AFFICHAGE PRÉFIXÉ D'UN ARBRE BINAIRE DE RECHERCHE

3)-a Description de l'objectif de l'algorithme traitée:

ENTRÉE :

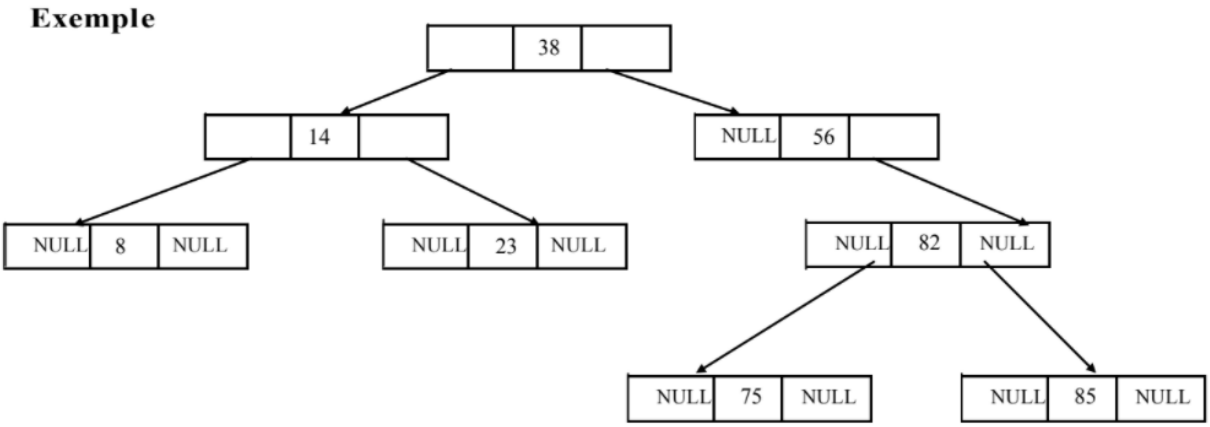
N : Nombre d'éléments que va contenir l'arbre

SORTIE :

Affichage préfixé de l'arbre binaire de recherche c'est à dire on commence par visiter (et traiter) la racine R, on effectue ensuite un parcours préfixé de tous les nœuds de A1 (fils gauche) jusqu'aux feuilles. On revient pour effectuer ensuite un parcours préfixé de tous les nœuds de A2 (filsdroit) selon l'ordre (R, A1, A2).

Mais avant d'afficher on doit créer d'abord l'arbre binaire de recherche dont chacun des nœuds X possède la propriété suivante : La valeur contenue dans X est supérieure à toute valeur contenue dans le sous-arbre gauche de X, et elle est inférieure à toute valeur contenue dans le sous-arbre droit de X.

La création se fera ainsi : créer la racine de l'arbre (comportant la première valeur donné par random()), ensuite pour chacune des autres valeurs appeler la fonction Inserer pour maintenir l'ordre dans l'arbre.



L'affichage de cet arbre binaire de recherche sera ainsi :

{ 38,14,8,23,56,82,75,85 }

On constate après l'affichage et aprés calcul du temps d'exécution que :

plus N sera grand (plus l'arbre contiendra de neoud)plus le temps d'execution sera long , le meilleur cas sera N=1 (L'arbre contient un seul élément).

3)-b Explication du fonctionnement de l'algorithme:

-REPRESENTATION DE L'ARBRE : L'arbre sera représenté par une structure dynamique non linéaire. Chaque élément sera composé de trois champs : un champ info représentant l'information contenue dans le nœud, un champ succ_gauche donnant l'adresse du fils gauche en mémoire et un champ succ_droit donnant l'adresse du fils droit en mémoire.

