CSE 327 - Homework #2

Master Theorem

$T(n) = a\,T(n/b) + f(n)$

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \varepsilon}) \\ \Theta(n^{\log_b a} \cdot \log n) & f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \varepsilon}) \end{cases}$$

$a f(n) \le b \cdot f(n)$

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

$a = 2 \quad b = 4 \quad f(n) = \sqrt{n \log n}$

$n^{\log_4 2} = n^{1/2} = \sqrt{n}$

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

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

$a \ge 1 \quad b > 1 \quad k \ge 0$

$p > -1 \quad k = 1/2 \quad p = 1/2$

$a = b^k$

$2 = 4^{1/2}$

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

$n^{\log_3 9} = n^2 = \Theta(n^2) = f(n)$ state 2

$\Theta(n^{\log_3 9} \cdot \log n) = \Theta(n^2 \log n)$

c) $T(n) = \frac{1}{2} T(n/2) + n$ (conditions $a \ge 1$, $b > 1$)

$a = 1/2$ master theorem can not apply on because a is $1/2$. "a" should become greater than 1 or equal 1.

d) $T(n) = 5 \cdot T(n/2) + \log n \quad a = 5 \quad b = 2 \quad f(n) = \log n$

$\Theta(n^{\log_2 5 - \varepsilon}) = \log n$ state 1

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

e) $T(n) = 4^n \cdot T(\frac{n}{5}) + 7$ $\quad a = 4^n$

Given equation is comparable with master theorem equation so it can't solve using master theorem.

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

$a = 7 \quad b = 4 \quad f(n) = n \log n$

$n^{\log_4 7} > n \log n \qquad n \log n \in O(n^{\log_4 3}) \quad$ false

$\Theta(n^{\log_4 7})$ $\quad$ a) $b \cdot \epsilon \quad k = 7$

7) 4.

g) $T(n) = 2 \cdot T(\frac{n}{3}) + 7n \qquad a = 2 \quad b = 3 \qquad f(n) = 7n$

$k = -7$

So it can not be solve using master theorem. Because k is -7. k should become greater than zero or equal.

h) $T(n) = \frac{2}{5} \cdot T(\frac{n}{5}) + n^5 \qquad a = \frac{2}{5} \quad b = 5 \qquad f(n) = n^5$

$k = 5$

it can not be solve using master theorem a is $\frac{2}{5}$. a should become greater than zero or equal.

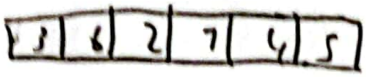
Solution

Let us apply the insertion sort algorithm on the given array

array is A = (3, 6, 2, 7, 4, 5)

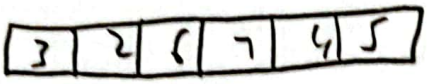
First: Initially the first case two element compared.

First two element to be ascending order. hence no swap occur.

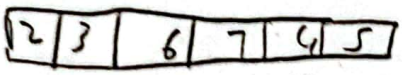
The array becomes: 

Second

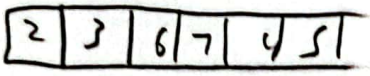
Now element 2 and 6 are compared. Both 6 and 2 are not present at their correct place so swap them



After swapping 2 and 3 are not sorted, thus swap again

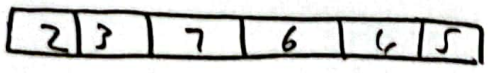


Now 2 and 3 are present in the sorted sub-array the array becomes

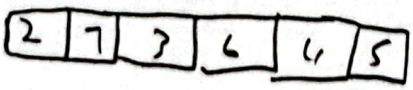


Third

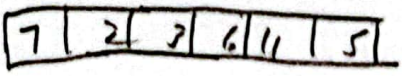
Now element 6 and 7 are compared 6 and 7 are not sorted at their correct place so swap them



- Both 7 and 3 are not present at their correct place so swap them

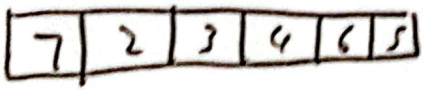


Here again 2 and 7 are not sorted hence swap it again



Fourth

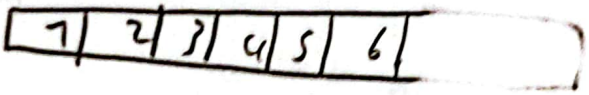
Now compare 6 and 4. Both 6 and 4 are not present at their correct place so swap them.



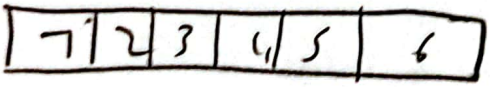
Now 7, 2, 3, 4 are sorted in sub array no swapping needed the array becomes

Fifth

Now compare 6 and 5. both 6 and 5 are not present at their correct place so swap them.



Now finally array is completely sorted



3)

Array is data structure which holds the collection of some type of data. array is used for fast accessing of data

Linked list is a linear data structure which holds the data in linked manner, each data is in the form of node and the node can have address of another node
if the node is last element in the list then the next address of node is set to null

the implementation goes from array to linked list is arrays size, once we create an array then it is fixed and we cannot increment and decrement the array size

```
int a[20]
```

the above case int a[20] which holds the array of size 20 of int data type

The below is the solution for the above problem

a) Here the linked list has the two pointers those are head which will points to the start of the list node and the tail pointer which will points to the last node in the list and n represents size of the linked list

