Algorithmique & programmation

Chapitre 2 : Vecteurs

Recherche séquentielle non triée

Accès à un élément de vecteur

- Accès par **position** (déjà vu)
  - V[i]

- Accès **associatif**
  - *soit* **elem** *une variable de même type que les éléments contenus dans* **V**
  - On veut déterminer
    si il existe un indice i ∈ [1..n] tel que **V[i] = elem**

Recherche séquentielle

- **séquentielle**

➔ le vecteur est parcouru

➔ avec un indice strictement croissant (resp. décroissant) de une unité

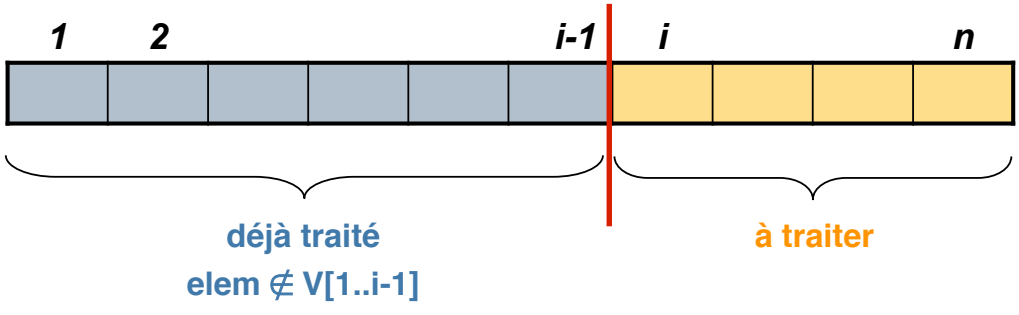
➔ à partir de l'indice le plus petit (resp. le plus grand)

Recherche séquentielle (non trié)

- Trouver l'hypothèse :
  - Si on est en train de rechercher, c'est que l'on n'a pas trouvé

donc

  - l'élément **elem** que l'on cherche n'est pas dans la partie du vecteur déjà parcourue



Recherche séquentielle (non trié)

- Le raisonnement par récurrence

Hypothèse elem ∉ V[1..i-1]

- i = n+1 ⇨ *elem ∉ V[1..n]* ⇨ résultat = faux ✳
- i ≤ n ⇨
  - V[i] = elem ⇨ *elem ∈ V[1..n]* ⇨ résultat = vrai ✳
  - V[i] ≠ elem ⇨ i := i + 1 ; ➡ ***H***

Trouver l'itération ?

Chemin depuis le retour à l'hypothèse vers les conditions

Trouver l'itération

- i ≤ n ⇨
  - V[i] = elem ⇨ *elem ∈ V[1..n]* ⇨ résultat = vrai ✳
  - V[i] ≠ elem ⇨ i := i + 1 ; ➡ ***H***

- Il faut que les deux conditions soient vérifiées
  - i ≤ n et V[i] ≠ elem

Attention !!!!!

- ***La première fois que i ≤ n devient faux i = n + 1***
- **Peut-on consulter V[n + 1] ?**
  - **NON !!!!!!!**

Trouver l'itération

- Il faut que les deux conditions soient vérifiées
  - i ≤ n **et** V[i] ≠ elem
- On doit éviter
  - d'évaluer V[i] ≠ elem lorsque i ≤ n est faux
- Solution ?
  - Remplacer le **et** par un **et alors**
  - Pour obtenir

$$i \leq n \textbf{ et alors } V[i] \neq elem$$

- *Commentaire*
  - *Ce n'est pas de l'optimisation, c'est **obligatoire** !!!*

Recherche séquentielle (non trié)

- Le raisonnement par récurrence

Hypothèse elem ∉ V[1..i-1]

➤ i = n+1 ⇨ *elem ∉ V[1..n]* ⇨ résultat = faux ✳

➤ i ≤ n ⇨

➤➤ V[i] = elem ⇨ *elem ∈ V[1..n]*
⇨ résultat = vrai ✳

➤➤ V[i] ≠ elem ⇨ i := i + 1 ; ➡ ***H***

Itération tantque (i ≤ n) **et alors** (V[i] ≠ elem) faire …

Trouver l'initialisation ?

Connaît-on un vecteur pour lequel on sait que elem ∉ V

Trouver l'initialisation

- Connaît-on un vecteur pour lequel on sait que elem ∉ V ?

  Oui !!

