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 9 questions pour 100 points au total.
- Le barème est établi à environ 1 point par minute.
- Cet examen contient 20 pages, incluant 2 pages à la fin pour vos brouillons.
- **Écrivez lisiblement et détaillez vos réponses**.
- Vous avez 100 minutes pour compléter cet examen.

BONNE CHANCE !

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

1. (15) On vous donne une séquence, `S`, de n entiers distincts en ordre croissant et un nombre k.

   a) (10) Décrivez un algorithme récursif pour trouver 2 entiers de `S` dont la somme donne k, si une telle paire existe.

```
def paire(S, k, i, j):
        if i > len(S) - 2:
                return None
        elif j > len(S) - 1
                return paire(S, k, i+1, i+2)
        elif S[i] + S[j] == k:
                return (S[i], S[j])
        else:
                return paire(S, k, i, j+1)


def trouver_paire(S):
        return paire(S, k, 0, 1)
```

b) (5) Quel est le temps d'exécution de votre algorithme ?

$O(n^2)$

2. (10) Les nombres d'opérations effectuées par l'algorithme A et B sont respectivement de $42n^2$ et $3n^3$. Déterminez n_0 tel que A est meilleur que B pour tout $n \geq n_0$.

$$42n^2 < 3n^3$$

$$42 < 3n$$

$$14 < n$$

vrai pour tout $n \geq n_0 = 15$

3. (10) Décrivez un algorithme récursif pour compter le nombre de noeuds dans une liste simplement chaînée, `L`. Assumez que la variable `head` est une référence sur le premier noeud de la liste.

```
def nb_noeuds(noeud):
	if noeud:
		return 1 + nb_noeuds(noeud.next)
	else:
		return 0


def compter_nb_noeuds(L):
	return nb_noeuds(L.head)
```

4. (10) Décrivez un algorithme qui n'utilise que les opérations de la classe `BinaryTree` (voir Appendice A) pour compter les feuilles d'un arbre binaire qui sont un enfant gauche de leur parent respectif.

```
def nb_feuilles_gauches(self, noeud):
    compte = 0
    for child in self.children(noeud):
        if self.is_leaf(child) and child == self.left(noeud):
            compte += 1
        compte += self.nb_feuilles_gauches(child)
    return compte


def compter_nb_feuilles_gauches(self):
    return self.nb_feuilles_gauches(self.root())
```

5. (10) Lorsqu'on utilise une implantation chaînée pour un monceau (voir Appendice B), une méthode alternative pour trouver le dernier noeud lors d'une insertion est de stocker dans le dernier noeud et dans chaque feuille une référence à la feuille immédiatement à sa droite (ou sur le premier noeud du niveau suivant pour le noeud le plus à droite). Montrez comment maintenir ces références à jour en O(*1*) en temps pour les opérations:

   a) (5) `remove_min`.

b) (5) `add`.

6. (15) La méthode `min` de la class `UnsortedPriorityQueue` (voir Appendice B) exécute en O(*n*).

   a) (5) Suggérez une modification simple pour que `min` exécute en O(*1*).

Il suffit de garder un pointeur sur le minimum.

   b) (5) Expliquez les changements nécessaires à effectuer dans les autres méthodes de la classe.

Dans la fonction add, il faut vérifier si le nouvel élément ajouté est plus petit que le min, et si oui mettre à jour le pointeur min sur ce nouvel élément.

Dans fonction remove_min, le travail de chercher le min va maintenant se faire après la suppression, afin de mettre à jour le pointeur min.

c) (5) Pouvez-vous adapter votre solution pour que `remove_min` exécute en O(*1*) ? Expliquez votre réponse.

Non. D'une façon ou d'une autre il faudra toujours chercher le nouveau minimum, et il faut pour cela parcourir les n éléments, puisqu'ils ne sont pas triés.

7. (10) Dessinez un exemple de monceau-min dont les clés sont les nombres impairs de 1 à 59 (sans répétition) et tel que l'insertion de la clé 32 la fait remonter jusqu'à un enfant de la racine.

8. (10) Donnez une version non récursive de la procédure `swim` pour la classe `ArrayHeapPriorityQueue` (voir Appendice C).

9. (10) Construisez un monceau-min en O(n) opérations pour les valeurs suivantes : 11, 19, 1, 28, 13, 12, 15, 5, 8, 21, 6, 7, 23, 16, 4, et 14.

