

République Algérienne Démocratique et Populaire  
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique



Université des Sciences et de la Technologie Houari Boumédiène

Faculté d'Electronique et d'Informatique  
Département Informatique

Master Systèmes Informatiques intelligents

Module : Conception et Complexité des Algorithmes

---

## Rapport de projet de TP

---

Réalisé par :

AIT AMARA Mohamed, 181831072170

BOUROUNA Rania, 181831052716

CHIBANE Ilies, 181831072041

HAMMAL Ayoub, 181831048403

Année universitaire : 2021 / 2022

# Chapitre 1

## Recherche séquentielle d'un élément dans un tableau

### 1.1 Description de l'objectif de l'algorithme

En informatique, un tableau est une structure de données représentant une séquence finie d'éléments définis par un index représentant leurs positions au sein du tableau. C'est un type de conteneur que l'on retrouve dans un grand nombre de langages de programmation et est l'un des plus utilisés dû à sa simplicité.

Les données du tableau étant accessible individuellement il est nécessaire de faire une recherche lorsque l'on souhaite accéder à une valeur spécifique du tableau. Cependant, lorsque la taille de la structure est grande il devient difficile d'y accéder efficacement. Dans ce chapitre nous allons voir la recherche séquentielle qui est une recherche très coûteuse en temps et nous allons aussi présenter une optimisation de cette recherche afin de gagner en complexité temporelle.

La recherche séquentielle ou recherche linéaire est un algorithme pour trouver une valeur dans une liste ou un tableau. Elle consiste simplement à considérer les éléments du tableau les uns après les autres, jusqu'à ce que l'élément soit trouvé, ou que toutes les cases aient été lues. Elle est aussi appelée recherche par balayage. (voir Figure 1.1).

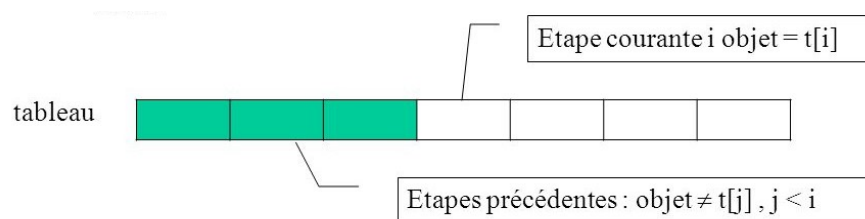


FIGURE 1.1 – Exemple graphique d'une recherche séquentielle

## 1.2 Fonctionnement de l'algorithme

La recherche séquentielle consiste à prendre les éléments du tableau les uns après les autres, jusqu'à avoir trouvé la cible, ou avoir épuisé le tableau. Elle ne demande aucune condition préalable pour le tableau en entrée ; par exemple : il n'est pas nécessaire que le tableau soit trié.

Nous pouvons le représenter via le pseudo code suivant :

**Fonction** sequentielle(Entrée : tab : tableau d'entier ; tailleTableau, valeur : entier ;)

**Variables :**

i : entier;

trouve : bool;

**début**

*trouve* ← *faux*;

*i* ← 0;

**tant que** *trouve* ← *false* *ET* *i* < *n* **faire**

**si** *tab*[*i*] == *valeur* **alors**

*trouve* ← *vrai*;

*i* ++;

**fin si**

**fin tq**

**si** *trouve* ← *vrai* **alors**

*retourner i* − 1;

**sinon**

*retourner* − 1;

**fin si**

**fin**

## 1.3 Calcul de complexité

### 1.3.1 Complexité temporelle

La complexité de la recherche séquentielle est toujours égale à :  $\mathcal{O}(n)$ .

le tableau suivant représente les temps d'exécution théorique en nanoseconde de l'algorithme selon la variation de la taille de l'expression :

N	10	50	100	500	1000	5000	10000	100000	1000000	10000000
t(ns)	10	50	100	500	1000	5000	10000	100000	1000000	10000000

La figure suivante (voir Figure 2.3) représente l'évolution du temps d'exécution théorique selon la longueur de l'expression.

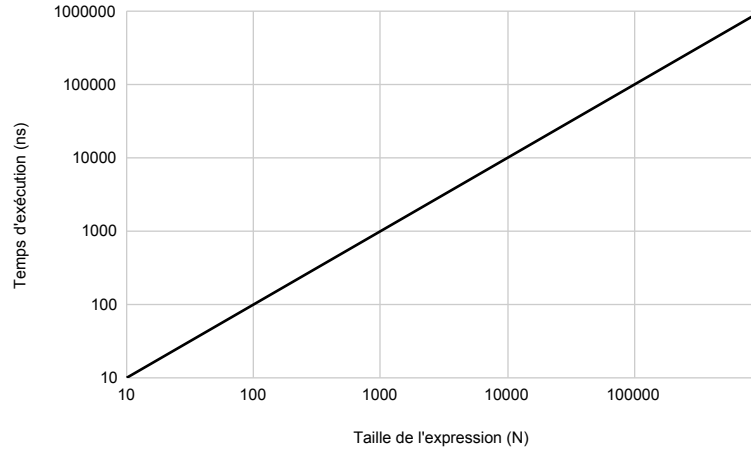


FIGURE 1.2 – Temps d'exécution théorique de l'algorithme de recherche linéaire

Depuis le graphe, on observe que le temps d'exécution évolue de manière linéaire avec l'évolution de la taille de l'expression.

### 1.3.2 Complexité spatiale

L'unique structure de données utilisée est un tableau d'entier a n éléments.

La taille d'un entier étant de 2 octets, la complexité spatiale est donc égale au produit de la taille du tableau et de la taille d'un entier :  $n * 2 \approx \mathcal{O}(n)$

## 1.4 Expérimentation

Le tableau suivant représente les temps d'exécution en nanoseconde de l'algorithme selon la variation de la taille de l'expression arithmétique en d'opérande.

N	10	50	100	500	1000	5000	10000	100000	1000000	10000000
t1(ns)	606	401	222	3224	12010	6864	106408	1066370	11130	11018
t2(ns)	866	1153	1851	5171	11383	52809	109357	1061730	3028000	38859000
t3(ns)	168	1066	1064	5714	10237	53084	2875	5070	4519680	56877700
t4(ns)	744	1119	1283	7111	747	2197	1504	11452	7636790	41331300
t5(ns)	588	785	829	1328	10459	3069	102717	1074940	1077270	54743600
t6(ns)	722	621	1402	1556	4204	10955	5509	1082590	1069420	11577
t7(ns)	638	499	375	5083	10259	6345	106589	493002	10463	38906200
t8(ns)	708	746	2192	7064	5008	8755	107439	1041410	1094640	40419900
t9(ns)	694	1225	1570	1118	11180	3275	122386	922361	10669	55940100
t10(ns)	666	1161	1504	2456	11349	10799	7470	1092110	7197500	40000300
Moyenne(ns)	2164	28704	92760	169733	341583	1047560	4598740	38774800	5626090000	37018600000

La figure suivante (voir Figure 1.3) représente l'évolution du temps d'exécution selon la longueur du tableau.

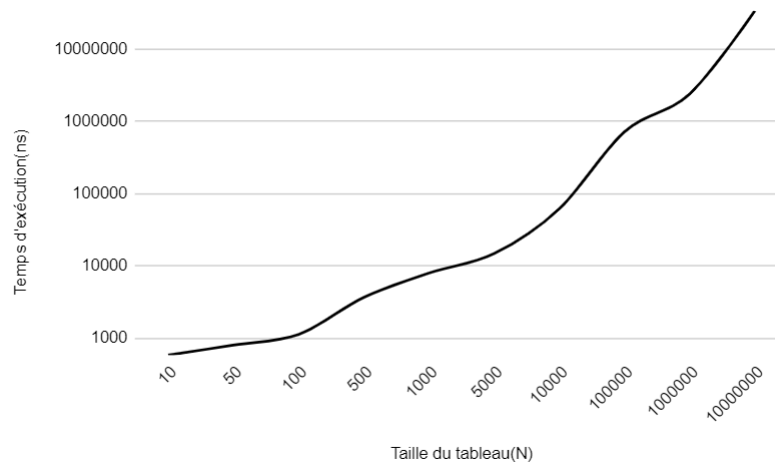


FIGURE 1.3 – Temps d'exécution du programme selon la longueur du tableau

Depuis le graphe, la courbe est sous forme d'un arc ascendant, on observe que le temps d'exécution évolue de manière presque linéaire avec l'augmentation de la taille du problème, ce qui correspond bien à la complexité théorique calculée auparavant. On a pas obtenu une droite linéaire car les testes étaient peu et aléatoires.

## 1.5 Amélioration

L'algorithme de recherche linéaire est coûteux en terme de temps, donc on a pensé à utiliser 2 processus qui vont rechercher la valeur en parallèle pour gagner du temps. Le premier processus commence à chercher depuis le début jusqu'à la moitié du tableau et le deuxième processus commence de la moitié jusqu'à la fin du tableau. Le premier d'entre eux qui trouve la valeur, il l'envoie au processus père.

