

# 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.

## 1 Casino

### 1.1 Implantation du Casino

Les figures 1 et 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 chaînée. 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 booléenne. C'est une approche qui facilite l'implémentation et qui favorise une utilisation simple de la structure de donnée.

#### 1.2.2 Classe CasinoQueue

Les instances de la classe CasinoQueue sont des files à 3 priorités. Elles possèdent les fonctions de base des files : `enqueue` et `dequeue`. Elle possède 3 files chaînées de la classe Queue correspondant à ses trois priorités (table brisée, changement de table et le reste). Les éléments mis dans les files sont des tuplets de joueur, temps d'entrée dans la file et, dans le cas de la file pour changement de table, la table désirée.

Lorsque appelée, la fonction `dequeue` retire, en tout respect de l'énoncé, une personne de la file ou retourne une exception `IndexError` si cela est impossible. `dequeue` possède un ordre constant lorsque il y a des éléments dans la queue pour table brisée et a un 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.

Il nous aurait été possible d'implémenter la vérification des doublons par une itération à travers les 3 queues. Pour cause de mauvaise complexité, nous avons refusé cette option. 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 refusé cette option. Nous avons implémenté `__contains__` qui test l'appartenance à un objet à `self.players` (un set!) lors de l'entrée d'un nouveau joueur dans le casino (ajout d'un joueur à la `normal_queue`).

#### 1.2.3 Analyse empirique de Queue

Tous les tests ont été exécutés sur 1000 entrées.

On constate que la file `enqueue` en temps constant.

On constate que la file `dequeue` en temps constant.

Dans ce cas, 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.

```

        self._normalize(self._centile)

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

    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

    __slots__ = ['first', 'last']

```

FIGURE 1 – 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 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 2 – La file du Casino est composée de trois files Queue.

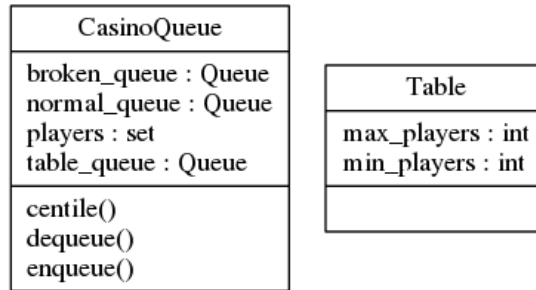


FIGURE 3 – Diagramme de classes du Casino.

