

Pocket
Guide



Data Structures and Algorithms with Go

Create efficient solutions and optimize your Go coding skills

Dušan Stojanović





Data Structures and Algorithms with Go

Create efficient solutions and
optimize your Go coding skills

Dušan Stojanović



www.bpbonline.com

First Edition 2024

Copyright © BPB Publications, India

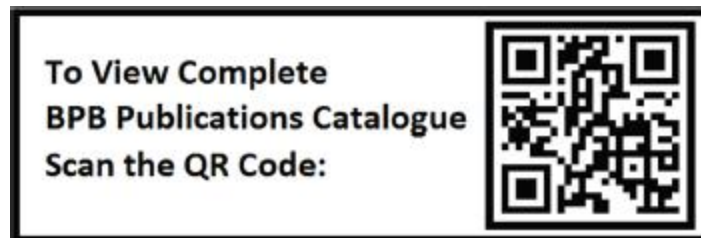
ISBN: 978-93-55518-897

All Rights No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.



www.bpbonline.com

Dedicated to

My beloved brother:
Nikola Stojanović

About the Author

Dušan Stojanović was born on May 27, 1989 in Smederevo, Serbia and raised in Baničina. He attended the University of Belgrade and received a Master's degree in computer science in 2013. Since then, he has been working in software development, playing central roles in numerous projects, such as user administration, online school platforms, e-commerce solutions, video-streaming platforms, and advertising solutions as a software engineer. His first book, *Building Server-side and Microservices with Go* was published in 2021. Furthermore, he has published and written several technical articles on microservice development with Go and related topics. Currently, he lives in Belgrade (Serbia), working as a senior software developer.

About the Reviewer

Mahima Singla is a passionate Principal Software Design Engineer with extensive professional experience and passion for designing and implementing robust, scalable software solutions in the domain of cloud assessment, cloud governance, cloud cost optimization, application fitment for cloud, cloud operating model, cutting-edge technologies, particularly in the realms of cloud computing, AWS, and Kubernetes in Go language.

She is currently working in Precisely Software and is part of the Studio Administrator Cloud project and Customer Onboarding project. She has played a pivotal role in architecting and developing solutions that harness the full potential of cloud platforms. She has proficiency in AWS, including services like AWS services EC2, S3, Lambda, has allowed her to create resilient and scalable applications that align with business objectives.

Additionally, her proficiency in Kubernetes reflects the commitment to staying at the forefront of container orchestration. She has successfully implemented and managed containerized workloads, ensuring efficient

managed containerized workloads, ensuring efficient resource utilization and high availability for applications.

Throughout the career, she has not only focused on technical excellence but also on driving innovation and fostering collaborative environments. Mahima takes pride in leading teams to deliver high-quality solutions that meet and exceed client expectations. She is dedicated to staying abreast of industry trends, and leveraging the latest technologies has allowed her to contribute meaningfully to the success of the projects she undertakes.

Acknowledgement

I want to express my deepest gratitude to my family and friends for their unwavering support and encouragement throughout the writing of this book, especially my brother Nikola and my parents, Slađana and Velimir.

I am also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. Revising this book was a long journey of, with valuable participation and collaboration from reviewers, technical experts, and editors. Thank you for your patience and cooperation.

I would also like to acknowledge the valuable contributions of my colleagues and co-workers who have taught me so much during the many years working in the tech industry. Special thanks to two of my dearest friends, Marijana Komatinović and Stefan Miletić, for supporting my work and providing valuable feedback.

I am grateful to Professor Milo Tomašević, who introduced me to the magical world of algorithms and data structures. Without him, this book would probably never have been written.

Finally, I would like to thank all the readers who have taken an interest in my book and for their support in making it a reality. Your encouragement has been invaluable.

Preface

Building a modern software solution can be a complex task that requires a comprehensive understanding of the latest technologies and the core of the problem. Knowing which data structure to use or some algorithm that can save resources and increase the solution's performance becomes essential. In addition, programming languages like Go are powerful tools that have become increasingly popular in software development.

This book is designed to provide a comprehensive guide for the most common data structures and the algorithms that can be executed on them. Everything will be followed with implementation in the Go programming language. Go's standard library offers completed implementations for some data structures (and algorithms). For others, everything will be developed from scratch. Numerous practical examples and illustrations will be provided to help you understand the concepts explained in each chapter.

This book is intended for developers familiar with the Go programming language who want to learn how to use it to implement algorithms. It is also helpful for developers who want to expand or refresh their knowledge of data structures and algorithms and improve their skills in building software solutions.

With this book, you will gain the knowledge and skills to become a proficient developer and recognize which data structure (or algorithm) suits a specific problem. I hope you will find this book informative and helpful.

[Chapter 1: Fundamentals of Data Structures and Algorithms](#) – This chapter introduces the reader to the concepts of data structures and algorithms. Some of the fundamentals that will be explained are characteristics of data structures, how data structures are represented in memory, how algorithms can be described, and how algorithms can be categorized. Furthermore, the chapter also gives a short history of algorithms and explains how algorithms and data structures are connected.

[Chapter 2: Arrays and Algorithms for Searching and Sorting](#) – This chapter presents a detailed overview of arrays (and slices as special types of arrays) and basic array operations. Furthermore, sorting and searching

algorithms will be shown, with detailed implementations and comparisons. This chapter will also tackle the subject of multidimensional arrays.

[Chapter 3: Lists](#) – This chapter covers lists and basic operations that can be performed on them. Explain the differences between types of lists, how lists are different from arrays, and how each list type can be implemented.

[Chapter 4: Stack and Queue](#) – This chapter introduces the reader to two linear data structures, stack and queue, that work on similar principles (there is an order in which elements are inserted and removed), demonstrating how to implement them. Besides regular queues, priority queues will be presented.

[Chapter 5: Hashing and Maps](#) – This chapter gives special attention to maps as one of the most popular data structures and the operations supported by them. This chapter also shows how to use maps in the Go programming language and explains the concept of hashing.

[Chapter 6: Trees and Traversal Algorithms](#) – This chapter covers all tree-related topics, like types of trees, usages,

and basic operations. The second part of the chapter will present and compare different traversal algorithms. Ultimately, the method of sorting an array with a tree will be shown.

[Chapter 7: Graphs and Traversal Algorithms](#) - This chapter introduces the fundamental concepts of graphs, with all operations that can be performed on them. This chapter also explains different traversal algorithms with details and numerous practical examples. The second part of the chapter explains what a spanning tree is, how to find the shortest path between nodes, how to calculate flow in graphs, and how to find the critical path.

Code Bundle and Coloured Images

Please follow the link to download the

Code Bundle and the Coloured Images of the book:

<https://rebrand.ly/8bb22e>

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the

quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our

products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Table of Contents

[1. Fundamentals of Data Structures and Algorithms](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Fundamentals of data structures](#)

[Characteristics of data structures](#)

[Memory representation](#)

[Structures in Go](#)

[History of algorithms](#)

[Fundamentals of algorithms](#)

[Representation of algorithms](#)

[Classification of algorithms](#)

[Algorithmic complexity and O-notation](#)

[Functions in Go](#)

[Data structures and algorithms](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[2. Arrays and Algorithms for Searching and Sorting](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Arrays](#)

[Operations](#)

[Arrays in Go](#)

[Slices](#)

[Slices in Go](#)

[Multidimensional arrays](#)

[Methods and interfaces in Go](#)

[Methods](#)

[Interface](#)

[Searching algorithms](#)

[Sequential search](#)

[Binary search](#)

[Searching algorithms in Go](#)

[Sorting algorithms](#)

[Insertion sort](#)

[Selection sort](#)

[Bubble sort](#)

[Quick sort](#)

[Sorting algorithms in Go](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[3. Lists](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Lists](#)

[Types of lists](#)

[Operations](#)

[Implementation of single-linked list](#)

[Lists versus arrays](#)

[Lists in Go](#)

[Double-linked list](#)

[Circular list](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[4. Stack and Queue](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Stack](#)

[Operations](#)

[Stack in Go](#)

[Stack implementation](#)

[Queue](#)

[Operations](#)

[Queue implementation](#)

[Priority_queue](#)

[Priority_queue in Go](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[5. Hashing and Maps](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Hashing](#)

[Hash function](#)

[Hash collision](#)

[Maps](#)

[Operations](#)

[Maps in Go](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[6. Trees and Traversal Algorithms](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Fundamentals of trees](#)

[Binary tree](#)

[Operations](#)

[Trees in Go](#)

[Insert operation](#)

[Delete operation](#)

[Traversal algorithms](#)

[Preorder](#)

[Inorder](#)

[Postorder](#)

[Level-order](#)

[Sorting an array with a tree](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[7. Graphs and Traversal Algorithms](#)

[Introduction](#)

[Structure](#)

Objectives

Fundamentals of graphs

Operations

Graphs in Go

Traversal algorithms

Breadth-first search

Depth-first search

Spanning tree

Prim's algorithm

Kruskal's algorithm

Transitive closure

Shortest paths

[Floyd's algorithm](#)

[Dijkstra's algorithm](#)

[Flow in graphs](#)

[Ford-Fulkerson algorithm](#)

[Topological sorting](#)

[Critical path](#)

[Conclusion](#)

[Points to remember](#)

[Multiple choice questions](#)

[Answers](#)

[Questions](#)

[Key terms](#)

[Index](#)

C
HAPTER
1

Fundamentals of Data Structures and Algorithms

Introduction

This chapter will introduce the general concepts related to data structures and algorithms. We will start by explaining some characteristics of data structures. After that, we will present a short history of algorithms, explain how to classify them and introduce O-notation. Near the end, the concept of functions in the Go programming language will be presented. This chapter will also explain the connection between algorithms and data structures.

Structure

The chapter covers the following topics:

Fundamentals of data structures

Characteristics of data structures

Memory representation

Structures in Go

History of algorithms

Fundamentals of algorithms

Representation of algorithms

Classification of algorithms

Algorithmic complexity and O-notation

Functions in Go

Data structures and algorithms

Objectives

By the end of this chapter, you will be able to understand the basic concepts of data structures and algorithms. This is a perfect starting point for implementing and solving many software problems. Structures and algorithms described in this book are often more complex than ones we will encounter in practical solutions.

Fundamentals of data structures

The data structure is a description of data organization. For example, a data structure representing a point in two-dimensional space should contain two values, each representing coordinates (x and y). Another example can be a structure that defines an address. That structure can contain information about street names, street numbers, cities, postal codes, and so on.

We have complete freedom to create data structures that best suit our needs. Later in this chapter, we will see how to use concepts of Go programming language to define desired data structure.

Besides these custom structures, there are a lot of well-known and established data structures, like arrays, graphs, maps, and so on. Through this book, we will get familiar with all of them.

On each data structure, operations specific to the given data structure can be performed. For example, we can perform a sort operation on an array. This operation will

sort array elements in the desired order. For each structure we will cover in this book, we will explain the basic operations that can be performed.

Characteristics of data structures

We have multiple data structures, but we can use some characteristics to classify them. In this section will present some of the most common characteristics and related classifications.

According to the relation of elements, we can classify data structures into the following groups:

Elements are arranged linearly (sequential). Each element is related to two neighboring elements (the previous and next elements). The most common linear structures are arrays, lists, stacks, and queues.

More complex relations between elements compared to linear data structures. One element can be in relation to multiple other elements. The well-known non-linear data structures are trees and graphs.

In the following chapters, we will cover all mentioned linear and non-linear data structures, with many code examples that will make everything clear and

understandable. The next section will illustrate how these structures are stored in memory.

We can classify elements based on the possibility of changing the size. Here, we have two classes:

These data structures have fixed size. There is no possibility of changing size during the execution of the program. For these data structures, memory usage is often not optimal.

Size can be changed during the execution of the program. The size will increase or decrease according to actual needs.

The third characteristic that can be used for classification is the type of elements. Here, we also have two classes:

Data structures consisting of elements of the same type. Array, where all elements are the same type, is an excellent example of a homogenous structure.

Heterogenous Opposite of homogenous, elements have different types. Some custom structure, where elements

have different types, is the most common example of heterogenous data structure. For example, if we define a structure representing a person, we will have string fields for name and address and an integer representing age.

Memory representation

To write and develop efficient programs, every software developer should be familiar with how elements of data structures are stored in memory. The two most common memory representations of data structures are sequential and linked.

In sequential representation elements are stored one after another in continuous memory space. The physical and logical order of elements is the same. Logical order represents element order defined inside our program, while physical order represents how elements are stored inside memory. It is illustrated as follows:



Figure 1.1: Sequential memory representation

It is important to note that one element can be stored in multiple memory locations. This depends on element and memory location sizes. If we have an element whose size

is 32 bits and the size of the memory location is 16 bits, one element will take two memory locations. An array of 5 such elements will take ten memory locations.

Before moving to linked memory representation, we must take a slight detour and explain the concept of pointers. The pointer can be defined as a variable that holds the memory address of another variable as a value or, as the name suggests, points to another variable. All modern programming languages support the concept of pointers, so Go is not an exception.

This statement will define the integer pointer:

```
var pi *int
```

Then, we can use the & (ampersand) operator to set the pointer to point to an integer variable:

```
i := 27
```

```
pi = &i
```

The pointer can be graphically represented as an arrow. In [Figure](#) we can see the memory space where the pointer and integer variable are stored:

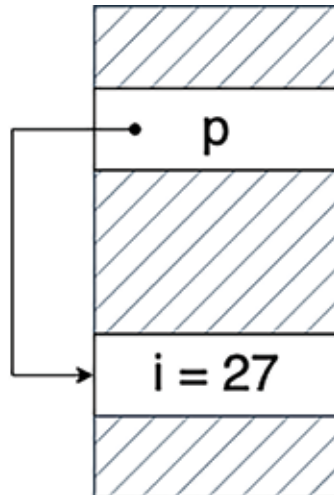


Figure 1.2: Pointers

Operator * (asterisk) is used to change or read pointed value (function `Println()` will display the value on standard output):

```
*pi = 18
```

```
fmt.Println(*pi)
```

In linked representation elements are arranged in non-continual space at random places in memory. Physical and logical order are usually different. Pointers are used to connect elements. This representation takes more space for individual elements because the pointer must be stored beside data, as shown:

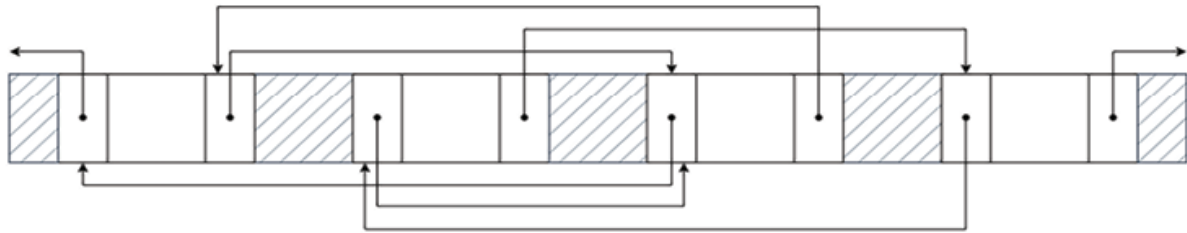


Figure 1.3: Linked representation

In the following chapters, when we cover specific data structures, we will see which memory representation is used for which data structure.

Structures in Go

Most popular data structures, like arrays and maps, are supported in Go and have dedicated data types. For all others, we can use a struct to create our custom data structures. Simply put, a struct can be defined as a collection of fields.

This code block will define a structure that represents a point in two-dimensional space:

```
type Point struct {  
  
    X int  
  
    Y int  
  
}
```

We can set values to all struct fields or omit some values. The default (zero) value will be assigned to omitted fields. In the following example, the value for Y

is omitted in the second statement, while both fields are omitted in the third statement:

```
var (  
  
    p1 = Point{27, 5}  
  
    p2 = Point{X: 18}  
  
    p3 = Point{ }  
  
)
```

We can access struct fields with the . (dot) operator:

```
p := Point{27, 5}  
  
p.X = 18
```

It is also possible to define a pointer to struct:

```
p := Point{27, 5}
```

```
pp := &p
```

Now that we have covered all topics related to data structures, we can proceed to algorithms.

History of algorithms

In this section, we will present a short history of algorithms. Persian mathematician Muhammad Al-Khwarizmi introduced the term algorithm. He wrote two texts, Book of Indian Computation and Addition and Subtraction in Indian where he introduced Indian numerals (which would later become known as Arabic numerals) and the decimal numeral system in Arabic mathematics.

In the XII century, these texts were translated into Latin as *Algorismi de numero indorum*, where the term *algorismi* is a Latinization and a bad translation of Al-Khwarizmi's name. Due to this mistranslation, the term algorithm was used for an extended period to denote procedure for arithmetic with a decimal numeric system.

The first algorithm, created for execution on a machine, was created in 1842 by Ada Lovelace (nee Byron), daughter of famous English poet George Gordon. It was an algorithm for generating Bernoulli numbers for Charles Babbage's analytical engine. Babbage died

before completing the machine, but this algorithm is recognized as the first computer algorithm.

Ada was later credited as the first computer programmer, and Ada programming language is named in her honor. Parts of Babbage's machine is displayed at the Science Museum in London.

Alan Turing is responsible for the introduction of algorithms in mathematics and logic. According to him, each well-defined algorithm can be executed on a Turing machine. This is not an actual machine; it is a mathematical model of computation. It can be described as an infinite strip of tape with symbols, where symbols are manipulated according to a table of rules. Head points to a symbol that is currently manipulated

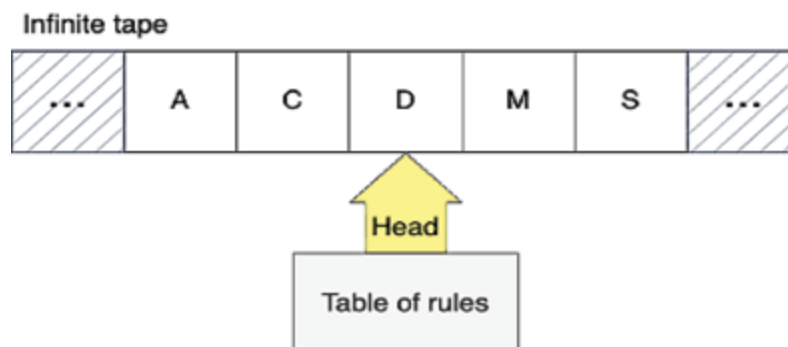


Figure 1.4: Turing machine

Fundamentals of algorithms

The algorithm can be defined as a process that takes some value or set of values and produces some value or set of values. Values taken by the algorithm are refereed as input, while values produced by the algorithm are refereed as output. In modern times, algorithms are mostly executed on computers.

The algorithm can also be defined as a process of transforming and manipulating input to create output. In other words, the algorithm performs some sort of data processing.

The correct algorithm will produce valid output for each input instance. An incorrect algorithm will not produce output for some input instances, or it will produce unexpected results. Here is an example of an incorrect algorithm. Let us assume that the result of the following equation $x^2 + x + 41$ will be a prime number for the non-negative integer x . This will be true in most cases, but for some integers greater than 39, the equation will produce a composite number. If we use this equation

algorithm will not produce the correct value for each input.

How can we know if the algorithm is correct or incorrect? We can use a proof of correctness, a mathematical procedure used to prove a theorem (algorithm in our case) with predicate calculus. The algorithm is expressed in logical expressions, and an invariant is defined. Invariant represents an expression whose value remains unchanged during the algorithm's running. An expression that was valid before the run was also valid after the run. Complete proof of correctness also checks if the algorithm will finish in a finite time. Here are the points to be kept in mind:

Discrete operations are executed in separate steps of the algorithm that leads to the final result (output).

The algorithm will produce an output after a finite number of steps (and time).

For the same input, the algorithm will always produce the same output. For example, if we pass value 2, which represents a side of a square, the algorithm that calculates the area of a square will always return value 4.

The algorithm is applicable to all types of inputs, not just for a particular set of inputs.

Representation of algorithms

Algorithms can be represented in many different ways. We can use the syntax of the actual programming language to describe and implement the algorithm, or we can describe the algorithm with natural language. Two popular methods for algorithm representation are pseudocode and flowcharts.

Pseudocode is a plain language used to describe algorithms. It is not an actual programming language but uses conventions that we can find in normal programming languages, like loops, if statements, functions, and so on. Its syntax is similar to

The following pseudocode describes a simple algorithm that determines which of two numbers is greater:

```
if (a >= b) then
```

```
    max = a
```


else

max = b

end_if

return max

Pseudocode is more focused on the essence of an algorithm rather than implementation, but it can be easily converted to any programming language.

Flowcharts are more graphical representations of algorithms. It is an excellent way to follow the execution flow of algorithm steps, but it can become unclear for complex algorithms. In [Table](#) we can see the most common building blocks used in flowcharts:

Table 1.1: Flowcharts building blocks

Building blocks are connected with arrows. The algorithm described with pseudocode in the previous

example can be represented with a flowchart from [Figure](#)

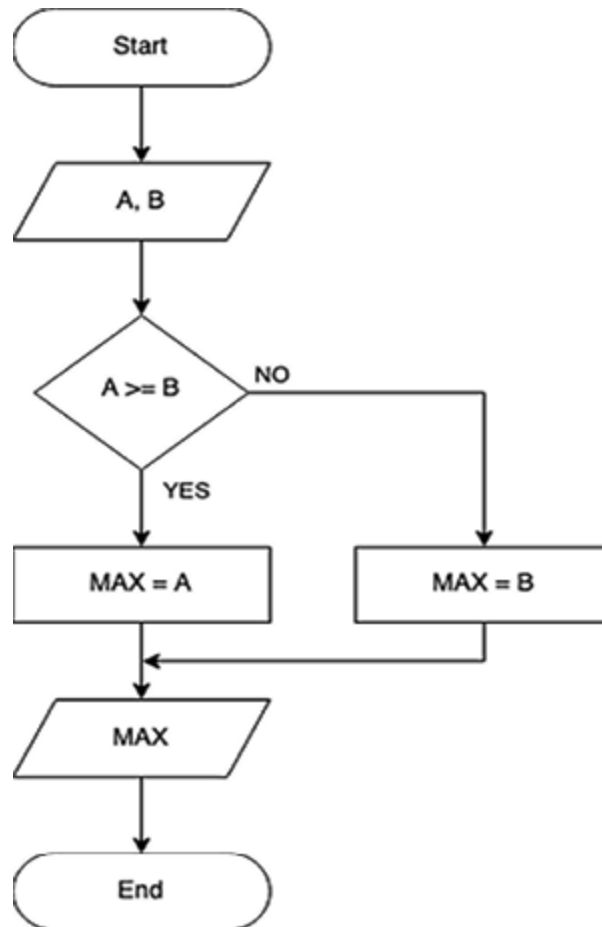


Figure 1.5: Flowchart

In this book, we will use Go to represent algorithms and present actual implementation. Methods mentioned in

this section are here to round the whole story, and they are essential concepts in modern software development.

Classification of algorithms

There are many different algorithms, and it is convenient to have some ways to classify them. In this section, we will see a couple of different classifications and mention some algorithm examples for each class without going into detail.

We can classify algorithms by design paradigm in the following classes:

Divide and Conquer The complexity of the problem is reduced by dividing the problem into two or more smaller problems of the same kind. The binary sort algorithm is a good example of this approach. An array is separated into multiple smaller arrays and then they are sorted.

Dynamic Programming Optimal solution can be constructed by overlapping optimal solutions of sub-problems. The same solution for a specific sub-problem can be used in different problem instances. For example, the shortest path between two cities can be found using the shortest

path between the destination city and neighboring cities of the starting city.

Greedy This approach is similar to dynamic programming, with one important difference. Sub-problem solutions are not always optimal. The algorithm will greedily select what seems to be the best option at the given moment. Kruskal's algorithm for finding minimal spanning tree from a graph is based on this approach.

Linear Mathematical methodology for modeling and solving the problem by finding the maximum or minimum of a linear function (by some criteria) under conditions expressed as linear equality or inequality.

Randomized Uses some degree of randomness as a part of the logic to find a solution.

Genetic Tries to find a solution by imitating an evolutionary process.

Heuristic Finds solution close to optimal solution. They are used when it is impractical to find an optimal

solution because of some limitations, like insufficient memory space for the execution of an algorithm.

We can also classify algorithms by implementation:

Iterative Uses loops to solve problems. Finding a minimal element in an array can be solved by looping through all elements.

Recursive They call themselves repeatedly until the condition is fulfilled. One of the usages of recursion is a calculation of Fibonacci numbers, which are part of the Fibonacci sequence. In the Fibonacci sequence, each number is the sum of the two preceding ones, ideal for recursion. We will see more examples of recursive algorithms in the following chapters. It is important to note that each problem solved with recursion can be solved iteratively. An iterative solution is often more complex than a recursive one.

Sequential Operations are executed in sequential order one by one. These algorithms are mainly designed for old computers with one CPU.

Parallel Divide the problem into a couple of sub-problems that can be executed in parallel. At the end, when processing is finished, results will be merged. These algorithms use the advantages of modern computers with multiple CPUs. Additional resources must be used for communication between processes that are executed in parallel.

Distributed Conceptually like parallel algorithms, with one difference, they use multiple computers connected via a network to execute sub-problems.

Algorithms can also be classified by field of study. Here are some of the most popular classes:

Search As the name suggests, they try to locate requested data in some data set (arrays, trees, graphs, and so on.) based on specified criteria.

Sorting Arrange a data set in a particular order (ascending, descending, alphabetical, and so on.).

Merge Merges data set by some established rule.

Algorithms for numerical Find approximate but accurate numeric solutions suitable for usage in other engineering branches.

Graph Algorithms that work with graphs.

String Algorithms that work with strings.

Combinatorial Generation of a large number of solutions for some combinatorial problem (like all lottery combinations).

Computational geometric Displaying geometrical shapes on a computer screen.

Machine learning Algorithms with the ability to adapt to new scenarios, learn from them and produce solutions based on previous experience.

Cryptography Algorithms for secure communication. During communication, the sending side encrypts data while the receiving side decrypts data. To interpret data correctly, the receiving side must somehow have information on which algorithm is used for encryption.

Compression Reduces physical space used for data storing. The most popular compression formats are zip and

Parsing Determine whether a given input sequence matches the syntax (or grammar) of a specified language.

There is one more classification of algorithms by complexity. This classification is very important and will be covered in a dedicated section.

Algorithmic complexity and O-notation

Algorithmic complexity is calculated in terms of the usage of two computer resources: space and time. Space represents occupied memory, while time represents the time used for CPU operations.

O-notation, often called Big is a way of presenting algorithmic complexity. O stands for which means an order of approximation.

For each class, we will again provide examples without details; we will see each algorithm class in future chapters. Based on complexity, we have the following classes:

Constant Complexity does not depend on input. The most preferable and rarest types of algorithms. Adding an element to the beginning of a list is an example of a constant algorithm.

Logarithmic algorithms Logarithmic function has slow growth, so algorithms from this class are preferred.

Logarithmic algorithms are based on the same principle: reduce the problem until the remaining problem is trivial. A binary search of an ordered array is designed based on this principle.

Linear algorithms Has for of loop that will be executed n times. An algorithm that processes all input data, like a sequential search of an unordered array, belongs to this class.

Log-linear algorithms log Most common class of algorithms, a combination of previous classes. The problem is split, but the whole input must be processed. The binary search algorithm is a representative of this class.

Square algorithms Has a form of two nested loops, where for each iteration of the outer loop, the nested loop will be executed n times. Direct sorting methods have square complexity.

Polynomial algorithms Has a form of k nested loops. Square algorithms are just a special case of polynomial algorithms, but because they are common, they are often placed into a separate class.

Exponential algorithms where $k > 2$ are not preferable, because exponential function has a fast growth. Algorithms for exhaustive search for all possibilities are exponential algorithms.

Classes are shown in order of increasing complexity. The least complex are constant algorithms, while the most complex are exponential algorithms. In general, the relationship between complexity can be described with the following expression:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(k^n) \text{ for } k > 2$$

If we have a choice between multiple algorithms, we should choose one with the lowest complexity. We will display the complexity of all algorithms presented in this book.

Functions in Go

All algorithms in this book will be implemented in some function, so this is a perfect place to introduce how to define functions in the Go programming language. The function is declared with the `func` keyword, function name, arguments, and return types.

The function can take zero or more arguments; in the following examples, we will see three functions. First will print Hello World to the standard output that takes zero arguments, the second increments the value of an integer that takes one argument, and the third one takes two arguments and the sum of two integers:

```
func hello() {  
  
    fmt.Println("Hello World")  
  
}  
  
func inc(i int) int {
```

```
    return i + 1

}

func sum(i int, j int) int {

    return i + j

}
```

As can be noticed, the argument type goes after the name. If more arguments share the same type, the type can be omitted from all arguments except the last one. Based on that, the function `sum()` from the previous example can be written in the following way:

```
func sum(i, j int) int {

    return i + j

}
```

The function may not return a value like the function `hello()` from the previous example; in that case, the return type is omitted from the definition. It is also possible for a function to return multiple results; in that case, types for all results must be placed between brackets, like in the following example:

```
func calc(i int) (int, int) {  
  
    return i*i, i+i  
  
}
```

Return values can be named. In that case, they will be treated as variables defined at the beginning of a function. The return statement without parameters will return named values. This is known as a naked return. The `inc()` function from previous examples can be written with named returns in the following way:

```
func inc(i int) (res int) {  
  
    res = i + 1  
  
    return
```

```
}
```

It is possible to pass a pointer as an argument. If the value referenced with the pointer is changed inside the function, even if we omit the return, the change will be visible outside the function. The function `inc()` can be rewritten to accept the pointer argument:

```
func inc(i *int) {
```

```
    *i = *i + 1
```

```
}
```

If we use this modified `inc()` function, the following code sample will print 6 because the integer variable is updated through the function:

```
func main() {
```

```
    i := 5
```

```
    pi := &i
```



```
inc(pi)
```

```
fmt.Println(i)
```

```
}
```

Data structures and algorithms

Data structures and algorithms represent two main building blocks of each software solution (program). One without the other does not make much sense. Generally speaking, data structures precede algorithms; the data must exist for the algorithm to manipulate it.

The complicity of data structures and algorithms are not directly related. It is possible to have a simple data structure over which a complicated algorithm is applied and vice versa.

A good choice of data structures and algorithms is essential for the efficiency of software solutions.

Conclusion

In this chapter, we covered all the fundamentals of data structures and algorithms necessary for understanding topics related to them. Some of these fundamentals include classes and characteristics of data structures and algorithms, how data structures are represented in memory, and how to express the complexity of algorithms.

Now, we are ready to start with our first data structure. In the following chapter, we will introduce arrays and algorithms that can be executed on them.

Points to remember

Specific operations can be performed on each data structure.

A single element (field) can be stored in multiple memory locations; this depends on the element (field) and memory location sizes.

The linked representation takes more space for individual elements because the pointer must be saved besides data.

In the Go programming language, the default (zero) value will be assigned to omitted structure fields.

Proof of correctness is a mathematical procedure used to prove a theorem (algorithm) with predicated calculus.

Pseudocode is not an actual programming language; it uses standard conventions from other programming languages like loops, if statements, or functions.

The algorithm complexity is calculated based on time used for CPU operations and space occupied in memory.

Data structures and algorithms are two building blocks for each program; one without the other does not make sense.

Multiple choice questions

Which pair does not represent a class of algorithms?

Linear, non-linear

Ordered, unordered

Static, dynamic

Homogenous, heterogenous

Pseudocode syntax is similar to which programming language syntax?

Go

Java

Python

Pascal

Which shape is used for process blocks in flowcharts?

Rectangle

Circle

Parallelogram

Diamond

Which shape is used for decision blocks in flowcharts?

Rectangle

Circle

Parallelogram

Diamond

Who was the first programmer?

Alan Turing

Ada Lovelace

Muhammad Al-Khwarizmi

Charles Babbage

Which O-notation represents an algorithm with constant complexity?

$O(n \log n)$

$O(1)$

$O(n)$

Which O-notation represents an algorithm with exponential complexity?

$O(n \log n)$

$O(1)$

$O(n)$

Answers

b

d

a

d

b

c

a

Questions

What are the differences between sequential and linear memory representation?

How is a correct algorithm defined?

What is a Turing machine?

How do divide and conquer algorithms work?

What is a recursion?

Key terms

Data Description of data organization.

Variable that holds the memory address of another variable.

A process that takes some value or set of values as an input and produces some values or set of values as output.

The plain language used to describe algorithms.

Graphical representation of algorithm.

C
HAPTER
2

Arrays and Algorithms for Searching and Sorting

Introduction

This chapter will introduce our first data structure, arrays. In the beginning, we will explain the basics of arrays and how to work with them in Go. Here, we will mention multidimensional arrays. Concepts of methods and interfaces will be presented in this chapter. These concepts are necessary for the understanding and implementation of certain algorithms that will be explained later. The second part will cover slices as a special type of arrays. After that, we will see some popular searching and sorting algorithms with implementations of the same.

Structure

The chapter covers the following topics:

Arrays

Slices

Multidimensional arrays

Methods and interfaces in Go

Searching algorithms

Sorting algorithms

Objectives

By the end of this chapter, you will be familiar with arrays and slices and how to use them in practical scenarios. Searching and sorting algorithms presented here can help you with everyday problems and provide new approaches to solving some issues.

Arrays

Arrays are a linear and homogenous data structure that contains a finite number of elements. It is an arranged structure, and it is known which element is in which place (which element is first in the array, which is second, and so on.). The index is the value that defines the elements' position in the array.

Arrays are stored in memory sequentially. [Figure 2.1](#) shows how an array containing four elements (with values 1, 4, 3, and 2) is stored. In this example, each element takes four memory locations. Hexadecimal numbers are usually used for memory addresses:

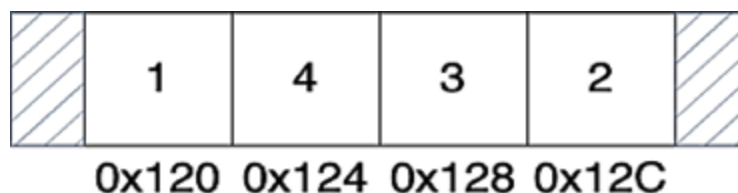


Figure 2.1: Memory representation of an array

Operations

For every data structure, we can create it and delete it. In this section, we will focus more on operations specific to specified structures.

For all array operations, the index is used for accessing specified elements. The following operations can be executed on arrays:

Select the value of the element.

Modify the value of the element.

The following section will show how to perform these operations with the Go programming language.

Arrays in Go

Arrays are one of the data types supported by Go.
Statement `[n]T` describes an array of `n` elements of type

The following statement will declare the variable as an array of eight elements (integers):

```
var a [8]int
```

If elements are not initialized, default (zero) values will be assigned to each element (0 for numerical values, false for Booleans, empty string for strings, and so on.).
If we add an initializer to the previous statement, we can initialize element values, as shown:

```
var a = [8]int{1, 18, 5, 27, 25, 8, 21, 9}
```

We can use an index to get the value of an element and assign it to another variable. Here we get the value of the element with index 3 and assign it to variable `b`:

```
var b = a[3]
```

An index can be used to change the value of an element. This statement will set value 7 to element with index 3:

$$a[3] = 7$$

As mentioned, an array has a fixed length and cannot be changed. In practice, length is an integral part of an array. This limit can be a problem for some real-world problems, but luckily, there is a data structure that can override it.

