SENECA COLLEGE OF APPLIED ARTS AND TECHNOLOGY

FACULTY OF CONTINUING EDUCATION

FINAL EXAMINATION

Winter 2025

SUBJECT NAME:	**Data Structures and Algorithms**
SUBJECT CODE:	**DSA456V1A**
EXAMINATION DATE:	**April 14th , 2025**
INSTRUCTOR NAME:	**AZER KARADAG**
MARKS ALLOTTED:	**100**
WEIGHTING:	**30%**

SPECIAL INSTRUCTIONS:

Exam Books:	Required:	Not Required: X
Exam Aids:	Permitted: X	Not Permitted:
Exam Question Paper:	Returned:	Not Returned:X

Exam approved by,

Sheri Ladoucier

_____

Sheri Ladoucier, Academic Program Manager

Academic Policy Section 9:
Engaging in any form of academic dishonesty to obtain any type of advantage or credit is an offence and will not be tolerated by the College. Such offences under this policy include, but are not limited to, cheating, plagiarism, falsification, impersonation, misrepresentation and procurement.

SENECA COLLEGE OF APPLIED ARTS AND TECHNOLOGY

FACULTY OF CONTINUING EDUCATION

FINAL EXAMINATION

Student: Luca Novello - 038515003

Exam: Data Structures and Algorithms
TOTAL MARKS: 100

PART A – Fundamental Concepts [2 marks each, 10 Marks Total]

1. Define a data structure. Give two examples and state why data structures are important in programming.

Answer:

- A data structure is a way of organizing and storing data so it can be accessed and modified efficiently.

Examples:

- Arrays
- Linked lists

- Data structures are essential for optimizing performance and managing complexity in algorithms.

---

2. Explain the difference between linear and non-linear data structures. Provide an example of each.

Answer:

- Linear structures store data sequentially, while non-linear structures have hierarchical or networked relationships.

Examples:

- **Linear**: Array - elements are stored in a single line and accessed by index.
- **Non-linear**: Binary Search Tree – elements are stored in nodes with left and right child pointers, representing a hierarchy.

---

3. What is recursion? Write a recursive function in pseudocode to compute the factorial of a number.

Answer:

- Recursion is when a function calls itself to solve subproblems until a base case is reached.

Pseudocode:

```
function factorial(n):
        if n == 0 or n == 1:
                return 1
        else:
return n * factorial(n - 1)
```

---

4. Describe stack and queue. How are they different in terms of operations?

Answer:

- A **stack** is a linear data structure that follows the Last-In, First-Out (LIFO) principle. This means the last element added to the stack is the first one to be removed. It uses operations like *push* and *pop*.
- A **queue** is a linear data structure that follows the First-In, First-Out (FIFO) principle. The first element added to the queue is the first one to be removed. It uses operations like *enqueue* and *dequeue*.

---

5. Explain what a priority queue is. How is it different from a regular queue?

Answer:

- A priority queue removes elements based on priority instead of their insertion order. A regular queue removes elements in FIFO order, regardless of value or priority. Priority queues are often implemented using heaps.

PART B – Applied Data Structures [5 marks each, 50 Marks Total]

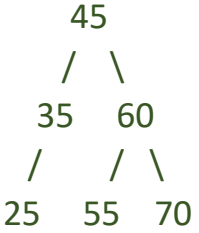
6. Binary Search Tree Deletion
   You are given a BST with nodes: [45, 30, 60, 25, 35, 55, 70].
   Delete the node with value 30. Show the updated tree.

Answer:

```
function deleteNode(root, key):
        if root is null:
                return null
        if key < root.value:
                root.left = deleteNode(root.left, key)
        else if key > root.value:
                root.right = deleteNode(root.right, key)
        else:
                if root.left is null:
                        return root.right
                if root.right is null:
                        return root.left
                minNode = findMin(root.right)
                root.value = minNode.value
                root.right = deleteNode(root.right, minNode.value)
        return root

function findMin(node):
        while node.left is not null:
                node = node.left
                return node
```

Updated Tree:



7. AVL Tree Balancing
   Insert the following into an AVL tree: [15, 20, 10, 25, 8].
   Show the AVL tree after each insertion, and explain any rotations needed.

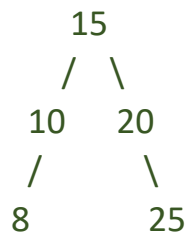
Answer:

- No rotations are needed because balance factors stay within -1 to 1.

Steps:

- Insert 15 -> root
- Insert 20 -> right of 15
- Insert 10 -> left of 15
- Insert 25 -> right of 20
- Insert 8 -> left of 10

Final Tree:



- All insertions keep the tree balanced. No rotation is required.

---

8. Heap Sort
   Given the array: [4, 10, 3, 5, 1],
   Use heap sort to sort the array in ascending order. Show the intermediate steps.

