1 Module A.5 : Collections

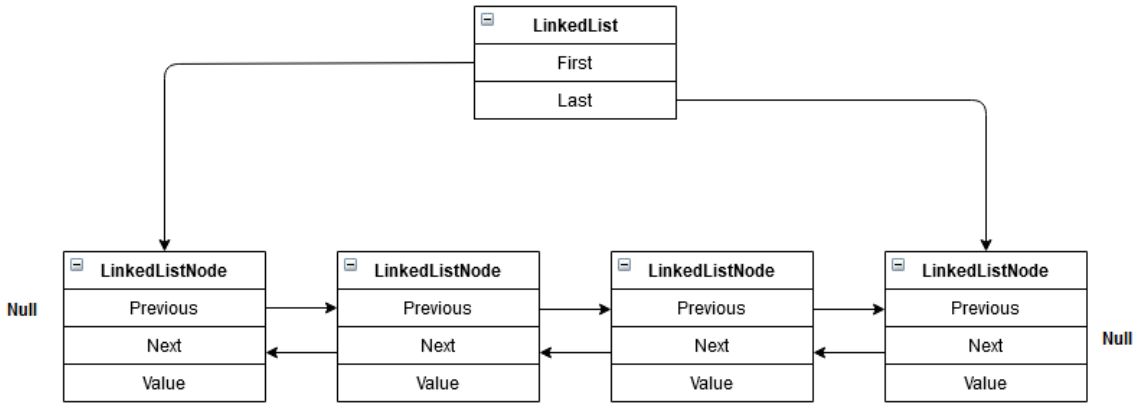
Les modules précédents ont étudié deux types de collections à savoir les tableaux de données de taille fixe et dynamique (**List<T>**). Ces derniers permettent de faire beaucoup de choses. L'utilisation des tableaux n'est pas toujours le choix le plus adapté pour représenter les données. La représentation des données est aussi fondamentale que les implémentations des processus fonctionnelles. Une structure de données adaptée simplifie grandement la manière de résoudre le problème traité. En C#, les bibliothèques **Collections** et **Collections.Generic** fournissent tout une série de structures de données utilisables dans les applications .NET. On peut classer ces collections en deux grandes familles à savoir les listes et les dictionnaires. **n** est le nombre d'éléments.

1.1 Les Listes

Les listes constituent la famille des structures de données où ses éléments sont accessibles. Il existe toute une série de listes en C# que nous allons présenter , leurs principales méthodes et leurs avantages et inconvénients. **List<T>** fait partie des listes. Étant donné qu'elle a déjà fait l'objet d'une présentation dans le module A.3, cette section ne l'aborde pas et se focalise sur les autres types de listes.

1.1.1 LinkedList<T>

La **LinkedList<T>** est une liste doublement chaînée. Une liste doublement chaînée est une liste de nœuds où chacun d'entre eux est relié à son prédécesseur et à son successeur. Chacun de ses nœuds dispose d'une valeur de type **T** défini lors de la déclaration de l'instance de la structure de données.



Le tableau suivant fournit une série de méthodes accessibles permettant d'interagir avec une **LinkedList<T>**.

Méthodes	Explications
void AddFirst (**LinkedListNode<T>** node)	Ajout d'un nœud au début.
LinkedListNode<T> AddFirst (**T** value)	Ajout d'un nœud au début avec la valeur **T** et retourne ce nœud.
void AddLast (**LinkedListNode<T>** node)	Ajout d'un nœud à la fin.
LinkedListNode<T> AddLast (**T** value)	Ajout d'un nœud à la fin avec la valeur **T** et retourne ce nœud.
void AddAfter (**LinkedListNode<T>** node, **LinkedListNode<T>** newNode)	Ajout d'un nouveau nœud après le nœud en paramètre.
LinkedListNode<T> AddAfter (**LinkedListNode<T>** node, **T** value)	Ajout d'un nouveau nœud après le nœud en paramètre avec la valeur **T** et retourne ce nœud.
void AddBefore (**LinkedListNode<T>** node, **LinkedListNode<T>** newNode)	Ajout d'un nouveau nœud avant le nœud en paramètre.
LinkedListNode<T> AddBefore (**LinkedListNode<T>** node, **T** value)	Ajout d'un nouveau nœud avant le nœud en paramètre avec la valeur **T** et retourne ce nœud.
void Clear()	Suppression de tous les nœuds.
void RemoveFirst()	Suppression du premier nœud.
void RemoveLast()	Suppression du dernier nœud.
bool Remove (**T** value)	Suppression de l'élément de valeur donnée, renvoie false si l'élément n'existe pas.
void Remove (**LinkedListNode<T>** node)	Suppression du nœud indiqué.
bool Contains (**T** value)	La **LinkedList** contient-elle la valeur en paramètre ?
LinkedListNode<T> Find (**T** value)	Retourne le premier nœud en partant de **First** dont la valeur correspond à celle passée en paramètre.
LinkedListNode<T> FindLast (**T** value)	Retourne le premier nœud en partant de **Last** dont la valeur correspond à celle passée en paramètre.

