GTU Department of Computer Engineering
CSE 321 – Fall 2022
Homework #2

Hasan Mutlu
1801042673

1.

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

$f(n) = \sqrt{n \cdot \log n} \notin \Theta(n^d)$ -> It cannot be solved using Master Theorem

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

$a = 9 \geq 1$ ✓

$b = 3 \geq 2$ ✓

$f(n) = 5n^2 \in \Theta(n^2)$ ✓

$9 = 3^2 \iff a = b^d$ $d = 2$

$\Rightarrow T(n) \in \Theta(n^2 \cdot \log n)$ //

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

$a = \frac{1}{2} < 1 \Rightarrow$ It cannot be solved

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

$f(n) = \log n \notin \Theta(n^d) \Rightarrow$ It cannot be solved

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

$a = 4^n$ is not a constant $\Rightarrow$ It cannot be solved

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

$f(n) = n \log n \notin \Theta(n^d) \Rightarrow$ It cannot be solved

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

$f(n) = \frac{1}{n} \in \Theta(n^{-1})$, $d < 0 \Rightarrow$ It cannot be solved

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

$a = \frac{2}{5} < 1 \Rightarrow$ It cannot be solved.

2. Insertion Sort

A = {3, 6, 2, 1, 4, 5}

➢ Key = 6	{3, **6**, 2, 1, 4, 5}
➢ Compare 6 with 3	{3, **6**, 2, 1, 4, 5}
➢ Stay as is	{3, **6**, 2, 1, 4, 5}
➢ Key = 2	{3, 6, **2**, 1, 4, 5}
➢ Compare 2 with 6	{3, 6, **2**, 1, 4, 5}
➢ Swap	{3, **2**, 6, 1, 4, 5}
➢ Compare 2 with 3	{3, **2**, 6, 1, 4, 5}
➢ Swap	{**2**, 3, 6, 1, 4, 5}
➢ Key = 1	{2, 3, 6, **1**, 4, 5}
➢ Compare 1 with 6	{2, 3, 6, **1**, 4, 5}
➢ Swap	{2, 3, **1**, 6, 4, 5}
➢ Compare 1 with 3	{2, 3, **1**, 6, 4, 5}
➢ Swap	{2, **1**, 3, 6, 4, 5}
➢ Compare 1 with 2	{2, **1**, 3, 6, 4, 5}
➢ Swap	{**1**, 2, 3, 6, 4, 5}
➢ Key = 4	{1, 2, 3, 6, **4**, 5}
➢ Compare 4 with 6	{1, 2, 3, 6, **4**, 5}
➢ Swap	{1, 2, 3, **4**, 6, 5}
➢ Compare 4 with 3	{1, 2, 3, **4**, 6, 5}
➢ Stay as is	{1, 2, 3, **4**, 6, 5}
➢ Key = 5	{1, 2, 3, 4, 6, **5**}
➢ Compare 5 with 6	{1, 2, 3, 4, 6, **5**}
➢ Swap	{1, 2, 3, 4, **5**, 6}
➢ Compare 5 with 4	{1, 2, 3, 4, **5**, 6}
➢ Stay as is	{1, 2, 3, 4, **5**, 6}

Note: During the swap operations, the values aren't actually swapped. The value on the left is copied to right. The term swap is used to show all elements of the array.

3.

a.) Time Complexity Analysis

i. Accessing the first element:

- Array: $\Theta(1)$
- You can access any element in the Array by simple pointer addition.

- Linked List: $\Theta(1)$
- There is a pointer to the first element in Linked List.

ii. Accessing the last element

- Array: $\Theta(1)$
- You can access any element in the Array by simple pointer addition.

- Linked List: $\Theta(n)$
- You need to traverse entire list in order to reach to the last element in Linked List. Total number of n elements must be traversed.

iii. Accessing any element in the middle.

- Array: $\Theta(1)$
- You can access any element in the Array by simple pointer addition.

- Linked List: $\Theta(n)$
- You need to traverse the first k elements of a Linked List to reach to kth element in the middle.
- Suppose that k = n-m, where m is the number of remaining elements in the list,
- $k = n - m \in \Theta(n)$

iv. Adding a new element at the beginning.

- Array: $\Theta(n)$
- In order to add a new element at the beginning of Array, you first need to shift each element by one. Total number of n shift operations is needed.

- Linked List: $\Theta(1)$
- You can create a new link from the new head node to the former head node and assign new node as the head of Linked List.

v. Adding a new element at the end.

- Array: $B(n) = \Theta(1)$, $W(n) = \Theta(n)$
- In best case, if the array is not full, you can add a new element at the end of an Array.
- However, in the worst case, if the array is full, you need to reallocate memory. During reallocation, all elements of the array must be copied to newly allocated memory. Total number of n copy operations is needed.

- Linked List: $\Theta(n)$
- You need to traverse entire list to reach the last element and create a new node connection to the new element. Total number of n elements must be traversed.

vi. Adding a new element in the middle.

- Array: $\Theta(n)$
- In order to open a space to the new element, all the elements after the index must be shifted ahead. Assume the index is k, n-k number of elements must be shifted. (Reallocation might be required as well.)
- $n-k \in \Theta(n)$.

