Applied Data Structures in C#

October 16, 2025

Contents

1 Overview

1.1 What Data Structures Are and Why They Matter

In software engineering, a **data structure** is a disciplined arrangement of data in memory that enables efficient access, modification, traversal, and storage. While the algorithms we study define *how* a computation proceeds, data structures define *where* and *how* information resides during that computation. Choosing an appropriate structure often determines whether a program runs in milliseconds or minutes.

From a practical perspective, modern high-performance code is dominated not by raw asymptotics alone, but by implementation details: cache locality, branch prediction, allocation patterns, garbage collection (GC) pressure, and the layout of objects in memory. Thus, two algorithms with identical $O(n)$ complexity can differ by orders of magnitude in real-world execution time depending on the underlying data structure.

Conceptual layers. Data structures can be understood along three interacting layers:

- **Abstract data type (ADT):** the logical model and supported operations (e.g., a stack supporting push and pop).
- **Concrete representation:** how the ADT is implemented in memory (e.g., stack as an array vs a linked list).
- **Language/runtime realization:** how the chosen representation interacts with the programming language's memory model and runtime system (e.g., garbage collection, bounds checking, object headers).

Performance tradeoffs. For C# in particular, asymptotic analysis provides only the outer boundary of performance; actual runtime behavior depends heavily on:

- **Contiguity and cache friendliness:** arrays and `List<T>` benefit from spatial locality, while linked structures incur pointer chasing and cache misses.
- **Allocation and lifetime:** large numbers of short-lived heap objects stress the GC; value types (structs) avoid this but are copied by value.
- **Boxing and indirection:** generic collections of value types avoid boxing, but non-generic collections like `ArrayList` introduce hidden allocations and type conversions.
- **Concurrency safety:** concurrent collections use locking or lock-free techniques, trading latency for thread safety.

The pragmatic viewpoint. A competent developer or quantitative engineer does not merely memorize data structures, but understands the *mechanical implications* of each:

- An array is fast because it is contiguous and indexable.
- A linked list is flexible because it decouples logical order from physical memory.
- A dictionary is powerful because it transforms key lookups from $O(n)$ to $O(1)$ using hashing.

Thus, mastering data structures in a modern language like C# means mastering not only the theory of access patterns, but also the *runtime behavior* that arises from managed memory, type safety, and compiler optimizations.

1.2 Memory Models and Representation in C#

Unlike languages such as C or C++, where programmers manage memory directly, C# relies on a **managed memory model** orchestrated by the Common Language Runtime (CLR). This model abstracts away manual allocation and deallocation, but understanding its rules is crucial for designing efficient data structures.

Stack vs Heap Allocation

The CLR divides process memory into two main regions:

- **Stack:** a contiguous region used for short-lived, statically sized data. Local variables of *value types* (e.g., `int`, `double`, structs without heap references) are typically stored here. Allocation and deallocation are implicit (LIFO discipline).
- **Heap:** a large pool for dynamically allocated objects and arrays. Reference types (`class` instances, arrays, strings) are created on the managed heap via `new` and later reclaimed by the garbage collector.

Example:

Listing 1: Stack vs heap allocation in C#

```
struct Point { public int X, Y; }
class Node { public int Value; public Node? Next; }

void Example() {
       Point p = new Point { X = 3, Y = 4 }; // allocated on stack
       Node n = new Node(); // allocated on heap
}
```

Here, `p` is a value type; its fields live directly on the stack frame of `Example`. `n`, however, is a reference type; only a pointer (the reference) resides on the stack, while the object data lives in the heap.

Concept Check - Boxing and Unboxing in C#

Concept. In C#, *boxing* is the process of wrapping a **value type** inside a heap-allocated `object` so it can be treated as a **reference type**. The CLR allocates a new object on the managed heap and copies the value into it. *Unboxing* reverses this by extracting the value type from that heap object.

Listing 2: Example of boxing and unboxing

```
int n = 42;
object o = n; // BOXING: int copied into heap object
n = 100; // modifies only stack copy, not box
int m = (int)o; // UNBOXING: copy value back out
Console.WriteLine(m); // prints 42
```

Memory layout.

Region	Contents	Example
Stack	Local variable `n = 42`	fast, short-lived
Heap	Boxed object [header][value: 42]	GC-managed, slower

When boxing occurs.

- Assigning a value type to an `object` or interface variable.
- Passing a value type to a method expecting `object`.
- Storing value types in non-generic collections such as `ArrayList`.

Costs.

- Heap allocation for each boxed value (GC pressure).
- Value copy semantics — the box and the original are independent.
- Unboxing requires an explicit cast and performs another copy.

Practical implication. Generic collections such as `List<int>` avoid boxing entirely, storing raw values contiguously for maximum performance. Non-generic or polymorphic containers (e.g. `ArrayList` or `List<object>`) store boxed objects instead, increasing memory footprint and latency.

Reference vs Value Types

C# distinguishes two fundamental categories of types:

- **Value types** (`struct`, built-in numerics): stored *by value*. Assignment or parameter passing copies the entire value.
- **Reference types** (`class`, `interface`, `delegate`): stored *by reference*. Assignment copies a pointer to a heap-allocated object; multiple references can alias the same instance.

This distinction affects data structures profoundly:

- A `List<int>` stores raw `int` values contiguously (no boxing).
- A `List<object>` containing boxed integers stores references to heap objects, multiplying memory and GC cost.
- A `LinkedList<Node>` adds an extra level of indirection for each element.

Object Headers and Layout

Each heap object in .NET contains a small hidden header:

- A **sync block index** (used for locking and GC bookkeeping).
- A **type handle pointer** referencing the object's runtime type metadata.

For small objects, these headers represent a significant fraction of memory cost, making large arrays of primitives or structs vastly more space-efficient than arrays of small reference objects.

The Garbage Collector (GC)

The CLR's garbage collector automates memory reclamation by tracing reachable objects and freeing those no longer referenced. It divides the heap into *generations* (Gen 0, 1, 2) to optimize for the empirical fact that most objects die young. Short-lived allocations (e.g., temporary collections or iterators) are cheap, but long-lived linked structures may be promoted into older generations, where collection becomes more expensive.

For high-performance or real-time systems, reducing allocation churn and favoring value types or pooled object reuse can dramatically cut GC latency.

Arrays and Object Identity

Arrays are themselves reference types: even an array of value types is allocated on the heap. The array object includes:

- its header (type handle + length),
- a contiguous block of element storage.

Hence, a `T[]` gives random access in $O(1)$ time with excellent cache locality, but large arrays are subject to large-object-heap (LOH) management if exceeding roughly 85 KB, triggering different GC behavior.

Summary Table

Category	Example	Lifetime / Location
Value type	`int`, `Point`	Usually stack (or inline within object)
Reference type	`string`, `Node`, `List<T>`	Always heap
Array of value type	`int[]`	Heap (contiguous block of values)
Array of reference type	`object[]`	Heap (contiguous block of pointers)

Understanding these distinctions forms the foundation for analyzing how each C# data structure manages storage, indirection, and lifetime—knowledge essential for predicting real performance, not just algorithmic complexity.

1.3 Complexity Review

Before exploring individual data structures, it is helpful to recall the asymptotic costs of common operations. The table below summarizes the time complexity of the canonical operations — access, search, insertion, and deletion — for the major families of data structures encountered in everyday C# programming. We use the standard big-O notation, omitting constant factors and practical cache effects.

Structure	**Access**	**Search**	**Insertion**	**Deletion**
Array / `T[]`	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Dynamic Array / `List<T>`	$O(1)$	$O(n)$	$O(1)$ amortized	$O(n)$
Singly Linked List	$O(n)$	$O(n)$	$O(1)$ (at head)	$O(1)$ (at head)
Doubly Linked List / `LinkedList<T>`	$O(n)$	$O(n)$	$O(1)$ (known node)	$O(1)$ (known node)
Stack / Queue (array-based)	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Deque / Ring Buffer	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Hash Table / `Dictionary<K,V>`	–	$O(1)$ avg. / $O(n)$ worst	$O(1)$ avg. / $O(n)$ worst	$O(1)$ avg. / $O(n)$ worst
Binary Search Tree (unbalanced)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Balanced BST (`SortedDictionary<K,V>`)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Binary Heap / Priority Queue	$O(1)$ (peek)	$O(n)$	$O(\log n)$	$O(\log n)$
Graph (Adjacency List)	–	$O(V+E)$ traversal	$O(1)$ add edge	$O(1)$ remove edge

Table 1: Asymptotic time complexities for common operations on practical data structures in C#.

Interpretation.

- The entries show theoretical bounds under idealized conditions (uniform hashing, balanced trees, etc.).
- The constant factors hidden in $O(1)$ or $O(\log n)$ can differ widely depending on language runtime, allocation behavior, and cache locality.
- "Amortized" means that while a single insertion may occasionally cost $O(n)$ due to a resize, the average cost per operation tends to a constant as n grows.
- The dash (–) indicates that the operation is not meaningful or well-defined for that structure (e.g. "access by index'' on a hash table).

Practical viewpoint. For C#, this table provides only a first-order estimate of performance. Runtime factors such as garbage-collector pressure, array resizing thresholds, and cache behavior often dominate micro-benchmarks. Later sections revisit these complexities in context, showing how each structure achieves its asymptotic behavior and where its hidden costs appear in real implementations.

1.4 CPU Caches: A Practical Recap

Modern performance is dominated by how well code exploits the CPU's cache hierarchy. Caches are tiny, fast memories placed close to each core to hide main-memory latency. They reward *spatial locality* (touching neighbors) and *temporal locality* (reusing what you just touched). Because our data-structure choices strongly affect memory access patterns, a working model of caches pays dividends in real-world C#.

The Hierarchy at a Glance

Level	Typical Size	Latency (cycles)	Scope
L1 (I/D)	32–64 KB per core	∼3–5	Private to core (instruction/data)
L2	256–2048 KB per core	∼10–15	Private to core
L3	4–64 MB shared	∼30–60	Shared (last-level cache)
Main Memory (DRAM)	GBs	∼100–300	System-wide

Table 2: Representative cache sizes and latencies; actuals vary by microarchitecture. Note that (I/D) signifies that the L1 core is actually separated into two separate caches: **L1I** = L1 Instruction Cache holds decoded/fetched instructions, **L1D** = L1 Data Cache holds recently used data (loads/stores). This separation lets the core fetch instructions and access data *in the same cycle* without contention

Key concepts/definitions:

- **CPU clock cycle:** One tick of the processor's core clock - the metronome that paces when circuits sample data, move it between pipeline stages, and retire work. It is the fundamental timing unit that coordinates all synchronous work inside the core; performance counters cache "latencies in cycles" are measured against this metronome.
- **Latency cycle:** One tick of the CPU's clock used as a unit for measuring how long something takes on the processor. When we say "L1 cache latency ≈ 4 cycles," we mean: from issuing a load until its data is ready to use takes about 4 CPU clock cycles (under ideal conditions).
- **Cache lines:** Data moves in fixed-size blocks (typically 64 B). Touching `arr[i]` pulls in neighbors; your next few accesses are "free" if they hit the same line.
- **Locality:** Linear scans exploit spatial locality; hot loops exploit temporal locality.
- **Prefetching:** Hardware predicts sequential/strided patterns and loads lines early.
- **Associativity:** Caches are set-associative; adversarial access patterns (e.g., power-of-two strides) can cause *conflict misses* even if the working set "fits."
- **Hot paths:** Section of code that runs very frequently and therefore dominates your program's total runtime (and often its allocations). Optimizing hot paths yields outsized performance gains.

Writes and Coherence (Multi-Core Realities)

- **Write policies:** Most CPUs use *write-back* + *write-allocate* (writes update cache, lines are flushed later). *Write-through* exists but is rarer in hot paths.
- **Coherence:** Protocols (MESI/MOESI family) keep per-core caches consistent. False sharing (two threads modifying different fields on the same line) can tank performance.
- **Topology:** L1/L2 are private; L3 is shared as a last-level cache (LLC). On multi-socket systems, memory is *NUMA* (non-uniform memory access): remote-node DRAM is slower.

Implications for C# Data Structures

- **Contiguous beats scattered:** `T[]` and `List<T>` pack data; linked structures scatter nodes.
- **Boxing hurts:** `List<object>` or `ArrayList` store references to heap objects, degrading density and locality.
- **Stride matters:** Accessing every kth element widens gaps between touched lines and can defeat prefetchers.

Why Boxing Hurts Density and Locality

What "boxing" means. In C#, boxing wraps a *value type* (e.g., `int`) into a heap-allocated `object`. A `List<int>` stores raw ints contiguously; a `List<object>` that holds ints stores *references* to separate boxed-int objects elsewhere.

Contiguity vs. useful payload. Yes, `object[]` is a contiguous array—but it's contiguous for *pointers*, not for the *payload* you want. Each access becomes:

$$\text{load reference} \Rightarrow \text{follow pointer to boxed object} \Rightarrow \text{read value.}$$

That extra indirection breaks spatial locality and increases miss rates.

Cache-line density (typical x64 intuition).

Container	What's contiguous	Extra touch?	Useful data / 64B line
`List<int>` / `int[]`	the *values*	No	∼16 ints (4B each)
`List<object>` (boxed ints)	8B *references*	Yes	∼8 refs (data elsewhere)

A boxed `int` object itself is much larger than 4B (object header + padding), so even when fetched you typically get only a few values per cache line—far less dense than a raw `int[]`.

Locality & prefetching. Sequential scans over `List<int>` stream through contiguous memory; hardware prefetchers excel. With boxed values, each element may live at an unrelated address; prefetching can't predict targets, causing more cache/TLB misses (pointer chasing).

GC and allocation overhead. Boxing creates one heap object per value:

- More allocations ⇒ more GC pressure and write barriers.
- Larger working set (headers/padding) ⇒ poorer cache/TLB behavior.

Practical guidance.

- Prefer generic collections of value types (`List<int>`, `List<double>`) to avoid boxing.
- If heterogeneity is required, consider tagged structs / discriminated unions instead of `List<object>`.
- Keep hot paths contiguous (arrays, `List<T>`, `Span<T>`); avoid boxing in tight loops.

Listing 3: Stride patterns influence cache behavior

```
// Sequential: cache-friendly (spatial locality)
for (int i = 0; i < a.Length; i++) a[i] *= 2;

// Strided: can induce conflict/compulsory misses for large strides
for (int i = 0; i < a.Length; i += 64) a[i] *= 2;
```

How CPU Caches Evolved (A Short History)

From same-speed memory to deep hierarchies. Early CPUs and RAM ran at comparable speeds; as clock rates climbed, off-chip DRAM couldn't keep up due to physical distance and signaling limits. The answer was a small, on-die, same-speed memory: **L1 cache**. DRAM became a slower, external backing store.

Capacity pressure and L2. Programs outgrew L1. Making L1 huge was costly and slow, so a larger-but-slower cache was interposed: **L2**. Initially off-die, L2 moved on-die as integration improved,

serving as a fast buffer for working sets too big for L1.
Multi-core and L3. With many cores, bandwidth and data sharing became bottlenecks. A **shared L3** emerged as a staging area: it buffers DRAM fetches, supplies lines to multiple cores, and eases data handoff without round-tripping to main memory. Prefetchers and larger transfers anticipate streaming reads so data arrives before it's needed.
Today's picture. Each core has private L1 (instruction+data) and L2; the socket shares an inclusive or mostly inclusive **L3** (LLC - Last Level Cache). Coherence protocols synchronize private copies; write-back policies reduce DRAM traffic. Software that is linear, dense, and predictable rides the hierarchy efficiently; pointer-chasing and irregular access fight it.
Why this matters. Data structures that keep related values adjacent (arrays, `List<T>`, struct-of-arrays) let hardware exploit spatial/temporal locality. Node-heavy or boxed representations inflate footprint and induce cache misses. Even within the same $O(n)$, access patterns drive *orders-of-magnitude* differences in wall time.

A Few Practical Rules of Thumb

- Prefer **linear scans** over random probing when possible; batch work to reuse lines.
- Pack hot fields together; avoid false sharing across threads (different data on the same 64 B line).
- Choose **contiguous** containers for tight loops; reserve linked/circular lists for splice-heavy workflows.
- Measure: microarchitectural effects (prefetchers, associativity, replacement) can make real timings diverge from pure big-O.

2 Arrays and Spans

2.1 Conceptual Model and Memory Layout

Arrays are the most fundamental data structure in both system programming and high-level languages such as C#. Conceptually, an array represents a **contiguous block of memory** divided into equal-sized elements of a uniform type. Each element is identified by an integer index, allowing constant-time random access:

$$A[i] \text{ is located at memory address } \text{Base}(A) + i \times \text{sizeof}(T)$$

where `T` is the element type.

Contiguous Memory

In contiguous memory, the i-th element directly follows the $(i-1)$-th element, eliminating pointer indirection and allowing efficient CPU prefetching and cache locality. This makes sequential traversal extremely fast compared to pointer-based structures like linked lists.

In C#, both primitive arrays (`int[]`, `double[]`, etc.) and generic arrays (`T[]`) are true contiguous blocks allocated on the managed heap. The array object contains:

- a small *object header* for GC and type metadata,
- an integer field for the array length,
- the contiguous data region for the elements.

Each element's position can therefore be computed directly from its index, making `arr[i]` an $O(1)$ operation.

Listing 4: Creating and indexing a one-dimensional array

```
int[] data = new int[5]; // contiguous block of 5 integers
data[0] = 10; // O(1) access
Console.WriteLine(data[4]); // valid indices: -04
```

Because arrays are fixed in size, resizing requires allocation of a new block and copying the old elements—an $O(n)$ operation. This is one of the reasons higher-level structures like `List<T>` wrap arrays with automatic resizing logic.

Fixed Capacity and Indexing Semantics

The capacity of a C# array is immutable after creation:

```
int[] a = new int[3];
a[1] = 42; // OK
a = new int[5]; // legal, but allocates a new array entirely
```

The indexer performs bounds checking at runtime to ensure safety, raising an `IndexOutOfRangeException` if violated. Although this adds a negligible constant cost, the JIT compiler often eliminates redundant checks in optimized loops.

Single-Dimensional vs. Multidimensional Arrays

C# distinguishes between two different notions of "multiple dimensions":

1. **Rectangular (multidimensional) arrays**, declared with commas (`[,]`, `[,,]`), store elements in a single contiguous block arranged in row-major order. Every row has the same length.

Listing 5: Rectangular array

```
int[,] grid = new int[3, 4]; // 3 rows × 4 columns, contiguous layout
grid[1,2] = 9;
```

2. **Jagged arrays**, declared as arrays of arrays (`int[][]`), are not contiguous across rows. Each element of the top-level array is a separate subarray reference on the heap. This allows each "row" to have a different length.

Listing 6: Jagged array

```
int[][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[5];
jagged[2] = new int[1];
jagged[1][3] = 42; // Access through two references
```

Rectangular arrays are faster for purely numeric or matrix operations, since the entire data block is contiguous and cache-friendly. Jagged arrays, however, offer flexibility for irregular data and reduce wasted space when row lengths vary significantly.

Memory Visualization

Structure	Memory Layout (conceptual)
`int[3,4]`	0 1 2 3 4 5 6 7 8 9 10 11 (single contiguous block)
`int[][]`	Top-level array of 3 references → each points to its own smaller array on heap

From the perspective of hardware efficiency, this distinction is significant: rectangular arrays benefit from contiguous memory access, whereas jagged arrays introduce additional pointer indirection at each row access.

Summary

- Arrays provide the fastest possible indexing in C#: $O(1)$ with optimal cache locality.
- They have fixed capacity; resizing requires full reallocation and copy.
- Multidimensional arrays are contiguous and uniform; jagged arrays are flexible but add indirection.
- Understanding how these memory layouts differ is key to reasoning about performance in algorithms, data processing, and numerical computation.

2.2 Arrays in C#

Arrays in C# are implemented as first-class reference types within the `System` namespace. They provide strongly-typed, bounds-checked, contiguous storage for elements of uniform type. While syntactically simple, arrays in the CLR are sophisticated objects with well-defined runtime behavior and metadata.

Listing 7: Array creation and usage in C#

```
int[] arr = new int[5];
arr[0] = 42;
Console.WriteLine(arr.Length); // 5
```

This example allocates a contiguous block on the managed heap to hold five 32-bit integers. The array object itself stores:

- a header (object reference, type handle, and GC bookkeeping),
- an integer field for the array length,
- the contiguous region containing all element values.

All elements are automatically zero-initialized when the array is created.

Bounds Checking

Every array access in C# is automatically bounds-checked by the JIT-compiled code. If the index is outside the valid range $[0, \texttt{Length}-1]$, the runtime throws an `IndexOutOfRangeException`. This guarantees memory safety and prevents buffer-overflow vulnerabilities typical of unmanaged languages like C or C++.

Listing 8: Bounds checking in C# arrays

```
int[] nums = { 10, 20, 30 };
Console.WriteLine(nums[1]); // 20
Console.WriteLine(nums[3]); // throws IndexOutOfRangeException
```

Although the check introduces a small constant overhead, the JIT compiler aggressively optimizes it away inside tight loops whenever it can prove that the index remains in range. Consequently, sequential iteration performance approaches that of raw pointer access in unmanaged code.

Dynamic Allocation and Cost

Arrays are allocated dynamically on the managed heap, even when declared inside a method. This allocation involves:

- computing the total size (header + element storage),
- allocating memory from the appropriate GC generation (typically Gen 0),
- zero-initializing the block,
- storing a pointer to the array object on the stack.

