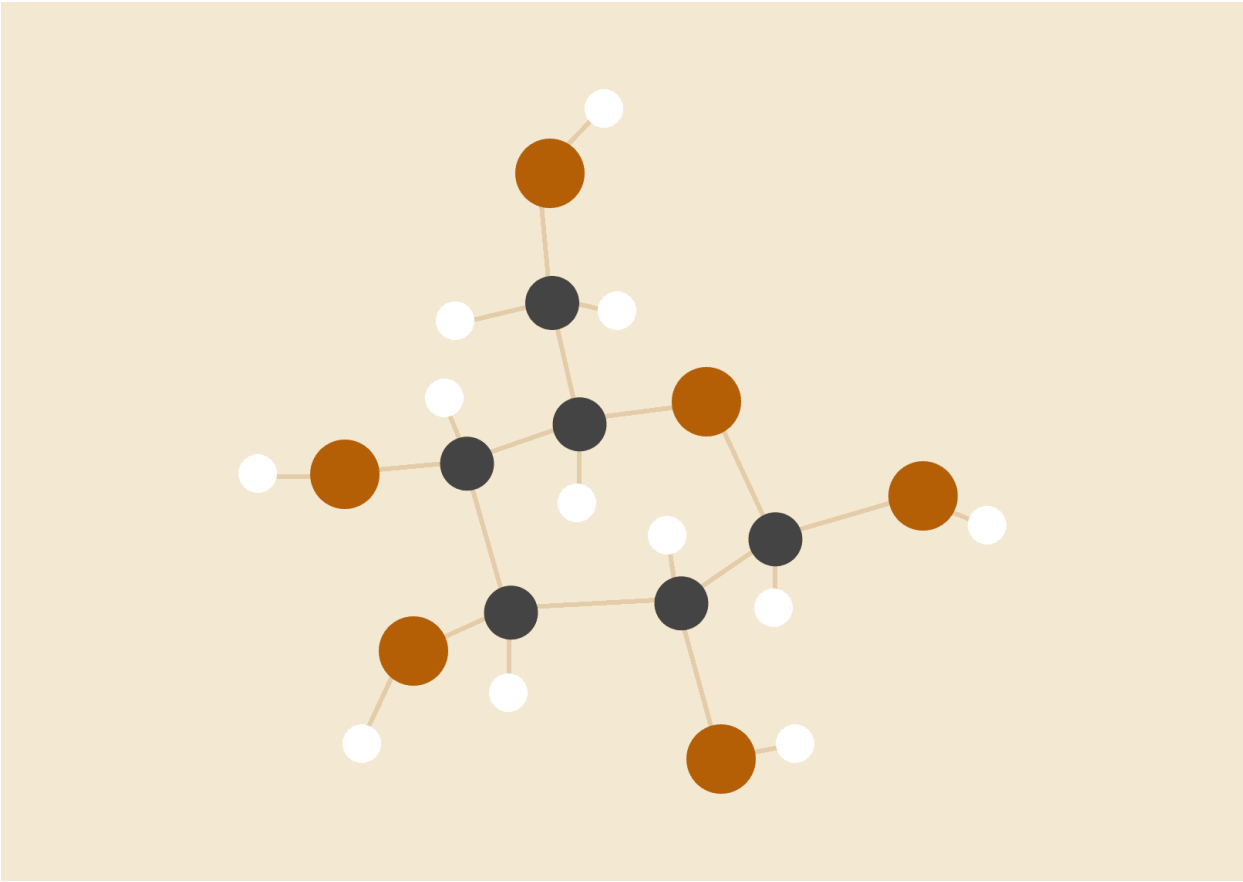
Rapport de TP
Arbres binaires de recherche et Tas binaires



Kamanda Aubin
Placé Louka

29/03/2023
UFR Sciences et Technique - L3 MIAGE
Conception & Analyse d'Algorithmes

Sommaire

I) Préambule

Les objectifs de ce TP sont (1) de manipuler deux structures de données vues en cours et TD, à savoir les arbres binaires de recherche (ABR) et les tas binaires (TB) ; (2) de vérifier, par la pratique, les complexités données pour certaines routines de base ; (3) de comparer ces complexités entre les deux structures étudiées. Le langage utilisé pour implémenter les algorithmes est Python.

