TP C#7 : Basic Data Structure

Consignes de rendu

À la fin de ce TP, vous devrez rendre un repository git respectant l'architecture suivante :

```
|-- csharp-tp7-{login}/
    |-- README
    |-- TP7
        |-- TP7.sln
        |-- TP7/
            |-- List.cs
            |-- Node.cs
            |-- Queue.cs
            |-- Stack.cs
            |-- TestList.cs
            |-- TestQueue.cs
            |-- TestStack.cs
            |-- ConsoleMain.cs
            |-- Everything except bin/ and obj/
```

N'oubliez pas de vérifier les points suivants avant de rendre :

— Le fichier `README` est obligatoire.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— **Le code doit compiler !**

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les bonus que vous aurez implémenté. Un `README` vide sera considéré comme une archive invalide (malus). Le README est le meilleur moyen d'expliquer pourquoi et comment vous avez implémenté vos bonus.

1 Introduction

Ce tp a pour but de vous faire comprendre comment est implémenté un certain nombre de structures de données. N'hésitez donc pas à poser des questions aux ACDC et à regarder vos cours d'algo sur le sujet. De plus n'oubliez pas de tester votre code.

2 Cours

Cette partie sera très succincte car ce sont normalement des rappels.

2.1 Stack

Une **Stack** ou **Pile** en français est une structure FIFO (First In First Out). L'idée est de pouvoir insérer en temps constant et de pouvoir récupérer le premier élément en temps constant. Par exemple si j'empile 1, 2, 3, 4, 5, 6 dans cet ordre, j'obtiendrai 6, 5, 4, 3, 2, 1 en dépilant tous les éléments.

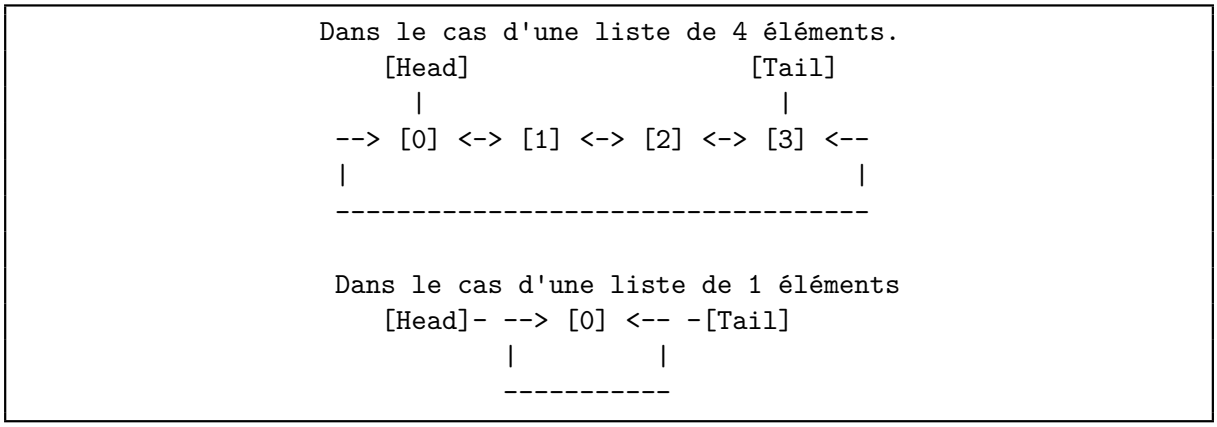
Bien entendu il existe de nombreuses implémentations possibles de **stack** comme les buffers circulaires, les stacks statiques et dynamiques. Dans notre cas on utilisera une implémentation dynamique.

2.2 Queue

Une **Queue** ou **File** en français est une structure LIFO (Last In First Out). L'idée est encore une fois d'avoir un temps d'insertion et de récupération constant. Il existe deux implémentations majeures, celle statique et celle dynamique. Dans le cadre de ce TP, nous ferons une implémentation dynamique.

2.3 Circular LinkedList

Cette structure de données est beaucoup plus complexe que les 2 premières. En effet pour implémenter une **Stack** ou une **Queue** on a seulement besoin de connaître la position du prochain noeud. Cependant dans le cas d'une **Circular LinkedList** nous devons avoir à la fois le suivant et le précédent. De plus la tête de liste et la fin de liste doivent être reliées.



