Written Assignment 4

Q1. Draw the 11-entry hash table that results from using the hash function h(k) = (3k - 5) mod 11 to hash the keys 31, 45, 14, 89, 24, 95, 12, 38, 27, 16, and 25, assuming that collisions are handled in the following ways:

---

a) Linear probing

Solutions:

Continue until collision occurs. For 31, 31*3-5 = 88, 88 mod 11 = **0**

For 45, h(45) = 130 mod 11 = **9**

For 14, h(14) = 37 mod 11 = **4**

For 89, h(89) = 262 mod 11 = 9 => therefore in the table, there is collision for 45 and 89 so in a linear probing way, simply entry is h(k) + i, where i is integer so 9 + 1 is 10 and there is no collision for **10**.

Entry	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Keys	31				14					45	89

For 24, h(24) = 67 mod 11 = **1**

For 95, h(95) = 280 mod 11 = **5**

For 12, h(12) = 31 mod 11 = **9** => Collision occurs, in a same way, keep adding integer value until there is no occupied entry exists. In this case, **2** is not occupied since 9, 10, 0, 1 are occupied.

Entry	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Keys	31	24			14	95				45	89

Entry	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Keys	31	24	12		14	95				45	89

For 38, h(38) = 109 mod 11 = 10 => entry numbers of 10, 0, 1, 2 are occupied. **3** is not occupied.

For 27, h(27) = 76 mod 11 = 10 => entry numbers of 10, 0, 1, 2, 3, 4, 5 are occupied. **6** is not occupied.

For 16, h(16) = 43 mod 11 = 10 => entry numbers of 10, 0, 1, 2, 3, 4, 5 are occupied. **7** is not occupied.

For 25, h(25) = 70 mod 11 = 4 => entry numbers of 4, 5, 6, 7 are occupied. **8** is not occupied.

So the answer for a) is

Entry	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Keys	31	24	12	38	14	95	27	16	25	45	89

b) Double hashing using the secondary hash function h′(k) = 5 – (k mod 5)

Solutions:

Using result of previous question, h(31) = **0**, h(45) = **9**, h(14) = **4**.

For 89, h(89) = 9, which makes collision. h′(89) = 5 – (89 mod 5) = 1 => 9 + 1 = **10**, since there is no entry for 10.

Entry	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Keys	31				14					45	89

h(24) = **1,** h(95) = **5**

For 12, h(12) = 31 mod 11 = **9** => h′(12) = 5 – (12 mod 5) = 3 => 9 + 3 = 12. Since 12 does not exist, 12 mod 11 = 1, which is the entry of 1. It's already occupied so keep adding +3. 1+3 = 4, already occupied, 4+3 = **7**

Entry	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Keys	31	24			14	95		12		45	89

For 38, h(38) = **10** => h′(38) = 5 – (38 mod 5) = 2, 10 + 2 = 12 mod 11 = 1, occupied. 1 + 2 = **3**.

For 27, h(27) = **10** => h′(27) = 5 – (27 mod 5) = 3, 10 + 3 = 13 mod 11 = **2**.

For 16, h(16) = **10** => h′(16) = 5 – (16 mod 5) = 4, 10 + 4 = 14 mod 11 = 3 occupied. 3 + 4 = 7. Occupied. 7 + 4 = 11 mod 11 = 0 occupied. 0 + 4 = 4 occupied. 4 + 4 = **8**.

For 25, h(25) = 4 => h′(25) = 5 – (25 mod 5) = 5, 4 + 5 = 9 occupied. 9 + 5 = 14 mod 11 = 3 occupied. 3 + 5 = 8 occupied. 8 + 5 = 13 mod 11 = 2 occupied. 2 + 5 = 7 occupied. 7 + 5 = 12 mod 11 = 1 occupied. 1 + 5 = **6**

Entry	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Keys	31	24	27	38	14	95	25	12	16	45	89