II) Création d'Arbre Binaire de Recherche

a) Le résultat du parcours suffixe de l'ABR complet à n = 31 noeuds est :
[1, 3, 2, 5, 7, 4,9, 11,10, 13, 15,14, 12, 8, 17,19,18, 21, 23, 22, 20, 25, 27 , 26, 29, 31, 30, 28, 24, 16]

Le résultat du parcours suffixe de l'ABR complet à n = 63 noeuds est :
[1, 3, 2, 5, 7, 6, 4, 9, 11, 10, 13, 15, 14, 12, 8, 17, 19, 18, 21, 23, 22, 20, 25, 27, 26, 29, 31, 30, 28, 24, 16, 33, 35, 34, 37, 39, 38, 36, 41, 43, 42, 45, 47, 46, 44, 40, 49, 51, 50, 53, 55, 54, 52, 57, 59, 58, 61, 63, 62, 60, 56, 48, 32]

b) Afin de générer le tableau T qui permet de créer un ABR filiforme à n = 2^(p+1)-1 on a procédé comme suit :

1/ Tout d'abord on a fait une fonction 'creer_abr_filiforme' qui prend en paramètre p.

2/ Ensuite, on a créé un tableau rempli de 0, de longueur allant de 1 à 2^(p+1)-1 à l'aide d'une première boucle

3/ Enfin, avec une seconde boucle on a parcouru tout le tableau afin d'y ajouter à chaque élément +1 qu'à son précédent, en commençant par 1. C'est alors que l'on retourne le tableau final T, rangé dans l'ordre croissant (T=[1,2,3,4,5,6…]).

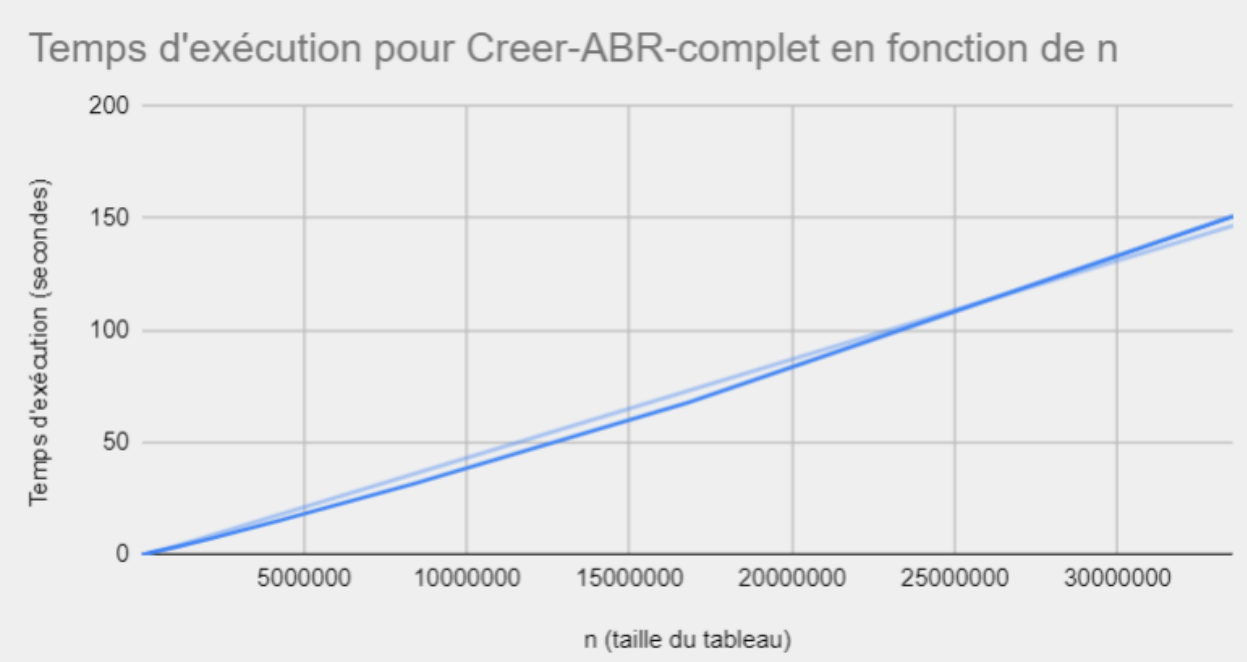
c) Dans le tableau ci-contre, on peut constater les temps d'exécutions en seconde des insertions suite aux fonctions Creer-ABR-complet et Creer-ABR-filiforme. Il est important de préciser que lors de l'exécution de l'insertion avec le tableau renvoyé par la fonction Creer-ABR-filiforme avec une valeur p en entrée supérieur à 8, on obtient cette erreur :

```
*** Remote Interpreter Reinitialized ***
Traceback (most recent call last):
  File "C:\Users\Power Media\Desktop\TP2exos1.py", line 95, in <module>
    racine2.insert(T2[i])
  File "C:\Users\Power Media\Desktop\TP2exos1.py", line 49, in insert
    self.enfant_droit.insert(valeur)
  File "C:\Users\Power Media\Desktop\TP2exos1.py", line 49, in insert
    self.enfant_droit.insert(valeur)
  File "C:\Users\Power Media\Desktop\TP2exos1.py", line 49, in insert
    self.enfant_droit.insert(valeur)
  [Previous line repeated 980 more times]
  File "C:\Users\Power Media\Desktop\TP2exos1.py", line 47, in insert
    self.enfant_droit = ArbreBinaire(valeur)
                        ^^^^^^^^^^^^^^^^^^^^
RecursionError: maximum recursion depth exceeded
>>>
```

