CSE 321 INTRODUCTION TO ALGORITHM DESIGN

HW2

REPORT 2

Merve Horuz

1801042651

Master Theorem: $T(n) = aT(n/b) + f(n)$

$a \geq 1$ $\quad f(n) = \theta(n^k \log^p n)$

$b > 1$

case 1: if $\log_b^a > k \rightarrow \theta(n^{\log_b^a})$

case 2: if $\log_b^a = k$

if $p > -1$ $\quad \theta(n^k \log^{p+1} n)$

if $p = -1$ $\quad \theta(n^k \log\log n)$

if $p < -1$ $\quad \theta(n^k)$

case 3: if $\log_b^a < k$

if $p \geq 0$ $\quad \theta(n^k \log^p n)$

if $p < 0$ $\quad \theta(n^k)$

---

1)

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

$\log_b^a = \log_4^2 = \frac{1}{2}$ $\quad f(n) = \sqrt{n\log n}$ $\quad k = \frac{1}{2}$ $\quad p = \frac{1}{2}$

case 2 is occurs and $p > -1$ so, $\theta(\sqrt{n\log^3 n})$

b) $T(n) = 9.\ T(n/3) + 5n^2$

$\log_b a = \log_3 3^2 = 2$ $\quad f(n) = 5n^2$ $\quad k = 2$ $\quad \log^p n = 5$

$p = 0$

$\log_b a = k$ $\quad$ case 2 is occurs and $p > -1$

so, $\theta(n^2 \log n)$

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

$a = \frac{1}{2}$ $\quad b = 2$ $\quad \log_2 2^{-1} = -1$ $\quad k = 1$ $\quad p = 0$

a is not ≥ 1 so this relation can't be solved by using master theorem,

d) $5.\ T(n/2) + \log n$

$\log_b a = \log_2^5 = 2{,}32$ $\quad f(n) = \log n$ $\quad k = 0,\ p = 1$

case 1 occurs so, $\theta(n^{\log_2^5})$

e) $T(n) = 4^n . T(n/5) + 1$

$\log_b^a = \log_5^{4^n}$ $f(n) = 1$ $p = 0$, $k = 0$

we cannot solve this problem with master theorem because there is unknown term (4^n).

f) $T(n) = 7 . T(n/4) + n\log n$

$\log_b^a = \log_4 7 = 1.4$ $f(n) = n\log n$ $k = 1$, $p = 1$

case I occurs so, $\Theta(n^{\log_4 7})$

g) $T(n) = 2 . T(n/3) + 1/n$

$\log_b^a = \log_3 2 = 0.63$ $f(n) = 1/n$ $k = -1$ $p = 0$

case I occurs, so, $\Theta(n^{\log_3 2})$

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

$\log_5 \frac{2}{5} = 0.57$ $f(n) = n^5$ $k = 5$, $p = 0$

$a = 2/5$

a is not ≥ 1 so this equation can not be solved with master theorem

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

Assume that indexes start at 1 not 0.

```
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
```

1. i=2, current = 6, position = 1
2. While (1 >= 1 and 6 > 3)   // this loop has no iterates due to current is not bigger L[position]
3. L [2] = 6
4. Current array A = {3, 6, 2, 1, 4, 5}

1. i=3, current = 2, position = 2
2. While (2 >= 1 and 2 < 6)   // this loop iterates two times
   a. L [3] = L [2]
   b. Position = position – 1
   c. …
3. L [1] = 2
4. Current array A = {2, 3, 6, 1, 4, 5}

1. i=4, current = 1, position = 3
2. While (3 >= 1 and 1 < 6)
   a. L [4] = L [3]
   b. Position = position – 1
   c. …   // It iterates until last array is A = {2, 2, 3, 6, 4, 5}
3. L [1] = 1
4. Current array A = {1, 2, 3, 6, 4, 5}

```
1. i=5, current = 4, position = 4
2. While (4 >= 1 and 4 < 6)
     a. L [5] = L [4]
     b. Position = position – 1
     c. ...                          // It iterates until last array is A = {1, 2, 3, 6, 6, 5}
3. L [4] = 4
4. Current array A = {1, 2, 3, 4, 6, 5}
```

```
1. i=6, current = 5, position = 5
2. While (5 >= 1 and 5 < 6)
     a. L [6] = L [5]
     b. Position = position – 1
     c. ...                          // It iterates until last array is A = {1, 2, 3, 4, 6, 6}
3. L [5] = 5
4. Current array A = {1, 2, 3, 4, 5, 6}
```

The last array is A = {1, 2, 3, 4, 5, 6}.

