Les piles

Une **pile** est une liste dans laquelle toutes les insertions/suppressions se font en tête. Le dernier élément introduit : ***sommet de pile*** joue un rôle particulier, puisque tout accès à la pile se fait par cet élément. L'accès à un élément quelconque se fait après le retrait de tous les éléments qui le précède. Une file a une structure "LIFO" (Last In, First Out) c'est-à-dire « dernier entré premier sorti ».

Les piles sont très utilisées en informatique :

- ***Par un compilateur, lors des appels de fonctions*** : Avant de se brancher vers la fonction appelée, le compilateur sauvegarde l'adresse de retour et les variables du programme appelant, afin de les restituer au retour de la fonction. De plus, les retours se font dans le sens inverse des appels, ce qui coïncide bien avec la structure LIFO de la pile.
- ***Par un compilateur, pour l'évaluation des expressions arithmétiques :*** ce dernier point est détaillé dans ce qui suit.
- …

Opérations sur les Piles :

Les opérations sur les piles sont les suivantes :

- **Empiler** un élément : l'action consistant à ajouter un nouvel élément au sommet de la pile.
- ***Désempiler*** d'un élément : l'action consistant à retirer l'élément se trouvant au sommet de la pile.
- ***Consultation*** : consulter l'élément se trouvant au sommet de la pile. L'état de la pile ne change pas suite à cette opération.

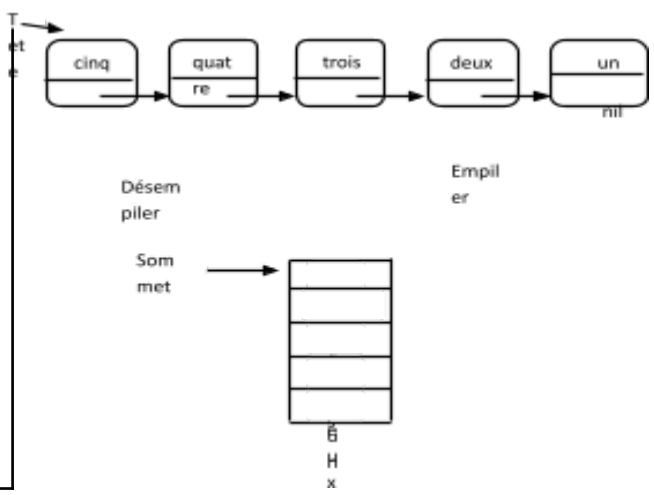
Représentation des piles en mémoire :

On peut représenter ou implémenter les piles en mémoire de façon contiguë ou chaînée.

1-Représentation chaînée

Déclaration

```
Type pile : ^ objet ;
Type objet : Enregistrement
     inf : <type_elt> ;
     svt : pile ;
 Fing ;
```



Allocation mémoire pour un élément de la pile

```
Fonction creer_noeud ( ) : pile
Début
  L : pile ;
  L<- nil ;
  Allouer(L) ;
```

```
Si (L=nil) alors
            écrire (« problème d'allocation » ;
Fsi ;
Retourner (L) ;
Fin ;
```

Test si la pile est vide

```
Fonction pile_vide(E/ p : pile) : booléen
Début
 Si (p=nil) alors  retourner  vrai ;
           sinon retourner faux ;
 Fsi ;
Fin ;
```

Initialiser une pile

```
Fonction Init_pile ( ) : pile
Début
   retourner (nil);
Fin ;
```

Consulter l'élément se trouvant en tête de pile

```
Fonction  sommet_pile(E/ p : pile) : <Type_elt>
Début
  retourner (p^.inf) ;
Fin ;
```

Ajout d'un élément

```
Procédure Empiler(ES/ p  pile,  E/ x :<Type_elt>)
tmp : pile

tmp<- creer_noeud ( ) ;
si (tmp ≠ nil) alors
                    tmp^.inf <- x ;
                    tmp^.svt <- p ;
Fsi;
p<- tmp ;     ATTENTION  ERREUR A CORRIGER

Fin ;
```