- Le vecteur vide (il ne contient aucun élément)
  - V[1..0]
- *Rappel*

  Hypothèse elem ∉ V[1..i-1]

- Quelle valeur de **i** pour avoir **V[1..0]** ?
  - Réponse : **i doit valoir 1**

Initialisation i := 1 ; ➨ ***H***

Recherche séquentielle (non trié)

- Le raisonnement par récurrence

Hypothèse elem ∉ V[1..i-1]

➤ i = n+1 ⇨ *elem ∉ V[1..n]* ⇨ résultat = faux ✳

➤ i ≤ n ⇨

➤➤ V[i] = elem ⇨ *elem ∈ V[1..n]*
⇨ résultat = vrai ✳

➤➤ V[i] ≠ elem ⇨ i := i + 1 ; ➨ ***H***

Itération tantque (i ≤ n) **et alors** (V[i] ≠ elem) faire …

Initialisation i := 1 ; ➨ ***H***

fonction **accès1**

fonction accès1 (d V[1..n] : vecteur ; d elem : t) : booléen ;
spécification *{n ≥ 0}* ➔ *{résultat = elem ∈ V[1..n]}*
    i : entier ;
debfonc
    i := 1 ;
    {elem ∉ V[1..i-1]}
    tantque (i ≤ n) **et alors** (V[i] ≠ elem) **faire**
        {elem ∉ V[1..i]}
        i := i + 1 ;
        {elem ∉ V[1..i-1]}
    finfaire ;
    {(i > n) ou sinon (V[i] = elem)}
    on doit rendre un résultat vrai ou faux
finfonc ;

fonction **accès1** (calcul du résultat)

- Condition de sortie du tantque
  - *{(i > n) ou sinon (V[i] = elem)}*
- La fonction doit rendre vrai ou faux (raisonnement)
  - Soit on a l'intuition du calcul
  - Soit on fait un tableau de sortie

i > n	**v[i] = elem**	**résultat**
vrai	non examiné	faux
faux (i≤n)	vrai	vrai
faux (i≤n)	faux	impossible (tantque)

fonction **accès1** (calcul du résultat)

- Lecture du tableau de sortie

i > n	v[i] = elem	résultat
vrai	non examiné	faux
faux (i≤n)	vrai	vrai
faux (i≤n)	faux	impossible (tantque)

- Le résultat est vrai lorsque i ≤ n, faux sinon
  - **retour** i ≤ n ;

fonction **accès1**

fonction accès1 (d V[1..n] : vecteur ; d elem : t) : booléen ;
spécification *{n ≥ 0}* ➔ *{résultat = elem ∈ V[1..n]}*
    i : entier ;
debfonc
    i := 1 ;
    {elem ∉ V[1..i-1]}
    tantque (i ≤ n) **et alors** (V[i] ≠ elem) **faire**
        {elem ∉ V[1..i]}
        i := i + 1 ;
        {elem ∉ V[1..i-1]}
    finfaire ;
    {(i > n) ou sinon (V[i] = elem)}
    retour i ≤ n ;
finfonc ;

si n=0, on ne rentre pas dans l'itération

⚠ fonction **accès1** commentaires ⚠

- Pour écrire la fonction **accès1**, on a considéré :
  - **V** un vecteur qui contient des éléments de type **t**
  - **elem** un élément de type **t**
- la fonction **accès1** proposée est générique, i.e. c'est le modèle d'une fonction d'accès
  - ce modèle peut s'appliquer à la recherche
    - d'un entier dans un vecteur d'entiers
    - d'un caractère dans un vecteur de caractères
    - d'une chaîne dans un vecteur de chaînes
    - …
- En Ada aussi, la généricité existe …

Généricité en Ada

- En Ada aussi, la généricité existe …
  - … on l'a vu pour le paquetage des entrées/sorties sur les types énumérés

```
type T_Jour is (Lundi,…,Dimanche);
package P_JourIO is new P_Enum(T_Jour);
```

- On la verra plus tard pour les fonctions et procédures !
- Pour l'instant, on ne va écrire que des fonctions et procédures avec des paramètres effectifs typés
  - entier, vecteur d'entiers
  - caractère, vecteur de caractères
  - chaîne, vecteur de chaînes

fonction **accès1** (vecteur d'entiers)

- soit V un vecteur d'entier de type **vectInt**

```
function accès1Int (V : in vectInt ; elem : in integer)
                                              return boolean is
--{V vide ou non} ➔ {résultat = elem ∈ V}
   i : integer ;
begin
  i := V'First ;
  --{elem ∉ V}
  while (i ≤ V'Last ) and then (V(i) ≠ elem) loop
    i := i + 1 ;
  end loop;
  --{(i > V'Last) ou sinon (V(i) = elem)}
  return i ≤ V'Last ;
end accès1Int ;
```