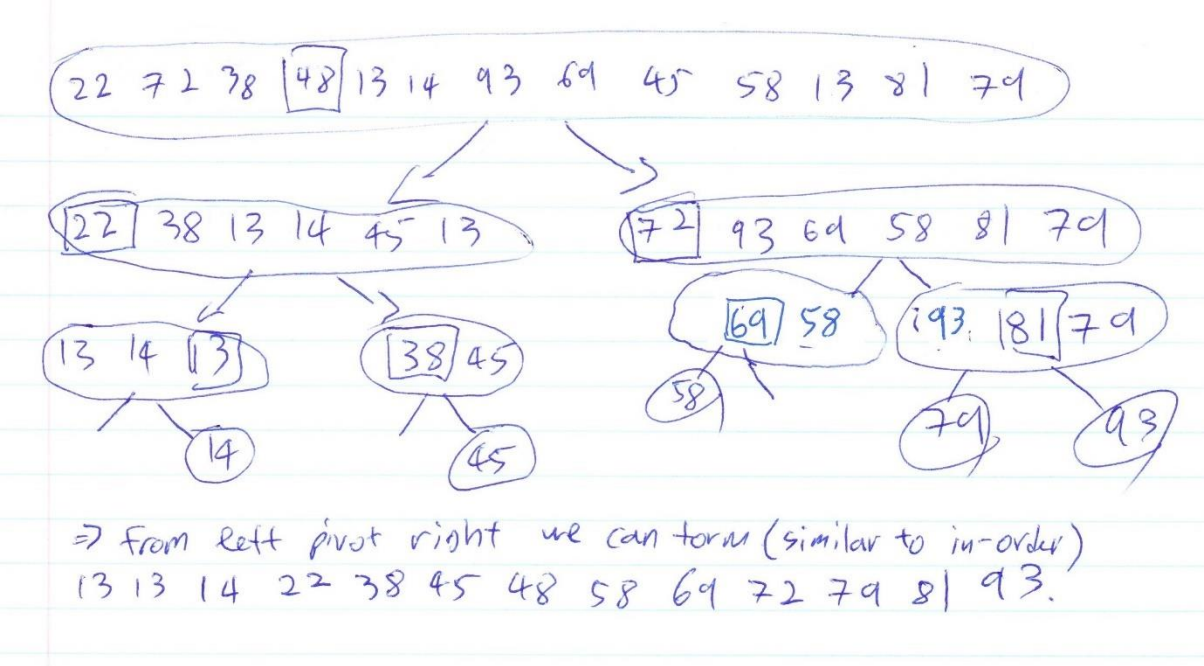
Q2. Answer the following questions:

a) Show (diagrammatically in the form of a divide-and-conquer tree, in the same format shown in your text book and slides) all the steps involved in the quick sort on the following input sequence: 22 72 38 48 13 14 93 69 45 58 13 81 79. You should sort the sequence in the non-decreasing order. Clearly show the pivots chosen at each node. (Note: Though in practice the pivot for each input is chosen at random, you may choose pivots here of your choice which make the tree balanced, i.e. only choose good pivots).

Solution:

The question asks to choose a good pivot, to make tree balanced.

To have balanced, I choose 48 to be a first pivot. In this way, left child and right child will have the same amounts of elements.



b) repeat part a) if the quick sort is performed in place.

Solutions:

Please note that blue color background means pivot, orange background color is less than pivot, and green background color is greater than pivot

22	72	38	48	13	14	92	69	45	58	13	81	79
22	72	38	48	13	14	92	69	45	58	13	81	79
22	13	38	48	13	14	92	69	45	58	72	81	79
22	13	38	48	13	14	45	69	92	58	72	81	79
22	13	38	45	13	14	48	69	92	58	72	81	79

22	13	38	45	13	14	48	69	92	58	72	81	79
22	13	38	45	13	14	48	69	92	58	72	81	79
22	13	14	45	13	38	48	69	92	58	72	81	79
22	13	14	13	45	38	48	69	58	92	72	81	79
13	13	14	22	45	38	48	69	58	72	92	81	79

13	13	14	22	45	38	48	69	58	72	92	81	79
13	13	14	22	38	45	48	58	69	72	79	81	92
13	13	14	22	38	45	48	58	69	72	79	81	92

In-place quick sort is completed

Q3. Answer the following questions:

a) Explain why Merge sort is the most suited for very large inputs (that do not fit inside memory) while quick sort and heap sort are not as suited. Note that these three sorting techniques have comparable time complexities.

Solutions:

Because for the merge sort, it's always dividing by 2 sections until there is only 1 (or 0) element left. After that compare 1 element with another element and merge them. After that sort and merging other sorted elements.

