Data Structures and Algorithms in Rust

Problem solving with algorithms and data structures using Rust

Shieber

· Rust China Community ·

Problem-solving with algorithms and data structures using Rust

Shieber

2023.03.28

Preface

The emergence of the transistor sparked a revolution in integrated circuits and chips, leading to the development of the central processing unit, large-capacity storage, and convenient communication facilities. Unix [1] was born out of the failure of Multics [2], which subsequently gave rise to the Linux kernel [3] and its various distributions [4]. By combining open source with network technology, the rapid development of IT was made possible. Technological progress provides a platform and tools for practical ideas, while social progress creates new demands and passions, further promoting technological progress. Although the upper layer of the computer world has given rise to the Internet, cloud computing, blockchain, AI, and the Internet of Things, the basic principles underlying the bottom layer remain unchanged. The fundament of computing is the combination of hardware and its abstract data types and algorithms. Whether it's a regular computer, a supercomputer, or a quantum computer [5], their functions are built on some abstraction of data structures and algorithms.

This book focuses on the design, implementation, and use of abstract data types as they play an important role in computer science. By learning to design abstract data types, one can better implement programming and deepen their understanding of them. The algorithm implementations in this book are not the most optimal or general-purpose engineering implementations, as these tend to be verbose and fail to focus on key principles, which can be harmful to learning. Instead, the code in this book takes different simplification measures for different cases, some using generics and others using specific types. These implementation measures are intended to simplify the code and ensure that each segment of code can be compiled and executed separately to obtain a result.

Prerequisites

Understanding abstract data types does not require specific forms, but implementing code requires consideration of specific forms, which in turn requires a certain level of proficiency in Rust. Although readers' familiarity with Rust and coding style may vary, the basic requirements remain the same. To read this book, readers should ideally have the following abilities and interests:

- Ability to implement complete programs using Rust, including the use of Cargo, rustc, test, etc.
- Ability to use basic data types and structures, including structs, enums, loops, matches, etc.
- Ability to use Rust generics, lifetimes, ownership system, pointers, unsafe code, macros, etc.
- Ability to use built-in libraries (crates) and external libraries, and to set up Cargo.toml.

If readers lack these abilities, they can refer to the section on Rust learning materials in Chapter 1 and find some recommended books and resources to learn Rust first, before returning to this book. The code for this book is available on Github, organized according to chapter and name. Readers are welcome to download and use it, and to point out any errors they find.

This book and all code were written in the Ubuntu 20.04 environment, with Rust version 1.58. The code environment has two types: code with and without line numbers. The former is used to show code, while the latter displays results or other content. All code except for simple or explanatory code, which does not give results, provides outputs. Shorter outputs are commented within the current code box, while more complex outputs are placed in a code box without line numbers.

Whole Book Structure

To provide readers with a clear understanding of the structure of the book and to help advanced users select chapters, we have summarized the contents of the entire book below. The book is divided into ten chapters. Chapter 1 organizes Rust learning materials and reviews basic knowledge. Chapters 2 and 3 introduce the concepts of computer science and algorithm analysis, forming the foundation of the book. Chapters 4 to 7 cover the design and implementation of simple data structures and algorithms, while Chapters 8 and 9 cover more complex tree and graph data structures, which are widely used. These two chapters build upon the higher-level topics covered in the previous chapters. The final chapter contains several practical projects that use the data structures and algorithms to solve real problems.

Readers are encouraged to start with the first three chapters and then progress through the middle chapters and practical project in order. However, readers can also choose to learn certain chapters first and then learn other content.

Chapter 1: Rust Basics. This chapter covers the basics of Rust, including installation of the Rust toolchain, organization of Rust learning resources, a review of Rust fundamentals, and a project. Readers who are already familiar with these topics can choose to skip this chapter. The final project summarizes the learned basic knowledge and demonstrates how Rust modules and code are organized.

Chapter 2: Computer Science. This chapter introduces the definition and concepts of computer science, which can help readers analyze real problems. This includes establishing abstract data type models for data structures, designing and implementing algorithms, and verifying algorithm performance. Abstract data types describe the logic of data operations and hide implementation details, which better abstracts the essence of the problem.

Chapter 3: Algorithm Analysis. Algorithm analysis is a method for understanding program execution time and space performance, and can help determine algorithm performance efficiency. The standard method of algorithm analysis is the Big-O notaion analysis.

Chapter 4: Basic Data Structures. This chapter covers basic data structures in Rust. These data structures include arrays, slices, Vec, and derivative stacks, queues, and more which are based on the linear memory model. The chapter focuses on using Vec to implement these basic data structures.

Chapter 5: Recursion. Recursion is an algorithmic technique that is another form of iteration. It must satisfy the three laws of recursion, and tail recursion is an optimization of recursion. Dynamic programming is an efficient algorithmic technique, which can be implemented using recursion or iteration.

Chapter 6: Search. This chapter covers search algorithms that are used to find elements in a data set or to determine if an element exists. Sequential and non-sequential search algorithms are explained based on whether the data set is ordered or not.

Chapter 7: Sorting. In the previous chapter, the sequential search algorithm required ordered data, but data is generally unordered. Thus, sorting algorithms are needed. This chapter covers ten common sorting algorithms, including bubble sort, quick sort, insertion sort, shell sort, merge sort, selection sort, heap sort, bucket sort, counting sort, and radix sort. Tim sort algorithm, a combination of some common sorting algorithms which is very efficient, has become the default sorting algorithm in many programming languages.

Chapter 8: Trees. This chapter explains how non-linear data structures can be constructed on linear systems. Trees are non-linear data structures that point to subtrees through pointers or references. Binary trees, binary heaps, binary search trees and balanced binary trees are explained. B-trees, B+ trees, and red-black trees are more complex trees.

Chapter 9: Graphs. Unlike trees, graphs have no parent-child node relationship and have connections that can go in any direction. This chapter explains how graphs, a non-linear data structure, are used to handle problems with a large number of nodes and connections, such as network traffic, and path searching. Graph storage forms, including adjacency lists and adjacency matrices, are also explained.

Chapter 10: Practical Projects. This chapter applies the knowledge we learned in the previous chapters to solve practical problems. Distance algorithms, trie, filters, cache eviction algorithms, consistent hashing algorithms, and blockchain are covered. These practical projects help readers deepen their understanding of data structures and improve their Rust coding skills.

Acknowledgments

Rust is an excellent language that provides efficient, secure, and convenient engineering management tools, and it is gradually being integrated into the Linux kernel, making it a potential replacement for some C/C++ work in the future. Although several Rust books are available in the market, the author noticed that there are no algorithm books about Rust, which led to many obstacles in the learning process. Therefore, the author decided to create a simple and convenient Rust book that can help newcomers learn algorithms and data structures. After extensive research [6], thinking, organizing, and combining the author's learning experience, this book was completed. Despite the steep learning curve of Rust, with the right direction and good resources, one can learn the language well. The author hopes that this book will make a small contribution to the Rust learning community.

The primary aim of writing this book is to learn and promote Rust and give back to the entire open-source community, which has enabled the author to learn and grow. The author expresses gratitude to PingCap for developing TiDB, the operating open-source community, and online courses. The author also thanks Zhang Handong, Mike Tang and other members of the Rust Chinese community for organizing Rust conferences and maintaining the community, Ling Hu Yi Chong for sharing Rust learning videos on Bilibili Danmu website, Zhang Handong for promoting Rust language through his book, "The Tao of Rust Programming," and RustMagazine Chinese monthly. The author also acknowledges the Rust Foundation [7], established by Mozilla, AWS, Facebook, Google, Microsoft, and Huawei, for creating a platform that motivates the author to write this book, given the shortage of learning resources.

Finally, the author thanks the University of Electronic Science and Technology of China(UESTC) for providing resources and environment and the care and help of the author's mentor and fellow students in the KC404 teaching and research section where the author learned various technologies and cultures, grew up, and found the direction for his life journey.

Shieber

Contents

Chapter 1

Rust Basic

1.1 Objectives

- Install Rust and learn its toolchain
- Explore learning resources for Rust in various areas
- Review the fundamentals of the Rust programming language

1.2 Install Rust and its Toolchain

Ubuntu 22.04 and later versions come with pre-installed Rust, so you can start using it right away. If you're using other Unix-like systems, use the command below to install Rust. For Windows, refer to the official website. As the code in this book is written in Linux, we recommend readers use Linux or Mac OS to avoid any inconsistencies and errors caused by different environments.

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

After installing Rust on Linux, you need to set environment variables to enable the system to locate Rust tools like the rustc compiler. Add the following three lines to the end of ~/.bashrc.

```
# Rust env-path variables
export RUSTPATH=$HOME/.cargo/bin
export PATH=$PATH:$RUSTPATH
```

Then, save and execute source ~/.bashrc. If you find this process cumbersome, you can download the install_rust.sh script from the first chapter of the source code of this book and run it to complete the Rust toolchain installation. The instructions above will install the rustup tool, which manages and installs the Rust toolchain, including the rustc compiler, cargo project management tool, rustup toolchain management tool, rustdoc documentation tool, rustfmt formatting tool, and rust-gdb debugging tool.

For simple projects, you can use the rustc compiler for compilation. However, for larger projects, you need the cargo tool for management, which internally calls rustc for compilation. Cargo is an excellent tool that handles project building, testing, compilation, and publishing, making it highly efficient. Programmers who have worked with C/C++ projects will appreciate the cargo tool's greatness in handling various libraries and dependencies. Most of the time, we use the rustc compiler instead of the cargo tool, as the projects in this book are not too complex. Rustup manages the installation, upgrade, and uninstallation of Rust tools. Note that the Rust language includes stable and nightly versions, both of which can coexist. You can install Nightly using the following command.

```
$ rustup default nightly
```

After installation, you can use rustup to check the current version in use.

```
$ rustup toolchain list
    stable-x86_64-unknown-linux-gnu
    nightly-x86_64-unknown-linux-gnu (default)
```

To switch between stable and nightly versions, please use the following command.

```
$ rustup default stable # nightly
```

1.3 Learning Resources

Rust is a systems programming language and has a certain level of difficulty. It is impossible to learn Rust without good resources. Below is a collection of learning materials in various fields of Rust compiled by the author. It may not be complete, but most fields are covered.

1.3.1 Books and Documents

Basic: *The Rust Programming Language*, *Easy to Understand Rust*, *Rustlings*, *Rust by Example*, *Rust Primer*, *Rust Cookbook*, *Rust in Action*, *Rust Course*

Advanced: *Cargo Documentation*, *The Tao of Rust Programming*, *Learn Rust With Entirely Too Many Linked Lists*, *Rust Design Patterns*

High-level: *rustc book*, *The Little Book of Rust Macros*, *The Rustonomicon*, *Asynchronous Programming in Rust*

1.3.2 Related Topics

Wasm: https://wasmer.io, https://wasmtime.dev, https://wasmedge.org
HTTP/3: https://github.com/cloudflare/quiche
Algorithms: https://github.com/TheAlgorithms/Rust
Games: https://github.com/bevyengine/bevy
Tools: https://github.com/rustdesk/rustdesk, https://github.com/uutils/coreutils
Blockchain: https://github.com/w3f/polkadot
Databases: https://github.com/tikv, https://github.com/tensorbase/tensorbase
Compilers: https://github.com/rust-lang/rustc_codegen_gcc
Operating Systems: https://github.com/Rust-for-Linux, https://github.com/rcore-os
Web Front-end: https://github.com/yewstack/yew, https://github.com/denoland/deno
Web Back-end: https://actix.rs/, https://github.com/tokio-rs/axum, https://github.com/poem-web/poem

1.3.3 Community and Blogs

Rust Official: https://www.rust-lang.org
Rust Source Code: https://github.com/rust-lang/rust
Rust Documentation: https://doc.rust-lang.org/stable
Rust Reference: https://doc.rust-lang.org/reference
Awesome Rust: https://github.com/rust-unofficial/awesome-rust
Rust lib/crate: https://crates.io, https://lib.rs
Rust Sources: https://www.yuque.com/zhoujiping/programming/rust-materials
Rust Cheat Sheet: https://cheats.rs

1.4 Review

Similar to C/C++, Rust is a systems programming language. This means that concepts learned in those languages can be applied to understand Rust better. However, Rust introduces unique concepts like mutability, ownership, borrowing, and lifetimes, which can be both advantages and challenges.

1.4.1 The History of Rust

Rust is a highly efficient and reliable general-purpose compiled language that utilizes LLVM as its backend. It offers a balance between development efficiency and execution efficiency, making it a popular language among developers. The language has consistently been ranked as the most loved language by developers on Stack Overflow for many years. In 2021, Google, Amazon, Huawei, Microsoft, and Mozilla established a foundation for Rust, and Google even funded the rewriting of the Internet infrastructure Apache httpd and OpenSSL in Rust. Rust also has its own mascot, a red crab named Ferris.

Rust was originally a personal project of Mozilla employee Graydon Hoare, and it received support from the Mozilla Research Institute starting in 2009. Rust was released to the public in 2010. Mozilla supported Rust because the Firefox Gecko engine was behind the times due to various vulnerabilities, historical baggage, and performance bottlenecks. Rust replaced the OCaml-written compiler implementation for self-hosting in 2010-2011 and released version 1.0 in 2015. Rust has a strong and active community, and a stable and testing version are released every six weeks, with a major edition released every three years. Rust employs modern engineering management tool Cargo and has over 100,000 packages published on crates.io.

In addition to development efficiency, Rust also exhibits excellent performance. In 2017, a team of six Portuguese researchers conducted a survey [8] on the performance of various programming languages, and Rust was found to be highly performant. They wrote solutions to ten problems based on 27 languages using the same algorithm, and then ran these solutions, recording the electricity consumption, speed, and memory usage of each language.

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

The figure above provides readers with a clear understanding of the resource usage of various programming languages. Rust's energy consumption, time consumption, and memory consumption indicators are all very favorable. While this comparison may not be completely accurate, the overall trend is clear: Rust is energy-saving and efficient. Considering the critical point of climate change we currently face, the use of energy-saving and efficient languages like Rust to develop software is in line with the trend of Carbon Peaking and Carbon Neutrality Goals. I believe that it can be an important tool for enterprise transformation and look forward to the industry and society reaching a consensus.

Rust has a wide range of applications that extends to command-line tools, DevOps tools, audio and video processing, game engines, search engines, blockchain, the Internet of Things, browsers, cloud-native, network servers, databases, and operating systems. Many universities and enterprises at home and abroad are using Rust extensively. For example, Tsinghua University uses Rust in its new students' rCore and zCore operating systems, ByteDance's Feishu, remote desktop software RustDesk, PingCap's TiDB database, js/ts runtime Deno, and Google's Fuchsia operating system.

1.4.2 Keywords, Comments, Naming conventions

Rust uses 39 keywords (which may increase in the future), and some of these keywords, such as Self and self, can be challenging to learn.

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

Keywords are crucial to programming languages and reflect the designers' thought process behind program design. Different keywords indicate different trade-offs and considerations for various tasks. For instance, Rust's match is powerful, while C's similar switch functionality is less so, resulting in different coding styles and ways of thinking.

To facilitate code understanding and explain complex logic, appropriate explanations must be provided within the program. Comments serve as a useful place for these explanations. Rust supports two types of comments: regular comments and documentation comments. Regular comments have three comment formats.

```
1  // comment style 1
2
3  /* comment style 2 */
4
5  /*
6   * comment style 3
7   * multi-line-comments
8   * multi-line-comments
9   * multi-line-comments
10  */
```

Documentation and testing are highly emphasized in Rust, which is why it features a special documentation comment feature. This feature enables the writing of documentation and test code within comments, allowing code testing through cargo test and document generation through cargo doc.

```
//! This symbol placed at the top of a module file to control
//! the generation of library documentation and to describe
//! the functionality of the entire module.
/// This symbol is placed above the object being described to
/// control the generation of library documentation and to
/// describe functions or structures.
```

Rust's documentation uses Markdown syntax, so using the # symbol is essential.

```
//! Math mod   <-- doc comment, describe mode
//!
/// # Add <-- doc comment, discribe function, test case
/// This function sum the inputs
///
/// # Example   <-- test code, use case
/// use math::add;
/// assert_eq!(3, add(1, 2));
fn add(x: i32, y: i32) -> i32 {
    // sum     <-- regular comment
    x + y
}
```

Naming conventions have been a topic of interest in programming, and Rust has its recommended practices. Rust suggests using UpperCamelCase for class-level content and snake_case for value-level content.

```
Item            Convention
Crate           snake_case
Type            UpperCamelCase
Trait           UpperCamelCase
Enum            UpperCamelCase
Function        snake_case
Method          snake_case
Constructor     new / with_more_details
Converter       from_other_type
Macros          snake_case!
Local variable  snake_case
Static variable SCREAMING_SNAKE_CASE
Constants       SCREAMING_SNAKE_CASE
Type Param      The alphabeta of UpperCamelCase, like T, U, K
Lifetime        lowercase,such as 'a, 'src, 'dest
```

In UpperCamelCase, initialisms and abbreviations of compound words count as one word. For example, Usize should be used instead of USize. In snake_case or SCREAMING_SNAKE_CASE, words should not be composed of single letters unless it is the last word. Therefore, btree_map should be used instead of b_tree_map, and PI_2 instead of PI2. The following code exemplifies Rust's naming conventions, which this book follows, and readers are encouraged to follow as well.

```
// Enum
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```
6
7  // Trait
8  pub trait From<T> {
9      fn from<T> -> Self;
10 }
11
12 // Struct
13 struct Rectangle {
14     height: i32,
15     width: i32,
16 }
17 impl Rectangle {
18     // constructor
19     fn new(height: i32, width: i32) -> Self {
20         Self { height, width }
21     }
22
23     // function
24     fn calc_area(&self) -> i32 {
25         self.height * self.width
26     }
27 }
28
29 // static and constant variables
30 static NAME: &str = "kew";
31 const AGE: i32 = 25;
32
33 // Macro definition
34 macro_rules! add {
35     ($a:expr, $b:expr) => {
36         {
37             $a + $b
38         }
39     }
40 }
41
42 // variable and macro call
43 let sum_of_nums = add!(1, 2);
```

As Rust's popularity grows and its usage expands, a unified coding standard becomes necessary. Currently, Professor Zhang Handong leads a good Rust Coding Standard, which readers can refer to.

1.4.3 Constants, Variables, Data Type

Variables and constants are fundamental concepts in programming, and Rust is no exception. However, Rust introduces mutability and immutability, which can be confusing, especially for beginners transitioning from other programming languages. In Rust, there are three types of values: constants, variables, and static variables. Novices may face difficulties when dealing with mutability and immutability, and they may have to struggle with the compiler to get their code to compile. By understanding the differences between constants, variables, and static variables, Rust programmers can write more concise and efficient code.

Constants are defined using const and are bound to a name that cannot be changed or reassigned.

```
// define constant(like #define used in C/C++)
const AGE: i32 = 1984;
// AGE = 1995; error, mutation not allowed

const NUM: f64 = 233.0;
// const NUM: f64 = 211.0; error, defined
```

Variables are defined using let and can have a value that is mutable or immutable depending on whether mut is used during its definition.

```
let x: f64 = 3.14; // let defines x which can be reassigned
                   // but not mutated.
// x = 6.28; error, x is unmutable
let x: f64 = 2.71 // reassign x

let mut y = 985; // let mut defines y which can be reassigned
                 // and mutated.
y = 996; // y is mutable
let y = 2019; // reassign y
```

Finally, static variables are defined using static and can be mutable or immutable depending on whether mut is used during its definition.

```
static NAME: &str = "shieber" // static variable which can be
                              // used as a constant
// NAME = "kew"; error,NAME is unmutable

static mut NUM: i32 = 100; // static mutable variable
unsafe {
    NUM += 1; // NUM is mutable
    println!("Num:{}",NUM);
}
```

The ”mut” keyword is a constraint in Rust programming, restricting variables from being changed unless ”mut” is added in front of it. Otherwise, the variable is immutable. This differs from other programming languages. Although static variables and constants have similarities, they’re different in reality. Constants are replaced inline as many times as they’re used, while static variables are referenced, with only one global instance. Static variables defined with ”static mut” are wrapped in ”unsafe” to indicate that they’re not safe. It’s recommended to use constants and variables instead, forget about static variables, and avoid coding errors.

Data types are the building blocks of a language, and Rust’s data types are similar to C, while some are similar to Go. There are two types of basic data types in Rust: scalar and compound. Scalar types represent a single value, such as integers, floating-point numbers, Boolean types, and character types. Compound types combine multiple values into one, such as tuples and arrays, which are Rust’s two native compound types.

Integers in Rust are numbers without a decimal part, classified into 12 types based on whether they’re signed or unsigned and their length, denoted by ”i” for signed types and ”u” for unsigned types. A 64-bit machine can handle a 128-bit number by storing it in segments and processing it using multiple registers. ”isize” and ”usize” are integer types that match the machine architecture. So on a 64-bit machine, ”isize” and ”usize” represent ”i64” and ”u64,” respectively. On a 32-bit machine, they represent ”i32” and ”u32.”

```
Size    Signed    Unsigned
8       i8        u8
```

```
16      i16       u16
32      i32       u32
64      i64       u64
128     i128      u128
arch    isize     usize
```

Floating-point numbers in Rust are numbers with a decimal part, having two types: ”f32” and ”f64,” with the default type being ”f64,” and both types being signed.

```
Size     Signed
32       f32
64       f64   (default)
```

The boolean type in Rust is represented by ”bool” and has only two values: true and false, consistent with other programming languages.

The character type ”char” in Rust is the most primitive type, similar to the C language. Characters are declared using single quotes, while string literals are declared using double quotes. The variables ”c” and ”c_str” are completely different types, with characters being four-byte Unicode scalar values, and strings represented as arrays.

```
1 // unicode scalar value
2 let c = 's';
3
4 // dynamic arrays
5 let c_str = "s";
```

Tuples are a type of composite value that combines multiple values of other types. Once declared, the length of a tuple cannot be increased or decreased. Tuples use parentheses to enclose the values, separated by commas. To retrieve values from a tuple, pattern matching and dot notation can be used, with indices starting at 0.

```
1 let tup1: (i8, f32, i64) = (-1, 2.33, 8000_0000);
2 // pattern match
3 let (x, y, z) = tup1;
4
5 let tup2 = (0, 100, 2.4);
6 let zero = tup2.0; // use symbol . to get value
7 let one_hundred = tup2.1;
```

In Rust, the let keyword is not just used to define variables, but can also destructure values through pattern matching. For example, x, y, and z can each obtain the values of -1, 2.33, and 80000000, respectively, through pattern matching. Defining variables is also a form of pattern matching, as let is essentially a pattern matching operation. The unit type in Rust is represented by an empty tuple (), and when there is only one value, it is written as (). The unit value is a special value of this type, and is implicitly returned when an expression does not return any value.

Unlike tuples, arrays in Rust must have elements of the same type, and the length cannot be changed once declared. If a mutable collection is needed, Vec can be used instead. It allows for dynamic resizing and is the preferred option in most situations.

```
1 // define arrays
2 let genders = ["Female", "Male", "Bigender"];
3 let gender_f = genders[0]; // indice element
4
5 // [type; num] define array
```

```
let digits[i32; 5] = [0, 1, 2, 3, 4];
let zeros = [0; 10]; // define an array which holds ten '0'
```

Rust allows for explicit type conversion between data types using the as keyword, but does not provide implicit type conversion between primitive types. Integer conversions in Rust follow the conventions of the C language, and all integer conversions are well-defined. For instance, converting i8 to i32 is a reasonable conversion that can be done explicitly.

```
// type_transfer.rs
#![allow(overflowing_literals)]    // Ignore overflow warnings
                                   // for type conversion.
fn main() {
    let decimal = 61.3214_f32;
    // let integer: u8 = decimal;  // Error, f32 cannot be
                                   // converted to u8.
    let integer = decimal as u8;   // Correct, use as
    let character = integer as char;
    println!("1000 as a u16: {}", 1000 as u16);
    println!("1000 as a u8: {}", 1000 as u8);
}
```

For some complex types, Rust also provides two traits, From and Into, for conversion.

```
pub trait From<T> {
    fn from<T> -> Self;
}
pub trait Into<T> {
    fn into<T> -> T;
}
```

With these two traits, you can provide conversion functionality for various types.

```
// integer_to_complex.rs
#[derive(Debug)]
struct Complex {
    real: i32, // real quantity
    imag: i32  // imaginary quantity
}

// Implement a conversion from i32 to a complex number, where
// the i32 is converted to the real part and the imaginary
// part is set to 0.
impl From<i32> for Complex {
    fn from(real: i32) -> Self {
        Self { real, imag: 0 }
    }
}

fn main() {
    let c1: Complex = Complex::from(2_i32);
    // implemented Into by default
    let c2: Complex = 2_i32.into();
    println!("c1: {:?}, c2: {:?}", c1, c2);
}
```

1.4.4 Statement, Expression, Computors and Flow Control

In Rust, there are two basic syntax categories: Statements and Expressions. Statements refer to operations to be executed, while expressions are simply used to evaluate values. Rust statements include declaration statements and expression statements. Declaration statements are used to declare variables, static variables, constants, structs, functions, external packages, and external modules. Statements that end with a semicolon are expression statements.

```
// Declaration statement for function
fn sum_of_nums(nums: &[i32]) -> i32{
    nums.iter().sum::<i32>()
}

let x = 5;     // The entire line is a statement, x = 5 is an
               // expression that calculates the value of x
x + 1;         // The entire line is an expression
let y = x + 1; // The entire line is a statement, y = x + 1
               // is an expression
println!("{y}");

let z = [1,2,3];
println!("sum is {:?}", sum_of_nums(&z));
```

Operators are symbols used to specify the type of calculation performed by an expression. Rust supports common types of operators including arithmetic, relational, logical, assignment, and reference operators. Here are some Rust operators and their functions:

```
Operator  Example                   Explanation
+         expr + expr               addition
+         trait + trait, 'a + trait compound type constraint
-         expr - expr               subtraction
-         - expr                    negation
*         expr * expr               multiply
*         *expr                     dereference
*         *const type, *mut type    raw pointer
/         expr / expr               division
%         expr % expr               remainder
=         var = expr, ident = type  asignmen/equality
+=        var += expr               addition and assignment
-=        var -= expr               subtraction and assignment
*=        var *= expr               multiply and assignment
/=        var /= expr               division and assignment
%=        var %= expr               remainder and assignment

==        expr == expr              equality comparison
!=        var != expr               inequality comparison
>         expr > expr               greater-than comparison
<         expr < expr               less-than comparison
>=        expr >= expr              >= comparison
<=        expr <= expr              <= comparison

&&        expr && expr              logical AND
||        expr || expr              logical OR
!         !expr                     bitwise NOT or logical NOT
```

```
&          expr & expr                 bitwise AND
&          &expr, &mut expr            borrow
&          &type, &mut type,           pointer/reference
|          pat | pat                   pattern matching
|          expr | expr                 bitwise OR
^          expr ^ expr                 bitwise XOR
<<         expr << expr                left shift
>>         expr >> expr                right shift
&=         var &= expr                 bitwise AND and assignment
|=         var |= expr                 bitwise OR and assignment
^=         var ^= expr                 bitwise XOR and assignment
<<=        var <<= expr                left shift and assignment
>>=        var >>= expr                right shift and assignment

.          expr.ident                  member access
..         .., expr..,
           ..expr, expr..expr          right-exclusive range
..         ..expr                      struct update syntax
..=        ..=expr, expr..=expr        right-inclusive range
:          pat: type, ident: type      type constraint
:          ident: expr                 struct field initialization
:          'a: loop {...}              loop label
;          [type; len]                 fixed-size array type
=>         pat => expr                 match arm
@          ident @ pat                 pattern binding
?          expr?                       error propagation
->         fn(...) -> type,
           |...| -> type               function or closure
                                       return type
```

The ability to decide whether to execute a piece of code based on a condition or to repeatedly run a piece of code depending on whether a condition is satisfied is a feature found in most programming languages. In Rust, if expressions and loops are the most common structures used to control execution flow.

```
1 let a = 3;
2
3 if a > 5 {
4     println!("Greater than 5");
5 } else if a > 3 {
6     println!("Greater than 3");
7 } else {
8     println!("less or equal to 3");
9 }
```

if and let can also be combined to control code execution. One way to do this is with a let if statement, which returns the result to the let section if the condition is met. Note that there is a semicolon at the end because let c = ..; is a statement.

```
1 let a = 3; let b = 2;
2 let c = if a > b {
3             true
```

```
4           } else {
5               false
6           };
```

Another way is with an if let statement, which executes code by pattern matching the value on the right.

```
1 let some_value = Some(100);
2 if let Some(value) = some_value {
3     println!("value: {value}");
4 } else {
5     println!("no value");
6 }
```

In addition to the if let matching statement, match can also be used to control code execution.

```
1 let a = 10;
2 match a {
3     0 => println!("0 == a"),
4     1..=9 => println!("1 <= a <= 9"),
5     _ => println!("10 <= a"),
6 }
```

Rust provides various loop types, including loop, while, and for in. Using the continue and break keywords, you can jump and stop code execution on demand. The loop keyword controls repeated code execution until a condition is met.

```
1  let mut val = 10;
2  let res = loop {
3      // break loop and return value
4      if val < 0  {
5          break val;
6      }
7
8      val -= 1;
9      if 0 == val % 2 {
10         continue;
11     }
12
13     println!("val = {val}");
14 }; // a semi-colon here
15
16 // dead loop
17 loop {
18     if res > 0 { break; }
19
20     println!("{res}");
21 } // no semi-colon here
```

In contrast, a while loop computes the loop condition externally.

```
1 let num = 10;
2 while num > 0 {
3     println!("{}", num);
4     num -= 1;
```

```
}

let nums = [1,2,3,4,5,6];
let mut index = 0;
while index < 6 {
    println!("val: {}", nums[index]);
    index += 1;
}
```

Iterating through an array in Rust can be achieved through the for in loop, which is more convenient than using index and length.

```
let nums = [1,2,3,4,5,6];

// iterate over array
for num in nums {
    println!("val: {num}");
}

// iterate over array in reverse order
for num in nums.iter().rev() {
    println!("val: {num}");
}
```

This approach is not only concise, but also reduces the chance of errors, as it avoids specifying the array length explicitly. Additionally, iter().rev() can be used to traverse the array in reverse order.

The combination of while and let can be used to form a pattern match, eliminating the need to write a stop condition. The let syntax automatically determines when the condition is met before continuing while.

```
let mut v = vec![1,2,3,4,5,6];
while let Some(x) = v.pop() {
    println!("{x}");
}
```

Overall, Rust provides many useful methods to control code flow, including match, if let, let if, while let, which align with Rust's coding specifications and are recommended.

1.4.5 Function, Program Structure

Functions are a crucial building block in programming languages, and in Rust, they are defined using the fn keyword followed by a snake-case function name. Function arguments are enclosed in parentheses and written as val: type, for example, x: i32. The return value is specified using -> res and may or may not have a value. The function implementation is wrapped in curly braces. Like C, Rust has a main function.

```
// define a function
fn func_name(parameters) -> return_types {
    code_body;  // code body

    return_value // return value, no semi-colon here
}
```

Define a summation function.

```rust
// main function
fn main() {
    let res = add(1, 2);
    println!("1 + 2 = {res}");
}

fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

It is important to note that function definitions can be written before or after the main function, and the return value can be written directly on the value line without adding a semicolon, although this syntax is unique to Rust and adding a semicolon is not recommended. Functions in Rust modules are private by default and require the pub keyword to export them to other programs.

In Rust, a program is composed of several parts.

```
package/lib/crate/mod
variable
statement/expression
function
trait
label
comment
```

The following example shows the various elements of a program.

```rust
// rust_example.rs

// import module from standard library
use std::cmp::max;

// public module
pub mod math {
    // public function
    pub fn add(x: i32, y: i32) -> i32 { x + y }

    // private function
    fn is_zero(num: i32) -> bool { 0 == num }
}

// struct
#[derive(Debug)]
struct Circle { radius: f32, // radius }

// implement a converter to convert f32 to Circle
impl From<f32> for Circle {
    fn from(radius: f32) -> Self {
        Self { radius }
    }
}

// comment: custom function
```

```
fn calc_func(num1:i32, num2:i32) -> i32 {
    let x = 5;
    let y = {
        let x = 3;
        x + 1 // expression
    }; // statement

    max(x, y)
}

// use function from math module
use math::add;

// main function
fn main() {
    let num1 = 1; let num2 = 2;

    // call function
    println!("num1 + num2 = {}", add(num1, num2));
    println!("res = {}", calc_func(num1, num2));

    let f: f32 = 9.85;
    let c: Circle = Circle::from(f);
    println!("{:?}", c);
}
```

1.4.6 Ownership, Scope, Lifetime

Rust's ownership system is a crucial concept that deals with resource management, dangling references, and other complexities in the type system. It introduces the concepts of ownership, borrowing, and lifetime to restrict the state and scope of various quantities. The scope rules in Rust are the strictest among all programming languages.

In Rust, memory is managed through the ownership system, which is unlike garbage collection or manual memory release in other languages. The compiler checks a set of ownership rules at compile time, and if any rule is violated, the program doesn't even compile. The ownership rules are simple:

- Each value has an owner (variable).
- Values have only one owner at any given time.
- When the owner leaves the scope, the value is discarded.

This means that values in Rust are managed by a unique object, and memory is released as soon as they are no longer in use.

Rust's ownership rules make it more memory efficient because it releases unused memory as soon as it is no longer needed, which can then be reused by later variables. In contrast, languages that use garbage collection, such as in Go, which reclaims memory through the tricolor method and could lead to stop the world problems and program lag, or manual memory release may cause memory leaks as unused memory may not be released. The ownership mechanism is achieved through the automatic call of the drop method when the variable leaves the scope.

```
fn main() {
    let long = 10;       <--- long enters scope (main)

    { // this is a new temp scope
        let short = 5;   <--- short enters scope (temp)
```

```
6           println!("inner short: {}", short);
7
8           let long = 3.14; <--- long enters scope (temp)
9           println!("inner long: {}", long);
10      }                    <--- long and short leave scope (temp)
11
12      let long = 'a';      <--- long reassigns to a char
13      println!("outer long: {}", long);
14  }                        <--- long leaves scope (main)
```

The above shows the scope of each variable. The ownership mechanism is achieved by automatically calling the drop method of the variable when it leaves the scope (e.g. } in this case). Note that the internal long and the external long are two different variables, and the internal long does not override the external long.

To illustrate Rust's ownership mechanism, we can compare it to human society. For example, when a reader buys a book, they own it. If a friend borrows the book, the ownership still belongs to the reader. However, if the reader gives the book to the friend, the ownership transfers to the friend. Rust also has the concepts of "borrow" and "move," which are illustrated in the below example.

```
1  fn main() {
2      let x = "Shieber".to_string(); // create a string
3      let y = x; // move ownership of x to y
4      // println!("{x}"); // x is no longer valid
5  }   <--- y calls drop
```

Let y = x; make a move, which transfers ownership of x to y. Some readers may wonder, "Isn't it released only when you leave the scope?", x and y are still in the main scope, how can it report an error? Actually, the second rule of ownership mechanism is very clear: a value can have only one owner at any time. So after the variable x is moved, it is immediately released and cannot be used later. Its scope only goes to the line let y = x; and not to the line }. In addition, this mechanism ensures that only y, not x, is freed at }, avoiding the memory safety problem of secondary freeing. To use both x and y, you can do something like the following.

```
1  fn main() {
2      let x = "Shieber".to_string(); // create a string
3      let y = &x; // borrow x
4      println!("{x}"); // x is still valid
5  }  <--- y calls drop
```

let y = &x; is called a borrowing, &x is a reference to x. A reference is like a pointer (address) from which you can access data stored at that address that belongs to another variable, and a reference is created to be borrowed by someone else.

The book we borrowed cannot, by definition, be altered in any way, i.e. it is immutable. However, there's nothing wrong with a friend sketching over it. Similarly, there are mutable and immutable variables borrowed from Rust, and the let y = &x; above shows that y is just a non-variable borrowed from x. If you want mutable, you need to add the mut modifier.

```
1  fn main() {
2      let x = "Shieber".to_string(); // create a string
3      let y = &mut x; // borrow x mutably
4      y.push_str(", handsome!");
5      // let z = &mut x; // cannot borrow x mutably twice
6      println!("{x}"); // x is still valid
7  }
```

On line: let z = &mut x;, an error is reported, indicating that there can be only one variable borrow. This is done to avoid data competition, such as writing data at the same time. More than one immutable reference can exist at the same time, because multiple immutable references do not affect the variable; only multiple mutable references cause errors.

Now look at another situation. If the reader buys an electronic version of this book, then you can give a copy to your friend by copying it (Please follow the intellectual property laws), so you both have a copy of the book, and both have ownership of the book. Extending this concept in Rust gives us ”copy”, ”clone”.

```
1 fn main() {
2     let x = "Shieber".to_string(); // create a string
3     let y = x.clone(); // clone x
4     println!("{x}, {y}"); // x and y both are valid
5 }
```

The clone function, as the word literally implies, makes a deep copy of the data to y. Borrowing is just getting a valid pointer, which is fast, while cloning requires copying the data, which is less efficient and doubles the memory consumption. If the reader tries to write the following without the clone function, and finds that it compiles and runs without errors, does it not satisfy the ownership rule?

```
1 fn main() {
2     let x = 10; // create an integer on the stack
3     let y = x;
4     println!("{x}, {y}"); // x and y both are valid. paradox?
5 }
```

In fact, let y = x; does not give 10 to y, but automatically copies a new 10 to y, so that x and y each have a 10, so there is no conflict with the ownership rule. Because these simple variables are on the stack, Rust implements a uniform trait for this kind of data called Copy, which allows for fast copying without releasing the old variable x. In Rust, values, booleans, characters, etc. all implement the Copy trait, so moving such variables is equivalent to copying. Here you can call clone as well, but it’s not necessary.

As mentioned earlier, a reference is a valid pointer. However, invalid pointers are often found, similar to the invalid dangling pointers in other programming languages.

```
1 fn dangle() -> &String {
2     let s = "Shieber".to_string();
3     &s
4 }
5
6 fn main() {
7     let ref_to_nothing = dangle();
8 }
```

The above code will compile with an error message similar to the following (with deletions).

```
error[E0106]: missing lifetime specifier
 --> dangle.rs:1:16
1 | fn dangle() -> &String {
  |                ^ expected named lifetime parameter
  | help: function's return type contains a borrowed value,
  | but there is no value for it to be borrowed from
  | help: consider using the 'static lifetime
1 | fn dangle() -> &'static String {
  |                ~~~~~~~~
```

In fact, the problem can be found by analyzing the code or the error message, the function returned an invalid reference.

```
fn dangle() -> &String { <--- return a reference to a String
    let s = "Shieber".to_string();
    &s  <---- return a reference to s
}       <---- s is dropped here, and it's memory goes away
```

According to the ownership analysis, s release is satisfied, &s is a pointer, returned, and seems to be fine, at most the pointer position is invalid. s and &s are two different things, the ownership system can only check the data according to the three rules, but it is impossible to know that the address &s points to is actually invalid. So why does the compilation error? The error message indicates that the lifetime specifier is missing. You can see that the suspension reference and the lifetime are in conflict, and that's why the error is reported. This means that even if your ownership system passes, but the lifetime does not, the error will be reported.

In fact, every reference in Rust has its own lifetime, i.e., the scope for which the reference remains valid. The ownership system does not guarantee that the data is absolutely valid, and requires a lifetime to ensure validity. Most of the time the lifetime is implicit and can be inferred, just as most of the time the type can be automatically inferred. When there is a reference in the scope, it is necessary to indicate the lifetime to show the interrelationship, so that the reference actually used at runtime is absolutely valid.

```
fn main() {
    let a;                  // ----------+'a, a lifetime begins
                            //           |
    {                       //           |
        let b = 10;         // --+'b     |    b lifetime begins
                            //   |       |
        a = &b;             //  -+       |    b lifetime ends
    }                       //           |
                            //           |
    println!("a: {}", a); // ----------+    a lifetime ends
}
```

a references b, and a's lifetime 'a is longer than b's lifetime 'b, so the compiler compiles with an error. For a to reference b properly, then b's lifetime must be at least as long as the end of a's lifetime. By comparing lifetimes, Rust can find references that don't make sense and thus avoid dangling reference problems.

To use variables legally, Rust requires that all data carry a lifetime specifier. Lifetimes are indicated by single quotes ' followed by a letter, such as &'a, &mut 't. References in functions also require lifetime markers.

```
fn longest<'a>(x: &'a String, y: &'a String) -> &'a String {
    if x.len() < y.len() {
        y
    } else {
        x
    }
}
```

The pointed brackets are lifecycle arguments, which are generic arguments that need to be declared in the pointed brackets of the function name and argument list to indicate that both arguments and the returned reference will live as long as they do. Because Rust automatically infers the lifetime, it is recommended to omit the lifetime specifier.

```
fn main() {
    // static lifetime, lives for the entire duration
    let s: &' static str = "Shieber";

    let x = 10;
    let y = &x; // can rewrite as let y = &'a x;
    println!("y: {}", y);
}
```

In this section, we learn about ownership system, borrowing, cloning, scope rules, lifetime. The ownership system is a memory management mechanism, borrowing and cloning are ways of using variables that extend the ownership system, and the lifetime is a complement to the ownership system, used to solve problems such as dangling references that the ownership system cannot handle.

1.4.7 Generic, Trait

To add two i64s, you can implement an add function of type i64 called add_i64.

```
fn add_i64(x: i64, y: i64) -> i64 {
    x + y
}
```

However, if you were to implement add functions for all numeric types, the code would become lengthy and the naming of functions would become cumbersome, leading to function names like add_i8, add_i16, and so on. The addition operation should be universal for any numeric type, and writing multiple sets of code is unnecessary. To tackle this problem, Rust uses generics. Generics are abstract substitutes for concrete types or properties, allowing developers to express the properties of the generic type without knowing its actual type when writing code.

```
// generic function: add
fn add<T>(x: T, y: T) -> T {
    x + y
}
```

The add function uses a generic type, and the generic parameter T is placed inside the parentheses immediately after add in the function declaration to indicate that this is a generic function. The generic parameter T represents any numeric type, and the return value type and the type of arguments involved in the function are both T, allowing it to handle all numeric types.

In addition to functions, generics can be used for other program components, such as enums. The Result in Rust ia an enum type, which handles errors, uses the generic parameters T and E, where T represents the return value type on success and E represents the return value type on error.

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Generic types can also be used for commonly used structs

```
struct Point<T> {
    x: T,
    y: T,
}

```

```
let point_i = Point { x: 3, y: 4 };
let point_f = Point { x: 2.1, y: 3.2 };
// let point_m = Point { x: 2_i32, y: 3.2_f32 }; Error
```

The generic parameter T is used to constrain the input parameters, ensuring that both parameters are of the same type. For instance, for the Point struct, it would be incorrect if x used i32 and y used f32.

Although a generic set of code has been implemented for the add function using generics, there are still some issues. The T parameter is not guaranteed to be a numeric type, so calling add on a type that does not support addition would be incorrect. It would be preferable to restrict the generic argument type of add to only numeric types. A trait in Rust is a way to define and restrict the behavior of a generic type, encapsulating the functionality shared by each type. Using traits, the behavior of each type can be well-controlled, and a trait can consist of three parts: method, type, and constant. It defines an abstract interface that can be implemented by a type or directly inherited from its default implementation. For example, if you have two types, teacher and student, and both have a greeting behavior, you can implement the following trait:

```
trait Greete {
    // default implementation
    fn say_hello(&self) {
        println!("Hello!");
    }
}

// contains different properties
struct Student {
    education: i32, // education years
}
struct Teacher {
    education: i32, // education years
    teaching: i32,  // teaching years
}

impl Greete for Student {}

impl Greete for Teacher {
    // override default implementation
    fn say_hello(&self) {
        println!("Hello, I am teacher Zhang!");
    }
}

// generic constraint
fn outer_say_hello<T: Greete>(t: &T) {
    t.say_hello();
}

fn main() {
    let s = Student{ education: 3 };
    s.say_hello();

    let t = Teacher{ education: 20, teaching: 2 };
    outer_say_hello(&t);
}
```

The outer_say_hello function above adds the generic constraint T: Greete, known as trait bound, indicating that only a type T that implements Greete can call the say_hello function. the previous add function can rewrite as below.

```
fn add<T: Addable>(x: T, y: T) -> T {
    x + y
}
```

The Addable trait is a constraint that ensures that only types implementing the Addable trait can be added together. This allows any type that doesn't meet the constraint to be reported as an error at compile time rather than waiting until runtime.

Another way to write the trait constraint is by using the impl keyword, as shown in the following example. Here, t must be a reference implementing the Greete trait.

```
fn outer_say_hello(t: &impl Greete) {
    t.say_hello();
}
```

Multiple trait bounds and multiple argument types can be combined using a comma and a plus sign.

```
fn some_func<T: trait1 + trait2, U: trait1> (x: T, y: U) {
    do_some_work();
}
```

To avoid using multiple trait bounds in parentheses, Rust has introduced a where syntax that separates the trait bound from the parentheses.

```
// where clause
fn some_func<T, U> (x: T, y: U)
    where T: trait1 + trait2,
          U: trait1,
{
    do_some_work();
}
```

The implementation of the say_hello method in Greete for Student and Teacher allows each type to encapsulate its own unique properties. While Student is more like a direct inheritance from Greete, Teacher overrides Greete. Although they use the same say_hello method, Student and Teacher display different states. The concepts of encapsulation, inheritance, and polymorphism are present here, which seems to imply that the concept of classes has been implemented through impl, trait, and thus object-oriented programming.

1.4.8 Enum and Match

When designing a questionnaire that requires multiple choice answers for gender, education, marital status, and other options, it is necessary to provide a variety of choices for participants to select. Enumerations in programming languages allow you to define a type by listing all possible members, similar to defining a structure. For instance, you can use an enumeration to represent gender.

```
enum Gender {
    Male,
    Female,
    TransGender,
}
```

To use an enumeration in Rust, simply use ”enum type::enum name”.

```
let male = Gender::Male;
```

A common enumeration type in Rust is Option, where Some represents a value and None indicates no value.

```
enum Option<T> {
    Some(T),
    None
}
```

Enumerations can also be combined with match for process control. A match expression compares a value to a series of patterns and executes the corresponding code based on the matching pattern. The patterns can include literal values, variables, wildcards, and other content. The matched values pass through each pattern of the match, and when the first matching pattern is encountered, it enters the associated block of code. Match expressions are like a classifier or sieve that filters values based on patterns. In Rust, the matched values can be filtered into more fine-grained classes. For example, an enumeration could represent the different denominations of currency values.

```
enum Cash { // US dollar($)
    One,
    Two,
    Five,
    Ten,
    Twenty,
    Fifty,
    Hundred,
}

fn cash_value(cash: Cash) -> u8 {
    match cash {
        Cash::One => 1,
        Cash::Two => 2,
        Cash::Five => 5,
        Cash::Ten => 10,
        Cash::Twenty => 20,
        Cash::Fifty => 50,
        Cash::Hundred => 100,
    }
}
```

Additionally, match supports wildcard and _ placeholder matches.

```
match cash {
    Cash::One => 1,
    Cash::Two => 2,
    Cash::Five => 5,
    Cash::Ten => 10,
    other => 0, // _ => 0,
}
```

For instance, ”other” can be replaced with _ if you don’t need to use the value directly. Here other is used instead of all denominations greater than 10.

The match expression must be exhaustive, which means it should cover all possible matching patterns. However, if you only need to match a specific pattern, using match can be cumbersome. Fortunately, Rust generalizes the matching pattern of match and introduces the if let matching pattern. The if let match counts all bills larger than $1.

```
let mut greater_than_one = 0;
if let Cash::One = cash { // only match Cash::One
    println!("cash is one");
} else {
    greater_than_one += 1;
}
```

1.4.9 Functional Programming

Rust prioritizes functional programming instead of class-based problem-solving methods used in object-oriented languages. This programming paradigm involves using functions as parameter values or returning them as the result of other functions. The key components of functional programming are closures and iterators.

Closures are anonymous functions that can be passed as parameters to other functions and capture variables. They can be created in one location and executed in different contexts. Closures allow the capturing of values in the caller's scope, making them particularly useful for defining functions that are only used once.

```
// define a function
fn function_name(parameters) -> return_types {
    code_body;
    return_value
}

// define closure
|parameters| {
    cody_body;
    return_value
}
```

Closures can be parameterless and may or may not return values. Rust infers closure parameter and return value types, eliminating the need to specify them explicitly. To use a closure, it must be assigned to a variable and called like a function. For instance, a closure function that determines odd or even numbers can be defined.

```
let is_even = |x| { 0 == x % 2 };

let num = 10;
println!("{num} is even: {}", is_even(num));
```

It's also possible to use external variables in closures.

```
let val = 2;
let add_val = |x| { x + val };

let num = 2;
let res = add_val(num);
println!("{num} + {val} = {res}")
```

The add_val function captures the external variable val. Closures can capture external variables by taking ownership, reference, or mutable reference, which Rust defines using three function traits: FnOnce, FnMut, Fn.

• The FnOnce trait consumes the captured variable from the surrounding scope and moves it into the closure when it is defined. This trait indicates that the closure can only be called once.

• The FnMut trait obtains a mutable borrowed value, allowing it to change the external variable.

• The Fn trait obtains an immutable borrowed value from its environment.

These traits correspond to the move, mutable reference, and reference implementation of the ownership system. All closures can be called at least once, making them implement the FnOnce trait. Closures that use mutable references but do not move variable ownership into the closure implement the FnMut trait, while those that do not require mutable access to the variables implement Fn.

The move keyword is used to force the ownership of an external variable to be moved into a closure.

```
let val = 2;
let add_val = move |x| { x + val };
// println!("{val}"); error: use of moved value: val
```

In Rust, the ”for in” loop is used to iterate through an array, which is an iterator that implements the iteration functionality by default. Iterators pass each element in a collection to a processing logic in sequence, allowing specific operations to be performed on the items in a sequence. They are responsible for iterating through each item in the sequence and determining when to stop.

By default, iterators must implement the Iterator trait, which has two methods: iter() and next().

```
iter(), return a iterator
next(), return the next element
```

The iter() method can be divided into three types depending on whether the data can be modified during iteration.

```
method              return type
iter()              return a immutable iterator
                    whose element type is &T
iter_mut()          return a mutable iterator
                    whose element type is &mut T
into_iter()         return a immutable iterator
                    whose element type is T, original data
                        consummed
```

In Rust, iterators can be classified into ”reentrant” and ”non-reentrant” types based on their behavior towards the original data after iteration. A ”reentrant” iterator allows the original data to be used again after iteration, while a ”non-reentrant” iterator consumes the original data. The iter(), iter_mut(), and into_iter() methods are used to implement iterators for reading values, changing values, and taking ownership of the original data, respectively.

```
let nums = vec![1,2,3,4,5,6];

// 1.iter immutable
for num in nums.iter() {
    println!("num: {num}")
};
println!("{:?}", nums); // use nums after iter

// 2.iter_mut mutable
for num in nums.iter_mut() {
```

```
    *num += 1;
}
println!("{:?}", nums); // use nums after iter_mut

// 3.into_iter transfer nums into iterator and consume nums
for num in nums.into_iter() {
    println!("num: {num}");
}
// println!("{:?}", nums); error: use of moved value "nums"
```

In addition to transferring ownership of the original data, iterators can also be consumed or regenerated. Consumers are special operations on an iterator that convert the iterator into a value of another type. Examples of consumers include sum, collect, nth, find, next, and fold. They perform operations on the iterator to obtain the final value. Producers, on the other hand, are adapters that traverse the iterator and generate another iterator. Examples of adapters include take, skip, rev, filter, map, zip, and enumerate. In Rust, an iterator itself is considered an adapter.

```
// adapter_consumer.rs

fn main() {
    let nums = vec![1,2,3,4,5,6];
    let nums_iter = nums.iter();
    let total = nums_iter.sum::<i32>(); // sum is a consumer

    let new_nums: Vec<i32> = (0..100).filter(|&n| 0 == n % 2)
                                     .collect(); // Adapter
    println!("{:?}", new_nums);

    // calculate the sum of all numbers less than 1000
    // that are divisible by 3 or 5
    let sum = (1..1000).filter(|n| n % 3 == 0 || n % 5 == 0)
                       .sum::<u32>();
                       // conbine adapter and consumer
    println!("{sum}");
}
```

The following code combines adapters, closures, and consumers to find the sum of all integers less than 1000 that are divisible by 3 or 5, showcasing the power of functional programming. This approach allows for complex calculations to be performed in a concise and efficient manner. In comparison, imperative programming may result in lengthy and difficult-to-understand code. Therefore, it is recommended to use closures with iterators, adapters, and consumers for functional programming.

```
fn main() {
    let mut nums: Vec<u32> = Vec::new();
    for i in 1..1000 {
        if i % 3 == 0 || i % 5 == 0 {
            nums.push(i);
        }
    }

    let sum = nums.iter().sum::<u32>();
    println!("{sum}");
}
```

Functional programming is a programming paradigm that is distinct from other paradigms such as imperative programming and declarative programming. Imperative programming is an abstraction of computer hardware, involving variables, assignment statements, expressions, control statements, etc. Structured programming and object-oriented programming, which are commonly used, fall under this paradigm. The main focus is on the steps that the computer executes, instructing it on what to do step by step.

In contrast, declarative programming expresses program logic as data structures, concentrating on what to do rather than how to do it. SQL statements are an example of declarative programming. While functional programming and declarative programming share a common idea of focusing on what to do, functional programming is not limited to declarative programming. It is an abstraction of mathematics, describing computation as the evaluation of expressions. In other words, a functional program is a mathematical expression. The functions in functional programming refer to mathematical functions, which map the independent variable to the dependent variable $y = f(x)$. The output of a function solely depends on the input parameter value, not on any other state. For instance, the sin() function in mathematics calculates the sine value of x. As long as x remains unchanged, the final result will always be the same, no matter how or when it is called.

1.4.10 Smart Pointer

Rust, being a systems language, still requires the use of pointers. Pointers refer to or point to other data by containing a memory address. The most common pointer in Rust is a reference, which is similar to C's concept of pointers but has no other functionality besides referencing data. On the other hand, smart pointers are data structures that behave like pointers and contain metadata. They provide additional functions such as memory management or binding checks, like managing file handles and network connections. In Rust, Vec and String can be considered smart pointers.

Smart pointers were first introduced in C++ and later adopted by Rust. Rust encapsulates two major traits for smart pointers: Deref and Drop. When a variable implements Deref and Drop, it becomes more than an ordinary variable. After implementing Deref, the variable overloads the dereference operator "*", which can be used as an ordinary reference and can be automatically or manually dereferenced when necessary. After implementing Drop, the variable is automatically released from the heap when it goes out of scope. Other functions can also be customized, such as releasing files or network connections, similar to some languages' destructor functions. The following are the main features of smart pointers:

1. Smart pointers usually own the data they point to.
2. Smart pointers are data structures that are typically implemented using structures.
3. Smart pointers implement two major traits: Deref and Drop.

Many common smart pointers are available, and users can also implement the smart pointers they need. The following are frequently used smart pointers or data structures. Note that although Cell and RefCell share similarities with smart pointers, they are not considered smart pointers according to the above characteristics.

• Box<T>: A unique ownership smart pointer that points to data of type T stored on the heap.

• Rc<T>: A reference counting smart pointer that shares ownership and is used to track the number of references to a value stored on the heap.

• Arc<T>: A thread-safe reference counting smart pointer that shares ownership and can be used for multi-threading.

• Cell<T>: A container that provides interior mutability and is not a smart pointer. It allows for borrowed mutable data with compile-time checks, requiring T to implement the Copy trait.

• RefCell<T>: A container that provides interior mutability and is not a smart pointer. It allows for borrowed mutable data with run-time checks, with no requirement for T to implement the Copy trait.

• Weak<T>: A weak reference type that corresponds to Rc and is used to solve the problem of circular references in RefCell.

• Cow<T>: A copy-on-write enumerated smart pointer that is primarily used to reduce memory allocation and copying. It is suitable for scenarios with frequent reads and infrequent writes.

In Rust, Deref and Drop are the two most important traits for smart pointers. By implementing Deref and Drop for a custom data type similar to Box, we can better understand the difference between references and smart pointers.

```
// define tuple struct
struct SBox<T>(T);
impl<T> SBox<T> {
    fn new(x: T) -> Self {
        Self(x)
    }
}

fn main() {
    let x = 10;
    let y = SBox::new(x);
    println!("x = {x}");
    // println!("y = {}", *y); cannot dereference a raw pointer
} <--- x, y call drop
```

Here is an example of implementation of Deref and Drop for SBox.

```
use std::ops::Deref;

// implement Deref for SBox
impl<T> Deref for SBox<T> {
    type Target = T; // define associated type,
                     // which is the type that Deref is
                        targeting to
    fn deref(&sefl) -> &Self::Target {
        &self.0 // .0 means the first element of tuple struct
    }
}

// implement Drop for SBox
impl<T> Drop for SBox<T> {
    fn drop(&mut self) {
        println("SBox drop itself!"); // just print a message
    }
}
```

If Deref is not implemented, a variable of our custom data type cannot be dereferenced. On the other hand, if Drop is not implemented, the variable will be released from the heap only when it goes out of scope. However, if the variable contains other references or smart pointers, they will not be released until the program exits, resulting in a memory leak. Therefore, it is crucial to implement Drop correctly to avoid memory leaks.

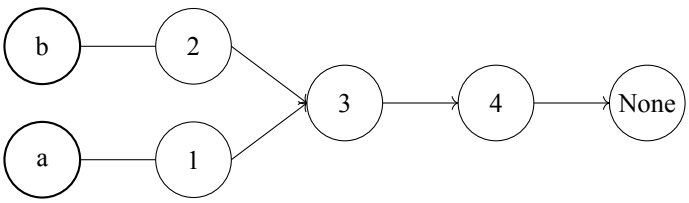
```
fn main() {
    let x = 10;
    let y = SBox::new(x);
    println!("x = {x}");
    println!("y = {}", *y); // *y equals *(y.deref())
    // y.drop(); error: drop twice
} <--- x, y call drop and y prints "SBox drop itself!"
```

Box stores its data on the heap, and after implementing Deref, it can be automatically dereferenced.