**Fonction** sequentielleprocessus(Entrée : tab : tableau d'entier tailleTableau, valeur : entier)

**Variables :**

i, pos, status,debut,fin : entier;

*pid1, pid2, pid* : *pid<sub>t</sub>*;

trouve : bool;

**début**

*debut*  $\leftarrow$  0;

*fin*  $\leftarrow$  *tailleTableau*/2;

*trouve*  $\leftarrow$  *faux*;

*pid1* = *fork*();

// on crée le fils 1

**si** *pid1* == -1 **alors**

*ecrire*("y'a un problème lors de la création du fils 1");

**sinon**

    // on commence la recherche du début jusqu'au milieu du tableau

**pour** *i*  $\leftarrow$  *debut* **à** *fin* **faire**

**si** *tab*[*i*] == *valeur* **alors**

*trouve*  $\leftarrow$  *vrai*;

*pos*  $\leftarrow$  *i*;

**fin si**

**fin pour**

    // si le fils 1 trouve la valeur cherchée il affichera sa position  
    sinon il affiche un nombre négatif

*ecrire*(*pos*);

    // si le fils 1 a trouvé la valeur on renvoie sa position sinon on  
    renvoie -1

*exit*(*f* ? *pos* : -1);

**fin si**

*debut*  $\leftarrow$  *tailleTableau*/2 + 1;

*fin*  $\leftarrow$  *tailleTableau* - 1;

**fin**

**Fonction** sequentielleprocessus(Entrée : tab : tableau d'entier tailleTableau, valeur : entier)

**Variables :**

i, pos, status,debut,fin : entier;

*pid1, pid2, pid* : *pid<sub>t</sub>*;

trouve : bool;

**début**

```
    pid2 = fork();
    // on crée le fils 2
    si pid2 == -1 alors
        |   ecrire("y'a un problème lors de la création du fils 1");
    sinon
        |   // on commence la recherche du milieu jusqu'à la fin du tableau
        |   pour i ← debut à fin faire
        |       |   si tab[i] == valeur alors
        |           |   trouve ← vrai;
        |           |   pos ← i;
        |       |   fin si
        |       fin pour
        |       ecrire(pos);
        |       exit(f ? pos : -1);
    fin si
    tant que (pid = wait(&status))! = -1 faire
        |   // le père attend l'arrivée du premier fils
        |   ecrire("on attend le premier fils");
    fin tq
    si f alors
        |   retourner 0;
    sinon
        |   ecrire ("la valeur cherchée n'existe pas dans le tableau") ;
        |   retourner - 1;
    fin si
fin
```

## 1.6 Conclusion

L'algorithme de la recherche sequentielle(linéaire) se caractérise par son fonctionnement inconditionnel sur n'importe quel tableau. Par ailleurs, il est coûteux en temps. Pour remédier au problème temporelle, il est possible d'utiliser la technique de 2 processus expliquée auparavant. Cela optimise le temps de recherche par 2 mais il augmente la complexité spatiale par 3 : (père, fils1, fils2). Donc l'utilisation des processus est utile quand on a suffisamment d'espace mémoire et on veut accélérer la recherche.



# Chapitre 2

## Représentation d'une expression arithmétique en arbre binaire

### 2.1 Description de l'objectif de l'algorithme

Une expression arithmétique est une succession de caractères mathématiques notamment des nombres, des opérateurs mathématiques (+ addition, - soustraction, \* multiplication, / division et % modulo) et des symboles impropres pour indiquer la priorité( ( ) les parenthèses). Il faut noter cependant que les opérateurs mathématiques ne possèdent pas tous la même priorité; La multiplication et la division sont plus prioritaires que l'addition et la soustraction. Dans le cas où la priorité de deux opérateurs est la même, celui le plus à gauche dans l'expression arithmétique devient plus prioritaire que celui à droite. En considérant ces contraintes, une des représentations les plus adaptées pour décrire une expression arithmétique est l'arbre binaire.

Un arbre binaire est une structure de données complexe, il est caractérisé par un élément racine qui contient à son tour un chemin vers un ou deux autres éléments appelés fils droit et fils gauche. Chaque élément intermédiaire de l'arbre par la suite a la même structure que la racine, jusqu'à arriver aux éléments terminaux qui ne possèdent pas de fils et qu'on nomme les feuilles de l'arbre.

Nous pouvons alors par extrapolation représenter chaque opération arithmétique par un élément de l'arbre, telle que l'élément de l'arbre contient l'opérateur, et les deux opérandes sont contenus dans les deux fils de l'élément. Pour qu'un arbre binaire puisse représenter la structure d'une expression arithmétique correctement, ce dernier doit respecter l'ordre et la priorité des opérations. Les opérations les moins prioritaires sont en haut de l'arbre car elles dépendent du résultat des opérations plus prioritaires qui elles, sont plus bas dans l'arbre binaire (voir Figure 2.1).

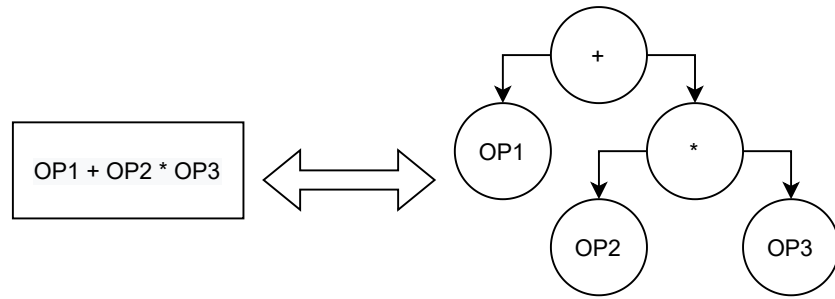


FIGURE 2.1 – Représentation d’une expression arithmétique en arbre binaire

## 2.2 Fonctionnement de l’algorithme

Le passage de l’expression arithmétique en arbre binaire passe par 2 étapes de traduction : l’analyse lexicale et puis la traduction dirigée par la syntaxe (voir Figure 2.2). L’expression arithmétique est d’abord lue du clavier sous forme de chaîne de caractères. Puis, un vecteur d’entités lexicales est généré à partir de cette chaîne. Enfin, on génère un arbre binaire à partir du vecteur d’entités lexicales.



FIGURE 2.2 – Les étapes de la traduction

Les structures de données utilisées dans les algorithmes qui vont suivre sont les suivantes :

**entité** : représente une entité lexicale et contient les champs type de l’entité et la valeur.

**noeud** : représente un élément d’un arbre et contient une valeur et des pointeurs vers les fils droit et gauche du noeud.

**arbre** : représente un arbre binaire et contient un pointeur vers la racine de l’arbre.

### 2.2.1 Analyse lexicale

Dans cette partie, on extrait les entités lexicales de la chaîne de caractères contenant l’expression arithmétique. On parcourt la chaîne de caractères depuis le début, et on génère les entités lexicales selon les caractères lus.

**Algorithme 1 : Analyse lexicale**

**Données :** expression : chaîne de caractères

**Résultat :** entites : vecteur d'entités

**Variables :**

e : entité;

i, j : entier;

**début**

Initialiser le vecteur entites à vide;

**pour**  $i \leftarrow 1$  **à** *expression.taille()* **faire**

**si** *expression[i]* **est un opérateur** **alors**

    // Reconnaître un opérateur

    e.type ← opérateur;

    e.operation ← type\_opération;

    entites.ajouter(e);

**sinon si** *expression[i]* **est une parenthèse** **alors**

    // Reconnaître une parenthèse

    e.type ← parenthèse;

    e.parenthèse ← type\_parenthèse;

    entites.ajouter(e);

**sinon si** *expression[i]* **est un chiffre ou un point** **alors**

    // Reconnaître un nombre

    j ← 1;

**tant que**  $i + j \leq \text{expression.taille}()$  **et** *expression[i]* **est un chiffre ou un point**

**faire** j ← j + 1;

    e.type ← nombre;

    e.nombre ← en\_nombre(expression.sous\_chaine(i, j));

    entites.ajouter(e);

    i ← i + j - 1;

**sinon**

    Lever une exception;

    // Erreur lexicale

**fin si**

**fin pour**

**fin**

## 2.2.2 Analyse syntaxico-sémantique ou traduction dirigée par la syntaxe

Nous utilisons l'algorithme de descente récursive pour parcourir le vecteur d'entités et générer l'arbre binaire. Dans cet algorithme chaque MGP (membre de gauche de production) est associé à

une fonction dans le programme. L'algorithme commence en appelant une première fois l'axiome de la grammaire et se termine une fois que tout le vecteur est parcouru.

La grammaire utilisée est comme suit :

$$E \rightarrow T + \{T\}^* | T - \{T\}^*$$

$$T \rightarrow F * \{F\}^* | F / \{F\}^*$$

$$F \rightarrow nb|(E)| + F | - F$$

**Algorithme 2 :** Descente récursive

**Données :** expression : chaîne de caractères

**Résultat :** entites : vecteur d'entités

**Variables :**

tc : entier;

bt : arbre;

**début**

    ct ← 0;

**si**  $tc = entites.taille()$  **alors** Lever une exception // Expression vide;

    bt ← e(entites, tc); // Appel de la fonction e

**retourner** bt;

**fin**

Cette fonction *e* traite les opérations d'addition et de soustraction qui sont moins prioritaires, et génère les noeuds correspondants. Le traitement des autres opérations est délégué à la fonction *t* qui va terminer son exécution avant la fonction *e* ; c-à-d. que les noeuds générés par les opérations plus prioritaires seront retournés à la fonction *e* qui va les ajouter dans les niveaux plus bas.

