COMP352 THEORY ASSIGNMENT 3

ARASH SHAFIEE (STUDENT ID: 40278142)

Question 1) Is it possible to have a single tree that satisfies the following traversals:

**Inorder** : **X H C K Y A S M W Q D R L**

**Postorder**: **X C H Y S A W M D Q L R K**

According to postorder traversal K is the root of the tree. In an inorder traversal, the root separates the traversal into two subarrays: one representing the left subtree and the other representing the right subtree. Here, 'K' divides the inorder traversal into 'X H C' for the left subtree and 'Y A S M W Q D R L' for the right subtree. So, the answer is **YES** there existing a single tree which satisfies the both traversal.

Question 2) Assume that linear probing is used for hash-tables. To improve the time complexity of the operations performed on the table, a special AVAILABLE object is used to mark a location when an item is removed from the location. Assuming that all keys are positive integers, the following two techniques

were suggested instead of marking the location as AVAILABLE:

i) When an entry is removed, instead of marking its location in the table as AVAILABLE, indicate the key in the location as the negative value of the removed key (e.g., if the removed key was 16, indicate the key as -16). Searching for an entry with the removed

key would then terminate once a negative value of the key is found (instead of continuing to search if AVAILABLE is used).

ii) Instead of using AVAILABLE, find a key in the table that should have been placed in the location of the removed entry, then place that key (the entire entry of course) in that location (instead of setting the location as AVAILABLE). The motive is to find the key faster since it is now in its hashed location. This would also avoid the dependence on the AVAILABLE object.

Will either of these approaches have an advantage? You should analyze in terms of both time and space complexities. Additionally, will any of these approaches result in misbehaviors (in terms of functionalities)? If so, explain clearly through illustrative examples.

Answer ) The first approach uses negative keys to mark removed entries. This can potentially speed up search operations, as the search will terminate when a negative value is encountered. However, this may cause issues if the application needs to store negative keys or iterate over the entire hash table, as the negative values could be misinterpreted as removed entries.

When searching for a key, the search will terminate as soon as a negative value is encountered, which can be more efficient than the traditional "AVAILABLE" approach where the search would continue until an empty slot is found. However, the **time complexities** for insertion and deletion remain the same as the traditional linear probing approach, which are O (1) on average and O(n) in the worst case.

In terms of **space complexity**, the negative key approach does not introduce any significant changes compared to the traditional approach. The negative values occupy the same amount of space as the "AVAILABLE" marker, so the overall space complexity remains the same.

The second approach relocates displaced keys to their correct hashed locations. This can also improve search times on average by reducing collisions. But it introduces the risk of infinite loops during the search process if the hash function or key distribution is not ideal. It may also increase memory usage by requiring additional space to accommodate the relocated entries.

The second approach, relocating displaced keys to their respective hashed locations, can also provide a potential time complexity advantage for search operations in the average case. By placing the displaced keys in their correct positions, it can lead to fewer collisions and shorter probe sequences. However, the **time complexities** for insertion and deletion are still the same as the traditional linear probing approach.

Regarding **space complexity,** the relocate displaced keys approach may require additional space to accommodate the relocated entries, leading to a potentially higher memory usage compared to the traditional approach. The exact space complexity would depend on the specific implementation.

In summary, once again both techniques have the potential to result in misbehaviors that could impact functionality. The negative key approach may cause issues if the application needs to store negative keys, as they could be misinterpreted as removed entries.

