CSE 321 - Homework 2

Mert
Gürtimşir
1901042646

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

$a = 2$
$b = 4$
$f(n) = \sqrt{n} \cdot \sqrt{\log n} = \Theta(n^{\log_4 2} \log^{1/2} n) = \Theta(\sqrt{n\log n})$

$k = 1/2$

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

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

$a = 9$
$b = 3$
$f(n) = 5n^2 \in \Theta(n^2) \rightarrow d = 2$
$a = 9 = b^d = 9$

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

c) $T(n) = \frac{1}{2} T(\frac{n}{2}) + n$ $\quad a = \frac{1}{2} < 1$ So this relation cannot be solved by using Master Theorem.

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

$a = 5$
$b = 2$
$f(n) = \log n \in \Theta(\log n) \rightarrow \Theta(n^k \log^p n) \,/\, k = 0,\ p = 1$
$a = 5 > b^k = 1$

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

e) $T(n) = 4^n \cdot T(\frac{n}{5}) + 1$ $\quad a = 4^n \rightarrow$ a is not a constant so this relation cannot be solved by using Master Theorem.

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

$a = 7$
$b = 4$
$f(n) = n\log n \in \Theta(n\log n) \rightarrow \Theta(n^k \log^p n) \,/\, k = 1,\ p = 1$
$a = 7 > b^k = 4$

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

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

$a = 2$
$b = 3$
$f(n) = n^{-1} \in \Theta(n^{-1})$ $\quad d = -1 < 0$ So this relation cannot be solved by using Master Theorem

h) $T(n) = \frac{2}{5} T(\frac{n}{5}) + n^5$ $\quad a = \frac{2}{5} < 1$ So this relation cannot be solved by using Master Theorem

$\boxed{3}\boxed{6}\boxed{2}\boxed{1}\boxed{4}\boxed{5}$

At first, we know first index is already sorted because there is no element at the left hand side so we start from second index

current = 6

$\boxed{3}\boxed{6}\boxed{2}\boxed{1}\boxed{4}\boxed{5}$

Starting from second index, we look each index at left which is first index only. 3 is at the first index and 3<6 so they are in correct place. We got first two index sorted

current = 2

$\boxed{3}\boxed{}\boxed{6}\boxed{1}\boxed{4}\boxed{5}$

$\boxed{2}\boxed{3}\boxed{6}\boxed{1}\boxed{4}\boxed{5}$

Starting from third index (2), we go left until we reach to beginning or find an element that is smaller than 2. 6 is bigger than 2 so we put 6 to one next index. 3 is bigger than 2 So we put 3 to one next index. We have reached to beginning So we put 2 to there. We got first three index sorted.

current = 1

$\boxed{2}\boxed{3}\boxed{}\boxed{6}\boxed{4}\boxed{5}$

$\boxed{2}\boxed{}\boxed{3}\boxed{6}\boxed{4}\boxed{5}$

$\boxed{1}\boxed{2}\boxed{3}\boxed{6}\boxed{4}\boxed{5}$

Starting from fourth index (1), we go left until we reach to beginning or find an element that is smaller than 1. 6 is bigger than 1 so we put 6 to one next index. 3 is bigger than 1 so we put 3 to one next index. 2 is bigger than 1 so we put 2 to one next index. We have reached to beginning so we put 1 to there. We got first four index sorted.

current = 4

$\boxed{1}\boxed{2}\boxed{3}\boxed{}\boxed{6}\boxed{5}$

$\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{6}\boxed{5}$

Starting from fifth index (4), we go left until we reach to beginning or find an element that is smaller than 4. 6 is bigger than 4 so we put 6 to one next index. 3 is smaller than 4 so we stop here and put 4 here. We got first five index sorted.

current = 5

$\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{}\boxed{6}$

$\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5}\boxed{6}$

Starting from sixth index (5), we go left until we reach to beginning or find an element that is smaller than 5. 6 is bigger than 5 so we put 6 to one next index. 4 is smaller than 5 so we stop here and put 5 here. We got the full array sorted.

For visualization purpose index is shown as empty but actually they contains the element which is put one next index. For instance, $\boxed{3}\boxed{}\boxed{6}\boxed{1}\boxed{4}\boxed{5}$ is actually $\boxed{3}\boxed{6}\boxed{6}\boxed{1}\boxed{4}\boxed{5}$

③ a

i) Accessing the first element for both linked list and array is $\Theta(1)$.
In array, we can simply access the first element by a pointer directly.
In linked list, we hold the head node of the list so we can directly access the first element.

ii) In array, we can access the last element directly by array[size-1]. This can be done in $\Theta(1)$ time by adding proper number to base address of the array.

