Récursivité

Définitions :

Une action paramétrée P( ) est dite récursive si son exécution provoque ou entraîne un ou plusieurs appels à P( ). Un algorithme récursif contient une ou plusieurs actions récursives.

Principes de construction d'algorithme récursifs :

Le principe de construction d'actions paramétrées récursives est donné par les points suivants :

Le nombre d'appels récursifs (niveaux) doit être fini. Il faut donc que les paramètres contiennent une ou plusieurs variables de commande qui sont testées à chaque niveau pour savoir si on doit continuer ou non les appels récursifs.

Déterminer le ou les cas particuliers qui sont exécutés directement sans appels récursifs.

Décomposer le problème initial en sous problèmes de même nature, telle que des décompositions successives aboutissent toujours à l'un des cas particuliers (appelés aussi cas triviaux).

Exemple 1:

Le calcul de la valeur factorielle d'un nombre donné n>=0 peut se faire de deux manières différentes :

1ère méthode :

n! = 1*2*3*…*n-1*n   sachant que 0!=1

On obtient alors l'algorithme itératif (classique) donné par la fonction suivante :

Fonction fact (E/n: entier): entier
i, p : entier ;
Début
~~**Si** (n=0 ou n=1) **alors** retourner 1 ; **fsi** ;~~
p   1 ;
Pour i   2 **à** n **faire** p   p*i; **fait** ;
retourner p;
Fin ;

2ème méthode :

0!=1 ; 1!=1 ; 2!=1*2 ; 3!=1*2*3 ; 4!=1*2*3*4 ; 5!=1*2*3*4*5 …
=2 =6 =24 =120 …

Nous remarquons que:
0!=1 ; 1!= 0!*1 ; 2!=1!*2 ; 3!= 2!*3 ; 4!=3!*4 ; 5!=4!*5

De façon générale pour un nombre n>0 on a : n! = (n-1)!*n
On constate donc la définition de n! est récursive, puisqu'elle se réfère à elle même quand elle applique (n-1)!

Le calcul de la valeur factorielle d'un nombre est défini par :
a) Si n=0 ou n=1 alors n!=1
b) Si n>0 alors n!= (n-1)!*n

Fonction fact (E/n: entier): entier
Début

```
        Si (n=0 ou n=1) alors retourner 1 ;
                      sinon retourner (n*fact (n-1));
        fsi ;
    Fin;
```

Déroulement de fact (4)

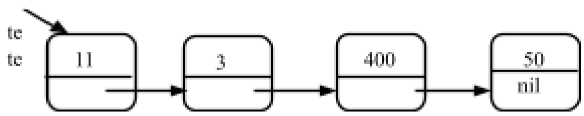
```
Début          niveau1        niveau2        niveau3      niveau4
                fact(4)        fact(3)        fact(2)      fact(1)

q<-fact(4)     4*fact(3)      3*fact(2)    2*fact(1)      =1

  =24          =4*6=24        =3*2=6       =2*1=2
Fin.             Fin ;          Fin ;        Fin ;         Fin ;
```

Remarques :

a) Quand n=0 ou n=1, la valeur de n! est donnée directement. 0 et 1 sont appelées : ***valeur de base***.
b) Pour n>1, la valeur de n! est définie en fonction d'une valeur plus petite que n et plus voisine de la valeur de base.
c) La variable n, est testée à chaque fois pour savoir s'il faut arrêter les appels récursifs ou non. n est appelé ***variable de commande***

Exemple 2:

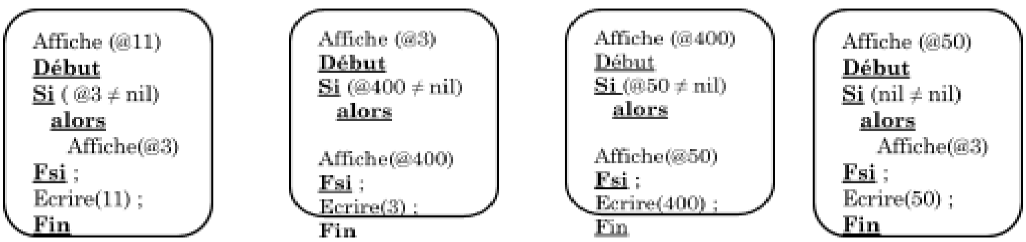
Etant donné la liste simplement chaînée ci-dessous. On veut utiliser le concept de la récursivité pour afficher les nombres de cette liste en commençant par le dernier élément jusqu'au premier c.-à-d. : 50 400 3 11.