Element = Enregistrement

info : entier ; succ_gauche : ^Ele ment; succ_droit : ^Elément ;

Finenreg ;

noeud = ^Elé ment ;

Racine : nœud ;

```
typedef struct elt
{int info ; struct elt *succ_gauche ; struct elt *succ_droit ;} *noeud ;
```

-CREATION DE L'ARBRE BINAIRE DE RECHERCHE:

Algorithme Const_Arbre ;

Var

n, x,y : entier; R, père: nœud;

Debut

/*definir le nombre n qui est la longueur de l'arbre défini par l'utilisateur*/

Ecrire ("donnez le nombre de valeurs") ; lire(n) ;

/*definir la valeur de la racine x selon la fonction random*/

Ecrire ("premiere valeur ") ; x=rand() ;

/* création de la racine (allouer de l'espace mémoire)*/

R <- Allouer (Taillede (nœud)) ;

/*le champ info de la racine recevra la valeur x donnée par rand() et les champs succ gauche et droit recevrons NIL avant la creation des autres noeuds */

^ R.info <- x ; ^R.succ_gauche <- nil ; ^R. succ_droit <- nil ;

/* création des autres nœuds*/

Pour i <- 2 à n

Faire

Ecrire ("autre valeur ") ; y=rand() ;

père <- nil ;

Insere (R, y, père) ; /* appel de la fonction insere pour inserer la valeur dans l'arbre selon l'ordre */

Fait ;

Debut du calcul du temps d'execution

/* Appel à la fonction Prefixe pour l'affichage de l'arbre */

Prefixe(R) ;

Fin du calcul du temps d'execution

Afficher le temps d'execution en nanosecondes

Fin.

```
int main()
{

noeud racine;
noeud pere;
int y,n,x;
  do{
        printf("Donner le nombre de valeurs \n");
        scanf("%d",&n);}
        while(n<0);
        printf("********************CREATION DE L'ARBRE BINAIRE DE RECHERCHE******
        printf(" premiere valeur \n");
        x=rand();
        printf("%d\n",x);

racine=(noeud)malloc(sizeof(struct elt));
racine->info=x;
racine->succ_gauche=NULL;
racine->succ_droit=NULL;
for(int i=2;i<=n;i++){
        printf("autre valeur \n");
        y=rand();
        printf("%d \n",y);
        pere=NULL;
        insere(racine,y,pere);
        }
/*affichage préfixé de l'arbre*/
printf("******************** AFFICHAGE PREFIXE DE L'ARBRE BINAIRE DE RECHERCHE ****
auto debut=chrono::steady_clock::now();
prefixe(racine);
auto fin=chrono::steady_clock::now();
printf("******************** TEMPS D'EXECUTION EN NANOSECONDES ******************

cout << "temps d'execution en nanosecondes:"<<chrono::duration_cast<chrono::nanosec
return 0;
}
```

-Procédure insere :

L'insertion d'une valeur dans un arbre binaire ordonné doit maintenir l'ordre des éléments dans l'arbre. On suppose que la répétition de valeurs n'est pas

permise. On écrit une fonction récursive insère, si la valeur val existe dans l'arbre on arrête le traitement, sinon on vérifie la valeur val par rapport à la valeur contenue dans le nœud courant. Selon les cas, on s'oriente soit vers le fils gauche de a, soit vers le fils droit de a.

L'insertion consiste à allouer un nouvel espace pour la valeur val et à mettre à jour les chaînages dans l'arbre (pour cela, il faut avoir l'adresse du nœud père).

Procédure Insere (E/ a: nœud ; E/ val : entier ; E/ père : noeud);

Var p: nœud ;

