Name :_____

Permanent code : _____

Predoctoral exam, check here ☐

Directives :

- Write your name and permanent code above.
- Read all questions and **answer directly on these pages**.
- Only a pen (or pencil) is allowed, **no documentation, computer, phone, or other objects**.
- This exam has 8 questions for 155 points (an average of about 1 point per minute).
- This exam contains 19 pages.
- **Write clearly**, especially when you develop your answers.
- You have 160 minutes to complete this exam.

GOOD LUCK AND SUMMER!

1	/ 15
2	/ 20
3	/ 10
4	/ 15
5	/ 10
6	/ 30
7	/ 20
8	/ 35
Total	/155

Q1. (15) We want to develop a system to manage dynamic memory allocations. We wrap the functions `malloc()` and `free()` with our own functions, which record the allocation-related information in a `block`. `malloc` is a function of the standard C library allowing one to dynamically allocate memory. The so allocated memory is freed using the free function. A `block` is storing the memory allocation address (the one returned by `malloc`) and the size of the requested memory in bytes. A `block` is added to a collection at each memory allocation. The `block` corresponding to an allocation is sought and destroyed when it is de-allocated. Propose a data structure to manage the `block` collection that will guarantee that its users will minimally be penalized in performances. Explain how and why your data structure guarantees its performance.

Q2. Consider the sorting methods we saw in the course.

a) (5) What is a stable sorting method?

b) (15) Is heapsort stable? Why?

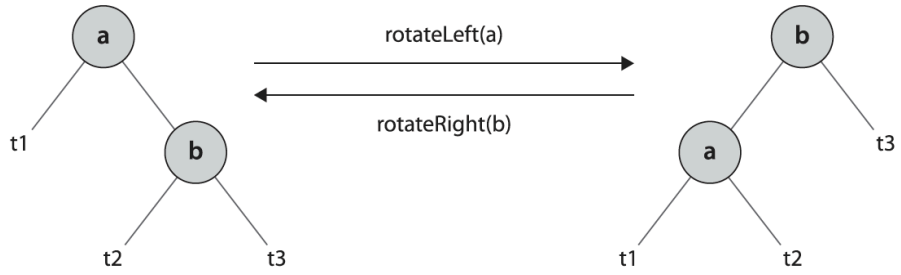
Q3. (10) Complete the code for the sequential search of the element `t` in the array `A` of `n` elements. The function returns `true` is `t` is in `A`, or `false` otherwise.

```
search( A, t )




end
```

Q4. Consider the left and right rotations of binary search trees:




```
public class RBTNode {

// A RBTNode contains its element (element), two references
// on its 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 couleur;
    protected RBTNode gauche, droite;

    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).

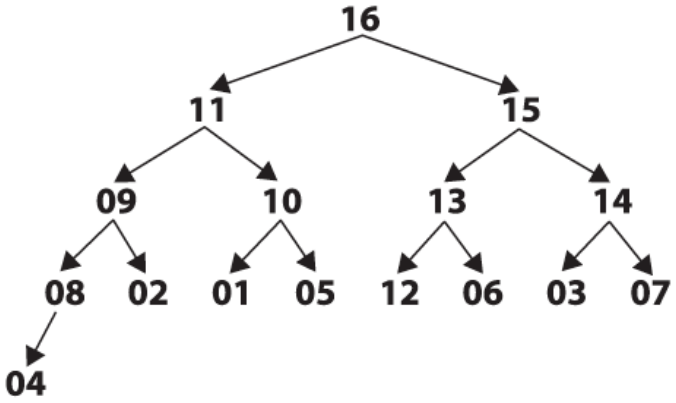
```
private RBTNode rotateLeft( RBTNode h ) {