**Fonction** e(entites : vecteur d'entités, Entrée/Sortie tc : entier) : arbre

**Variable :**

bt, rt, lt : arbre;

e : entité;

**début**

bt ← t(entites, tc); // Appel de la fonction t

**tant que**  $tc \leq \text{entites.taille}()$  *et entites[tc] est un opérateur d'addition ou de soustraction* **faire**

    e ← entites[tc];

    lt ← bt;

    tc ← tc + 1;

**si**  $tc > \text{entites.taille}()$  **alors** Lever une exception // Erreur syntaxique;

    rt ← t(entites, tc); // Appel de la fonction t

    bt.racine ← e;

    bt.fils\_gauche ← lt;

    bt.fils\_droit ← rt;

**fin tq**

**retourner** bt;

**fin**

Cette fonction *t* traite les opérations de multiplication, de division et de modulo. Elle utilise la fonction *f* pour générer les noeuds des nombres et des sous-expressions plus prioritaires comme les parenthèses.

**Fonction**  $t$ (entites : vecteur d'entités, Entrée/Sortie tc : entier) : arbre

**Variable :**

bt, rt, lt : arbre;

e : entité;

**début**

bt ← f(entites, tc); // Appel de la fonction f

**tant que**  $tc \leq \text{entites.taille}()$  *et entites[tc] est un opérateur de multiplication ou de division ou de modulo* **faire**

    e ← entites[tc];

    lt ← bt;

    tc ← tc + 1;

**si**  $tc > \text{entites.taille}()$  **alors** Lever une exception // Erreur syntaxique;

    rt ← f(entites, tc); // Appel de la fonction f

    bt.racine ← e;

    bt.fils\_gauche ← lt;

    bt.fils\_droit ← rt;

**fin tq**

**retourner** bt;

**fin**

Cette fonction  $f$  génère les noeuds des nombres et des opérateurs unaires, elle repasse le contrôle à la fonction  $e$  en cas d'utilisation de parenthèses pour traiter la sous-expression contenue dedans.

**Fonction** f(entites : vecteur d'entités, Entrée/Sortie tc : entier) : arbre

**Variable :**

bt, rt, lt : arbre;

e, gauche, droit : entité;

**début**

**si** entites[tc] est une parenthèse ouvrante **alors**

        tc ← tc + 1;

**si** tc > entites.taille() **alors** Lever une exception                   // Erreur syntaxique;

        bt ← e(entites, tc);   // Appel de la fonction e

**si** tc ≤ entites.taille() et entites[tc] est une parenthèse fermante **alors**

            tc ← tc + 1;

**sinon**

            Lever une exception   // Erreur Syntaxique

**fin si**

**sinon si** entites[tc] est un opérateur d'addition ou de soustraction **alors**

        // Cas d'un opérateur unaire

        e ← entites[tc];

        gauche.type ← type\_nombre;

        gauche.nombre ← 0;

        tc ← tc + 1;

**si** tc > entites.taille() **alors** Lever une exception                   // Erreur syntaxique;

        rt ← f(entites, tc);   // Appel de la fonction f

        lt.racine ← gauche;

        bt.racine ← e;

        bt.fils\_gauche ← lt;

        bt.fils\_droit ← rt;

**sinon**

        // Cas d'un nombre (feuille de l'arbre)

        e ← entites[tc];

        bt.racine ← e;

        tc ← tc + 1;

**fin si**

**retourner** bt;

**fin**

## 2.3 Calcul de complexité

### 2.3.1 Complexité temporelle

La complexité de l'analyse lexicale est toujours égale à la longueur de la chaîne dans tous les cas c-à-d. :  $\mathcal{O}(n)$  tel que  $n$  est la longueur de l'expression en de caractères.

La complexité de l'analyse syntaxique est quant à elle égale à la longueur du vecteur d'entités lexicales, car il est parcouru une seule fois, c-à-d. :  $\mathcal{O}(n')$  tel que  $n'$  est la longueur du vecteur d'entités.

La complexité temporelle de l'algorithme devient alors :  $\mathcal{O}(n + n')$ . Sachant que  $n > n'$  (le nombre de caractères est supérieur au nombre d'opérandes) on peut remplacer  $n'$  par  $n$ , et la formule devient :  $\mathcal{O}(n)$  qui est une complexité linéaire.

Le tableau suivant représente les temps d'exécution théorique en nanoseconde de l'algorithme selon la variation de la taille de l'expression :

N	10	50	100	500	1000	5000	10000	100000	1000000	10000000
t(ns)	10	50	100	500	1000	5000	10000	100000	1000000	10000000

La figure suivante (voir Figure 2.3) représente l'évolution du temps d'exécution théorique selon la longueur de l'expression.

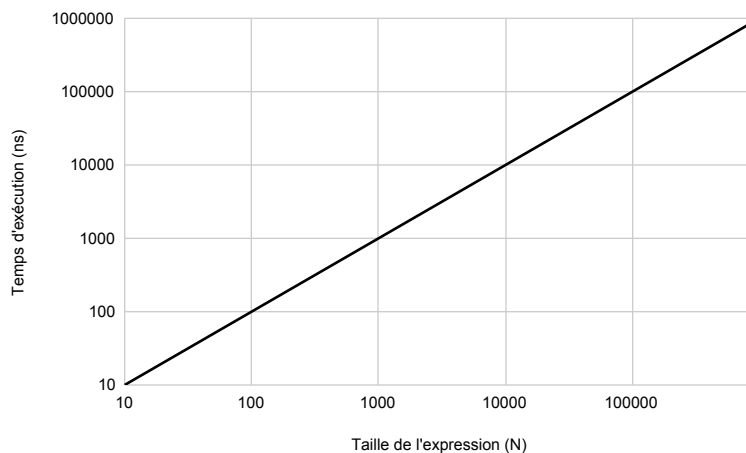


FIGURE 2.3 – Temps d'exécution théorique de l'algorithme de représentation d'expression arithmétique en arbre binaire

Depuis le graphe, on observe que le temps d'exécution évolue de manière linéaire avec l'évolution de la taille de l'expression.

### 2.3.2 Complexité spatiale

L'expression arithmétique est d'abord stockée dans une chaîne de caractères de longueur  $n$ , puis dans un vecteur d'entités de longueur  $n'$ , puis dans un arbre binaire qui contient lui aussi  $n'$  éléments.



Sachant qu'une entité contient le type de l'entité et la valeur en elle-même, cependant la valeur est stockée comme un nombre réel qui prend 8 octets comparé à la représentation en chaîne de caractères où chaque caractère est sous 1 octet.

Donc la taille d'un élément d'une chaîne de caractères est 1 unité (octet), la taille d'une entité est 9 unités, et la taille d'un élément d'un arbre est égale à  $9 + 4 * 2$  unités (la taille d'un pointeur est égale à 4 unités) donc 17 unités.

La complexité spatiale est égale à la somme des trois complexités (taille de la chaîne + taille du vecteur + taille de l'arbre) :  $n * 1 + n' * 9 + n' * 17 = n + n' * 26 \approx \mathcal{O}(n + n')$

## 2.4 Expérimentation

Le tableau suivant représente les temps d'exécution en nanoseconde de l'algorithme selon la variation de la taille de l'expression arithmétique.

N	10	50	100	500	1000	5000	10000	100000	1000000
t1(ns)	177584	437769	381575	418252	686466	3111420	6786260	60986100	642610000
t2(ns)	101653	193648	339100	354777	2322770	3307840	20424500	59967200	639145000
t3(ns)	31972	47574	216601	354751	2115200	3375350	7450650	60956700	639422000
t4(ns)	100708	149167	259485	337421	820084	3283810	19698800	61631500	641354000
t5(ns)	24343	140434	95821	345987	683119	5158840	7575520	61750100	640897000
t6(ns)	30901	161277	337395	807634	671850	3292160	21818400	61473700	641686000
t7(ns)	52941	140159	288308	329206	693785	4360970	8418610	60870500	667895000
t8(ns)	69185	163825	242197	520649	2050330	5460620	22419800	63008000	639987000
t9(ns)	69510	97011	93270	356979	692529	5279490	6420450	61725500	640873000
t10(ns)	83819	142928	92484	1071680	803473	13108600	6558390	60088300	639895000
Moyenne(ns)	74262	167379	234624	489734	1153961	4973910	12757138	61245760	643376400

La figure suivante (voir Figure 2.4) représente l'évolution du temps d'exécution selon la longueur de l'expression arithmétique.

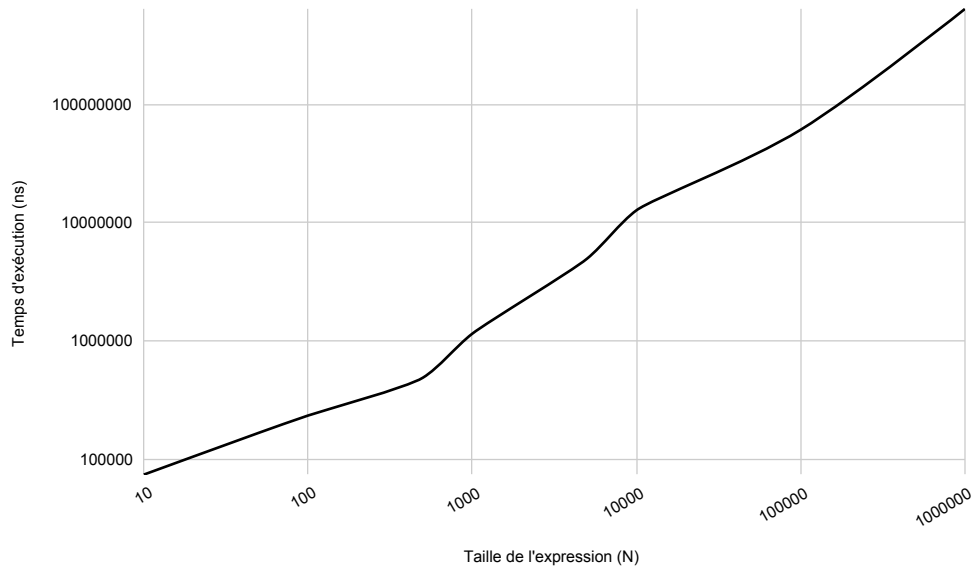


FIGURE 2.4 – Temps d'exécution du programme selon la longueur de l'expression

Depuis le graphe, la courbe est sous forme d'une droite, on observe que le temps d'exécution évolue de manière linéaire avec l'augmentation de la taille du problème, ce qui correspond bien à la complexité théorique calculée auparavant.