Suppression d'un élément

```
Procédure désempiler(ES/ p : pile, ES/ x :<Type_elt>)
Début
  tmp : pile ;  // test si la pile est vide se fait dans le prog

   x <- sommet_pile(p) ;
   tmp<- p ;
   p<- p^.svt ;
   Libérer (tmp) ;

Fin ;
```

2-Représentation contigüe

On peut représenter une pile par un tableau alloué d'une manière statique. On peut choisir le couple suivant : (T, sommet) pour manipuler cette pile.

un	deux	trois	quatre	cinq	……. ……				

1) Déclaration

```
Constante max 100
Type Pile = Enregistrement
    T : tableau [max] <Type_elt> ;
    sommet : entier ;
Fing ;
```

2) Initialisation

```
Fonction InitPile ( ) : Pile
Début
  p : Pile
  p. sommet <-  0;
  retourner (p);
Fin
```

3) Consulter l'élément se trouvant en tête

```
Fonction sommet_pile (E/ p : Pile) : <Type_elt>
Début
   retourner (p .T [p. sommet]) ;
Fin
```

4) Test si la pile est vide

```
Fonction PileVide (E/ p : Pile) : booléen
Début
   si (p. sommet = 0) alors retourner (vrai) ;
                      sinon retourner (faux) ;
   fsi ;
Fin ;
```

5) Test si la pile est pleine

```
Fonction PilePleine (E/ p : Pile) : booléen
Début
   si (p. sommet = max) alors retourner (vrai) ;
                        sinon retourner (faux) ;
   fsi ;
Fin ;
```

6) Ajout d'un élément

```
Procédure Empiler (ES/  p: Pile, E/ x : <Type_elt>)
Début
  p. sommet <-   p. sommet + 1 ;
  p. T [p. sommet] <-   x ;
Fin
```

7) Suppression d'un élément

```
Procédure Désempiler (ES/ p: Pile, E/S x : <Type_elt>)
Début
  x <-    p. T [p. sommet] ;
  p. sommet  <-    p. sommet - 1 ;
Fin
```

Utilisation des piles dans l'évaluation des expressions arithmétiques

Une utilisation courante des piles est l'élaboration par le compilateur d'une forme intermédiaire de l'expression à évaluer (exemple : A+B). Après l'analyse lexicale et syntaxique, l'expression est traduite en une forme intermédiaire plus facilement évaluable (exemple : AB+).

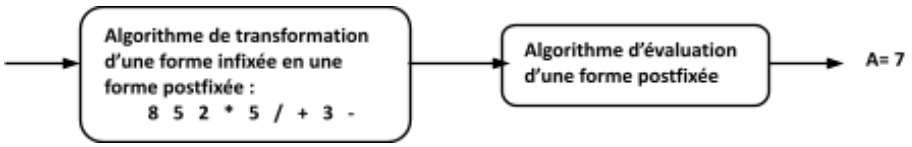
La notation ***< Opérande> < Opérateur> < Opérande>*** est dite INFIXE.

A + B

La notation ***< Opérande Gauche > < Opérande Droit > < Opérateur>*** est dite POSTFIXE

A B +

L'évaluation des expressions comme A+B*C n'est pas directe puisque dès la rencontre d'un opérateur on ne sait pas quelle opération doit-on évaluer d'abord ? Ceci dépend des priorités des opérateurs dans le cas d'une expression non parenthésée ou de la position des parenthèses dans l'expression dans le cas d'une expression parenthésée.



A= 8 + (5 * 2 / 5) – 3

Exemples de Transformation:

Forme Infixée	Forme Postfixée
25 -11	25 11-
25 + 11*5	25 11 5 * +
(25 + 11) *5	25 11 + 5 *

Algorithme de transformation d'une forme infixée en une forme postfixée

L'algorithme utilise deux **piles** P et R. Il reçoit comme donnée l'expression arithmétique déjà analysée, rangée dans un vecteur T de taille : *Taille_Expression*. Il utilise les fonctions suivantes supposées définies.