Slices

Slices can be defined as flexible arrays. Array size is a fixed value, while slices have a dynamic size, which is more common in practice. Generally, a slice represents a pointer to an array. We will see more detail when discussing how slices are implemented in Go.

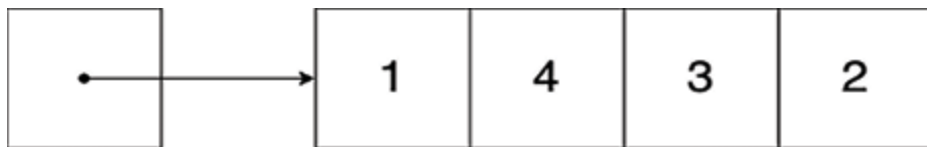


Figure 2.2: Slice

Slices have the same operations as arrays (get and write element).

Slices in Go

Statement `[]T` describes a slice with elements of type `T`. As we can see, the statement is similar to an array declaration statement, where length is omitted in the declaration.

A slice is formed by specifying two indices, known as low and high bound, separated with a colon, as shown:

```
s[low:high]
```

This low-high range is half-open; the first element is included, last is excluded. In the next example, slice `s` will contain elements 18 and 27 from the original slice. The last line will print the first element of a slice (element with index 0). In our case, the value 18 will be displayed:

```
a := [8]int{1, 18, 5, 27, 8, 25, 9, 21}
```

```
var s []int = a[1:4]
```

```
fmt.Println(s[0])
```

If we try to print the length and the last element of a slice, values 3 (length) and 27 (last element) will be displayed:

```
fmt.Println(len(s))
```

```
fmt.Println(s[len(s)-1]) // last element
```

Slice does not store actual data; it references part of the original (underlying) array. If we change any slice element, the corresponding element from the underlying array will be changed. If multiple slices reference the same array, change will affect all of them.

We can also use initializers to initialize elements of slices; the statement is similar to one used for array initialization:

```
var s = []int{1, 18, 5, 27, 25, 8, 21, 9}
```

The previous statement will create an underlying array and a slice that references it.

The default value for the low bound is 0, while the default value for the high bound is the length of the slice. If some bound is omitted, the default value will be used. The length of the slice created in the previous example is In the following code segment, we will create four slices, where we will omit some bounds and show elements in comments on the right side:

```
s1 := s[1:4] // [18 5 27]
```

```
s2 := s[:4] // [1 18 5 27]
```

```
s3 := s[1:] // [18 5 27 8 25 9 21]
```

```
s4 := s[:] // [1 18 5 27 8 25 9 21]
```

Slices have two attributes that are part of type: length and capacity. Length represents the number of elements the slice contains, while capacity represents the number of elements in the underlying array (counting from the first element of the slice). In the following example, slice `s`, created from underlying array `a`, will have a length of 3 and a capacity of 7:

```
a := [8]int{1, 18, 5, 27, 8, 25, 9, 21}
```



```
var s []int = a[1:4]
```

The length and capacity of slice `s` can be obtained with functions `len(s)` and `cap(s)`. The length can be extended if there is sufficient capacity.

The default (zero) value for a slice is `nil`. The length and capacity of `nil` slice are 0, and there is no underlying array.

Slice can also be created with the `make()` function in the following way:

```
s := make([]int, 0, 5)
```

This function takes three parameters: type, length, and capacity. An array with default (zeroed) values will be created with a slice referencing that array.

It is possible to create a slice from another slice; these slices will reference the same underlying array:

```
s1 := []int{1, 18, 5, 27, 8, 25, 9, 21}
```

```
s2 := s1[1:4]
```

We can add an element at the end of the slice with the function `append`. The result will be a slice that contains all elements of the original slice plus added ones:

```
s := []int{1, 18, 5, 27, 8, 25, 9, 21}
```

```
s = append(s, 3, 21, 12, 30)
```

The function takes the original slice and elements that will be added as parameters. If the underlying array is too small for all new elements, a new, bigger one will be allocated. The return value of the `append()` function is often assigned to the same slice used as a function parameter.

Multidimensional arrays

Arrays and slices described in this chapter can be defined as one-dimensional. It is possible to define multidimensional arrays. Two-dimensional arrays, also known as are the most widespread.

The following statement will declare a matrix with a dimension of 3x3:

```
var matrix [3][3]int
```

Default (zero) values will be assigned to each uninitialized element. We can use an initializer to initialize element values:

```
var matrix = [3][3]int{{1, 18, 5}, {27, 25, 8}, {21, 9, 12}}
```

Indices are used to get the value of an element. Here, we get the value of the element from the first row and the second column (columns and rows start from 0) and assign it to variable n:

```
var n = matrix[1][2]
```

Indices can also be used to modify elements. This statement will set value 7 to element with indices 2 and

```
a[2][0] = 7
```

Matrices are called squared when both dimensions are identical (like 3x3). The first dimension represents the number of columns, the second number of rows. We can declare matrices with unequal dimensions (like 3x5 or 5x2).

We can omit a number of columns or both dimensions from the matrix declaration:

```
var m1 [3][]int
```

```
var m2 [][]int
```

If we omit some dimension, we can only add elements with the `append()` function by adding whole rows in the following way:

```
var matrix [][]int
```

```
matrix = append(matrix, []int{1, 18, 5}, []int{27, 8,  
25})
```

Matrices are stored in memory sequentially, by rows, as we can see in [Figure](#). Values presented inside blocks that represent elements are indices. As we can see, the first row will be stored after the zero row is stored:

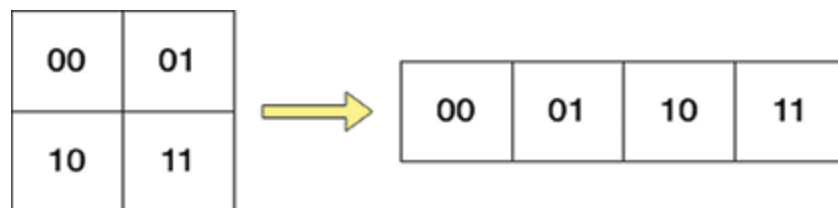


Figure 2.3: Memory representation of the matrix

Alternatively, matrices can be stored by columns.

Methods and interfaces in Go

For the implementation of some algorithms, we will use methods and interfaces. To completely understand all the examples and code samples presented in this book, we should get familiar with methods and interfaces in the Go programming languages.

Methods

The method can be defined as a function declared on a type or as a function with a receiver argument. The easiest way to explain this concept is with an example. We will define a struct that represents rectangular:

```
type Rectangle struct {  
  
    a, b int  
  
}
```

The integer fields a and b represent the length and width of a rectangle. We can use regular functions to calculate area and but we can also define methods on a rectangle type in the following way:

```
func (r Rectangle) Area() int {  
  
    return r.a * r.b  
  
}
```

```
func (r Rectangle) Perimeter() int {  
  
    return 2*r.a + 2*r.b  
  
}
```

As we can see, receiver argument r is placed in brackets between the func keyword and method name. The syntax diagram for the function and method declaration is represented in [Figure](#)

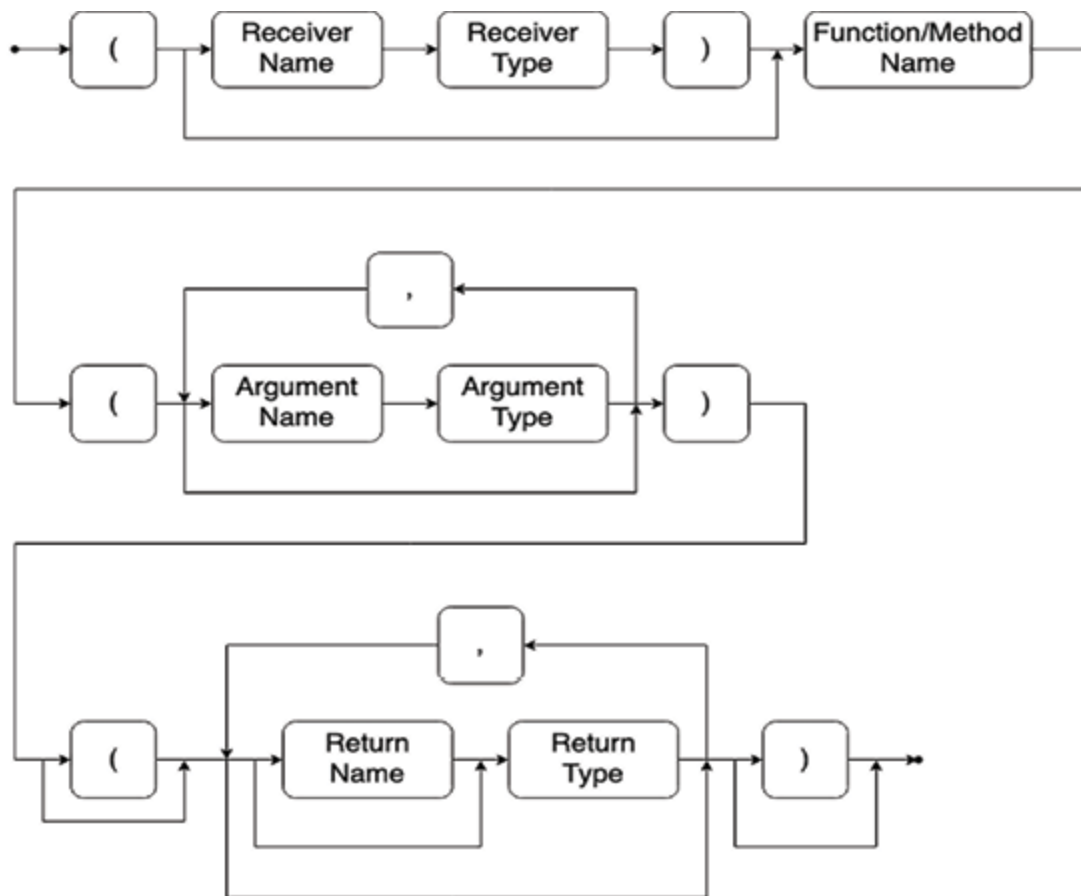


Figure 2.4: Syntax diagram for the function and method declaration

It is possible to use a pointer receiver. In this way, the variable that calls the method will be changed. This is common in practice.

Here is a small code example that uses methods from the previous example:

```
func main() {  
  
    r := Rectangle{2, 3}  
  
    fmt.Println("Area: ", r.Area())  
  
    fmt.Println("Perimeter: ", r.Perimeter())  
  
}
```

Interface

An interface is a set of method declarations. The type can implement an interface by implementing all its methods. There is no dedicated keyword for this.

We can define an interface for a mathematical shape with two methods: one will calculate the area, and another will calculate the perimeter. The keyword `interface` is used in the definition, as we can see in the following example:

```
type Shape interface {  
  
    Area() int  
  
    Perimeter() int  
  
}
```

We can define a struct for different shapes that can implement these methods. In the following code

segment, we will see two structs and that implement the Shape interface:

```
type Rectangle struct {  
  
    a, b int  
  
}  
  
func (r Rectangle) Area() int {  
  
    return r.a * r.b  
  
}  
  
func (r Rectangle) Perimeter() int {  
  
return 2*r.a + 2*r.b  
  
}  
  
type Square struct {
```

```

    a int

}

func (s Square) Area() int {

    return s.a * s.a

}

func (s Square) Perimeter() int {

    return 4 * s.a

}

```

To provide a whole solution, we will present the main() function that uses interfaces and methods from the previous examples:

```

func main() {

    r := Rectangle{2, 3}

    fmt.Println("Area: ", r.Area())
}

```

```
fmt.Println("Perimeter: ", r.Perimeter())
```

```
s := Square{4}
```

```
fmt.Println("Area: ", s.Area())
```

```
fmt.Println("Perimeter: ", s.Perimeter())
```

```
}
```

Now, when we are familiar with the concepts of methods and interfaces, we can proceed to concrete algorithms.

Searching algorithms

In general, searching represents the process of locating the desired data in a data set based on some identification. Identification is often called a The data set being searched can be sorted or unsorted.

There are two possible search results:

Data is found.

Data is not found.

We will cover two popular searching algorithms: sequential search and binary search.

Sequential search

Sequential search is quite a simple technique; the required key is compared to each element in the array until a match is found (successful search) or all elements are checked (unsuccessful search). This technique is usually the least effective; the complexity of this algorithm is $O(n)$. The flowchart shown in [Figure 2.5](#) describes the sequential search algorithm:

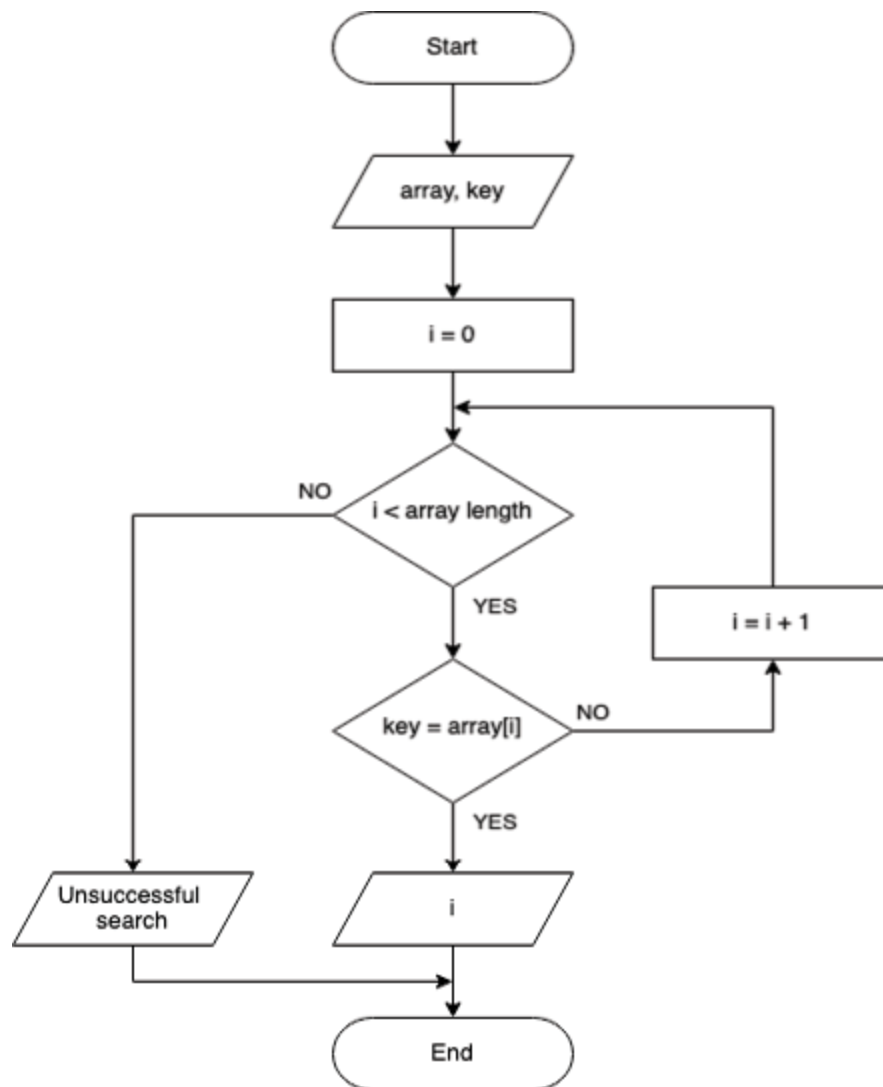


Figure 2.5: Flowchart diagram for sequential search

This algorithm can be used for both ordered and unordered arrays. Actually, for unordered arrays, this is the only search option. For ordered arrays, efficiency increases because we can stop searching when the elements become greater than the key.

Implementation in Go is simple: one for loop that iterates through elements, and when we find a match, we can break the loop and return the index of the matching element. In case of an unsuccessful search, we can return some illegal value. In our example, we will return -1 (minus one). All examples will be provided for integer arrays, as shown:

```
func SeqSearch(array []int, key int) int {  
  
    for i := 0; i < len(array); i++ {  
  
        if key == array[i] {  
  
            return i  
  
        }  
  
    }  
  
    return -1  
  
}
```

We can also implement this algorithm with the for range loop. This is a special type of for loop in Go designed for iterations over arrays, slices, and maps. Two values are returned in each iteration, an index and a copy of the element at that index:

```
func SeqSearch(array []int, key int) int {  
  
    for i, elem := range array {  
  
        if key == elem {  
  
            return i  
  
        }  
  
    }  
  
    return -1  
  
}
```

Binary search

Sequential search will, for sorted arrays, discard only one element in case of non-match. A binary search algorithm fixes this issue and discards a group of elements. This is an algorithm from the divide and conquer class that divides an array's remaining elements in two halves in each iteration to find the specified key. It can only be used with sorted arrays.

The algorithm will calculate three values in each iteration: low, mid, and high. The mid represents the middle index and can be calculated when the sum of low and high values is divided by 2.

If the element on the mid position matches the key, the search is successful, and the mid will be returned. If that is not the case, we check which half of the array key can be found; if it is lower than the value of the mid element, it can be found in the lower half of the array. The low value will stay the same, and the new high value will be a mid minus one. This will exclude all elements from the higher half of the array and the mid element from future searches.

In case the key is greater than the value on the mid position, the high will stay the same, while the new low value will be mid plus one. This will exclude all elements from the lower half and the mid element from future searches.

The search will continue until a match is found (successful search) or the low value is greater or equal to the high value (unsuccessful search). The flowchart for binary search is shown in [Figure](#)

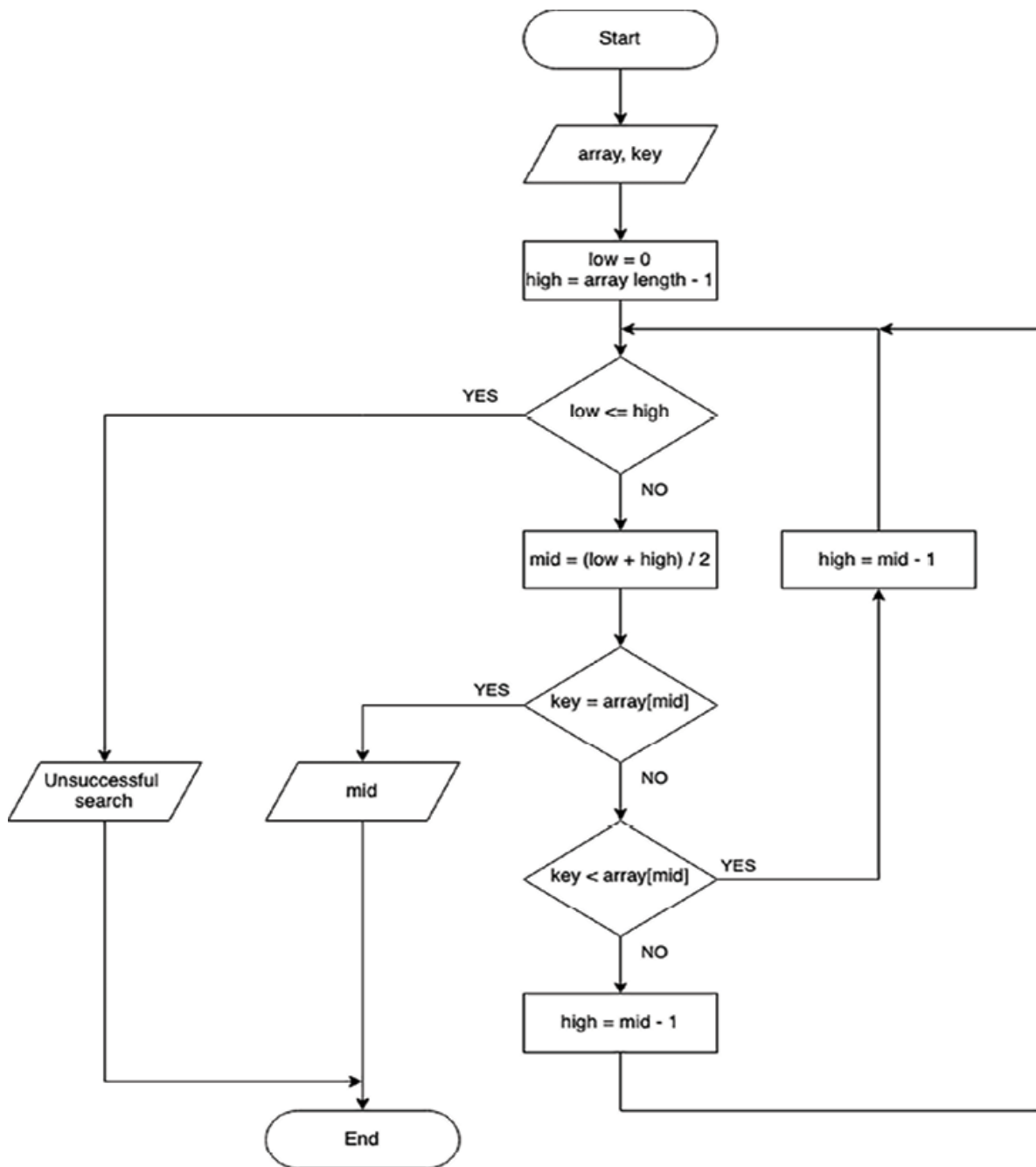


Figure 2.6: Flowchart diagram for binary search

Iterations of this algorithm are illustrated in [Figure 2.7](#) when an array is searched for key 18:

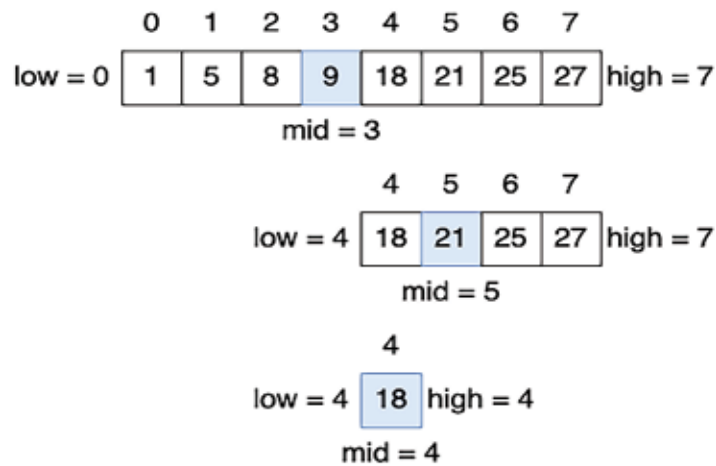


Figure 2.7: Binary search on key 18

Now it is clear why the array must be sorted. The algorithm will constantly reduce problems based on element value and prediction in which half key can be found. For an unsorted array, this prediction makes no sense. The complexity of binary search is $O(n \log n)$ so this algorithm is sometimes called logarithmic

Here is a Go implementation of the binary search algorithm; again, in case of an unsuccessful search, value -1 will be returned:

```
func BinSearch(array []int, key int) int {
```

```
    low := 0
```

```
high := len(array) - 1
```

```
for low <= high {
```

```
    mid := (low + high) / 2
```

```
    if key == array[mid] {
```

```
        return mid
```

```
    } else if key < array[mid] {
```

```
        high = mid - 1
```

```
    } else {
```

```
        low = mid + 1
```

```
    }
```

```
}
```

```
return -1
```


}

Some search algorithms are part of the Go standard library.
We will see how to use them in the following section.

[Searching algorithms in Go](#)

Go has three functions in package `sort` that implement binary search algorithm:

```
func SearchInts(a []int, x int) int
```

Searches for `x` in a sorted slice of ints.

```
func SearchFloat64s(a []float64, x float64) int
```

Searches for `x` in a sorted slice of floats.

```
func SearchStrings(a []string, x string) int
```

Searches for `x` in a sorted slice of strings.

All functions return the index of the matching element. If a match is not found, the returned value will represent the index where `x` should be inserted. Slice must be sorted in ascending order.

Previously presented functions are just wrappers around Search() function for common cases. The Search() function uses binary search to find the smallest index of the specified value in a sorted array or slice. It accepts two parameters, the slice (array) length and the bool function that is used for comparison:

```
func Search(n int, f func(int) bool) int
```

Function for example, is implemented in the following way:

```
func SearchInts(a []int, x int) int {  
  
    return Search(len(a), func(i int) bool { return a[i]  
    >= x})  
  
}
```

For a slice sorted in ascending order, if we search for value 18, SearchInts() will return the smallest index for which a[i] is greater or equal to 18. This will return the first index where a match is found or the index where an element with value 18 should be inserted. For slices

sorted in descending order, the operator \leq would be used instead of the \geq operator.

Sorting algorithms

Sorting is defined as rearranging a set of data in a specified order. It is a common operation, and most algorithms are based on a direct comparison of elements. In this section, we will see a couple of well-known algorithms.

Insertion sort

In this algorithm, an array or slice that should be sorted is viewed as two subarrays, sorted and unsorted. In each iteration, one element from the unsorted subarray is inserted in the appropriate place in the sorted subarray. Initially, only the first element of the array will belong to the sorted subarray. The algorithm will end when the unsorted subarray is empty.

The following code sample implements insertion sort with Go. Because the first element (one with index 0) is a part of the sorted subarray, the outer loop will start from the second element (one with index 1). The inner loop will go from the current element, which now becomes part of a sorted subarray, to the beginning of the array to insert it in the proper place and move other elements:

```
func InsertionSort(array []int) []int {  
  
    for i := 1; i < len(array); i++ {  
  
        k := array[i]  
  
        j := i - 1
```

```
for j >= 0 && array[j] > k {  
  
    array[j+1] = array[j]  
  
    j = j - 1  
  
}  
  
array[j+1] = k  
  
}  
  
return array  
  
}
```

The previous code can be described with a flowchart from [Figure](#)

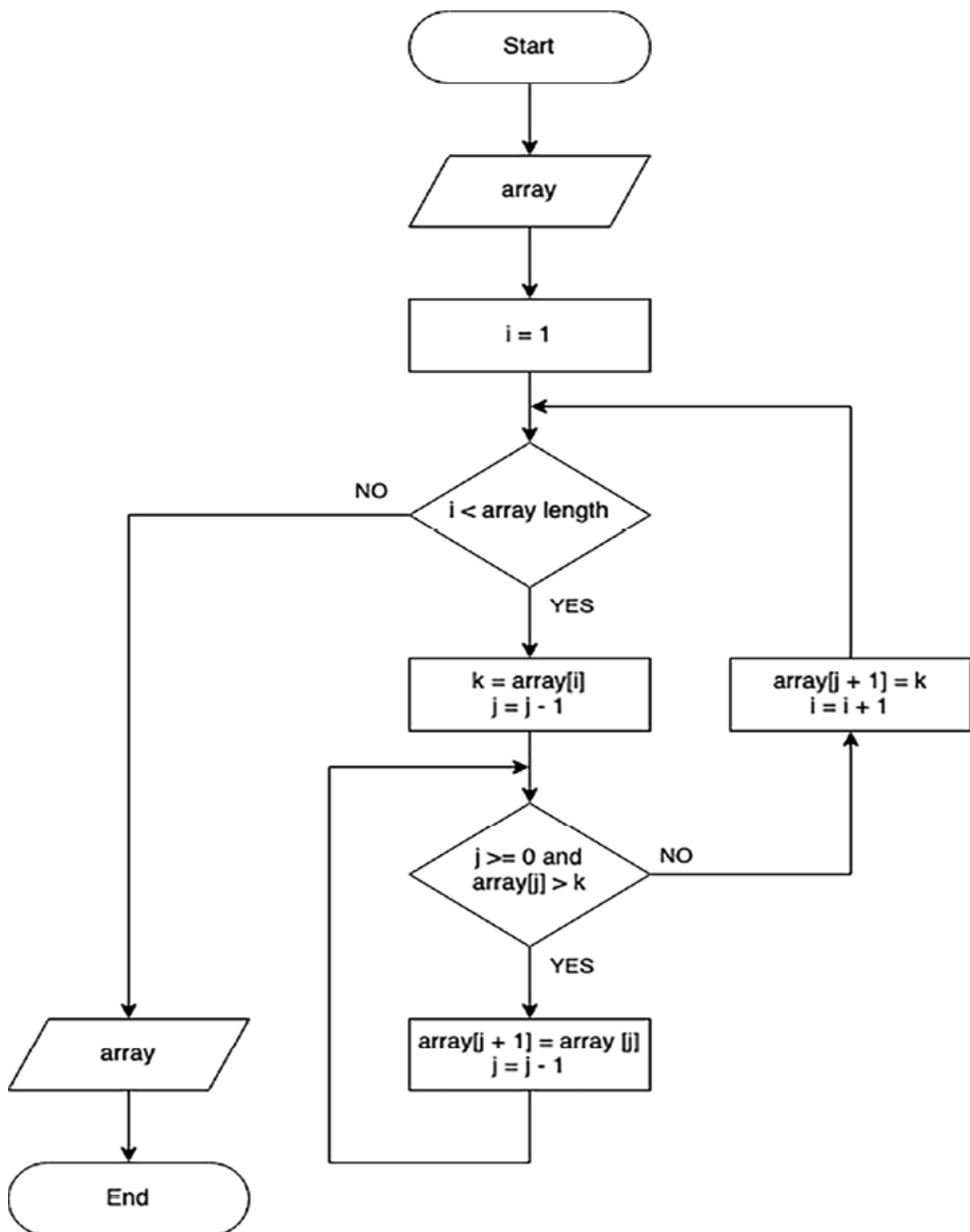


Figure 2.8: Flowchart diagram for insertion sort

In [Figure](#) we can see how sorted and unsorted subarrays are changed with each iteration:

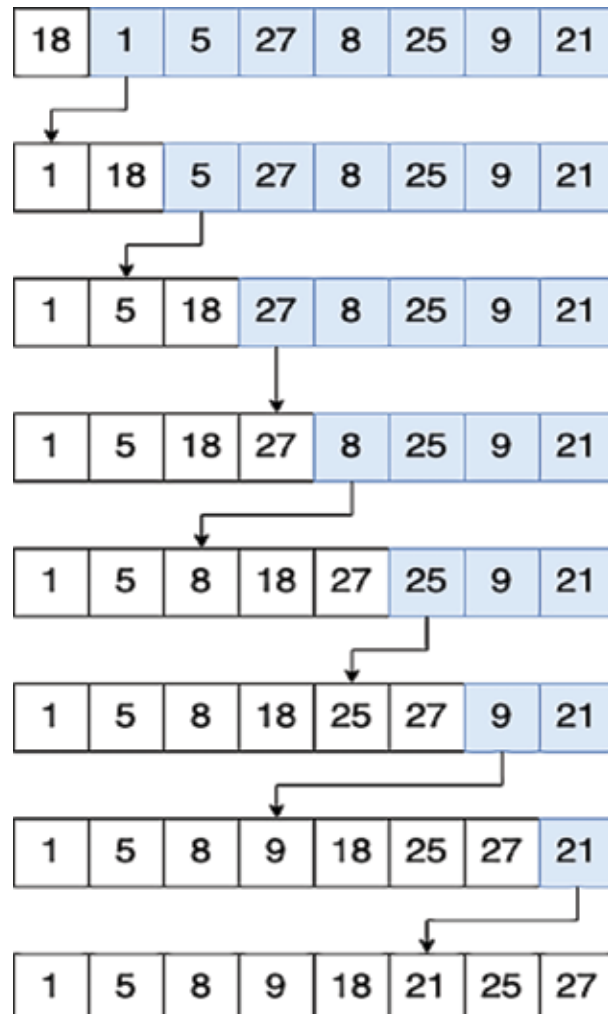


Figure 2.9: Insertion sort iterations

The complexity for insertion sort is $O(n^2)$ which is unsuitable for large arrays and slices.

Selection sort

This algorithm also separates arrays or slices into sorted and unsorted subarrays. All elements are part of unsorted subarrays at the beginning. In each iteration, the smallest element from the unsorted subarray will be found and placed on the end of the sorted subarray.

That position is currently occupied by the first element of the unsorted subarray, so the position of that element and the smallest element will be switched. The algorithm will end when the unsorted subarray is empty. In [Figure](#) the selection sort flowchart is shown:

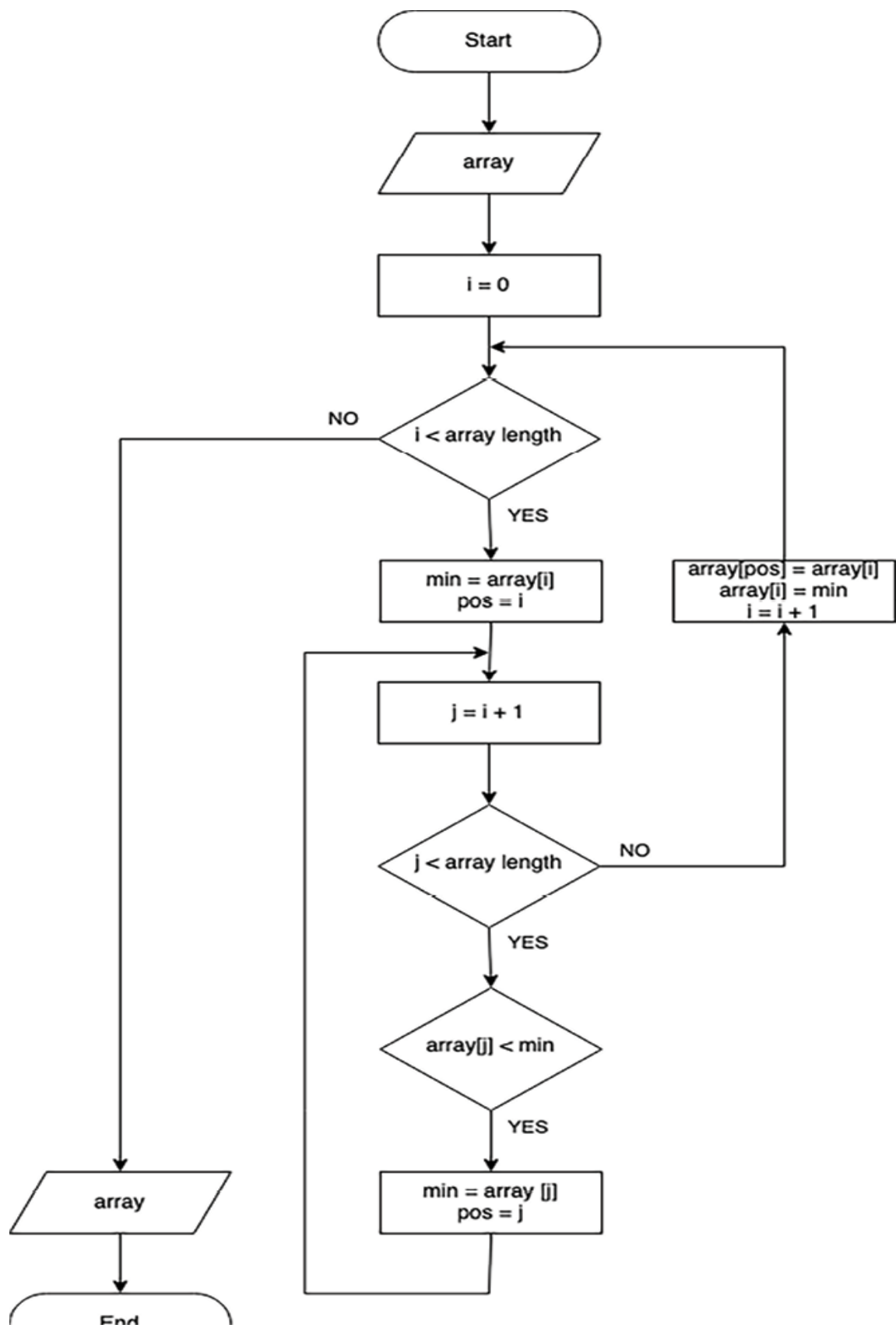


Figure 2.10: Flowchart diagram for selection sort

Iterations of the selection sort algorithms are presented in [Figure](#)

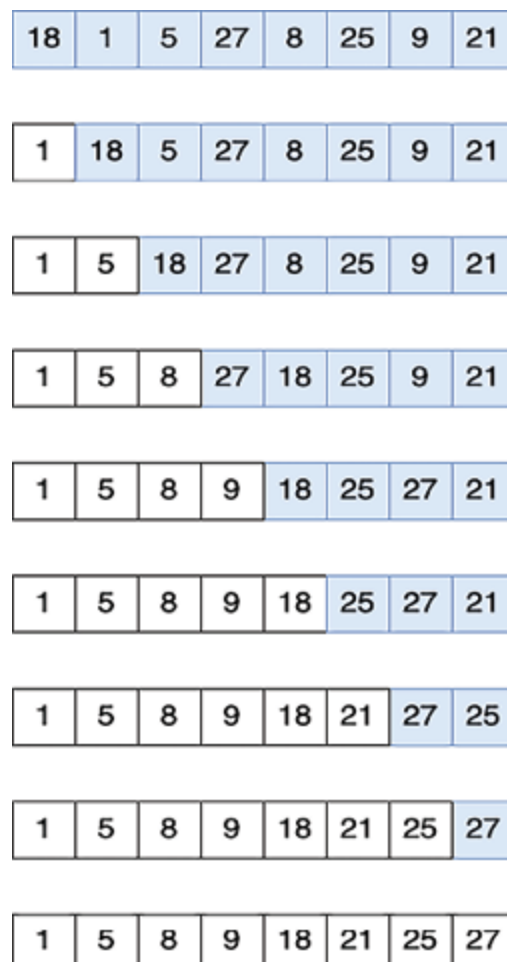


Figure 2.11: Selection sort iterations

Selection sort can be implemented with the following code snippet. The inner loop will find the smallest element in the unsorted subarray, after which the element is placed at the end of the sorted subarray:

```
func SelectionSort(array []int) []int {  
  
    for i := 0; i < len(array)-1; i++ {  
  
        min := array[i]  
  
        pos := i  
  
        for j := i + 1; j < len(array); j++ {  
  
            if array[j] < min {  
  
                min = array[j]  
  
                pos = j  
  
            }  
        }  
    }  
}
```

```
}  
  
array[pos] = array[i]  
  
array[i] = min  
  
}  
  
return array  
  
}
```

The complexity of this algorithm is and it is suitable for smaller arrays and slices.

