l[i] \text{ alors } i \text{ sinon } \emptyset \text{ fin} \\ \text{avec: } i = \text{init } 1 \text{ tq } (i < n) \wedge (x > l[i]) \\ \quad \text{rep } i + 1 \text{ frép} \\ \text{avec: } n = \text{lgr}(l) \end{cases}$

b) Recherche dichotomique

/ * tous distincts */

$\text{rech}(l, x) = \text{rdich}(l, x, 1, \text{lgr}(l))$
avec $\text{rdich}(l, x, a, b)$
 $\text{Si } l[m] == x \text{ alors } m$
 $\text{sinon si } x > l[m] \text{ alors } \text{rdich}(l, x, m+1, b)$
 $\quad \text{sinon } \text{rdich}(l, x, a, m-1) \text{ fin rdich}$
avec $m = \lfloor (a+b) / 2 \rfloor$ plancher

/ * pas forcément tous distincts */

$\text{rdich}(l, x, a, b) :$ $\begin{cases} \text{si } (a == b) \\ \text{alors si } l[a] == x \text{ alors } a \text{ sinon } \emptyset \text{ fin} \\ \text{sinon} \\ \quad \text{si } (x > l[m]) \text{ alors } \text{rdich}(l, x, m+1, b) \\ \quad \text{sinon } \text{rdich}(l, x, a, m) \\ \quad \text{fin} \\ \text{fin} \\ \text{avec } m = \lfloor (a+b) / 2 \rfloor \end{cases}$

Chapitre 3: Suite

3. Propriétés d'une spécification

3.1 Complétude hiérarchique

La spécification est complète si elle n'introduit pas d'éléments parasites (junk) dans les spécifications importées (ex: dans Nat à partir de ENS \emptyset).

Exemple: Supposons que dans ENS \emptyset nous ayons omis l'axiome

$$(e5) |\emptyset| = 0$$

$$|-| : \text{Ens} \rightarrow \text{Nat}$$

On a alors fabriqué un nouveau terme $|\emptyset|$ différent de tout autre terme de sorte Nat.

→ en réalité, une infinité de nouveaux termes $|\emptyset| + 1$

3.2 Consistance hiérarchique

La spécification est consistante si elle n'introduit pas d'identification de valeurs ou confusions dans les spécifications importées.

Exemple: Supposons que l'axiome suivant soit malencontreusement introduit

$$(e1\emptyset) x \in i(\emptyset, 0) = \text{vrai}$$

Cet axiome, pour les techniques de preuves, conduit à Faux = vrai qui n'est clairement pas un théorème des Bool. Alors ENS \emptyset ne serait pas consistante hiérarchiquement récursivement à Bool.

Autre exemple de confusion:

$$e(11) |i(\emptyset, 0)| = 0$$

Cet axiome introduit une confusion dans Nat car $|i(\emptyset, 0)| = 0$ devient un théorème des ENS \emptyset alors que c'est faux dans Nat.

3.3 Indépendance ou non redondance

Un axiome ne doit pas être parallèle à partir des autres.

Exemple: $(e12) |i(\emptyset, x)| = 1$ rend la spécification redondante car $(e12)$ peut être déduit à partir de $(e5)$ et $(e6)$.

Autre exemple: le rajout de tout théorème déductif est indicatif comme axiome, créer une redondance

$$(t5) |s(e, z)| = |e| - 1$$

si $x \in e$

4. Validité d'une implantation

- On vérifie qu'une implantation (fonctionnelle) valide la spécification.
- Pour s'en assurer, il faut prouver que chaque axiome est validée par l'implantation.
- Ici, nous allons faire ses preuves de programmes «à la main». Celles-ci peuvent être automatisées.

Définition: Un programme termine s'il ne tombe pas dans une boucle infinie, pour aucune donnée. Dans le cas où il termine, il est correct si il valide la spécification.

4.1 Terminaison

- à partir de l'implantation fonctionnelle des ensembles

- ex1: Ensvide()

preuve: Ne contient pas d'itérations, termine nécessairement.