Operateur(x)	détermine si x est un opérateur : + (addition), - (soustraction), *(multiplication), / (division) et % (reste de la division entière)
Operande(x)	détermine si x est un opérande. Un opérande peut être une suite de symboles représentant un entier ou un réel
Priorite(x)	retourne la priorité de l'opérateur x Priorité(+) = Priorité(-) < Priorité(*) = Priorité(/) = Priorité(%)
Opération (x1, x2, operateur)	effectue l'opération (x1 operateur x2) et retourne le résultat

Algorithme de Transformation

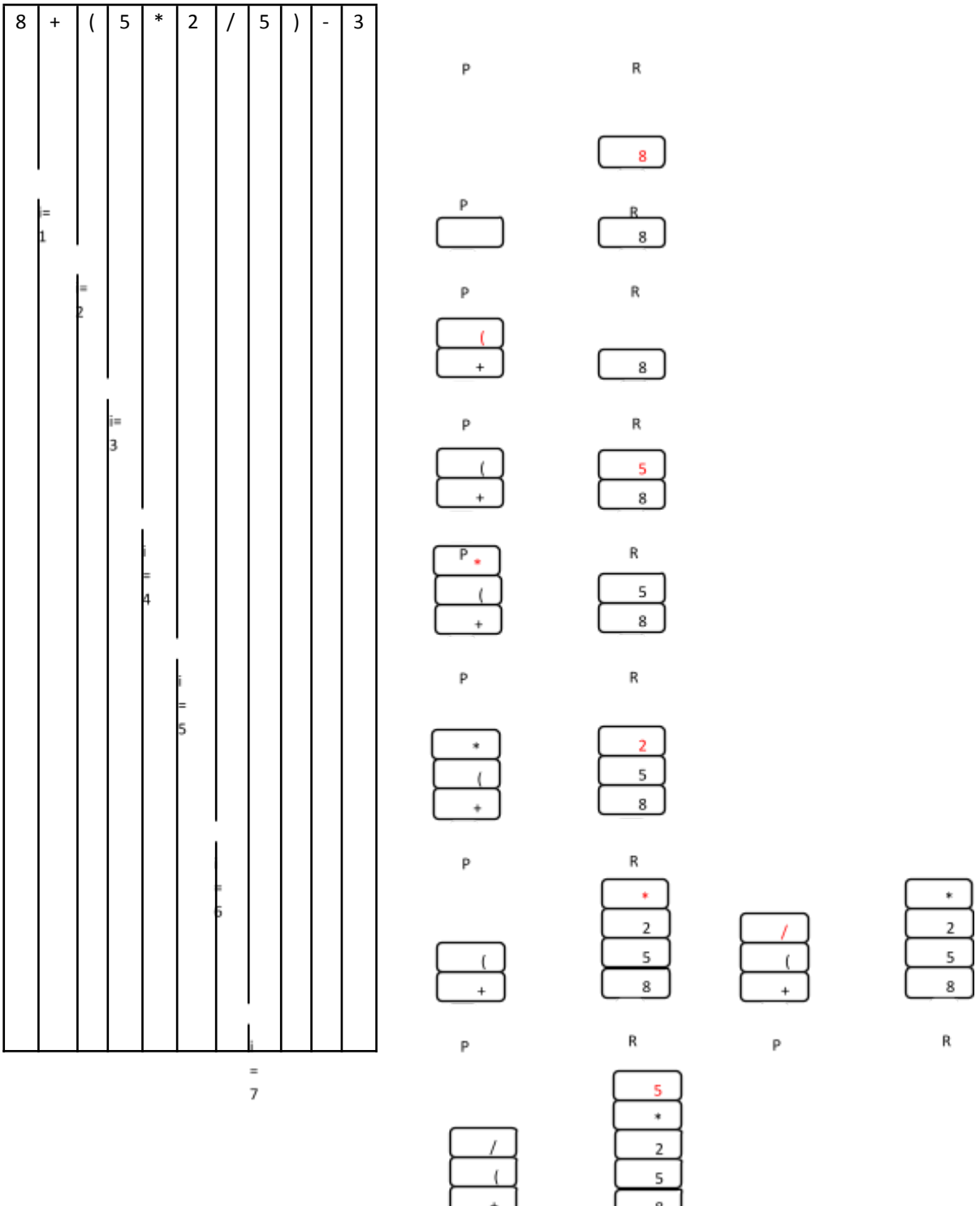
```
Début
 P<- Init_pile ();
 R<- Init_pile ();

 Pour i <- 1 à Taille_Expression
  Faire
    Si (Operande (T[i]) alors Empiler (R, T[i]) ; fsi ;
    Si (T[i]= '(' )  alors Empiler (P, T[i]) ; fsi ;
    Si (Operateur ( T[i] ) alors
          Tant que (non PileVide(P) et Operateur (SommetPile(P)) et (Priorité (T[i]) ≤ Priorité (SommetPile(P)))
             Faire
               Désempiler (P, x) ;
               Empiler (R, x) ;
           Fait
          Empiler (P, T[i]);
    Fsi ;
    Si (T[i] = ')' )  alors
          Tant que (non PileVide(P) et (SommetPile (P) ≠ '('  )
             Faire
               Désempiler (P, x) ;
               Empiler (R, x) ;
             Fait
          Désempiler (P, x) ;
    Fsi
 Fait
  Tant que (non PileVide(R))
```