Bubble sort

This is the simplest but the most ineffective approach for sorting an array or slice. This algorithm will iterate through the array multiple times and compare each element with the next one. If elements are not in proper order, their places will be switched. After the first iteration, the largest element will be at the end of the array; after the second iteration, the largest unsorted element will be placed at the penultimate place, and so on. This procedure is described in the flowchart in [Figure](#)

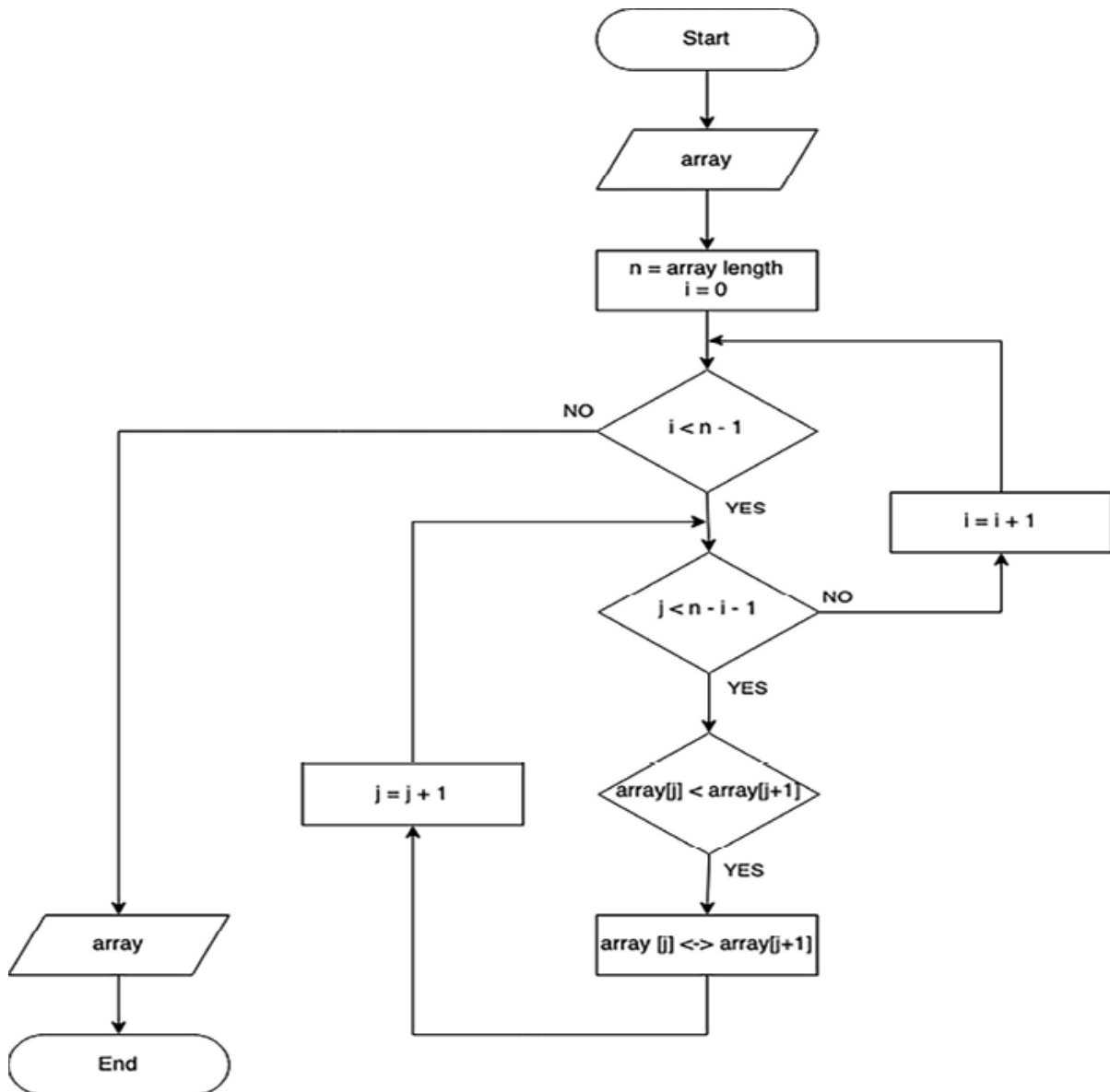


Figure 2.12: Flowchart diagram for bubble sort

In [Figure](#) we can see how elements will be placed after each iteration. The algorithm got its name because elements float to the surface, similar to bubbles:

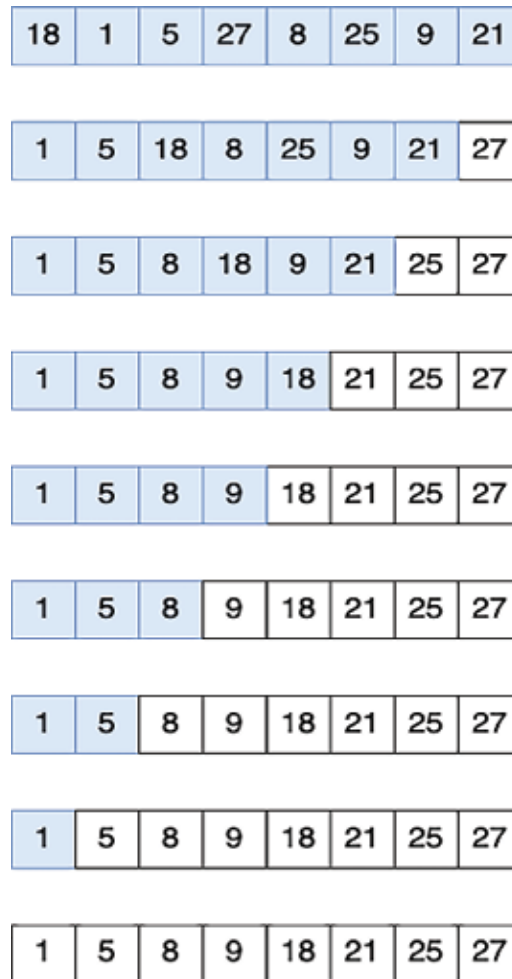


Figure 2.13: Bubble sort iterations

Implementation is simple; two nested loops will iterate through all elements and switch their position, if necessary, as shown:

```
func BubbleSort(array []int) []int {  
  
    n := len(array)
```

```

for i := 0; i < n-1; i++ {

    for j := 0; j < n-i-1; j++ {

        if array[j] > array[j+1] {

            array[j], array[j+1] =

            array[j+1], array[j]

        }

    }

}

return array

}

```

The complexity of bubble sort is $O(n^2)$ and it is not recommended for usage on larger arrays and slices.

Quick sort

This is one of the algorithms from the divide and conquer class. The element that separates the array (or slice) into two parts, called is selected. Usually, we select the first element from the array (slice). Then, all other elements are classified into two partitions: lower, where elements are lower or equal to the pivot, and higher, where elements are greater than the pivot. When elements are classified, we apply the same procedure to the two partitions.

In [Figure](#) we can see how the pivot is selected. First, we choose the first element as a pivot (with a value of 18). Other elements are classified around the pivot, and we have two partitions, lower with elements and 9 and higher with elements and Element with value 5 will be selected as a pivot for lower partition, while element with value 25 will be chosen as a pivot for higher partition. This process will repeat until each partition has only one element. Therefore, the array is sorted, as shown:

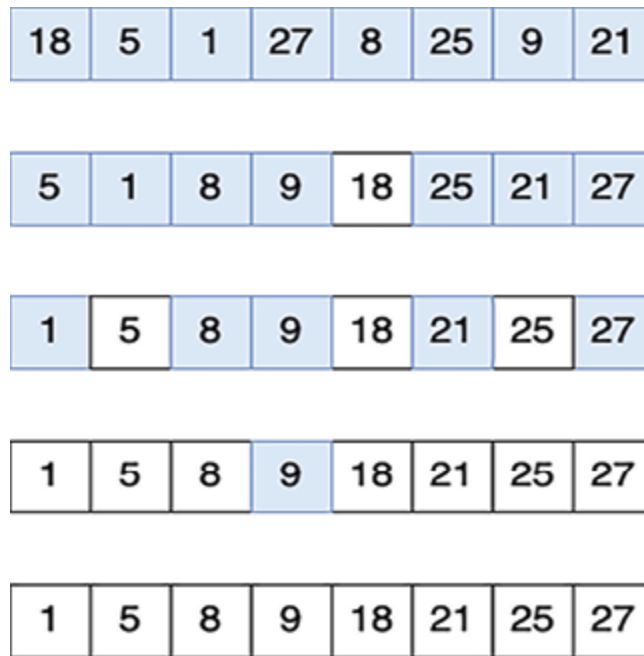


Figure 2.14: Quick sort

The described procedure for finding a pivot is shown in the flowchart in [Figure](#)

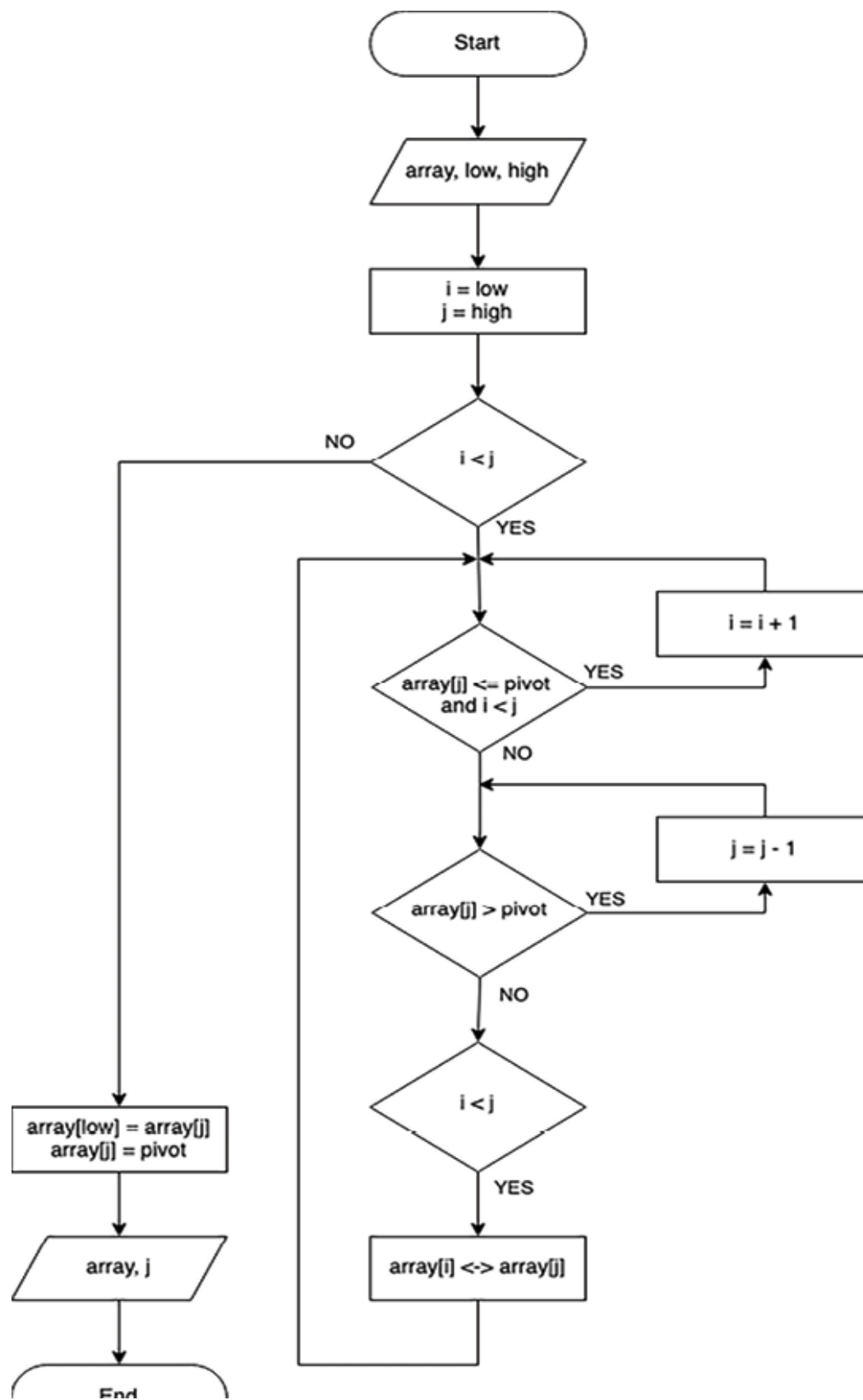




Figure 2.15: Flowchart diagram for finding pivot procedure

This algorithm can be implemented recursively because the same process is constantly repeated. Function `partition()` will return the position of the pivot, and for every partition that is formed, we will accompany two values that represent low and high boundaries. For the lower partition, new boundaries will be the original partition's low boundary and an index with a value of one less than the pivot position for the high boundary. For a higher partition, the low boundary will be one greater than the pivot position, while the high boundary will have a value of the high boundary from the original partition:

```
func QuickSort(array []int, low, high int) []int {  
  
    if low < high {  
  
        array, j := partition(array, low, high)  
  
        QuickSort(array, low, j-1)
```

```

        QuickSort(array, j+1, high)

    }

    return array

}

```

Inside the partition() function, two indices, i and j are used to classify elements in proper partitions. Index i starts from the lower bound of the processed partition, moves across elements that are not greater than the pivot, and marks the current higher bound of the lower partition. The other index, j, starts from the higher bound of the processing partition, moves across elements greater than the pivot, and marks the current lower bound of the higher partition. If elements on positions i and j are in the wrong order, they will switch positions. The process is repeated until indices meet each other:

```

func partition(array []int, low, high int) (int, int) {

    pivot := array[low]

```


i := low

j := high

for i < j {

for array[i] <= pivot && i < j {

i++

}

for array[j] > pivot {

j--

}

if i < j {

array[i], array[j] =

array[j], array[i]

```
}
```

```
}
```

```
array[low], array[j] = array[j], pivot
```

```
return array, j
```

```
}
```

The efficiency of the quick sort algorithm depends on the partition size balance. Best performances are archived when the partition can be separated into two equally sized partitions in each step. In best-case scenarios, complexity is $O(n \log n)$ while it can be in worst-case scenarios. The average complexity is $O(n \log n)$

In [Table](#) we can see the complex comparison of sorting algorithms. For some algorithms, array and slice size will also impact performance, so additional notes are provided for some of them:

Table 2.1: Complexity of sorting algorithms

[Sorting algorithms in Go](#)

The following functions from the sort package can be used for sorting:

```
func Ints(x []int)
```

Sorts a slice of ints in ascending order.

```
func Float64s(x []float64)
```

Sorts a slice of floats in ascending order.

```
func Strings(x []string)
```

Sorts a slice of strings in ascending order.

The following code example will sort a slice of

```
array = []int{18, 1, 5, 27, 8, 25, 9, 21}
```

```
sort.Ints(array)
```

Previously presented functions are wrappers around the Sort() function. This function will sort anything that implements sort.Interface interface. Sorting will be determined by the Less() method. The Ints() function is implemented as shown:

```
func Ints(x []int) { Sort(IntSlice(x)) }
```

To implement the data type must implement the following methods:

Len() int:

Returns the number of elements in a collection.

Less(i, j int) bool:

Determines if the element with index i must be placed before the element with index Elements are considered equal if Less(i, j) and Less(j, i) are false.

Swap(i, j int):

Swaps the elements with indices i and

To round a whole story, we will present a definition of the

```
type Interface interface {
```

```
    Len() int
```

```
    Less(i, j int) bool
```

```
    Swap(i, j int)
```

```
}
```

Data type IntSlice used in Ints() function implements sort.Interface in the following way:

```
type IntSlice []int
```

```
func (x IntSlice) Len() int      { return len(x) }
```

```
func (x IntSlice) Less(i, j int) bool { return x[i] < x[j] }
```

```
func (x IntSlice) Swap(i, j int)    { x[i], x[j] = x[j], x[i] }
```

If we want to keep the original order of equal elements, we should use the `Stable()` function from the `sort` package instead of `The`. The presented functions implement an optimized version of the quick sort algorithm with the complexity of $O(n \log n)$ in the worst case.

Conclusion

This chapter introduced our first data structures, arrays, and slices (as a particular type of an array), and basic operations. We learned how to work with them in the Go programming language and how to use and implement searching and sorting algorithms.

The following chapter will present different kinds of linear structures called

Points to remember

Arrays are arranged data structures; each element has a known position.

Arrays use sequential representation for memory storage.

Slices represent a pointer to an underlying array.

A method can use a pointer receiver to change the variable that calls that method.

The type can implement an interface by implementing all its methods.

There are two possible search results: data is found (successful), and data is not found (unsuccessful).

The `Search()` function from package `sort` uses binary search to find the smallest index for a specified value in a sorted array (slice).

Sorting functions from the sort package implementing an optimized version of the quick sort algorithm.

Multiple choice questions

Which bound can be omitted from the slice definition?

Low

High

No one

Both

What is the default value for high bound?

There is no default value.

0

Length of slice.

Length of the underlying array.

Where in slice will function `append()` add an element?

At the end.

At the beginning.

In the middle.

Nowhere.

Which algorithm can have complexity $O(n \log n)$?

Selection sort.

Quick sort.

Bubble sort.

Insertion sort.

Which statement will declare a square matrix?

```
var matrix [2][7]int
```

```
var matrix [7][2]int
```

```
var matrix [5][5]int
```

```
var matrix [1][8]int
```

Answers

d

c

a

b

c

Questions

Which operations can be performed on an array?

What are slice length and capacity?

How does sequential search work?

How does binary search work?

How does insertion sort work?

How does selection sort work?

How does bubble sort work?

How does quick sort work?

How can matrices be stored in memory?

Key terms

A linear and homogenous data structure that contains a finite number of elements.

Value that defines the position of an array element.

Flexible array with dynamic size.

A function declared on a type.

Set of method declarations.

Locating the desired data in the data set based on identification.

Rearranging a set of data in a specified order.

Two-dimensional array.

Lists

C
HAPTER
3

Introduction

You will be introduced to lists in this chapter. First, basic terms and concepts will be presented; we will see different types of lists and operations that can be performed on them. The second part of the chapter will explain how lists are implemented in Go through the standard library and how single-linked lists can be implemented from scratch. Ultimately, we will compare and explain the similarities and differences between lists and arrays.

Structure

The chapter covers the following topics:

Lists

Types of lists

Operations

Implementation of single-linked lists

Lists versus arrays

Lists in Go

Objectives

By the end of the chapter, the reader will be familiar with the basic concepts of the list. You will be able to decide in which situations the list is a better solution than an array (slice) and learn how to implement data structure with the Go programming language.

Lists

Lists are linear collections where elements are connected with pointers. The list element is called a node and consists of data and pointers. The number of pointers depends on the list type. We will discuss list types in the next section.

The list is usually accessed via a pointer that points to the first node. That pointer itself is not a part of the list. The first node in the list is often called the head while the last node is called the tail.

If the order of nodes is essential (values must be sorted in ascending order, for example), the list is ordered. Otherwise, it is unordered.

Lists use linked representation for memory storage. This means that neighboring nodes are not placed in sequential memory locations, and they can be spread across memory space.

Types of lists

Lists are categorized into the following types, depending on the way nodes are connected:

Single-linked list Each node contains a pointer to the next node in the list.

Circular list The last node points to the first node, known as the ring.

Double-linked list Each node contains two pointers. The first points to the next node, and the second points to the previous node.

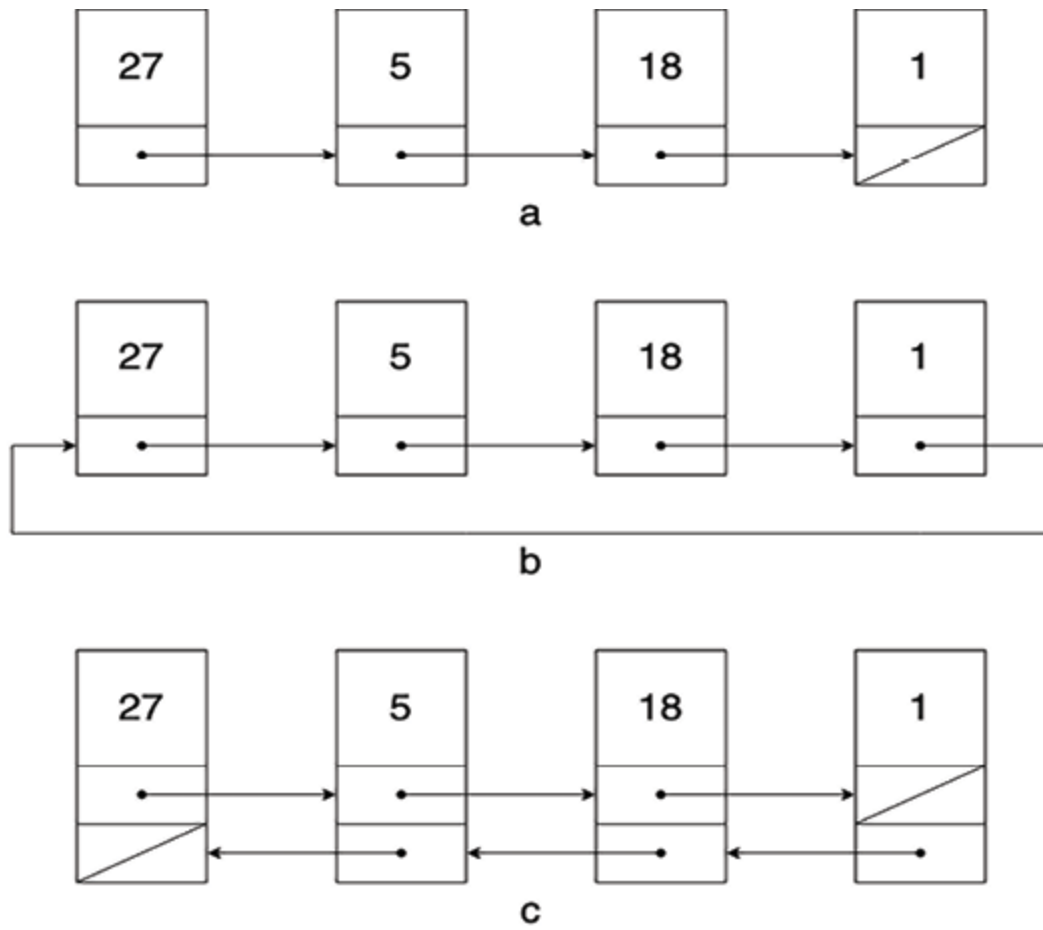


Figure 3.1: Types of lists

Operations

All list operations can be reduced to pointer manipulations; we will see why in this section. The following operations can be performed on lists:

Inserting a node

Removing node

Searching

Concatenation

The insert operation is different for ordered and unordered lists. We can insert a node at any place in the unordered list. Usually, it will be inserted at the beginning of the list, but other approaches are also valid. We will explain this operation for a single-linked list.

In the first step a new node is created. The list pointer will usually point to the first node of the list. The inserted node's next pointer should be set to point to the same node as the list pointer. In the last step, the list pointer will point to the inserted node

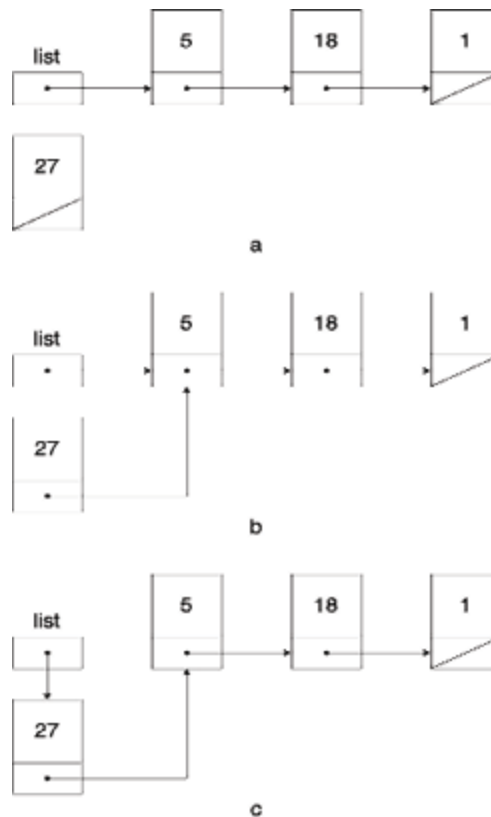


Figure 3.2: Inserting node to the unordered list

An additional pointer, temp, must be introduced to insert a node in the ordered list. We will iterate through

nodes until we find a node, after which a new node should be inserted. The pointer of the new node will be set to point to the next node from the one temp points to. As a last step, the pointer of the node that temp points to will be set to point to the new node as shown:

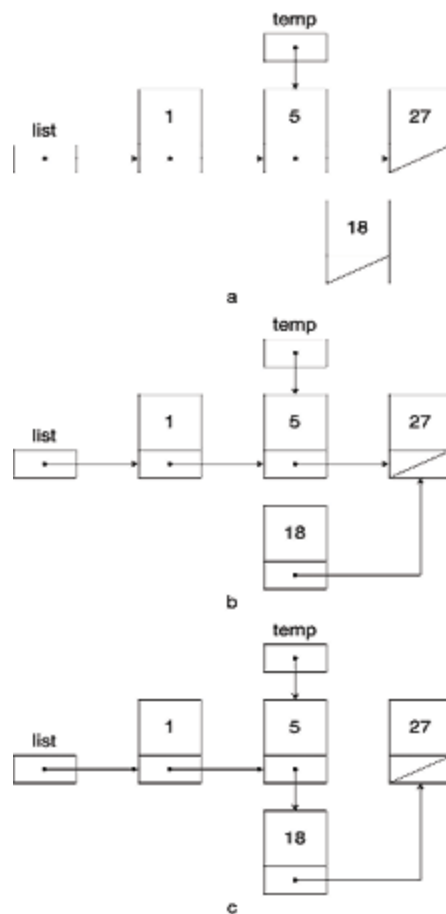


Figure 3.3: Inserting node to the ordered list

To remove a node, we will again use a temp pointer. This pointer will now reference the node that precedes the

one being deleted The pointer of the node, referenced with temp, should be set to point to the node next to the node being deleted Since the removed node is not referenced, it will be picked up and cleaned by a garbage collector in most modern programming languages:

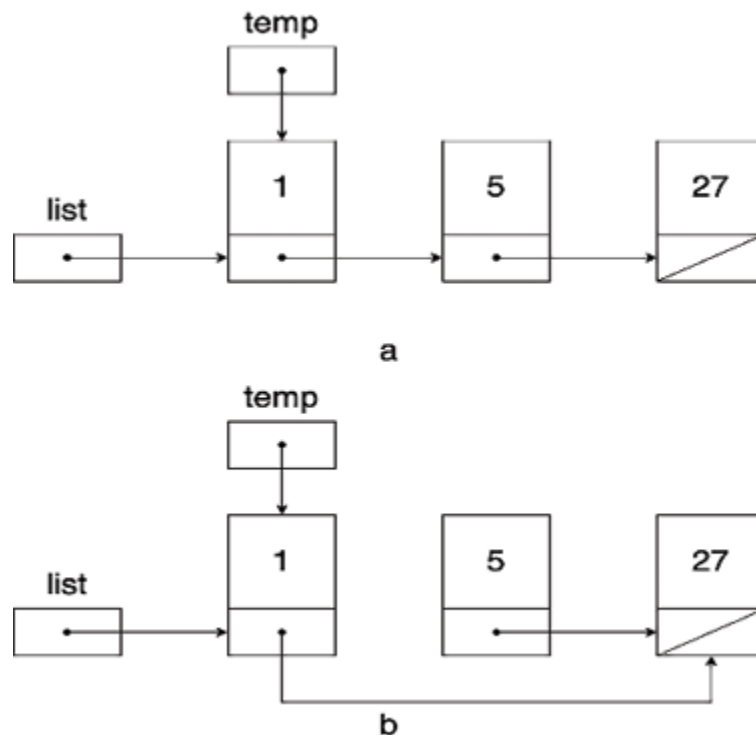


Figure 3.4: Removing of a node

The garbage collector was created and introduced by the American computer scientist John around 1959. It will run parallel with the main program, search for memory spaces occupied by unused objects (known as

and free it. This will relieve developers from manual memory management.

To be sure and avoid unnecessary memory leaks, we can use one more pointer that will reference the node that will be removed to set the pointer of the removed node to nil and properly detach it from the list.

We can use pointers to iterate through nodes to find a specified node. The search operation is easy with ordered lists. An unsuccessful search can be detected earlier.

Two lists can be concatenated by setting the last node of the first list to reference the first node of the second list.

For other types of lists, operations are implemented in the same way. We only have to take into account more pointers for double-linked lists.

Implementation of single-linked list

We will start implementing a single-linked list with a declaration of the struct representing the node. This struct will have two fields, one representing the pointer to the next node and the second representing stored value. We will implement a list that holds integers:

```
type Node struct {  
  
    next *Node  
  
    value int  
  
}
```

The struct that represents the list will contain a pointer to the first node in the list (head) and an additional integer value representing a number of nodes:

```
type List struct {
```

```
head *Node
```

```
len int
```

```
}
```

The first function that we will implement is the function `New()`. This simple function will initialize and return an empty list. The pointer `head` will be set to `nil`, and the length will, logically, be set to 0.

```
func New() List {
```

```
    return List{
```

```
        head: nil,
```

```
        len: 0,
```

```
    }
```

```
}
```

As mentioned, the insert operation differs depending on whether the list is ordered or unordered.

Implementations for both approaches will be provided.

Method `Insert()` will insert a new node at the beginning of the list. First, the new node will be created, and then pointers will be set to reference proper nodes. If the list is empty (head pointer is `nil`) we just set head to point to the new node and increment value for `len`. When the list is not empty, we will set the node's next pointer to point to the current head of the list and then execute the same statements as in the previous case, as shown:

```
func (l *List) Insert(v int) {
```

```
    node := Node{
```

```
        next: nil,
```

```
        value: v,
```

```
    }
```

```
    if l.head != nil {
```

```
        node.next = l.head
```

```
}
```

```
l.head = &node
```

```
l.len++
```

```
}
```

If the insert has to be executed on the ordered list, the procedure is slightly different, as we illustrated and demonstrated before. It is good practice to handle specific (edge) cases first, in our case, a situation when the list is empty. If the list is not empty, we must iterate through the list until we find a node, after which a new node will be inserted, perform a set of pointer manipulation statements, and increment the number of nodes in the list:

```
func (l *List) InsertOrdered(v int) {
```

```
    node := Node{
```

```
        next: nil,
```

```
    value: v,  
  
  }  
  
  if l.head == nil {  
  
    l.head = &node  
  
    l.len++  
  
    return  
  
  }  
  
  temp := l.head  
  
  for temp.next != nil && temp.next.value < v {  
  
    temp = temp.next  
  
  }  
  
  node.next = temp.next
```



```
temp.next = &node
```

```
l.len++
```

```
}
```

When removing a node from the list, we have two edge cases that we must check. If the list is empty, there is no use in trying to locate a node with the specified value, so we return from the method. The second case is when the list contains only one node. Here, we just set the head pointer to nil and decrease the number of nodes.

A number of nodes are used to check if the list contains only one node, but we can archive this with the pointer. If the list contains only one node, the next pointer of a node that is referenced with the head will be set to nil.

