Devoir 3
Structure de Données

Guillaume Poirier-Morency p1053380
Vincent Antaki p1038646

24 novembre 2014

Résumé

Implémentation et analyse empirique d'une file à priorités multiples.

Le projet utilise benchpy pour faire l'analyse empirique des temps d'exécution des fonctions. Ce module est très pratique, car il permet de faire le suivit d'une fonction à l'aide d'un décorateur.

Le travail est accompagné de tests unitaires qui assurent le bon fonctionnement des structures de données. Ceux-ci peuvent être exécutés avec l'utilitaire nosetests.

Vous pouvez l'installer les dépendances avec pip :

```
pip install --user benchpy nosetests
```

1 Casino

1.1 Implantation du Casino

Les Figure 1 et Figure 2 décrivent minimalement l'implantation du Casino.

1.2 Description du fonctionnement du Casino

1.2.1 Classe Queue

Notre implémentation du Casino est basée sur la classe Queue. La file est représentée par une liste simplement chainée. Elle respecte l'interface enqueue dequeue et remove.

Elle utilise les fonctions magiques de Python pour fournir un iterable, la longueur, l'élément suivant, l'appartenance et la valeur booléen. C'est une approche qui facilite l'implémentation et qui favorise une utilisation simple de la structure de donnée.

1.2.2 Classe CasinoQueue

Une instance de la classe CasinoQueue est une file à 3 priorités. Elle possède les fonctions de base des files : enqueue et dequeue. Elle possède 3 files basé sur la classe Queue correspondant à ses trois priorités :

— table brisée ;
— changement de table ;
— nouveaux joueurs.

Les éléments mis dans les files sont des tuplets de joueur, table désirée et temps d'entrée. None comme valeur de table signifie que le joueur ne veut pas de table en particulier.

Lorsque appelée, la fonction dequeue retire, en tout respect de l'énoncé, une personne de la file. Une exception IndexError est lancée si il n'y a plus d'élément dans la file.

dequeue possède un ordre constant lorsque il y a des éléments dans la queue pour table brisée et a une ordre linéaire par rapport au nombre de joueur dans la queue pour changement de table dans tous les autres cas.

Chaque appel de la fonction enqueue vérifie que le nom entrée n'est pas un doublon d'un nom existant déjà dans le casino. Si ce n'est pas le cas, les paramètres table et broken détermineront à quelle file sera enqueue le joueur. Une fois ajouté dans la file, le joueur est aussi ajouté à l'ensemble des joueurs de la file pour accélérer la recherche.

FIGURE 1 – Diagramme de classes des structures de données

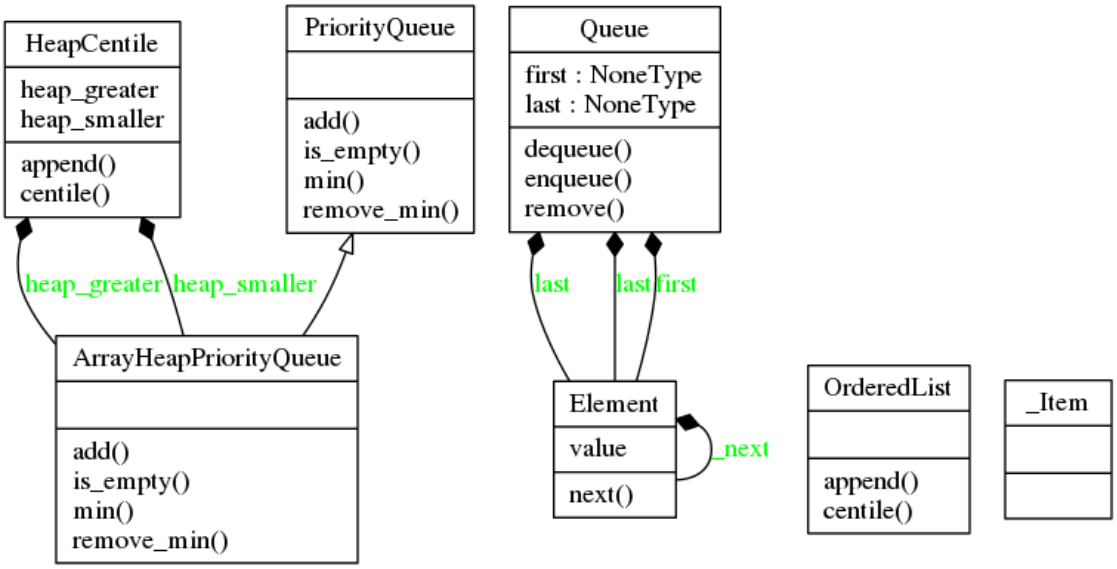


FIGURE 2 – Diagramme de classes du Casino.

