Data Structures and Algorithm Analysis

Department of Computer Science

The Course

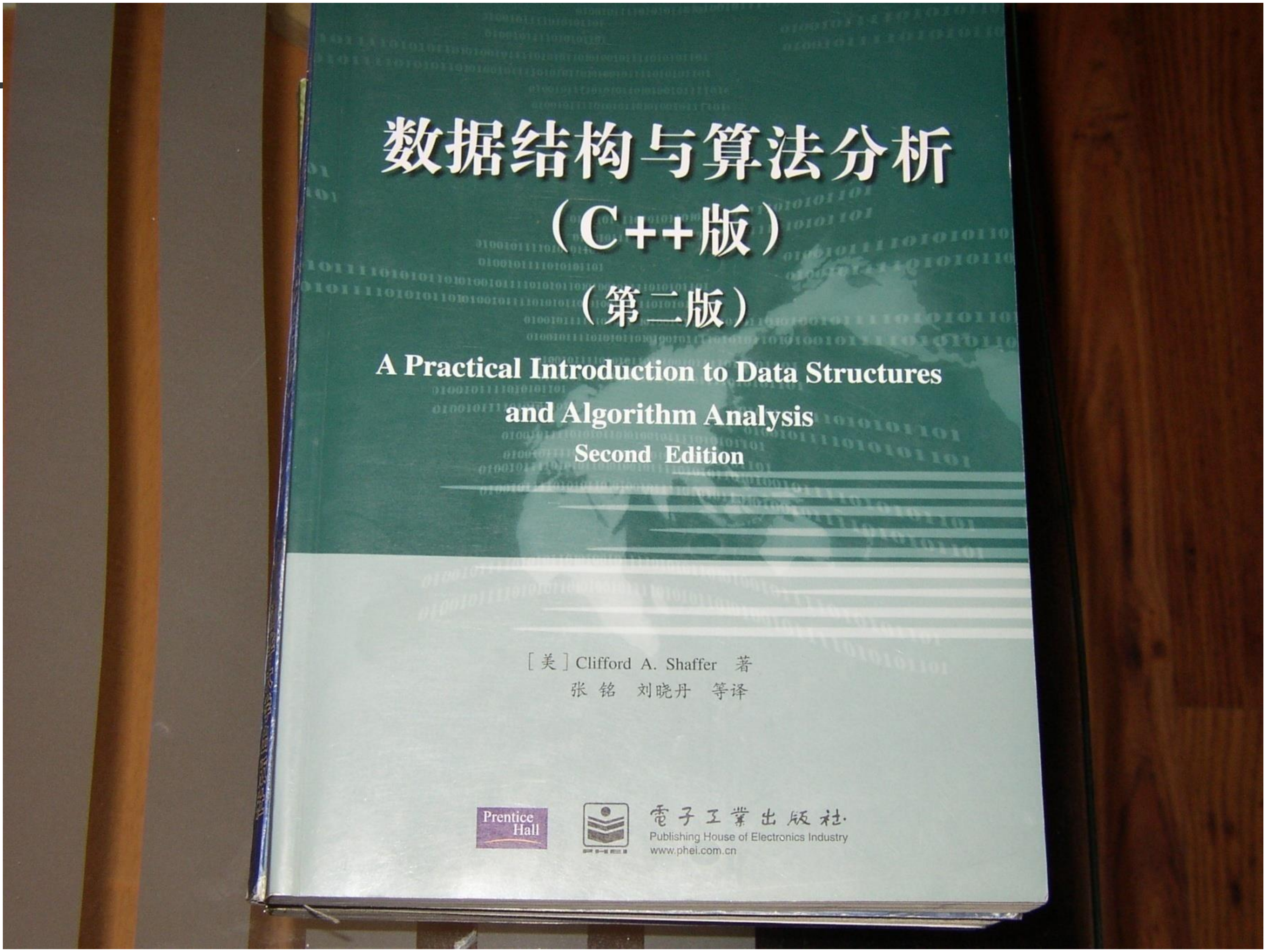
- Purpose: a rigorous introduction to the design and analysis of data structures and algorithms
  - Not a lab or programming course
  - Not a math course, either
- Prerequisites:
  - C or C++
  - Maths

The Course

- Grading policy:
  - Final: 70~80%
  - Exercises: 20~30%
- Format
  - Three hours lectures/week
  - Six or seven exercises
  - final exam

References

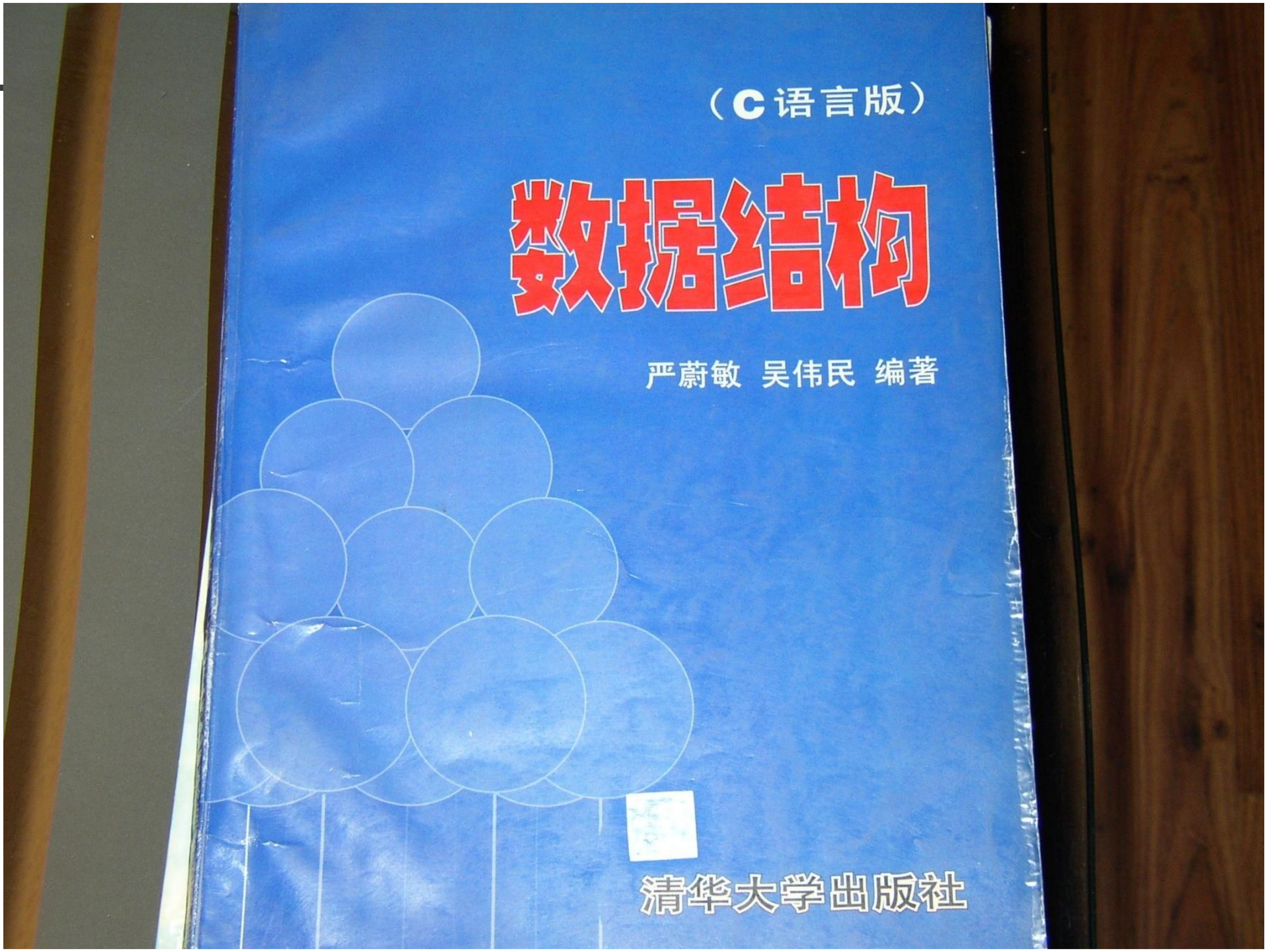
- 数据结构与程序设计
  - *Robert L.Kruse & Alexander J.Ryba*
  - *高等教育出版社*
- 数据结构与算法分析
  - *DATA STRUCTURES AND ALGORITHM ANALYSIS*
  - *CLIFFORD A. SHAFFER著　　张铭 刘晓丹译*
  - *电子工业出版社 PRENTICE HALL出版公司*
- 数据结构（C语言版）
  - *严蔚敏 吴伟民 编著*
  - *清华大学出版社*