In all other cases, we locate a node that precedes one that will be deleted. Here, we will use one additional pointer (node) to properly unlink a node from the list. Ultimately, the value representing the number of nodes will be decreased:

```
func (l *List) Remove(v int) {  
  
    if l.head == nil {  
  
        return  
  
    }  
  
    if l.len == 1 {  
  
        l.head = nil  
  
        l.len--  
  
        return  
  
    }  
  
    for temp := l.head; temp != nil; temp = temp.next {  
  
        if temp.next.value == v {  
  
            node := temp.next
```

```

temp.next = node.next

node.next = nil

l.len--

return

}

}

}

```

Method Find() will search for a node with the specified value and return a pointer to that node in case of a successful search; otherwise returns nil. The list will be iterated from the head node till there are no more nodes:

```

func (l *List) Find(v int) *Node {

    for temp := l.head; temp != nil; temp = temp.next {

```

```

    if temp.value == v {

        return temp

    }

}

return nil

}

```

To concatenate two lists, we must move to the end of the first one and set its last node to point to the second one's first node (head). The head pointer of the second list can be set to nil when concatenation is completed:

```

func (l *List) Concatenate(l2 List) {

    temp := l.head

    for temp.next != nil {

        temp = temp.next
    }
}

```

```
}
```

```
temp.next = l2.head
```

```
l2.head = nil
```

```
}
```

Having a method or function that will print data structure in some human-readable format is a good practice. This method will iterate through the list and print node values:

```
func (l *List) Print() {
```

```
    fmt.Print("[")
```

```
    for temp := l.head; temp != nil; temp = temp.next {
```

```
        if temp.next != nil {
```

```
            fmt.Printf("%v, ", temp.value)
```

```

    } else {

        fmt.Print(temp.value)

    }

}

fmt.Println("]")

}

```

The previous method will display a list that contains nodes with values and 27 in the following way:

```
[5, 8, 25, 27]
```

In the end, we can provide two simple getter methods, one that returns the head pointer and the second that returns a number of nodes:

```
func (l *List) Len() int {
```

```
    return l.len  
  
}  
  
func (l *List) Head() *Node {  
  
    return l.head  
  
}
```

Here is an example of the main() function where all previously described functions and methods are utilized (list implementation is placed inside the linkedlist package):

```
func main() {  
  
    l1 := linkedlist.New()  
  
    l1.Insert(1)  
  
    l1.Insert(18)  
  
    l1.Insert(9)
```

l1.Insert(21)

l1.Print()

l1.Remove(9)

l1.Print()

l2 := linkedlist.New()

l2.InsertOrdered(5)

l2.InsertOrdered(27)

l2.InsertOrdered(25)

l2.InsertOrdered(8)

l2.Print()

n1 := l1.Find(18)


```
fmt.Println(n1)
```

```
n2 := l1.Find(27)
```

```
fmt.Println(n2)
```

```
l1.Concatenate(l2)
```

```
l1.Print()
```

```
}
```

Initially, list l1 is unordered, while list l2 is ordered, but at the end, they are concatenated in an unordered list.

Lists versus arrays

This section will compare lists and arrays and explain why each data structure is better for specific uses.

In terms of memory usage, an individual node of the list takes up more space than one element of an array because additional space must be allocated for pointers. If we look at the bigger picture, the array allocates the whole capacity, while the capacity of the list will be gradually added, depending on actual needs.

It is much faster to access specific elements of an array; the index is all we need. To access a particular list node, we must iterate through nodes until we find a required one.

Adding and removing operations are more accessible with lists. To insert the element in an array on a specific position, all elements, starting from that position, must be shifted to the right for one place. When the shift is completed, we can insert a new element in the specified position. We should also consider capacity; if an array is

full, the last element will be dropped, or the array must be reallocated for additional capacity. To insert a new node in the list, we manipulate pointer(s), as explained previously.

To remove an element from the array, we must shift elements to the left for one place, starting from the position next to the removed element to the end of an array. To remove a node from the list, we need to disconnect it from the list.

An array is the most suitable data structure for situations when capacity is known in advance, and fast access to elements is essential. A list is the better option if we have many insert/remove operations at random places.

Lists in Go

Go provides implementations for double-linked and circular lists through the standard library.

Double-linked list

Implementation of a double-linked list is provided through a package container/list. The struct that represents lists contains two fields and a root node (named element in this implementation), that is not part of the list and list length:

```
type List struct {  
  
    root Element  
  
    len int  
  
}
```

The struct that represents the node contains four fields. The four fields are: pointer to the previous and next node, the value stored in the node, and a pointer to the list to which the node belongs:

```
type Element struct {
```

next, prev *Element

list *List

Value any

}

Method New() will create an empty list ready for use.
The following statement will create a new list:

l := list.New()

We have four options for insertion:

Insert a new node with the provided value at the front of the list:

head := l.PushFront(5)

Insert a new node with the provided value at the end of the list:

tail := l.PushBack(27)

Insert a new node with the provided value before the specified node:

```
l.InsertBefore(1, tail)
```

Insert new node with the provided value after specified node:

```
l.InsertAfter(18, head)
```

Each of these methods will return the inserted node.

Method Remove() will remove the specified node (tail node in our example):

```
l.Remove(tail)
```

Method Len() returns the current length of the list:

```
l.Len()
```

To iterate through the list, we can use the next and previous pointer of nodes. With methods Front() and we

can get the first and last node from the list, respectively. The following loop will iterate through the list from the first to the last node:

```
for node := l.Front(); node != nil; node = node.Next() {  
  
    fmt.Print(node.Value)  
  
}
```

We can also iterate from the last to the first node:

```
for node := l.Back(); node != nil; node = node.Prev() {  
  
    fmt.Print(node.Value)  
  
}
```


Circular list

The circular list implementation is in the container/ring package from Go standard library. The struct that represents a circular list (or ring as it is called in Go) contains three fields, pointers to the previous and next node, and value:

```
type Ring struct {  
  
    next, prev *Ring  
  
    Value    any  
  
}
```

Because a circular list does not have an actual beginning and end, a pointer to any node can be used to reference the whole list. The default (zero) value for the circular list is the one-node list with the value set to nil.

Function `New()` will create a circular list and return a pointer to the created list. This function accepts an integer value representing the number of nodes as an argument. The following statement will create a circular list of 4 nodes:

```
r := ring.New(4)
```

Similarly with other data structures shown so far, method `Len()` will return a number of nodes:

```
r.Len()
```

Methods `Next()` and `Prev()` will return the next and previous node, respectively. These methods are often used to iterate through circular lists. In this example, we use the method `Next()` in combination with the for loop to initialize the values of nodes:

```
for i := 0; i < r.Len(); i++ {
```

```
    r.Value = i
```

```
    r = r.Next()
```

```
}
```

Method `Link()` concatenates two circular lists and returns a pointer referencing a new list. The following example will concatenate two lists:

```
r3 := r1.Link(r2)
```

The last interesting method from this package is `Do()`. It will call function provided as an argument, on each node of the circular list. The following example will print the list content:

```
r.Do(func(p any) {  
  
    fmt.Println(p.(int))  
  
})
```

As you probably noticed, there is no implementation for a single-linked list. In the following section, we will implement it ourselves.

Conclusion

In this chapter, we were introduced to different types of lists and operations that can be performed on them. For some list types, we have Go implementation; for some (single-linked list), we implement it by ourselves. We also presented differences between lists and arrays and provided hints on when to use one and when to use the other.

The next chapter will introduce two linear and dynamic data structures with specific rules for accessing elements, stacks, and queues.

Points to remember

The number of node's pointers depends on the list type.

The list is usually accessed via a pointer referencing the first node. That pointer is not a part of the list.

Lists use linked representation for memory storage.

Most operations are different for unordered and ordered lists. For example, we can insert a new node in an unordered list anywhere, but it is much faster to declare an unsuccessful search for ordered lists.

Multiple choice questions

What is the first node in the list called?

Start

Head

Front

First

What is the last node in the list called?

Tail

Back

End

Rear

What is the other name for the circular list?

Loop

Ring

Circuit

Disc

For which types of lists Go provides implementation through the standard library?

Single-linked and double-linked lists.

Single-linked and circular lists.

Double-linked and circular lists.

Go has no list implementations.

Answers

b

a

b

c

Questions

What are the differences between ordered and unordered lists?

What types of lists do we have?

Which operations can be performed on the list?

In which scenarios list is better than an array?

Key terms

Linear collections where elements are connected with pointers.

Element of the list.

Stack and Queue

Introduction

This chapter will introduce two new data structures: and
The first half of the chapter will cover stack. You will be presented with basic stack concepts, with operations that can be performed and how it is utilized in Go. The second half of the chapter will cover queues. Fundamentals of queues and operations will be presented, as well as their implementation with Go. At the end of the chapter, the priority queue will be covered.

Structure

The chapter covers the following topics:

Stack

Operations

Stack in Go

Stack implementation

Queue

Operations

Queue implementation

Priority queue

Priority queue in Go

Objectives

After reading this chapter, you will be familiar with stacks and queues and understand where and how they can be used. As with some previous chapters, you will learn how to implement data structures with Go or use existing solutions from the standard library.

Stack

A stack is a linear data structure with specific rules for accessing elements. The element can be added or removed only from the top of the stack. The top of the stack is referenced with a special pointer called Stack Pointer

In [Figure](#) we can see an illustration of a stack. Stack uses the Last First Out method for data organization. This means that the last element added to the stack (element with value which is now on the top, is the first one that can be removed from the stack. We must remove all the above elements to remove an element with value 27 (the first added to the stack):

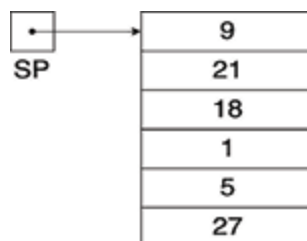


Figure 4.1: Stack

The end of the stack opposite the top is called the bottom. The stack can grow downward or upward relative to the bottom. In real life, this means that the stack will take the previous or next memory location close to the top of the stack to add a new element:

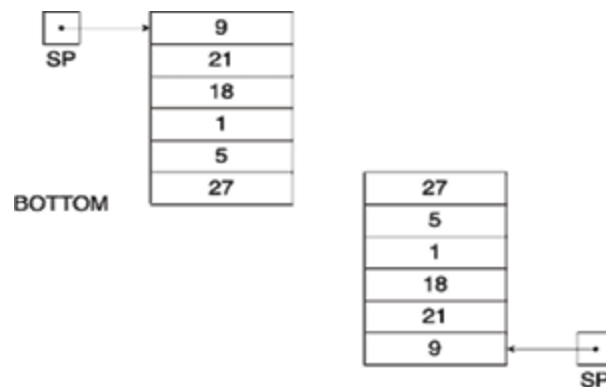


Figure 4.2: Stack growth

Stack is a dynamic data structure. It grows and shrinks depending on needs. Related to the types of stored elements, there are no rules and restrictions on what can be stored in the stack. Usually, elements of the same type are stored, so the stack is a homogeneous data structure.

Operations

Two operations can be performed on the stack:

Adding an element to the stack (push).

Removing an element from the stack (pop).

In [Figure](#) the push operation is illustrated. Element with value 25 has been added to the top of the stack, and the stack pointer is moved to reference it, as shown:

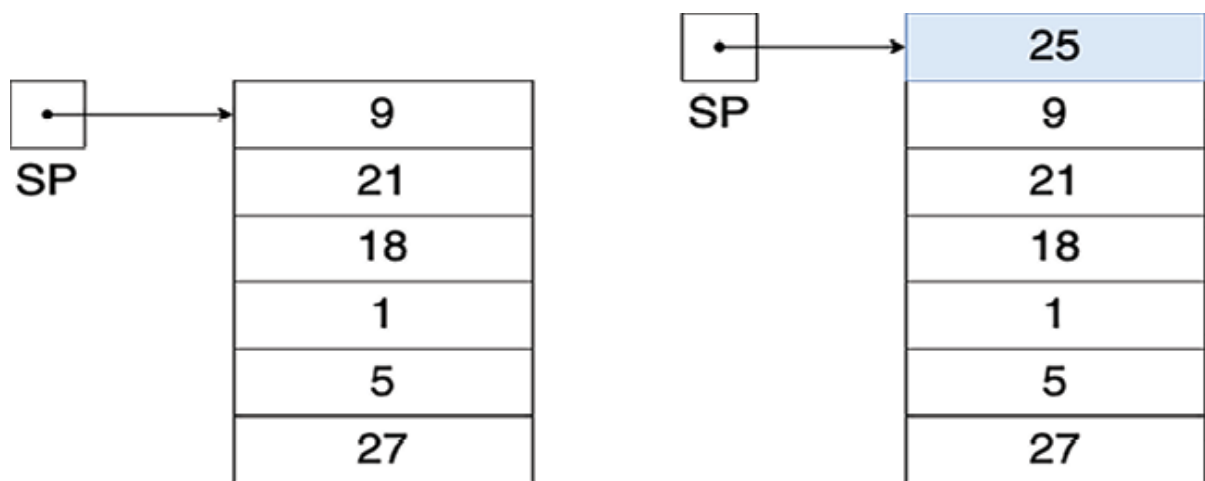


Figure 4.3: Push operation

This can be described with the following code:

```
func Push(v int) {  
  
    sp = sp + 1  
  
    stack[sp] = v  
  
}
```

To perform operation illustrated in [Figure](#) value 25 must be passed to the function.

Before attempting to pop an element from the stack, it should be checked if it is empty. Pop from an empty stack is an irregular operation. In [Figure](#) we can see an illustration of the pop operation. Element with value 25 is removed from the top of the stack, and the stack pointer now points to the element below:

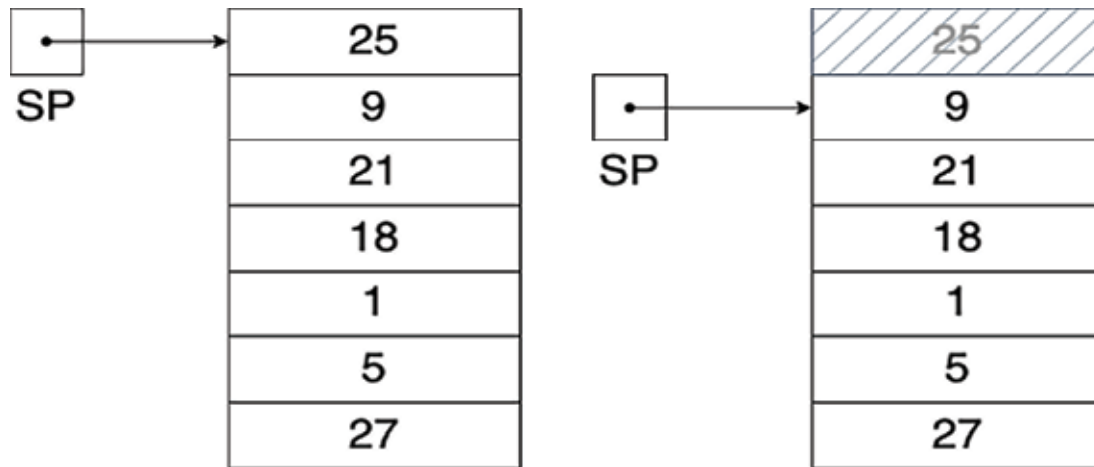


Figure 4.4: Pop operation

In reality, we only move the stack pointer to the lower element. Element 25 is still in physical memory but no longer referenced. If we push a new element, element 25 will be overridden with a new element.

The pop can be described with the following code:

```
func Pop() int {
```

```
    if sp == 1 {
```

```
        return -1
```

```
    }
```

```
    v := stack[sp]
```

```
sp = sp - 1
```

```
return v
```

```
}
```

Since pop from the empty stack is an irregular operation, an irregular value can be returned. Previous functions assume that variables stack and sp are defined outside functions.

Stack in Go

Go has no exposed stack implementation that can be used for solution development. However, Go has an internal stack implementation used for defer statements.

The defer statement delays the execution of the function until the surrounding function returns. Arguments of deferred functions will be calculated, but the function will be executed only when the surrounding function returns. In the following example, the execution of `fmt.Println("a")` will be delayed until the `main()` function finishes, so `ba` will be printed as the standard output:

```
func main() {  
  
    defer fmt.Println("a")  
  
    fmt.Println("b")  
  
}
```

Calls of deferred functions are pushed to stack. When the surrounding function returns, function calls will be popped from the stack and executed in LIFO order. The following code sample will print 1234 as standard output:

```
func main() {  
  
    defer fmt.Print(4)  
  
    defer fmt.Print(3)  
  
    defer fmt.Print(2)  
  
    defer fmt.Print(1)  
  
}
```

[Figure 4.5](#) shows a stack of deferred function before the surrounding main() function from the previous code sample returns:

<code>fmt.Println(1)</code>
<code>fmt.Println(2)</code>
<code>fmt.Println(3)</code>
<code>fmt.Println(4)</code>

Figure 4.5: Deferred stack

Stack implementation

Stack is similar to arrays with a specific approach, so we can use an array (slice) to implement it. The end of the slice will logically represent the top of the stack, while the highest index will represent the stack pointer. The stack struct will have two fields: a slice representing the stack and an integer representing the stack pointer:

```
type Stack struct {  
  
    stack    []int  
  
    stackPointer int  
  
}
```

We can use only slice and calculate the highest index with the `len()` function, but we will include a stack pointer to follow the school example of the stack. Elements of the slice will start from index 0, so the stack pointer will always have a value of one less than the

length of the slice. If the stack pointer has value -1, the stack is empty.

Method Push() will use the append() function to add an element to the end of the slice (top of stack) and update the stack pointer, as shown:

```
func (s *Stack) Push(v int) {  
  
    s.stack = append(s.stack, v)  
  
    s.stackPointer = len(s.stack) - 1  
  
}
```

Method Pop() will check if the stack is empty and return an invalid value. If the stack is not empty, the top element will be sliced off and returned. In the end, the value of the stack pointer is updated:

```
func (s *Stack) Pop() int {  
  
    if s.stackPointer == -1 {
```

```
    return -1

}

element := s.stack[s.stackPointer]

s.stack = s.stack[:s.stackPointer]

s.stackPointer--

return element

}
```

Here is a small example of code that uses this stack implementation. In the first part of the main() function, values and 18 are pushed to stack, while in the second part, these values are popped from it. The last pop operation will return an invalid value because the stack is empty at that moment:

```
func main() {

    var s stack.Stack
```

```
s.Push(27)
```

```
s.Push(5)
```

```
s.Push(1)
```

```
s.Push(18)
```

```
fmt.Println(s)
```

```
fmt.Println(s.Pop())
```

```
fmt.Println(s.Pop())
```

```
fmt.Println(s.Pop())
```

```
fmt.Println(s.Pop())
```

```
fmt.Println(s.Pop())
```

```
fmt.Println(s)
```

```
}
```

Queue

Like a stack, the queue is a linear data structure with specific rules for accessing elements. Each queue has two access ends. Elements are inserted at the beginning (front) and removed from the end of the queue (rear):

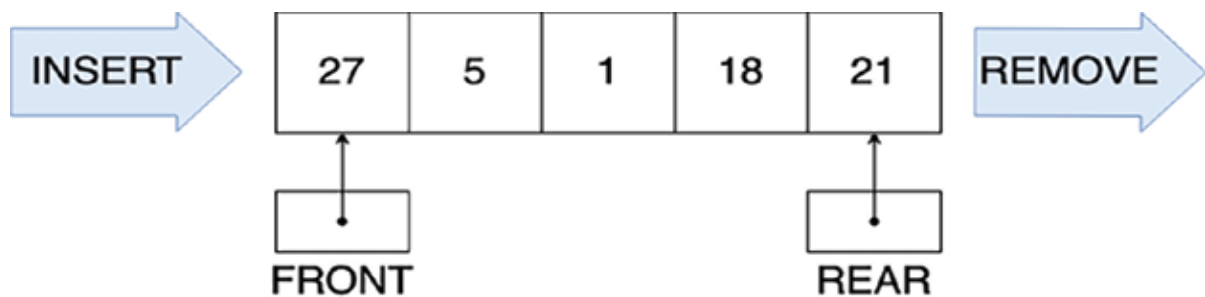


Figure 4.6: Queue

Queue uses the First First Out order. The first element added to the queue will be removed (element with value 21 in [Figure](#)). Like a stack, the queue is dynamic and (usually) a homogenous data structure.

There is a particular type of queue, a double-ended queue, where elements can be inserted and removed on both ends.

Operations

Two operations can be performed with queue:

Inserting element to queue (enqueue).

Removing an element from the queue (dequeue).

An example of an enqueue operation is illustrated in [Figure](#). An element with value 9 is added at the beginning of the queue, and the front now references a new element:

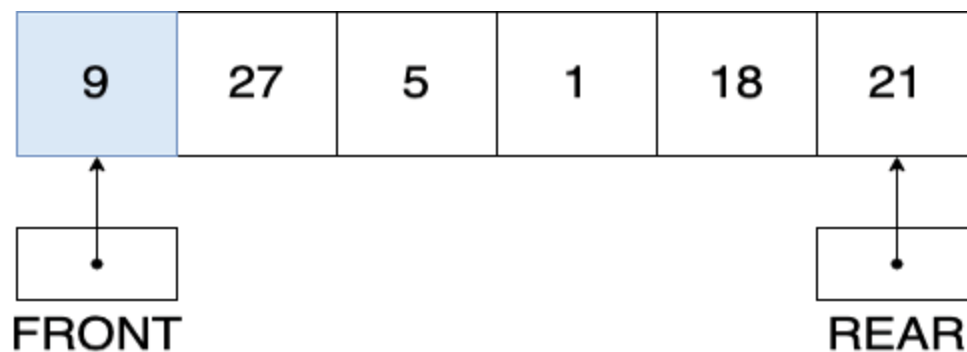


Figure 4.7: Enqueue operation

Enqueue operation can be described with the following code segment:

```
func Enqueue(v int) {  
  
    front = front - 1  
  
    queue[front] = v  
  
}
```

Dequeue operation can only be executed on a non-empty queue. Dequeuing from the empty queue is an invalid operation, so this condition must be checked. In [Figure](#) the element with value 21 is removed from the end of the queue, and the rear now references the element with value

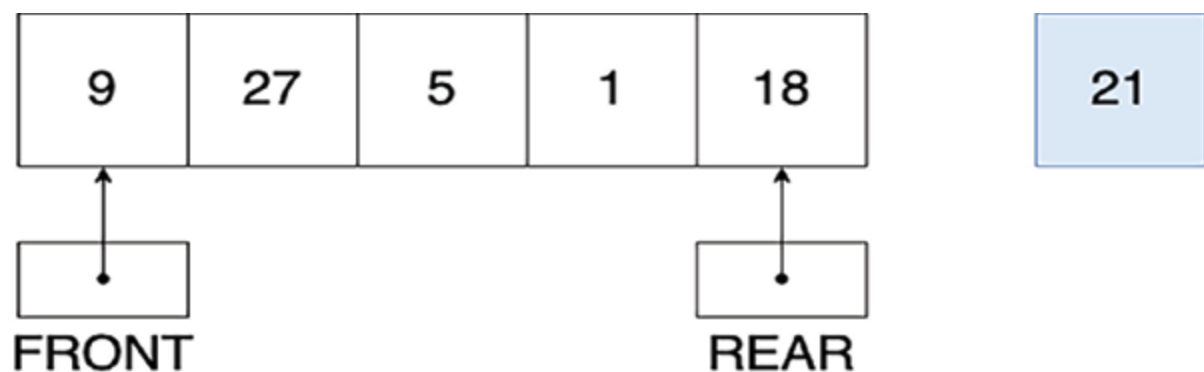


Figure 4.8: Dequeue operation

Enqueue operation can be described with the following code segment (the queue is empty if the front and rear have the same value):

```
func Dequeue() int {  
  
    if front == rear {  
  
        return -1  
  
    }  
  
    v := queue[rear]  
  
    rear = rear - 1  
  
    return v  
  
}
```

Previous functions are provided just for the illustration. There is no check of boundaries, and it is assumed that variable queue, front, and rear are defined outside functions.

Queue implementation

There is no queue implementation in Go, but it is easy to implement with existing or custom data structures. It is possible to implement it with a slice in the following way: a new element will be appended at the end, while the first element (with index 0) will be sliced off during the dequeue operation. This implementation can be confusing because the beginning of the slice is queue rear while the end is queue front.

We can also use the double-linked list from the container/list package. This list already contains pointers to the beginning and end of the list, so we need just a couple of adjustments. In a nutshell, a struct that represents a queue will have a pointer to list:

```
type Queue struct {  
  
    queue *list.List  
  
}
```

Function New() will create an empty queue and return a pointer to it:

```
func New() *Queue {  
  
    return &Queue{queue: list.New()}  
  
}
```

Method Enqueue() will use PushFront() method from the list to add an element to the beginning of the queue, as shown:

```
func (q *Queue) Enqueue(v int) {  
  
    q.queue.PushFront(v)  
  
}
```

Inside the Dequeue() method, we will check if the queue is empty and return an invalid value in that case. The element will be fetched from the rear (back) of the queue and removed from it. The method returns the value stored in the element:

```
func (q *Queue) Dequeue() int {  
  
    if q.queue.Len() == 0 {  
  
        return -1  
  
    }  
  
    element := q.queue.Back()  
  
    q.queue.Remove(element)  
  
    return element.Value.(int)  
  
}
```

We can additionally have the Print() method, similar to the one used for lists, as shown in the following code snippet:

```
func (q *Queue) Print() {  
  
    fmt.Print("[")
```

```

for e := q.queue.Front(); e != nil; e = e.Next() {

    if e.Next() != nil {

        fmt.Printf("%v, ", e.Value)

    } else {

        fmt.Print(e.Value)

    }

}

fmt.Println("]")

}

```

The following code example uses our queue implementation. In this example, values and 18 are enqueued and dequeued from the queue. The last dequeue will return an invalid value because the queue will be empty at that moment:

```
func main() {  
  
    q := queue.New()  
  
    q.Enqueue(27)  
  
    q.Enqueue(5)  
  
    q.Enqueue(1)  
  
    q.Enqueue(18)  
  
    q.Print()  
  
    fmt.Println(q.Dequeue())  
  
    fmt.Println(q.Dequeue())  
  
    fmt.Println(q.Dequeue())  
  
    fmt.Println(q.Dequeue())  
}
```

```
fmt.Println(q.Dequeue())
```

```
q.Print()
```

```
}
```

Priority_queue

For queues in previous sections, the order of dequeued elements depends exclusively on the order of arrival (enqueue) of the elements. That is not the case with priority queues. Here, we will remove elements by priority.

Priority can be defined as the value assigned to each element that will be used to determine which element to remove from the queue. It can be the value already stored in the element and does not have to be some special value assigned to an element. [Figure 4.9](#) shows an example where the element with the highest value (priority) was dequeued:

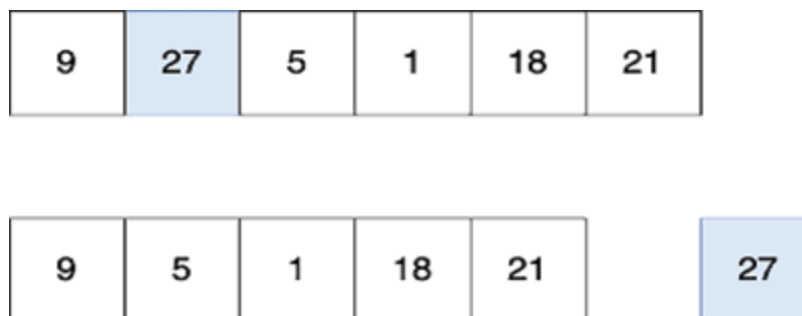


Figure 4.9: Dequeuing from priority queue

There are two types of priority queues:

Min priority The element with the smallest priority value will be dequeued first.

Max priority Opposite from the previous, the element with the highest priority value will be dequeued first.

Order of enqueue is essential only for the elements with the same priority value. If more elements have the same priority value, the one that arrived first will be the dequeued first.

[Priority_queue in Go](#)

Go does not have implementation for priority queue but provides heap operations for any type that implements interface from the container.heap package. We will discuss more about heap in [Chapter](#) For now, it is enough to know that heap is the most common way to implement a priority queue.

To implement a heap the data type must implement the following:

Push(x Adds x as a new element

Removes and returns an element

Methods from sort.Interface

Methods Push() and Pop() will be used for enqueue and dequeue operations, respectively. We have already discussed sort.Interface in [Chapter](#) Here is a declaration of the heap interface:

```
type Interface interface {  
  
    sort.Interface  
  
    Push(x any)  
  
    Pop() any  
  
}
```

We will implement a min priority queue, where integer values will be stored. There will be no additional values for priority; stored integer value will be used for that purpose, so the struct that represents the element will have one field:

```
type Element struct {  
  
    value int  
  
}
```

The priority queue will be represented as a slice of elements:

```
type PriorityQueue []Element
```

Method Len() is simple. It returns the length of the slice that represents the priority queue:

```
func (pq PriorityQueue) Len() int {  
  
    return len(pq)  
  
}
```

The second method, determines if the element with index i must be placed before the element with index j. This function will be used to determine priority, so we need to check if the value of the element with index i is less than that of the element with index j as shown:

```
func (pq PriorityQueue) Less(i, j int) bool {  
  
    return pq[i].value < pq[j].value  
  
}
```

For the max priority queue, the operator greater than is used instead of the operator less than. This can be confusing because the method name and condition are contradictory, but it will produce the desired behavior.

A third method, will swap places of elements with indices i and To avoid index out-of-bound errors, it should be checked if the queue is empty before attempting to change the position of elements:

```
func (pq PriorityQueue) Swap(i, j int) {  
  
    if pq.Len() == 0 {  
  
        return  
  
    }  
  
    pq[i], pq[j] = pq[j], pq[i]  
  
}
```

The method which will be used for enqueueing, creates a new element and appends it to the slice. Because the method must accept the argument of type any,

conversion to an integer must be performed, as shown in the following example:

```
func (pq *PriorityQueue) Push(v any) {  
  
    element := Element{  
  
        value: v.(int),  
  
    }  
  
    *pq = append(*pq, element)  
  
}
```

The last method, slices off an element from the end of the queue and returns that element if the queue is not empty, as shown:

```
func (pq *PriorityQueue) Pop() any {  
  
    if pq.Len() == 0 {  
  
        return -1
```

```
}  
  
queue := *pq  
  
n := pq.Len() - 1  
  
element := queue[n]  
  
*pq = queue[0:n]  
  
return element  
  
}
```

Something is strange here. The Pop() does not check for priority, so this implements a regular queue. We must use heap.Push() and heap.Pop() methods to get the desired behavior of the min priority queue for enqueueing and dequeuing. These methods can be used only on data types that implement heap interface and will call priority queue methods in the background.

Here is a code sample that shows the usage of priority queue implementation:

```
func main() {  
  
    pq := make(priorityqueue.PriorityQueue, 0)  
  
    heap.Push(&pq, 27)  
  
    heap.Push(&pq, 5)  
  
    heap.Push(&pq, 1)  
  
    heap.Push(&pq, 18)  
  
    fmt.Println(pq)  
  
    fmt.Println(heap.Pop(&pq))  
  
    fmt.Println(heap.Pop(&pq))  
  
    fmt.Println(heap.Pop(&pq))  
  
    fmt.Println(heap.Pop(&pq))  
}
```

```
fmt.Println(heap.Pop(&pq))
```

```
fmt.Println(heap.Pop(&pq))
```

```
fmt.Println(pq)
```

```
}
```


Conclusion

This chapter introduced two linear data structures with specific rules for accessing elements, stack, and queue. We learned all the operations that can be performed on them and all types (for queues). We also learned how to implement stack and queue with the Go programming language.

The following chapter will introduce maps and explain the concept of hashing.

Points to remember

Go has no stack implementation that can be used for solution development. It internally uses the stack for deferring function calls.

We can use the value already stored in the element for priority. It does not have to be some new value assigned to the element.

Heap is the most common way to implement the priority queue.

Multiple choice questions

Which method does the data organization stack use?

First In First Out

First In Last Out

Last In First Out

Last In Last Out

Which operation adds an element to the stack?

Enqueue

Push

Dequeue

Pop

Which method is used by queue for data organization?

First In First Out

First In Last Out

Last In First Out

Last In Last Out

Which operation removes an element from the queue?

Enqueue

Push

Dequeue

Pop

Answers

c

b

a

c

Questions

What is a stack pointer?

How can stack grow?

Which operations can be performed on the stack?

How are the beginning and end of the queue named?

Which operations can be performed in the queue?

What types of priority queues do we have?

Key terms

Linear, dynamic, and (usually) homogenous data structure where elements can be added and removed only from the top.

Linear, dynamic, and (usually) homogenous data structure where elements are inserted at the beginning and removed from the end.

Queue where elements are removed by priority.

Hashing and Maps

Introduction

In this chapter, we will introduce the basic concepts of hashing, explain a hash function, and learn how to handle collisions. The second part of the chapter will cover maps, operations with maps, and using maps in the Go programming language.

Structure

The chapter covers the following topics:

Hashing

Hash function

Hash collision

Maps

Operations

Maps in Go

Objectives

After reading this chapter, the reader will be familiar with the fundamentals of hashing. You will be able to choose a suitable hash function and learn to use maps in some situations instead of, for example, arrays and slices.

Hashing

Hashing is a process of transforming any given key into value. These values are stored in an array (table) with a capacity equal to the number of different keys. We will see why in the following section. Each value has a unique position directly accessed via the key's value as shown:

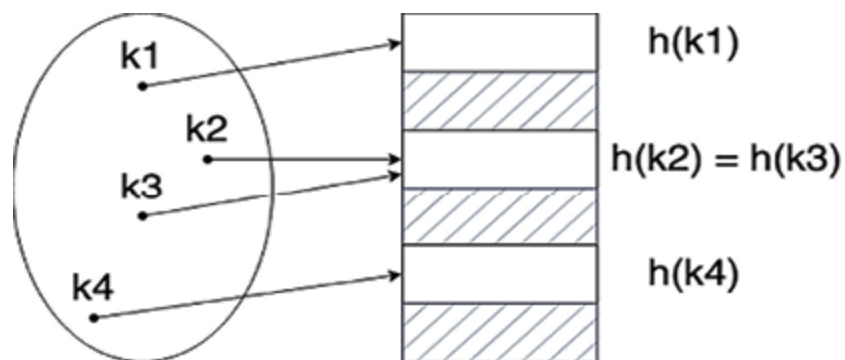


Figure 5.1: Hashing

The function that transforms the key into a number belonging to a table index range is called the hash function (function $h()$ in [Figure](#)). Ideally, each key gives a unique position in the table. A hash table (map) is an array that uses this approach for indexing.

A collision occurs when two or more keys map to the same position. These keys are called In [Figure](#) keys k_2 and k_3 are synonyms. A set of synonyms is known as an equivalence

Hash function

The hash function is a fundamental feature in the hashing process and should meet the following criteria:

To be as simple as possible, to make it easier and faster to calculate because it is performed at every access.

To be as uniform as possible to avoid collisions.

The hash function can be dependent or independent of key distribution. Key distribution is not known in advance for the first ones, but some basic properties of key sets are known. Some of the following methods belong to that group:

Division The result of a hash function is the remainder obtained by dividing the integer key by value n , where n is less than or equal to the size of a hash table. Take a look at the syntax:

$$h(k) = k \bmod n$$

Here is an example of the key with value 27 and table with 10 elements:

$$h(27) = 27 \bmod 10 = 7$$

The collision depends on the choice of value for For example, if n is even, even keys will be mapped to even indices and odd to odd indices. Only half of the indices will be burdened if keys are mostly even.