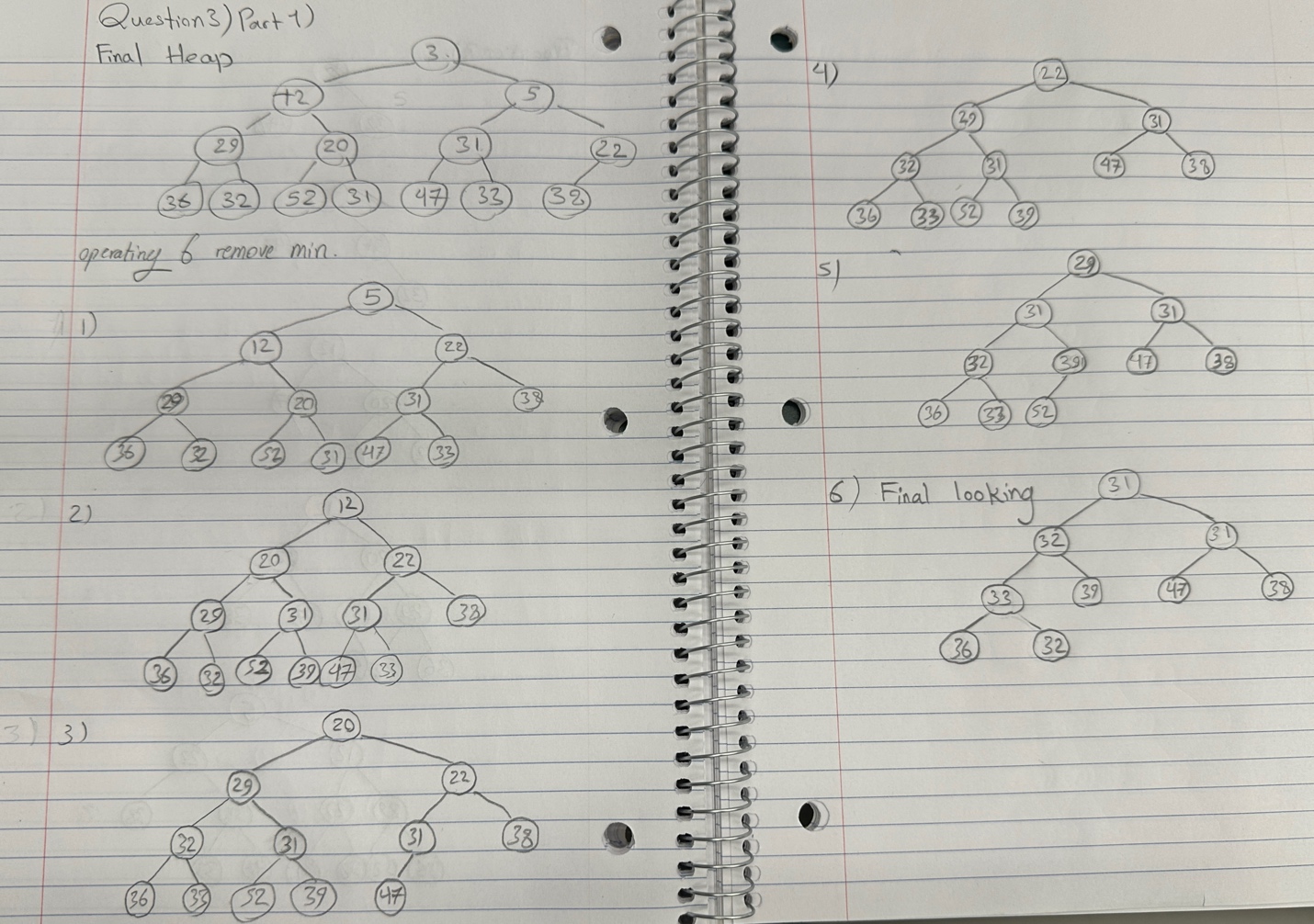
The relocate displaced keys approach introduces the risk of infinite loops during the search process, if the hash function or key distribution is not well-designed. It may also increase memory usage by requiring additional space to accommodate the relocated entries. These misbehaviors could result in unpredictable behavior, incorrect results, and even runtime errors.

Question 3)

