Nom :____

Code permanent : ____________________________________

Numéro de place : ___________________________________

Directives pédagogiques :

- Inscrivez votre nom, prénom, code permanent et le numéro de votre place.
- Lisez attentivement toutes les questions et **répondez directement sur le questionnaire**.
- Seule l'utilisation d'un crayon ou stylo est permise, **aucune documentation, calculatrice, téléphone cellulaire, ordinateur, ou autre objet permis**.
- Cet examen contient 7 questions pour 110 points au total (**10 points bonis**)
- Le barème est établi à environ 1 point par minute.
- Cet examen contient 30 pages, incluant 5 pages à la fin pour vos brouillons.
- Vous pouvez détacher les Appendices et les brouillons de l'examen.
- **Écrivez lisiblement et détaillez vos réponses**.
- Vous avez 110 minutes pour compléter cet examen.

BONNE CHANCE !

1	/ 20
2	/ 15
3	/ 15
4	/ 15
5	/ 15
6	/ 10
7	/ 20
Total	/ 100

1. (20) On a une séquence **S** de n éléments.

   a) (2) Un algorithme **A** exécute un calcul sur chaque élément de **S** en temps dans O(log n). Quel est le temps en pire cas de **A** ?

   $O(n \log n)$

   b) (3) Un algorithme **B** choisit log n éléments de **S** au hasard et exécute un calcul en temps dans O(n) pour chacun. Quel est le temps en pire cas de **B** ?

   $O(n \log n)$

c) (5) Un algorithme **C** exécute un calcul en temps dans O(n) pour chaque élément pair de **S** et un calcul en temps dans O(log n) pour chaque élément impair de **S**. Quels sont les temps d'exécution en meilleur et pire cas de **C** ?

Meilleur cas, tous les éléments sont impairs : O($n \log n$)

Pire cas, tous les éléments sont pairs : O(n^2)

d) (10) Un algorithme **D** appelle un algorithme **E** sur chaque élément **S**[i] de **S**. L'algorithme **E** exécute un calcul en temps dans O(i) lorsqu'il est appelé sur l'élément **S**[i]. Quel est le temps en pire cas de **D** ?

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2+n}{2} = O(n^2)$$

2. (15) Décrivez une fonction récursive, son temps d'exécution et l'espace utilisé pour :

   a) (5) Trouver l'élément maximum dans une séquence non ordonnée, **S**, de *n* éléments.

```
def max( S ):
    if( len( S ) == 0 ):
        return None
    else:
        max = S[0]
        return mymax( S, 1, max )

def mymax( S, j, max ):
    if( j == len( S ) ):
        return( max )
    else:
        if( S[j] > max ):
            max = S[j]
        return( mymax( S, j+1, max ) )
```

Temps d'exécution dans O(*n*)

Espace utilisé dans O(*n*)

b) (10) Calculer le *n*ième nombre harmonique, $H_n = \sum_{i=1}^{n} 1/i$.

```python
def harmonic( n ):
    if( n == 1 ):
        return n
    else:
        return 1/n + harmonic( n - 1 )
```

Temps d'exécution dans O(n)

Espace utilisé dans O(n)

3. (15) Une opération utile dans les bases de données est la *jointure naturelle*. On peut voir une base de données comme une liste ordonnée de paires d'objets. La jointure naturelle de deux bases de données, **A** et **B**, est la liste ordonnée de triplets (*x*,*y*,*z*) tel que la paire (*x*,*y*) est dans **A** et la paire (*y*,*z*) est dans **B**. Décrivez un algorithme efficace, ***jointureNaturelle***(**A**, **B**), pour calculer la jointure naturelle d'une liste **A** de *n* paires et d'une liste **B** de *m* paires et analysez son temps d'exécution.