- ex2: app₁(e, z) = app₁(e, z, e.n)

```
Bool app1 (Ens e, Nat z, Nat k) {  
    /* vrai ssi z est en position <= k-1 de e.v */  
    if (k == φ) return faux;  
    else if (e.v[k-1] == 1) return vrai;  
    else return app1(e, z, k-1);  
}
```

preuve: • Montrons que l'appel app₁(e, z, e.n) termine à k avec $0 \leq k \leq e.n \leq 100$.

• Raisonnement par récurrence sur le paramètre K de la récursivité.

→ 1^e cas initial: $K = \emptyset$, alors app₁(e, z, φ) termine en renvoyant faux

→ 2^e cas général: L'hypothèse de récurrence est que app₁(e, z, k-1) termine. Alors, montrons que app₁(e, z, k) termine.

• Deux sous cas sont à envisager:

→ a) cas $e.v[k-1] == z$: l'exécution termine et renvoie vrai

→ b) cas $e.v[k-1] != z$: l'exécution se poursuit avec l'appel récursif app₁(e, z, k-1), qui termine d'après l'if.

- ex3: s(e, z), en C cette fonction contient l'itération

```
for (K = e.n ; e.v[K] != 0 ; K--) ;
```

preuve: La condition d'arrêt $e.\text{v}[k] == x$ se produit toujours à cause de

la précondition :

$$\text{P.R.} : s(e, x) = x \in e$$

→ l'examen de app (ou app1) montre que ceci ne produit sauf si il existe k , $0 \leq k \leq e.\text{n}-1$ tel que $e.\text{v}[k] == x$. C'est exactement la condition d'arrêt.

4.2 Correction

Pour chaque axiome, il faut regarder si l'implémentation le n'aide = c'est alors partiellement correct

- ex1: $(e1) x \in \emptyset = \text{Faux}$

Pour prouver que cet axiome est valide, il faut se reporter à la programmation de EnsN'aide et de app et il faut montrer que $\text{app}(x, \emptyset) = \text{Faux}$.

preuve: La programmation de ensn'aide définit $e.\text{n}=0$ et dans ce cas, l'appel $\text{app}(e, x)$ renvoie faux.

- ex2: $(e2) x \in i(e, y) = x == y \text{ ou } x \in e$

Se reporter à la prog. de l'insertion i et de app. et montrer que:

$$\text{app}(i(e, y), x) = x == y \text{ || app}(e, x)$$

preuve: → Posons $e1 = i(e, y)$ et réécrivons le corps de app1.

→ Suite à l'appel $\text{app}(e1, x)$ en remplaçant e par $e1$.

```
app1(e1, x, e1.n) {  
    if (e1.n == 0) return faux;  
    else if (e1.n - 1 == x) return vrai;  
    else return app1(e1, x, e1.n - 1);  
}
```

Voir
la fct on ch1

→ Compte que $e1 = i(e, y)$, d'après l'insertion i on a $e1.n = e.n + 1 > 0$. donc le 1^{er} cas de app1 n'est pas considéré.

→ Toujours d'après i, $e1.\text{v}[e.n]$ est égal à y et les tableaux $e.\text{v}$ et $e1.\text{v}$ sont égaux sur la tranche $[0, e.n - 1]$, donc app1 devient:

```
{ if (e1.\text{v}[e.n] == x) return vrai;  
    else return app1(e, x, e.n);  
}
```

→ Je réécris app1 {
 if ($y == x$) return vrai;
 else return app(e, x);
}

□

Conclusion sur la validation (partie 4) :

- Une implantation qui vérifie tous les axiomes d'une spécification en est un modèle.
- Une implantation qui n'est pas un modèle, peut réflire pourvu qu'elle soit un comportement acceptable pour l'usage qui en est fait, et à condition de savoir exactement quels axiomes sont relâchés.

Chapitre 4 : Complexité ou analyse de programmes

Objectif: évaluer le temps d'exécution et l'espace mémoire des programmes.

1. Exemple :

Recherche dans une liste triée de $n \geq 1$ éléments de sorte S , fini de l'ordre total \leq et on suppose que la liste est implantée de manière contigüe.