## 2.5 Améliorations

Une amélioration possible de l'algorithme consiste à fusionner l'analyse lexicale et syntaxique dans une même itération, c-à-d. que l'entité lexicale est injectée dans l'analyse syntaxique dès qu'elle est reconnue par l'analyse lexicale, sans effectuer une deuxième fois le parcours de l'expression. Cela va diminuer la complexité temporelle de moitié, et la complexité spatiale d'un tiers.

## 2.6 Conclusion

L'algorithme de la descente récursive propose une complexité optimale pour la représentation de l'expression arithmétique en arbre binaire, égale à la longueur de l'expression arithmétique, car il parcourt l'expression un nombre minimal de fois (une seule fois). Nous avons bien vu pendant les expériences que l'évolution du temps d'exécution d'une telle complexité est relativement contrôlé et suit une courbe droite et linéaire.

# Chapitre 3

## Recherche dichotomique d'un élément dans un tableau

### 3.1 Description de l'objectif de l'algorithme

En informatique, un tableau est une structure de données représentant une séquence finie d'éléments défini par un index représentant sa position au sein du tableau nous permettant d'y accéder. C'est un type de conteneur que l'on retrouve dans un grand nombre de langages de programmation et est l'un des plus utilisés du de sa simplicité.

Les données du tableau étant accessible individuellement il est nécessaire de faire une recherche lorsque l'on souhaite accéder a une valeur spécifique du tableau cependant lorsque la taille de la structure est grande il devient difficile d'y accéder efficacement, c'est pour cela que de nombreux algorithmes ont été conçu afin d'optimiser cette tâche est l'un de ses algorithmes les plus performants est celui de la recherche dichotomique.

La recherche dichotomique ou recherche par dichotomie, est un algorithme de recherche pour trouver la position d'un élément dans un tableau. La seule condition a son application étant que le tableau soit trié, il est utilisable dans de nombreuses problématiques. Son principe consiste comparer l'élément avec la valeur de la case au milieu du tableau, si les valeurs sont égales, on met fin à l'exécution, sinon si la valeur recherchée est inférieure à la valeur de la case au milieu, on recommence dans la moitié du tableau contenant les valeurs plus petites que celle située au milieu du tableau et dans le cas contraire on prend la moitié contenant les valeurs supérieures, et ceci jusqu'à avoir trouvé la valeur souhaitée ou avoir un sous tableau n'ayant qu'une seule valeur empêchant de continuer la recherche dans le cas où la valeur n'existe pas dans le tableau. (voir Figure 3.1).

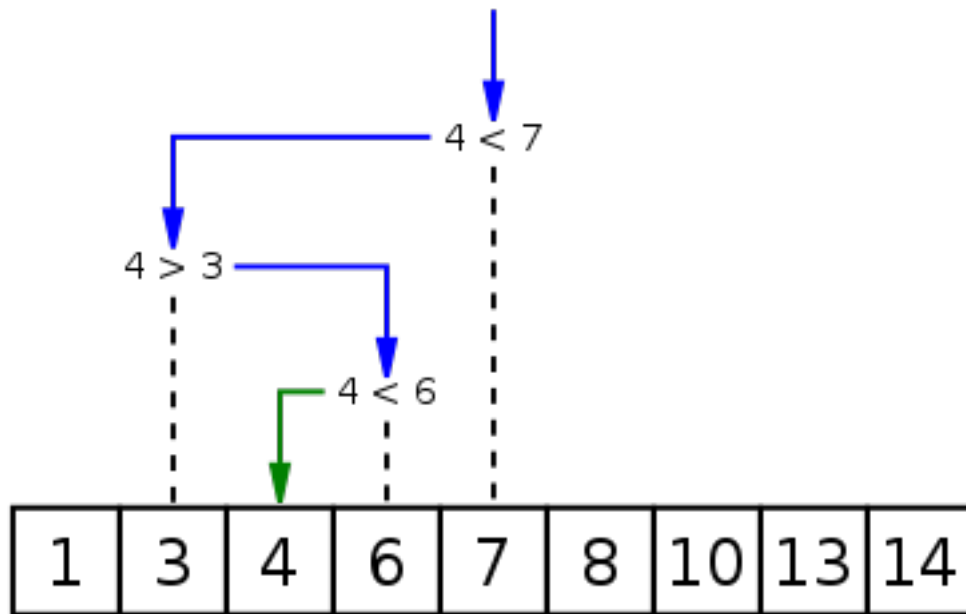


FIGURE 3.1 – Exemple graphique d’une recherche dichotomique

## 3.2 Fonctionnement de l’algorithme

Il nous est possible de distinguer 2 étapes distinctes et essentielles au bon déroulement de cet algorithme la première étant le tri du tableau, car comme précisé précédemment la recherche par dichotomie nécessite que le tableau soit trié et ne pouvant pas garantir de recevoir un tableau trié en entrée il est nécessaire de le trier au préalable avant de commencer la recherche afin de pouvoir le plus de cas que possible, par la suite la seconde étape consiste à appliquer la recherche dichotomique.

### 3.2.1 Tri du tableau

Dans cette partie, on applique un tri ascendant sur tableau et pour cela nous appliquons l’algorithme de Tri Fusion.

Le tri par fusion aussi appeler tri dichotomique est un exemple classique d’algorithme de division pour régner. L’opération principale de l’algorithme est la fusion, qui consiste à réunir deux listes triées en une seule. L’efficacité de l’algorithme vient du fait que deux listes triées peuvent être fusionnées en temps linéaire (voir Figure 3.2). On peut résumer son fonctionnement en deux étapes :

1. Divisez la liste non triée en sous-listes jusqu’à ce qu’il y ait  $N$  sous-listes avec un élément dans chacune ( $N$  est le nombre d’éléments dans la liste non triée).
2. Fusionnez les sous-listes deux à la fois pour produire une sous-liste triée, répétez cette opération jusqu’à ce que tous les éléments soient inclus dans une seule liste.

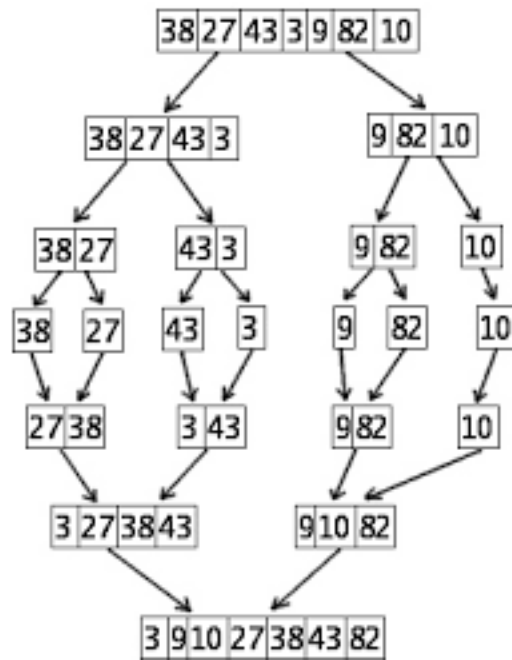


FIGURE 3.2 – Exemple graphique d'un tri dichotomique

Nous pouvons le représenter via le pseudo code suivant :

**Fonction** Fusion(Entrée : tab : tableau d'entier ; droite, gauche, milieu : entier ;)

**Variables :**

TabG, TabD : tableau d'entier;

SousTab1, SousTab2, SousTab1Index, SousTab2Index, SousTabFusionIndex : entier;

**début**

*SousTab1*  $\leftarrow$  *milieu* - *gauche* + 1;

*SousTab2*  $\leftarrow$  *droite* - *milieu*;

**pour** *i*  $\leftarrow$  1 **à** *SousTab1* **faire**

    | *tabG*[*i*]  $\leftarrow$  *tab*[*gauche* + *i*];

**fin pour**

**pour** *j*  $\leftarrow$  1 **à** *SousTab2* **faire**

    | *tabG*[*j*]  $\leftarrow$  *tab*[*mid* + 1 + *j*];

**fin pour**

*SousTab1*  $\leftarrow$  0;

*SousTab2*  $\leftarrow$  0;

*SousTabFusionIndex*  $\leftarrow$  *gauche*;

**tant que** *SousTab1Index* < *SousTab1* **et** *SousTab2Index* < *SousTab2* **faire**

**si** *tabG*[*SousTab1Index*] <= *tabD*[*SousTab2Index*] **alors**

        | *tab*[*SousTabFusionIndex*]  $\leftarrow$  *tabG*[*SousTab2Index*];

        | *SousTab1Index* ++;

