DSA END-SEM RUBRIC

Q1)

a)

- **Correctness** - Does the pseudocode accurately merge two sorted singly linked lists *in place*? Does it return the head of the merged list? If yes → 4 marks
- **Efficiency** - Does the algorithm achieve linear time complexity? If yes → 2 marks
- **Clarity** - Is the pseudocode clear and easy to understand? Are appropriate variable names used? Is the logic well-structured and explained? If yes → 2 marks
- **Sample Pseudocode** - A correct pseudocode would look somewhat like this:

```
MergeLists(head1, head2):
    // If either list is empty, return the other list
    if head1 = NULL:
        return head2
    if head2 = NULL:
        return head1

    // Determine the head of the merged list
    if head1.data < head2.data:
        merged_head = head1
        current1 = head1.next
        current2 = head2
    else:
        merged_head = head2
        current1 = head1
        current2 = head2.next

    // Initialize the tail of the merged list
    merged_tail = merged_head

    // Merge the lists while maintaining sorted order
    while current1 != NULL and current2 != NULL:
        if current1.data < current2.data:
            // Append current1 to the merged list
            merged_tail.next = current1
            // Update the tail and move to the next node in current1
            merged_tail = current1
            current1 = current1.next
        else:
            // Append current2 to the merged list
```

```
            merged_tail.next = current2
            // Update the tail and move to the next node in current2
            merged_tail = current2
            current2 = current2.next

    // Append any remaining nodes from either list
    if current1 = NULL:
        merged_tail.next = current2
    else:
        merged_tail.next = current1

    // Return the head of the merged list
    return merged_head
```

b)

- **Explanation of Quicksort Algorithm** - Does the explanation cover the key steps of the Quicksort algorithm including *partitioning* and *recursive* calls? Is it clear and easy to understand? If yes → 3 marks
- **Pseudocode** - Is the pseudocode provided accurate and complete? Does it implement the Quicksort algorithm correctly? If yes → 3 marks
- **Complexity Analysis** -
  Best-case → **O(n*log(n))** → 1 mark
  Average-case → **O(n*log(n))** → 1 mark
- **Sample Pseudocode** - Can be found here

c)

- **Insert() and update() in a Max heap**
  (For both sections) They were allowed to write Delete(root) or Delete(any element) also in place of update(). Please grade accordingly.
  - ➢ **Accuracy -** Correctly states that the average case time complexity of both Insert() and Update() operations in a Max heap is O(log n), where n is the number of elements in the heap. → 1 mark
  - ➢ **Explanation -** Offers sufficient explanation regarding why the average case time complexity of these operations is O(log n) in a Max heap. → 3 marks.

- **Search in a Binary Search Tree**
  - ➢ **Accuracy -** Correctly states that the average case time complexity of searching in a Binary Search Tree (BST) is O(log n), where n is the number of nodes in the tree. → 1 mark
  - ➢ **Explanation -** Offers sufficient explanation regarding why the average case time complexity of searching in a BST is O(log n). → 3 marks

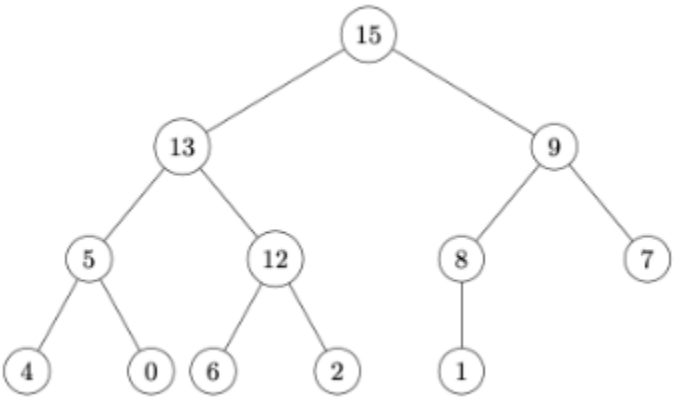
- **<u>Kruskal's algorithm</u>**
  - ➢ **Accuracy -** Correctly states that the average case time complexity of Kruskal's algorithm for finding the Minimum Spanning Tree (MST) is O(E log E), where E is the number of edges in the graph. → 1 mark
  - ➢ **Explanation -** Offers sufficient explanation regarding why the average case time complexity of Kruskal's algorithm is O(E log E). → 3 marks.