Therefore, merge sort is whether the inputs are big or not, it always keeps running left to the right and merging them.

However, for heap sort, searching algorithm is required and if it is large input, it needs to search whole heap tree. Similarly, quick sort requires to traverse whole sequences (or array).

In conclusion, merge sort is the most suited for very large inputs since it does not need to search or traverse the sequence and running from all elements from left to the right.

b) Can Merge sort be performed in place? Explain your understanding.

Solution:

No, In-place merge sort is not possible because when it all elements are divides to 1 element array and when it tries to merge, it needs the double size of array. Since size of array are variable and merged with double sized array, it cannot be done in place.

There would have another implementation of merge sort and it might run as in-place during merge step; however, it requires some restrictions and we did not cover in class. As well, the running algorithm is also different from O(nlog(n)) if we implement this method.

Therefore, the answer is **no**, merge sort cannot be performed in place since it requires auxiliary array.

Q4. Show the steps in performing radix sort on the following input: 69 81 45 48 13 38 58 9 14 22 93 79 5 72. You should sort the sequence in the non-decreasing order.

Solutions:

69	81	45	48	13	38	58	9	14	22	93	79	5	72

We have learned how to radix-sort with binary number in class. In a similar way, decimal number is also possible to use radix sort. Firstly, bucket sort on last digit in range [0-9] then,

81	69	45	48	13	38	58	9	14	22	93	79	5	72
81	22	72	69	45	48	13	38	58	9	14	93	79	5
81	22	72	13	93	69	45	48	38	58	9	14	79	5
81	22	72	13	93	14	69	45	48	38	58	9	79	5
81	22	72	13	93	14	45	5	69	48	38	58	9	79
81	22	72	13	93	14	45	5	48	38	58	69	9	79
81	22	72	13	93	14	45	5	48	38	58	69	9	79
81	22	72	13	93	14	45	5	48	38	58	69	9	79

As we observed, 81 is in first element for now since 1 is the largest number. For 22 and 72, since 22 previously has lower index number than 72, 22 comes first and 72 is followed. The same rule applies for whole sequence.

Now, we need to do bucket sort on first digit in range [0-9], 9 is actually 09.

5	9	81	22	72	13	93	14	48	38	58	69	9	79
5	9	13	14	81	22	72	93	45	48	38	58	69	79
5	9	13	14	22	81	72	93	45	48	38	58	69	79
5	9	13	14	22	38	81	72	93	45	48	58	69	79
5	9	13	14	22	38	45	48	81	72	93	58	69	79
5	9	13	14	22	38	45	48	58	81	72	93	69	79
5	9	13	14	22	38	45	48	58	69	81	72	93	79
5	9	13	14	22	38	45	48	58	69	72	79	81	93
5	9	13	14	22	38	45	48	58	69	72	79	81	93
5	9	13	14	22	38	45	48	58	69	72	79	81	93