1.1 Recherche séquentielle (version itérative)

$\text{rech}(l, z) = \begin{cases} \text{mi } g == l[i] \text{ alors } i \text{ sinon } \emptyset \text{ fini} \\ \text{avec } i = \underline{\text{init}} \ 1 \ \underline{\text{tq}} \ i < n \wedge l[i] < z \\ \quad \underline{\text{rep}} \ i + 1 \ \underline{\text{fin}} \\ n = \text{lgr}(l) \end{cases}$

* Le coût de $\text{lgr}(l)$ est ignoré compte tenu de l'implantation contigüe.
→ on peut la stocker

Temps max effectif.

Prog C: Nat $\text{rech}(\text{Liste } l, S z) \{ \dots \}$ A passage des paramètres initiales
 $\quad \text{Nat } n = \text{lgr}(l), i = 1;$
 $\quad \text{while } ((i < n) \& \& \text{el}(l, i) < z) \quad i++; \quad$ B Boucle
 $\quad \text{return } (\text{z} == \text{el}(l, i) ? i : \emptyset); \quad \text{C}$

- Temps maximum effectif de cette recherche séquentielle

$$\underset{\substack{\uparrow \\ \text{lgr}(l)}}{\text{Max}}_{\text{seq}}(n) = A + B \times (n-1) + C$$

. Comportement asymptotique: $n \rightarrow +\infty, \quad \mathcal{O} B \times n$

Ce qui retient notre attention, c'est le comportement asymptotique de la fonction quand $n \rightarrow \infty$.



Ordre de grandeur: il existe $k > 0$ tel que

$$\max_{n \geq 0} (n) \leq k \cdot n,$$

alors on dit que

$$\max_{n \geq 0} (x) \in \Theta(x)$$

Espace utilisé: liste + quelques cases v. itérative

liste + pile d'appels récursifs v. récursive
 $\Theta(x)$ de n .

1.2 Recherche dichotomique

03-Dic-19

- version C itérative

```

Nat rech (Liste l, S z) {
    Nat a=1, b=logr(l);
    Nat m;
    while (a <= b) {
        m = (a+b)/2;
        if z <= el(l,m) b = m;
        else a = m+1;
    }
    return el(l,a) == z ? a : -1;
}
  
```

$$\max_{\text{dich}}(n) = A' + B' p + C'$$

→ où p est le nombre max d'exécution de la boucle B'.

$$2^{p-1} < n \leq 2^p$$

avec p , le plus petit entier

$$\log_2 n \leq p$$

$$p = \lceil \log_2 n \rceil$$

n	x	p	2^p
1			
2			
3		2	
4			
5			
6			
7			
8			
9			

- Abstraction

. Comportement asymptotique

$n \rightarrow \infty$

$$\max_{\text{dich}}(n) = B' \log_2 n$$

- Abstraction

- Comportement asymptotique

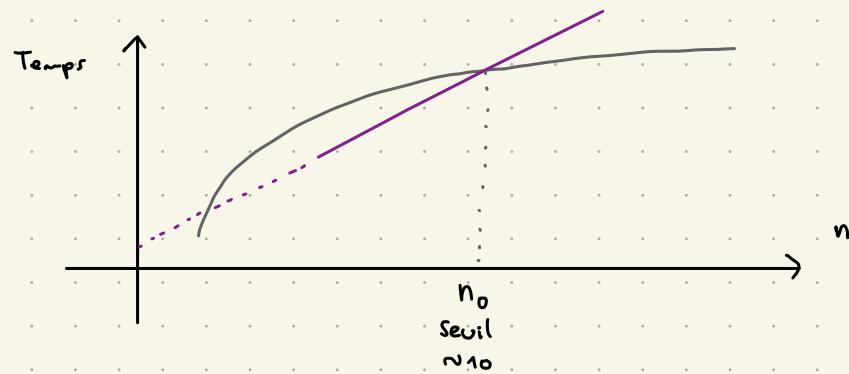
$$n \rightarrow +\infty \quad \underset{\text{dich}}{\text{Max}}(n) = B' \log_2 n$$

- Ordre de grandeur

$$\underset{\text{dich}}{\text{Max}}(n) \leq K \log n$$

Temps $\underset{\text{dich}}{\text{Max}}(n) \in \Theta(\log n)$

1.3 Comparaison entre $\underset{\text{seq}}{\text{Max}}(n)$ et $\underset{\text{dich}}{\text{Max}}(n)$



$$n < n_0 \quad \underset{\text{seq}}{\text{Max}}(n) < \underset{\text{dich}}{\text{Max}}(n)$$

$$n > n_0 \quad \underset{\text{seq}}{\text{Max}}(n) > \underset{\text{dich}}{\text{Max}}(n)$$

1.4 Analyse d'algorithmes

En fait, on obtiendrait le même coût algorithmique en comptant uniquement certaines instructions fondamentales.

