**REPONSE**

1. Liste des classes et fonctions étudiées au cours de l’examen des structures des données.

* CircularQueue.py
* LinkedDeque.py
* \_DoublyLinked\_Base.py
* ArrayStack.py
* ArrayQueue.py
* LinkedStack.py
* LinkedQueue.py
* is\_matched\_html.py
* is\_matched.py
* reverse\_file.py
* DynamicArray.py
  1. **ArrayStack**

Nous utilisons le pattern Adapter pour définir une classe **ArrayStack** qui utilise une Liste Python pour le stockage des éléments. Nous avons choisi le nom **ArrayStack** pour souligner que le stockage sous-jacent est intrinsèquement basé sur un tableau. Une question qui demeure est ce que notre code doit faire si un utilisateur appelle **pop()** ou **top()** lorsque la pile est vide. Notre ADT Stack suggère qu’une erreur se produise, mais nous devons décider quel type d’erreur. Lorsque pop() est appelé sur une liste Python vide, il se produit formellement une IndexError, car les listes sont des séquences basées sur des indices. Ce choix ne semble pas approprié pour un stack, puisque il n’y a pas d’hypothèse d’indices. Au lieu de cela, nous pouvons définir une nouvelle classe d’exception qui est plus approprié.

* 1. **Reverse\_file**

En conséquence du fait qu’elle utilise le protocole LIFO, une pile peut-être utilisée comme outil général pour inverser une séquence de données. Par exemple, si les valeurs 1, 2 et 3 sont poussées dans une pile dans cet ordre, elles seront extraites de la pile dans l’ordre 3, 2, puis 1. Avec cette fonction, nous pourrions imprimer les lignes d’un fichier dans l’ordre inverse. Par exemple, afin d’afficher un ensemble de données en ordre décroissant plutôt que dans l’ordre croissant.

* 1. **is\_matched**

Nous supposons que l’entrée est une séquence de caractères, telle que [(5+x)- (y+z)] . Nous effectuons un balayage de gauche à droite de la séquence originale, en utilisant une pile S pour faciliter la correspondance entre les symboles d’ouverture et de fermeture de regroupement. Chaque fois que nous rencontrons un symbole d’ouverture, nous poussons ce symbole dans S, et chaque fois que nous rencontrons un symbole de fermeture, nous faisons expulser un symbole de la pile S (en supposant que S n’est pas vide). Nous vérifierons ensuite que ces deux symboles forment une paire valide. Si nous atteignons la fin de l’expression et que la pile est vide, alors l’expression d’origine était bien formée avec les paires de délimiteurs correctement balancés. Sinon, il doit y avoir un délimiteur d’ouverture sur la pile sans symbole de fermeture correspondant.

* 1. **Is\_matched\_html**

Nous faisons un passage de gauche à droite à travers la chaîne brute, en utilisant l’indice j pour suivre notre progression. Nous utilisons la méthode **find()** de la classe **str** pour localiser les caractères < et > qui définissent les balises. Les balises d’ouverture sont poussées dans la pile et comparées aux balises de fermeture qui en sont extraites.

* 1. **ArrayQueue**

Nous réservons initialement une liste de taille modérée pour le stockage des données, bien que la file d’attente ait formellement la taille zéro. Par souci de technicité, nous initialisons l’indice avant à zéro. Lorsque **first()** ou **dequeue()** sont appelés sans élément dans la file d’attente, nous levons une instance de l’exception Empty.

* 1. **LinkedStack**

Chaque instance de la pile conserve deux variables. Le membre principal est une référence au nœud en tête de liste (ou à None, si la pile est vide). Nous gardons une trace du nombre actuel d’éléments avec la variable d’instance size, car sinon nous serions obligés de parcourir toute la liste pour compter le nombre d’éléments lors de la déclaration de la taille de la pile. La mise en œuvre de push reflète essentiellement le pseudo-code pour l’insertion en tête d’une liste à chaînée simple. Quand on pousse un nouvel élément e dans la pile, nous effectuons les changements nécessaires au en appelant le constructeur de la classe Node comme suit :

***self.\_head = self.\_Node(e, self.\_head)***

Lors de l’implémentation de la m´méthode top, le but est de retourner l’élément qui est au sommet de la pile. Lorsque la pile est vide, nous levons une exception ”Stack is Empty”. Lorsque la pile n’est pas vide, **self.\_head** est une référence au premier nœud de la liste chaînée. L’élément situé au top de la liste peut être identifié comme **self.\_head.element**. Notre implémentation de **pop()** reflète essentiellement le pseudo-code que nous avons donné à cet effet, sauf que nous maintenons une référence locale `a l’élément qui est stocké au nœud qui est supprimé, et nous renvoyons cet élément à l’appelant de **pop().**

* 1. **LinkedQueue**

De nombreux aspects de notre implémentation sont similaires à ceux de la classe **LinkedStack**. A titre d’exemple la définition de la classe imbriquée Node. Notre mise en œuvre de **dequeue** pour **LinkedQueue** est similaire à celle de pop pour **LinkedStack**, car les deux suppriment en tête de la liste chaînée. Cependant, il y a une différence subtile parce que notre file d’attente doit maintenir avec précision la référence de queue (aucune variable de ce type n’a été maintenue pour notre pile). En général, une opération à la tête n’a aucun effet sur la queue, mais quand **dequeue** est invoqué sur une file d’attente avec un élément, nous supprimons simultanément la queue de la liste. On affecte alors None à **self.\_queue** pour maintenir la cohérence. Il y a une complication similaire dans notre implémentation de la méthode **enqueue**. Le plus récent nœud devient toujours la nouvelle queue. Pourtant, une distinction est faite si ce nouveau nœud est le seul nœud de la liste. Dans ce cas, il devient également la nouvelle tête de liste. En termes de performances, la classe **LinkedQueue** est similaire à la classe **LinkedStack** dans la mesure où toutes les opérations s’exécutent dans le pire des cas en un temps constant, et l’utilisation de l’espace est proportionnellement linéaire au nombre actuel d’éléments.