Debut

```
Si ( non V ide(a))
    Alors  Si (^a.info = val) alors Ecrire ("La valeur existe déjà");
            Sinon pere <- a ;
                Si (^a.info > val)
                    alors Insere (Fils_gauche(a), val, père ) ;
                sinon Insere (Fils_droit(a), val, père) ;
Fsi ; Fsi ;
Sinon / * créer le noeud*/
        p <- Allouer (Taillede (nœud)) ; /* création du nouvel élément */
        ^ p.info <- val ; ^p.succ_gauche <- nil ; ^p. succ_droit <- nil ;
/* raccorder au nœud père */
        Si (^père. info >val) alors ^père.succ_gauche <- p ;
            Sinon ^père.succ_droit <- p ;
Fsi ;
Fin ;
```

```
void insere(noeud a,int val,noeud pere ){
noeud p;

    if(Vide(a)==0){
        if(a->info == val){
            printf("La valeur existe deja");
        }
        else{
            pere=a;
            if(a->info > val){
                insere(a->succ_gauche,val,pere
            }
            else{
                insere(a->succ_droit,val,pere)
  }
        }
    }
    else{
        p=(noeud)malloc(sizeof(struct elt));
        p->info=val;
        p->succ_gauche=NULL;
        p->succ_droit=NULL;
        if(pere->info > val){
            pere->succ_gauche=p;
        }
        else{
           pere->succ_droit=p;

        }
    }


}
```

Dans la procédure insere on a utiliser la fonction vide qu'on definira ci-dessous :

Elle nous permettera de vérifier si un noeud a est vide retourne 1 si vide sinon 0 .

Fonction Vide (E/ a : nœud) : boolée n ;

Debut

Si (a = nil) alors retourner (vrai) ;

sinon retourner (faux) ; fsi ;

Fin ;

```
int Vide(noeud a){
if (a == NULL) return(1) ;
else return(0);
}
```

Procédure prefixe:

Dans notre partie du code principale on a appeler une fonction prefixe qui nous permettra de parcourir l'arbre d'une manière préfixé pour l'afficher :

On commence par visiter (et traiter) la racine R, on effectue ensuite un parcours préfixé de tous les nœuds de A1 (fils gauche) jusqu'aux feuilles. On revient pour effectuer ensuite un parcours préfixé de tous les nœuds de A2 (filsdroit).

Procédure Préfixe (E/ a : nœud);

Début

Si (non Vide(a))

Ecrire (^a.info) ;

Si (non Feuille (a))

/* 1er appel récursif */

Préfixe (Fils_gauche(a)) ;

/* 2ème appel récursif */

Préfixe (Fils_droit(a)) ;

fsi ;

Fin ;

```
void prefixe(noeud a)
{ if ( Vide(a)== 0)
{ printf(" %d \n", a->info) ;
if ( Feuille(a)==0)
{ prefixe (a->succ_gauche) ;
prefixe (a->succ_droit) ; }
}
}
```

Dans la procédure prefixe on appeler une fonction feuille qui verifie si un nœud est une feuille ou non .

Fonction Feuille: Vérifier si un nœud a est une feuille c'est à dire le successeur droit et gauche sont à NIL retourne vrai sinon retourne faux.

Fonction Feuille (E/ a : nœud) : boolée n ;

Debut

Si (^a.succ_gauche = nil et ^a.succ_droit = nil)

alors retourner (vrai) ;

sinon retourner (faux) ; fsi ;

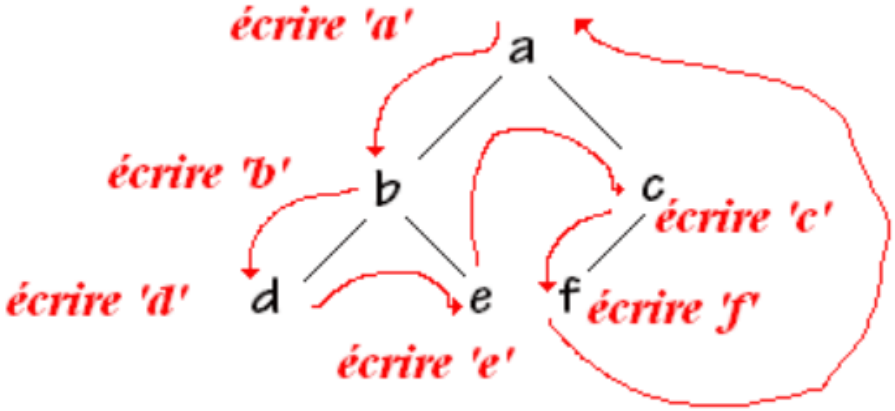
Fin ;

```
int Feuille (noeud a)
{ if (a->succ_gauche == NULL && a->succ_droit==NULL)
return(1) ;
else return(0) ; }
```

