INFO0947: Polylignes

Groupe 06: Maxime Deravet, Luca Matagne

Table des matières

1 Remarque sur la compréhension du problème

Nous nous sommes retrouvé face à une divergence de compréhension sur ce que voulais dire fermer/ouvrir une polyligne. Après une longue discussion à ce sujet, nous avons décidé que pour nous, fermer une polyligne ne consistait pas juste à relier la cellule du dernier point à la cellule du premier point mais bien à **ajouter** un point en fin de liste ayant **les mêmes** coordonnées que le premier point. Cela nous permet d'ajouter un nouveau point à une polyligne déjà fermée par exemple. Il est important de comprendre notre vision du problème pour comprendre comment fonctionne nos implémentations.

2 TAD : Point2D

2.1 Signature

TYPE : Point2D
UTILISE : Réels
OPERATIONS :

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

2.2 Sémantique

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

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

2.3 Structure

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

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

1. Les lettres vertes permettront, durant tout ce rapport, de mettre en évidence les observateurs (O), le créateur (C) et les transformateurs (T).

2.4 Fonctions et procédures

```
1
2  /*
3   * @pre: /
4   * @post: (get_x(create_Point2D) = x
5   * ∧
6   * get_y(create_Point2D) = y)
7   */
8  Point2D* CreatePoint2D(float x, float y);
9  /*
10  * @pre: A != NULL
11  * @post: A = A0∧get_x = A− > x
12  */
13 float get_x(Point2D* A);
14 /*
15  * @pre: A != NULL
16  * @post: A = A0∧get_y = A− > y
17  */
18 float get_y(Point2D* A);
19 /*
20  * @pre: A != NULL ∧  B != NULL
21  * @post: A = A0 ∧ B = B0 ∧ EuclDist = √((Xa − Xb) + (Ya − Yb))
22  */
23 unsigned float EuclDist(Point2D* A, Point2D* B);
24 /*
25  * @pre: A != NULL ∧ B != NULL
26  * @post: A = Translate(A,B) ∧ B = B0
27  */
28 void TranslatePoint2D(Point2D* A, Point2D* B);
29 /*
30  * @pre: A != NULL ∧ B != NULL
31  * @post: A = Rotate(A,B),x ∧ B = B0
32  */
33 void RotatePoint2D(Point2D* A, Point2D* B, float x);
```

3 TAD : Polyligne

3.1 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

3.2 Sémantique

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

- SuppPoint(P) est défini ssi2 $\leq NbrPoint(P)$
- 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)
- 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]

3.3 Implémentation par tableau

3.3.1 Structure

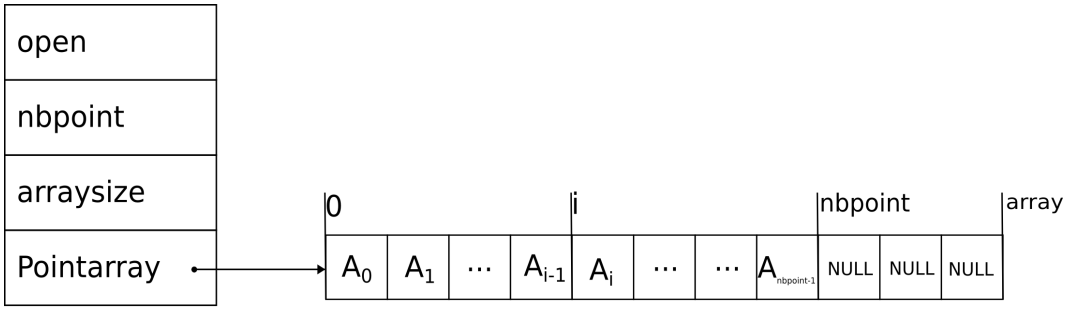
```
struct Polyligne{
   boolean open;
   unsigned nbpoint;
   unsigned arraySize;
   Point2D** pointArray;
};
```

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

2. Les Polylignes seeront toujours ouvertes à la création

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"

