

Final Exam

CS1027A

*University of Western Ontario***Exam Code 111. Part I and II.**

Mark the multiple choice questions strictly according to this answer sheet.

1. A
2. B
3. B
4. B
5. A
6. B
7. B
8. A
9. B
10. B
11. A
12. E
13. C
14. B
15. D
16. G
17. B
18. C
19. B
20. C
21. B
22. A
23. B
24. C
25. C
26. E
27. E
28. A
29. A
30. B

1 (24 marks) Part III. Analysis of Algorithms

For each one of the following 4 questions, compute the time complexity of the given code. You **must explain how you computed the time complexity** and you must give the order (big-Oh) of the time complexity. For method *mystery()*, assume it was called on the root of a tree containing n nodes.

31. (3 marks)

```
1  int i, j, k, x = 0;
2  for (i = n; i > 0; i--) {
3      k = i * n;
4      for (j = k; j < n; j++) {
5          x = x + j;
6      }
7      i = i - k;
8  }
```

32. (3 marks)

```
1  for (int i = 0; i < n; i++) {
2      for (int j = 0; j < 5; j++) {
3          for (int k = 0; k < n; k++) {
4              if ((i != j) && (i != k)) {
5                  System.out.println(k);
6              }
7          }
8      }
9  }
```

33. (3 marks)

```
1 public String modify(String str) {
2     if (str.length() <= 1) return "";
3     int half = str.length() / 2;
4     modify(str.substring(half));
5 }
```

34. (3 marks)

```
1 public void mystery(BinaryTreeNode<T> node) {
2     if (node == null) return;
3     if (node.getLeft() != null || node.getRight() != null)
4         System.out.println(node.getElement());
5     mystery(node.getLeft());
6     mystery(node.getRight());
7 }
```

35. (3 marks) Write **exactly one** for loop whose start condition is *int i = 0;* and whose end condition is *i < n;* such that your code fragment has a time complexity of $O(n^2)$.
36. (3 marks) Write **exactly three** for loops that are nested inside of each other whose start conditions are *int i = 0;*, *int j = 0;*, and *int k = 0;*, respectively, and whose end conditions are *i < n;*, *j < n;*, and *k < n;*, respectively, such that your code fragment has a time complexity of $O(n)$.

37. (3 marks) Consider a sorted list implemented using a circular array. Let *first* be the index of the first element in the list and *last* be the index of the last element in the list. Let the list store n integer values, where n is an odd number. Operation *median()* returns the value of the median or the $\frac{n+1}{2}$ -th smallest element in the list. For example, the median of this sorted list: 1, 4, 5, 8, 9, 12, 15, is 8. What is the time complexity of the best possible implementation of operation *median()*? Explain how it would work.
38. (3 marks) Consider the same previous question, but this time the sorted list is implemented using a doubly linked list. Reference variable *first* points to the first node in the list and *last* points to the last node in the list. What is the time complexity of the best possible implementation of operation *median()*? Explain how it would work.

2 (35 marks) Part V. Binary Trees

39. For each of the following tree diagrams, determine whether or not the tree is a binary search tree. If it is not a binary search tree, explain why.

