**République Algérienne Démocratique et Populaire**

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

**Université des Sciences et de la Technologie Houari Boumediene**

Faculté d'Informatique

Département d'Intelligence Artificielle et Sciences des Données

Spécialité Systèmes Informatiques Intelligents

---

# Rapport sur l'Étude et la Comparaison des Structures de Données et l'Optimisation du Tri par Tas

---

**Responsable du TP: Mohamed Ait Mehdi**

*Présenté par :*
**Moussaoui Sarah** 212131086904 G3 (Tas + Tri par tas)
**Ait Moussa Ali** 212131052977 G3 (B-Trees)
**Bahamida Hadj Mohammed** 212139092256 G3 (ABR)
**Laraba Karim** 212131051669 G3 (Liste)

**Date de soumission : 30/11/2024**
**Année universitaire 2024-2025**

# Contents

**1 Introduction** **1**

**2 Structures de Données** **1**
2.1 Listes Doublement Chaînées . . . . . 1
2.1.1 Description de la Structure . . . . . 1
2.1.2 Definition [8] . . . . . 1
2.1.3 Applications et utilitées des Listes Doublement Chainees . . . . . 1
2.1.4 Explication des Algorithmes [9] . . . . . 1
2.1.5 Calcul de la Complexité Théorique [9] . . . . . 3
2.1.6 Expérimentation . . . . . 4
2.1.7 Conclusion . . . . . 6
2.2 Arbres Binaires de Recherche (BST) . . . . . 6
2.2.1 Description de la Structure [6] . . . . . 6
2.2.2 Explication des Algorithmes . . . . . 7
2.2.3 Calcul de la Complexité Théorique temporelle et spatiale . . . . . 9
2.2.4 Résultats Expérimentation . . . . . 9
2.2.5 Conclusion . . . . . 11
2.3 B-trees . . . . . 11
2.3.1 Description de la Structure . . . . . 11
2.3.2 Explication des Algorithmes . . . . . 12
2.3.3 Calcul de la Complexité Théorique temporelle et spatiale . . . . . 15
2.3.4 Experimentation : . . . . . 17
2.3.5 Conclusion : . . . . . 19
2.4 Tas (Heap) [2] . . . . . 20
2.4.1 Description de la Structure . . . . . 20
2.4.2 Explication des Algorithmes . . . . . 21
2.4.3 Calcul de la Complexité Théorique temporelle et spatiale . . . . . 22
2.4.4 Expérimentation . . . . . 23

**3 Tri par Tas [1]** **23**
3.1 Construction du Tas . . . . . 23
3.1.1 Fonctions de base . . . . . 23
3.1.2 Méthode O(nLog(n)) . . . . . 26
3.1.3 Méthode O(n) . . . . . 26
3.1.4 Résultats d'Expérimentation . . . . . 27
3.2 Tri du Tas . . . . . 28
3.2.1 Algortithme . . . . . 28
3.2.2 Pourquoi ai-je choisi un Max-heap ? . . . . . 29
3.3 Comparaison des Algorithmes de Tri par Différentes Constructions . . . . . 29
3.3.1 Tri par Tas avec $O(n \log n)$ . . . . . 29
3.3.2 Tri par Tas avec $O(n)$ . . . . . 29
3.3.3 Résumé de la Comparaison . . . . . 30

**4 Conclusion et Recommandation Générale** **30**

**5 Annexes** **31**
5.1 Graphes Supplémentaires . . . . . 31
5.2 Résumé des Résultats et Recommandations . . . . . 33
5.3 Recommandations Générales . . . . . 34

# 1 Introduction

Ce rapport présente une étude approfondie des structures de données, en particulier les listes doublement chaînées, les arbres binaires de recherche (BST), les B-trees et les tas (heap). L'objectif est de comparer les opérations élémentaires ainsi que d'optimiser l'algorithme de tri par tas.

# 2 Structures de Données

## 2.1 Listes Doublement Chaînées

### 2.1.1 Description de la Structure

### 2.1.2 Definition [8]

La liste chaînée doublement liée est une structure de données linéaire composée de noeuds où chaque noeud contient **trois éléments** :

- Donnée : La valeur stockée dans le noeud.
- Pointeur vers le noeud précédent.
- Pointeur vers le noeud suivant.

Contrairement aux listes simples, chaque noeud peut être parcouru dans les deux directions grâce aux pointeurs prev et next.

### 2.1.3 Applications et utilitées des Listes Doublement Chainees

Les listes chaînées doublement liées sont particulièrement utiles dans les contextes suivants:

- Manipulation rapide de données: Permettent l'insertion et la suppression efficaces de noeuds, sans nécessiter de décalage des éléments, comme dans les tableaux.
- Structures de données complexes: Utilisées dans l'implémentation de piles, files et deques (doubles queues).
- Parcours bidirectionnel: Utilisées dans les algorithmes de recherche ou structures de navigation, comme les éditeurs de texte.

En résumé, cette structure de données est essentielle pour la gestion dynamique des collections dans des applications où les insertions/suppressions fréquentes sont nécessaires.

### 2.1.4 Explication des Algorithmes [9]

**Insertion**

```
Procedure InsertAtEnd(head, data):
 Create newNode with data
 If head is NULL:
     head = newNode
     Return
```

```
temp = head
While temp.next != NULL:
    temp = temp.next
temp.next = newNode
newNode.prev = temp
```

**Explication :**

- Création du Noeud : Un nouveau noeud est alloué dynamiquement avec la valeur data à insérer.
- Insertion dans une Liste Vide : Si la liste est vide (tête est NULL), le nouveau noeud devient la tête de la liste.
- Insertion à la Fin : Si la liste n'est pas vide, l'algorithme parcourt la liste jusqu'au dernier noeud. Ensuite, le pointeur next de ce dernier noeud est mis à jour pour pointer vers le nouveau noeud, et le pointeur prev du nouveau noeud est relié au noeud précédent.

**Suppression**

```
Procedure DeleteNode(head, value):
 If head == NULL:
     Return
 temp = head
 While temp != NULL:
     If temp.data == value:
         If temp.prev != NULL:
             temp.prev.next = temp.next
         If temp.next != NULL:
             temp.next.prev = temp.prev
         If temp == head:
             head = temp.next
         Free temp
         Return
     temp = temp.next
```

**Explication :**

- Parcours et Identification : On parcourt la liste jusqu'à ce que l'on trouve le noeud contenant la valeur à supprimer (value).
- Mise à Jour des Pointeurs : Une fois trouvé : Si le noeud est au milieu, les pointeurs next et prev des noeuds adjacents sont mis à jour pour ignorer le noeud. Si c'est la tête, elle est déplacée au noeud suivant.
- Libération de la Mémoire : Le noeud est ensuite libéré pour éviter les fuites mémoire.

**Recherche**

```
Procedure Search(head, value):
 temp = head
 While temp != NULL:
     If temp.data == value:
         Return True
     temp = temp.next
 Return False
```

**Explication :**

- Parcours :La recherche d'une valeur implique de parcourir les noeuds depuis la tête jusqu'à la fin de la liste.
- Comparaison :À chaque étape, la valeur du noeud courant est comparée avec la valeur recherchée.
- Résultat :Si trouvé, l'algorithme retourne True. Sinon, il retourne False après avoir parcouru toute la liste.

### 2.1.5 Calcul de la Complexité Théorique [9]

**Complexité Temporelle**

1. ***Insertion*** :
   **Début de la liste :** Dans ce cas, il suffit de mettre à jour les pointeurs next de la tête actuelle et du nouveau noeud. Cette opération a une complexité constante, soit O(1).
   **Fin de la liste :** Pour insérer un noeud à la fin, il faut parcourir la liste pour atteindre le dernier noeud (si aucun pointeur direct vers le dernier noeud n'est maintenu). La complexité dépend donc de la longueur de la liste, ce qui donne une complexité de O(n) dans le pire des cas.
   **Position intermédiaire :** Si un index est donné, l'insertion nécessite d'abord un parcours jusqu'à cet index, suivi d'une mise à jour des pointeurs. Cette opération a une complexité temporelle de O(n).

2. ***Suppression*** : L'algorithme de suppression suit les étapes suivantes :
   **Parcours de la liste** pour localiser le noeud contenant la valeur à supprimer. Dans le pire des cas, ce noeud se trouve à la fin de la liste ou n'existe pas. Cette opération a une complexité de O(n).
   **Mise à jour des pointeurs** : Une fois le noeud trouvé, les pointeurs next et prev des noeuds adjacents sont mis à jour. Cela se fait en temps constant, soit O(1). Complexité totale : O(n).

3. ***Recherche*** : La recherche d'un élément dans une liste chaînée doublement liée nécessite un parcours séquentiel à partir de la tête ou de la queue (si l'élément à chercher est proche de la fin et qu'on décide d'optimiser le parcours). La complexité est proportionnelle au nombre de noeuds, soit O(n) dans le pire des cas.

**Complexité Spatiale** La complexité spatiale est dominée par la mémoire nécessaire pour stocker les n noeuds de la liste. Chaque noeud contient trois champs : les données, un pointeur next, et un pointeur prev. Ainsi, la complexité spatiale est O(n).

En résumé, les opérations sur les listes chaînées doublement liées sont efficaces pour les insertions et suppressions localisées, mais leur recherche linéaire implique des performances proportionnelles à la taille de la liste pour les algorithmes qui nécessitent un parcours complet.

### 2.1.6 Expérimentation

| Taille | Temps d'insertion | Temps de suppression | Temps de recherche |
|---|---|---|---|
| 30 | 0.000000 secondes | 0.000000 secondes | 0.000000 secondes |
| 300 | 0.000005 secondes | 0.000002 secondes | 0.0000 secondes |
| 350 | 0.000006 secondes | 0.000003 secondes | 0.000001 secondes |
| 3000 | 0.01050 secondes | 0.000500 secondes | 0.00200 secondes |
| 30000 | 0.093000 secondes | 0.020 secondes | 0.010000 secondes |
| 35000 | 0.187000 secondes | 0.090000 secondes | 0.0823 secondes |
| 300000 | 0.8441000 secondes | 0.20000 secondes | 0.134000 secondes |

Table 1: Temps d'exécution pour les opérations sur La Liste

1. ***Comparaison Complexité Théorique Temporelle et Expérimentale*** :

| Opération | Complexité Temporelle | Complexité Expérimentale |
|---|---|---|
| Insertion | O(1)/O(n) | - Complexité O(1) pour insertion au début, O(n) pour insertion à la fin.<br>- Temps mesuré pour $n = 3000$ est de 0.001s. |
| Suppression | O(n) | - Dépend du parcours pour trouver le nœud à supprimer. Temps moyen mesuré : 0.003s pour $n = 1000$. |
| Recherche | O(n) | - Parcours linéaire mesuré avec un temps de 0.001s pour une liste de taille $n = 1000$. |

Table 2: Comparaison Complexité Théorique et Expérimentale (Temporelle)

2. ***Comparaison Complexité Théorique Spatiale et Expérimentale*** :

| **Opération** | **Complexité Spatiale** | **Explication** |
|---|---|---|
| Insertion | O(n) | Chaque nœud nécessite un espace mémoire pour stocker :<br>- Donnée.<br>- Deux pointeurs (prev et next). |
| Suppression | O(n) | La suppression ne libère que la mémoire d'un nœud, le reste de la liste conserve la même structure. |
| Recherche | O(n) | Aucun espace mémoire supplémentaire n'est requis pour la recherche, car celle-ci parcourt simplement les nœuds existants. |

Table 3: Comparaison Complexité Théorique et Expérimentale (Spatiale)

