Henry
Letellier
T1

mercredi 13 janvier 2021

Numériques et sciences de l'informatique (NSI)
DST d'NSI
Tiers Temps

Numériques et sciences de l'informatique (NSI)

DST d'NSI

Note : _____	Appréciations :	Signature :

Exercice 1 :

1)

A l'initialisation du programme, la pile P contient la suite 4,2,5,8 et la pile Q n'existe pas.

A la fin de l'exécution du programme, la pile P sera vide et la Pile Q contiendras la suite : 4,2,5,8

2)

a)

```
def hauteur_pile(P) :
    Q=creer_pile_vide()
    n=0
    while not(est_vide(P)) :
        n+=1
        x=depiler(P)
        empiler(Q,x)
    while not(est_vide(P)) :
        x=depiler(Q)
        empiler(P,x)
    return n
```

Les éléments en gras sont ceux qu'il fallait mettre à la place des « ??? ».

b)

Le code pour la fonction **max_pile** est :

```
def max_pile(P,i):                                  #On défini un def
    assert i <= hauteur_pile(P), "La liste est vide"  #On vérifie que la liste est pleine
    indice=0                                        #On innitialise la variable indice à 0
    maxo=0                                          #On fait de même pour maxo
    j=0                                             #On fait de même pour j
    Q=creer_pile_vide()                             #On créé une pile Q vide
    while j<=i:                                     #Tant que j est inférieur à i on boucle
        F=depiler(P)                                #On dépile le premier élément de P dans la variable F
        if F>=maxo:                                 #On vérifie que F est supérieur ou égal à maxo
            maxo=F                                  #Si c'est le cas, maxo prend la valeur de F
            indice=j                                #La variable indice prend la valeur de J
        j+=1                                        #On incrémente j de 1
        empiler(Q,F)                                #On empile F dans Q
    while not(est_vide(Q)==True):                   #Tant que Q est pleine
        empiler(P,depiler(Q))                       #On dépile Q dans P
    return indice                                   #A la fin de l'exécution on renvoie la valeur de la variable indice.
```

Ainsi, pour la pile P contenant les valeurs 4,8,2,5 et un appel **max_pile(P,2)** retourneras **1** car il retournera l'emplacement de la plus grande valeur qui est **8**. Dans le cas de figure de ce programme, la liste commence à 0 et non à 1.

3)

Le code pour la fonction **retourner** est :

```
def retourner(P,j):                                  #On défini un def
    assert j <= hauteur_pile(P), "La liste est vide" #On vérifie que la liste est pleine
    Q=creer_pile_vide()                              #On créé une pile Q vide
    T=creer_pile_vide()                              #On créé une pile T vide
    rang=0                                           #On innitialise la variable rang à 1
    while rang<=j:                                   #Tant que rang est inférieur à j on boucle
        empiler(Q,depiler(P))                        #On dépile le premier élément de P dans Q
        rang+=1                                      #On incrémente la variable rang de 1
    while not(est_vide(P)==True):                    #Tant que P est pleine on boucle
        empiler(T,depiler(P))                        #On dépile le premier élément de Q dans T
    long=hauteur_pile(Q)                             #La variable long prend la valeur de la hauteur de la pile Q

    for i in range(long-1):                          #On boucle pour la valeur -1 de la variable long
        empiler(Q,depiler(Q))                        #On empile Q dans Q
    while not(est_vide(Q)==True):                    #Tant que Q est pleine
        empiler(P,depiler(Q))                        #On dépile Q dans P
    while not(est_vide(T)==True):                    #Tant que T est pleine
        empiler(P,depiler(T))                        #On dépile T dans P
    return None                                      #On renvoie None.
```

Si on donne la liste P=[4,2,5,8] à la fonction **retourner(P,3)**, cette dernière ne retourneras rien.

En affichant P, cette dernière vaut alors : [8, 4, 2, 5]

4)

Le code pour la fonction **tri_crepes** est :

def **tri_crepes**(P): assert not(est_vide(P)), "Il n'y a pas de crêpes à trier"	#On défini un def #On vérifie que la liste est pleine
Q=creer_pile_vide() h=hauteur_pile(P) for i in range(0,h-1):	#On créé une pile Q vide #On innitialise h avec la hauteur de la pile P #On boucle pour la valeur -1 de la variable h tout en commençant par la valeur 0
rang_maxi=max_pile(P,h-i-1)	#On innitialise rang_maxi avec le résultat de la fonction max_pile(P,h-i-1)
retourner(P,rang_maxi)	#On retourne la liste P une première fois tout en se servant de la valeur de rang_maxi comme index
empiler(Q,depiler(P)) while not(est_vide(Q)==True): empiler(P,depiler(Q)) return None	#On dépile le premier élément de P dans Q #Tant que Q est pleine #On dépile Q dans P #On renvoie None.

