Abdurrahman Bulut - 1901042258

CSE 321 - Homework 2

Master Theorem

1) $\therefore T(n) = aT(n/b) + f(n)$, $a \geq 1$, $b > 1$ and $f(n)$ is asymptotic +

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) \rightarrow \text{case 1} & , f(n) = O(n^{\log_b a - \varepsilon}) \\ \Theta(n^{\log_b a} . \log n) \rightarrow \text{case 2} & , f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) \rightarrow \text{case 3} & , f(n) = \Omega(n^{\log_b a + \varepsilon}) \end{cases} \quad \varepsilon > 0, \; c < 1$$

control: $af(n/b) < c.f(n)$ → Regularity condition

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

* $a = 2$
  $b = 4$
  $f(n) = \sqrt{n \log_2 n}$

* $n^{\log_4 2} \Leftrightarrow \sqrt{n \log_2 n}$
  $\sqrt{n} < \sqrt{n \log_2 n}$ (case 3)

Regularity Condition

* we need to test: $a.f(\frac{n}{b}) < c.f(n)$
* $2.\sqrt{\frac{n}{4}.\log_2 \frac{n}{4}} < c.\sqrt{n \log n}$, $c = \frac{1}{2} < 1$

* for $n = 4$, $c = \frac{1}{2} \Rightarrow 2.\sqrt{0} < \frac{1}{2}.\sqrt{4.\log_2 4} \rightarrow 0 < \sqrt{\log_2 4} \rightarrow 0 < \sqrt{2}$, it satisfies for sufficiently large n values,

* Case 3: $T(n) = \Theta(f(n)) \Rightarrow T(n) = \Theta(\sqrt{n \log n})$

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

* $a = 9$
  $b = 3$
  $f(n) = 5n^2$

* $n^{\log_3 9} \Leftrightarrow \cancel{5}n^2$
* $n^2 = n^2$ (case 2)

* Case 2: $T(n) = \Theta(n^{\log_b a}.\log_2 n) \Rightarrow T(n) = \Theta(n^2 \log n)$

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

* $a = \frac{1}{2}$ → Since $a = \frac{1}{2} < 1$, Master theorem cannot apply on it.
  $b = 2$
  $f(n) = n$

d) $T(n) = 5 \cdot T(\frac{n}{2}) + \log_2 n$

* $a = 5$ * $n^{\log_2 5} \Leftrightarrow \log_2 n$
 $b = 2$ $n^{\log_2 5} > \log n$ (case 1)
 $f(n) = \log n$

* Case 1: $T(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_2 5})$

---

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

* $a = 4^n$
 $b = 5$
 $f(n) = 1$

* This algorithm cannot be solved by using master theorm. Because, "a" is not a constant and this, makes it comparable.

---

f) $T(n) = 7 \cdot T(\frac{n}{4}) + n\log_2 n$

* $a = 7$ * $n^{\log_4 7} \Leftrightarrow n\log_2 n$
 $b = 4$ $n^{\log_4 7} > n\log_2 n$ (Case 1)
 $f(n) = n\log_2 n$

* Case 1: $T(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_4 7})$

---

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

* $a = 2$
 $b = 3$
 $f(n) = \frac{1}{n}$

* This algorithm cannot be solved by using master theorem. Because, $f(n)$ which is combination time, It should be (asymptotic +)

---

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

* $a = \frac{2}{5}$ * This algorithm cannot be solved by using master theorem.
 Since, $a \ngtr 1$
 $b = 5$
 $f(n) = n^5$

2) Insertion Sort - Ascending Order

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

* First Pass: * At the beginning, first two elements will be compared. Since, $3 < 6$, it does not need to swap. The element 3 is stored in a sorted Sub-array.

Current:

3	6	2	1	4	5

* Second Pass: * The next two elements will be compared. They are 6 and 2. Since $6 > 2$, They need to be swap.

Current:

3	2	6	1	4	5

* Then, when we look at previous elements, 2 and 3 are not sorted. So, they should be swapped.

Current:

2	3	6	1	4	5

* For now, 2 and 3 are in the sorted Sub-array.