3)c Calcul de la complexité (temporelle+spatiale):

Affichage prefixé d'un arbre binaire de recherche (but du travail):

Le parcours de l'arbre se fera selon le graphe si dessous donc on parcourra tous les éléments pour les afficher .Donc la complexité temporelle sera **O(n)** si il y'a n éléments dans l'arbre .on visite chaque élément de l'arbre une et une seule fois, D'ailleurs, sur n'importe quel arbre, un parcours d'un arbre sera toujours en O(n). Il n'y a pas de pire des cas.



le **meilleur cas** sera le cas où l'arbre contient un seul élément la compléxité temporelle sera en **O(1)** car elle executera que le printf de un seul élément par contre la création de cet element sera en O(4) car elle executera 4 instructions qui sont :

racine=(noeud)malloc(sizeof(struct elt));

racine->info=x;

racine->succ_gauche=NULL;

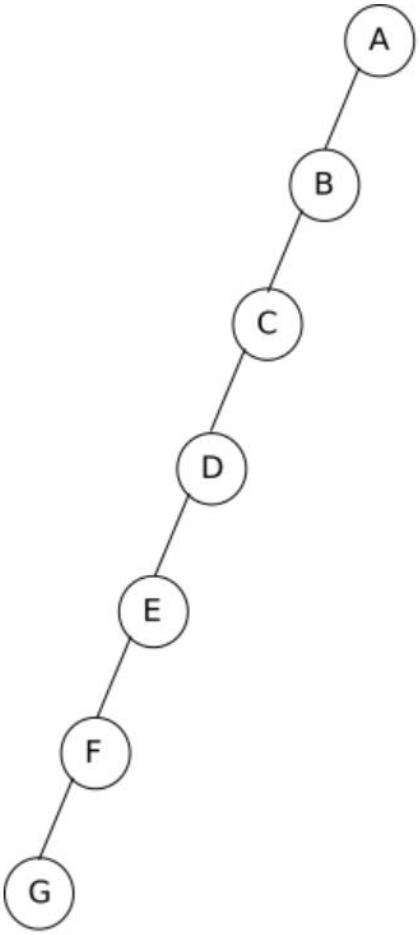
racine->succ_droit=NULL;

Complexité de la création de l'arbre binaire de recherche (informations supplémentaire):

Pour insérer une valeur dans l'arbre dans le meilleur cas et en moyenne on élimine une moitié d'arbre à chaque comparaison jusqu'a tomber sur un emplacement libre donc pour compter la complexité on va calculer combien de fois on divise n par 2 jusqu'a tomber sur 1 donc il s'agit d'une fonction logarithmique (en base 2) , l'algorithme d'insertion est donc O(Log(n)) . Pour insérer n valeurs on aura logiquement O(n*Log(n)) pour la création.

Dans le pire des cas, l'arbre est totalement balancé d'un coté comme le graphe présenté ci-dessous . Pour l'insertion, cela impose donc un parcours de tous les éléments et une complexité pour l'insertion en O(n).

La construction de l'ABR se fera donc en $O(n^2)$, appelée complexité quadratique.