In linked list, if we don't maintain the pointer to last element, accessing the last element will be done in $\Theta(n)$ time because we traverse through the all list. If pointer to last element exists (like in Java's linked list), we can access the last element in $\Theta(1)$ time.

iii) In array, we can access the middle element by adding proper number to base address of the array. This can be done in $\Theta(1)$ time.

In linked list, we have to traverse through $\frac{n}{2}$ elements to access the middle element one by one. So this takes $\Theta(n)$ time.

iv) In array, we have to create new array because length of the first array is fixed when it is first created with size n+1 and copy all the elements to this array besides adding new element at the beginning of this array. Copying all elements and adding new element take $\Theta(n)$ time.

In linked list, we hold pointer the head node and head node has pointer to next node. So we can create new node and connects it to the former head. In this way we can add new element at the beginning in $\Theta(1)$ time.

v) In array, we have to create new array with size n+1 and copy all the elements to this array besides adding new element at the end of this array. Copying all elements and adding new element take $\Theta(n)$ time.

In linked list, if we don't have pointer to the last node in this situation we have to traverse until we reach to end of the list. So traversing one by one in n sized list and adding at the end take $\Theta(n)$ time.
If we have pointer to last node in linked list, then we can add at the end in $\Theta(1)$ time.

vi) In array, we have to create new array with size $n+1$ and copy all the elements to this array besides adding new element in the middle. Copying all elements adding new element take $\Theta(n)$ time.

In linked list, we have to traverse $n/2$ nodes to reach the middle element and add new element there. So this takes $\Theta(n)$ time.

vii) In array, we have to create new array with size $n-1$ and copy all the elements to this array except the first element. This takes $\Theta(n)$ time.

In linked list, we can access the second element by head node and make it our new head while deleting the former head node. This can be done in $\Theta(1)$ time.

viii) In array, we have to create new array with size $n-1$ and copy all the elements to this array except the last element. This takes $\Theta(n)$ time.

In linked list, if we don't have pointer to former node for each node and pointer to last node, we have to traverse until we reach end of the list. So traversing one by one in n sized list and deleting the last node via making null of the where "next to last node" points take $\Theta(n)$ time. If we have double linked list and pointer to last node, then we can delete the last element in $\Theta(1)$ time.

ix) In array, we have to create new array with size $n-1$ and copy all the elements to this array except the middle element. This takes $\Theta(n)$ time.

In linked list, we have to traverse $n/2$ nodes to reach the middle element and delete it via changing the where one former node points (it should point where the deleted node points). If we use double linked list, we should also change where the next node's "previous" part points. Traversing $n/2$ elements takes $\Theta(n)$ time.

b

In array we keep only elements sequentially in memory. Therefore an array with size n requires n * sizeof(element) space where element is what the array keeps.

In linked list, we have also pointers. On 64-bit system, pointer has 8 bytes. So we have to keep extra 8 byte space for each node if it is a single linked list. Therefore a linked list with size n requires n * (sizeof(element) + 8 byte) and also extra 8 byte is needed for head node.

4

```
convertToBST (tree)
    root ← root of the tree
    if root == null then
        return
    end if

    array ← []
    i ← 0
    traversal (root, array, pointer to i)

    for nextPos = 1 to len (array)
        tempPos ← nextPos
        nextValue ← array [nextPos]
        while (tempPos > 0 and nextValue < array [tempPos -1])
            array [tempPos] = array [tempPos -1]
            tempPos ← tempPos -1
        end while
        array [tempPos] = nextValue
    end for

    i ← 0
    traverseArray (root, array, pointer to i)
end

traversal (node, array, pointer to i)
    if node == null then
        return
    end if

    traversal (node→left, array, pointer to i)
    array [i] = node→data
    i ← i+1
    traversal (node→right, array, pointer to i)
end

traverseArray (node, array, pointer to i)
    if node == null then
        return
    end if

    traverseArray (node→left, array, pointer to i)
    node→data = array [i]
    i ← i+1
    traverseArray (node→right, array, pointer to i)
end
```

INSERTION SORT

In convertToBST function, I take the binary tree. Then I add values of this tree into an array with the help of traversal function. This travelsal is inorder traversal so order is "left subtree - root - right subtree". In traversal, I visit each element once so complexity is simply $\Theta(n)$.

After adding elements of tree into the array, I have sorted the array in ascending order by insertion sort. Best case for this step occurs if the array is already sorted:

$$B(n) = \sum_{i=2}^{n} 1 = n-1 \in \Theta(n)$$

→ For loop always executes
→ While loop doesn't execute

Worst case occurs if the array is reversely sorted:

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

→ For loop always executes
→ While loop always executes $i-1$ times where i is the current index.

