POLYTECHNIQUE
MONTRÉAL

INF2010

Structures de données et algorithmes

TP4 : Heap/Monceaux

(Différence entre Binary Tree et Heap)

Automne 2023

Soumis par:

Massoud, IBRAHIM – 1538561

Nazim, BERTOUCHE – 2065601

Rapport pour la partie 2 du TP4

Voici un tableau qui représente nos données pour les temps d'exécutions (en nanosecondes) des opérations dans le cas d'un Binary Tree et d'un Heap. Nous avons pris la moyenne de 3 temps d'exécutions pour chacune des opérations. Nous avons ensuite fait 2 graphiques avec les données en ms pour mieux visualiser les résultats.

			Taille de données(n)		
		Opérations	**5000**	**10000**	**17000**
Binary Tree	**Cas moyen**	**insertion**	2000000	2766667	3000000
		suppression	1666667	2000000	3666667
		GetMin()/GetMax()	355567	552067	959400
	Pire cas	**insertion**	35333333	141000000	399000000
		suppression	41333333	139000000	412333333
		GetMin()/GetMax()	44658533	175057233	504080233
Heap	**Cas moyen**	**insertion**	691667	624100	793800
		suppression	1372800	1819567	1836067
		GetMin()/GetMax()	171533	303400	244200
	Pire cas	**insertion**	376467	702167	1037533
		suppression	490167	946067	1839833
		GetMin()/GetMax()	57833	85900	160067

Façon de faire les tests :

- ***Pour Binary Tree*** :

Dans le fichier Partie2Test.java, nous avons crée un arbre binaire non-balancé pour simuler *le scénario du pire des cas*. Nous avons donc ajouté les éléments de l'arbre en ordre croissant (voir un exemple à la figure 1 ci-après). Cela fait en sorte que la hauteur de l'arbre est de 6 et la complexité serait O(N) pour chercher l'élément 5 par exemple.

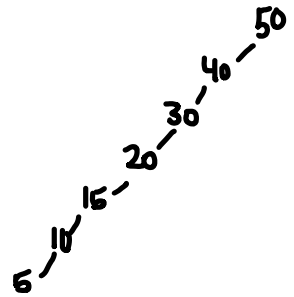

Figure 1: Arbre binaire pour le pire cas.

Pour le scénario (*cas moyen*), nous mélangeons les chiffres ajouté avec la méthode ***Collections.shuffle*** afin de pouvoir ajouter les éléments de manière plus équilibrée. Cela va créer un arbre qui ressemble à la figure 2 suivante.

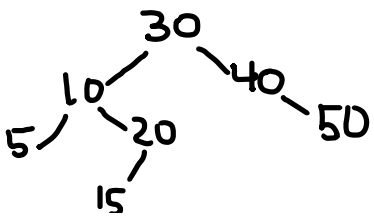

Figure 2: Arbre binaire pour le cas moyen

Par la suite, nous avons 3 différentes tailles de données (5000, 10 000 et 17 000) pour faire les tests sur les opérations d'addition, de suppressions et GetMin()/GetMax(). 17 000 était le maximum que nous avons pu y'aller due à la mémoire disponible de l'ordinateur.

Pour *l'opération de retrait*, nous faisons le retrait à partir du bas de l'arbre pour faire en sorte qu'il y a encore plus de comparaisons à faire pour simuler le pire des cas. Ensuite pour *l'opération getMax()*, étant donné que nous avons créé notre arbre binaire avec des éléments en ordre croissant, alors il est plus judicieux d'utilise la méthode ***findMax(curNode)***. La raison étant que les comparaisons vont se faire vers la droite de l'arbre. Tandis que si les nous avions un arbre avec les éléments placés en ordre décroissant, il faudrait faire les comparaisons vers la gauche avec l'opération ***findMin(curNode)*** pour arriver au pire des cas. Dans les 2 cas, le temps de complexité est le même pour un même arbre binaire.

- ***Pour Heap*** :

Dans le cas d'un Priority Queue (pour un minHeap par exemple), l'élément le plus petit sera placé à la racine de l'arbre. Alors que les éléments les plus grand vers les feuilles de l'arbre. Sachant que lors de *l'ajout* ou du *retrait* d'un élément, si l'opération est faite en ordre croissant, il y aura seulement une comparaison avec le nœud parent (cas moyen). Nous avons donc ajouté les éléments de façon aléatoire puisqu'il y aura plusieurs comparaisons pour distribuer les nœuds ajouter vers la racine de l'arbre (pour le pire des cas).

