Name :____

Permanent code : ______________________________________

Directives:

- Write your name and permanent code above
- Read all questions and **answer directly on the sheets**
- Only a pen/pencil is permitted, **no documentation**, calculator, phone, computer, or any other object
- This exam has 9 questions for 120 points
- This exam has 19 pages
- For the developing questions, **write clearly** and **detail your answers**
- You have 100 minutes to complete this exam

GOOD LUCK!

1	/ 10
2	/ 10
3	/ 10
4	/ 25
5	/ 15
6	/ 12
7	/ 14
8	/ 12
9	/ 12
Total	/120

Q1. (10) Cross the box if the statement is true.

a)	Adding information to the data encoding of a problem saves computing time.	
b)	Changing computer does not change execution time by more than a constant factor.	
c)	Taking empirical measures of execution times always reveal surprises.	
d)	$O(n!)$ is in $O(2^n)$, but $O(2^n)$ is not in $O(n!)$.	
e)	We can always determine the performance behavior of an algorithm by looking at the code of the program in which it is implemented.	
f)	A recursive implementation of an algorithm always outperforms the non-recursive version.	
g)	Inserting a new element in an array of n elements takes on average $O(n/2) = O(n)$ operations.	
h)	Deleting an element in an array of n elements can be done on average in $O(n^2/4) = O(n)$ operations.	
i)	Deleting the last element of a single-linked list implemented using pointers on the first and last elements can be done in $O(1)$ operation.	
j)	Being in $O(\log n)$ in the worst case for a collection of n elements, the binary search is a better choice than a binary search tree when we expect to do more searches than any other operation.	

Q2. (10) Show that it is impossible to sort a data collection using binary comparisons in less than O(n log n) comparisons.

Q3. (10) Draw the recursive calls associated to quicksort when applied to the following collection if we always choose the middle element as the pivot, i.e. at index $\lfloor(\text{max}-\text{min}+1)/2\rfloor$, where min and max are the lower and upper bounds of the region of the array to be sorted.

15	9	8	1	$\boxed{4}$	11	7	12	3

index pivot = $\lfloor 9/2 \rfloor = 4$

Q4. Consider the following collection:

15	9	8	1	4	11	7	12	3

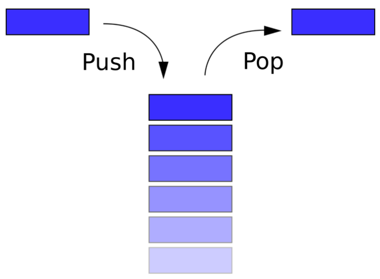
a) (10) Draw the heap resulting from applying the following "buildHeap" procedure:

```
buildHeap (A)
1.   for i = ⌊n/2⌋ − 1 downto 0 do
2.     heapify (A, i, n)
end

heapify (A, idx, max)
1.   left = 2*idx + 1
2.   right = 2*idx + 2
3.   if (left < max and A[left] > A[idx]) then
4.     largest = left
5.   else largest = idx
6.   if (right < max and A[right] > A[largest]) then
7.     largest = right
8.   if (largest ≠ idx) then
9.     swap A[i] and A[largest]
10.    heapify (A, largest, max)
end
```

b) (5) Draw the corresponding heap in an array.

c) (10) Explain in your words quicksort's worst case and why heapsort cannot fall in this case and stays in O(n log n) in worst case.

Q5. Consider a stack:



a) (5) Propose a data structure based on an array that would allow for "push" and "pop" operations in O(1) when there is enough room in the array to store all elements. Describe your data structure by completing the code of the ArrayStack **class** (PS. pseudo-code ok). See the Appendix for the **Stack** interface.

```
public class ArrayStack implements Stack {
    // Implantation of a stack using an array.
    // top points the element on top of the stack.
```