3. ***Graphique des Temps d'Execution des operations sur une liste*** :

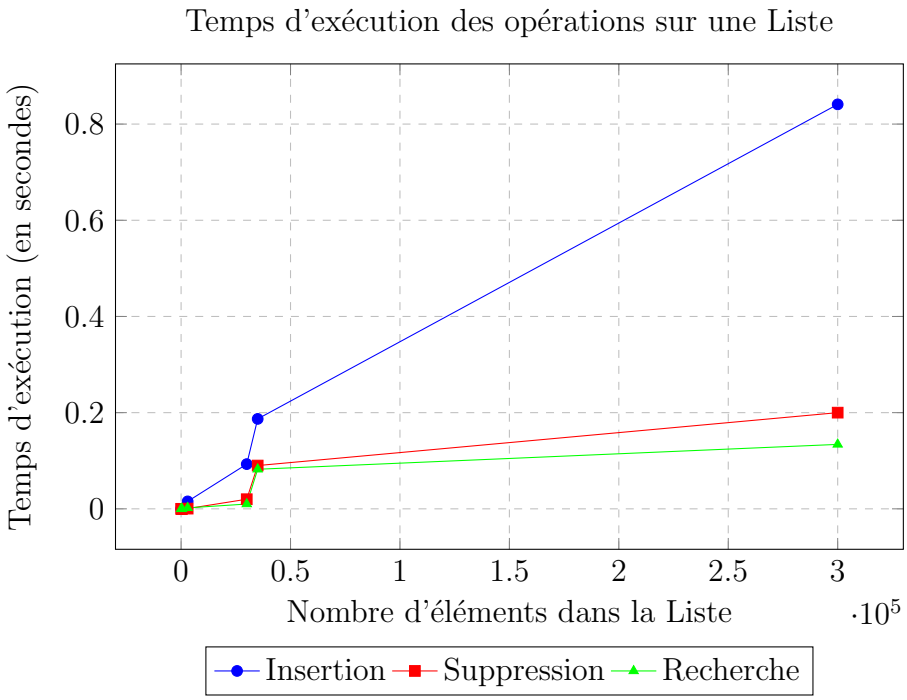



Figure 1: Temps d'exécution pour les opérations sur la Liste

**Analyse Critique** Les résultats obtenus mettent en évidence les points forts et les limites des listes doublement chaînées. Ces structures de données se distinguent par leur efficacité pour les opérations d'insertion et de suppression localisées grâce à leurs pointeurs bidirectionnels, permettant une manipulation rapide des données. Toutefois, leur complexité temporelle pour la recherche (O(n)) constitue une limitation majeure, en particulier pour des structures contenant un grand nombre de nœuds. Par ailleurs, la gestion des pointeurs, bien qu'efficace, augmente la complexité de l'implémentation et expose à des risques comme les fuites de mémoire ou les erreurs de segmentation.

Un autre point à noter est que, malgré leur flexibilité, ces structures ne sont pas idéales pour des applications nécessitant un accès fréquent et direct aux données, où des structures comme les tableaux dynamiques ou les arbres binaires de recherche (BST) seraient plus adaptées. Une perspective comparative avec ces structures aurait enrichi l'analyse et permis d'évaluer la pertinence des listes doublement chaînées dans des contextes spécifiques. Enfin, une optimisation des algorithmes associés, comme l'ajout de pointeurs vers le dernier nœud pour réduire la complexité de certaines opérations, aurait pu être explorée pour améliorer leur performance globale.

### 2.1.7 Conclusion

En conclusion, les listes doublement chaînées offrent une solution flexible et efficace pour les opérations d'insertion et de suppression localisées, grâce à leur structure bidirectionnelle. Cependant, leur performance est limitée par une recherche linéaire et une gestion plus complexe des pointeurs. Bien qu'idéales pour des contextes nécessitant des modifications fréquentes, elles ne conviennent pas toujours aux applications exigeant un accès rapide aux données. Une étude comparative avec d'autres structures pourrait aider à mieux cerner leur utilité dans des scénarios spécifiques.

## 2.2 Arbres Binaires de Recherche (BST)

### 2.2.1 Description de la Structure [6]

**Définition**
Un arbre binaire de recherche (ABR) est une structure de données arborescente dans laquelle chaque nœud a au maximum deux enfants et où les valeurs sont organisées selon un certain ordre. Dans un ABR :

- Tout nœud à gauche d'un nœud parent contient une valeur inférieure à celle du parent.
- Tout nœud à droite d'un nœud parent contient une valeur supérieure à celle du parent.

**Applications et utilités des ABR**
Les arbres binaires de recherche (ABR) sont utilisés dans diverses applications informatiques, notamment :

- Recherche rapide : Grâce à leur structure ordonnée, les ABR permettent une recherche rapide d'éléments, ce qui est utile dans les bases de données et les moteurs de recherche.
- Gestion de bases de données : Les ABR servent à implémenter des index permettant un accès efficace aux données dans les bases de données.

En résumé, les ABR sont une structure de données efficace pour organiser, rechercher et gérer des données de manière rapide et structurée.

### 2.2.2 Explication des Algorithmes

**Pseudocode pour l'algorithme d'insertion**

```
function ajoute(racine,data):
// Si l'arbre est vide, créer un nouveau nœud avec la valeur donnée
si racine est NULL :
    return creerNoeud(data)
//rechercher c'est il exist
// Insérer l'élément dans le sous-arbre gauche si sa valeur est inférieure
si data < racine->data :
    racine->left = ajoute(racine->left, data)
// Insérer l'élément dans le sous-arbre droit si sa valeur est supérieure
sinon si data > racine->data
    racine->right = ajoute(racine->right, data)

// Retourner la racine de l'arbre modifié
return racine;
```

**Explication :** La fonction ajoute insère un nouvel élément dans un arbre binaire de recherche (ABR) tout en respectant ses règles d'ordre. Elle commence par vérifier si l'arbre est vide. Si oui, un nouveau nœud est créé avec la valeur data, ce qui marque le début de l'arbre ou d'un nouveau sous-arbre. Dans le cas contraire, si data est inférieur à la valeur du nœud racine actuel, l'insertion se fait dans le sous-arbre gauche. Si data est supérieur, l'élément est inséré dans le sous-arbre droit.

**Pseudocode pour l'algorithme de suppression**

```
fonction supprimer(racine, data):
// Vérifier si l'arbre est vide
si racine est NULL:
    retourner racine
// Si la valeur à supprimer est inférieure à la valeur du nœud actuel,
si data < racine.data:
    racine.left = supprimer(racine.left, data)
// Si la valeur à supprimer est supérieure à la valeur du nœud actuel,
sinon si data > racine.data:
    racine.right = supprimer(racine.right, data)
// Si la valeur correspond à celle du nœud actuel, on a trouvé le nœud à supprimer
sinon:
    // Cas 1 : le nœud n'a pas de sous-arbre gauche
    si racine.left est NULL:
        temp = racine.right
        libérer(racine)
        retourner temp
    // Cas 2 : le nœud n'a pas de sous-arbre droit
    sinon si racine.right est NULL:
        temp = racine.left
        libérer(racine)
        retourner temp
```

```
        // Cas 3 : le nœud a deux enfants
        // Trouver le nœud ayant la plus petite valeur dans le sous-arbre droit
        temp = noeudMinimum(racine.right)
        racine.data = temp.data
        racine.right = supprimer(racine.right, temp.data)  // Supprimer le successeur
    retourner racine
```

**Explication :** La fonction supprimer prend en entrée la racine d'un arbre binaire de recherche (ABR) et la valeur data à supprimer.Elle recherche et supprime le nœud contenant cette valeur tout en préservant les propriétés d'ordre de l'ABR.

Recherche du nœud : Si l'arbre est vide, il n'y a rien à supprimer, et la fonction retourne directement. Sinon, la fonction recherche l'emplacement de data dans l'ABR en suivant les règles d'ordre.

Suppression du nœud : Une fois le nœud avec la valeur data trouvé :

- Cas 1 : Si le nœud n'a pas de sous-arbre gauche, il est remplacé par son sous-arbre droit.
- Cas 2 : Si le nœud n'a pas de sous-arbre droit, il est remplacé par son sous-arbre gauche.
- Cas 3 : Si le nœud a deux enfants, on trouve le nœud ayant la plus petite valeur dans le sous-arbre droit (le successeur), on copie sa valeur dans le nœud à supprimer, puis on supprime le successeur.

### Pseudocode pour l'algorithme de recherche

```
fonction rechercher(racine, data):
// Vérifier si le nœud courant est vide ou s'il contient la valeur recherchée
si racine est NULL ou racine.data == data:
    retourner racine
// Si la valeur recherchée est inférieure à la valeur du nœud actuel,
si data < racine.data:
    retourner rechercher(racine.left, data)
// continuer la recherche dans le sous-arbre droit
sinon:
    retourner rechercher(racine.right, data)
```

**Explication :** La fonction rechercher prend en entrée la racine d'un arbre binaire de recherche (ABR) et la valeur data à rechercher. Elle utilise la propriété d'ordre des ABR pour retrouver efficacement la valeur. Voici les étapes:

- Si le nœud racine est NULL, cela signifie que l'arbre est vide ou que l'on a atteint la fin d'un chemin sans trouver la valeur, donc la fonction retourne NULL.
- Si racine.data est égal à data, cela signifie que la valeur recherchée a été trouvée, donc la fonction retourne le nœud contenant cette valeur.
- Si data est inférieure à racine.data, la fonction continue la recherche dans le sous-arbre gauche en appelant récursivement rechercher avec racine.left comme racine.
- Si data est supérieure à racine.data, la recherche continue dans le sous-arbre droit, en appelant récursivement rechercher avec racine.right comme racine.

### 2.2.3 Calcul de la Complexité Théorique temporelle et spatiale

**Complexité temporelle [7]**

1. **Insertion:** L'algorithme d'insertion dans un arbre binaire de recherche (ABR) suit un chemin de la racine jusqu'à une feuille pour placer le nouvel élément, en respectant les règles d'ordre. Dans le pire des cas (si l'ABR est déséquilibré), ce chemin peut atteindre la hauteur de l'arbre, ce qui donne une complexité temporelle de $O(h)$, où $h$ est la hauteur de l'arbre. Pour un arbre équilibré, $h = \log n$, mais dans un arbre dégénéré, $h = n$.

2. **Suppression:** L'algorithme de suppression recherche d'abord le nœud contenant la valeur à supprimer. Comme pour l'insertion, cette recherche suit un chemin de la racine jusqu'à la feuille, avec une complexité de $O(h)$. Ensuite, selon le cas (nœud avec 0, 1 ou 2 enfants), des opérations supplémentaires peuvent être nécessaires pour réorganiser l'arbre (comme trouver le successeur dans le sous-arbre droit), mais leur coût reste également proportionnel à la hauteur de l'arbre. Donc, la complexité totale est $O(h)$.