**sinon**

        | *tab*[*SousTabFusionIndex*]  $\leftarrow$  *tabD*[*SousTab2Index*];

        | *SousTab2Index* ++;

**fin si**

    | *SousTabFusionIndex* ++;

**fin tq**

**tant que** *SousTab1Index* < *SousTab1* **faire**

    | *tab*[*SousTabFusionIndex*]  $\leftarrow$  *tabG*[*SousTab1Index*];

    | *SousTab1Index* ++;

    | *SousTabFusionIndex* ++;

**fin tq**

**tant que** *SousTab2Index* < *SousTab2* **faire**

    | *tab*[*SousTabFusionIndex*]  $\leftarrow$  *tabG*[*SousTab2Index*];

    | *SousTab1Index* ++;

    | *SousTabFusionIndex* ++;

**fin tq**

**fin**

**Fonction** TriFusion(Entrée : tab : tableau d'entier ; debut, fin : entier ;)

**Variables :**

milieu : entier;

**début**

```
// On divise le tableau en 2 de manière recursive puis on les tri avant  
de les fusionner
```

```
si debut >= fin alors
```

```
    retour;
```

```
fin si
```

```
// On calcule l'index du milieu du tableau
```

```
milieu ← debut + (fin - debut)/2;
```

```
TriFusion(tab, debut, milieu);
```

```
TriFusion(tab, milieu + 1, fin);
```

```
// On utilise la fonction fusion pour trier puis fusionner les sous  
tableaux en un seul tableau trié
```

```
Fusion(tab, debut, mid, fin);
```

**fin**

Afin d'optimiser le déroulement du tri, nous utilisons 2 fonctions distinctes. Une fonction TriFusion qui divise le tableau récursivement et une fonction Fusion qui elle trie les sous tableaux avant de les fusionner à nouveau

### 3.2.2 Recherche dichotomique

La recherche dichotomique consiste à rechercher dans un tableau trié en divisant l'intervalle de recherche en deux. On commence par un intervalle couvrant tout le tableau. Si la valeur de la clé de recherche est inférieure à l'élément situé au milieu de l'intervalle, on limite l'intervalle à la moitié inférieure. Sinon, le réduire à la moitié supérieure. On vérifie à plusieurs reprises jusqu'à ce que la valeur soit trouvée ou que l'intervalle soit vide.

Nous pouvons le représenter via le pseudo code suivant :

**Fonction** dichotomie(Entrée : *tab* : tableau d'entier ; *n*, *r* : entier ; Sortie : *pos*)

**Variables :**

*debut*, *fin*, *milieu* : entier;

**début**

```
// si la valeur recherchée est superieur ou inferieur au bornes du
// tableau on retourne -1 si elle est égale a une des bornes on retourn
// leur positions
si  $r < tab[0]$  ou  $r > tab[n - 1]$  alors
|    $pos \leftarrow -1$ ;
|   retour pos;
fin si
si  $tab[0] == r$  alors
|    $pos \leftarrow 0$ ;
|   retour pos;
fin si
si  $tab[n - 1] == r$  alors
|    $pos \leftarrow n - 1$ ;
|   retour pos;
fin si
tant que  $fin \geq debut$  faire
|   // Calcul de la position de l'élément du milieu
|    $i \leftarrow (debut + fin)/2$ ;
|   // Si l'élément du milieu est l'élément recherché on retourne sa
|   // position
|   si  $tab[i] == r$  alors
|   |    $pos \leftarrow i$ ;
|   |   retour pos;
|   // Si la valeur recherchée est plus petite que la valeur du l'élément
|   // du milieu Alors on regarde le sous-tableau de gauche
|   sinon si  $tab[i] > r$  alors
|   |    $fin = i - 1$ ;
|   // sinon on regarde le sous-tableau de droite
|   sinon
|   |    $debut = i + 1$ ;
|   fin si
fin tq
```

**fin**



## 3.3 Calcul de complexité

### 3.3.1 Complexité temporelle

La complexité du tri dichotomique est toujours égale à :  $\mathcal{O}(n \log(n))$  que ce soit dans le cas meilleur ou le cas tel que  $n$  est la longueur du tableau.

La complexité de la recherche dichotomique est quant à elle égale à  $\mathcal{O}(\log(n'))$  tel que  $n'$  est la longueur du tableau.

La complexité temporelle de l'algorithme devient alors :  $\mathcal{O}(n \log(n) + \log(n')) = \mathcal{O}(\log(n^n + n'))$ .

Le tableau suivant représente les temps d'exécution théorique en nanoseconde de l'algorithme selon la variation de la taille du tableau

N	10	50	100	500	1000	5000	10000	100000	1000000	10000000
t(ns)	11	87	202	1352	3003	18499	40004	500005	6000006	70000007

La figure suivante (voir Figure 3.3) représente l'évolution du temps d'exécution théorique selon la longueur du tableau.

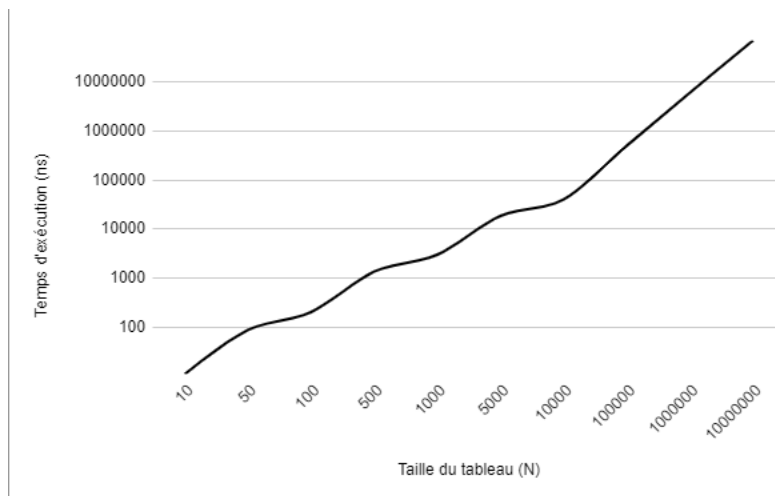


FIGURE 3.3 – Temps d'exécution théorique du programme selon la longueur du tableau

Depuis le graphe, on observe que le temps d'exécution évolue de manière linéarithmique avec l'augmentation de la taille du problème.

### 3.3.2 Complexité spatiale

L'expression arithmétique est d'abord stockée dans une chaîne de caractères de longueur  $n$ , puis dans un vecteur d'entités de longueur  $n'$ , puis dans un arbre binaire qui contient lui aussi  $n'$  éléments.

L'unique structure de données utilisée est un tableau d'entier a n éléments.

La taille d'un entier étant de 2 octets la complexité spatiale est égale au produit de la taille du tableau et de la taille d'un entier :  $n * 2 \approx \mathcal{O}(n)$

### 3.4 Expérimentation

Le tableau suivant représente les temps d'exécution expérimentale en nanoseconde de l'algorithme selon la variation de la taille du tableau.

N	10	50	100	500	1000	5000	10000	100000	1000000	10000000
t1(ns)	2164	28704	92760	169733	341583	1047560	4598740	35465600	5626090000	37018600000
t2(ns)	6674	35401	75401	251223	600691	1642120	3621170	34767200	3774520000	33572800000
t3(ns)	5654	33403	53403	275397	771243	1225350	2874540	40542400	2998740000	36542700000
t4(ns)	4233	42553	82553	251567	477149	988353	4714569	37218300	3411720000	36325400000
t5(ns)	5904	58861	58861	163133	348047	1253250	3148226	37992800	3845760000	29451700000
t6(ns)	5645	20634	70634	252684	801493	1991560	2132380	40874100	298370000	401745900000
t7(ns)	3820	43164	64307	251082	481145	1035130	3258380	37476600	3220570000	36945100000
t8(ns)	5641	32783	57953	283887	428472	2290770	2811460	28514100	3859450000	28416500000
t9(ns)	7242	47059	65421	258297	429683	1988180	3223750	3589600	3042580000	35473800000
t10(ns)	5242	31262	41654	233613	530896	1041460	3706200	45122700	1687400000	33447800000
Moyenne(ns)	5221	37382	66294	239061	521040	1271437	3429518	37552644	3176520000	70894030000

La figure suivante (voir Figure 3.4) représente l'évolution du temps d'exécution selon la longueur du tableau.

