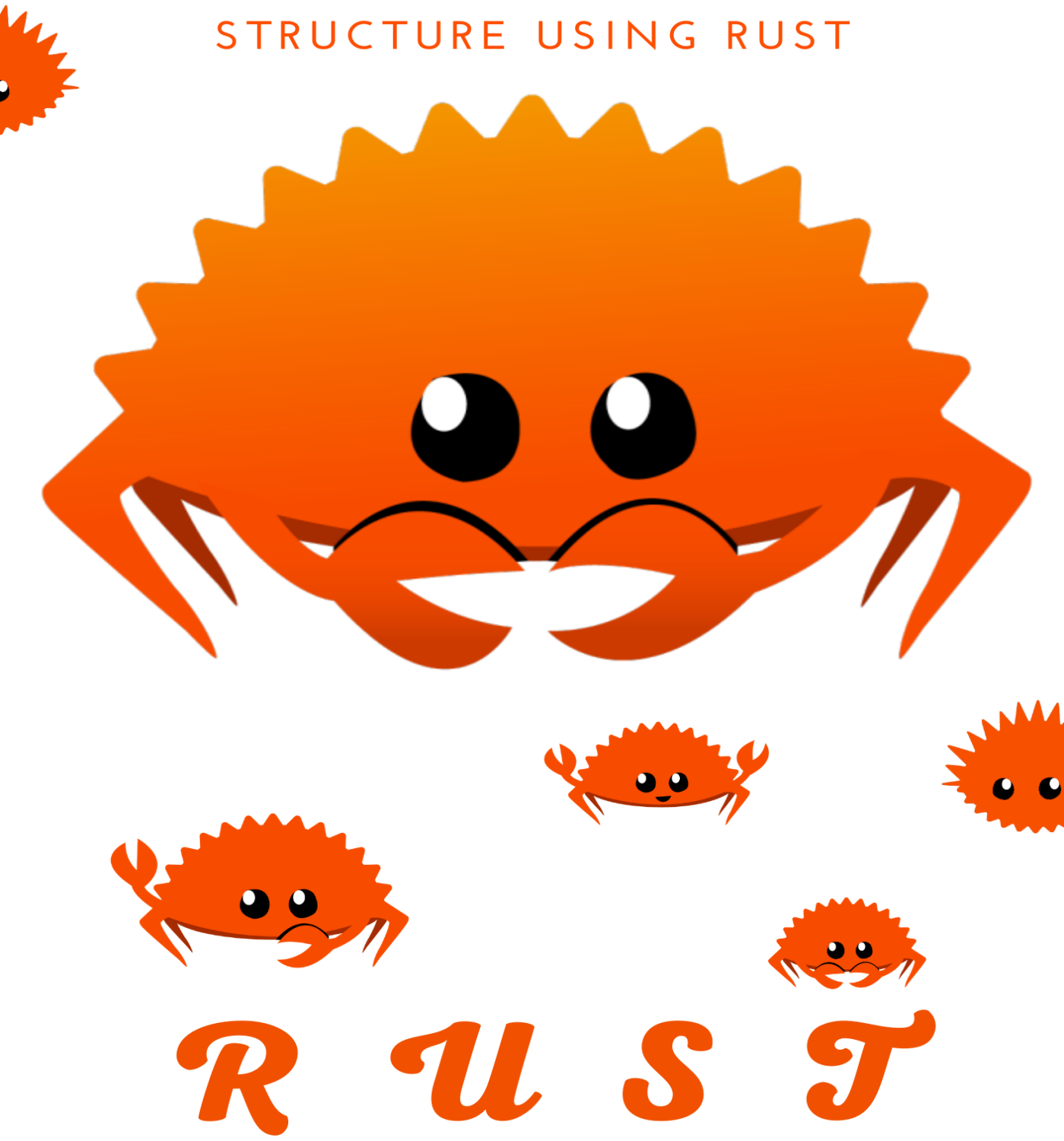
FOR OPEN SOURCE COMMUNITY

PROBLEM-SOLVING WITH ALGORITHMS AND DATA STRUCTURE USING RUST



Problem-solving with algorithms and data structures using Rust

Shieber

2021.02.11

序

晶體管的出現引發了集成電路和芯片革命，人類有了中央處理器，大容量存儲器和便捷的通訊設施。Multics [1] 的失敗催生了 Unix [2]，而後出現的 Linux 內核 [3] 及發行版 [4] 則將開源與網絡技術緊密結合起來，使得信息技術得到飛速發展。技術的進步爲社會提供了實踐想法的平臺和工具，社會的進步則創造了新的需求和激情，繼而又推動技術再進步。儘管計算機世界的上層誕生了互聯網、區塊鏈、雲計算、物聯網等，但在底層，其基本原理保持不變。變化的特性往往有不變的底層原理作爲支撐，就像各種功能的蛋白質也只是幾十種氨基酸組成的肽鏈的曲折變化一樣。計算機世界的底層是基本硬件及其抽象數據類型（數據結構）和操作（算法）的組合，這是變化的上層技術取得進步的根本。

不管是普通計算機，超級計算機亦或將來的量子計算機 [5]，其功能都要建立在某種數據結構和算法的抽象之上。數據結構和算法作爲抽象數據類型在計算機科學中具有重要地位，是完成各種計算任務的核心工具。本書重點關注抽象數據類型的設計、實現和使用。通過學習設計抽象數據類型有助於編程實現，而通過編程實現則可加深對抽象數據類型的理解。

本書的算法實現往往不是最優或最通用的工程實現，因爲工程實現代碼非常冗長，讓人抓不住重點，這對於學習原理是有害的。本書代碼對不同問題採取不同簡化措施，有的直接用泛型，有的則使用具體類型。這些實現方法一是爲簡化代碼，二是確保每段代碼都可單獨編譯通過並得到運行結果。注意，本書使用的 Rust 版本為 1.58。

基本要求

雖然理解抽象數據類型概念不涉及到具體的形式，但代碼實現卻必須要考慮具體的形式，這就要求本書讀者具有一定的 Rust 基礎。雖然每個人對 Rust 的熟悉程度和編碼習慣有所不同，但基本要求卻是相通的。要閱讀本書，讀者最好具有以下能力和興趣：

能使用 Rust 實現完整的程序，包括使用 Cargo、rustc、test 等。

能使用基本的數據類型和結構，包括結構體、枚舉體、循環、匹配等。

能使用 Rust 泛型，生命週期，所有權系統、指針、非安全代碼、宏等。

能使用內置庫（crate）、外部庫，會設置 Cargo.toml。

如果你不會這些，那麼可以翻到第一章末尾，從推薦的學習材料裏找一些書籍和資料來學習 Rust 這門語言，之後再回到本書，從頭開始學習。本書代碼均按照章節和名稱保存在 github code 和 gitee code 倉庫，歡迎下載使用及指出錯誤。

全書結構

爲讓讀者對全書結構有較好的把握以及便於高級用戶選擇閱讀的章節，下面將全書內容做一個總結。全書分爲九章，前兩章介紹計算機科學的概念以及算法分析，是整本書的基礎。第二到第六章是簡單數據結構和算法的設計和實現。第七和八章是較複雜的樹及圖數據結構，樹和圖是許多大型軟件的底層實現，這兩章是基於前幾章的更高級主題。最後一章是利用前面所學內容進行的實戰項目，通過實戰將所學數據結構和算法用於解決實際問題。當然，全書的章節不是死板的，你可以選擇先學某些章節再學其他內容。但總的來說，前兩章要先看，中間的章節及最後的實戰也建議按順序閱讀。

第一章：計算機科學。通過學習計算科學定義和概念能指導個人如何分析問題。其中包括如何建立抽象數據類型模型，設計、實現算法及結果檢驗。抽象數據類型是對數據操作的邏輯描述，隱藏了實現細節，有助於更好地抽象出問題本質。本章的最後是 Rust 基礎知識回顧和學習資源總結。

第二章：算法分析。算法分析是理解程序執行時間和空間性能的方法。由算法分析能得到算法執行效率，比如時間、內存消耗。大 O 分析法是算法分析的一種標准方法。

第三章：基本數據結構。計算機是個線性的系統，對於內存來說也不例外。基本數據結構保存在內存中，所以也是線性的。Rust 中基於這種線性內存模型建立的基本數據結構有數組、切片、Vec 以及衍生的棧、隊列等。本章學習用 Vec 這種基本數據結構來實現棧、隊列、雙端隊列、鏈表。

第四章：遞歸。遞歸是一種算法技巧，是迭代的另一種形式，必須滿足遞歸三定律。尾遞歸是對遞歸的一種優化。動態規劃是一類高效算法的代表，通常利用遞歸或迭代來實現。

第五章：查找。查找算法用於在某類數據集合中找到某個元素或判斷元素是否存在，是使用最廣泛的算法。根據數據集合是否排序可以粗略地分爲順序查找和非順序查找。非順序查找算法包括二分查找和哈希查找等算法。

第六章：排序。前一章的順序查找算法要求數據有序，而數據一般是無序的，所以需要排序算法。常見的排序算法有十類，包括冒泡排序、快速排序、插入排序、希爾排序、歸併排序、選擇排序、堆排序、桶排序、計數排序、基數排序。蒂姆排序算法是結合歸併和插入排序的排序算法，效率非常高，已經是 Java、Python、Rust 等語言的默認排序算法。

第七章：樹。計算機是線性系統，但在線性系統上，通過適當的方法也能構造出非線性的數據結構。樹，正是一種非線性數據結構，它通過指針或引用來指向子樹。樹中最基礎的是二叉樹，由它衍生了二叉堆、二叉查找樹、平衡二叉樹、八叉樹。當然，還有 B 樹、B+ 樹、紅黑樹等更復雜的樹。

第八章：圖。樹的連接從根到子，且數量少。如果將這些限制取消，那麼就得到了圖數據結構。圖是一種解決複雜問題的非線性數據結構，連接方向可有可無，沒有父子結點的區別。雖然是非線性數據結構，但其存儲形式也是線性的，包括鄰接表和鄰接矩陣。圖用於處理含有大量結點和連接關係的問題，例如網絡流量、交通流量、路徑搜索等問題。

第九章：實戰。在前面八章的基礎上，本章通過運用所學知識來解決實際問題，實現一些有用的數據結構和算法。包括距離算法、字典樹、過濾器、緩存淘汰算法、一致性哈希算法以及區塊鏈。通過這些實戰項目，讀者定能加深對數據結構的認識並提升 Rust 編碼水平。

致謝

高效、安全以及便捷的工程管理工具使得 Rust 成爲了一門非常優秀的語言，也是未來可能替代部分 C/C++ 工作的最佳語言（目前 Rust 正逐步加入 Linux 內核）。市面上已經出版了許多 Rust 書籍，但關於算法的書籍要麼是作者沒看到，要麼就是沒有。因爲沒有 Rust 算法書籍，所以作者自己在學習過程中也不斷碰壁。在這樣的情況下，一部簡單便捷的 Rust 書籍對新人學習算法和數據結構來說必然大有幫助。經過一段時間的資料查閱 [6]、思考、整理並結合自己的學習經歷，作者完成了這本書。雖然 Rust 學習曲線陡峭，但只要找准方向，有好的資源，就一定能學好，希望本書能做點微小的貢獻。

編寫這本書主要是爲了學習推廣 Rust，以及回饋整個開源社區，是開源社區的各種資源讓作者學習和成長。感謝 PingCap 開發的 TiDB 以及其運營的開源社區和線上課程。感謝 Rust 語言中文社區的 Mike Tang 等成員對 Rust 會議的組織、社區的建設維護。感謝令胡壹衝在 Bilibili 彈幕網分享的 Rust 學習視頻。感謝張漢東等前輩對 Rust 語言的推廣，包括他寫的優秀書籍《Rust 編程之道》以及 RustMagazine 中文月刊。當然還要感謝 Mozilla、AWS、Facebook、谷歌、微軟、華爲公司爲 Rust 設立基金會 [7]，正是這個基金會使得作者斷定 Rust 的未來一片光明，還缺少學習資源，也纔有了去寫本書的動力。

最後，要感謝電子科技大學提供的學習資源和環境，感謝導師和 KC404 的衆位師兄弟姐妹的關心和幫助。在這裏，作者學到了各種技術、文化，得到了成長，更找准了人生前行的方向。

Shieber 成都

目錄

Chapter 1

計算機科學

1.1 本章目標

瞭解計算機科學的思想
瞭解抽象數據類型的概念
回顧 Rust 編程語言基礎知識

1.2 快速開始

在計算機技術領域，每個人都要花相當多時間來學習該領域的基礎知識，希望有足夠的能力把問題弄清楚並想出解決方案。但是對有些問題，你發現要編寫代碼卻很困難。問題的複雜性和解決方案的複雜性往往會掩蓋與解決問題過程相關的思想。一個問題，往往存在多種解決方案，每種方案都由其問題陳述結構和邏輯所限定。然而，你可能會將解決 A 問題的陳述結構和解決 B 問題的邏輯結合起來，然後自己給自己製造麻煩。本章首先回顧計算機科學、算法和數據結構，特別是研究這些主題的原因，並希望藉此幫助我們看清解決問題的陳述結構和邏輯。其次，本章還回顧了 Rust 編程語言，給出了一些學習資料。

1.3 什麼是計算機科學

計算機科學往往難以定義，這可能是由於在名稱中使用了 “計算機” 一詞。如你所知，計算機科學不僅僅是對計算機的研究，雖然計算機作爲一個工具在其中發揮着最爲重要的作用，但它只是個工具。計算機科學是對問題、解決方案及產生方案的過程的研究。給定某個問題，計算機科學家的目標是開發一個算法，一系列指令列表，用於解決可能出現的該類問題的任何實例。遵循這套算法，在有限的時間內就能解決類似問題。計算機科學可以認爲是對算法的研究，但必須認識到，某些問題可能沒有解決的算法。這些問題可能是 NPC 問題 [8]，目前不能解決，但對其的研究卻是很重要的，因爲解決這些難題意味着技術的突破。就像 “歌德

巴赫猜想[9]”一樣，單是對其的研究就發展出不少工具。或許可以這麼定義計算機科學：一門研究可解決問題方案和不可解決問題思想的科學。

在描述問題和解決方案時，如果存在算法能解決這個問題，那麼就稱該問題是可計算的。計算機科學的另一個定義是“針對那些可計算和不可計算的問題，研究是不是存在解決方案”。你會注意到“計算機”一詞根本沒有出現在此定義中。解決方案獨立於機器，是一整套思想，和是否用計算機無關。

計算機科學涉及問題解決過程的本身，也就是抽象。抽象使人類能夠脫離物理視角而採用邏輯視角來觀察問題並思考解決方案。假設你開車上學或上班。作爲老司機，你爲了讓汽車載你到目的地，會和汽車有些互動。你進入汽車，插入鑰匙，點火，換擋，制動，加速和轉向。從抽象的角度來說，你看到的是汽車的邏輯面，你正在使用汽車設計師提供的功能將你從一個位置運輸到另一個位置，這些功能有時也被稱爲接口。另一方面，汽車修理師則有一個截然不同的視角。他不僅知道如何開車，還知道汽車內部所有必要的細節。他了解發動機是如何工作的，變速箱如何變速，溫度如何控制，雨刷如何轉動等等。這些問題都屬於物理面，細節都發生在“引擎蓋下”。

普通用戶使用計算機也是基於這樣的視角。大多數人使用計算機寫文檔、收發郵件、看視頻、瀏覽新聞、聽音樂等，但用戶並不知道讓這些程序工作的細節。他們從邏輯或用戶視角看計算機。計算機科學家、程序員、技術支持人員和系統管理員看計算機的角度則截然不同，他們必須知道操作系統工作細節，如何配置網絡協議，以及如何編寫各種控制功能腳本，他們必須能夠控制計算機的底層。

這兩個例子的共同點就是用戶態的抽象，有時也稱爲客戶端。用戶不需知道細節，只要知道接口工作方式就能與底層溝通。比如 Rust 的數學計算函數 sin，你可以直接使用。

```
1 // sin_function.rs
2 fn main() {
3     let x: f32 = 2.0;
4     let y = x.sin();
5     println!("sin(2) is {y}"); // 0.9092974
6 }
```

這就是抽象。我們不一定知道如何計算正弦值，但只要知道函數是什麼以及如何使用它就行了。如果正確地輸入，就可以相信該函數將返回正確的結果。我們知道一定有人實現了計算正弦的算法，但細節我們不知道，所以這種情況也被稱爲“黑箱”。只要簡單地描述下接口：函數名稱、參數、返回值，我們就能夠使用，細節都隱藏在黑箱裏面，如圖（1.1）所示。

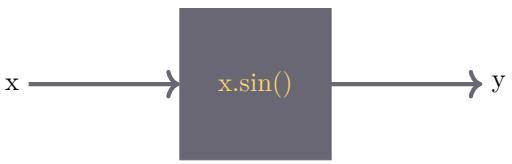


图 1.1: sin 函數

1.4 什麼是編程

編程是將算法（解決方案）編碼爲計算機指令的過程。雖然有許多編程語言和不同類型的計算機存在，但第一步是需要有解決方案，沒有算法就沒有程序。計算機科學不是研究編程，然而編程是計算機科學家的重要能力。編程通常是爲解決方案創建的表現形式，是問題及解決思路的陳述，這種陳述主要提供給計算機，其實編程就是梳理自己頭腦中問題的陳述結構的過程。

算法描述了依據問題實際數據所產生的解決方案和產生預期結果所需的一套步驟。編程語言必須提供一種表示方法來表示過程和數據。爲此，編程語言必須提供控制方法和各種數據類型。控制方法允許以簡潔而明確的方式表示算法步驟。至少，算法需要執行順序處理、決策選擇和重復迭代。只要語言提供這些基本語句，它就可用於算法表示。

計算機中的所有數據項都以二進制形式表示。爲了賦予二進制形式數據具體含義，就需要有數據類型。數據類型提供了對二進制數據的解釋方法和呈現形式。數據類型是對物理世界的抽象，用於表示問題所涉及的實體。這些底層的數據類型（有時稱爲原始數據類型）爲算法開發提供了基礎。例如，大多數編程語言提供整數、浮點數數據類型。內存中的二進制數據可以解釋爲整數、浮點數，並且和現實世界的數字（例如-3、2.5）相對應。此外，數據類型還描述了數據可能存在的操作。對於數字，諸如加減乘除法的操作是最基本的。通常遇到的困難是問題及其解決方案非常複雜，編程語言提供的簡單結構和數據類型雖然足以表示複雜的解決方案，但通常不便於使用。要控制這種複雜性，需要採用更合理的數據管理方式（數據結構）和操作流程（算法）。

1.5 爲什麼學習數據結構

爲了管理問題的複雜性和獲取解決問題的具體步驟，計算機科學家通過抽象以使自己能夠專注於大問題而不會迷失在細節中。通過創建問題域模型，計算機能夠更有效地解決問題。這些模型允許以更加一致的方式描述算法要處理的數據。

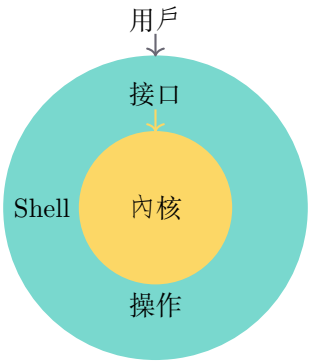


图 1.2: 系統層級圖

先前，我們將解決方案抽象稱爲隱藏特定細節的過程，以允許用戶或客戶端從高層使用。

現在轉向類似的思想，即對數據的抽象。抽象數據類型（Abstract Data Type，ADT）是對如何查看數據和操作數據的邏輯描述。這意味着只用關心數據表示什麼，而不關心它最終形式。通過提供這種級別的抽象，就給數據創建了一個封裝，隱藏了實現細節。上圖展示了抽象數據類型是什麼以及如何操作。用戶與接口的交互是抽象的操作，用戶和 shell 是抽象數據類型。

抽象數據類型的實現要求從物理視圖使用原始數據類型來構建新的數據類型，我們又稱其爲數據結構。通常有許多不同的方法來實現抽象數據類型，但不同的實現要有相同的物理視圖，允許程序員在不改變交互方式的情況下改變實現細節，用戶則繼續專注於問題。

1.6 爲什麼學習算法

在計算機世界中，使用一個更快，或者佔用更少內存的算法是我們的目標，因爲這具有很多顯而易見的好處。在最壞的情況下，可能有一個難以處理的問題，沒有什麼算法在可預期的時間內能給出答案。但重要的是能夠區分具有解決方案的問題和不具有解決方案的問題，以及存在解決方案但需要大量時間或其他資源的問題。作爲計算機科學家需要一遍又一遍比較，然後決定某個方案是否是一個好的方案並決定採用的最終方案。

通過看別人解決問題來學習是一種高效的學習方式。通過接觸不同問題的解決方案，看不同的算法設計如何幫助我們解決具有挑戰性的問題。通過思考各種不同的算法，我們能發現其核心思想，並開發出一套具有普適性的算法，以便下一次出現類似的問題時能夠很好地解決。同樣的問題，不同人給出的算法實現常彼此不同。就像前面看到的計算 sin 的例子，完全可能存在許多種不同的實現版本。如果一種算法可以使用比另一種更少的資源，比如另一個算法可能需要 10 倍的時間來返回結果。那麼即使兩個算法都能完成計算 sin 函數，但時間少的顯然更好。

1.7 Rust 基礎

1.7.1 安裝 Rust

Mac OS，Linux 等類 Unix 系統請使用如下命令安裝 Rust，Windows 下的安裝方式請讀者到官網查看。

$ curl –proto '=https' –tlsv1.2 -sSf https://sh.rustup.rs | sh

安裝好後還需設置環境變量，讓系統能夠找到 rustc 編譯器等工具的位置。對於 Linux 系統，將如下三行加入 ~/.bashrc 末尾，注意修改 usrname 爲自己用戶名。

```
1 # Rust 語言環境變量
2 export RUSTPATH=/home/usrname/.cargo/bin
3 export PATH=$PATH:$RUSTPATH
```

保存退出後再執行 source ~/.bashrc 就可以了。如果嫌麻煩，可以到本書源碼第一章下載 install_rust.sh，執行該腳本就可以完成安裝。

1.7.2 Rust 工具鏈

上面的指令會安裝 rustup 這個工具來管理 Rust 工具鏈。Rust 工具鏈包括編譯器 rustc、項目管理工具 cargo、管理工具 rustup、文檔工具 rustdoc、格式化工具 rustfmt、調試工具 rust-gdb。

平時寫寫簡單代碼可以使用 rustc 編譯，但涉及大項目時還是需要使用 cargo 工具來管理，這是一個非常好的工具，尤其是管理過 C++ 工程的程序員肯定會歡呼這個工具的偉大。cargo 工具集項目構建、測試、編譯、發佈於一體。當然，cargo 內部是調用 rustc 來編譯的，本書項目多不復雜，所以大部分時候也使用 rustc 編譯器而不是 Cargo。

rustup 管理着 Rust 工具的安裝、升級、卸載。注意，Rust 語言包括穩定版和 nightly 版。nightly 版包括最新特性，更新也更快，首次安裝時默認是沒有的，可用如下命令安裝。

$ rustup default nightly

安裝好後可用 rustup 查看。

$ rustup toolchain list

stable-x86_64-unknown-linux-gnu

nightly-x86_64-unknown-linux-gnu (default)

要切換回 stable 版使用如下命令。

$ rustup default stable

1.7.3 Rust 回顧

Rust 是一門類似 C/C++ 的底層編程語言，這意味着你在那兩門語言中學習的很多概念都能用於幫助理解 Rust。然而 Rust 也提出了自己獨到的見解，特別是可變、所有權、借用、生命週期這幾大概念，是 Rust 的優點，也是其難點。

```
1  // rust_basic.rs
2  fn sum_of_val(nums: &[i32], num: i32) -> i32 {
3      let mut sum: i32 = 0;
4      for n in nums {
5          sum += n;
6      }
7      sum + num
8  }
9
10 fn main() {
11     let num = 10;
12     let nums = [1,2,3,4,5,6,7,8];
13     let sum = sum_of_val(&nums, num);
14     println!("sum is {sum}");
15 }
```

上述代碼表明 Rust 需要 main 函數，展示了 println 的格式化輸出，使用 fn 來定義下劃🄴風格的函數以及用 -> 表示返回值，最後一行用無分號來表示返回（return）。

下面是 Rust 目前在用的（未來可能增加）關鍵字，共 39 個，還是比較多的，所以學習要困難一些，特別注意 Self 和 self。

```
1  Self      enum     match     super
2  as        extern   mod       trait
3  async     false    move      true
4  await     fn       mut       type
5  break     for      pub       union
6  const     if       ref       unsafe
7  continue  impl     return    use
8  crate     in       self      where
9  dyn       let      static    while
10 else      loop     struct
```

1.7.4 Rust 學習資料

書籍文檔

入門：《Rust 程序設計語言》、《深入淺出 Rust》、《Rust 編程之道》、《通過例子學 Rust》、《Rust Primer》、《Rust Cookbook》、《Rust in Action》、《Rust 語言聖經》

進階：《Cargo 教程》、《Rustlings》、《通過鏈表學 Rust》、《Rust 設計模式》

高階：《rustc 手冊》、《Rust 宏小冊》、《Rust 死靈書》、《Rust 異步編程》

特定領域

Wasm：https://wasmer.io、https://wasmtime.dev、https://wasmedge.org

HTTP/3: https://github.com/cloudflare/quiche

coreutils: https://github.com/uutils/coreutils

算法：https://github.com/TheAlgorithms/Rust

遊戲：https://github.com/bevyengine/bevy

工具：https://github.com/rustdesk/rustdesk

區塊鏈：https://github.com/w3f/polkadot

數據庫：https://github.com/tikv、https://github.com/tensorbase/tensorbase

編譯器：https://github.com/rust-lang/rustc_codegen_gcc

操作系統：https://github.com/Rust-for-Linux、https://github.com/rcore-os

Web 前端：https://github.com/yewstack/yew、https://github.com/denoland/deno

Web 後端：https://actix.rs/、https://github.com/tokio-rs/axum、https://github.com/poem-web/poem

資源網站

Rust 官網：https://www.rust-lang.org
Rust 源碼：https://github.com/rust-lang/rust
Rust 文檔：https://doc.rust-lang.org/stable
Rust 參考：https://doc.rust-lang.org/reference
Rust 雜誌：https://rustmagazine.github.io/rust_magazine_2021
Rust 庫/箱：https://crates.io、https://lib.rs
Rust 中文社區：https://rustcc.cn
Rust 樂酷論壇：https://learnku.com/rust
Rust LeetCode：https://rustgym.com/leetcode
Awesome Rust：https://github.com/rust-unofficial/awesome-rust
Rust Cheat Sheet：https://cheats.rs
芽之家：https://budshome.com/books.html
令狐壹衝：https://space.bilibili.com/485433391

1.8 總結

本章介紹了計算機科學的思想和抽象數據類型的概念，明確了算法和數據結構的定義和作用。其次，回顧了 Rust 基礎知識並總結了部分學習資源。

Chapter 2

算法分析

2.1 本章目標

理解算法分析的重要性
能夠使用大 O 符號分析算法執行時間
理解 Rust 數組等數據結構的大 O 分析結果
理解 Rust 數據結構的實現是如何影響算法分析的
學習對簡單的 Rust 程序做性能基准測試

2.2 什麼是算法分析

正如我們在第一章中所說，算法是一個通用的，解決某種問題的指令列表。它是用於解決一類問題任何實例的方法，給定特定輸入會產生期望的結果。另一方面，程序是使用某種編程語言編碼的算法。由於程序員知識水平各異且所使用的編程語言各有不同，存在描述相同算法的不同程序。

一個普遍的現象是，剛接觸計算機的學生會將自己的程序和其他人的相比較。你可能注意到，這些程序看起來很相似。那麼當兩個看起來不同的程序解決同樣的問題時，哪一個更好呢? 要探討這種差異，請參考如下的函數 sum_of_n，該函數計算前 n 個整數的和。

```
1 // sum_of_n.rs
2 fn sum_of_n(n: i32) -> i32 {
3     let mut sum: i32 = 0;
4     for i in 1..=n {
5         sum += i;
6     }
7     sum
8 }
```

該算法使用初始值爲 0 的累加器變量。然後迭代 n 個整數，將每個值依次加到累加器。現在再看看下面的函數，你可以看到這個函數本質上和前一個函數在做同樣的事情。不直觀的原因在於編碼習慣不好，代碼中沒有使用良好的標識符名稱，所以代碼不易讀，且在迭代步驟中使用了一個額外的賦值語句，這個語句其實是不必要的。

```
1 // foo.rs
2 fn foo(tom: i32) -> i32 {
3     let mut fred = 0;
4     for bill in 1..=tom  {
5         let barney = bill;
6         fred = fred + barney;
7     }
8     fred
9 }
```

先前提出了一個的問題：哪個函數更好？答案其實取決於讀者的評價標准。如果你關注可讀性，函數 sum_of_n 肯定比 foo 好。你可能已經在各種介紹編程的書或課程中看到過類似例子，其目的之一就是幫助你編寫易於閱讀和理解的程序。然而，在本書中，我們對算法本身的陳述更感興趣（乾淨的寫法當然重要，但那不屬於算法和數據結構的知識）。

算法分析是基於算法使用的資源量來進行比較的。說一個算法比另一個算法好就在於它在使用資源方面更有效率，或者說使用了更少的資源。從這個角度來看，上面兩個函數看起來很相似，它們都使用基本相同的算法來解決求和問題。在資源計算這點上，重要的是要找准真正用於計算的資源。評價算法使用資源往往要從時間和空間兩方面來看。

算法使用的空間指的是內存消耗，解決方案所需的內存通常由問題本身的規模和性質決定。但有時，部分算法會有一些特殊的空間需求，這種情況下需要非常仔細地審查。

算法使用的時間就是指算法執行所有步驟經過的時間，這種評價方式也被稱爲算法的執行時間，可以通過基准分析來測量函數 sum_of_n 的執行時間。

在 Rust 中，可以記錄函數運行前後的系統時間來計算代碼運行時間。在 std::time 中有獲取系統時間的 SystemTime 函數，它可在被調用時返回系統時間並在之後給出經過的時間。通過在開始和結束的時候調用該函數，就可以得到函數執行時間。

```
1 // static_func_call.rs
2 use std::time::SystemTime;
3 fn sum_of_n(n: i64) -> i64 {
4     let mut sum: i64 = 0;
5     for i in 1..=n {
6         sum += i;
7     }
8     sum
9 }
```

```
10
11 fn main() {
12     for _i in 0..5 {
13         let now  = SystemTime::now();
14         let _sum = sum_of_n(500000);
15         let duration = now.elapsed().unwrap();
16         let time = duration.as_millis();
17         println!("func used {time} ms");
18     }
19 }
```

執行這個函數 5 次，每次計算前 500,000 個整數的和，得到了如下結果:

func used 10 ms

func used 6 ms

func used 6 ms

func used 6 ms

func used 6 ms

我們發現時間是相當一致的，執行這個函數平均需要 6 毫秒。第一執行耗時 10 毫秒是因爲函數要初始化准備，而後面四次執行不需要，此時執行得到的耗時纔是比較准確的，這也是爲什麼需要執行多次。如果我們運行計算前 1,000,000 個整數的和，其結果如下:

func used 17 ms

func used 12 ms

func used 12 ms

func used 12 ms

func used 12 ms

可以看到，第一次的耗時還是更長，後面的時間都一樣，且恰好是計算前 500,000 個整數耗時的二倍，這說明算法的執行時間和計算規模成正比。現在考慮如下的函數，它也是計算前 n 個整數的和，只是不同於上一個 sum_of_n 函數的思路，它利用數學公式 $\sum_{i=0}^{n} = \frac{n(n+1)}{2}$ 來計算，效率更高。

```
1 // static_func_call2.rs
2 fn sum_of_n2(n: i64) -> i64 {
3     n * (n + 1) / 2
4 }
```

修改 static_func_call.rs 中的 sum_of_n 函數，然後再做同樣的基准測試，使用 3 個不同的 n （100,000、500,000、1,000,000），每個計算 5 次取均值，得到了如下結果:

func used 1396 ns

func used 1313 ns

func used 1341 ns

在這個輸出中有兩點要重點關注，首先上面記錄的執行時間是納秒，比之前任何例子都短，這 3 個計算時間都在 0.0013 毫秒左右，和上面的 6 毫秒可是差着幾個數量級。其次是執行時間和 n 無關，n 增大了，但計算時間不變，看起來此時計算幾乎不受 n 的影響。

這個基准測試告訴我們，使用迭代的解決方案 sum_of_n 做了更多的工作，因爲一些程序步驟被重復執行，這是它需要更長時間的原因。此外，迭代方案執行所需時間隨着 n 遞增。另外要意識到，如果在不同計算機上或者使用不用的編程語言運行類似函數，得到的結果也不同，你的電腦計算值可能不是 1341 納秒（我使用的電腦是聯想拯救者 R7000P，CPU 是 16 核的 AMD R7-4800H）。如果使用老舊的計算機，則需要更長時間才能執行完 sum_of_n2。

我們需要一個更好的方法來描述這些算法的執行時間。基准測試計算的是程序執行的實際時間，但它並不真正地提供給我們一個有用的度量，因爲執行時間取決於特定的機器，且毫秒，納秒間也涉及到數量級轉換。我們希望有一個獨立於所使用的程序或計算機的度量，這個度量能獨立地判斷算法，並且可以用於比較不同算法實現的效率，就像加速度這個度量值能明確指出速度每秒的改變值一樣。在算法分析領域，大 O 分析法就是一種很好度量方法。

2.3 大 O 分析法

要獨立於任何特定程序或計算機來表徵算法的效率（複雜度），重要的是要量化算法所需操作的數量。對於先前的求和算法，一個比較好的度量單位是對執行語句進行計數。在 sum_of_n 中，賦值語句的計數爲 1 (the_sum = 0 的次數)，the_sum += i 計數爲 n。

使用函數 T 表示總的次數：T(n)=1 + n 。參數 n 通常稱爲問題的規模，T(n) 是解決問題規模爲 n 的問題所花費的時間。在上面的求和函數中，使用 n 來表示問題大小是有意義的。我們可以說對 100,000 個整數求和比對 1000 個整數求和的規模大，因此所需時間也更長。我們的目標是表示出算法的執行時間是如何相對問題規模的大小而改變的。

將這種分析技術進一步擴展，確定操作步驟所有數量不如確定 T(n) 最主要的部分來得重要。換句話說，當問題規模變大時，T(n) 函數某些部分的分量會遠遠超過其他部分。函數的數量級表示隨着 n 增加而增加最快的那些部分。數量級通常用大 O 符號表示，寫作 O(f(n))，用於表示對計算中的實際步數的近似。函數 f(n) 表示 T(n) 最主要部分。

在上述示例中，T(n) = n + 1。當 n 變大時，常數 1 對於最終結果變得越來越不重要。如果我們找的是 T(n) 的近似值，可以刪除 1，運行時間就是 O(T(n)) = O(n + 1) = O(n)。要注意，1 對於 T(n) 肯定是重要的，但當 n 變大時，有沒有它 O(n) 近似都是准確的。比如對於 $T(n) = n^3 + 1$，n 爲 1 時，T(n) = 2，如果此時舍掉 1 就不合理，因爲這樣就相當於 F 掉了一半的運行時間。但是當 n 等於 10 時，T(n) = 1001，此時 1 已經不重要了，即便舍掉了，T(n) = 1000 依然是一個很准確的指標。

假設對於一些算法，確定的步數是 $T(n) = 5n^2 + 27n + 1005$。當 n 很小時，例如 1 或 2 ，常數 1005 似乎是函數的主要部分。然而，隨着 n 變大，n^2 這項變得越來越重要。事實上，當 n 很大時，其他兩項在最終結果中所起的作用變得不重要。當 n 變大時，爲了近似 T(n)，我們可以忽略其他項，只關注 $5n^2$ 。係數 5 也變得不重要。我們說 T(n) 具有的複雜度數量級爲 n^2，或者 $O(n^2)$ 。

但有時算法的複雜度取決於數據的確切值，而不是問題規模的大小。對於這類算法，需要根據最佳情況，最壞情況或平均情況來表徵它們的性能。最壞情況是指導致算法性能特別差的特定數據集。而相同的算法，不同數據集可能具有非常不同的性能。大多數情況下，算法執行效率處在最壞和最優兩個極端之間（平均情況）。對於程序員而言，重要的是瞭解這些區別，使它們不被某一個特定的情況誤導。

在學習算法時，一些常見的數量級函數將會反覆出現，見下表。爲了確定這些函數中哪個是最主要的部分，我們需要看到當 n 變大時它們相互關係如何。

表 2.1: 不同數量級的函數

T(n)	O(1)	O(logn)	O(n)	O(nlogn)	$O(n^2)$	$O(n^3)$	$O(2^n)$
性能	常數	對數	線性	線性對數	平方指數	立方指數	冪指數

下圖畫出了各種函數的增長情況，當 n 很小時，函數彼此間不能很好的定義。很難判斷哪個是主導的。隨着 n 的增長，就有一個很明確的關係，很容易看出它們之間的大小關係。注意在 $n = 10$ 時，2^n 會大於 n^3，圖中沒有畫出完整的情況。

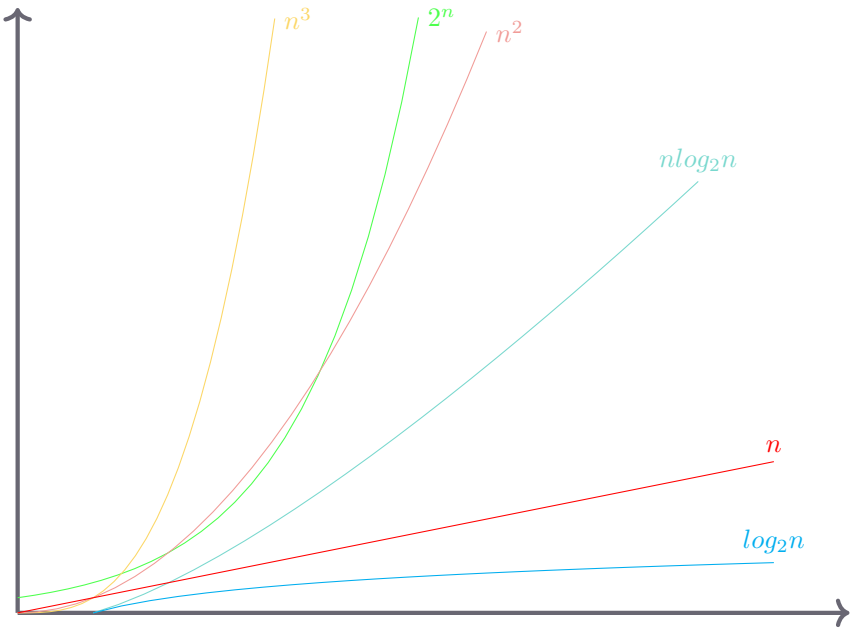


图 2.1: 複雜度曲線

通過上圖可以得出這些不同數量級的區別，在一般情況下（$n > 10$），存在 $O(2^n) > O(n^3) > O(n^2) > O(nlogn) > O(n) > O(logn) > O(1)$。這對於我們設計算法很有幫助，因爲對於每個算法，我們都能計算其複雜度，如果得到類似 $O(2^n)$ 複雜度的算法，我們知道這一定不實用，然後可以對其進行優化以取得更好的性能。

下面的代碼只做了些加減乘除，但我們可以用它來試試新學的算法複雜度分析。

```
fn main() {
    let a = 1;
    let b = 2;
    let c = 3;
    let n = 1000000;

    for i in 0..n {
        for j in 0..n {
            let x = i * i;
            let y = j * j;
            let z = i * j;
         }
    }

    for k in 0..n {
        let w = a*b + 45;
        let v = b*b;
    }

    let d = 33;
}
```

分析上述代碼的 T(n)，分配操作數 a，b，c，n 的時間爲常數 4。第二項是 $3n^2$，因爲嵌套迭代，有三個語句執行 n^2 次。第三項是 2n, 有兩個語句迭代執行了 n 次。最後，第四項是常數 1，表示最終賦值語句 d = 33。最後得出 $T(n) = 4 + 3n^2 + 2n + 1 = 3n^2 + 2n + 5$，查看指數，可以看到 n^2 項是最顯著的，因此這個代碼段是 O(n^2)。當 n 增大時，其他項及主項上的係數都可以忽略。

2.4 亂序字符串檢查

一個展示不同數量級複雜度的例子是亂序字符串檢查算法。亂序字符串是指一個字符串只是另一個字符串的重新排列。例如，'heart' 和'earth' 就是亂序字符串，'rust' 和'trus' 也是。爲簡單起見，假設所討論的兩個字符串具有相等的長度，且都由 26 個小寫字母集合組成。我們的目標是寫一個函數，它將兩個字符串做爲參數並返回它們是不是亂序字符串。

2.4.1 窮舉法

解決亂序字符串最笨的方法是暴力窮舉法，把每種情況都列舉出來。首先可以生成 s1 的所有亂序字符串列表，然後查看這些列表裏是否有一個和 s2 相同。這種方法特別費資源，既

費時間又費內存。當 s1 生成所有可能的字符串時，第一個位置有 n 種可能，第二個位置有 n-1 種，第三個位置有 n-3 種...。總數爲 $n \times (n-1) \times (n-2)...3 \times 2 \times 1$，即 n!。雖然一些字符串可能是重復的，程序也不可能提前知道，所以會生成 n! 個字符串。

如果 s1 有 20 個字符長，則將有 20! = 2,432,902,008,176,640,000 個字符串產生。如果每秒處理一種可能的字符串，那麼需要 77,146,816,596 年才能過完整個列表。事實證明，n! 比 n^2 增長還快，所以暴力窮舉法這種解決方案是不行的，在任何學習及真正的軟件項目裏都不可能使用這種方法，當然，瞭解它的存在並盡力避免這種情況卻是很有必要的。

2.4.2 檢查法

亂序字符串問題的第二種解法是檢查第一個字符串中的字符是否出現在第二個字符串中。如果檢測到每個字符都存在，那這兩個字符串一定是亂序。可以通過用' ' 替換字符來了解一個字符是否完成檢查。詳細代碼請參見源碼中 anagram_solution2.rs。

```
1  // anagram_solution2.rs
2
3  fn anagram_solution2(s1: &str, s2: &str) -> bool {
4      // 字符串長度不同，一定不是亂序字符串
5      if s1.len() != s2.len() { return false; }
6
7      // s1 和 s2 中的字符分別加入 alist, blist
8      let mut alist = Vec::new();
9      let mut blist = Vec::new();
10     for c in s1.chars() { alist.push(c); }
11     for c in s2.chars() { blist.push(c); }
12
13     let mut pos1: usize = 0; // pos1、pos2 索引字符
14     let mut ok = true; // 亂序字符串標示、控制循環進程
15     while pos1 < s1.len() && ok {
16         let mut pos2: usize = 0;
17
18         // found 標示字符是否在 s2 中
19         let mut found = false;
20         while pos2 < blist.len() && !found {
21             if alist[pos1] == blist[pos2] {
22                 found = true;
23             } else {
24                 pos2 += 1;
25             }
26         }
```

```
27
28          // 某字符存在於 s2 中，將其替換成 ' ' 避免再次比較
29          if found {
30              blist[pos2] = ' ';
31          } else {
32              ok = false;
33          }
34
35          // 處理 s1 中的下一個字符
36          pos1 += 1;
37      }
38
39      ok
40  }
41
42  fn main() {
43      let s1 = "rust";
44      let s2 = "trus";
45      let result: bool = anagram_solution2(&s1, &s2);
46      println!("s1 and s2 is anagram: {result}");
47  }
```

分析這個算法，注意到 s1 的每個字符都會在 s2 中進行最多 n 次迭代檢查。blist 中的 n 個位置將被訪問一次以匹配來自 s1 的字符。總的訪問次數可以寫成 1 到 n 的整數和。

$$\begin{aligned}\sum_{i=1}^{n} i &= \frac{n(n+1)}{2} \\ &= \frac{1}{2}n^2 + \frac{1}{2}n\end{aligned} \tag{2.1}$$

當 n 變大，n^2 這項占據主導，1/2 可以忽略。所以這個算法複雜度爲 $O(n^2)$。暴力法是 n!，而檢查法一下就降到了 $O(n^2)$

2.4.3 排序和比較法

亂序字符串的另一個解決方案是利用這樣一個事實：即使 s1，s2 不同，它們都是由完全相同的字符組成的。所以可以按照字母順序從 a 到 z 排列每個字符串，如果排列後的兩個字符串相同，那這兩個字符串就是亂序字符串。

```
1  // anagram_solution3.rs
2
3  fn anagram_solution3(s1: &str, s2: &str) -> bool {
```

```
4       if s1.len() != s2.len() { return false; }
5
6       // s1 和 s2 中的字符分别加入 alist、blist 并排序
7       let mut alist = Vec::new();
8       let mut blist = Vec::new();
9       for c in s1.chars() { alist.push(c); }
10      for c in s2.chars() { blist.push(c); }
11      alist.sort(); blist.sort();
12
13      // 逐個比較排序的集合，任何字符不匹配就退出循環
14      let mut pos: usize = 0;
15      let mut matched = true;
16      while pos < alist.len() && matched {
17          if alist[pos] == blist[pos] {
18              pos += 1;
19          } else {
20              matched = false;
21          }
22      }
23
24      matched
25  }
26
27  fn main() {
28      let s1 = "rust";
29      let s2 = "trus";
30      let result: bool = anagram_solution3(s1, s2);
31      println!("s1 and s2 is anagram: {result}");
32  }
```

乍一看，只有一個 while 循環，所以應該是 O(n)。可調用排序函數 sort() 也是有成本的。其複雜度通常是 $O(n^2)$ 或 $O(nlogn)$，所以該算法複雜度跟排序算法屬於同樣數量級。

2.4.4 計數和比較法

上面的方法中，總是要創建 alist 和 blist，這非常費內存。當 s1 和 s2 比較短時，alist 和 blist 還算合適，但若是 s1 和 s2 達到百萬字符呢？這時 alist 和 blist 就非常大。通過分析可知，s1 和 s2 只含 26 個小寫字母，所以用兩個長度爲 26 的列表，統計各個字符出現的頻次就可以了。每遇到一個字符，就增加該字符在列表對應位置的計數。最後如果兩個列表

計數一樣，則字符串爲亂序字符串。

```rust
// anagram_solution4.rs

fn anagram_solution4(s1: &str, s2: &str) -> bool {
    if s1.len() != s2.len() { return false; }

    // 大小爲 26 的集合，用於將字符映射爲 ASCII 值
    let mut c1 = [0; 26];
    let mut c2 = [0; 26];
    for c in s1.chars() {
        let pos = (c as usize) - 97; // 97 爲 a 的 ASCII 值
        c1[pos] += 1;
    }
    for c in s2.chars() {
        let pos = (c as usize) - 97;
        c2[pos] += 1;
    }

    // 逐個比較 ascii 值
    let mut pos = 0;
    let mut ok = true;
    while pos < 26 && ok {
        if c1[pos] == c2[pos] {
            pos += 1;
        } else {
            ok = false;
        }
    }

    ok
}

fn main() {
    let s1 = "rust";
    let s2 = "trus";
    let result: bool = anagram_solution4(s1, s2);
    println!("s1 and s2 is anagram: {result}");
}
```

這個方案也存在多個迭代，但和前面的解法不一樣，首先迭代不是嵌套的，其次第三個迭代，比較兩個計數列表，只需要 26 次，因爲只有 26 個小寫字母。所以 $T(n) = 2n + 26$，即 $O(n)$。這是一個線性複雜度的算法，其空間和時間複雜度都比較優秀。當然，s1 和 s2 也可能比較短，用不到 26 個字符，這樣該算法也犧牲了部分存儲空間。很多情況下，你需要在空間和時間之間做出權衡，要思考你的算法應對的真實場景，然後再決定採用哪種算法。

2.5 Rust 數據結構的性能

2.5.1 標量和複合類型

本節的目標是探討 Rust 內置的各種基本數據類性的大 O 性能，重要的是瞭解這些數據結構的效率，因爲它們是 Rust 中最基礎和最核心的模塊，其他所有複雜的數據結構都由它們構建而成。

在 Rust 中，每個值都屬於某一數據類型，這告訴 Rust 編譯器它被指定爲何種數據，以便明確數據存儲和操作方式。Rust 裏面有兩大類基礎數據類型：標量和複合類型。

標量類型代表一個單獨的值，複合類型是標量類型的組合。Rust 中有四種基本的標量類型：整型、浮點型、布爾型、字符型；有兩種複合類型：元組、數組。標量類型都是最基本的和內存結合最緊密的原生類型，運算效率非常高，可以視爲 O(1)，而複合類型則複雜一些，複雜度隨其數據規模而變化。

```
1  fn main() {
2      let a: i8 = -2;
3      let b: f32 = 2.34;
4      let c: bool = true;
5      let d: char = 'a';
6
7      let x: (i32, f64, u8) = (200, 5.32, 1);
8      let xi32 = x.0;
9      let xf64 = x.1;
10     let xu8  = x.2;
11 }
```

元組是將多個各種類型的值組合成一個複合類型的數據結構。元組長度固定，一旦聲明，其長度不能增大或縮小。元組的索引從 0 開始，直接用 “.” 號獲取值。

數組一旦聲明，長度也不能增減，但與元組不同的是，數組中每個元素的類型必須相同。數組非常有用，基於數組的切片其實更靈活，所以也更常用。

```
1  fn main() {
2      let months = ["January", "February", "March", "April",
3                    "May", "June", "July", "August", "September",
4                    "October", "November", "December"];
```

```
5       let first_month = months[0]
6       let halfyear = &months[..6];
7
8       let mut monthsv = Vec::new();
9       for month in months { monthsv.push(month); }
10  }
```

Rust 裏的其他數據類型都是由標量和複合類型構成的集合類型，如 Vec、HashMap 等。Vec 類型是標准庫提供的一個允許增加和縮小長度的類似數組的集合類型。能用數組的地方都可用 Vec，所以當不知道該用哪個時，用 Vec 不算錯，而且還更有擴展性。

2.5.2 集合類型

Rust 的集合類型是基於標量和複合類型構造的，其中又分爲線性和非線性兩類。線性的集合類型有：String、Vec、VecDeque、LinkedList，而非線性集合類型的有：HashMap、BTreeMap、HashSet、BTreeSet、BinaryHeap。這些線性和非線性集合類型多涉及到索引、增、刪操作，對應的複雜度多是 O(1)、O(n) 等。具體性能如以下兩表。由表可見 Rust 實現的集合數據類型都是非常高效的，複雜度最高也就是 O(n)。

表 2.2: 線性集合類型的性能

Type	get	insert	remove	append	split_off
Vec	O(1)	O(n-i)	O(n-i)	O(m)	O(n-i)
VecDeque	O(1)	O(min(i,n-i))	O(min(i,n-i))	O(m)	O(min(i,n-i))
LinkedList	O(min(i,n-i))	O(min(i,n-i))	O(min(i,n-i))	O(1)	O(min(i,n-i))

表 2.3: 非線性集合類型的性能

Type	get	insert	remove	predecessor	append
HashMap	O(1)	O(1)	O(1)	N/A	N/A
HashSet	O(1)	O(1)	O(1)	N/A	N/A
BTreeMap	O(logn)	O(logn)	O(logn)	O(logn)	O(n+m)
BTreeSet	O(logn)	O(logn)	O(logn)	O(logn)	O(n+m)
Type	push	pop	peek	peek_mut	append
BinaryHeap	O(1)	O(1ogn)	O(1)	O(1)	O(n+m)

Rust 實現的 String 底層基於數組，所以 String 同數組一樣不可更改。若要使用字符串中部分字符則可使用 &str，&str 實際是基於 String 數組的切片，便於索引。Vec 則類似於 Python 中的列表，基於分配在堆上的數組。VecDeque 擴展了 Vec，支持在序列兩端插入數據，所以是一個雙端隊列。LinkedList 是鏈表，當需要一個未知大小的 Vec 時可以採用。

Rust 實現的 HashMap 類似於 Python 的字典，BTreeMap 則是 B 樹，其節點上包含數據和指針，多用於實現數據庫，文件系統等需要存儲內容的地方。Rust 實現的 HashSet 和 BTreeSet 類似於 Python 的集合，都用於記錄出現過的值。HashSet 底層採用的是 HashMap，而 BTreeSet 底層則採用的 BTreeMap。BinaryHeap 類似優先隊列，存儲一堆元素，可在任何時候提取出最大的值。

2.6 總結

本章學習了算法複雜度分析的大 O 分析法：計算代碼執行的步數，並取其最大數量級。接着學習了 Rust 實現的基本數據類型和集合數據類型的複雜度，通過對比學習，可知 Rust 內置的標量、複合以及集合數據類型都非常高效，基於這些集合類型來實現自定義的數據結構也就更容易做到高效實用。

Chapter 3

基本數據結構

3.1 本章目標

理解抽象數據類型 Vec、棧、隊列、雙端隊列
能夠使用 Rust 實現堆棧、隊列、雙端隊列
瞭解基本線性數據結構實現的性能
瞭解前綴、中綴和後綴表達式格式
使用棧來實現後綴表達式並計算值
使用棧將中綴表達式轉換爲後綴表達式
能夠識別問題該使用棧、隊列還是雙端隊列
能夠使用節點和引用將抽象數據類型實現爲鏈表
能夠比較自己實現鏈表與 Rust 自帶的 Vec 的性能

3.2 線性數據結構

數組、棧、隊列、雙端隊列這一類數據結構都是保存數據的容器，數據項之間的順序由添加或刪除的順序決定，一旦數據項被添加，它相對於前後元素一直保持位置不變，諸如此類的數據結構被稱爲線性數據結構。線性數據結構有兩端，被稱爲“左”和“右”，某些情況也稱爲“前”和“後”，當然，也可以稱爲頂部和底部，具體的名字不重要，重要的是這種命名展現出的位置關係表明了數據組織方式就是線性的。這種線性特性和內存緊密相關，因爲內存便是一種線性硬件，由此也可以看出軟件和硬件是如何關聯在一起的。

區分不同線性數據結構的方法是看其添加和移除數據項的方式，特別是添加和移除項的位置。例如一些數據結構只允許從一端添加項，另一些則允許從另一端移除項，還有的數據結構允許兩端操作數據項。這些變種及其組合形式產生了許多計算機科學領域最有用的數據結構，它們出現在各種算法中，並用於執行各種實際且重要的任務。

3.3 棧

棧就是一種特別有用的數據結構，可用於函數調用、網頁數據記錄等。棧是一個項的有序集合，其中添加移除新項總發生在同一端，這一端稱爲頂部，與之相對的端稱爲底部。棧的底部很重要，因爲在棧中靠近底部的項是存儲時間最長的，最近添加的項是會最先被移除的。這種排序原則有時被稱爲後進先出（Last In Fisrt Out，LIFO）或者先進後出（Fisrt In Last Out, FILO），所以較新的項靠近頂部，較舊的項靠近底部。

棧的例子很常見，工地堆的磚，桌上的書堆，餐廳堆疊的盤子都是棧的物理模型。要拿到最下面的磚、書、盤子，只有先把上面的一個個拿走。下圖是棧的示意圖，其中保存着一些概念名稱（計算機是量子力學理論的衍生物）。

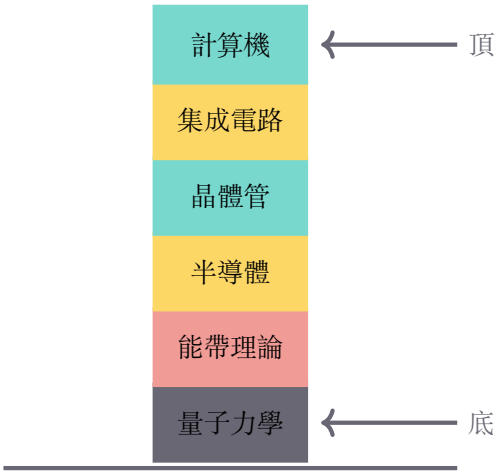


图 3.1: 棧

要理解棧的作用，最好的方式是觀察棧的形成和清空。假設從一個乾淨的桌面開始，現在把書一本本疊起來，你就在構造一個棧。考慮下移除一本書會發生什麼。移除的順序跟剛剛放置的順序相反。棧之所以重要是因爲它能反轉項的順序，插入跟刪除順序相反。下圖展現了數據對象創建和刪除的過程，注意觀察數據的順序。

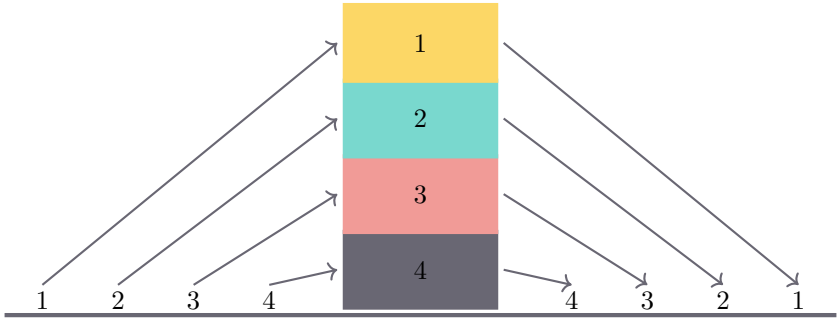


图 3.2: 棧逆反數據

這種反轉的屬性特別有用，你可以想到使用計算機的時候所碰到的例子。例如，每個瀏覽器都有返回按鈕，當你瀏覽網頁時，這些網頁就被放在一個棧中，你當前查看的網頁始終在頂部，第一次查看的網頁在底部。如果你按返回鍵，將按相反的順序回到剛纔的頁面。由這個例子也可看出數據結構的重要性，對某些功能，選好數據結構，能讓事情簡單不少。

3.3.1 棧的抽象數據類型

棧的抽象數據類型由其結構和操作定義。如上所述，棧被構造爲項的有序集合，其中項被添加和移除的位置稱爲頂部。棧的一些操作如下。

new() 創建一個空棧，它不需要參數，返回一個空棧。

push(item) 將數據項 item 添加到棧頂，它需要 item 做參數，不返回任何內容。

pop() 從棧中刪除頂部數據項，它不需要參數，返回數據項，棧被修改。

peek() 從棧返回頂部數據項，但不會刪除它，不需要參數，不修改棧。

is_empty() 測試棧是否爲空，不需要參數，返回布爾值。

size() 返回棧中數據項的數量，不需要參數，返回一個 usize 型整數。

假設 s 是已創建的空棧，此處用 [] 表示，下表展示了棧操作後的結果，棧頂在右邊。

表 3.1: 棧操作及結果

棧操作	棧當前值	操作返回值
s.is_empty()	[]	true
s.push(1)	[1]	
s.push(2)	[1,2]	
s.peek()	[1,2]	2
s.push(3)	[1,2,3]	
s.size()	[1,2,3]	3
s.is_empty()	[1,2,3]	false
s.push(4)	[1,2,3,4]	
s.push(5)	[1,2,3,4,5]	
s.size()	[1,2,3,4,5]	5
s.pop()	[1,2,3,4]	5
s.pop()	[1,2,3]	4
s.size()	[1,2,3]	3

3.3.2 Rust 實現棧

前文已定義了棧的抽象數據類型，現在使用 Rust 來實現棧，抽象數據類型的實現又稱爲數據結構。在 Rust 中，抽象數據類型的實現多選擇創建新的結構體 struct，棧操作實現爲結構體的函數。此外，爲了實現作爲元素集合的棧，使用由 Rust 自帶的基本數據結構對實現棧及其操作大有幫助。

這裏使用 Vec 這種集合容器來作爲棧的底層實現，因爲 Rust 中的 Vec 提供了有序集合機制和一組操作方法，只需要選定 Vec 的哪一端是棧頂部就可以實現其他操作了。以下棧實現假定 Vec 的尾部將保存棧的頂部元素，隨着棧增長，新項將被添加到 Vec 末尾，因爲不知道插入數據類型，所以採用了泛型數據類型 T。

```
1  // stack.rs
2
3  #[derive(Debug)]
4  struct Stack<T> {
5      top: usize,      // 棧頂
6      data: Vec<T>,    // 棧數據容器
7  }
8
9  impl<T> Stack<T> {
10     fn new() -> Self {
11         Stack {
12             top: 0,
13             data: Vec::new()
14         }
15     }
16
17     fn push(&mut self, val: T) {
18         self.data.push(val); // 數據保存在 Vec 末尾
19         self.top += 1;
20     }
21
22     fn pop(&mut self) -> Option<T> {
23         if self.top == 0 { return None; }
24         self.top -= 1; // 棧頂 - 1 後再彈出數據
25         self.data.pop()
26     }
27
28     fn peek(&self) -> Option<&T> { // 數據不能移動，只能返回引用
29         if self.top == 0 { return None; }
30         self.data.get(self.top - 1)
31     }
32
33     fn is_empty(&self) -> bool {
34         0 == self.top
```

```
35      }
36
37      fn size(&self) -> usize {
38          self.top // 栈頂恰好就是栈中元素個數
39      }
40  }
41
42  fn main() {
43      let mut s = Stack::new();
44      s.push(1); s.push(2); s.push(4);
45      println!("top {:?}, size {}",s.peek().unwrap(), s.size());
46      println!("pop {:?}, size {}",s.pop().unwrap(), s.size());
47      println!("is_empty:{}, stack:{:?}", s.is_empty(), s);
48  }
```

3.3.3 括號匹配

上面實現了棧數據結構，下面利用棧來解決真正的計算機問題。第一個是括號匹配問題，任何人都見過如下這種計算值的算術表達式。

$$(5+6)\times(7+8)/(4+3)$$

Lisp 語言是一種非常依賴括號的語言，比如下面這個 multiply 函數。

```
1  (defun multiply(n)
2      (* n n))
```

這裏關注點不在數字而在括號，因爲括號更改了操作優先級，限定了語言的語義，非常重要。若括號不完整，那麼整個式子就是錯的。這對於人來說是再好懂不過了，可是計算機又如何知道呢？可見，計算機必然檢測了括號是否匹配，並根據情況報錯。

上面這兩個例子中，括號都必須以成對匹配的形式出現。括號匹配意味着每個開始符號具有相應的結束符號，並且括號正確嵌套，這樣計算機才能正確處理。

考慮下面正確匹配的括號字符串:

(()()()())

((((()))

(()((())()))

以及這些不匹配的括號:

(((((((())

()))

(()()((

這些括號表達式省去了包含的具體值，只保留了括號自身，其實程序中經常出現這種括號及嵌套的情況。

區分括號匹配對於計算機程序來說是十分重要的，只有這樣才能決定下一步操作。具有挑戰的是如何編寫一個算法能夠從左到右讀取一串符號，並決定括號是否平衡。要解決這個問題，需要對括號及其匹配有比較深入地瞭解。從左到右處理符號時，最近的左開始括號‘(’必須與下一個右關閉符號‘)’相匹配（如圖3.3)。此外，處理的第一個左開始括號必須等待直到其匹配最後一個右關閉括號。結束括號以相反的順序匹配開始括號，從內到外匹配，這是一個可以用棧來解決的問題。

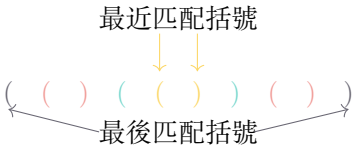


图 3.3: 括號匹配

一旦採用棧來保存括號，則算法的具體實現就很簡單了，因爲棧的操作無非就是出入棧和判斷而已。從空棧開始，從左到右處理括號字符串。如果一個符號是一個左開始符號，將其入棧，如果是結束符號，則彈出棧頂元素，並開始匹配這兩個符號。如果恰好是左右匹配的，那麼就繼續處理下一個括號，直到字符串處理完成。最後，當所有符號都被處理後，棧應該是空的。只要不爲空，就說明有括號不匹配。下面是 Rust 實現的括號匹配程序。

```
1  // par_checker1.rs
2
3  fn par_checker1(par: &str) -> bool {
4      let mut char_list = Vec::new();
5      for c in par.chars() {
6          char_list.push(c);
7      }
8
9      let mut index = 0;
10     let mut balance = true; // 括號是否匹配(平衡)標示
11     let mut stack = Stack::new(); // 使用前面實現的栈
12     while index < char_list.len() && balance {
13         let c = char_list[index];
14
15         if '(' == c { // 如果爲開符號，入栈
16             stack.push(c);
17         } else { // 如果爲閉符號，判斷栈是否爲空
18             if stack.is_empty() {
```

```
19                  balance = false; // 爲空則不平衡
20              } else {
21                  let _r = stack.pop();
22              }
23          }
24          index += 1;
25      }
26
27      // 平衡且栈爲空，則括號表達式是匹配的
28      balance && stack.is_empty()
29  }
30
31  fn main() {
32      let sa = "(()())";
33      let sb = "((((()";
34      let res1 = par_checker1(sa);
35      let res2 = par_checker1(sb);
36      println!("sa balanced:{res1}, sb balanced:{res2}");
37  }
```

當然，括號不僅限於 ()，還有 {} 及 []，下文將這三種都稱爲括號。上面顯示的匹配括號問題非常簡單，只有 (和) 這兩個符號，但其實常用的括號有三種，分別是 ()、[]、{}。匹配和嵌套不同種類的左開始和右結束括號的情況經常發生。例如，在 Rust 中，方括號 [] 用於索引，花括號 {} 用於格式化輸出，() 用於函數參數，元組，數學表達式等。只要每個符號都能保持自己的左開始和右結束關係，就可以混合嵌套符號。

{ { ([ ] [ ]) } ( ) }
[[{ { (( )) } }]]
[ ] [ ] [ ] ( ) { }

上面這些括號都匹配，每個開始符號都有對應的結束符號，而且符號類型也匹配。相反下面這些字符串符號就不匹配。

(} [ ]
((( )]))
[{ ( )]

前面的那個括號檢查程序 par_checker1 只能檢測 ()，要處理三種括號，需要對其進行擴展。算法流程依舊不變，每個左開始括號被壓入棧中，等待匹配的右結束括號出現。出現結束括號時，此時要做的是檢查括號的類型是否匹配。如果兩個括號不匹配，則字符串不匹配。如果整個字符串都被處理完並且棧爲空，則字符串括號匹配。