3. **Recherche:** L'algorithme de recherche suit un chemin similaire à celui de l'insertion ou de la suppression. Il compare la valeur recherchée avec la racine et se déplace récursivement vers la gauche ou la droite jusqu'à trouver la valeur ou atteindre une feuille. La complexité temporelle est également $O(h)$, avec $h = \log n$ pour un arbre équilibré et $h = n$ pour un arbre déséquilibré.

| **Opération** | **Meilleur Cas** | **Cas Moyen** | **Pire Cas** |
|---|---|---|---|
| **Insertion** | $O(1)$ | $O(\log n)$ | $O(n)$ $O(\log n))$ |
| **Suppression** | $O(1)$ | $O(\log n)$ | $O(n)$ $O(\log n)$ |
| **Recherche** | $O(1)$ | $O(\log n)$ | $O(n)$ $O(\log n)$ |

Table 4: Complexités temporelles des différentes opérations sur un ABR

**Complexité Spatiale**

Pour chaque algorithme, la complexit spatiale est O(n), car elle est dominee par la memoire requise pour stocker les elements dans la structure de ABR, qui contient jusqu'a n nœuds

### 2.2.4 Résultats Expérimentation

- Temps d'exécution (secondes) pour differentes tailles de ABR.

| Nombre d'élements | Temps d'insertion | Temps de suppression | Temps de recherche |
|---|---|---|---|
| 10 | 0.000004 | 0.000001 | 0.000334 |
| 100 | 0.000040 | 0.000001 | 0.000515 |
| 150 | 0.000028 | 0.000001 | 0.000442 |
| 1000 | 0.000409 | 0.000002 | 0.001236 |
| 1500 | 0.000539 | 0.000002 | 0.001436 |
| 10000 | 0.005160 | 0.000003 | 0.001989 |
| 15000 | 0.008999 | 0.000003 | 0.002019 |
| 100000 | 0.071235 | 0.000003 | 0.003903 |
| 1000000 | 0.933577 | 0.000004 | 0.004 |

Table 5: Temps d'exécution (secondes) pour differentes tailles de ABR.

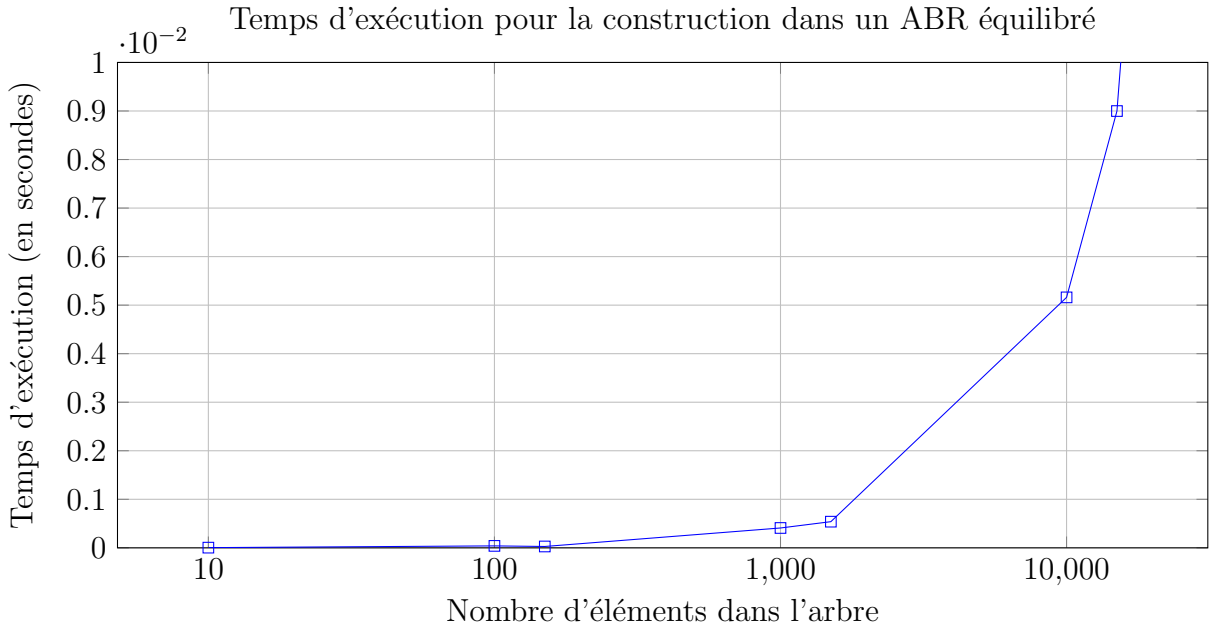
1. ***Comparaison Complexité Théorique Temporelle et Expérimentale*** :

| Opération | Complexité Temporelle | Complexité Expérimentale |
|---|---|---|
| Insertion | O(log n) | - La complexité dépend de la hauteur de l'arbre. Dans le meilleur des cas (arbre équilibré), elle est logarithmique. Dans le pire des cas (arbre dégénéré), elle est linéaire $O(n)$. Expérimentalement, pour des ABR équilibrés contenant entre 10 et 1000000 éléments, le temps d'insertion reste très rapide (millisecondes). |
| Suppression | O(log n) | - Comme pour l'insertion, la complexité dépend de la hauteur de l'arbre. Elle est logarithmique pour un arbre équilibré et linéaire pour un arbre dégénéré. En pratique, pour un ABR équilibré, la suppression est rapide, avec des résultats similaires à l'insertion. |
| Recherche | O(log n) | - La recherche est logarithmique pour un arbre équilibré, mais linéaire dans un arbre dégénéré. Expérimentalement, les performances restent bonnes pour des arbres équilibrés, même avec un grand nombre de nœuds (10 à 1000000). |

2. ***Comparaison Complexité Théorique Spatiale et Expérimentale*** :

| Opération | Complexité Spatiale | Explication |
|---|---|---|
| Insertion | O(n) | - La complexité spatiale est dominée par la mémoire utilisée pour stocker les nœuds de l'ABR. Chaque insertion ajoute un nœud à la structure, nécessitant $O(1)$ pour ce nœud, mais la mémoire totale reste proportionnelle à $n$. |
| Suppression | O(n) | - Aucune mémoire supplémentaire n'est nécessaire pour la suppression, car les modifications se font directement dans la structure existante de l'ABR. |
| Recherche | O(n) | - Comme la recherche ne modifie pas la structure de l'ABR, elle n'a pas de consommation de mémoire supplémentaire, hormis la pile d'appels pour les appels récursifs, qui dépend de la hauteur de l'arbre. |

3. ***Graphique des Temps d'Exécution pour la Construction d'un ABR*** :




**Analyse Critique** Les données obtenues montrent que le temps d'insertion dans un arbre binaire de recherche (ABR) suit globalement une tendance croissante, cohérente avec une complexité logarithmique O(log n) pour des arbres équilibrés. Cependant, ce résultat peut varier en fonction des performances de la machine utilisée.

### 2.2.5 Conclusion

Les arbres binaires de recherche (ABR) offrent une solution efficace pour rechercher, insérer et supprimer des opérations grâce à leur structure hiérarchique. Ses performances sont idéales dans sa version équilibrée, avec une complexité logarithmique pour la plupart des opérations. Cependant, son efficacité peut être compromise en cas de déséquilibre, conduisant à une complexité linéaire dans le pire des cas. Bien adapté aux contextes qui nécessitent des recherches fréquentes et rapides.

## 2.3 B-trees

### 2.3.1 Description de la Structure

**Définition :**

Un B-tree est une structure de données arborescente équilibrée qui permet d'organiser des données de manière efficace pour la recherche, l'insertion et la suppression. Il est largement utilisé dans les bases de données et sur les disques pour minimiser les accès excessifs et améliorer la gestion des données à grande échelle [4].

**Particuliarités des B-trees [5] :**

- **Équilibrage automatique** : Un B-tree est toujours équilibré, car toutes les feuilles se trouvent au même niveau, ce qui garantit que la profondeur de l'arbre reste faible, même avec un grand nombre de données.

- **Nombre de clés variable dans chaque nœud** : Chaque nœud contient entre t−1 et 2t−1 clés, où t est le degré minimal de l'arbre. La flexibilité du nombre de clés permet une gestion efficace de l'espace mémoire.
- **Clés triées dans chaque nœud** : Les clés dans chaque nœud sont toujours triées, ce qui facilite les recherches binaires à l'intérieur des nœuds.
- **Gestion dynamique des insertions et suppressions** : Les B-trees ajustent automatiquement leur structure en devisant ou en fusionnant des noeuds pour maintenir les propriétés de l'arbre, mêm aprés des opérations complexes.

**Applications et utilités des B-Tree[3] :**

Les B-Trees sont largement utilisés dans divers domaines, notamment pour la gestion des données et dans les systèmes où les accès disque sont coûteux. Parmi leurs principales applications, on trouve :

- **Bases de données relationnelles** : Comme Mysql, PostgreSQL et Oracle, les B-trees sont utilisés pour structurer les index, permettant une recherche rapide des enregistrements dans des bases de données.
- **Systèmes de fichiers** : Comme NTFS (Windows), qui utilise des B-trees pour organiser les fichiers et repertoirs, ce qui garantit un accés rapide en minimisant les opérations de lectures/écritures sur dique.
- **Systèmes de gestion de mémoire secondaire** : Les B-trees permettent une gestion efficace des données stockées sur disque, où l'accès séquentiel et aléatoire est coûteux. Ils minimisent les lectures et écritures en regroupant les données dans des blocs optimisés pour le matériel.

Pour conclure, les B-trees sont une structure de données qui facilite la gestion des ensembles de données volumineux, ce qui les rend particulièrement adaptés aux base de données et aux systèmes de fichiers.

### 2.3.2 Explication des Algorithmes

**Pseudocode pour l'algorithme d'insertion**

```
Procédure inserer(arbre, k)
Entrée : - k : La valeur à insérer.
         - arbre : La racine
Sortie : le B-tree aprés suppression
var
    i:entier;
    Début
        si(arbre == nill) alors
            créer un nouveau noeud comme racine
            ajouter la clé k a la racine
            return;
        sinon
            si( k existe ) alors
```

```
            afficher("valeur deja existante");
            return;
        Fsi;
        si( racine pleine ) alors
            -créer un nouveau noeud temporaire s
            -Affecter la racine actuelle comme premier enfant de s
            - Appeler Deviser(s,0,racine) pour deviser
            le premier enfant de s
            i=0;
            si (s.cles[0] < k) alors choisir le deuxieme    enfant, i++;
                sinon choisir le premier
            Fsi;
            InsererNonRempli(s.C[i], k) //C, les fils
            Metrre s comme nouvelle racine
        Sinon //racine non pleine
            insererNonRempli(arbre.racine, k)
        Fsi;
Fin
```

---

**Explication :** L'insertion dans un B-arbre, repose sur plusieurs étapes essentielles, toutes orchestrées par la fonction principale `inserer`. Lorsqu'une clé doit être insérée dans le B-arbre, cette fonction commence par vérifier si l'arbre est vide. Si oui, un nouveau nœud racine est créé via `CreerNoeudArbre`, et la clé y est insérée directement. Si l'arbre n'est pas vide, la fonction vérifie si la clé existe déjà dans l'arbre en appelant `chercherVal`. Si la clé existe, un message est affiché et l'insertion est interrompue. Sinon, si la racine est pleine (c'est-à-dire que son nombre de clés atteint $2t - 1$, où $t$ est le degrée minimale de l'arbre), une opération de division est déclenchée via `Deviser`. Un nouveau nœud racine est alors créé pour contenir le résultat de cette division, et la clé est insérée dans l'un des deux nœuds résultants en utilisant la fonction `insererNonRempli`. Si la racine n'est pas pleine, la clé est insérée directement dans l'arbre à l'aide de `insererNonRempli`, qui gère deux cas :

- Si le nœud est une feuille, la clé est ajoutée à la bonne position tout en décalant les autres clés pour maintenir l'ordre.
- Si le nœud n'est pas une feuille, la fonction localise le sous-arbre approprié où insérer la clé. Si ce sous-arbre est plein, il est d'abord divisé avec `Deviser`, et la clé peut être insérée dans le sous-nœud correct après mise à jour.

Ainsi, les fonctions appelées pour une insertion complète sont `CreerNoeudArbre`, `chercherVal`, `Deviser`, et `insererNonRempli`.

