Durée : 01h30'

EXAMEN

Lundi 08 Janvier 2024

ALGORITHMIQUES ET COMPLEXITE

<u>Exercice n°1 :</u> (4pts)

Analyser la complexité des deux algorithmes suivants. Que fait chaque algorithme ?

```
int i=2;
while (i*i<=n)
{
  if (n%i==0) return false;
  i++;
}
return true;
```

```
void fctA(int *A, int n, int k) {
int i=0;
while (i<n) {if (A[i]==k) fctB(A,n,i); else i++;}
}
void fctB(int *A, int &n, int i) {
while (i<n-1) { A[i]=A[i+1]; i++; }
n--;
}
```

<u>Exercice n°2 :</u> (4pts)

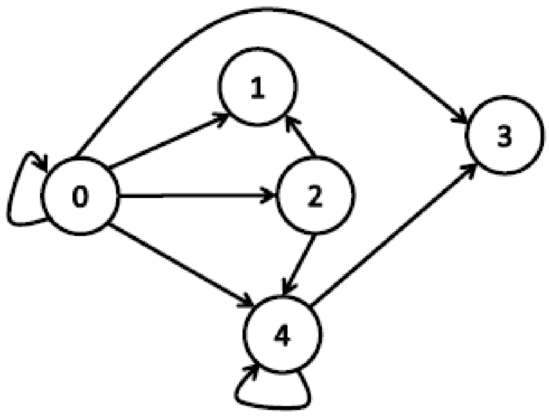
1. Donner la structure de données résultante de l'insertion des clés 1,2,3,4,5,6,7 dans cet ordre dans le cas d'un : (Indiquer pour les cas a et b, le type ou la particularité de l'arbre obtenu)
   a. Arbre binaire de recherche (BST) : Donner le facteur d'équilibrage pour chaque nœud.
   b. Arbre binaire de recherche équilibré (AVL) : Indiquer les opérations de rotations utilisées.
   c. Min-Tas (min-Heap)
   d. Max-Tas (max-Heap)
2. Donner pour chaque cas la structure de données résultante après suppression de la racine.

<u>Exercice n°3 :</u> (6pts)

Etant donné un AVL de taille N représenté de manière dynamique. Ecrire un algorithme permettant d'afficher les clés comprises entre deux clés données a et b (avec $a \leq b$). Analyser la complexité de votre algorithme. Si votre algorithme n'a pas une complexité strictement inférieur à O(N), proposer une amélioration à votre algorithme.

<u>Exercice n°4 :</u> (6pts)

Etant donné le graphe orienté suivant :



1. Donner la représentation du graphe en matrice d'adjacence puis en listes d'adjacence contiguës. Donner la complexité spatiale de chaque structure.

2. Considérons les poids des arcs suivants :
   w(0,0)=5, w(0,2)=w(2,1)=10, w(0,1)=w(2,4)=20,
   w(0,4)=w(4,3)=50 et w(0,3)=w(4,4)=90.
   Dérouler l'algorithme de Dijkstra à partir du sommet 0 et indiquer les chemins optimaux trouvés.

3. Ecrire un algorithme qui prend en entrée un graphe orienté représenté en listes d'adjacence chaînées et retourne **true** si le graphe contient un triangle. Un triangle dans un graphe est un triplet de sommets (u,v,w) tel que les 3 arcs (u,v), (v,w) et (u,w) appartiennent au graphe. Analyser la complexité de votre algorithme.

Bon courage…

Solution & Barème

Exercice n°1 : (4pts : 2pts par programme = 1.5pts complexité + 0.5pt objectif du prg)

Programme 1 :

```
int i=2;
while (i*i<=n)
{ if (n%i==0) return false;
  i++; }
return true;
```

$i^2 <= \ n\ donc\ i <= \sqrt{n}, la\ complexité\ est\ donc\ O(\sqrt{n})$

Ce programme teste si le nombre n est premier.

Programme 2 :

```
void fctA(int *A, int n, int k) {
int i=0;
while (i<n) {if (A[i]==k) fctB(A,n,i); else i++;}}
void fctB(int *A, int &n, int i) {
while (i<n-1) { A[i]=A[i+1]; i++; }
n--;}
```

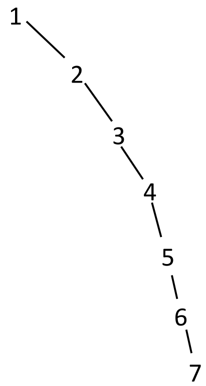
Dans la fonction fctA, on parcourt le tableau jusqu'à trouver l'élément k puis dans la fonction fctB on continue le parcours jusqu'à la fin du tableau pour faire les décalages. Si l'élément existe une seule fois, la complexité est donc O(n), sinon dans le pire des cas la complexité sera $O(n^2)$.

Ce programme supprime toutes les occurrences de l'élément k du tableau A.