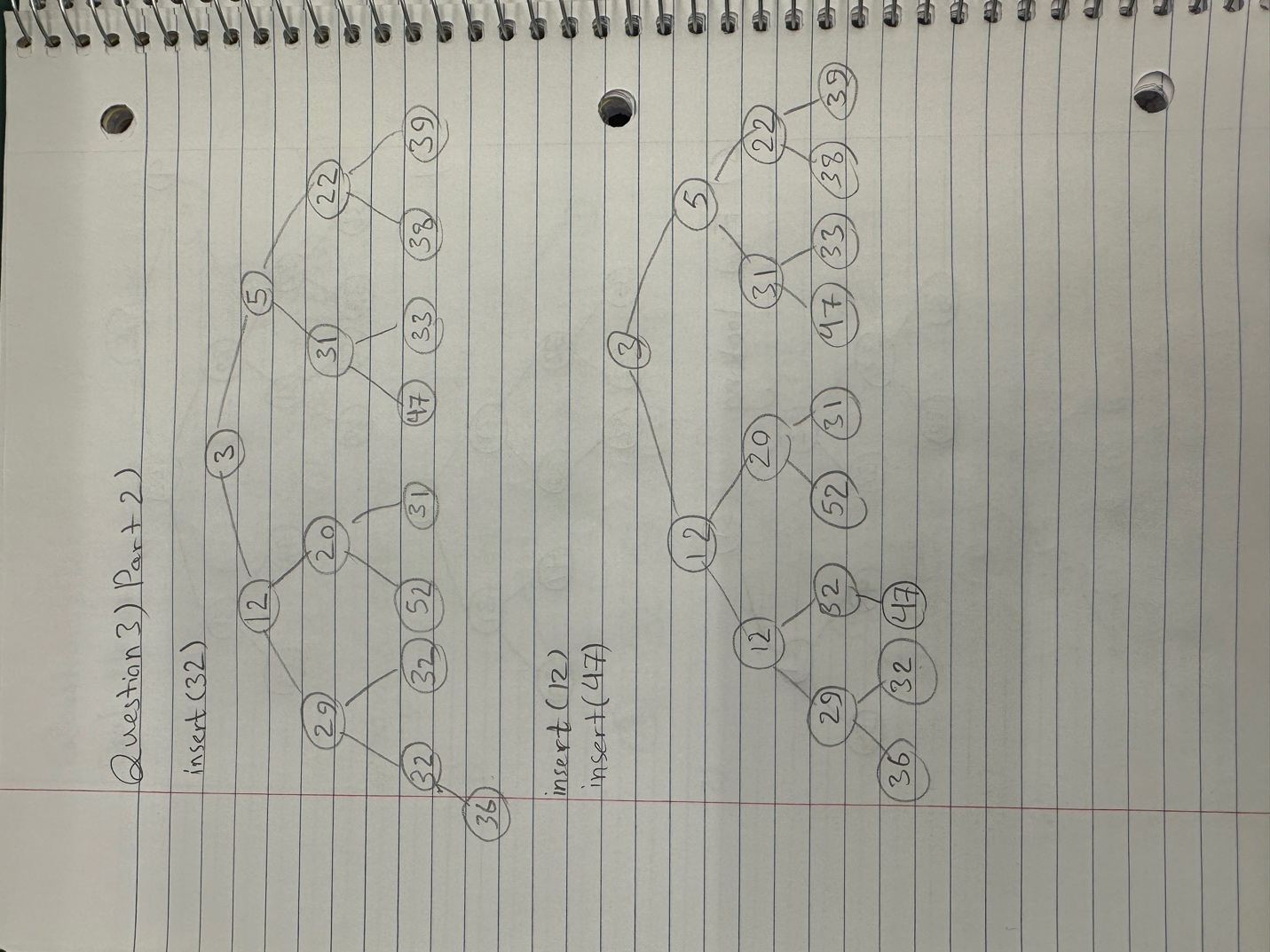
i )Draw the min-heap that results from the bottom-up heap construction algorithm on the following list of values:

32, 12, 47, 31, 20, 22, 38, 36, 29, 52, 5, 31, 33, 3, 39.

Starting from the bottom layer, use the values from left to right as specified above. Show immediate steps and the final tree representing the min-heap. Afterwards perform the operation removeMin 6 times and show the resulting min-heap after each step.



ii) Create again a min-heap using the list of values from the above part (i) of this question but this time you have to insert these values step by step (i.e. one by one) using the order from left to right (i.e. insert 32, then insert 12, then 47, etc.) as shown in the above question. Show the tree after each step and the final tree representing the min-heap.



Question 4)

Assume a hash table utilizes an array of 13 elements and that collisions are handled by separate chaining. Considering the hash function is defined as: h(k) = k mod 13.

i ) Draw the contents of the table after inserting elements with the following keys:

56, 472, 352, 140, 217, 120, 18, 21, 182, 204, 91, 93, 178, 78, 70, 33, 51, 90.

ii) What is the maximum number of collisions caused by the above insertions ?

|  |  |
| --- | --- |
| index | Chain |
| 0 | [182,91,78] |
| 1 | [352] |
| 2 | [93] |
| 3 | [120] |
| 4 | [56, 472] |
| 5 | [18, 70] |
| 6 | [] |
| 7 | [33] |
| 8 | [21] |
| 9 | [217,204,178] |
| 10 | [140] |
| 11 | [] |
| 12 | [51,90] |

ii) Maximum number of collisions : indices 0 and 9 which gives 2 collisions.

