Mesures de temps d’exécution d’opérations sur différentes structures de données avec Python

BONNAIRE Lucas & RENAULT Rémi

16/12/2024

Toutes les mesures ont été prises sur la base de scripts Python exécutés sur un MacBook Pro M4 Pro avec 24Go de RAM.

Le code source utilisé pour ces mesures est disponible sur ce repository GitHub :

https://github.com/CORT1N/esgi-algorithmic-works

Toutes les mesures ont été moyennées sur la base de 100 exécutions.

Table des matières

1. Le tas

Un tas (ou heap) est une structure de données arborescente qui respecte la propriété d'un ordre partiel, où pour un tas max, chaque parent est supérieur ou égal à ses enfants, et pour un tas min, chaque parent est inférieur ou égal à ses enfants.

Pour 100 valeurs entre 0 et 9 :

```
The build_max_heap function took on average 0.014004706900000012 ms
The heap_sort function took on average 0.07533550339999998 ms
```

Pour 1000 valeurs entre 0 et 9 :

```
The build_max_heap function took on average 0.14733076050000002 ms
The heap_sort function took on average 1.1390876774999998 ms
```

Pour 10000 valeurs entre 0 et 9 :

```
The build_max_heap function took on average 1.5088915826000002 ms
The heap_sort function took on average 15.518636703700006 ms
```

Pour 100 valeurs entre 0 et 99 :

```
The build_max_heap function took on average 0.015234948000000012 ms
The heap_sort function took on average 0.07939577149999998 ms
```

Pour 1000 valeurs entre 0 et 99 :

```
The build_max_heap function took on average 0.15716314220000002 ms
The heap_sort function took on average 1.2498736381999995 ms
```

Pour 10000 valeurs entre 0 et 99 :

```
The build_max_heap function took on average 1.5876746175000007 ms
The heap_sort function took on average 16.923911571699996 ms
```

Pour 100 valeurs entre 0 et 999 :

```
The build_max_heap function took on average 0.014588833700000008 ms
The heap_sort function took on average 0.08021593149999998 ms
```

Pour 1000 valeurs entre 0 et 999 :

```
The build_max_heap function took on average 0.1576852793000001 ms
The heap_sort function took on average 1.3103222847000005 ms
```

Pour 10000 valeurs entre 0 et 999 :

```
The build_max_heap function took on average 1.7072987555999999 ms
The heap_sort function took on average 17.849421500999995 ms
```

2. La file de priorité

Une file de priorité est une structure de données dans laquelle chaque élément possède une priorité, et les éléments sont extraits en fonction de leur priorité (le plus élevé en priorité est extrait en premier), généralement implémentée à l'aide d'un tas.

Pour 100 valeurs entre 0 et 9 :

```
The max_heap function took on average 3.09943e-05 ms
The extract_max_heap_sort function took on average 0.0006604173999999994 ms
The increase_key_heap function took on average 0.0008273099 ms
The insert_max_heap function took on average 0.00044107339999999975 ms
```

Pour 1000 valeurs entre 0 et 9 :

```
The max_heap function took on average 8.1062e-05 ms
The extract_max_heap_sort function took on average 0.0007319442000000003 ms
The increase_key_heap function took on average 0.0010108925000000006 ms
The insert_max_heap function took on average 0.00048398869999999983 ms
```

Pour 10000 valeurs entre 0 et 9 :

```
The max_heap function took on average 6.67569e-05 ms
The extract_max_heap_sort function took on average 0.0009512890000000001 ms
The increase_key_heap function took on average 0.0019884114000000004 ms
The insert_max_heap function took on average 0.0006413449999999997 ms
```

Pour 100 valeurs entre 0 et 99 :

```
The max_heap function took on average 2.14576e-05 ms
The extract_max_heap_sort function took on average 0.0005841238999999996 ms
The increase_key_heap function took on average 0.0006008129999999993 ms
The insert_max_heap function took on average 0.0004935250999999997 ms
```