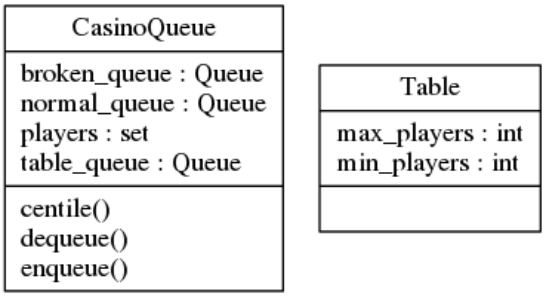


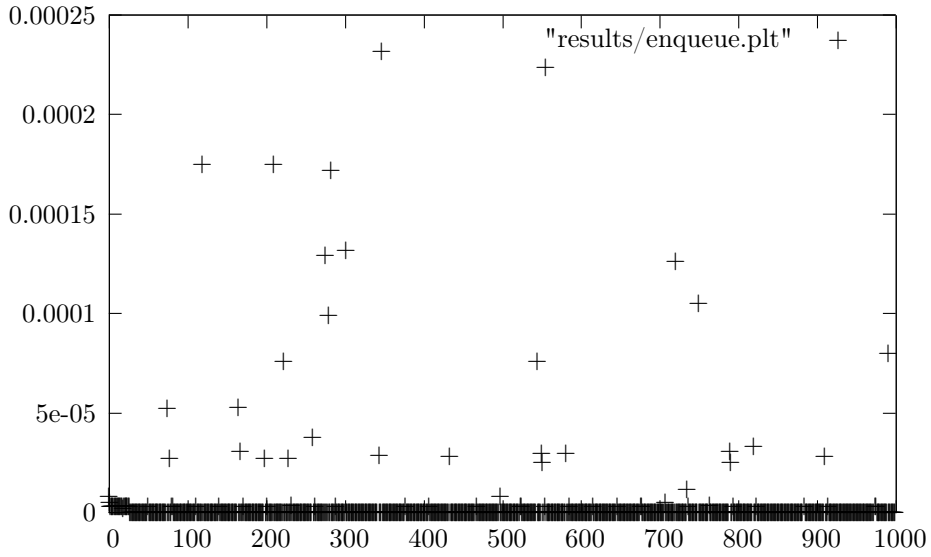
FIGURE 3 – La file du Casino est composée de trois files Queue.

```
class CasinoQueue:
    """
␣␣␣␣Représentation␣du␣casino.

␣␣␣␣Ce␣casino␣est␣un␣ensemble␣de␣3␣files␣abstrait␣comme␣une␣seule␣file␣de
␣␣␣␣priorité.

␣␣␣␣Les␣opérations␣de␣base␣sont␣O(1).
␣␣␣␣"""
    def __init__(self, players=set(), centile=OrderedList()):
        """Initialise␣un␣casino␣avec␣des␣joueurs␣initiaux"""
        self.players = set() # set de tout les players

        self._centile = centile

        # files
        self.broken_queue = Queue()
        self.table_queue = Queue()
        self.normal_queue = Queue()
        for player in players:
            self.enqueue(player)
```

FIGURE 4 – La file est implémentée avec une liste chaînée et conserve une référence vers le début et la fin de la liste pour enqueue et dequeue dans l'ordre de $O(1)$.

```

class Queue:
    """
␣␣␣␣Cette␣file␣est␣composée␣d'éléments␣simplement␣chaînées.

␣␣␣␣La␣plupart␣des␣opérations␣sont␣réduites␣à␣O(1).
␣␣␣␣"""
    class Element:
        """Élément␣de␣la␣file"""
        __slots__ = ['value', '_next']
        def __init__(self, value, n=None):
            self.value = value
            self._next = n

        def __next__(self):
            return self._next

        def next(self):
            return self._next
```

FIGURE 5 – Temps d'exécution de enqueue en fonction du nombre d'entrée.



Il nous aurait été possible d'implémenter la vérification des doublons par une itération à travers les 3 files, mais cela aurait coûté une complexité d'ordre $O(n)$. Il nous aurait été possible de faire un arbre qui stocke les noms de tout les joueurs qui sont dans le casino et qui font des recherches en $O(log(n))$.