数据结构与算法分析
（C++版）
（第二版）
A Practical Introduction to Data Structures and Algorithm Analysis
Second Edition

[美] Clifford A. Shaffer 著
张 铭 刘晓丹 等译

Prentice Hall
電子工業出版社
Publishing House of Electronics Industry
www.phei.com.cn

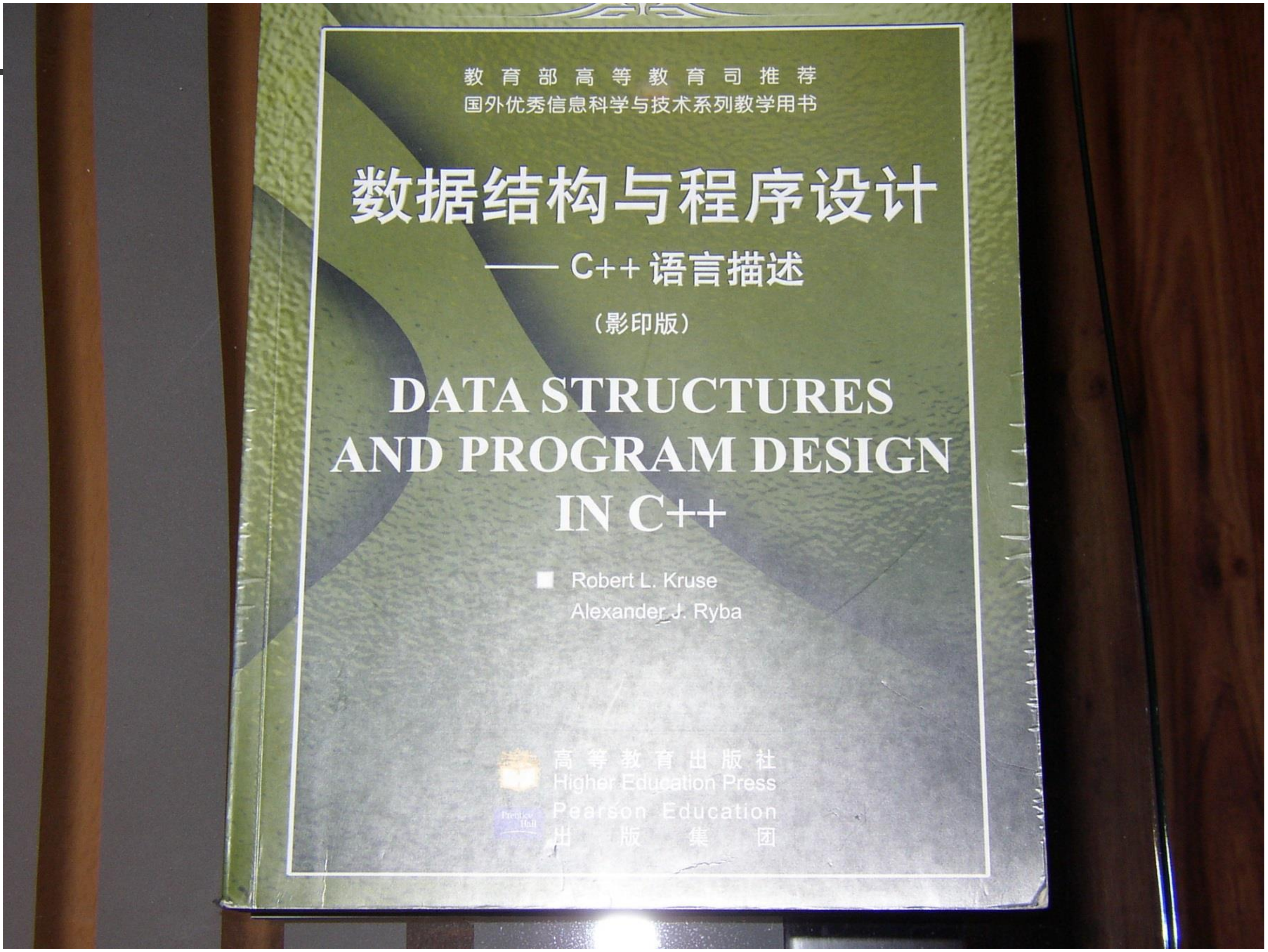


（C语言版）

数据结构

严蔚敏 吴伟民 编著