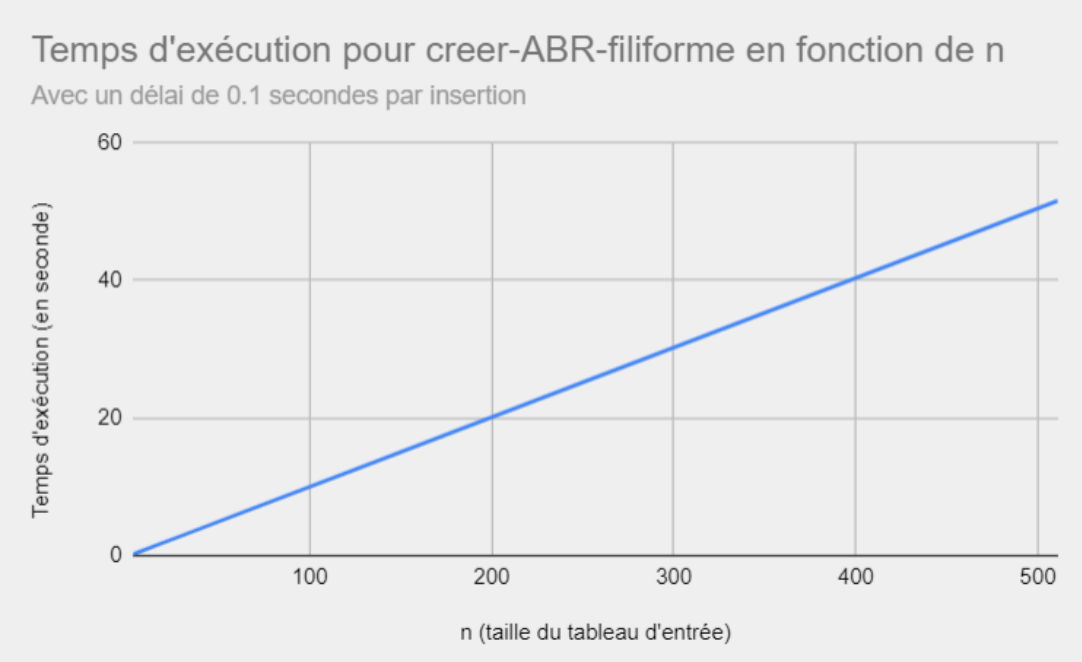
C'est pourquoi on a rajouté un délai de 0,1 seconde dans la boucle de l'insertion. C'est à dire que l'on a *(n-1) * 0.1* secondes de délai car l'insertion du premier élément n'étant pas dans notre boucle. Cela était nécessaire afin d'obtenir des valeurs n'étant pas égale à 0, ce qui était le cas avant l'ajout du délai.

