CPT102

> 内容完全来自于**LearningMall**的PPT。仅供学习和交流使用。任何机构或个人不得将其用于商业用途或未经授权的传播。如需引用或分享，请注明出处并保留此声明。
>
> GITHUB REPOSITORY
>
> VISIT NOTES AUTHOR'S PAGE CLICK ME

CPT102 **Data Structures and Algorithms** 数据结构和算法

■ [Week 1 Lecture 0]

课程情况

Steven Guan	**Room SD425**	**steven.guan@xjtlu.edu.cn**
Kok Hoe WONG	**Room SD431**	kh.wong@xjtlu.edu.cn

课程占比

1. 5学分
2. 共2次assessments，占比共20%
3. 一个 2 小时的考试，占比 80%

Q&A

1. Name three typical type of data values in common data collections.

> 三种集合的典型数据类型值：整数（Integer）、字符串（String）、布尔值（Boolean）

2. Name three typical structures seen in common data collections.

> 三种典型的结构：数组（Array）、链表（Linked List）、栈（Stack）

3. Name three typical operations seen in common data collections.

> 三种典型操作：插入（Insert）、删除（Delete）、搜索（Search）。

4. Why do we learn data structure?

> 为什么我们学习数据结构？数据结构帮助我们有效地存储（storage）和管理（manage）数据，优化（optimize）数据处理的性能，是计算机科学的基础。

5. Why do we insist an algorithm must terminate?

> 为什么我们要求算法必须终止？
>
> - 算法的主要目的是解决问题，而问题需要有一个明确的解决方案。

> - 如果算法不终止，它将无限运行，无法提供结果，从而无法达到解决问题的目的。
> - 终止性（Termination）是算法有效性（Effectiveness）和实用性（Practicality）的基本要求。

6. Why do we insist an algorithm must be precise?

> 为什么我们要求算法必须精确？精确的指令确保算法能够被正确执行，避免歧义（ambiguity）或错误。

7. Why instructions in an algorithm are written in a sequence?

> 为什么算法中的指令是按顺序编写的？顺序性（Sequentially）确保算法步骤的逻辑清晰，便于理解和执行。

■ [Week 1 Lecture 1]

数据结构是一种 **系统化的(**Systematic) 方式，用于组织数据集合以便**高效访问**(Efficiency access)。

每种数据结构都需要多种算法来处理其中的数据，因此，我们从以下方面研究数据结构

1. **属性** (properties)
2. **组织方式** (organisation)
3. **操作** (operations)

为什么我们需要数据结构?

> 虽然 原始类型（例如`int`, `double`这种） 和 简单数组 可以满足基本需求，但数据结构提供了**更高级、更结构化的方式**来表达和处理复杂信息，使算法描述更清晰高效

我们将会探讨如下数据结构

1. 数组 **Arrays**
2. 列表 **Lists**
3. 栈 **Stacks**
4. 队列 **Queues**
5. 映射 **Maps**
6. 树 **Trees**
7. 图 **Graphs**

好的数据结构能优化效率，做出更好的软件。

因此，接下来我们探讨，有助于开发高质量软件的主要原则：**抽象**（abstraction）、**信息隐藏**（information hiding）、**封装**（encapsulation）。以及**静态结构**、动态结构（static, dynamic structures）之间的选择。

高质量软件3大原则

> 一个抽象的例子是，当我们查看导航时，地图绘制的是一片森林，而不是一颗一颗的树木。

[1] 抽象

我们可以将 抽象 视为一个过程（process）或一个实体（entity）。

- 作为过程，抽象指的是提取（extracting）一个项目或一组项目的基本细节，同时忽略非必要的细节。
- 作为实体，抽象指的是一个模型、视图或某种实际项目的表示。

在软件开发的背景下，我们又可以把 抽象 视为：

- 数据抽象（data abstraction）：确定数据存储和操作方式中哪些细节是重要的，哪些是不重要的。
- 过程抽象（procedural abstraction）：确定任务完成方式中哪些细节是重要的，哪些是不重要的。

