Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediene
Faculté d'Informatique
Département IA et SD

Rapport devoir maison n° 3

Module : Conception et Complexité des Algorithmes

Traitement sur les arbres

Travail présenté par :

- **BENKOUITEN Aymen————-191931046409**

2022/2023

Table des matières

Table des figures

Liste des tableaux

Première partie

Développement de l'algorithme et du programme correspondant.

0.1 Pseudo code et la complexité

Les pseudos code des algorithmes :

Question 03 :

Version recursive :

Pseudo code :

```
entier rec_findElem (bt : BTree, e : Element)
Debut
  Si (arbrevide (bt)) Alors  retourner 0 ;
    Si non
      si (root (bt) == e)  Alors   retourner 1 ;
        Si non retourner rec_findElem(leftChild(bt),e)|| rec_findElem(rightChild(bt),e)

      Fsi ;
  Fsi ;
Fin.
```

FIGURE 1 – pseudo code de la fonction recfindElem()

Complexité de l'algorithme : la complexité $O(n)$, dans le cas récursive on s'intéresse au pire cas qui veut dire l'élément recherche est une feuille donc c'est le dernier niveau de l'arbre pour cela on doit parcourir toute l'arbre qui veut dire la complexité c'est $O(n)$.

Version itérative :

Pseudo code :

```
entier ite_findElem (x : *Node, e : Element)
Debut
  Si(x == NULL) Alors  retourner 0 ;  Fsi ;
  pt : *pile ;
  pt = new_pile(3) ;
  Empiler (pt, x) ;
  Tant que(nonpilevide(pt))
  Faire
     x : *Node ;
     x = depiler(pt) ;
     Si(x.elem == e) Alors retourner 1 ;  Fsi ;
     Si(x.left) Alors empiler(pt, x.left) ;  Fsi ;
     Si(x.right) Alors empiler(pt, x.right) ;  Fsi ;
  Faits
    retourner 0 ;
Fin.
```

FIGURE 2 – pseudo code de la fonction itefindElem()

Complexité de l'algorithme :

la complexité $O(n)$, dans le cas itératif le même principe dans le pire cas l'élément et la hauteur la plus grande dans une arbre qui veut dire empiler et dépiler toutes les nœuds de l'arbre ce dernier mène a dire que la complexité c'est $O(n)$.

Question 04 :

Version recursive :

Pseudo code :

```
Element rec_findMin (bt : BTree)
Debut
Si (arbrevide(bt)) Alors ecrire('min impossible') ; Fsi ;
 Si (isLeaf(bt)) Alors root(bt) ;
    Si non   m : Element ;
                m = root(bt) ;
                Si(non arbrevide(leftChild(bt))) Alors
                    m=min(m, rec_findMin(leftChild(bt))) ;
                Fsi ;
                Si(non arbrevide(rightChild(bt))) Alors
                    m=min(m, rec_findMin(rightChild(bt))) ;
                Fsi ;
                retourner m ;
 Fsi ;
Fin.
```

FIGURE 3 – pseudo code de la fonction recfindMin()

Complexité de l'algorithme : Dans les appelle récursive chaque nœud de l'arbre est traite une seule fois donc la complexité c'est le nombre de fois quand parcours un nœud on conclut quand a parcourir N nœud donc la complexité c'est $O(n)$.

Version itérative :
Pseudo code :

```
entier ite_findMin (x : *Node)
Debut
  x : entier ;
  si  (x == NULL) Alors retourner 0 ;
    si non  min = root(x) ;
  Fsi ;
  pt : *pile ;
  pt = new_pile(3) ;
  Empiler (pt, x) ;
  Tant que (nonpilevide(pt))
  Faire
    x : *Node ;
    x = depiler(pt) ;
    Si (x.elem < min) Alors min = x.elem ; Fsi ;
    Si (x.left) Alors empiler (pt, x.left) ;  Fsi ;
    Si (x.right) Alors empiler (pt, x.right) ;  Fsi ;
  Faits ;
  retourner min ;
Fin.
```

FIGURE 4 – pseudo code de la fonction itefindMin()

Complexité de l'algorithme :
On doit empiler tous les nœuds dans la pile.