清华大学出版社



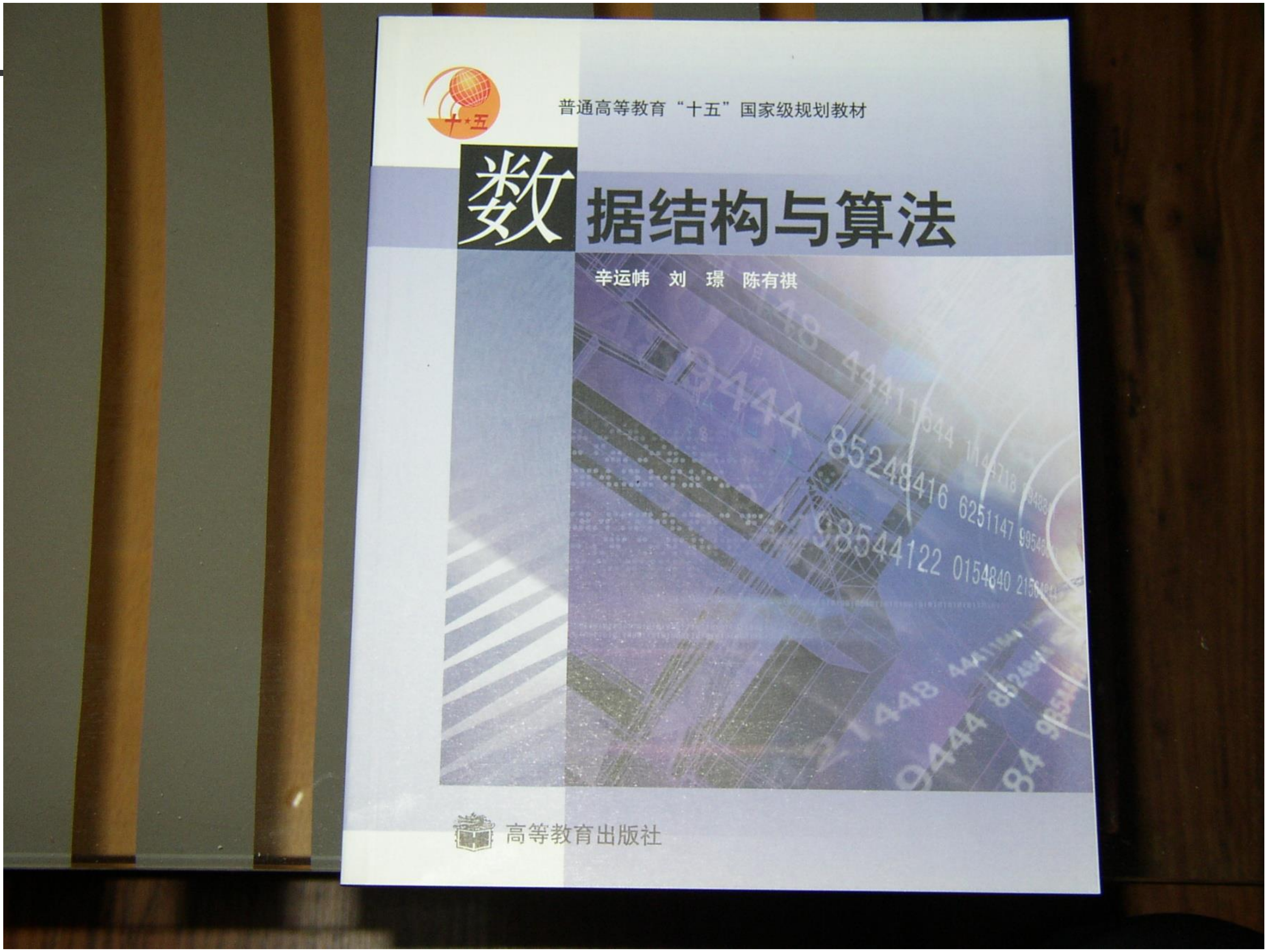
教育部高等教育司推荐
国外优秀信息科学与技术系列教学用书

数据结构与程序设计
—— C++ 语言描述
（影印版）

DATA STRUCTURES
AND PROGRAM DESIGN
IN C++

Robert L. Kruse
Alexander J. Ryba

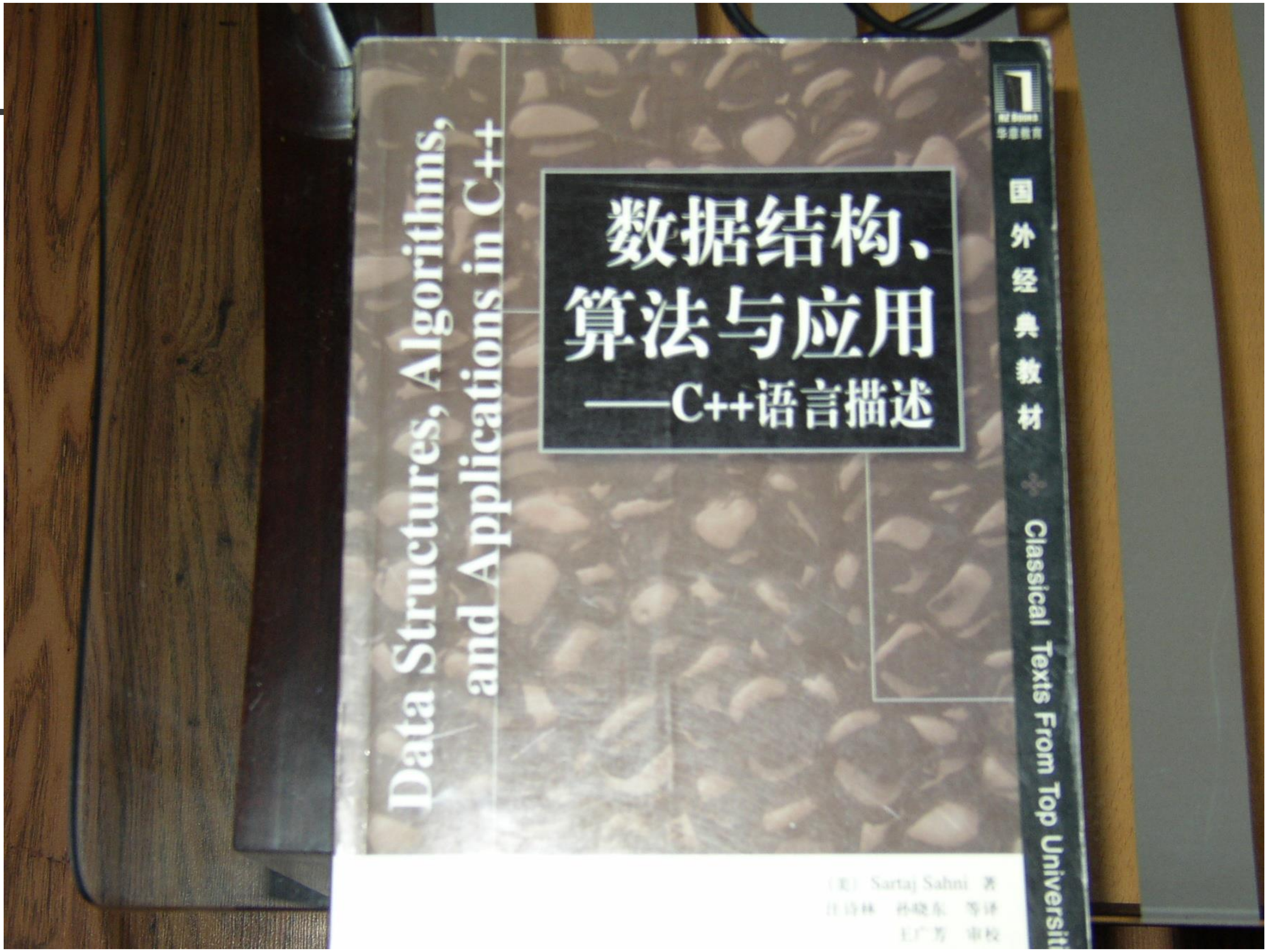
高等教育出版社
Higher Education Press
Pearson Education
出版集团