Temps en secondes			
Insertion dans l'ABR		Entrée fonction	Taille du tableau
Creer-ABR-complet	Creer-ABR-filiforme	p	n
0	0,2080183029	1	3
0	0,6056070328	2	7
0	1,415968895	3	15
0	3,028066397	4	31
0	6,257048607	5	63
0	12,7066133	6	127
0,0009968280792	25,6534977	7	255
0,002316236496	51,51107645	8	511
0,00385928154	-	9	1023
0,007392644882	-	10	2047
0,01475214958	-	11	4095
0,04135298729	-	12	8191
0,09054946899	-	13	16383
0,1470286846	-	14	32767
0,2646260262	-	15	65535
0,4797365665	-	16	131071
0,990323782	-	17	262143
2,007477045	-	18	524287
3,561056137	-	19	1048575
7,251919985	-	20	2097151
15,22348452	-	21	4194303
31,82863069	-	22	8388607
67,72663093	-	23	16777215
150,6996105	-	24	33554431

Voici le graphique associé au tableau ci-dessus qui prend en compte le temps d'exécution de Creer-ABR-complet en fonction de n. La courbe bleu transparent est une courbe de tendance.



Voici le graphique associé au tableau ci-dessus qui prend en compte le temps d'exécution de Creer-ABR-filiforme en fonction de n.



d) En cours on a vu que la complexité au pire de l'insertion d'un ABR était de O(n), on constate que les courbes sont linéaires pour l'insertion de Creer-ABR-filiforme et Creer-ABR-complet. On peut en conclure que nos résultats sont corrects et cohérents à ce qu'il a été vu en cours.

Et pour la création sa complexité est en O(n*log(n)) en moyenne, dans le cas du tas

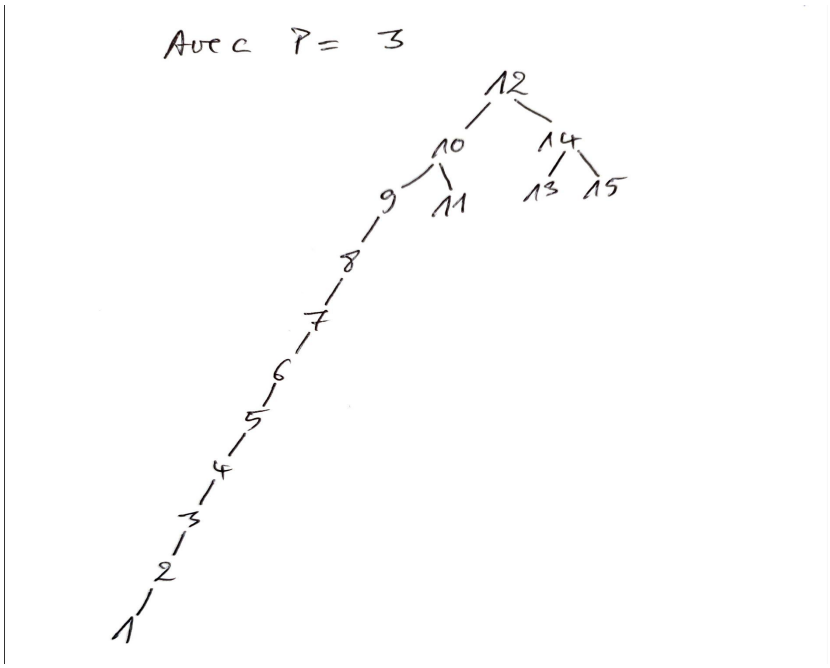
Performances comparées : tableaux, listes, ABR
Complexité au pire

Opération	TNT	TT	LNT	LT	ABR
Insertion	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Suppression	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Recherche	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
Création	$O(n)$	$O(n \log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$