Exemple :
jointureNaturelle(**A** = [(1,1),(2,3),(2,4),(3,1)], **B** = [(1,2),(4,1)]) = [(1,1,2),(2,4,1),(3,1,2)]

```python
def naturalJoin( A, B ):
    if( A == [] or B == [] ):
        return []
    #sort the elements in A using the 2nd value
    A.sort( key = lambda x: x[1] )

    result = []
    j = 0
    for i in range( len( A ) ):
        (w,x) = A[i]
        (y,z) = B[j]
        #get to the y == x tuples if any
        while( y < x and j < len( B ) - 1 ):
            j += 1
            (y,z) = B[j]

        firstj = j
        #either we have a match, then take 'em
        while( x == y and j < len( B ) ):
            result.append( (w,x,z) )
            j += 1
            if( j < len( B ) ):
                (y,z) = B[j]
            else:
                break

        #or not, then take the next element in A
        #and reposition j to the last one that matched
        j = firstj
    return result
```

Le tri coûte O(n log n)-temps. Par la suite, **dans le meilleur cas**, on peut former une seule paire pour chaque élément de A (càd que les valeurs de jointure sont uniques), et on a un parcours dans O(n)-temps. Globalement dans ce cas, le coût le plus élevé est le tri dans O(n log n)-temps, en utilisant un tri que serait dans O(n log n)-temps.

meilleur cas dans O(n log n)

Dans le pire cas, tous les tuples dans A peut être joints au premier tuple dans B, et dans ce cas on aurait un parcours dans O(nm).

pire cas dans O(nm)

4. (15) Supposez une pile **S** contenant *n* éléments et une file **Q** initialement vide. Décrivez comment utiliser **Q** pour chercher si **S** contient un certain élément *x*, ***chercherPile***(**S**, *x*). Votre algorithme doit retourner les éléments de **S** dans l'ordre original. Vous ne pouvez qu'utiliser **S** et **Q** et un nombre constant de variables additionnelles. Pour les opérations sur les piles et files, voir l'**Appendice A**.

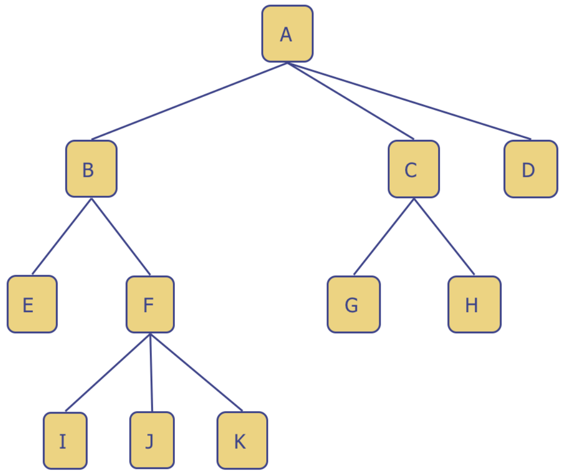
Exemples :
S = [1, 2, 3, 4](size = 4)[top = 3] ; ***chercherPile***(**S**, 2) = ([1, 2, 3, 4](size = 4)[top = 3], True)
S = [1, 2, 3, 4](size = 4)[top = 3] ; ***chercherPile***(**S**, 0) = ([1, 2, 3, 4](size = 4)[top = 3], False)

```python
from ListStack import ListStack
from ListQueue import ListQueue

def checkStack( S, x ):
    if( S.is_empty() ):
        return (str(S),False)

    Q = ListQueue()
    found = False

    while( not S.is_empty() ):
        y = S.pop()
        Q.enqueue( y )
        if( x == y):
            found = True

    #Fill Q back but in the wrong order
    while( not Q.is_empty() ):
        y = Q.dequeue()
        S.push( y )

    #Reverse the order
    while( not S.is_empty() ):
        y = S.pop()
        Q.enqueue( y )

    #restablish the original stack
    while( not Q.is_empty() ):
        y = Q.dequeue()
        S.push( y )

    return (str( S ),found)
```

5. (15) Un client veut faire une extension de *PositionalList*, *FavoritesList*, permettant de déplacer un élément en position *p* à la première position de la liste, tout en gardant les autres éléments inchangés. Augmentez la classe *PositionalList* (**Appendice B**) pour supporter une nouvelle méthode, ***move_to_front***(*p*), réalisant cette tâche en reliant le noeud existant (sans créer de nouveau noeud).

```
def move_to_front( self, p ):
    node = self._validate( p )
    #remove the node from current location
    node.prev.next = node.next
    node.next._prev = node.prev
    #move the node to the head
    self._head.next.prev = node
    node.next = self._head.next
    node.prev = self._head
    self._head.next = node
    return p
```