Faire
Désempiler (R, x) ;
Empiler (P, x) ;
Fait
Fin

A la fin de cet algorithme, la pile P contiendra la forme postfixée de l'expression.

Exemple : soit l'expression suivante : 8 + (5 * 2 / 5) - 3



8	+	(	5	*	2	/	5	)	-	3

8	+	(	5	*	2	/	5	)	-	3

8	+	(	5	*	2	/	5	)	-	3

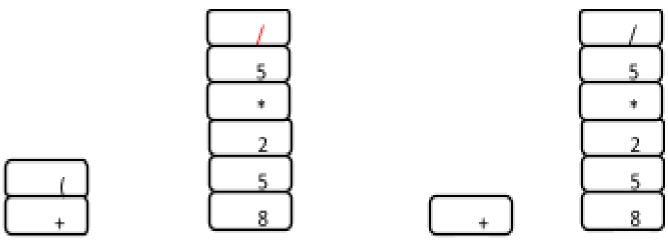
8	+	(	5	*	2	/	5	)	-	3

8	+	(	5	*	2	/	5	)	-	3

8	+	(	5	*	2	/	5	)	-	3

8	+	(	5	*	2	/	5	)	-	3

8	+	(	5	*	2	/	5	)	-	3



8	+	(	5	*	2	/	5	)	-	3

8	+	(	5	*	2	/	5	)	-	3

8	+	(	5	*	2	/	5	)	-	3

Algorithme d'évaluation d'une forme postfixée

L'algorithme utilise deux piles P et R. La pile P contient la forme postfixée de l'expression et la pile R étant initialement vide. Le résultat final se trouvera dans la pile R et la pile P devient vide.