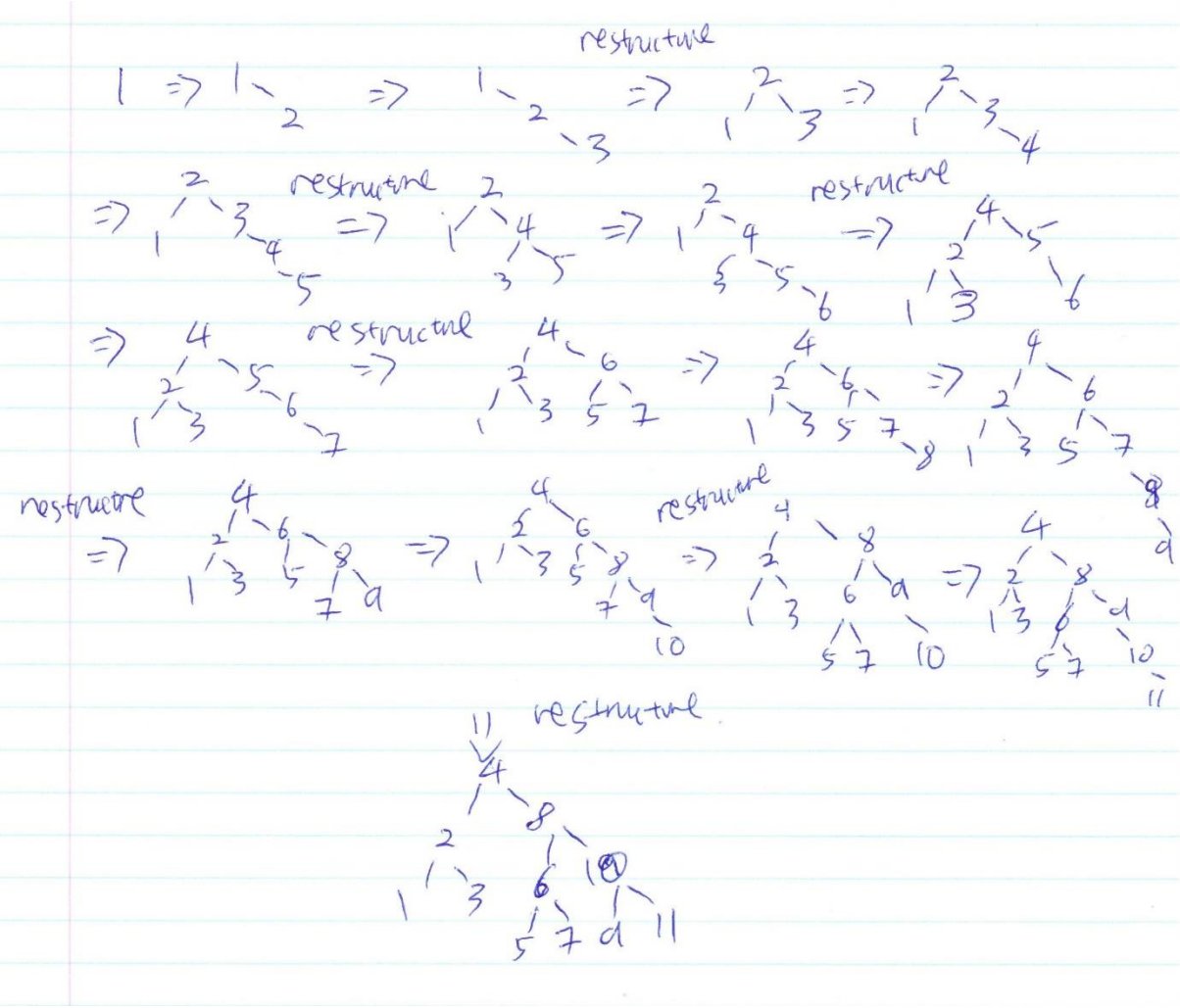
In this way, radix sort is applied and the sequence is non-decreasing order as well.

Q5. Answer the following questions:

a) Consider the following sequence of keys inserted into an initially empty AVL tree T in this specific order: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11. Draw the AVL tree T after each insertion

Solutions:

Please note that an insertion is made on the right most child all the times since it inserts increasing order. The newly added value is always the largest number so that is why insertion is made always right-most child. As it occurs imbalance, restructure is also applied.



b) Is an AVL tree for a specific set of inputs unique? If your answer is yes, then explain why it is so. If your answer is no then demonstrate using the set of inputs in part a) above.