```
fn main() {
    let num = 10; // num stored in stack
    let n_box = Box::new(num); // n_box stored in heap
    println!("n_box = {}", n_box); // n_box deref to num
                                   // automatically
    println!("{}", 10 == *n_box);  // n_box deref to num
                                   // manually
}
```

The ownership system rules state that a value can only have one owner at any given time, but in some scenarios, we need values to have multiple owners. To address this, Rust provides the Rc smart pointer. Rc is a shareable reference counting smart pointer that can produce multiple ownership values. Reference counting means that it determines whether a value is still in use by keeping track of the number of references to it, and if the reference count is zero, the value can be cleaned up.

Rust's ownership system mandates that a value can have only one owner at any given time. However, in some cases, multiple ownership is necessary. To solve this problem, Rust provides the Rc smart pointer, which is a reference counting smart pointer that can produce multiple ownership values. It tracks whether a value is still in use by counting the number of references to it. Once the reference count drops to zero, the value can be cleaned up.



In the example shown, 3 is shared by variables a(1) and b(2), and cloning an Rc increases the reference count. Sharing is like turning off the lights in a classroom, only the last person leaving needs to turn off the lights. Similarly, each user of Rc will only clear the data on the last use. Cloning an Rc will increase the reference count, just like when a new person enters a classroom.

```
use std::rc::Rc;
fn main() {
    let one = Rc::new(1);
    // increase reference count
    let one_1 = one.clone();
    // display reference count
    println!("sc:{}", Rc::strong_count(one_1));
}
```

Rc can be used for shared ownership in single-threaded environments only. For multithreaded environments, Rust provides Arc (atomic reference counting), a thread-safe version of Rc. It allocates a T type value with shared ownership on the heap. Cloning an Arc increases the reference count while producing a new Arc instance pointing to the same heap as the source Arc. Arc is immutable by default and requires locking mechanisms like Mutex to modify it between multiple threads.

By default, Rc and Arc do not allow internal data modification. Rust provides two containers with internal mutability, Cell and RefCell, for scenarios where modifying internal values is necessary. Internal mutability is a Rust design pattern that allows data modification while having immutable references. Cell provides methods for getting and changing internal values. For data types that implement Copy, the get method is used to view the internal value, while the take method replaces the internal value with the

default value and returns the replaced value. For all data types, the replace method replaces and returns the replaced value, while the into_inner method consumes Cell and returns the internal value.

```
// use_cell.rs

use std::cell::Cell;

struct Fields {
    regular_field: u8,
    special_field: Cell<u8>,
}

fn main() {
    let fields = Fields {
        regular_field: 0,
        special_field: Cell::new(1),
    };

    let value = 10;
    // fields.regular_field = value;
    // error[E0594]: cannot assign to immutable field

    fields.special_field.set(value);
    // although fields is immutable,
    // we can still change the value of special_field

    println!("special: {}", fields.special_field.get());
}
```

To enable modification of certain fields internally, Cell provides a backdoor to immutable struct Fields.

RefCell has similar characteristics to Cell, but instead of using get and set methods to modify internal data, it directly obtains a mutable reference to the data. For example, RefMut<_> can be used to modify HashMap through RefCell. Here, shared_map obtains a RefMut<_> type map directly through borrow_mut(), and then adds an element to the map through insert, which modifies shared_map.

```
// use_refcell.rs

use std::cell::{RefCell, RefMut};
use std::collections::HashMap;
use std::rc::Rc;

fn main() {
    let shared_map: Rc<RefCell<_>> =
        Rc::new(RefCell::new(HashMap::new()));
    {
        let mut map: RefMut<_> = shared_map.borrow_mut();
        map.insert("kew", 1);
        map.insert("shieber", 2);
        map.insert("mon", 3);
        map.insert("hon", 4);
    }

```

```
18      let total: i32 = shared_map.borrow().values().sum();
19      println!("{}", total);
20  }
```

While both Cell and RefCell can modify internal values, they do so differently. Cell replaces values directly, while RefCell modifies them through mutable references. Since Cell needs to replace and move values, it must implement Copy, while RefCell is suitable for data types that do not implement Copy. Cell is checked at compile time, while RefCell is checked at runtime, and misuse can lead to panics.

Although Rust provides memory safety guarantees, RefCell may still cause circular references and memory leaks. To prevent this, Rust provides the Weak smart pointer, which allows for the value to be cleared even when there are circular references. The upgrade method can be used to check if the referenced value is still valid. Each time Rc is cloned, the strong reference count strong_count of the instance is increased, and the instance is only cleared when strong_count is 0. strong_count will never be 0 in circular references. When using the Weak smart pointer, the value of strong_count is not increased, but the value of weak_count does. weak_count can be cleared without being 0, which solves the problem of circular references.

As Weak smart pointer, to ensure that the referenced value is still valid, its upgrade method can be called, which returns Option<Rc<T>». If the Rc<T> value has not been dropped, the result is Some. If dropped, the result is None. The following is an example of using Weak to solve circular references. Car and Wheel have mutual references, and circular references will occur if they are both used with Rc.

```
1   // use_weak.rs
2
3   use std::cell::RefCell;
4   use std::rc::{Rc, Weak};
5
6   struct Car {
7       name: String,
8       wheels: RefCell<Vec<Weak<Wheel>>>, // refence to Wheel
9   }
10
11  struct Wheel {
12      id: i32,
13      car: Rc<Car>, // reference to Car
14  }
15
16  fn main() {
17      let car: Rc<Car> = Rc::new(
18          Car {
19              name: "Tesla".to_string(),
20              wheels: RefCell::new(vec![]),
21          }
22      );
23
24      let wl1 = Rc::new(Wheel { id:1, car: Rc::clone(&car) });
25      let wl2 = Rc::new(Wheel { id:2, car: Rc::clone(&car) });
26
27      let mut wheels = car.wheels.borrow_mut();
28
29      // downgrade to Weak
30      wheels.push(Rc::downgrade(&wl1));
31      wheels.push(Rc::downgrade(&wl2));
32
```

```
33      for wheel_weak in car.wheels.borrow().iter() {
34          let wl = wheel_weak.upgrade().unwrap(); // Option
35          println!("wheel {} owned by {}", wl.id, wl.car.name);
36      }
37  }
```

Now, let's take a look at Cow (Copy on Write) Smart Pointer.

```
1  pub enum Cow<'a, B>
2      where B: 'a + ToOwned + 'a + ?Sized {
3      Borrowed(&'a B), // wrap a reference
4      Owned(<B as ToOwned>::Owned), // wrap a onwed value
5  }
```

The concept of borrowing involves accessing borrowed content in an immutable way, while ownership means cloning the data when mutable borrowing is needed. To filter out all spaces in a string, we can use the following code.

```
1  // use_cow.rs
2  fn delete_spaces(src: &str) -> String {
3      let mut dest = String::with_capacity(src.len());
4      for c in src.chars() {
5          if ' ' != c {
6              dest.push(c);
7          }
8      }
9      dest
10 }
```

While this code works, its efficiency is not optimal. First, we need to decide whether to use &str or String for the parameter. If we use String but input &str, we need to clone it before calling the function. If we use String, the string will be moved to the interior after the call, and the external cannot use it anymore. In either case, a string generation and copy are performed in the function. If the string does not contain whitespace characters, it is best to return it as is without copying. This is where Cow comes in handy as it reduces copying and improves efficiency.

```
1  // use_cow.rs
2  use std::borrow::Cow;
3  fn delete_spaces2<'a>(src: &'a str) -> Cow<'a, str> {
4      if src.contains(' ') {
5          let mut dest = String::with_capacity(src.len());
6          for c in src.chars() {
7              if ' ' != c { dest.push(c); }
8          }
9          // capture the ownership and drop dest
10         return Cow::Owned(dest);
11     }
12     return Cow::Borrowed(src); // borrow the src
13 }
```

Here is an example that utilizes Cow.

```
1  // use_cow.rs
2  fn main() {
3      let s = "i love you";
```

```
4       let res1 = delete_spaces(s);
5       let res2 = delete_spaces2(s);
6       println!("{res1}, {res2}");
7   }
```

When there are no whitespace characters, the function returns directly, avoiding copying and improving efficiency.

```
1   // more_cow_usage.rs
2
3   use std::borrow::Cow;
4   fn abs_all(input: &mut Cow<[i32]>) {
5       for i in 0..input.len() {
6           let v = input[i];
7           if v < 0 { input.to_mut()[i] = -v; }
8       }
9   }
10  fn main() {
11      // read only, no write, no copy
12      let a = [0, 1, 2];
13      let mut input = Cow::from(&a[..]);
14      abs_all(&mut input);
15      assert_eq!(input, Cow::Borrowed(a.as_ref()));
16      // copy on write, copy when read -1
17      let b = [0, -1, -2];
18      let mut input = Cow::from(&b[..]);
19      abs_all(&mut input);
20      assert_eq!(input, Cow::Owned(vec![0,1,2]) as Cow<[i32]>);
21      // no copy on write, because already owned
22      let mut c = Cow::from(vec![0, -1, -2]);
23      abs_all(&mut c);
24      assert_eq!(c, Cow::Owned(vec![0,1,2]) as Cow<[i32]>);
25  }
```

This section contains a lot of diverse content, and readers are encouraged to study it further in conjunction with other dedicated books. The information provided here is only brief and not deeply discussed.

1.4.11 Exception

Rust provides a unique exception handling mechanism as it does not have try-catch blocks like other languages. This section is titled ”Exceptions,” but the author has used the term to refer to failures, errors, and exceptions collectively. There are four types of exceptions in Rust: Option, Result, Panic, and Abort.

Option is used to handle possible failure cases, indicating whether the operation succeeded or failed with Some and None. For instance, when getting a value that may not exist, the result may be None, which should not cause an error but needs to be handled appropriately. Failure is different from error, as it is an anticipated outcome that will not cause problems for the program. The definition of Option, which we have already learned, is provided below.

```
1   enum Option<T> {
2       Some(T),
3       None,
4   }
```

Result is used to handle recoverable errors and represents success and failure. An error may not necessarily cause the program to crash, but it needs to be specifically handled to allow the program to continue executing. The definition of Result is as follows:

```
enum Result<T,E> {
    Ok(T),
    Err(E),
}
```

Opening a file that does not exist, accessing a file without permission, or attempting to convert a non-numeric string to a number will result in Err(E).

```
use std::fs::File;
use std::io::ErrorKind;

let f = File::open("kw.txt");
let f = match f {
    Ok(file) => file,
    Err(err) => match err.kind() {
    ErrorKind::NotFound => match File::create("kw.txt"){
        Ok(fc) => fc,
        Err(e) => panic("Error while creating file!"),
    }
    ErrorKind::PermissionDenied => panic("No permission!"),
    other => panic!("Error while openning file"),
}
```

In Rust, handling errors is crucial, and there are various mechanisms to achieve it. One of them is the panic mechanism, which should only be used when no other solution is available. When encountering an unrecoverable error, panic will immediately stop the program's execution to allow the programmer to identify the problem. This mechanism is also useful for cleaning up memory. However, if you do not want to use panic to clean up when encountering such errors, you can use the abort mechanism to let the operating system handle it instead.

Although error handling in Rust may look verbose when using match, there are more concise options available. For example, you can use the unwrap or expect methods to handle errors.

```
use std::fs::File;
use std::io;
use std::io::Read;

fn main() {
   let f = File::open("kew.txt").unwrap();
   let f = File::open("mon.txt").expect("Open file failed!");
   // expect gives more specific error message than unwrap
}
```

If you only want to propagate the error and not handle it, you can use the "?" operator. In this case, the return type is Result, which returns String on success and io::Error on failure.

```
fn main() {
    let s = read_from_file();
    match s {
        Err(e) => println!("{}", e),
        Ok(s) => println!("{}", s),
    }
```

```
}

fn read_from_file() -> Result<String, io::Error> {
    let f = File::open("kew.txt")?; // if error, throw it
    let mut s = String::new();
    f.read_to_string(&mut s);

    Ok(s)
}
```

When working with errors in Rust, you have to choose between using Option or Result depending on the type of failure or error you are handling. Option is used for handling possible failures and represents success or failure with Some or None. On the other hand, Result is used for handling recoverable errors and represents success and failure with Ok or Err. It is essential to handle errors appropriately to avoid program crashes. Finally, note that Option and Result are very similar, as Option<T> can be seen as Result<T, ()>.

1.4.12 Macro

Rust does not have built-in library functions, meaning that programmers must define everything themselves. However, Rust offers a set of powerful macros, including declarative and procedural macros, that can accomplish many tasks. Unlike C macros, Rust macros are more powerful and widely used. For instance, using the ”derive” macro, new functionality can be added to structures, and commonly used methods like ”println!”, ”vec!”, and ”panic!” are also macros.

There are two main categories of macros in Rust: declarative macros declared using ”macro_rules!” and procedural macros, further divided into custom derive macros, function-like macros, and attribute macros. We’ve already covered declarative and custom derive macros, so this section will still focus on those. Readers can refer to related materials for other macro types.

The format of a declarative macro is ”macro_name!()”, ”macro_name![]”, or ”macro_name!”. These macros all use a macro name followed by an exclamation mark, which is of one of these bracket types (), [], {}. Different macro purposes require different brackets, with ”vec![]” not being the same as ”vec!()”. The latter looks more like a function and is the reason ”println!()” uses parentheses. However, any of these brackets can be used and different brackets are just to satisfy the unity of meaning and form.

```
macro_rules! macro_name {
    ($matcher) => {
        $code_body;
        return_value
    };
}
```

In the macro definition, ”$matcher” marks syntax elements like empty, identifiers, literals, keywords, symbols, patterns, and regular expressions. The ”$” symbol captures values, which are then used in ”$code_body” to process and potentially return a value. For example, the following macro can be used to calculate the left and right child nodes of a binary tree parent node ”p”, with the left and right child node indices being ”2p” and ”2p+1”, respectively.

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

```
7       ($parent:ident) => {
8           ($parent << 1) + 1
9       };
10  }
```

When calculating left and right child nodes, macros such as ”left_child!(p)” and ”right_child!(p)” can be used instead of expressions like ”2 * p” and ”2 * p + 1”. This simplifies the code and clarifies its meaning. The ”indent” and colon in the macro are metavariable specifiers. ”Indent” is an attribute of ”parent”, indicating that ”parent” is a value. Rust’s macros rely on these specifiers to function correctly.

The second form of macros is called procedural macros because they are more like a process or function. They take code as input and produce other code as output. Derive macros, a type of procedural macro, add new functionality to code directly. The following example uses the ”derive” macro to add the ”Clone” method to ”Student”, so that ”Student” can directly call ”clone()” to achieve copying. This is a simplified writing of ”impl Clone for Student”.

```
1  #[derive(Clone)]
2  struct Student;
```

Multiple traits can also be placed in ”derive” simultaneously, allowing for implementation of various functions.

```
1  #[derive(Debug, Clone, Copy)]
2  struct Student;
```

Macros are a complex topic and should only be used when necessary and when they simplify the code. It’s not recommended to use macros for their own sake.

1.4.13 Code Organization and Dependency

In Rust, there are several concepts to understand for organizing code, including packages, crates, libraries, and modules. When using the ”cargo new” command, Rust generates a package that contains multiple directories, each of which is a crate. After compilation, a crate can either be a binary executable or a library that can be used by other functions. Within a crate, there are typically multiple .rs files, which are called modules and are accessed using the ”use” keyword. The table below provides a detailed overview of these concepts and their corresponding terms in Rust.

```
package --> crates  (dirs)     a package have multiple crates
crate   --> modules (lib/EFL)  a crate have multiple modules
                               which can be compiled into a
                               library or executable file
module  --> file.rs (file)     a module can contain one or
                               more .rs files
package             <-- package name
|-- Cargo.toml
|-- src             <-- crate
|    |-- main.rs    <-- main module
|    |-- lib.rs     <-- module, library module
|    |-- math       <-- module, math module
|        |- mod.rs  <-- module, import add/sub from modules
|        |- add.rs  <-- module, implement add of math module
|        |- sub.rs  <-- module, implement sub of math module
|-- file            <-- crate
     |-- core       <-- module, file operation module
     |-- clear      <-- module, clear module
```

To organize a large project, it is recommended to follow this approach when using Rust libraries. The Tikv codebase is a good example to follow.

Rust also offers many standard libraries for solving general programming tasks. By studying these libraries, developers can deepen their understanding of Rust and find inspiration for solving real-world problems. The standard libraries in Rust include:

```
alloc         env       i64       pin         task
any           error     i128      prelude     thread
array         f32       io        primitive   time
ascii         f64       isize     process     u8
borrow        ffi       iter      raw         u16
boxed         fmt       marker    mem         u32
cell          fs        net       ptr         u64
char          future    num       rc          u128
clone         hash      ops       result      usize
cmp           hint      option    slice       vec
collections   i8        os        str         backtrace
convert       i16       panic     string      intrinsics
default       i32       path      sync        lazy
```

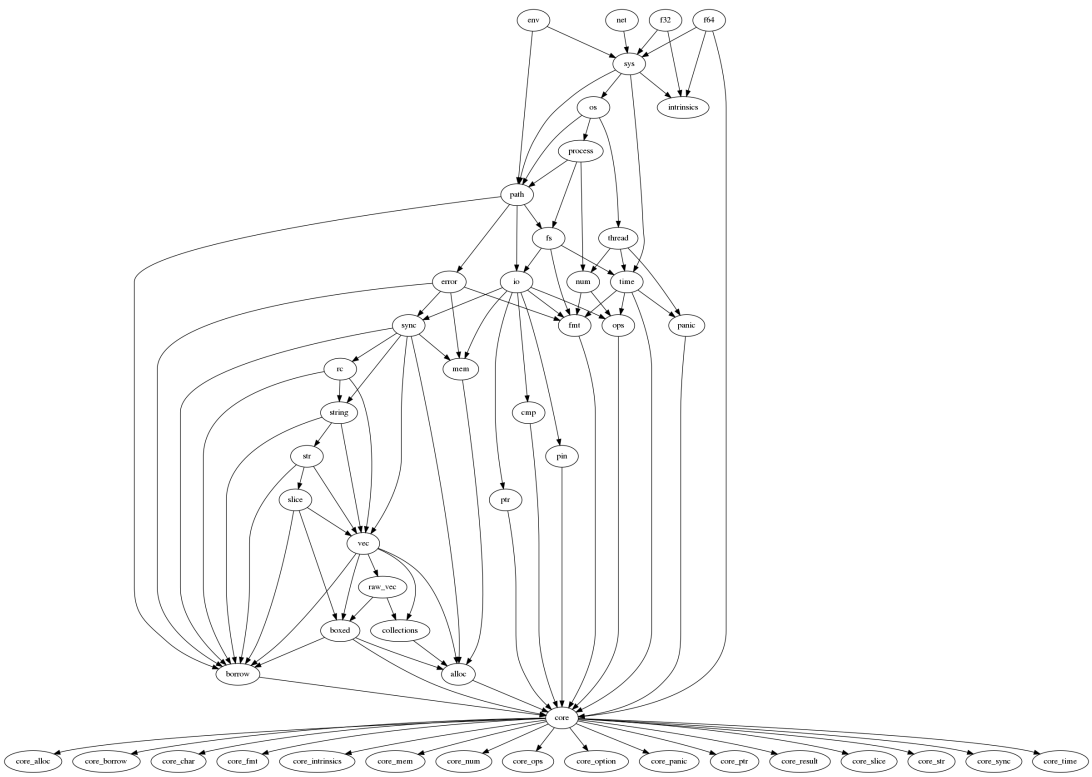


Figure 1.1: Rust package dependency

The image shown here is a simplified dependency graph of Rust libraries. It was created to illustrate how Rust as a language is built and how the language's libraries are interrelated. These libraries are considered the de facto standards of Rust, with some being fundamental and serving as dependencies for other libraries.

An analysis of Rust's standard libraries revealed that they can be divided into three layers: core, alloc, and std. Each layer builds upon the previous one, with std being the top layer, alloc in the middle, and core serving as the most critical layer. Core is responsible for defining and implementing various core concepts in Rust's basic syntax, such as variables, numbers, characters, and Boolean types. These are the foundational concepts that beginners learn when studying Rust's basics.

1.4.14 Project: a password generator

In the previous sections, we covered many basic Rust concepts. Now, let's apply them to a comprehensive project. In this project, readers will learn how to organize Rust project code, import modules, use various annotations, write tests, and build command-line tools.

In the age of the internet, people have numerous accounts and passwords to remember. Unfortunately, it's challenging to remember unique passwords for each account, and people often set similar or identical passwords, leaving themselves vulnerable to hacking. Although password management software can help, relying on someone else's software to manage your own passwords can make you uneasy. To solve this problem, the author created PasswdGenerator, a command-line tool that generates passwords that are at least 16 characters long and appear like random strings, such as 9KbAM4QWMCcyXAar.

PasswdGenerator can generate complex passwords of default length 16 by controlling the seed parameter, and the length parameter can adjust password length. This tool makes generating different passwords for all accounts easy, allowing users to input a digit or character that only they know to generate complex passwords. Using PasswdGenerator, the author generated different passwords for all his accounts and, because he thought of the seed himselve, he didn't have to worry about forgetting it. PasswdGenerator has been used for several years, and the author doesn't even know his passwords for WeChat, Alipay, Twitter, Youtube, Github accounts etc.

PasswdGenerator is a command-line tool, and its usage method is similar to that of command-line tools in Unix-like systems. Therefore, its first usage is "PasswdGenerator -h" or "PasswdGenerator –help".

```
shieber@kew $ PasswdGenerator -h
PasswdGenerator 0.1.0
A simple password generator for any account

USAGE:
    PasswdGenerator [OPTIONS] --seed <SEED>

OPTIONS:
    -h, --help               Prints help information
    -l, --length <LENGTH>    Length of password [default: 16]
    -s, --seed <SEED>        Seed to generate password
    -V, --version            Prints version information
```

The USAGE section provides instructions on how to use the tool, while the OPTIONS section displays the control parameters.

```
shieber@kew $ PasswdGenerator -s wechat
wechat: GQnaoXobRwrgW21A
```

The output of the PasswdGenerator tool begins with a short seed, such as "wechat," which can be customized with prefixes or suffixes to associate it with a specific account. The seed serves to indicate that the password was generated specifically for that account.

```
shieber@kew $ PasswdGenerator -s wechat -l 20
wechat: GQnaoXobRwrgW21Ac2Pb
```

The length of the password is controlled by the ”-l” parameter. If this parameter is not included, the password defaults to a length of 16, which is sufficient for all accounts.

To handle command-line arguments, the clap library can be used, as shown below.

```
use clap::Parser;

/// A simple password generator for any account
#[derive(Parser, Debug)]
#[clap(version, about, long_about= None)]
struct Args {
    /// Seed to generate password
    #[clap(short, long)]
    seed: String,

    /// Length of password
    #[clap(short, long, default_value_t = 16)]
    length: usize,
}
```

The ”seed” and ”length” are the two command-line arguments, including their short and long forms, with ”length” having a default value of 16. The command-line arguments can be obtained by calling Args::parse().

Once the command-line arguments are obtained, the next step is to generate the password. My approach was to use a combination of a hash algorithm and base64 encoding. While there are many types of hash algorithms and various libraries available for generating hashes, it is practical to write a hash algorithm oneself. Here, I chose to use Mersenne primes to generate hash values.

To generate the hash value, the ASCII value of each character in the seed is multiplied by its corresponding index value, and the result is taken modulo the Mersenne prime 127. The remainder is then raised to the third power to obtain the hash value of the seed.

Mersenne primes are prime numbers of the form $M_n = 2^n - 1$, where n is also a prime number. While $2^n - 1$ may not always be a prime number, as in the case of primes such as 11, 23, and 29, Mersenne primes have the property that both n and M_n must be prime, making them suitable for calculating the hash value here. Some examples of Mersenne primes include 3, 7, 31, and 127 are shown below, in which 127 is the fourth Mersenne prime.

No	n	Mn
1	2	3
2	3	7
3	5	31
4	7	127
5	13	8191
6	17	131071
7	19	524287
8	31	2147483647
9	61	2305843009213693951
10	89	618970019642690137449562111

One method to convert a hash value to a string is by character replacement. The hash value is first modulo-ed with a fixed-length string (password), which results in a remainder used as an index to obtain a character. Repeating this process cyclically generates a string of characters that can be concatenated to form the password. For more complexity, the seed can be split and concatenated with the generated password.

```
// conbine seed and passwd

let interval = passwd.clone();
for c in seed.chars() {
    passwd.push(c);
    passwd += &interval;
}
```

To make the password suitable for use, any unsuitable characters in the password string can be converted to Base64 encoding, which uses 64 visible characters. Additionally, ”+” and ”/” in Base64 can be replaced with ”*” in the password.

```
// encode passwd (base64)
passwd = encode(passwd);
passwd = passwd.replace("+", "*").replace("/", "*");
```

If the password is still not long enough, it can be cyclically written into itself and wrapped with the seed and password using format, then returned.

```
let interval = passwd.clone();
while passwd.len() < length {
    passwd += &interval;
}

format!("{}: {}", seed, &passwd[..length])
```

The tool uses three functions: ”hash” crate to calculate the Mersenne hash value, ”encryptor” crate to generate the password, and the main module to obtain command-line arguments. The main module calls the ”generate_password” function from ”encryptor” crate, which in turn calls ”mersenne_hash” function from ”hash” crate to calculate the hash value.

To organize the code, we can use Cargo’s workspace mechanism to generate corresponding crates by specifying their names. The first step is to create and enter a directory.

```
shieber@kew $ mkdir PasswdGenerator && cd PasswdGenerator
```

Then, we need to write a ”Cargo.toml” file that defines the names of the crates we want to generate.

```
1 [workspace]
2 members = [
3     "hash",
4     "encryptor",
5     "main",
6 ]
```

After that, we can use ”cargo” to generate the three crates. Note that different parameters should be used to generate the library and the main program.

```
shieber@kew $ cargo new hash --lib
    Created library hash package
shieber@kew $ cargo new encryptor --lib
    Created library encryptor package
shieber@kew $ cargo new main
    Created binary (application) main package
```

Once generated, the directory structure should look like the following table.

```
.
|-- Cargo.toml
|-- encryptor
|   |-- Cargo.toml
|   |-- src
|       |-- lib.rs
|-- hash
|   |-- Cargo.toml
|   |-- src
|       |-- lib.rs
|-- main
    |-- Cargo.toml
    |-- src
        |-- main.rs

6 directories, 7 files
```

In each crate, the modules need to be exported for use by the main program. For example, in the ”hash” library, we can implement the ”mersenne_hash” function in ”merhash.rs” and export it in ”lib.rs”. Similarly, in the ”encryptor” library, we can implement the function for generating passwords in ”password.rs” and export it in ”lib.rs”. The ”lib.rs” files of each crate should only be used to export functions. For mersenne_hash, it can be encapsulated in merhash.rs, while the function for generating passwords can be encapsulated in password.rs. The final directory looks like this:

```
|-- Cargo.toml
|-- encryptor
|   |-- Cargo.toml
|   |-- src
|       |-- lib.rs
|       |-- password.rs
|-- hash
|   |-- Cargo.toml
|   |-- src
|       |-- lib.rs
|       |-- merhash.rs
|-- main
    |-- Cargo.toml
    |-- src
        |-- main.rs

6 directories, 9 files
```

It is important to note that the ”hash” library does not depend on the external libraries and can be implemented first.

```
1 // hash/src/lib.rs
2
3 pub mod merhash; // export module merhash
4
5 #[cft(test)] // test module
6 mod tests {
```

```
7       use crate::merhash::mersenne_hash;
8
9       #[test]
10      fn mersenne_hash_works() {
11          let seed = String::from("jdxjp");
12          let hash = mersenne_hash(&seed);
13          assert_eq!(2000375, hash);
14      }
15  }
```

Then, we implement merhash.

```
1   // hash/src/merhash.rs
2
3   //! Mersenne Hash
4   //!
5   /// calculate hash value with mersenne prime 127
6   ///
7   /// # Example
8   /// use hash::merhash::mersenne_hash;
9   ///
10  /// let seed = "jdxjp";
11  /// let hash = mersenne_hash(&seed);
12  /// assert_eq!(2000375, hash);
13  pub fn mersenne_hash(seed: &str) -> usize {
14      let mut hash: usize = 0;
15
16      for (i, c) in seed.chars().enumerate() {
17          hash += (i + 1) * (c as usize);
18      }
19
20      (hash % 127).pow(3) - 1
21  }
```

To ensure the code is well-documented and easy to maintain, document comments and module comments are used throughout the project. Tests are also included in the document comments, which is very convenient for testing and subsequent document maintenance.

To complete the ”encryptor” library, we need to rely only on the external ”base64” library. In addition, to handle error situations, the ”anyhow” library is also introduced. To do this, we open the ”Cargo.toml” file under the ”encryptor” directory and add the necessary dependencies under the ”[dependencies]” section, including the ”anyhow” and ”base64” libraries, as well as the ”hash” library that was previously written.

```
1   [package]
2   name = "encryptor"
3   authors = ["shieber"]
4   version = "0.1.0"
5   edition = "2021"
6
7   [dependencies]
8   anyhow = "1.0.56"
9   base64 = "0.13.0"
10  hash = { path = "../hash" }
```

The implementation of the ”password.rs” module is shown below.

```
// encryptor/src/lib.rs

pub mod password; // export module password

// encryptor/src/password.rs
use anyhow::{bail, Error, Result};
use base64::encode;
use hash::merhash::mersenne_hash;

/// crypto(length=100), you can exchange/add/delete characters
const CRYPTO: &str = "!pqHr$*+ST1Vst_uv:?wWS%X&Y-/Z01_2.34<ABl
9ECo|x#yDE^F{GHEI[]JK>LM#NOBWPQ:RaKU@}cde56R7=8f/9gIhi,jkzmn";

/// Password generator with hash value, use hash value's high
/// power to get character from CRYPTO
///
/// #Example
/// use encryptor::password::generate_password;
/// let seed = "jdwnp";
/// let length = 16;
/// let passwd = generate_password(seed, length);
/// match passwd {
///     Ok(val) => println!("{:#?}", val),
///     Err(err) => println!("{:#?}", err),
/// }
pub fn generate_password(seed: &str, length: usize)
    -> Result<String, Error>
{
    // check password length
    if length < 6 {
        bail!("length must >= 6"); // return error
    }

    // calculate mer_hash
    let p = match length {
        6..=10 => 1,
        11..=15 => 2,
        16..=20 => 3,
        _ => 3,
    };
    let mut mer_hash = mersenne_hash(seed).pow(p);

    // calculate password by mer_hash
    let mut passwd = String::new();
    let crypto_len = CRYPTO.len();
    while mer_hash > 9 {
        let loc = mer_hash % crypto_len;
        let nthc = CRYPTO.chars()
                         .nth(loc)
                         .expect("Error while getting char!");
```

```
51          passwd.push(nthc);
52          mer_hash /= crypto_len;
53      }
54
55      // combine seed and passwd
56      let interval = passwd.clone();
57      for c in seed.chars() {
58          passwd.push(c);
59          passwd += &interval;
60      }
61
62      // encode passwd to base64
63      passwd = encode(passwd);
64      passwd = passwd.replace("+", "*").replace("/", "*");
65
66      // length is not enough, use interval to fill
67      let interval = passwd.clone();
68      while passwd.len() < length {
69          passwd += &interval;
70      }
71
72      // return first length characters as password
73      Ok(format!("{}: {}", seed, &passwd[..length]))
74  }
```

In the ”Cargo.toml” of the main module, we introduce the necessary libraries, and note that the name is ”PasswdGenerator”.

```
1  [package]
2  name = "PasswdGenerator"
3  authors = ["shieber"]
4  version = "0.1.0"
5  edition = "2021"
6
7  [dependencies]
8  anyhow = "1.0.56"
9  clap = { version= "3.1.6", features= ["derive"] }
10 encryptor = { path = "../encryptor"}
```

Finally, in the main function, we call the ”encryptor” and command line structure to generate passwords and return them.

```
1  // passwdgenerate/src/main.rs
2
3  use anyhow::{bail, Result};
4  use clap::Parser;
5  use encryptor::password::generate_password;
6
7  /// A simple password generator for any account
8  #[derive(Parser, Debug)]
9  #[clap(version, about, long_about= None)]
10 struct Args {
11     /// Seed to generate password
```

```
12      #[clap(short, long)]
13      seed: String,
14
15      /// Length of password
16      #[clap(short, long, default_value_t = 16)]
17      length: usize,
18  }
19
20  fn main() -> Result<()> {
21      let args = Args::parse();
22
23      // seed must be longer than 4 characters
24      if args.seed.len() < 4 {
25          bail!("seed {} length must >= 4", &args.seed);
26      }
27
28      let (seed, length) = (args.seed, args.length);
29      let passwd = generate_password(&seed[..], length);
30      match passwd {
31          Ok(val) => println!("{}", val),
32          Err(err) => println!("{}", err),
33      }
34
35      Ok(())
36  }
```

Once the program is complete, compile it with ”cargo build –release” to get the ”PasswdGenerator” command line tool in the ”target/release/” directory. You can put it in the ”/usr/local/bin” directory so that you can use it anywhere on the system. Of course, using ”cargo doc” can also generate a ”doc” directory, which contains detailed documentation about the project.

Through this small project, readers have become familiar with Rust code organization, module usage, testing methods, and coding styles. Although this password generator is simple, it is completely sufficient for personal use.

1.5 Summary

In this chapter, we provide an overview of Rust fundamentals. Firstly, we begin by introducing the installation of the Rust toolchain and summarizing some learning resources. We then review Rust basics, covering variables, functions, ownership, lifetimes, generics, traits, smart pointers, error handling, and macro systems.

After that, we present a comprehensive project that implements a command-line tool using Rust. Although the content of this chapter is relatively basic, it is essential to understand these concepts to build more complex projects in Rust. Readers who need further clarification are recommended to refer to the Rust documentation.

By the end of this chapter, readers will have a general understanding of Rust. In the following chapters, we will delve into learning data structures and algorithms.

Chapter 2

Computer Science

2.1 Objectives

- Understand the ideas of computer science.
- Understand the concept of abstract data types.
- Review the basics of the Rust programming language.

2.2 Getting Started

In the realm of computer technology, acquiring a solid understanding of the fundamentals is crucial to effectively solving problems. However, even with this foundation, tackling some problems can still be challenging. The complexity of a problem and its corresponding solution can often obscure the key ideas needed to reach a resolution. Additionally, since problems can have multiple solutions, each with its own constraints and logic, trying to combine the structure of one problem with the solution logic of another can lead to trouble.

This chapter focuses on computer science, algorithms, and data structures, specifically examining why studying these topics is essential. By understanding the structure and logic of problem statements, we can better approach solving problems.

2.3 What is Computer Science?

Computer science is a challenging field to define, possibly due to the word ”computer” in its name. However, computer science is not limited to the study of computers alone, as it encompasses the study of problems, solutions, and the process of generating solutions. The objective of computer scientists is to create an algorithm, a set of instructions, to solve a given problem instance. Following this algorithm allows similar problems to be solved in a finite amount of time. Although computer science can be regarded as the study of algorithms, it is important to recognize that some problems do not have algorithms to solve them, such as NP-complete problems [9]. Although these problems cannot currently be solved, researching them is essential because solving them could result in technological advancements. Like the ”Goldbach Conjecture [10]”, research into it has developed many tools. Therefore, computer science can be defined as the study of solvable problem solutions and unsolvable problem ideas.

When describing problems and solutions, a computable problem is one that has an algorithm to solve it. In contrast, computer science can be defined as ”the study of whether there is a solution to problems that are computable and non-computable.” It is noteworthy that this definition does not mention the word ”computer.” Solutions are independent of the machine and consist of ideas, regardless of whether a computer is employed.

Computer science focuses on the art of problem-solving, which is achieved through abstraction. Abstraction allows us to consider problems and their solutions from a logical perspective, rather than being bogged down by physical details. An everyday example of this is driving a car. As a driver, you interact with the car to reach your destination, using functions provided by the car's designer, such as the key, engine, gears, brakes, accelerator, and steering wheel. These functions are interfaces that simplify the driving experience. In contrast, a car mechanic has a different perspective. They understand the inner workings of the car, including the engine, transmission, temperature control, and windshield wipers. These details operate at the physical level and happen "under the hood."

Similarly, users interact with computers from a logical perspective, using programs such as email, document editors, and media players. In contrast, computer scientists, programmers, and system administrators work at a lower level, understanding how the operating system works, how to configure network protocols, and how to write control function scripts. The key concept behind both of these examples is abstraction. By hiding the complex details of the system, an interface simplifies the user's interaction with the computer. Rust's mathematical calculation functions are a good example of this.

```
1 // sin_cos_function.rs
2
3 fn main() {
4     let x: f32 = 1.0;
5     let y = x.sin();
6     let z = x.cos();
7     println!("sin(1) = {y}, cos(1) = {z}");
8     // sin(1) = 0.84147096, cos(1) = 0.5403023
9 }
```

We may not know how to calculate the sine and cosine values, but as long as we understand how to use the function, that's all we need. Someone has implemented the algorithm for calculating sine and cosine, and we can trust that the function will return the correct result. This is the idea of a "black box." The interface describes the function's input and output, while the details of the algorithm are hidden within.

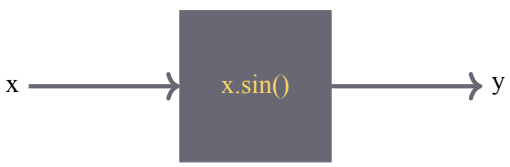


Figure 2.1: calculate the sine value

2.4 What is Programming?

Programming is the process of converting algorithms into computer instructions. An algorithm is a set of steps that produces a solution based on data and expected results. It is essential to have a solution before writing a program. Although computer science is not solely about programming, programming is a crucial skill for computer scientists. When programming, one organizes the problem statement structure in their mind and expresses it clearly for efficient computer processing.

To represent processes and data, programming languages must provide control methods and various data types. Control methods allow for algorithms to be expressed clearly and unambiguously, with sequential processing, selection, and iterative repetition being the minimum requirements.

All data items in a computer are represented in binary form, and data types provide an interpretation and presentation of this binary data. Data types are abstractions of the physical world used to represent

entities in the problem, with underlying primitive data types such as integer and floating-point data types being the foundation for algorithm development.

The operations that can be performed on data are also described in the data type. For example, the most fundamental operations for numbers are addition, subtraction, multiplication, and division. However, programming languages' simple structures and data types may not always suffice to represent complex solutions. To control this complexity, more reasonable data management methods (data structures) and operation processes (algorithms) are required.

2.5 Why Study Data Structures and Abstract Data Types?

Computer scientists use abstraction to manage the complexity of problems and obtain concrete steps to solve them. Abstraction allows them to focus on the big picture without getting lost in the details. Creating a domain model enables computers to solve problems more effectively, and algorithms can be described in a more consistent manner.

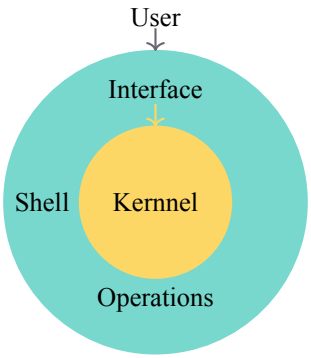


Figure 2.2: system level diagram

An Abstract Data Type (ADT) is a logical description of how to view and operate on data. It provides a level of abstraction where only the meaning of the data is of concern, not its final storage form. By wrapping the data in this way, implementation details can be hidden. The user's interaction with the interface is an abstract operation, while the user and shell are abstract data types. Although the specific operations are not known, understanding their interaction mechanism is still possible. This is the benefit of abstract data types for algorithm design.

To implement abstract data types, primitive data types are used to build new data types from a physical view, known as data structures. There are usually many different ways to implement abstract data types, but different implementations should have the same physical view. This allows programmers to change implementation details without altering the interaction method, and users to continue to focus on the problem.

The common logic for abstract data types includes creating new data types, retrieving data, adding, deleting, searching, modifying, checking for empty data, and computing size. For example, to implement a queue, its abstract data type logic should include at least: new(), is_empty(), len(), clear(), enqueue(), and dequeue(). Once the specific operational logic is known, implementation becomes simple, and there are various ways to implement it as long as these abstract logic exists.

2.6 Why Study Algorithms?

In the computer world, the goal is to use algorithms that are efficient in terms of speed and memory usage. However, some problems may be difficult to handle, and no algorithm can provide an answer within an expected time frame. In such cases, it is important to distinguish between problems that have

solutions and those that do not, as well as problems that have solutions but require significant time or resources. As computer scientists, we need to evaluate and compare different solutions to determine the most appropriate one.

Learning from the problem-solving approaches of others is an efficient way to develop our skills. By examining different solutions, we can gain insights into how various algorithm designs help us tackle challenging problems. By studying various algorithms, we can uncover their core principles and develop a universal algorithm that can be applied to similar problems in the future. However, for the same problem, different people may provide different algorithm implementations. As we saw earlier in the example of computing sine and cosine, there can be many different implementation versions. If one algorithm can achieve the same result as another algorithm in less time, it is obviously the better option.

2.7 Summary

This chapter delves into the concepts of computer science thinking and abstract data types, providing clear definitions and highlighting the roles of algorithms and data structures. By abstracting data types, specific implementation and operation logic is removed, which clarifies the boundaries between data structures and algorithms and significantly simplifies the process of algorithm design. In the forthcoming chapters, we will continue to utilize abstract data types to design various data structures.

Chapter 3

Algorithm Analysis

3.1 Objectives

- Understanding the Importance of Algorithm Analysis.
- Learning to perform performance benchmark tests on Rust programs.
- Being able to use the Big O notation to analyze the complexity of algorithms.
- Understanding the Big O analysis results for Rust data structures such as arrays.
- Understanding how Rust data structure implementations affect algorithm analysis.

3.2 What is Algorithm Analysis?

In Chapter 2, we defined an algorithm as a set of instructions that solves a specific problem by producing an expected result given a specific input. A program is an algorithm written in a programming language. As programmers use different programming languages and have varying levels of knowledge, different programs may describe the same algorithm.

New computer science students may compare their programs with others. To explore the differences between programs that solve the same problem, let's consider the functions "sum_of_n" and "tik_tok," which both calculate the sum of the first n integers. The first function uses an accumulator variable initialized to 0 and iterates over the n integers, adding each value to the accumulator. The "tik_tok" function does the same thing but uses unclear identifier names and an unnecessary assignment.

```
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 { sum += i; }
5      sum
6  }
7
8  // tiktok.rs
9  fn tiktok(tik: i32) -> i32 {
10     let mut tok = 0;
11     for k in 1..=tik  {
12         let ggg = k;
13         tok = tok + ggg;
14     }
15     tok
16 }
```

When evaluating the two functions, the answer depends on the reader's criteria. If readability is the concern, then "sum_of_n" is better. However, in this book, we focus more on the statements of the algorithms themselves, and clean coding is not part of the discussion.

Algorithm analysis compares the amount of resources used by algorithms. An algorithm is better than another algorithm if it uses fewer resources or is more efficient in using resources. When comparing the "sum_of_n" and "tik_tok" functions, they appear very similar in terms of resource usage. Identifying the resources used for computation is crucial and requires considering both time and space perspectives.

• The space used by an algorithm is determined by its memory consumption, which is typically influenced by the size and nature of the problem. However, some algorithms have special space requirements.

• Algorithm execution time refers to the time it takes for an algorithm to execute all its steps.

For instance, the execution time of the "sum_of_n" function can be analyzed through benchmark testing. In Rust, the execution time of code can be calculated by recording the system time before and after function execution. The SystemTime function, which is available in std::time, returns the system time when called and gives the elapsed time afterward. By calling this function at the beginning and end of function execution, we can obtain the function execution time.

```
1  // static_func_call.rs
2
3  use std::time::SystemTime;
4
5  fn sum_of_n(n: i64) -> i64 {
6      let mut sum: i64 = 0;
7      for i in 1..=n {
8          sum += i;
9      }
10     sum
11 }
12
13 fn main() {
14     for _i in 0..5 {
15         let now  = SystemTime::now();
16         let _sum = sum_of_n(500000);
17         let duration = now.elapsed().unwrap();
18         let time = duration.as_millis();
19         println!("func used {time} ms");
20     }
21 }
```

To illustrate this, we executed the "sum_of_n" function five times, each time calculating the sum of the first 500,000 integers and 1000,000 integers. The results were as follows:

```
func used 10 ms
func used 6 ms
func used 6 ms
func used 6 ms
func used 6 ms
```

```
func used 17 ms
func used 12 ms
func used 12 ms
func used 12 ms
func used 12 ms
```

After comparing the results of two executions, it was found that the time taken by the function to execute is basically the same. On average, the function takes 6 milliseconds to execute. The first execution takes longer at 10 milliseconds because the function requires initialization and preparation. The subsequent four executions do not require initialization, and their times are more accurate, hence why multiple executions are necessary. When calculating the sum of the first 1,000,000 integers, it can be observed that the first execution takes longer, and the subsequent executions take exactly twice the time taken to calculate the sum of the first 500,000 integers. This indicates that the execution time of the algorithm is proportional to the size of the calculation.

To further explore this idea, consider the following function, which also calculates the sum of the first n integers, but uses the mathematical formula $\sum_{i=0}^{n} = \frac{n(n+1)}{2}$ to compute it.

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

The sum_of_n function in static_func_call.rs is modified to use this formula, and a benchmark test is performed using three different values of n (100,000, 500,000, 1,000,000). Each calculation is repeated five times, and the average execution time is recorded. The results are as follows:

```
func used 1396 ns
func used 1313 ns
func used 1341 ns
```

There are two important points to note in the output. First, the recorded execution time is in nanoseconds, which is significantly shorter than the previous examples. The execution times for all three calculations are approximately 0.0013 milliseconds, which is orders of magnitude shorter than the previous 6 milliseconds. Second, the execution time is independent of n. As n increases, the calculation time remains the same, indicating that the calculation is almost unaffected by n.

This benchmark test indicates that the iterative solution sum_of_n performs more work because some program steps are repeated, which is why it takes longer. Moreover, the execution time of the iterative solution increases with n. Additionally, it should be noted that running similar functions on different computers or using different programming languages may yield different results. The calculated value of 1341 nanoseconds mentioned in the text refers to the computer used in this study, which is Lenovo Legion R7000P with a 16-core AMD R7-4800H CPU. Older computers may take longer to execute sum_of_n2.

It is necessary to find a better way to describe the execution time of algorithms. Although benchmark tests measure the actual time taken by a program to execute, they do not provide a useful measure because the execution time depends on the specific machine, and there are also magnitude conversions between milliseconds and nanoseconds. We require a measure that is independent of the program or computer used, and can be used to compare the efficiency of different algorithm implementations. Big O notation is a good method for measuring algorithmic efficiency in the field of algorithm analysis, analogous to acceleration measuring the change in speed per second.

3.3 Big-O Notation Analysis

To evaluate the performance of an algorithm, we need to quantify the number of operation steps and storage space required independently of any specific program or computer. For the summing algorithm, counting the execution statements is a good measure. In the function sum_of_n, there is one assignment statement (the_sum = 0), and n addition statements (the_sum += i), making the statement count for the function equal to n + 1. The storage space used is two variables, n and sum, so the count is 2.

We use the function T to represent the total number of executions to measure the algorithm's time complexity, where $T(n) = 1 + n$. The parameter n represents the problem size, and $T(n)$ is the time spent solving a problem with a size greater than or equal to n. Using n to represent the problem size makes sense for the summing function, as summing 100,000 integers is a larger problem than summing 1000 integers, and thus takes longer. Our goal is to demonstrate how the algorithm's execution time changes relative to the problem size.

For measuring the algorithm's space complexity, we use the function S to represent the total memory consumption, where $S(n) = 2$. The parameter n still represents the problem size, but $S(n)$ is independent of n. To analyze the performance of an algorithm, it is crucial to determine the primary components of its time complexity function $T(n)$ and space complexity function $S(n)$, rather than merely counting the number of operation steps and space usage. As the problem size grows, some components of these functions become dominant. The fastest-growing parts of these functions with n are represented by the big O notation, denoted as $O(f(n))$. The function $f(n)$ signifies the primary part of $T(n)$ or $S(n)$.

For instance, $T(n) = n + 1$ in the previous example. As n increases, the constant 1 becomes less significant. Thus, we can approximate the running time of $T(n)$ as $O(T(n)) = O(n + 1) = O(n)$. While the constant 1 is undoubtedly important for $T(n)$, its contribution becomes negligible when n is large. However, when $T(n) = n^3 + 1$, (n = 1), discarding the constant 1 is unreasonable since it would mean discarding a significant portion of the running time. In this case, using $O(T(n)) = O(n^3 + 1)$ is appropriate for small values of n. But as n increases, the 1 becomes less significant, and we can approximate $T(n)$ as $O(T(n)) = O(n^3)$.

For $S(n)$, since it is a constant, $O(S(n)) = O(2) = O(1)$. The big O notation only represents the order of magnitude. Therefore, although the actual complexity of $S(n)$ is $O(2)$, we use $O(1)$ instead.

Assuming an algorithm has a number of operational steps determined by $T(n) = 6n^2 + 37n + 996$. When n is small, such as 1 or 2, the constant term 996 may seem to be the main part of the function. However, as n grows larger, the significance of the n^2 term increases. In fact, as n becomes very large, the other two terms become negligible in the final result. Thus, we can approximate $T(n)$ by ignoring the other terms and focusing only on $6n^2$. Moreover, the coefficient 6 also becomes insignificant as n grows larger. Thus, we say that $T(n)$ has a complexity order of n^2, or $O(n^2)$.

However, the complexity of some algorithms depends on the exact values of the data, rather than the size of the problem. For such algorithms, their performance needs to be characterized based on best case, worst case, or average case scenarios. Worst case refers to a specific dataset that leads to particularly poor algorithm performance, while the same algorithm may have significantly different performance under different datasets. In most cases, the efficiency of algorithm execution lies between the two extremes of worst and best cases, i.e., the average case. Thus, it is important for programmers to understand these differences and avoid being misled by a particular case.

In the study of algorithms, some common order functions appear repeatedly, as shown in the table and figure below. To determine which of these functions is the main part, we need to observe how they relate to each other as n grows larger.

Table 3.1: Complexity functions

$O(1)$	$O(logn)$	$O(n)$	$O(nlogn)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
constants	logarithmic	linear	linear logarithmic	quadratic	cubic	power exponential.

The figure below illustrates the growth of various functions. At small values of n, it can be difficult to discern which function dominates the others. However, as n increases, the relative sizes of the functions become clear. It is worth noting that at $n = 10$, 2^n exceeds n^3, though this case is not shown on the graph. The figure also highlights the differences between different orders of magnitude. Generally, for $n > 10$, we have $O(2^n) > O(n^3) > O(n^2) > O(n \log n) > O(n) > O(\log n) > O(1)$. This information is useful in algorithm design, as it allows us to calculate the complexity of an algorithm. If we determine that an algorithm has a complexity similar to $O(2^n)$, for example, we know that it is not practical and can look to optimize it for better performance.

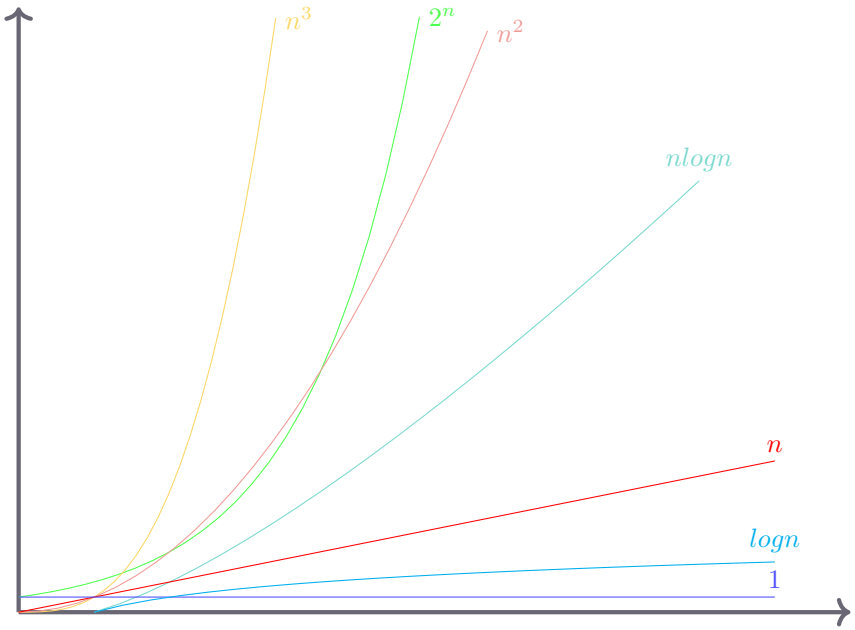


Figure 3.1: Complexity curve

The above analysis examines time and space complexity, but often time complexity is the primary concern because space is typically difficult to optimize. For example, the space complexity of an input array is limited from the outset, but different algorithms can result in significant differences in the time required to run on the array. Moreover, as storage becomes cheaper and more abundant, time becomes the most important factor because it cannot be increased. Therefore, most of the complexity analysis in this book focuses on time complexity, unless otherwise specified.

The following code only employs addition, subtraction, multiplication, and division. We can use it to apply our newly acquired understanding of algorithm complexity analysis.

```
1  // big_o_analysis.rs
2  fn main() {
3      let a = 1; let b = 2;
4      let c = 3; let n = 1000000;
5
6      for i in 0..n {
7          for j in 0..n {
8              let x = i * i;
9              let y = j * j;
10             let z = i * j;
11          }
12     }
13
14     for k in 0..n {
15         let w = a*b + 45;
16         let v = b*b;
17     }
18
19     let d = 996;
20 }
```

The code above has a time complexity of O(n^2), which can be obtained by analyzing the code. Firstly, the time to allocate operations a, b, c, and n is a constant 4. Then, the second term is $3n^2$, which comes from the execution of three statements n^2 times due to nested iteration. The third term is $2n$, as two statements are iteratively executed n times. Finally, the fourth term is a constant 1, representing the final assignment statement d = 996. Thus, the function that determines the number of operational steps is $T(n) = 4 + 3n^2 + 2n + 1 = 3n^2 + 2n + 5$. By examining the exponent, we can see that the n^2 term is the most significant, indicating that the time complexity of this code is O(n^2).

3.4 Anagram Detection

The problem of checking if two strings are permutations of each other serves as an example to demonstrate different levels of complexity. A permutation is defined as one string, s1, being a rearrangement of another string, s2. For instance, ”heart” and ”earth” are permutations of each other, as are ”rust” and ”trus”. Let us assume that both strings have the same length and consist only of 26 lowercase letters. Our goal is to write a function that takes two strings as inputs and returns whether or not they are permutations of each other.

3.4.1 Brute Force

The most naive solution to this problem is to use brute force and check all possible permutations of s1. This method is highly resource-intensive, consuming both time and memory. When generating all possible strings of s1, there are n possibilities for the first position, n-1 for the second position, n-2 for the third position, and so on, resulting in $n! = n \times (n-1) \times (n-2) \times ... \times 3 \times 2 \times 1$ total possibilities. Although some strings may be duplicates, the program cannot know this in advance, and thus it will generate $n!$ strings.

For example, if s1 has a length of 20, there will be 20! = 2,432,902,008,176,640,000 possible strings. If we process one possible string per second, it would take 77,146,816,596 years to exhaust the entire list. In fact, $n!$ grows even faster than n^2, making brute force an infeasible solution for any learning or real software project. Nonetheless, being aware of its existence and making an effort to avoid such situations is essential.

3.4.2 Checking Off

A more efficient solution to the problem involves checking if each character in the first string is present in the second string. If all characters are found, then the two strings are permutations of each other. We can use the ’ ’ character to replace a character once it has been checked.

```
1  // anagram_solution2.rs
2
3  fn anagram_solution2(s1: &str, s2: &str) -> bool {
4      if  s1.len() != s2.len() {
5          return false;
6      }
7
8      // put the characters of s1 and s2 into
9      // vec_a and vec_b respectively
10     let mut vec_a = Vec::new();
11     let mut vec_b = Vec::new();
12     for c in s1.chars() {
13         vec_a.push(c);
14     }
15     for c in s2.chars() {
```

```
16          vec_b.push(c);
17      }
18
19      // index: pos1, pos2
20      let mut pos1: usize = 0;
21      let mut pos2: usize;
22
23      // loop controlling
24      let mut is_anagram = true;
25      let mut found: bool;
26      while pos1 < s1.len() && is_anagram {
27          pos2 = 0;
28          found = false;
29          while pos2 < vec_b.len() && !found {
30              if vec_a[pos1] == vec_b[pos2] {
31                  found = true;
32              } else {
33                  pos2 += 1;
34              }
35          }
36
37          // replace a character with ' '
38          if found {
39              vec_b[pos2]= ' ';
40          } else {
41              is_anagram = false;
42          }
43
44          // tackle the next character of s1
45          pos1 += 1;
46      }
47
48      is_anagram
49  }
50
51  fn main() {
52      let s1 = "rust";
53      let s2 = "trus";
54      let result = anagram_solution2(s1, s2);
55      println!("s1 and s2 is anagram: {result}");
56      // s1 and s2 is anagram: true
57  }
```

Upon analyzing this algorithm, we observe that each character in s1 is compared to all characters in s2, which leads to a maximum of n comparisons for each character in s1. As a result, the n positions in blist will be visited once to match the characters from s1. The total number of visits can be expressed as the sum of integers from 1 to n.

$$1 + 2 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n \tag{3.1}$$

The time complexity of this algorithm is $O(n^2)$, as the term n^2 dominates as n grows. This is much better than the brute force algorithm that has a complexity of $O(n!)$.

3.4.3 Sort and Compare

An alternative solution for scrambled strings is to take advantage of the fact that even if s1 and s2 are different, they consist of exactly the same characters. Therefore, we can sort each string in alphabetical order from a to z, and if the two sorted strings are the same, then they are scrambled strings.

