**COMP 6481 Assignment 2 Report**

**Zexin Peng**

**Student number: 40166520**

**Question 1**

**（1）**

Algorithm findConsecutiveElements(arr)

Input: the input array arr

for i <- 0 to arr.length() - 2 do

index <- i;

times <- 1;

for j <- i + 1 to arr.length() - 1 && arr[j] = arr[i] do

times <- times + 1;

i = j;

if times > 1

print("Value " + arr[i] + " is repeated " + times + " times starting at Index " + index);

**a.answer:**

I just iterate the whole array, and for every element I meet, I check its following element whether equals to it and count until meet a different one.

**b.answer:**

The time complexity of my algorithm is O(n), since the total cost of my algorithm is: 5n + n = 6n, which 6n >= n, so f(n) >= n, for n >= 0.

**c.answer:**

the time complexity of my algorithm is Ω(n), since the total cost of my algorithm is: 5n + n = 6n, which 6n >=n, So f(n) >= n, for n >= 0. g(n) = n, so my algorithm f(n) is Ω(n).

**d.answer:**

my space complexity is O(1) since I just create two variables to store the intermediate results.

**（2）**

Algorithm findConsecutiveElementsWithQueue(arr)

Input: the input array arr

Queue<Integer> indexStack <- new LinkedList<>();  
Queue<Integer> timesStack <- new LinkedList<>();

for i <- 0 to arr.length() - 2 do

index <- i;

times <- 1;

for j <- i + 1 to arr.length() - 1 && arr[j] = arr[i] do

times <- times + 1;

i = j;

if times > 1

indexStack.add(index);  
timesStack.add(times);

while !indexStack.isEmpty() do  
 index <- indexStack.poll();  
 print("Value " + arr[index] + " is repeated " + timesStack.poll() + " times starting at Index " + index);

System.out.println("Value " + arr[index] + " is repeated " + + " times starting at Index " + index);

**a.answer:**

I iterate the whole array, and for every element I meet, I check its following element whether equals to it, if they are equal, I add their index and the number of times that they repeat into queues. At last, I will iterate the queue and print the information as required.

**b.answer:**

The time complexity of my algorithm is O(n), since the total cost of my algorithm is: 5n + n + 6 \* n = 12n, which 12n >= n, so f(n) >= n, for n >= 0.

**c.answer:**

the time complexity of my algorithm is Ω(n), since the total cost of my algorithm is: 5n + n + 6 \* n = 12n, which 12n >=n, So f(n) >= n, for n >= 0. g(n) = n, so my algorithm f(n) is Ω(n).

**d.answer:**

my space complexity is O(n) since I use two queues to store the index and the number of times that they repeat of all repetitive elements.

**Question 2**

For question g and f, I answer them below the pseudocode.

**Case1**

**e.answer:**

I divide an array into 2 parts with the same length, the stack bottom positions are in the middle of the array. The two stacks are stack first and second respectively. Variable first and second are their stack top pointers.

**f.answer:**

**for push()**

Algorithm pushFirst(element)

Input: the pushed element

if isFirstFull() then   
 print("stack first is full!");  
 return;  
arr[first] = element;

first <- first – 1;

Algorithm pushSecond(element)

Input: the pushed element

if isSecondFull() then  
 print("stack second is full!");  
 return;  
arr[second] = element;

second <- second + 1;

The time complexity of my algorithm is O(1), since the total cost of my algorithm is: 1 + 1 + 1 = 3, which 3 >= 1, so f(n) >= 1, for n >= 0.

the time complexity of my algorithm is Ω(1), since the total cost of my algorithm is: 1 + 1 + 1 = 3, which 3 >= 1, So f(n) >= 1, for n >= 0. g(n) = 1, so my algorithm f(n) is Ω(1).