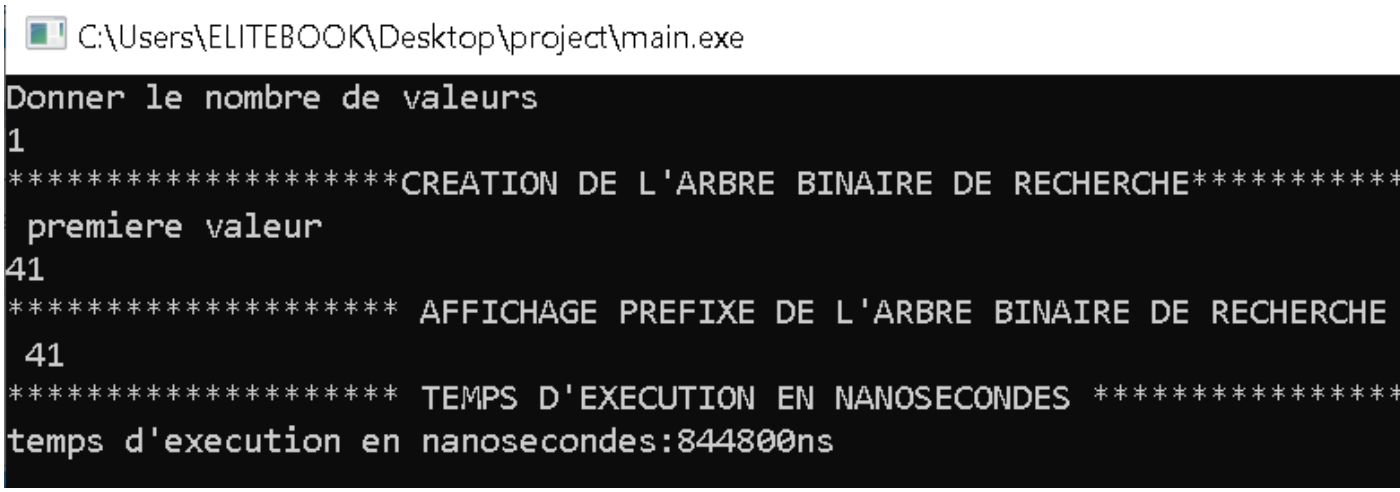
Complexité spatiale :

-pour la creation de l'arbre la complexité spatiale est $O(n)$ car on alloue de l'espace mémoire pour n éléments .

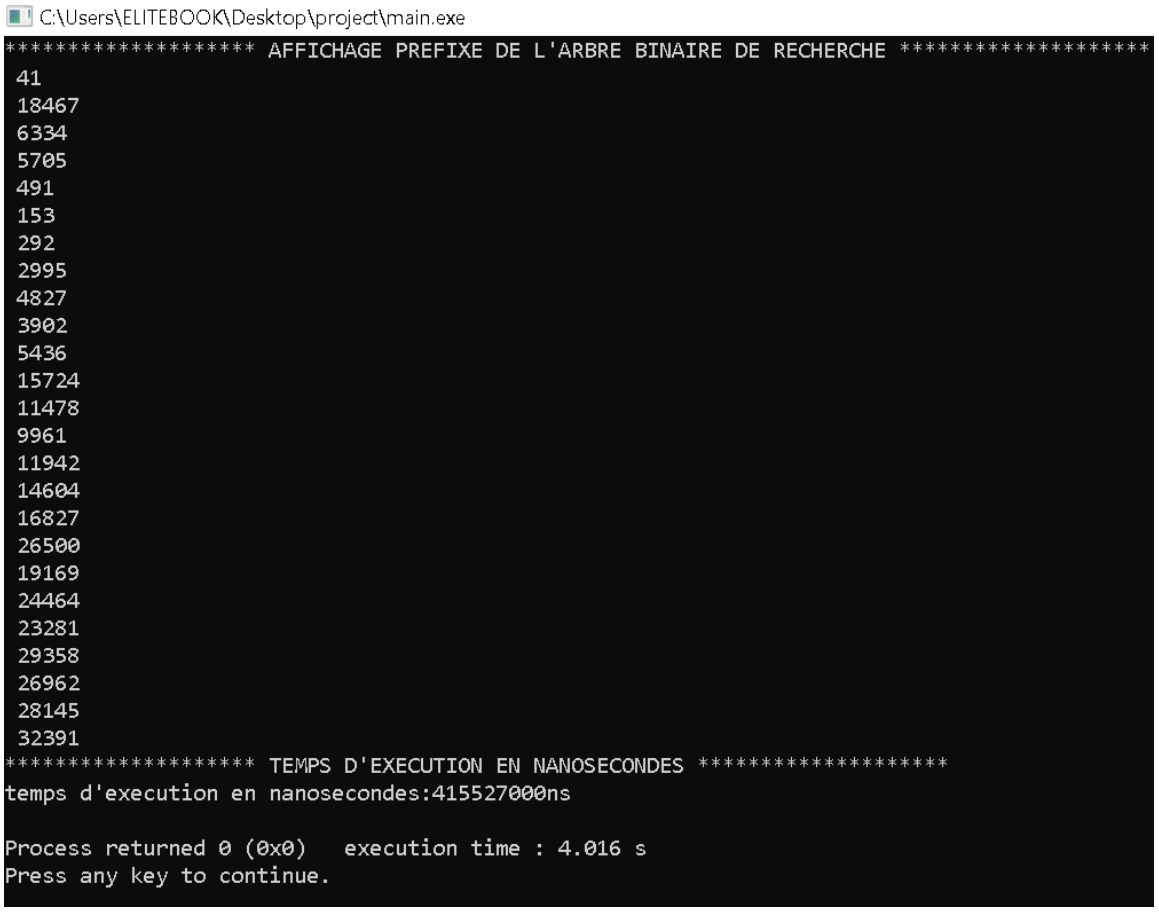
par contre pour l'affichage des éléments de l'arbre binaire de recherche la complexité spatiale est $O(1)$ car on ne crée pas d'objet supplémentaire on affiche seulement , l'algorithme utilise un fixe (petite) quantité d'espace qui ne dépend pas de l'entrée. Pour chaque taille de l'entrée de l'algorithme prendra le même (constant) montant de l'espace.

