1 Structure de pile

Objectifs

- connaître la structure de données *pile*
- savoir en donner une implantation
- savoir l'utiliser

1.1 Motivation

Structure de données importante en informatique :

- pile des appels de fonctions
- utile pour parcours de graphes (trouver un chemin)
- exemple de l'évaluation d'expressions postfixées

1.2 Opérations primitives sur les piles

Les *piles* sont des structures de données linéaires admettant les *opérations primitives* suivantes :

- création d'une pile ;
- empilement d'un élément sur une pile ;
- dépilement d'une pile ;
- consultation du sommet d'une pile ;
- test de vacuité d'une pile.

Remarque

Les piles peuvent contenir en principe autant d'éléments qu'on veut (cependant en réalité elles sont en général limitées par la mémoire disponible). Mais il n'est pas possible d'accéder directement à n'importe lequel de ces éléments : seul l'élément qui y a été placé en dernier est accessible, les autres ne l'étant pas tant que ce dernier élément n'a pas été sorti de la pile. C'est le principe du *dernier entré, premier sorti* ou *Last In First Out* (LIFO en abrégé).

1.2.1 Constructeur

La construction d'une nouvelle pile ne nécessite aucun paramètre :

```
def __init__(self):
    """ Build a new empty stack"""
```

Exemple d'appel au constructeur :

```
>>> from apstack import *
>>> st = ApStack()
>>> print(st)
+-+
```

1.2.2 Sélecteur

Le seul sélecteur sur les piles est la méthode de consultation du sommet d'une pile :

```
def top(self) -> T:
    """ Return the element on top of self without removing it
        Précondition : self must be non empty """
```

1.2.3 Modificateurs

La structure de pile est une structure de données *mutable*. Cela signifie que sa valeur ou son état peut changer au cours du temps.

Deux opérations primitives permettent cette mutabilité :

- l'empilement ;
- et le dépilement.

La méthode d'empilement :

```
def push(self, el: T):
    """ Add el on top of the stack. """
```

et celle de dépilement :

```
def pop(self) -> T:
    """ Return the element on top of self
        Side effect: self contains an element less
        Précondition : self must be non empty"""
```

Exemples d'utilisation de ces sélecteurs et modificateurs :

```
>>> st.push(42)
>>> st.push(3)
>>> st.push(14)
>>> st.top()
14
>>> print(st)
|14|
| 3|
|42|
+--+
>>> st.pop()
14
>>> st.pop()
3
>>> print(st)
|42|
+--+
```

1.2.4 Prédicat

Un prédicat permettant de tester la vacuité d'une pile :

```
def is_empty(self) -> bool:
    """ Return:
            * ``True`` if self is empty
            * ``False`` otherwise """
```

Exemple d'utilisation du prédicat de vacuité :

```
>>> st.is_empty()
False
>>> st.pop()
42
>>> st.is_empty()
True
```

Remarque

Supposons que `x` est une variable associée à une valeur et `rq` une pile vide
`>>> rq = ApStack()`
`>>> rq.is_empty()`
`True`
À l'aide de ces fonctions primitives, le principe du dernier entré premier sorti peut être formalisé par le fait qu'à l'issue des appels aux deux méthodes `push` et `pop`,
`>>> rq.push(x)`
`>>> y = rq.pop()`
la variable `y` a la même valeur que la variable `x` et que l'état de la pile `rq` est exactement le même qu'avant l'appel à la première des deux méthodes.
`>>> x == y`
`True`

```
>>> rq.is_empty()
True
```
Et de même, en supposant que `rq` n'est pas vide, après l'appel aux deux méthodes, la pile `rq` est inchangée.
```
>>> tmp = deepcopy(rq)
>>> y = rq.pop()
>>> rq.push(y)
>>> rq == tmp
True
```