3a)

	i	Ii	iii	iυ	υ	υi	υii	υiii	ix
array	O (1)	O (1)	O (1)	O (n)	O (n)	O (n)	O (n)	O (n)	O (n)
linkedL	O (1)	O (n)	O (n)	O (1)	O (n)	O (n)	O (1)	O (n)	O (n)

i) Accessing the first element of array O (1) because of first element can be accessed by index. Accessing the first element of linked list O (1) because head of linked list is stored in another υariable.

ii) Accessing the last element of array O (1) because of last element can be accessed by index. Accessing the last element of linked list O (n) because iteration is required through all elements (n).

iii) Accessing any element of array O (1) because any element can be accessed by index. Accessing any element of linked list O (n) because iteration is required through maximum n-1 elements. So, we can consider O (n-1) as O (n) by eliminating constants.

iυ) Adding a new element at the beginning of array O (n). New memory allocation is required (n+1 cell) and all (n+1) elements are added to the new array so, time complexity is O (n). Adding a new element at the beginning of linked list O (1). The head of linked list is stored in another υariable, pointer of new element is set to head of linked list and head of linked list is updated to new added element.

υ) Adding a new element at the end of the array O (n). New memory allocation is required (n+1 cell) and all (n+1) elements are added to the new array so, time complexity is O (n). Adding a new element at the end of the linked list O (n) because iteration is required through all elements (n) so that access the end of the list.

υi) Adding a new element in the middle of the array O (n). New memory allocation is required (n+1 cell) and all (n+1) elements are added to the new array so, time complexity is O (n). Adding a new element in the middle of the linked list O (n) because iteration is required through some elements (n-1 or less) to access the place in the list where the element is to be added.

υii) Deleting the first element of the array O (n). New memory allocation is required (n-1) and all n-1 elements are added to the new array. Deleting the first element of the linked list O (1). The head of linked list is stored in another υariable, head of linked list is updated to second element and freed first element.

υiii) Deleting the last element of the array O (n). New memory allocation is required (n-1) and all n-1 elements are added to the new array. Deleting the last element of the linked list O (n) if the tail of the linked list is not stored in another υariable. Iteration is required through all elements (n) so that access the end of the list and freed last element by setting pointer of penultimate element to null.

ix) Deleting any element in the middle of array O (n). In the worst case, deleting second element, all elements except 1st and 2nd elements must be shifted to left which becomes O (n-2) so, eliminate constants -> O (n). Deleting any element in the middle of linked list O (n) because iteration is required to item which will be deleted (In worst case, (n-1)th element).

3b)

i) No need for extra space for both data structures.
ii) No need for extra space for both data structures.
iii) No need for extra space for both data structures.
iv) For array, n+1 extra space is required. For the linked list, only 1 extra space is required.
v) For array, n+1 extra space is required. For the linked list, only 1 extra space is required.
vi) For array, n+1 extra space is required. For the linked list, only 1 extra space is required.
vii) For array, n-1 extra space is required. For the linked list, no need for extra space.
viii) For array, n-1 extra space is required. For the linked list, no need for extra space.
ix) For array, n-1 extra space is required. For the linked list, no need for extra space.

4)

i) Values of nodes in binary tree are added to array as inorder.

