Contents

[✅ 1. Array 4](#_Toc204553930)

[✅ 2. List (Dynamic Array / List<T> in C#) 4](#_Toc204553931)

[✅ 3. LinkedList (Doubly Linked List) 5](#_Toc204553932)

[✅ 4. Stack (LIFO) 5](#_Toc204553933)

[✅ 5. Queue (FIFO) 6](#_Toc204553934)

[✅ 6. Heap (Priority Queue) 6](#_Toc204553935)

[✅ 7. HashSet<T> 7](#_Toc204553936)

[✅ 8. Dictionary<TKey, TValue> 7](#_Toc204553937)

[✅ 9. Binary Tree / BST 7](#_Toc204553938)

[✅ 10. Trie (Prefix Tree) 8](#_Toc204553939)

[✅ 11. Graph (Adjacency List) 8](#_Toc204553940)

[📚 Summary Table 9](#_Toc204553941)

[. 9](#_Toc204553942)

[APPENDIX: UNDER THE HOOD (INCLDES USE-CASES) (Leetcode Problems) 10](#_Toc204553943)

[1. 🔢 Arrays 10](#_Toc204553944)

[Use Cases: 10](#_Toc204553945)

[Index-based access (e.g. sliding window, prefix sum) 10](#_Toc204553946)

[Simplest data container 10](#_Toc204553947)

[2. 📋 List (List<T> in C#) 10](#_Toc204553948)

[Use Cases: 11](#_Toc204553949)

[Variable-length containers 11](#_Toc204553950)

[Common in backtracking, permutations, etc. 11](#_Toc204553951)

[3. 🔗 Linked List 11](#_Toc204553952)

[Use Cases: 11](#_Toc204553953)

[LRU Cache 11](#_Toc204553954)

[Custom stack/queue 11](#_Toc204553955)

[4. 🗂️ Stack 11](#_Toc204553956)

[Use Cases: 12](#_Toc204553957)

[Valid Parentheses 12](#_Toc204553958)

[Infix/Postfix evaluation 12](#_Toc204553959)

[DFS traversal 12](#_Toc204553960)

[5. 📬 Queue 12](#_Toc204553961)

[Use Cases: 12](#_Toc204553962)

[BFS traversal 12](#_Toc204553963)

[Sliding window maximum 12](#_Toc204553964)

[6. ⏫ Priority Queue / Heap 12](#_Toc204553965)

[Implementation: 12](#_Toc204553966)

[Use Cases: 13](#_Toc204553967)

[Kth largest/smallest 13](#_Toc204553968)

[Dijkstra’s Algorithm 13](#_Toc204553969)

[Merging K Sorted Lists 13](#_Toc204553970)

[7. 🔁 HashSet 13](#_Toc204553971)

[Use Cases: 13](#_Toc204553972)

[Duplicate detection 13](#_Toc204553973)

[Fast lookups 13](#_Toc204553974)

[8. 🔑 Dictionary / HashMap 13](#_Toc204553975)

[Use Cases: 14](#_Toc204553976)

[Two Sum 14](#_Toc204553977)

[Frequency counting 14](#_Toc204553978)

[Caching 14](#_Toc204553979)

[9. 🌳 Binary Tree / BST 14](#_Toc204553980)

[Use Cases: 14](#_Toc204553981)

[Tree-based problems 14](#_Toc204553982)

[Recursive structure 14](#_Toc204553983)

[Balanced variants used in maps/sets 14](#_Toc204553984)

[10. 🧠 Trie (Prefix Tree) 14](#_Toc204553985)

[Use Cases: 14](#_Toc204553986)

[Word search 14](#_Toc204553987)

[Autocomplete 14](#_Toc204553988)

[Dictionary-style prefix storage 14](#_Toc204553989)

[11. 🌐 Graph 15](#_Toc204553990)

[Use Cases: 15](#_Toc204553991)

[Connectivity, cycles 15](#_Toc204553992)

[Shortest path problems 15](#_Toc204553993)

[Scheduling and dependencies 15](#_Toc204553994)

Yes please, proceed with pasting the entire comprehensive explanation (with all internals, methods, and plain English breakdowns) directly here in chat as one full message.

