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
BOUROUINA Rania, 181831052716
CHIBANE Ilies, 181831072041
HAMMAL Ayoub, 181831048403

Année universitaire : 2021 / 2022

Table des matières

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 a 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).

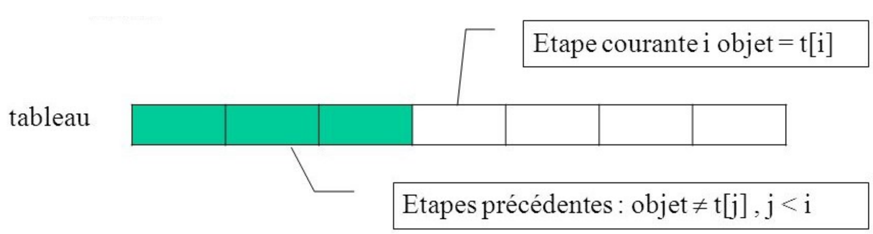

FIGURE 1.1 – Exemple graphique d'une recherche séquentielle

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 au préalable pour le tableau en entrée ; par exemple : il n'est pas necessaire 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.

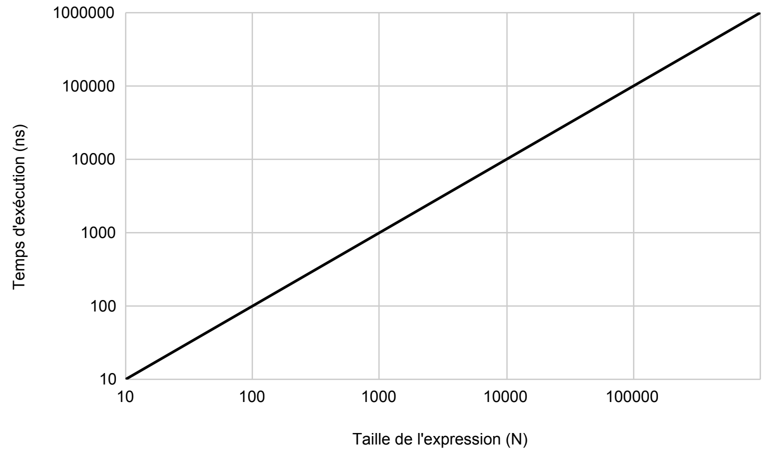

FIGURE 1.2 – Temps d'exécution théorique de l'algorithme de recherche linéaire

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)	582,727273	802,3636	1126,545	3665,909	7985,091	14832	62023,09	62023,09	2423233	34281881,4

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

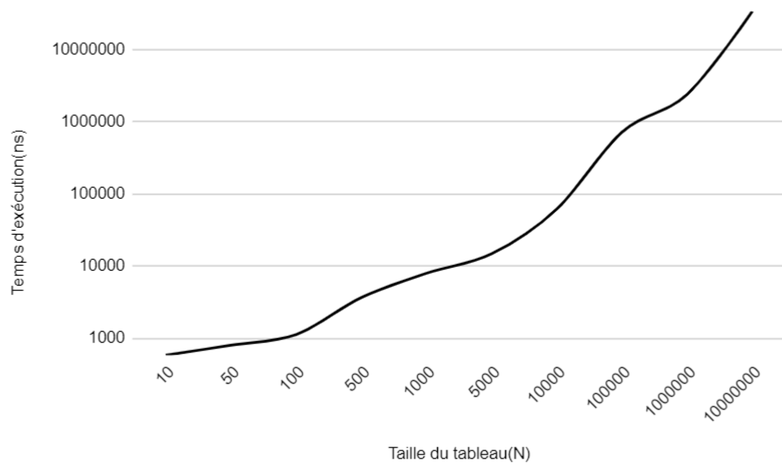

FIGURE 1.3 – Temps d'exécution du programme selon la longueur du tableau

Depuis le graphe, la courbe est sous forme d'un arc ascendant, on observe que le temps d'exécution évolue de manière presque liénaire 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;

$pid1, pid2, pid : pid_t$;

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 recheche 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 revoie -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_t;
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 recheche 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 tomporelle, 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 suffisament d'espace mémoire et on veut accélerer 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).

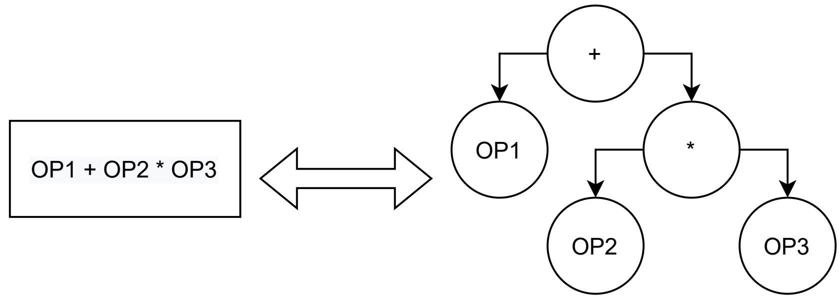

FIGURE 2.1 – Représentation d'une expression arithmétique en arbre binaire

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.

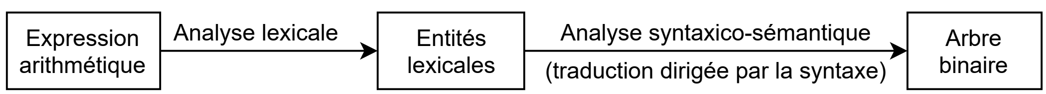

FIGURE 2.2 – Les étapes de la traduction

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 ← 1à expression.taille() faire
      si expression[i] est un opérateur alors
        // Reconnaitre un opérateur
        e.type←opérateur;
        e.operation←type_opération;
        entites.ajouter(e);
      sinon si expression[i] est une parenthèse alors
        // Reconnaitre une parenthèse
        e.type←parenthèse;
        e.parenthèse←type_parenthèse;
        entites.ajouter(e);
      sinon si expression[i] est un chiffre ou un point alors
        // Reconnaitre un nombre
        j←1;
        tant que i + j ≤ 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 ≤ entites.taille() et entites[tc] est un opérateur d'addition ou de
     soustraction faire
      e←entites[tc];
      lt←bt;
      tc←tc + 1;
      si tc > 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 ≤ 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 > 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[ct];
      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 linéraire.

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.

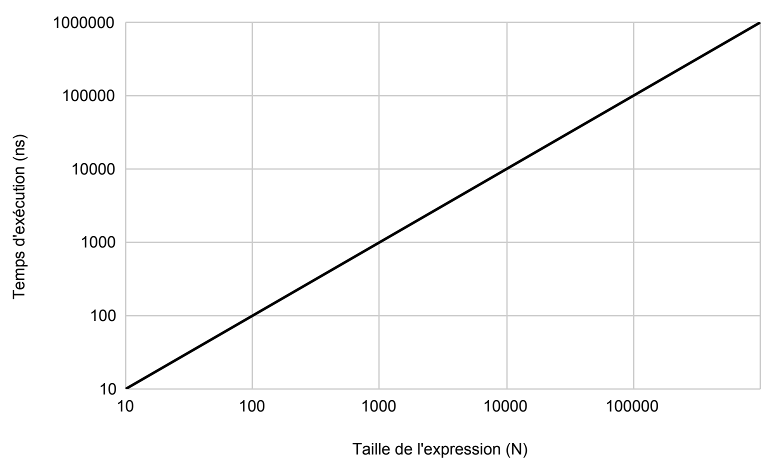

FIGURE 2.3 – Temps d'exécution théorique de l'algorithme de représentation d'expression arithmétique en arbre binaire

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.

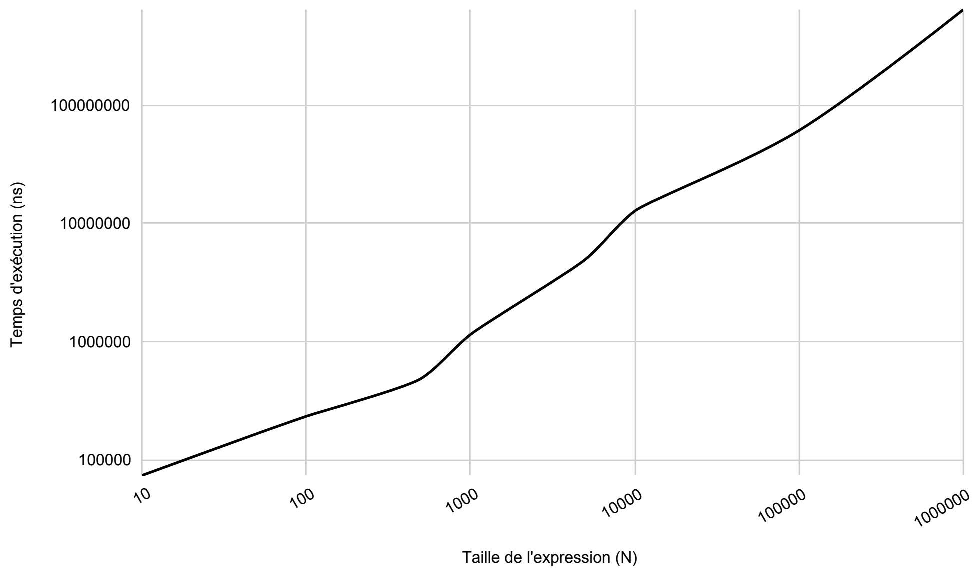

FIGURE 2.4 – Temps d'exécution du programme selon la longueur de l'expression

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 tier.

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 tache 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 trie, 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 inférieur 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 moitie contenant les valeurs supérieures, et ceci jusqu'à avoir trouvé la valeur souhaite ou avoir un sous tableau n'ayant qu'une seule valeur empêchant de continuer la recherche dans le cas ou la valeur n'existe pas dans le tableau. (voir Figure 3.1).