Question 05 :

Version recursive :
Pseudo code :

```
rec_printWidhtOrder(root : *Node)
Debut
  h, i : entier ;
  h = height(root) ;
  pour(i de 1 à h)
  Faire
    printCurrentLevel(root, i) ;
  Fait ;
Fin.
```

```
printCurrentLevel (root : *Node, level :entier)
Debut
  Si (root == NULL)  Alors retourner 0 ;  Fsi ;
  Si (level == 1)   Alors ecrire('root.elem') ;
    Si non si (level > 1)  Alors printCurrentLevel(root.left, level - 1) ;
                                 printCurrentLevel(root.right, level - 1) ;
            Fsi ;
  Fsi ;
Fin.
```

FIGURE 5 – pseudo code de la fonction recPrintWithOrder()

```
entier height(node : *Node)
Debut
  Si (node == NULL)  Alors retourner 0 ;
    Si non
      Lheight = height(node.left) ;
      Rheight = height(node.right) ;
      Si (lheigh > rheigt)
          retourner (lheight + 1) ;
          si non  retourne (rheight + 1) ;
    Fsi ;
  Fsi ;
Fin.
```

FIGURE 6 – pseudo code de la fonction height()

Complexité de l'algorithme : la complexité $O(n)$.

Version itérative :
Pseudo code :

```
ite_printWidhtOrder (x : *Node)
Debut
  Si(x == NULL) Alors ecrire('Arbre vide') ;
    Si non
            file *q ;
            q = file_new() ;
            enfiler (q, x) ;
            Tant que (q != NULL)
            Faire
              n : *Node ;
              n = defiler(q) ;
              ecrire (root(n)) ;
              Si (n.left) Alors enfiler (q, n.left) ;
              Si (n.right) Alors enfiler (q, n.right) ;
            Fait ;
 Fsi ;
Fin.
```

FIGURE 7 – pseudo code de la fonction itePrintWithOrder()

Complexité de l'algorithme :
la complexité $O(n)$.

Deuxième partie

Mesure du temps d'exécution et illustation graphique.

0.2 Illustration graphique

Question 03 :

0.2.1 Temps d'execution des trois fonctions en deux versions

	VERSION	10000	50000	70000	80000	100000	500000	900000	1000000
findElem	récursive	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.03
	itérative	0.00	0.00	0.00	0.00	0.00	0.01	0.05	0.05
findMin	récursive	0.00	0.009	0.00	0.00	0.02	0.07	0.20	0.19
	itérative	0.00	0.01	0.01	0.02	0.01	0.04	0.15	0.20
printWidthOrder	récursive	1.82	9.51	14.42	18.91	26.43	—	—	—
	itérative	0.00	0.00	0.20	0.39	0.77	5.26	9.52	13.81

FIGURE 8 – Temps d'execution des 3 questions en deux versions

Représentation graphique :

findElem() :

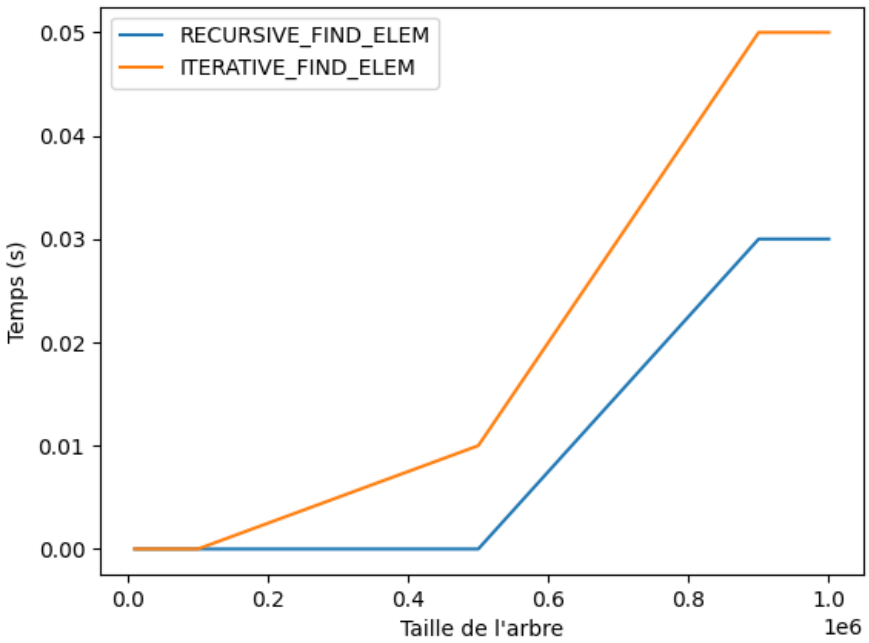

FIGURE 9 – Représentation graphique des résultats d'exécution de findElem() en deux versions

findMin() :