- **<u>DFS() in a graph</u>**
  - ➢ **Accuracy -** Correctly states that the average case time complexity of Depth-First Search (DFS) in a graph is O(V + E), where V is the number of vertices and E is the number of edges in the graph. → 1 mark
  - ➢ **Explanation -** Offers sufficient explanation regarding why the average case time complexity of DFS in a graph is O(V + E). → 3 marks.

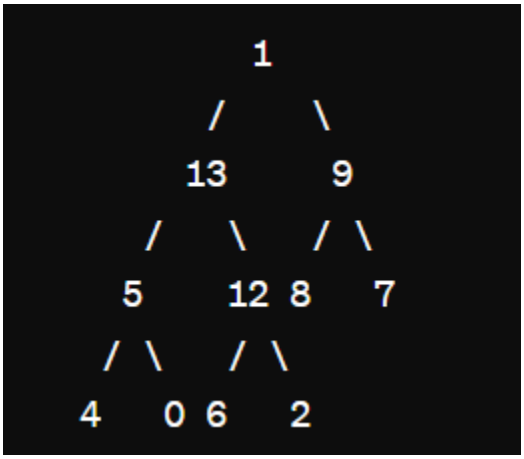
Q2)

a)

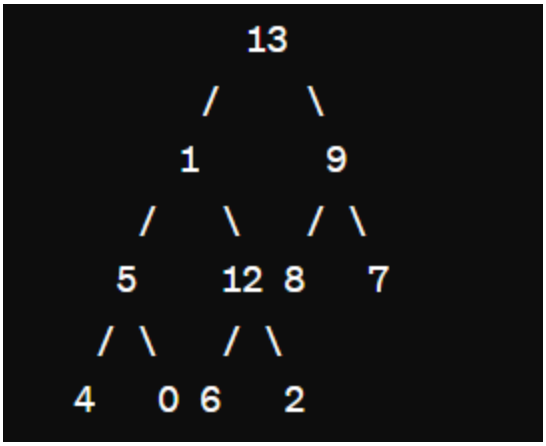
- ➢ **Initial heap**:

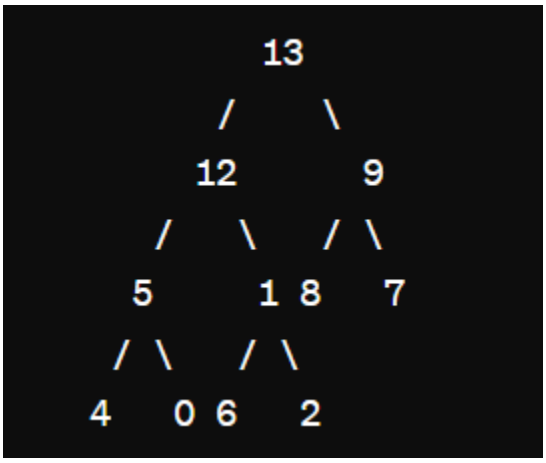


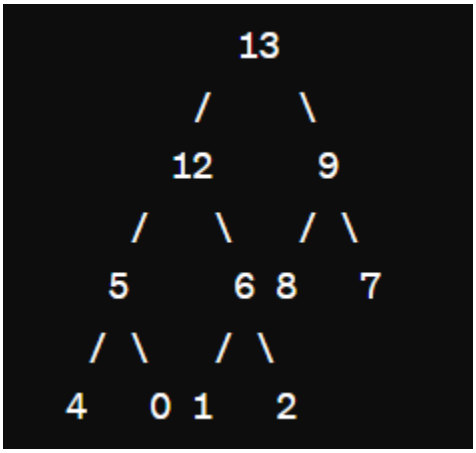
- ➢ **After extract max** (15 is swapped with 1 and then 15 is removed):



- ➢ **Final Heap After Heapify** (keep swapping 1 with its bigger child node):