Procédure Affiche (E/ p: liste)
Début
    Si (p^.svt ≠ nil) **alors** Affiche (p^.svt) ; **fsi** ;
    Ecrire (p^.inf) ;
Fin;

Déroulement

Début
Affiche (@ 11);
Fin



Dans ce cas, la programmation récursive remplace les instructions de boucle (while ou for) par des appels de fonctions.

***A faire**: mettez l'instruction Ecrire (p^.info) dans le sinon. Exécutez une autre fois cette action.*

Schémas généraux d'actions récursives
Dans une action P( ) récursive on a deux parties :
- Celle qui correspond au cas de base (ou cas trivial)
- Celle qui contient au moins un appel

On peut donc représenter P( ) sous forme :
1er schéma :

```
Action P (x1, … xn)
<Déclaration des variables locales> ;
Début
  Si (Test d'arrêt) alors  Q // instructions du point d'arrêt
  Sinon
       I1 ;  // instructions ;
       P (h (x1, … xn))      /* appel récursif */
       I2 ; // instructions ;
  Fsi ;
Fin ;
```

h(x₁, … xₙ) doit faire évoluer le paramètre x_i vers le point d'arrêt, afin que l'action se termine.
I1 et *I2* des blocs d'instructions (éventuellement vide) qui ne comportent aucun appel à *P*

2ème schéma :

```
Action P(…)
Début
     Tant que < condition> faire  G(P(…), …)
        Q ;
Fin ;
```

où le bloc *G(P(…),…)* contient un appel récursif et le bloc Q permet la résolution directe du problème (ne contient pas d'appel récursif).

Récursivité directe et récursivité indirecte :

Une action qui fait appel à elle-même explicitement dans sa définition est dite récursive directe ou récursivité simple.
Si une action A fait référence (ou appel) à une action B qui elle-même fait appel directement ou indirectement à A, on parle alors de récursivité indirecte ou récursivité croisée.

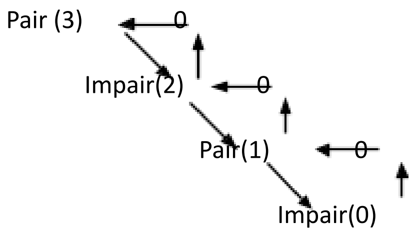
```
Action A( )
Début
  Si < condition> alors  B( )
                  sinon QA ;
  fsi ;
Fin ;
```

```
Action B( )
Début
  Si < condition> alors  A( )
                  sinon QB ;
  fsi ;
Fin ;
```

Exemple :

Fonction Pair (E/n: entier): booléen	Fonction Impair (E/n: entier) : booléen
Début **si** (n=0) **alors** retourner vrai **sinon** retourner Impair (n-1); **fsi** ;	**Début** **si** (n=0) **alors** retourner faux **sinon** retourner Pair (n-1); **fsi** ;

Fin;	***<u>Fin</u>*** *;*



Fonctionnement de la récursivité

Un programme ne peut s'exécuter que s'il est chargé en mémoire centrale, chaque instruction du programme se trouve à une adresse donnée de la mémoire. Lorsqu'un programme fait appel à une fonction, le système sauvegarde l'adresse de retour (adresse de l'instruction qui suit l'appel), ainsi que les valeurs des variables locales.
Quand une fonction ***f*** appelle une fonction ***g,*** on doit sauvegarder l'adresse de retour de ***f*** (paramètres et variables locales) avant l'appel de ***g***, ce contexte doit être récupéré après le retour de ***g***.
S'il y a plusieurs appels imbriqués, le système gère ***<u>une pile</u>*** pour sauvegarder (empiler) les différents contextes des différents appels récursifs.

Les paramètres de l'appel récursif changent. A chaque appel les variables locales sont stockées dans une pile. Ensuite les paramètres ainsi que les variables locales sont désempilées au fur et à mesure qu'on remonte les niveaux.

Lors de l'ième appel, le compilateur empile :

- Les valeurs des paramètres au niveau i
- Les valeurs des variables locales du niveau i
- L'adresse de retour au niveau i