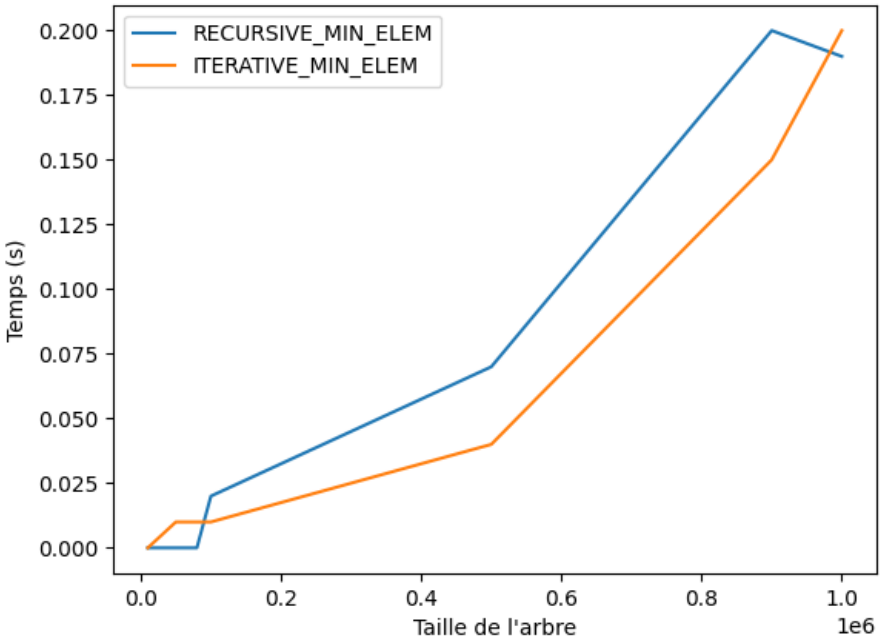

FIGURE 10 – Représentation graphique des résultats d'exécution de findMin() en deux versions

printWidthOrder() :

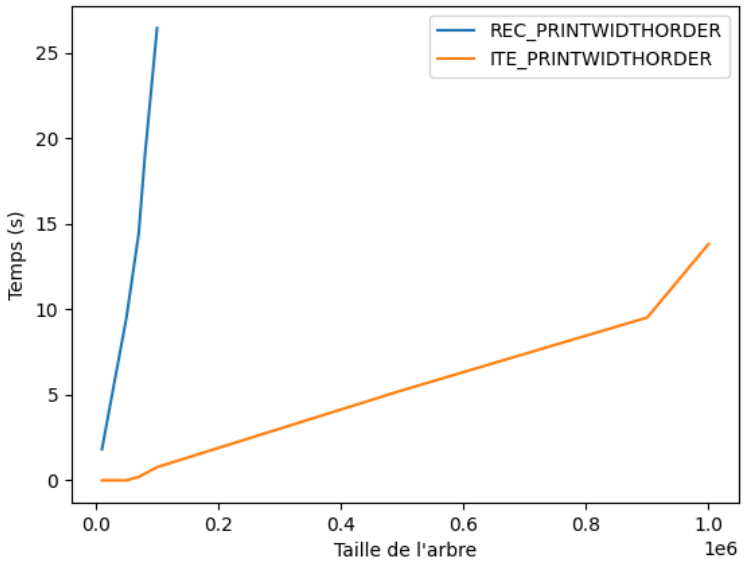

FIGURE 11 – Représentation graphique des résultats d'exécution de printWidthOrder() en deux versions

Analyse globale :

On remarque à partir de ces derniers graphes que dans la racherche d'un element dans un graphe l'algorithme de la verion itérative est le plus rapide mais quand on doit afficher une arbre la version recursive est la plus optimale et rapide et on remarque aussi que dans la recherche de minimum dans une arbre les deux versions recursive et itérative sont presque identique.