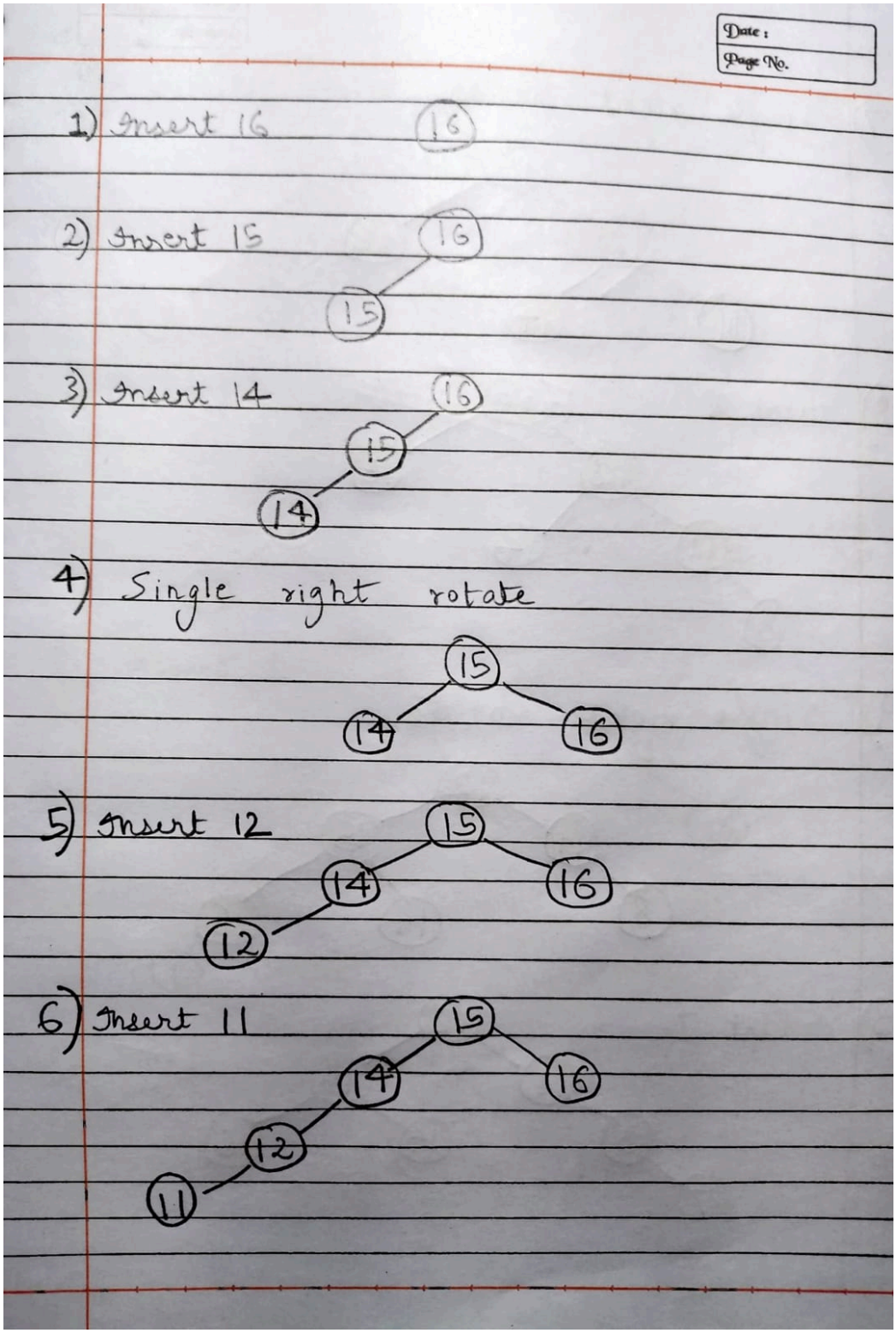




- ➢ If final heap is correct → **3 marks**
- ➢ If explanation is correct (steps are explained properly) → **7 marks**
- ➢ For any incorrect step, **1 mark** may be deducted from the total obtained through above mentioned points.

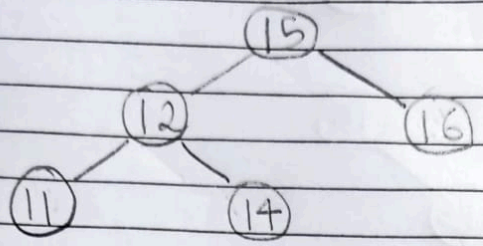
b) The steps are attached below. There are a total of 15 steps. **For each correct step, award 0.5 marks. Steps like insertion and rotation might be merged. Marks will be given accordingly**

If sufficient explanation (1 line is enough) is there for all steps, **award 2.5 marks.**

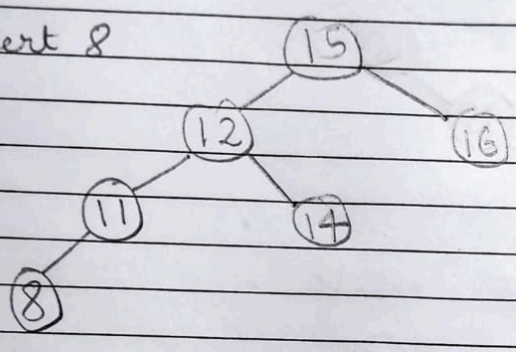


Date :
Page No.

