INFO0947: Polylignes, Milestone 1

Groupe 06: Maxime Deravet, Luca Matagne

Table des matières

1 TAD : Point2D

1.1 Signature

TYPE : Point2D
UTILISE : Réels
OPERATIONS :

- Create : Réels × Réels → Point2D C [1]
- GetX : $Point2D$ → Réels O
- GetY : $Point2d$ → Réels O
- EuclDist : $Point2D$ × $Point2d$ → Réels O
- Translate : $Point2D$ × $Point2d$ → $Point2D$ T
- Rotate : $Point2D$ × $Point2d$× Réels → $Point2D$ T

1.2 Sémantique

PRECONDITIONS :
AXIOMES : $\forall X, Y \in Reels$

- GetX(Create(a,b)) = a
- GetY(Create(a,b)) = b
- EuclDist(X,Y) = $\sqrt{(GetX(X) - GetX(Y))^2 + (GetY(X) - GetY(Y))^2}$
- GetX(Translate(U,V)) = GetX(U) + GetX(V)
- GetY(Translate(U,V)) = GetY(U) + GetY(V)
- GetX(Rotate(U,V,f)) = $cos(f) \times (GetX(U) - GetX(V)) - sin(f) \times (GetY(U) - GetY(V)) + GetX(V)$
- GetY(Rotate(U,V,f)) = $sin(f) \times (GetX(U) - GetX(V)) + cos(f) \times (GetY(U) - GetY(V)) + GetY(V)$

1.3 Structure

Un point est créé sur base du couple de ses coordonnées (x,y).

```
struct Point2D{
    float x;
    float y;
};
```

1.4 Focntions et procédures

```

/*
 * @pre: /
 * @post: (get_x(create_Point2D) = x
 * ∧
 * get_y(create_Point2D) = y)
 */
Point2D* CreatePoint2D(float x, float y);
/*
 * @pre: A != NULL
 * @post: A = A_0 ∧ get_x = x
```

1. Les lettres vertes permettront, durant tout ce rapport, de mettre en évidence les observateurs et les opérations internes.

```
 */
float get_x(Point2D* A);
/*
 * @pre: A != NULL
 * @post: A = A_0 ∧ get_y = y
 */
float get_y(Point2D* A);
/*
 * @pre: A != NULL ∧  B != NULL
 * @post: A = A_0 ∧ B = B_0 ∧ EuclDist = sqrt((X_a − X_b) + (Y_a − Y_b))
 */
unsigned float EuclDist(Point2D* A, Point2D* B);
/*
 * @pre: A != NULL ∧ B != NULL
 * @post: A = Translate_(A,B) ∧ B = B_0
 */
void TranslatePoint2D(Point2D* A, Point2D* B);
/*
 * @pre: A != NULL ∧ B != NULL
 * @post: A = Rotate_(A,B),x ∧ B = B_0
 */
void RotatePoint2D(Point2D* A, Point2D* B, float x);
```

2 TAD : Polyligne

2.1 Signature

TYPE : Polyligne
UTILISE : Point2D, Réels, Naturels, Boolean
OPERATIONS :

- Create : $Point2D \times Point2D \times Boolean \rightarrow Polyligne$
- Close : $Polyligne \rightarrow Polyligne$ T
- Open : $Polyligne \rightarrow Polyligne$ T
- IsOpen : $Polyligne \rightarrow Boolean$ O
- NbrPoint : $Polyligne \rightarrow Naturels$ O
- GetPoint : $Polyligne \times Naturels \rightarrow Point2D$ O
- Length : $Polyligne \times Reels$ O
- AddPoint : $Polyligne \times Point2D \rightarrow Polyligne$T
- SuppPoint : $Polyligne \rightarrow Polyligne$ T
- PolyTranslate : $Polyligne \times Point2D \rightarrow Polyligne$ T
- PolyRotate : $Polyligne \times Reels \times Point2D \rightarrow Polyligne$ T

2.2 Sémantique

PRECONDITIONS : $\forall P \in Polyligne, \forall A \in Point2D, \forall x \in Naturels, \forall n \in Boolean$

- SuppPoint(P) est défini ssi$2 \leq NbrPoint(P)$
- GetPoint(A,x) est défini ssi $0 \leq x < NbrPoint(P)$
- AddPoint(P,A) est défini ssi $2 \leq NbrPoint(P)$