爲了檢查括號類型是否匹配，我們新增了一個符號類型檢測函數 par_match()，它可以同時檢測常用的三種括號。

```rust
// par_checker2.rs

// 同時檢測多種開閉符號是否匹配
fn par_match(open: char, close: char) -> bool {
    let opens = "([{";
    let closers = ")]}";
    opens.find(open) == closers.find(close)
}

fn par_checker2(par: &str) -> bool {
    let mut char_list = Vec::new();
    for c in par.chars() {
        char_list.push(c);
    }

    let mut index = 0;
    let mut balance = true;
    let mut stack = Stack::new();
    while index < char_list.len() && balance {
        let c = char_list[index];

        // 同時判斷三種開符號
        if '(' == c || '[' == c || '{' == c {
            stack.push(c);
        } else {
            if stack.is_empty() {
                balance = false;
            } else {
                // 比較當前括號和棧頂括號是否匹配
                let top = stack.pop().unwrap();
                if !par_match(top, c) {
                    balance = false;
                }
            }
        }
        index += 1;
    }

```

```
39      balance && stack.is_empty()
40  }
41
42  fn main() {
43      let sa = "(){}[]";
44      let sb = "(){)[}";
45      let res1 = par_checker2(sa);
46      let res2 = par_checker2(sb);
47      println!("sa balanced:{res1}, sb balanced:{res2}");
48  }
```

現在我們的代碼能處理多種括號匹配的問題，但是如果出現像下面這種字符串，其中含有其他字符，那麼上面的程序就又不能處理了。

(a+b)(c*d)func()

這個問題看起來複雜，因爲似乎要處理各種字符。但實際上，問題還是括號匹配檢測，所以非括號不用處理，直接跳過。程序處理字符時，上面的字符串中非括號自動忽略，只剩下括號，相當於字符串：()()()。可見問題和原來一樣，只需修改部分代碼就能檢測包含任意字符的字符串是否匹配。下面的代碼是修改 par_checker2.rs 後得到的 par_checker3.rs。

```
1   // par_checker3.rs
2
3   fn par_checker3(par: &str) -> bool {
4       let mut char_list = Vec::new();
5       for c in par.chars() {
6           char_list.push(c);
7       }
8
9       let mut index = 0;
10      let mut balance = true;
11      let mut stack = Stack::new();
12      while index < char_list.len() && balance {
13          let c = char_list[index];
14
15          // 開符號入棧
16          if '(' == c || '[' == c || '{' == c {
17              stack.push(c);
18          }
19
20          // 閉符號則判斷是否平衡
```

```
21          if ')' == c || ']' == c || '}' == c {
22              if stack.is_empty() {
23                  balance = false;
24              } else {
25                  let top = stack.pop().unwrap();
26                  if !par_match(top, c) {
27                      balance = false;
28                  }
29              }
30          }
31
32          // 非括號直接跳過
33          index += 1;
34      }
35
36      balance && stack.is_empty()
37  }
38
39  fn main() {
40      let sa = "(2+3){func}[abc]";
41      let sb = "(2+3)*(3-1";
42      let res1 = par_checker3(sa);
43      let res2 = par_checker3(sb);
44      println!("sa balanced:{res1}, sb balanced:{res2}");
45  }
```

這個例子表明棧是重要的數據結構，任何嵌套符號匹配問題都可以用棧處理。

3.3.4 進制轉換

對你來說二進制應該很熟悉。二進制是計算機世界的底座，是計算機世界底層真正通用的數據格式，因爲存儲在計算機內的所有值都是以 0 和 1 的電壓形式存儲的。如果沒有能力在二進制數和普通字符之間轉換，與計算機之間的交互將非常困難。整數值是常見的數據形式，一直用於計算機程序和計算。我們在數學課上學習過，當然是學的十進制表示。十進制 233(10) 以及對應的二進制表示 11101001(2) 分別解釋爲：

$$\begin{aligned} 2\times 10^2+3\times 10^1+3\times 10^0 &= 233 \\ 1\times 2^7+1\times 2^6+1\times 2^5+0\times 2^4+1\times 2^3+0\times 2^2+0\times 2^1+1\times 2^0 &= 233 \end{aligned} \tag{3.1}$$

將整數值轉換爲二進制最簡單的方法是“除 2”算法，它用棧來跟二進制結果的數字。除 2 算法假定從大於 0 的整數開始。不斷迭代地將十進制數除以 2，並跟蹤余數。第一個除以 2 的余數說明了這個值是偶數還是奇數。偶數的余數爲 0，奇數余數爲 1。在迭代除 2 的過程中將這些余數記錄下來，就得到了二進制數字序列，第一個余數實際上是序列中的最後一個數字。見下圖，數字是反轉的，第一次除法得到的余數放在棧底，出棧所有數字得到的表示就是原十進制數字的二進制表示。很明顯，棧是解決這個問題的關鍵。

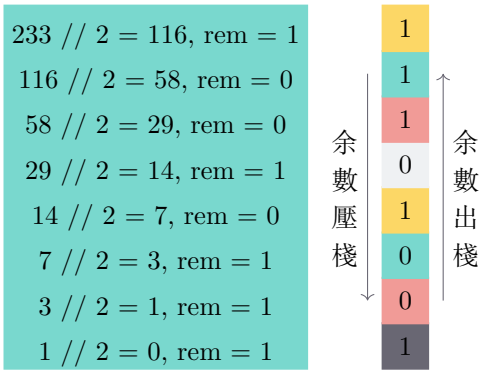


图 3.4: 除二法

下面的 Rust 代碼實現了除 2 算法，函數 divideBy2 傳入一個十進制的參數，並重復除以 2。第 9 行使用內置的模運算符% 來提取余數，第 10 行將余數加入棧中。當除到 0 後程序結束並構造返回一個二進制字符串。

```
1  // divide_by_two.rs
2
3  fn divide_by_two(mut dec_num: u32) -> String {
4      // 用棧來保存余數 rem
5      let mut rem_stack = Stack::new();
6
7      // 余數 rem 入棧
8      while dec_num > 0 {
9          let rem = dec_num % 2;
10         rem_stack.push(rem);
11         dec_num /= 2;
12     }
13
14     // 棧中元素出棧組成字符串
15     let mut bin_str = "".to_string();
16     while !rem_stack.is_empty() {
17         let rem = rem_stack.pop().unwrap().to_string();
```

```
18          bin_str += &rem;
19      }
20
21      bin_str
22  }
23
24  fn main() {
25      let bin_str: String = divide_by_two(10);
26      println!("10 is b{bin_str}");
27  }
```

這個用於十進制到二進制轉換的算法可以很容易地擴展到執行任何進制數間的轉換。在計算機科學中，通常會使用不同的進制數，其中最常見的是二進制，八進制和十六進制。十進制 233 對應的八進制爲 351(8)，十六進制爲 e9(16)。

可以修改 divideBy2 函數，使它不僅能接受十進制參數，還能接受預定轉換的基數。除 2 的概念被替換成更通用的除基數。在下面展示的是一個名爲 base_converter 函數。採用十進制數和 2 到 16 之間的任何基數作爲參數。余數部分仍然入棧，直到被轉換的值爲 0。有一個問題是，超過 10 的進制，比如 16 進制，它的余數必然會出現大於 10 的數，爲了簡化字符顯示，最好將大於 10 的余數顯示爲單個字符，我們選擇 A-F 分別表示 10-15，當然也可以用小寫形式的 a-f，或者其他的字符序列如 u-z，U-Z。

```
1  // base_converter.rs
2
3  fn base_converter(mut dec_num: u32, base: u32) -> String {
4      // digits 對應各種余數的字符形式，尤其是 10 - 15
5      let digits = ['0', '1', '2', '3', '4', '5', '6', '7',
6                    '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'];
7      let mut rem_stack = Stack::new();
8
9      // 余數入棧
10     while dec_num > 0 {
11         let rem = dec_num % base;
12         rem_stack.push(rem);
13         dec_num /= base;
14     }
15
16     // 余數出棧，取對應字符來拼接成字符串
17     let mut base_str = "".to_string();
18     while !rem_stack.is_empty() {
```

```
19          let rem = rem_stack.pop().unwrap() as usize;
20          base_str += &digits[rem].to_string();
21      }
22
23      base_str
24  }
25
26  fn main() {
27      let bin_str: String = base_converter(10, 2);
28      let hex_str: String = base_converter(43, 16);
29      println!("10 is b{bin_str}, 43 is x{hex_str}");
30  }
```

3.3.5 前中後綴表達式

編寫一個算術表達式，如 B * C ，表達式形式使你能正確理解它。在這種情況下，你知道是 B 乘 C，操作符是乘法運算符 * 。這種類型的符號稱爲中綴表達式，因爲運算符處於它處理的兩個操作數 A 和 B 中間，而且讀法 “A 乘 B” 和表達式的順序一致，自然好理解。

再看另外一個中綴表達式的例子，A + B * C，此時運算符 + 和 * 處於操作數之間。這裏的問題是，如何區分運算符分別作用於哪個運算數上呢？到底是 + 作用於 A 和 B，還是 * 作用於 B 和 C？當然，你肯定覺得這很簡單，當然是 * 作用於 B 和 C，然後結果再和 A 相加。這是因爲你知道每個運算符都有優先級，優先級較高的運算符在優先級較低的運算符之前使用。唯一改變順序的是括號的存在，算術運算符的優先順序是將乘法和除法置於加法和減法之前。如果出現具有相等優先級的兩個運算符，則使用從左到右的順序排序或關聯。

任何學過基礎數學知識的都知道這些。對於中綴表達式 A + B * C，用人腦算的時候，你的眼睛會自動移到後面兩個數上，最後再計算加法。實際上，你可能沒意識到，自己的大腦已經添加了括號並劃分好了計算順序：(A + (B * C))。

可是，計算機並沒有知識，它只能按照規則，按照順序來處理這種表達式。而我們平時使用計算機計算時並沒有出錯，說明計算機內部一定有某種規則（算法）使得它能正確計算。

藉助上面的思路，似乎保證計算機不會對操作順序產生混淆的方法就是創建一個完全括號表達式，這種類型的表達式對每個運算符都使用一對括號。括號指示着操作的順序。可問題是，計算機是從左到右處理數據，類似 (A + (B * C)) 這種完全括號表達式，計算機如何跳到內部括號計算乘法然後再跳到外部括號計算加法呢？人腦能跳着計算是因爲人有智能，知道判斷，而計算機是死腦筋，對人看起來簡單的任務，直接讓它處理也是非常困難的。

所以計算機採用的一定是某種不同於人腦計算的模式。完全括號表達式將操作符和操作數混合在一起，這種模式對計算機很困難，所以可以考慮將操作符號移動到操作數外面，將操作符和操作數分開，這樣計算機拿到操作符再去取操作數，計算的結果就作爲當前值，再參與後面的運算，直到完成整個運算表達式的計算。

可以將中綴表達式 A + B 的 “+” 號移動出來，既可以放前面也可以放後面，所以得到的分別是 + A B 和 A B + 這樣看起來有點兒怪的表達式。這兩種不同的表達式分別稱爲前綴表達式和後綴表達式，同中綴表達式可以區分開來。前綴表達式要求所有運算符在處理的兩個操作數之前，後綴表達式則要求其操作符在相應的操作數之後，下面是更多這類表達式的例子。

表 3.2: 前中後綴表達式

中綴表達式	前綴表達式	後綴表達式
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +
(A + B) * C	* + A B C	A B + C *
A + B - C	- + A B C	A B + C -
A * B + C	+ * A B C	A B * C +
A * B / C	/ * A B C	A B * C /

可以看到，這個表達式還挺複雜的，尤其加不加括號，得到表達式很不一樣。對於這種表達式需要讀者習慣，要能分析並知道正確的寫法。

注意到上面第三個中綴表達式裏明明有括號，但前綴和後綴表達式中並沒有括號。這是因爲前綴和後綴表達式已經將計算邏輯表達得很明確了，沒有模棱兩可的地方，計算機按照這種表達式計算不會出錯。只有中綴才需要括號，前綴和後綴表達式的操作順序完全由操作符的順序決定，所以不用括號。下表是一些更復雜的表達式。

表 3.3: 複雜的前中後綴表達式

中綴表達式	前綴表達式	後綴表達式
A + B * C - D	- + A * B C D	A B C * + D -
A * B - C / D	- * A B / C D	A B * C D / -
A + B + C + D	+ + + A B C D	A B + C + D +
(A + B) * (C - D)	* + A B - C D	A B + C D - *

有了前綴和後綴表達式，計算看起來更復雜了。比如 A + B * C 的前綴表達式 + A * B C ，這怎麼計算呢? A + B * C 要先算 B * C 再計算加法，可是 + A * B C 的乘號還是在內部，仍舊無法計算。要是能將乘號和加號顛倒順序就好了。前面我們學習過棧具有顛倒順序的功能，所以可以採用兩個棧，一個保存操作符號，一個保存操作數，直接按照從左到右的順序將其分別入棧，結果如圖（3.5）。

計算時先將操作符號出棧，然後將兩個操作數出棧，此時用操作符計算這兩個操作數，結果再入棧，如圖（3.6）。接着再重複這個計算步驟，直到操作符號棧空，此時彈出操作數棧頂數據，這個值就是整個表達式的計算結果。可以看到，這個計算和完全括號表達式計算是一樣的，而且計算機不用處理括號，只用出入棧，非常高效。

若是後綴表達式，也用棧來計算，且只用一個棧。比如 A + B * C 的後綴表達式 A B

C * +，先將 A B C 入棧，接着發現 * 號，彈出兩個操作數 B C，計算得到結果 BC，再將其入棧，接着遇到 + 號，彈出兩個操作數 A 和 BC，計算得到 A + BC。可見不論前綴還是後綴表達式，都能用棧快速地計算出來。

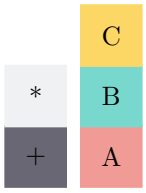


图 3.5: 棧保存表達式

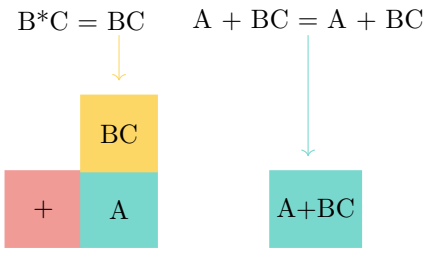


图 3.6: 棧計算表達式

3.3.6 中綴轉前後綴表達式

上面的計算過程說明了中綴表達式需要轉換爲前綴或後綴表達式後計算才高效。所以第一步是，如何得到前後綴表達式？一種方法是採用完全括號表達式。

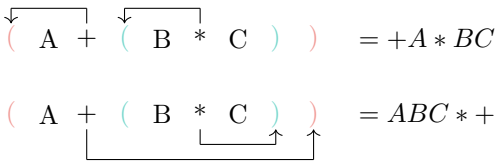


图 3.7: 中綴表達式轉前後綴表達式

A + B * C 可寫成 (A + (B * C))，表示乘法優先於加法。仔細觀察可以看到每個括號對還表示操作數對的開始和結束。(A + (B * C)) 內部表大式是 (B * C)，如將乘法符號移到左括號位置並刪除該左括號和配對的右括號，實際上就已經將子表達式轉換爲了前綴符號。如果加法運算符也被移動到其相應的左括號位置並刪除匹配的右括號，則將得到完整的前綴表達式 + A * B C。如將符號向右移動到右括號位置，並刪除對應的左括號，則可得到後綴表達式。

爲了轉換表達式，無論是轉換爲前綴還是後綴表達式，都要先根據操作的順序把表達式轉換成完全括號表達式。然後再將括號內運算符移動到左或右括號的位置。一個更復雜的例

子是 (A + B) * C - (D + E) / (F + G)，下圖顯示瞭如何將其轉換爲前綴或後綴表達式，對人來說此結果非常複雜，但計算機卻能很好地處理。

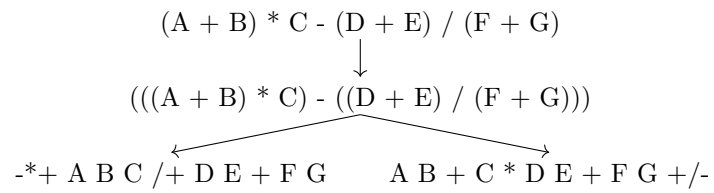


图 3.8: 中綴表達式轉前後綴表達式

然而，得到完全括號表達式本身就很困難，而且移動字符再刪除字符涉及修改字符串，所以這種方法還不夠通用。仔細考察轉換的過程，比如 (A + B) * C 轉換爲 A B + C *，如果不看操作符，則操作數 A B C 還保持着原來的相對位置，只有操作符在改變位置。既然這樣，將操作符單獨處理更方便。遇到操作數不改變位置，直接保留，遇到操作符才處理。可是操作符有優先級，往往會反轉順序。反轉順序是棧的特點，所以可以用棧來保存操作符。

(A + B) * C 這個中綴表達式，從左到右先看到 + 號，由於括號它的優先級高於 * 號。遇到左括號時，表示高優先級的運算符將出現，所以保存它，該操作符需要等到相應的右括號出現以表示其位置。當右括號出現時，可以從棧中彈出操作符。當從左到右掃描中綴表達式時，用棧來保留運算符，棧頂將始終是最近保存的運算符。每當讀取到新的運算符時，需要將其與在棧上的運算符（如果有的話）比較優先級並決定是否彈出。

假設中綴表達式是一個由空格分隔的標記字符串，操作符只有 + - * /，以及左右括號 ()。操作數用字符 A，B，C 表示。下圖展示了將 A * B + C * D 轉換爲後綴表達式的過程，最下面一行是後綴表達式。

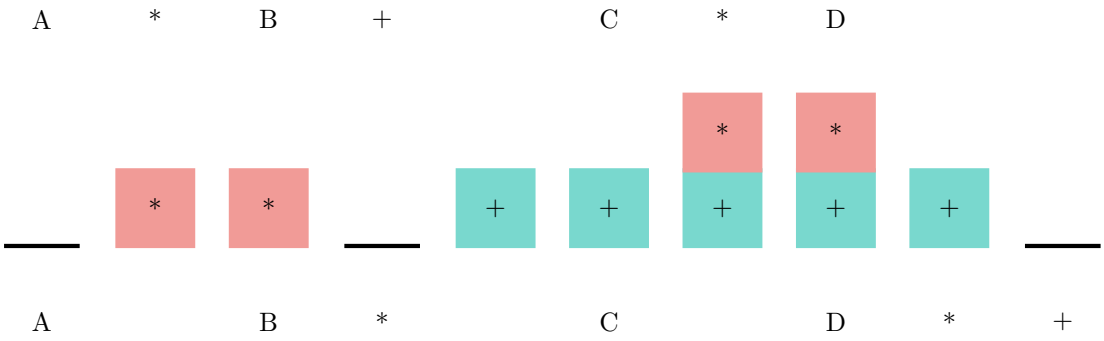


图 3.9: 棧構造後綴表達式

上述轉換的具體步驟如下：

1. 創建一個名爲 op_stack 的空棧以保存運算符。給輸出創建一個空列表 postfix。
2. 通過使用字符串方法拆分將輸入的中綴字符串轉換爲標記列表 src_str。
3. 從左到右掃描標記列表。

如果標記是操作數，將其附加到輸出列表的末尾。

如果標記是左括號，將其壓到 op_stack 上。

如果標記是右括號，則彈出 op_stack，直到刪除相應左括號，將運算符加入 postfix。

如果標記是運算符 + - * /，則壓入 op_stack。但先彈出 op_stack 中更高或相等優先級的運算符到 postfix。

4. 當輸入處理完後，檢查 op_stack，仍在棧上的運算符都可彈出到 postfix。

Algorithm 3.1: 中綴表達式轉後綴表達式算法

Input: 中綴表達式字符串

Output: 後綴表達式字符串

```
1  創建 op_stack 棧保存操作符
2  創建 postfix 列表保存後綴表達式字符串
3  將中綴表達式字符串轉換爲列表 src_str
4  for c ∈ src_str do
5      if c ∈ "A − Z" then
6          postfix.append(c)
7      else if c =='(' then
8          op_stack.push(c)
9      else if c ∈ "+ − ∗ /" then
10         while op_stack.peek() prior to c do
11             postfix.append(op_stack.pop())
12         end
13         op_stack.push(c)
14     else if c ==')' then
15         while op_stack.peek() != '(' do
16             postfix.append(op_stack.pop())
17         end
18     end
19 end
20 while !op_stack.is_empty() do
21     postfix.append(op_stack.pop())
22 end
23 return' '.join(postfix)
```

爲了在 Rust 中編寫這個後綴表達式轉換算法，我們使用了一個名爲 prec 的 HashMap 來保存操作符優先級，它將每個運算符映射爲一個整數，用於與其他運算符優先級進行比較。左括號賦予最低的優先級，這樣，與其進行比較的任何運算符都具有更高的優先級。操作符只有 +-*/，操作數定義爲任何大寫字符 A-Z 或數字 0-9。第一個 if 判斷用於檢測括號是否

匹配，是前面已經學習過的內容。

```
// infix_to_postfix.rs

use std::collections::HashMap;

fn infix_to_postfix(infix: &str) -> Option<String> {
    // 括號匹配檢驗
    if !par_checker3(infix) { return None; }

    // 設置各個符號的優先級
    let mut prec = HashMap::new();
    prec.insert("(", 1); prec.insert(")", 1);
    prec.insert("+", 2); prec.insert("-", 2);
    prec.insert("*", 3); prec.insert("/", 3);

    // ops 保存操作符號、postfix 保存後綴表達式
    let mut op_stack = Stack::new();
    let mut postfix = Vec::new();
    for token in infix.split_whitespace() {
        // 0 - 9  和 A-Z 範圍字符入棧
        if ("A" <= token && token <= "Z") ||
            ("0" <= token && token <= "9") {
            postfix.push(token);
        } else if "(" == token  {
            // 遇到開符號，將操作符入棧
            op_stack.push(token);
        } else if ")" == token  {
            // 遇到閉符號，將操作數入棧
            let mut top = op_stack.pop().unwrap();
            while top != "(" {
                postfix.push(top);
                top = op_stack.pop().unwrap();
            }
        } else {
            // 比較符號優先級來判斷操作符是否加入 postfix
            while (!op_stack.is_empty()) &&
                (prec[op_stack.peek().unwrap()] >= prec[token]) {
                postfix.push(op_stack.pop().unwrap());
```

```
38              }
39
40              op_stack.push(token);
41          }
42      }
43
44      // 剩下的操作數入栈
45      while !op_stack.is_empty() {
46          postfix.push(op_stack.pop().unwrap())
47      }
48
49      // 出栈{\CJKfamily{fangsong}🄵}組成字符串
50      let mut postfix_str = "".to_string();
51      for c in postfix {
52          postfix_str += &c.to_string();
53          postfix_str += " ";
54      }
55
56      Some(postfix_str)
57  }
58
59  fn main() {
60      let infix = "( A + B ) * ( C + D )";
61      let postfix = infix_to_postfix(infix);
62      match postfix {
63          Some(val) => {
64              println!("infix: {infix} -> postfix: {val}");
65          },
66          None => {
67              println!("{infix} is not a corret infix string");
68          },
69      }
70  }
```

計算後綴表達式的方法前面已經陳述過，但要注意 - , / 號，這個操作符不像 + 和 * 操作符。- 和 / 運算符要考慮操作數的順序，A / B 和 B / A，A - B 和 B - A 是完全不同的，不能像 +，* 那樣簡單的處理。

假設後綴表達式是一個由空格分隔的標記字符串，運算符爲 +-*/，操作數爲整數，輸出也是一個整數。下面是計算後綴表達式的算法步驟。

1. 創建一個名爲 op_stack 的空棧。

2. 拆分字符串爲符號列表。

3. 從左到右掃描符號列表。

如果符號是操作數，將其從字符串轉換爲整數，並將值壓到 op_stack。如果符號是運算符，彈出 op_stack 兩次。第一次彈出的是第二個操作數，第二次彈出的是第一個操作數。執行算術運算後，將結果壓到操作數棧中。

4. 當輸入的表達式被完全處理後，結果就在棧上，彈出 op_stack 得到最終運算值。

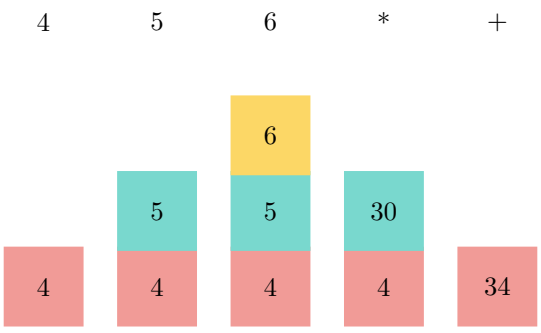


图 3.10: 用棧來計算後綴表達式

具體計算的代碼如下所示。

```
1  //postfix_eval.rs
2
3  fn postfix_eval(postfix: &str) -> Option<i32> {
4      // 少於五個字符，不是有效的後綴表達式，因爲表達式
5      // 至少兩個操作數加一個操作符，還需要兩個空格隔開
6      if postfix.len() < 5 { return None; }
7
8      let mut op_stack = Stack::new();
9      for token in postfix.split_whitespace() {
10         if "0" <= token  && token <= "9" {
11             op_stack.push(token.parse::<i32>().unwrap());
12         } else {
13             // 對於 - 和 /，順序有要求
14             // 所以先出棧的是第二個操作數
15             let op2 = op_stack.pop().unwrap();
16             let op1 = op_stack.pop().unwrap();
17             let res = do_calc(token, op1, op2);
18             op_stack.push(res);
19         }
```

```
20      }
21
22      Some(op_stack.pop().unwrap())
23  }
24
25  // 執行四則數學運算
26  fn do_calc(op: &str, op1: i32, op2: i32) -> i32 {
27      if "+" == op  {
28          op1 + op2
29      } else if "-" == op {
30          op1 - op2
31      } else if "*" == op {
32          op1 * op2
33      } else {
34          if 0 == op2 {
35              panic!("ZeroDivisionError: Invalid operation!");
36          }
37          op1 / op2
38      }
39  }
40
41  fn main() {
42      let postfix = "1 2 + 1 2 + *";
43      let res = postfix_eval(postfix);
44      match res {
45          Some(val) => println!("res is {val}"),
46          None => println!("{postfix} isn't a valid postfix"),
47      }
48  }
```

3.4 隊列

隊列是項的有序結合，其中添加新項的一端稱爲隊尾，移除項的一端稱爲隊首。當一個元素從隊尾進入隊列後，會一直向隊首移動，直到它成爲下一個需要移除的元素爲止。最近添加的元素必須在隊尾等待，集合中存活時間最長的元素在隊首，因爲它經歷了從隊尾到隊首的移動。這種排序稱爲先進先出（First In First Out, FIFO），同棧的 LIFO 相反。

隊列其實在生活中也很常見，單是從其名稱也可見其普遍性。春運時火車站大排長龍等待進站，在公交車站排隊上車，自助餐廳排隊取餐等等都是隊列。隊列的行爲是有限制的，因

爲它只有一個入口，一個出口，不能插隊，也不能提前離開，只有等待一定的時間才能到前面。

操作系統也使用隊列這種數據結構，它使用多個不同的隊列來控制進程。調度算法通常基於儘可能快地執行程序和儘可能多地服務用戶的排隊算法來決定下一步做什麼。還有一種現象，有時敲擊鍵盤，屏幕上出現的字符會有延遲，這是由於計算機在那一刻在做其他工作，按鍵內容被放置在類似隊列的緩衝器中，使得它們最終可以正確地顯示在屏幕上。下圖展示了一個簡單的 Rust 數字隊列。

图 3.11: 隊列

3.4.1 隊列的抽象數據類型

如上所述，隊列被構造成在隊尾添加項的有序集合，並能從隊首移除數據項。隊列保持 FIFO 排序屬性，其抽象數據類型由以下結構和操作定義。

new() 創建一個新隊列，不需要參數，返回一個空隊列。

enqueue(item) 將新項添加到隊尾，需要 item 作爲參數，不返回任何內容。

dequeue() 從隊首移除項，不需要參數，返回 item，隊列被修改。

is_empty() 檢查隊列是否爲空，不需要參數，返回布爾值。

size() 返回隊列中的項數，不需要參數，返回一個整數。

假設 q 是已經創建的空隊列，下表展示了隊列各種操作後的結果，左邊爲隊首。

表 3.4: 隊列操作

隊列操作	隊列當前值	操作返回值
q.is_empty()	[]	true
q.enqueue(1)	[1]	
q.enqueue(2)	[1,2]	
q.enqueue(3)	[1,2,3]	
q.enqueue(5)	[1,2,3,5]	
q.dequeue()	[2,3,5]	1
q.size()	[2,3,5]	3
q.is_empty()	[2,3,5]	false
q.enqueue(4)	[2,3,5,4]	
q.enqueue(6)	[2,3,5,4,6]	
q.dequeue()	[3,5,4,6]	2
q.size()	[3,5,4,6]	4

3.4.2 Rust 實現隊列

前文已定義了隊列的抽象數據類型，現在使用 Rust 來實現隊列。和前面棧類似，這種線性的數據集合用 Vec 就可以，稍微限制下對 Vec 加入和移除元素的用法就能實現隊列。我們選擇 Vec 的左端作爲隊尾，右端作爲隊首，這樣移除數據的複雜度是 O(1)，加入數據的複雜度爲 O(n)。爲了防止隊列無限增長，添加了一個 cap 參數用於控制隊列長度。

```
1  // queue.rs
2
3  // 隊列定義
4  #[derive(Debug)]
5  struct Queue<T> {
6      cap: usize,   // 容量
7      data: Vec<T>, // 數據容器
8  }
9
10 impl<T> Queue<T> {
11     fn new(size: usize) -> Self {
12         Queue {
13             data: Vec::with_capacity(size),
14             cap: size,
15         }
16     }
17
18     // 判斷是否有剩余空間、有則數據加入隊列
19     fn enqueue(&mut self, val: T) -> Result<(), String> {
20         if Self::size(&self) == self.cap {
21             return Err("No space available".to_string());
22         }
23         self.data.insert(0, val);
24
25         Ok(())
26     }
27
28     // 數據出隊
29     fn dequeue(&mut self) -> Option<T> {
30         if Self::size(&self) > 0 {
31             self.data.pop()
32         } else {
33             None
```

```
34          }
35      }
36
37      fn is_empty(&self) -> bool {
38          0 == Self::size(&self)
39      }
40
41      fn size(&self) -> usize {
42          self.data.len()
43      }
44  }
45
46  fn main() {
47      let mut q = Queue::new(3);
48      let _r1 = q.enqueue(1);
49      let _r2 = q.enqueue(2);
50      let _r3 = q.enqueue(3);
51      if let Err(error) = q.enqueue(4) {
52          println!("Enqueue error: {error}");
53      }
54
55      if let Some(data) = q.dequeue() {
56          println!("data: {data}");
57      } else {
58          println!("empty queue");
59      }
60
61      println!("size: {}, empty: {}", q.size(), q.is_empty());
62      println!("content: {:?}", q);
63  }
```

3.4.3 燙手山芋

隊列的典型應用是模擬需要以 FIFO 方式管理數據的真實場景。一個例子是燙手山芋遊戲，在這個遊戲中（見圖3.12），孩子們圍成一個圈，並儘可能快地將一個山芋遞給旁邊的孩子。在某一個時刻，停止傳遞動作，有山芋的孩子從圈中移除，繼續遊戲直到剩下最後一個孩子。這個遊戲使得孩子們儘可能快的把山芋傳遞出去，就像山芋很燙手一樣，所以這個遊戲叫燙手山芋。

這個遊戲相當於著名的約瑟夫問題，一世紀歷史學家弗拉維奧 · 約瑟夫斯講述的傳奇故事。他和 39 個戰友被羅馬軍隊包圍在洞中，他們決定寧願死，也不成爲羅馬人的奴隸。他們圍成一個圈，其中一人被指定爲第一個人，順時針報數到第七人，就將他殺死，直到剩下一人。約瑟夫斯是一個數學家，他立即想出了應該坐到哪個位置才能成爲最後一人。最後，他成了最後一人，加入了羅馬的一方。這個故事有各種不同的版本，有些說是每次報數到第三個人，有人說允許最後一個人逃跑。無論如何，兩個故事思想是一樣的，都可用隊列來模擬。程序將輸入多個人名代表孩子，一個 num 常量用於設定報數到第幾人。

假設拿山芋的孩子始終在隊列的前面。當拿到山芋的時候，這個孩子將先出隊再入隊，把自己放在隊列的最後，這相當於他把山芋傳遞給了下一個孩子，而那個孩子必須處於隊首，所以他自己出隊，讓出了隊首的位置，自己加入了隊尾。經過 num 次的出隊入隊後，前面的孩子被永久移除。接着另一個週期開始，繼續此過程，直到只剩下一個名字。

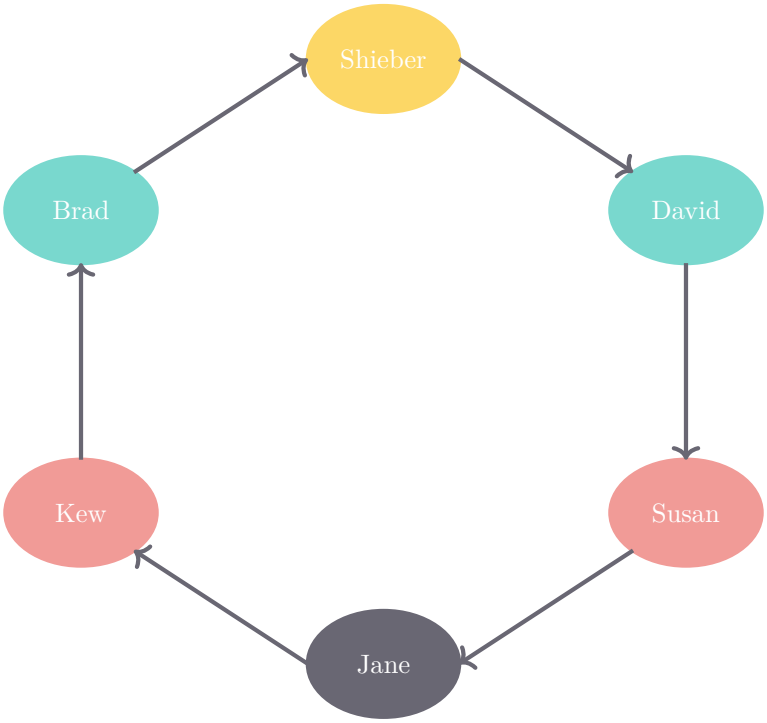


图 3.12: 燙手山芋

通過上面的分析可以得到如下燙手山芋遊戲的隊列模型。

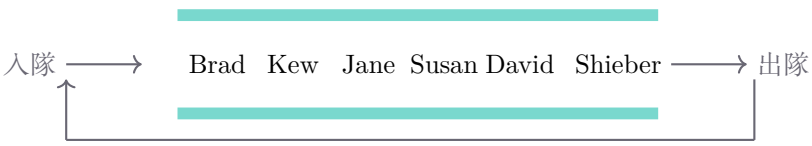


图 3.13: 模擬燙手山芋

上面兩幅圖表示的出入隊列模型十分清楚，下面按照物理模型來實現燙手山芋遊戲。

```
// hot_potato.rs

fn hot_potato(names: Vec<&str>, num: usize) -> &str {
    // 初始化隊列、名字入隊
    let mut q = Queue::new(names.len());
    for name in names {
        let _rm = q.enqueue(name);
    }

    while q.size() > 1 {
        // 出入棧名字，相當於傳遞山芋
        for _i in 0..num {
            let name = q.dequeue().unwrap();
            let _rm = q.enqueue(name);
        }

        // 出入棧達到 num 次，剔除一人
        let _rm = q.dequeue();
    }

    q.dequeue().unwrap()
}

fn main() {
    let name = vec!["Shieber","David","Susan","Jane","Kew","Brad"];
    let rem = hot_potato(name, 8);
    println!("The left person is {rem}");
}
```

請注意，在實現中，計數值 8 大於隊列中人數 6。但這不存在問題，因爲隊列像是一個圈，到尾後會重新回到首部，直到達到計數值，所以最終總是有個人會出隊。

3.5 雙端隊列

deque 又稱爲雙端隊列，是與隊列類似的項的有序集合。它有兩個端部，首端和尾端。deque 不同於 queue 的地方是添加和刪除項是非限制性的，可以在首端尾後端添加項，同樣也可以從任一端移除項。在某種意義上，這種混合線性結構提供了棧和隊列的所有功能。

即使 deque 擁有棧和隊列的許多特性，但它不需要像那些數據結構那樣強制的 LIFO 和 FIFO 排序，這取決於如何添加和刪除數據。你把它當棧用，它就是棧，當隊列用就是隊列，但一般來說，deque 就是 deque，不要當成棧或隊列使用，不同的數據結構都有其獨特性，是爲了不同的計算目的而設計的。下圖展示了一個 deque。

图 3.14: 雙端隊列

3.5.1 雙端隊列的抽象數據類型

如上所述，deque 被構造爲項的有序集合，其中項從首部或尾部的任一端添加和移除，deque 抽象數據類型由以下結構和操作定義。

new() 創建一個新的 deque，不需要參數，返回空的 deque。

add_front(item) 將新項 item 添加到 deque 首部，需要 item 參數，不返回任何內容。

add_rear(item) 將新項 item 添加到 deque 尾部，需要 item 參數，不返回任何內容。

remove_front() 從 deque 中刪除首項，不需要參數，返回 item。deque 被修改。

remove_rear() 從 deque 中刪除尾項，不需要參數，返回 item，deque 被修改。

is_empty() 測試 deque 是否爲空，不需要參數，返回布爾值。

size() 返回 deque 中的項數，不需要參數，返回一個整數。

假設 d 是已經創建的空 deque，下表展示了一系列操作後的結果。注意，首部在右端。將 item 移入和移出時，注意跟蹤前後內容，因爲兩端修改使得結果看起來有些混亂。

表 3.5: 雙端隊列操作

雙端隊列操作	雙端隊列當前值	操作返回值
d.is_empty()	[]	true
d.add_rear(1)	[1]	
d.add_rear(2)	[2,1]	
d.add_front(3)	[2,1,3]	
d.add_front(4)	[2,1,3,4]	
d.size()	[2,1,3,4]	4
d.is_empty()	[2,1,3,4]	false
d.remove_rear()	[1,3,4]	2
d.remove_front()	[1,3]	4
d.size()	[1,3]	2

3.5.2 Rust 實現雙端隊列

前文定義了雙端隊列的抽象數據類型，現在使用 Rust 來實現雙端隊列。和前面隊列類似，這種線性的數據集合用 Vec 就可以。選擇 Vec 的左端作爲隊尾，右端作爲隊首。爲防止隊列無限增長，添加了一個 cap 參數用於控制雙端隊列長度。

```
1  // deque.rs
2
3  // 雙端隊列
4  #[derive(Debug)]
5  struct Deque<T> {
6      cap: usize,   // 容量
7      data: Vec<T>, // 數據容器
8  }
9
10 impl<T> Deque<T> {
11     fn new(cap: usize) -> Self {
12         Deque {
13             cap: cap,
14             data: Vec::with_capacity(size),
15         }
16     }
17
18     // Vec 末尾爲隊首
19     fn add_front(&mut self, val: T) -> Result<(), String> {
20         if Self::size(&self) == self.cap {
21             return Err("No space available".to_string());
22         }
23         self.data.push(val);
24
25         Ok(())
26     }
27
28     // Vec 首部爲隊尾
29     fn add_rear(&mut self, val: T) -> Result<(), String> {
30         if Self::size(&self) == self.cap {
31             return Err("No space available".to_string());
32         }
33         self.data.insert(0, val);
34
```

```
35          Ok(())
36      }
37
38      // 從隊首移除數據
39      fn remove_front(&mut self) -> Option<T> {
40          if Self::size(&self) > 0 {
41              self.data.pop()
42          } else {
43              None
44          }
45      }
46
47      // 從隊尾移除數據
48      fn remove_rear(&mut self) -> Option<T> {
49          if Self::size(&self) > 0 {
50              Some(self.data.remove(0))
51          } else {
52              None
53          }
54      }
55
56      fn is_empty(&self) -> bool {
57          0 == Self::size(&self)
58      }
59
60      fn size(&self) -> usize {
61          self.data.len()
62      }
63  }
64
65  fn main() {
66      let mut d = Deque::new(4);
67      let _r1 = d.add_front(1); let _r2 = d.add_front(2);
68      let _r3 = d.add_rear(3);  let _r4 = d.add_rear(4);
69      if let Err(error) = d.add_front(5) {
70          println!("add_front error: {error}");
71      }
72
```

```
73      if let Some(data) = d.remove_rear() {
74          println!("data: {data}");
75      } else {
76          println!("empty queue");
77      }
78
79      println!("size: {}, is_empty: {}", d.size(), d.is_empty());
80      println!("content: {:?}", d);
81  }
```

可見，Deque 同 Queue 和 Stack 都有相似之處，感覺像是兩者的合集一樣。具體的實現是可以商榷的，哪端是首，哪端是尾要依據情況而定。

3.5.3 迴文檢測

迴文是一個字符串，距離首尾兩端相同位置處的字符相同。例如 radar，sos，rustsur 這類字符串。可以寫一個算法來檢查一個字符串是否是迴文。一種方式是用隊列 Queue，將字符串入隊，然後字符再出隊。此時出隊的字符和原字符串的倒序相比較，如果相等就繼續出隊比較下一個，直到完成。這種方式很直觀，但是費內存。檢查迴文的另一種方案是使用 Deque（如下圖）。

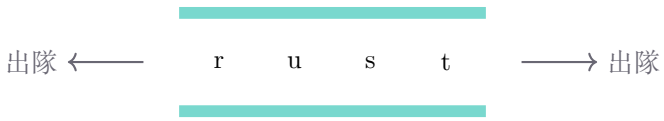


图 3.15: 迴文檢測

從左到右處理字符串，將每個字符添加到 Deque 尾部，此時 deque 的首部保存字符串的第一個字符，尾部保存最後一個字符。此時可以直接利用 Deque 的雙邊出隊特性，刪除首尾字符並比較，只有當首尾字符相等時才繼續。如果可以持續匹配首尾字符，最終要麼用完字符，留下空隊列，要麼留出大小爲 1 的隊列。在這兩種情況下，字符串均是迴文，這取決於原始字符串的長度是偶數還是奇數。下面是迴文檢查的完整代碼。

```
1  // pal_checker.rs
2
3  fn pal_checker(pal: &str) -> bool {
4      let mut d = Deque::new(pal.len());
5      for c in pal.chars() {
6          let _r = d.add_rear(c);
7      }
8
```

```
 9      let mut is_pal = true;
10      while d.size() > 1 && is_pal {
11          let head = d.remove_front();
12          let tail = d.remove_rear();
13          if head != tail { // 比較首尾字符，若不同則非回文
14              is_pal = false;
15          }
16      }
17
18      is_pal
19  }
20
21  fn main() {
22      let pal = "rustsur";
23      let is_pal = pal_checker(pal);
24      println!("{pal} is palindrome string: {is_pal}");
25  }
```

3.6 鏈表

有序的數據項集合能保證數據的相對位置，可以高效地索引。數組和鏈表都能做到將數據有序地收集起來並保存在相對的位置，所以數組和鏈表都可用於實現有序數據類型，比如 Rust 默認實現的 Vec 就是用的數組這種有序集合。當然，本節研究鏈表，我們要先實現鏈表再用其來實現其他的有序數據類型。

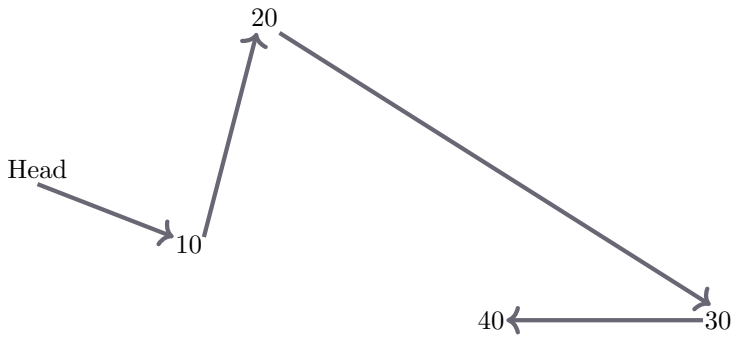


图 3.16: 鏈表數據關係

數組是一片連續的內存，且增刪元素涉及到內存複製和移動等操作，非常耗時。鏈表不要求元素保存在連續的內存中。如上圖所示的項集合，這些值隨機放置，只要每個項中都有

下一項的位置，則項的位置可以通過簡單地從一個項到下一個項的鏈接來表示，用不着像數組那樣去分配整塊內存，這樣效率會更高。

必須明確地指定鏈表的第一項位置，一旦知道第一項在哪裏，就可以知道第二項在哪裏，直到整個鏈表結束。鏈表對外提供的通常是鏈表的頭（Head），類似地，最後一個項要設置下一個項爲空（None）或稱爲接地。

3.6.1 鏈表的抽象數據類型

鏈表的抽象數據類型由其結構和操作定義。如上所述，鏈表被構造爲節點項的有序集合，可以從頭節點遍歷整個鏈表數據。鏈表結構很簡單，只有一個頭節點引用。下面是鏈表節點 Node 的具體的定義和操作。

new() 創建一個新的頭節點用於指向 Node，不需要參數，返回指針。

push(item) 添加一個新的 Node，需要 item 參數，返回 Node。

pop() 刪除鏈表頭節點，不需要參數，返回 Node。

peek() 返回鏈表頭節點，不需要參數，返回對值的引用。

peek_mut() 返回鏈表頭節點，不需要參數，返回對值的可變引用。

into_iter() 改變鏈表爲可迭代形式，不需要參數。

iter() 返回鏈表不可變迭代形式，鏈表不變，不需要參數。

iter_mut() 返回鏈表可變迭代形式，鏈表不變，不需要參數。

into_iter() 返回鏈表頭節點，不需要參數，返回對值的可變引用。

is_empty() 返回當前鏈表是否爲空，不需要參數，返回布爾值。

size() 計算鏈表的長度，不需要參數，返回整數值。

假設 l 是已經創建的空鏈表，下表展示了鏈表操作序列後的結果，左端爲頭節點，注意 Link<num> 表示指向 num 所在節點的地址。

表 3.6: 鏈表操作

鏈表操作	鏈表當前值	操作返回值
l.is_empty()	[None->None]	true
l.push(1)	[1->None]	
l.push(2)	[2->1->None]	
l.push(3)	[3->2->1->None]	
l.peek()	[3->2->1->None]	Link<3>
l.pop()	[2->1->None]	3
l.size()	[2->1->None]	2
l.push(4)	[4->2->1->None]	
l.peek_mut()	[4->3->2->None]	mut Link<4>
l.iter()	[4->3->2->None]	[4,3,2]
l.is_empty()	[4->3->2->None]	false
l.into_iter()	[None->None]	[4,3,2]

3.6.2 Rust 實現鏈表

鏈表中的每項都可抽象成一個節點 Node，節點保存了數據項和下一項位置。當然，節點還提供獲取和修改數據項的方法。如下圖所示，Node 包含數據和下一結點的地址。

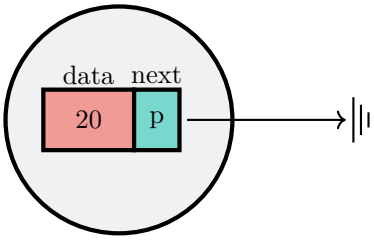


图 3.17: 鏈表節點

Rust 中的 None 將在 Node 和鏈表中發揮重要作用，None 地址代表沒有下一個節點。請注意在 new 函數中，最初創建的節點 next 被設置爲 None，這被稱爲接地節點，將 None 顯式的分配給 next 是個好主意，這避免了 C++ 等語言中容易出現的懸蕩指針。

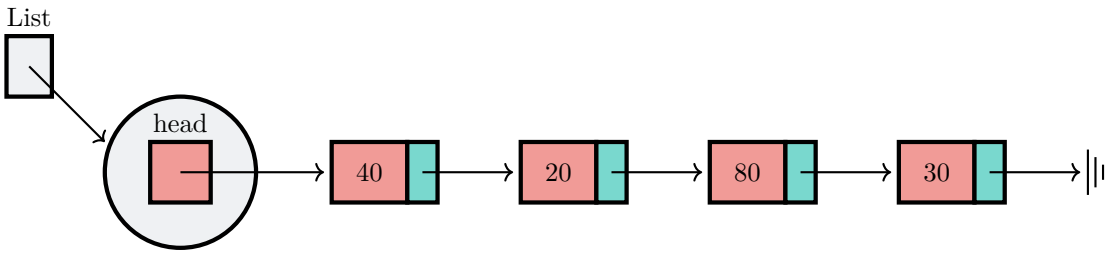


图 3.18: 鏈表 List

下面是鏈表的實現代碼，爲了對鏈表實現迭代功能，我們添加了 IntoIter、Iter、IterMut 三個結構體，分別完成三種迭代功能，InotIter 是將鏈表整個轉換爲可迭代類型，Iter 只迭代，不修改鏈表，IterMut 也是迭代，但可以修改鏈表結點。Drop 類似析構函數。

```
1  // linked_list.rs
2  // 節點連接用 Box 指針（大小確定），因爲確定大小才能分配空間
3  type Link<T> = Option<Box<Node<T>>>;
4
5  // 鏈表定義
6  pub struct List<T> {
7      size: usize,   // 鏈表節點數
8      head: Link<T>, // 頭節點
9  }
10
11 // 鏈表節點
```

```
struct Node<T> {
    elem: T,       // 數據
    next: Link<T>, // 下一個節點鏈接
}

impl<T> List <T> {
    pub fn new() -> Self {
        List { size: 0, head: None }
    }

    pub fn is_empty(&self) -> bool {
        0 == self.size
    }

    pub fn size(&self) -> usize {
        self.size
    }

    // 新節點總是加到頭部
    pub fn push(&mut self, elem: T) {
        let node = Box::new(Node {
            elem: elem,
            next: self.head.take(),
        });
        self.head = Some(node);
        self.size += 1;
    }

    // take 會取出數據然後留下空位
    pub fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            self.head = node.next;
            self.size -= 1;
            node.elem
        })
    }

    // peek 不改變值，只能是引用
```

```
50      pub fn peek(&self) -> Option<&T> {
51          self.head.as_ref().map(|node| &node.elem )
52      }
53
54      // peek_mut 可改變值，是可變引用
55      pub fn peek_mut(&mut self) -> Option<&mut T> {
56          self.head.as_mut().map(|node| &mut node.elem )
57      }
58
59      // 以下是實現的三種迭代功能
60      // into_iter: 鏈表改變，成爲迭代器
61      // iter: 鏈表不變，只得到不可變迭代器
62      // iter_mut: 鏈表不變，得到可變迭代器
63      pub fn into_iter(self) -> IntoIter<T> {
64          IntoIter(self)
65      }
66
67      pub fn iter(&self) -> Iter<T> {
68          Iter { next: self.head.as_deref() }
69      }
70
71      pub fn iter_mut(&mut self) -> IterMut<T> {
72          IterMut { next: self.head.as_deref_mut() }
73      }
74  }
75
76  // 實現三種迭代功能
77  pub struct IntoIter<T>(List<T>);
78  impl<T> Iterator for IntoIter<T> {
79      type Item = T;
80      fn next(&mut self) -> Option<Self::Item> {
81          self.0.pop()
82      }
83  }
84
85  pub struct Iter<'a, T: 'a> { next: Option<&'a Node<T>> }
86  impl<'a, T> Iterator for Iter<'a, T> {
87      type Item = &'a T;
```

```
88      fn next(&mut self) -> Option<Self::Item> {
89          self.next.map(|node| {
90              self.next = node.next.as_deref();
91              &node.elem
92          })
93      }
94  }
95
96  pub struct IterMut<'a, T: 'a> { next: Option<&'a mut Node<T>> }
97  impl<'a, T> Iterator for IterMut<'a, T> {
98      type Item = &'a mut T;
99      fn next(&mut self) -> Option<Self::Item> {
100         self.next.take().map(|node| {
101             self.next = node.next.as_deref_mut();
102             &mut node.elem
103         })
104     }
105 }
106
107 // 爲鏈表實現自定義 Drop
108 impl<T> Drop for List<T> {
109     fn drop(&mut self) {
110         let mut link = self.head.take();
111         while let Some(mut node) = link {
112             link = node.next.take();
113         }
114     }
115 }
116
117 fn main() {
118     fn basics() {
119         let mut list = List::new();
120         list.push(1); list.push(2); list.push(3);
121         assert_eq!(list.pop(), Some(3));
122         assert_eq!(list.peek(), Some(&2));
123         assert_eq!(list.peek_mut(), Some(&mut 2));
124         list.peek_mut().map(|val| {
125             *val = 4;
```

```
126         });
127         assert_eq!(list.peek(), Some(&4));
128         println!("basics test Ok!");
129     }
130 
131     fn into_iter() {
132         let mut list = List::new();
133         list.push(1); list.push(2); list.push(3);
134         let mut iter = list.into_iter();
135         assert_eq!(iter.next(), Some(3));
136         assert_eq!(iter.next(), Some(2));
137         assert_eq!(iter.next(), Some(1));
138         assert_eq!(iter.next(), None);
139         println!("into_iter test Ok!");
140     }
141 
142     fn iter() {
143         let mut list = List::new();
144         list.push(1); list.push(2); list.push(3);
145         let mut iter = list.iter();
146         assert_eq!(iter.next(), Some(&3));
147         assert_eq!(iter.next(), Some(&2));
148         assert_eq!(iter.next(), Some(&1));
149         assert_eq!(iter.next(), None);
150         println!("iter test Ok!");
151     }
152 
153     fn iter_mut() {
154         let mut list = List::new();
155         list.push(1); list.push(2); list.push(3);
156         let mut iter = list.iter_mut();
157         assert_eq!(iter.next(), Some(&mut 3));
158         assert_eq!(iter.next(), Some(&mut 2));
159         assert_eq!(iter.next(), Some(&mut 1));
160         assert_eq!(iter.next(), None);
161         println!("iter_mut test Ok!");
162     }
163 
```

```
164     basics();
165     into_iter();
166     iter();
167     iter_mut();
168 }
```

3.6.3 鏈表棧

前面使用 Vec 實現了棧，其實還可用鏈表來實現棧，因爲都是線性數據結構。假設鏈表的頭部將保存棧頂部元素，隨着棧增長，新項將被添加到鏈表頭部，實現如下。

注意，push 和 pop 函數會改變鏈表的結點，所以使用了 take 函數來取出結點值。

```
1  // list_stack.rs
2  // 鏈表節點
3  #[derive(Debug, Clone)]
4  struct Node<T> {
5      data: T,
6      next: Link<T>,
7  }
8
9  type Link<T> = Option<Box<Node<T>>>;
10
11 impl<T> Node<T> {
12     fn new(data: T) -> Self {
13         Node {
14             data: data, next: None, // 初始化時無下一鏈接
15         }
16     }
17 }
18
19 // 鏈表棧
20 #[derive(Debug, Clone)]
21 struct Stack<T> {
22     size: usize,
23     top: Link<T>,  // 棧頂控制整個棧
24 }
25
26 impl<T: Clone> Stack<T> {
27     fn new() -> Self {
```

```
28          Stack { size: 0, top: None }
29      }
30
31      // take 取出 top 中節點，留下空位，所以可以回填節點
32      fn push(&mut self, val: T) {
33          let mut node = Node::new(val);
34          node.next = self.top.take();
35          self.top = Some(Box::new(node));
36          self.size += 1;
37      }
38
39      fn pop(&mut self) -> Option<T> {
40          self.top.take().map(|node| {
41              let node = *node;
42              self.top = node.next;
43              node.data
44          })
45      }
46
47      // as_ref 將 top 節點轉爲引用對象
48      fn peek(&self) -> Option<T> {
49          self.top.as_ref().map(|node| {
50              &node.data
51          })
52      }
53
54      fn size(&self) -> usize {
55          self.size
56      }
57
58      fn is_empty(&self) -> bool {
59          0 == self.size
60      }
61  }
62
63  fn main() {
64      let mut s = Stack::new();
65      s.push(1); s.push(2); s.push(4);
```

```
66      println!("top {:?}, size {}",s.peek().unwrap(), s.size());
67      println!("pop {:?}, size {}",s.pop().unwrap(), s.size());
68      println!("is_empty:{}, stack:{:?}", s.is_empty(), s);
69  }
```

3.7 Vec

在對基本數據結構的討論中，我們使用 Vec 這一基礎數據類型來實現了棧，隊列等多種抽象數據類型。Vec 是一個強大但簡單的數據容器，提供了數據收集機制和各種各樣的操作，這也是反覆使用它來作爲底層數據結構的原因。Vec 類似於 Python 中的 List，使用非常方便。然而，不是所有的編程語言都包括 Vec，或者說不是所有的數據類型都適合你。在某些情況下，Vec 或類似的數據容器必須由程序員單獨實現。

3.7.1 Vec 的抽象數據類型

如上所述，Vec 是項的集合，其中每個項保持相對於其他項的相對位置。下面給出了 Vec 抽象數據類型的各種操作。

new() 創建一個新的 Vec，不需要參數，返回一個空 Vec。

push(item) 將新項添加到 Vec 末尾，需要 item 參數，不返回任何內容。

pop() 刪除 Vec 中的末尾項，不需要參數，返回刪除的項。

insert(pos,item) 在 Vec 的 pos 處插入新項，需要 pos 和 item 參數，不返回任何內容。

remove(index) 從列表中刪除第 index 項，需要 index 作爲索引，返回刪除項。

find(item) 在 Vec 中檢查 item 項是否存在，需要 item 參數，返回一個布爾值。

is_empty() 檢查 Vec 是否爲空，不需要參數，返回布爾值。

size() 計算 Vec 的項數，不需要參數，返回一個整數。

假設 v 是已經創建的空 Vec，下表展示了 Vec 不同操作後的結果，其中左側爲首部。

表 3.7: Vec 操作

Vec 操作	Vec 當前值	操作返回值
v.is_empty()	[]	true
v.push(1)	[1]	
v.push(2)	[1,2]	
v.size()	[1,2]	2
v.pop()	[1]	2
v.push(5)	[1,5]	
v.find(4)	[1,5]	false
v.insert(0,8)	[8,1,5]	
v.remove(0)	[8,1,5]	8

3.7.2 Rust 實現 Vec

如上所述，Vec 將用一組鏈表節點來構建，每個節點通過顯式引用鏈接到下一個節點。只要知道在哪裏找到第一個節點，之後的每項都可以通過連續獲取下一個鏈接找到。

考慮到引用在 Vec 中的作用，Vec 必須保持對第一個節點的引用。創建如圖（3.19）所示的鏈表，None 用於表示鏈表不引用任何內容。鏈表的頭指代列表的第一節點，該節點保存下一個節點的地址。要注意到 Vec 本身不包含任何節點對象，相反，它只包含對鏈表結構中第一個節點的引用。

那麼，如何將新項加入鏈表呢? 加到首部還是尾部呢? 鏈表結構只提供了一個入口點，即鏈表頭部。所有其他節點只能通過訪問第一個節點，然後跟隨下一個鏈接到達。這意味着添加新節點的最簡單的地方就在鏈表的頭部，換句話說，將新項作爲鏈表的第一項，現有項🄵接到這個新項後面。

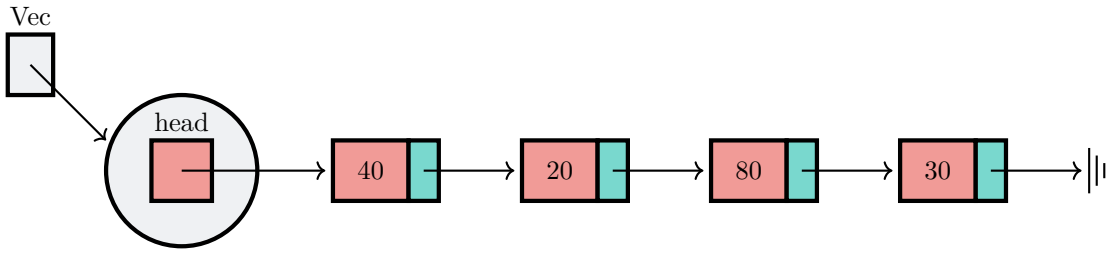


图 3.19: 鏈表節點組成的的 Vec

我們實現的 Vec 是無序的，若要實現有序 Vec 只需添加數據項比較函數。下面實現的 LVec，僅實現了 Rust Vec 中的部分功能，包括 new、insert、pop、remove、push、append 等方法，print_lvec 用於打印 LVec 數據項。

```
1  // lvec.rs
2  use std::fmt::Debug;
3
4  // 節點
5  #[derive(Debug)]
6  struct Node<T> {
7      elem: T,
8      next: Link<T>,
9  }
10
11 type Link<T> = Option<Box<Node<T>>>;
12
13 impl<T> Node<T> {
14     fn new(elem: T) -> Node<T> {
15         Node { elem: elem, next: None }
```

```
16      }
17  }
18
19  // 鏈表 Vec
20  #[derive(Debug)]
21  struct LVec<T> {
22      size: usize,
23      head: Link<T>,
24  }
25
26  impl<T: Copy + Debug> LVec<T> {
27      fn new() -> Self {
28          LVec { size: 0, head: None }
29      }
30
31      fn clear(&mut self) {
32          self.size = 0;
33          self.head = None;
34      }
35
36      fn len(&self) -> usize {
37          self.size
38      }
39
40      fn is_empty(&self) -> bool {
41          0 == self.size
42      }
43
44      fn push(&mut self, elem: T) {
45          let node = Node::new(elem);
46
47          if self.is_empty() {
48              self.head = Some(Box::new(node));
49          } else {
50              let mut curr = self.head.as_mut().unwrap();
51
52              // 找到最後一個節點
53              for _i in 0..self.size-1 {
```

```
54                  curr = curr.next.as_mut().unwrap();
55              }
56
57              // 最後一個節點後插入新數據
58              curr.next = Some(Box::new(node));
59          }
60
61          self.size += 1;
62      }
63
64      // 棧末尾加入數據
65      fn append(&mut self, other: &mut Self) {
66          while let Some(node) = other.head.as_mut().take() {
67              self.push(node.elem);
68              other.head = node.next.take();
69          }
70          other.clear();
71      }
72
73      fn insert(&mut self, mut index: usize, elem: T) {
74          if index >= self.size {
75              index = self.size;
76          }
77
78          // 分三種情況插入新節點
79          let mut node = Node::new(elem);
80          if self.is_empty() { // LVec 爲空, 直接插入
81              self.head = Some(Box::new(node));
82          } else if index == 0 { // 插入鏈表首部
83              node.next = self.head.take();
84              self.head = Some(Box::new(node));
85          } else { // 插入鏈表中間
86              let mut curr = self.head.as_mut().unwrap();
87              for _i in 0..index - 1 { // 找到插入位置
88                  curr = curr.next.as_mut().unwrap();
89              }
90              node.next = curr.next.take();
91              curr.next = Some(Box::new(node));
```

```
92          }
93          self.size += 1;
94      }
95
96      fn pop(&mut self) -> Option<T> {
97          self.remove(self.size - 1)
98      }
99
100     fn remove(&mut self, index: usize) -> Option<T> {
101         if index >= self.size { return None; }
102
103         // 分兩種情形去除節點，首節點去除最好處理
104         let mut node;
105         if 0 == index {
106             node = self.head.take().unwrap();
107             self.head = node.next.take();
108         } else { // 非首節點需要找到待去除點，然後處理前後鏈接
109             let mut curr = self.head.as_mut().unwrap();
110             for _i in 0..index-1 {
111                 curr = curr.next.as_mut().unwrap();
112             }
113             node = curr.next.take().unwrap();
114             curr.next = node.next.take();
115         }
116         self.size -= 1;
117
118         Some(node.elem)
119     }
120
121     // 打印 LVec，當然也可以實現 ToString 特性再用 println 打印
122     fn print_lvec(&self) {
123         let mut curr = self.head.as_ref();
124         while let Some(node) = curr {
125             println!("lvec val: {:#?}", node.elem);
126             curr = node.next.as_ref();
127         }
128     }
129 }
```

```
130
131 fn main() {
132     let mut lvec: LVec<i32> = LVec::new();
133     lvec.push(10); lvec.push(11);
134     lvec.push(12); lvec.push(13);
135     lvec.insert(0,9);
136
137     let mut lvec2: LVec<i32> = LVec::new();
138     lvec2.insert(0, 8);
139     lvec2.append(&mut lvec);
140     println!("lvec2 len: {}", lvec2.len());
141     lvec2.print_lvec();
142
143     let res1 = lvec2.pop();
144     let res2 = lvec2.remove(0);
145     println!("pop {:#?}", res1.unwrap());
146     println!("remove {:#?}", res2.unwrap());
147     lvec2.print_lvec();
148 }
```

LVec 是具有 n 個節點的鏈表，insert，push，pop，remove 等都需要遍歷結點，雖然平均來說可能只需要遍歷節點的一半，但總體上都是 O(n)，因爲在最壞的情況下，都要處理鏈表中的每個節點。

3.8 總結

本章主要學習了棧、隊列、雙端隊列、鏈表、Vec 這些線性數據結構。棧是維持後進先出（LIFO）排序的數據結構，其基本操作是 push，pop，is_empty。棧對於設計計算解析表達式算法非常有用，棧可以提供反轉特性，在操作系統函數調用，網頁保存方面非常有用。前綴，中綴和後綴表達式都是表達式，可以用棧來處理，但計算機不用中綴表達式。隊列是維護先進先出（FIFO）排序的簡單數據結構，其基本操作是 enqueue，dequeue，is_empty，隊列在系統任務調度方面很實用，可以幫助構建定時仿真。雙端隊列是允許類似棧和隊列的混合行爲的數據結構，其基本操作是 is_empty，add_front，add_rear，remove_front，remove_rear。鏈表是項的集合，其中每個項保存在相對位置。鏈表的實現本身就能保持邏輯順序，不需要物理順序存儲。修改鏈表頭是一種特殊情況。Vec 是 Rust 自帶的數據容器，默認實現是用的動態數組，本章使用的是鏈表。