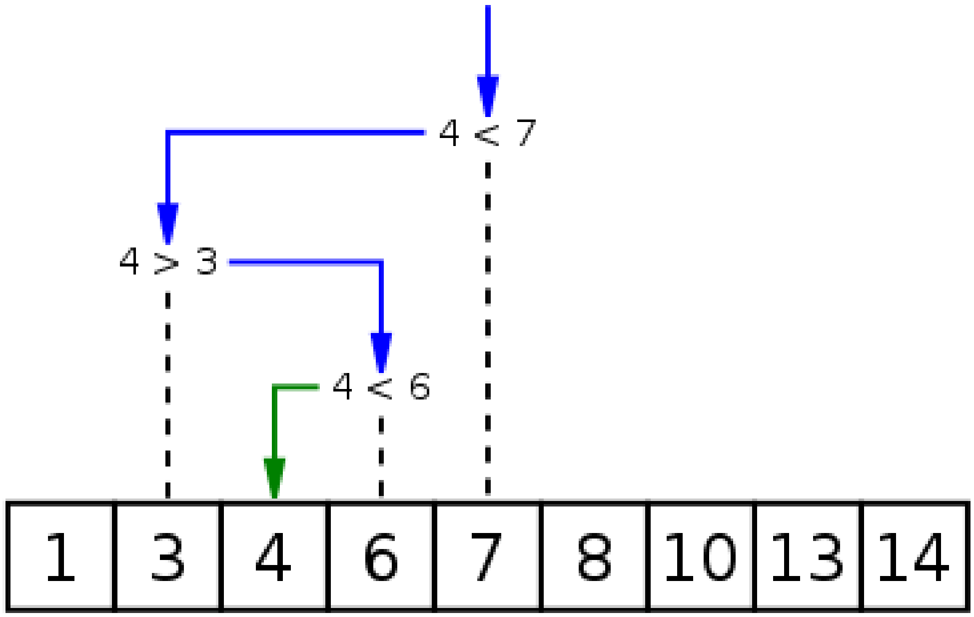

FIGURE 3.1 – Exemple graphique d'une recherche dichotomique

3.2 Fonctionnement de l'algorithme

Il nous est possible de distinguer 2 étapes distinctes et essentiels au bon déroulement de cet algorithme la première étant le trie 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 le 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.

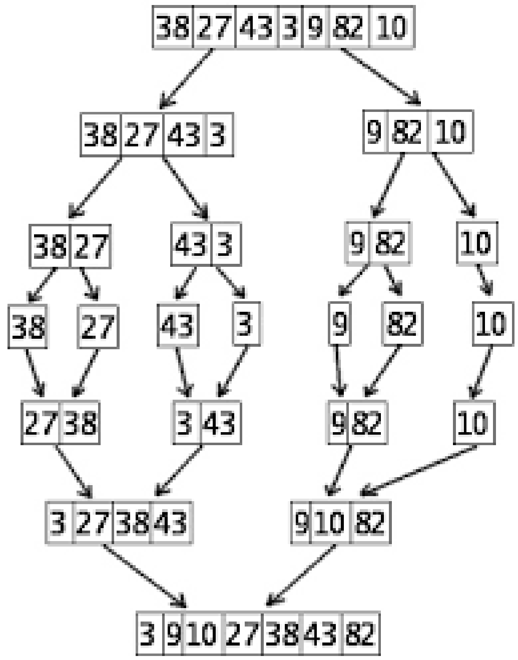

FIGURE 3.2 – Exemple graphique d'un tri dichotomique

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 divive 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 fusioner 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 ← −1;
      retour pos;
    fin si
    si tab[0] == r alors
      pos ← 0;
      retour pos;
    fin si
    si tab[n − 1] == r alors
      pos ← n − 1;
      retour pos;
    fin si
    tant que fin >= debut faire
      // Calcul de la position de l'élément du milieu
      i ← (debut + fin)/2;
      // Si l'élément du milieu est l'élément recherché on retourne sa
         position
      si tab[i] == r alors
        pos ← 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 : $\mathcal{O}(n\log(n)+\log(n')) = O(\log(n^n+n'))$.

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.

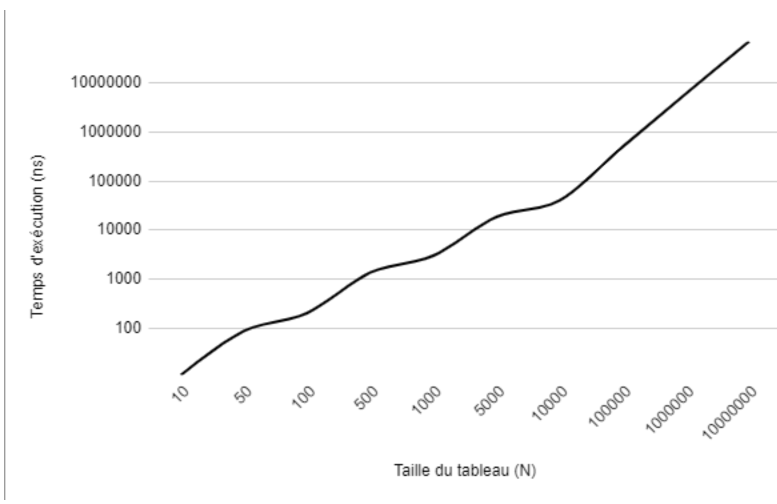

FIGURE 3.3 – Temps d'exécution théorique du programme selon la longueur du tableau

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.

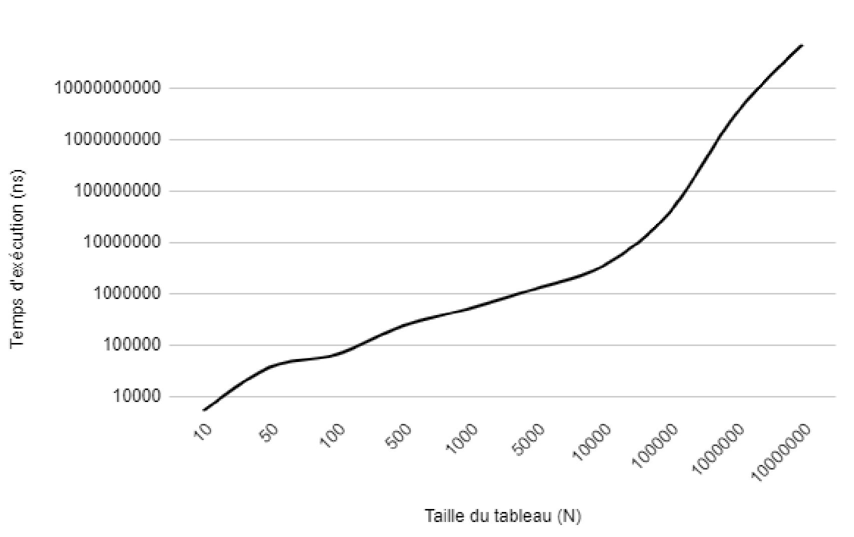

FIGURE 3.4 – Temps d'exécution du programme selon la longueur du tableau

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 l'utilisation d'une fonction de tri dichotomique aura fait passer notre algorithme d'une complexité logarithmique a 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 élèment 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.

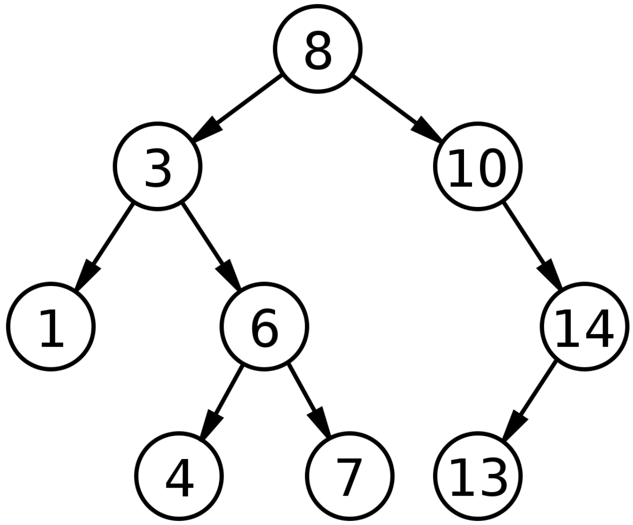

FIGURE 4.1 – Exemple graphique d'un arbre binaire de recherche

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).

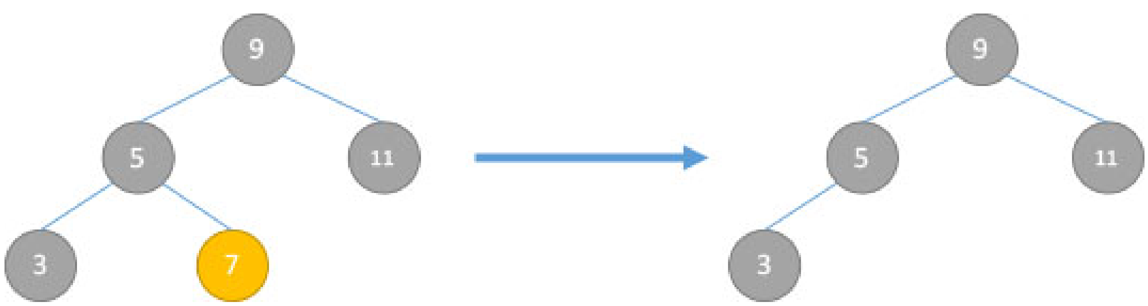

FIGURE 4.2 – Exemple de suppression d'une feuille

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

On décroche le noeud de l'arbre En reliant directement son père et son fils. Ensuite on libère l'espace qu'occupe le noeud 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 nœud 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).