普通高等教育“十五”国家级规划教材

数据结构与算法

辛运帏　刘　璟　陈有祺

高等教育出版社



华章教育

国外经典教材

数据结构、算法与应用
——C++语言描述

Data Structures, Algorithms, and Applications in C++

Classical Texts From Top Universit

（美）Sartaj Sahni 著
汪诗林 孙晓东 等译
王广芳 审校

General Programming Standards

- Internal Documentation Requirements:
  - Each source and header file must begin with a comment block
  - The source file containing main() must include a comment block
  - Each function must be accompanied by a header comment

General Programming Standards

- Internal Documentation Requirements:
  - Declarations of all local variables and constants must be accompanied by a brief description of purpose.
    - double radius;   // The radius , NO good
    - double area;   // The area , NO good
    - double radius;   // In inches
    - double area;   // In square inches
  - Major control structures should be preceded by a block comment
  - Use a sensible, consistent pattern of indentation and other formatting style
  - Each class must have a header comment block

General Programming Standards

- Certain tags that begin with the symbol @ appear within the comments to identify various aspects of the method.
  - @param to identify a parameter,
  - @return to identify a return value
  - @throws to indicate an exception that the method throws.

General Programming Standards

- **A precondition is a statement of the conditions that must be true before a method begins execution.**
  - **The method should not be used, and cannot be expected to perform correctly, unless the precondition is satisfied.**
  - **A precondition can be related to the description of a method’s parameters. For example, a method that computes the square root of *x can have $x \geq 0$ as a precondition.***

General Programming Standards

- A postcondition is a statement of what is true after a method completes its execution, assuming that the precondition was met.
  - For a valued method, the postcondition will describe the value returned by the method.
  - For a void method, the postcondition will describe actions taken and any changes to the calling object.
  - In general, the postcondition describes all the effects produced by a method invocation.

Example

```
/** Computes the square root of a number.
@param x A real number >= 0.
@return The square root of x.
*/

/** Computes the square root of a number.
@param x A real number.
@return The square root of x if x >= 0.
@throws ArithmeticException if x < 0.
*/
```

General Programming Standards

- An assertion is a statement of truth about some aspect of your program's logic.
- You can think of it as a boolean expression that is true, or that at least should be true, at a certain point.
- Preconditions and postconditions, for example, are assertions made about conditions at the beginning and end of a method.
- If one of these assertions is false, something is wrong with your program.

The assert statement

- You can enforce the assertion by using an **assert statement**, such as

```
assert sum > 0;
```

- If the **boolean expression** that follows the reserved word assert is true, the statement does nothing.
- If it is false, an assertion error occurs and program execution terminates. An **error message** is displayed.
  - assert Expression1
  - assert Expression1:Expression2

General Programming Standards

- **Procedural Coding Requirements:**
  - **Identifier names should be descriptive.**
  - **When a constant is appropriate, use a named constant instead of a "magic number".**
  - **Use enumerated types for internal labeling and classification of state.**
  - **Do not use global or file-scoped variables under any circumstances. It IS permissible to use globally scoped type definitions (including class declarations).**

General Programming Standards

- **Procedural Coding Requirements:**
  - **Pass function parameters with appropriate access.**
    - **Use pass-by-reference or pass-by-pointer only when the called function needs to modify the value of the actual parameter.**
    - **Use pass-by-constant-reference or pass-by-constant-pointer when passing large structures that are not modified by the called function.**
    - **Use pass-by-value when the called function must modify the formal parameter (internal to the call) but the actual parameter should remain unmodified.**

General Programming Standards

- **Procedural Coding Requirements:**
  - **Store character data (aside from single characters) in string objects, rather than char arrays.**
  - **Use new-style C++ at all times; Do not mix old and new style C++ headers.**
  - **Use stream I/O instead of C-style I/O.**

General Programming Standards

- **Object-Oriented Coding Requirements:**
  - **Use classes where they are appropriate. Do not implement struct types with member functions.**
  - **When specified, use a template class for container structures such as lists, trees, etc.**
  - **Design each class to have a coherent set of responsibilities.**
  - **Except for node classes used only with an encapsulating class, all data members of a class should be private.**

General Programming Standards

- Object-Oriented Coding Requirements:
  - In many cases, some of the member functions of a class should also be private. Watch for that situation.
  - If a class data member is a pointer to dynamically allocated memory, implement a destructor to deallocate that memory.
  - If a class data member is a pointer to dynamically allocated memory, implement a deep copy constructor and assignment operator overload.
  - Use inheritance only when it makes sense to do so.

Naming Variables

- Names without meaning are almost never good variable names. The name you give to a variable should suggest what the variable is used for.
  - If the variable holds a count of something, you might name it count.
  - If the variable holds a tax rate, you might name it taxRate.
- In addition to choosing names that are meaningful and legal in language, you should follow the normal practice of other programmers.

Naming Variables

- By convention,
  - each variable name begins with a lowercase letter
  - each class name begins with an uppercase letter
  - If the name consists of more than one word, use a capital letter at the beginning of each word, as in the variable numberOfTries and the class StringBuffer.
  - Use all uppercase letters for named constants to distinguish them from other variables
  - Use the underscore character to separate words, if necessary, as in INCHES_PER_FOOT

匈牙利命名法

- **规则1**　　标识符的名字以一个或者多个小写字母开头，用这些字母来指定数据类型。下面列出了常用的数据类型的标准前缀：

前缀	数据类型
c	字符（char）
s	整数（short）
cb	用于定义对象（一般为一个结构）尺寸的整数
n	整数（integer）
sz	以’\0’结尾的字符串
b	字节
i	int（整数）

匈牙利命名法

- **规则2**　　在标识符内，前缀以后就是一个或者多个首字母大写的单词，这些单词清楚地指出了源代码内那个对象的用途。

匈牙利命名法

x	短整数（坐标x）
y	短整数（坐标y）
f	BOOL
w	字（WORD，无符号短整数）
l	长整数（long）
h	HANDLE（无符号int）
m	类成员变量
fn	函数（function）
dw	双字（DWORD，无符号长整数）

Creating Classes from Other Classes

- **In the first way, you simply declare an instance of an existing class as a data field of your new class. Since your class is composed of objects, this technique is called composition.**
- **The second way is to use inheritance, whereby your new class inherits properties and behaviors from an existing class, augmenting or modifying them as desired. This technique is more complicated than composition.**