}
```

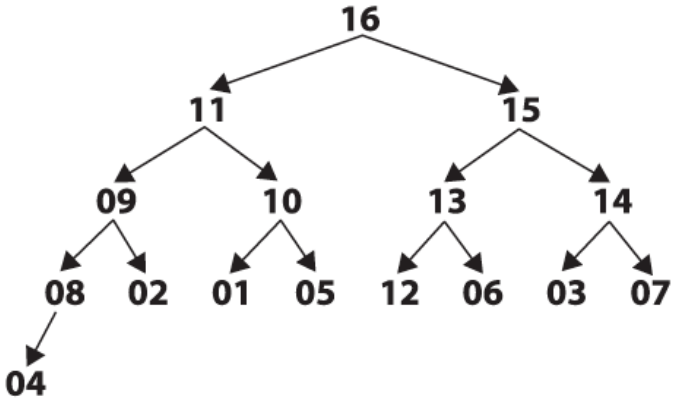
c) (5) What are the complexities of the operations you implemented in (a) and (b)?

Q5. Consider the heap data structure.

a) (5) Draw the heap resulting from changing the 01 priority into 13 in the following heap.



b) (5) Draw the heap resulting from changing the 16 priority into 03 in the following heap.

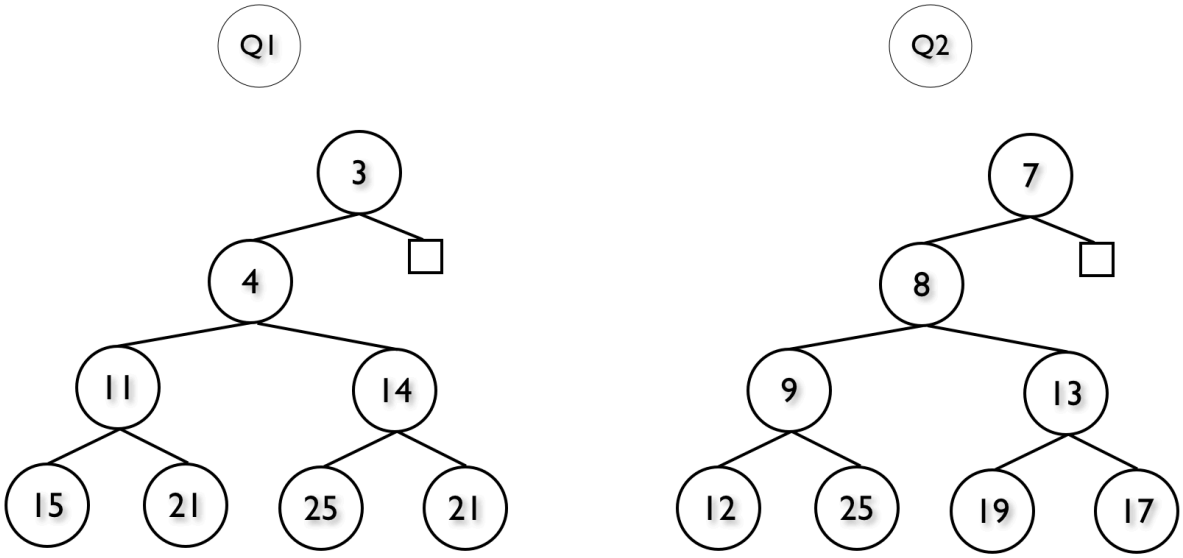


Q6. Consider binomial queues to manage a collection of priorities-min, that is the smaller is a key the highest priority it is given.

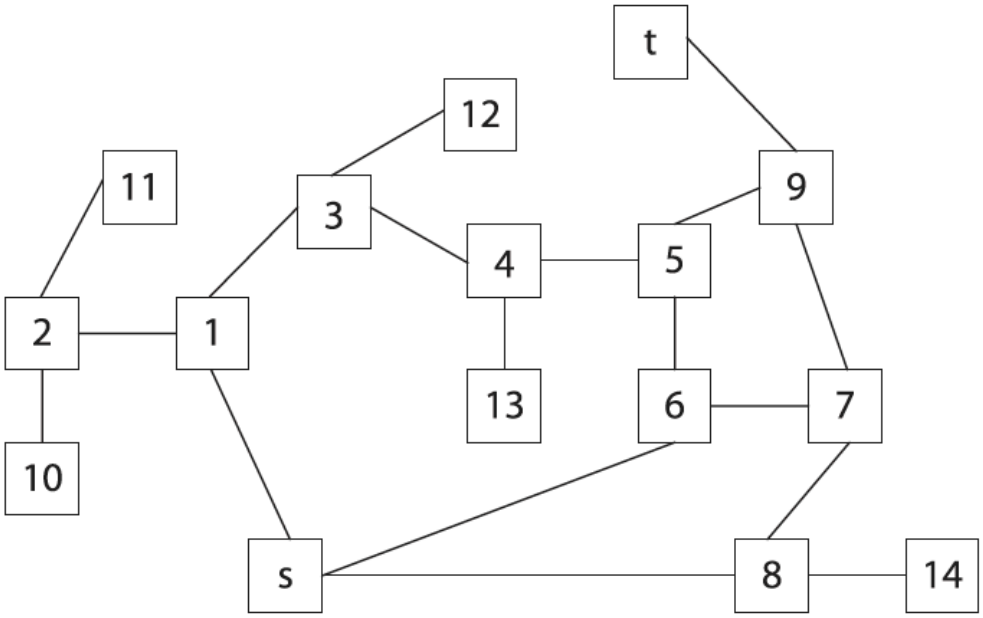
a) (10) Draw the binomial queue min resulting from inserting the following keys in this order: 9, 5, 17, 21, 99, 12, 23, 12, 77, 33, 24, 23, 53.

b) (10) Draw the binomial queue min resulting from applying two "getMin" operations on the binomial queue you obtained in (a).

c) (10) Draw the binomial queue min resulting from merging queues, `Q1` and `Q2`, below:

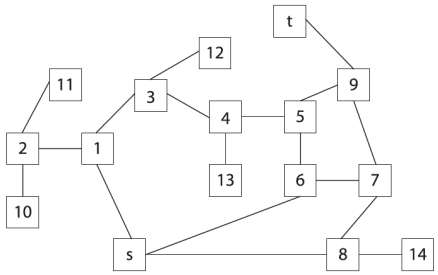


Q7. Consider the following graph (s = 0; t = 15) :

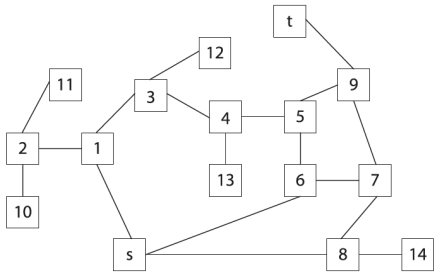