**ChatGPT said:**

Absolutely — here's the **full comprehensive explanation** of all major data structures used in LeetCode-style coding problems. Each one includes:

* Key internal components (variables, fields, structs/classes)
* Core methods
* Clear plain English explanation
* Real-world use in LeetCode problems

# ✅ 1. Array

**🔧 Internal Components:**

* **Fixed-size contiguous memory block**
* **Index-based access**
* Backed by raw memory; elements are stored next to each other

**📦 Core Fields:**

* baseAddress: memory location of the first element
* length: total number of elements
* elementSize: size in bytes per element

**🔁 Core Methods:**

* GetElementAt(index)
* SetElementAt(index, value)

**💡 How It Works (Plain English):**

Arrays store elements in one block of memory. Accessing an index is fast (O(1)) because the location of each item can be calculated directly from the index. The downside is that resizing an array is expensive (O(n)) and inflexible.

**💥 Why It’s Useful:**

* Used in sliding window, prefix sum, and dynamic programming problems.
* Examples: **Maximum Subarray**, **Product of Array Except Self**

# ✅ 2. List (Dynamic Array / List<T> in C#)

**🔧 Internal Components:**

* Backed by an **internal array**
* **Capacity** grows as needed
* Tracks number of actual elements separately

**📦 Core Fields:**

* \_items[]: internal array
* \_size: number of actual elements
* \_capacity: size of \_items array
* \_version: tracks modification count for safe iteration

**🔁 Core Methods:**

* Add(item)
* RemoveAt(index)
* Insert(index, item)
* Contains(item)
* Clear()
* Capacity, Count

**💡 How It Works:**

A List<T> starts with a small internal array. When you add more elements than it can hold, it resizes (usually doubling the capacity), copies the old elements, and adds the new one. This keeps average Add time at **O(1)** (amortized).

**💥 Why It’s Useful:**

* Resizable and versatile
* Used in backtracking, stack emulation, and BFS/DFS
* Examples: **Combination Sum**, **Generate Parentheses**

# ✅ 3. LinkedList (Doubly Linked List)

**🔧 Internal Components:**

* LinkedListNode<T>:
  + Value
  + Next
  + Previous
* LinkedList<T>:
  + Head
  + Tail
  + Count

**🔁 Core Methods:**

* AddFirst(value)
* AddLast(value)
* Remove(node)
* Find(value)
* Clear()

**💡 How It Works:**

Each node stores a reference to its next and previous nodes. Insertion and deletion at ends are constant time (O(1)), but accessing by index is **linear (O(n))**.

**💥 Why It’s Useful:**

* Great for LRU Cache, Deque, and undo/redo stacks
* Examples: **Design Linked List**, **LRU Cache**

# ✅ 4. Stack (LIFO)

**🔧 Internal Components:**

* Backed by an array or linked list
* int \_size: tracks number of elements
* \_items[]: internal array

**🔁 Core Methods:**

* Push(item)
* Pop()
* Peek()
* Contains(item)

**💡 How It Works:**

Stack follows Last-In, First-Out. Only the top element is accessible. All operations are **O(1)** using the internal array or linked list.

**💥 Why It’s Useful:**

* Used for parentheses matching, expression parsing, and backtracking
* Examples: **Valid Parentheses**, **Daily Temperatures**

# ✅ 5. Queue (FIFO)

**🔧 Internal Components:**

* Circular array
* int head, tail, count
* \_array[]: internal buffer

**🔁 Core Methods:**

* Enqueue(item)
* Dequeue()
* Peek()
* Contains(item)
* Clear()

**💡 How It Works:**

The queue stores items in a circular buffer to support efficient enqueue/dequeue from both ends. The head and tailpointers wrap around.