Multiplication The key is multiplied by a suitably chosen real constant with a value between 0 and 1. The part after the decimal point is taken, multiplied by table size, and converted to an integer to get the index. Here is the syntax:

$$h(k) = \text{mod } 0 < c < 1$$

Mathematica function `floor()` will convert real to integer numbers by returning the greatest integer less or equal to the provided real number. The following example will calculate an index for the key with a value of a table with 10 elements, and a constant of

$$h(27) = \text{floor}(10 * (0.518 * 27 \bmod 1) = \text{floor}(10 * (13.986 \bmod 1)) = \text{floor}(10 * 0.986) = \text{floor}(9.86) = 9$$

It is a good practice to select a constant close to the golden ratio

Mid-square The value of the key is multiplied by itself, and the method takes as many digits as necessary from a fixed position in the middle of the square for indexing the table (array). For example, two digits are needed if the table has 100 slots (indices range from 0 to 99). In [Figure](#) we can see how the index is calculated for a table with 1000 slots (3 digits are needed for the index) and a key with a value of

$$1989 * 1989 = 3956121$$




Figure method

Digit folding The key is divided into parts of the same length corresponding to the number of digits needed to index a table. The last part can be shorter in some situations. The parts add up (fold) to get an index. If the

calculated value exceeds the table size, the index is the remainder of dividing that value by the table size. [Figure 5.3](#) illustrates the digit folding method for the key with a value of 27051989 and a table with 100 slots:

$$\begin{array}{rcl}
 27051989 & \Rightarrow & 27 \\
 & & 05 \\
 & & 19 \\
 & & 89 \\
 \hline
 & & 140 \\
 & & \downarrow \\
 & & 40
 \end{array}$$

Figure folding method

Radix conversion If the key represents a number in a numeral system with radix (base) of p to get an index, this method will treat it like a number in a numeral system with radix (base) of q where q is greater than p . If the resulted number is greater than table size, only the necessary number of digits will be taken from it. The following example will treat a key with a value of 275 in the decimal numeral system (radix $p = 10$) as a number from a numeral system with radix $q = 12$.

$$k = 275, p = 10, q = 12$$

$$2 * 122 + 7 * 121 + 5 * 120 = 288 + 84 + 5 = 377$$

If the length of the table is 100, the two lowest digits (77) can be picked as an index.

Perfect It is a minimal function where collisions cannot occur. It will map n different keys to n unique indices. It is hard to find a perfect hash function.

The key set is known in advance with hash functions dependent on key distribution. The digit analysis (heuristic) method is the most popular method from this group. It will execute the following steps:

The set of numerical representations of all key values is analyzed.

A table is created where, for each digit position in the key, the number of occurrences of certain digits is given for all keys.

As many positions as needed to index the hash table are selected. The columns from the table treated in the

previous step that show the slightest variation in the appearance of different digits are selected.

All hash functions presented in this section (besides perfect hash) can cause a collision. In the following section, we will learn how to resolve these collisions.

Hash collision

As mentioned, a collision occurs when multiple keys are indexed to the same slot in a hash table. When there are more keys than table slots, the simplest way to fix collision is to increase the size of a hash table. More slots provide more outcomes of mapping results. Therefore, there is a lesser chance of a collision.

A metric called load factor expresses how much hash table is filled. It is calculated by dividing the number of elements in the hash table by the size of the table. In [Figure](#) we can see hash tables with various load factors:

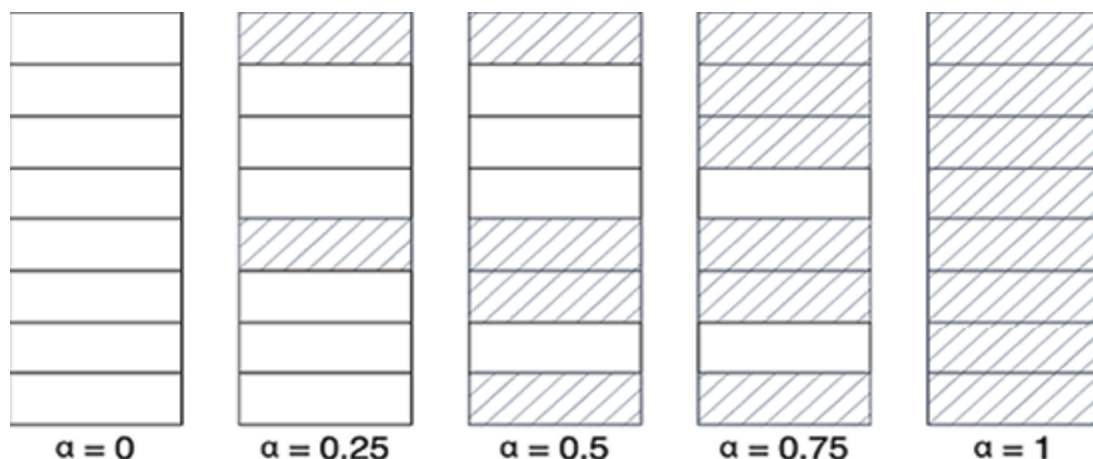


Figure 5.4: Load factor

The two standard methods for collision resolution are open addressing and separate chaining.

With open addressing, when a collision occurs, the calculated address for a given key is occupied, and another address is found for that key. For each key, the probe sequence is generated. The probe sequence represents an array of addresses assigned to the key. When a new entry is inserted, the probe sequence is examined until the empty slot is found.

The probe sequence generation is often called rehashing because the new address is calculated based on address occupancy. We can use one of the following generation methods:

Linear If the index generated by hash function $h(k)$ is occupied, the value will be stored in the next available slot. This approach is represented with the following function:

$$h(k, i) = (h(k) + i) \bmod n$$

Value n represents the hash table size, while i represents the index in the probe sequence. In [Figure](#) we can see an example of how collision is resolved with linear probing when the hash function for keys 27 and 21 returns the same value:

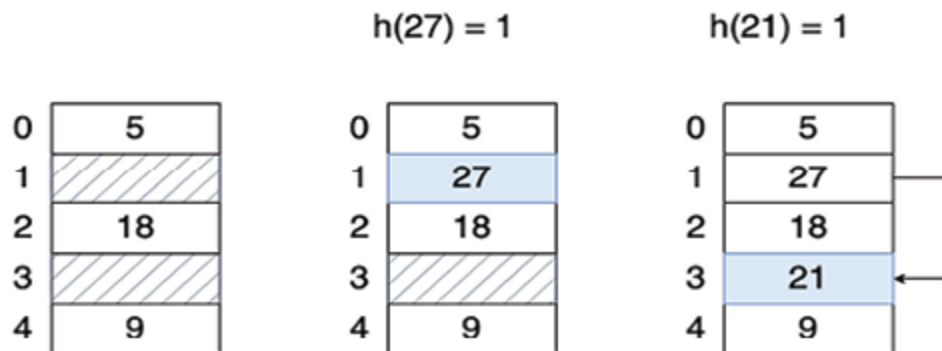


Figure probing

If the search for the empty slots reaches the end of the hash table, it will continue from the beginning.

Random A pseudo-random sequence of numbers from 0 to $n-1$, where n represents the size of the hash table, will be generated and used as a probe sequence.

Quadratic Similar to linear probing, where the interval between probes increases quadratically. This is represented with the following formula:

$$h(k, i) = (h(k) + i) \bmod n$$

Here is an example of a probe sequence for the hash table with 5 slots, where $h(k)$ is equal to 2:

$$h(k, 0) = 2 + 0 = 2$$

$$h(k, 1) = 2 + 1 = 3$$

$$h(k, 2) = 2 + 4 = 6$$

$$h(k, 3) = 2 + 9 = 11$$

$$h(k, 4) = 2 + 16 = 18$$

Double Two independent hash functions, $h_1(k)$ and $h_2(k)$ will be used for the generation of probe sequence in the following way:

$$h(k, i) = (h_1(k) + i * h_2(k)) \bmod n$$

In separated chaining, synonyms are chained into the list. The list headers are in hash table slots. [Figure 5.6](#)

shows an example of a separated chaining method for a hash table with 5 slots and a hash function $h(k) = k \bmod$

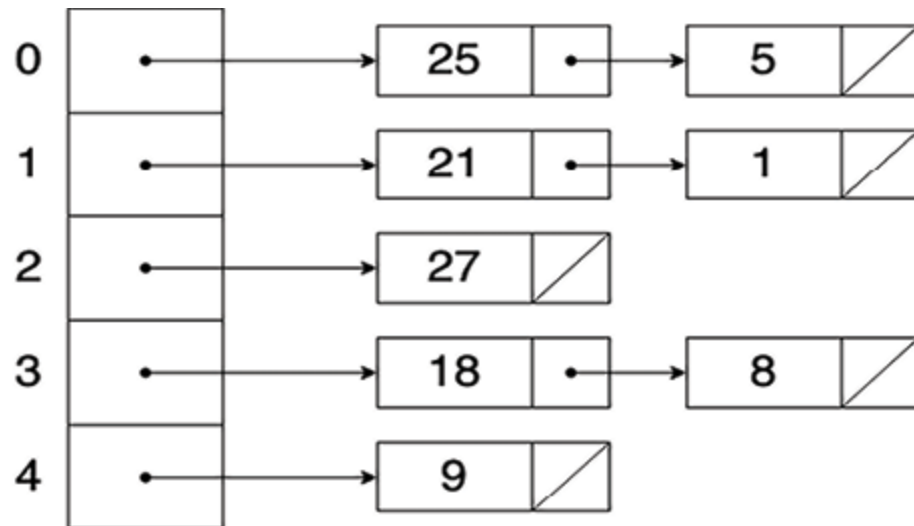


Figure 5.6: Separated chaining

Maps

The map is a data structure that provides keys to values. It is a collection that stores key-value pairs. Each key appears only once on the map.

Because of all the mentioned characteristics, a map can be used for hashing. The developer will define mapping rules (has function), while when a collision occurs, the stored value will (usually) be overridden with a new one.

Operations

The key is used in all map operations. The following operations can be performed on a map:

Inserting the value into the map.

Updating existing map value.

Getting value from the map.

Removing value from the map.

Checking if a key is present in the map.

The following section will show how these operations can be performed with the Go programming language.

Maps in Go

The map is one of the data structures supported by the Go programming language. The default (zero) value for the map is nil. The nil map does not contain keys, and keys cannot be added. The following statement will define a nil map:

```
var m map[string]int
```

The type between square brackets represents a key type, while the type after square brackets represents a stored element type. We must note that we can use custom types, defined with struct, in maps.

The function `make()` will create an initialized map ready for use. If we create a map with the `make()` function, we can add keys and use the map regularly, as demonstrated:

```
m := make(map[string]int)
```

We can use initializers to initialize the map. The following example will initialize a map that maps strings, representing days in a week to integers:

```
var m = map[string]int{

    "Monday": 1,

    "Tuesday": 2,

    "Wednesday": 3,

    "Thursday": 4,

    "Friday": 5,

    "Saturday": 6,

    "Sunday": 7,

}
```

We can use a key to add or update elements of a map. If the key is presented in the map, the element value will

be updated; if it is not, a new element will be added. The following statement will create/update the map element:

```
m["Monday"] = 0
```

A key can be used to get elements from a map in the following way:

```
day := m["Monday"]
```

If the key is not presented in the map, the default (zero) value for the element type will be returned. In our case, the default (zero) value for an integer is 0.

Elements can be removed from a map with the `delete()` function that accepts the map and key of an element that should be deleted as arguments. The following example will delete the element that corresponds with the key with the value Thursday:

```
delete(m, "Thursday")
```

It is also possible to check if a key is presented on the map in the following way:

```
day, ok := m["Monday"]
```

If a key is presented in map ok will be set to true; otherwise, it will be set to false. If a key is not presented in the map, the default (zero) value for the element type will be assigned to a variable day.

The range form of the for loop can be used to iterate through the map. The first value returned for each iteration represents the key and the second represents the value as shown:

```
for key, element := range m {  
  
    fmt.Println(key, element)  
  
}
```

Now, we are familiar with hashing and maps and can move to the next data structure.

Conclusion

This chapter presented maps, a collection that stores key-value pairs, and hashing, the process of transforming key into value. We learned how to choose a proper hash function for our needs and why some functions are better than others. We also saw how maps are supported by the Go programming language and how we can use them to solve everyday problems.

The next chapter will introduce our first non-linear data structure (tree) and display some traversal algorithms.

Points to remember

The capacity of the hash map is equal to the number of different keys.

The hash function should be as simple and uniform as possible.

Perfect hash is a minimal function without collisions; finding a perfect one is tricky.

The load factor is calculated by dividing the number of elements in the hash table by the size of the table.

The range form of the for loop can be used to iterate through the map.

Multiple choice questions

Which of these methods depends on key distribution?

Division method.

Digit folding method

Digit analysis method

Radix conversion method

The constant used in the multiplication method should be close to which mathematical constant?

Euler's number.

Golden ration.

Archimedes' constant Pi

Pythagoras' constant

Which Greek letter represents the load factor?

α (alpha)

β (beta)

γ (gamma)

δ (delta)

Answers

c

b

a

Questions

Why should the hash function be uniform?

How does the division method work?

How do the multiplication methods work?

How does the mid-square method work?

How does the radix conversion method work?

How does linear probing work?

How does random probing work?

How does quadratic probing work?

How does double hashing work?

Which operations can be performed on a map?

Key terms

Process of transforming any given key into value.

The function that transforms the key into an array (table) index.

Multiple keys store data in the same position in a hash table.

A data structure that maps keys to values.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Trees and Traversal Algorithms

Introduction

The fundamental concepts of trees will be introduced in this chapter. After that, a binary tree, a specific type of tree with operations and implementation, will be presented. The second part of the chapter will cover traversal algorithms with examples and illustrations. Ultimately, we will learn how to sort an array using trees.

Structure

The chapter covers the following topics:

Fundamentals of trees

Binary trees

Trees in Go

Traversal algorithms

Sorting an array with a tree

Objectives

By the end of this chapter, you will be familiar with trees and how to use them in practical solutions. Traversal algorithms presented here can be applied to multiple classes of solutions and can help in real-world problems.

Fundamentals of trees

A finite, non-empty set of elements is called a The elements of trees are called nodes and can store any data type (integer, string, and so on.). A tree is non-linear and (usually) homogenous. It is possible to define a tree where nodes store different data types. An example of a tree is shown in [Figure](#)

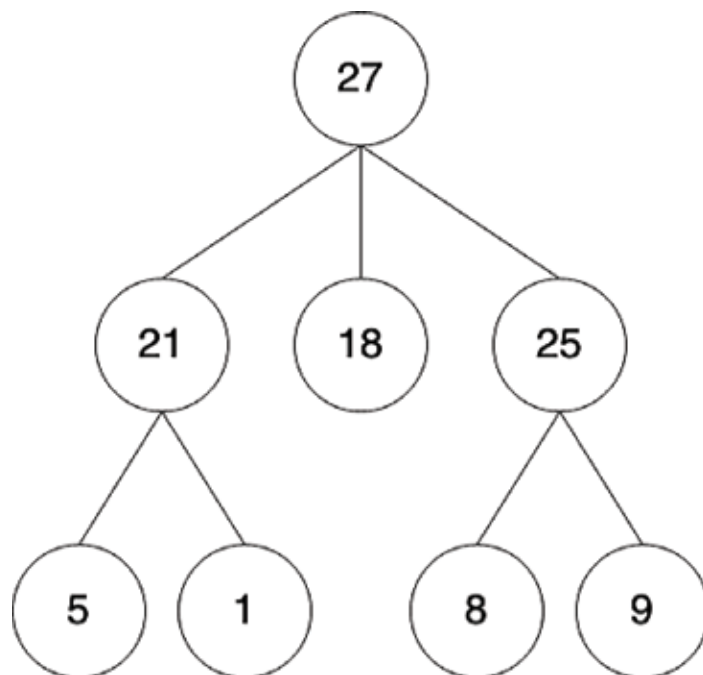


Figure 6.1: Tree

Edges connect these tree nodes. The number of edges coming out of the node is called the In [Figure](#) a node with value 27 has a degree while a node with value 21 has a degree 2. The degree of a tree is the maximum degree of a node in the tree (3 for the tree in [Figure](#)

There are two special types of nodes:

The A node without input branches; no node precedes it (node 27 in [Figure](#)

Nodes with a degree equal to 0 (nodes and 9 in [Figure](#) In other words, leaves are nodes without output branches.

Nodes, excluding the root node, can be extracted from the original tree in a subset of nodes, which are also nodes. These trees are called Nodes and 1 in [Figure 6.1](#) can form one subtree; node 21 is the root of a subtree. Edges are implicitly directed from root to subtrees.

The parent node is a node from which subtrees branch out. The child nodes have a common parent and serve as roots of subtrees branched from a parent. For example, in [Figure](#) node 27 is the parent node, while nodes and 25 are child nodes.

The path is a set of nodes where the next node in the sequence is a child node of the previous node. For

example, nodes and 5 from [Figure 6.1](#) form one path. The path length is equal to the number of edges in the path. We have two edges in the path formed by nodes and so the path length is two.

All nodes that form a path from the root to that node are ancestors. For node 5 in [Figure](#) nodes 27 and 21 are ancestors. All nodes in a node's subtree are descendants. For example, nodes 5 and 1 are descendants of node 21

The level of a node is the number of edges in the path between it and the root node. In [Figure](#) the levels of nodes are shown. Tree height is the maximum value of leaf levels in the tree (length of the longest path). For the tree in [Figure](#) the height is

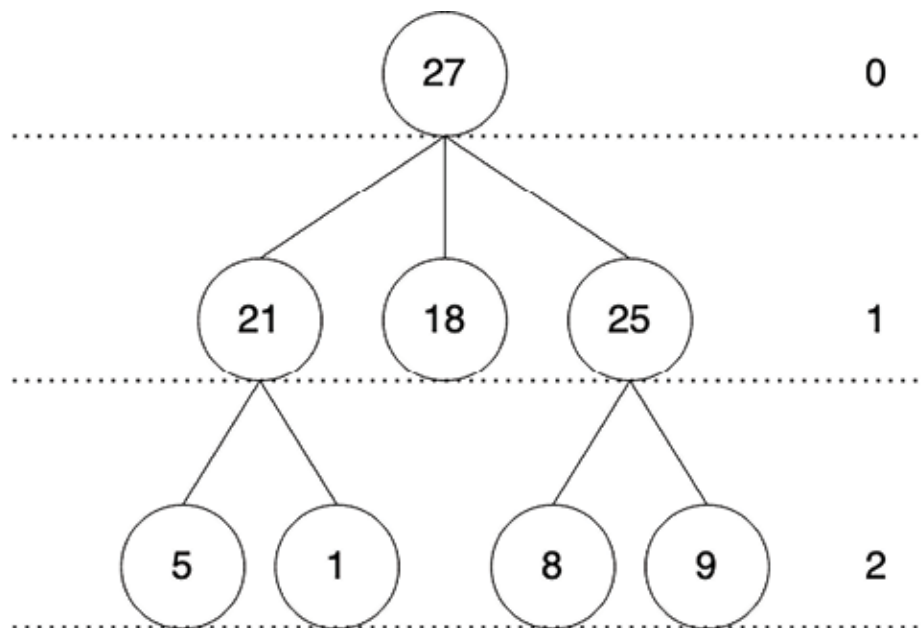


Figure 6.2: Node levels

Two trees are structurally similar if they have the same number of nodes and edges and the same topology. In [Figure](#) we can see two similar trees. If corresponding nodes of structurally similar trees have the same content, trees are identical. Trees from [Figure 6.3](#) will be identical if both roots have a value of left child nodes have a value of and right child nodes have a value of

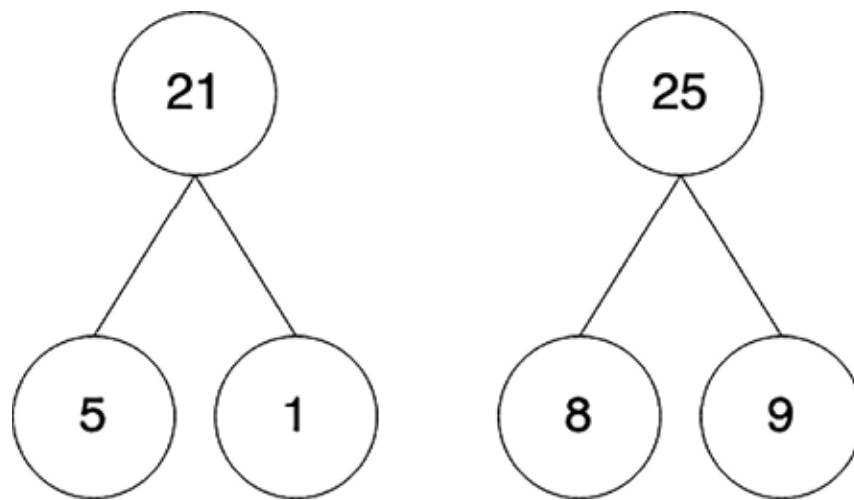


Figure 6.3: Structurally similar trees

The tree is ordered if the subtree of each node forms an ordered set, which means that the children of each node are somehow ordered (for example, by numerical or

alphabetical value of nodes). In [Figure](#) we can see an ordered version of the tree from [Figure](#)

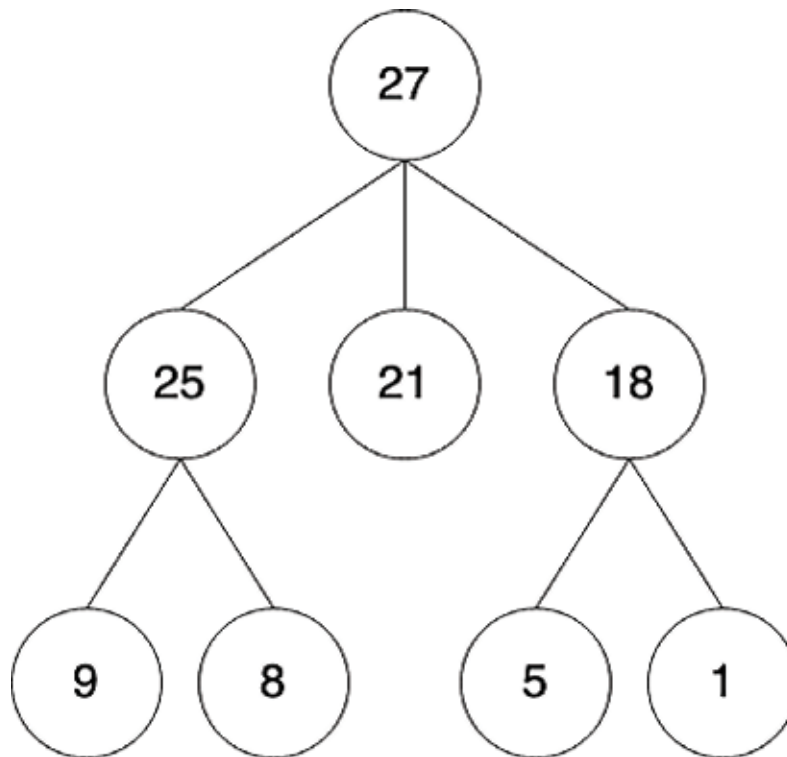


Figure 6.4: Ordered tree

The internal path length is a sum of the lengths of the paths from the root to all nodes. The following formula can represent this:

$$PI = \sum_i (in_i)$$

In the previous formula, represents the number of nodes in level i . In the tree from [Figure 6.4](#) we have one node at level zero, three nodes at level one, and four nodes at level two, so we can compute the internal path in the following way:

$$PI = \sum_i (in_i) = 0 \cdot 1 + 1 \cdot 3 + 2 \cdot 4 = 0 + 3 + 8 = 11$$

We can define a maximal number of nodes in a tree with degree m and height h . Zero level will have only one node the first level will maximally have m nodes the second level will maximally have m^2 nodes, and so on. A maximal number of nodes can be calculated with the following formula:

$$n = \sum_{i=0}^h m^i = \frac{m^{h+1} - 1}{m - 1}$$

The height of the tree in [Figure 6.4](#) is 3 while the degree is 3 (node 27 has three output edges). If we apply these values on the previous formula, we can calculate a maximal number of nodes:

$$\frac{m^{h+1} - 1}{m - 1} = \frac{3^{3+1} - 1}{3 - 1} = \frac{3^4 - 1}{2} = \frac{27 - 1}{2} = \frac{26}{2} = 13$$

Trees used linked representation for memory storage. In [Figure](#) we can see how the tree is represented in memory. To make everything cleaner, we will only convert the tree into objects stored in memory:

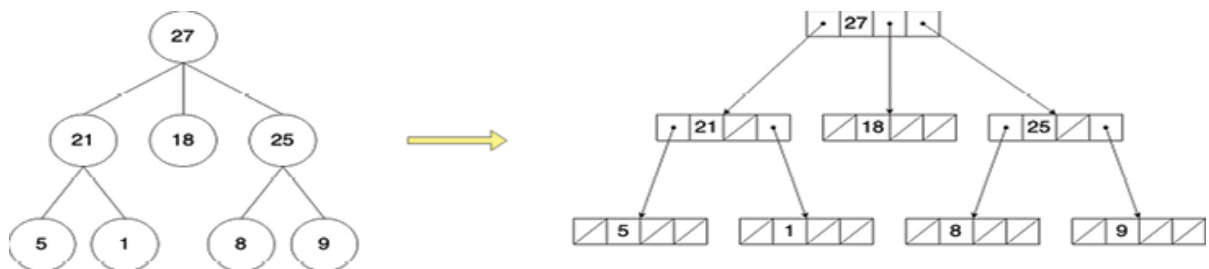


Figure 6.5: Memory representation of a tree

The trees are suitable for modeling objects that depict hierarchical organization, like family trees, the organization of employees in the company, or brackets at sports tournaments. In [Figure](#) the knockout stage of the 2022 FIFA World Cup is modeled on a tree:

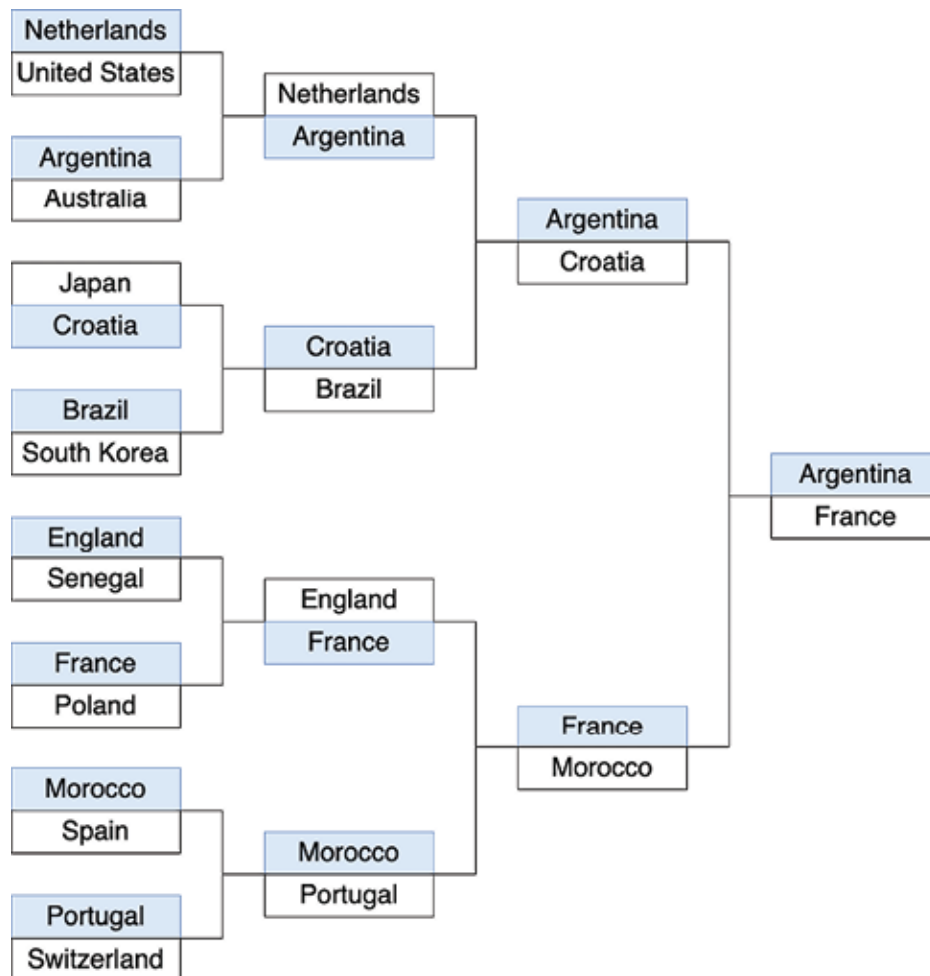


Figure 6.6: Sport tournament bracket modeled with tree

The following section will deal with binary trees as a specific tree type.

Binary tree

The tree where the node degree is at most 2 is called a binary tree. Each node, which is not a leaf, will have two or one child nodes. A node with one child can have a left or right child node.

We can define the following types of binary trees:

Full binary tree The binary tree where all internal (non-leaf) nodes have both children.

Complete binary tree A full binary tree where all leaves are on the same level.

Almost complete binary tree A tree where the final level is partially full and is filled successively, starting from the leftmost leaf, with less than nodes.

The complete binary tree is always the full binary tree, while an almost complete binary tree can be a full tree but does not have to be:

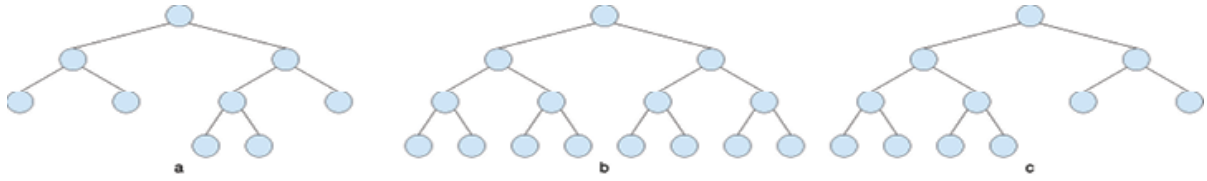


Figure 6.7: Types of binary trees

The binary tree is balanced if, for each node, the number of nodes in the left and right subtree does not differ in height by more than 1. [Figure 6.8](#) shows an example of one balanced binary tree:

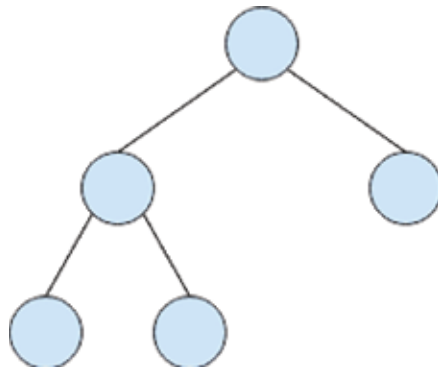


Figure 6.8: Balanced binary tree

Operations

The following operations can be performed on a tree:

Inserting a new node

Deleting node

Traversing tree

Inserting and deleting depend on where the nodes are inserted and deleted. Inserting or deleting a leaf is easy.

To insert a new leaf, an edge should be added from the parent node to the inserted one. In [Figure](#) a node with a value of 9 is added as a leaf:

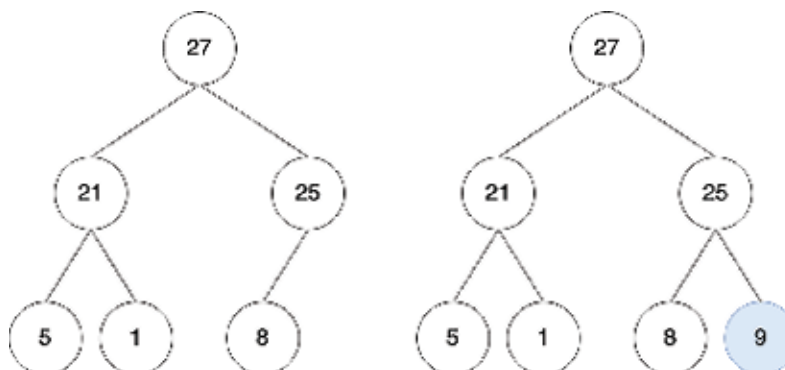


Figure 6.9: Inserting leaf node

Inserting an internal (non-leaf) node is complex. The edge of the parent node will now connect the parent node and a new one. A new edge must be added to connect the new node and the old child node of the parent node. In [Figure](#) a node with a value of 18 is added between nodes with values 27 and

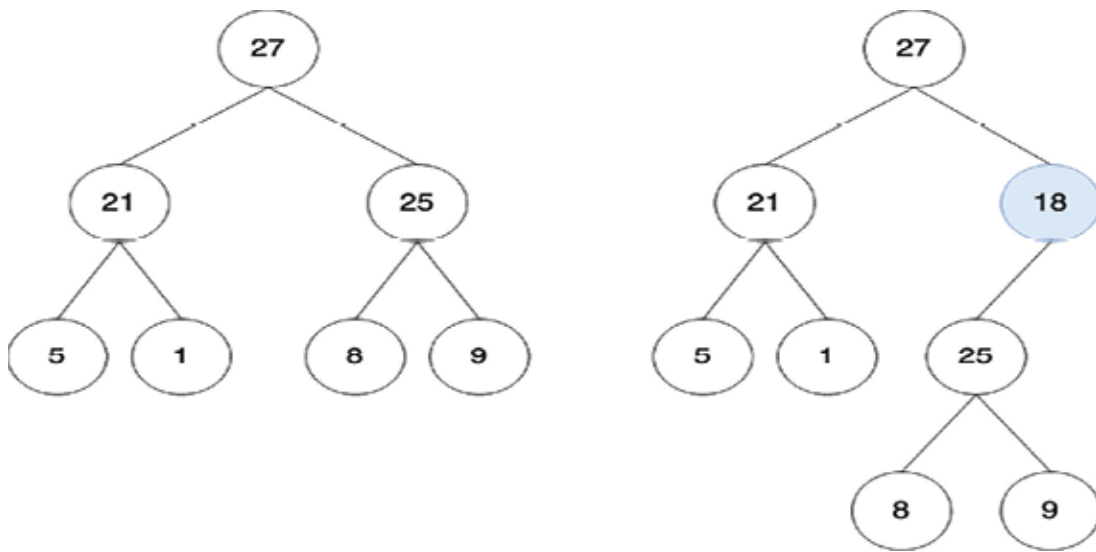


Figure 6.10: Inserting a new node

When the leaf node should be deleted, the node and edge between that node and the parent node are removed from the tree. In [Figure](#) a node with a value of 9 is deleted:

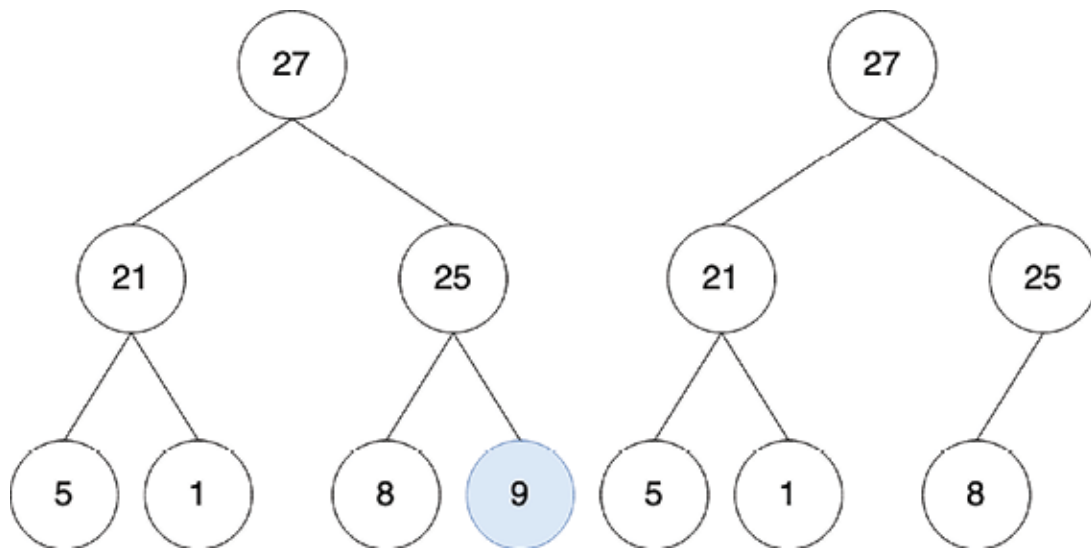


Figure 6.11: Deletion of leaf node

The easiest way to delete an internal (non-leaf) node is to reduce it to the deletion of the leaf node. This can be achieved by selecting one leaf, according to some criteria, from the subtree of the node that will be deleted. The value of the selected leaf node is assigned to one that will be deleted, and now the chosen leaf node can be deleted. In [Figure](#) a node with a value of 25 is deleted. The leftmost leaf is selected from the subtree of the deleted node:

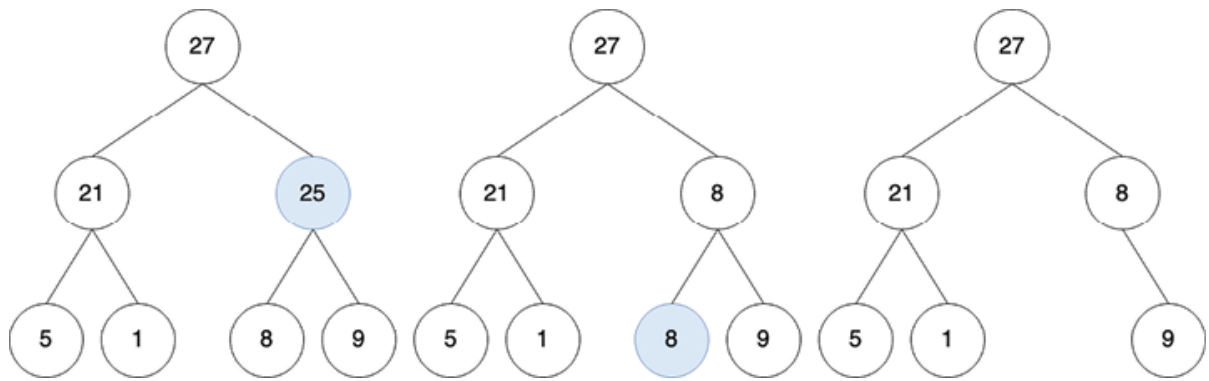


Figure 6.12: Deletion of an internal node

Traversal is quite a complex operation, and traversal algorithms will be covered in the following sections.