Pour aller plus loin

`https://en.wikipedia.org/wiki/Circular_buffer`
`https://en.wikipedia.org/wiki/Stack_(abstract_data_type)`
`https://en.wikipedia.org/wiki/Queue_(abstract_data_type)`
`https://en.wikipedia.org/wiki/Doubly_linked_list`

Un super site qui permet de visualiser comment utiliser ces structures.
`https://visualgo.net/en/list`

3 Exercices

3.1 Node

Dans cette section vous implémenterez la brique élémentaire de vos structures. Ce module doit être le plus polyvalent possible, c'est pour cela qu'il n'est pas forcement adapté parfaitement aux **Stack** ou **Queue**. La difficulté d'implémentation est en ordre croissant, il est donc fortement conseillé de respecter l'ordre du TP.

3.1.1 Implémentation

Cette classe possède 2 attributs public `Next`, `Prev` qui pointe la `Node` suivante et la `Node` précédente. De plus le dernier attribut détient la donnée contenue par le noeud courant.

Le squelette de la classe est :

```
1 public class Node<T>
2 {
3     // TODO
4
5     public Node(T data)
6     {
7         // TODO
8     }
9 }
```

Vous pouvez constater que la classe `Node` est générique. Elle permettra donc de construire des `Stack`, `Queue` et `Circular LinkedList` génériques.

3.2 `Stack`

Vous n'avez pas le droit d'utiliser une autre classe que `Node` dans cette partie. On vous conseille de d'abord visualiser la pile avant de l'implémenter. En effet vous devez penser à tout les cas particuliers. La pile ne possède que 2 attributs. La taille de la pile qui doit être toujours à jour et l'attribut qui pointe sur la tête de la pile.

```
Dans le cas d'une pile vide.
    [Head | Size = 0]

Dans le cas d'une pile de 3 éléments
   [Head | Size = 3]
     |
    [2]
     ^
     |
    [1]
     ^
     |
    [0]
```

Bien entendu vous pouvez choisir d'utiliser l'attribut `Next` ou `Prev`. Il n'y a pas de bonne réponse. Faites ce qui vous semble le plus naturel. Cependant, quelque soit votre choix, la valeur initiale (à la construction de la Stack) est `Null`.

Le squelette de la classe est :

```
1  public class Stack<T>
2  {
3      private Node<T> _head; // Représente la valeur de tête de la pile.
4      private int _size; // La taille de la pile.
5
6      public Stack()
7      {
8          // TODO
9      }
10     public Stack(T[] data)
11     {
12         // TODO
13     }
14
15     public void Push(T data)
16     {
17         // TODO
18     }
19
20     public T Pop()
21     {
22         // TODO
23     }
24
25     public T Peek()
26     {
27         // TODO
28     }
29
30     public int Count()
31     {
32         // TODO
33     }
34 }
```

3.2.1 Constructeurs

Le premier constructeur est plutôt basique, il doit juste initialiser tous les champs à leur valeur par défaut. Le deuxième constructeur est beaucoup plus complexe (implémentez le en dernier). Il permet de créer à partir d'un tableau une `Stack`. Attention, un tableau vide est valide.

```
int[] array_tmp = new[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
Stack<int> stack = new Stack<int>(array_tmp);

stack.Count(); // 10

stack.Peek(); // 0

stack.Pop(); // 0
stack.Pop(); // 1
stack.Pop(); // 2
stack.Pop(); // 3
stack.Pop(); // 4
stack.Pop(); // 5
stack.Pop(); // 6
stack.Pop(); // 7
stack.Pop(); // 8
stack.Pop(); // 9
stack.Pop(); // ArgumentOutOfRangeException
```

3.2.2 `Peek`

Doit retourner la valeur de tête de la pile ou `ArgumentOutOfRangeException`.

3.2.3 `Push`

Doit ajouter un élément à la pile. Cette élément doit être relié à l'ancienne valeur de `_head`. N'oubliez pas de mettre à jour la taille de la pile.

3.2.4 `Pop`

Doit retourner `_head` ou doit déclencher l'exception `ArgumentOutOfRangeException` et doit remplacer la valeur de tête par le suivante.