**💥 Why It’s Useful:**

* Used for BFS, level-order traversal
* Examples: **Number of Islands**, **Course Schedule**

# ✅ 6. Heap (Priority Queue)

**🔧 Internal Components:**

* Array-based binary heap
* Count
* Comparer<T> for custom ordering

**📦 Heap Index Rules:**

* Parent(i) = (i - 1) / 2
* LeftChild(i) = 2i + 1
* RightChild(i) = 2i + 2

**🔁 Core Methods:**

* Enqueue(item) (insert + heapify up)
* Dequeue() (remove root + heapify down)
* Peek()

**💡 How It Works:**

A heap maintains the **heap property**: parent is smaller (min-heap) or larger (max-heap) than its children. Insertions and deletions take **O(log n)**.

**💥 Why It’s Useful:**

* Solves Top-K, scheduling, and shortest path problems
* Examples: **Kth Largest Element**, **Merge K Sorted Lists**

# ✅ 7. HashSet<T>

**🔧 Internal Components:**

* \_buckets[]: maps hash to entry index
* \_entries[]: stores elements with hashCode, value, next
* \_freeList, \_freeCount: manages deleted entry reuse
* \_comparer: custom IEqualityComparer<T>

**🔁 Core Methods:**

* Add(item)
* Contains(item)
* Remove(item)
* Clear()
* UnionWith, IntersectWith, etc.

**💡 How It Works:**

Uses a hash code to find a **bucket**, then traverses a chain of entries for duplicates. Fast lookups & inserts average **O(1)**. Collision handled via chaining.

**💥 Why It’s Useful:**

* Membership tracking, duplicate removal, visited sets
* Examples: **Longest Consecutive Sequence**, **Two Sum (complement lookup)**

# ✅ 8. Dictionary<TKey, TValue>

**🔧 Internal Components:**

* \_buckets[]: hash to index mapping
* \_entries[]: stores key, value, hashCode, and next
* \_freeList, \_freeCount: tracks deleted slots
* \_comparer: custom key equality logic

**🔁 Core Methods:**

* Add(key, value)
* TryGetValue(key, out value)
* ContainsKey(key)
* Remove(key)
* this[key]: indexer for get/set
* Keys, Values, Count

**💡 How It Works:**

Same hashing as HashSet, but each entry holds a key-value pair. On lookup, hash the key, go to the right bucket, and follow the chain comparing keys.

**💥 Why It’s Useful:**

* Frequency maps, memoization, graph adjacency lists
* Examples: **Two Sum**, **Group Anagrams**, **LRU Cache**

# ✅ 9. Binary Tree / BST

**🔧 Internal Components:**

csharp

CopyEdit

class TreeNode {

  public int val;

  public TreeNode left;

  public TreeNode right;

}

* BST has ordering: left < root < right

**🔁 Core Methods:**

* Insert(val)
* Search(val)
* Delete(val)
* Traversals: InOrder, PreOrder, PostOrder, LevelOrder

**💡 How It Works:**

Each node links to two children. In BST, search, insert, and delete take **O(log n)** (if balanced). Traversals process nodes in desired order.

**💥 Why It’s Useful:**

* Sorted data, interval trees, binary search
* Examples: **Validate BST**, **Kth Smallest**, **Lowest Common Ancestor**

# ✅ 10. Trie (Prefix Tree)

**🔧 Internal Components:**

csharp

CopyEdit

class TrieNode {

  public Dictionary<char, TrieNode> Children;

  public bool IsEndOfWord;

}

* Root: empty starting node

**🔁 Core Methods:**

* Insert(word)
* Search(word)
* StartsWith(prefix)

**💡 How It Works:**

Each character in a word leads to a child node. A flag marks valid words. Traversal follows characters for lookups.

