Chapitre 2 : Listes

2.1 Définition et terminologie

Une liste est une suite ordonnée et finie d'éléments d'un type donné, non nécessairement distincts mais repérés par leur rang. Exemple d'une liste d'entiers : (6, 2, 7, 14, 2, 3, 9).

La **longueur** d'une liste correspond au nombre d'éléments composant la liste (Une liste de longueur zéro est une liste vide).

Une liste non vide comprend un premier élément appelé **tête**. Le reste de la liste est appelé **queue**. À chaque élément d'une liste est associée une position ou **rang** (un entier).

Un **préfixe** est une **sous-liste** qui commence au début de la liste. Un **suffixe** est une **sous-liste** qui se termine à la fin de la liste.

> ☺ **Exercice 2.A :** on considère la liste d'entiers suivante : (4, 7, 9, 9, 3, 7, 8, 12). Indiquer la tête et la queue de la liste, le rang de la première occurrence de 9, une sous-liste de longueur 3, le préfixe de longueur 4 et le suffixe de longueur 6.

2.2 Spécification algébrique

```
Type MyList[E]

Opérations

     empty : -> MyList[E]                          -- créer une liste vide
     addFirst : MyList[E] X E -> MyList[E]         -- ajouter un élt en tête
     addLast : MyList[E] X E -> MyList[E]          -- ajouter un élt en queue
     add : MyList[E] X E X Integer -> MyList[E]    -- ajouter un élt à un
                                                      rang donné
     add : MyList[E] X MyList[E] -> MyList[E]      -- concaténer deux
                                                      listes
     remove : MyList[E] X E -> MyList[E]           -- supprimer la 1ère
                                                      occur. d'un élt
     remove : MyList[E] X Integer -> MyList[E]     -- supprimer l'élt de
                                                      rang donné
     set : MyList[E] X Integer X E -> MyList[E]    -- modifier l'élément de
                                                      rang donné
     first : MyList[E] -> E                        -- premier élément
     queue : MyList[E] -> MyList[E]                -- liste privée de sa tête
     get : MyList[E] X Integer -> E                -- élément de rang i
     contains : MyList[E] X E -> Boolean           -- vrai si l'élément est
                                                      dans la liste
     size : MyList[E] -> Integer                   -- nombre d'éléments
     isEmpty : MyList[E] -> Boolean                -- vrai lorsque liste vide
     indexOf : MyList[E] X E -> Integer            -- rang d'un élément
```

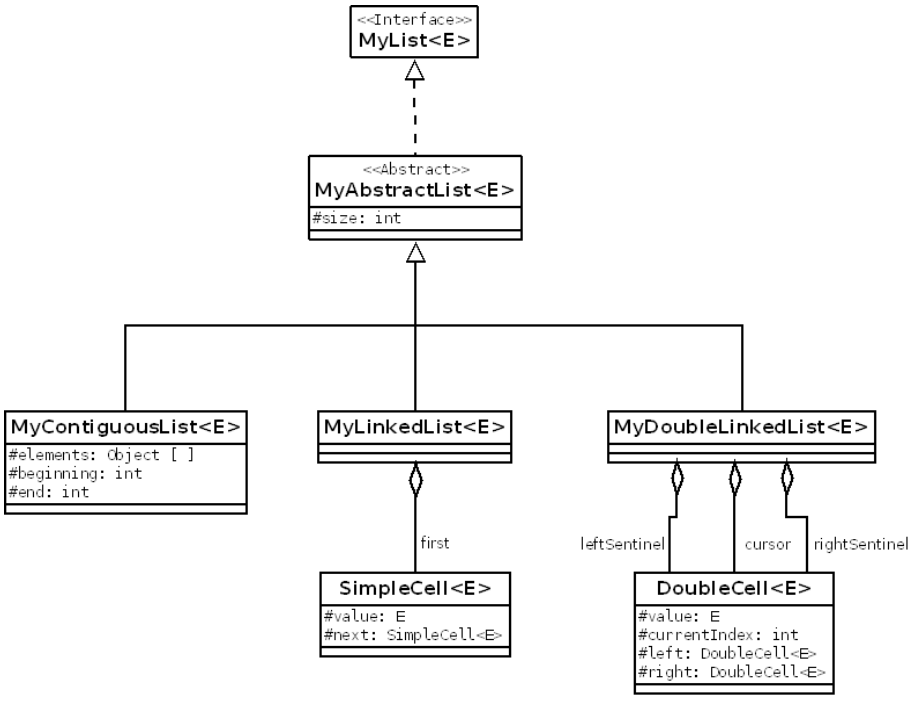
☺ **Exercice 2.B :** identifier les opérations internes et les observateurs du type MyList[E], puis compléter la spécification algébrique du type MyList[E] en indiquant les préconditions et les axiomes correspondants. On se limitera à l'écriture d'un sous-ensemble d'axiomes : application de l'observateur first sur toutes les opérations internes, puis application des autres observateurs limitée aux opérations internes empty et addFirst.