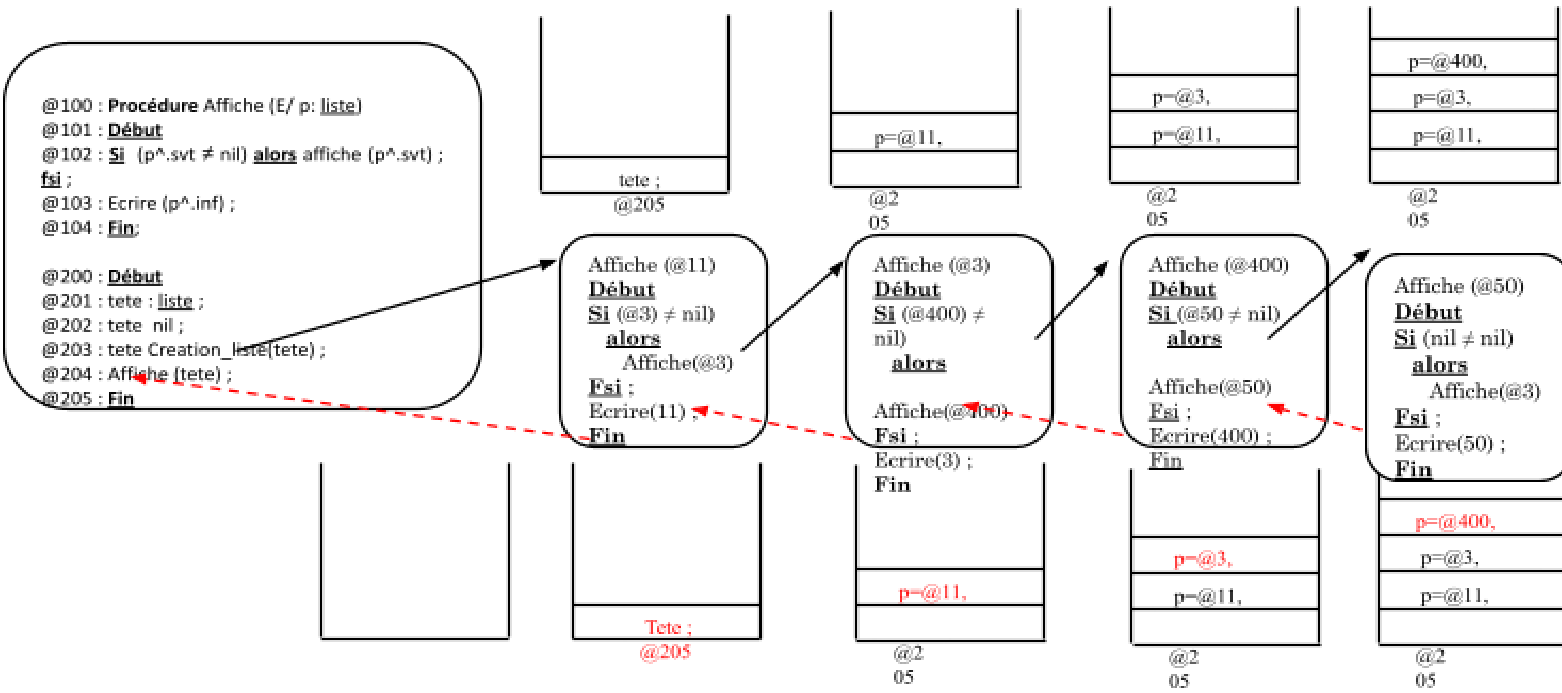
A la fin des appels récursifs retour du niveau i+1 au niveau i :

- Retour au programme principal si la pile est vide
- Dépiler l'adresse de retour
- Dépiler le contexte du niveau i : les valeurs des variables du niveau i
- Exécuter l'instruction suivant le dernier appel

<u>Exemple</u> : soit l'algorithme suivant :

@100 : **Procédure** Affiche (E/ p: <u>liste</u>)
@101 : **<u>Début</u>**
@102 : **<u>Si</u>** (p^.svt ≠ nil) **<u>alors</u>** affiche (p^.svt) ; **<u>fsi</u>** ;
@103 : Ecrire (p^.inf) ;
@104 : **<u>Fin</u>**;

@200 : **<u>Début</u>**
@201 : tete : <u>liste</u> ;
@202 : tete nil ;
@203 : tete Creation_liste(tete) ; // une action qui crée une liste
@204 : Affiche (tete) ;
@205 : **<u>Fin</u>**



Différents types de récursivité :

a- **Récursivité terminale :**

Si l'exécution d'un appel récursif n'est jamais suivie par l'exécution d'une autre instruction, cet appel est dit récursif à droite ou encore appelée ***récursivité terminale***. Dans ce type de récursivité, les appels récursifs n'ont pas besoin d'êtres empilés car l'appel suivant remplace simplement l'appel précédent dans le contexte d'exécution.