Average case:

T_i = # of basic operations at step i for $1 \leq i \leq n-1$

$$T = T_1 + T_2 + \dots + T_{n-1} = \sum_{i=1}^{n-1} T_i$$

$$A(n) = E[T] = \sum_{i=1}^{n-1} E[T_i] = \text{Expected value of } T \quad \rightarrow E[T_i] = \sum_{j=1}^{i} j \cdot \underbrace{P(T_i = j)}_{\text{probability that there are } j \text{ comparisons in the } i^{th} \text{ step}}$$

$$P(T_i = j) = \begin{cases} \frac{1}{i+1} & \text{if } 1 \leq j \leq i-1 \\ \frac{2}{i+1} & \text{if } j = i \end{cases}$$

$$E[T_i] = \sum_{j=1}^{i-1} j \cdot \frac{1}{i+1} + i \cdot \frac{2}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}$$

$$A(n) = E[T] = \sum_{i=1}^{n-1} E[T_i] = \sum_{i=1}^{n-1} \left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{n \cdot (n-1)}{4} + n - 1 - \underbrace{\sum_{i=1}^{n-1} \frac{1}{i+1}}$$

$$\hookrightarrow \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \rightarrow \underbrace{H(n) - 1}_{\text{Harmonic series}}$$

$$A(n) = \frac{n \cdot (n-1)}{4} + n - H(n) \in \Theta(n^2)$$

After sorting the array, I can change values of the nodes of the tree so that it is converted into a BST. I do this with traverseArray function. This function makes inorder traversal and put the sorted array elements one by one. Traversal is inorder because in BST left node < root node < right node inequality holds. In traversal, I visit each element once so complexity is simply $\Theta(n)$.

Best case → $\Theta(n)$
Worst case → $\Theta(n^2)$
Average case → $\Theta(n^2)$

(5) # Similar algorithm to 4^{th} question in last homework is applied.

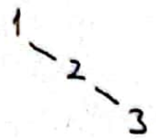
```
def linearPairDetection (A, x):

    size = len (A)

    if x < 0:
        print ("Invalid x value.")
        quit()

    frequency_dict = dict()

    for i in range (0, size):
        if frequency_dict.get (A[i]):
            frequency_dict [A[i]] = frequency_dict[A[i]] + 1
        else:
            frequency_dict [A[i]] = 1

        if frequency_dict [A[i]] >= 2 and x==0:
            print (" pair: ({}, {})". format (A[i], A[i])

    if x != 0:
        for i in range(0, size):
            if frequency_dict.get (x+ A[i]):
                print ("pair: ({}, {})". format (A[i], x+A[i])
```

Firstly, algorithm controls if x is negative. If so, algorithm ends because absolute value of something cannot be negative. Then a python dictionary is created. This dictionary keeps array elements as keys and their frequencies as values. Reason for choosing the dictionary is accessing the value of a key in constant time with good hashing for integers. In first for loop, algorithm traverses through the array and put proper value for array elements which are keys. This loop takes $O(n)$ time where n is the size of the array. All the operations inside the loop is $\Theta(1)$. Accessing the value of a key can be done in constant time as I said. At the end of the loop I control if x is 0 and an element occurs at least twice so that I can indicate element - element $=0$. At the end of the algorithm there is another for loop. Here algorithm traverses through the array and check if there is an integer with value x + current element which actually ensures $|a_i - a_j| = x$. This loop is also $O(n)$ because it traverses all the array elements and each iteration takes constant time. All in all, algorithm takes $O(n)$ time. I did not use array with size max element +1 and keep frequencies there because array may contain negative values.

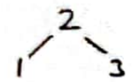
6

a) T

It depends on the insertion order. For example, suppose that we want to add 1, 2, 3 to the BST. If we add in this order we get the worst tree:



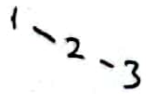
If we add in (2, 1, 3), we get the perfect tree:



So adding the middle element first is good idea when creating a BST.

b) T

It might be linear. If we add elements to BST in sorted order we get very unbalanced BST. For example if we add 1, 2, 3 in order to BST;



and if we try to access 3, we have to go through all the elements so this results in linear time complexity.

c) F

It cannot be done in constant time if array is not sorted. Maybe it can be found by luck but it is theoretically impossible because we have to check all the elements to determine if an element is bigger or smaller than all other elements.

d) F

This is true for array but not for linked list because accessing the middle element already takes $\Theta(n)$ time. So worst case is worse than $O(\log n)$ because of accessing the elements.

e) F

It is $\Theta(n^2)$. For each iteration, $i-1$ comparisons have to be performed for i^{th} element. This is the maximum number of comparisons that occurs if array is reversely sorted.

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