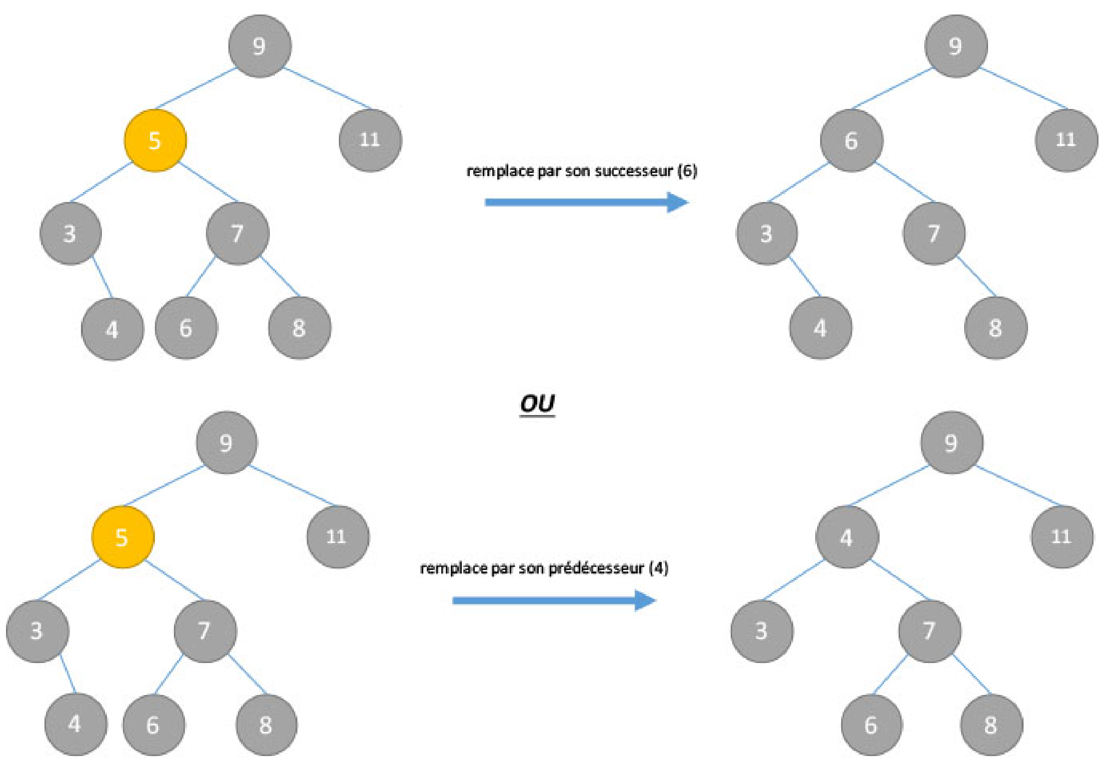

FIGURE 4.4 – Exemple de suppression d'un noeud à deux fils

Fonction supprimerec(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'élèment en cours, on
   supprime à gauche
```

$racine.filsg \leftarrow supprimerec(racine.filsg, valeur)$;

fin si

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

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

$racine.filsd \leftarrow supprimerec(racine.filsd, valeur)$;

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 supprimerec(Entrée : racine : arbre, valeur : entier) : arbre

début

$racine.valeur \leftarrow Temp.valeur$;

$racine.filsd \leftarrow supprimerec(racine.filsd, Temp.valeur)$;

```
// Peut etre remplacé par
```

$racine.filsg \leftarrow supprimerec(racine.filsg, Temp.valeur)$;

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 pourrons utiliser pour effectuer ces deux tâches :

Fonction Min(Entrée : racine : arbre) : arbre

Variables :
min : arbre;

début

$min \leftarrow racine$;

tant que $min <> Nil$ *et* $min.filsg <> Nil$ **faire**

$min \leftarrow min.filsg$;

fin tq

retour min ;

fin

Fonction Max(Entrée : racine : arbre) : arbre

Variables :
max : arbre;

début

$max \leftarrow racine$;

tant que $max <> Nil$ *et* $max.filsd <> Nil$ **faire**

$max \leftarrow max.filsd$;

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 pourrons 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 supprimeiter(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 supprimeiter(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.filsg;
    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;
FIN.
```

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 se fera que sur la motié de l'ABR équilibre, 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 :

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

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

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 $\mathcal{O}(n)$.

La complexité temporelle de la suppression d'un nœud ne diffère pas grandement entre l'approche récursive et l'itérative étant donné que chaque boucle dans cette dernière est remplacée par un appel récursif de la fonction dans la deuxième approche.

Dans la suppression itérative, on commence d'abord par chercher la valeur à supprimer en utilisant une boucle à n itérations. Ensuite une fois trouvée, si le nœud contenant cette valeur a deux fils, nous devons une fois de plus faire la recherche du minimum (resp. Maximum) dans son sous arbre à droite (resp. Gauche) qui contient n' éléments.

Dans le meilleur et moyen cas, l'arbre sera complètement équilibré et donc la complexité temporelle sera $\mathcal{O}(log_2(n-n')+log_2(n')) \approx \mathcal{O}(log_2(n))$.

Dans le pire cas la complexité sera $\mathcal{O}((n-n'))+O(n') \approx O(n)$.

Dans la suppression récursive, on compare à chaque fois la valeur cherchée avec la valeur en cours et on effectue des appels récursifs jusqu'à trouver la valeur voulue. Ensuite, si le nœud contenant cette valeur a deux fils, nous devons faire la recherche du minimum (resp. Maximum) dans son sous arbre à droite (resp. Gauche) afin d'écraser sa valeur. Finalement, on effectue un autre appel récursif sur le même sous-arbre pour supprimer la valeur du minimum.

Dans le meilleur et moyen cas, l'arbre sera complètement équilibré et donc la complexité temporelle sera $\mathcal{O}(log_2(n-n')+log_2(n'))+\mathcal{O}(n'/2) \approx \mathcal{O}(n)$

Dans le pire cas la complexité sera $\mathcal{O}((n-n'))+O(n')+O(n'/2) \approx O(n)$

Le tableau suivant représente les temps d'exécution théorique en nanoseconde de l'algorithme de suppression itérative et récursive d'une feuille selon la variation de la taille de l'arbre.

N	10	50	100	500	1000	5000	10000	100000	1000000	10000000
Itérative t(ns)	250	429	531	569	589	986	845	897	1181	1462
Récursive t(ns)	365	479	563	589	804	895	929	985	1489	1680

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

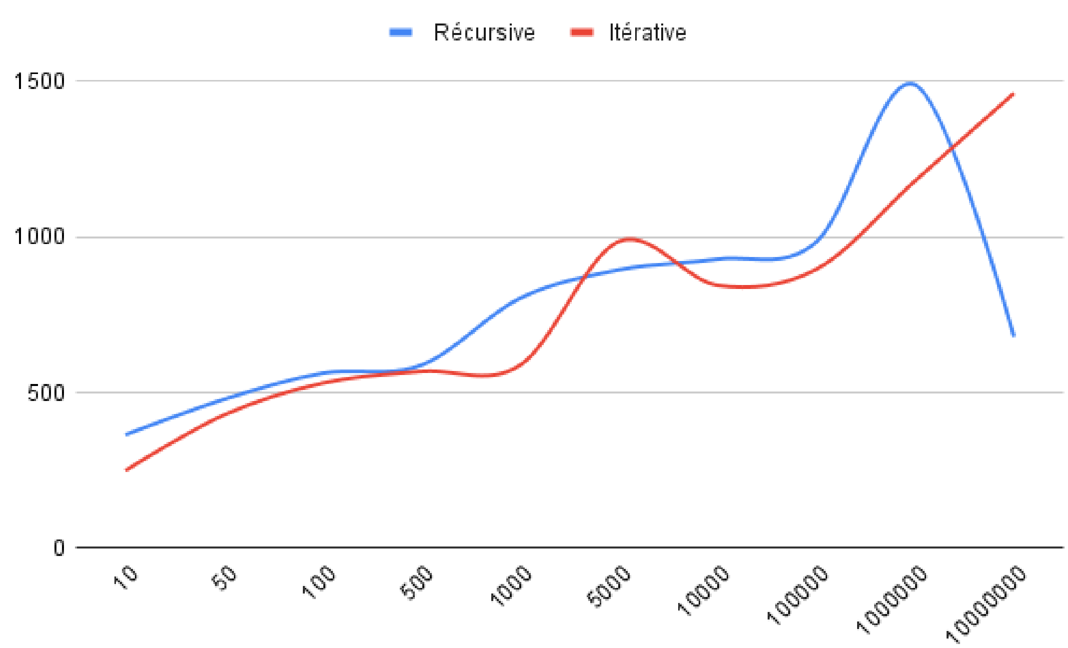

FIGURE 4.5 – Temps d'exécution théorique du programme selon la taille de l'arbre

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 stockées 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 èlèment 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'équillibre 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 complétement équilibré	noeud avec deux fils	290
		noeud avec un fils	280
		noeud feuille	220
	Arbre complétement 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 complétement équilibré	noeud avec deux fils	260
		noeud avec un fils	200
		noeud feuille	160
	Arbre complétement 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 nœud 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)	86	363	501	581	1137	3642	1365	2213	1261	1731
t2(ns)	161	160	282	349	496	869	885	734	793	931
t3(ns)	87	79	164	357	483	371	838	593	1110	1158
t4(ns)	47	79	216	325	566	323	797	897	1355	1623
t5(ns)	43	93	170	326	409	440	749	541	917	1897
t6(ns)	125	101	176	391	308	315	779	553	902	924
t7(ns)	128	80	214	347	338	265	570	1105	1439	843
t8(ns)	47	111	228	320	307	307	868	876	930	1135
t9(ns)	64	74	222	321	374	294	742	940	925	1457
t10 (ns)	117	114	199	307	362	276	687	611	618	1006
Moyenne (ns)	90	125	237	362	478	710	828	906	1025	1270