Exemple :

Procédure AfficheRecT(E/n, i : entier) ; //1er appel i=1

Début

si (i<=n) alors écrire(i);

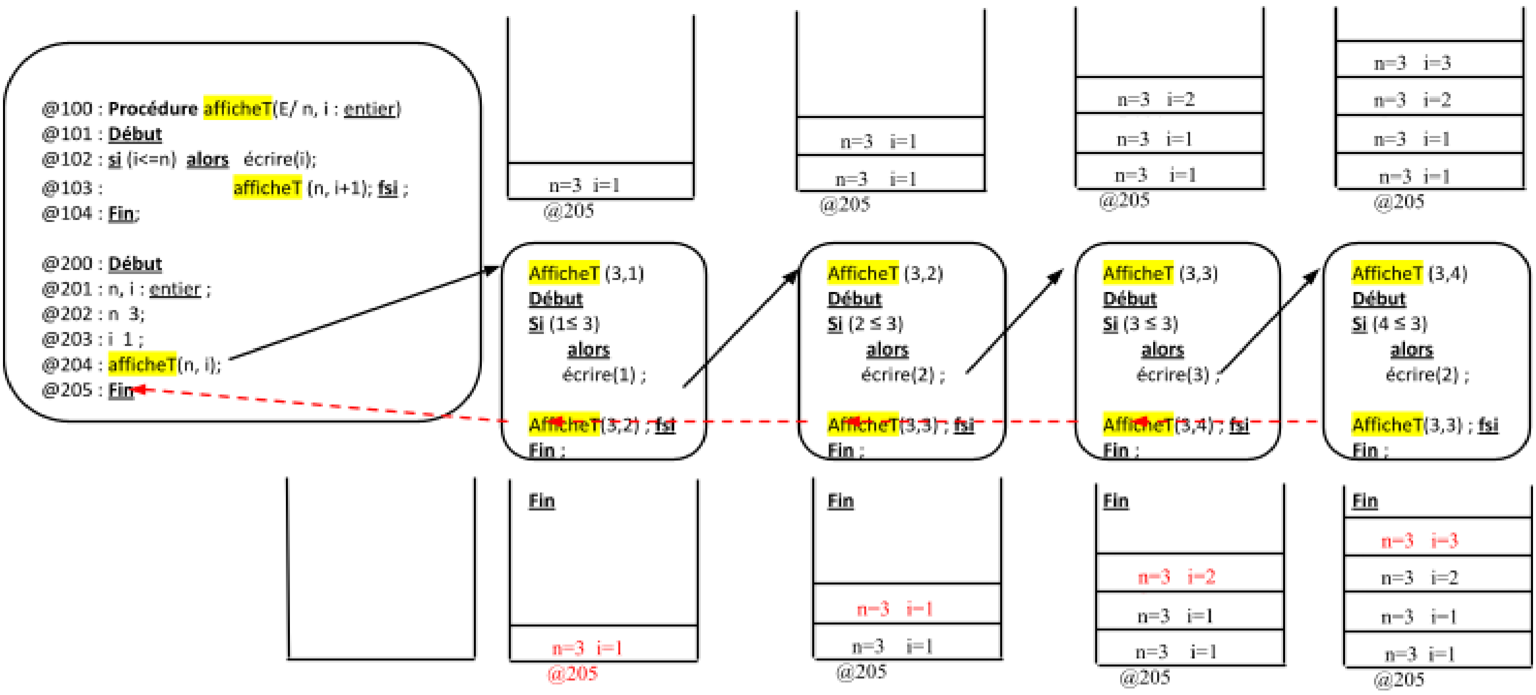
AfficheRecT (n, i+1);

fsi;

Fin ;

Remarque :

- Après exécution de AfficheRecT (3, 1), le résultat sera **1 2 3**
- L'action AfficheRecT (n, i) est une fonction récursive terminale car aucun traitement n'est effectué après l'appel de cette action



b- Récursivité non terminale :

Une action récursive est dite non terminale si l'exécution d'un appel récursif est suivie par l'exécution d'une autre instruction ou l'appel récursif est utilisé pour réaliser un traitement. Une action récursive ***non terminale nécessite une pile***.

Exemple :

Procédure afficheRecNT (E/ n, i : entier) ; // 1er appel i=1
Début
    si (i<=n) **alors** afficheRecNT(n, i+1);
                  écrire(i);
    fsi;
Fin ;

Remarque :

- Après exécution de AfficheRecNT (3, 1), le résultat sera **3 2 1**
- L'action AfficheRecNT (n, i) est une fonction récursive non terminale car il existe un traitement à effectuer après l'appel de cette action.

Elimination de la récursivité :

La récursivité simplifie la structure d'un programme mais la plupart du temps, le gain en simplicité vaut une baisse relative des performances d'exécution. La récursivité est souvent couteuse en temps et en espace mémoire car elle nécessite l'emploi du concept de la pile.

