Name:____

Permanent code: __________________________________

Place number: __________________________________

Directives:

- Write your name, first name, permanent code and place number above.
- Read all questions carefully and **answer on the questionnaire**.
- Use only a pen or a pencil; **documentation, calculator, phone, computer, or the use of any other object is forbidden**.
- This exam contains 7 questions for 110 points in total (**10 bonus points**)
- The evaluation scale was established to about 1 point per minute.
- This exam contains 30 pages, including 5 pages at the end for drafting.
- You can remove the Appendix and draft pages from the exam.
- **Write visibly and detail your answers**.
- You have 110 minutes to complete this exam.

GOOD LUCK!

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

1. (20) We have a sequence **S** of n elements.

   a) (2) Algorithm **A** runs a process on each element of **S** in O(log n)-time. What is the worst-case running time of **A**?

   b) (3) Algorithm **B** chooses log n elements of **S** at random and runs a process on each selected element in O(n)-time. What is the worst-case running time of **B**?

c) (5) Algorithm **C** runs a process in O(n)-time for each even element in **S** and a process in O(log n)-time for each odd element in **S**. What are the **C** runtimes in best- and worst -case?

d) (10) Algorithm **D** calls algorithm **E** on each element **S**[i] of **S**. **E** runs a process in O(i)-time when called on element **S**[i]. What is the worst-case running time of **D**?

2. (15) Describe a recursive function, its running time and the space used to:

   a) (5) Find the maximum element in an unordered sequence, **S**, of *n* elements.

```
def max( S ):
```

b) (10) Compute the nth harmonic number, $H_n = \sum_{i=1}^{n} 1/i$.

```
def harmonic( i ):
```

3. (15) A useful operation in databases is the *natural join*. We can see a database as an ordered list of pairs of objects. The nature join of two databases, **A** and **B**, is the ordered list of triplets (*x,y,z*) so that the pair (*x,y*) is in **A** and the pair (*y,z*) is in **B**. Describe an efficient algorithm, ***NaturalJoin***(**A**, **B**), to compute the natural join of a list **A** of *n* pairs and list **B** of *m* pairs and analyze its runtime.

   Example :
   naturalJoin(**A** = [(1,1),(2,3),(2,4),(3,1)], **B** = [(1,2),(4,1)]) = [(1,1,2),(2,4,1),(1,3,1)]

```
def naturalJoin( A, B ):
```

4. (15) Suppose a stack **S** containing *n* elements and a queue **Q** initially empty. Describe how to use **Q** to search if **S** contains a given element *x*, ***findStack***(**S**, *x*). Your algorithm must return the elements in **S** in the original order. You must use **S** and **Q** and a constant number of additional variables. For the stack and queue operations, see **Appendix A**.

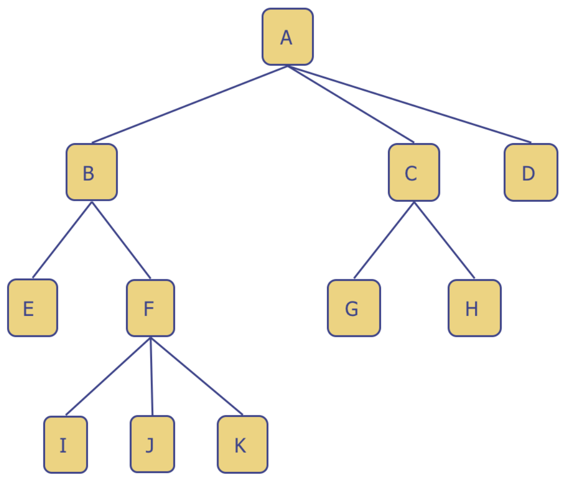
Examples :
S = [1, 2, 3, 4](size = 4)[top = 3] ; ***findStack***(**S**, 2) = ([1, 2, 3, 4](size = 4)[top = 3], True)
S = [1, 2, 3, 4](size = 4)[top = 3] ; ***findStack***(**S**, 0) = ([1, 2, 3, 4](size = 4)[top = 3], False)

```
def findStack( S, x ):
```

5. (15) A client wants to extend the *PositionalList*, *FavoritesList*, allowing to move an element in position *p* to the first position of the list, while keeping all other elements unchanged. Increase the class *PositionalList* (**Appendix B**) to support the new method, ***move_to_front***(*p*), which realize this task by linking the existing node (without creating a new node).

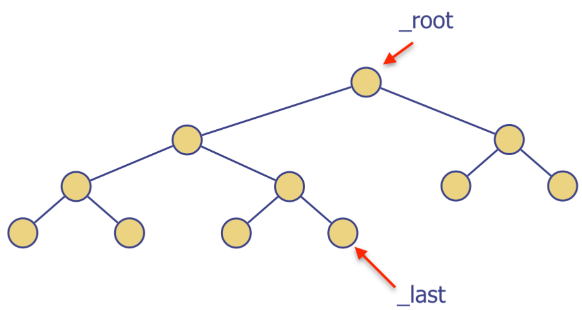
```
def move_to_front( p ):
```

6. (10) Consider the method to breadth-visit the nodes of a tree and a variant of it, *funny* (**Appendix C**), which uses a stack rather than a queue. Knowing that the breadth-first traversal applied to the tree below visits the nodes in the following order: A, B, C, D, E, F, G, H, I, J, K. Say in which order the nodes of this tree will be visited by the *funny* traversal.

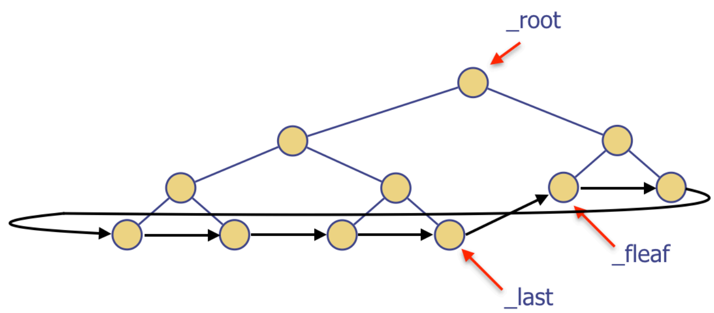


7. (20) Consider the implementation of *HeapTree* (**Appendix D**). A *HeapTree* uses two references, *_root* and *_last* to point the root of a *HeapTree* and its last node, respectively, and as illustrated below (**A**). We added a reference, *_fleaf*, to point to the first leaf of *HeapTree* and in each *_Node* a reference to the leaf on its right, *_rleaf*, so that we can create a circular list of the leaves of a *HeapTree*, as illustrated below (**B**). Write the method *_link_leaves* of the class *HeapTree* to link in a circular list the leaves of a *HeapTree*.

A)



B)




```
def _link_leaves( self ):
```

Appendix A: Stack and queue operations

Stack

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]

Queue

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]

Appendix 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 )
```

Appendix B: PositionalList (cont'd)

```
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
```

Appendix 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 (cont'd)

```
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
```

Appendix B: DoublyLinkedNode and 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
```

Appendix C: Breadth-first and funny traversals

```
#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 )
```

Appendix 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
```

Appendix 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 ) )
```

Appendix 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 ) )
```

Appendix D: Tree (cont'd)

```
#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 )
```

Draft 1

Draft 2

Draft 3

Draft 4

Draft 5