Operation Name	Array	Array Explanation	Linked List	Linked List Explanation
Accessing the first element	O(1)	In array we can use the indexing element so the acces first element we can use the 0 index	O(1)	Because the head will points to the first node so we can easly access the head in single operation
Accessing to last element	O(1)	we can use the indexing here also so length -1 index is used to acces to last element	O(1)	Because tail will points to the last node in the linked list. so we can acces the last element in the single operation
Accessing any element in middle	O(1)	we can acces element in the middle so it takes single operation only	O(n)	because access element in the middle we need to move the curr pointer to middle from head. so it takes n operations at worst
Adding a new element at the end	O(1)	we can use the indexing to set the last index with some value so it takes the single operation	O(1)	we can make the tail.next as new node and then make the inserted element node as tail
Adding the new element at the beginning	O(n)	adding the element at beginning takes 1 operation only. But after adding we need to shift all the existing element right by one position	O(1)	we can access the head and then we can create the new node and new node.next as head and using new node as head

Adding a new element at the end	O(1)	indexing to last element takes the single operation	O(1)	take new interest address tail take single operation
Adding the new element at the beginning	O(n)	adding the element until beginning takes 1 operation only But after adding we need to shift all existing element right by one position	O(1)	we can access the head one then we can create the new node one node.next as head and assigns new node as head
Adding new element in the middle	O(n)	adding element at the middle takes 1 operation only. but after adding we need to shift all indexes after middle to right by 1 position	O(n)	we need loop from head to middle and then we need to create a new node and change the links. it takes n operations at worst case
Deleting the first element	O(n)	after deleting the first element we need to shift the all elements from index 1 to length by 1 to the left side	O(1)	we need to change the head node only. and after changing head node we need to free the previous head node
Deleting the last element	O(1)	we need to delete or insert the last index value only there are no shifting if needed	O(n)	we need to loop the list from head to last before node or tail and then we need to free the tail node and assign the before node or tail as tail node
Deleting any elements in the middle	O(n)	after deleting the element we need to shift the elements in the right side by 1 position to left side		we need to loop the list from head to the middle node and then we need to remove the middle node and then we need to change the links

(1)

worst case	best case
$O(n^2)$	$O(n)$

Average case:

$O(n^2)$

```
Binary to BST (Node root) {
  if (root = null)
    return;                    } 7
  int n = countNodes(root);    O(n)
  int arr[n];                  } 7
  storeInorder(root, arr)      O(n)
  Arrays.sort(arr)  → worst case = O(n^2) best case → O(n) average case → O(n^2)
  index = 0;                   } 7
  ArrayToBST(arr, root)        O(n)
}                              T(n) = O(n^2) + 3O(n) + 3
```

```
int countNodes(Node root) {
  if(root == null)
    return 0;      } 7
  return countNodes(root.left) + countNodes(root.right) + 7;
}
```

count node function

$T(n) = 2T(n/2) + 7$

we can use master theorem

$a = 2 \quad b = 2 \quad f(n) = 7$

$n^{\log_2 2 - \varepsilon} \rightarrow$ epsilon

$n^{\log_2 2 - 3} \in O(7)$ state 7

$n^{\log_2 2} = n^7 \quad O(n)$

```
storeInorder(Node node, int inorder[]) {
  if(node == null)
    return;            } 7
  storeInorder(node.left, inorder);   T(n/2)
  inorder[index] = node.data;
  index++;             } 2
  storeInorder(node.right, inorder);  T(n/2)
}
```

$T(n) = 2T(n/2) + 3$

$a = 2 \quad b = 2 \quad f(n) = 3$

we can use master theorem

$n^{\log_2 2 - 3} \in O(7)$ state 7 → $n^{\log_2 2} = n^7 \quad \Theta(n)$

void arrayToBST(int[] arr, Node root){

if(root == null)
    return; } 1

arrayToBST(arr, root.left); T(n/2)
root.data = arr[index]; 1
index++; 1
arrayToBST(arr, root.right) T(n/2)
}

T(n) = 2T(n/2) + 3

we can use master theorem

a = 2 b = 2 f(n) = 3

$n^{\log_2 2 - 3}$ ∈ O(1) state 1 → $n^{\log_2 2} = n^1 = O(n)$

```
def solution(arr, x):
    max = max(arr)
    min = min(arr)
    result_list = []
    if min < 0:
        min = -1
    mid_size = max(min, max)

    size_array = np.zeros(mid_size*2+1, dtype=np.int64)
    for i in range(0, len(arr)):
        if arr[i] > 0:
            size_array[element] += 1
        else:
            size_array[element] += 1

    for i in range(0, len(size_array)):
        if size_array[i] > 0 and size_array[i+1] == 0:
            a = 0
            b = i+1
            if i > mid_size:
                a = -2*mid_size + a - 1
            if i+1 > mid_size:
                b = -2*mid_size + b - 1
            result_list.append([a, b])
            return result_list
```

O(n)

O(n)

O(n)

Time complexity

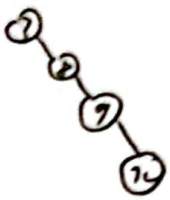
O(n)

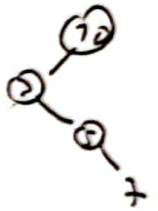
6.)

a) True depends into the order. The tree decision route with greater or smaller.

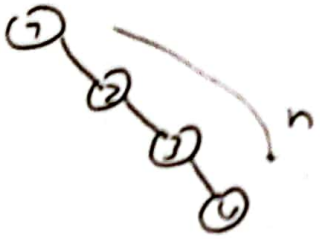
example

38 9 7 70 10, 3, 5 7





b) Yes True. Some cases access element linear. For example below



c) Yes True, if array is sorted, can access with constant time else we can access O(n) time

d) Linked worst case is O(n) because n is length of linked list. This is False.

e) No False, best case is $\Theta(n)$ is insertion sort Algorithm. worst case is $\Theta(n^2)$ if array is reversed order.