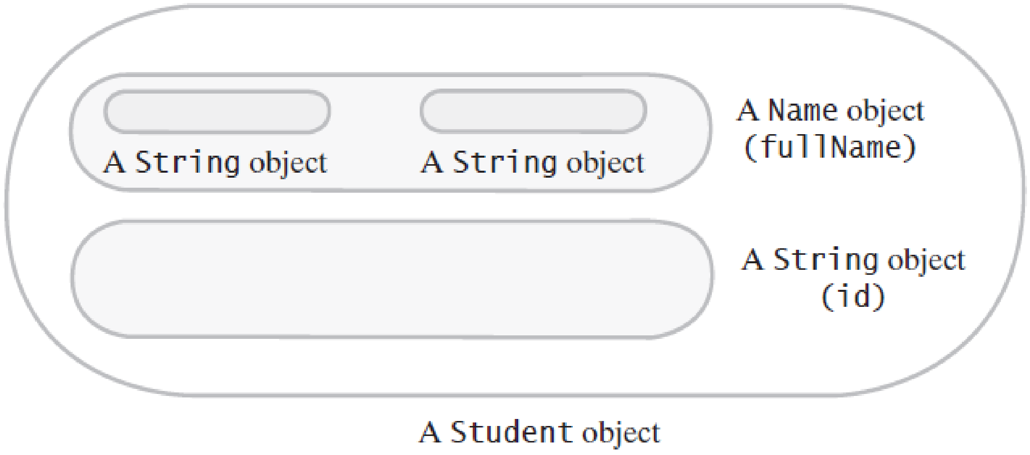
Composition (*has a*)

- A class uses composition when it has objects as data fields. The class's implementation has no special access to such objects and must behave as a client would. That is, the class must use an object's methods to manipulate the object's data. Since the class "has a," or contains, an instance (object) of another class, the classes are said to have a *has a relationship*.

Example

- The data fields in Name are instances of the class String. A class uses composition when it has a data field that is an instance of another class. And since the class Name has an instance of the class String as a data field, the relationship between Name and String is called a ***has a relationship***.

```
private Name fullName;
private String id;
```

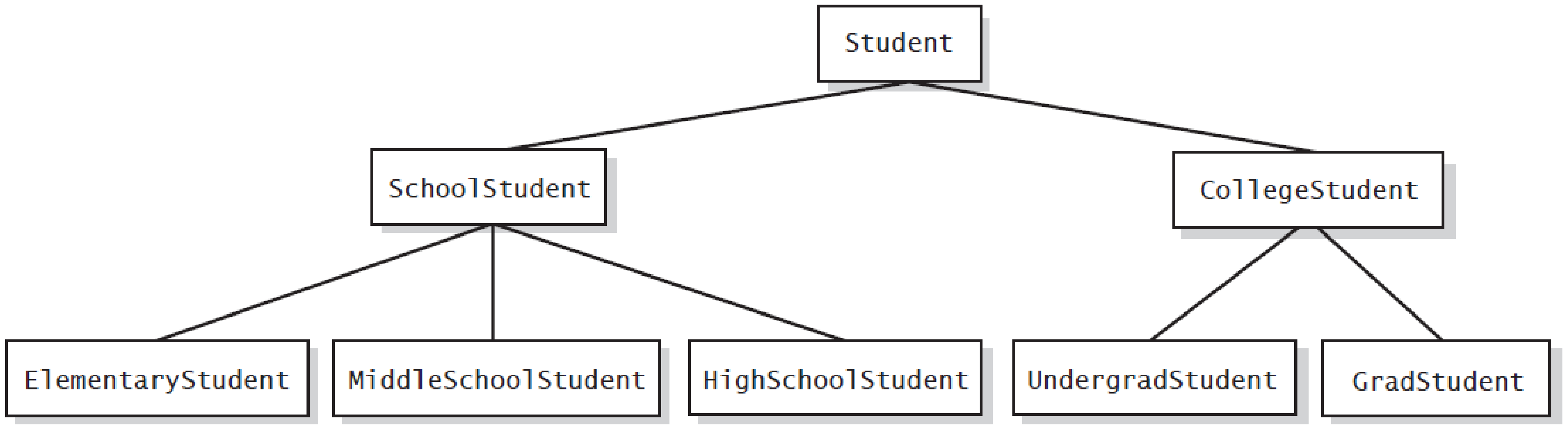


Inheritance(*is a relationship*)

- **Inheritance is a way of organizing classes so that common properties and behaviors can be defined only once for all the classes involved. Using inheritance, you can define a general class and then later define more specialized classes that add to or revise the details of the older, more general class definition.**
- **With inheritance, an instance of a subclass is also an instance of the superclass. Thus, you should use inheritance only when the *is a relationship between classes is meaningful.***

Example

public class CollegeStudent extends Student



The Need for Data Structures

- Data structures organize data ==>more efficient programs.
  - More powerful computers =>more complex applications.
  - More complex applications demand more calculations.
  - Complex computing tasks are unlike our everyday experience.

The Need for Data Structures

- **Any organization for a collection of records can be searched, processed in any order, or modified.**
  - **The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.**

Efficiency

- A solution is said to be efficient if it solves the problem within its resource constraints.
  - Space
  - time
- The cost of a solution is the amount of resources that the solution consumes.

Selecting a Data Structure

- Select a data structure as follows:
  - 1. Analyze the problem to determine the resource constraints a solution must meet.
  - 2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
  - 3. Select the data structure that best meets these requirements.

Some questions to ask:

- Are all data **inserted** into the data structure at the beginning, or are insertions interspersed with other operations?
- Can data be **deleted**?
- Are all data processed in some well-defined **order**, or is **random** access allowed?

Data Structure Philosophy

- Each data structure has costs and benefits.
- Rarely is one data structure better than another in all situations.
- A data structure requires:
  - space for each data item it stores,
  - time to perform each basic operation,
  - programming effort.

Data Structure Philosophy

- **Each problem has constraints on available space and time.**

- **Only after a careful analysis of problem characteristics can we know the best data structure for the task.**

Goals of this Course

- Reinforce the concept that there are costs and benefits for every data structure.
- Learn the commonly used data structures. These form a programmer's basic data structure :toolkit.
- Understand how to measure the effectiveness of a data structure or program.
  - These techniques also allow you to judge the merits of new data structures that you or others might invent.

Definitions

- A ***type*** is a set of values.
- A ***data type*** is a type and a collection of operations that manipulate the type.
- A ***data item*** or ***element*** is a piece of information or a record.
- A data item is said to be a ***member*** of a data type.
- A ***simple data item*** contains no subparts.
- An ***aggregate data item*** may contain several pieces of information.