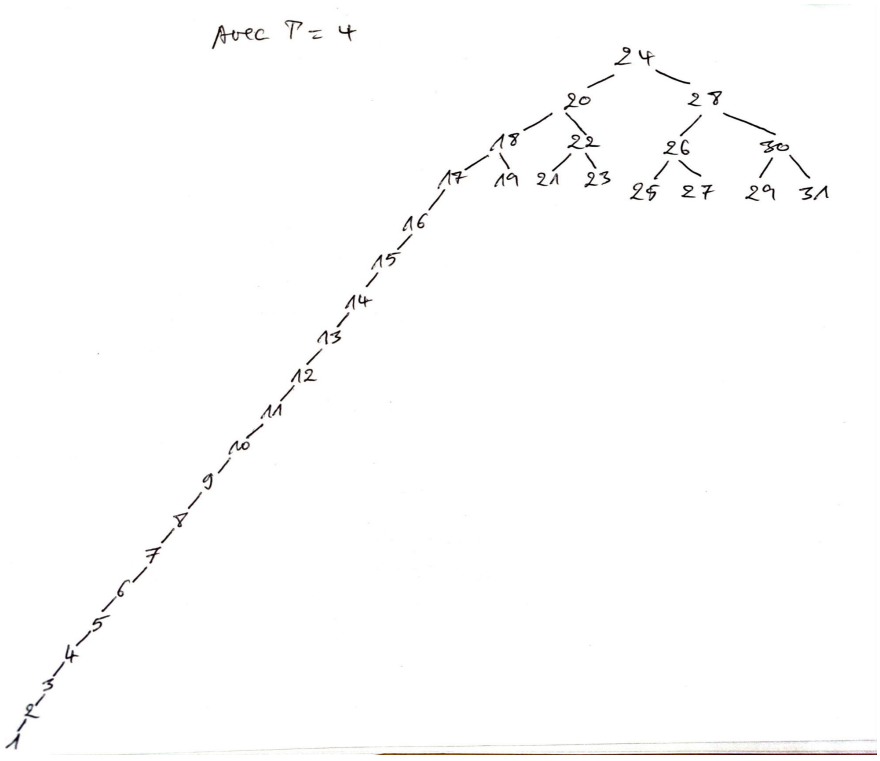
III) Suppression, insertion et recherche dans un ABR

a) Dessiner l'ABR A′ obtenu pour p = 3 et pour p = 4

Pour p = 3


Avec p = 3
12
10
14
9
11
13
15
8
7
6
5
4
3
2
1

pour p = 4


Avec p = 4
24
20
28
18
22
26
30
17
19
21
23
25
27
29
31
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1

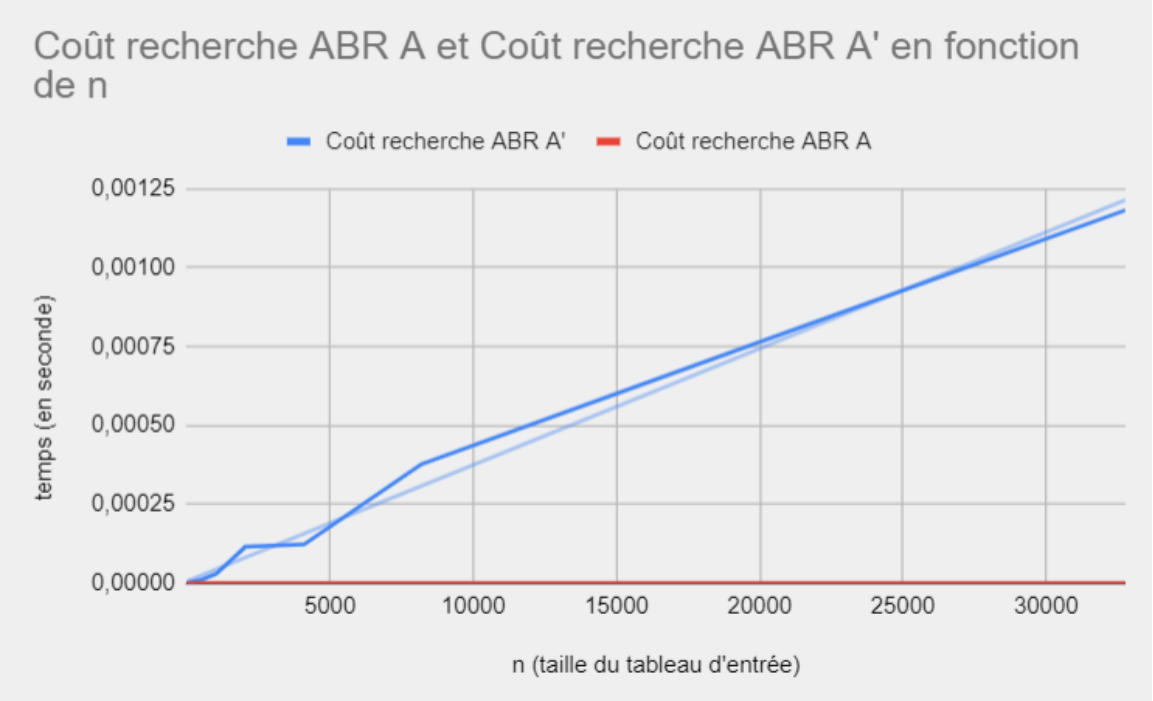
b) De façon général on aura toujours la profondeur de A’ :

profondeur A' = profondeur A + 2^p

Car pour chaque itération on supprime le sommet, puis on le rajoute directement en une feuille, ce qui allonge la longueur à chaque fois de 1. Donc au final on aura la longueur initiale plus le nombre d’itération de supprimer/insérer.

c) Dans le tableau ci-contre, on peut constater les temps d’exécutions en seconde pour la recherche de l’élément 1 dans A et A’ en fonction de n.