```
// anagram_solution3.rs

fn anagram_solution3(s1: &str, s2: &str) -> bool {
    if  s1.len() != s2.len() {
        return false;
    }

    // put the characters of s1 and s2 into
    // vec_a and vec_b respectively then sorting
    let mut vec_a = Vec::new();
    let mut vec_b = Vec::new();
    for c in s1.chars() {
        vec_a.push(c);
    }
    for c in s2.chars() {
        vec_b.push(c);
    }
    vec_a.sort();
    vec_b.sort();

    // compare vec_a and vec_b, if thers is any mismatch of
    // any character then break the loop
    let mut pos: usize = 0;
    let mut is_anagram = true;
    while pos < vec_a.len() && is_anagram {
        if vec_a[pos] == vec_b[pos] {
            pos += 1;
        } else {
            is_anagram = false;
        }
    }

    is_anagram
}

fn main() {
    let s1 = "rust";
    let s2 = "trus";
    let result = anagram_solution3(s1, s2);
    println!("s1 and s2 is anagram: {result}");
    // s1 and s2 is anagram: true
}
```

At first glance, the time complexity of the algorithm that checks for scrambled strings seems to be $O(n)$ because it has only one while loop. However, the complexity of calling the sort() function is not negligible, and it is usually $O(n^2)$ or $O(n \log n)$, which makes the algorithm's complexity of the same order of magnitude as sorting algorithms.

3.4.4 Count and Compare

The previous methods always create vec_a and vec_b, which can be memory-intensive when s1 and s2 are long. To solve this problem, we can use two lists with a length of 26 to count the frequency of each character in s1 and s2. This approach involves multiple iterations, but the iterations are non-nested. Moreover, the third iteration that compares the two frequency lists only needs 26 iterations because there are only 26 lowercase letters. Thus, the time complexity of this algorithm is $O(n)$, where n is the length of s1 or s2. This is a linear time complexity algorithm, and both its space and time complexities are relatively efficient.

```rust
// anagram_solution4.rs

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

    // with the size of 26, c1 and c2 are storage
    // for saving ASCII charaters
    let mut c1 = [0; 26];
    let mut c2 = [0; 26];
    for c in s1.chars() {
        let pos = (c as usize) - 97; // 97: ASCII value of a
        c1[pos] += 1;
    }
    for c in s2.chars() {
        let pos = (c as usize) - 97;
        c2[pos] += 1;
    }

    // compare the value of ASCII
    let mut pos = 0;
    let mut is_anagram = true;
    while pos < 26 && is_anagram {
        if c1[pos] == c2[pos] {
            pos += 1;
        } else {
            is_anagram = false;
        }
    }

    is_anagram
}

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

However, if s1 and s2 are short and do not use all 26 characters, this algorithm sacrifices some storage space. When deciding which algorithm to use, it is essential to make a trade-off between space and time and consider the real scenarios that your algorithm will face.

3.5 Performance of Rust Data Structures

3.5.1 Scalar and Complex Data Structures

In this section, we will explore the big O performance of Rust's basic data types. Understanding the performance of these data structures is crucial because they are the core building blocks in Rust, and more complex data structures are constructed from them. Rust's basic data types are divided into two major categories: scalar and composite types. Scalar types represent a single value, while composite types are combinations of scalar types.

Rust has four basic scalar types: integer, floating-point, boolean, and character. On the other hand, Rust has two composite types: tuple and array. Scalar types are the most basic and memory-coupled primitive types, with computational efficiency that is typically $O(1)$. Meanwhile, composite types are more complex, and their complexity varies with their data scale.

Let's take a look at some examples of using basic data types.

```
let a: i8 = -2;
let b: f32 = 2.34;
let c: bool = true;
let d: char = 'a';

// tuple with various types
let x: (i32, f64, u8) = (200, 5.32, 1);
let xi32 = x.0;
let xf64 = x.1;
let xu8  = x.2;
```

The tuple is a composite data structure that combines multiple values of various types, with a fixed length. Once declared, its length cannot be increased or decreased. Tuple indexing starts from 0, and its values can be directly accessed using the "."(dot notation). Arrays, on the other hand, must have the same type for each element, and once declared, their length cannot be increased or decreased.

```
let months = ["January","February","March","April",
              "May","June","July","August","September",
              "October","November","December"
             ];

let first_month = months[0]
let halfyear = &months[..6];

let mut monthsv = Vec::new();
for month in months { monthsv.push(month); }
```

Rust offers various data types, composed of scalar and composite types, such as Vec and HashMap. Vec is a collection type similar to an array, which allows for adding, shrinking, and limiting its length. It provides greater scalability and can be used wherever an array can be used.

3.5.2 Collection Data Structures

Rust's collection types are divided into linear and non-linear types, constructed based on scalar and composite types. The linear collection types include String, Vec, VecDeque, and LinkedList, while the non-linear collection types include HashMap, BTreeMap, HashSet, BTreeSet, and BinaryHeap. These collection types involve indexing, adding, and deleting operations, and their respective complexities are often $O(1)$, $O(n)$, etc.

Rust's String type is based on Vec, which allows String to be modified like Vec. To use a portion of the String, one can use &str, which is a slice of the String type that is easy to index. However, since &str is based on String, it cannot be changed because modifying the slice would change the data in String, which may be used elsewhere. In Rust, mutable strings use String, while immutable strings use &str. Vec is similar to lists in other languages, and VecDeque extends Vec to support inserting data at both ends of the sequence, making it a double-ended queue. LinkedList is a linked list and can be used when an unknown size Vec is desired.

The HashMap in Rust is similar to dictionaries in other languages, while BTreeMap is a B-tree whose nodes contain data and pointers. It is often used to implement databases, file systems, and other content storage applications. The HashSet and BTreeSet in Rust are similar to the Set collection in other languages, which are used to record a single value that has appeared once. HashSet is implemented using HashMap as its underlying data structure, while BTreeSet is implemented using BTreeMap. BinaryHeap is a priority queue, storing a set of elements that can extract the maximum value at any time.

Rust's collection data types are highly efficient, as can be seen from the tables below that show the performance of various data structures. The highest complexity of these data structures is $O(n)$.

Table 3.2: Time complexity of linear composite types

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	$O(min(i, n-i))$	$O(min(i, n-i))$	$O(min(i, n-i))$	$O(1)$	$O(min(i, n-i))$

Table 3.3: Time complexity of non-linear composite types

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	$O(1)$	$O(1ogn)$	$O(1)$	$O(1)$	$O(n+m)$

3.6 Summary

This chapter introduced the big O notation analysis method for evaluating algorithm complexity, which involves calculating the number of code execution steps and determining the maximum quantity level. We then examined the complexity of Rust's basic data types and collection data types. Upon comparison and analysis, it becomes clear that Rust's built-in scalar, composite, and collection data types are highly efficient. Additionally, custom data structures implemented based on these collection types can also achieve efficient.

Chapter 4

Basic Data Structures

4.1 Objectives

- Understanding Abstract Data Types Vec, Stack, Queue, Deque, Linked List
- Being able to implement stack, queue, deque, and linked list using Rust
- Understanding the performance (complexity) of basic linear data structures
- Understanding prefix, infix, and postfix expression formats
- Using a stack to implement postfix expression and calculate the value
- Using a stack to convert infix expressions to postfix expressions
- Being able to identify whether to use a stack, queue, deque, or linked list for a given problem
- Being able to implement abstract data types as linked lists using nodes and references
- Being able to compare the performance of self-implemented Vec with Rust's built-in Vec.

4.2 Linear Structures

Data structures are used to store data in a container, and there are several types of data structures available such as arrays, stacks, queues, dequeues, and linked lists. These data structures are called linear data structures because they maintain the order in which the data items are added or removed. Linear data structures have two ends, which can be named as the left and right or the front and back. The names are not important, but they indicate that the organization of data is linear, and this linear characteristic is related to memory, which is linear hardware. This relationship between software and hardware is important to understand.

It is important to note that the term "linear" in linear data structures does not refer to the way data is stored but rather to the way it is accessed. In a linked list, for example, data items may not be adjacent in memory, but the access is still linear.

To differentiate between linear data structures, we need to look at the way items are added and removed, particularly their positions. Some data structures only allow adding items from one end, while others only allow removing items from the other end. Some data structures allow manipulation of data items from both ends. These variations and combinations have led to the creation of many useful data structures in computer science, which are used in various algorithms to perform important and practical tasks.

4.3 Stack

A stack is a linear data structure that has numerous applications, such as in function calls and webpage data recording. It is an ordered collection of data items where new items are added or removed from

the top, while the bottom refers to the opposite end. The bottom is significant because the items closest to it have been stored for the longest time, and the most recently added item will be the first one to be removed. This is known as the Last In First Out (LIFO) or First In Last Out (FILO) principle, which means that newer items are closer to the top, while older items are closer to the bottom.

Stacks are ubiquitous in everyday life, like bricks on a construction site, books on a desk, and plates in a restaurant. To access the bottom brick, book, or plate, we need to remove the items on top first. The schematic diagram of a stack is shown below with some conceptual names.

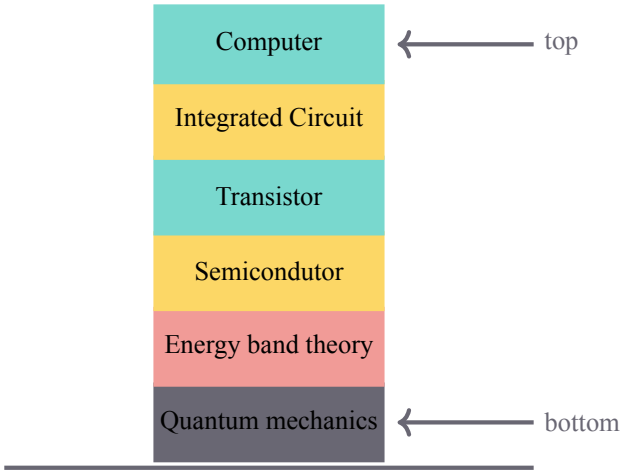


Figure 4.1: Stack

To understand the function of stacks, it is best to observe how they are formed and emptied. Suppose we stack books on a clean desktop one by one; we are creating a stack. Now, if we remove a book, the order of removal is opposite to the order in which the books were placed. The significance of stacks lies in their ability to reverse the order of items, inserting and deleting in the opposite order. The following diagram shows the process of creating and deleting data objects, with particular attention to the order of the data.

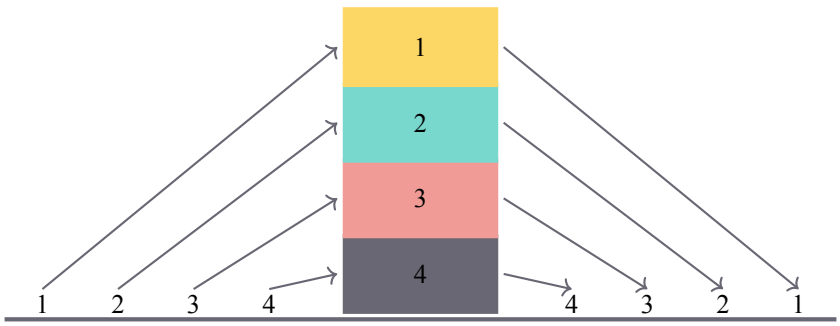


Figure 4.2: Stack reverse order

The ability to reverse the order of items makes the reversal property of stacks particularly useful, as demonstrated in various computer applications. For instance, when browsing news on a web browser, the back function is implemented using a stack. As the user browses web pages, they are stacked, with the current page at the top and the first page at the bottom. Pressing the back button takes the user to the previous page in the opposite order. Without the power of stacks, designing this back function would be nearly impossible. This example highlights the importance of data structures in simplifying certain functions, as choosing the right data structure can make a significant difference.

4.3.1 The Stack Abstract Data Type

A stack is an abstract data type (ADT) that is defined by its structure and operations. It is an ordered collection of items where new items are added and removed from the top. The following operations can be performed on a stack:

• new(): creates an empty stack with no arguments and returns an empty stack.

• push(item): adds the data item 'item' to the top of the stack, taking 'item' as an argument and returning nothing.

• pop(): removes the top data item from the stack, taking no arguments and returning the data item. The stack is modified.

• peek(): returns the top data item from the stack without removing it, taking no arguments and not modifying the stack.

• is_empty(): tests if the stack is empty, taking no arguments and returning a Boolean value.

• size(): returns the number of data items in the stack as a 'usize' integer, taking no arguments.

• iter(): returns an immutable iteration form of the stack, where the stack is not modified and takes no arguments.

• iter_mut(): returns a mutable iteration form of the stack, where the stack can be modified and takes no arguments.

• into_iter(): changes the stack into an iterable form, consuming the stack and taking no arguments.

To illustrate the stack operations, assume that 's' is an already created empty stack (represented as '[]' here), and the table below shows the results of stack operations, with the stack top on the right.

Table 4.1: Stack Operations

Operation	Value	Return Value
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.push(6)	[1,2,3,4,6]	
s.peek()	[1,2,3,4,6]	6

4.3.2 Implemeting a Stack in Rust

The abstract data type of a stack has been defined above, and now we will use Rust to implement it as a data structure. In Rust, abstract data types are implemented by creating a new struct and implementing the operations as functions of the struct. Using the basic data structures provided by Rust to implement a stack and its operations is very helpful.

We will use the Vec collection container as the underlying implementation of the stack because it provides an ordered collection mechanism and a set of operation methods. To implement other operations, we only need to choose which end of the Vec is the top of the stack. The implementation assumes that the tail of Vec will save the top element, and as the stack grows, new items will be added to the end of Vec. Since the type of data to be inserted is unknown, we use the generic data type T. We have also added three structs, IntoIter, Iter, and IterMut, to complete the three types of iteration.

```rust
// stack.rs

#[derive(Debug)]
struct Stack<T> {
    size: usize,  // stack size
    data: Vec<T>, // stack data
}

impl<T> Stack<T> {
    // initialize a stack
    fn new() -> Self {
        Self {
            size: 0,
            data: Vec::new()
        }
    }

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

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

    // clear stack
    fn clear(&mut self) {
        self.size = 0;
        self.data.clear();
    }

    // put the item into the tail of Vec
    fn push(&mut self, val: T) {
        self.data.push(val);
        self.size += 1;
    }

    // size decrease by 1 and then return the value
    fn pop(&mut self) -> Option<T> {
        if 0 == self.size {
            return None;
        }
        self.size -= 1;
        self.data.pop()
    }

    // return reference to the top value
    fn peek(&self) -> Option<&T> {
        if 0 == self.size {
            return None;
        }
```

```
52
53          self.data.get(self.size - 1)
54      }
55
56      fn peek_mut(&mut self) -> Option<&mut T> {
57          if 0 == self.size {
58              return None;
59          }
60
61          self.data.get_mut(self.size - 1)
62      }
63
64      // Implementing iteration for stack
65      // into_iter: stack modified and becomes a iterator
66      // iter: stack unmodified and get a unmutable iterator
67      // iter_mut: stack unmodified and get a mutable iterator
68      fn into_iter(self) -> IntoIter<T> {
69          IntoIter(self)
70      }
71
72      fn iter(&self) -> Iter<T> {
73          let mut iterator = Iter { stack: Vec::new() };
74          for item in self.data.iter() {
75              iterator.stack.push(item);
76          }
77
78          iterator
79      }
80
81      fn iter_mut(&mut self) -> IterMut<T> {
82          let mut iterator = IterMut { stack: Vec::new() };
83          for item in self.data.iter_mut() {
84              iterator.stack.push(item);
85          }
86
87          iterator
88      }
89  }
90
91  // Implementation of 3 iterations
92  struct IntoIter<T>(Stack<T>);
93  impl<T: Clone> Iterator for IntoIter<T> {
94      type Item = T;
95      fn next(&mut self) -> Option<Self::Item> {
96          if !self.0.is_empty() {
97              self.0.size -= 1;
98              self.0.data.pop()
99          } else {
100             None
101         }
102     }
103 }
```

```
104
105 struct Iter<'a, T: 'a> { stack: Vec<&'a T>, }
106 impl<'a, T> Iterator for Iter<'a, T> {
107     type Item = &'a T;
108     fn next(&mut self) -> Option<Self::Item> {
109         self.stack.pop()
110     }
111 }
112
113 struct IterMut<'a, T: 'a> { stack: Vec<&'a mut T> }
114 impl<'a, T> Iterator for IterMut<'a, T> {
115     type Item = &'a mut T;
116     fn next(&mut self) -> Option<Self::Item> {
117         self.stack.pop()
118     }
119 }
120
121 fn main() {
122     basic();
123     peek();
124     iter();
125
126     fn basic() {
127         let mut s = Stack::new();
128         s.push(1); s.push(2); s.push(3);
129
130         println!("size: {}, {:?}", s.len(), s);
131         println!("pop {:?}, size {}",s.pop().unwrap(), s.len())
                ;
132         println!("empty: {}, {:?}", s.is_empty(), s);
133
134         s.clear();
135         println!("{:?}", s);
136     }
137
138     fn peek() {
139         let mut s = Stack::new();
140         s.push(1); s.push(2); s.push(3);
141
142         println!("{:?}", s);
143         let peek_mut = s.peek_mut();
144         if let Some(top) = peek_mut {
145             *top = 4;
146         }
147
148         println!("top {:?}", s.peek().unwrap());
149         println!("{:?}", s);
150     }
151
152     fn iter() {
153         let mut s = Stack::new();
154         s.push(1); s.push(2); s.push(3);
```

```
155
156         let sum1 = s.iter().sum::<i32>();
157         let mut addend = 0;
158         for item in s.iter_mut() {
159             *item += 1;
160             addend += 1;
161         }
162
163         let sum2 = s.iter().sum::<i32>();
164         println!("{sum1} + {addend} = {sum2}");
165         assert_eq!(9, s.into_iter().sum::<i32>());
166     }
167 }
```

After executing the code, we obtain the following results.

```
size: 3, Stack { size: 3, data: [1, 2, 3] }
pop 3, size 2
empty: false, Stack { size: 2, data: [1, 2] }
Stack { size: 0, data: [] }
Stack { size: 3, data: [1, 2, 3] }
top 4
Stack { size: 3, data: [1, 2, 4] }
6 + 3 = 9
```

4.3.3 Simple Balanced Parentheses

Previously, we implemented the stack data structure and now we will use it to solve the parenthesis matching problem. Arithmetic expressions that evaluate values contain parentheses as shown below.

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

Similarly, in the Lisp language, the multiply function shown below also has parentheses.

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

The focus here is on parentheses, as they alter the order of operations and constrain the language semantics. It is essential to ensure that the parentheses are complete; otherwise, the entire expression will be incorrect. Humans can quickly identify incomplete parenthesis, but how can a computer do this? Clearly, the computer must verify whether the parentheses match and report an error if they don't.

In both the examples above, the parentheses must appear in pairs. Parenthesis matching means that each opening symbol has a corresponding closing symbol, and the parentheses are correctly nested so that the computer can process them correctly.

Consider the following correctly matched parenthesis strings and these mismatched parenthesis.

```
(()()()())
((((()))
(()((())()))
```

```
(((((((())
()))
((()(()
```

To solve the parenthesis matching problem, a deeper understanding of brackets and their matching is needed. When processing symbols from left to right, the nearest left starting bracket '(' must match the next right closing symbol ')' (as shown in Figure 4.3). In addition, the first left starting bracket processed must wait until it matches the last right closing bracket. Ending brackets match starting brackets in the opposite order, from inside to outside. This is a problem that can be solved using a stack.

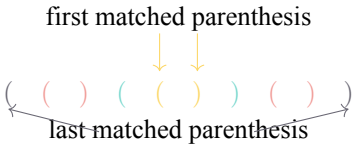


Figure 4.3: Parenthesis matching

Matching parenthesis is crucial in computer programs since it determines the next operation. The challenge is to write an algorithm that can read a string of symbols from left to right and determine whether the brackets match. Parenthesis and their matching are essential for computer programs since bracket and nesting are prevalent in programs.

The implementation of the algorithm for matching parentheses using a stack is quite simple since it only involves push, pop, and judgment operations on the stack. To start, process the bracket string from left to right with an empty stack. If the symbol is a left starting symbol, push it onto the stack; if it is an ending symbol, pop the top element of the stack and match these two symbols. If they match, continue to process the next bracket until the string is completely processed. At the end of the processing, the stack should be empty; if not, it indicates that there are unmatched brackets. Here is a Rust implementation of the bracket matching program.

```
// par_checker1.rs

fn par_checker1(par: &str) -> bool {
    // adding characters into Vec
    let mut char_list = Vec::new();
    for c in par.chars() {
        char_list.push(c);
    }

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

        if '(' == c { // if is '(', put data into stack
            stack.push(c);
        } else {      // if is ')', determine if stack is empty
            if stack.is_empty() { // empty stack, matched
                balance = false;
            } else {
                let _r = stack.pop();
            }
        }

        index += 1;
    }
```

```
28
29      // parenthesis matched: balanced and empty stack
30      balance && stack.is_empty()
31  }
32
33  fn main() {
34      let sa = "(()())";
35      let sb = "(()(()";
36      let res1 = par_checker1(sa);
37      let res2 = par_checker1(sb);
38      println!("{sa} balanced:{res1}, {sb} balanced:{res2}");
39      // (()()) balanced:true, (()(() balanced:false
40  }
```

While the above example only involves matching the parentheses '()', there are actually three commonly used types of parentheses: '()', '[]', and '{}'. These different types of left and right parentheses are often nested together, as shown in Rust, where square brackets '[]' are used for indexing, curly brackets '{}' are used for formatting output, and parentheses '()' are used for function parameters, tuples, mathematical expressions, and more. As long as each symbol maintains its own left starting and right ending relationship, mixed nesting symbols can be used.

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

All of the parentheses shown above are matched. On the contrary, the following expression is not matched.

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

To handle three types of parentheses, the previous parentheses checking program, par_checker1(only able to detect '()' parentheses), needs to be extended. However, the algorithm process remains the same. Each left starting parenthesis is pushed onto the stack, waiting for the matching right ending parenthesis to appear. When an ending parenthesis appears, the program checks whether the types of parentheses match. If the two parentheses do not match, then the string does not match. If the entire string has been processed and the stack is empty, then the parentheses expression matches.

To detect whether the types of parentheses match, a new function called par_match() has been added. This function can detect the three commonly used types of parentheses. The detection principle is very simple: the program arranges the parentheses in order and checks whether their indices match.

```
1  // par_checker2.rs
2
3  // check if parentheses match of various symbols
4  fn par_match(open: char, close: char) -> bool {
5      let opens = "([{";
6      let closers = ")]}";
7      opens.find(open) == closers.find(close)
8  }
9
10 fn par_checker2(par: &str) -> bool {
11     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];
        // check 3 open symbols simultaneously
        if '(' == c || '[' == c || '{' == c {
            stack.push(c);
        } else {
            if stack.is_empty() {
                balance = false;
            } else {
                // determine if match
                let top = stack.pop().unwrap();
                if !par_match(top, c) {
                    balance = false;
                }
            }
        }
        index += 1;
    }
    balance && stack.is_empty()
}

fn main() {
    let sa = "(){}[]";
    let sb = "(){)[}";
    let res1 = par_checker2(sa);
    let res2 = par_checker2(sb);
    println!("sa balanced:{res1}, sb balanced:{res2}");
    // (){}[] balanced:true, (){)[} balanced:false
}
```

The current implementation can handle different types of bracket matching problems. However, if the input expression contains non-bracket characters, the program will fail to work properly.

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

The apparent complexity of the problem can be deceiving because the actual issue is still related to detecting matching brackets. Hence, non-bracket characters can be skipped during processing. In the case of the given example, the non-bracket characters can be ignored, leaving only the brackets, resulting in the string: ()(){}(). The problem remains the same as before, and the code can be modified to detect matching brackets even in strings containing non-bracket characters. The following implementation is the modified code based on par_checker2.rs.

```
// par_checker3.rs

fn par_checker3(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];
        // open mark, push
        if '(' == c || '[' == c || '{' == c {
            stack.push(c);
        }
        // close mark, determine if balanced
        if ')' == c || ']' == c || '}' == c {
            if stack.is_empty() {
                balance = false;
            } else {
                let top = stack.pop().unwrap();
                if !par_match(top, c) { balance = false; }
            }
        }
        // skip if not the bracket
        index += 1;
    }
    balance && stack.is_empty()
}

fn main() {
    let sa = "(2+3){func}[abc]"; let sb = "(2+3)*(3-1";
    let res1 = par_checker3(sa); let res2 = par_checker3(sb);
    println!("sa balanced:{res1}, sb balanced:{res2}");
 // (2+3){func}[abc] balanced:true, (2+3)*(3-1 balanced:false
}
```

4.3.4 Converting Decimal Numbers to Binary Numbers

Binary is a fundamental concept in the computer world, as it is the universal data format at the low-level of computing. All values stored in computers are in the form of 0s and 1s, making it critical to convert between binary numbers and ordinary characters for successful interaction with computers. Integer values are commonly used in computer programs and calculations, but we typically learn them in decimal form. For instance, decimal 233(10) can also be represented in binary as 11101001(2).

$$\begin{aligned} 2 \times 10^2 + 3 \times 10^1 + 3 \times 10^0 &= 233 \\ 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 &= 233 \end{aligned} \tag{4.1}$$

To convert an integer to binary, the ”divide by 2” algorithm (as shown in Figure 4.4) is a simple and effective method that involves using a stack to track the binary result’s digits. Starting with a positive integer greater than 0, the algorithm iteratively divides the decimal number by 2, keeping track of the remainders. The first remainder after dividing by 2 determines whether the value is even or odd, with an even number producing a remainder of 0 and an odd number resulting in a remainder of 1. By recording these remainders during the iteration and reversing the sequence, the binary digit sequence is obtained, with the first remainder (i.e., the last digit in the sequence) placed at the bottom of the stack. Popping all the digits from the stack results in the binary representation of the original decimal number.

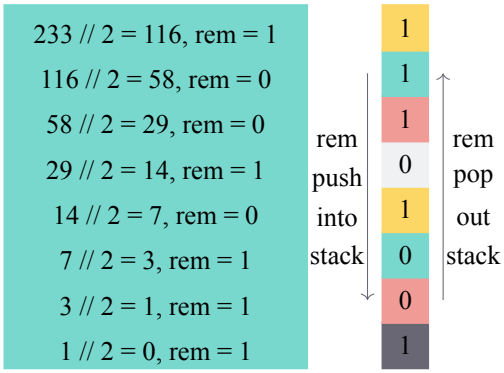


Figure 4.4: Divided by 2

The function shown below takes a decimal parameter and uses Rust's modulus operator (%) to extract the remainder, which is then pushed onto a stack. The parameter is repeatedly divided by 2 until it reaches 0, and the program constructs the binary representation of the original value.

```
// divide_by_two.rs

fn divide_by_two(mut dec_num: u32) -> String {
    // save the remainder in a stack
    let mut rem_stack = Stack::new();

    // push rem into the stack
    while dec_num > 0 {
        let rem = dec_num % 2;
        rem_stack.push(rem);
        dec_num /= 2;
    }

    // pop out elems from the stack to form a string
    let mut bin_str = "".to_string();
    while !rem_stack.is_empty() {
        let rem = rem_stack.pop().unwrap().to_string();
        bin_str += &rem;
    }
    bin_str
}

fn main() {
    let num = 10;
    let bin_str: String = divided_by_two(num);
    println!("{num} = b{bin_str}");
    // 10 = b1010
}
```

This algorithm for converting from decimal to binary can be extended to perform conversions between any two bases commonly used in computer science, such as binary, octal, and hexadecimal. For example, the decimal number 233(10) corresponds to 351(8) in octal and e9(16) in hexadecimal.

To make the function more general, it can be modified to accept a predetermined conversion base,

replacing the concept of dividing by 2 with dividing by the conversion base. The remainders are still pushed onto a stack until the value being converted reaches 0. However, for bases greater than 10, remainders greater than 10 will inevitably occur, so it is best to represent them as a single character. In the base_converter function, we choose to use A-F to represent 10-15, but lowercase letters such as a-f or other character sequences can also be used.

```
// base_converter.rs

fn base_converter(mut dec_num: u32, base: u32) -> String {
    // digits is the string form of integers(especially for 10
       -15)
    let digits = ['0', '1', '2', '3', '4', '5', '6', '7',
                  '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'];
    let mut rem_stack = Stack::new();

    // push rem into the stack
    while dec_num > 0 {
        let rem = dec_num % base;
        rem_stack.push(rem);
        dec_num /= base;
    }

    // pop out elems from the stack to form a string
    let mut base_str = "".to_string();
    while !rem_stack.is_empty() {
        let rem = rem_stack.pop().unwrap() as usize;
        base_str += &digits[rem].to_string();
    }
    base_str
}

fn main() {
    let num1 = 10;
    let num2 = 43;
    let bin_str: String = base_converter(num1, 2);
    let hex_str: String = base_converter(num2, 16);
    println!("{num1} = b{bin_str}, {num2} = x{hex_str}");
    // 10 = b1010, 43 = x2B
}
```

4.3.5 Prefix, Infix, Postfix Expressions

In mathematics, an arithmetic expression, like B * C, has a specific format that makes it easy to understand. For instance, B multiplied by C, with the multiplication operator ”*” in between two operands, forms an infix expression. This type of expression is naturally easy to read and follow since it maintains the order of the expression.

However, an infix expression like A + B * C, with the ”+” and ”*” operators sandwiched between operands, requires an understanding of operator priority to determine the order of operations. When operators of equal priority are present, they evaluate from left to right. Parentheses can alter the order of the operation, and arithmetic operator priority places multiplication and division before addition and subtraction.

While humans can easily solve such expressions by mentally adding parentheses, computers require

specific algorithms to perform the same task. A computer can avoid confusion by using fully parenthesized expressions, where each operator has a pair of parentheses, indicating the order of the operation. However, processing these expressions can be difficult since computers work from left to right. Therefore, even simple tasks that seem easy for humans can be complex for computers, as they lack intelligence and the ability to understand expressions intuitively.

To ensure that computers do not confuse the order of operations, a fully parenthesized expression can be used, where each operator is enclosed in a pair of parentheses indicating the order of operations. However, since computers process data from left to right, they may have difficulty jumping between inner and outer parentheses to calculate expressions like (A + (B * C)). While humans can do this intuitively, it is challenging for computers since they are rigid and require explicit instructions.

A more intuitive method for computers is to separate the operator and operands by moving the operator outside the parentheses. This creates prefix and postfix expressions, which can be distinguished from the original infix expression. In prefix expressions, operators come before the operands, and in postfix expressions, operators come after the corresponding operands. Calculations are performed by taking the operator and operands, calculating the result as the current value, and then proceeding to subsequent operations until the expression is fully calculated.

For example, the infix expression A + B can be written as + A B in prefix notation or A B + in postfix notation. By following the rules for each notation, readers can calculate more complex expressions in prefix or postfix notation and confirm the correct result.

Table 4.2: Prefix-Infix-Postfix Expression

prefix	infix	postfix
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 /

To analyze complex expressions, readers must understand the importance of parentheses in expressing the order of operations. Even small changes in the placement of parentheses can result in vastly different expressions. While infix expressions require parentheses to disambiguate the order of operations, prefix and postfix expressions do not need them. This is because the order of operations is entirely determined by the position of operators in these notations. As a result, prefix and postfix expressions can clearly express the calculation logic without ambiguity, ensuring that calculations based on them are error-free. The table below shows some examples of complex expressions in different notations.

Table 4.3: Complex Pre-In-Postfix Expressions

infix	prefix	postfix
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 - *

Prefix and postfix expressions may appear more complicated to compute at first. For instance, the prefix expression ”+ A * B C” for ”A + B * C” requires calculating ”B * C” before adding it to A. However, the multiplication sign is still inside the addition operator, making it impossible to calculate. To resolve this issue, stacks can be used to reverse the order of operations. By pushing the operands and operators onto separate stacks from left to right, the order of operations can be reversed, as shown in Figure (4.5). This approach enables prefix and postfix expressions to be computed efficiently and accurately.

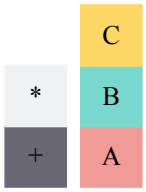


Figure 4.5: Expressions on a stack

To calculate expressions in prefix notation, the operator and operands are pushed onto separate stacks from left to right. To compute the expression, the top operator is popped from the operator stack, and the two top operands are popped from the operand stack. The operator is applied to these operands to obtain a result, which is then pushed back onto the operand stack. This process is repeated until the operator stack is empty. Finally, the top value in the operand stack is the result of the expression. This method is as efficient as fully bracketed expressions since the computer only needs to push and pop from the stacks, and does not need to handle parentheses, as shown in Figure (4.6).

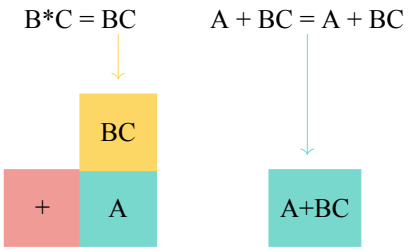


Figure 4.6: calculating expression by the stack

Similarly, postfix expressions can also be calculated using stacks. Only one stack is required to perform the calculation. For example, the postfix expression for A + B * C, which is A B C * +, is calculated by pushing A, B, and C onto the stack. When the * operator is encountered, the top two operands, B and C, are popped, multiplied, and pushed back onto the stack. Then, when the + operator is encountered, the top two operands, A and BC, are popped, added, and pushed back onto the stack, resulting in the final result of A + BC. This approach demonstrates the efficiency of stack-based computations, which can handle expressions in any notation without the need for parentheses.

4.3.6 Conversion of Infix Expressions to Prefix and Postfix

The calculation process demonstrated that converting infix expressions into prefix or postfix expressions is necessary to achieve efficiency. The first step in this process is to obtain prefix or postfix expressions, and a method to achieve this is through the use of fully bracketed expressions.

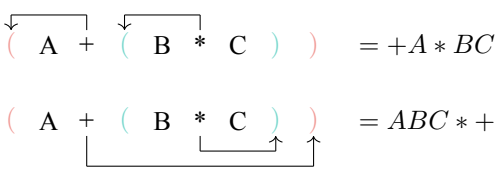


Figure 4.7: infix to prefix and postfix

For instance, A + B * C can be expressed as (A + (B * C)) to indicate that multiplication has a higher priority than addition. It is noteworthy that each pair of brackets represent the start and end of

an operand pair. The internal expression of (A + (B * C)) is (B * C), and by moving the multiplication symbol to the left bracket position and deleting the left and matching right brackets, the subexpression can be converted to a prefix expression. Similarly, by moving the addition operator to its corresponding left bracket position and deleting the matching right bracket, the complete prefix expression + A * B C can be obtained. By repeating this operation for all operators, the complete postfix expression can be obtained.

To convert an expression, regardless of whether it is a prefix or postfix expression, the expression must first be converted into a fully bracketed expression based on the order of operations. Once the expression is fully bracketed, the operator inside the brackets can be moved to the position of the left or right bracket to achieve the desired notation. For example, the more complex expression (A + B) * C - (D + E) / (F + G) can be converted to a prefix or postfix expression using this method. Although the result may be complex for humans, computers can handle this process with ease.

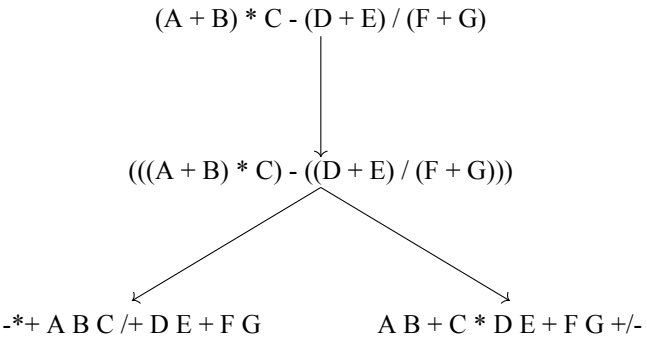


Figure 4.8: Infix to Prefix and Postfix

Obtaining a fully parenthesized expression is a difficult task, and it requires modifying the string by moving and deleting characters. Therefore, this method is not universal enough. A more convenient approach is to handle operators separately. When converting an infix expression, if the operands are not considered, they maintain their original relative positions, and only the operators change position. Therefore, it is not necessary to change the position of operands when encountering them; instead, only operators need to be handled when they are encountered. However, operators have priorities and often reverse the order. The characteristic of reversing the order is inherent to the stack data structure, which can be used to store operators.

To convert an infix expression to a postfix expression, we can use a stack to store operators while scanning the expression from left to right. When an operand is encountered, it is added to the postfix expression. If an operator is encountered, we compare its priority with the operator at the top of the stack. If the current operator has higher priority, it is pushed onto the stack. If the current operator has lower or equal priority, we pop operators from the stack and add them to the postfix expression until we reach an operator with lower priority or an opening parenthesis. When we encounter a closing parenthesis, we pop operators from the stack and add them to the postfix expression until we reach the corresponding opening parenthesis, which is then discarded. The result is a postfix expression that can be evaluated efficiently.

For example, to convert the infix expression (A + B) * C, we scan it from left to right. We first encounter the operand A, which is added to the postfix expression. Next, we encounter the operator +, which has lower priority than the opening parenthesis, so we push it onto the stack. We then encounter the operand B, which is added to the postfix expression. Finally, we encounter the closing parenthesis, so we pop the operator + from the stack and add it to the postfix expression, followed by the operator *. The resulting postfix expression is A B + C *.

In the same way, we can convert the infix expression A * B + C * D to the postfix expression A B * C D * +, as shown in the figure below.

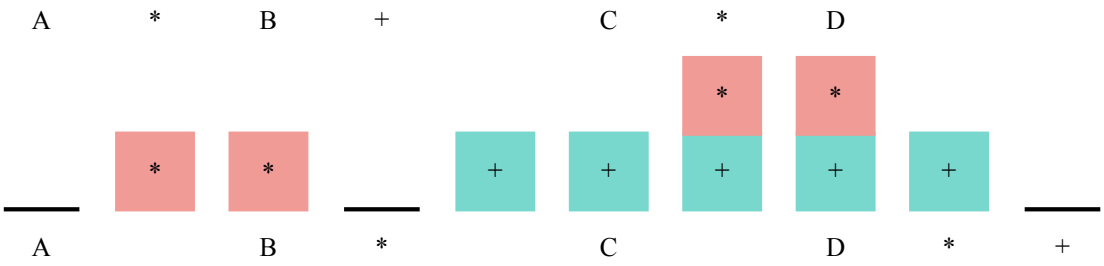


Figure 4.9: Postfix expression by a stack

To convert an infix expression to postfix notation, follow these steps:

1. Create an empty stack op_stack to hold operators, an empty list postfix to store the output.
2. Split the input infix string into tokens and store them in a list called src_str.
3. Iterate through the tokens in src_str from left to right:

If the token is an operand, append it to the end of the postfix list.

If the token is a left parenthesis, push it onto the op_stack.

If the token is a right parenthesis, pop operators from op_stack and append them to postfix until the corresponding left parenthesis is removed.

If the token is an operator (+, -, *, or /), push it onto the op_stack. However, first pop any higher or equal priority operators from op_stack and append them to postfix.

4. After processing the input, check the op_stack. Pop any remaining operators from the op_stack and append them to postfix. The final postfix list is the result of the conversion.

Algorithm 4.1: Algorithm: expression conversion from infix to postfix

```
   Input: infix expression
   Output: postfix expression
 1 Create an empty stack op_stack, an empty list postfix
 2 Convert the input infix expression into a token list(src_str)
 3 for c ∈ src_str do
 4 |   if c ∈ 'A − Z' then
 5 |   |   postfix.append(c)
 6 |   else if c == '(' then
 7 |   |   op_stack.push(c)
 8 |   else if c ∈ ' + − ∗ /' then
 9 |   |   while op_stack.peek() prior to c do
10 |   |   |   postfix.append(op_stack.pop())
11 |   |   end
12 |   |   op_stack.push(c)
13 |   else if c == ')' then
14 |   |   while op_stack.peek() != '(' do
15 |   |   |   postfix.append(op_stack.pop())
16 |   |   end
17 |   end
18 end
19 while !op_stack.is_empty() do
20 |   postfix.append(op_stack.pop())
21 end
22 return ' '.join(postfix)
```

To convert an infix expression to postfix expression, we can use a HashMap named "prec" to store the priority of operators. This HashMap maps each operator to an integer that is used to compare the priority of operators. The priority of parentheses is assigned the lowest value so that any operator compared with them has higher priority. Operators are limited to "+-*/", while operands are defined as uppercase letters $A - Z$ or digits $0 - 9$.

```
// infix_to_postfix.rs
use std::collections::HashMap;

fn infix_to_postfix(infix: &str) -> Option<String> {
    // check parenthesis
    if !par_checker3(infix) {
        return None;
    }

    // set priority of all symbols
    let mut prec = HashMap::new();
    prec.insert("(", 1); prec.insert(")", 1);
    prec.insert("+", 2); prec.insert("-", 2);
    prec.insert("*", 3); prec.insert("/", 3);

    // ops: svae operators, postfix: svae postfix expression
    let mut ops = Stack::new();
    let mut postfix = Vec::new();
    for token in infix.split_whitespace() {
        // characters range from 0 - 9  and  A-Z can
        // be pushed onto the stack
        if ("A" <= token && token <= "Z") ||
            ("0" <= token && token <= "9") {
            postfix.push(token);
        } else if "(" == token  {
            // open mark, push onto the stack
            ops.push(token);
        } else if ")" == token  {
            // close mark, pop out from stack
            let mut top = ops.pop().unwrap();
            while top != "(" {
                postfix.push(top);
                top = ops.pop().unwrap();
            }
        } else {
            // check the priority of operators
            while (!ops.is_empty()) &&
                (prec[ops.peek().unwrap()]
                                    >= prec[token]) {
                postfix.push(ops.pop().unwrap());
            }
            ops.push(token);
        }
    }

    // push the remaining operators onto the stack
```

```
    while !ops.is_empty() {
        postfix.push(ops.pop().unwrap())
    }
    // pop out operators and create the postfix expression
    let mut postfix_str = "".to_string();
    for c in postfix {
        postfix_str += &c.to_string();
        postfix_str += " ";
    }

    Some(postfix_str)
}

fn main() {
    let infix = "( A + B ) * ( C + D )";
    let postfix = infix_to_postfix(infix);
    match postfix {
        Some(val) => { println!("{infix} -> {val}"); },
        None => {
            println!("{infix} isn't a correct infix string");
        },
    }
    // ( A + B ) * ( C + D ) -> A B + C D + *
}
```

When calculating postfix expressions, special attention must be given to the ”-” and ”/” operators. Unlike the ”+” and ”*” operators, the order of operands matters for ”-” and ”/”. For example, A / B and B / A, A - B and B - A are completely different and cannot be handled like ”+”, and ”*”. Assuming the postfix expression is a space-separated string with operators ”+-*/” and operands as integers, the output is also an integer. The following are the algorithm steps for calculating postfix expressions:

1. Create an empty stack named op_stack.
2. Split the string into a list of symbols.
3. Scan the symbol list from left to right. If the symbol is an operand, convert it from a character to an integer and push the value onto op_stack. If the symbol is an operator, pop op_stack twice. The second pop is the first operand, and the first pop is the second operand. Perform the arithmetic operation and push the result back onto the operand stack.
4. When the entire input expression has been processed, the result is on the stack. Pop op_stack to get the final result of the operation.

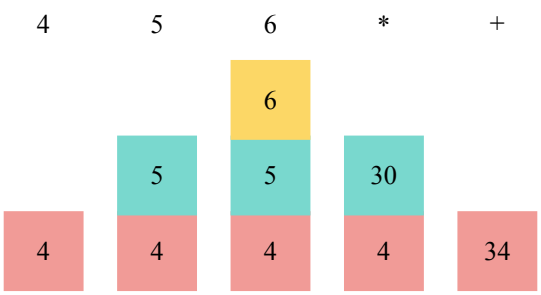


Figure 4.10: Evaluation of Postfix Expression

The specific steps for evaluating postfix expressions are shown in the above figure, and below is the implementation code.

```rust
// postfix_eval.rs

fn postfix_eval(postfix: &str) -> Option<i32> {
    // the expression needs at least two operands and
    // one operator, and two spaces to separate them.
    if postfix.len() < 5 { return None; }

    let mut ops = Stack::new();
    for token in postfix.split_whitespace() {
        // Strings can be compared directly
        if "0" <= token && token <= "9" {
            ops.push(token.parse::<i32>().unwrap());
        } else {
            // For subtraction and division, the order matters
            let op2 = ops.pop().unwrap();
            let op1 = ops.pop().unwrap();
            let res = do_calc(token, op1, op2);
            ops.push(res);
        }
    }
    // The value remained in the stack is the result
    Some(ops.pop().unwrap())
}

// Perform arithmetic operations
fn do_calc(op: &str, op1: i32, op2: i32) -> i32 {
    if "+" == op  {
        op1 + op2
    } else if "-" == op {
        op1 - op2
    } else if "*" == op {
        op1 * op2
    } else if "/" == op {
        if 0 == op2 {
            panic!("ZeroDivisionError: Invalid operation!");
        }
        op1 / op2
    } else {
        panic!("OperatorError: Invalid operator: {:?}", op);
    }
}

fn main() {
    let postfix = "1 2 + 1 2 + *";
    let res = postfix_eval(postfix);
    match res {
        Some(val) => println!("res = {val}"),
        None => println!("{postfix} isn't a valid postfix"),
    }
    // res = 9
}
```

4.4 Queue

A queue is an ordered collection of items that has a front and a rear end. New items are added at the rear, and items are removed from the front. Each element added to the queue moves towards the front until it becomes the next item to be removed. This type of ordering is known as First In First Out (FIFO). A stack, in contrast, uses Last In First Out (LIFO) ordering.

Queues are commonly used in everyday situations such as long lines of people waiting to board a train or bus, or at self-service restaurants. Queues have limited behavior because they have only one entrance and one exit. It is not possible to cut in line or leave early; one must wait for their turn. While it is true that real-life queues and queue data structures may allow cutting in line, this discussion does not consider that possibility.

Operating systems use queues as a data structure to control processes. Multiple different queues are used in scheduling algorithms to prioritize executing programs as quickly as possible and service as many users as possible. When typing on a keyboard, a delay may occur before characters appear on the screen. This is because the keystrokes are placed in a buffer similar to a queue.

Figure 4.11: Queue

4.4.1 The Queue Abstract Data Type

A queue is a data structure that maintains the FIFO sorting property. It is defined by the following structure and operations:

- new() creates a new queue, takes no parameters, and returns an empty queue.
- enqueue(item) adds a new item to the end of the queue, takes item as a parameter, no return.
- dequeue() removes an item from the front of the queue, takes no parameters, returns the removed item, and modifies the queue.
- is_empty() checks whether the queue is empty, takes no parameters, and returns a Boolean value.
- size() returns the number of items in the queue, takes no parameters, and returns a usize integer.
- iter() returns an immutable iteration of the queue, the queue remains unchanged, and takes no parameters.
- iter_mut() returns a mutable iteration of the queue, the queue is mutable, and takes no parameters.
- into_iter() changes the queue into an iteration, consumes the queue, and takes no parameters.

Assuming an empty queue q has already been created, the table below shows the expected results of various queue operations, with the front of the queue on the left:

Table 4.4: Queue Operations

operation	value	return
q.is_empty()	[]	true
q.enqueue(1)	[1]	
q.enqueue(2)	[1,2]	
q.enqueue(3)	[1,2,3]	
q.dequeue()	[2,3]	1
q.enqueue(4)	[2,3,4]	
q.enqueue(5)	[2,3,4,5]	
q.deque()	[3,4,5]	2
q.size()	[3,4,5]	3
q.is_empty()	[3,4,5]	false

4.4.2 Implementing a Queue in Rust

To implement a queue in Rust, we can use a Vec with the left end as the rear end of the queue and the right end as the front end. Adding an element to the queue has a complexity of $O(n)$, while removing an element has a complexity of $O(1)$. To prevent the queue from growing infinitely, we can set a cap parameter to limit its length.

```rust
// queue.rs

#[derive(Debug)]
struct Queue<T> {
    cap: usize,   // capacity
    data: Vec<T>, // store elements
}

impl<T> Queue<T> {
    fn new(size: usize) -> Self {
        Self { cap: size, data: Vec::with_capacity(size) }
    }

    fn is_empty(&self) -> bool { 0 == Self::len(&self) }

    fn is_full(&self) -> bool { self.len() == self.cap }

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

    fn clear(&mut self) {
        self.data = Vec::with_capacity(self.cap);
    }

    fn enqueue(&mut self, val: T) -> Result<(), String> {
        if self.len() == self.cap { // check left space
            return Err("No space available".to_string());
        }
        self.data.insert(0, val);
        Ok(())
    }

    // pop out values
    fn dequeue(&mut self) -> Option<T> {
        if self.len() > 0 {
            self.data.pop()
        } else {
            None
        }
    }

    // Implemntion of iterations for a queue
    // into_iter: queue modified and became a iterator
    // iter: queue unmodified and get a unmutable iterator
    // iter_mut: queue unmodified and get a mutable iterator
    fn into_iter(self) -> IntoIter<T> {
        IntoIter(self)
```

```
47      }
48
49      fn iter(&self) -> Iter<T> {
50          let mut iterator = Iter { stack: Vec::new() };
51          for item in self.data.iter() {
52              iterator.stack.push(item);
53          }
54          iterator
55      }
56
57      fn iter_mut(&mut self) -> IterMut<T> {
58          let mut iterator = IterMut { stack: Vec::new() };
59          for item in self.data.iter_mut() {
60              iterator.stack.push(item);
61          }
62          iterator
63      }
64  }
65
66  // Implemention of 3 iterations
67  struct IntoIter<T>(Queue<T>);
68  impl<T: Clone> Iterator for IntoIter<T> {
69      type Item = T;
70      fn next(&mut self) -> Option<Self::Item> {
71          if !self.0.is_empty() {
72              Some(self.0.data.remove(0))
73          } else {
74              None
75          }
76      }
77  }
78
79  struct Iter<'a, T: 'a> { stack: Vec<&'a T>, }
80  impl<'a, T> Iterator for Iter<'a, T> {
81      type Item = &'a T;
82      fn next(&mut self) -> Option<Self::Item> {
83          if 0 != self.stack.len() {
84              Some(self.stack.remove(0))
85          } else {
86              None
87          }
88      }
89  }
90
91  struct IterMut<'a, T: 'a> { stack: Vec<&'a mut T> }
92  impl<'a, T> Iterator for IterMut<'a, T> {
93      type Item = &'a mut T;
94      fn next(&mut self) -> Option<Self::Item> {
95          if 0 != self.stack.len() {
96              Some(self.stack.remove(0))
97          } else {
98              None
```

```
99          }
100     }
101 }
```

```
1  fn main() {
2      basic();
3      iter();
4      fn basic() {
5          let mut q = Queue::new(4);
6          let _r1 = q.enqueue(1); let _r2 = q.enqueue(2);
7          let _r3 = q.enqueue(3); let _r4 = q.enqueue(4);
8          if let Err(error) = q.enqueue(5) {
9              println!("Enqueue error: {error}");
10         }
11         if let Some(data) = q.dequeue() {
12             println!("dequeue data: {data}");
13         } else {
14             println!("empty queue");
15         }
16         println!("empty: {}, len: {}", q.is_empty(), q.len());
17         println!("full: {}", q.is_full());
18         println!("q: {:?}", q);
19         q.clear();
20         println!("{:?}", q);
21     }
22
23     fn iter() {
24         let mut q = Queue::new(4);
25         let _r1 = q.enqueue(1); let _r2 = q.enqueue(2);
26         let _r3 = q.enqueue(3); let _r4 = q.enqueue(4);
27         let sum1 = q.iter().sum::<i32>();
28         let mut addend = 0;
29         for item in q.iter_mut() {
30             *item += 1;
31             addend += 1;
32         }
33         let sum2 = q.iter().sum::<i32>();
34         println!("{sum1} + {addend} = {sum2}");
35         println!("sum = {}", q.into_iter().sum::<i32>());
36     }
37 }
```

Here is the result of the execution.

```
Enqueue error: No space available
dequeue data: 1
empty: false, len: 3
full: false
q: Queue { cap: 4, data: [4, 3, 2] }
Queue { cap: 4, data: [] }
10 + 4 = 14
sum = 14
```

4.4.3 Hot Potato

A queue is commonly used to simulate real-world scenarios where data is processed in a FIFO (first-in, first-out) manner. For instance, the game of hot potato involves children forming a circle and passing a potato quickly to the child next to them, with the objective of keeping the potato moving as if it were hot (see Figure 4.12). When the passing action stops, the child holding the potato is removed from the circle. The game continues until only one child remains. The game is similar to musical chairs, where people perform an action such as speaking, dancing, or leaving their seat when the music stops.

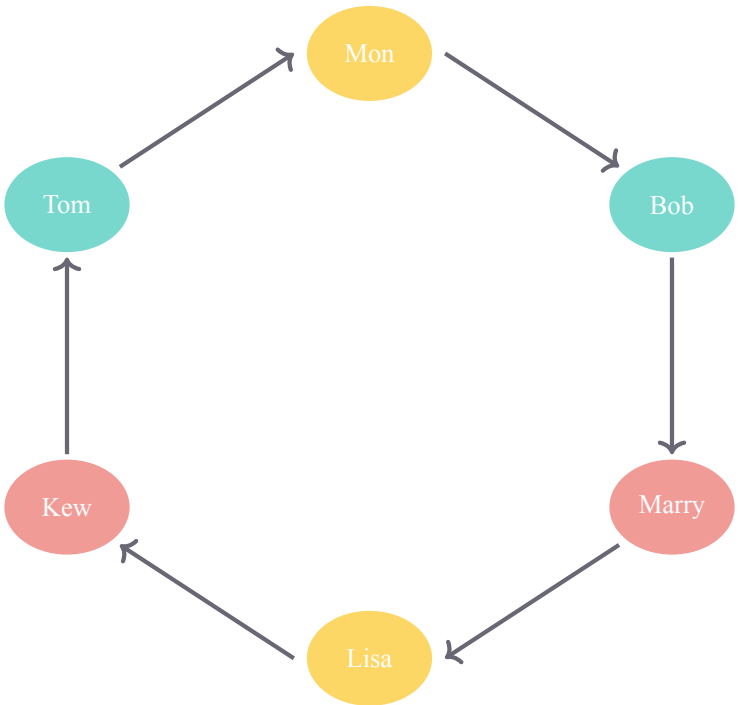


Figure 4.12: hot potato

The game of hot potato is comparable to the Josephus problem, a legendary story recounted by the historian Flavius Josephus. Josephus and his comrades were trapped by the Roman army in a cave and chose to die instead of becoming Roman slaves. They formed a circle and counted clockwise to the eighth person, who was then killed. This continued until only one person was left alive. As a mathematician, Josephus figured out where he should sit to be the last person alive and ultimately joined the Roman side. While there are variations of this story, the fundamental ideas of both the hot potato game and the Josephus problem can be simulated using a queue.

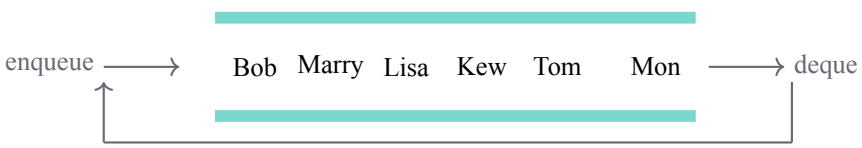


Figure 4.13: Simulation of game of hot potato

To implement the hot potato game, the program accepts multiple names of children and a constant num to set the count of how many children to skip before removing one. Assuming the child holding the hot potato is always at the front of the queue, they leave the queue and re-enter at the back, effectively passing the potato to the next child who must be at the front of the queue. After num rounds of dequeuing

and enqueuing, the child at the front of the queue is permanently removed. The process repeats until only one name remains.

The queue model for the Hot Potato game is clearly illustrated in the two figures above. Here is the implementation of the game based on this model.

```
// hot_potato.rs

fn hot_potato(names: Vec<&str>, num: usize) -> &str {
    // Initialize the queue and enqueue the names
    let mut q = Queue::new(names.len());
    for name in names {
        let _nm = q.enqueue(name);
    }

    while q.size() > 1 {
        // Dequeue and enqueue the names,
        // which simulates passing the potato
        for _i in 0..num {
            let name = q.dequeue().unwrap();
            let _rm = q.enqueue(name);
        }

        // After num dequeue/enqueue cycles
        // remove one person
        let _rm = q.dequeue();
    }

    q.dequeue().unwrap()
}
```

```
fn main() {
    let name = vec!["Mon","Tom","Kew","Lisa","Marry","Bob"];
    let survivor = hot_potato(name, 8);
    println!("The survival person is {survivor}");
    // The survival person is Marry
}
```

It is worth noting that the counting value used in the implementation is 8, which is greater than the number of people in the queue (which is 6). However, this is not a problem because the queue functions like a circle that returns to the head after reaching the tail, and continues until the count is reached. Therefore, there will always be someone dequeued in the end.