- Linked List: Θ(n).
- You need to traverse to the required index' location. Then you need to just create a new node connection. Total number of n elements must be traversed.

vii. Deleting the first element.

- Array: Θ(n)
- In order to delete the first element of the array, all of the remainig elements must be shifted to left by one. Total number of n-1 elements must be shifted.
- n-1 ∈ Θ(n)

- Linked List: Θ(1)
- You just need to traverse to the second element and assign it as the head node.

viii. Deleting the last element.

- Array: Θ(1)
- You directly access to last element and mark it as deleted. (Or simply decrease the size variable.)

- Linked List: Θ(n)
- You need to traverse to the penultimate element and break the node connection to the last element. Total number of n-1 elements must be traversed.
- n-1 ∈ Θ(n)

ix. Deleting any element in the middle.

- Array: Θ(n)
- You need to shift all the remaining elements to the left by one in order to delete an element. To delete kth element, you need to shift n-k elements.
- n-k ∈ Θ(n)

- Linked List: Θ(n)
- You need to traverse to the index of the element to be deleted and break the node connection to it. Suppose that k = n-m, where m is the number of remaining elements in the list,
- k = n - m ∈ Θ(n)

b.) Space Requirements

- Array requires n*sizeof(data) amount of space.
- Linked List requires n*(sizeof(data) + sizeof(pointer)) amount of space.

4.) Binary Tree to BST

```
def BTtoArray(root, array):
    if(root = null):
        return
    else:
        BTtoArray(root.left, array)
        array.append(root.data)
        BTtoArray(root.right, array)

def ArraytoBST(root, array):
    if(root = null):
        return
    else:
        ArraytoBST(root.left, array)
        root.data = array[0]
        array.remove(array[0])
        ArraytoBST(root.right, array)

def BTtoBST(binary_tree):
    n = binary_tree.size()
    array = new Array(n)
    BTtoArray(binary_tree, array)  # Recursively adds all elements in the binary_tree to
the array
    array.sort()    # Assume Merge Sort is used.
    ArraytoBST(binary_tree, array) # Recursively assigns all elements in the array to the
binary tree in order.
```

The main method is BTtoBST. It takes a Binary Tree as argument.

1) It first calculates and assigns the size of the binary_tree. Logically, .size() function takes Θ(n) time as it traverses every element of the array.
2) Then, it creates a new array with the size of n. Its running time is ambigious so it is neglected in this analysis.
3) After that, it calls BTtoArray method to copy every element of the tree into an array. This method's base case is when root is null. So, it traverses all the nodes, which means it takes Θ(n) time as well.
4) After array is created, it is sorted. Its running time depends on the sorting algorithm we have used.
5) Finally, the elements in the sorted array are assigned back to the binary tree in order. Similarly to BTtoArray method, ArraytoBST method also traverses all nodes, which means it takes Θ(n) time as well.

When all of these steps are taken into account, it is observed that the determining step is the fourth step, which is sorting the array. So, the best-case, worst-case, and average-case time complexities are determined solely by the sorting algorithm which is used.

Suppose that we used Insertion Sort;

- Best case: B(n) ∈ Θ(n)
- Worst case: W(n) ∈ Θ(n^2)
- Average case: A(n) ∈ Θ(nlogn)

5.) Find Pair

```
def findPair(array, n, dif):
    dictionary = new Array(n)
    for i in range(n):
        if(dif + array[i] in dictionary):
            return (array[i], dif + array[i])
        else
            dictionary.append(array[i])
    return -1
```

The method takes an array, its size and the searched difference as arguments.

- It creates a new array of size n named dictionary. This dictionary keeps track of the elements that are past.
- In a for loop.
- In a for loop, it is checked whether there is an element in the dictionary that its difference with current array element is the searched difference. If the current element doesn't have a pair, it is added to the dictionary as well.
- If such pair is found, it is returned right away.

Assuming that the "in" operation inside the if condition takes constant time, this algorithm has $O(n)$ time complexity at worst case.

6.) **True – False**

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

True

The BST doesn't check how many elements are there on the each side of the current node. It just makes comparison between the data and finds the optimal place to insert the new element.

For example, if the elements are inserted to a BST in a sorted manner, all elements would be assigned to the leftest or rightest leaves of the BST. In that case, BST would have a linear shape.

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

True

If the BST has a linear shape due to the example above, It wouldn't be structurally any different than a linked list so accessing an element would take linear time.

c.) Finding an array's maximum or minimum element can be done in constant time.

Depends on the Array

If the array is sorted, it can be done in constant time. Otherwise, it would take linear time as all the elements in the array needs to be checked.

d.) The worst-case time complexity of binary search on a linked list is $O(log(n))$ where n is the length of the list.

False

In order to access the element on the middle, the list must be traversed from the head and tail nodes at the same rate. That would give a linear time penalty in each iteration of dividing the list by half. So that would take O(nlogn) time.

e.) Worst-case time complexity of the insertion sort algorithm is $O(n)$ if the given array is reversely sorted.

False

If the array is reversely sorted, each element L[i] needs to be compared by all the elements that come before it. That would take total number of 1 + 2 + 3 + …. + n comparisons.

1 + 2 + 3 + …. + n = n*(n+1)/2 ∈ Θ(n^2)