2.3 Compléments relatifs à l'écriture des tests

Le test de la spécification algébrique s'appuie sur l'hypothèse que les préconditions sont satisfaites. Si on souhaite que le programme les vérifie, il faut adopter un style de programmation défensive, qui oblige chaque méthode à se protéger elle-même des mauvais usages. Ainsi, dans chaque implantation, le corps de la fonction `public void add(E el, int i)` commencera par la séquence : `assert (i >= 0 && i <= size()) : "Violation de la précondition de add"` qui déclenche une exception si la précondition n'est pas vérifiée.

2.4 Implantations de MyList[E]

L'interface `MyList<E>`, obtenue par dérivation du type abstrait, fournira les profils des différentes méthodes. Dans la classe abstraite `MyAbstractList<E>`, il faut écrire complètement toutes les méthodes qui sont indépendantes d'une implantation particulière (contiguë ou chaînée). On y placera un attribut `size` pour mémoriser la taille de la liste (ce qui évitera de la recalculer à chaque appel de la méthode `size`, mais ce qui impliquera la mise à jour de cet attribut pour les méthodes d'ajout et de suppression). Définir, dans cette classe, les méthodes `isEmpty`, `size`, `first`, `remove`, `add` (lorsque l'argument est une liste), `addFirst`, `addLast`, et `contains` paraît évident. Toutes les autres méthodes nécessitent un parcours de la liste. Au premier abord, on voit mal comment factoriser ce parcours, et l'écrire indépendamment de l'implantation.

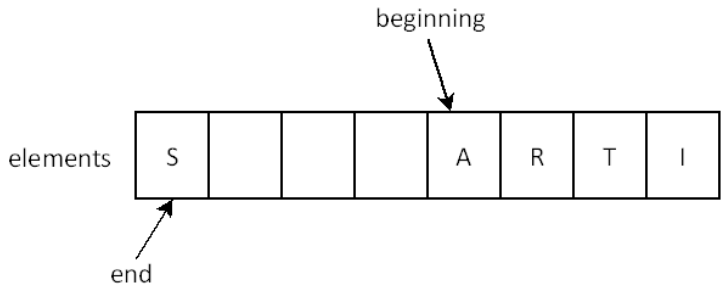


☺ **Exercice 2.C :** en vous aidant de l'interface `MyList<E>` décrite ci-dessous, écrire les méthodes `addFirst`, `addLast`, `add(MyList<E> l)`, `size`, `isEmpty` et `first` de la classe abstraite `MyAbstractList<E>`.

```
public interface MyList <E>  {

    public void addFirst (E el);
    public void addLast (E el);
    public void add (E el, int rang);
    public void add (MyList<E> l);
    public void remove (E el);
    public void remove (int rang);
    public void set (int rang, E el);
    public E first ();
    public void queue ();
    public E get (int rang);
    public boolean contains (E el);
    public int size ();
    public boolean isEmpty ();
    public int indexOf (E el);

}
```

2.4.1 Implantation contiguë

Dans une implantation contiguë, les éléments de la liste sont rangés les uns à côté des autres dans un tableau (attribut `elements`). Une solution simple consiste à placer l'élément de rang 0 à la position 0 dans le tableau, l'élément de rang 1 à la position 1, etc. Afin de réduire le nombre de décalages lors d'un ajout ou de la suppression d'un élément, il est possible de mettre en œuvre une gestion circulaire en mémorisant le début (attribut `beginning`) et la fin (attribut `end`) de la liste.