* Third Pass: * At this time, 6 and 1 will be compared. They are not at their correct place. So, they need to be swapped.

Current:

2	3	1	6	4	5

* 3 and 1 are not in ascending order, so they need to swap.

Current:

2	1	3	6	4	5

* 2 and 1 are not in ascending order, so they need to swap

Current:

1	2	3	6	4	5

* Fourth Pass: * Now, 6 and 4 will be compared. They are not in ascending order. Therefore, they need to swap.

Current:

1	2	3	4	6	5

For Fourth Pass, there is no more swapping, Because 1, 2, 3, 4, are in ascending order. →

2 - Continued)

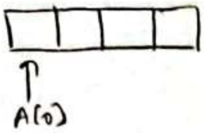
* Fifth Pass: * 6 and 5 will be compared. Since they are not in ascending order, they need to swap.

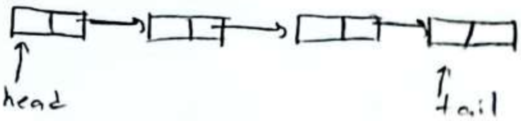
Current: | 1 | 2 | 3 | 4 | 5 | 6 |

* There is no more swapping for 1,2,3,4,5, Because they are already sorted.

Result Array: | 1 | 2 | 3 | 4 | 5 | 6 |

---

3) Array: 

, LinkedList: 

a)

* Linkedlist have 2 pointers which are head and tail. head: start of the linked list. Tail: last node.

1) Accessing the first element:

* Array: O(1) → we can use indexing to access first element. A[0]
* Linkedlist: O(1) → we can use head pointer which points first node. head →

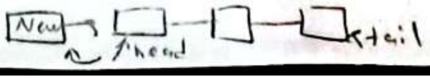
2) Accessing the last element:

* Array: O(1) → Again, we can use indexing, Length -1 gives the last index. A[n-1]
* Linkedlist: O(1) → we can acess to last element with using the tail. Tail pointer points to the last node. tail →

3) Accessing any element in the middle.

* Array: O(1) → we can use indexing. and access with index of element. Single operation. A[middle]
* LinkedList: O(n) → we need to move target point from head point. So it takes n operation.

4) Adding a new element at the begining

* Array: O(n) → Adding element at the beginning takes constant time but after that all elements need to be shifted.
* LinkedList: O(1) → we can create new node and we can assign as head. so, it takes Constant time. 

5) Adding a new element at the end.

* Array: $O(1)$ → we can use indexing. A[last] = new value. It takes single operation.

* Linkedlist: $O(1)$ → we can create a new node and the tail.next will be new node. Finally, new node would be tail node.

6) Adding a new element in the middle.

* Array: $O(n)$ → Adding element takes constant time (single operation) but Shifting is needed. All indexes after middle should be shifted to right by 1 index. $n/2 \to O(n)$

* LinkedList: $O(n)$ → we need to move from head to middle node and create new node at that point. It will take n operation.

7) Deleting the first element.

* Array: $O(n)$ → we need to shift to the left by 1 index after deleting. So, shifting takes n operation.

* Linkedlist: $O(1)$ → we need to make head as next node. Then first node will be seperated from linkedlist.

8) Deleting the last element.

* Array: $O(1)$ → Shifting is not required. we can simply remove by indexing. It takes constant time.

* Linkedlist: $O(n)$ → we need to move from head to tail. we can assign tail to previous node and make last node free. Otherwise, we can lose tail pointer.

9) Deleting any element in the middle

* Array: $O(n)$ → we need to shift to the left side the elements on the right side by 1 index. So it needs $n/2$ operation. $n/2 \to n$

* Linkedlist: $O(n)$ → We need to move from head to middle node to remove it. So it will take $n/2$ operation. $n/2 \to O(n)$

b) The space complexity for arrays is O(n) and it is fixed. It is O(n) for linked list at worst case scenario. In array, elements are stored in contiguous memory location or consecutive manner in the memory. In a linked list, new elements can be stored anywhere in the memory. Array: Static Memory Allocation Linked List: Dynamic Memory Allocation.