Abstract Data Types

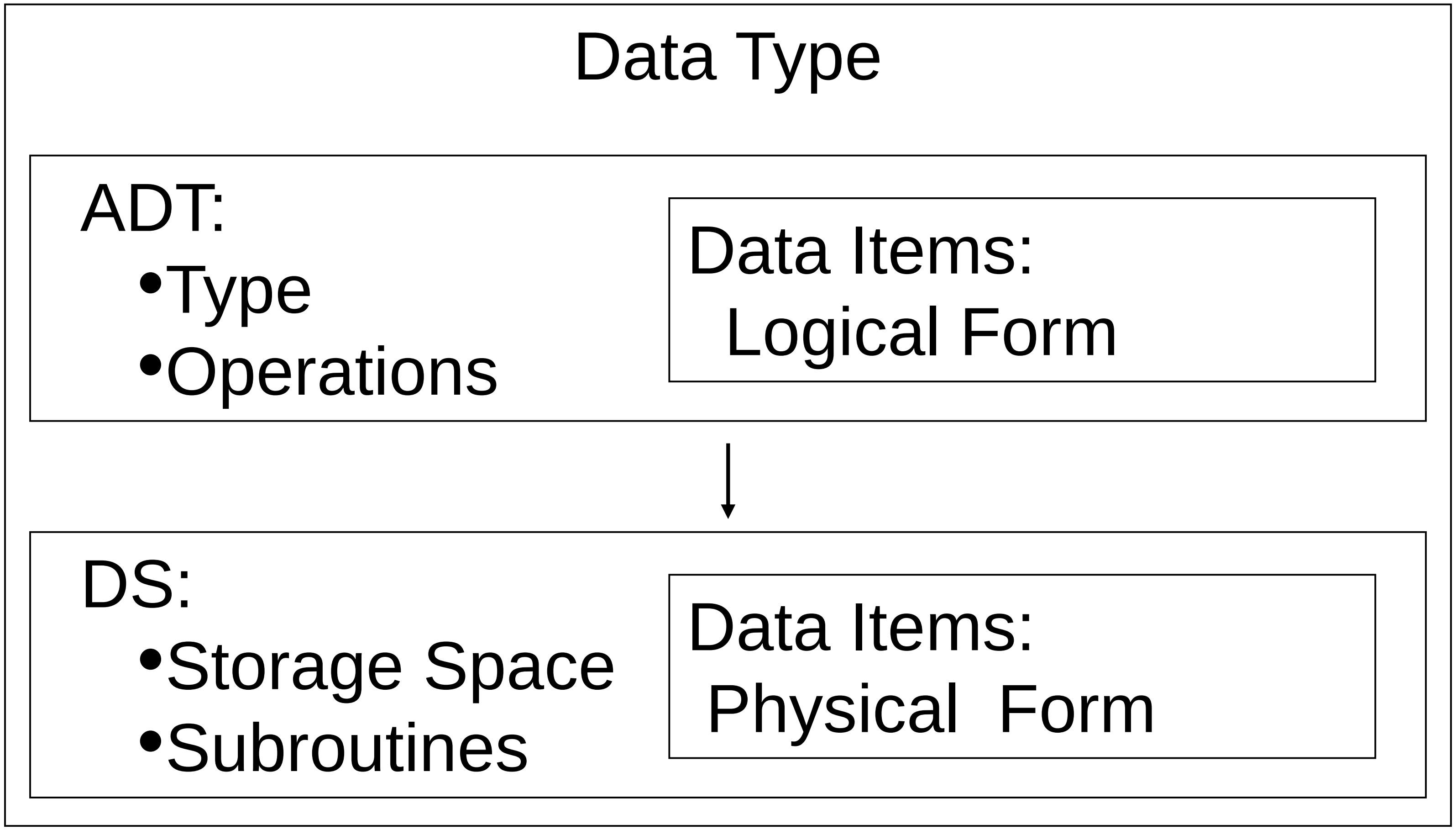
- ***Abstract Data Type (ADT):*** **a definition for a data type solely in terms of a set of values and a set of operations on that data type.**
- **Each ADT *operation* is defined by its inputs and outputs.**
- ***Encapsulation*: hide implementation details**

Abstract Data Types

- A ***data structure*** is the physical implementation of an ADT.
  - Each operation associated with the ADT is implemented by one or more subroutines in the implementation.
- Data structure usually refers to an organization for data in main memory.
- ***File structure***: an organization for data on peripheral storage, such as a disk drive or tape.
- An ADT manages complexity through abstraction: metaphor.

Logical vs. Physical Form

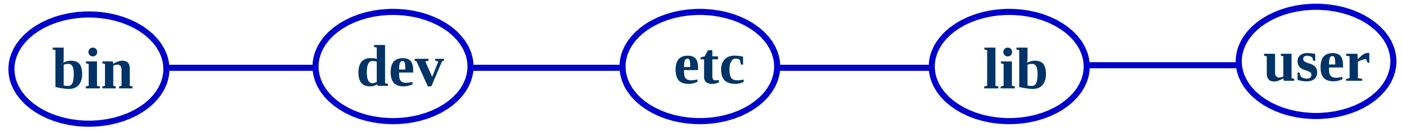
- Data items have both a logical and a physical form.
- ***Logical form***: definition of the data item within an ADT.
- ***Physical form***: implementation of the data item within a data structure.


Data Type
ADT:
•Type
•Operations
Data Items:
Logical Form
DS:
•Storage Space
•Subroutines
Data Items:
Physical Form

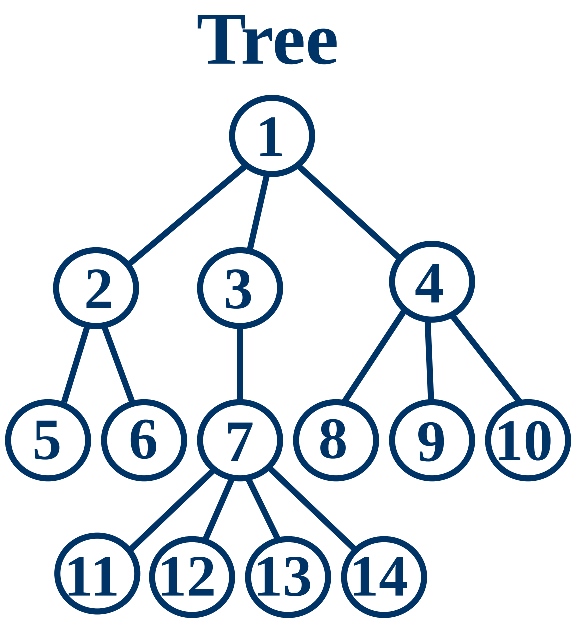
Classification of Logical Form

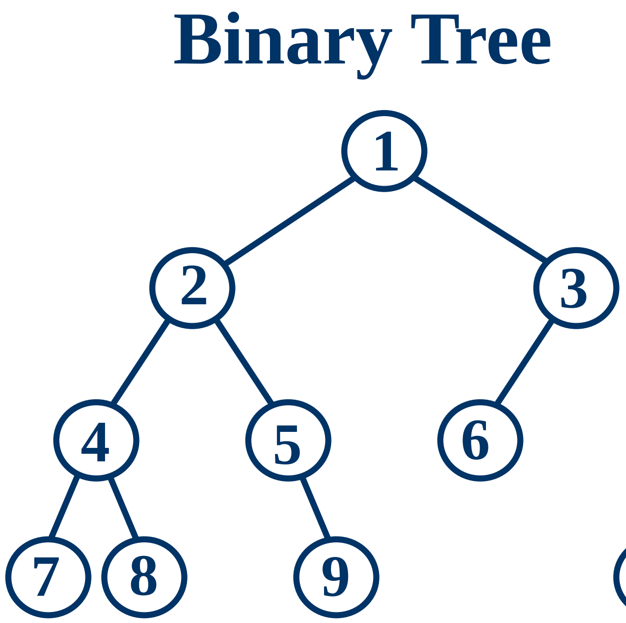
- Linear Structure
  - Linear List
- Non-Linear Structure
  - Multiple Dimensional Array
  - List
  - Tree
  - Graph（or Network）
- Unstructured
  - Set