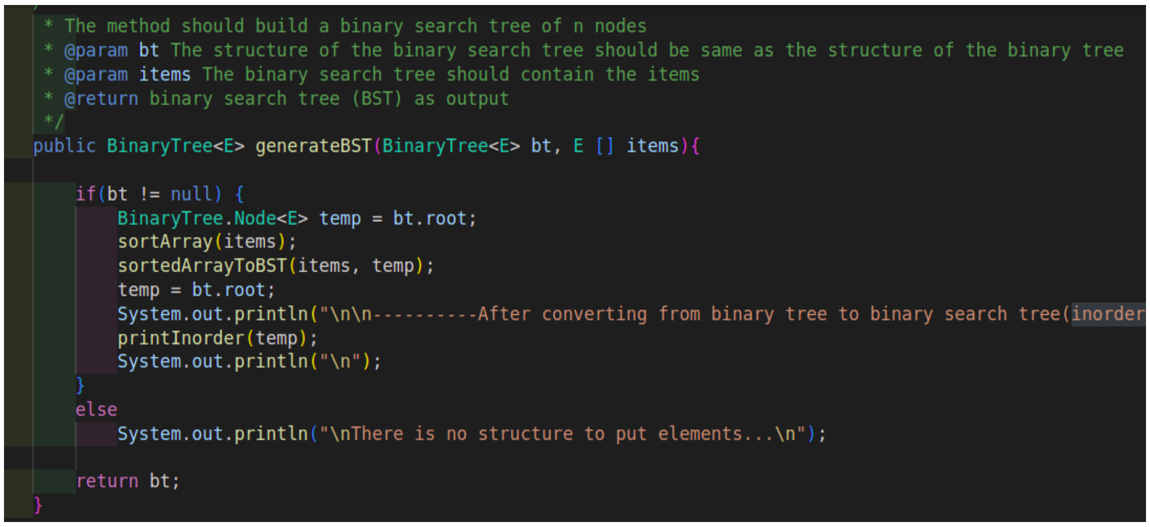
ii) Convert sorted array to binary search tree. //sortedArrayToBST(items, root)

a) If root is null return

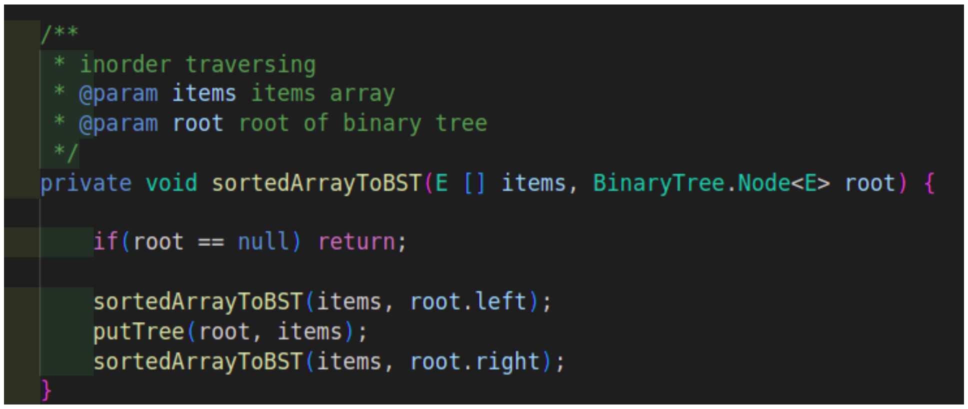
b) Call itself of this function by giving root as root.left

