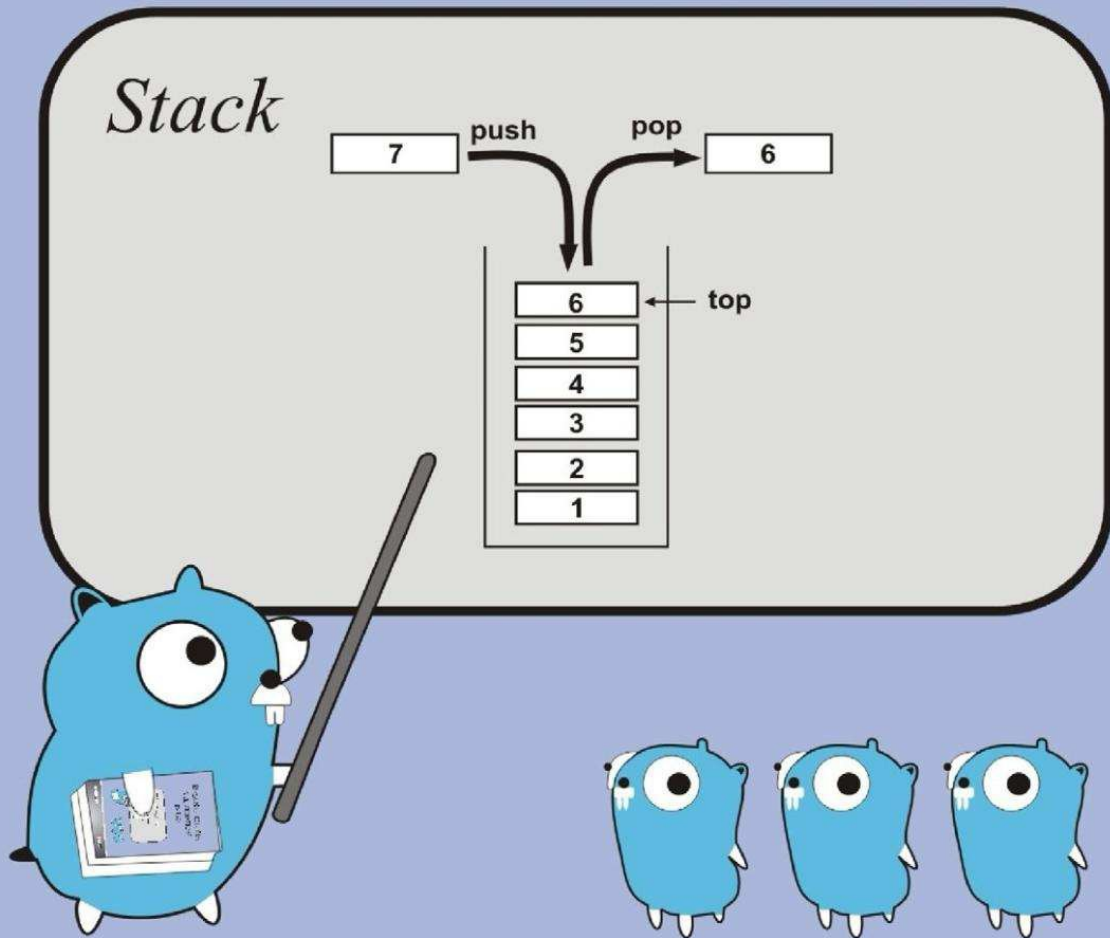
DATA STRUCTURES & ALGORITHMS IN GO



HEMANT JAIN

1st EDITION

DATA STRUCTURES
&
ALGORITHMS
IN GO

FIRST EDITION

HEMANT JAIN

Data Structures & Algorithms In Go
Hemant Jain

ACKNOWLEDGEMENT

The author is very grateful to GOD ALMIGHTY for his grace and blessing.

Deepest gratitude for the help and support of my brother Dr. Sumant Jain. This book would not have been possible without the support and encouragement he provided.

I would like to express profound gratitude to my guide/ my friend Naveen Kaushik for his invaluable encouragement, supervision and useful suggestion throughout this book writing work. His support and continuous guidance enable me to complete my work successfully.

Finally yet importantly, I am thankful to Love Singhal, Anil Berry and Others who helped me directly or indirectly in completing this book.

Hemant Jain

TABLE OF CONTENTS

CHAPTER 0: HOW TO USE THIS BOOK

What this book is about

This book introduces you to the world of data structures and algorithms. Data structure defines the way data is arranged in computer memory for fast and efficient access while algorithm is a set of instruction to solve problems by manipulating these data structures.

Designing an efficient algorithm is a very important skill that all computer companies e.g. Microsoft, Google, Facebook etc. pursue. Most of the interview for these companies is focused on knowledge of data structure and algorithm. They look for how candidate use these to solve complex problem efficiently, which is also very important in everyday coding. Apart from knowing, a programming language you also need to have good command on these key Computer fundamentals to not only qualify the interview but also excel in the top high paying jobs.

This book assumes that you are a Go language developer. You are not an expert in Go language, but you are well familiar with concepts of class, references, functions, list, tuple, dictionary and recursion. At the start of this book, we will be revising Go language fundamentals that will be used throughout this book. We will be looking into some of the problems in Lists and recursion too.

Then in the coming chapter we will be looking into Complexity Analysis. Followed by the various data structures and their algorithms. We will be looking into a Linked-List, Stack, Queue, Trees, Heap, Hash-Table and Graphs. We will also be looking into Sorting, Searching techniques.

We will be looking into algorithm analysis of various algorithm techniques. Such as, Brute-Force algorithms, Greedy algorithms, Divide and Conquer algorithms, Dynamic Programming, Reduction and Backtracking.

In the end, we will be looking into System Design that will give a systematic approach to solve the design problems.

Preparation Plans

Given the limited time you have before your next interview, it is important to have a solid preparation plan. The preparation plan depends upon the time and companies you are planning to target. Below are the three-preparation plan for 1 Month, 3 Month and 5 Month durations.

1 Month Preparation Plans

Below is a list of topics and approximate time user need to finish these topics. These are the most important chapters that must be prepared before appearing for an interview.

This plan should be used when you have a limited preparation time for an interview. These chapters cover 90% of data structures and algorithm based interview questions. In this plan since we are reading about the various ADT and Go collections (or built in data structures.) in chapter 4 so we can use these datatype easily without knowing the internal details how they are implemented.

Chapter 24 is for system design, you must read this chapter if you have three or more years of experience. Anyway, reading this chapter will give the reader a broader perspective of various designs.

Time	Chapters	Explanation
Week 1	Chapter 1: Introduction - Programming Overview Chapter 2: Algorithms Analysis Chapter 3: Approach To Solve Algorithm Design Problems Chapter 4: Abstract Data Type & Go Collections	You will get a basic understanding of how to find complexity of a solution. You will know how to handle new problems. You will read about a variety of datatypes and their uses.
Week 2	Chapter 5: Searching Chapter 6: Sorting Chapter 14: String Algorithms	Searching, Sorting and String algorithm consists of a major portion of the interviews.
Week 3	Chapter 7: Linked List Chapter 8: Stack Chapter 9: Queue	Linked list, Stack and Queue are one of the favorites in an interview.
Week 4	Chapter 10: Tree Chapter 23: Interview Strategy Chapter 24: System Design	This portion you will read about Trees and System Design. You are good to go for interviews. Best of luck.

3 Month Preparation Plan

This plan should be used when you have some time to prepare for an interview. This preparation plan includes nearly everything in this book except various algorithm techniques. Algorithm problems that are based on “dynamic programming”, “divide & conquer” etc. Which are asked in vary specific companies like Google, Facebook, etc. Therefore, until you are planning to face interview with them you can park these topics for some time and focus on the rest of the topics.

Again, same thing here with system design problems, the more experience you are, the more important this chapter becomes. However, if you are a fresher from college, then also you should read this chapter.

Time	Chapters	Explanation
Week 1	Chapter 1: Introduction Chapter 2: Algorithms Analysis Chapter 3: Approach To Solve Algorithm Design Problems Chapter 4: Abstract Data Type & Go Collections	You will get a basic understanding of how to find complexity of a solution. You will know how to handle new problems. You will read about a variety of datatypes and their uses.
Week 2 & Week 3	Chapter 5: Searching Chapter 6: Sorting Chapter 14: String Algorithms	Searching, sorting and string algorithm consists of a major portion of the interviews.

Week 4 & Week 5	Chapter 7: Linked List Chapter 8: Stack Chapter 9: Queue	Linked list, Stack and Queue are one of the favorites in an interview.
Week 6 & Week 7	Chapter 10: Tree Chapter 11: Heap	This portion you will read about trees and heap data structures.
Week 8 & Week 9	Chapter 12: Hash-Table Chapter 13: Graphs	Hash-Table is used throughout this book in various places, but now it is time to understand how Hash-Table are actually implemented. Graphs are used to propose a solution many real life problems.
Week 10	Chapter 23: Interview Strategy Chapter 24: System Design	Interview strategy and system design chapter are the final chapters of this course.
Week 11 & Week 12	Revision of the chapters listed above.	At this time, you need to revise all the chapters that we have seen in this book. Whatever is left needs to be completed and the exercise that may be left needing to be solved in this time

5 Month Preparation Plan

This preparation plan is made on top of 3-month plan. In this plan, students should look for algorithm design chapters. In addition, in the rest of the time they need to practice more and more from www.topcoder.com and other resources. If you are targeting google, Facebook, etc., Then it is highly recommended to join topcoder and practice as much as possible.

Time	Chapters	Explanation
Week 1 Week 2	Chapter 1: Introduction - Programming Overview Chapter 2: Algorithms Analysis Chapter 3: Approach To Solve Algorithm Design Problems Chapter 4: Abstract Data Type & Go Collections	You will get a basic understanding of how to find complexity of a solution. You will know how to handle new problems. You will read about a variety of datatypes and their uses.
Week 3 Week 4 Week 5	Chapter 5: Searching Chapter 6: Sorting Chapter 14: String Algorithms	Searching, sorting and string algorithm consists of a major portion of the interviews.
Week 6 Week 7 Week 8	Chapter 7: Linked List Chapter 8: Stack Chapter 9: Queue	Linked list, Stack and Queue are one of the favorites in an interview.
Week 9 Week 10	Chapter 10: Tree Chapter 11: Heap	This portion you will read about trees and priority queue.
Week 11 Week 12	Chapter 12: Hash-Table Chapter 13: Graphs	Hash-Table is used throughout this book in various places, but now it is time to understand how Hash-Table are actually implemented. Graphs are used to propose a solution many real life problems.
Week 13 Week 14 Week 15 Week 16	Chapter 15: Algorithm Design Techniques Chapter 16: Brute Force Chapter 17: Greedy Algorithm Chapter 18: Divide-And-Conquer, Decrease-And-Conquer Chapter 19: Dynamic Programming Chapter 20: Backtracking And Branch-And-Bound Chapter 21: Complexity Theory And Np Completeness	These chapters contain various algorithms types and their usage. Once the user is familiar with most of this algorithm. Then the next step is to start solving topcoder problems from topcoder.
Week 17 Week 18	Chapter 22: Interview Strategy Chapter 23: System Design	Interview strategy and system design chapter are the final chapters of this course.
Week 19 Week 20	Revision of the chapters listed above.	At this time, you need to revise all the chapters that we have seen in this book. Whatever is left needs to be completed and the exercise that may be left needing to be solved in this period.

Summary

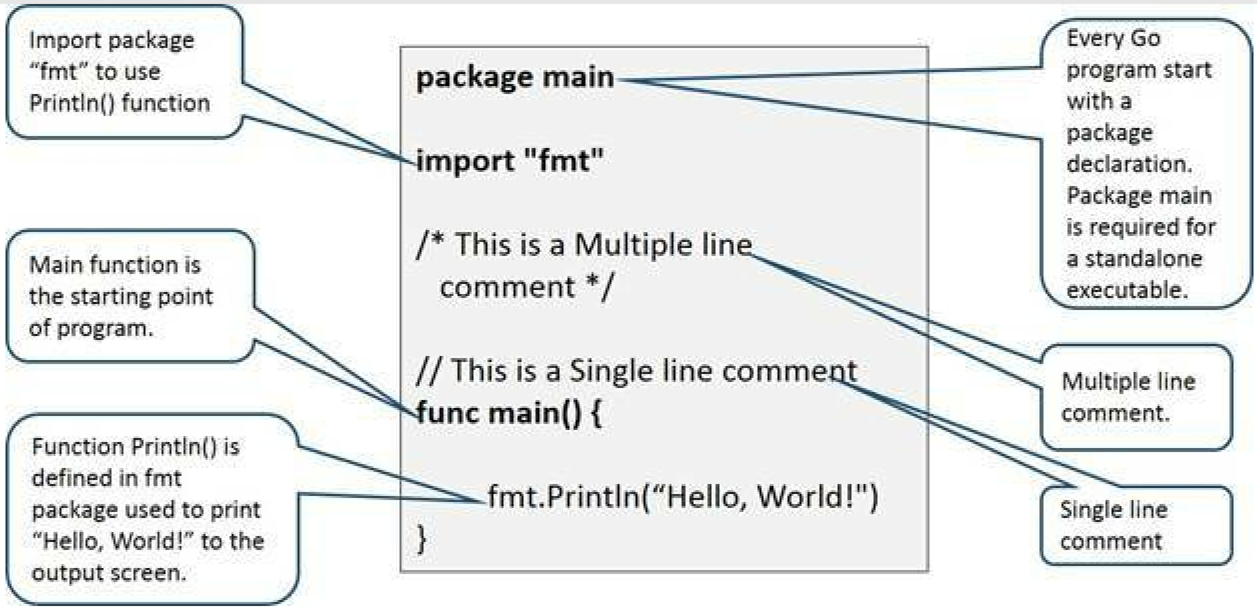
These are few preparation plans that can be followed to complete this book while preparing for the interview. It is highly recommended that you should read the problem statement, try to solve the problems by himself and then only you should look into the solution to find the approach of this book. Practicing more and more problems will increase your thinking capacity and you will be able to handle new problems in an interview. System design is a topic that is not asked much from a college fresher, but as you gain experience its importance increase. We will recommend practicing all the problems given in this book, then solve more and more problems from online resources like www.topcoder.com, www.careercup.com, www.geekforgeek.com etc.

CHAPTER 1: INTRODUCTION - PROGRAMMING OVERVIEW

Introduction

This chapter emphasizes on brush up of the fundamentals of the Go Programming language. It will talk about variables, pointer, structure, loops, recursion, array, slice, map etc. We assume that the reader is familiar with the syntax of the Go programming language and knows the basics of Object-Orientation.

First Go Program



It is tradition to discuss a HelloWorld program in the start which will print the phrase "Hello, World!" to the output screen. So let us start discussing it. This is a small program but it contains many common features of all the Go programs.

1. The Go language uses packages to organize the associated program code together, and the main() function must be in the main package

2. The Go language uses the import statement to include third-party modules. After importing the corresponding package, the program can use the functions / variables exported by the corresponding package. We had imported "fmt" package and will be using its Println() function.

3. The main() function is the entry point of the Go program. Every function definition starts with func keyword.

4. Comments are ignored by Go compiler. There are two types of comments in Go. Single line comments which start with // symbol and ends with end of line and Multiline comment which start with /* and ends with */ anything between two *'s are ignored by compiler.

Variables & Constants

"Variables" are simply storage locations for data. For every variable, some memory is allocated. The size of this memory depends on the type of the variable.

In Go programming variables can be declared using the var keyword. Go is a statically typed programming language. Which means that variables always have type associated with it and their type cannot be changed.

Example 1.1:

```
func main() {
    var v1 int
    var v2 int
    v1 = 100
    fmt.Println("Value stored in variable v1 :: ", v1)
    fmt.Println("Value stored in variable v2 :: ", v2)
}
```

Output:

```
Value stored in variable v1 ::  100
Value stored in variable v2 ::  0
```

Analysis:

Two variable v1 and v2 of integer type are created. Value 100 is stored in v1. Finally, value stored in v1 and v2 is printed to screen.

There is no uninitialized variable in Go. The compiler will give the default value to the variables according to their type. For example, Integer variables are automatically initialized to 0. String variables are initialized to empty “” string, pointers are initialized to nil etc.

In Go Programming, constants are declared using const keyword. A constant must be initialized at the time of declaration, and a constant cannot be changed once it is created.

```
const PI = 3.14
```

:= operator allows short declaration of variable. This does declaration and initialization of variable at once. The type of the variable is automatically determined by the compiler according to the type of value assigned

Example 1.2:

```
func main() {
    v1 := 100
    fmt.Println("Value stored in variable v1 :: ", v1)
}
```

Output:

```
Value stored in variable v1 ::  100
```

Analysis:

Go automatically determine that variable v1 is of type integers, as integer 100 is assigned to it.

Basic Data Types

Basic data types are the data types, which are defined in the Go language.

There are eight important family of primitive data types – Boolean, Byte, Integer, Unsigned Integer, Float, String, Rune, and Complex :

1. **Boolean :** These used to store True or False.
   Ex: `var B bool = true`

2. **Integer:** Int or Integer represent whole numbers. Integer can be negative or positive. When an integral literal is assigned to a variable. There are various type of integers depending upon the number of bytes they contain and how much big data they can contain. The other variation of integers are int8, int16, int32 & int64.
   Ex: `var I int = -1000`

3. **Unsigned Integer:** uint or Unsigned Integer are special type of integers, which can store only positive values. The other variation of integers are uint8, uint16, uint32 & uint64
   Ex: `var j uint = 1000`

4. **Float**: Decimal point number are stored inside Floating Point variable.
   Ex: `var f float32 = 1.2345`

5. **Strings** are sequences of Unicode characters. String are immutable.
   Ex: `var s string = “Hello, World!”`

6. **Byte** are alias of uint8

7. **Rune** are alias of int32 and are used to store Unicode characters.

8. **Complex** are used to store complex number.

Example 1.3:

```
import "math"

func main() {
    maxInt8 := math.MaxInt8
    minInt8 := math.MinInt8
    maxInt16 := math.MaxInt16
    minInt16 := math.MinInt16
    maxInt32 := math.MaxInt32
    minInt32 := math.MinInt32
    maxInt64 := math.MaxInt64
    minInt64 := math.MinInt64
    maxUint8 := math.MaxUint8
    maxUint16 := math.MaxUint16
    maxUint32 := math.MaxUint32
    maxUint64 := math.MaxUint64
    maxFloat32 := math.MaxFloat32
    maxFloat64 := math.MaxFloat64

    fmt.Println("Range of Int8 :: ", minInt8, " to ", maxInt8)
    fmt.Println("Range of Int16 :: ", minInt16, " to ", maxInt16)
```

```
    fmt.Println("Range of Int32 :: ", minInt32, " to ", maxInt32)
    fmt.Println("Range of Int64 :: ", minInt64, " to ", maxInt64)
    fmt.Println("Max Uint8 :: ", maxUint8)
    fmt.Println("Max Uint16 :: ", maxUint16)
    fmt.Println("Max Uint32 :: ", maxUint32)
    fmt.Println("Max Uint64 :: ", maxUint64)
    fmt.Println("Max Float32 :: ", maxFloat32)
    fmt.Println("Max Float64 :: ", maxFloat64)
}
```

Output:

```
Range of Int8 ::  -128  to  127
Range of Int16 ::  -32768  to  32767
Range of Int32 ::  -2147483648  to  2147483647
Range of Int64 ::  -9223372036854775808  to  9223372036854775807
Max Uint8 ::  255
Max Uint16 ::  65535
Max Uint32 ::  4294967295
Max Uint64 ::  18446744073709551615
Max Float32 ::  3.4028234663852886e+38
Max Float64 ::  1.7976931348623157e+308
```

In the above program, the various basic datatype max and min values are printed to screen.

String

A String is a sequence of Unicode character. String is an immutable type variable. Double quotes are used to declare strings.

```
s := "hello, World!"
```

Strings are immutable so you cannot change its content once created. You need to first convert into a slice of rune then do the changes and in the end convert it back to string.

Example 1.4:

```
func main() {
	s := "hello, World!"
	r := []rune(s)
	r[0] = 'H'
	s2 := string(r)
	fmt.Println(s2)
}
```

Output:

```
Hello, World!
```

Below is table, which explain some of the operations on string.

```
mystring := “Hello World!”
```

Expression	Value	Explanation
len(mystring)	12	Used to find the number of characters in mystring
“hello”+“world”	“helloworld”	Two strings are concatenated into a single string
“world” == “hello” “world” == “world”	False True	Equality can be tested using “==” sign
“a” < “b” “b” < “a”	True False	Unicode value can also be used to compare
mystring[0]	“h”	Indexing: String are indexed same as array.
mystring[1:4]	"ell"	Slicing

Conditions and Loops

IF Condition

If condition consists of a Boolean condition followed by one or more statements. It allows you to take different paths of logic, depending on a given Boolean condition.

```
if <Boolean expression> {
  <Statements>
}
```

If statement can be followed by an optional else statement which is executed when the Boolean condition is false.

```
if <Boolean expression> {
  <Statements>
} else {
  <Statements>
}
```

Example 1.5: Basic if

```
func more(x, y int) bool {
    if x > y {
        return true
    }
    return false
}
```

Analysis: In the above code two variables x & y are passed as argument to more() function. When x > y, true is returned else false is returned.

Example 1.6: Basic if with else.

```
func max(x, y int) int {
    var max int
    if x > y {
        max = x
    } else {
        max = y
    }
    return max
}
```

Analysis: In the above code two variables x & y are passed as argument to max() function. When x > y, value stored in x is assigned to max else value stored in y is assigned to max and finally value stored in max is returned.

Go language if condition allow precondition along with Boolean condition.

Example 1.7: If with precondition.

```
func maxAreaCheck(length, width, limit int) bool {
    if area := length * width; area < limit {
        return true
    } else {
        return false
```

```
    }
}
```

Switch

switch statement is easy to read and understand when used to handle multiple conditional branches.

The syntax of switch basic form

```
switch <Initialization>; <condition> {
    case <value1>:
        <statements>
    case<value2>:
        <statements>
    /* We can have any number of case statements */
    default:  /* Optional */
        <statements>
}
```

The syntax of switch with conditions

```
switch {
    case <condition>:
        <statements>
    case <condition>:
        <statements>
    default:
        <statements>
}
```

The syntax of switch for type.

```
switch var.(type){
    case <type>:
        <statements>
    case <type>:
        <statements>
    default:
        <statements>
}
```

When a case is match, all the statements under that case are executed and switch statement is over.

Example 1.8:

```
func main() {
    i := 2
    switch i {
    case 1:
        fmt.Println("one")
    case 2:
        fmt.Println("two")
    case 3:
        fmt.Println("three")
    default:
        fmt.Println("something else")
```

```
    }
}
```

Analysis: Above program, demonstrate basic switch use. The value of variable “i” is checked in each case. “i” has value 2 so second case will be executed and “two” will be printed to screen.

Example 1.9:
```
func main() {
    i := 2
    switch i {
    case 1, 2, 3:
        fmt.Println("one, two or three")
    default:
        fmt.Println("something else")
    }
}
```

Analysis: Above program, demonstrate that various cases can be merged. The value of variable “i” is checked in each case. “i” has value 2 so “one, two or three” line will be printed to screen.

Example 1.10:
```
func isEven(value int) {
    switch {
    case value%2 == 0:
        fmt.Println("I is even")
    default:
        fmt.Println("I is odd")
    }
}
```

Analysis: The above program demonstrate switch with conditions, which is an alternative of multiple if-else.

For Loop

For loop help to iterate through a group of statements multiple times.

The Go for loop has four forms:
1. for <initialization>; <condition>; <increment/decrement> { }
2. for <condition> { } - like a while loop
3. for { } - an infinite while loop.
4. for with range.

It is the most standard and normal usage. Used in initial value / conditional expression / incremental expression form.

Example 1.11:
```
func main() {
    numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    sum := 0
    for i := 0; i < len(numbers); i++ {
        sum += numbers[i]
    }
    fmt.Println("Sum is :: ", sum)
```

```
}
```

If you just have conditional expression, it works as a while loop which runs until condition statement is false.

Example 1.12:
```
func main() {
	numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
	sum := 0
	i := 0
	n := len(numbers)
	for i < n {
		sum += numbers[i]
		i++
	}
	fmt.Println("Sum is :: ", sum)
}
```

If you omit the initial value / conditional expression / increment or decrement expression, it works as an infinite loop until the `break` statement is encountered.

Example 1.13:
```
func main() {
	numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
	sum := 0
	i := 0
	n := len(numbers)
	for {
		sum += numbers[i]
		i++
		if i >= n {
			break
		}
	}
	fmt.Println("Sum is :: ", sum)
}
```

Using `for` statement with a `range` statement is used to traverse various elements of array, slice etc.

Example 1.14:
```
func main() {
	numbers := […]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
	sum := 0
	for _, val := range numbers {
		sum += val
	}
	fmt.Println("Sum is :: ", sum)
}
```

Analysis: Various elements of array numbers is traversed using range in for loop.

Range

The range keyword is used in for loop to iterate data in data structures (arrays, slices, string, maps etc.). Behaviour of range keyword for different data structures:

1. With arrays and slices, range provides indexes and values.
2. With maps, range provide key-value pairs.
3. With string, range provide index of each Unicode characters in string and their corresponding Unicode characters.
4. If index values are not needed then they can be discarded using _.

Example 1.15:

```
func main() {
    numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    sum := 0
    for index, val := range numbers {
        sum += val
        fmt.Print("[", index, ",", val, "] ")
    }
    fmt.Println("\nSum is :: ", sum)
    kvs := map[int]string{1: "apple", 2: "banana"}
    for k, v := range kvs {
        fmt.Println(k, " -> ", v)
    }
    str := "Hello, World!"
    for index, c := range str {
        fmt.Print("[", index, ",", string(c), "] ")
    }
}
```

Output:

```
[0,1] [1,2] [2,3] [3,4] [4,5] [5,6] [6,7] [7,8] [8,9] [9,10]
Sum is ::  55
1  ->  apple
2  ->  banana
[0,H] [1,e] [2,l] [3,l] [4,o] [5,,] [6, ] [7,W] [8,o] [9,r] [10,l] [11,d] [12,!]
```

Analysis:

- A slice is created which contain numbers from 1 to 10.
- Slice is traversed and its content is printed to screen. Sum of all the elements of slice is calculated and printed to screen.
- If we are not interested in index then we can use single underscore (_) to tell the compiler that we do not need this variable.
- We had created a map and its content is traversed and printed to screen.
- We had created a string str and its characters are traversed and printed to screen.

Function

Functions are used to provide modularity to the program. By using function, we can divide complex tasks into smaller manageable tasks. The use of the function is also to avoid duplicate code. For example, we can define a function max() to find bigger value. Then we can use this function multiple times whenever we want compare two integer values.

Program 1.16: Demonstrating Function Calls

```
func max(x, y int) int {
    if x > y {
        return x
    }
    return y
}

func main() {
    fmt.Println(max(10, 20));
}
```

Output:

```
20
```

Analysis:

- Function max() is declared, which will compare two integer variable x and y. This function will return greater value among two.
- Inside main() function value 10 and 20 are passed to max() function.
- Variables passed to max function are copied into x and y local variables.
- The values are compared and grater among two is returned. Finally, 20 is returned by the function and printed to screen.

Parameter passing, Call by value

Arguments can be passed from one function to other using parameters. By default, all the parameters are passed by value. That means a separate copy is created inside the called function and the variable in the calling function remains unchanged.

Example 1.17:

```
func IncrementPassByValue(x int) {
    x++
}
func main1() {
    i := 10
    fmt.Println("Value of i before increment is :  ", i)
    IncrementPassByValue(i)
    fmt.Println("Value of i after increment is :  ", i)
}
```

Output:

```
Value of i before increment is : 10
Value of i before increment is : 10
```

Analysis:

- Variable “i” is declared and value 10 is initialized to it.
- Value of ”i” is printed.
- Increment function is called. When a function is called the value of the parameter is copied into another variable of the called function. Flow of control goes to line no 1.
- Value of var is incremented by 1. However, remember, it is just a copy inside the increment function.
- When the function exits, the value of ”i” is still 10.

Points to remember:

1. Pass by value just creates a copy of variable.
2. Pass by value, value before and after the function call remain same.

Pointers

Pointers are nothing more than variables that store memory addresses of another variable and can be used to access the value stored at those addresses. Various operators such as *, &, and [], enable us to use pointers.

Example 1.18:

```
func main() {
    data := 10
    ptr := &data
    fmt.Println("Value stored at variable var is ", data)
    fmt.Println("Value stored at variable var is ", *ptr)
    fmt.Println("The address of variable var is ", &data)
    fmt.Println("The address of variable var is ", ptr)
}
```

Output:

```
Value stored at variable var is  10
Value stored at variable var is  10
The address of variable var is  0xc04200e210
The address of variable var is  0xc04200e210
```

Analysis:

- An integer type variable var is created, which store value 10
- A pointer ptr is created, which store address of variable var.
- Value stored in variable var is printed to screen. Using * Operator value stored at the pointer location is accessed.
- Memory address of var is printed to the screen. & operator is used to get the address of a variable.

Parameter passing, Call by Pointer / Reference

If you need to change the value of the parameter inside the function, then you should use call by reference. C language by default passes by value. Therefore, to make it happen, you need to pass the address of a variable and changing the value of the variable using this address inside the called function.

Program 1.19:

```
func IncrementPassByPointer(ptr *int) {
	(*ptr)++
}
func main1() {
	i := 10
	fmt.Println("Value of i before increment is :  ", i)
	IncrementPassByPointer(&i)
	fmt.Println("Value of i after increment is :  ", i)
}
```

Output:

```
Value of i before increment is : 10
Value of i before increment is : 11
```

Analysis:

- Address of “i” is passed to the function increment. Function increment takes a pointer to int as argument.
- Variable at the address ptr is accessed and its value is incremented.
- Finally, incremented value is printed to screen.

Points to remember:

1. Call by reference is implemented indirectly by passing the address of a variable to the function.

Structures

Go language supports structures, which are a collection of multiple data types as a single entity.

Example 1.20:

```
type student struct {
	rollNo int
	name   string
}
func main() {
	stud := student{1, "Johnny"}
	fmt.Println(stud)
	fmt.Println("Student name ::",stud.name) // Accessing inner fields.
	pstud := &stud
	fmt.Println("Student name ::",pstud.name) // Accessing inner fields.
	fmt.Println(student{rollNo: 2, name: "Ann"}) // Named initialization.
	fmt.Println(student{name: "Ann", rollNo: 2}) // Order does not matter.
	fmt.Println(student{name: "Alice"}) // Default initialization of rollNo.
}
```

Output:

```
{1 Johnny}
Student name :: Johnny
Student name :: Johnny
{2 Ann}
{2 Ann}
{0 Alice}
```

Analysis:

- We have created a student stud. Whose name is “Johnny” and roll number is “1”
- Printed structure stud to the output screen.
- We can access the fields of structure using dot (.) Operator
- Structure pointers can also be used to access inner fields using same dot (.) Operator. Go language automatically access the inner fields by dereferencing.
- When we create structure, we can initialize fields using field name.
- When we initialize fields using field name. The order does not matter.
- When we omit some fields name in the structure while creating it. The omitted fields are initialized to their default zero value. Example. 0 for integers, nil for pointers or empty string “” for strings.

Methods

Go is an object-oriented language. However, it does not have any class keyword. We can associate functions directly by structures and other data types. Between func keyword and name of function, we add data type called “receiver”. Once function is associated by a receiver, we can directly call function over structure using dot (.) operator.

Example 1.21:

```
type Rect struct {
	width  float64
	height float64
}
func (r Rect) Area() float64 {
	return r.width * r.height
}

func (r Rect) Perimeter() float64 {
	return 2 * (r.width + r.height)
}

func main() {
	r := Rect{width: 10, height: 10}
	fmt.Println("Area: ", r.Area())
	fmt.Println("Perimeter: ", r.Perimeter())

	ptr := &Rect{width: 10, height: 5}
	fmt.Println("Area: ", ptr.Area())
	fmt.Println("Perimeter: ", ptr.Perimeter())
}
```

Output:

```
Area:  100
Perimeter:  40
Area:  50
Perimeter:  30
```

Analysis:

- In the above program, we have Rect structure, which represent rectangle. Rect has width and height fields.
- We associate two function Area() and Parameter() and associate them with Rect data type.
- In the main() function we had created an instance of Rect, with width10 and height 10.
- Then Area() and Perimeter() functions are called using dot(.) operator over Rect instance r.
- Then another instance of Rect is created and its address is stored in pointer ptr.
- Again Area() and Perimeter() function is called using dot(.) operator but this time we are calling the associated function over pointer. The go language automatically convert the pointer to value and called the associated functions.

There are two ways of defining associated function of a data type.

1. Accessor function, which take receiver as value. Go passes a copy of the instance this function (Just like pass by value.). Any change done over the object is not reflected in the calling object.

The syntax of accessor function:

```
func (r <Receiver Data type>) <Function Name>(<Parameter List>) (<Return List>)
```

2. Modifier function, which take receiver as pointer. Go passes actual instance of the object to this function (Just like pass by pointer.) Any change done over the object is reflected on the original object.

The syntax of modifier function:

```
func (r *<Receiver Data type>) <Function Name>(<Parameter List>) (<Return List>)
```

Example 1.22:

```
type MyInt int

func (data MyInt) increment1() {
	data = data + 1
}
func (data *MyInt) increment2() {
	*data = *data + 1
}
func main() {
	var data MyInt = 1
	fmt.Println("value before increment1() call :", data)
	data.increment1()
	fmt.Println("value after increment1() call :", data)
	data.increment2()
	fmt.Println("value after increment2() call :", data)
}
```

Output:

```
value before increment1() call : 1
value after increment1() call : 1
value after increment2() call : 2
```

Analysis:

- Accessor function increment1() does changes on a local copy of the object. Therefore, changes done are lost.
- Modifier function increment2() does changes on the actual instance so changes done are preserved.