De plus, certains langages de programmation n'admettent pas la récursivité (exemple : Fortran). A cet effet il est intéressant de noter que l'on peut montrer que, si le langage de programmation utilisé le permet, il est toujours possible de transformer une action itérative en une action récursive ; cependant, la réciproque n'est pas vraie.

D'une manière générale, on évitera d'utiliser la récursivité lorsqu'on peut la remplacer par une définition itérative, à moins de bénéficier d'un gain considérable en simplicité.

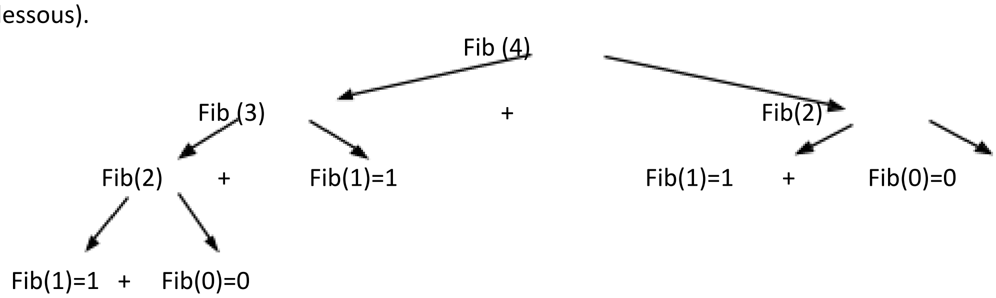
<u>Exemple :</u> Les nombres de Fibonnaci : $F_0=0$, $F_1=1$

$F_n=F_{n-1} + F_{n-2}$ n>=2.

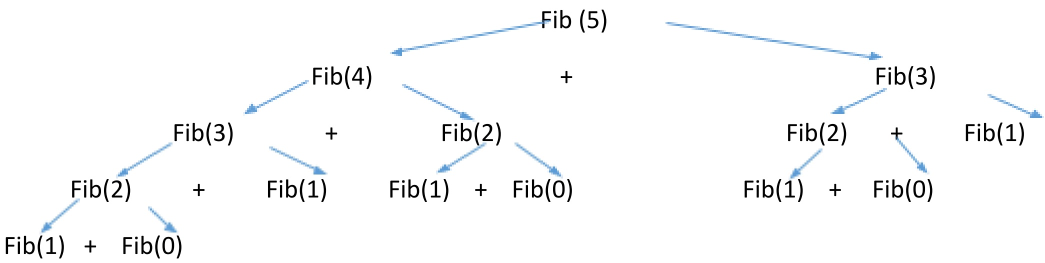
La fonction récursive permettant d'obtenir ces nombres est :

Fonction Fib(E/ n : <u>entier</u>) : <u>entier</u>
<u>Début</u>
 <u>Si</u> (n=0) **<u>alors</u>** retourner 0
  <u>Sinon si</u> (n=1) **<u>alors</u>** retourner 1
   <u>Sinon</u> retourner Fib(n-1)+Fib(n-2);
  <u>fsi</u>;
 <u>fsi</u>;
<u>Fin</u> ;

Pour n=4, la fonction Fib(4) est égale à 3. Fib(4) a besoin de 9 appels pour arriver au résultat (voir l'arbre ci-dessous).



Pour n=5, Fib(5)=5 (15 appels pour arriver au résultat). Les valeurs successive de cette suite : 0, 1, 1, 2, 3, 5, 8, 12, 21, 34, 55,…



Voici maintenant la fonction itérative équivalente à la fonction récursive Fib.
Fonction Fib (E/ n : <u>entier</u>) : <u>entier</u>
x, y, z, i : <u>entier</u> ;
<u>Début</u>
 <u>Si</u> (n=0) **<u>alors</u>** retourner 0 ; **<u>fsi</u>** ;
 <u>Si</u> (n=1) **<u>alors</u>** retourner 1 ; **<u>fsi</u>** ;
 x 0 ; y 1 ; z 1 ; /* x=Fib(0) et y= Fib(1) */
 pour i=2 **<u>à</u>** n **<u>faire</u>**
  z x+y ;
  x y ;
  y z ;
 <u>fait</u> ;

```
  retourner z ;
Fin ;
```

Pour n=4 : x=0
y=1
i=2 : z=1 x=1 y=1
i=3 : z=2 x=1 y=2
i=4 : z=3 x=2 y=3
Résultat z=3 avec 3 itérations.