**💥 Why It’s Useful:**

* Efficient prefix checking, autocomplete, word search
* Examples: **Word Dictionary**, **Replace Words**, **Word Search II**

# ✅ 11. Graph (Adjacency List)

**🔧 Internal Components:**

* Dictionary<int, List<int>>: node to neighbor list
* Optional: edge weights via Dictionary<int, List<(int neighbor, int weight)>>
* Visited sets, parent maps, queues for BFS/DFS

**🔁 Core Algorithms:**

* AddEdge(u, v)
* BFS(start)
* DFS(start)
* Dijkstra(start)
* TopologicalSort()

**💡 How It Works:**

Each node maps to its neighbors. BFS uses queue for breadth exploration. DFS uses recursion or stack. Edge weights guide pathfinding.

**💥 Why It’s Useful:**

* Fundamental in scheduling, routing, connectivity
* Examples: **Course Schedule**, **Number of Islands**, **Shortest Path**

# 📚 Summary Table

|  |  |  |  |
| --- | --- | --- | --- |
| **Structure** | **Key Features** | **Time Complexities (avg)** | **Common Use Cases** |
| Array | Fixed size, fast index access | Access: O(1), Resize: O(n) | Sliding windows, prefix sums |
| List<T> | Dynamic size, indexed access | Insert: O(1)/O(n), Get: O(1) | Backtracking, combinations |
| LinkedList | Doubly-linked nodes | Add/Remove at ends: O(1) | LRU Cache, Deques |
| Stack | LIFO, single-end access | Push/Pop: O(1) | Expression parsing, DFS |
| Queue | FIFO, single-end insert/remove | Enq/Deq: O(1) | BFS, scheduling |
| Heap | Priority-based element selection | Enq/Deq: O(log n) | Top-K, Dijkstra |
| HashSet | Unique elements, fast lookup | Add/Check: O(1) | Duplicates, visited states |
| Dictionary | Key-value mapping | Add/Get: O(1) | Two Sum, caching |
| BST | Ordered data | Search: O(log n) | Kth smallest, range queries |
| Trie | Prefix tree for strings | Insert/Search: O(k) | Word search, autocomplete |
| Graph | Node-edge relationships | BFS/DFS: O(V + E) | Paths, cycles, topological sort |

# .

# APPENDIX: UNDER THE HOOD (INCLDES USE-CASES) (Leetcode Problems)

**📚 “Under the Hood” of Common Data Structures in LeetCode**

This guide covers how key data structures used in LeetCode problems work internally — beyond just syntax. For each structure, we include:

* **Internal implementation**
* **Key operations**
* **Time complexities**
* **Real-world use cases in LeetCode**

# 1. 🔢 Arrays

**Implementation**:

* Contiguous block of memory, each element accessible via an index.
* Fixed size after allocation.

**Key Operations**:

* Access: O(1)
* Insert/Delete at end: O(1) (if space)
* Insert/Delete in middle: O(n) (shift needed)

## Use Cases:

### Index-based access (e.g. sliding window, prefix sum)

### Simplest data container

**C# Example**:

csharp

CopyEdit

int[] arr = new int[5];

arr[0] = 10;

# 2. 📋 List (List<T> in C#)

**Implementation**:

* Dynamic array that resizes itself (usually doubles in size when full).

**Key Operations**:

* Access: O(1)
* Append: O(1) amortized
* Insert/Delete in middle: O(n)

**Under the Hood**:

* Maintains a backing array.
* Resizes by allocating a new array and copying old data.

## Use Cases:

### Variable-length containers

### Common in backtracking, permutations, etc.

# 3. 🔗 Linked List

**Implementation**:

* Nodes linked via references/pointers.
* Singly or Doubly linked.