L'exécution de la fonction **tri_crepes** avec P contenant [6,3,9,5] donneras une pile P contenant les valeurs [9,6,5,3].

Exercice 2 :

1)

a)

Pour arriver à la case (2,3) en partant de la case (0,0) à on feras 3 déplacements vers la droite et 2 déplacements vers le bas.

b)

Comme on se déplace d'un pas à chaque étape, il faudra faire 2+3 cases, soit 5 déplacements. Chaque déplacement nous amène une nouvelle case. En n'oubliant pas d'inclure la case (0,0) il faut donc parcourir 2+3+1 cases, soit 6 cases.

2) Le tableau contenant tous les chemins est :

Chemins	Somme
(0,0) → (0,1) → (0,2) → (0,3) → (1,3) → (2,3)	11
(0,0) → (0,1) → (0,2) → (1,2) → (1,3) → (2,3)	10
(0,0) → (0,1) → (0,2) → (1,2) → (2,2) → (2,3)	14
(0,0) → (0,1) → (1,1) → (1,2) → (1,3) → (2,3)	9
(0,0) → (0,1) → (1,1) → (1,2) → (2,2) → (2,3)	13
(0,0) → (0,1) → (1,1) → (2,1) → (2,2) → (2,3)	12
(0,0) → (1,0) → (1,1) → (1,2) → (1,3) → (2,3)	10
(0,0) → (1,0) → (1,1) → (1,2) → (2,2) → (2,3)	14
(0,0) → (1,0) → (1,1) → (2,1) → (2,2) → (2,3)	13
(0,0) → (1,0) → (2,0) → (2,1) → (2,2) → (2,3)	16

La plus grande valeur est de la somme maximal est 16.

3)

a. Le Tableau T' est le suivant :

4	5	6	9
6	6	8	10
9	10	15	16

b.

La valeur T'[0][j] où j est non nul correspondra à la somme des cases (0,0) à (0,j), soit les cases de la première ligne du tableau.

Il n'y a qu'un seul chemin qui corresponde à cette somme mais il passe obligatoirement par la case à gauche (d'indice j-i).
Donc pour calculer la somme T'[0][j] on ajoute simplement la valeur de la case (0,j) (c'est-à-dire T[0][j]) à la somme obtenue à la case précédente (c'est-à-dire T' [0][j-1]).
On a donc bien T'[0][j] = T[0][j]+T'[0][j-1].

4)

Si i et j sont non-nuls, il y aura deux chemins menant à la case (i, j).
Le premier proviendra de la case du dessus (i-1,j), le second de la case de gauche (i,j-1).
La valeur de T'[i][j] s'obtiendra donc en ajoutant la valeur de T[i][j] au maximum des deux chemins menant à cette case, soit : max(T'[i-1][j],T'[i][j-1]).

5)

a.

Le cas de base est atteint lorsque l'on atteint une case de la première ligne (i vaut 0) ou de la première colonne (j vaut 0). Dans ce cas on calcule la somme en additionnant la valeur de la case en question avec le résultat de somme_max avec comme argument T est la case précédente (sur la ligne si i=0 ou la colonne si j=0).

b.

Le code pour la fonction **somme_max** est :

def **somme_max**(T,i,j):	#On défini un def
if i==0 and j==0 :	#Si i et j sont nul
return T[0][0]	#On renvoie T[0][0]
elif i==0 :	#Sinon, si i est nul
return T[0][j]+somme_max(T,0,j-1)	#On renvoie T[0][j] plus la fonction somme_max(T,0,j-1)
elif j==0 :	#Sinon si j est nul
return T[i][0]+somme_max(T,i-1,0)	#On renvoie T[i][0] plus la fonction somme_max(T,i-1,0)
else :	#Sinon
return T[i][j]+max(somme_max(T,i-1,j), somme_max(T,i,j-1))	# On renvoie T[i][j] plus la fonction max de somme_max(T,i-1,j) et somme_max(T,i,j-1)

c.

On appelle somme_max(T,2,3),

avec T étant égal aux valeurs suivantes [[4,1,1,3],[2,0,2,1],[3,1,5,1]] pour exécuter la fonction.

Exercice 3 :

1)

La taille d'un arbre dépend du nombre de nœuds qu'il comporte. Dans le cas présent, l'arbre vaut 9. La hauteur est la longueur du plus long chemin entre la racine et l'une des feuilles. Dans notre cas : 4.

2)

a.

1010 est le numéro qui est associé à G.

b.

en binaire 1101 est le chiffre 13. Ce dernier correspondra donc au nœud numéro 1

c.

Les nœuds les plus bas sont numérotés sur plusieurs bit (ici : 4).

d.

Un arbre de hauteur h peut avoir au minimum n nœuds (nœud par niveau).

Donc $h \leq n$.

A l'autre extrême, si l'arbre est complet, c'est-à-dire que tous les niveaux sont remplis, alors la racine aura pour identifiant 1.