**for pop()**

Algorithm popFirst()

Output: the element popped from the first stack

if isFirstEmpty() then  
 print("stack first is empty!");  
 return null;

first <- first + 1;  
return arr[first - 1];

Algorithm popSecond()

Output: the element popped from the second stack

if isSecondEmpty() then  
 print("stack second is empty!");  
 return null;

second <- second - 1;  
return arr[second + 1];

The time complexity of my algorithm is O(1), since the total cost of my algorithm is: 1 + 1 + 1 = 3, which 3 >= 1, so f(n) >= 1, for n >= 0.

the time complexity of my algorithm is Ω(1), since the total cost of my algorithm is: 1 + 1 + 1 = 3, which 3 >= 1, So f(n) >= 1, for n >= 0. g(n) = 1, so my algorithm f(n) is Ω(1).

**for size()**

Algorithm sizeFirst()

Output: the size of the first stack

return arr.length / 2 - 1 - first;

Algorithm sizeSecond()

Output: the size of the second stack

return (second - arr.length) / 2;

The time complexity of my algorithm is O(1), since the total cost of my algorithm is 1, so f(n) >= 1 for n >= 0.

the time complexity of my algorithm is Ω(1), since the total cost of my algorithm is 1, So f(n) >= 1, for n >= 0. g(n) = 1, so my algorithm f(n) is Ω(1).

**for isEmpty()**

Algorithm isFirstEmpty()

Output: whether the first stack is empty

return first = arr.length / 2 - 1? true: false;

Algorithm isSecondEmpty()

Output: whether the second stack is empty

return second = arr.length / 2 ? true: false;

The time complexity of my algorithm is O(1), since the total cost of my algorithm is 1, so f(n) >= 1 for n >= 0.

the time complexity of my algorithm is Ω(1), since the total cost of my algorithm is 1, So f(n) >= 1, for n >= 0. g(n) = 1, so my algorithm f(n) is Ω(1).

**for isFull()**

Algorithm isFirstFull()

Output: whether the first stack is full

return first = -1 ? true: false;

Algorithm isSecondFull()

Output: whether the second stack is full

return second = arr.length? true: false;

The time complexity of my algorithm is O(1), since the total cost of my algorithm is 1, so f(n) >= 1 for n >= 0.

the time complexity of my algorithm is Ω(1), since the total cost of my algorithm is 1, So f(n) >= 1, for n >= 0. g(n) = 1, so my algorithm f(n) is Ω(1).

**Case2**

**e.answer:**

The stack bottom of the first stack is on the far left side, and the stack bottom of the second stack is on the far right side. They will meet in the middle of the array.

**f.answer:**

**for push()**

Algorithm pushFirst(element)

Input: the pushed element push

if isFull() then  
 print("stack first is full!");  
 return;  
arr[first] = element;

first <- first + 1;

Algorithm pushSecond(element)

if isFull() then  
 print("stack second is full!");  
 return;  
arr[second] = element;

second <- second - 1;

The time complexity of my algorithm is O(1), since the total cost of my algorithm is: 1 + 1 + 1 = 3, which 3 >= 1, so f(n) >= 1, for n >= 0.

the time complexity of my algorithm is Ω(1), since the total cost of my algorithm is: 1 + 1 + 1 = 3, which 3 >= 1, So f(n) >= 1, for n >= 0. g(n) = 1, so my algorithm f(n) is Ω(1).

**for pop()**

Algorithm popFirst()

Output: the element popped from the first stack

if isFirstEmpty() then  
 print("stack first is empty!");  
 return null;

first <- first - 1;  
return arr[first + 1];

Algorithm popSecond()

Output: the element popped from the second stack

if isSecondEmpty() then  
 print("stack second is empty!");  
 return null;

second <- second + 1;  
return arr[second - 1];

The time complexity of my algorithm is O(1), since the total cost of my algorithm is: 1 + 1 + 1 = 3, which 3 >= 1, so f(n) >= 1, for n >= 0.

the time complexity of my algorithm is Ω(1), since the total cost of my algorithm is: 1 + 1 + 1 = 3, which 3 >= 1, o f(n) >= 1, for n >= 0. g(n) = 1, so my algorithm f(n) is Ω(1).