3.3 `Queue`

Vous n'avez pas le droit d'utiliser une autre classe que `Node` dans cette partie. La `Queue` est un peu plus compliqué qu'une stack. En effet, vous devez ajouter un attribut, `_tail`.

Le squelette de la classe est :

```
1  public class Queue<T>
2  {
3      private Node<T> _head; // la tête de la file
4      private Node<T> _tail; // la fin de la file
5      private int _size;
6
7      public Queue()
8      {
9          // TODO
10     }
11     public Queue(T[] data)
12     {
13         // TODO
14     }
15
16     public void Push(T data)
17     {
18         // TODO
19     }
20
21     public T Pop()
22     {
23         // TODO
24     }
25
26     public T Peek()
27     {
28         // TODO
29     }
30     public int Count()
31     {
32         // TODO
33     }
34 }
```

3.3.1 Constructeurs

Le premier constructeur est plutôt basique, il doit juste initialiser tout les champs à leur valeur par défaut. Le deuxième constructeur est beaucoup plus complexe. Il permet de créer à partir d'un tableau une `Queue`. Attention, un tableau vide est valide.

```
int[] array_tmp = new[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
Queue<int> queue = new Queue<int>(array_tmp);

queue.Count(); // 10

queue.Peek(); // 9

queue.Pop(); // 9
queue.Pop(); // 8
queue.Pop(); // 7
queue.Pop(); // 6
queue.Pop(); // 5
queue.Pop(); // 4
queue.Pop(); // 3
queue.Pop(); // 2
queue.Pop(); // 1
queue.Pop(); // 0
queue.Pop(); // ArgumentOutOfRangeException
```

3.3.2 `Peek`

Doit retourner la valeur en fin de file ou `ArgumentOutOfRangeException`.

3.3.3 `Push`

Doit ajouter un élément à la file. Cette élément doit être relié à l'ancienne valeur de `_head`. N'oubliez pas de mettre à jour la taille de la pile et lorsque file est vide, vous devez aussi initialiser `_tail`.

3.3.4 `Pop`

Doit retourner `_tail` ou doit déclencher l'exception `ArgumentOutOfRangeException`. Et doit remplacer la dernière valeur de file par l'avant dernière.

3.3.5 Exemple

```
                        Dans le cas d'une file de 1 élément.
                                [Tail]
                                  |
                                 [0]
                                  |
                                [Head | Size = 1]

                        Dans le cas d'une file de 3 élément.
                                [Tail]
                                  |
                                 [0]
                                  ^
                                  |
                                 [1]
                                  ^
                                  |
                                 [2]
                                  |
                                [Head | Size = 3]
```

```
1  Queue<int> queue = new Queue<int>();
2
3  queue.Count(); // 0
4
5  queue.Push(0);
6  queue.Peek(); // 0
7  queue.Count(); // 1
8
9  queue.Push(1);
10 queue.Push(2);
11 queue.Push(3);
12 queue.Peek(); // 0
13 queue.Count(); // 4
14
15
16 queue.Pop(); // 0
17 queue.Count(); // 3
18
19 queue.Pop(); // 1
20 queue.Pop(); // 2
21 queue.Pop(); // 3
22 queue.Count(); // 0
23
24 queue.Pop(); // ArgumentOutOfRangeException
```

3.4 `Circular LinkedList`

Vous n'avez pas le droit d'utiliser une autre classe que `Node` dans cette partie. C'est dans cette partie que les tests sont primordiaux. Si vous ne faites pas de tests, vous ne pouvez pas

valider cet exercice. Cette structure ne demande pas plus d'attribut que précédemment. Vous allez juste devoir utiliser à la fois l'attribut `Next` et `Prev` de `Node`. Vous pouvez utiliser `https://visualgo.net/en/list` en cas d'incompréhension. Mais attention, ce site n'implémente qu'une **LinkedList** et pas une **Circular LinkedList**. Pensez à bien mettre à jour `_head` et `_tail`.