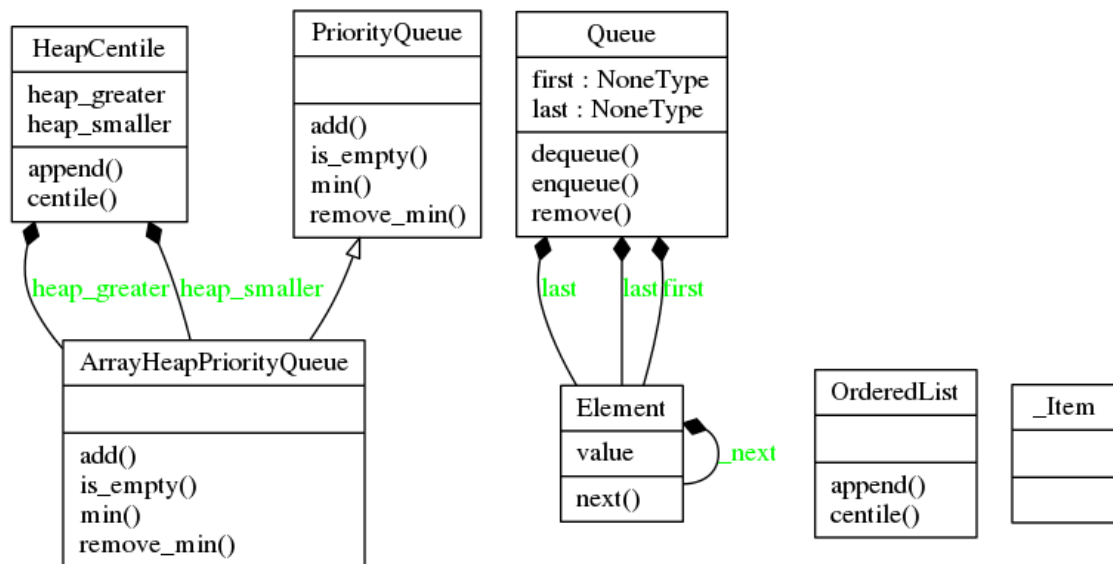


FIGURE 4 – Diagramme de classes des structures de données

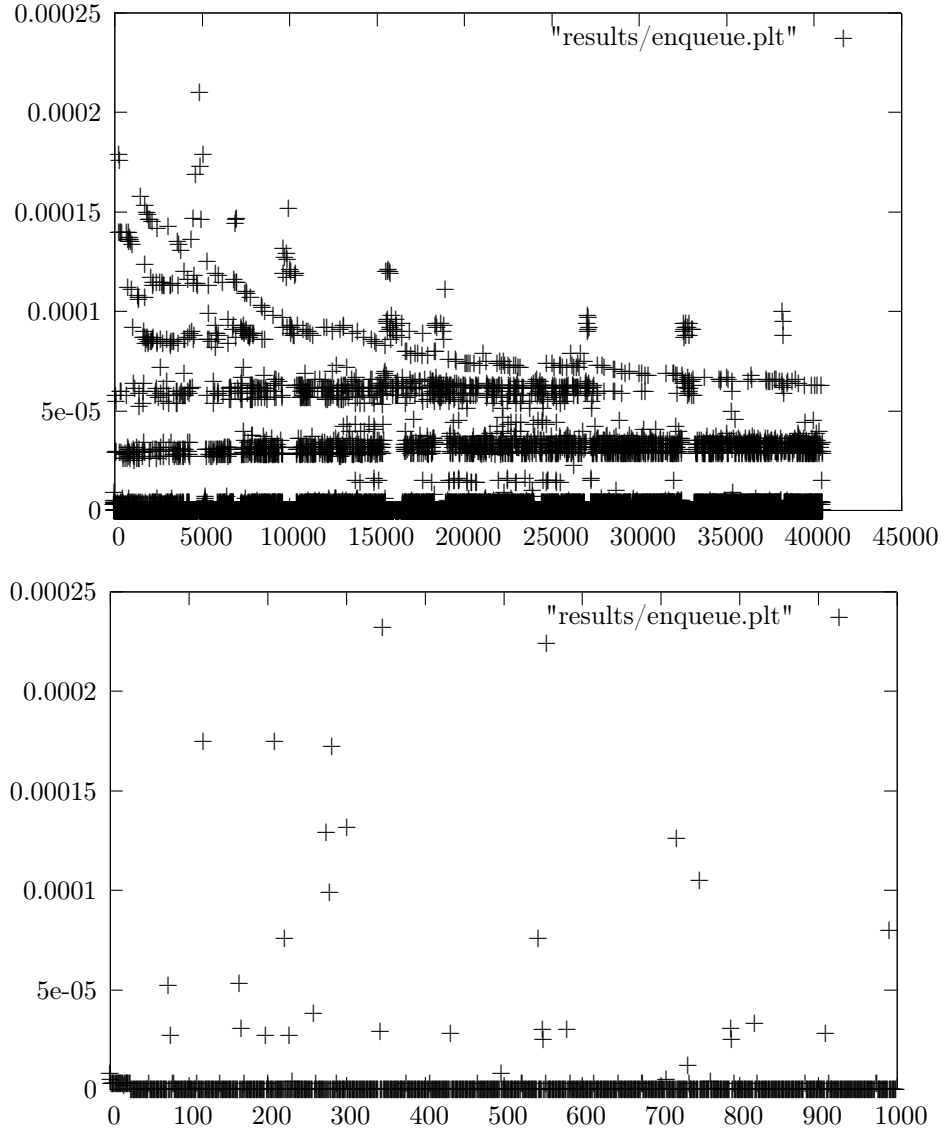


FIGURE 5 – Temps d'exécution de enqueue en fonction du nombre d'entrée.