AXIOMES : $\forall P \in Polyligne, \forall A, B, C \in Point2D, \forall x \in Naturels, \forall n \in Boolean$

- Open(Create(A,B,n)) = Create(A,B,n) [2]
- Close(Create(A,B,n)) = AddPoint(Create(A,B,False),C)

2. Les Polylignes seeront toujours ouvertes à la création

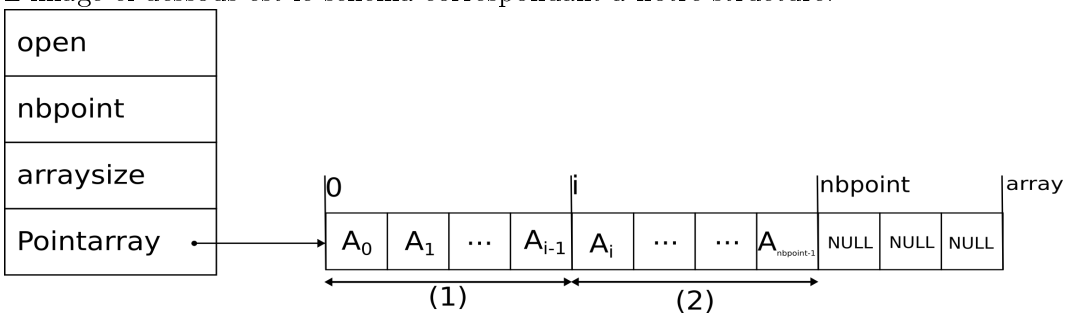
— NbrPoint(Create(A,B,n)) = 2
— NbrPoint(AddPoint(P,C, x)) = NbrPoint(P) + 1
— NbrPoint(SuppPoint(P, x)) = NbrPoint(P) - 1
— NbrPoint(Translate(P, A)) = NbrPoint(P)
— NbrPoint(Rotate(P, A)) = NbrPoint(P)
— GetPoint(Create(A,B,n), 0) = A
— GetPoint(AddPoint(P, C), NbrPoint(P) = C
— GetPoint(PolyTranslate(P, C), x) = Translate(GetPoint(P, x), C)
— GetPoint(PolyRotate(P, C), x) = Rotate(GetPoint(P, x), C)
— Length(Create(A,B,n)) = EuclDist(A,B)
— Length(P) = $\sum_{x=0}^{NbrPoint(P)-1} EuclDist(GetPoint(P,x), GetPoint(P,x+1))$
— Length(Close(P)) = IF(IsOpen(P) = False) : Length(P=P_0 [3]) ELSE : Length(P_0) + EuclDist(GetPoint(P, NbrPoint(P_0)), GetPoint(P, NbrPoint(P)))
— Length(Open(P)) = IF(IsOpen(P) = True) : Length(P=P_0) ELSE : Length(P_0) - EuclDist(GetPoint(P, NbrPoint(P_0)), GetPoint(P, NbrPoint(P)))
— Length(AddPoint(P)) & Length(SuppPoint(P)) [4]

2.3 Implémentation par tableau

2.3.1 Structure

```
struct Polyline{
   boolean open;
   unsigned nbpoint;
   unsigned arraySize;
   Point2D** pointArray;
};
```

L'image ci-dessous est le schéma correspondant à notre structure.



2.3.2 fonctions et procédures

```
/*
 * @pre: A != NULL ∧ B != NULL ∧
 * @post: A = A0 ∧ B = B0 ∧ open=open ∧ create_Polyligne = P ∧
 * nbpoint(P) = NbrPoint(P) ∧ length(P) = Length(P)
 */
Polyline* CreatePolyline(Point2D* A, Point2D* B, boolean open);
/*
```

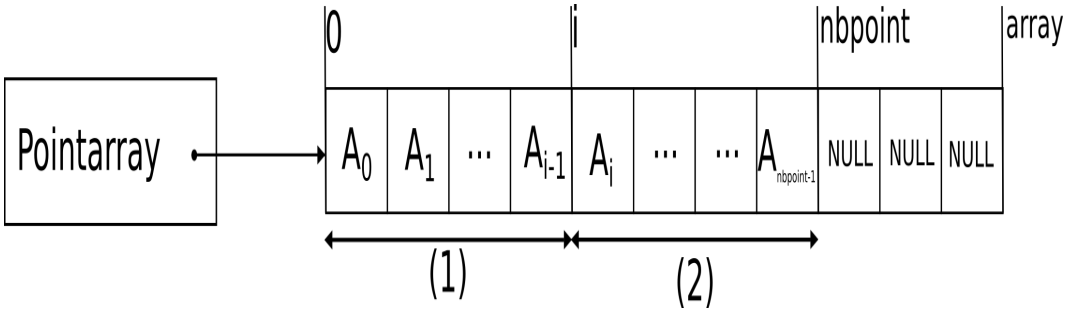
3. Ici, l'indice "0" est utilisé pour parler de l'état initial de la polyligne, un raisonnement similaire pourrait être utilisé dans le suite du rapport

