Algorithmique & programmation

Chapitre 4 : Listes chaînées

Accès

Recherche

Accès par position

- On souhaite écrire la fonction suivante

fonction pointk (d liste : pointeur ; d k : entier) : pointeur ;

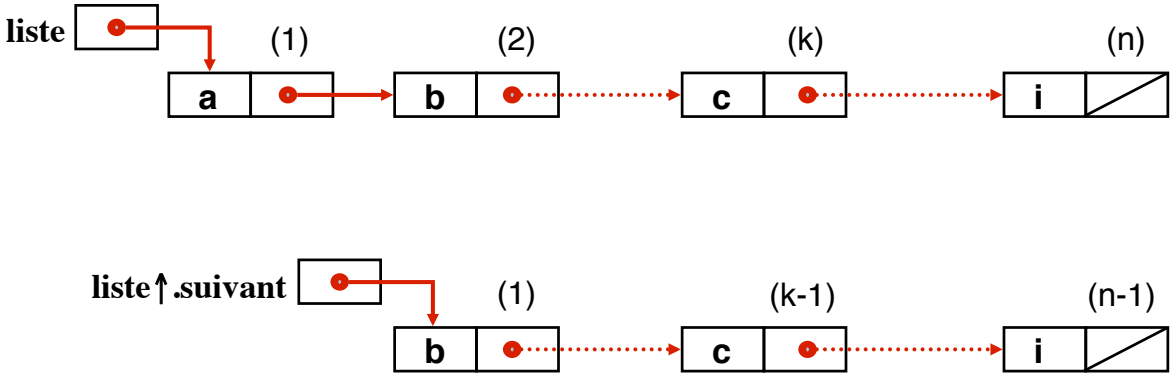
spécification *{ }* ➔ *{(résultat = adresse du k^ième élément, s'il existe)*

v (résultat = nil, s'il n'existe pas)}

- Vers le schéma récursif (liste non vide)
  - Si k = 1 alors
    - la cellule recherchée est la première
    - i.e. : celle qui est pointée par **liste**
  - Si k > 1 alors
    - on va chercher le k-1^iéme dans la **liste** privée de sa première cellule
    - on va chercher le k-1^iéme dans **liste↑.suivant**

Accès par position

Illustration



Accès par position

Schéma récursif

- liste⁺ = <> ⇨ résultat = liste (=nil) ; ✳
- liste⁺ ≠ <>
  - k = 1
    ⇨ résultat = liste ; ✳
  - k ≠ 1
    ⇨ résultat = pointk (liste↑.suivant, k - 1) ;

Accès par position

fonction pointk (d liste : pointeur ; d k : entier) : pointeur ;
spécification *{ }* ➔ *{(résultat = adresse du k^ième élément, s'il existe)*
v (résultat = nil, s'il n'existe pas)}

debfonc
  si liste = nil **alors**
    retour liste ;
  sinonsi k = 1 **alors**
    retour liste ;
  sinon
    retour pointk (liste↑.suivant, k-1) **;**
  finsi ;
finfonc ;

Accès par position (**ada**)

```
function pointk (liste : in Ta_ListEnt ;
                 k : in integer) return Ta_ListEnt is
begin
   if liste = null then
     return liste ;
   else if k = 1 then
     return liste ;
   else
     return pointk (liste.all.suivant, k-1) ;
   end if ;
end pointk ;
```

Accès par position

- mise en facteur de **retour** liste ;

fonction pointk (d liste : pointeur ; d k : entier) : pointeur ;
spécification { } ➔ *{(résultat = adresse du k^ième élément, s'il existe)*
v (résultat = nil, s'il n'existe pas)}

```
debfonc
    si (liste = nil) ou (k = 1) alors
        retour liste ;
    sinon
        retour pointk (liste↑.suivant, k-1) ;
    finsi ;
finfonc ;
```

- k ≤ 0 : algorithme correct mais parcours de toute la liste

➔ précondition k > 0 souhaitable

Accès par position (**ada**)

- mise en facteur de **return** liste ;

```
function pointk (liste : in Ta_ListEnt ;
                 k : in integer) return Ta_ListEnt is
begin
   if liste = null or k =1 then
     return liste ;
   else
     return pointk (liste.all.suivant, k-1) ;
   end if ;
end pointk ;
```

- k ≤ 0 : algorithme correct mais parcours de toute la liste

➔ précondition k > 0 souhaitable

Accès associatif

- On souhaite écrire la fonction suivante

fonction point (d liste : pointeur ; d val : t) : pointeur ;

spécification { } ➔ {*(résultat = adresse de la première occurrence de val, val* ∈ *liste*⁺*)* ∨ *(résultat = nil, val* ∉ *liste*⁺*)*}

- Vers le schéma récursif (liste non vide)
  - Si **liste↑.info = val** alors
    - la cellule recherchée est la première
    - i.e. : celle qui est pointée par **liste**
  - Si **liste↑.info ≠ val** alors
    - on va chercher **val** dans **liste↑.suivant**

