UNIVERSITE DE KINSHASA

FACULTE POLYTECHNIQUE

COURS D'ALGORITHMIQUE ET PROGRAMMATION

PROJET SUR SUR LA STRUCTURE DES DONNES

Par :
Le groupe 4
MFWAMBA NDAYA 2GEI
SHIMBA ILUNGA 2GEI
BADIBANGA BADIBANGA 2GEI

Dirigé par l'assistant **MOBISA**

Année académique 2021-2022

STRUCTURES DES DONNEES

Liste des classes et fonctions :

1. CircularQueue.py
2. LinkedDeque.py
3. _DoublyLinked.py
4. ArrayStack.py
5. ArrayQueue.py
6. LinkedStack.py
7. LinkedQueue.py
8. is_matched_html.py
9. is_matched.py
10. reverse_file.py
11. DynamicArraysizeEvaluation.py

TUTORIEL DU MODULE

classe 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.

Classe 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.

Classe 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 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().

Classe 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.

Classe 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 méthode enqueue est appelée, un nouveau nœud est placé juste après la queue de la liste, mais avant la t^ete 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 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.

Classe 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 détaillée du module ctypes. Ici nous nous sommes contentés d'encapsuler la commande nécessaire pour déclarer le tableau brut dans un utilitaire privé, la méthode

```
  _make_array(self,c).
```

0.0.1 LinkedDequeu

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.