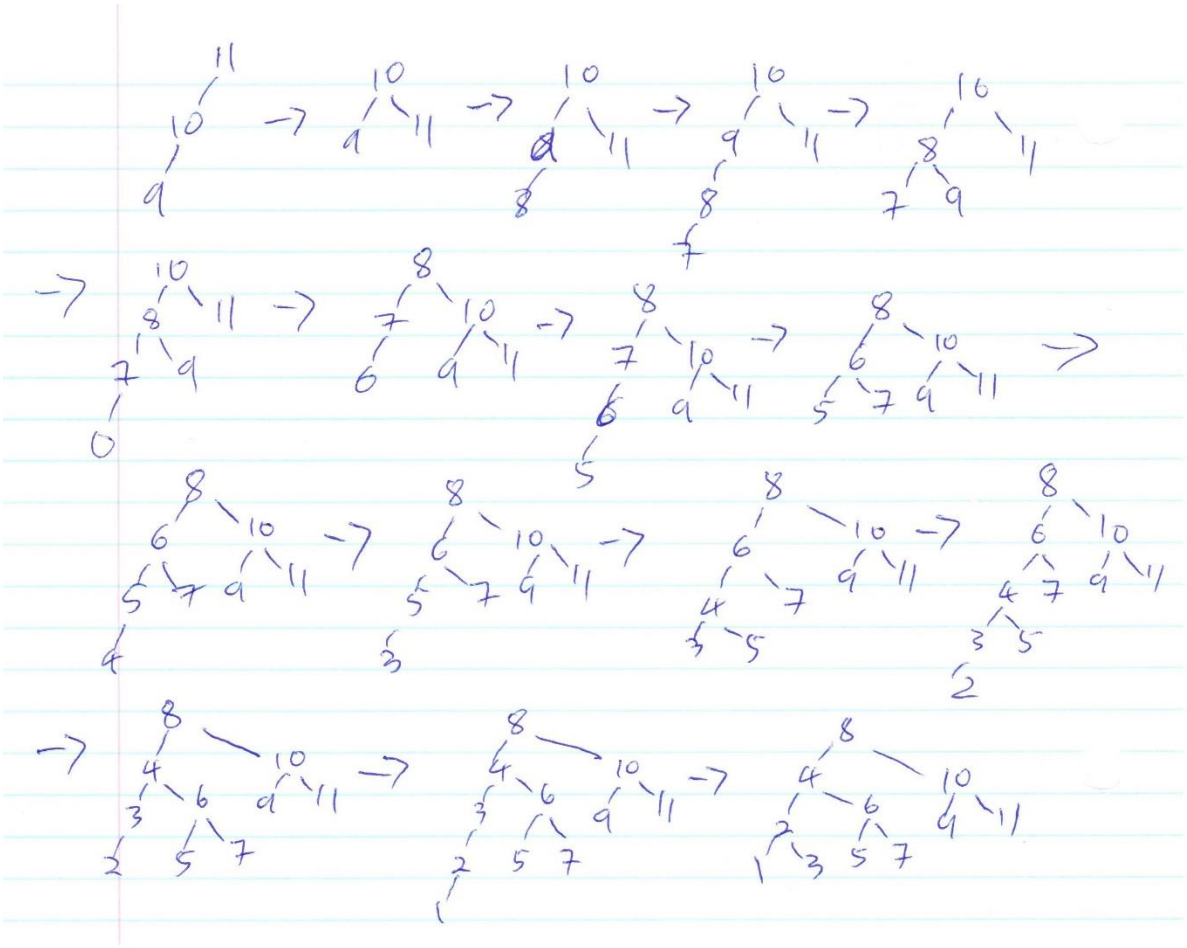
Solutions:

No, specific set of inputs does not mean that the inputs will be inserted in the same order. Therefore, for example, if we use the same input, (1,2,3,4,...,11) in a reverse order, AVL tree will be different.

Please note that specific set of inputs do not mean that will be inserted in the same order. If it's the case,

it will have a unique AVL tree.

If we insert 11, 10, 9, … , 1, AVL tree has different structure. Please check the image below.



Therefore, AVL tree in (a) and (b) are not the same, and still, there is another AVL tree which uses the same set of inputs (in a different order), so the answer is **No.**

Q6. Answer the following questions:

a) A team of biologists keeps information about DNA structures in an AVL tree using as key the specific weight (an integer) of a structure. The biologists routinely ask questions of the type: "Are there any structures in the tree with specific weights between a and b (both inclusive)", and they hope to get an answer as soon as possible. Design an efficient algorithm that given integers a and b returns true if there is a key x in the tree such that $a \leq x \leq b$, and returns false if no such key exits. Describe your algorithm in pseudo-code.

Solution:

Algorithm isExist (treeAVL, a, b):

Input: a and b are a key (weight) the biologists wants to find. a and b are integer. treeAVL is given tree to be searched.

Output: If there exists a key between a and b (inclusive) return true; otherwise, if it does not exist, return false.

```
Node t = root of treeAVL;

while n is null:

        if a <= t.getKey() <= b:

                return true; // it means there is a value exists in the range between a and b

        else if a > t.getKey():

                t = t.right(n);

        else:

                t = t.left(n);

return false; // At this point, n does not exist and it means whether a is larger than largest value in AVL tree or b is smaller than the smallest value in AVL tree.
```

---

b) What (and why) is the time complexity of the algorithm?