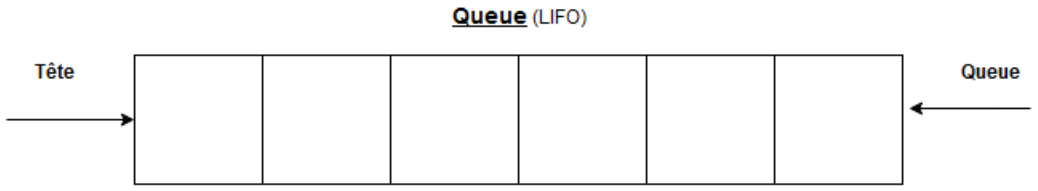
LinkedList<T> ne permet pas de référencer directement les éléments qui la constitue. Les seuls éléments accessibles directement sont **First** et **Last**. L'intérêt principal de cette structure de données est le fait que l'insertion, la suppression des nœuds est efficace (O(1) contrairement aux **List<T>** avec O(n)). La recherche d'éléments est peu efficace (O(n)) mais on peut utiliser cette structure dans une boucle **foreach**.

Exemple :

```
1  var notes = new LinkedList<string>();
2  notes.AddFirst ("do");                    // do.
3  notes.AddLast ("so");                     // do - so.
4
5  notes.AddAfter (notes.First, "re");       // do - re - so.
6  notes.AddAfter (notes.First.Next, "mi"); // do - re - mi - so.
7  notes.AddBefore (notes.Last, "fa");       // do - re - mi - fa - so.
8
9  notes.RemoveFirst();                      // re - mi - fa - so.
10 notes.RemoveLast();                       // re - mi - fa.
11
12 LinkedListNode<string> miNode = notes.Fin ("mi");
13 notes.Remove (miNode);                    // re - fa.
14 notes.AddFirst (miNode);                  // mi - re - fa.
15
16 foreach (string s in notes) Console.WriteLine (s);
```

1.1.2 Queue<T>

Queue<T> est une structure de données de type **LIFO** (Last-In, First-Out). Il permet de traiter les données dans un contexte de file d'attente. Si des traitements doivent être réalisés selon l'ordre d'arrivée, on peut utiliser une **Queue<T>**. Le premier élément de la liste sera sorti en premier. Elle est gérée en interne avec des tableaux et des index pour représenter la queue et la tête. Il existe des queues avec des règles de priorité (l'élément sorti est l'élément de valeur maximale, minimale...) qui sont beaucoup utilisées dans nombre d'algorithmes comme l'algorithme de Dijkstra, mais n'ont pas d'implémentations dans les collections standards (voir exercices).



Le tableau suivant fournit une série de méthodes accessibles permettant d'interagir avec une instance de type **Queue<T>**.

Méthodes	Explications
void Clear ()	Supprime tous les éléments.
bool Contains (**T** item)	La queue contient-elle l'élément en paramètre ?
int Count ()	Le nombre d'éléments de la queue.
T Dequeue ()	Suppression du premier élément et le retourne.
void Enqueue (**T** item)	Ajout d'un élément.
T Peek ()	Retour du premier élément sans suppression.