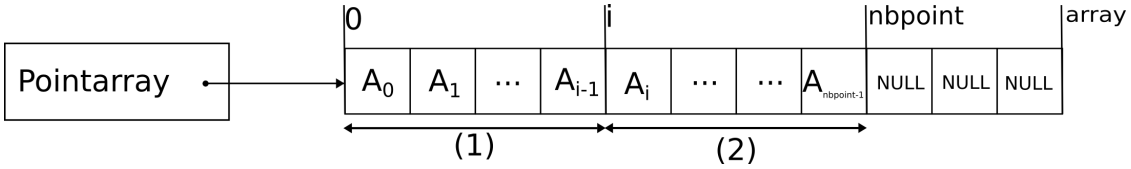


3.3.2 Fonctions et procédures

```
/*
 * @pre: A != NULL ∧ B != NULL ∧ A != B
 * @post: A = A0 ∧ B = B0 ∧ P->open=True ∧ create_Polyligne(P) = P ∧
 * P->nbpoint = NbrPoint(P)
 */
Polyligne* CreatePolyligne(Point2D* A, Point2D* B);
/*
 * @pre: P != NULL
 * @post: P = P0 ∧ P->open = True ∧ P− > nbpoint = P0− > nbpoint − 1
 */
void Open(Polyligne* P);
/*
 * @pre: P != NULL
 * @post: P = P0 ∧ P->open = False ∧ P− > nbpoint = P0− > nbpoint + 1
 */
void Close(Polyligne* P);
/*
 * @pre: P != NULL
 * @post: P = P0 ∧ IsOpen(P)=P->open
 */
Boolean IsOpen(Polyligne* P);
/*
 * @pre: P != NULL
 * @post: P = P0 ∧ NbrPoint(P) = P->nbpoint
 */
unsigned int NbrPoint(Polyligne* P);
/*
 * @pre: P != NULL ∧ number < nbpoint
 * @post: P = P0 ∧ GetPoint(P) = Anumber
 */
Point2D GetPoint(Polyligne* P, unsigned number);
/*
 * @pre:  P != NULL ∧ A != NULL
 * @post: A = A0 ∧ P− > nbpoint = P0− > nbpoint + 1
 */
Polyligne* AddPoint2D(Polyligne* P, Point2D* A);
/*
 * @pre:  P != NULL ∧ A != NULL
 * @post: A = A0 ∧ P− > nbpoint = P− > nbpoint − 1
 */
Polyligne* DeletePoint2D(Polyligne* 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 cette fonction.

3.3.3 Invariant et spécifications : Length

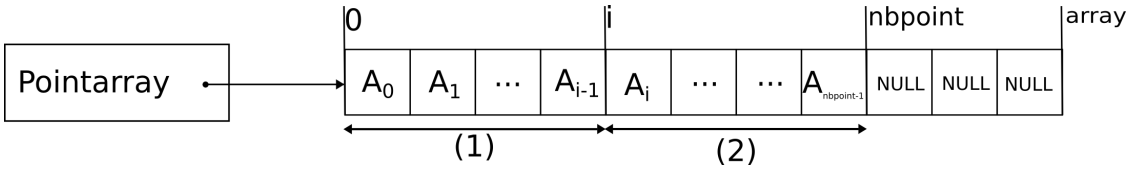


(1) : Length = $\sum_{x=0}^{i-2} EuclDist(A_x, A_{x+1})$
(2) : Length à calculer

L'invariant formel qui en découle est le suivant :
$pointArray = pointArray_0 \wedge \forall x, 0 \leq x \leq i-2, Length = \sum_{x=0}^{i-2} EuclDist(A_x, A_{x+1}) \wedge arraySize = arraySize_0 \wedge nbpoint = nbpoint_0$

```
/*
 * @pre:  P != NULL
 * @post: P = P_0 ∧ Length(P) = ∑_{x=0}^{NbrPoint(P)-2} EuclDist(GetPoint(P,x), GetPoint(P,x+1)) ∧ P->open = P_0->open ∧ P->nbpoint = P_0->nbpoint
 */
float Length(Polyligne* P);
```

3.3.4 Invariant et spécifications : PolyTranslate

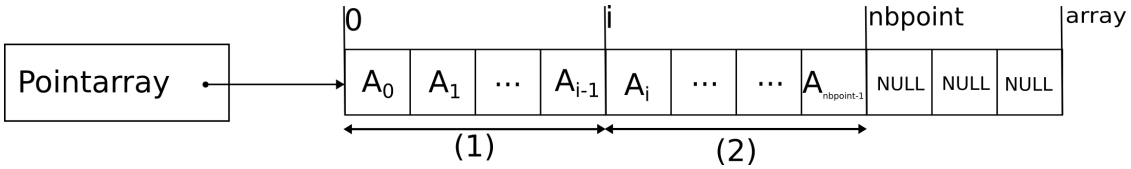


(1) : La translation de chaque point est effectuée
(2) : Translation à appliquer sur les points restants

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 ∧ K = K_0 ∧ ∀x, 0 ≤ x ≤ P->nbpoint - 1, A_x = Translate((A_x), K)
 */
Polyligne* PolyTranslate(Polyligne* P, Point2D K);
```