3)d Experimentation (en modifiant la taille du probléme pour montrer l'impact de la complexité(theorique+ experimentale):

pour un arbre contenant 1 seul élément:



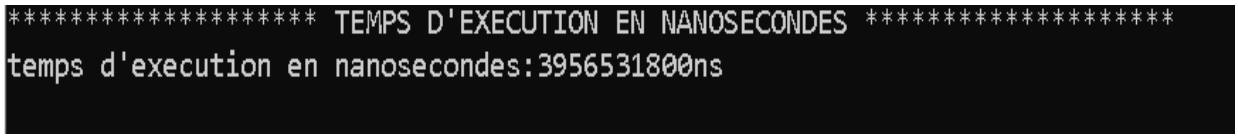
Pour un arbre contenant 25 éléments:



Pour un arbre contenant 50 éléments:

Pour un arbre contenant 75 éléments :

Pour un arbre contenant 250 éléments:



Pour un arbre contenant 500 éléments :

```
******************** TEMPS D'EXECUTION EN NANOSECONDES ********************
temps d'execution en nanosecondes:8177579800ns
```

Pour un arbre contenant 750 éléments :

```
******************** TEMPS D'EXECUTION EN NANOSECONDES ********************
temps d'execution en nanosecondes:12070913400ns
```

Pour un arbre contenant 2500 éléments :

```
******************** TEMPS D'EXECUTION EN NANOSECONDES ********************
temps d'execution en nanosecondes:39079834400ns
```

Pour un arbre contenant 5000 éléments :

```
******************** TEMPS D'EXECUTION EN NANOSECONDES ********************
temps d'execution en nanosecondes:81577598600ns
```