1.2.5 Une nouvelle exception

Le sélecteur `top` et le modificateur `pop` sont soumis à une contrainte de non vacuité de la pile à laquelle ils s'appliquent. Le non respect de cette contrainte déclenche une exception définie par le

```
class ApStackEmptyError(Exception):
    """ Exception for empty stacks """
    def __init__(self, msg):
        self.message = msg
```

Exemple de déclenchement d'une exception avec message d'erreur :

```
>>> st.pop()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/bpapegay/enseignement/licence/ap/docs/apstack.py", line 83, in pop
    raise ApStackEmptyError('empty stack, nothing to pop')
apstack.ApStackEmptyError: empty stack, nothing to pop
```

2 Structure de file

Objectifs

- connaître la structure de données *file*
- savoir en donner une implantation
- savoir l'utiliser

2.1 Motivation

Structure de données importante en informatique :

- file d'attente d'impression
- utile pour parcours de graphes (trouver un chemin)

2.2 Opérations primitives sur les files

Les *files* sont des structures de données linéaires admettant les *opérations primitives* suivantes :

- création d'une file ;
- enfilement d'un élément sur une file ;
- défilement d'une file ;
- test de vacuité d'une file.

Remarque

Les files peuvent contenir en principe autant d'éléments qu'on veut (cependant en réalité elles sont en général limitées par la mémoire disponible). Mais il n'est pas possible d'accéder directement à n'importe lequel de ces éléments : seul l'élément qui y a été placé en premier est accessible, les autres ne l'étant pas tant que ce premier élément n'a pas été sorti de la file. C'est le principe du *premier entré, premier sorti* ou *First In First Out* (FIFO en abrégé).

2.2.1 Constructeur

La construction d'une nouvelle file ne nécessite aucun paramètre :

```
def __init__(self):
    """ Build  a new empty queue """
```

Exemple d'utilisation du constructeur :

```
>>> from apqueue import *
>>> qu = ApQueue()
>>> print(qu)
↦↤
```

2.2.2 Modificateurs

La structure de file est une structure de données *mutable*. Cela signifie que sa valeur ou son état peut changer au cours du temps.

Deux opérations primitives permettent cette mutabilité :

- l'enfilement ;
- et le défilement.

La méthode d'enfilement :

```
def enqueue(self, elt: T):
    """ Insert an element elt at the begining of the queue """
```

et celle de défilement :

```
def dequeue(self) -> T:
    """ Return the element on top of self
        Side effect: self contains an element less
        Precondition: self must be non empty """
```

Exemples d'utilisation des modificateurs :

```
>>> qu.enqueue(42)
>>> qu.enqueue(3)
>>> qu.enqueue(14)
>>> print(qu)
↦14|3|42↤
>>> qu.dequeue()
42
>>> qu.dequeue()
3
>>> print(qu)
↦14↤
```

Remarque

Pour accéder à la valeur de l'élément placé en premier d'une file, il est obligatoire de le sortir de la file. Cette valeur ne peut pas être uniquement consultée (comme pour les piles).

2.2.3 Prédicat

Un prédicat permettant de tester la vacuité d'une file :

```
def is_empty(self) -> bool:
    """ Return:
          * ``True`` if s is empty
          * ``False`` otherwise """
```

Exemples d'utilisation de ce prédicat :

```
>>> qu.is_empty()
False
```

```
>>> qu.dequeue()
14
>>> qu.is_empty()
True
```

2.2.4 Une nouvelle exception

Le modificateur `dequeue` est soumis à une contrainte de non vacuité de la file à laquelle ils s'appliquent. Le non respect de cette contrainte déclenche une exception définie par le

```
class ApQueueEmptyError(Exception):
    """ Exception for empty stacks """
```

Exemple de déclenchement d'une exception avec message d'erreur :

```
>>> qu.dequeue()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/bpapegay/enseignement/licence/ap/docs/apqueue.py", line 79, in dequeue
    raise ApQueueEmptyError('empty queue, nothing to dequeue')
apqueue.ApQueueEmptyError: empty queue, nothing to dequeue
```