Chapter 4

遞歸

4.1 本章目標

要理解簡單的遞歸解決方案
學習如何用遞歸寫出程序
理解和應用遞歸三個定律
將遞歸理解爲一種迭代形式
將問題公式化地實現成遞歸
瞭解計算機如何實現遞歸

4.2 什麼是遞歸

遞歸是一種解決問題的方法，通過將問題分解爲更小的子問題，直到得到一個足夠小的基本問題。這個基本問題可以被很簡單地解決，再通過合併基本問題的結果得到大問題的結果。因爲這些基本問題是類似的，可以重用解決方案，所以遞歸涉及到函數調用自身。遞歸允許你編寫非常優雅的解決方案解決看起來可能很難的問題。

舉個簡單的例子。假設你想計算整數 [2,1,7,4,5] 的總和，最直觀的就是用一個加法累加器，逐個將值與之相加，具體代碼如下。

```
1 fn nums_sum(nums: Vec<i32>) -> i32 {
2     let mut sum = 0;
3     for num in nums {
4         sum += num;
5     }
6
7     sum
8 }
```

現在假設一種編程語言沒有 while 循環或 for 循環，那麼上面的代碼就不成立了，此時你又將如何計算該整數列表的總和呢？要解決這個問題得換個角度思考問題，當解決大問題（數列求和）困難時，要考慮使用小問題去替代。加法是兩個操作數和一個加法符號組合的運算邏輯，這是任何複雜加法的基本問題。所以如果能將數列求和分解爲一個個小的加法和，則總是能解決這個求和問題的。

構造小加法需要使用你小學學過的知識，構造完全括號表達式，當然這種括號表達式有多種形式。

$$\begin{aligned} 2+1+7+4+5 &= ((((2+1)+7)+4)+5) \\ sum &= (((3+7)+4)+5) \\ sum &= ((10+4)+5) \\ sum &= (14+5) \\ sum &= 19 \\ &= (2+(1+(7+(4+5)))) \\ sum &= (2+(1+(7+9))) \\ sum &= (2+(1+16)) \\ sum &= (2+17) \\ sum &= 19 \end{aligned} \tag{4.1}$$

上面的式子，右側兩種括號表達式都是正確的。運用括號優先，內部優先的規則，那麼上述括號表達式就是一個一個的小加法，完全可以不用 While 或 For 循環來模擬實現上面這種括號表達式。

觀察以 sum 開頭的計算式，從下往上看，先是 19，接着是 (2 + 17)，然後是 (2 + (1 + 16))。可以發現，總和是第一項和右端剩下項的和，而右端剩下項又可以分解爲它的第一項和右端剩下項的和。用數學表達式就是：

$$Sum(nums) = First(nums) + Sum(restR(nums))$$

這是括號表達式中的第二種表示法的計算方式，當然還有第一種括號表達式的計算方式，具體如下：

$$Sum(nums) = Last(nums) + Sum(restL(nums))$$

方程式中，First(nums) 返回數列第一個元素，restR(nums) 返回除第一個元素之外的所有右端數字項。Last(nums) 返回數列最後一個元素，restL(nums) 返回除最後一個元素外的所有左端數字項。

用 Rust 遞歸實現的兩種表達式計算方法如下。nums_sum1 使用 nums[0] 加剩下的項來求和，而 nums_sum2 使用最後一項和前面所有項來求和，兩實現的效率幾乎相等。

```rust
// nums_sum12.rs

fn nums_sum1(nums: &[i32]) -> i32 {
    if 1 == nums.len() {
        nums[0]
    } else {
        let first = nums[0];
        first + nums_sum1(&nums[1..])
    }
}

fn nums_sum2(nums: &[i32]) -> i32 {
    if 1 == nums.len() {
        nums[0]
    } else {
        let last = nums[nums.len() - 1];
        nums_sum2(&nums[..nums.len() - 1]) + last
    }
}

fn main() {
    let nums = [2,1,7,4,5];
    let sum1 = nums_sum1(&nums);
    let sum2 = nums_sum2(&nums);
    println!("sum1 is {sum1}, sum2 is {sum2}");
}
```

代碼中關鍵處是 if 和 else 語句及其形式。if 1 == nums.len() 檢查是至關重要的，因爲這是函數的轉折點，這裏返回數字。else 語句中調用自身，實現了類似逐層解括號並計算值的效果，這也是被稱之爲遞歸的原因。遞歸函數總會調用自身，直到到達基本情況。

4.2.1 遞歸三定律

通過分析上面的代碼可以看出，所有遞歸算法必須遵從三個基本規律:

```
遞歸算法必須具有基本情況
遞歸算法必須向基本情況靠近
遞歸算法必須以遞歸方式調用自身
```

第一條就是算法停止的情況，此處是 if 1 == nums.len() 子句，第二條意味着問題的分解，在 else 中，我們返回了數字，然後使用除返回數字項的集合再次計算，這減小了原來的 nums 中需要計算的元素個數，可見總會有 nums.len() == 1 的時候，所以 else 子句確實在向基本情況靠近。第三條，調用自身，也是在 else 子句實現的。要注意，調用自身不是循環，這裏也沒有 while 和 for 語句。綜上，我們的求和遞歸算法是符合這三個定律的。

下圖展示了上面的遞歸調用函數處理過程，一系列方框即函數調用關係圖，每次遞歸都是解決一個小問題，直到小問題達到基本情況不能再分解，最後再回溯這些中間計算值來求大問題的結果。

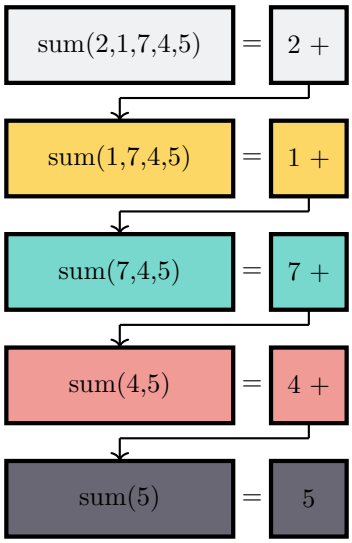


图 4.1: 遞歸求值

4.2.2 到任意進制的轉換

前面在棧那一節實現了整數轉換爲二進制和十六進制字符串。例如，將整數 10 轉換爲二進制字符串 1010。其實使用遞歸來實現進制的轉換也是可以的。

要用遞歸設計一個進制轉換算法，遵循遞歸三定律將是必須的，也是算法的設計指南。

```
1 將原始數字簡化爲一系列單個數字。
2 使用查找法將單個數字轉爲字符。
3 將單個字符連接在一起以形成最終結果。
```

改變數字狀態並向基本情況靠近的方法是採用除法，用數字除以基數，當數字小於基數時停止運算，返回結果。比如 10 進制數 996，以 10 爲進制，看轉換後的結果。先整除 10，余數 6，商 99。余數小於 10，可以直接求得其字符 “6”，商 99 小於 996，在向基本情況靠近。接着再遞歸調用自身，把 99 除以 10 商 9 余 9，余數小於 10，轉換爲字符 “9”，繼續計算再得到一個 “9”，最終得到 996 的 10 進制字符串 “996”。一旦能解決 10 進制，那麼 2-16

進制就都沒有問題了。

下面是整數到任意進制（2-16 進制）字符串的轉換算法，BASESTR 中保存着不同數字對應的字符形式，大於 10 的數字用字符 A-F 來表示。

```
// num2str_rec.rs

const BASESTR: [&str; 16] = ["0","1","2","3","4","5","6","7",
                             "8","9","A","B","C","D","E","F"];

fn num2str_rec(num: i32, base: i32) -> String {
    if num < base {
        BASESTR[num as usize].to_string()
    } else {
        // 余數加在末尾
        num2str_rec(num/base, base) +
            BASESTR[(num % base) as usize]
    }
}

fn main() {
    let num = 100;
    let sb = num2str_rec(num,2);
    let so = num2str_rec(num,8);
    let sh = num2str_rec(num,16);
    println!("{num} is b{sb}, o{so}, x{sh}");
}
```

前面我們用棧實現了數字到任意進制的轉換，這裏用遞歸再次實現了類似功能，這說明棧和遞歸是有關係的。實際上可以把遞歸看成是棧，只是這個棧是由編譯器爲我們隱式調用的，代碼中只用了遞歸，但編譯器使用了棧來保存數據。

上面的遞歸代碼改用棧來實現的話，應該是下面這樣的代碼，和遞歸實現的代碼結構非常相似。

```
// num2str_stk.rs

fn num2str_stk(mut num: i32, base: i32) -> String {
    let digits: [&str; 16] = ["0","1","2","3","4","5","6","7",
                              "8","9","A","B","C","D","E","F"];

    let mut rem_stack = Stack::new();
```

```
8       while num > 0 {
9           if num < base {
10              rem_stack.push(num); // 不超過 base 直接入栈
11          } else { // 超過 base 余數入栈
12              rem_stack.push(num % base);
13          }
14          num /= base;
15      }
16
17      // 出栈余數以組成字符串
18      let mut numstr = "".to_string();
19      while !rem_stack.is_empty() {
20          numstr += digits[rem_stack.pop().unwrap() as usize];
21      }
22
23      numstr
24  }
25
26  fn main() {
27      let num = 100;
28      let sb = num2str_stk(100, 2);
29      let so = num2str_stk(100, 8);
30      let sh = num2str_stk(100, 16);
31      println!("{num} is b{sb}, o{so}, x{sh}");
32  }
```

4.2.3 漢諾塔

漢諾塔是由法國數學家愛德華 · 盧卡斯在 1883 年發明的。他的靈感來自一個傳說。一個印度教寺廟，將謎題交給年輕的牧師。在開始的時候，牧師們被給予三根杆和 64 個金碟，每個盤比它下面一個小一點。他們的任務是將所有 64 個盤子從三個杆中一個轉移到另一個。但是移動是有限制的，一次只能移動一個盤子，且大盤子不能放在小的盤子上面，換句話說，小盤子始終在上，下面的盤子更大。牧師門日夜不停，每秒鐘移動一塊盤子。當他們完成工作時，傳說，寺廟會變成灰塵，世界將消失。

實際上移動 64 個盤子的漢諾塔所需的時間是 $2^{64} - 1 = 18446744073709551,615$ s $=$ 5850 億年，超過了已知宇宙存在的時間 138 億年。

圖4.2展示了盤從第一杆移動到第三杆的示例。請注意，如規則指定，每個杆上的盤子都被堆疊起來，以使較小的盤子始終位於較大盤的頂部，這看起來類似一個棧。如果你以前沒

有玩兒過這個，你現在可以嘗試下。不需要盤子，一堆磚，書或紙都可以。你可以試試是不是真的那麼費時。

如何用遞歸解決這個問題呢？首先回想遞歸三定律，找出基本情況是什麼？假設有一個漢諾塔，有左中右三根杆，五個盤子在左杆上。如果你已經知道如何將四個盤子移動到中杆上，那麼可以輕松地將最底部的盤子移動到右杆，然後再將四個盤子從中杆移動到右杆。但是如果不知道如何移動四個盤子到中杆怎麼辦？這時可以假設你知道如何移動三個盤子到右杆，那麼很容易將第四個盤子移動到中杆，並將右杆上盤子移動到中杆盤子的頂部。但是還不知道如何移動三個盤子呢？這時再假設知道何將兩個盤子移動到中杆，接着將第三個盤子移動到右杆，然後再移動兩個盤子到它的頂部？可是兩個盤子的移動仍然不知道，所以再假設你知道移動一個盤子到右。這看起來就像是基本情況。實際上，上面的描述雖然繞得很，但這個過程其實就是移動盤子的過程的抽象，最基本的情況就是移動一個盤子。

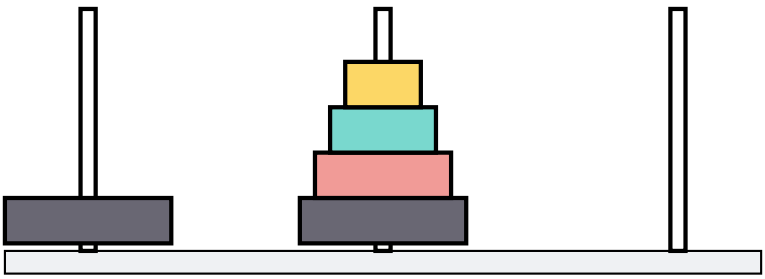


图 4.2: 漢諾塔

將上面的操作過程抽象描述整理爲以下算法。

```
1 藉助目標杆將 height - 1 個盤子移動到中間杆
2 將最後一個盤子移動到目標杆
3 藉助起始杆將 height-1 個盤子從中間杆移動到目標杆。
```

只要遵守移動規則，較大的盤子保留在棧的底部，可以使用遞歸三定律來處理任何更多的盤子。最簡單的漢諾塔就是隻有一個盤子的塔，在這種情況下，只需要將盤子移動到其最終目的地就可以了，這就是基本情況。此外，上述算法在步驟 1 和 3 中減小了漢諾塔的高度，使漢諾塔趨向基本情況。下面是使用遞歸解決漢諾塔問題的 Rust 代碼，總共也沒幾行代碼。

```
1 // move2tower.rs
2
3 // p: pole 杆
4 fn move2tower(height: u32, src_p: &str,
5                des_p: &str, mid_p: &str) {
6     if height >= 1 {
7         move2tower(height - 1, src_p, mid_p, des_p);
8         println!("moving disk from {src_p} to {des_p}");
9         move2tower(height - 1, mid_p, des_p, src_p);
```

```
10      }
11  }
12
13  fn main() {
14      move2tower(1, "A", "B", "C");
15      move2tower(2, "A", "B", "C");
16      move2tower(3, "A", "B", "C");
17      move2tower(4, "A", "B", "C");
18  }
```

你可以用幾張紙和三根筆來模擬盤在漢諾塔上的移動過程，並按照 height 等於 1，2，3，4 時 move2tower 的輸出去移動紙，看看最終是否能將所有紙移動到某一個杆上。

4.3 尾遞歸

前面的遞歸計算稱爲普通遞歸，它需要在計算過程中保存值，如代碼中的 first 和 last。我們知道，函數調用參數會保存在棧上，如果遞歸調用過多，棧會非常深。而內存又是有限的，所以遞歸存在爆棧的情況。如果這些中間值不單獨處理，而是直接傳遞給遞歸函數作爲參數，在參數傳遞時先計算，那麼棧的內存消耗不會猛增，代碼也會更簡潔。因爲所有參數都放到遞歸函數裏，且函數的最後一行一定是遞歸，所以又叫尾遞歸。

尾遞歸就是把當前的運算結果如 first，last 等變量處理過後再直接當成遞歸函數的參數用於下次調用，深層遞歸函數所面對的參數是前面多個子問題的和，這個和形式上看起來該越來越複雜，但因爲參數的和可以先求出來，又相當於減小了問題的規模，優化了算法，所有又叫尾遞歸優化。所以上面的普通遞歸版 nums 求和代碼可以改成下面的尾遞歸形式。

```
1  // nums_sum34.rs
2
3  fn nums_sum3(sum: i32, nums: &[i32]) -> i32 {
4      if 1 == nums.len() {
5          sum + nums[0]
6      } else {
7          // 使用 sum 來接收中間計算值
8          nums_sum3(sum + nums[0], &nums[1..])
9      }
10 }
11
12 fn nums_sum4(sum: i32, nums: &[i32]) -> i32 {
13     if 1 == nums.len() {
14         sum + nums[0]
```

```
15      } else {
16          nums_sum4(sum + nums[nums.len() - 1],
17                    &nums[..nums.len() - 1])
18      }
19  }
20
21  fn main() {
22      let nums = [2,1,7,4,5];
23      let sum1 = nums_sum3(0, &nums);
24      let sum2 = nums_sum4(0, &nums);
25      println!("sum1 is {sum1}, sum2 is {sum2}");
26  }
```

尾遞歸代碼是簡潔了，但是看起來更不好懂了。因爲子問題的結果需要直接當成參數輸入下一次遞歸調用。從上面的代碼也可以看出，函數最後一行纔是遞歸調用，且參數是相加的形式，較爲複雜。所以具體的遞歸程序實現要看程序員個人，如果尾遞歸不會爆棧，而且自己能寫得清晰明瞭，那麼使用尾遞歸就沒有任何問題。

4.3.1 遞歸和迭代

計算 [2,1,7,4,5] 這個數列，我們採用了循環迭代、遞歸、尾遞歸三種代碼，可見遞歸和迭代能實現相同的目的。那麼遞歸和迭代的有沒有什麼關係呢?

遞歸：用來描述以自相似方法重複事務的過程，在數學和計算機科學中，指的是在函數定義中使用函數自身的方法。

迭代：重複反饋過程的活動，每一次迭代的結果會作爲下一次迭代的初始值。

遞歸調用展開的話，是一個類似樹的結構。從字面意思可以理解爲重複遞推和回溯的過程，當遞推到達底部時就會開始回溯，其過程相當於樹的深度優先遍歷。迭代是一個環結構，從初始狀態開始，每次迭代都遍歷這個環，並更新狀態，多次迭代直到結束狀態。所有的迭代都可以轉換爲遞歸，但遞歸不一定可以轉換成迭代。畢竟環改成樹一定可以，但樹改成環卻未必能行。

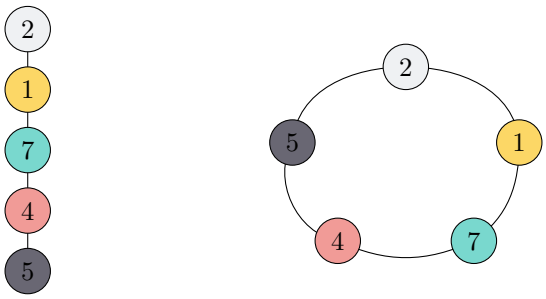


图 4.3: 遞歸和迭代

4.4 動態規劃

計算機科學中有許多求最值的問題，例如滴滴順風車要規劃兩個地點的最短路線，要找到最適合的幾段路，或找到滿足某些標准的最小道路集，這對於實現雙碳目標、節約能源以及提升乘客體驗都是至關重要的。本節的目標是向你展示幾種求最值問題的不同解決方案，並表明動態規劃是解決該類最優化問題的好辦法。

優化問題的一個典型例子是使用最少的紙幣/硬幣來找零，這在地鐵購票，自動售貨機上很常見。我們希望每個交易返回最少的紙幣張數，比如找零六元這筆交易，直接給一張五元加一張一元的共兩張紙幣，而不是向顧客吐出六張一元紙幣。

現在的問題是：怎麼從一個總任務，比如找零六元，規劃出該怎麼返回不同面值的紙幣。最直觀的就是從最大額的紙幣開始找零，先🄴可能使用大額紙幣，然後再去找下一個小一點的紙幣，並儘可能多的使用它們，直到完成找零。這種方法被稱爲貪婪方法，因爲總是試圖儘快解決大問題。

使用中國，美國貨幣時，這套貪婪法工作正常。但假設你決定在埃爾博尼亞部署自動販賣機，除了通常的 1，5，10，25 分硬幣，他們還有一個 21 分硬幣。在這種情況下，貪婪法找不到找零 63 分的最佳解決方案。隨着 21 分硬幣的加入，貪婪法仍然會找到一個解決方案，但卻有六個硬幣（25x2, 10x1, 1x3），然而最佳答案是三個 21 分。

如果採用遞歸方法，那麼找零問題能很好地解決。首先要找准基本問題，那就是找零一個幣，面額不定但是恰好等於要找零的金額，因爲剛好找零隻用一個幣，數量是除零之外的最小值了。如果金額不匹配，可以設置多個選項。爲便於讀者理解，假如使用中國紙幣來找零，有 1，5，10，20, 50 元的紙幣。對於用一元來找零的情況，找零紙幣數量等於一加上總找零金額減去一元后所需的找零紙幣數，對於 5 元的，是一加上原始金額減去五元后金額所需的紙幣數量，如果是 10 元，則是一加上總金額減去十元后所需的找零數量等等。因此，對原始金額找零紙幣數量可以根據下式計算:

$$numCoins(amount) = \begin{cases} 1 + numCoins(amount - 1) \\ 1 + numCoins(amount - 5) \\ 1 + numCoins(amount - 10) \\ 1 + numCoins(amount - 20) \\ 1 + numCoins(amount - 50) \end{cases} \tag{4.2}$$

numCoins 計算找零紙幣數量，amount 爲找零金額。按照找零紙幣的不同面額，找零任務分爲了五種情況。具體算法如下，在第 3 行，先檢查基本情況，也就是說，當前找零額度是否和某個紙幣面值等額。如果沒有等額的紙幣則遞歸調用小於找零額的不同面額的情況，此時問題規模減小了。注意，遞歸調用前先加 1，說明計算了當前正在使用的一張面額的紙幣，因爲要求的就是紙幣數量。

```
1 // rec_mc1.rs
2
```

```
3  fn rec_mc1(cashes: &[u32], amount: u32) -> u32 {
4      // 全用 1 元紙幣時的最少找零紙幣數
5      let mut min_cashes = amount;
6
7      if cashes.contains(&amount) {
8          return 1;
9      } else {
10         // 提取符合條件的幣種 (找零的幣值肯定要小於找零值)
11         for c in cashes.iter()
12                         .filter(|&c| *c <= amount)
13                         .collect::<Vec<&u32>>() {
14
15             // amount - c, 表示使用了一張面額爲 c 的紙幣
16             // 所以要加 1
17             let num_cashes = 1 + rec_mc1(&cashes, amount - c);
18
19             // num_cashes 若比 min_cashes 小則更新
20             if num_cashes < min_cashes {
21                 min_cashes = num_cashes;
22             }
23         }
24     }
25
26     min_cashes
27 }
28
29 fn main() {
30     // cashes 保存各種面額的紙幣
31     let cashes = [1,5,10,20,50];
32     let amount = 31u32;
33     let cashes_num = rec_mc1(&cashes, amount);
34     println!("need refund {cashes_num} cashes");
35 }
```

你可以將 31 改成 81，然後你發現程序沒有結果返回。實際上，程序需要非常多的遞歸調用來找到 4 張紙幣：[50，20, 10, 1] 的組合。要理解遞歸方法中的缺陷，可以看看下面這個函數遞歸調用。其中用了很多重複計算來找到正確的支付組合。程序運行中計算了大量沒用的組合，然後才能返回唯一正確的答案。圖中的下劃線加數字表示這個找零金額出現的次數，比如 11_2 表明找零 11 元這個任務第二次出現了。

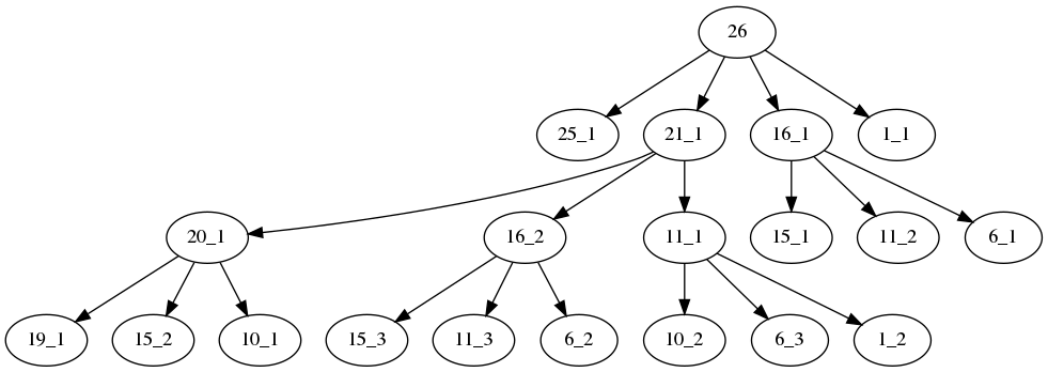


图 4.4: 遞歸求解找零任務

要減少程序的工作量，關鍵是要記住過去已經計算過的結果，這樣可以避免重複計算。一個簡單的解決方案是將當前最小數量紙幣值存儲在切片中。然後在計算新的最小值之前，首先查切片，看看結果是否存在。如果已經有結果了就直接使用切片中的值，而不是重新計算。這是算法設計中經常出現的用空間換時間的例子。

```
1  // rc_mc2.rs
2
3  fn rec_mc2(cashes: &[u32],
4             amount: u32,
5             min_cashes: &mut [u32]) -> u32 {
6      // 全用 1 元紙幣的最小找零紙幣數量
7      let mut min_cashe_num = amount;
8
9      if cashes.contains(&amount) {
10         // 收集和當前待找零值相同的幣種
11         min_cashes[amount as usize] = 1;
12         return 1;
13     } else if min_cashes[amount as usize] > 0 {
14         // 找零值 amount 有最小找零紙幣數，直接返回
15         return min_cashes[amount as usize];
16     } else {
17         for c in cashes.iter()
18                         .filter(|c| *(*c) <= amount)
19                         .collect::<Vec<&u32>>() {
20             let cashe_num = 1 + rec_mc2(cashes,
21                                         amount - c,
22                                         min_cashes);
```

```
23
24              // 更新最小找零紙幣數
25              if cashe_num < min_cashe_num {
26                  min_cashe_num = cashe_num;
27                  min_cashes[amount as usize] = min_cashe_num;
28              }
29          }
30      }
31
32      min_cashe_num
33  }
34
35  fn main() {
36      let amount = 81u32;
37      let cashes: [u32; 5] = [1,5,10,20,50];
38      let mut min_cashes: [u32; 82] = [0; 82];
39      let cashe_num = rec_mc2(&cashes, amount, &mut min_cashes);
40      println!("need refund {cashe_num} cashes");
41  }
```

新的 rec_mc 的計算就沒有那麼耗時了，因爲使用了變量 min_cashes 來保存中間值。本節是講動態規劃，然而這兩個程序都是遞歸而非動態規劃，第二個程序只是在遞歸中保存了中間值，是一種記憶手段或者緩存。

4.4.1 什麼是動態規劃

動態規劃（dynamic programming，DP）是運籌學的一個分支，是求解決策過程最優化的數學方法。上面的找零就屬於最優化問題，求的是最小紙幣數量，數量就是優化目標。

動態規劃是對某類問題的解決方法，重點在於如何肯定一類問題是動態規劃可解的而不是糾結用什麼解決方法。動態規劃中狀態是非常重要的，它是計算的關鍵，通過狀態的轉移來實現問題的求解。當嘗試使用動態規劃解決問題時，其實就是要思考如何將這個問題表達成狀態以及如何在狀態間轉移。

前文的貪婪算法是從大到小去湊值，這種算法很笨。而動態規劃總是假設當前已取得最好結果，再依據此結果去推導下一步行動。遞歸法將大問題分解爲小問題，調用自身。而動態規劃從小問題推導到大問題，推導過程的中間值要緩存起來，這個推導過程稱爲狀態轉移。比如找零問題，動態規劃先求出找零一元所需要紙幣數並保存，那麼兩元找零問題等於兩個一元找零問題，計算得到的值保存起來，接着是三元找零問題，等於兩元找零加一元找零，此時查表可得到具體值。通過這種從小到大的步驟，可以逐步構建出任何金額的找零問題。

對上面的找零問題，如果用動態規劃，那麼需要三個參數：可用紙幣列表 cs，找零金額

amount, 一個包含各個金額所需最小找零紙幣數量的列表 min_cs。當函數完成計算時, 列表內將包含從零到找零值的所有金額所需的最小找零紙幣數量。下面是實現的算法，可以看到動態規劃使用了迭代。

```
// dp_rec_mc.rs

fn dp_rec_mc(cashes: &[u32], amount: u32,
             min_cashes: &mut [u32]) -> u32 {
    // 動態收集從 1 到 amount 的最小找零幣值數量
    // 然後從小到大拼出找零紙幣數量
    for denm in 1..=amount {
        let mut min_cashe_num = denm;
        for c in cashes.iter()
                       .filter(|&c| *c <= denm)
                       .collect::<Vec<&u32>>() {
            let index = (denm - c) as usize;

            let cashe_num = 1 + min_cashes[index];
            if cashe_num < min_cashe_num {
                min_cashe_num = cashe_num;
            }
        }
        min_cashes[denm as usize] = min_cashe_num;
    }

    // 因爲收集了各個值的最小找零紙幣數，所以直接返回
    min_cashes[amount as usize]
}

fn main() {
    let amount = 81u32;
    let cashes = [1,5,10,20,50];
    let mut min_cashes: [u32; 82] =  [0; 82];
    let cash_num = dp_rec_mc(&cashes,amount,&mut min_cashes);
    println!("Refund for ${amount} need {cash_num} cashes");
}
```

動態規劃代碼是迭代，比遞歸代碼簡潔不少，不像前兩個遞歸版本算法，它減少了棧的使用。但要意識到，能爲一個問題寫遞歸解決方案並不意味着它就是最好的的解決方案。

雖然上面的動態規劃算法找出了所需紙幣最小數量，但它不顯示到底是哪些面額的紙幣。如果希望得到具體的面額，可以擴展該算法，使之記住具體使用的紙幣面額及其數量。爲此需要添加一個記錄使用紙幣的表 cashes_used。只需記住爲每個金額添加它所需的最後一張紙幣的金額到該列表，然後不斷地在列表中找前一個金額的最後一張紙幣，直到結束。

```
// dp_rc_mc_show.rs

// 使用 cashes_used 收集使用過的各面額紙幣
fn dp_rec_mc_show(cashes: &[u32], amount: u32,
    min_cashes: &mut [u32], cashes_used: &mut [u32]) -> u32 {
    for denm in 1..=amount {
        let mut min_cashe_num = denm ;
        let mut used_cashe = 1; // 最小面額是 1 元
        for c in cashes.iter()
                        .filter(|&c| *c <= denm)
                        .collect::<Vec<&u32>>() {
            let index = (denm - c) as usize;
            let cashe_num = 1 + min_cashes[index];
            if cashe_num < min_cashe_num {
                min_cashe_num = cashe_num;
                used_cashe = *c;
            }
        }

        // 更新各金額對應的最小紙幣數
        min_cashes[denm as usize] = min_cashe_num;
        cashes_used[denm as usize] = used_cashe;
    }

    min_cashes[amount as usize]
}

// 打印輸出各面額紙幣
fn print_cashes(cashes_used: &[u32], mut amount: u32) {
    while amount > 0 {
        let curr = cashes_used[amount as usize];
        println!("${curr}");
        amount -= curr;
    }
```

```
35 }
36
37 fn main() {
38     let amount = 81u32; let cashes = [1,5,10,20,50];
39     let mut min_cashes: [u32; 82] = [0; 82];
40     let mut cashes_used: [u32; 82] = [0; 82];
41     let cs_num = dp_rec_mc_show(&cashes, amount,
42                                 &mut min_cashes,
43                                 &mut cashes_used);
44     println!("Refund for ${amount} need {cs_num} cashes:");
45     print_cashes(&cashes_used, amount);
46 }
```

4.4.2 動態規劃與遞歸

遞歸是一種調用自身，通過分解大問題爲小問題以解決問題的技術，而動態規劃則是一種利用小問題解決大問題的技術。遞歸費棧，容易爆內存。動態規劃則不好找准轉移規則和起始條件，而這兩點又是必須的，所以動態規劃好用，不好理解，代碼很簡單，理解很費勁兒。同樣的問題，有時遞歸和動態規劃都能解決，比如斐波那契數列問題，用兩者都能解決。

```
1  // dp_rec.rs
2
3  fn fibnacci_rec(n: u32) -> u32 {
4      if n == 1 || n == 2 {
5          return 1;
6      } else {
7          fibnacci(n-1) + fibnacci(n-2)
8      }
9  }
10
11 fn fibnacci_dp(n: u32) -> u32 {
12     // 只用兩個位置來保存值，節約存儲空間
13     let mut dp = [1, 1];
14
15     for i in 2..=n {
16         let idx1 = (i % 2) as usize;
17         let idx2 = ((i - 1) % 2) as usize;
18         let idx3 = ((i - 2) % 2) as usize;
19         dp[idx1] = dp[idx2] + dp[idx3];
```

```
20      }
21
22      dp[((n-1) % 2) as usize]
23  }
24
25  fn main() {
26      println!("fib 10: {}", fibnacci(10));
27      println!("fib 10: {}", fibnacci_dp(10));
28  }
```

注意，能用遞歸解決的，用動態規劃不一定都能解決。因爲這兩者本身就是不同的方法，動態規劃需要滿足的條件，遞歸時不一定能滿足。這一點一定牢記，不要爲了動態規劃而動態規劃。

4.5 總結

在本章中我們討論了遞歸算法和迭代算法。所有遞歸算法都必須要滿足三定律，遞歸在某些情況下可以代替迭代，但迭代不一定能替代遞歸。遞歸算法通常可以自然地映射到所解決的問題的表達式，看起來很直觀簡潔。遞歸併不總是好的方案，有時遞歸解決方案可能比迭代算法在計算上更昂貴。尾遞歸是遞歸的優化形式，能一定程度上減少棧資源使用。動態規劃可用於解決最優化問題，通過小問題逐步構建大問題，而遞歸是通過分解大問題爲小問題來逐步解決。

Chapter 5

查找

5.1 本章目標

能夠實現順序查找，二分查找
理解哈希作爲查找技術的思想
使用 Vec 實現 HashMap 抽象數據類型

5.2 查找

在實現了多種數據結構（棧，隊列，鏈表）後，現在開始利用這些數據結構來解決一些實際問題，即查找和排序問題。查找和排序是計算機科學的重要內容，大量的軟件和算法都是圍繞這兩個任務來開展的。回憶你自己使用過的各種軟件的查找功能，應該不陌生。比如 Word 文檔裏你用查找功能來找到某些字符，在 google 瀏覽器搜索欄你鍵入要搜索的內容，然後搜索引擎返回查找到的數據。搜索和查找是一個意思，本書不加區分。

查找是在項集合中找到特定項的過程，查找通常對於項是否存在返回 true 或 false，有時它也返回項的位置。在 Rust 中，有一個非常簡單的方法來查詢一個項是否在集合中，那就是 contains() 函數。該函數字面理解是是否包含某數據的意思，但其實就是查找（search, find）。

```
1 fn main() {
2     let data = vec![1,2,3,4,5];
3     if data.contains(&3) {
4         println!("Yes");
5     } else {
6         println!("No");
7     }
8 }
```

這種查找算法很容易寫，Vec 的 contains 函數操作替我們完成了查找工作。查找算法不只有一種，事實上有很多不同的方法來進行查找，包括順序查找，二分查找，哈希查找。我們感興趣的是這些不同的查找算法如何工作以及它們的性能、複雜度如何。

5.3 順序查找

當數據項存儲在諸如 Vec，數組，切片這樣的集合中時，數據具有線性關係，因爲每個數據項都存儲在相對於其他數據項的位置。在切片中，這些相對位置是數據項的索引值。由於索引值是有序的，可以按順序訪問，所以這樣的數據結構也是線性的。回想前面學習的棧，隊列，鏈表都是線性的。基於這種和物理世界相同的線性邏輯，一種很自然的查找技術就是線性查找，或者叫順序查找。

下圖展示了這種查找的工作原理。查找從切片中的第一個項目開始，按照順序從一個項移動到另一個項，直到找到目標所在項或遍歷完整個切片。如果遍歷完整個切片後還沒找到，則說明待查找的項不存在。

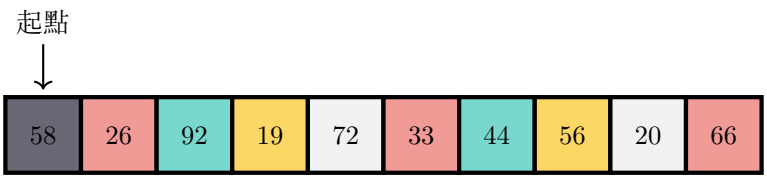


图 5.1: 順序查找

5.3.1 Rust 實現順序查找

下面是 Rust 實現的線性查找代碼，整個程序非常直觀。

```
1  // sequential_search.rs
2
3  fn sequential_search(nums: &[i32], num: i32) -> bool {
4      let mut pos = 0;
5      let mut found = false;
6
7      // found 表示是否找到
8      // pos 在索引範圍且未找到就繼續循環
9      while pos < nums.len() && !found {
10         if num == nums[pos] {
11             found = true;
12         } else {
13             pos += 1;
```

```
14          }
15      }
16
17      found
18  }
19
20  fn main() {
21      let num = 8;
22      let nums = [9,3,7,4,1,6,2,8,5];
23      let found = sequential_search(&nums, num);
24      println!("{num} is in nums: {found}");
25  }
```

當然，順序查找也可以返回查找項的具體位置，如果沒有就返回 None，所以返回值類型我們使用的 Option。

```
1   // sequential_search_pos.rs
2
3   fn sequential_search_pos(nums: &[i32], num: i32)
4       -> Option<usize> {
5       let mut pos: usize = 0;
6       let mut found = false;
7
8       while pos < nums.len() && !found {
9           if num == nums[pos] {
10              found = true;
11          } else {
12              pos += 1;
13          }
14      }
15
16      if found {
17          Some(pos)
18      } else {
19          None
20      }
21  }
22
23  fn main() {
```

```
24      let num = 8;
25      let nums = [9,3,7,4,1,6,2,8,5];
26      match sequential_search_pos(&nums, num) {
27          Some(pos) => println!("index of {num} is {pos}"),
28          None => println!("{num} is not in nums"),
29      }
30  }
```

5.3.2 順序查找複雜度

爲了分析順序查找算法複雜度，需要設定一個基本計算單位。對於查找來說，比較操作是主要的操作，所以統計比較的次數是最重要的。數據項是隨機放置，無序的，每次比較都有可能找到目標項。從概率論角度來說，數據項在集合中任何位置的概率是一樣的。

如果目標項不在集合中，知道這個結果的唯一方法是將目標與集合中所有數據項進行比較。如果有 n 個項，則順序查找需要 n 次比較，此時的複雜度是 O(n)。如果目標項在集合中，那麼複雜度是多少呢？還是 O(n) 嗎？

這種情況下，分析不如前一種情況那麼簡單。實際上有三種不同的可能。最好的情況下，目標就在集合開始處，只需要比較一次就找到目標了，此時複雜度是 O(1)，最差的情況是目標在最後，要比較 n 次才能知道，此時複雜度爲 O(n)，除此之外，目標分佈在中間，且任何位置的概率相同。此時的複雜度有 n - 2 種可能，目標在第二位，則複雜度爲 O(2)，直到 O(n-1)。綜合來看，當目標項在集合中時，查找可能比較的次數在 1 至 n 次，其複雜度平均來說等於所有可能的複雜度之和除以總的次數。

$$\sum_{i=1}^{n} O(i)/n = O(n/2) = O(n) \tag{5.1}$$

當 n 很大時，1/2 可以不考慮，可見隨機序列的順序查找複雜度就是 O(n)。假設數據項是無序的，得到的複雜度是 O(n)，如果數據不是隨機放置而是有序的，是否查找性能要好一些呢？假設數據集合按升序排列，且假設目標項存在於集合中，且在 n 個位置上的概率依舊相同。如果目標項不存在，則可以通過一些技巧來加快查找的速度。下圖展示了這個過程。假如查找目標項 50。此時，比較按順序進行，直到 56。此時，可以確定的是後面一定沒有目標值 50 了，因爲是升序排序，後面的項比 54 還大，所以不會有 50，算法停止查找。

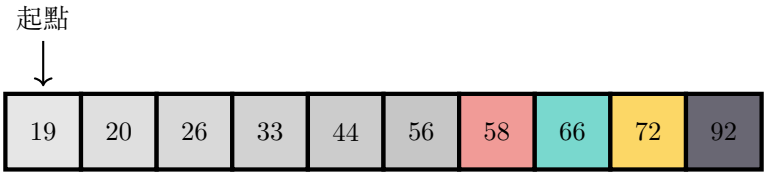


图 5.2: 有序集合順序查找

下面是用在已排序數據集上的順序查找算法，通過設置 stop 變量來控制查找超出範圍時立即停止查找以節約時間，算是對算法進行了一次優化。

```rust
// ordered_sequential_search.rs

fn ordered_sequential_search(nums: &[i32], num: i32) -> bool {
    let mut pos = 0;
    let mut found = false;
    let mut stop = false; // 控制遇到有序數據時退出

    while pos < nums.len() && !found && !stop {
        if num == nums[pos] {
            found = true;
        } else if num < nums[pos] {
            stop = true; // 數據有序，退出
        } else {
            pos += 1;
        }
    }

    found
}

fn main() {
    let nums = [1,3,8,10,15,32,44,48,50,55,60,62,64];

    let num = 44;
    let found = ordered_sequential_search(&nums, num);
    println!("{num} is in nums: {found}");

    let num = 49;
    let found = ordered_sequential_search(&nums, num);
    println!("{num} is in nums: {found}");
}
```

因爲數據排序好了，所以數據項不在集合中時，如果小於第一項，那麼比較一次就知道結果了，最差是比較 n 次，平均比較 n/2 次，複雜度還是 O(n)。但是這個 O(n) 比無序的查找好，因爲大多數情況的查找符合平均情況，而平均情況的複雜度，有序數據集查找能提升一倍速度。可見，排序對於提升查找複雜度很有幫助，所以排序也是計算機科學裏的重要議

題，我們將在下一章中學習各種排序算法。綜合無序和有序數據集合的順序查找，可得下表。

表 5.1: 順序查找複雜度

情況	最少比較次數	平均比較次數	最多比較次數	查找類型
目標存在	1	$\frac{n}{2}$	n	無序查找
目標不存在	n	n	n	無序查找
目標存在	1	$\frac{n}{2}$	n	有序查找
目標不存在	1	$\frac{n}{2}$	n	有序查找

5.4 二分查找

有序數據集合對於查找算法是很有利的。在有序集合中順序查找時，當與第一個項進行比較，如果第一項不是要查找的，則還有 n-1 項待比較。當遇到超過範圍的值時，可以停止查找，綜合來看這種有序查找速度還是比較慢。對於排好序的數據集有沒有更快的查找算法呢? 當然有，那就是二分查找，這是一個非常重要的查找算法，下面來仔細分析並用 Rust 實現二分查找算法。

5.4.1 Rust 實現二分查找

其實二分查找的意思體現在其名字上了，說白了就是把數據集分成兩部分來查找，通過 low，mid，high 來控制查找的範圍。從中間項 mid 開始，而不是按順序查找。如果中間項是正在尋找的項，則完成了查找。如果它不是，可以使用排序集合的有序性質來消除一半剩余項。如果正在查找的項大於中間項，就可以消除中間項以及比中間項小的一半元素，也就是第一項到中間項都可以不用去比較了。因爲目標項大於中間值，那麼不管是否在集合中，那肯定不會在前一半。相反，若是目標項小於中間項，則後面的一半數據就都可以不用比較了。去除了一半數據後在剩下的一半數據項裏再查看其中間項，重複上述的比較和省略過程，最終得到結果。這個查找速度是比較快的，而且也非常形象，所以叫二分查找。

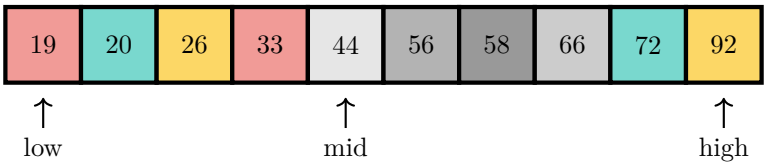


图 5.3: 二分查找

二分查找示意圖如上圖。初始時 low 和 high 分居最左和最右。比如你要查 60，那麼先和中間值比較，大於 44，則 low 移動到 56 處，mid 移動到 66。此時和 66 比較，發現小於 66，所以 high 移動到 58，mid 移動到 56 和 low 重合了。比較發現大於 56，所以 low 移動到 58，mid 也移動到 58，發現大於 58。此時 low, high, mid 三者在一個位置。接着 low 移

動到 66，發現下標大於 high 了，不滿足條件，查找停止，退出。根據上面的描述和示意圖，可以用 Rust 寫出如下的二分查找代碼。

```rust
// binary_search.rs
fn binary_search1(nums: &[i32], num: i32) -> bool {
    let mut low = 0;
    let mut high = nums.len() - 1;
    let mut found = false;

    // 注意是 <= 不是 <
    while low <= high && !found {
        let mid: usize = (low + high) >> 1;

        // 若 low + high 可能溢出，可用如下的形式
        //let mid: usize = low + ((high - low) >> 1);

        if num == nums[mid] {
            found = true;
        } else if num < nums[mid] {
            high = mid - 1; // num 小於中間值，省去後半部數據
        } else {
            low = mid + 1;  // num 大於等於中間值，省去前半部數據
        }
    }

    found
}

fn main() {
    let nums = [1,3,8,10,15,32,44,48,50,55,60,62,64];

    let target = 3;
    let found = binary_search1(&nums, target);
    println!("{target} is in nums: {found}");

    let target = 63;
    let found = binary_search1(&nums, target);
    println!("{target} is in nums: {found}");
}
```

二分法其實是把大問題分解成小問題，採取的是分而治之策略。前面我們學習了分解大問題爲小問題用遞歸來解決，所以二分法和遞歸是否有相似，二分法是否能用遞歸實現呢?

我們發現二分查找時，找到或沒找到是最終的結果，是一個基本情況。而二分法不斷減小問題的尺度，不斷向基本情況靠近，且二分法是在不斷重複自身步驟。所以二分法滿足遞歸三定律，所以可以用遞歸實現二分法，具體實現如下。

```
// binary_search.rs

fn binary_search2(nums: &[i32], num: i32) -> bool {
    // 基本情况1: 項不存在
    if 0 == nums.len() { return false; }

    let mid: usize = nums.len() >> 1;

    // 基本情况2: 項存在
    if num  == nums[mid] {
        return true;
    } else if num < nums[mid] {
        // 縮小問題規模
        return binary_search2(&nums[..mid], num);
    } else {
        return binary_search2(&nums[mid+1..], num);
    }
}

fn main() {
    let nums = [1,3,8,10,15,32,44,48,50,55,60,62,64];

    let num = 3;
    let found = binary_search2(&nums, num);
    println!("{num} is in nums: {found}");

    let num = 63;
    let found = binary_search2(&nums, num);
    println!("{num} is in nums: {found}");
}
```

遞歸實現的查找涉及數據集合的切片，注意再查找時 mid 項需要舍去，所以使用了 mid + 1。遞歸算法總是涉及棧的使用，有爆棧風險，一般來說二分查找最好用迭代法來解決。

5.4.2 二分查找複雜度

二分查找算法最好的情況是中間項就是目標，此時複雜度爲 O(1)。二分查找每次比較能消除一半的剩余項，可以計算最多的比較次數得到該算法的最差複雜度。第一次比較後剩 $n/2$，第二次比較後剩余 $n/4$，直到 $n/8$，$n/16$，$n/2^i$ 等等。當 $n/2^i = 1$ 時，二分結束。所以

$$\begin{aligned} \frac{n}{2^i} &= 1 \\ i &= log_2(n) \end{aligned} \tag{5.2}$$

所以二分查找算法最多比較 $log_2(n)$ 次，複雜度也就是 $O(log_2(n))$，這是一個比 O(n) 算法還要優秀的算法。但要注意，在上述的遞歸版實現中，默認的棧使用是會消耗內存的，複雜度不如迭代版。

二分查找看起來很好，但如果 n 很小，就不值得去排序再使用二分，此時直接使用順序查找可能複雜度還更好。此外，對於很大數據集，對其排序很耗時和耗內存，那麼直接採用順序查找複雜度可能也更好。好在實際項目中大量的數據集即不多也不少，非常適合二分查找，這也是我們花大量篇幅來闡述二分查找算法的原因。

5.4.3 內插查找

內插查找是一種二分查找的變形，適合在排序數據中進行查找。如果數據是均分的，則可以使用內插查找快速逼近待搜索區域，從而提高效率。

內插查找不是像二分查找算法中那樣直接使用中值來定界，而是通過插值算法找到上下界。回憶中學學過的線性內插法，給定直線兩點 (x_0, y_0) 和 (x_1, y_2)，可以求出 $[x_0, x_1]$ 範圍內任意點 x 對應的值 y 或者任意 y 對應的 x.

$$\begin{aligned} \frac{y - y_0}{x - x_0} &= \frac{y_1 - y_0}{x_1 - x_0} \\ x &= \frac{(y - y_0)(x_1 - x_0)}{y_1 - y_0} + x_0 \end{aligned} \tag{5.3}$$

比如要在 [1,9,10,15,16,17,19,23,27,28,29,30,32,35] 這個已排序的 14 個元素的集合中查找到元素 27，那麼可以將索引當做 x 軸，元素值當做 y 軸。可知 $x_0 = 0, x_1 = 13$，而 $y_0 = 1, y_1 = 35$。所以可以計算 $y = 27$ 對應的 x 值。

$$\begin{aligned} x &= \frac{(27 - 1)(13 - 0)}{35 - 1} + 0 \\ x &= 9 \end{aligned} \tag{5.4}$$

查看 nums[9] 發現值爲 28，大於 27，所以將 28 當做上界。28 的下標爲 9，所以搜索 [0,8] 範圍內的元素，繼續執行插值算法。

$$\begin{aligned} x &= \frac{(27 - 1)(8 - 0)}{27 - 1} + 0 \\ x &= 8 \end{aligned} \tag{5.5}$$

查看 nums[8] 發現值恰爲 27，找到目標，算法停止。具體實現代碼如下。

```rust
// interpolation_search.rs

fn interpolation_search(nums: &[i32], target: i32) -> bool {
    if nums.is_empty() { return false; }

    let mut low  = 0usize;
    let mut high = nums.len() - 1;
    loop {
        let low_val  = nums[low];
        let high_val = nums[high];

        if high <= low || target < low_val
            || target > high_val {
            break;
        }

        // 計算插值位置
        let offset = (target - low_val)*(high - low) as i32
                     / (high_val - low_val);
        let interpolant = low + offset as usize;

        // 更新上下界 high、low
        if nums[interpolant] > target {
            high = interpolant - 1;
        } else if nums[interpolant] < target {
            low = interpolant + 1;
        } else {
            break;
        }
    }

    // 判斷最終確定的上界是否是 target
    if target == nums[high] {
        true
    } else {
        false
    }
```

```
38 }
39
40 fn main() {
41     let nums = [1,9,10,15,16,17,19,23,27,28,29,30,32,35];
42     let target = 27;
43     let found = interpolation_search(&nums, target);
44     println!("{target} is in nums: {found}");
45 }
```

仔細分析代碼可以發現，除了 interpolant 的計算方式不同外，其他的和二分查找算法幾乎一樣。內插查找算法在數據均分時複雜度是 $O(loglog(n))$，具體證明較複雜，請看此論文 [10] 瞭解，最差和平均複雜度均是 $O(n)$。

5.4.4 指數查找

指數查找是另一種二分查找的變體，它劃分中值的方法不是使用平均或插值而是用指數函數來估計，這樣可以快速找到上界，加快查找，該算法適合已排序且無邊界的數據。算法查找過程中不斷比較 $2^0, 2^1, 2^2, 2^k$ 位置上的值和目標值的關係，進而確定搜索區域，之後在該區域內使用二分查找算法查找。

假設要在 [2,3,4,6,7,8,10,13,15,19,20,22,23,24,28] 這個 15 個元素已排序集合中查找 22，那麼首先查看 $2^0 = 1$ 位置上的數字是否超過 22，得到 $3 < 22$，所以繼續查找 $2^1, 2^2, 2^3$ 位置處元素，發現對應的值 4, 7, 15 均小於 22。繼續查看 $16 = 2^4$ 處的值，可是 16 大於集合元素個數，超出範圍了，所以查找上界就是最後一個索引 14。

下面是實現的指數搜索代碼。注意 14 行的下界是 high 的一半，此處用的是移位操作。我們能找到一個上界，那麼說明前一次訪問處一定小於待查找的值，作爲下界是合理的。當然用 0 作下界也可以，但是效率就低了。

```
1  // exponential_search.rs
2
3  fn exponential_search(nums: &[i32], target: i32) -> bool {
4      let size = nums.len();
5      if size == 0 { return false; }
6
7      // 逐步找到上界
8      let mut high = 1usize;
9      while high < size && nums[high] < target {
10         high <<= 1;
11     }
12
13     // 上界的一半一定可以作爲下界
```

```
14      let low = high >> 1;
15
16      // 使用前面實現的二分查找
17      binary_search(&nums[low..size.min(high+1)], target)
18  }
19
20  fn main() {
21      let nums = [1,9,10,15,16,17,19,23,27,28,29,30,32,35];
22      let target = 27;
23      let found = exponential_search(&nums, target);
24      println!("{target} is in nums: {found}");
25  }
```

通過分析發現，指數查找分爲兩部分，第一部分是找到上界用於劃分區間，第二部分是二分查找。劃分區間的複雜度和查找目標 i 相關，其複雜度爲 $O(logi)$，而二分查找時複雜度爲 $O(logn)$，n 爲查找區間長度，在這裏可以知道區間長度爲 $high - low = 2^{logi} - 2^{logi-1} = 2^{logi-1}$，所以複雜度爲 $O(log(2^{logi-1})) = O(logi)$，最後總的複雜度爲 $O(logi+logi) = O(logi)$

5.5 哈希查找

前面的查找算法都是利用項在集合中相對於彼此存儲的位置信息來進行查找。通過排序集合，可以使用二分查找在對數時間內查找到數據項。這些數據項在集合中的位置信息由集合的有序性質提供，查找算法無從得知，所以要不斷比較。

如果我們的算法能對不同的項保存地址有先驗知識，那麼查找時就不用依次比較，而是可以直接獲取。這種通過數據項直接獲得其保存地址的方法稱爲哈希查找 (Hash Search)，是一種複雜度爲 O(1) 的查找算法，也就是最快的查找算法。

爲了做到這一點，當在集合中查找項時，項地址要首先存在，所以我們得提供一個保存項且獲取地址方便的數據結構，這就是哈希表 (散列表)。哈希表以容易找到數據項的方式存儲着數據項，每個數據項位置通常稱爲一個槽 (地址)。這些槽可以從 0 開始命名，當然也可以是其他的數字。一旦選定首個槽的命名，那麼接着所有的槽名都相應的加一。最初，哈希表不包含項，因此每個槽都爲空。可以通過 Vec 來實現一個哈希表，每個元素初始化爲 None。下圖展示了大小 m = 11 的哈希表，換句話說，在表中有 m 個槽，命名爲 0 到 10。

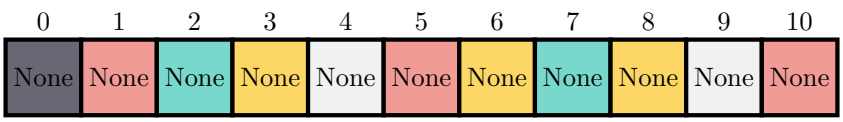
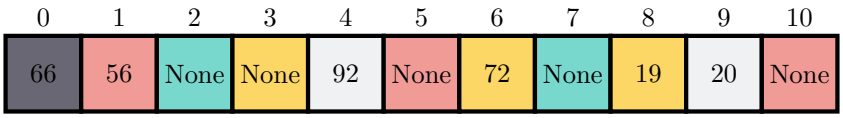


图 5.4: 哈希表

數據項及其在哈希表中所屬槽之間的映射關係被稱爲 hash 函數或散列函數。hash 函數

接收集合中的任何項，並返回具體的槽名，這個動作稱爲哈希或散列。假設有整數項 [24, 61, 84, 41, 56, 31]，和一個容量爲 11 的哈希槽，那麼只要將每個項輸入到哈希函數，就能得到它們在哈希表中的位置。一種簡單的哈希函數就是求余，因爲任何數對 11 求余，余數一定在 11 以內，也就是在 11 個槽範圍內，這能保證數據總是有槽來存放。

$$hash(item) = item\%11$$



一旦計算了哈希值，就可以將項插入到指定位置的槽中，如上圖所示。注意，11 個插槽中的 6 個現在已被佔用，此時哈希表的負載可用佔用的槽除以總槽數，該比值稱爲負載因子，用 λ 表示，其計算方式是 λ = 項數/表大小，此處就是 $\lambda = 6/11$。這個負載因子可以作爲評估指標，尤其是程序需要保存很多項時。若是負載因子太大，那麼剩下的位置就不夠，所以可以根據負載因子控制是否要擴容。在 Rust，Go 等語言中都是通過這種機制來擴容的，負載因子超過一個閾值，哈希表就開始擴容，爲後面插入數據作准備。

從圖中可見哈希表保存的數據不是有序的，相反，它是無序的，而且非常亂。我們有哈希函數，不管它怎麼亂，都可以通過計算哈希值獲取數據項的槽。比如要查詢 66 是否存在，那麼通過哈希計算，得到 hash(66) = 0，查看槽 0，發現爲 66，所以 66 存在。該查找操作複雜度爲 O(1)，因爲只用在恆定時間算出槽位置並查看。

哈希表及哈希查找非常優秀，但也要注意衝突。假如現在加入 99，那麼 hash(99) = 0，而槽 0 處是 66，不等於 99，此時哈希槽出現衝突。衝突必須解決，不然哈希表就無法使用。

5.5.1 哈希函數

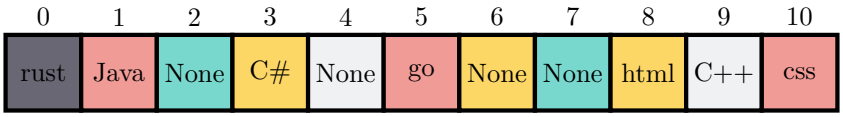
上節使用的哈希函數是直接對項求余，使得余數在一定範圍內。可見一個算法只要能根據項求得一個在一定範圍內的數，那麼這個算法就可以看成是哈希函數。通過對哈希函數的改進，我們能減小衝突的概率，實現一個可用的哈希表。

第一種改進方法是分組求和法，它將項劃分爲相等大小的塊，（最後一塊可能不等），然後將這些項加起來再求余。例如，如果數據項是電話號碼 316-545-0134，可以將號碼分成兩位數，不足補 0。那麼可以得到 [31,65,45,01,34]，將其求和得 176，再對 11 求余得到哈希值（槽）爲 0。當然，號碼也可以分成三位數，甚至反轉數字，最後再求余。哈希函數的種類非常多，因爲只要能得出一個值再求余就行，而這個值可以採用各種方法得到。

分組求和法可以求哈希值，平法取中法也可以，這是另一種哈希算法。首先對數據項求平方，然後提取平方的中間部分作爲值去求余。比如數字 36，平方爲 1296，取中間部分 29，求余得到 hash(29) = 7，所以 36 應該保存在槽 7 處。

如果保存字符串，還可以基於字符的 ascii 值求余。字符串 “rust” 包含四個字符，其 ascii 值分別爲 [114，117，115，116]，求和得到 462，求哈希的 hash(462) = 0。當然這 rust 字符

串看起來很巧，居然就是 114 - 117。我們可以選擇其他的字符串試試，比如 Java，其 ascii 值爲 [74,97,118,97]，求和得 386，求哈希 hash(386) = 1，所以 Java 該保存在槽 1，如下圖。



ASCII 哈希函數具體如下。

```
// hash.rs

fn hash1(astr: &str,  size: usize) -> usize {
    let mut sum = 0;
    for c in astr.chars() {
        sum += c as usize;
    }

    sum % size
}

fn main() {
    let size = 11;
    let s1 = "rust"; let s2 = "Rust";
    let p1 = hash1(s1, size);
    let p2 = hash1(s2, size);
    println!("{s1} in slot {p1}, {s2} in slot {p2}");
}
```

使用這個函數時，衝突比較嚴重，所以可以稍微修改一下。比如使用 “rust” 中不同字符的位置爲權重，考慮將其 ascii 值乘以其位置權重。

$$\begin{aligned} hash(rust) &= (0*114 + 1*117 + 2*115 + 3*116)\%11 \\ &= 695\%11 \\ &= 2 \end{aligned} \tag{5.6}$$

當然，下標不一定從 0 開始，從 1 開始比較好，因爲從 0 開始，那麼第一個字符對總和沒有影響。具體計算和代碼如下。

$$\begin{aligned} hash(rust) &= (1*114 + 2*117 + 3*115 + 4*116)\%11 \\ &= 1157\%11 \\ &= 2 \end{aligned} \tag{5.7}$$

```rust
// hash.rs

fn hash2(astr: &str,  size: usize) -> usize {
    let mut sum = 0;
    for (i, c) in astr.chars().enumerate() {
        sum += (i + 1) * (c as usize);
    }
    sum % size
}
```

重要的是，哈希函數必須非常高效，以免其耗時成爲主要部分。如果哈希函數太複雜，甚至比順序或二分查找還耗時，這將打破哈希表的 O(1) 複雜度，那就得不償失了。

5.5.2 解決衝突

前面我們都沒有處理哈希衝突的問題，比如上面以位置爲權重的 hash 函數，若下標從 0 開始，則 “rust” 和 “Rust” 字符串會發生衝突。當兩項散列到同一個槽時，必須以某種方法將第二項放入哈希表中，這個過程稱爲衝突解決。

若哈希函數是完美的，則永遠不會發生衝突。然而，由於內存有限且真實情況複雜，完美的哈希表不存在。解決衝突的一種直觀方法就是查找哈希表，嘗試找到下一個空槽來保存衝突項。最簡單的方法是從原哈希衝突處開始，以順序方式移動槽，直到遇到第一個空槽。注意，遇到末尾後可以循環從頭開始查找。這種衝突解決方法被稱爲開放尋址法，具體是線性探測法，它試圖在哈希表中線性地探測到下一個空槽。下圖是槽爲 11 的哈希表。

0	1	2	3	4	5	6	7	8	9	10
None	56	24	None	None	None	61	84	41	31	None

當我們嘗試插入 35 時，其位置應爲槽 2，可我們發現槽 2 存在 24，所以此時從槽 2 開始查找空槽，發現槽 3 空的，所以在此處插入 35。

0	1	2	3	4	5	6	7	8	9	10
None	56	24	35	None	None	61	84	41	31	None

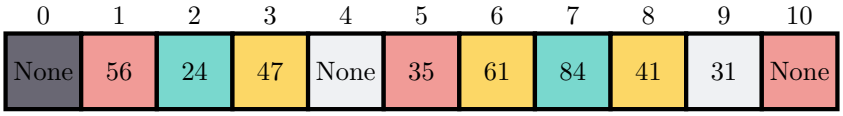
再插入 47，發現槽 3 有 35，所以查找下一個空槽，發現槽 4 爲空，所以插入 47。

0	1	2	3	4	5	6	7	8	9	10
None	56	24	35	47	None	61	84	41	31	None

一旦使用開放尋址法建立了哈希表，後面就必須遵循相同的方法來查找項。假設想查找項 56，計算哈希爲 1，查詢表發現正是 56，所以返回 true。如果查找 35，計算哈希得 2，查表發現是 24，不是 35，此時不能返回 false。因爲可能發生過衝突，所以要順序查找，直到找到 35 或空槽或循環一圈再回到 24 時才能停止並返回結果。

線性查找的缺點是數據項聚集。項在表中聚集，這意味着如果在相同哈希槽處發生多次衝突，則將通過線性探測來填充多個後續槽，結果原本該插入這些槽的值被迫插入其他地方，而這種順序查找非常費時，複雜度就不是 O(1)。比如 35 和 47 連續衝突。

處理數據項聚集的一種方式是擴展開放尋址技術，發生衝突時不是順序查找下一個開放槽，而是跳過若干個槽，從而更均勻地分散引起衝突的項。比如加入 35 時，發生衝突，那麼從此處開始，查看第三個槽，也就是每次隔三個槽來查看，這樣就將衝突分散開了。此時再插入 47，就沒有衝突了，這種方式有一定效果，能緩解數據聚集。



在衝突後尋找另一個槽的過程叫重哈希（重散列, rehash），其計算方法如下：

$$rehash(pos) = (pos + n)\%size$$

要注意，跳過的大小必須使得表中的所有槽最終都能被訪問。爲確保這一點，建議表大小是素數，這也爲什麼示例中要使用 11。

解決衝突的另一種方法是拉鏈法，也就是說對每個衝突的位置，我們設置一個鏈表來保存數據項，如圖（5.5）。查找時，發現衝突後就再到鏈上順序查找，複雜度爲 $O(n)$。當然，衝突鏈上的數據可以排序，然後再借助二分查找，這樣哈希表複雜度爲 $O(log_2(n))$。如果拉鏈太長，還可以將鏈改成樹，這樣其結構會更加穩定。拉鏈法是許多編程語言內置的哈希表數據結構解決衝突的默認實現。

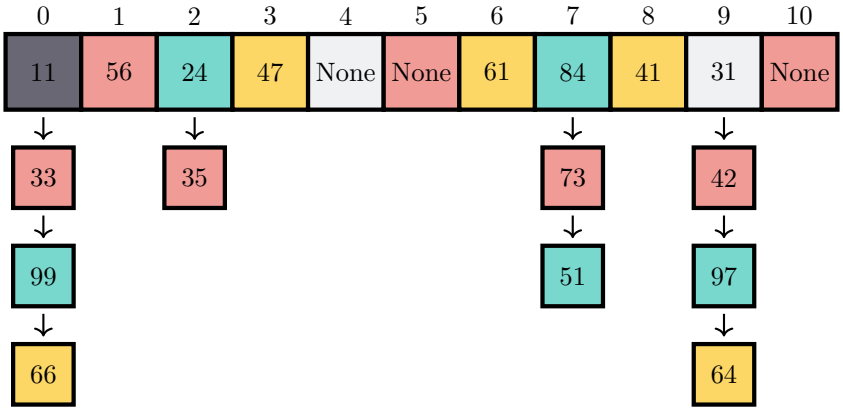


图 5.5: 拉鏈法解決衝突

5.5.3 Rust 實現 HashMap

Rust 集合類型中最有用的是 HashMap，它是一種關聯數據類型，可以在其中存儲鍵值對。鍵用於查找位置，因爲數據位置不定，這種查找類似在地圖（map）上查找一樣，所以這種數據結構稱爲 HashMap。

HashMap 的抽象數據類型定義如下。該結構是鍵值間關聯的無序集合，其中的鍵都是唯一的，鍵值間存在一對一的關係。

new() 創建一個新的 HashMap，不需要參數，返回一個空的 HashMap 集合。

insert(k,v) 向 HashMap 中添加一個新的鍵值對，需要參數 k、v，如果鍵存在，那麼用新值 v 替換舊值。

remove(k) 從 HashMap 中刪除某個 k，需要參數 k，返回 k 對應的 v。

get(k) 給定鍵 k，返回存儲在 HashMap 中的值 v（可能爲空 None），需要參數 k。

contains(k) 如果鍵 k 存在，則返回 true，需要參數 k。

len() 返回存儲在 HashMap 中的鍵值對數量，不需要參數。

這裏採用兩個 Vec (slot、data) 來分別保存鍵和值，data 保存數據，slot 保存鍵，下標從 1 開始，0 爲槽中默認值，HashMap 由一個 struct 封裝。

```
// hashmap.rs

// 用 slot 保存位置，data 保存數據
#[derive(Debug, Clone, PartialEq)]
struct HashMap <T> {
    size: usize,
    slot: Vec<usize>,
    data: Vec<T>,
}
```