Interface

Interfaces are defined as a set of methods.
Syntax of interface:

```
Type <Interface name> interface {
    <Method name>  <Return type>
}
```

In Go, to implement an interface an object just need to implement all methods of interface. When an object implement a particular interface, its object can be assigned to an interface type variable.

Example 1.23:

```
type Shape interface {
    Area() float64
    Perimeter() float64
}

type Rect struct {
    width  float64
    height float64
}

type Circle struct {
    radius float64
}

func (r Rect) Area() float64 {
    return r.width * r.height
}

func (r Rect) Perimeter() float64 {
    return 2 * (r.width + r.height)
}

func (c Circle) Area() float64 {
    return math.Pi * c.radius * c.radius
}

func (c Circle) Perimeter() float64 {
    return 2 * math.Pi * c.radius
}

func TotalArea(shapes ...Shape) float64 {
    var area float64
    for _, s := range shapes {
        area += s.Area()
    }
    return area
}

func TotalPerimeter(shapes ...Shape) float64 {
    var peri float64
```

```
    for _, s := range shapes {
        peri += s.Perimeter()
    }
    return peri
}

func main() {
    r := Rect{width: 10, height: 10}
    c := Circle{radius: 10}
    fmt.Println("Total Area: ", TotalArea(r, c))
    fmt.Println("Total Perimeter: ", TotalPerimeter(r, c))
}
```

Output:

```
Total Area:  414.1592653589793
Total Perimeter:  102.83185307179586
```

Analysis:

- A Shape interface is created which contain two methods Area() and Perimeter().
- Rect and Circle implements Shape interface as they implement Area() and Perimeter() methods.
- TotalArea() and TotalPerimeter() are two functions which expect list of object of type Shape.

Array

An array is a collection of variables of the same data type

Example 1.24:

```
func main() {
	var arr [10]int
	fmt.Println(arr)
	for i := 0; i < 10; i++ {
		arr[i] = i
	}
	fmt.Println(arr)
	count := len(arr)
	fmt.Println("Length of array", count)
}
```

Output:

```
[0 0 0 0 0 0 0 0 0 0]
[0 1 2 3 4 5 6 7 8 9]
Length of array 10
```

Analysis:

- We had declared array arr of size 10. In Go language, the array size is a part of array. The array name is whole array and not pointer to first element like that in C/ C++.
- By default, all the elements of array are initialized to their default value. For our example, the default value of int type is 0.
- We can read and set each element of array.
- We can get size of array by using built-in function len().

Slice

Go Array is a collection of variables of same data type. Arrays are fixed in length and does not have any inbuilt method to increase their size.

Go Slice is an abstraction over Array. It actually uses arrays as an underlying structure. The various operations over slice are:

1. Inbuilt append() function is used to add the elements to a slice. If the size of underlying array is not enough then automatically a new array is created and content of the old array is copied to it.
2. The len() function returns the number of elements presents in the slice.
3. The cap() function returns the capacity of the underlying array of the slice.
4. The copy() function, the contents of a source slice are copied to a destination slice.
5. Re-slices the slice, the syntax <Slice Name>[start : end] , It returns a slice object containing the elements of base slice from index start to end-1. Length of the new slice will be (end - start), and capacity will be cap (base slice) - start.

Slice provides these utility functions because of which it is widely used in Go programming.
To define a slice, you can declare it as an array without specifying its size. Alternatively, you can use make function to create a slice.

Example 1.25:

```
func main() {
    var s []int
    for i := 1; i <= 17; i++ {
        s = append(s, i)
        PrintSlice(s)
    }
}
func PrintSlice(data []int) {
    fmt.Printf("%v :: len=%d cap=%d \n", data, len(data), cap(data))
}
```

Output:

```
[1] :: len=1 cap=1
[1 2] :: len=2 cap=2
[1 2 3] :: len=3 cap=4
[1 2 3 4] :: len=4 cap=4
[1 2 3 4 5] :: len=5 cap=8
[1 2 3 4 5 6] :: len=6 cap=8
[1 2 3 4 5 6 7] :: len=7 cap=8
[1 2 3 4 5 6 7 8] :: len=8 cap=8
[1 2 3 4 5 6 7 8 9] :: len=9 cap=16
[1 2 3 4 5 6 7 8 9 10] :: len=10 cap=16
[1 2 3 4 5 6 7 8 9 10 11] :: len=11 cap=16
[1 2 3 4 5 6 7 8 9 10 11 12] :: len=12 cap=16
[1 2 3 4 5 6 7 8 9 10 11 12 13] :: len=13 cap=16
[1 2 3 4 5 6 7 8 9 10 11 12 13 14] :: len=14 cap=16
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15] :: len=15 cap=16
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16] :: len=16 cap=16
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17] :: len=17 cap=32
```

Analysis:

- First, we had created an empty slice s.
- Using append() function we are adding elements to the slice.
- The capacity of the underlying array is increasing from 1 to 2. Then from 2 to 4. Then 8 to 16 and finally 32.
- As we keep adding elements to the slice, its capacity is managed automatically.

Example 1.26:

```
func main() {
    s := []int{1,2,3,4,5,6,7,8,9,10}
    PrintSlice(s)        // [1 2 3 4 5 6 7 8 9 10] :: len=10 cap=10

    a := make([]int, 10)
    PrintSlice(a)        // [0 0 0 0 0 0 0 0 0 0] :: len=10 cap=10

    b := make([]int, 0, 10)
    PrintSlice(b)        //  [] :: len=0 cap=10

    c := s[0:4]
    PrintSlice(c)        // [1 2 3 4] :: len=4 cap=10

    d := c[2:5]
    PrintSlice(d)        // [3 4 5] :: len=3 cap=8
}
```

Analysis:

- First, we had created a slice s using array like initialization.
- Then we had created slice “a” with 10 elements (Capacity is also 10) which are all initialized to 0
- Then we had created slice “b” with 0 elements with capacity of 10 elements.
- Then we are using re-slicing to create another slice “c”, which contain elements of slice “s” from index 0 to 3. Capacity of the slice “c” remain 10.
- Then we are using re-slicing to create another slice “d”, which contain elements of slice “c” from index 2 to 5. Capacity of the slice “d” reduced to 8.

Map / Dictionary

A map is a collection of Key-Value pairs. Hash-Table is used to store elements in a Map so it is unordered.

```
var <variable> map[<key datatype>]<value datatype>
```

Maps have to be initialized using make() before they can be used.

```
var <variable> map[<key datatype>]<value datatype> = make(map[<key datatype>]<value datatype>)
```

or

```
<variable> := make(map[<key datatype>]<value datatype>)
```

Various operation on map:

1. Assignment: < variable>[<key>] = <value>
2. Delete: delete(<variable >, <key>)
3. Access: value, ok = < variable >[<key>] , the first value will have the value of key in the map. If the key is not present in the map, it will return zero value corresponding to the value data-type. The second argument returns whether the map contains the key.

Example 1.27:

```
func main() {
	m := make(map[string]int)
	m["Apple"] = 40
	m["Banana"] = 30
	m["Mango"] = 50
	for key, val := range m {
		fmt.Print("[ ",key," -> ", val," ]")
	}
	fmt.Println("Apple price:", m["Apple"])
	delete(m, "Apple")

	value, ok := m["Apple"]
	fmt.Println("Apple price:", value, "Present:", ok)

	value2, ok2 := m["Banana"]
	fmt.Println("Banana price:", value2, "Present:", ok2)
}
```

Output:

```
[ Apple -> 40 ][ Banana -> 30 ][ Mango -> 50 ]
Apple price: 40
Apple price: 0
Apple price: 0 Present: false
Banana price: 30 Present: true
```

Note: Dictionaries are implemented using Hash Table, so order of keys is not guaranteed.

Array Interview Questions

The following section will discuss the various algorithms that are applicable to Lists and will follow by list of practice problems with similar approaches.

Sum List

Write a method that will return the sum of all the elements of the integer list, given list as an input argument.
Example 1.28:
```
func SumArray(data []int) int {
	size := len(data)
	total := 0
	for index := 0; index < size; index++ {
		total = total + data[index]
	}
	return total
}
```

Sequential Search

Example 1.29: Write a method, which will search a list for some given value.
```
func SequentialSearch(data []int, value int) bool {
	size := len(data)
	for i := 0; i < size; i++ {
		if value == data[i] {
			return true
		}
	}
	return false
}
```

Analysis:
- Since we have no idea about the data stored in the list, or if the data is not sorted then we have to search the list in sequential manner one by one.
- If we find the value, we are looking for we return true.
- Else, we return False in the end, as we did not found the value we are looking for.

In the above example, the data are not sorted. If the data is sorted, a binary search can be used. We examine the middle position at each step. Depending upon the data that we are searching is greater or smaller than the middle value. We will search either the left or the right portion of the array. At each step, we are eliminating half of the search space there by making this algorithm very efficient.

Binary Search

Example 1.30: Binary search in a sorted list.
```
func BinarySearch(data []int, value int) bool {
	size := len(data)
	var mid int
	low := 0
	high := size - 1
	for low <= high {
```

```
        mid = low + (high-low)/2
        if data[mid] == value {
            return true
        } else {
            if data[mid] < value {
                low = mid + 1
            } else {
                high = mid - 1
            }
        }
    }
    return false
}
```

Analysis:

- Since we have data sorted in increasing / decreasing order, we can apply more efficient binary search. At each step, we reduce our search space by half.
- At each step, we compare the middle value with the value we are searching. If mid value is equal to the value we are searching for then we return the middle index.
- If the value is smaller than the middle value, we search the left half of the list.
- If the value is greater than the middle value then we search the right half of the list.
- If we find the value we are looking for then its index is returned or -1 is returned otherwise.

Rotating a list by K positions.

Given a list, you need to rotate its elements K number of times. For example, a list [10,20,30,40,50,60] rotate by 2 positions to [30,40,50,60,10,20]

Example 1.31:

```
func RotateArray(data []int, k int) {
    n := len(data)
    ReverseArray(data, 0, k-1)
    ReverseArray(data, k, n-1)
    ReverseArray(data, 0, n-1)
}

func ReverseArray(data []int, start int, end int) {
    i := start
    j := end
    for i < j {
        data[i], data[j] = data[j], data[i]
        i++
        j--
    }
}
```

Analysis:

- Rotating list is done in two parts trick. In the first part, we first reverse elements of list first half and then second half.

1,2,3,4,5,6,7,8,9,10 => 5,6,7,8,9,10,1,2,3,4
1,2,3,4,5,6,7,8,9,10 => 4,3,2,1,10,9,8,7,6,5 => 5,6,7,8,9,10,1,2,3,4

· Then we reverse the whole list there by completing the whole rotation.

Find the largest sum contiguous subarray.

Given a list of positive and negative integers, find a contiguous subarray whose sum (sum of elements) is maximum.

Example 1.32:

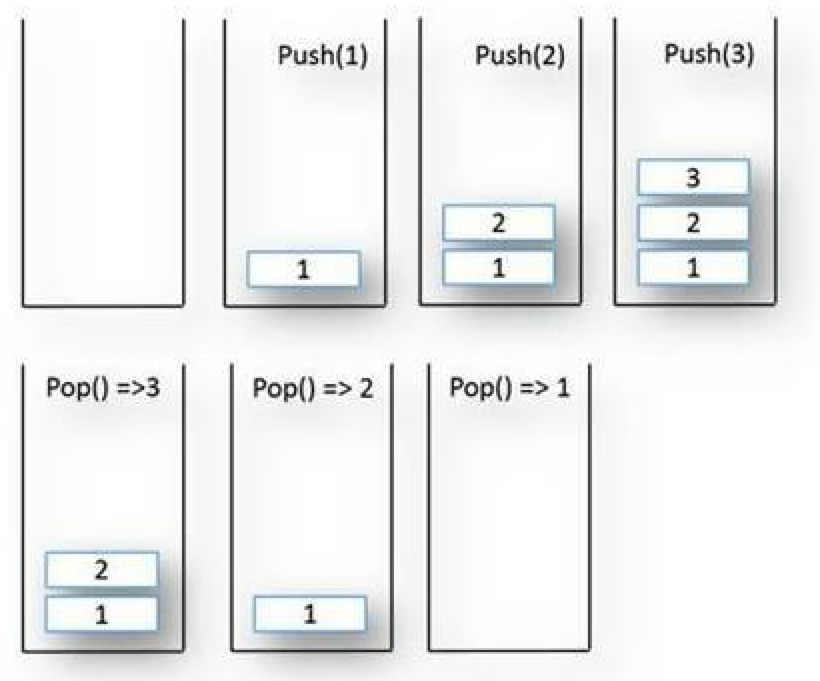
```
func MaxSubArraySum(data []int) int {
	size := len(data)
	maxSoFar := 0
	maxEndingHere := 0
	for i := 0; i < size; i++ {
		maxEndingHere = maxEndingHere + data[i]
		if maxEndingHere < 0 {
			maxEndingHere = 0
		}
		if maxSoFar < maxEndingHere {
			maxSoFar = maxEndingHere
		}
	}
	return maxSoFar
}

func main() {
	data := []int{1, -2, 3, 4, -4, 6, -14, 8, 2}
	fmt.Println("Max sub array sum :", MaxSubArraySum(data))
}
```

Analysis:

- Maximum subarray in a list is found in a single scan. We keep track of global maximum sum so far and the maximum sum, which include the current element.
- When we find global maximum value so far is less than the maximum value containing current value we update the global maximum value.
- Finally return the global maximum value.

Concept of Stack

A stack is a memory in which values are stored and retrieved in “last in first out” manner. Data is added to stack using push operation and data is taken out of stack using pop operation.



1. Initially the stack was empty. Then value 1 is added to stack using push(1) operator.
2. Similarly, push(2) and push(3)
3. Pop operation take the top of the stack. In Stack, data is added and deleted in “last in, first out” manner.
4. First pop() operation will take 3 out of the stack.
5. Similarly, other pop operation will take 2 then 1 out of the stack
6. In the end, the stack is empty when all the elements are taken out of the stack.

System stack and Method Calls

When a method is called, the current execution is stopped and the control goes to the called method. After the called method exits / returns, the execution resumes from the point at which the execution was stopped.

To get the exact point at which execution should be resumed, the address of the next instruction is stored in the stack. When the method call completes, the address at the top of the stack is taken out.

Example 1.33:

```
func function2() {
    fmt.Println("fun2 line 1")
}

func function1() {
    fmt.Println("fun1 line 1")
    function2()
    fmt.Println("fun1 line 2")
}

func main() {
    fmt.Println("main line 1")
    function1()
    fmt.Println("main line 2")
}
```

Output:

```
main line 1
fun1 line 1
fun2 line 1
fun1 line 2
main line 2
```

Analysis:

· Every program starts with main() method.
· The first statement of main() will be executed. This will print “main line 1” as output.
· function1() is called. Before control goes to function1() then next instruction that is address of next line is stored in the system stack.
· Control goes to function1() method.
· The first statement inside function1() is executed, this will print “fun1 line 1” to output.
· function2() is called from function1(). Before control goes to function2() address of the next instruction that is address of next line is added to the system stack.
· Control goes to function2() method.
· “fun2 line 1” is printed to screen.
· When function2() exits, control come back to function1(). Then the program reads the next instruction from the stack, and the next line is executed and print “fun1 line 2” to screen.
· When fun1 exits, control comes back to the main method. Then program reads the next instruction from the stack and execute it and finally “main line 2” is printed to screen.

Points to remember:

1. Methods are implemented using a stack.
2. When a method is called the address of the next instruction is pushed into the stack.
3. When a method is finished the address of the execution is taken out of the stack.

Recursive Function

A recursive function is a function that calls itself, directly or indirectly.
A recursive method consists of two parts: Termination Condition and Body (which include recursive expansion).

1. **Termination Condition:** A recursive method always contains one or more terminating condition. A condition in which recursive method process a simple case and do not call itself.

2. **Body** (including recursive expansion): The main logic of the recursive method contained in the body of the method. It also contains the recursion expansion statement that in turn calls the method itself.

Three important properties of recursive algorithm are:
1) A recursive algorithm must have a termination condition.
2) A recursive algorithm must change its state, and move towards the termination condition.
3) A recursive algorithm must call itself.

Note: The speed of a recursive program is slower because of stack overheads. If the same task can be done using an iterative solution (using loops), then we should prefer an iterative solution in place of recursion to avoid stack overhead.

Note: Without termination condition, the recursive method may run forever and will finally consume all the stack memory.

Factorial

Example 1.34: Factorial Calculation. N! = N* (N-1)…. 2*1.

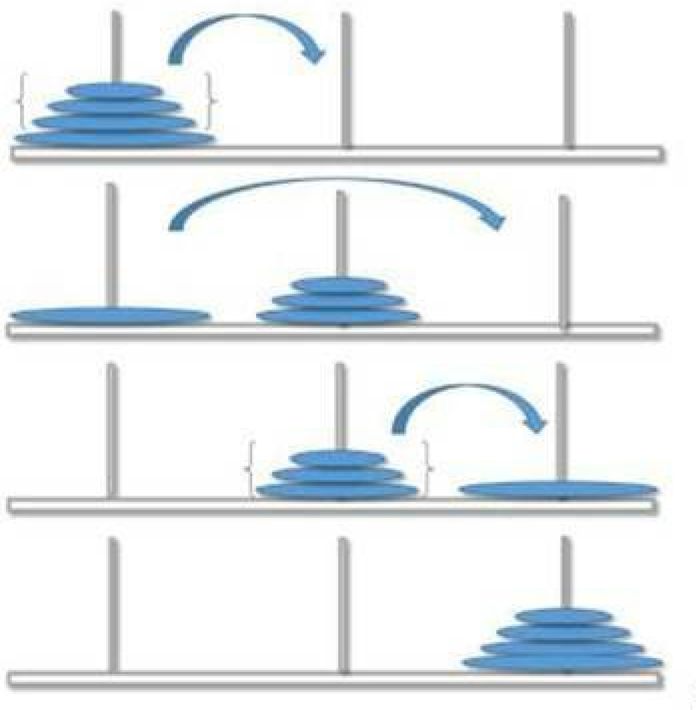
```
func Factorial(i int) int {
	// Termination Condition
	if i <= 1 {
		return 1
	}
	// Body, Recursive Expansion
	return i * Factorial(i-1)
}

func main(){
	fmt.Println("factorial 5 is :: ", Factorial(5))
}
```

Analysis: **Factorial**() function is calling itself back inside the function. Continue calling itself recursively until you call the **Factorial**(1) function.

Time Complexity is **O(N)**

Tower of Hanoi

The **Tower of Hanoi** (also called the **Tower of Brahma**) We are given three rods and N number of disks, initially all the disks are added to first rod (the leftmost one) in decreasing size order. The objective is to transfer the entire stack of disks from first tower to third tower (the rightmost one), moving only one disk at a time and never a larger one onto a smaller.



Example 1.35:
```
func main() {
    TowersOfHanoi(3)
}

func TOHUtil(num int, from string, to string, temp string) {
    if num < 1 {
        return
    }
    TOHUtil(num-1, from, temp, to)
    fmt.Println("Move disk ", num, " from peg ", from, " to peg ", to)
    TOHUtil(num-1, temp, to, from)
}

func TowersOfHanoi(num int) {
    fmt.Println("The sequence of moves involved in the Tower of Hanoi are :")
    TOHUtil(num, "A", "C", "B")
}
```

Analysis: To move N disks from source to destination, we first move N-1 disks from source to temp then move the lowest Nth disk from source to destination. Then will move N-1 disks from temp to destination.

Greatest common divisor (GCD)

Example 1.36: Find greatest common divisor.
```
func GCD(m int, n int) int {
    if m < n {
        return GCD(n, m)
    }

    if m%n == 0 {
```

```
        return n
    }

    return GCD(n, m%n)
}
```

Analysis: Euclid’s algorithm is used to find gcd. GCD(n, m) == GCD(m, n mod m).

Fibonacci number

Example 1.37: Given N find the Nth number in the Fibonacci series. .

```
func fibonacci(n int) int {
    if n <= 1 {
        return n
    }
    return fibonacci(n-1) + fibonacci(n-2)
}
```

Analysis: Fibonacci number are calculated by adding sum of the previous two number.

Note:- There is an inefficiency in the solution we will look better solution in coming chapters.

All permutations of an integer list

Example 1.38: Generate all permutations of an integer list.

```
func Permutation(data []int, i int, length int) {
    if length == i {
        PrintSlice(data)
        return
    }
    for j := i; j < length; j++ {
        swap(data, i, j)
        Permutation(data, i+1, length)
        swap(data, i, j)
    }
}

func swap(data []int, x int, y int) {
    data[x], data[y] = data[y], data[x]
}

func main() {
    var data [5]int
    for i := 0; i < 5; i++ {
        data[i] = i
    }
    Permutation(data[:], 0, 5)
}
```

Analysis: In permutation method at each recursive call number at index, “i” is swapped with all the numbers that are right of it. Since the number is swapped with all the numbers in its right one by one it will produce all the permutation possible.

Binary search using recursion

Example 1.39: Search a value in an increasing order sorted list of integers.

```
func BinarySearchRecursive(data []int, low int, high int, value int) int {
	mid := low + (high-low)/2 // To afunc the overflow
	if data[mid] == value {
		return mid
	} else if data[mid] < value {
		return BinarySearchRecursive(data, mid+1, high, value)
	} else {
		return BinarySearchRecursive(data, low, mid-1, value)
	}
}
```

Analysis: Similar iterative solution we had already seen. Now let us look into the recursive solution of the same problem. In this solution, we are diving the search space into half and discarding the rest. This solution is very efficient as each step we are rejecting half the search space/ list.

Exercises

1) Find average of all the elements in a list.

2) Find the sum of all the elements of a two dimensional list.

3) Find the largest element in the list.

4) Find the smallest element in the list.

5) Find the second largest number in the list.

6) Using AllPermutation function discussed before, write a function, which give only distinct solutions.

7) Write a method to compute Sum(N) = 1+2+3+…+N.

8) Print all the maxima's in a list. (A value is a maximum if the value before and after its index are smaller than it is or does not exist.)

   Hint: Start traversing list from the end and keep track of the max element. If we encounter an element whose value is grater then max, print the element and update max.

9) Given a list of intervals, merge all overlapping intervals.

   Input: {[1, 4], [3, 6], [8, 10]}, Output: {[1, 6], [8, 10]}

10) Reverse a list in-place. (You cannot use any additional list in other wards Space Complexity should be **O(1).**)

   Hint: Use two variable, start and end. Start set to 0 and end set to (n-1). Increment start and decrement end. Swap the values stored at arr[start] and arr[end]. Stop when start is equal to end or start is greater than end.

11) Given a list of 0s and 1s. We need to sort it so that all the 0s are before all the 1s.

   Hint: Use two variable, start and end. Start set to 0 and end set to (n-1). Increment start and decrement end. Swap the values stored at arr[start] and arr[end] only when arr[start] == 1 and arr[end] ==0. Stop when start is equal to end or start is greater than end.

12) Given an array of 0s, 1s and 2s. We need to sort it so that all the 0s are before all the 1s and all the 1s are before 2s.

   Hint: Same as above first think 0s and 1s as one group and move all the 2s on the right side. Then do a second pass over the array to sort 0s and 1s.

13) Find the duplicate elements in a list of size n where each element is in the range 0 to n-1.

Hint:

Approach 1: Compare each element with all the elements of the list (using two loops) $O(n^2)$ solution

Approach 2: Maintain a Hash-Table. Set the hash value to 1 if we encounter the element for the first time. When we same value again we can see that the hash value is already 1 so we can print that value. **O(n)** solution, but additional space is required.

Approach 3: We will exploit the constraint "every element is in the range 0 to n-1". We can take a list arr[] of size n and set all the elements to 0. Whenever we get a value say val1. We will increment the value at arr[var1] index by 1. In the end, we can traverse the list arr and print the repeated values. Additional Space Complexity will be **O(n)** which will be less than Hash-Table approach.

14) Find the maximum element in a sorted and rotated list. Complexity: **O(logn)**

Hint: Use binary search algorithm.

15) Given a slice with 'n' elements & a value 'x', find two elements in the list that sums to 'x'.

Hint:

Approach 1: Sort the list.

Approach 2: Using a Hash-Table.

16) Write a method to find the sum of every number in an int number. Example: input= 1984, output should be 32 (1+9+8+4).

CHAPTER 2: ALGORITHMS ANALYSIS

Introduction

We learn by experience. By looking into various problem solving algorithms or problem solving techniques we begin to develop a pattern that will help us in solving similar problems when they come in front of us.

Algorithm

An algorithm is a set of steps to accomplish a task. Or an algorithm in a computer program in which a set of steps applied over a set of input to produce a set of output.

Knowledge of algorithm helps us to get desired result faster by applying the right algorithm.

The most important properties of an algorithm are:

1. **Correctness**: The algorithm should be correct. It should be able to process all the given inputs and provide correct output.

2. **Efficiency**: The algorithm should be efficient in solving problems. Efficiency is measured in two parameters. Firs is Time-Complexity, how fast result is provided by an algorithm. And the second is Space-Complexity, how much RAM that an algorithm is going to consume to give desired result.

Time-Complexity is represented by function T(n) - time versus the input size n.
Space-Complexity is represented by function S(n) - memory used versus the input size n.

Asymptotic analysis

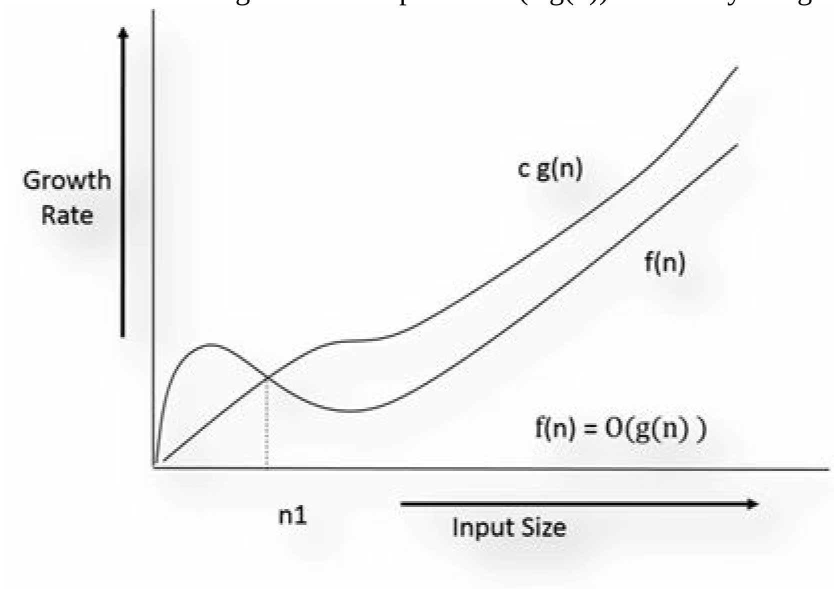
Asymptotic analysis is used to compare the efficiency of algorithm independently of any particular data set or programming language.

We are generally interested in the order of growth of some algorithm and not interested in the exact time required for running an algorithm. This time is also called Asymptotic-running time.

Big-O Notation

Definition: “f(n) is big-O of g(n)” or f(n) = O(g(n)), if there are two +ve constants c and n0 such that
$f(n) \le c\, g(n)$ for all $n \ge n0$,
In other words, c g(n) is an upper bound for f(n) for all $n \ge n0$
The function f(n) growth is slower than c g(n)

We can simply say that after a sufficient large value of input N the (c.g(n)) will always be greater than f(n).

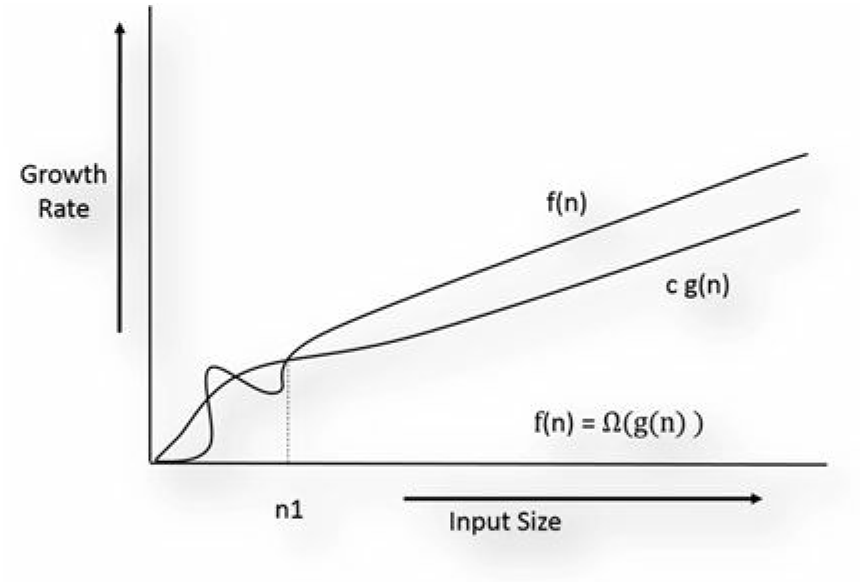


Example: $n^2 + n = O(n^2)$

Omega-Ω Notation

Definition: “f(n) is omega of g(n).” or f(n) = Ω(g(n)) if there are two +ve constants c and n0 such that
c g(n) ≤ f(n) for all n ≥ n0

In other words, c g(n) is lower bound for f(n)
Function f(n) growth is faster than c g(n)

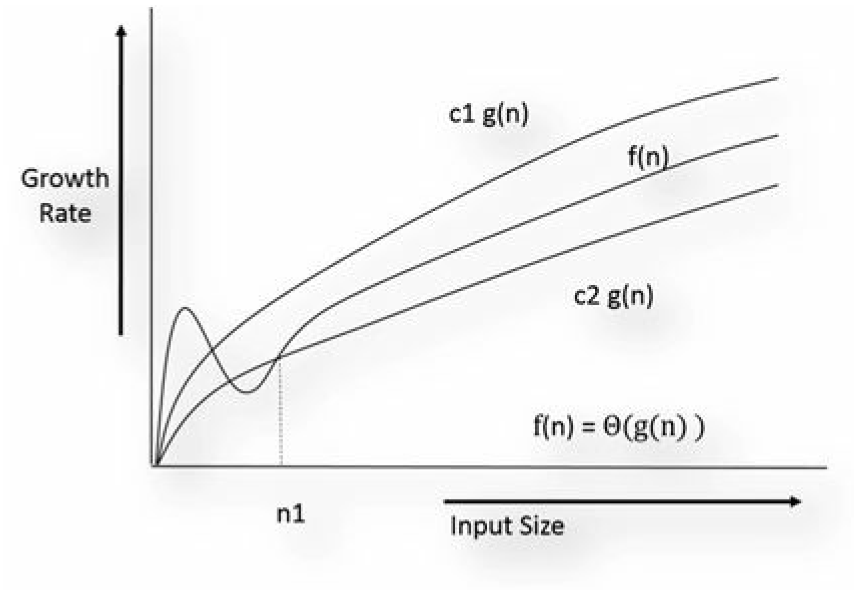


Find relationship of $f(n) = n^c$ and $g(n) = c^n$
f(n) = Ω(g(n))

Theta-Θ Notation

Definition: "f(n) is theta of g(n)." or $f(n) = \Theta(g(n))$ if there are three +ve constants c1, c2 and n0 such that $c1\ g(n) \leq f(n) \leq c2\ g(n)$ for all $n \geq n0$

Function g(n) is an asymptotically tight bound on f(n). Function f(n) grows at the same rate as g(n).



Example: $n^3 + n^2 + n = \Theta(n^3)$
Example: $n^2 + n = \Theta(n^2)$

Find relationship of $f(n) = 2n^2 + n$ and $g(n) = n^2$
$f(n) = O(g(n))$
$f(n) = \Theta(g(n))$
$f(n) = \Omega(g(n))$

Note:- Asymptotic Analysis is not perfect, but that is the best way available for analyzing algorithms.

For example, say there are two sorting algorithms first take f(n) = 10000*n*log(n) and second $f(n) = n^2$ time. The asymptotic analysis says that the first algorithm is better (as it ignores constants) but actually for a small set of data when n is small then 10000, the second algorithm will perform better. To consider this drawback of asymptotic analysis case analysis of the algorithm is introduced.

Complexity analysis of algorithms

1. **Worst Case complexity:** It is the complexity of solving the problem for the worst input of size n. It provides the upper bound for the algorithm. This is the most common analysis done.
2. **Average Case complexity**: It is the complexity of solving the problem on an average. We calculate the time for all the possible inputs and then take an average of it.
3. **Best Case complexity**: It is the complexity of solving the problem for the best input of size n.

Time Complexity Order

A list of commonly occurring algorithm Time Complexity in increasing order:

Name	Notation
Constant	O(1)
Logarithmic	O(log n)
Linear	O(n)
N-LogN	O(n log n)
Quadratic	$O(n^2)$
Polynomial	$O(n^c)$ c is a constant & c>1
Exponential	$O(c^m)$ c is a constant & c>1
Factorial or N-power-N	O(n!) or $O(n^n)$

Constant Time: O(1)

An algorithm is said to run in constant time if the output is produced in constant time regardless of the input size.

Examples:
1. Accessing n^{th} element of a list
2. Push and pop of a stack.
3. Enqueue and remove of a queue.
4. Accessing an element of Hash-Table.

Linear Time: O(n)

An algorithm is said to run in linear time if the execution time of the algorithm is directly proportional to the input size.

Examples:
1. List operations like search element, find min, find max etc.
2. Linked list operations like traversal, find min, find max etc.

Note: when we need to see/ traverse all the nodes of a data-structure for some task then complexity is no less than **O(n)**

Logarithmic Time: O(logn)

An algorithm is said to run in logarithmic time if the execution time of the algorithm is proportional to the logarithm of the input size. Each step of an algorithm, a significant portion of the input is pruned out without traversing it.

