VIDEO GAME DESIGN WORKSHOP – 2022

GIUSEPPE TURINI

Kettering
UNIVERSITY

TABLE OF CONTENTS

STUDY GUIDE

Study Material:

- These slides.
- Your notes.
- * "Introduction to the Design and Analysis of Algorithms (3rd Ed.)", chap. 1, pp. 1-40.

Practice Exercises:

- Exercises from *: 1.1.4-6, 1.1.8-9, 1.2.1-2, 1.2.4-5, 1.2.9, 1.3.1, 1.4.2-4, 1.4.9-10.

Additional Resources:

- "Introduction to Algorithms (3rd Ed.)", chap. 1, pp. 1-42.

INTRODUCTION TO VIDEO GAME DESIGN

https://en.wikipedia.org/wiki/Early_history_of_video_games

https://en.wikipedia.org/wiki/History_of_video_games

https://en.wikipedia.org/wiki/List_of_video_game_genres

https://en.wikipedia.org/wiki/Video_game_design

The study of algorithms, sometimes called **algorithmics**, ...

"The reason for this may be understood in the following way: it has often been said that a person does not really understand something until after teaching it to someone else. Actually, ***a person does not really understand something until after teaching it to a computer, i.e., expressing it as an algorithm****."*

DONALD E. KNUTH. SELECTED PAPERS ON COMPUTER SCIENCE. 1996.

WHY STUDY ALGORITHMS?

Theoretical Importance: Algorithms are the core of computer science.

Practical Importance: Provide a toolkit of known algorithms.
Provide a framework to design/analyze algorithms.

Two Main Issues Related to Algorithms:

- How to design algorithms?
- How to analyze algorithms?

GAME LEVEL DESIGN

https://en.wikipedia.org/wiki/Level_(video_games)

WHY STUDY ALGORITHMS?

Theoretical Importance: Algorithms are the core of computer science.

Practical Importance: Provide a toolkit of known algorithms.
Provide a framework to design/analyze algorithms.

Two Main Issues Related to Algorithms:

- How to design algorithms?
- How to analyze algorithms?

GAME MECHANIC DESIGN

https://en.wikipedia.org/wiki/Level_(video_games)

WHY STUDY ALGORITHMS?

Theoretical Importance: Algorithms are the core of computer science.

Practical Importance: Provide a toolkit of known algorithms.
Provide a framework to design/analyze algorithms.

Two Main Issues Related to Algorithms:

- How to design algorithms?
- How to analyze algorithms?

GAME USER INTERFACE DESIGN

https://en.wikipedia.org/wiki/Level_(video_games)

WHY STUDY ALGORITHMS?

Theoretical Importance: Algorithms are the core of computer science.

Practical Importance: Provide a toolkit of known algorithms.
Provide a framework to design/analyze algorithms.

Two Main Issues Related to Algorithms:

- How to design algorithms?
- How to analyze algorithms?

TRASH

https://en.wikipedia.org/wiki/Level_(video_games)

WHAT IS AN ALGORITHM?

Algorithm Definition: An algorithm is a sequence of unambiguous instructions for solving a problem, that is: a procedural solution to a problem, including specific instructions to get the correct answer.

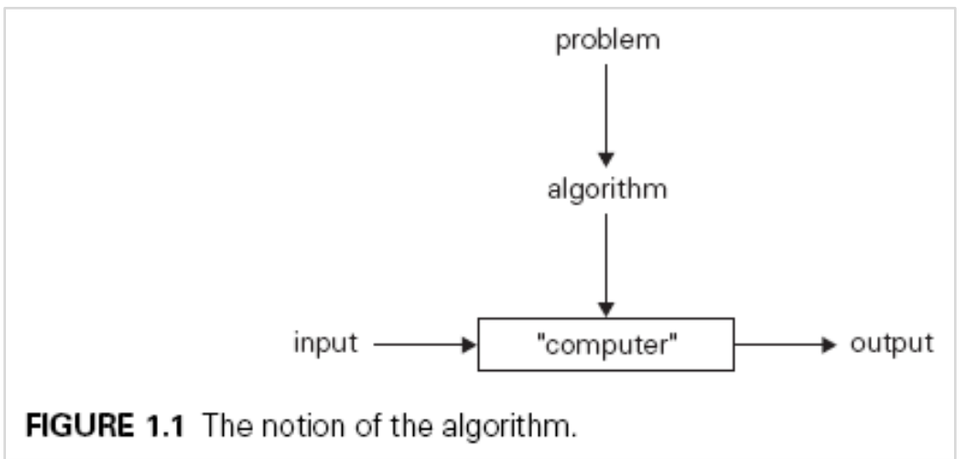

FIGURE 1.1 The notion of the algorithm.