Pour 1000 valeurs entre 0 et 99 :

```
The max_heap function took on average 2.14576e-05 ms
The extract_max_heap_sort function took on average 0.0005841238999999996 ms
The increase_key_heap function took on average 0.0006008129999999993 ms
The insert_max_heap function took on average 0.0004935250999999997 ms
```

Pour 10000 valeurs entre 0 et 99 :

```
The max_heap function took on average 2.14576e-05 ms
The extract_max_heap_sort function took on average 0.0005841238999999996 ms
The increase_key_heap function took on average 0.0006008129999999993 ms
The insert_max_heap function took on average 0.0004935250999999997 ms
```

Pour 100 valeurs entre 0 et 999 :

```
The max_heap function took on average 2.14576e-05 ms
The extract_max_heap_sort function took on average 0.0005841238999999996 ms
The increase_key_heap function took on average 0.0006008129999999993 ms
The insert_max_heap function took on average 0.0004935250999999997 ms
```

Pour 1000 valeurs entre 0 et 999 :

```
The max_heap function took on average 2.14576e-05 ms
The extract_max_heap_sort function took on average 0.0005841238999999996 ms
The increase_key_heap function took on average 0.0006008129999999993 ms
The insert_max_heap function took on average 0.0004935250999999997 ms
```

Pour 10000 valeurs entre 0 et 999 :

```
The max_heap function took on average 2.14576e-05 ms
The extract_max_heap_sort function took on average 0.0005841238999999996 ms
The increase_key_heap function took on average 0.0006008129999999993 ms
The insert_max_heap function took on average 0.0004935250999999997 ms
```

3. La pile

Une pile (ou stack) est une structure de données qui suit le principe “Last In, First Out” (LIFO), où le dernier élément ajouté est le premier à être retiré.

Pour 100 valeurs entre 0 et 9 :

```
The push function took on average 0.0015401833999999998 ms
The pop function took on average 0.16236299000000007 ms
```

Pour 1000 valeurs entre 0 et 9 :

```
The push function took on average 0.012969970900000006 ms
The pop function took on average 1.592159250000001 ms
```

Pour 10000 valeurs entre 0 et 9 :

```
The push function took on average 0.1207542425 ms
The pop function took on average 15.708923379999998 ms
```

Pour 100 valeurs entre 0 et 99 :

```
The push function took on average 0.0015640252000000004 ms
The pop function took on average 0.17428396000000007 ms
```

Pour 1000 valeurs entre 0 et 99 :

```
The push function took on average 0.012202261500000002 ms
The pop function took on average 1.5308857300000014 ms
```

Pour 10000 valeurs entre 0 et 99 :

```
The push function took on average 0.12033462519999993 ms
The pop function took on average 15.020608869999997 ms
```

Pour 100 valeurs entre 0 et 999 :

```
The push function took on average 0.0016307828 ms
The pop function took on average 0.1676082300000001 ms
```

Pour 1000 valeurs entre 0 et 999 :

```
The push function took on average 0.018033980900000007 ms
The pop function took on average 1.913785859999998 ms
```

Pour 10000 valeurs entre 0 et 999 :

```
The push function took on average 0.12476682660000003 ms
The pop function took on average 15.840053600000003 ms
```

4. La file

Une file (ou queue) est une structure de données qui suit le principe “First In, First Out” (FIFO), où le premier élément ajouté est le premier à être retiré.

Pour 100 valeurs entre 0 et 9 :

```
The enqueue function took on average 0.0015687939000000003 ms
The dequeue function took on average 0.0024032601 ms
```

Pour 1000 valeurs entre 0 et 9 :

```
The enqueue function took on average 0.012905598500000002 ms
The dequeue function took on average 0.060484408899999995 ms
```

Pour 10000 valeurs entre 0 et 9 :

```
The enqueue function took on average 0.1186990742 ms
The dequeue function took on average 3.5740780835 ms
```

Pour 100 valeurs entre 0 et 99 :

```
The enqueue function took on average 0.0014257422000000012 ms
The dequeue function took on average 0.0023579605 ms
```