Temps en secondes		Entrée fonction	Taille du tableau
Coût recherche ABR A'	**Coût recherche ABR A**	**p**	**n**
0,0000009532743164	0,0000007152557373	1	4
0,000001430611475	0,0000007152557373	2	8
0,000001907258633	0,0000004768371582	3	16
0,000001431511475	0,0000007152557373	4	32
0,000002384185801	0,0000002384185791	5	64
0,000003814697266	0,0000002384185791	6	128
0,000006198893057	0,0000004768371582	7	256
0,000011205674	0,0000004768371582	8	512
0,00003051750813	0,0000004768371582	9	1024
0,000116109908	0,0000004768371582	10	2048
0,0001237392826	0,0000004768371582	11	4096
0,0003776550893	0,0000004768371582	12	8192
0,0006461144494	0,0000004768371582	13	16384
0,001182317934	0,0000007152557373	14	32768



d) En cours on a vu que la complexité au pire de la recherche dans ABR était de O(n), on constate ici que la courbe pour pour l'ABR A' est linéaire, car effectivement c'est le pire des cas. On allonge l'arbre en ajoutant beaucoup de profondeur dans le sous arbre gauche à chaque fois. Donc on va devoir aller au plus profond de l'arbre pour aller chercher le 1 car c'est la dernière valeur à être rajouter tout à la fin.

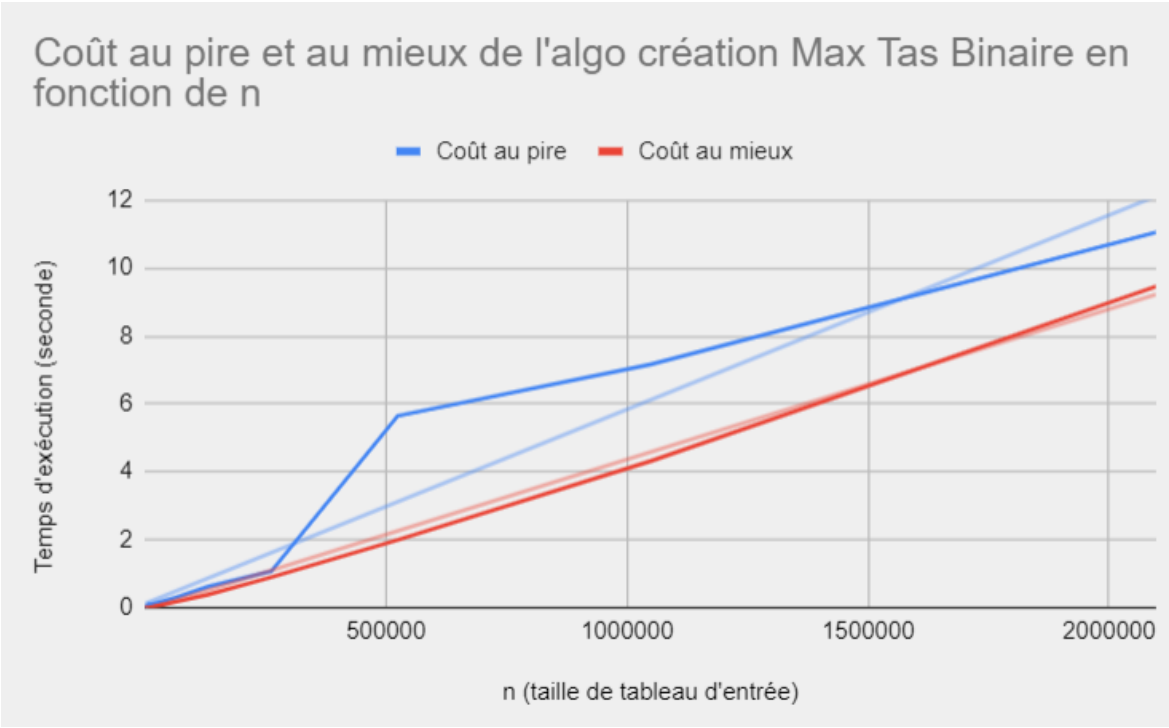
   En ce qui concerne la recherche pour l'ABR A, on a vu que moyenne c'était en O(log(n), effectivement car grâce aux propriétés de l'ABR on peut effectuer une recherche dichotomique qui elle est en O(log(n)), comme notre courbe pour sa recherche de l'élément 1.