Note: Although most algorithms are intended for computer implementation, the notion of algorithm does not depend on such an assumption!

CONSIDERATIONS ON ALGORITHMS

We will consider different methods for solving the same problem, and this will help us to understand the following important points:

- The **non-ambiguity requirement** for each algorithm step cannot be compromised.
- The **range of inputs** for which an algorithm works has to be specified carefully.
- The same **algorithm can be represented in different ways**.
- There may exist **multiple algorithms to solve the same problem**.
- Algorithms to solve the same problem can be based on very different ideas and can solve the problem in different times.

ALGORITHM EXAMPLE: GCD USING EUCLID'S ALGORITHM

Problem: Find gcd(m, n), the greatest common divisor of two non-negative, not both zero integers m and n.

Examples: gcd(60, 24) = 12, or gcd(60, 0) = 60, or gcd(0, 0) = ?

Euclid's algorithm is based on the repeated application of this equality:

$$gcd(\,m,n\,) = gcd(\,n, m \bmod n\,)$$

... until the second argument becomes 0, which makes the problem trivial. For example:

$$gcd(60{,}24) = gcd(24{,}60 \bmod 24) = gcd(24{,}12) = gcd(12{,}24 \bmod 12) = gcd(12{,}0) = 12$$

Next, we discuss two different descriptions of Euclid's algorithm to compute the GCD ...

ALGORITHM EXAMPLE: GCD USING EUCLID'S ALGORITHM 2

Euclid's Algorithm v1

Step 1: If n = 0, return m and stop; otherwise go to Step 2.
Step 2: Divide m by n and assign the remainder to r.
Step 3: Assign n to m and r to n, then go to Step 1.

```
while( n != 0 ) {
   r = m mod n;
   m = n;
   n = r; }
return m;
```

Euclid's Algorithm v2

Step 1: Assign the value of min(m, n) to t.
Step 2: If m mod t == 0, go to Step 3; otherwise, go to Step 4.
Step 3: If n mod t == 0, return t; otherwise, go to Step 4.
Step 4: Decrease t by 1 and go to Step 2.

```
t = min( m, n );
while( true ) {
   if( m mod t == 0 ) {
      if( n mod t == 0 ) {
         return t; } }
   t--; }
```

ALGORITHM EXAMPLE: GCD USING MIDDLE SCHOOL PROCEDURE

Problem: Find gcd(m, n), the greatest common divisor of two non-negative, not both zero integers m and n.

Examples: gcd(60, 24) = 12, or gcd(60, 0) = 60, or gcd(0, 0) = ?

Middle School Procedure

Step 1: Find the prime factorization of m.
Step 2: Find the prime factorization of n.
Step 3: Find all the common prime factors.
Step 4: Return product of all common prime factors.

Question: Is this an algorithm?

Answer: In this form, the middle-school procedure does **not** qualify as a legitimate algorithm because the prime factorization steps (Step 1 and Step 2) are not defined unambiguously (e.g., they require a list of prime numbers).

ALGORITHM EXAMPLE: GENERATE PRIMES USING SIEVE OF ERATOSTHENES

Problem: Generate all prime numbers not exceeding n (n≥2).

Examples: primes(6) = { 2, 3, 5 }, or primes(7) = { 2, 3, 5, 7 }.

This is the pseudocode for the Sieve of Eratosthenes:

```
// Init array A with integers from 2 to n.
for( p = 2; p <= n; p++ ) {
   A[p] = p; }
// Generate all prime numbers from 2 to sqrt(n).
for( p = 2; p <= floor( sqrt(n) ) ) {
   // Check if p has not been previously eliminated from array A.
   if( A[p] != 0 ) {
      // Remove multiples of p starting from p^2 (other multiples already removed).
      j = p * p;
      while( j <= n ) {
         A[j] = 0; // If multiple of p is less than n, remove current multiple of p.
         j = j + p; // Update j to check next multiple of p.
      }
   }
}
```

ALGORITHMIC PROBLEM SOLVING

The steps to design and analyze an algorithm are:

1. Understand the problem.
2. Check computing device capabilities.
3. Choose algorithm characteristics.
4. Choose algorithm design technique.
5. **Design algorithm** and data structures.
6. Specify-describe the algorithm.
7. Prove algorithm correctness.
8. **Analyze the algorithm**.
9. Code the algorithm.