3.3.5 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 :
$\forall x, 0 \leq x \leq i-1, A_x = Rotate((A_x)_0, K, u)$ [6]$) \wedge arraySize = arraySize_0 \wedge nbpoint = nbpoint_0$

5. K est le point de référence lors de cette Translation
6. K est le point de référence lors de cette Rotation et u est l'anglede référence de cette rotation

```
/*
 * @pre:  P! = NULL ∧ K! = NULL
 * @post: P = P_0 ∧ K = K_0 ∧ u = u_0 ∧ ∀x, 0 ≤ x ≤ P− > nbpoint − 1, A_x = Rotate(A_x, K, u)
 */
Polyligne* PolyRotate(Polyligne* P, Point2D K, float u);
```

3.4 Implémentation par liste chainée

3.4.1 Structure

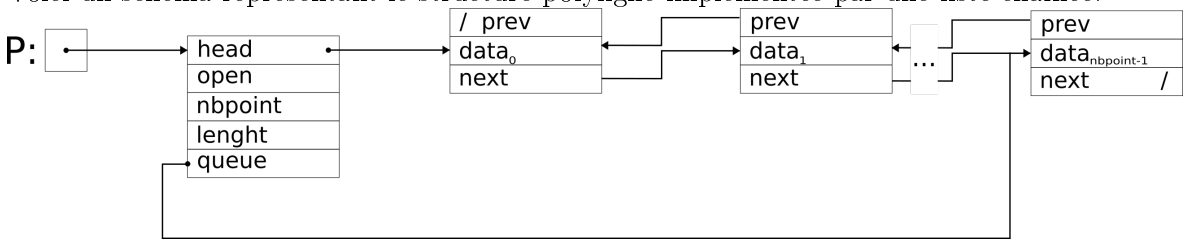
La structure Point2D est déjà définie plus haut mais il est important de préciser qu'inclure cette structure(Point2D) dans une liste chaînée nécessite de définir une nouvelle structure représentant chaque cellule au sein de la liste. Notre liste est doublement chaînée ce qui implique que chaque cellule à un pointeur vers la cellule suivante mais aussi un pointeur vers la cellule précédente.

```
struct cell_t{
    cell* prev;
    Point2D *data;
    cell* next;
};
```

Nous décidons de donner une cellule d'en-tête à notre liste afin d'avoir accès à des données utiles comme la longueur de la polyligne, le nombre de points qui la composent et l'état d'ouverture. Dans cette cellule d'en-tête, nous mettons nos accès à notre liste : Un pointeur vers la première cellule (head) et un pointeur vers la dernière cellule (tail).

```
struct Polyligne_t{
    Boolean open;
    unsigned int nbpoint;
    float length;
    cell* head;
    cell* tail;
};
```

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

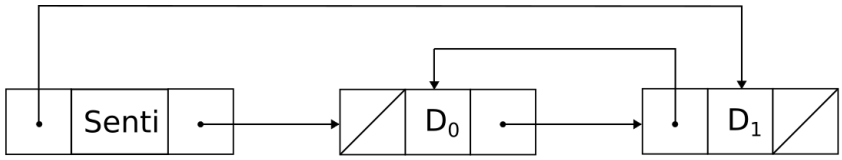


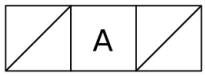
3.4.2 Fonctions et procédures

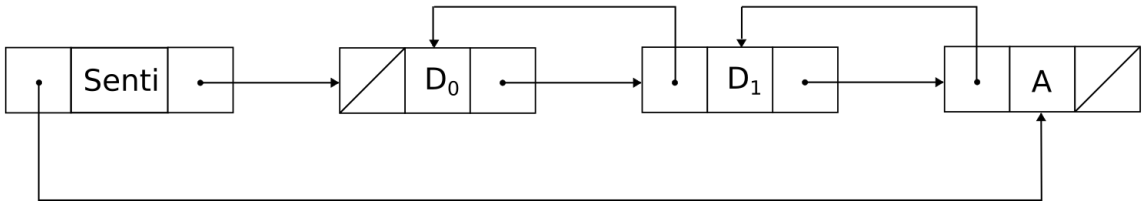
Les spécifications des fonctions et des procédures sont semblables à l'implémentation par tableau, nous allons juste reprendre deux schémas qui expliquent le fonctionnement des fonctions qui nécessitaient un invariant. Deux schémas et non trois car la longueur de notre polyligne sera stockée dans la cellule d'en-tête 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 Copier le point entré en argument dans le champ utile d'une nouvelle cellule
2 Initialiser le champ *next* de la denière cellule vers la nouvelle
3 Initialiser le champ *prev* de la nouvelle cellule vers la dernière
4 Stocker l'adresse de notre nouvelle cellule dans le pointeur de fin de la cellule d'en-tête *tail*.