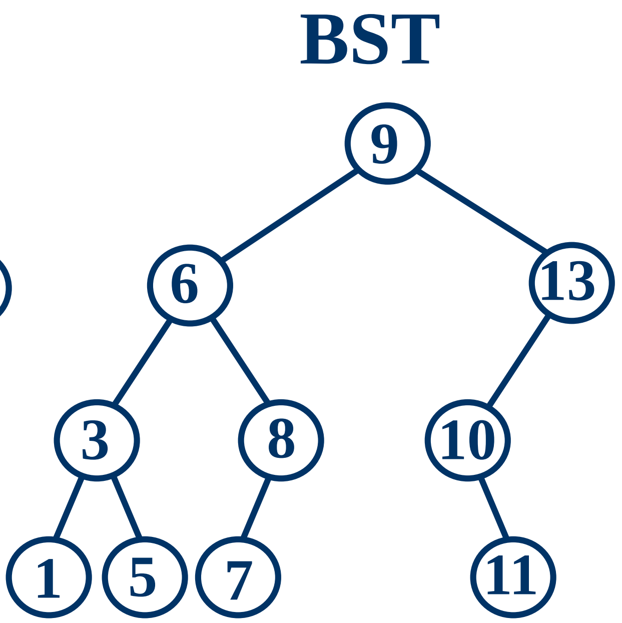
Linear Structure



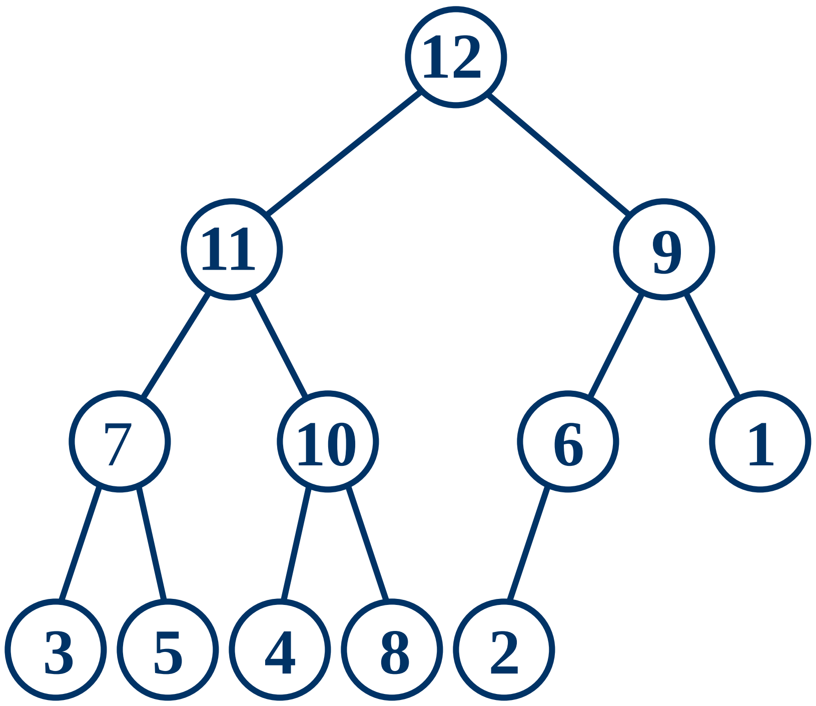
Tree Structure



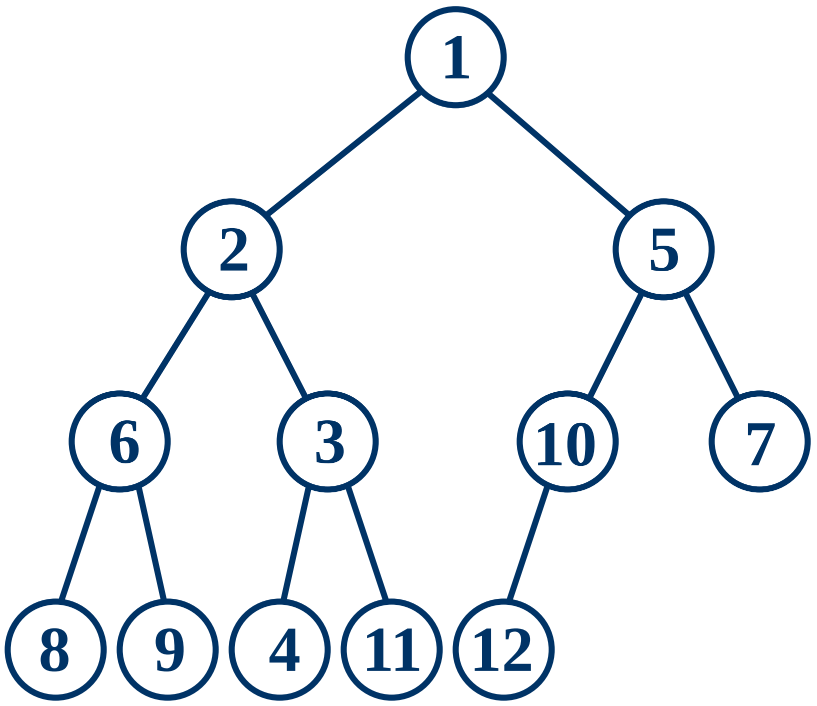




Heap

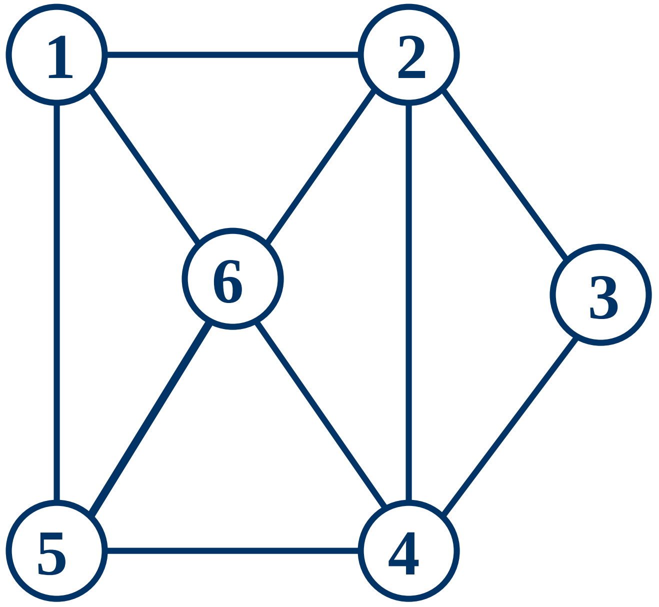


Max Heap

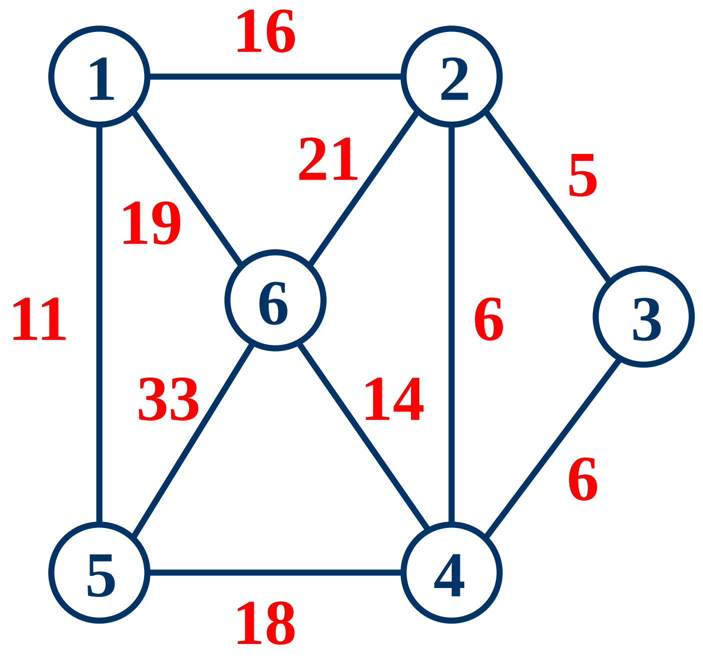


Min Heap

Graph



Network



Physical Form

- Logical structure realized in programming language， three task：
  - Content/Data
  - Relation
  - Additional space/ Temporary space
- Classification
  - Sequential storage representation
  - Linked storage representation
  - Index storage representation
  - Hash storage representation

Problems

- ***Problem***: **a task to be performed.**
  - **Best thought of as inputs and matching outputs.**
  - **Problem definition should include constraints on the resources that may be consumed by any acceptable solution.**

Problems⇔mathematical functions

- A ***function*** is a matching between inputs (the domain) and outputs (the range).
- An ***input*** to a function may be single number, or a collection of information.
- The values making up an input are called the ***parameters*** of the function.
- A particular input must always result in the same output every time the function is computed.

Algorithms and Programs

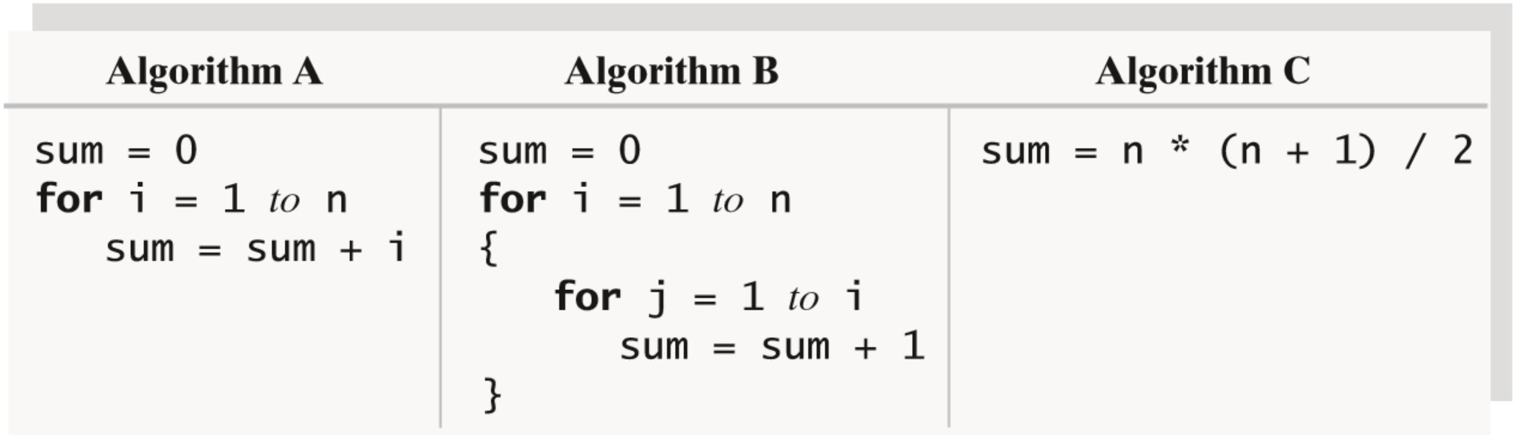
- ***Algorithm***: a method or a process followed to solve a problem.
- An algorithm takes the input to a problem (function) and transforms it to the output.