Example: Binary search, we will read about these algorithms in this book.

N-LogN Time: O(nlog(n))

An algorithm is said to run in logarithmic time if the execution time of an algorithm is proportional to the product of input size and logarithm of the input size.

Example:
1. Merge-Sort
2. Quick-Sort (Average case)
3. Heap-Sort

Note: Quicksort is a special kind of algorithm to sort a list of numbers. Its worst-case complexity is $O(n^2)$ and average case complexity is O(n log n).

Quadratic Time: $O(n^2)$

An algorithm is said to run in quadratic time if the execution time of an algorithm is proportional to the square of the input size.

Examples:

1. Bubble-Sort
2. Selection-Sort
3. Insertion-Sort

Deriving the Runtime Function of an Algorithm

Constants

Each statement takes a constant time to run. Time Complexity is O(1)

Loops

The running time of a loop is a product of running time of the statement inside a loop and number of iterations in the loop. Time Complexity is O(n)

Nested Loop

The running time of a nested loop is a product of running time of the statements inside loop multiplied by a product of the size of all the loops. Time Complexity is **$O(n^c)$**. Where c is a number of loops. For two loops, it will be **$O(n^2)$**

Consecutive Statements

Just add the running times of all the consecutive statements

If-Else Statement

Consider the running time of the larger of if block or else block. Moreover, ignore the other one.

Logarithmic statement

If each iteration the input size is decreases by a constant factors. Time Complexity = O(log n).

Time Complexity Examples

Example 2.1

```
func fun1(n int) int {
    m := 0
    for i := 0; i < n; i++ {
        m += 1
    }
    return m
}
```

Time Complexity: **O(n)**

Example 2.2

```
func fun2(n int) int {
    m := 0
    for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
            m += 1
        }
    }
    return m
}
```

Time Complexity: $O(n^2)$

Example 2.3

```
func fun3(n int) int {
    m := 0
    for i := 0; i < n; i++ {
        for j := 0; j < i; j++ {
            m += 1
        }
    }
    return m
}
```

Time Complexity: $O(N+(N-1)+(N-2)+...) == O(N(N+1)/2) == O(n^2)$

Example 2.4

```
func fun4(n int) int {
    m := 0
    i := 1
    for i < n {
        m += 1
        i = i * 2
    }
    return m
}
```

Each time problem space is divided into half. Time Complexity: **O(log(n))**

Example 2.5

```
func fun5(n int) int {
    m := 0
```

```
    i := n
    for i > 0 {
        m += 1
        i = i / 2
    }
    return m
}
```

Same as above each time problem space is divided into half. Time Complexity: **O(log(n))**

Example 2.6

```
func fun6(n int) int {
    m := 0
    for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
            for k := 0; k < n; k++ {
                m += 1
            }
        }
    }
    return m
}
```

Outer loop will run for n number of iterations. In each iteration of the outer loop, inner loop will run for n iterations of their own. Final complexity will be n*n*n.

Time Complexity: $\mathbf{O(n^3)}$

Example 2.7

```
func fun7(n int) int {
    m := 0
    for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
            m += 1
        }
    }
    for i := 0; i < n; i++ {
        for k := 0; k < n; k++ {
            m += 1
        }
    }
    return m
}
```

These two groups of for loop are in consecutive so their complexity will add up to form the final complexity of the program. Time Complexity: $\mathbf{O(n^2) + O(n^2) = O(n^2)}$

Example 2.8

```
func fun8(n int) int {
    m := 0
    for i := 0; i < n; i++ {
        sq := math.Sqrt(float64(n))
        for j := 0; j < int(sq); j++ {
            m += 1
```

```
        }
    }
    return m
}
```

Time Complexity: $\mathbf{O(n * \sqrt{n}) = O(n^{3/2})}$

Example 2.9

```
func fun9(n int) int {
    m := 0
    for i := n; i > 0; i /= 2 {
        for j := 0; j < i; j++ {
            m += 1
        }
    }
    return m
}
```

Each time problem space is divided into half. Time Complexity: **O(log(n))**

Example 2.10

```
func fun10(n int) int {
    m := 0
    for i := 0; i < n; i++ {
        for j := i; j > 0; j-- {
            m += 1
        }
    }
    return m
}
```

O(N+(N-1)+(N-2)+...) = O(N(N+1)/2) // arithmetic progression.
Time Complexity: $\mathbf{O(n^2)}$

Example 2.11

```
func fun11(n int) int {
    m := 0
    for i := 0; i < n; i++ {
        for j := i; j < n; j++ {
            for k := j + 1; k < n; k++ {
                m += 1
            }
        }
    }
    return m
}
```

Time Complexity: $\mathbf{O(n^3)}$

Example 2.12

```
func fun12(n int) int {
    j := 0
    m := 0
```

```
    for i := 0; i < n; i++ {
        for ; j < n; j++ {
            m += 1
        }
    }
    return m
}
```

Think carefully once again before finding a solution, j value is not reset at each iteration.
Time Complexity: **O(n)**

Example 2.13

```
func fun13(n int) int {
    m := 0
    for i := 1; i <= n; i *= 2 {
        for j := 0; j <= i; j++ {
            m += 1
        }
    }
    return m
}
```

The inner loop will run for 1, 2, 4, 8,… n times in successive iteration of the outer loop.
Time Complexity: T(n) = O(1+ 2+ 4+ ….+n/2+n) = **O(n)**

Example 2.14

```
func main1() {
    fmt.Println("N = 100, Number of instructions O(n):: ", fun1(100))
    fmt.Println("N = 100, Number of instructions O(n^2):: ", fun2(100))
    fmt.Println("N = 100, Number of instructions O(n^2):: ", fun3(100))
    fmt.Println("N = 100, Number of instructions O(log(n)):: ", fun4(100))
    fmt.Println("N = 100, Number of instructions O(log(n)):: ", fun5(100))
    fmt.Println("N = 100, Number of instructions O(n^3):: ", fun6(100))
    fmt.Println("N = 100, Number of instructions O(n^2):: ", fun7(100))
    fmt.Println("N = 100, Number of instructions O(n^(3/2)):: ", fun8(100))
    fmt.Println("N = 100, Number of instructions O(log(n)):: ", fun9(100))
    fmt.Println("N = 100, Number of instructions O(n^2):: ", fun10(100))
    fmt.Println("N = 100, Number of instructions O(n^3):: ", fun11(100))
    fmt.Println("N = 100, Number of instructions O(n):: ", fun12(100))
    fmt.Println("N = 100, Number of instructions O(n):: ", fun13(100))
}
```

Output:

```
N = 100, Number of instructions O(n)::  100
N = 100, Number of instructions O(n^2)::  10000
N = 100, Number of instructions O(n^2)::  4950
N = 100, Number of instructions O(log(n))::  7
N = 100, Number of instructions O(log(n))::  7
N = 100, Number of instructions O(n^3)::  1000000
N = 100, Number of instructions O(n^2)::  20000
N = 100, Number of instructions O(n^(3/2))::  1000
N = 100, Number of instructions O(log(n))::  197
N = 100, Number of instructions O(n^2)::  4950
```

```
N = 100, Number of instructions O(n^3)::  166650
N = 100, Number of instructions O(n)::  100
N = 100, Number of instructions O(n)::  134
```

Master Theorem

The master theorem solves recurrence relations of the form: $T(n) = a\,T(n/b) + f(n)$

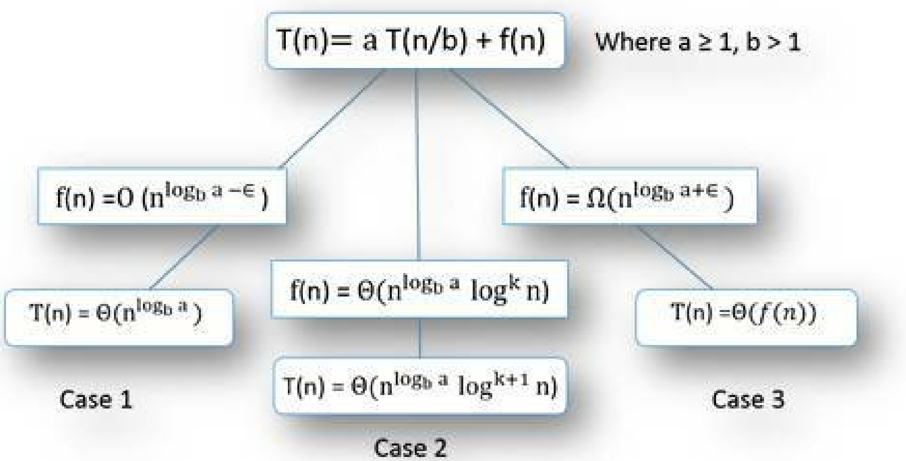
Where $a \geq 1$ and $b > 1$.
"n" is the size of the problem. "a" is a number of sub problem in the recursion. “n/b” is the size of each sub-problem. "f(n)" is the cost of the division of the problem into sub problem and merge of results of sub problems to get the final result.

It is possible to determine an asymptotic tight bound in these three cases:

· Case 1: when $f(n)=O(n^{\log_b a - \epsilon})$ and constant $\epsilon > 1$, then the final Time Complexity will be: $T(n) = \Theta(n^{\log_b a})$

· Case 2: when $f(n) = \Theta(n^{\log_b a} \log^k n)$ and constant $k \geq 0$, then the final Time Complexity will be: $T(n)= \Theta(n^{\log_b a} \log^{k+1} n)$

· Case 3: when $f(n) = \Omega(n^{\log_b a+\epsilon})$ and constant $\epsilon > 1$, Then the final Time Complexity will be: $T(n) = \Theta(f(n))$



Example 2.15: Take an example of Merge-Sort, $T(n) = 2\,T(n/2) + n$
Sol:- $\log_b a = \log_2 2 = 1$

$f(n) = n = \Theta(n^{\log_2 2} \log^0 n)$

Case 2 applies and $T(n)= \Theta(n^{\log_2 2} \log^{0+1} n)$
$T(n) = \Theta(n \log(n))$

Example 2.16: Binary Search $T(n) = T(n/2) + O(1)$
Sol:- $\log_b a = \log_2 1 = 0$

$f(n) = 1 = \Theta(n^{\log_2 1} \log^0 n)$

Case 2 applies and $T(n)= \Theta(n^{\log_2 1} \log^{0+1} n)$
$T(n) = \Theta(\log(n))$

Example 2.17: Binary tree traversal $T(n) = 2T(n/2) + O(1)$
Sol:- $\log_b a = \log_2 2 = 1$

$f(n) = 1 = O(n^{\log_2 2-1})$

Case 1 applies and $T(n) = \Theta(n^{\log_2 2})$
$T(n) = \Theta(n)$

Example 2.18: Take an example $T(n) = 2\ T(n/2) + n^2$

Sol:- $\log_b a = \log_2 2 = 1$

$f(n) = n^2 = \Omega(n^{\log_2 2+1})$

Case 3 applies and $T(n) = \Theta(f(n))$

$T(n) = \Theta(n^2)$

Example 2.19: Take an example $T(n) = 4\ T(n/2) + n^2$

Sol:- $\log_b a = \log_2 4 = 2$

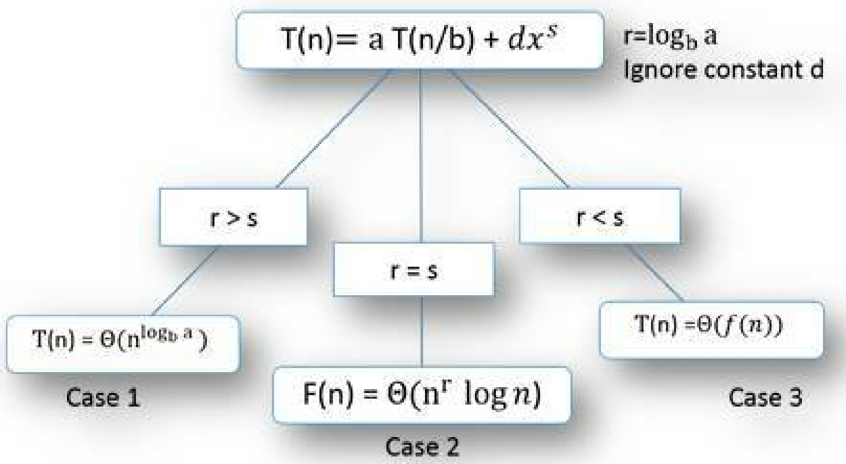
$f(n) = n^2 = \Theta(n^{\log_2 4} \log^0 n)$

Case 2 applies and $T(n)=\Theta(n^{\log_2 4} \log^{0+1} n)$

$T(n) = \Theta(n^2 \log n)$

Modified Master theorem

This is a shortcut to solving the same problem easily and fast. If the recurrence relation is in the form of T(n) = a T(n/b) + dx^s



Example 2.20: T(n) = 2 T(n/2) + n^2
Sol:- r = $\log_2 2$ = 1
s = 2
Case 3: r < s
T(n) = Θ (f(n)) = $\Theta(n^2)$

Example 2.21: T(n) = T(n/2) + 2n
Sol:- r = $\log_2 1$ = 0
s = 1
Case 3
T(n)= Θ(n)

Example 2.22: T (n) = 16T (n/4) + n
Sol:- r = 2 , s = 1
Case 1
T(n)= $\Theta(n^2)$

Example 2.23: T (n) = 2T (n/2) + n log n
Sol:- There is logn in f(n) so use master theorem, shortcut will not word.
$f(n) = n \log(n) = \Theta(n^{\log_2 2} \log^1 n)$
$T(n) = \Theta(n^{\log_2 2} \log^{0+1} n)$ = Θ(n log(n))

Example 2.24: T(n) = 2 T(n/4) + $n^{0.5}$
Sol:- r = $\log_4 2$ = 0.5 = s
Case 2: $T(n) = \Theta(n^{\log_4 2} \log^{0.5+1} n)$ = $\Theta(n^{0.5} \log^{1.5} n)$

Example 2.25: T(n) = 2 T(n/4) + $n^{0.49}$
Sol:- Case 1:
$T(n) = \Theta(n^{\log_4 2})$ = $\Theta(n^{0.5})$

Example 2.26: T (n) = 3T (n/3) + √ n

Sol:-   r = $\log_3 3$ = 1

s = ½

Case 1

T(n) = Θ(n)

Example 2.27:T (n) = 3T (n/4) + n log n

Sol:- There is logn in f(n) so see if master theorem applies.

f(n) = n log n = $\Omega(n^{\log_4 3} \log^1 n)$

Case 3:

T(n) = Θ(n log(n))

Example 2.28: T (n) = 3T (n/3) + n/2

Sol:-   r=1=s

Case 2:

T(n) = Θ(n log(n))

Exercise

1. True or false
   a. $5 n + 10 n^2 = O(n^2)$
   b. $n \log n + 4 n = O(n)$
   c. $\log(n^2) + 4 \log(\log n) = O(\log n)$
   d. $12 n^{1/2} + 3 = O(n^2)$
   e. $3^n + 11 n^2 + n^{20} = O(2^n)$

2. What is the best-case runtime complexity of searching a list?

3. What is the average-case runtime complexity of searching a list?

CHAPTER 3: APPROACH TO SOLVE ALGORITHM DESIGN PROBLEMS

Introduction

Theoretical knowledge of the algorithm is essential, but it is not sufficient. When an interviewer asks to develop a program in an interview, than interviewee should follow our five-step approach to solve it. Master this approach and you will perform be better than most of the candidates in interviews.

Five steps for solving algorithm design questions are:
1. Constraints
2. Ideas Generation
3. Complexities
4. Coding
5. Testing

Constraints

Solving a technical question is not just about knowing the algorithms and designing a good software system. The interviewer wants to know you approach towards any given problem. Many people make mistakes, as they do not ask clarifying questions about a given problem. They assume many things and begin working with that. There is lot of data that is missing that you need to collect from your interviewer before beginning to solve a problem.

In this step, you will capture all the constraints about the problem. We should never try to solve a problem that is not completely defined. Interview questions are not like exam paper where all the details about a problem are well defined. In the interview, the interviewer actually expects you to ask questions and clarify the problem.

For example, when the problem statement says that write an algorithm to sort numbers.

1. The first information you need to capture is what king of data. Let us suppose interviewer respond with the answer Integer.
2. The second information that you need to know what is the size of data. Your algorithm differs if the input data size if 100 integers or 1 billion integers.

Basic guideline for the Constraints for a list of numbers:

1. How many numbers of elements in the list?
2. What is the range of value in each element? What is the min and max value?
3. What is the kind of data in each element is it an integer or a floating point?
4. Does the list contain unique data or not?

Basic guideline for the Constraints for a list of string:

1. How many numbers of elements in the list?
2. What is the length of each string? What is the min and max length?
3. Does the list contain unique data or not?

Basic guideline for the Constraints for a Graph

1. How many nodes are there in the graph?
2. How many edges are there in the graph?
3. Is it a weighted graph? What is the range of weights?
4. Is the graph directed or undirected?
5. Is there is a loop in the graph?
6. Is there negative sum loop in the graph?
7. Does the graph have self-loops?

We will see this in graph chapter that depending upon the constraints the algorithm applied changes and so is the complexity of the solution.

Idea Generation

We will covered a lot of theoretical knowledge in this book. It is impossible to cover all the questions as new ones are created every day. Therefore, we should know how to handle new problems. Even if you know the solution of a problem asked by the interviewer then also you need to have a discussion with the interviewer and reach to the solution. You need to analyze the problem also because the interviewer may modify a question a little bit and the approach to solve it will vary.

How to solve a new problem? The solution to this problem is that you need to do a lot of practice and the more you practice the more you will be able to solve any new question, which come in front of you. When you have solved enough problems, you will be able to see a pattern in the questions and able to solve new problems easily.

Following is the strategy that you need to follow to solve an unknown problem:

1. Try to simplify the task in hand.
2. Try a few examples
3. Think of a suitable data-structure.
4. Think about similar problems you have already solved.

Try to simplify the task in hand

Let us look into the following problem: Husbands and their wives are standing in random in a line. They have been numbered for husbands H1, H2, H3 and so on. Their corresponding wives have number W1, W2, W3 and so on. You need to arrange them so that H1 will stand first, followed by W1, then H2 followed by W2 and so on.

At the first look, it looks difficult, but it is a simple problem. Try to find a relation of the final position.
P(Hi) = i*2 – 1,   P(Wi) = i*2

The rest of the algorithm we are leaving you to do something like Insertion-Sort and you are done.

Try a few examples

In the same above problem if you have tried it with some example for 3 husband and wife pair then you may have reached to the same formula that we have shown in the previous section. Some time thinking some more examples try to solve the problem at hand.

Think of a suitable data-structure

For some problems, it is straightforward to choose which data structure will be most suitable. For example, if we have a problem finding min/max of some given value, then probably heap is the data structure we are looking for. We have seen a number of data structure throughout this book. We have to figure out which data-structure will suite our need.

Let us look into a problem: We are given a stream of data at any time we can be asked to tell the median value of the data and maybe we can be asked to pop median data.

We can think about some sort of tree, may be balanced tree where the root is the median. Wait but it is not so easy to make sure the tree root to be a median.
A heap can give us minimum or maximum so we cannot get the desired result from it too. However, what if we use two heap one max heap and one min heap. The smaller values will go to max heap and the bigger values will go to min heap. In addition, we can keep the count of how many elements are there in the heap. The rest of the algorithm you can think yourself.

For every new problem think about the data structure, you know and may be one of them or some combination of them will solve your problem.

Think about similar problems you have already solved. Let us suppose you are given, two linked list head reference and they meet at some point and need to find the point of intersection. However, in place of the end of both the linked list to be a null reference there is a loop.

You know how to find intersection point of two intersecting linked-list, you know how to find if a linked list have a loop (three-reference solution). Therefore, you can apply both of these solutions to find the solution of the problem in hand.

Complexities

Solving a problem is not just finding a correct solution. The solution should be fast and should have reasonable memory requirement. You have already read about Big-O notation in the previous chapters. You should be able to do Big-O analysis. In case you think the solution you have provided is not that optimal and there is some more efficient solution, then think again and try to figure out this information.

Most interviewers expect that you should be able to find the time and Space Complexity of the algorithms. You should be able to compute the time and Space Complexity instantly. Whenever you are solving some problem, you should find the complexity associated with it from this you would be able to choose the best solutions. In some problems there is some trade-offs between space and Time Complexity, so you should know these trade-offs. Sometime taking some bit more space saves a lot of time and make your algorithm much faster.

Coding

At this point, you have already captured all the constraints of the problem, proposed few solutions, evaluated the complexities of the various solutions and picked the one solution to do final coding. Never ever, jump into coding before discussing constraints, Idea generation and complexity with the interviewer.

We are accustomed to coding in an IDE like visual studio. So many people struggle when asked to write code on a whiteboard or some blank sheet. Therefore, we should have a little practice to the coding on a sheet of paper. You should think before coding because there is no back button in sheet of paper. Always try to write modular code. Small functions need to be created so that the code is clean and managed. If there is a swap function so just use this function and tell the interviewer that you will write it later. Everybody knows that you can write swap code.

Testing

Once the code is written, you are not done. It is most important that you test your code with several small test cases. It shows that you understand the importance of testing. It also gives confidence to your interviewer that you are not going to write a buggy code. Once you are done with, your coding it is a good practice that you go through your code line-by-line with some small test cases. This is just to make sure your code is working as it is supposed to work.

You should test few test cases.
Normal test cases: These are the positive test cases, which contain the most common scenario, and focus is on the working of the base logic of the code. For example, if we are solving some problem for linked list, then this test may contain, what will happen when a linked list with 3 or 4 nodes is given as input. These test cases you should always run in your head before saying the code is done.

Edge cases: These are the test cases, which are going to test the boundaries of the code. For the same linked list algorithm, edge cases may be how the code behaves when an empty list is passed or just one node is passed. Edge cases may help to make your code more robust. Just few checks need to be added to the code to take care of these conditions.

Load testing: In this kind of test, your code will be tested with a huge data. This will allow us to test if your code is slow or too much memory intensive.

Always follow these five steps never jump to coding before doing constraint analysis, idea generation, and Complexity Analysis: At least never, miss the testing phase.

Example

Let us suppose the interviewer ask you to give a best sorting algorithm.
Some interviewee will directly jump to Quick-Sort **O(nlogn)**. Oops, mistake you need to ask many questions before beginning to solve this problem.

Questions 1: What is the kind of data? Are they integers?
Answer: Yes, they are integers.

Questions 2: How much data are we going to sort?
Answer: May be thousands.

Questions 3: What exactly is this data about?
Answer: They store a person's age

Questions 4: What kind of data-structure used to hold this data?
Answer: Data are given in the form of some list

Questions 5: Can we modify the given data-structure? In addition, many, many more…?
Answer: No, you cannot modify the data structure provided

Ok from the first answer, we will deduce that the data is integer. The data is not so big it just contains a few thousand entries. The third answer is interesting from this we deduce that the range of data is 1-150. Data is provided in a list. From fifths answer we deduce that we have to create our own data structure and we cannot modify the list provided. So finally, we conclude, we can just use bucket sort to sort the data. The range is just 1-150 so we need just 151-capacity integral list. Data is under thousands so we do not have to worry about data overflow and we get the solution in linear time **O(N)**.

Note: We will read sorting in the coming chapters.

Summary

At this point, you know the process of handling new problems very well. In the coming chapter we will be looking into a lot of various data structure and the problems they solve. It may be the user is not able to understand some portion of this chapter as knowledge of rest of the book is needed so they can read this chapter again after they had read the rest of the data structures portion. A huge number of problems are solved in this book. However, it is recommended that first try to solve them by yourself, and then look for the solution. Always think about the complexity of the problem. In the interview interaction is the key to get problem described completely and discuss your approach with the interviewer.

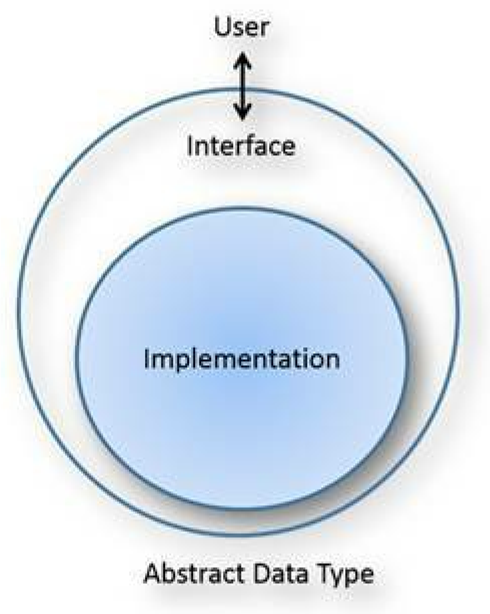
CHAPTER 4: ABSTRACT DATA TYPE & GO COLLECTIONS

Abstract data type (ADT)

An abstract data type (ADT) is a logical description of how we view the data and the operations that are allowed on it. ADT is defined as a user point of view of a data type. ADT concerns about the possible values of the data and what are interface exposed by it.

ADT does not concern about the actual implementation of the data structure.

For example, a user wants to store some integers and find a mean of it. Does not talk about how exactly it will be implemented.



Data-Structure

Data structures are concrete representations of data and are defined as a programmer point of view. Data-structure represents how data will be stored in memory. All data-structures have their own pros and cons. Depending upon the type problem we pick a data-structure that is best suited for it.

For example, we can store data in an array, a linked-list, stack, queue, tree, etc.

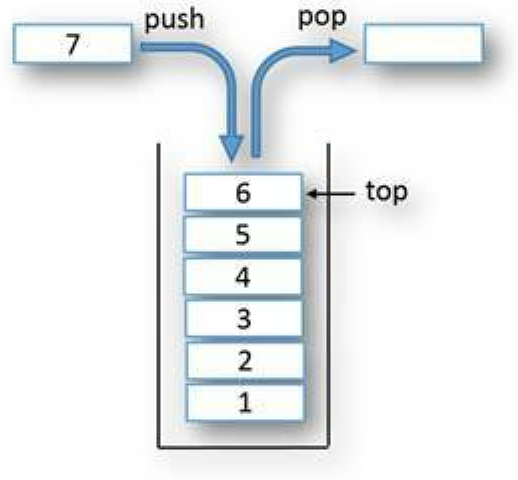
Go Collection Framework

Go programming language provides a Go Collection Framework, which is a set of high quality, high performance & reusable data-structures and algorithms.

The following advantages of using a Go collection framework:

1. Programmers do not have to implement basic data structures and algorithms repeatedly. Thereby it prevents the reinvention the wheel. Thus, the programmer can devote more effort in business logic.
2. The Go Collection Framework code is well-tested, high quality, high performance code there by increasing the quality of the programs.
3. Development cost is reduced as basic data structures and algorithms are implemented in Collections framework.
4. Easy for the review and understanding others programs as other developers also use the Collection framework. In addition, collection framework is well documented.

Stack



Stack is a special kind of data structure that follows Last-In-First-Out (LIFO) strategy. This means that the element that is added last will be the first to be removed.

The various applications of stack are:

1. Recursion: recursive calls are preformed using system stack.
2. Postfix evaluation of expression.
3. Backtracking
4. Depth-first search of trees and graphs.
5. Converting a decimal number into a binary number etc.

Stack ADT Operations

- **Push(k):** Adds a new item to the top of the stack
- **Pop():** Remove an element from the top of the stack and return its value.
- **Top():** Returns the value of the element at the top of the stack
- **Size():** Returns the number of elements in the stack
- **IsEmpty():** determines whether the stack is empty. It returns 1 if the stack is empty or return 0.

Note: All the above Stack operations are implemented in **O(1)** Time Complexity.

Stack implementation using Go collections

Example 4.1

```
import "github.com/golang-collections/collections/stack"
func main() {
    s := stack.New()
    s.Push(2)
    s.Push(3)
    s.Push(4)

    for s.Len() != 0 {
        val := s.Pop()
        fmt.Print(val, " ")
    }
```

```
}
```

Output:

```
4 3 2
```

Queue

A queue is a First-In-First-Out (FIFO) kind of data structure. The element that is added to the queue first will be the first to be removed and so on.

Queue ADT Operations:

- **Add():** Add a new element to the back of the queue.
- **Remove():** remove an element from the front of the queue and return its value.
- **Front():** return the value of the element at the front of the queue.
- **Size():** returns the number of elements inside the queue.
- **IsEmpty():** returns 1 if the queue is empty otherwise return 0

Note: All the above Queue operations are implemented in **O(1)** Time Complexity.

Queue implementation in Go Collection

Deque is the class implementation of doubly ended queue. If we use append(), popleft() it will behave as a queue.

Example 4.2

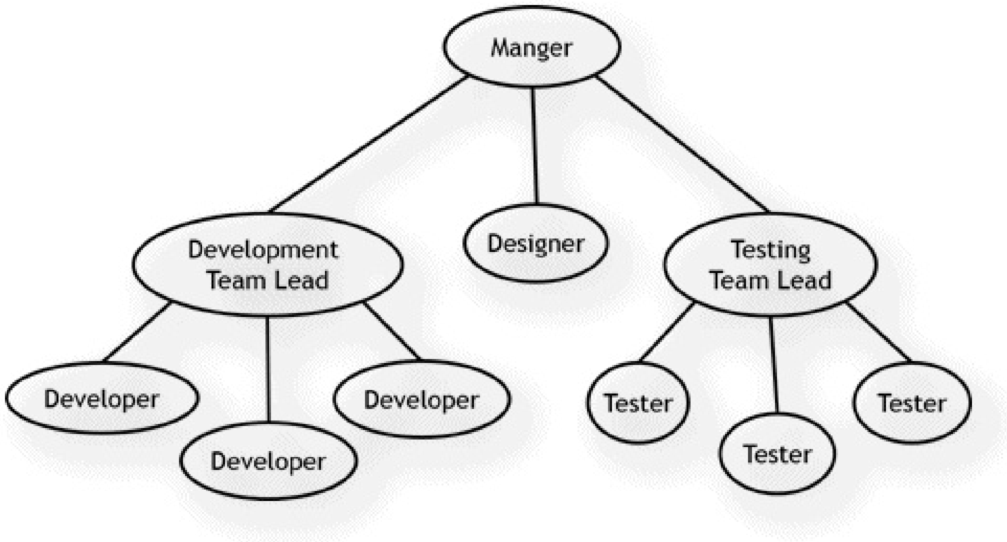
```
import "github.com/golang-collections/collections/queue"
func main() {
    q := queue.New()
    q.Enqueue(2)
    q.Enqueue(3)
    q.Enqueue(4)

    for q.Len() != 0 {
        val := q.Dequeue()
        fmt.Print(val, " ")
    }
}
```

Output

```
2 3 4
```

Tree

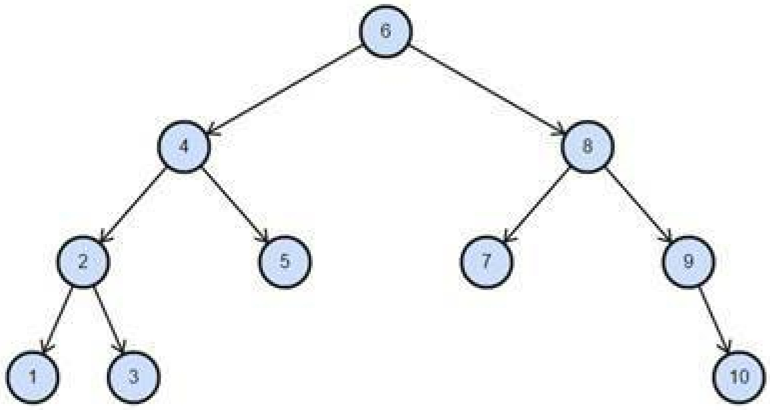
Tree is a hierarchical data structure. The top element of a tree is called the root of the tree. Except the root element, every element in a tree has a parent element, and zero or more child elements. The tree is the most useful data structure when you have hierarchical information to store.



Binary Tree

A binary tree is a type of tree in which each node has at most two children (0, 1 or 2) which are referred as left child and right child.

Binary Search Trees (BST)



A binary search tree (BST) is a binary tree on which nodes are ordered in the following way:
1. The key in the left subtree is less than the key in its parent node.
2. The key in the right subtree is greater or equal the key in its parent node.

Binary Search Tree ADT Operations

- **Insert(k):** Insert an element k into the tree.
- **Delete(k):** Delete an element k from the tree.
- **Search(k):** Search a particular value k into the tree if it is present or not.
- **FindMax():** Find the maximum value stored in the tree.
- **FindMin():** Find the minimum value stored in the tree.

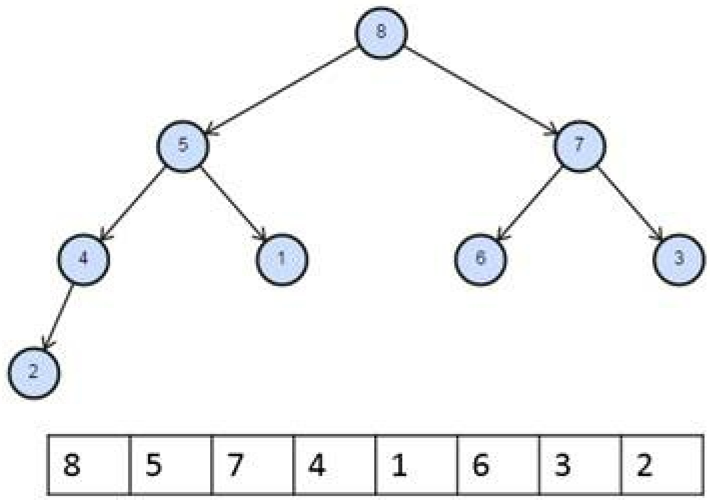
The average Time Complexity of all the above operations on a binary search tree is O(log n), the case when the tree is balanced. The worst-case Time Complexity will be **O(n)** when the tree is skewed. A binary tree is skewed when tree is not balanced.