Question 5)

Bubble sort is a sorting algorithm that works as follows: Go through the input list/array element by element, comparing the current element with the one that follows it (i.e. compare A[i] with A[i+1]), the swap the two values if they are not sorted. This process is repeated through the list until no swaps can be performed during a pass, meaning that the list is already sorted. Given the following elements:

23 32 72 76 22 73 40 30 20 60 16 74 28 14

i ) Show the needed steps for sorting these values into ascending order using Bubble Sort.

ii) Does Bubble Sort have better time complexity compared to Selection Sort? Explain.

1st attempt

Compare 23 and 32: no swap

Updated list : 23 32 72 76 22 73 40 30 20 60 16 74 28 14

Compare 32 and 72: no swap

Updated list : 23 32 72 76 22 73 40 30 20 60 16 74 28 14

Compare 72 and 76: no swap

Updated list : 23 32 72 76 22 73 40 30 20 60 16 74 28 14

Compare 76 and 22: swap

Updated list : 23 32 72 22 76 73 40 30 20 60 16 74 28 14

Compare 76 and 73: swap

Updated list : 23 32 72 22 73 76 40 30 20 60 16 74 28 14

Compare 76 and 40: swap

Updated list : 23 32 72 22 73 40 76 30 20 60 16 74 28 14

Compare 76 and 30: swap

Updated list : 23 32 72 22 73 40 30 76 20 60 16 74 28 14

Compare 76 and 20: swap

Updated list : 23 32 72 22 73 40 30 20 76 60 16 74 28 14

Compare 76 and 60: swap  
Updated list: 23 32 72 22 73 40 30 20 60 76 16 74 28 14

Compare 76 and 16: swap  
Updated list: 23 32 72 22 73 40 30 20 60 16 76 74 28 14

Compare 76 and 74: swap  
Updated list: 23 32 72 22 73 40 30 20 60 16 74 76 28 14

Compare 76 and 28: swap  
Updated list: 23 32 72 22 73 40 30 20 60 16 74 28 76 14

Compare 76 and 14: swap  
Updated list: 23 32 72 22 73 40 30 20 60 16 74 28 14 76

We see that 76 is now the greatest number in array is sorted in the right-hand side

2nd attempt

Compare 23 and 32: no swap  
Updated list: 23 32 72 22 73 40 30 20 60 16 74 28 14 76

Compare 32 and 72: no swap  
Updated list: 23 32 72 22 73 40 30 20 60 16 74 28 14 76

Compare 72 and 22: swap  
Updated list: 23 32 22 72 73 40 30 20 60 16 74 28 14 76

Compare 72 and 73: no swap  
Updated list: 23 32 22 72 73 40 30 20 60 16 74 28 14 76

Compare 73 and 40: swap  
Updated list: 23 32 22 72 40 73 30 20 60 16 74 28 14 76

Compare 73 and 30: swap  
Updated list: 23 32 22 72 40 30 73 20 60 16 74 28 14 76

Compare 73 and 20: swap  
Updated list: 23 32 22 72 40 30 20 73 60 16 74 28 14 76

Compare 73 and 60: swap  
Updated list: 23 32 22 72 40 30 20 60 73 16 74 28 14 76

Compare 73 and 16: swap  
Updated list: 23 32 22 72 40 30 20 60 16 73 74 28 14 76

Compare 73 and 74: no swap  
Updated list: 23 32 22 72 40 30 20 60 16 73 74 28 14 76

Compare 74 and 28: swap  
Updated list: 23 32 22 72 40 30 20 60 16 73 28 74 14 76

Compare 74 and 14: swap  
Updated list: 23 32 22 72 40 30 20 60 16 73 28 14 74 76

We see that 74 which is the second greatest number in array is sorted in the right-hand side

3rd attempt

Compare 23 and 32: no swap  
Updated list: 23 32 22 72 40 30 20 60 16 73 28 14 74 76

Compare 32 and 22: swap  
Updated list: 23 22 32 72 40 30 20 60 16 73 28 14 74 76

Compare 32 and 72: no swap  
Updated list: 23 22 32 72 40 30 20 60 16 73 28 14 74 76

Compare 72 and 40: swap  
Updated list: 23 22 32 40 72 30 20 60 16 73 28 14 74 76

Compare 72 and 30: swap  
Updated list: 23 22 32 40 30 72 20 60 16 73 28 14 74 76

Compare 72 and 20: swap  
Updated list: 23 22 32 40 30 20 72 60 16 73 28 14 74 76

Compare 72 and 60: swap  
Updated list: 23 22 32 40 30 20 60 72 16 73 28 14 74 76

Compare 72 and 16: swap  
Updated list: 23 22 32 40 30 20 60 16 72 73 28 14 74 76

Compare 72 and 73: no swap  
Updated list: 23 22 32 40 30 20 60 16 72 73 28 14 74 76

Compare 73 and 28: swap  
Updated list: 23 22 32 40 30 20 60 16 72 28 73 14 74 76

Compare 73 and 14: swap  
Updated list: 23 22 32 40 30 20 60 16 72 28 14 73 74 76

We see that 73 which is the third greatest number in array is sorted in the right-hand side

.

