CSE 321
Homework 2

Abdullah Gelk
171044002

1) Master Theorem: $T(n) = a \cdot T(\frac{n}{b}) + \Theta(n^k \log^p n)$ where

n is size of problem,

a is number of subproblems in the recursion and $a \geq 1$,

$\frac{n}{b}$ is size of the each subproblem,

$b > 1$, $k \geq 0$ and $p \in \mathbb{R}$

Then,

$$T(n) \in \begin{cases} \Theta(n^{\log_b^a}) & \text{if } a > b^k \\ \Theta(n^{\log_b^a} \cdot \log^{p+1} n) & \text{if } a = b^k,\ p > -1 \\ \Theta(n^{\log_b^a} \cdot \log(\log n)) & \text{if } a = b^k,\ p = -1 \\ \Theta(n^{\log_b^a}) & \text{if } a = b^k,\ p < -1 \\ \Theta(n^k \log^p n) & \text{if } a < b^k,\ p \geq 0 \\ \Theta(n^k) & \text{if } a < b^k,\ p < 0 \end{cases}$$

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

$a = 2,\ b = 4,\ k = \frac{1}{2},\ p = \frac{1}{2}$

$a = b^k$, $p \geq 0$ then $T(n) \in \Theta(n^{\frac{1}{2}} \cdot \log^{\frac{3}{2}} n)$

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

$a = 9,\ b = 3,\ k = 2,\ p = 0$

$a = b^k$, $p = 0$ then $T(n) \in \Theta(n^{\log_3^9} \cdot \log n) \Rightarrow T(n) \in \Theta(n^2 \log n)$

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

$a = \frac{1}{2}$ then a should be greater than 1. So master theorem cannot solve the equation.

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

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

$a > b^k$ then $T(n) \in \Theta(n^{\log_2^5})$

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

The equation form does not fit moster theorem form. Moster theoren cannot solve this equation.

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

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

$a > b^k$ then $T(n) \in \Theta(n^{\log_4 7})$

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

$k = -1$. k should be greater and equal then 0. $(k \geq 0)$.

Master theorem cannot be use.

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

$a = \frac{2}{5}$. a should be greater and equal then 1. $(a \geq 1)$

Master theorem cannot be use.

2) procedure InsertionSort (L [1 : n])

```
        for i=2 to n do
            current = L[i]
            position = i-1
            while ( position ≥1 and current < L[position]) do
                L [position +1] = L [position]
                position = position -1
            end while
            L [position +1] = current
        end for
    end
```

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

	current	position	L[position]	L[position +1]	while condition	L	
i=2	6	1	3	6	false		
i=3	2	2	6	2	true		place element 2 to first position
		1		6		3, 6, 6, 1, 4, 5	
	2	1	3	6	true		
		0		3		3, 3, 6, 1, 4, 5	
	2	0	–	3	false		
	2	0	–	2		2, 3, 6, 1, 4, 5	
i=4	1	3	6	1	true		place element 1 to first position
		2		6		2, 3, 6, 6, 4, 5	
	1	2	3	6	true		
		1		3			
	1	1	2	3	true	2, 3, 3, 6, 4, 5	
		0		2			
	1	0	–	2	false	2, 2, 3, 6, 4, 5	
	1	0	–	1		1, 2, 3, 6, 4, 5	
i=5	4	4	6	4	true		place element 4 to right position
		3		6		1, 2, 3, 6, 6, 5	
	4	3	3	6	false		
	4	3	3	4		1, 2, 3, 4, 6, 5	
i=6	5	5	6	5	true		place element 5 to right position
		4		6		1, 2, 3, 4, 6, 6	
	5	4	4	6	false		
	5	4	4	5		1, 2, 3, 4, 5, 6	

3)