1 de V[1..n] {borne inférieure}

n de V[1..n] {borne supérieure}

V(i) remplace V[i]

n de V[1..n] {borne supérieure}

fonction **accès1** (vecteur de caractères)

- soit V un vecteur de caractères de type **vectChar**

```
function accès1Char (V : in vectChar ;
                     elem : in character) return boolean is
--{V vide ou non} ➔ {résultat = elem ∈ V}
  i : integer ;
begin
  i := V'First ;
  --{elem ∉ V}
  while (i ≤ V'Last ) and then (V(i) ≠ elem) loop
    i := i + 1 ;
  end loop;
  --{(i > V'Last) ou sinon (V(i) = elem)}
  return i ≤ V'Last ;
end accès1Char ;
```

fonction **accès1** (vecteur de chaînes)

- soit V un vecteur de chaînes de 10 caractères de type **vectCh10**

```
function accès1Ch10 (V : in vectCh10 ;
                     elem : in ch10) return boolean is
--{V vide ou non} ➔ {résultat = elem ∈ V}
  i : integer ;
begin
  i := V'First ;
  --{elem ∉ V}
  while (i ≤ V'Last ) and then (V(i) ≠ elem) loop
    i := i + 1 ;
  end loop;
  --{(i > V'Last) ou sinon (V(i) = elem)}
  return i ≤ V'Last ;
end accès1Ch10 ;
```


Recherche séquentielle (non trié, non vide)

- Considérons de V n'est pas vide (**n≥1**)
- Ici, on peut s'arrêter
  - Lorsque **i** vaut **n**
    - au pire, n vaut 1 et on peut consulter V[1]

  ou
  - si on a trouvé **elem**

Recherche séquentielle (non trié, non vide)

- Raisonnement par récurrence

Hypothèse elem ∉ V[1..i-1], **i ≤ n**

➤ i = n ⇨ *elem ∈ V[1..n] ssi V[i]=elem*
⇨ résultat = (V[i]=elem) ✳

➤ i < n ⇨
➤➤ V[i] = elem ⇨ *elem ∈ V[1..n]*
⇨ résultat = vrai ✳
➤➤ V[i] ≠ elem ⇨ i := i + 1 ; ➡ ***H***

Trouver l'itération ?

Chemin depuis le retour à l'hypothèse vers les conditions

Trouver l'itération

➤ i < n ⇨
➤➤ V[i] = elem ⇨ *elem ∈ V[1..n]*
⇨ résultat = vrai ✳
➤➤ V[i] ≠ elem ⇨ i := i + 1 ; ➡ ***H***

- Il faut que les deux conditions soient vérifiées
  - i < n et V[i] ≠ elem
- Est-ce que cela pose un problème ?
  - **Non !**
    - Au pire i = n et on peut consulter V[n]

Itération tantque (i < n) **et** (V[i] ≠ elem) faire …

Recherche séquentielle (non trié, non vide)

- Raisonnement par récurrence

Hypothèse elem ∉ V[1..i-1], **i ≤ n**

- i = n ⇨ *elem ∈ V[1..n] ssi V[i]=elem*
  ⇨ résultat = (V[i]=elem) ✳
- i < n ⇨
  - V[i] = elem ⇨ *elem ∈ V[1..n]*
    ⇨ résultat = vrai ✳
  - V[i] ≠ elem ⇨ i := i + 1 ; ➡ ***H***

Itération tantque (i < n) **et** (V[i] ≠ elem) faire …

Trouver l'initialisation ?

Sachant que l'on arrête lorsque i vaut n

Trouver l'initialisation

- Au pire i = n
- On s'arrête lorsque i vaut n
  - **i doit valoir 1**

Initialisation i := 1 ; ➡ ***H***

Recherche séquentielle (non trié, non vide)

- Raisonnement par récurrence

Hypothèse elem ∉ V[1..i-1], **i ≤ n**

➤ i = n ⇨ *elem ∈ V[1..n] ssi V[i]=elem*
⇨ résultat = (V[i]=elem) ✳

➤ i < n ⇨

➤➤ V[i] = elem ⇨ *elem ∈ V[1..n]*
⇨ résultat = vrai ✳

➤➤ V[i] ≠ elem ⇨ i := i + 1 ; ➡ ***H***

Itération tantque (i < n) **et** (V[i] ≠ elem) faire …

Initialisation i := 1 ; ➡ ***H***

fonction **accès2**

fonction accès2 (d V[1..n] : vecteur ; d elem : t) : booléen ;
spécification *{n > 0}* ➔*{résultat = elem ∈ V[1..n]}*
i : entier ;
debfonc
i := 1 ;
{elem ∉ V[1..i-1]}
tantque (i < n) **et** (V[i] ≠ elem) **faire**
{elem ∉ V[1..i]}
i := i + 1 ;
{elem ∉ V[1..i-1]}
finfaire ;
{(i = n) ou (V[i] = elem)}
on doit rendre un résultat vrai ou faux
finfonc ;

fonction **accès2** (calcul du résultat)

