1-)

Emre YILMAZ
1901042606

* Advanced Master Theorem that is used in this question :

→ $T(n) = a.T(n/b) + \Theta(n^k \log^p n)$

- n = Size of the problem
- a = number of subproblems, $a \geq 1$
- $b > 1$, $k \geq 0$, p is real number

If $a > b^k$, then $T(n) = \Theta(n^{\log_b^a})$

If $a = b^k$ then

if $p \geq -1$, then $T(n) = \Theta(n^{\log_b^a} . \log^{p+1} n)$

if $p = -1$, then $T(n) = \Theta(n^{\log_b^a} \log\log n)$

if $p < -1$, then $T(n) = \Theta(n^{\log_b^a})$

If $a < b^k$, then

if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$

if $p < 0$, then $T(n) = \Theta(n^k)$

Now, let's go to the questions.

a-) $T(n) = 2.T(\frac{n}{4}) + \sqrt{n\log n}$

$a = 2$, $b = 4$, $k = 1/2$, $p = 1$. There is no problem

$a = b^k$ $(2 = 4^{1/2})$ and $p \geq -1$, (case 2.1)

then $T(n) = \Theta(\sqrt{n} . \log^2 n)$

b-) $T(n) = 9.T(\frac{n}{3}) + 5n^2$

$a = 9$, $b = 3$, $k = 2$, $p = 0$. There is no problem

$a = b^k$ $(9 = 3^2)$, and $p \geq -1$ (case 2.1)

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

c-) $T(n) = \frac{1}{2} \cdot T(\frac{n}{2}) + n$

$a = \frac{1}{2}$, a must greater or equal than 1.

So, it cannot be solved.

d-) $T(n) = 5. T(\frac{n}{2}) + \log n$

$a=5$, $b=2$, $k=0$, $p=1$

$a > b^k$ $(5 > 1)$ (case 1)

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

e-) $T(n) = 4^n. T(\frac{n}{5}) + 1$

$a = 4^n$, it is not a proper form. It must be constant.

So, it cannot be solved.

f-) $T(n) = 7. T(\frac{n}{4}) + n \log n$

$a=7$, $b=4$, $k=1$, $p=1$

$a > b^k$ $(7 > 4)$ (case 1)

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

g-) $T(n) = 2. T(\frac{n}{3}) + 1/n$

In this question , $k = -1$. It is not in a proper form. It must be greater or equal than 0.

So, it cannot be solved.

h-) $T(n) = \frac{2}{5} \cdot T(\frac{n}{5}) + n^5$

$a = 1/2$ and it is not in a proper form.

a must be greater or equal than 1.

So, it cannot be solved.

CamScanner ile tarandı

2-) Step-1 (Place "$\underline{6}$" to proper position) (↔ sign represents comparison)

- 3, $\underline{6}$, 2, 1, 4, 5 $6 > 3$, no change
- 3, $\underline{6}$, 2, 1, 4, 5   ✓

Step-2 (Place "$\underline{2}$" to proper position)

- 3, 6, $\underline{2}$, 1, 4, 5 $2 < 6$, change
- 3, $\underline{2}$, 6, 1, 4, 5 $2 < 3$, change
- $\underline{2}$, 3, 6, 1, 4, 5   ✓

Step-3 (Place "1" to proper position

- 2, 3, 6, $\underline{1}$, 4, 5 $1 < 6$, change
- 2, 3, $\underline{1}$, 6, 4, 5 $1 < 3$, change
- 2, $\underline{1}$, 3, 6, 4, 5 $1 < 2$, change
- $\underline{1}$, 2, 3, 6, 4, 5   ✓

Step-4 (Place "4" to proper position

- 1, 2, 3, 6, $\underline{4}$ 5, $4 < 6$, change
- 1, 2, 3, $\underline{4}$, 6, 5 $4 > 3$, no change
- 1, 2, 3, $\underline{4}$, 6, 5   ✓

Step-5 (Place "5" to proper position

- 1, 2, 3, 4, 6, $\underline{5}$ $5 < 6$, change
- 1, 2, 3, 4, 5, 6 $5 > 4$, no change
- 1, 2, 3, 4, 5, 6   ✓

3.)

a.) I-) Accessing first element

* Array[0] can access directly the first element and it is $\Theta(1)$

* A proper double linked list must have head and tail references. So, list.getFirst() method directly returns head.data. Hence, it is $\Theta(1)$

II-) Accessing the last element

* Array[n-1] can access directly the last element and it is $\Theta(1)$.

