**DSA PROJECT**

**HashMap:-**

A HashMap is a data structure that stores key-value pairs, where each key is unique. The key is passed through a hash function, which computes an index where the corresponding value is stored in an array. This allows for fast data retrieval based on the key.

Average Time Complexity:

- Access/Insertion/Deletion: O(1) in average cases. This is because the hash function ideally distributes keys uniformly across the array, allowing for constant-time operations.

- Worst-case: O(n) if many keys collide and are placed in the same index (bucket), but good hash functions minimize this.

**Types of hashing:-**

1. Separate Chaining:

Separate chaining is another method to handle collisions in a hash table. Instead of storing multiple elements at the same array index, each index points to a linked list (or another data structure) of elements that hash to the same value. If a collision occurs, the new element is simply added to the list.

1. Open Addressing:

In open addressing, all elements are stored directly in the hash table array. When a collision occurs, the algorithm uses a probing technique to find an alternative empty slot within the array. This contrasts with separate chaining, where each bucket can hold multiple elements.

Probing:

Probing is a technique used in open addressing to resolve collisions in a hash table. When two keys hash to the same index, probing searches for the next available slot using a specific probing sequence:

1. Linear Probing : If a collision occurs, the algorithm checks the next slot in the array (index + 1) until an empty slot is found.
2. Quadratic Probing: Instead of checking the next slot, it checks at intervals that increase quadratically (index + 1, index + 4, index + 9, etc.).
3. Double Hashing: Uses a secondary hash function to determine the step size for probing.

**Advantages of HashMap:-**

1. Fast Access Time: HashMap offers constant-time complexity, O(1), for basic operations like insertion, deletion, and lookup on average. This makes it very efficient for scenarios where quick access to data is required.
2. Efficient Data Storage: HashMap efficiently handles large amounts of data by distributing entries across different buckets, minimizing the chances of collisions (when two keys map to the same bucket).
3. Flexibility in Key Types: HashMap allows keys to be any object type, offering flexibility in how keys are defined, as long as the key's hashCode() and equals() methods are properly implemented.

**Disadvantages of HashMap:**

1. No Ordering:HashMap does not maintain any order of the keys or values. If you need to maintain insertion order or sorted order, you would need to use LinkedHashMap or TreeMap instead.
2. Collision Handling Overhead: While hash functions are designed to minimize collisions, they can still occur. Collisions require additional handling (e.g., chaining or probing), which can slow down operations and degrade performance from O(1) to O(n) in the worst case.
3. Dependency on Good Hash Function: The efficiency of a HashMap is highly dependent on the quality of the hash function. A poorly designed hash function can lead to many collisions, reducing performance.

**Summary:**

- HashMap: A key-value data structure with average O(1) time complexity.

- Separate Chaining: Collisions are handled using linked lists (or similar structures).

- Open Addressing: Collisions are resolved by finding another slot within the array using probing.

- Probing: A collision resolution method in open addressing.

**TRIE : -**

A **Trie** (pronounced "try") is a type of tree data structure that is used to store a dynamic set of strings, usually to represent a dictionary of words. It is particularly efficient for tasks that involve searching, such as autocomplete, spell checking, and prefix matching.

**Key Characteristics of a Trie:**

1. Node Structure: Each node in a trie typically represents a single character of a string. The root node is often empty, and each subsequent level represents the next character in the string.
2. Edges and Children: Edges connect nodes, and each edge represents a character transition. Each node can have multiple children, each corresponding to a different character.
3. End of Word Marker: Nodes often contain a flag or marker to indicate whether the path from the root to that node represents a complete word in the trie.

**Example:**

Suppose you have the words "cat", "cap", "dog", and "dot". The trie structure would look like this:

(root)

/ |

c d

/ | \

a o o

/ \ | |

t p g t

In this trie:

* The path from the root through "c", "a", and "t" represents the word "cat".
* The path from the root through "d", "o", and "g" represents the word "dog".
* The word "cap" would be found by traversing "c", "a", "p".

**Time Complexity:**

* **Insertion/Search**: O(m), where m is the length of the string. Since each character requires only one step down the tree, the operation is proportional to the length of the string, not the total number of strings in the trie.
* **Space Complexity**: Tries can use a lot of memory because each node must store references to its children, and the total number of nodes can be large, especially if there are many different strings with little overlap.

**Applications:**

* **Autocomplete**: Quickly finding all words that start with a given prefix.
* **Spell Checkers**: Efficiently checking whether a word exists in a dictionary.
* **IP Routing**: Implementing longest prefix matching in IP routing tables.
* **Word Games**: Finding all possible words in a letter grid.

**Advantages:**

* Efficient for prefix-based searches.
* Can be more space-efficient than a hash table when dealing with a large number of small strings with common prefixes.

**Disadvantages:**

* High memory usage, especially with a sparse dataset.
* More complex implementation compared to simpler data structures like hash tables or arrays.

**LRU Cache :-**

An **LRU Cache** (Least Recently Used Cache) is a type of data structure that is used to manage a limited amount of memory by storing only the most recently used items. When the cache reaches its capacity and a new item needs to be added, the least recently used item (i.e., the item that hasn't been accessed for the longest time) is removed to make space.