.

.

We continue these attempts until the list is sorted (13 attempts is needed for this process and technically one final pass to make sure it is sorted but nothing will happen)

The final sorted array:

14 16 20 22 23 28 30 32 40 60 72 73 74 76

Answer ii) Bubble Sort and Selection Sort both have a time complexity of O(n2) in the average and worst cases, making them inefficient for large datasets. However, Bubble Sort can have a better performance in the best case (O(n)) when the list is nearly sorted, as it can terminate early if no swaps are needed in a pass. Selection Sort, on the other hand, always performs a fixed number of comparisons, regardless of the list’s initial order.

Question 6) Show the steps that a radix sort takes when sorting the following array of 3-tuple Integer keys (notice

that each digit in the following values represent a key):

**5,4,3 3,5,6 2,9,5 6,9,2 4,9,1 9,4,7 7,8,3 9,9,2 4,7,2 1,8,2 2,6,4**

Step 1: Sorting the array based on the right most digit

Bucket 1: **4,7,2 1,8,2 2,6,4**

Bucket 2: **2,9,5 5,4,3 6,9,2**

Bucket 3: **7,8,3 9,4,7 9,9,2**

Bucket 4: **3,5,6 4,9,1**

Step 2: Sorting the array based on the middle digit

Bucket 1: **4,9,1 1,8,2 2,6,4**

Bucket 2: **4,7,2 6,9,2 7,8,3**

Bucket 3: **2,9,5 5,4,3 9,4,7 9,9,2**

Step 3: Sorting the array based on the left most significant digit

Bucket 1: **1,8,2 2,6,4 2,9,5**

Bucket 2: **3,5,6 4,7,2 4,9,1**

Bucket 3: **5,4,3 6,9,2 7,8,3 9,4,7 9,9,2**

The final look of the array:

**1,8,2 2,6,4 2,9,5 3,5,6 4,7,2 4,9,1 5,4,3 6,9,2 7,8,3 9,4,7 9,9,2**

Question 7) Assume an *open addressing* hash table implementation, where the size of the array is *N* = 19, and that *double hashing* is performed for collision handling. The second hash function is defined as: *d(k) = q – k* *mod q,* where *k* is the key being inserted in the table and the prime number *q* is = 7. Use simple modular operation (*k mod N*) for the first hash function.

i) Show the content of the table after performing the following operations, in order:

**put(45), put(25), put(12), put(61), put(38), put(88), remove(12), put(39), remove(61),put(18), put(29), put(29), put(35).**

i) What is the size of the longest cluster caused by the above insertions?

ii)What is the number of occurred collisions as a result of the above operations?

iii) What is the current value of the table’s *load factor*?

* Array size : 19
* First hash function : h(k) = k mod N
* Second hash function : d(k)= 7 – k mod 7

Implement it for every value we have:

|  |  |  |
| --- | --- | --- |
| value | h(k) | d(k) |
| 45 | 7 | 4 |
| 25 | 6 | 3 |
| 12 | 12 | 2 |
| 61 | 4 | 2 |
| 38 | 0 | 4 |
| 88 | 12 | 3 |
| 39 | 1 | 3 |
| 18 | 18 | 3 |
| 29 | 10 | 6 |
| 35 | 16 | 7 |

So, on the hash-map we have :

|  |  |
| --- | --- |
| index | value |
| 0 | 38 |
| 1 | 39 |
| 2 | empty |
| 3 | empty |
| 4 | Deleted slot |
| 5 | empty |
| 6 | 25 |
| 7 | 45 |
| 8 | empty |
| 9 | empty |
| 10 | 29 |
| 11 | empty |
| 12 | Deleted slot |
| 13 | empty |
| 14 | empty |
| 15 | 88 |
| 16 | 35 |
| 17 | empty |
| 18 | 18 |

Answer ii) Only one collision occurred during the entire sequence of operations, when inserting 88 (position 12 was initially occupied).

Answer iii) number of elements/ table size = 8/19 which is approximately 0.421 so the load factor is 0.421