Appendice 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 )
```

Appendice B : PriorityQueue and UnsortedPriorityQueue

```python
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
```

Appendice 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
```

Brouilon 1

Brouilon 2

6. (15) La méthode `min` de la class `UnsortedPriorityQueue` (voir Appendice B) exécute en O(*n*).

   a) (5) Suggérez une modification simple pour que `min` exécute en O(*1*).

```
def min( self ):
    if self.is_empty():
        return None
    #return the min in O(1)
    the_min = self._Q[self._min_index]
    return the_min
```

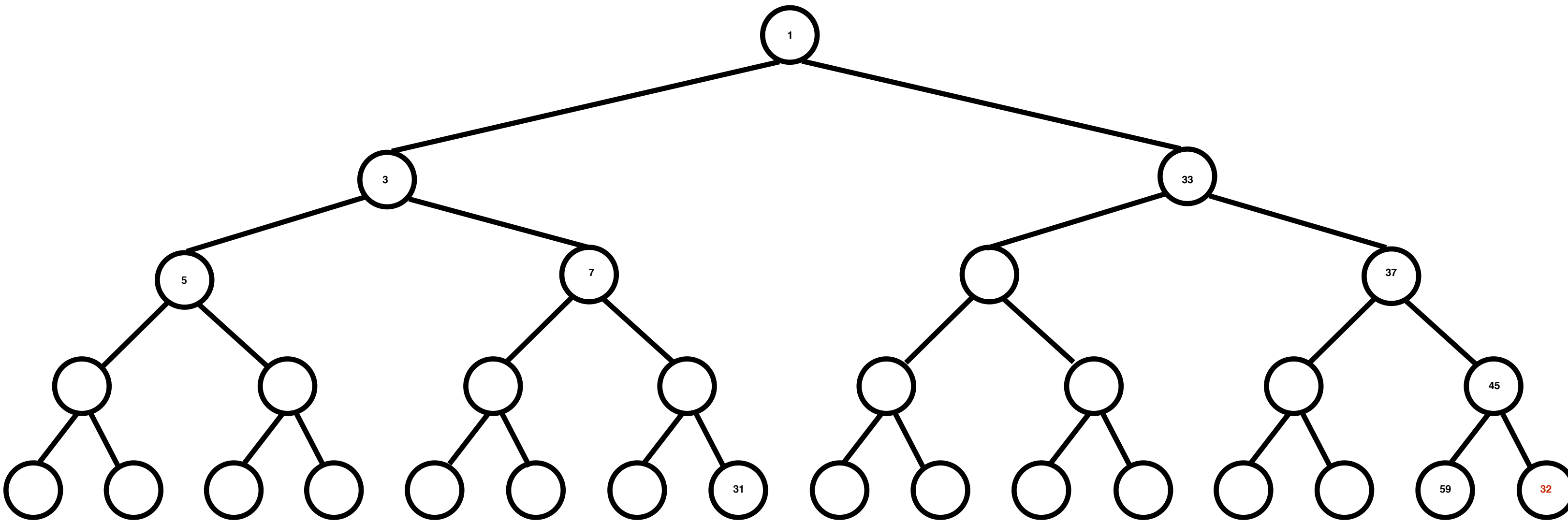
   b) (5) Expliquez les changements nécessaires à effectuer dans les autres méthodes de la classe.

```
def __init__( self ):
    self._Q = []
    self._min_index = None

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

    # À ajouter dans la nouvelle version
    # Mettre à jour min_index si nouvelle insertion
    # possède une clé < plus petite clé avant l'insertion
    if k < self._Q[self._min_index]:
        self._min_index = len( self._Q ) - 1

def remove_min( self ):
    if self is empty():
        return None
    #save the min
    the_min = self._Q[self._min_index]
    #delete the min
    del self._Q[self._min_index]
    #search the new min index in O(n) on average
    self._min_index = 0
    for i in range( 1, len( self ) ):
        if self._Q[i] < self._Q[self._min_index]:
            self._min_index = i

    #return the deleted item
    return the_min
```

c) (5) Pouvez-vous adapter votre solution pour que `remove_min` exécute en O(*1*) ? Expliquez votre réponse.

Non, il n'est pas possible de trouver le prochain min en *O*(1) ;
il faut parcourir tous les éléments de la file pour le trouver, ce qui se fait en *O*(n).