Nos observations sur la différence entre le Heap et un arbre binaire

Nous avons créé 2 graphiques aux pages 7 et 8 (**fig.3 et 4**) pour représenter les données du tableau de la page précédente.

- ***Binary Tree (pour le pire des cas)*** :

Sur la figure 3, le temps d'exécution des opérations d'insertion, de retrait et GetMin()/GetMax() pour le pire des cas pour un Binary Tree sont plus longs que les autres résultats. Pour cette raison nous avons focussé sur la partie du pire des cas pour les données reliées au Binary Tree pour mieux visualiser le graphique. Nous observons que le temps d'exécution pour l'insertion dans un Binary Tree augmente de 35,3 ms à 141 ms à 353ms (en allant de 5000 à 10 000 à 17 000 nombres d'éléments respectivement). Les résultats sont similaire pour l'opération de suppression et de GetMin()/GetMax(). Ceci fait du sens puisque plus nous augmentons le nombre d'éléments dans l'arbre binaire, plus le temps d'exécution sera élevé. De plus, pour les 3 types d'opérations, Si on observe la courbe bleu (n=5000), on remarque que le temps d'exécution est similaire pour les 3 opérations. On remarque la même chose avec la courbe verte et rouge (pour n= 10 000 et 17 000 respectivement. Ceci confirme que la complexité temporelle pour les 3 opérations est la même. Cela est justifié puisque nous avons vu dans le cours que la complexité des 3 opérations est de O(N) dans le pire des cas. Comme expliqué à la fig.1 à la page 2, la hauteur de l'arbre binaire est N ce qui fera en sorte qu'il y aura N comparaisons pour ajouter/supprimer/trouver un élément à la fin de l'arbre (puisque l'arbre est construit avec des éléments en ordre).

- ***Binary Tree (pour le cas moyen)*** :

Nous constatons que la courbe rouge est plus élevée que la courbe verte et la courbe bleu. Cela confirme que plus y a des données à traiter, plus le temps d'exécution est grand. Pour une insertion ou une suppression, le temps d'exécution est très similaire autour de 2 ms. Pour l'opération GetMin()/GetMax(), le temps d'exécution est entre 0,36 et 0,96ms.

Tableau de la complexité temporelle d'un arbre binaire BST et Heap vue au cours.

Binary Tree	Insertion	Suppression	GetMin()/GetMax()
Cas moyen	O(log N)	O(log N)	O(log N)
Pire cas	O(N)	O(N)	O(N)

Heap	Insertion	Suppression	GetMin()/GetMax()
Cas moyen	O(1)	O(log N)	O(1)
Pire cas	O(log N)	O(log N)	O(1)

Nous avons vu dans le cours que les complexités temporelles des 2 tableaux ci-dessus. Donc en comparant les résultats obtenus, ~2ms pour le cas moyen et (entre 41ms et 412ms) pour le pire des cas, ceci est ben confirmé avec les complexité O(N) et O(log N) pour le pire et cas moyen respectivement. Ceci est aussi dû à la structure de l'arbre comme expliqué à la page 2 avec la fig.2. L'arbre étant balancé, sa hauteur est log N ce qui fait en sorte que le temps d'exécution est moindre.

- ***Heap (pour le cas moyen / pire cas)*** :

Nous remarquons que le temps d'exécution pour ***l'insertion*** est très rapide (entre 0,62 et 0,79ms) pour le cas moyen et (entre 0,38 et 1,04 ms) pour le pire cas. La stratégie d'insertion « *percolate up* » est de créer un trou dans la feuille de l'arbre et de l'amener vers le haut jusqu'à ce que la condition que les nœuds enfants soient plus grands que le nœud parent soit satisfaite. Étant donné que la Heap est en ordre pour le cas moyen, l'ajout se fera très rapidement comparé au pire cas. Ceci est dû au fait qu'il n'y a pas besoin de « *percolate* » l'élément vers le haut de l'arbre. Nos résultats correspondent bien à la complexité O(1) pour le cas moyen et O(log N) pour le pire des cas.