IV) Tas Binaire et Arbre Binaire Recherche

a) Dans le tableau ci-contre, on peut voir tous nos résultats sur les temps d'exécutions au mieux et au pire pour la création d'un Max Tas Binaire ainsi que la recherche de l'élément 1 dans celui-ci en fonction de n (et p qui sont relié via la formule n = 2^(p+1)-1).

Temps en secondes				Entiers	
Algo création Max Tas Binaire		Recherche de l'élément 1		Entrée fonction	Taille du tableau
Coût au pire	Coût au mieux	Coût au pire	Coût au mieux	p	n
0	0	0	0	1	3
0	0	0	0	2	7
0	0	0	0	3	15
0	0	0	0	4	31
0,003446578979	0	0	0	5	63
0,001740217209	0	0,001000165939	0	6	127
0,01553606987	0	0,0009963512421	0	7	255
0,01112985611	0	0,001000165939	0	8	511
0,01829266548	0	0,0009973049164	0	9	1023
0,03606653214	0	0,01562380791	0	10	2047
0,03735899925	0	0,00484418869	0	11	4095
0,1010782719	0,00982093811	0,04340934753	0	12	8191
0,1119098663	0,03231215477	0,01623272896	0	13	16383
0,1656489372	0,07810354233	0,01694321632	0	14	32767
0,3015253544	0,1750228405	0,01693224907	0	15	65535
0,6154720783	0,3785223961	0,02040410042	0	16	131071
1,066222668	0,8948726654	0,04009652138	0	17	262143
5,645759344	1,992353201	0,1227986813	0,0312461853	18	524287
7,164779425	4,311563492	0,1502826214	0,06368350983	19	1048575
11,05443692	9,460172892	0,5140659809	0,158621788	20	2097151

Voici le graphique associé au tableau ci-dessus qui prend en compte le temps d'exécution au mieux et au pire de de la création d'un Max Tas binaire, *Creer-TB* en fonction de *n*. Les valeurs au mieux et au pire ont été déterminées à l'aide d'une boucle qui faisait 50 essais.Les courbes de tendances sont respectivement la transparent de sa courbe originale.