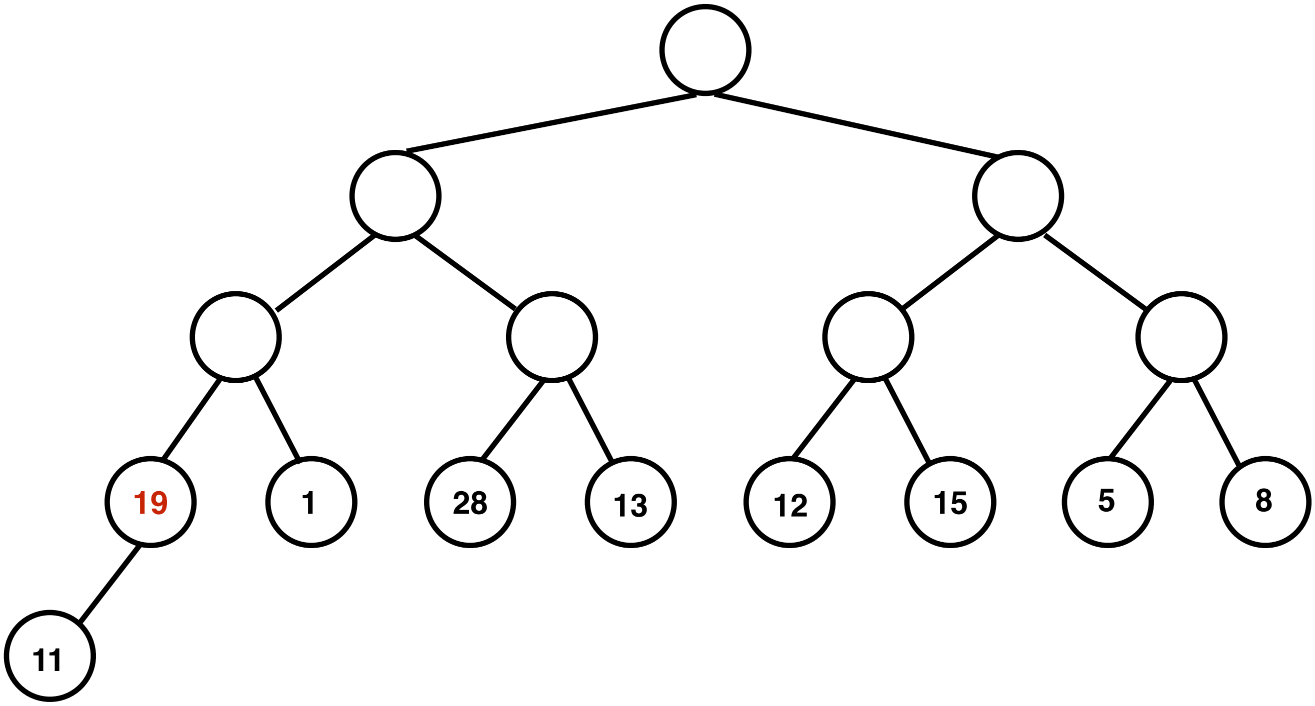
7. (10) Dessinez un exemple de monceau-min dont les clés sont les nombres impairs de 1 à 59 (sans répétition) et tel que l'insertion de la clé 32 la fait remonter jusqu'à un enfant de la racine.

