Answers Final Exam Summer

MCQs

1. Which sorting algorithm works by repeatedly inserting an element into a sorted portion of the list? Answer: b) Insertion Sort
2. Which sorting algorithm is most efficient for sorting a large collection of elements? Answer: d) Quick Sort
3. In hashing, what converts a key into an array index? Answer: d) Hash function
4. Which data structure is implemented by hashing in Python? Answer: d) Dictionary
5. Which collision resolution technique in hashing uses linked lists to store collided elements? Answer: c) Chaining
6. In insertion sort, what is the maximum number of comparisons required to sort a list of n elements in the worst case? Answer: b) n^2
7. In merge sort, what is the time complexity for merging two sorted arrays of sizes n and m? Answer: c) O(n + m)
8. What should be added at the "Incomplete code" line to handle collision in linear probing? Answer: a) hash\_value = (hash\_value + 1) % len(hash\_table)
9. What is the missing code in the Quick Sort algorithm? Answer: a) pi = partition(arr, low, high)
10. Suppose you have the following list of numbers to sort: [11, 7, 12, 14, 19, 1, 6, 18, 8, 20] which list represents the partially sorted list after three complete passes of selection sort? Answer: c) [11, 7, 12, 14, 1, 6, 8, 18, 19, 20]
11. Suppose you are given the following set of keys to insert into a hash table that holds exactly 11 values: 113 , 117 , 97 , 100 , 114 , 108 , 116 , 105 , 99. Which of the following best demonstrates the contents of the hash table after all the keys have been inserted using linear probing? Answer: b) 99, 100, \_\_, 113, 114, \_\_, 116, 117, 105, 97, 108
12. For a successful search using open addressing with linear probing, the average number of comparisons is approximately. Answer: c) 1/2(1+1/(1-λ))
13. For an unsuccessful search using open addressing with linear probing, the average number of comparisons is approximately. Answer: d) 1/2(1+〖(1/(1-λ))〗^2)
14. Which collision resolution technique in hashing uses linked lists to store collided elements? Answer: c) Chaining
15. Suppose you have the following list of numbers to sort: [15, 5, 4, 18, 12, 19, 14, 10, 8, 20] which list represents the partially sorted list after three complete passes of insertion sort? Answer: a) [4, 5, 12, 15, 14, 10, 8, 18, 19, 20]

Top of Form

Q2. The provided **fast\_count** method is an attempt to count the number of elements in a doubly-linked list with a sentinel head node. Let's analyze the time complexity of this method:

pythonCopy code

def fast\_count(self): n = 0 h = self.head.next while h is not self.tail: n, h = n + 2, h.next.next if h.next is not self.head: n, h = n - 1, h.prev.prev return n

In this method, we use a single pointer **h** to traverse the list in a step-wise manner. The pointer starts at the first element after the sentinel head node and moves through the list, incrementing the count **n** as it goes.

The loop condition **while h is not self.tail** ensures that we traverse the list until we reach the tail sentinel node, indicating the end of the list.

Inside the loop:

* **n** is incremented by 2 for each iteration (**n, h = n + 2, h.next.next**). This corresponds to advancing **h** by two nodes in each iteration.
* However, there's a conditional check: **if h.next is not self.head**. If this condition is met, it means that **h** is not at the last node of the list, and it has a non-null **next** pointer. In this case, **n** is decremented by 1, and **h** is moved back by two nodes (**n, h = n - 1, h.prev.prev**). This corresponds to undoing the previous advancement of **h**.

Let's break down the time complexity:

1. The loop iterates as long as **h** is not equal to **self.tail**, so it will iterate N times, where N is the number of elements in the list.
2. In each iteration, we perform a constant number of operations, including incrementing and decrementing **n**, and advancing or moving back **h**. These operations are all constant time.

Therefore, the time complexity of the **fast\_count** method is O(N), where N is the number of elements in the doubly-linked list. Despite the use of incremental and decremental steps, the overall complexity remains linear because each element in the list is visited once.

Top of Form

Q3. class MinHeap:

def \_\_init\_\_(self):

self.heap = []

# ... (other methods)

def insert(self, data):

# Step 1: Append data to the end of the heap (as a leaf node).

self.heap.append(data)

index = len(self.heap) - 1

# Step 2: Perform up-heapify to maintain the heap property.

while index > 0:

parent\_index = self.parent(index)

if self.heap[index] < self.heap[parent\_index]:

self.swap(index, parent\_index)

index = parent\_index

else:

break

def heapify(self, i):

left\_index = self.left\_child(i)

right\_index = self.right\_child(i)

smallest = i

# Find the index of the smallest element among i, left, and right.

if left\_index < len(self.heap) and self.heap[left\_index] < self.heap[smallest]:

smallest = left\_index

if right\_index < len(self.heap) and self.heap[right\_index] < self.heap[smallest]:

smallest = right\_index

# If the smallest element is not at index i, swap and recursively heapify.

if smallest != i:

self.swap(i, smallest)

self.heapify(smallest)

# Example usage:

min\_heap = MinHeap()

min\_heap.insert(12)

min\_heap.insert(10)

min\_heap.insert(15)

min\_heap.insert(20)

print(min\_heap) # Should print: "10 12 15 20"

Q8. To implement the specified operations on the AVL tree storing patient records, follow these steps for each operation:

1. **Insert a Patient Record**:
   * Start at the root of the AVL tree.
   * Traverse the tree to find the appropriate location for the new patient record based on their unique ID.
   * Insert the new patient record as a leaf node while maintaining the AVL property.
   * Update the height and balance factor of nodes along the path from the inserted node to the root.
   * Perform necessary rotations (single or double) to rebalance the tree.
   * Repeat steps 3-4 as you move up the tree.
   * Ensure that the AVL property is maintained at each node.
   * The final result is an AVL tree with the new patient record inserted.
2. **Delete a Patient Record**:
   * Start at the root of the AVL tree.
   * Traverse the tree to find the node containing the patient record to be deleted based on their unique ID.
   * If the node is a leaf node or has only one child, remove the node and replace it with its child (if any).
   * If the node has two children, find the in-order successor (or predecessor) node.
   * Copy the data from the successor node to the node to be deleted and remove the successor node (which has at most one child).
   * Update the height and balance factor of nodes along the path from the deleted node to the root.
   * Perform necessary rotations (single or double) to rebalance the tree.
   * Repeat steps 4-5 as you move up the tree.
   * Ensure that the AVL property is maintained at each node.
   * The final result is an AVL tree with the patient record deleted.
3. **Update Severity Level**:
   * Start at the root of the AVL tree.
   * Traverse the tree to find the node containing the patient record to be updated based on their unique ID.
   * Update the severity level of the patient record.
   * The AVL tree's structure remains unchanged.
   * Ensure that the AVL property is maintained.
4. **Search for Patients by Severity**:
   * Perform an in-order traversal of the AVL tree.
   * At each node, compare the severity level with the specified range.
   * If it falls within the range, add the patient record to the result list.
   * Continue traversing the tree until all nodes are visited.
   * The result list will contain all patient records within the specified severity range.
5. **Display Patients by Severity**:
   * Perform an in-order traversal of the AVL tree.
   * Collect patient records at each node and store them in a list.
   * Sort the list of patient records by severity level.
   * Display the sorted list of patient records.

These steps ensure that you can efficiently insert, delete, update, search, and display patient records in an AVL tree while maintaining the AVL property through automatic re-balancing after each insertion or deletion operation.

Top of Form