Pour cause de flemmardise, nous avons utilisé un HashSet (set en Python) pour tester rapidement $O(1)$ l'appartenance du joueur à la file du Casino. Le test s'effectue à travers la méthode magique __contains__ de CasinoQueue.

1.2.3 Analyse empirique de Queue

Tous les tests ont été exécutés sur 1000 entrées.

L'opération enqueue devrait se faire en temps constant, car la file possède une référence vers la fin, last.

Par analyse empirique en Figure 5, on constate que la file enqueue en temps constant avec une trentaine de données aberrantes.

L'opération dequeue devrait se faire en temps constant, car il correspond à substituer le deuxièment élément de la file par le premier et de retourner le premier. Avec une référence sur le premier élément, le tout se fait en temps constant.

On constate que la file dequeue en temps constant sur la Figure 6. Comme pour l'enqueue, il y a une trentaine de données aberrantes négligables.

L'opération remove devrait se faire en temps $O(n)$ en pire cas lorsque l'élément qui est enlevé se trouve à la fin de la liste.

Dans la Figure 7, la file était initialisé à 1000 items à chaque opération. On constate qu'enlever un élément de la liste se fait en temps linéaire sur le nombre d'éléments.

2 Calcul du n-ième centile

Les centils sont calculés avec soit une liste ordonnée OrderedList ou deux monceaux. L'implantation du monceaux était celle fournit avec dans le cadre du cours. Les calculateurs étaient interfacés pour fournir une méthode unique d'ajout append, qui est testé dans ce cas d'analyse.

2.1 Calcul de centile par une liste ordonnée

La liste ordonnée est implantée en héritant du type list de Python. La méthode append a été surchargée afin d'insérer de manière ordonnée dans la structure de donnée.

Il faut un temps p pour déterminer que un nouvel élément doit être inséré à la position p et un temps $n - p$ pour décaler les éléments qui suivent l'endroit ou l'insertion se fait. Par conséquent, la

FIGURE 6 – Temps d'exécution de dequeue en fonction du nombre d'entrée.

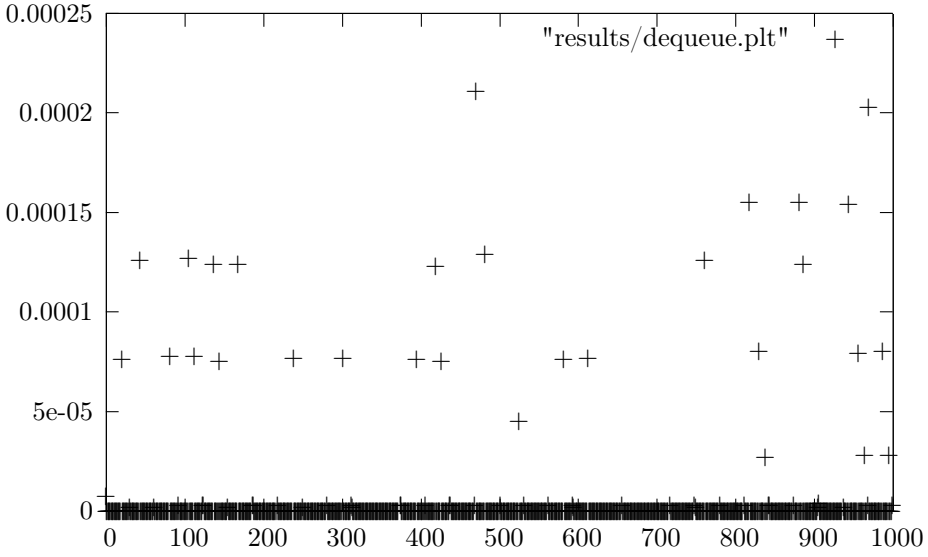


FIGURE 7 – Temps d'exécution de remove en fonction du nombre d'entrée.