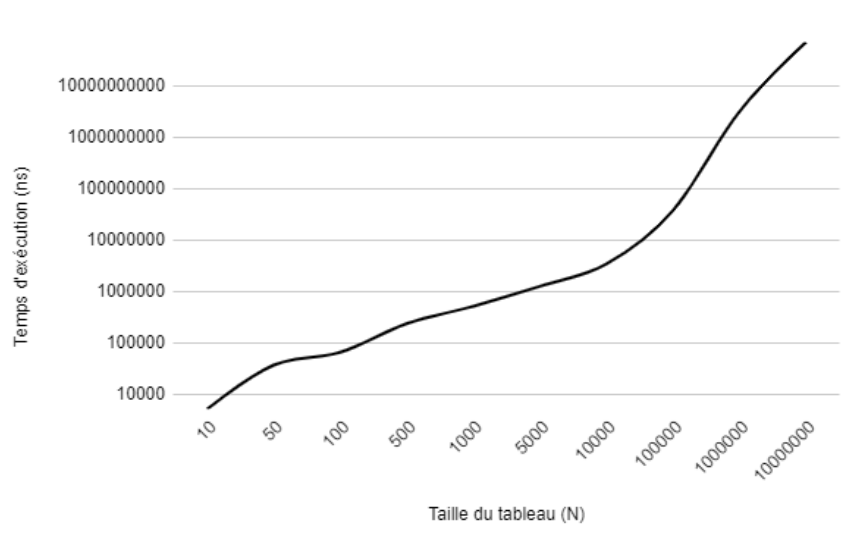


FIGURE 3.4 – Temps d'exécution du programme selon la longueur du tableau

Depuis le graphe, on observe que le temps d'exécution évolue de manière linéarithmique avec l'augmentation de la taille du problème, ce qui correspond bien à la complexité théorique et d'exécution théorique calculée auparavant.

## 3.5 Conclusion

L'algorithme de la recherche dichotomique propose une complexité optimale pour la recherche dans un tableau, cependant sa dépendance d'une fonction de tri afin de pouvoir fonctionner. Quel que soit le tableau donné impacte négativement sa complexité, dans notre utilisation d'une fonction de tri dichotomique aura fait passer notre algorithme d'une complexité logarithmique à une complexité linéarithmique. Nous pouvons en conclure que la recherche par dichotomie est effectivement une bonne option, mais qu'afin de pouvoir l'utiliser à son plein potentiel il est nécessaire de l'utiliser sur des tableaux préalablement triés ou accompagné d'une fonction de tri ayant une complexité égale ou meilleure à elle.

# Chapitre 4

## Suppression d'un élément dans un arbre binaire de recherche

### 4.1 Description de l'objectif de l'algorithme

Les arbres binaires représentent une structure de données très utilisée pour effectuer des tâches en informatique, d'une part parce que ce type de données permet de stocker des données volumineuses facilement accessibles, et d'autre part, les informations sont souvent hiérarchisées, et peuvent être représentées naturellement sous une forme arborescente.

Un arbre est un ensemble organisé de nœuds dans lequel chaque nœud a un seul père, à l'exception de la racine qui est un nœud sans père.

Si le nœud  $p$  est le père du nœud  $f$ , nous dirons que  $f$  est un fils de  $p$ , et si le nœud  $p$  n'a pas de fils nous dirons que c'est une feuille. Chaque nœud porte une valeur (également appelée clé ou étiquette) et deux pointeurs, un qui pointe son fils gauche et l'autre sur le droit.

Un arbre binaire de recherche (ABR) est un arbre qui permet de représenter un ensemble de valeurs ayant une relation d'ordre. C'est à dire que pour tout nœud de cet arbre, sa valeur est strictement plus grande que les valeurs figurant dans son sous-arbre gauche et strictement plus petite que les valeurs figurant dans son sous-arbre droit. Cela implique qu'une valeur n'apparaît au plus qu'une seule fois dans un arbre de recherche. (voir Figure 4.1).

Les opérations caractéristiques sur les arbres binaires de recherche sont l'insertion, la suppression, et la recherche d'une valeur. Le cout de ces dernières dépend du degré d'équilibre de l'arbre.

La suppression d'un élément dans un arbre binaire de recherche est une opération compliquée. En effet, on doit d'abord trouver le noeud, le supprimer, ensuite nous devons faire le nécessaire pour conserver la structure de l'arbre.

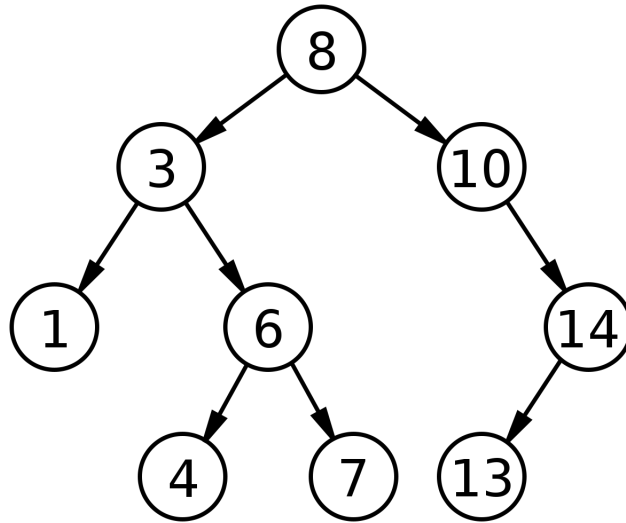


FIGURE 4.1 – Exemple graphique d'un arbre binaire de recherche

## 4.2 Fonctionnement de l'algorithme

L'opération de suppression d'un nœud dépend du nombre de ses fils dans l'arbre. Les différents cas de figure possibles sont les suivants

### 4.2.1 Cas 1 : Cas d'une feuille. Le nœud à supprimer n'a pas de fils

Il suffit d'enlever le nœud en modifiant le lien du père, s'il existe. Sinon l'arbre devient un arbre vide. (voir Figure 4.2).



FIGURE 4.2 – Exemple de suppression d'une feuille

### 4.2.2 Cas 2 : Cas d'un unique fils. Le nœud à supprimer à un seul fils

On décroche le nœud de l'arbre En reliant directement son père et son fils. Ensuite on libère l'espace qu'occupe le nœud à supprimer. Si son père n'existe pas, l'arbre est réduit au fils unique du nœud supprimé. (voir Figure 4.3).



FIGURE 4.3 – Exemple de suppression d'un noeud à un seul fils

### 4.2.3 Cas 3 : Cas de deux fils. Le noeud à supprimer à deux fils

On cherche le successeur en ordre du noeud. Ensuite, on copie le contenu du successeur d'ordre dans le noeud et on supprime le successeur d'ordre. Notez que le prédécesseur d'ordre peut également être utilisé.

Il est important de noter que le successeur d'ordre peut être obtenu en trouvant la valeur minimale dans le sous-arbre de droite du noeud à supprimer et que le prédécesseur d'ordre est le maximum de son sous-arbre gauche.(voir Figure 4.4).

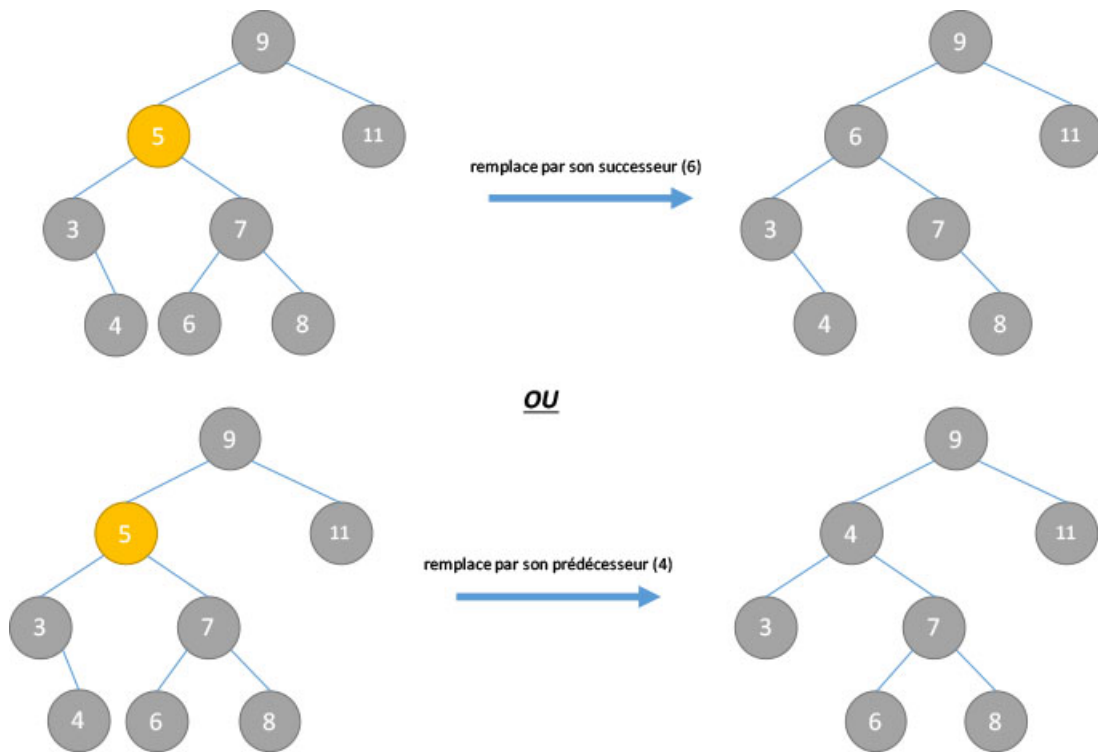


FIGURE 4.4 – Exemple de suppression d'un noeud à deux fils

**Fonction** `supprimerrec`(Entrée : `racine` : arbre, `valeur` : entier) : arbre

**Variables :**

`Temp` : arbre;

**début**

**si** `racine = Nil` **alors**

        retour Nil;

**fin si**

**si** `valeur < racine.valeur` **alors**

        // Si la valeur cherchée est inférieure à l'élément en cours, on  
        supprime à gauche

`racine.filsg`  $\leftarrow$  `supprimerrec(racine.filsg, valeur)`;