抽象的关键：提取组件（components）的共性（commonality）并隐藏其细节。抽象通常关注对象/概念的**外部视图**（outside view）。

一个抽象的例子就是：**哈夫曼编码**（Huffman coding）。它是一种有效的数据编码方法，常用于通信。编码存储在**码本**（Code Book）中。在所有情况下，必须传输 码本 和 编码数据（Encoded data） 以进行解码。

哈夫曼编码

要了解霍夫曼编码是如何工作的，建议看 **这个视频（这个视频讲的非常好！）**。当然，接下来是用图文的形式简单介绍哈夫曼编码。

二进制编码

假设我们需要编码串`A B A A C D C`，那么我们可以用二进制编码。在这里，字母出现的顺序分别是`A B C D`， 因此他们可以用二进制编码：

字母	二进制
A	`0`
B	`1`
C	`10`
D	`11`

然后这个`A B A A C D C`就可以被加密为`0 1 0 0 10 11 10`也就是`0100101110`。但是在解密的时候就会发生**歧义**。例如：组合`BA`(`10`)可能被翻译为`C`。这是因为`B`充当了`C`的前缀。

因此，为了解决歧义，引入了等长编码。

等长编码

字母	等长编码
A	`00`
B	`01`

字母	等长编码
C	`10`
D	`11`

这种编码的好处是一定不会出现前缀，每个字母对应的编码唯一，不会出现歧义。`00 01 00 00 10 11 10`

但是这种编码方式还是太长了。有什么编码方案能满足：

1. 编码出来最短
2. 没有歧义（能加密和唯一解密）

答案是：**哈夫曼编码。**

哈夫曼编码

哈夫曼编码的流程是：

1. 计算每个字母出现的次数。把这些次数当成**节点**
2. 每次相加两个最小的节点，构建成一个二叉树，其父节点是两个最小节点之和。
3. 继续步骤2，但是父节点也要算进去