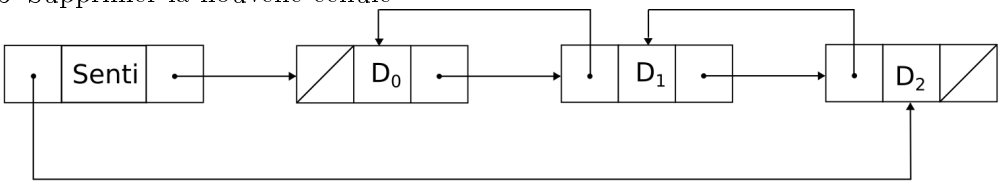


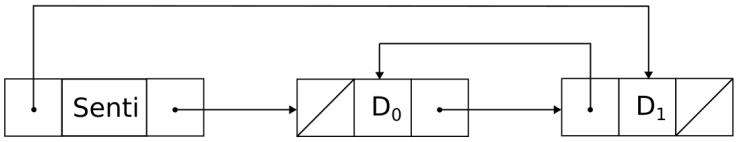




SuppPoint Souvenons-nous que notre fonction de suppression va supprimer le dernier point de la liste. Les étapes pour cela seront les suivantes :

1 Copier la dernière de la liste dans une nouvelle cellule
2 Vider le champ utile de la dernière cellule
3 Vider le champ *next* de la dernière cellule
4 Stocker le champ *prev* de notre nouvelle cellule dans le pointeur de fin de la cellule d'en-tête *tail*.
5 Supprimer la nouvelle cellule





3.4.3 Justification des constructions récursives

Rappelons que la construction récursive est basée sur 3 questions (cfr. INFO0947 Chap. 4) :

1 Peut-on trouver un paramètre récursif ?
2 Peut-on trouver un cas de base ?
3 Peut-on exprimer le problème de manière récursive ?

GetPoint Nous pouvons répondre à la première question de la construction récursive par l'affirmative car la fonction GetPointRec(P, x) prend en entrée le point courant dans le parcours de la Polyligne mais aussi l'indice du point que l'on veut dans la polyligne. Cet indice semble parfait pour être notre paramètre récursif.

Ce Paramètre étant un nombre, il est aisé de trouver un cas de base à notre problème : x=0.

On peut exprimer récursivement le problème. Lorsque x est égal à 0, il nous suffit de retourner le point courant passé en paramètre. Dans les autres cas, on rappelle la fonction avec l'indice diminué de 1.

PolyTranslateRec ET PolyRotateRec Ces deux fonction étant relativement semblable dans leur approche constructive, nous les traiterons ensemble. Ces deux fonctions sont des fonctions pour lesquelles la récursivité nous sert à parcourir la polyligne. Le paramètre récursif est donc la cellule sur laquelle on applique la fonction.

Le cas de base est le cas où la cellule courante est la dernière cellule, à savoir, le cas où $C->next == \text{NULL}$.

Quand $C->next == \text{NULL}$, nous appliquons la translation/rotation de type $Point2D$ sur le point courant. Dans les autre cas, nous rappelons notre fonction récursive avec la cellule suivante $(C - next)$ dans la polyligne. Notre problème est donc défini récursivement.

4 Complexité

Dans cette section nous allons uniquement nous intéresser aux processus dont la complexité change selon l'implémentation

4.1 GetPoint

4.1.1 Implémentation par tableau

```
Point2D* GetPoint(Polyligne* P, unsigned int number){
    assert(P!=NULL && number < P->nbpoint);

    return P->pointArray[number];
}//end GetPoint()
```

Ici, la complexité est triviale. Ce processus ne comporte que deux instructions. Elle est donc **constante** $(O(2))$

4.1.2 Implémentation par liste

En comparaison, le processus GetPoint pour l'implémentation sous forme de liste se définit comme suis :

```
static Point2D* GetPointRec(cell* C, unsigned int number){
    if(number==0)
        return(C->data);
    else{
        return(GetPointRec(C->next, number-1));
    }
}

Point2D* GetPoint(Polyligne* P, unsigned int number){
    assert(P!=NULL && number < P->nbpoint);

```

```
12      if(P->tail==NULL)
13          return NULL;
14
15      return GetPointRec(P->head, number);
16  }//end GetPoint()
```

La fonction appelée en premier est GetPoint. Cette dernière est de complexité constante car elle ne réalise qu'une simple comparaison, et un appel à une autre fonction GetPointRec. C'est cette fonction qui va nous intéresser en particulier.