4.5 Deque

A deque also is a linear data structure with two ends: the front and the rear. Unlike a queue, a deque allows items to be added and removed from both ends. This flexibility makes it a hybrid linear structure that can function as both a stack and a queue.

Although a deque shares similarities with both a stack and a queue, it does not enforce LIFO or FIFO ordering. The order of adding and removing data determines whether it acts like a stack or a queue. It is important to note that a deque should not be used as a replacement for either a stack or a queue, as each data structure has its own unique properties and is designed for specific computational purposes. The image below provides an example of Deque.

Figure 4.14: Deque

4.5.1 The Deque Abstract Data Type

The deque data structure is an ordered collection of items that allows adding and removing items from either end. It is defined by the following operations:

• new() creates a new deque with no arguments and returns an empty deque.

• add_front(item) adds item to the front of the deque, requiring item as a parameter, and does not return any content.

• add_rear(item) adds item to the back of the deque, requiring item as a parameter, and does not return any content.

• remove_front() removes the front item from the deque, requiring no arguments, returns item, and modifies the deque.

• remove_rear() removes the rear item from the deque, requiring no arguments, returns item, and modifies the deque.

• is_empty() tests whether the deque is empty, requires no arguments, and returns a boolean value.

• size() returns the number of items in the deque, requires no arguments, and returns a usize integer.

• iter() returns an immutable iteration form of the deque, with the deque unchanged and no arguments required.

• iter_mut() returns a mutable iteration form of the deque, with the deque mutable and no arguments required.

• into_iter() changes the deque into an iterable form, with the deque consumed and no arguments required.

Assuming that d is an already created empty deque, the following table shows the results of a series of operations. Note that the front is on the right-hand side. When moving items in and out, it is important to keep track of the front and back contents, as modifying both ends can make the result look somewhat messy.

Table 4.5: Deque Operations

operations	value	return
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.remove_rear()	[1,3,4]	2
d.remove_front()	[1,3]	4
d.size()	[1,3]	2
d.is_empty()	[1,3]	false
d.add_front(2)	[1,3,2]	
d.add_rear(4)	[4,1,3,2]	
d.size()	[4,1,3,2]	4
d.is_empty()	[4,1,3,2]	false
d.add_rear(5)	[5,4,1,3,2]	
d.add_front(10)	[5,4,1,3,2,10]	

4.5.2 Implementing a Deque in Rust

To implement the deque abstract data type in Rust, we can use a Vec as the underlying data structure. The left end of the Vec serves as the rear of the deque, while the right end serves as the front. To prevent the deque from growing indefinitely, a capacity parameter is added to control the maximum length of the deque.

```
// deque.rs

// deque definition
#[derive(Debug)]
struct Deque<T> {
    cap: usize,   // capacity
    data: Vec<T>, // save elements in data
}

impl<T> Deque<T> {
    fn new(cap: usize) -> Self {
        Self {
            cap: cap,
            data: Vec::with_capacity(cap),
        }
    }

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

    fn is_full(&self) -> bool {
        self.len() == self.cap
    }

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

    fn clear(&mut self) {
        self.data = Vec::with_capacity(self.cap);
    }

    // use the tail of a Vec as the start of the deque
    fn add_front(&mut self, val: T) -> Result<(), String> {
        if self.len() == self.cap {
            return Err("No space avaliable".to_string());
        }
        self.data.push(val);

        Ok(())
    }

    // the head of the Vec is the tail of the deque
    fn add_rear(&mut self, val: T) -> Result<(), String> {
        if self.len() == self.cap {
```

```
47              return Err("No space avaliabl".to_string());
48          }
49          self.data.insert(0, val);
50
51          Ok(())
52      }
53
54      // remove data from queue head
55      fn remove_front(&mut self) -> Option<T> {
56          if self.len() > 0 {
57              self.data.pop()
58          } else {
59              None
60          }
61      }
62
63      // remove data from queue tail
64      fn remove_rear(&mut self) -> Option<T> {
65          if self.len() > 0 {
66              Some(self.data.remove(0))
67          } else {
68              None
69          }
70      }
71
72      // Implemention of iteration for the deque
73      // into_iter: deque modified and became a iterator
74      // iter: deque unmodified and get a unmutable iterator
75      // iter_mut: deque unmodified and get a mutable iterator
76      fn into_iter(self) -> IntoIter<T> {
77          IntoIter(self)
78      }
79
80      fn iter(&self) -> Iter<T> {
81          let mut iterator = Iter { stack: Vec::new() };
82          for item in self.data.iter() {
83              iterator.stack.push(item);
84          }
85
86          iterator
87      }
88
89      fn iter_mut(&mut self) -> IterMut<T> {
90          let mut iterator = IterMut { stack: Vec::new() };
91          for item in self.data.iter_mut() {
92              iterator.stack.push(item);
93          }
94
95          iterator
96      }
97  }
98
```

```
// Implementing 3 iterations
struct IntoIter<T>(Deque<T>);
impl<T: Clone> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        // first element of a tuple is not empty
        if !self.0.is_empty() {
            Some(self.0.data.remove(0))
        } else {
            None
        }
    }
}

struct Iter<'a, T: 'a> { stack: Vec<&'a T>, }
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<Self::Item> {
        if 0 != self.stack.len() {
            Some(self.stack.remove(0))
        } else {
            None
        }
    }
}

struct IterMut<'a, T: 'a> { stack: Vec<&'a mut T> }
impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;
    fn next(&mut self) -> Option<Self::Item> {
        if 0 != self.stack.len() {
            Some(self.stack.remove(0))
        } else {
            None
        }
    }
}

fn main() {
    basic();
    iter();

    fn basic() {
        let mut d = Deque::new(4);
        let _r1 = d.add_front(1);
        let _r2 = d.add_front(2);
        let _r3 = d.add_rear(3);
        let _r4 = d.add_rear(4);

        if let Err(error) = d.add_front(5) {
            println!("add_front error: {error}");
        }
```

```
151         println!("{:?}", d);
152
153         match d.remove_rear() {
154             Some(data) => println!("remove rear data {data}"),
155             None => println!("empty deque"),
156         }
157
158         match d.remove_front() {
159             Some(data) => println!("remove front data {data}"),
160             None => println!("empty deque"),
161         }
162         println!("empty: {}, len: {}", d.is_empty(), d.len());
163         println!("full: {}, {:?}", d.is_full(), d);
164
165         d.clear();
166         println!("{:?}", d);
167     }
168
169     fn iter() {
170         let mut d = Deque::new(4);
171         let _r1 = d.add_front(1);
172         let _r2 = d.add_front(2);
173         let _r3 = d.add_rear(3);
174         let _r4 = d.add_rear(4);
175
176         let sum1 = d.iter().sum::<i32>();
177         let mut addend = 0;
178         for item in d.iter_mut() {
179             *item += 1;
180             addend += 1;
181         }
182
183         let sum2 = d.iter().sum::<i32>();
184         println!("{sum1} + {addend} = {sum2}");
185         assert_eq!(14, d.into_iter().sum::<i32>());
186     }
187 }
```

Here is an example output of an execution.

```
add_front error: No space avaliable
Deque { cap: 4, data: [4, 3, 1, 2] }
remove rear data 4
remove front data 2
empty: false, len: 2
full: false, Deque { cap: 4, data: [3, 1] }
Deque { cap: 4, data: [] }
10 + 4 = 14
```

As we can see from the output, a deque has similarities to both a queue and a stack, and can be thought of as a combination of the two. The specific implementation of a deque can vary depending on the situation, and which end is considered the front and which end is considered the rear should be determined based on the context.

4.5.3 Palindrome Checker

Palindromes are strings in which the characters at the same position from both ends are the same, such as ”radar”, ”sos”, and ”rustsur”. In this section, we will present an algorithm for checking whether a given string is a palindrome.

One approach is to use a queue to enqueue the characters of the string and then dequeue the characters to compare them with the reversed original string. This method is simple, but it requires a lot of memory. A more efficient approach is to use a deque.

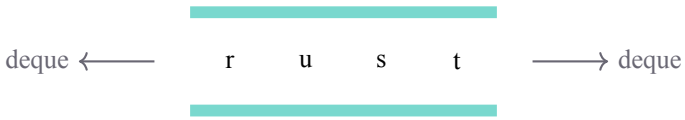


Figure 4.15: Palindrome Checker

To check for palindromes using a deque, we start by processing the input string from left to right and adding each character to the rear of the deque. At this point, the front of the deque holds the first character of the string, while the rear holds the last character. We can then use the deque's feature of dequeuing from both ends to compare the front and rear characters. If they match, we continue dequeuing the front and rear characters until either all the characters are used up, leaving an empty deque, or a deque of size 1 is left. In both cases, the string is a palindrome. Any other situation indicates that the string is not a palindrome. Below is the code implementaion of palindrome detection using a deque.

```
1  // palindrome_checker.rs
2  fn palindrome_checker(pal: &str) -> bool {
3      let mut d = Deque::new(pal.len());
4      for c in pal.chars() {
5          let _r = d.add_rear(c);
6      }
7
8      let mut is_pal = true;
9      while d.size() > 1 && is_pal {
10         let head = d.remove_front();
11         let tail = d.remove_rear();
12         if head != tail {
13             is_pal = false;
14         }
15     }
16     is_pal
17 }
18
19 fn main() {
20     let pal = "rustsur";
21     let is_pal = palindrome_checker(pal);
22     println!("{pal} is palindrome string: {is_pal}");
23     // rustsur is palindrome string: true
24
25     let pal = "panda";
26     let is_pal = palindrome_checker(pal);
27     println!("{pal} is palindrome string: {is_pal}");
28     // panda is palindrome string: false
29 }
```

4.6 LinkedList

An ordered collection of data items is important to maintain the relative position of data and efficient indexing. Arrays and linked lists both allow data to be collected in an ordered manner and saved in relative positions, making them suitable for implementing ordered data types. For example, Rust's default implementation of Vec uses an array. This section focuses on linked lists, which offer some unique advantages.

Unlike arrays, linked lists do not require elements to be stored in contiguous memory. Each item in the collection has a reference to the next item, so the items can be randomly placed without the need to allocate a block of memory, resulting in higher efficiency. To use a linked list, we need to specify the position of the first item explicitly. Once we know the position of the first item, we can determine the position of the second item and so on until the end of the entire linked list.

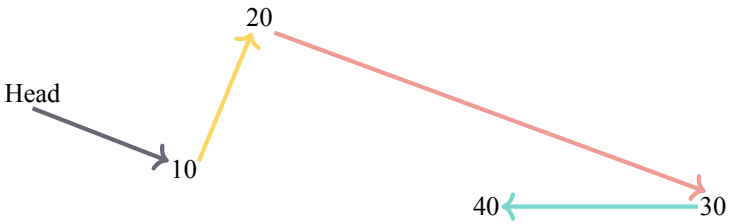


Figure 4.16: Linked list

Usually, linked lists provide a reference to the head of the list, and the last item must be set to null or an empty next item, also known as the tail. With this setup, we can easily traverse the linked list from the head to the tail, or vice versa, by following the references between items. This makes linked lists an efficient data structure for implementing other ordered data types.

4.6.1 The LinkedList Abstract Data Type

The abstract data type of a linked list consists of its structure and operations. As previously mentioned, a linked list is an ordered collection of nodes, and the entire list can be traversed from the head node. The structure of a linked list is simple, with each node containing an element and a reference to the next node. The following definitions and operations apply to a linked list:

- new(): Creates a new head node that points to a Node, takes no arguments, and returns a pointer.
- push(item): Adds a new Node, takes an item parameter, and returns None.
- pop(): Deletes the head node of the linked list, takes no arguments, and returns a Node.
- peek(): Returns a referent to the head node of the linked list, takes no arguments.
- peek_mut(): Returns the head node of the linked list, takes no arguments, and returns a mutable reference to the node.
- is_empty(): Returns a Boolean value indicating whether the current linked list is empty, takes no arguments.
- size(): Returns the length of the linked list as a usize integer value, takes no arguments.
- iter(): Returns an immutable iteration of the linked list without modifying the linked list, takes no arguments.
- iter_mut(): Returns a mutable iteration of the linked list with the linked list mutable, takes no arguments.
- into_iter(): Consumes the linked list and returns an iterable form of the linked list, takes no arguments.

Assuming that l is an empty linked list, the table below shows the result of a sequence of linked list operations, with Link<num> representing the address pointing to the node containing num, and the left end of the list being the head node.

Table 4.6: LinkedList Operations

Operation	Value	Return
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->2->1->None]	mut Link<4>
l.iter()	[4->2->1->None]	[4,2,1]
l.is_empty()	[4->2->1->None]	false
l.size()	[4->2->1->None]	3
l.into_iter()	[None->None]	[4,2,1]

4.6.2 Implementing a LinkedList in Rust

To represent an item in a linked list, we use a node abstraction that includes the data item and the address of the next item. The node also provides methods to access and modify the data item. As shown in the figure below, a Node consists of data and the address of the next node.

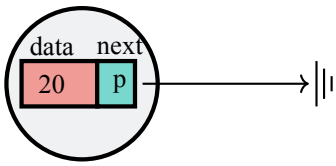


Figure 4.17: Node

In Rust, None is used to represent the absence of a next node in both the Node and the linked list. The new function initializes a grounded node with its next field set to None. Explicitly assigning None to the next field is a good practice that helps avoid dangling pointers, a common issue in languages like C/C++.

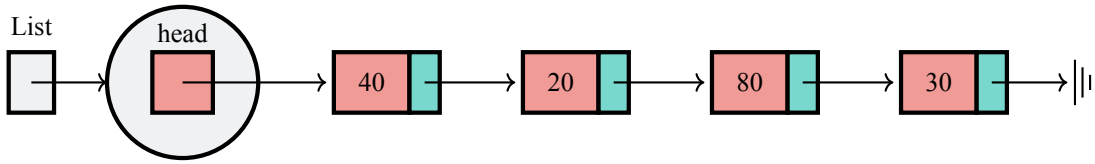


Figure 4.18: LinkedList

The following code shows the implementation of the linked list, with the node link defined as Link for code clarity.

```
1 // linked_list.rs
2
3 // The node link uses a Box pointer (size determined),
4 // because only with a determined size can memory be allocated.
5 type Link<T> = Option<Box<Node<T>>>;
6
```

```
// Linked list definition
struct List<T> {
    size: usize,   // Number of nodes in the linked list
    head: Link<T>, // Head node
}

// Linked list node
struct Node<T> {
    elem: T,       // data
    next: Link<T>, // Link to the next node
}

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

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

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

    fn clear(&mut self) {
        self.size = 0;
        self.head = None;
    }

    // add a new node ahead of the head node
    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 will move data out from node and left a vaccum
    fn pop(&mut self) -> Option<T> {
        self.head.take().map(|node| {
            self.head = node.next;
            self.size -= 1;
            node.elem
        })
    }

    // peek get a unmutable reference
    fn peek(&self) -> Option<&T> {
        self.head.as_ref().map(|node| &node.elem )
    }
```

```
59
60      // peek_mut get a mutable reference
61      fn peek_mut(&mut self) -> Option<&mut T> {
62          self.head.as_mut().map(|node| &mut node.elem )
63      }
64
65      // Implementation of iteration for the linked list.
66      // into_iter: makes the linked list an iterator
67      //            by consuming it
68      // iter: returns an immutable iterator without modifying
69      //       the linked list
70      // iter_mut: returns a mutable iterator without modifying
71      //           the linked list
72      fn into_iter(self) -> IntoIter<T> {
73          IntoIter(self)
74      }
75
76      fn iter(&self) -> Iter<T> {
77          Iter { next: self.head.as_deref() }
78      }
79
80      fn iter_mut(&mut self) -> IterMut<T> {
81          IterMut { next: self.head.as_deref_mut() }
82      }
83  }
84
85  // Implementation of three iterations
86  struct IntoIter<T>(List<T>);
87  impl<T> Iterator for IntoIter<T> {
88      type Item = T;
89      fn next(&mut self) -> Option<Self::Item> {
90          // (List<T>) tuple's 0th item
91          self.0.pop()
92      }
93  }
94
95  struct Iter<'a, T: 'a> { next: Option<&'a Node<T>> }
96  impl<'a, T> Iterator for Iter<'a, T> {
97      type Item = &'a T;
98      fn next(&mut self) -> Option<Self::Item> {
99          self.next.map(|node| {
100             self.next = node.next.as_deref();
101             &node.elem
102         })
103     }
104 }
105
106 struct IterMut<'a, T: 'a> { next: Option<&'a mut Node<T>> }
107 impl<'a, T> Iterator for IterMut<'a, T> {
108     type Item = &'a mut T;
109     fn next(&mut self) -> Option<Self::Item> {
110         self.next.take().map(|node| {
```

```
111             self.next = node.next.as_deref_mut();
112             &mut node.elem
113         })
114     }
115 }
116
117 // Custom implementation of Drop for the linked list
118 impl<T> Drop for List<T> {
119     fn drop(&mut self) {
120         let mut link = self.head.take();
121         while let Some(mut node) = link {
122             link = node.next.take();
123         }
124     }
125 }
126
127 fn main() {
128     basic_test();
129     into_iter_test();
130     iter_test();
131     iter_mut_test();
132
133     fn basic_test() {
134         let mut list = List::new();
135         list.push(1); list.push(2); list.push(3);
136
137         assert_eq!(list.len(), 3);
138         assert_eq!(list.is_empty(), false);
139         assert_eq!(list.pop(), Some(3));
140         assert_eq!(list.peek(), Some(&2));
141         assert_eq!(list.peek_mut(), Some(&mut 2));
142
143         list.peek_mut().map(|val| {
144             *val = 4;
145         });
146
147         assert_eq!(list.peek(), Some(&4));
148         list.clear();
149         println!("basics test Ok!");
150     }
151
152     fn into_iter_test() {
153         let mut list = List::new();
154         list.push(1); list.push(2); list.push(3);
155
156         let mut iter = list.into_iter();
157         assert_eq!(iter.next(), Some(3));
158         assert_eq!(iter.next(), Some(2));
159         assert_eq!(iter.next(), Some(1));
160         assert_eq!(iter.next(), None);
161
162         println!("into_iter test Ok!");
```

```
163     }
164
165     fn iter_test() {
166         let mut list = List::new();
167         list.push(1); list.push(2); list.push(3);
168
169         let mut iter = list.iter();
170         assert_eq!(iter.next(), Some(&3));
171         assert_eq!(iter.next(), Some(&2));
172         assert_eq!(iter.next(), Some(&1));
173         assert_eq!(iter.next(), None);
174         println!("iter test Ok!");
175     }
176
177     fn iter_mut_test() {
178         let mut list = List::new();
179         list.push(1); list.push(2); list.push(3);
180
181         let mut iter = list.iter_mut();
182         assert_eq!(iter.next(), Some(&mut 3));
183         assert_eq!(iter.next(), Some(&mut 2));
184         assert_eq!(iter.next(), Some(&mut 1));
185         assert_eq!(iter.next(), None);
186         println!("iter_mut test Ok!");
187     }
188 }
```

4.6.3 LinkedList Stack

To implement a stack, we can also use a linked list, which is another type of linear data structure. In this case, we can use the head of the linked list to store the top element of the stack. As new items are added to the stack, they will be inserted at the head of the linked list. The implementation for this approach is shown below. Note that since the push and pop functions modify the linked list nodes, we use the take function to extract the node value.

```
1  // list_stack.rs
2
3  // Linked List Node
4  #[derive(Debug, Clone)]
5  struct Node<T> {
6      data: T,
7      next: Link<T>,
8  }
9
10 // Node Self-contained Reference
11 type Link<T> = Option<Box<Node<T>>>;
12
13 impl<T> Node<T> {
14     fn new(data: T) -> Self {
15         Self {
16             data: data,
17             next: None  // No next link when initialized
```

```
18          }
19      }
20  }
21
22  // Linked list stack
23  #[derive(Debug, Clone)]
24  struct LStack<T> {
25      size: usize,
26      top: Link<T>, // Top controls the entire stack
27  }
28
29  impl<T: Clone> LStack<T> {
30      fn new() -> Self {
31          Self {
32              size: 0,
33              top: None
34          }
35      }
36
37      fn is_empty(&self) -> bool {
38          0 == self.size
39      }
40
41      fn len(&self) -> usize {
42          self.size
43      }
44
45      fn clear(&mut self) {
46          self.size = 0;
47          self.top = None;
48      }
49
50      // take out the node on the top, leaving an empty space
51      // that could be filled later
52      fn push(&mut self, val: T) {
53          let mut node = Node::new(val);
54          node.next = self.top.take();
55          self.top = Some(Box::new(node));
56          self.size += 1;
57      }
58
59      fn pop(&mut self) -> Option<T> {
60          self.top.take().map(|node| {
61              let node = *node;
62              self.top = node.next;
63              self.size -= 1;
64              node.data
65          })
66      }
67
68      // Return a reference to the data in the linked list stack
69      fn peek(&self) -> Option<&T> {
```

```
70          self.top.as_ref().map(|node| &node.data)
71      }
72
73      fn peek_mut(&mut self) -> Option<&mut T> {
74          self.top.as_deref_mut().map(|node| &mut node.data)
75      }
76
77      // into_iter: the linked list stack changes
78      // iter: the linked list stack remains unchanged
79      // iter_mut: the linked list stack remains unchanged
80      fn into_iter(self) -> IntoIter<T> {
81          IntoIter(self)
82      }
83
84      fn iter(&self) -> Iter<T> {
85          Iter { next: self.top.as_deref() }
86      }
87
88      fn iter_mut(&mut self) -> IterMut<T> {
89          IterMut { next: self.top.as_deref_mut() }
90      }
91  }
92
93  // Implement three iterations
94  struct IntoIter<T: Clone>(LStack<T>);
95  impl<T: Clone> Iterator for IntoIter<T> {
96      type Item = T;
97      fn next(&mut self) -> Option<Self::Item> {
98          self.0.pop()
99      }
100 }
101
102 struct Iter<'a, T: 'a> { next: Option<&'a Node<T>> }
103 impl<'a, T> Iterator for Iter<'a, T> {
104     type Item = &'a T;
105     fn next(&mut self) -> Option<Self::Item> {
106         self.next.map(|node| {
107             self.next = node.next.as_deref();
108             &node.data
109         })
110     }
111 }
112
113 struct IterMut<'a, T: 'a> { next: Option<&'a mut Node<T>> }
114 impl<'a, T> Iterator for IterMut<'a, T> {
115     type Item = &'a mut T;
116     fn next(&mut self) -> Option<Self::Item> {
117         self.next.take().map(|node| {
118             self.next = node.next.as_deref_mut();
119             &mut node.data
120         })
121     }
```

```
122 }
123
124 fn main() {
125     basic();
126     iter();
127
128     fn basic() {
129         let mut s = LStack::new();
130         s.push(1); s.push(2); s.push(3);
131
132         println!("empty: {:?}", s.is_empty());
133         println!("top: {:?}, size: {}", s.peek(), s.len());
134         println!("pop: {:?}, size: {}", s.pop(), s.len());
135
136         let peek_mut = s.peek_mut();
137         if let Some(data) = peek_mut {
138             *data = 4
139         }
140         println!("top {:?}, size {}", s.peek(), s.len());
141
142         println!("{:?}", s);
143         s.clear();
144         println!("{:?}", s);
145     }
146
147     fn iter() {
148         let mut s = LStack::new();
149         s.push(1); s.push(2); s.push(3);
150
151         let sum1 = s.iter().sum::<i32>();
152         let mut addend = 0;
153         for item in s.iter_mut() {
154             *item += 1;
155             addend += 1;
156         }
157         let sum2 = s.iter().sum::<i32>();
158         println!("{sum1} + {addend} = {sum2}");
159
160         assert_eq!(9, s.into_iter().sum::<i32>());
161     }
162 }
```

The output of the code will depend on how the stack is used and what functions are called.

```
empty: false
top: Some(3), size: 3
pop: Some(3), size: 2
top Some(4), size 2
LStack { size: 2, top: Some(Node { data: 4,
          next: Some(Node { data: 1, next: None }) }) }
LStack { size: 0, top: None }
6 + 3 = 9
```

4.7 Vec

In this chapter, we have demonstrated how the basic data type Vec can be used to implement various abstract data types, including stacks, queues, deques, and linked lists. Vec is a powerful yet simple data container that offers mechanisms for data collection and various operations, which is why we repeatedly use it as the underlying data structure for implementing other data types. It is similar to Python's List and is very convenient to use. However, not all programming languages include Vec, or not all data types may be suitable for it. In some cases, Vec or similar data containers must be implemented separately by programmers.

4.7.1 The Vec Abstract Data Type

Vec is a collection of items that maintain a relative position to other items. Here are the various operations of the Vec abstract data type:

- new(): Creates a new empty Vec, takes no arguments.
- push(item): Adds the item to the end of the Vec, takes item as a parameter.
- pop(): Deletes the last item in the Vec and returns it, take no arguments.
- insert(pos, item): Inserts the item at position pos in the Vec, takes pos and item parameters.
- remove(idx): Removes the item at index idx from the Vec and returns it, takes a parameter.
- find(item): Checks if the item exists in the Vec and returns a boolean value, takes an item parameter.
- is_empty(): Checks if the Vec is empty and returns a boolean value, takes no arguments.
- size(): Calculates the number of items in the Vec and returns a usize integer, takes no arguments.
- iter(): Returns an immutable iteration form of the Vec.
- iter_mut(): Returns a mutable iteration form of the Vec, takes no arguments.
- into_iter(): Consumes the Vec and returns an iterable form of the Vec, takes no arguments.

Assuming v is an empty Vec that has been created, the table below shows the Vec after different operations, with the left side being the head.

Table 4.7: Operations on a Vec

operation	value	return
v.is_empty()	[]	true
v.push(1)	[1]	
v.push(2)	[1,2]	
v.push(3)	[1,2,3]	
v.size()	[1,2,3]	3
v.pop()	[1,2]	3
v.push(5)	[1,2,5]	
v.find(4)	[1,2,5]	false
v.insert(0,8)	[8,1,2,5]	
v.pop()	[8,1,2]	5
v.remove(0)	[1,2]	8
v.size()	[1,2]	2

Finally, it is worth noting that while Vec is a powerful data structure, it may not always be the best choice for a particular use case. Programmers should carefully consider the requirements of their project and choose the appropriate data structure accordingly.

4.7.2 Implementing a Vec in Rust

Vec is composed of linked list nodes, where each node contains a data item and an explicit reference to the next node. Given the location of the first node, subsequent items can be accessed by following the next link.

Since references play a crucial role in Vec, it is necessary to maintain a reference to the first node. The linked list is created with None indicating that the list does not reference any content, as shown in Figure (4.19). The head of the linked list refers to the first node of the list, which stores the address of the next node. It is important to note that the Vec itself does not contain any node objects but only a reference to the first node in the linked list structure.

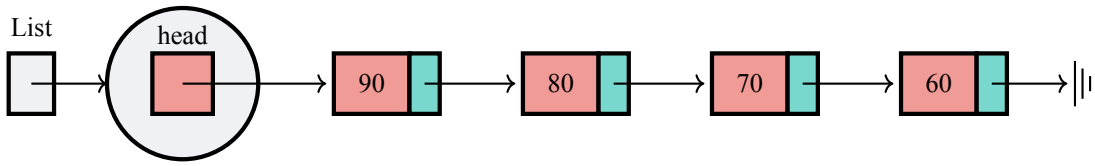


Figure 4.19: LinkedList Formed Vec

To add a new item to the linked list, the only entry point is through the head of the list. As all other nodes can only be accessed by following the next link from the first node, the most efficient way to add a new node is to add it to the head of the linked list. This approach adds the new item as the first element, with the existing items linked to it.

While the implementation of Vec presented here is unordered, it is possible to implement an ordered Vec using a data comparison function with total or partial orders. The following LVec provides only a portion of the functionality of the standard library Vec, and print_lvec is used to print its data items.

```
1  // lvec.rs
2
3  use std::fmt::Debug;
4
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) -> Self {
15         Self {
16             elem: elem,
17             next: None
18         }
19     }
20 }
21
22 // LinkedList Vec definition
23 #[derive(Debug)]
24 struct LVec<T> {
25     size: usize,
26     head: Link<T>,
27 }
28
29 impl<T: Copy + Debug> LVec<T> {
30     fn new() -> Self {
31         Self { size: 0, head: None }
```

```
32      }
33
34      fn is_empty(&self) -> bool {
35          0 == self.size
36      }
37
38      fn len(&self) -> usize {
39          self.size
40      }
41
42      fn clear(&mut self) {
43          self.size = 0;
44          self.head = None;
45      }
46
47      fn push(&mut self, elem: T) {
48          let node = Node::new(elem);
49          if self.is_empty() {
50              self.head = Some(Box::new(node));
51          } else {
52              let mut curr = self.head.as_mut().unwrap();
53
54              // find the last node in the list
55              for _i in 0..self.size-1 {
56                  curr = curr.next.as_mut().unwrap();
57              }
58
59              // insert the new data after the last node
60              curr.next = Some(Box::new(node));
61          }
62          self.size += 1;
63      }
64
65      // add a new LVec to the end of the stack
66      fn append(&mut self, other: &mut Self) {
67          while let Some(node) = other.head.as_mut().take() {
68              self.push(node.elem);
69              other.head = node.next.take();
70          }
71          other.clear();
72      }
73
74      fn insert(&mut self, mut index: usize, elem: T) {
75          if index >= self.size { index = self.size; }
76
77          // three cases for inserting a new node
78          let mut node = Node::new(elem);
79          if self.is_empty() { // LVec is empty
80              self.head = Some(Box::new(node));
81          } else if index == 0 { // insert at the beginning of
                the list
82              node.next = self.head.take();
```

```
83              self.head = Some(Box::new(node));
84          } else { // insert int the middle of the list
85              let mut curr = self.head.as_mut().unwrap();
86              for _i in 0..index - 1 { // find the right insert
                    position
87                  curr = curr.next.as_mut().unwrap();
88              }
89              node.next = curr.next.take();
90              curr.next = Some(Box::new(node));
91          }
92          self.size += 1;
93      }
94
95      fn pop(&mut self) -> Option<T> {
96          if self.size < 1 {
97              return None;
98          } else {
99              self.remove(self.size - 1)
100         }
101     }
102
103     fn remove(&mut self, index: usize) -> Option<T> {
104         if index >= self.size { return None; }
105
106         // two cases for deleting a node
107         let mut node;
108         if 0 == index {
109             node = self.head.take().unwrap();
110             self.head = node.next.take();
111         } else { // find the position which will be deleteed
                and arrange the links properly
112             let mut curr = self.head.as_mut().unwrap();
113             for _i in 0..index - 1 {
114                 curr = curr.next.as_mut().unwrap();
115             }
116             node = curr.next.take().unwrap();
117             curr.next = node.next.take();
118         }
119         self.size -= 1;
120
121         Some(node.elem)
122     }
123
124     fn into_iter(self) -> IntoIter<T> {
125         IntoIter(self)
126     }
127
128     fn iter(&self) -> Iter<T> {
129         Iter { next: self.head.as_deref() }
130     }
131
132     fn iter_mut(&mut self) -> IterMut<T> {
```

```
133         IterMut { next: self.head.as_deref_mut() }
134     }
135
136     // print LVec
137     fn print_lvec(&self) {
138         if 0 == self.size {
139             println!("Empty lvec");
140         }
141         for item in self.iter() {
142             println!("{:?}", item);
143         }
144     }
145 }
146
147 // Implemtion of three iterations
148 struct IntoIter<T: Copy + Debug >(LVec<T>);
149 impl<T: Copy + Debug> Iterator for IntoIter<T> {
150     type Item = T;
151     fn next(&mut self) -> Option<Self::Item> {
152         self.0.pop()
153     }
154 }
155
156 struct Iter<'a, T: 'a> { next: Option<&'a Node<T>> }
157 impl<'a, T> Iterator for Iter<'a, T> {
158     type Item = &'a T;
159     fn next(&mut self) -> Option<Self::Item> {
160         self.next.map(|node| {
161             self.next = node.next.as_deref();
162             &node.elem
163         })
164     }
165 }
166
167 struct IterMut<'a, T: 'a> { next: Option<&'a mut Node<T>> }
168 impl<'a, T> Iterator for IterMut<'a, T> {
169     type Item = &'a mut T;
170     fn next(&mut self) -> Option<Self::Item> {
171         self.next.take().map(|node| {
172             self.next = node.next.as_deref_mut();
173             &mut node.elem
174         })
175     }
176 }
177
178 fn main() {
179     basic();
180     iter();
181
182     fn basic() {
183         let mut lvec1: LVec<i32> = LVec::new();
184         lvec1.push(10); lvec1.push(11);
```

```
185         lvec1.push(12); lvec1.push(13);
186         lvec1.insert(0,9);
187
188         lvec1.print_lvec();
189
190         let mut lvec2: LVec<i32> = LVec::new();
191         lvec2.insert(0, 8);
192         lvec2.append(&mut lvec1);
193
194         println!("len: {}", lvec2.len());
195         println!("pop {:?}", lvec2.pop().unwrap());
196         println!("remove {:?}", lvec2.remove(0).unwrap());
197
198         lvec2.print_lvec();
199         lvec2.clear();
200         lvec2.print_lvec();
201     }
202
203     fn iter() {
204         let mut lvec: LVec<i32> = LVec::new();
205         lvec.push(10); lvec.push(11);
206         lvec.push(12); lvec.push(13);
207
208         let sum1 = lvec.iter().sum::<i32>();
209         let mut addend = 0;
210         for item in lvec.iter_mut() {
211             *item += 1;
212             addend += 1;
213         }
214         let sum2 = lvec.iter().sum::<i32>();
215         println!("{sum1} + {addend} = {sum2}");
216
217         assert_eq!(50, lvec.into_iter().sum::<i32>());
218     }
219 }
```

Here is the execution result.

```
9
10
11
12
13
len: 6
pop 13
remove 8
9
10
11
12
Empty lvec
46 + 4 = 50
```

It's important to note that LVec is a linked list with n nodes, and any method that requires traversing nodes, such as insert, push, pop, remove, etc., has a time complexity of $O(n)$. Although on average it may only require traversing half as many nodes, in the worst case, every node in the list must be processed.

4.8 Summary

This chapter focuses on several linear data structures, including stack, queue, double-ended queue, linkedList, and Vec.

A stack is a type of data structure that maintains last-in-first-out (LIFO) ordering and has basic operations such as push, pop, and is_empty. Stacks are useful for designing algorithms that compute parsed expressions and can provide inversion properties that are useful in implementing operating system function calls, website save functions, and more. Prefix, infix, and postfix expressions can all be handled with a stack, but computers do not typically use infix expressions.

A queue is a simple data structure that maintains a first-in-first-out (FIFO) sorting property and has basic operations such as enqueue, dequeue, and is_empty. Queues are very useful in scheduling system tasks and can help build timed task emulations.

A double-ended queue(Deque) is a data structure that allows a mixed behavior of stack and queue. Its basic operations include is_empty, add_front, add_rear, remove_front, and remove_rear. Although a double-ended queue can be used as a stack or a queue, it is recommended to use it only as a double-ended queue.

A linkedList is a collection of items, each of which is stored in a relative position in the list. The implementation of a linked list itself maintains logical order and does not need to be stored in physical order. Modifying the head of a linked list is a special case.

Vec is a data container that comes with Rust, and the default implementation uses dynamic arrays. In this chapter, however, we use a linked list for this purpose.

Chapter 5

Recursion

5.1 Objectives

- Understanding simple recursive solutions
- Learning how to write programs using recursion
- Understanding and applying the three laws of recursion
- Understanding recursion as a form of iteration
- Formulating problems into recursive solutions
- Understanding how computers implement recursion

5.2 What is Recursion?

Recursion is a problem-solving method that involves breaking down a complex problem into smaller sub-problems until a simple base case is reached. By solving the base case and combining the results, the solution to the original problem can be obtained. Recursion is particularly useful for solving difficult problems elegantly.

For instance, consider the task of computing the sum of an integer array like [2, 1, 7, 4, 5]. The most straightforward way to do it is by using an accumulator to add each value iteratively.

```
1  // iterative sum
2
3  fn nums_sum(nums: Vec<i32>) -> i32 {
4      // use sum as accumulator
5      let mut sum = 0;
6
7      for num in nums {
8          sum += num;
9      }
10
11     sum
12 }
```

However, if a programming language lacks for or while loops, the above sum code won't work. In such cases, one can still compute the sum by using recursion to solve smaller problems. To solve the sum problem recursively, you can break down the array into smaller additions, which can be constructed using the basic problem of addition. By recursively solving these smaller problems, you can eventually solve the original sum problem without loops.

The process of constructing small additions can be done using elementary school-level knowledge of constructing fully parenthesized expressions, of which there are of course two forms as shown below.

$$\begin{aligned} 2+1+7+4+5 &= 2+1+7+4+5 \\ sum &= ((((2+1)+7)+4)+5) \\ sum &= (((3+7)+4)+5) \\ sum &= ((10+4)+5) \\ sum &= (14+5) \\ sum &= 19 \\ &= 2+1+7+4+5 \\ sum &= (2+(1+(7+(4+5)))) \\ sum &= (2+(1+(7+9))) \\ sum &= (2+(1+16)) \\ sum &= (2+17) \\ sum &= 19 \end{aligned} \tag{5.1}$$

Overall, recursion provides an elegant and effective solution to seemingly difficult problems, allowing for efficient computation and solving programming challenges even in languages with limited capabilities.

The given expression can be correctly parenthesized using either form of parentheses on the right-hand side. By following the rule of parentheses precedence and internal priority, we can treat the parenthesized expression as a sequence of small additions. The pattern of both parts and the entire expression is entirely recursive, and we can simulate it without using While or For loops.

If we observe the calculation starting with ”sum” and read it from bottom to top, we can see that the first term is 19, followed by (2 + 17), and then (2 + (1 + 16)). The total sum is the sum of the first term and the remaining terms on the right-hand side. We can further decompose it into the sum of its first term and the remaining terms on the right-hand side. Therefore, we can express this mathematically as follows:

$$Sum(nums) = First(nums) + Sum(restR(nums)) \tag{5.2}$$

This is the calculation method for the second form of parenthesized expressions. Similarly, we also have a calculation method for the first form of parenthesized expressions.

$$Sum(nums) = Last(nums) + Sum(restL(nums)) \tag{5.3}$$

The equation defines various functions such as First(nums), restR(nums), Last(nums), and restL(nums) that return the required elements from the array.

We can implement two recursive methods to calculate expressions in Rust. The nums_sum1 function sums the remaining terms after nums[0], while the nums_sum2 function sums the last term and all the terms before it. Both implementations have the same time and space complexities and do not differ from each other.

```
1  // nums_sum12.rs
2
3  // Form : Sum(nums) = First(nums) + Sum(restR(nums))
4  fn nums_sum1(nums: &[i32]) -> i32 {
5      if 1 == nums.len() {
6          nums[0]
7      } else {
8          let first = nums[0];
9          first + nums_sum1(&nums[1..])
10     }
```

```
}

// Form : Sum(nums) = Last(nums) + Sum(restL(nums))
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 = {sum1}, sum2 = {sum2}");
    // sum1 = 19, sum2 = 19

    let nums = [-1,7,1,2,5,4,10,100];
    let sum1 = nums_sum1(&nums);
    let sum2 = nums_sum2(&nums);
    println!("sum1 = {sum1}, sum2 = {sum2}");
    // sum1 = 128, sum2 = 128
}
```

The crucial part of the code is the if and else statements and their respective forms. The condition of if 1 == nums.len() is crucial as it marks the turning point of the function. At this point, the numerical value is directly returned without any further mathematical calculation. In the else statement, the function calls itself, achieving the effect of solving parentheses layer by layer and calculating their values. This is the essence of recursion, where a function calls itself until it reaches a base case.

5.2.1 The Three Laws of Recursion

By analyzing the above code, we can see that all recursive algorithms must follow these three basic laws:

```
Recursive algorithms must have a base case
Recursive algorithms must move towards the base case
Recursive algorithms must call themselves in a recursive manner
```

The first law is the base case, which is when the algorithm stops, in this case, 1 == nums.len(). The second law involves the decomposition of the problem, where we return the number in the else clause and then calculate it again using the set of elements except for the returned number. This reduces the number of elements in the original array nums that need to be calculated. It is evident that the else clause moves towards the base case(nums.len == 1), which occurs when nums.len() > 1. The third law involves calling itself, which is implemented in the else clause. It is important to note that calling itself is not a loop, and there are no while or for statements here. In summary, our recursive sum algorithm satisfies these three laws.

The recursive function call process can be visualized using a series of boxes representing the function call graph. Each recursion solves a small problem until the problem reaches the base case, where it can no longer be further decomposed. Finally, these intermediate calculated values are used to solve the larger problem.

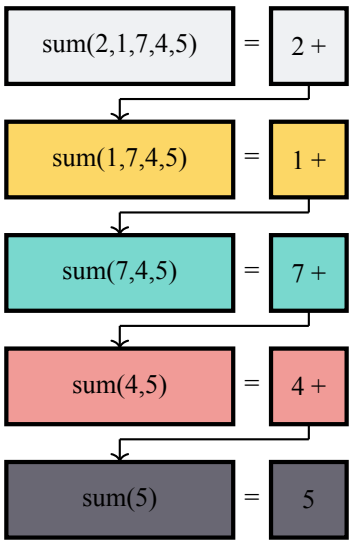


Figure 5.1: caculation by iteration

5.2.2 Converting an Integer to a String in Any Base

In the previous section, we discussed implementing algorithms for converting integers into binary and hexadecimal strings using stacks. However, recursive methods can also be used for base conversion. To design a recursive algorithm for base conversion, we must adhere to the three laws of recursion.

```
1 The recursive algorithm must have a base case
2 The recursive algorithm must move towards the base case
3 The recursive algorithm must call itself
```

The algorithm for base conversion can be derived as follows:

```
1 Simplify the original number into a series of individual digits
2 Use a lookup method to convert each digit into a character
3 Concatenate the individual characters to form the final result
```

To move towards the base case, we use division to change the number state. We divide the number by the base and obtain a remainder and a quotient. If the quotient is less than the base, we stop the operation and return the result. For instance, consider the decimal number 996 and converting it into a base-10 string. We divide 996 by 10, and the remainder is 6, and the quotient is 99. Since the remainder is less than 10, we can obtain the character '6'. The quotient 99 is less than 996, which moves us closer to the base case. We then recursively call the function with 99 as the input. We divide 99 by 10, and the remainder is 9, and the quotient is 9. Since the remainder is less than 10, we can obtain the character '9'. Finally, we get the decimal string '996'. Once we can solve base-10 conversion, we can easily handle base-2 to base-16 conversion.

Here's an algorithm for converting integers to any base (base-2 to base-16) string. The variable BASESTR holds the character representation for different digits, with digits greater than 10 represented by characters A-F.

```
1 // num2str_rec.rs
2
3 //Character table for various digits
4 const BASESTR: [&str; 16] = ["0","1","2","3","4","5","6","7",
5                              "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 {
        // Append remainder to the end of the string
        num2str_rec(num/base, base) +
            BASESTR[(num % base) as usize]
    }
}

fn main() {
    let num = 100;
    let sb = num2str_rec(num,2);  // sb = str_binary
    let so = num2str_rec(num,8);  // so = str_octal
    let sh = num2str_rec(num,16); // sh = str_hexdecimal
    println!("{num} = b{sb}, o{so}, x{sh}");
    // 100 = b1100100, o144, x64

    let num = 1000;
    let so = num2str_rec(num,8);
    let sh = num2str_rec(num,16);
    println!("{num} = o{so}, x{sh}");
    // 1000 = o1750, x3E8
}
```

In the previous section, we utilized a stack to convert numbers to any base. Now, we still achieve similar functionality using recursion, indicating a relationship between stacks and recursion. Recursion can be viewed as a stack, but the stack is implicitly called by the compiler. The code only uses recursion, but the compiler employs a stack to save data. If we were to implement the following recursive code using a stack, the structure would be very similar.

```
// 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();
    while num > 0 {
        if num < base {
            // Push directly to stack if not exceeding the base
            rem_stack.push(num);
        } else {
            // Push remainder to stack if exceeding the base
            rem_stack.push(num % base);
        }
        num /= base;
    }

    // Pop remainder from stack and form a string
    let mut numstr = "".to_string();
    while !rem_stack.is_empty() {
        numstr += digits[rem_stack.pop().unwrap() as usize];
```

```
23      }
24
25      numstr
26  }
27
28  fn main() {
29      let num = 100;
30      let sb = num2str_stk(100, 2);
31      let so = num2str_stk(100, 8);
32      let sh = num2str_stk(100, 16);
33      println!("{num} = b{sb}, o{so}, x{sh}");
34      // 100 = b1100100, o144, x64
35  }
```

5.2.3 Tower of Hanoi

The Tower of Hanoi puzzle was invented by French mathematician Edouard Lucas in 1883. The puzzle involves moving 64 gold discs of decreasing size from one of three poles to another, one disc at a time, with the constraint that a larger disc cannot be placed on a smaller one. The inspiration for this puzzle came from a legend about a puzzle given to young priests by the abbot of an Indian Hindu temple. In reality, moving all 64 discs would take 5.85 trillion years, longer than the known age of the universe.

Figure (5.2) illustrates the process of moving disks from the first peg to the third peg, which resembles a stack. If you haven't played this before, you can try it now. You don't need disks, a pile of bricks, books, or paper will do, and you don't need 64, 10 is enough. See if it really takes that long.

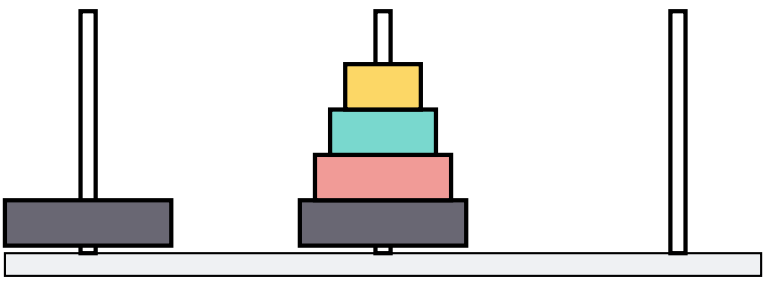


Figure 5.2: Hanoi

To solve this problem recursively, we need to determine the base case. Suppose there is a tower of Hanoi with three pegs: left, middle, and right, and five disks on the left peg. We can assume that we know how to move one disk to the right. This is the base case. If we know how to move n-1 disks to the middle peg, we can easily move the bottom disk to the right peg and then move the n-1 disks from the middle peg to the right peg. If we don't know how to move n-1 disks to the middle peg, we can assume that we know how to move n-2 disks to the middle peg, then move the n-1th disk to the right peg, and then move the n-2 disks from the middle peg to the top of the n-1th disk on the right peg. This process is an abstraction of the disk movement process.

In summary, we can organize the above operation process into the following algorithm to solve the Tower of Hanoi puzzle recursively.

```
1  Move height-1 disks to the middle peg using the target peg as a
       helper.
2  Move the last disk to the target peg.
3  Move height-1 disks from the middle peg to the target peg using
       the starting peg as a helper.
```

To solve the Towers of Hanoi problem, the recursive three laws must be followed to ensure that larger discs remain at the bottom of the stack. The base case is when there is only one disc, which can be moved to its final destination. The algorithm then reduces the height of the Towers of Hanoi through steps 1 and 3, moving towards the base case. The solution can be implemented in a few lines of Rust code using recursion, as shown below.

```
// hanoi.rs

// p: pole
fn hanoi(height:u32, src_p:&str, des_p:&str, mid_p:&str) {
    if height >= 1 {
        hanoi(height - 1, src_p, mid_p, des_p);
        println!("move disk[{height}] from
                    {src_p} to {des_p}");
        hanoi(height - 1, mid_p, des_p, src_p);
    }
}

fn main() {
    hanoi(1, "A", "B", "C");
    hanoi(2, "A", "B", "C");
    hanoi(3, "A", "B", "C");
    hanoi(4, "A", "B", "C");
    hanoi(5, "A", "B", "C");
    hanoi(6, "A", "B", "C");
}
```

To simulate the problem, one can use three pens and paper to move the discs according to the output of the hanoi function for heights 1, 2, 3, and 4.

5.3 Tail Recursion

The aforementioned recursive computation is called regular recursion, which involves saving values during the computation process. This can lead to stack overflow if there are too many recursive calls and memory becomes limited. To avoid this, tail recursion can be used. Tail recursion processes the current result of the operation as a parameter for the next recursive function call, which reduces the memory consumption on the stack and makes the code more concise.

Tail recursion optimization is particularly useful when dealing with subproblems that need to be summed. It reduces the problem size by calculating the sum of the parameters beforehand, which optimizes the algorithm. However, tail-recursive code may be more difficult to understand since the results of subproblems are directly used as parameters for the next recursive call. The regular recursive version of the code for summing the numbers can be converted into a more concise tail-recursive form.

```
// nums_sum34.rs

fn nums_sum3(sum: i32, nums: &[i32]) -> i32 {
    if 1 == nums.len() {
        sum + nums[0]
    } else {
        nums_sum3(sum + nums[0], &nums[1..])
    }
```

```
}

fn nums_sum4(sum: i32, nums: &[i32]) -> i32 {
    if 1 == nums.len() {
        sum + nums[0]
    } else {
        nums_sum4(sum + nums[nums.len() - 1],
                  &nums[..nums.len() - 1])
    }
}

fn main() {
    let nums = [2,1,7,4,5];
    let sum1 = nums_sum3(0, &nums);
    let sum2 = nums_sum4(0, &nums);
    println!("sum1 is {sum1}, sum2 is {sum2}");
    // sum1 is 19, sum2 is 19
}
```

Ultimately, the implementation of recursive programs depends on the programmer's ability to write clear and efficient code. If tail recursion does not cause stack overflow and is easy to understand, it can be a useful tool in algorithm optimization.

5.3.1 Recursion VS Iteration

To compute the array [2, 1, 7, 4, 5], we employed three methods: loop iteration, recursion, and tail recursion. These methods demonstrate that recursion and iteration can accomplish the same goal. So, what is the relationship between recursion and iteration?

• Recursion refers to a process of repeating a transaction in a self-similar way. In mathematics and computer science, it involves using the function itself in its definition. When recursion is unfolded, it creates a tree-like structure. This can be thought of as a process of repetitive recursion and backtracking. Once the recursion reaches the bottom, backtracking begins, which is equivalent to a depth-first traversal of the tree.

• On the other hand, iteration involves repeating a feedback process, where the outcome of each iteration becomes the starting point for the next iteration. Iteration is a loop structure, starting from the initial state, where each iteration traverses the loop and updates the state. Multiple iterations continue until the end state is reached.

While all iterations can be converted into recursion, converting recursion to iteration is not always possible because changing a tree into a loop is not always feasible. Therefore, it is essential to choose the appropriate method for a given problem, taking into account its complexity, readability, and efficiency.

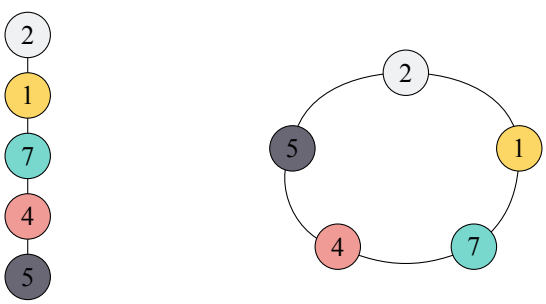


Figure 5.3: Recursion VS Iteration

5.4 Dynamic Programming

In computer science, optimization problems that involve finding the maximum or minimum value of a function are common. These problems are crucial for achieving carbon neutrality, saving energy, and improving the passenger experience. This section aims to demonstrate different solutions to optimization problems and highlight dynamic programming as an effective approach for solving such problems.

A typical example of an optimization problem is the ”minimum number of bills/coins to make change” problem, commonly seen in subway ticketing and vending machines. The objective is to minimize the number of bills/coins returned for a single transaction. For example, returning two bills (a five and a one) for a six dollars transaction, instead of six one-dollar bills.

The problem is how to plan the return of different denominations of bills/coins from a total task, such as making change for six dollars. The most intuitive approach is the greedy method, which starts with the largest denomination of bills/coins and uses as many of them as possible, then moves on to the next smaller denomination and repeats the process until the change is complete.

However, the greedy method may not always yield the optimal solution. For instance, if a country has a complex currency system that includes not only the usual 1, 5, 10, and 25 dollar bills/coins, but also a 22 dollar bill/coin, the greedy method cannot find the best solution for making change for 66 dollars. Even with the addition of the 22 dollars bill/coin, the greedy method still finds a solution, but it requires five bills (two 25 dollar bills, one 10 dollar bill, one 5 dollar bill, and one 1 dollar bill), while the optimal answer is three 22 dollar bills.

To solve the change-making problem, recursion is an effective method. The first step is to identify the basic problem, which is to make change for one bill/coin with an unspecified denomination but equal to the total amount needed. This is because making change with only one bill/coin represents the minimum number of bills/coins required except when the amount is zero. Assuming US dollars bills with denominations of 1, 5, 10, 20, and 50 dollar are used, the number of bills needed to make change for the original amount can be calculated by adding one to the number of bills needed for the remaining amount after subtracting the denomination from the total amount. Therefore, the formula for calculating the number of bills needed is straightforward.

$$numBills(amount) = \begin{cases} 1 + numBills(amount - 1) \\ 1 + numBills(amount - 5) \\ 1 + numBills(amount - 10) \\ 1 + numBills(amount - 20) \\ 1 + numBills(amount - 50) \end{cases} \tag{5.4}$$

To implement this approach, the numBills function is used to calculate the number of bills required for a given amount of change. The function considers five cases, where the change can be made up of different denominations of bills. The algorithm checks if the change amount is equal to a denomination of bill on line 8, which serves as a base case. If not, the function recursively calls itself with a reduced change amount and the next smallest denomination of bill. The count of bills used is incremented before each recursive call to keep track of the total number of bills used in the final solution

```
1  // rec_mc1.rs
2
3  fn rec_mc1(cashes: &[u32], amount: u32) -> u32 {
4      // Calculates the minimum number of coins required for
5      // change when using only 1-yuan bills.
6      let mut min_cashes = amount;
7
8      if cashes.contains(&amount) {
9          return 1;
10     } else {
```

```
11          // Extracts valid denominations (denominations must
12          // be smaller than the change amount).
13          for c in cashes.iter()
14                          .filter(|&&c| c <= amount)
15                          .collect::<Vec<&u32>>() {
16              // Subtracts c from amount to indicate that a bill
17              // of denomination c has been used,
18              // and increments num_cashes by 1.
19              let num_cashes = 1 + rec_mc1(&cashes, amount - c);
20
21              // If num_cashes is smaller than min_cashes,
22              // update min_cashes.
23              if num_cashes < min_cashes {
24                  min_cashes = num_cashes;
25              }
26          }
27      }
28
29      min_cashes
30  }
31
32  fn main() {
33      // cashes stores denominations of bills.
34      let cashes = [1,5,10,20,50];
35      let amount = 31u32;
36      let cashes_num = rec_mc1(&cashes, amount);
37      println!("need refund {cashes_num} cashes");
38      // need refund 3 cashes
39  }
```

To avoid lengthy wait times for the program to return a result, it is important to optimize the code by reducing redundant calculations. In the given code, changing the amount to 90 results in a long wait time because the program needs to make a large number of recursive calls to find the combination of three bills: [50, 20, 20].

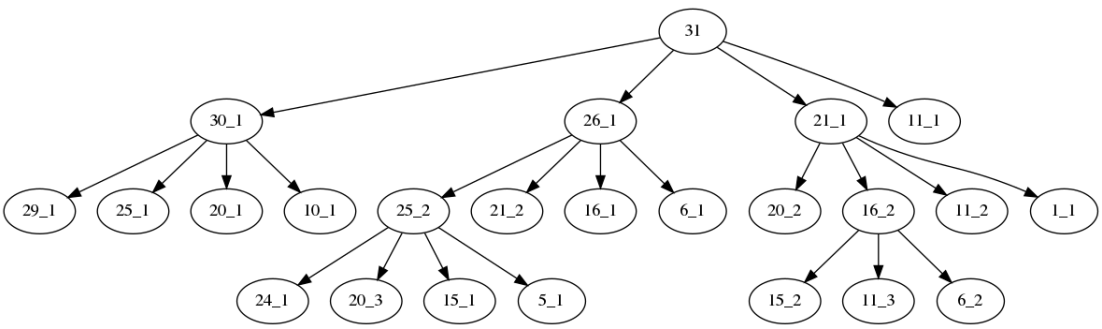


Figure 5.4: solve the task by recursion

To address this issue, one solution is to store previously calculated results to avoid redundant calculations. One approach is to store the current minimum number of bills in a list and check this list before calculating a new minimum value. If the result already exists, use the stored value instead of recalculating it. This technique is an example of trading space for time in algorithm design.

```rust
// rc_mc2.rs

fn rec_mc2(cashes: &[u32], amount: u32,
           min_cashes: &mut [u32]) -> u32
{
    // Calculates the minimum number of coins required for
    // change when using only 1-yuan bills.
    let mut min_cashe_num = amount;

    if cashes.contains(&amount) {
        // Collects denominations that match the
        // current change value.
        min_cashes[amount as usize] = 1;
        return 1;
    } else if min_cashes[amount as usize] > 0 {
        // If the change amount already has the minimum number
        // of coins required for change, return directly.
        return min_cashes[amount as usize];
    } else {
        for c in cashes.iter()
                       .filter(|&&c| c <= amount)
                       .collect::<Vec<&u32>>() {

            let cashe_num = 1 + rec_mc2(cashes,
                                        amount - c,
                                        min_cashes);

            // Updates the minimum number of coins
            // required for change.
            if cashe_num < min_cashe_num {
                min_cashe_num = cashe_num;
                min_cashes[amount as usize] = min_cashe_num;
            }
        }
    }

    min_cashe_num
}

fn main() {
    let amount = 90u32;
    let cashes: [u32; 5] = [1,5,10,20,50];
    let mut min_cashes: [u32; 91] = [0; 91];
    let cashe_num = rec_mc2(&cashes, amount, &mut min_cashes);
    println!("need refund {cashe_num} cashes");
    // need refund 3 cashes
}
```

The rec_mc2 is less time-consuming because it uses the variable min_cashes to store intermediate values. It is important to note that although this section discusses dynamic programming, both programs presented are recursive rather than dynamic programming. The second program only saves intermediate values during recursion, using a memory or cache technique to reduce the time required for calculations.