### Pseudocode pour l'algorithme de suppression

```
Fonction supprimer(nœud, clé)
Entrée : racine, et la clé qu'on souhaite supprimer
Sortie : l'arbre avec l'element supprimer
Var :
```

```
i: entier
Début
i ← chercherIndex(nœud, clé)
Si i < nombreDeClés(nœud) et clé == nœud.clés[i]:
    Si (nœud.estFeuille):
        Supprimer clé de nœud.clés
    Sinon:
        Si (nœud.fils[i] a au moins t clés):
            prédécesseur ← obtenirPrédécesseur(nœud.fils[i])
            nœud.clés[i] ← prédécesseur
            supprimer(nœud.fils[i], prédécesseur)
        Sinon
            si (nœud.fils[i+1]) a au moins t clés:
            successeur ← obtenirSuccesseur(nœud.fils[i+1])
            nœud.clés[i] ← successeur
            supprimer(nœud.fils[i+1], successeur)
            Sinon:
                Fusionner(nœud.fils[i], nœud.fils[i+1])
                supprimer(nœud.fils[i], clé)
            Fsi;
        Fsi;
    Fsi;
Sinon
    si nœud.estFeuille:
        Retourner (clé non trouvée)
    Sinon:
     enfant ← nœud.fils[i]
        Si enfant a moins de t clés:
             AssurerQueNonVide(nœud, i)
             supprimer(nœud.fils[i], clé)
        Fsi;
    Fsi;
Fsi;
Fin
```

---

**Explication :** L'opération de suppression dans un B-arbre est plus complexe et dépend de plusieurs cas :

- Si la clé se trouve dans un nœud feuille, elle est simplement supprimée.
- Si la clé se trouve dans un nœud interne :
  - Si le sous-nœud gauche (prédécesseur) ou le sous-nœud droit (successeur) possède au moins $t$ clés, la clé est remplacée par son prédécesseur ou successeur, puis supprimée de ce sous-nœud.
  - Si les deux sous-nœuds ont moins de $t$ clés, ils sont fusionnés, et la suppression continue dans le nœud résultant.

- Si la clé n'est pas dans le nœud courant, mais potentiellement dans un sous-arbre, une étape préalable consiste à s'assurer que ce sous-arbre contient au moins $t$ clés pour éviter des problèmes de structure lors de la suppression.

Les fonctions clés utilisées dans la suppression sont `chercherIndex`, `obtenirPrédécesseur`, `obtenirSuccesseur`, `Fusionner`, et `AssurerQueNonVide`.

**Pseudocode pour l'algorithme de recherche**

```
Fonction rechercher(nœud, clé)
Entrée : noeud racine, et la valeur
Sortie : Boolean
Var :
    i : entier
    Début
        Si nœud est NULL:
             Retourner False (clé non trouvée)
        Sinon:
        i ← 0
        Tant que i < nombreDeClés(nœud) et clé > nœud.clés[i]:
            i ← i + 1
            Si i < nombreDeClés(nœud) et clé == nœud.clés[i]:
                Retourner True (clé trouvée)
            Si nœud.estFeuille == True:
                Retourner False (fin de la recherche dans une feuille)
            Sinon:
                Retourner rechercher(nœud.fils[i], clé)
        Fsi;
    Fin;
```

---

**Explication :**La recherche d'une clé dans un B-Tree est effectuée de manière récursive en suivant les propriétés de l'arbre. La fonction `rechercher` commence par examiner les clés du nœud courant, en avançant jusqu'à trouver la position où la clé pourrait se situer. Si la clé est trouvée, la recherche s'arrête et retourne un résultat positif. Si le nœud est une feuille et que la clé n'est pas trouvée, cela signifie que la clé n'existe pas dans l'arbre. Dans le cas où le nœud n'est pas une feuille, la recherche continue dans le sous-arbre approprié.

### 2.3.3 Calcul de la Complexité Théorique temporelle et spatiale

**Complexité Théorique temporelle :**

1. **Insertion :** L'insertion dans un B-tree suit une procédure qui combine recherche, insertion et gestion des devisions, et sa complexité temporelle théorique est $O(m.log(n))$, où **m** est l'ordre de l'arbre (le nombre maximum d'enfants qu'un nœud peut avoir) et **n** le nombre total de clés. La recherche de la position d'insertion commence à la racine et descend jusqu'à un nœud feuille, avec une recherche séquentielle dans

chaque nœud prenant $O(m)$, ce processus se répétant sur une hauteur de $O(log(n))$. Une fois la position trouvée, si le nœud a de l'espace, la clé est insérée en $O(m)$. Si le nœud est plein, une devision est effectuée, ce qui inclut la division des clés en deux et la propagation de la clé médiane vers le parent, pouvant également nécessiter des scissions successives jusqu'à la racine. Chaque devision reste limitée à $O(m)$ et le nombre maximal de scissions correspond à la hauteur de l'arbre, soit $O(log(n))$. Ainsi, l'insertion globale dans un B-tree combine ces étapes et garantit une efficacité logarithmique même dans le pire des cas.

2. **Suppression :**La suppression dans un B-tree, implique plusieurs étapes, telles que la recherche de l'élément à supprimer, le réajustement des nœuds après la suppression, et la gestion des cas où un nœud devient trop vide après la suppression. La recherche du nœud et de la clé à supprimer prend $O(log(n))$, car l'arbre est équilibré et possède une hauteur $O(log(n))$, avec chaque niveau nécessitant une recherche dans un nœud de $O(m)$ ou m est l'ordre du B-tree. Après avoir trouvé la clé, plusieurs opérations peuvent être nécessaires pour réajuster l'arbre : si un nœud est trop vide, il faudra soit le décaler avec un nœud adjacent (opération $O(m)$), soit fusionner deux nœuds, ce qui implique aussi une complexité $O(m)$ pour la gestion des valeurs et des liens. En cas de fusion ou de déplacement de valeurs, ces étapes peuvent être récursivement appliquées à des nœuds parents, jusqu'à ce qu'une condition d'équilibre soit rétablie. La hauteur de l'arbre, qui est $O(log(n))$, détermine donc le nombre maximum de niveaux où ces ajustements peuvent avoir lieu. Ainsi, la complexité temporelle théorique de la suppression dans un B-tree est $O(m.log(n))$, où m est l'ordre du B-tree et n est le nombre total de clés dans l'arbre.

3. **Recherche :** La complexité temporelle de la recherche d'une valeur dans un B-tree, est $O(log(n))$, où n est le nombre total de clés dans l'arbre. À chaque niveau de l'arbre, la recherche parcourt un nœud et décide dans quel sous-arbre continuer la recherche en fonction de la valeur comparée. Le processus de recherche descend donc d'un niveau à chaque étape, et comme un B-tree est équilibré, le nombre de niveaux est $O(log(n))$. Le temps pour traverser un nœud est constant, donc la recherche est logarithmique par rapport au nombre de clés dans l'arbre.
   **A noter que la variable m ( l'ordre du B-tree ) il peut être negligable car il peut être considerer comme une constante et donc la complexité finale est indiquer dans la Table 8.**

| **Opération** | **Meilleur Cas** | **Cas Moyen** | **Pire Cas** |
|---|---|---|---|
| **Insertion** | $O(1)$ | $O(\log n)$ | $O(\log n)$ |
| **Suppression** | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| **Recherche** | $O(1)$ | $O(\log n)$ | $O(\log n)$ |

Table 8: Complexités temporelles des différentes opérations sur un B-tree

**Complexité spatiale :**

| Opération | Complexité spatiale |
|---|---|
| **Recherche** | La complexité spatiale de la recherche est O(1). La recherche dans un B-tree utilise un nombre constant d'espace mémoire supplémentaire, car elle ne nécessite pas de structures de données supplémentaires (autres que la pile d'appel pour la récursion). À chaque niveau de l'arbre, la recherche décide simplement quel sous-arbre explorer, et chaque appel récursif ne stocke que les informations de position pour le nœud actuel. |
| **Suppression** | La complexité spatiale de la suppression est O(h), où h est la hauteur de l'arbre, car la suppression fait appel à une récursion pour descendre dans l'arbre et ajuster les nœuds. Dans le pire des cas, l'opération de suppression peut nécessiter une profondeur de récursion égale à la hauteur de l'arbre. Cela signifie que l'espace utilisé est proportionnel à la hauteur de l'arbre à cause des appels récursifs. |
| **Insertion** | La complexité spatiale de l'insertion est O(h), pour des raisons similaires à la suppression. Lors de l'insertion, le processus peut impliquer une récursion pour descendre l'arbre et éventuellement séparer ou réorganiser les nœuds. Dans le pire des cas, l'insertion peut entraîner des appels récursifs jusqu'à la hauteur de l'arbre, ce qui donne une complexité spatiale de O(h). Comme h = O(log n), la complexité spatiale est donc O($\log n$). |

Table 9: Complexité spatiale des différentes opérations sur un B-tree

### 2.3.4 Experimentation :

Soit la table 3, un tableau qui illustre les differents temps d'execution pour chaque operation dans un B-tree en fonction de la taille du B-tree. **On suppose que le B-tree est d'ordre 4.**

| Taille | Temps d'insertion | Temps de suppression | Temps de recherche |
|---|---|---|---|
| 100 | 0.0 secondes | 0.0 secondes | 0.0 secondes |
| 1,000 | 0.0 secondes | 0.0 secondes | 0.0 secondes |
| 10,000 | 0.003 secondes | 0.001 secondes | 0.001 secondes |
| 100,000 | 0.076 secondes | 0.023 secondes | 0.013 secondes |
| 1,000,000 | 0.507 secondes | 0.261 secondes | 0.158 secondes |
| 10,000,000 | 4.653 secondes | 3.058 secondes | 2.012 secondes |

Table 10: Temps d'exécution pour les opérations sur le B-tree

| Taille | Log(taille) |
|---|---|
| 100 | 2 |
| 1,000 | 3 |
| 10,000 | 4 |
| 100,000 | 5 |
| 1,000,000 | 6 |
| 10,000,000 | 7 |