Les nœuds du premier niveau auront pour identifiant 10 et 11, ceux du deuxième niveau 100, 101, 110 et 111, et ainsi de suite.

Les identifiants des nœuds du niveau h s'écrirons sur h bits.

L'arbre étant complet, tous les nombres entier pouvant s'écrire sur h bits correspondrons à des identifiants sauf la valeur 0 car la racine est le 1.

Il y aura 2^h valeurs possibles sur h bits.

Donc en retirant le 0 on obtiendra 2^h-1.

On aura donc bien : $h \leq n \leq 2^h-1$.

3)

a.

Le tableau est qui représente l'arbre binaire est :

[None,"A","B","C","D","E","F","G","H","I","J","K","L","M","N","O"]

b.

L'indice du père d'un nœud d'indice $i \geq 2$ sera le quotient entier de i divisé par 2, soit i//2 en python.

4)

Le code pour la fonction **recherche** est :

```
def recherche(arbre, element) :                    #On défini un def
    taille=len(arbre)                              #On innitialise taille avec la longeur de la pile P
    i=1                                            #On innitialise i à 1
    while i<taille :                               #On boucle tant que i est inférieur à taille.
        if abre[i]==element :                      #Si abre[i] est égal à element
            return True                            #On retourne True
        elif element<arbre[i] :                    #Sinon si element est inférieur à arbre[i]
            i=2*i                                  #On affecte à i l'opération 2*i
        else :                                     #Sinon
            i=2*i+1                                #On affecte la l'opérations 2*i+1 à i
    return False                                   #On retourne False
```

Exercice 4 :

1)

a.

L'attribut **num_eleve** est la clé primaire, cette dernière permettras d'identifier de façon certaine des objets de la table (ou relation).

b.

La requête SQL permettant d'enregistrer l'élève ACHIR Mussa est :

```
INSERT INTO seconde(num_eleve, langue1,langue2,classe)     #On insère dans seconde les variables num_eleve, langue1, langue2, classe
VALUES ("133310FE","anglais","espagnol","2A")              #On affecte des données à ces variables.
```

c.

La requête SQL de mises à jour permettant de corriger les données étonnées est :

```
UPDATE seconde                                 #On informe SQL que l'on va modifier le tableau seconde
SET langue1="allemand"                         #On affecte une nouvelle valeur à la variable langue1 (variable qui était erronée)
WHERE num_eleve = "156929JJ"                   #On donne l'identifiant de la ligne à changer.
```

2)

a.

Le résultat de la requête **SELECT num_eleve FROM seconde;** renverras les numéros d'identification de tous les élèves de seconde. Il s'agit donc de toutes les données de la première "colonne" du fichier csv

b.

Le résultat de la requête **SELECT COUNT(num_eleve) FROM seconde;** compteras le nombre d'élèves de seconde. Pour pouvoir afficher un résultat valide de cette fonction sur la base seconde.csv, il faudrait que nous disposions de la totalité du tableau, et non d'un extrait qui se trouve sur la page précédente.

c.

La requête qui permettras de connaitre tous les élèves qui font allemand en langue1 ou langue2 est :

SELECT COUNT(*) FROM seconde	#On sélectionne tous les éléments du tableau seconde
WHERE langue1 = "allemand" OR langue2="allemand"	#On localise la colonne langue1 puis vérifie si la donnée est allemand, on localise aussi la colonne langue2 puis on vérifie si la donnée est allemand #Si l'une des deux conditions est validée, rien d'autre ne se passe pour permettre d'en informer l'utilisateur.

3)

a.

L'ajout d'une clé étrangère permet de s'assurer que les données des tables se correspondent. Elle peut aussi permettre d'empêcher d'ajouter des objets dans une table s'ils ne sont pas présents dans l'autre.

b.

La commande qui permet d'établir cette liste est :

SELECT nom, prenom, datedenaissance FROM eleve	#On sélectionne les variables nom, prenom, datedenaissance du tableau eleve
JOIN seconde ON seconde.num_eleve=eleve.num_eleve	#On lie les variables num_eleve de la colonne num_eleve sur le tableau seconde
WHERE seconde.classe="2A"	#On informe que la colonne classe "2A" est la destination.

4)

La structure d'une table **coordonnees** est la suivante :

coordonnées		
num_eleve	clé primaire	clé étrangère de la table seconde
adresse		
code postal		
ville		
mail		

Exercice 5 :

1)

a.

En lisant la table de routage de A suivie de celle de C et de F on obtient A→C→F→G.

b.

On pourras donc avoir comme trajet:

Table de routage du routeur G		
Destination	Routeur suivant	Distance
A	E	3
B	E	3
C	E	2
D	E	2
E	E	1
F	F	1

2)

On aura donc la table suivante

Table de routage du routeur A		
Destination	Routeur suivant	Distance
B	B	1
D	D	1
E	D	2
F	D	4
G	D	3