4. Ces deux cas ne sont pas oubliés mais sont bel et bien pris en compte dans les cas "Length(Open(P))" et "Length(Close(P)) car comme nous le verrons, fermer une polyligne ouverte lui ajoute un point (raisonnement opposé pour l'ouverture d'une polyligne"

```
 * @pre: P != NULL
 * @post: P = P_0 ∧ open = False ∧ nbpoint = nbpoint_0
 */
void Open(Polyline* P);
/*
 * @pre: P != NULL
 * @post: P = P_0 ∧ open(P) = True ∧ nbpoint = nbpoint_0
 */
void Close(Polyline* P);
/*
 * @pre: P != NULL
 * @post: P = P_0 ∧
 */
void IsOpen(Polyline* P);
/*
 * @pre: P != NULL
 * @post: P=P_0 ∧ nbpoint = NbrPoint(P)
 */
unsigned NbrPoint(Polyline* P);
/*
 * @pre: P! = NULL∧  numero < nbpoint
 * @post: P = P_0∧GetPoint = A_numero
 */
Point2D GetPoint(Polyline* P, unsigned numero);
/*
 * @pre:  P != NULL ∧ A != NULL
 * @post: A = A_0 ∧ open = open_0 ∧ nbpoint = nbpoint_0 + 1
 */
void AddPoint2D(Polyline* P, Point2D* A);
/*
 * @pre:  P != NULL ∧ A != NULL
 * @post: A = A_0 ∧ open = open_0 ∧ nbpoint = nbpoint_0 − 1
 */
 void SuppPoint2D(Polyline* P);
```

Les fonctions et procédures reprisent ci-dessus ne nécessitent pas d'invariant spécifique. On pourrait imaginer qu'il en faut un pour l'ajout et la suppression d'un point mais nous prenons la décision de n'ajouter et de supprimer que le dernier point à chaque appel de cet fonction. Le seul élément de l'ajout/la suppression d'un point qui pourrait encore nécessiter un invariant est le calcul de la longueur de la polyligne. Une fonction étant totalement dédiée à ce clacul, nous allons présenter l'invariant pour celle-ci.

2.3.3 Invariant et spécifications : Length



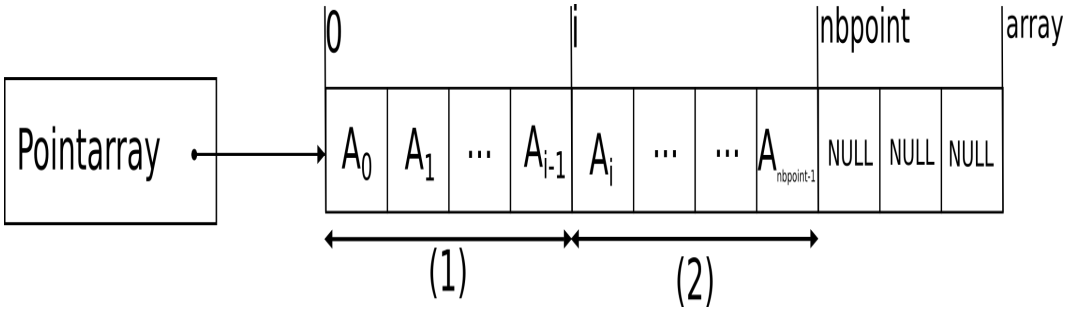
(1) : Length $= \sum_{x=0}^{i-2} EuclDist(A_x, Ax+1)$
(2) : Length à calculer

L'invariant formel qui en découle est le suivant :

