Nom	
Prénom	
Groupe	E1

Note	13/20

1	1
2	2
3	3
4	1,5
5	2,5
6	3

Algorithmique
INFO-SUP S2
Contrôle n° 2 (C2)
20 février 2023 - 8 : 30
Feuilles de réponses

Réponses 1 **(Arbre général – *2 points*)**

Si un ordre n'existe pas, écrire ∅ dans le champ de réponse.

La liste des nœuds de l'arbre T rencontré en ordre préfixe est :

15-3-(-6)-1-8-11-0-4-2-5-9 0,5

La liste des nœuds de l'arbre T rencontré en ordre infixe est :

~~(-6)-3-1-15-0-11-4-8-2-5-9~~ 0

La liste des nœuds de l'arbre T rencontré en ordre suffixe est :

(-6)-1-3-0-4-11-2-5-9-15 0,5

1

Réponses 2 **(Arbre binaire de recherche – *2 points*)**

	OUI	NON	Valeur si NON
23 - 81 - 70 - 35 - 56 - 38 - 40 - 42	X		
70 - 62 - 18 - 36 - 53 - 91 - 49 - 42		X	91
15 - 65 - 19 - 49 - 61 - 57 - 55 - 42		X	61
28 - 32 - 33 - 81 - 55 - 45 - 37 - 42	X		

2

Réponses 3 **(Jeu de dames –** ***4 points*****)** 3

1. **Spécifications :**
   La fonction `check_line(L, n)` vérifie si les valeurs booléenne de la liste `L` de longueur `n` alternent entre `True` et `False`.

```
def check_line(L,n):
    if n<1:
        return True
    else:
        s= not L[0]
        i=1
        while i<n and L[i]==s:
            s=not s
            i+=1
        return i==n
```

2. **Spécifications :**
   La fonction `check_checkers(M)` prend une matrice (supposée non vide) de valeurs booléennes `M` en paramètre, et vérifie si la matrice `M` est un plateau de dames.

```
def check_checkers(M):
    l=len(M)
    s=True
    i=0
    while i<l and M[i][0]==s and check_line2(M[i],l):
        s=not s
        i+=1
    return i==l
```

* Même fonction que la précédente sans un test inutile

```
*def check_line2(L,n):
    s=not L[0]
    i=1
    while i<n and L[i]==s:
        s=not s
        i+=1
    return i==n
```

Réponses 4 (**Tas** – *2 points*)

Spécifications :

La fonction `heappush(H, elt, val)` prend en paramètres un tas représenté par un vecteur `H`, un élément à ajouter `elt` (peut être de n'importe quel type) et sa valeur `val` (un entier), et ajoute la paire `(elt, val)` au tas `H`.

```
def heappush(H,elt,val):
    l=len(H)
    if l==0:
        return H.append((elt,val))
    else:
        i,s=l,l
        H.append((elt,val))
        if i%2==0:
            i//=2
        else:
            i-=1
            i//=2
        while i>0:
            if H[i][1]>val:
                H[i],H[s]=H[s],H[i]
                i,s=i//2,s//2
        return H
```

Réponses 5 *(n^{ieme}* **noeud** *– **5 points**)*

Spécifications :

La fonction `get_node(B, lvl, n)` prend en paramètres un arbre binaire `B`, deux entiers `lvl` et `n`, et retourne la clé du n^{ieme} noeud (base 0) du niveau `lvl` de `B` s'il existe, `None` sinon. Les entiers `lvl` et `n` sont supposés positifs ou nuls.

```
def get_node(B, lvl, n):
    if B == None:
        return None
    else:
        q = Queue()
        q.enqueue(B)
        q.enqueue(None)
        num = -1
        lv = 0
        while not q.isempty() or (lv != lvl and n != num):
            noeud = q.dequeue()
            num += 1
            if noeud == None:
                num = 0
                lv += 1
                q.enqueue(None)
            else:
                if noeud.left != None:
                    q.enqueue(B.left)
                if noeud.right != None:
                    q.enqueue(B.right)
        if lv == lvl and n == num:
            return noeud.key
        else:
            return None
```

Réponses 6 **(Parenté – *5 points*)**

Spécifications :

La fonction `get_kinship(B, x, y)` prend en paramètres un arbre binaire `B`, deux valeurs `x` et `y`, et renvoie le degré de parenté entre les noeuds de `B` contenants `x` et `y` s'ils sont sur la même branche, -1 sinon.

```
def aux_y(B,y,n):                    Calcule la distance entre
    if B.key==y:                       le noeud B et le noeud
        return(n+2)                        contenant y
    elif B.left==None                                  * n=-1
        if B.right==None
            *return None
        else:
            return aux_x(B.right,x,n+1)
    else:
        if B.right==None:
            return aux_x(B.left,x,n+1)
        else:
            l=aux_x(B.left,x,n)
            if l!=None:
                return l
            else:
                return aux_x(B.right,x,n)

Recherche l'élément x et retourne
def aux_x(B,x): le noeud le          def get_kinship(B,x,y):
    if B==None:  contenant               if B==None:
        return None                          return -1
    else:                                else:
        if B.key==x:                         V=aux_x(B,x)
            return B                         if V!=None:
        l=aux_x(B.left,x)                        return aux_y(V,y,-1)
        if l!=None:                          else:
            return l                             return(-1)
        else:
            return aux_x(B.right,x)
```