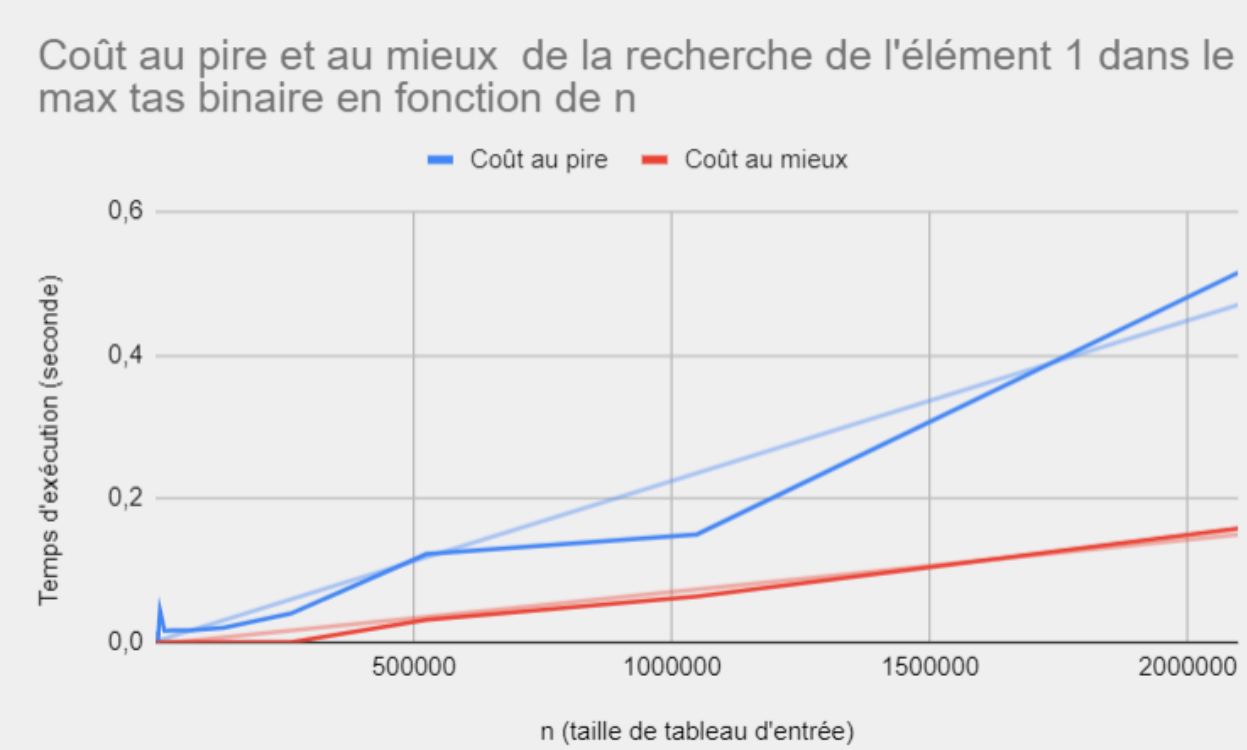


En cours, on a vu que la création d'un Tas-Binaire était au pire et au mieux en O(n) donc Θ(n). On le constate sur nos courbes qui sont elles linéaires, les courbes de tendances sont présentes afin d'accentuer leur linéarité car malheureusement nos résultats ne peuvent être sur à 100% dû à des facteurs externes comme la performance de l'ordinateur.

$$\Rightarrow \texttt{Creer-TB est en } O(n) \text{ au pire – et donc en } \Theta(n)$$

b) On constate que les performances du tas binaire sont meilleurs pour la création de celui- ci comparé à la création d'un arbre binaire de recherche qui lui est en O(n*log(n))

c) Voici le graphique associé au tableau ci-dessus qui prend en compte le temps d'exécution au mieux et au pire de de la recherche de l'élément 1 dans un Max Tas binaire en fonction de *n*. Les valeurs au mieux et au pire ont été déterminées à l'aide d'une boucle qui faisait 50 essais.Les courbes de tendances sont respectivement la transparent de sa courbe originale.



En cours, on a vu que la recherche dans un Tas-Binaire était au pire en O(n). On le constate sur nos courbes qui sont elles linéaires, les courbes de tendances sont présentes afin d'accentuer leur linéarité car malheureusement nos résultats ne peuvent être sur à 100% dû à des facteurs externes comme la performance de l'ordinateur.

Performances comparées

Complexité au pire

Opération	TNT	TT	LNT	LT	ABR	TB
Insertion	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$
Suppression	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
Recherche	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Création	$O(n)$	$O(n \log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n)$

d) On constate que pour la recherche d'un élément dans les deux structures de données ABR et TB sont identiques pour leur complexité au pire. Et en moyenne elles sont encore une fois identique O(log n)

V) Conclusion

En conclusion, le tas binaire et l'arbre binaire de recherche ont des différences significatives en termes de complexité pour leurs fonctions de création et de recherche.

Pour la création, le tas binaire a une complexité de O(n) dans le pire des cas, où n est le nombre d'éléments à insérer, car chaque élément doit être inséré un par un dans l'arbre et réorganisé pour maintenir la propriété de tas. En revanche, l'arbre binaire de recherche a une complexité de O(n log n) dans le pire des cas, car l'insertion d'un élément peut nécessiter de parcourir l'ensemble de l'arbre pour trouver l'emplacement approprié.

Pour la recherche, le tas binaire a une complexité de O(log n) en moyenne, où n est le nombre d'éléments dans le tas, car la recherche se fait par une recherche binaire dans l'arbre. En revanche, l'arbre binaire de recherche a une complexité de O(log n) dans le pire des cas pour la recherche, car la recherche se fait également par une recherche binaire, mais avec une garantie que chaque sous-arbre est équilibré.

En somme, le choix entre le tas binaire et l'arbre binaire de recherche dépend des besoins spécifiques de la tâche à accomplir. Si la priorité est d'avoir une insertion rapide des éléments, le tas binaire est préférable. Si la priorité est d'avoir une recherche rapide des éléments, l'arbre binaire de recherche est le meilleur choix.