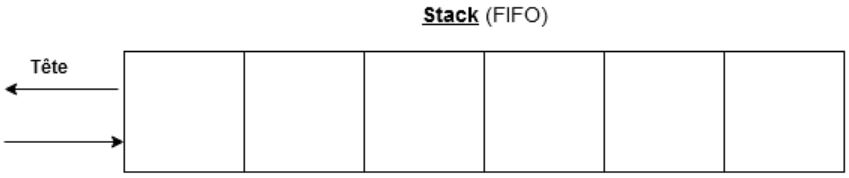
Les opérations d'ajout et de suppression d'éléments sont rapides (O(1)) mais il n'y a pas d'accès direct aux éléments de la queue.

Exemple :

```
1  var q = new Queue<int>();
2  q.Enqueue (10);                 // Queue : 10.
3  q.Enqueue (20);                 // Queue : 10 - 20.
4  Console.WriteLine (q.Count);    // 2.
5  Console.WriteLine (q.Peek());   // 10.
6  Console.WriteLine (q.Dequeue); // 10.
7  Console.WriteLine (q.Dequeue); // 20.
8  Console.WriteLine (q.Dequeue); // Exception levée (queue vide).
```

1.1.3 Stack<T>

Stack<T> est une structure de données de type **FIFO** (First-In, First-Out). Il permet de traiter les données dans un contexte de pile. Un exemple bien connu est celui des tours de Hanoï. **Stack<T>** est gérée d'une façon semblable à la structure **Queue<T>**. Le dernier élément est sorti en premier.



Le tableau suivant fournit une série de méthodes accessibles permettant d'interagir avec une instance de type **Stack<T>**.

Méthodes	Explications
void Clear ()	Supprime tous les éléments.
bool Contains (**T** item)	La pile contient-elle l'élément en paramètre ?
int Count ()	Le nombre d'éléments de la pile.
T Pop ()	Suppression du dernier élément et le retourne.
void Push (**T** item)	Ajout d'un élément.
T Peek ()	Retour du dernier élément sans suppression.

Les opérations d'ajout et de suppression d'éléments sont rapides (O(1)) mais il n'y a pas d'accès direct aux éléments de la pile.

Exemple :

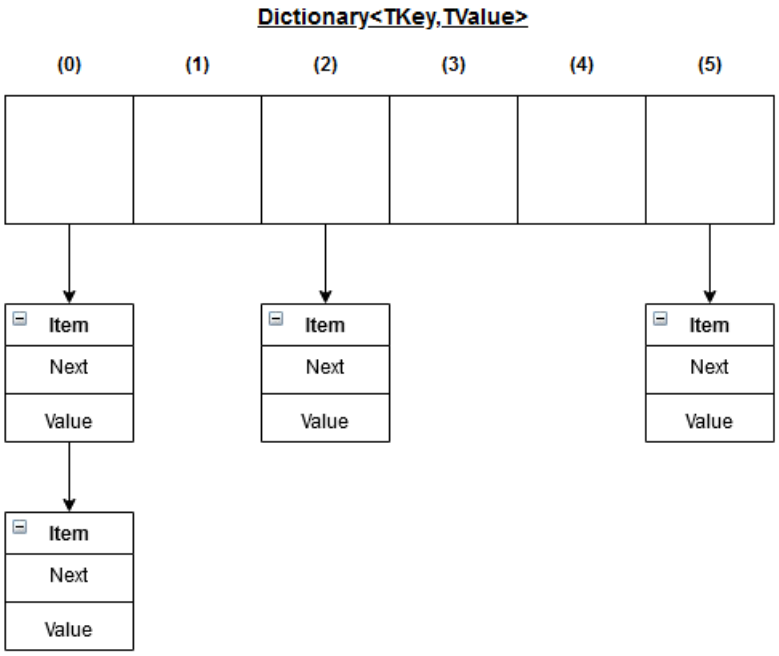
```
1  var s = new Stack<int>();
2  s.Push (1);                    // Pile : 1.
3  s.Push (2);                    // Pile : 1 - 2.
4  s.Push (3);                    // Pile : 1 - 2 - 3.
5  Console.WriteLine (s.Count);  // 3.
6  Console.WriteLine (s.Peek()); // 3.
7  Console.WriteLine (s.Pop());  // 3.
8  Console.WriteLine (s.Pop());  // 2.
9  Console.WriteLine (s.Pop());  // 1.
10 Console.WriteLine (s.Pop());  // Exception levée (pile vide).
```