重哈希 rehash 設置爲加 1 的線性探索方法。初始大小固定爲 11，當然也可以是其他素數值，比如 13，17，19，23，29 等。下面是 HashMap 的實現代碼。

```
// hashmap.rs

impl<T: Clone + PartialEq + Default> HashMap<T> {
    fn new() -> Self {
        // 初始化 slot 和 data
        let size = 11usize;
        let mut slot = Vec::with_capacity(size);
        let mut data = Vec::with_capacity(size);
        for _i in 0..size {
            slot.push(0);
            data.push(Default::default());
```

```
12          }
13
14          HashMap { size, slot, data }
15      }
16
17      fn hash(&self, key: usize) -> usize {
18          key % self.size
19      }
20
21      fn rehash(&self, pos: usize) -> usize {
22          (pos + 1) % self.size
23      }
24
25      fn insert(&mut self, key: usize, value: T) {
26          if 0 == key { panic!("Error: key mut > 0"); }
27
28          let pos = self.hash(key);
29          if 0 == self.slot[pos] { // 槽無數據直接插入
30              self.slot[pos] = key;
31              self.data[pos] = value;
32          } else { // 插入槽有數據再找下一個可行的位置
33              let mut next = self.rehash(pos);
34              while 0 != self.slot[next]
35                  && key != self.slot[next] {
36                  next = self.rehash(next);
37              }
38
39              // 在找到的槽插入數據
40              if 0 == self.slot[next] {
41                  self.slot[next] = key;
42                  self.data[next] = value;
43              } else {
44                  self.data[next] = value;
45              }
46          }
47      }
48
49      fn remove(&mut self, key: usize) -> Option<T> {
```

```
50          if 0 == key { panic!("Error: key mut > 0"); }
51
52          let pos = self.hash(key);
53          if 0 == self.slot[pos] { // 槽中無數據，返回 None
54              None
55          } else if key == self.slot[pos] {
56              self.slot[pos] = 0; // 更新 slot 和 data
57              let data = Some(self.data[pos].clone());
58              self.data[pos] = Default::default();
59
60              data
61          } else {
62              let mut data: Option<T>  = None;
63              let mut stop = false;
64              let mut found = false;
65              let mut curr = pos;
66
67              while 0 != self.slot[curr] && !found && !stop {
68                  if key == self.slot[curr] { // 找到了值，剔除數據
69                      found = true;
70                      self.slot[curr] = 0;
71                      data = Some(self.data[curr].clone());
72                      self.data[curr] = Default::default();
73                  } else {
74                      curr = self.rehash(curr);
75                      // 再哈希回到最初位置，表明找了一圈也未找到
76                      if curr == pos { stop = true; }
77                  }
78              }
79
80              data
81          }
82      }
83
84      fn get(&self, key: usize) -> Option<&T> {
85          if 0 == key { panic!("Error: key mut > 0"); }
86
87          // 計算數據位置
```

```
88          let pos = self.hash(key);
89          let mut data: Option<&T> = None;
90          let mut stop = false;
91          let mut found = false;
92          let mut curr = pos;
93
94          // 循環查找數據
95          while 0 != self.slot[curr] && !found && !stop {
96              if key == self.slot[curr] {
97                  found = true;
98                  data = self.data.get(curr);
99              } else {
100                 curr = self.rehash(curr);
101
102                 // 再哈希回到了最初位置，表明找了一圈還未找到
103                 if curr == pos {
104                     stop = true;
105                 }
106             }
107         }
108
109         data
110     }
111
112     fn contains(&self, key: usize) -> bool {
113         if 0 == key { panic!("Error: key mut > 0"); }
114         self.slot.contains(&key)
115     }
116
117     fn len(&self) -> usize {
118         let mut length = 0;
119         for d in self.slot.iter() {
120             if &0 != d  { // 槽數據不爲 0，表示有數據，len + 1
121                 length += 1;
122             }
123         }
124
125         length
```

```
126         }
127     }
128
129     fn main() {
130         let mut hmap = HashMap::new();
131         hmap.insert(10,"cat");
132         hmap.insert(2,"dog");
133         hmap.insert(3,"tiget");
134
135         println!("HashMap size {:?}", hmap.len());
136         println!("HashMap contains key 2: {}", hmap.contains(2));
137         println!("HashMap key 3: {:?}", hmap.get(3));
138         println!("HashMap remove key 3: {:?}", hmap.remove(3));
139         println!("HashMap remove key 3: {:?}", hmap.remove(3));
140     }
```

5.5.4 HashMap 複雜度

最好的情況下，哈希表提供 O(1) 的查找複雜度。然而，由於衝突，查找過程中比較的數量通常會變動。這種衝突越劇烈，則性能越差。一種好的評估指標是負載因子 λ。如果負載因子小，則碰撞的機會低，這意味着項更可能在它們所屬的槽中。如果負載因子大，意味着鏈快填滿了，則存在越來越多的衝突。這意味着衝突解決更復雜，需要更多的比較來找到一個空槽。即便使用拉鏈法，衝突增加意味着鏈上的項數增加，則查找的時候鏈上查找花費時間占主要部分。

每次查找的結果是成功或不成功。對於使用線性探測的開放尋址法進行的成功查找，平均比較次數大約爲 $\frac{1+\frac{1}{1-\lambda}}{2}$，不成功時查找次數大約爲 $\frac{1+(\frac{1}{1-\lambda})^2}{2}$ 。即便使用拉鏈法，對於成功的情況，平均比較數目是 $1+\lambda/2$，查找不成功，則是 λ 次比較。總體來說哈希表的查找複雜度爲 $O(\lambda)$ 左右。

5.6 總結

本章我們學習了查找算法，包括順序查找，二分查找和哈希查找。順序查找是最簡單和直觀的查找算法，其複雜度爲 O(n)，二分查找算法每次都去掉一半數據，速度比較快，但要求數據集有序，複雜度爲 $O(log_2(n))$。基於二分查找衍生了類似內插查找和指數查找這樣的算法，它們適合的數據分佈類型不同。哈希查找是利用 HashMap 實現的一種 O(1) 查找算法，要注意的是哈希表容易衝突，需要採取合理措施解決衝突，比如開放尋址法、拉鏈法。學習查找算法的過程中我們發現排序對於查找算法加快速度很有幫助，所以下一章我們來學習排序算法。

Chapter 6

排序

6.1 本章目標

學習瞭解排序思想
能用 Rust 實現十大基本排序算法

6.2 什麼是排序

排序是以某種順序在集合中放置元素的過程。例如，鬥地主時大家拿到自己的牌都會抽來抽去的將各種牌按順序或花色放到一起，這樣便於思考和出牌。一堆散亂的單詞可以按字母順序或長度排序。中國的城市則可按人口、種族、地區排序。

排序是計算機科學的一個重要研究領域，大量領域內前輩對排序作出了傑出的貢獻，推動了計算機科學的發展。我們已經看過許多能夠從排序數據集合中獲益的算法，包括順序查找和二分算法等，這也說明了排序的重要性。

與查找算法一樣，排序算法的效率與處理的項數有關。對於小集合，複雜的排序方法開銷太高。另一方面，對於大的數據集，簡單的算法性能又太差。在本章中，我們將討論多種排序算法，並對它們的運行性能進行比較。

在分析特定算法之前，首先要知道排序設計的核心操作是比較。因爲序列、順序是靠比較得出來的，這和人的幸福感一樣，靠比較才知道自己有多幸福。比較就是查看哪個更大，哪個更小。其次，如果比較發現順序不對，就涉及數據位置交換。交換是一種昂貴的操作，交換的總次數對於評估算法的效率很關鍵。前面我們學習過大 O 分析法，本章的所有算法都要用大 O 分析法來分析性能，這是評價排序算法最直觀的指標。

此外，排序還存在穩定與否的問題。例如 [1,4,9,8,5,5,2,3,7,6] 中，有兩個 5，那麼不同的排序可能會將兩個 5 交換順序，雖然它們最終是挨着的，但已經破壞了原有的次序關係。當然，對於純數字是無所謂的，但若是這些數字是某個數據結構的某個鍵，例如下面這樣的結構，包含個人信息，雖然其 amount 參數都是 5，但顯然兩個人的姓名和年齡都不同，貿然排序會改變次序。

```
people {
    amount: 5,
    name: 張三,
    age: 20,
}

people {
    amount: 5,
    name: 王五,
    age: 24,
}
```

張三如果原本在前，排序後排到後面了，這就會出現問題，尤其是有的算法依賴序列的穩定性，比如扣款這樣的操作。所以評價排序算法除了時間空間複雜度，還要看穩定性。

對集合的排序，存在各種各樣的排序算法。但有十類排序算法是各種排序算法的基礎，它們是：冒泡排序，快速排序，選擇排序，堆排序，插入排序，希爾排序，歸併排序，計數排序，桶排序，基數排序。

除此之外，基於十類基礎算法還衍生了許多改進的算法，本文選擇了部分進行講解，包括雞尾酒排序、梳子排序、二分插入排序、Flash 排序、蒂姆排序。

6.3 冒泡排序

冒泡排序需要多次遍歷集合，它比較相鄰的項並交換那些無序的項。每次遍歷列表都將最大的值放在其正確的位置。這就類似燒開水時，壺底的水泡往上冒的過程，這也是它叫冒泡排序的原因。

下一頁的圖（6.1）展示了冒泡排序的第一次遍歷，陰影項正在比較是否亂序。如果在列表中有 n 項，則第一遍有 n-1 次項比較。在第二次遍歷數據的時候，數據集中的最大值已經在正確的位置，剩下的 n-1 個數據還需要排序，這意味着將有 n-2 次比較。由於每次通過將下一個最大值放置在適當位置，所需的遍歷的總數將是 n-1。在完成 n-1 輪遍歷比較後，最小項肯定在正確的位置，不需要進一步處理。

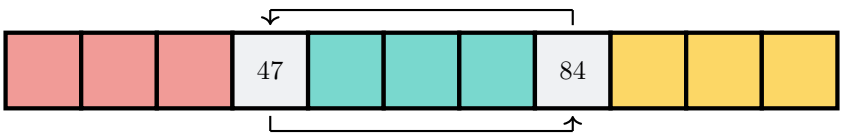
仔細看對角線，可以發現是最大值，而且它在不斷往最右側移動。冒泡排序涉及頻繁的交換操作（swap），交換是比較中常用的輔助操作，在 Rust 中 Vec 數據結構就默認實現了 swap 函數，當然你也可以實現如下的交換操作。

```
// swap
let temp = data[i];
data[i] = data[j];
data[j] = temp;
```

84	92	66	56	44	31	72	19	24
84	92	66	56	44	31	72	19	24
84	66	92	56	44	31	72	19	24
84	66	56	92	44	31	72	19	24
84	66	56	44	92	31	72	19	24
84	66	56	44	31	92	72	19	24
84	66	56	44	31	72	92	19	24
84	66	56	44	31	72	19	92	24
84	66	56	44	31	72	19	24	92

图 6.1: 冒泡排序

有些語言，如 Python，可以不用臨時變量便直接交換值，如：data[i], data[j] = data[j], data[i]，這是語言實現的特性，其內部還是使用了變量，只不過是同時操作兩個。



結合上所述，我們可以用 Vec 來實現冒泡排序。注意，爲簡化算法設計，本章中所有待排序集合裏保存的都是數字。

```
1  // bubble_sort.rs
2
3  fn bubble_sort1(nums: &mut [i32]) {
4      if nums.len() < 2 {
5          return;
6      }
7
8      for i in 1..nums.len() {
9          for j in 0..nums.len()-i {
10             if nums[j] > nums[j+1] {
```

```
11                  nums.swap(j, j+1);
12              }
13          }
14      }
15  }
16
17  fn main() {
18      let mut nums = [54,26,93,17,77,31,44,55,20];
19      bubble_sort1(&mut nums);
20      println!("sorted nums: {:?}", nums);
21  }
```

除用 for 循環外，也可用 while 循環實現冒泡排序。這兩程序均在 bubble_sort.rs 中。

```
1   // bubble_sort.rs
2
3   fn bubble_sort2(nums: &mut [i32]) {
4       let mut len = nums.len() - 1;
5       while len > 0 {
6           for i in 0..len {
7               if nums[i] > nums[i+1] {
8                   nums.swap(i, i+1);
9               }
10          }
11          len -= 1;
12      }
13  }
14
15  fn main() {
16      let mut nums = [54,26,93,17,77,31,44,55,20];
17      bubble_sort2(&mut nums);
18      println!("sorted nums: {:?}", nums);
19  }
```

注意，不管項在初始集合中如何排列，算法都將進行 n-1 輪遍歷以排序 n 個數字。第一輪要比較 n - 1 次，第二輪 n - 2 次，直到 1 次。總的比較次數爲 1 + 2 +..., n - 1，爲前 n 個整數之和 $\frac{n^2}{2} + \frac{n}{2}$，所以性能爲 $O(n^2)$。

上面的冒泡算法都實現了排序，但仔細分析發現，即便序列已經排好序了，算法也要不斷的比較，只是不交換值。一個有序的集合就不應該再排序了。修改上面算法，添加一個 compare 變量來控制是否繼續比較，在遇到已排序集合時直接退出。

```rust
// bubble_sort.rs
fn bubble_sort3(nums: &mut [i32]) {
    let mut compare = true;
    let mut len = nums.len() - 1;

    while len > 0 && compare {
        compare = false;
        for i in 0..len {
            if nums[i] > nums[i+1] {
                nums.swap(i, i+1);
                compare = true; // 數據無序，還需繼續比較
            }
        }
        len -= 1;
    }
}
```

冒泡排序是從第一個數開始，依次往後比較，相鄰的元素兩兩比較，根據大小來交換元素的位置。這個過程中，元素是單向交換的，也就是說只有從左往右交換。那麼是否可以再從右到左來個冒泡排序呢？從左到右是升序排序，如果從右到左採取降序排序，那麼這種雙向排序法也一定能完成排序。這種排序稱爲雞尾酒排序，是冒泡排序的一種變體排序。雞尾酒稍微優化了冒泡排序，其複雜度還是 $O(n^2)$，若序列已經排序，則接近 $O(n)$。

```rust
// cocktail_sort.rs
fn cocktail_sort(nums: &mut [i32]) {
    if nums.len() <= 1 { return; }

    // bubble 控制是否繼續冒泡
    let mut bubble = true;
    let len = nums.len();
    for i in 0..(len >> 1) {
        if bubble {
            bubble = false;

            // 從左到右冒泡
            for j in i..(len - i - 1) {
                if nums[j] > nums[j+1] {
                    nums.swap(j, j+1);
                    bubble = true
```

```
17                  }
18              }
19
20              // 從右到左冒泡
21              for j in (i+1..=(len - i - 1)).rev() {
22                  if nums[j] < nums[j-1] {
23                      nums.swap(j-1, j);
24                      bubble = true
25                  }
26              }
27          } else {
28              break;
29          }
30      }
31  }
32
33  fn main() {
34      let mut nums = [1,3,2,8,3,6,4,9,5,10,6,7];
35      cocktail_sort(&mut nums);
36      println!("sorted nums {:?}", nums);
37  }
```

在冒泡排序中，只會比較數組中相鄰的項，比較間距爲 1。一種稱爲梳排序的算法提出比較間距可以大於 1。梳排序中，開始比較間距設定爲數組長度，並在循環中以固定的比率遞減，通常遞減率爲 1.3，該數字是原作者通過實驗得到的最有效遞減率。因爲乘法計算中耗時比除法短兩個時鐘週期，所以間距計算會取遞減率的倒數與數組長度相乘，即 0.8。當間距爲 1 時，梳排序就退化成了冒泡排序。梳排序通過儘量把逆序的數字往前移動並保證當前間隔內的數有序，類似梳子理順頭髮，間隔則類似梳子梳齒間隙。梳排序時間複雜度是 O(nlogn)，空間複雜度爲 O(1)，屬於不穩定的排序算法。

```
1  // comb_sort.rs
2  fn comb_sort(nums: &mut [i32]) {
3      if nums.len() <= 1 { return; }
4
5      let mut i;
6      let mut gap: usize = nums.len();
7
8      // 大致排序，數據基本有序
9      while gap > 0 {
```

```
        gap = (gap as f32 * 0.8) as usize;
        i = gap;
        while i < nums.len() {
            if nums[i-gap] > nums[i] {
                nums.swap(i-gap, i);
            }
            i += 1;
        }
    }

    // 細致調節部分無序數據，exchange 控制是否繼續交換數據
    let mut exchange = true;
    while cnt > 0 {
        exchange = false;
        i = 0;
        while i < nums.len() - 1 {
            if nums[i] > nums[i+1] {
                nums.swap(i, i+1);
                exchange = true;
            }
            i += 1;
        }
    }
}

fn main() {
    let mut nums = [1,2,8,3,4,9,5,6,7];
    comb_sort(&mut nums);
    println!("sorted nums {:?}", nums);
}
```

冒泡排序還有一個問題，就是它需要合理的安排好邊界下標值如 i、j、i+1、j+1，一點都不能錯。下面是 2021 年新發表的一種不需要處理邊界下標值的排序算法 [11]，非常直觀，乍一看以爲是冒泡排序，但它實際類似插入排序。看起來是降序排序，實際是升序排序。

```
// CantBelieveItCanSort.rs

fn cbic_sort1(nums: &mut [i32]) {
    for i in 0..nums.len() {
```

```
5           for j in 0..nums.len() {
6               if nums[i] < nums[j] {
7                   nums.swap(i, j);
8               }
9           }
10      }
11  }
12
13  fn main() {
14      let mut nums = [54,32,99,18,75,31,43,56,21,22];
15      cbic_sort1(&mut nums);
16      println!("sorted nums {:?}", nums);
17  }
```

當然，也可以優化此排序算法，像下面這樣。

```
1   // CantBelieveItCanSort.rs
2
3   fn cbic_sort2(nums: &mut [i32]) {
4       if nums.len() < 2 {
5           return;
6       }
7
8       for i in 1..nums.len() {
9           for j in 0..i {
10              if nums[i] < nums[j] {
11                  nums.swap(i, j);
12              }
13          }
14      }
15  }
16
17  fn main() {
18      let mut nums = [54,32,99,18,75,31,43,56,21,22];
19      cbic_sort2(&mut nums);
20      println!("sorted nums {:?}", nums);
21  }
```

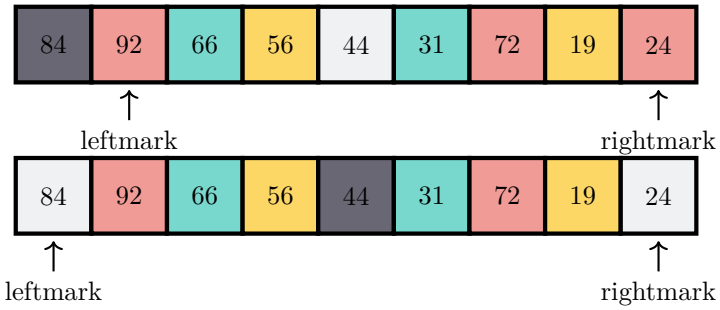
這個排序算法是直覺上最像冒泡定義的排序算法，然而它並不是冒泡排序算法。讀者若要實現降序排序，只需要改變小於符號爲大於符號就行。

6.4 快速排序

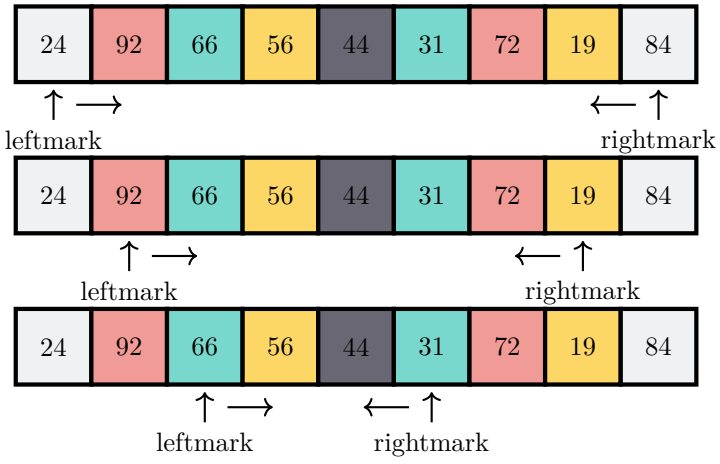
快速排序和冒泡排序有相似之處，應該說快速排序是冒泡排序的升級版。快速排序使用分而治之的策略來加快排序速度，這又和二分思想、遞歸思想有些類似。

快速排序只有兩個步驟，一是選擇中樞值，二是分區排序。首先在集合中選擇某個值作爲中樞，其作用是幫助拆分集合。注意中樞值不一定要選集合中間位置的值，中樞值應該是在最終排序集合中處於中間或靠近中間的值，這樣排序速度才快。有很多不同的方法用於選擇中樞值，本文只爲說明原理，不考慮算法優化，所以直接選擇第一項作中樞值。下圖選擇了 84 作爲中樞值，實際上，排好序後，84 並不在中間，而是在倒數第二位。56 纔是在中間。所以如果能選到 56 作中樞值將會非常高效。

選擇好中樞值後 (灰色值)，需要再設置兩個標記，用於比較。這兩個標記稱爲左標記和右標記，分別處於集合中除中樞值的最左和最右端。下圖展示了左右標記的位置。



可以看到，左右標記儘可能遠離，處於左右兩個極端。分區的目標是移動相對於中樞值 (44) 錯位的項，通過比較左右標記和中樞處的值，交換小值到左標記處，大值到右標記處。



首先右移左標記，直到找到一個大於等於中樞值的值。然後左移遞減右標記，直到找到小於等於中樞值的值。如果左標記值大於右標記值就交換值。此處 84 和 24 恰好滿足條件，所以直接交換值。然後重複該過程。直到左右標記相互越過對方。比較越過後左右標記值，若右小於左，則將此時右標記值和中樞值交換，右標記值作爲分裂點，將集合分爲左右兩個區間，然後在左右區間遞歸調用快速排序，直到最後完成排序。

可以看到，只要左右兩側分別執行快速排序，最終就能完成排序。如果集合長度小於或等於一，則它已經排序，直接退出。具體實現如下，我們爲分區設置了專門的分區函數 partition，算法中取第一項做中樞值。

```
// quick_sort.rs

fn quick_sort1(nums: &mut [i32], low: usize, high: usize) {
    if low < high {
        let split = partition(nums, low, high);
        if split > 1 { // 防止越界和語法錯誤 (split <= 1 的情形)
            quick_sort1(nums, low, split - 1);
        }
        quick_sort1(nums, split + 1, high);
    }
}

fn partition(nums: &mut [i32], low: usize, high: usize)
    -> usize {
    let mut lm = low;  // 左標記
    let mut rm = high; // 右標記
    loop {
        // 左標記不斷右移
        while lm <= rm && nums[lm] <= nums[low] {
            lm += 1;
        }
        // 右標記不斷右移
        while lm <= rm && nums[rm] >= nums[low] {
            rm -= 1;
        }

        // 左標記越過右標記時退出并交換左右標記數據
        if lm > rm {
            break;
        } else {
```

```
31                 nums.swap(lm, rm);
32             }
33         }
34         nums.swap(low, rm);
35
36         rm
37     }
38
39     fn main() {
40         let mut nums = [54,26,93,17,77,31,44,55,20];
41         let len = nums.len();
42         quick_sort1(&mut nums, 0, (len - 1) as isize);
43         println!("sorted nums: {:?}", nums);
44     }
```

對於長度爲 n 的集合，如果分區總在中間，則會再出現 $log_2(n)$ 個分區。爲找到分割點，需要針對中樞值檢查 n 項中的每一個，複雜度爲 $nlog_2(n)$ 。最壞的情況下，分裂點可能不在中間，非常偏左或右。此時會不斷的對 1 和 n-1 項重複排序 n 次，複雜度爲 $O(n^2)$。

快速排序需要遞歸，深度過深性能會下降。內觀排序克服了這個缺點，其在遞歸深度超過 $log(n)$ 後轉爲堆排序。在數量少（$n < 20$）時，則轉爲插入排序。這種多個排序混合而成的排序能在常規數據集上實現快速排序的高性能，又能在最壞情況下仍保持 O(nlog(n)) 的性能，內觀排序是 C++ 的內置排序算法。此外，對快速排序分析發現，它總是將待排序數組分成兩個區域來排序。如果待排序集合有大量重複元素，則快速排序會重複比較，造成性能浪費。解決方法是將數組分成三區排序，把重複元素放到第三個區域，排序時只對另外兩個區域排序。選擇重複數據作爲 pivot，小於 pivot 的放到左區，大於 pivot 的放到右區，等於的放到中區。然後對左右區域遞歸調用三區快速排序。

6.5 插入排序

插入排序，就像它的名字暗示的一樣，是通過插入數據項來實現排序。儘管仍然性能是 $O(n^2)$，但其工作方式略有不同。它始終在數據集的較低位置處維護一個有序的子序列，然後將新項插入子序列，使得子序列擴大，最終實現集合排序。

假設開始的子序列只有一項，位置爲 0。在下次遍歷時，對於項 1 至 n-1，將其與第一項比較，如果小於該項，則將其插入到該項前。如果大於該項，則增長子序列，使長度加一。接着重複比較過程，在剩余的未排序項裏取數據來比較。結果是要麼插入子序列某個位置，要麼增長子序列，最終得到排好序的集合。插入排序的具體實現和圖示如下。

```
1  // insertion_sort.rs
2  fn insertion_sort(nums: &mut [i32]) {
```

```
3       for i in 1..nums.len() {
4           let mut pos = i;
5           let curr = nums[i];
6
7           while pos > 0 && curr < nums[pos-1] {
8               nums[pos] = nums[pos-1]; // 向後移動數據
9               pos -= 1;
10          }
11
12          nums[pos] = curr; // 插入數據
13      }
14  }
15
16  fn main() {
17      let mut nums = [54,32,99,18,75,31,43,56,21];
18      insertion_sort(&mut nums);
19      println!("sorted nums: {:?}", nums);
20  }
```

84	92	66	56	44	31	72	19	24	假設 84 已排序
84	92	66	56	44	31	72	19	24	仍舊有序
66	84	92	56	44	31	72	19	24	插入 66
56	66	84	92	44	31	72	19	24	插入 56
44	56	66	84	92	31	72	19	24	插入 44
31	44	56	66	84	92	72	19	24	插入 31
31	44	56	66	72	84	92	19	24	插入 72
19	31	44	56	66	72	84	92	24	插入 19
19	24	31	44	56	66	72	84	92	插入 24

图 6.2: 插入排序

插入排序總是需要和前面已排序的元素逐個比較來找到具體位置，而第五章我們學習了二分查找法，可以快速找到元素在已排序子序列中的位置，所以可以利用二分法來加快插入排序的速度。具體的改動就是在插入時，用二分法找到數據該插入的位置，然後直接移動數據到相應位置。

```rust
// bin_insertion_sort.rs

fn bin_insertion_sort(nums: &mut [i32]) {
    let mut temp;
    let mut left; let mut mid; let mut right;

    for i in 1..nums.len() {
        left = 0;       // 已排序數組左右邊界
        right = i - 1;
        temp = nums[i]; // 待排序數據

        // 二分法找到 temp 的位置
        while left <= right {
            mid = (left + right) >> 1;
            if temp < nums[mid] {
                // 防止出現 right = 0 - 1
                if 0 == mid { break; }
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        // 將數據後移，留出空位
        for j in (left..=i-1).rev() {
            nums.swap(j, j+1);
        }

        // 將 temp 插入空位
        if left != i {
            nums[left] = temp;
        }
    }
}
```

```
35
36 fn main() {
37     let mut nums = [1,3,2,8,6,4,9,7,5,10];
38     bin_insertion_sort(&mut nums);
39     println!("sorted nums {:?}", nums);
40 }
```

6.6 希爾排序

希爾排序，也稱遞減遞增排序。它將原始集合分爲多個較小的子集合，然後對每個集合運用插入排序。選擇子集合的方式是希爾排序的關鍵。希爾排序不是將集合均勻拆分爲連續項的子列表，而是隔幾個項選擇一個項加入子集合，隔開的距離稱爲增量 gap。

84	92	66	56	44	31	72	19	24
84	92	66	56	44	31	72	19	24
84	92	66	56	44	31	72	19	24

這可以通過上圖來加以理解。該集合有九項，如果使用三爲增量，則總共會有三個子集合，每個子集合三項，顏色一樣。這些隔開的元素可以看成是連接在一起的，這樣可以通過插入排序來對同顏色元素進行排序。

完成排序後，總的集合還無序，如下圖。雖然總集合無序，但這個集合並非完全無序，可以看到，同種顏色的子集合是有序的。只要再對整個集合進行插入排序，則很快就能將集合完全排序。可以發現，此時的插入排序移動次數非常少，因爲挨着的幾個項都處於自己所在子集合的有序位置，那麼這些挨着的項也幾乎有序，所以只用少量插入次數就能完成排序。

56	19	24	72	44	31	84	92	66

希爾排序中，增量是關鍵，也是其特徵。可以使用不同的增量，但增量爲幾，那麼子集合就有幾個。下面是希爾排序的實現，通過不斷調整 gap 值，實現排序。

```
1 // shell_sort.rs
2
3 fn shell_sort(nums: &mut [i32]) {
4     // 插入排序函數(內部)，數據相隔距離爲 gap
5     fn ist_sort(nums: &mut [i32], start: usize, gap: usize) {
6         let mut i = start + gap;
```

```
 7
 8          while i < nums.len() {
 9              let mut pos = i;
10              let curr = nums[pos];
11              while pos >= gap && curr < nums[pos - gap] {
12                  nums[pos] = nums[pos - gap];
13                  pos -= gap;
14              }
15
16              nums[pos] = curr;
17              i += gap;
18          }
19      }
20
21      // 每次讓 gap 縮小一半直到 1
22      let mut gap  = nums.len() / 2;
23      while gap > 0 {
24          for start in 0..gap {
25              ist_sort(nums, start, gap);
26          }
27          gap /= 2;
28      }
29  }
30
31  fn main() {
32      let mut nums = [54,32,99,18,75,31,43,56,21,22];
33      shell_sort(&mut nums);
34      println!("sorted nums: {:?}", nums);
35  }
```

乍一看，希爾排序並不比插入排序更好，因爲它最後一步還是執行了完整的插入排序。然而，希爾排序分割子序列對其排序後，最後一次插入排序進行的插入操作就非常少了，因爲該集合已經被較早的增量插入排序預排序了。換句話說，隨着 gap 值向 1 靠攏，整個集合都比上一次更有序，這使得總的排序非常高效。

希爾排序的複雜度分析稍微複雜一些，但其大致分佈在 $O(n)$ 到 $O(n^2)$ 之間。改變 gap 的值，使其按照 $2^k - 1$ 變化（1, 3, 7, 15, 31），那麼其複雜度大概在 $O(n^{1.5})$ 左右，也是非常快的。當然，前面對插入排序可以用二分法改進，那麼對希爾排序也可以使用二分法改進，具體思路同前面一樣，只是下標處理不是連續的，要計算上 gap 值。

6.7 歸併排序

現在轉向使用分而治之策略作爲提高排序算法性能的另外一種方法，歸併排序。歸併排序和快速排序都是一種分而治之的遞歸算法，通過不斷將列表折半來進行排序。如果集合爲空或只有一個項，則按基本情況進行排序。如果有多項，則分割集合，並遞歸調用兩個區間的歸併排序。一旦對這兩個區間排序完成，就執行合併操作。合併是獲取兩個子排序集合並將它們組合成單個排序新集合的過程。因爲歸併排序是一種結合遞歸和合並操作的排序，所以名字就叫歸併排序，如下圖所示。

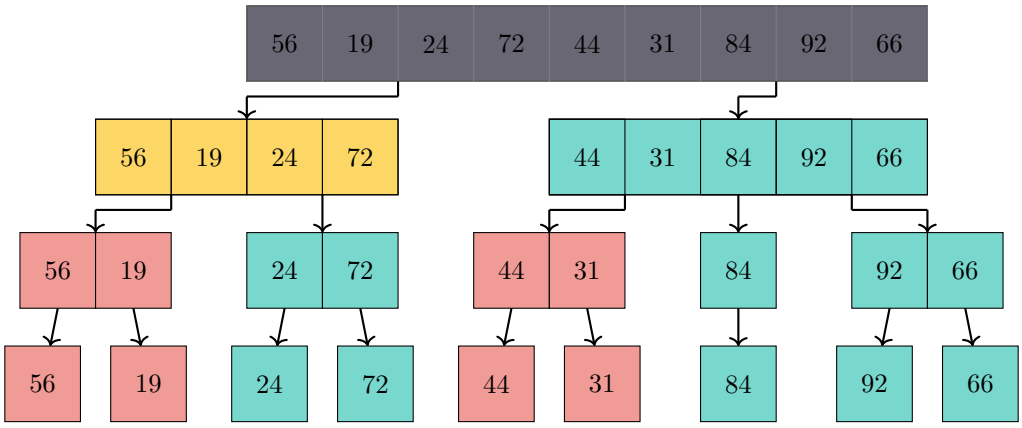


图 6.3: 歸併排序分解

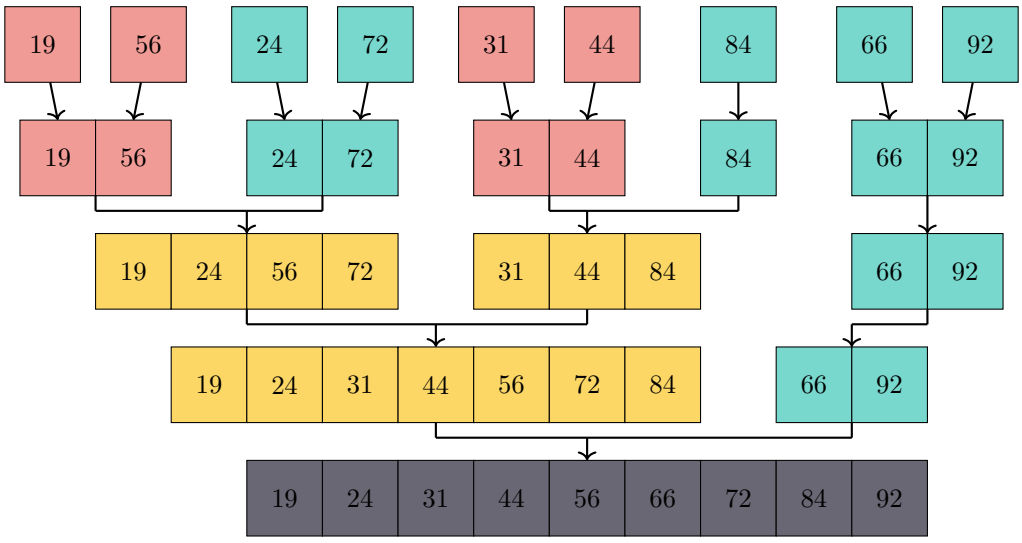


图 6.4: 歸併排序合併

歸併排序時遞歸分解集合到只有兩個元素或一個元素的基本情況，便於直接比較。分解後是合併過程，首先對基本情況的最小子序列進行排序，接着開始兩兩合併，直到完成集合

排序，如上圖所示。歸併排序分割集合時，可能不是均分，因爲數據不一定是偶數，但最多相差一個元素，不影響性能。合併操作其實很簡單，因爲每一次合併時，子序列都已經排好序，只需要逐個比較，先取小的，最後組合的序列一定也是有序的。下面是歸併排序的實現代碼。

```rust
// merge_sort.rs

fn merge_sort(nums: &mut [i32]) {
    if nums.len() > 1 {
        let mid = nums.len() >> 1;
        merge_sort(&mut nums[..mid]); // 排序前半部分
        merge_sort(&mut nums[mid..]); // 排序後半部分
        merge(nums, mid); // 合并排序結果
    }
}

fn merge(nums: &mut [i32], mid: usize) {
    let mut i = 0;   // 標記前半部分數據
    let mut k = mid; // 標記後半部分數據
    let mut temp = Vec::new();

    for _j in 0..nums.len() {
        if k == nums.len() || i == mid {
            break;
        }

        // 數據放到臨時集合 temp
        if nums[i] < nums[k] {
            temp.push(nums[i]);
            i += 1;
        } else {
            temp.push(nums[k]);
            k += 1;
        }
    }

    // 合并兩部分數據長度大概率不一樣長
    // 所以要將未處理完集合的數據全部加入
    if i < mid && k == nums.len() {
```

```
35          for j in i..mid {
36              temp.push(nums[j]);
37          }
38      } else if i == mid && k < nums.len() {
39          for j in k..nums.len() {
40              temp.push(nums[j]);
41          }
42      }
43
44      // temp 數據放回 nums, 完成排序
45      for j in 0..nums.len() {
46          nums[j] = temp[j];
47      }
48  }
49
50  fn main() {
51      let mut nums = [54,32,99,22,18,75,31,43,56,21];
52      merge_sort(&mut nums);
53      println!("sorted nums: {:?}", nums);
54  }
```

爲了分析歸併排序的性能，首先將集合分成兩部分。在二分查找一節已經學習過，總共需要分 $log_2(n)$ 次。第二個過程是合併，集合中的每個項最終將被放置在排好序的列表上，對於 n 個數據，最多就放 n 次。遞歸和合並兩個操作是結合在一起的，所以歸併排序是一種性能爲 $O(nlog_2(n))$ 的算法。

歸併排序空間複雜度爲 O(n)，這是比較高的，自然的想法就是減少空間使用。我們知道插入排序空間複雜度是 O(1)，所以可以利用插入排序來優化歸併排序。當長度小於某閾值時直接調用插入排序，大於閾值時採用歸併排序，這種算法叫插入歸併排序，在一定程度上優化了歸併排序算法。

6.8 選擇排序

選擇排序是對冒泡排序的改進，每次遍歷集合只做一次交換。爲做到這一點，選擇排序在遍歷時只尋找最大值的下標，並在完成遍歷後，將該最大項交換到正確的位置。與冒泡排序一樣，在第一次遍歷後，集合的最大項在最後一個位置；第二遍後，次個值在倒數第二位，遍歷 n-1 次纔會排序完 n 個項。下圖展示了整個排序過程。每次遍歷時，選擇未排序的最大的項，然後放置在適當位置。

選擇排序與冒泡排序具有相同數量的比較，因此性能也是 $O(n^2)$。然而，由於選擇排序每輪只進行一次數據交換，所以比冒泡排序更快，下面是其實現代碼。

```rust
// selection_sort.rs
fn selection_sort(nums: &mut Vec<i32>) {
    let mut left = nums.len() - 1; // 待排序數據下標
    while left > 0 {
        let mut pos_max = 0;
        for i in 1..=left {
            if nums[i] > nums[pos_max] {
                pos_max = i; // 選擇當前輪次最大值下標
            }
        }
        // 數據交換，完成一個數據的排序，待排序數據量 - 1
        nums.swap(left, pos_max);
        left -= 1;
    }
}
fn main() {
    let mut nums = vec![54,32,99,18,75,31,43,56,21,22];
    selection_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
}
```

下圖是選擇排序的示意圖，值被挑出來放到正確位置，而不像冒泡排序那樣交換。

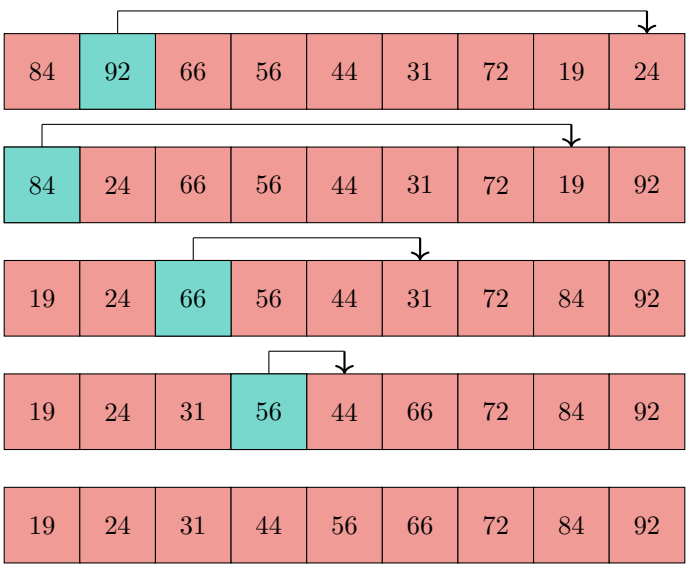


图 6.5: 選擇排序

可以看到，即使冒泡和選擇排序複雜度一樣，仍然有優化點。這對於今後學習、研發很有幫助，要在看起來很複雜的問題中找出優化點，儘可能地優化它。前面我們學習雞尾酒排序可以雙向同時排序，此處的選擇排序也可以實現雙向排序。這種改進的選擇排序的複雜度和常規的選擇排序是一樣的，唯一改變的是複雜度的係數，具體請讀者自己思考實現。

6.9 堆排序

前面章節學習了棧，隊列這些線性數據結構並用這些數據結構實現了各種算法。除此之外，計算機裏還有非線性的數據結構，其中一種是堆。堆是一種非線性的完全二叉樹，具有以下性質：每個結點的值都小於或等於其左右孩子結點的值，稱爲小頂堆；每個結點的值都大於或等於其左右孩子結點的值，稱爲大頂堆，如圖（6.6）所示是一個小頂堆。

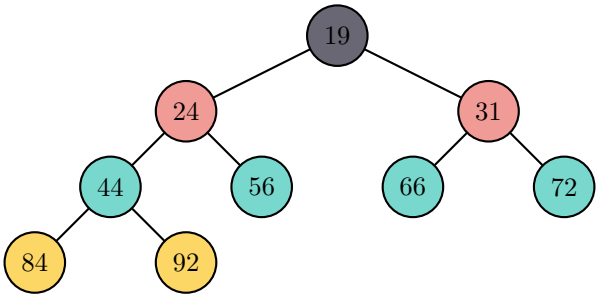


图 6.6: 小頂堆

堆排序是利用堆數據結構設計的一種排序算法，是一種選擇排序，通過不斷選擇頂元素到末尾，然後再重建堆實現排序。它的最壞、最好、平均時間複雜度均爲 $O(nlog_2(n))$，它是不穩定排序。雖然在第七章纔開始學習樹，但這裏稍微瞭解一下有助於理解堆的性質。通過下面的圖可以看到，這種堆類似具有多個連接的鏈表。如果對堆中的結點按層進行編號，將這種邏輯結構映射到數組中就像下圖一樣。其中第一位，下標爲 0，此處用 0 來占位。

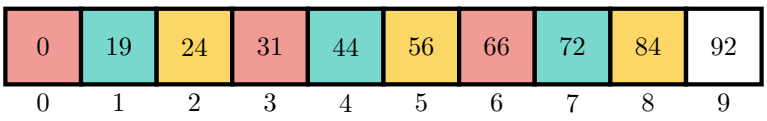


图 6.7: 小頂堆數組表示

可見，我們不一定非得用鏈表，用 Vec 或者數組都能表示堆，其實堆就其意義上來說，也類似一堆東西聚合在一起，所以數組或 Vec 表示的堆比之二叉樹結構更貼近堆這個詞彙的本身意義。注意此處我們的下標從 1 開始，這樣可以將左右子節點表示爲 arr[2i] 和 arr[2i+1]。

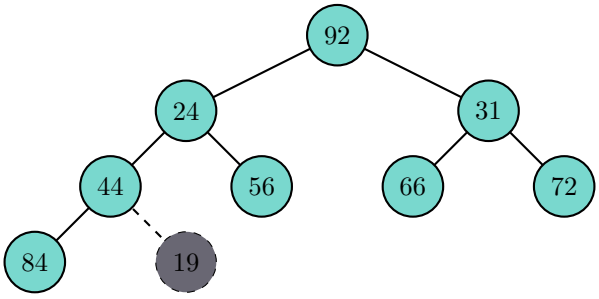
藉助數組表示，按照二叉樹的節點關係，堆滿足如下的定義：

大頂堆：arr[i] >= arr[2i] 且 arr[i] >= arr[2i+1]

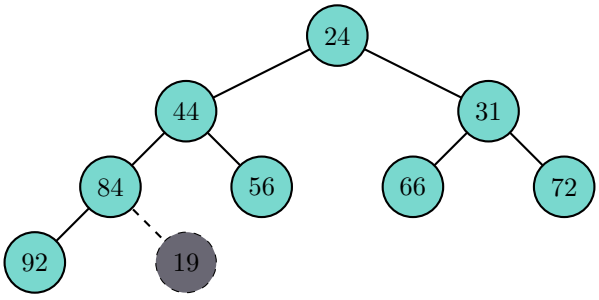
小頂堆：arr[i] <= arr[2i] 且 arr[i] <= arr[2i+1]

堆排序的基本思想是：將待排序序列構造成一個小頂堆，此時，整個序列的最小值就是堆頂根節點。將其與末尾元素進行交換，此時末尾就爲最小值。這個最小值不再計算到堆內，那麼再將剩余的 n - 1 個元素重新構造成一個堆，這樣會得到一個新的最小值。此時將該最小值再次交換到新堆的末尾，這樣就有了兩個排序的值。重複這個過程，直到得到一個有序序列。當然，小頂堆得到的是降序排序，大頂堆得到的纔是升序排序。

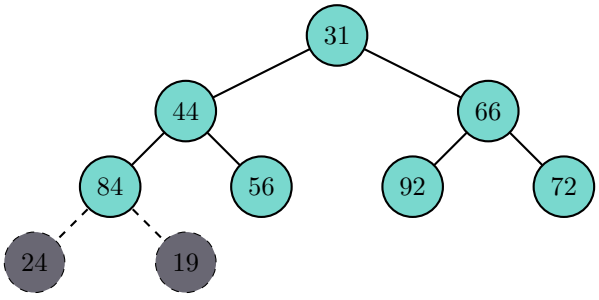
爲便於讀者理解堆排序過程，可藉助圖來演示其排序過程。下圖中灰色爲最小元素，是從頂部和 92 替換到了此處，不再計入堆內。交換後，92 位於堆頂，堆不再是小頂堆，所以交換後需要重新構建小頂堆，使得最小值 24 在堆頂。之後再交換 24 和堆中最後一個元素，則倒數第二個將變爲灰色。注意虛線表示該元素已經不屬於堆了，重建堆和交換時都不再處理它。繼續交換下去，則灰色排序的子序列從最後一層倒序着逐步填滿整個堆，最終實現排序，此時序列是從大到小逆序排序。



下面是重新構建小頂堆後 24 位於堆頂時堆的情況。



下面時交換堆頂元素及重建堆後的情況，此時 92 已經交換到右側，31 是堆中最小元素。



有了上面的圖示，可以實現如下堆排序。

```rust
// heap_sort.rs

macro_rules! parent { // 計算父節點下標宏
    ($child:ident) => {
        $child >> 1
    };
}
macro_rules! left_child {// 計算左子節點下標宏
    ($parent:ident) => {
        ($parent << 1) + 1
    };
}
macro_rules! right_child {// 計算右子節點下標宏
    ($parent:ident) => {
        ($parent + 1) << 1
    };
}

fn heap_sort(nums: &mut [i32]) {
    if nums.len() <= 1 { return; }

    let len = nums.len();
    let last_parent = parent!(len);
    for i in (0..=last_parent).rev() {
        move_down(nums, i); // 第一次建小頂堆
    }

    for end in (1..nums.len()).rev() {
        nums.swap(0, end);
        move_down(&mut nums[..end], 0); // 重建堆
    }
}

// 大的數據項下移
fn move_down(nums: &mut [i32], mut parent: usize) {
    let last = nums.len() - 1;
    loop {
        let left = left_child!(parent);
```

```
39          let right = right_child!(parent);
40          if left > last { break; }
41
42          // right <= last 確保存在右子節點
43          let child = if right <= last
44                      && nums[left] < nums[right] {
45              right
46          } else {
47              left
48          };
49
50          // 子節點大於父節點，交換數據
51          if nums[child] > nums[parent] {
52              nums.swap(parent, child);
53          }
54
55          // 節點移動，更新父子
56          parent = child;
57      }
58  }
59
60  fn main() {
61      let mut nums = [54,32,99,18,75,31,43,56,21,22];
62      heap_sort(&mut nums);
63      println!("sorted nums: {:?}", nums);
64  }
```

堆就是二叉樹,其複雜度是 $O(nlog_2(n))$。這裏我們使用了宏 parent!,left_child!,right_child! 來獲取節點下標，當然也可以用函數來實現，但這裏就當作複習 rust 基礎知識。堆排序和選擇排序有些類似，都是在集合中選擇出最值並放到適當位置。

6.10 桶排序

前面學習的排序多涉及到比較操作，其實還有一些排序不用比較，只要按照數學規律就能自動映射數據到正確位置。這類非比較算法主要是桶排序，計數排序，基數排序。

非比較排序通過確定每個元素之前有多少個元素存在來排序。比如對集合 nums，計算 nums[i] 之前有多少個元素，則唯一確定了 nums[i] 在排序集合中的位置。非比較排序只要確定每個元素之前的已有的元素個數即可，所以一次遍歷即可完成排序，時間複雜度 O(n)。

雖然非比較排序時間複雜度低，但由於非比較排序需要占用額外空間來確定位置，所以

對數據規模和數據分佈有一定的要求。因爲不是所有數據都適合這類排序，數據本身必須包含可索引的信息用於確定位置。而比較排序的優勢是，適用於各種規模的數據，也不在乎數據的分佈，都能進行排序。可以說，比較排序適用於一切需要排序的情況，非比較排序只適合特殊數據（尤其是數字）的排序。

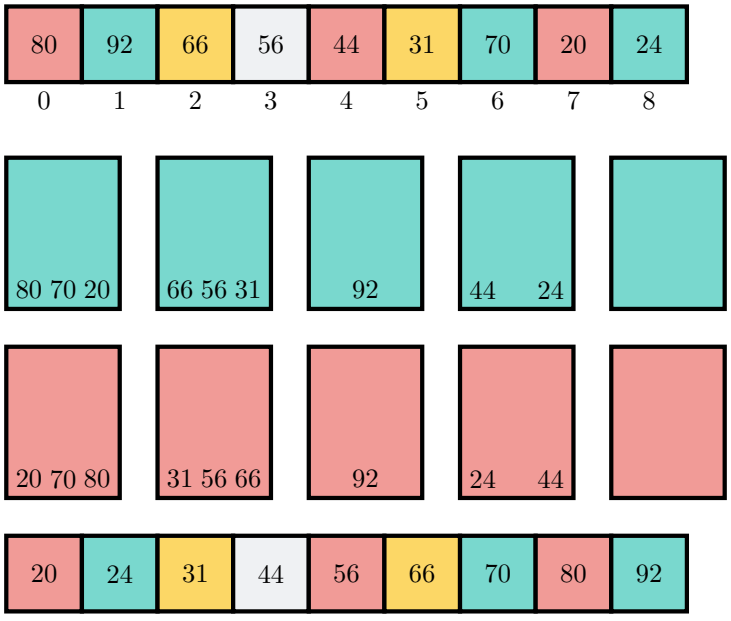
第一種非比較排序是桶排序，桶和哈希表中槽的概念是類似的，只是槽只能裝一個元素，而桶可以裝若干元素。槽用於保存元素，桶用於排序元素，桶排序基本思路是：

```
1 第一步，將待排序元素劃分到不同的桶，先遍歷求出 maxV 和 minV，
2 設桶個數爲 k ，則把區間 [minV, maxV] 均勻劃分成 k 個區間，
3 每個區間就是一個桶，將序列中的元素分配到各自的桶（求余法）。
4 第二步，對每個桶內的元素進行排序，排序算法可用任意排序算法。
5 第三步，將各個桶中的有序元素合并成一個大的有序集合。
```

在 Rust 裏，可以定義桶爲一個結構體，包含哈希函數和數據集合。

```
1 // bucket_sort.rs
2
3 // hasher 是一個函數，計算時傳入
4 // values 是數據容器，保存數據
5 struct Bucket<H, T> {
6     hasher: H,
7     values: Vec<T>,
8 }
```

下圖是桶排序示意圖，先分散數據到桶，然後桶內排序並將各桶數據合併得到有序集合。



下面是桶排序的實現。

```
// bucket_sort.rs

impl<H, T> Bucket<H, T> {
    fn new(hasher: H, value: T) -> Bucket<H, T> {
        Bucket { hasher: hasher, values: vec![value] }
    }
}

// 桶排序，Debug 特性是爲了打印 T
fn bucket_sort<H, T, F>(nums: &mut [T], hasher: F)
    where H: Ord, T: Ord + Clone + Debug, F: Fn(&T) -> H {
    let mut buckets: Vec<Bucket<H, T>> = Vec::new();

    for val in nums.iter() {
        let hasher = hasher(&val);

        // 對桶中數據二分搜索并排序
        match buckets.binary_search_by(|bct|
                                       bct.hasher.cmp(&hasher)) {
            Ok(idx) => buckets[idx].values.push(val.clone()),
            Err(idx) => buckets.insert(idx, Bucket::new(hasher,
                                            val.clone())),
        }
    }

    // 拆桶，將所有排序數據融合到一個 Vec
    let ret = buckets.into_iter().flat_map(|mut bucket| {
        bucket.values.sort();
        bucket.values
    }).collect::<Vec<T>>();

    nums.clone_from_slice(&ret);
}

fn main() {
    let mut nums = [54,32,99,18,75,31,43,56,21,22];
    bucket_sort(&mut nums, |t| t / 5);
```

```
38      println!("sorted nums: {:?}", nums);
39  }
```

桶排序實現要複雜些，因爲需要先實現桶的結構，然後再基於此結構實現排序算法。這裏數據放到各個桶的依據是對 5 求余，當然可以是其他值，那麼桶個數就要相應改變。求余方法 hasher 採用的是閉包函數。

假設數據是均勻分佈的，則每個桶的元素平均個數爲 n/k。假設選擇用快速排序對每個桶內的元素進行排序，那麼每次排序的時間複雜度爲 O(n/klog(n/k)) 。總的時間複雜度爲 O(n)+O(k)O(n/klog(n/k)) = O(n+nlog(n/k)) = O(n+nlogn-nlogk)。當 k 接近於 n 時，桶排序的時間複雜度就可以認爲是 O(n) 。即桶越多，時間效率就越高，而桶越多，空間就越大，越費內存，可見這是用空間換時間。

桶排序的一個缺點是桶的數量太多。比如待排序數組 [1,100,20,9,4,8,50]，按照桶排序算法會創建 100 個桶，然而大部分桶用不上，造成了空間浪費。

FlashSort 是一種優化的桶排序，它的思路很簡單，就是減少桶的數量。對於待排序數組 [1,100,20,9,4,8,50]，先找到最大最小值 A_{max}, A_{min}，然後用這兩個值來估算大概需要的桶數量。思路是這樣的，如果按照桶排序計算出的桶個數大於待排序元素個數，那麼就通過 m = f*n 來計算桶數，f 是個小數，比如 f = 0.2。元素入桶的規則如下：

$$K(A_i) = 1 + int((m-1)\frac{A_i - A_{min}}{A_{max} - A_{min}})$$

假設有 n 個元素，每個桶平均有 n/m 個元素，對每個桶使用插入排序，總複雜度爲 $O(\frac{n^2}{m})$。原作者通過實驗發現，m = 0.42 n 時，性能最優，爲 O(n)。當 m = 0.1n 時，只要 n > 80，這個算法就比快速排序快，當 n = 10000 時，快 2 倍。當 m = 0.2n 時，比 m = 0.1n 還快 15%。就算 m 只有 0.05n，當 n>200 是也要顯著地比快速排序快。

6.11 計數排序

第二種非比較排序是計數排序，計數排序是桶排序的特殊情況，它的桶就只處理同種數據，所以比較費空間，基本思路是：

```
1  第一步，初始化長度爲 maxV - minV + 1 的計數器集合，值全爲 0，
2  其中 maxV 爲待排序集合的最大值，minV 爲最小值。
3  第二步，掃描待排序集合，以當前值減 minV 作下標，并對計數器中
4  此下標的計數加 1。
5  第三步，掃描一遍計數器集合，按順序把值寫回原集合，完成排序。
```

舉個例子，nums=[0,7,1,7,3,1,5,8,4,4,5]，首先遍歷 nums 獲取最小值和最大值，maxV=8，minV=0，於是初始化一個長度爲 8 - 0 + 1 的計數器集合 counter。

$$[0, 0, 0, 0, 0, 0, 0, 0, 0]$$

接着掃描 nums，計算當前值減 minV 作爲下標，如掃描到 0，則下標爲 0 - 0 = 0，所以 counter 下標 0 處值加 1。counter 此時爲

$$[1,0,0,0,0,0,0,0,0]$$

接着掃描到 7，下標爲 7 - 0 = 7 ，所以對應位置加一，counter = [1,0,0,0,0,0,0,1,0]。繼續掃描，最終 counter 爲

$$[1,2,0,1,2,2,0,2,1]$$

有了 counter，那麼遍歷 counter 時只要某下標處數字不爲 0，則將對應下標值寫入 nums，counter 中值減一。比如 counter 第一個位置 0 處爲 1，說明 nums 中有一個 0，此時寫入 nums，繼續，下標 1 處值爲 2，說明 nums 中有兩個 1，寫入 nums。最終 nums 爲

$$[0,1,1,3,4,4,5,5,7,7,8]$$

此時集合已經有序，且排序過程中不涉及比較、交換等操作，所以速度快。

```rust
// counting_sort.rs

fn counting_sort(nums: &mut [usize]) {
    if nums.len() <= 1 { return; }

    // 桶數量爲 nums 中最大值加 1，保證數據都有桶放
    let max_bkt_num = nums.iter().max().unwrap() + 1;
    let mut counter = vec![0; max_bkt_num];
    for &v in nums.iter() {
        counter[v] += 1; // 將數據標記到桶
    }

    // 數據寫回原 nums 切片
    let mut j = 0;
    for i in 0..max_bkt_num {
        while counter[i] > 0 {
            nums[j] = i;
            counter[i] -= 1;
            j += 1;
        }
    }
}

```

```
24 fn main() {
25     let mut nums = [54,32,99,18,75,31,43,56,21,22];
26     counting_sort(&mut nums);
27     println!("sorted nums: {:?}", nums);
28 }
```

6.12 基數排序

第三種非比較排序是基數排序，它利用正數的進制規律來排序，基本是收集分配這樣一個思路，其思想具體如下。

```
1 第一步，找到 nums 中最大值，得到位數，將數據統一爲相同位數，
2 不够補零。
3 第二步，從最低位開始，依次進行穩定排序，收集，再排序高位，
4 直到排序完成。
```

舉個例子，有一個整數序列，[1,134,532,45,36,346,999,102]。下面是排序過程，見圖 (6.8)。第零次排序，首先找到最大值 999，三位數，所以要進行個位，十位，百位三輪排序。補 0 得到最左側集合。第一輪對個位排序，第二輪對十位排序，第三輪對百位排序。可以看到紅色位就是當前輪次排序的位，該位上的數字都是有序的。

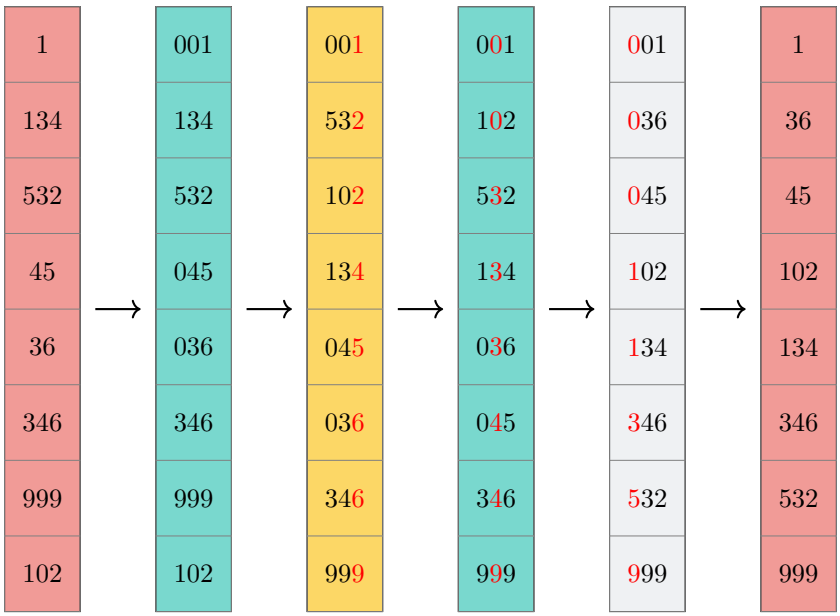


图 6.8: 計數排序

下面是基數排序的具體實現。

```rust
// radix_sort.rs

fn radix_sort(nums: &mut [usize]) {
    if nums.len() <= 1 { return; }

    // 找到最大的數，它的位最多
    let max_num = match nums.iter().max() {
        Some(&x) => x,
        None => return,
    };

    // 找最接近且 >= nums 長度的 2 的次冪值作爲桶大小，如：
    // 最接近且 >= 10 的 2 的次冪值是 2^4 = 16
    // 最接近且 >= 17 的 2 的次冪值是 2^5 = 32
    let radix = nums.len().next_power_of_two();

    // digit 代表小於某個位對應桶的所有數
    // 個、十、百、千分別爲在 1, 2, 3, 4 位
    // 起始從個位開始，所以是 1
    let mut digit = 1;
    while digit <= max_num {
        // index_of 計算數據在桶中哪個位置
        let index_of = |x| x / digit % radix;

        // 計數器
        let mut counter = vec![0; radix];
        for &x in nums.iter() {
            counter[index_of(x)] += 1;
        }

        for i in 1..radix {
            counter[i] += counter[i-1];
        }

        // 排序
        for &x in nums.to_owned().iter().rev() {
            counter[index_of(x)] -= 1;
            nums[counter[index_of(x)]] = x;
```

```
39          }
40
41          // 跨越桶
42          digit *= radix;
43      }
44  }
45
46  fn main() {
47      let mut nums = [54,32,99,18,75,31,43,56,21,22];
48      radix_sort(&mut nums);
49      println!("sorted nums: {:?}", nums);
50  }
```

爲什麼同一數位的排序要用穩定排序？因爲穩定排序能將上一次排序的成果保留下來。例如十位數的排序過程能保留個位數的排序成果，百位數的排序過程能保留十位數的排序成果。能不能用 2 進制？能。可以把待排序序列中的每個整數都看成是 01 組成的二進制數值。這樣任意一個非負整數序列都可以用基數排序算法解決？假設待排序序列中最大整數 64 位，則時間複雜度爲 O(64n)，此時 64 也很重要，不可忽視。

既然任意一個非負整數序列都可以在線性時間內完成排序，那麼基於比較排序的算法有什麼意義呢？基於比較的排序算法，時間複雜度是 O(nlogn)，看起來比 O(64n) 慢，但其實不是，O(nlogn) 只有當序列非常長時 logn 纔會達到 64，所以 64 這個係數太大了，基於比較的算法還是更快的。當使用 2 進制時，k=2 最小，位數最多，時間複雜度 O(nd) 會變大，空間複雜度 O(n+k) 會變小。當用最大值作爲基數時，k=maxV 最大，位數最小，此時時間複雜度 O(nd) 變小，但是空間複雜度 O(n+k) 會急劇增大，此時基數排序退化成了計數排序。

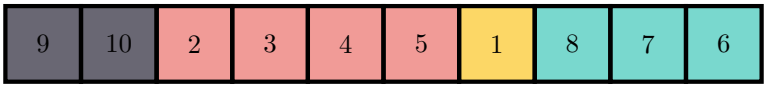
綜合來看，三個非比較排序是相互有關係的。計數排序是桶排序的特殊情況，基數排序若採用最少的位來排，則此時也退化成計數排序。所以基數排序和計數排序都可以看作是桶排序，計數排序是桶取最大值時的桶排序，基數排序是每個數位上的桶排序，是多輪桶排序。當用最大值作基數時，基數排序退化成計數排序。桶排序適合元素分佈均勻的場景，計數排序要求 maxV 和 minV 差距小，基數排序只能處理正數，也要求 maxV 和 minV 儘可能接近。所以，這三個排序只能排序少量的數據，最好總量小於 10000。

6.13 蒂姆排序

我們已經學習了十類排序算法，然而這些算法各有優缺點，不能很好的適合各種情況的排序。爲此 Tim Peters 提出了結合多種排序的混合排序算法 TimSort。該排序算法高效，穩定且自適應數據分佈，比大多數排序算法都優秀。

Tim Peters 在 2002 年提出了 TimSort，並首先在 Python 中實現爲 sort 操作的默認算法，目前許多編程語言和平臺都將 TimSort 或其改進版作爲默認排序算法，包括 Java，Python，Rust，Android 平臺。

TimSort 是一種混合的排序算法，結合了歸併和插入排序，旨在更好地處理多種數據。現實中需要排序的數據通常有部分已經排好序了（包括逆序），如下圖，同顏色數據是有序的。



Timsort 正是利用了這一特點來劃分區塊並進行排序。Timsort 稱這些排好序的數據塊爲 run，可將其視爲一個個分區，運算單元。在排序時，Timsort 迭代數據元素，將其放到不同的 run 裏，同時針對這些 run，按規則合併至只剩一個，則這個僅剩的 run 即爲排好序的結果。當然，爲了設置合適的分區，TimSort 設置了 minrun 這個參數，即分區數據不能少於 minrun，如果小於 minrun，就利用插入排序擴充 run 到 minrun，然後再合併。