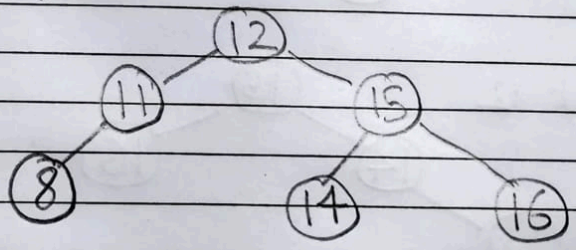
7) Single right rotate



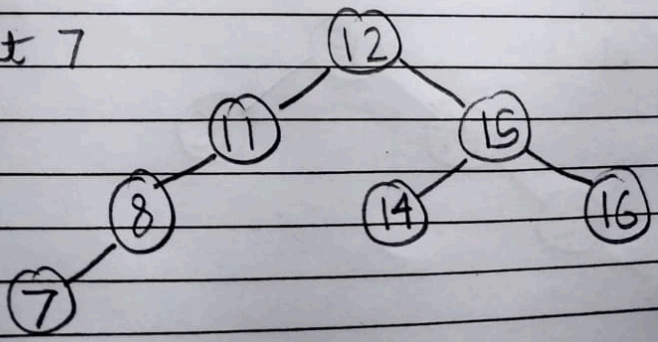
8) Insert 8



9) Single right rotate

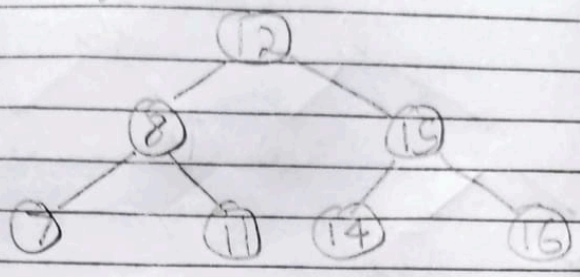


10) Insert 7

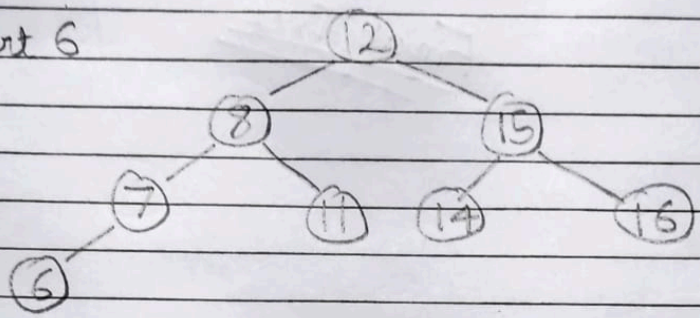


Date :
Page No.