**for size()**

Algorithm sizeFirst()

Output: the size of the first stack

return first;

Algorithm sizeSecond()

Output: the size of the second stack

return arr.length - second - 1;

The time complexity of my algorithm is O(1), since the total cost of my algorithm is 1, so f(n) >= 1 for n >= 0.

the time complexity of my algorithm is Ω(1), since the total cost of my algorithm is 1, So f(n) >= 1, for n >= 0. g(n) = 1, so my algorithm f(n) is Ω(1).

**for isEmpty()**

Algorithm isFirstEmpty()

Output: whether the first stack is empty

return first == 0 ? true: false;

Algorithm isSecondEmpty()

Output: whether the second stack is empty

return second == arr.length - 1 ? true: false;

The time complexity of my algorithm is O(1), since the total cost of my algorithm is 1, so f(n) >= 1 for n >= 0.

the time complexity of my algorithm is Ω(1), since the total cost of my algorithm is 1, So f(n) >= 1, for n >= 0. g(n) = 1, so my algorithm f(n) is Ω(1).

**for isFull()**

Algorithm isFull()

Output: whether the stack is full

return first > second ? true: false;

The time complexity of my algorithm is O(1), since the total cost of my algorithm is 1, so f(n) >= 1 for n >= 0.

the time complexity of my algorithm is Ω(1), since the total cost of my algorithm is 1, So f(n) >= 1, for n >= 0. g(n) = 1, so my algorithm f(n) is Ω(1).

**Question 3**

**a.answer:**

I use a stack inside the class to store the current maximum number, so for max() method, I only need to pop the first element.

Algorithm max ()

Output: the current max element in the stack

return maxStack.peek();

**b.answer:**

The time complexity of my algorithm is O(1), since the total cost of my algorithm is 1, so f(n) >= 1 for n >= 0.

the time complexity of my algorithm is Ω(1), since the total cost of my algorithm is 1, So f(n) >= 1, for n >= 0. g(n) = 1, so my algorithm f(n) is Ω(1).

**c.answer:**

Yes, it could be O(1). I use a stack inside the class to store the current maximum number named maxStack and a variable named maxNum to store the current max number. When a new number pushed into the stack, I compare it to the maxNum, if it is greater than maxNum set maxNum to it, otherwise push the maxNum into the maxStack. For pop() method, I pop the element from the maxStack too, and then I compare the element popped from the maxStack with the maxNum, if they are equal, set the maxValue to the current top element in the maxStack.

Algorithm push(item)

Input: the pushed element

if item > maxNum then  
 maxNum = item;  
addElement(item);  
maxStack.push(maxNum);

Algorithm pop()

Output: top element of the stack

len <- size();  
topElement <- peek();  
removeElementAt(len - 1);  
top <- maxStack.pop();  
if max = maxNum then  
 if maxStack.size() = 0 then  
 maxNum <- Integer.MIN\_VALUE;  
 else then  
 maxNum = maxStack.peek();  
return topElement;

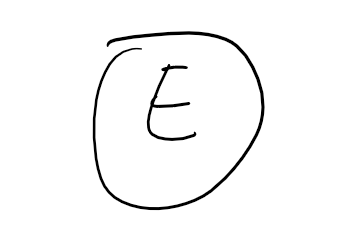
**Question 4**

It is impossible to draw such a single tree. My reason is as below:

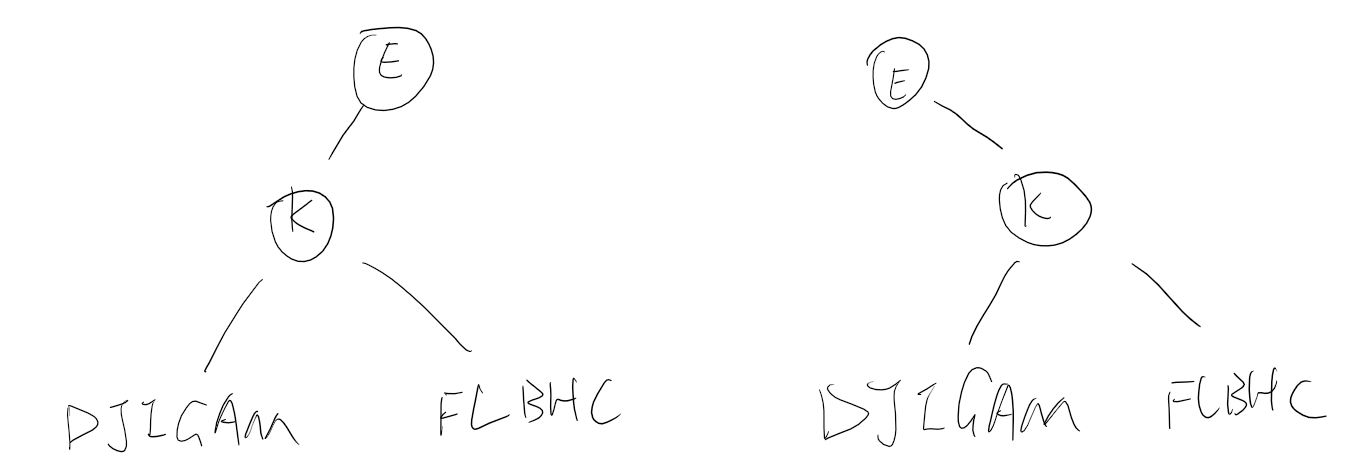
A preorder traversal of T yields: E K D M J G I A C F H B L;

A postorder traversal of T yields: D J I G A M K F L B H C E.

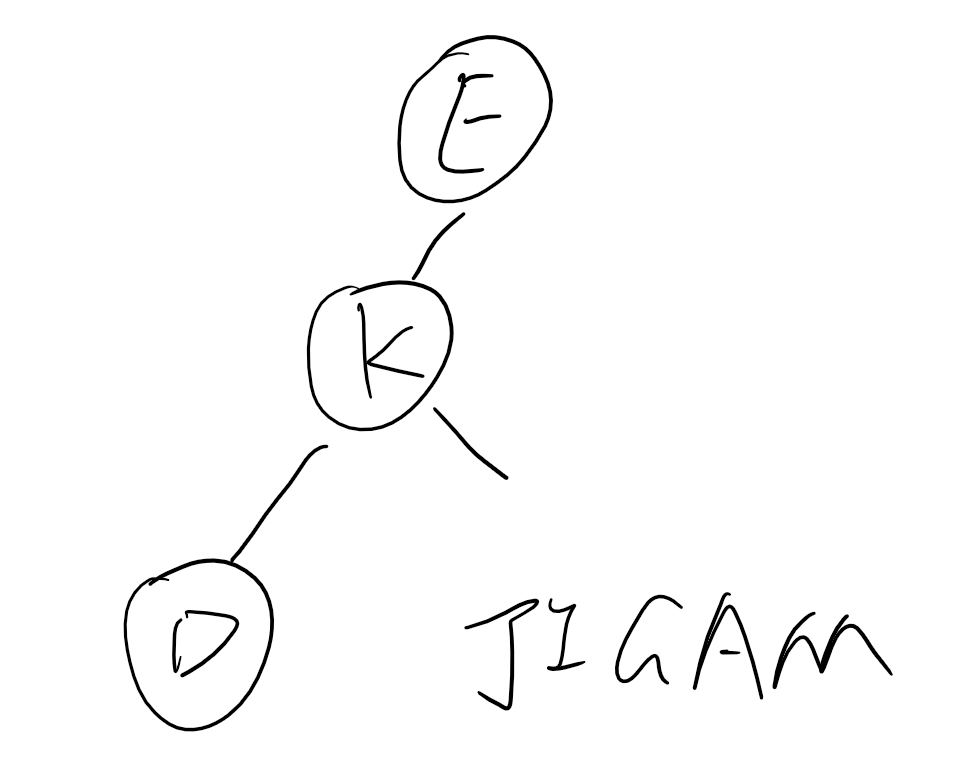
In preorder, the first element is the root, so the tree will be:



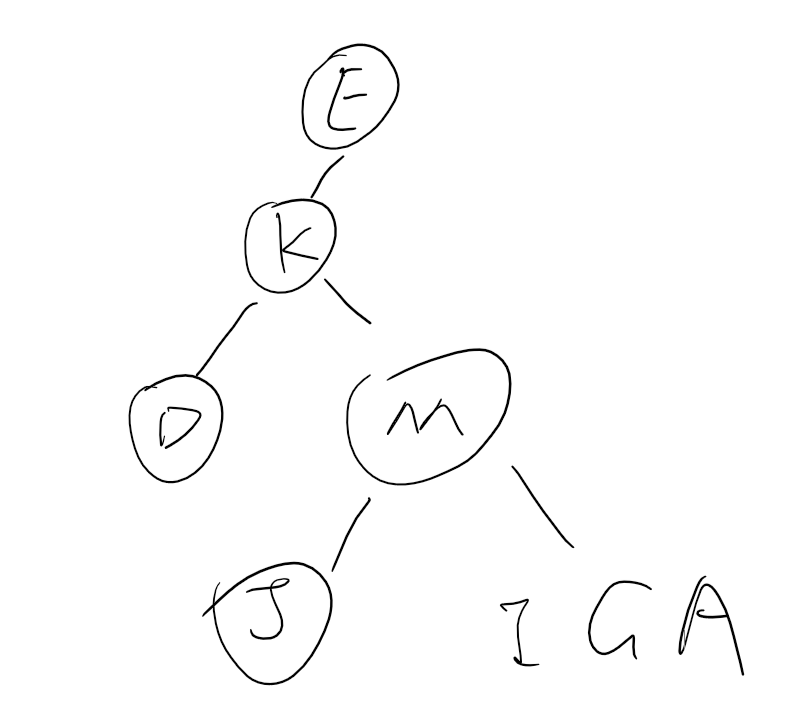
The second element is K, and we could find the ‘K’ in the postorder traversal, so the two tree could be either of the below:



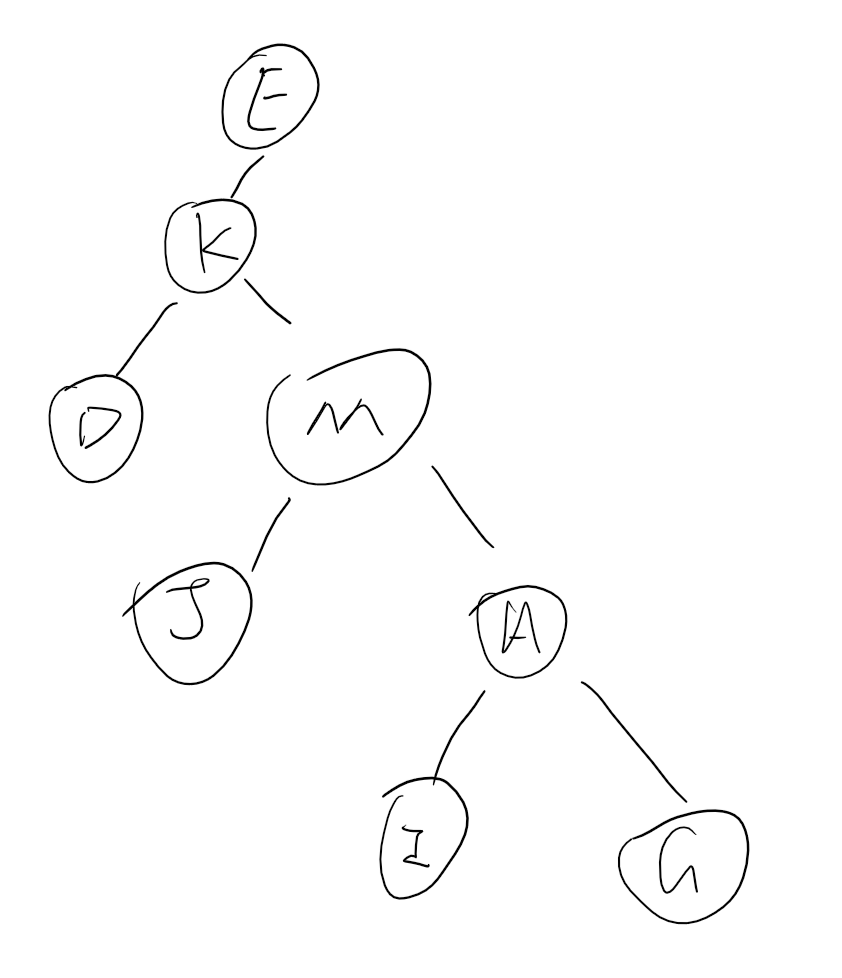
Let me suppose the left one. After analyzing the letter ‘D’, the tree will be