Creating very large arrays (above ∼85 KB) places them in the Large Object Heap (LOH), which is collected less frequently and can lead to heap fragmentation if overused.

Listing 9: Dynamic allocation of arrays

```
void Allocate() {
       int[] data = new int[1_000_000]; // ~4 MB on heap
       // ... use array ...
} // eligible for GC after method exits
```

As a result, high-performance applications often reuse arrays or leverage `ArrayPool<T>` to minimize allocation churn and GC pressure.

Multidimensional and Jagged Arrays

C# supports both rectangular (true multidimensional) and jagged arrays.

Rectangular arrays (declared with commas) are single heap objects storing all elements contiguously in row-major order:

Listing 10: Rectangular array

```
int[,] matrix = new int[2, 3];
matrix[1, 2] = 7; // Access via two indices
```

Every row has the same number of columns, and access remains $O(1)$, but indexing involves slightly more arithmetic at runtime since the offset must be computed as:

$$\text{offset} = i \times \text{cols} + j$$

Jagged arrays (`T[][]`) are arrays of array references:

Listing 11: Jagged array

```
int[][] jagged = new int[2][];
jagged[0] = new int[3];
jagged[1] = new int[5];
```

Each subarray is an independent object on the heap. This means that jagged arrays are flexible (rows may differ in length) but not contiguous across dimensions. Every access like `jagged[i][j]` requires two pointer dereferences.

Performance Implications

- Rectangular arrays are optimal for dense, uniform data such as images or matrices; their cache-friendly layout favors numerical computation.
- Jagged arrays trade spatial locality for flexibility and are often used when row sizes vary.
- Both types perform runtime bounds checks for each dimension.
- For performance-critical workloads, low-level optimizations such as `Span<T>` or unsafe pointers can eliminate the overhead of managed bounds checking while preserving type safety in controlled contexts.

Key Takeaways

- Arrays in C# are reference types allocated on the managed heap.
- Element access is $O(1)$ with automatic safety via bounds checks.
- Resizing requires allocation and copy (no dynamic expansion).
- Rectangular arrays are contiguous; jagged arrays provide flexibility but add indirection.

2.3 Spans and Memory<T>

While arrays in C# offer contiguous storage and fast access, they are always allocated on the managed heap and carry the cost of garbage collection. In high-performance scenarios—for example, parsing binary data, streaming network buffers, or working with large value-type collections—the overhead of heap allocations and GC pressure can become substantial.

To address this, modern .NET introduces the family of stack-based and allocation-free abstractions `Span<T>`, `ReadOnlySpan<T>`, and `Memory<T>`. These types provide safe, zero-allocation slices into existing data, allowing developers to work with memory regions without copying or allocating new arrays.

Design and Motivation

`Span<T>` represents a *view* over a contiguous region of memory, regardless of where that memory physically resides: it may point into a managed array, a stack-allocated buffer, or even unmanaged memory obtained via interop.

Conceptually:

`Span<T>` = `(reference to first element, length)`

Unlike `T[]`, a span does not own the memory it references. It simply describes a segment of existing storage, ensuring type safety and bounds checking without allocating anything on the heap.

Listing 12: Creating spans over different sources

```
int[] data = { 1, 2, 3, 4, 5 };
Span<int> all = data; // span over entire array
Span<int> slice = all.Slice(1, 3); // span over [2,3,4]
```

```
Span<int> stackSpan = stackalloc int[3]; // allocated on stack
stackSpan[0] = 10;
stackSpan[1] = 20;
```

In this example, `data` lives on the heap, while `stackSpan` is created directly on the call stack using `stackalloc`, completely bypassing the GC.

Stack-Only Nature

A key property of `Span<T>` is that it is a *ref struct*. Ref structs are special stack-only types introduced in C# 7.2:

- They cannot be boxed or captured by closures.
- They cannot be assigned to fields of reference types.
- They cannot be returned from async methods or stored on the heap.

These restrictions guarantee that a span can never outlive the memory it references, eliminating the possibility of dangling references that plague unmanaged languages.

Listing 13: Stack-only safety of Span<T>

```
Span<int> GetSpan() {
       Span<int> s = stackalloc int[5];
       return s; // compile error: cannot return ref struct
}
```

The compiler enforces these rules statically, ensuring that a span's lifetime always remains valid.

ReadOnlySpan<T>

`ReadOnlySpan<T>` is the immutable counterpart of `Span<T>`. It provides the same slicing and indexing features but exposes only read operations. This is particularly useful when reading data from byte buffers, string literals, or read-only memory regions:

Listing 14: Using ReadOnlySpan<T>

```
ReadOnlySpan<char> chars = "Hello, world!".AsSpan();
Console.WriteLine(chars[7]); // 'w'
```

Internally, even `string.AsSpan()` avoids copying; it simply exposes a read-only window onto the existing string's memory.

Memory<T> and IMemoryOwner<T>

Because `Span<T>` is stack-only, it cannot be stored in fields or used across asynchronous calls. For these scenarios, .NET provides `Memory<T>`, a heap-storable wrapper that represents the same concept of a contiguous region, but without the stack restriction.

- `Span<T>` — stack-only, high-performance, for immediate/ephemeral access.
- `Memory<T>` — heap-storable, safe to keep in objects and async methods.

A `Memory<T>` can be converted into a `Span<T>` at any time via the `.Span` property, giving access to fast stack semantics inside the performance critical code path.

Listing 15: Memory<T> to Span<T> conversion

```
Memory<byte> buffer = new byte[1024];
Span<byte> slice = buffer.Span.Slice(100, 50);
```

For advanced pooling scenarios, the `IMemoryOwner<T>` interface can provide lifetime control over the underlying memory, useful when integrating with `ArrayPool<T>` or custom buffer managers.

Safety and Type Guarantees

Both `Span<T>` and `Memory<T>` are fully type-safe:

- Bounds are always checked, preventing buffer overruns.
- The JIT compiler may elide redundant checks for optimized loops.
- Access is performed directly against raw memory, but without unsafe pointers.

They are therefore the bridge between managed safety and native efficiency.

Performance Characteristics

- No heap allocations: creating or slicing spans is $O(1)$ and zero-cost.
- Indexing and iteration performance equal to raw arrays.
- Stack spans are ideal for short-lived buffers (parsing, I/O framing, etc.).
- `Memory<T>` enables async and object-oriented use without compromising correctness.

Summary: Choosing Between Array, Span, and Memory

Type	Allocation	Typical Use Case
`T[]`	Heap	Persistent storage, general use
`Span<T>`	Stack (or borrowed array)	High-performance, stack-local access
`ReadOnlySpan<T>`	Stack (read-only)	Safe read-only slicing, parsing text/binary
`Memory<T>`	Heap (ref-safe)	Long-lived or async-friendly memory regions

Together, these types enable developers to write allocation-free, cache-friendly, type-safe code that operates directly on memory. They form the cornerstone of modern low-latency .NET programming, providing C-like performance while preserving the memory safety guarantees of managed code.

2.4 Use Cases and Performance Notes

Spans and memory slices are not abstract curiosities—they are the foundation of many high-performance programming patterns in modern C#. They allow developers to process data efficiently by reusing existing memory rather than copying or reallocating it. Because they are allocation-free, small, and cache-efficient, spans are used extensively in the .NET Base Class Library itself (e.g., in `System.Text.Json`, `System.IO.Pipelines`, and the core numeric/vectorization APIs).

Vectorized and Bulk Operations

Since `Span<T>` provides direct access to contiguous memory, it is ideal for **vectorized operations** and numerical loops. The JIT compiler can unroll loops and apply SIMD (Single Instruction, Multiple Data) optimizations automatically when possible.

Listing 16: Bulk arithmetic using Span<T>

```
Span<int> data = stackalloc int[4] { 1, 2, 3, 4 };
for (int i = 0; i < data.Length; i++)
data[i] *= 2; // operates directly on stack memory
```

Such operations incur no heap allocation, and all writes go directly to the target memory region. Combined with hardware intrinsics, this enables C-like performance while preserving type safety.

Zero-Copy Slicing and Subviews

Traditional approaches to extracting subarrays involve creating a new array and copying elements:

```
int[] slice = data.Skip(10).Take(5).ToArray(); // allocates new array
```

With spans, slicing is zero-cost:

Listing 17: Zero-allocation slicing

```
Span<int> sub = data.AsSpan(10, 5); // view of existing memory
```

The new span simply adjusts its starting reference and length; no copying occurs. This pattern is essential in large-scale parsing, streaming, and message decoding systems.

Data Streaming and Buffer Pipelines

In high-throughput I/O, repeatedly allocating arrays or buffers can destroy performance. Instead, `Span<T>` and `Memory<T>` allow direct manipulation of pooled or preallocated buffers.

Listing 18: Processing byte streams using Span<T>

```
void ProcessBuffer(Span<byte> buffer)
{
       // interpret first bytes as header
       var header = buffer.Slice(0, 8);
       var payload = buffer.Slice(8);

       // process payload in-place without copying
       for (int i = 0; i < payload.Length; i++)
       payload[i] ^= 0xFF; // bitwise inversion
}
```

This approach eliminates transient heap allocations and makes back-pressure management much easier in streaming architectures. Libraries like `System.IO.Pipelines` build on this concept, allowing millions of messages to be processed per second without significant GC involvement.

Contiguous vs. Non-Contiguous Memory Layout

To understand why spans are so effective, consider how CPU caching interacts with memory layout. Contiguous data enables predictable prefetching and minimizes cache misses, while scattered references force the CPU to load multiple cache lines and follow pointers.

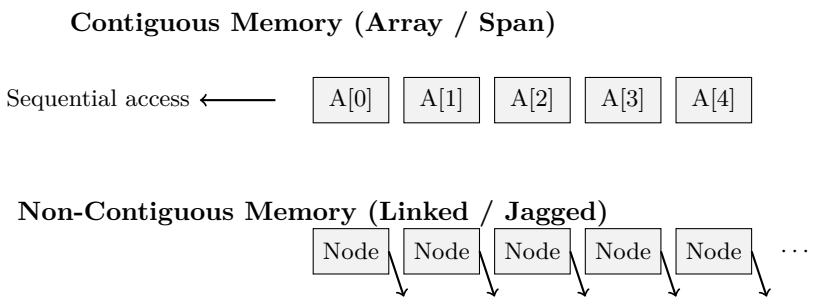


Figure 1: Contiguous arrays enable sequential prefetching; scattered layouts force pointer chasing.

In contiguous storage, accessing $A[i]$ and $A[i+1]$ loads consecutive bytes from memory. In contrast, linked or jagged structures store nodes or row references at arbitrary locations, requiring additional pointer dereferences and breaking spatial locality.

Practical Performance Notes

- **Use spans for transient, performance-critical sections.** Keep them on the stack whenever possible (`stackalloc`, parsing, buffer slices).
- **Avoid copying unless ownership semantics change.** A slice via `Span<T>` or `Memory<T>` is always cheaper than duplication.
- **Combine with vectorization.** The JIT's hardware intrinsics and the `System.Numerics.Vector<T>` API integrate seamlessly with spans.
- **Beware of lifetime.** A span must never outlive the data it views; use `Memory<T>` for async operations or object fields.

Summary

Spans unify the performance of unmanaged pointer arithmetic with the safety of managed code. They allow developers to express algorithms directly in terms of logical data windows—rather than copies—and are a key tool for modern systems-level C#. The combination of `Span<T>` and `Memory<T>` effectively replaces ad hoc buffering strategies with a formal, type-safe abstraction that scales from stack allocations to long-lived memory pools.

3 Lists and Dynamic Arrays

3.1 Amortized Growth and Resizing

A `List<T>` in C# provides dynamic, array-backed storage that grows as new elements are added. Unlike a fixed-length array, a list automatically manages its internal capacity, expanding as needed to accommodate additional data. Understanding how this growth occurs is essential for predicting both performance and memory behavior.

The Need for Dynamic Resizing

Because arrays in C# have a fixed capacity, growing a collection cannot be achieved without allocating a new array and copying the existing elements into it. Manually performing this operation for every insertion would be prohibitively expensive, since each resize would take $O(n)$ time.

To solve this, dynamic arrays such as `List<T>` grow their capacity in *geometric steps* rather than one element at a time. This strategy ensures that while occasional resizes are expensive, they occur rarely enough that the *average* cost per insertion remains constant.

The Doubling Strategy

When the internal backing array becomes full, the list allocates a new array—typically **double** the previous capacity—and copies all elements into the new space.

Listing 19: Simplified growth logic of List<T>

```
public void Add(T item)
{
	if (_size == _items.Length)
	EnsureCapacity(_size + 1); // trigger resize if needed
	_items[_size++] = item;
}

private void EnsureCapacity(int min)
{
	if (_items.Length < min)
	{
		int newCapacity = _items.Length == 0 ? 4 : _items.Length * 2;
		if (newCapacity < min) newCapacity = min;
		T[] newItems = new T[newCapacity];
		Array.Copy(_items, newItems, _size);
		_items = newItems;
	}
}
```

In practice, the growth factor may vary slightly (commonly $\times 2$, but occasionally $\times 1.5$ for memory efficiency). The key insight is that capacity expansion grows *geometrically*, never linearly.

Amortized $O(1)$ Append

Let us analyze why repeated appends are considered $O(1)$ *on average* despite occasional $O(n)$ copies.

Assume a list doubles its capacity whenever full. For n insertions starting from an empty list:

- Most insertions (roughly $n - \log_2 n$ of them) cost $O(1)$.
- A few insertions (those that trigger resizing) cost $O(k)$, where k is the size of the list before resizing.

The total work over all n insertions is approximately:

$$n + \frac{n}{2} + \frac{n}{4} + \cdots < 2n$$

Thus, the *average* cost per insertion tends to a constant, yielding an amortized complexity of $O(1)$ for appends.

Amortized Cost Intuition

Analogy: Expanding a dynamic array is like buying storage boxes in bulk. You occasionally pay for a large upgrade, but the cost per item averaged over time remains nearly constant.

Memory Tradeoffs

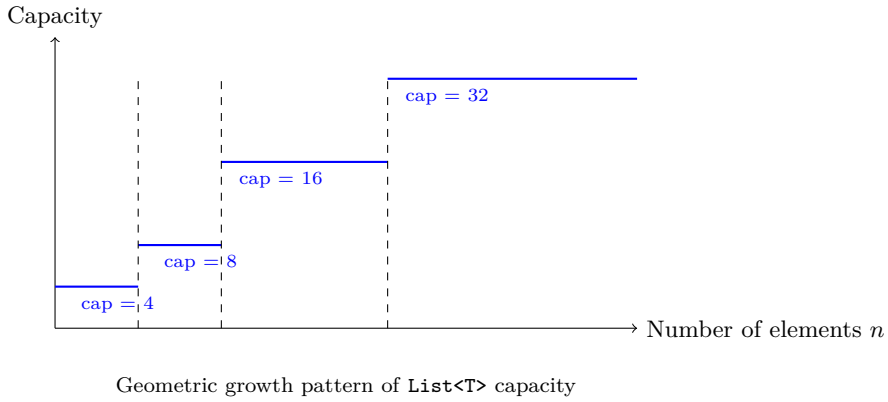
The geometric growth strategy balances time and space:

- **Time efficiency:** Resizing is infrequent, keeping append operations fast on average.
- **Space overhead:** At most, half of the allocated slots may be unused immediately after a resize.

Developers can mitigate space waste by using `List<T>.TrimExcess()` or setting the initial capacity if the approximate size is known:

```
var list = new List<int>(10_000); // preallocate capacity
```

Visualizing Growth



Geometric growth pattern of `List<T>` capacity

Figure 2: Capacity doubles at each resize event, resulting in amortized $O(1)$ appends.

Summary

- `List<T>` expands geometrically (typically doubling) when full.
- Each resize costs $O(n)$, but occurs infrequently.
- Average append cost over many insertions is constant time, $O(1)$ amortized.
- Memory overhead is bounded by a constant factor of the used capacity.

This combination of contiguous storage, amortized $O(1)$ growth, and bounded overhead explains why `List<T>` is one of the most efficient and widely used collection types in modern C# programming.

3.2 C# List<T> Implementation Details

The `List<T>` class is a thin, efficient wrapper around a resizable array. It implements `IList<T>`, `IReadOnlyList<T>`, and `ICollection<T>`, providing both indexed access and dynamic growth while maintaining contiguous memory storage. Internally, it is remarkably simple—its performance stems not from complex algorithms but from careful engineering around array semantics and memory management.

Internal Fields

At its core, a `List<T>` maintains only three key fields:

Listing 20: Internal fields of List<T> (simplified)

```
public class List<T>
{
        private T[] _items; // backing array, contiguous storage
        private int _size; // number of used elements
        private int _version; // modification counter for enumerators
}
```

- **__items**: The backing array that holds elements in contiguous memory. Its capacity may exceed the logical count.
- **__size**: The logical number of elements currently in use. Always ≤ _items.Length.
- **__version**: Tracks structural modifications to invalidate stale enumerators, ensuring iteration safety.

The class also defines a shared empty array instance for new lists:

```
private static readonly T[] _emptyArray = new T[0];
```

This avoids unnecessary allocations for lists created and never populated.

Adding Elements

Insertion occurs at index _size and increments it after assignment. When the list reaches full capacity, it triggers a resize through the geometric doubling mechanism discussed earlier.

Listing 21: Simplified Add() method

```
public void Add(T item)
{
        if (_size == _items.Length)
        EnsureCapacity(_size + 1);
        _items[_size++] = item;
        _version++;
}
```

Because elements are written sequentially into a contiguous array, appending is cache-friendly and usually $O(1)$ amortized.

Cache Locality and Contiguous Access in Arrays

Concept. When we say that writing or reading sequentially from a contiguous array is *cache-friendly*, we mean that the CPU can exploit its memory hierarchy efficiently. Accesses proceed in predictable, adjacent memory addresses, allowing the processor's cache system to prefetch and buffer upcoming data before it is explicitly requested.

1. The Memory Hierarchy. Modern CPUs are much faster than main memory (RAM). To bridge this gap, they maintain several small, fast caches:

Level	Typical Size	Latency (cycles)	Scope
L1 Cache	32–64 KB	3–5	Per core (fastest)
L2 Cache	256–512 KB	10–15	Per core
L3 Cache	4–30 MB	30–60	Shared among cores
Main Memory (RAM)	GBs	100–300	Global, slowest

When data is not in cache, the CPU must wait hundreds of cycles for a main memory fetch—a

significant stall compared to arithmetic or branch operations.

2. Cache Lines. Caches move data between memory and CPU in fixed-size blocks called *cache lines*, typically 64 bytes. When your code accesses `arr[i]`, the CPU loads not just that element but the entire 64-byte line containing nearby elements. For an `int[]` (4 bytes per element), each line holds 16 integers.
Thus, reading `arr[i]` implicitly loads `arr[i+1]`, `arr[i+2]`, ... into cache as well.

3. Sequential (Contiguous) Access. Iterating linearly over an array matches the CPU's natural prediction pattern:

```
for (int i = 0; i < arr.Length; i++)
arr[i] *= 2;
```

The CPU recognizes the ascending address sequence and prefetches the next cache lines automatically. Most accesses hit the fast L1 or L2 cache, minimizing stalls and maximizing throughput. This is what we mean by *cache-friendly access*.

4. Pointer-Chasing and Non-Contiguous Access. In contrast, data structures such as linked lists scatter elements across memory. Each node references the next via a pointer, forcing the CPU to load from unpredictable locations. Prefetchers cannot anticipate where the next element resides, so nearly every access incurs a cache miss:

```
LinkedList<int> list = ...;
foreach (var x in list) sum += x; // unpredictable addresses
```

This pattern defeats spatial locality and can make linked-list traversals tens of times slower than equivalent array loops.

5. Temporal and Spatial Locality.

- **Spatial locality:** accessing nearby memory locations consecutively allows one cache line to serve many operations.
- **Temporal locality:** recently accessed data is likely to be reused soon, keeping it hot in cache.

Contiguous arrays benefit from both properties—each cache line fetched serves multiple nearby elements and often gets reused quickly.

6. Sequential Writes. Writing sequentially to contiguous memory is also cache-efficient. When a cache line is modified, it stays in the CPU's write-back buffer until full, then flushes as a single 64-byte block back to RAM. This batching minimizes bus traffic and maximizes write throughput.

7. Practical Implications.

- Arrays and `List<T>` leverage contiguous storage to achieve predictable, near-optimal performance.
- Data structures with pointer indirection (linked lists, trees, jagged arrays) incur higher latency due to scattered memory access.
- Sequential, predictable loops enable automatic hardware prefetching and SIMD vectorization.
- Even within $O(n)$ complexity, cache locality can produce 10–50× runtime differences.

Summary. Contiguous memory access aligns perfectly with CPU caching behavior. Each fetched cache line contains several elements used immediately afterward, keeping the processor's pipelines full and avoiding costly main-memory stalls. This is why arrays and lists vastly outperform pointer-based structures in tight numerical or streaming loops.

Access and Indexing

Access via the indexer simply reads or writes to the internal array with bounds checking:

Listing 22: Indexer in List<T>

```
public T this[int index]
{
	get
	{
		if ((uint)index >= (uint)_size)
		throw new ArgumentOutOfRangeException();
		return _items[index];
	}
	set
	{
		if ((uint)index >= (uint)_size)
		throw new ArgumentOutOfRangeException();
		_items[index] = value;
		_version++;
	}
}
```

Both read and write operations are $O(1)$, identical in cost to direct array indexing. The unsigned comparison trick `(uint)index >= (uint)_size` is an idiomatic optimization: it collapses two range checks (negative and upper bound) into one.