这么看有点抽象，让我们来写一个例子。

还是这题，例如字符串`A B A A C D C`如何才能构建出哈夫曼编码？

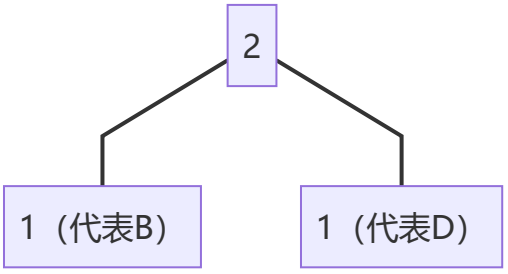
步骤1：计算每个字母出现的次数

`A`：3次
`B`：1次
`C`：2次
`D`：1次

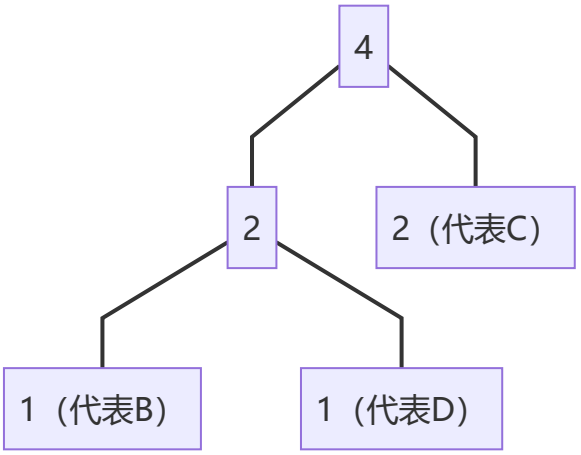
步骤2：构建哈夫曼树

首先有四个节点，3，2，1，1，因此最小的两个节点是`1`和`1`。

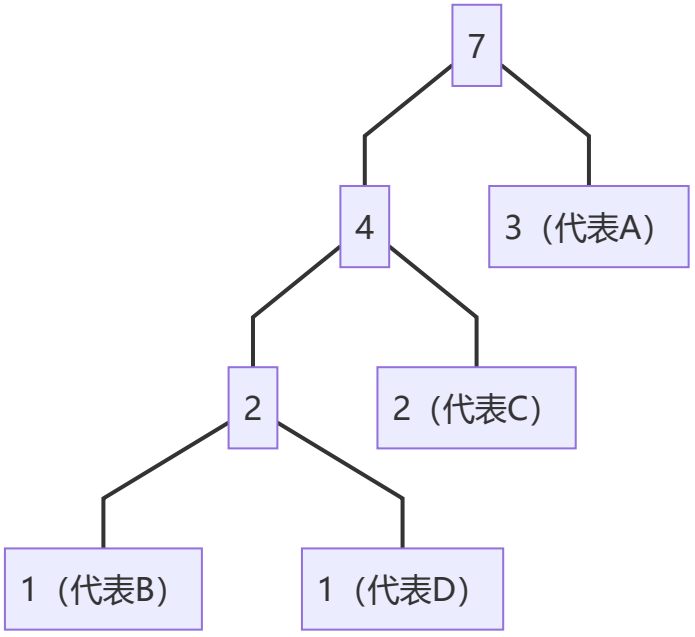
然后相加求出它们的父节点



接下来我们有还剩下三个节点：`3`（代表`A`）、`2`（代表`C`）、`2`（父节点）。因此最小的两个是`2`（代表`C`）和`2`（父节点）。把他们拼接。



最后，我们剩下两个节点：`3`（代表`A`）和`4`（父节点）。因此，把他们拼接。



最后，我们定义左子树的边为`0`，右子树的边为`1`。这样，通过二叉树遍历的方式就能得出，`A,B,C,D`的编码是：

`A`：`1`
`B`：`000`
`C`：`01`
`D`：`001`

因此，`A B A A C D C`对应的哈夫曼编码是：`1000110100101`；

为什么哈夫曼编码能够解决前缀冲突（歧义）问题？因为所有的字母都在叶子节点上。假设A是B的前缀，那么A必须要在B的必经之路上。然而哈夫曼树的每一个字母都必然在叶子上，因此A不可能在B的必经之路上。这就解决了冲突问题。

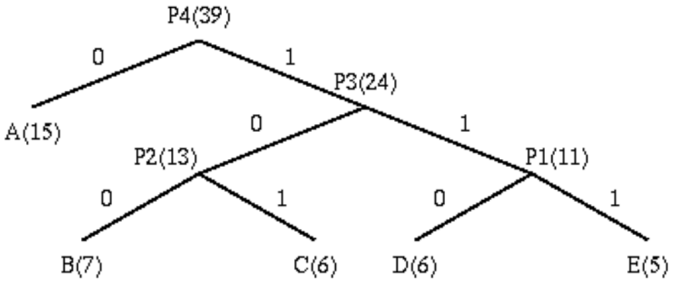
为什么哈夫曼编码能够最短？因为频率越大的字母越往树头靠，这样编码出来的就越短，压缩率更大。

值得注意的是，最开始我们操作`B`和`D`的时候，左右是没有讲究的。虽然左右会影响哈夫曼编码出来的值，但是并不影响其性质。也就是说，**对于一个串，哈夫曼编码的值不是唯一的。**

`Subtotal (# of bits)`指的是**每个符号在编码后所占用的总位数。**实际上计算公式如下：

$$subtotal = frequency * length$$

这里的`frequency`就是频率，`length`就是路径长度（就是走了多少步）。例如：



Symbol	Count	Code	Subtotal (# of bits)
A	15	0	15
B	7	100	21
C	6	101	18
D	6	110	18
E	5	111	15
		TOTAL (# of bits): 87	

这里的`A`对应的解码是`0`，那么其`subtotal = 15 * 1 = 15`。

`B`的`subtotal = 7 * 3 = 21`； 如此往复。

哈夫曼编码使用了**优先队列**(Priority queues)和**二叉树**(Binary trees)。

[2] 信息隐藏

信息隐藏：指模块的使用者只需了解该模块的必要细节（通过**抽象**识别）

> users of a module need to know only the essential details of this module

而抽象帮助我们识别出哪些细节重要，哪些不重要。

信息隐藏旨在将内部数据或信息屏蔽，避免被直接操控（direct manipulation）。

> **信息隐藏示例：汽车与驾驶员**
>
> 汽车通过为驾驶员提供一个**标准化接口**（如踏板、方向盘、变速杆、信号灯、开关等），展示了信息隐藏的概念。这个接口隐藏了汽车内部的复杂机制（如发动机、变速箱、电子系统等）。驾驶员通过培训和考试学习使用这个接口，使他们能够操作任何汽车，而无需了解其内部工作原理。
>
> 或者Java中，通过**封装**的方法来进行信息隐藏。