Trees in Go

The Go programming language does not have tree implementation, so we must implement it ourselves. We will implement a binary tree.

Let us start with a struct representing the node of the tree. That struct will contain an integer field that holds the value stored in the node, pointers to the left and right children, and a pointer to the parent node. The parent pointer is not necessary, but it will facilitate the implementation and delete operation:

```
type Node struct {
```

```
    value int
```

```
    left *Node
```

```
    right *Node
```

```
    parent *Node
```

```
}
```

The struct representing the binary tree will have a pointer to the root node:

```
type BinaryTree struct {
```

```
    root *Node
```

```
}
```

Insert operation

The method that implements the insert operation will receive three arguments: the node that will be the parent of the inserted node, the string side that can have values left and right (and determines if an inserted node will be a left or right child), and value that will be stored in the node. The method will return a pointer to the inserted node.

The situation when a node is inserted in the empty tree is handled first. The root pointer is set to reference the inserted node, and the root is returned. If the new node is inserted as a leaf, the operation is simple. Based on the value of the side parameter, a pointer to the left or right child of a parent node is set to reference a new node.

If the new node is inserted as internal, it will inherit the child from the node that becomes the parent to the inserted node, so it is necessary to set the pointer values of a new node. We must not forget to set the parent pointer of the inserted node, as shown:

```
func (bt *BinaryTree) Insert(node *Node, side string,
    value int) *Node {

    newNode := &Node{value, nil, nil, nil}

    // inserting to empty tree

    if bt.root == nil {

        bt.root = newNode

        return bt.root

    }

    if side == "left" {

        if node.left == nil {

            // inserting a leaf

            node.left = newNode
```

```
} else {
```

```
    // inserting internal node
```

```
    newNode.left = node.left
```

```
    node.left = newNode
```

```
}
```

```
} else {
```

```
    if node.right == nil {
```

```
        // inserting a leaf
```

```
        node.right = newNode
```

```
    } else {
```

```
        // inserting internal node
```

```
        newNode.right = node.right
```

```
    node.right = newNode  
  
    }  
  
}  
  
newNode.parent = node  
  
return newNode  
  
}
```

Delete operation

As mentioned, deletion can be a complex operation. Two methods will be defined for a node. The first method will delete the leaf node by detecting if a node is a left or right child and setting a proper pointer to

```
func (n *Node) DeleteLeaf() {
```

```
    if n.parent.left == n {
```

```
        n.parent.left = nil
```

```
    } else {
```

```
        n.parent.right = nil
```

```
    }
```

```
}
```

The second method will find and return the leftmost node of the current node by trying to go left whenever possible:

```
func (n *Node) findLeftmost() *Node {  
  
    node := n  
  
    next := n  
  
    for next != nil {  
  
        node = next  
  
        if next.left != nil {  
  
            next = next.left  
  
        } else {  
  
            next = next.right  
  
        }  
    }  
}
```



```

    }

    return node

}

```

Method Delete() will remove a node from the tree. It receives one argument, a node that will be removed. If a node that will be deleted is a leaf, we can use the previously defined DeleteLeaf() method. If the pointers to the children nodes are nil, the node is a leaf.

Here, we should check if the node that will be removed is the only node in the tree. In that case, we can just set the root pointer to

We will reduce the deletion of the internal (non-leaf) node to the deletion of the leaf node. The method FindLeftmost() will return the leftmost node, the value of that node will be assigned to the internal node, and the leaf node will be deleted, as shown:

```

func (bt *BinaryTree) Delete(node *Node) {

    // delete leaf

```

```
if node.left == nil && node.right == nil {
```

```
// delete root when the root is the only node
```

```
if node == bt.root {
```

```
    bt.root = nil
```

```
} else {
```

```
    node.DeleteLeaf()
```

```
}
```

```
// delete internal node
```

```
} else {
```

```
    leftmostNode := node.FindLeftmost()
```

```
    node.value = leftmostNode.value
```

```
    leftmostNode.DeleteLeaf()

}

}
```

The last method that will be presented in this section returns a pointer to the root node:

```
func (bt *BinaryTree) GetRoot() *Node {

    return bt.root

}
```

Here is a small code example that uses Insert() and Delete() methods created in this section:

```
func main() {

    var bt binarytree.BinaryTree

    node18 := bt.Insert(nil, "left", 18)
```

```
node8 := bt.Insert(node18, "left", 8)

node25 := bt.Insert(node18, "right", 25)

node5 := bt.Insert(node8, "left", 5)

bt.Insert(node8, "right", 9)

bt.Insert(node25, "left", 21)

bt.Insert(node25, "right", 27)

bt.Delete(node25)

bt.Delete(node5)

}
```

Traversal operation is essential, and there are many algorithms that can be used for it. We will see a couple of traversal algorithms in the following section.

Traversal algorithms

Traversal represents an operation where the nodes of a tree are accessed in a systematic order. Each node is visited only once. Traversal algorithms usually follow the convention that the left subtree is visited before the right one.

[Figure 6.13](#) presents a binary tree where we will demonstrate each algorithm. Four popular traversal algorithms are:

Preorder

Inorder

Postorder

Level-order

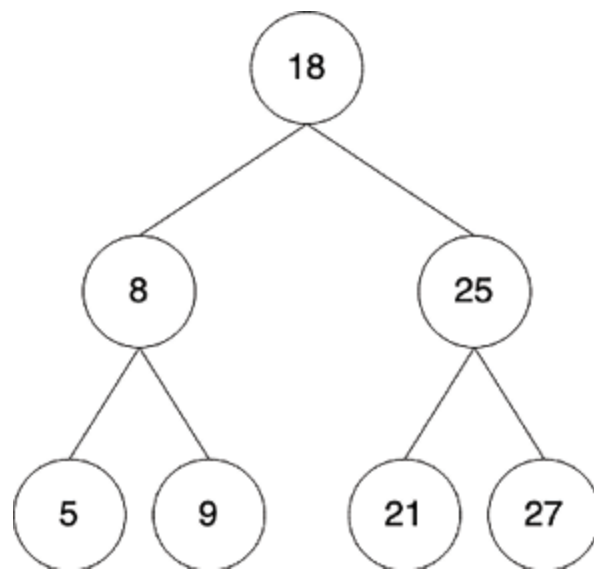


Figure 6.13: Binary tree

Preorder

With preorder traversal algorithms, nodes are visited in the following order:

Visit the root node.

Visit the left subtree (in a preorder way).

Visit the right subtree (in a preorder way).

Preorder traversal will first visit node 18 (root node) from a tree in [Figure](#). After that, the algorithm will move to the left subtree, where the subtree's root (node is visited). The algorithm will proceed to the left subtree of node which contains only one node, node. Since all nodes from the node 8 left subtree are visited, we move to the right subtree (node and so on. This process will be repeated until all nodes are visited.

The preorder traversal sequence for the tree in [Figure 6.13](#) is

As we can see, the steps of the algorithm are so it is ideal to implement it using a recursion:

```
func Preorder(node *Node) {  
  
    if node != nil {  
  
        fmt.Println(node.value)  
  
        Preorder(node.left)  
  
        Preorder(node.right)  
  
    }  
  
}
```


Inorder

The inorder traversal algorithm will visit nodes in the following order:

Visit the left subtree (in an inorder way).

Visit the root node.

Visit the right subtree (in an inorder way).

For the tree in [Figure](#) the algorithm will move across the left subtrees until it finds a node with no subtree (node 5). After that, the parent node, which is a root of the previous left subtree (node 8), is visited, and the algorithm will move to the right subtree (node 9). The algorithm will repeat this process until all nodes are visited. The inorder traversal sequence is 5, 8, 9, 18, 21, 25, 27.

Again, we can implement an algorithm with recursion:

```
func Inorder(node *Node) {
```

```
if node != nil {  
  
    Inorder(node.left)  
  
    fmt.Println(node.value)  
  
    Inorder(node.right)  
  
}  
  
}
```

Postorder

With postorder traversal algorithms, nodes are visited in the following order:

Visit the left subtree (in a postorder way).

Visit the right subtree (in a postorder way).

Visit the root node.

For the tree in [Figure](#) the algorithm will move across the left subtrees to find a node with no subtree (node 5). After that, a right subtree of the parent node is visited (node 9), and then the parent node (node 8), which is a root for the current subtree, is visited. The algorithm repeats until all nodes are visited. The root node of a tree will be visited last. The postorder traversal sequence is 5, 9, 8, 21, 27, 25, 18.

Here is a recursive implementation of the postorder algorithm:

```
func Postorder(node *Node) {
```

```
    if node != nil {
```

```
        Postorder(node.left)
```

```
        Postorder(node.right)
```

```
        fmt.Println(node.value)
```

```
    }
```

```
}
```

Level-order

The level-order algorithm is used less often. Starting from the root, all nodes in a level are visited from left to right before moving on to the next level. This is repeated until all nodes are visited. Level-order sequence for the tree in [Figure 6.13](#) is 18, 8, 25, 5, 9, 21, 27.

It is possible to implement a level-order algorithm with a queue and a for loop. We will start by adding a root node to the queue. In each iteration, one node will be dequeued and processed, and its children will be added to the queue. The process will be repeated until the queue is empty (there are no more nodes to process):

```
func Levelorder(node *Node) {
```

```
    next := node
```

```
    queue := New()
```

```
    queue.Enqueue(node)
```

```
for !queue.IsEmpty() {  
  
    next = queue.Dequeue()  
  
    fmt.Println(next.value)  
  
    if next.left != nil {  
  
        queue.Enqueue(next.left)  
  
    }  
  
    if next.right != nil {  
  
        queue.Enqueue(next.right)  
  
    }  
  
    }  
  
}
```

We can use the implementation of a queue similar to the one presented in [Chapter](#) Stack and This queue implementation will store a pointer to node instead of an integer. Here is a queue implementation:

```
func New() *Queue {  
  
    return &Queue{queue: list.New()}  
  
}  
  
func (q *Queue) Enqueue(node *Node) {  
  
    q.queue.PushFront(node)  
  
}  
  
func (q *Queue) Dequeue() *Node {  
  
if q.queue.Len() == 0 {  
  
    return nil  
  
}
```

```

    element := q.queue.Back()

    q.queue.Remove(element)

    return element.Value.(*Node)

}

func (q *Queue) IsEmpty() bool {

    return q.queue.Len() == 0

}

```

The following code sample will create a binary tree from [Figure 6.13](#) and execute all traversal algorithms from this section:

```

func main() {

    // Create a tree

    var bt binarytree.BinaryTree

```



```
node18 := bt.Insert(nil, "left", 18)
```

```
node8 := bt.Insert(node18, "left", 8)
```

```
node25 := bt.Insert(node18, "right", 25)
```

```
bt.Insert(node8, "left", 5)
```

```
bt.Insert(node8, "right", 9)
```

```
bt.Insert(node25, "left", 21)
```

```
bt.Insert(node25, "right", 27)
```

```
fmt.Println("Preorder")
```

```
binarytree.Preorder(bt.GetRoot())
```

```
fmt.Println("Inorder")
```

```
binarytree.Inorder(bt.GetRoot())
```

```
binarytree.Postorder(bt.GetRoot())
```

```
fmt.Println("Levelorder")
```

```
binarytree.Levelorder(bt.GetRoot())
```

```
}
```

In the following section, we will present one interesting usage of binary trees.

Sorting an array with a tree

The tree can be used to sort an array in the following way:

The elements from an unordered array are inserted in a binary tree. Where the element will be inserted is determined by its value. For example, elements with smaller values are placed into the left subtree. Array element with index 0 will become the root node.

The tree is traversed with the inorder algorithm.

This procedure is presented in [Figure](#) The algorithm complexity is $O(n \log n)$:

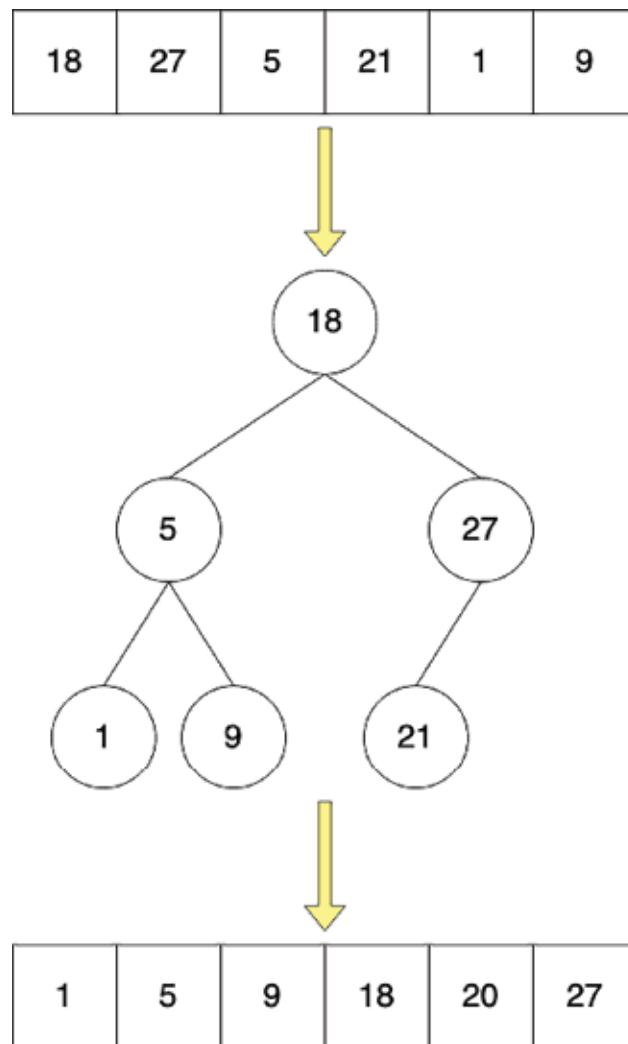


Figure 6.14: Array sorting with a tree

The following function will convert an array to a binary tree. It will iterate through elements and insert it as a tree node in the proper place:

```
func ArrayToTree(array []int) *Node {
```

```

var bt BinaryTree

root := bt.Insert(nil, "left", array[0])

for i := 1; i < len(array); i++ {

    side, node := find(array[i], root)

    bt.Insert(node, side, array[i])

}

return root

}

```

The find() function is the core of this process. It will go through the tree, starting from the root node, and find where to insert a node. If the value of a new node is less than that of the node currently visited, it will go to the left; otherwise, it will go to the right. The new node will always be inserted as a leaf, as shown:

```

func find(value int, root *Node) (side string, node *Node)
{

```

```
next := root
```

```
for next != nil {
```

```
    node = next
```

```
    if value <= next.value {
```

```
        side = "left"
```

```
        next = next.left
```

```
    } else {
```

```
        side = "right"
```

```
        next = next.right
```

```
    }
```

```
}
```

```
return
```

```
}
```

The following code segment will sort an array:

```
func main() {  
  
    array := []int{18, 27, 5, 21, 1, 9}  
  
    root := binarytree.ArrayToTree(array)  
  
    binarytree.Inorder(root)  
  
}
```

This solution can be improved. The heap is generated instead of a binary tree. The heap is a complete or almost complete binary tree, where the value stored in the parent node is always greater or equal to the value stored in the child node, which means that the greatest value is stored in the root. [Figure 6.15](#) presents how the array from the previous example is converted to a heap:

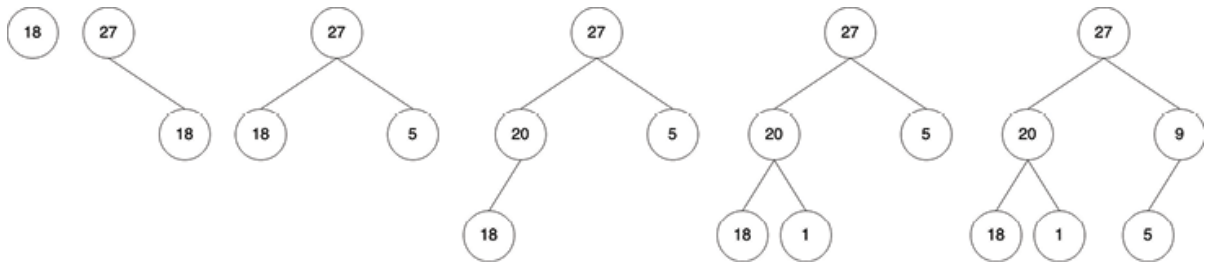


Figure 6.15: Heap creation

As mentioned in [Chapter](#) Stack and the Go standard library contains a heap interface in the container/heap package. We will define the type which is an array of integers:

```
type IntHeap []int
```

This type will implement a heap interface in the following way:

```
func (h IntHeap) Len() int {
```

```
    return len(h)
```

```
}
```

```
func (h IntHeap) Less(i, j int) bool {
```

```
    return h[i] > h[j]
```



```
}
```

```
func (h IntHeap) Swap(i, j int) {
```

```
h[i], h[j] = h[j], h[i]
```

```
}
```

```
func (h *IntHeap) Push(v any) {
```

```
*h = append(*h, v.(int))
```

```
}
```

```
func (h *IntHeap) Pop() any {
```

```
old := *h
```

```
n := len(old)
```

```
v := old[n-1]
```

```
*h = old[0 : n-1]
```

```
    return v
```

```
}
```

The method `Less()` is implemented this way to ensure that the largest element is in the root. Everything else is straightforward and similar to the interface implementation from [Chapter](#) `Stack` and

When the heap is generated, the root will be taken from the tree and placed into the array. The heap is reorganized to select a new root node (the `heap.Pop()` function will perform this task), which is repeated until the heap is empty. This sorting algorithm is known as heap

```
func Heapsort(array *IntHeap) []int {
```

```
    heap.Init(array)
```

```
    n := array.Len()
```

```
    sortedArray := make([]int, n)
```

```
    for i := n - 1; array.Len() > 0; i-- {
```

```
        sortedArray[i] = heap.Pop(array).(int)

    }

    return sortedArray

}
```

The following main() function will use a heap sort algorithm to sort an array of integers:

```
func main() {

    array := &binarytree.IntHeap{18, 27, 5, 21, 1, 9}

    fmt.Println(binarytree.Heapsort(array))

}
```

Conclusion

This chapter introduced the first non-linear structure, tree, operations that can be performed on it, and different types of trees. Besides that, we learned how to implement trees with the Go programming language and presented multiple traversal algorithms. We also saw how to use trees to sort an array.

Now that we are familiar with trees, the only data structure left to discuss is the graph. We will be discussing graphs in the next chapter.

Points to remember

The root is a node without input branches. Leaves are nodes with degrees equal to 0.

The trees are suitable for modeling objects that depict hierarchical organization.

The complete binary tree is always the full binary tree.

The almost complete binary tree can be full but does not have to be.

Inserting or deleting a leaf node is easy.

The easiest way to delete an internal node is to reduce it to the deletion of a leaf node.

The Go programming language does not have tree implementation.

The tree can be used to sort an array.

Multiple choice questions

What is the degree of the leaf?

0

1

2

3

What is the maximal degree of the node in a binary tree?

0

1

2

3

What is the binary tree where all non-leaf have both children called?

Complete binary tree

Full binary tree

Almost complete binary tree

Balanced binary tree

How many times can a node be visited during traversal?

1

2

3

4

With which algorithm root node will be visited last?

Inorder

Level-order

Preorder

Postorder

Which data structure can be used to implement a level-order traversal algorithm?

Stack

Queue

Map

Array

Answers

a

c

b

a

d

b

Questions

What is the degree of node?

How is the degree of a tree calculated?

What is a subtree?

What are ancestor and descendant nodes?

When are trees structurally similar, and when they are identical?

When is a binary tree balanced?

Which operations can be performed on a tree?

Key terms

A finite non-empty set of nodes.

The elements of the tree.

Connection between tree nodes.

Binary The tree where the node degree is at most 2.

The operation where the nodes of a tree are accessed in a systematic order.

Complete or almost complete binary tree, where the value stored in the parent node is always greater or equal to that stored in the child node.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Graphs and Traversal Algorithms

Introduction

This chapter will introduce the fundamental concepts of graphs and all operations that can be performed on them. We will show how to implement graphs with the Go programming language and present some popular traversal algorithms. The second part of this chapter will cover other usages of graphs. We will learn what spanning trees and transitive closures are, how to find the shortest path and introduce the term flow in graphs. Ultimately, we will present topological sorting and the concept of critical path.

Structure

The chapter covers the following topics:

Fundamentals of graphs

Operations

Graphs in Go

Traversal algorithms

Spanning tree

Transitive closure

Shortest paths

Flow in graphs

Topological sorting

Critical path

Objectives

By the end of this chapter, you will be familiar with graphs and how to use them in real-life solutions. Algorithms presented in this chapter can be used to simulate a wide range of everyday issues. Also, you can use graphs to find and avoid potential problems (this is especially true for finding a flow and a critical path).

Fundamentals of graphs

Graph G is defined as a pair of sets where V is a finite, non-empty set, while E represents binary relations between elements from set V . The elements from set V are called nodes and the elements from set E are called edges. Graphs are perfect for modeling non linear relations.

The number of nodes represents the order of a graph, while the number of edges represents the size of a graph. Each edge from the set E relates to only one pair of nodes from the set V , which it joins. That edge is incident on nodes u and v .

If node pairs from the set V are ordered (it is known from which node the edge goes to which node), the graph is directed. The graph is undirected if node pairs are unordered. The mixed graph contains directed and undirected edges:

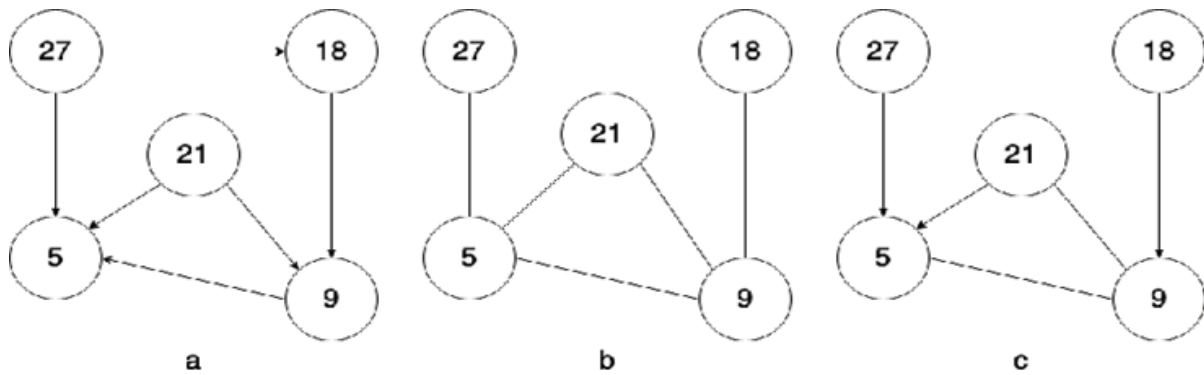


Figure 7.1: Types of graphs

The degree of a node is equal to the number of incident edges. For directed graphs, the number of incident edges leading into the node represents indegree. Conversely, the number of incident edges leading away from the node represents outdegree.

An edge that starts and ends at the same node is called a loop (edge related to node 5 in [Figure](#) Multiple edges that join the same nodes are called parallel edges (edges between nodes 9 and 18 in [Figure](#) In practice, the existence of parallel edges is often prohibited. A graph that contains parallel edges and loops is called a multigraph A graph without parallel edges and loops is called a simple Refer to the following figure for a reference of the multigraph:

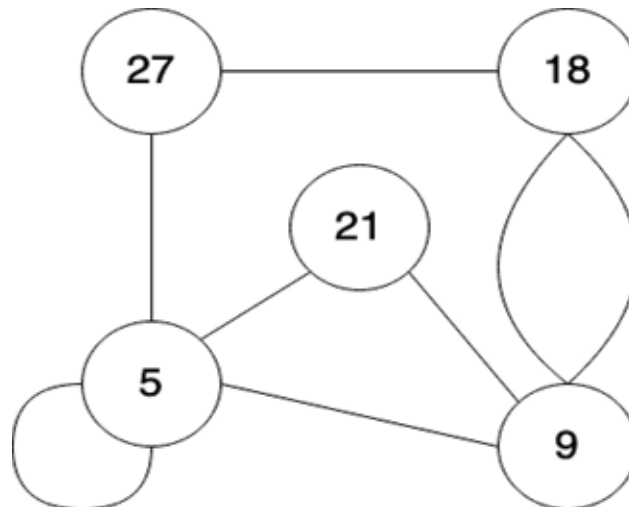


Figure 7.2: Multigraph

The subgraph $G' = (V', E')$ of graph $G = (V, E)$ is a graph where V' is a subset of V and E' is a subset of E . In [Figure](#) nodes and corresponding edges form a subgraph. If weight is assigned to the edges of a graph, that graph is called a weighted graph. A directed weighted graph is known as a network. Refer to the following figure for a subgraph:

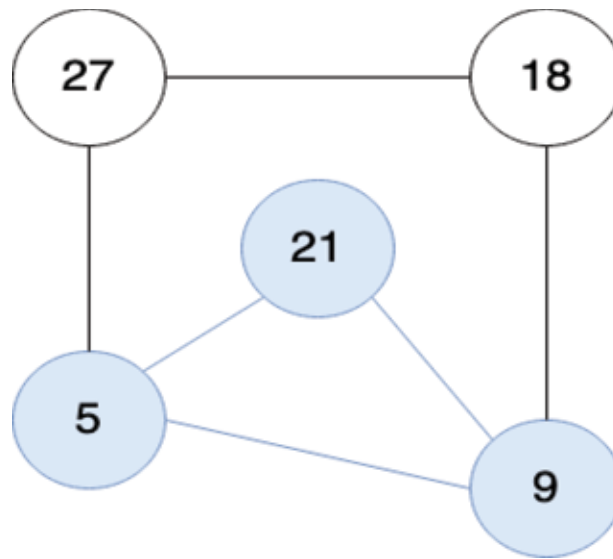


Figure 7.3: Subgraph

The path between two graph nodes, u and v is defined as an array of nodes $u, v_1, v_2, \dots, v_{n-1}, v$, where $v_i = v_{i+1}$ and edge (v_{i-1}, v_i) belongs to a set of edges E for $i = 1, 2, \dots, n-1$, which makes the next branch starts from the node where the previous branch ends. If a non-zero length path exists between nodes u and node v is reachable from node

If all nodes on the path are distinct except (maybe) the first and last (the path does not cross more than once through the same node), the path is called a simple path. On the graph from [Figure](#) nodes 5 and 9 form a simple path.

The path that starts and finishes in the same node is called a cycle (the path formed by nodes 5 and 9 in the graph from

[Figure](#) The graph that contains cycles is a cyclic graph; otherwise, the graph is

A complete graph is an undirected graph where every pair of nodes is connected by an edge. The number of edges in the complete graph (maximum number of edges) can be defined with the following formula:

$$e = \frac{n \cdot (n - 1)}{2}$$

In the previous formula, n represents the number of nodes. For the graph in [Figure](#) we can calculate the number of edges:

$$e = \frac{n \cdot (n - 1)}{2} = \frac{4 \cdot (4 - 1)}{2} = \frac{4 \cdot 3}{2} = \frac{12}{2} = 6$$

The graph is called a dense graph if the number of edges is relatively large (close to the maximum number of edges). On the other hand, if the number of edges is small, the graph is called a sparse. For example, the maximum number of edges for a graph with four nodes is six. If that graph has five or four edges, it is a dense graph; if it has one or two edges, it is a sparse graph.

Take a look at the following complete graph:

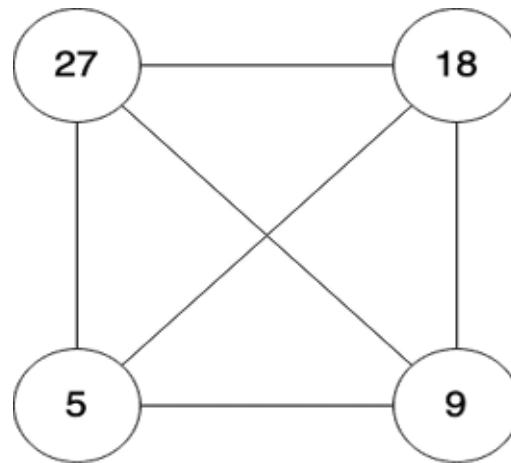


Figure 7.4: Complete graph

A connected graph is an undirected graph with a path between every pair of nodes. If this is not the case, the graph consists of multiple connected components. A strongly connected graph is a directed graph where any two nodes are reachable from each other, as shown:

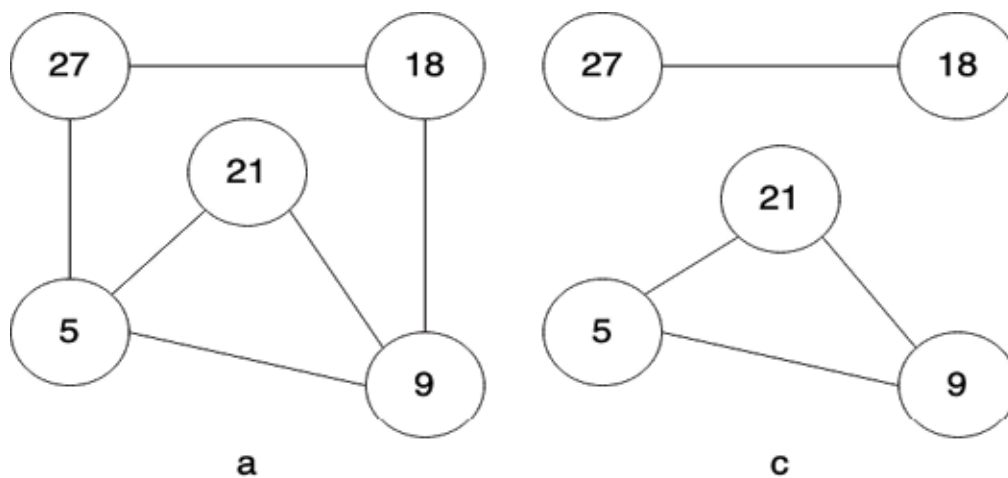


Figure 7.5: Graph connectivity

Operations

The following operations can be performed on a graph:

Inserting a new node

Deleting node

Traversing graph

Inserting is a simple operation. A new node and related edges are added to the corresponding sets. In [Figure](#) the node with a value of 1 is inserted with edges between it and nodes 9 and

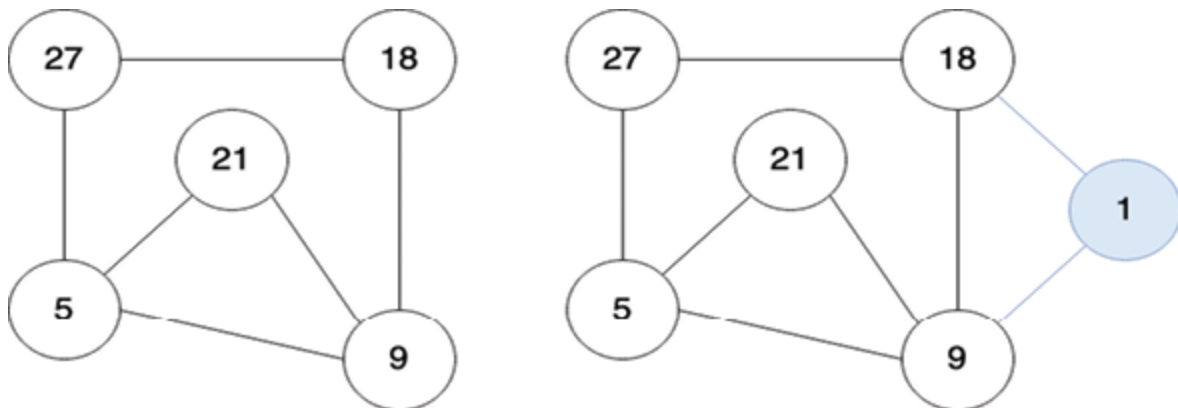


Figure 7.6: Inserting a new node

Delete is an opposite operation. A selected node and all incident edges are removed from corresponding sets. In [Figure](#) node 21 and the edges between it and nodes 5 and 9 are removed:

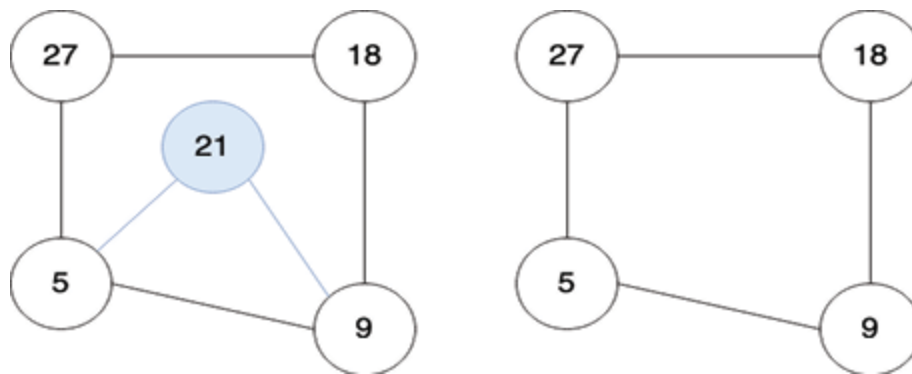


Figure 7.7: Node deletion

Other operations are more complex and include interesting algorithms. They will be covered in separate sections.