```
v[0](W): [< 1, 0 >, < 6, 0 >, < 8, 0 >]
v[1](W): [< 0, 0 >, < 2, 0 >, < 3, 0 >]
v[2](W): [< 1, 0 >, < 11, 0 >, < 10, 0 >]
v[3](W): [< 1, 0 >, < 12, 0 >, < 4, 0 >]
v[4](W): [< 3, 0 >, < 13, 0 >, < 5, 0 >]
v[5](W): [< 4, 0 >, < 6, 0 >, < 9, 0 >]
v[6](W): [< 0, 0 >, < 5, 0 >, < 7, 0 >]
v[7](W): [< 6, 0 >, < 8, 0 >, < 9, 0 >]
v[8](W): [< 0, 0 >, < 7, 0 >, < 14, 0 >]
v[9](W): [< 5, 0 >, < 7, 0 >, < 15, 0 >]
v[10](W): [< 2, 0 >]
v[11](W): [< 2, 0 >]
v[12](W): [< 3, 0 >]
v[13](W): [< 4, 0 >]
v[14](W): [< 8, 0 >]
v[15](W): [< 9, 0 >]
```

a) (10) In which order the vertices of this graph will be visited if we apply a depth-first search starting at vertex s? The depth-first search uses a stack to store the vertices to be visited. Show the status of the stack at each step of the search.



b) (10) Same question as (a), but if we apply a breadth-first search. The breadth-first search uses a queue to store the vertices to be visited. Show the status of the queue at each step the search.



Q8. Consider a search tree for a game where at state `b` more states are reachable. Consider the classical search methods in game trees: depth-first, breadth-first, and A*.

a) (5) How many states this game has at depth `p`?

b) (5) How many different states at minimum will be explored by depth-first search if a winning state is found at depth `p`?

c) (5) Same question as (b), but for breadth-first search.

d) (5) Consider the A* search method. In your own wording, what is an admissible heuristic?

e) (5) In the case where we use an admissible heuristic with A*, how many different states of the game at minimum will be visited if a winning state is found at depth p?

f) (5) Same question as (e), but when A* does not use an admissible heuristic.

g) (5) Same question as (f), but in the worst case, that is the number of different states that could be visited at maximum.