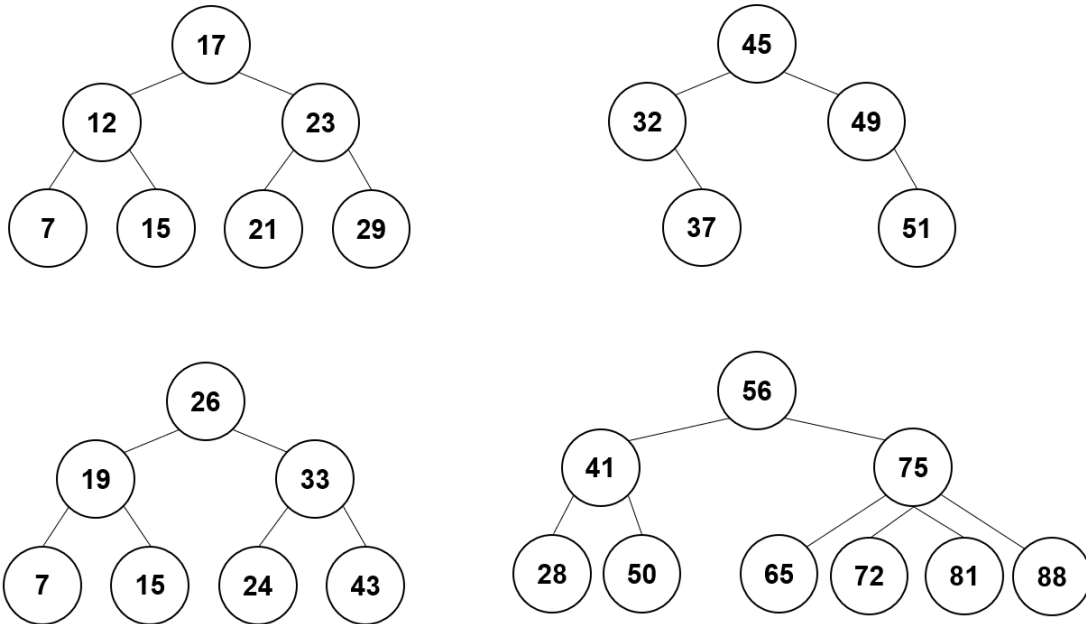


Figure 1

(Top-left) (1 mark)

(Top-right) (1 mark)

(Bottom-left) (1 mark)

(Bottom-right) (1 mark)

40. Consider the following tree:

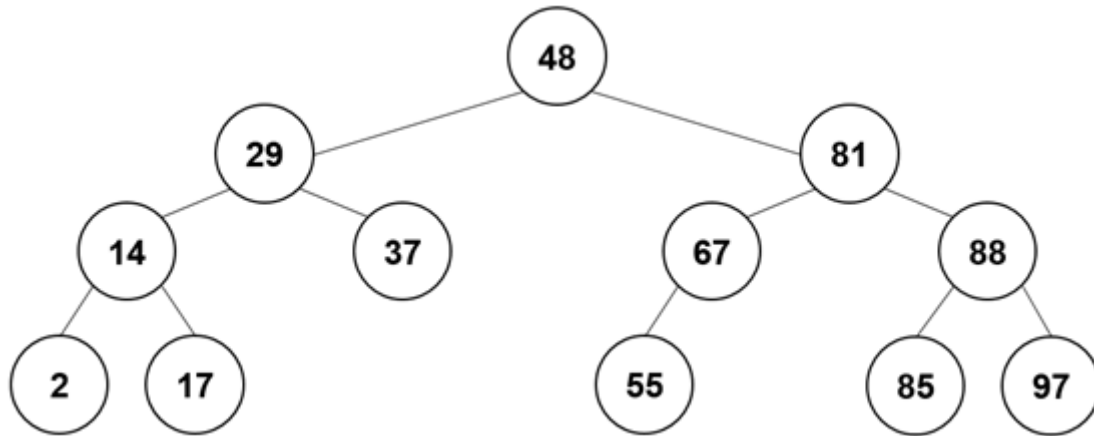


Figure 2

- a) (1 mark) What is the height of this tree?
- b) (1 mark) What is the level of the node containing "17"?
- c) (1 mark) What is the value contained in this tree's root node?
- d) (1 mark) What are the ancestors of the node containing "81"?
- e) (1 mark) How many leaf nodes are contained in this tree?
- f) (1 mark) What is the degree of the node containing "67"?
- g) (1 mark) What is the degree of this tree?
- h) (1 mark) Trace out the pre-order traversal of this tree.
- i) (1 mark) Trace out the in-order traversal of this tree.
- j) (1 mark) Trace out the post-order traversal of this tree.
- k) (1 mark) Trace out the level-order traversal of this tree.

41. (3 marks) Draw the binary tree from the given level-order traversal. We have not included "null" for every empty position in every level of the tree; instead, we have written null when an **existing** node has no child in that position.

- 27, 14, 11, null, 5, 21, 32, 19, null, null, 7, 36, null, null, null, null, null, 10, null, null, null

42. (3 marks) Draw the binary tree such that:

- a pre-order traversal visits the nodes in this order: E, A, D, B, C, and
- a post-order traversal visits the nodes in this order: A, B, C, D, E.

43. (4 marks) Consider the binary tree shown in Figure 3. Perform the iterative pre-order tree traversal using a stack as shown to you in class. Write your answers in the U-shaped diagrams in Figure 4. For the first stack, show the value on the stack before entering the loop. For the rest of the stacks, show the values on the stack at the end of each loop iteration.