Exemple : → Pour les algo de recherche, il suffit de compter les comparaisons (ex. $z \leq el(l, i)$)

→ L'évaluation du coût peut se faire au niveau de l'algo.

3a) Ordres de grandeur

Définitions

- borne supérieure "asymptotique":

$f \in \Theta(g)$ lorsque f est dominée asymptotiquement par g

$$\exists K_1, n_0 \in \mathbb{N}^+ \text{ tels que } \forall n > n_0, f(n) \leq K_1 g(n)$$

- borne inférieure "asymptotique"

$$f \in \Omega(g) \quad K_2 g(n) \leq f(n)$$

- encadrement "asymptotique"

$$\exists K_1, K_2, n_0 \in \mathbb{N}^+, K_2 g(n) \leq f(n) \leq K_1 g(n) \quad \text{où } f \text{ et } g \text{ ont le même ordre de grandeur}$$

Exemple

$$f(n) = \frac{2}{3}n^2 + \frac{1}{2}n \in \Theta(n^2)$$

$$\text{car } c_1 n^2 \leq f(n) \leq c_2 n^2$$

$$\text{c.à.d. } c_1 \leq \frac{2}{3} + \frac{1}{2n} \leq c_2$$

$$n_0 = 3 \quad c_1 = \frac{2}{3} \quad c_2 = \frac{5}{6}$$

Exemple de seuil:

$$\begin{cases} M_A(n) = 10^6 n \\ M_B(n) = 100 n^2 \end{cases}$$

2 algos A et B tels que

On a bien $M_A(n) \in \Theta(M_B(n))$

$$\Delta \quad n=100 \quad M_A(100) = 10^8 \quad M_A(100) = 100 \times M_B(100)$$

$$M_B(100) = 10^6$$

$\log_2(n)$	n	$n \cdot \log_2(n)$	n^2	n^3	2^n
3	10	33	100	1000	1000
7	100	660	10000	1000000	13 10 ³⁰
10	1000	10000	10 ⁶	10 ⁹	10 ³⁰
13	10 000	13000	10 ⁸	10 ¹²	10 ³⁹
17	100 000	1,7.10 ⁶	10 ¹⁰	10 ¹⁵	10 ⁵¹
20	1 000 000	20.10 ⁶	10 ¹²	10 ¹⁸	10 ⁶⁴

20 ms

17 min

32 années

40 M d'années

3.b) Complexité en temps

Exemple

rech séquentielle / dichotomique

pour évaluer un algo :

- unité d'évaluation de temps = durée d'une comparaison (1)
 $z \leq l[i]$ ou $z == l[i]$ ou $z < l[i]$
- unité d'évaluation de la taille de données = longueur de la liste = n
On note D_n , l'ensemble de toutes les données possibles de taille n .

Pour un algo A, le coût en temps ou complexité $C_A(d)$ est considérée dans 3 cas :

- Le meilleur des cas : $\text{Min}_A(n) = \min_{d \in D_n} C_A(d)$

- Le pire des cas : $\text{Max}_A(n) = \max_{d \in D_n} C_A(d)$
(le cas le plus probable)

- En moyenne : $\text{Moy}_A(n) = \sum_A \text{pr}(d) \cdot C_A(d)$

où $\text{pr}(d)$ est la probabilité, dans D_n , de traiter la donnée d avec :

$$0 \leq \text{pr}(d) \text{ et } \sum_{d \in D_n} \text{pr}(d) = 1.$$

- pour une donnée $d \in D_n$,

$$\boxed{\text{Min}_A(n) \leq \text{Moy}_A(n) \leq \text{Max}_A(n)}.$$

ex (suite) :