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

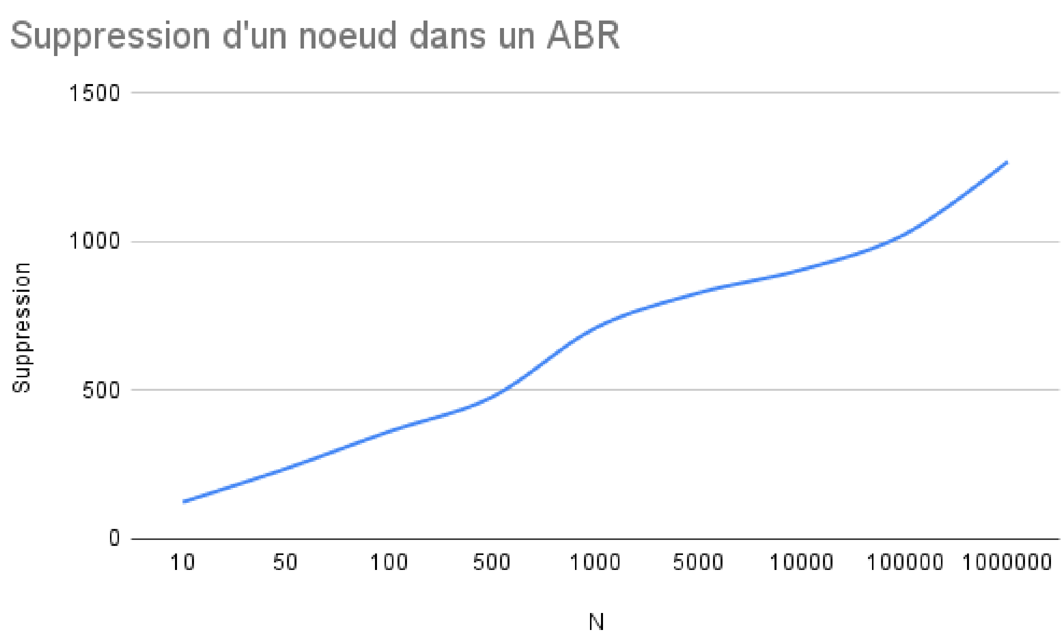

Figure 4.6 – Temps d'exécution du programme selon le nombre de noeuds

Depuis le graphe, 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 plutot 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.

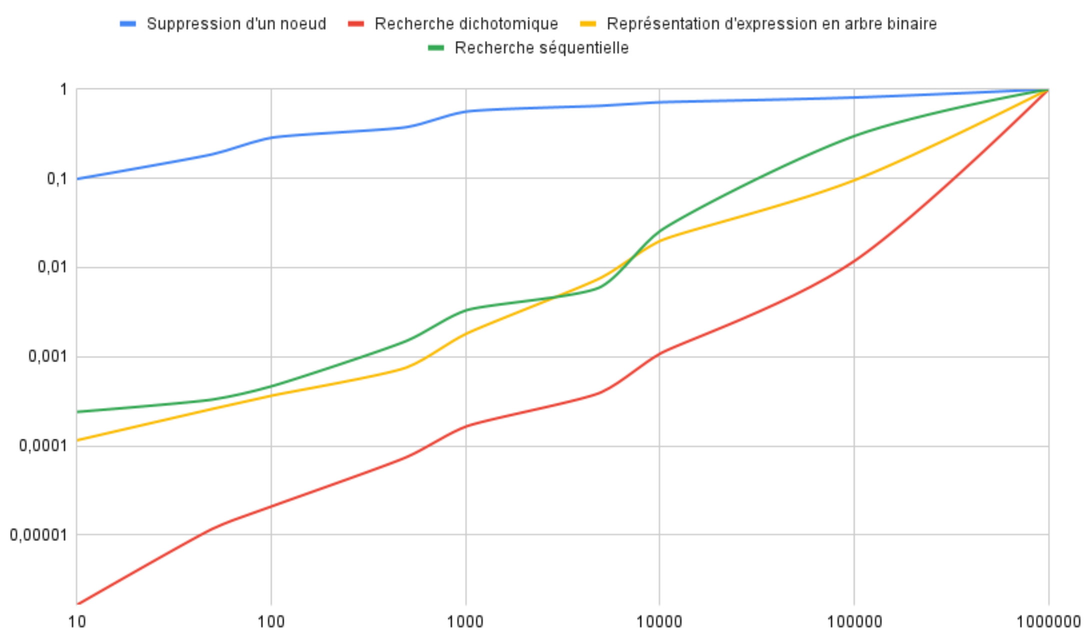

FIGURE 5.1 – Comparaison des temps d'exécution des quatres algorithmes après mise à l'échelle

Chapitre 6

Annexe

6.1 Recherche séquentielle d'un élément dans un tableau

6.1.1 main.cpp

```cpp
#include <iostream>
#include <stdlib.h>
#include <bits/stdc++.h>

#include "RechercheSequentielle.hpp"

using namespace std;

int main()
{
    cout << "choisissez la taille du tableau" << endl;

    unsigned long long n;
    cin >> n;

    int *t = (int *)malloc(n * sizeof(int));
    remplirTabRandom(t, n);

    AffichTab(t, n);

    cout << "choisissez le nombre que vous voulez rechercher dans le tableau" << endl;
    int val;
    cin >> val;


   chrono::time_point<chrono::steady_clock> start = chrono::steady_clock::now(), stop;
//choissisez une seule fonction en mettant l'autre en commentaire
```

```
    int pos = sequentielle(t, n, val);
    //int pos = sequentielle_processus(t, n, val);


     stop = chrono::steady_clock::now();
     chrono::duration<double, nano> duration = stop - start;

     if(pos!=-1)
         cout << "la valeur " << val << " apres avoir trier le tableau se trouve a
          ↪  la position " << pos << endl;
     else
         cout << "la valeur " << val << " n'existe pas dans le tableau" << endl;

     cout << "la fonction de recherche sequentielle apres le tri le tableau aura
      ↪  prit " << duration.count()
          << "ns afin de terminer son execution" <<endl;



        return 0;
}
```

6.1.2 CreationTableau.hpp

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <bits/stdc++.h>

using namespace std;

//fonction qui remplie le tableau avec des valeurs aléatoire
void remplirTabRandom(int *tab, unsigned long long n)
{
    auto R = n * 10;

    int *liste = (int *)malloc(R * sizeof(int));

    srand(time(NULL));
    //On remplie une liste contenant une liste de nombre de 1 a R

    for (auto i = 0; (unsigned)i < R; ++i) {
        liste[i] = i;
    }

    //On mélange par la suite l'ordre des nombres dans la liste aléatoirement
```

```cpp
    for (auto i = 0; (unsigned)i < R; ++i) {
        auto j = i + rand() % (R - i);
        int temp = liste[i];
        liste[i] = liste[j];
        liste[j] = temp;
    }

    //On remplie notre tableau avec les n première valeur de la liste mélangée
    for (auto i = 0; (unsigned)i < n; ++i) {
        tab[i] = liste[i];
    }
    system("cls");

    //On informe l'utilisateur que le tableau a été créer avec succès
    cout << "Tableau de "<< n << " creer avec succes"<< endl;

    free(liste);
}


void AffichTab(int *tab, unsigned long long n)
{
    //Afin de ne pas surchargé la console nous n'afficheront que les 1000
    ↪ premières valeurs du tableau
    cout << "[";
    for(auto i = 0; (unsigned)i < 1000; ++i)
    {
        cout << tab[i];
        if((unsigned)i != (n - 1))
            cout << ", ";
        if(((unsigned)i==999)&&(n!=1000))
            cout << ".......";
        else if((unsigned)i == (n - 1))
            break;
    }
    cout << "]" << endl;
}
```

6.1.3 RechercheSequentielle.hpp

```cpp
#include <iostream>
#include <stdlib.h>
#include <bits/stdc++.h>
#include <vector>
#include <sys/types.h>
#include <unistd.h>
```

```
#include <wait.h>
#include <signal.h>

#include "CreationTableau.hpp"

using namespace std;

int sequentielle(int *tab, unsigned long long n, int r)
{
  unsigned long long i=0;
  bool trouve = false;
  do
  {
      if (tab[i] == r) //si la valeur recherchée est égale à l'élément actuel
      {
          trouve = true;// on met la variable trouve à vrai
      }
     i++;

  } while (trouve == false && i != n);

  if (trouve == true) return i-1; //si la valeur est trouvée on renvoie sa position
  else return -1;//sinon, on renvoie -1

}

int sequentielle_processus(int *tab, unsigned long long n, int r)
{
    int  pos,status;
    bool f = false;
    pid_t pid1, pid2, pid;

    int debut = 0, fin = n/ 2;
    switch (pid1 = fork()) {//on crée le fils 1
        case -1: //dans le cas où y'a un problème lors de la création
            break;
        case 0: // le fils 1 est créé avec succès
            for (int i = debut; i <= fin; i++) {//on commence la recheche du début jusqu'au milieu du tableau
                if (tab[i] == r) {
                    f = true;
                    pos = i;
                    break;
                }
            }
            cout << "la valeur cherchée se trouve a la position " << pos << endl;
```

```
            //si le fils 1 trouve la valeur cherchée il affichera sa position
             ↪  sinon il affiche
            //un nombre négatif
            exit( f ? pos : -1);//si le fils 1 a trouvé la valeur on renvoie sa
             ↪  position
            //sinon on revoie -1
        default:
            break;
    }
    debut = n/ 2 + 1, fin = n - 1;
    switch (pid2 = fork()) {//on crée le fils 2
        case -1://dans le cas où y'a un problème lors de la création
            break;
        case 0:// le fils 2 est créé avec succès
            for (int i = debut; i <= fin; i++) {//on commence la recheche du
             ↪  milieu jusqu'à la fin du tableau
                if (tab[i] == r) {
                    f = true;
                    pos = i;
                    break;
                }
            }
            cout << "la valeur cherchée se trouve a la position " << pos << endl;
            //si le fils 2 trouve la valeur cherchée il affichera sa position
             ↪  sinon il affiche
            //un nombre négatif
            exit(f ? pos : -1);//si le fils 2 a trouvé la valeur on renvoie sa
             ↪  position
            //sinon on revoie -1
        default:
            break;
    }

    while ((pid = wait(&status)) != -1) {//le père attend l'arrivée du premier
     ↪  fils
        if (WIFEXITED(status))
            if (WEXITSTATUS(status)) {
                f = true;
                if (pid == pid1) {
                    kill(pid2, SIGTERM);//si le fils 1 arrive on tue le fils 2
                } else {
                    kill(pid1, SIGTERM);//si le fils 2 arrive on tue le fils 1
                }
            }
    }

    if (f) return 0;
```

```
    else{
        //si aucun fils n'a trouvé la valeur recherché on affiche le message
         ↪ suivant:
        cout << "la valeur cherchée n'existe pas dans le tableau" << endl;
        return -1;
    }