Le squelette de la classe est :

```
1  public class List<T>
2  {
3      private Node<T> _head;
4      private Node<T> _tail;
5      private int _size;
6
7      public List() {} // TODO
8
9      public List(T[] data) {} // TODO
10
11     public void AddFirst(T data) {} // TODO
12
13     public void AddLast(T data) {} // TODO
14
15     public void Add(int index, T data) {} // TODO
16
17     public void RemoveFirst() {} // TODO
18
19     public void RemoveLast() {} // TODO
20
21     public void Remove(int index) {} // TODO
22
23     public T Get(int index) {} // TODO
24
25     public int Count() {} // TODO
26 }
```

3.4.1 Constructeurs

Le premier constructeur est plutôt basique, il doit juste initialiser tous les champs à leur valeur par défaut. Le deuxième constructeur doit créer une liste à partir d'un tableau.

3.4.2 `AddFirst`

Ajoute en tête un élément à la liste.

```
1  List<int> list = new List<int>();
2
3  list.AddFirst(0); // [0]
4  list.AddFirst(1); // [1, 0]
5  list.AddFirst(2); // [2, 1, 0]
6  list.AddFirst(3); // [3, 2, 1, 0]
7  list.Count(); // 4
```

3.4.3 AddLast

Ajoute en fin de liste un élément.

```
List<int> list = new List<int>();

list.AddLast(0); // [0]
list.AddLast(1); // [0, 1]
list.AddLast(2); // [0, 1, 2]
list.AddLast(3); // [0, 1, 2, 3]
list.Count(); // 4
```

3.4.4 Add

Ajoute à l'`index` l'élément dans la list. Si on ne peut pas, vous devez déclencher l'exception `ArgumentOutOfRangeException`. Attention, un index égal à la taille est valide.

```
List<int> list = new List<int>();

list.Add(0, 0); // [0]
list.Add(1, 1); // [0, 1]
list.Add(1, 2); // [0, 2, 1]
list.Add(0, 3); // [3, 0, 1, 2]
list.Count(); // 4

list.Add(1221121212, 42); // ArgumentOutOfRangeException
```

3.4.5 RemoveFirst

Supprime le premier élément de la liste. Si ce n'est pas possible, déclenchez l'exception `ArgumentOutOfRangeException`.

```
int[] array_tmp = new[] {0, 1, 2};
List<int> list = new List<int>(array_tmp);

list.RemoveFirst(); // [1, 2]
list.RemoveFirst(); // [2]
list.RemoveFirst(); // []

list.RemoveFirst() // ArgumentOutOfRangeException
```

3.4.6 RemoveLast

Supprime le dernier élément de la liste. Si ce n'est pas possible, déclenchez l'exception `ArgumentOutOfRangeException`.

```
int[] array_tmp = new[] {0, 1, 2};
List<int> list = new List<int>(array_tmp);

list.RemoveLast(); // [0, 1]
list.RemoveLast(); // [1]
list.RemoveLast(); // []

list.RemoveLast() // ArgumentOutOfRangeException
```

3.4.7 `Remove`

Supprime l'élément désigné par l'index. Si ce n'est pas possible, déclenchez l'exception `ArgumentOutOfRangeException`.

```
int[] array_tmp = new[] {0, 1, 2, 3};
List<int> list = new List<int>(array_tmp);

list.Remove(1); // [0, 2, 3]
list.Remove(0); // [1, 2]
list.Remove(0); // [2]

list.Remove(1) // ArgumentOutOfRangeException
```

3.4.8 `Get`

Renvoie le contenu de la `Node` désignée par l'index. Si ce n'est pas possible, déclenchez l'exception `ArgumentOutOfRangeException`.

```
int[] array_tmp = new[] {0, 1, 2};
List<int> list = new List<int>(array_tmp);

list.Get(0); // 0
list.Get(1); // 1
list.Get(2); // 2

list.Get(3) // ArgumentOutOfRangeException
```

3.5 Test

Si vous avez fait attention, nous ne demandons pas la suppression des tests. En effet, pour chaque structure de donné , les tests correspondant seront notés.

4 Bonus

4.1 Nunit

Utilisez la bibliothèque NUnit pour tester vos fonctions.

4.2 Upgrade

Améliorez `List` en implémentant les méthodes de votre choix. Vous devrez justifier votre choix dans le README.

If you knew time as well as I do, You wouldn't talk about wasting it