Table 11: Complexité temporelle pour les opérations sur le B-tree

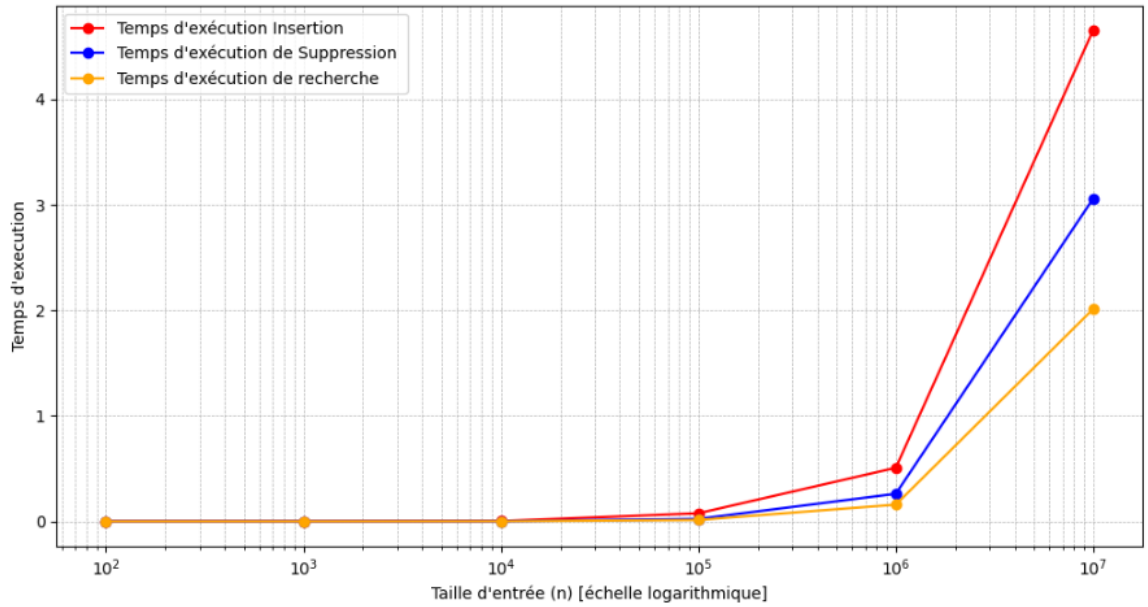



Figure 2: Temps d'execution des operations en fonction de la taille sur un log

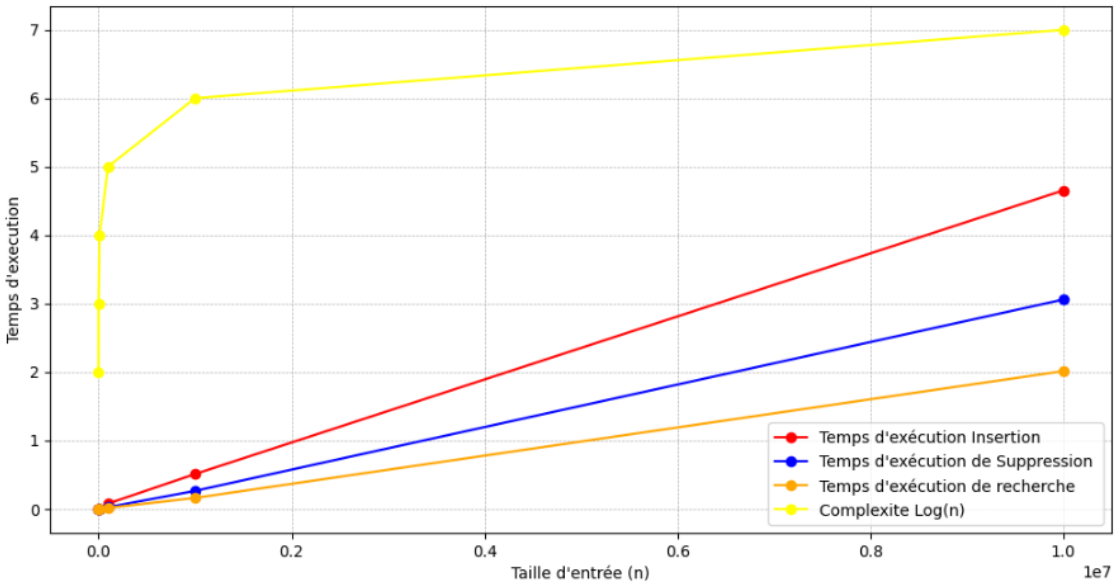



Figure 3: Temps d'insertion en fonction du Log de la taille

**Comparaison entre la complexite theorique et experimental :** La comparaison entre la complexité théorique et expérimentale des B-trees révèle une concordance notable. Sur le plan temporel, les opérations d'insertion, de suppression et de recherche présentent une complexité théorique de O(logn), reflétant la structure équilibrée des B-trees. Les résultats expérimentaux confirment cette performance, avec des temps d'exécution rapides même pour des tailles importantes, par exemple 0.507 secondes pour une insertion, 0.261 secondes pour une suppression, et 0.158 secondes pour une recherche sur 1,000,000 éléments.

En ce qui concerne la complexité spatiale, l'insertion et la suppression ont une consommation proportionnelle à la hauteur de l'arbre O(logn), principalement due aux appels récursifs. La recherche, quant à elle, nécessite un espace constant (O(1)) car elle n'utilise pas de structures auxiliaires. Ces résultats expérimentaux soulignent l'efficacité des B-trees, tant en termes de temps que de mémoire, pour gérer des ensembles de données volumineux tout en garantissant des performances optimales.

**Analyse des resultats obtenus :** La complexité d'insertion, recherche ou suppresion dans un B-arbre $O(logn)$ est efficace et adaptée pour gérer de grandes quantités de données, mais elle présente certaines limites et critiques lorsque la taille des données devient extrêmement grande (selon la figure 1 et 2) ou le temps d'execution augmente d'une maniere un peux plus rapide. Voici une analyse critique :

1. **Coût des scissions :** Bien que $O(logn)$ paraisse optimal, les devisions au sein des nœuds peuvent impliquer plusieurs ajustements entre les niveaux. Pour des très grands M (ordre de l'arbre), ces ajustements deviennent coûteux, augmentant la constante cachée dans la complexité.

2. **Fragments mémoire :** Dans des environnements où la mémoire est limitée, l'allocation dynamique pour les nœuds peut entraîner de la fragmentation mémoire, ralentissant les opérations.

3. **Temps de gestion de grands nœuds :** Si M est très grand (comme dans certains B-arbres utilisés pour les bases de données), la recherche dans un nœud O(M) peut devenir un facteur limitant. Par exemple si M est a 1,000,000 chaque recherche dans un nœud devient coûteuse, et l'efficacité logarithmique est partiellement compromise.

4. **Problèmes avec des tailles massives de données distribuées :** Les B-arbres traditionnels ne gèrent pas efficacement la distribution de données entre plusieurs machines. Dans un environnement distribué, L'insertion peut nécessiter des splits qui propagent des modifications sur plusieurs machines, augmentant ainsi les temps de synchronisation

### 2.3.5 Conclusion :

Pour finir, La complexité $O(logN)$ reste théoriquement idéale pour des tailles de données croissantes. Cependant, dans des scénarios extrêmes (très grand M), les splits, les coûts de gestion mémoire, et les contraintes d'accès disque peuvent provoquer des dégradations de performance. Cela nécessite souvent l'adoption de structures plus adaptées, comme les B+ arbres, pour maintenir une insertion efficace.

Bien que la structure de B-Tree ne soit pas toujours la plus adaptée dans tous les cas, elle reste néanmoins efficace dans de nombreuses applications. Sa capacité à maintenir un équilibre et à garantir des performances logarithmiques en fait une solution robuste pour gérer de grandes quantités de données, notamment dans des systèmes nécessitant un accès rapide et une gestion efficace des pages de mémoire. Par exemple, les bases de données relationnelles, telles que MySQL et PostgreSQL, utilisent des arbres B pour l'indexation des données, permettant des recherches rapides et efficaces sur des ensembles de données massifs. De même, les systèmes de fichiers comme NTFS (New Technology File System) emploient des arbres B pour organiser et accéder aux fichiers de manière optimisée.

Même si certaines limitations pratiques peuvent apparaître avec des tailles de données extrêmement grandes, l'arbre B demeure une structure de données versatile, parfaitement adaptée à des scénarios où l'efficacité de l'insertion, de la suppression et de la recherche est primordiale, comme dans les systèmes de gestion de bases de données ou les systèmes de fichiers distribués.

## 2.4 Tas (Heap) [2]

### 2.4.1 Description de la Structure

**Définition** Un tas, ou heap, est une structure de données arborescente qui satisfait la propriété de tas, ce qui signifie que chaque parent est supérieur (dans un tas max) ou inférieur (dans un tas min) à ses enfants. Les tas sont souvent utilisés pour implémenter des files d'attente prioritaire, où l'élément avec la plus haute priorité est toujours à la racine.

**Particuliarités des Tas**

- Arbre Binaire : Un tas est généralement représenté sous forme d'un arbre binaire complet, ce qui signifie que tous les niveaux de l'arbre, sauf peut-être le dernier, sont complètement remplis, et les nœuds sont aussi à gauche que possible.
- Propriété de Tas : Pour un tas max, la valeur de chaque nœud est supérieure ou égale à celle de ses enfants, tandis que pour un tas min, la valeur de chaque nœud est inférieure ou égale à celle de ses enfants.
- Implémentation : Les tas sont souvent implémentés à l'aide de tableaux, où pour un nœud à l'index $i$, ses enfants sont situés à $2i + 1$ (gauche) et $2i + 2$ (droite), et son parent est à $\lfloor (i - 1)/2 \rfloor$.

**Applications et utilités des Tas**
Les tas sont utilisés dans diverses applications informatiques, notamment :

- Tri par tas (Heap Sort) : Un algorithme de tri qui utilise un tas pour trier des éléments.
- Files d'attente prioritaire : Structure de données qui gère les éléments en fonction de leur priorité.
- Algorithmes de graphes : Comme l'algorithme de Dijkstra, qui utilise un tas pour gérer les nœuds à explorer.

En résumé, les tas sont une structure de données efficace pour gérer des priorités et effectuer des opérations de tri.

### 2.4.2 Explication des Algorithmes

**Pseudocode pour l'algorithme d'insertion**

```
FONCTION Insert(heap, value):
    SI heap.size >= MAX_HEAP_SIZE:
        AFFICHER "Le tas est plein!"
        RETOURNER -1

    // Insérer le nouvel élément à la fin
    heap.data[heap.size] ← value
    heap.size ← heap.size + 1

    // Déplacer le nouvel élément vers le haut pour maintenir la propriété
    du tas maximal
    index ← heap.size - 1
    TANT QUE index > 0 ET heap.data[Parent(index)] < heap.data[index]:
        SWAP(heap.data[Parent(index)], heap.data[index])
        index ← Parent(index)
```

**Explication :** La fonction insert ajoute une nouvelle valeur dans un tas (heap) en vérifiant d'abord si la capacité maximale est atteinte. Si le tas n'est pas plein, la valeur est placée à la fin de la liste heap.data, et la taille du tas est augmentée. Ensuite, pour maintenir la propriété de max-heap, la fonction remonte l'élément inséré en échangeant sa place avec celle de son parent tant que ce dernier est plus petit. Cela garantit que la structure du tas respecte l'ordre max-heap après l'insertion de la nouvelle valeur.

**Pseudocode pour l'algorithme de recherche**

```
FONCTION SearchInHeap(heap, number):
    POUR i DE 0 À heap.size - 1:
        SI heap.data[i] == number:
            RETOURNER i
    RETOURNER -1
```

**Explication :** La fonction searchInHeap parcourt les éléments d'un tas (heap) pour rechercher une valeur spécifique. Elle utilise une boucle pour vérifier chaque élément de heap.data de l'index 0 jusqu'à heap.size - 1. Si elle trouve une correspondance avec la valeur number, elle retourne l'index de l'élément. Si la valeur n'est pas trouvée, la fonction retourne -1.

**Pseudocode pour l'algorithme de suppression**

```
FONCTION DeleteInHeap(heap, number):
    index ← SearchInHeap(heap, number)
    SI index != -1:
        // Échanger l'élément avec le dernier élément
```

```
    SWAP(heap.data[heap.size - 1], heap.data[index])
    heap.size ← heap.size - 1
    // Restaurer la propriété du tas
    Heapify(heap, heap.size, index)
SINON:
    AFFICHER "L'élément n'existe pas dans le tas!"
```

**Explication :** La fonction deleteInHeap supprime un élément spécifique dans un tas (heap) en commençant par rechercher l'index de l'élément via searchInHeap. Si l'élément est trouvé, il est échangé avec le dernier élément du tas, puis la taille du tas est réduite. Ensuite, la fonction heapify est appelée pour rétablir la propriété de max-heap à partir de la position de l'élément déplacé. Si l'élément n'est pas présent, un message indique qu'il n'existe pas dans le tas.