$pointArray = pointArray_0 \wedge \forall x, 0 \leq x \leq i-2, Length = \sum_{x=0}^{i-2} EuclDist(A_x, Ax+1) \wedge arraySize = arraySize_0 \wedge nbpoint = nbpoint_0$

```
/*
 * @pre:  P != NULL
 * @post: P = P_0 ∧ Length = Σ_{x=0}^{NbrPoint(P)-1} EuclDist(GetPoint(P,x), GetPoint(P,x+1))
 */
float Length(Polyline* P);
```

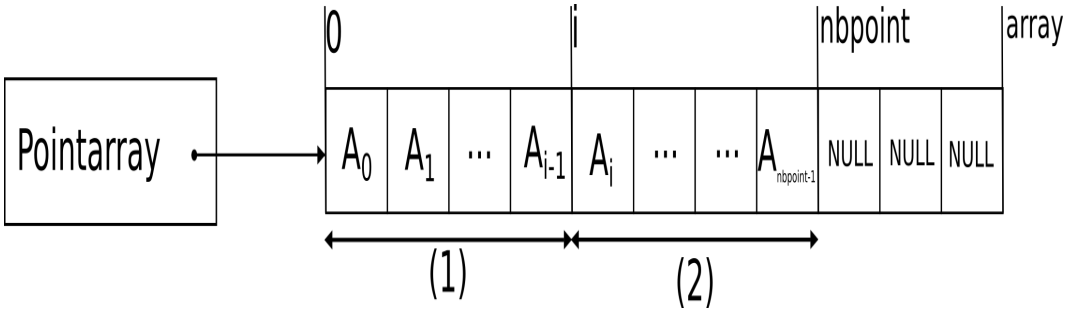
2.3.4 Invariant et spécifications : PolyTranslate



(1) : La translation de chaque point est effectuée
(2) : Translation à appliquer sur les points restants
L'invariant formel qui en découle est le suivant :
$\forall x, 0 \leq x \leq i-1, A_x = Translate((A_x)_0, K)$ [5]$) \wedge arraySize = arraySize_0 \wedge nbpoint = nbpoint_0$

```
/*
 * @pre:  P! = NULL ∧ K! = NULL
 * @post: P = P_0 ∧ C = C_0 ∧ ∀x, 0 ≤ x ≤ i-1, A_x = Translate((A_x)_0, K)
 */
void PolyTranslate(Polyligne* P, Point2D K);
```

2.3.5 Invariant et spécifications : PolyRotate



(1) : La rotation de chaque point est effectuée
(2) : Rotation à appliquer sur les points restants
L'invariant formel qui en découle est le suivant :
$\forall x, 0 \leq x \leq i-1, A_x = Rotate((A_x)_0, K)$ [6]$) \wedge arraySize = arraySize_0 \wedge nbpoint = nbpoint_0$

```
/*
 * @pre:  P! = NULL ∧ K! = NULL
 * @post: P = P_0 ∧ C = C_0 ∧ ∀x, 0 ≤ x ≤ i-1, A_x = Rotate((A_x)_0, K)
```

5. K est le point de référence lors de cette Translation
6. K est le point de référence lors de cette Rotation

```
 */
void PolyRotate(ePolyligne* P, Point2D K);
```

2.4 Implémentation par liste chainée

2.4.1 structure

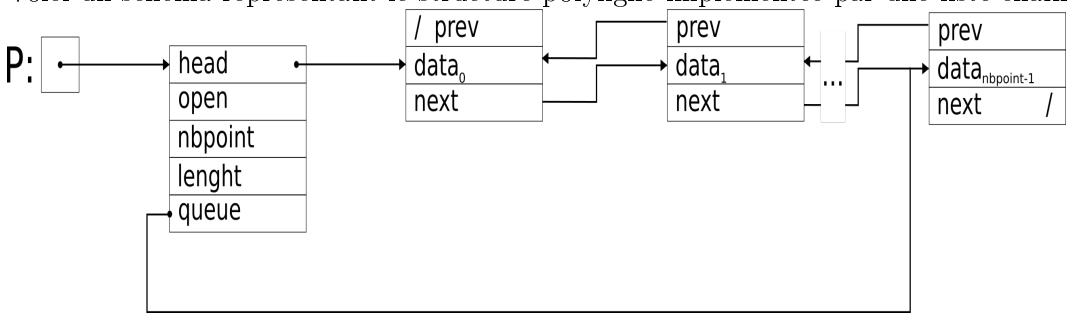
La structure Point2D est déjà définie plus haut mais il est important de préciser qu'inclure cette structure(Point2D) dans une liste chaînée nécessite de définir une nouvelle structure.

```
struct Point{
    Point* prec;
    2DPoint* point;
    Point* suiv;
};
```

Dans la liste chaînée gérant la polyligne, nous décidons de garder la longueur et le nombre de point dans une cellule d'en-tete afin d'y accéder plus facilement.

```
struct Polyline{
   char open;
   unsigned nbrpoint;
   unsigned float length;
   Point* head;
};
```

Voici un schéma représentant le structure polyligne implémentée par une liste chaînée.