- Condition de sortie du tantque
  - $\{(i = n)\ ou\ (V[i] = elem)\}$
- La fonction doit rendre vrai ou faux (raisonnement)
  - Intuition du calcul ou **tableau de sortie**

i = n	**v[i] = elem**	**résultat**
vrai	vrai	vrai
vrai	faux	faux
faux (i<n)	vrai	vrai
faux (i<n)	faux	impossible (tantque)

fonction **accès2** (calcul du résultat)

- Lecture du tableau de sortie

i = n	**v[i] = elem**	**résultat**
vrai	vrai	vrai
vrai	faux	faux
faux (i<n)	vrai	vrai
faux (i<n)	faux	impossible (tantque)

- Le résultat est vrai lorsque V[i]=elem, faux sinon
  - **retour** V[i]=elem ;

fonction **accès2**

fonction accès2 (d V[1..n] : vecteur ; d elem : t) : booléen ;
spécification *{n > 0} ➔ {résultat = elem ∈ V[1..n]}*
    i : entier ;
debfonc
    i := 1 ;
    {elem ∉ V[1..i-1]}
    tantque (i < n) **et** (V[i] ≠ elem) **faire**
        {elem ∉ V[1..i]}
        i := i + 1 ;
        {elem ∉ V[1..i-1]}
    finfaire ;
    {(i = n) ou (V[i] = elem)}
    retour V[i] = elem ;
finfonc ;

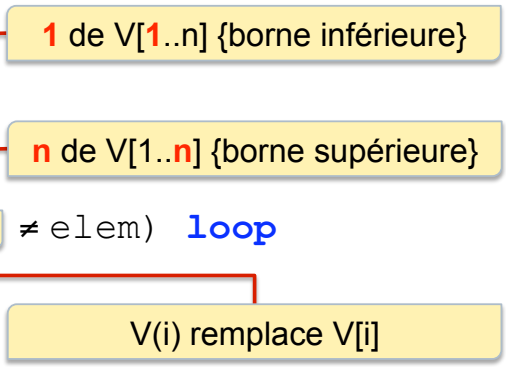
⚠ fonction **accès2** commentaires ⚠

- Pour écrire la fonction **accès2**, on a considéré :
  - **V** un vecteur qui contient des éléments de type **t**
  - **elem** un élément de type **t**
- la fonction **accès1** proposée est générique, i.e. c'est le modèle d'une fonction d'accès
  - ce modèle peut s'appliquer à la recherche
    - d'un entier dans un vecteur d'entiers
    - d'un caractère dans un vecteur de caractères
    - d'une chaîne dans un vecteur de chaînes
    - …

fonction **accès2** (vecteur d'entiers)

- soit V un vecteur d'entier de type **vectInt**

```
function accès2Int (V : in vectInt ; elem : in integer)
                                              return boolean is
--{V non vide} ➔ {résultat = elem ∈ V}
   i : integer ;
begin
   i := V'First ;
   --{elem ∉ V}
   while (i < V'Last) and (V(i) ≠ elem) loop
      i := i + 1 ;
   end loop;
   --{(i = V'Last) ou (V(i) = elem)}
   return V(i) = elem ;
end accès2Int ;
```



fonction **accès2** (vecteur de caractères)

- soit V un vecteur de caractères de type **vectChar**

```
function accès2Char (V : in vectChar ;
                     elem : in character) return boolean is
--{V non vide} ➔ {résultat = elem ∈ V}
   i : integer ;
begin
   i := V'First ;
   --{elem ∉ V}
   while (i < V'Last) and (V(i) ≠ elem) loop
      i := i + 1 ;
   end loop;
   --{(i = V'Last) ou (V(i) = elem)}
   return V(i) = elem ;
end accès2Char ;
```

fonction **accès2** (vecteur de chaînes)

- soit V un vecteur de chaînes de 10 caractères de type **vectCh10**

```
function accès2Ch10 (V : in vectCh10 ;
                     elem : in ch10) return boolean is
--{V non vide} ➔ {résultat = elem ∈ V}
  i : integer ;
begin
  i := V'First ;
  --{elem ∉ V}
  while (i < V'Last) and (V(i) ≠ elem) loop
    i := i + 1 ;
  end loop;
  --{(i = V'Last) ou (V(i) = elem)}
  return V(i) = elem ;
end accès2Ch10 ;
```