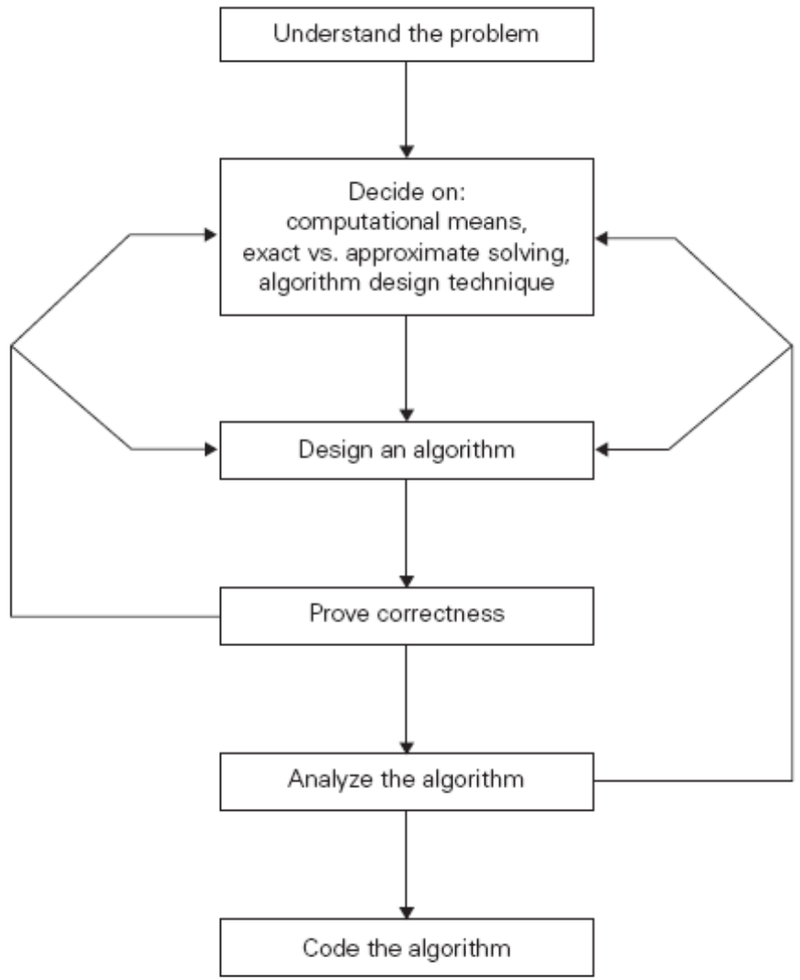

FIGURE 1.2 Algorithm design and analysis process.

UNDERSTAND THE PROBLEM

To design an algorithm, the first step is to understand completely the problem given; so:

- Read carefully the **problem description**.
- Solve a few **examples of the problem** by hand.
- Think about **special cases**.

Problem Instance: An input to an algorithm specifies an instance of the problem the algorithm solves. It is important to specify exactly the instance set of instances the algorithm can handle.

Correct Algorithm: An algorithm that works correctly for all legitimate inputs (not an algorithm that works most of the time).

CHECK COMPUTING DEVICE CAPABILITIES

Sequential Algorithm: If your target computing device is a random-access machine (RAM), which executes instructions sequentially (one after another, one operation at time), you need to design a sequential algorithm.
See: Wikipedia - Sequential Algorithm

Parallel Algorithm: If your target computing device is a computer able to execute operations in parallel, you need to design a parallel algorithm.
See: Wikipedia - Parallel Algorithm

Concurrent Algorithm: Also called multi-threaded algorithms, these are algorithms in which computations are executed concurrently (during overlapping time periods) instead of sequentially (or in parallel).
See: Wikipedia - Concurrent Computing

The methods to design/analyze sequential algorithms remain the pillars of algorithmics.