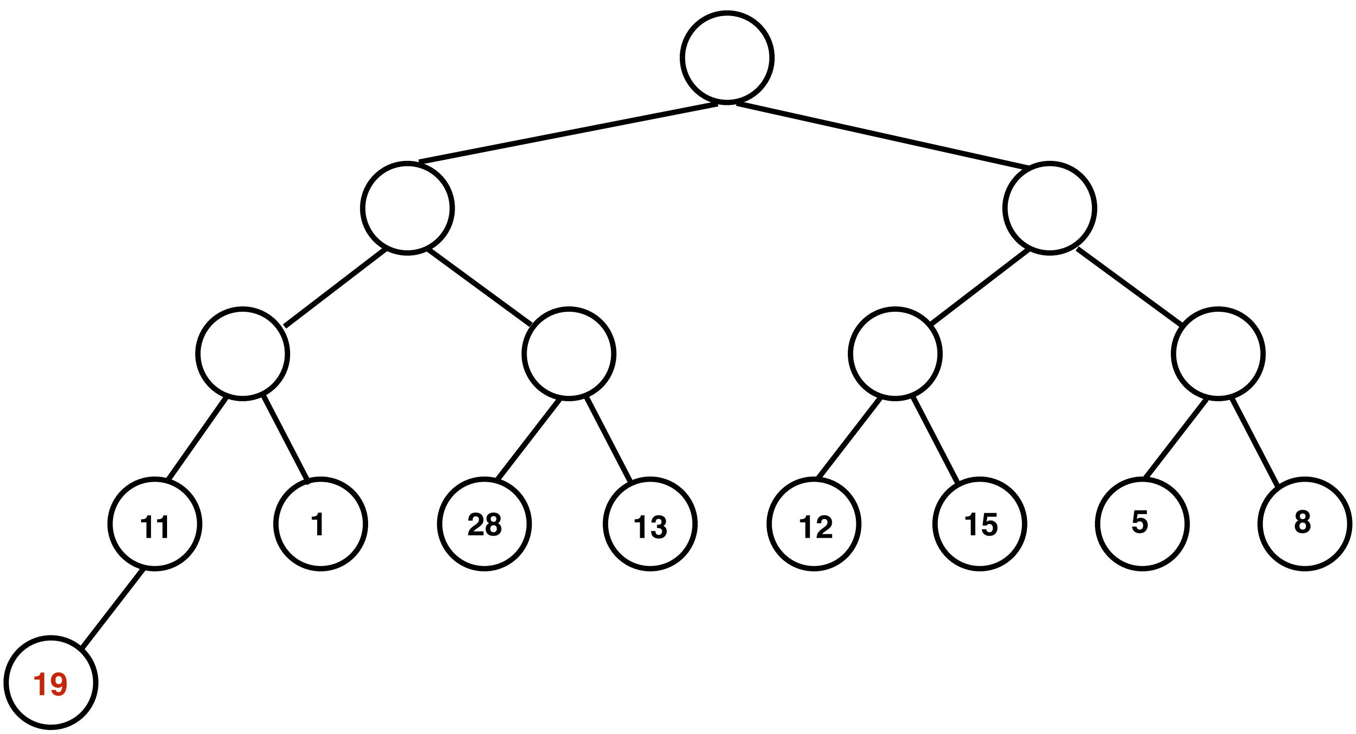


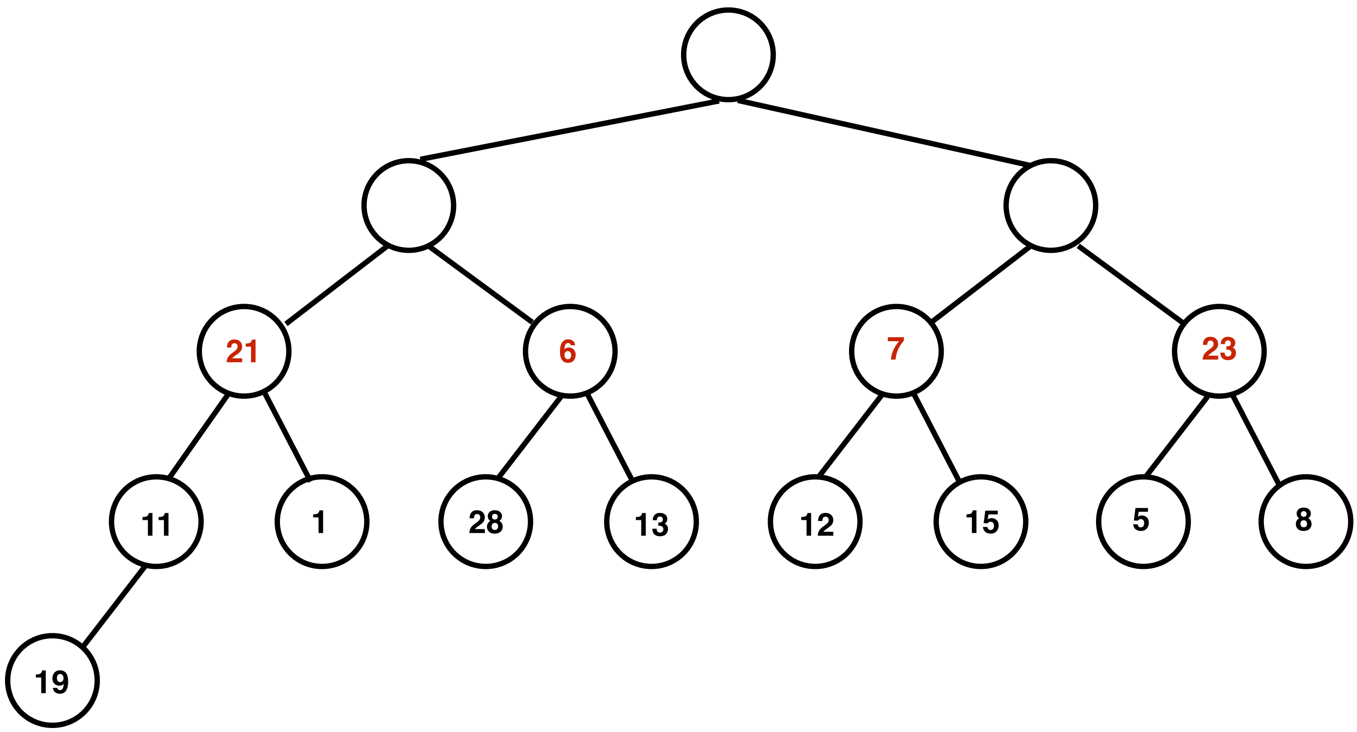
8. (10) Donnez une version non récursive de la procédure `swim` pour la classe `ArrayHeapPriorityQueue` (voir Appendice C).