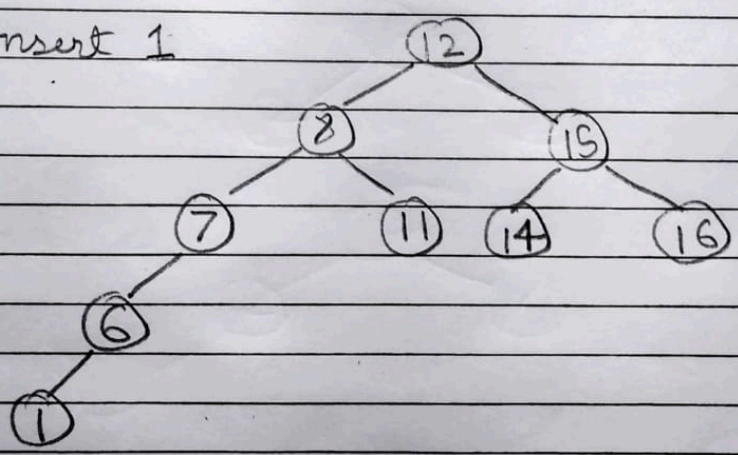
11) Single right rotate



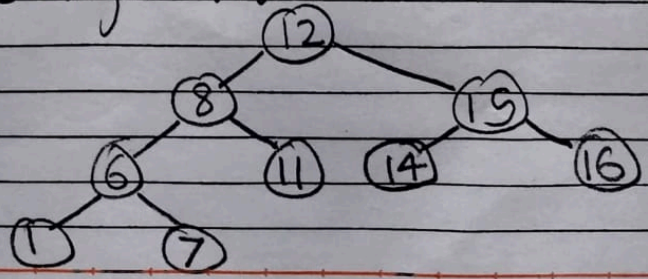
12) Insert 6

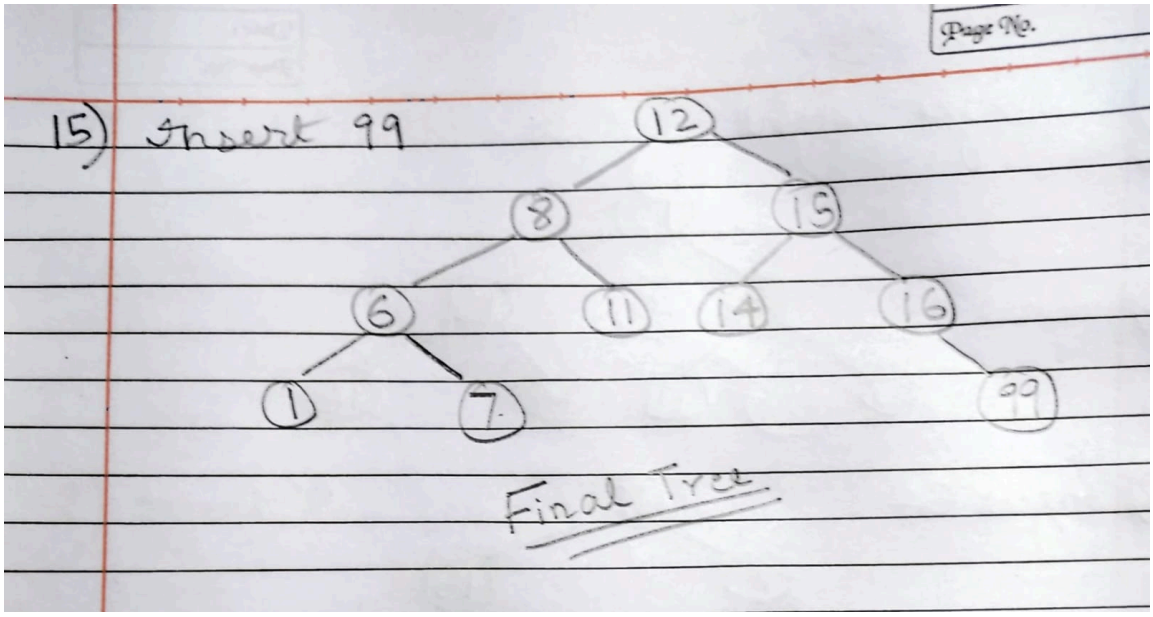


13) Insert 1



14) Single right rotate





Q3

(a)

- **Algorithm given correctly and completely : i.e. DIJKSTRA - 18 marks (if DIJKSTRA's algo is given which gives the highest probability and not the path with the highest probability, 14 marks would be given)**
  - **+ The modification that the weights will be calculated by Multiplying and not summing the weights - 9 marks**
- **Proper explanation is given - 9 marks.**

Pseudo code may or may not be given. But the following steps of DIJKSTRA's algo should be mentioned:-

1. **Initialize the distance from the starting node to all other nodes as infinity, except for the starting node, which should be set to 0.**
2. **Select the unvisited node with the smallest tentative distance from the source node.**
3. **Mark the selected node as visited.**
4. **Update the tentative distance of all unvisited neighbors of the visited node.**
5. **Repeat steps 2-4 until the destination node is reached or all nodes have been visited.**

We now view the weight of a path as the reliability of a path, and it is computed by taking the product of the reliabilities of the edges on the path. Our algorithm will be similar to that of DIJKSTRA, and have the same runtime, but we now wish to maximize weight, and RELAX will be done inline by checking products instead of sums, and switching the inequality since we want to maximize reliability. Finally, we track that path from y back to x and print the vertices as we go.

