Name / Nouran Sameh Mohamed

Search

1. What is Collision? (ما هو التصادم؟)

- *In Hashing: When two different inputs produce the same hash value.*
  (عندما يُنتج مدخلان مختلفان نفس قيمة الهاش.)
- *In Networking: When two devices send data at the same time, causing a clash.*
  (عندما يرسل جهازان بيانات في نفس الوقت، مما يُسبب تعارضًا.)

Why Do Collisions Happen?

Causes:

1. *Shared Medium (Hubs, old Ethernet).*
2. *No Coordination (Devices transmit whenever ready).*
3. *High Traffic (Too many devices → more collisions).*

Visual: Bus topology vs. Star topology comparison.

***Methods to Solve Collisions*:**

1. Separate Chaining (Open Hashing)

- *Uses linked lists to handle collisions.*
- *Each bucket (hash table entry) points to a list of elements with the same hash.*
- *Example: Java's HashMap (before Java 8).*

2. Open Addressing (Closed Hashing)

- *All elements are stored in the hash table itself.*
- *Probing methods used to find next available slot:*
  - *Linear Probing: Check next slot sequentially.*
  - *Quadratic Probing: Check slots using a quadratic function.*
  - *Double Hashing: Use a second hash function to determine step size.*

3. Robin Hood Hashing

- *A variation of open addressing.*
- ***Reduces variance** in probe lengths by "stealing" slots from richer keys (those with shorter probe sequences).*

4. Cuckoo Hashing

- *Uses **two hash tables** with different hash functions.*
- *If a collision occurs, the existing key is **kicked out** and reinserted into the second table.*

5. Dynamic Resizing (Rehashing)

- *When load factor (α = entries/slots) exceeds a threshold, **resize the table** and rehash all keys.*
- *Improves efficiency by reducing collisions.*

6. Hopscotch Hashing

- *Combines **open addressing** with **neighborhoods** (fixed-size regions).*
- *Reduces cache misses by keeping related keys close.*

7. Perfect Hashing (for Static Data)

- *Guarantees **zero collisions** by using a two-level hash scheme.*
- *Ideal for fixed datasets (e.g., compiler keyword tables).*

8. Coalesced Hashing

- *Mix of **separate chaining** and **open addressing**.*
- *Colliding elements are stored in the table but linked in a chain-like structure.*

===

Array

Array Operations Time Complexity

Operation	First (Beginning)	Last (End)	Any Index (Middle)	Why?
Insert	`O(n)`	`O(1)`	`O(n)`	- Insert at **first/middle**: Requires shifting all right elements. - Insert at **end**: No shifting needed.
Delete	`O(n)`	`O(1)`	`O(n)`	- Delete at **first/middle**: Requires shifting all right elements left. - Delete at **end**: No shifting needed.
Search	`O(n)` (if by value)	`O(n)`	`O(n)`	- Must scan the array linearly (unless indexed).
Update	`O(1)`	`O(1)`	`O(1)`	- Direct access via index (always constant time).

Key Explanations:

1. Insertion
   - Beginning/Middle: O(n) (requires shifting all subsequent elements).
   - End: O(1) (no shifting needed).
2. Deletion
   - Beginning/Middle: O(n) (shifts elements to fill the gap).
   - End: O(1) (no shifting needed).

3. Search
    - Always O(n) for value-based search (must check each element).
    - Index-based access: O(1) (not listed above since it's trivial).
4. Update
    - Always O(1) (arrays support random access by index).
5. Sorting
    - Average: O(n log n) for efficient algorithms.
    - Worst-case: O(n^2) for QuickSort (if poorly pivoted).

==

Single linked list

Single Linked List Operations Time Complexity

Operation	First (Head)	Last (Tail)	Any Index (Middle)	Why?
Insert	`O(1)`	`O(n)`*	`O(n)`	- **First**: Just update head. - **Last/Middle**: Must traverse entire list.
Delete	`O(1)`	`O(n)`*	`O(n)`	- **First**: Update head. - **Last/Middle**: Traverse to find previous node.
Search	`O(n)`	`O(n)`	`O(n)`	Must traverse node-by-node (no random access).
Update	`O(n)`	`O(n)`	`O(n)`	Must traverse to find the node first.

* With a **tail pointer**, insertion/deletion at last becomes `O(1)`.

Key Notes:

1. Insert/Delete at Head:
   - Always O(1) → Just update the head pointer.
2. Insert/Delete at Tail:
   - Without tail pointer: O(n) (traverse entire list).
   - With tail pointer: O(1) (direct access to tail).
3. Search/Update:
   - Always O(n) → No random access; must traverse from head.

==

Double linked list

Double Linked List Operations Time Complexity

Operation	First (Head)	Last (Tail)	Any Index (Middle)	Why?
Insert	`O(1)`	`O(1)`*	`O(n)`	- **First/Last**: Direct head/tail access - **Middle**: Must traverse
Delete	`O(1)`	`O(1)`*	`O(n)`	- **First/Last**: Direct access - **Middle**: Traverse + update neighbors
Search	`O(n)`	`O(n)`	`O(n)`	Must traverse node-by-node
Update	`O(n)`	`O(n)`	`O(n)`	Must find node first

* Assumes maintaining **both head and tail pointers**

Comparison Table: Double vs Single vs Array

Operation	Double Linked	Single Linked	Array
Insert (First)	O(1)	O(1)	O(n)
Insert (Last)	O(1)	O(n)	O(1)
Delete (Middle)	O(n)	O(n)	O(n)
Random Access	O(n)	O(n)	O(1)
Memory Overhead	Higher (2 pointers/node)	Lower (1 pointer/node)	None

Thank you