Nous remarquons que le temps d'exécution est très court pour ***GetMin()/GetMax()***. Ils sont environ de 0,24 ms et 0,16 ms pour le cas moyen et pire cas respectivement. Selon le livre de Weiss[1], un Heap à une propriété d'ordre « *heap-order property* » qui fait en sorte que dans un min-Heap, l'élément le plus petit est toujours placé à la racine de l'arbre. (Pour un max-Heap, le maximum sera à la racine). Ceci fait en sorte que pour trouver le minimum, ça se fait en temps constant avec une complexité O(1) dans le pire et cas moyen. Cependant, si on veut trouver le

[1] Mark Allen Weiss, *Data Structures and Algorithm Analysis in Java*, Edition 3, Addison Wesley, 2012.

maximum dans un min-Heap, nous pouvons inverser le comparateur dans la fonction et avoir un max-Heap.

Pour l'opération de retrait, on remarque que le temps d'exécution est plus élevé (environ 1.84 ms) comparé aux opération d'insertion et GetMin(). Pour la suppression, nous avons utilisé la méthode « poll » dans la priority queue. Cette méthode nous permet de retirer l'élément de la racine de l'arbre. Une restructure est faite avant le prochain retrait et ceci explique pourquoi le temps est légèrement plus élevé.

Réponses aux questions :

- **Quelle(s) est/sont la/les différence(s) de les insertions et suppression de donnée?**
  - Pour un arbre binaire, une insertion se fait en faisant des comparaisons, à partir de la racine de l'arbre, de l'élément à ajouter avec les autres nœuds de l'arbre jusqu'à ce qu'il arrive à la bonne position. Lors d'une suppression, il y a aussi des comparaisons qui se font, mais l'arbre binaire est aussi réorganisé par la suite en maintenant ses propriétés. Cette réorganisation est faite de manière à garder l'ordre des éléments.
  - Pour un Heap, la stratégie d'insertion « *percolate up* » est de créer un trou dans la feuille de l'arbre et de l'amener vers le haut jusqu'à ce que la condition que les nœuds enfants soient plus grands que le nœud parent soit satisfaite dans le cas d'un min-Heap. Pour la suppression, une fois que la racine est retirée et la dernière feuille sera déplacée vers la racine. Par la suite, il y aura un « percolate down » pour maintenir les propriétés de la Heap.

- **Donner 2 cas où l'utilisation d'un heap est meilleur qu'un arbre binaire.**
  1. *Accéder au plus petit ou plus grand élément rapidement*, car cet élément se situe à la racine de la Heap. Donc si on implémente par exemple une file de tâches avec une certaine priorité, nous voulons être capable de faire la tâche avec la plus haute priorité en premier (équivalent à chercher l'élément de la racine). Ceci permet un accès rapide avec un complexité temporelle O(1). L'utilisation de la Heap est donc meilleure qu'un arbre binaire puisque ce dernier possède une plus grande complexité de O(log N) ou O(N) dans le cas moyen ou pire cas respectivement.
  2. Si nous avons un cas avec une application de gestion de tâches où c'est possible d'ajuster la priorité des tâches. L'utilisation d'un Heap va permettre de faire une insertion et de mettre à jour rapidement la priorité d'une tâche. Puisque l'arbre reste complet et l'insertion dans un Heap le permet de le faire avec une complexité O(log N) en pire des cas. Tandis que pour un arbre binaire, il pourrait ne pas être balancé. De plus, l'insertion pourrait nécessiter des réorganisations pour maintenir l'équilibre de l'arbre. Dans le pire des cas, la complexité est de O(N). Alors l'utilisation d'un Heap est plus adéquat dans ce cas.

- **Si on compare un Heap à un arbre AVL, est-ce que les différences sont encore aussi grande en termes de complexité? Expliquer vos résultats.**

  En terme de complexité, un arbre AVL offre une complexité de O(log N) pour l'ajout, le retrait ou trouver le minimum/maximum d'un élément. Et ceci est dans le pire ou le cas moyen. La raison étant que dans un arbre AVL, il y a un balancement qui est fait après chaque ajout/retrait. Ceci produit un arbre plus balancé.