Enumeration Semantics

Enumeration over a list uses the `List<T>.Enumerator` struct, which is a **value type** (not a class) to avoid heap allocation during iteration. It captures a snapshot of the list's _version at construction:

Listing 23: Simplified enumerator logic

```
public struct Enumerator : IEnumerator<T>
{
	private readonly List<T> _list;
	private int _index;
	private readonly int _version;
	private T _current;

	public bool MoveNext()
	{
		List<T> localList = _list;
		if (_version == localList._version && _index < localList._size)
		{
			_current = localList._items[_index++];
			return true;
		}
		return MoveNextRare(); // handles version mismatch
	}
}
```

If the list is structurally modified (insert or remove) during enumeration, the _version field changes, causing subsequent calls to throw an `InvalidOperationException`. This guarantees deterministic behavior and prevents silent iterator corruption.

Capacity vs. Count

A key distinction in `List<T>` is between:

- **Count** — the number of elements currently stored (_size).
- **Capacity** — the size of the internal array (_items.Length).

The capacity may be larger than the count to accommodate future growth. Developers can query or adjust capacity directly:

```
List<int> nums = new List<int>(8);
Console.WriteLine(nums.Capacity); // 8
Console.WriteLine(nums.Count); // 0
```

Using `TrimExcess()` reclaims unused memory by resizing the backing array to match the current count.

Copy Behavior

Because `List<T>` stores elements in a single contiguous array, copy operations can exploit fast bulk memory transfer using `Array.Copy`. This ensures that cloning or range copying is linear but highly optimized.

Performance Notes

- **Iteration and index access** are as fast as arrays—single pointer dereference and bounds check.
- **Insert or remove in the middle** requires shifting trailing elements; cost $O(n)$.
- **Add** is $O(1)$ amortized; resize events are rare.
- **Enumeration** is allocation-free because the enumerator is a struct.
- **Capacity management** gives predictable scaling and avoids per-insert allocations.

List<T> vs. Array Summary

Feature	`T[]`	`List<T>`
Length / Count	Fixed	Grows dynamically
Index access	$O(1)$	$O(1)$
Add / Append	Not supported	$O(1)$ amortized
Insert / Remove mid-array	Manual (copy)	$O(n)$ shift
Bounds checking	Always	Always
Heap allocation count	1	2 (wrapper + array)
Enumeration cost	Low	Low (struct enumerator)

Summary

`List<T>` achieves near-array performance for access and iteration while providing automatic resizing and safe memory management. Its design balances simplicity with pragmatic engineering: a single contiguous array, a size counter, and a version tag. Because of this, it serves as the foundation for many higher-level collection types in the .NET ecosystem.

3.3 Common Operations and Benchmarks

Beyond amortized appends, understanding the practical cost of insertion, removal, and lookup operations is essential for writing performance-aware C# code. While `List<T>` achieves near-array performance for sequential operations, certain patterns—particularly mid-array modifications—carry higher costs due to memory shifting and cache effects.

Complexity Overview

The following table summarizes both theoretical time complexity and empirical behavior for the most common `List<T>` operations.

Operation	Asymptotic Cost	Average Cost (ns)	Memory Behavior	Notes
`Add(item)`	$O(1)$ amortized	$\approx$ 5–15	Append to contiguous array	Doubling resize when full
`Insert(i,item)`	$O(n)$	$\approx$ 200–600	Shift tail elements up by one	Proportional to distance from end
`RemoveAt(i)`	$O(n)$	$\approx$ 200–600	Shift tail elements down by one	Middle removals expensive
`Remove(item)`	$O(n)$ search + $O(n)$ shift	$\approx$ 500+	Linear scan for equality	Combine with `Contains()` cost
`Contains(item)`	$O(n)$	$\approx$ 150–300	Sequential linear search	Uses `EqualityComparer<T>`
`IndexOf(item)`	$O(n)$	$\approx$ 150–300	Sequential linear search	Same cost as `Contains()`
`Clear()`	$O(n)$	$\approx$ 100–400	Zeroes references	Retains allocated capacity
Iteration (`foreach`)	$O(n)$	$\approx$ 3–10 per element	Sequential, cache-friendly	Allocation-free struct enumerator

Table 3: Representative performance characteristics for common `List<T>` operations. Timings assume primitive types and modern JIT optimizations on x64.

Sequential iteration and appending dominate the performance landscape. Both are extremely fast because the data is contiguous and pre-allocated. Operations that modify the interior of the list, however, incur full element shifts using `Array.Copy`, which is highly optimized but still linear in cost.

Iteration and Cache Behavior

`List<T>` iteration is among the fastest in managed code because elements are laid out consecutively in memory. The CPU's cache prefetcher can predict subsequent accesses, making tight loops and vectorized arithmetic particularly efficient:

Listing 24: High-performance iteration over List<T>

```
List<float> data = Enumerable.Range(0, 1_000_000).Select(i => (float)i).ToList();
float sum = 0;
for (int i = 0; i < data.Count; i++)
sum += data[i];
```

Because the `Enumerator` is a struct, a `foreach` loop compiles to code comparable to an indexed `for` loop, avoiding heap allocations altogether.

Pre-Sizing with Capacity

When the approximate number of elements is known in advance, specifying an initial `Capacity` can eliminate expensive resizing operations and reduce GC activity.

Listing 25: Using Capacity to preallocate

```
int expected = 100_000;
var list = new List<int>(expected); // allocate backing array upfront
for (int i = 0; i < expected; i++)
list.Add(i);
```

Without pre-sizing, the list would undergo roughly $\log_2 n$ resizes, each involving a full copy of the existing elements. Pre-allocating thus ensures all insertions remain purely $O(1)$ and avoids transient arrays that add GC pressure.

Capacity Management Tips

- **Estimate capacity** if you know the data volume; avoid repeated resizing.
- **TrimExcess()** to reclaim unused space once growth stabilizes.
- **Reserve then reuse:** reusing a single list instance across frames or iterations avoids both allocation and garbage collection.

Practical Notes

- **Append-heavy workloads:** lists are ideal for data aggregation, logging buffers, or batch construction.
- **Frequent mid-list insertions or deletions:** consider `LinkedList<T>` or specialized structures (e.g. deque, gap buffer).
- **Membership testing:** for repeated lookups, prefer `HashSet<T>` or `Dictionary<K,V>` to avoid $O(n)$ scans.
- **Iteration:** remains one of the most cache-efficient traversal patterns in managed languages.

Summary

- `List<T>` achieves near-array performance for access and iteration.
- Insertions and removals in the middle are $O(n)$ due to shifting elements.
- `Capacity` tuning can dramatically reduce transient allocations.
- For read-mostly or append-heavy data, lists are among the most efficient containers in C#.

4 Linked Lists

4.1 Singly Linked Lists

A **singly linked list** is a sequence of *nodes*, each storing a value and a pointer (reference) to the next node. Unlike arrays (contiguous memory), a linked list's elements can reside anywhere on the heap; the list's logical order is maintained by pointers. This decouples logical adjacency from physical adjacency, making certain operations (e.g., inserting after a known node) $O(1)$ while random access becomes $O(n)$ due to pointer traversal.

Listing 26: Singly linked list node structure

```
public class Node<T> {
       public T Value;
       public Node<T>? Next;
}
```

Core Operations and Costs

- **PushFront (prepend)**: insert at head in $O(1)$.
- **Append (push back)**: $O(n)$ without a tail pointer; $O(1)$ with a maintained tail.
- **InsertAfter(node)**: $O(1)$ given a node reference.
- **Find(predicate / value)**: $O(n)$ traversal.
- **RemoveAfter(node)** or **PopFront**: $O(1)$ given a node reference / head.

A Minimal Manual Implementation (C#)

Listing 27: A minimal singly linked list with head (and optional tail)

```
public class SinglyLinkedList<T>
{
	public Node<T>? Head { get; private set; }
	public Node<T>? Tail { get; private set; } // optional; maintained for O(1)
	    append
	public int Count { get; private set; }

	// O(1): insert new node at the front
	public void PushFront(T value)
	{
		var n = new Node<T> { Value = value, Next = Head };
		Head = n;
		if (Tail is null) Tail = n;
		Count++;
	}

	// O(1) if Tail is maintained; else O(n) to find last
	public void Append(T value)
	{
		var n = new Node<T> { Value = value, Next = null };
		if (Head is null) { Head = Tail = n; Count = 1; return; }
		Tail!.Next = n; // Tail is non-null if list non-empty
		Tail = n;
		Count++;
	}

	// O(1): insert after a known node handle
	public void InsertAfter(Node<T> node, T value)
	{
		if (node is null) throw new ArgumentNullException(nameof(node));
		var n = new Node<T> { Value = value, Next = node.Next };
		node.Next = n;
		if (Tail == node) Tail = n;
		Count++;
	}

	// O(1): remove first element if present
	public bool PopFront(out T? value)
	{
		if (Head is null) { value = default; return false; }
		value = Head.Value;
		Head = Head.Next;
		if (Head is null) Tail = null;
		Count--;
		return true;
	}

	// O(n): remove first node whose value equals target (using
	    EqualityComparer<T>)
	public bool RemoveFirst(T target)
	{
		if (Head is null) return false;
		var cmp = EqualityComparer<T>.Default;

		if (cmp.Equals(Head.Value, target))
```

```
            {
                Head = Head.Next;
                if (Head is null) Tail = null;
                Count--;
                return true;
            }

            var prev = Head;
            var cur = Head.Next;
            while (cur is not null)
            {
                if (cmp.Equals(cur.Value, target))
                {
                    prev.Next = cur.Next;
                    if (Tail == cur) Tail = prev;
                    Count--;
                    return true;
                }
                prev = cur;
                cur = cur.Next;
            }
            return false;
        }

        // O(n): search by predicate
        public Node<T>? Find(Predicate<T> match)
        {
            for (var cur = Head; cur is not null; cur = cur.Next)
            if (match(cur.Value)) return cur;
            return null;
        }

        // O(n): in-place iterative reverse (classic 3-pointer technique)
        public void Reverse()
        {
            Node<T>? prev = null;
            var cur = Head;
            Tail = Head; // new tail becomes old head
            while (cur is not null)
            {
                var next = cur.Next;
                cur.Next = prev;
                prev = cur;
                cur = next;
            }
            Head = prev;
        }

        // Simple enumerator (allocation-free if exposed as IEnumerable<T> with
            struct enumerator)
        public IEnumerable<T> AsEnumerable()
        {
            for (var cur = Head; cur is not null; cur = cur.Next)
            yield return cur.Value!;
        }
    }
```

Usage Example

Listing 28: Basic usage

```
var list = new SinglyLinkedList<int>();
list.PushFront(3);
list.PushFront(2);
list.Append(4); // list: 2 -> 3 -> 4
var node3 = list.Find(x => x == 3)!;
list.InsertAfter(node3, 99); // list: 2 -> 3 -> 99 -> 4
foreach (var x in list.AsEnumerable()) Console.WriteLine(x);
list.Reverse(); // list: 4 -> 99 -> 3 -> 2
```

Edge Cases and Pitfalls

- **Head/Tail maintenance**: after removals or reverse, ensure `Tail` remains correct.
- **Empty vs single-node list**: handle null transitions carefully.
- **Enumeration during mutation**: if you expose `IEnumerable<T>`, consider a version check (like `List<T>`) to detect concurrent modification.
- **Memory locality**: nodes are scattered on the heap (pointer chasing), so traversal is cache-unfriendly compared to arrays/`List<T>`.

Performance Notes

- **Strength**: $O(1)$ inserts/deletes with a node handle; excellent for queue-like head operations or splice edits.
- **Weakness**: poor random access; per-element overhead (object header + pointer) and GC pressure (many small objects).
- **When to prefer**: frequent inserts/removes near head or via known nodes; when stability of iterators across insertions is required.

Linked Lists vs Arrays (Operational Tradeoffs)

- Arrays/`List<T>`: $O(1)$ indexing, contiguous, cache-friendly; middle insert/remove costs $O(n)$ due to shifting.
- Singly Linked List: $O(1)$ insert/remove *given a node*; $O(n)$ access/search; scattered in memory (cache misses).
- If you need frequent tail appends, maintain a **tail pointer** for $O(1)$ append; otherwise appends degrade to $O(n)$.

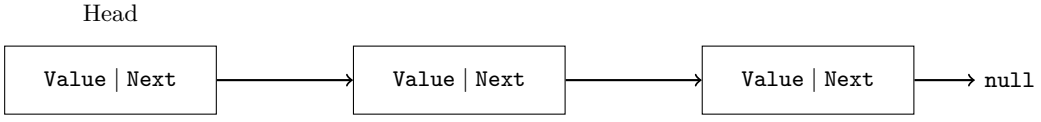


Figure 3: Singly linked list: nodes connected by `Next` references.

Singly linked lists trade cache locality and random access for constant-time structural edits at known positions. They are a foundational pointer-based structure and a useful foil for understanding why contiguous arrays (and `List<T>`) are so fast in tight loops.

4.2 Doubly and Circular Lists

A **doubly linked list** augments each node with a backward link (`Prev`) in addition to the forward link (`Next`). This enables $O(1)$ deletion of a *known* node without searching for its predecessor, and simplifies splice operations. A **circular** variant connects the ends so that the last node's `Next` points to the first, and the first node's `Prev` points to the last. Using a **sentinel** (a dummy header node) eliminates edge cases: in an empty list the sentinel's `Next` and `Prev` point to itself; in a non-empty list, all insertions and deletions become uniform pointer rewrites around the target location.

Why backward links and sentinels help

- $O(1)$ **deletion with a handle:** in a singly list, deleting a node requires its predecessor; with `Prev`, we can unlink directly.
- **No head/tail special cases:** a circular *sentinel* acts as both head and tail placeholder; code paths don't branch on empty/singleton cases.
- **Splice-friendly:** moving whole sublists is pointer surgery (constant-time) when you have both directions and a sentinel anchor.

Node and List (sentinel-based) — C#

Listing 29: Doubly linked list with circular sentinel

```
public class DNode<T>
{
	public T Value = default!;
	public DNode<T>? Prev;
	public DNode<T>? Next;
	public DNode() { } // sentinel uses default ctor
	public DNode(T value) { Value = value; }
}

public class DLinkedList<T>
{
	// Circular sentinel: when empty, Sentinel.Next == Sentinel.Prev == Sentinel
	public DNode<T> Sentinel { get; } = new DNode<T>();
	public int Count { get; private set; }

	public DLinkedList()
	{
		Sentinel.Next = Sentinel;
		Sentinel.Prev = Sentinel;
	}

	// Insert new node 'n' after node 'p' (p is a handle; O(1))
	public void InsertAfter(DNode<T> p, DNode<T> n)
	{
		n.Next = p.Next;
		n.Prev = p;
		p.Next!.Prev = n;
		p.Next = n;
		Count++;
	}

	// Insert new node 'n' before node 'p' (O(1))
	public void InsertBefore(DNode<T> p, DNode<T> n)
	{
		n.Prev = p.Prev;
```

```
            n.Next = p;
            p.Prev!.Next = n;
            p.Prev = n;
            Count++;
        }

        // Convenience: prepend / append relative to sentinel
        public DNode<T> Prepend(T value)
        {
            var n = new DNode<T>(value);
            InsertAfter(Sentinel, n);
            return n;
        }

        public DNode<T> Append(T value)
        {
            var n = new DNode<T>(value);
            InsertBefore(Sentinel, n); // before sentinel == at tail
            return n;
        }

        // Remove node 'n' (O(1)); does nothing for sentinel
        public void Remove(DNode<T> n)
        {
            if (n == Sentinel) return;
            n.Prev!.Next = n.Next;
            n.Next!.Prev = n.Prev;
            n.Prev = n.Next = null; // help GC
            Count--;
        }

        public bool IsEmpty => Count == 0;

        // Simple forward iteration
        public IEnumerable<T> Forward()
        {
            for (var cur = Sentinel.Next; cur != Sentinel; cur = cur!.Next)
            yield return cur!.Value;
        }

        // Simple backward iteration
        public IEnumerable<T> Backward()
        {
            for (var cur = Sentinel.Prev; cur != Sentinel; cur = cur!.Prev)
            yield return cur!.Value;
        }
    }
```

Usage

Listing 30: Uniform insertion/deletion via sentinel

```
var dl = new DLinkedList<int>();
var a = dl.Append(1); // [1]
var b = dl.Append(2); // [1,2]
var c = dl.Prepend(0); // [0,1,2]
dl.InsertAfter(a, new DNode<int>(99)); // [0,1,99,2]
dl.Remove(b); // [0,1,99]
```

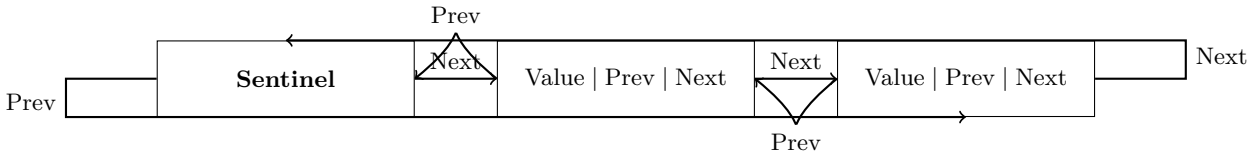


Figure 4: Circular doubly linked list with a sentinel node: uniform forward/backward links.

Complexity and Practical Notes

- **Given a node handle:** `InsertBefore/After` and `Remove` are $O(1)$.
- **Search by value:** still $O(n)$; use a hash map if you need fast membership tests.
- **Sentinel pattern:** no special cases for empty or head/tail edits; code remains branch-light and robust.
- **Memory/GC:** each node is a separate object (header + two pointers); traversal is not cache-friendly.
- **Iterators:** if exposing `IEnumerable<T>`, consider a version counter to detect concurrent modification (like `List<T>`).

When to use Doubly/Circular + Sentinel

- Frequent splicing or deletions at arbitrary positions when you already hold node references.
- Deques and LRU caches (move-to-front in $O(1)$), often combined with a dictionary for $O(1)$ lookup.
- Uniform insertion/deletion logic without branching on head/tail/empty cases.

4.3 C# `LinkedList<T>` Internals

The .NET `LinkedList<T>` is implemented as a **circular, doubly linked** list that maintains references to both ends (`First`, `Last`). Each element is wrapped in a `LinkedListNode<T>` that stores:

- the element `Value` (generic, no boxing for value types),
- pointers to `Next` and `Previous` nodes,
- a back-pointer to the owning `LinkedList<T>` (for validation).

This design enables $O(1)$ **insertion and deletion** when you already have a node handle (e.g., `AddAfter(node, value)`, `AddBefore(node, value)`, `Remove(node)`), and supports efficient deque-like head/tail operations.

Shape and Sentinels

Internally, the list keeps a single reference `head` to the first node. When non-empty, the structure is circular:

$$\texttt{head.Prev} \equiv \texttt{Last}, \qquad \texttt{Last.Next} \equiv \texttt{head}.$$

This circularity makes edge cases uniform (empty vs. singleton vs. general) and keeps insertion/deletion code branch-light. Public properties `First` and `Last` are exposed for convenience.

Key Types (Simplified Sketch)

Listing 31: Essential fields and relationships

```
public sealed class LinkedListNode<T>
{
        internal LinkedList<T>? List; // owning list (null if not in a list)
        internal LinkedListNode<T>? Next;
        internal LinkedListNode<T>? Prev;
        public T Value { get; internal set; }

        internal LinkedListNode(T value) { Value = value; }
}

public class LinkedList<T> : ICollection<T>, IReadOnlyCollection<T>
{
        internal LinkedListNode<T>? head; // first node (null if empty)
        public LinkedListNode<T>? First => head;
        public LinkedListNode<T>? Last => head is null ? null : head.Prev;
        public int Count { get; private set; }
        private int _version; // enumeration invalidation
```

The `LinkedListNode<T>.List` back-pointer allows the implementation to:

- prevent inserting a node that already belongs to another list,
- invalidate a node upon removal (set `List = null`) so misuse can be detected,
- support $O(1)$ membership checks when operating with node handles.

O(1) Insertions with a Node Handle

Given any existing node `x` in the list, `AddAfter(x, value)` and `AddBefore(x, value)` run in constant time by rewiring a fixed set of pointers.

Listing 32: Pointer rewiring for AddAfter (simplified)

```
public LinkedListNode<T> AddAfter(LinkedListNode<T> node, T value)
{
        if (node.List != this) throw new InvalidOperationException();
        var newNode = new LinkedListNode<T>(value) { List = this };

        // link newNode between node and node.Next
        newNode.Next = node.Next; // successor
        newNode.Prev = node; // predecessor
        node.Next!.Prev = newNode;
        node.Next = newNode;

        Count++; _version++;
        return newNode;
}
```

Empty-list insertion is a special case handled once: create a node whose `Next` and `Prev` both reference itself and set `head = node`. Subsequent inserts reuse the uniform rewiring logic.

Removals in O(1)

Deletion given a node handle similarly splices the node out in constant time:

Listing 33: Remove by node (simplified)

```
public void Remove(LinkedListNode<T> node)
```

```
{
        if (node.List != this) return;

        if (node.Next == node) // singleton
        head = null;
        else
        {
                node.Next!.Prev = node.Prev;
                node.Prev!.Next = node.Next;
                if (head == node) head = node.Next;
        }

        node.Next = node.Prev = null; // help GC, mark as detached
        node.List = null;
        Count--; _version++;
}
```

Because nodes know their neighbors, no search is required to remove them. This is the principal advantage over singly linked lists where deletion needs the predecessor.