Remarque :
Le problème se situe au nombre d'appels à la fonction, nous constatons que pour la solution récursive le nombre d'appels est un nombre **exponentiel** (c'est une mauvaise solution très coûteuse) alors que la solution itérative ne coute que **n appels**.
Cette version itérative peut à son tour se convertir en une nouvelle version récursive :

```
Fonction Fib(E/ x, y, n : entier) : entier
Début
 Si (n=0) alors retourner 0
        Sinon Si (n=1) alors retourner y
                    Sinon retourner Fib(y, x+y, n-1);
              Fsi ;
 Fsi ;
Fin ;
```

n=4 : x=0, y=1
Fib(0,1,4) ⟶ Fib(1,1,3) ⟶ Fib(1,2,2) ⟶ Fib(2,3,1) = 3 donc Fib(4)=3. On a que 4 appels, le temps d'exécution est devenu ***linéaire***.

Comme on peut le constater, l'élimination de la récursivité est parfois très simple, elle revient à écrire une boucle, à condition d'avoir bien fait attention à l'exécution. Mais parfois elle est extrêmement difficile à mettre en œuvre.

a- Elimination de la récursivité terminale

Un algorithme est dit récursif terminal (ou récursif à droite) s'il ne contient aucun traitement après un appel récursif.
- Dans ce cas le contexte de la fonction n'est pas empilé.
- L'appel récursif sera remplacé par une boucle tant que.

Cas1 :

f(x) /* récursive*/	f(x) /* itérative*/
Début **si** (condition(x)) **alors** A ; f(g(x)); **fsi** ; **Fin** ;	**Début** **Tantque** (condition(x)) **faire** A ; x=g(x) ; **fait** ; **Fin** ;

Cas2 :

f(x) / récursive*/*	*f(x) /* iterative*/*
Début ***si** (condition(x)) **alors** A ; f(g(x));* ***sinon** B ;*	***Début*** ***Tantque** (condition(x)) **faire*** *A ; x=g(x) ;*

fsi; ***Fin***;	***Fait***; ***B*** ; ***Fin*** ;

b- Elimination de la récursivité non terminale

1) cas d'un seul appel récursif:

Ici pour pouvoir éliminer la récursivité, il va falloir sauvegarder le contexte de l'appel récursif.

Cas1 :

f(x) /* récursive*/
Début
si (condition(x))
alors A ; f(g(x));
fsi ;/*nécessite une pile */
B ;
Fin;
	f(x) /* iterative*/
	Début
	pile p=initpile();
	Tantque (condition(x))
	faire A ;
	empiler(p, x);
	x=g(x) ;
	fait ;
	B ;
	Tantque (!pilevide(p))
	faire desempiler(p, x) ;
	B ;
	fait ;
	Fin ;

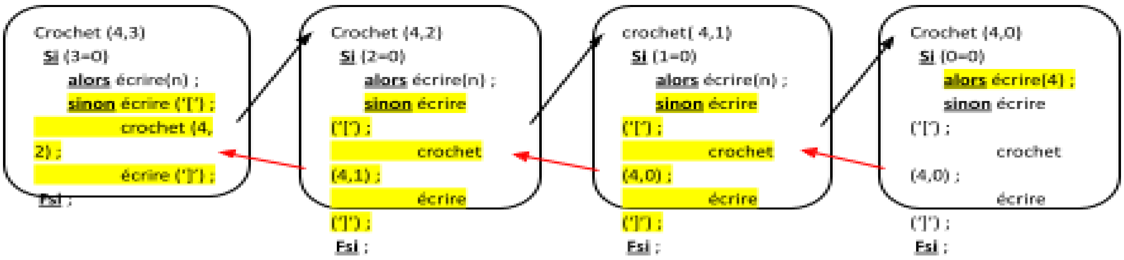
Cas2 :

f(x) /* récursive*/
Début
si (condition(x)) **Alors**
A1 ;
f(g(x));
A2 ;
Sinon B ;
Fsi;
Fin;
	f(x) /* iterative*/
	Début
	pile p=initpile();
	Tantque (condition(x))
	faire
	A1 ;
	empiler(p, x);
	x=g(x) ;
	Fait ;
	B;
	Tantque (!pilevide(p))
	faire
	désempiler (p, x) ;
	A2 ;
	fait ;
	Fin ;

Exercice 1 : écrire une action récursive qui affiche un entier n entouré de nc (entier) de paires de crochets '[' et ']'. Exemple pour n=4 et nc=3 l'action affiche : [[[4]]]

Procédure crochet(E/ n, nc : entier)
Début
Si (nc=0) **alors** écrire(n) ;
sinon écrire ('[') ;

crochet (n, nc-1) ;
écrire (']') ;
fsi ;
Fin.