**fin si**

**sinon si** `valeur > racine.valeur` **alors**

        // Si la valeur cherchée est supérieure à l'élément en cours, on  
        supprime à droite

`racine.filsd`  $\leftarrow$  `supprimerrec(racine.filsd, valeur)`;

**fin si**

**sinon**

        // Si la valeur est trouvée, on distigue 3 cas

        // Le noeud n'a pas de fils

**si** `racine.filsg = Nil` et `racine.filsd = Nil` **alors**

            retour Nil;

**fin si**

**sinon si** `racine.filsd = Nil` **alors**

            // Si le noeud a un fils gauche seulement

`Temp`  $\leftarrow$  `racine.filsg` ;

            Libérer(`racine`);

            retour `Temp` ;

**fin si**

**sinon si** `racine.filsg = Nil` **alors**

            // Si le noeud a un fils à droite seulement

`Temp`  $\leftarrow$  `racine.filsd` ;

            Libérer(`racine`);

            retour `Temp` ;

**fin si**

`Temp`  $\leftarrow$  `Min(racine.filsd)`;

        // Peut etre remplacé par `Temp`  $\leftarrow$  `Max(racine.filsg)`;

**fin si**

**fin**

**Fonction** *supprimerrec*(Entrée : *racine* : arbre, *valeur* : entier) : arbre

**début**

```

    racine.valeur ← Temp.valeur ;
    racine.filsd ← supprimerrec(racine.filsd, Temp.valeur) ;
    // Peut etre remplacé par
        racine.filsg ← supprimerrec(racine.filsg, Temp.valeur) ;

    retour racine ;

```

**fin**

Comme nous l'avons mentionné avant, quand on veut supprimer un nœud qui a deux fils, nous devons écraser sa valeur en optant pour l'une des deux options : soit on cherche le successeur en ordre du nœud à supprimer, soit son prédécesseur en ordre. Il est à noter que le successeur en ordre du nœud représente le minimum dans son sous-arbre à droite, et que son prédécesseur représente le maximum de son sous-arbre gauche. Voici donc les deux fonctions que nous pourrions utiliser pour effectuer ces deux tâches :

**Fonction** *Min*(Entrée : *racine* : arbre) : arbre

**Variables :**

*min* : arbre;

**début**

```

    min ← racine ;
    tant que min <> Nil et min.filsg <> Nil faire
        | min ← min.filsg ;
    fin tq
    retour min ;

```

**fin**

**Fonction** *Max*(Entrée : *racine* : arbre) : arbre

**Variables :**

*max* : arbre;

**début**

```

    max ← racine ;
    tant que max <> Nil et max.filsd <> Nil faire
        | max ← max.filsd ;
    fin tq
    retour max ;

```

**fin**

Puisque l'opération de suppression est une tâche complexe, nous avons, en premier lieu opter pour la méthode récursive. Celle-ci nous permet d'avoir une vue globale sur les cas auxquels nous pourrions être confrontés et comment procéder pour garder la structure de l'arbre intacte même après la suppression.



Maintenant, nous allons écrire l'algorithme itératif de suppression pour comprendre comment s'effectue la suppression en détail.

**Fonction** supprimer(Entrée : racine : arbre, valeur : entier) : arbre

**Variables :**

inter, actuel, min, pere : arbre;

**début**

*actuel* ← *racine* ;

*pere* ← *Nil* ;

**tant que** *actuel* <> *Nil* et *actuel.valeur* <> *valeur* **faire**

*pere* ← *actuel* ;

**si** *valeur* < *actuel.valeur* **alors**

*actuel* ← *actuel.filsg*;

**sinon**

*actuel* ← *actuel.filsd*;

**fin si**

**fin si**

**fin tq**

**si** *actuel* = *Nil* **alors**

        Ecrire("La valeur que vous voulez supprimer n'existe pas dans l'arbre");

        retour *Nil*;

**fin si**

**si** *actuel.filsg* = *Nil* ou *actuel.filsd* = *Nil* **alors**

        // Si le noeud a au plus un fils

**si** *actuel.filsg* = *Nil* **alors**

*inter* ← *actuel.filsd*;

**fin si**

**sinon**

*inter* ← *actuel.filsg*;

**fin si**

**si** *pere* = *Nil* **alors**

            // Si le noeud est une racine

            retour *inter*;

**fin si**

**si** *actuel* = *pere.filsd* **alors**

            // Si le noeud est une fils à droite

*pere.filsd* = *inter*;

**fin si**

**sinon**

*pere.filsg* = *inter*;

**fin si**

        Libérer(*actuel*);

**fin si**

**fin**

**Fonction** supprimer(Entrée : racine : arbre, valeur : entier) : arbre

```
sinon
  // Si le noeud à supprimer a deux fils
  // On commence par chercher le successeur en ordre
  min ← actuel.filsd;
  pere ← racine;
  tant que min.filsg <> Nil faire
    pere ← min;
    min ← min.filsd;
  fin tq
  si pere <> racine alors
    // On peut mettre fils droit du successeur comme fils gauche du père
    // du successeur
    pere.filsg ← min.filsd;
  fin si
  sinon
    actuel.filsd ← min.filsd;
  fin si
  actuel.valeur ← min.valeur;
  Libérer(min);
fin si
retour racine;
```

## 4.3 Calcul de complexité

### 4.3.1 Complexité temporelle

Dans le meilleur cas, nous aurons un ABR équilibré, c'est-à-dire que le nombre des fils à gauche est le même qu'à la droite. Ainsi, l'opération de suppression se fera que sur la moitié de l'ABR équilibré, ce qui donne une complexité de  $\mathcal{O}(\log_2(n))$ .

Nous avons obtenu une complexité temporelle logarithmique parce que la suppression dans un arbre binaire de recherche est un cas où un problème de taille  $n$  est divisé en sous-problèmes de taille  $n/2$  jusqu'à atteindre un problème de taille 1.

Le calcul de la complexité se fait donc de la manière suivante :

$$\frac{a}{b^x} = 1 \quad [Dans \text{ un arbre binaire } b = 2]$$

*i.e :  $n = 2^x$       qui est  $\log_2 n$  par définition de la fonction logarithme*

Dans le cas moyen, la complexité temporelle est de l'ordre de la hauteur de l'arbre binaire de recherche. Puisque ce dernier est en moyenne  $\mathcal{O}(\log_2(n))$ , la complexité temporelle du cas moyen est de l'ordre  $\mathcal{O}(\log_2(n))$ .

Par ailleurs, si on traverse de la racine à une feuille, c'est-à-dire toute la hauteur  $h$  de l'arbre. Si ce dernier n'est pas équilibré, on sera amené à parcourir tous les nœuds et la hauteur de l'arbre deviendra  $n$ . par conséquent la complexité temporelle dans le pire des cas de l'opération de suppression est  $\mathcal{O}(n)$ .

Le tableau suivant représente les temps d'exécution théorique en nanoseconde de l'algorithme de suppression de la racine selon la variation de la taille de l'arbre.

N	10	50	100	500	1000	5000	10000	100000	1000000	10000000
t(ns)	490	560	720	830	1130	1280	1380	1590	2489	2680

La figure suivante (voir Figure 4.5) représente l'évolution du temps d'exécution théorique selon la taille de l'arbre binaire de recherche.

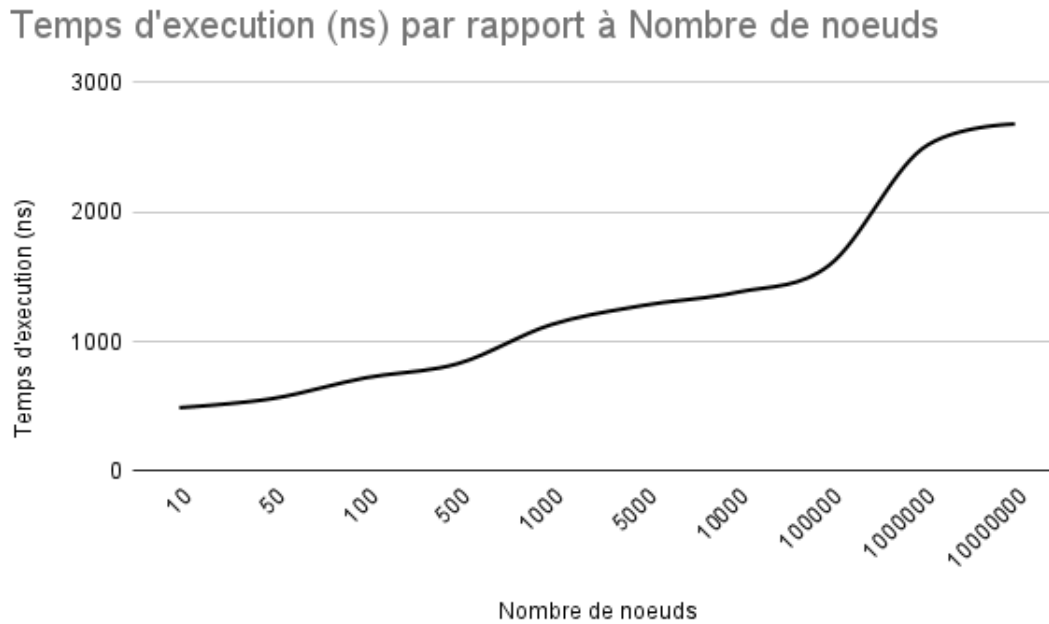


FIGURE 4.5 – Temps d'exécution théorique du programme selon la taille de l'arbre

Depuis le graphe, on observe que le temps d'exécution évolue de manière linéarithmique avec l'augmentation de la taille du problème.