CHOOSE ALGORITHM CHARACTERISTICS

Exact Algorithm: An algorithm is called an exact algorithm if it solves the problem exactly.

Approximation Algorithm: An algorithm that solves the problem approximately (usually withing a margin of error) is called an approximation algorithm.

Note: We will study only exact algorithms (approximation algorithms are more complex).

Question: Why would someone opt for an approximation algorithm?

Answer: There are important problems that cannot be solved exactly for some instances.
For a specific problem, exact algorithms can be slower than approx. algorithms.
An approx. algorithm can part of another algorithm that solves a problem exactly.

CHOOSE ALGORITHM CHARACTERISTICS 2

Out-of-Core Algorithm: Also called external memory algorithms, these algorithms are designed to process data too large to fit into main memory at once. Such algorithms usually fetch and access data stored in auxiliary memory (e.g., HDs, network memory, etc.).
See: Wikipedia - External Memory Algorithm

In-Place Algorithm: An algorithm which "transforms" input using no auxiliary data structure.
See: Wikipedia - In-Place Algorithm

Stable (Sorting) Algorithm: A sorting algorithm that preserves the order of input data items with equal keys.
See: Wikipedia - Stable Algorithm

CHOOSE THE ALGORITHM DESIGN TECHNIQUE

Algorithm Design Technique: An algorithm design technique is a general approach to solve problems algorithmically, applicable to a variety of problems from different areas of computing.

Algorithm design techniques provide guidance for designing algorithms for new problems, and they make it possible to classify algorithms according to an underlying design idea.

The main algorithm design techniques are:

- Brute Force, and Exhaustive Search.
- Divide-and-Conquer, Decrease-and-Conquer, and Transform-and-Conquer.
- Space and Time Tradeoffs.
- Greedy Approach.
- Dynamic Programming.
- Iterative Improvement.

DESIGN ALGORITHM AND DATA STRUCTURES

With practice both (1) choosing among the general algorithm design techniques, and (2) applying them, get easier, but these two tasks are rarely easy.

Of course, one should pay close attention to choosing data structures appropriate for the operations performed by the algorithm.

Question: Why the Sieve of Eratosthenes would run longer if we used a linked list instead of an array (A) in its implementation?

Answer: Because the only operation required on the list (A) is the access (by index), which runs faster on arrays in respect to linked lists.

SPECIFY-DESCRIBE THE ALGORITHM

Once you have designed an algorithm, you need to specify-describe it in some fashion:

- **Natural Language:** It is easy but its inherent ambiguity makes difficult to describe an algorithm in a succinct and clear way.

- **Pseudocode:** It is a mix of a natural language and programming language constructs. It is usually more concise and precise than natural language.
  See: Wikipedia - Pseudocode

- **Flowchart:** It is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the steps of the algorithm. It is convenient only for very simple algorithms.
  See: Wikipedia - Flowchart

PROVE ALGORITHM CORRECTNESS

Once specified, you have to prove the algorithm correctness, that is:
prove that the algorithm yields the correct result for every legit input in finite time.

A common technique for proving correctness is to use **mathematical induction** because the iterations of an algorithm provide a natural sequence of steps needed for such proofs.

To prove that an algorithm is incorrect, find 1 problem instance for which the algorithm fails.

Note: The correctness for approximation algorithms is shown demonstrating that the error produced does not exceed a predefined limit.

ANALYZE THE ALGORITHM

To analyze an algorithm, we usually focus on one of these target algorithm qualities:

- **Correctness:** Providing a correct result for every legitimate input in a finite time.
- **Time Efficiency:** How fast the algorithm runs.
- **Space Efficiency:** How much "extra" memory the algorithm uses.
- **Simplicity:** Easy to understand, to program, and to debug.
- **Generality of Problem:** Capable of solving different problems (of same general type).
- **Generality of Input:** Handling a set of inputs that is natural for the problem.

If you are not satisfied with the qualities of your algorithm, redesign your algorithm!

"... perfection is finally attained not when there is no longer anything to add,
but when there is no longer anything to take away ..."
ANTOINE DE SAINT-EXUPÉRY. WIND, SAND AND STARS. 1939.

ANALIZE THE ALGORITHM 2

Once a target quality (e.g., time efficiency) is selected, we can analyze the algorithm:

Theoretical Analysis: Analytically-mathematically evaluate the target quality (e.g., time efficiency) of a given algorithm. Sometimes, this type of analysis is only possible under substantial simplifications, or not at all. When possible, this type of analysis does not require the algorithm implementation.