**Key Concepts:**

1. **Capacity**: The maximum number of items the cache can hold.
2. **Access**: When an item is accessed (read or written), it is considered "recently used."
3. **Eviction Policy**: If the cache is full and a new item needs to be added, the least recently used item is evicted to make room.

**Operations:**

1. **Get(Key)**: Retrieves the value associated with a key if it exists in the cache. If the key is found, the item is marked as recently used.
2. **Put(Key, Value)**: Adds a new key-value pair to the cache. If the cache is full, it evicts the least recently used item first.

**Data Structure:**

An LRU Cache is typically implemented using a combination of:

* **HashMap (or Dictionary)**: To store key-value pairs for fast access.
* **Doubly Linked List**: To keep track of the order of access. The most recently accessed items are moved to the front of the list, and the least recently used items are at the back.

**Time Complexity:**

* **Get/Put**: O(1) average time complexity. This is achieved using the HashMap for quick access and a doubly linked list for quick updates to the order of usage.

**Applications:**

* **Operating Systems**: Managing virtual memory pages.
* **Web Browsers**: Caching web pages or resources.
* **Databases**: Caching query results to speed up access.
* **Software Caches**: In applications where frequent access to a small subset of data is needed.

**Advantages:**

* Efficient management of limited memory by ensuring that frequently accessed data stays in the cache.
* Simple and effective in scenarios where recent usage is a good predictor of future access.

**Disadvantages:**

* Overhead of maintaining the order of usage, especially in very large caches.
* Not suitable if the access pattern is not predictable or if older data may still be frequently accessed.

**Linked List :-**

linked list is a linear data structure where elements, known as nodes, are connected using pointers. Each node contains two parts: the data and a reference (or pointer) to the next node in the sequence. Linked lists are dynamic, meaning they can easily grow and shrink in size by adding or removing nodes.

**1. Singly Linked List (SLL):**

In a singly linked list (SLL), each node contains two elements:

* Data: The value stored in the node.
* Next Pointer: A reference to the next node in the list.

In an SLL, nodes are linked in a single direction, from the first node (head) to the last node, where the last node’s next pointer points to `None` or `null`, indicating the end of the list. Traversal is only possible in the forward direction.

**2. Doubly Linked List (DLL):**

A doubly linked list (DLL) is an extension of the singly linked list, where each node contains an additional pointer:

* Data: The value stored in the node.
* Next Pointer: A reference to the next node in the list.
* Previous Pointer: A reference to the previous node in the list.

In a DLL, nodes are linked in both directions, allowing traversal from both the head (forward) and the tail (backward).

**Applications of Doubly Linked List:**

1. Navigation Systems: Used in browsers for forward and backward page navigation.
2. Text Editors: Enables undo and redo functionalities.
3. Media Players: Manages playlists, allowing smooth forward and backward navigation.
4. Deques (Double-ended Queues): Efficiently supports insertion and deletion from both ends.
5. Memory Management: Used in operating systems for managing free memory blocks.

**Advantages of Doubly Linked List:-**

1. Bidirectional Traversal: You can traverse the list in both directions, forward and backward.
2. Efficient Insertions and Deletions: Insertion and deletion operations are easier and more efficient, especially when dealing with nodes in the middle or end of the list.
3. Enhanced Flexibility: Some operations and algorithms are more easily implemented due to the ability to move both forward and backward through the list.

**Disadvantages of Doubly Linked List:**

1. Increased Memory Usage: Each node requires more memory due to the additional `prev` pointer.
2. Complex Implementation: Managing both `next` and `prev` pointers adds complexity to operations like insertion, deletion, and traversal.
3. Slower Operations: The overhead of maintaining two pointers per node can slow down operations, especially in memory-constrained environments.

**Regular Expression :-**

Regex (short for Regular Expression) is a sequence of characters that forms a search pattern. It is used for pattern matching within strings, allowing you to search, match, and manipulate text based on specific patterns or rules.

Example:

* Matching an Email Address: [a-zA-Z0-9.\_%+-][+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}](mailto:+@[a-zA-Z0-9.-]+\.%5ba-zA-Z%5d%7b2,%7d)
* Matching a URL: https?:\/\/(www\.)?[a-zA-Z0-9-]+\.[a-zA-Z]{2,6}(\/[a-zA-Z0-9#]+\/?)\*

**Applications:-**

* **Search and Replace**: Regex can be used to find patterns in text and replace them, such as formatting strings or sanitizing input.
* **Data Validation**: Validating user inputs like email addresses, phone numbers, or passwords.
* **Text Parsing**: Extracting information from strings, like pulling out specific parts of a log file.
* **Web Scraping**: Extracting data from HTML or other text-based content.

**Advantages:-**

* Powerful and flexible for text processing.
* Allows for concise expression of complex search patterns.

**Disadvantages:-**

* Can be difficult to read and write for complex patterns.
* Performance can be an issue with very complex patterns or very large input strings.

**Made By:-**

1. Allan Santosh Joseph (2301010014)
2. Nischay Bhatia (2301010077)
3. Satyam Mahato (2301010110)