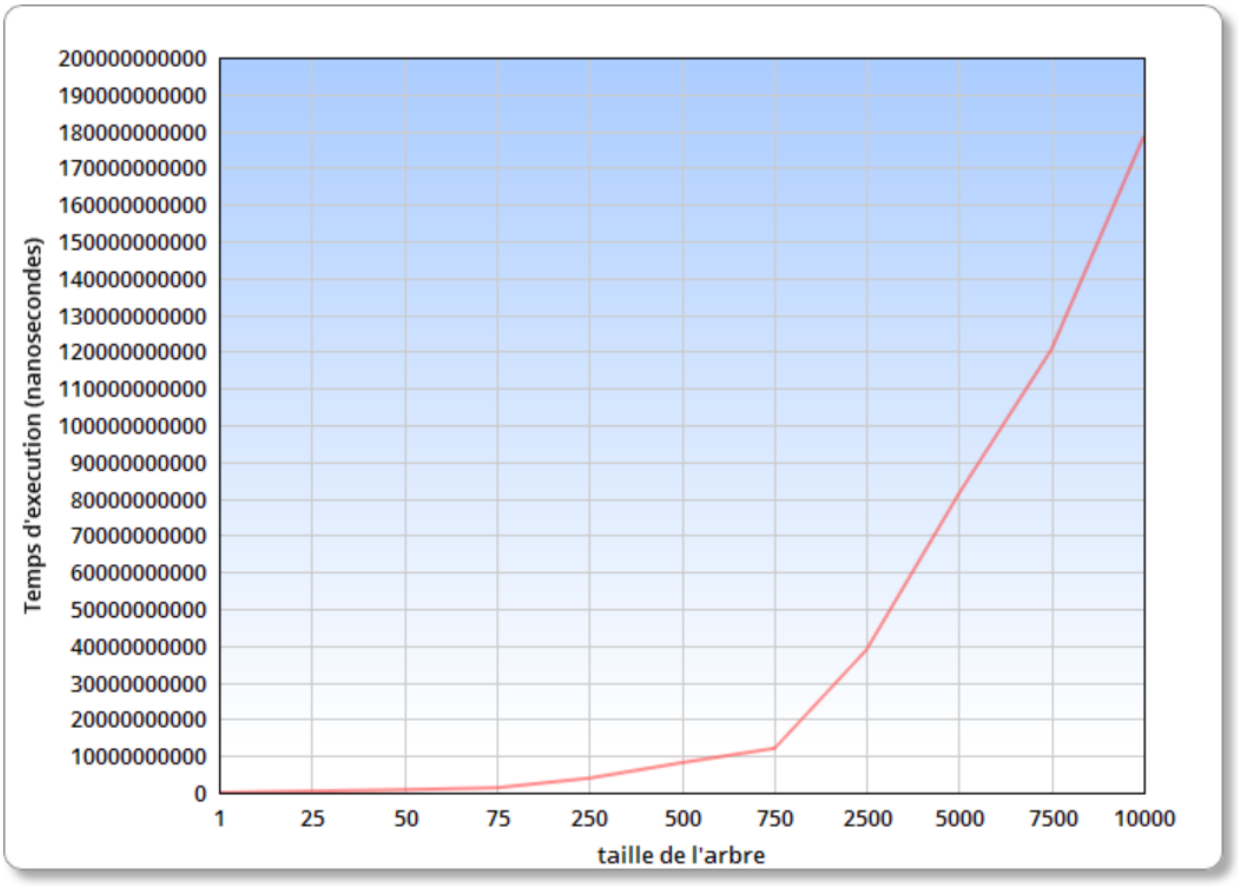
Pour un arbre contenant 7500 éléments :

```
******************** TEMPS D'EXECUTION EN NANOSECONDES ********************
temps d'execution en nanosecondes:120757121100ns
```

Pour un arbre contenant 10000 éléments :