TimSort 根據排序數據集合大小產生 minrun，劃分 $\frac{n}{minrun}$ 個 run，$\frac{n}{minrun}$ 小於等於 2 的次冪。若 run 的長度等於 64，則 minrun = 64，直接調用二分查找插入排序. 當集合元素個數大於 64 時，選擇 [32, 64] 中某個值爲 minrun。使得 $\frac{n}{minrun}$ 小於等於 2 的次冪。具體而言，選擇集合長度的六個二進制位爲 minrun，若剩余標誌位不爲 0，則 minrun 加 1。

a. 集合長度 189: 10111101, 前六位 101111 = 47, 剩余位 01, 則 minrun = 48, n/minrun = 4。

b. 集合長度 976: 1111010000，前六位 111101 = 61，剩余位 0000，則 minrun = 61。

尋找 run，若長度小於 minrun，則調用插入排序擴充，將後面元素填充到 run。通過兩兩合併 run，直到只剩一個 run，則集合排序完成。

6.14 總結

本章學習了十類排序算法。冒泡及選擇排序和插入排序是 $O(n^2)$ 算法，其余排序算法的複雜度多是 $O(nlog_2(n))$。選擇排序是對冒泡排序的改進，希爾排序是對插入排序的改進，堆排序是對選擇排序的改進，快速排序和歸併排序均利用分而治之的思想。這些排序都是通過比較來排序，還有不需要比較只依靠數值規律而排序的算法，這類排序算法是非比較排序算法，分別有桶排序、計數排序、基數排序。它們的複雜度都是 O(n) 左右，適合少量數據排序。計數排序是特殊的桶排序，基數排序是多輪桶排序，基數排序可以退化成計數排序。除了基礎排序算法，本章還學習了部分算法的改進版，尤其是蒂姆算法，是高效穩定的混合排序算法，其改進版已經是許多語言和平臺的默認排序算法。下一頁是十大排序算法的總結，讀者可自行對照理解，以便加深印象。

表 6.1: 十大排序算法的時間和空間複雜度

序	排序算法	最差複雜度	最優複雜度	平均複雜度	空間複雜度	穩定性	綜合類別
1	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	穩定	交換比較類
2	快速排序	$O(n^2)$	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	不穩定	交換比較類
3	選擇排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不穩定	選擇比較類
4	堆排序	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(1)$	不穩定	選擇比較類
5	插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	穩定	插入比較類
6	希爾排序	$O(n^2)$	$O(n)$	$O(n^{1.3})$	$O(1)$	不穩定	插入比較類
7	歸併排序	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(n)$	穩定	分治比較類
8	計數排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	穩定	非比較類
9	桶排序	$O(n^2)$	$O(n)$	$O(n+k)$	$O(n+k)$	穩定	非比較類
10	基數排序	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$	穩定	非比較類

Chapter 7

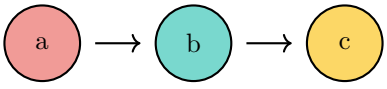
樹

7.1 本章目標

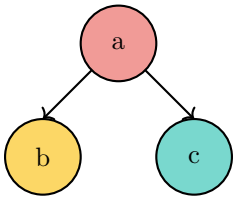
理解樹及其使用方法
理解二叉樹及平衡二叉樹
用二叉堆實現優先級隊列
實現二叉樹及平衡二叉樹

7.2 什麽是樹

前面章節我們學習了鏈表、棧、隊列等數據結構，這些數據結構都是線性的，一個數據結點只能連接後面一項數據。

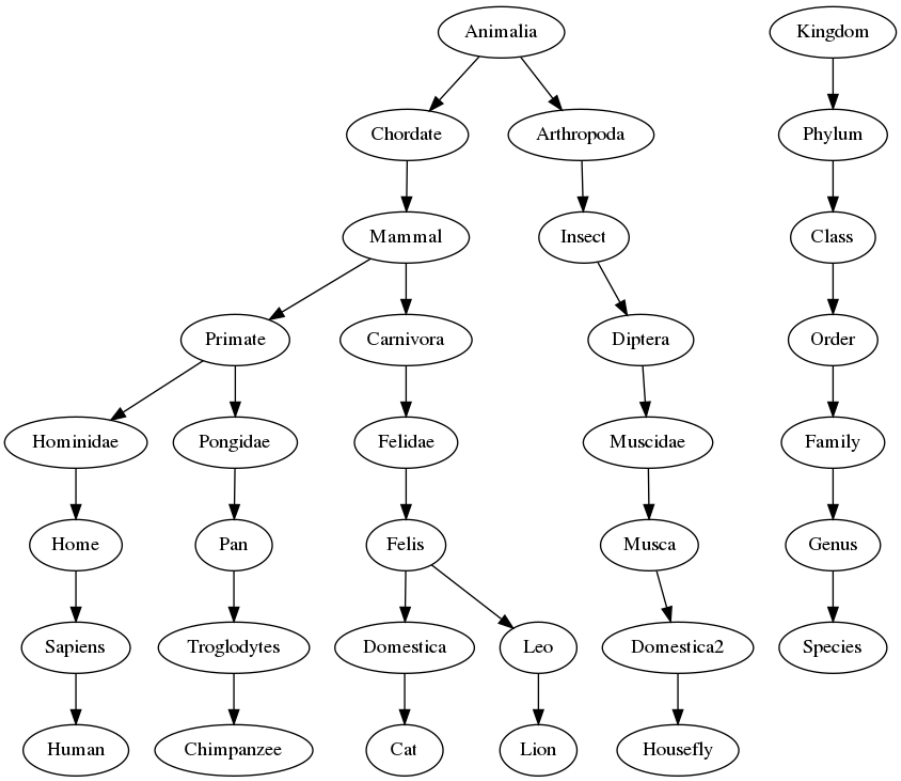


如果對線性數據進行拓展，爲數據結點連接多項，那麽就可以得到一種新的數據結構。



這種新的數據結構就像樹一樣，它有一個根，然後發散出了枝條和葉子，並且相互連接在一起。這種新的數據結構就稱爲樹。自然界中的樹和計算機科學中的樹之間的區別在於樹數據結構的根在頂部，葉在底部。樹在計算機科學的許多領域中使用，包括操作系統，圖形，數據庫和計算機網絡等。爲簡化行文，下文將樹數據結構簡稱爲樹。

在開始研究樹之前，先來看幾個常見的樹例子。第一個例子是生物學的分類樹，如下圖。從這幅圖可以看到人具體所處的位置（左下側），這對研究事物關係和性質非常有幫助。

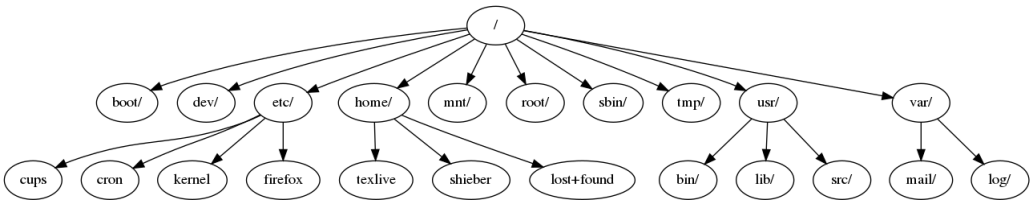


從這幅圖中我們可以瞭解到樹的屬性。第一點，樹是分層的。通過分層，樹具有良好的層次結構，具體到這幅圖就是“種、屬、科、目、綱、門、界”七大層次。接近頂部的是抽象層次最高的事物，底層是最具體的事物。人種很具體，但動物界就太抽象了，包含所有動物，不僅限於人，還有蟲、魚、鳥、獸。從這棵樹的根部開始，沿着箭頭一直走到底部，會給出一條完整的路徑，該路徑表明了底層物種的全稱。比如人只是簡稱，全稱是“動物界-脊椎門-哺乳綱-靈長類目-人科-人屬-智人種”。每一個生物都能在這棵生命大樹上找到自己的位置，並顯示出相對關係。

樹的第二個屬性是一個節點的所有子節點獨立於另一個節點的子節點。智人種這個子節點不會屬於昆蟲綱及其子節點，這使得彼此之間關係明確，同時也意味着改變一個節點的子節點對其他節點沒有影響。比如發現新的昆蟲，這使得生命樹又龐大了，但人科下面毫無變化，整個生物學知識的更新也不會涉及到人科。這種性質非常有用，尤其是樹作爲數據存儲容器的時候，可以藉助工具來修改某些結點上的數據而保持其他數據不變。

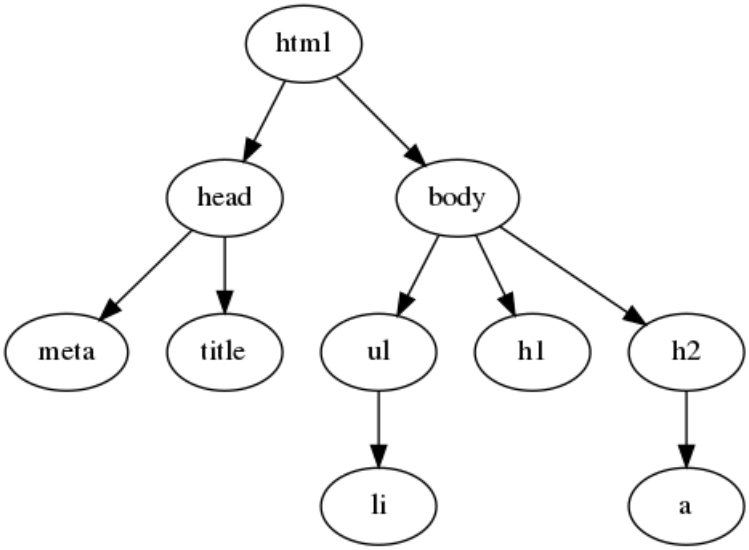
樹的第三個屬性是每個葉節點是唯一的。你可以從樹的根到葉子節點找出一條唯一的路徑，這種性質使得保存數據非常有效，既然路徑唯一，那麼就可以用來作爲數據的存儲路徑。實際上，我們電腦裏的文件系統就是通過改進的樹來保存文件的。文件系統樹與生物分類樹有很多共同之處，從根目錄到任何子目錄的路徑唯一標識了該子目錄及其中的所有文件。如

果你使用類 Unix 操作系統，那麼你應該熟悉類似 /root，/home/user，/etc 這樣的路徑，這些路徑都是樹上的節點，顯然 / 是根節點，是樹根。



還有一個使用樹的例子是網頁文件。網頁是資源的集合，也具有樹的層次結構。下面是 google.cn 的查找界面網頁數據，可以看到這些 <> 括號標籤也是有層次的。

```
<html lang="zh"><head><meta charset="utf-8">
  <head>
      <meta charset="utf-8">
      <title>Google</title>
      <style>
        html { background: #fff; margin: 0 1em; }
        body { font: .8125em/1.5 arial, sans-serif; }
      </style>
  </head>
  <body>
  <div>
    <a href="http://www.google.com.hk/webhp?hl=zh-CN">
      <img src="//www.google.cn/landing/cnexp/search.png"
        alt="Google" width="586" height="257">
    </a>
    <h1><a href="http://www.google.com.hk/webhp?hl=zh-CN">
        <strong id="target">google.com.hk</strong></a></h1>
    <p>請收藏我們的網址</p>
  </div>
  <ul>
    <li><a href="http://translate.google.cn/">翻譯</a></li>
  </ul>
  <p id="footer"> ©2011 - <span>ICP證合字B2-20070004號</span>
  </p>
  </body>
</html>
```



7.2.1 樹的定義

我們已經看了樹的示例，現在來定義樹的各種屬性。

節點：節點是樹的基本部分，它還有一個名稱叫做 “鍵”。節點也可以有附加信息，附加信息稱爲 “有效載荷”。雖然有效載荷信息不是許多樹算法的核心，但在利用樹的應用中通常是關鍵的，比如樹節點上存儲時間，文件名等。

根：根是樹中唯一沒有傳入邊的節點，它處於頂層，所有的節點都可以從根找到，類似操作系統的 / 或 C 盤這樣的概念。

邊：邊是樹的另一個基本部分，又叫分支。邊連接兩個節點以保持之間存在的關係。每個節點，除根之外，都恰好有一個輸入邊和若干輸出邊。邊就是路徑，可以通過它來找到某個節點的具體位置。

路徑：路徑是由邊連接節點的有序序列，它本身並不存在，是由其邏輯結構涌現出來的一種信息。比如 /home/user/files/sort.rs 就是一條路徑，它標識了 sort.rs 這個文件的具體位置。

子節點：子節點是某個節點的下一級，所有子節點都源自同一個上層節點。比如上面的 sort.rs 就是 files/ 的子節點。子節點不唯一，可以存在零個、一個、多個子節點，這和人類社會的父子關係一樣，所以名字也借用了人類的親屬關係稱謂詞。

父節點：父節點是所有下級節點的源，所有子節點都源自同一個父節點。比如 files/ 就是 sort.rs 的父節點。父節點唯一，這很好理解，一個孩子只可能有一個父親。

子樹：子樹是由父節點和該父節點的所有後代組成的一組節點和邊。因爲樹這種遞歸結構，從樹中任意節點取出一部分，它的結構都仍然是樹，這部分取出的內容就稱爲子樹。

葉節點：葉節點是沒有子節點的節點，處於最底層。

中間節點：中間節點有子節點，有父節點。

層數：節點 n 的層數爲從根到該結點所經過的分支數目。根節點的層數爲零，/home-

/user/files 中，files 的層數爲 2。層數爲零不代表沒有層數，而是層數就是零，此零不是無，沒有的意思，而是第零層。

高度：樹的高度等於樹中任何節點的最大層數。

有了這些基礎知識就可以給出樹的定義。

```
1 樹具有一個根節點。
2 除根節點外，每個節點通過其他節點的邊互相連接父和子節點（若有）。
3 從根遍歷到任何節點的路徑全局唯一。
```

下圖爲一個樹結構，左右子節點爲 lc 和 rc。因爲樹的結構是遞歸的，所以子樹的結構和父樹一致。

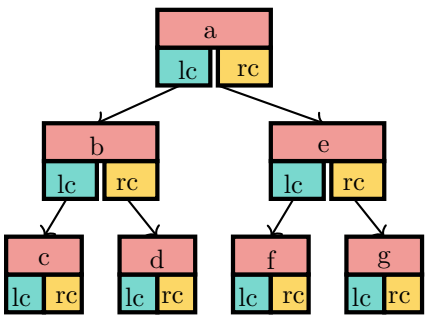


图 7.1: 樹的節點表示

7.2.2 樹的表示

樹是一種非線性數據結構，然而計算機的存儲硬件都是線性的。所以，計算機要表示樹必然涉及用線性結構來表徵非線性結構。一種表徵方法是用數組來構成樹，下面的 tree 就是通過數組來構造的。

```
1  tree = [ 'a',
2             ['b',
3               ['c',[],[]],
4               ['d',[],[]],
5             ],
6             ['e',
7               ['f',[],[]],
8               []
9             ],
10         ]
```

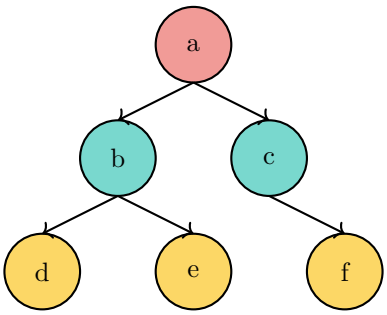
從這個數組結構可以看到樹是如何保存在數組中的。我們知道數組在內存中是連續的，所以這個數在內存中也是連續的，可以利用數組的訪問方式來獲取數據。比如 tree[0] 就是'a'，

左子樹是 tree[1]，包含'b', 'c', 'd'，而該子樹還是數組，繼續使用數組訪問方法，則 tree[1][0] 就是數據'c'。數組表示方法的一個好處是，表示子樹的數組仍然遵循樹的定義，結果是這整個結構本身是遞歸的，可以不斷獲取子樹及元素。

```
println!("root {:?}", tree[0]);
println!("left subtree {:?}", tree[1]);
println!("right subtree {:?}", tree[2]);
```

用數組保存樹雖然可行，但是嵌套太深，十分複雜。如果有十層，那麼獲取子樹和元素非常麻煩，試想獲取第四層的某個葉節點，你得用類似 tree[1][1][1][2] 這種形式訪問，這對計算機和對人來說都太複雜了。所以用數組保存樹，理論上可行，實際上不行。

另外一種可行的樹保存方式是節點（Node）。回想鏈表一節的內容，我們發現鏈表節點和此處的節點概念是一致的，且鏈表裏的鏈就是樹裏的邊，就像下圖。



這種結構看起來非常直觀，且不用嵌套，避免了元素訪問的麻煩。現在的關鍵是：如何定義樹節點？有了根結點才能保存其子節點，一種可行的辦法是使用 struct 定義節點。

```
// binary_tree.rs
use std::fmt::{Debug, Display}

// 子節點鏈接
type Link<T> = Option<Box<BinaryTree<T>>>;

// 二叉樹定義
// key 保存數據
// left 和 right 保存左右子節點鏈接
#[derive(Debug, Clone)]
struct BinaryTree<T> {
    key: T,
    left: Link<T>,
    right: Link<T>,
}
```

key 中保存數據，left 和 right 保存左右子節點的地址，這樣就可以通過訪問地址獲取子節點。可以通過插入函數來爲根加入子節點。

```
// binary_tree.rs

impl<T: Clone> BinaryTree<T> {
    fn new(key: T) -> Self {
        BinaryTree { key: key, left: None, right: None }
    }

    // 新子節點作爲根節點的左子節點
    fn insert_left_tree(&mut self, key: T) {
        if self.left.is_none() {
            let node = BinaryTree::new(key);
            self.left = Some(Box::new(node));
        } else {
            let node = BinaryTree::new(key);
            node.left = self.left.take();
            self.left = Some(Box::new(node));
        }
    }

    // 新子節點作爲根節點的右子節點
    fn insert_right_tree(&mut self, key: T) {
        if self.right.is_none() {
            let node = BinaryTree::new(key);
            self.right = Some(Box::new(node));
        } else {
            let node = BinaryTree::new(key);
            node.right = self.right.take();
            self.right = Some(Box::new(node));
        }
    }
}
```

插入子節點時，必須考慮兩種情況。第一種情況是節點沒有子節點，此時直接插入就行。第二種情況是節點具有子節點，則先將子節點接到新節點的子節點位置，然後再將新節點作爲根的子節點。

代碼裏 node = BinaryTree 可能讓你感覺混淆，怎麼又是節點又是樹？實際上，我們定

義的 BinaryTree 看起來是一個節點，但多個節點鏈接起來後節點又成了樹。當插入節點時，可以將整個子樹看成一個節點，這樣便於操作。但在使用時，它本身又是棵樹，可以獲取內部節點信息。最好的理解方法就是，節點是樹的一部分，它本身也可用樹來表示，來插入，來移動，因爲樹是遞歸的。但使用時，其內部結構很重要，所以寫的是 BinaryTree 而不是 Node。

爲獲取二叉樹節點數據，可以爲其實現獲取左右子節點以及根節點值的函數。

```
// binary_tree.rs

impl<T: Clone> BinaryTree<T> {
    // 獲取左右子節點及根節點，注意使用了 clone
    fn get_left(&self) -> Link<T> {
        self.left.clone()
    }

    fn get_right(&self) -> Link<T> {
        self.right.clone()
    }

    fn get_key(&self) -> T {
        self.key.clone()
    }

    fn set_key(&mut self, key: T) {
        self.key = key;
    }
}
```

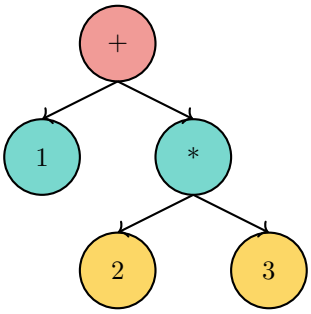
有了創建和操作二叉樹的函數，讓我們使用它來創建上面提到的二叉樹吧。這個二叉樹非常簡單，只使用插入函數就能構造。

```
// binary_tree.rs

fn main() {
    let mut bt = BinaryTree::new('a');

    let root = bt.get_key();
    println!("root val is {:?}", root);

    let left = bt.get_left();
    println!("left child is {:#?}", left);
```

```
11      let right = bt.get_right();
12      println!("left child is {:#?}", right);
13
14      bt.insert_left_tree('b');
15      bt.insert_right_tree('e');
16
17      let left = bt.get_left();
18      println!("left child is {:#?}", left);
19      let right = bt.get_left();
20      println!("right child is {:#?}", right);
21  }
```

7.2.3 分析樹

有了樹的定義和操作函數，現在可以來思考樹在保存數據時的工作原理，比如用樹來存儲類似 (1 + (2 * 3)) 這種式子。前面已經學過完全括號表達式，通過括號可以指示優先級。同樣，由於有括號，所以可以指明符號保存的順序。如果將算術表達式表示成樹，那麼應該類似下面這樣的樹結構。



那麼，樹是如何把數據保存進去的呢？根據樹的結構，可以定義一些規則，然後按照規則把數據保存到節點上。定義規則如下:

```
1  若當前符號是 '(', 添加新節點作爲左子節點, 然後下降到左子節點。
2  若當前符號在 ['+',' - ','/','*'] 中, 將根值置爲當前符號。
3      添加新右子節點, 下降到該子節點。
4  若當前符號是數字, 將根值置爲該數字, 返回到父節點。
5  若當前符號是 ')', 則轉到當前節點的父節點。
```

利用這套規則，我們可以將 (1 + (2 * 3)) 轉換成上圖所示的樹。具體步驟如下，灰色表示當前節點。

a. 創建根節點。

b. 讀取符號 (，創建新左子節點，下降到該節點。

c. 讀取符號 1，將節點值置爲 1，返回父節點。

d. 讀取符號 +，將節點值置爲 +，創建新右子節點，下降到該節點。

e. 讀取符號 (，創建新左子節點，下降到該節點。

f. 讀取符號 2，將節點值置爲 2，返回父節點。

g. 讀取符號 *，將節點值置爲 *，創建新右子節點，下降到該節點。

h. 讀取符號 3，將節點值置爲 3，返回父節點。

i. 讀取符號)，返回父節點。

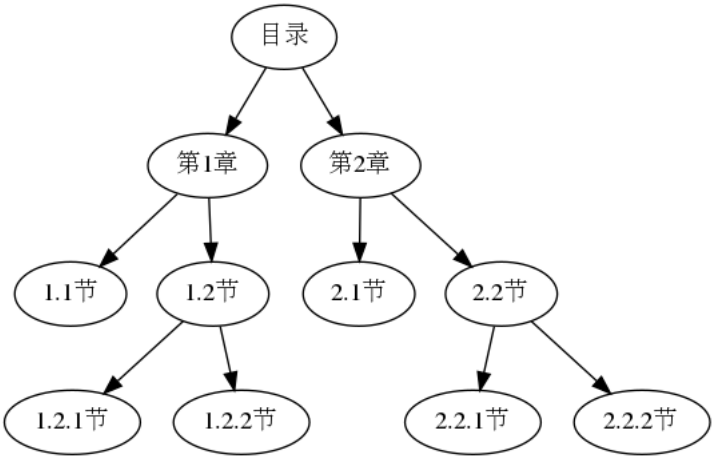
算術表達式保存需要關注子節點和父節點，然而我們沒有實現獲取父節點的功能，所以可以採用棧來保存當前節點的父節點，或者實現獲取父節點的功能 get_parent()。

我們用樹完成了算術表達式的二叉樹保存，樹能維持數據的結構信息。實際上，編程語言在編譯時，也是用樹來保存所有的代碼並生成抽象語法樹。通過分析語法樹各部分功能，生成中間代碼，優化，生成最終代碼。如果你瞭解編譯原理，那麼這對你來說應該很熟悉。

7.2.4 樹的遍歷

保存數據的目的是爲了更高效的使用數據，包括增加、刪除、查找、修改。其中增刪改的前提是要找到數據所在位置，對於樹來說，是定位具體的節點。所以查找，訪問或查找節點是首先要完成的功能。不同於線性數據結構，可以通過下標遍歷所有數據。樹是非線性結構，所以樹的查找方法不同於棧、數組這類線性數據結構。

有三種常用的方法來訪問樹節點，這些方法間的差異主要是節點被訪問的順序。參照線性數據結構的遍歷方法，我們也稱樹的節點訪問方法爲遍歷。這三種遍歷方法分別是前序遍歷、中序遍歷、後序遍歷。首先用一個例子來說明這三種遍歷。假如把這本書表示爲樹，那麼目錄是根，各章是其子節點，而小節是各章各自的子節點，依此類推。



想象自己讀本書的順序，是不是按照第一章第一節，第二節；第二章第一節，第二節這樣的順序？你可以看看各章節所在位置，並從目錄開始按照看書的順序在圖中勾勒出閱讀軌F，這種順序就是前序遍歷 preorder。前序遍歷從根節點開始，左子樹繼後，右子樹最後。

算法遍歷時，從根節點開始，遞歸調用左子樹前序遍歷。對於上面的樹，preorder 得出的數據訪問順序和目錄的線性順序一致。先訪問目錄，然後第一章第一節，第二節... 一章完後，回到目錄，選擇第二章繼續遞歸調用 preorder，訪問第二章第一節，第二節...。

前序遍歷算法理解起來不復雜，實現也很簡單，利用遞歸就可以做到。下面是單獨實現的函數，可以前序遍歷樹。

```
// 前序遍歷： 外部實現
fn preorder<T: Clone + Debug>(bt: Link<T>) {
    if !bt.is_none() {
        println!("key is {:?}", bt.as_ref().unwrap().get_key());
        preorder(bt.as_ref().unwrap().get_left());
        preorder(bt.as_ref().unwrap().get_right());
    }
}
```

當然，也可以將遍歷方法直接實現爲 BinaryTree 的內部函數。

```
// binary_tree.rs

impl<T: Clone> BinaryTree<T> {
    // 前序遍歷：{\CJKfamily{fangsong}F}部實現
    fn preorder(&self) {
        println!("kes: {:?}", &self.key);
        if !self.left.is_none() {
            self.left.as_ref().unwrap().preorder();
        }
        if !self.right.is_none() {
            self.right.as_ref().unwrap().preorder();
        }
        // as_ref() 獲取節點引用，因爲打印不能更改節點
    }
}
```

後序遍歷和前序遍歷類似，它先查看左子樹，然後右子樹，最後根節點。

```
// binary_tree.rs 後序遍歷：內部與外部實現

fn postorder(bt: Link<T>) {
    if !bt.is_none() {
        postorder(bt.as_ref().unwrap().get_left());
        postorder(bt.as_ref().unwrap().get_right());
```

```
7           println!("key is {:?}", bt.as_ref().unwrap().get_key());
8       }
9   }
10
11  impl<T: Clone> BinaryTree<T> {
12      fn postorder(&self) {
13          if !self.left.is_none() {
14              self.left.as_ref().unwrap().postorder();
15          }
16          if !self.right.is_none() {
17              self.right.as_ref().unwrap().postorder();
18          }
19          println!("kes is {:?}", &self.key);
20      }
21  }
```

現在回到前一節的算術表達式 (1 + (2 * 3))，我們用樹保存了這個表達式，如果要計算它，總是需要先取到操作符和操作數，然後施加運算。要獲得這三個值，需要正確的順序。前序遍歷會盡可能的從根節點開始，然而計算表達式要從葉節點數據開始，所以後序遍歷纔是正確的數據獲取方法。通過先獲取左右子節點值，再獲取根節點操作符，可以做一次運算，並將該結果保存在運算符號所在位置，然後繼續後序遍歷，以先前計算的值爲左節點，然後訪問右子節點，直到計算出最終值。

最後一種遍歷是中序遍歷。中序遍歷首先訪問左子樹，然後訪問根，最後訪問右子樹。

```
1   // binary_tree.rs 中序遍歷：內部與外部實現
2
3   fn inorder(bt: Link<T>) {
4       if !bt.is_none() {
5           inorder(bt.as_ref().unwrap().get_left());
6           println!("key is {:?}", bt.unwrap().get_key());
7           inorder(bt.as_ref().unwrap().get_right());
8       }
9   }
10
11  impl<T: Clone> BinaryTree<T> {
12      fn inorder(&self) {
13          if !self.left.is_none() {
14              self.left.as_ref().unwrap().inorder();
15          }
```

```
16          println!("kes is {:?}", &self.key);
17          if !self.right.is_none() {
18              self.right.as_ref().unwrap().inorder();
19          }
20      }
21  }
```

對保存算術表達式 (1 + (2 * 3)) 的樹使用中序遍歷可以得到原來的表達式 1 + 2 * 3。注意，因爲樹沒有保存括號，恢復的表達式只是順序正確，但優先級不一定對，可以修改中序遍歷使得輸出包含括號。

```
1  // binary_tree.rs
2
3  // 按照節點位置返回節點組成的字符串
4  fn get_exp<T: Clone + Debug + Display>(bt: Link<T>) -> Sring {
5      let mut exp = "".to_string();
6      if !bt.is_none() {
7          exp = "(".to_string() + &get_exp(bt.unwrap().get_left());
8          exp += &bt.as_ref().unwrap().get_key().to_string();
9          exp += &(get_exp(bt.as_ref().unwrap().get_right()) + ")");
10     }
11
12     exp
13 }
```

讓我們簡化三種遍歷訪問順序的描述，前序遍歷簡化成“根左右”，表示先訪問根，再訪問左子樹，最後訪問右子樹。綜合三種遍歷可以得出，根左右是前序遍歷，左根右是中序遍歷，左右根是後序遍歷。實際上，還可以有根右左這種訪問順序，但這是前序遍歷的鏡像。因爲左和右是相對的，所以左根右和右根左可以看成互爲鏡像。同理，右左根是後序遍歷的鏡像；右根左是中序遍歷的鏡像。

表 7.1: 前中後序遍歷及其鏡像遍歷總結

序	遍歷方法	遍歷順序	鏡像遍歷順序	鏡像遍歷方法
1	前序遍歷	根左右	根右左	前序鏡像遍歷
2	中序遍歷	左根右	右根左	中序鏡像遍歷
3	後序遍歷	左右根	右左根	後序鏡像遍歷
4	前序鏡像遍歷	根右左	根左右	前序遍歷
5	中序鏡像遍歷	右根左	左根右	中序遍歷
6	後序鏡像遍歷	右左根	左右根	後序遍歷

7.3 基於二叉堆的優先隊列

在前面的章節中，我們學習了隊列這種先進先出的線性數據結構。隊列的一個變種稱爲優先級隊列，它的作用就像一個隊列，你可以通過從隊首出隊數據項。然而，在優先級隊列中，項的順序不是按照從末尾加入的順序，而是由數據項的優先級確定。最高優先級項在隊列的首部，最先出隊。因此，將項加入優先級隊列時，如果新項的優先級夠高，那麼它會一直往隊首移動，其實就是利用某個指標來排序，使得該項排到前面去。

優先級隊列是很有用的一種數據結構，尤其對於涉及到優先級的事務，用這種隊列管理就非常有效。比如，操作系統會調度各個進程，那麼哪個進程該排在前面呢？這時優先級隊列就非常有用了。通過某種算法，操作系統可以得到進程的優先級，然後據此排序。比如，你在用手機聽音樂，同時還在瀏覽新聞，這時一個電話打過來，那麼系統會將電話直接提到最高優先級，直接打斷新聞瀏覽界面和音樂，展示一個來電呼叫界面。這就是利用優先級隊列來管理的進程，直接賦予來電呼叫很高的優先級。

如果叫你來實現這個優先級隊列，你會用什麼辦法呢？這種優先級隊列一定是根據某種規則排序，把高優先級項排到最前面。然而，插入隊列複雜度是 O(n)，且排序隊列複雜度至少也有 O(nlogn)。要做得更快的話，可以採取用堆來排序。堆其實是一種完全二叉樹，所以用來實現優先隊列的堆又稱爲二叉堆。二叉堆允許在 O(logn) 時間內排隊和出隊，這對於高效的調度系統是非常重要的。

二叉堆是很有趣的一種結構，雖然它定義上是二叉樹，但我們不必真的用鏈接的節點來實現二叉堆，相反我們採取前面提到過的用數組、切片或 Vec 這類線性數據結構來實現。只要我們的操作是按照二叉樹的定義，那麼線性數據結構一樣可以當成非線性數據結構來用。其實真正的樹在內存裏也是線性放置的，因爲內存本身就是線性的，只是使用時採取非線性方式。

二叉堆有兩種常見的形態，一種是最小堆，或稱小頂堆，最小的數據項在堆頂；另一種是最大堆，或稱大頂堆，最大數據項在堆頂。不管大頂還是小頂堆，算法邏輯除了取大或取小外沒有差別。

7.3.1 二叉堆定義

根據前面的描述，我們選擇定義小頂二叉堆的基本操作如下。

new() 創建一個新的二叉堆，不需要參數，返回空堆。

push(k) 向堆添加一個新項，需要參數 k，不返回任何內容。

pop() 返回堆的最小項，從堆中刪除該項，不需要參數，修改堆。

min() 返回堆的最小項，不需要參數，不修改堆。

size() 返回堆中的項數，不需要參數，返回數字。

is_empty() 返回堆狀態，不需要參數，返回布爾值。

build(arr) 從數組或 vec 構建新堆，需要保存數據的參數 arr。

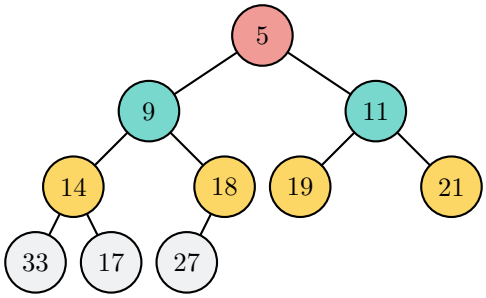
假設 h 是已經創建的二叉堆，下表展示了堆操作序列後的結果，堆頂在右邊。此處將加入項的值作爲優先級，越小則越優先，所以小的在右側。

表 7.2: 二叉堆操作

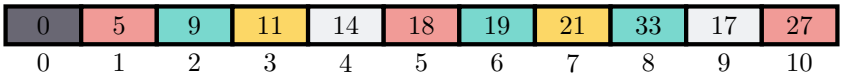
堆操作	堆當前值	操作返回值	堆操作	堆當前值	操作返回值
h.is_empty()	[]	true	h.is_empty()	[8,6,3]	false
h.push(8)	[8]		h.push(4)	[8,6,4,3]	
h.push(6)	[8,6]		h.min()	[8,6,4,3]	3
h.min()	[8,6]	6	h.pop()	[8,6,4]	3
h.push(3)	[8,6,3]		h.pop()	[8,6]	4
h.size()	[8,6,3]	3	h.build([1,2])	[8,6,2,1]	

7.3.2 Rust 實現二叉堆

爲使二叉堆高效工作，可以利用其對數性質。二叉堆採用線性數據結構來保存，爲保證性能，必須保持二叉堆平衡。平衡二叉堆在根的左右子樹中具有大致相同數量的節點，它儘量將每個節點的左右子節點填滿，最多有一個節點的子節點不滿。



用 Vec 保存二叉堆，因父子節點處於線性數據結構中，所以父子節點的關係易於計算。一個節點若處於下標 p，則其左子節點在 2p，右子節點在 2p + 1 ，p 爲從 1 開始的下標，下標爲 0 處不放數據，直接置 0 佔位。Vec [0,5,9,11,14,18,19,21,33,17,27]，其樹表示如下。



可以看到 5 的下標 p 爲 1，左子節點在下標 2*p = 2 處，此處值爲 9，樹結構圖中 9 正好是 5 的左子節點。同樣的，任意子節點的父節點位於下標 p/2 處，比如 9 的下標 p = 2，則父節點在 2/2 = 1 處，右子節點 11 的下標爲 p = 3，則它的父節點在 3/2 = 1 處，除法值向下取整。綜上，任意子節點的父節點計算只用一個計算表達式 p/2，而子節點的計算爲 2p 和 2p + 1。前面我們定義過宏來計算父子節點下標，此處我們依然使用宏來計算。

```
1 // binary_heap.rs
2 macro_rules! parent { //計算父節點下標
3     ($child:ident) => {
4         $child >> 1
```

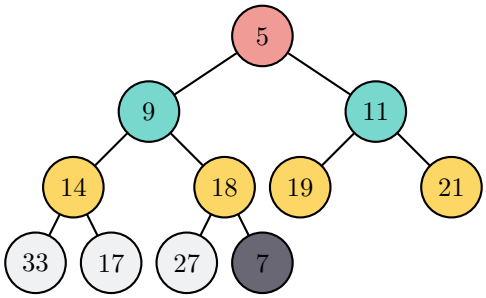
```
    };
}
macro_rules! left_child { //計算左子節點下標
    ($parent:ident) => {
        $parent << 1
    };
}
macro_rules! right_child { //計算右子節點下標
    ($parent:ident) => {
        ($parent << 1) + 1
    };
}
```

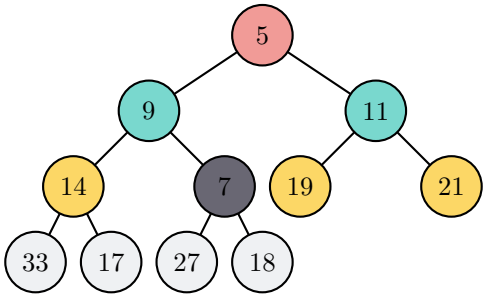
首先定義二叉堆。爲跟蹤堆大小情況，添加了一個表示數據項的字段 size，注意第一個數據 0 不算。初始化時，下標 0 處有數據但 size 也置爲 0，此處保存的數據默認爲 i32。

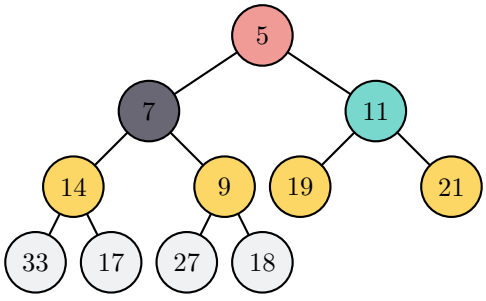
```
// binary_heap.rs

// 二叉堆定義
#[derive(Debug, Clone)]
struct BinaryHeap {
    size: usize,    // 數據量
    data: Vec<i32>, // 數據容器
}

impl BinaryHeap {
    fn new() -> Self {
        BinaryHeap {
            size: 0, // vec 首位置 0，但不計入總數
            data: vec![0]
        }
    }

    fn len(&self)  -> usize {
        self.size
    }

    fn is_empty(&self) -> bool {
        0 == self.size
```

```
24      }
25
26      // 獲取堆中最小數據
27      fn min(&self) -> Option<i32> {
28          if self.size == 0 {
29              None
30          } else {
31              // Some(self.data[1].clone()); 泛型數據用 clone
32              Some(self.data[1])
33          }
34      }
35  }
```

有了空堆，可以加入數據項。在堆尾加入數據可能破壞平衡，所以需要向上移動到堆頂。







每加入一個數據，size 加一，然後從此開始往上移動 move_up （若需要）以維持平衡。

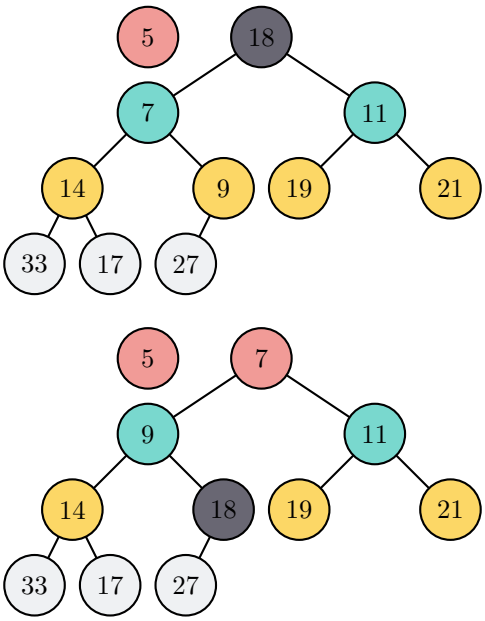
```
// binary_heap.rs

impl BinaryHeap {
    // 末尾添加數據，調整堆
    fn push(&mut self, val: i32) {
        self.data.push(val);
        self.size += 1;
        self.move_up(self.size);
    }

    // 小數據上冒
    fn move_up(&mut self, mut c: usize) {
        loop {
            let p = parent!(c);
            if p <= 0 { break; }

            if self.data[c] < self.data[p] {
                self.data.swap(c, p);
            }
            c = p;
        }
    }
}
```

假設要獲取堆最小值，則要考慮三種情況：堆中無數據返回 None；堆中有一個數據，直接彈出；堆中有多項數據，交換堆頂和堆尾數據，調整堆，之後返回末尾最小值。下面實現了元素向下移動功能以維持平衡，min_child 用於找出最小子節點。

```
// binary_heap.rs

impl BinaryHeap {
    fn pop(&mut self) -> Option<i32> {
        if 0 == self.size { // 無數據，返回 None
            None
        } else if 1 == self.size {
            self.size -= 1; // 一個數據，比較好處理
            self.data.pop()
        } else { // 多個數據，先交換并彈出數據，再調整堆
```

```
11              self.data.swap(1, self.size);
12              let val = self.data.pop();
13              self.size -= 1;
14              self.move_down(1);
15              val
16          }
17      }
18
19      // 大數據下沉
20      fn move_down(&mut self, mut c: usize) {
21          loop {
22              let lc = left_child!(c);
23              if lc > self.size { break; }
24
25              let mc = self.min_child(c);
26              if self.data[c] > self.data[mc] {
27                  self.data.swap(c, mc);
28              }
29              c = mc;
30          }
31      }
32
33      // 最小子節點位置
34      fn min_child(&self, i: usize) -> usize {
35          let (lc, rc) = (left_child!(i), right_child!(i));
36          if rc > self.size {
37              lc
38          } else if self.data[lc] < self.data[rc] {
39              lc
40          } else {
41              rc
42          }
43      }
44  }
```

下面來看看刪除堆中最小元素的過程，如下圖。首先將堆頂部的元素拿出，並將最後一個元素移動到堆頂部。此時堆頂不是最小值，所以不滿足堆定義，需要重新建堆。此時建堆需要將頂部元素其向下移動 move_down，通過節點下標計算宏來計算該和左還是右字節點交換，最終頂部元素 18 和 7 左子節點交換，重複這個過程，直到 18 到達某個節點。



除了一個個地 push 到堆中外，還可以通過對集合集中處理。比如 [5,4,3,1,2] 這個切片，可以一次性加入堆中，避免頻繁調用 push 函數。假設原始堆中有數據 [6,7,8,9,10]，則加入切片數據有兩種方式，一是保持原始數據，切片數據爲新加，則最終堆爲 [1,2,3,4,5,6,7,8,9,10]。二是刪除原始數據，再添加切片元素，則最終堆爲 [1,2,3,4,5]。

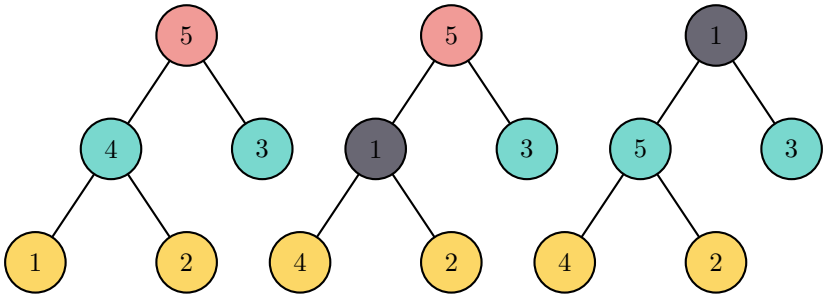


图 7.2: 刪除數據並重新構建堆

爲實現這兩種功能，分別定義了 build_new 和 build_add 函數如下。

```
1  // binary_heap.rs
2
3  impl BinaryHeap {
4      // 構建新堆
5      fn build_new(&mut self, arr: &[i32]) {
6          // 剔除原始數據
7          for _i in 0..self.size {
```

```
8               let _rm = self.data.pop();
9           }
10
11          // 添加新數據
12          for &val in arr {
13              self.data.push(val);
14          }
15          self.size = arr.len();
16
17          // 調整小頂堆
18          let size = self.size;
19          let mut p = parent!(size);
20          while p > 0 {
21              self.move_down(p);
22              p -= 1;
23          }
24      }
25
26      // 切片數據逐個加入堆
27      fn build_add(&mut self, arr: &[i32]) {
28          for &val in arr {
29              self.push(val);
30          }
31      }
32  }
```

至此我們完成了二叉小頂堆的構建，整個過程理解起來應該非常簡單。當然，你可以據此寫出大頂堆的代碼。二叉小頂堆的完整代碼如下。

```
1   // binary_heap.rs 完整代碼
2
3   // 計算節點下標的宏
4   macro_rules! parent {
5       ($child:ident) => {
6           $child >> 1
7       };
8   }
9
10  macro_rules! left_child {
```

```
    ($parent:ident) => {
        $parent << 1
    };
}

macro_rules! right_child {
    ($parent:ident) => {
        ($parent << 1) + 1
    };
}

// 定義
#[derive(Debug, Clone)]
struct BinaryHeap {
    size: usize,    // 數據量
    data: Vec<i32>, // 數據容器
}

impl BinaryHeap {
    fn new() -> Self {
        BinaryHeap {
            data: vec![0],
            size: 0,
        }
    }

    fn len(&self)  -> usize {
        self.size
    }

    fn is_empty(&self) -> bool {
        0 == self.size
    }

    // 獲取堆中最小數據
    fn min(&self) -> Option<i32> {
        if self.size == 0 {
            None
```

```
49          } else {
50              Some(self.data[1])
51          }
52      }
53
54      // 末尾添加一個數據，調整堆
55      fn push(&mut self, val: i32) {
56          self.data.push(val);
57          self.size += 1;
58          self.move_up(self.size);
59      }
60
61      // 小數據上冒 c(child), p(parent)
62      fn move_up(&mut self, mut c: usize) {
63          loop {
64              // 計算當前節點的父節點
65              let p = parent!(c);
66              if p <= 0 { break; }
67
68              // 當前節點數據小於父節點數據，交換
69              if self.data[c] < self.data[p] {
70                  self.data.swap(c, p);
71              }
72
73              // 父節點成爲當前節點
74              c = p;
75          }
76      }
77
78      fn pop(&mut self) -> Option<i32> {
79          if 0 == self.size { // 無數據，返回 None
80              None
81          } else if 1 == self.size {
82              self.size -= 1; // 一個數據，比較好處理
83              self.data.pop()
84          } else { // 多個數據，先交換并彈出數據，再調整堆
85              self.data.swap(1, self.size);
86              let val = self.data.pop();
```

```
87              self.size -= 1;
88              self.move_down(1);
89              val
90          }
91      }
92
93      // 大數據下沉 l(left), r(right)
94      fn move_down(&mut self, mut c: usize) {
95          loop {
96              // 計算當前節點的左子節點位置
97              let lc = left_child!(c);
98              if lc > self.size { break; }
99
100             // 計算當前節點的最小子節點位置
101             let mc = self.min_child(c);
102
103             // 當前節點數據大於最小子節點數據，交換
104             if self.data[c] > self.data[mc] {
105                 self.data.swap(c, mc);
106             }
107
108             // 最小子節點成爲當前節點
109             c = mc;
110         }
111     }
112
113     // 計算最小子節點位置
114     fn min_child(&self, c: usize) -> usize {
115         // 同時計算左右子節點位置
116         let (lc, rc) = (left_child!(c), right_child!(c));
117
118         // 1.如果右子節點位置超過 size，表示只有左子節點
119         // 則左子節點就是最小子節點
120         // 2.否則，同時存在左右子節點，需具體判斷左右子
121         // 節點數據大小，然後返回最小的子節點位置
122         if rc > self.size {
123             lc
124         } else if self.data[lc] < self.data[rc] {
```

```
125             lc
126         } else {
127             rc
128         }
129     }
130
131     fn build_new(&mut self, arr: &[i32]) {
132         // 剔除原始數據
133         for _i in 0..self.size {
134             let _rm = self.data.pop();
135         }
136
137         // 添加新數據
138         for &val in arr {
139             self.data.push(val);
140         }
141         self.size = arr.len();
142
143         // 調整堆，使其爲小頂堆
144         let sz = self.size;
145         let mut p = parent!(sz);
146         while p > 0 {
147             self.move_down(p);
148             p -= 1;
149         }
150     }
151
152     // 切片數據逐個加入堆
153     fn build_add(&mut self, arr: &[i32]) {
154         for &val in arr {
155             self.push(val);
156         }
157     }
158 }
159
160 fn main() {
161     let mut bh = BinaryHeap::new();
162     let nums = [-1,0,2,3,4];
```

```
163     bh.push(10); bh.push(9);
164     bh.push(8); bh.push(7); bh.push(6);
165
166     bh.build_add(&nums);
167     println!("empty: {:?}", bh.is_empty());
168     println!("min: {:?}", bh.min());
169     println!("pop min: {:?}", bh.pop());
170
171     bh.build_new(&nums);
172     println!("size: {:?}", bh.len());
173     println!("pop min: {:?}", bh.pop());
174 }
```

7.3.3 二叉堆分析

二叉堆雖然是放到 Vec 裏面，線性放置的，但其排序是按照樹的方式來操作的。前面學習樹時就分析過，樹高度是 $O(nlog_2(n))$，而堆排序就是從樹底層移到頂層，移動步驟爲樹的層數，所以排序複雜度應該是 $O(nlog_2(n))$。構建堆需要處理所有 n 項數據，所以複雜度是 O(n)。

7.4 二叉查找樹

二叉堆用線性數據結構模擬樹操作，但這隻適合少量數據。一旦數據多了，數據複製移動非常耗時，所以本節來研究用節點實現樹。我們定義的樹只有兩個子節點，這種樹被稱爲二叉樹。二叉樹是最基本的樹，許多樹都是從二叉樹衍生的。爲了使用和分析二叉樹，本節來研究一種用於查找的二叉樹：二叉查找樹。

我們通過學習樹在查找任務上的性能來熟悉這種數據結構。前面學習 HashMap 時，是用鍵來獲取值，二叉查找樹類似 HashMap，也是用鍵值來存值。

7.4.1 二叉查找樹操作

二叉查找樹用於查找，所以可以定義二叉查找樹如下的抽象數據類型。

new() 創建一棵新樹，不需要參數，返回一個空樹。

insert(k, v) 將數據 (k, v) 存儲到樹，需要 k 鍵，v 值，不返回任何內容。

search(&k) 在樹中查找是否包含鍵 k，需要參數 &k，返回布爾值。

get(&k) 從樹返回鍵 k 的值 v，但不會刪除它，需要參數 &k。

max() 返回樹中最大鍵 k 及其值 v，不需要參數。

min() 返回樹中最小鍵 k 及其值 v，不需要參數。

len() 返回樹中數據量，不需要參數，返回一個 usize 型整數。

is_empty() 測試樹是否爲空，不需要參數，返回布爾值。
iter() 返回樹的迭代形式，不需要參數，不改變樹。
preorder() 前序遍歷，不需要參數，輸出各個 k-v 值。
inorder() 中序遍歷，不需要參數，輸出各個 k-v 值。
postorder() 後序遍歷，不需要參數，輸出各個 k-v 值。
假設 t 是空二叉查找樹，下表展示了各種操作後的結果，此處用元組來表示樹。

表 7.3: 二叉查找樹操作

二叉樹操作	二叉樹當前值	操作返回值
t.is_empty()	[]	true
t.insert(1,'a')	[(1,'a')]	
t.insert(2,'b')	[(1,'a'),(2,'b')]	
t.len()	[(1,'a'),(2,'b')]	2
t.get(&4)	[(1,'a'),(2,'b')]	None
t.get(&2)	[(1,'a'),(2,'b')]	Some('b')
t.min()	[(1,'a'),(2,'b')]	(Some(1), Some('a'))
t.max()	[(1,'a'),(2,'b')]	(Some(2), Some('b'))
t.search(2)	[(1,'a'),(2,'b')]	true
t.insert(2,'c')	[(1,'a'),(2,'c')]	

7.4.2 Rust 實現二叉查找樹

不同於堆的左右子節點不考慮大小關係，二叉查找樹左子節點鍵要小於父節點的鍵，右子節點的鍵要大於父節點鍵。也就是 left < parent < right 這個規律，其遞歸地適用於所有子樹。

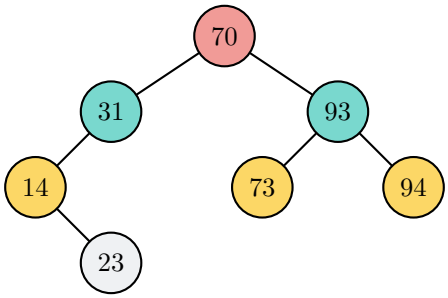


图 7.3: 二叉查找樹

上圖中，70 作爲根節點，31 比 70 小，作爲左節點，93 更大，作爲右節點。接着插入 14，比 70 小，下降到 31 ，比 31 還小，則作爲 31 的左節點，有其他數據插入同理，最終形成了二叉查找樹。該樹的中序遍歷是 [14,23,31,70,73,93,94]，正是從小到大排序的數據，所以二

叉查找樹也可以用來排序數據，只要使用中序遍歷就能得到升序排序，使用中序遍歷的鏡像遍歷法也就是按照“右根左”的順序遍歷就得到降序排序結果。爲實現樹，我們定義樹爲一個結構體 BST（BinarySearchTree）。按照抽象數據類型的定義，下面是實現的二叉查找樹。

```
// bst.rs
use std::cmp::Ordering;
use std::ops::Deref;

// 二叉查找樹子節點鏈接
type Link<T, U> = Option<Box<BST<T,U>>>;

// 二叉查找樹定義
struct BST<T,U> {
    key: Option<T>,
    val: Option<U>,
    left: Link<T,U>,
    right: Link<T,U>,
}

impl<T,U> BST<T,U>
where T: Clone + Ord + std::fmt::Debug,
      U: Clone + std::fmt::Debug {
    fn new() -> Self {
        BST {
            key: None, val: None, left: None, right: None,
        }
    }

    fn is_empty(&self) -> bool {
        self.key.is_none()
    }

    fn len(&self) -> usize {
        self.calc_len(0)
    }

    // 遞歸計算節點個數
    fn calc_len(&self, mut i: usize) -> usize {
        if self.key.is_none() {
```

```
36              return i;
37          }
38
39          // 當前節點加入總節點數 i
40          i += 1;
41
42          // 計算左右子節點數
43          if !self.left.is_none() {
44              i = self.left.as_ref().unwrap().calc_len(i);
45          }
46          if !self.right.is_none() {
47              i = self.right.as_ref().unwrap().calc_len(i);
48          }
49
50          i
51      }
52
53      // 前中後序遍歷
54      fn preorder(&self) {
55          println!("key:{:#?},val:{:#?}",&self.key,&self.val);
56          match &self.left {
57              Some(node) => node.preorder(),
58              None => (),
59          }
60          match &self.right {
61              Some(node) => node.preorder(),
62              None => (),
63          }
64      }
65
66      fn inorder(&self) {
67          match &self.left {
68              Some(node) => node.inorder(),
69              None => (),
70          }
71          println!("key:{:#?},val:{:#?}",&self.key,&self.val);
72          match &self.right {
73              Some(node) => node.inorder(),
```

```
74              None => (),
75          }
76      }
77
78      fn postorder(&self) {
79          match &self.left {
80              Some(node) => node.postorder(),
81              None => (),
82          }
83          match &self.right {
84              Some(node) => node.postorder(),
85              None => (),
86          }
87          println!("key:{:#?},val:{:#?}",&self.key,&self.val);
88      }
89
90      fn insert(&mut self, key: T, val: U) {
91          // 無數據，直接插入
92          if self.key.is_none() {
93              self.key = Some(key);
94              self.val = Some(val);
95          } else {
96              match &self.key {
97                  Some(k) => {
98                      // 存在 key，更新 val
99                      if key == *k {
100                         self.val = Some(val);
101                         return;
102                     }
103
104                     // 未找到 key，需要插入新節點
105                     // 先找到需要插入的子樹
106                     let child = if key < *k {
107                         &mut self.left
108                     } else {
109                         &mut self.right
110                     };
111
```

```
                    // 根據節點遞歸下去，直到插入
                    match child {
                        Some(ref mut node) => {
                            node.insert(key, val);
                        },
                        None => {
                            let mut node = BST::new();
                            node.insert(key, val);
                            *child = Some(Box::new(node));
                        },
                    }
                },
                None => (),
            }
        }
    }

    fn search(&self, key: &T) -> bool {
        match &self.key {
            Some(k) => {
                // 比較 key 值，判斷是否繼續遞歸查找
                match k.cmp(&key) {
                    Ordering::Equal => { true }, // 找到數據
                    Ordering::Greater => { // 在左子樹查找
                        match &self.left {
                            Some(node) => node.search(key),
                            None => false,
                        }
                    },
                    Ordering::Less => { // 在右子樹查找
                        match &self.right {
                            Some(node) => node.search(key),
                            None => false,
                        }
                    },
                }
            },
            None => false,
```

```
150         }
151     }
152
153     fn min(&self) -> (Option<&T>, Option<&U>) {
154         // 最小值一定在最左側
155         match &self.left {
156             Some(node) => node.min(),
157             None => match &self.key {
158                 Some(key) => (Some(&key), self.val.as_ref()),
159                 None => (None, None),
160             },
161         }
162     }
163
164     fn max(&self) -> (Option<&T>, Option<&U>) {
165         // 最大值一定在最右側
166         match &self.right {
167             Some(node) => node.max(),
168             None => match &self.key {
169                 Some(key) => (Some(&key), self.val.as_ref()),
170                 None => (None, None),
171             },
172         }
173     }
174
175     // 獲取值，和查找流程相似
176     fn get(&self, key: &T) -> Option<&U> {
177         match &self.key {
178             None => None,
179             Some(k) => {
180                 match k.cmp(&key) {
181                     Ordering::Equal => self.val.as_ref(),
182                     Ordering::Greater => {
183                         match &self.left {
184                             None => None,
185                             Some(node) => node.get(key),
186                         }
187                     },
```

```
188                         Ordering::Less => {
189                             match &self.right {
190                                 None => None,
191                                 Some(node) => node.get(key),
192                             }
193                         },
194                     }
195                 },
196             }
197         }
198 }
199
200 fn main() {
201     let mut bst = BST::<i32,char>::new();
202     bst.insert(8,'e'); bst.insert(6,'c'); bst.insert(7,'d');
203     bst.insert(5,'b'); bst.insert(10,'g'); bst.insert(9,'f');
204     bst.insert(11,'h'); bst.insert(4,'a');
205
206     println!("empty: {:?}, len: {:?}", bst.is_empty(), bst.len());
207     println!("max: {:?}, min: {:?}", bst.max(), bst.min());
208     println!("key: 5, val: {:?}", bst.get(&5));
209     println!("5 in bst: {:?}", bst.search(&5));
210
211     println!("inorder: "); bst.inorder();
212     println!("preorder: "); bst.preorder();
213     println!("postorder: "); bst.postorder();
214 }
```

有了樹就可以考慮插入問題了，通過使用 insert 函數向樹中插入數據 76。如下圖。

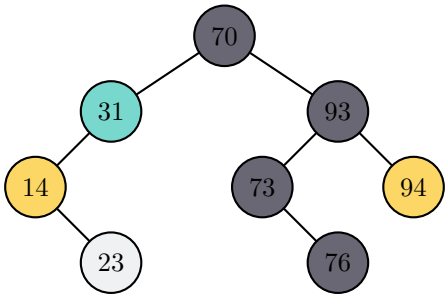


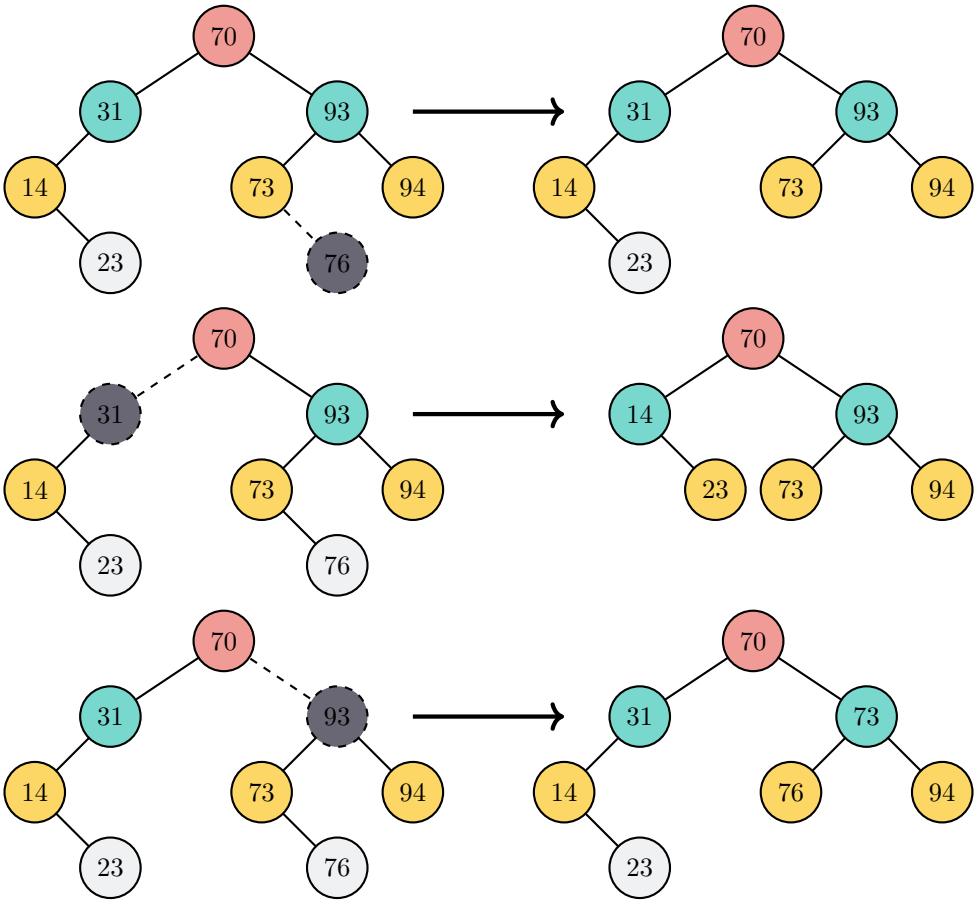
图 7.4: 二叉查找樹插入數據

最後來看最複雜的操作：刪除節點。刪除一個鍵，首先要找到它。此時樹可能存在三種情況：樹沒有節點、樹有一根節點、樹有若干節點。針對後兩種情況，都需要檢查要刪除的節點是否存在。若該節點存在，則此時需要考慮該節點是否有子節點，又有三種情況：該節點是葉節點，該節點有一子節點，該節點有兩個子節點。

表 7.4: k 節點刪除狀況

序	樹節點狀況	k 子節點	刪除方法	序	樹節點狀況	k 子節點	刪除方法
1	無節點	無	直接返回	4	有多個節點	有一個子節點	子節點替換 k
2	有一根節點	無	直接刪除	5	有多個節點	有兩個子節點	後繼節點替換 k
3	有多個節點	無	直接刪除				

在找到節點 k 的情況下，若它是葉節點，無子節點，則直接刪除父節點對其的引用。若它有一個子節點，則修改父節點的引用直接指向子節點。若它有兩個子節點，則找到右子樹中的最小節點，該節點又叫後繼節點，它是右子樹中最小值，用該節點直接替換 k 就相當於刪除了 k。當然該後繼節點本身可能有零個或一個子節點，替換時也需要處理後繼節點的父子引用關係。具體的情況如下面的圖所示，虛線框爲待刪除節點 k，右側爲刪除後的二叉樹。



7.4.3 二叉查找樹分析

我們終於完成了二叉查找樹，現在可以來看看各個函數的時間複雜度了。對於三種遍歷，因爲要處理所有 n 個數據，所以複雜度一定是 O(n)。len() 也利用了前序遍歷的方法來計算元素個數，所以也爲 O(n)。

search 函數查找數據，因爲它會不斷和左右子節點比較，並根據比較結果選擇一條分支，那麼它最多走樹中從根到葉節點的最長路徑。根據二叉樹的性質，我們知道，樹的節點總數數爲

$$2^0 + 2^1 + 2^i ... + 2^h = n \tag{7.1}$$

可得最大路徑長度爲 $log_2(n)$ 左右，所以 search 的性能爲 $O(log_2(n))$。增刪查改的基礎是查，因爲你需要定位元素位置才能繼續處理。增刪改都能在常量時間內完成，結果其時間複雜度只和查找有關。綜上 search，insert，remove, get 的性能都是 $O(log_2(n))$，限制這些方法性能的因素是二叉樹的高度 h。當然，如果插入的數據一直處於有序狀態，那麼樹會退化成線性鏈表，此時 search，insert, remove 的性能均爲 O(n)，如下圖所示。

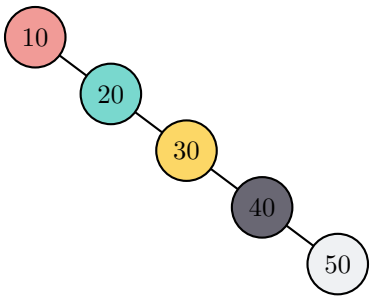


图 7.5: 二叉查找遭遇線性狀況

二叉樹具有 $O(log_2(n))$ 的時間複雜度，這是非常優秀的。因爲樹高 h 和 n 有關係。只要通過改變子節點數量，讓二叉樹衍生爲多叉樹，既每個節點保存多個子節點，比如 1000 個。這樣能大幅度降低樹高度，性能會更好。比較常見的多叉樹是 B 樹，B+ 樹，它們的子節點都比較多，樹很矮，查詢非常快，廣泛用於實現數據庫和文件系統。

比如 MySQL 數據庫底層就是用的 B+ 樹來保存數據，它的節點是一個 16K 大的內存頁。如果一條數據爲 1k 大小，那麼一個節點能存 16 條數據。如果用於中間層存儲索引，鍵使用 bigint，8 字節，索引 6 字節，共 14 字節，則 16k 能存放 16 * 1024 / 14 = 1170 個索引。對於高度爲 3 的 B+ 樹，能存放的索引有 1170 * 1170 * 16 = 21902400 條，也就意味着能存儲大概二千萬條數據。而每次獲取數據最多需要兩次查詢，因爲高度爲 3，幾乎就是常量時間，這也是數據庫查詢速度快的原因。對數據庫感興趣的讀者可以去閱讀 MySQL 相關書籍瞭解更多內容。