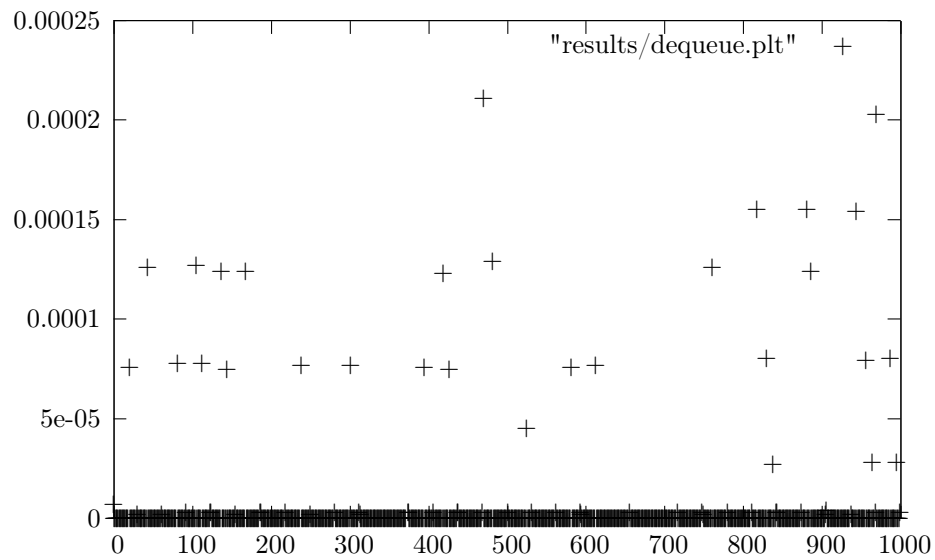


FIGURE 6 – Temps d'exécution de `dequeue` en fonction du nombre d'entrée.

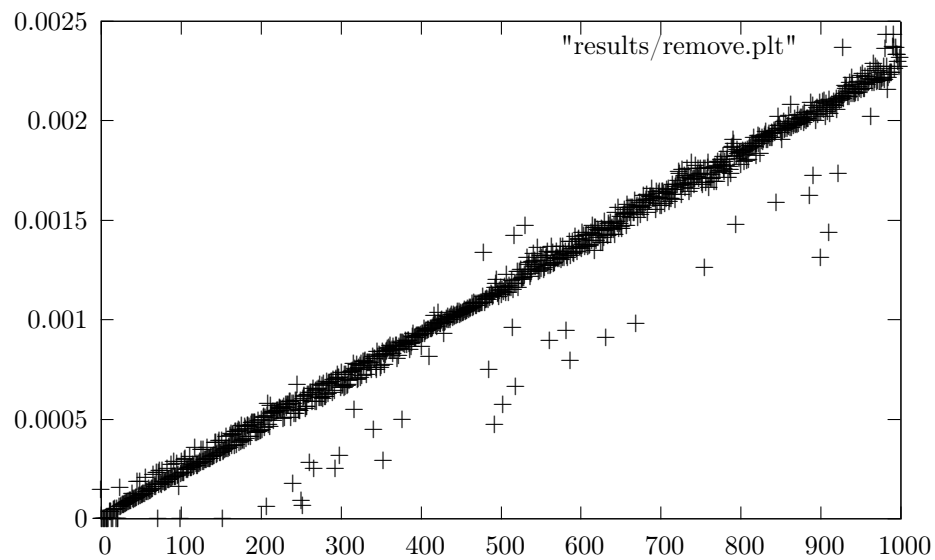


FIGURE 7 – Temps d'exécution de `remove` en fonction du nombre d'entrée.