☺ **Exercice 2.D :** compléter la classe `MyContiguousList<E>` ci-dessous en implantant la méthode `position` qui retourne la position dans le tableau d'un élément de rang donné en argument. Nous considérons une gestion circulaire des éléments du tableau. Vous noterez que la méthode `add` distingue les quatre cas rencontrés lors de l'ajout d'un élément (décalage vers la gauche sans et avec dépassement du tableau, et décalage vers la droite avec ou sans dépassement). Nous supposons pour l'exercice que la capacité du tableau est toujours suffisante par rapport au nombre d'éléments ajoutés.

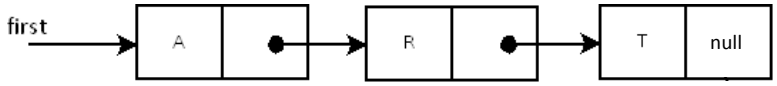
```
public class MyContiguousList<E> extends MyAbtractList<E> {

        protected Object [] elements;  // Gestion circulaire
        protected int beginning;
        protected int end;

        public MyContiguousList(int capacity) {
            elements = new Object[capacity];
            beginning = capacity / 2;
            end = beginning - 1;
        }

        public MyContiguousList() { this(100);}

        public int position(int rang) { return ................. };

        public void add(E el, int rang) {

          // Mise à jour de l'attribut size (hérité de MyAbstractList)
          size++ ;

          // Choix du sens du décalage, selon le rang
            int middle = (beginning + end) / 2 ;

            if (rang < middle) {
                // Décalage vers la gauche
                if (beginning == 0) { // Dépassement du tableau
                    elements[elements.length - 1] = elements[0];
                    for (int k = 0; k < (position(rang)-1); k++) {
                        elements[k] = elements[k+1];}
                    beginning = elements.length - 1;
                } else {
                    for (int k=beginning-1; k<(position(rang)-1); k++) {
                        elements[k] = elements[k + 1];}
                    beginning-- ;
                }}
               else {
                // Décalage vers la droite
                if (end == elements.length - 1 ) { // Dépassement
                   elements[0] = elements[elements.length - 1];
                   for(int k=elements.length-1;k>(position(rang)+1);k--){
                        elements[k] = elements[k - 1];
                    }
                   end = 0 ;}
                   else {
                    for (int k = end+1; k>(position(rang)+1); k--) {
                        elements[k] = elements[k - 1];
                    }
                    end++;
                }
            }
            elements[position(rang)] = el ;
        }

        public E get(int rang) {
            return (E) elements[position(rang)];
        }
}
```

2.4.2 Implantation chaînée

Une liste chaînée est constituée de « maillons » ou « cellules » ; les cellules peuvent contenir des données et l'adresse d'autres cellules. Les données contenues par les cellules sont du même type d'une cellule à l'autre.

2.4.2.1 Liste chaînée simple

Dans le cas d'une liste chaînée simple, une cellule pointe (uniquement) sur la cellule suivante de la liste. Une liste chaînée composée des caractères A, R, T pourra être représentée très sommairement comme sur le schéma ci-dessous :