* A proper double linked list must have head and tail references So, list.getLast() method directly returns tail.data. Hence, it is $\Theta(1)$

III-) Accessing any element in middles

* Array[x] will directly returns the x^{th} element of list in $\Theta(1)$ time.

* In a linked list, going a specific element in the list must include iteration operation. We have to go forward until we reach the target element. So, it is $\Theta(n)$

IV-) Adding a new element at the beginning

* Assuming there are only n spaces for n element array, to add an element to beginning, we have to create a new array, put the new element to beginning, then we will copy the old array's elements to new one. So, we have to traverse all array. Complexity is $\Theta(n)$

* In a linked list implementation, adding a new element to beginning is simple. Create a node, make it head and fix the references. It will no need any iteration. So, complexity is $\Theta(1)$

CamScanner ile tarandı

V-) Adding a new element to the end

* Assuming there are only n spaces for n element array, to add an new element to end we have to create a new array, copy the old one to new. (This operation make us need to traverse all array). Then, we will add the new element to new array[n]. The complexity is $\Theta(n)$

* In a linked list implementation, adding a new element to end is too simple. Create a node make the connection between this node and tail, then make this new node tail. There is no need to iterate anything. We do not need a loop. Complexity is $\Theta(1)$

Vi-) Adding a new element to middles

* Assuming there are only n spaces for n element,

Similar to addi an element to beginning or end, there is a need to copy the array's element to a new array. there is a need to traverse whole array. The complexity is $\Theta(n)$.

* In the linked list, going to specific position that we add the element (in the worst case it is $(n-1)^{th}$ element) needs traversing the linked list. We must go forward until we reach the target. So, the complexity is $= \Theta(n)$

VII-) Deleting the first element

* When we want to remove the first element of the array, we will need shifting operation. We will create a new array, we put the exact elements of old array to new one until we reach the target element. We will pass the target, then continue to copying. So, we must traverse the whole array. Complexity is $\Theta(n)$

* When we want to remove the first element of linked list, we only rearrange the references. Make the head->next, head. That's it. There is no need to iterate anything. Complexity is $\Theta(1)$

VIII -) Deleting the last element

* When we want this, we need create a new array again, then copy old array to new one except the last element. So, we will traverse whole array. Complexity is $\Theta(n)$

* When we want this in a linked list, we only need rearrange the references. We will change the tail. We will NOT iterate anything. So, complexity is $\Theta(1)$

IX -) Deleting any element in the middle

* When we want this, we need create a new array. We put the exact elements of old array to new one until we reach the target element. We will pass the target, Then continue to copying. So, we must traverse the whole array. Complexity is $\Theta(n)$

* When we want to do this in a linked list, firstly we must reach the target element, This operation needs traversing the list. (In the worst case target is $(n-1)^{th}$ element, After we found the element, we will rearrange the references. So, complexity is $\Theta(n)$

b-) Linked list needs store data and store pointer while array needs only store data. So, linked list needs more memory. On the other hand, in array implementations, for better time complexity, array has more space than required. It may lead wastage of memory space.

CamScanner ile tarandı

4-)

```
BTtoArray ( root, arraylist )
{
    if  root  is  null
        return

    BTtoArray (root.left, arraylist )
    arraylist . append ( root. data )    // Adds to end of list
    BTtoArray (root. right , arraylist )
}

ArrayToBST ( root, arraylist )
{
    if  root  is  null
        return;

    ArrayToBST ( root.left,  arraylist )
    root. data  =  arraylist . get (0)
    arraylist , remove (0)
    ArrayToBST ( root. right,  arraylist )
}

main ( BinaryTree )
{
    ArrayList  arr  =  new  ArrayList()  // It is explained in the next page

    BTtoArray ( BinaryTree. head,  arr )  → $\Theta(n)$
    arr  =  Merge Sort (arr)  →  $\Theta(n \log n)$
}   ArrayToBST ( BinaryTree. head,  arr )  →  $\Theta(n^2)$
```

Go to next page ...