5.4.1 What is Dynamic Programming?

Dynamic programming(DP) is a mathematical technique for solving optimization problems in decision-making processes. The change-making problem described above is an optimization problem that seeks to minimize the number of bills/coins used, and this is the optimization objective.

The key to dynamic programming is identifying whether a problem can be solved by this technique and expressing the problem as a state that can be transitioned between. The state is critical to the calculation and is used to derive the next step in the problem-solving process.

Unlike the previous greedy algorithm, which goes from large to small denominations, dynamic programming assumes that the current best result has been obtained and deduces the next step based on this result. The technique decomposes a large problem into smaller sub-problems and deduces from small to large problems. The intermediate values of the deducing process are cached, and this process is known as state transition.

For the change-making problem, three parameters are required for dynamic programming: a list of available bills, the amount to be changed, and a list that contains the minimum number of bills needed for each amount. The algorithm saves the calculated values to the list, starting from the smallest amount and gradually building up to the desired change amount. The implementation of the algorithm uses iteration.

By using dynamic programming, redundant calculations can be avoided, and a significant improvement in efficiency can be achieved. The program caches previously calculated results, which reduces the workload and processing time.

```rust
// dp_rec_mc.rs

fn dp_rec_mc(cashes: &[u32], amount: u32,
             min_cashes: &mut [u32]) -> u32 {
    // Collect the minimum number of coins required to
    // make change from 1 to amount, and then reconstructs
    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;
    }
    // directly return the result
    min_cashes[amount as usize]
}

fn main() {
    let amount = 90u32;
    let cashes = [1,5,10,20,50];
    let mut min_cashes: [u32; 91] =  [0; 91];
    let cash_num = dp_rec_mc(&cashes,amount,&mut min_cashes);
    println!("Refund for ${amount} need {cash_num} cashes");
    // Refund for $90 need 3 cashes
}
```

The iterative dynamic programming code is much more concise than the previous recursive versions and reduces the use of the stack. However, it is important to note that just because a problem can be solved using recursion does not mean it is the best solution.

While the dynamic programming algorithm finds the minimum number of bills required, it does not indicate which denominations of bills are used. To obtain this information, a table, cashes_used, can be added to record the denominations and quantities of bills used. The algorithm can be extended by adding the denomination of the last bill used for each amount to the cashes_used table and then continuously finding the last bill used for the previous amount until the end.

```
// dp_rc_mc_show.rs

// The algorithm uses cashes_used to collect
// the denominations of coins used,
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 ;
        // With a minimum denomination of 1 yuan
        let mut used_cashe = 1;
        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;
            }
        }
        // update the minimum number of coins
        // required for each amount
        min_cashes[denm as usize] = min_cashe_num;
        cashes_used[denm as usize] = used_cashe;
    }

    min_cashes[amount as usize]
}

// prints the denominations of coins used
fn print_cashes(cashes_used: &[u32], mut amount: u32) {
    while amount > 0 {
        let curr = cashes_used[amount as usize];
        println!("${curr}");
        amount -= curr;
    }
}

fn main() {
    let amount = 81u32; let cashes = [1,5,10,20,50];
    let mut min_cashes: [u32; 82] = [0; 82];
    let mut cashes_used: [u32; 82] = [0; 82];
```

```
44      let cs_num = dp_rec_mc_show(&cashes, amount,
45                                  &mut min_cashes,
46                                  &mut cashes_used);
47      println!("Refund for ${amount} need {cs_num} cashes:");
48      print_cashes(&cashes_used, amount);
49  }
```

For instance, when making change for 90 dollars, only three bills ([50, 20, 20]) are needed.

```
Refund for $90 requires 3 cashes:
$20
$20
$50
```

5.4.2 Dynamic Programming VS Recursion

Recursion and dynamic programming are two different techniques used to solve problems. Recursion breaks down a large problem into smaller ones and solves them by calling itself, while dynamic programming solves large problems using solutions to smaller problems. Recursion can be memory-intensive and lead to stack overflow, while dynamic programming requires determining transition rules and initial conditions, but has simple code and a steep learning curve. Although both techniques can solve the same problems, such as the Fibonacci sequence problem.

```
1   // fibnacci_dp_rec.rs
2
3   fn fibnacci_dp(n: u32) -> u32 {
4       // Using only two positions to store values can save memory
5       let mut dp = [1, 1];
6       for i in 2..=n {
7           let idx1 = (i % 2) as usize;
8           let idx2 = ((i - 1) % 2) as usize;
9           let idx3 = ((i - 2) % 2) as usize;
10          dp[idx1] = dp[idx2] + dp[idx3];
11      }
12
13      dp[((n-1) % 2) as usize]
14  }
15
16  fn fibnacci_rec(n: u32) -> u32 {
17      if n == 1 || n == 2 {
18          return 1;
19      } else {
20          fibnacci_rec(n-1) + fibnacci_rec(n-2)
21      }
22  }
23
24  fn main() {
25      println!("fib(10): {}", fibnacci_dp(10));
26      println!("fib(10): {}", fibnacci_rec(10));
27      // fib(10): 55
28      // fib(10): 55
29  }
```

It is important to note that a problem that can be solved with dynamic programming may not necessarily be solvable with recursion as they have different requirements.

5.5 Summary

In this chapter, we covered both recursive and iterative algorithms. Recursive algorithms must satisfy three laws, and while recursion can sometimes replace iteration, it is not always the optimal solution. Recursion can be a natural way to express a problem, but tail recursion is an optimization technique that can reduce stack usage. Dynamic programming is a useful approach for solving optimization problems by breaking down large problems into smaller ones and gradually constructing larger solutions. While recursion solves problems by breaking them down, dynamic programming works by building up from small to large problems.

Chapter 6

Searching

6.1 Objectives

- Be able to implement sequential search and binary search algorithms.
- Understand the concept of using hash tables as a search technique.
- Use Vec to implement a HashMap data structure.

6.2 What is Searching?

In this chapter, we will use the data structures we have previously implemented (stacks, queues, and linked lists) to solve practical search and sort problems. Search and sort are fundamental tasks in computer science, and many software applications and algorithms are developed around them. You are likely familiar with the search function in various software applications you have used, such as finding specific characters in a Word document or using the search bar in Goolge browser to find information. Search and lookup mean the same thing, and this book does not distinguish between the two.

Searching is the process of finding a specific item within a set of items. Search algorithms typically return either true or false to indicate whether the item exists in the set, and sometimes they also return the position of the item. In Rust, we can use the contains() function to query whether an item is in a set. This function checks whether the data contains a certain value and performs a search for us.

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

While contains() is a simple search algorithm, there are many different search methods available, including sequential search, binary search, and hash search. We will explore how these different algorithms work and what their complexities are.

6.3 The Sequential Search

When data items are stored in collections such as Vec, arrays, or slices, they have a linear relationship since each item is stored at a position relative to the other items. This means that the index values of the

data items in slices are ordered, and can be accessed in order, making these data structures linear as well. Similarly, the stacks, queues, and linked lists that we have previously studied are also linear. Based on this same linear logic inherent in the physical world, a natural search technique is linear search, which is also known as sequential search.

Linear search works by starting at the first item in the slice and moving from one item to another in order until the target item is found or the entire slice is traversed. If the item being searched for is not found after traversing the entire slice, it means that the item does not exist. The following figure illustrates how this search works.

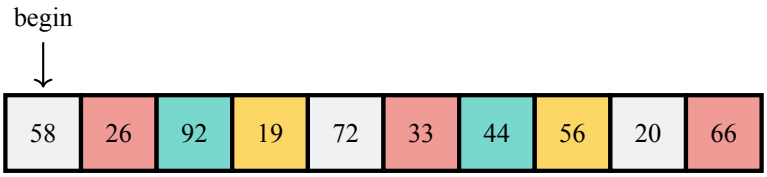


Figure 6.1: Sequential search

6.3.1 Implementing an Sequential Search in Rust

The following Rust code implements sequential search, and it is straightforward to understand. The program takes a slice of integers and a target integer as input. It iterates through each element in the slice and compares it with the target value. If the value is found, the function returns a boolean value: true. Otherwise, it returns false.

```
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      // Continue looping if pos is within the index range and
8      // the item has not been found.
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 }
```

Here is an example of sequential search execution.

```
1  fn main() {
2      let num = 8;
3      let nums = [9,3,7,4,1,6,2,8,5];
4      let found = sequential_search(&nums, num);
5      println!("nums contains {num}: {found}");
6      // nums contains 8: true
7  }
```

Of course, sequencial search can also return the specific position of the search item, or return None if it is not found.

```rust
// sequential_search_pos.rs

fn sequential_search_pos(nums:&[i32],num:i32) -> Option<usize>{
    let mut pos: usize = 0;
    let mut found = false;
    while pos < nums.len() && !found {
        if num == nums[pos] {
            found = true;
        } else {
            pos += 1;
        }
    }

    if found { Some(pos) } else { None }
}

fn main() {
    let num = 8;
    let nums = [9,3,7,4,1,6,2,8,5];
    match sequential_search_pos(&nums, num) {
        Some(pos) => println!("{num}'s index: {pos}"),
        None => println!("nums does not contain {num}"),
    }
    // 8's index: 7
}
```

6.3.2 Analysis of Sequential Search

To analyze the complexity of the sequential search algorithm, we need to set a basic computational unit. Since comparison operation is the primary operation in searching, counting the number of comparisons is the most important. Data items in a set are randomly placed and unordered, and the probability of a data item being at any position in the set is the same according to probability theory.

If the target item is not in the set, the only way to know this result is to compare the target with all the data items in the set. If there are n items, then the sequential search requires n comparisons, and the complexity is $O(n)$. What if the target item is in the set? Is the complexity still $O(n)$?

When searching for a target item in a set, the analysis of the algorithm's complexity is not always straightforward. There are three possibilities: the best case scenario is when the target item is at the beginning of the set, and only one comparison is needed, resulting in a complexity of $O(1)$. In the worst case scenario, the target item is at the end of the set, and it takes n comparisons to find it, resulting in a complexity of $O(n)$. Finally, the target item may be distributed throughout the set, and there are n-2 possible complexities, ranging from $O(2)$ to $O(n-1)$. In this scenario, the number of possible comparisons ranges from 1 to n times, and the average complexity equals the sum of all possible complexities divided by the total number of comparisons.

$$\sum_{i=1}^{n} O(i)/n = O(n/2) = O(n) \tag{6.1}$$

When the set becomes large, the complexity of sequential search on a random sequence is $O(n)$, as 1/2 can be ignored. However, if the data items are ordered in ascending order and the probability of the

target item existing in any of the n positions is still the same, the search performance can be improved. If the target item does not exist, the search can be accelerated through techniques like binary search. For example, if searching for the target item 50, the comparison is performed in order until 56. At this point, it can be determined that there is no target value of 50 behind it because the items behind 56 are larger than 56, and the algorithm stops searching.

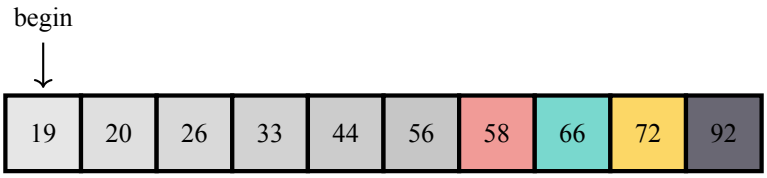


Figure 6.2: Sequential search on ordered data

The sequential search algorithm used on sorted data sets is shown below. It optimizes the algorithm by setting the ”stop” variable to control the search and stop immediately when it goes beyond the range for saving time.

```
// ordered_sequential_search.rs

fn ordered_sequential_search(nums:&[i32], num:i32) -> bool {
    let mut pos = 0;
    let mut found = false;
    // return when reads on ordered data
    let mut stop = false;

    while pos < nums.len() && !found && !stop {
        if num == nums[pos] {
            found = true;
        } else if num < nums[pos] {
            // ordered data, return
            stop = true;
        } 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!("nums contains {num}: {found}");
    // nums contains 44: true

    let num = 49;
    let found = ordered_sequential_search(&nums, num);
    println!("nums contains {num}: {found}");
    // nums contains 49: false
}
```

In the case of sorted data, if the target item is not in the set and is less than the first item, only one comparison is needed to determine that the item is not in the set. In the worst case, the algorithm would need to compare the target item with all n items in the set, resulting in n comparisons. The average number of comparisons is still n/2, and the complexity remains at $O(n)$. However, the $O(n)$ complexity is better than the search on unordered data because most searches follow the average case. In fact, the average case complexity for sorted data sets can be twice as fast as unordered ones. Thus, sorting has always been a crucial topic in computer science. The complexity of sequential search for unordered and ordered data sets is summarized in the table below.

Table 6.1: Complexity of Sequential Search

Case	Min	Avg	Max	Search Class
target exist	$O(1)$	$O(\frac{n}{2})$	$O(n)$	unorderd search
target not exist	$O(n)$	$O(n)$	$O(n)$	unorderd search
target exist	$O(1)$	$O(\frac{n}{2})$	$O(n)$	ordered search
target not exist	$O(1)$	$O(\frac{n}{2})$	$O(n)$	ordered search

6.4 The Binary Search

An ordered dataset is advantageous for search algorithms because sequential search on sorted data sets requires comparing only n-1 remaining items if the first item is not the one being searched for. However, although this method can be terminated when a value outside of the range is encountered, it is still relatively slow. Is there a faster search algorithm for sorted datasets? Absolutely, it's called binary search. In the following discussion, we will analyze and implement binary search in Rust.

6.4.1 Implementing a Binary Search

As its name implies, binary search is a very important search algorithm that works by dividing the dataset into two parts for searching using low, mid, and high to control the range of the search. It starts from the middle item, not in order. If the middle item is the one being searched for, the search is completed. If it is not, we can use the ordered property of the sorted set to eliminate half of the remaining items. If the item being searched for is greater than the middle item, we can eliminate the middle item and the half of the elements smaller than the middle item, which means we don't need to compare the first item to the middle item. This is because, if the target item is greater than the middle value, it is definitely not in the first half, whether it is in the set or not. Conversely, if the target item is less than the middle item, the second half of the data can be skipped. After removing half of the data, look at the middle item in the remaining half of the data, repeat the comparison and omission process, and finally obtain the result. This search is relatively fast and very intuitive, so it is called binary search.

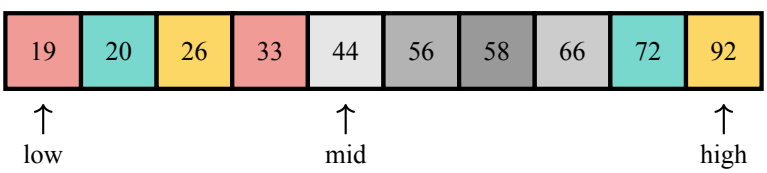


Figure 6.3: Binary search

Binary search is a search algorithm that is used on sorted data sets. It is faster than sequential search since it divides the data set into two parts using low, mid, and high to control the range of the search. If the middle item is the one being searched for, the search is completed. If not, the ordered property of the

sorted set is used to eliminate half of the remaining items. By repeating the comparison and omission process, the target item is found relatively quickly.

To implement binary search, we set low and high to the far left and far right, respectively. For instance, if we want to find 60, we compare it with the middle value of 44. If it is greater than 44, we move low to 56 and mid to 66. We then compare it with 66 and find that it is less than 66, so we move high to 58 and mid to 56. This process continues until we reach the target item.

```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;

    // note: <= not <
    while low <= high && !found {
        let mid: usize = (low + high) >> 1;

        //low + high may cause overflow, use substraction
        //let mid: usize = low + ((high - low) >> 1);

        if num == nums[mid] {
            found = true;
        } else if num < nums[mid] {
            // num < mid, drop right half of data
            high = mid - 1;
        } else {
            // num >= mid, drop left half of data
            low = mid + 1;
        }
    }

    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!("nums contains {target}: {found}");
    // nums contains 3: true

    let target = 63;
    let found = binary_search1(&nums, target);
    println!("nums contains {target}: {found}");
    // nums contains 63: false
}
```

Binary search satisfies the three laws of recursion and can be implemented using recursion. However, recursive implementation involves slicing the dataset and discarding the mid item, which can cause stack overflow risk. Therefore, it is generally recommended to use an iterative approach to implement binary search.

```rust
// binary_search.rs

fn binary_search2(nums: &[i32], num: i32) -> bool {
    // base case1: target does not exisit
    if 0 == nums.len() { return false; }

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

    // base case2: target exisits
    if num  == nums[mid] {
        return true;
    } else if num < nums[mid] {
        // minimize problem size
        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 target = 3;
    let found = binary_search2(&nums, target);
    println!("nums contains {target}: {found}");
    // nums contains 3: true

    let target = 63;
    let found = binary_search2(&nums, target);
    println!("nums contains {target}: {found}");
    // nums contains 63: false
}
```

In summary, binary search is a fast and intuitive search algorithm that works best on sorted data sets. It can be implemented using recursion but an iterative approach is generally preferred to avoid stack overflow risk.

6.4.2 Analysis of Binary Search

The binary search algorithm has a best-case scenario when the middle item is the target, and its time complexity is $O(1)$. With each comparison, binary search eliminates half of the remaining items, allowing us to determine the worst-case complexity by finding the maximum number of comparisons. After the first comparison, there are $n/2$ remaining items, then $n/4$, and so on, until the remaining items are reduced to $n/2^i = 1$, at which point the binary search ends.

$$\begin{aligned} \frac{n}{2^i} &= 1 \\ i &= log_2(n) \end{aligned} \tag{6.2}$$

As a result, the binary search algorithm performs at most $\log_2(n)$ comparisons, giving it a time complexity of $O(\log_2(n))$, which is superior to algorithms with a time complexity of $O(n)$. However, it's important to note that in the recursive implementation of binary search, the default stack usage consumes memory, resulting in a space complexity that's inferior to the iterative version.

While binary search may seem efficient, it's not worth sorting and using binary search when n is very small. In such cases, sequential search may be more efficient. Additionally, sorting large datasets for binary search can be time-consuming and memory-intensive, making sequential search a more efficient option. However, binary search is well-suited for datasets that are neither too large nor too small, making it an ideal choice for many practical scenarios.

6.4.3 The Interpolation Search

Interpolation search is a variant of binary search that's specifically designed for sorted data. When the data is evenly distributed, interpolation search can quickly approach the search area and improve efficiency.

Unlike binary search, interpolation search doesn't use the median directly to bound the search range. Instead, it finds the upper and lower bounds using an interpolation algorithm. This algorithm uses two points (x_0, y_0) and (x_1, y_1) to perform linear interpolation and determine the value of y corresponding to any point x in the range $[x_0, x_1]$ or the value of x corresponding to any y.

$$\begin{aligned} \frac{y - y_0}{x - x_0} &= \frac{y_1 - y_0}{x_1 - x_0} \\ x &= \frac{(y - y_0)(x_1 - x_0)}{y_1 - y_0} + x_0 \end{aligned} \tag{6.3}$$

To illustrate this, let's consider an example of searching for the element 27 in the sorted set of 14 elements [1, 9, 10, 15, 16, 17, 19, 23, 27, 28, 29, 30, 32, 35]. We use the index as the x-axis and the element value as the y-axis. It's found that $x_0 = 0, x_1 = 13$, and $y_0 = 1, y_1 = 35$. Therefore, the x value corresponding to $y = 27$ can be calculated.

$$\begin{aligned} x &= \frac{(27 - 1)(13 - 0)}{35 - 1} + 0 \\ x &= 9 \end{aligned} \tag{6.4}$$

Starting with nums[9] which has a value of 28, which is greater than 27, we use it as the upper bound. Since the index of 28 is 9, we continue the interpolation algorithm to search for elements in the range [0,8].

$$\begin{aligned} x &= \frac{(27 - 1)(8 - 0)}{27 - 1} + 0 \\ x &= 8 \end{aligned} \tag{6.5}$$

Finally, we arrive at nums[9], which has a value of 27, and the algorithm ends.

```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 {
```

```
16              break;
17          }
18
19          // calculate the position for insertion
20          let offset = (target - low_val)*(high - low) as i32
21                       / (high_val - low_val);
22          let interpolant = low + offset as usize;
23
24          // update high and low
25          if nums[interpolant] > target {
26              high = interpolant - 1;
27          } else if nums[interpolant] < target {
28              low = interpolant + 1;
29          } else {
30              break;
31          }
32      }
33
34      // determine if the value in high position equals target
35      target == nums[high]
36  }
```

Here is an example of interpolation search.

```
1  fn main() {
2      let nums = [1,9,10,15,16,17,19,23,27,28,29,30,32,35];
3      let target = 27;
4      let found = interpolation_search(&nums, target);
5      println!("nums contains {target}: {found}");
6      // nums contains 27: true
7
8      let nums = [0,1,2,10,16,19,31,35,36,38,40,42,43,55];
9      let found = interpolation_search(&nums, target);
10     println!("nums contains {target}: {found}");
11     // nums contains 27: false
12 }
```

Upon careful analysis of the code, we can see that the only difference between interpolation search and binary search is the calculation method of the interpolant. In all other aspects, the two algorithms are almost identical.

When the data is evenly distributed, the time complexity of interpolation search is $O(loglog(n))$. The proof of this result is complex and can be found in this paper [11]. However, the worst-case and average-case complexity of interpolation search are both $O(n)$.

6.4.4 The Exponential Search

Exponential search is a variant of binary search that utilizes the exponential function to estimate the midpoint, making it suitable for sorted data without bounds. The algorithm continuously compares the value at positions of 2^0, 2^1, 2^2, 2^k to the target value, then determines the search range, and finally performs binary search within that range. The time complexity of exponential search is $O(log(i))$, where i is the position of the target value.

For instance, to find the value 22 in the sorted set of 15 elements [2,3,4,6,7,8,10,13,15,19,20,22,23,24,28], exponential search first checks whether the value at position $2^0 = 1$ is greater than 22. It then continues

to check the values at positions 2^1, 2^2, and 2^3, which are 4, 7, and 15, all of which are less than 22. Checking position $2^4 = 16$ is beyond the range, so the upper bound is the last index, 14.

```
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      // find the upper bound
8      let mut high = 1usize;
9      while high < size && nums[high] < target {
10         high <<= 1;
11     }
12     // use the half of the upper bound as the lower bound
13     let low = high >> 1;
14
15     // use binary_search method implemented previously
16     binary_search(&nums[low..size.min(high+1)], target)
17 }
18
19 fn main() {
20     let nums = [1,9,10,15,16,17,19,23,27,28,29,30,32,35];
21     let target = 27;
22     let found = exponential_search(&nums, target);
23     println!("nums contains {target}: {found}");
24     // nums contains 27: true
25 }
```

Note that in the implementation of the exponential search algorithm, the lower bound at line 13 is half of the high but can also be set to 0. However, setting it to 0 may reduce efficiency.

The complexity of exponential search is divided into two parts: finding the upper bound for dividing the search range, and performing binary search. The complexity of finding the upper bound is related to the target value i, and its complexity is $O(log(i))$. The complexity of binary search is $O(log(n))$, where n is the length of the search range. The length of the search range is $high - low = 2^{log(i)} - 2^{log(i)-1} = 2^{log(i)-1}$, and its complexity is $O(log(2^{log(i)-1})) = O(log(i))$. Therefore, the total complexity is $O(log(i) + log(i)) = O(log(i))$.

6.5 The Hash Search

The search algorithms we have discussed so far use the positional information of items in a collection to perform searches. Binary search can find data items in logarithmic time by sorting the collection. However, if an algorithm has prior knowledge of the stored addresses of different items, it can retrieve them directly without comparing them one by one. This method is called Hash Search, a search algorithm with a complexity of $O(1)$, making it the fastest search algorithm available.

To implement Hash Search, the item's address must exist first. A data structure that provides an efficient way to store and retrieve items' addresses is needed. This data structure is known as a hash table, which stores data items in such a way that they are easy to locate, with each item's position referred to as a slot or address. These slots are named starting from 0, or any other value. Once the first slot is chosen, all subsequent slots are incremented accordingly. Initially, the hash table contains no items, and each slot is empty. A hash table can be implemented using Vec, with each element initialized to None. The figure below shows a hash table with size m = 11, meaning that there are m slots named 0 to 10.

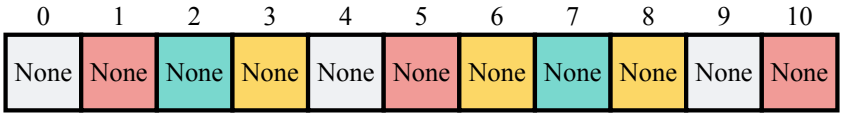


Figure 6.4: Hash table

The mapping between data items and their corresponding slots in the hash table is accomplished through a hash function. The hash function takes any item in the collection and returns a specific slot name, a process known as hashing. For example, if there are integer items [24, 61, 84, 41, 56, 31], and a hash table with a capacity of 11 slots, each item's location in the hash table can be obtained by inputting it into the hash function. A simple hash function involves using modulo, since any number modulo 11 will have a remainder within 11, ensuring that there is always a slot available to store the data.

$$hash(item) = item\%11 \tag{6.6}$$

Here is the result after modulo hash calculation.

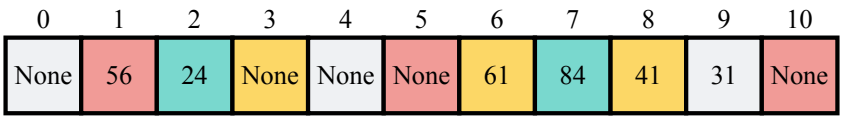


Figure 6.5: Modulo hash

Calculated the hash value, the item can be inserted into the corresponding slot in the hash table, as depicted in the figure above. Currently, 6 of the 11 slots in the table are occupied, resulting in a load factor of $\lambda = 6/11$. The load factor is a useful metric to evaluate the hash table, particularly when it needs to store a large number of items. A high load factor indicates that there is limited space for additional items, necessitating resizing the table. In languages like Rust and Go, resizing occurs automatically when the load factor exceeds a particular threshold, preparing for future data insertion.

Although the data stored in the hash table is unsorted, the hash function enables the calculation of the slot of a data item regardless of its disorderliness. For instance, to check if the number 56 exists in the table, its hash value can be computed (hash(56) = 1), and slot 1 can be examined to locate 56, indicating its presence in the table. The search operation has a complexity of $O(1)$, making hash search very efficient. However, conflicts may occur and must be resolved, or else the hash table cannot be utilized. For example, if 97 is added, and its hash value (hash(97) = 9) corresponds to an occupied slot (9) that contains 31 instead of 97, a collision occurs that necessitates resolution.

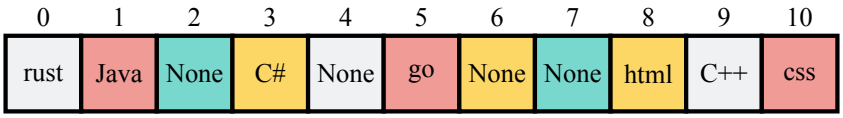
6.5.1 Hash Functions

In the previous section, the hash function computed the remainder of the item directly, ensuring that the remainder falls within a specific range. Any algorithm that can produce a value within a particular range based on the item can be used as a hash function. By improving the hash function, we can minimize the probability of collisions and create an effective hash table.

One such approach to improving the hash function is the "sum of groups" method, which divides items into equal-sized blocks (the last block may be different), adds them up, and then computes the remainder. For instance, if the data item is the phone number "316-545-0134," it can be split into two-digit numbers, padded with zeroes if required, resulting in [31, 65, 45, 01, 34]. The sum is 176, and taking the remainder of 11 yields a hash value (slot) of 0. The phone number may also be split into three-digit numbers or reversed before computing the remainder. Many hash functions are available, as any value that can be computed and then have its remainder taken can be used in various ways.

Another hash algorithm is the ”mid-square” method. The item is squared first, and then the middle portion of the square is extracted as the value for which the remainder is computed. For example, if the number is 36, its square is 1296, and taking the middle portion, 29, and then computing the remainder of 11 yields hash(29) = 7, indicating that 36 should be stored in slot 7.

When storing a string, the remainder can also be calculated based on the ASCII values of its characters. The string ”rust” has four characters with ASCII values of [114, 117, 115, 116], and their sum is 462. Computing the remainder results in hash(462) = 0. Other strings, such as ”Java” with ASCII values [74, 97, 118, 97], may also be tried. Its sum is 386, and taking the remainder gives hash(386) = 1, indicating that ”Java” should be stored in slot 1, as shown in the figure below.



The following is an implementation of hash function for ASCII value.

```
// 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 s1 = "rust"; let s2 = "Rust";
    let size = 11;

    let p1 = hash1(s1, size);
    let p2 = hash1(s2, size);
    println!("{s1} in slot {p1}, {s2} in slot {p2}");
    // rust in slot 0, Rust in slot 1
}
```

To mitigate collisions that may arise when using the above hash function, one can modify it slightly. For example, one can use the position of different characters in a string as weights and multiply the ASCII values by their positional weights.

$$\begin{aligned} hash(rust) &= (0 * 114 + 1 * 117 + 2 * 115 + 3 * 116)\%11 \\ &= 695\%11 \\ &= 2 \end{aligned} \tag{6.7}$$

Starting the index from 1 is preferred to ensure that the first character contributes to the total. Below is an example calculation and code snippet.

$$\begin{aligned} hash(rust) &= (1 * 114 + 2 * 117 + 3 * 115 + 4 * 116)\%11 \\ &= 1157\%11 \\ &= 2 \end{aligned} \tag{6.8}$$

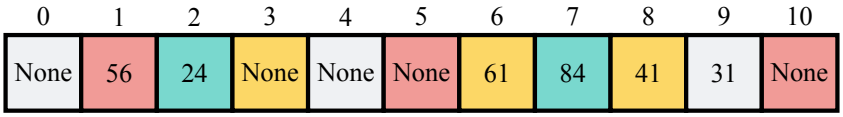
```rust
// hash.rs
fn hash2(astr: &str,  size: usize) -> usize {
    let mut sum = 0;
    for (i, c) in astr.chars().enumerate() {
        sum += (i + 1) * (c as usize);
    }
    sum % size
}

fn main() {
    let (s1, s2, size) = ("rust", "Rust", 11);
    let p1 = hash2(s1, size);
    let p2 = hash2(s2, size);
    // rust in slot 2, Rust in slot 3
}
```

Efficiency is crucial when designing a hash function to ensure that it does not become the bottleneck of the system. If the hash function is too complex, it could break the $O(1)$ complexity.

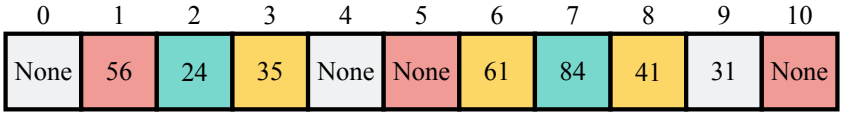
6.5.2 Collison Resolution

To address the issue of hash collisions, collision resolution methods are necessary. For instance, if a hash function with position as weight is used and the index starts from 0, the strings ”rust” and ”Rust” will collide. When two items are hashed to the same slot, a method must be used to insert the conflicting item into another slot of the hash table. This process is called collision resolution.

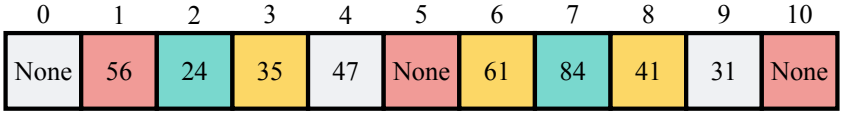
Although perfect hash tables do not exist due to limited memory and complex real-world situations, one intuitive method for resolving collisions is to search the hash table and attempt to find the next empty slot to save the conflicting item. The simplest method is called open addressing, specifically linear probing. Linear probing attempts to linearly probe the next empty slot in the hash table, starting from the original collision point until the first empty slot is found. Note that you can start searching again from the beginning after reaching the end. The following figure shows a hash table with 11 slots, where we try to insert 35 and 47.



When inserting 35, its position should be slot 2, but we find that slot 2 already contains 24. Therefore, we start searching for an empty slot from slot 2 and find that slot 3 is empty, so we insert 35 there.

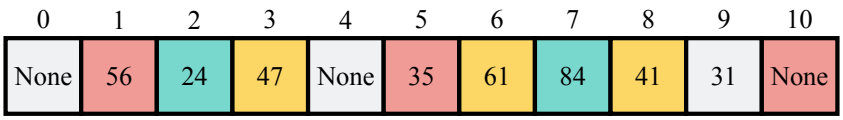


When we insert 47, we find that slot 3 has the value 35, so we search for the next empty slot and find that slot 4 is empty, so we insert 47 there.



To search for items in a hash table built using open addressing, we must use the same method as used during its construction. For instance, if we are searching for item 56 with a hash of 1, we locate it in slot 56 and return true. However, if we are looking for item 35, which has a hash of 2, we may find another item like 24 instead of 35 due to collisions. To avoid returning false, we need to conduct a sequential search until we locate 35, an empty slot, or we loop back to 24 before returning the result.

However, using a sequential search may lead to data clustering, where items cluster in the table due to multiple collisions occurring in the same hash slot. Linear probing fills subsequent slots with conflicting items, forcing the originally intended values to be inserted elsewhere. Sequential searches are time-consuming, with a complexity greater than $O(1)$. To address data clustering, open addressing technology can be extended to skip several slots instead of sequentially searching for the next open slot. By checking every three slots when a conflict occurs, for example, the conflicting items are more evenly distributed, thus dispersing the conflict. This method has a significant effect and can alleviate data clustering, as shown in the figure below.



When adding item 35, a conflict may occur, prompting the search to skip every three slots from that point onwards, thus dispersing the conflict. Consequently, item 47 can be inserted without any conflict. This method has proven effective in alleviating data clustering, as demonstrated in the following figure.

Rehashing is the process of finding an alternative slot in a hash table after a collision occurs. This process involves calculating a new hash value using a specified skip size, which must ensure that all slots in the table can eventually be accessed. To guarantee this, it is advisable to use a prime number as the table size, as exemplified by the use of 11 in the accompanying example.

$$rehash(pos) = (pos + n)\%size \tag{6.9}$$

Another approach to resolving conflicts is the chain method. This method involves setting up a linked list to store data items for each conflicting position, as shown in Figure (6.6). When searching, conflicts are resolved by sequentially searching the chain, which has a complexity of $O(n)$. If the data in the conflict chain is sorted, a binary search can be utilized to achieve a complexity of $O(log2(n))$. If the chain becomes too long, it can be transformed into a red-black tree to enhance its structural stability. In many programming languages, the chain method is the default implementation for resolving conflicts in hash table data structures.

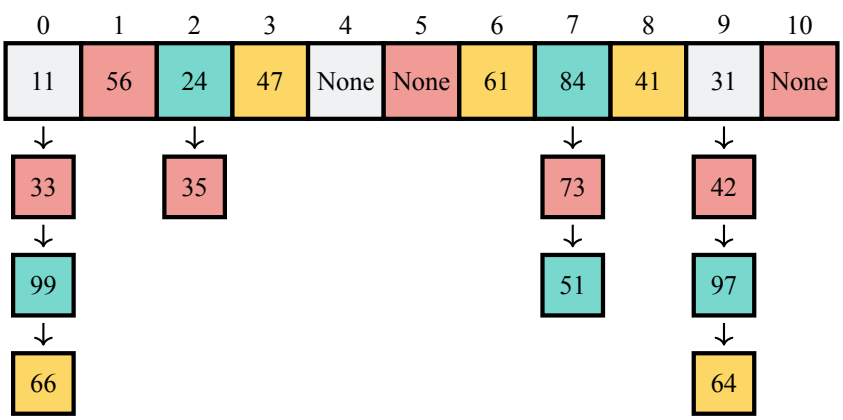


Figure 6.6: Collison Resolution by Chain Method

6.5.3 Implementing a HashMap in Rust

The most commonly used collection type in Rust is the HashMap. It is an associative data type that stores key-value pairs. The key is utilized to locate the position of the value, and due to the position uncertainty, this type of search is akin to searching on a map. Thus, this data structure is called HashMap.

HashMap is an unordered collection of key-value pairs where the keys are unique, and there is a one-to-one relationship between the keys and the values. Here is the abstract data type definition of HashMap:

• new(): creates a new HashMap, takes no arguments and returns an empty HashMap collection.

• insert(k,v): adds a new key-value pair to the HashMap, taking parameters k and v. If the key exists, the old value is replaced with the new value. No return value is provided.

• remove(k): removes value corresponding to k from the HashMap, takes parameter k, and returns the corresponding value v.

• get(k): returns the value v stored in the HashMap for the given key k (which may be empty None), takes parameter k.

• contains(k): returns true if the key k exists in the HashMap, otherwise false, takes parameter k.

• len(): returns the number of key-value pairs stored in the HashMap, takes no arguments.

The table below displays the results of operations assuming that 'h' is an already created empty HashMap (represented as {} here).

Table 6.2: HashMap operations

Operation	Value	Return
h.is_empty()	{}	true
h.insert("a", 1)	{a:1}	
h.insert("b", 2)	{a:1, b:2}	
h.insert("c", 3)	{a:1, b:2, c:3}	
h.get("b")	{a:1, b:2, c:3}	Some(2)
h.get("d")	{a:1, b:2, c:3}	None
h.len()	{a:1, b:2, c:3}	3
h.contains("c")	{a:1, b:2, c:3}	true
h.contains("e")	{a:1, b:2, c:3}	false
h.remove("a")	{b:2, c:3}	Some(1)
h.remove("a")	{b:2, c:3}	None
h.insert("a", 1)	{b:2, c:3, a:1}	
h.contains("a")	{b:2, c:3, a:1}	true
h.len()	{b:2, b:3, a:1}	3
h.get("a")	{b:2, c:3, a:1}	Some(1)
h.remove("c")	{b:2, a:1}	Some(3)

In actual implementation, a HashMap in Rust is made up of two separate Vecs - one for storing the keys (called slot) and the other for storing the values (called data). The slot Vec saves the keys with the index starting from 1, while the default value in the slot Vec is 0. The HashMap is encapsulated by a struct, and a cap parameter is added to control the capacity.

```
1 // hashmap.rs
2
3 #[derive(Debug, Clone, PartialEq)]
4 struct HashMap <T> {
5     cap: usize, // capacity
6     slot: Vec<usize>, // store data address (index)
7     data: Vec<T>, // store elements
8 }
```

The rehash function in Rust can be implemented using a linear search method that adds 1, which is simple and easy to implement. The initial size of the HashMap is set to 11, but it can also be set to other prime numbers, such as 13, 17, 19, 23, 29, etc. Below is the complete implementation code for a HashMap in Rust.

```
// hashmap.rs

impl<T: Clone + PartialEq + Default> HashMap<T> {
    fn new(cap: usize) -> Self {
        // Initialize slot and data
        let mut slot = Vec::with_capacity(cap);
        let mut data = Vec::with_capacity(cap);
        for _i in 0..cap{
            slot.push(0);
            data.push(Default::default());
        }

        HashMap { cap, slot, data }
    }

    fn len(&self) -> usize {
        let mut len = 0;
        for &d in self.slot.iter() {
            // If slot is not empty, then increase len by 1
            if 0 != d  {
                len += 1;
            }
        }
        len
    }

    fn is_empty(&self) -> bool {
        let mut empty = true;
        for &d in self.slot.iter() {
            if 0 != d  {
                empty = false;
                break;
            }
        }
        empty
    }

    fn clear(&mut self) {
        let mut slot = Vec::with_capacity(self.cap);
        let mut data = Vec::with_capacity(self.cap);
        for _i in 0..self.cap{
            slot.push(0);
            data.push(Default::default());
        }

        self.slot = slot;
        self.data = data;
```

```
48      }
49
50      fn hash(&self, key: usize) -> usize {
51          key % self.cap
52      }
53
54      fn rehash(&self, pos: usize) -> usize {
55          (pos + 1) % self.cap
56      }
57
58      fn insert(&mut self, key: usize, value: T) {
59          if 0 == key { panic!("Error: key must > 0"); }
60
61          let pos = self.hash(key);
62          if 0 == self.slot[pos] {
63              // If the slot is empty, insert directly
64              self.slot[pos] = key;
65              self.data[pos] = value;
66          } else {
67              // If the slot is not empty, then
68              // find next available position
69              let mut next = self.rehash(pos);
70              while 0 != self.slot[next]
71                  && key != self.slot[next] {
72                  next = self.rehash(next);
73
74                  // If the slot is full, exit
75                  if next == pos {
76                      println!("Error: slot is full!");
77                      return;
78                  }
79              }
80
81              // Insert the data in the found slot
82              if 0 == self.slot[next] {
83                  self.slot[next] = key;
84                  self.data[next] = value;
85              } else {
86                  self.data[next] = value;
87              }
88          }
89      }
90
91      fn remove(&mut self, key: usize) -> Option<T> {
92          if 0 == key { panic!("Error: key must > 0"); }
93
94          let pos = self.hash(key);
95          if 0 == self.slot[pos] {
96              // If the slot is empty, return None
97              None
98          } else if key == self.slot[pos] {
99              // If found the same key,
```

```
100             // update the slot and data
101             self.slot[pos] = 0;
102             let data = Some(self.data[pos].clone());
103             self.data[pos] = Default::default();
104             data
105         } else {
106             let mut data: Option<T>  = None;
107             let mut stop = false;
108             let mut found = false;
109             let mut curr = pos;
110 
111             while 0 != self.slot[curr] && !found && !stop {
112                 if key == self.slot[curr] {
113                     // If the value is found, delete the data
114                     found = true;
115                     self.slot[curr] = 0;
116                     data = Some(self.data[curr].clone());
117                     self.data[curr] = Default::default();
118                 } else {
119                     // If rehashing returns to the initial
120                     // position, it means that it has searched
121                     // a full circle and still not found
122                     curr = self.rehash(curr);
123                     if curr == pos {
124                         stop = true;
125                     }
126                 }
127             }
128 
129             data
130         }
131     }
132 
133     fn get_pos(&self, key: usize) -> usize {
134         if 0 == key { panic!("Error: key must > 0"); }
135 
136         // Calculate the data position
137         let pos = self.hash(key);
138         let mut stop = false;
139         let mut found = false;
140         let mut curr = pos;
141 
142         // Loop through to find the data
143         while 0 != self.slot[curr] && !found && !stop {
144             if key == self.slot[curr] {
145                 found = true;
146             } else {
147                 // If rehashing returns to the initial position
148                 // it means that it has searched a full circle
149                 // and still not found
150                 curr = self.rehash(curr);
151                 if curr == pos {
```

```
152                         stop = true;
153                     }
154                 }
155             }
156
157             curr
158         }
159
160         // Get a reference and a mutable reference to val
161         fn get(&self, key: usize) -> Option<&T> {
162             let curr = self.get_pos(key);
163             self.data.get(curr)
164         }
165
166         fn get_mut(&mut self, key: usize) -> Option<&mut T> {
167             let curr = self.get_pos(key);
168             self.data.get_mut(curr)
169         }
170
171         fn contains(&self, key: usize) -> bool {
172             if 0 == key {
173                 panic!("Error: key must > 0");
174             }
175             self.slot.contains(&key)
176         }
177
178         // Implement iteration and mutable iteration for hashmap
179         fn iter(&self) -> Iter<T> {
180             let mut iterator = Iter { stack: Vec::new() };
181             for item in self.data.iter() {
182                 iterator.stack.push(item);
183             }
184             iterator
185         }
186
187         fn iter_mut(&mut self) -> IterMut<T> {
188             let mut iterator = IterMut { stack: Vec::new() };
189             for item in self.data.iter_mut() {
190                 iterator.stack.push(item);
191             }
192             iterator
193         }
194     }
195
196     // Implementation of iteration
197     struct Iter<'a, T: 'a> { stack: Vec<&'a T>, }
198     impl<'a, T> Iterator for Iter<'a, T> {
199         type Item = &'a T;
200         fn next(&mut self) -> Option<Self::Item> {
201             self.stack.pop()
202         }
203     }
```

```
204
205 struct IterMut<'a, T: 'a> { stack: Vec<&'a mut T>, }
206 impl<'a, T> Iterator for IterMut<'a, T> {
207     type Item = &'a mut T;
208     fn next(&mut self) -> Option<Self::Item> {
209         self.stack.pop()
210     }
211 }
212
213 fn main() {
214     basic();
215     iter();
216
217     fn basic() {
218         let mut hmap = HashMap::new(11);
219         hmap.insert(2,"dog");
220         hmap.insert(3,"tiger");
221         hmap.insert(10,"cat");
222
223         println!("empty: {}, size: {:?}",
224                     hmap.is_empty(), hmap.len());
225         println!("contains key 2: {}", hmap.contains(2));
226
227         println!("key 3: {:?}", hmap.get(3));
228         let val_ptr = hmap.get_mut(3).unwrap();
229         *val_ptr = "fish";
230         println!("key 3: {:?}", hmap.get(3));
231         println!("remove key 3: {:?}", hmap.remove(3));
232         println!("remove key 3: {:?}", hmap.remove(3));
233
234         hmap.clear();
235         println!("empty: {}, size: {:?}",
236                     hmap.is_empty(), hmap.len());
237     }
238
239     fn iter() {
240         let mut hmap = HashMap::new(11);
241         hmap.insert(2,"dog");
242         hmap.insert(3,"tiger");
243         hmap.insert(10,"cat");
244
245         for item in hmap.iter() {
246             println!("val: {item}");
247         }
248
249         for item in hmap.iter_mut() {
250             *item = "fish";
251         }
252
253         for item in hmap.iter() {
254             println!("val: {item}");
255         }
```

```
256         }
257     }
```

Here is the code execution output.

```
empty: false, size: 3
contains key 2: true
key 3: Some("tiger")
key 3: Some("fish")
remove key 3: Some("fish")
remove key 3: None
empty: true, size: 0
val: cat
val:
val:
val:
val:
val:
val:
val: tiger
val: dog
val:
val:
val: fish
val: fish
val: fish
val: fish
val: fish
val: fish
val: fish
val: fish
val: fish
val: fish
val: fish
```

6.5.4 Analysis of HashMap

A hash table provides a lookup complexity of $O(1)$ in the best case. However, due to collisions, the number of comparisons during the lookup process can vary. The severity of collisions affects the performance of the hash table, and a good evaluation metric is the load factor λ. A small load factor means the chance of collisions is low, and items are more likely to be in their corresponding slot. A large load factor means the table is filling up quickly, and there are more collisions, leading to a more complex conflict resolution process.

The result of each search is either success or failure. For searches performed using linear probing open addressing, the average number of comparisons for success is about $\frac{1+\frac{1}{1-\lambda}}{2}$, and for failure, it is about $\frac{1+(\frac{1}{1-\lambda})^2}{2}$. Even when using chaining, the average number of comparisons for success is $1 + \lambda/2$, and for failure, it is λ. Overall, the lookup complexity of a hash table is around $O(\lambda)$.

When using the chaining method in a hash table, an increase in collisions results in more items being added to the same chain. As a result, searching for a specific item in the chain takes longer, making it the main contributor to the search time.

6.6 Summary

This chapter covers several search algorithms, namely sequential search, binary search, interpolation search, exponential search and hash search. Sequential search is a straightforward algorithm that has a complexity of $O(n)$. Binary search, on the other hand, is a fast algorithm that cuts the data set in half each time, but it requires the data to be sorted, with a complexity of $O(\log_2(n))$. Other search algorithms, such as interpolation search and exponential search, build on binary search and are suitable for different types of data distribution. Hash search is a highly efficient $O(1)$ search algorithm that uses a HashMap. However, it is crucial to consider that hash tables are prone to collisions and need appropriate measures, such as open addressing and chaining, to resolve them. Sorting is a helpful technique that speeds up search algorithms, and in the next chapter, we will delve into sorting algorithms.

Chapter 7

Sorting

7.1 Objectives

- Learn sorting algorithms.
- Be able to implement the ten basic sorting algorithms in Rust.

7.2 What is Sorting?

Sorting involves arranging elements in a specific order in a collection. For example, playing cards are shuffled and arranged by suit or order to play effectively, and words can be sorted alphabetically or by length. Similarly, Chinese cities can be sorted by population, ethnicity, or region. Orderliness is a crucial skill for humans as it enables smooth progress in various areas.

Sorting is an important field in computer science, and many pioneers have contributed to the development of sorting algorithms. Various algorithms, including sequential search and binary search, benefit from sorting datasets. Sorting algorithm efficiency is dependent on the number of items being processed, and complex sorting methods are too expensive for small collections while simple algorithms are not suitable for large datasets. In this chapter, we will explore different sorting algorithms and compare their performance.

Comparison is the core operation in sorting, as it determines sequence, order, and orderliness. Comparison involves looking at which element is larger or smaller based on a criterion, which can be simple (numerical size) or abstract (health index). If the comparison shows that the order is incorrect, data positions need to be exchanged. However, exchanging data positions is an expensive operation in computer science, and the total number of exchanges is crucial for evaluating the efficiency of algorithms. In this chapter, we will analyze various sorting algorithms using the Big O analysis method, which is the most intuitive indicator for evaluating their efficiency.

Apart from efficiency, sorting also has the issue of stability. For instance, in a list like [1,4,9,8,5,5,2,3,7,6], there are two 5s, and different sorts may swap the order of the two 5s. Although they are adjacent in the end, the original order relationship has been destroyed. For numeric data, this may not matter, but for data structures containing keys like personal information, rash sorting may change the order, which is not desirable.

```
person {            person {
    amount: 5,          amount: 5,
    name: Shieber,      name: Kew,
    phone: 133xx,       phone: 133xx,
    age: 20,            age: 24,
}                   }
```

To ensure that the order of the elements in a sorted collection remains stable, it is important to consider stability when evaluating sorting algorithms. Stability refers to whether the relative order of equal elements in the original collection is maintained in the sorted collection. For example, if the amount: 5 appears twice in a collection and sorting changes the order of the two occurrences, this can cause issues for algorithms that rely on the stability of the sequence, such as deduction operations.

There are numerous sorting algorithms available for collections, but there are ten basic types of sorting algorithms that serve as the foundation for most of them. These include bubble sort, quick sort, selection sort, heap sort, insertion sort, shell sort, merge sort, counting sort, bucket sort, and radix sort. Many improved algorithms have been derived from these ten basic algorithms. In this article, we will explain several of these improved algorithms, including new bubble sort, cocktail sort, comb sort, binary insertion sort, flash sort, and Tim sort.

7.3 The Bubble Sort

Bubble sort is a simple sorting algorithm that requires multiple iterations through the collection, comparing adjacent items and swapping those that are unordered. Each iteration places the largest value in its correct position. The algorithm gets its name from the similarity of the sorting process to bubbles rising to the top of a pot of boiling water.

In the first pass, the algorithm compares adjacent items and swaps them if they are in the wrong order. If there are n items in the list, there will be n-1 comparisons in the first pass. In the second pass, the largest value in the data set is already in the correct position, and the remaining n-1 items need to be sorted, which means there will be n-2 comparisons. The algorithm requires n-1 iterations to place the next largest value in its proper position. After completing n-1 rounds of iteration and comparison, the smallest item will definitely be in the correct position, and no further processing is needed.

92	84	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

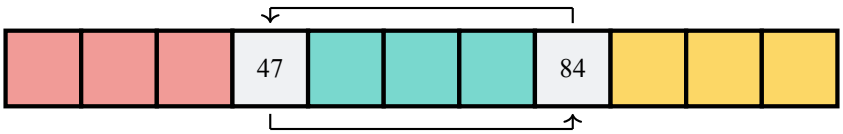
Figure 7.1: Bubble sort

The diagonal line in the figure above represents the maximum value, and it keeps moving towards the right, just like bubbles rising to the top. Bubble sort involves frequent swap operations, which are com-

monly used auxiliary operations in comparisons. In Rust, the Vec data structure defaults to implementing the swap() function, but you can also implement the following swap operation.

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

Some programming languages provide a convenient way to swap values without using temporary variables, such as: data[i], data[j] = data[j], data[i]. Although this feature still uses variables internally, it operates on two variables simultaneously, as shown in the figure below.



In this chapter, we only deal with sets of numbers to simplify the algorithm design. Therefore, we can use a Vec to implement bubble sort.

```
// bubble_sort.rs

fn bubble_sort1(nums: &mut [i32]) {
    if nums.len() < 2 {
        return;
    }

    for i in 1..nums.len() {
        for j in 0..nums.len()-i {
            if nums[j] > nums[j+1] {
                nums.swap(j, j+1);
            }
        }
    }
}
```

Here is an example of bubble sort.

```
fn main() {
    let mut nums = [54,26,93,17,77,31,44,55,20];
    bubble_sort1(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [17, 20, 26, 31, 44, 54, 55, 77, 93]
}
```

It is worth noting that to avoid a double for-loop, we can also use a while-loop to implement bubble sort.

```
// bubble_sort.rs

fn bubble_sort2(nums: &mut [i32]) {
    let mut len = nums.len() - 1;

    while len > 0 {
        for i in 0..len {
```

```
8               if nums[i] > nums[i+1] {
9                   nums.swap(i, i+1);
10              }
11          }
12
13          len -= 1;
14      }
15  }
16
17  fn main() {
18      let mut nums = [54,26,93,17,77,31,44,55,20];
19      bubble_sort2(&mut nums);
20      println!("sorted nums: {:?}", nums);
21      // sorted nums: [17, 20, 26, 31, 44, 54, 55, 77, 93]
22  }
```

Bubble sort involves comparing adjacent items and swapping those that are unordered, which repeats until the largest item is in its correct position. The algorithm performs n - 1 rounds of traversal to sort n numbers, regardless of the initial arrangement. During the first round, n - 1 comparisons are required, followed by n - 2 comparisons in the second round, and so on, until only one comparison is needed. The total number of comparisons can be calculated using the formula:

$$1 + 2 + ..., +n - 1 = \frac{n^2}{2} + \frac{n}{2} \tag{7.1}$$

This means that the time complexity of bubble sort is $O(\frac{n^2}{2} + \frac{n}{2}) = O(n^2)$.

Although both bubble sort algorithms presented above achieve sorting, even a sorted set requires continuous comparisons and swapping of data items. To optimize the algorithm, we can add a variable to control whether comparisons should continue and exit directly when encountering a sorted set.

```
1   // bubble_sort.rs
2
3   fn bubble_sort3(nums: &mut [i32]) {
4       // compare controls whether to continue comparing
5       let mut compare = true;
6       let mut len = nums.len() - 1;
7
8       while len > 0 && compare {
9           compare = false;
10          for i in 0..len {
11              if nums[i] > nums[i+1] {
12                  // Data is unordered and need to compare
13                  nums.swap(i, i+1);
14                  compare = true;
15              }
16          }
17
18          len -= 1;
19      }
20  }
21
22  fn main() {
23      let mut nums = [54,26,93,17,77,31,44,55,20];
```

```
    bubble_sort3(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [17, 20, 26, 31, 44, 54, 55, 77, 93]
}
```

Bubble sort is a sorting algorithm that compares adjacent elements in an array, starting from the first number and exchanging positions based on their relative size. Elements are only swapped from left to right. However, can we perform bubble sort from right to left? Yes, we can, and this bidirectional sorting method is called cocktail sort. Cocktail sort is a variant of bubble sort that sorts in descending order when sorting from right to left. While it slightly optimizes bubble sort, its time complexity is still $O(n^2)$, but it approaches $O(n)$ if the sequence is already sorted.

```
// cocktail_sort.rs

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

    // bubble controls the sort process
    let mut bubble = true;
    let len = nums.len();
    for i in 0..(len >> 1) {
        if bubble {
            bubble = false;
            // bubble from left to right
            for j in i..(len - i - 1) {
                if nums[j] > nums[j+1] {
                    nums.swap(j, j+1);
                    bubble = true
                }
            }
            // bubble from right to left
            for j in (i+1..=(len - i - 1)).rev() {
                if nums[j] < nums[j-1] {
                    nums.swap(j-1, j);
                    bubble = true
                }
            }
        } else {
            break;
        }
    }
}

fn main() {
    let mut nums = [1,3,2,8,3,6,4,9,5,10,6,7];
    cocktail_sort(&mut nums);
    println!("sorted nums {:?}", nums);
    // sorted nums [1, 2, 3, 3, 4, 5, 6, 6, 7, 8, 9, 10]
}
```

In contrast to bubble sort, comb sort can compare items with a distance greater than 1. Comb sort starts by setting the gap to the length of the array and decreasing it by a fixed ratio in each iteration of the loop, typically by multiplying it by 0.8, which is the most effective ratio determined by the original

author through experimentation. When the gap is 1, comb sort degenerates into bubble sort. Comb sort aims to move inverted numbers forward as much as possible and ensure that the numbers in the current gap are sorted, similar to combing hair with a comb, where the gap is similar to the gap between the teeth of the comb. Comb sort has a time complexity of $O(nlogn)$, with a space complexity of $O(1)$, and it is an unstable sorting algorithm.

```rust
// comb_sort.rs

fn comb_sort(nums: &mut [i32]) {
    if nums.len() <= 1 { return; }
    let mut i;
    let mut gap: usize = nums.len();

    // ordered basicly
    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;
        }
    }

    // rearrange the element properly.
    // exchange controls the process
    let mut exchange = true;
    while exchange {
        exchange = false;
        i = 0;
        while i < nums.len() - 1 {
            if nums[i] > nums[i+1] {
                nums.swap(i, i+1);
                exchange = true;
            }
            i += 1;
        }
    }
}
```

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

Bubble sort suffers from the problem that the boundary index, such as i, j, i+1, j+1, must be arranged properly and cannot be wrong. In 2021, a new sorting algorithm [12] was published that does not require handling boundary index values. It is intuitive and resembles bubble sort at first glance, but it is actually similar to insertion sort. Although it looks like a descending sort, it is actually an ascending sort.

```
// CantBelieveItCanSort.rs

fn cbic_sort1(nums: &mut [i32]) {
    for i in 0..nums.len() {
        for j in 0..nums.len() {
            if nums[i] < nums[j] { nums.swap(i, j); }
        }
    }
}

fn main() {
    let mut nums = [54,32,99,18,75,31,43,56,21,22];
    cbic_sort1(&mut nums);
    println!("sorted nums {:?}", nums);
    // sorted nums [18, 21, 22, 31, 32, 43, 54, 56, 75, 99]
}
```

Of course, it can also be implemented as a descending sort by changing the less than symbol to a greater than symbol.

```
// CantBelieveItCanSort.rs

fn cbic_sort2(nums: &mut [i32]) {
    for i in 0..nums.len() {
        for j in 0..nums.len() {
            if nums[i] > nums[j] {
                nums.swap(i, j);
            }
        }
    }
}
```

```
fn main() {
    let mut nums = [54,32,99,18,75,31,43,56,21,22];
    cbic_sort2(&mut nums);
    println!("sorted nums {:?}", nums);
    // sorted nums [99, 75, 56, 54, 43, 32, 31, 22, 21, 18]
}
```

This algorithm uses only two for loops, and the index values do not need to be handled separately. While it resembles the definition of bubble sort, it is not actually a bubble sort algorithm.