Graphs in Go

The Go programming language does not have a graph implementation. That is not a problem; we have become experienced enough to implement it ourselves. Let us implement the graph by strictly following the definition, with two sets, one for the nodes and another for the edges.

A struct representing a graph node will have one filed integer value held by the node. Edge will be defined as a struct that contains two nodes joined by that edge. We can add one simple function that initializes and returns a new node, as shown:

```
type Node struct {
```

```
    value int
```

```
}
```

```
type Edge struct {
```

```

    u, v Node

}

func NewNode(value int) Node {

    return Node{value}

}

```

A set can be simulated with a where the key is some defined struct or and an empty struct is a value. The graph struct will have node and edge sets, with function New() that will initialize them and return a pointer to a graph:

```

type Graph struct {

    nodes map[Node]struct{}

    edges map[Edge]struct{}

}

```

```
func New() *Graph {  
  
    return &Graph{  
  
        nodes: make(map[Node]struct{}),  
  
        edges: make(map[Edge]struct{}),  
  
    }  
  
}
```

Insert operation can be implemented with two simple functions. Function `AddNode()` receives a node as an argument and inserts it into the node set, while function `AddEdge()` accepts two nodes as arguments, creates a new edge that connects these nodes, and adds an edge to the set, as shown:

```
func (g *Graph) AddNode(n Node) {  
  
    g.nodes[n] = struct{}{}  
  
}
```

```
func (g *Graph) AddEdge(u, v Node) {
```

```
    e := Edge{u, v}
```

```
    g.edges[e] = struct{}{}
```

```
}
```

A function that implements a delete operation will remove a specified node from the set of nodes and all edges incident to that node from the set of edges. Incident edges can be found by iterating through the set:

```
func (g *Graph) RemoveNode(n Node) {
```

```
    delete(g.nodes, n)
```

```
    for e := range g.edges {
```

```
        if e.u == n || e.v == n {
```

```
            delete(g.edges, e)
```

}

}

}

Now, when everything is set, we will move forward on how to traverse through the graph.

Traversal algorithms

Traversal is an operation where all graph nodes are visited only once in a systematic order, and some processing is performed on the visited node. The graph does not have the root node, so the starting node must be selected somehow.

If a node can be encountered more than once, it can be visited only during the first pass and ignored later. This is the reason why the node should be marked as

Traversing the graph creates a traversal tree. The graph traversal algorithms classify edges into the following four categories:

Tree edges: Connects parent and child nodes and represent an edge through which the child is visited. Only these edges are included in the traversal tree.

Forward edges: Connect the node to its descendant that has already visited.

Back edges: Connect the node to its ancestor that has already been visited.

Cross edges: Connect two nodes that are not in an ancestor-descendent relation.

[Figure 7.8](#) presents a directed graph where we will demonstrate traversal algorithms:

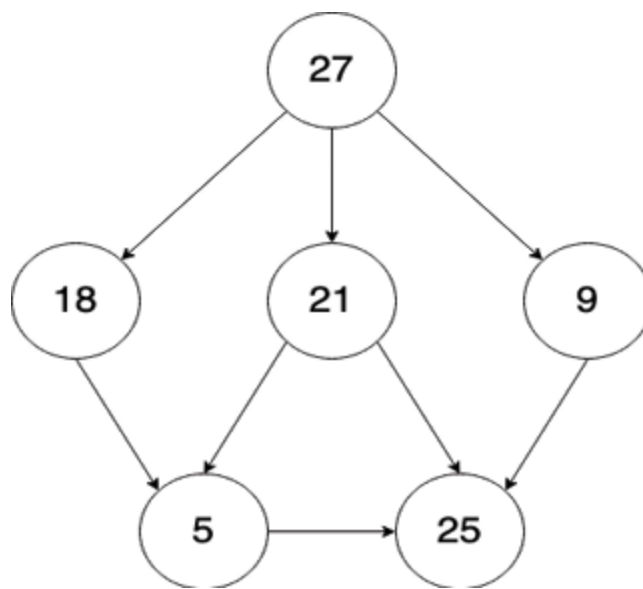


Figure 7.8: Directed graph

Two popular traversal algorithms are:

Breadth-first search

Depth-first search

Breadth-first search

With breadth-first search, the starting node is visited first, then visits all neighboring nodes (in some sequential order), then all their unvisited neighbors, and so on. The algorithm will end when there are no more unvisited nodes. As the name suggests, the algorithm goes by levels of the same distance from the starting node.

If we select node 27 as a starting node, the breadth-first search will create the following traversal sequence: 25 (for the graph in [Figure](#)

A visit vector is used in both algorithms to mark which nodes are visited until the current moment. This vector will be implemented as a map with the node's value as a key and a Boolean as a value. If the Boolean is equal to true, the node is visited.

Breadth-first search implementation will use a queue. We can use the integer queue implemented in [Chapter](#) Stack and When a node is visited, it will be marked in the visit vector and added to the queue. Order inside the queue determines traversal order. When a node is

removed from the queue, its unvisited neighbors will be added. Unvisited neighbors are found by iterating through the set of edges and examining incident edges.

The following function implements a breadth-first search traversal algorithm:

```
func BFS(g *Graph, start *Node) {
```

```
    visit := make(map[int]bool)
```

```
    for n := range g.nodes {
```

```
        visit[n.value] = false
```

```
    }
```

```
    visit[start.value] = true
```

```
    queue := NewQueue()
```

```
    queue.Enqueue(start)
```

```
    for !queue.IsEmpty() {
```

```
    u := queue.Dequeue()

    fmt.Println(u.value)

    for edge := range g.edges {

        if edge.u.value == u.value &&

            !visit[edge.v.value] {

                visit[edge.v.value] = true

                n := edge.v

                queue.Enqueue(&n)

            }

        }

    }
```

```
}
```

The first part of the function will initialize the visit vector. The starting node must be provided through a function argument.

Depth-first search

The depth-first search follows a path from the starting node in single directions as far as possible. When there are no more nodes that the path can visit, the algorithm returns to a previous node on the traversed path and follows another path through the graph until all nodes are visited.

Traversal begins from the selected starting node, then one of the neighboring nodes is visited, then the unvisited neighbor's neighbor, and so on, until the node without neighbors or node where all neighbors are visited is reached (node 25 from the graph in [Figure](#)). The algorithm then returns to the last visited node on a path with unvisited neighbors (node 27 from the graph in [Figure](#)) and starts following the new path from that node through an unvisited neighbor (node 21 from the graph in [Figure](#)).

The depth-first search traversal sequence for the graph in [Figure 7.8](#) is and

This algorithm can be easily implemented with recursion. Function DFS() initializes visit vector and call dfsVisit() function:

```
func DFS(g *Graph, start *Node) {  
  
    visit := make(map[int]bool)  
  
    for n := range g.nodes {  
  
        visit[n.value] = false  
  
    }  
  
    dfsVisit(g, start, visit)  
  
}
```

Function dfsVisit() will mark the starting node as visited, select one of the unvisited neighboring nodes (with for loop), and recursively repeat that process:

```
func dfsVisit(g *Graph, u *Node, visit map[int]bool) {
```

```
visit[u.value] = true

fmt.Println(u.value)

for edge := range g.edges {

    if edge.u.value == u.value &&

        !visit[edge.v.value] {

            dfsVisit(g, &edge.v, visit)

        }

    }

}
```

The following code will create a graph from [Figure 7.8](#) and execute the traversal operation:

```
main() {

    node27 := graph.NewNode(27)
```

node18 := graph.NewNode(18)

node21 := graph.NewNode(21)

node9 := graph.NewNode(9)

node5 := graph.NewNode(5)

node25 := graph.NewNode(25)

g := graph.New()

g.AddNode(node27)

g.AddNode(node18)

g.AddNode(node21)

g.AddNode(node9)

g.AddNode(node5)

```
g.AddNode(node25)
```

```
g.AddEdge(node27, node18)
```

```
g.AddEdge(node27, node21)
```

```
g.AddEdge(node27, node9)
```

```
g.AddEdge(node18, node5)
```

```
g.AddEdge(node21, node5)
```

```
g.AddEdge(node21, node25)
```

```
g.AddEdge(node9, node25)
```

```
g.AddEdge(node5, node25)
```

```
fmt.Println("BFS")
```

```
graph.BFS(g, &node27)
```

```
fmt.Println("DFS")
```

```
graph.DFS(g, &node27)
```

```
}
```

Spanning tree

If we have undirected connected graph G = the spanning tree of graph G is tree $ST = (U, E)$, which meets the following conditions:

Spanning tree ST contains all nodes of graph G ($U = V$).

Spanning tree ST contains only a certain number of edges ($|E| = |V| - 1$) necessary to connect all nodes so there are no cycles.

The spanning tree is not unique and can be generated with traversal algorithms (with minor corrections). In [Figure](#) we can see two different spanning trees generated from the same graph:

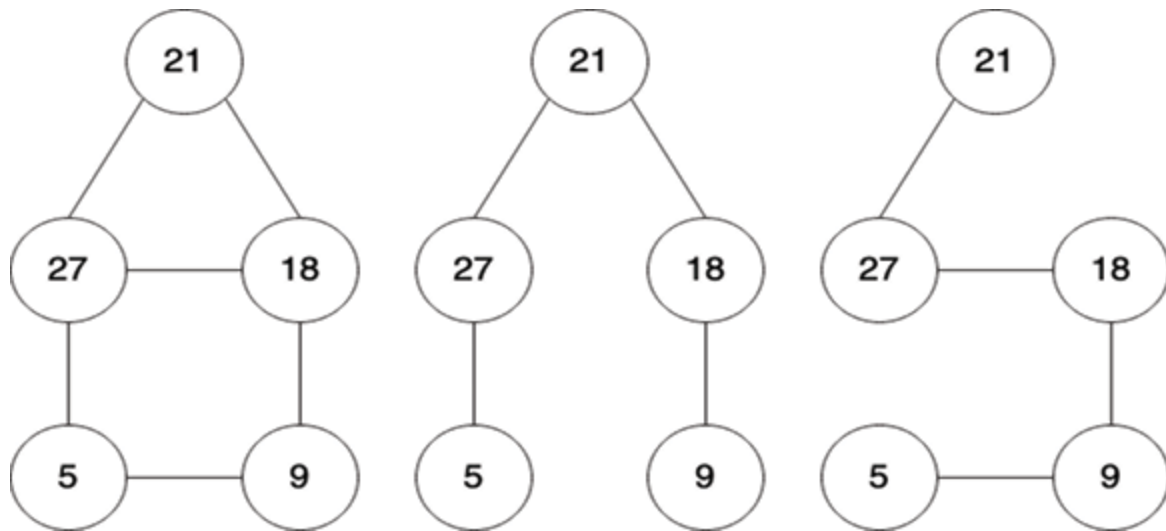


Figure 7.9: Spanning trees

It is often necessary to find a unique spanning tree. For example, we have a graph where nodes represent cities while edges represent potential roads between them. We should find a spanning tree representing a road network that connects all cities, selected by some criteria (shortest distances, lower construction costs, and so on). To archive that, we must introduce the cost of spanning tree, which is defined as the sum of the weights of all edges in the tree:

$$\sum w(i, v), (u, v) \in E'$$

A minimal-cost spanning tree is a spanning tree of a graph that has the lowest price. The two most popular algorithms for finding an MST are:

Prim's algorithm

Kruskal's algorithm

Before we proceed with implementation, we must implement a weighted graph. Implementation will be the same, with one difference: the edge will have assigned weight. Here is a complete code:

```
type WeightedEdge struct {  
  
    u, v Node  
  
    weight int  
  
}  
  
type WeightedGraph struct {  
  
    nodes map[Node]struct{}
```



```
edges map[WeightedEdge]struct{}
```

```
}
```

```
func NewWeightedGraph() *WeightedGraph {
```

```
    return &WeightedGraph{
```

```
        nodes: make(map[Node]struct{}),
```

```
        edges: make(map[WeightedEdge]struct{}),
```

```
    }
```

```
}
```

```
func (wg *WeightedGraph) AddNode(n Node) {
```

```
    wg.nodes[n] = struct{}{}
```

```
}
```

```

func (wg *WeightedGraph) AddEdge(u, v Node, w int)
{

    e := WeightedEdge{u, v, w}

    wg.edges[e] = struct{}{}

}

func (wg *WeightedGraph) RemoveNode(n Node) {

    delete(wg.nodes, n)

    for e := range wg.edges {

        if e.u == n || e.v == n {

            delete(wg.edges, e)

        }

    }

}

```

```
}
```

We can also create a struct that represents a minimal-cost spanning tree that will contain node and edge sets with the Print() method:

```
type MST struct {  
  
    nodes map[Node]struct{}  
  
    edges map[WeightedEdge]struct{}  
  
}
```

```
func (m MST) Print() {  
  
    fmt.Println("Nodes:")  
  
    for n := range m.nodes {  
  
        fmt.Printf("%v ", n.value)  
  
    }
```

```
    fmt.Println()
```

```
fmt.Println("Edges: ")
```

```
for e := range m.edges {
```

```
    fmt.Printf("%v ", e)
```

```
}
```

```
}
```

We will try to find an MST for the weighted graph presented in [Figure](#)

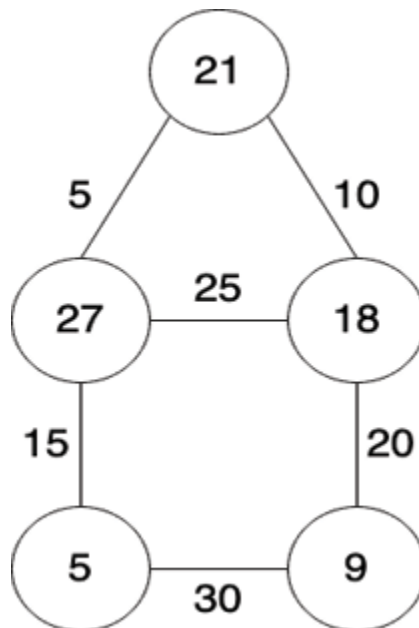


Figure 7.10: Weighted graph

Prim's algorithm

Prim's algorithm, named after American mathematician Robert Clay Prim is a greedy algorithm that will incrementally increase a connected component and choose a minimal weighted edge at each step. The algorithm begins from the starting node and ends when MST is formed. Interestingly, Prim did not develop the algorithm, but it was developed by Czech mathematician Vojtech Jarnik (that is why it is sometimes called Jarnik's and later republished by Prim. Algorithm steps are presented in [Figure](#) where node 21 is a starting node:

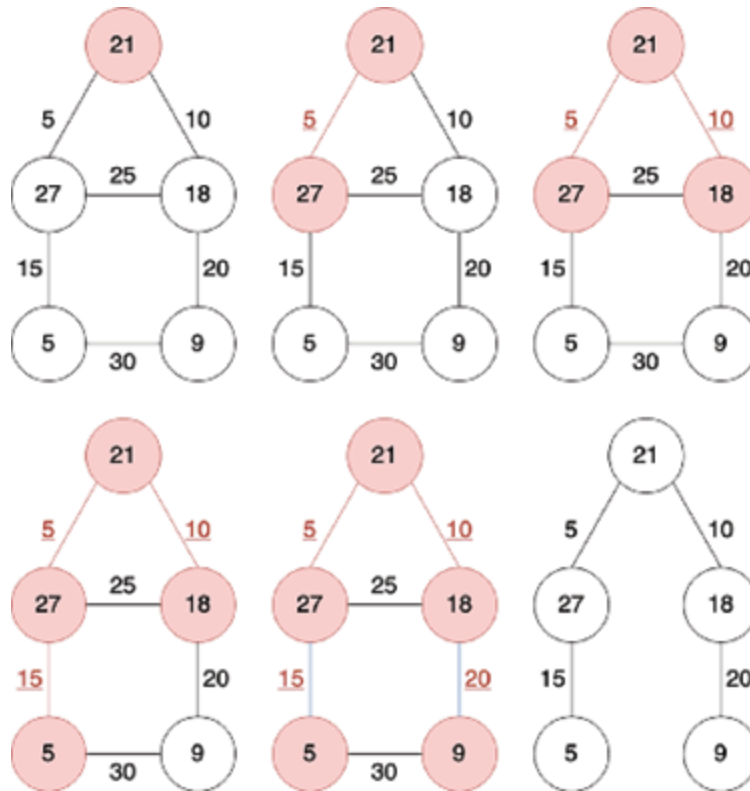


Figure 7.11: Prim's algorithm

The following code segment will implement the algorithm. In each iteration, one node and the minimum weighted edge will be added to the respective set until all graph's nodes become part of MST:

```
func Prim(wg *WeightedGraph, start *Node) *MST {

    treeNodes := make(map[Node]struct{})
```

```

treeEdges := make(map[WeightedEdge]struct{})

treeNodes[*start] = struct{}{}

for len(treeNodes) != len(wg.nodes) {

    node, minEdge := minEdge(wg, treeNodes)

    treeNodes[node] = struct{}{}

    treeEdges[minEdge] = struct{}{}

}

return &MST{treeNodes, treeEdges}

}

```

The minEdge() function will return the minimum weighted edge where one node belongs to the set of MST nodes while the second node does not:

```

func minEdge(wg *WeightedGraph,

```



```
nodes map[Node]struct{}) (n Node, we WeightedEdge)
{
```

```
    min := 1000
```

```
    for e := range wg.edges {
```

```
        _, ok1 := nodes[e.u]
```

```
        _, ok2 := nodes[e.v]
```

```
        if ok1 && !ok2 && e.v.value < min {
```

```
            n = e.v
```

```
            we = e
```

```
        }
```

```
    }
```

```
    return
```

}

Kruskal's algorithm

Described by American mathematician, statistician, computer scientist, and psychometrician Joseph Bernard Kruskal Kruskal's algorithm starts from unconnected nodes without edges (forest). It adds an edge with a minimal weight that connects two detached components in each iteration. The algorithm steps are shown in [Figure](#)

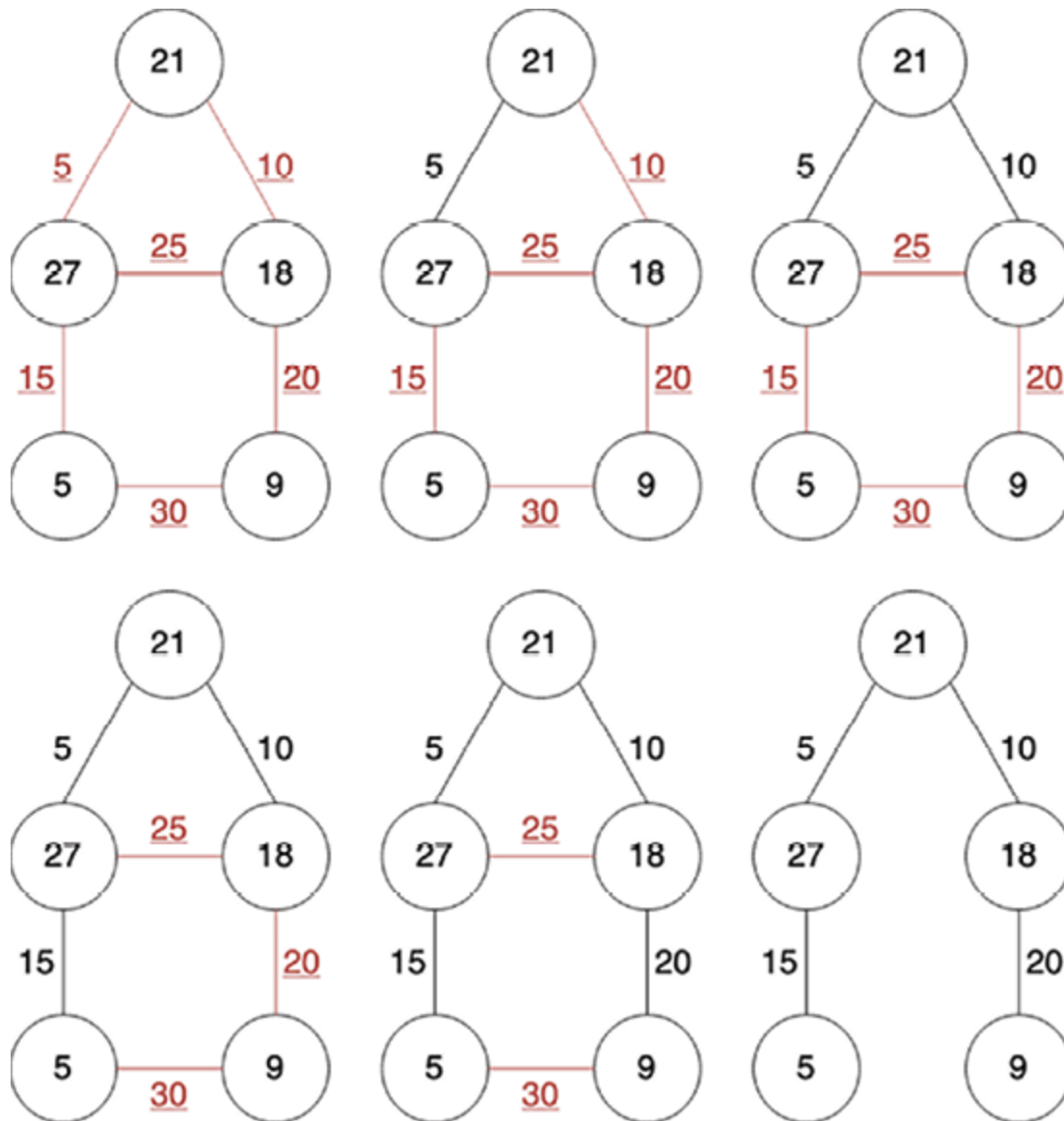


Figure 7.12: Kruskal's algorithm

We will need a priority queue to implement Kruskal's algorithm. We can modify one implemented in [Chapter](#) Stack and Here, we will use `WeightedEdge` instead of integer, and weight will determine priority. The edge

with a minimal weight value is prioritized. Here is a complete code for the priority queue:

```
type Element struct {  
  
    value WeightedEdge  
  
}  
  
type PriorityQueue []Element  
  
func (pq PriorityQueue) Len() int {  
  
    return len(pq)  
  
}  
  
func (pq PriorityQueue) Less(i, j int) bool {  
  
    return pq[i].value.weight < pq[j].value.weight  
  
}  
  
func (pq PriorityQueue) Swap(i, j int) {
```

```
if pq.Len() == 0 {  
  
    return  
  
}  
  
pq[i], pq[j] = pq[j], pq[i]  
  
}  
  
func (pq *PriorityQueue) Push(v any) {  
  
    element := Element{  
  
        value: v.(WeightedEdge),  
  
    }  
  
    *pq = append(*pq, element)  
  
}
```

```

func (pq *PriorityQueue) Pop() any {

    if pq.Len() == 0 {

        return -1

    }

    queue := *pq

    n := pq.Len() - 1

    element := queue[n]

    *pq = queue[0:n]

    return element

}

```

At the beginning of the function that implements Kruskal's algorithm, the initial set of MST edges will be empty, and we will put all edges of the weighted graph in the priority queue. Each node will represent one

unconnected component. This will be described as a map, where a key is an integer, and a value is an array of nodes.

The number of edges in MST is one less than the number of nodes, so we will iterate until this condition is met or there are no more edges that can be considered for inclusion in MST. Each iteration picks the edge with the smallest weight from the queue.

Suppose an edge connects two separated components (their nodes are not part of the same component). In that case, it will be added to MST, and unconnected components will be merged (this is represented with `append()` and `delete()` functions). At the end, the MST edge counter is incremented. The following code segment implements Kruskal's algorithm:

```
func Kruskal(wg *WeightedGraph) *MST {  
  
    treeEdges := make(map[WeightedEdge]struct{ })  
  
    pq := make(PriorityQueue, 0)  
  
    for edge := range wg.edges {
```



```
    heap.Push(&pq, edge)

}

// forest

forest := make(map[int][]Node)

i := 0

for node := range wg.nodes {

    forest[i] = append(forest[i], node)

    i++

}

for n := 0; n < len(wg.nodes)-1 || pq.Len() == 0; {

    edge := heap.Pop(&pq).(Element).value

    i := findInForest(forest, edge.u.value)
```

```

    j := findInForest(forest, edge.v.value)

    if i != j {

        treeEdges[edge] = struct{}{}

        forest[i] = append(forest[i],
                           forest[j]...)

        delete(forest, j)

        n++

    }

}

return &MST{wg.nodes, treeEdges}

}

```

The function `findInForest()` will locate a specified node in the forest by iterating through the map that represents it and through node arrays, as shown:

```
func findInForest(forest map[int][]Node, node int) int {  
  
    for i, v := range forest {  
  
        // Check all nodes in the tree  
  
        for _, n := range v {  
  
            if n.value == node {  
  
                return i  
  
            }  
  
        }  
  
    }  
  
    return -1  
  
}
```

The following code segment will create a weighted graph from [Figure 7.10](#) and execute Prim's and Kruskal's algorithm:

```
func main() {  
  
    node27 := graph.NewNode(27)  
  
    node18 := graph.NewNode(18)  
  
    node21 := graph.NewNode(21)  
  
    node9 := graph.NewNode(9)  
  
    node5 := graph.NewNode(5)  
  
    node25 := graph.NewNode(25)  
  
    wg := graph.NewWeightedGraph()  
  
    wg.AddNode(node21)
```

```
wg.AddNode(node27)
```

```
wg.AddNode(node18)
```

```
wg.AddNode(node5)
```

```
wg.AddNode(node9)
```

```
wg.AddEdgee(node21, node27, 5)
```

```
wg.AddEdgee(node21, node18, 10)
```

```
wg.AddEdgee(node27, node18, 25)
```

```
wg.AddEdgee(node27, node5, 15)
```

```
wg.AddEdgee(node18, node9, 20)
```

```
wg.AddEdgee(node5, node9, 30)
```

```
mstPrim := graph.Prim(wg, &node21)
```

```
mstPrim.Print()
```

```
mstKruskal := graph.Kruskal(wg)
```

```
mstKruskal.Print()
```

```
}
```

Transitive closure

It is often necessary to check if one node is reachable from another node in the graph (there is a path between them). Reachability matrix P is a matrix in which elements have the following values:

$p[i, j]$ = if a path exists between nodes i and

$p[i, j]$ = if no paths exist between nodes i and

The reachability matrix is called the transitive closure of a graph.

To calculate the reachability matrix first, we need an adjacency matrix a square matrix indicating whether a pair of nodes are adjacent (or not) in the graph. If nodes are adjacent, $a[i, j] = 1$ otherwise, $a[i, j] = 0$

To make everything as simple as possible, we will mark nodes starting from 0 so element $a[0, 2]$ will represent adjacency between nodes 0 and 2. In [Figure](#) we can see a directed graph with a corresponding adjacency matrix:

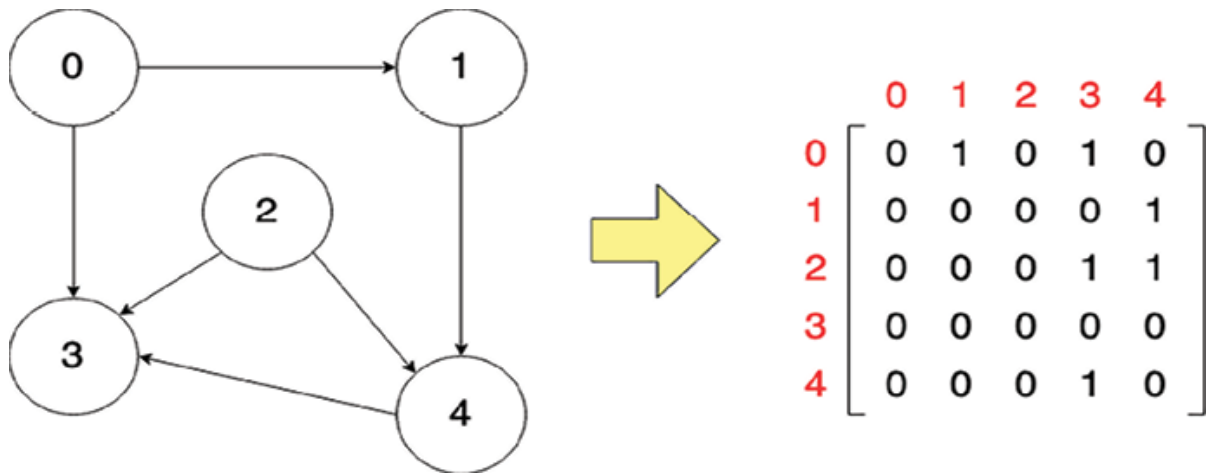


Figure 7.13: Adjacency matrix

Warshall's algorithm (defined by American computer scientist Stephen starts from an adjacency matrix and iteratively checks if the path between two nodes and can be established through any other node The complexity of this algorithm is

To simplify implementation, we will use a Boolean matrix, where the value true will imitate integer value 1, while the value false will take the role of integer value 0:

```
func Warshall(a [][]bool) (p [][]bool) {
```

```
    p = a
```

```
    for k := 0; k < len(p); k++ {
```



```

for i := 0; i < len(p); i++ {

    for j := 0; j < len(p); j++ {

        p[i][j] = p[i][j] ||

            (p[i][k] && p[k][j])

    }

}

return

}

```

The following code segment will calculate transitive closure for the graph presented in [Figure](#)

```

func main() {

```

```

a := [][]bool{

    {false, true, false, true, false},

    {false, false, false, false, true},

    {false, false, false, true, true},

    {false, false, false, false, false},

    {false, false, false, true, false},

}

p := graph.Warshall(a)

fmt.Println(p)

}

```

Transitive closure is mainly calculated for directed graphs, but it can also be calculated for undirected ones. In that case, the matrix will be symmetric, so if we calculate a value for $p[i, j]$, element $p[j, i]$ will have the same value. If the directed graph contains cycles, some elements on the main diagonal $p[i, i]$ where i is equal to i will have a value of 1.

Shortest paths

Transitive closure does not check the weight of edges; it just checks if there is a path between nodes. Often, it is crucial to find the shortest path between nodes (a path with minimal weight).

If $G = (V, E)$ is a directed weighted graph, for the path $p = v_1, v_2, \dots, v_k$, path weight is defined as the sum of the weights of the edges on that path:

$$w_{p_{1k}} = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

The shortest path between two nodes i and j is defined as:

$d(i,j) = \min\{w(p)\}$, the path between i and j with lowest weight.

∞ (infinite) if node j is not reachable from node i

Theoretically, some edges can have negative weight, but some algorithms do not support that.

When finding the shortest path between all pairs of nodes is necessary, it is suitable to use Floyd's algorithm (published by American computer scientist Robert Floyd). Input for this algorithm is a cost matrix W , in which elements have the following values:

$w[i, j] = c$ if i equals j

$w[i, j] = c$ is equal to the weight of the edge that connects i and j if i is not equal to j and the edge belongs to the set of edges.

$w[i, j] = \infty$ if i is not equal to j and the edge that connects nodes i and j does not belong to the set of edges.

The algorithm produces distance matrix D as output, where elements have the following values:

$d[i, j]$ equal to the weight of the shortest path (path with the smallest weight) between nodes i and j

$d[i, j] = \infty$ if there are no paths between i and j

In [Figure](#) we can see a weighted graph with a cost matrix:

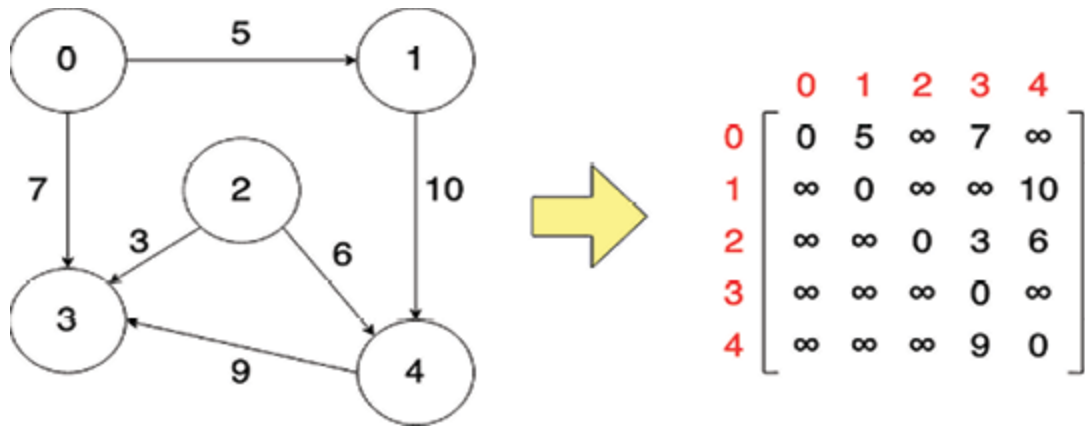


Figure 7.14: Cost matrix

Floyd's algorithm

Floyd's algorithm is a relaxation algorithm. The upper limit of the estimated path distance is maintained at every moment; initial values are equal to the weight of the edge that connects nodes or infinite if there is no edge. In subsequent iterations, whether the current estimation can be reduced with the path that passes through another internode is checked.

The following code snippet implements Floyd's algorithm:

```
func [][]int) (d [][]int) {  
  
    d = w  
  
    for k := 0; k < len(d); k++ {  
  
        for i := 0; i < len(d); i++ {  
  
            for j := 0; j < len(d); j++ {
```

```

        if d[i][j] > d[i][k]+d[k][j] {

d[i][j] = d[i][k] + d[k][j]

        }

    }

}

return

}

```

The following code segment will create a cost matrix and calculate shortest paths for the graph in [Figure](#)

```

func main() {

    const INF = 99999

```

```

w := [][]int{

    {0, 5, INF, 7, INF},

    {INF, 0, INF, INF, 10},

    {INF, INF, 0, 3, 6},

    {INF, INF, INF, 0, INF},

    {INF, INF, INF, 9, 0},

}

d := graph.Floyd(w)

fmt.Println(d)

}

```

We defined an integer constant inside the main() function with a high value to simulate an infinite value.

The eccentricity of the node is defined as the maximum distance between that node and other nodes in the

graph:

The center of the graph is a node with minimum eccentricity ().

Dijkstra's algorithm

We can use Dijkstra's algorithm to find the shortest path between one node and all others in the graph. It accepts the cost matrix as an input but does not support negative weights. Dutch computer scientist Edsger Wybe Dijkstra conceived it.

The working principle of the algorithm is based on maintaining the set that stores nodes for which the absolute shortest distance from the starting node has been found up to that moment. The algorithm also maintains a set where the remaining nodes (differences between the initial set of nodes and nodes for which the shortest distance was found) are stored for which there are current estimates of the shortest distances from the starting node. Still, these can be changed later.

Current evaluations of the shortest paths are stored inside vector. When the algorithm finishes, that vector will hold the shortest paths between the selected starting node and all other nodes.