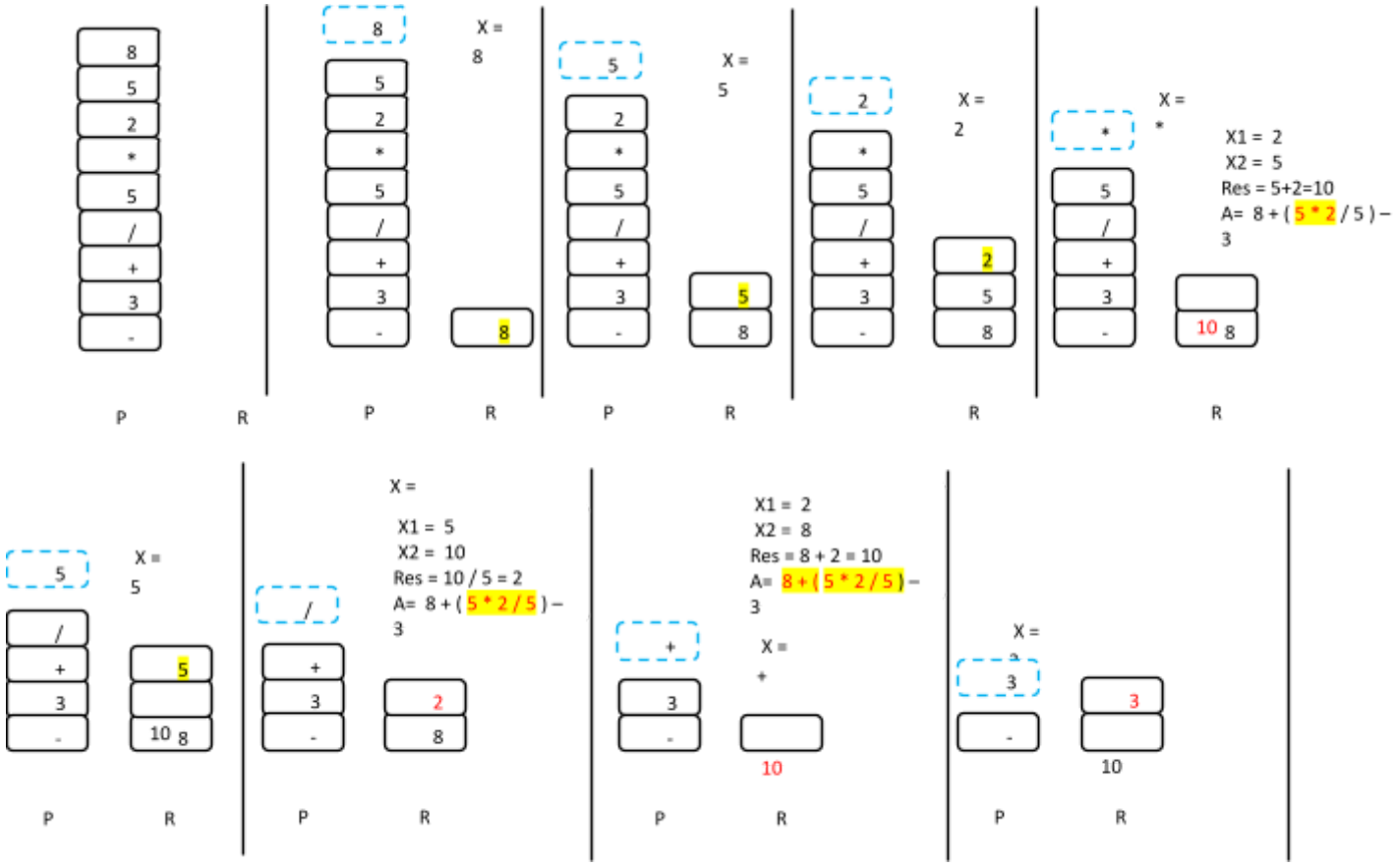
Algorithme d'Evaluation

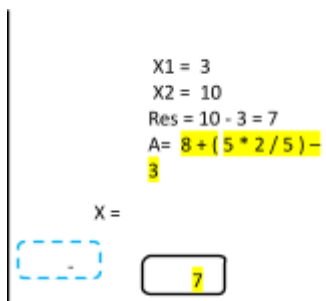
Début

```
R <- Init_pile ();
Tant que (non PileVide(P))
Faire
    Désempiler (P, x) ;
    Si (Operande (x)) alors Empiler (R, x) ;
                        Sinon /* c'est un opérateur */
                            Désempiler (R, x1) ;
                            Désempiler (R, x2) ;
                            Res = Opération (x2, x1, x) ;
                            Empiler (R, Res) ;
    Fsi ;
Fait ;
```

Fin

Exemple : soit l'expression suivante A= 8 + (5 * 2 / 5) – 3. Sa forme postfixée est sauvegardée dans la pile P ci-contre. Appliquant l'algorithme ci-dessus pour trouver le résultat de A.





Exercice 1 : évaluer l'expression suivante : A= ((2*(5+3)/2)-((10+5*3)/5))

Exercice 2 : Etant donné une matrice carrée T[N][N] où N > 1 et N est impair. On veut représenter un triangle dans la moitié supérieure de cette matrice en mettant à jour certaines cases de la matrice T. Pour effectuer le remplissage du triangle on procède comme suit : à chaque fois qu'on met à jour une case par un '1', on sauvegarde les coordonnées entourant cette case (gauche, droite, haut et bas) dans une pile.