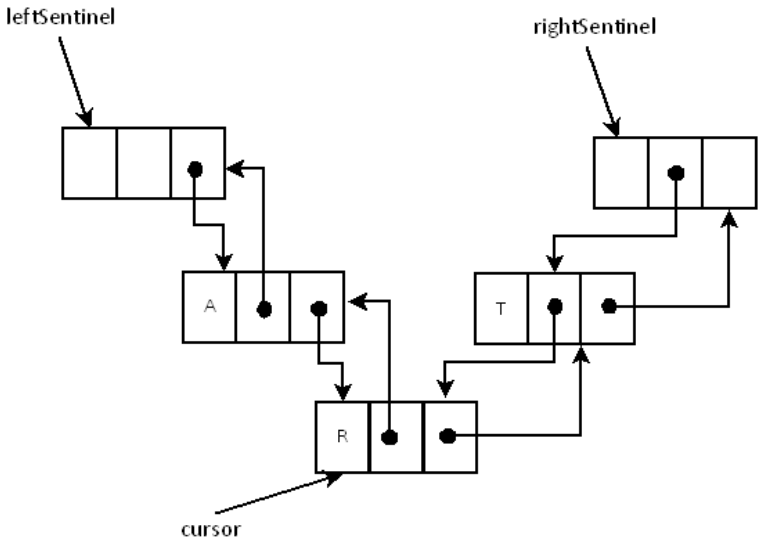
☺ **Exercice 2.D :** soit la classe `SimpleCell<E>` représentant les cellules d'une liste chaînée simple, complétez la classe `MyLinkedList<E>` en implantant les méthodes d'ajout et de suppression d'un élément. Vous pourrez vous aider de schémas mémoires.

```
public class SimpleCell<E> {

    protected E value;
    protected SimpleCell<E> next;

    public SimpleCell(SimpleCell<E> next, E value) {
        this.value = value; this.next = next ; }

    public E getValue(){return value;}
    public void setValue(E value) {this.value = value ;}

    public SimpleCell<E> getNext(){return next;}
    public void setNext(SimpleCell<E> next) {this.next = next ;}
}
```

```
public class MyLinkedList<E> extends MyAbstractList<E> {

    protected SimpleCell<E> first;

    public MyLinkedList() {
        size = 0;
    }

    public void add(E el, int rang) {
        ..................
    }

    public void remove(int rang) {
        ..................
    }
}
```

2.4.2.2 Liste doublement chaînée

Une liste chaînée simple ne permet un parcours que dans un sens (de la gauche vers la droite) ; le coût lié à la recherche d'un élément peut être réduit en autorisant un parcours dans les deux sens. Ainsi, dans le cas d'une liste doublement chaînée, une cellule indique à la fois l'adresse de la cellule précédente et l'adresse de la cellule suivante. Pour faciliter la manipulation de la liste, nous introduisons des sentinelles aux extrémités de la liste.



☺ **Exercice 2.E :** soit la classe `DoubleCell<E>` représentant les cellules d'une liste doublement chaînée, complétez la classe `MyDoubleLinkedList<E>` en implantant les méthodes d'ajout et de suppression d'un élément. Vous introduirez une méthode privée `positionCursor` qui permet de placer le curseur au rang passé en argument. Cette méthode permettra de minimiser le nombre d'opérations pour effectuer le déplacement entre la position courante et la position souhaitée (le déplacement pouvant s'effectuer depuis la sentinelle gauche, la sentinelle droite ou le curseur).

```
public class DoubleCell<E> {

    protected E value;
    protected DoubleCell<E> left;
    protected DoubleCell<E> right;

    public DoubleCell(DoubleCell<E> left, DoubleCell<E> right, E value) {
        this.left = left; this.right = right; this.valeur = valeur;}

    public DoubleCell<E> getLeft() { return left;}
    public void setLeft(DoubleCell<E> left) { this.left = left;}

    public DoubleCell<E> getRight() {return right;}
    public void setRight(DoubleCell<T> right) {this.right = right;}

    public E getValue() {return value;}
    public void setValue(E value) {this.value = value ;
```

```
public class MyDoubleLinkedList<E> extends MyAbstractList<E> {

    protected DoubleCell<E> leftSentinel;
    protected DoubleCell<E> rightSentinel;
    protected DoubleCell<E> cursor;
    protected int currentIndex; // rang du curseur

    public MyDoubleLinkedList() {
        size = 0;
        leftSentinel = new DoubleCell<E>();
        rightSentinel = new DoubleCell<E>();
        leftSentinel.setRight(rightSentinel);
        rightSentinel.setLeft(leftSentinel);
        cursor = leftSentinel;
        currentIndex = -1;
    }

    public void add(E el, int index) {
      ..........................
    }

    public void remove(int index) {
      ..........................
    }

    // Positionner le curseur pour qu'il désigne l'élément de rang index
    private void positionCursor(int index) {

        // Nous sommes déjà à la bonne position
        if (currentIndex == index) {
            return;
        }

        // Nous souhaitons aller à la première place
        if (index == 0) {
            currentIndex = 0;
            cursor = leftSentinel.getRight();
            return;
        }

        // Nous souhaitons aller après la dernière place
        ........................

        // Nous souhaitons aller dans la moitié gauche, à gauche du curseur
        ........................

        // Nous souhaitons aller dans la moitié droite, à gauche du curseur
        ........................

        // Nous souhaitons aller dans la moitié gauche à droite du curseur
        ........................

        // Nous souhaitons aller dans la moitié droite à droite du curseur
        ........................

    }
}
```