Solution:

Since it visits all the children node from the root (to external node), and since AVL is balanced binary search tree and balanced binary search tree has height is less than 2log(n), the time complexity of given algorithm is **O(log(n)).**

To explain why AVL tree height is log(n), if the root is N(h) where N denotes number of nodes, in the worst

case, one of the child can have N(h-1) and another child can have N(h-2).

N(h) = N(h-1) + N(h-2) + 1 (root itself needs to be incremented by 1)

and we know that N(h-1) = N(h-2)+N(h-3)+1

N(h) = 2N(h-2) + N(h-3) + 1 + 1 ...

As we observe N(h) > 2N(h-2), Then 2N(h-2) > 2*2N(h-4) > 2*2*2N(h-6) > ... > $2^{h/2}$.

So we get N(h) > $2^{h/2}$, apply both sides $\log_2$, We get log(N(h) > h/2, h < 2log(N(h))

h is height and 2log(N(h)) = 2log(n).

T(n) = 2log(n) + C <= clog(n) where c = (2+2^C), n_0>= 1 we can say the time complexity of given algorithm is **O(log(n))** since it only visits number of times of heights.

Note that in the algorithm that I make, there is only one while loop and only visits its child (until there is no more child).

Q7. Describe an efficient algorithm for computing the height of a given AVL tree. Your algorithm should run in time O(log n) on an AVL tree of size n. In the pseudocode, use the following terminology: T.left, T.right, and T.parent indicate the left child, right child, and parent of a node T and T.balance indicates its balance factor (-1, 0, or 1). For example if T is the root we have T.parent=nil and if T is a leaf we have T.left and T.right equal to nil. The input is the root of the AVL tree. Justify correctness of the algorithm and provide a brief justification of the runtime.

Solution:

Algorithm computeHeight(root):

Input: a node which wants to be calculated. When it calls, root of the AVL tree is called.

Output: height of current

```
If root.left and root.right equal to nil OR if root is nil then
        return 0
If (T.balance >=0):
        Return computeHeight(root.left) + 1;
else:
        return computeHeight(root.right) +1;
```

Explanation of algorithm : If both of root.left and root.right are nil (which means it is a leaf), current height is 1 so return 1. If it is not the case, keep recursively calling from root to the leaf. At the end, it will give either leftHeight or rightHeight depending on balance factor, since it's AVL tree, either they're equal or one of them will be larger by 1. Because there is balance factor, it will choose always larger path so whichever larger height from child will be chosen.

Since AVL tree is balanced binary search tree, the algorithm is very similar to find for height of a binary search tree except there is balance factor. If we implement exactly same algorithm as finding height for binary search tree, the time complexity is O(n) and it is not O(log(n)) so using balance factor is actually important to reduce time complexity.

As we solved in Q6, balanced binary search tree for finding height time complexity is **O(log(n))** since height of AVL tree is less than 2log(n) which results O(log(n)). To have detailed explanation, please refer to Question 6 above (I have proved height of AVL tree is less than 2log(n)).

To justify, my algorithm actually takes O(log(n)), in the worst case, assume that left child has h-1 and right child has h-2. By using balance factor, left child will be chosen. For height h-1, it has left child h-2, and right

child h-3, then h-2 is choosing by that. In this way, as I mentioned in Q6, h < 2log(n).

T(n) = T(N(h)) = T(N(h-1)) + T(N(h-2) + T(N(h-3)) + … T(1) + C. (In the worst case, balance factor is always 1 or -1).

It means that total number of T(x) is {h-1,h-2,h-3,h-4, … 1}. Since h <2 log(n) we can say that total number of operations is 2log(n)*C because each T(x) < C, so C + C + C + … C (2log(n) times of C because there is another constant time C from T(n)) < 2log(n)*C