### 2.4.3 Calcul de la Complexité Théorique temporelle et spatiale

**Complexité temporelle**

1. **Insertion:** L'algorithme d'insertion commence par placer l'élément à la fin du tas, une opération en temps constant $O(1)$. Ensuite, pour rétablir la propriété de max-heap, il "remonte" l'élément ajouté en le comparant et en l'échangeant avec ses parents successifs si nécessaire. Dans le pire des cas, l'élément peut remonter jusqu'à la racine, et le nombre de remontées est proportionnel à la hauteur du tas, qui est de $O(\log n)$ pour un tas de $n$ éléments. La complexité théorique de l'algorithme `insert` est donc $O(\log n)$.

2. **Suppression**: L'algorithme de suppression recherche d'abord l'élément à supprimer en parcourant les éléments du tas avec la fonction `searchInHeap`, qui a une complexité de $O(n)$ dans le pire des cas, car il pourrait devoir examiner chaque élément. Une fois l'élément trouvé, il est échangé avec le dernier élément, puis la taille du tas est réduite. Pour restaurer la propriété de max-heap, la fonction `heapify` est ensuite appelée et a une complexité de $O(\log n)$ pour réorganiser le tas. Comme la recherche initiale est l'opération la plus coûteuse, la complexité totale de la suppression est donc $O(n)$.

3. **Recherche**: L'algorithme de recherche, `searchInHeap`, parcourt linéairement les éléments du tas pour trouver une valeur donnée. Dans le pire des cas, l'algorithme doit examiner tous les éléments pour vérifier la présence de la valeur, ce qui entraîne une complexité de $O(n)$. Cette complexité est attendue pour une recherche dans un tas non ordonné de manière séquentielle, car l'accès direct aux éléments en fonction de leur valeur n'est pas possible dans une structure de tas max.

| **Opération** | **Meilleur Cas** | **Cas Moyen** | **Pire Cas** |
|---|---|---|---|
| **Insertion** | $O(1)$ | $O(\log n)$ | $O(\log n)$ |
| **Suppression** | $O(\log n)$ | $O(\log n)$ | $O(n)$ |
| **Recherche** | $O(1)$ | $O(n)$ | $O(n)$ |

Table 12: Complexités temporelles des différentes opérations sur un tas

**Complexité Spaciale**
Pour chaque algorithme, la complexité spatiale est $O(n)$, car elle est dominée par la mémoire requise pour stocker les éléments dans la structure de tas, qui contient jusqu'à $n$ éléments.

### 2.4.4 Expérimentation

Nous avons effectué les 3 operations precedentes tout en variant la taille du tas. La figure ci-dessous resume les resultats obtenus des temps d'executions en secondes:

| Taille du tas | Temps d'insertion | Temps de suppression | Temps de recherche |
|---|---|---|---|
| 100 | 0.000000 | 0.000000 | 0.000000 |
| 1,000 | 0.000000 | 0.000000 | 0.000000 |
| 10,000 | 0.000000 | 0.000000 | 0.000000 |
| 100,000 | 0.000000 | 0.001000 | 0.000000 |
| 1,000,000 | 0.000000 | 0.024000 | 0.009000 |
| 10,000,000 | 0.000000 | 0.032000 | 0.031000 |

Table 13: Temps d'exécution pour les opérations sur le tas

# 3 Tri par Tas [1]

## 3.1 Construction du Tas

### 3.1.1 Fonctions de base

**Opérations élementaires**

```
// Fonction pour initialiser un tas maximal
FONCTION InitMaxHeap(heap):
    heap.size ← 0  // Initialiser la taille à 0

// Fonction pour obtenir l'index du parent
FONCTION Parent(index):
    RETOURNER (index - 1) / 2  // -1 car l'index commence à 0 dans le langage C

// Fonction pour obtenir l'index du fils gauche
FONCTION LeftChild(index):
    RETOURNER (2 * index + 1)

// Fonction pour obtenir l'index du fils droit
FONCTION RightChild(index):
    RETOURNER (2 * index + 2)

// Fonction pour échanger deux éléments
FONCTION Swap(a, b):
    temp ← a
    a ← b
    b ← temp
```

**Explications:**

- **Initialisation du Tas :**
  - La fonction `initMaxHeap` initialise un tas vide.
  - La taille du tas est définie à 0, ce qui signifie qu'il n'y a pas encore d'éléments dans le tas.
- **Calcul de l'Indice du Parent :**
  - La fonction `parent` calcule l'indice du parent d'un élément dans le tas.
  - Pour un élément à l'indice $index$, l'indice du parent est donné par $\left(\frac{index-1}{2}\right)$.
  - Cette relation est basée sur la représentation du tas binaire sous forme de tableau.
- **Calcul de l'Indice du Fils Gauche :**
  - La fonction `leftChild` calcule l'indice du fils gauche d'un élément dans le tas.
  - Pour un élément à l'indice $index$, l'indice du fils gauche est donné par la formule $2 \times index + 1$.
  - Cette relation est également basée sur la structure binaire du tas.
- **Calcul de l'Indice du Fils Droit :**
  - La fonction `rightChild` calcule l'indice du fils droit d'un élément dans le tas.
  - Pour un élément à l'indice $index$, l'indice du fils droit est donné par la formule $2 \times index + 2$.
- **Échange de Deux Éléments :**
  - La fonction `swap` échange les valeurs de deux éléments dans le tableau représentant le tas.
  - Elle utilise une variable temporaire pour échanger les valeurs des deux éléments.
  - Cette fonction est essentielle pour maintenir la propriété du tas lorsque des éléments sont déplacés.

**Extraire le maximum**

```
FONCTION ExtractMax(heap):
    SI heap.size == 0:
        AFFICHER "Le tas est vide!"
        RETOURNER -1  // Retourner une valeur invalide

    // Obtenir la valeur maximale (la racine du tas)
    max_element ← heap.data[0]
```

```
    // Remplacer la racine par le dernier élément
    heap.data[0] ← heap.data[heap.size - 1]
    heap.size ← heap.size - 1

    // Déplacer la nouvelle racine vers le bas pour maintenir la propriété
    du tas maximal
    index ← 0
    TANT QUE LeftChild(index) < heap.size:
        largerChild ← LeftChild(index)  // Commencer avec le fils gauche
        // Comparer le fils droit avec le fils gauche pour trouver le plus grand
        SI RightChild(index) < heap.size ET
        heap.data[RightChild(index)] > heap.data[largerChild]:
            largerChild ← RightChild(index)  // Mettre à jour le fils droit
            si c'est le plus grand

        // Si la nouvelle racine est plus grande que les deux enfants, arrêter
        SI heap.data[index] >= heap.data[largerChild]:
            CAS  // La propriété du tas maximal est satisfaite

        // Échanger la nouvelle racine avec le plus grand enfant
        Echanger(heap.data[index], heap.data[largerChild])
        index ← largerChild  // Descendre vers le plus grand enfant

    RETOURNER max_element  // Retourner la valeur maximale
```

**Explication:** L'algorithme extractMax permet d'extraire le maximum d'un max-heap. Il commence par vérifier si le tas est vide et renvoie une valeur invalide dans ce cas. Si le tas n'est pas vide, il récupère la valeur maximale (le premier élément du tableau, représentant la racine du tas). Ensuite, il remplace la racine par le dernier élément du tas et réduit la taille du tas. Pour restaurer la propriété de max-heap, l'algorithme déplace le nouvel élément racine vers le bas en le comparant avec ses enfants gauche et droit. Si l'un des enfants est plus grand que l'élément actuel, l'algorithme échange l'élément avec l'enfant le plus grand et continue le processus récursivement jusqu'à ce que la propriété de max-heap soit rétablie. Enfin, l'algorithme retourne l'élément maximum extrait.

### Heapify

```
FONCTION Heapify(heap, taille, index):
    largest ← index  // Initialiser largest comme la racine
    left ← LeftChild(index)  // Index du fils gauche
    right ← RightChild(index)  // Index du fils droit

    // Vérifier si le fils gauche existe et est plus grand que la racine
    SI left < taille ET heap.data[left] > heap.data[largest]:
        largest ← left
```

```
    // Vérifier si le fils droit existe et est plus grand que
    le plus grand trouvé jusqu'à présent
    SI right < taille ET heap.data[right] > heap.data[largest]:
        largest ← right

    // Si le plus grand n'est pas la racine, échanger et continuer
    à appliquer le heapify
    SI largest != index:
        Echanger(heap.data[index], heap.data[largest])
        Heapify(heap, taille, largest)  // Appliquer récursivement
        heapify au sous-arbre affecté
```

**Explication:** L'algorithme heapify permet de maintenir la propriété de max-heap dans un tableau qui représente un tas binaire. Il commence par comparer le nœud actuel (racine du sous-arbre) avec ses enfants gauche et droit pour identifier le plus grand élément. Si l'un des enfants est plus grand que le nœud actuel, l'algorithme échange les éléments et effectue récursivement la procédure heapify sur le sous-arbre affecté. Cela garantit que la propriété de max-heap (où chaque parent est plus grand que ses enfants) est rétablie, en descendant jusqu'à ce qu'il n'y ait plus d'échanges nécessaires. Ce processus se répète pour chaque nœud du sous-arbre, ce qui permet de maintenir la structure du tas après des opérations comme l'extraction du maximum.

### 3.1.2 Méthode O(nLog(n))

```
POUR i DE 0 À max - 1:
    Inserer(heap, T[i])
```

### 3.1.3 Méthode O(n)

```
heap2.size ← max
POUR i DE 0 À max - 1:
    heap2.data[i] ← T[i]  // Assigner les éléments de T aux données du tas

POUR i DE (max/2)- 1 À 0 PAR DÉCRÉMENT:
    Heapify(heap2, max, i)
```

**Explication du calcul de la complexite:** La boucle allant de (max/2)-1 a 1 applique l'opération `heapify` à chaque nœud, en commençant par le dernier nœud non-feuille jusqu'à la racine. Pour un nœud donné, `heapify` a une complexité $O(h-k)$, où $h$ est la hauteur totale de l'arbre ($h = \lfloor \log n \rfloor$) et $k$ est le niveau du nœud. Au niveau $k$, il y a $2^k$ nœuds, et le travail total à ce niveau est $2^k \cdot O(h-k)$. La complexité totale est donc donnée par la somme sur tous les niveaux $k$ :

$$\text{Travail total} = \sum_{k=0}^{h} 2^k \cdot (h-k).$$

En simplifiant, cette somme équivaut à $O(n)$ car plus de la moitié des nœuds sont proches des feuilles, où le travail est négligeable ($h - k \approx 0$), tandis que seuls quelques nœuds proches de la racine nécessitent un travail significatif. Ainsi, la contribution totale de tous les niveaux reste linéaire par rapport au nombre total de nœuds, $n$.

