Sorting, Binary Trees, Priority Queues, and Graphs

Sorting

1. Implement the **Selection Sort** algorithm and analyze its best-case and worst-case time complexity.
2. Implement the **Insertion Sort** algorithm and analyze its best-case and worst-case time complexity.
3. Implement the **Quick Sort** algorithm and analyze its best-case and worst-case time complexity.

Binary Trees

4. Implement the missing methods in the code available at:
   `https://github.com/rcpsilva/PCC104_DesignAndAnalysisOfAlgorithms/blob/main/2025-1/Problem%20sets/Latex%20Source/5_Sorting_BinaryTrees_Heaps_Graphs/binary_search_tree.py`
   For each implemented method, present its time complexity analysis. You may define auxiliary methods if needed.

Heaps

5. Using the base code available at:
   `https://github.com/rcpsilva/PCC104_DesignAndAnalysisOfAlgorithms/blob/main/2025-1/Problem%20sets/Latex%20Source/5_Sorting_BinaryTrees_Heaps_Graphs/heap.py`
   Implement a **min-heap** using a Python list as the underlying data structure. Ensure that the heap supports insertion and removal of the minimum element.

Graphs

6. Implement the **Depth-First Search (DFS)** and **Breadth-First Search (BFS)** algorithms for a graph represented using an adjacency list. Analyze the time and space complexity of both algorithms in terms of the graph's depth h and average branching factor b.
7. Implement the **DFS**, **BFS**, and **A**** algorithms to find a path in a maze. The maze is represented as an $m \times n$ matrix, where each cell may contain:
   - `0`: free space;
   - `1`: wall;
   - `s`: starting position;
   - `g`: goal (maze exit).

   Your implementation should return a valid path (if one exists) from the start to the goal position.