Accès associatif

- Schéma récursif

➤ liste⁺ = <> ⇨ résultat = liste (=nil) ; ✳

➤ liste⁺ ≠ <>

➤➤ liste↑.info = val

⇨ résultat = liste ; ✳

➤➤ liste↑.info ≠ val

⇨ résultat = point (liste↑.suivant, val) ;

Accès associatif

fonction point (d liste : pointeur ; d val : t) : pointeur ;
spécification { } ➔ {(*résultat = adresse de la première occurrence de val,* $val \in liste^+$) ∨ (*résultat = nil,* $val \notin liste^+$)}
debfonc
    si liste = nil **alors** {*résultat = nil,* $val \notin liste^+$}
        retour liste ;
    sinonsi liste↑.info = val **alors**
            {*résultat = adresse 1ère occurrence de val,* $val \in liste^+$}
        retour liste ;
    sinon
        retour point (liste↑.suivant, val) ;
    finsi ;
finfonc ;

Accès associatif (**ada**)

```
function point (liste : in Ta_ListEnt ;
                val : in integer) return Ta_ListEnt is
begin
   if liste = null then
     return liste ;
   else if liste.all.info = val then
     return liste ;
   else
     return point (liste.all.suivant, val) ;
   end if ;
end point ;
```

Accès associatif

- mise en facteur de **retour** liste ; ⇒ **ou sinon**

fonction point (d liste : pointeur ; d val : t) : pointeur ;
spécification { } ➔ {(*résultat = adresse de la première occurrence de val,*
val ∈ liste⁺) ∨ (résultat = nil, val ∉ liste⁺)}
debfonc
  si (liste = nil) **ou sinon** (liste↑.info = val) **alors**
    retour liste ;
  sinon
    retour point (liste↑.suivant,val) ;
  finsi;
finfonc ;

Accès associatif (**ada**)

- mise en facteur de **`return`** `liste ;` ⇒ **`or else`**

```
function point (liste : in Ta_ListEnt ;
                val : in integer) return Ta_ListEnt is
begin
  if liste = null or else liste.all.info = val then
    return liste ;
  else
    return point (liste.all.suivant, val) ;
  end if ;
end point ;
```

Définition d'une liste triée

- une liste vide est triée
- une liste composée d'un seul élément est triée
- une liste de plus d'un élément est triée
  - si tous les éléments consécutifs vérifient la relation d'ordre :
    - si liste ≠ nil et liste↑.suivant ≠ nil
    - alors liste↑.info ≤ liste↑.suivant↑.info

Accès associatif dans une liste triée

- On souhaite écrire la fonction suivante

fonction point (d liste : pointeur ; d val : t) : pointeur ;

spécification *{liste+ est trié}* ➔ *{(résultat = adresse de la première occurrence de val, val ∈ liste⁺) ∨ (résultat = nil, val ∉ liste⁺)}*

Accès associatif dans une liste triée

- Schéma récursif

➤ liste⁺ = <> ⇨ résultat = liste (=nil) ; ✳

➤ liste⁺ ≠ <>

➤➤ liste↑.info < val

⇨ résultat = point (liste↑.suivant, val) ;

➤➤ liste↑.info > val

⇨ résultat = nil ; ✳

➤➤ liste↑.info = val

⇨ résultat = liste ; ✳

Accès associatif dans une liste triée

fonction point (d liste : pointeur ; d val : t) : pointeur ;

spécification *{liste+ est trié}* ➔ *{(résultat = adresse de la première occurrence de val, val ∈ liste⁺) ∨ (résultat = nil, val ∉ liste⁺)}*

debfonc

si liste = nil **alors**

retour nil ;

sinonsi liste↑.info < val **alors**

retour point (liste↑.suivant, val) ;

sinonsi liste↑.info > val **alors**

retour nil ;

sinon *{liste↑.info = val}*

retour liste ;

finsi ;

finfonc ;

Accès associatif dans une liste triée (ada)

```
function point (liste : in Ta_listEnt;
                val : in integer) return Ta_ListEnt is
begin
   if liste = null then
     return null ;
   else if liste.all.info < val then
     return point (liste.all.suivant, val) ;
   else if liste.all.info > val then
     return null ;
   else
     return liste ;
   end if ;
end point ;
```


Accès associatif dans une liste triée

- mise en facteur de **retour** nil ; ⇒ **ou sinon**

fonction point (d liste : pointeur ; d val : t) : pointeur ;
spécification *{liste+ est trié}* ➔ *{(résultat = adresse de la première occurrence de val, val ∈ liste⁺) ∨ (résultat = nil, val ∉ liste⁺)}*
debfonc
  si (liste = nil) **ou sinon** (liste↑.info > val) **alors**
    retour nil ;
  sinonsi liste↑.info < val **alors**
    retour point (liste↑.suivant, val) ;
  sinon *{liste↑.info = val}*
    retour liste ;
  finsi ;
finfonc ;

Accès associatif dans une liste triée (**ada**)

- mise en facteur de **return** null ; ⇒ **or else**

```
function point (liste : in Ta_listEnt;
                val : in integer) return Ta_ListEnt is
begin
  if liste = null or else liste.all.info > val then
    return null ;
  else if liste.all.info < val then
    return point (liste.all.suivant, val) ;
  else
    return liste ;
  end if ;
end point ;
```