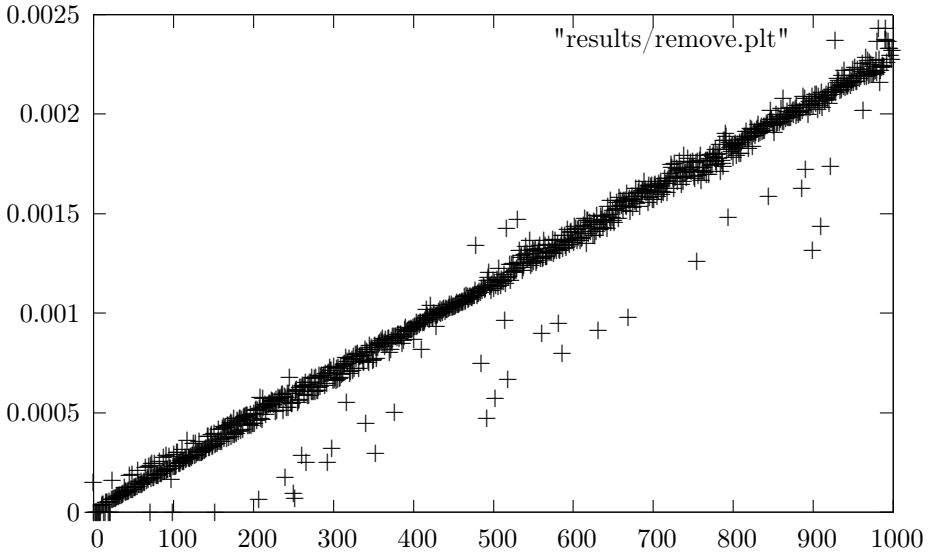
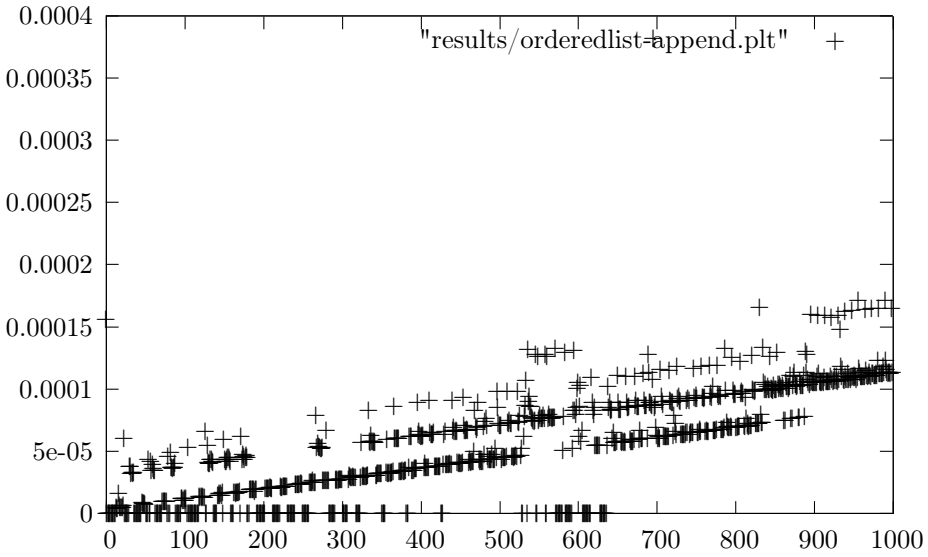


FIGURE 8 – Temps d'exécution de append sur OrderedList en fonction du nombre d'entrée.



complexité est toujours d'ordre $O(n)$. Une insertion à la fin nécéssite un parcours complet et une insertion au début nécéssite un décalage complet.

On peut observer 3 droites fortes et une droite faible en Figure 8. Le fait de mélanger décalage et parcours est probablement la raison de cette observation.

2.2 Calcul de centile par deux monceaux

Pour implanter le calcul du centile à l'aide de deux monceaux, nous avons créé la classe HeapCentile qui stocke des données dans deux monceaux minumum et maximum. Après chaque ajout par append, le monceaux sont normalisés autours du centile de référence.

La conception de la fonction _normalize est de telle sorte qu'elle peut normaliser les deux monceaux autours de n'importe quel centile comme pour le OrderedList qui retourne le centile demandé.

Les deux droites constantes observées en Figure 9 sont dûes au fait de normaliser à chaque ajout dans le monceau. Lorsque les monceaux possède un nombre différent d'entrées, un coût d'ordre $O(log(n))$ s'ajoute au fait de compter d'enlever et d'insérer dans un monceau.

Quant à faire, il aurait été préférable de simplement insérer dans le monceau sans normaliser afin de pouvoir observer la coube logarithmique associée aux opérations sur les monceaux, mais nous avons préféré respecter l'énoncé.

FIGURE 9 – Temps d'exécution de append sur HeapCentile en fonction du nombre d'entrée.