Enumeration Semantics and Safety

`LinkedList<T>` uses a struct enumerator that captures the list's `_version` at construction. Any structural modification (`Add`/`Remove`) bumps the version. If the version changes mid-iteration, subsequent `MoveNext()` triggers `InvalidOperationException`, preventing silent data-race bugs.

Listing 34: Enumerator outline

```
public struct Enumerator : IEnumerator<T>
{
        private readonly LinkedList<T> _list;
        private LinkedListNode<T>? _node; // current node; null => end
        private readonly int _version;
        private T _current;

        public bool MoveNext()
        {
                if (_version != _list._version) ThrowModified();
                if (_node is null) return false;
                _current = _node.Value;
                _node = _node.Next == _list.head ? null : _node.Next;
                return true;
        }

        public T Current => _current;
        // ... Reset/Dispose elided ...
}
```

Complexity and Practical Behavior

- **Given a node handle:** `AddAfter`, `AddBefore`, `Remove` are $O(1)$.
- **Find/Contains by value:** `O(n)` traversal (pointer chasing; cache-unfriendly).
- **Deque ops:** `AddFirst`/`AddLast`/`RemoveFirst`/`RemoveLast` are $O(1)$.
- **No indexing:** random access is `O(n)`; use `List<T>` for index-heavy use.
- **Memory layout:** each node is a separate object (object header + two references + value), increasing per-element overhead and GC pressure relative to contiguous arrays/lists.

- **No boxing for value types:** the generic `LinkedListNode<T>` stores `T` directly; .NET's reified generics avoid boxing for `T` = value type.

When `LinkedList<T>` Shines

- You frequently insert/delete using **existing node handles** (splicing, LRU *move-to-front*, editor gap buffers).
- You need **O(1) deque** operations at both ends without capacity concerns.
- You combine it with a **Dictionary<K,LinkedListNode<T»** for $O(1)$ lookups + $O(1)$ splices (e.g., caches).

Summary

`LinkedList<T>` trades cache locality and indexing for constant-time structural edits when a node is known. By storing head/tail implicitly via a circular doubly linked shape and validating nodes through back-pointers, it offers a robust, allocation-driven list ideal for splicing and deque patterns—complementary to the contiguous, indexable `List<T>`.

4.4 Comparison: List vs LinkedList

Arrays and array-backed lists (`List<T>`) emphasize *contiguity* and $O(1)$ indexing, while pointer-based lists (`LinkedList<T>`) emphasize *constant-time splices* given a node handle. The right choice depends on access patterns, mutation style, and memory behavior.

At a Glance

Dimension	`List<T>` (array-backed)	`LinkedList<T>` (doubly, circular)	Implication
Memory layout	Contiguous array of `T`	Scattered heap nodes (Prev/Next + Value)	Cache locality vs pointer chasing
Indexing	$O(1)$ random access	$O(n)$ (no indexer)	Use list for index-heavy algorithms
Iteration	Sequential, very cache-friendly	Pointer chasing, cache-unfriendly	Loops over `List<T>` are typically much faster
Append front/back	Back: $O(1)$ amortized; Front: $O(n)$	Both ends $O(1)$ (AddFirst/AddLast)	Deque patterns favor linked list
Insert/delete middle (by position)	$O(n)$ (shift tail block)	$O(n)$ to find position	Both are linear if you don't have a node handle
Insert/delete (with handle)	Not applicable (no stable node handle)	$O(1)$ via node rewiring	Splicing with saved handles favors linked list
Enumerator	Allocation-free struct; invalidated on modify	Struct; invalidated on modify	Safe iteration; mid-iteration edits throw
Per-element overhead	Minimal (just the value)	High (object header + 2 refs + value)	More GC pressure for linked list
Resizing behavior	Geometric growth; occasional copy	None (node-by-node allocation)	Lists benefit from pre-sizing; linked lists don't resize
Best for	Dense data, searching by index, scans	Frequent splices, deque ops, LRU caches	Choose by dominant operation

Table 4: `List<T>` vs `LinkedList<T>`: operational and memory tradeoffs.

Cache Locality and Iteration Patterns

List<T>. Elements are contiguous; iterating in index order aligns perfectly with CPU cache lines and prefetchers. Tight loops (`for`/`foreach`) typically approach the throughput of raw arrays:

```
var xs = new List<float>(n);
// ... fill ...
float sum = 0;
for (int i = 0; i < xs.Count; i++) sum += xs[i]; // sequential, cache-friendly
```

LinkedList<T>. Each step follows a pointer to an arbitrary address; hardware prefetching is ineffective, causing frequent cache misses. Traversals are notably slower despite both being $O(n)$ in theory:

```
var ll = new LinkedList<float>();
// ... fill ...
float sum = 0;
for (var node = ll.First; node is not null; node = node.Next) sum += node.Value; //
    pointer chasing
```

Random Access vs. Splicing

- **Random access / index-based algorithms** (binary search, partitioning, SIMD-able scans): prefer `List<T>`.
- **Splice-heavy workflows** (move-to-front, concatenating sublists, deque semantics): prefer `LinkedList<T>` with saved node handles.
- **Membership tests**: both are $O(n)$ by value; if you need $O(1)$ lookup, pair either with `Dictionary<K,V>` or `HashSet<T>`.

Mutation Costs and Stability

List<T>. Inserting/removing in the middle shifts a contiguous tail block ($O(n)$). Indexes to later elements change; iterators are invalidated (versioning).

LinkedList<T>. Given a node handle, insertion/removal is $O(1)$ and does not move other nodes. Saved node references remain valid unless explicitly removed. However, *finding* a node by value is still $O(n)$.

Memory and GC Effects

`List<T>` holds values tightly packed in one array object (excellent density). `LinkedList<T>` allocates one object per element (node header + two references + value), increasing memory footprint and GC activity. This overhead can dominate for small `T` (e.g., `int`), making `LinkedList<int>` comparatively expensive.

Choosing Between `List<T>` and `LinkedList<T>`

- Choose **`List<T>`** when:
  - You need fast random access or heavy iteration.
  - Your workload is append-heavy with occasional inserts/removes.
  - You can *pre-size* via `Capacity` to avoid resizes.
- Choose **`LinkedList<T>`** when:
  - You perform frequent *splices* with known node handles.
  - You require $O(1)$ deque operations at both ends without shifting.
  - You combine with a hash map for $O(1)$ locate + $O(1)$ splice (e.g., LRU cache).
- If you primarily test membership or need key-based access, consider **`HashSet<T>`** or **`Dictionary<K,V>`** instead of either list.

Summary

`List<T>` exploits contiguous memory for superior iteration and random access; mid-array edits are costly due to shifts. `LinkedList<T>` sacrifices cache locality to gain $O(1)$ local edits and deque operations via explicit node handles. Pick based on the dominant operations and whether you can trade contiguity for pointer-based flexibility.

5 Stacks and Queues

5.1 Conceptual Overview

Stacks and queues are foundational *access disciplines* for collections: a stack follows **LIFO** (Last-In, First-Out) order, while a queue follows **FIFO** (First-In, First-Out) order. These policies constrain where insertion/removal occurs and thus shape algorithmic usage and performance.

Stack (LIFO)

- **Core operations:** `Push(x)` adds to the *top*; `Pop()` removes and returns the top; `Peek()` reads the top without removing it.
- **Behavior:** the most recently added item is served first.
- **Complexity (array-backed):** `Push`/`Pop`/`Peek` are $O(1)$ amortized; iteration is $O(n)$.
- **Examples:**
  - *Undo stack* in editors: each action is pushed; `Undo` pops to revert.
  - *Expression evaluation / parsing:* operators and operands managed via operator/operand stacks.
  - *DFS* (depth-first search): frontier handled as a stack.

Queue (FIFO)

- **Core operations:** `Enqueue(x)` inserts at the *back*; `Dequeue()` removes and returns the *front*; `Peek()` reads the front without removing it.
- **Behavior:** the earliest enqueued item is served first.
- **Complexity (ring-buffer / linked):** `Enqueue`/`Dequeue`/`Peek` are $O(1)$; iteration is $O(n)$.
- **Examples:**
  - *Print queue*: jobs are processed in arrival order.
  - *BFS* (breadth-first search): frontier handled as a queue.
  - *Task scheduling / producer-consumer*: threads enqueue work; workers dequeue.

C# at a Glance

- `Stack<T>` is typically array-backed with geometric growth (amortized $O(1)$ `Push`/`Pop`).
- `Queue<T>` uses a circular buffer (ring) over an array for $O(1)$ `Enqueue`/`Dequeue`.
- Concurrent variants: `ConcurrentStack<T>`, `ConcurrentQueue<T>` support multi-threaded producers/consumers.

Tiny Examples (C#)

Listing 35: Undo stack (LIFO) concept

```
var undo = new Stack<string>();
undo.Push("typed␣A");
undo.Push("typed␣B");
Console.WriteLine(undo.Pop()); // "typed B"
Console.WriteLine(undo.Pop()); // "typed A"
```

Listing 36: Print queue (FIFO) concept

```
var jobs = new Queue<string>();
jobs.Enqueue("job-1");
jobs.Enqueue("job-2");
Console.WriteLine(jobs.Dequeue()); // "job-1"
Console.WriteLine(jobs.Dequeue()); // "job-2"
```

When to Use Which

- Choose a **stack** when the most recent item should be handled first (backtracking, nested scopes, undo).
- Choose a **queue** when fairness and arrival order matter (task dispatch, BFS, I/O buffering).

Operational Summary

Structure	Policy	Primary Ops	Typical Cost
Stack	LIFO	Push, Pop, Peek	$O(1)$ amortized
Queue	FIFO	Enqueue, Dequeue, Peek	$O(1)$

Implementation tip (C#): prefer array-backed stacks and circular-buffer queues for cache-friendly $O(1)$ operations. Linked variants are useful when you must splice or maintain handles but are less cache-efficient.

5.2 Stack<T> and Queue<T> Implementations

Both `Stack<T>` and `Queue<T>` in .NET are array-backed collections designed for constant-time operations on their respective ends. The stack uses a simple *top index* over a contiguous array; the queue uses a *circular buffer* (ring) over a contiguous array with head/tail pointers. Growth follows the same geometric strategy discussed for `List<T>`.

Stack<T>: Array with Top Index

Listing 37: Stack<T> basic implementation sketch

```
public class Stack<T> {
       private T[] _array = new T[4];
       private int _count = 0;
       public void Push(T item) => _array[_count++] = item;
       public T Pop() => _array[--_count];
}
```

The minimal sketch shows the core idea (top at `_count`). A production-ready version adds capacity checks, empty checks, and optional `Clear` zeroing (to release references for GC).

Listing 38: Array-backed stack with geometric growth

```
public class StackEx<T>
{
       private T[] _items;
       private int _count;
       private int _version;

       public StackEx(int capacity = 4) => _items = new T[Math.Max(1, capacity)];
       public int Count => _count;

       public void Push(T item)
       {
```

```
        if (_count == _items.Length) Grow(_count + 1);
        _items[_count++] = item;
        _version++;
    }

    public T Pop()
    {
        if (_count == 0) throw new InvalidOperationException("Empty stack");
        var idx = --_count;
        T value = _items[idx];
        _items[idx] = default!; // clear reference for GC
        _version++;
        return value;
    }

    public T Peek()
    {
        if (_count == 0) throw new InvalidOperationException("Empty stack");
        return _items[_count - 1];
    }

    private void Grow(int min)
    {
        int newCap = _items.Length * 2;
        if (newCap < min) newCap = min;
        Array.Resize(ref _items, newCap);
    }

    public void Clear()
    {
        Array.Clear(_items, 0, _count); // zero references
        _count = 0; _version++;
    }
}
```

Costs. `Push`/`Pop`/`Peek` are $O(1)$; growth triggers occasional $O(n)$ copies but appends are amortized $O(1)$. Contiguity makes iteration cache-friendly.

Queue<T>: Circular Buffer over an Array

A queue keeps two indices:

- **head**: index of the next element to dequeue,
- **tail**: index to write the next enqueued element,

and wraps both around `capacity` via modulo arithmetic. When full, it grows by allocating a larger array and copying elements in logical order starting at `head`.

Listing 39: Array-based circular queue

```
public class RingQueue<T>
{
    private T[] _items;
    private int _head; // next to Dequeue
    private int _tail; // next slot to Enqueue
    private int _count;
    private int _version;

    public RingQueue(int capacity = 4) => _items = new T[Math.Max(1, capacity)];
    public int Count => _count;
```

```
        public int Capacity => _items.Length;
        public bool IsEmpty => _count == 0;

        public void Enqueue(T item)
        {
            if (_count == _items.Length) Grow(_count + 1);
            _items[_tail] = item;
            _tail = (_tail + 1) % _items.Length;
            _count++; _version++;
        }

        public T Dequeue()
        {
            if (_count == 0) throw new InvalidOperationException("Empty queue");
            T item = _items[_head];
            _items[_head] = default!; // clear reference for GC
            _head = (_head + 1) % _items.Length;
            _count--; _version++;
            return item;
        }

        public T Peek()
        {
            if (_count == 0) throw new InvalidOperationException("Empty queue");
            return _items[_head];
        }

        private void Grow(int min)
        {
            int newCap = Math.Max(_items.Length * 2, min);
            var newArr = new T[newCap];
            // copy in logical order [head, head+count)
            if (_count > 0)
            {
                int right = Math.Min(_items.Length - _head, _count);
                Array.Copy(_items, _head, newArr, 0, right);
                Array.Copy(_items, 0, newArr, right, _count - right);
            }
            _items = newArr;
            _head = 0;
            _tail = _count;
        }

        public void Clear()
        {
            if (_count > 0)
            {
                if (_head < _tail) Array.Clear(_items, _head, _count);
                else { Array.Clear(_items, _head, _items.Length - _head);
                    Array.Clear(_items, 0, _tail); }
            }
            _head = _tail = _count = 0; _version++;
        }
    }
```

Costs. `Enqueue`/`Dequeue`/`Peek` are $O(1)$; grow is $O(n)$ but infrequent (amortized $O(1)$). The ring avoids shifting elements and preserves cache-friendly access for sequential consumption.

Implementation Notes and Pitfalls

- **Bounds/emptiness:** throw on `Pop`/`Dequeue` when empty. In performance code, you can expose `TryPop`/`TryDequeue`.
- **GC hygiene:** clear reference slots after removal so objects become collectible.
- **Versioning:** if exposing `IEnumerable<T>`, capture a `_version` to detect concurrent modification (like `List<T>`).
- **Capacity hints:** allow constructor capacity to reduce early resizes; consider `TrimExcess()` for long-lived queues that shrink.
- **Contiguity:** both structures are array-backed → excellent spatial locality in tight loops relative to linked variants.

Stack vs Queue (Array-Backed) — Operational Summary

Structure	Primary Ops	Cost	Notes
Stack<T>	Push / Pop / Peek	$O(1)$ amortized	Top at `_count`; geometric growth
Queue<T>	Enqueue / Dequeue / Peek	$O(1)$ amortized	Circular buffer; head/tail wrap

Use array-backed designs for cache-friendly, branch-light hot paths. Fall back to linked deques only when you need stable node handles or frequent splicing.

5.3 Deques and Ring Buffers

A **ring buffer** (circular buffer) stores elements in an array and treats the ends as adjacent by wrapping indices modulo `capacity`. A **deque** (double-ended queue) extends this with $O(1)$ `PushFront`/`PushBack` and `PopFront`/`PopBack`. These designs avoid bulk shifts, making them ideal for producer/consumer queues, sliding windows, and scheduler run queues.

Indexing and Wrap-Around

Maintain two indices:

- **head**: index of the current front element,
- **tail**: index *one past* the back element.

The number of elements is `count` = (`tail` − `head` + `capacity`) mod `capacity`. To advance an index `i` by one: `i` = (`i` + 1) mod `capacity`. To move backwards: `i` = (`i` − 1 + `capacity`) mod `capacity`.

Fixed-Capacity Ring Buffer (Overwrite-on-Full)

Listing 40: Fixed-size ring buffer (overwrites oldest when full)

```
public class RingBuffer<T>
{
        private readonly T[] _buf;
        private int _head; // index of oldest
        private int _tail; // index one past newest
        private int _count;

        public RingBuffer(int capacity)
        {
                if (capacity <= 0) throw new
                    ArgumentOutOfRangeException(nameof(capacity));
```

```
        _buf = new T[capacity];
    }

    public int Count => _count;
    public int Capacity => _buf.Length;
    public bool IsFull => _count == _buf.Length;
    public bool IsEmpty => _count == 0;

    public void PushBack(T item) // enqueue newest; overwrite oldest if full
    {
        _buf[_tail] = item;
        _tail = (_tail + 1) % _buf.Length;
        if (_count == _buf.Length)
        {
            _head = (_head + 1) % _buf.Length; // drop oldest
        }
        else
        {
            _count++;
        }
    }

    public bool TryPopFront(out T? value)
    {
        if (_count == 0) { value = default; return false; }
        value = _buf[_head];
        _buf[_head] = default!;
        _head = (_head + 1) % _buf.Length;
        _count--;
        return true;
    }
}
```

This pattern is common in telemetry and streaming: constant $O(1)$ time, fixed memory, no reallocations.

Resizable Deque<T> (Double-Ended)

Listing 41: Array-backed deque with $O(1)$ amortized end operations

```
public class Deque<T>
{
    private T[] _items;
    private int _head; // index of front
    private int _tail; // one past back
    private int _count;
    private int _version;

    public Deque(int capacity = 8) => _items = new T[Math.Max(2, capacity)];
    public int Count => _count;
    public bool IsEmpty => _count == 0;
    public int Capacity => _items.Length;

    public void PushFront(T item)
    {
        if (_count == _items.Length) Grow(_count + 1);
        _head = (_head - 1 + _items.Length) % _items.Length;
        _items[_head] = item;
        _count++; _version++;
    }
```

```
        public void PushBack(T item)
        {
                if (_count == _items.Length) Grow(_count + 1);
                _items[_tail] = item;
                _tail = (_tail + 1) % _items.Length;
                _count++; _version++;
        }

        public T PopFront()
        {
                if (_count == 0) throw new InvalidOperationException("Empty deque");
                T v = _items[_head];
                _items[_head] = default!;
                _head = (_head + 1) % _items.Length;
                _count--; _version++;
                return v;
        }

        public T PopBack()
        {
                if (_count == 0) throw new InvalidOperationException("Empty deque");
                _tail = (_tail - 1 + _items.Length) % _items.Length;
                T v = _items[_tail];
                _items[_tail] = default!;
                _count--; _version++;
                return v;
        }

        public T PeekFront()
        {
                if (_count == 0) throw new InvalidOperationException("Empty deque");
                return _items[_head];
        }
        public T PeekBack()
        {
                if (_count == 0) throw new InvalidOperationException("Empty deque");
                int last = (_tail - 1 + _items.Length) % _items.Length;
                return _items[last];
        }

        private void Grow(int min)
        {
                int newCap = Math.Max(_items.Length * 2, Math.Max(2, min));
                var arr = new T[newCap];
                // copy logical sequence [head, head+count)
                if (_count > 0)
                {
                        int right = Math.Min(_items.Length - _head, _count);
                        Array.Copy(_items, _head, arr, 0, right);
                        Array.Copy(_items, 0, arr, right, _count - right);
                }
                _items = arr;
                _head = 0;
                _tail = _count;
        }
}
```

Notes.

- **Wrapping arithmetic:** Using `(i + 1) % capacity` is clear. For power-of-two capacities, use a bitmask (`i = (i + 1) & (capacity - 1)`) to avoid division.
- **Amortized growth:** Doubling capacity keeps end operations $O(1)$ on average; growth performs a single linear copy in logical order.
- **GC hygiene:** Clear vacated slots to let referenced objects be collected.
- **Iteration:** Logical order is $[0..\texttt{Count} - 1]$ mapped via `(head + k) % capacity`.

Deque vs. Queue vs. Linked Deque

- **Array deque:** $O(1)$ amortized `Push/Pop` at both ends; best cache locality; single array copy on growth.
- **Queue<T> (ring):** $O(1)$ amortized `Enqueue/Dequeue` at fixed ends; simpler when only FIFO is needed.
- **Linked deque:** $O(1)$ worst-case at ends without resizing; poorer locality; higher per-element overhead; good for splicing.

Example Usage

Listing 42: Sliding window with a deque

```
var dq = new Deque<int>();
dq.PushBack(1); dq.PushBack(2); dq.PushBack(3); // [1,2,3]
dq.PushFront(0); // [0,1,2,3]
Console.WriteLine(dq.PopBack()); // 3
Console.WriteLine(dq.PopFront()); // 0
```

5.4 Applications

Stacks and queues appear pervasively in real systems. Below are common, practical uses with compact C# sketches and notes on correctness and complexity.

Depth-First Search (DFS) with a Stack (LIFO)

Iterative DFS replaces recursion with an explicit `Stack<T>` to avoid stack overflows on deep graphs.

Listing 43: DFS using Stack<T>

```
void DfsIterative(Dictionary<int, List<int>> g, int start)
{
	var seen = new HashSet<int>();
	var st = new Stack<int>();
	st.Push(start);
	while (st.Count > 0)
	{
		int u = st.Pop();
		if (!seen.Add(u)) continue;
		// visit u ...
		foreach (var v in g[u]) st.Push(v); // LIFO => deep before wide
	}
}
```

Notes. Using a stack makes traversal order naturally depth-first; iteration is $O(V + E)$.