  Cependant, pour un Heap, dans le pire des cas la complexité est similaire avec O(log N) pour l'insertion et la suppression d'un élément. Trouver le minimum ou le maximum se fait en temps constant O(1) dans le cas moyen et pire des cas. Dans les 2 cas, les complexités sont similaires. Cependant, dépendamment du cas d'utilisation, si on cherche le minimum ou maximum, utiliser un Heap sera plus rapide qu'un arbre AVL. Voici un tableau qui résume la complexité pour Heap et les arbres AVL :

Arbre AVL	Insertion	Suppression	GetMin()/GetMax()
Cas moyen	O(log N)	O(log N)	O(log N)
Pire cas	O(log N)	O(log N)	O(log N)
Heap	**Insertion**	**Suppression**	**GetMin()/GetMax()**
Cas moyen	O(1)	O(log N)	O(1)
Pire cas	O(log N)	O(log N)	O(1)

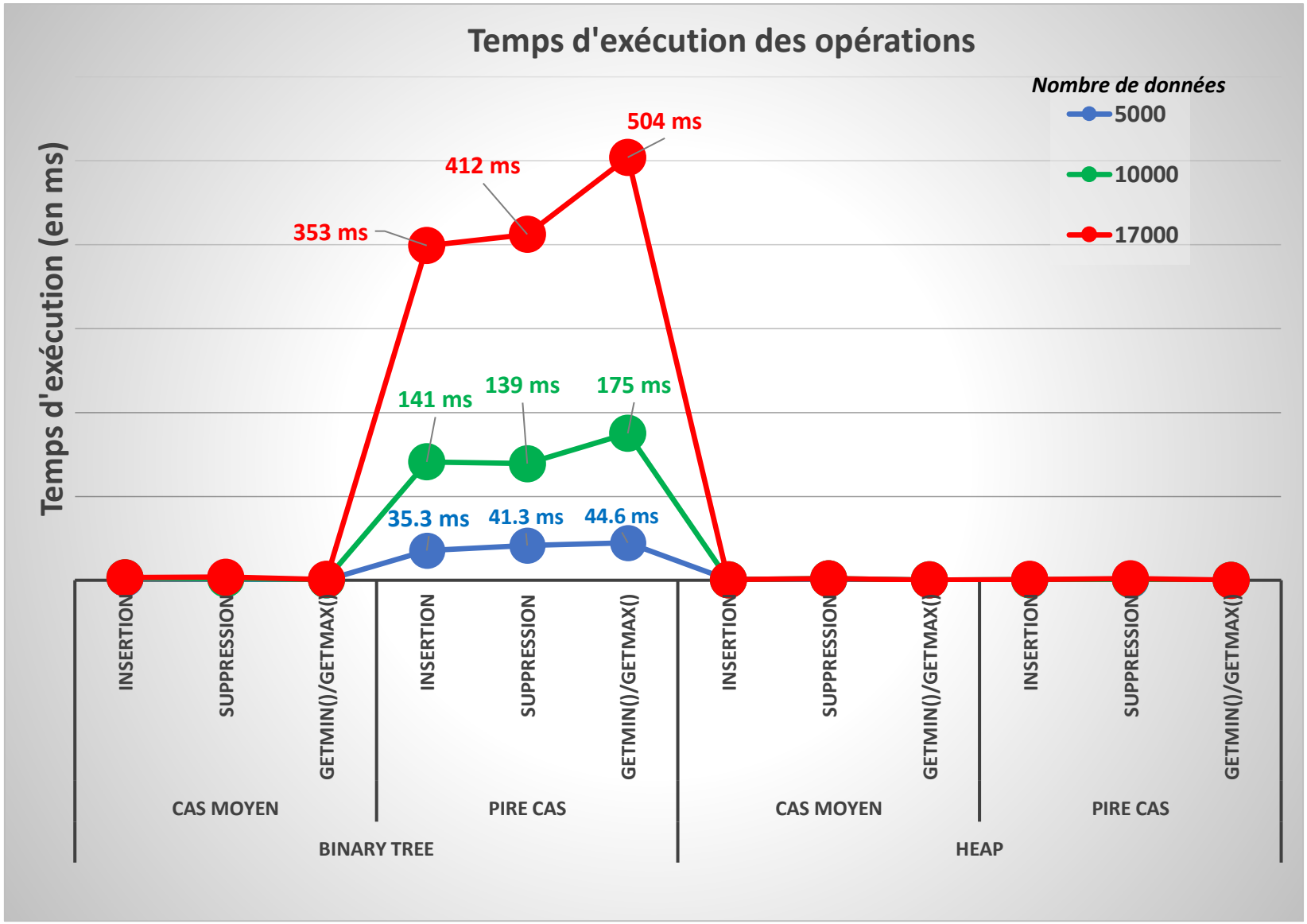

Figure 3 : Temps d'exécution des opérations pour un Binary Tree et Heap

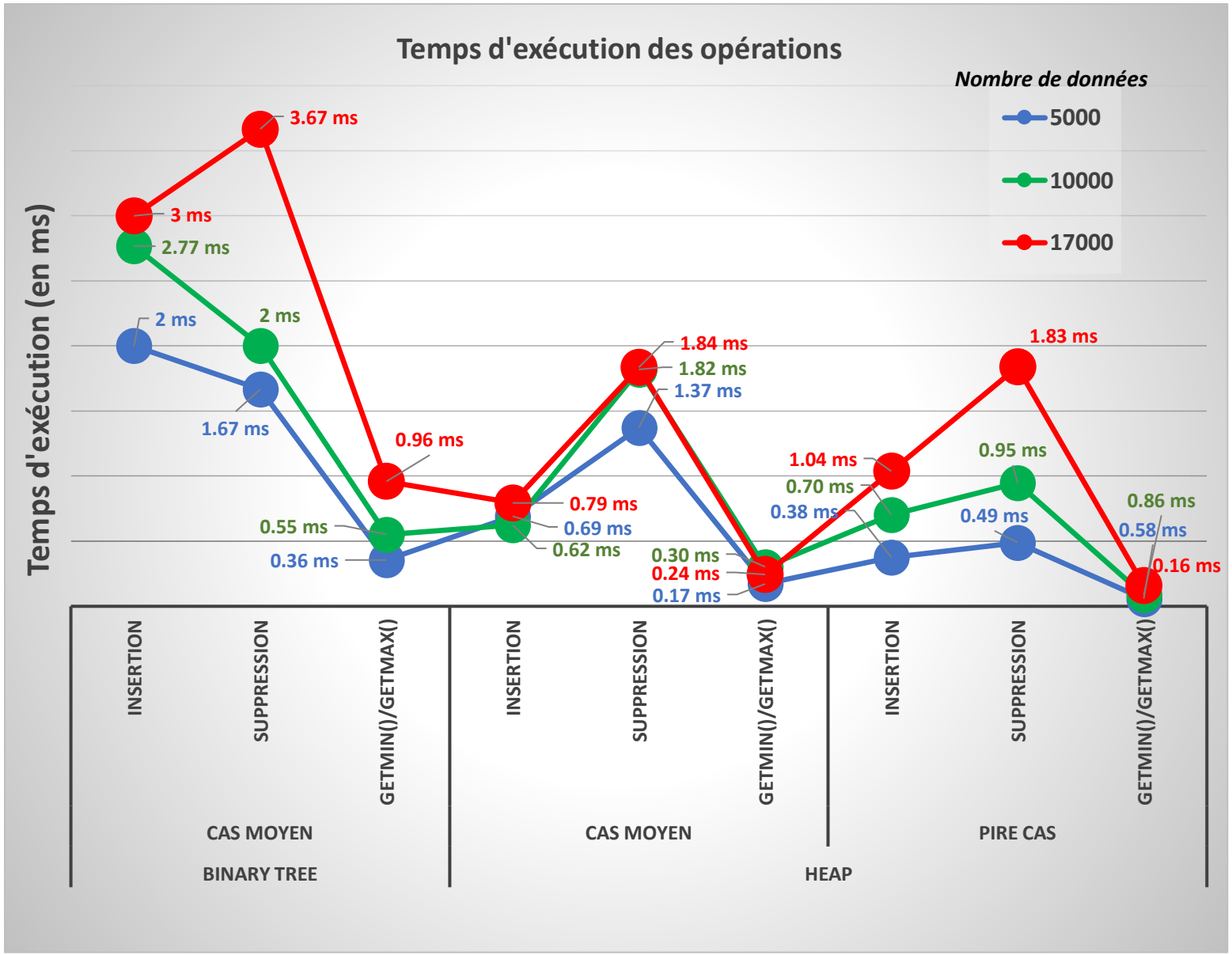

Figure 4 : Temps d'exécution des opérations pour un Binary Tree et Head