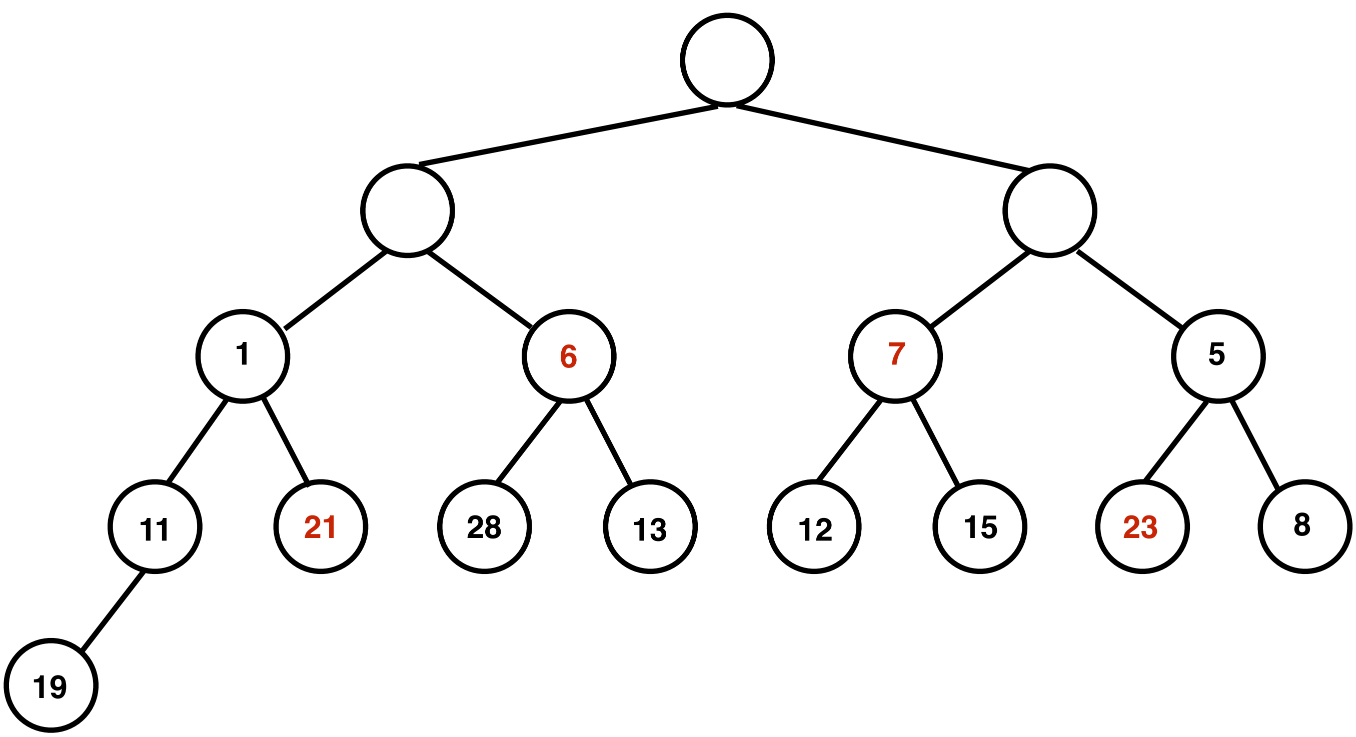
```
def _swim( self, j ):
    while j > 0:
        parent = self._parent( j )
        if self._Q[j] < self._Q[parent]:
            self._swap( j, parent )
            j = parent
        j = 0
```

