Module 6 – Hashing & Advanced Topics

Introduction

Modern software systems rely on fast access to data. Whether retrieving a record in a database, keeping track of variable names inside a compiler, or storing millions of user accounts inside a server, performance matters. Traditional data structures such as arrays and linked lists cannot always provide the efficiency needed for these tasks. This module introduces hashing, hash tables, priority queues, heaps, and three important algorithmic problem-solving paradigms: divide-and-conquer, greedy algorithms, and dynamic programming. These topics form the foundation for advanced computer science and efficient software development.

Hashing and Hash Tables

Hashing is a technique for transforming a piece of data—called a *key*—into a numeric value known as a hash code. This hash code determines where the data should be stored inside a structure called a *hash table*. The purpose is simple: instead of searching sequentially through a list, hashing allows near-instant access to the location of a value.

A hash function should be fast, deterministic, and capable of spreading keys uniformly across the available storage space. A simple demonstration can show how a hash function turns a string into an index. In the example below, each character in the string contributes to the final hash value:

```
static int Hash(string key, int tableSize)
{
    int hash = 0;
    foreach (char c in key)
        hash += c;
    return hash % tableSize;
}
```

Hash tables use this computed index to determine where a given key–value pair must be placed. When two keys produce the same index—a situation known as a collision—the table must apply a collision-handling strategy.

One common approach is **separate chaining**, where each slot of the table holds a list of elements that hash to that position. If multiple keys land in the same bucket, the list simply grows:

```
static List<string>[] table = new List<string>[5];

static void Insert(string key)
{
    int index = Hash(key, table.Length);
    if (table[index] == null)
        table[index] = new List<string>();
    table[index].Add(key);
}
```

Another method, known as **open addressing**, keeps all entries directly in the table. When a collision occurs, the algorithm probes the table to look for the next available slot. A simple approach to probing is linear probing—checking the next cell, then the next, and repeating until an empty spot is found:

```
while (table[index] != null)
    index = (index + 1) % table.Length;
```

Hashing lies behind real-world structures such as dictionaries, symbol tables, associative arrays, and many in-memory index systems. Its average-case constant time makes it essential in high-performance applications.

Priority Queues and Heaps

A priority queue is an advanced data structure where each element carries a priority, and removal always retrieves the highest-priority element rather than the earliest inserted one. This idea appears everywhere: CPU scheduling, pathfinding algorithms, event simulation, and more.

A highly efficient implementation of a priority queue uses a **heap**, which is a complete binary tree satisfying the heap property. In a **min-heap**, the smallest value is always at the root. Despite being a tree, a heap is stored inside an array, where the relationships between parent and child nodes are established using index calculations:

- left child: 2i + 1

- right child: 2i + 2

When inserting a value into a heap, the element is placed at the end of the array and then moved upward until the heap property is restored:

```
heap.Add(value);
int i = heap.Count - 1;

while (i > 0 && heap[i] < heap[(i - 1) / 2])
{
    (heap[i], heap[(i - 1) / 2]) = (heap[(i - 1) / 2], heap[i]);
    i = (i - 1) / 2;
}
```

Because of this structure, both insertion and removal operate in logarithmic time. Many modern programming languages also provide built-in priority queue implementations for convenience. In C#, the `PriorityQueue<TElement,TPriority>` class makes priority queues easy to create:

```
var pq = new PriorityQueue<string, int>();
pq.Enqueue("Clean Room", 3);
pq.Enqueue("Finish Homework", 1);
Console.WriteLine(pq.Dequeue());
```

Heaps also form the basis of the efficient **Heapsort** algorithm and are crucial in algorithms such as Dijkstra's shortest path.

Divide and Conquer

Divide and conquer is an algorithmic strategy where a large problem is split into smaller subproblems, each solved independently, and the solutions combined. The power of this technique lies in its ability to break down complex tasks into manageable parts.

A simple example is **binary search**, which repeatedly halves the search space to locate a target value:

```
static int BinarySearch(int[] arr, int key, int low, int high)
{
    if (low > high) return -1;
```

```
    int mid = (low + high) / 2;

    if (arr[mid] == key) return mid;
    if (key < arr[mid])
        return BinarySearch(arr, key, low, mid - 1);

    return BinarySearch(arr, key, mid + 1, high);
}
```

This recursive division enables binary search to operate in O(log n) time, making it one of the most efficient searching methods for sorted data.

Greedy Algorithms

A greedy algorithm builds a solution step by step by always choosing the option that appears best at the moment. This approach works only for problems where local optimal choices lead to a globally optimal solution.

A classic example is the activity selection problem, where activities must be chosen such that no two overlap. Sorting activities by finishing time allows a greedy selection:

```
int lastEnd = 0;
for (int i = 0; i < start.Length; i++)
{
    if (start[i] >= lastEnd)
    {
        Console.WriteLine($"Activity {i}");
        lastEnd = end[i];
    }
}
```

Greedy methods power network routing, scheduling, spanning tree algorithms (Prim’s and Kruskal’s), and compression schemes such as Huffman coding.

Dynamic Programming

Dynamic programming (DP) is a systematic method for solving problems with *overlapping subproblems* and *optimal substructure*. Instead of recomputing results repeatedly, DP stores previous computations and reuses them.

One of the clearest examples is the computation of Fibonacci numbers. The naive recursive approach recomputes the same values many times, but the DP approach stores them in a table:

```
int[] dp = new int[n + 1];
dp[0] = 0;
dp[1] = 1;

for (int i = 2; i <= n; i++)
    dp[i] = dp[i - 1] + dp[i - 2];
```

Dynamic programming enables efficient solutions for previously intractable problems, such as knapsack optimization, longest common subsequence, matrix chain multiplication, and several graph algorithms.