- A problem can have many algorithms.

An algorithm‘s properties

- **An algorithm possesses the following properties:**
  - **1. It must be correct.**
  - **2. It must be composed of a series of concrete steps.**
  - **3. There can be no ambiguity as to which step will be performed next.**
  - **4. It must be composed of a finite number of steps.**
  - **5. It must terminate.**

- Consider the problem of computing the sum

  $1 + 2 + \ldots + n$ for any positive integer n.
- Figure below contains pseudocode showing three ways to solve this problem.



Algorithm A

```
sum = 0
for i = 1 to n
    sum = sum + i
```

Algorithm B

```
sum = 0
for i = 1 to n
{
    for j = 1 to i
        sum = sum + 1
}
```

Algorithm C

```
sum = n * (n + 1) / 2
```

- Algorithm A computes the sum 0 + 1 + 2 + . . .+ n from left to right.
- Algorithm B computes 0 + (1) + (1 + 1) + (1 + 1 + 1) + . . . + (1 + 1 + . . . + 1).
- Finally, Algorithm C uses an algebraic identity to compute the sum.

// Computing the sum of the consecutive integers from 1 to n:

- long n = 10000; // Ten thousand
- // Algorithm A

```
long sum = 0;
for (long i = 1; i <= n; i++)
        sum = sum + i;
System.out.println(sum);
```

- // Algorithm B

```
sum = 0;
for (long i = 1; i <= n; i++)
{
        for (long j = 1; j <= i; j++)
              sum = sum + 1;
 } // end for
System.out.println(sum);
```

- // Algorithm C

```
sum = n * (n + 1) / 2;
System.out.println(sum);
```