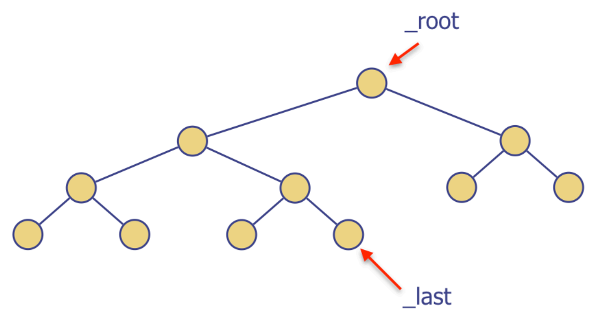
6. (10) Considérez la méthode de *parcours en largeur* d'un arbre et une variante de celle-ci, *funny* (**Appendice C**), qui utilise une pile plutôt qu'un file. Sachant que le *parcours en largeur* sur l'arbre ci-dessous visite les noeuds dans l'ordre suivant : A, B, C, D, E, F, G, H, I, J, K. Dites dans quel ordre les noeuds de cet arbre seront visités par le parcours *funny*.



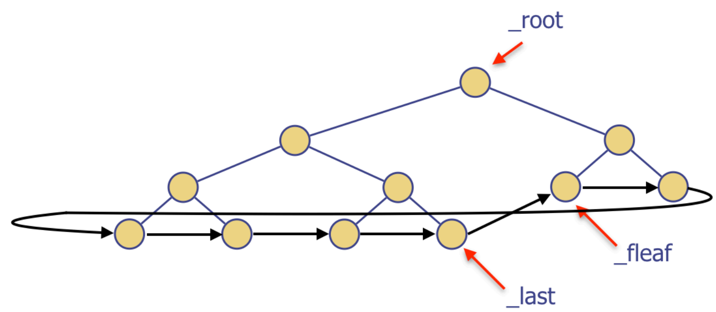
A, D, C, H, G, B, F, K, J, I, E

7. (20) Considérez l'implémentation *HeapTree* (**Appendice D**). Un *HeapTree* utilise deux références, *_root* et *_last* pour, respectivement pointer la racine d'un *HeapTree* et son dernier noeud et tel qu'indiqué ci-dessous en (**A**). On a ajouté une référence, *_fleaf*, pour pointer la première feuille d'un *HeapTree* et à chaque *_Node* une référence sur sa feuille à droite, *_rleaf*, de manière à pouvoir créer une liste circulaire des feuilles d'un HeapTree, tel qu'indiqué ci-dessous en (**B**). Écrivez la méthode *_link_leaves* de la classe *HeapTree* pour enchaîner dans une liste circulaire les feuilles d'un *HeapTree*.

A)



B)




```
def _link_leaves( self ):
    #on fait un parcours en largeur
    Q = ListQueue()
    prev = None
    found = False
    Q.enqueue( self._root )
    while not Q.is_empty():
        p = Q.dequeue()
        #on link la feuille précédente, prev, à p
        if( not prev == None ):
            prev._rleaf = p
            prev = p
        #on trouve la première feuille
        if( not found and p._is_leaf() ):
            self._fleaf = p
            found = True
            prev = p
        for c in p._children():
            Q.enqueue( c )
    self._last._rleaf = self._fleaf
```

Appendice A : Opérations de pile et file

Pile

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	"error"	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop( )	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop( )	8	[7, 9, 6]

File

Operation	Return Value	first $\leftarrow Q \leftarrow$ last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue( )	7	[9, 4]

Appendice B : PositionalList

```
from DoublyLinkedList import DoublyLinkedList

#ADT PositionalList "interface"
class PositionalList( DoublyLinkedList ):

    class Position:
        #Une abstaction de la position d'un élément

        def __init__( self, container, node ):
            #constructeur
            self._container = container
            self._node = node

        def element( self ):
            #retourne l'élément stocké à cette position
            return self._node.element

        def __eq__( self, other ):
            #retourne True si other est du même type et réfère à la même position
            return type( other ) is type( self ) and other._node is self._node

        def __ne__( self, other ):
            #retourne True si other ne représente pas la même position
            return not( self == other )
```