* 1. **CircularQueue**

Les deux seules variables d’instance sont **tail**, qui est une référence au nœud de queue (ou à None lorsque la liste est vide) et la taille (size), qui est la valeur actuelle du nombre d’éléments dans la file d’attente. Lorsqu’une opération implique l’avant de la file d’attente, nous reconnaissons **self.\_tail.\_next** comme la tête de la file d’attente. Lorsque la m´méthode **enqueue** est appelée, un nouveau nœud est placé juste après la queue de la liste, mais avant la tˆete actuelle. Ainsi le nouveau nœud devient la queue de la file d’attente. En plus des opérations de file d’attente traditionnelles, la classe **CircularQueue** prend en charge une m´méthode **rotate** qui met en œuvre plus efficacement la combinaison de l’enlèvement d’un élément de l’avant de la file d’attente et sa réinsertion à l’arrière de la file d’attente. Avec une liste circulaire comme support de stockage des éléments, nous effectuons l’affectation suivante *self.\_tail = self.\_tail.\_next* pour transformer l’ancienne tête de la file d’attente en sa nouvelle queue.

* 1. **\_DoublyLinkedBase**

Le constructeur de la classe **\_DoublyLinkedBase** instancie les deux nœuds sentinelles et les relie l’un à l’autre. Nous maintenons une donnée membre **\_size** et fournissons un soutien public aux m´méthodes **\_\_len\_\_** et **is \_empty**. Les deux autres m´méthodes de notre classe sont les utilitaires non publics, **\_insert\_between** et **\_delete\_node** qui permettent l’insertion et la suppression d’un nœud. Ceux-ci fournissent un support générique pour les insertions et les suppressions, respectivement, mais n´nécessitent une ou plusieurs références de nœuds en tant que paramètres. La mise en œuvre de la méthode **\_insert\_between** est basée sur l’algorithme illustrée à la figure 7.10 (cfr notes du cours). Tout commence par la création d’un nouveau noud. Ensuite les champs de ce nouveau noud sont liés à ceux des nœuds voisins spécifiés. Pour plus de commodité plus tard, la méthode renvoie une référence au nœud nouvellement créé. L’implémentation de la méthode **\_delete\_node** est basée sur l’algorithme illustré à la figure 7.11 (cfr notes du cours). Les voisins du nœud à supprimer sont liés directement l’un `a l’autre, contournant ainsi le nœud à supprimer de la liste. A titre de formalité, nous réinitialisons intentionnellement les champs **prev**, **next** et **element** du nœud supprimé `a None (après enregistrement de l’élément `a retourner). Bien que le nœud supprimé soit ignoré par le reste de la liste, d´définir ses champs sur None est avantageux car il peut aider le ramasse-miettes de Python, car les liens inutiles vers les autres nœuds sont éliminés. Nous nous appuierons également sur cette configuration pour reconnaître un nœud comme ”deprecated” lorsqu’il ne fait plus partie de la liste.

* 1. **DynamicArray**

Bien que cohérent avec l’interface de la classe **list** de Python, nous ne fournissons que quelques fonctionnalités limitées sous la forme de quelques méthodes d’ajout, et les accesseurs **len**. La prise en charge de la création de tableaux de bas niveau est fournie par un module nommé **ctypes.** Parce que nous n’utiliserons généralement pas une structure de niveau aussi bas dans le reste de ces notes de cours, nous omettons une explication d´détaillée du module ctypes. Ici nous nous sommes contentés d’encapsuler la commande nécessaire pour d´eclarer le tableau brut dans un utilitaire privé, la méthode **\_make\_array(self,c)**.

* 1. **LinkedDequeue**

**LinkedDeque** hérite de la classe **\_DoublyLinkedBase** . Nous ne fournissons pas de méthode explicite d’initialisation **\_\_init\_\_** pour la classe **LinkedDeque**, car la version héritée suffit pour initialiser une nouvelle instance. Nous nous appuyons également sur les versions héritées des méthodes **\_\_len\_\_** et **is\_empty** pour répondre aux besoins de l’ADT **deque**. Avec l’utilisation de sentinelles, la clé de notre mise en œuvre est de se rappeler que l’en-tête ne stocke pas le premier élément du **deque**. C’est le nœud juste après l’en-tête qui stocke le premier élément (en supposant que **deque** n’est pas vide). De la même manière, c’est le nœud juste avant la queue qui stocke le dernier élément du **deque**. Nous utilisons la méthode d’insertion héritée **\_insert\_between** pour effectuer une quelconque opération d’insertions. A noter que ces opérations réussissent, même lorsque le **deque** est vide ; dans une telle situation, le nouveau nœud est placé entre les deux sentinelles. Lors de la suppression d’un élément d’un **deque** non vide, nous nous s’appuyons sur la méthode de suppression héritée **\_delete\_node**, sachant que le nœud désigné connait ses voisins de gauche et de droite.

1. Création d’un module
2. La question consiste à supprimer les implémentations redondantes et gardez une seule implémentation. Et de s’assurer que la classe ou la fonction ainsi réorganisée fournit les services requis à toutes les classes et fonctions qui en ont besoin. Ce qui es fait dans le fichier python.
3. **Processus suivi dans la mise au point de ces tests**.