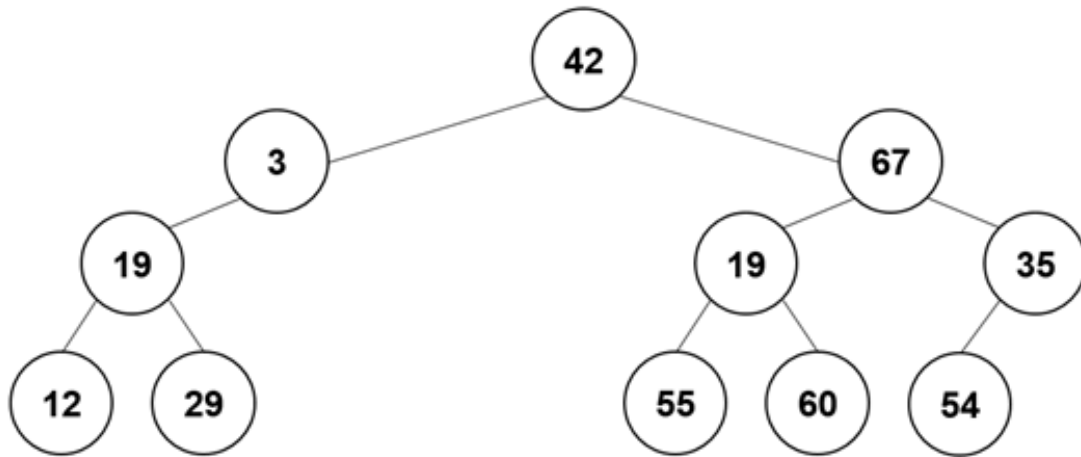


Figure 3

Visited nodes: _____

--	--	--	--	--	--

--	--	--	--	--	--

Figure 4

44. (4 marks) Evaluate the following expression tree. Show your work by drawing on the Figure (write numbers above nodes where appropriate, as shown in class).

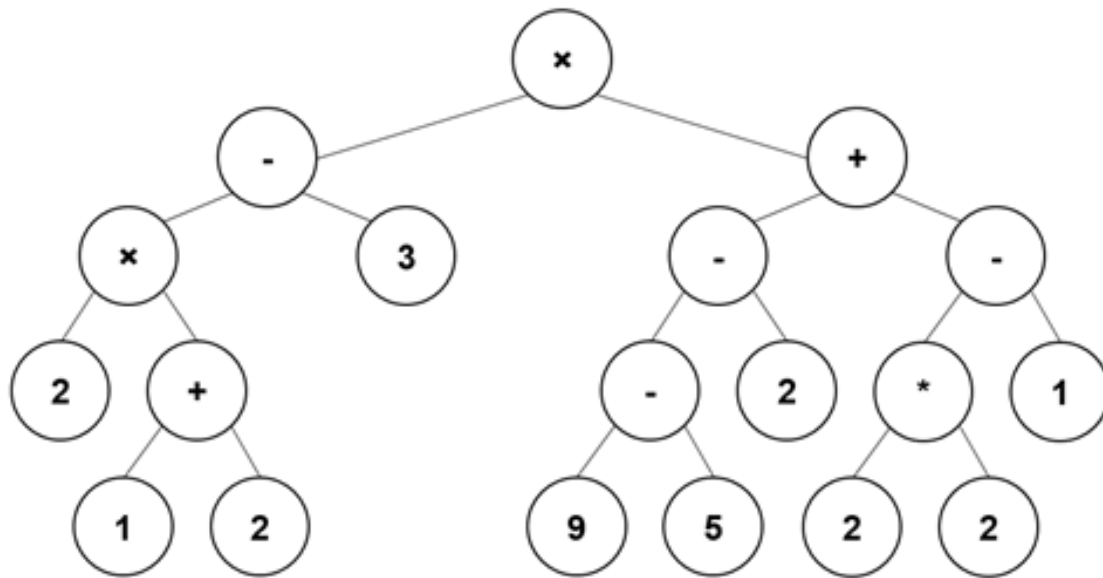


Figure 5

45. (1 mark) What kind of traversal did you use to evaluate the expression tree?
46. (1 mark) Will the root of an expression tree always be the final operator?

In the following 3 questions, explain the general approach to building an expression tree using a stack.

47. (1 mark) What is the input to the algorithm that builds an expression tree?
48. (1 mark) What variable type is being stored in the stack?
49. (2 marks) When building an expression tree, explain what happens when you come across an operator from the input.