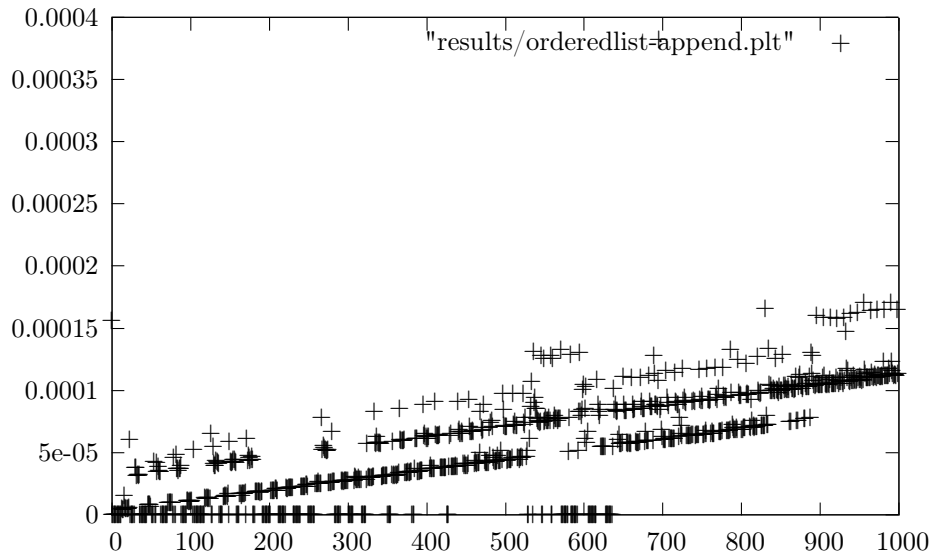


FIGURE 8 – Temps d'exécution de `append` en fonction du nombre d'entrée.

## 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

Dans ce cas, 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.

### 2.2 Calcul de centile par deux monceaux

Dans ce cas, la file était initialisé à 500 items à chaque opération. On constate qu'enlever un élément de la liste se fait en temps linéaire sur le nombre d'éléments.

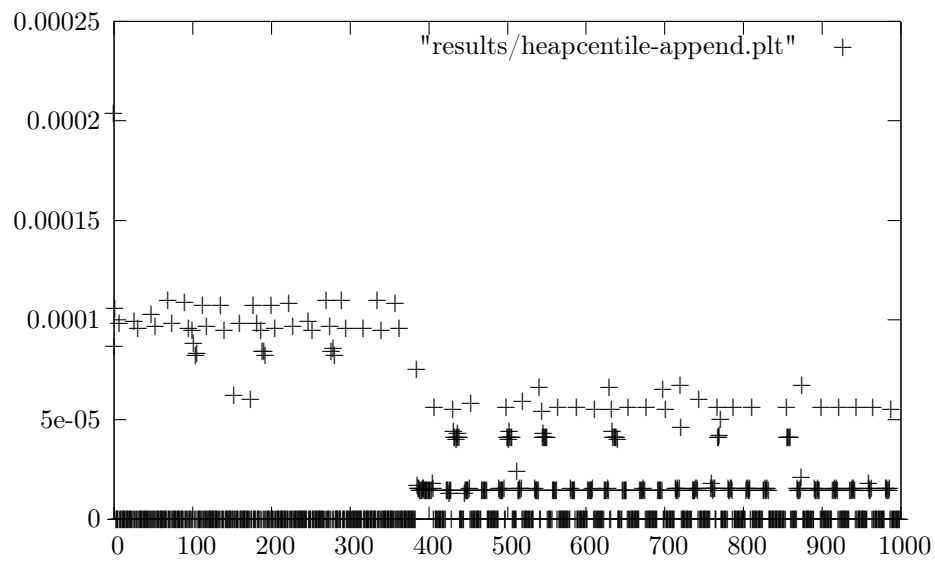


FIGURE 9 – Temps d'exécution de `append` en fonction du nombre d'entrée.