[[[4]]]

Exercice 2 : écrire une fonction récursive qui recherche par dichotomie un élément dans un tableau d'entier trié. La fonction renvoie l'indice de l'élément s'il existe ou -1 sinon.

1/ solution de votre camarade

```
int decho(int T[100],int n,int x)
{
if (T[n/2]==x) {
printf("la position de %d est = %d",x,n/2);
                 }
else {
       return decho(T,n-1,x);
      }
}
```

Fonction dicho(E/ t : tab[100] : entier, E/ binf, bsup, v : entier) : entier
Début
p : entier ;
Si (binf > bsup) **alors** retourner -1 ;
Sinon p (binf + bsup)/2 ; // chercher l'indice du milieu
Si(t[p]=v) **alors** retourner p ;
Sinon **si**(t[p] > v) **alors** // chercher dans la moitié gauche du tableau
retourner dicho(t, binf, p-1, v) ;
Sinon // chercher dans la droite du tableau
retourner dicho(t, p+1, bsup, v) ;
Fsi ;
Fsi;
Fsi ;
Fin.

Premier appel de la fonction : x dicho(t, 1, 10, 23)

-20	-10	-5	0	12	13	27	44	66	100

binf =1 p=5 bsup=10

dicho(t, 6, 10, 23)

binf =6 p=8 bsup=10

dicho (t, 6, 7, 23)

-20	-10	-5	0	12	13	27	44	66	100

-20	-10	-5	0	12	13	27	44	66	100

p=6 bsup= 7

dicho (t, 6, 5, 23)

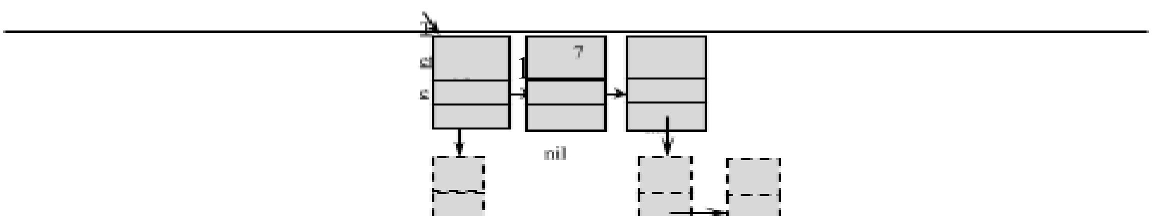
bsup =5 binf =6

dicho (t, 6, 5, 23) =-1 //car binf > bsup

-20	-10	-5	0	12	13	27	44	66	100

Exercice 2

1. Donner le type de cette structure.
2. Ecrire une action paramétrée **récursive** Affiche_R1( ) qui affiche les éléments d'une liste d'entier. On commence par afficher le dernier élément, avant dernier,…, premier.
3. Ecrire une seconde action paramétrée **récursive** Affiche_R2( ) qui affiche les éléments de la structure de la figure ci-dessous. Penser à utiliser la fonction récursive Affiche_R1( ) dans le corps de la seconde action. Les nombres seront affichés dans cet ordre : **191** : 911 119 **7** : **15** : 51
4. Ecrire une action paramétrée **récursive** Efface_R2( ) qui libère l'espace mémoire occupé par la structure de la figure ci-dessous. Penser à utiliser le même principe donné par les questions 2 et 3. Les éléments seront libérés dans cet ordre : 911 119 **191** **7** 51 **15**