1. Définissez le type « coordo » : les coordonnées (x, y) d'une matrice et donnez le type de la pile.
2. Réécrivez les deux actions Désempiler et Empiler en tenant compte du nouveau type « coordo ».
3. Ecrivez une action qui place des '1' dans la partie supérieure de la matrice T en suivant le principe énoncé ci-dessus.
   Note : on prend toujours comme configuration initiale les deux côtés du triangle et une case de départ au milieu du triangle exemple : T [2][4] ≠1 (voir par exemple la figure 1). Le résultat final est donné par la figure 2.

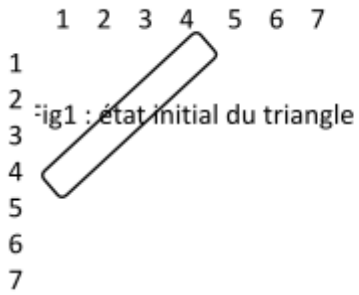


Fig1 : état initial du triangle

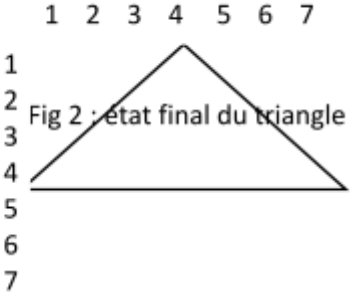


Fig 2 : état final du triangle

Solution exercice 2 :

1/ Définissez le type « coordo » : les coordonnées (x, y) d'une matrice et donnez le type de la pile.

Type coordo : Enregistrement l : entier ; c : entier ; Feng ;	Type pile : ^ cellule ; Type cellule : Enregistrement inf : coordo ; svt : pile ; Feng ;

2/ Réécrivez les deux actions Désempiler et Empiler en tenant compte du nouveau type « coordo ».

Ajout d'un élément

Procédure Empiler(ES/ p Pile, E/ x :<Type_elt>) tmp : pile tmp<- creer_noeud ( ) ; si (tmp ≠ nil) alors tmp^.inf <- x ; tmp^.svt <- p ; p<- tmp ; Fsi ; Fin ;	**Procédure** Empiler(ES/ p Pile, E/ x : coordo) tmp : pile Allouer (tmp) ; si (tmp ≠ nil) alors tmp^.inf <- x ; tmp^.svt <- p ; p<- tmp ; Fsi ; Fin ;

Suppression d'un élément

Procédure désempiler(ES/ p : pile, ES/ x :<Type_elt>) Début tmp : pile ; // test pile_vide se fait dans le prog x <- sommet_pile(p) ; tmp<- p ; p<- p^.svt ; Libérer (tmp) ; Fin ;	**Procédure** désempiler(ES/ p : pile, ES/ x : coordo) Début tmp : pile ; // test pile_vide se fait dans le prog x <- sommet_pile(p) ; tmp<- p ; p<- p^.svt ; Libérer (tmp) ; Fin ;

3. Ecrivez une action qui place des ‘1’ dans la partie supérieure de la matrice T en suivant le principe énoncé ci-dessus.

```
Constante N 15 ;
Procédure Remplir_mat_triangle_sup(E/ T[N][N] : entier, E/ lc : coordo)
Début        // lc : coordo du point de départ
  P : Pile ;
  x, y : coordo ;
  borne : entier ;

  borne <-  (N+1)/2 ;   // la borne inférieure du triangle
  x<- lc ;

  Empiler (p, lc) ;
  Tant que (pile_vide(p)=faux)
   Faire
      Désempiler (p, x) ;
      Si (T [x. l] [x. c]=0 et x. l<= borne) alors
                T [x. l] [x. c] <- 1 ;
                y <- x ;     y. l <- x. l-1 ;                    empiler (p, y) ;  //haut
                y <- x ;     y. l <- x. l+1 ;                     empiler (p, y) ;  //Bas
                y <- x ;                    y. c <- x. c+1 ;    empiler (p, y) ; //Droite
                y <- x ;                    y. c <- x. c -1;     empiler (p, y) ;  //Gauche
      Fsi ;
   Fait ;

Fin
```