**Comparaison Complexité Théorique Temporelle et Expérimentale**

| **Opération** | **Complexité Temporelle** | **Complexité Expérimentale** |
|---|---|---|
| Insertion | O(log n) | - Dépend du nombre d'éléments générés (petits/trop grands, répétition des nombres). Testée pour des tas de taille allant de 10 000 à 25 000, et le temps d'exécution est de l'ordre de la milliseconde dans la plupart des cas, avec une légère augmentation pour des tailles plus grandes. |
| Suppression | O(n) | - Variable, dépend de la taille du tas et de l'élément à supprimer. La recherche d'un élément et l'appel à 'heapify' pour restaurer la propriété de max-heap ajoutent une complexité supplémentaire qui peut se mesurer en secondes pour des tas plus grands (tests effectués pour des tailles de 10 000 à 25 000 éléments). |
| Recherche | O(n) | - En fonction de la taille du tas, les résultats sont proportionnels à 'n'. La recherche dans un tas non ordonné peut entraîner des temps plus longs si l'élément recherché est proche de la fin du tas. Les temps d'exécution sont comparables à ceux de la suppression. |

**Comparaison Complexité Théorique Spaciale et Expérimentale**

| **Opération** | **Complexité Spatiale** | **Explication** |
|---|---|---|
| Insertion | O(n) | - La complexité spatiale est dominée par la mémoire utilisée pour stocker les éléments dans le tas. Pour chaque insertion, un seul élément est ajouté sans nécessiter d'espace supplémentaire pour d'autres structures de données. |
| Suppression | O(n) | - Similaire à l'insertion, l'algorithme de suppression modifie la structure de tas existante et n'introduit pas de consommation de mémoire supplémentaire autre que la gestion des éléments stockés dans le tas. |
| Recherche | O(n) | - Aucun espace supplémentaire n'est nécessaire pour la recherche, car l'algorithme parcourt simplement les éléments existants sans changer la structure du tas. |

### 3.1.4 Résultats d'Expérimentation

Les figures ci-dessous donne une idée sur le temps d'executions en secondes pour les deux methodes.

| Taille Tas | Méthode nlog(n) | Méthode O(n) |
|---|---|---|
| 100 | 0.000000 | 0.000000 |
| 1000 | 0.000000 | 0.000000 |
| 10000 | 0.000000 | 0.000000 |
| 100000 | 0.009000 | 0.006000 |
| 1000000 | 0.025000 | 0.006000 |
| 10000000 | 0.175000 | 0.091000 |
| 100000000 | 1.809000 | 0.794000 |

Table 16: Temps d'exécution (secondes) pour différentes tailles de tas avec les méthodes nlog(n) et O(n).

**Graphique des Temps d'Exécution pour la Construction du Tas**

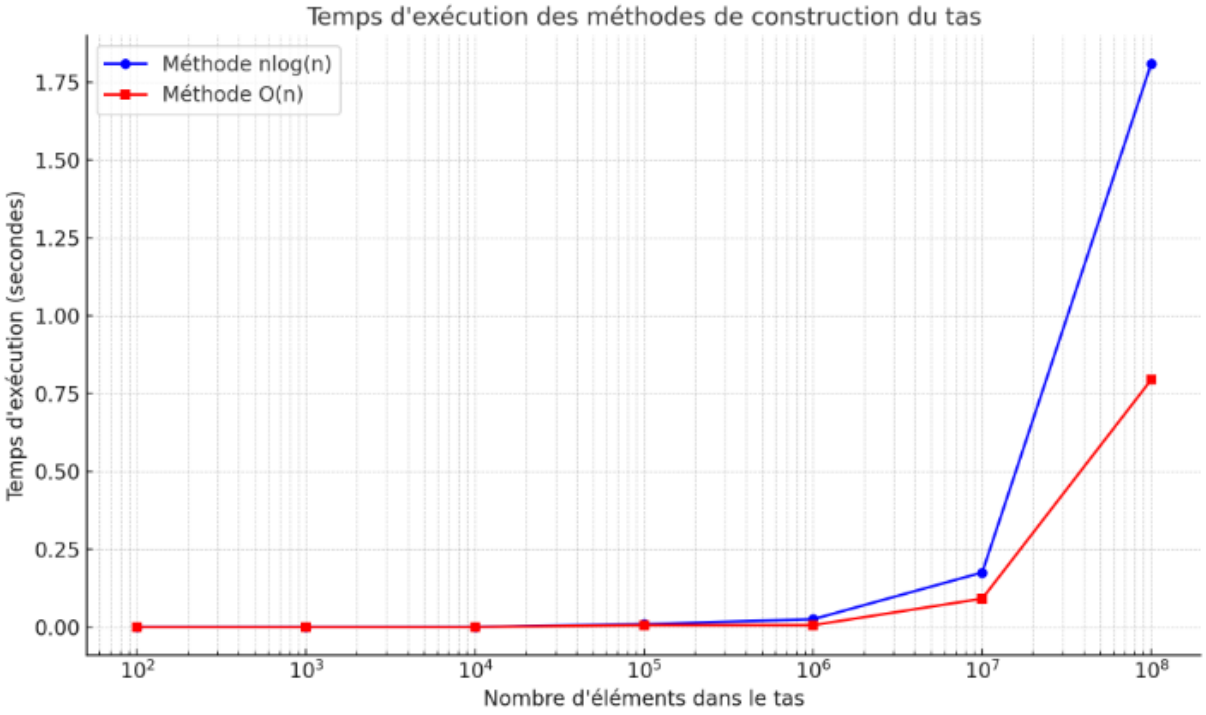



Figure 4: Temps d'execution des methodes de construction du tas

**Analyse Critique** Les résultats montrent que les deux algorithmes se comportent comme prévu selon leurs complexités théoriques, mais des améliorations expérimentales sont nécessaires pour mieux comprendre les différences dans des scénarios réels à grande échelle. L'algorithme O(n) semble plus stable pour les petites tailles de tas, tandis que l'algorithme nlog(n) commence à montrer une augmentation plus marquée des temps d'exécution avec l'augmentation du nombre d'éléments, ce qui est conforme à la théorie.

## 3.2 Tri du Tas

### 3.2.1 Algortithme

```
i ← max - 1
max_element ← null
```

```
TANT QUE i >= 0:
    max_element ← ExtractMax(heap)
    heap.data[i] ← max_element
    i ← i - 1

heap.size ← max
```

**Explication:** L'algorithme de tri par tas (ou heap sort) fonctionne en extrayant successivement l'élément maximum du tas (la racine du tas), puis en le réinsérant à la fin du tableau qui représente le tas. Le processus commence en initialisant i à max-1 (le dernier indice du tas) et en itérant jusqu'à ce que tous les éléments soient extraits. À chaque itération, l'élément maximum est extrait à l'aide de la fonction extractMax, et cet élément est placé à la position i dans le tableau de données. Ensuite, i est décrémenté pour insérer le prochain maximum à la position suivante. Après avoir extrait tous les éléments, le tas est trié dans l'ordre décroissant, car les éléments sont extraits du tas et placés progressivement à la fin du tableau. Finalement, la taille du tas est réinitialisée à sa taille initiale avec heap.size = max.

### 3.2.2 Pourquoi ai-je choisi un Max-heap ?

J'ai choisi d'utiliser un max-heap pour le tri, car cette structure permet de maintenir efficacement l'ordre des éléments tout en offrant des performances optimales pour l'extraction du maximum. L'idée est de tirer parti de la propriété du max-heap, où l'élément maximum est toujours situé à la racine. Plutôt que de créer un tableau séparé pour insérer les éléments triés, je réorganise directement le tas en extrayant successivement l'élément maximum et en le réinsérant à sa position appropriée dans le tas, tout en réduisant progressivement la taille de ce dernier. Cela permet de trier les éléments directement à l'intérieur du tas, sans avoir besoin de stockage supplémentaire, ce qui rend l'algorithme plus efficace en termes de mémoire. Ce processus de "tri par extraction" garantit que le tas devient trié tout en maintenant la structure de données intacte et optimisée pour chaque opération.

## 3.3 Comparaison des Algorithmes de Tri par Différentes Constructions

### 3.3.1 Tri par Tas avec $O(n \log n)$

Après avoir construit le tas, l'algorithme extrait successivement le maximum (ou le minimum pour un min-heap) et le place à la fin du tableau. Chaque extraction nécessite une opération de "heapify" qui prend $O(\log n)$. Étant donné que nous extrayons $n$ éléments, cela donne une complexité de $O(n \log n)$ pour la phase de tri, ce qui mène à une complexité totale du tri de $O(n \log n)$.

### 3.3.2 Tri par Tas avec $O(n)$

Si un tas est construit en $O(n)$, la phase de tri en elle-même (extraction du maximum et réajustement du tas) reste $O(n \log n)$, car chaque extraction nécessite une

"heapification" après chaque retrait du maximum. La complexité totale du tri reste donc $O(n \log n)$, mais la phase de construction du tas est plus rapide grâce à la méthode $O(n)$.

### 3.3.3 Résumé de la Comparaison

**Complexité du Tri :** Dans les deux cas, la phase de tri par extraction reste $O(n \log n)$, mais la méthode $O(n)$ permet de réduire le temps de construction du tas, ce qui peut avoir un impact significatif pour de grandes entrées.

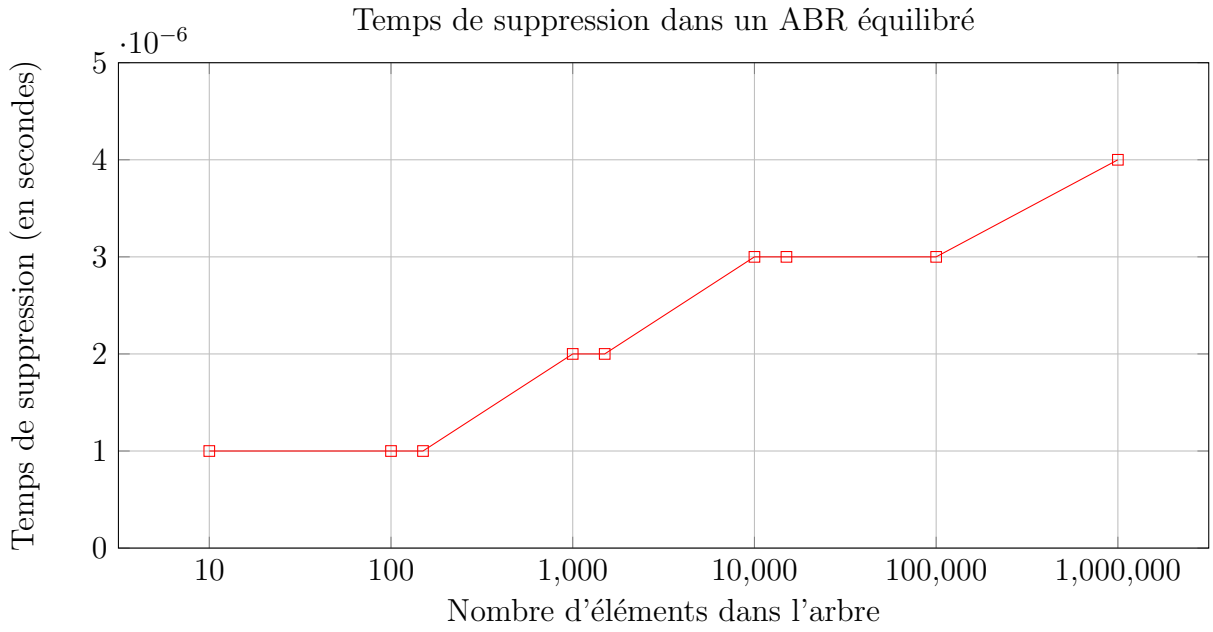
**En conclusion**, bien que les deux méthodes de construction de tas soient valides, la méthode $O(n)$ est généralement préférée pour sa rapidité de construction, surtout lorsqu'on traite de grandes quantités de données, tandis que $O(n \log n)$ reste simple et efficace pour des implémentations plus petites ou plus simples.