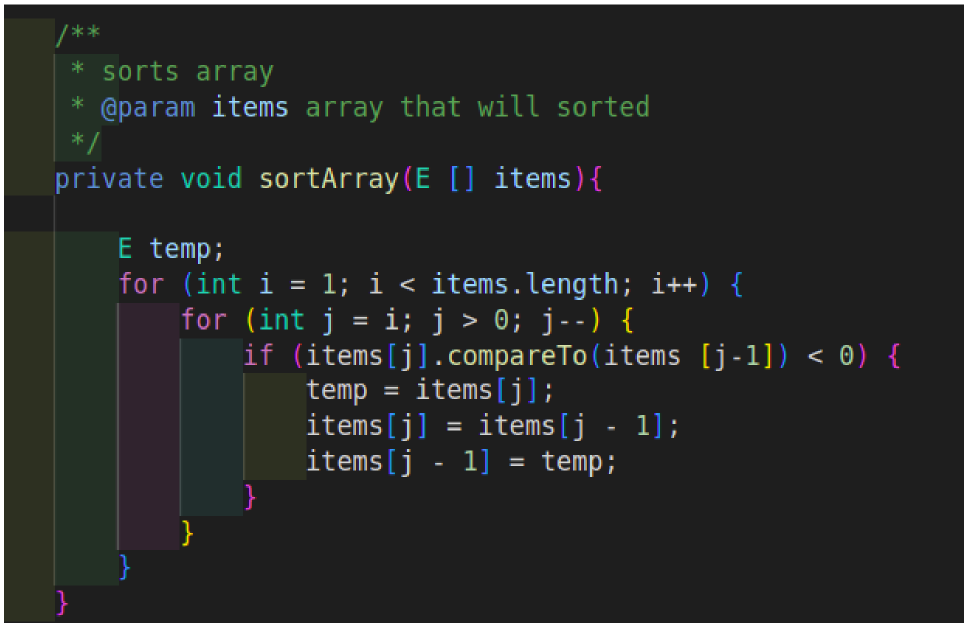
c) Put element in the next index to specific node of tree

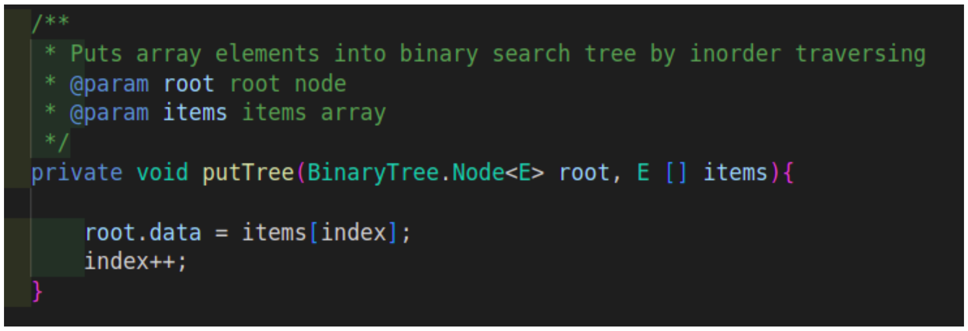
d) Call itself of this function by giving root as root.right



```java
 * The method should build a binary search tree of n nodes
 * @param bt The structure of the binary search tree should be same as the structure of the binary tree
 * @param items The binary search tree should contain the items
 * @return binary search tree (BST) as output
 */
public BinaryTree<E> generateBST(BinaryTree<E> bt, E [] items){

    if(bt != null) {
        BinaryTree.Node<E> temp = bt.root;
        sortArray(items);
        sortedArrayToBST(items, temp);
        temp = bt.root;
        System.out.println("\n\n----------After converting from binary tree to binary search tree(inorder
        printInorder(temp);
        System.out.println("\n");
    }
    else
        System.out.println("\nThere is no structure to put elements...\n");

    return bt;
}
```


```
/**
 * inorder traversing
 * @param items items array
 * @param root root of binary tree
 */
private void sortedArrayToBST(E [] items, BinaryTree.Node<E> root) {

    if(root == null) return;

    sortedArrayToBST(items, root.left);
    putTree(root, items);
    sortedArrayToBST(items, root.right);
}
```


```
/**
 * sorts array
 * @param items array that will sorted
 */
private void sortArray(E [] items){

    E temp;
    for (int i = 1; i < items.length; i++) {
        for (int j = i; j > 0; j--) {
            if (items[j].compareTo(items [j-1]) < 0) {
                temp = items[j];
                items[j] = items[j - 1];
                items[j - 1] = temp;
            }
        }
    }
}
```


```
/**
 * Puts array elements into binary search tree by inorder traversing
 * @param root root node
 * @param items items array
 */
private void putTree(BinaryTree.Node<E> root, E [] items){

    root.data = items[index];
    index++;
}
```

Time complexities:

Since the basic operation in the above algorithm is the sorting of the values in the array, only the time complexity of the sort function is checked and insertion sort was used.