Answer:

- Build max-heap**:** [10, 5, 3, 4, 1]
- Swap 10 and 1 -> [1, 5, 3, 4, 10]
- Heapify -> [5, 4, 3, 1, 10]
- Swap 5 and 1 -> [1, 4, 3, 5, 10]
- Heapify -> [4, 1, 3, 5, 10]
- Swap 4 and 3 -> [3, 1, 4, 5, 10]
- Swap 3 and 1 -> [1, 3, 4, 5, 10]

Final Sorted Array:

- [1, 3, 4, 5, 10]

---

9. Hash Table with Open Addressing
   Implement a hash table using linear probing with a table size of 7. Insert keys: [10, 20, 15, 7, 5].
   Show the final table.

Answer:

- Use linear probing with a hash function h(k) = k % 7
- 10 -> 10 % 7 = 3 -> insert at index 3
- 20 -> 20 % 7 = 6 -> insert at index 6
- 15 -> 15 % 7 = 1 -> insert at index 1
- 7 -> 7 % 7 = 0 -> insert at index 0
- 5 -> 5 % 7 = 5 -> insert at index 5

Final Hash Table:

Index	0	1	2	3	4	5	6
Value	7	15	-	10	-	5	20

---

10. Circular Linked List Insertion
    Given a circular linked list: [12 -> 23 -> 34 -> 45 -> 12],
    Write pseudocode to insert a new node with value 28 after the node with value 23.

Answer:

```
function insertAfter(head, target, value):
    current = head
    do:
        if current.value == target:
            newNode = Node(value)
            newNode.next = current.next
            current.next = newNode
            break
        current = current.next
    while current != head
```

---

11. Graph Representation (Adjacency List)
    Create an adjacency list for the graph with edges:
    (1, 2), (1, 3), (2, 4), (3, 4), (4, 5).

- 1 -> 2, 3
- 2 -> 1, 4
- 3 -> 1, 4
- 4 -> 2, 3, 5
- 5 -> 4

---

12. Depth-First Search (DFS)
    Perform DFS on the graph above starting from node 1. Show the order of traversal.

Answer:

```
function DFS(graph, node):
    mark node as visited
    for neighbor in graph[node]:
        if neighbor not visited:
        DFS(graph, neighbor)
```

Traversal Order:

- 1 -> 2 -> 4 -> 3 -> 5

---

13. Implementing a Stack with an Array
    Write pseudocode to implement push and pop operations for a stack using a fixed-size array.

Answer:

```
function push(x):
    if top == MAX - 1:
        print "Stack Overflow"
    else:
        top = top + 1
        stack[top] = x

function pop():
    if top == -1:
        print "Stack Underflow"
    else:
```

```
value = stack[top]
top = top - 1
```

---

14. Advanced Trie Operations – Autocomplete Feature

Given a trie storing the words: [“tree”, “trie”, “trip”, “track”, “trap”, “trick”],

a) Show the trie structure.

b) Write pseudocode for an autocomplete function that, given the prefix "tri", returns all matching words from the trie.

c) What is the time complexity of this operation in terms of number of nodes?

Hint:

- Use DFS or BFS to explore all valid continuations of the prefix.
- Keep track of word-end nodes as you build the result list.

Answer:

```
function autocomplete(root, prefix):
   node = root
   for char in prefix:
      if char in node.children:
         node = node.children[char]
      else:
         return []
   result = []
   DFS(node, prefix, result)
   return result

function DFS(node, word, result):
   if node.isEnd:
      add word to result
   for char in node.children:
      DFS(node.children[char], word + char, result)
```

---

15. Advanced Cycle Detection with Union-Find

Given the undirected graph with edges:

(1, 2), (2, 3), (3, 4), (4, 1), (3, 5)

a) Use the Union-Find algorithm with path compression to detect whether the graph contains a cycle.

Answer:

- Process (1, 2): no cycle
- Process (2, 3): no cycle
- Process (3, 4): no cycle
- Process (4, 1): cycle detected
- Process (3, 5): no cycle
- Conclusion: A cycle is detected at edge (4, 1).

b) Show the parent array and rank array after each union.

Answer:

Initial arrays:

- parent = [0, 1, 2, 3, 4, 5]
- rank = [0, 0, 0, 0, 0, 0]

After union(1, 2):

- parent = [0, 1, 1, 3, 4, 5]
- rank = [0, 1, 0, 0, 0, 0]

After union(2, 3):

- parent = [0, 1, 1, 1, 4, 5]
- rank = [0, 1, 0, 0, 0, 0]

After union(3, 4):

- parent = [0, 1, 1, 1, 1, 5]
- rank = [0, 1, 0, 0, 0, 0]

Attempt union(4, 1):

- Cycle detected (no changes)

After union(3, 5):

- parent = [0, 1, 1, 1, 1, 1]
- rank = [0, 1, 0, 0, 0, 0]

c) What is the amortized time complexity?

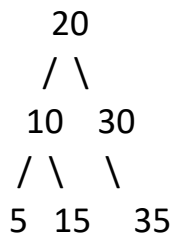
Answer:

- The amortized time complexity of Union-Find with path compression and union by rank is O(a(n)) - where a(n) is the inverse Ackermann function.