There are two types of skewed tree.
1. Right Skewed binary tree: A binary tree in which each node is having either a right child or no child.
2. Left Skewed binary tree: A binary tree in which each node is having either a left child or no child.

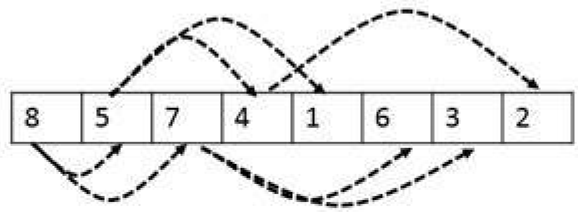
Balanced Binary search tree

There are few binary search tree, which keeps themselves balanced. Most important among them are Red-Black Tree and AVL tree. Ordered dictionary in collections is implemented using Red Black-Tree.

Priority Queue (Heap)



Max Heap



Priority queue is implemented using a binary heap data structure. In a heap, the records are stored in an array. Each node in the heap follows the same rule that the parent value is greater than its children are.

There are two types of the heap data structure:

1. Max heap: each node should be greater than or equal to each of its children.
2. Min heap: each node should be smaller than or equal to each of its children.

A heap is a useful data structure when you want to get max/min one by one from data. Heap-Sort uses max heap to sort data in increasing/decreasing order.

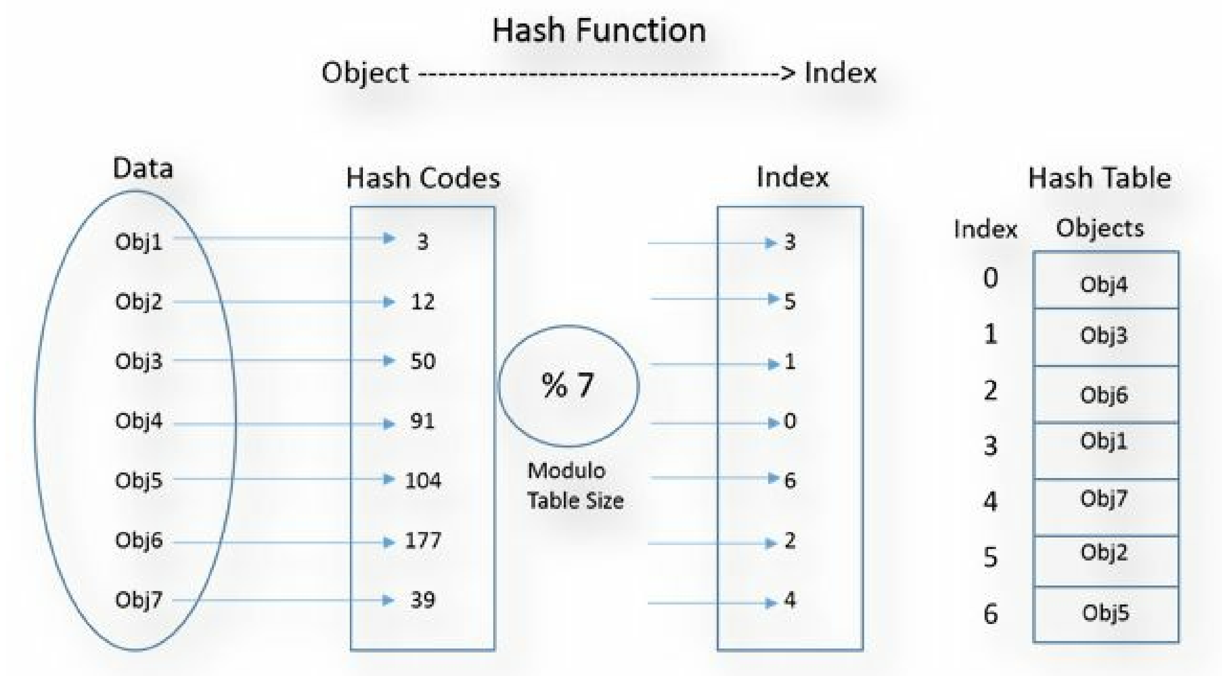
Heap ADT Operations

- **Insert() -** Adding a new element to the heap. The Time Complexity of this operation is O(log(n))
- **remove() -** Extracting max for max heap case (or min for min heap case). The Time Complexity of this operation is O(log(n))
- **Heapify()** – To convert a list of numbers in a list into a heap. This operation has a Time Complexity **O(n)**

PriorityQueue implementation

For implementation of priority queue, please refer Priority queue chapter.

Hash-Table



A Hash-Table is a data structure that maps keys to values. Each position of the Hash-Table is called a slot. The Hash-Table uses a hash function to calculate an index of a list. We use the Hash-Table when the number of key actually stored is small relatively to the number of possible keys.

The process of storing objects using a hash function is as follows:
1. Create a list of size M to store objects; this list is called Hash-Table.
2. Find a hash code of an object by passing it through the hash function.
3. Take module of hash code by the size of Hash-Table to get the index where objects will be stored.
4. Finally store these objects in the designated index.

The process of searching objects in Hash-Table using a hash function is as follows:
1. Find a hash code of the object we are searching for by passing it through the hash function.
2. Take module of hash code by the size of Hash-Table to get the index where object is stored.
3. Finally, retrieve the object from the designated index.

Hash-Table Abstract Data Type (ADT)

ADT of Hash-Table contains the following functions:
Insert(x): Add object x to the data set.
Delete(x): Delete object x from the data set.
Search(x): Search object x in data set.
The Hash-Table is a useful data structure for implementing dictionary. The average time to search for an element in a Hash-Table is **O(1)**. A Hash Table generalizes the notion of a list.

Map / Dictionary implementation in Go Collection

A Dictionary is a data structure that maps keys to values. A Dictionary is using a hash table so the key value pairs are not stored in sorted order. Dictionary does not allow duplicate keys buts values can be duplicate.

Example 4.3

```go
func main() {
	m := make(map[string]int)
	m["Apple"] = 40
	m["Banana"] = 30
	m["Mango"] = 50
	for key, val := range m {
		fmt.Print("[ ",key," -> ", val," ]")
	}
	fmt.Println("Apple price:", m["Apple"])
	delete(m, "Apple")
	fmt.Println("Apple price:", m["Apple"])

	v, ok := m["Apple"]
	fmt.Println("Apple price:", v, "Present:", ok)

	v2, ok2 := m["Banana"]
	fmt.Println("Banana price:", v2, "Present:", ok2)
}
```

Output:

```
[ Apple -> 40 ][ Banana -> 30 ][ Mango -> 50 ]
Apple price: 40
Apple price: 0
Apple price: 0 Present: false
Banana price: 30 Present: true
```

Set implementation of Go Collections

Set is used to store only unique elements. Set is implemented using a hash table so elements are not stored in sequential order.

Example 4.4: Set using go collection.

```go
import "github.com/golang-collections/collections/set"

func main1() {
	st := set.New()
	st.Insert(1)
	st.Insert(2)
	fmt.Println(st.Has(1))
	fmt.Println(st.Has(3))
}
```

Example 4.5 : Set implemented using Map.

```go
type Set map[interface{}]bool

func (s *Set) Add(key interface{}) {
	(*s)[key] = true
}

func (s *Set) Remove(key interface{}) {
	delete((*s), key)
}
```

```
func (s *Set) Find(key interface{}) bool {
	return (*s)[key]
}

func main() {
	mp := make(Set)
	mp.Add("a")
	mp.Add("b")
	mp.Add("a")
	fmt.Println(mp.Find("a"))
	fmt.Println(mp.Find("b"))
	fmt.Println(mp.Find("c"))
}
```

Counter implementation

Counters are used to count the number of occurrence of values in a List.

Example 4.6

```
type Counter map[interface{}]int

func (s *Counter) Add(key interface{}) {
	(*s)[key] += 1
}

func (s *Counter) Find(key interface{}) bool {
	_, ok := (*s)[key]
	return ok
}

func (s *Counter) Get(key interface{}) (int, bool) {
	val, ok := (*s)[key]
	return val, ok
}

func main() {
	mp := make(Counter)
	mp.Add("a")
	mp.Add("b")
	mp.Add("a")
	fmt.Println(mp.Find("a"))
	fmt.Println(mp.Find("b"))
	fmt.Println(mp.Find("c"))
	fmt.Println(mp.Get("a"))
	fmt.Println(mp.Get("b"))
	fmt.Println(mp.Get("c"))
}
```

Output

```
true
true
```

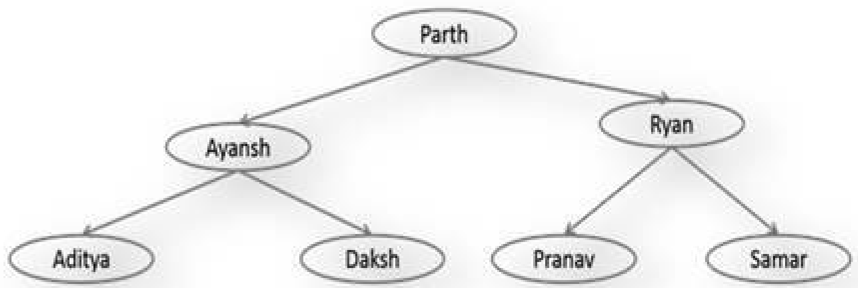
```
false
2 true
1 true
0 false
```

Dictionary / Symbol Table

A symbol table is a mapping between a string (key) and a value, which can be of any data type. A value can be an integer such as occurrence count, dictionary meaning of a word and so on.

Binary Search Tree (BST) for Strings

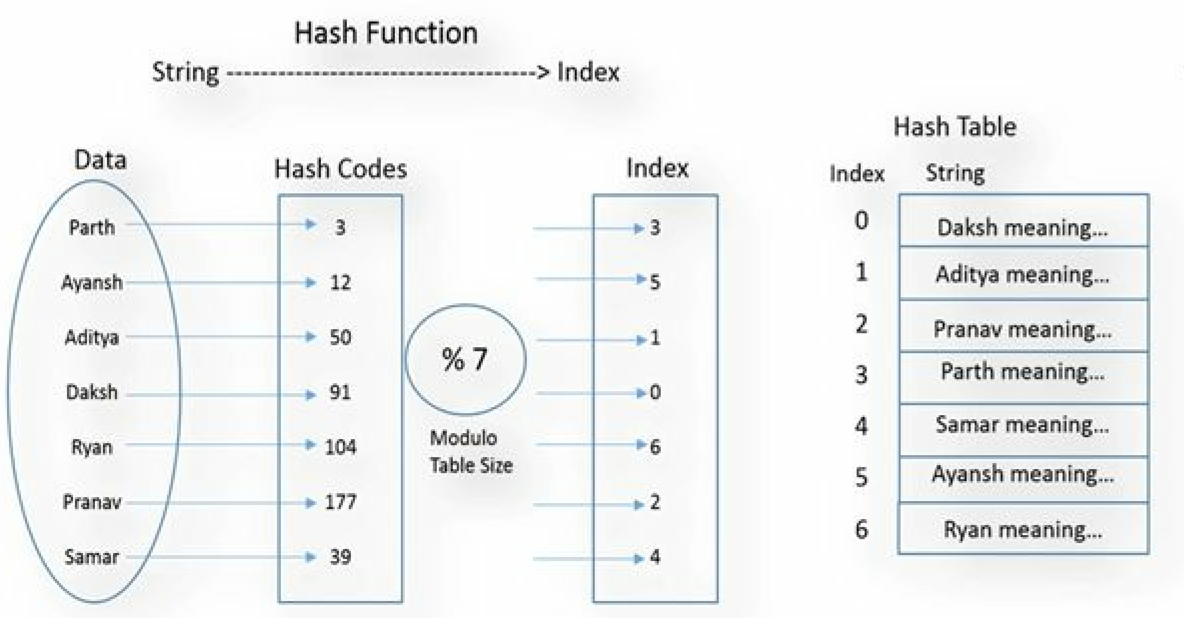
Binary Search Tree (BST) is the simplest way to implement symbol table. Simple string compare function can be used to compare two strings. If all the keys are random, and the tree is balanced. Then on an average key lookup can be done in logarithmic time.



BINARY SEARCH TREE AS DICTIONARY

Hash-Table

The Hash-Table is another data structure, which can be used for symbol table implementation. Below Hash-Table diagram, we can see the name of that person is taken as the key, and their meaning is the value of the search. The first key is converted into a hash code by passing it to appropriate hash function. Inside hash function the size of Hash-Table is also passed, which is used to find the actual index where values will be stored. Finally, the value that is meaning of name is stored in the Hash-Table, or you can store a reference to the string which store meaning can be stored into the Hash-Table.



Hash-Table has an excellent lookup of constant time.

Let us suppose we want to implement autocomplete the box feature of Google search. When you type some string to search in google search, it proposes some complete string even before you have done typing. BST cannot solve this problem as related strings can be in both right and left subtree.

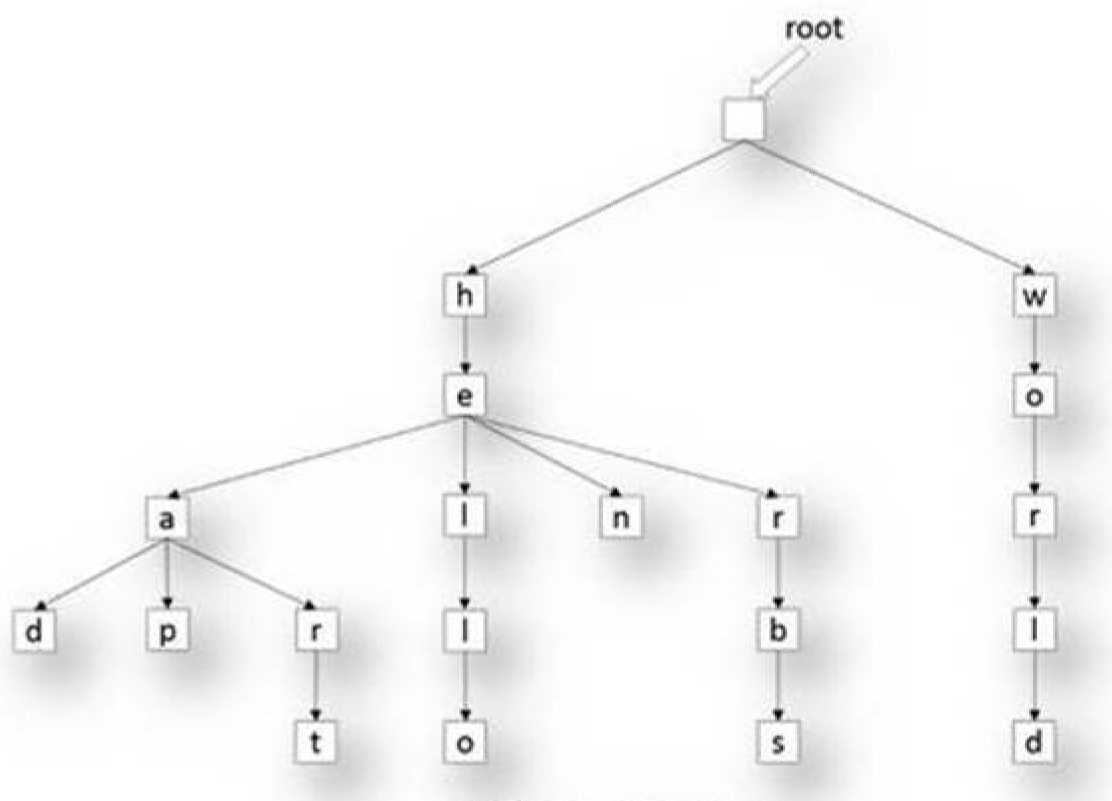
The Hash-Table is also not suited for this job. One cannot perform a partial match or range query on a Hash-Table. Hash function transforms string to a number. Moreover, a good hash function will give a distributed hash code even for partial string and there is no way to relate two strings in a Hash-Table.

Trie and Ternary Search tree are a special kind of tree, which solves partial match, and range query problem well.

Trie

Trie is a tree, in which we store only one character at each node. This final key value pair is stored in the leaves. Each node has K children, one for each possible character. For simplicity purpose, let us consider that the character set is 26, corresponds to different characters of English alphabets.

Trie is an efficient data structure. Using Trie, we can search the key in O(M) time. Where M is the maximum string length. Trie is suitable for solving partial match and range query problems.

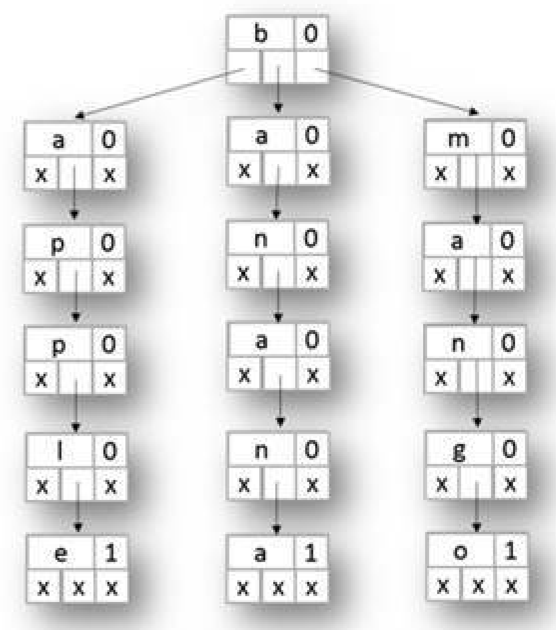


TRIE DICTIONARY

Ternary Search Trie/ Ternary Search Tree

Tries having a very good search performance of O(M) where M is the maximum size of the search string. However, tries having very high space requirement. Every node Trie contain references to multiple nodes, each reference corresponds to possible characters of the key. To avoid this high space requirement Ternary Search Trie (TST) is used. A TST avoid the heavy space requirement of the traditional Trie while still keeping many of its advantages. In a TST, each node contains a character, an end of key indicator, and three references. The three references are corresponding to current char hold by the node (equal), characters less than and character greater than.

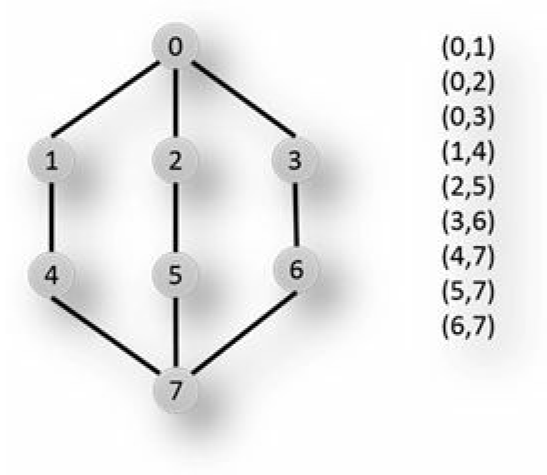
The Time Complexity of ternary search tree operation is proportional to the height of the ternary search tree. In the worst case, we need to traverse up to 3 times that many links. However, this case is rare. Therefore, TST is a very good solution for implementing Symbol Table, Partial match and range query.



Ternary Search Tree

Graphs

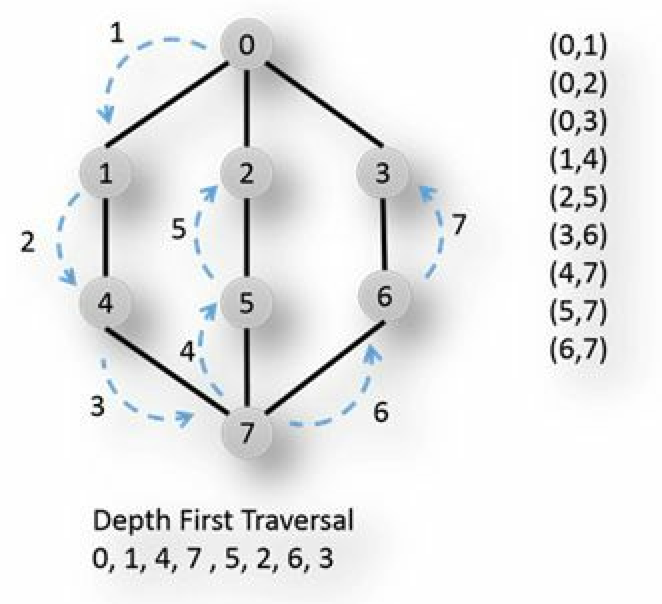
A graph is a data structure that represents a network that connects a collection of nodes called vertices, and their connections, called edges. An edge can be seen as a path between two nodes. These edges can be either directed or undirected. If a path is directed then you can move only in one direction, while in an undirected path you can move in both the directions.



Graph Algorithms

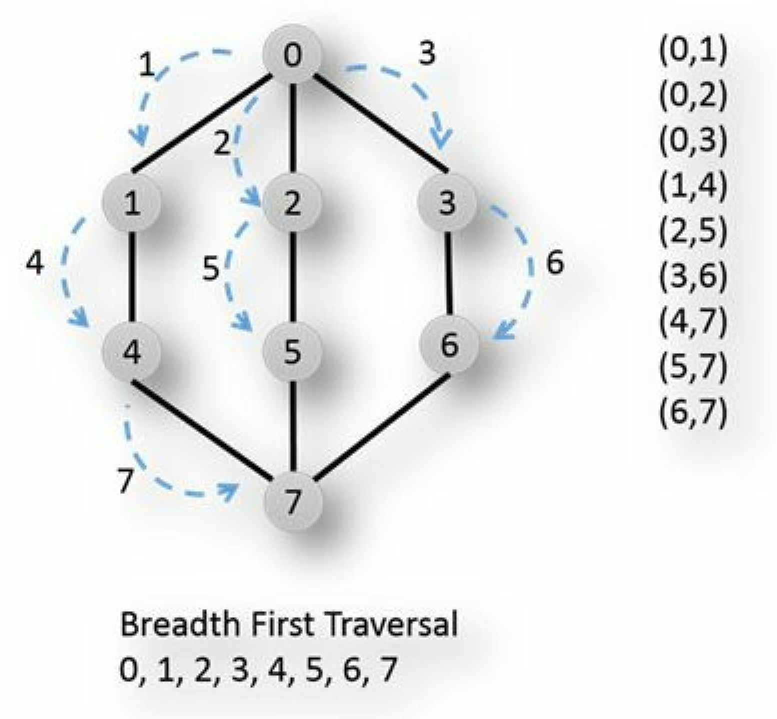
Depth-First Search (DFS)

The DFS algorithm we start from starting point and go into depth of graph until we reach a dead end and then move up to parent node (Backtrack). In DFS, we use stack to get the next vertex to start a search. Alternatively, we can use recursion (system stack) to do the same.

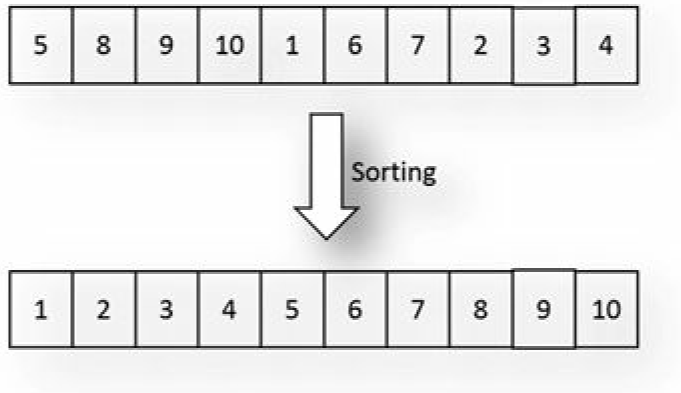


Breadth-First Search (BFS)

In BFS algorithm, a graph is traversed in layer-by-layer fashion. The graph is traversed closer to the starting point. The queue is used to implement BFS.



Sorting Algorithms



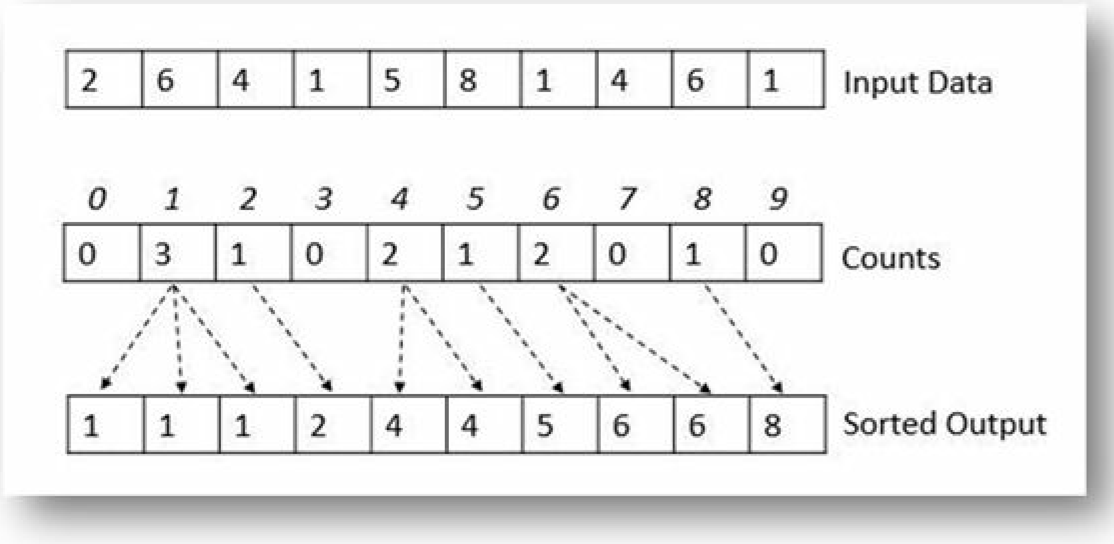
Sorting is the process of placing elements from a collection into ascending or descending order.
Sorting arranges data elements in order so that searching become easier.

There are good sorting functions available which does sorting in **O(nlogn)** time, so in this book when we need sorting we will use sort() function and will assume that the sorting is done in **O(nlogn)** time.

Counting Sort

Counting sort is the simplest and most efficient type of sorting. Counting sort has a strict requirement of a predefined range of data.

Like, sort how many people are in which age group. We know that the age of people can vary between 1 and 130.



If we know the range of input, then sorting can be done using counting in O(n+k). Where n is the number of people and k is the max age possible let us suppose 130.

End note

This chapter has provided a brief introduction of the various data structures, algorithms and their complexities. In the next chapters, we will look into all these data structure in details. If you know the interface of the various data structures, then you can use them while solving other problems without knowing the internal details how they are implemented.

CHAPTER 5: SEARCHING

Introduction

Searching is the process of finding a particular item in a collection of items. The item may be a keyword in a file, a record in a database, a node in a tree or a value in a list etc.

Why Searching?

Imagine you are in a library with millions of books. You want to get a specific book with specific title. How will you find? You will search the book in the section of library, which contains the books whose name starts with the initial letter of the desired book. Then you continue matching with a whole book title until you find your book. (By doing this small heuristic you have reduced the search space by a factor of 26, consider we have an equal number of books whose title begin with particular char.)

Similarly, computer stores lots of information and to retrieve this information efficiently, we need very efficient searching algorithms. To make searching efficient, we keep the data in some proper order. There are certain ways of organizing the data. If you keep the data in proper order, it is easy to search required element. For example, Sorting is one of the process for making data organized.

Different Searching Algorithms

- Linear Search – Unsorted Input
- Linear Search – Sorted Input
- Binary Search (Sorted Input)
- String Search: Tries, Suffix Trees, Ternary Search.
- Hashing and Symbol Tables

Linear Search – Unsorted Input

When elements of a list are not ordered or sorted and we want to search for a particular value, we need to scan the full list unless we find the desired value. This kind of algorithm known as unordered linear search. The major problem with this algorithm is less performance or high Time Complexity in worst case.

Example 5.1

```
func linearSearchUnsorted(data []int, value int) bool {
	size := len(data)
	for i := 0; i < size; i++ {
		if value == data[i] {
			return true
		}
	}
	return false
}
```

Time Complexity: O(n). As we need to traverse the complete list in worst case. Worst case is when your desired element is at the last position of the list. Here, ‘n’ is the size of the list.
Space Complexity: O(1). No extra memory is used to allocate the list.

Linear Search – Sorted

If elements of the list are sorted either in increasing order or in decreasing order, searching for a desired element will be much more efficient than unordered linear search. In many cases, we do not need to traverse the complete list. Following example explains when you encounter a greater value element from the increasing sorted list, you stop searching further. This is how this algorithm saves the time and improves the performance.

Example 5.2

```
func linearSearchSorted(data []int, value int) bool {
	size := len(data)
	for i := 0; i < size; i++ {
		if value == data[i] {
			return true
		} else if value < data[i] {
			return false
		}
	}
	return false
}
```

Time Complexity: **O(n)**. As we need to traverse the complete list in worst case. Worst case is when your desired element is at the last position of the sorted list. However, in the average case this algorithm is more efficient even though the growth rate is same as unsorted.
Space Complexity: **O(1)**. No extra memory is used to allocate the list.

Binary Search

How do we search a word in a dictionary? In general, we go to some approximate page (mostly middle) and start searching from that point. If we see the word that we are searching is same then we are done with the search. Else, if we see the page is before the selected pages, then apply the same procedure for the first half otherwise to the second half. Binary Search also works in the same way. We get to the middle point from the sorted list and start comparing with the desired value.

Note: Binary search requires the list to be sorted otherwise binary search cannot be applied.

Example 5.3

```
//  Binary Search Algorithm - Iterative Way
func Binarysearch(data []int, value int) bool {
	size := len(data)
	low := 0
	high := size - 1
	mid := 0

	for low <= high {
		mid = low + (high-low)/2
		if data[mid] == value {
			return true
		} else if data[mid] < value {
			low = mid + 1
		} else {
			high = mid - 1
		}
	}
	return false
}
```

Time Complexity: O(logn). We always take half input and throwing out the other half. So the recurrence relation for binary search is $T(n) = T(n/2) + c$. Using master theorem (divide and conquer), we get $T(n) = O(logn)$
Space Complexity: O(1)

Example 5.4: Binary search implementation using recursion.

```
// Binary Search Algorithm - Recursive Way
func BinarySearchRecursive(data []int, low int, high int, value int) bool {
	if low > high {
		return false
	}
	mid := low + (high-low)/2
	if data[mid] == value {
		return true
	} else if data[mid] < value {
		return BinarySearchRecursive(data, mid+1, high, value)
	} else {
		return BinarySearchRecursive(data, low, mid-1, value)
	}
}
```

Time Complexity: O(logn). Space Complexity: O(logn) for system stack in recursion

String Searching Algorithms

Refer String chapter.

Hashing and Symbol Tables

Refer Hash-Table chapter.

How sorting is useful in Selection Algorithm?

Selection problems can be converted to sorting problems. Once the list is sorted, it is easy to find the minimum / maximum (or desired element) from the sorted list. The method ‘Sorting and then Selecting’ is inefficient for selecting a single element, but it is efficient when many selections need to be made from the list. It is because only one initial expensive sort is needed, followed by many cheap selection operations.

For example, if we want to get the maximum element from a list. After sorting the list, we can simply return the last element from the list. What if we want to get second maximum. Now, we do not have to sort the list again and we can return the second last element from the sorted list. Similarly, we can return the kth maximum element by just one scan of the sorted list.
So with the above discussion, sorting is used to improve the performance. In general this method requires **O(nlogn)** (for sorting) time. With the initial sorting, we can answer any query in one scan, **O(n)**.

Problems in Searching

Print Duplicates in List

Given a list of n numbers, print the duplicate elements in the list.

First approach: Exhaustive search or Brute force, for each element in list find if there is some other element with the same value. This is done using two for loop, first loop to select the element and second loop to find its duplicate entry.

Example 5.5

```
func printRepeating(data []int) {
	size := len(data)
	fmt.Print("Repeating elements are : ")
	for i := 0; i < size; i++ {
		for j := i + 1; j < size; j++ {
			if data[i] == data[j] {
				fmt.Print(" ", data[i])
			}
		}
	}
	fmt.Println()
}
```

The Time Complexity is $O(n^2)$ and Space Complexity is O(1)

Second approach: Sorting, Sort all the elements in the list and after this in a single scan, we can find the duplicates.

Example 5.6

```
func printRepeating2(data []int) {
	size := len(data)
	sort.Ints(data) // Sort(data,size)
	fmt.Print("Repeating elements are : ")

	for i := 1; i < size; i++ {
		if data[i] == data[i-1] {
			fmt.Print(" ", data[i])
		}
	}
	fmt.Println()
}
```

Sorting algorithms take O(n log n) time and single scan take O(n) time.
The Time Complexity of an algorithm is O(n log n) and Space Complexity is O(1)

Third approach: Hash-Table, using Hash-Table, we can keep track of the elements we have already seen and we can find the duplicates in just one scan.

Example 5.7

```
func printRepeating3(data []int) {
	s := make(Set)
	size := len(data)
	fmt.Print("Repeating elements are : ")
	for i := 0; i < size; i++ {
```

```
        if s.Find(data[i]) {
            fmt.Print(" ", data[i])
        } else {
            s.Add(data[i])
        }
    }
    fmt.Println()
}
```

Hash-Table insert and find take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity is also O(n)

Forth approach: Counting, this approach is only possible if we know the range of the input. If we know that, the elements in the list are in the range 0 to n-1. We can reserve a list of length n and when we see an element, we can increase its count. In just one single scan, we know the duplicates. If we know the range of the elements, then this is the fastest way to find the duplicates.

Example 5.8

```
func printRepeating4(data []int, intrange int) {
    size := len(data)
    count := make([]int, intrange)
    for i := 0; i < size; i++ {
        count[i] = 0
    }
    fmt.Println("Repeating elements are : ")
    for i := 0; i < size; i++ {
        if count[data[i]] == 1 {
            fmt.Println(" ", data[i])
        } else {
            count[data[i]]++
        }
    }
    fmt.Println()
}
```

Counting approach just uses a list so insert and find take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity for creating count list is also O(n)

Find max, appearing element in a list

Given a list of n numbers, find the element, which appears maximum number of times.

