Search 3

Meaning of Collision in Computer Science:

A collision in computer science refers to a situation where two different inputs result in the same output in a system that expects unique results.

Where Collisions Happen:

1. **Hashing:**
   - **Two different values produce the same hash.**
   - **Example:**
     hash("dog") → 4567
     hash("cat") → 4567 ← *collision!*
2. **Networking:**
   - **Two devices try to send data at the same time, causing a data collision.**
   - **Happens in older Ethernet networks.**
3. **Cryptography:**
   - **Two different messages create the same digital signature or hash. This is dangerous because it can break security.**

Solutions to Handle Collisions

1. Separate Chaining (Linked Lists)

- **How it works: Each bucket stores a linked list of entries. Colliding keys are appended to the list.**
- **Example:**

```
table[3] = ["Alice"] → ["Alice", "Bob"]  # After collision
```

- **Best for: Databases (e.g., Java `HashMap`).**
- **Drawback: Poor cache locality (pointer chasing).**

2. Open Addressing (Probing)

- How it works: If a bucket is occupied, probe the next available slot using:
  - Linear Probing: `(hash(key) + i) % size`
  - Quadratic Probing: `(hash(key) + i²) % size`
  - Double Hashing: `(hash1(key) + i * hash2(key)) % size`
- Example:

```
table[3] = "Alice" → table[4] = "Bob"  # Linear probing
```

- Best for: Memory-constrained systems (e.g., Python `dict`).
- Drawback: Clustering (long probe sequences).

3.Robin Hood Hashing (Variant of Open Addressing)

- How it works: Prioritize entries with shorter probe distances. If a new key has a longer probe distance, it "steals" the slot from an existing key.
- Example:
  - `"Alice"` (probe=0) stays at index 3.
  - `"Bob"` (probe=1) moves to index 4.
- Best for: Real-time systems (predictable latency).
- Drawback: Complex insertion logic.

4.Cuckoo Hashing

- How it works: Use two hash functions. If a collision occurs, evict the existing key and reinsert it into the second table.
- Example:

```
table1[3] = "Alice" → table2[5] = "Bob"  # If both collide
```

- Best for: High-performance lookups (e.g., Linux kernel).
- Drawback: Rehashing on cycles.

5. Hopscotch Hashing

- **How it works: Each bucket has a "neighborhood" (e.g., 32 slots). Colliding entries must land within this range.**
- **Example:**
  - **Bucket 3's neighborhood: slots 3–6.**
  - `"Bob"` **collides at 3 → placed at 5 (within neighborhood).**
- **Best for: Multicore systems (reduces lock contention).**
- **Drawback: Limited by neighborhood size.**

6. Dynamic Perfect Hashing

- **How it works: Two-level hashing:**

1. **First level: Buckets with collisions.**
2. **Second level: Mini hash tables for each colliding bucket.**

- **Example:**

```
table1[3] = SubTable["Alice", "Bob"]  # SubTable uses a different hash
```

- **Best for: Static datasets (e.g., compiler symbol tables).**
- **Drawback: High memory overhead.**

7. Coalesced Chaining

- **How it works: Hybrid of chaining and open addressing. Colliding entries chain within the main table.**
- **Example:**

```
table[3] = "Alice" → table[3].next = 7 → table[7] = "Bob"
```

- **Best for: Embedded systems (no external memory).**
- **Drawback: Degrades to linear probing under high load.**

8. Extendible Hashing

- **How it works: Uses a directory to point to buckets. Buckets split when full.**
- **Example:**
  - **Directory maps** `hash(key)` **to bucket pages.**

- **Buckets split when full.**
- **Best for: Disk-based storage (e.g., databases like MySQL).**
- **Drawback: Directory indirection overhead.**

Time Complexity Table Array

Operation	At First	At Index (i)	At End	Notes
Insert	O(n)	O(n)	O(1)*	Insertion at the beginning or middle requires shifting elements; at the end is O(1) if space is available.
Delete	O(n)	O(n)	O(1)	Deletion from beginning or middle requires shifting elements; deleting from the end is O(1).
Search (by value)	O(n)	O(n)	O(n)	Search requires scanning all elements.
Update (by index)	O(1)	O(1)	O(1)	Direct access to any index, so update is constant time.
Access (by index)	O(1)	O(1)	O(1)	Arrays provide constant time access by index.
Sort	O(n log n)	O(n log n)	O(n log n)	Sorting requires algorithms like Merge Sort or Quick Sort.