Empirical Analysis: Design an experiment to collect algorithm running data so that the target quality (e.g., time efficiency) can be analyzed. This type of analysis is only possible if we can run an implementation of the algorithm.

CODE THE ALGORITHM

Most algorithms are destined to be ultimately implemented as computer programs.

Programming-implementing an algorithm presents both a peril and an opportunity:

The Peril: lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently.

The Opportunity: is in allowing an empirical analysis of the algorithm.

IMPORTANT PROBLEM TYPES

This is a short list of the most important problem types:

- **Sorting:** Rearranging items in a container in a specific order (e.g., non-decreasing).
- **Searching:** Finding a target value in a container.
- **String Processing:** Searching a word in a text, etc.
- **Graph Problems:** Finding shortest path between two nodes/vertices, etc.
- **Combinatorial Problems:** Graph-coloring problem, etc.
- **Geometric Problems:** Find the closest-pair, generate the convex-hull, etc.
- **Numerical Problems:** Problems involving continuous mathematical objects.

These problem types will be used to discuss different algorithm design techniques, and to learn multiple methods to analyze algorithms.

FUNDAMENTAL DATA STRUCTURES

Most algorithms work on data, so data organization is critical for algorithm design/analysis.

Data Structure: A data structure is a particular organization of data items.

This is a short list of the fundamental data structures for computer algorithms:

- **Linear Data Structures:** If the data items form a sequence (e.g., arrays, linked lists, stacks, queues, circular arrays/lists, doubly-linked lists, etc.).
- **Trees:** A connected directed acyclic graph (DAG)(e.g., reference-based, array-based, BSTs, heaps, etc.).
- **Graphs:** Sets of vertices and edges, with some vertex pairs connected by edges (e.g., adjacency matrix, adjacency lists, etc.).
- **Sets:** An unordered collection of distinct data items.
- **Dictionary:** A data structure implementing search, insertion, and removal operations.

ARRAYS

Array: A sequence of n data items of the same data type: stored contiguously in computer memory, and accessible by index in constant time (see Figure 1).

Figure 1. An example of array capable of storing n items.

LINKED LISTS

Linked List: It is a sequence of zero or more nodes, each containing 2 elements:

- Some data items.
- Some links (pointers or references) to other nodes of the linked lists.

It can be: singly-linked (see Figure 2), doubly-linked (see Figure 3), circular, etc.

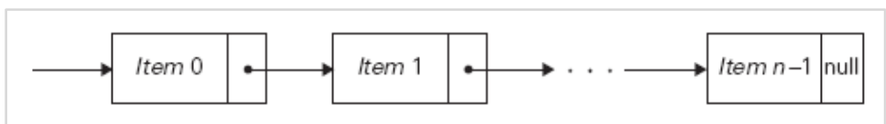

Figure 2. An example of a singly-linked list storing n items.

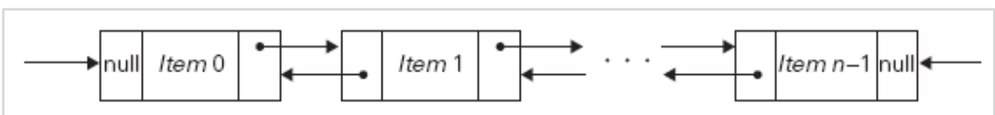

Figure 3. An example of a doubly-linked list storing n items.

STACKS AND QUEUES

Stack: A container in which operations can be done only at one end/position: called the top (the other end being the bottom). This container provides 2 main operations:

- **Push:** to insert a new data item at the top of the stack.
- **Pop:** to remove and return the data item at the top of the stack.

As a result, the stack strategy is called: LIFO (last-in-first-out).

Queue: A container which operates like real-world queues: insertions are performed at the back of the queue, whereas removals are performed at the front of the queue. This container provides 2 main operations:

- **Enqueue:** to insert a new data item at the back of the queue.
- **Dequeue:** to remove and return the data item at the front of the queue.

As a result, the queue strategy is called: FIFO (first-in-first-out).