---

4) We can perform an in-order traverse of the binary tree and put the nodes in an array to convert it to a binary search tree (BST). This array can then be sorted and used to produce the binary search tree (BST). We first need to sort the array, which takes O(nlogn) time, after doing an in-order traverse of the tree, which takes O(n) time. Therefor, this algorithm has an O(nlogn) time complexity where n is the number of nodes in the tree.

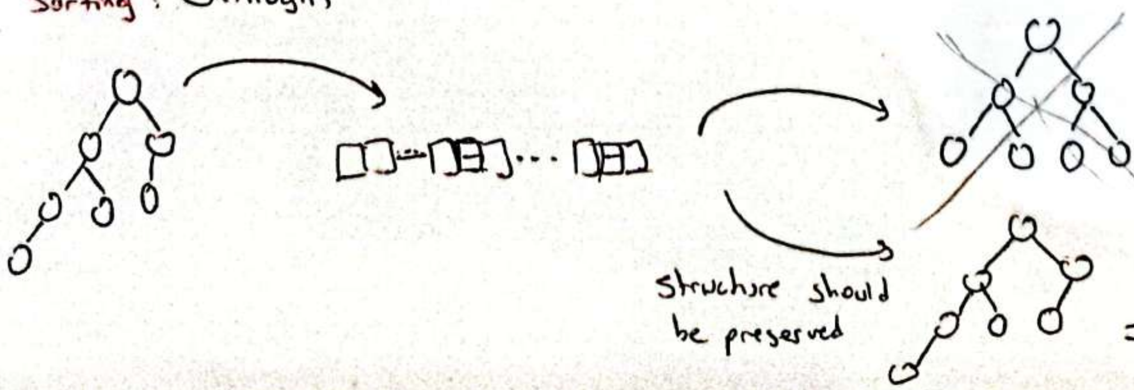
This is due to fact that if the tree is already a BST, all that is required is an O(n) time in order traversal of the tree. Best Case: O(n)

If tree is not a BST, we need to first traverse it in order. which requires O(n) time. After that we must sort the array which requires O(nlogn) time. So, the worst-case time complexity is O(nlogn) for this algorithm.

For the average-case time complexity, we need to sort the array which takes O(nlogn) time, after doing an in-order traverse of the tree which takes O(n) time. That's why the algorithm has an O(nlogn) average-case time complexity.

In-order traversal: O(n)

Sorting: O(nlogn)



⇒ continued

Algorithm BT → BST

Pseudocode

```
* function binaryTreeToBST (root,n) #n = size of the tree
      if (root = None) then
          return root
      end if

      newArr = []      # creating new array to store tree nodes with inorder
                       # traversal

      call moveTreeToArray(root, newArr) # store elements to newArr array O(n)

      call newArr.sort()               # Sort newArr array. O(nlogn)

      call arrToBST (root, newArr)
  end

* function moveTreeToArray(root, newArr)
      if (root= None)
          return root
      end if

      call moveTreeToArray(root.left, newArr) # Move to left to store left sub-tree

      newArr.append (root.data) # store node data on to newArr array.

      call moveTreeToArray(root.right, newArr) # Move to right to store right sub-tree

  end

* function arrToBST ( root, newArr)
      if(root = None)
          return root
      end if
      call arrToBST (root.left, newArr)   # first store the left sub tree
      root.data = newArr[0]   # assign first element in the array and then delete it.
      newArr.pop(0)
      call arrToBST (root.right, newArr)
  end
```

5) $|a_i - a_j| = x \Rightarrow a_i - a_j = x \;,\; a_i - a_j = -x$ # absolute value

```
def findPairs(array, diff):

    dictionary ← {}

    n ← length of the array

  for i from 0 to n:

     If array[i] - diff is in dictionary then

         return (array[i], array[i]-diff)

     else if array[i]+diff is in dictionary then

         return (array[i], array[i]+diff)

     else
         append array[i] to dictionary

     endif

  endfor
  return -1

end
```