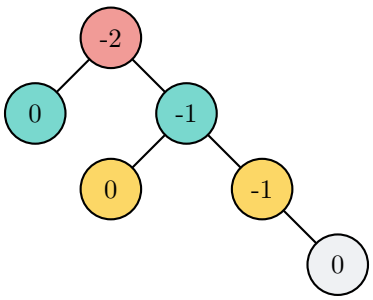
7.5 平衡二叉樹

在前面一節中，我們學習並構建了一個二叉查找樹，其性能在某些特殊情況可能降級到 $O(n)$。比如樹不平衡，一側有非常多節點，而另一側幾乎沒有，則其性能會退化，導致後續操作都非常低效。所以，構建一個平衡的二叉樹是高效處理數據的前提。在本節中，我們將討論一種平衡的二叉查找樹，它能自動保持平衡狀態。這種平衡的二叉查找樹稱爲 AVL 樹，名字來源於其發明人：G.M. Adelson-Velskii 和 E.M.Land。

AVL 樹的是普通二叉查找樹，唯一區別是樹的操作執行方式。爲實現 AVL 樹，操作過程中需要跟踪樹中節點的平衡因子。平衡因子是節點左右子樹高度差，其定義爲：

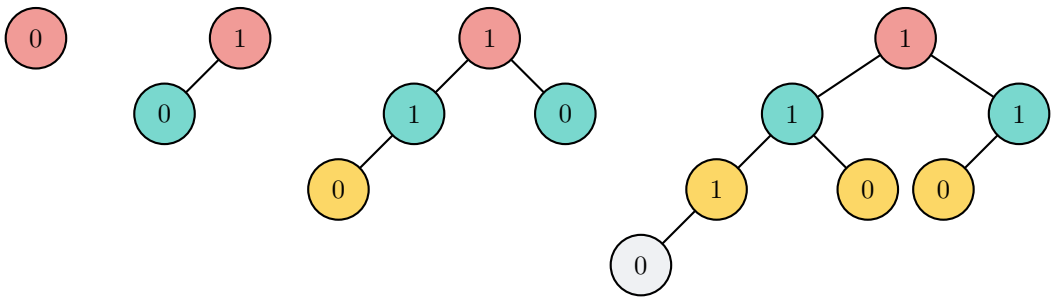
$$balanceFactor = height(leftSubTree) - height(rightSubTree)$$

使用這個平衡因子定義，如果平衡因子大於零，則左子樹重，如果平衡因子小於零，則右子樹重；如果平衡因子是零，那麼樹是平衡的。爲了實現高效的 AVL 樹，可以將平衡因子 -1，0，1 三種情況均視爲平衡，因爲平衡因子爲 1，-1 時，左右子樹高度差只有 1，基本平衡。一旦樹中節點平衡因子處於這個範圍之外，比如 2，-2，則需要將樹旋轉使之維持平衡。下圖展示了左右子樹不平衡的情況，每個節點上標示的值就是該節點的平衡因子。



7.5.1 AVL 平衡二叉樹

要得到平衡的樹，則需要滿足平衡因子要求。樹只有三種情況，左重，平衡，右重。只要樹滿足左重或右重的情況下依然滿足平衡因子爲-1，0，1 的要求，則這樣的左或右重樹還是平衡的。考慮高度爲 0，1，2，3 的樹，下圖是滿足平衡因子條件的最不平衡左重樹。



分析樹中節點總數，可以發現對於高度爲 0 的樹，只有 1 個節點；對於高度爲 1 的樹，有 $1+1=2$ 個節點；對於高度爲 2 的樹則有 $1+1+2=4$ 個節點；對於高度爲 3 的樹，有 $1+2+4=7$ 個節點。綜上，更一般地，可以看到高度爲 h 的樹中節點數量滿足如下式：

$$N_h = 1 + N_{h-1} + N_{h-2}$$

這個公式看起來非常類似於斐波那契數列。給定樹中節點數量，可以利用斐波那契公式來導出 AVL 樹的高度公式。對於斐波那契數列，第 i 個斐波那契數由下式給出：

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \end{aligned} \tag{7.2}$$

這樣，可以將 AVL 樹高度和節點公式轉化爲下式：

$$N_h = F_{h+2} - 1$$

隨着 i 增大，F_i/F_{i-1} 趨近於黃金比率 $\Phi = (1+\sqrt{5})/2$，所以可以使用 Φ 來表示 F_i，通過計算可得 $F_i = \frac{\Phi^i}{5}$，則：

$$N_h = \frac{\Phi^{h+2}}{\sqrt{5}} - 1$$

通過取 2 爲底的對數，可以求解 h。

$$\begin{aligned} log(N_h+1) &= log(\frac{\Phi^{h+2}}{\sqrt{5}}) \\ log(N_h+1) &= (h+2)log\Phi - \frac{1}{2}log5 \\ h &= \frac{log(N_h+1) - 2log\Phi + \frac{1}{2}log5}{log\Phi} \\ h &= 1.44log(N_h) \end{aligned} \tag{7.3}$$

所以，AVL 樹的高度最高等於樹中節點數目的對數值的 1.44 倍，忽略係數，則查找時的複雜度限制爲 $O(logN)$，這還是非常高效的。

7.5.2 Rust 實現平衡二叉樹

現在來看看如何插入新節點到 AVL 樹。由於所有新節點將作爲葉節點插入到樹中，並且葉的平衡因子爲 0，所以剛插入的新節點不用處理。但添加了新葉後，其父節點的平衡因子會改變，所以需要更新父節點的平衡因子。新插入的葉節點如何影響父節點的平衡因子取決於葉節點是左子節點還是右子節點。如果新節點是右子節點，則父節點的平衡因子將減 1；如果新節點是左子節點，則父節點的平衡因子將加 1。

這個關係可以應用到新節點的祖父節點，直到樹根，這是一個遞歸過程。有兩種情況不會更新平衡因子：（1）遞歸調用已到達樹根。（2）父節點的平衡因子已調整爲零，其祖先節點的平衡因子不會改變。

爲了簡潔，我們將 AVL 樹實現爲枚舉體，其中 Null 表示空樹，Tree 表示存在對樹節點的引用。樹節點 AvlNode 用於存放數據和左右子樹及平衡因子。

```
// avl.rs

// Avl 樹定義
#[derive(Debug)]
enum AvlTree<T> {
    Null,
    Tree(Box<AvlNode<T>>),
}

// Avl 樹節點定義
#[derive(Debug)]
struct AvlNode<T> {
    val: T,
    left: AvlTree<T>,
    right: AvlTree<T>,
    bfactor: i8,
}
```

首先需要爲 AVL 樹添加插入節點的功能 insert，然而插入節點後又需要處理平衡因子，所以還需要添加一個再平衡函數 rebalance 用於更新平衡因子。爲實現節點數據的比較，數據需要滿足排序 Ord 特性，所以引入了 Ording 做比較。爲更新值和計算樹高還需要 replace 和 max 函數。

```
// avl.rs

use std::cmp::Ordering::*;
use std::cmp::max;
use std::mem::replace;
use AvlTree::*;

impl<T> AvlTree<T> where T : Ord {
    // 新樹是空的
    fn new() -> AvlTree<T> {
        Null
```

```
12      }
13
14      fn insert(&mut self, val: T) -> (bool, bool) {
15          let ret = match *self {
16              // 無節點，直接插入
17              Null => {
18                  let node = AvlNode {
19                      val: val,
20                      left: Null,
21                      right: Null,
22                      bfactor: 0,
23                  };
24                  *self = Tree(Box::new(node));
25
26                  (true, true)
27              },
28              Tree(ref mut node) => match node.val.cmp(&val) {
29                  // 比較節點值，再判斷該從哪邊插入
30                  // inserted 表示是否插入
31                  // deepened 表示是否加深
32                  Equal => (false, false), // 相等，無需插入
33                  Less => { // 比節點數據大，插入右邊
34                      let (inserted, deepened) = node.right
35                                                     .insert(val);
36                      if deepened {
37                          let ret = match node.bfactor {
38                              -1 => (inserted, false),
39                               0 => (inserted, true),
40                               1 => (inserted, false),
41                               _ => unreachable!(),
42                          };
43                          node.bfactor += 1;
44
45                          ret
46                      } else {
47                          (inserted, deepened)
48                      }
49                  },
```

```
50                  Greater => { // 比節點數據小, 插入左邊
51                      let (inserted, deepened) = node.left
52                                                      .insert(val);
53                      if deepened {
54                          let ret = match node.bfactor {
55                              -1 => (inserted, false),
56                               0 => (inserted, true),
57                               1 => (inserted, false),
58                               _ => unreachable!(),
59                          };
60                          node.bfactor -= 1;
61
62                          ret
63                      } else {
64                          (inserted, deepened)
65                      }
66                  },
67              },
68          };
69          self.rebalance();
70
71          ret
72      }
73
74      // 調整各節點的平衡因子
75      fn rebalance(&mut self) {
76          match *self {
77              // 無數據, 不用調整
78              Null => (),
79              Tree(_) => match self.node().bfactor {
80                  // 右子樹重
81                  -2 => {
82                      let lbf = self.node().left.node().bfactor;
83                      if lbf == -1 || lbf == 0 {
84                          let (a, b) = if lbf == -1 {
85                              (0, 0)
86                          } else {
87                              (-1,1)
```

```
88                          };
89                          self.rotate_right(); // 不平衡，旋轉
90                          self.node().right.node().bfactor = a;
91                          self.node().bfactor = b;
92                      } else if lbf == 1 {
93                          let (a, b) = match self.node()
94                                                 .left.node()
95                                                 .right.node()
96                                                 .bfactor {
97                              -1 => (1,0),
98                               0 => (0,0),
99                               1 => (0,-1),
100                              _ => unreachable!(),
101                         };
102
103                         // 先左旋再右旋
104                         self.node().left.rotate_left();
105                         self.rotate_right();
106                         self.node().right.node().bfactor = a;
107                         self.node().left.node().bfactor = b;
108                         self.node().bfactor = 0;
109                     } else {
110                         unreachable!()
111                     }
112                 },
113                 // 左子樹重
114                 2 => {
115                      let rbf=self.node().right.node().bfactor;
116                      if rbf == 1 || rbf == 0 {
117                          let (a,b) = if rbf == 1 {
118                             (0,0)
119                          } else {
120                             (1,-1)
121                          };
122                          self.rotate_left();
123                          self.node().left.node().bfactor = a;
124                          self.node().bfactor = b;
125                      } else if rbf == -1 {
```

```
                            let (a, b) = match self.node()
                                                .right.node()
                                                .left.node()
                                                .bfactor {
                                 1 => (-1,0),
                                 0 => (0,0),
                                -1 => (0,1),
                                 _ => unreachable!(),
                            };

                            // 先右旋再左旋
                            self.node().right.rotate_right();
                            self.rotate_left();
                            self.node().left.node().bfactor = a;
                            self.node().right.node().bfactor = b;
                            self.node().bfactor = 0;
                        } else {
                            unreachable!()
                        }
                    },
                    _ => (),
                },
            }
        }
    }
```

rebalance 函數完成了再平衡工作，insert 完成了平衡因子更新，這個過程是遞歸進行的。有效的再平衡是使 AVL 樹在不犧牲性能的情況下正常工作的關鍵。爲使 AVL 樹恢復平衡，需要將樹執行一次或多次旋轉，可能是左旋轉也可能是右旋轉。

要執行左旋轉，要執行的操作如下：

(1) 提升右孩子 (B) 爲子樹的根。

(2) 將舊根 (A) 移動爲新根的左子節點。

(3) 如果新根 (B) 已經有一個左孩子，那麼使它成爲新左孩子 (A) 的右孩子。

要執行右旋轉，要執行的操作如下：

(1) 提升左孩子 (B) 爲子樹的根。

(2) 將舊根 (A) 移動爲新根的右子節點。

(3) 如果新根 (B) 已經有一個右孩子，那麼使它成爲新右孩子 (A) 的左孩子。

下圖中的兩棵樹都不平衡，通過以 A 爲根左右旋轉得到了再平衡的右圖。

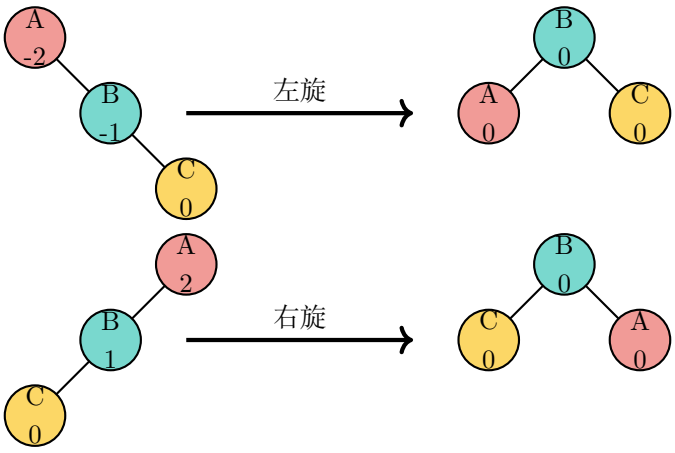
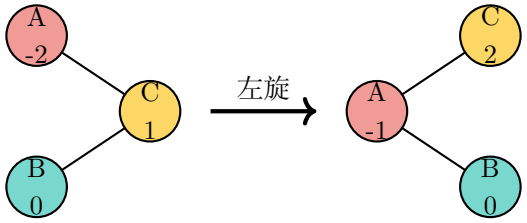


图 7.6: 不平衡樹及旋轉

知道了如何對子樹進行左右旋轉的規則，來試試將其運用到如下的這棵比較特殊的子樹並進行旋轉。



在左旋後，發現在另一方向還是失去了平衡。如果右旋糾正這種情況，則又回到最開始的情況。要糾正這個問題，必須使用新的旋轉規則，具體如下：

（1）如子樹需左旋使其平衡則首先檢查右子節點平衡因子，若右孩子是重的，則對右孩子做右旋轉，然後是再執行左旋轉。

（2）如子樹需右旋使其平衡則首先檢查左子節點平衡因子，若左孩子是重的，則對左孩子做左旋轉，然後是再執行右旋轉。

可見子樹旋轉過程在概念上相當容易理解，但是代碼實現非常複雜，因爲首先需要按照正確的順序移動節點，以便保留二叉查找樹的所有屬性。此外，還需要確保適當地更新所有的父指針。而 Rust 中移動和所有權機制的存在使得移動節點和更新指針關係頗爲複雜，十分容易出錯。瞭解了旋轉的概念和工作原理後可以看看下面實現的旋轉函數，包括右旋轉和左旋轉的代碼。其中 node 和 left、right 函數用於獲取節點和子樹。

```
1 // avl.rs
2
3 impl<T> AvlTree<T> where T : Ord {
4     // 獲取節點
```

```
5       fn node(&mut self) -> &mut AvlNode<T> {
6           match *self {
7               Null => panic!("Empty tree"),
8               Tree(ref mut n) => n,
9           }
10      }
11
12      // 獲取左右子樹
13      fn left_subtree(&mut self) -> &mut Self {
14          match *self {
15              Null => panic!("Empty tree"),
16              Tree(ref mut node) => &mut node.left,
17          }
18      }
19
20      fn right_subtree(&mut self) -> &mut Self {
21          match *self {
22              Null => panic!("Empty tree"),
23              Tree(ref mut node) => &mut node.right,
24          }
25      }
26
27      // 左右旋
28      fn rotate_left(&mut self) {
29          let mut v = replace(self, Null);
30          let mut right = replace(v.right_subtree(), Null);
31          let right_left = replace(right.left_subtree(), Null);
32          *v.right_subtree() = right_left;
33          *right.left_subtree() = v;
34          *self = right;
35      }
36
37      fn rotate_right(&mut self) {
38          let mut v = replace(self, Null);
39          let mut left = replace(v.left_subtree(), Null);
40          let left_right = replace(left.right_subtree(), Null);
41          *v.left() = left_right;
42          *left.right_subtree() = v;
```

```
43          *self = left;
44      }
45  }
```

通過旋轉操作，我們始終維持着樹的平衡。爲獲取樹的節點數、節點值、樹高，我們還需要實現 len、depth、value 等函數。

```
1  // avl.rs
2
3  impl<T> AvlTree<T> where T : Ord {
4      // 樹節點數是左右子樹節點數加根節點數，遞歸計算
5      fn len(&self) -> usize {
6          match *self {
7              Null => 0,
8              Tree(ref v) => 1 + v.left.len() + v.right.len(),
9          }
10     }
11
12     // 樹深度是左右子樹深度最大值 + 1, 遞歸計算
13     fn depth(&self) -> usize {
14         match *self {
15             Null => 0,
16             Tree(ref v) => max(v.left.depth(),
17                                v.right.depth()) + 1,
18         }
19     }
20
21     fn is_empty(&self) -> bool {
22         match *self {
23             Null => true,
24             _ => false,
25         }
26     }
27
28     // 數據查找
29     fn search(&self, val: &T) -> bool {
30         match *self {
31             Null => false,
32             Tree(ref v) => {
```

```
33                  match v.val.cmp(&val) {
34                      Equal => { true },
35                      Greater => {
36                          match &v.left {
37                              Null => false,
38                              _ => v.left.search(val),
39                          }
40                      },
41                      Less => {
42                          match &v.right {
43                              Null => false,
44                              _ => v.right.search(val),
45                          }
46                      },
47                  }
48              },
49          }
50      }
51  }
52
53  fn main() {
54      let mut avl = AvlTree::new();
55      for i in 0..10 {
56          let (_r1, _r2) = avl.insert(i);
57      }
58      println!("empty: {}", avl.is_empty());
59      println!("length: {}", avl.len());
60      println!("depth: {}", avl.depth());
61      println!("9 in avl: {}", avl.search(&9));
62  }
```

7.5.3 平衡二叉樹分析

AVL 平衡二叉樹相比二叉樹添加了左旋轉和右旋轉操作，這些旋轉操作用於維持二叉樹自身的平衡，這樣它的其他各種操作性能就能維持在比較優秀的水平，使得其最差性能都是 $O(log_2(n))$。

7.6 總結

在本章中，我們學習了樹這種高效的數據結構。樹使得我們能編寫許多有用和高效的算法，它廣泛用於存儲，網絡等領域。本章我們用樹執行了以下操作：

（1）解析和計算表達式。

（2）實現二叉樹，二叉查找樹，二叉平衡樹、紅黑樹

（3）實現堆及優先隊列。

在前面幾章中，我們已經學習了可用於實現映射關係（Map）的幾種抽象數據類型，包括有序表、散列表、二叉查找樹和平衡二叉查找樹，下面是它們的各種操作的最差性能。

表 7.5: 各種抽象數據結構的操作性能

操作	有序列表	哈希表	二叉查找樹	平衡二叉樹	紅黑樹
insert	$O(n)$	$O(1)$	$O(n)$	$O(log(n))$	$O(log(n))$
search	$O(log(n))$	$O(1)$	$O(n)$	$O(log(n))$	$O(log(n))$
delete	$O(n)$	$O(1)$	$O(n)$	$O(log(n))$	$O(log(n))$

Chapter 8

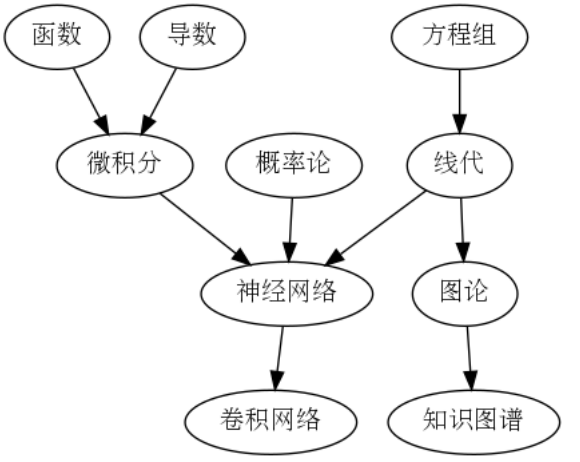
圖

8.1 本章目標

瞭解圖的概念及其使用方法
用 Rust 來實現圖數據結構
使用圖來解決各類現實問題

8.2 什麼是圖

上一章學習了樹，尤其是二叉樹。這一章來學習一種樹的更普遍的形式：圖。樹可以看成是簡化的圖，或者精心挑選的圖，因爲它的節點關係是有規律的。樹有根節點，但圖沒有，樹有方向，從上到下，圖的方向可有可無，樹中無環，但圖可以有環。圖涉及點、邊及點邊關係，可以用來表示真實世界中存在的事物，包括航班圖、網絡連接、菜譜、課程規劃等。下面就是一幅圖，它包含多個節點，存在多個連接。



8.2.1 圖定義

因爲圖 (graph) 是樹更普遍的形式，所以圖的定義是樹定義的延伸。

頂點：也就是樹中所說的節點，是圖的元素，它有一個名稱：鍵。一個頂點也可能有額外的信息：有效載荷。

邊：圖的另一個元素。邊連接兩個頂點，表明點之間的關係。邊可以是單向的或雙向的。如圖中邊都是單向的，稱該圖是有向圖。

權重：是邊的度量。用一個數值來表示從一個頂點到另一個頂點的距離、成本、時間、親密度。

利用這些概念，可以定義圖。圖用 G 來表示，G = (V, E)。圖中核心元素是點集合 V 和邊集合 E。每兩個不同點的組合 (v, w, q) 表示一條屬於 E 的邊 v-w，權重爲 q。下圖是帶權重的有向圖，其點集合爲 V = (V0, V1, V2, V2, V4, V5)，邊集合爲 E = ((V0,V1,5),(V1,V2,5),(V2,V3,5),(V3,V4,5),(V4,V0,5),(V0,V5,5),(V5,V4,5),(V3,V5,5),(V5,V2,1))。

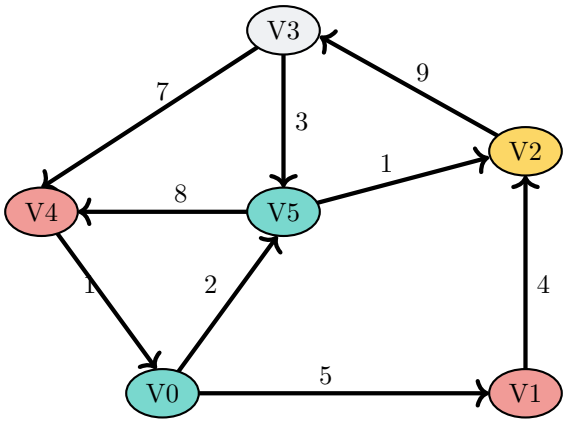


图 8.1: 圖

除了定義的點、邊、權重，還使用路徑來表示各個點的先後順序。路徑就是用頂點序列來表示點連接的前後順序，如 (v,w,x,y,z)。因爲圖中的點連接是任意的，所以可能出現有環圖，例如圖中 V5 -> V2 -> V3 -> V5。如果圖中沒有環，那麼這種圖稱爲無環圖或 DAG 圖。許多重要的問題都可以用 DAG 圖來表示。

8.3 圖的存儲形式

計算機內存是線性的，圖是非線性的，所以一定有要某種方法來保存圖到線性的存儲設備中。最常用的保存方法有兩種，一種是鄰接矩陣，一種是鄰接表。

鄰接矩陣採用二維矩陣存儲圖的節點和邊及權重，對於 N 個點的圖來說，需要 N^2 個空間。鄰接表類似哈希表，通過對每個頂點開一條鏈來保存所有與之相關的點和邊，其存儲空間根據邊的連接而定，一般情況下遠遠小於 N^2。

通過上述分析可以知道，計算機應該使用的是鄰接表來存儲圖。

8.3.1 鄰接矩陣

保存圖最簡單的方法就是用二維矩陣。在矩陣中，每行和每列表示圖中的頂點。存儲在行 v 和列 w 的交叉點處的值表示是頂點 v 到頂點 w 的邊存在且權重爲該值。當兩個頂點通過邊連接時，我們說它們是相鄰的。下圖就是鄰接矩陣，存儲的是圖（8.1）的點和邊。

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

图 8.2: 鄰接矩陣

鄰接矩陣簡單、直觀，對於小圖，容易看出節點連接關係。觀察矩陣可以發現大多數單元是空的，矩陣很稀疏。如果有 N 個頂點，那麼矩陣所需的存儲爲 N^2，非常浪費內存。

8.3.2 鄰接表

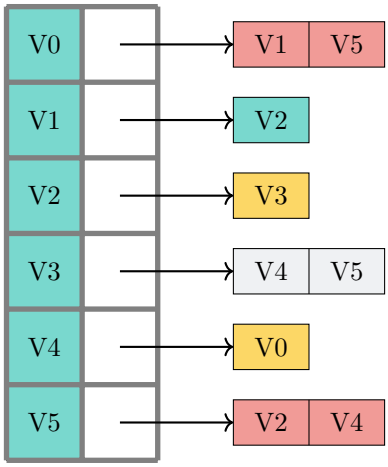


图 8.3: 鄰接表

實現圖高效保存的方法是使用鄰接表，如上圖。在鄰接表中，使用數組來保存所有頂點的主列表，然後圖中的每個主頂點維護一個連接到其他頂點的鏈表。這樣，通過訪問各個頂點的鏈接表，就能知道它鏈接多少點，這類似 HashMap 解決衝突時用的拉鏈法。

採用類似 HashMap 的結構來保存這些鏈接的點，主要是因爲邊帶有權重，而數組只能保存頂點。用鄰接表實現的圖是緊湊的，沒有內存浪費，這種結構保存也非常方便。從這裏也看出了，基礎數據結構的重要性，在這裏使用到了前面章節實現過的 HashMap 。

8.4 圖的抽象數據類型

有了圖的基本定義，下面定義圖的抽象數據結構。圖中核心的元素是點和邊，所以操作是圍繞點和邊來開展的。

new() 創建一個空圖，不需要參數，返回空圖。

add_vertex(v) 添加一個頂點，需要參數 v，無返回內容。

add_edge(fv,tv,w) 添加帶權重的有向邊，需要起點 fv 和終點 tv 及權重，無返回值。

get_vertex(vk) 在圖中找到鍵爲 vk 的點，需要參數 vk，發揮點。

get_vertices() 返回圖中所有頂點的列表，不需要參數。

vert_nums() 返回圖中頂點數，不需要參數。

edge_nums() 返回圖中邊數，不需要參數。

contains(vk) 判斷點是否在圖中，需要參數 vk，返回布爾值。

is_empty() 判斷圖是否爲空，不需要參數，返回布爾值。

假設 g 是新創建的空圖，下表展示了圖各種操作後的結果。[] 裝點，() 表示邊，其中前兩個值是點，第三個值是邊權重，如 (1,5,2) 表示點 1 和點 5 間邊權重爲 2。

表 8.1: 圖操作及其結果

圖操作	圖當前值	操作返回值
g.is_empty()	[]	true
g.add_vertex(1)	[1]	
g.add_vertex(5)	[1,5]	
g.add_edge(1,5,2)	[1,5,(1,5,2)]	
g.get_vertex(5)	[1,5,(1,5,2)]	5
g.get_vertex(4)	[1,5,(1,5,2)]	None
g.edge_nums()	[1,5,(1,5,2)]	1
g.vert_nums()	[1,5,(1,5,2)]	2
g.contains(1)	[1,5,(1,5,2)]	true
g.get_verteces()	[1,5,(1,5,2)]	[1,5]
g.add_vertex(7)	[1,5,7,(1,5,2)]	
g.add_vertex(9)	[1,5,7,9,(1,5,2)]	
g.add_edge(7,9,8)	[1,5,7,9,(1,5,2),(7,9,8)]	

8.5 圖的實現

首先來實現基於鄰接矩陣的圖，然後再實現基於鄰接表的圖。根據圖的定義和抽象數據類型，我們需要實現點（Vertex）和邊（Edge），然後用矩陣來存儲邊關係。Rust 中二維的 Vec 可以用來構造矩陣。

```rust
// graph_matrix.rs

// 點定義
#[derive(Debug)]
struct Vertex<'a> {
    id: usize,
    name: &'a str,
}

impl Vertex<'_> {
    fn new(id: usize, name: &'static str) -> Self {
        Self { id, name }
    }
}
```

邊只需要用布爾值表示是否存在即可，因爲邊是關係，不需要構造一個實體出來。

```rust
// graph_matrix.rs

// 邊定義
#[derive(Debug, Clone)]
struct Edge {
    edge: bool, // 表示是否有邊，不需要構造一個邊實體
}

impl Edge {
    fn new() -> Self {
        Self { edge: false }
    }

    fn set_edge() -> Self {
        Edge { edge: true }
    }
}
```

圖（Graph）中實現邊關係，並保存在二維的 Vec 中。

```
// graph_matrix.rs

// 圖定義
#[derive(Debug)]
struct Graph {
    nodes: usize,
    graph: Vec<Vec<Edge>>, // 每個點的邊放一個 vec
}

impl Graph {
    fn new(nodes: usize) -> Self {
        Self {
            nodes,
            graph: vec![vec![Edge::new(); nodes]; nodes],
        }
    }

    fn len(&self) -> usize {
        self.nodes
    }

    fn is_empty(&self) -> bool {
        0 == self.nodes
    }

    // 添加邊，設置邊屬性爲 true
    fn add_edge(&mut self, n1: &Vertex, n2: &Vertex) {
        if n1.id < self.nodes && n2.id < self.nodes {
           self.graph[n1.id][n2.id] = Edge::set_edge();
        } else {
            panic!("error");
        }
    }
}

fn main() {
    let mut g = Graph::new(4);
```

```
38      let n1 = Vertex::new(0, "n1");
39      let n2 = Vertex::new(1, "n2");
40      let n3 = Vertex::new(2, "n3");
41      let n4 = Vertex::new(3, "n4");
42
43      g.add_edge(&n1,&n2); g.add_edge(&n1,&n3);
44      g.add_edge(&n2,&n3); g.add_edge(&n2,&n4);
45      g.add_edge(&n3,&n4); g.add_edge(&n3,&n1);
46
47      println!("{:#?}", g);
48      println!("graph empth: {}", g.is_empty());
49      println!("graph nodes: {}", g.len());
50  }
```

下面再用 Rust 的 HashMap 來實現鄰接表圖。因爲點是核心元素，邊是點的關係，所以 Graph 還是將創建表示點元素的數據結構 Vertex。對 Vertex，需要的操作有新建點、獲取點自身的值、添加鄰接點、獲取所有鄰接點、獲取到鄰接點的權重。connects 變量用於保存所有鄰接點。

```
1   // graph_adjlist.rs
2
3   use std::hash::Hash;
4   use std::collections::HashMap;
5
6   // 點定義
7   #[derive(Debug, Clone)]
8   struct Vertex<T> {
9       key: T,
10      connects: Vec<(T, i32)>, // 鄰點集合
11  }
12
13  impl<T: Clone + PartialEq> Vertex<T> {
14      fn new(key: T) -> Self {
15          Self { key: key, connects: Vec::new() }
16      }
17
18      // 判斷與當前點是否相鄰
19      fn adjacent_key(&self, key: &T) -> bool {
20          for (nbr, _wt) in self.connects.iter() {
```

```
21              if nbr == key { return true; }
22          }
23
24          false
25      }
26
27      fn add_neighbor(&mut self, nbr: T, wt: i32) {
28          self.connects.push((nbr, wt));
29      }
30
31      // 獲取相鄰的點集合
32      fn get_connects(&self) -> Vec<&T> {
33          let mut connects = Vec::new();
34          for (nbr, _wt) in self.connects.iter() {
35              connects.push(nbr);
36          }
37
38          connects
39      }
40
41      // 返回到鄰點的邊權重
42      fn get_nbr_weight(&self, key: &T) -> &i32 {
43          for (nbr, wt) in self.connects.iter() {
44              if nbr == key {
45                  return wt;
46              }
47          }
48
49          &0
50      }
51  }
```

Graph 是實現的圖數據結構，其中包含將頂點名稱映射到頂點對象的 HashMap。

```
1  // graph_adjlist.rs
2
3  // 圖定義
4  #[derive(Debug, Clone)]
5  struct Graph <T> {
```

```
6       vertnums: u32, // 點數
7       edgenums: u32, // 邊數
8       vertices: HashMap<T, Vertex<T>>, // 點集合
9   }
10
11  impl<T: Hash + Eq + PartialEq + Clone> Graph<T> {
12      fn new() -> Self {
13          Self {
14              vertnums: 0,
15              edgenums: 0,
16              vertices: HashMap::<T, Vertex<T>>::new(),
17          }
18      }
19
20      fn is_empty(&self) -> bool {
21          0 == self.vertnums
22      }
23
24      fn vertex_num(&self) -> u32 {
25          self.vertnums
26      }
27
28      fn edge_num(&self) -> u32 {
29          self.edgenums
30      }
31
32      fn contains(&self, key: &T) -> bool {
33          for (nbr, _vertex) in self.vertices.iter() {
34              if nbr == key {
35                  return true;
36              }
37          }
38
39          false
40      }
41
42      fn add_vertex(&mut self, key: &T) -> Option<Vertex<T>> {
43          let vertex = Vertex::new(key.clone());
```

```
44          self.vertnums += 1;
45          self.vertices.insert(key.clone(), vertex)
46      }
47
48      fn get_vertex(&self, key: &T) -> Option<&Vertex<T>> {
49          if let Some(vertex) = self.vertices.get(key) {
50              Some(&vertex)
51          } else {
52              None
53          }
54      }
55
56      // 獲取所有節點的 key
57      fn vertex_keys(&self) -> Vec<T> {
58          let mut keys = Vec::new();
59          for key in self.vertices.keys() {
60              keys.push(key.clone());
61          }
62
63          keys
64      }
65
66      // 剔除點 (同時要剔除邊)
67      fn remove_vertex(&mut self, key: &T) -> Option<Vertex<T>> {
68          let old_vertex = self.vertices.remove(key);
69          self.vertnums -= 1;
70
71          // 剔除從當前點出發的邊
72          self.edgenums -= old_vertex.clone()
73                                      .unwrap()
74                                      .get_connects()
75                                      .len() as u32;
76
77          // 剔除到當前點的邊
78          for vertex in self.vertex_keys() {
79              if let Some(vt) = self.vertices.get_mut(&vertex) {
80                  if vt.adjacent_key(key) {
81                      vt.connects.retain(|(k, _)| k != key);
```

```
                        self.edgenums -= 1;
                    }
                }
            }

            old_vertex
        }

        fn add_edge(&mut self, from: &T, to: &T, wt: i32)  {
            // 若點不存在要先添加點
            if !self.contains(from) {
                let _fvert = self.add_vertex(from);
            }

            if !self.contains(to) {
                let _tvert = self.add_vertex(to);
            }

            // 添加邊
            self.edgenums += 1;
            self.vertices.get_mut(from)
                         .unwrap()
                         .add_neighbor(to.clone(), wt);
        }

        // 判斷兩個點是否相鄰
        fn adjacent(&self, from: &T, to: &T) -> bool {
            self.vertices.get(from).unwrap().adjacent_key(to)
        }
    }
```

使用 Graph 可以創建頂點 V0-V5，再據此創建邊，HashMap 保存了整個圖的點和邊。

```
// graph_adjlist.rs

fn main() {
    let mut g = Graph::new();

    for i in 0..6 { g.add_vertex(&i); }
```

```
7       println!("graph empty: {}", g.is_empty());
8
9       let vertices = g.vertex_keys();
10      for vertex in vertices { println!("Vertex: {:#?}", vertex); }
11
12      g.add_edge(&0,&1,5); g.add_edge(&0,&5,2);
13      g.add_edge(&1,&2,4); g.add_edge(&2,&3,9);
14      g.add_edge(&3,&4,7); g.add_edge(&3,&5,3);
15      g.add_edge(&4,&0,1); g.add_edge(&4,&4,8);
16      println!("vert nums: {}", g.vertex_num());
17      println!("edge nums: {}", g.edge_num());
18      println!("contains 0: {}", g.contains(&0));
19
20      let vertex = g.get_vertex(&0).unwrap();
21      println!("key: {}, to nbr 1 weight: {}",
22                vertex.key, vertex.get_nbr_weight(&1));
23
24      let keys = vertex.get_connects();
25      for nbr in keys { println!("nighbor: {nbr}"); }
26
27      for (nbr, wt) in vertex.connects.iter() {
28          println!("0 nighbor: {nbr}, weight: {wt}");
29      }
30
31      let res = g.adjacent(&0, &1);
32      println!("0 adjacent to 1: {res}");
33      let res = g.adjacent(&3, &2);
34      println!("3 adjacent to 2: {res}");
35
36      let rm = g.remove_vertex(&0).unwrap();
37      println!("remove vertex: {}", rm.key);
38      println!("left vert nums: {}", g.vertex_num());
39      println!("left edge nums: {}", g.edge_num());
40      println!("contains 0: {}", g.contains(&0));
41  }
```

8.5.1 圖解決字梯問題

有了圖數據結構，就可以解決些實際問題了。一個問題是字梯難題，指將某個單詞轉換成另一個單詞。比如將 FOOL 轉換爲單詞 SAGE。在字梯中你通過逐個改變字母而使單詞發生變化，但新的單詞必須是存在的而不是任意單詞。這個遊戲是《愛麗絲夢遊仙境》的作者劉易斯於 1878 年發明的。下面的單詞序列顯示了對上述問題的多種解決方案。

```
 1  a      b     c      d      e      f      g
 2  ---------------------------------------
 3  FOOL   FOOL  FOOL   FOOL   FOOL   FOOL   FOOL
 4  POOL   FOIL  FOIL   COOL   COOL   FOUL   FOUL
 5  POLL   FAIL  FAIL   POOL   POOL   FOIL   FOIL
 6  POLE   FALL  FALL   POLL   POLL   FALL   FALL
 7  PALE   PALL  PALL   POLE   PALL   FALL   FALL
 8  SALE   PALE  PALE   PALE   PALE   PALL   PALL
 9  SAGE   PAGE  SALE   SALE   SALE   POLL   POLL
10        SAGE  SAGE   SAGE   SAGE   PALE   PALE
11                                    PAGE   SALE
12                                    SAGE   SAGE
```

可見轉換的路徑有多條。我們的目標是，通過使用圖算法來找出從起始單詞轉換爲結束單詞的最小轉換次數，如上圖中 a 列。利用圖首先將單詞轉換爲頂點，然後將這些能前後變化得到的單詞鏈接起來。最後使用圖搜索算法來搜索最短路徑。如果兩個詞只有一個字母不同，表明它們可以相互轉換，則就創建這兩個單詞的有向邊，結果如下圖。

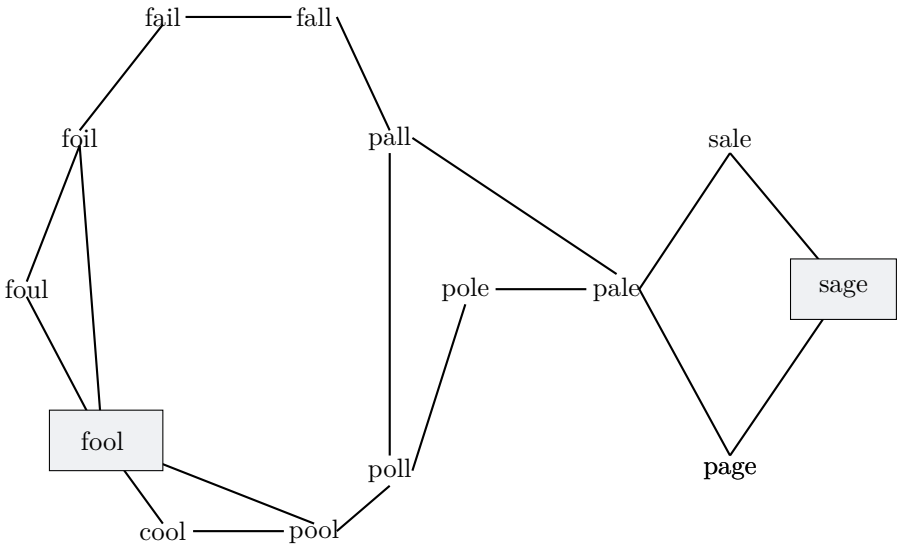


图 8.4: 字梯圖

可以使用多種不同方法來創建這個問題的圖模型。假設有一個長度相同的單詞列表，首先可以在圖中爲列表中的每個單詞創建一個頂點。爲了弄清楚如何連接單詞，必須比較列表中的每個單詞。比較時看有多少字母是不同的，如果所討論的兩個單詞只有一個字母不同，則可以在圖中創建它們之間的邊。對於小的單詞列表，這種方法可以正常工作。然而如果使用大學四級單詞表，假設有 3000 個單詞，那麼上述比較是 $O(n^2)$，需要比較九百萬次。如果是六級詞🄵表，假設 5000 單詞，則比較次數達到二千五百萬。顯然相互間能轉換的單詞不過幾十個，但每次比較卻要幾千次，非常低效。

然而，要是換個思路，只看單詞的字母位，對每個位置搜尋類似單詞並放在一起，那麼最終這些單詞都能和該單詞連接。比如 SOPE 去掉第一個字母，剩下 _OPE，那麼凡是結尾是 OPE 的四個字母的單詞都要和 SOPE 鏈接，將他們收集起來放到一個集合裏面。接着是 S_PE，凡是符合這種模式的單詞也收集起來。最終，符合模式的單詞都收集了，如下圖。

```
1  _OPE P_PE  PO_E  POP_
2  POPE POPE  POPE  POPE
3  ROPE PIPE  POLE  POPS
4  NOPE PAPE  PORE
5  HOPE       POSE
6  LOPE       POKE
7  COPE
```

可以使用 HashMap 來實現這種方案。上述集合保存了相同模式的單詞，以這種模式作爲 HashMap 的鍵，然後存儲類似的單詞集合。一旦建立了單詞集合，就可以創建圖。通過爲 HashMap 中的每個單詞創建一個頂點來開始圖，然後與字典中相同鍵下找到的所有頂點間創建邊。實現了圖後，就可以完成字梯搜索任務。

```
1  // word_lader.rs
2
3  fn build_word_graph(words: &[String]) -> Graph {
4      let mut d: HashMap<String, Vec<String>> = HashMap::new();
5
6      for word in words {
7          for i in 0..word.len() {
8              let bucket = (word[..i].to_string() + "_")
9                           + &word[i+1..];
10
11             let wd = word.to_string();
12             if d.contains_key(&bucket) {
13                 d.get_mut(&bucket).unwra().push(wd);
14             } else {
15                 d.insert(bucket, vec![wd]);
```

```
16              }
17          }
18      }
19
20      let mut g = Graph::new();
21      for bucket in d.keys() {
22          for wd1 in &d[bucket] {
23              for wd2  in &d[bucket] {
24                  if wd1 != wd2 {
25                      g.add_edge(wd1,wd2, 1);
26                  }
27              }
28          }
29      }
30
31      g
32  }
33
34  fn world_lader(mut g: Graph<String>, mut start: Vertex<String>,
35                 end: Vertex<String>, len: usize) -> uu32 {
36      start.set_distance(0);
37      start.set_pred(None);
38
39      let mut vertex_queue = Queue::new(len);
40      let _r = vertex_queue.enqueue(start);
41
42      while vertex_queue.size() > 0 {
43          let mut currv = vertex_queue.dequeue.unwrap();
44          for nbr in currv.get_connects() {
45              let nbv = g.vertices.get_mut(nbr).unwrap();
46              if 0 == nbv.color {
47                  nbv.set_color(1);
48                  nbv.set_distance(currv.dist + 1);
49                  nbv.set_pred(Some(currv.key.clone()));
50                  let v = g.vertices.get(nbr).unwrap().clone();
51                  let _r = vertex_queue.enqueue(v);
52              }
53          }
```

```
54            currv.set_color(2);
55        }
56
57        g.vertices.get(&end.key).unwrap().dist
58    }
```

8.6 廣度優先搜索

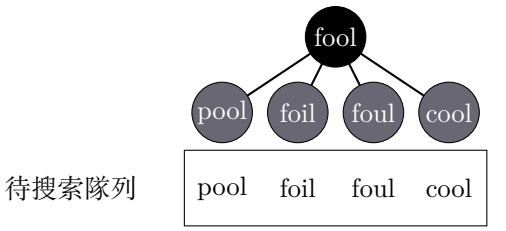
有了單詞構建的圖，接下來是如何查找到最短路徑。不像線性數據結構的查找，圖非線性，所以不能用傳統的二分和線性查找算法來搜索最短路徑。

圖及研究圖的專門理論-圖論，都是非常複雜的。對圖的研究也產生了各種優秀算法，比如用於搜索的算法有深度優先搜索和廣度優先搜索。對於字梯問題，可以採用圖的廣度優先搜索來查找最短路徑。注意，樹是特殊的圖，所以適合圖的深度優先搜索和廣度優先搜索也可用於在樹中搜索。深度和廣度優先搜索是搜索圖的最簡單算法，也是衆多其他重要圖算法的基礎。

給定圖 G 和起始頂點 s，廣度優先搜索通過探索圖中的邊以找到 G 中的所有頂點，其中存在從 s 開始的路徑。通過廣度優先搜索，它找到和 s 相距爲 1 的所有頂點，然後找到距離爲 2 的所有頂點。

可以將這個過程看成是逐層搜索，先將直接聯繫的一層頂點放到隊列裏逐個搜索。可以用三種顏色來標誌頂點的狀態，初始時點均爲白色，搜索時將與當前搜索點相連的點都置爲灰色，逐個查看灰色點，一旦確定該點及連接點都不是目標值，則將當前點置爲黑色，繼續搜索灰色點連接的白色點，重複搜索過程。這似乎和某些語言的垃圾收集機制有些類似，比如 Go 的三色垃圾收集法。

爲處理灰色點，可用一個隊列來保存灰色點，然後出隊灰色點並比較。爲對點顏色進行設置和修改，需要對 Vertex 擴展，添加表示顏色的參數，與起始點的距離，前導節點。



8.6.1 實現廣度優先搜索

BFS 從起始頂點 s 開始，將顏色設置爲灰色，表明它正在被搜索。另外兩個值，即距離和前導，對於起始頂點分別初始化爲 0 和 None，最後將其放到一個隊列中。下一步是開始系統地檢查隊列裏的頂點，通過迭代鄰接表來探索新節點。當檢查鄰接表上的節點時，要檢查其顏色，如果是白色的，表明該頂點是未被搜索的。在搜索頂點時會出現如下四種情況:

```
頂點是新的，未被搜索，則將其着色爲灰色。
頂點的前導被設置爲當前接點。
到頂點的距離設置爲當前距離加一。
頂點添加到隊尾。
```

下面來實現廣度優先搜索算法，首先定義一個圖，此處只爲實現廣度優先算法，所以圖可以簡化。

```
// bfs.rs

use std::rc::Rc;
use std::cell::RefCell;

// 因爲節點存在多個共享的鏈接，Box 不可共享，Rc 才可共享
// 又因爲 Rc 不可變，所以使用具有內部可變性的 RefCell 包裹
type Link = Option<Rc<RefCell<Node>>>;

// 節點
struct Node {
    data: usize,
    next: Link,
}

impl Node {
    fn new(data: usize) -> Self {
        Self {
            data: data,
            next: None
        }
    }
}

// 圖定義及實現
struct Graph {
    first: Link,
    last: Link,
}

impl Graph {
```

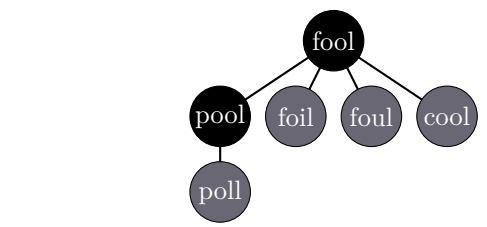
```
32      fn new() -> Self {
33          Self { first: None, last: None }
34      }
35
36      fn is_empty(&self) -> bool {
37          self.first.is_none()
38      }
39
40      fn get_first(&self) -> Link {
41          self.first.clone()
42      }
43
44      // 打印節點
45      fn print_node(&self) {
46          let mut curr = self.first.clone();
47          while let Some(val) = curr {
48              print!("[{}]", &val.borrow().data);
49              curr = val.borrow().next.clone();
50          }
51
52          print!("\n");
53      }
54
55      // 插入節點，RefCell 使用 borrow_mut 修改
56      fn insert(&mut self, data: usize) {
57          let node = Rc::new(RefCell::new(Node::new(data)));
58          if self.is_empty() {
59              self.first = Some(node.clone());
60              self.last = Some(node);
61          } else {
62              self.last.as_mut()
63                       .unwrap()
64                       .borrow_mut()
65                       .next = Some(node.clone());
66              self.last = Some(node);
67          }
68      }
69  }
```

下面是實現的廣度優先搜索算法。

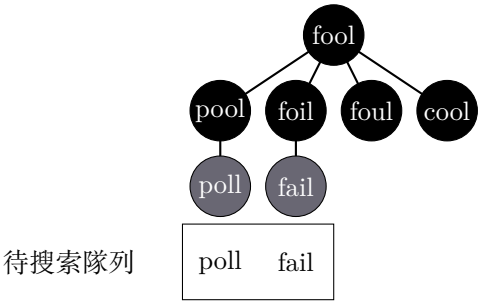
```
// bfs.rs

// 根據 data 構建圖
fn create_graph(data: [[usize;2];20]) -> Vec<(Graph, usize)> {
    let mut arr: Vec<(Graph, usize)> = Vec::new();
    for _ in 0..9 {
        arr.push((Graph::new(), 0));
    }

    for i in 1..9 {
        for j in 0..data.len() {
            if data[j][0] == i {
                arr[i].0.insert(data[j][1]);
            }
        }
        print!("[{i}]->");
        arr[i].0.print_node();
    }

    arr
}

fn bfs(graph: Vec<(Graph, usize)>) {
    let mut gp = graph;
    let mut nodes = Vec::new();

    gp[1].1 = 1;
    let mut curr = gp[1].0.get_first().clone();

    // 打印圖
    print!("{1}->");
    while let Some(val) = curr {
        nodes.push(val.borrow().data);
        curr = val.borrow().next.clone();
    }

    // 打印寬度優先圖
```

```
38      loop {
39          if 0 == nodes.len() {
40              break;
41          } else {
42              let data = nodes.remove(0);
43              if 0 == gp[data].1 {
44                  gp[data].1 = 1;
45                  print!("{data}->");
46                  let mut curr = gp[data].0.get_first().clone();
47                  while let Some(val) = curr {
48                      nodes.push(val.borrow().data);
49                      curr = val.borrow().next.clone();
50                  }
51              }
52          }
53      }
54
55      println!();
56  }
57
58  fn main() {
59      let data = [[1,2],[2,1],[1,3],[3,1],[2,4],[4,2],[2,5],
60                  [5,2],[3,6],[6,3],[3,7],[7,3],[4,5],[5,4],
61                  [6,7],[7,6],[5,8],[8,5],[6,8],[8,6]];
62      let gp = create_graph(data);
63      bfs(gp);
64  }
```

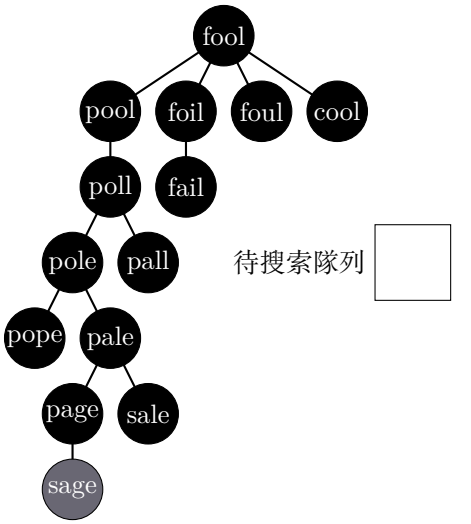
bfs 函數構造出如下圖的廣度優先搜索樹。開始取所有與 fool 相鄰的節點，並將它們添加到樹中。相鄰節點包括 pool，foil，foul，cool。然後這些節點會被添加到新節點的隊列以進行搜索。



在下一步驟中，bfs 從隊列前面刪除下一個節點 pool，並對其所有相鄰節點重複該過程。當 bfs 檢查節點 cool 時，發現它是灰色，說明有較短的路徑到 cool。在檢查 pool 時，又添加了新節點 poll。



隊列上的下一個頂點是 foil，可以添加到樹中的唯一新節點是 fail。當 bfs 繼續處理隊列時，接下來的兩個節點都不向隊列添加新點。



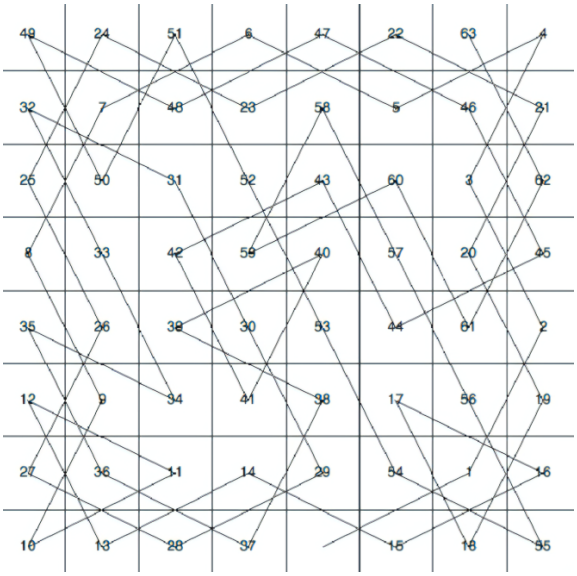
重複操作，直到完成圖搜索。這個圖看起來就像是一棵樹。

8.6.2 廣度優先搜索分析

bfs 對圖中的每個頂點 V 最多執行一次 while 循環。因爲頂點必須是白色，才能被檢查和添加到隊列，此操作過程性能爲 O(v)。嵌套在 while 內部的 for 循環對圖中的每個邊執行最多一次，因爲每個頂點最多被出隊一次，並且僅當節點 u 出隊時，才檢查從節點 u 到節點 v 的邊。所以 for 循環的性能爲 O(E)，總的性能爲 O(V+E)。

8.6.3 騎士之旅

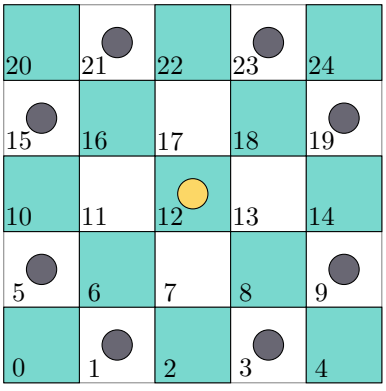
另一個可以用圖來解決的問題是騎士之旅。騎士之旅是將棋盤上的棋子當騎士玩，目的是找到一系列的動作，讓騎士訪問板上的每個格一次，這樣的序列被稱爲旅遊。騎士之旅難題多年來一直吸引着象棋玩家、數學家和計算機科學家。一個 8×8 的棋盤可能的遊覽次數上限爲 1.305×10^{35}，即有這麼多條不重複路徑滿足到訪問每個格子一次且不重複。

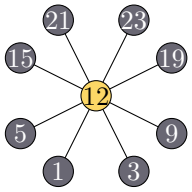


經過多年的研究，也發明了多種不同的算法來解決騎士旅遊問題，圖搜索是最容易理解的解決方案。圖算法需要兩個步驟來解決騎士之旅，一是如何表示騎士在棋盤上的動作，其次是查找長度爲 $rows \times columns - 1$，的路徑，-1 是因爲自身所佔格子不計入總數。

8.6.4 圖解決騎士之旅

爲將騎士旅遊問題表示爲圖，可以將棋盤上的每個正方形表示爲圖中的一個點，騎士的每次合法移動（馬走日）表示爲圖中的邊，如下圖，黃色處是騎士。





騎士能移動到的點

要構建一個 $n \times n$ 的完整圖，可使用如下函數。該函數在整個板上進行一次遍歷，在每個方塊上爲板上的位置創建一個移動列表，所有移動在圖中轉換爲邊。另一個輔助函數 pos_to_node_id 按照行和列將板上的位置轉換爲頂點。

```
// kinght_tour2.rs

fn kinght_graph(bdsize: u32) -> Graph {
    let mut g = Graph::new();

    for row in 0..bdsize {
        for col in 0..bdsize {
            let nodeid = pos_2_node_id(row,col,bdsize);
            let pos = gen_legal_move(row,col,bdsize);
            for e in pos {
                let nid = pos_2_node_id(e[0], e[1], bdsize);
                g.add_edge(nodeid, nid);
            }
        }
    }

    g
}

fn pos_2_nod_id(row: u32, col: u32, bdsize: u32) -> u32 {
    row * bdsize + col
}

fn gen_legal_move(x: u32, y: u32, bdsize) -> Vec<(u32, u32)> {
    let mut moves = vec![];

    // 馬移動是座標變化，共 8 個方向
    let move_offsets = [(-1,-2),(-1,2),(-2,-1),(-2,1),
                        (1,-2),(1,2),(2,-1),(2,1)];
    for offset in move_offsets {
        let newx = x + offset[0];
        let newy = y + offset[1];
        if legal_coord(newx, bdsize)
            && legal_coord(newy, bdsize) {
            moves.push((newx,newy));
```

```
36          }
37      }
38      moves
39  }
40
41  fn legal_coord(x: u32, bdsize: u32) -> boool {
42      if x >= 0 && x < bdsize {
43          true
44      } else {
45          false
46      }
47  }
```

解決騎士旅遊問題的搜索算法稱爲深度優先搜索。在前面討論過廣度優先搜索算法是儘可能廣的在同一層搜索頂點，而深度優先搜索則是儘可能深地探索樹的多層。可以通過設置多種不同的策略來利用深度優先搜索解決騎士之旅問題。第一種是通過明確禁止節點被訪問多次來解決騎士之旅，第二種實現則更通用，允許在構建樹時多次訪問節點。深度優先搜索算法在找到死角（沒有可移動的地方）時，它將回溯到上一個最深的點，繼續探索。

```
1  // depth: 走過的路徑長度， u: 起始點，path: 保存訪問過的點
2  fn kinght_tour(depth: u32, path: &mut Vec,
3                 u: &mut Vec, limit: u32) -> bool {
4      u.set_color("gray");
5      path.push(u);
6      let mut done = false;
7      if depth < limit {
8          let nbrs = u.get_connects();
9          let mut i = 0;
10         while i < nbrs.len() && !done {
11             if nbrs[i].get_color() == "white" {
12                 done = kinght_tour(depth+1,path,nbrs[i],limit);
13             }
14             i += 1;
15         }
16         if !done {
17             let _rm = path.pop();
18             u.set_color("white");
19         }
20     } else {
```

```
21            done = true;
22        }
23
24        done
25    }
```

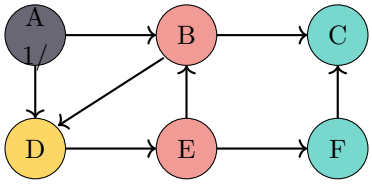
kinght_tour 是遞歸的深度優先搜索算法，若找到的路徑等於 64，則返回，否則找尋下一個頂點來探索。深度優先搜索還使用顏色來跟蹤圖中的哪些頂點已被訪問。未訪問的頂點是白色的，訪問的頂點是灰色的。如果已經探索了特定頂點的所有鄰居，並且尚未達到 64 的目標長度，表明已經到達死衚衕，此時必須回溯。當狀態爲 false 的 knightTour 返回時，會發生回溯。在廣度優先搜索中，我們使用了一個隊列來跟蹤下一個要訪問的頂點。由於深度優先搜索是遞歸的，所以有一個隱式的棧在幫助算法回溯。

8.7 深度優先搜索

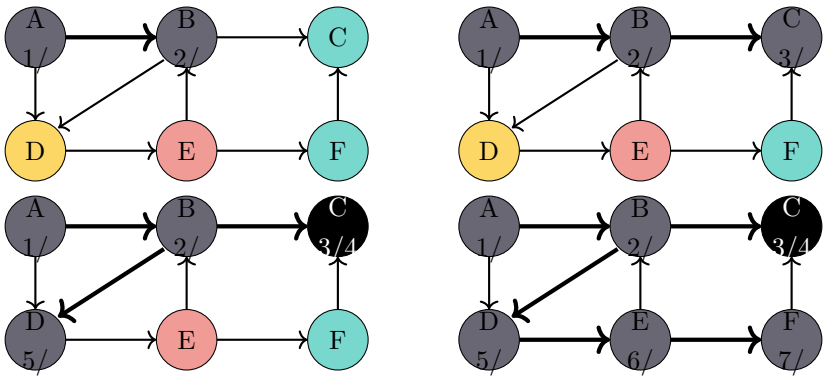
騎士之旅是深度優先搜索的特殊情況，其目的是創建一棵最深的樹。更一般的深度優先搜索實際上是對有多個分支的圖進行搜索。其目標是儘可能深地搜索，連接儘可能多的節點，並在必要時創建分支。

深度優先搜索可能創建多於一個樹。當深度優先搜索算法創建一組樹時，稱之爲深度優先森林。與廣度優先搜索一樣，深度優先搜索使用前導鏈接來構造樹。此外，深度優先搜索將在頂點類中使用兩個附加的變量。新變量是發現和結束時間。發現時間跟F首次遇到頂點之前的步驟數，結束時間是頂點着色爲黑色之前的步驟數。

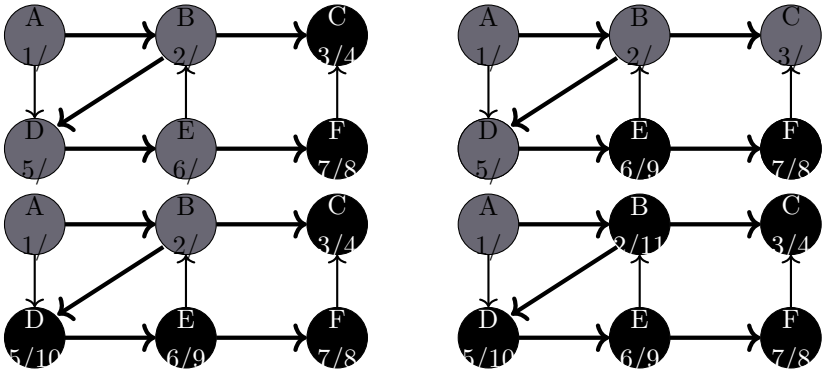
下圖展示了一個小的圖深度優先搜索算法。在這些圖中，虛線指示檢查的邊，在邊的另一端的節點已經被添加到深度優先樹。搜索從圖的頂點 A 開始。由於所有頂點在搜索開始時都是白色的，所以算法隨機開始訪問頂點 A。訪問頂點的第一步是將顏色設置爲灰色，這表示正在探索頂點，並且將發現時間設置爲 1，由於頂點 A 具有兩個相鄰的頂點 (B，D)，因此每個頂點也需要被訪問，我們就按字母順序來訪問相鄰頂點。



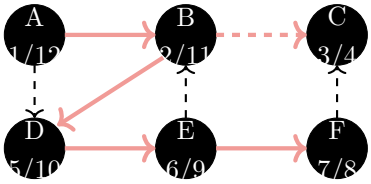
接下來訪問頂點 B，設置顏色爲灰色並且發現時間設置爲 2。頂點 B 也與兩個其他節點 (C，D) 相鄰，因此接下來將訪問節點 C。訪問頂點 C 使我們到了樹的一個分支末端，這意味着結束了對節點 C 的探索，因此可以將頂點着色爲黑色，並將結束時間設置爲 4。現在必須返回到頂點 B，繼續探索與 B 相鄰的節點。從 B 中探索的額外一個頂點是 D，接着訪問 D。頂點 D 引導我們到頂點 E。頂點 E 具有兩個相鄰的頂點 B 和 F 。由於 B 已經是灰色的，所以算法識別出它不應該訪問 B，繼續探索的下一個頂點是 F。



頂點 F 只有一個相鄰的頂點 C，但由於 C 是黑色的，沒有別的點可以探索，算法已經到達另一個分支的末尾。因此算法必須回溯並不斷將訪問過的點標及爲黑色。



最終，算法將回溯到最初的點，並退出搜索，得到如下圖所示訪問路徑。紅線是訪問過的路徑，最深路徑爲 A->B->D->E->F。



8.7.1 實現深度優先搜索

結合前面的內容及圖示，可以得到下面更通用深度優先搜索算法。

```
1 // dfs.rs
2 use std::rc::Rc;
3 use std::cell::RefCell;
```

```
4
5  // 鏈接
6  type Link = Option<Rc<RefCell<Node>>>;
7
8  // 節點
9  struct Node {
10     data: usize,
11     next: Link,
12 }
13
14 impl Node {
15     fn new(data: usize) -> Self {
16         Self { data: data, next: None }
17     }
18 }
19
20 // 圖
21 struct Graph {
22     first: Link,
23     last: Link,
24 }
25
26 impl Graph {
27     fn new() -> Self {
28         Self { first: None, last: None }
29     }
30
31     fn is_empty(&self) -> bool {
32         self.first.is_none()
33     }
34
35     fn get_first(&self) -> Link {
36         self.first.clone()
37     }
38
39     fn print_node(&self) {
40         let mut curr = self.first.clone();
41         while let Some(val) = curr {
```

```
42              print!("[{}]", &val.borrow().data);
43              curr = val.borrow().next.clone();
44          }
45          print!("\n");
46      }
47
48      // 插入數據
49      fn insert(&mut self, data: usize) {
50          let node = Rc::new(RefCell::new(Node::new(data)));
51          if self.is_empty() {
52              self.first = Some(node.clone());
53              self.last = Some(node);
54          } else {
55              self.last.as_mut()
56                       .unwrap()
57                       .borrow_mut()
58                       .next = Some(node.clone());
59              self.last = Some(node);
60          }
61      }
62  }
63
64  // 構建圖
65  fn create_graph(data: [[usize;2];20]) -> Vec<(Graph, usize)> {
66      let mut arr: Vec<(Graph, usize)> = Vec::new();
67      for _ in 0..9 {
68          arr.push((Graph::new(), 0));
69      }
70
71      for i in 1..9 {
72          for j in 0..data.len() {
73              if data[j][0] == i {
74                  arr[i].0.insert(data[j][1]);
75              }
76          }
77          print!("[{i}]->");
78          arr[i].0.print_node();
79      }
```

```
80
81      arr
82  }
83
84  fn dfs(graph: Vec<(Graph, usize)>) {
85      let mut gp = graph;
86      let mut nodes: Vec<usize> = Vec::new();
87      let mut temp: Vec<usize> = Vec::new();
88
89      gp[1].1 = 1;
90      let mut curr = gp[1].0.get_first().clone();
91
92      // 打印圖
93      print!("{1}->");
94      while let Some(val) = curr {
95          nodes.insert(0,val.borrow().data);
96          curr = val.borrow().next.clone();
97      }
98
99      // 打印深度優先圖
100     loop{
101         if 0 == nodes.len() {
102             break;
103         }else{
104             let data = nodes.pop().unwrap();
105             if 0 == gp[data].1 {
106                 gp[data].1 = 1;
107                 print!("{data}->");
108
109                 // 節點加入 temp
110                 let mut curr = gp[data].0.get_first().clone();
111                 while let Some(val) = curr {
112                     temp.push(val.borrow().data);
113                     curr = val.borrow().next.clone();
114                 }
115
116                 while !temp.is_empty(){
117                     nodes.push(temp.pop().unwrap());
```

```
                    }
                }
            }
        }

        println!("");
    }

    fn main() {
        let data = [[1,2],[2,1],[1,3],[3,1],[2,4],[4,2],[2,5],
                    [5,2],[3,6],[6,3],[3,7],[7,3],[4,5],[5,4],
                    [6,7],[7,6],[5,8],[8,5],[6,8],[8,6]];

        let gp = create_graph(data);
        dfs(gp);
    }
```

dfs 迭代所有白點，並使用 dfsvisit 來訪問節點。dfsvisit 從一頂點開始，並儘可能深地探查所有相鄰白色頂點，dfsvisit 和 dfs 算法幾乎一樣，除了 for 循環最後一行，dfsvisit 遞歸調用自己進行深度搜索，而 bfs 負責控制回溯後要訪問的頂點。bfs 使用的的是隊列，因爲節點的訪問是按先後順序的，而 dfsvisit 使用棧來管理遞歸。通過這裏可以看到棧、隊列這些基礎的數據結構在解決大問題中的作用。