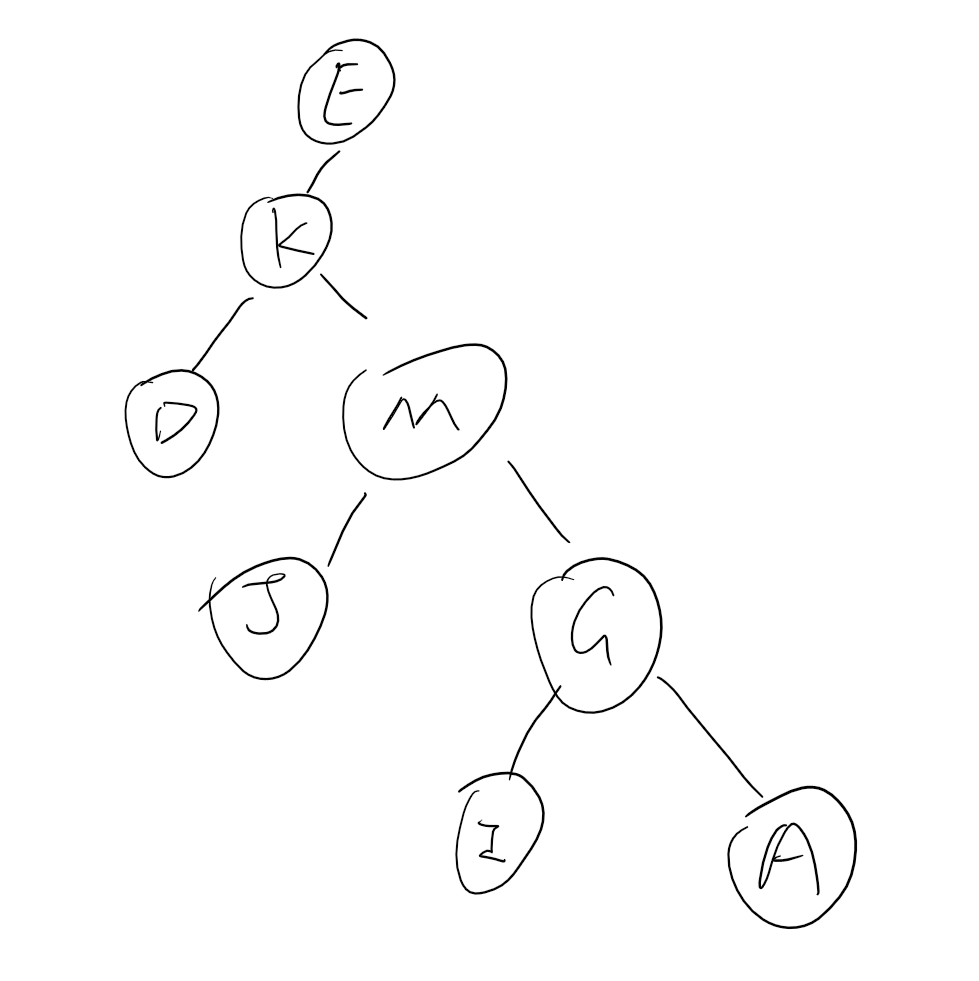
After analyzing the letter ‘M’, the tree will be



But in this step, if we follow the postorder of the traversal, it will be:



But if we follow the preorder of the traversal, it will be:



They are conflicting, so we cannot get such a tree.