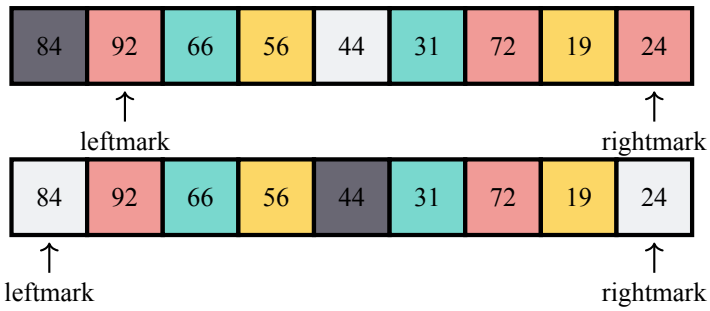
7.4 The Quick Sort

Quicksort and bubble sort do have some similarities, but it is not accurate to say that quicksort is an upgraded version of bubble sort. Quicksort is a completely different algorithm that uses the divide-and-conquer strategy to speed up the sorting process.

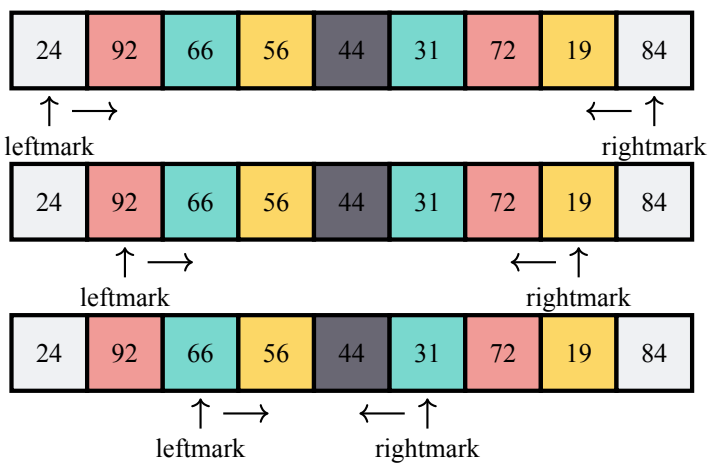
The quicksort algorithm involves two main steps: selecting a pivot value and partitioning the set. The pivot value can be any value in the collection, but it is typically chosen to be a value that is close to the middle of the final sorted collection for the fastest sorting speed. There are many different methods for selecting a pivot value, but this article only explains the principle and does not consider algorithm optimization.

In the following figure, the pivot value of 84 is not necessarily at the middle of the sorted collection, and it would be more efficient to choose a value closer to the middle, such as 56, as the pivot value. However, choose a correct pivot value is not the focus of this article.

To implement quicksort, two markers need to be set for comparison after selecting the pivot value (the dark gray value). The left and right markers should be located at the far left and far right extremes of the collection, except for the pivot value.



The goal of partitioning is to move the items that are misaligned with respect to the pivot value. By comparing the values at the left and right markers with the pivot value and swapping the smaller value to the left marker and the larger value to the right marker, a basicly sorted collection can be achieved quickly through repeated swaps.

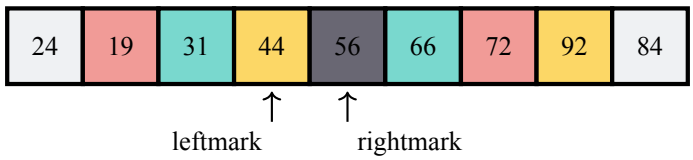


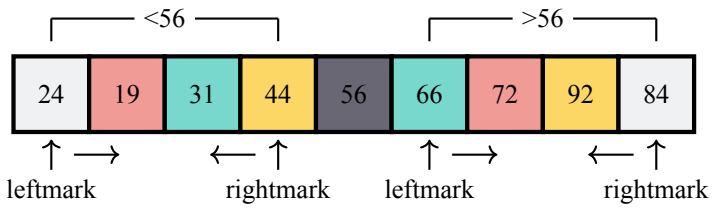
To begin partitioning, right shift the left index until a value greater than or equal to the pivot value is found. Then left shift the decreasing right index until a value less than or equal to the pivot value is found. If the value of the left index is greater than the right index, swap the values. In this case, 84 and 24 satisfy this condition, so the values are directly swapped. Repeat this process until the left and right indices cross each other.

After crossing, compare the values of the left and right indices. If the right is less than the left, swap the right index value with the pivot value. Otherwise, swap the left index value with the pivot value. The right index value serves as the splitting point, dividing the set into two intervals.

Recursively call quicksort on the left and right intervals until the sorting is completed. It is important to note that the pivot value does not necessarily have to be the value in the middle of the collection but

should be a value that is in the middle or close to the middle of the final sorted collection for the fastest sorting speed.





Once quicksort is executed on both the left and right sides, the sorting process is completed. If the set's length is less than or equal to one, it's already sorted, and the program can exit directly. To implement quicksort, we have a dedicated partition function that selects the first item(or selects a random value in the data set) as the pivot value.

```
1  // quick_sort.rs
2
3  fn quick_sort1(nums: &mut [i32], low: usize, high: usize) {
4      if low < high {
5          let split = partition(nums, low, high);
6
7          // avoid out of range (split <= 1) and syntax error
8          if split > 1 {
9              quick_sort1(nums, low, split - 1);
10         }
11
12         quick_sort1(nums, split + 1, high);
13     }
14 }
15
16 fn partition(nums:&mut[i32], low:usize,high:usize) -> usize {
17     // left marker and right marker
18     let mut lm = low;
19     let mut rm = high;
20
21     loop {
22         // left mark move to right gradually
23         while lm <= rm && nums[lm] <= nums[low] {
24             lm += 1;
25         }
26
27         // right mark move to left gradually
28         while lm <= rm && nums[rm] >= nums[low] {
29             rm -= 1;
30         }
```

```
31
32          // once lm > rm, return and exchange data between
33          // postion lm and rm
34          if lm > rm {
35              break;
36          } else {
37              nums.swap(lm, rm);
38          }
39      }
40
41      nums.swap(low, rm);
42
43      rm
44  }
45
46  fn main() {
47      let mut nums = [54,26,93,17,77,31,44,55,20];
48      let high = nums.len() - 1;
49      quick_sort1(&mut nums, 0, high);
50      println!("sorted nums: {:?}", nums);
51      // sorted nums: [17, 20, 26, 31, 44, 54, 55, 77, 93]
52  }
```

However, it's also possible to implement quicksort directly through the recursive method without using the partition function.

```
1   // quick_sort.rs
2
3   fn quick_sort2(nums: &mut [i32], low: usize, high: usize) {
4       if low >= high {
5           return;
6       }
7
8       // left marker and right marker
9       let mut lm = low;
10      let mut rm = high;
11      while lm < rm {
12          // right marker move to left gradually
13          while lm < rm && nums[low] <= nums[rm] {
14              rm -= 1;
15          }
16
17          // left marker move to right gradually
18          while lm < rm && nums[lm] <= nums[low] {
19              lm += 1;
20          }
21
22          // exchange data between position lm and rm
23          nums.swap(lm, rm);
24      }
25
26      // exchange data between position low and lm
27      nums.swap(low, lm);
```

```
28
29      if lm > 1 {
30          quick_sort2(nums, low, lm - 1);
31      }
32
33      quick_sort2(nums, rm + 1, high);
34  }
35
36  fn main() {
37      let mut nums = [54,26,93,17,77,31,44,55,20];
38      let high = nums.len() - 1;
39      quick_sort2(&mut nums, 0, high);
40      println!("sorted nums: {:?}", nums);
41      // sorted nums: [17, 20, 26, 31, 44, 54, 55, 77, 93]
42
43      let mut nums = [1000,-1,2,-20,89,64,0,99,73];
44      let high = nums.len() - 1;
45      quick_sort2(&mut nums, 0, high);
46      println!("sorted nums: {:?}", nums);
47      // sorted nums: [-20, -1, 0, 2, 64, 73, 89, 99, 1000]
48  }
```

To partition a set of length n in quicksort, there will be $log(n)$ partitions if the partition is always in the middle. However, finding the splitting point requires checking the pivot value against each of the n items, resulting in a complexity of $nlog(n)$. In the worst case, when the splitting point is far from the middle, sorting of items 1 and n - 1 will be repeated n times, leading to a complexity of $O(n^2)$.

Quicksort relies on recursion, but excessive depth can decrease its performance. To overcome this limitation, introsort switches to heap sort after the recursion depth exceeds $log(n)$. For small numbers (n < 20), introsort switches to insertion sort. This mixed sorting algorithm can achieve the high performance of quicksort on regular datasets and maintain a performance of $O(nlog(n))$ in the worst case. Introsort is a built-in sorting algorithm in C++.

Quicksort divides the array to be sorted into two areas for sorting. However, if there are a large number of duplicate elements, quicksort will repeatedly compare them, resulting in wasted performance. To address this issue, the array can be divided into three areas for sorting(three-area quicksort). Duplicate elements are placed in the third area, and only the other two areas are sorted. The repeated data is chosen as the pivot value, and the less-than pivot value is put into the left area, the greater-than pivot value into the right area, and the equal pivot value into the middle area. Then, three-area quicksort is recursively called on the left and right areas.

7.5 The Insertion Sort

Insertion sort achieves sorting by iteratively inserting data items into a sorted subsequence. Although its performance is still $O(n^2)$, its approach is distinct from other sorting algorithms. It always maintains a sorted subsequence at the lower end of the dataset and inserts new items into the subsequence, expanding it until the entire collection is sorted.

The algorithm starts by assuming that the initial subsequence has only one item located at position 0. During the next traversal, items 1 to n-1 are compared with the first item. If an item is smaller than the first item, it is inserted before the first item. If it is larger than the first item, the subsequence is expanded by one. This process is repeated for the remaining unsorted items until the entire collection is sorted. In the following figure, the sorted region is shown in red, and the unsorted region in blue. Each row represents a sorting operation, and the bottom row shows the sorted result.

84	92	66	56	44	31	72	19	24	84 ordered
84	92	66	56	44	31	72	19	24	still ordered
66	84	92	56	44	31	72	19	24	insert 66
56	66	84	92	44	31	72	19	24	insert 56
44	56	66	84	92	31	72	19	24	insert 44
31	44	56	66	84	92	72	19	24	insert 31
31	44	56	66	72	84	92	19	24	insert 72
19	31	44	56	66	72	84	92	24	insert 19
19	24	31	44	56	66	72	84	92	insert 24

Figure 7.2: Insertion sort

```rust
// insertion_sort.rs
fn insertion_sort(nums: &mut [i32]) {
    if nums.len() < 2 { return; }
    for i in 1..nums.len() {
        let mut pos = i;
        let curr = nums[i];
        while pos > 0 && curr < nums[pos-1] {
            // move element to right
            nums[pos] = nums[pos-1];
            pos -= 1;
        }

        // insert element: curr
        nums[pos] = curr;
    }
}
fn main() {
    let mut nums = [54,32,99,18,75,31,43,56,21];
    insertion_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [18, 21, 31, 32, 43, 54, 56, 75, 99]
}
```

The insertion sort requires comparing each new element with the already sorted elements one by one. However, the binary search algorithm discussed in Chapter 6 can efficiently locate the position of an element in a sorted subsequence. Therefore, binary search can be utilized for acceleration.

```rust
// binary_insertion_sort.rs

fn binary_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;

        // Sorted array boundaries
        right = i - 1;

        // Data to be sorted
        temp = nums[i];

        // Binary search finds the position of temp
        while left <= right {
            mid = (left + right) >> 1;
            if temp < nums[mid] {
                // To prevent right = 0 - 1 case
                if 0 == mid {
                    break;
                }
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        // Move data back
        for j in (left..=i-1).rev() {
            nums.swap(j, j+1);
        }

        // Insert temp into the empty space
        if left != i {
            nums[left] = temp;
        }
    }
}

fn main() {
    let mut nums = [1,3,2,8,6,4,9,7,5,10];
    binary_insertion_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
}
```

7.6 The Shell Sort

Shell sort, also known as diminishing increment sort, is a sorting algorithm that divides the original set into smaller subsets and applies insertion sort to each subset. The way of selecting subsets is the key to Shell sort. Rather than evenly splitting the set into contiguous sublists, Shell sort selects items every few items to add to the subset, with the distance between them called the gap.

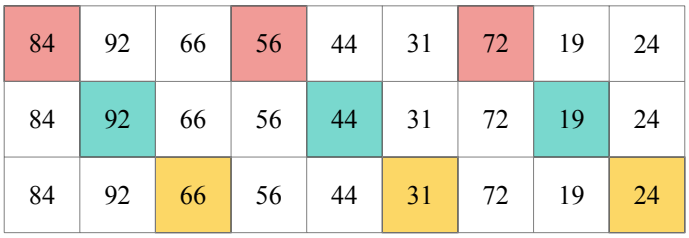


Figure 7.3: Shell sort

To understand Shell sort, consider the figure above. The set in the figure has nine items. If a gap of three is used, there will be a total of three subsets, each with three items of the same color. These separated elements can be considered as connected together, so insertion sort can be used to sort elements of the same color.

After sorting the subsets, the overall set is still unordered, as shown in the figure below. Although the overall set is unordered, it is not completely unordered. Subsets of the same color are ordered. By sorting the entire set using insertion sort, the set can be completely sorted quickly. The number of insertion sort moves is small at this point because adjacent items are in their own subsets' ordered positions, and these adjacent items are almost ordered. Therefore, only a few insertion moves are needed to complete the sorting.

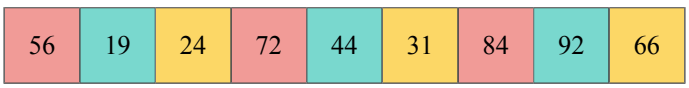


Figure 7.4: Shell sort

In Shell sort, the increment is the key feature, and different increments can be used to determine the number of subsets. The gap value is adjusted continuously in the implementation of Shell sort to achieve sorting.

```
1  // shell_sort.rs
2
3  fn shell_sort(nums: &mut [i32]) {
4      // Internal function for insertion sort
5      // with elements exhcanged distance is gap
6      fn ist_sort(nums: &mut [i32], start: usize, gap: usize) {
7          let mut i = start + gap;
8
9          while i < nums.len() {
10             let mut pos = i;
11             let curr = nums[pos];
12
13             while pos >= gap && curr < nums[pos - gap] {
14                 nums[pos] = nums[pos - gap];
15                 pos -= gap;
```

```
            }

            nums[pos] = curr;
            i += gap;
        }
    }

    // minimize the gap in every loop untill 1
    let mut gap  = nums.len() / 2;
    while gap > 0 {
        for start in 0..gap {
            ist_sort(nums, start, gap);
        }

        gap /= 2;
    }
}
```

The following are use cases.

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

    let mut nums = [1000,-1,2,-20,89,64,0,99,73];
    shell_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [-20, -1, 0, 2, 64, 73, 89, 99, 1000]
}
```

Shell sort divides the original set into smaller subsets and applies insertion sort to each subset. The subsets are selected by choosing an item every few items with the distance between them called the gap. The set is then considered as connected together, and insertion sort is used to sort elements of the same color. The final insertion operation in Shell sort is much less than in insertion sort because the collection has been pre-sorted by earlier increments. This makes the overall sorting very efficient.

Although the complexity analysis of Shell sort is slightly more complicated, it is roughly between $O(n)$ and $O(n^2)$. By changing the gap value according to the formula $2^k - 1$ (1, 3, 7, 15, 31), the complexity of Shell sort is approximately $O(n^{\frac{3}{2}})$, which is very fast. Additionally, binary search can be used to further improve Shell sort, similar to insertion sort. However, the index processing in binary search needs to add the gap value. Readers are encouraged to think about implementing this improved algorithm.

7.7 The Merge Sort

The previouly learned divide and conquer strategy can be used to improve the performance of sorting algorithms, specifically merge sort. Both merge sort and quicksort use recursion to continuously divide the list in half for sorting. If the set has only one item, it is considered sorted. If there are multiple items, the set is split, and recursive calls to merge sort are made for both intervals. Once sorted, a merge operation combines the two intervals into a single sorted set. This process is illustrated in the figures shown below.

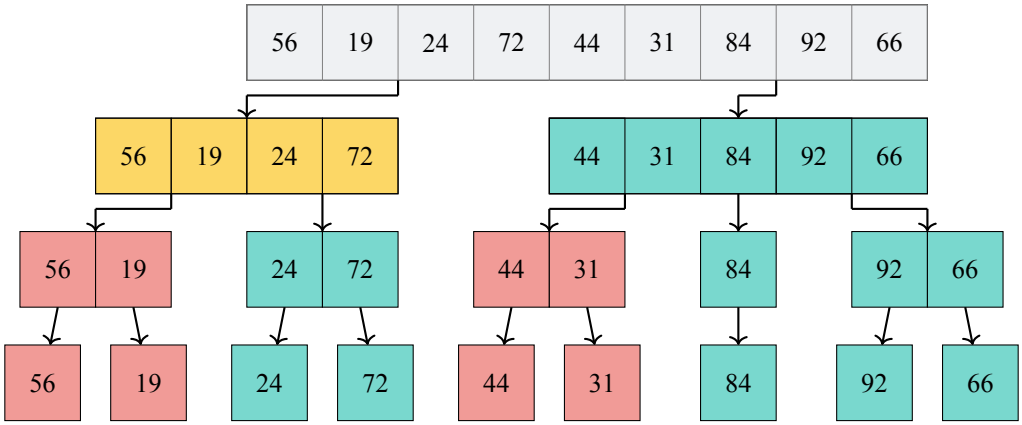


Figure 7.5: Split procedures of merge sort

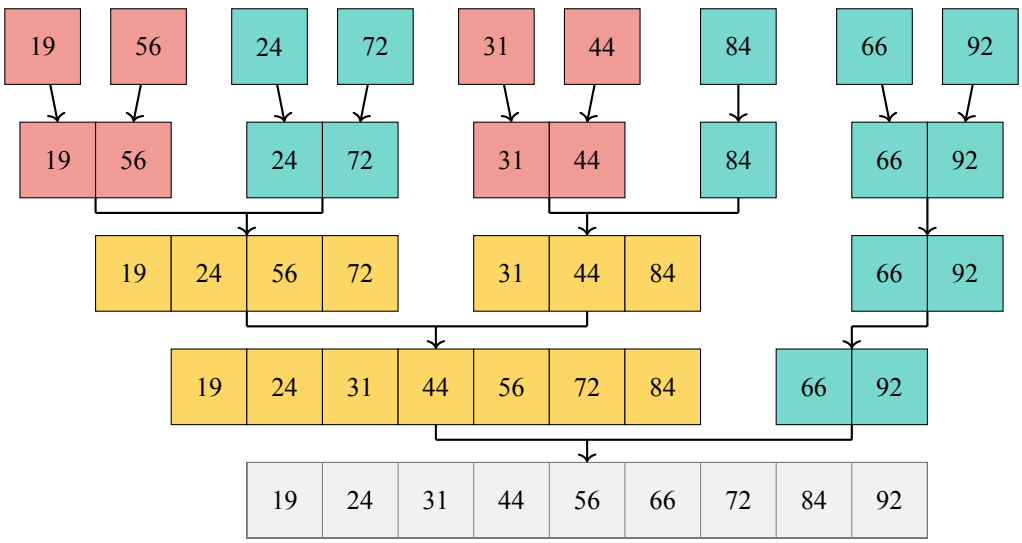


Figure 7.6: Merge procedures of merge sort

Merge sort breaks down the set into basic cases with one or two elements, allowing for easy direct comparison. Then, the merge process begins by first sorting the smallest subsequence of the basic case. Two-by-two merging is performed until the set is completely sorted. Even if the set is not evenly divided, the performance is not affected, as the difference is at most one element. The merge operation is straightforward, as each subsequence is already sorted, and only one comparison is necessary at a time. The resulting sequence is always ordered. Below is a Rust implementation of merge sort, consisting of two sorts and one merge operation, which is straightforward.

```
1  // merge_sort.rs
2
3  fn merge_sort(nums: &mut [i32]) {
4      if nums.len() > 1 {
5          let mid = nums.len() >> 1;
6          merge_sort(&mut nums[..mid]); // sort the first half
7          merge_sort(&mut nums[mid..]); // sort the last half
8          merge(nums, mid); // merge all
```

```
    }
}

fn merge(nums: &mut [i32], mid: usize) {
    let mut i = 0;   // mark element in first half of data
    let mut k = mid; // mark element in last half of data
    let mut temp = Vec::new();

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

        // put into a temp colletion
        if nums[i] < nums[k] {
            temp.push(nums[i]);
            i += 1;
        } else {
            temp.push(nums[k]);
            k += 1;
        }
    }

    // to make sure all data been solved
    if i < mid && k == nums.len() {
        for j in i..mid {
            temp.push(nums[j]);
        }
    } else if i == mid && k < nums.len() {
        for j in k..nums.len() {
            temp.push(nums[j]);
        }
    }

    // put temp data back to nums, finish sort
    for j in 0..nums.len() {
        nums[j] = temp[j];
    }
}

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

To analyze the time complexity of merge sort, we can divide the sorting process into two parts: sorting and merging. The sorting complexity is $O(log_2(n))$, as we learned in the binary search section. The merging step involves placing each item in the set on the sorted list, with a maximum of n times, resulting in a complexity of $O(n)$. As recursive and merging operations are combined, the performance of merge sort is $O(nlog_2(n))$.

The space complexity of merge sort is relatively high at $O(n)$. To reduce space usage, we can consider optimizing merge sort using insertion sort. One approach is to use insertion sort directly when the length is less than a certain threshold(such as 64) and to use merge sort when the length is greater than the threshold. This algorithm is called insertion merge sort and improves the merge sort algorithm to some extent. However, implementation details are left to the reader.

7.8 The Selection Sort

Selection sort is a more efficient version of bubble sort, as it performs only one data exchange per round by seeking only the index of the maximum value during traversal. After completing the traversal, it swaps that maximum item into its correct position. Like bubble sort, after the first traversal, the maximum item is in the last position, and after the second traversal, the second-largest value is in the second-to-last position. To sort n items, n-1 passes are required.

Although selection sort and bubble sort have the same number of comparisons, selection sort is faster because of its reduced number of data exchanges. The time complexity of both algorithms is $O(n^2)$. The figure below shows the entire sorting process of selection sort, in which the maximum unsorted item is selected during each traversal and placed in the correct position. Unlike bubble sort, selection sort does not exchange pairs during the traversal.

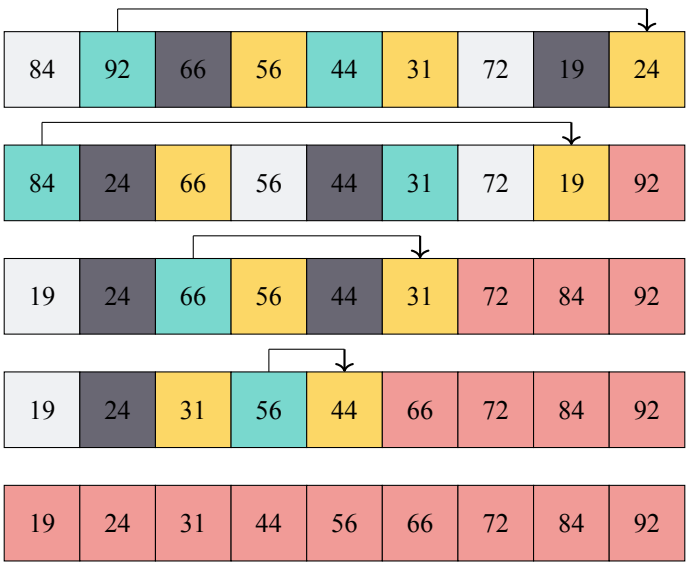


Figure 7.7: Selection sort

```rust
// selection_sort.rs

fn selection_sort(nums: &mut Vec<i32>) {
    let mut left = nums.len() - 1; // left marker
    while left > 0 {
        let mut pos_max = 0;
        for i in 1..=left {
            if nums[i] > nums[pos_max] {
                pos_max = i; // current loop's max value
            }
        }
        nums.swap(left, pos_max);
```

```
            left -= 1; // unsorted elements reduced by 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);
        // sorted nums: [18, 21, 22, 31, 32, 43, 54, 56, 75, 99]
    }
```

Although bubble sort and selection sort have the same time complexity, there are still opportunities for optimization. One such optimization is to perform bidirectional sorting, which can be achieved by adopting the cocktail sort approach. Similarly, selection sort can also be modified to perform bidirectional sorting, which changes the coefficient of complexity but not the overall time complexity. Readers are encouraged to consider the implementation of this approach.

7.9 The Heap Sort

In addition to linear data structures like stacks and queues, non-linear data structures also exist in computers, one of which is the heap. A heap is a non-linear complete binary tree with left and right child nodes. The height of a tree with n nodes is $log(n)$. Though trees are not studied until Chapter 8, comprehending them here can help understand the properties of heaps. There are two types of heaps: min-heaps and max-heaps. A min-heap has every node with a value less than or equal to its left and right child nodes, whereas a max-heap has every node with a value greater than or equal to its left and right child nodes. Figure (7.8) illustrates an example of a min-heap.

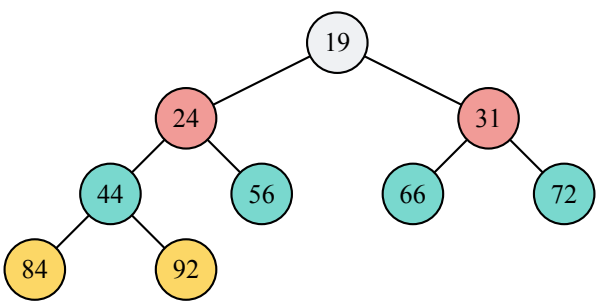


Figure 7.8: Min heap

Heap sort is a sorting algorithm designed using the heap data structure. It repeatedly selects the top element and moves it to the end, then rebuilds the heap to implement the sorting. It is a selection sort with worst-case, best-case, and average time complexity of $O(nlog_2(n))$ and is an unstable sort. As shown in the figure above, the heap is similar to a linked list with multiple connections. The nodes in the heap are numbered by layer, and if this logical structure is mapped to an array, it looks like the figure below, where the first position, index 0, is occupied by 0.

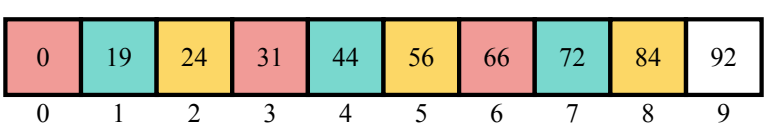


Figure 7.9: Array as a min heap

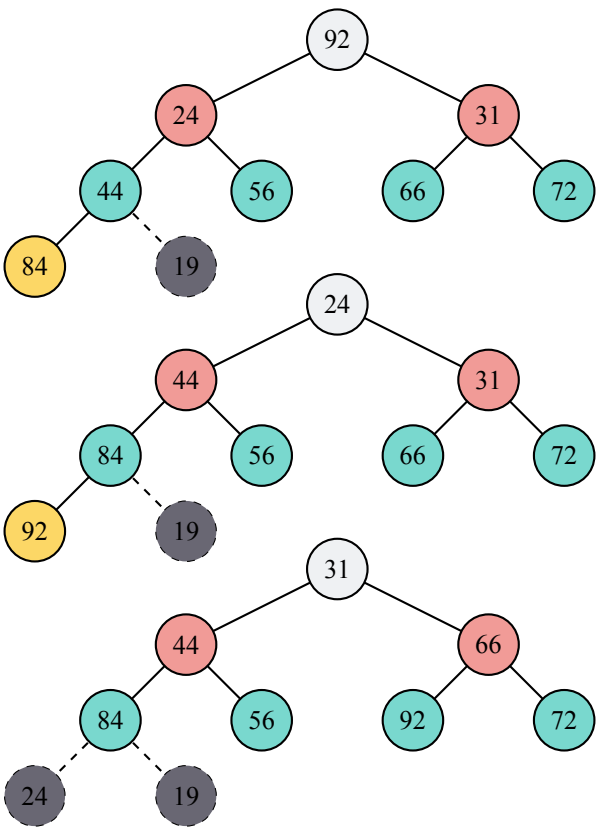
It is possible to represent heaps not only with trees, but also with arrays or Vecs(show in figure above). In fact, using arrays or Vecs to represent heaps is more in line with the literal meaning of the word ”heap,” which refers to a collection of things gathered together. Note that our index starts from 1, which allows us to represent the indices of the left and right child nodes as 2i and 2i+1, making it easier to calculate.

To satisfy the requirements of a binary tree’s node relationships, a heap represented by an array should meet the following criteria:

- Max heap: arr[i] >= arr[2i] and arr[i] >= arr[2i+1]
- Min heap: arr[i] <= arr[2i] and arr[i] <= arr[2i+1]

Heap sort involves constructing an unsorted sequence into a min heap. The minimum value of the entire sequence is then the root node of the heap. Swap it with the last element of the sequence, and the last element becomes the minimum value. This minimum value is no longer considered in the heap, and the remaining n-1 elements are reconstructed into a heap, producing a new minimum value. Swap this minimum value to the end of the new heap, and you will have two sorted values. Repeat this process until the entire sequence is sorted.A min heap produces a descending order sorting, while a max heap produces an ascending order sorting.

To better illustrate the heap sort process, a figure is provided below. The dark gray color represents the minimum element, which has been replaced by 92 and is no longer part of the heap. When 92 is at the top of the heap, it is no longer a min heap, so a new min heap is constructed to make the minimum value 24 at the top of the heap. Then, 24 is swapped with the last element in the heap, and the second-to-last element becomes dark gray. Note that the dotted line indicates that this element no longer belongs to the heap. Continuing to swap, the dark gray subsequence gradually fills the entire heap in reverse order from the last level, achieving a reverse sorting from largest to smallest. To achieve sorting from smallest to largest, a max heap should be constructed, and the corresponding logic in the min heap should be modified.



Here is an implementation of heap sort.

```
// heap_sort.rs

// calculate index of parent node
macro_rules! parent {
    ($child:ident) => {
        $child >> 1
    };
}

// calculate index of left child node
macro_rules! left_child {
    ($parent:ident) => {
        $parent << 1
    };
}

// calculate index of right child node
macro_rules! right_child {
    ($parent:ident) => {
        ($parent << 1) + 1
    };
}

fn heap_sort(nums: &mut [i32]) {
    if nums.len() < 2 { return; }

    let len = nums.len() - 1;
    let last_parent = parent!(len);
    for i in (1..=last_parent).rev() { // build a min-heap
        move_down(nums, i);            // begins from 1
    }

    for end in (1..nums.len()).rev() { // rebuild min-heap
        nums.swap(1, end);
        move_down(&mut nums[..end], 1);
    }
}

// move down greater element
fn move_down(nums: &mut [i32], mut parent: usize) {
    let last = nums.len() - 1;
    loop {
        let left = left_child!(parent);
        let right = right_child!(parent);
        if left > last { break; }

        // right <= last: determin if right child node exists
        let child = if right <= last
                    && nums[left] < nums[right] {
            right
```

```
51          } else {
52              left
53          };
54
55          // swap data if child node is greater than parent node
56          if nums[child] > nums[parent] {
57              nums.swap(parent, child);
58          }
59
60          // update parent and child relationship
61          parent = child;
62      }
63  }
64
65  fn main() {
66      let mut nums = [0,54,32,99,18,75,31,43,56,21,22];
67      heap_sort(&mut nums);
68      println!("sorted nums: {:?}", nums);
69      // sorted nums: [0, 18, 21, 22, 31, 32, 43, 54, 56, 75, 99]
70  }
```

The heap is a binary tree with a time complexity of $O(nlog(n))$. The time is mainly consumed in two parts: building the heap and adjusting the heap n times. Building the heap processes n elements with a complexity of $O(n)$. The longest path for each adjustment is from the root to the leaf, which is the height of the heap, logn. Therefore, the time complexity of n adjustments is $O(nlog(n))$, and the final total time complexity is $O(nlog(n))$. Macros are used here to obtain node indexes, and functions can also be used to implement them. Heap sort and selection sort are similar in that they both find the maximum or minimum value in a set.

7.10 The Bucket Sort

The previously discussed different sorting techniques, including comparison-based algorithms. However, there are also sorting algorithms that do not involve comparisons, such as bucket sort, counting sort, and radix sort. Non-comparison sorting works by determining the number of elements that come before each element in the sorted set. For example, given a set of numbers ”nums”, calculating the number of elements before ”nums[i]” uniquely determines its position in the sorted set. Non-comparison sorting can be completed in a single pass with a time complexity of $O(n)$, as it only requires counting the number of elements before each element.

Although non-comparison sorting has a low time complexity, it requires additional space to determine the position, making it suitable for specific data sizes and distributions(especially numbers). Non-comparison sorting requires indexable information within the data to determine the position. In contrast, comparison sorting is suitable for data of different sizes and distributions, making it applicable in various sorting situations.

Bucket sorting is one type of non-comparison sorting, where buckets are similar to slots in a hash table, but can hold multiple elements. The basic idea of bucket sorting are:

• Step 1: Divide the elements into different buckets, setting the number of buckets to k after finding maxV and minV, and dividing the interval [minV, maxV] into k equal intervals. Elements in the sequence are hashed to each bucket using a hash function (such as taking the remainder).

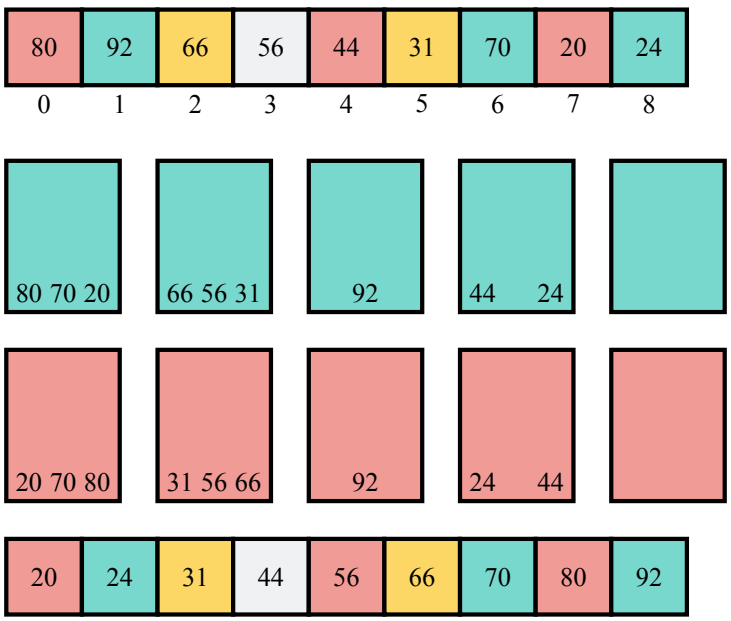
• Step 2: Sort the elements in each bucket with any available sorting algorithm.

• Step 3: Merge the ordered elements in each bucket into a large ordered set.

In Rust, a bucket can be defined as a structure that contains a hash function named hasher and a dataset named values.

```
// bucket_sort.rs

struct Bucket<H, T> {
    hasher: H,      // hasher: a function, recieved when called
    values: Vec<T>, // values: a container for data
}
```

To help readers understand bucket sort, the figure below illustrates the process of hashing data into buckets, sorting within each bucket, and merging the sorted data to obtain the final ordered set.



Here is an implementation of bucket sort.

```
// bucket_sort.rs

use std::fmt::Debug;

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

// Bucket sort, Debug feature is used for printing 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);

        // Binary search and sort bucket data
        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())),
        }
    }

    // Split the buckets and merge them into a single 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 = [0,54,32,99,18,75,31,43,4,56,21,22,1,100];
    bucket_sort(&mut nums, |t| t / 5);
    println!("{:?}", nums);
    // [0, 1, 4, 18, 21, 22, 31, 32, 43, 54, 56, 75, 99, 100]
}
```

Implementing bucket sort is more complex than comparison-based sorting algorithms because it involves creating a bucket structure and implementing a sorting algorithm that operates on this structure. In this implementation, the data is distributed into buckets using division by 5. If another value is used, the number of buckets needs to be adjusted accordingly.

Bucket sort has a time complexity of $O(n + nlogn - nlogk)$, where n is the number of elements to be sorted and k is the number of buckets used. Assuming the data is uniformly distributed, the average number of elements in each bucket is n/k. If quicksort is used to sort each bucket, the time complexity of each sorting operation is $O(n/klog(n/k))$. Therefore, the total time complexity can be expressed as $O(n) + kO(n/klog(n/k)) = O(n + nlog(n/k))$. When k is close to n, bucket sorting can be regarded as having a time complexity of $O(n)$.

One drawback of bucket sort is that it may create too many unnecessary buckets, resulting in wasted space. To address this issue, FlashSort is an optimized bucket sorting algorithm that estimates the number of buckets needed based on the number of elements to be sorted. If the number of buckets is greater than the number of elements, the number of buckets can be calculated by $m = f * n$, where f is a decimal such as 0.2. The rule for placing elements into buckets is defined by equation shown below, where $K(A_i)$ denotes the bucket index of element A_i.

$$K(A_i) = 1 + int((m-1)\frac{A_i - A_{min}}{A_{max} - A_{min}}) \tag{7.2}$$

Assuming each bucket has an average of n/m elements, using insertion sort for each bucket results in a total time complexity of $O(n^2/m)$. Experimental results show that FlashSort performs optimally at $O(n)$ when $m = 0.42n$. Moreover, when $m = 0.1n$, FlashSort is faster than quicksort as long as $n > 80$, and when $n = 10000$, it is twice as fast. Additionally, when $m = 0.2n$, it is 15% faster than when $m = 0.1n$. Even when m is as low as $0.05n$, FlashSort is still significantly faster than quicksort when $n > 200$.

7.11 The Counting Sort

Another type of non-comparison sorting is counting sort, which is a special case of bucket sort that only handles data of the same type, and therefore requires more space. The basic idea of counting sort is as follows:

• First, initialize a counter set of length maxV - minV + 1, where maxV is the maximum value in the set to be sorted, and minV is the minimum value. Set all values in the counter set to 0.

• Second, scan the set to be sorted and use the current value minus minV as the index in the counter set, and increment the counter at this index.

• Third, scan the counter set and write the values back to the original set in order to complete the sorting.

For example, given nums=[0,7,1,7,3,1,5,8,4,4,5], first traverse nums to obtain the minimum and maximum values, maxV=8, minV=0, and then initialize a counter set of length 8 - 0 + 1.

$$counter = [0, 0, 0, 0, 0, 0, 0, 0, 0]$$

Then scan nums and calculate the index by subtracting minV from the current value. For example, when scanning 0, the index is 0 - 0 = 0, so the value at index 0 of counter is incremented by 1. At this point, counter is [1,0,0,0,0,0,0,0,0]. Continue scanning until the final value of counter is obtained.

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

When traversing the counter set, if a value at a certain index is not 0, write the corresponding index value into nums, and decrement the value in counter. For example, if the value at the first position 0 of counter is 1, indicating that there is one 0 in nums, write it into nums. Continuing, the value at index 1 is 2, indicating that there are two 1s in nums, write them into nums. Finally, nums is sorted as:

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

The sorting process is completed by scanning the counter set, and the sorting process does not involve comparison, exchange, or other operations, making it very fast. Here is an implementation of counting sort:

```
// counting_sort.rs

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

    // bucket number is the maximum value in nums plus 1
    let max_bkt_num = 1 + nums.iter().max().unwrap();

    // save the number of each value in nums
    let mut counter = vec![0; max_bkt_num];
    for &v in nums.iter() {
        counter[v] += 1;
    }

    // write data back to original nums slice
    let mut j = 0;
    for i in 0..max_bkt_num {
        while counter[i] > 0 {
            nums[j] = i;
```

```
22                 counter[i] -= 1;
23                 j += 1;
24             }
25         }
26     }
27
28     fn main() {
29         let mut nums = [54,32,99,18,75,31,43,56,21,22];
30         counting_sort(&mut nums);
31         println!("sorted nums: {:?}", nums);
32         // sorted nums: [18, 21, 22, 31, 32, 43, 54, 56, 75, 99]
33     }
```

7.12 The Radix Sort

The third type of non-comparison sorting is radix sort, which uses the radix rule of positive numbers to sort. The sorting process involves collecting and distributing data, and the specific steps are as follows:

• Step 1: Find the maximum value in the given set of integers, and determine the number of digits in it. Then, pad zeros to the left of the numbers to unify the number of digits.

• Step 2: Starting from the lowest digit, perform stable sorting, collect the numbers according to their digits, and then sort the numbers according to the next higher digit until the sorting is complete.

For instance, consider an integer sequence [1, 134, 532, 45, 36, 346, 999, 102]. The maximum value in the set is 999, which has three digits. Therefore, three rounds of sorting are required for the ones, tens, and hundreds digits. Firstly, pad zeros to the left of the numbers to make all numbers have three digits, as shown in the second column of the figure. Then, sort the numbers according to their ones digit, tens digit, and hundreds digit in order. After three rounds of sorting, the data is sorted. The red digit in the figure represents the digit being sorted in the current round, and it is apparent that the numbers are sorted based on their corresponding digit at each iteration.

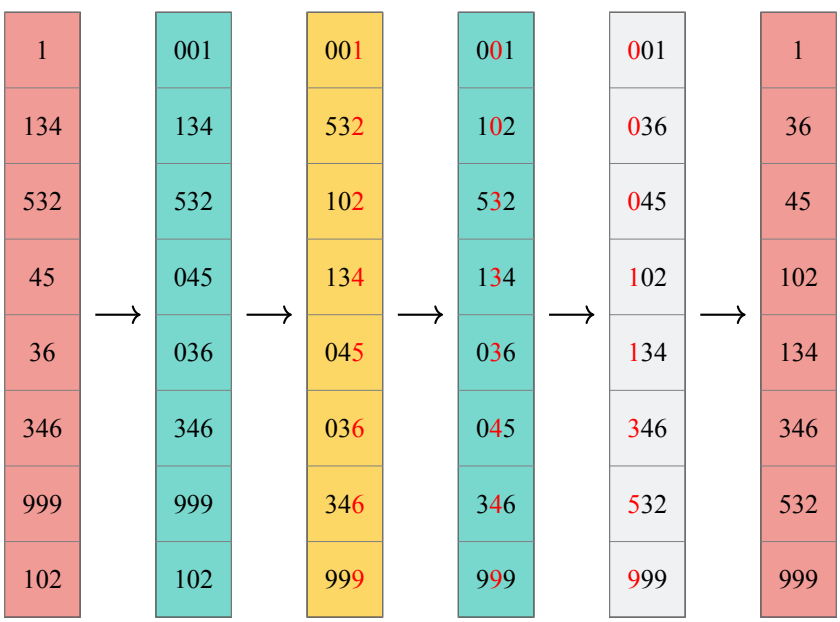


Figure 7.10: Radix sort

```rust
// radix_sort.rs

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

    // Find the largest number, which has the most digits.
    let max_num = match nums.iter().max() {
        Some(&x) => x,
        None => return,
    };

    // Find the power of 2 that is greater than or equals to
    // the length of nums as the bucket size. For example,
    // the closest and greater power of 2 to 10 is 2^4 = 16,
    // the closest and greater power of 2 to 17 is 2^5 = 32.
    let radix = nums.len().next_power_of_two();

    // The variable 'digit' represents the count of numbers
    // in a bucket that are less than a certain digit.
    // The ones, tens, hundreds, and thousands digits
    // correspond to positions 1, 2, 3, and 4, respectively.
    // The counting starts from the ones digit, so it is 1.
    let mut digit = 1;
    while digit <= max_num {
        // Calculate the position of the data in the bucket.
        let index_of = |x| x / digit % radix;
        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];
        }

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

        // process next bucket
        digit *= radix;
    }
}

fn main() {
    let mut nums = [0,54,32,99,18,75,31,43,56,21,22,100];
    radix_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [18, 21, 22, 31, 32, 43, 54, 56, 75, 99]
}
```

To sort by the same digit, stable sorting is used because it preserves the previous sorting result. For instance, sorting the tens digit preserves the sorting result of the ones digit, and sorting the hundreds digit preserves the sorting result of the tens digit. Binary can also be used to solve any non-negative integer sequence with the radix sorting algorithm. The time complexity is $O(64n)$ assuming the maximum integer in the sequence is 64 bits. Although comparison-based sorting algorithms have a time complexity of $O(nlog(n))$, they are still faster than radix sorting since the coefficient of 64 is too large to be practical. To use binary, k=2 is the smallest and the number of digits is the largest, so the time complexity becomes $O(nd)$ and the space complexity becomes $O(n + k)$, which is smaller. Conversely, when using the maximum value as the radix, k=maxV is the largest and the number of digits is the smallest, resulting in a smaller time complexity of $O(nd)$ but an increased space complexity of $O(n + k)$, leading to radix sorting degenerating into counting sorting.

In summary, these three non-comparison sorting methods (counting sorting, bucket sorting, and radix sorting) are interconnected. Counting sorting is a specific case of bucket sorting, and radix sorting degenerates into counting sorting if it uses the minimum number of digits to sort. Bucket sorting is suitable for evenly distributed elements, counting sorting requires a small difference between maxV and minV, and radix sorting can only handle positive numbers with maxV and minV being as close as possible. Therefore, these three sorting methods are only applicable for sorting small amounts of data, ideally less than 10000.

7.13 The Tim Sort

The TimSort hybrid sorting algorithm, proposed by Tim Peters in 2002, combines multiple sorting methods to address the limitations of individual algorithms. TimSort is efficient, stable, and adaptive to data distribution, making it superior to most sorting algorithms. It is currently the default sorting algorithm in many programming languages and platforms, including Java, Python, Rust etc.

TimSort is a hybrid stable sorting algorithm that uses merge and insertion sorts. When the number of elements to be sorted is less than 64, TimSort calls insertion sort directly. However, when the number of elements is greater than 64, TimSort combines insertion and merge sorts.

In practice, the data to be sorted often contains partially ordered blocks, where some parts of the data are already sorted in ascending or descending order, as shown in the figure below. TimSort leverages this characteristic of the data to sort it, and the existence of partially ordered blocks is the core of TimSort.

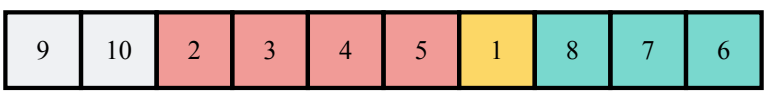


Figure 7.11: Partially ordered data

To sort the set of elements, TimSort divides them into runs or partitions. These runs can be seen as individual computational units. TimSort iterates through the elements and places them in different runs while merging the runs according to certain rules. This merging continues until only one run remains, which is the sorted result. To set the appropriate partition size, TimSort sets the minrun parameter. Each partition cannot have fewer elements than minrun. If a partition has fewer elements than minrun, the number of runs is expanded to minrun using insertion sort and then merged.

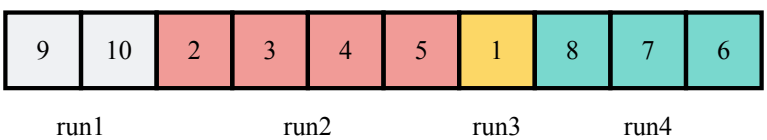


Figure 7.12: Data Runs

The approximate process of TimSort is as follows:

• Scan the set to be sorted to determine whether the number of elements is greater than 64.

• If it is less than or equal to 64, use the insertion sort algorithm to sort the data and return.

• If it is greater than 64, scan the set, calculate the minrun, and find various ordered blocks.

• Merge the blocks in pairs. If an individual block has fewer elements than minrun, use insertion sort to expand it.

• Repeat this process until only one block remains, which is the sorted set.

TimSort is a sorting algorithm that can be completed by addressing three important points: determining the minrun value, finding ordered blocks, and expanding and merging small blocks.

The first point is to determine the value of minrun. TimSort selects the first six binary digits of the length n of the collection to be sorted as the minrun. If the remaining digits are not 0, then minrun is incremented by 1. The range of minrun is from 32 to 64.

• For a collection of length n = 189, its binary representation is: 10111101. The first six digits, 101111, equal 47, and the remaining digits, 01, lead to a minrun value of 48.

• For a collection of length n = 976, its binary representation is: 1111010000. The first six digits, 111101, equal 61, and the remaining digits, 0000, lead to a minrun value of 61.

In fact, the maximum value of the first six binary digits is 111111 = 63, and the minimum is 100000 = 32. Therefore, the minrun value ranges from 32 to 64.

The second point is to find ordered blocks. TimSort finds ordered blocks by identifying increasing or decreasing sequences in the collection. This is achieved by an algorithm that only needs to judge the compare relationship of consecutive data to determine whether it is a block.

The third point is to expand and merge small blocks. Once the minrun value is calculated, TimSort adjusts any blocks that are in reverse order in place to be in the correct order. If the block has fewer elements than minrun, insertion sort is used to expand the block until it reaches minrun. The algorithm then checks the relationship between the current block and the previous two blocks. If the length of the three blocks does not satisfy $A > B + C, B > C$, then A and B or B and C are merged using insertion sort depending on the situation. If not, the algorithm continues to find the next block and repeat the block expansion and merging. Finally, the collection becomes several ordered blocks, and these merged blocks have lengths from longest to shortest. After the block division is completed, the small blocks are merged in pairs from the end of the collection to the beginning, and the collection becomes a large block, indicating that the sorting is completed.

The first two points are clear, but the third point may require further explanation. Reversing the order of a block is a simple process. To expand a block, new elements are inserted into the current block. However, merging blocks can be complicated as there are six possible scenarios.

When there are only two blocks, the algorithm only needs to consider the condition of $A > B$. When there are three blocks, the algorithm checks whether $A > B + C$ and $B > C$. Here are the illustrations of the six cases, where A is at the bottom of the stack, and B or C is at the top:

• $A > B$: Do not merge.

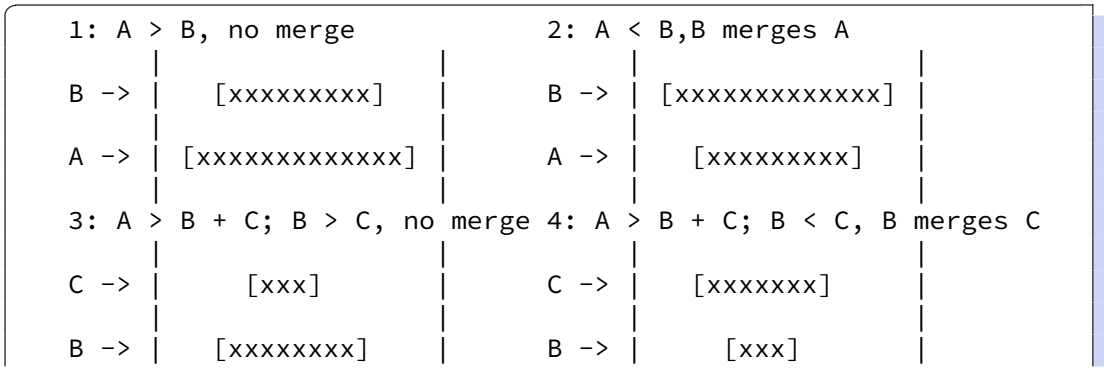
• $A > B + C$ and $B > C$: Do not merge.

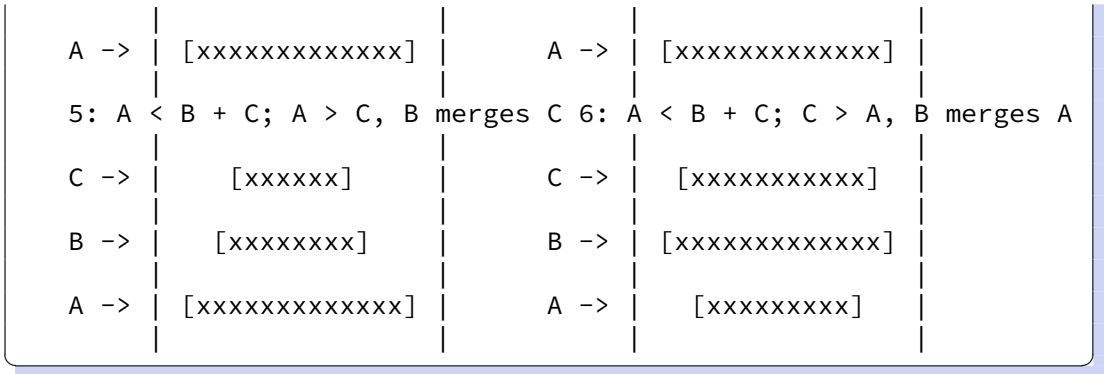
• $A < B + C$ and $A > C$: Merge run B with C.

• $A < B$: Merge run A with B.

• $A > B + C$ and $B < C$: Merge run B with C.

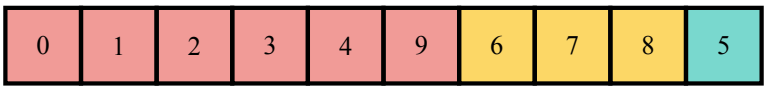
• $A < B + C$ and $A < C$: Merge run A with B.





The figure depicts the merging of three ordered blocks with a minimum run of 3. The characters A, B, and C represent the run blocks [xx], while the vertical lines represent a temporary stack used to merge these blocks. To determine whether and how to merge the blocks, the lengths of the three blocks are compared. The ideal merged state is shown in cases 1 and 3, while the other four merging scenarios are used to approximate these two cases.

The condition that $A > B + C, B > C$ is necessary to ensure efficient merging. For example, merging the block [6,7,8] directly with [0,1,2,3,4,9,5] would result in two remaining blocks with a large difference in length, making insertion sort inefficient. In contrast, when the blocks have similar lengths, merging them is very fast. Therefore, in this case, the blocks are not merged and are processed separately before being merged in reverse order at the end: [6,7,8] and [5], followed by [0,1,2,3,4,9] and [5,6,7,8].



To simplify the sorting process, a minimum run length is set, below which binary insertion sort is used. If the run length equals 64, minrun is also set to 64. Otherwise, a value between 32-64 is chosen for minrun such that $k = \frac{n}{minrun}$ is less than or equal to a power of 2. k represents the number of remaining blocks after scanning and processing, and their lengths must be sorted from largest to smallest. This enables merging from the tail, resulting in longer blocks and faster merging, akin to binary search. Hence, k is required to be less than or equal to a power of 2.

1	2	4	7	8	23	19	16	14	13	12	10	20	18	17	15	11	9	0	5	6	1	3	21	22
2	2	4	7	8	23	19	16	14	13	12	10	20	18	17	15	11	9	0	5	6	1	3	21	22
3	2	4	7	8	23	19	16	14	13	12	10	20	18	17	15	11	9	0	5	6	1	3	21	22
4	2	4	7	8	23	10	12	13	14	16	19	20	18	17	15	11	9	0	5	6	1	3	21	22
5	2	4	7	8	10	12	13	14	16	19	23	20	18	17	15	11	9	0	5	6	1	3	21	22
6	2	4	7	8	10	12	13	14	16	19	23	20	18	17	15	11	9	0	5	6	1	3	21	22
7	2	4	7	8	10	12	13	14	16	19	23	9	11	15	17	18	20	0	5	6	1	3	21	22
8	2	4	7	8	10	12	13	14	16	19	23	9	11	15	17	18	20	0	5	6	1	3	21	22
9	2	4	7	8	10	12	13	14	16	19	23	9	11	15	17	18	20	0	1	3	5	6	21	22
10	2	4	7	8	10	12	13	14	16	19	23	9	11	15	17	18	20	0	1	3	5	6	21	22
11	2	4	7	8	10	12	13	14	16	19	23	9	11	15	17	18	20	0	1	3	5	6	21	22
12	2	4	7	8	10	12	13	14	16	19	23	0	1	3	5	6	9	11	15	17	18	20	21	22
13	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Figure 7.13: TimSort

To illustrate the block expansion and merging mechanism, diagrams are presented above. For this example, minrun = 5, and each row represents an operation cycle. The leftmost column indicates the cycle number, and the square box contains the elements to be sorted.