Appendice B : PositionalList (suite)

```
def _validate( self, p ):
    #retourne le noeud de la position, ou lance une exception si invalide
    if not isinstance( p, self.Position ):
        raise TypeError( "p must be proper Position type" )
    if p._container is not self:
        raise ValueError( "p does not belong to this container" )
    if p._node.next is None: #convention pour noeud désassigné
        raise ValueError( "p is no longer valid" )
    return p._node

#Utilitaires
def _make_position( self, node ):
    #retourne une instance de Position pour un noeud donné (ou None si sentinelle)
    if node is self._head or node is self._tail:
        return None
    else:
        return self.Position( self, node )

#Méthodes d'accès
def first( self ):
    return self._make_position( self._head.next )

def last( self ):
    return self._make_position( self._tail.prev )

def before( self, p ):
    node = self._validate( p )
    return self._make_position( node.prev )

def after( self, p ):
    node = self._validate( p )
    return self._make_position( node.next )

def __iter__( self ):
    #itérateur des éléments de la liste
    cursor = self.first()
    while cursor is not None:
        yield cursor.element()
        cursor = self.after( cursor )

#Méthodes de mutations
#override les méthodes héritées pour retourner des Position plutôt que des noeuds.
def insert( self, e ):
    node = super().insert( e )
    return self._make_position( node )

def append( self, e ):
    node = super().append( e )
    return self._make_position( node )

def replace( self, p, e ):
    #remplace l'élément p par e
    #retourne l'élément qui était à la position p
    original = self._validate( p )
    old_value = original.element
    original.element = e
    return old_value
```

Appendice B : DoublyLinkedList

```
from DoublyLinkedNode import DoublyLinkedNode
from List import List

class DoublyLinkedList( List ):

    #implements the ADT List (List.py)
    #uses the DoublyLinkedNode class (DoublyLinkedNode.py)

    def __init__( self ):
        self._head = DoublyLinkedNode( None, None, None )
        self._tail = DoublyLinkedNode( None, None, None )
        self._head.next = self._tail
        self._tail.prev = self._head
        self._size = 0

    def __len__( self ):
        return self._size

    def __str__( self ):
        if self.is_empty():
            return "[](size = 0)"
        else:
            pp = "["
            curr = self._head.next
            while curr.next != self._tail:
                pp += str( curr.element ) + ", "
                curr = curr.next
            pp += str( curr.element ) + "]"
            pp += "(size = " + str( self._size ) + ")"
        return pp

    def is_empty( self ):
        return self._size == 0
```

Appendice B : DoublyLinkedList (suite)

```
def append( self, element ):
    newNode = DoublyLinkedNode( element, self._tail.prev, self._tail )
    self._tail.prev.next = newNode
    self._tail.prev = newNode
    self._size += 1
    return newNode

def insert( self, element ):
    newNode = DoublyLinkedNode( element, self._head, self._head.next )
    self._head.next.prev = newNode
    self._head.next = newNode
    self._size += 1
    return newNode

def remove( self, k ):
    # lists start at index 0
    if not 0 <= k < self._size:
        raise IndexError( 'DoublyLinkedList: index out of bounds' )
    else:
        curr = self._head.next
        for i in range( k ):
            curr = curr.next
        curr.prev.next = curr.next
        curr.next.prev = curr.prev
        curr.next = None #convention pour un noeud désassigné
        self._size -= 1
        return curr.element

def find( self, element ):
    if self.is_empty():
        return None
    else:
        curr = self._head.next
        for i in range( self._size ):
            if curr.element == element:
                return i
            else:
                curr = curr.next
        return None

def last( self ):
    if self.is_empty():
        return None
    else:
        return self._tail.prev.element

def first( self ):
    if self.is_empty():
        return None
    else:
        return self._head.next.element
```

Appendice B : DoublyLinkedNode et List

```
class DoublyLinkedNode:

    def __init__( self, element, prev, next ):
        self.element = element
        self.prev = prev
        self.next = next



#ADT List "interface"
class List:

    def __init__( self ):
        pass

    #return the number of elements in List
    def __len__( self ):
        pass

    #convert a List into a string:
    # elements listed between brackets
    # separated by commas
    # size and capacity of the data structure
    # indicated when relevant
    def __str__( self ):
        pass

    #add element at the end of list
    def append( self, element ):
        pass

    #remove the kth element
    def remove( self, k ):
        pass

    #find and return the rank of
    #element if in list, False otherwise
    def find( self, element ):
        pass
```