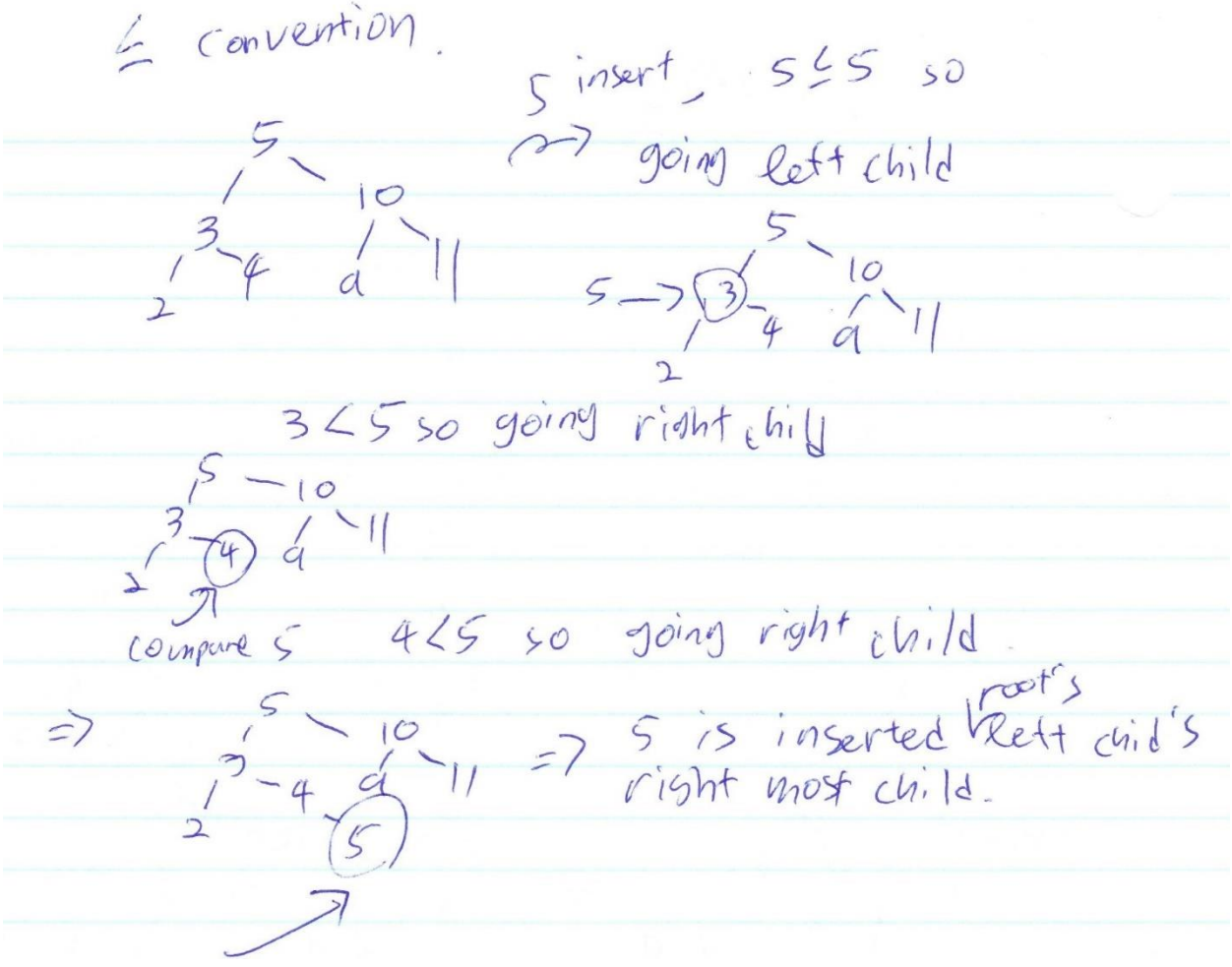
Therefore T(n) < 2log(n)*C <= k*log(n) where k = 2C n_0 >= 1. Then, The time complexity is **O(log(n))**.

Q8. Given a balanced binary search tree that somehow allows duplicates populated by a sequence S of n elements on which a total order relation is defined, describe an efficient algorithm for determining whether there are two equal elements in S. What is the running time of your algorithm?

Solution:

If there are duplicates, it means that either the duplicate value exists left child or right child. However, we know that left child < root < right child so if there is duplicate, it means that the value will be stored left child's right most leaf or right child's left most leaf depending on the convention.

Because if convention applies less than equal, duplicated value goes to the left child from the root. After that, the values going right since all elements of left child from root are smaller than root itself, so it will place in the root's left child's right most child. If convention is greater than equal, then, the same rule applies.