TREES

Tree: A general tree T is a set of nodes with a hierarchical structure ("parent-child" relationships between nodes) such that T is partitioned into disjoint subsets:

- **The root node r:** the topmost node of the tree T.
- **Subtrees of r:** subsets of T that are trees/branches attached to r.

A tree can be implemented using references or arrays (see Figure 4).

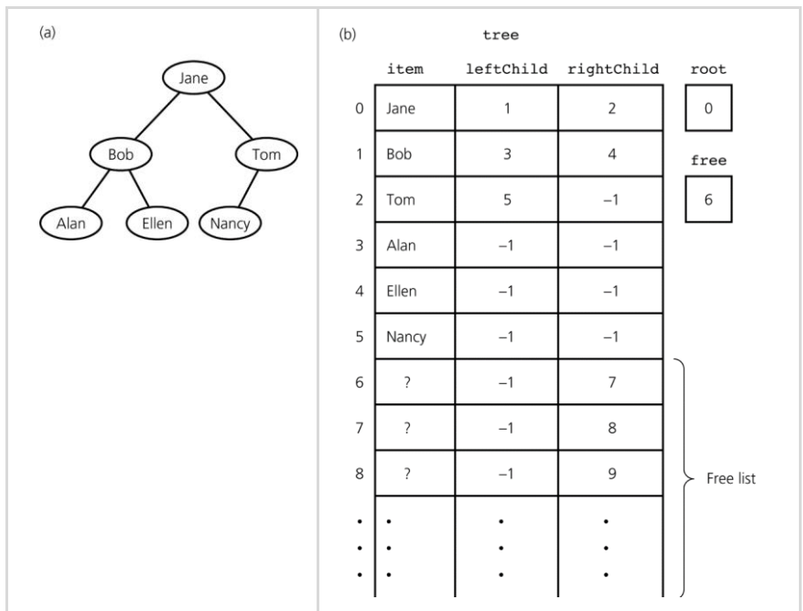

Figure 4. An example of tree implemented using references (a) and arrays (b).

GRAPHS

Graph: A set of vertices/nodes with some vertex pairs connected by edges/links. Formally:

- **V:** a finite non-empty set of vertices/nodes.
- **E:** a set of edges/links, each defined as an ordered/unordered vertex pair.

A graph can be implemented/represented as: an adjacency matrix, an adjacency lists, or an edge list (see Figure 5).

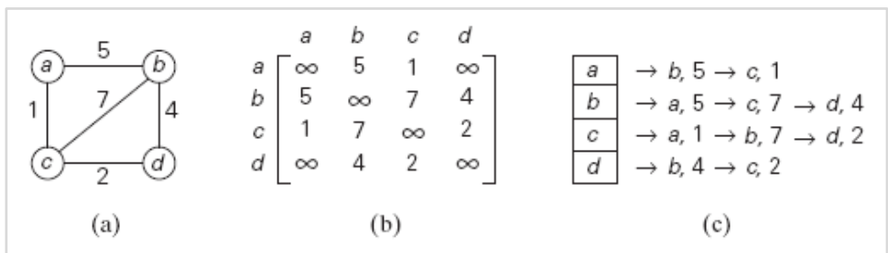

Figure 5. A graph (a) and its representation as adjacency matrix (b) or adjacency list (c).

SETS

Set: An unordered collection of distinct data items (zero or more), usually called elements. The most common operations on sets are:

- Check if a given data item is in a given set.
- Create the union of two sets.
- Create the intersection of two sets.

Sets can be implemented in computer applications in two ways:

- Consider only sets that are subsets of some large set **U (the universal set)**. If set U has n elements, then any subset S of U can be represented by a bit string of size n, called a **bit vector**, in which the i^{th} element is 1 (true) if and only if the i^{th} element of U is included in the subset S.
- Use a **list** to indicate the elements of the set. Note that the list must ensure that the set represented does not contain identical/duplicate elements.

Note that both these implementations work only for finite sets.

DICTIONARIES

Dictionary: Also called table, it is a container that uses a key (search key) to identify its data items (records). The most common operations on dictionaries are:

- Insert a data item.
- Search a data item by key.
- Delete a data item by key.

Usually, dictionaries require stored data items to have distinct keys. So, insertions must fail if attempting to insert a data item with a duplicate key. However, in some cases, it is possible to manage data items with duplicate keys properly.

Dictionaries can be implemented in several different ways, the most common/efficient being using **hashtables**.