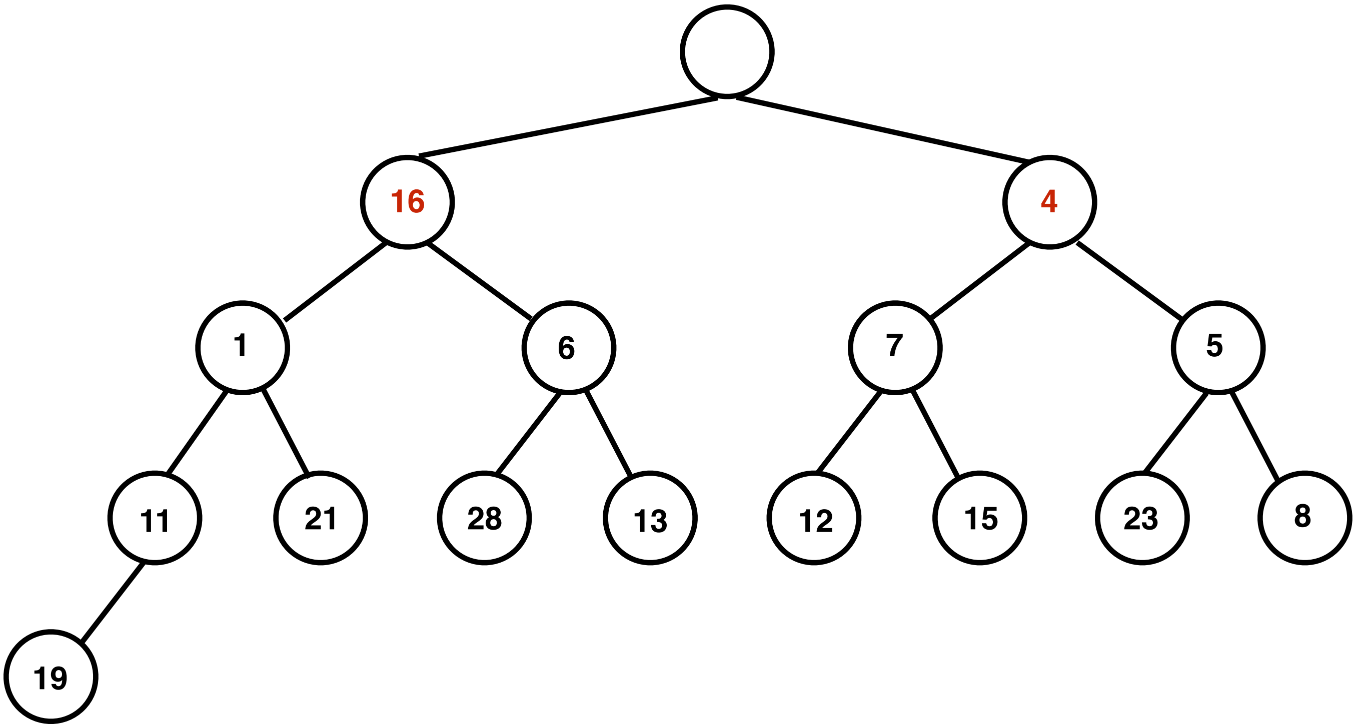
9. (10) Construisez un monceau-min en O(n) opérations pour les valeurs suivantes : 11, 19, 1, 28, 13, 12, 15, 5, 8, 21, 6, 7, 23, 16, 4, et 14.

