DEVOIR MAISON

Informatique théorique

07 DECEMBRE 2020

PANETIER CAMILLE & CONSTANCEAU ENOLA
Université de Bordeaux, L3 MIASHS

Devoir maison

Informatique théorique

Le but de ce projet est de comparer deux implémentations de file avec priorités bornées.
Queues : les éléments arrivent avec un « ticket » contrairement aux files classiques, c'est leur priorité (compris entre $0 \leq p < n$). On fait donc sortir de la queue selon les tickets et pas l'ordre d'arrivée (le premier sorti est le premier entré avec la plus haute priorité, c'est-à-dire la plus petite valeur). Cependant, s'il y a deux éléments avec la même priorité (ticket) alors là on prend en compte l'ordre d'arrivée (premier arrivé=premier sorti). Plus la valeur de la priorité est faible, plus la priorité est élevée.

Faire toutes les fonctions pour les deux méthodes

- **BoundedOneQueue** : file avec priorité borné avec une seule liste simplement chaînée, munie de 2 sentinelles

V	P
V	P
…	…
…	…

- **BoundedListQueue** : file avec priorité bornée avec liste de taille fixe, le ième élément de la liste correspond à la file de priorité. Les informations stockées sont les valeurs.

Liste	**Indice (donne priorité)**
V V V	0
V V	1
V	2
…	…

Ici nous allons utiliser des files avec priorité bornée. Pour caractériser une file on peut identifier les fonctions suivantes :

Lexique fr	Réalisation
Créer	__init__
Priorité max	max_priority
Longueur	__len__
Ôter	pop
Ajouter	push
Premier	first
Vide	empty
Liste présente	to_list
Élément priorité	howmany
Sommaire priorité	summary

Types :
Ici $\mathbb{V}$ désigne l'ensemble des valeurs, $\mathbb{P}$ est l'ensemble des priorités, $\mathbb{E}$ est l'ensemble $\mathbb{V} \times \mathbb{P}$
Queue [$\mathbb{E}$] désignera les queues permettant de stocker des éléments de type $\mathbb{E}$.

Fonctions :
Cette partie précise les opérations disponibles sur le TdA et leurs signatures, c'est à-dire le nombre et le type de paramètres attendus ainsi que le type du résultat.
<u>BoundedOneQueue</u>

- __init__ : $\mathbb{N} \rightarrow$ BoundedOneQueue [$\mathbb{E}$]
- max_priority : BoundedOneQueue [$\mathbb{E}$]$\rightarrow \mathbb{N}$
- __len__ : BoundedOneQueue [$\mathbb{E}$] $\rightarrow \mathbb{N}$
- pop : BoundedOneQueue [$\mathbb{E}$]$\nrightarrow$None
- push : BoundedOneQueue[$\mathbb{E}$]x $\mathbb{E}$ $\rightarrow$None
- first : BoundedOneQueue [$\mathbb{E}$] $\nrightarrow \mathbb{E}$
- empty : BoundedOneQueue [$\mathbb{E}$] $\rightarrow$ Bool
- to_list : BoundedOneQueue [$\mathbb{E}$]$\rightarrow$list[$\mathbb{E}$]
- howmany : $\mathbb{N} \rightarrow \mathbb{N}$
- summary : BoundedOneQueue [$\mathbb{E}$] $\rightarrow$ list[$\mathbb{P}$]

<u>BoundedListQueue</u>

- __init__ : $\mathbb{N} \rightarrow$ BoundedListQueue [$\mathbb{E}$]
- max_priority : BoundedListQueue [$\mathbb{E}$]$\rightarrow \mathbb{N}$
- __len__ : BoundedListQueue [$\mathbb{E}$] $\rightarrow \mathbb{N}$
- pop : BoundedListQueue [$\mathbb{E}$] $\nrightarrow$None
- push : BoundedListQueue[$\mathbb{E}$]x $\mathbb{E}$ $\rightarrow$None
- first : BoundedListQueue [$\mathbb{E}$] $\nrightarrow \mathbb{E}$
- empty : BoundedListQueue [$\mathbb{E}$] $\rightarrow$ Bool
- to_list : BoundedListQueue [$\mathbb{E}$]$\rightarrow$list[$\mathbb{E}$]
- howmany : $\mathbb{N} \rightarrow \mathbb{N}$
- summary : BoundedListQueue [$\mathbb{E}$] $\rightarrow$ list[$\mathbb{P}$]

On a bien les mêmes signatures pour les deux méthodes.