COST OF LINE	NO. OF TIMES IT IS RUN
L1	1
L2	n
L3	$\sum^{n-1}_{j=1}(t_j)$
L4	$\sum^{n-1}_{j=1}(t_{j-1})$
L5	$\sum^{n-1}_{j=1}(t_{j-1})$
L6	$\sum^{n-1}_{j=1}(t_{j-1})$
L7	$\sum^{n-1}_{j=1}(t_{j-1})$

$T(n) = L1*1 + L2*n + L3*\sum^{n-1}_{j=1}(t_j) + (L4 + L5 + L6 + L7) * \sum^{n-1}_{j=1}(t_j)$

In Best Case i.e., when the array is already sorted, $t_j = 1$
Therefore, $T(n) = L1*1 + L2*n + L3*(n-1) + (L4 + L5 + L6 + L7)*(n-2)$
which when further simplified has dominating factor of n and gives $T(n) = L*(n)$ or $O(n)$.

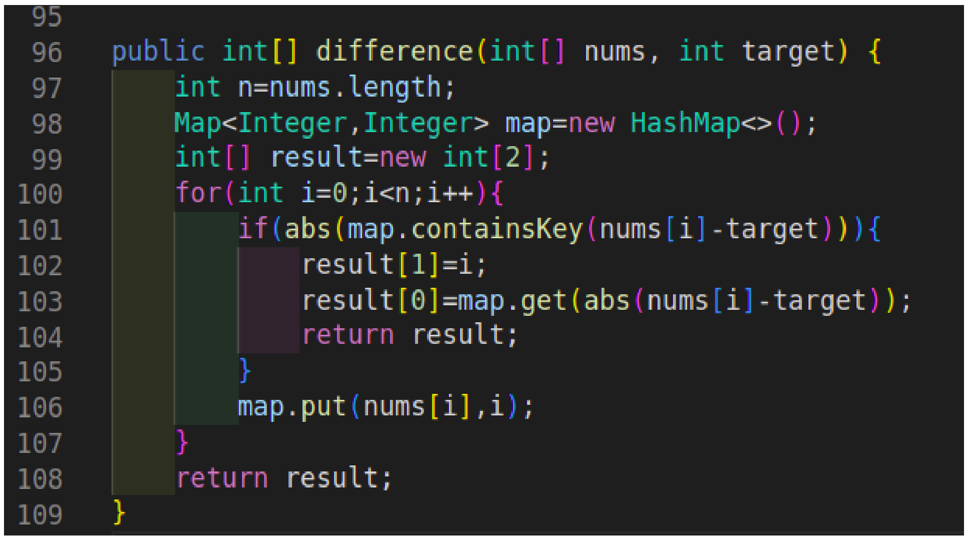
In Worst Case i.e., when the array is reversely sorted (in descending order), $t_j = j$
Therefore, $T(n) = L1*1 + L2*n + L3*((n-1)(n)/2) + (L4 + L5 + L6 + L7)*((n-1)(n)/2-1)$
which when further simplified has dominating factor of n^2 and gives $T(n) = L*(n^2)$ or $O(n^2)$

In average case, $t_j = (j-1)/2$
Therefore, $T(n) = L1*1 + L2*n + L3/2*((n-1)(n)/2) + (L4 + L5 + L6 + L7)/2*((n-1)(n)/2-1)$
which when further simplified has dominating factor of n^2 and gives $T(n) = L*(n^2)$ or $O(n^2)$

5)

1. Store number of items in a variable
2. Create a new HashMap object
3. Create a new array to store indexes of a pair
4. Iterate 0 to n (for loop)
5. If map has such a key nums[i]-target
    a. Store indexes of the results
    b. Return result
6. Otherwise put the nums[i] as key to map and i as value to map
7. Return result

Since a1 - a2 = x, to put a1 as a HashMap key and where there is a key equal to (x+a2) (because a1 = (x+a2)), it is a1, which is the value we are looking for, so (x+a2) When we find an equal value, we find the pair we are looking for. The aim of using HashMap, put, get and containsKey methods runs in O (1) time.