Explanation:

- **Insert at First or Middle**: **O(n)** because elements need to be shifted to make room for the new element.
- **Insert at End**: **O(1)** if there's space, but **O(n)** if resizing the array is necessary.
- **Delete at First or Middle**: **O(n)** because elements need to be shifted.
- **Delete at End**: **O(1)**, since no shifting is required.
- **Search, Update, and Access by Index**: These are **O(n)** or **O(1)** depending on the operation, as arrays allow random access.
- **Sort**: Sorting an array typically requires **O(n log n)** time, depending on the algorithm.

Time Complexity Table Singly Linked List

Operation	At First	At Index (i)	At End	Notes
Insert	O(1)	O(n)	O(n)	Inserting at the beginning is quick, while at index/end requires traversal.
Delete	O(1)	O(n)	O(n)	Deleting the first node is fast, but deleting at index/end requires traversal.
Search (by value)	O(n)	O(n)	O(n)	You need to search through the list for a value.
Update (by index)	O(n)	O(n)	O(n)	You need to find the index to update the value.
Access (by index)	O(n)	O(n)	O(n)	Requires traversing to the specified index.
Sort	O(n log n)	O(n log n)	O(n log n)	Sorting usually involves algorithms like Merge Sort or Quick Sort.

Explanation:

- **Insert at First**: **O(1)** because you only need to change the head pointer.
- **Insert at Index or End**: **O(n)** because you need to traverse the list to find the position.
- **Delete at First**: **O(1)** because you only need to update the head pointer.
- **Delete at Index or End**: **O(n)** because you need to find the previous node to adjust pointers.
- **Search, Update, and Access by Index**: These operations all require **O(n)** time because you need to traverse the list.

Time Complexity Table Doubly Linked List Operations

Operation	At First	At Index (i)	At End	Notes
Insert	O(1)	O(n)	O(1)	Insertion at the beginning and end is **O(1)** if tail pointer is available. Inserting at an index requires traversal.
Delete	O(1)	O(n)	O(1)	Deletion at the beginning and end is **O(1)**; deletion at an index requires finding the previous node.
Search (by value)	O(n)	O(n)	O(n)	No direct access, requires traversal from head or tail.
Update (by index)	O(n)	O(n)	O(n)	Need to traverse to the index (from head or tail).
Access (by index)	O(n)	O(n)	O(n)	Need to traverse from head or tail for access.
Sort	O(n log n)	O(n log n)	O(n log n)	Sorting typically involves algorithms like Merge Sort or Quick Sort.

Explanation:

- **Insert at First or End: O(1) since you can update the head or tail directly. If a tail pointer is maintained, insertion at the end is also O(1).**
- **Insert at Index: O(n) because you may need to traverse to the desired position.**
- **Delete at First or End: O(1) because you can update the head or tail pointers directly.**
- **Delete at Index: O(n) as you need to find the node before the deletion point.**
- **Search and Access by Index: These operations require O(n) time, as you may need to traverse either from the head or tail.**
- **Update: Requires O(n) time to traverse to the index and update the value.**
- **Sort: Sorting a doubly linked list typically takes O(n log n), depending on the sorting algorithm (e.g., Merge Sort).**

Time Complexity Comparison: Array vs Singly Linked List vs Doubly Linked List

Operation	Array	Singly Linked List	Doubly Linked List	Notes
Insert at First	O(n)	O(1)	O(1)	Array requires shifting; Linked lists just update head/tail.
Insert at End	O(1)*	O(n)	O(1)	Array is O(1) if there's space left; Linked list needs traversal.
Insert at Index (i)	O(n)	O(n)	O(n)	Both types of linked lists require traversal; Arrays need shifting.
Delete at First	O(n)	O(1)	O(1)	Array requires shifting; Linked lists just update head/tail.
Delete at End	O(1)	O(n)	O(1)	Array is O(1) if no shifting is needed; Linked list requires traversal.
Delete at Index (i)	O(n)	O(n)	O(n)	All need traversal to find the previous node.
Search (by value)	O(n)	O(n)	O(n)	No direct access in any of the structures, requires full scan.
Update (by index)	O(1)	O(n)	O(n)	Arrays provide direct access; Linked lists need traversal.
Access (by index)	O(1)	O(n)	O(n)	Arrays provide direct access, while linked lists require traversal.
Sort	O(n log n)	O(n log n)	O(n log n)	Sorting algorithms like Merge Sort or Quick Sort are commonly used.