Axiomes :
Les axiomes sont des formules logiques toujours vraies, ils permettent de spécifier le comportement des fonctions.
Ils sont notés sur le fichier python.

Préconditions :
Du fait de l'existence de fonctions partielles dans le TdA, il faut exprimer à quelle(s) condition(s) la fonction fournira un résultat. On utilisera le mot clef « nécessite » ou « require », pour décrire quand les arguments appartiennent au domaine de la fonction.

- $\forall$ v, p $\in$ Queue[$\mathbb{E}$], pop(v, p) nécessite not empty(v, p)
- $\forall$ v, p $\in$ Queue[$\mathbb{E}$], first(v, p) nécessite not empty(v, p)

Afin de comparer les implémentations de chaque méthode, nous nous sommes intéressées aux complexités en nombres d'instructions.

Complexité :

QNode	Instructions	Complexité
__init__	3	$O(1)$
__repr__	1	$O(1)$
__str__	1	$O(1)$
Value	1	$O(1)$
Priority	1	$O(1)$
Next	1	$O(1)$
Next setter	2+max(1,0)	$O(1)$
Méthode **BoundedOneQueue**	Instructions	Complexité
__init__	4	$O(1)$
max_priority	1	$O(1)$
cpt	1	$O(1)$
hq	1	$O(1)$
tq	1	$O(1)$
__len__	1	$O(1)$
pop	1+max(1, 2+(1+max(1,0))) = 5	$O(1)$
push	11+7n	$O(n)$
first	1+max(1,1) = 2	$O(1)$
empty	1	$O(1)$
to_list	2+3n+1=3+3n	$O(n)$
howmany	2+(n*(1+max(1,0)+1))+1= 3+3n	$O(n)$
summary	4+(n*(2+max(2,(1+1+max(1,0)))))+1= 5+5n	$O(n)$
Méthode **BoundedListQueue**	Instructions	Complexité
__init__	3+(n*append)	$O(n)$
max_priority	1	$O(1)$
cpt	1	$O(1)$
list	1	$O(1)$
__len__	1	$O(1)$
pop	1+max(1,3+n*(1+max(2,0)+1)))=4+4n	$O(n)$
push	1+max(1,2)=3	$O(1)$
first	1+max(1, 1+n*(1+max(1,0) +1))=2+3n	$O(n)$
empty	1	$O(1)$
to_list	2+n*(1+max(n+append,0)+1)+1=3+(n*(2+n+append))	$O(n^2)$
howmany	len()	$O(len)$
summary	3+n(append)	$O(n * append)$

Nous devons comparer les deux méthodes en termes de complexité. Nous allons donc comparer les fonctions entre elles pour voir lesquelles sont les « meilleures ».
Un algorithme de forte complexité a un comportement moins efficace qu'un algorithme de faible complexité, il est donc généralement plus lent.

Fonctions	**BoundedOneQueue A**	**BoundedListQueue B**	Meilleure méthode
__init__	$O(1)$	$O(n)$	A
max_priority	$O(1)$	$O(1)$	Équivalente
__len__	$O(1)$	$O(1)$	Équivalente
pop	$O(1)$	$O(n)$	A
push	$O(n)$	$O(1)$	B
first	$O(1)$	$O(n)$	A
empty	$O(1)$	$O(1)$	Équivalente
to_list	$O(n)$	$O(n^2)$	A
Howmany	$O(n)$	$O(len)$	A
summary	$O(n)$	$O(n * append)$	A

Nous avons comparé les deux méthodes BoundedOneQueue et BoundedListQueue dans le tableau ci-dessus.
On remarque que la méthode B, est meilleur en termes de complexité, que pour la fonction push. Ensuite, concernant les fonctions __init__, pop, first, to_list, howmany et summary, la méthode A est plus efficace. Enfin, pour les fonctions restantes, il n'y pas de différence entre les complexités.
On peut supposer qu'avec le programme que nous avons écrit et les complexités associées à chaque fonction, la méthode **BoundedOneQueue** sera plus rapide et plus efficace en termes de complexité.