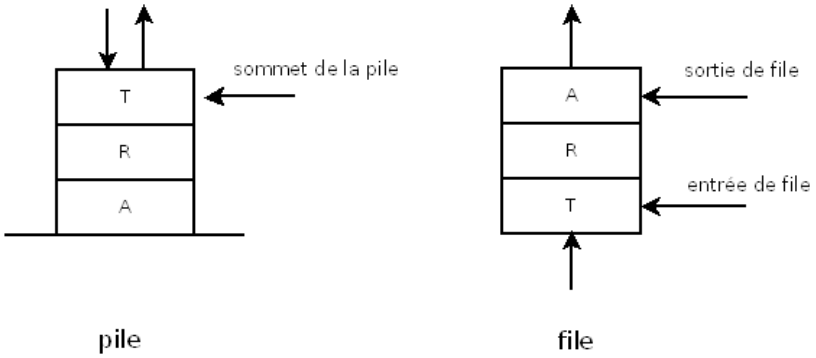
2.5 Performances comparées des implantations

L'implantation contiguë offre un accès facile au ième élément, mais les opérations d'ajout et de suppression sont coûteuses à cause des décalages (même dans le cas d'une gestion circulaire). Elle pose également des problèmes d'allocation mémoire lorsque la taille du tableau varie.

Les implantations chaînées n'ont pas de longueur fixée mais sont plus coûteuse en mémoire (cellules avec pointeur sur le suivant, voire sur le précédent). L'insertion et la suppression sont rapides, mais il n'y a pas d'accès direct au ième élément.

> ☺ **Exercice 2.F :** évaluer et comparer la complexité algorithmique dans le pire des cas pour l'ajout/suppression en queue, l'ajout/suppression en tête, l'ajout/suppression d'un élément de rang i, et l'accès à l'élément de rang i, pour les trois implantations.

2.6 Rappels sur les listes à accès restreint



2.6.1 Pile

Une pile est une liste à accès restreint où les éléments sont ajoutés et supprimés à une même extrémité (le sommet). Il s'agit d'une approche « dernier entré, premier sorti » ou LIFO (Last In, First Out).

Les restrictions portent sur :

- l'accès (méthode peek ou sommet) qui se fait uniquement sur l'élément placé au sommet de la pile,
- l'insertion (méthode push ou empiler) qui ne peut se faire qu'au sommet d'une pile non pleine,
- la suppression (méthode pop ou dépiler) qui porte sur le sommet d'une pile non vide.

2.6.2 File

Une file est une liste à accès restreint où les éléments ne peuvent être ajoutés qu'à une extrémité (la queue) et ne peuvent être supprimés qu'à l'autre (la tête). Il s'agit d'une structure « premier entré, premier sorti » ou FIFO (First In, First Out).

Les restrictions portent donc sur :

- l'accès (méthode head ou tête) qui se fait uniquement sur l'élément en tête de la file,
- l'insertion (méthode add ou ajouter) qui se fait uniquement en queue de la file,
- la suppression (méthode remove ou retirer) qui se fait uniquement en tête de file.