}
```

6.2 Représentation d'une expression arithmétique en arbre binaire

6.2.1 main.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <chrono>

#include "lex.hpp"
#include "tree.hpp"
#include "sem.hpp"

/*  Représentation d'une expression arithmétique en arbre binaire
 *  -------------------------------------------------------------
 *
 *  Entrée : Expression arithmétique sous forme d'une chaîne de caractères.
 *  Sortie : Une représentation en arbre binaire de l'expression arithmétique.
 *
 *  Détails de l'algorithme
 *  -----------------------
 *
 *  Le traitement de la chaîne passe par les étapes suivantes :
 *      - Analyse lexicale : à l'issue de laquelle on génére les entités
 *      lexicales.
 *      - Analyse syntaxico-sémantique par descente récursive : Où le
 *      séquencement d'entités lexicales est transformé en arbre binaire.
 *
 *      expression --<analyse lexicale>--> tableau d'entités
 *                                --<analyse syntaxico-sémantique>--> arbre binaire
 */

int main(int argc, char **argv) {
    // Lecture de l'expression arithmétique
```

```
    std::string expression;
    std::cout << "entrez l'expression : ";
    std::getline(std::cin, expression);
    std::cout << "expression: " << expression << std::endl << std::endl;

    std::vector<entity> entities;
    binary_tree<entity> bt;


    std::chrono::time_point<std::chrono::steady_clock> start =
    ↪  std::chrono::steady_clock::now(), stop;
    try {
        // Génération des entités lexicales
        entities = string_to_entities(expression);
        /*std::cout << "entités lexicales : ";
        for (int i = 0; i < entities.size(); i++)
            std::cout << entities[i] << " ";
        std::cout << std::endl << std::endl;*/
    } catch (int e) {
        std::cout << "Erreur lexicale" << std::endl;
        return 1;
    }

    try {
        // Génération de l'arbre syntaxique
        bt = entities_to_binary_tree(entities);
        stop = std::chrono::steady_clock::now();
        //bt.print_tree();
    } catch (int e) {
        std::cout << "Erreur syntaxique" << std::endl;
        return 1;
    }

    std::chrono::duration<double, std::nano> duration = stop - start;
    std::cout << "temps = " << duration.count() << "ns" << std::endl;

    return 0;
}
```

6.2.2 tree.hpp

```
#ifndef TREE_HPP_
#define TREE_HPP_

#include <iostream>

```

```
template <class T>
struct node {
    T value;
    node<T>* left;
    node<T>* right;
};

template <class T>
class binary_tree{
    private:
        node<T>* root_;

        void print(const std::string& prefix, node<T> *node, bool is_left);

    public:
        binary_tree();
        binary_tree(binary_tree<T>& lt, binary_tree<T>& rt, T& value);
        void set_root(const T& value);
        void set_left(binary_tree<T>& tree);
        void set_right(binary_tree<T>& tree);
        void print_tree();
};

#include "tree.tcc"

#endif
```

6.2.3 tree.tcc

```
// Constructeur par défaut
template <class T>
binary_tree<T>::binary_tree() : root_{nullptr} {
}

// Constructeur
template <class T>
binary_tree<T>::binary_tree(binary_tree<T>& lt, binary_tree<T>& rt, T& value)
    : root_{new node<T>} {
        root_->value = value;
        set_left(lt);
        set_right(rt);
}

// Initialise la racine de l'arbre
template <class T>
void binary_tree<T>::set_root(const T& value) {
```

```cpp
    root_ = new node<T>;
    root_->left = nullptr;
    root_->right = nullptr;
    root_->value = value;
}

// Initialise le fils gauche
template <class T>
void binary_tree<T>::set_left(binary_tree<T>& tree) {
    if (root_ != nullptr)
        root_->left = tree.root_;
}

// Initialise le fils droit
template <class T>
void binary_tree<T>::set_right(binary_tree<T>& tree) {
    if (root_ != nullptr)
        root_->right = tree.root_;
}

// Affiche un noeud de l'arbre et ses fils récursivement
template <class T>
void binary_tree<T>::print(const std::string& prefix, node<T> *node, bool is_left) {
    if (node != nullptr) {
        std::cout << prefix;
        std::cout << (is_left ? "|--" : "'--" );

        // Affichage du noeud courant
        std::cout << node->value << std::endl;

        // Affichage des noeuds fils
        print(prefix + (is_left ? "|   " : "    "), node->left, true);
        print(prefix + (is_left ? "|   " : "    "), node->right, false);
    }
}

// Affiche l'arbre
template <class T>
void binary_tree<T>::print_tree() {
    print("", root_, false);
}
```

6.2.4 lex.hpp

```cpp
#ifndef LEX_HPP_
#define LEX_HPP_

#include <string>
#include <iostream>
#include <vector>

enum class entity_type {
    number,
    operation,
    parenthesis
};

enum class operation_type {
    add,
    sub,
    mul,
    div,
    mod
};

enum class parenthesis_type {
    open,
    close
};

class entity {
    public:
        entity_type type;
        union {
            long double number;
            operation_type operation;
            parenthesis_type parenthesis;
        } value;

        friend std::ostream& operator<<(std::ostream& stream, const entity&
         ↪  entity);
        entity();
};

std::vector<entity> string_to_entities(const std::string &expression);

#endif
```

6.2.5 lex.cpp

```cpp
#include <string>
#include <vector>
#include <iostream>

#include "lex.hpp"


std::vector<entity> string_to_entities(const std::string &expression) {
    std::vector<entity> entities;

    // could use regex tho ...
    for (int i = 0; i < expression.size(); i++) {
        entity e;
        // Cas d'un operateur
        if (expression[i] == '+') {
            e.type = entity_type::operation;
            e.value.operation = operation_type::add;
            entities.push_back(e);
        } else if (expression[i] == '-') {
            e.type = entity_type::operation;
            e.value.operation = operation_type::sub;
            entities.push_back(e);
        } else if (expression[i] == '*') {
            e.type = entity_type::operation;
            e.value.operation = operation_type::mul;
            entities.push_back(e);
        } else if (expression[i] == '/') {
            e.type = entity_type::operation;
            e.value.operation = operation_type::div;
            entities.push_back(e);
        } else if (expression[i] == '%') {
            e.type = entity_type::operation;
            e.value.operation = operation_type::mod;
            entities.push_back(e);

        // Cas de parenthèses
        } else if (expression[i] == '(') {
            e.type = entity_type::parenthesis;
            e.value.parenthesis = parenthesis_type::open;
            entities.push_back(e);
        } else if (expression[i] == ')') {
            e.type = entity_type::parenthesis;
            e.value.parenthesis = parenthesis_type::close;
            entities.push_back(e);
```

```

        // Cas d'un nombre
        } else if (isdigit(expression[i]) || expression[i] == '.') {
            int j = 1;
            while (i + j < expression.size() &&
                    (isdigit(expression[i + j]) || expression[i + j] == '.'))
                j++;

            e.type = entity_type::number;
            e.value.number = std::stold(expression.substr(i, j));

            entities.push_back(e);
            i = i + j - 1;
        // Cas d'un blanc
        } else if (expression[i] == ' ') {
            continue;
        } else {
            throw 1;
        }
    }

    return entities;
}

entity::entity() {
}

// Affichage d'une entité
std::ostream& operator<<(std::ostream& stream, const entity& entity) {
    if (entity.type == entity_type::number) {
        stream << "[type: number, value: " << entity.value.number << "]";
    } else if (entity.type == entity_type::operation) {
        stream << "[type: operation, value: ";
        switch (entity.value.operation) {
            case operation_type::add:
                stream << "addition]";
                break;
            case operation_type::sub:
                stream << "subtraction]";
                break;
            case operation_type::mul:
                stream << "multiplication]";
                break;
            case operation_type::div:
                stream << "division]";
                break;
            case operation_type::mod:
```

```
                stream << "modulo]";
                break;
            default:
                break;
        }
    } else {
        stream << "[type: parenthesis, value: ";
        switch (entity.value.parenthesis) {
            case parenthesis_type::open:
                stream << "open]";
                break;
            case parenthesis_type::close:
                stream << "close]";
                break;
            default:
                break;
        }
    }
    return stream;
}
```

6.2.6 sem.hpp

```
#ifndef SEM_HPP_
#define SEM_HPP_

#include "lex.hpp"
#include "tree.hpp"

binary_tree<entity> entities_to_binary_tree(const std::vector<entity>& entities);

binary_tree<entity> e(const std::vector<entity>&, int& ct);
binary_tree<entity> t(const std::vector<entity>&, int& ct);
binary_tree<entity> f(const std::vector<entity>&, int& ct);

#endif
```

