Mohammad Ali Zahir

COMP 352: Data Structures and Algorithms

22nd February 22, 2020

Assignment 2

1.

a)

Function max()

/ Assume that this is an array of integers*

Declare and initialise an int called max_element =0

Declare and initialise an int called element=0

Stack stk = new Stack ();

Create a new stack temp to store the max

while stk is not empty {

do

element = stk.pop();

if element > max_element then

element ← max_element

temp.push(element)

end while

while temp is not empty

do

element ← *temp.pop()*

stk.push(element)

end while loop

return max_element

b) The complexity of this solution is **O(n)**, since to find the max value in the stack it would have to check one by one through the whole stack. The worst case for this is if the last element is the max, **hence the complexity is O(n).**

c) Yes. It is possible for all three to have O(1) method.

Create a variable max_element that tracks and stores the current max element in the stack as we go through.

We also would need to change the pop(), and the push() method.

int element =0;

int point = 0; // to keep track of what will be returned

For **pop():**

if stack is empty then

return

element ← stack.pop()

if element > max_element

t ⟵ max_element

max_element ⟵ 2* max_element – element

return t;

else

return element;

For **push(t) :**

if stack is empty then

stack.add(t)

max_element ⟵ t

else if t <= max_element then

stack.add(t)

else

stack.add(2*t– max_element)

max_element ⟵ t

For max() :

if stack is empty

return 0

else

return max_element

2.

a) By using an array for the implementation of two stacks: we are able to keep a tracker for the two pointers for the right and left position of the array. Whenever an element is pushed on the first stack, it grows to the right, and the second stack grows to the left. Once both pointers are equal to each (left and right), the array is then full and we can't insert anything more in the array.

b)

Create an array a1[] of size n.
Create two pointers to keep where we are in the array.

l ⟵ 0

r ⟵ 0

pushleft(int e)
if (l + r is equal n)

print ("stack is full")

end if statement

change a1[l] ⟵ e

increment l

end pushleft

pushright(int e)
if (l + r is equal n)

print ("stack is full")

end if statement

decrement r

change a[r] ⟵ e

end pushright

popleft()
if l is equals 0

print ("stack is empty")

end if statement

decrement l

return a[l+1]

end **popleft**

popright()
if r is equals to n
print ("stack is empty")end if statement

increment r
return a[r-1]

end **popright**

isEmptyleft()
if l is equals to 0
return true
else
return false
end if statement

end **isEmptyleft**

isEmptyright()
if r is equals to n
return true
else
return false
end if statement

end **isEmptyright**

isFull()
if l + r is equals to n
return true
else
return false
end if statement

end **isFull**

c)

The Big-O for these methods is O (1) since we keep track of the pointer from left and right, we never need to go through the whole array.

Hence, the Big – O for these methods is O (1).

d) The Big-Omega for these methods is Ω (1), since we can’t do any better than O (1).

3. i)
   $f(n) = \log^3 n$

   $g(n) = \sqrt{n} \log n.$

   $f(n) \le g(n)$

   $\log^3 n \le \sqrt{n} \log n.$ Take out logn from both sides

   $\log^2 n \le \sqrt{n}$ If we take n = 4,

   $0.326 \le 2$ This statement is true

   Hence this definition matches the Big-O one, and $f(n) = O(\sqrt{n} \log n.)$

   ii)
   $f(n) = n\sqrt{n} + \log n;$
   $g(n) = \log n^4.$

   $f(n) \le g(n)$

   $n\sqrt{n} + \log n \le \log n^4$ Assume n= 2 and c =1

   $2(1.414) + 1 \le 1$

   $3.838 \le 1$ This statement is false.

   This means than $f(n) \ge g(n)$, which is the definition of Big-Omega, and $f(n) = \Omega(\log n^4)$.

iii)
$f(n) = 2n;$
$g(n) = \log^2 n.$

$f(n) \le g(n)$

$2n \le \log^2 n$ No matter what value we choose for n, 2n (f(n)) will still be more significant and will always increase faster than $\log^2 n$.

This means that f(n) ≥ g(n), which is the definition of Big-Omega, and f(n)= Ω ($\log^2$ n).

v)

$f(n) = \sqrt{n}$;
$g(n) = 2^{\sqrt{\log n}}$

f(n) ≤ g(n)

$\sqrt{n} \leq 2^{\sqrt{\log n}}$ If we consider g(n) as a function of the form of 2^n, we can say that g(n) is an exponential term. We know for complexity that exponential has the highest growth of any function.

Hence, here g(n) ≥ f(n), which is the definition of Big-O, and f(n) = O ($2^{\sqrt{\log n}}$).

vi)

f(n) = 2n;

g(n) = n^n.

f(n) ≤ g(n)

$2^n \leq n^n$ While both of them are exponential functions, the most significant term here is n.

8 ≤ 27 If we use n= 3 for example, we can already see the huge difference for g(n).

Hence, f(n) ≤ g(n), which is the definition of Big-O, and f(n) =O (n^n).

vii)

vi) f(n) = 50;

g(n) = log 60.

f(n) ≤ g(n)

50 ≤ log (60)

50 ≤ 5.906 These are always constant

Since both of them are constant, this is the definition of Big-Theta, and f(n) = θ (log 60).

4.
a) Function findDuplicate()

Create an arr[] a1 with all the values that you want to add in the array

Create a stack with integer type st which is empty

Create an int number = 0,

for (int i =0 ⟵ a1.length, i++)

if st.search(a1[i]) is equal to -1

st.push(a1[i])

number++

end of for loop

Create an arr[] a2 , with the size of number

for(int i = number -1 ⟵ >= 0, i--)

a2[i] = st.pop()

end for loop

return a2

print(a1)

print(a2)

b)

The Big-O for this notation would be O (n).

Because for both loops, it would have to loop through the length of the array (which is n).

If we talk just about inserting into the stack, this would be O (1)

c)

The Big-Omega for this notation would also be Ω (n)

Even if all values are repeated, there would still need to be a loop that goes through the whole to figure out if the values are the same or not.

Big-Omega would be Ω (1) here If there was just one element in the array. Inserting in the stack would also give you Ω (1).

d)

The max space it can use is the whole array, hence the space complexity of the solution is O (n).