Évaluation de la complexité dans une liste l non vide et non triée

rech (l, z) = si $z == l[i]$ alors i sinon \emptyset fin
avec $n = \text{lgr}(l)$
 $i = \text{init } 1 \text{ tq } (i < n) \wedge (z \neq l[i])$ rep $i+1$ finp

Renvoie la position la + petite ou ZÉRO

$$D_n = \{(l, z) \mid \text{lgr}(l) = n\}, n \geq 1$$

↑ ↓
liste élément

Hypothèses pour les données

* p = probabilité constante que $z \in l$

* $q = \frac{1-p}{1-p} = 1-p$

* $\text{pr}(l, l[i]) = p_i = p/n$

probabilité uniforme que z soit $l[i]$, $i=1, \dots, n$

→ Autant de chance de trouver z à n'importe quelle position i .

$$\underset{\text{rech}}{\text{Min}}(n) = 1+1$$

$$\underset{\text{rech}}{\text{Max}}(n) = n-1+1$$

$$\left. \begin{array}{l} \\ \end{array} \right\} \underset{\text{rech}}{\text{Moy}}(n) = \text{pr}(l, z \notin l) \times C_{\text{rech}}(l, z \notin l) \\ + \text{pr}(l, l[1]) \times C_{\text{rech}}(l, l[1]) \\ + \text{pr}(l, l[2]) \times C_{\text{rech}}(l, l[2]) \\ \vdots \\ + \text{pr}(l, l[n]) \times C_{\text{rech}}(l, l[n]) \\ = \sum_{i=1}^n p_i \times i + q \times n = \frac{p}{n} \times \frac{n(n+1)}{2} + (1-p) \times n$$

Cas particuliers :

- recherche positive $z \in l$

$$p=1 \quad \underset{\text{rech}}{\text{Moy}}(n) = \frac{n+1}{2}$$

- recherche négative $z \notin l$

$$p=\emptyset \quad \underset{\text{rech}}{\text{Moy}}(n) = n$$

• Si on a une chance sur 2 que z soit dans l .

$$p=q=\frac{1}{2} \quad \underset{\text{rech}}{\text{Moy}}(n) = (3n+1)/4.$$

4. Optimalité

- Étant un problème P, on cherche le mieux meilleur algorithme pour résoudre dans la classe C de tous les résolvant P.
 - avec les données organisées d'une certaine manière
 - en utilisant un certain type d'opérations

Déf: La complexité optimale de la classe C d'algorithmes résolvant P, pour des données de taille n,

C'est la borne inférieure des complexités de tous les algos de la classe.

Un algo est optimal si sa complexité atteint cette forme

Exemple :

Optimalité de P = détermination du minimum de n nombres rangés dans une liste avec l'opération de comparaison $x \leq y$.

$$\left\lceil \frac{n}{2} \right\rceil \leq M_{C,\text{opt}}(n) \leq n - 1$$

algo comparant
 chaque él^t à un autre algo de rech seq
 de minimum

Exemple :

P : recherche dans une liste triée l de $n \geq 1$ él^t (ordre total \leq)

- du rang le plus petit d'un él^t z, si il y est
- de l'intervalle où il se trouve, s'il n'y est pas.,