On y trouve une structure $if\ else$. Sous le if, se cache le cas de base, qui est un unique `return`, de complexité 1. Ensuite sous le $else$, l'appel récursif. L'appel se fait en diminuant à chaque fois l'argument `number` d'une seule unité. `number` étant le paramètre évalué pour déterminer le cas de base (number $==$ 0) , le nombre d'appel sera toujours plus petit que N le nombre d'élément dans notre polyligne. La complexité de cette partie vaut donc $O(N)$. Au total, nous avons une complexité **linéaire.**

4.2 Lenght

4.2.1 Implémentation par tableau

```
1   float Length(Polyligne* P){
2       assert(P!=NULL);
3
4       float length = 0;
5
6       for(unsigned int i=0; i<P->nbpoint-1; i++){
7           length += EuclDist(P->pointArray[i], P->pointArray[i+1]);
8       };
9
10      return length;
11  }//end Length()
```

Cette fonctions ne contient que quelques instructions simples, et une boucle appelant $nbpoint-1$ fois la fonction `EuclDist`, qui permet de calculer la distance euclidienne entre deux points, et qui n'est composé que d'instructions simples elle aussi. Au total nous obtenons donc :

$$\begin{aligned} O &= 2 + ((nbpoint - 1) \times 1) + 1 \\ &= (nbpoint - 1) + 3 \end{aligned} \tag{1}$$

Borné par $O(nbpoint - 1)$.
La complexité est donc **linéaire**

4.2.2 Implémentation par liste

```
1   float Length(Polyligne* P){
2       assert(P!=NULL);
3
4       return(P->length);
5   }//end length()
```

La longueur de la polyligne étant stockée dans la cellule d'en-tête, et actualisée à chaque modification de la taille (ajout ou suppression d'un point), il nous suffit d'accéder à ce champ. Il s'agit d'un instruction simple. La complexité est donc **constante**. $O(1)$

5 Tests unitaires

5.1 Length

Dans le but de tester la fonction `Length`, nous créons plusieurs points et une polyligne. Cela nous permettra de recueillir la longueur de la polyligne avant et après ajouts de points afin de vérifier que la fonction produit bien le résultat attendu.

5.2 NbrPoint

Nous commençons par créer une polyligne composée de 2 points. Nous vérifions déjà cette valeur grâce à `NbrPoint`. Ensuite nous ajoutons et supprimons des points, tout en vérifiant le nombre de points de la liste à chaque étape intermédiaire

5.3 AddPoint

Il est tout naturel que notre fonction `AddPoint` doit, elle aussi, fonctionner correctement vu que nous l'utilisons dans les test ci-dessus. Pour cela, nous allons ajouter des points à une polyligne et regarder les coordonnées (x,y) du dernier point de cette polyligne après chaque nouvel ajout.

6 Comparaison de la liste et du tableau.

Le Problème qui nous a le plus sauté aux yeux dans l'implémentation par tableau est la taille limitée et prédéfinie du tableau.

Étant limitée, nous devons en permanence vérifier que nous n'avons pas rempli le tableau quand on souhaite ajouter un nouveau point. Si c'est le cas, alors, nous devons faire appel à une fonction de ré-allocation de mémoire pour agrandir notre tableau.

De l'autre coté, le fait que la taille soit prédéfinie nous amène, la majorité du temps, à avoir des cases vides dans notre tableau comme on peut le voir dans le schéma représentant notre structure dans la section 3.3.5. Nous devons aussi tenir deux tailles à jour au lieu d'une seule : le nombre de point et la taille du tableau.

L'implémentation par tableau a des inconvénients mais elle a aussi des avantages ! Elle permet d'éviter la récursivité dans plusieurs fonctions : la translation , la rotation, l'accesseur à un point. Dans le cas de l'accesseur, on voit même que cette implémentation permet de passer d'une complexité linéaire à une complexité constante.