6.2.7 sem.cpp

```
#include "lex.hpp"
#include "tree.hpp"
#include "sem.hpp"

/*  Grammaire:
```

```
 *  Z --> E #
 *  E --> T + { T }* | T - { T }*
 *  T --> F x { F }* | F / { F }*
 *  F --> nb | ( E ) | - F | + F
 */

binary_tree<entity> entities_to_binary_tree(const std::vector<entity>& entities) {
    int ct = 0;
    if (entities.size() == 0)
        throw 1;
    binary_tree<entity> bt = e(entities, ct);
    if (ct < entities.size())
        throw 1;

    return bt;
}

// Routine pour reconnaitre les expressions : gère les opérations + et -
binary_tree<entity> e(const std::vector<entity>& entities, int& ct) {
    binary_tree<entity> bt, rt, lt;
    try {
        // Premier opérande
        bt = t(entities, ct);
        while (ct < entities.size() &&
                entities[ct].type == entity_type::operation &&
                (entities[ct].value.operation == operation_type::add ||
                entities[ct].value.operation == operation_type::sub)) {
            entity op = entities[ct];
            lt = bt;
            ct = ct + 1;
            if (ct >= entities.size())
                throw 1;

            // Deuxième opérande
            rt = t(entities, ct);

            // Construction du noeud d'opération
            bt.set_root(op);
            bt.set_left(lt);
            bt.set_right(rt);
        }
    } catch (int e) {
        throw e;
    }
    return bt;
}
```

```

// Routine pour reconnaitre les termes : gène les opérations de *, / et %
binary_tree<entity> t(const std::vector<entity>& entities, int& ct) {
    binary_tree<entity> bt, rt, lt;
    try {
        // Premier opérande
        bt = f(entities, ct);
        while (ct < entities.size() &&
                entities[ct].type == entity_type::operation &&
                (entities[ct].value.operation == operation_type::mul ||
                entities[ct].value.operation == operation_type::div ||
                entities[ct].value.operation == operation_type::mod)) {
            entity op = entities[ct];
            lt = bt;
            ct = ct + 1;
            if (ct >= entities.size())
                throw 1;

            // Deuxième opérande
            rt = f(entities, ct);

            // Construction du noeud d'opération
            bt.set_root(op);
            bt.set_left(lt);
            bt.set_right(rt);
        }
    } catch (int e) {
        throw e;
    }
    return bt;
}

// Routine pour reconnaitre les entités, les parenthèses et les signes
binary_tree<entity> f(const std::vector<entity>& entities, int& ct) {
    binary_tree<entity> bt, rt, lt;
    // Cas de parenthèses
    if (entities[ct].type == entity_type::parenthesis &&
            entities[ct].value.parenthesis == parenthesis_type::open) {
        // Lecture de la première parenthèse ouvrante
        ct = ct + 1;
        if (ct >= entities.size())
            throw 1;
        // Passage à l'expression englobée par les parenthèses
        bt = e(entities, ct);
        // Lectures de la deuxième parenthèse fermante
        if (ct < entities.size() &&
            entities[ct].type == entity_type::parenthesis &&
```

```
            entities[ct].value.parenthesis == parenthesis_type::close) {
            ct = ct + 1;
        } else {
            throw 1;
        }
    // Cas d'un signe
    } else if (entities[ct].type == entity_type::operation &&
                (entities[ct].value.operation == operation_type::sub ||
                entities[ct].value.operation == operation_type::add)) {
        // Construction d'un noeud (0 <signe> opérande)
        entity op = entities[ct];

        entity left;
        left.type = entity_type::number;
        left.value.number = 0;

        ct = ct + 1;
        if (ct >= entities.size())
            throw 1;

        rt = f(entities, ct);
        lt.set_root(left);

        bt.set_root(op);
        bt.set_left(lt);
        bt.set_right(rt);
    // Cas d'un nombre
    } else {
        // Création d'une feuille contenant le nombre
        entity e = entities[ct];
        bt.set_root(e);
        ct = ct + 1;
    }
    return bt;
}
```

6.3 Recherche dichotomique d’un élément dans un tableau

6.3.1 main.cpp

```
#include <iostream>
#include <stdlib.h>
#include <bits/stdc++.h>

#include "RechercheDichotomique.hpp"
```

```
using namespace std;

int main()
{
    cout << "choisissez la taille du tableau" << endl;

    unsigned long long n;
    cin >> n;

    int *t = (int *)malloc(n * sizeof(int));
    remplirTabRandom(t, n);

    AffichTab(t, n);

    cout << "choisissez le nombre que vous voulez rechercher dans le tableau" << endl;

    int val;
    cin >> val;

    chrono::time_point<chrono::steady_clock> start = chrono::steady_clock::now(), stop;

    TriFusion(t, 0, n - 1);

    int pos = dichotomie(t, n, val);

    stop = chrono::steady_clock::now();
    chrono::duration<double, nano> duration = stop - start;

    if(pos!=-1)
        cout << "la valeur " << val << " apres avoir trier le tableau se trouve a la position " << pos << endl;
    else
        cout << "la valeur " << val << " n'existe pas dans le tableau" << endl;

    cout << "la fonction de recherche dichotomique apres le tri le tableau aura prit " << duration.count()
         << "ns afin de terminer son execution" <<endl;
    return 0;
}
```

6.3.2 CreationTableau.hpp

```cpp
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <bits/stdc++.h>

using namespace std;

//fonction qui remplie le tableau avec des valeurs aléatoire
void remplirTabRandom(int *tab, unsigned long long n)
{
    auto R = n * 10;

    int *liste = (int *)malloc(R * sizeof(int));

    srand(time(NULL));
    //On remplie une liste contenant une liste de nombre de 1 a R

    for (auto i = 0; (unsigned)i < R; ++i) {
        liste[i] = i;
    }

    //On mélange par la suite l'ordre des nombres dans la liste aléatoirement
    for (auto i = 0; (unsigned)i < R; ++i) {
        auto j = i + rand() % (R - i);
        int temp = liste[i];
        liste[i] = liste[j];
        liste[j] = temp;
    }

    //On remplie notre tableau avec les n première valeur de la liste mélangée
    for (auto i = 0; (unsigned)i < n; ++i) {
        tab[i] = liste[i];
    }
    system("cls");

    //On informe l'utilisateur que le tableau a été créer avec succès
    cout << "Tableau de "<< n << " creer avec succes"<< endl;

    free(liste);
}

//L'algo de dichotomie nécessitant que le tableau soit trier nous utilisons une
↪  fonction de trie afin d'éviter toute erreur
// On applique l'algo de tri fusion
```

```
void Fusion(int *tab, unsigned long long const gauche, unsigned long long const
↪ mid, unsigned long long const droite)
{
    auto const subtab1 = mid - gauche + 1;
    auto const subtab2 = droite - mid;

    // Création de tableau temporaire
    int *tabG = (int *)malloc(subtab1 * sizeof(int)),
        *tabD = (int *)malloc(subtab1 * sizeof(int));

    // On copie le contenue du tableau dans tabG et tabD
    for (auto i = 0; (unsigned)i < subtab1; ++i)
        tabG[i] = tab[gauche + i];

    for (auto j = 0; (unsigned)j < subtab2; ++j)
        tabD[j] = tab[mid + 1 + j];

    auto subtab1_index = 0, // Index initial du premier sous tableau
        subtab2_index = 0; // Index initial du second sous tableau
    int tabfusion_index = gauche; // Index initial du tableau fusionné

    // On fusionne les 2 tableau en prennant a chaque fois la valeur le plus
    ↪ petite des 2 tableaux
    while ((unsigned)subtab1_index < subtab1 && (unsigned)subtab2_index <
    ↪ subtab2) {
        if (tabG[subtab1_index] <= tabD[subtab2_index]) {
            tab[tabfusion_index] = tabG[subtab1_index];
            subtab1_index++;
        }
        else {
            tab[tabfusion_index] = tabD[subtab2_index];
            subtab2_index++;
        }
        tabfusion_index++;
    }
    // On copie le contenue du restant du tableau tabG
    while ((unsigned)subtab1_index < subtab1) {
        tab[tabfusion_index] = tabG[subtab1_index];
        subtab1_index++;
        tabfusion_index++;
    }
    // On copie le contenue du restant du tableau tabD
    while ((unsigned)subtab2_index < subtab2) {
        tab[tabfusion_index] = tabD[subtab2_index];
        subtab2_index++;
        tabfusion_index++;
    }
```

```
}

// On divive le tableau en 2 de manière recursive puis on les tri avant de les fusionner
void TriFusion(int *tab, unsigned long long const debut, unsigned long long const fin)
{
    if (debut >= fin)
        return;

    auto mid = debut + (fin - debut) / 2;
    TriFusion(tab, debut, mid);
    TriFusion(tab, mid + 1, fin);
    Fusion(tab, debut, mid, fin);
}

void AffichTab(int *tab, unsigned long long n)
{
    //Afin de ne pas surchargé la console nous n'afficheront que les 1000 premières valeurs du tableau
    cout << "[";
    for(auto i = 0; (unsigned)i < 1000; ++i)
    {
        cout << tab[i];
        if((unsigned)i != (n - 1))
            cout << ", ";
        if(((unsigned)i==999)&&(n!=1000))
            cout << ".......";
        else if((unsigned)i == (n - 1))
            break;
    }
    cout << "]" << endl;
}
```