Breadth-First Search (BFS) with a Queue (FIFO)

BFS discovers vertices in increasing distance (edges assumed unit weight), ideal for shortest paths in unweighted graphs.

Listing 44: BFS using Queue<T> (returns distance)

```
int[] Bfs(Dictionary<int, List<int>> g, int s, int n)
{
	var dist = Enumerable.Repeat(int.MaxValue, n).ToArray();
	var q = new Queue<int>();
	dist[s] = 0; q.Enqueue(s);

	while (q.Count > 0)
	{
		int u = q.Dequeue();
		foreach (var v in g[u])
		if (dist[v] == int.MaxValue)
		{
			dist[v] = dist[u] + 1;
			q.Enqueue(v); // FIFO => wavefront expands layer by layer
		}
	}
	return dist;
}
```

Notes. Complexity $O(V + E)$; the queue models the expanding frontier.

Parsing and Expression Evaluation (Stacks)

Parsing often uses one or more stacks (operands/operators) to implement precedence and associativity (Dijkstra's shunting yard).

Listing 45: Tiny infix evaluator (digits + +-*), shunting-yard style

```
int Eval(string s)
{
	int Prec(char op) => op == '+' || op == '-' ? 1 : 2;

	var vals = new Stack<int>();
	var ops = new Stack<char>();

	void Apply()
	{
		int b = vals.Pop(), a = vals.Pop(); char op = ops.Pop();
		vals.Push(op switch { '+' => a + b, '-' => a - b, '*' => a * b, _ =>
		    throw new() });
	}

	for (int i = 0; i < s.Length; )
	{
		if (char.IsWhiteSpace(s[i])) { i++; continue; }
		if (char.IsDigit(s[i]))
		{
			int v = 0;
			while (i < s.Length && char.IsDigit(s[i])) v = 10 * v + (s[i++]
			    - '0');
			vals.Push(v);
		}
		else // operator
		{
```

```
                        char op = s[i++];
                        while (ops.Count > 0 && Prec(ops.Peek()) >= Prec(op)) Apply();
                        ops.Push(op);
                    }
                }
                while (ops.Count > 0) Apply();
                return vals.Pop();
            }
```

Notes. Balanced parentheses checking is a classic single-stack variant; tokenization benefits from `ReadOnlySpan<char>` to avoid allocations.

Undo / Redo (Two Stacks)

Editor-style undo/redo can be modeled by two stacks: `undo` and `redo`.

Listing 46: Undo/Redo with two stacks of commands

```
interface ICommand { void Do(); void Undo(); }

class History
{
        private readonly Stack<ICommand> _undo = new();
        private readonly Stack<ICommand> _redo = new();

        public void Execute(ICommand cmd) { cmd.Do(); _undo.Push(cmd); _redo.Clear();
            }
        public bool Undo() { if (_undo.Count == 0) return false; var c = _undo.Pop();
            c.Undo(); _redo.Push(c); return true; }
        public bool Redo() { if (_redo.Count == 0) return false; var c = _redo.Pop();
            c.Do(); _undo.Push(c); return true; }
}
```

Notes. Each operation is $O(1)$; memory grows with history length.

Producer–Consumer (Batched Queues)

For multi-threaded pipelines, a concurrent queue decouples producers and consumers. `BlockingCollection<T>` wraps a `ConcurrentQueue<T>` with blocking semantics.

Listing 47: Producer–consumer with BlockingCollection<T>

```
var bc = new BlockingCollection<byte[]>(boundedCapacity: 1024);

// Producer
Task.Run(() => {
        foreach (var chunk in ReadChunks(/* ... */))
        bc.Add(chunk); // blocks when full (back-pressure)
        bc.CompleteAdding();
});

// Consumer(s)
var workers = Enumerable.Range(0, Environment.ProcessorCount).Select(_ =>
    Task.Run(() => {
        foreach (var chunk in bc.GetConsumingEnumerable())
        Process(chunk); // work in parallel
})).ToArray();

Task.WaitAll(workers);
```

Notes. Bounded capacity provides natural back-pressure; chunk size should balance cache locality with syscall overhead. For single-threaded contexts, a simple ring-buffer queue (previous section) is often faster.

Design Tips for Stack/Queue Applications

- **Prefer array-backed** stacks/queues for cache locality and predictable $O(1)$ end-ops.
- **DFS vs BFS:** choose based on traversal order (deep-first vs level-order) and memory profile.
- **Parsing:** minimize allocations; use `Span<T>` / `ReadOnlySpan<char>` in hot paths.
- **Undo/Redo:** model reversible actions; clear redo stack on new action.
- **Producer–consumer:** batch work; use bounded queues to apply back-pressure; size batches to fit cache lines.

Array-Backed `Stack<T>` vs. `List<T>` (When You Only Need LIFO)

Bottom line. If your workload is purely LIFO, `Stack<T>` mainly buys you *clarity and guardrails* with essentially the same performance characteristics as a `List<T>` used at its end (`Add`, `RemoveAt(Count-1)`).

What `Stack<T>` gives you over `List<T>`

- **Stronger intent & fewer foot-guns:** API exposes only `Push/Pop/Peek`. You can't accidentally do costly mid-list ops like `Insert(0,x)` or `RemoveAt(i)`.
- **Tight, branch-light paths:** Specialized for top edits; easy for the JIT to inline.
- **Enumeration semantics:** Iterates from *top to bottom* (natural for a stack).
- **Ergonomic helpers:** `TryPop`/`TryPeek`, `Clear` (zeroes refs), `TrimExcess`, etc.
- **Concurrency-ready cousin:** `ConcurrentStack<T>` with matching semantics.

What is (nearly) the same

- **Asymptotics:** `Push` ≈ `Add` is $O(1)$ amortized; `Pop` ≈ `RemoveAt(Count-1)` is $O(1)$.
- **Resizing:** Both use geometric growth; occasional $O(n)$ copy on expand.
- **Memory & locality:** Contiguous arrays in both → excellent cache behavior.
- **Overhead:** Similar per-instance fields (array + count + version).

Guidance

- If the structure is *logically a stack*, use `Stack<T>` for intent, safety, and clean iteration.
- If you *also* need non-LIFO edits, use `List<T>`—but keep hot-path operations at the end to retain $O(1)$ behavior.
- In microbenchmarks, differences are typically in the noise; let **readability and correctness** decide.

6 Dictionaries and Hash Tables

6.1 Hashing Fundamentals

Hash tables provide (on average) constant-time *membership*, *insertion*, and *deletion* by mapping a key to an integer index via a **hash function**. In C#, `Dictionary<TKey,TValue>` and `HashSet<T>` rely on two contracts:

1. **Equality** via `IEqualityComparer<T>.Equals(x,y)` (or `Equals` on `T`).
2. **Hashing** via `IEqualityComparer<T>.GetHashCode(x)` (or `GetHashCode` on `T`).

Keys that are equal *must* yield the same hash code. A hash *distributes* many keys across a fixed number of buckets/slots. When two keys map to the same bucket, we have a **collision**.

Hash Functions (Conceptual)

A hash function computes an integer digest from a key:

$$h : \texttt{Key} \to \{0, \ldots, m-1\}, \quad \text{typically } h(k) = \big(\text{mix}(k)\big) \bmod m.$$

Good hash functions are:

- **Deterministic** (k always maps the same way),
- **Uniform** (keys spread evenly across buckets),
- **Fast** (few instructions on the hot path),
- **Stable** w.r.t. equality (if $x = y$ then $h(x) = h(y)$).

C# practice. Built-ins like `string.GetHashCode()` (with randomized seeding per process) and `HashCode.Combine(a,b,…)` provide high-quality mixing. For composite keys, combine fields:

Listing 48: Custom comparer for a composite key

```
public readonly record struct Point(int X, int Y);

public sealed class PointComparer : IEqualityComparer<Point>
{
       public bool Equals(Point a, Point b) => a.X == b.X && a.Y == b.Y;
       public int GetHashCode(Point p) => HashCode.Combine(p.X, p.Y);
}
```

Load Factor and Resizing

The **load factor** $\alpha = \frac{n}{m}$ (elements per bucket/slot) controls performance. As α grows, collisions rise. Tables *resize* (allocate a bigger array and rehash) to keep α in a target range, preserving $O(1)$ average operations.

Collision Resolution

Two classic families handle collisions:

1) Chaining (Separate Buckets). Each bucket holds a small structure (traditionally a list) of entries with the same hash index. Insertions prepend or append to the bucket; lookups linearly scan the bucket.

- **Pros:** Natural deletes (no special markers), less sensitive to clustering, easy to grow table size without complex probe logic.
- **Cons:** Extra pointer indirection; potential cache misses if buckets are linked lists (many runtimes use arrays/flat nodes to improve locality).
- **Costs:** Average $O(1)$ if α bounded; worst $O(n)$ if everything collides.

2) Open Addressing (In-Table Probing). All entries live *inside* the slot array. On collision, try other slots following a probe sequence:

- **Linear probing:** $i, i+1, i+2, \ldots$ (wrap at m). Simple and cache-friendly; suffers from *primary clustering*.
- **Quadratic probing:** $i, i+1^2, i+2^2, \ldots \pmod m$. Reduces clustering; still contiguous-ish.
- **Double hashing:** $i, i+d, i+2d, \ldots$, with d a second hash not divisible by m. Best distribution; slightly more compute.

Deletes in open addressing require *tombstones* to keep probe chains intact, or a backward-shift compaction step. Rebuilds (rehash) can clear tombstones.

Open Addressing vs. Chaining (Practical Tradeoffs)

Collision Strategies at a Glance

	Chaining	**Open Addressing**	**Implications**
Storage	Buckets of nodes	Single flat slot array	OA is very cache-friendly
Insert/Find avg.	$O(1)$ (bounded α)	$O(1)$ (low α)	Both hinge on load factor
Delete	Simple unlink	Tombstones or compaction	Chaining is simpler to delete
Clustering	Less sensitive	Needs good probing	Double-hash helps
Memory	Pointers/headers	Dense, minimal overhead	OA denser; chaining more flexible
Resize	Rehash buckets	Rehash slots	Both linear in n
Worst-case	$O(n)$ bucket scan	$O(n)$ probe scan	Adversarial keys hurt both

From Hash to Index (Masking)

Implementations map a 32/64-bit hash to a table slot:

$$\texttt{index} = \text{mix}(h) \;\&\; (\texttt{capacity} - 1) \quad \text{(power-of-two capacity)}$$

or $\texttt{index} = \text{mix}(h) \bmod \texttt{capacity}$ for arbitrary sizes. Power-of-two capacities allow fast masking and nicely support linear/quadratic probing.

Correctness Contracts (C#)

- If `Equals(a,b)` is `true`, then `GetHashCode(a) == GetHashCode(b)` *must* hold.
- **Do not mutate** fields participating in equality/hashing while an element is in the table.
- Pick the right comparer: e.g., `StringComparer.OrdinalIgnoreCase` for case-insensitive sets/maps.

Micro-Examples (Probe vs Chain)

Listing 49: Linear probing (illustrative; not production)

```
int FindSlot((int hash, bool used)[] slots, int hash, int capacity)
{
        int i = hash & (capacity - 1);
        while (slots[i].used && slots[i].hash != hash)
        i = (i + 1) & (capacity - 1); // linear probe + wrap
        return i;
}
```

Listing 50: Chaining bucket (illustrative)

```
sealed class Entry<K,V> { public K Key; public V Val; public Entry<K,V>? Next; /*
    ... */ }

int bucket = Hash(key) & (buckets.Length - 1);
for (var e = buckets[bucket]; e is not null; e = e.Next)
if (cmp.Equals(e.Key, key)) return e.Val;
// else prepend new Entry at buckets[bucket]
```

Key Takeaways

- Hash tables are fast when the hash function is uniform and the load factor is controlled.
- Collisions are inevitable; choose *how* to resolve them: chaining (simple deletes) or open addressing (dense, cache-friendly).
- In C#, equality and hashing are defined by the comparer; ensure **consistency** and avoid mutating keys in-place.

6.2 C# `Dictionary<TKey,TValue>` Internals

Dictionary is a *separate-chaining* hash table engineered for speed and density. It uses two parallel structures:

- **Buckets:** an `int[]` where each slot holds the index (or index+1) of the first entry in that bucket, or a sentinel for empty.
- **Entries:** an array of structs; each element stores `hashCode`, `next` (linked-list index), `key`, `value`.

This design keeps the per-bucket metadata compact while storing keys/values densely in one array, improving locality over pointer-heavy node lists.

Core Layout (schematic)

```
struct Entry<TKey,TValue>
{
       public int hashCode; // cached, >= 0 when used
       public int next; // index of next entry in chain (-1 if end)
       public TKey key;
       public TValue value;
}

class Dictionary<TKey,TValue>
{
       private int[] _buckets; // bucket -> first entry index (or -1)
       private Entry<TKey,TValue>[] _entries; // storage for items
       private int _count; // used entries (incl. free slots)
       private int _freeList; // head of free list (-1 if none)
       private int _freeCount; // number of free slots
       private IEqualityComparer<TKey> _comparer; // Equals + GetHashCode
       private int _version; // iterator invalidation
}
```

Indexing into Buckets

Given a 32/64-bit hash, the bucket index is computed by masking or modulo:

$$\texttt{bucket} = \text{mix}(\texttt{hash}) \;\&\; (\texttt{capacity} - 1) \quad \text{(power-of-two capacity)}$$

or `hash` mod `capacity` for arbitrary sizes.

Insertion (pseudocode)

The algorithm caches the key's hash, walks the chain in that bucket to find duplicates, and either *updates in place* (key exists) or *inserts a new entry* at the head of the chain. It reuses slots from a free list when possible and grows/rehashes when the load factor passes a threshold.

Listing 51: Simplified insertion in Dictionary<TKey,TValue>

```
void Insert(TKey key, TValue value, bool overwrite)
{
	// 1) Initialize tables if first insert
	if (_buckets == null) Initialize(capacity: 4);

	// 2) Compute hash and bucket
	int hash = _comparer.GetHashCode(key) & 0x7FFFFFFF;
	int bucket = hash & (_buckets.Length - 1); // or % for non power-of-two

	// 3) Walk chain to check for existing key
	for (int i = _buckets[bucket]; i >= 0; i = _entries[i].next)
	{
		if (_entries[i].hashCode == hash &&
		_comparer.Equals(_entries[i].key, key))
		{
			if (overwrite) { _entries[i].value = value; _version++; }
			else throw new InvalidOperationException("Duplicate key");
			return;
		}
	}

	// 4) Need a new slot: from free list or at end; grow if full
	int index;
	if (_freeCount > 0)
	{
		index = _freeList;
		_freeList = _entries[index].next; // pop free list
		_freeCount--;
	}
	else
	{
		if (_count == _entries.Length) { Resize(); bucket = hash &
		    (_buckets.Length - 1); }
		index = _count++;
	}

	// 5) Link new entry at head of bucket chain
	_entries[index].hashCode = hash;
	_entries[index].next = _buckets[bucket]; // previous head (or -1)
	_entries[index].key = key;
	_entries[index].value = value;
	_buckets[bucket] = index; // new head

	_version++;
}
```

Your snippet matches step (3) above—the chain walk within a bucket:

```
int bucket = key.GetHashCode() % _buckets.Length;
for (int i = _buckets[bucket]; i >= 0; i = _entries[i].next) {
	if (_entries[i].key.Equals(key)) { _entries[i].value = value; return; }
}
```

Deletion and Free List

Removing an entry splices it out of its bucket chain and pushes its slot onto the *free list*:

- set `entries[i].hashCode = -1` and `entries[i].next = _freeList`,
- `_freeList = i`, `_freeCount++`.

Future insertions prefer free slots before growing.

Resizing and Rehashing

When the load factor exceeds a target (to keep chains short), the dictionary allocates larger arrays and *rehashes* all live entries into the new bucket array (recomputing bucket index from the cached hash). Resize cost is linear in the number of elements and happens infrequently, so average operation cost remains $O(1)$.

Notes and Practicalities

- **Comparer defines identity:** keys are "equal" per the dictionary's `IEqualityComparer<TKey>` (e.g., case-insensitive strings).
- **Do not mutate keys in-place:** changes that affect hash/equality break lookup.
- **Enumeration safety:** `_version` invalidates iterators on mutation (like `List<T>`).
- **HashSet<T> shares the layout:** conceptually the same structure but stores only keys (no values).

6.3 HashSet<T> and Equality Comparers

`HashSet<T>` is a hash-table-backed collection that stores *unique elements* ("values," not key/value pairs). It provides $O(1)$ average `Add`, `Remove`, and `Contains`, and supports set algebra (`UnionWith`, `IntersectWith`, `ExceptWith`, `SymmetricExceptWith`). Uniqueness is defined entirely by the set's `IEqualityComparer<T>`.

Relationship to Dictionary<TKey,TValue>

Conceptually, `HashSet<T>` is like a `Dictionary<TKey,TValue>` that stores only keys. Internally both use a bucket array plus an entries/slots array and the same hash/equality contracts:

- **Dictionary** enforces unique *keys* and maps each to a value.
- **HashSet** enforces unique *elements* (no associated values).

Both rely on a comparer to compute hashes and test equality; both resize/rehash to keep load factor healthy and maintain $O(1)$ average-time operations.

Uniqueness Enforcement

`Add(x)` inserts `x` if no equal element exists; otherwise it returns `false`. `Contains(x)` tests membership; `Remove(x)` deletes it if present. Equality is taken from the set's comparer:

Listing 52: Uniqueness is comparer-defined

```
var ci = StringComparer.OrdinalIgnoreCase;
var hs = new HashSet<string>(ci);
Console.WriteLine(hs.Add("ALPHA")); // True
Console.WriteLine(hs.Add("alpha")); // False (equal under comparer)
Console.WriteLine(hs.Contains("Alpha")); // True
```

Equality and Hashing in C#

Correctness and performance hinge on the equality/hash contract:

- If `Equals(a,b)` is `true`, then `GetHashCode(a) == GetHashCode(b)` *must* hold.
- The converse is not required (different items may share a hash), but good hashing reduces collisions.
- Do not mutate fields that affect equality/hashing while an element is in the set; lookups may fail (*aliasing bug*).

Built-in behavior. Most primitive/value types and `string` supply high-quality `Equals`/`GetHashCode`. `ValueTuple<...>` also implements structural equality and hashing, so composite keys often "just work":

Listing 53: Composite elements without a custom comparer

```
var cells = new HashSet<(int Row, int Col)>();
cells.Add((1, 2));
Console.WriteLine(cells.Contains((1, 2))); // True
```

Custom Comparers (When and How)

Use a custom `IEqualityComparer<T>` when domain equality differs from the default (case-folded strings, culture-aware comparison, key-projection, etc.).

Listing 54: Custom comparer for a domain-specific key

```
public readonly record struct Point(int X, int Y);

public sealed class PointComparer : IEqualityComparer<Point>
{
       public bool Equals(Point a, Point b) => a.X == b.X && a.Y == b.Y;
       public int GetHashCode(Point p) => HashCode.Combine(p.X, p.Y);
}

var set = new HashSet<Point>(new PointComparer());
set.Add(new Point(10, 20));
Console.WriteLine(set.Contains(new Point(10, 20))); // True
```

Projection-based comparer. Sometimes "uniqueness by a key selector" is handy; a lightweight comparer can delegate:

Listing 55: Comparer by projection (key selector)

```
public sealed class KeyComparer<T, TKey> : IEqualityComparer<T>
{
       private readonly Func<T, TKey> _key;
       private readonly IEqualityComparer<TKey> _cmp;
       public KeyComparer(Func<T, TKey> key, IEqualityComparer<TKey>? cmp = null)
       { _key = key; _cmp = cmp ?? EqualityComparer<TKey>.Default; }
       public bool Equals(T a, T b) => _cmp.Equals(_key(a), _key(b));
       public int GetHashCode(T x) => _cmp.GetHashCode(_key(x));
}

// Usage: unique by Email (case-insensitive)
var users = new HashSet<User>(new KeyComparer<User,string>(u => u.Email,
    StringComparer.OrdinalIgnoreCase));
```

Operations and Complexity (Practical)

- `Add`/`Remove`/`Contains`: $O(1)$ average, $O(n)$ worst under heavy collision.
- `UnionWith`/`ExceptWith`/`IntersectWith`: linear in the size of the input(s), dominated by membership checks.
- `EnsureCapacity`/`TrimExcess`: manage allocations to reduce rehash events.

Implementation Notes & Gotchas

- **Comparer choice = semantics.** Pick `StringComparer.OrdinalIgnoreCase` for case-insensitive strings; avoid culture-sensitive comparers on hot paths unless required.
- **Mutability hazards.** If `T` is a reference type, avoid mutating fields used in equality/hashing while the element is in the set. For value types, be careful with mutable structs.
- **Locality vs pointers.** A hash set is faster than a `List<T>` for membership but less cache-linear for scans; don't use it for ordered/indexed workflows.
- **Thread safety.** `HashSet<T>` is not thread-safe for concurrent writes; use external synchronization or `ConcurrentDictionary<T,bool>` as a stand-in when needed.

HashSet<T> vs. Dictionary<TKey,TValue> (mental model)

	HashSet<T>	Dictionary<TKey,TValue>	Notes
Uniqueness	Elements	Keys	Enforced by comparer
Primary op	Membership	Key→Value map	Both $O(1)$ avg
Comparer	`IEqualityComparer<T>`	`IEqualityComparer<TKey>`	Same contract
Values	None	Arbitrary	HashSet = "dictionary without values"
Set ops	Built-in	N/A	Use for dedupe/algebra

6.4 Performance Notes

Hash-table performance is governed by three interacting factors: the **load factor** (α), the **cost of rehashing** when the table grows, and the **behavior under collisions**. This section gives a practical model for reasoning about each, with C#-specific tips.