First approach: Exhaustive search or Brute force, for each element in array find how many times this particular value appears in array. Keep track of the maxCount and when some element count is greater than maxCount then update the maxCount. This is done using two for loop, first loop to select the element and second loop to count the occurrence of that element.

The Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Example 5.9

```
func getMax(data []int) int {
    size := len(data)
    max := data[0]
```

```
    count := 1
    maxCount := 1
    for i := 0; i < size; i++ {
        count = 1
        for j := i + 1; j < size; j++ {
            if data[i] == data[j] {
                count++
            }
        }
        if count > maxCount {
            max = data[i]
            maxCount = count
        }
    }
    return max
}
```

Second approach: Sorting, Sort all the elements in the list and after this in a single scan, we can find the counts. Sorting algorithms take O(n.logn) time and single scan take O(n) time. The Time Complexity of an algorithm is O(n.logn) and Space Complexity is O(1)

Example 5.10

```
func getMax2(data []int) int {
    size := len(data)
    max := data[0]
    maxCount := 1
    curr := data[0]
    currCount := 1
    sort.Ints(data) // Sort(data,size)
    for i := 1; i < size; i++ {
        if data[i] == data[i-1] {
            currCount++
        } else {
            currCount = 1
            curr = data[i]
        }
        if currCount > maxCount {
            maxCount = currCount
            max = curr
        }
    }
    return max
}
```

Third approach: Counting, This approach is only possible if we know the range of the input. If we know that, the elements in the list are in the range 0 to n-1. We can reserve a list of length n and when we see an element, we can increase its count. In just one single scan, we know the duplicates. If we know the range of the elements, then this is the fastest way to find the max count.

Counting approach just uses list so to increase count take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity for creating count list is also O(n)

Example 5.11

```
func getMax3(data []int, dataRange int) int {
	max := data[0]
	maxCount := 1
	size := len(data)
	count := make([]int, dataRange)
	for i := 0; i < size; i++ {
		count[data[i]]++
		if count[data[i]] > maxCount {
			maxCount = count[data[i]]
			max = data[i]
		}
	}
	return max
}
```

Majority element in a list

Given a list of n elements. Find the majority element, which appears more than n/2 times. Return 0 in case there is no majority element.

First approach: Exhaustive search or Brute force, for each element in array find how many times this particular value appears in array. Keep track of the maxCount and when some element count is greater than maxCount then update the maxCount. This is done using two for loop, first loop to select the element and second loop to count the occurrence of that element.

Once we have the final, maxCount we can see if it is greater than n/2, if it is greater than we have a majority if not we do not have any majority.

The Time Complexity is $O(n^2) + O(1) = O(n^2)$ and Space Complexity is $O(1)$

Example 5.12

```
func getMajority(data []int) (int, bool) {
	size := len(data)
	max := 0
	count := 0
	maxCount := 0
	for i := 0; i < size; i++ {
		for j := i + 1; j < size; j++ {
			if data[i] == data[j] {
				count++
			}
		}
		if count > maxCount {
			max = data[i]
			maxCount = count
		}
	}
	if maxCount > size/2 {
		return max, true
	}
	fmt.Println("MajorityDoesNotExist")
	return 0, false
```

```
}
```

Second approach: Sorting, Sort all the elements in the list. If there is a majority then the middle element at the index n/2 must be the majority number. So just single scan can be used to find its count and see if the majority is there or not.

Sorting algorithms take O(n.logn) time and single scan take O(n) time.
The Time Complexity of an algorithm is O(n.logn) and Space Complexity is O(1)

Example 5.13
```
func getMajority2(data []int) (int, bool) {
	size := len(data)
	majIndex := size / 2
	sort.Ints(data) // Sort(data,size)
	candidate := data[majIndex]
	count := 0
	for i := 0; i < size; i++ {
		if data[i] == candidate {
			count++
		}
	}
	if count > size/2 {
		return data[majIndex], true
	}
	fmt.Println("MajorityDoesNotExist")
	return 0, false
}
```

Third approach: This is a cancelation approach (Moore's Voting Algorithm), if all the elements stand against the majority and each element is cancelled with one element of majority if there is majority then majority prevails.

- Set the first element of the list as majority candidate and initialize the count to be 1.
- Start scanning the list.
  - If we get some element whose value same as a majority candidate, then we increase the count.
  - If we get an element whose value is different from the majority candidate, then we decrement the count.
  - If count become 0, that means we have a new majority candidate. Make the current candidate as majority candidate and reset count to 1.
  - At the end, we will have the only probable majority candidate.
- Now scan through the list once again to see if that candidate we found above have appeared more than n/2 times.

Counting approach just scans through list two times. The Time Complexity of the algorithm is O(n) time. Space Complexity for creating count list is also O(1)

Example 5.14
```
func getMajority3(data []int) (int, bool) {
	majIndex := 0
	count := 1
	size := len(data)

	for i := 1; i < size; i++ {
		if data[majIndex] == data[i] {
			count++
		} else {
			count--
		}
```

```
        if count == 0 {
            majIndex = i
            count = 1
        }
    }
    candidate := data[majIndex]
    count = 0
    for i := 0; i < size; i++ {
        if data[i] == candidate {
            count++
        }
    }
    if count > size/2 {
        return data[majIndex], true
    }
    fmt.Println("MajorityDoesNotExist")
    return 0, false
}
```

Find the missing number in a list

Given a list of n-1 elements, which are in the range of 1 to n. There are no duplicates in the list. One of the integer is missing. Find the missing element.

First approach: Exhaustive search or Brute force, for each value in the range 1 to n, find if there is some element in list which have the same value. This is done using two for loop, first loop to select value in the range 1 to n and the second loop to find if this element is in the list or not.

The Time Complexity is $O(n^2)$ and Space Complexity is O(1)

Example 5.15

```
func findMissingNumber(data []int) (int, bool) {
    var found int
    size := len(data)
    for i := 1; i <= size; i++ {
        found = 0
        for j := 0; j < size; j++ {
            if data[j] == i {
                found = 1
                break
            }
        }
        if found == 0 {
            return i, true
        }
    }
    fmt.Println("NoNumberMissing")
    return 0, false
}
```

Second approach: Sorting, Sort all the elements in the list and after this in a single scan, we can find the duplicates.

Sorting algorithms take O(n.logn) time and single scan take O(n) time.
The Time Complexity of an algorithm is O(n.logn) and Space Complexity is O(1)

Third approach: Hash-Table, using Hash-Table, we can keep track of the elements we have already seen and we can find the missing element in just one scan.

Hash-Table insert and find take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity is also O(n)

Forth approach: Counting, we know the range of the input so counting will work. As we know that, the elements in the list are in the range 0 to n-1. We can reserve a list of length n and when we see an element, we can increase its count. In just one single scan, we know the missing element.

Counting approach just uses a list so insert and find take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity for creating count list is also O(n)

Fifth approach: You are allowed to modify the given input list. Modify the given input list in such a way that in the next scan you can find the missing element.

When you scan through the list. When at index "index", the value stored in the list will be arr[index] so add the number "n + 1" to arr[arr[index]]. Always read the value from the list using a reminder operator "%". When you scan the list for the first time and modified all the values, then one single scan you can see if there is some value in the list which is smaller than "n+1" that index is the missing number.
In this approach, the list is scanned two times and the Time Complexity of this algorithm is O(n). Space Complexity is O(1)

Sixth approach: Summation formula to find the sum of n numbers from 1 to n. Subtract the values stored in the list and you will have your missing number.
The Time Complexity of this algorithm is O(n). Space Complexity is O(1)

Seventh approach: XOR approach to find the sum of n numbers from 1 to n. XOR the values stored in the list and you will have your missing number.
The Time Complexity of this algorithm is O(n). Space Complexity is O(1)

Example 5.16

```
func findMissingNumber2(data []int, dataRange int) int {
    size := len(data)
    xorSum := 0
    // get the XOR of all the numbers from 1 to dataRange
    for i := 1; i <= dataRange; i++ {
        xorSum ^= i
    }
    // loop through the array and get the XOR of elements
    for i := 0; i < size; i++ {
        xorSum ^= data[i]
    }
    return xorSum
}
```

Note: Same problem can be asked in many forms (sometimes you have to do the xor of the range sometime you do not):

1. There are numbers in the range of 1-n out of which all appears single time but one that appear two times.

2. All the elements in the range 1-n are appearing 16 times and one element appear 17 times. Find the element that appears 17 times.

Find Pair in a list

Given a list of n numbers, find two elements such that their sum is equal to “value”

First approach: Exhaustive search or Brute force, for each element in list find if there is some other element, which sum up to the desired value. This is done using two for loop, first loop to select the element and second loop to find another element.
The Time Complexity is $O(n^2)$ and Space Complexity is O(1)

Example 5.17

```
func FindPair(data []int, value int) bool {
	size := len(data)
	ret := false
	for i := 0; i < size; i++ {
		for j := i + 1; j < size; j++ {
			if (data[i] + data[j]) == value {
				fmt.Println("The pair is : ", data[i], ",", data[j])
				ret = true
			}
		}
	}
	return ret
}
```

Second approach: Sorting, Steps are as follows:

1. Sort all the elements in the list.
2. Take two variable first and second. Variable first= 0 and second = size -1
3. Compute sum = arr[first]+arr[second]
4. If the sum is equal to the desired value, then we have the solution
5. If the sum is less than the desired value, then we will increase first
6. If the sum is greater than the desired value, then we will decrease the second
7. We repeat the above process until we get the desired pair or we get first >= second

Sorting algorithms take O(n.logn) time and single scan take O(n) time.
The Time Complexity of an algorithm is O(n.logn) and Space Complexity is O(1)

Example 5.18

```
func FindPair2(data []int, value int) bool {
	size := len(data)
	first := 0
	second := size - 1
	ret := false
	sort.Ints(data) // Sort(data, size)
	for first < second {
		curr := data[first] + data[second]
		if curr == value {
			fmt.Println("The pair is ", data[first], ",", data[second])
			ret = true
		}
```

```
        if curr < value {
            first++
        } else {
            second--
        }
    }
    return ret
}
```

Third approach: Hash-Table, using Hash-Table, we can keep track of the elements we have already seen and we can find the pair in just one scan.

1. For each element, insert the value in Hashtable. Let say current value is arr[index]
2. If value - arr[index] is in the Hashtable then we have the desired pair.
3. Else, proceed to the next entry in the list.

Hash-Table insert and find take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity is also O(n)

Example 5.19

```
func FindPair3(data []int, value int) bool {
    s := make(Set)
    size := len(data)
    ret := false
    for i := 0; i < size; i++ {
        if s.Find(value - data[i]) {
            fmt.Println(i, "The pair is:", data[i],",",(value - data[i]))
            ret = true
        } else {
            s.Add(data[i])
        }
    }
    return ret
}
```

Forth approach: Counting, This approach is only possible if we know the range of the input. If we know that, the elements in the list are in the range 0 to n-1. We can reserve a list of length n and when we see an element, we can increase its count. In place of the Hashtable in the above approach, we will use this list and will find out the pair.

Counting approach just uses a list so insert and find take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity for creating count list is also O(n)

Find the Pair in two Lists

Given two list X and Y. Find a pair of elements (x_i, yi) such that $x_i \in X$ and $y_i \in Y$ where $x_i + y_i = value$.

First approach: Exhaustive search or Brute force, loop through element xi of X and see if you can find (value – xi) in Y. Two for loop.
The Time Complexity is $O(n^2)$ and Space Complexity is O(1)

Second approach: Sorting, Sort all the elements in the second list Y. For each element if X you can see if that element is there in Y by using binary search.

Sorting algorithms take O(m.logm) and searching will take O(n.logm) time.
The Time Complexity of an algorithm is O(n.logm) or O(m.logm) and Space Complexity is O(1)

Third approach: Sorting, Steps are as follows:

1. Sort the elements of both X and Y in increasing order.
2. Take the sum of the smallest element of X and the largest element of Y.
3. If the sum is equal to value, we got our pair.
4. If the sum is smaller than value, take next element of X
5. If the sum is greater than value, take the previous element of Y

Sorting algorithms take O(n.logn) + O(m.logm) for sorting and searching will take O(n+m) time.
The Time Complexity of an algorithm is O(n.logn) Space Complexity is O(1)

Forth approach: Hash-Table, Steps are as follows:

1. Scan through all the elements in the list Y and insert them into Hashtable.
2. Now scan through all the elements of list X, let us suppose the current element is xi see if you can find (value - xi) in the Hashtable.
3. If you find the value, you got your pair.
4. If not, then go to the next value in the list X.

Hash-Table insert and find take constant time O(1) so the total Time Complexity of the algorithm is O(n) O(n)time. Space Complexity is also O(n)

Fifth approach: Counting, This approach is only possible if we know the range of the input. Same as Hashtable implementation just use a simple list in place of Hashtable and you are done.

Counting approach just uses a list so insert and find take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity for creating count list is also O(n)

Two elements whose sum is closest to zero

Given a List of integers, both +ve and -ve. You need to find the two elements such that their sum is closest to zero.

First approach: Exhaustive search or Brute force, for each element in list find the other element whose value when added will give minimum absolute value. This is done using two for loop, first loop to select the element and second loop to find the element that should be added to it so that the absolute of the sum will be minimum or close to zero.
The Time Complexity is $O(n^2)$ and Space Complexity is O(1)

Example 5.20

```
func minAbsSumPair(data []int) {
    var sum int
    size := len(data)
    // Array should have at least two elements
    if size < 2 {
        fmt.Println("InvalidInput")
    }
    // Initialization of values
    minFirst := 0
    minSecond := 1
    minSum := abs(data[0] + data[1])
    for l := 0; l < size-1; l++ {
        for r := l + 1; r < size; r++ {
```

```
            sum = abs(data[l] + data[r])
            if sum < minSum {
                minSum = sum
                minFirst = l
                minSecond = r
            }
        }
    }
    fmt.Println(" The two elements with minimum sum are : ", data[minFirst], " , ", data[minSecond])
}
```

Second approach: Sorting

Steps are as follows:

1. Sort all the elements in the list.
2. Take two variable firstIndex = 0 and secondIndex = size -1
3. Compute sum = arr[firstIndex]+arr[secondIndex]
4. If the sum is equal to the 0 then we have the solution
5. If the sum is less than the 0 then we will increase first
6. If the sum is greater than the 0 then we will decrease the second
7. We repeat the above process 3 to 6, until we get the desired pair or we get first >= second

Example 5.21

```
func minAbsSumPair2(data []int) {
    var sum int
    size := len(data)
    // Array should have at least two elements
    if size < 2 {
        fmt.Println("InvalidInput")
    }
    sort.Ints(data) // Sort(data, size)

    // Initialization of values
    minFirst := 0
    minSecond := size - 1
    minSum := abs(data[minFirst] + data[minSecond])
    for l, r := 0, (size - 1); l < r; {
        sum = (data[l] + data[r])
        if abs(sum) < minSum {
            minSum = abs(sum) /// just corrected......hemant
            minFirst = l
            minSecond = r
        }

        if sum < 0 {
            l++
        } else if sum > 0 {
            r--
        } else {
            break
        }
    }
    fmt.Println(" The two elements with minimum sum are : ", data[minFirst], " , ", data[minSecond])
```

```
}
```

Find maxima in a bitonic list

A bitonic list comprises of an increasing sequence of integers immediately followed by a decreasing sequence of integers.

Since the elements are sorted in some order, we should go for algorithm similar to binary search. The steps are as follows:

1. Take two variable for storing start and end index. Variable start=0 and end=size-1
2. Find the middle element of the list.
3. See if the middle element is the maxima. If yes, return the middle element.
4. Alternatively, If the middle element in increasing part, then we need to look for in mid+1 and end.
5. Alternatively, if the middle element is in the decreasing part, then we need to look in the start and mid-1.
6. Repeat step 2 to 5 until we get the maxima.

Example 5.22

```
func SearchBitonicArrayMax(data []int) (int, bool) {
	size := len(data)
	start := 0
	end := size - 1
	mid := (start + end) / 2
	maximaFound := 0
	if size < 3 {
		fmt.Println("InvalidInput")
		return 0, false
	}
	for start <= end {
		mid := (start + end) / 2

		if data[mid-1] < data[mid] && data[mid+1] < data[mid] { //maxima
			maximaFound = 1
			break
		} else if data[mid-1] < data[mid] && data[mid] < data[mid+1] { // increasing
			start = mid + 1
		} else if data[mid-1] > data[mid] && data[mid] > data[mid+1] { // decreasing
			end = mid - 1
		} else {
			break
		}
	}
	if maximaFound == 0 {
		fmt.Println("NoMaximaFound")
		return 0, false
	}
	return mid, true
}
```

Search element in a bitonic list

A bitonic list comprises of an increasing sequence of integers immediately followed by a decreasing sequence of integers. To search an element in a bitonic list:

1. Find the index or maximum element in the list. By finding the end of increasing part of the list, using modified binary search.
2. Once we have the maximum element, search the given value in increasing part of the list using binary search.
3. If the value is not found in increasing part, search the same value in decreasing part of the list using binary search.

Example 5.23

```
func SearchBitonicArray(data []int, key int) int {
	size := len(data)
	maxIndex, _ := FindMaxBitonicArray(data)
	k := BinarySearch(data, 0, maxIndex, key, true)
	if k != -1 {
		return k
	} else {
		return BinarySearch(data, maxIndex+1, size-1, key, false)
	}
}

func FindMaxBitonicArray(data []int) (int, bool) {
	size := len(data)
	start := 0
	end := size - 1
	mid := 0
	if size < 3 {
		fmt.Println("InvalidInput")
		return -1, false
	}
	for start <= end {
		mid = (start + end) / 2
		if data[mid-1] < data[mid] && data[mid+1] < data[mid] /* maxima */ {
			return mid, true
		} else if data[mid-1] < data[mid] && data[mid] < data[mid+1] /* increasing */ {
			start = mid + 1
		} else if data[mid-1] > data[mid] && data[mid] > data[mid+1] /* decreasing */ {
			end = mid - 1
		} else {
			break
		}
	}
	fmt.Println("error")
	return -1, false
}

func BinarySearch(data []int, start int, end int, key int, isInc bool) int {
	if end < start {
		return -1
	}
	mid := (start + end) / 2
	if key == data[mid] {
		return mid
	}
	if isInc != false && key < data[mid] || isInc == false && key > data[mid] {
		return BinarySearch(data, start, mid-1, key, isInc)
	} else {
```

```
        return BinarySearch(data, mid+1, end, key, isInc)
    }
}
```

Occurrence counts in sorted List

Given a sorted list arr[] find the number of occurrences of a number.

First approach: Brute force, Traverse the list and in linear time we will get the occurrence count of the number. This is done using one loop.
The Time Complexity is O(n) and Space Complexity is O(1)

Example 5.24
```
func findKeyCount(data []int, key int) int {
    count := 0
    size := len(data)
    for i := 0; i < size; i++ {
        if data[i] == key {
            count++
        }
    }
    return count
}
```

Second approach: Since we have sorted list, we should think about some binary search.
1. First, we should find the first occurrence of the key.
2. Then we should find the last occurrence of the key.
3. Take the difference of these two values and you will have the solution.

Example 5.25
```
func findKeyCount2(data []int, key int) int {
    size := len(data)
    firstIndex := findFirstIndex(data, 0, size-1, key)
    lastIndex := findLastIndex(data, 0, size-1, key)
    return (lastIndex - firstIndex + 1)
}

func findFirstIndex(data []int, start int, end int, key int) int {
    if end < start {
        return -1
    }
    mid := (start + end) / 2
    if key == data[mid] && (mid == start || data[mid-1] != key) {
        return mid
    }

    if key <= data[mid] {
        return findFirstIndex(data, start, mid-1, key)
    }
    return findFirstIndex(data, mid+1, end, key)
}
```

```
func findLastIndex(data []int, start int, end int, key int) int {
	if end < start {
		return -1
	}

	mid := (start + end) / 2
	if key == data[mid] && (mid == end || data[mid+1] != key) {
		return mid
	}

	if key < data[mid] {
		return findLastIndex(data, start, mid-1, key)
	}
	return findLastIndex(data, mid+1, end, key)
}
```

Separate even and odd numbers in List

Given a list of even and odd numbers, write a program to separate even numbers from the odd numbers.

-
First approach: allocate a separate list, then scan through the given list, and fill even numbers from the start and odd numbers from the end.

Second approach: Algorithm is as follows.

1. Initialize the two variable left and right. Variable left=0 and right= size-1.
2. Keep increasing the left index until the element at that index is even.
3. Keep decreasing the right index until the element at that index is odd.
4. Swap the number at left and right index.
5. Repeat steps 2 to 4 until left is less than right.

Example 5.26

```
func seperateEvenAndOdd(data []int) {
	size := len(data)
	left := 0
	right := size - 1
	for left < right {
		if data[left]%2 == 0 {
			left++
		} else if data[right]%2 == 1 {
			right--
		} else {
			data[left], data[right] = data[right], data[left] // swap
			left++
			right--
		}
	}
}
```

Stock purchase-sell problem

Given a list, in which nth element is the price of the stock on nth day. You are asked to buy once and sell once, on what date you will be buying and at what date you will be selling to get maximum profit.

Or

Given a list of numbers, you need to maximize the difference between two numbers, such that you can subtract the number, which appear before form the number that appear after it.

First approach: Brute force, for each element in list find if there is some other element whose difference is maximum. This is done using two for loop, first loop to select, buy date index and the second loop to find its selling date entry.
The Time Complexity is $O(n^2)$ and Space Complexity is O(1)

Second approach: Another clever solution is to keep track of the smallest value seen so far from the start. At each point, we can find the difference and keep track of the maximum profit. This is a linear solution.
The Time Complexity of the algorithm is O(n) time. Space Complexity for creating count list is also O(1)

Example 5.27
```
func maxProfit(stocks []int) {
	size := len(stocks)
	buy := 0
	sell := 0
	curMin := 0
	currProfit := 0
	maxProfit := 0

	for i := 0; i < size; i++ {
		if stocks[i] < stocks[curMin] {
			curMin = i
		}

		currProfit = stocks[i] - stocks[curMin]
		if currProfit > maxProfit {
			buy = curMin
			sell = i
			maxProfit = currProfit
		}
	}
	fmt.Println("Purchase day is- ", buy, " at price ", stocks[buy])
	fmt.Println("Sell day is- ", sell, " at price ", stocks[sell])
	fmt.Println("Max Profit :: ", maxProfit)
}
```

Find a median of a list

Given a list of numbers of size n, if all the elements of the list are sorted then find the element, which lie at the index n/2.

First approach: Sort the list and return the element in the middle.
Sorting algorithms take O(n.logn).
The Time Complexity of an algorithm is O(n.logn) and Space Complexity is O(1)

Example 5.28
```
func getMedian(data []int) int {
	size := len(data)
	sort.Ints(data)
	return data[size/2]
```

```
}
```

Second approach: Use QuickSelect algorithm. This algorithm we will look into the next chapter. In QuickSort algorithm just skip the recursive call that we do not need.

The average Time Complexity of this algorithm will be O(1)

Find median of two sorted Lists.

First approach: Keep track of the index of both the list, say the index are i and j. keep increasing the index of the list which ever have a smaller value. Use a counter to keep track of the elements that we have already traced.

The Time Complexity of an algorithm is O(n) and Space Complexity is O(1)

Example 5.29

```
func findMedian(dataFirst []int, dataSecond []int) int {
	sizeFirst := len(dataFirst)
	sizeSecond := len(dataSecond)
	// cealing function.
	medianIndex := ((sizeFirst+sizeSecond)+(sizeFirst+sizeSecond)%2)/2
	i := 0
	j := 0
	count := 0
	for count < medianIndex-1 {
		if i < sizeFirst-1 && dataFirst[i] < dataSecond[j] {
			i++
		} else {
			j++
		}
		count++
	}
	if dataFirst[i] < dataSecond[j] {
		return dataFirst[i]
	}
	return dataSecond[j]
}
```

Search 01 List

Given a list of 0’s and 1’s. All the 0’s come before 1’s. Write an algorithm to find the index of the first 1.

Or

You are given a list which contains either 0 or 1, and they are in sorted order Ex. a [] = {1, 1, 1, 1, 0, 0, 0} How will you count no of 1`s and 0's?

First approach: Binary Search, since the list is sorted using binary search to find the desired index.
The Time Complexity of an algorithm is O(logn) and Space Complexity is O(1)

Example 5.30

```
func BinarySearch01(data []int) int {
	size := len(data)
	if size == 1 && data[0] == 1 {
		return -1
```

```
    }
    return BinarySearch01Util(data, 0, size-1)
}
func BinarySearch01Util(data []int, start int, end int) int {
    if end < start {
        return -1
    }
    mid := (start + end) / 2
    if 1 == data[mid] && 0 == data[mid-1] {
        return mid
    }
    if 0 == data[mid] {
        return BinarySearch01Util(data, mid+1, end)
    }
    return BinarySearch01Util(data, start, mid-1)
}
```

Search in sorted rotated List

Given a sorted list s of n integer. s is rotated an unknown number of times. Find an element in the list.

First approach: Since the list is sorted, we can use modified binary search to find the element.
The Time Complexity of an algorithm is O(logn) and Space Complexity is O(1)

Example 5.31

```
func BinarySearchRotateArray(data []int, key int) int {
    size := len(data)
    return BinarySearchRotateArrayUtil(data, 0, size-1, key)
}
```

```
func BinarySearchRotateArrayUtil(data []int, start int, end int, key int) int {
    if end < start {
        return -1
    }
    mid := (start + end) / 2
    if key == data[mid] {
        return mid
    }
    if data[mid] > data[start] {
        if data[start] <= key && key < data[mid] {
            return BinarySearchRotateArrayUtil(data, start, mid-1, key)
        }
        return BinarySearchRotateArrayUtil(data, mid+1, end, key)
    } else {
        if data[mid] < key && key <= data[end] {
            return BinarySearchRotateArrayUtil(data, mid+1, end, key)
        }
        return BinarySearchRotateArrayUtil(data, start, mid-1, key)
    }
}
```

First Repeated element in the list

Given an unsorted list of n elements, find the first element, which is repeated.

First approach: Exhaustive search or Brute force, for each element in list find if there is some other element with the same value. This is done using two for loop, first loop to select the element and second loop to find its duplicate entry. The Time Complexity is $O(n^2)$ and Space Complexity is O(1)

Example 5.32

```
func FirstRepeated(data []int) int {
	size := len(data)
	for i := 0; i < size; i++ {
		for j := i + 1; j < size; j++ {
			if data[i] == data[j] {
				return data[i]
			}
		}
	}
	return 0
}
```

Second approach: Hash-Table, using Hash-Table, we can keep track of the number of times a particular element came in the list. First scan just populate the Hashtable. In the second, scan just look the occurrence of the elements in the Hashtable. If occurrence is more for some element, then we have our solution and the first repeated element. Hash-Table insert and find take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity is also O(n) for maintaining hash.

Transform List

How would you swap elements of a list like [a1 a2 a3 a4 b1 b2 b3 b4] to convert it into [a1 b1 a2 b2 a3 b3 a4 b4]?
Approach:

- First swap elements in the middle pair
- Next swap elements in the middle two pairs
- Next swap elements in the middle three pairs
- Iterate n-1 steps.

Example 5.33

```
func transformArrayAB1(str string) string {
	data := []rune(str)
	size := len(data)
	N := size / 2
	for i := 1; i < N; i++ {
		for j := 0; j < i; j++ {
			data[N-i+2*j], data[N-i+2*j+1] = data[N-i+2*j+1], data[N-i+2*j]
		}
	}
	return string(data)
}
```

Find 2nd largest number in a list with minimum comparisons

Suppose you are given an unsorted list of n distinct elements. How will you identify the second largest element with minimum number of comparisons?

First approach: Find the largest element in the list. Then replace the last element with the largest element. Then search the second largest element int the remaining n-1 elements.
The total number of comparisons is: (n-1) + (n-2)

Second approach: Sort the list and then give the (n-1) element. This approach is still more inefficient.

Third approach: Using priority queue / Heap. This approach we will look into heap chapter. Use buildHeap() function to build heap from the list. This is done in n comparisons. Arr[0] is the largest number, and the greater among arr[1] and arr[2] is the second largest.
The total number of comparisons are: (n-1) + 1 = n

Check if two Lists are permutation of each other

Given two integer Lists. You have to check whether they are permutation of each other.

First approach: Sorting, Sort all the elements of both the Lists and Compare each element of both the Lists from beginning to end. If there is no mismatch, return true. Otherwise, false.
Sorting algorithms take O(n.logn) time and comparison take O(n) time.
The Time Complexity of an algorithm is O(n.logn) and Space Complexity is O(1)

Example 5.34

```
func CheckPermutation(data1 []int, data2 []int) bool {
	size1 := len(data1)
	size2 := len(data2)
	if size1 != size2 {
		return false
	}

	sort.Ints(data1)
	sort.Ints(data2)
	for i := 0; i < size1; i++ {
		if data1[i] != data2[i] {
			return false
		}
	}
	return true
}
```

Second approach: Hash-Table (Assumption: No duplicates).
1. Create a Hash-Table for all the elements of the first list.
2. Traverse the other list from beginning to the end and search for each element in the Hash-Table.
3. If all the elements are found in, the Hash-Table return true, otherwise return false.

Hash-Table insert and find take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity is also O(n)
Time Complexity = O(n) (For creation of Hash-Table and look-up),
Space Complexity = O(n) (For creation of Hash-Table).

Example 5.35

```
func checkPermutation2(data1 []int, data2 []int) bool {
	size1 := len(data1)
```

```
    size2 := len(data2)
    h := make(map[int]int)

    if size1 != size2 {
        return false
    }

    for i := 0; i < size1; i++ {
        h[data1[i]]++
        h[data2[i]]--
    }

    for i := 0; i < size1; i++ {
        if h[data1[i]] != 0 {
            return false
        }
    }
    return true
}
```

Remove duplicates in an integer list

First approach: Sorting, Steps are as follows:

1. Sort the list.
2. Take two references. A subarray will be created with all unique elements starting from 0 to the first reference (The first reference points to the last index of the subarray). The second reference iterates through the list from 1 to the end. Unique numbers will be copied from the second reference location to first reference location and the same elements are ignored.

Time Complexity calculation:
Time to sort the list = **O(nlogn)**. Time to remove duplicates = **O(n).**
Overall Time Complexity = **O(nlogn)**.
No additional space is required so Space Complexity is **O(1)**.

Example 5.36

```
func removeDuplicates(data []int) int {
    j := 0
    size := len(data)
    if size == 0 {
        return 0
    }

    sort.Ints(data)
    for i := 1; i < size; i++ {
        if data[i] != data[j] {
            j++
            data[j] = data[i]
        }
    }
    return j + 1
}
```

Searching for an element in a 2-d sorted list

Given a 2 dimensional list. Each row and column are sorted in ascending order. How would you find an element in it?

The algorithm works as:
1. Start with element at last column and first row
2. If the element is the value we are looking for, return true.
3. If the element is greater than the value we are looking for, go to the element at previous column but same row.
4. If the element is less than the value we are looking for, go to the element at next row but same column.
5. Return false, if the element is not found after reaching the element of the last row of the first column. Condition (row < r && column >= 0) is false.

Example 5.37

```
func FindElementIn2DArray(data [][]int, r int, c int, value int) bool {
	row := 0
	column := c - 1
	for row < r && column >= 0 {
		if data[row][column] == value {
			return true
		} else if data[row][column] > value {
			column--
		} else {
			row++
		}
	}
	return false
}
```

Running time = **O(N)**.

Exercise

1. Given a list of n elements, find the first repeated element.

2. Given a list of n elements, write an algorithm to find three elements in a list whose sum is a given value.
   Hint: Try to do this problem using a brute force approach. Then try to apply the sorting approach along with a brute force approach. The Time Complexity will be $O(n^2)$

3. Given a list of –ve and +ve numbers, write a program to separate –ve numbers from the +ve numbers.

4. Given a list of 1's and 0's, write a program to separate 0's from 1's.
   Hint: QuickSelect, counting

5. Given a list of 0's, 1's and 2's, write a program to separate 0's , 1's and 2's.

6. Given a list whose elements is monotonically increasing with both negative and positive numbers. Write an algorithm to find the point at which list becomes positive.

7. Given a sorted list, find a given number. If found return the index if not, find the index of that number if it is inserted into the list.

8. Find max in sorted rotated list.

9. Find min in the sorted rotated list.

10. Find kth Smallest Element in the Union of Two Sorted Lists

CHAPTER 6: SORTING

Introduction

Sorting is the process of placing elements from a list into ascending or descending order. For example, when we play cards, sort cards, according to their value so that we can find the required card easily.

When we go to some library, the books are arranged according to streams (Algorithm, Operating systems, Networking etc.). Sorting arranges data elements in order so that searching become easier. When books are arranged in proper indexing order, then it is easy to find a book we are looking for.

This chapter discusses algorithms for sorting a list of N items. Understanding sorting algorithms are the first step towards understanding algorithm analysis. Many sorting algorithms are developed and analyzed.

A sorting algorithm like Bubble-Sort, Insertion-Sort and Selection-Sort are easy to implement and are suitable for the small input set. However, for large dataset they are slow.

A sorting algorithm like Merge-Sort, Quick-Sort and Heap-Sort are some of the algorithms that are suitable for sorting large dataset. However, they are overkill if we want to sort the small dataset.

Some algorithm, which is suitable when we have some range information on input data.

Some other algorithm is there to sort a huge data set that cannot be stored in memory completely, for which external sorting technique is developed.

Before we start a discussion of the various algorithms one by one. First, we should look at comparison function that is used to compare two values.

Less function will return 1 if value1 is less than value2 otherwise it will return 0.

```
func less(value1 int, value2 int) bool {
    return value1 < value2
}
```

More function will return 1 if value1 is more than value2 otherwise it will return 0.

```
func more(value1 int, value2 int) bool {
    return value1 > value2
}
```

The value in various sorting algorithms is compared using one of the above functions and it will be swapped depending upon the return value of these functions. If more() comparison function is used, then sorted output will be increasing in order and if less() is used then resulting output will be in descending order.