8.7.2 深度優先搜索分析

深度優先搜索 dfs 中的循環都在 O(V) 中運行，可以不計入 dfsvisit 中的時間，因爲它們對圖中的每個頂點執行一次。在 dfsvisit 中，對當前頂點的鄰接表中的每個邊執行一次循環。由於只有當頂點爲白色時，dfsvisit 才被遞歸調用，所以循環對圖中的每個邊執行最多一次，其複雜度爲 O(E)。綜上，深度優先搜索的總時間性能是 O(V+E)。

8.7.3 拓撲排序

圖的出現使得我們可以把許多現實世界的問題抽象成圖，並利用圖的性質和算法來解決現實世界的大問題。讓我們考慮做煎餅的問題，如圖（8.5）所示。菜譜很簡單：雞蛋 1 個，煎餅粉 1 杯，1 湯匙油和 3/4 杯牛奶。要製作煎餅，你必須加熱爐子，將所有的材料混合在一起，勺子攪拌。當開始冒泡，再把它們翻過來，直到底部變金黃色。在你吃煎餅之前，你可能還要熱一些糖漿。製作煎餅的困難是不知道先做那一個步驟。從圖中看，你可以先打開煎鍋，或混合原材料。爲了決定每個步驟的精確順序，可以用算法來將圖排序，這種排序稱爲拓撲排序。拓撲排序採用有向無環圖，並且產生所有頂點的線性排序。定向非循環圖可以用來指示事件的優先級，設定軟件項目計劃，產生數據庫查詢的優先圖等。

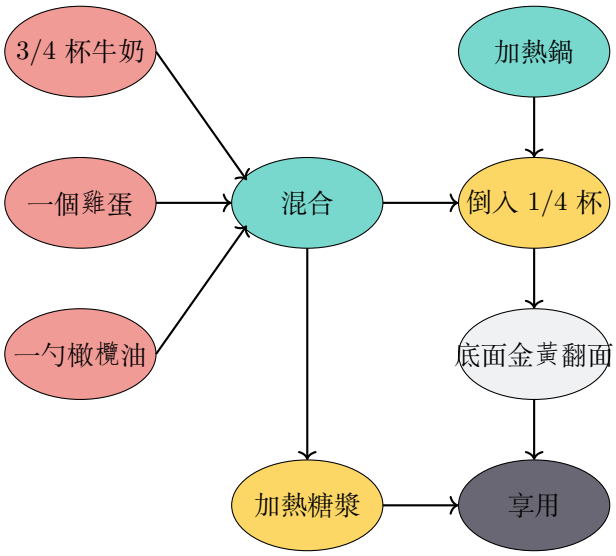
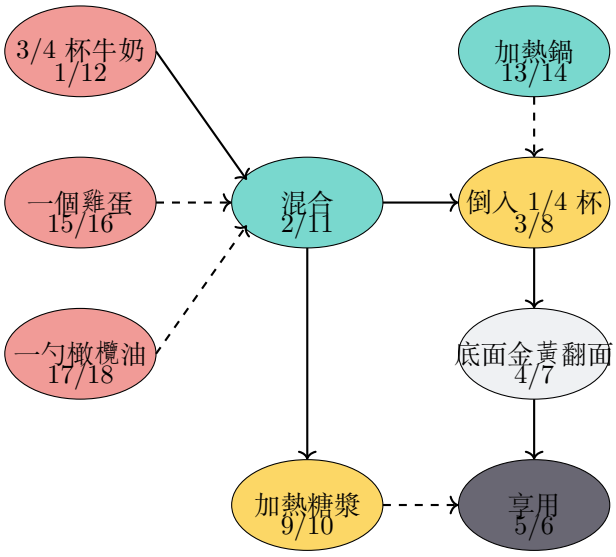


图 8.5: 製作煎餅

拓撲排序是深度優先搜索的一種改造，拓撲排序的算法如下:

```
1 對圖 g 調用 dfs(), 用深度優先搜索的來計算每個頂點的結束時間。
2 以結束時間的遞減順序將頂點存儲在列表中。
3 返回有序列表作爲拓撲排序的結果。
```

拓撲排序可以將圖轉化成具有線性關係的圖，如將上述菜譜將轉化成如下深度優先森林及步驟關係圖。



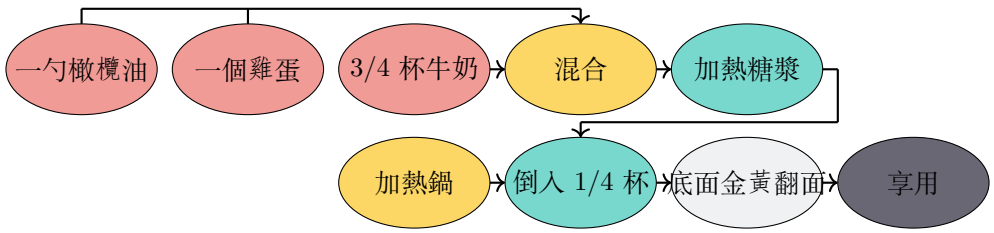


图 8.6: 製作煎餅的深度優先森林及步驟關係圖

8.8 強連通分量

一些非常大的圖，比如網頁間產生的鏈接，也是圖。百度，谷歌等搜索引擎存儲的海量鏈接就是龐大的有向圖。爲將萬維網變換爲圖，可將頁面視爲頂點，並將頁面上的超鏈接作爲頂點間連接的邊。下圖是 Google 主站點鏈接的其他站點，整個 Internet 絡都在圖中。

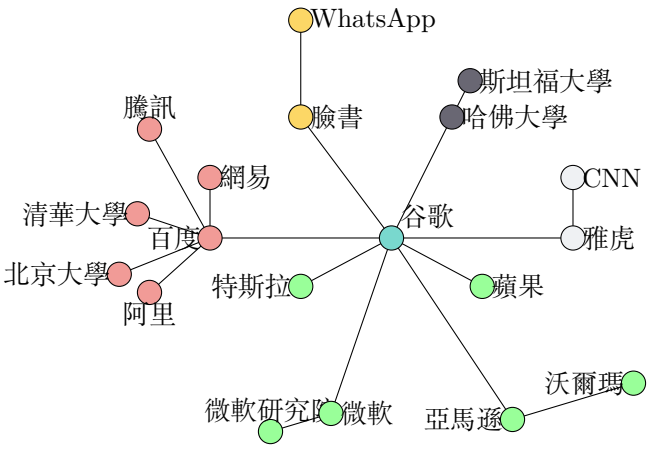


图 8.7: Internet 連接圖

這類圖有個明顯的特點：某些節點鏈接特別多。如谷歌幾乎和世界上任何網站連接起來，而有的節點卻只有一個鏈接。也就是說，節點分區域聚集，高度互連。聚集的點區域稱爲連通區域，節點是強連通的。強連通是指在某個區域內，從任意節點可在有限路徑內到達另一節點。爲找到哪些節點組成強連通區域，可以用連通分量算法來計算。

強連通分量 C ∈ G，其中每個點 v, w ∈ G，且點 v 和 w 相互可達，圖（8.8）是強連通分量圖及其簡化圖，左側三種顏色區域強連通。確定了強連通分量，就可以將其看成一個點以簡化圖。爲獲取強連通分量，需要對圖調用深度優先搜索。首先將圖連接反轉 G^T，生成的圖還是強連通的，且連通分量不變，如圖（8.9）。計算強連通分量的算法步驟如下：

```
1 調用 dfs 爲 G 計算每個點的結束時間
2 計算 G^T，對圖 G^T 調用 dfs，計算每個點結束時間
3 輸出每個森林中每棵樹頂點的標記組件
```

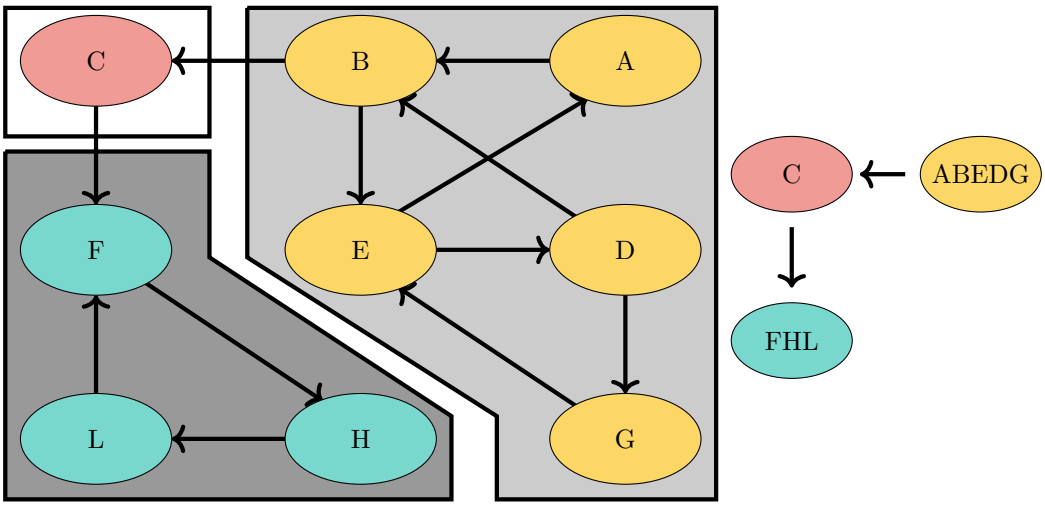


图 8.8: 強連同分量圖（左），及其簡化圖（右）

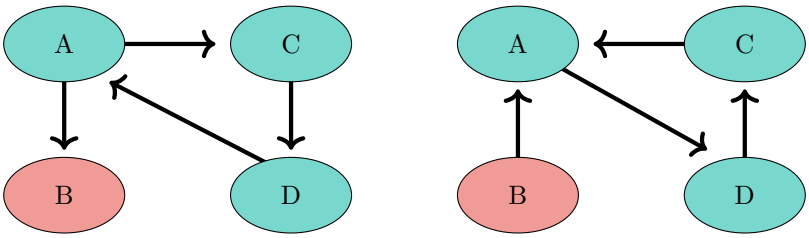
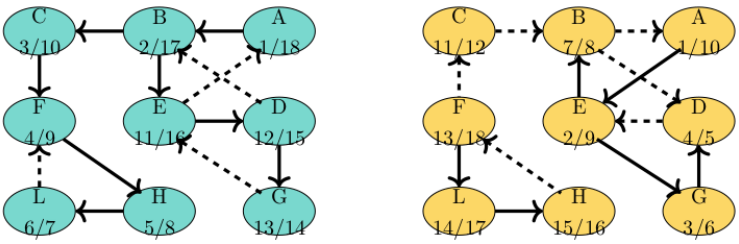
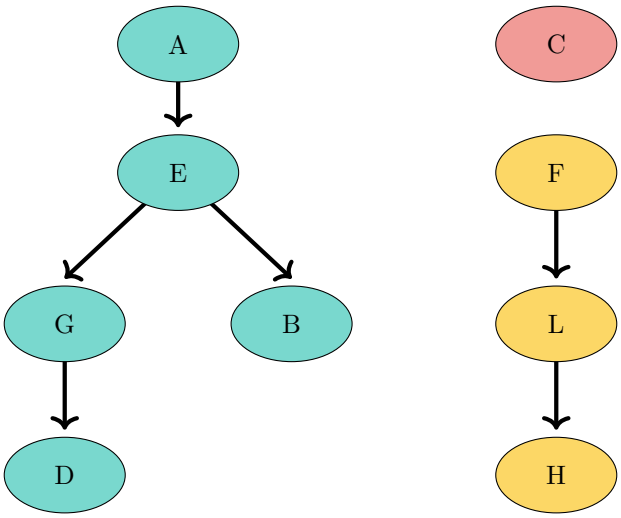


图 8.9: 圖及其反轉圖

通過對連通圖及其轉置圖使用深度優先搜索可得到如下連通圖。實線表示連通區域內的搜索路徑，虛線表示可連通路徑，數字表示搜索的起止時間。無論是圖還是其反轉圖，都能通過該算法准確得到連通區域。

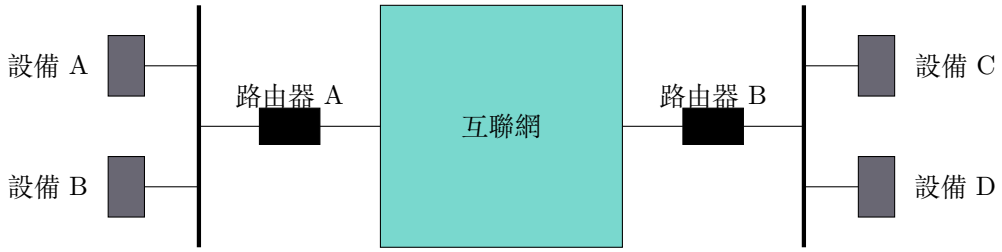


通過對連通圖使用強連通分量算法可以得到三棵樹，其實就是三個連通分量。在下圖中可以看到連通區域是如何簡化的。其中 C 是一個獨立區域，雖然只有它自己一個節點。F，L，H 三點是一個連通區域，A，E，G，D，B 也是一個連通區域。通過強連通分量算法得到的三棵樹就是三個連通區域，該算法將問題由節點層面轉到了連通區域層面，極大地降低了問題難度，便於後續分析處理。



8.9 最短路徑問題

上網看短視頻，發郵件，或從校園外登陸錄實驗室計算機時，信息是由網絡傳輸的。研究信息如何通過互聯網從一臺計算機流向另一臺計算機是計算機網絡中的一個大課題。



上圖展示了 Internet 上的通信如何工作。使用瀏覽器從服務器請求網頁時，請求必須通過局域網傳輸，並通過路由器傳輸到 Internet 上。該請求通過因特網傳播，並最終到達服務器所在的局域網路由器，服務器返回的網頁再通過相同的路由器回到您的瀏覽器。如果你的計算機支持 tracepath 命令，可以用它來查看你的電腦到某個鏈接的路徑，比如如下追蹤到達 xxx.cn 網站經過了 13 個路由器，其中第一二個是自己所在網絡組的網關路由器。

```
1   1?: [LOCALHOST]                      pmtu 1500
2   1:  _gateway                         4.523 毫秒
3   1:  _gateway                         3.495 毫秒
4   2:  10.253.0.22                      2.981 毫秒
5   3:  無應答
6   4:  ???                              6.166 毫秒
7   5:  202.115.254.237                558.609 毫秒
8   6:  無應答
9   7:  無應答
```

```
10   8:  101.4.117.54                48.822 毫秒 asymm 16
11   9:  無應答
12  10:  101.4.112.37                48.171 毫秒 asymm 14
13  11:  無應答
14  12:  101.4.114.74                44.981 毫秒
15  13:  202.97.15.89                49.560 毫秒
```

互聯網上的每個路由器都連接到一個或多個路由器。因此，如果在一天的不同時間運行 tracepath，你看到的輸出不一定一樣。你很可能會看到信息在不同的時間不同，信息流經了不同的路由器。這是因爲路由器之間的連接存在成本，同時還取決於網絡流量情況。你可以將網絡鏈接看成帶有權重的圖，連接會根據網絡情況作調整。

我們的目標是找到具有最小總權重的路徑，用於沿着該路徑傳送消息。這個問題類似於廣度優先搜索解決的問題，這裏關心的是路徑的總權重，而不是路徑條數。應當注意，如果所有權重相等，則問題是相同的。

8.9.1 Dijkstra 算法

研究網絡圖最短路徑算法的前輩們提出了各種各樣的算法，其中 Dijkstra 算法是搜索圖中最短路徑的好算法。Dijkstra 算法是一種貪心迭代算法，它爲我們提供從一個特定起始節點到圖中所有其他節點的最短路徑，這有點類似於廣度優先搜索。

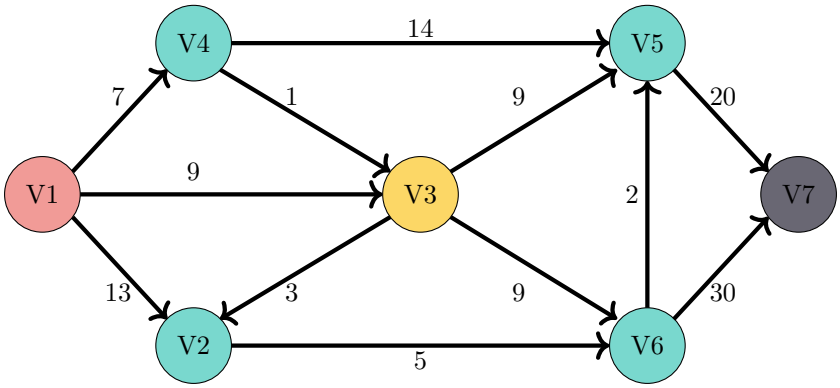


图 8.10: Dijkstra 圖示

如上圖，需要找到從 V1 到 V7 的最短路徑，通過一定時間的探索，讀者定能得出最短路徑是 [V1->V4->V3->V2->V6->V5->V7]，最短距離爲 38。如果通過算法來計算，就需要跟蹤並計算各個距離並求和。

爲跟蹤從起始節點到每個目標節點的總距離，將使用圖頂點中的 dist 實例變量。該實例變量將包含從開始到目標節點的路徑總權重。Dijkstra 算法對圖中的每個節點重複一次，在節點上迭代的順序由優先級隊列控制，而用於確定優先級隊列中對象順序的值便是 dist 值。首次創建節點時，dist 被設置爲非常大的數。理論上，將 dist 設置爲無窮大，但在實踐中，

只將它設置爲一個數字，大於任何真正的距離就可以了，比如光在一秒內行進的距離，也就是地球和月亮間的距離也是一個合理的值，因爲地球上沒有哪兩個點的距離會達到這麼大。

8.9.2 實現 Dijkstra 算法

Dijkstra 算法使用優先級隊列處理頂點，用於保存一個鍵值對元組，值作爲優先級別。

```
1  // dijkstra.rs
2
3  use std::cmp::Ordering;
4  use std::collections::{BinaryHeap, HashMap, HashSet};
5
6  // 點
7  #[derive(Debug, Copy, Clone, PartialEq, Eq, Hash)]
8  struct Vertex<'a> {
9      name: &'a str,
10 }
11
12 impl<'a> Vertex<'a> {
13     fn new(name: &'a str) -> Vertex<'a> {
14         Vertex { name }
15     }
16 }
17
18 // 訪問過的點
19 #[derive(Debug)]
20 struct Visit<V> {
21     vertex: V,
22     distance: usize, // 距離
23 }
24
25 // 爲 Visited 添加全序比較功能
26 impl<V> Ord for Visit<V> {
27     fn cmp(&self, other: &Self) -> Ordering {
28         other.distance.cmp(&self.distance)
29     }
30 }
31
32 impl<V> PartialOrd for Visit<V> {
33     fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
```

```
34          Some(self.cmp(other))
35      }
36  }
37
38  impl<V> Eq for Visit<V> {}
39
40  impl<V> PartialEq for Visit<V> {
41      fn eq(&self, other: &Self) -> bool {
42          self.distance.eq(&other.distance)
43      }
44  }
45
46  // 最短路徑算法
47  fn dijkstra<'a>(
48      start: Vertex<'a>,
49      adj_list: &HashMap<Vertex<'a>, Vec<(Vertex<'a>, usize)>>
50      ) -> HashMap<Vertex<'a>, usize> {
51      let mut distances = HashMap::new();   // 距離
52      let mut visited = HashSet::new();     // 已訪問過的點
53      let mut to_visit = BinaryHeap::new(); // 待訪問的點
54
55      // 設置起始點和起始距離
56      distances.insert(start, 0);
57      to_visit.push(Visit {
58          vertex: start,
59          distance: 0,
60      });
61
62      while let Some(Visit {vertex, distance}) = to_visit.pop() {
63          // 已訪問過該點，繼續下一個點
64          if !visited.insert(vertex) { continue; }
65
66          // 獲取鄰點
67          if let Some(neighbors) = adj_list.get(&vertex) {
68              for (neighbor, cost) in neighbors {
69                  let new_distance = distance + cost;
70                  let is_shorter = distances
71                      .get(&neighbor)
```

```
72                      .map_or(true, |&curr| new_distance < curr);
73
74                  // 若距離更近，則插入新距離和鄰點
75                  if is_shorter {
76                      distances.insert(*neighbor, new_distance);
77                      to_visit.push(Visit {
78                          vertex: *neighbor,
79                          distance: new_distance,
80                      });
81                  }
82              }
83          }
84      }
85
86      distances
87  }
88
89  fn main() {
90      let s = Vertex::new("s");
91      let t = Vertex::new("t");
92      let x = Vertex::new("x");
93      let y = Vertex::new("y");
94      let z = Vertex::new("z");
95
96      let mut adj_list = HashMap::new();
97      adj_list.insert(s, vec![(t, 10), (y, 5)]);
98      adj_list.insert(t, vec![(y, 2), (x, 1)]);
99      adj_list.insert(x, vec![(z, 4)]);
100     adj_list.insert(y, vec![(t, 3), (x, 9), (z, 2)]);
101     adj_list.insert(z, vec![(s, 7), (x, 6)]);
102
103     let distances = dijkstra(s, &adj_list);
104
105     for (v, d) in &distances {
106         println!("{} to {}, min distance: {d}",
107                  s.name, v.name);
108     }
109
```

```
110     assert_eq!(distances.get(&t), Some(&8));
111     assert_eq!(distances.get(&s), Some(&0));
112     assert_eq!(distances.get(&y), Some(&5));
113     assert_eq!(distances.get(&x), Some(&9));
114     assert_eq!(distances.get(&z), Some(&7));
115 }
```

在互聯網上使用 Dijkstra 算法的一個問題是，爲了使算法運行，你必須有完整的網絡的圖表示。這意味着每個路由器都有完整的互聯網中所有路由器的地圖，實際上是根本不可能的。這意味着通過因特網路由器發送消息需要使用其他算法來找到最短路徑。實際中網絡信息傳遞使用的是距離矢量路由算法和狀態路由算法，些類算法允許路由器在發送信息時發現對方路由器保存的網絡圖，這些圖包含相互連通的節點信息。通過實時發現這樣的方式獲取網絡圖內容更高效，同時容量大大減小了。

8.9.3 Dijkstra 算法分析

最後，來看 Dijkstra 算法的時間複雜度。構建優先級隊列需要 O(v)，一旦構造了隊列，則對於每個頂點要執行一次 while 循環。因爲頂點都在開始處添加，並且在那之後才被移除。在循環中每次調用 pop，需要 $O(logV)$ 時間。將該部分循環和對 pop 的調用取爲 $O(VlogV)$。for 循環對於圖中的每條邊執行一次，在 for 循環中，對 decrease_key 的調用需要時間 $O(ElogV)$，因此，總的時間複雜度爲 $O((V+E)logV)$。

8.10 總結

本章學習了圖抽象數據類型，以及圖的實現。圖在網絡，交通，計算機，人工智能知識圖譜等領域非常有用。圖使我們能夠解決許多問題，只要可以將原始問題轉換爲圖表示。圖在以下領域有較好的應用。

```
1 强通分量用於簡化圖。
2 深度優先搜索圖的深分支。
3 拓撲排序用於理清繁雜的圖連接。
4 Dijkstra 用於搜索加權圖的最短路徑。
5 廣度優先搜索用於搜索無加權圖的最短路徑。
```

Chapter 9

實戰

9.1 本章目標

用 Rust 數據結構和算法來完成實戰項目
學習並理解實戰項目中的數據結構和算法

9.2 編輯距離

9.2.1 漢明距離

漢明距離（Hamming distance）是指兩個相同長度的序列在相同位置上有多少個符號不同，對二進位序列來說就是“相異的比特數目”。漢明距離同時也是一種編輯距離，即將一個字符串轉換成另一個字符串需要經過多少次替換操作。比如下圖中，trust 轉換爲 rrost 只需要替換兩個字符，所以漢明距離是 2。

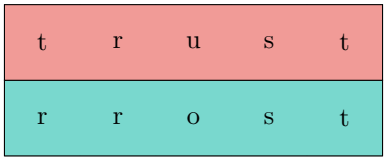


图 9.1: 字符串的漢明距離

漢明距離多用於編碼中的錯誤更正，漢明碼 [12] 中計算距離的算法即爲漢明距離。爲簡化代碼，我們將處理數字和處理字符的漢明距離算法分別實現。計算數字的漢明距離非常簡單，因爲數字可以用位運算直接比較異同，下面是計算數字漢明距離的代碼。

```
1 // hamming_distance.rs
2 fn hamming_distance1(source: u64, target: u64) -> u32 {
3     let mut count = 0;
```

```
4       let mut xor = source ^ target;
5
6       // 异或取值
7       while xor != 0 {
8           count += xor & 1;
9           xor >>= 1;
10      }
11
12      count as u32
13  }
14
15  fn main() {
16      let source = 1;
17      let target = 2;
18      let distance = hamming_distance1(source, target);
19      println!("the hamming distance is {distance}");
20  }
```

通過異或操作可以讓數字 source 和 target 中相同位爲 0，不同位爲 1，如果結果不等於 0，則說明有不同的位，所以從最後一位逐步計算不同的位。xor 與 1 相與就能得到最後一位是 0 還是 1。每計算一位就要移除一位以便比較前面的比特位，所以加入了右移操作。當然，前面的實現需要自己計算二進制中的 1 個數，實際上 Rust 的數字自帶一個 count_ones() 函數用於計算 1 的個數，所以上述代碼可以化簡成如下代碼，非常簡單。

```
1  // hamming_distance.rs
2
3  fn hamming_distance2(source: u64, target: u64) -> u32 {
4      (source ^ target).count_ones()
5  }
6
7  fn main() {
8      let source = 1;
9      let target = 2;
10     let distance = hamming_distance2(source, target);
11     println!("the hamming distance is {distance}");
12 }
```

有了上面的基礎，下面來實現字符版的漢明距離，當然，此時無法用位運算。

```
1  // hamming_distance.rs
2  fn hamming_distance_str(source: &str, target: &str) -> u32 {
```

```
    let mut count = 0;
    let mut source = source.chars();
    let mut target = target.chars();

    // 兩字符串逐字符比較可能出現如下四種情況
    loop {
        match (source.next(), target.next()) {
            (Some(cs), Some(ct)) if cs != ct => count += 1,
            (Some(_), None) | (None, Some(_)) => panic!("Must
                                        have the same lenght"),
            (None, None) => break,
            _ => continue,
        }
    }

    count as u32
}

fn main() {
    let source = "abce";
    let target = "edcf";
    let distance = hamming_distance_str(source, target);
    println!("the hamming distance is {distance}");
}
```

字符版漢明距離算法還是接受 source 和 target 兩個輸入，然後用 chars 方法取出 Unicode 字符來比較。使用 Unicode 而非 ASCII 是因爲字符可能不只有字母，還有中文、日文、韓文等其他字符。if c1 != c2 是在模式匹配之外，額外的條件檢查，只有 source 和 target 都有下一個字符且兩字符不相等時纔會進入該匹配分支。若有任何一個字符是 None，另外一個是 Some，表示輸入字串的長度不同，可直接返回。如果都沒有下一個字符了，則結束。其他情況表示兩個字符相同，則繼續比較下一個字符。漢明距離需要計算所有的字符，所以時間複雜度爲 O(n)，空間複雜度爲 O(1)。

9.2.2 萊文斯坦距離

萊文斯坦距離又稱編輯距離（Edit distance），是一種量化兩字符串差異的算法，表示的是從一個字符串轉換爲另一個字符串最少需要多少次編輯操作。這些操作包括插入、刪除、替換。編輯距離概念非常好理解，操作也簡單，可用於簡單的字符修正。比如用萊文斯坦距離算法來計算單詞 sitting 和 kitten 的編輯距離，可以用如下步驟將 kitten 轉換爲 sitting。

（1）sitting -> kitting，替換 s 爲 k。

（2）kitting -> kitteng，替換 i 爲 e。

（3）kitteng -> kitten，刪除 g。

因爲處理了 3 次，所以編輯距離爲 3。現在的問題是，如何證明 3 就是最少的編輯次數呢？因爲兩個字符串間的轉換隻有三種操作：刪除，插入，替換，所以可以分類單獨計算每種操作的次數，最後取最小值。

一種極端情況是從空字符串轉換爲某長度的字符串 s，此時的編輯距離很明顯就是 s 的長度。比如從空字符串轉換爲 abc，那麼需要插入三個字符，編輯距離就是 3。這同時也說明編輯距離的上限就是較長字符串的長度，用數學公式表達就是下式。

$$max(i,j) \quad if \ \ min(i,j) = 0 \tag{9.1}$$

min(i,j) = 0 表示沒有公共子串，此時取最長字符串的長度。除了這種極端情況，還有可能出現三種操作，而每種操作都會使編輯距離加 1，所以可以分別計算三種操作得到的距離再取最小值，其中 edi 指編輯距離。

$$edi_{a,b}(i,j) = min \begin{cases} edi_{a,b}(i-1,j)+1 \\ edi_{a,b}(i,j-1)+1 \\ edi_{a,b}(i-1,j-1)+1_{a \neq b} \end{cases} \tag{9.2}$$

$edi_{a,b}(i-1,j)+1$ 表示從 a 到 b 要刪除 1 個字符，編輯距離加 1；$edi_{a,b}(i,j-1)+1$ 表示從 a 到 b 要插入 1 個字符，編輯距離再加 1；$edi_{a,b}(i-1,j-1)+1_{a \neq b}$ 表示從 a 到 b 要替換 1 個字符，編輯距離加 1。注意 a 和 b 不等才替換，距離才加 1，相等就跳過。這些函數計算是遞歸定義的，其空間複雜度爲 $O(3^{m+n-1})$，m 和 n 爲字符串的長度。

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1							
i	2							
t	3							
t	4							
e	5							
n	6							

前面我們學習過動態規劃可以處理遞歸，所以此處採用動態規劃算法。首先需要一個矩陣來存儲各種操作後的編輯距離，最基本的情況就是從空字符串到不同長度的字符串所需的編輯距離，如上圖所示。

接下來要計算字符 k 和 s 的編輯距離，分三種情況，如下圖所示。

（1）紅色上方累積刪除的編輯距離 1，加上刪除操作，則編輯距離爲 2。

（2）紅色左方累積插入的編輯距離 1，加上插入操作，則編輯距離爲 2。

（3）紅色對角線累積替換的編輯距離 0，加上替換操作，則編輯距離爲 1。

仔細觀察，可以發現處理的數值都是下圖中黃色區域的值，開始計算時選擇左上角，通過對F色區域的三個值進行計算，最後選擇了結果中最小的值作爲編輯距離填入紅色處，得到了的新的編輯距離。下圖中青色值爲空字符串轉換爲當前長度字符串所需要的編輯次數，這些數值是最極端情況下的編輯距離。

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1	1						
i	2							
t	3							
t	4							
e	5							
n	6							

根據上面的描述和圖可以寫出如下的算法。

```
1  // edit_distance.rs
2
3  use std::cmp::min;
4
5  fn edit_distance(source: &str, target: &str) -> usize {
6      // 極端情況：空字符串到字符串的轉換
7      if source.is_empty() {
8          return target.len();
9      } else if target.is_empty() {
10         return source.len();
```

```
    }

    // 建立矩陣存儲過程值
    let source_c = source.chars().count();
    let target_c = target.chars().count();
    let mut distance = vec![vec![0;target_c+1]; source_c+1];
    for i in 1..=source_c {
        distance[i][0] = i;
    }
    for j in 1..=target_c {
        distance[0][j] = j;
    }

    // 存儲過程值，取增、刪、改中的最小步驟數
    for (i, cs) in source.chars().enumerate() {
        for (j, ct) in target.chars().enumerate() {
            let ins = distance[i+1][j] + 1;
            let del = distance[i][j+1] + 1;
            let sub = distance[i][j] + (cs != ct) as usize;
            distance[i+1][j+1] = min(min(ins, del), sub);
        }
    }

    // 返回最後一行最後一列的值
    *distance.last().and_then(|d| d.last()).unwrap()
}

fn main() {
    let source = "abce";
    let target = "adcf";
    let distance = edit_distance(source, target);
    println!("the edit distance is {distance}");

    let source = "bdfc";
    let target = "adcf";
    let distance = edit_distance(source, target);
    println!("the edit distance is {distance}");
}
```

可通過逐步移動黃色區域來選擇需要計算的三個值，再取計算結果中的最小值填入當前黃色區域的右下角，計算的最終結果如下圖。

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	3

算完整個編輯距離矩陣後，最後一行最後一列的值就是編輯距離。仔細分析上面的圖就會發現，整個矩陣是二維的，處理時要仔細使用下標。一種比較直觀的方式是將矩陣的每一行拉出來放到數組中組成一個大數組，總量還是 m*n，但維度卻小了。但是計算值只有最後一個值有用，而大量中間值浪費了太多內存。因爲計算過程中只需要矩陣中左側黃色區域的值，那麼就可以再優化算法，在計算過程中可以反覆利用一個數組來計算和保存值。將矩陣縮小爲 m + 1 或 n + 1 長度的數組。經過優化的編輯距離算法代碼如下。

```rust
// edit_distance.rs

use std::cmp::min;

fn edit_distance2(source: &str, target: &str) -> usize {
    if source.is_empty() {
        return target.len();
    } else if target.is_empty() {
        return source.len();
    }

    // distance 存儲了到各種字符串的編輯距離
    let target_c = target.chars().count();
    let mut distances = (0..=target_c).collect::<Vec<_>>();
```

```
15      for (i, cs) in source.chars().enumerate() {
16          let mut substt = i;
17          distances[0] = substt + 1;
18          for (j, ct) in target.chars().enumerate() {
19              let dist = min(min(distances[j], distances[j+1])+1,
20                             substt + (cs != ct) as usize);
21              substt = distances[j+1];
22              distances[j+1] = dist;
23          }
24      }
25
26      // 最後一個距離值就是最終答案
27      distances.pop().unwrap()
28  }
29
30  fn main() {
31      let source = "abce";
32      let target = "adcf";
33      let distance = edit_distance2(source, target);
34      println!("the edit distance is {distance}");
35  }
```

優化過的編輯距離算法最壞時間複雜度爲 O(mn)，最壞空間複雜度由矩陣的 O(mn) 降爲 O(min(m,n))，這是非常大的進步。

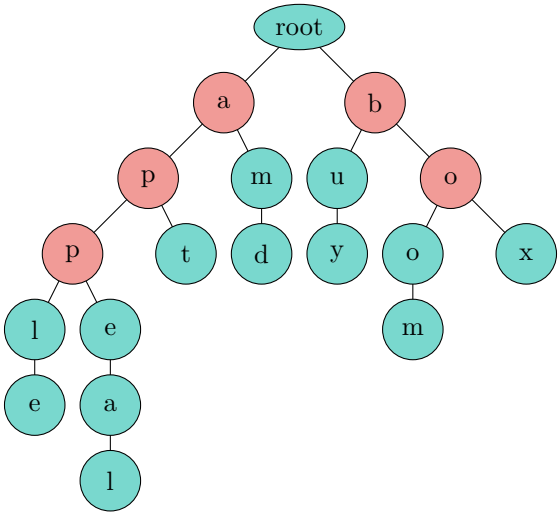
最後提一下微軟的 Word 軟件。Word 也有拼寫檢查功能，但不是用的編輯距離，而是用的散列表。它將常用的幾十萬單詞存儲到散列表，每輸入一個單詞就到散列表中查找，找不到就報錯。散列表速度非常快，而幾十萬單詞也就幾 M 內存，所以非常高效。

9.3 字典樹

Trie (音 try) 是一種樹數據結構，又稱爲字典樹，前綴樹，用於檢索某個單詞或前綴是否存在於樹中。Trie 應用廣泛，包括打字預測，自動補全，拼寫檢查等。

平衡樹和哈希表也能夠用於搜索單詞，爲什麼還需要 Trie 呢? 哈希表能在 O(1) 時間內找到單詞，但卻無法快速地找到具有同一前綴的全部單詞或按字典序枚舉出所有存儲的單詞。Trie 樹優於哈希表的另一個點是，單詞越多，哈希表就越大，這意味着可能出現大量衝突，時間複雜度可能增加到 O(n)。與哈希表相比，Trie 樹存儲多個具有相同前綴的單詞時可以使用更少的空間，時間複雜度也只有 O(m)，m 爲單詞長度，而在平衡樹中查找單詞的時間複雜度爲 O(m log(n)) 。

Trie 樹結構如下，可以發現，存儲單詞只用處理 26 個字母就夠了，而且同樣前綴的單詞共享前綴，節省了存儲空間，比如 apple 和 appeal 共享 app，boom 和 box 共享 bo。



爲實現 Trie，我們首先要抽象出結點 Node ，類似上圖中的結點。Node 中保存子結點的引用和當前結點的狀態。狀態指此結點是否是單詞結束（end），在查詢時可用於判斷單詞是否結束。此外根結點（root）是 Trie 的入口，可以將其用於代表整個 Trie。

```
1  // trie.rs
2
3  // 字典樹定義
4  #[derive(Default)]
5  struct Trie {
6      root: Node,
7  }
8
9  // 節點
10 #[derive(Default)]
11 struct Node {
12     end: bool,
13     children: [Option<Box<Node>>; 26], // 字符節點列表
14 }
15
16 impl Trie {
17     fn new() -> Self {
18         Self::default()
19     }
```

```
20
21      // 單詞插入
22      fn insert(&mut self, word: &str) {
23          let mut node = &mut self.root;
24          // 逐個字符插入
25          for c in word.as_bytes() {
26              let index = (c - b'a') as usize;
27              let next = &mut node.children[index];
28              node = next.get_or_insert_with(Box::<Node>::default);
29          }
30          node.end = true;
31      }
32
33      fn search(&self, word: &str) -> bool {
34          self.word_node(word).map_or(false, |n| n.end)
35      }
36
37      // 判斷是否存在以某個前綴開頭的單詞
38      fn start_with(&self, prefix: &str) -> bool {
39          self.word_node(prefix).is_some()
40      }
41
42      // 前綴字符串
43      // wps: word_prefix_string
44      fn word_node(&self, wps: &str) -> Option<&Node> {
45          let mut node = &self.root;
46          for c in wps.as_bytes() {
47              let index = (c - b'a') as usize;
48              match &node.children[index] {
49                  None => return None,
50                  Some(next) => node = next.as_ref(),
51              }
52          }
53          Some(node)
54      }
55  }
56
57  fn main() {
```

```
58      let mut trie = Trie::new();
59      trie.insert("box"); trie.insert("insert");
60      trie.insert("apple"); trie.insert("appeal");
61
62      let res1 = trie.search("apple");
63      let res2 = trie.search("apples");
64      let res3 = trie.start_with("ins");
65      let res4 = trie.start_with("ina");
66      println!("word 'apple' in Trie: {res1}");
67      println!("word 'apples' in Trie: {res2}");
68      println!("prefix 'ins' in Trie: {res3}");
69      println!("prefix 'ina' in Trie: {res4}");
70  }
```

9.4 過濾器

9.4.1 布隆過濾器

在軟件開發中，常要判斷一個元素是否在一個集合中。比如在字處理軟件中，需要檢查一個英語單詞是否拼寫正確（也就是判斷它是否在已知的字典中，編輯距離一節已經講過 Word 的拼寫檢查）；在 FBI，要快速判斷一個嫌疑人名字是否在嫌疑名單上並給出 FBI Warning；在網絡爬蟲裏，判斷一個網址是否被訪問過等等。最直接的方法就是將集合中全部的元素存在計算機中，遇到一個新元素時，將它和集合中的元素直接比較即可。一般來講，計算機中的集合是用哈希表來存儲的，其好處是快速准確，缺點是費空間。

當集合比較小時，這個問題不顯著，但是當集合巨大時，哈希表存儲效率低的問題就顯現出來了。比如說，一個像雅虎或谷歌郵箱這樣的電子郵件提供商，總是需要過濾垃圾郵件。一個辦法就是記錄下那些發垃圾郵件的地址，由於那些發送者不停註冊新地址，將其都存起來則需要大量的網絡服務器。如果用哈希表，存儲一億個郵件地址就需要 1.6G 左右的內存。將這些信息存入哈希表理論上是可行的，但哈希表存在負載因子，通常空間不能被用滿。此外，如果數據集存儲在遠程服務器上，要在本地接受輸入，而數據集非常大不可能一次性讀進內存構建出哈希表，這時候也會存在問題。

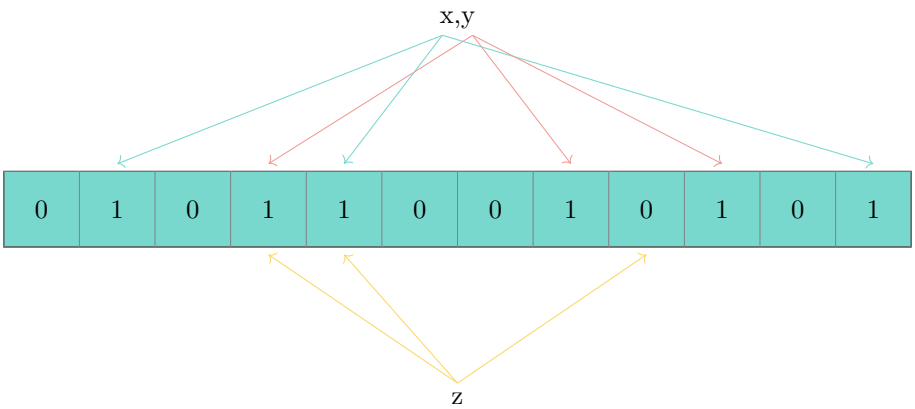
這時候就需要考慮類似布隆過濾器這樣的數據結構。布隆過濾器由布隆（Burton Howard Bloom）在 1970 年提出，它由一個很長的二進制向量和一系列隨機映射函數組成。布隆過濾器可用於檢索一個元素是否在一個集合中，它的優點是空間效率和查詢效率都遠遠超過一般的數據結構，缺點是有一定的識別誤差率且刪除較困難。布隆過濾器本質上是一種巧妙的概率型數據結構。

布隆過濾器包含一個能保存 n 個數據的二進制向量（位數組）和 k 個哈希函數。布隆過濾器支持插入和查詢兩種基本操作，但插入的值總數在設計時就定好了，所以針對不同問題，

要詳細設計布隆過濾器的大小。

初始化布隆過濾器時，所有位置置 0。插入數據時，利用 k 個哈希函數計算數據在過濾器中的位置並將對應位置置 1。比如 k = 3 時，計算三個哈希值作爲下標，並將對應值全部置爲 1。查詢時同樣通過 k 個哈希函數產生 k 個哈希值作爲索引，若所有索引對應的值皆爲 1，則代表該值可能存在。

下圖是三個值對應的情況，x 和 y 都在布隆過濾器中，而 z 因爲最後一個哈希值爲 0，所以一定不存在。想要體驗布隆過濾器的可以到 bloomfilter 這個地址嘗試。



已知布隆過濾器長度爲 n，在可容忍的誤差率爲 ϵ 的情況下，此時最佳的存儲個數爲 m：

$$m = -\frac{nln\epsilon}{(ln2)^2} \tag{9.3}$$

而此時需要的哈希函數個數 k 爲：

$$k = -\frac{ln\epsilon}{ln2} = log_2\epsilon \tag{9.4}$$

假如容忍的誤差率 $\epsilon = 8\%$，那麼 k = 3，k 越大代表誤差率越大。在不改變容錯率的情況下，可以組合迭代次數和兩個基本哈希函數來模擬 k 個哈希函數。

$$g_i(x) = h_1(x) + ih_2(x) \tag{9.5}$$

爲實現布隆過濾器，我們採用一個結構體來封裝所需的全部結構，包括存儲位的集合和哈希函數。因爲只有 1 和 0 兩種情況，我們將其轉換爲 true 或 false 並保存到 Vec 中。這樣在判斷的時候，值是布爾值，可以直接表示是否存在。因爲布隆過濾器只判斷值是否存在，或者說此前是否出現過並有所記錄，所以它必定需要適合任意類型的數據，必然採用泛型。

```
1 // bloom_filter.rs
2
3 use std::collections::hash_map::DefaultHasher;
4
5 // 布隆過濾器
```

```
struct BloomFilter<T> {
    bits: Vec<bool>,              // 比特桶
    hash_fn_count: usize,         // 哈希函數個數
    hashers: [DefaultHasher; 2], // 兩個哈希函數
}
```

但是上述代碼編譯出錯，因爲泛型 T 並沒有被哪個字段用了，那麼編譯器認爲這是非法的。要讓它編譯通過，則需要使用 Rust 中的幽靈數據（PhantomData）來占位，假裝使用了 T，但又不占內存，其實就是騙編譯器，使它放過我們的代碼。最後，爲了儘可能多的支持存儲的數據類型，對於編譯期不定大小的數據我們也要支持，所以加上?Sized 特性讓過濾器支持不定大小的數據。_phantom 前綴表示該字段不使用，占用爲 0。但它帶上了 T ，所以可以騙過編譯器。此外，我們使用兩個隨機的哈希函數來模擬 k 個哈希函數。

```
// bloom_filter.rs

use std::collections::hash_map::DefaultHasher;
use std::marker::PhantomData;

// 布隆過濾器
struct BloomFilter<T: ?Sized> {
    bits: Vec<bool>,
    hash_fn_count: usize,
    hashers: [DefaultHasher; 2],
    _phantom: PhantomData<T>,   // T 占位，欺騙編譯器
}
```

爲了實現過濾器的功能，我們還需要爲其實現三個函數，分別是初始化函數 new，新增元素函數 insert，檢測函數 contains。此外還有輔助函數，用於實現前面三個函數。new 函數需要根據容錯率和大致的存儲規模計算出 m 的大小並初始化過濾器。

```
// bloom_filter.rs

use std::hash::{BuildHasher, Hash, Hasher};
use std::collections::hash_map::RandomState;

impl<T: ?Sized + Hash> BloomFilter<T> {
    fn new(cap: usize, ert: f64) -> Self {
        let ln22 = std::f64::consts::LN_2.powf(2f64);
        // 計算比特桶大小和哈希函數個數
        let bits_count = -1f64 * cap as f64 * ert.ln() / ln22;
        let hash_fn_count = -1f64 * ert.log2();
```

```
12
13          // 隨機哈希函數
14          let hashers = [
15              RandomState::new().build_hasher(),
16              RandomState::new().build_hasher(),
17          ];
18
19          Self {
20              bits: vec![false; bits_count.ceil() as usize],
21              hash_fn_count: hash_fn_count.ceil() as usize,
22              hashers: hashers,
23              _phantom: PhantomData,
24          }
25      }
26
27      // 按照 hash_fn_count 計算值、置比特桶相應位爲 true
28      fn insert(&mut self, elem: &T) {
29          let hashes = self.make_hash(elem);
30          for fn_i in 0..self.hash_fn_count {
31              let index = self.get_index(hashes, fn_i as u64);
32              self.bits[index] = true;
33          }
34      }
35
36      // 數據查詢
37      fn contains(&self, elem: &T) -> bool {
38          let hashes = self.make_hash(elem);
39          (0..self.hash_fn_count).all(|fn_i| {
40              let index = self.get_index(hashes, fn_i as u64);
41              self.bits[index]
42          })
43      }
44
45      // 計算哈希
46      fn make_hash(&self, elem: &T) -> (u64, u64) {
47          let hasher1 = &mut self.hashers[0].clone();
48          let hasher2 = &mut self.hashers[1].clone();
49
```

```
50          elem.hash(hasher1);
51          elem.hash(hasher2);
52
53          (hasher1.finish(), hasher2.finish())
54      }
55
56      // 獲取比特桶某位下標
57      fn get_index(&self, (h1,h2): (u64,u64), fn_i: u64)
58          -> usize {
59          let ih2 = fn_i.wrapping_mul(h2);
60          let h1pih2 = h1.wrapping_add(ih2);
61          ( h1pih2 % self.bits.len() as u64) as usize
62      }
63  }
64
65  fn main() {
66      let mut bf = BloomFilter::new(100, 0.08);
67      (0..20).for_each(|i| bf.insert(&i));
68      let res1 = bf.contains(&2);
69      let res2 = bf.contains(&200);
70      println!("2 in bf: {res1}, 200 in bf: {res2}");
71  }
```

分析布隆過濾器可以發現，其空間複雜度爲 O(m)，插入 insert 和檢測 contains 的時間複雜度爲 O(k)，因爲 k 非常小，所以可以看成 O(1)。

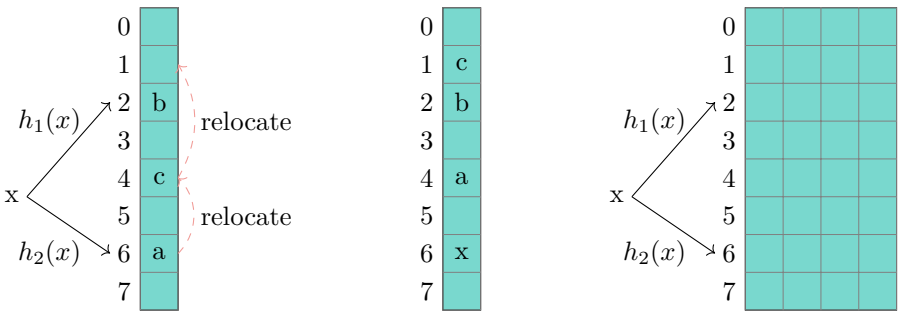
9.4.2 布穀鳥過濾器

前面實現的布隆過濾器容易實現，但也有許多缺點，第一是隨着插入數據越多，誤差率越來越大，第二是不能刪除數據，最後一點是布隆過濾器隨機存儲，在具有 Cache 的 CPU 上性能不好，具體參見 CloudFlare 的博文《When Bloom filters don't bloom》。爲解決布隆過濾器的缺點，布穀鳥過濾器 [13]（Cuckoo filter）應運而生。布穀鳥過濾器是改進的布隆過濾器，它的哈希函數是成對的，分別將數據映射到兩個位置，一個是保存的位置，另一個是備用位置，用於處理碰撞。

布穀鳥過濾器名字來源於布穀鳥，也叫杜鵑，就是“莊生曉夢迷蝴蝶，望帝春心託杜鵑”裏這個杜鵑。這種鳥有一種狡猾又貪婪的習性，它不自己築巢，而是把蛋下到別種鳥的巢裏，由別的鳥幫助孵化出自己的後代。它的幼鳥比別的鳥早出生，所以布穀幼鳥一出生就會拼命把未出生的其它鳥蛋擠出巢，以便今後獨享養父母的食物，真真是鳩佔鵲巢。藉助生物學上的這一現象，布穀鳥過濾器處理碰撞的方法也是把原來位置上的元素踢走，不過被踢出去的

元素還比鳥蛋要幸運些，因爲它還有一個備用位置可以安置，如果備用位置上還有人，再把它踢走，如此往復，直到被踢的次數達到一個上限，才確認哈希表已滿。

布穀鳥過濾器裏存儲的元素不是 0 或 1，而是一定比特位的數據，又稱爲指紋。指紋長度由假陽性率 ϵ 決定，小的 ϵ 需要更長的指紋。布穀鳥過濾器基於布穀鳥哈希表，通過擴展爲二維矩陣得到可以存儲多個指紋的表，如下圖。



布穀鳥哈希表插入 x 時，發現兩個桶都有數據，則隨機踢出 a 到 c，而後 c 移到最上面。布穀鳥過濾器則將桶擴展到 4 個，一個位置可以存儲多個數據，支持插入，刪除，查找。

在左側圖示的標准布穀鳥散列中，將新項插入到現有哈希表中需要方法來訪問原始項，以便確定在需要時將其遷移併爲新項騰出空間。然而，布穀鳥過濾器只存儲指紋，因此沒有辦法重新散列原始項以找到其替代位置。爲突破這個限制，可以利用一種稱爲部分鍵布穀鳥散列的技術來根據其指紋得到項的備用位置。對於項 x，通過散列函數計算兩個候選桶的索引方式如下：

$$
\begin{aligned}
h_1(x) &= hash(x) \\
h_2(x) &= h_1(x) \oplus hash(figureprint(x))
\end{aligned}
\tag{9.6}
$$

異或操作 $\oplus$ 確保 $h_1(x)$ 和 $h_2(x)$ 可以用同一個公式計算出來，這樣就不用管 x 到底是什麼，都可以用式（9.7）計算出備用桶的位置。

$$
j = i \oplus hash(figureprint(x)) \tag{9.7}
$$

查找方法很簡單，利用式（9.6）計算出待查找元素的指紋和兩個備用桶位置，然後讀取兩個桶，任何桶中有值與待查找元素的指紋相等則表示存在。刪除方法也很簡單，檢查兩個備用桶的值，如果有匹配的值，那麼刪除該桶中指紋的副本。同時要注意，刪除前請確保插入了該項，否則可能把碰巧具有相同指紋的其他值刪除了。

通過反覆實驗和測試，桶大小爲 4 時性能非常優秀，甚至就是最佳值。布穀鳥過濾器具有以下四個主要優點：

（1）支持動態添加和刪除項。

（2）比布隆過濾器更高的查找性能，即使當其接近滿載。

（3）比其他的布隆過濾器諸如商過濾器等替代品更容易實現。

（4）在實際應用中，若假陽性率 ϵ 小於 3%，則其使用空間小於布隆過濾器。

表 9.1: 各種過濾器對比

過濾器類型	空間使用	哈希函數個數	刪除功能
布隆過濾器	1	k	no
塊布隆過濾器	1x	1	no
計數布隆過濾器	3x-4x	k	yes
d-left 計數布隆過濾器	1.5x-2x	d	yes
商數過濾器	1x-1.2x	≥ 1	yes
布穀鳥過濾器	$\leq 1x$	2	yes

下面來實現布穀鳥過濾器，前面雖然已經實現過布隆過濾器，但此處需要擴展到二維，所以要新增指紋結構體 FingerPrint，桶結構體 Bucket 用於存儲指紋。因爲涉及到隨機獲取，哈希操作等，所以代碼中還使用了 Rng 和 Serde 等庫。我們將 CuckooFilter 實現爲一個 Rust 庫（lib），其中 bucket.rs 包含指紋和桶的定義及操作，util.rs 包含計算指紋和桶索引的結構體 FaI，整個代碼結構如下。

```
shieber@Kew:cuckoofilter/ tree
 .
 /Cargo.toml
 /src
    |- bucket.rs
    |- lib.rs
    |- util.rs

1 directory, 4 files
```

布穀鳥過濾器代碼非常多，這裏僅將 lib.rs 中列出，其他請參閱隨書源碼。

```
// lib.rs
mod bucket;
mod util;

use std::fmt;
use std::cmp::max;
use std::iter::repeat;
use std::error::Error;
use std::hash::{Hash, Hasher};
use std::marker::PhantomData;
use std::collections::hash_map::DefaultHasher;

```

```
13 // 序列化
14 use rand::Rng;
15 #[cfg(feature = "serde_support")]
16 use serde_derive::{Serialize, Deserialize};
17
18 use crate::util::FaI;
19 use crate::bucket::{Bucket, FingerPrint,
20                     BUCKET_SIZE, FIGERPRINT_SIZE};
21
22 const MAX_RELOCATION: u32 = 100;
23 const DEFAULT_CAPACITY: usize = (1 << 20) - 1;
24
25 // 錯誤處理
26 #[derive(Debug)]
27 enum CuckooError {
28     NotEnoughSpace,
29 }
30
31 // 添加打印輸出功能
32 impl fmt::Display for CuckooError {
33     fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
34         f.write_str("NotEnoughSpace")
35     }
36 }
37
38 impl Error for CuckooError {
39     fn description(&self) -> &str {
40         "Not enough space to save new element, operation failed!"
41     }
42 }
43
44 // 布谷鳥過濾器
45 struct CuckooFilter<H> {
46     buckets: Box<[Bucket]>,   // 桶
47     len: usize,               // 長度
48     _phantom: PhantomData<H>,
49 }
50
```

```
// 添加默認值功能
impl Default for CuckooFilter<DefaultHasher> {
    fn default() -> Self {
        Self::new()
    }
}

impl CuckooFilter<DefaultHasher> {
    fn new() -> Self {
        Self::with_capacity(DEFAULT_CAPACITY)
    }
}

impl<H: Hasher + Default> CuckooFilter<H> {
    fn with_capacity(cap: usize) -> Self {
        let capacity = max(1, cap.next_power_of_two()
                            / BUCKET_SIZE);
        Self {
            // 構建 capacity 個 Bucket
            buckets: repeat(Bucket::new())
                    .take(capacity)
                    .collect::<Vec<_>>(),
                    .into_boxed_slice(),
            len: 0,
            _phantom: PhantomData,
        }
    }

    fn try_insert<T: ?Sized + Hash>(&mut self, elem: &T)
        -> Result<bool, CuckooError> {
        if self.contains(elem) {
            Ok(false)
        } else {
            self.insert(elem).map(|_| true)
        }
    }

    fn insert<T: ?Sized + Hash>(&mut self, elem: &T)
```

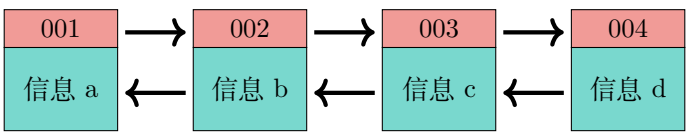
```
89          -> Result<(), CuckooError> {
90          let fai = FaI::from_data::<_, H>(elem)
91          if self.put(fai.fp, fai.i1)
92              || self.put(fai.fp, fai.i2) {
93              return Ok(());
94          }
95
96          // 插入數據衝突，重定位
97          let mut rng = rand::thread_rng();
98          let mut i = fai.random_index(&mut rng);
99          let mut fp = fai.fp;
100         for _ in 0..MAX_RELOCATION {
101             let other_fp;
102             {
103                 let loc = &mut self.buckets[i % self.len]
104                                     .buffer[rng.gen_range(0,
105                                             BUCKET_SIZE)];
106                 other_fp = *loc;
107                 *loc = fp;
108                 i = FaI::get_alt_index::<H>(other_fp, i);
109             }
110             if self.put(other_fp, i) {
111                 return Ok(());
112             }
113             fp = other_fp;
114         }
115         Err(CuckooError::NotEnoughSpace)
116     }
117
118     // 加入指紋
119     fn put(&mut self, fp: FingerPrint, i: usize) -> bool {
120         if self.buckets[i % self.len].insert(fp) {
121             self.len += 1;
122             true
123         } else {
124             false
125         }
126     }
```

```
127
128     fn remove(&mut self, fp: FingerPrint, i: usize) -> bool {
129         if self.buckets[i % self.len].delete(fp) {
130             self.len -= 1;
131             true
132         } else {
133             false
134         }
135     }
136
137     fn contains<T: ?Sized + Hash>(&self, elem: &T) -> bool {
138         let FaI { fp, i1, i2 } = FaI::from_data::<_, H>(elem);
139         self.buckets[i1 % self.len]
140             .get_fp_index(fp)
141             .or_else(|| {
142                 self.buckets[i2 % self.len]
143                     .get_fp_index(fp)
144              })
145             .is_some()
146     }
147 }
```

從代碼中可以看到，布穀鳥過濾器支持插入、刪除、查詢功能。

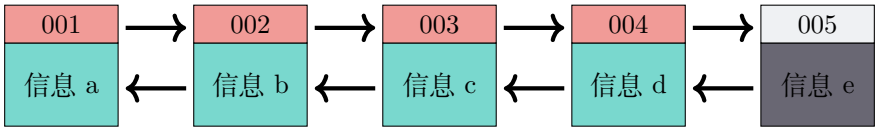
9.5 緩存淘汰算法 LRU

緩存淘汰算法或頁面置換算法，是一種典型的內存管理算法，常用於虛擬頁式存儲，數據緩存。這種算法的原理是“如果數據最近被訪問過，那麼將來被訪問的機率也更高”。對於在內存中但又不用的數據塊，會根據哪些數據屬於最近最少使用而將其移出內存，騰出空間。在這類淘汰算法中，LRU 很常用。LRU（Least recently used，最近最少使用）算法用於在存儲有限的情況下，根據數據的訪問記錄來淘汰數據。假設使用哈希鏈表來緩存用戶信息，容量爲 5，目前緩存了 4 個用戶信息，按時間順序依次從右端插入。

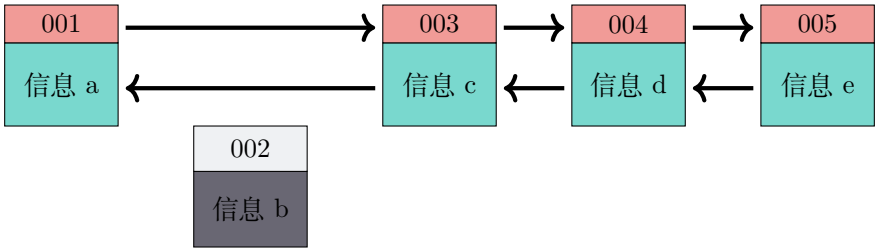


此時，業務方訪問用戶 5，由於哈希鏈表中沒有用戶 5 的數據，我們必須要從數據庫中讀取出來。爲了後續訪問方便，我們將其插入到緩存當中。這時候，鏈表中最右端是最新訪

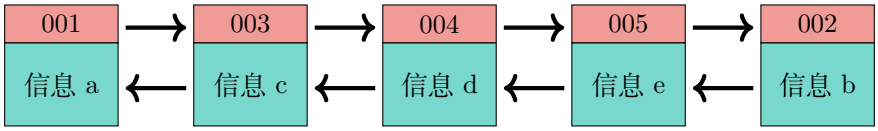
問到的用戶 5，最左端是最近最少訪問的用戶 1。



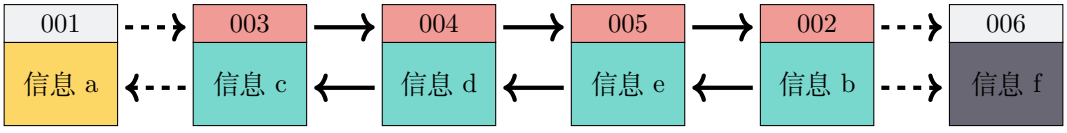
接下來，業務方訪問用戶 2，哈希鏈表中存在用戶 2 的數據，我們怎麼做呢？我們把用戶 2 從它的前驅節點和後繼節點之間移除，重新插入到鏈表最右端。這時候，鏈表中最右端變成了最新訪問到的用戶 2，最左端仍然是最近最少訪問的用戶 1。



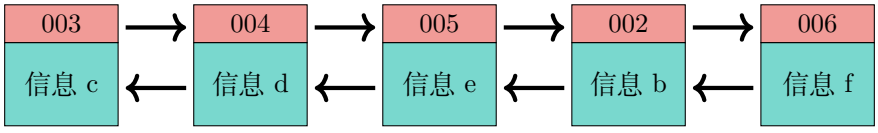
更新數據後，結果如下圖。



後來業務方又訪問了用戶 6，而用戶 6 在緩存裏沒有，需要插入到哈希鏈表。



但這時候緩存容量已達到上限，必須先刪除最近最少訪問的數據，那麼位於哈希鏈表最左端的用戶 1 就會被刪除掉，然後再把用戶 6 插入到最右端。



通過上述圖示，相信你一定已經理解了 LRU 算法的原理。要實現它，就要從圖中抽象出數據結構和操作來。基本上，LRU 需要管理插入數據的鍵（key），數據項（entry），前後指針。操作函數應當包含，插入（insert）、刪除（remove）、查詢（contains），此外還包含許多輔助函數。

如上分析，我們需要首先定義數據項和緩存的數據結構。我們用 HashMap 來存儲鍵，用 Vec 來存儲項，頭尾指針則簡化成了 Vec 中的下標。

```rust
// lru.rs

use std::collections::HashMap;

// LRU 上的元素項
struct Entry<K, V> {
    key: K,
    val: Option<V>,
    next: Option<usize>,
    prev: Option<usize>,
}

// LRU 緩存
struct LRUCache<K, V> {
    cap: usize,
    head: Option<usize>,
    tail: Option<usize>,
    map: HashMap<K, usize>,
    entries: Vec<Entry<K, V>>,
}
```

爲了自定義緩存容量，我們將實現一個 with_capacity 函數，此外默認的 new 則將容量設置爲 100。

```rust
use std::hash::Hash;
const CACHE_SIZE: usize = 100;

impl<K: Clone + Hash + Eq, V> LRUCache<K, V> {
    fn new() -> Self {
        Self::with_capacity(CACHE_SIZE)
    }

    fn with_capacity(cap: usize) -> Self {
        LRUCache {
            cap: cap,
            head: None,
            tail: None,
            map: HashMap::with_capacity(cap),
            entries: Vec::with_capacity(cap),
```

```
16          }
17      }
18  }
```