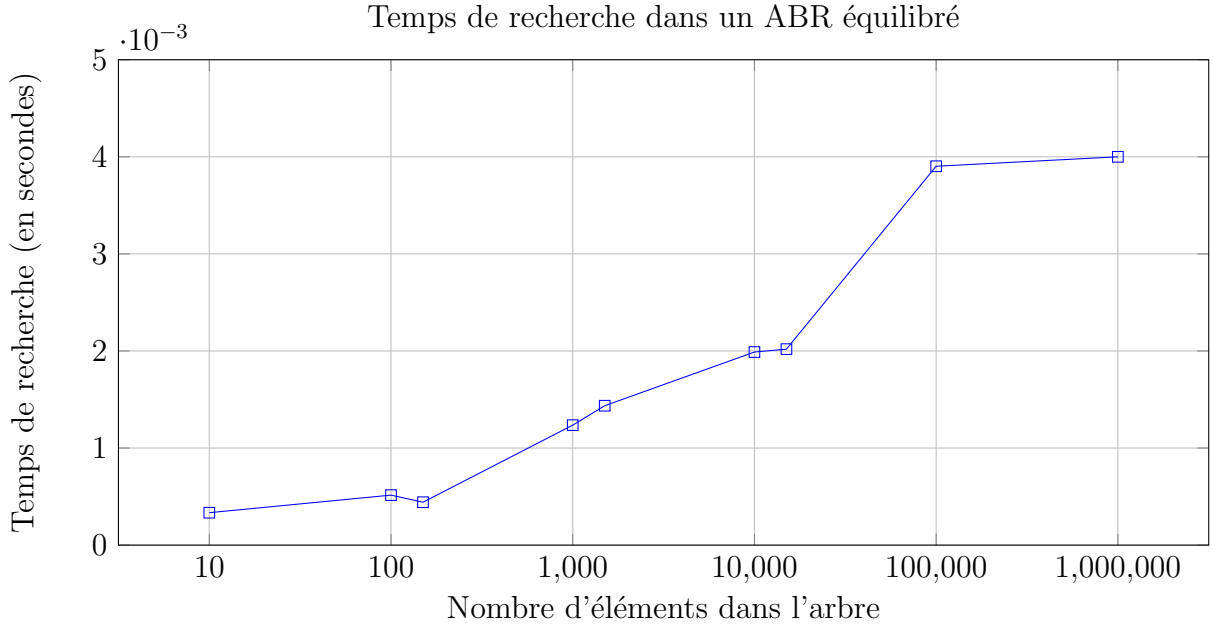
# 4 Conclusion et Recommandation Générale

Les recommandations pour le choix des structures de données incluent l'utilisation des listes chaînées pour des structures dynamiques simples, des arbres binaires de recherche (ABR) ou B-Tree pour des recherches fréquentes, et des tas pour gérer les priorités. Pour l'optimisation, il est conseillé d'utiliser des arbres auto-équilibrés, comme les AVL ou Red-Black Trees, afin de garantir des performances optimales. Enfin, pour les données volumineuses stockées en mémoire externe, le B-Tree est recommandé en raison de son efficacité en termes de mémoire et de temps.

# 5 Annexes

## 5.1 Graphes Supplémentaires



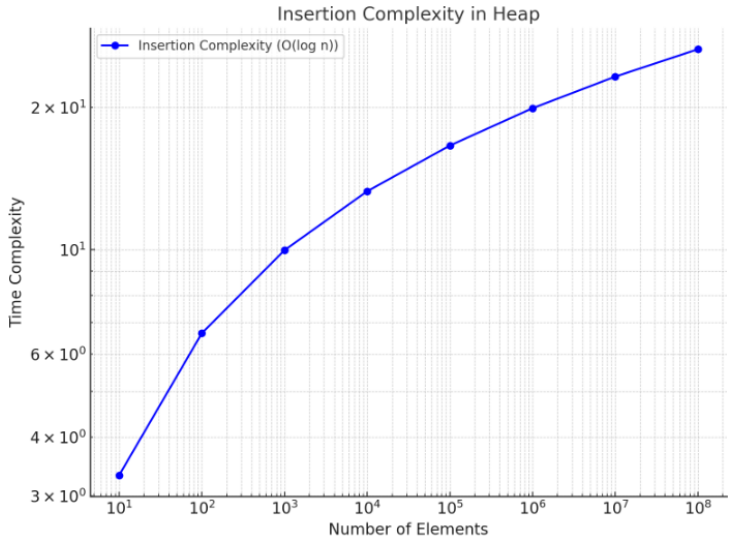



Figure 5: Complexite d'insertion dans un tas(O(log n))

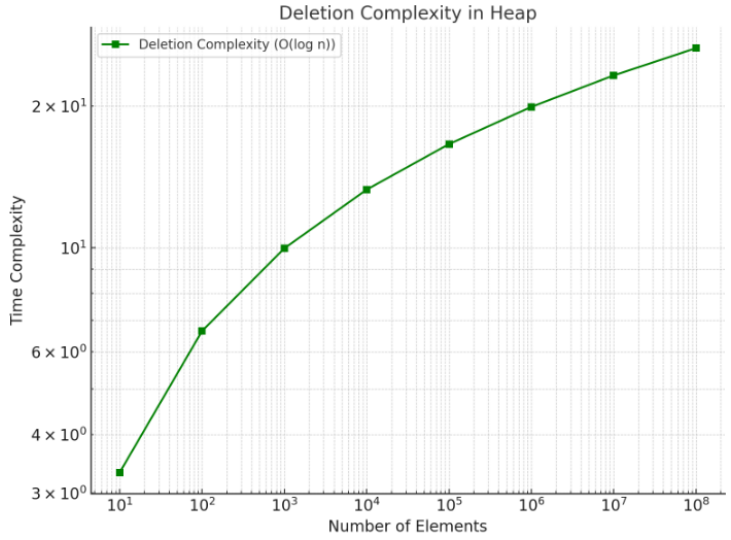



Figure 6: Complexite de suppression dans un tas(O(log n))

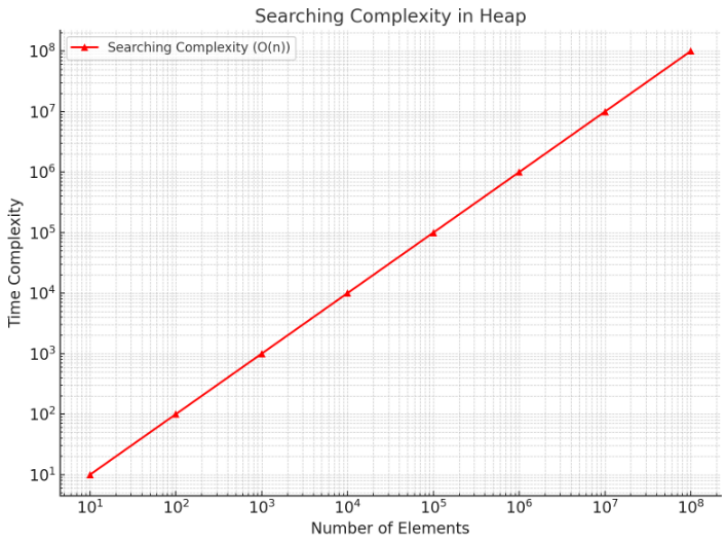



Figure 7: Complexite de recherche dans un tas(O(n))

## 5.2 Résumé des Résultats et Recommandations

| **Structure** | **Résultats (Insertion, Recherche, Suppression)** | **Recommandations** |
|---|---|---|
| Liste Chaînée | Insertion : Performante pour ajouter en tête ou en queue, coûteuse pour une position intermédiaire. Recherche : Linéaire ($O(n)$), inefficace pour les grandes données. Suppression : Simple si la position est connue, lente si une recherche préalable est nécessaire. | Utiliser pour des données de petite taille ou un accès séquentiel. Éviter pour un accès aléatoire fréquent ou les grandes données. |
| Arbre Binaire de Recherche (ABR) | Insertion : Performante sur un arbre équilibré, mais peut devenir linéaire dans un arbre déséquilibré. Recherche : Efficace dans un arbre équilibré ($O(\log n)$), mais linéaire si déséquilibré. Suppression : Complexe pour les nœuds ayant deux enfants, bien gérée avec des implémentations robustes. | Utiliser des arbres équilibrés comme AVL ou Red-Black Trees pour garantir une bonne performance. Éviter les ABR simples si les données sont triées ou insérées séquentiellement. |
| B-Tree | Insertion : Très efficace pour les grandes bases de données grâce au faible nombre de niveaux ($O(\log n)$). Recherche : Efficace ($O(\log n)$) grâce à la minimisation des accès disque. Suppression : Complexe mais bien gérée avec des opérations de fusion et redistribution. | Utiliser pour des bases de données ou systèmes de fichiers volumineux. Préférer au ABR pour des données dépassant la mémoire principale. |

| **Structure** | **Résultats (Insertion, Recherche, Suppression)** | **Recommandations** |
|---|---|---|
| Tas (Heap) | Insertion : Performante, $O(\log n)$.<br>Recherche : Limitée au minimum ou maximum dans un tas binaire.<br>Suppression : Rapide pour supprimer la racine ($O(\log n)$). | Utiliser pour des files de priorité ou lorsque l'accès au min/max est essentiel.<br>Inadapté pour des recherches générales ou sur de grandes données. |

## 5.3 Recommandations Générales

| **Aspect** | **Recommandations** |
|---|---|
| Choix des structures | Liste chaînée pour des structures dynamiques simples.<br>ABR ou B-Tree pour des recherches fréquentes.<br>Tas pour des priorités. |
| Optimisation | Privilégier des arbres auto-équilibrés pour maintenir des performances optimales. |
| Mémoire et temps | Préférer le B-Tree pour les données volumineuses stockées en mémoire externe. |

# References


[1] Abdul Bari, *Tas - Tri par tas - Heapify - Files d'attente prioritaires*, [Vidéo en ligne]. Disponible sur YouTube via `https://www.youtube.com/watch?v=HqPJF2L5h9U`. Consulté le : 10/11/2024.

[2] Ritesh Gupta, *Analysis of Algorithms: Heapsort*, [PDF en ligne]. Disponible à : `https://fr.slideshare.net/RiteshGupta113/analysis-of-algorithmsheapsort`. Consulté le : 10/11/2024.

[3] Issa Raihana garga. "B-TREE 1". Universite de rennes [PDF en ligne]. Disponible à : `https://fr.scribd.com/document/666740544/B-TREE-1`. Consulté le : 17/11/2024.

[4] Amazing Algorithms. "B-Tree." Amazing Algorithms [PDF en ligne]. Disponible à : `https://amazingalgorithms.com/definitions/b-tree/`. Consulté le : 13/11/2024.

[5] TutorialsPoint, *B-tree properties and definition*, [Vidéo en ligne]. Disponible sur YouTube Via `https://youtu.be/r37E6RNxB6w`. Consulté le : 13/11/2024.

[6] Oggi AI - Artificial Intelligence Today,Binary Search Trees (BST)properties and definition, [Vid´eo en ligne]. Disponible sur YouTube Via : `https://www.youtube.com/watch?v=mtvbVLK5xDQ` Consulté le : 16/11/2024

[7] GeeksforGeeks, Complexity of Different Operations Binary Search Tree Disponible à : `https://www.geeksforgeeks.org/complexity-different-operations-binary-tree-binary-search-tree-avl-tree/?ref=oin_asr1`. Consulté le : 16/11/2024.

[8] ListeDouble, *la definition des Listes et leurs propriétées*. Disponible sur Google Via `https://chgi.developpez.com/dblist/`. Consulté le : 17/11/2024.

[9] DVD-MIAGE. "Liste Chainee." DVDMIAGE-Algo-Chapitre-10-Listes [PDF en ligne]. Disponible à : `https://miage.univ-nantes.fr/miage/DVD-MIAGEv2/Algo_files/DVDMIAGE_Algo_Chapitre_10_Listes.pdf`. Consulté le : 20/11/2024.