3 (24 marks) Part VI. Sorting

50. (7 marks) Consider the array $A = [7, 9, 2, 5]$. Trace through a stack-based implementation of selection sort on this array. Assume that during each step, you scan through the entire array to find the minimum value, then replace that index of the array with *null*. After finding the minimum value in each step (write down what this number is), draw 3 stacks: (1) the *sorted* stack with the new number in it, (2) the *temp* stack holding whatever numbers are necessary to have drawn the previous *sorted* stack, and (3) the *sorted* stack with everything from *temp* in it.

51. (1 mark) Analyze this algorithm using Big-Oh notation.

52. (8 marks) Consider the array $A = [18, 9, 7, 2, 15, 12, 5]$. Trace through the quicksort algorithm on this array. Use the **last** element as the pivot. For each recursive call, draw the execution stack (don't worry about other methods such as *main()*), show the pivot, and show the arrays smaller, equal, and larger. During some of the recursive calls some of the sub-arrays will be adjusted so that they become sorted, make sure this happens in the correct recursive call!

53. (8 marks) *MergeSort* is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sub-list consists of a single element. These sub-lists are then merged together in a manner that forms a larger sorted sub-list. Eventually, the algorithm will have sorted the list and will terminate.

Consider the following pseudocode for mergesort:

```
1 MergeSort(int[] A, int left, int right) {
2     if (right > left) {
3         int middle = (left + right) / 2;
4         MergeSort(A, left, middle);
5         MergeSort(A, middle + 1, right);
6         Merge(A, left, middle, right);
7     }
8 }
```

We first call *MergeSort()* with the parameters $A = [3, 1, 7, 2]$, $left = 0$, and $right = A.length - 1$. Therefore, this first method call represents sorting the entire array; however, as this is a recursive method, our first step is to divide the full array into two sub-arrays. Thus, on line 4, we call *MergeSort()* again but adjust the value of *right* so that this second method call represents sorting only the first half of the entire array.

This process of dividing the array into smaller sub-arrays is continued until we are considering sub-arrays of size 1. This is the base case. Only after we have encountered the base case will the method *Merge* be called to piece the sub-arrays back together in a sorted manner.

You can assume that calling method *Merge()* will modify the array *A* such that the elements in the range of *left* to *right* are now a sorted sub-set.

Fill in the Figures on the following 2 pages as you trace through the algorithm on the input array $A = [3, 1, 7, 2]$. We have provided the state of the execution stack after every method invocation and after every method termination. You need to track the values of *A*, *left*, and *right* (and sometimes *middle*).

Hint. You might find it useful to write comments to yourself beside each activation record to remind yourself where in the function you are returning to. For example, the first activation record (provided for you) represents calling *MergeSort* the first time. The second activation record represents calling *MergeSort* from line 4 of the previous activation record.