下面是插入函數，如果插入數據已存在，則直接更新值，那麼插入新值後要將原始值返回。又因爲插入值可能不存在，那麼返回的原始值應該是 None，所以返回是 Option 類型。access 函數用於刪除原始值並更新信息，ensure_room 用於在緩存達到容量時刪除最少使用的數據。

```
1  // lru.rs
2
3  impl<K: Clone + Hash + Eq, V> LRUCache<K, V> {
4      fn insert(&mut self, key: K, val: V) -> Option<V> {
5          if self.map.contains_key(&key) { // 存在 key 就更新
6              self.access(&key);
7              let entry = &mut self.entries[self.head.unwrap()];
8              let old_val = entry.val.take();
9              entry.val = Some(val);
10             old_val
11         } else { // 不存在就插入
12             self.ensure_room();
13
14             // 更新原始頭指針
15             let index = self.entries.len();
16             self.head.map(|e| {
17                 self.entries[e].prev = Some(index);
18             });
19
20             // 新的頭結點
21             self.entries.push(Entry {
22                 key: key.clone(),
23                 val: Some(val),
24                 prev: None,
25                 next: self.head,
26             });
27             self.head = Some(index);
28             self.tail = self.tail.or(self.head);
29             self.map.insert(key, index);
30
```

```
31              None
32          }
33      }
34
35      fn get(&mut self, key: &K) -> Option<&V> {
36          if self.contains(key) {
37              self.access(key);
38          }
39
40          let entries = &self.entries;
41          self.map.get(key).and_then(move |i| {
42              entries[*i].val.as_ref()
43          })
44      }
45
46      fn get_mut(&mut self, key: &K) -> Option<&mut V> {
47          if self.contains(key) {
48              self.access(key);
49          }
50
51          let entries = &mut self.entries;
52          self.map.get(key).and_then(move |i| {
53              entries[*i].val.as_mut()
54          })
55      }
56
57      fn contains(&mut self, key: &K) -> bool {
58          self.map.contains_key(key)
59      }
60
61      // 確保容量够，滿了就移除末尾的元素
62      fn ensure_room(&mut self) {
63          if self.cap == self.len() {
64              self.remove_tail();
65          }
66      }
67
68      fn remove_tail(&mut self) {
```

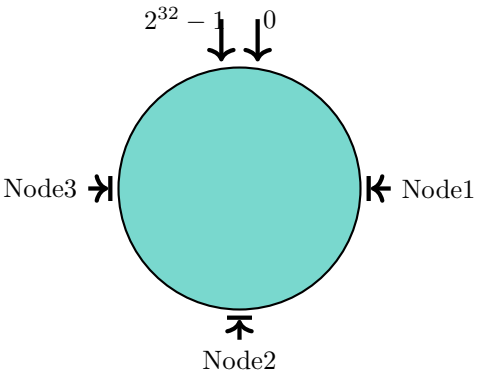
```
69          if let Some(index) = self.tail {
70              self.remove_from_list(index);
71              let key = &self.entries[index].key;
72              self.map.remove(key);
73          }
74          if self.tail.is_none() {
75              self.head = None;
76          }
77      }
78
79      // 獲取某個 key 的值，移除原來位置的值后再從頭部加入
80      fn access(&mut self, key: &K) {
81          let i = *self.map.get(key).unwrap();
82          self.remove_from_list(i);
83          self.head = Some(i);
84      }
85
86      fn remove(&mut self, key: &K) -> Option<V> {
87          self.map.remove(&key).map(|index| {
88                  self.remove_from_list(index);
89                  self.entries[index].val.take().unwrap()
90              })
91      }
92
93      fn remove_from_list(&mut self, i: usize) {
94          let (prev, next) = {
95              let entry = self.entries.get_mut(i).unwrap();
96              (entry.prev, entry.next)
97          };
98
99          match (prev, next) {
100             // 數據項在緩存中間
101             (Some(j), Some(k)) => {
102                 let head = &mut self.entries[j];
103                 head.next = next;
104                 let next = &mut self.entries[k];
105                 next.prev = prev;
106             },
```

```
107                 // 數據項在緩存末尾
108                 (Some(j), None) => {
109                     let head = &mut self.entries[j];
110                     head.next = None;
111                     self.tail = prev;
112                 },
113                 // 數據項在緩存頭部
114                 _ => {
115                     if self.len() > 1 {
116                         let head = &mut self.entries[0];
117                         head.next = None;
118                         let next = &mut self.entries[1];
119                         next.prev = None;
120                     }
121                 },
122             }
123         }
124
125         fn len(&self) -> usize {
126             self.map.len()
127         }
128
129         fn is_empty(&self) -> bool {
130             self.map.is_empty()
131         }
132
133         fn is_full(&self) -> bool {
134             self.map.len() == self.cap
135         }
136     }
```

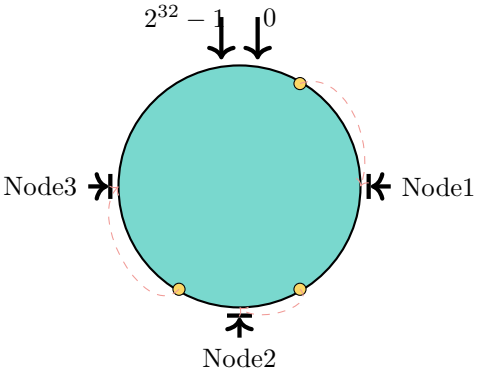
9.6 一致性哈希算法

假設用 Redis 來緩存圖片，數據量小訪問量也不大時，一臺 Redis 就能搞定，最多用個主從就夠了。然而數據量一旦變大，訪問量也增加的時候，全部數據放在一臺機器上不行，畢竟資源有限。這時候，往往會選擇搭建集羣，讓數據分散存儲到多臺機器。比如 5 臺機器，則圖片對應的位置 index = hash(key) % 5。key 是和圖片相關的某個指標。但若是要添加新機器或者有機器出現故障，那麼 N 就會改變，上述計算的 index 就不對。一致性哈希算法的出

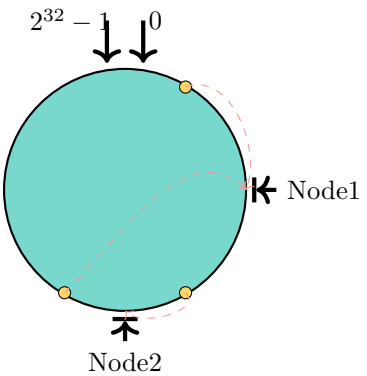
現就是爲了解決這個問題，它以 0 爲起點，在 $2^{32}-1$ 處停止，將這些點圍成一個圓圈。讓數據一定落在圓圈某個位置上。



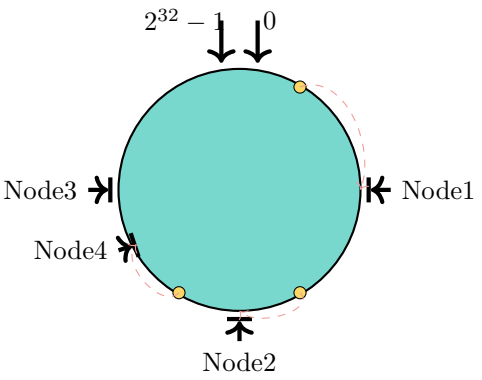
加入數據，則其哈希值會落在某段環上，將數據順時針放到對應的結點可實現緩存。



現在假設結點 Node 3 宕機了，只有 Node2 到 Node3 間的數據項受到影響，它會轉存到結點 Node1 上。



同樣的，如果加入新的機器 Node 4，那麼原本屬於結點 Node 3 的數據會存儲到 Node 4 上。



一致性哈希算法對於節點的增減都只需重定位環空間中的一小部分數據，有很好的容錯性和可擴展性，這也是它能作到一致性的原因。

下面來考慮實現一致性哈希算法。如上分析，我們需要環（Ring）來存儲結點，而結點（Node）代表機器，可以保存主機名（host），ip 和端口（port）爲關鍵信息。

```
1  // conshash.rs
2  use std::fmt::Debug;
3  use std::clone::Clone;
4  use std::string::ToString;
5  use std::hash::{Hash, Hasher};
6  use std::collections::BTreeMap;
7  use std::collections::hash_map::DefaultHasher;
8
9  // 環上節點
10 #[derive(Clone, Debug)]
11 struct Node {
12     host: &'static str,
13     ip: &'static str,
14     port: u16,
15 }
16
17 // 爲 Node 添加 to_string() 功能
18 impl ToString for Node {
19     fn to_string(&self) -> String {
20         format!("{}:{}", self.ip.to_string(),
21                          self.port.to_string())
22     }
23 }
24
```

```
// 環
struct Ring<T: Clone + ToString + Debug> {
    replicas: usize,         // 分區數
    ring: BTreeMap<u64, T>, // 保存數據的環
}
```

Ring 中的 replicas 是爲防止結點聚集而導致數據也集中存儲到少量結點上。對一個結點產生多個虛擬結點，那麼這些結點會更均勻的分佈到環上，就能解決結點聚集問題。

哈希計算可以採用標准庫提供的默認哈希計算器，默認的結點是 10 個，可自定義創建結點數。對一致性哈希算法，我們至少還需要支持插入結點、刪除結點、查詢功能。當然，爲了批量處理，插入和刪除都可以實現批量處理版本。

```
const DEFAULT_REPLICAS: usize = 10;

// 哈希計算函數
fn hash<T: Hash>(val: &T) -> u64 {
    let mut hasher = DefaultHasher::new();
    val.hash(&mut hasher);
    hasher.finish()
}

impl<T> Ring<T> where T: Clone + ToString + Debug {
    fn new() -> Self {
        Self::with_capacity(DEFAULT_REPLICAS)
    }

    fn with_capacity(replicas: usize) -> Self {
        Ring {
            replicas: replicas,
            ring: BTreeMap::new(),
        }
    }

    // 批量插入結點
    fn add_multi(&mut self, nodes: &[T]) {
        if !nodes.is_empty() {
            for node in nodes.iter() { self.add(node); }
        }
    }
```

```
28
29      fn add(&mut self, node: &T) {
30          for i in 0..self.replicas {
31              let key = hash(&format!("{}{}", node.to_string(),
32                                                  i.to_string()));
33              self.ring.insert(key, node.clone());
34          }
35      }
36
37      // 批量刪除結點
38      fn remove_multi(&mut self, nodes: &[T]) {
39          if !nodes.is_empty() {
40              for node in nodes.iter() { self.remove(node); }
41          }
42      }
43
44      fn remove(&mut self, node: &T) {
45          assert!(!self.ring.is_empty());
46          for i in 0..self.replicas {
47              let key = hash(&format!("{}{}", node.to_string(),
48                                                  i.to_string()));
49              self.ring.remove(&key);
50          }
51      }
52
53      // 查詢結點
54      fn get(&self, key: u64) -> Option<&T> {
55          if self.ring.is_empty() { return None; }
56          let mut keys = self.ring.keys();
57          keys.find(|&k| k >= &key)
58              .and_then(|k| self.ring.get(k))
59              .or(keys.nth(0).and_then(|x| self.ring.get(x)))
60      }
61  }
62
63  fn main() {
64      let replica = 3;
65      let mut ring = Ring::with_capacity(replica);
```

```
66
67      let node = Node{ host: "localhost", ip: "127.0.0.1",port: 23 };
68      ring.add(&node);
69
70      for i in 0..replica {
71          let key = hash(&format!("{}{}", node.to_string(),
72                                          i.to_string()));
73          let res = ring.get(key);
74          assert_eq!(node.host, res.unwrap().host);
75      }
76
77      println!("{:?}", &node);
78      ring.remove(&node);
79  }
```

9.7 Base58 編碼

Base58 和 Base64 一樣，是一種編碼算法。用於表示比特幣錢包地址，由中本聰引入。Base58 在 Base64 的基礎上刪除了易引起歧義的（表9.2中紅色）字符，包括 0（零）、O（大寫 O）、I（大寫 i）、l（小寫 L），以及 + 和 / 字符，剩下的 58 個字符作爲編碼字符。這些字符既不容易認錯，又避免了 / 等字符在複F時斷行的問題。

表 9.2: Base 64 編碼字符

編號	字符	編號	字符	編號	字符	編號	字符	編號	字符
0	0	13	D	26	Q	39	d	52	q
1	1	14	E	27	R	40	e	53	r
2	2	15	F	28	S	41	f	54	s
3	3	16	G	29	T	42	g	55	t
4	4	17	H	30	U	43	h	56	u
5	5	18	I	31	V	44	i	57	v
6	6	19	J	32	W	45	j	58	w
7	7	20	K	33	X	46	k	59	x
8	8	21	L	34	Y	47	l	60	y
9	9	22	M	35	Z	48	m	61	z
10	A	23	N	36	a	49	n	62	+
11	B	24	O	37	b	50	o	63	/
12	C	25	P	38	c	51	p		

Base58 的編碼其實是大數進制轉換，先將字符轉換爲 ASCII，然後轉換爲 10 進制，接着是 58 進制，最後按照編碼表選擇對應字符組成 Base58 編碼字符串。因爲涉及數的進制轉換，所以效率比較低，其編碼原理如算法（9.1）。

Algorithm 9.1: Base58 編碼流程

Data: 原始字符串 s
Result: 編碼後字符串 b58
初始化一個空字符串 b58 用於保存結果
for $c \in s$ **do**
    將 s 中字節 c 轉換成 ASCII 值（256 進制）
    將 256 進制數字轉換成 10 進制數字
    將 10 進制數字轉換成 58 進制數字
    將 58 進制數字按照 Base58 字符表轉換成對應字符
    將得到的字符加入 b58
end
返回編碼後的字符串 b58

解碼是個逆過程，同樣的也是大數的進制間轉換，先將其中 Base58 字符串中字符轉換爲 ASCII 值，然後再轉到 10 進制，接着轉到 256 進制，最後再轉到 ASCII 字符。具體解碼原理如算法（9.2）。

Algorithm 9.2: Base58 解碼流程

Data: 編碼後字符串 b58s
Result: 解碼後字符串 s
初始化一個空字符串 s 用於保存結果
for $c \in b58$ **do**
    將 b58 中字節 c 轉換成 ASCII 值（58 進制）
    將 58 進制數字轉換成 10 進制數字
    將 10 進制數字轉換成 256 進制數字
    將 256 進制數字按照 ASCII 表轉換成對應字符
    將得到的字符加入 s
end
返回解碼後的字符串 s

其實編碼和解碼就是兩個空間的字符串轉換，實際上更像是編碼空間的映射。知道了 Base58 的編解碼原理，下面來實現一個簡易的 Base58 編解碼器。首先是准備 Base58 的編碼字符 ALPHABET 和編碼轉換表 BASE58_DIGITS_MAP。

```
1 // base58.rs
```

```
2
3  // base58 變碼字符
4  const ALPHABET: &[u8;58] = b"123456789ABCDEFGHJKLMNPQRSTUVWXYZ
5  abcdefghijkmnopqrstuvwxyz";
6
7  // 進制映射關系表
8  const BASE58_DIGITS_MAP: &'static [i8] = &[
9      -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
10     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
11     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
12     -1, 0, 1, 2, 3, 4, 5, 6, 7, 8,-1,-1,-1,-1,-1,-1,
13     -1, 9,10,11,12,13,14,15,16,-1,17,18,19,20,21,-1,
14     22,23,24,25,26,27,28,29,30,31,32,-1,-1,-1,-1,-1,
15     -1,33,34,35,36,37,38,39,40,41,42,43,-1,44,45,46,
16     47,48,49,50,51,52,53,54,55,56,57,-1,-1,-1,-1,-1,
17 ];
```

爲了應對編解碼可能出現的錯誤, 我們爲 Base58 編碼實現了自定義的錯誤類型, 用於處理字符非法和長度錯誤及其他情況。編碼和解碼我們實現成 str 類型的兩個 trait：Encoder, Decoder, 兩者分別含有 encode 和 decode 方法, 返回 String 和 Result<String, Err>。

```
1  // base58.rs
2
3  // 解碼錯誤類新
4  #[derive(Debug, PartialEq)]
5  pub enum DecodeError {
6      Invalid,
7      InvalidLength,
8      InvalidCharacter(char, usize),
9  }
10
11 // 編解碼 trait
12 pub trait Encoder {
13     fn encode(&self) -> String;
14 }
15
16 pub trait Decoder {
17     fn decode(&self) -> Result<String, DecodeError>;
18 }
```

定義了 trait，接下來就是分別實現 encode 和 decode 方法，具體原理如前所述。此處的 trait 是爲 str 實現的，但內部計算用 u8 比較好，因爲字符串中的字符可能包含多個 u8。

```
// base58.rs

// 實現 base58 編碼
impl Encoder for str {
    fn encode(&self) -> String {
        let str_u8 = self.as_bytes();
        let zero_count = str_u8.iter()
                               .take_while(|&&x| x == 0)
                               .count();
        let size = (str_u8.len() - zero_count) * 138 / 100 + 1;

        // 字符進制轉換
        let mut i = zero_count;
        let mut high = size - 1;
        let mut buffer = vec![0u8; size];
        while i < str_u8.len() {
            let mut j = size - 1;
            let mut carry = str_u8[i] as u32;

            while j > high || carry != 0 {
                carry += 256 * buffer[j] as u32;
                buffer[j] = (carry % 58) as u8;
                carry /= 58;

                if j  > 0 { j -= 1; }
            }

            i += 1;
            high = j;
        }

        // 處理多個前置 0
        let mut base58_str = String::new();
        for _ in 0..zero_count { // 處理多個前置 0
            base58_str.push('1');
        }
```

```
37
38          // 獲取編碼後的字符并拼接成字符串
39          let mut j = buffer.iter()
40                              .take_while(|x| **x == 0)
41                              .count();
42          while j < size {
43              base58_str.push(ALPHABET[buffer[j] as usize]
44                              as char);
45              j += 1;
46          }
47
48          base58_str
49      }
50  }
```

解碼就是將 Base58 編碼逆編碼的過程，具體如下。

```
1   // base58.rs
2
3   // 實現 base58 解碼
4   impl Decoder for str {
5       fn decode(&self) -> Result<String, DecodeError> {
6           let mut bin = [0u8; 132];
7           let mut out = [0u32; (132 + 3) / 4];
8           let bytesleft = (bin.len() % 4) as u8;
9           let zeromask = match bytesleft {
10              0 => 0u32,
11              _ => 0xffffffff << (bytesleft * 8),
12          };
13
14          let zero_count = self.chars()
15                              .take_while(|&x| x == '1')
16                              .count();
17          let mut i = zero_count;
18          let b58: Vec<u8> = self.bytes().collect();
19          while i < self.len() {
20              if (b58[i] & 0x80) != 0 {
21                  return Err(DecodeError::InvalidCharacter(
22                              b58[i] as char, i));
```

```
23              }
24
25              if BASE58_DIGITS_MAP[b58[i] as usize] == -1 {
26                  return Err(DecodeError::InvalidCharacter(
27                              b58[i] as char, i));
28              }
29
30              let mut j = out.len();
31              let mut c = BASE58_DIGITS_MAP[b58[i] as usize]
32                          as u64;
33              while j != 0 {
34                  j -= 1;
35                  let t = out[j] as u64 * 58 + c;
36                  c = (t & 0x3f00000000) >> 32;
37                  out[j] = (t & 0xffffffff) as u32;
38              }
39
40              if c != 0 {
41                  return Err(DecodeError::InvalidLength);
42              }
43
44              if (out[0] & zeromask) != 0 {
45                  return Err(DecodeError::InvalidLength);
46              }
47
48              i += 1;
49          }
50
51          let mut i = 1;
52          let mut j = 0;
53          bin[0] = match bytesleft {
54              3 => ((out[0] & 0xff0000) >> 16) as u8,
55              2 => ((out[0] & 0xff00) >> 8) as u8,
56              1 => {
57                  j = 1;
58                  (out[0] & 0xff) as u8
59              },
60              _ => {
```

```
61                  i = 0;
62                  bin[0]
63              }
64          };
65
66          while j < out.len() {
67              bin[i] = ((out[j] >> 0x18) & 0xff) as u8;
68              bin[i + 1] = ((out[j] >> 0x10) & 0xff) as u8;
69              bin[i + 2] = ((out[j] >> 8) & 0xff) as u8;
70              bin[i + 3] = ((out[j] >> 0) & 0xff) as u8;
71              i += 4;
72              j += 1;
73          }
74
75          let leading_zeros = bin.iter()
76                                  .take_while(|x| **x == 0)
77                                  .count();
78          let new_str = String::from_utf8(
79                                   bin[leading_zeros - zero_count..]
80                                   .to_vec());
81          match new_str {
82              Ok(res) => Ok(res),
83              Err(_) => Err(DecodeError::Invalid),
84          }
85      }
86  }
87
88  fn main() {
89      println!("{:#?}","abc".encode());
90      println!("{:#?}","ZiCa".decode().unwrap());
91      println!("{:#?}","我愛你iloveu".encode());
92      println!("{:#?}","7T5VrNjIVPDWaPpCZW8Fe".decode().unwrap());
93  }
```

至此，整個 Base58 算法就完成了。同樣的，實現 Base32、Base36、Base62、Base64、Base85、Base92 等編解碼算法也是用類似的方法，讀者可自行思考並實現感興趣的編碼算法。本書之所以實現 Base58，是因爲它在數字貨幣中使用非常普遍，這和下面要講的區塊鏈緊密相關。

9.8 區塊鏈

互聯網上的貿易，幾乎都需要藉助金融機構作爲可資信賴的第三方來處理電子支付信息。雖然這類系統在絕大多數情況下都運作良好，但是這類系統仍然內生性地受制於“基於信用的模式”的弱點。我們無法實現完全不可逆的交易，因爲金融機構總是不可避免地會出面協調爭端。而金融中介的存在，也會增加交易的成本，並且限制了實際可行的最小交易規模，也限制了日常的小額支付交易。並且潛在的損失還在於，很多商品和服務本身是無法退貨的，如果缺乏不可逆的支付手段，互聯網的貿易就大大受限。因爲有潛在的退款的可能，就需要交易雙方擁有信任。而商家也必須提防自己的客戶，因此會向客戶索取完全不必要的個人信息。而實際的商業行爲中，一定比例的欺詐性客戶也被認爲是不可避免的，相關損失計入了銷售費用裏。而在使用物理現金的情況下，這些銷售費用和支付問題上的不確定性卻是可以避免的，因爲此時沒有第三方信用中介的存在。所以，我們非常需要這樣一種電子支付系統，它基於密碼學原理而不基於信用，使得任何達成一致的雙方，能夠直接進行支付，從而不需要第三方中介的參與。杜絕回滾支付交易的可能，就可以保護特定的賣家免於欺詐；而對於想要保護買家的人來說，在此環境下設立通常的第三方擔保機制也可謂輕松加愉快。在這篇論文中，我們將提出一種通過點對點分佈式的時間戳服務器來生成依照時間前後排列並加以記錄的電子交易證明，從而解決雙重支付問題。只要誠實的節點所控制的計算能力的總和大於有合作關係的攻擊者的計算能力的總和，該系統就是安全的

上面的這段話是比特幣發明人中本聰在比特幣白皮書 [14]《比特幣：一種點對點電子現金系統》中的介紹，這回答了比特幣發明的原因。其實更實際的問題是 2008 年全球陷入金融危機，通貨膨脹，各國都遭到嚴重衝擊。中本聰對這樣的金融環境不滿，他相信未來國家退出發行貨幣角色就可以規避通貨膨脹。在此基礎上，他結合自己的專業知識發明了比特幣及區塊鏈，其完整概念載於 10 頁白皮書。

9.8.1 區塊鏈及比特幣原理

區塊鏈和比特幣是什麼關係呢？近些年，媒體對區塊鏈和比特幣報到很多，但多是宏觀的敘述，沒有技術細節。其實區塊鏈技術是利用鏈式數據結構來驗證與存儲數據、利用分佈式節點共識算法來生成和更新數據、利用密碼學來保證數據傳輸和訪問安全、利用自動化腳本組成的智能合約來編程和操作數據的一種全新的分佈式基礎架構與計算範式。

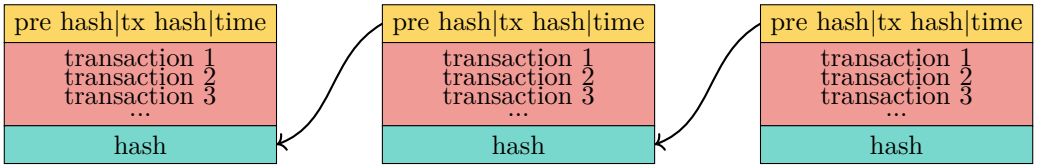
簡單來說，區塊鏈就是去中心化的分佈式賬本。去中心化，就是沒有中心，或者說每個人都可以是中心，這是和傳統的中心化方式不同的地方。分佈式賬本，意味着數據的存儲不只是在每一個節點上，而是每一個節點會複製並共享整個賬本的數據。

區塊鏈是記錄交易的賬本，而記錄交易是非常耗費資源的，所以在記錄交易（打包）的過程中產生了獎勵和手續費，這種獎勵和手續費是一種數字貨幣，用於維持系統的運行。中本聰發明的區塊鏈中的數字貨幣就是大名鼎鼎的比特幣，它也是第一種數字貨幣。

所以可以看出，區塊鏈是一個分佈式的交易媒介，比特幣交易的保障，是一種副產品，一種激勵。區塊鏈作爲一個系統，它本身存在區塊、區塊鏈、交易、賬戶、礦工、手續費、獎勵等組件。要實現區塊鏈系統，就要從這些基本的組件開始，逐一實現。

9.8.2 基礎區塊鏈

一個簡單的區塊包含區塊頭、區塊體、區塊哈希。其中頭包括前一個區塊的哈希(pre_hash)、當前區塊交易哈希（tx_hash)、區塊打包時間（time)；區塊體包含所有交易（transactions)；區塊哈希（hash）是計算區塊頭和區塊體得到的哈希值。區塊及區塊鏈結構如下圖。



從上面的結構可以看出，哈希值是非常重要的，所以第一項工作，我們首先來實現哈希計算。一般來說，計算之前先將區塊結構序列化，再計算哈希值更高效。

我們的簡單區塊鏈的第一個功能是實現序列化和哈希值計算，具體如下面的代碼。?Sized 是爲了處理不定大小的區塊，因爲交易可能多也可能少，數量不一。bincode 用於序列化，crypto 的 Sha3 用於計算哈希。爲了方便查看，我們將所有哈希全轉換成字符串。serialize 序列化後的數據是 &[u8] 類型，而 hash_str 獲取該類型數據並返回字符串。這樣我們就完成了數據結構序列化和哈希。

```
1  // serializer.rs
2  use bincode;
3  use serde::Serialize;
4  use crypto::digest::Digest;
5  use crypto::sha3::Sha3;
6
7  // 序列化數據
8  pub fn serialize<T: ?Sized>(value: &T) -> Vec<u8>
9      where T: Serialize {
10     bincode::serialize(value).unwrap()
11 }
12
13 // 計算 value 哈希值并以 String 形式返回
14 pub fn hash_str(value: &[u8]) -> String {
15     let mut hasher = Sha3::sha3_256();
16     hasher.input(value);
17     hasher.result_str()
18 }
```

有了計算哈希的函數，區塊裏的 hash，pre_hash，tx_hash 就都能計算了，時間則可採用生成區塊時的時間，只有交易 transaction 不定。爲了簡化問題，最開始就用字符串來模擬交易，通過將其放入 Vec 中來表示多筆交易。Rust 中可以用 struct 來表示區塊和區塊頭。

```
pub struct Block {
    pub header: BlockHeader,
    pub tranxs: String,
    pub hash: String,
}

pub struct BlockHeader {
    pub time: i64,
    pub pre_hash: String,
    pub txs_hash: String,
}
```

對於每個區塊，首先要能新建。新建後，還需要更新區塊的哈希值，區塊的實現如下。

```
// block.rs
use std::thread;
use std::time::Duration;
use chrono::prelude::*;
use utils::serializer::{serialize, hash_str};
use serde::Serialize;

// 區塊頭
#[derive(Serialize, Debug, PartialEq, Eq)]
pub struct BlockHeader {
    pub time: i64,
    pub pre_hash: String,
    pub txs_hash: String,
}
// 區塊
#[derive(Debug)]
pub struct Block {
    pub header: BlockHeader,
    pub tranxs: String,
    pub hash: String,
}

impl Block {
    pub fn new(txs: String, pre_hash: String) -> Self {
        // 用延遲 3 秒來莫以挖礦
```

```
26          println!("Start mining .... ");
27          thread::sleep(Duration::from_secs(3));
28
29          // 准備時間、計算交易哈希值
30          let time = Utc::now().timestamp();
31          let txs_ser = serialize(&txs);
32          let txs_hash = hash_str(&txs_ser);
33          let mut block = Block {
34              header: BlockHeader {
35                  time: time,
36                  txs_hash: txs_hash,
37                  pre_hash: pre_hash,
38              },
39              tranxs: txs,
40              hash: "".to_string(),
41          };
42          block.set_hash();
43          println!("Produce a new block!\n");
44          block
45      }
46
47      // 計算并設置區塊哈希值
48      fn set_hash(&mut self) {
49          let header = serialize(&(self.header));
50          self.hash = hash_str(&header);
51      }
52  }
```

有了區塊，接下來就是鏈了。一條鏈需要保存多個區塊，可以用 Vec 來存儲。此外還要能產生第一個區塊（創世區塊）以及添加新區塊。第一個區塊沒有 pre_hash，所以需要手動設置一個，我選擇的是"Bitcoin hit $60000" 的 base64 值作爲創世區塊的 pre_hash。

```
1  // blockchain.rs
2  use crate::block::Block;
3
4  // 創世區塊 pre_hash
5  const PRE_HASH: &str = "22caaf24ef0aea3522c13d133912d2b7
6                          22caaf24ef0aea3522c13d133912d2b7";
7  pub struct BlockChain {
```

```
8       pub blocks: Vec<Block>,
9   }
10
11  impl BlockChain {
12      pub fn new() -> Self {
13          BlockChain { blocks: vec![Self::genesis_block()] }
14      }
15
16      // 生成創世區塊
17      fn genesis_block() -> Block {
18          Block::new("創始區塊".to_string(),PRE_HASH.to_string())
19      }
20
21      // 添加區塊，形成區塊鏈
22      pub fn add_block(&mut self, data: String) {
23          // 獲取前一個區塊的哈希值
24          let pre_block = &self.blocks[self.blocks.len() - 1];
25          let pre_hash  = pre_block.hash.clone();
26          // 構建新區塊并加入鏈
27          let new_block = Block::new(data, pre_hash);
28          self.blocks.push(new_block);
29      }
30
31      // 打印區塊信息
32      pub fn block_info(&self) {
33          for b in self.blocks.iter() { println!("{:#?}", b); }
34      }
35  }
```

爲了運行區塊鏈，我們需要構造交易，用於生成區塊。下面的 main 文件裏採用字符串代表交易 tx，並在打包交易及結束後分別打印出信息。

```
1   // main.rs
2   use core::blockchain::BlockChain as BC;
3
4   fn main() {
5       println!("----------------Mine Info-------------------");
6
7       let mut bc = BC::new();
```

```
8       let tx = "0xabcd -> 0xabce: 5 btc".to_string();
9       bc.add_block(tx);
10      let tx = "0xabcd -> 0xabcf: 2.5 btc".to_string();
11      bc.add_block(String::from(tx));
12
13      println!("-----------------Block Info------------------");
14      bc.block_info();
15  }
```

這些代碼要按照邏輯將其組織起來才能很好的工作，哈希計算放到 utils 下當工具使用，因爲它本身和區塊鏈沒有關係，而 block 和 blockchain 需要放到 core 目錄下，main 用來調用 core，實現區塊的新建和添加。可以用 Cargo 來生成項目 blockchain，具體參見 github 上的 blockchain1 倉庫或 gitee 上的 blockchain1 倉庫。通過上述代碼，我們實現了一個最基本的區塊鏈項目，它能新建及添加區塊，下面是運行的結果，包含挖礦信息、區塊信息。

```
---------------------------------Mine Info-------------------------------------
Start mining ...
Produced a new block!

Start mining ...
Produced a new block!

Start mining ...
Produced a new block!
---------------------------------Block Info-------------------------------------
Block {
    header: BlockHeader {
        time: 1619011220,
        txs_hash: "b868068f9515f7f89a2a0d691508fb380af41166fd4834fee4969bed33b38839",
        pre_hash: "22caaf24ef0aea3522c13d133912d2b722caaf24ef0aea3522c13d133912d2b7",
    },
    tranxs: "创世区块",
    hash: "1215955b17955d31bbda7ba638d7ca240e3431d06fb0bb1a4f06fa21e6bf3ac5",
}
Block {
    header: BlockHeader {
        time: 1619011223,
        txs_hash: "84eeeb7be34240b4a5c45534fc0951ec8f375d93cd3a77d2f154fbdfdde080c1",
        pre_hash: "1215955b17955d31bbda7ba638d7ca240e3431d06fb0bb1a4f06fa21e6bf3ac5",
    },
    tranxs: "0xabcd -> 0xabce: 5 btc",
    hash: "9defaba787ac4034ac37fa516b2e9b5a325901585d198a73827be9036f2f18d2",
}
Block {
    header: BlockHeader {
        time: 1619011226,
        txs_hash: "ca51ee57941a2af26ae2fa02d05f6276f6c2c050535314d6393501b797b13438",
        pre_hash: "9defaba787ac4034ac37fa516b2e9b5a325901585d198a73827be9036f2f18d2",
    },
    tranxs: "0xabcd -> 0xabcf: 2.5 btc",
    hash: "80d05dfffaf6bf476651604de38f4f880013372b82f3286375abb5526e9523a1",
}
```

9.8.3 工作量證明

很明顯，我們的區塊鏈中並沒有計算區塊的函數，都是直接生成區塊，這和實際中比特幣十分鐘才能產生一個是不同的。區塊鏈系統通過維持一個計算任務的難度，使得產生區塊穩定有序。計算任務又稱工作量證明，通過要求礦工對區塊計算一個哈希值並滿足某個難度要求，若通過則加入鏈，並給予獎勵。

我們將計算任務實現在 core 目錄下的 pow.rs 裏面。因爲計算的哈希值往往較大，所以採用 U256 來表示。每打包一個區塊就是要構造一個工作量任務 ProofOfWork。new 通過計算當前難度得到一個目標值，run 則計算值並和這個目標值比較，若不符合，則改變 nonce 值（加一），再計算哈希值，如此往復。

```
1  // pow.rs
2
3  use std::thread;
4  use std::time::Duration;
5  use crate::block::Block;
6  use utils::serializer::{serialize, hash_str, hash_u8};
7  use bigint::U256;
8
9  // nonce 最大值
10 const MAX_NONCE: u32 = 0x7FFFFFFF;
11
12 // 工作量證明任務
13 pub struct ProofOfWork {
14     target: U256,
15 }
16
17 impl ProofOfWork {
18     // 計算當前任務難度值
19     pub fn new(bits: u32) -> Self {
20         let (mant, expt) = {
21             let unshifted_expt = bits >> 24;
22             if unshifted_expt <= 3 {
23                 ((bits & 0xFFFFFF) >>
24                     (8 * (3-unshifted_expt as usize)), 0)
25             } else {
26                 (bits & 0xFFFFFF, 8 * ((bits >> 24) - 3))
27             }
28         };
29
```

```
30          if mant > 0x7FFFFF {
31              Self {
32                  target: Default::default(),
33              }
34          } else {
35              Self {
36                target:U256::from(mant as u64)<<(expt as usize),
37              }
38          }
39      }
40
41      // 開啓工作量證明任務，即挖礦
42      pub fn run(&self, mut block: &mut Block) {
43          println!("Start mining .... ");
44          thread::sleep(Duration::from_secs(3));
45
46          let mut nonce: u32 = 0;
47          while nonce <= MAX_NONCE {
48              // 計算值
49              let hd_ser = Self::prepare_data(&mut block, nonce);
50              let mut hash_u: [u8; 32] = [0; 32];
51              hash_u8(&hd_ser, &mut hash_u);
52
53              // 判斷值是否滿足要求，滿足則計算并設置區塊哈希值
54              let hash_int = U256::from(hash_u);
55              if hash_int <= self.target {
56                  block.hash = hash_str(&hd_ser);
57                  println!("Produce a new block!\n");
58                  return;
59              }
60
61              nonce += 1;
62          }
63      }
64
65      // 准備區塊頭數據
66      fn prepare_data(block: &mut Block, nonce: u32) ->Vec<u8> {
67          block.header.nonce = nonce;
```

```
68          serialize(&(block.header))
69      }
70  }
```

因爲 U256 類型是非常大的數字，所以需要從 [u8] 類型數據生成，這是使用 hash_u8 的原因。自然的，utils 下的 serializer 中需要實現 hash_u8 的函數。

```
1  // serializer.rs
2  // 省略前面內容 ....
3
4  // 計算 value 哈希值後傳遞給 out 參數
5  pub fn hash_u8(value: &[u8], mut out: &mut [u8]) {
6      let mut hasher = Sha3::sha3_256();
7      hasher.input(value);
8      hasher.result(&mut out);
9  }
```

爲了使用 ProofOfWork，blockchain.rs 中相應代碼也需要修改，其結果如下。下面只列出了新增和改變的內容。

```
1  // block.rs
2  // ...
3  use crate::pow::ProofOfWork;
4
5  #[derive(Serialize, Debug, PartialEq, Eq)]
6  pub struct BlockHeader {
7      pub nonce: u32,
8      pub bits: u32,
9      pub time: i64,
10     pub txs_hash: String,
11     pub pre_hash: String,
12 }
13
14 impl Block {
15     pub fn new(txs:String, pre_hash:String, bits:u32) ->Self {
16         // ....
17         let mut block = Block {
18             header: BlockHeader {
19                 time: time,
20                 txs_hash: txs_hash,
21                 pre_hash: pre_hash,
```

```
22                  bits: bits,
23                  nonce: 0,
24              },
25              tranxs: txs,
26              hash: "".to_string(),
27          };
28
29          // 初始化挖礦任務後開始挖礦
30          let pow = ProofOfWork::new(bits);
31          pow.run(&mut block);
32
33          block
34      }
35  }
```

可以發現，此區塊鏈項目是在基礎區塊鏈上逐個添加功能得到的。添加了工作量證明機制的完整區塊鏈項目在 github 上的 blockchain2 倉庫或 gitee 上的 blockchain2倉庫。

9.8.4 區塊鏈存儲

前面我們實現了區塊鏈，但每次運行後，上一次的鏈就沒有了，所以可以考慮將鏈保存下來。保存鏈有很多種辦法，這裏我們選擇 LevelDB [15] 來保存，當然用 RocketDB [16] 也可以。

LevelDB 是由谷歌的 Jeff Dean 和 Sanjay Ghemawat 開發並開源的鍵值存儲數據庫。爲使用此數據庫，我們還需要實現鍵的定義，具體可以實現到 utils 中。

```
1  // bkey.rs
2
3  use bigint::U256;
4  use db_key::Key;
5  use std::mem::transmute;
6
7  // 定義大 Key
8  #[derive(Debug, Copy, Clone, Eq, PartialEq, Ord, PartialOrd)]
9  pub struct BKey {
10     pub val: U256,
11 }
12
13 impl Key for BKey {
14     fn as_slice<T, F: Fn(&[u8]) -> T>(&self, func: F) -> T {
```

```
15          let val = unsafe {
16              transmute::<_, &[u8; 32]>(self)
17          };
18          func(val)
19      }
20
21      fn from_u8(key: &[u8]) -> Self {
22          assert!(key.len() == 32);
23          let mut res: [u8; 32] = [0; 32];
24
25          for (i, val) in key.iter().enumerate() {
26              res[i] = *val;
27          }
28
29          unsafe {
30              transmute::<[u8; 32], Self>(res)
31          }
32      }
33  }
```

而存儲則作爲核心功能放到 core 下面。

```
1   // bcdb.rs
2
3   use leveldb::kv::KV;
4   use leveldb::database::Database;
5   use leveldb::options::{Options, WriteOptions};
6   use utils::bkey;
7   use std::{env, fs};
8
9   pub struct BlockChainDb;
10
11  impl BlockChainDb {
12      // 新建并返回數據庫
13      pub fn new(path: &str) -> Database<bkey::BKey> {
14          let mut dir = env::current_dir().unwrap();
15          dir.push(path);
16
17          let path_buf = dir.clone();
```

```
18          fs::create_dir_all(dir).unwrap();
19
20          let path = path_buf.as_path();
21          let mut opts = Options::new();
22          opts.create_if_missing = true;
23
24          let database = match Database::open(path, opts) {
25              Ok(db) => db,
26              Err(e) => panic!("Failed to open db: {:?}", e),
27          };
28
29          database
30      }
31
32      // 數據寫入數據庫
33      pub fn write_db(db: &mut Database<bkey::BKey>,
34                      key: bkey::BKey, val: &[u8]) {
35          let write_opts = WriteOptions::new();
36          match db.put(write_opts, key, &val) {
37              Ok(_) => (),
38              Err(e) => panic!("Failed to write block: {:?}", e),
39          }
40      }
41  }
```

這樣，在 blockchain.rs 裏就可以使用 bcdb.rs 中的存儲函數來存儲區塊鏈。

```
1  // blockchain.rs
2  // ...
3  use utils::bkey::BKey;
4  use leveldb::database::Database;
5
6  const SAVE_DIR: &str = "bc_db";
7
8  pub struct BlockChain {
9      pub blocks: Vec<Block>,
10     curr_bits: u32,
11     blocks_db: Box<Database<BKey>>,
12 }
```

```
13
14 impl BlockChain {
15     pub fn new() -> Self {
16         let mut db = BlockChainDb::new(SAVE_DIR);
17         let genesis = Self::genesis_block();
18         BlockChain::write_block(&mut db, &genesis);
19         BlockChain::write_tail(&mut db, &genesis);
20         println!("New produced block saved!\n");
21
22         BlockChain {
23             blocks: vec![genesis],
24             curr_bits: CURR_BITS,
25             blocks_db: Box::new(db),
26         }
27     }
28
29     fn genesis_block() -> Block {
30         Block::new("創世區塊".to_string(),
31                    PRE_HASH.to_string(), CURR_BITS)
32     }
33
34     pub fn add_block(&mut self, txs: String) {
35         let pre_block = &self.blocks[self.blocks.len() - 1];
36         let pre_hash  = pre_block.hash.clone();
37         let new_block = Block::new(txs,pre_hash,self.curr_bits);
38
39         // 數據寫入庫
40         Self::write_block(&mut (self.blocks_db), &new_block);
41         Self::write_tail(&mut (self.blocks_db), &new_block);
42
43         println!("New produced block saved!\n");
44         self.blocks.push(new_block);
45     }
46
47     fn write_block(db: &mut Database<BKey>, block: &Block) {
48         // 基於區塊鏈的 header 生成 key
49         let header_ser = serialize(&(block.header));
50         let mut hash_u: [u8; 32] = [0; 32];
```

```
51          hash_u8(&header_ser, &mut hash_u);
52
53          let key = BKey{ val: U256::from(hash_u) };
54          let val = serialize(&block);
55          BlockChainDb::write_db(db, key, &val);
56      }
57
58      // 將區塊哈希值作爲尾巴寫入
59      fn write_tail(mut db: &mut Database<BKey>, block:&Block) {
60          let key = BKey{ val: U256::from("tail".as_bytes()) };
61          let val = serialize(&(block.hash));
62          BlockChainDb::write_db(&mut db, key, &val);
63      }
64  }
```

添加了存儲功能的項目在 github 上的 blockchain3 倉庫或 gitee 上的 blockchain3 倉庫。

9.8.5 交易

前面我們的交易都是用的字符串來代替，其實交易涉及交易雙方、金額、手續費等信息，這些適合封裝到結構體裏面，其具體字段如下。

```
1  pub struct Transaction {
2      pub nonce: u64,
3      pub amount: u64,
4      pub fee: u64,
5      pub from: String,
6      pub to: String,
7      pub sign: String,
8      pub hash: String,
9  }
```

nonce 作爲交易記錄值，amount 和 fee 是金額和手續費，from 和 to 交易雙方的地址，sign 標記一些具體信息，hash 是整個交易的哈希值。下面是完整的交易實現。

```
1  // transaction.rs
2  use serde::Serialize;
3  use utils::serializer::{serialize, hash_str};
4
5  // 交易體
6  #[derive(Serialize, Debug)]
```

```
pub struct Transaction {
    pub nonce: u64,
    pub amount: u64,
    pub fee: u64,
    pub from: String,
    pub to: String,
    pub sign: String,
    pub hash: String,
}

impl Transaction {
    pub fn new(from: String, to: String,
               amount: u64, fee: u64,
               nonce: u64, sign: String) -> Self
    {
        let mut tx = Transaction {
            nonce,
            amount,
            fee,
            from,
            to,
            sign,
            hash:"".to_string(),
        };
        tx.set_hash();

        tx
    }

    pub fn set_hash(&mut self) {
        let txs_ser = serialize(&self);
        self.hash = hash_str(&txs_ser);
    }
}
```

一旦有了交易結構體，那麼可以在區塊中使用，替代前面使用的字符串。

```
// block.rs
// ...
```

```
use crate::transaction::Transaction;

#[derive(Serialize, Debug)]
pub struct Block {
    pub header: BlockHeader,
    pub tranxs: Vec<Transaction>, // 改成了具體交易
    pub hash: String,
}

impl Block {
    pub fn new(txs: Vec<Transaction>, pre_hash: String,
                bits: u32) -> Self {
        // ...
    }
}
```

同樣的，blockchain.rs 也需要修改其中的交易的類型。

```
// ...
use crate::transaction::Transaction;

impl BlockChain {
    fn genesis_block() -> Block {
        // ...
        let tx = Transaction::new(from, to, 0, 0, 0, sign);
    }

    pub fn add_block(&mut self, txs: Vec<Transaction>) {
        // ...
    }
}
```

最後，main 函數也得修改。

```
// ...
use core::transaction::Transaction;

fn main() {
    // ...
    let sign = format!("{} -> {}: 9 btc", from, to);
    let tx   = Transaction::new(from, to, 9, 1, 0, sign);
```

```
8
9       // ...
10      let sign = format!("{} -> {}: 6 btc", from, to);
11      let tx   = Transaction::new(from, to, 6, 1, 0, sign);
12
13      // ...
14  }
```

添加了交易功能的項目在 github 上的 blockchain4 倉庫或 gitee 上的 blockchain4 倉庫。

9.8.6 賬戶

前面的交易結構體包含 from 和 to 字段，表示交易的賬戶，然而實際上我們並沒有賬戶。所以，本節來實現賬戶。賬戶要包含地址、余額，其次還可以包含姓名、哈希標誌。

```
1  pub struct Account {
2      pub nonce: u64,
3      pub balance: u64,
4      pub name: String,
5      pub address: String,
6      pub hash: String,
7  }
```

有了賬戶，可以將交易（轉款）實現爲其函數，這樣賬戶和交易就綁定在一起，比較好理解。爲了模擬，假設初始時，每個賬戶有 100 個比特幣（現實是 0）。

```
1  use crate::transaction::Transaction;
2  use utils::serializer::{serialize, hash_str};
3  use serde::Serialize;
4
5  // 賬戶
6  #[derive(Serialize, Debug, Eq, PartialEq, Clone)]
7  pub struct Account {
8      pub nonce: u64,
9      pub balance: u64,
10     pub name: String,
11     pub address: String,
12     pub hash: String,
13 }
14
15 impl Account {
```

```
16      pub fn new(address: String, name: String) -> Self {
17          let mut account = Account {
18              nonce: 0,
19              name: name,
20              balance: 100,
21              address: address,
22              hash: "".to_string(),
23          };
24          account.set_hash();
25
26          account
27      }
28
29      fn set_hash(&mut self) {
30          let data = serialize(&self);
31          self.hash = hash_str(&data);
32      }
33
34      // 交易，此處就只是轉移比特幣
35      pub fn transfer_to(&mut self, to: &mut Self,
36          amount: u64, fee: u64) -> Result<Transaction, String>
37      {
38          if amount + fee > self.balance {
39              return Err("Error:not enough amount!".to_string());
40          }
41
42          self.balance -= amount;
43          self.balance -= fee;
44          self.nonce += 1;
45          self.set_hash();
46
47          to.balance += amount;
48          to.nonce += 1;
49          to.set_hash();
50          let sign = format!("{} -> {}: {} btc",
51                             self.address.clone(),
52                             to.address.clone(),
53                             amount);
```

```
54          let tx = Transaction::new(self.address.clone(),
55                                    to.address.clone(),
56                                    amount,fee,self.nonce,sign);
57          Ok(tx)
58      }
59
60      pub fn account_info(&self) {
61          println!("{:#?}", &self);
62      }
63  }
```

有了賬戶，那麼 main 函數中的交易就可以用賬戶來處理。

```
1   // main.rs
2   // ..
3   use core::account::Account;
4
5   fn main() {
6       let user1 = "Kim".to_string();
7       let user2 = "Tom".to_string();
8       let user3 = "Jim".to_string();
9       let mut acnt1 = Account::new("0xabcd".to_string(), user1);
10      let mut acnt2 = Account::new("0xabce".to_string(), user2);
11      let mut acnt3 = Account::new("0xabcf".to_string(), user3);
12
13      println!("---------------Mine Info-------------------");
14      // ...
15
16      let res = acnt1.transfer_to(&mut user2, 9, 1);
17      match res {
18          Ok(tx) => bc.add_block(vec![tx]),
19          Err(e) => panic!("{}", e),
20      }
21
22      let res = acnt2.transfer_to(&mut user3, 6, 1);
23      match res {
24          Ok(tx) => bc.add_block(vec![tx]),
25          Err(e) => panic!("{}", e),
26      }
```

```
27
28      println!("--------------Account Info-----------------");
29      let users = vec![&acnt1, &acnt2, &acnt2];
30      for u in users {
31          u.account_info();
32      }
33
34      // ..
35  }
```

添加了賬戶功能的項目在 github 上的 blockchain5 倉庫或 gitee 上的 blockchain5 倉庫。

9.8.7 梅根哈希

前面我們分析過，tx_hash 是所有交易哈希兩兩計算得到的最終結果，但實際還沒有實現，我們前面其實使用的就是對所有交易的哈希。本節來實現計算所有交易的哈希值：梅根哈希。

首先創建一個梅根樹來放置所有哈希值，然後兩兩合併再求哈希值並放入梅根樹中，最後得到唯一的一個哈希值作爲 tx_hash 填入 header 。

```
1  impl Block {
2      pub fn new(txs: Vec<Transaction>, pre_hash: String,
3                 bits: u32) -> Self {
4          let time = Utc::now().timestamp();
5          let txs_hash = Self::merkle_hash_str(&txs);
6
7          // ...
8
9          let pow = ProofOfWork::new(bits);
10         pow.run(&mut block);
11
12         block
13     }
14
15     // 计算梅根哈希值
16     fn merkle_hash_str(txs: &Vec<Transaction>) -> String {
17         if txs.len() == 0 {
18             return "00000000".to_string();
19         }
20
```

```
21          let mut merkle_tree: Vec<String> = Vec::new();
22          for tx in txs {
23              merkle_tree.push(tx.hash.clone());
24          }
25
26          let mut j: u64 = 0;
27          let mut size = merkle_tree.len();
28          while size > 1 {
29              let mut i: u64 = 0;
30              let temp = size as u64;
31
32              while i < temp {
33                  let k = Self::min(i + 1, temp - 1);
34                  let idx1 = (j + i) as usize;
35                  let idx2 = (j + k) as usize;
36                  let hash1 = merkle_tree[idx1].clone();
37                  let hash2 = merkle_tree[idx2].clone();
38                  let merge = format!("{}-{}",hash1, hash2);
39                  let merge_ser = serialize(&merge);
40                  let merge_hash = hash_str(&merge_ser);
41                  merkle_tree.push(merge_hash);
42                  i += 2;
43              }
44
45              j += temp;
46              size = (size + 1) >> 1;
47          }
48
49          if merkle_tree.len() != 0 {
50              merkle_tree.pop().unwrap()
51          } else {
52              "00000000".to_string()
53          }
54      }
55
56      fn min(num1: u64, num2: u64) -> u64 {
57          if num1 >= num2 {
58              num2
```

```
59          } else {
60              num1
61          }
62      }
63  }
```

添加了梅根哈希的項目在 github 上的 blockchain6 倉庫或 gitee 上的 blockchain6 倉庫。

9.8.8 礦工及挖礦

區塊鏈的交易會通過打包來保存成區塊，打包過程又被稱爲挖礦。可是前面的實現中，並沒有礦工，挖礦也只是區塊來調動的，這是不符合實際的，所以還需要設置礦工。有了用戶賬戶、礦工，交易就可以通過礦工打包來存儲區塊到鏈上。一個礦工要包含自己的地址，用於接受比特幣，一個賬戶余額，當然也可以加上名字。

```
1  pub struct Miner {
2      name: String,
3      balance: u64,
4      address: String,
5  }
```

礦工第一筆交易是 coinbase 交易，如果最後打包成功則獲得挖礦獎勵 50 比特幣（隨時間半衰）。其他的交易則陸續加入。然後進行工作量證明，開始挖礦。

```
1  // miner.rs
2  use crate::block::Block;
3  use crate::pow::ProofOfWork;
4  use crate::transaction::Transaction;
5
6  const MINER_NAME: &str = "anonymous";
7
8  // 礦工
9  #[derive(Debug, Clone)]
10 pub struct Miner {
11     name: String,
12     balance: u64,
13     address: String,
14 }
15
16 impl Miner {
17     pub fn new(address: String) -> Self {
```

```
18          Miner {
19              name: MINER_NAME.to_string(),
20              balance: 0,
21              address: address,
22          }
23      }
24
25      pub fn mine_block(&self, txs: &mut Vec<Transaction>,
26                        pre_hash: String, bits: u32) -> Block {
27          let from = "0x0000".to_string();
28          let to = self.address.clone();
29          let sign = format!("{} -> {}: 50 btc", from, to);
30          let coinbase = Transaction::new(from,to,0,0,0,sign);
31
32          // 加入 coinbase 交易和普通交易
33          let mut txs_2: Vec<Transaction> = Vec::new();
34          txs_2.push(coinbase);
35          txs_2.append(txs);
36
37          Self::mine_job(txs_2, pre_hash, bits)
38      }
39
40      // 挖礦任務-工作量證明
41      fn mine_job(txs: Vec<Transaction>, pre_hash: String,
42                  bits: u32) -> Block {
43          let mut block = Block::new(txs, pre_hash, bits);
44          let pow = ProofOfWork::new(bits);
45          pow.run(&mut block);
46
47          block
48      }
49
50      pub fn miner_info(&self) {
51          println!("{:#?}", &self);
52      }
53  }
```

有了礦工，我們還需要將其挖到的礦（區塊）加入區塊鏈中。一種直觀的方法是將區塊鏈和礦工放到一起組成挖礦任務，每當礦工挖到一個區塊就立馬添加到區塊鏈中。我們的挖

礦任務可以定義爲結構體 Mine，內含 Miner 和 BlockChain 兩個變量。

```
pub struct Mine {
    pub miner: Miner,
    pub blockchain: BlockChain,
}
```

這樣 miner 挖到區塊後就可以直接加入鏈。

```
// mine.rs
use crate::miner::Miner;
use crate::blockchain::BlockChain;
use crate::transaction::Transaction;

const MINER_ADDRESS: &str = "0x1b2d";

// 挖礦任務
pub struct Mine {
    pub miner: Miner,
    pub blockchain: BlockChain,
}

impl Mine {
    pub fn new() -> Self {
        Mine {
            blockchain: BlockChain::new(),
            miner: Miner::new(MINER_ADDRESS.to_string()),
        }
    }

    pub fn mining(&mut self, txs: &mut Vec<Transaction>) {
        // 准備 pre_hash 和難度值
        let pre_hash = self.blockchain.curr_hash.clone();
        let bits = self.blockchain.curr_bits.clone();
        // 核心代碼點：開始挖礦
        let block = self.miner.mine_block(txs,pre_hash,bits);
        // 區塊保存
        self.blockchain.add_block(block);
    }
}
```

有了挖礦任務結構體 Mine，我們又可以修改 main 裏的挖礦任務。同時要注意到，我們打印區塊的信息移到了鏈裏，增加了打印礦工信息的代碼。

```
// main.rs
// ...
use core::mine::Mine;

fn main() {
    // ...
    println!("----------------Mine Info-----------------");
    let mut mine = Mine::new();

    let mut txs: Vec<Transaction> = Vec::new();
    let res = user1.transfer_to(&mut user2, 9, 1);
    match res {
        Ok(tx) => txs.push(tx),
        Err(e) => panic!("{}", e),
    }
    let res = user1.transfer_to(&mut user2, 5, 1);
    match res {
        Ok(tx) => txs.push(tx),
        Err(e) => panic!("{}", e),
    }
    mine.mining(&mut txs);

    let mut txs: Vec<Transaction> = Vec::new();
    let res = user2.transfer_to(&mut user3, 6, 1);
    match res {
        Ok(tx) => txs.push(tx),
        Err(e) => panic!("{}", e),
    }
    let res = user2.transfer_to(&mut user3, 3, 1);
    match res {
        Ok(tx) => txs.push(tx),
        Err(e) => panic!("{}", e),
    }
    mine.mining(&mut txs);

    println!("--------------Miner Info-----------------");
```

```
37      mine.miner.miner_info();
38
39      // ...
40
41      println!("--------------Block Info--------------------");
42      mine.blockchain.block_info();
43  }
```

因爲有了這些改變，所以 blockchain.rs 也需要相應改變。

```
1   // blockchain.rs
2   // ...
3   use std::sync::Mutex;
4   use std::collections::HashMap;
5
6   pub struct BlockChain {
7       blocks_db: Box<Database<BKey>>,
8       blocks_index: Mutex<HashMap<String, Block>>,
9       pub gnes_hash: String,
10      pub curr_hash: String,
11      pub curr_bits: u32,
12  }
13
14  impl BlockChain {
15      pub fn new() -> Self {
16          // ...
17
18          let gene_block = genesis.clone();
19          let mut block_index = Mutex::new(HashMap::new());
20          Self::update_hmap(&mut block_index, gene_block);
21
22          // ....
23      }
24
25      fn genesis_block() -> Block {
26          println!("Start mining .... ");
27          let from = "0x0000".to_string();
28          let to   = "0x0000".to_string();
29          let sign = "創世區塊".to_string();
```

```
30          let tx = Transaction::new(from, to, 0, 0, 0, sign);
31          let mut block  = Block::new(vec![tx],
32                                      PRE_HASH.to_string(), INIT_BITS);
33
34          let header_ser=ProofOfWork::prepare_data(&mut block,0);
35          block.hash = hash_str(&header_ser);
36          println!("Produced a new block!");
37
38          block
39      }
40
41      pub fn add_block(&mut self, block: Block) {
42          // ...
43          self.curr_hash = block.hash.clone();
44          self.curr_bits = block.header.bits.clone();
45          Self::update_hmap(&mut self.blocks_index, block);
46      }
47
48      fn update_hmap(hmap: &mut Mutex<HashMap<String, Block>>,
49                     block: Block) {
50          let mut hmap = hmap.lock().unwrap();
51          let hash = block.hash.clone();
52          hmap.insert(hash, block);
53      }
54
55      pub fn  block_info(&self) {
56          let mut hash = self.curr_hash.clone();
57          let hmap = self.blocks_index.lock().unwrap();
58          let mut blocks: Vec<Block> = Vec::new();
59
60          loop {
61              if let Some(b) = hmap.get(&hash) {
62                  blocks.push(b.clone());
63                  hash = b.header.pre_hash.clone();
64              } else {
65                  panic!("Error getting block");
66              }
67
```

```
68                 if hash == self.gnes_hash {
69                     if let Some(b) = hmap.get(&hash) {
70                         blocks.push(b.clone());
71                     }
72                     break;
73                 }
74             }
75             blocks.reverse();
76 
77             for b in blocks {
78                 println!("{:#?}", b);
79             }
80         }
81     }
```

添加了礦工的項目在 github 上的 blockchain7 倉庫或 gitee 上的 blockchain7 倉庫。

9.8.9 比特幣獎勵

最後一步就是將礦工挖礦該得的收益支付給礦工，只需要修改少量代碼就可以了。

```
1  impl Miner {
2      pub fn mine_block(&mut self,
3                        txs: &mut Vec<Transaction>,
4                        pre_hash: String,
5                        bits: u32)  -> Block {
6          let mut fee = 0; // 挖礦手續費
7          for tx in txs.iter() {
8              fee += tx.fee.clone();
9          }
10         // ...
11         // 挖礦奖勵，實際中會半衰 50、25、12.5
12         self.balance += 50;
13         self.balance += fee;
14 
15         block
16     }
17 }
```

最終實現的完整項目在 github 上的 blockchain8 倉庫或 gitee 上的 blockchain8 倉庫。

9.8.10 回顧

現在得到了一個完整區塊鏈項目，我們從基本的區塊鏈開始逐步向其中添加了許多功能，最終的區塊鏈實現了賬戶、礦工、挖礦、交易等功能，這些內容是對前面學習的數據結構的總複習。

下圖是運行過程中輸出的挖礦及區塊鏈、礦工及用戶信息，還可以看到用戶和礦工的余額 balance，錢包地址 address 等信息。

```
--------------------------------Mine Info-------------------------------------
Start mining ...
Produced a new block!
New produced block saved!

Start mining ...
Produced a new block!
New produced block saved!

Start mining ...
Produced a new block!
New produced block saved!
--------------------------------Miner Info-------------------------------------
Miner {
        name: “anonymous”,
        balance: 204,
        address: “0x1b2d”,
}
--------------------------------Account Info-----------------------------------
Account {
        nonce: 2,
        name: “Kim”,
        balance: 84,
        address: “0xabcd”,
        hash: “19a1b86d52dc298fb8ec67080b3cfe4672cb640b79e9b4bfd43fcaacce40e618”,
}
Account {
        nonce: 4,
        name: “Tom”,
        balance: 103,
        address: “0xabce”,
        hash: “bcc52eda8d4b4ef78e2b7c9e5ed720141fe4a4e05646ccf5418ed3fbd2ea7541”,
}
Account {
        nonce: 2,
        name: “Jim”,
        balance: 109,
        address: “0xabcf”,
        hash: “3d6cb5ed0f318bf9c57bd5912937eb8561def6702f84835424de6bc2c37e24c8”,
}
```

下圖是生成的一個區塊，裏面是交易信息，每條交易包含了轉帳金額，手續費，交易雙方賬戶。最後的 hash 值是整個區塊的計算得到的哈希，也是下一個區塊的 pre_hash。

```
Block {
    header: BlockHeader {
        nonce: 0,
        time: 1619008034,
        bits: 553713663,
        txs_hash: "20c4a3b94c760d6a9f0bd9b2cd0dcb683cd6ad57d8c8fd3287df2df2eb5c1d3c",
        pre_hash: "912fdc07d0b5ddb7da343a295f856acdd1d4b4df8d9f0d82e441aeb443df9c7e",
    },
    tranxs: [
        Transaction {
            nonce: 0,
            amount: 0,
            fee: 0,
            from: "0x0000",
            to: "0x1b2d",
            sign: "0x0000 -> 0x1b2d: 50 btc",
            hash: "9a1cd23a5b0ecb6326621ae93c218e8db4499283386ce6b6008d496d469364c2",
        },
        Transaction {
            nonce: 1,
            amount: 9,
            fee: 1,
            from: "0xabcd",
            to: "0xabce",
            sign: "0xabcd -> 0xabce: 9 btc",
            hash: "53d5cc1ac2b19555233eb2a9e8fdb62fb4a19387127f5f3b45b195edaa568305",
        },
        Transaction {
            nonce: 2,
            amount: 5,
            fee: 1,
            from: "0xabcd",
            to: "0xabce",
            sign: "0xabcd -> 0xabce: 5 btc",
            hash: "84fa895cb4f0d21d555e66366af23648987fd81da539d99cb4f6810558863c2d",
        },
    ],
    hash: "163e2ffcb1abc15c941f6cae4534a0383d001857092d31dd8a4b55e27a4da44e",
}
```

9.9 總結

本章的實踐包含許多數據結構，而且都非常有用。我們學習瞭如何實現字典樹，布隆和布穀鳥過濾器；懂得了漢明距離的原理，編輯距離需要動態規劃來解決；緩存淘汰算法 LRU 和一致性哈希算法非常常用。最後逐步實現的區塊鏈是最複雜的項目，它使用到了各種數據結構，算是全書綜合複習。

本書行文至此，學習了各種數據結構，也寫了非常多 Rust 代碼，希望本書能對讀者有所幫助並促進 Rust 在中國的發展。

參考文獻

[1] Multicians. Multics. Website, 1995. https://www.multicians.org.

[2] The Open Group. Unix. Website, 1995. https://unix.org/what_is_unix.html.

[3] Linus. Linux 內核官網. Website, 1991. https://www.kernel.org.

[4] GNU. Gnu/linux. Website, 2010. https://www.gnu.org.

[5] Wikipedia. 量子計算機. Website, 2022. https://en.wikipedia.org/wiki/Quantum_computing.

[6] Bradley N. Miller and David L. Ranum. *Problem Solving with Algorithms and Data Structures Using Python*. Franklin, Beedle & Associates, US, 2011.

[7] Rust Foundation. Rust 基金會. Website, 2021. https://foundation.rust-lang.org/members/.

[8] Wikipedia. Np 完全問題. Website, 2021. https://zh.wikipedia.org/wiki/NP%E5%AE%8C%E5%85%A8.

[9] Wikipedia. 歌德巴赫猜想. Website, 2021. https://zh.wikipedia.org/zh-cn/%E5%93%A5%E5%BE%B7%E5%B7%B4%E8%B5%AB%E7%8C%9C%E6%83%B3.

[10] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search—a log logn search. *Commun. ACM*, 21(7):550–553, jul 1978.

[11] Stanley P. Y. Fung. Is this the simplest (and most surprising) sorting algorithm ever?, 2021.

[12] Wikipedia. 漢明碼. Website, 2022. https://zh.wikipedia.org/zh-hans/%E6%B1%89%E6%98%8E%E7%A0%81.

[13] Bin Fan and David G Andarsen. Cuckoo filter: Practically better than bloom. Website, 2014. https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf.

[14] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Website, 2008. https://bitcoin.org/bitcoin.pdf.

[15] Google. Leveldb. Website, 2010. https://github.com/google/leveldb.

[16] Facebook. Rocksdb. Website, 2014. https://github.com/facebook/rocksdb.