Type of Sorting

Internal Sorting: All the elements can be read into memory at the same time and sorting is performed in memory.

1. Selection-Sort
2. Insertion-Sort
3. Bubble-Sort
4. Quick-Sort
5. Merge-Sort

External Sorting: In this, the dataset is so big that it is impossible to load the whole dataset into memory so sorting is done in chunks.

1. Merge-Sort

Three things to consider in choosing, sorting algorithms for application:

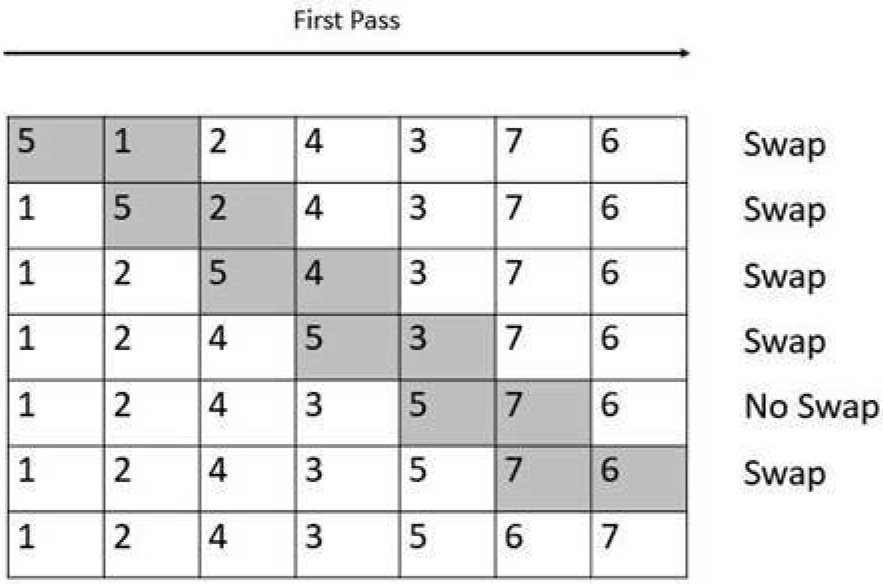
1. Number of elements in list
2. A number of different orders of list required
3. The amount of time required to move the data or not move the data

Bubble-Sort

Bubble-Sort is the slowest algorithm for sorting. It is easy to implement and used when data is small.

In Bubble-Sort, we compare each pair of adjacent values. We want to sort values in increasing order so if the second value is less than the first value then we swap these two values. Otherwise, we will go to the next pair. Thus, largest values bubble to the end of the list.

After the first pass, the largest value will be in the rightmost position. We will have N number of passes to get the list completely sorted.



Example 6.1

```
func BubbleSort(arr []int, comp func(int, int) bool) {
    size := len(arr)
    for i := 0; i < (size - 1); i++ {
        for j := 0; j < size-i-1; j++ {
            if comp(arr[j], arr[j+1]) {
                /* Swapping */
                arr[j+1], arr[j] = arr[j], arr[j+1]
            }
        }
    }
}

func main() {
    data := []int{9, 1, 8, 2, 7, 3, 6, 4, 5}
    BubbleSort(data, more)
    fmt.Println(data)
}
```

Analysis:

- The outer for loops represents the number of swaps that are done for comparison of data.
- The inner loop is actually used to do the comparison of data. At the end of each inner loop iteration, the largest value is moved to the end of the list. In the first iteration the largest value, in the second iteration the second largest and so on.
- more() function is used for comparison which means when the value of the first argument is greater than the value

of the second argument then perform a swap. By this we are sorting in increasing order if we have, the less() function in place of more() then list will be sorted in decreasing order.

Complexity Analysis:
Each time the inner loop execute for (n-1), (n-2), (n-3)…(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = n(n-1)/2

Worst case performance	$O(n^2)$
Average case performance	$O(n^2)$
Space Complexity	O(1) as we need only one temp variable
Stable Sorting	Yes

Modified (improved) Bubble-Sort

When there is no more swap in one pass of the outer loop. It indicates that all the elements are already in order so we should stop sorting. This sorting improvement in Bubble-Sort is extremely useful when we know that, except few elements rest of the list is already sorted.

Example 6.2

```
func BubbleSort2(arr []int, comp func(int, int) bool) {
	size := len(arr)
	swapped := 1
	for i := 0; i < (size-1) && swapped == 1; i++ {
		swapped = 0
		for j := 0; j < size-i-1; j++ {
			if comp(arr[j], arr[j+1]) {
				arr[j+1], arr[j] = arr[j], arr[j+1]
				swapped = 1
			}
		}
	}
}
```

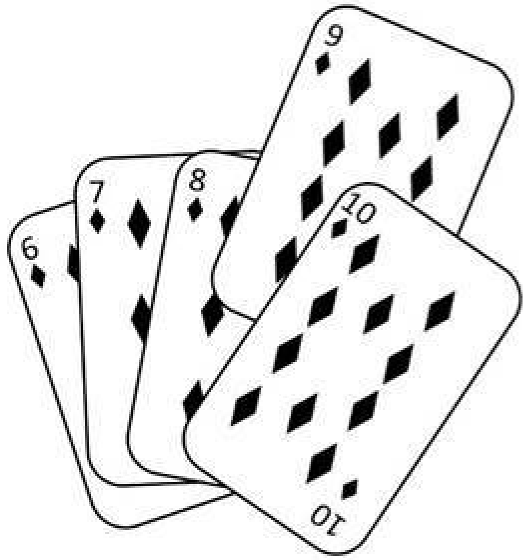
By applying this improvement, best case performance of this algorithm when a list is nearly sorted, is improved. In this case we just need one single pass and best case complexity is **O(n)**

Complexity Analysis:

Worst case performance	$O(n^2)$
Average case performance	$O(n^2)$
Space Complexity	O(1)
Adaptive: When list is nearly sorted	O(n)
Stable Sorting	Yes

Insertion-Sort

Insertion-Sort Time Complexity is $\mathbf{O(n^2)}$ which is same as Bubble-Sort but perform a bit better than it. It is the way we arrange our playing cards. We keep a sorted subarray. Each value is inserted into its proper position in the sorted sub-array in the left of it.



5	6	2	4	7	3	1	Insert 5
5	6	2	4	7	3	1	Insert 6
2	5	6	4	7	3	1	Insert 2
2	4	5	6	7	3	1	Insert 4
2	4	5	6	7	3	1	Insert 7
2	3	4	5	6	7	1	Insert 3
1	2	3	4	5	6	7	Insert 1

Example 6.3

```
func InsertionSort(arr []int, comp func(int, int) bool) {
	size := len(arr)
	var temp, i, j int
	for i = 1; i < size; i++ {
		temp = arr[i]
		for j = i; j > 0 && comp(arr[j-1], temp); j-- {
			arr[j] = arr[j-1]
		}
		arr[j] = temp
	}
}

func main() {
	data := []int{9, 1, 8, 2, 7, 3, 6, 4, 5}
```

```
    InsertionSort(data, more)
    fmt.Println(data)
}
```

Analysis:

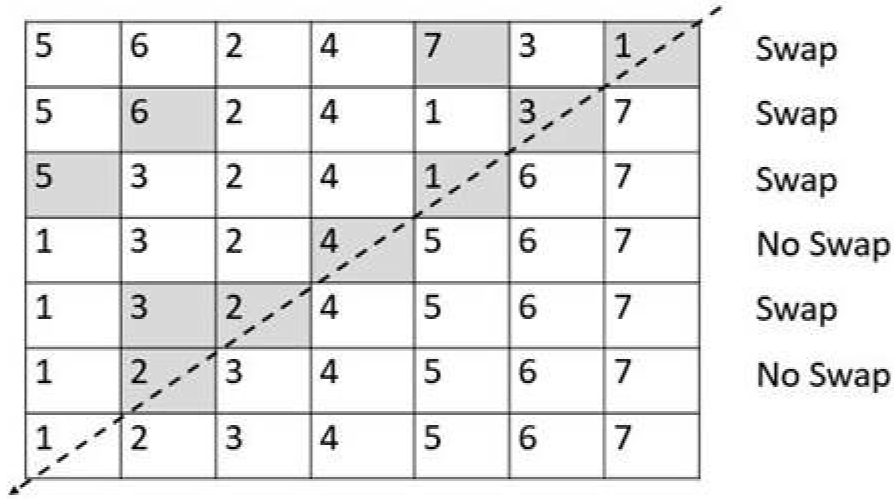
- The outer loop is used to pick the value we want to insert into the sorted list in left.
- The value we want to insert we have picked and saved in a temp variable.
- The inner loop is doing the comparison using the more() function. The values are shifted to the right until we find the proper position of the temp value for which we are doing this iteration.
- Finally, the value is placed into the proper position. In each iteration of the outer loop, the length of the sorted list increase by one. When we exit the outer loop, the whole list is sorted.

Complexity Analysis:

Worst case Time Complexity	$O(n^2)$
Best case Time Complexity	O(n)
Average case Time Complexity	$O(n^2)$
Space Complexity	O(1)
Stable sorting	Yes

Selection-Sort

Selection-Sort searches the whole unsorted array and put the largest value at the end of it. This algorithm is having the same Time Complexity, but performs better than both bubble and Insertion-Sort as less number of comparisons required. The sorted list is created backward in Selection-Sort.



Example 6.4:

```
func SelectionSort(arr []int) {
	size := len(arr)
	var i, j, max, temp int
	for i = 0; i < size-1; i++ {
		max = 0
		for j = 1; j < size-1-i; j++ {
			if arr[j] > arr[max] {
				max = j
			}
		}
		arr[size-1-i], arr[max] = arr[max], arr[size-1-i]
	}
}
```

Analysis:

- The outer loop decide the number of times the inner loop will iterate. For an input of N elements, the inner loop will iterate N number of times.
- In each iteration of the inner loop, the largest value is calculated and is placed at the end of the list.
- This is the final replacement of the maximum value to the proper location. The sorted list is created backward.

Complexity Analysis:

Worst Case Time Complexity	$O(n^2)$
Best Case Time Complexity	$O(n^2)$
Average case Time Complexity	$O(n^2)$
Space Complexity	O(1)
Stable Sorting	No

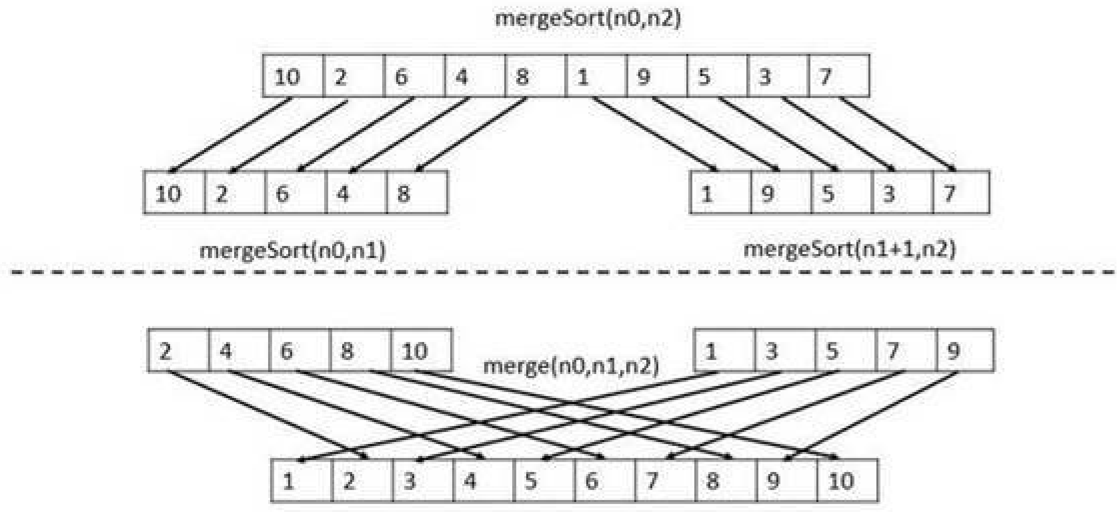
The same algorithm can be implemented by creating the sorted list in the front of the list.

Example 6.5:

```
func SelectionSort2(arr []int) {
	size := len(arr)
	var i, j, min, temp int
	for i = 0; i < size-1; i++ {
		min = i
		for j = i + 1; j < size; j++ {
			if arr[j] < arr[min] {
				min = j
			}
		}
		arr[i], arr[min] = arr[min], arr[i]
	}
}
```

Merge-Sort

Merge sort divide the input into half recursive in each step. It sort the two parts separately recursively and finally combine the result into final sorted output.



Example 6.6:

```
func MergeSort(arr []int, comp func(int, int) bool) {
	size := len(arr)
	tempArray := make([]int, size)
	mergeSrt(arr, tempArray, 0, size-1, comp)
}

func merge(arr []int, tempArray []int, lowerIndex int, middleIndex int, upperIndex int, comp func(int, int) bool) {
	lowerStart := lowerIndex
	lowerStop := middleIndex
	upperStart := middleIndex + 1
	upperStop := upperIndex
	count := lowerIndex
	for lowerStart <= lowerStop && upperStart <= upperStop {
		if comp(arr[lowerStart], arr[upperStart]) == false {
			tempArray[count] = arr[lowerStart]
			lowerStart++
		} else {
			tempArray[count] = arr[upperStart]
			upperStart++
		}
		count++
	}
	for lowerStart <= lowerStop {
		tempArray[count] = arr[lowerStart]
		count++
		lowerStart++
	}
	for upperStart <= upperStop {
		tempArray[count] = arr[upperStart]
		count++
		upperStart++
```

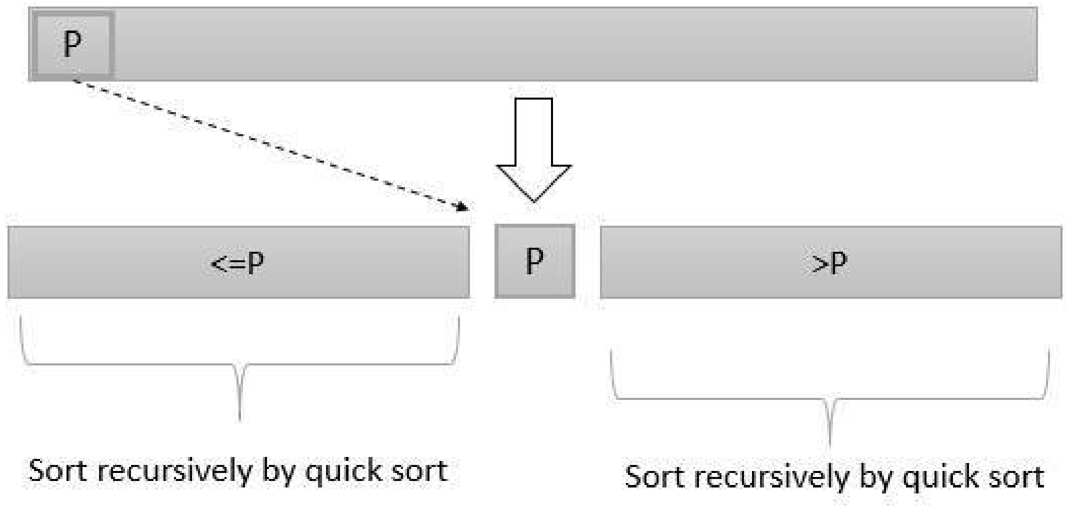
```
    }
    for i := lowerIndex; i <= upperIndex; i++ {
        arr[i] = tempArray[i]
    }
}

func mergeSrt(arr []int, tempArray []int, lowerIndex int, upperIndex int, comp func(int, int) bool) {
    if lowerIndex >= upperIndex {
        return
    }
    middleIndex := (lowerIndex + upperIndex) / 2
    mergeSrt(arr, tempArray, lowerIndex, middleIndex, comp)
    mergeSrt(arr, tempArray, middleIndex+1, upperIndex, comp)
    merge(arr, tempArray, lowerIndex, middleIndex, upperIndex, comp)
}
func main() {
    data := []int{9, 1, 8, 2, 7, 3, 6, 4, 5}
    MergeSort(data, more)
    fmt.Println(data)
}
```

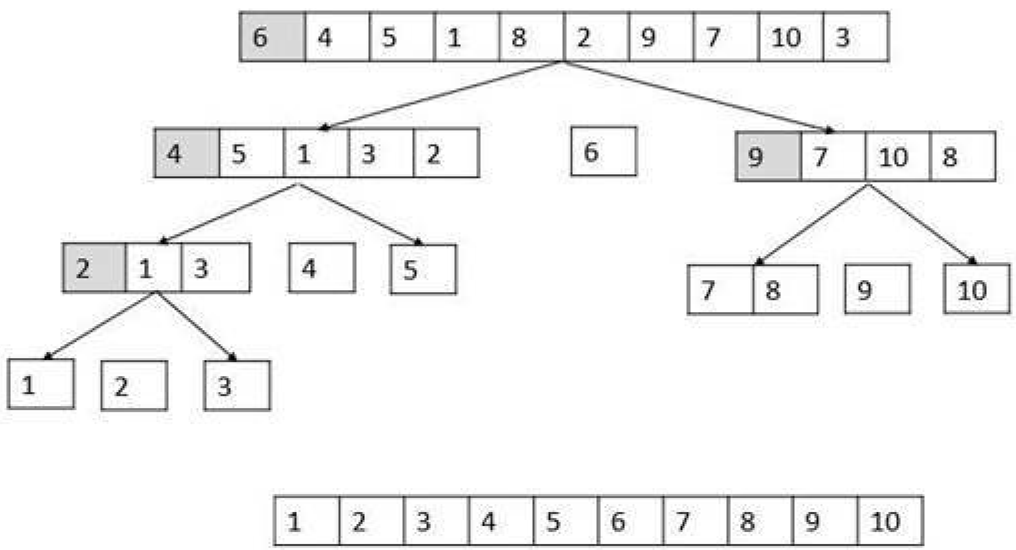
- The Time Complexity of Merge-Sort is **O(nlogn)** in all 3 cases (best, average and worst) as Merge-Sort always divides the list into two halves and take linear time to merge two halves.
- It requires the equal amount of additional space as the unsorted list. Hence, it is not at all recommended for searching large unsorted lists.
- It is the best Sorting technique for sorting Linked Lists.

Complexity Analysis:

Worst Case Time Complexity	O(nlogn)
Best Case Time Complexity	O(nlogn)
Average Time Complexity	O(nlogn)
Space Complexity	O(n)
Stable Sorting	Yes

Quick-Sort





Quick sort is also a recursive algorithm.

- In each step, we select a pivot (let us say first element of list).
- Then we traverse the rest of the list and copy all the elements of the list which are smaller than the pivot to the left side of list
- We copy all the elements of the list, which are greater than pivot to the right side of the list. Obviously, the pivot is at its sorted position.
- Then we sort the left and right subarray separately.
- When the algorithm returns the whole list is sorted.

Example 6.7:

```
func QuickSort(arr []int, comp func(int, int) bool) {
    size := len(arr)
    quickSortUtil(arr, 0, size-1, comp)
```

```
}

func quickSortUtil(arr []int, lower int, upper int, comp func(int, int) bool) {
	if upper <= lower {
		return
	}
	pivot := arr[lower]
	start := lower
	stop := upper

	for lower < upper {
		for comp(arr[lower], pivot) == false && lower < upper {
			lower++
		}
		for comp(arr[upper], pivot) && lower <= upper {
			upper--
		}
		if lower < upper {
			swap(arr, upper, lower)
		}
	}
	swap(arr, upper, start)			// upper is the pivot position
	quickSortUtil(arr, start, upper-1, comp) // pivot -1 is the upper for left sub array.
	quickSortUtil(arr, upper+1, stop, comp)  // pivot + 1 is the lower for right sub array.
}

func swap(arr []int, first int, second int) {
	arr[first], arr[second] = arr[second], arr[first]
}
func main() {
	data := []int{9, 1, 8, 2, 7, 3, 6, 4, 5}
	QuickSort(data, more)
	fmt.Println(data)
}
```

- The space required by Quick-Sort is very less, only O(nlogn) additional space is required.
- Quicksort is not a stable sorting technique. It can reorder elements with identical keys.

Complexity Analysis:

Worst Case Time Complexity	$O(n^2)$
Best Case Time Complexity	O(nlogn)
Average Time Complexity	O(nlogn)
Space Complexity	O(nlogn)
Stable Sorting	No

Quick Select

Quick select algorithm is used to find the element, which will be at the Kth position when the list will be sorted without actually sorting the whole list. Quick select is very similar to Quick-Sort in place of sorting the whole list we just ignore the one-half of the list at each step of Quick-Sort and just focus on the region of list on which we are interested.

Example 6.8:

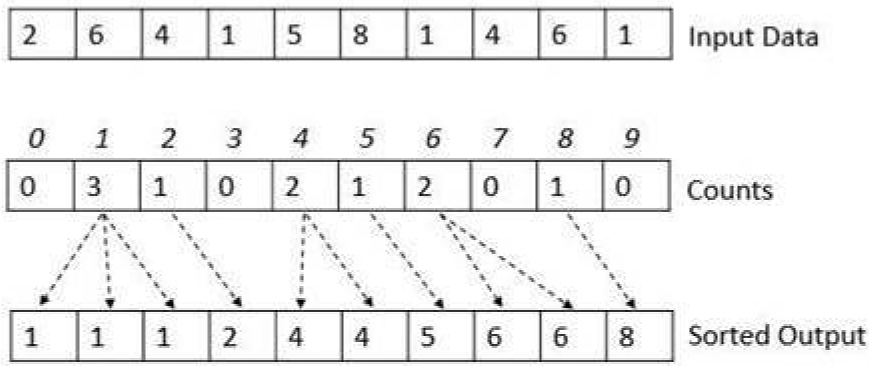
```
func QuickSelect(arr []int, key int) int {
	size := len(arr)
	QuickSelectUtil(arr, 0, size-1, key)
	return arr[key-1]
}

func QuickSelectUtil(arr []int, lower int, upper int, key int) {
	if upper <= lower {
		return
	}
	pivot := arr[lower]
	start := lower
	stop := upper
	for lower < upper {
		for arr[lower] <= pivot && lower < upper {
			lower++
		}
		for arr[upper] > pivot && lower <= upper {
			upper--
		}
		if lower < upper {
			swap(arr, upper, lower)
		}
	}
	swap(arr, upper, start)			// upper is the pivot position
	QuickSelectUtil(arr, start, upper-1, key) // pivot -1 is the upper for left sub array.
	QuickSelectUtil(arr, upper+1, stop, key)  // pivot + 1 is the lower for right sub array.
}

func swap(arr []int, first int, second int) {
	arr[first], arr[second] = arr[second], arr[first]
}

func main() {
	data := []int{9, 1, 8, 2, 7, 3, 6, 4, 5}
	fmt.Println(QuickSelect(data, 5))
}
```

Complexity Analysis:

Worst Case Time Complexity	$O(n^2)$
Best Case Time Complexity	O(logn)
Average Time Complexity	O(logn)
Space Complexity	O(nlogn)

Bucket Sort

Bucket sort is the simplest and most efficient type of sorting. Bucket sort has a strict requirement of a predefined range of data.
Like, sort how many people are in which age group. We know that the age of people can vary between 0 and 130.



Example 6.9:

```
func BucketSort(data []int, lowerRange int, upperRange int) {
    rng := upperRange - lowerRange
    size := len(data)
    count := make([]int, rng)
    for i := 0; i < rng; i++ {
        count[i] = 0
    }
    for i := 0; i < size; i++ {
        count[data[i]-lowerRange]++
    }
    j := 0
    for i := 0; i < rng; i++ {
        for ; count[i] > 0; count[i]-- {
            data[j] = i + lowerRange
            j++
        }
    }
}
func main() {
    data := []int{23, 24, 22, 21, 26, 25, 27, 28, 21, 21}
    BucketSort(data, 20, 30)
    fmt.Println(data)
}
```

Analysis:

- We have created a count list to store counts.
- Count list elements are initialized to zero.
- Index corresponding to input list is incremented.
- Finally, the information stored in count list is saved in the list.

Complexity Analysis:

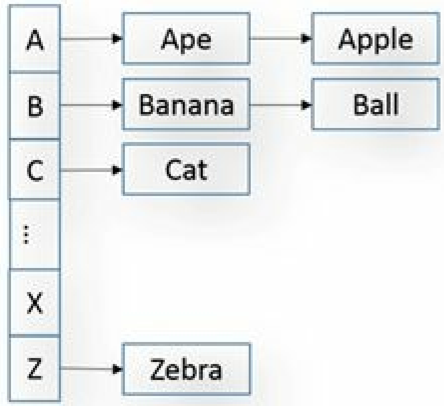
Data structure	List
Worst case performance	O(n+k)
Average case performance	O(n+k)

Worst case Space Complexity	O(k)

k - number of distinct elements.
n - total number of elements in list.

Generalized Bucket Sort

There are cases when the element falling into a bucket are not unique but are in the same range. When we want to sort an index of a name, we can use the reference bucket to store names.



The buckets are already sorted and the elements inside each bucket can be kept sorted by using an Insertion-Sort algorithm. We are leaving this generalized bucket sort implementation to the reader of this book. The similar data structure will be defined in the coming chapter of Hash-Table using separate chaining.

Heap-Sort

Heap-Sort we will study in the Heap chapter.

Complexity Analysis:

Data structure	List
Worst case performance	O(nlogn)
Average case performance	O(nlogn)
Worst case Space Complexity	O(1)

Tree Sorting

In-order traversal of the binary search tree can also be seen as a sorting algorithm. We will see this in binary search tree section of tree chapter.

Complexity Analysis:

Worst Case Time Complexity	$O(n^2)$
Best Case Time Complexity	O(nlogn)
Average Time Complexity	O(nlogn)
Space Complexity	O(n)
Stable Sorting	Yes

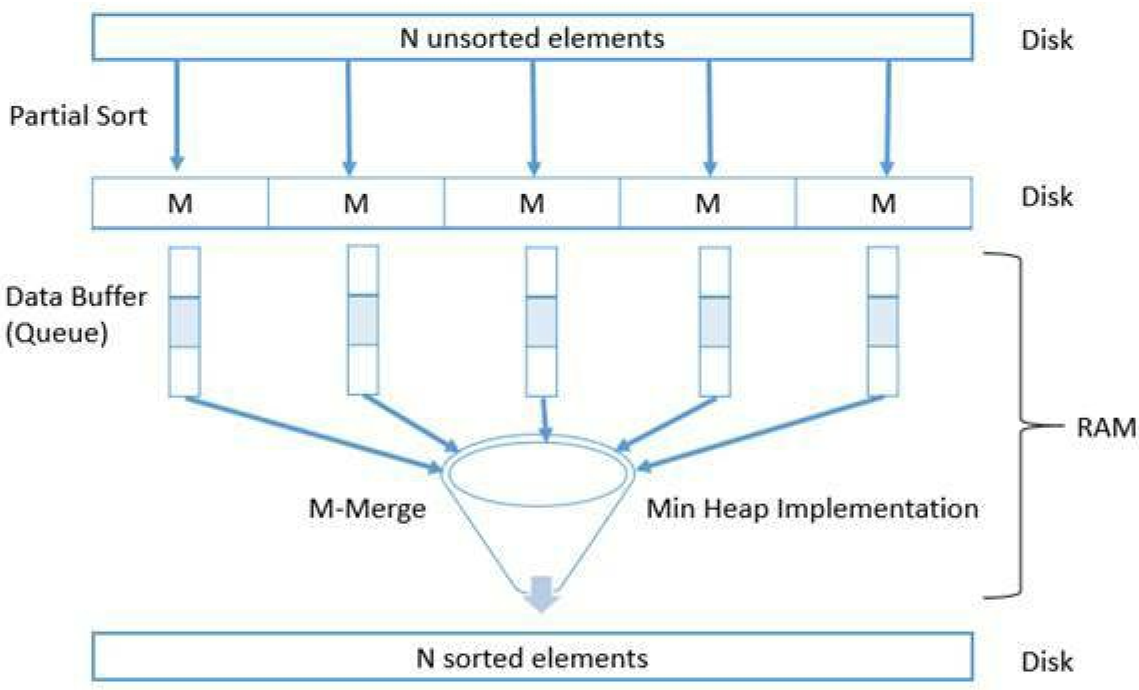
External Sort (External Merge-Sort)

When data need to be sorted is huge and it is not possible to load it completely in memory (RAM), for such a dataset we use external sorting. Such data is sorted using external Merge-Sort algorithm. First data is picked in chunks and it is sorted in memory. Then this sorted data is written back to disk. Whole data are sorted in chunks using Merge-Sort. Now we need to combine these sorted chunks into final sorted data.

Then we create queues for the data, which will read from the sorted chunks. Each chunk will have its own queue. We will pop from this queue and these queues are responsible for reading from the sorted chunks. Let us suppose we have K different chunks of sorted data each of length M.

The third step is using a Min-Heap, which will take input data from each of this queue. It will take one element from each queue. The minimum value is taken from the Heap and added to the final sorted element output. Then queue from which this min element is inserted in the heap will again popped and one more element from that queue is added to the Heap. Finally, when the data is exhausted from some queue that queue is removed from the input list. Finally, we will get a sorted data came out from the heap.

We can optimize this process further by adding an output buffer, which will store data coming out of Heap and will do a limited number of the write operation in the final Disk space.



Note: No one will be asking to implement external sorting in an interview, but it is good to know about it.

Comparisons of the various sorting algorithms.

Below is comparison of various sorting algorithms:

Sort	Average Time	Best Time	Worst Time	Space	Stable
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	Yes
Modified Bubble Sort	$O(n^2)$	O(n)	$O(n^2)$	O(1)	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No
Insertion Sort	$O(n^2)$	O(n)	$O(n^2)$	O(1)	Yes
Heap Sort	O(n * log(n))	O(n * log(n))	O(n * log(n))	O(1)	No
Merge Sort	O(n * log(n))	O(n * log(n))	O(n * log(n))	O(n)	Yes
Quick Sort	O(n * log(n))	O(n * log(n))	$O(n^2)$	O(n) worst case O(log(n)) average case	No
Bucket Sort	O(n k)	O(n k)	O(n k)	O(n k)	Yes

Selection of Best Sorting Algorithm

No sorting algorithm is perfect. Each of them has their own pros and cons. Let us read one by one:

Quick-Sort: When you do not need a stable sort and average case performance matters more than worst-case performance. When data is random, we prefer the Quick-Sort. Average case Time Complexity of Quick-Sort is **O(nlogn)** and worst-case Time Complexity is **$O(n^2)$.** Space Complexity of Quick-Sort is O**(logn)** auxiliary storage, which is stack space used in recursion.

Merge-Sort: When you need a stable sort and Time Complexity of **O(nlogn)**, Merge-Sort is used. In general, Merge-Sort is slower than Quick-Sort because of lot of copy happening in the merge phase. There are two uses of Merge-Sort when we want to merge two sorted linked lists and Merge-Sort is used in external sorting.

Heap-Sort: When you do not need a stable sort and you care more about worst-case performance than average case performance. It has guaranteed to be **O(nlogn)**, and uses **O(1)** auxiliary space, meaning that you will not unpredictably run out of memory on very large inputs.

Insertion-Sort: When we need a stable sort, When N is guaranteed to be small, including as the base case of a Quick-Sort or Merge-Sort. Worst-case Time Complexity is **$O(n^2)$**. It has a very small constant factor multiplied to calculate actual time taken. Therefore, for smaller input size it performs better than Merge-Sort or Quick-Sort. It is also useful when the data is already pre-sorted. In this case, its running time is **O(N)**.

Bubble-Sort: Where we know the data is nearly sorted. Say only two elements are out of place. Then in one pass, Bubble Sort will make the data sorted and in the second pass, it will see everything is sorted and then exit. Only takes 2 passes of the list.

Selection-Sort: Best Worst Average Case running time all **$O(n^2)$**. It is only useful when you want to do something quick. They can be used when you are just doing some prototyping.

Counting-Sort: When you are sorting data within a limited range.

Radix-Sort: When log(N) is significantly larger than K, where K is the number of radix digits.

Bucket-Sort: When your input is more or less uniformly distributed.

Note: A stable sort is one that has guaranteed not to reorder elements with identical keys.

Exercise

1. Given a text file, print the words with their frequency. Now print the kth word in term of frequency.
   Hint:-
   a) First approach may be you can use the sorting and return the kth element.
   b) Second approach: You can use the kth element quick select algorithm.
   c) Third approach: You can use Hashtable or Trie to keep track of the frequency. Use Heap to get the Kth element.

2. Given K input streams of number in sorted order. You need to make a single output stream, which contains all the elements of the K streams in sorted order. The input streams support ReadNumber() operation and output stream support WriteNumber() operation.
   Hint:-
   a) Read the first number from all the K input streams and add them to a Priority Queue. (Nodes should keep track of the input stream)
   b) Dequeue one element at a time from PQ, Put this element value to the output stream, Read the input stream number and from the same input stream add another element to PQ.
   c) If the stream is empty, just continue
   d) Repeat until PQ is empty.

3. Given K sorted Lists of fixed length M. Also, given a final output list of length M*K. Give an efficient algorithm to merge all the Lists into the final list, without using any extra space.
   Hint: you can use the end of the final list to make PQ.

4. How will you sort 1 PB numbers? 1 PB = 1000 TB.

5. What will be the complexity of the above solution?

6. Any other improvement on question 3 solution if the number of cores is eight.

7. Given an integer list that support three function findMin, findMax, findMedian. Sort the list.

8. Given a pile of patient files of High, mid and low priority. Sort these files such that higher priority comes first, then mid and last low priority.
   Hint: Bucket sort.