Pour 1000 valeurs entre 0 et 99 :

```
The enqueue function took on average 0.013401507200000001 ms
The dequeue function took on average 0.055298804299999998 ms
```

Pour 10000 valeurs entre 0 et 99 :

```
The enqueue function took on average 0.11907815960000002 ms
The dequeue function took on average 3.583829403100001 ms
```

Pour 100 valeurs entre 0 et 999 :

```
The enqueue function took on average 0.0018072125999999992 ms
The dequeue function took on average 0.0026082999 ms
```

Pour 1000 valeurs entre 0 et 999 :

```
The enqueue function took on average 0.0242042551 ms
The dequeue function took on average 0.061342715199999995 ms
```

Pour 10000 valeurs entre 0 et 999 :

```
The enqueue function took on average 0.12762546610000008 ms
The dequeue function took on average 3.6565446855000006 ms
```

5. La liste chaînée

Une liste chaînée est une structure de données composée d'une séquence d'éléments, appelés "nœuds", où chaque nœud contient une valeur et une référence (ou lien) vers le nœud suivant, permettant une gestion dynamique de la mémoire.

Pour 100 valeurs entre 0 et 9 :

```
The insertion function took on average 0.0032377230000000005 ms
The search function took on average 0.0002098079 ms
The deletion function took on average 0.00036716329999999993 ms
```

Pour 1000 valeurs entre 0 et 9 :

```
The insertion function took on average 0.14765739409999995 ms
The search function took on average 0.0016808501000000007 ms
The deletion function took on average 0.00310421 ms
```

Pour 10000 valeurs entre 0 et 9 :

```
The insertion function took on average 12.899484634600006 ms
The search function took on average 0.014634132000000001 ms
The deletion function took on average 0.03199577270000001 ms
```

Pour 100 valeurs entre 0 et 99 :

```
The insertion function took on average 0.003321169700000001 ms
The search function took on average 0.00022172840000000003 ms
The deletion function took on average 0.00037431629999999999 ms
```

Pour 1000 valeurs entre 0 et 99 :

```
The insertion function took on average 0.14466762469999994 ms
The search function took on average 0.0018239016999999995 ms
The deletion function took on average 0.0032067296000000012 ms
```

Pour 10000 valeurs entre 0 et 99 :

```
The insertion function took on average 12.977037429400003 ms
The search function took on average 0.013866424199999998 ms
The deletion function took on average 0.0315618512 ms
```

Pour 100 valeurs entre 0 et 999 :

```
The insertion function took on average 0.0032711018000000014 ms
The search function took on average 0.00021696010000000002 ms
The deletion function took on average 0.0004148470999999998 ms
```

Pour 1000 valeurs entre 0 et 999 :

```
The insertion function took on average 0.14839887589999998 ms
The search function took on average 0.0015306465000000003 ms
The deletion function took on average 0.0028800960999999987 ms
```

Pour 10000 valeurs entre 0 et 999 :

```
The insertion function took on average 13.1405615813 ms
The search function took on average 0.014843940200000006 ms
The deletion function took on average 0.03206729879999999 ms
```

Interprétation des résultats

En commençant par le moins probant, le tas paraît très affecté durant son tri lorsqu'on atteint les 10000 valeurs.

Ensuite, la file de priorité semble résister à n'importe quelle quantité de valeurs.

La pile quant à elle ne semble être affectée qu'en suppression, dépassant les 10000 valeurs.

Au contraire, la file n'a pas l'air de souffrir des mêmes symptômes.

Enfin, pour la même quantité de valeurs, la liste chaînée devient plus lente en insertion.

Pour conclure, nous pourrions affirmer qu'une structure de données adaptée à tout serait la file ou la file de priorité, tandis que si la suppression n'est pas quelque chose de récurrent, la pile devient envisageable. Et enfin, si cette fois l'insertion n'est pas quelque chose de récurrent, c'est la liste chaînée qui le devient.