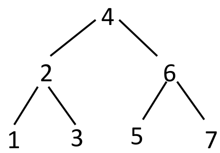
Exercice n°2 : (4pts)

1. Donner la structure de données résultante de l'insertion des clés 1,2,3,4,5,6,7 dans cet ordre dans le cas d'un : (Indiquer pour chaque cas, le type ou la particularité de l'arbre obtenu)

a. BST (0.75pt : 0.25 BST + 0.25 Arbre dégénéré + 0.25 facteur d'équilibrage)



b. AVL : (1.5pts : 1 AVL + 0.25 Arbre complet + 0.25 Rotation simple à gauche)



c. min-Heap : (0.25pt : min-Heap)

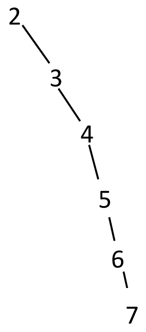
1	2	3	4	5	6	7

d. max-Heap : (0.5pt : max-Heap)

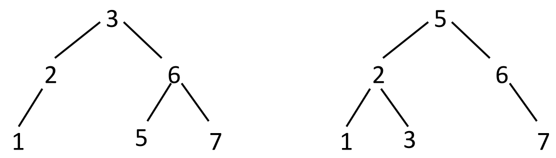
7	4	6	1	3	2	5

2. Suppression de la racine. (1pt : 0.25 pour chaque structure)

a. (BST)



b. AVL : l'un des deux AVL suivants :



c. min-Heap :

2	4	3	7	5	6

d. max-Heap :

6	4	5	1	3	2

<u>Exercice n°3 :</u> (6pts, Algorithme naïf : 3pts, Algorithme optimisé : 5pts, Complexité : 1pt)

(50% de la note pour le principe de fonctionnement)

Algorithme naïf :
Faire un parcours (infixe par exemple) et afficher toute clé comprise entre a et b.

```
void affichCles(Noeud *racine, int a, int b) {
      if (racine!=NULL) {
            affichCles(racine->filsG,a,b);
            if (racine->val>=a && racine->val<=b) cout<<racine->val<<endl;
            affichCles(racine->filsD,a,b);
      }
}
```

Complexité : T(N) = 2T(N/2) + O(1) = O(N)

Algorithme optimisé :
Eviter de parcourir les sous arbres dont les clés ne peuvent pas être dans l'intervalle [a,b]. On compare la clé du nœud courant avec a et b, puis on décide quel sous arbre explorer en fonction de cela :

```
void affichCles(Noeud *racine, int a, int b) {
      if (racine==NULL) return;
      if (racine->val < a) affichCles(racine->filsD,a,b);
      if (racine->val > b) affichCles(racine->filsG,a,b);
      if (racine->val >= a && racine->val <= b) {
            cout<<racine->val<<endl;
            affichCles(racine->filsG,a,b);
            affichCles(racine->filsD,a,b);
      }
}
```

Dans ce cas, la complexité est strictement inférieur à O(N) et est égal au nombre de clés comprises entre a et b.

Exercice n°4 : (6pts)

1. Donner la représentation du graphe en matrice d'adjacence puis en listes d'adjacence contiguës. (1pt : 0.5 par structure = 0.25 SD + 0.25 Complexité)

Matrice d'adjacence : Complexité spatiale $N^2 = O(N^2)$

	0	1	2	3	4
0	1	1	1	1	1
1	0	0	0	0	0
2	0	1	0	0	1
3	0	0	0	0	0
4	0	0	0	1	1

Listes d'adjacence : Complexité spatiale N+M+1 = O(N+M)

Tete

0	1	2	3	4	5
0	5	5	7	7	9

Succ

0	1	2	3	4	5	6	7	8
0	1	2	3	4	1	4	3	4

2. Algorithme de Dijkstra (1.25pts : 0.75pt Value array, 0.25 Path array, 0.25 Path details)

0	1	2	3	4	Mark
0	∞	∞	∞	∞	0
	20	10	90	50	2
	20		90	30	1
			90	30	4
			80		3
0	20	10	80	30	Final result

0	1	2	3	4
0	-1 0	-1 0	-1 0 4	-1 0 2

0 → 0 : 0 0 → 1 : 20 0 → 2 : 10 0 → 2 → 4 : 30 0 → 2 → 4 → 3 : 80

3. Détection de triangles : (u,v,w) tel que les 3 arcs (u,v), (v,w) et (u,w) appartiennent au graphe. (3.75pts : 3pts algorithme, 0.75pt complexité, +1 si algorithme optimisé) (50% de la note pour le principe de fonctionnement)

Principe de fonctionnement :

Algorithme naïf :

Générer tous les triplets de sommets (u,v,w) en $O(N^3)$ et tester l'existence des arcs (u,v), (v,w) et (u,w) en $O(d^+)$ ou O(N) dans le pire des cas, ce qui donne une complexité $O(MN^2)$ ou $O(N^4)$ (pire cas)

Algorithme optimisé :

Pour chaque sommet u, parcourir ses successeurs v. pour chaque successeur v, parcourir ses successeurs w. Pour chaque sommet w tester l'existence de l'arc u. Ceci donne une complexité de O(MN) ou dans le pire des cas de $O(N^3)$