### 4.3.2 Complexité spatiale

Nos noeuds sont directement stockés dans un arbre de recherche à  $n$  éléments. Dans notre cas, un noeud contient une valeur entière et deux pointeurs pour les deux fils. La taille de la valeur stockée ainsi que celle d'un pointeur fait 4 octets. Donc la taille d'un élément dans l'arbre est égale à  $4 + 4 * 2$  unités donc 12 unités.

La complexité spatiale de l'algorithme est  $\mathcal{O}(n)$ .

## 4.4 Expérimentation

Le tableau suivant représente les temps d'exécution en nanoseconde de l'algorithme de suppression dans un arbre à 10 noeuds selon l'approche utilisée, l'équilibre de l'arbre et le nombre de fils du noeud à supprimer.

Type d'approche	Type d'arbre	Type de Suppression	Temps d'execution (ns)
Récursive	Arbre partiellement équilibré	noeud avec deux fils	1651
		noeud avec un fils	1960
		noeud feuille	1080
	Arbre complètement équilibré	noeud avec deux fils	290
		noeud avec un fils	280
		noeud feuille	220
	Arbre complètement Déséquilibré	noeud avec deux fils	290
		noeud avec un fils	290
		noeud feuille	230
Itérative	Arbre partiellement équilibré	noeud avec deux fils	160
		noeud avec un fils	300
		noeud feuille	410
	Arbre complètement équilibré	noeud avec deux fils	260
		noeud avec un fils	200
		noeud feuille	160
	Arbre complètement Déséquilibré	noeud avec deux fils	250
		noeud avec un fils	280
		noeud feuille	200

En effet, nous constatons du tableau que l'algorithme itératif est toujours plus rapide que le récursif. Par ailleurs, la complexité temporelle de la suppression augmente selon le nombre de fils du noeud à supprimer. Plus il a de fils, plus le temps d'exécution est élevé.

Le tableau suivant représente les temps d'exécution en nanoseconde de l'algorithme de suppression de la feuille la plus lointaine de la racine dans un arbre quelconque en utilisant une approche itérative.

N	10	50	100	500	1000	5000	10000	100000	1000000	10000000
t1(ns)	540	691	610	960	910	1251	1980	1581	6980	8430
t2(ns)	300	220	450	690	880	960	1350	2250	6961	11260
t3(ns)	210	260	470	980	780	920	1900	2420	6611	10610
t4(ns)	200	230	680	1160	740	950	1160	4460	6971	8520
t5(ns)	200	290	340	610	1040	1010	2049	3331	5601	9040
t6(ns)	270	200	300	690	550	1220	1250	3810	6140	9061
t7(ns)	330	250	361	580	690	1170	1380	3680	6690	12340
t8(ns)	250	310	279	670	980	819	1310	1720	6430	8790
t9(ns)	230	450	290	650	840	820	1450	2291	6191	7999
t10 (ns)	210	681	310	779	760	960	1520	2690	6399	9900
Moyenne (ns)	274	358	409	776	817	1008	1534	2823	6497	9595

La figure suivante (voir Figure 4.6) représente l'évolution du temps d'exécution selon la taille de l'arbre.

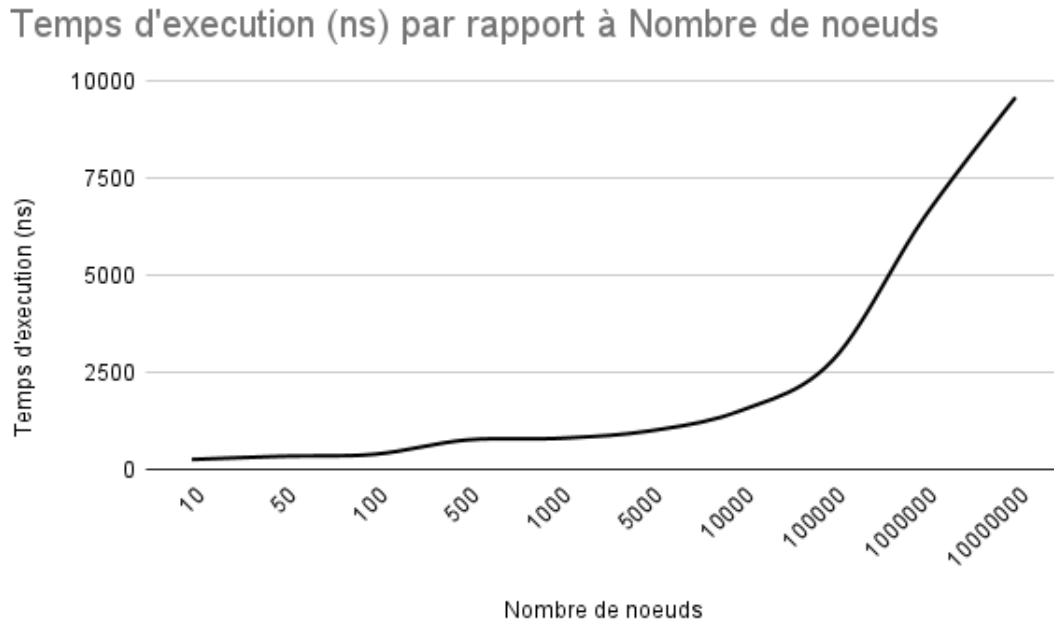


FIGURE 4.6 – Temps d’exécution du programme selon le nombre de noeuds

Depuis le graphe, la courbe est sous forme de courbe ascendante, on observe que le temps d’exécution évolue avec l’augmentation de la taille du problème, ce qui correspond bien à la complexité théorique calculée auparavant.

## 4.5 Conclusion

La suppression d’un nœud dans un arbre binaire de recherche est l’une des opérations les plus compliquées à faire étant donné qu’il faut appliquer un traitement différent selon le nombre de fils que le nœud à supprimer possède. Dans cette partie, nous avons essayé les deux approches : itérative et récursive. Nous constatons qu’en utilisant cette dernière, le code est plutôt petit et facile à comprendre contrairement à la façon itérative qui est beaucoup plus lourde. En revanche, cette dernière nous aide à comprendre de manière détaillée tout ce qui se passe en interne dans l’algorithme et comment la suppression est réellement effectuée. De plus, elle est plus rapide que l’approche récursive.

# Chapitre 5

## Conclusion

Après une mise à l'échelle des graphes afin d'éviter de laisser transparaître les différences de puissances de nos machines respectives, nous constatons que l'algorithme de recherche dichotomique (accompagné de l'algorithme de tri dichotomique) possède une tendance exponentielle, car il est de complexité  $O(\log(n^n + n'))$ . Les autres algorithmes, eux, possédant une complexité  $O(n)$  ont une tendance linéaire avec des pentes différentes, mais n'ont pas le même temps d'exécution (voir Figure 5.1). On peut en conclure que même si des algorithmes distincts possèdent la même complexité ne signifie pas que le temps d'exécution sera le même.

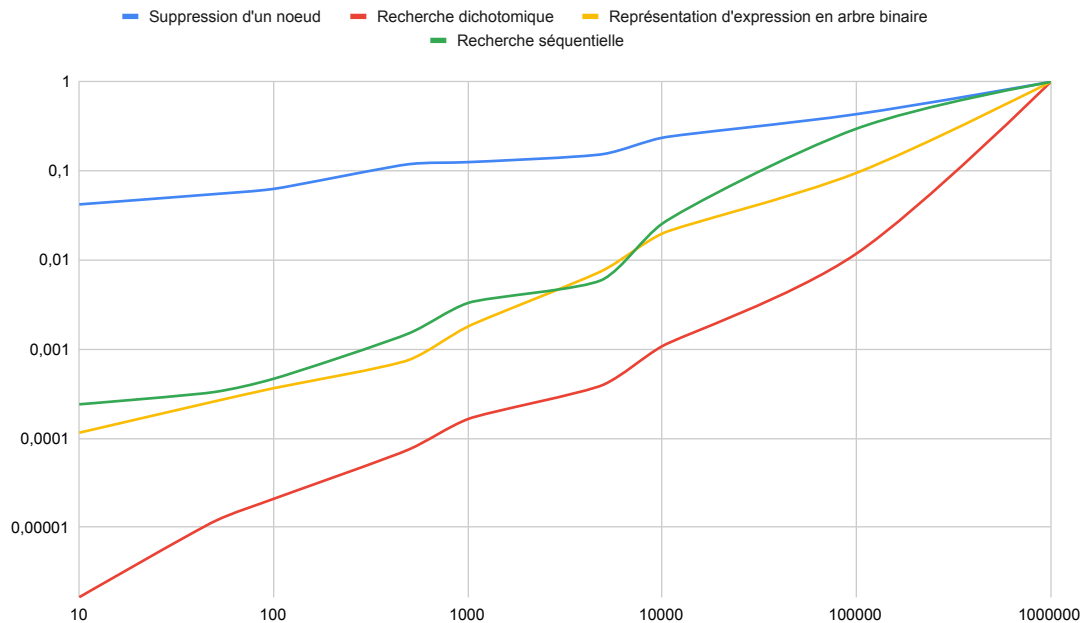


FIGURE 5.1 – Comparaison des temps d'exécution des quatres algorithmes après mise à l'échelle