PART C – Algorithms and Problem Solving [5 marks each, 40 Marks Total]

16. Binary Search Tree – kth Smallest Element

Given a Binary Search Tree (BST):



a. Write an iterative algorithm to find the kth smallest element in the BST (assume k = 3).

Answer:

```
function kthSmallest(root, k):
    stack = empty stack
    current = root
    while stack is not empty or current is not null:
        while current is not null:
            push current to stack
            current = current.left
        current = stack.pop()
        k = k - 1
        if k == 0:
            return current.value
        current = current.right
```

b. Explain how this algorithm leverages in-order traversal.

Answer:

- By counting nodes visited in in-order, we can stop once we reach the kth smallest node.
- In-order traversal of this tree: 5 -> 10 -> 15 -> 20 -> 30 -> 35
- The 3rd smallest element is 15.

c. Analyze its time and space complexity.

Answer:

- **Time:** O(h + k), where h is the height of the tree
- **Space:** O(h) for the stack (in worst case, skewed tree)

---

17. Dynamic Programming – Longest Palindromic Subsequence

Given a string s = "agbdba",

a) Write a dynamic programming algorithm to compute the length of the longest palindromic subsequence.

Answer:

```
function longestPalindromeSubseq(s):
    n = length of s
    dp = 2D array of size n x n
    for i from 0 to n-1:
        dp[i][i] = 1
    for length from 2 to n:
        for i from 0 to n - length:
            j = i + length - 1
            if s[i] == s[j]:
                dp[i][j] = 2 + dp[i+1][j-1]
            else:
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])
    return dp[0][n-1]
```

b) Show the DP table used in your solution.

Answer:

i \ j	0 (a)	1 (g)	2 (b)	3 (d)	4 (b)	5 (a)
0 (a)	1	1	1	1	3	5
1 (g)		1	1	1	3	3
2 (b)			1	1	3	3
3 (d)				1	1	1
4 (b)					1	1
5 (a)						1

c) What is the time and space complexity?

Hint:

- Use a 2D array dp[i][j], where dp[i][j] stores the length of the longest palindromic subsequence in s[i...j].
- If s[i] == s[j], then add 2 and move inward; else, take the max between excluding s[i] or s[j].

Answer:

- **Time:** O(n^2)
- **Space:** O(n^2)

---

18. Binary Search – Recursive Implementation

Write a recursive implementation of binary search.

Use it to search for 25 in the array [5, 10, 15, 20, 25, 30].

Answer:

```
function binarySearch(arr, left, right, target):
    if left > right:
        return -1
    mid = (left + right) // 2
    if arr[mid] == target:
        return mid
    else if arr[mid] < target:
        return binarySearch(arr, mid + 1, right, target)
    else:
    return binarySearch(arr, left, mid - 1, target)
```

---

19. Fibonacci with Dynamic Programming

Write a bottom-up DP solution to compute the nth Fibonacci number.

Answer:

```
function fibonacci(n):
    if n == 0: return 0
    if n == 1: return 1
    create array fib[0..n]
    fib[0] = 0
    fib[1] = 1
    for i from 2 to n:
```

```
            fib[i] = fib[i-1] + fib[i-2]
    return fib[n]
```

---

20. Topological Sort

    Given a directed acyclic graph (DAG) with vertices and edges:
    Vertices: A, B, C, D
    Edges: (A, B), (B, C), (A, C), (C, D)
    Perform a topological sort.

Answer:

- Topological sort produces a linear ordering of vertices in a DAG
- Every directed edge (u -> v), vertex u comes before v in the ordering.
- We can use Kahn's algorithm or DFS to determine the order.

Calculate in-degrees

- A: 0
- B: 1 (from A)
- C: 2 (from A and B)
- D: 1 (from C)

Topological sort order

- Start with vertex with in-degree 0 -> A
- Remove A -> reduce in-degrees:
  - B -> 0
  - C -> 1
  - B -> remove -> C -> 0
  - C -> remove -> D -> 0
  - D -> remove

Final Order:

- A -> B -> C -> D

---

21. Prim's Algorithm – MST

    Given the weighted graph:

    (A, B, 1), (A, C, 4), (B, C, 2), (B, D, 5), (C, D, 1)

    Use Prim's algorithm to find the Minimum Spanning Tree starting at node A.

Answer:

Steps:

- Add A -> MST
- Choose edge (A, B, 1) -> add B
- Choose edge (B, C, 2) -> add C
- Choose edge (C, D, 1) -> add D

MST edges:

- (A, B, 1)
- (B, C, 2)
- (C, D, 1)

Total weight: 4

---

22. Knapsack – Recursive vs Dynamic Programming

    Explain the difference between recursive and DP solutions for the 0/1 knapsack problem.

    Which is more efficient and why?

Answer:

Recursive Approach:

- Tries all combinations (include/exclude each item)
- Time complexity: O(2^n)
- Recomputes subproblems, inefficient for large n

Dynamic Programming Approach:

- Uses table to store subproblem results
- Time complexity: O(nW), where W is capacity
- Avoids recomputation

Which is more efficient and why:

- Dynamic programming is more efficient due to memoization and is practical for larger input sizes.