Load Factor α (elements per bucket/slot)

Let n be the number of elements and m the number of buckets (or slots).

$$\alpha = \frac{n}{m}.$$

Keeping α in a target band (commonly around 0.5–0.8) limits average chain lengths / probe lengths and preserves $O(1)$ behavior.

Chaining intuition (Dictionary/HashSet in .NET). With good hashing, the expected bucket length is α. A successful lookup examines about $1 + \alpha/2$ entries on average; an unsuccessful lookup examines α entries.

Open addressing intuition (for comparison). For linear probing with uniform hashing:

$$\mathbb{E}[\text{probes (successful)}] \approx \tfrac{1}{2}\left(1 + \tfrac{1}{1-\alpha}\right), \qquad \mathbb{E}[\text{probes (unsuccessful)}] \approx \tfrac{1}{2}\left(1 + \tfrac{1}{(1-\alpha)^2}\right).$$

As $\alpha \to 1$, probe counts rise sharply—why tables resize before becoming too full.

Rule of Thumb: Choosing Capacity

If you expect n items, aim for capacity $m \approx \lceil n/0.75 \rceil$. In C#, call `EnsureCapacity(n)` (or constructor with capacity) to avoid multiple rehashes.

Rehashing Cost (Growth Events)

When α exceeds a threshold, the table allocates larger arrays and *rehashes* all live entries:

- **Time cost:** $O(n)$ to copy and relink/reindex every entry once.
- **Amortization:** Insertions remain $O(1)$ *on average* because growth happens geometrically (capacity roughly doubles or otherwise grows multiplicatively).
- **GC consideration:** One large allocation (new buckets/entries) + linear copy; fewer, larger growth steps are usually better than many small steps.

Practical tips (C#).

- Use `new Dictionary<TKey,TValue>(capacity)` or `EnsureCapacity` when n is known.
- After large removals, `TrimExcess()` can reduce memory (rehash to a smaller table).
- Avoid frequent oscillations (grow–shrink–grow); reuse instances where possible.

Collision Behavior and Hash Quality

Collisions are inevitable: distinct keys can map to the same bucket index.

Good hashing & distribution. Uniform mixing spreads keys across buckets, keeping α predictive. In C#, prefer:

- Built-in `GetHashCode` for primitives and `string` (seeded per process).
- `HashCode.Combine(…)` for composite keys.
- A domain-appropriate `IEqualityComparer<T>` (e.g., `StringComparer.OrdinalIgnoreCase`).

Skew and clustering. Poor hash functions or adversarial inputs inflate certain buckets:

- **Chaining:** Overfull buckets cause longer linear scans in those chains.
- **Open addressing:** Clusters form; probe sequences get long and correlated.

Mitigations include better mixing, randomized seeding (already applied to strings), and resizing (which re-buckets with a different mask).

Mutation hazards. Never mutate fields that participate in equality/hash while a key is stored; subsequent `Contains`/`Remove` may fail because the item appears in the "wrong" bucket.

Throughput vs Latency (What to Expect)

- With healthy α and decent hashing, `Add`/`Contains`/`Remove` are effectively a few memory accesses: average $O(1)$ with tight constants.
- Resize spikes appear as occasional $O(n)$ pauses; pre-sizing smooths them out.
- Iteration is $O(n)$ and cache-friendly over the entries array, but *membership* (random lookups) is the intended hot path; don't use a dictionary as an ordered list.

Checklist (C#)

- Pick the right comparer; ensure `Equals` and `GetHashCode` are consistent.
- Pre-size with `EnsureCapacity(n)` when possible; avoid rehash storms.
- Keep α reasonable (defaults do this); pathological slowdowns usually trace to bad hashing, adversarial inputs, or key mutation.

7 Tuples, Records, and Struct Aggregates

7.1 ValueTuple vs Tuple

C# exposes two tuple families:

- `System.ValueTuple<...>` — a *value type* (struct). Used by the C# tuple literal syntax `(a, b, c)`, deconstruction, and pattern matching.
- `System.Tuple<...>` — a *reference type* (class). Pre–C# 7 era tuples; still available but rarely preferred today.

Allocation and Memory Behavior

- **ValueTuple (struct):** typically *no heap allocation* when used as a local or field of another struct; it lives inline. It will be allocated on the heap only if captured by a heap object (e.g., field of a class, element of a heap array) or when *boxed* (e.g., cast to `object` or an interface).
- **Tuple (class):** *always* a heap object; creating a tuple always allocates and adds GC pressure.

Listing 56: Allocation differences (illustrative)

```
// ValueTuple: no allocation as a local
(int x, int y) p = (1, 2);

// Tuple<T1,T2>: always an object allocation
var t = Tuple.Create(1, 2);
```

Syntax and Usability

- **Literals and deconstruction:** Only **ValueTuple** integrates with tuple literals and deconstruction.

Listing 57: Tuple literals, element names, and deconstruction

```
(string first, string last) name = ("Ada", "Lovelace");
Console.WriteLine(name.first); // "Ada" (compile-time name)
var (f, l) = name; // deconstruction
```

Element names. Named tuple elements are a *compile-time* feature on top of `ValueTuple`; at runtime the fields are `Item1`, `Item2`, ... (names may be carried in metadata, but APIs typically see `ItemN`).

Equality Semantics

- **ValueTuple:** value semantics by default. Implements `IEquatable<...>`; `Equals` and `==` compare component-wise with `EqualityComparer<T>.Default`.
- **Tuple (class):** reference type, but `Tuple<...>` *overrides* `Equals` and hash code to perform *structural* equality over components (not reference identity). Still, it's an allocated object.

Listing 58: Equality: structural for both, but one allocates

```
var a = (X: 1, Y: 2); // ValueTuple<int,int>
var b = (X: 1, Y: 2);
Console.WriteLine(a == b); // True (value semantics)

var c = Tuple.Create(1, 2);
var d = Tuple.Create(1, 2);
Console.WriteLine(c.Equals(d)); // True (structural), but both allocated
```

Interoperability and Limits

- **Arity:** Both families support up to 8 components by nesting a `Rest` item; in practice, prefer small, named records for readability beyond a few fields.
- **APIs and reflection:** Many APIs and serializers treat `ValueTuple` as plain structs with `ItemN` fields. Element names are not first-class at runtime in most reflection views.
- **Older frameworks:** `System.ValueTuple` required a NuGet package on legacy target frameworks (< .NET 4.7); modern .NETs include it in the BCL.

When to Use Which (Practical Guidance)

Choosing Between `ValueTuple` and `Tuple`

	ValueTuple<...>	Tuple<...>	Notes
Type kind	**struct** (value)	class (reference)	ValueTuple avoids allocations in many cases
Syntax	**Literals, deconstruction, patterns**	No language sugar	C# tuple syntax maps to ValueTuple
Equality	**Value-based** (`IEquatable`)	Structural via overrides	Both compare component-wise
GC pressure	**Low** (no alloc unless boxed/embedded)	**High** (always alloc)	Hot paths prefer ValueTuple
Element names	Compile-time aliases	`Item1`, `Item2`, ...	Names not first-class at runtime
Recommendation	**Prefer**	Legacy interop only	Use records for long-lived domain objects

Rule of thumb. Use **ValueTuple** for transient, glue-level returns (multiple values, deconstruction, pattern matching) in hot paths. For long-lived, self-documenting domain data, favor **record types** (next section) which give named fields, immutability options, and clear semantics without the positional ambiguity of tuples.

7.2 Records and Immutability

Records are C# types designed for *data-centric* programming: they provide *structural equality* (compare by contents) and ergonomic syntax for *immutable* carriers. By default, `record` defines a *reference type* (class) with value-based equality; `record struct` defines a *value type* with the same semantics. Records are ideal as keys in dictionaries/sets, for message passing, and for clear, self-documenting data models.

Positional Records (Reference Type)

A positional record generates a constructor, deconstruction, `with`-expression support, and value-based `Equals`/`GetHashCode` automatically.

Listing 59: Reference record with structural equality

```
public readonly record Person(string First, string Last);

var a = new Person("Ada", "Lovelace");
var b = new Person("Ada", "Lovelace");

Console.WriteLine(a == b); // True (value-based equality)
var (f, l) = a; // deconstruction
var a2 = a with { Last = "King" }; // non-destructive mutation (copy with change)
```

Immutability. The parameters of a positional `record` become `init`-only properties by default; callers can set them only in object initializers or the primary constructor. Marking the record `readonly` discourages accidental field mutation in methods.

Property-Based Records

You can also write records with named properties (use `init` to keep immutability):

Listing 60: Property-based record with init-only setters

```
public record Order
{
       public required int Id { get; init; }
       public required string Sku { get; init; }
       public decimal Price { get; init; }
}

var o1 = new Order { Id = 1, Sku = "ABC-123", Price = 9.99m };
var o2 = o1 with { Price = 12.50m }; // copy with one property changed
```

Record Structs

For stack-friendly, allocation-free carriers (e.g., many short-lived values), use `record struct`. It's a value type with the same "compare by contents" semantics.

Listing 61: Value-type version: record struct

```
public readonly record struct Point(int X, int Y);

var p = new Point(3, 4);
var q = new Point(3, 4);
Console.WriteLine(p == q); // True (component-wise)
```

Equality, Hashing, and Collections

Records synthesize `Equals`, `GetHashCode`, and `==`/`!=` to compare component-wise using `EqualityComparer<T>.Default`. This makes them natural keys:

Listing 62: Records as dictionary keys and set elements

```
var seen = new HashSet<Person>();
seen.Add(new Person("Ada", "Lovelace"));
Console.WriteLine(seen.Contains(new Person("Ada", "Lovelace"))); // True
```

Note on mutability of members. Record equality is only as immutable as its members. If a record holds a mutable reference (e.g., `List<T>`), changing the list mutates the record's logical state. Prefer immutable member types (e.g., arrays replaced by `ImmutableArray<T>` or `IReadOnlyList<T>` with discipline).

Pattern Matching and Deconstruction

Records integrate with pattern matching for concise, intention-revealing code:

Listing 63: Property and positional patterns with records

```
static string Describe(Person p) =>
p switch
{
       { First: "Ada" } => "Pioneer",
       var (f, l) when l.Length > 5 => "Long␣surname",
       _ => "Someone␣else"
};
```

Performance and Design Tips

- **Reference records** allocate like any class; use for long-lived identities and interop.
- **Record structs** avoid allocations in many scenarios; great for hot paths and small aggregates.
- **Non-destructive mutation** via `with` copies fields; it's a *shallow* copy—nested references are shared.
- Use **required** + `init` to enforce initialization and preserve immutability.
- Avoid putting heavy logic inside records; they are best as *data carriers*.

Records at a Glance (C# 9+)

	record (class)	record struct	Guidance
Type kind	Reference	Value	Pick by lifetime / allocation goals
Equality	Structural (by contents)	Structural (by contents)	Good for keys and dedupe
Immutability	`init` by default	Can be `readonly`	Favor immutable members
Syntax	Positional / property-based	Positional / property-based	`with`, deconstruction, patterns
Copying	`with` (shallow)	`with` (shallow)	Deep-copy manually if needed

Rule of thumb. Use **records** for small, identity-by-data types where you want clear equality and immutable defaults. Choose **record struct** for high-throughput, allocation-sensitive paths; choose **record (class)** for domain objects that naturally live on the heap and are passed around by reference.

7.3 Structs vs Classes

In C#, **structs** are *value types* and **classes** are *reference types*. This distinction drives *allocation* behavior, *copy semantics*, and *equality*—and has concrete performance implications in hot paths.

Allocation: Stack vs Heap (and the Real Story)

- **Classes (reference types):** instances are *heap* objects; variables hold *references* (pointers) to those objects.
- **Structs (value types):** instances live *inline* where they are declared:
  - as locals/temporaries: typically on the *stack* (or in registers),
  - as fields of a class: *inline inside that object's heap memory*,
  - as elements of an array of `T`: *inline inside the array buffer*.

Note. "Structs are on the stack" is an oversimplification: they can be on the heap when contained in a heap object/array, or when boxed. The key is *inline storage* vs *indirection*, not a specific region.

Copy Semantics and Passing

- **Class variables** copy the *reference*. Two variables can refer to the same object.
- **Struct variables** copy the *entire value*. Assignment or passing *by value* duplicates all fields.
- Avoid large struct copies in hot paths by passing `in` (readonly by-ref), `ref`, or `out`.

Listing 64: Copy vs reference semantics

```
struct Point { public int X, Y; }

var a = new Point { X = 1, Y = 2 };
var b = a; // copies the fields
b.X = 42;
// a.X == 1, b.X == 42

class Node { public int V; }
var n1 = new Node { V = 7 };
var n2 = n1; // copies the reference
n2.V = 99;
// n1.V == 99 (same object)
```

Equality Defaults

- **Classes:** default `==` compares *reference identity*; override `Equals`/`GetHashCode` (or use `record`) for structural equality.
- **Structs:** default `Equals` is field-wise; you can implement `IEquatable<T>` for faster/explicit semantics.

Boxing and Its Costs (Structs)

A value-type instance *boxes* when converted to `object` or an interface type:

- Allocates a heap object and copies the value into it,
- Adds indirection and GC pressure,
- Unboxing copies the value back.

Avoid boxing in hot paths (e.g., prefer `IEqualityComparer<T>` over non-generic `IComparer`).

Listing 65: Boxing pitfalls

```
struct S { public int X; }
object o = 123; // boxes an int
IComparable c = 123; // boxes again to interface
```

Mutability, `readonly` Structs, and Defensive Copies

Mutable structs can lead to surprising copies, especially through properties/indexers:

- Accessing a struct *returned by value* (e.g., from a property) gives a *copy*. Mutating the copy does not affect the original.
- Mark performance-critical value types as `readonly struct` to prevent hidden defensive copies when passed by `in`.

Listing 66: Hidden copies with mutable structs

```
struct Vec { public int X; public void Inc() => X++; }
class Bag { public Vec V; }

var bag = new Bag { V = new Vec { X = 1 } };
// bag.V returns a copy; incrementing it modifies the copy, not bag.V
bag.V.Inc(); // warning: modifies a temporary (no effect on bag.V)
```

`ref struct` and Stack-Only Types

Types like `Span<T>` are `ref structs`:

- **stack-only**: cannot be boxed, cannot be fields of classes, cannot be captured by lambdas,
- enable safe, allocation-free slices over memory (great for hot paths).

Performance Heuristics

- **Small, POD-like aggregates** ($\leq$ 16–32 bytes, few fields) work well as `structs`—compact, inlined, cache-friendly.
- **Large structs** increase copy costs; prefer classes or pass by `in`/`ref`.
- Avoid putting **mutable** structs in collections exposed via properties; callers may mutate copies unintentionally.
- Use `record struct` for small immutable carriers with structural equality.

Structs vs Classes (at a glance)

	Struct (value type)	**Class (reference type)**	**Implications**
Storage	Inline (stack/inline in objs/arrays)	Heap object + reference	Structs avoid indirection
Copy	Copies *data*	Copies *reference*	Large structs can be pricey
Equality (default)	Field-wise	Reference identity	Records give structural for classes
Boxing	Possible (costly)	N/A	Avoid object/interface casts for structs
Immutability	Use `readonly struct`	Use `record`/`init` props	Safer semantics for keys
By-ref ops	`in`/`ref`/`out` to avoid copies	Usually not needed	Prefer `in` for read-only large structs
Special	`ref struct` (stack-only)	Inheritable, polymorphic	Choose per design/interop needs

Rule of thumb. Use **structs** for small, often-embedded, immutable (or carefully controlled) aggregates on hot paths; pass by `in` to avoid copies when large. Use **classes** for shared, long-lived objects with identity, polymorphism, or when mutation through references is desired.

8 Heaps and Priority Queues

8.1 Binary Heap Basics

A **binary heap** is a complete binary tree (the *shape property*) that also obeys an ordering constraint (the *heap property*). Heaps are the canonical building block for *priority queues*: `Insert`/`Push` and `ExtractMin`/`Pop` run in $O(\log n)$, while `Peek` is $O(1)$.

Shape Property (Completeness)

- The tree is *complete*: all levels are full except possibly the last, which is filled *left to right*.
- This compact shape lets us store the heap in a single contiguous array with no pointers.

Heap Property (Ordering)

- **Min-heap:** each node $\leq$ its children $\Rightarrow$ the minimum is at the root.
- **Max-heap:** each node $\geq$ its children $\Rightarrow$ the maximum is at the root.

Heaps are *partially ordered*: siblings are not ordered relative to each other. Duplicate keys are allowed; "stability" (relative order of equals) is *not* guaranteed.

Array Mapping (0-based indices)

For an array `a` of length `n` storing the heap level-by-level:

$$\begin{aligned} \texttt{parent}(i) &= \left\lfloor \frac{i-1}{2} \right\rfloor \quad (i > 0), \\ \texttt{left}(i) &= 2i + 1, \\ \texttt{right}(i) &= 2i + 2. \end{aligned}$$

A node at index `i` has children within bounds iff `left(i) < n` (and `right(i) < n`).

(For 1-based arrays: $\texttt{parent}(i) = \lfloor i/2 \rfloor$, $\texttt{left}(i) = 2i$, $\texttt{right}(i) = 2i + 1$*.)*

Core Operations (Min-Heap)

- **Insert / Push(x):** place x at the end (`a[n]=x`), then *sift up*: while x violates the heap property with its parent, swap with parent. Cost $O(\log n)$.
- **Peek / Min():** return `a[0]`. Cost $O(1)$.
- **ExtractMin / Pop():** save `a[0]`, move `a[n-1]` to `a[0]`, shrink n, then *sift down*: swap with the smaller child while the heap property is violated. Cost $O(\log n)$.
- **Build-heap:** given arbitrary array, *heapify* bottom-up by sifting down all internal nodes (from $\lfloor n/2 \rfloor - 1$ to 0). Cost $O(n)$.

Tiny Example (0-based)

Index i	0	1	2	3
Value $a[i]$	2	4	5	9

Here, `a[0]=2` is the root (min). Children of $i = 0$ are 1 and 2 (values 4 and 5). Child of $i = 1$ is 3 (value 9). The array already satisfies the min-heap property.

Why Heaps Map Well to Arrays

- **Shape guarantees density:** completeness ⇒ no holes in the array, excellent cache locality.
- **Pointer-free navigation:** parent/child indices are computed arithmetically (no extra memory, branch-light).
- **Predictable costs:** $O(1)$ peek; $O(\log n)$ push/pop; $O(n)$ build-heap; duplicates allowed; not stable.

8.2 Manual Implementation

Below is a compact, allocation-aware *min-heap* implemented over a contiguous array (backed by `List<T>`). It uses zero-based indexing with the standard parent/child formulas. The heap property is restored by **sift-up** (after inserts) and **sift-down** (after removals).

Listing 67: Binary heap core (min-heap with sift-up/sift-down)

```
public sealed class BinaryHeap<T>
{
	private readonly List<T> _data;
	private readonly IComparer<T> _cmp;

	public BinaryHeap(int capacity = 0, IComparer<T>? comparer = null)
	{
		_data = capacity > 0 ? new List<T>(capacity) : new List<T>();
		_cmp = comparer ?? Comparer<T>.Default;
	}

	public int Count => _data.Count;
	public bool IsEmpty => _data.Count == 0;

	public T Peek()
	{
		if (_data.Count == 0) throw new InvalidOperationException("Empty␣
		    heap");
		return _data[0];
	}

	// ---- Push: append at end, then sift-up to restore heap property
	public void Push(T item)
	{
		_data.Add(item);
		SiftUp(_data.Count - 1);
	}

	// ---- Pop (ExtractMin): swap root with last, remove last, then sift-down
	public T Pop()
	{
		if (_data.Count == 0) throw new InvalidOperationException("Empty␣
		    heap");
		T root = _data[0];
		int last = _data.Count - 1;

		_data[0] = _data[last];
		_data.RemoveAt(last);
		if (_data.Count > 0) SiftDown(0);

		return root;
```

```
        }

        // ---- Sift-up: while parent > child, swap up
        private void SiftUp(int i)
        {
                while (i > 0)
                {
                        int parent = (i - 1) / 2;
                        if (_cmp.Compare(_data[parent], _data[i]) <= 0) break;
                        Swap(parent, i);
                        i = parent;
                }
        }

        // ---- Sift-down: push element at i down to its correct spot
        private void SiftDown(int i)
        {
                int n = _data.Count;
                while (true)
                {
                        int left = 2 * i + 1;
                        int right = left + 1;
                        if (left >= n) break;

                        // pick smaller child
                        int m = (right < n && _cmp.Compare(_data[right], _data[left]) <
                            0) ? right : left;

                        if (_cmp.Compare(_data[i], _data[m]) <= 0) break;
                        Swap(i, m);
                        i = m;
                }
        }

        private void Swap(int a, int b)
        {
                if (a == b) return;
                ( _data[a], _data[b] ) = ( _data[b], _data[a] );
        }
}
```

Notes.

- **Comparator:** Passing a custom `IComparer<T>` flips ordering (e.g., use `Comparer<T>.Create((x,y)=>y.CompareTo(x))` for a max-heap).
- **Costs:** `Push` and `Pop` are $O(\log n)$; `Peek` is $O(1)$.
- **Stability:** Heaps are not stable; equal priorities may reorder.
- **Initialization:** For bulk input, a bottom-up heapify runs in $O(n)$ (add a constructor from `IEnumerable<T>` that copies into `_data` then sifts down from $\lfloor n/2 \rfloor - 1$ to 0).