As it is observed, single rotation for duplicate place will only occur when another duplicate is inserted; otherwise, double rotation will occur but in terms of root's view, even if root has been replaced and double rotation is on, it will be still its right most child of left child from this node (for <= convention case). Thus, we can conclude that in-order traversal will be listed all elements in increasing order including duplicates.

Because for root, left child's most right child is lastly listed before the root is visited in <= convention and right child's most left child is firstly listed after the root is visited in >= convention.

For example, for the image above, in-order traversal will be

2	3	4	5	5	9	10	11

The algorithm is following

Algorithm isDuplicated(Node root):

Input: root of the checking tree.

Output: return true if there is duplicate; otherwise return false

```
Stack<node> stk;
Queue q;
Node temp = root;
While temp is not null:
        stk.push(temp);
        temp = temp.left;
while stk.size() > 0:
        temp = stk.pop();
        q.enqueue(temp.key);
        if temp.right != null:
                while temp is not null:
                        stk.push(temp)
                        temp = temp.right;
// Making a queue to have ordered list is done at this point. (Phase 1), Using with in-order traversal.
temp1 = q.dequeue();
while q.size()>0:
        temp2 = q.dequeue();
        if temp1 is equal to temp2:
```

```
                return true;
        temp1 = temp2;
return false;
// Comparing values (phase 2)
```

Explanation: For this algorithm, there is 2 phases. First phase constructs queue to have ordered list with in-order traverse. As it observed, it will go through all the nodes from while loop. We can denote T(n) = T(n-1) + C = T(n-2) + 2C = … = C*n.

After that, it will compare in ordered list. As I explained above, the duplicates will be sorted if we use in-order traverse, so we just need 2 elements in neighbor. That also takes T(n) = C*n since queue size is n.

Total time complexity is T(n) = C*n + C*n = 2C*n <= cn where c = 2C+1 n_0>= 1; therefore, time complexity is **O(n)**.

Q9. Given the following hash functions, what is the probability that two randomly chosen

keys collide? (justify your answer).

a) h(k) = k mod 13

h(k) is determined by modular 13. That means there are 13 possible entries and the probability of collision is **1/13** since there are 13 entries. For example, if k is 1, then, 2 to 13 will not cause collision but 14 will cause collision. 15 to 26 will not cause collision but 27 will cause collision. And so on.

b) h(k) = 2k mod 8

2k mod 8 means there are only modular value of 0, 2, 4, 6 and there is no possible way to get 1, 3, 5, 7 as modular of 8. Any integer multiplied by 2 is even number and even number mod 8 is only possible to have 0, 2, 4, 6. Even if there are 8 entries, but we only use 4 of them, the probability of collision is **1/4**

c) h(k) = (3k+1) mod 12

Similar to question b, (3k + 1) mod 12 means there are only modular value of 1, 4, 7, 10. For example, for modular value of 0, it requires to have (3k+1) to be 12*i where i is integer but 12*i = 4*3*i and it is impossible to have the modular value of 0. So on 12, there are only 4 possible modular values so there are 12 entries, but only 4 are using so the probability of collision is **1/4**

d) h(k) = (((5k+7) mod 13)+1) mod 17

Firstly, we need to get (5k+7) mod 13. Forget about the +7 and just checking 5k mod 13, since both 5 and 13 are prime numbers, all the modular values from 0 to 12 are possible. That means greatest common divisor in this case is 1. At that point, adding 7 will not change the range of modular.

We can simply show that there is all modular value of 0 to 12.

k	0	1	2	3	4	5	6	7	8	9	10	11	12
modular	7	12	4	9	1	6	11	3	8	0	5	10	2

Secondly, modular 0 to 12 + 1 will be 1 to 13. 1 to 13 mod 17 is literally the same as 1 to 13 (since modular of 13's maximum value is 12 and + 1 is 13 which is the maximum value). That means there is 17 entries but only 13 entries are using. The probability of collision is **1/13**

e) h(k) = (k mod 6)^2

k mod 6 has 6 modular values, 0 to 5. Regardless of entry size, there is only 6 possible h(k) values 0, 1, 4, 9, 16, 25. Therefore, the probability of collision is 1/6