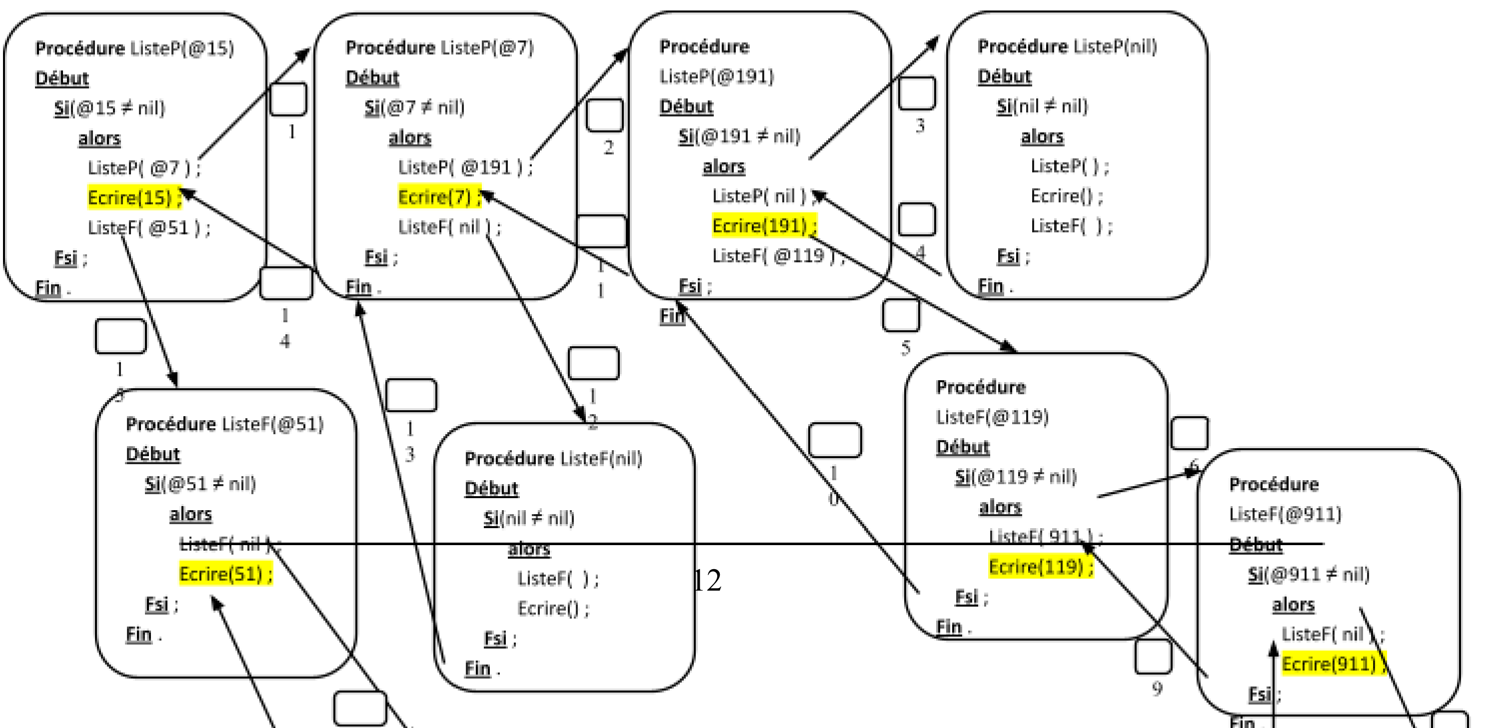


Solution

1/

Type <u>listeF</u> : ^ objetF ; Type objetF : Enregistrement val : <u>entier</u> ; svt : <u>listeF</u> ; Fing ;	Type <u>listeP</u> : ^ objetP ; Type objetP : Enregistrement val : <u>entier</u> ; svt1 : <u>listeP</u> ; svt2 : <u>listeF</u> ; Fing ;

2/ **Procédure** Affiche_listeF(E/ t : <u>listeF</u>) **<u>Début</u>** **<u>Si</u>**(t ≠ nil) **<u>alors</u>** Affiche_listeF(t^.svt) ; Ecrire(t^.val) ; **<u>Fsi</u>** ; **<u>Fin</u>** .	3/ **Procédure** Affiche_listeP(E/ t : <u>listeP</u>) **<u>Début</u>** **<u>Si</u>**(t ≠ nil) **<u>alors</u>** Affiche_listeP(t^.svt1) ; Ecrire(t^.val) ; Affiche_listeF(t^.svt2) ; **<u>Fsi</u>** ; **<u>Fin</u>** .



4 : 191 **8 : 911** **9 : 119** **11 : 7** **14 : 15** **17 : 51**

4/ **Procédure** detruire_listeF(E/ t : <u>listeF</u>) **<u>Début</u>** **<u>Si</u>**(t ≠ nil) **<u>alors</u>** detruire _listeF(t^.svt) ; Liberer(t) ; t nil ; **<u>Fsi</u>** ; **<u>Fin</u>** .	**Procédure** detruire_listeP(E/ t : <u>listeP</u>) **<u>Début</u>** **<u>Si</u>**(t ≠ nil) **<u>alors</u>** detruire_listeP (t^.svt1) ; detruire_listeF (t^.svt2) ; Liberer(t) ; t nil ; **<u>Fsi</u>** ; **<u>Fin</u>** .

<u>A faire</u> : Dessiner le schéma d'appel de l'action detruire_listeP(E/ t : <u>listeP</u>).