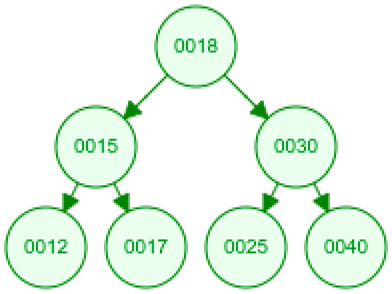


```
public int[] difference(int[] nums, int target) {
    int n=nums.length;
    Map<Integer,Integer> map=new HashMap<>();
    int[] result=new int[2];
    for(int i=0;i<n;i++){
        if(abs(map.containsKey(nums[i]-target))){
            result[1]=i;
            result[0]=map.get(abs(nums[i]-target));
            return result;
        }
        map.put(nums[i],i);
    }
    return result;
}
```

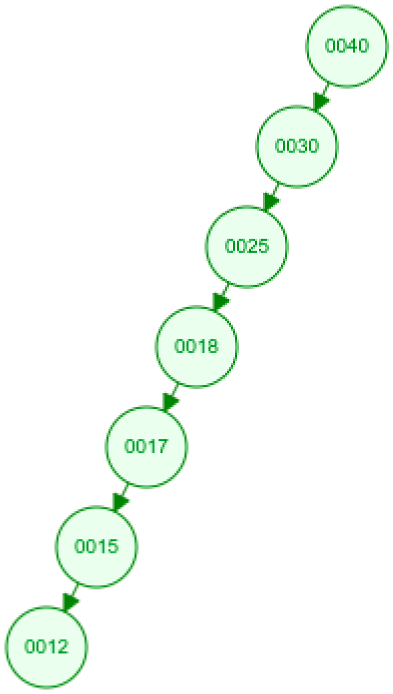
6)

a) True, it depends according to insertion order. A Binary Tree is a full binary tree if every node has 0 or 2 children. For example, if [18, 15, 30, 12, 17, 25, 40] array is inserted in this order, tree becomes full binary search tree, but if [40, 30, 25, 18, 17, 15, 12] array is inserted in this order, tree is not a full binary search tree.

This is a full binary tree, if elements are inserted in the first order:

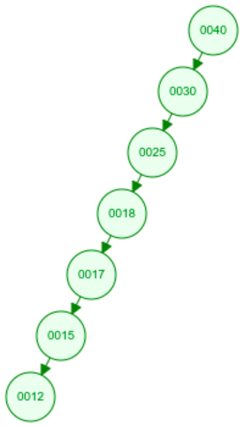


This is not a full binary tree, if elements are inserted in the second order:



b) True, time complexity of accessing 12 of a such BST is linear:

First it checks root (40), then 12 is less than 40, it checks root of left subtree, then 12 is less than 30, it checks root of the left subtree again, then 12 is less than 25, it checks root of the left subtree again, then 12 is less than 18, it checks root of left subtree again, then 12 is less than 17, it checks root of left subtree again, then 12 is less than 15, it checks root of left subtree again, then 12 is equal to 12, it finds in n times so, this is the O(n).



c) True, finding an array's maximum or minimum element can be done in constant time by comparing all elements using only if statements.

d) False, linked list is linear data structures. In the worst case, looking for the last node, searching all list elements is required. So, this takes n time -> O (n).

e) False, in each step, all elements of the sorted sub-array must, therefore, be shifted to the right so that the element to be sorted - which is smaller than all elements already sorted in each step - can be placed at the very beginning. So, n*(n-1)/2 = (n^2-n)/2 if constants and lower terms are eliminated, then => O (n^2).