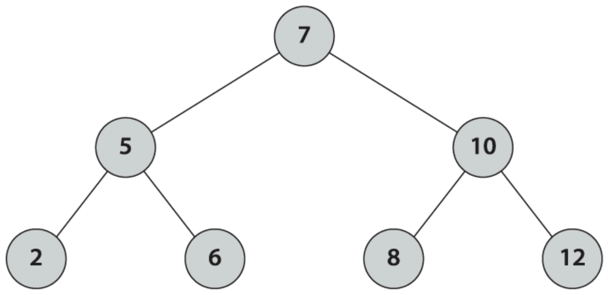
```
}
```

b) (5) Implement the "push", "pop" and "top" operations of your stack (PS. pseudo-code ok).

```
public void push( Object x ) {
  //add an object on the top of the stack.








}


 public Object pop() {
   //return and remove the top of the stack.








 }

 public Object top() {
   //return the object on the top of the stack.








 }
```

c) (5) What would be the complexity of the "push" operation when there is no room to insert a new element in the stack?

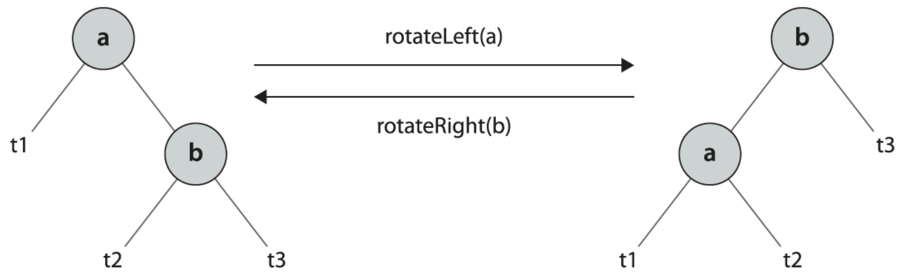
Q6. Consider the following binary search tree:



a) (8) Draw the trees resulting from each following insertions: 9, 4, 11 et 13 (in this order).

b) (4) Use the last tree obtained in (a) and draw the resulting trees after the deletions of the values 10 and 7 (in this order).

Q7. Consider the left and right binary rotations:



```
public class RBTNode {

// A RBTNode contains an element (element), two references
// on the left (left) and right (right) children and a color.
// For all x in the left subtree:  x.compareTo(element) < 0
// For all y in the right subtree: y.compareTo(element) > 0

    protected Comparable element;
    protected boolean color;
    protected RBTNode left, right;

    private static final boolean Red = true;
    private static final boolean Black = false;
```

a) (5) Implement "rotateRight" (PS. pseudo-code ok).

```
private RBTNode rotateRight( RBTNode h ) {








}
```

b) (5) Implement "rotateLeft" (PS. pseudo-code ok).

```
private RBTNode rotateLeft( RBTNode h ) {









}
```

c) (4) What is the complexity of the operations you implemented in (a) and (b)?

Q8. Consider a hash table implemented in an array of N single-linked lists (N > 1) and the hash function hash(e) = e mod N.

a) (8) Draw the resulting table if we insert e = 1, 4, 8, 9, 11, 15 and 17 (in this order) when N = 4.

b) (2) What is the complexity in best case, average, and worst case of the insertion?

c) (2) What is the complexity in best case, average and worst case of the deletion?

Q9. Consider storing a very large collection of files and a function allowing for retrieving a file by its name. We know the size of the collection, N. You must perform many retrievals and very few updates (insertions and deletions). The name of the file serves as the key. Describe in your words the pros and cons of using the following data structures to manage your collection:

a) (2) An array of size N using binary searches.

b) (2) A stack of size N.

c) (2) A binary search tree.

d) (2) A red-black tree.

e) (2) A hash table of N single-linked lists.

f) (2) Which data structure would you propose to manage this collection if in another application the numbers of retrievals and updates (insertions and deletions) would be almost equal (#retrievals ~= #updates), and why?

Appendix : abstract type **Stack**

```
import java.util.Iterator;

public interface Stack {

    // Access methods ...

    public boolean isEmpty();
    // Return true iff the stack is empty.

    public int size();
    // Return the size of the stack.

    // Transformation methods ...

    public void clear();
    // Empty the stack.

    public void push( Object x );
    // Pushes x on the top of the stack.

    public Object pop();
    // Unstack,
    // return null if the stack is empty.

    // Iterator ...
    public Iterator iterator();
    // Return an iterator which visits all elements
    // in the stack, from the first to the last.
}
```