Appendice C : Parcours en largeur et funny

```
#print the subtree rooted by position p
#using a breadth-first traversal
def breadth_first_print( self ):
    Q = ListQueue()
    Q.enqueue( self.root() )
    while not Q.is_empty():
        p = Q.dequeue()
        print( p )
        for c in self.children( p ):
            Q.enqueue( c )
```

```
#print the subtree rooted by position p
#using a funny traversal
def funny_print( self ):
    S = ListStack()
    S.push( self.root() )
    while not S.is_empty():
        p = S.pop()
        print( p )
        for c in self.children( p ):
            S.push( c )
```

Appendice D : HeapTree

```
from BinaryTree import BinaryTree

class HeapTree( BinaryTree ):

    #inner class _Node
    class _Node:
        #create a static structure for _Node using __slots__
        __slots__ = '_element', '_parent', '_left', '_right', '_rleaf'
        #adding a reference to the rigth leaf (for linking the leaves)
        def __init__( self, element,
                      parent = None,
                      left = None,
                      right = None,
                      rleaf = None ):
            self._element = element
            self._parent = parent
            self._left = left
            self._right = right
            self._rleaf = rleaf

    #HeapTree constructor
    def __init__( self ):
        #create an initially empty heap tree
        #adding a reference to the first leaf of the Heap (fleaf)
        self._root = None
        self._last = None
        self._fleaf = None
        self._size = 0

    #get the size
    def __len__( self ):
        return self._size

    #get the root
    def _root( self ):
        return self._root
```

Appendice D : BinaryTree

```python
from Tree import Tree

class BinaryTree( Tree ):

    #get the left child of a position
    def left( self, p ):
        pass

    #get the right child of a position
    def right( self, p ):
        pass

    #get the sibling of a position
    def sibling( self, p ):
        #return the sibling Position
        parent = self.parent( p )
        if parent is None:
            return None
        else:
            if p == self.left( parent ):
                return self.right( parent )
            else:
                return self.left( parent )

    #get the children as a generator
    def children( self, p ):
        if self.left( p ) is not None:
            yield self.left( p )
        if self.right( p ) is not None:
            yield self.right( p )

    #print the subtree rooted by position p
    #using an inorder traversal
    def inorder_print( self, p ):
        if self.left( p ) is not None:
            self.inorder_print( self.left( p ) )
        print( p )
        if self.right( p ) is not None:
            self.inorder_print( self.right( p ) )
```

Appendice D : Tree

```
from ListQueue import ListQueue

#ADT Tree "interface"
class Tree:

    #inner class position
    class Position:

        def element( self ):
            pass

        def __eq__( self, other ):
            pass

        def __ne__( self, other):
            return not( self == other )

    #get the root
    def root( self ):
        pass

    #get the parent
    def parent( self, p ):
        pass

    #get the number of children
    def num_children( self, p ):
        pass

    #get the children
    def children( self, p ):
        pass

    #get the number of nodes
    def __len__( self ):
        pass

    #ask if a position is the root
    def is_root( self, p ):
        return self.root() == p

    #ask if a position is a leaf
    def is_leaf( self, p ):
        return self.num_children( p ) == 0

    #ask if the tree is empty
    def is_empty( self ):
        return len( self ) == 0

    #get the depth of a position
    def depth( self, p ):
        #returns the number of ancestors of p
        if self.is_root( p ):
            return 0
        else:
            return 1 + self.depth( self.parent() )

    #get the height of a position by descending the tree (efficient)
    def height( self, p ):
        #returns the height of the subtree at Position p
        if self.is_leaf( p ):
            return 0
        else:
            return 1 + max( self.height( c ) for c in self.children( p ) )
```

Appendice D : Tree (suite)

```
#print the subtree rooted by position p
#using a preorder traversal
def preorder_print( self, p, indent = "" ):
    print( indent + str( p ) )
    for c in self.children( p ):
        self.preorder_print( c, indent + "    " )

#print the subtree rooted by position p
#using a postorder traversal
def postorder_print( self, p ):
    for c in self.children( p ):
        self.postorder_print( c )
    print( p )

#print the subtree rooted by position p
#using a breadth-first traversal
def breadth_first_print( self ):
    Q = ListQueue()
    Q.enqueue( self.root() )
    while not Q.is_empty():
        p = Q.dequeue()
        print( p )
        for c in self.children( p ):
            Q.enqueue( c )
```

Brouilon 1

Brouilon 2

Brouilon 3

Brouilon 4

Brouilon 5