<u>Example:</u>

Inputs:

array = {8, 2, 9} , difference = 7

$\boxed{8}\ \boxed{2}\ \boxed{9}$ $\boxed{\text{dictionary} = \{\}\ ,\ \text{diff} = 7}$

$\boxed{8}\ \boxed{2}\ \boxed{9}$

is 8+7 in dictionary?: False
is 8-7 in dictionary?: False } Then add 8 to dictionary

$\boxed{8}\ \boxed{2}\ \boxed{9}$ $\boxed{\text{dictionary} = \{8\}\ ,\ \text{diff} = 7}$

=> Continued

$\boxed{8}$ $\boxed{2}$ $\boxed{9}$

$\uparrow$

dictionary: {8}, diff = 7

is 2-7 in dictionary? : False

is 2+7 in dictionary? : True → Then return (2, 9)

In the best case, the algorithm returns the firstly compared pair. That means the loop will iterate only once. Therefore, the best case complexity is $\Omega(1)$

In the worst case, there might not be such a pair in the array, or the pair might be the last one to be compared. Since we go over the array only once, we will make at most n comparisons and then return. Therefore, the worst time complexity is $O(n)$.

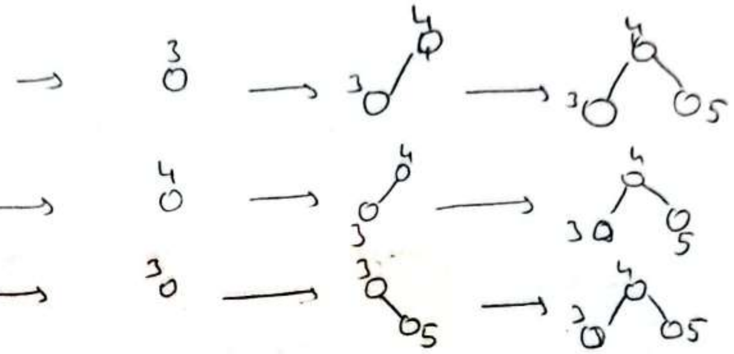
6)

a) False

↳ The shape of a BST (full, balanced, etc.) doesn't depend on the insertion order. Because we will always get a unique BST structure. Only number of elements are matter. If we have same number of elements for any two sets, we will get same BST at each

<u>Example</u> 3, 4, 5

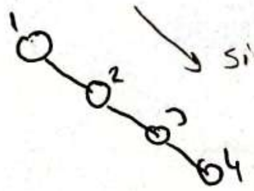
order
3 → 4 → 5
4 → 3 → 5
3 → 5 → 4



b) True

↳ Time complexity will be O(n) for BST if the tree is skewed.

<u>Example:</u>



↳ similar with <u>Linear</u> search.

c) If array is already sorted: YES

If array is not sorted: NO

↳ The simplest method for finding maximum/minimum value among array elements is to store initial element in a temporary variable, loop it while comparing it to other elements, and find the maximum/minimum value between two elements in each pass. Then, while looping, perform a new initialization of this maximum/minimum value to the temporary value.

Time complexity in here: O(n)

If the array is sorted, Getting initial or last element is constant time: O(1)

⇒ Continued

Making a method that can sort an array in less than $O(n)$ may be $O(\log n)$ will result in complexity of $O(\log n)$ for choosing the maximum element from an array, depending on whether it is sorted.

---

d) True

↳ The worst case time complexity of Binary Search is $O(\log n)$. We can use skip-list or create similar structure.

<u>For example</u>

$0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10$

gap between elements $g \leq \log_2 n$

To find an element, just binary search the array and then go to that spot in the linked list, traverse the list until the element is found or greater than the target, return true or false respectively. As we are traversing the list to find the element, every g elements (g is the gap size), we just move the next "pole" (poles are the items in the array) over to the current element. If there is no next pole, add a pole to the end of the array.

---

e) False

↳ Worst case occurs when the array elements are required to be sorted in reverse order.

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1).n}{2} \in \Theta(n^2)$$

<u>Algorithm</u>

```
for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j ≥ 0 and A[j] > v do
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← v
```