Algorithm 2 RELIABILITY(G, r, x, y)

```
1: INITIALIZE-SINGLE-SOURCE(G, x)
2: S = ∅
3: Q = G.V
4: while Q ≠ ∅ do
5:     u = EXTRACT-MIN(Q)
6:     S = S ∪ {u}
7:     for each vertex v ∈ G.Adj[u] do
8:         if v.d < u.d · r(u, v) then
9:             v.d = u.d · r(u, v)
10:            v.π = u
11:        end if
12:    end for
13: end while
14: while y ≠ x do
15:    Print y
16:    y = y.π
17: end while
18: Print x
```

(b) **There should be only one possible answer for the question i.e. given below.**

- **1.5 marks for each correctly written Discovery and Finished time.**
- **6 marks for a little explanation.**
- **Deduct only 4 marks if the time starts with 0 in place of 1 , therefore reducing all values by 1.**

If a student has used an algo where he pushes all the adjacent vertices of a node in a stack and then extracts them, then the answers might be different, as stack is LIFO. That solution is also acceptable, without any penalty. In that case, the answers might vary a little.

The following table gives the discovery time and finish time for each vetex in the graph.

Vertex	Discovered	Finished
q	1	16
r	17	20
s	2	7
t	8	15
u	18	19
v	3	6
w	4	5
x	9	12
y	13	14
z	10	11

Q4

(a)

For each of the two representations i.e. adjacency matrix and adjacency list:-

- **The algorithm is given correctly (in words or psudocode) - 6 marks**
- **Time complexity is given and explained a little - 4 marks.**

For the adjacency matrix representation, to compute the graph transpose, we just take the matrix transpose. This means looking along every entry above the diagonal, and swapping it with the entry that occurs below the diagonal. This takes time $O(|V|^2)$.

For the adjacency list representation, we will maintain an initially empty adjacency list representation of the transpose. Then, we scan through every list in the original graph. If we are in the list corresponding to vertex v and see u as an entry in the list, then we add an entry of v to the list in the transpose graph corresponding to vertex u. Since this only requires a scan through all of the lists, it only takes time $O(|E| + |V|)$

(b)

Here are the steps to find all possible words on a boggle board from a given dictionary.

It might be given as Psudo code or in words.

1. **[1 Marks]**Create a function `find_words(board, dictionary)` that takes in a 2D list `board` representing the boggle board and a set `dictionary` containing valid words.
2. **[1 Marks]**Create a helper function `check_adjacent(word, pos, board)` that takes in a word, a position `(i, j)`, and the board. This function will check if the word can be formed by traversing adjacent characters on the board in any of the eight directions.
3. **[2 Marks]**Within `check_adjacent`, initialize an empty set `visited` to keep track of visited cells.
4. **[12 Marks]**Within `check_adjacent`, recursively check if the current character of the word matches the character at the current position on the board. If it does, mark the current position as visited, and recursively check the adjacent positions in all eight directions. If the adjacent position is within the bounds of the board and has not been visited before, continue the recursion. If the adjacent position is out of bounds or has been visited before, backtrack and return False. If all characters of the word have been checked and found to match the corresponding characters on the board, return True. Unmark current cell from visited before returning false.
5. **[1 Marks]**Within `find_words`, initialize an empty list `results` to store the valid words found on the board.
6. **[3 Marks]**Within `find_words`, iterate through each position `(i, j)` on the board. For each position, iterate through each word in the dictionary. For each word, call `check_adjacent(word, (i, j), board)`. If `check_adjacent` returns True, add the word to the `results` list.

Psudocode:-

```
function find_words(board, dictionary):
  results = []
  for i in range(len(board)):
    for j in range(len(board[0])):
      for word in dictionary:
        if check_adjacent(word, i, j, board):
          results.append(word)
  return results

function check_adjacent(word, i, j, board):
  if word is empty:
    return true
```

```
  if i or j is out of bounds or current cell has been visited
    return false
  if current character of board does not match first character of word
    return false
  mark current cell as visited
  for each adjacent cell in 8 directions:
    if check_adjacent(word.substr(1), adjacent cell's row, adjacent cell's col, board) is
true:
      return true
  unmark current cell as visited
  return false
```