quiz 2 - notes

Data Structures & Algorithms Cheat Sheet

Week 1: Mar 3-7

Stack ADT

- **LIFO** (Last In First Out) structure
- **Operations**:
  - `push(item)`: Add to top - O(1)
  - `pop()`: Remove from top - O(1)
  - `peek()`: View top item - O(1)
  - `is_empty()`: Check if empty - O(1)
- **Implementation**: Python list (append/pop from end)
- **Applications**: Function call stack, undo operations, HTML tag matching

Algorithm Analysis (Big-Oh)

- **Time Complexity**: How runtime grows with input size
- **Common complexities**:
  - O(1): Constant
  - O(log n): Logarithmic

Week 3: Mar 24-28

Mutability

- Mutable objects can change after creation (lists, dicts)
- Immutable objects cannot (tuples, strings, numbers)
- Implications for function arguments (pass by object reference)

Trees

- Hierarchical structure with nodes and edges
- **Binary Tree**: Each node has ≤ 2 children
- **Binary Search Tree (BST)**: Left < Parent < Right
  - Operations: O(h) where h is height
  - Balanced BST: h = O(log n)

Priority Queue ADT

- Each element has priority
- Highest priority element served first
- Operations:

- $O(n)$: Linear
- $O(n \log n)$: Linearithmic
- $O(n^2)$: Quadratic
- $O(2^n)$: Exponential

- **Rules**:
  1. Drop constants ($5n \rightarrow O(n)$)
  2. Drop lower order terms ($n^2 + n \rightarrow O(n^2)$)
  3. Worst case usually considered

Recursion & Merge Sort

- **Recursion**: Function calls itself
  - Base case: Stopping condition
  - Recursive case: Calls itself with smaller input
- **Merge Sort**:
  - Divide and conquer algorithm - $O(n \log n)$
  - Steps:
    1. Divide array into halves
    2. Recursively sort each half
    3. Merge sorted halves

Amortized Analysis

- Average time over sequence of operations
- **Dynamic array example**:

  - `insert(item, priority)`
  - `get_highest_priority()`
  - `delete_highest_priority()`

Heaps

- Complete binary tree with heap property
- **Min-Heap**: Parent ≤ Children
- **Max-Heap**: Parent ≥ Children
- Operations:
  - `insert`: $O(\log n)$
  - `extract_min/max`: $O(\log n)$
  - `heapify`: $O(n)$ to build heap from array

Week 4: Mar 31-Apr 4

Recursion Revisited

- **Tail recursion**: Recursive call is last operation
- Can be optimized to iterative (constant space)
- **Memoization**: Cache results to avoid recomputation
- **Divide and conquer**: Break problem into smaller subproblems

Common Patterns

- **Two pointers**: Track positions in array/list

- When full, resize (double) and copy elements
- Most appends O(1), occasional O(n)
- Amortized O(1) per append

Week 2: Mar 10-14

Maze Searching (DFS & BFS)

- **DFS (Depth-First Search)**:
  - Uses stack (LIFO)
  - Explores as far as possible along each branch
  - Not optimal (may not find shortest path)
- **BFS (Breadth-First Search)**:
  - Uses queue (FIFO)
  - Explores all neighbors first
  - Finds shortest path in unweighted graph

Queue ADT

- **FIFO** (First In First Out) structure
- **Operations**:
  - `enqueue(item)`: Add to back - O(1) (with dynamic array)
  - `dequeue()`: Remove from front - O(n) with list (shifting needed), O(1) with linked list
  - `peek()`: View front item - O(1)

- **Sliding window**: Maintain subset of data
- **Greedy algorithms**: Make locally optimal choices
- **Backtracking**: Try options and undo if fail

Implementation Notes

- Python `list` as stack: use `append()` and `pop()`
- Python `collections.deque` for queue/deque operations
- For priority queues: `heapq` module or implement heap

    - `is_empty()` : Check if empty - O(1)

Linked Lists

- **Singly Linked List**:
    - Nodes with `data` and `next` pointer
    - Operations:
        - `append` : O(n) (must traverse)
        - `prepend` : O(1)
        - `pop` : O(n) from end, O(1) from front
- **Doubly Linked List**:
    - Nodes with `data` , `next` , and `prev` pointers
    - Operations:
        - `append` : O(1) (with tail pointer)
        - `prepend` : O(1)
        - `pop` : O(1) from both ends

Deque ADT

- Double-ended queue
- Supports O(1) operations at both ends
- Can be implemented with doubly linked list