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 9 questions for 100 points in total.
- The evaluation scale was established to about 1 point per minute.
- This exam contains 20 pages, including 2 pages at the end for drafting.
- **Write visibly and detail your answers**.
- You have 100 minutes to complete this exam.

GOOD LUCK!

1	/ 15
2	/ 10
3	/ 10
4	/ 10
5	/ 10
6	/ 15
7	/ 10
8	/ 10
9	/10
Total	/ 100

1. (15) You are given a sequence, `S`, of *n* distinct integer in increasing order and a number *k*.

   a) (10) Describe a recursive algorithm to find two integers of `S` which sum gives *k*, if such a pair exist.

   b) (5) What is the running time of your algorithm?

2. (10) The number of operations executed by algorithms A and B are respectively of $42n^2$ and $3n^3$. Determine n_0 such that A is better than B for all $n \geq n_0$.

3. (10) Describe a recursive algorithm to count the number of nodes in a singly linked list, `L`. Assume that the variable `head` is a reference on the first node of the list.

4. (10) Describe an algorithm that uses only the operations of the `BinaryTree` class (see Appendix A) to count the leaves of a binary tree that are a left child of their respective parents.

5. (10) When using a linked implementation of a heap (see Appendix B), an alternative method to find the last node during an insertion is to store in the last node and every leave a reference to the leaf immediately to its right (or on the first node of the next level for the rightmost node). Show how to maintain updated these references in O(*1*) in time for the following operations:

    a) (5) `remove_min`.

b) (5) `add`.

6. (15) The `min` method of the class `UnsortedPriorityQueue` (see Appendix B) runs in O(n).

   a) (5) Suggest a modification so that `min` runs in O(1).

   b) (5) Explain the changes that are necessary in the other methods of the class.

c) (5) Can you adapt your solution so that `remove_min` runs in O(*1*) ? Explain your answer.

7. (10) Draw an example of a min-heap with the odd numbers between 1 and 59 (without repetition) so that the insertion of the key 32 makes it move up until a child of the root.

8. (10) Give a non-recursive version of the `swim` procedure for the `ArrayHeapPriorityQueue` class (see Appendix C).

9. (10) Build a min-heap in O(n) operations for the following values: 11, 19, 1, 28, 13, 12, 15, 5, 8, 21, 6, 7, 23, 16, 4, and 14.

Appendix A: Tree and BinaryTree

```
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

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

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

    #the tree is empty?
    def is_empty( self ):
        return len( self ) == 0
```

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

    #get the height of position p
    def height( self, p ):
        if p is None:
            p = self.root()
        if self.is_leaf( p ):
            return 0
        else:
            return 1 + max( self.height(c) for c in self.children(p))
```

```
from Tree import Tree

class BinaryTree( Tree ):

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

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

    #get the sibling of position p
    def sibling( self, p ):
        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 of position p 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 )
```

Appendix B: PriorityQueue and UnsortedPriorityQueue

```
class PriorityQueue:

    #Nested class for the items
    class _Item:
        #efficient composite to store items
        __slots__ = '_key', '_value'

        def __init__( self, k, v ):
            self._key = k
            self._value = v

        def __lt__( self, other ):
            return self._key < other._key

        def __gt__( self, other ):
            return self._key > other._key

    def __init__( self ):
        pass

    #get the number of elements in queue
    def __len__( self ):
        pass

    #queue is empty?
    def is_empty( self ):
        return len( self ) == 0

    #next element
    def min( self ):
        pass

    #add element to queue
    def add( self, k, x ):
        pass

    #remove the next element
    def remove_min( self ):
        pass
```

```
from PriorityQueue import PriorityQueue

class UnsortedPriorityQueue( PriorityQueue ):

    def __init__( self ):
        self._Q = []

    def __len__( self ):
        return len( self._Q )

    def __getitem__( self, i ):
        return self._Q[i]

    def is_empty( self ):
        return len( self ) == 0

    def min( self ):
        if self.is_empty():
            return None
        #search the min in O(n) on average
        the_min = self._Q[0]
        for item in self:
            if item < the_min:
                the_min = item
        return the_min

    def add( self, k, x ):
        #in O(1)
        self._Q.append( self._Item( k, x ) )

    def remove_min( self ):
        if self.is_empty():
            return None
        #search the index of min in O(n) on average
        index_min = 0
        for i in range( 1, len( self ) ):
            if self._Q[i] < self._Q[index_min]:
                index_min = i
        the_min = self._Q[index_min]
        #delete the min
        del self._Q[index_min]
        #return the deleted item
        return the_min
```

Appendix C: ArrayHeapPriorityQueue

```
from PriorityQueue import PriorityQueue

class ArrayHeapPriorityQueue( PriorityQueue ):

    def __init__( self ):
        self._Q = []

    def __len__( self ):
        return len( self._Q )

    def __getitem__( self, i ):
        return self._Q[i]

    def is_empty( self ):
        return len( self ) == 0

    def _parent( self, j ):
        return (j-1) // 2

    def _left( self, j ):
        return 2*j + 1

    def _right( self, j ):
        return 2*j + 2

    def _has_left( self, j ):
        return self._left( j ) < len( self )

    def _has_right( self, j ):
        return self._right( j ) < len( self )

    def min( self ):
        if self.is_empty():
            return None
        #min is in the root
        return self._Q[0]

    def _swap( self, i, j ):
        tmp = self._Q[i]
        self._Q[i] = self._Q[j]
        self._Q[j] = tmp
```

```
    def _swim( self, j ):
        parent = self._parent( j )
        if j > 0 and self._Q[j] < self._Q[parent]:
            self._swap( j, parent )
            self._swim( parent )

    def _sink( self, j ):
        if self._has_left( j ):
            left = self._left( j )
            small_child = left
            if self._has_right( j ):
                right = self._right( j )
                if self._Q[right] < self._Q[left]:
                    small_child = right
            if self._Q[small_child] < self._Q[j]:
                self._swap( j, small_child )
                self._sink( small_child )

    def add( self, k, x ):
        #in O(log n)
        item = self._Item( k, x )
        self._Q.append( item )
        #swim the new item in O(log n)
        self._swim( len(self)-1 )
        #return the new item
        return item

    def remove_min( self ):
        if self.is_empty():
            return None
        #min is at the root
        the_min = self._Q[0]
        #move the last item to the root
        self._Q[0] = self._Q[len(self)-1]
        #delete the last item
        del self._Q[len(self)-1]
        if self.is_empty():
            return the_min
        #sink the new root in O(log n)
        self._sink( 0 )
        #return the min
        return the_min
```

Draft 1

Draft 2