CSE 321 - Homework 2

Due: 13.11.22

Q1)

a: $T(n) = 2 \cdot T(n/4) + \sqrt{n \log n}$

$a = 2$, $b = 4$ $f(n) = \Theta((n \log n)^{1/2})$ $k = 1/2$ $p = 1/2$

$\log_b a = 1/2$ $\log_b a = k$ they case 2, $p > -1$

$T(n) = \Theta(n^{1/2} \log^{3/2} n)$ //

b: $T(n) = 9 \cdot T(n/3) + 5n^2$

$a = 9$, $b = 3$ $k = 2$ $p = 0$ $f(n) = \Theta(5n^2)$

$\log_b a = k$ (case 2), $p = 0$ $(p > -1)$ they

$T(n) = \Theta(n^2 \log n)$ //

c: $T(n) = 1/2\, T(n/2) + n$

$a = 1/2$ $b = 2$ $k = 1$ $p = 0$

can not solveable with master theorem because $a = 1/2$, a must be ≥ 1 //

E-5 Karayolu Üzeri Hasanpaşa Mah.
Lavanta Sok. No:24 D:3 Kadıköy - İSTANBUL

T: +90 216 464 67 72 pbx
F: +90 216 464 67 75

www.antinsaat.com
info@antinsaat.com

d: $T(n) = 5T(n/2) + \log n$

$a = 5,\ b = 2 \quad k = 0,\ p = 1 \quad f(n) = \log n$

$\log_b a = 2 \quad k = 0 \quad \log_b a > k$, (case 1)

$T(n) = \Theta(n^{\log_2 5})$ //

e: $T(n) = 4^n \cdot T(n/5) + 1$

$a = 4^n,\ b = 5 \quad k = 0,\ p = 0$

can not solveable with master theorem, because a should be $a \geq 1$ constant. //

f: $T(n) = 7T(n/4) + n\log n$

$a = 7,\ b = 4 \quad k = 1 \quad p = 1 \quad \log_4 7 > k$ then,

$T(n) = \Theta(n^{\log_4 7})$ //

g: $T(n) = 2 \cdot T(n/3) + 1/n$

$a = 2,\ b = 3 \quad f(n) = 1/n \quad k = -1 \quad p = 0$

$\log_b a = \log_3 2,\ k = -1 \quad \log_b a > k$ then

$T(n) = \Theta(n^{\log_3 2})$ //

h: $T(n) = 2/5 \cdot T(n/5) + n^5$

$a = 2/5,\ b = 5,\ k = 5,\ p = 0$

can not solveable by master theorem.
a must be ≥ 1. //

3

Q2) A = { 3, 6, 2, 1, 4, 5 } - insertion sort

- step 1: first elemnt is assumed as sorted. take second elemnt as key.

step 2: compare key with 1. elemnt.
if key is greater than 1. element it should be placed the behind of 1. elemnt.

{ 3, 6, 2, 1, 4, 5 } key = 6.

- step 3: key = 2. compare the key with the elemnts left of it. one by one. leave on the place when it is smaller than the elemnt behind of it.

{ 3, **2**, 6, 1, 4, 5 } → { **2**, 3, 6, 1, 4, 5 }

- step 4: key is 1.

{ 2 3. **1**, 6, 4, 5 } → { 2, **1**, 3, 6, 4, 5 } → { **1**, 2, 3, 6, 4, 5 }

- step 5: key = 4.

{ 1, 2, 3, **4**, 6, 5 }

- step 6: key = 5.

{ 1, 2, 3, 4, **5**, 6 } ← updated version of arry

E-5 Karayolu Üzeri Hasanpaşa Mah.
Lavanta Sok. No:24 D:3 Kadıköy - İSTANBUL

T: +90 216 464 67 72 pbx
F: +90 216 464 67 75

www.antinsaat.com
info@antinsaat.com

(4)

Q3)

a: worst-case time complexity of operations on both array- linked-list

i. accessing the first element;

array: O(1) : accessing the first element is happen by giving the index of it on arrays. it takes O(1).

linked-list: O(1) : first node includes the date we are trying to access. it takes O(1).

ii. accessing the last element:

array: O(1). it is similar as accessing the first element. Indexed elements accessed directly on arrays.

linked-list : O(n) : to access last element, from start to end of list, each element should be traversed it takes O(n).

iii. accessing any element in the middle:

array: O(1). same with first two operations.

linked-list: to access $n/2^{st}$ element takes $O(n/2)$ constant multipliers can be ignored as here $(O(\frac{1}{2}n))$ it is O(n).

iv. adding a new element at the beginning.