9. Write pros and cons of Heap-Sort, Merge-Sort and Quick-Sort.

10. Given a rotated - sorted list of N integers. (The list was sorted then it was rotated some arbitrary number of times.) If all the elements in the list were unique the find the index of some value.
    Hint: Modified binary search

11. In the problem 9, what if there are repetitions allowed and you need to find the index of the first occurrence of the element in the rotated-sorted list.

12. Merge two sorted Lists into a single sorted list.
    Hint: Use merge method of Merge-Sort.

13. Given a list contain 0’s and 1’s, sort the list such that all the 0’s come before 1’s.

14. Given a list of English characters, sort the list in linear time.

15. Write a method to sort a list of strings so that all the anagrams are next to each other.

Hint:-

a) Loop through the list.
b) For each word, sort the characters and add it to the hash map with keys as sorted word and value as the original word. At the end of the loop, you will get all anagrams as the value to a key (which is sorted by its constituent chars).
c) Iterate over the hashmap, print all values of a key together and then move to the next key.
   Space Complexity: O(n), Time Complexity: O(n)

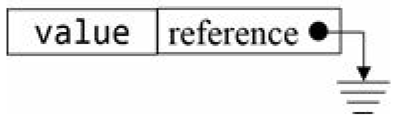
CHAPTER 7: LINKED LIST

Introduction

Let us suppose we have a list that contains following five elements 1, 2, 4, 5, 6. We want to insert a new element with value “3” in between “2” and “4”. In the list, we cannot do so easily. We need to create another list that is long enough to store the current values and one more space for “3”. Then we need to copy these elements in the new space. This copy operation is inefficient. To remove this copy operation linked list is used.

Linked List

The linked list is a list of items, called nodes. Nodes have two parts, value part and link part. Value part is used to stores the data. Either the value part of the node can be a basic data-type like an integer or it can be some other data-type like an object of some class.
The link part is a reference, which is used to store addresses of the next element in the list.

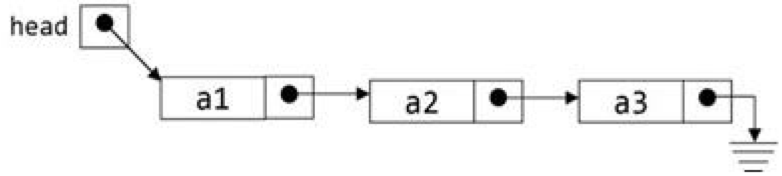


Types of Linked list

There are different types of linked lists. The main difference among them is how their nodes refer to each other.

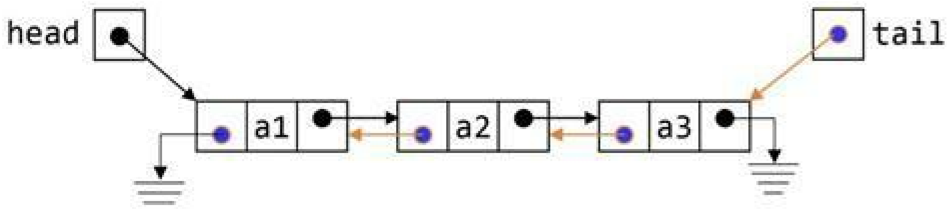
Singly Linked List

Each node (Except the last node) has a reference to the next node in the linked list. The link portion of node contains the address of the next node. The link portion of the last node contains the value null



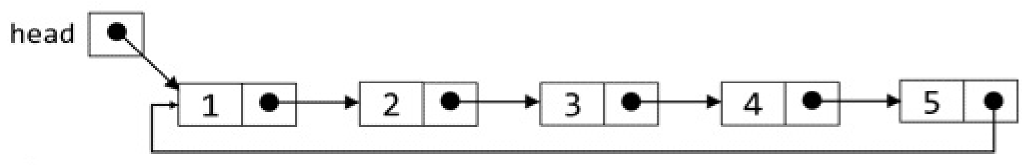
Doubly Linked list

The node in this type of linked list has reference to both previous and the next node in the list.



Circular Linked List

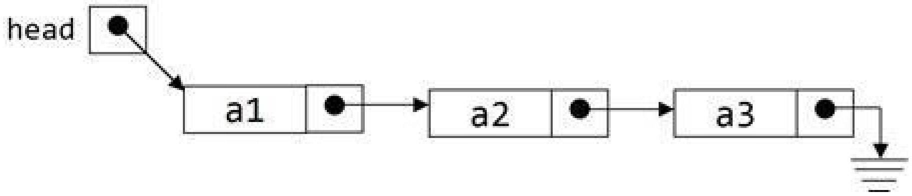
This type is similar to the singly linked list except that the last element have reference to the first node of the list. The link portion of the last node contains the address of the first node.



The various parts of linked list

1. Head: Head is a reference that holds the address of the first node in the linked list.
2. Nodes: Items in the linked list are called nodes.
3. Value: The data that is stored in each node of the linked list.
4. Link: Link part of the node is used to store the reference of other node.
    a. We will use “next” and “prev” to store address of next or previous node.

Singly Linked List



Let us look at the Node. The value part of node is of type integer, but it can be some other data-type. The link part of node is named as next in the below class definition.

Note: For a singly linked, we should always test these three test cases before saying that the code is good to go. This one node and zero node case is used to catch boundary cases. It is always to take care of these cases before submitting code to the reviewer.

- Zero element / Empty linked list.
- One element / Just single node case.
- General case.

The various basic operations that we can perform on linked lists, many of these operations require list traversal:

- Insert an element in the list, this operation is used to create a linked list.
- Print various elements of the list.
- Search an element in the list.
- Delete an element from the list.
- Reverse a linked list.

You cannot use Head to traverse a linked list because if we use the head, then we lose the nodes of the list. We have to use another reference variable of same data-type as the head.

Example 7.1:
```
type List struct {
    head  *Node
    count int
}

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

Size of List

Example 7.2:
```
func (list *List) Size() int {
    return list.count
}
```

IsEmpty function

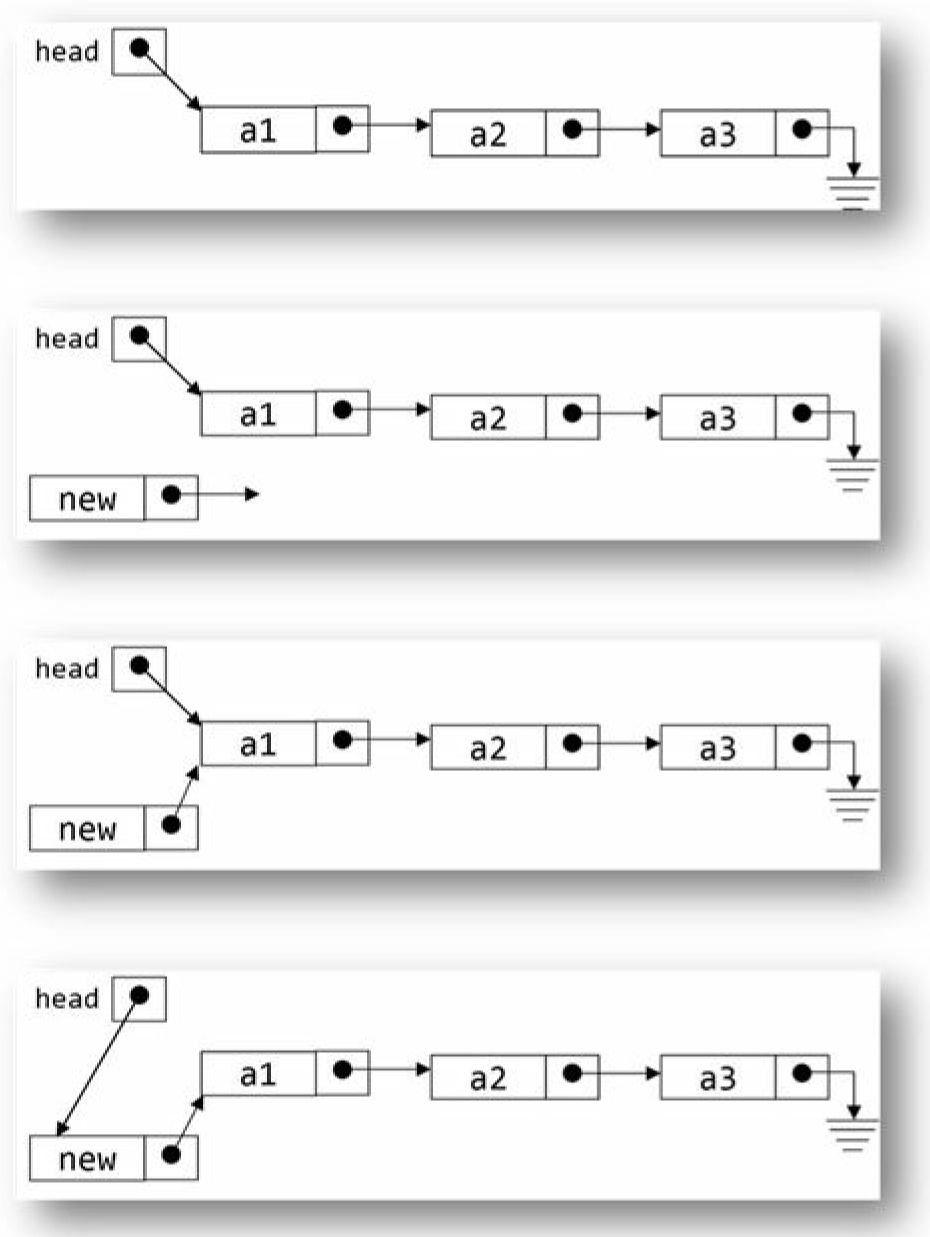
Example 7.3:
```
func (list *List) IsEmpty() bool {
```

```
    return (list.count == 0)
}
```

Insert element in linked list

An element can be inserted into a linked list in various orders. Some of the example cases are mentioned below:

1. Insertion of an element at the start of linked list
2. Insertion of an element at the end of linked list
3. Insertion of an element at the Nth position in linked list
4. Insert element in sorted order in linked list



Insert element at the Head

Example 7.4:

```
func (list *List) AddHead(value int) {
    list.head = &Node{value, list.head}
```

```
    list.count++
}
```

Analysis:

- We need to create a new node with the value passed to the function as argument.
- While creating the new node the reference stored in head is passed as argument to Node() constructor so that the next reference will start pointing to the node or null which is referenced by the head node.
- The newly created node will become head of the linked list.
- Size of the list is increased by one.

Insertion of an element at the end

Example 7.5: Insertion of an element at the end of linked list

```
func (list *List) addTail(value int) {
    curr := list.head
    newNode := &Node{value, nil}

    if curr == nil {
        list.head = newNode
        return
    }

    for curr.next != nil {
        curr = curr.next
    }

    curr.next = newNode
}
```

Analysis:

- New node is created and the value is stored inside it.
- If the list is empty. Next of new node is null. And head will store the reference to the newly created node.
- If list is not empty then we will traverse until the end of the list.
- Finally, new node is added to the end of the list.

Note: This operation is un-efficient as each time you want to insert an element you have to traverse to the end of the list. Therefore, the complexity of creation of the list is n^2. So how to make it efficient we have to keep track of the last element by keeping a tail reference. Therefore, if it is required to insert element at the end of linked list, then we will keep track of the tail reference also.

Traversing Linked List

Example 7.6: Print various elements of a linked list

```
func (list *List) Print() {
    temp := list.head
    for temp != nil {
        fmt.Print(temp.value, " ")
        temp = temp.next
    }
    fmt.Println("")
}
```

Analysis:
We will store the reference of head in a temporary variable temp.
We will traverse the list by printing the content of list and always incrementing the temp by pointing to its next node.

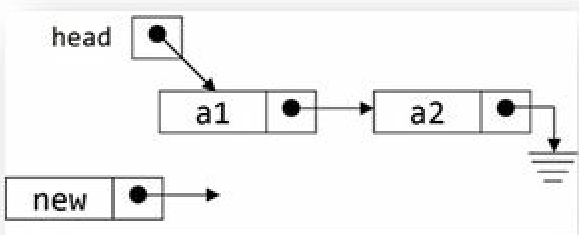
Complete code for list creation and printing the list.

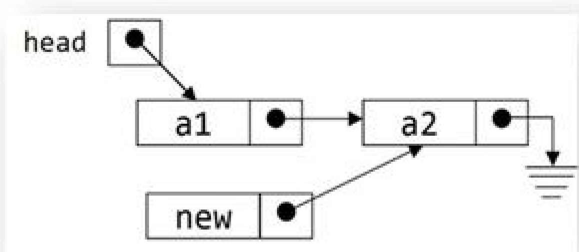
Example 7.7:
```
func main() {
    //lst := new(List)
    lst := &List{}
    lst.AddHead(1)
    lst.AddHead(2)
    lst.AddHead(3)
    lst.Print()
}
```

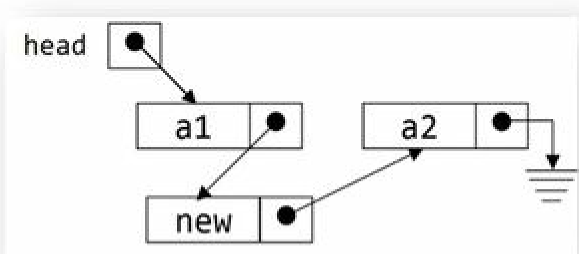
Analysis:
New instance of linked list is created. Various elements are added to list by calling addHead() method.
Finally all the content of list is printed to screen by calling print() method.

Sorted Insert

Insert an element in sorted order in linked list given Head reference







Example 7.8:

```
func (list *List) SortedInsert(value int) {
	newNode := &Node{value, nil}
	curr := list.head

	if curr == nil || curr.value > value {
		newNode.next = list.head
		list.head = newNode
		return
	}
	for curr.next != nil && curr.next.value < value {
		curr = curr.next
	}
	newNode.next = curr.next
	curr.next = newNode
}
```

Analysis:

- Head of the list is stored in curr.
- A new empty node of the linked list is created. And initialized by storing an argument value into its value. Next of the node will point to null.
- It checks if the list was empty or if the value stored in the first node is greater than the current value. Then this new created node will be added to the start of the list. And head need to be modified.
- We iterate through the list to find the proper position to insert the node.
- Finally, the node will be added to the list.

Search Element in a Linked-List

Search element in linked list. Given a head reference and value. Returns true if value found in list else returns false.

Note: Search in a single linked list can be only done in one direction. Since all elements in the list have reference to the next item in the list. Therefore, traversal of linked list is linear in nature.

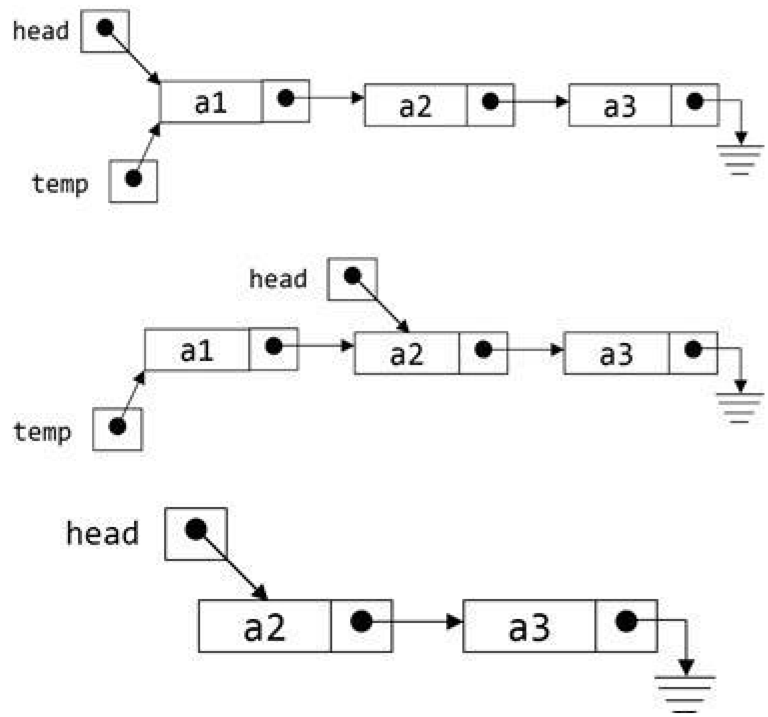
Example 7.9:

```
func (list *List) IsPresent(data int) bool {
	temp := list.head
	for temp != nil {
		if temp.value == data {
			return true
		}
		temp = temp.next
	}
	return false
}
```

Analysis:

- We create a temp variable, which will point to head of the list.
- Using a “for” loop we will iterate through the list.
- Value of each element of list is compared with the given value. If value is found, then the function will return true.
- If the value is not found, then false will be returned from the function in the end.

Delete element from the linked list



Delete First element in a linked list.

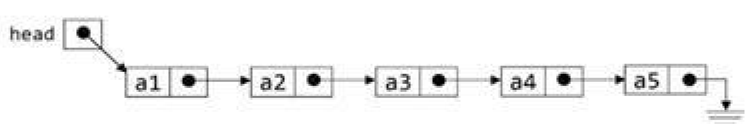
Example 7.10:

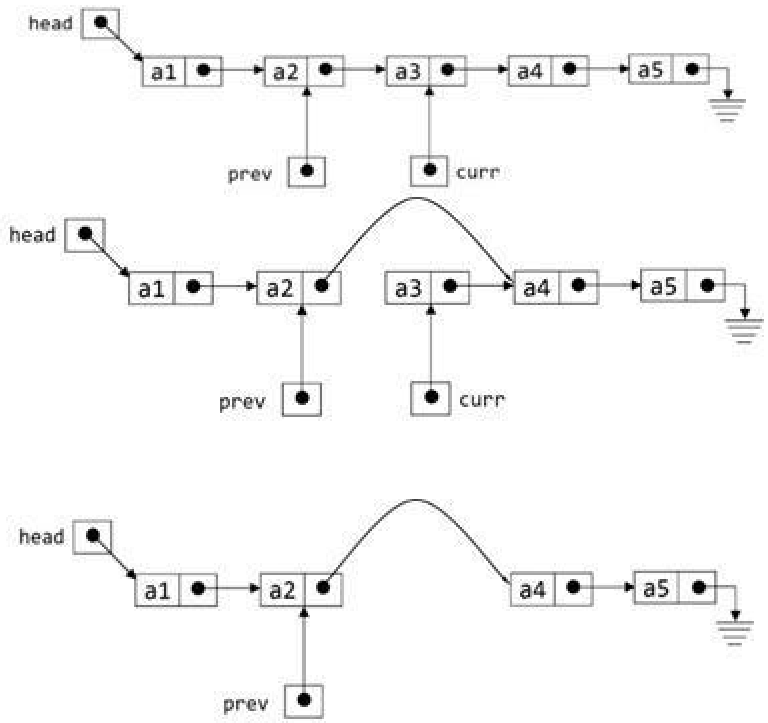
```
func (list *List) RemoveHead() (int, bool) {
    if list.IsEmpty() {
        fmt.Println("EmptyListError")
        return 0, false
    }
    value := list.head.value
    list.head = list.head.next
    list.count--
    return value, true
}
```

Analysis:

- First, we need to check if the list is empty. If list is already empty then we print error statement and return error.
- If list is not empty then store the value of head node in a temporary variable value.
- We need to find the second element of the list and assign it as head of the linked list.
- Since the first node is no longer referenced so it will be automatically deleted.
- Decrease the size of list. Then return the value stored in temporary variable value.

Delete node from the linked list given its value.





Example 7.11:

```
func (list *List) DeleteNode(delValue int) bool {
	temp := list.head
	if list.IsEmpty() {
		fmt.Println("EmptyListError")
		return false
	}
	if delValue == list.head.value {
		list.head = list.head.next
		list.count--
		return true
	}

	for temp.next != nil {
		if temp.next.value == delValue {
			temp.next = temp.next.next
			list.count--
			return true
		}
		temp = temp.next
	}
	return false
}
```

Analysis:

- If the list is empty then we will return false from the function which indicate that the deleteNode() method executed with error.
- If the node that need to be deleted is head node then head reference need is modified and point to the next node.
- In a for loop we will traverse the link list and try to find the node that need to be deleted. If the node is found then, we will point its reference to the node next to it and return true.
- If the node is not found then we will return false.

Delete all the occurrence of particular value in linked list.

Example 7.12:

```
func (list *List) DeleteNodes(delValue int) {
    currNode := list.head
    for currNode != nil && currNode.value == delValue {
        list.head = currNode.next
        currNode = list.head
    }
    for currNode != nil {
        nextNode := currNode.next
        if nextNode != nil && nextNode.value == delValue {
            currNode.next = nextNode.next
        } else {
            currNode = nextNode
        }
    }
}
```

Analysis:

- In the first for loop will delete all the nodes that are at the front of the list, which have valued equal to delValue. In this, we need to update head of the list.
- In the second for loop, we will be deleting all the nodes that are having value equal to the delValue. Remember that we are not returning even though we have the node that we are looking for.

Delete a single linked list

Delete all the elements of a linked list, given a reference to head of linked list.

Example 7.13:

```
func (list *List) FreeList() {
    list.head = nil
    list.count = 0
}
```

Analysis: We just need to point head to null. The reference to the list is lost so it will automatically deleted.

Reverse a linked list.

Reverse a singly linked List iteratively using three Pointers

Example 7.14:

```
func (list *List) Reverse() {
    curr := list.head
    var prev, next *Node
    for curr != nil {
        next = curr.next
        curr.next = prev
        prev = curr
        curr = next
    }
    list.head = prev
}
```

Analysis: The list is iterated. Make next equal to the next node of the curr node. Make curr node's next point to prev node. Then iterate the list by making prev point to curr and curr point to next.

Recursively Reverse a singly linked List

Reverse a singly linked list using Recursion.

Example 7.15:
```
func (list *List) ReverseRecurse() {
	list.head = list.reverseRecurseUtil(list.head, nil)
}

func (list *List) reverseRecurseUtil(currentNode *Node, nextNode *Node) *Node {
	var ret *Node
	if currentNode == nil {
		return nil
	}
	if currentNode.next == nil {
		currentNode.next = nextNode
		return currentNode
	}

	ret = list.reverseRecurseUtil(currentNode.next, currentNode)
	currentNode.next = nextNode
	return ret
}
```

Analysis:
- ReverseRecurse function will call a reverseRecurseUtil function to reverse the list and the reference returned by the reverseRecurseUtil will be the head of the reversed list.
- The current node will point to the nextNode that is previous node of the old list.

Note: A linked list can be reversed using two approaches the first approach is by using three references. The Second approach is using recursion both are linear solution, but three-reference solution is more efficient.

Remove duplicates from the linked list

Remove duplicate values from the linked list. The linked list is sorted and it contains some duplicate values, you need to remove those duplicate values. (You can create the required linked list using SortedInsert() function)

Example 7.16:
```
func (list *List) RemoveDuplicate() {
	curr := list.head
	for curr != nil {
		if curr.next != nil && curr.value == curr.next.value {
			curr.next = curr.next.next
		} else {
			curr = curr.next
		}
	}
}
```

Analysis: for loop is used to traverse the list. Whenever there is a node whose value is equal to the next node's value, that current node next will point to the next of next node. Which will remove the next node from the list.

Copy List Reversed

Copy the content of linked list in another linked list in reverse order. If the original linked list contains elements in order 1,2,3,4, the new list should contain the elements in order 4,3,2,1.

Example 7.17:
```
func (list *List) CopyListReversed() *List {
    var tempNode, tempNode2 *Node
    curr := list.head
    for curr != nil {
        tempNode2 = &Node{curr.value, tempNode}
        curr = curr.next
        tempNode = tempNode2
    }
    ll2 := new(List)
    ll2.head = tempNode
    return ll2
}
```

Analysis: Traverse the list and add the node's value to the new list. Since the list is traversed in the forward direction and each node's value is added to another list so the formed list is reverse of the given list.

Copy the content of given linked list into another linked list

Copy the content of given linked list into another linked list. If the original linked list contains elements in order 1,2,3,4, the new list should contain the elements in order 1,2,3,4.

Example 7.18:
```
func (list *List) CopyList() *List {
    var headNode, tailNode, tempNode *Node
    curr := list.head

    if curr == nil {
        ll2 := new(List)
        ll2.head = nil
        return ll2
    }

    headNode = &Node{curr.value, nil}
    tailNode = headNode
    curr = curr.next

    for curr != nil {
        tempNode = &Node{curr.value, nil}
        tailNode.next = tempNode
        tailNode = tempNode
        curr = curr.next
    }
```

```
    ll2 := new(List)
    ll2.head = headNode
    return ll2
}
```

Analysis: Traverse the list and add the node's value to new list, but this time always at the end of the list. Since the list is traversed in the forward direction and each node's value is added to the end of another list. Therefore, the formed list is same as the given list.

Compare List

Compare the values of two linked lists given their head pointers.

Example 7.19: Compare two list given

```
func (list *List) CompareList(ll *List) bool {
    return list.compareListUtil(list.head, ll.head)
}

func (list *List) compareListUtil(head1 *Node, head2 *Node) bool {
    if head1 == nil && head2 == nil {
        return true
    } else if (head1 == nil) || (head2 == nil) || (head1.value != head2.value) {
        return false
    } else {
        return list.compareListUtil(head1.next, head2.next)
    }
}
```

Analysis:

- List is compared recursively. Moreover, if we reach the end of the list and both the lists are null. Then both the lists are equal and so return true.
- List is compared recursively. If either one of the list is empty or the value of corresponding nodes is unequal, then this function will return false.
- Recursively calls compare list function for the next node of the current nodes.

Find Length

Example 7.20: Find the length of given linked list.

```
func (list *List) FindLength() int {
    curr := list.head
    count := 0
    for curr != nil {
        count++
        curr = curr.next
    }
    return count
}
```

Analysis: Length of linked list is found by traversing the list until we reach the end of list.

Nth Node from Beginning

Example 7.21: : Find Nth node from beginning

```
func (list *List) NthNodeFromBegining(index int) (int, bool) {
	if index > list.Size() || index < 1 {
		fmt.Println("TooFewNodes")
		return 0, false
	}
	count := 0
	curr := list.head
	for curr != nil && count < index-1 {
		count++
		curr = curr.next
	}
	return curr.value, true
}
```

Analysis: Nth node can be found by traversing the list N-1 number of time and then return the node. If list does not have N elements then the method return null.

Nth Node from End

Example 7.22: Find Nth node from end

```
func (list *List) NthNodeFromEnd(index int) (int, bool) {
	size := list.findLength()
	if size != 0 && size < index {
		fmt.Println("TooFewNodes")
		return 0, false
	}
	startIndex := size - index + 1
	return list.NthNodeFromBegining(startIndex)
}
```

Analysis: First, find the length of list, then nth node from end will be (length – nth +1) node from the beginning.

Example 7.23:

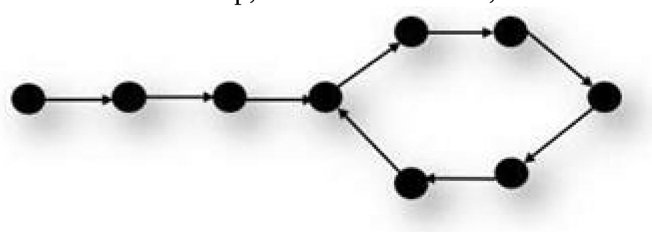
```
func (list *List) NthNodeFromEnd2(index int) (int, bool) {
	count := 1
	forward := list.head
	curr := list.head
	for forward != nil && count <= index {
		count++
		forward = forward.next
	}

	if forward == nil {
		fmt.Println("TooFewNodes")
		return 0, false
	}

	for forward != nil {
		forward = forward.next
		curr = curr.next
	}
	return curr.value, true
```

```
}
```

Analysis: Second approach is to use two references one is N steps / nodes ahead of the other when forward reference reach the end of the list then the backward reference will point to the desired node.

Loop Detect

Find if there is a loop in a linked list. If there is a loop, then return 1 if not, then return 0.



There are many ways to find if there is a loop in a linked list:

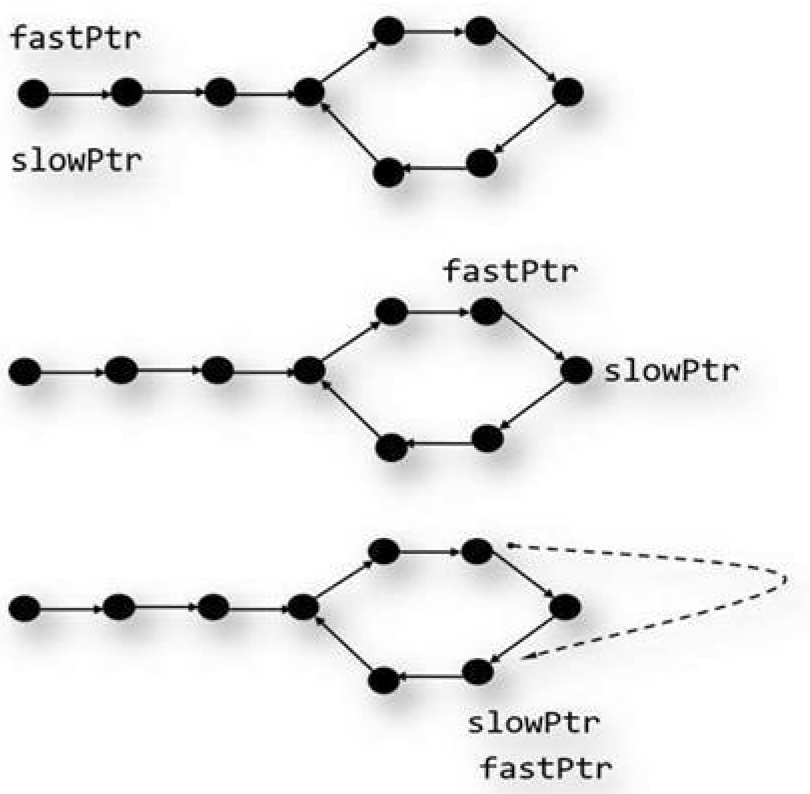
Approach 1: User some map or hash-table

a) Traverse through the list.
b) If the current node is, not there in the Hash-Table then insert it into the Hash-Table.
c) If the current node is already in the Hashtable then we have a loop.

Approach 2: Slow reference and fast reference approach (SPFP)
Approach 3: Reverse list approach"

Slow reference and fast reference approach (SPFP)

We will use two references, one will move 2 steps at a time and another will move 1 step at time. If there is, a loop then both will meet at a point.



Example 7.24:

```
func (list *List) LoopDetect() bool {
	slowPtr := list.head
	fastPtr := list.head

	for fastPtr.next != nil && fastPtr.next.next != nil {
		slowPtr = slowPtr.next
		fastPtr = fastPtr.next.next
		if slowPtr == fastPtr {
			fmt.Println("loop found")
			return true
		}
	}
	fmt.Println("loop not found")
	return false
}
```

Analysis:

· The list is traversed with two references, one is slow reference and another is fast reference. Slow reference always moves one-step. Fast reference always moves two steps. If there is no loop, then control will come out of for loop. So return false.

· If there is a loop, then there came a point in a loop where the fast reference will come and try to pass slow reference and they will meet at a point. When this point arrives, we come to know that there is a loop in the list. So return true.

Reverse List Loop Detect

If there is a loop in a linked list, then reverse list function will give head of the original list as the head of the new list.

Example 7.25: Find if there is a loop in a linked list. Use reverse list approach.