1.2 Les Dictionnaires

Les dictionnaires constituent une série de structures de données où l'on ne stocke pas seulement une valeur, mais un couple d'une clef plus une valeur. Ce type de structure de données est important si on cherche à distinguer les éléments les uns des autres. Par exemple, un dictionnaire sera approprié pour la représentation d'un annuaire téléphonique. La clef représente le nom du contact et la valeur le numéro de téléphone.
Il existe plusieurs sortes de dictionnaires qui se distinguent par un certain nombre de caractéristiques. Est-ce que le stockage des éléments est trié ? Est-ce que l'accès aux éléments se fait à l'aide de la clef ou/et avec un indice ? On ne présente que la structure de données la plus générale classique.

1.2.1 Dictionary<TKey,TValue>

Dictionary<TKey,TValue> est le dictionnaire le plus utilisé. Contrairement aux différentes listes, il suppose que l'on définisse le type de la valeur et le type de la clef. Il existe une contrainte sur le type de la clef, elle doit permettre les tests d'égalité. Cela est dû au fait que les clefs ont une contrainte d'unicité.
Dictionary<TKey,TValue> est représentée en interne comme une table de hachage. Chaque clef est convertie en un code de hachage. Ce code permet de choisir la cellule dans laquelle va être stockée la valeur. Chaque cellule peut être représentée comme une liste simplement chaînée (**LinkedList<T>** est doublement chaînée, car chaque nœud possède une référence à son prédécesseur et à son successeur, ici seulement à son successeur). Si le code de hachage a déjà été utilisé, il y a ajout d'un nouveau nœud à la fin de la chaîne. La structure est optimisée pour que le nombre d'éléments reste proche de 1, celle-ci étant redimensionnée lorsque cette condition n'est pas respectée.



Le tableau suivant fournit une série de méthodes accessibles permettant d'interagir avec un **Dictionary<TKey,TValue>**.

Méthodes	Explications
bool ContainsKey (**TKey** key)	Test sur l'existence de la clef en paramètre.
bool ContainsValue (**TValue** value)	Test sur l'existence de la valeur en paramètre.
bool TryGetValue (**TKey** key, out **TValue** value)	Tentative de récupérer la valeur associée à la clef dans le paramètre de sortie. Retourne si l'opération a réussi ou échoué.
void Add (**TKey** key, **TValue** value)	Ajout de la valeur associée à une clef. Si la clef existe déjà, une exception est levée.
void Remove (**TKey** key)	Suppression de la valeur associée à la clef spécifiée.

Dictionary<TKey,TValue> permet l'accès aux différents éléments à l'aide de sa clef (même syntaxe que pour les tableaux où l'on remplace l'indice par la clef). La liste complète des valeurs et celle des clefs est disponibles à l'aide des propriétés **Values** et **Keys**. La recherche par clef, l'ajout et la suppression de données sont efficaces (O(1) dans le cas amorti). La recherche par valeur est par contre lente (il faut vérifier le dictionnaire donc O(n)).

Exemple :

```
1  var d = new Dictionary<string,int>();
2  d.Add("One", 1);
3  d["Two"] = 2;    // Ajout de la valeur car la clef n'est pas présente.
4  d["Two"] = 22;   // Mise à jour de la valeur car la clef est déjà présente.
5  d["Three"] = 3;
6
7  Console.WriteLine (d["Two"]); // 22.
8  Console.WriteLine (d.ContainsKey ("One")); // True.
9  Console.WriteLine (d.ContainsValue (3));   // True.
10
11 int val = 0;
12 if (!d.TryGetValue ("onE", out val))
13   Console.WriteLine ("Not val");           // Not val.
14
15 foreach (KeyValuePair<string, int> kv in d)     // One ; 1.
16   Console.WriteLine (kv.Key + ";" + kv.Value);  // Two ; 22.
17                                                 // Three ; 3.
18
19 foreach (string s in d.Keys) Console.Write (s); // OneTwoThree.
20 Console.WriteLine ();
21 foreach (int i in d.Values) Console.Write (i);   // 1223.
```