array: O(n). it requires to shift all the existing elements by one position

linked-list: O(1) we have the first node's address. so we directly can link the new element at the beginning of linked list

v. adding a new element at the end

array: either O(1) or O(n). if there is room left, elemnt can be added at the end of array in constant time, otherwise it takes O(n). → it is worst-case actually.

linkedlist: for worst-case it's assumed that we don't have the last node's address so the linkked-list should be traversed till the end (O(n)) and at the end, add the element (O(1)) → O(n). O(1) → O(n).

vi. adding a new element in the middle,

array: O(n): array should shift to the end from the middle one by one to insert the elemnt. O(n/2) → O(n).

linked-list: O(n): getting to the middle node takes O(n/2) = O(n).
insertıy: O(1). but it is a compact operation, so O(n). O(1) = O(n).

vii. deleting the first element

array: O(n): find the elemnt to delete takes O(1). but to delete the element, all the remaining elements should shift to start. it takes O(n). O(1). O(n) = O(n).

linked-list: O(1): just unlinking the first node from the remaing of the linked-list. it takes O(1). not need to be shifted.

viii. deleting the last element

array: O(1). indexing the last elemnt and remoay it is O(1). no need to shift.

linked-list. to get last element first, it should be iterated till the end it is O(n). delety = O(1) O(n) x O(1). = O(n).

6

ix. deleting an element in the middle

array: O(n) accesing the element = O(1).
shifting the last half to the right = O(n/2) ≈ O(n)] O(n)

linked-list: O(n) : accessing : O(n/2), deleting the element and link the nodes next of it to each other = O(1)] O(n)

b: Space Requirements

		Array	Linked-List
ACCESS	first	O(1)	O(1)
	middle		
	last		
ADD	first	to copy another array O(n)	O(1)
	middle		
	last		
DELETE	first	to copy another array O(n).	O(1)
	middle		
	last		

7

Q4: n-sized BT to BST (pseudo-code)

```
function store-in-order (root, inorder) {
    If root is null then return.
    Store the tree inorder traversal
    First store the left subtree.
    Copy the data of root
    Then store the right subtree.
}

function count_nodes (root) {
    if root is null then return.
    if root is not null, then
    return count_nodes(root->left) + count_nodes(right) +1
}

function copy-arrey-to-BT (array, root){
    if root is null then return.
    // This function helps us to copies the content of
  sorted arrey to Binary tree
    update the left subtree as doing copy-arrey-to-BT(arrey,
  root->left)
    update root's data. to first elemet of arrey. then
    delete from array.
    then update the right subtree just like left side.
}

function BT-to-BST (root) {
    if root is null then return.
    n = count-nodes (root)
    create an empty array arr.
    store_inorder (root, arr)
    sort the arrey.
copy_arrey-to_BT (arr, root)
}
```

Qu. cont>

8

- Complexity analysis;

Time complexity. O(nlogn).

This is the complexity of sort on last function. Other operations happen in linear time.

Space complexity. O(n)

Array uses this space while storing the binary tree inorder.

Q5> An integer array $A = \{a_0, a_1, a_2 \dots a_n\}$ and integer x is given.

To solve this problem with O(n) time complexity, we can use hash table. Because hash tables operates searching on O(1) time. (when loop on it, it will give O(n))

Pseudo-code:

```
boolean check ( int A[], int x ) {

    Take Hashtable H of size O(n).

    for (i = 0, to A.size() - 1 ) {

        if A[i] >= x {
            int check_number = A[i] - x }
        else {
            int check_number = x - A[i] }

        if H.seach(check_number) is true
            H.insert(A[i])
    } return.
}
```

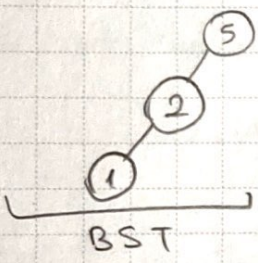
Q6> True - False

9

a> Shape of BST (full, balanced, etc.) depends on the insertion order.

True Because the first node added to BST is the root and cannot be changed or swapped after insertion. Ordering matters in BST for its shape.

b> The time complexity of accessing an element of a BST might be linear in some cases.



True. To access 1 on this BST, it has to be traversed on all elements (in order 5-2-1). Therefore to access 1, which is worst-case, it is O(n) - linear.

c> Finding an array's max or min element can be done in constant time

False. If we lucky enough, we can pick one of the elements on O(1) time and it is the larger / smaller element, yes. But when it comes to proving, no. Sorting an array can not happen in O(1). Without sorting can not find min or max, so. False.

d> **False.** Binary search takes O(log n) if and only if linked list is sorted. If it is not which is the worst case, it takes O(n).

e> Reversely-sorted arrays are worst case for insertion sort, because each time all elements should change their place. It takes $O(n^2)$. not O(n). **False**