EXPLANATION IS IN THE NEXT PAGE !

ADVISED, FIRSTLY READ IT...

REGARDS.

CamScanner ile tarandı

4n) Continue

* The data type ArrayList that you see in the algorithm is thought with the logic of ArrayList in Java, or equivalent to List in Python. For more clear understanding, ArrayList is defined as Java style in the algorithm.

While append function (append) to the list amortised constant time, removing an element (remove function) works linear due to shifting. (Append adds an element to end of the list, so there is no need to shift)

* In the algorithm, firstly the tree is stored to an ArrayList by inorder traversal

Secondly, we sort the array using merge sort in ascending order. It is $\Theta(n\log n)$

Then, we make inorder traversal and put the appropriate elements to the nodes.

* The function BTtoArray has a recurrence $T(n) = \overset{\text{recursive calls}}{2T(\frac{n}{2})} + \overset{\text{append function}}{1}$.

It can be easily seen it is linear both worst and best cases

$\Theta(n)$

The function ArrayToBST has a recurrence $T(n) = \overset{\text{recursive calls}}{2T(\frac{n}{2})} + \overset{\text{remove function}}{n}$

It can be seen easily it is quadratic in both worst and best cases

$\Theta(n^2)$

The merge sort is always $\Theta(n\log n)$

So, the whole algorithm best case => $\Theta(n^2)$

worst case => $\Theta(n^2)$

average case => $\Theta(n^2)$

CamScanner ile tarandı

5-)

```
findMax( array[], size )
{
    max = array[0]
    for i=0 to size
        if array[i] > max
            max = array[i]
        endif
    endfor

    return max
}
```

T= a·b

```
findPair( array[], size, target )
{
    max = findMax(array, size)
int count[max]   // Declare a new array that store the occurences of numbers in array
                 // It is assumed that all elements of this array are 0 by
                    default.

  for i=0 to size
      count[array[i]] = 1
  endfor

  for i=0 to size
      addSize = target + array[i]
      if ( count[addSize] == 1 )
        Print("The pair is %d and %d", addSize, array[i] )
      endif
  endfor
```

Continue to next page

CamScanner ile tarandı

5-) contiue

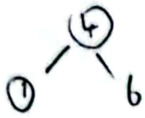
* The count array in the algorithm can be thought as a hash map. The logic behind it is the same. Key is an element in array, Value is occurence number of that element. We traverse the array and we check whether the needed element is in the array using that count array (hash map). If there is, it means that we find a pair.

* We find the max element using a loop → $\Theta(n)$
* We traverse the array to find occurances → $\Theta(n)$
* We traverse the array one more time to find pairs → $\Theta(n)$

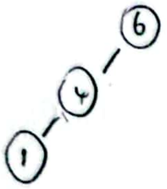
→ $\Theta(n) + \Theta(n) + \Theta(n) = \Theta(n)$ ✓

6-)

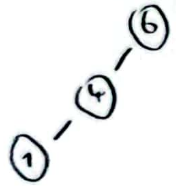
a-) This is a true statement. For example, if we insert elements 4, 1, 6 in order, the tree is shaped like this :



If we insert 6, 4, 1 in order, the tree is :



b-) This is true. If we insert 6, 4, 1 respectively, the tree is look like exactly a linked list. So, complexity may be linear.



c-) It is impossible unless we do NOT store min and max elements. To find min and max elements, we have to look and check al of the array. So, it is linear.

d-) It is impossible use binary search on a linked list. Actually, it does not make sense. Any random access is a prerequisite for binary search. But, there is no random access in linked list. List must be traversed to go a specific element. It is going to be O(n)

e-) There is n-1 step in insertion sort. In the worst case, there will be i comparison in each step i

$$\sum_{i=1}^{n-1} i = \frac{n.(n+1)}{2} = \Theta(n^2)$$

So, the statement is false. It must be $\Theta(n^2)$

CamScanner ile tarandı