[3] 封装

与抽象类似，我们可以将**封装**视为一种**过程**或一种**实体**

- 作为一种过程，封装意味着：将一项或多项（数据/函数）纳入一个**容器**中的行为。
- 作为一种实体，封装意味着：一个**容器**，其中包含一项或多项（数据/函数）。（该容器内部与外部的分隔有时被称为“**墙**(Wall)”或“屏障(Barrier)”）

面向对象（O-O, Object-Orient）中的封装

在面向对象编程中，封装是指将对象正常运行所需的全部资源（即方法和数据）包含在对象**内部**。其核心思想是：不要告诉我你是怎么做的；只要让我知道你能做什么！

通信（Communication）中的封装

在通信领域，封装是指将一个数据结构包含在另一个结构内，从而使第一个数据结构被隐藏。

例如，一个 TCP/IP 格式的数据包可以被封装在一个 ATM 帧中。封装的核心目的是将复杂的细节隐藏在一个更简单或标准化的形式中，以便于传输和处理。

总结

抽象 (Abstraction)、**信息隐藏 (Information Hiding)** 和 **封装 (Encapsulation)** 是不同的概念，但它们彼此相关。

- **抽象** 是一种帮助我们识别某个模块中哪些信息对用户是重要的、哪些是不重要的技术。
- **信息隐藏** 是一项原则（Principle），即所有不重要的信息都应该对用户隐藏。
- **封装** 则是将信息打包的技术，目的是隐藏那些应该隐藏的内容，同时暴露那些需要可见的内容。

抽象和封装在**信息隐藏**的指导原则下，我们享有以下优势：

1. 更简单、模块化的程序，更易于设计和理解
2. 消除或最小化**直接操作**(direct manipulation)数据带来的副作用
3. 让错误局部化（只有类上定义的方法才能操作类的数据），从而支持局部化测试
4. 程序模块更易于阅读、修改和维护（maintain）

静态和动态数据结构的选择

除了**时间**和空间效率之外，选择数据结构的另一个重要标准是:能存储的数据项数量是否可以根据我们的需求进行调整，或者是固定的。

动态数据结构：可以在运行时根据当前需求进行增长或缩小。

> 例如用于模拟交通流量的结构。

静态数据结构：在创建时就是固定的

> 例如，用于存储邮政编码或信用卡号码的结构（这些数据具有固定格式）

注意，静态（或动态）的是**结构**，而不是数据。

因此，在静态数据结构中，存储的数据可以随时间变化，但结构是固定的。

> 例如C++中的数组。一般而言，数组长度不能改变。

示例

静态数据结构的一个示例是**数组**：

```
int[] a = new int[50];
```

这行代码分配了可以容纳50个整数的内存空间。

Java提供了`new`一词来向**编译器/解释器**指示需要分配多少空间以及哪种数据类型的空间。

即使我们编写以下代码，数组也总是动态分配的（Dynamically allocated）：

```
int[] a = {10, 20, 30, 40};
```

然而，数组的大小是固定的。但问题是，什么叫**动态分配**?

> 在 Java 中，数组即使在编译时定义大小（例如`int[] a = {10, 20, 30, 40};`），它们仍然是在**运行时分配内存的**。这意味着数组的内存空间是在程序**运行期间**通过堆内存动态获取的（`new`出来的都在堆内存中），而不是在编译时静态地确定和分配。这与静态内存分配的语言（如C中的静态数组）不同，后者在编译时已经确定了内存分配。

问题：Java数组是静态还是动态数据结构?