	In array	In linked list
i) Accessing the first element	Calculate offset and access first element. $w(n) \in \Theta(1)$	Access to the head node. $w(n) \in \Theta(1)$
ii) Accessing the lost element	Calculate offset and access lost element. $w(n) \in \Theta(1)$	I assume that there is no tail node. Traverse all nodes for accesing lost node. $w(n) \in \Theta(n)$
iii) Accessing any element in the middle	Calculate offset and access middle element. $w(n) \in \Theta(1)$	Traverse from first node to middle node. $w(n) \in \Theta(\frac{n}{2})$ $w(n) \in \Theta(n)$
iv) Adding a new element at the beginning	Shift each element to the position of the next element. Place first element to the beginning. $w(n) \in \Theta(n)$	Add first node to the head. $w(n) \in \Theta(1)$
v) Adding a new element at the end	Allocate memory. Copy elements to new memory location. Place new elemet at the end. $w(n) \in \Theta(n)$	I assume that there is no tail node. Traverse all node for accesing lost node.. Add new element. $w(n) \in \Theta(n)$
vi) Adding a new element in the middle	Shift elements after the middle element. Place new element. $w(n) \in \Theta(n)$	Iterate first to middle node. Add node. $w(n) \in \Theta(n)$
vii) Deleting the first element	Each element must be shifted to the position of the previous element. $w(n) \in \Theta(n)$	Set head to second node. $w(n) \in \Theta(1)$
viii) Deleting the last element	There is no shifting. $w(n) \in \Theta(1)$	Iterate first to lost node for accesing lost node. Delete lost node. $w(n) \in \Theta(n)$
ix) Deleting any element in the middle	Elements after the middle element should be shifted to the position of the previous element. $w(n) \in \Theta(n)$	Iterate first to middle node for accessing middle node. Delete middle node $w(n) \in \Theta(n)$

b) Since each node of the linked list is connected to the next node it requires more memory than the array. Array will be more advantageous if our use case is to access elements. If our use case is about adding and deleting elements, linked list will be better

4) → Build an array and store nodes in binary tree while inorder traverse

→ Sort this array

→ Traverse binary tree and store the elements from the sorted array to yield binary search tree.

Pseudo code:

```
procedure   binaryTreeToBinarySearchTree ( Root )
        if Root is null then
            return
        end if

        storeInOrder(Root , array)
        sort(array)
        restoreInOrder(Root , array, 0)
end

procedure   storeInOrder ( Root , A[1:n])
        if Root is null then
            return
        end if

        storeInOrder( Root.left , A)
        A.add (Root.data)
        storeInOrder(Root.right, A)
end
procedure   restoreInOrder ( Root , A[1:n] , i)
        if Root is null then
            return
        restoreInOrder (Root.left , A, i)
        Root.data = A[i]
        i = i+1
        restoreInOrder (Root.right , A, i)
end
```

Complexity:

- Inorder traversal complexity is $\Theta(n)$.
- Sorting complexity is $\Theta(n \log n)$. (quick sort, merge sort)
- Total complexity is $A(n) \in \Theta(n \log n)$
- The same operation will be done in best case and worst case.
  Therefore $A(n), B(n), W(n) \in \Theta(n \log n)$

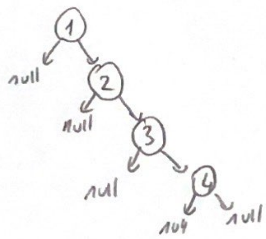
5) While iterating over the array, calculate the $x - A[i]$ and $x + A[i]$.
Then check are there calculated value in the hash table. If any return x and item
If not, insert the item to the hash table.

```
procedure findPair ( A[1:n], x)
    for i=1 to n do                          -- Θ(n)
        y = A[i]+x                           Θ(1)
        z = A[i] -x                          Θ(1)

        if table.search(y) is true  ---      Θ(1)        A(n) ∈ Θ(n)
            return {A[i], y}        ---      Θ(1)
        end if

        if table.search (z) is true --       Θ(1)
            return {A[i], z}        ----     Θ(1)
        end if

        table.insert (A[i])         ----     Θ(1)
    end for
    return ∅
end
```

I use hash table, because inserting and accesing element in hash table is constant. Then the remaining is to iterate over array. Table can be selected according to the programming language used. (hash, dictionary etc.)

6) a) Yes it depends. When adding in BST, if the new element is smaller than the element, it is positioned in the left subtree. If it is greater than the element, it is positioned in the right subtree. For example, since the first element to be added will be the first element of the tree. It also determines where the next element to be added will be located.

b) True. When BST is not balanced, searching takes linear time. The sample tree is below.



c) True. If the array is sorted and we know this, the minimum and maximum elements will be in the first and last positions. Accesing time is constant.

d) False. Accesing an element in the linked list is linear time. If accesing time is constant time then binary search is $\log n$. If accesing an element is n then, binary search in linked list is $n \log n$. $A(n) \in \Theta(n \log n)$

e) False. Worst case of insertion sort $\in \Theta(n^2)$. When the array is in reverse order, we will move each element up to the first position. The elements up to the current element will be shifted in each iteration.