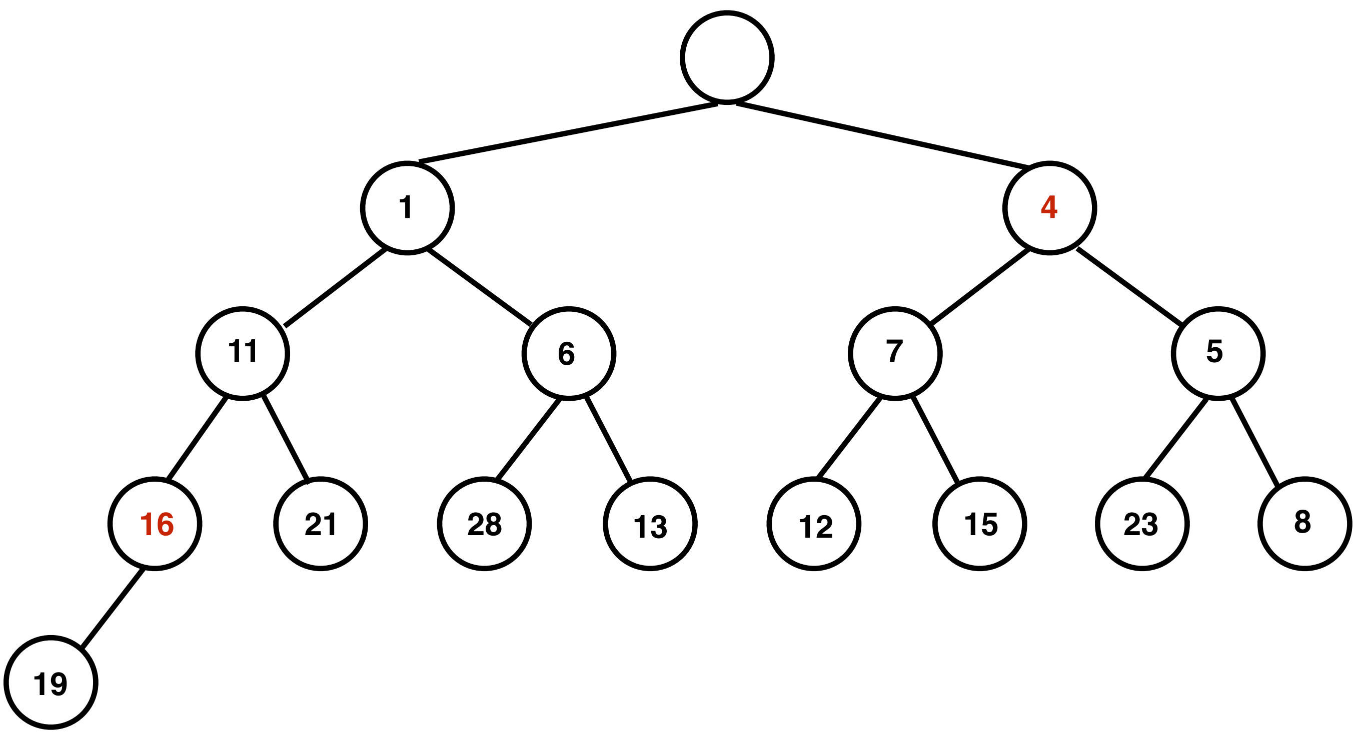


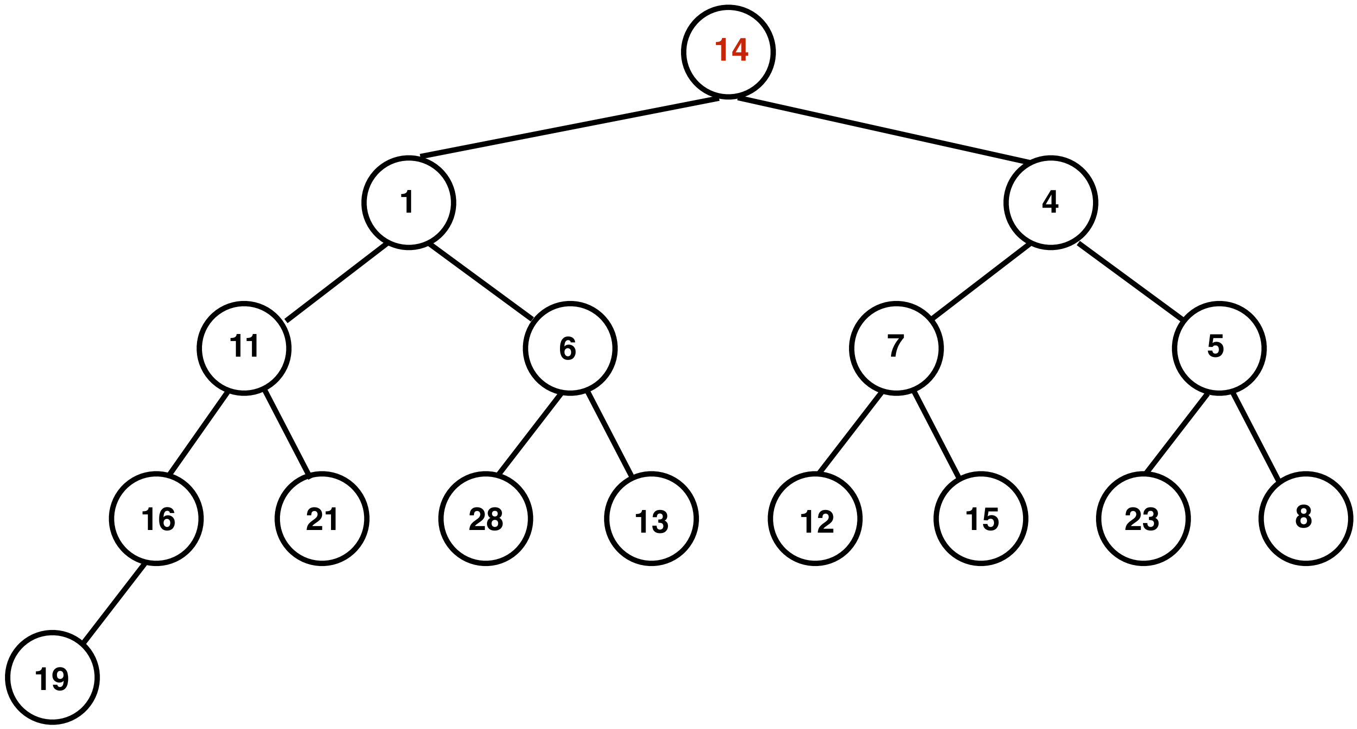


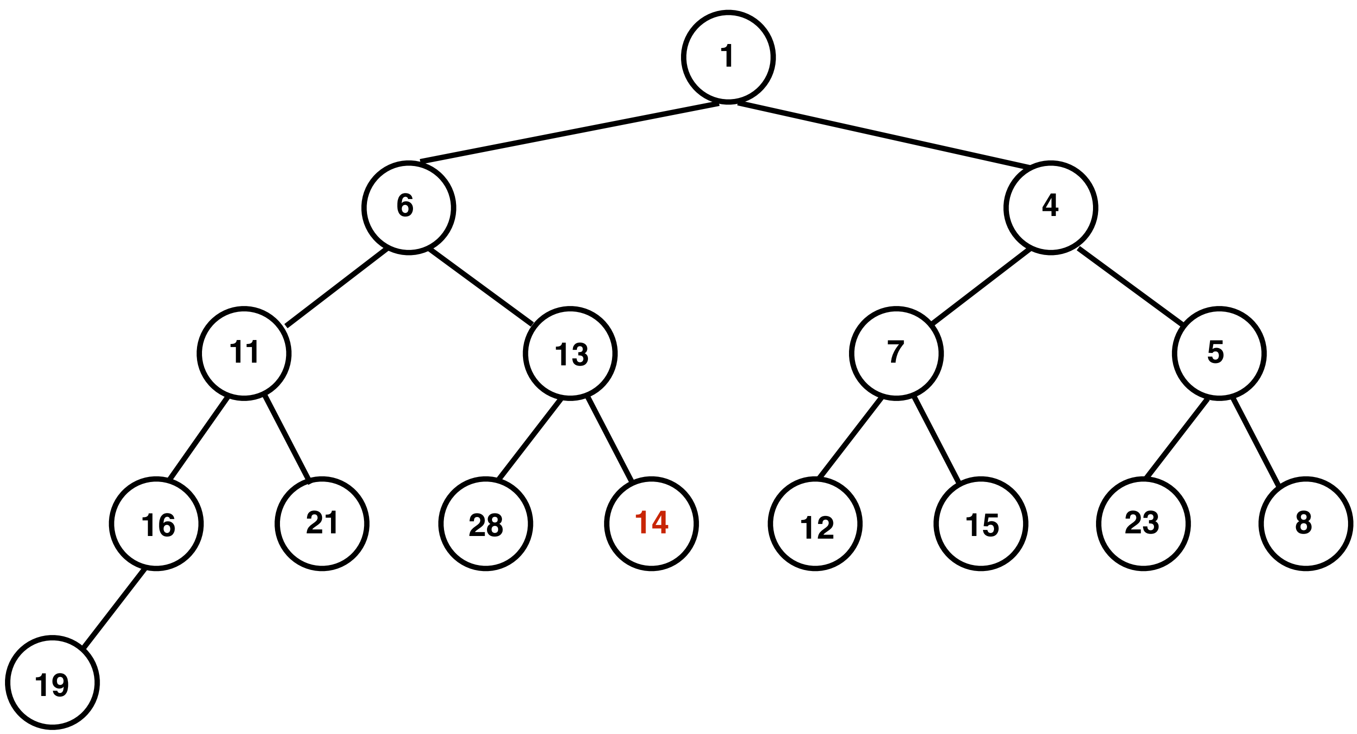

21
6
7
23
11
1
28
13
12
15
5
8
19


1
6
7
5
11
21
28
13
12
15
23
8
19


16
4
1
6
7
5
11
21
28
13
12
15
23
8
19


1
4
11
6
7
5
16
21
28
13
12
15
23
8
19


14
1
4
11
6
7
5
16
21
28
13
12
15
23
8
19


1
6
4
11
13
7
5
16
21
28
14
12
15
23
8
19

Appendice 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 )
```

Appendice 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
```

Appendice C : ArrayHeapPriorityQueue

```python
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
```