Let us make things a little bit more interesting. We can change the graph used in the previous example so all

nodes can be reachable from node 0. The new graph and cost matrix are presented in [Figure](#)

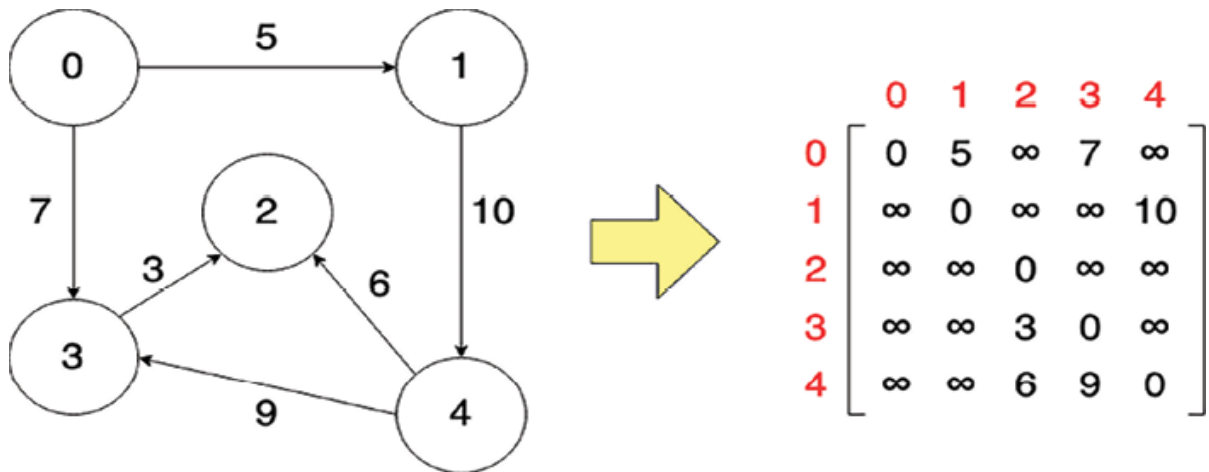


Figure 7.15: Graph and cost matrix when all nodes are reachable from node 0

The function that implements the algorithm will receive the starting node and cost matrix. Initially, it will initialize two sets and a vector (we will use the map to implement it). The algorithm will update the vector until set v is empty:

```
func Dijkstra(start int, w [][]int) (d map[int]int) {
    n := len(w)

    s := make(map[int]struct{})
```

```
s[start] = struct{}{}
```

```
v := make(map[int]struct{})
```

```
d = make(map[int]int)
```

```
for i := 0; i < n; i++ {
```

```
    if i != start {
```

```
        v[i] = struct{}{}
```

```
        d[i] = w[start][i]
```

```
    }
```

```
}
```

```
for len(v) != 0 {
```

```
    i := findMin(d, v)
```

```
    s[i] = struct{}{}
```

```

delete(v, i)

for j := range v {

    if d[i]+w[i][j] < d[j] {

        d[j] = d[i] + w[i][j]

    }

}

}

return

}

```

Function findMin() will return a node that is still in the set of remaining nodes with the lowest estimation for the shortest path. Again, we will use a high integer value to simulate an infinite value:

```

func findMin(d map[int]int, v map[int]struct{}) (node int) {

```

```

min := INF

for i := range d {

    _, ok := v[i]

    if d[i] <= min && ok {

node = i

        min = d[i]

    }

}

return

}

```

The following main() function will execute the algorithm for the graph in [Figure](#)

```

func main() {

```

```

w := [][]int{

    {0, 5, INF, 7, INF},

    {INF, 0, INF, INF, 10},

    {INF, INF, 0, INF, INF},

    {INF, INF, 3, 0, INF},

    {INF, INF, 6, 9, 0},

}

d := graph.Dijkstra(0, w)

fmt.Println(d)

}

```

For the end, in [Table](#) we can see how set S and vector d are changed through each step of Dijkstra's algorithm:

Table 7.1: Steps of Dijkstra's algorithm.

Flow in graphs

Flow network $G = (V, E)$ is defined as a directed graph where each edge has a non-negative weight called capacity. In this type of graph, there are two specific nodes:

Source which has no input edges.

Target which has no output edges.

The common problem is to find the maximum flow in the network as the largest amount that can be transferred from the source to the target. For example, liquid flow in the water pipe system. This can be simulated with flow networks.

The function $f(u, v)$ that represents the current flow between nodes u and v is defined to fulfill the following constraints:

Capacity constraint: current flow cannot exceed capacity, $f(u, v) \leq c(u, v)$ for all edges.

Symmetry constraint: flow between nodes u and v equals the negation of the flow between nodes v and u , $f(u, v) = -f(v, u)$ for all edges.

Flow conservation constraint: Incoming flow equals the outgoing flow, except for source and target. This means that the total node flow is zero:

$$\sum_{v \in V} f(u, v) = 0, u \in V$$

The algorithm determining the maximum flow in the graph starts from zero initial flow in edges and iteratively tries to increase it until the maximum flow is reached. A residual network consists of edges whose capacity is not fully utilized (current flow can be increased). This potential increase is known as residual capacity

= -

In a residual network, residual capacity for all edges is a positive value. If there is a path between the source and target where all edges have positive residual capacity,

flow can be increased. That path is called the augmenting path. The maximal flow increase on the augmenting path (residual path capacity) is equal to the smallest residual capacity of the edge on the path:

$$= \min_{e \in P} c_e - f_e$$

Ford-Fulkerson algorithm

The flow maximization problem can be solved with the Ford-Fulkerson algorithm, published by American mathematicians Lester Randolph Ford and Delbert Ray In [Figure](#) we can see an illustration of the algorithm. As mentioned, it starts with zero initial flow and finds the residual capacity of the first augmenting path 1, 2, When the flow is updated for all edges on the path, the algorithm will find another augmenting path 3, and add its capacity to the maximal flow. There are no augmenting paths, and the algorithm will return the calculated value

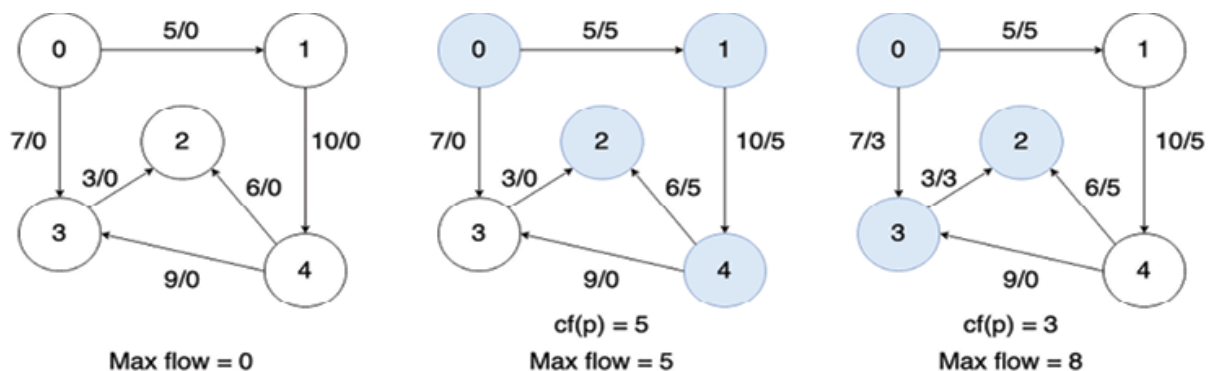


Figure 7.16: Ford-Fulkerson algorithm

To implement the Ford-Fulkerson algorithm, we must implement a flow network. It resembles a regular graph,

but the edges must support capacity and flow. Here is an implementation:

```
type Node struct {
```

```
    value int
```

```
}
```

```
type FlowEdge struct {
```

```
    u, v    Node
```

```
    capacity int
```

```
    flow    int
```

```
}
```

```
type FlowGraph struct {
```

```
    nodes map[Node]struct{}
```

```
    edges map[FlowEdge]struct{}
```

```
}
```

```
func NewFlowGraph() *FlowGraph {
```

```
    return &FlowGraph{
```

```
        nodes: make(map[Node]struct{}),
```

```
        edges: make(map[FlowEdge]struct{}),
```

```
    }
```

```
}
```

```
func (fg *FlowGraph) AddNode(n Node) {
```

```
    fg.nodes[n] = struct{}{}
```

```
}
```

```
func (fg *FlowGraph) AddEdge(u, v Node, c int) {
```

```
    e := FlowEdge{u, v, c, 0}
```

```

    fg.edges[e] = struct{}{}

}

func (fg *FlowGraph) RemoveNode(n Node) {

    delete(fg.nodes, n)

    for e := range fg.edges {

        if e.u == n || e.v == n {

            delete(fg.edges, e)

        }

    }

}

```

The FordFulkerson() function implemented algorithm. After initialization, in each iteration, the algorithm will find the augmenting path, capacity for that path, and update flow for each node. It accepts graph, source node, and target node as arguments:

```
func FordFulkerson(graph *FlowGraph, source,
target *Node) (maxFlow int) {

    for e := range graph.edges {

        e.flow = 0

    }

    ok, path := AugmentingPath(graph, source, target)

    for ok {

        cf := pathFlow(path)

        for _, edge := range path {

            delete(graph.edges, edge)

            edge.flow += cf

            graph.edges[edge] = struct{}{}

        }

    }

}
```



```
}
```

```
maxFlow += cf
```

```
ok, path = AugmentingPath(graph, source, target)
```

```
}
```

```
return
```

```
}
```

Simple function `pathFlow()` accepts augmenting path as argument and returns its capacity (minimal residual capacity for edges in the path):

```
func pathFlow(path []FlowEdge) (flow int) {
```

```
    flow = INF
```

```
    for _, edge := range path {
```

```
        cf := edge.capacity - edge.flow
```

```
    if cf < flow {  
  
        flow = cf  
  
    }  
  
}  
  
return  
  
}
```

The function that returns the augmenting path modifies the breadth-first search algorithm. The change is reflected in reconstructing the path that crosses the nodes, besides traversal. When the target node is reached, we can stop the algorithm's execution and return the path and boolean value of true. If a target node is not reached, a value false will be returned.

The edge will be added to the path if its source node is not already in the path as the source of some edge from the path and its source node is in the path as a destination of some edge from the path. Additional functions for these checks are also provided in the following code segment:

```
func AugmentingPath(g *FlowGraph, source *Node,  
  
    target *Node) (ok bool, path []FlowEdge) {  
  
    ok = false  
  
    visit := make(map[int]bool)  
  
    for n := range g.nodes {  
  
        visit[n.value] = false  
  
    }  
  
    visit[source.value] = true  
  
    queue := NewQueue()  
  
    queue.Enqueue(source)  
  
    for !queue.IsEmpty() {  
  
        u := queue.Dequeue()
```

```
for edge := range g.edges {  
  
    if edge.u.value == u.value &&  
  
        !visit[edge.v.value] &&  
  
        edge.capacity != edge.flow {  
  
            if len(path) == 0 ||  
  
                !inPath(edge.u.value, path) &&  
  
inPathDest(edge.u.value, path) {  
  
                visit[edge.v.value] = true  
  
                path = append(path, edge)  
  
n := edge.v  
  
                queue.Enqueue(&n)  
  
            if edge.v.value == target.value {
```

```

        ok = true

        return

    }

}

}

}

}

}

return

}

func inPath(node int, path []FlowEdge) bool {

    for _, edge := range path {

        if edge.u.value == node {

            return true

        }

    }

}

```

```
}
```

```
}
```

```
return false
```

```
}
```

```
func inPathDest(node int, path []FlowEdge) bool {
```

```
    for _, edge := range path {
```

```
        if edge.v.value == node {
```

```
            return true
```

```
        }
```

```
    }
```

```
return false
```

```
}
```

For the end of this section, we will show the `main()` function that will create a graph from [Figure 7.16](#) and calculate maximal flow:

```
func main() {  
  
    node0 := graph.NewNode(0)  
  
    node1 := graph.NewNode(1)  
  
    node2 := graph.NewNode(2)  
  
    node3 := graph.NewNode(3)  
  
    node4 := graph.NewNode(4)  
  
    g := graph.NewFlowGraph()  
  
    g.AddNode(node0)  
  
    g.AddNode(node1)  
  
    g.AddNode(node2)
```

```
g.AddNode(node3)
```

```
g.AddNode(node4)
```

```
g.AddEdge(node0, node1, 5)
```

```
g.AddEdge(node0, node3, 7)
```

```
g.AddEdge(node1, node4, 10)
```

```
g.AddEdge(node3, node2, 3)
```

```
g.AddEdge(node4, node2, 6)
```

```
g.AddEdge(node4, node3, 9)
```

```
maxFlow := graph.FordFulkerson(g, &node0, &node2)
```

```
fmt.Println(maxFlow)
```

```
}
```


Topological sorting

Complex projects consisting of mutually conditioned activities can be well simulated with graphs. Nodes represent events, while edges represent certain activities. When the activities cannot be performed in parallel because, for example, there is only one person to do them, the project lasts as long as the sum of the duration of all activities. Activities must be performed in the order dictated by their interdependence. This order can be obtained by topological sorting.

If $G = (V, E)$ is a directed acyclic graph, then the topological sorting of this graph is the determination of a linear project of all nodes so that for each edge, node u appears before the node v in this order.

The output of the topological sorting algorithm is a vector T in which the nodes are arranged according to the found topological order. The algorithm iteratively repeats the following activities:

Find the node that does not have a predecessor (indegree is zero). If there are multiple such nodes, anyone can be taken; topological order does not need to be unique.

Include the node in topological order.

Removes a node from the graph with all branches coming out of it.

The algorithm is based on the acyclic graph property that there is at least one node with no predecessor; otherwise, the graph would have a cycle. In [Figure](#) we can see the iteration of the topological sorting algorithm:

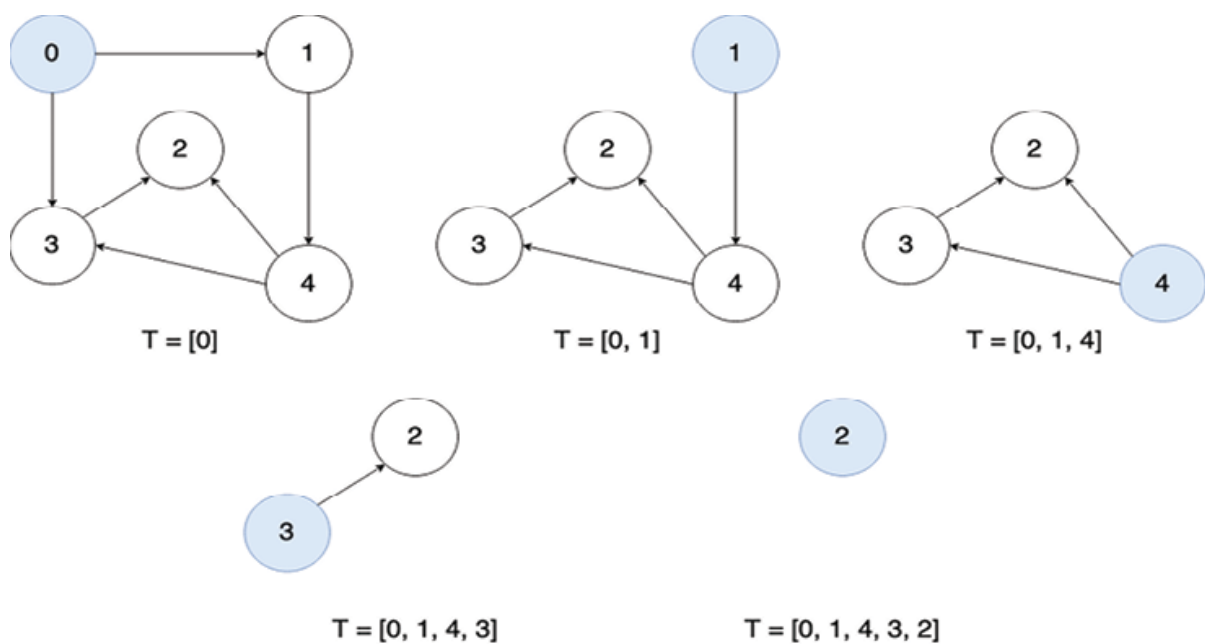


Figure 7.17: Topological sorting algorithm

The following function will implement a topological sorting algorithm:

```
func TopSort(graph *Graph) []Node {  
  
    nodes := graph.nodes  
  
    edges := graph.edges  
  
    n := len(nodes)  
  
    t := make([]Node, n)  
  
    for i := 0; i < n; i++ {  
  
        zeroIndegreeNode :=  
  
        findZeroIndegreeNode(nodes, edges)  
  
        t[i] = zeroIndegreeNode  
  
        delete(nodes, zeroIndegreeNode)  
  
        for edge := range edges {
```

```

        edge.u.value == zeroIndegreeNode.value {

            delete(edges, edge)

        }

    }

}

return t

}

```

Function findZeroIndegreeNode() will search for a node that is not a target for any edge. The function will assume that the provided graph is acyclic, so a situation when empty not is returned should never happen:

```

func findZeroIndegreeNode(nodes map[Node]struct{},
    edges map[Edge]struct{ }) Node {

    for node := range nodes {

```

```
isZeroDegree := true
```

```
for edge := range edges {
```

```
    if edge.v.value == node.value {
```

```
        isZeroDegree = false
```

```
    }
```

```
}
```

```
if isZeroDegree {
```

```
    return node
```

```
}
```

```
}
```

```
return Node{}
```

```
}
```

The main() function will create a graph and execute the topological sorting algorithm, as illustrated in [Figure](#)

```
func main() {  
  
    node0 := graph.NewNode(0)  
  
    node1 := graph.NewNode(1)  
  
    node2 := graph.NewNode(2)  
  
    node3 := graph.NewNode(3)  
  
    node4 := graph.NewNode(4)  
  
    tsGraph := graph.New()  
  
    tsGraph.AddNode(node0)  
  
    tsGraph.AddNode(node1)  
  
    tsGraph.AddNode(node2)  
  
    tsGraph.AddNode(node3)
```

```
tsGraph.AddNode(node4)
```

```
    tsGraph.AddEdge(node0, node1)
```

```
    tsGraph.AddEdge(node0, node3)
```

```
    tsGraph.AddEdge(node1, node4)
```

```
    tsGraph.AddEdge(node3, node2)
```

```
    tsGraph.AddEdge(node4, node2)
```

```
    tsGraph.AddEdge(node4, node3)
```

```
    t := graph.TopSort(tsGraph)
```

```
    fmt.Println(t)
```

```
}
```

Critical path

In the graph that simulates the organization of activities in some projects, it is essential to find the critical activities and the shortest possible duration of the project. For that purpose, the method of finding the critical path is used.

In the directed acyclic graph that models a project, the weight of each edge represents the number of time units required to perform the given activity. The node with zero indegree simulates the start of the project (source), while the node with zero outdegree simulates the end of the project (target).

In such graphs, usually, there are multiple paths between the source and the target. As the project cannot be completed earlier than required by the most extended series of dependent activities, the project's duration is determined by the maximum length between the source and the target. This path is called a critical while activities on that path are critical activities.

To calculate the critical path's length for each node, the earliest start time must be calculated. The earliest start time is the earliest time when an activity, simulated with output edges for that node, can start. No activities on the critical path can be delayed; every delay increases the project's duration. This means that the earliest start time must also be the latest start time for the activity.

The algorithm will calculate each node's earliest and latest start time. In the following formulas, $P(i)$ represents a set of predecessors while $S(i)$ represents a set of successors:

$$= + j \in$$

$$= - j \in$$

In [Figure](#) we can see a weighted graph that will be used to illustrate the algorithm for the critical path. The algorithm will start by executing a topological sorting. For each node, it will find predecessors and successors and calculate values for EST and LST. Additionally, the algorithm will produce a latency vector with the following formula:

$$L[i] = LST[i] - EST[i]$$

The activities where latency equals zero are critical activities, which are part of the critical path. Let us take a look at the weighted graph referenced in the following figure:

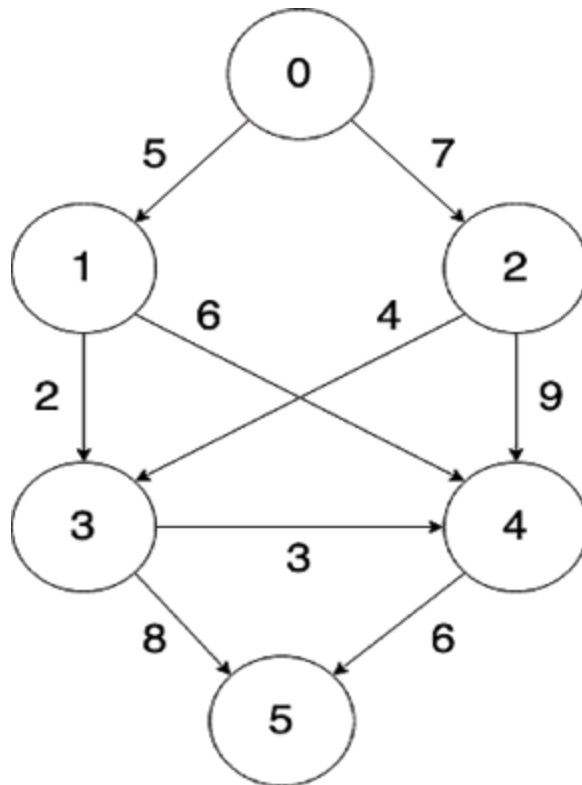


Figure 7.18: Weighted graph

We illustrated a topological sorting algorithm on the regular graph in the previous section. For this usage, we must adapt it to a directed graph. Also, the map will be passed by reference, so when we remove a node or edge, it will affect the graph. So, we should make a copy for each map. Here is the modified topology sorting function:

```
func TopSortWG(graph WeightedGraph) []Node {  
  
    nodes := make(map[Node]struct{})  
  
    edges := make(map[WeightedEdge]struct{})  
  
    for k, v := range graph.nodes {  
  
        nodes[k] = v  
  
    }  
  
    for k, v := range graph.edges {  
  
edges[k] = v
```

```
}
```

```
n := len(nodes)
```

```
t := make([]Node, n)
```

```
for i := 0; i < n; i++ {
```

```
    zeroIndegreeNode :=
```

```
        findZeroIndegreeNodeWG(nodes, edges)
```

```
    t[i] = zeroIndegreeNode
```

```
delete(nodes, zeroIndegreeNode)
```

```
for edge := range edges {
```

```
    if edge.u.value == zeroIndegreeNode.value  
    {
```

```
        delete(edges, edge)
```

```

    }

    }

    }

    return t

}

```

The following function accepts the graph as an argument and returns three vectors (EST, LST, and L). EST for the source node is zero, while LST for the target node is equal to the EST of the targeting node:

```

func CriticalPath(graph WeightedGraph) ([]int, []int,
[]int) {

    t := TopSortWG(graph)

    n := len(t)

    est := make([]int, n)

```

```
est[0] = 0
```

```
for i := 1; i < n; i++ {
```

```
    k := t[i].value
```

```
    est[k] = findMax(k, est, graph.edges)
```

```
}
```

```
lst := make([]int, n)
```

```
lst[n-1] = est[n-1]
```

```
for i := n - 2; i >= 0; i-- {
```

```
    k := t[i].value
```

```
    lst[k] = findMinLST(k, lst, graph.edges)
```

```
}
```

```
l := make([]int, n)
```

```

for i := 0; i < n; i++ {

    l[i] = lst[i] - est[i]

}

return est, lst, l

}

```

The function findMax() will find all predecessors for the provided node and calculate the EST value for that node using the formula provided in this section:

```

func findMax(nodeValue int, est []int,

    edges map[WeightedEdge]struct{ }) int {

    max := -1

    // find predecessors

    predecessors := make(map[int]int)

```

```
for edge := range edges {  
  
    if edge.v.value == nodeValue {  
  
predecessors[edge.u.value] = edge.weight  
  
    }  
  
}  
  
// find maximum  
  
for p, v := range predecessors {  
  
    if est[p]+v > max {  
  
        max = est[p] + v  
  
    }  
  
}
```



```
    return max  
  
}
```

The function findMinLST() will find all successors for the provided node and calculate the LST value for that node using the previously provided formula:

```
func findMinLST(nodeValue int, lst []int,  
  
    edges map[WeightedEdge]struct{ }) int {  
  
    min := INF  
  
    // Find successors  
  
    successors := make(map[int]int)  
  
    for edge := range edges {  
  
        if edge.u.value == nodeValue {  
  
            successors[edge.v.value] = edge.weight
```

```

    }

}

// Find minimum

for p, v := range successors {

    if lst[p]-v < min {

        min = lst[p] - v

    }

}

return min

}

```

The following main() function will create a graph from the [Figure 7.18](#) and calculate all vectors:

```
func main() {  
  
    node0 := graph.NewNode(0)  
  
    node1 := graph.NewNode(1)  
  
    node2 := graph.NewNode(2)  
  
    node3 := graph.NewNode(3)  
  
    node4 := graph.NewNode(4)  
  
    node5 := graph.NewNode(5)  
  
    wg := graph.NewWeightedGraph()  
  
    wg.AddNode(node0)  
  
    wg.AddNode(node1)  
  
    wg.AddNode(node2)  
  
    wg.AddNode(node3)
```

```
wg.AddNode(node4)
```

```
wg.AddNode(node5)
```

```
wg.AddEdgee(node0, node1, 5)
```

```
wg.AddEdgee(node0, node2, 7)
```

```
wg.AddEdgee(node1, node3, 2)
```

```
wg.AddEdgee(node1, node4, 6)
```

```
wg.AddEdgee(node2, node3, 4)
```

```
wg.AddEdgee(node2, node4, 9)
```

```
wg.AddEdgee(node3, node4, 3)
```

```
wg.AddEdgee(node3, node5, 8)
```

```
wg.AddEdgee(node4, node5, 6)
```

```
est, lst, l := graph.CriticalPath(wg)

fmt.Println(est, lst, l)

}
```

[Table 7.2](#) shows EST, LST, and L for each node and their predecessors and successors:

Table 7.2: Values calculated with critical path algorithm

As we can see, the following nodes form a critical path: 0, 2, 4, and 5. Based on data from [Table](#) we can calculate the allowed latency for each edge with the following formula:

$$l(i, j) = LST[j] - EST[i] - w(i, j)$$

Conclusion

This chapter introduced the second non-linear data structure, graphs, with related operations. We learned how to implement them with the Go programming language and presented several traversal algorithms. Some other valuable concepts of graphs, like spanning trees, transitive closure, and shortest path, were also presented. We saw how to find a maximum flow in a graph and topologically sort the graph. At the end of the chapter, we learned how to find a critical path and determine critical activities in our projects.

A lot of algorithms are presented in this chapter. [Table 7.3](#) is a small recapitulation, with complexity for each algorithm represents the number of nodes, while e represent the number of edges) and when it is used:

Table 7.3: Graph algorithms

This chapter concludes our journey through the enlightening world of data structures and algorithms. Through this book, we introduced the most common

data structures that are used for solving everyday practical problems. Some are used more often than others (trees and graphs). In the standard web service, where functionalities offered to users are logging into the system, reading, creating, editing, or deleting some data, arrays, slices, and maps will be used more often. Trees and graphs are used for complex problems such as simulations or artificial intelligence algorithms.

The algorithms presented in this book are basic ones. Most of the modern solutions use or are based on them. The implementations shown here are not 100% optimal, but they are presented as close as possible to the original idea and design. Feel free to improve them and implement them differently.

Points to remember

The Go programming language does not have a graph implementation.

Traversing the graph creates a traversal tree.

The spanning tree is not unique.

Transitive closure does not check the weight of edges. It just checks for a path between nodes.

Dijkstra's algorithm does not support negative weights.

In the residual network, residual capacity for all edges is a positive value.

Complex projects consisting of mutually conditioned activities can be well simulated with graphs.

The topological order does not need to be unique.

In the graph that simulates the organization of activities in a project, it is essential to find the critical activities and the shortest possible duration of the project.

Multiple choice questions

In how many categories do traversal algorithms classify graph edges?

2

3

4

5

Who developed Prim's algorithm?

Robert Clay Prim III

Joseph Bernard Kruskal Jr.

Stephen Warshall

Vojtech Jarnik

Which matrix is an output of the Wharshall's algorithm?

Adjacency matrix

Reachability matrix

Cost matrix

Distance matrix

Which matrix is an input for the Floyd's algorithm?

Adjacency matrix

Reachability matrix

Cost matrix

Distance matrix

Answers

C

D

B

C

Questions

What is the degree of a node?

What are the differences between indegree and outdegree?

What are the differences between multigraphs and simple graphs?

Which operations can be performed on the graph?

Which are the two popular traversal algorithms?

Which are the two popular algorithms for finding an MST?

What is the augmenting path?

How can the maximum flow in the graph be calculated?

How to perform the topological sorting on the graph?

Key terms

A pair of sets where V is a finite, nonempty set, while E represents binary relations between elements from set

An operation where all graph nodes are visited only once in a systematic order, and some processing is performed on the visited node.

Spanning The tree created from a graph contains all nodes and certain edges necessary to connect all nodes without cycles.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Index

A

algorithmic complexity [12](#)

algorithmic complexity, classes

constant algorithms $O(1)$ [12](#)

exponential algorithms where $k > 1$ [13](#)

linear algorithms $O(n)$ [13](#)

logarithmic algorithms $O(\log n)$ [13](#)

log-linear algorithms $O(n \log n)$ [13](#)

polynomial algorithms [13](#)

square algorithms [13](#)

algorithms

classification

combinatorial algorithm [12](#)

compression algorithm [12](#)

computational geometric algorithm [12](#)

cryptography algorithm [12](#)

distributed algorithm [11](#)

divide and conquer [10](#)

dynamic programming [10](#)

for numerical analysis [11](#)

fundamentals [8](#)

genetic algorithm [10](#)

graph algorithm [12](#)

greedy algorithm [10](#)

heuristic algorithm [11](#)

history [7](#)

iterative algorithm [11](#)

linear programming [10](#)

machine learning algorithm [12](#)

merge algorithm [11](#)

parallel algorithm [11](#)

parsing algorithm [12](#)

randomized algorithm [10](#)

recursive algorithm [11](#)

representation [9](#)

search algorithm [11](#)

sequential (serial) algorithm [11](#)

sorting algorithm [11](#)

string algorithm [12](#)

arrays [20](#)

in Go [21](#)

multidimensional arrays [24](#)

operations [20](#)

sorting, with tree

versus, lists [59](#)

augmenting path [147](#)

B

Big O-notation [12](#)

binary search algorithm

binary tree [98](#)

almost complete binary tree [98](#)

complete binary tree [98](#)

full binary tree [98](#)

operations

types [99](#)

breadth-first search traversal algorithm [128](#)

bubble sort [40](#)

C

capacity (c) [146](#)

capacity constraint [147](#)

circular list [62](#)

connected graph [123](#)

critical path [156](#)

calculating

D

data structures

characteristics [2](#)

dynamic [3](#)

fundamentals [2](#)

heterogenous [3](#)

homogenous [3](#)

in Go [6](#)

linear [3](#)

non-linear [3](#)

static [3](#)

defer statement [68](#)

degree [94](#)

dense graph [122](#)

depth-first search traversal algorithm [129](#)

Dijkstra's algorithm

double hashing [86](#)

double-linked list [61](#)

E

earliest start time (EST) [156](#)

edges [94](#)

equivalence class [82](#)

F

First In, First Out [71](#)

flow conservation constraint [147](#)

flow network, graph [147](#)

Ford-Fulkerson algorithm

Floyd's algorithm [143](#)

Ford-Fulkerson algorithm [147](#)

implementing

functions, Go

G

garbage [53](#)

Go

arrays [21](#)

data structures [6](#)

functions

maps

priority queue

searching algorithms [33](#)

slices [23](#)

sorting algorithms [45](#)

trees [101](#)

graph

complete graph [123](#)

dense graph [122](#)

flow [146](#)

fundamentals [120](#)

in Go [125](#)

multigraph [121](#)

operations [124](#)

simple graph [121](#)

sparse graph [122](#)

subgraph [121](#)

transitive closure [139](#)

types [121](#)

weighted graph [121](#)

Graph G [120](#)

graph traversal algorithms [126](#)

back edges [126](#)

breadth-first search [128](#)

cross edges [126](#)

depth-first search [129](#)

forward edges [126](#)

tree edges [126](#)

H

hash collision [85](#)

hash function [82](#)

digit folding method [84](#)

division method [83](#)

mid-square method [83](#)

multiplication method [83](#)

perfect hash [84](#)

radix conversion method [84](#)

hashing [82](#)

head [50](#)

I

inorder traversal algorithm [107](#)

insertion sort

interface

J

Jarnik's algorithm [133](#)

K

key [28](#)

Kruskal's algorithm

L

Last In, First Out (LIFO) [66](#)

latency vector (L) [157](#)

level-order algorithm

linear probing [86](#)

lists [50](#)

circular list [50](#)

double-linked list [50](#)

operations

single-linked list [50](#)

versus, arrays [60](#)

lists, in Go

circular list [62](#)

double-linked list [61](#)

load factor (α) [85](#)

logarithmic search [32](#)

loop [121](#)

M

maps [87](#)

in Go

operations [87](#)

matrices [24](#)

memory representation [3](#)

linked [5](#)

sequential [4](#)

methods [26](#)

minEdge() function [134](#)

minimal-cost spanning tree (MST) [131](#)

multidimensional arrays [24](#)

multigraph [121](#)

N

node [50](#)

nodes, tree [94](#)

child node [95](#)

leaves [94](#)

parent node [95](#)

root [94](#)

P

parallel edges [121](#)

partition() function [43](#)

pivot [41](#)

postorder traversal algorithm [107](#)

preorder traversal algorithm [106](#)

Prim's algorithm [134](#)

priority queue [75](#)

in Go

max priority queue [75](#)

min priority queue [75](#)

Q

quadratic probing [86](#)

queue [71](#)

implementation [74](#)

operations

priority queue [75](#)

quick sort

R

random probing [86](#)

rehashing [85](#)

residual capacity (cf) [147](#)

S

Search() function [33](#)

searching algorithms

binary search

in Go [33](#)

sequential search

selection sort

sequential search algorithm

shortest path [142](#)

Dijkstra's algorithm

Floyd's algorithm [143](#)

simple graph [121](#)

simple path [122](#)

single-linked list [50](#)

implementing

slices [21](#)

in Go [23](#)

sorting algorithms

bubble sort [40](#)

in Go [45](#)

insertion sort

quick sort

selection sort

spanning tree

Kruskal's algorithm

Prim's algorithm [134](#)

sparse graph [122](#)

stack [67](#)

implementation [70](#)

in Go [69](#)

operations [68](#)

Stack Pointer (SP) [66](#)

subgraph [121](#)

subtrees [94](#)

symmetry constraint [147](#)

synonyms [82](#)

T

tail [50](#)

topological sorting algorithm

transitive closure

traversal algorithms [105](#)

inorder [107](#)

level-order

postorder [107](#)

preorder [106](#)

trees

array, sorting with

binary tree [98](#)

fundamentals [95](#)

memory representation [98](#)

nodes [94](#)

ordered trees [96](#)

structurally similar trees [96](#)

subtrees [94](#)

trees, in Go [101](#)

delete operation

insert operation [102](#)

V

visited node [126](#)

W

Warshall's algorithm [140](#)

weighted graph [121](#)