6.3.3 RechercheDichotomique.hpp

```
#include <iostream>
#include <stdlib.h>
#include <bits/stdc++.h>

#include "CreationTableau.hpp"

using namespace std;

//Algorithme de Recherche dichotomique d'un élément dans un tableau compléxité O(Log(n))
```

```cpp
int dichotomie(int *tab, unsigned long long n, int r)
{

    unsigned long long debut = 0; //Indice du premier élément du sous-tableau analysé
    unsigned long long fin = n; //Indice du dernier élément du sous-tableau analysé
    unsigned long long i; //Indice de l'élément du milieu du sous-tableau analysé

    //Dans le cas ou la valeur recherché est plus grande ou plus petite que les bornes du tableau on retourne -1
    if((r < tab[0])||(r > tab[n-1]))
        return -1;
    //Dans le cas ou la valeur recherché est égale a la valeur de la borne inférieur du tableau on retourne la position 0
    if(tab[0] == r)
        return 0;
    //Dans le cas ou la valeur recherché est égale a la valeur de la borne supérieur du tableau on retourne la position (n-1)
    if(tab[n-1] == r)
        return (n - 1);

    //on réitére la boucle tant qu'il existe des valeur entre le premier et dernier élément du sous tableau
    while ((fin >= debut))
    {
        //Calcul de la position de l'élément du milieu
        i = (debut + fin) / 2;
        /*cout << i << " " << tab[i] <<endl;
        cout << "debut = " << debut <<" fin = "<< fin <<endl;*/
        //Si l'élément du milieu est l'élément recherché on retourne sa position
        if(tab[i] == r)
            return i;
        /*Si la valeur recherchée est plus petite que la valeur du l'élément du milieu alors on regarde le sous-tableau
        de gauche*/
        else if ( tab[i] > r)
            fin = i - 1;
        //sinon on regarde le sous-tableau de droite
        else
            debut = i + 1;
    }
    //Dans le cas ou la valeur n'a pas été trouvé dans le tableau on retourne la valeur -1

    return -1;
```

```
}
```

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

6.4.1 main.cpp

```cpp
#include <iostream>
#include <bits/stdc++.h>
#include <stdlib.h>
#include "suppression.hpp"
using namespace std;





int main(){
    int n = 10;
    // test alétoire
        /*cout << "----------------------------Supression d'un arbre quelconque----------------------------" << endl;
        int *arr = (int *)malloc(n * sizeof(int));
    remplirTabRandom(arr, n);
    element* arbre_aleatoire = creer_arbre(arr, n);
    cout<<"Arbre avant"<<endl;
    traverser_par_niveau(arbre_aleatoire);

        cout << "choisissez le nombre que vous voulez supprimer dans l'arbre " << endl;
    int val;
    cin >> val;

    CalculateTimeRecursive(arbre_aleatoire, val);
    CalculateTimeIterative(arbre_aleatoire, val);
    cout<<"Arbre après"<<endl;
    traverser_par_niveau(arbre_aleatoire);



    // test sur un arbre équilibré
        cout << "----------------------------Supression d'un arbre équilibré----------------------------" << endl;
    TriFusion(arr, 0, n - 1); //il faut trier le tableau avant de créer un arbre equilibré
```

```
    element* arbre_equilibree = creer_arbre_equilibree(arr,0,n-1);
    cout<<"Arbre avant"<<endl;
    traverser_par_niveau(arbre_equilibree);



    CalculateTimeRecursive(arbre_equilibree, val);
    CalculateTimeIterative(arbre_equilibree, val);
    cout<<"Arbre après"<<endl;
    traverser_par_niveau(arbre_equilibree);



    // test sur un arbre completement non équilibré, arr est déja trié donc on
↪   crée normalement l'arbre et tous les noeud vont etre à droite
        cout << "----------------------------Supression d'un arbre complètement
↪   déséquilibré----------------------------" << endl;
    element* arbre_non_equilibree = creer_arbre(arr, n); //arr est déjà trié
    cout<<"Arbre avant"<<endl;
    traverser_par_niveau(arbre_non_equilibree);



    CalculateTimeRecursive(arbre_non_equilibree, val);
    CalculateTimeIterative(arbre_non_equilibree, val);
    cout<<"Arbre après"<<endl;
    traverser_par_niveau(arbre_non_equilibree);
*/

        int *arr = (int *)malloc(n * sizeof(int));
        int i = 0, val, tem = 0, exet;
        element* arbre_aleatoire;


        for (i = 0; i<10; i++){

        remplirTabRandom(arr, n);
        arbre_aleatoire = creer_arbre(arr, n);
        cout<<"Arbre avant"<<endl;
        traverser_par_niveau(arbre_aleatoire);


        CalculateTimeIterative(arbre_aleatoire, arbre_aleatoire->valeur , exet);
                tem = tem + exet;

        cout<<"Arbre après"<<endl;
        traverser_par_niveau(arbre_aleatoire);
```

```

        }
    printf("Le temps d'execution moyen est : %f \n",(float)(tem/10));

        return 0;
}
```

6.4.2 CreationTableau.hpp

```
#include <iostream>
#include <stdlib.h>
#include <bits/stdc++.h>

using namespace std;

//fonction qui remplie le tableau avec des valeurs aléatoire
void remplirTabRandom(int *tab, unsigned long long n)
{
    auto R = n * 10;

    int *liste = (int *)malloc(R * sizeof(int));

    //srand(time(NULL));
    //On remplie une liste contenant une liste de nombre de 1 a R

    for (auto i = 0; (unsigned)i < R; ++i) {
        liste[i] = i;
    }

    //On mélange par la suite l'ordre des nombres dans la liste aléatoirement
    for (auto i = 0; (unsigned)i < R; ++i) {
        auto j = i + rand() % (R - i);
        int temp = liste[i];
        liste[i] = liste[j];
        liste[j] = temp;
    }

    //On remplie notre tableau avec les n première valeur de la liste mélangée
    for (auto i = 0; (unsigned)i < n; ++i) {
        tab[i] = liste[i];
    }

    free(liste);
}

```

```c
//L'algo de dichotomie nécessitant que le tableau soit trier nous utilisons une fonction de trie afin d'éviter toute erreur
// On applique l'algo de tri fusion
void Fusion(int *tab, unsigned long long const gauche, unsigned long long const mid, unsigned long long const droite)
{
    auto const subtab1 = mid - gauche + 1;
    auto const subtab2 = droite - mid;

    // Création de tableau temporaire
    int *tabG = (int *)malloc(subtab1 * sizeof(int)),
        *tabD = (int *)malloc(subtab1 * sizeof(int));

    // On copie le contenue du tableau dans tabG et tabD
    for (auto i = 0; (unsigned)i < subtab1; ++i)
        tabG[i] = tab[gauche + i];

    for (auto j = 0; (unsigned)j < subtab2; ++j)
        tabD[j] = tab[mid + 1 + j];

    auto subtab1_index = 0, // Index initial du premier sous tableau
        subtab2_index = 0; // Index initial du second sous tableau
    int tabfusion_index = gauche; // Index initial du tableau fusionné

    // On fusionne les 2 tableau en prennant a chaque fois la valeur le plus petite des 2 tableaux
    while ((unsigned)subtab1_index < subtab1 && (unsigned)subtab2_index < subtab2) {
        if (tabG[subtab1_index] <= tabD[subtab2_index]) {
            tab[tabfusion_index] = tabG[subtab1_index];
            subtab1_index++;
        }
        else {
            tab[tabfusion_index] = tabD[subtab2_index];
            subtab2_index++;
        }
        tabfusion_index++;
    }
    // On copie le contenue du restant du tableau tabG
    while ((unsigned)subtab1_index < subtab1) {
        tab[tabfusion_index] = tabG[subtab1_index];
        subtab1_index++;
        tabfusion_index++;
    }
    // On copie le contenue du restant du tableau tabD
    while ((unsigned)subtab2_index < subtab2) {
        tab[tabfusion_index] = tabD[subtab2_index];
```

```
        subtab2_index++;
        tabfusion_index++;
    }
}

// On divive le tableau en 2 de manière recursive puis on les tri avant de les
 ↪  fusionner
void TriFusion(int *tab, unsigned long long const debut, unsigned long long const
 ↪  fin)
{
    if (debut >= fin)
        return;

    auto mid = debut + (fin - debut) / 2;
    TriFusion(tab, debut, mid);
    TriFusion(tab, mid + 1, fin);
    Fusion(tab, debut, mid, fin);
}
```

6.4.3 tree.hpp

```
#include <iostream>
#include <stdlib.h> //pour génerer des entiers aléatoirement
#include <bits/stdc++.h>
#include "CreationTableau.hpp"



using namespace std;

//Création de la structure de base
struct element{ //Définition de la structure
    int valeur;
    struct element *gauche=nullptr, *droit=nullptr;
};

struct element* creer_element(int val){ //Création d'un élement dans l'arbre
    struct element* intermediaire = (struct element*)malloc(sizeof(struct
     ↪  element));
    intermediaire->valeur= val;
    intermediaire->droit = intermediaire->gauche =nullptr;
    return intermediaire;
}

struct element* inserer_element(struct element* elt, int val){ //insertion d'un
 ↪  element dans l'arbre
```

```
    if(elt==nullptr)
        return creer_element(val);

    if(val > elt->valeur)
        elt->droit = inserer_element(elt->droit, val);

    else
        elt->gauche = inserer_element(elt->gauche, val);

    return elt;

}




struct element* creer_arbre_equilibree(int arr[], int debut, int fin){

    if (debut > fin)
    return nullptr;

    int milieu = (debut +fin)/2;
    element* inter = creer_element(arr[milieu]);

    //Le coté gauche du tableau fera un sous-arbre droit de l'elt du milieu du
    ↪  tableau
    inter->gauche = creer_arbre_equilibree(arr, debut, milieu - 1);

    //Le coté droit du tableau fera un sous-arbre droit de l'elt du milieu du
    ↪  tableau
    inter->droit = creer_arbre_equilibree(arr, milieu + 1, fin);

    return inter;
}



//Remplissage de l'arbre binaire avec un nombre n d'entiers aléatoires
struct element* creer_arbre(int arr[], int taille){
    struct element *elt =nullptr;
    for(int i=0;i<taille; i++){
        elt = inserer_element(elt, arr[i]);
    }
    return elt;
}
```

```
void chercher(element *racine, int level, int &maxLevel, int &res) //charcher la valeur du noeud le plus profond
{
    if (racine != nullptr)
    {
        chercher(racine->gauche, ++level, maxLevel, res);

        // mettre à jour les vals
        if (level > maxLevel)
        {
            res = racine->valeur;
            maxLevel = level;
        }

        chercher(racine->droit, level, maxLevel, res);
    }
}


int get_feuille(element *racine)
{
    // Initialisation
    int res = -1;
    int maxLevel = -1;


    chercher(racine, 0, maxLevel, res);
    return res;
}



//visualization basique de l'arbre binaire

void traverser_arbre_ordonne(struct element* racine){ //afficher l'arbre binaire en ordre infixé
    if(racine != nullptr){
        traverser_arbre_ordonne(racine->gauche);
        cout <<racine->valeur<<" ";
        traverser_arbre_ordonne(racine->droit);
    }
}

void traverser_arbre_preordonne(struct element* racine){ //afficher l'arbre binaire en ordre préfixé
    if(racine != nullptr){
        cout <<racine->valeur<<endl;
```

```
        traverser_arbre_preordonne(racine->gauche);
        traverser_arbre_preordonne(racine->droit);
    }
}

void traverser_arbre_postordonne(struct element* racine){ //afficher l'arbre binaire en ordre postfixé
    if(racine != nullptr){
        traverser_arbre_postordonne(racine->gauche);
        traverser_arbre_postordonne(racine->droit);
        cout <<racine->valeur<<endl;
    }
}

void traverser_par_niveau(element *racine)
{
    // cas de base
    if (racine == nullptr) return;

    // on utilise une file pour traverser
    queue<element *> q;

    // enfiler la racine
    q.push(racine);

    while (q.empty() == false)
    {
        // Le nombre de noeuds par niveau
        int nombre_noeuds = q.size();

        // defiler tous les noeuds d'un niveau et enfiler les noeuds du niveau prochain
        while (nombre_noeuds > 0)
        {
            element *element = q.front();
            cout << element->valeur << " ";
            q.pop();
            if (element->gauche != nullptr)
                q.push(element->gauche);
            if (element->droit != nullptr)
                q.push(element->droit);
            nombre_noeuds--;
        }
        cout << endl;
    }
}
```