Listing 68: Binary heap push operation

```
public void Push(T item) {
        _data.Add(item);
        SiftUp(_data.Count - 1);
}
```

Listing 69: Binary heap pop (extract-min) operation

```
public T Pop() {
       if (_data.Count == 0) throw new InvalidOperationException("Empty␣heap");
       T root = _data[0];
       int last = _data.Count - 1;
       _data[0] = _data[last];
       _data.RemoveAt(last);
       if (_data.Count > 0) SiftDown(0);
       return root;
}
```

Index Math (0-based array)

`parent`$(i) = \lfloor (i-1)/2 \rfloor$, `left`$(i) = 2i+1$, `right`$(i) = 2i+2$. Pick the smaller child during sift-down for a min-heap (larger for a max-heap).

8.3 .NET `PriorityQueue<TElement,TPriority>`

The built-in `PriorityQueue<TElement,TPriority>` (in `System.Collections.Generic`) is a **binary heap**-backed *min-priority* queue. It stores pairs (`element`, `priority`) and orders by `TPriority` using an `IComparer<TPriority>` (default is `Comparer<TPriority>.Default`). Core operations are $O(\log n)$; `Peek` is $O(1)$.

Internal Structure (Conceptual)

- **Heap shape:** complete binary tree stored in a contiguous array (excellent locality).
- **Nodes:** each slot holds `element` and `priority`; comparisons use the priority only.
- **Ordering:** min-heap by default; pass a custom comparer to invert or customize ordering.
- **Stability:** *not stable*—equal priorities may be dequeued in any order.

API at a Glance

- `Enqueue(element, priority)`: insert; $O(\log n)$ (sift-up).
- `TryDequeue(out element, out priority)` / `Dequeue()`: remove-min; $O(\log n)$ (sift-down).
- `Peek()` / `TryPeek(...)`: view min; $O(1)$.
- `Count`, `Clear()`, optional capacity-aware `EnsureCapacity(int)` (on recent frameworks).
- `UnorderedItems`: enumerate heap contents in *heap order* (not sorted).

Basic Usage

Listing 70: PriorityQueue basics

```
var pq = new PriorityQueue<string, int>(); // lower int = higher priority
pq.Enqueue("parse", 2);
pq.Enqueue("compile", 1);
pq.Enqueue("link", 3);

Console.WriteLine(pq.Peek()); // "compile"
while (pq.TryDequeue(out var task, out _))
Console.WriteLine(task); // compile, parse, link
```

Custom ordering. Use a comparer to make a max-heap or domain-specific order:

Listing 71: Max-heap via custom comparer

```
var maxCmp = Comparer<int>.Create((a,b) => b.CompareTo(a)); // larger int first
var pqMax = new PriorityQueue<string, int>(maxCmp);
```

Scheduling Use Case (Time- or Score-Based)

Listing 72: Scheduler sketch: run earliest-deadline first

```
var sched = new PriorityQueue<(Action job, DateTime due), DateTime>();

void Schedule(Action job, DateTime due) => sched.Enqueue((job, due), due);

while (true)
{
	if (sched.TryPeek(out var next, out var due) && due <= DateTime.UtcNow)
	{
		sched.Dequeue().job(); // run the job with earliest due time
	}
	// ... sleep or wait ...
}
```

Dijkstra's Algorithm (Without Decrease-Key)

`PriorityQueue` does *not* expose a decrease-key operation. The standard pattern is to **re-enqueue** a node with a better priority and *skip stale entries* when they surface.

Listing 73: Dijkstra with re-enqueue and stale-skip

```
IReadOnlyList<(int to, int w)>[] G = /* adjacency list */;
int n = G.Length;
var dist = Enumerable.Repeat(int.MaxValue, n).ToArray();

var pq = new PriorityQueue<(int node, int d), int>(); // priority = distance
int s = 0;
dist[s] = 0;
pq.Enqueue((s, 0), 0);

while (pq.TryDequeue(out var item, out var prio))
{
	int u = item.node;
	if (prio != dist[u]) continue; // stale entry (skip)

	foreach (var (v, w) in G[u])
	{
		int nd = dist[u] + w;
		if (nd < dist[v])
		{
			dist[v] = nd;
			pq.Enqueue((v, nd), nd); // re-enqueue with improved priority
		}
	}
}
```

This achieves the classic $O((V+E)\log V)$ complexity: each improvement triggers a heap insert; stale entries are filtered with the `dist` array guard.

Complexity and Practical Notes

- **Costs:** `Enqueue`/`Dequeue` are $O(\log n)$; `Peek` is $O(1)$; iteration via `UnorderedItems` is $O(n)$ but not sorted.
- **Capacity planning:** if you expect n items, pre-size (constructor with capacity or `EnsureCapacity`) to reduce resizes.
- **Comparer cost:** priorities should be cheap to compare (primitives, tuples, or precomputed scores).
- **Stability:** if you need stable ordering among equals, bake a tie-breaker into the priority (e.g., a monotonically increasing sequence number).
- **Dequeue-many:** for bulk draining into sorted order, repeatedly `Dequeue`—the heap outputs ascending priorities.

When to Reach for `PriorityQueue`

- **Schedulers** (timers, rate-limiters, EDF): priority = deadline or next-run time.
- **Graph algorithms** (Dijkstra/A*): priority = tentative distance or $g + h$.
- **Top-k / streaming** selection: maintain a size-k max-heap of best items.
- **Anytime you need "next best" repeatedly** with predictable $O(\log n)$ updates.

9 Trees (Applied View)

9.1 Binary Search Trees and Balancing

A **binary search tree (BST)** stores keys in nodes so that for any node with key k, every key in its left subtree is $< k$ and every key in its right subtree is $> k$ (assuming a strict total order). This invariant enables **logarithmic-time** search and updates on *balanced* trees; in the worst case (a degenerate chain) operations degrade to $O(n)$.

BST Search and Insertion (Core Logic)

Search. Starting at the root, compare the target key with the current node: go left if the target is smaller, right if larger, stop if equal (*hit* or *miss*). Depth visited is the *height* h; cost is $O(h)$.

Listing 74: BST Node and Search (C#-style sketch)

```
public class Node<T>
{
	public T Key;
	public Node<T>? Left, Right;
	public Node(T key) { Key = key; }
}

public static Node<T>? Find<T>(Node<T>? root, T key, IComparer<T> cmp)
{
	for (var cur = root; cur is not null; )
	{
		int c = cmp.Compare(key, cur.Key);
		if (c == 0) return cur;
		cur = (c < 0) ? cur.Left : cur.Right;
	}
	return null;
}
```

Insertion. Search for the *null* position where the key belongs; attach a new node there. Without balancing, repeated inserts in sorted order produce height $h = n$ (worst case).

Listing 75: BST Insert (unbalanced)

```
public static Node<T> Insert<T>(Node<T>? root, T key, IComparer<T> cmp)
{
	if (root is null) return new Node<T>(key);
	Node<T> cur = root, parent = root;
	bool goLeft = false;

	while (cur is not null)
	{
		parent = cur;
		int c = cmp.Compare(key, cur.Key);
		if (c == 0) return root; // ignore duplicates (policy choice)
		if (c < 0) { goLeft = true; cur = cur.Left; }
		else { goLeft = false; cur = cur.Right; }
	}
	if (goLeft) parent.Left = new Node<T>(key);
	else parent.Right = new Node<T>(key);
	return root;
}
```

Self-Balancing Ideas (High Level)

Balanced BSTs maintain height $h = O(\log n)$ by performing **local rotations** after updates to restore a relaxed structural invariant. Two canonical families:

AVL Trees (height-balanced). Maintain a *balance factor* at each node (usually bf = height(left) − height(right) $\in \{-1, 0, 1\}$). After insert/delete, walk back up, update heights, and rotate when $|\text{bf}| > 1$.

- **Rotations:** single (LL, RR) and double (LR = left-then-right, RL = right-then-left).
- **Pros:** very tight balance ⇒ excellent query times.
- **Cons:** slightly more bookkeeping per update (heights and possibly two rotations).

Red–Black Trees (color-balanced). Each node has a *color* (red/black) and the tree satisfies: (1) root is black; (2) no two red nodes are adjacent; (3) every path from a node to descendant nulls has the same number of black nodes. Insert/delete fixups recolor and rotate to restore these invariants.

- **Pros:** simpler metadata (one color bit), good $O(\log n)$ bounds; standard in many libraries.
- **Cons:** slightly less tightly balanced than AVL (but practically excellent).

Rotations (Local Restructuring)

A rotation is a constant-time pointer change that preserves in-order sequence.

Listing 76: Right rotation (LL fix) — schematic

```
static Node<T> RotateRight<T>(Node<T> y)
{
        var x = y.Left!; // assume not null when called
        var T2 = x.Right; // will become y.Left
        x.Right = y;
        y.Left = T2;
        return x; // new subtree root
}
```

A left rotation is symmetric. AVL/LR and RL cases use a *double rotation* (rotate child, then parent).

Costs and Guarantees

- **Unbalanced BST:** operations are $O(h)$ with h potentially $O(n)$.
- **Balanced (AVL/RB):** height $h = O(\log n)$; `search/insert/delete` are $O(\log n)$ worst-case.
- **Space:** one node per key plus small metadata (height or color).

Practical Guidance (C#)

- Use `SortedDictionary<TKey,TValue>` / `SortedSet<T>` for a production-grade, balanced map/set (red–black under the hood) with $O(\log n)$ ops and in-order iteration.
- Choose **AVL** if you implement your own and want faster lookups (tighter balance); choose **RB** for simpler updates and widely known behavior.
- Keep comparisons cheap: prefer cached keys or key selectors; avoid expensive culture-sensitive comparisons in hot paths.

BST Balancing in One Page

- **BST invariant:** in-order traversal yields sorted keys; search/insert traverse one path.
- **Why balance?** Prevent height blow-up from adversarial insert orders.
- **How to balance?** After updates, use *rotations* guided by a relaxed invariant:
  - AVL: keep per-node heights and ensure $|\text{bf}| \leq 1$.
  - Red–Black: maintain color rules; fix with recolor + rotations.
- **Result:** worst-case $O(\log n)$ for search/insert/delete; in-order iteration is sorted; memory overhead is modest (height or color per node).

9.2 SortedDictionary<TKey,TValue>

`SortedDictionary<TKey,TValue>` is a map that maintains its keys in *sorted order* according to an `IComparer<TKey>` (default: `Comparer<TKey>.Default`). Unlike `Dictionary` (hash table), `SortedDictionary` is backed by a **balanced binary search tree**—specifically a **red–black tree**—so `Add`, `Remove`, and `TryGetValue` run in $O(\log n)$ worst case, and in-order iteration yields keys in ascending order.

Why a Red–Black Tree? (under the hood)

A red–black (RB) tree is a BST with additional invariants (node colors and path-black-height rules) that keep the height $h = O(\log n)$. On insert/delete it performs *local rotations* and recoloring to restore balance. The result is predictable $O(\log n)$ bounds without the pointer chasing overhead of separate list nodes per bucket (as in chaining hash tables).

```
internal sealed class Node<TKey,TValue> {
       internal TKey Key;
       internal TValue Value;
       internal Node<TKey,TValue>? Left, Right, Parent;
       internal bool IsRed; // color bit
}
```

Insertion follows BST order (via `Comparer<TKey>`) then fixes colors/rotates. Deletion removes a node and fixes potential black-height violations with rotations/recoloring.

Key Properties and API
Simplified node sketch (conceptual).

- **Comparer-driven order:** pass a custom `IComparer<TKey>` (e.g., case-insensitive strings).
- **Sorted enumeration:** `foreach` visits pairs in ascending key order.
- **Ops & costs:** `Add`, `Remove`, `ContainsKey`, `TryGetValue` are $O(\log n)$; `this[key]` lookup/set is $O(\log n)$; enumeration is $O(n)$ in-order traversal.
- **Views:** `Keys` and `Values` expose dynamic views that reflect updates.

Basic Usage

Listing 77: Sorted map with custom ordering

```
var map = new SortedDictionary<string,int>(StringComparer.OrdinalIgnoreCase);
map["beta"] = 2;
```

```
map.Add("alpha", 1);
map["gamma"] = 3;

foreach (var (k,v) in map)
Console.WriteLine($"{k}␣->␣{v}"); // alpha -> 1, beta -> 2, gamma -> 3

if (map.TryGetValue("ALPHA", out var a))
Console.WriteLine(a); // 1 (case-insensitive comparer)
```

SortedDictionary vs Dictionary vs SortedList

	Dictionary<TKey,TValue>	**SortedDictionary<TKey,TValue>**	**SortedList<TKey,TValue>**
Backing	Hash table	Red–black tree	Sorted arrays (keys/values)
Lookup	$O(1)$ avg ($O(n)$ worst)	$O(\log n)$ worst	$O(\log n)$ via binary search
Insert/Delete	$O(1)$ avg (amort.)	$O(\log n)$	$O(n)$ (shifts)
Order	Unspecified	**Sorted**	**Sorted** + indexable
Memory	Dense slots	Node per entry (ptrs+color)	Dense arrays
Best for	Fast membership/map	Online updates + sorted iteration	Static-ish maps, index-based access

Practical Notes

- **Choose by workload:** need sorted traversal and many online inserts/deletes? Use `SortedDictionary`. Need fastest membership with no order? Use `Dictionary`. Need indexing and mostly-static data? `SortedList`.
- **Comparer cost:** keep `Comparer<TKey>` cheap; expensive comparisons can dominate $O(\log n)$ operations.
- **No range views built-in:** unlike `SortedSet<T>.GetViewBetween`, `SortedDictionary` lacks native range slices; emulate via iteration from a lower bound (or consider `SortedList` if frequent indexed range ops are needed).
- **Related type:** `SortedSet<T>` uses the same RB-tree strategy over keys only (no values).

Red–Black Internals: What to remember

- Balanced by *color invariants* + *rotations* $\Rightarrow$ height $O(\log n)$.
- Operations follow BST order (via `Comparer<TKey>`) with $O(\log n)$ search path.
- In-order traversal yields keys sorted ascending; memory per entry includes pointers and a color bit.

9.3 Practical Uses

Balanced, ordered containers (`SortedDictionary`, `SortedSet`, `SortedList`) shine when you need *ordering-aware* queries: ranges, predecessor/successor, and interval logic. Below are patterns you can drop into real systems.

Range Queries (Key $\in [L, H]$)

A frequent need is: "iterate only keys between L and H." `SortedDictionary` maintains order but has no direct indexing; `SortedList` adds indexable `Keys` enabling binary search for fast range starts.

Listing 78: Range view via binary search on SortedList.Keys

```
static IEnumerable<KeyValuePair<int,string>> Range(
SortedList<int,string> sl, int lo, int hi)
{
       // Left bound (first index >= lo)
       int i = sl.Keys.BinarySearch(lo);
       if (i < 0) i = ~i;
       for (; i < sl.Count && sl.Keys[i] <= hi; i++)
       yield return new KeyValuePair<int,string>(sl.Keys[i], sl.Values[i]);
}
```

Costs. Start-up is $O(\log n)$; output is $O(k)$ for k results. *When to use.* You do many range slices and can tolerate $O(n)$ inserts (arrays shift).

With `SortedDictionary<TKey,TValue>`. Without indexing, you can still do:

Listing 79: Simple (but linear) range over SortedDictionary

```
foreach (var kv in map)
{
       if (kv.Key < lo) continue;
       if (kv.Key > hi) break;
       // process kv
}
```

This is $O(n)$ in the worst case. If you need fast range starts on a tree, consider:

- Keep a parallel `SortedList<TKey,int>` (for index) or `SortedSet<TKey>` for views.
- Use a third-party B-tree/treap with explicit `LowerBound`/`UpperBound`.

Ordered Maps (Floor/Ceiling, Min/Max, Leaderboards)

`SortedDictionary` gives you in-order iteration and quick Min/Max via first/last element:

Listing 80: Min/Max and ordered iteration

```
var sd = new SortedDictionary<int,string>();
// ... fill ...
var min = sd.First(); // smallest key
var max = sd.Last(); // largest key
foreach (var (k,v) in sd) { /* ascending by k */ }
```

To emulate *floor/ceiling* (predecessor/successor) efficiently, pair with a `SortedSet<TKey>`:

Listing 81: Floor/Ceil with a view (SortedSet)

```
var keys = new SortedSet<int>(sd.Keys);
int x = 42;
var ceilView = keys.GetViewBetween(x, int.MaxValue);
int? ceil = ceilView.Count > 0 ? ceilView.Min : null;

var floorView = keys.GetViewBetween(int.MinValue, x);
int? floor = floorView.Count > 0 ? floorView.Max : null;
```

(`SortedSet` supports `GetViewBetween` with cheap *view* iteration.)

Interval Problems (Scheduling, "Meeting Rooms", Coverage)

Classic interval tasks include: detecting overlaps, finding max concurrent intervals, and maintaining a set of *merged* disjoint intervals.

Sweep-line (max overlap / rooms). Encode starts as $+1$, ends as -1 (end-exclusive). Accumulate in a `SortedDictionary<Time,int>` and sweep in order.

Listing 82: Sweep-line to compute max overlap

```
int MaxOverlap((int start, int end)[] intervals)
{
	var events = new SortedDictionary<int,int>();
	foreach (var (s,e) in intervals) {
		if (!events.TryAdd(s, +1)) events[s] += 1;
		if (!events.TryAdd(e, -1)) events[e] -= 1; // treat [s,e) to avoid
		    double count
	}
	int cur = 0, best = 0;
	foreach (var delta in events.Values) {
		cur += delta;
		if (cur > best) best = cur;
	}
	return best;
}
```

Use cases. Meeting rooms, concurrent jobs, bandwidth peaks.

Maintaining merged disjoint intervals. Store merged segments keyed by start; on insert $[L, R]$, merge any touching/overlapping neighbors. For fast neighbor discovery, `SortedList` yields predecessor via `BinarySearch`.

Listing 83: Insert-merge into disjoint intervals (SortedList)

```
class IntervalSet
{
	private readonly SortedList<int,int> _seg = new(); // start -> end (inclusive)

	public void Add(int L, int R)
	{
		if (L > R) (L, R) = (R, L);
		// find first segment with start >= L
		int i = _seg.Keys.BinarySearch(L);
		if (i < 0) i = ~i;

		// consider predecessor for overlap/touch
		int start = L, end = R;
		int pred = i - 1;
		if (pred >= 0 && _seg.Values[pred] + 1 >= L) { // overlap/touch
			start = _seg.Keys[pred];
			end = Math.Max(end, _seg.Values[pred]);
			_seg.RemoveAt(pred); i--;
		}

		// merge forward while next start <= end+1
		while (i < _seg.Count && _seg.Keys[i] <= end + 1) {
			end = Math.Max(end, _seg.Values[i]);
			_seg.RemoveAt(i);
		}
		_seg.Add(start, end);
	}

	public IEnumerable<(int L,int R)> Intervals()
	{
		for (int i = 0; i < _seg.Count; i++) yield return (_seg.Keys[i],
		    _seg.Values[i]);
```

```
            }
        }
```

Costs. Each `Add` is dominated by `SortedList` shifts: worst-case $O(n)$, but merges are linear in the number of overlapping neighbors only. *When to use.* Moderate n, many queries; for heavier online updates, consider a balanced tree with explicit predecessor/successor primitives.

Which Ordered Container for Which Job?

Task	**Best fit**	Why	Notes
Many range slices	`SortedList`	Indexable `Keys` + binary search	Inserts $O(n)$ (shifts)
Online ordered map	`SortedDictionary`	RB-tree $O(\log n)$ updates	No indexing; pair with `SortedSet` for views
Predecessor/successor	`SortedSet`	`GetViewBetween`/Min/Max	Store values in a side map if needed
Sweep-line events	`SortedDictionary`	Natural key-order accumulation	Perfect for overlap/rooms
Merged intervals	`SortedList`/RB-tree	Fast pred + local merges	Pick by update vs query mix

Rule of thumb. If you need *ordered* iteration with frequent updates, start with **SortedDictionary**. If you need *by-index* range slices, prefer **SortedList**. Use **SortedSet** for key-only order queries (views, floor/ceil), pairing with a dictionary when you also need values.

10 Graph and Composite Structures

10.1 Adjacency Representations

Graphs are commonly stored as either an **adjacency list** (neighbors per vertex) or an **adjacency matrix** (bit/weight per possible edge). Each has distinct time/space and API tradeoffs that show up immediately in algorithms and memory use.

Adjacency List (sparse-friendly)

Store, for each vertex u, the set/list of its outgoing neighbors. In C#, a practical model is a map `u` $\mapsto$ `List<v>`; for weighted graphs, store `List<(v, w)>`.

Listing 84: Adjacency list in C# (directed, unweighted)

```
var adj = new Dictionary<int, List<int>>();
void AddEdge(int u, int v) {
       if (!adj.TryGetValue(u, out var list)) adj[u] = list = new List<int>();
       list.Add(v);
}
```

Space. $\Theta(n+m)$ pointers/entries; excellent for sparse graphs ($m \ll n^2$). **Iteration.** Enumerating neighbors of u is $O(\deg(u))$. **Edge existence test.** `Contains(v)` is $O(\deg(u))$ with a list; use `HashSet<T>` per vertex for $O(1)$ expected membership (higher overhead). **Best for.** BFS/DFS, Dijkstra/A*, MSTs—algorithmic work scales with m.