$$\text{rech}(l, x) = \text{rdich}(l, x, 1, \text{lgr}(l))$$

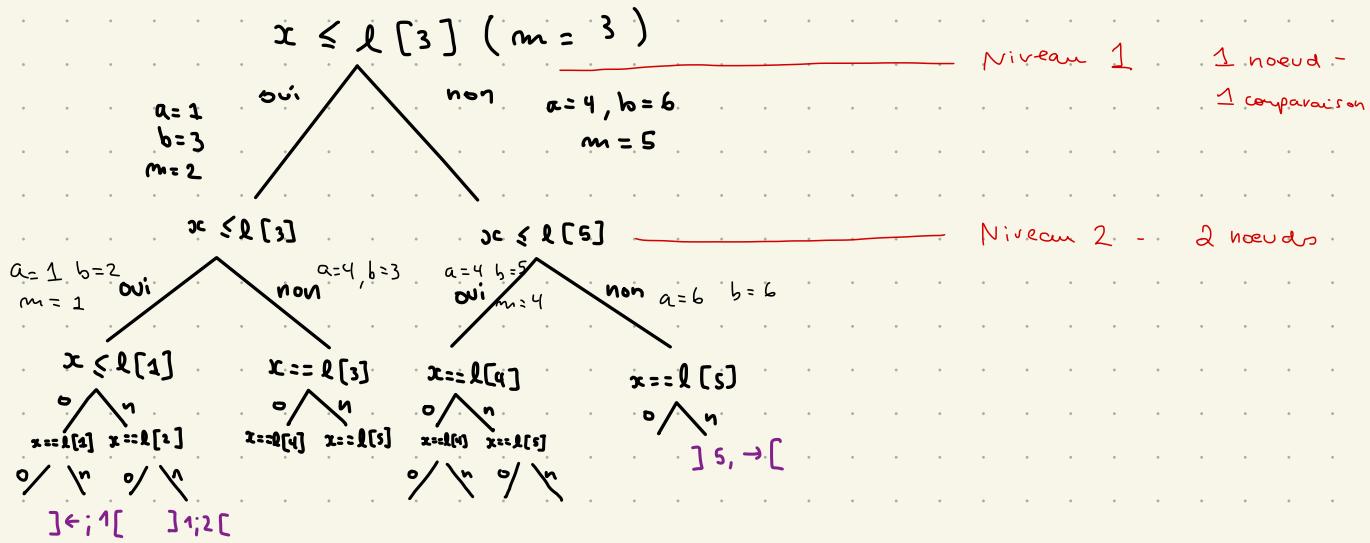
avec $\text{rdich}(l, x, a, b) =$

- si $a = b$ alors si $l[a] = x$ alors a sinon 0 fin
- sinon si $l[a] < x$ alors $\text{rdich}(l, x, a+1, b)$
- sinon $\text{rdich}(l, x, a, m)$ fin
- avec $m = \lfloor (a+b)/2 \rfloor$

fin

ex : $n = 6$

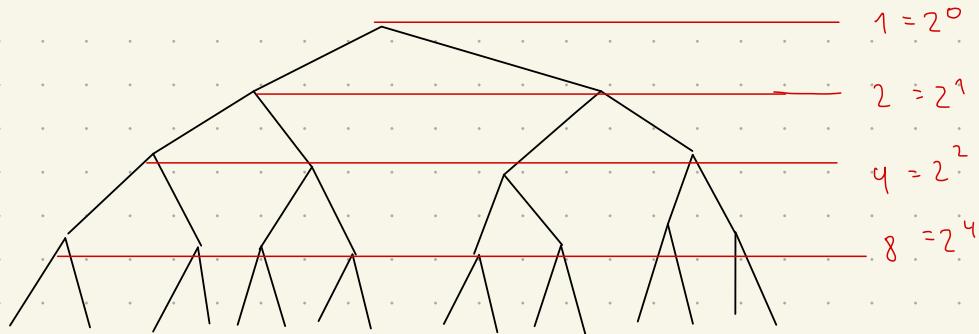
mettre l'algo sous-forme
 d'un arbre
 $a=1, b=6, m=3$.



Chaque branche correspond à une configuration de données, donc à une exécution de l'algorithme.

↳ longueur des branches donne le nombre de comparaison

→ Pour obtenir un algo optimal, il faut remplir l'arbre binaire de la racine vers les feuilles en saturant les niveaux.



Comment placer $2^n - 1$ comparaisons sur un arbre de hauteur minimale ?

$$1 + 2 + \dots + 2^{p-2} < 2^n - 1 \leq 1 + 2 + \dots + 2^{p-1}$$

$$2^{p-1} - 1 < 2^n - 1 \leq 2^p - 1$$

$$2^{p-2} < n \leq 2^{p-1}$$

$$p-2 < \log_2 n \leq p-1$$

$$p-1 < \log_2 n+1 \leq p$$

d'où $p = \lceil \log_2 n \rceil + 1$ comparaisons.

La recherche dichotomique est optimale dans le pire des cas.