6.4.4 suppression.hpp

```
#include "tree.hpp"
#include<chrono>

//Recherche d'un element dans un arbre binaire
using namespace std;



//Recherche de la valeur minimale dans un arbre, utile pour la suppression d'un element avec deux fils.

struct element* valeur_minimale(struct element* elt){
    struct element* min = elt;
    while(min && min->gauche != nullptr)
        min=min->gauche;
    return min;
}
/***
 * SUPPRESSION SIMPLE ET RECURSIVE
 *
 *
 *
 *
 *
 *
 *
 * ***/
//suppression d'un element dans un arbre binaire
struct element* supprimer_element_recursive(struct element* racine, int val){ //premier essai, pas très optimisé.

    if(racine==nullptr) //cas de base pour la récursivité
        return racine;

    if(val < racine->valeur)  //si la val a supprimer est inf à la valeur de l'elt alors on va supprimer à gauche.
        racine->gauche = supprimer_element_recursive(racine->gauche,val);

    else if(val > racine->valeur)  //si la val a supprimer est inf à la valeur de l'elt alors on va supprimer à droite.
        racine->droit = supprimer_element_recursive(racine->droit,val);

    else { //si val == racine->valeur (on a trouvé l'elt à supprimer)

```

```
        //si l'elt n'a aucun fils
        if(racine->droit==nullptr && racine->droit==nullptr)
            return nullptr;

        //si l'elt a un seul fils
        else if(racine->gauche==nullptr){
            struct element* intermediaire = racine->droit;
            free(racine);
            return intermediaire;
        }else if(racine->droit==nullptr){
            struct element* intermediaire = racine->gauche;
            free(racine);
            return intermediaire;
        }

        //si l'elt a deux fils : on cherche le minimum du coté droit de l'elt et on écrase la valeur de cet elt avec le min trouvé
        struct element* min = valeur_minimale(racine->droit);
        //on écrase la valeur de l'elt à supprimer
        racine->valeur = min->valeur;
        //on supprime le min
        racine->droit = supprimer_element_recursive(racine->droit, min->valeur); //on peut éviter cet appel récursif et diminuer la complexité
    }
    return racine;
}
/***
 * SUPPRESSION NON RECURSIVE
 *
 *
 *
 *
 *
 *
 *
 * ***/


struct element* supprimer_element(struct element* racine, int val){
    struct element* pere=nullptr;
    struct element* actuel = racine;

    // on cherche l'elt à supprimer, pere point sur le pere de actuel
    while(actuel != nullptr && actuel->valeur != val){
        pere = actuel;

        if(val < actuel->valeur)
```

```
            actuel = actuel->gauche;
        else
            actuel = actuel->droit;
    }

    //si la valeur cherchée est inexistante
    if(actuel==nullptr){
        printf("La valeur que vous cherchez à supprimer n'existe pas\n");
        return nullptr;
    }

    // si le noeud a supp a au plus un fils , on cherche c'est quel fils
    if (actuel->gauche == nullptr || actuel->droit == nullptr) {

        // un ptr pour replacer le noeud à supprimer
        element* inter; //

        // si le noeud a un fils droit, on pointe sur lui
        if (actuel->gauche == nullptr)
            inter = actuel->droit;
        else
            inter = actuel->gauche;


        // si le noeud à supprimer est la racine, on retourne directement le ptr
        if (pere == nullptr)
            return inter;

        //si le noeud à supprimer n'est pas une racine, on lie son père avec
        ↪ notre ptr
        if (actuel == pere->gauche)
            pere->gauche = inter;
        else
            pere->droit = inter;

        // on libère l'espace du  noeud actuel et on garde le ptr intermediaire
        free(actuel);
    }

    // si le noeud à supp a deux fils
    else {
        element* min;
        pere = racine;

        // on cherche le min du sous arbre droit du noeud à supprimer on garddant
        ↪ trace du pere
        min = actuel->droit;
```

```
        while (min->gauche != nullptr) {
            pere = min;
            min = min->gauche;
        }

        if (pere != racine) //vérifie si le pere du successeur en ordre sup est le courant ou non. s'il ne l'est pas, alors l'enfant gauche de son pere est égal à l'enfant droit de sup.
            pere->gauche = min->droit;
        else
            actuel->droit = min->droit;

        actuel->valeur = min->valeur;
        free(min);
    }
    return racine;
}


// Autre fonction de suppression
struct element* supprimer_element0(element* racine, int val)
{
    element* pere = nullptr;
    element* actuel = racine;

    // on cherche l'elt à supprimer, pere point sur le pere de actuel
    while(actuel != nullptr && actuel->valeur != val){
        pere = actuel;

        if(val < actuel->valeur)
            actuel = actuel->gauche;
        else
            actuel = actuel->droit;
    }

    //si la valeur cherchée est inexistante
    if(actuel==nullptr){
        printf("La valeur que vous cherchez à supprimer n'existe pas\n");
        return nullptr;
    }

    // Cas 1: le noeud n'a aucun fils
    if (actuel->gauche == nullptr && actuel->droit == nullptr){
        // si le noeud à supp n'est pas une racine, on met son pere gauche/droit à Null
        if (actuel != racine)
        {
```

```
            if (pere->gauche == actuel)
                pere->gauche = nullptr;

            else
                pere->droit = nullptr;
        }
        // si nous avons qu'une racine
        else {
            return nullptr;
        }

        // on libère l'espace
        free(actuel);        // ou bien delete actuel en appelant le destructeur
    }

    // Cas 2: noeud à supp a deux fils
    else if (actuel->gauche && actuel->droit){
        element* min; // pour chercher le succ en ordre
        pere = racine;

        // on cherche le min du sous arbre droit du noeud à supprimer on garddant
        ↪  trace du pere
        min = actuel->droit;
        while (min->gauche != nullptr) {
            pere = min;
            min = min->gauche;
        }

        if (pere != racine) //vérifie si le pere du successeur en ordre sup est
        ↪  le courant ou non. s'il ne l'est pas, alors l'enfant gauche de son
        ↪  pere est égal à l'enfant droit de sup.
            pere->gauche = min->droit;
        else
            actuel->droit = min->droit;

        actuel->valeur = min->valeur;
        free(min);
    }
    // Cas 3: le noeud à supp a un seul fils
    else {
        // choose a intermediaire node
        element* intermediaire = (actuel->gauche)? actuel->gauche: actuel->droit;

        //si le noeud n'est pas une racine, on lie le pere gauche/droit avec
        ↪  l'intermediaire
        if (actuel != racine)
        {
```

```
            if (actuel == pere->gauche)
                pere->gauche = intermediaire;

            else
                pere->droit = intermediaire;

        }
        //si le noeud  est la racine, l'intermediaire devient direct la racine
        else
            return intermediaire;

        // deallocate the memory
        free(actuel);
    }
    return racine;
}



bool existe(struct element* noeud, int val){ //vérifier si le noeud existe pour l'approche récursiv
    if (noeud == nullptr)
        return false;

    if (noeud->valeur == val)
        return true;


    bool res1 = existe(noeud->gauche, val);

    if(res1) return true;


    bool res2 = existe(noeud->droit, val);

    return res2;
}

void CalculateTimeRecursive(element* &racine, int val){
    if (! existe(racine,val)){
        printf("La valeur que vous cherchez à supprimer n'existe pas\n");
    }
    else{
        //time start
        chrono::time_point<chrono::steady_clock> start = chrono::steady_clock::now(), stop;
        racine = supprimer_element_recursive(racine, val);
```

```
        //time ends
        stop = chrono::steady_clock::now();
        chrono::duration<double, nano> duration = stop - start;
        printf("Le temps d'execution avec une approche Récursive est : %lf nanosecondes\n", duration.count());
    }
}

void CalculateTimeIterative(element* &racine, int val, int &ex){
        //time start
        chrono::time_point<chrono::steady_clock> start = chrono::steady_clock::now(), stop;
        racine = supprimer_element(racine, val);
        //time ends
        stop = chrono::steady_clock::now();
        chrono::duration<double, nano> duration = stop - start;
        printf("Le temps d'execution avec une approche Itérative est : %lf nanosecondes\n", duration.count());
            ex = duration.count();

}
```