The TimSort algorithm consists of several rounds of processing to sort a given set of data. In the first round, the algorithm obtains the data to be sorted from its parameters. In the second round, it searches for partitions and identifies a partition of length equal to the minimum run value, which is a predetermined set of values [2,4,7,8,23]. In the third round, it searches for the next partition and finds an inversion [19,16,14,13,12,10], which is then corrected to be in order as values [10,12,13,14,16,19]in the fourth round.

In the fifth round, the algorithm compares the length of the current partition to that of the previous partition and merges them using insertion sort if the current partition is longer. The result is [2,4,7,8,10,12,13,14,16,19,23], a new partition that is a combination of the two previous partitions. In the sixth round, it finds the next partition [20,18,17,15,11,9], detects an inversion, and corrects it to be in order in the seventh round. It then compares its length to the previous partition and continues searching for the next partition since it is shorter.

In the eighth round, the algorithm finds a new partition [0,5,6] that is shorter than the minimum run value and expands it using insertion sort. It then compares its length to the previous two partitions and determines that they satisfy certain conditions, which means that they do not need to be merged. This process continues until all the partitions are complete, which occurs after ten rounds.

In the eleventh round, the algorithm starts merging the smaller partitions from the end to the beginning([0,1,3,5,6] and [21, 22]). In the twelfth round, it merges the two remaining partitions [9,11,15,17,18,20] and [0,1,3,5,6,21,22], which are of similar length. Finally, in the thirteenth round, the algorithm performs the final merge to obtain the sorted output.

To implement TimSort, the process can be broken down into several steps. Firstly, the original data list and the minimum number of elements required for merging (MIN_MERGE) need to be prepared. Next, the starting position of each run and its corresponding run can be obtained through the partition process. After that, since merge sorting involves temporary stacks and processing of two runs, these data related to sorting tasks can be implemented in a structure. Finally, the actual merging process can be performed from the small partitions at the end to the beginning, with longer partitions merged with shorter ones and the partitions of decreasing length for easier merging.

```rust
// tim_sort_without_gallop.rs

// The minimum length of a sequence involved in merging,
// shorter than which insertion sort is used.
const MIN_MERGE: usize = 64;

// Sorting state structure
struct SortState<'a> {
    list: &'a mut [i32],
    runs: Vec<Run>, // store all run(s)
    pos: usize,
}

// Define the Run entity to save the starting index and
// interval length of the run in the list.
#[derive(Debug, Copy, Clone)]
struct Run {
    pos: usize,
    len: usize,
}

```

```
// merge_lo: a sorting state structure, used for merging
// sorting A and B.
struct MergeLo<'a> {
    list_len: usize,     // The length of the data
    first_pos: usize,    // first position of run1
    first_len: usize,    // length of run1
    second_pos: usize,   // first position of run2
    dest_pos: usize,     // index of the sorted result.
    list: &'a mut [i32], // partial interval of the data
                         // to be sorted
    temp: Vec<i32>,      // with a length set as the minimum
                         // value between the lengths of
                         // run1 and run2.
}

// merge_hi: a sorting state structure, used for merging
// sorting B and C.
struct MergeHi<'a> {
    first_pos: isize,
    second_pos: isize,
    dest_pos: isize,
    list: &'a mut [i32],
    temp: Vec<i32>,      // used for memory alignment
}
```

For blocks with less than MIN_MERGE elements, binary insertion sort can be used to speed up the sorting process. This method has already been implemented in the previous section, so no further code will be provided.

When using TimSort, the first step is to calculate the minrun value and identify the sorted runs (including reverse runs). If necessary, the reverse runs should be converted to regular runs.

```
// tim_sort_without_gallop.rs

// Calculate minrun, actual range is [32, 64]
fn calc_minrun(len: usize) -> usize {
    // If any bit in the low bits of len is 1, r is set to 1
    let mut r = 0;
    let mut new_len = len;
    while new_len >= MIN_MERGE {
        r |= new_len & 1;
        new_len >>= 1;
    }

    new_len + r
}

// Calculate the starting index of the run and convert
// the reverse run to a forward run
fn count_run(list: &mut [i32]) -> usize {
    let (ord, pos) = find_run(list);
    if ord { // convert reverse run to a forward run
        list.split_at_mut(pos).0.reverse();
    }
```

```
23
24      pos
25  }
26
27  // Determine whether the relationship between list[i]
28  // and list[i+1] is ascending or descending and
29  // return the inflection point index
30  fn find_run(list: &[i32]) -> (bool, usize) {
31      let len = list.len();
32      if len < 2 {
33          return (false, len);
34      }
35
36      let mut pos = 1;
37      if list[1] < list[0] {
38          // descending: list[i+1] <  list[i]
39          while pos < len - 1 && list[pos + 1] < list[pos] {
40              pos += 1;
41          }
42          (true, pos + 1)
43      } else {
44          // ascending: list[i+1] >= list[i]
45          while pos < len - 1 && list[pos + 1] >= list[pos] {
46              pos += 1;
47          }
48          (false, pos + 1)
49      }
50  }
```

Next, to sort the SortState, you will need to implement a constructor and a sort function. Finally, when the length of the partitions does not meet the requirements, partition merging needs to be performed using merge sort.

```
1   // tim_sort_without_gallop.rs
2
3   impl<'a> SortState<'a> {
4       fn new(list: &'a mut [i32]) -> Self {
5           SortState {
6               list: list,
7               runs: Vec::new(),
8               pos: 0,
9           }
10      }
11
12      fn sort(&mut self) {
13          let len = self.list.len();
14          // calculate the  minrun
15          let minrun = calc_minrun(len);
16
17          while self.pos < len {
18              let pos = self.pos;
19              let mut run_len = count_run(self.list
20                                              .split_at_mut(pos)
```

```
                                                .1);

            // check if the remaining number of elements is
            // less than minrun, if so,
            // let run_minlen = len - pos
            let run_minlen = if minrun > len - pos {
                len - pos
            } else {
                minrun
            };

            // If the run is very short, extend its length
            // to run_minlen, and the extended run needs to
            // be sorted, so use binary insertion sort
            if run_len < run_minlen {
                run_len = run_minlen;
                let left = self.list
                                .split_at_mut(pos).1
                                .split_at_mut(run_len).0;
                binary_insertion_sort(left);
            }

            // Stack the runs, with each run having a
            // different length
            self.runs.push(Run {
                pos: pos,
                len: run_len,
            });

            // Find the next run position
            self.pos += run_len;

            // Merge runs that do not conform to the
            // A > B + C and B > C rules
            self.merge_collapse();
        }


        // Forcefully merge all remaining runs from the top
        // of the stack until only one run remains,
        // completing the tim_sort sort
        self.merge_force_collapse();
    }

    // Merge runs to satisfy A > B + C and B > C
    // If A <= B + C, merge B with the shorter of A and C
    // If only A and B, and A <= B, merge A and B
    fn merge_collapse(&mut self) {
        let runs = &mut self.runs;
        while runs.len() > 1 {
            let n = runs.len() - 2;

```

```
            // check the relationships between A, B, C, and D
            // with D to prevent special cases of bugs
            // A <= B + C || D <= A + B
            if (n >= 1 && runs[n - 1].len
                <= runs[n].len + runs[n + 1].len)
                || (n >= 2 && runs[n - 2].len
                    <= runs[n].len + runs[n - 1].len)
            {
                // Determine the length relationship between
                // three consecutive runs A, B, and C,
                // and merge n-1 corresponding to A,
                // n corresponding to B, and n+1
                // corresponding to C
                let (pos1, pos2) = if runs[n-1].len
                                        < runs[n+1].len {
                    (n - 1, n) // merge A and B
                } else {
                    (n, n + 1) // merge B and C
                };

                // Take out run1 and run2 to be merged
                let (run1, run2) = (runs[pos1], runs[pos2]);
                debug_assert_eq!(run1.pos+run1.len, run2.pos);

                // Merge run to run1, i.e., update run1
                // and delete run2,
                // run1 index remains unchanged, and length
                // becomes the len(run1) + len(run2)
                runs.remove(pos2);
                runs[pos1] = Run {
                    pos: run1.pos,
                    len: run1.len + run2.len,
                };

                // Take the merged run1 for merge sort
                let new_list = self.list
                    .split_at_mut(run1.pos).1
                    .split_at_mut(run1.len + run2.len).0;
                merge_sort(new_list, run1.len, run2.len);
            } else {
                break;
            }
        }
    }

    // Forcefully merge any remaining runs until
    // only one run remains
    fn merge_force_collapse(&mut self) {
        let runs = &mut self.runs;
        while runs.len() > 1 {
            let n = runs.len() - 2;

```

```
125                 // Check the length relationship between three
126                 // consecutive runs A, B, and C,
127                 // and merge n-1 corresponding to A,
128                 // n corresponding to B, n+1 corresponding to C
129                 let (pos1, pos2) = if n > 0
130                        && runs[n - 1].len < runs[n + 1].len {
131                     (n - 1, n)
132                 } else {
133                     (n, n + 1)
134                 };
135 
136                 // Take out run1 and run2 to be merged
137                 let (run1, run2) = (runs[pos1], runs[pos2]);
138                 debug_assert_eq!(run1.len, run2.pos);
139 
140                 // Merge run to run1, i.e., update run1
141                 // and delete run2,
142                 // run1 index remains unchanged, and length
143                 // becomes the sum of the lengths of run1 and run2
144                 runs.remove(pos2);
145                 runs[pos1] = Run {
146                     pos: run1.pos,
147                     len: run1.len + run2.len,
148                 };
149 
150                 // Take the merged run1 for merge sort
151                 let new_list = self.list
152                     .split_at_mut(run1.pos).1
153                     .split_at_mut(run1.len + run2.len).0;
154                 merge_sort(new_list, run1.len, run2.len);
155             }
156         }
157     }
```

There are six cases of partitioning, which may require merging A and B or merging B and C. Since A, B, and C are contiguous in memory, merging A and B can be implemented separately from merging B and C, taking advantage of their relative positions.

```
1  // tim_sort_without_gallop.rs
2  
3  // merge sort for A, B and C
4  fn merge_sort(
5      list: &mut [i32],
6      first_len: usize,
7      second_len: usize)
8  {
9      if 0 == first_len || 0 == second_len { return; }
10 
11     if first_len > second_len {
12         // Merge B and C, and merge from the end
13         // of the list using temp
14         merge_hi(list, first_len, second_len);
15     } else {
```

```
16          // Merge B and A, and merge from the beginning
17          // of the list using temp
18          merge_lo(list, first_len);
19      }
20  }
21
22  // merge A and B to a single run
23  fn merge_lo(list: &mut [i32], first_len: usize) {
24      unsafe {
25          let mut state = MergeLo::new(list, first_len);
26          state.merge();
27      }
28  }
29
30  impl<'a> MergeLo<'a> {
31      unsafe fn new(list: &'a mut [i32], first_len: usize) ->
            Self {
32          let mut ret_val = MergeLo {
33              list_len: list.len(),
34              first_pos: 0,
35              first_len: first_len,
36              second_pos: first_len, // run1 close to run2,
37                                    // the start index of run2
38                                    // is exactly equals the
39                                    // length of run1
40              dest_pos: 0,          // write the sorted data
41                                    // back to run1
42              list: list,
43              temp: Vec::with_capacity(first_len),
44          };
45
46          // copy run1 to temp
47          ret_val.temp.set_len(first_len);
48          for i in 0..first_len {
49              ret_val.temp[i] = ret_val.list[i];
50          }
51
52          ret_val
53      }
54
55      // merge sort
56      fn merge(&mut self) {
57          while self.second_pos > self.dest_pos
58               && self.second_pos < self.list_len {
59              debug_assert!((self.second_pos - self.first_len) +
60                  self.first_pos == self.dest_pos);
61
62              if self.temp[self.first_pos]
63                      > self.list[self.second_pos] {
64                  self.list[self.dest_pos]
65                      = self.list[self.second_pos];
66                  self.second_pos += 1;
```

```
67              } else {
68                  self.list[self.dest_pos]
69                      = self.temp[self.first_pos];
70                  self.first_pos += 1;
71              }
72              self.dest_pos += 1;
73          }
74      }
75  }
76
77  // clear temp stack
78  impl<'a> Drop for MergeLo<'a> {
79      fn drop(&mut self) {
80          unsafe {
81              // put left date of temp into list(high digit)
82              if self.first_pos < self.first_len {
83                  for i in 0..(self.first_len - self.first_pos) {
84                      self.list[self.dest_pos + i]
85                          = self.temp[self.first_pos + i];
86                  }
87              }
88
89              // set the lenght of temp to 0
90              self.temp.set_len(0);
91          }
92      }
93  }
94
95  // merge B and C as a new run block
96  fn merge_hi(
97      list: &mut [i32],
98      first_len: usize,
99      second_len: usize)
100 {
101     unsafe {
102         let mut state = MergeHi::new(list,
103                                      first_len,
104                                      second_len);
105         state.merge();
106     }
107 }
108
109 impl<'a> MergeHi<'a> {
110     unsafe fn new(
111         list: &'a mut [i32],
112         first_len: usize,
113         second_len: usize) -> Self
114     {
115         let mut ret_val = MergeHi {
116             first_pos: first_len as isize - 1,
117             second_pos: second_len as isize - 1,
118             dest_pos: list.len() as isize - 1,// from the tail
```

```
119             list: list,
120             temp: Vec::with_capacity(second_len),
121         };
122
123         // copy run2 to temp
124         ret_val.temp.set_len(second_len);
125         for i in 0..second_len {
126             ret_val.temp[i] = ret_val.list[i + first_len];
127
128         }
129
130         ret_val
131     }
132
133     // merge sort
134     fn merge(&mut self) {
135         while self.first_pos < self.dest_pos
136              && self.first_pos >= 0 {
137             debug_assert!(self.first_pos + self.second_pos + 1
138                 == self.dest_pos);
139             if self.temp[self.second_pos as usize]
140                 >= self.list[self.first_pos as usize] {
141                 self.list[self.dest_pos as usize]
142                     = self.temp[self.second_pos as usize];
143                 self.second_pos -= 1;
144             } else {
145                 self.list[self.dest_pos as usize]
146                     = self.list[self.first_pos as usize];
147                 self.first_pos -= 1;
148             }
149             self.dest_pos -= 1;
150         }
151     }
152 }
153
154 // clear temp stack
155 impl<'a> Drop for MergeHi<'a> {
156     fn drop(&mut self) {
157         unsafe {
158             // put left date of temp into list(low digit)
159             if self.second_pos >= 0 {
160                 let size = self.second_pos + 1;
161                 let src  = 0;
162                 let dest = self.dest_pos - size;
163                 for i in 0..size {
164                     self.list[(dest + i) as usize]
165                         = self.temp[(src + i) as usize];
166                 }
167             }
168
169             // set the length of temp stack to 0
170             self.temp.set_len(0);
```

```
171             }
172         }
173     }
```

Here is the main function of TimSort.

```
1  // timSort entry point
2
3  fn tim_sort(list: &mut [i32]) {
4      if list.len() < MIN_MERGE {
5         binary_insertion_sort(list);
6      } else {
7         let mut sort_state = SortState::new(list);
8         sort_state.sort();
9      }
10 }
```

Here is an example of TimSort.

```
1  fn main() {
2      let mut nums: Vec<i32> = vec![
3           2,  4,  7,  8, 23, 19, 16, 14, 13, 12, 10, 20,
4          18, 17, 15, 11,  9, -1,  5,  6,  1,  3, 21, 40,
5          22, 39, 38, 37, 36, 35, 34, 33, 24, 30, 31, 32,
6          25, 26, 27, 28, 29, 41, 42, 43, 44, 45, 46, 47,
7          48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
8          60, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70,
9          61, 62, 63, 64, 65, 66, 67, 68, 69, 95, 94, 93,
10         92, 91, 90, 85, 82, 83, 84, 81, 86, 87, 88, 89,
11     ];
12     tim_sort(&mut nums);
13     println!("sorted nums: {:?}", nums);
14 }
```

```
sorted nums: [-1,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
              12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
              24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
              36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
              48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
              60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
              72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
              84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95]
```

The implemented TimSort algorithm only works for i32 type numbers, but it can be extended to support sorting algorithms for various types of numbers through the use of generics. Additionally, during merging, some data may already be sorted, but the implemented TimSort algorithm still goes through each comparison one by one. However, merging can be accelerated through a strategy called galloping. The TimSort implementation in this book's source code in tim_sort_without_gallop.rs is the non-accelerated version, but an accelerated version of Timsort has also been implemented in tim_sort.rs. Interested readers can refer to and compare the differences between the two algorithms.

Some readers may wonder how TimSort determines if the data to be sorted is block-sorted originally. In reality, the algorithm cannot determine this; it is a property of the partially ordered data that the author discovered. In physics, there is a concept of entropy [13] that refers to the degree of disorder in a physical

system. The more disorderly a system is, the greater the entropy, and conversely, the more ordered it is, the smaller the entropy. In most cases, things have a certain degree of order. For example, humans are ordered animals in reverse entropy, and only become disordered with death. Another example is the principle of locality of reference [14], where when accessing data on a hard drive, the CPU will read the data around it into memory because it is likely to be accessed next. These two phenomena are natural laws that are consistent with statistical principles, and Tim wrote TimSort based on these laws. This illustrates the importance of data structures and how different understandings can lead to very different algorithms.

7.14 Summary

In this chapter, we learned about ten different types of sorting algorithms. Bubble sort, selection sort, and insertion sort are $O(n^2)$ algorithms, while most other sorting algorithms have a complexity of $O(nlog_2(n))$. Selection sort is an improvement on bubble sort, shell sort improves on insertion sort, heap sort improves on selection sort, and quicksort and merge sort both use the divide-and-conquer approach.

All of these sorts are based on comparison, but there are also non-comparison sorts that rely solely on numerical patterns for sorting, such as bucket sort, counting sort, and radix sort. These sorts have a complexity of approximately $O(n)$ and are suitable for sorting small amounts of data. Counting sort is a special case of bucket sort, and radix sort is a multi-round bucket sort that can degrade into counting sort.

In addition to the basic sorting algorithms, we also learned about improved versions of some algorithms, particularly the TimSort algorithm, which is an efficient and stable hybrid sorting algorithm. Its improved version is already the default sorting algorithm for many languages and platforms.

The table below summarizes the various sorting algorithms, which readers can compare and understand on their own to deepen their understanding.

Table 7.1: Time and space complexity of various sorting algorithms

No	Algorithm	Time-worst	Time-best	Time-average	Space	Stability
1	Bubble sort	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	stable
2	Quick sort	$O(n^2)$	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	unstable
3	Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	unstable
4	Heap sort	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(1)$	unstable
5	Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	stable
6	Shell sort	$O(n^2)$	$O(n)$	$O(n^{1.3})$	$O(1)$	unstable
7	Merge sort	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(n)$	stable
8	Counting sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	stable
9	Bucket sort	$O(n^2)$	$O(n)$	$O(n+k)$	$O(n+k)$	stable
10	Radix sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$	stable
11	Tim sort	$O(nlog(n))$	$O(n)$	$O(nlong(n))$	$O(n)$	stable

Chapter 8

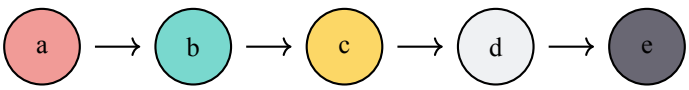
Trees

8.1 Objectives

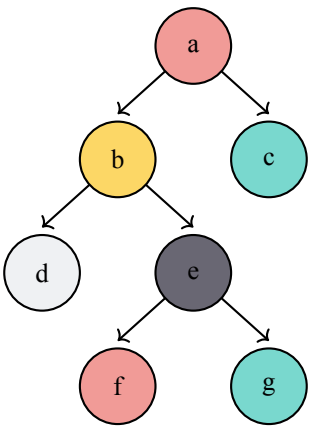
- Understanding trees and their usage.
- Implementing a priority queue using a binary heap.
- Understanding binary search trees and balanced binary trees.
- Implementing binary search trees and balanced binary trees.

8.2 What is Tree?

In previous chapters, we covered linear data structures such as stacks, queues, and linked lists, where each data item is connected to the next one.

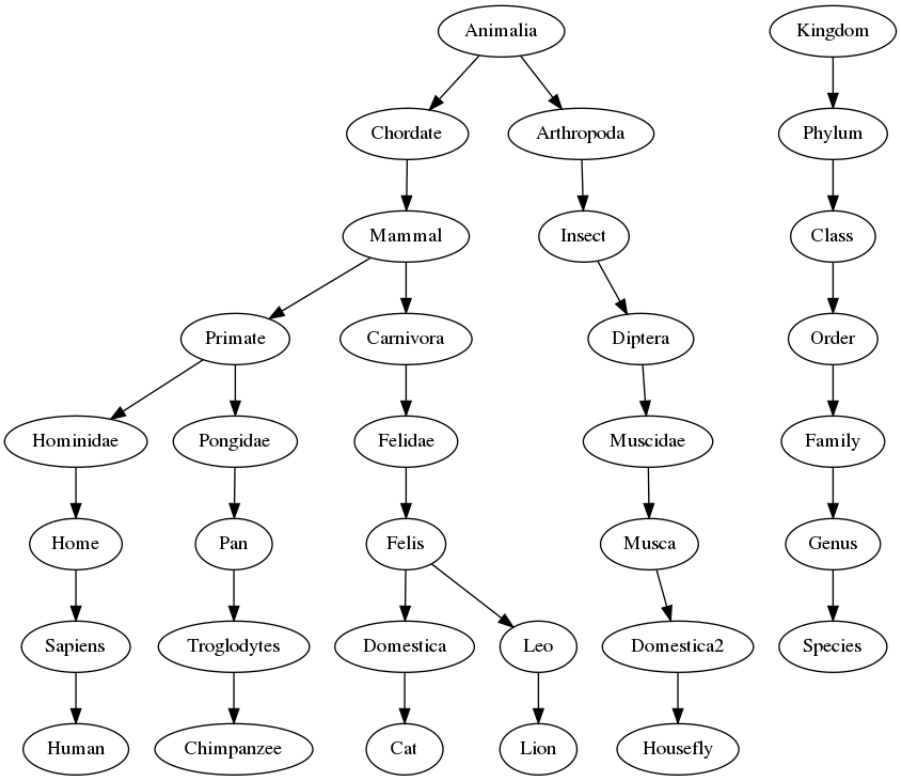


A tree is a new data structure that expands upon this linear structure by connecting multiple data items.



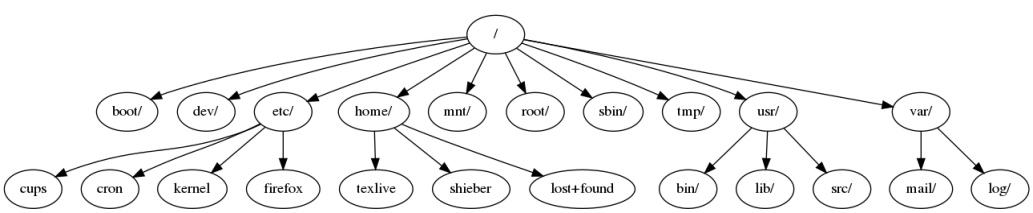
Like a natural tree, this data structure has a root, branches, and leaves, which are interconnected. Trees are used in various fields of computer science, such as operating systems, graphics, databases, and computer networks. Throughout the rest of the text, we will refer to this data structure as a ”tree.”

Before delving into trees, it's essential to understand some common tree examples, such as a biological classification tree. From this graph, we can see the specific location of entry 'People' (lower left), which is very helpful for studying relationships and properties.



This type of tree is hierarchical, with well-defined structures comprising seven levels: kingdom, phylum, class, order, family, genus, and species. The highest level represents the most abstract concept, while the lowest level represents the most specific. By starting from the root of the tree and following the arrows to the bottom, we can obtain a complete path that indicates the full name of a species. Organisms can find their place on this tree of life, displaying their relative relationships. For instance, the complete name of the "human" species is "animal kingdom-chordate phylum-mammal class-primate order-hominidae family-homo genus-homo sapiens species."

Each node's set of child nodes is independent of another node's set of child nodes, which makes the relationships between nodes clear. This property also means that modifying one node's child nodes will not affect other nodes. It is especially useful when trees are used as data storage containers, as it allows tools to modify data on specific nodes while keeping other data unchanged.



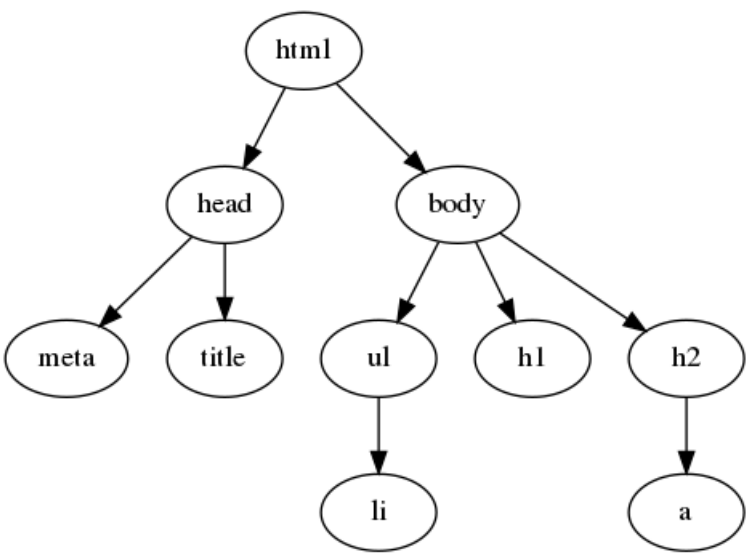
Lastly, each leaf node is unique, and there is only one unique path from the root to each leaf node. This property makes data storage efficient since the unique path can be used as a storage path. The file system in our computers is based on an improved tree structure. The file system tree and biological classification tree have many similarities, and the path from the root directory to any subdirectory uniquely identifies

that subdirectory and all files within it. If you're familiar with Unix-like operating systems, you should recognize paths like /root, /home/user, and /etc as nodes on a tree, where / is the root node.

Trees can also be used to represent web page files, which are collections of resources that have a hierarchical structure. For example, the web page data of Google.cn's search interface shows that the <> tag elements are also arranged hierarchically.

```
<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>
    <h1><a href="http://www.google.com.hk/webhp?hl=zh-CN">
        <strong id="target">google.com.hk</strong></a></h1>
    <p>Please save our website</p>
  </div>
  <ul>
    <li><a href="http://translate.google.cn/">translate</a></li
      >
  </ul>
  </body>
</html>
```

In web page files, the HTML root encapsulates all the other elements, as illustrated in the diagram below. This hierarchical structure allows web pages to be displayed in a consistent manner across different devices and browsers. By using trees to represent web page files, we can efficiently store and retrieve data and modify individual elements without affecting the entire page's layout.



8.2.1 Vocabularies and Definitions of Tree

After discussing some examples of trees, we will now define the various properties of a tree.

• Node: A node is the fundamental unit of a tree, consisting of a ”key” name and an optional ”payload” information. Payload information is not always necessary for tree algorithms, but it is typically important in applications that require additional information on nodes, such as file names, file content, or timestamps.

• Root: The root is the top-level node in a tree, with no incoming edges. All nodes in a tree can be reached by following edges from the root. This is analogous to the concepts of the / or C drive in an operating system.

• Edge: An edge connects two nodes and represents the relationship between them. Every node in a tree, except the root, has exactly one incoming edge and potentially several outgoing edges. An edge defines a path that can be used to locate a specific node.

• Path: A path is a sequence of edges that connect nodes. It is not a structure in itself, but rather an associative relationship that emerges from the logical arrangement of edges. For instance, /home/user/-files/sort.rs is a path that identifies the location of the file sort.rs.

• Child Node: A child node is a node that is one level below another node, originating from the same parent node. For example, sort.rs is a child node of files/. Child nodes are not unique, and there may be zero, one, or many child nodes, similar to the parent-child relationship in human society. Hence, child nodes are often named after kinship terms.

• Parent Node: A parent node is a node from which all lower-level nodes originate. For instance, files/ is the parent node of sort.rs. The parent node is unique, as a child can only have one parent.

• Subtree: A subtree is a set of nodes and edges that consists of a parent node and all its descendants. Since a tree is a recursive structure, taking any node from the tree creates a new subtree.

• Leaf Node: A leaf node is a node without child nodes, at the bottom level of the tree.

• Internal Node: An internal node has child nodes and a parent node.

• Depth: The depth of a node is the number of edges traversed from the root to that node. The depth of the root node is zero. In /home/user/files, the depth of files is 2. A depth of zero does not mean that there is no depth, but rather the depth is zero, indicating the first level.

• Height: The height of a tree is the maximum depth of any node in the tree.

To define a tree, we start with the foundational knowledge presented above:

• A tree has a root node, which is the topmost node in the hierarchy.

• Each node, except for the root node, is connected to a parent node and may have zero or more child nodes, forming a branching structure.

• The path from the root node to any other node in the tree is unique, meaning there is only one path to reach that node.

The following image shows an example of a tree structure with left and right child nodes labeled as ”lc” and ”rc,” respectively. As trees are recursive structures, the structure of a subtree is the same as that of its parent node.

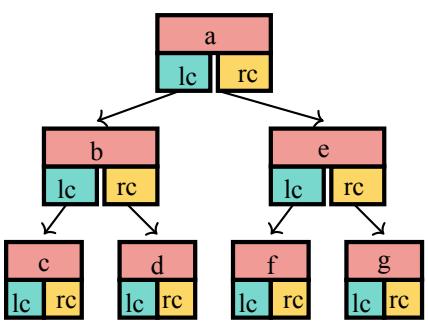


Figure 8.1: tree

8.2.2 Tree Representation

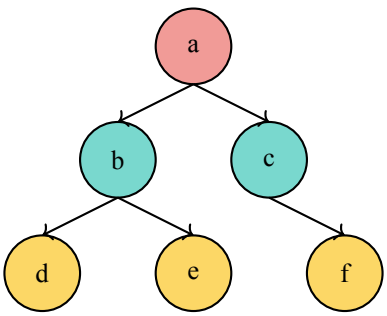
Trees are a powerful non-linear data structure, but representing them in a computer requires linear data structures. One common approach to representing a tree is using an array. In this approach, the elements of the tree are stored in an array, with the root at the first index and the child nodes following in a logical order.

```
tree = [ 'a',
           ['b',
             ['c',[],[]],
             ['d',[],[]],
           ],
           ['e',
             ['f',[],[]],
             []
           ],
       ]
```

Array access methods can be used to access the tree elements, making it easy to obtain subtrees and elements. However, this approach becomes cumbersome and impractical for larger trees with many levels, as accessing deeper nodes requires complex nested access methods.

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

An alternative approach to representing trees is to use nodes. In this approach, each node in the tree is represented by an object or a struct, which stores the node's key and payload information, as well as links to its child nodes. The links between nodes form the edges of the tree. This approach is similar to the linked list data structure, where each node has a link to the next node. The node-based representation of a tree is more flexible than the array-based approach and can handle larger trees with ease. The figure below shows an example of a binary tree represented using nodes.



This structure is straightforward and avoids the complexity of nested arrays for accessing elements. The key now is how to define tree nodes. One feasible method is to use the struct to define a node.

```
1 use std::cmp::{max, Ordering::*};
2 use std::fmt::{Debug, Display};
3
4 // Binary tree child node link
5 type Link<T> = Option<Box<BinaryTree<T>>>;
6
7 // Binary tree definition
```

```
#[derive(Debug, Clone, PartialEq)]
struct BinaryTree<T> {
    key: T,          // data storage
    left: Link<T>,  // left child node address storage
    right: Link<T>, // right child node address storage
}
```

The struct has a key that stores data and left and right pointers that store the addresses of the left and right child nodes.

```
// binary_tree.rs

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

    // New child node is added as the
    // left child node of the root node
    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 mut node = BinaryTree::new(key);
            node.left = self.left.take();
            self.left = Some(Box::new(node));
        }
    }

    // New child node is added as the
    // right child node of the root node
    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 mut node = BinaryTree::new(key);
            node.right = self.right.take();
            self.right = Some(Box::new(node));
        }
    }
}
```

When inserting a new child node into a tree, there are two possible cases to consider. The first case is when the node has no child nodes, in which case the new node can be directly inserted as its child. The second case is when the node already has child nodes, in which case the new node's child nodes must first be attached to the appropriate positions in the tree before the new node itself can be added as a child of the original node.

The code defining the tree nodes may be confusing since it uses the term "BinaryTree" instead of "Node." However, when multiple nodes are linked together to form a tree, the entire subtree can be considered a single node, which simplifies the insertion process. It is important to note, however, that the internal structure of the node is still relevant, so it is written as "BinaryTree" rather than "Node."

Nonetheless, either term can be used to implement the tree.

Below is the code for calculating the number of different types of nodes and the depth of the tree.

```
// binary_tree.rs

impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
    // calculate total node number
    fn size(&self) -> usize {
        self.calc_size(0)
    }

    fn calc_size(&self, mut size: usize) -> usize {
        size += 1;

        if !self.left.is_none() {
            size =self.left.as_ref().unwrap().calc_size(size);
        }
        if !self.right.is_none() {
            size =self.right.as_ref().unwrap().calc_size(size);
        }

        size
    }

    // calculate the total leaf nodes number
    fn leaf_size(&self) -> usize {
        // both left and right nodes are none
        // which means current node is a leaf node
        if self.left.is_none() && self.right.is_none() {
            return 1;
        }

        // calculate the total leaf nodes of
        // both left and right substrees
        let left_leaf = match &self.left {
            Some(left) => left.leaf_size(),
            None => 0,
        };
        let right_leaf = match &self.right {
            Some(right) => right.leaf_size(),
            None => 0,
        };

        // leaf nodes = leaf nodes(left) + leaf nodes(right)
        left_leaf + right_leaf
    }

    // calculate non-leaf nodes,
    // it is actually very easy to calculate
    fn none_leaf_size(&self) -> usize {
        self.size() - self.leaf_size()
    }
```

```
50
51      // calculate the depth of a tree
52      fn depth(&self) -> usize {
53          let mut left_depth = 1;
54          if let Some(left) = &self.left {
55              left_depth += left.depth();
56          }
57
58          let mut right_depth = 1;
59          if let Some(right) = &self.right {
60              right_depth += right.depth();
61          }
62
63          // return the max depth
64          max(left_depth, right_depth)
65      }
66  }
```

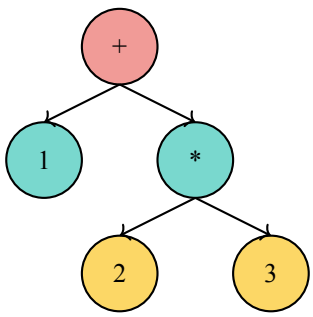
To manipulate the data of a binary tree node, you need to implement methods for accessing the left and right child nodes, setting and getting the root node value, and modifying node values. Additionally, methods for determining the existence of a node value and finding the maximum and minimum node values can also be helpful.

```
1   // binary_tree.rs
2
3   impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
4       // get left subtree
5       fn get_left(&self) -> Link<T> {
6           self.left.clone()
7       }
8
9       fn get_right(&self) -> Link<T> {
10          self.right.clone()
11      }
12
13      // get and set key
14      fn get_key(&self) -> T {
15          self.key.clone()
16      }
17
18      fn set_key(&mut self, key: T) {
19          self.key = key;
20      }
21
22      // find min/max key in the tree
23      fn min(&self) -> Option<&T> {
24          match self.left {
25              None => Some(&self.key),
26              Some(ref node) => node.min(),
27          }
28      }
29
30      fn max(&self) -> Option<&T> {
```

```
31              match self.right {
32                  None => Some(&self.key),
33                  Some(ref node) => node.max(),
34              }
35          }
36
37          // determine whether a key is in the tree
38          fn contains(&self, key: &T) -> bool {
39              match &self.key.cmp(key) {
40                  Equal => true,
41                  Greater => {
42                      match &self.left {
43                          Some(left) => left.contains(key),
44                          None => false,
45                      }
46                  },
47                  Less => {
48                      match &self.right {
49                          Some(right) => right.contains(key),
50                          None => false,
51                      }
52                  },
53              }
54          }
55      }
```

8.2.3 Parse Tree

The following is an explanation of how a binary tree can be used to store data, using the example of storing the expression (1 + (2 * 3)). The expression is fully parenthesized, which indicates the order of operations. The tree structure that corresponds to this expression is shown in the accompanying diagram.



To store data in the tree, we follow certain rules based on the tree's structure. These rules are as follows:

- If the current symbol is (, a new node is added as the left child node, and we descend to that node.
- If the current symbol is one of "+", "-", "*", or "/", we set the root value to that symbol, add a new right child node, and descend to the right child node.
- If the current symbol is a number, we set the root value to that number and return to the parent node.
- If the current symbol is), we return to the parent node of the current node.

To convert the mathematical expression (1 + (2 * 3)) into the tree shown in the figure, we can use the data storage rules defined for the tree. The specific steps are as follows:

(1) Create the root node.

(2) Read the symbol (, create a new left child node, and descend to the node.

(3) Read the symbol 1, set the node value to 1, and return to the parent node.

(4) Read the symbol +, set the node value to +, create a new right child node, and descend to the node.

(5) Read the symbol (, create a new left child node, and descend to the node.

(6) Read the symbol 2, set the node value to 2, and return to the parent node.

(7) Read the symbol *, set the node value to *, create a new right child node, and descend to the node.

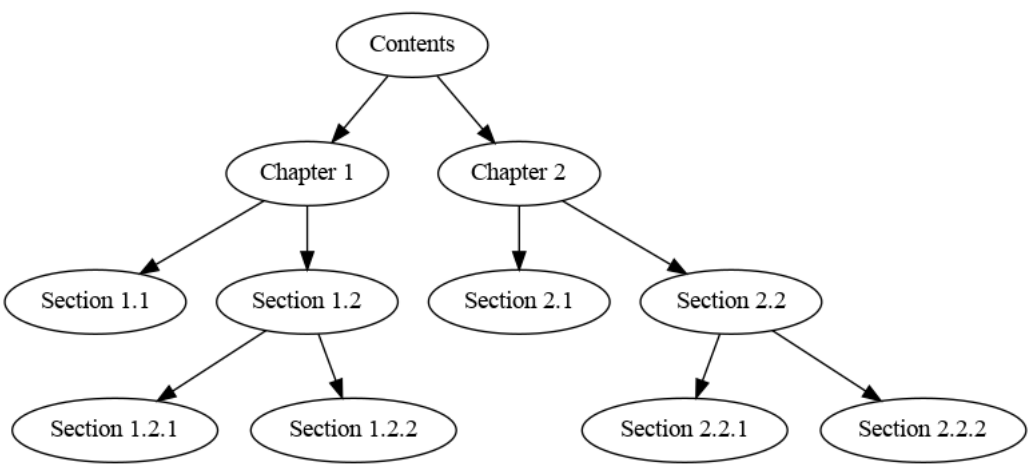
(8) Read the symbol 3, set the node value to 3, and return to the parent node.

(9) Read the symbol), return to the parent node.

Using a tree allows for the storage of the arithmetic expression while maintaining the structural information of the data. In fact, programming languages use trees to store all code during compilation and generate abstract syntax trees. By analyzing the functionality of each part of the syntax tree, intermediate code is generated, optimized, and then final code is generated. If you understand the principles of compilation, this should be familiar to you.

8.2.4 Tree Traversals

The purpose of data storage is to efficiently perform operations such as adding, deleting, searching, and modifying data. In a tree structure, locating a specific node is essential to perform these operations. However, unlike linear data structures that can be traversed using indexes, trees are nonlinear structures, so the search method is different. To access nodes in a tree, there are three commonly used methods that differ in the order in which nodes are visited. These methods are pre-order traversal, in-order traversal, and post-order traversal, and they are analogous to the traversal methods of linear data structures. To illustrate these methods, imagine this book as a tree with the table of contents as the root, each chapter as its child nodes, and each subsection as the child nodes of their respective chapters.



Pre-order traversal is the first method, and it starts from the root node, then moves to the left subtree, and finally the right subtree. To traverse a tree algorithmically using pre-order traversal, recursively call the pre-order traversal method on the left subtree from the root node. For example, in the book tree described above, the pre-order traversal algorithm visits the table of contents first, followed by chapter 1, section 1, section 2, and so on. Once chapter 1 is finished, the algorithm returns to the table of contents and recursively calls pre-order traversal on chapter 2, visiting chapter 2, section 1, section 2, and so on.

The pre-order traversal algorithm is simple and can be implemented as an internal method or an external function.

```
// binary_tree.rs

impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
    fn preorder(&self) {
        println!("key: {:?}", &self.key);
        match &self.left {
            Some(node) => node.preorder(),
            None => (),
        }
        match &self.right {
            Some(node) => node.preorder(),
            None => (),
        }
    }
}

// pre-order: implemented externally [by recursion]
fn preorder<T: Clone + Ord + ToString + Debug>(bt: Link<T>) {
    if !bt.is_none() {
        println!("key: {:?}", bt.as_ref().unwrap().get_key());
        preorder(bt.as_ref().unwrap().get_left());
        preorder(bt.as_ref().unwrap().get_right());
    }
}
```

The post-order traversal starts from the left subtree, followed by the right subtree, and finally the root node.

```
// binary_tree.rs

impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
    fn postorder(&self) {
        match &self.left {
            Some(node) => node.postorder(),
            None => (),
        }
        match &self.right {
            Some(node) => node.postorder(),
            None => (),
        }
        println!("key: {:?}", &self.key);
    }
}

// post-order: implemented externally [by recursion]
fn postorder<T: Clone + Ord + ToString + Debug>(bt: Link<T>) {
    if !bt.is_none() {
        postorder(bt.as_ref().unwrap().get_left());
        postorder(bt.as_ref().unwrap().get_right());
        println!("key: {:?}", bt.as_ref().unwrap().get_key());
```

```
23      }
24  }
```

The in-order traversal starts from the left subtree, followed by the root node, and finally the right subtree.

```
1   // binary_tree.rs
2
3   impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
4       fn inorder(&self) {
5           if self.left.is_some() {
6               self.left.as_ref().unwrap().inorder();
7           }
8           println!("key: {:?}", &self.key);
9           if self.right.is_some() {
10              self.right.as_ref().unwrap().inorder();
11          }
12      }
13  }
14
15  // in-order: implemented externally [by recursion]
16  fn inorder<T: Clone + Ord + ToString + Debug>(bt: Link<T>) {
17      if !bt.is_none() {
18          inorder(bt.as_ref().unwrap().get_left());
19          println!("key: {:?}", bt.as_ref().unwrap().get_key());
20          inorder(bt.as_ref().unwrap().get_right());
21      }
22  }
```

To evaluate an arithmetic expression like (1 + (2 * 3)) that is stored in a tree, we need to retrieve the operator and operands in the correct order. While pre-order traversal starts as much as possible from the root node, it is not suitable for computing expressions as we need to start from the leaf nodes. Post-order traversal, on the other hand, is the correct method to retrieve data as it starts from the left subtree, followed by the right subtree, and finally the root node. By first retrieving the values of the left and right child nodes, then retrieving the operator of the root node, we can perform one operation and save the result in the position of the operator, and then continue post-order traversal by visiting the right child node until we calculate the final value.

Additionally, in-order traversal can be used on the tree that saves the arithmetic expression (1 + (2 * 3)) to retrieve the original expression 1 + 2 * 3. However, since the tree does not save parentheses, the recovered expression is only in the correct order, but the priority may not be correct. To include parentheses in the output, we can modify the in-order traversal.

```
1   // binary_tree.rs
2
3   impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
4       // form a expression: [internal implementation]
5       fn iexp(&self) -> String {
6           let mut exp = "".to_string();
7
8           exp += "(";
9           let exp_left = match &self.left {
10              Some(left) => left.iexp(),
11              None => "".to_string(),
```

```
        };
        exp += &exp_left;

        exp += &self.get_key().to_string();

        let exp_right = match &self.right {
            Some(right) => right.iexp(),
            None => "".to_string(),
        };
        exp += &exp_right;
        exp += ")";

        exp
    }
}

// form a expression: [external implementation]
fn oexp<T>(bt: Link<T>) -> String
  where: T: Clone + Ord + ToString + Debug + Display
{
    let mut exp = "".to_string();
    if !bt.is_none() {
        exp = "(".to_string() +
              &oexp(bt.as_ref().unwrap().get_left());
        exp += &bt.as_ref().unwrap().get_key().to_string();
        exp += &(oexp(bt.as_ref().unwrap().get_right()) + ")");
    }

    exp
}
```

In addition to the three traversal methods discussed earlier, there is another method called level-order traversal that visits the nodes layer by layer. As we have previously implemented the queue data structure required for level-order traversal, we can directly use it to implement this traversal method.

```
// binary_tree.rs

impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
    fn levelorder(&self) {
        let size = self.size();
        let mut q = Queue::new(size);

        // enqueue the root node
        let _r = q.enqueue(Box::new(self.clone()));
        while !q.is_empty() {
            // dequeue the first node, and output its value
            let front = q.dequeue().unwrap();
            println!("key: {:?}", front.get_key());

            // enqueue the left/right child node
            match front.get_left() {
                Some(left) => {
                  let _r = q.enqueue(left);
```

```
                },
                None => {},
            }

            match front.get_right() {
                Some(right) => {
                  let _r = q.enqueue(right);
                },
                None => {},
            }
        }
    }
}

// level-order: implemented externally [by recursion]
fn levelorder<T: Clone + Ord + ToString + Debug>(bt: Link<T>) {
    if bt.is_none() { return; }

    let size = bt.as_ref().unwrap().size();
    let mut q = Queue::new(size);

    let _r = q.enqueue(bt.as_ref().unwrap().clone());
    while !q.is_empty() {
        // dequeue the first node, and print its value
        let front = q.dequeue().unwrap();
        println!("key: {:?}", front.get_key());

        match front.get_left() {
            Some(left) => {
              let _r = q.enqueue(left);
            },
            None => {},
        }

        match front.get_right() {
            Some(right) => {
              let _r = q.enqueue(right);
            },
            None => {},
        }
    }
}
```

Here are some use examples of binary trees.

```
// binary_tree.rs

fn main() {
    basic();
    order();

    fn basic() {
        let mut bt = BinaryTree::new(10usize);
```

```
 9
10          let root = bt.get_key();
11          println!("root key: {:?}", root);
12
13          bt.set_key(11usize);
14          let root = bt.get_key();
15          println!("root key: {:?}", root);
16
17          bt.insert_left_tree(2usize);
18          bt.insert_right_tree(18usize);
19
20          println!("left child: {:#?}", bt.get_left());
21          println!("right child: {:#?}", bt.get_right());
22
23          println!("min key: {:?}", bt.min().unwrap());
24          println!("max key: {:?}", bt.max().unwrap());
25
26          println!("tree nodes: {}", bt.size());
27          println!("tree leaves: {}", bt.leaf_size());
28          println!("tree internals: {}", bt.none_leaf_size());
29          println!("tree depth: {}", bt.depth());
30          println!("tree contains '2': {}", bt.contains(&2));
31      }
32
33      fn order() {
34          let mut bt = BinaryTree::new(10usize);
35          bt.insert_left_tree(2usize);
36          bt.insert_right_tree(18usize);
37
38          println!("internal pre-in-post-level order");
39          bt.preorder();
40          bt.inorder();
41          bt.postorder();
42          bt.levelorder();
43
44          let nk = Some(Box::new(bt.clone()));
45          println!("outside pre-in-post-level order");
46          preorder(nk.clone());
47          inorder(nk.clone());
48          postorder(nk.clone());
49          levelorder(nk.clone());
50
51          println!("internal exp: {}", bt.iexp);
52          println!("outside exp: {}", oexp(nk));
53      }
54  }
```

```
root key: 10
root key: 11
left child: Some(
    BinaryTree {
        key: 2,
```

```
            left: None,
            right: None,
        },
    )
    right child: Some(
        BinaryTree {
            key: 18,
            left: None,
            right: None,
        },
    )
    min key: 2
    max key: 18
    tree nodes: 3
    tree leaves: 2
    tree internals: 1
    tree depth: 2
    tree contains '2': true
    internal pre-in-post-level order:
    key: 10
    key: 2
    key: 18
    key: 2
    key: 10
    key: 18
    key: 2
    key: 18
    key: 10
    key: 10
    key: 2
    key: 18
    outside pre-in-post-level order:
    key: 10
    key: 2
    key: 18
    key: 2
    key: 10
    key: 18
    key: 2
    key: 18
    key: 10
    key: 10
    key: 2
    key: 18
    internal exp: ((2)10(18))
    outside exp: ((2)10(18))
```

To simplify the descriptions of the three traversal orders, we can use abbreviations. Pre-order traversal can be abbreviated as ”rt-l-r” to indicate that we first visit the root, then the left subtree, and finally the right subtree. Combining the three traversal orders, we have ”rt-l-r” for pre-order traversal, ”l-rt-r” for in-order traversal, and ”l-r-rt” for post-order traversal. It’s worth noting that there can also be a ”rt-r-l” traversal order, but this is simply the mirror image of pre-order traversal. Since left and right are relative,

”l-rt-r” and ”r-rt-l” can be regarded as mirror images of each other. Similarly, ”r-l-rt” is the mirror image of post-order traversal, and ”r-rt-l” is the mirror image of in-order traversal. The table below summarizes the traversal methods for easy reference:

Table 8.1: Pre-In-Post order and their mirror order method

No	tranversal method	order	mirror-order	mirror tranversal method
1	pre-order	rt-l-r	rt-r-l	mirror-pre-order
2	in-order	l-rt-r	r-rt-l	mirror-in-order
3	post-order	l-r-rt	r-l-rt	mirror-post-order
4	mirror-pre-order	rt-r-l	rt-l-r	pre-order
5	mirror-in-order	r-rt-l	l-rt-r	in-order
6	mirror-post-order	r-l-rt	l-r-rt	post-order

8.3 Binary Heap

In the previous sections, we learned about the queue, a first-in, first-out linear data structure. Another variation of the queue is the priority queue, which dequeues items based on priority rather than their order of addition. In this type of queue, the highest-priority item is dequeued first, regardless of when it was added. When a new item is added to a priority queue, it will move to the front if its priority is high enough. This movement is a way of sorting based on a certain metric, to ensure that high-priority items are at the front.

The priority queue is useful for tasks that involve priorities, such as scheduling processes in an operating system. In this scenario, a priority queue is used to obtain the priority of each process and sort them accordingly. For example, if you are listening to music and browsing news on your phone, and receive an incoming call, the system will give the call the highest priority and interrupt both the news and music to display the incoming call interface. This is an example of managing processes using a priority queue, and assigning a very high priority to the incoming call.

To implement a priority queue, we need to sort it based on some rule to place high-priority items at the front. However, inserting items into the queue has a complexity of $O(n)$, and sorting the queue has a complexity of at least $O(nlog(n))$. To achieve faster results, a binary heap can be used to sort the priority queue. A binary heap is essentially a complete binary tree and allows queuing and dequeuing in $O(log(n))$ time, making it ideal for efficient scheduling systems.

Despite being defined as a binary tree, a linear data structure such as an array, slice, or Vec can be used to implement the binary heap. As long as the operations follow the definition of a binary heap, a linear data structure can implement its functionality and be used as a non-linear data structure. Additionally, there are two common forms of binary heaps: the min heap and the max heap, depending on whether the smallest or largest data item is at the top. This means that there are also two forms of priority queues.

8.3.1 The Binary Heap Abstract Data Type

A min-heap can be used as a priority queue, and its abstract data type includes the following methods:
- new() creates a new binary heap with no parameters and returns an empty heap.
- push(k) adds a new item with parameter value k to the heap and returns nothing.
- pop() returns and removes the minimum item from the heap and modifies the heap.
- min() returns the minimum item in the heap without modifying the heap.
- size() returns the number of items in the heap as a numerical value.
- is_empty() returns a Boolean value indicating whether the heap is empty or not.
- build(arr) constructs a new heap from an array or vector of data, with arr as the parameter.

If a binary heap 'h' has already been created as a priority queue, the table below shows the results of the heap operations performed, with the top item of the heap on the right. The priority value of an item is its value, with smaller values being higher priority and therefore appearing on the right.

Table 8.2: Operations on binary heap

opertions	heap value	return value	operations	heap value	return value
h.is_empty()	[]	true	h.is_empty()	[8,6,3,2,1]	false
h.push(3)	[3]		h.pop()	[8,6,3,2]	1
h.push(8)	[8,3]		h.min()	[8,6,3,2]	2
h.min()	[8,3]	3	h.pop()	[8,6,3]	2
h.push(6)	[8,6,3]		h.pop()	[8,6]	3
h.size()	[8,6,3]	3	h.build([5,4])	[8,6,5,4]	
h.push(2)	[8,6,3,2]		h.build([1])	[8,6,5,4,1]	
h.push(1)	[8,6,3,2,1]		h.min()	[8,6,5,4,1]	1

8.3.2 Implementing a Binay Heap in Rust

To ensure efficient performance of a binary heap, it is crucial to leverage its logarithmic property. For a binary heap stored in a linear data structure, keeping it balanced is key to achieving logarithmic performance. A balanced binary heap has approximately equal numbers of nodes in its left and right subtrees, and it strives to fill each node's left and right child nodes, with at most one node having a non-full set of children in the worst case.

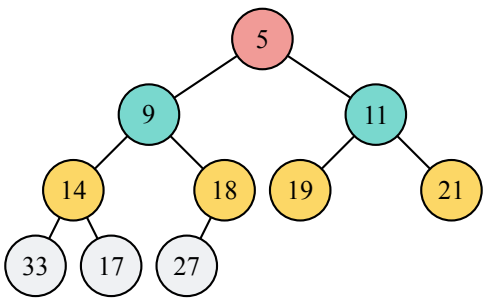


Figure 8.2: Binary Heap

To illustrate, let's consider a heap [0,5,9,11,14,18,19,21,33,17,27] stored using a Vec, and the corresponding tree structure displayed above. As the parent and child nodes are stored in a linear data structure, their relationship is straightforward to compute. If a node is at index p, then its left child node is at 2p, and its right child node is at 2p + 1. Here, p starts from index 1, and index 0 is not used for data, so it is set to 0 as a placeholder.

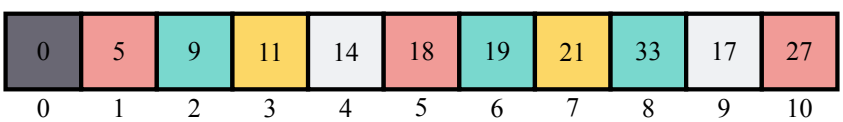


Figure 8.3: Heap in Vec

For instance, the index of 5 is p = 1, and its left child node is at index $2 * p = 2$, where the value is 9. Therefore, 9 is the left child node of 5 in the tree structure. Similarly, the parent node of any child node is located at index p/2. For example, if p = 2 for 9, then its parent node is at index 2/2=1, and if p = 3

for the right child node 11, then its parent node is at index 3/2 = 1 (the division result is rounded down). Hence, computing the parent node of any child node only requires the expression p/2, while child nodes are computed using 2p and 2p+1. We previously defined macros for computing parent and child node indices, and we will continue to use them for calculation purposes.

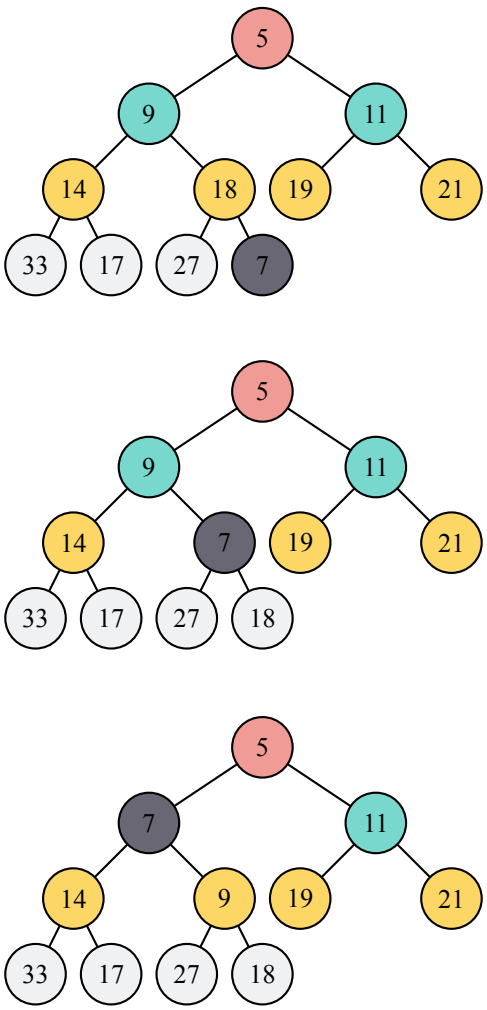
```
// binary_heap.rs

// calculate parent node index
macro_rules! parent {
    ($child:ident) => {
        $child >> 1
    };
}

// calculate left child node index
macro_rules! left_child {
    ($parent:ident) => {
        $parent << 1
    };
}

// calculate right child node index
macro_rules! right_child {
    ($parent:ident) => {
        ($parent << 1) + 1
    };
}
```

To begin with, the binary heap is defined as a data structure that includes a field representing the size of the heap. The size field does not include the first data item 0, which is considered a placeholder. The data saved in the heap is assumed to be i32. When initializing the heap, data is present at index 0, but the size is set to 0.

```
// binary_heap.rs

// Binary heap definision
// Implement Debug and Clone Trait
#[derive(Debug, Clone)]
struct BinaryHeap {
    size: usize,    // data count
    data: Vec<i32>, // data container
}

impl BinaryHeap {
    fn new() -> Self {
        BinaryHeap {
            size: 0,
            data: vec![0] // first 0 not count in total
        }
    }

    fn size(&self)  -> usize {
        self.size
```

```
21      }
22
23      fn is_empty(&self) -> bool {
24          0 == self.size
25      }
26
27      // Get the minimum data in the heap
28      fn min(&self) -> Option<i32> {
29          if 0 == self.size {
30              None
31          } else {
32              // Some(self.data[1].clone());
33              // clone used for genric type
34              Some(self.data[1])
35          }
36      }
37  }
```

When adding data to the heap, adding it to the end of the heap will disrupt the balance, so the data needs to be moved up to maintain balance.