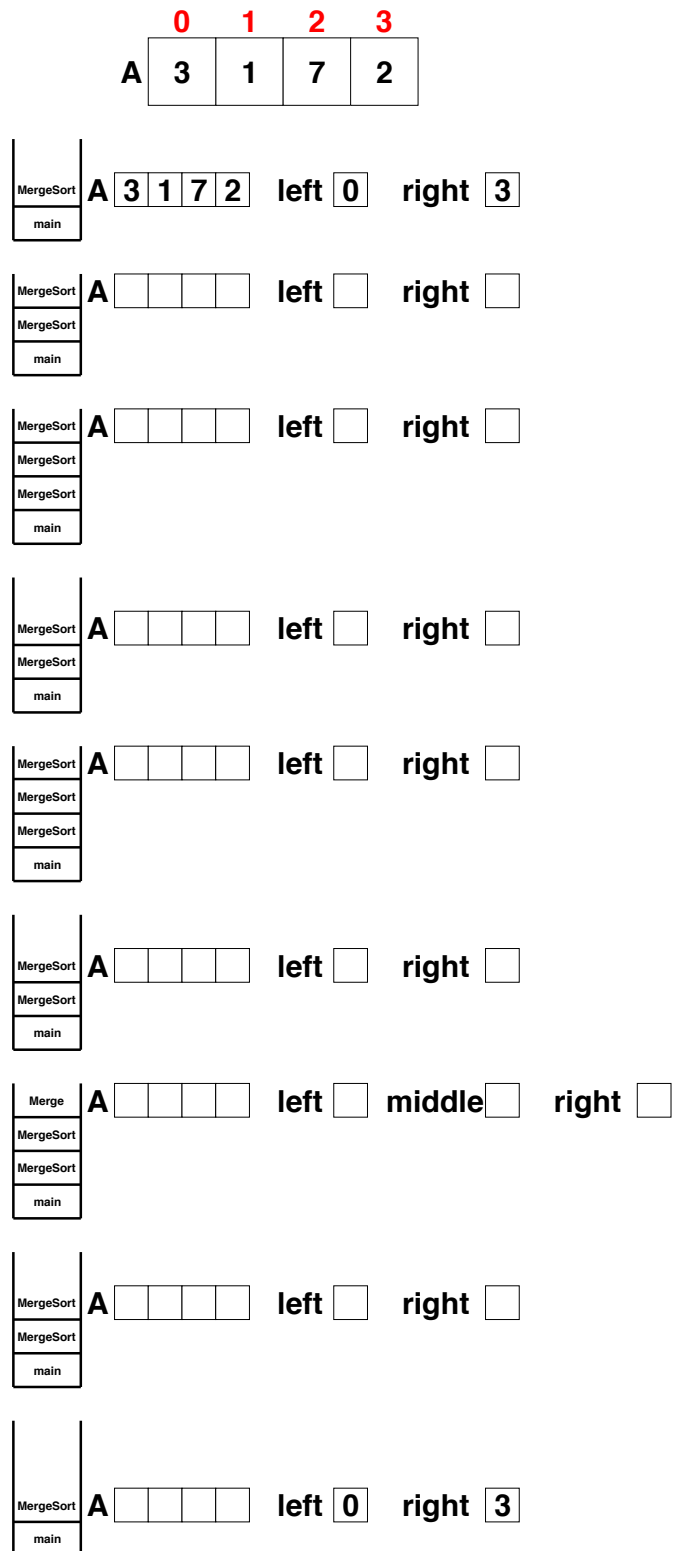


Figure 6

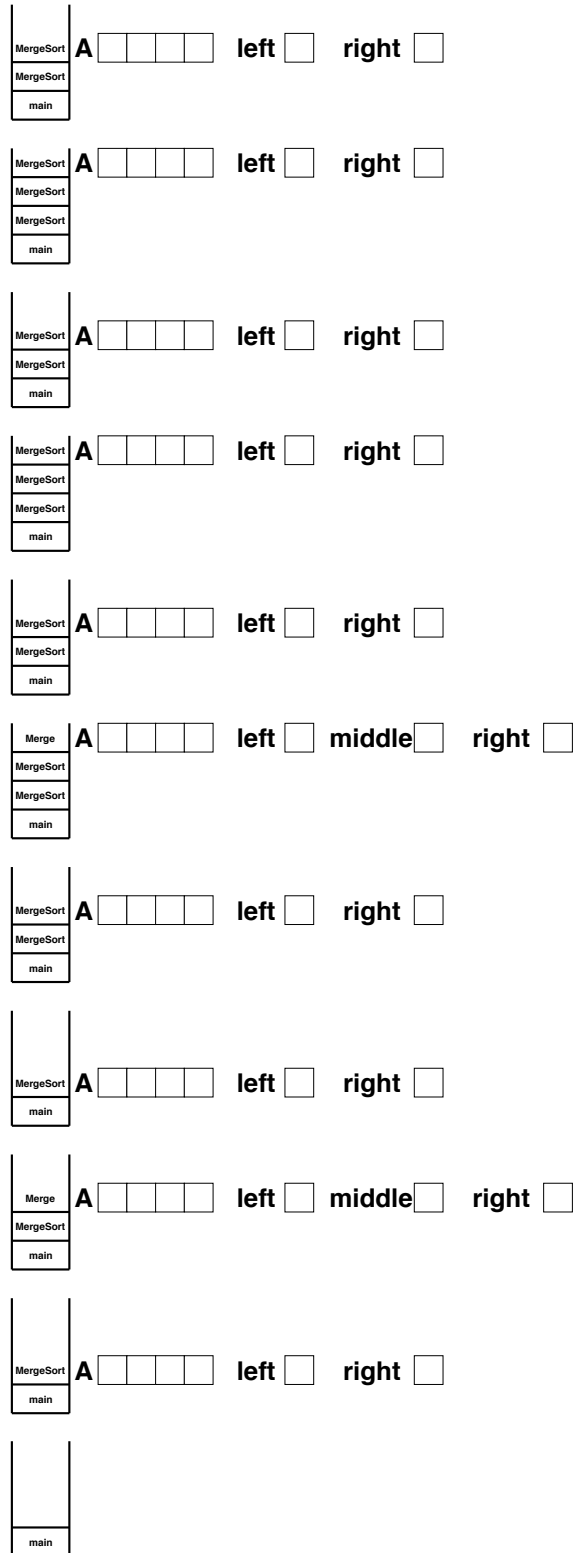


Figure 7

4 (24 marks) Part VII. Algorithm Design

54. "Simon" is a memory-based game that requires its player to remember a given sequence of colors. It's an electronic game on a device that contains 4 colored areas that can light up. A typical game plays like this:

- A new color is added to the existing sequence of colors.
- The colors in the sequence are displayed for the player one at a time.
- The player then has to enter the sequence of colors that they can remember.
- If the player's answer was correct, repeat this process.

a) (1 mark) Which of the collection types that were taught in this course would be best suited for tracking the sequence of colors in the "Simon" game? Give a brief explanation for your response.

b) (6 marks) Write an algorithm *public void playSimon(int n)* in Java or in **detailed** Java-like pseudocode that implements the game as described above, where *n* is the maximum length of the color sequence for a particular game. You can make the following assumptions:

- Whichever collection you decide to use has already been declared as an instance variable, you just need to initialize it and use it.
- There exists a method *public Color getNextColor()* that returns the next color in the sequence.
- There is a method called *public void showColor(Color col)* that you can call to display the color for the player. You can assume that this method displays the color for an appropriate amount of time to the player.
- There exists a method *public boolean checkAnswer()* that will handle everything to do with checking the players response; if the player was correct, this method will return true. Otherwise, this method will return false.

You can use the following page to write your answer.

(This page is for the algorithm *public void playSimon(int n)*)

c) (1 mark) Analyze your algorithm using Big-Oh notation.

55. (8 marks) Consider a binary search tree formed by linking together node objects of class *BinaryTreeNode*. The *BinaryTreeNode* class provides methods *getLeft()*, *setLeft()*, *getRight()*, *setRight()*, and *getElement()*. You can create a new node by calling the constructor of this class using *BinaryTreeNode<>(element)* to create a new node storing value *element* whose left and right children are *null*.

Write an algorithm *public void addReverse(BinaryTreeNode<T> node, T element)* in Java or in **detailed** Java-like pseudocode that adds a new node storing *element* into the binary search tree so that the ordering of the binary search tree is backwards compared to how we described it in class. More specifically, in this binary search tree, every node should have the following properties:

- Nodes in the left sub-tree should contain larger values.
- Nodes in the right sub-tree should contain smaller values.

You can assume that the variable *element* is of a class that implements *Comparable<T>*.

Additionally, you must:

- Increment an instance variable called *size* when you add a node to the tree.
- Adjust the instance variable called *root* if adding the first node to the tree.
- Throw a *DuplicatedKeyException* if *element* is already in the tree. Assume that *DuplicatedKeyException.java* already exists.

You can use the following page to write your answer.

(This page is for the algorithm *public void addReverse(BinaryTreeNode<T> node, T element)*)

(This page is for the algorithm *public int countSingleChild(BinaryTreeNode<T> node)*)

5 (8 marks) BONUS

57. (8 marks) Let q be a queue storing n integer values. Write an algorithm *public void partition(Queue q , int $target$)* in Java or in **detailed** Java-like pseudocode that receives as parameters the queue q and an integer value $target$ and it re-arranges the values in the queue so that all the values smaller than $target$ appear before the value of $target$ and all the values larger than $target$ appear after the value of $target$. Your algorithm must handle the possibility of duplicate entries of the value of $target$.

For this algorithm you can use **one** auxiliary stack s . You **cannot** use any other auxiliary data structures. You can use the following queue methods: *dequeue()*, *enqueue()*, *isEmpty()*, and *size()*. You can also use the following stack methods: *push()*, *pop()*, and *isEmpty()*.

For example, for the following queue on the left side and $target = 6$, your algorithm should produce a queue like the one on the right.

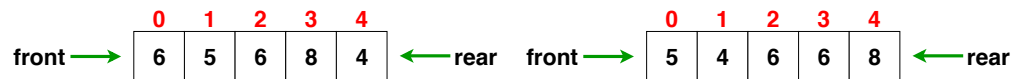


Figure 9

You can use the following page to write your answer.

(This page is for the algorithm *public void partition(Queue q, int target)*)

(This page for rough work; this page will not be marked)

(This page for rough work; this page will not be marked)

(This page for rough work; this page will not be marked)

(This page for rough work; this page will not be marked)