> Java数组从**内存分配（Memory allocated）**的角度来看是**动态的**，因为它们的内存是在运行时分配的。
>
> 然而，数组的大小在初始化时（at the time of initialize）是固定的，因此在大小上表现出静态特性。

静态数据结构优点和缺点

优点

- 易于指定（Ease of specification）。编程语言通常提供一种简便的方式来创建**任意大小**的静态数据结构。

  没有内存分配开销。

> **没有内存分配开销**指：静态数据结构在运行时**不需要动态分配或释放内存**，从而避免了与动态内存管理相关的性能开销。

缺点:

由于静态数据结构的大小是固定的:

1. 不存在可以用来扩展静态结构的操作;
2. **如果要对静态数据结构进行扩展（例如增加元素），就需要为其分配额外的内存空间**，而分配内存的操作是需要时间的，因此会产生时间和性能上的消耗
3. 一方面，**必须确保有足够的容量**；另一方面，如果存储的数据项数量较少，那么静态数据结构的一部分将保持为空，但这些内存已经被分配，无法用于存储其他数据

动态数据结构的优点和缺点

优点

1. **无需事先知道确切的数据项数量**
2. **内存空间的高效利用**
    1. 在需要添加数据项时（如果这些数据项无法存储在当前结构中），可以扩展动态数据结构的大小。

2. 在动态数据结构中存在未使用的空间时，可以收缩其大小。这样，结构总是能够保持恰到好处的大小，不会浪费任何内存空间。

缺点

1. **内存分配/释放开销**：这意味着在使用动态数据结构（如链表、堆栈或队列）时，系统需要分配新的内存空间来存储增加的数据，或者释放不再需要的内存空间。这些操作会产生一定的性能开销。
2. **每当动态数据结构增长或缩小时，分配给数据结构的内存空间必须增加或减少（这需要时间）**：这意味着随着动态数据结构的扩展或缩小，系统需要调整内存分配，这过程涉及到重新分配内存并可能移动数据，因此会花费额外的时间。

Q&A

1. **空间效率(Space efficienty)**和**时间效率(Time efficienty)**都是用来评估算法（以及数据结构）性能的指标。（对还是错？）

> 这句话是正确的。空间效率衡量的是算法运行所需的最大内存空间，而时间效率衡量的是算法完成任务所需的时间。这两者都是评估算法性能的重要标准。

2. 动态数据结构通常更具空间效率。（对还是错？）

> 这句话是错误的。动态数据结构在内存使用上并不总是比静态数据结构更高效。相反，因为动态数据结构需要**额外的内存来存储指针或链接**，而且可能会导致**内存碎片**，所以静态数据结构在某些情况下可能更节省内存。

3. 静态数据结构通常更具有时间效率。（对还是错？）

> 这句话是正确的。静态数据结构（如数组）由于内存分配是预先确定的，不需要频繁的内存分配和释放操作，因此在访问和操作数据时通常更快，因此具有更高的时间效率。

4. 信息隐藏是指：软件组件的用户只需要知道**如何初始化**和访问组件的必要细节，而不需要了解实现的细节。（对还是错？）

> 对的。比如你开车，只需要知道如何初始化（启动发动机）和如何把握方向盘这些即可。不需要知道内燃机是如何工作的，这属于细节方面的。

总结

数据结构：

数据结构是计算机中**组织和存储数据的方式**，决定了数据的访问和处理效率。

抽象：

抽象是指隐藏复杂的实现细节，只关注问题的核心概念，是设计和理解数据结构的重要方法。

抽象的应用：

1. 数据挖掘（Data Mining）
2. 大数据分析（Big Data Analytics）
3. 数据库知识发现（Knowledge Discovery in Databases / KDD）

哈夫曼编码与动态队列：

哈夫曼编码是一种用于**数据压缩的技术**，动态队列（Dynamic queues）是一种根据数据量动态调整大小的队列结构。

信息隐藏：

信息隐藏是指将实现**细节**与用户接口**分离**，用户只需知道如何使用组件，而不必了解其实现。

封装：

封装是将数据和操作数据的方法绑定在一起，**隐藏内部细节**，确保数据的安全性和一致性。

空间与时间效率：

空间效率是指算法所需的内存资源，时间效率是指算法执行所需的时间，两者都是评估算法性能的关键指　标。

静态与动态数据结构：

静态数据结构（如数组）的大小在编译时确定，而动态数据结构（如链表）的大小可以根据需要动态调整，但动态数据结构通常会牺牲一些空间或时间效率。

■ [Week 2 Lecture 1]

> 使用 Java `Collection` 库

1. 数据结构编程主题概述
2. 使用库进行编程
3. 集合
4. 使用对象列表进行编程

数据结构编程主题概述