As each data item is added, the size of the heap is increased, and the data is moved up if necessary to maintain balance.

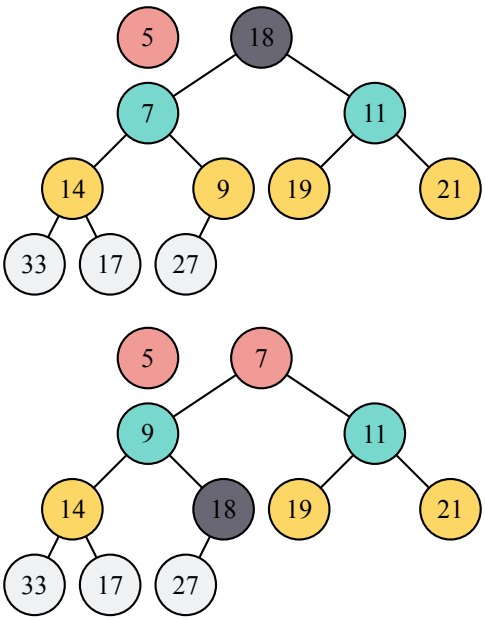
```
// binary_heap.rs

impl BinaryHeap {
    // Add a data to the end and adjust the heap
    fn push(&mut self, val: i32) {
        self.data.push(val);
        self.size += 1;
        self.move_up(self.size);
    }

    // little data move up.
    // c(child, current), p(parent)
    fn move_up(&mut self, mut c: usize) {
        loop {
            // calculate the parent index of current node
            let p = parent!(c);
            if p <= 0 {
                break;
            }

            // If the current node's data is smaller than
            // the parent node's data, swap them
            if self.data[c] < self.data[p] {
                self.data.swap(c, p);
            }

            // The parent node becomes the current node
            c = p;
        }
    }
}
```

To retrieve the minimum value from the heap, three cases need to be considered: when there is no data in the heap, return None; when there is only one data item, pop it directly; when there are multiple data items, swap the top and end data of the heap, adjust the heap, and then return the minimum value at the end. The move_down function is used to move elements down to maintain balance, and the min_child function is used to find the minimum child node.

```
// binary_heap.rs

impl BinaryHeap {
    // pop out the top value
    fn pop(&mut self) -> Option<i32> {
        if 0 == self.size {
            // no data, return None
            None
        } else if 1 == self.size {
            self.size -= 1;

            self.data.pop()
```

```
13          } else {
14              // swap data and then adjust the heap
15              self.data.swap(1, self.size);
16              let val = self.data.pop();
17              self.size -= 1;
18              self.move_down(1);
19
20              val
21          }
22      }
23
24      // bigger data move down
25      fn move_down(&mut self, mut c: usize) {
26          loop {
27              let lc = left_child!(c);
28              if lc > self.size { break; }
29
30              // the index of mininum child node of current node
31              let mc = self.min_child(c);
32              if self.data[c] > self.data[mc] {
33                  self.data.swap(c, mc);
34              }
35
36              // the mininum child node becomes current node
37              c = mc;
38          }
39      }
40
41      // Calculate the index of the minimum child node
42      fn min_child(&self, c: usize) -> usize {
43          let (lc, rc) = (left_child!(c), right_child!(c));
44
45          if rc > self.size {
46              // right child node is out of range,
47              // left child node is the minimum child node
48              lc
49          } else if self.data[lc] < self.data[rc] {
50              // left child node is smaller than right child node
51              lc
52          } else {
53              // right child node is smaller than left child node
54              rc
55          }
56      }
57  }
```

The process of deleting the minimum element from the heap involves taking out the element at the top of the heap and moving the last element to the top. The top element is not the minimum value at this point and does not meet the heap definition, so the heap needs to be rebuilt. To rebuild the heap, the top element needs to be moved down using the move_down function. This function uses a macro to calculate whether to swap with the left or right child node based on the node index, eventually swapping the top element with its minimum child node until the heap is restored.



Finally, data can be added to the heap in batches. For instance, a slice of data [0,5,4,3,1,2] can be added to the heap at once to avoid frequent calls to the push function. The original data in the heap can either be kept unchanged while adding the slice data one by one or deleted before adding the slice data. The latter approach results in a new heap that includes only the data in the slice.

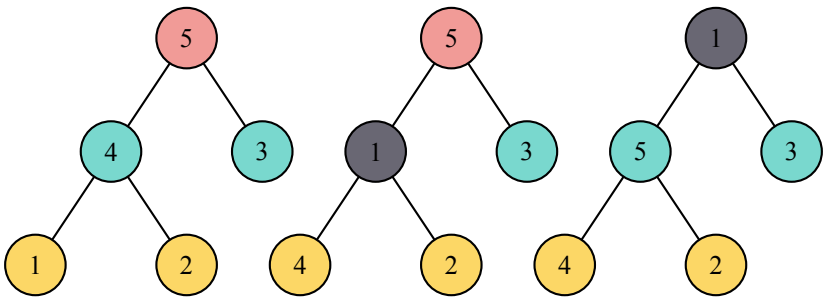


Figure 8.4: delete data and rebuild the heap

To implement the binary min heap, we define the build_new and build_add functions, as shown below.

```
// binary_heap.rs

impl BinaryHeap {
    // build a new heap
    fn build_new(&mut self, arr: &[i32]) {
        // delete all data
        for _i in 0..self.size {
            let _rm = self.data.pop();
        }

        // add new data
        for &val in arr {
```

```
13              self.data.push(val);
14          }
15
16          // change the size
17          self.size = arr.len();
18
19          // adjust the heap to make it a min-heap
20          let size = self.size;
21          let mut p = parent!(size);
22          while p > 0 {
23              self.move_down(p);
24              p -= 1;
25          }
26      }
27
28      // add slice data one by one
29      fn build_add(&mut self, arr: &[i32]) {
30          for &val in arr {
31              self.push(val);
32          }
33      }
34  }
```

With these functions, we have completed the construction of a binary min heap. The entire process should be easy to understand, and based on this, one can write the code for a binary max heap as well. Here's an example of using a binary heap.

```
1   // binary_heap.rs
2
3   fn main() {
4       let mut bh = BinaryHeap::new();
5       let nums = [-1,0,2,3,4];
6       bh.push(10); bh.push(9);
7       bh.push(8); bh.push(7); bh.push(6);
8
9       bh.build_add(&nums);
10      println!("empty: {:?}", bh.is_empty());
11      println!("min: {:?}", bh.min());
12      println!("pop min: {:?}", bh.pop());
13
14      bh.build_new(&nums);
15      println!("size: {:?}", bh.len());
16      println!("pop min: {:?}", bh.pop());
17  }
```

The following are the outputs after execution.

```
empty: false
min: Some(-1)
size: 10
pop min: Some(-1)
size: 5
pop min: Some(-1)
```

8.3.3 Analysis of Binary Heap

The binary heap stores data linearly in a Vec, but its sorting is based on a tree structure. As discussed earlier, the height of a tree is $O(nlog_2(n))$, and heap sorting moves from the bottom layer to the top layer of the tree, with the number of movements being equal to the number of layers. Hence, the time complexity of heap sorting is $O(nlog_2(n))$. Since constructing a heap requires processing all n items, the complexity is $O(n)$. Overall, the time complexity of the binary heap is $O(nlog_2(n)) + O(n) = O(nlog_2(n))$.

8.4 Binary Search Tree

However, linear data structures for trees are not suitable for large amounts of data as copying and moving data become very time-consuming. To address this, this section explores implementing trees using nodes. Specifically, this section focuses on binary trees, which have only two child nodes. To understand binary trees better, we study binary search trees for searching, which store data using key-value pairs, similar to HashMap.

8.4.1 The Binary Search Tree Abstract Data Type

The binary search tree provides the following abstract data types:

• new() creates a new tree, takes no parameters, and returns an empty tree.
• insert(k, v) stores the data (k, v) in the tree, takes the key k and value v, and returns nothing.
• contains(&k) searches the tree for the key k, takes a parameter &k, and returns a boolean.
• get(&k) returns the value v of key k from the tree without deleting it, takes a parameter &k.
• max() returns the maximum key k and its value v in the tree, takes no parameters.
• min() returns the minimum key k and its value v in the tree, takes no parameters.
• len() returns the number of data items in the tree, takes no parameters, and returns an integer of type usize.
• is_empty() tests whether the tree is empty, takes no parameters, and returns a boolean.
• iter() returns the tree in iterator form, takes no parameters, and does not change the tree.
• preorder() performs a pre-order traversal, takes no parameters, and outputs each k-v pairs.
• inorder() performs an in-order traversal, takes no parameters, and outputs each k-v pairs.
• postorder() performs a post-order traversal, takes no parameters, and outputs each k-v pairs.

The table below displays the binary search tree after different operations, represented by tuples for tree nodes and enclosed in square brackets, assuming t is a newly created empty tree.

Table 8.3: Operations on binary search tree

operation	tree value	return value
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.contains(2)	[(1,‘a’),(2,‘b’)]	true
t.insert(2,‘c’)	[(1,‘a’),(2,‘c’)]	
t.insert(3,‘d’)	[(1,‘a’),(2,‘c’),(3,‘d’)]	
t.contains(4)	[(1,‘a’),(2,‘c’),(3,‘d’)]	false
t.get(&3)	[(1,‘a’),(2,‘c’),(3,‘d’)]	Some(‘d’)

8.4.2 Implementing a Binary Search Tree in Rust

In contrast to heaps, where the left and right child nodes are not compared to their value, in a binary search tree, the key of the left child node is smaller than the key of the parent node, and the key of the right child node is greater than the key of the parent node. This rule of left < parent < right is applied recursively to all subtrees.

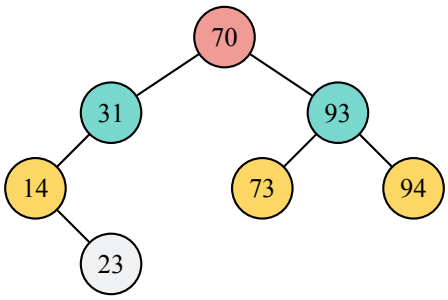


Figure 8.5: Binary search tree

In the example above, 70 is the root node, 31 is smaller and becomes the left node, and 93 is larger and becomes the right node. Then 14 is inserted, which is smaller than 70 and descends to 31, and as it is smaller than 31, it becomes the left node of 31. Similar steps are taken to insert other data, and finally, a binary search tree is formed. The inorder traversal of the tree is [14, 23, 31, 70, 73, 93, 94], which is sorted from small to large, so the binary search tree can also be used to sort data. Using inorder traversal, we obtain the ascending order sorting result, and using the mirrored inorder traversal, i.e., the ”right-root-left” order traversal, we obtain the descending order sorting result.

To implement the binary search tree, we define it as a struct BST (BinarySearchTree), which includes key values and links to left and right child nodes. Following the definition of the abstract data type, the implementation of the binary search tree is shown below.

```
1  // bst.rs
2  use std::cmp::{max, Ordering::*};
3  use std::fmt::Debug;
4
5  // Binary search tree node link
6  type Link<T,U> = Option<Box<BST<T,U>>>;
7
8  // Definition of binary search tree
9  #[derive(Debug,Clone)]
10 struct BST<T,U> {
11     key: Option<T>,
12     val: Option<U>,
13     left: Link<T,U>,
14     right: Link<T,U>,
15 }
16
17 impl<T,U> BST<T,U>
18     where T: Copy + Ord + Debug,
19           U: Copy + Debug
20 {
21     fn new() -> Self {
22         Self {
23             key: None,
```

```
24              val: None,
25              left: None,
26              right: None,
27          }
28      }
29
30      fn is_empty(&self) -> bool {
31          self.key.is_none()
32      }
33
34      fn size(&self) -> usize {
35          self.calc_size(0)
36      }
37
38      // Recursively count the number of nodes
39      fn calc_size(&self, mut size: usize) -> usize {
40          if self.key.is_none() { return size; }
41
42          // Add current node count to total node count 'size'
43          size += 1;
44          // Count left and right child nodes
45          if !self.left.is_none() {
46              size = self.left.as_ref().unwrap().calc_size(size);
47          }
48          if !self.right.is_none() {
49              size = self.right
50                      .as_ref().unwrap().calc_size(size);
51          }
52
53          size
54      }
55
56      // Count leaf nodes
57      fn leaf_size(&self) -> usize {
58          // If both left and right are empty,
59          // current node is a leaf node, return 1
60          if self.left.is_none() && self.right.is_none() {
61              return 1;
62          }
63
64          // Count leaf nodes in left and right subtree
65          let left_leaf = match &self.left {
66              Some(left) => left.leaf_size(),
67              None => 0,
68          };
69          let right_leaf = match &self.right {
70              Some(right) => right.leaf_size(),
71              None => 0,
72          };
73
74          // total sum of leaf nodes
75          left_leaf + right_leaf
```

```
76      }
77
78      // Count non-leaf nodes
79      fn none_leaf_size(&self) -> usize {
80          self.size() - self.leaf_size()
81      }
82
83      // Calculate tree depth
84      fn depth(&self) -> usize {
85          let mut left_depth = 1;
86          if let Some(left) = &self.left {
87              left_depth += left.depth();
88          }
89
90          let mut right_depth = 1;
91          if let Some(right) = &self.right {
92              right_depth += right.depth();
93          }
94
95          max(left_depth, right_depth)
96      }
97
98      fn insert(&mut self, key: T, val: U) {
99          // If no data, insert directly
100         if self.key.is_none() {
101             self.key = Some(key);
102             self.val = Some(val);
103         } else {
104             match &self.key {
105                 Some(k) => {
106                     // If key exists, update val
107                     if key == *k {
108                         self.val = Some(val);
109                         return;
110                     }
111
112                     // If no same key found,
113                     // Find the subtree to insert new node
114                     let child = if key < *k {
115                         &mut self.left
116                     } else {
117                         &mut self.right
118                     };
119
120                     // Recursively go down the tree
121                     // until insertion
122                     match child {
123                         Some(ref mut node) => {
124                             node.insert(key, val);
125                         },
126                         None => {
127                             let mut node = BST::new();
```

```
128                             node.insert(key, val);
129                             *child = Some(Box::new(node));
130                         },
131                     }
132                 },
133                 None => (),
134             }
135         }
136     }
137
138     // find the key in tree
139     fn contains(&self, key: &T) -> bool {
140         match &self.key {
141             None => false,
142             Some(k) => {
143                 // Compare key value and determine
144                 // whether to continue searching recursively
145                 match k.cmp(key) {
146                     Equal => true, // find it
147                     Greater => { // search in left subtree
148                         match &self.left {
149                             Some(node) => node.contains(key),
150                             None => false,
151                         }
152                     },
153                     Less => { // search in right subtree
154                         match &self.right {
155                             Some(node) => node.contains(key),
156                             None => false,
157                         }
158                     },
159                 }
160             },
161         }
162     }
163
164     // get the min/max node value
165     fn min(&self) -> (Option<&T>, Option<&U>) {
166         // miniumun value is on the left subtree
167         match &self.left {
168             Some(node) => node.min(),
169             None => match &self.key {
170                 Some(key) => (Some(&key), self.val.as_ref()),
171                 None => (None, None),
172             },
173         }
174     }
175
176     fn max(&self) -> (Option<&T>, Option<&U>) {
177         // maxiumun value is on the right subtree
178         match &self.right {
179             Some(node) => node.max(),
```

```
180                 None => match &self.key {
181                     Some(key) => (Some(&key), self.val.as_ref()),
182                     None => (None, None),
183                 },
184             }
185         }
186 
187         // get subtree
188         fn get_left(&self) -> Link<T,U> {
189             self.left.clone()
190         }
191 
192         fn get_right(&self) -> Link<T,U> {
193             self.right.clone()
194         }
195 
196         // get a value reference with parameter key
197         fn get(&self, key: &T) -> Option<&U> {
198             match &self.key {
199                 None => None,
200                 Some(k) => {
201                     match k.cmp(key) {
202                         Equal => self.val.as_ref(),
203                         Greater => {
204                             match &self.left {
205                                 None => None,
206                                 Some(node) => node.get(key),
207                             }
208                         },
209                         Less => {
210                             match &self.right {
211                                 None => None,
212                                 Some(node) => node.get(key),
213                             }
214                         },
215                     }
216                 },
217             }
218         }
219 }
```

Here are the implementation of preorder, inorder, postorder, and level-order traversals.

```
1  // bst.rs
2 
3  impl<T,U> BST<T,U>
4      where T: Copy + Ord + Debug,
5            U: Copy + Debug
6  {
7      // Internal implementation
8      fn preorder(&self) {
9          println!("key: {:?}, val: {:?}",self.key, self.val);
10         match &self.left {
```

```
11              Some(node) => node.preorder(),
12              None => (),
13          }
14          match &self.right {
15              Some(node) => node.preorder(),
16              None => (),
17          }
18      }
19
20      fn inorder(&self) {
21          match &self.left {
22              Some(node) => node.inorder(),
23              None => (),
24          }
25          println!("key: {:?}, val: {:?}",self.key, self.val);
26          match &self.right {
27              Some(node) => node.inorder(),
28              None => (),
29          }
30      }
31
32      fn postorder(&self) {
33          match &self.left {
34              Some(node) => node.postorder(),
35              None => (),
36          }
37          match &self.right {
38              Some(node) => node.postorder(),
39              None => (),
40          }
41          println!("key: {:?}, val: {:?}",self.key, self.val);
42      }
43
44      fn levelorder(&self) {
45          let size = self.size();
46          let mut q = Queue::new(size);
47
48          let _r = q.enqueue(Box::new(self.clone()));
49          while !q.is_empty() {
50              let front = q.dequeue().unwrap();
51              println!("key: {:?}, val: {:?}",
52                      front.key, front.val);
53
54              match front.get_left() {
55                  Some(left) => { let _r = q.enqueue(left); },
56                  None => (),
57              }
58              match front.get_right() {
59                  Some(right) => { let _r = q.enqueue(right); },
60                  None => (),
61              }
62          }
```

```
63      }
64  }
65
66  // External implementation
67  fn preorder<T, U>(bst: Link<T,U>)
68  where T: Copy + Ord + Debug,
69        U: Copy + Debug
70  {
71      if !bst.is_none() {
72          println!("key: {:?}, val: {:?}",
73                    bst.as_ref().unwrap().key.unwrap(),
74                    bst.as_ref().unwrap().val.unwrap());
75          preorder(bst.as_ref().unwrap().get_left());
76          preorder(bst.as_ref().unwrap().get_right());
77      }
78  }
79
80  fn inorder<T, U>(bst: Link<T,U>)
81  where T: Copy + Ord + Debug,
82        U: Copy + Debug
83  {
84      if !bst.is_none() {
85          inorder(bst.as_ref().unwrap().get_left());
86          println!("key: {:?}, val: {:?}",
87                    bst.as_ref().unwrap().key.unwrap(),
88                    bst.as_ref().unwrap().val.unwrap());
89          inorder(bst.as_ref().unwrap().get_right());
90      }
91  }
92
93  fn postorder<T, U>(bst: Link<T,U>)
94  where T: Copy + Ord + Debug,
95        U: Copy + Debug
96  {
97      if !bst.is_none() {
98          postorder(bst.as_ref().unwrap().get_left());
99          postorder(bst.as_ref().unwrap().get_right());
100         println!("key: {:?}, val: {:?}",
101                   bst.as_ref().unwrap().key.unwrap(),
102                   bst.as_ref().unwrap().val.unwrap());
103     }
104 }
105
106 fn levelorder<T, U>(bst: Link<T,U>)
107 where T: Copy + Ord + Debug,
108       U: Copy + Debug
109 {
110     if bst.is_none() { return; }
111
112     let size = bst.as_ref().unwrap().size();
113     let mut q = Queue::new(size);
114     let _r = q.enqueue(bst.as_ref().unwrap().clone());
```

```
115     while !q.is_empty() {
116         let front = q.dequeue().unwrap();
117         println!("key: {:?}, val: {:?}", front.key, front.val);
118 
119         match front.get_left() {
120             Some(left) => { let _r = q.enqueue(left); },
121             None => {},
122         }
123 
124         match front.get_right() {
125             Some(right) => { let _r = q.enqueue(right); },
126             None => {},
127         }
128     }
129 }
```

The following are the outputs after execution of binary search tree.

```
1  // bst.rs
2 
3  fn main() {
4      basic();
5      order();
6 
7      fn basic() {
8          let mut bst = BST::<i32, char>::new();
9          bst.insert(8, 'e'); bst.insert(6,'c');
10         bst.insert(7, 'd'); bst.insert(5,'b');
11         bst.insert(10,'g'); bst.insert(9,'f');
12         bst.insert(11,'h'); bst.insert(4,'a');
13 
14         println!("bst is empty: {}", bst.is_empty());
15         println!("bst size: {}", bst.size());
16         println!("bst leaves: {}", bst.leaf_size());
17         println!("bst internals: {}", bst.none_leaf_size());
18         println!("bst depth: {}", bst.depth());
19 
20         let min_kv = bst.min();
21         let max_kv = bst.max();
22         println!("min key-val: {:?}-{:?}", min_kv.0, min_kv.1);
23         println!("max key-val: {:?}-{:?}", max_kv.0, max_kv.1);
24         println!("bst contains 5: {}", bst.contains(&5));
25         println!("key: 5, val: {:?}", bst.get(&5).unwrap());
26     }
27 
28     fn order() {
29         let mut bst = BST::<i32, char>::new();
30         bst.insert(8, 'e'); bst.insert(6,'c');
31         bst.insert(7, 'd'); bst.insert(5,'b');
32         bst.insert(10,'g'); bst.insert(9,'f');
33         bst.insert(11,'h'); bst.insert(4,'a');
34 
35         println!("internal inorder, preorder, postorder: ");
```

```
36          bst.inorder();
37          bst.preorder();
38          bst.postorder();
39          bst.levelorder();
40          println!("outside inorder, preorder, postorder: ");
41          let nk = Some(Box::new(bst.clone()));
42          inorder(nk.clone());
43          preorder(nk.clone());
44          postorder(nk.clone());
45          levelorder(nk.clone());
46      }
47  }
```

The following are outputs after execution.

```
bst is empty: false
bst size: 8
bst leaves: 4
bst internals: 4
bst depth: 4
min key: Some(4), min val: Some('a')
max key: Some(11), max val: Some('h')
bst contains 5: true
key: 5, val: 'b'
internal inorder, preorder, postorder:
key: 4, val: 'a'
key: 5, val: 'b'
key: 6, val: 'c'
key: 7, val: 'd'
key: 8, val: 'e'
key: 9, val: 'f'
key: 10, val: 'g'
key: 11, val: 'h'
key: 8, val: 'e'
key: 6, val: 'c'
key: 5, val: 'b'
key: 4, val: 'a'
key: 7, val: 'd'
key: 10, val: 'g'
key: 9, val: 'f'
key: 11, val: 'h'
key: 4, val: 'a'
key: 5, val: 'b'
key: 7, val: 'd'
key: 6, val: 'c'
key: 9, val: 'f'
key: 11, val: 'h'
key: 10, val: 'g'
key: 8, val: 'e'
key: 8, val: 'e'
key: 6, val: 'c'
key: 10, val: 'g'
key: 5, val: 'b'
```

```
key: 7, val: 'd'
key: 9, val: 'f'
key: 11, val: 'h'
key: 4, val: 'a'
outside inorder, preorder, postorder:
key: 4, val: 'a'
key: 5, val: 'b'
key: 6, val: 'c'
key: 7, val: 'd'
key: 8, val: 'e'
key: 9, val: 'f'
key: 10, val: 'g'
key: 11, val: 'h'
key: 8, val: 'e'
key: 6, val: 'c'
key: 5, val: 'b'
key: 4, val: 'a'
key: 7, val: 'd'
key: 10, val: 'g'
key: 9, val: 'f'
key: 11, val: 'h'
key: 4, val: 'a'
key: 5, val: 'b'
key: 7, val: 'd'
key: 6, val: 'c'
key: 9, val: 'f'
key: 11, val: 'h'
key: 10, val: 'g'
key: 8, val: 'e'
key: 8, val: 'e'
key: 6, val: 'c'
key: 10, val: 'g'
key: 5, val: 'b'
key: 7, val: 'd'
key: 9, val: 'f'
key: 11, val: 'h'
key: 4, val: 'a'
```

When insert 76 into the tree, its search path is shown by the black nodes.

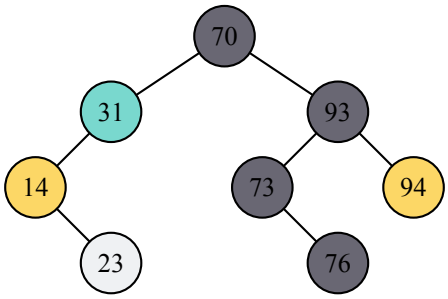


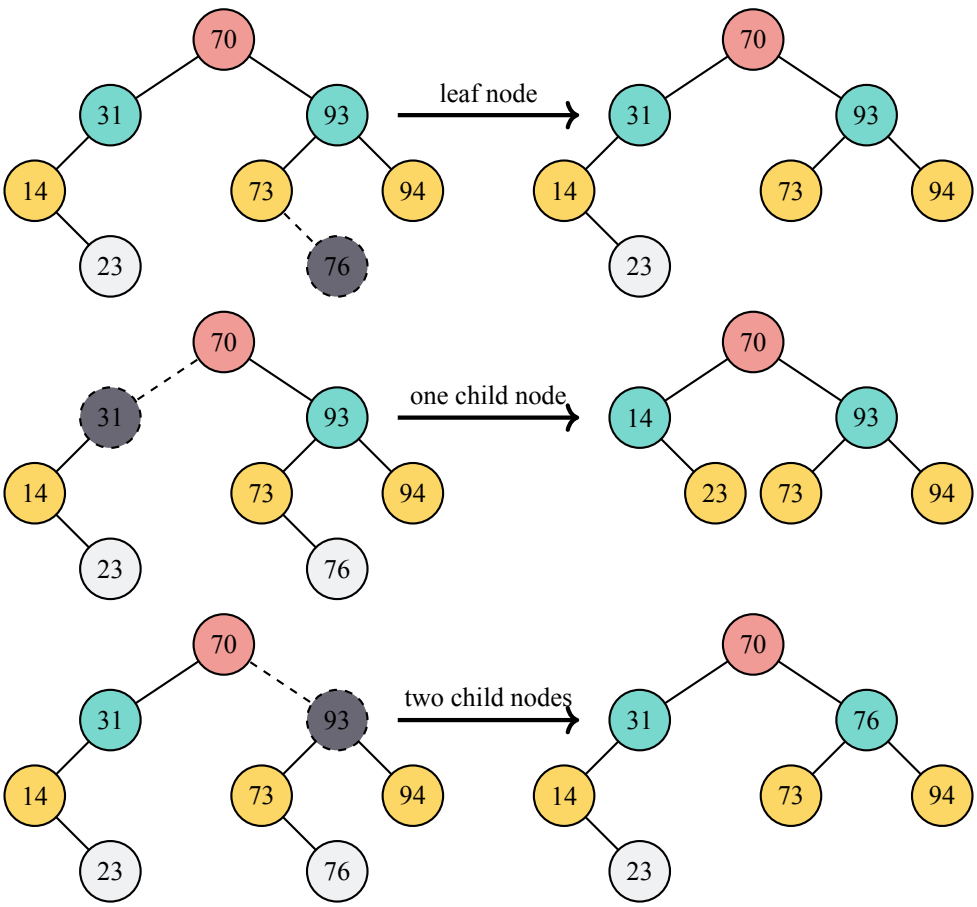
Figure 8.6: Insertion into a BST

Deleting a node in a binary search tree is a complex operation. The first step is to find the node to be deleted, which may not exist in the tree. Once the node is found, we need to determine if it has children, which can be one of three cases: no children, one child, or two children.

Table 8.4: Key deletion cases

No	Node	node k	delete mothod
1	No node	None	Directly return
2	One root node	None	Directly delete
3	Multiple nodes	None	Directly delete
4	Multiple nodes	One child	Replace k with child node
5	Multiple nodes	Two children	Replace k with the successor node

If the node is a leaf node with no children, we can simply remove its reference from its parent node. If it has one child, we modify the parent node's reference to point directly to the child node. The most challenging case is when the node has two children. In this case, we find the minimum node in the right subtree, called the successor node, and replace the node to be deleted with the successor node. The successor node may also have children, and we must adjust their relationships accordingly. The specific situations are shown in the following figure, where the dashed box represents the node k to be deleted, and the binary tree obtained after deleting the node is shown on the right.



Deleting a leaf node is the simplest case, while deleting an internal node with one child is straightforward. The most difficult case is when the node has two children, as this requires adjusting the relationships of multiple nodes.

8.4.3 Analysis of Binary Search Tree

Now that the binary search tree has been fully implemented, let's analyze the time complexity of each method. Traversals and len() both have a time complexity of $O(n)$ since they need to process all n elements. The contains() method compares data with the left and right child nodes and selects a branch based on the result. It walks the longest path from the root to the leaf node at most. The height of a binary tree is related to the total number of nodes.

$$\begin{aligned} 2^0 + 2^1 + 2^i ... + 2^h &= n \\ h &= log_2(n) \end{aligned} \tag{8.1}$$

Using the binary tree properties, we can approximate the maximum path length to be $h = log_2(n)$. Thus, the time complexity of contains() is $O(log_2(n))$. Insertion, deletion, and modification are based on search because the element must be located before processing can continue. Their time complexity depends only on search and can be completed in constant time. Therefore, the performance of contains(), insert(), and get() methods are all $O(log_2(n))$. The height h of the binary tree is the limiting factor of their performance. If the inserted data is always in a sorted state, the binary tree may degenerate into a linear linked list, and the performance of contains(), insert(), and remove() methods will be $O(n)$. The figure below illustrates this scenario.

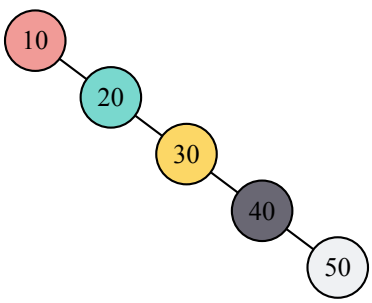


Figure 8.7: bst in a linear linked list

In this section, we did not implement the remove() function because it is not part of the abstract data type definition, and binary search trees are primarily used for data insertion and search, not deletion. However, interested readers can try to implement the remove() function as an exercise.

To improve performance by reducing the tree height, we can convert the binary tree into a multi-way tree such as B-trees and B+ trees. These trees have many child nodes, resulting in a short tree height and fast queries. They are commonly used in implementing databases and file systems. For instance, MySQL database uses B+ trees to store data, with nodes of 16K memory pages. If each data is 1k in size, one node can store 16 pieces of data. When used to store an index with a bigint key, 8 bytes are used, and the index is 6 bytes, for a total of 14 bytes. Thus, one node can store approximately $16 * 1024/14 = 1170$ indexes. With a height of 3, a B+ tree can store around $1170 * 1170 * 16 = 21902400$ indexes, which can hold about 20 million pieces of data. Retrieving data only requires at most two queries, which explains why database queries are fast. Readers interested in this topic can read MySQL-related books to learn more.

8.5 Balanced Binary Search Tree

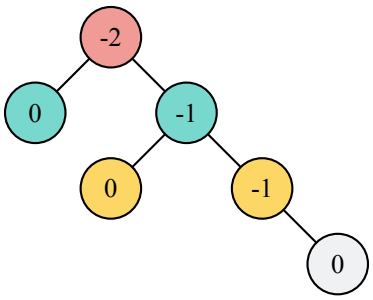
In the previous section, we constructed a binary search tree and learned that its performance can degrade to $O(n)$ in certain cases, such as when the tree is unbalanced with one side having significantly more nodes than the other. This inefficiency can lead to poor subsequent operations. To ensure efficient data processing, building a balanced binary tree is essential. In this section, we will discuss an AVL tree,

a balanced binary search tree that can automatically maintain balance. It is named after its inventors: G.M. Adelson-Velskii and E.M.Land.

The AVL tree is also an ordinary binary search tree, but with a difference in the way the tree operations are performed. AVL trees use a balance factor to determine if the tree is balanced during operations. The balance factor is the difference in height between the left and right subtrees of a node, and it is defined as:

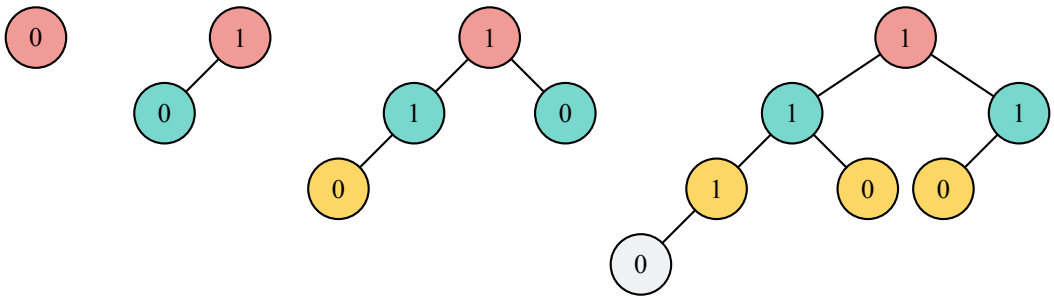
$$balanceFactor = height(leftSubTree) - height(rightSubTree) \tag{8.2}$$

Given this balance factor definition, if the balance factor is greater than zero, then the left subtree is heavy. If it is less than zero, then the right subtree is heavy, and if it is zero, then the tree is balanced. To implement an AVL tree efficiently, balance factors of -1, 0, and 1 are all considered balanced because the difference in height between the left and right subtrees is only 1 in these cases, which is essentially balanced. Once a node's balance factor is outside this range, such as 2 or -2, the tree needs to be rotated to maintain balance. The following figure illustrates the case of unbalanced left and right subtrees, and each node's balance factor is indicated by its value.



8.5.1 AVL Tree

To ensure a balanced binary tree, it is crucial to fulfill the balance factor condition, which allows for only three possible cases: left-heavy, balanced, or right-heavy. It is worth noting that a left-heavy or right-heavy tree that still meets the condition of having a balance factor of -1, 0, or 1 can also be considered balanced. The figure below depicts the most imbalanced left-heavy tree that meets the balance factor condition for trees of height 0, 1, 2, and 3.



By analyzing the total number of nodes in the tree, we can derive a formula for the number of nodes in a tree of height h, as follows: For a tree of height 0, there is only one node; for a tree of height 1, there are 2 nodes; for a tree of height 2, there are 4 nodes; for a tree of height 3, there are 7 nodes, and so on.

$$N_h = 1 + N_{h-1} + N_{h-2} \tag{8.3}$$

Remarkably, this formula resembles the Fibonacci sequence. With the number of nodes in the tree, we can obtain the height formula for an AVL tree using the Fibonacci equation, where the ith Fibonacci

number is given by the following equation:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \end{aligned} \tag{8.4}$$

To calculate the number of nodes in an AVL tree, we can use the following formula where $F_0 = 1$.

$$N_h = F_{h+2} - 1 \tag{8.5}$$

As i increases, the ratio F_i/F_{i-1} approaches the golden ratio $\Phi = (1 + \sqrt{5})/2$, so Φ can be used to represent F_i, which can be calculated as $F_i = \frac{\Phi^i}{5}$.

$$N_h = \frac{\Phi^h}{\sqrt{5}} + 1 \tag{8.6}$$

Based on these formulas, we can derive a height formula for AVL trees.

$$\begin{aligned} log(N_h - 1) &= log(\frac{\Phi^h}{\sqrt{5}}) \\ log(N_h - 1) &= hlog\Phi - \frac{1}{2}log5 \\ h &= \frac{log(N_h - 1) + \frac{1}{2}log5}{log\Phi} \\ h &= 1.44log(N_h) \end{aligned} \tag{8.7}$$

In this formula, h is the height of the AVL tree, and N_h is the number of nodes. This implies that the height of an AVL tree is at most 1.44 times the logarithm of the number of nodes, and the search complexity is $O(logN)$, which is highly efficient.

8.5.2 Implementing a AVL Tree in Rust

To insert a new node into an AVL tree, we first add it as a leaf node, which automatically gives it a balance factor of 0. However, the balance factor of the parent node will need to be updated, depending on whether the new node is a left or right child. If it is a right child, the parent node's balance factor will be reduced by 1, and if it is a left child, the balance factor will be increased by 1. This update is propagated up the tree recursively, until we reach the root or a node whose balance factor becomes 0.

In our implementation, we represent the AVL tree as an enumeration with two cases: Null, which represents an empty tree, and Tree, which represents a node in the tree. The AvlNode type is used to store the data, left and right subtrees, and balance factor of each node.

```
1  // avl.rs
2
3  // Use a enum to define the Avl tree
4  #[derive(Clone, Debug, PartialEq)]
5  enum AvlTree<T> {
6      Null,
7      Tree(Box<AvlNode<T>>),
8  }
9
10 // Definish of Avl tree node
11 #[derive(Debug)]
12 struct AvlNode<T> {
```

```
    key: T,
    left: AvlTree<T>,  // left subtree
    right: AvlTree<T>, // right subtree
    bfactor: i8,       // balance factor
}
```

To implement the AVL tree, we need to add two functions: insert and rebalance. To compare node data, we also introduce a comparison property called Ording. Additionally, we need the replace and max functions to update values and calculate the tree height.

```
// avl.rs

use std::cmp::{max, Ordering::*};
use std::fmt::Debug;
use std::mem::replace;
use AvlTree::*;

impl<T> AvlTree<T> where T : Clone + Ord + Debug {
    // new tree is Empty
    fn new() -> AvlTree<T> {
        Null
    }

    fn insert(&mut self, key: T) -> (bool, bool) {
        let ret = match self {
            Null => {
                // If there is no node, insert directly
                let node = AvlNode {
                    key: key,
                    left: Null,
                    right: Null,
                    bfactor: 0,
                };
                *self = Tree(Box::new(node));

                (true, true)
            },
            Tree(ref mut node) => match node.key.cmp(&key) {
                // Compare the value of the node and determine
                // which side to insert from
                // inserted: whether insertion is performed
                // deepened: whether the depth is increased

                // If they are equal, no insertion is needed
                Equal => (false, false),
                // node value is smaller, insert to the right
                Less => {
                    let (inserted, deepened)
                        = node.right.insert(key);
                    if deepened {
                        let ret = match node.bfactor {
                            -1 => (inserted, false),
                             0 => (inserted, true),
```

```
                                1 => (inserted, false),
                                _ => unreachable!(),
                            };
                            node.bfactor += 1;

                            ret
                        } else {
                            (inserted, deepened)
                        }
                    },
                    // node value is greater, insert to the left
                    Greater => {
                        let (inserted, deepened)
                            = node.left.insert(key);
                        if deepened {
                            let ret = match node.bfactor {
                                -1 => (inserted, false),
                                 0 => (inserted, true),
                                 1 => (inserted, false),
                                 _ => unreachable!(),
                            };
                            node.bfactor -= 1;

                            ret
                        } else {
                            (inserted, deepened)
                        }
                    },
                },
            };
            self.rebalance();

            ret
        }

        // Adjust the balance factors of each node
        fn rebalance(&mut self) {
            match self {
                // If there is no data, no adjustment is needed
                Null => (),
                Tree(_) => match self.node().bfactor {
                    // If the right subtree is heavy
                    -2 => {
                        let lbf = self.node()
                                      .left
                                      .node()
                                      .bfactor;

                        if lbf == -1 || lbf == 0 {
                            let (a, b) = if lbf == -1 {
                                (0, 0)
                            } else {
```

```
                            (-1,1)
                        };

                        // Rotate and update the balance factor
                        self.rotate_right();
                        self.node().right.node().bfactor = a;
                        self.node().bfactor = b;
                    } else if lbf == 1 {
                        let (a, b) = match self.node()
                                               .left.node()
                                               .right.node()
                                               .bfactor
                        {
                            -1 => (1, 0),
                             0 => (0, 0),
                             1 => (0,-1),
                             _ => unreachable!(),
                        };

                        // First rotate left, then rotate right
                        // finally update the balance factor
                        self.node().left.rotate_left();
                        self.rotate_right();
                        self.node().right.node().bfactor = a;
                        self.node().left.node().bfactor = b;
                        self.node().bfactor = 0;
                    } else {
                        unreachable!()
                    }
                },
                // If the left subtree is heavy
                2 => {
                     let rbf=self.node().right.node().bfactor;
                     if rbf == 1 || rbf == 0 {
                         let (a, b) = if rbf == 1 {
                            (0, 0)
                         } else {
                            (1,-1)
                         };

                         self.rotate_left();
                         self.node().left.node().bfactor = a;
                         self.node().bfactor = b;
                     } else if rbf == -1 {
                        let (a, b) = match self.node()
                                               .right.node()
                                               .left.node()
                                               .bfactor
                        {
                             1 => (-1,0),
                             0 => (0, 0),
                            -1 => (0, 1),
```

```
148                                 _ => unreachable!(),
149                             };
150
151                             // First rotate right, then rotate left
152                             // finally update the balance factor
153                             self.node().right.rotate_right();
154                             self.rotate_left();
155                             self.node().left.node().bfactor = a;
156                             self.node().right.node().bfactor = b;
157                             self.node().bfactor = 0;
158                         } else {
159                             unreachable!()
160                         }
161                     },
162                     _ => (),
163                 },
164             }
165         }
166     }
```

The insert function updates the balance factor recursively, while the rebalance function restores balance to the AVL tree. Effective rebalancing is crucial for maintaining proper AVL tree functionality without compromising performance. Rebalancing involves rotating the tree once or multiple times, which can be done through left or right rotations.

To perform a left rotation, we can follow the steps shown in Figure (8.8):

(1) Promote the right child (B) as the new root of the subtree.

(2) Move the old root (A) to the left child of the new root.

(3) If the new root (B) already has a left child, make it the right child of the new left child (A).

Similarly, to perform a right rotation, we can follow these steps:

(1) Promote the left child (B) as the new root of the subtree.

(2) Move the old root (A) to the right child of the new root.

(3) If the new root (B) already has a right child, make it the left child of the new right child (A).

The two trees in the figure are unbalanced, and we can rebalance them into the tree on the right by performing left and right rotations with A as the root.

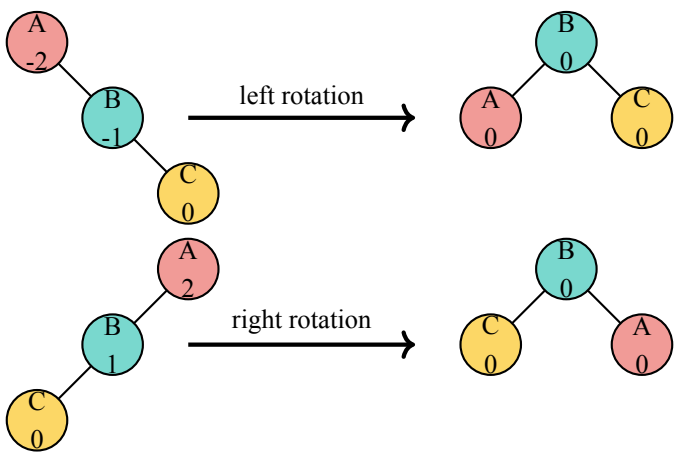
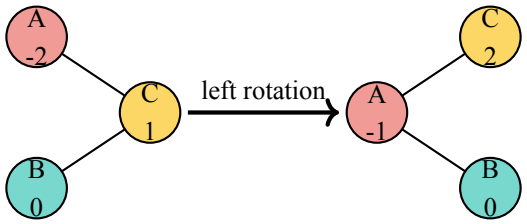


Figure 8.8: Unbalanced tree and its rotations

To perform rotations on a subtree, we can use left and right rotation rules. However, sometimes after performing one rotation, the balance may still be lost in the opposite direction.



In such cases, new rotation rules need to be used as follows:

• To balance a subtree with left rotation, we first check the balance factor of the right child node. If the right child is heavy, we perform a right rotation on it, and then a left rotation.

• Similarly, to balance a subtree with right rotation, we first check the balance factor of the left child node. If the left child is heavy, we perform a left rotation on it, and then a right rotation.

Although the process of subtree rotation is easy to understand conceptually, its code implementation can be complex because of the need to move nodes in the correct order while preserving all properties of the binary search tree. Additionally, ownership mechanisms in Rust make moving nodes and updating pointer relationships prone to errors. To implement rotation functions, we use node and subtree retrieval functions such as left_subtree and right_subtree.

```
// avl.rs

impl<T> AvlTree<T> where T : Ord {
    // get a avlNode
    fn node(&mut self) -> &mut AvlNode<T> {
        match self {
            Null => panic!("Empty tree"),
            Tree(node) => node,
        }
    }

    // get the left/right subtrees
    fn left_subtree(&mut self) -> &mut Self {
        match self {
            Null => panic!("Error: Empty tree!"),
            Tree(node) => &mut node.left,
        }
    }

    fn right_subtree(&mut self) -> &mut Self {
        match self {
            Null => panic!("Error: Empty tree!"),
            Tree(node) => &mut node.right,
        }
    }

    fn rotate_left(&mut self) {
        let mut n = replace(self, Null);
        let mut right = replace(n.right_subtree(), Null);
        let right_left = replace(right.left_subtree(), Null);
        *n.right_subtree() = right_left;
```

```
32          *right.left_subtree() = n;
33          *self = right;
34      }
35
36      fn rotate_right(&mut self) {
37          let mut n = replace(self, Null);
38          let mut left = replace(n.left_subtree(), Null);
39          let left_right = replace(left.right_subtree(), Null);
40          *n.left() = left_right;
41          *left.right_subtree() = n;
42          *self = left;
43      }
44  }
```

To maintain the balance of the tree, we can perform rotation operations. However, we also need to implement methods to obtain important information about the tree, such as the number of nodes, node values, tree height, minimum and maximum values, and node queries. This requires implementing methods such as size, leaf_size, depth, node, min, max, and contains for the balanced binary tree.

```
1   // avl.rs
2
3   impl<T> AvlTree<T> where T : Ord {
4       // Calculate the nodes number: number of left and right
5       // child nodes + root node, calculated recursively
6       fn size(&self) -> usize {
7           match self {
8               Null => 0,
9               Tree(n) => 1 + n.left.size() + n.right.size(),
10          }
11      }
12
13      // Calculate the leaf nodes number: calculated recursively
14      fn leaf_size(&self) -> usize {
15          match self {
16              Null => 0,
17              Tree(node) => {
18                  if node.left == Null && node.right == null {
19                      return 1;
20                  }
21                  let left_leaf = match node.left {
22                      Null => 0,
23                      _ => node.left.leaf_size(),
24                  };
25                  let right_leaf = match node.right {
26                      Null => 0,
27                      _ => node.right.leaf_size(),
28                  };
29                  left_leaf + right_leaf
30              },
31          }
32      }
33
34      // Calculate the number of non-leaf nodes
```

```
35      fn none_leaf_size(&self) -> usize {
36        self.size() - self.leaf_size()
37      }
38
39      // The depth of the tree is the maximum depth of the left
40      // and right subtrees plus one, calculated recursively
41      fn depth(&self) -> usize {
42          match self {
43            Null => 0,
44            Tree(n) => max(n.left.depth(),n.right.depth()) + 1,
45          }
46      }
47
48      fn is_empty(&self) -> bool {
49          match self {
50              Null => true,
51              _ => false,
52          }
53      }
54
55      // Get the maximum and minimum node values of the tree
56      fn min(&self) -> Option<&T> {
57          match self {
58              Null => None,
59              Tree(node) => {
60                  match node.left {
61                      Null => Some(&node.key),
62                      _ => node.left.min(),
63                  }
64              },
65          }
66      }
67
68      fn max(&self) -> Option<&T> {
69          match self {
70              Null => None,
71              Tree(node) => {
72                  match node.right {
73                      Null => Some(&node.key),
74                      _ => node.right.min(),
75                  }
76              },
77          }
78      }
79
80      // determine if key is in the tree
81      fn contains(&self, key: &T) -> bool {
82          match self {
83              Null => false,
84              Tree(n) => {
85                  match n.key.cmp(&key) {
86                      Equal => { true },
```

```
                        Greater => {
                            match &n.left {
                                Null => false,
                                _ => n.left.contains(key),
                            }
                        },
                        Less => {
                            match &n.right {
                                Null => false,
                                _ => n.right.contains(key),
                            }
                        },
                    }
                },
            }
        }
    }
}
```

In addition to these methods, we can implement the four traversal methods commonly used for binary trees: inorder, preorder, postorder, and levelorder. These traversal methods allow us to visit and manipulate each node in the tree in a specific order, which can be useful in various scenarios such as printing the tree, searching for a specific node, or computing some statistics about the tree.

```
// avl.rs

impl<T> AvlTree<T> where T : Ord {
    // Internal implementation of preorder, inorder, postorder,
    // and level-order traversal
    fn preorder(&self) {
        match self {
            Null => (),
            Tree(node) => {
                println!("key: {:?}", node.key);
                node.left.preorder();
                node.right.preorder();
            },
        }
    }

    fn inorder(&self) {
        match self {
            Null => (),
            Tree(node) => {
                node.left.inorder();
                println!("key: {:?}", node.key);
                node.right.inorder();
            },
        }
    }

    fn postorder(&self) {
        match self {
            Null => (),
```

```
31              Tree(node) => {
32                  node.left.postorder();
33                  node.right.postorder();
34                  println!("key: {:?}", node.key);
35              },
36          }
37      }
38
39      fn levelorder(&self) {
40          let size = self.size();
41          let mut q = Queue::new(size);
42
43          let _r = q.enqueue(self);
44          while !q.is_empty() {
45              let front = q.dequeue().unwrap();
46              match front {
47                  Null => (),
48                  Tree(node) => {
49                      println!("key: {:?}", node.key);
50                      let _r = q.enqueue(&node.left);
51                      let _r = q.enqueue(&node.right);
52                  },
53              }
54          }
55      }
56  }
57
58  // External implementation of preorder, inorder, postorder,
59  // and level-order traversal
60  fn preorder<T: Clone + Ord + Debug>(avl: &AvlTree<T>) {
61      match avl {
62          Null => (),
63          Tree(node) => {
64              println!("key: {:?}", node.key);
65              preorder(&node.left);
66              preorder(&node.right);
67          },
68      }
69  }
70
71  fn inorder<T: Clone + Ord + Debug>(avl: &AvlTree<T>) {
72      match avl {
73          Null => (),
74          Tree(node) => {
75              inorder(&node.left);
76              println!("key: {:?}", node.key);
77              inorder(&node.right);
78          },
79      }
80  }
81
82  fn postorder<T: Clone + Ord + Debug>(avl: &AvlTree<T>) {
```

```
83      match avl {
84          Null => (),
85          Tree(node) => {
86              postorder(&node.left);
87              postorder(&node.right);
88              println!("key: {:?}", node.key);
89          },
90      }
91  }
92
93  fn levelorder<T: Clone + Ord + Debug>(avl: &AvlTree<T>) {
94      let size = avl.size();
95      let mut q = Queue::new(size);
96      let _r = q.enqueue(avl);
97      while !q.is_empty() {
98          let front = q.dequeue().unwrap();
99          match front {
100             Null => (),
101             Tree(node) => {
102                 println!("key: {:?}", node.key);
103                 let _r = q.enqueue(&node.left);
104                 let _r = q.enqueue(&node.right);
105             },
106         }
107     }
108 }
109
110 fn main() {
111     basic();
112     order();
113
114     fn basic() {
115         let mut t = AvlTree::new();
116         for i in 0..5 { let (_r1, _r2) = t.insert(i); }
117
118         println!("empty:{},size:{}",t.is_empty(),t.size());
119         println!("leaves:{},depth:{}",t.leaf_size(),t.depth());
120         println!("internals:{}",t.none_leaf_size());
121         println!("min-max key:{:?}-{:?}",t.min(), t.max());
122         println!("contains 9:{}",t.contains(&9));
123     }
124
125     fn order() {
126         let mut avl = AvlTree::new();
127         for i in 0..5 { let (_r1, _r2) = avl.insert(i); }
128
129         println!("internal pre-in-post-level order");
130         avl.preorder(); avl.inorder();
131         avl.postorder(); avl.levelorder();
132         println!("outside pre-in-post-level order");
133         preorder(&avl); inorder(&avl);
134         postorder(&avl); levelorder(&avl);
```

```
135         }
136     }
```

The following are outputs afterr execution.

```
empty:false,size:5
leaves:3,depth:3
internals:2
min-max key:Some(0)-Some(4)
contains 9:false
internal pre-in-pos-level order
key: 1
key: 0
key: 3
key: 2
key: 4
key: 0
key: 1
key: 2
key: 3
key: 4
key: 0
key: 2
key: 4
key: 3
key: 1
key: 1
key: 0
key: 3
key: 2
key: 4
outside pre-in-pos-level order
key: 1
key: 0
key: 3
key: 2
key: 4
key: 0
key: 1
key: 2
key: 3
key: 4
key: 0
key: 2
key: 4
key: 3
key: 1
key: 1
key: 0
key: 3
key: 2
key: 4
```

8.5.3 Analysis of AVL Tree

An AVL balanced binary tree includes a rebalancing function and left and right rotation operations to maintain its balance, which enables its operations to maintain a relatively good performance with a worst-case complexity of $O(log_2(n))$. However, it is important to note that AVL trees perform a significant number of rotation operations to maintain balance. A red-black tree is an optimized binary tree that requires fewer rotations compared to an AVL tree. It is a weakened version of the AVL tree and can be used in scenarios with frequent insertions, deletions, and modifications to achieve better performance.

8.6 Summary

This chapter introduced us to trees, an efficient data structure that enables us to implement a wide range of useful algorithms. Trees find extensive applications in storage, networking, and other domains. Throughout this chapter, we completed the following tasks using trees:

- Parseed and evaluated expressions.
- Implemented binary heaps as priority queues.
- Implemented binary trees, binary search trees, and balanced binary search trees.

In the previous chapters, we learned about several abstract data types used to implement mapping relationships (Maps), including ordered tables, hash tables, binary search trees, and balanced binary search trees. The table below compares the worst-case performance of various operations supported by these data types. While red-black trees are an improvement over AVL trees, their complexity and performance remain similar to AVL trees, with only differences in coefficients. For further information, readers are encouraged to consult relevant resources.

Table 8.5: Performance of various abstract data types

Operation	Ordered table	Hash table	BST	AVL	Red-black tree
insert	$O(n)$	$O(1)$	$O(n)$	$O(log(n))$	$O(log(n))$
contains	$O(log(n))$	$O(1)$	$O(n)$	$O(log(n))$	$O(log(n))$
delete	$O(n)$	$O(1)$	$O(n)$	$O(log(n))$	$O(log(n))$

Bibliography

[1] The Open Group. Unix. Website, 1995. https://unix.org/what_is_unix.html.

[2] Multicians. Multics. Website, 1995. https://www.multicians.org.

[3] Linus. Linux kernel. Website, 1991. https://www.kernel.org.

[4] GNU. Gnu/linux. Website, 2010. https://www.gnu.org.

[5] Wikipedia. Quantum computer. 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 foundation. Website, 2021. https://foundation.rust-lang.org/members/.

[8] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 256–267, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Wikipedia. Np-complete problem. Website, 2021. https://zh.wikipedia.org/wiki/NP%E5%AE%8C%E5%85%A8.

[10] Wikipedia. Goldbach's conjecture. 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.

[11] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search—a log logn search. *Commun. ACM*, 21(7):550–553, jul 1978.

[12] Stanley P. Y. Fung. Is this the simplest (and most surprising) sorting algorithm ever?, 2021.

[13] Wikipedia. Entropy. Website, 2022. https://zh.wikipedia.org/wiki/%E7%86%B5.

[14] Wikipedia. Locality of reference. Website, 2022. https://zh.wikipedia.org/wiki/%E8%AE%BF%E9%97%AE%E5%B1%80%E9%83%A8%E6%80%A7.

[15] Wikipedia. Distance vector routing protocol. Website, 2021. https://en.wikipedia.org/wiki/Distance-vector_routing_protocol.

[16] Wikipedia. Link state routing protocol. Website, 2022. https://en.wikipedia.org/wiki/Link-state_routing_protocol.

[17] Wikipedia. Hamming code. Website, 2022. https://zh.wikipedia.org/zh-hans/%E6%B1%89%E6%98%8E%E7%A0%81.

[18] Bin Fan and David G Andarsen. Cuckoo filter: Practically better than bloom. Website, 2014. https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf.

[19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Website, 2008. https://bitcoin.org/bitcoin.pdf.