Listing 85: Weighted lists (tuple or struct)

```
var wadj = new Dictionary<int, List<(int v, int w)>>();
// AddEdge(u, (v, w))...
```

Adjacency Matrix (dense-friendly)

Store an $n \times n$ table where cell (u, v) encodes edge presence/weight.

Listing 86: Adjacency matrix in C# (bool presence)

```
bool[,] M = new bool[n,n];
void AddEdge(int u, int v) => M[u,v] = true;
bool HasEdge(int u, int v) => M[u,v];
```

Space. $\Theta(n^2)$ regardless of m; prohibitive for large sparse graphs. **Edge existence test.** $O(1)$ direct access $\Rightarrow$ great for dense graphs or algorithms that probe many arbitrary pairs. **Iteration.** Scanning neighbors of u is $O(n)$ (row scan), even if $\deg(u)$ is small. **Weights.** Use `float[,]`, `double[,]`, or sentinel for "no edge."

Directed vs Undirected

For undirected graphs:

- Lists: insert both directions (u, v) and (v, u).
- Matrix: mirror entries `M[u,v]` = `M[v,u]`.

Complexity & Memory (at a glance)

Operation / Aspect	Adjacency List	Adjacency Matrix	Notes
Space	$\Theta(n+m)$	$\Theta(n^2)$	Lists win on sparse graphs
Add edge	$O(1)$ amortized	$O(1)$	List append vs cell write
Remove edge	$O(\deg(u))$ (list) / $O(1)$ (set)	$O(1)$	List removal cost depends on structure
HasEdge(u,v)	$O(\deg(u))$ or $O(1)$ with set	$O(1)$	Matrix excels at membership
Iterate neighbors(u)	$O(\deg(u))$	$O(n)$	Lists scale with actual degree
BFS/DFS runtime	$O(n+m)$	$O(n^2)$	Matrix forces full-row scans
Dense algorithm kernels	OK	**Great**	Many pair probes benefit from $O(1)$ access
Cache locality	Good within a list	Good for contiguous rows	Matrix rows are contiguous, but n^2 space

Choosing a Representation (Heuristics)

- **Use lists** when $m = O(n)$ or generally $m \ll n^2$; when neighbor iteration dominates; for pathfinding, connectivity, MSTs.
- **Use a matrix** when the graph is **dense** or the algorithm repeatedly tests membership across many pairs (e.g., DP on graphs, transitive-closure-like kernels, bitset tricks).
- **Hybrid:** lists + per-vertex `HashSet<T>` for fast `HasEdge(u,v)`; compressed sparse row (CSR) for cache-friendly bulk traversal; bitset matrices for memory-efficient dense/unweighted graphs.

Rule of Thumb

If you will mostly *traverse* edges, favor an **adjacency list**. If you will mostly *query* arbitrary edge existence on a dense graph, favor an **adjacency matrix**. Pre-size lists (`Capacity`) for known degrees; consider CSR or `ReadOnlySpan<T>` views to maximize locality.

10.2 Implementation Example

Below is a compact adjacency-list graph with directed edges, plus **BFS** (level-order) and **DFS** (preorder) traversals. The representation uses `Dictionary<int,List<int>>` for simplicity; you can switch `List<int>` to `HashSet<int>` if you need fast `HasEdge(u,v)` checks.

Listing 87: Adjacency list in C#

```
var graph = new Dictionary<int, List<int>> {
       [1] = new() { 2, 3 },
       [2] = new() { 4 },
       [3] = new() { 4 }
};
```

Listing 88: Minimal Graph class with BFS/DFS

```
public sealed class Graph
{
       private readonly Dictionary<int, List<int>> _adj = new();

       public void AddEdge(int u, int v) // directed
       {
              if (!_adj.TryGetValue(u, out var list))
              _adj[u] = list = new List<int>(4);
              list.Add(v);
              // ensure vertex exists even if it has no outgoing edges
              if (!_adj.ContainsKey(v)) _adj[v] = new List<int>(0);
       }
```

```
        public IEnumerable<int> Bfs(int start)
        {
                if (!_adj.ContainsKey(start)) yield break;
                var seen = new HashSet<int>();
                var q = new Queue<int>();
                seen.Add(start);
                q.Enqueue(start);
                while (q.Count > 0)
                {
                        int u = q.Dequeue();
                        yield return u;
                        foreach (var v in _adj[u])
                        if (seen.Add(v)) q.Enqueue(v);
                }
        }

        public IEnumerable<int> Dfs(int start)
        {
                if (!_adj.ContainsKey(start)) yield break;
                var seen = new HashSet<int>();
                var st = new Stack<int>();
                st.Push(start);
                while (st.Count > 0)
                {
                        int u = st.Pop();
                        if (!seen.Add(u)) continue;
                        yield return u;
                        // Push neighbors in reverse to visit left-to-right
                            deterministically
                        var nbrs = _adj[u];
                        for (int i = nbrs.Count - 1; i >= 0; --i)
                        if (!seen.Contains(nbrs[i])) st.Push(nbrs[i]);
                }
        }
}
```

Listing 89: Usage

```
var g = new Graph();
g.AddEdge(1, 2); g.AddEdge(1, 3);
g.AddEdge(2, 4); g.AddEdge(3, 4);

Console.WriteLine("BFS␣from␣1:␣" + string.Join(",", g.Bfs(1))); // 1,2,3,4
Console.WriteLine("DFS␣from␣1:␣" + string.Join(",", g.Dfs(1))); // 1,2,4,3 (given
    insertion order)
```

Notes.

- **Directed vs undirected:** for an undirected graph, call `AddEdge(u,v)` and `AddEdge(v,u)`.
- **Capacity hinting:** if you know typical out-degree, pass it to the list constructor for fewer resizes.
- **Determinism:** DFS order depends on neighbor order; if you require sorted traversal, store neighbors in a `SortedSet<int>` or sort lists before traversing.
- **Weights:** use `Dictionary<int, List<(int v, int w)>>` and adapt traversal (e.g., Dijkstra with a priority queue).

11 Specialized and Modern Structures

11.1 Immutable Collections

The `System.Collections.Immutable` package provides *persistent* (immutable) collections: operations like `Add`/`Remove`/`SetItem` return a *new* collection while the old one remains valid. Under the hood they use **structural sharing** so edits copy only the nodes along an update path, not the entire structure—perfect for functional styles, undo/redo, and highly concurrent readers.

ImmutableList<T> (persistent, indexable sequence)

`ImmutableList<T>` behaves like a list but is implemented as a balanced tree (AVL-like), not a flat array:

- **Ops (big picture):** `Add`/`Insert`/`Remove`/`SetItem` are $O(\log n)$; indexer `[i]` is $O(\log n)$; enumeration is $O(n)$.
- **Sharing:** a modification returns a new list that reuses most of the old tree.
- **Builder:** `ImmutableList<T>.Builder` offers a *mutable* window for batched updates, then `ToImmutable()` to seal with one snapshot.

Listing 90: Basic use and structural sharing

```
using System.Collections.Immutable;

var l0 = ImmutableList<int>.Empty;
var l1 = l0.Add(10); // l1 = [10], l0 unchanged
var l2 = l1.AddRange(new[]{1,2,3});
var l3 = l2.SetItem(1, 42); // replace at index 1 -> [10,42,2,3]
```

Listing 91: Builder pattern to minimize intermediate allocations

```
var b = l3.ToBuilder();
for (int i = 0; i < 1000; i++) b.Add(i);
var l4 = b.ToImmutable(); // one snapshot; l3 still valid
```

When to choose `ImmutableList<T>`.

- You need snapshots/versions (undo/redo, event-sourcing).
- Many readers, occasional writers; no locks needed for readers.
- You want safe sharing between threads without copies.

If you need flat-array semantics and $O(1)$ indexing/appends, prefer `ImmutableArray<T>` or mutable `List<T>`.

ImmutableDictionary<K,V> (persistent hash map)

`ImmutableDictionary<K,V>` is a persistent hash map built over a hash-trie (HAMT-style):

- **Ops:** `Add`/`SetItem`/`Remove`/`TryGetValue` are expected $O(\log n)$ with a small base (branching factor), typically close to constant factors in practice.
- **Comparer-defined** identity via `IEqualityComparer<K>`.
- **Builder:** `ImmutableDictionary<K,V>.Builder` supports efficient bulk updates.

Listing 92: Immutable dictionary usage

```
using System.Collections.Immutable;

var d0 = ImmutableDictionary<string,int>.Empty
.Add("alpha", 1)
.Add("beta", 2);

var d1 = d0.SetItem("beta", 3); // update returns a new map
var d2 = d1.Remove("alpha"); // old maps remain usable

if (d1.TryGetValue("beta", out var v)) Console.WriteLine(v); // 3
```

Listing 93: Bulk updates via Builder

```
var db = d1.ToBuilder();
for (int i = 0; i < 1000; i++) db[$"k{i}"] = i;
var d3 = db.ToImmutable();
```

Costs, Benefits, and When to Use Batching with a Builder.

Aspect	ImmutableList<T>	ImmutableDictionary<K,V>	Notes
Core ops	$O(\log n)$ edits, $O(\log n)$ index	Expected $O(\log n)$ ops	Trees/tries with sharing
Allocation	Path-copy on edit	Path-copy on edit	Old versions remain valid
Threading	Readers lock-free	Readers lock-free	Great for snapshots and concurrency
Locality	Tree/Trie (less cache-dense)	Trie buckets (indirection)	Slower scans than flat arrays/hash tables
Batching	`.Builder`	`.Builder`	Use builders for heavy mutations
Best for	Versions, undo/redo, snapshots	Persistent maps, config/state snapshots	Functional & concurrent reads

Guidance

- Choose **immutable** collections for correctness (no accidental mutation), easy sharing across threads, and cheap snapshots.
- For mutation-heavy hot paths, either use **builders** to batch or stick with mutable `List<T>`/`Dictionary<K,V>` and copy at boundaries.
- Keys in `ImmutableDictionary` obey the same equality/hashing rules as `Dictionary`; pick the right comparer.

11.2 Concurrent Collections

The `System.Collections.Concurrent` namespace provides **thread-safe**, highly scalable collections designed for multi-producer/multi-consumer (MPMC) scenarios. They avoid external locking in user code and expose non-blocking `Try*` methods for progress under contention.

What "thread-safe" means here

- **Linearizable operations:** individual calls like `Enqueue`, `TryDequeue`, `GetOrAdd` appear atomic and are safe under concurrency.
- **Enumeration:** produces a *moment-in-time* snapshot (may not reflect concurrent edits).
- **No external locks required:** internal synchronization (often lock-free/CAS or striped locks) is handled by the collection.

`ConcurrentQueue<T>` (MPMC FIFO)

A lock-free, growable FIFO. Ideal for producer→consumer pipelines when you want *non-blocking* `TryDequeue` loops.

Listing 94: Basic ConcurrentQueue<T> usage

```
using System.Collections.Concurrent;

var q = new ConcurrentQueue<int>();
Parallel.For(0, 1_000_000, i => q.Enqueue(i));

int x;
int taken = 0;
while (q.TryDequeue(out x)) taken++;
Console.WriteLine(taken);
```

Design sketch. Internally uses *segmented arrays* and atomic operations (`Interlocked`) for head/tail movement. This is cache- and contention-friendly under heavy MPMC workloads.

Guidance.

- Prefer `TryDequeue`/`TryPeek` loops; avoid polling `Count` (not constant-time under concurrency and quickly stale).
- For **back-pressure** (bounded capacity, blocking waits), wrap in `BlockingCollection<T>` (next).

`BlockingCollection<T>` (bounding + blocking over a concurrent store)

Wraps a concurrent collection (default: `ConcurrentQueue<T>`) and adds **bounded capacity** and **blocking** producers/consumers.

Listing 95: Bounded producer–consumer with BlockingCollection<T>

```
var bc = new BlockingCollection<byte[]>(boundedCapacity: 1024);

// Producer(s)
var prod = Task.Run(() => {
       foreach (var chunk in ReadChunks()) bc.Add(chunk); // blocks when full
       bc.CompleteAdding();
});

// Consumer(s)
var workers = Enumerable.Range(0, Environment.ProcessorCount).Select(_ =>
    Task.Run(() => {
       foreach (var chunk in bc.GetConsumingEnumerable())
       Process(chunk);
})).ToArray();

Task.WaitAll(workers);
```

Why use it.

- **Back-pressure:** prevents unbounded memory growth.
- **Simple shutdown:** `CompleteAdding()` cleanly signals end-of-stream.

`ConcurrentDictionary<TKey,TValue>` (concurrent map)

A high-throughput dictionary with **atomic composites**: `GetOrAdd`, `AddOrUpdate`.

Listing 96: ConcurrentDictionary primitives

```
var cd = new ConcurrentDictionary<string, int>(StringComparer.OrdinalIgnoreCase);

int v = cd.GetOrAdd("alpha", _ => 1); // atomic insert if missing
int newV = cd.AddOrUpdate("alpha", 1, (_,old) => old + 1); // atomic upsert
cd.TryRemove("alpha", out _); // atomic remove
```

Notes. Internally uses striped locks for buckets (or similar) to reduce contention; individual operations are atomic and safe. Avoid long-running work inside value factories to reduce lock hold times.

Other useful types

- **`ConcurrentStack<T>`**: lock-free LIFO; great for work-stealing adjuncts, but order is LIFO and not stable under concurrency.
- **`ConcurrentBag<T>`**: unordered, thread-local buckets + work-stealing; maximal throughput for "dump-and-drain" workloads, but iteration order is undefined and duplicates are fine.
- **`Partitioner`**: builds partitions over sources for balanced parallel consumption (e.g., with PLINQ/`Parallel.ForEach`).

Choosing the right concurrent collection

Goal	Best fit	Why	Notes
FIFO pipeline (non-blocking)	`ConcurrentQueue<T>`	Lock-free MPMC	Add blocking via `BlockingCollection`
Bounded producer–consumer	`BlockingCollection<T>`	Capacity + blocking	Default store = queue (can swap)
Concurrent map (upsert)	`ConcurrentDictionary<K,V>`	Atomic `GetOrAdd`/`AddOrUpdate`	Choose comparer carefully
LIFO work pile	`ConcurrentStack<T>`	Fast push/pop	Not fair; not stable
Unordered dump-and-drain	`ConcurrentBag<T>`	Thread-local buckets	Unordered; duplicates OK

Signal-and-drain with `ConcurrentQueue<T>` (no blocking)

If you do not need back-pressure (or block elsewhere), a simple signal loop works well:

Listing 97: ConcurrentQueue + SemaphoreSlim signal

```
var q = new ConcurrentQueue<int>();
var sem = new SemaphoreSlim(0);

// Producer
_ = Task.Run(() => {
       for (int i = 0; i < 1000; i++) { q.Enqueue(i); sem.Release(); }
});

// Consumer(s)
_ = Task.Run(async () => {
       while (true) {
              await sem.WaitAsync();
              if (q.TryDequeue(out var x)) Consume(x);
       }
});
```

Performance and correctness tips

- **Avoid `Count` in hot paths**: it can be $O(n)$ or contended; drive loops with `Try*` methods or blocking APIs.
- **Minimize allocations**: reuse buffers (`ArrayPool<T>`), avoid boxing in per-item state.
- **Back-pressure matters**: without it, queues can blow up RAM under producer surges—use `BlockingCollection` for bounded pipelines.
- **Snapshots**: enumeration is a snapshot; do not expect it to reflect concurrent writes.
- **Fairness**: none of these guarantee strict FIFO across threads—only per-structure semantics (e.g., *logical* FIFO for queues).

Concurrent Collections — Quick Guide

- Prefer `ConcurrentQueue<T>` for MPMC pipelines; wrap with `BlockingCollection<T>` to add bounding and blocking.
- Use `ConcurrentDictionary<K,V>` for atomic cache/upsert patterns (`GetOrAdd`, `AddOrUpdate`).
- Reach for `ConcurrentBag<T>` only when order doesn't matter and throughput is king.
- Drive consumers with `GetConsumingEnumerable()` or a semaphore; avoid spinning on `Count`.

11.3 Memory-Efficient Variants

Real-world systems often optimize *layout* and *lifetime* to reduce GC pressure and improve cache locality. Three practical tools in C#: **bit arrays** (pack booleans to bits), **object/array pooling** (reuse buffers), and **`stackalloc`** (stack memory for short-lived scratch).

Bit Arrays (pack 8× more `bool`)

A `bool[]` typically uses ≈1 byte per element (plus array/object headers). When you only need on/off flags at scale, pack them into **bits**:

- `System.Collections.BitArray`: dynamic length, bitwise ops (`And`/`Or`/`Not`).
- Manual packing with `uint`/`ulong` + shifts for hot loops (max perf, minimal overhead).

Listing 98: Dense flags with BitArray (e.g., sieve or visited set)

```
using System.Collections;

int n = 1_000_000;
var bits = new BitArray(n); // all false
bits[0] = bits[1] = true; // mark non-primes, for example

// Bitwise combine
var mask = new BitArray(n, defaultValue: false);
mask[2] = true;
bits.Or(mask); // in-place OR
```

Listing 99: Manual bit packing (ultra-compact flags)

```
public sealed class BitFlags
{
        private readonly uint[] _words; // 32 flags per word
```

```
        public int Length { get; }

        public BitFlags(int length)
        {
                Length = length;
                _words = new uint[(length + 31) / 32];
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Get(int i) => (_words[i >> 5] & (1u << (i & 31))) != 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Set(int i) => _words[i >> 5] |= (1u << (i & 31));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear(int i) => _words[i >> 5] &= ~(1u << (i & 31));
}
```

When to use. Huge boolean grids (visited states, bloom-filter-like masks, sieves), compression of sparse features, or any place a `bool[]` dominates memory.

Space intuition

Representation	Bytes per flag (approx)	Notes
`bool[]`	1.0	plus array/object overhead, pointer indirection
`BitArray`	0.125	packed into machine words, easy API
Manual bit-pack (`uint[]`)	0.125	fastest in hot loops, minimal overhead

Object/Array Pooling (reduce GC pressure)

Frequent allocate/free of large temporary arrays ⇒ LOH/GC churn. **Pooling** lets you *rent* and *return* buffers instead of allocating every time.

Listing 100: Rent/return with ArrayPool<T>

```
using System.Buffers;

var pool = ArrayPool<byte>.Shared;
byte[] buf = pool.Rent(64 * 1024); // >= requested size

try
{
        int read = stream.Read(buf, 0, buf.Length);
        // ... process ...
}
finally
{
        // Consider clearing for security if the data is sensitive:
        // Array.Clear(buf, 0, read);
        pool.Return(buf, clearArray: false);
}
```

Guidelines.

- Do not assume rented arrays are zeroed; *they contain old data.*
- Return promptly; do not keep pooled buffers beyond their logical scope.

- For reference types, clear elements if holding references to large objects to avoid prolonging lifetimes.

`ObjectPool<T>` (Microsoft.Extensions.ObjectPool). Pool *objects* with expensive initialization (e.g., parsers, reusable builders).

Listing 101: Lightweight object pooling

```
using Microsoft.Extensions.ObjectPool;

public sealed class StringBuilderPooledPolicy : PooledObjectPolicy<StringBuilder>
{
       public override StringBuilder Create() => new(capacity: 4096);
       public override bool Return(StringBuilder sb) { sb.Clear(); return
            sb.Capacity <= 1<<20; }
}

var pool = new DefaultObjectPool<StringBuilder>(new StringBuilderPooledPolicy());

var sb = pool.Get();
try { /* use sb */ }
finally { pool.Return(sb); }
```

`stackalloc` for Scratch Buffers (zero GC)

`stackalloc` allocates memory on the *stack* (lifetime = current scope). Combined with `Span<T>` / `ReadOnlySpan<T>` it gives allocation-free temporary buffers and parsing workspaces.

Listing 102: Parsing into a stack buffer with Span<T>

```
ReadOnlySpan<byte> src = stackalloc byte[] { (byte)'A', (byte)'B', (byte)'C' };

Span<byte> tmp = stackalloc byte[256]; // stays within current scope
int n = Transform(src, tmp); // write into tmp without GC
var slice = tmp.Slice(0, n);
// ... consume slice ...
```

Rules and limits.

- Lifetime is the current stack frame: do *not* store/return a reference to it.
- Sizes should be modest (a few KBs). Large `stackalloc` can blow the stack.
- Works safely with `Span<T>` (no `unsafe` needed). Not allowed for managed reference types.
- Perfect for small, fixed-size encoders/decoders, temporary format conversions, and tight loops.

Putting it together: pooled or stack scratch?

Scenario	Best tool	Why	Notes
Tiny, short-lived scratch (< 2–4 KB)	`stackalloc` + `Span<T>`	zero GC, hottest path	Scope-limited, careful with size
Medium/large, reusable buffers	`ArrayPool<T>`	avoids LOH/GC churn	Remember to `Return`
Huge boolean grids	`BitArray`/bit-pack	8 × denser than `bool[]`	Bitwise ops are SIMD-friendly
Reusable heavy objects	`ObjectPool<T>`	amortize init cost	Clear state on return

Practical tips

- Profile: optimize hot paths first; pooling helps most where allocations dominate.
- For sensitive data in pooled arrays, clear before returning (or set `clearArray:true`).
- Prefer `Span<T>`/`Memory<T>` APIs to write allocation-free code paths that can use `stackalloc` or pooled buffers interchangeably.
- Bit-pack booleans and small enums; leverage vectorized operations (`System.Numerics`) over word arrays.