List of practice problems for data structures practical course

1. Implement basic data structures such as arrays, stacks, and queues in a chosen programming language (e.g., Python, Java, or C++).
2. Write a program to rotate an array by k positions to the left or right. Implement the solution in an efficient way.
3. Implement a single linked list and a doubly linked list. Perform various operations like insertion (at the head, tail, or a specific position), deletion, searching, and reversing the list.
4. Implement a circular queue using an array. Perform enqueue, dequeue, and display operations, ensuring that the circular behavior is maintained.
5. Implement a stack using a linked list. Support basic stack operations like push, pop, peek, and check if the stack is empty.
6. Write a program to convert an infix expression to a postfix expression using a stack. Evaluate the resulting postfix expression.
7. Implement a function to reverse a singly linked list. The solution should reverse the links between nodes rather than just printing the list in reverse order.
8. Write a program to implement binary search on a sorted array. Both iterative and recursive solutions should be provided.
9. Implement a hash table using arrays. Handle collisions using techniques like separate chaining or open addressing (e.g., linear probing).
10. Given two sorted linked lists, write a program to merge them into a single sorted linked list.
11. Implement a priority queue using a binary heap. Perform operations like insertion, deletion, and finding the maximum/minimum element.
12. Implement a binary tree and perform the following traversals: preorder, inorder, postorder, and level-order. Ensure both recursive and iterative implementations.
13. Write a function to find the height (maximum depth) of a binary tree. Implement both recursive and iterative solutions.
14. Implement a function to check if a given binary tree is a binary search tree (BST).
15. Implement a graph using an adjacency list. Perform basic operations like adding/removing vertices and edges, and implement traversal algorithms like DFS and BFS.
16. Implement an algorithm to detect cycles in a directed and undirected graph using DFS and Union-Find methods.
17. Implement Dijkstra's algorithm to find the shortest path in a weighted graph. Extend the solution to implement Bellman-Ford for graphs with negative weights.
18. Implement Kruskal's and Prim's algorithms to find the minimum spanning tree (MST) of a graph.
19. Implement a Trie (prefix tree) to store a list of words. Include operations for inserting, searching, and deleting words, and finding all words with a given prefix.
20. Implement a solution to the N-Queens problem using backtracking. Find all possible solutions for placing N queens on an NxN chessboard.
21. Implement a double-ended queue (deque) using a dynamic array or a doubly linked list. Perform operations such as insertion and deletion from both ends.
22. Write a program to check whether a given string containing parentheses, braces, and brackets is balanced. Use a stack to validate the structure.

Graphical User Interface Problems

1. Create a GUI application that allows users to perform stack operations (push, pop, peek). Visually represent the stack's elements on the screen and update it dynamically as operations are performed.
2. Develop a GUI-based queue simulation where users can enqueue and dequeue elements. Use graphical elements to show the queue dynamically as operations are executed.
3. Build a GUI application that allows users to create a single or doubly linked list. Users should be able to insert nodes at the head, tail, or a specific position and delete nodes, with a graphical representation of the linked list updating in real-time.
4. Implement a GUI application that constructs a Binary Search Tree (BST). Allow users to insert, delete, and search for nodes. Visually show the tree structure and highlight the paths taken during the search, insert, or delete operations.
5. Develop a GUI tool to create and visualize graphs. Implement Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms. Allow users to add nodes and edges, and then watch the graph traversal visually.
6. Build a GUI-based circular queue simulation where users can enqueue, dequeue, and see the circular nature of the queue. Graphically represent how the queue wraps around when it reaches its limit.
7. Implement a GUI-based hash table. Allow users to add, search, and delete keys, and choose between collision-handling methods like separate chaining or open addressing. Visualize hash collisions and how they are resolved.
8. Implement a GUI-based shortest path finder using Dijkstra’s algorithm. Allow users to create a weighted graph, select a start node, and visualize the algorithm's step-by-step process of finding the shortest path.
9. Build a GUI application that creates a graph and visualizes the construction of a Minimum Spanning Tree (MST) using Kruskal’s or Prim’s algorithm. Show each step in the algorithm, including edge selection and set formation.
10. Develop a GUI to solve the graph coloring problem using backtracking. Users can input the number of vertices and colors, and the application should visualize the assignment of colors to graph nodes.
11. Implement a GUI application that visualizes topological sorting of a Directed Acyclic Graph (DAG). Users can create a graph and watch as the vertices are ordered topologically.
12. Implement a GUI-based Union-Find (Disjoint Set) data structure. Allow users to perform union and find operations on sets, and show how sets are merged and parent pointers updated.