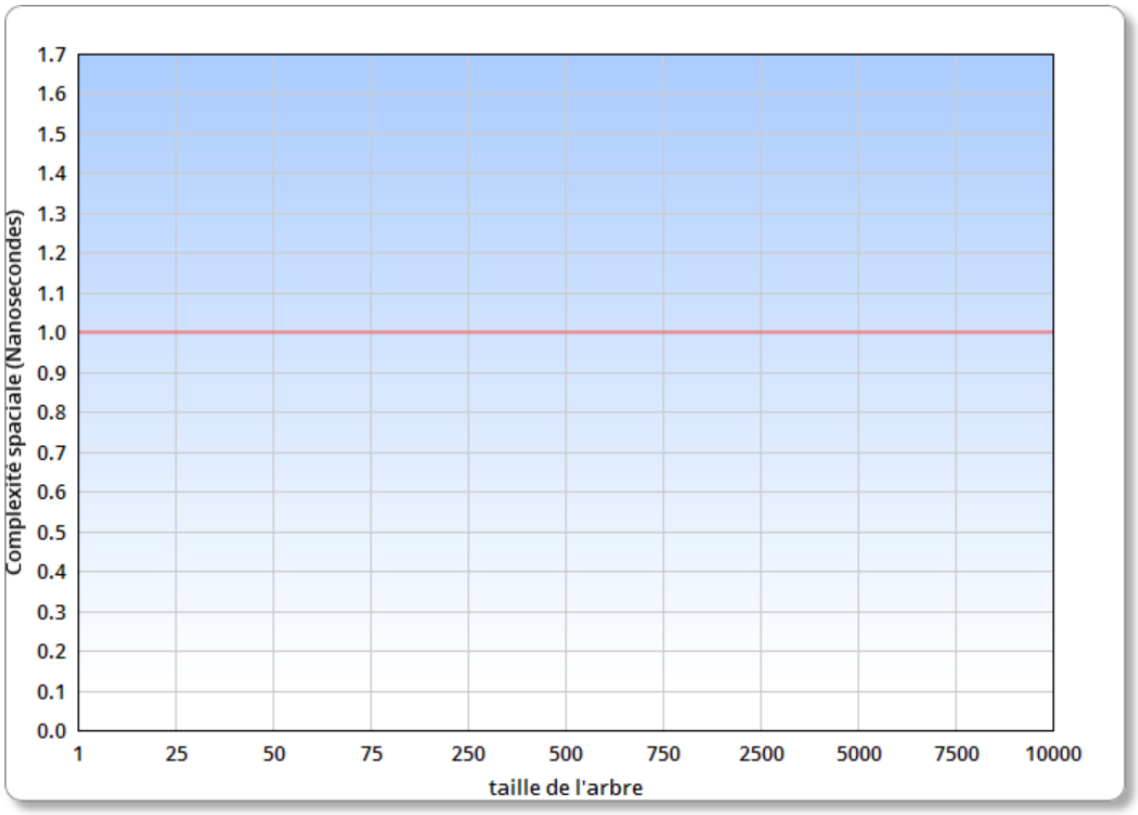
```
******************** TEMPS D'EXECUTION EN NANOSECONDES ********************
temps d'execution en nanosecondes:178852743400ns
```

Graphe de compléxité réelle en fonction de la taille du probléme (taille de l'arbre) :



Valeurs	Taille=1	Taille=25	Taille=50	Taille=75	Taille=250	Taille=500	Taille=750	Taille=2500	Taille=5000	Taille=7500	Taille=10000
Complexité réels (Nanosecondes)	844800	415527000	795461300	1337291000	3956531800	8177579800	12070913400	39079834400	81577598600	120757121100	178852743400
Complexité théorique (Nanosecondes)	1	1	1	1	1	1	1	1	1	1	1

Graphe de compléxité théorique en fonction de la taille du probléme (taille de l'arbre) :



3)e Conclusion:

D'après les résultats obtenus pour le parcours (affichage) d'un arbre binaire de recherche on remarque que plus le nombre d'éléments de l'arbre sera grand plus le temps d'exécution augmentera , pour l'affichage la complexité est la même que si on utiliser une autre structure de données (tableau,liste..) O(n) par contre l'avantage d'utiliser les arbres binaires de recherche apparait lors de l'insertion , la suppression ou la recherche d'un élément car la complexité temporelle sera dans ce cas là logarithmique (à chaque fois une moitié de l'arbre sera éliminé)O(log(n)) et on sait que les algorithmes de complexité logarithmique ont un temps d'exécution très faible donc dans ces cas les arbres binaires de recherche sont les plus efficaces car elle nous permet d'optimiser du temps d'execution.