**Key Operations**:

* Insert/Delete at head or tail: O(1)
* Search: O(n)
* No indexing

**C# Example**:

csharp

CopyEdit

LinkedList<int> list = new LinkedList<int>();

list.AddFirst(1);

## Use Cases:

### LRU Cache

### Custom stack/queue

# 4. 🗂️ Stack

**Implementation**:

* LIFO (Last-In, First-Out)
* Backed by array or linked list

**Key Operations**:

* Push: O(1)
* Pop: O(1)
* Peek: O(1)

## Use Cases:

### Valid Parentheses

### Infix/Postfix evaluation

### DFS traversal

**C# Example**:

csharp

CopyEdit

Stack<int> stack = new Stack<int>();

stack.Push(10);

# 5. 📬 Queue

**Implementation**:

* FIFO (First-In, First-Out)
* Backed by circular array or linked list

**Key Operations**:

* Enqueue: O(1)
* Dequeue: O(1)
* Peek: O(1)

## Use Cases:

### BFS traversal

### Sliding window maximum

**C# Example**:

csharp

CopyEdit

Queue<int> q = new Queue<int>();

q.Enqueue(5);

# 6. ⏫ Priority Queue / Heap

# Implementation:

* Binary Heap (Min or Max)
* Backed by array where parent/child are index-based

**Key Operations**:

* Insert: O(log n)
* Remove min/max: O(log n)
* Peek min/max: O(1)

## Use Cases:

### Kth largest/smallest

### Dijkstra’s Algorithm

### Merging K Sorted Lists

**C# Example**:  
Use PriorityQueue<TElement, TPriority> (introduced in .NET 6):

csharp

CopyEdit

var pq = new PriorityQueue<string, int>();

pq.Enqueue("task", 1);

# 7. 🔁 HashSet

**Implementation**:

* Hash table for fast uniqueness checks
* Keys only, no values

**Key Operations**:

* Add/Remove/Contains: O(1) average

**Under the Hood**:

* Uses hashing + bucket indexing
* Handles collisions internally

## Use Cases:

### Duplicate detection

### Fast lookups

**C# Example**:

csharp

CopyEdit

HashSet<int> set = new HashSet<int>();

set.Add(10);

# 8. 🔑 Dictionary / HashMap

✅ Already covered in detail — but recap:

**Implementation**:

* Hash table (hash code → index → bucket)

**Key Operations**:

* Add/Get/Remove: O(1) average

## Use Cases:

### Two Sum

### Frequency counting

### Caching

# 9. 🌳 Binary Tree / BST

**Implementation**:

* Each node has left and right children
* BST adds ordering: left < root < right

**Key Operations**:

* Insert/Search/Delete (BST): O(log n) average, O(n) worst (unbalanced)
* Traversals (inorder, preorder, postorder): O(n)

## Use Cases:

### Tree-based problems

### Recursive structure

### Balanced variants used in maps/sets

# 10. 🧠 Trie (Prefix Tree)

**Implementation**:

* Tree of characters
* Each node represents a prefix

**Key Operations**:

* Insert: O(k) (k = word length)
* Search: O(k)
* Memory-heavy

## Use Cases:

### Word search

### Autocomplete

## Dictionary-style prefix storage

**C# Example**:  
Typically custom-implemented using classes and child dictionaries.

# 11. 🌐 Graph

**Implementation**:

* Adjacency list or matrix

**Key Operations**:

* AddEdge, RemoveEdge
* BFS, DFS
* Topological sort, shortest path

## Use Cases:

### Connectivity, cycles

### Shortest path problems

### Scheduling and dependencies

**🧠 Final Thoughts**

|  |  |
| --- | --- |
| **Structure** | **Best For** |
| Array | Simple indexed access |
| List | Dynamic sizing |
| Stack | LIFO logic |
| Queue | FIFO logic |
| HashSet | Fast existence checks |
| Dictionary | Key-value lookups |
| Heap | Prioritized elements |
| Tree | Hierarchies, recursive structure |
| Graph | Network relationships |
| Trie | Prefix-based lookups |