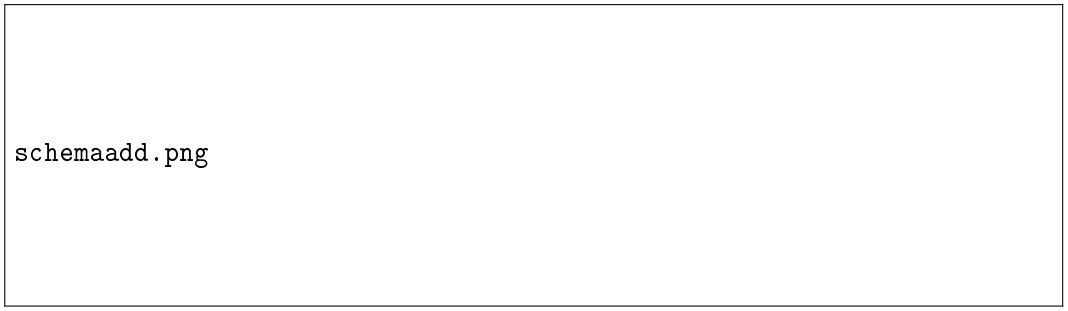


2.4.2 Fonctions et procédures

Les spécifications des fonctions et des procédures sont semblables à l'implémentation par tableau, nous allons juste reprendre deux schémas qui expliquent le fonctionnement des fonctions qui nécessitaient un invariant. Deux schémas et non trois car la longueur de notre polyligne sera stockée dans la cellule d'en tëte de notre liste. Il ne sera donc plus nécessaire de parcourir toute la liste afin de la calculer

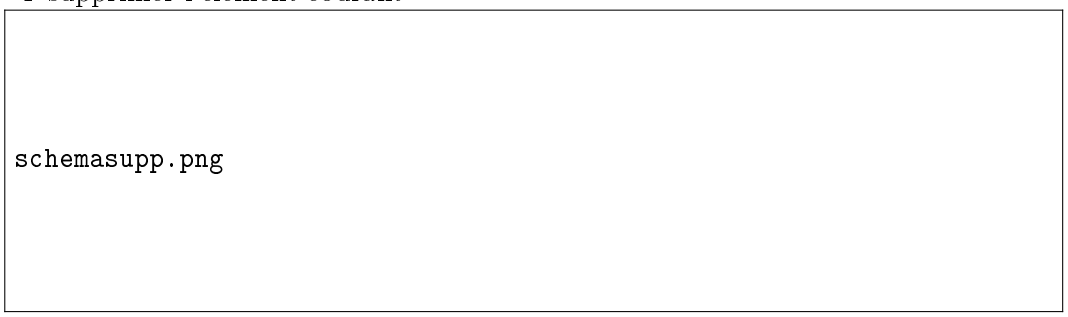
AddPoint Souvenons nous que notre fonction d'ajout va ajouter notre nouveau point à la fin de la liste. les étapes pour cela seront les suivantes :

1 Amener l'élément courant au dernier élément de la liste
2 Initialiser le champ "suiv" de la nouvelle cellule vers la première de la liste
3 Initialiser le champ "prec" de la nouvelle cellule vers l'élément courant
4 Stocker l'adresse de la nouvelle cellule dans dans le champ "suiv" de l'élément courant
5 Stocker l'adresse de la nouvelle cellule dans le champ "prec" du premier élément de la liste



SuppPoint Souvenons nous que notre fonction d'ajout va ajouter notre nouveau point à la fin de la liste. les étapes pour cela seront les suivantes :

1 Amener l'élément courant au dernier élément de la liste
2 Changer le champ "suiv" de l'avant dernier élément pour le champ "suiv" de l'élément courant
3 Initialiser le champ "prec" du premier élément pour le champ "prec" de l'élément courant
4 supprimer l'élément courant



3 Complexité

4 Comparaison des implémentations