```
func (list *List) ReverseListLoopDetect() bool {
	tempHead := list.head
	list.Reverse()
	if tempHead == list.head {
		list.Reverse()
		fmt.Println("loop found")
		return true
	}
	list.Reverse()
	fmt.Println("loop not found")
	return false
}
```

Analysis:

· Store reference of the head of list in a temp variable.

· Reverse the list

· Compare the reversed list head reference to the current list head reference.

· If the head of reversed list and the original list are same then reverse the list back and return true.

· If the head of the reversed list and the original list are not same, then reverse the list back and return false. Which means there is no loop.

Note: Both SPFP and Reverse List approaches are linear in nature, but still in SPFP approach, we do not require to modify the linked list so it is preferred.

Loop Type Detect

Find if there is a loop in a linked list. If there is no loop, then return 0, if there is loop return 1, if the list is circular then 2. Use slow reference fast reference approach.

Example 7.26:
```
func (list *List) LoopTypeDetect() int {
	slowPtr := list.head
	fastPtr := list.head
	for fastPtr.next != nil && fastPtr.next.next != nil {
		if list.head == fastPtr.next || list.head == fastPtr.next.next {
			fmt.Println("circular list loop found")
			return 2
		}
		slowPtr = slowPtr.next
		fastPtr = fastPtr.next.next
		if slowPtr == fastPtr {
			fmt.Println("loop found")
			return 1
		}
	}
	fmt.Println("loop not found")
	return 0
}
```

Analysis: This program is same as the loop detect program only if it is a circular list than the fast reference reach the slow reference at the head of the list this means that there is a loop at the beginning of the list.

Remove Loop

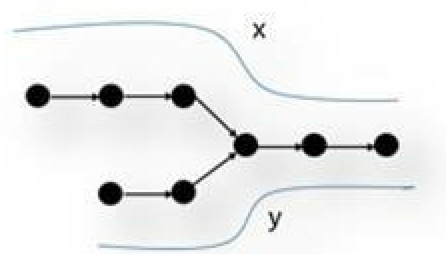
Example 7.27: Given there is a loop in linked list remove the loop.
```
func (list *List) RemoveLoop() {
	loopPoint := list.LoopPointDetect()
	if loopPoint == nil {
		return
	}
	firstPtr := list.head
	if loopPoint == list.head {
		for firstPtr.next != list.head {
			firstPtr = firstPtr.next
		}
		firstPtr.next = nil
		return
	}
	secondPtr := loopPoint
	for firstPtr.next != secondPtr.next {
		firstPtr = firstPtr.next
		secondPtr = secondPtr.next
	}
	secondPtr.next = nil
}
```

```
func (list *List) LoopPointDetect() *Node {
	slowPtr := list.head
	fastPtr := list.head
	for fastPtr.next != nil && fastPtr.next.next != nil {
		slowPtr = slowPtr.next
		fastPtr = fastPtr.next.next
		if slowPtr == fastPtr {
			return slowPtr
		}
	}
	return nil
}
```

Analysis:

- Loop through the list by two reference, one fast reference and one slow reference. Fast reference jumps two nodes at a time and slow reference jump one node at a time. The point where these two reference intersect is a point in the loop.
- If that intersection point is head of the list, this is a circular list case and you need to again traverse through the list and make the node before head point to null.
- In the other case, you need to use two-reference variables one start from head and another start form the intersection-point. They both will meet at the point of loop. (You can mathematically prove it ;))

Find Intersection



Example 7.28: Given two linked list which meet at some point find that intersection point.

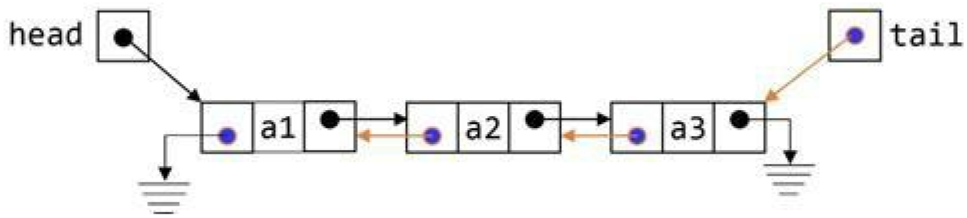
```
func (list *List) FindIntersection(head *Node, head2 *Node) *Node {
	l1 := 0
	l2 := 0
	tempHead := head
	tempHead2 := head2
	for tempHead != nil {
		l1++
		tempHead = tempHead.next
	}
	for tempHead2 != nil {
		l2++
		tempHead2 = tempHead2.next
	}
	var diff int
	if l1 < l2 {
		temp := head
		head = head2
		head2 = temp
```

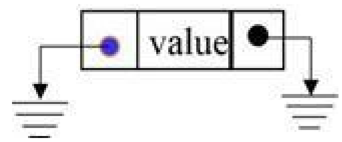
```
        diff = l2 - l1
    } else {
        diff = l1 - l2
    }
    for ; diff > 0; diff-- {
        head = head.next
    }
    for head != head2 {
        head = head.next
        head2 = head2.next
    }
    return head
}
```

Analysis: Find length of both the lists. Find the difference of length of both the lists. Increment the longer list by diff steps, and then increment both the lists and get the intersection point.

Doubly Linked List

In a Doubly Linked list, there are two references in each node. These references are called prev and next. The prev reference of the node will point to the node before it and the next reference will point to the node next to the given node.





Let us look at the Node. The value part of the node is of type integer, but it can be of some other data-type. The two link references are prev and next.

Search in a single linked list can be only done in one direction. Since all elements in the list has reference to the next item in the list. Therefore, traversal of linked list is linear in nature. In a doubly linked list, we keep track of both head of the linked list and tail of linked list.

In doubly linked list linked list below are few cases that we need to keep in mind while coding:

- Zero element case (head and tail both can be modified)
- Only element case (head and tail both can be modified)
- First element (head can be modified)
- General case
- The last element (tail can be modified)

Note: Any program that is likely to change head reference or tail reference is to be passed as a double reference, which is pointing to head or tail reference.

Basic operations of Linked List

Basic operation of a linked list requires traversing a linked list. The various operations that we can perform on linked lists, many of these operations require list traversal:

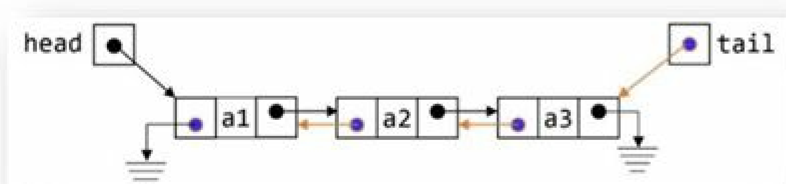
1. Insert an element in the list, this operation is used to create a linked list.
2. Print various elements of the list.
3. Search an element in the list.
4. Delete an element from the list.
5. Reverse a linked list.

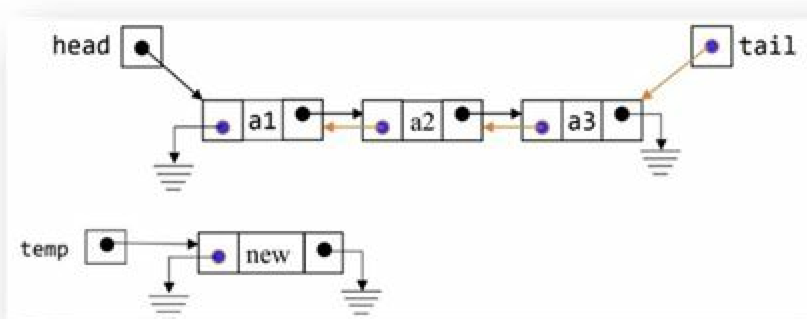
For doubly linked list, we have following cases to consider:

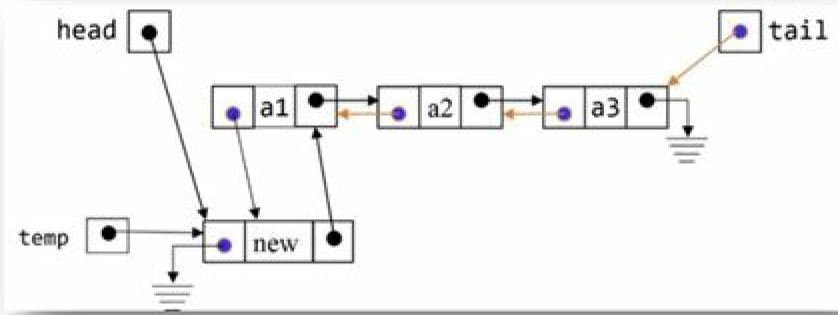
1. null values (head and tail both can be modified)
2. Only element (head and tail both can be modified)
3. First element (head can be modified)
4. General case
5. Last element (tail can be modified)

Example 7.29:
```
type DoublyLinkedList struct {
	head  *Node
	tail  *Node
	count int
}

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

func (list *DoublyLinkedList) Size() int {
	return list.count
}

func (list *DoublyLinkedList) IsEmpty() bool {
	return list.count == 0
}

func (list *DoublyLinkedList) Peek() (int, bool) {
	if list.IsEmpty() {
		fmt.Println("EmptyListError")
	  return 0, false
	}
	return list.head.value, true
}
```

Insert at Head







Example 7.30:

```
func (list *DoublyLinkedList) AddHead(value int) {
    newNode := &Node{value, nil, nil}
    if list.count == 0 {
        list.tail = newNode
        list.head = newNode
    } else {
        list.head.prev = newNode
        newNode.next = list.head
        list.head = newNode
    }
    list.count++
}
```

Analysis: Insert in double linked list is same as insert in a singly linked list.

- Create a node assign null to prev reference of the node.
- If the list is empty then tail and head will point to the new node.
- If the list is not empty then prev of head will point to newNode and next of newNode will point to head. Then head will be modified to point to newNode.

Insert at Tail

Example 7.31: Insert an element at the end of the list.

```
func (list *DoublyLinkedList) AddTail(value int) {
    newNode := &Node{value, nil, nil}
    if list.count == 0 {
        list.head = newNode
        list.tail = newNode
    } else {
        newNode.prev = list.tail
        list.tail.next = newNode
        list.tail = newNode
    }
    list.count++
}
```

Analysis: Find the proper location of the node and add it to the list. Manage next and prev reference of the node so that list always remain double linked list.

Remove Head of doubly linked list

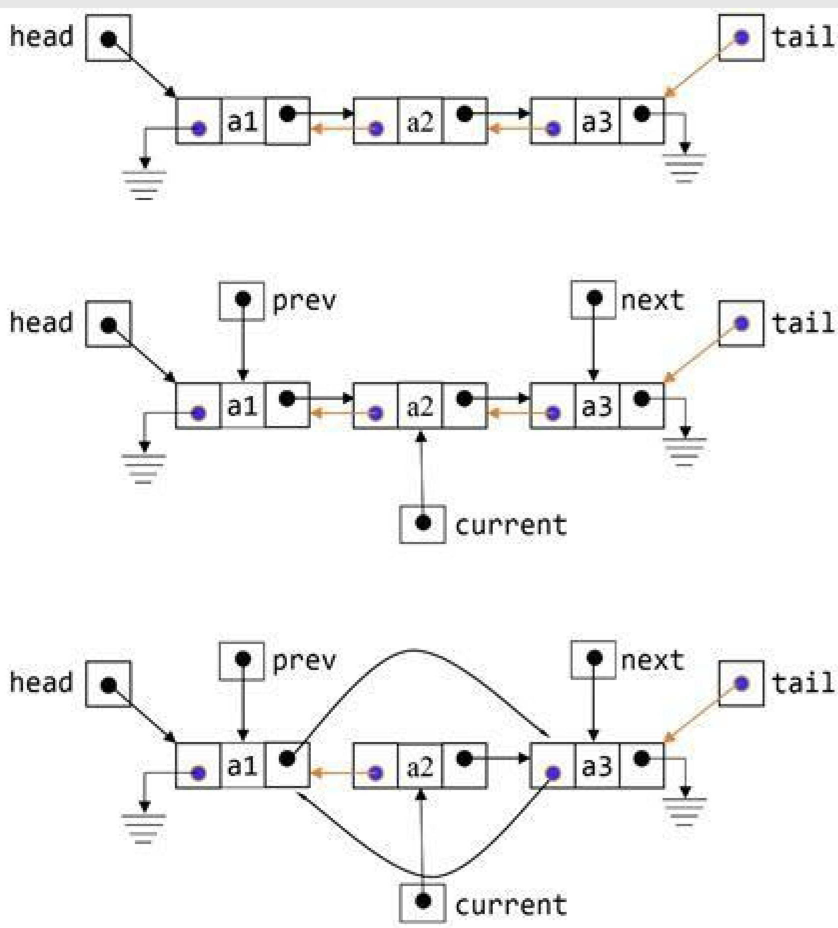
Example 7.32:

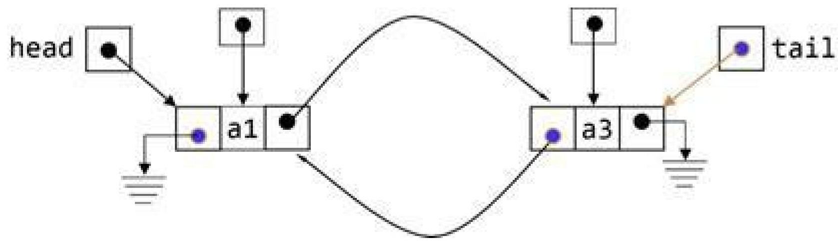
```
func (list *DoublyLinkedList) RemoveHead() (int, bool) {
    if list.IsEmpty() {
        fmt.Println("EmptyListError")
        return 0, false
    }
    value := list.head.value
    list.head = list.head.next
    if list.head == nil {
        list.tail = nil
    } else {
        list.head.prev = nil
    }
    list.count--
    return value, true
}
```

Analysis:

- If the list is empty then EmptyListError message will be printed.
- Now head will point to its next.
- If head is null then this was single node list case, tail also need to be made null.
- In all the general case head. Prev will be set to null.
- Size of list will be reduced by one and value of node is returned.

Delete a node given its value





Example 7.33: Delete node in linked list

```
func (list *DoublyLinkedList) RemoveNode(key int) bool {
    curr := list.head
    if curr == nil { // empty list
        return false
    }
    if curr.value == key { // head is the node with value key.
        curr = curr.next
        list.count--
        if curr != nil {
            list.head = curr
            list.head.prev = nil
        } else {
            list.tail = nil // only one element in list.
        }
        return true
    }
    for curr.next != nil {
        if curr.next.value == key {
            curr.next = curr.next.next
            if curr.next == nil { // last element case.
                list.tail = curr
            } else {
                curr.next.prev = curr
            }
            list.count--
            return true
        }
        curr = curr.next
    }
    return false
}
```

Analysis: Traverse the list find the node, which need to be deleted. Then remove it and adjust next reference of the node before it and prev reference of the node next to it.

Search list

Example 7.34:

```
func (list *DoublyLinkedList) IsPresent(key int) bool {
    temp := list.head
    for temp != nil {
        if temp.value == key {
            return true
        }
```

```
        temp = temp.next
    }
    return false
}
```

Analysis: Traverse the list and find if some value is resent or not.

Free List

Example 7.35:

```
func (list *DoublyLinkedList) FreeList() {
    list.tail = nil
    list.head = nil
    list.count = 0
}
```

Analysis: Just head and tail references need to point to null. The rest of the list will automatically deleted by garbage collection.

Print list

Example 7.36:

```
func (list *DoublyLinkedList) Print() {
    temp := list.head
    for temp != nil {
        fmt.Print(temp.value, " ")
        temp = temp.next
    }
    fmt.Println()
}
```

Analysis: Traverse the list and print the value of each node.

Reverse a doubly linked List iteratively

Example 7.37:

```
func (list *DoublyLinkedList) ReverseList() {
    curr := list.head
    var tempNode *Node
    for curr != nil {
        tempNode = curr.next
        curr.next = curr.prev
        curr.prev = tempNode
        if curr.prev == nil {
            list.tail = list.head
            list.head = curr
            return
        }
        curr = curr.prev
    }
    return
}
```

Analysis: Traverse the list. Swap the next and prev. then traverse to the direction curr.prev, which was next before swap. If you reach the end of the list then set head and tail.

Copy List Reversed

Example 7.38: Copy the content of the list into another list in reverse order.

```
func (list *DoublyLinkedList) CopyListReversed(dll *DoublyLinkedList) {
    curr := list.head
    for curr != nil {
        dll.AddHead(curr.value)
        curr = curr.next
    }
}
```

Analysis:

- Create a DoublyLinkedList class object dll.
- Traverse through the list and copy the value of the nodes into another list by calling addHead() method.
- Since the new nodes are added to the head of the list, the new list formed have nodes order reverse there by making reverse list.

Copy List

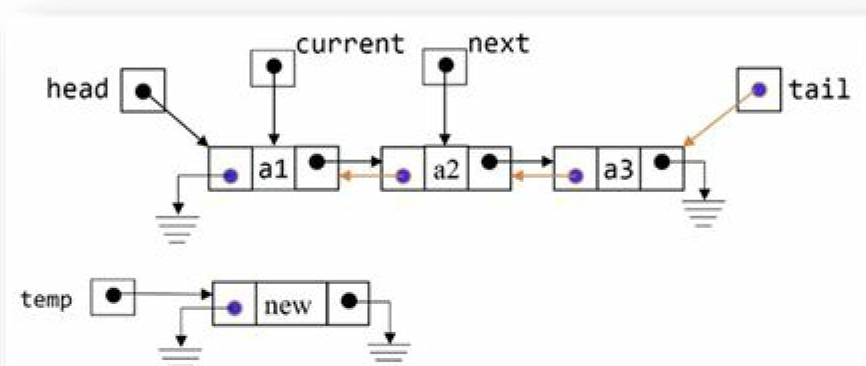
Example 7.39:

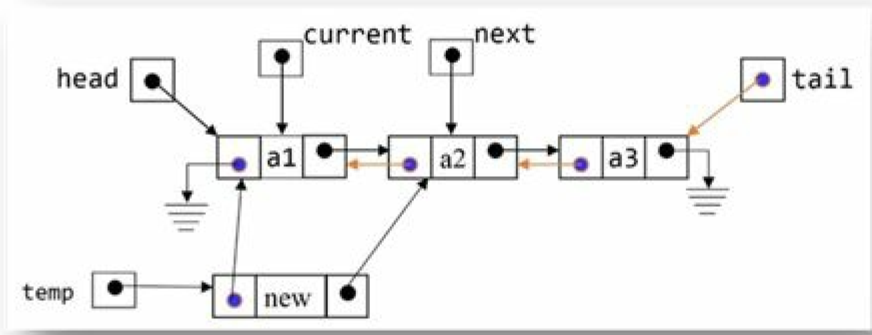
```
func (list *DoublyLinkedList) CopyList(dll *DoublyLinkedList) {
    curr := list.head
    for curr != nil {
        dll.AddTail(curr.value)
        curr = curr.next
    }
}
```

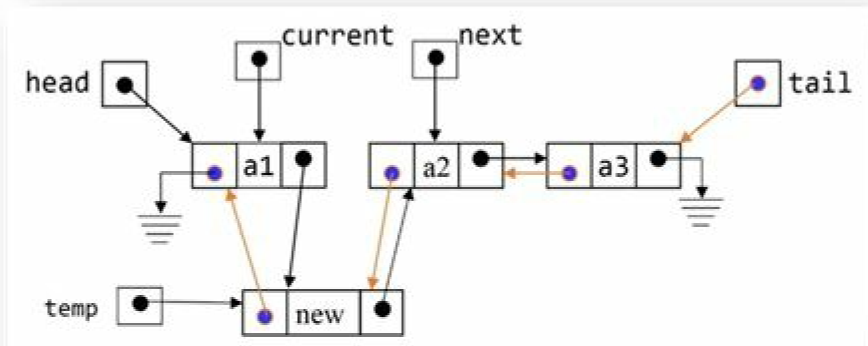
Analysis:

- Create a DoublyLinkedList class object dll.
- Traverse through the list and copy the value of the nodes into another list by calling addTail() method.
- Since the new nodes are added to the tail of the list, the new list formed have nodes order same as the original list.

Sorted Insert







Example 7.40:

```
func (list *DoublyLinkedList) SortedInsert(value int) {
    temp := &Node{value, nil, nil}
    curr := list.head
    if curr == nil { // first element
        list.head = temp
        list.tail = temp
    }

    if list.head.value <= value { // at the begining
        temp.next = list.head
        list.head.prev = temp
        list.head = temp
    }

    for curr.next != nil && curr.next.value > value { // treversal
        curr = curr.next
    }

    if curr.next == nil { // at the end
        list.tail = temp
        temp.prev = curr
        curr.next = temp
    } else { // all other
        temp.next = curr.next
        temp.prev = curr
        curr.next = temp
        temp.next.prev = temp
    }
```

```
}
```

Analysis:

- We need to consider only element case first. In this case, both head and tail will modify.
- Then we need to consider the case when head will be modified when new node is added to the beginning of the list.
- Then we need to consider general cases
- Finally, we need to consider the case when tail will be modified.

Remove Duplicate

Example 7.41: Consider the list as sorted remove the repeated value nodes of the list.

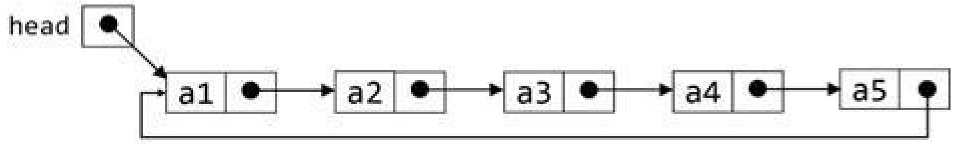
```
#   Remove Duplicate
func (list *DoublyLinkedList) RemoveDuplicate() {
	curr := list.head
	var deleteMe *Node
	for curr != nil {
		if (curr.next != nil) && curr.value == curr.next.value {
			deleteMe = curr.next
			curr.next = deleteMe.next
			curr.next.prev = curr
			if deleteMe == list.tail {
				list.tail = curr
			}
		} else {
			curr = curr.next
		}
	}
}
```

Analysis:

- Remove duplicate is same as single linked list case.
- Head can never modify only the tail can modify when the last node is removed.

Circular Linked List

This type is similar to the singly linked list except that the last element points to the first node of the list. The link portion of the last node contains the address of the first node.

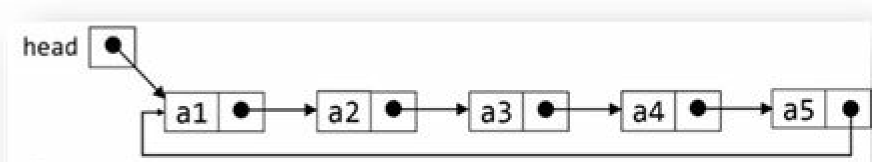


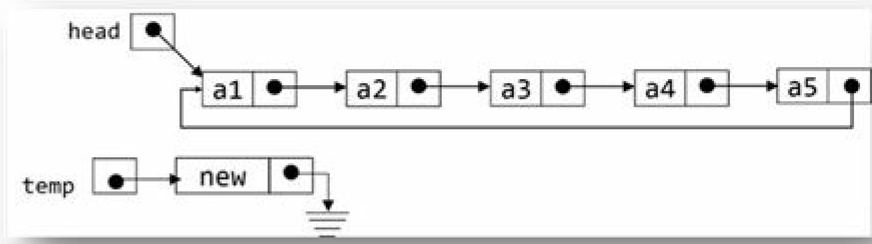
Example 7.42:

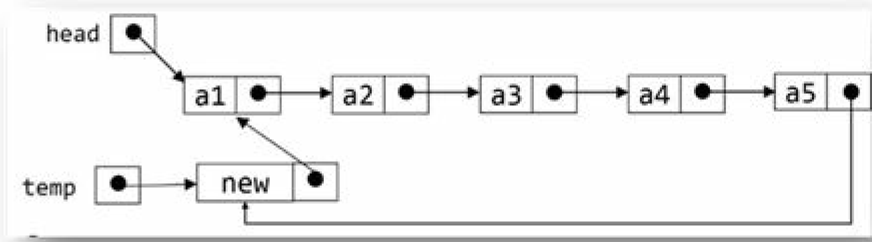
```
type CircularLinkedList struct {
    tail  *Node
    count int
}

type Node struct {
    value int
    next  *Node
}

func (list *CircularLinkedList) Size() int {
    return list.count
}

func (list *CircularLinkedList) IsEmpty() bool {
    return list.count == 0
}

func (list *CircularLinkedList) Peek() (int, bool) {
    if list.IsEmpty() {
        fmt.Println("EmptyListError")
        return 0 , false
    }
    return list.tail.next.value , true
}
```

Analysis: In the circular linked list, we just need the pointer to the tail node. As head node can be easily reached from tail node. Size(), isEmpty() and peek() functions remains the same.

Insert element in front







Example 7.43:
```
func (list *CircularLinkedList) AddHead(value int) {
    temp := &Node{value, nil}
    if list.IsEmpty() {
        list.tail = temp
        temp.next = temp
    } else {
        temp.next = list.tail.next
        list.tail.next = temp
    }
    list.count++
}
```

Analysis:
- First, we create node with given value and its next pointing to null.
- If the list is empty then tail of the list will point to it. In addition, the next of node will point to itself
- If the list is not empty then the next of the new node will be next of the tail. In addition, tail next will start pointing to the new node.
- Thus, the new node is added to the head of the list.
- The demo program create an instance of CircularLinkedList class. Then add some value to it and finally print the content of the list.

Insert element at the end

Example 7.44:
```
func (list *CircularLinkedList) AddTail(value int) {
    temp := &Node{value, nil}
    if list.IsEmpty() {
        list.tail = temp
        temp.next = temp
    } else {
        temp.next = list.tail.next
```

```
        list.tail.next = temp
        list.tail = temp
    }
    list.count++
}
```

Analysis: Adding node at the end is same as adding at the beginning. Just need to modify tail reference in place of the head reference.

Remove element in the front

Example 7.45:

```
func (list *CircularLinkedList) RemoveHead() (int, bool) {
    if list.IsEmpty() {
        fmt.Println("EmptyListError")
        return 0, false
    }
    value := list.tail.next.value

    if list.tail == list.tail.next {
        list.tail = nil
    } else {
        list.tail.next = list.tail.next.next
    }
    list.count--
    return value, true
}
```

Analysis:

· If the list is empty, then false will be returned.
· The value stored in head is stored in local variable value.
· If tail is equal to its next node that means there is only one node in the list so the tail will become null.
· In all the other cases, the next of tail will point to next element of the head.
· Finally, the value is returned.

Search element in the list

Example 7.46:

```
func (list *CircularLinkedList) IsPresent(data int) bool {
    temp := list.tail
    for i := 0; i < list.count; i++ {
        if temp.value == data {
            return true
        }
        temp = temp.next
    }
    return false
}
```

Analysis: Iterate through the list to find if particular value is there or not.

Print the content of list

Example 7.47:
```
func (list *CircularLinkedList) Print() {
    if list.IsEmpty() {
        return
    }
    temp := list.tail.next
    for temp != list.tail {
        fmt.Println(temp.value, " ")
        temp = temp.next
    }
    fmt.Println(temp.value)
}
```

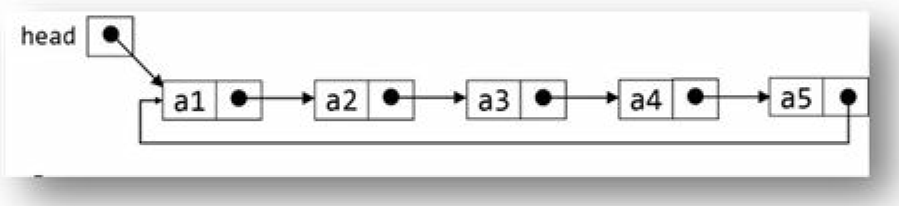
Analysis: In circular list, tail is used to check end of the list.

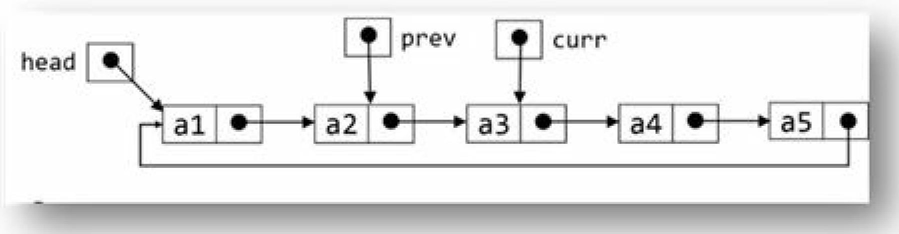
Delete List

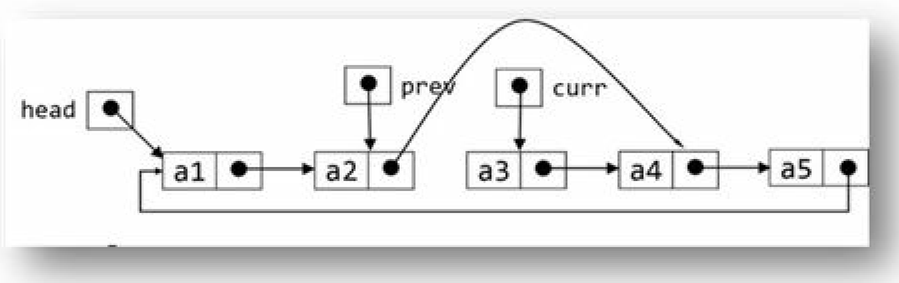
Example 7.48:
```
func (list *CircularLinkedList) FreeList() {
    list.tail = nil
    list.count = 0
}
```

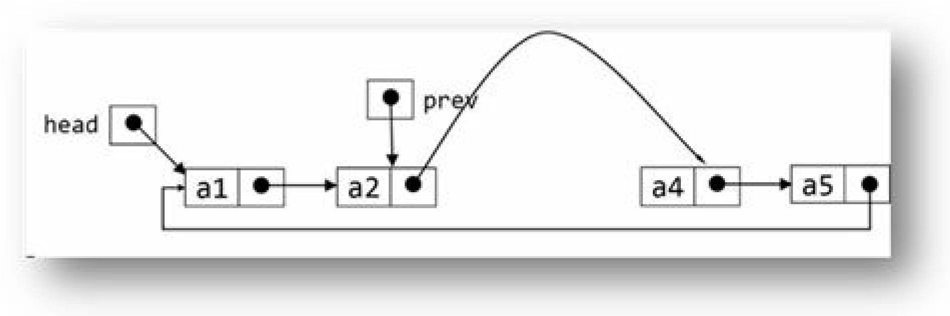
Analysis: The reference to the list is tail. By making tail null, the whole list is deleted.

Delete a node given its value









Example 7.49:
```
func (list *CircularLinkedList) RemoveNode(key int) bool {
	if list.IsEmpty() {
		return false
	}
	prev := list.tail
	curr := list.tail.next
	head := list.tail.next

	if curr.value == key { // head and single node case.
		if curr == curr.next { // single node case
			list.tail = nil
		} else { // head case
			list.tail.next = list.tail.next.next
		}
		return true
	}
	prev = curr
	curr = curr.next

	for curr != head {
		if curr.value == key {
			if curr == list.tail {
				list.tail = prev
			}
			prev.next = curr.next
			return true
		}
		prev = curr
		curr = curr.next
	}
	return false
}
```

Analysis: Find the node that need to free. Only difference is that while traversing the list end of list is tracked by the head reference in place of null.

Copy List Reversed

Example 7.50:
```
func (list *CircularLinkedList) CopyListReversed() *CircularLinkedList {
	cl := new(CircularLinkedList)
	curr := list.tail.next
```

```
    head := curr

    if curr != nil {
        cl.AddHead(curr.value)
        curr = curr.next
    }
    for curr != head {
        cl.AddHead(curr.value)
        curr = curr.next
    }
    return cl
}
```

Analysis: The list is traversed and nodes are added to new list at the beginning. There by making the new list reverse of the given list.

Copy List

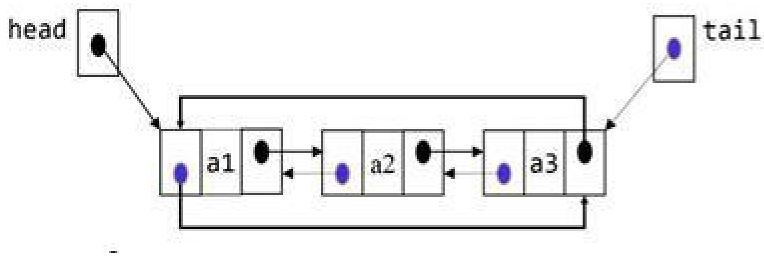
Example 7.51:

```
func (list *CircularLinkedList) CopyList() *CircularLinkedList {
    cl := new(CircularLinkedList)
    curr := list.tail.next
    head := curr
    if curr != nil {
        cl.addTail(curr.value)
        curr = curr.next
    }
    for curr != head {
        cl.addTail(curr.value)
        curr = curr.next
    }
    return cl
}
```

Analysis: List is traversed and nodes are added to the new list at the end. There by making the list whose value are same as the input list.

Doubly Circular list

1. For any linked list there are only three cases zero element, one element, general case
2. To doubly linked list we have a few more things
    a) null values
    b) Only element (it generally introduces an if statement with null)
    c) Always an “if” before “while”. Which will check from this head.
    d) General case (check with the initial head kept)
    e) Avoid using recursion solutions it makes life harder



Example 7.52:

```
type DoublyCircularLinkedList struct {
    head  *Node
    tail  *Node
    count int
}
type Node struct {
    value int
    next  *Node
    prev  *Node
}
func (list *DoublyCircularLinkedList) Size() int {
    return list.count
}
func (list *DoublyCircularLinkedList) IsEmpty() bool {
    return list.count == 0
}
func (list *DoublyCircularLinkedList) peekHead() (int, bool) {
    if list.IsEmpty() {
        fmt.Println("EmptyListError")
        return 0, false
    }
    return list.head.value, true
}
```

Search value

Example 7.53:

```
func (list *DoublyCircularLinkedList) IsPresent(key int) bool {
    temp := list.head
    if list.head == nil {
        return false
    }
    for true {
```

```
        if temp.value == key {
            return true
        }
        temp = temp.next
        if temp == list.head {
            break
        }
    }
    return false
}
```

Analysis: Traverse through the list and see if given key is present or not. We if statement to terminate the loop.

Delete list

Example 7.54:

```
func (list *DoublyCircularLinkedList) FreeList() {
    list.head = nil
    list.tail = nil
    list.count = 0
}
```

Analysis: Remove the reference and list will be freed.

Print List

Example 7.55:

```
func (list *DoublyCircularLinkedList) Print() {
    if list.IsEmpty() {
        return
    }
    fmt.Println("List size is ::", list.count)
    fmt.Print("List content :: ")
    temp := list.head
    for true {
        fmt.Print(temp.value, " ")
        temp = temp.next
        if temp == list.head {
            break
        }
    }
    fmt.Println()
}
```

Analysis: Traverse the list and print its content. For loop with if condition to exit loop is used as we want to terminate when temp is head. Moreover, want to process head node once.

Insert Node at head

Example 7.56: Insert value at the front of the list.

```
func (list *DoublyCircularLinkedList) AddHead(value int) {
    newNode := new(Node)
```

```
    newNode.value = value
    if list.count == 0 {
        list.tail = newNode
        list.head = newNode
        newNode.next = newNode
        newNode.prev = newNode
    } else {
        newNode.next = list.head
        newNode.prev = list.head.prev
        list.head.prev = newNode
        newNode.prev.next = newNode
        list.head = newNode
    }
    list.count++
}
```

Analysis:
- A new node is created and if the list is empty then head and tail will point to it. The newly created newNode's next and prev also point to newNode.
- If the list is not empty then the pointers are adjusted and a new node is added to the front of the list. Only head need to be changed in this case.
- Size of the list is increased by one.

Insert Node at tail

Example 7.57:
```
func (list *DoublyCircularLinkedList) AddTail(value int) {
    newNode := new(Node)
    newNode.value = value
    if list.count == 0 {
        list.head = newNode
        list.tail = newNode
        newNode.next = newNode
        newNode.prev = newNode
    } else {
        newNode.next = list.tail.next
        newNode.prev = list.tail
        list.tail.next = newNode
        newNode.next.prev = newNode
        list.tail = newNode
    }
    list.count++
}
```

Analysis:
- A new node is created and if the list is empty then head and tail will point to it. The newly created newNode's next and prev also point to newNode.
- If the list is not empty then the pointers are adjusted and a new node is added to the end of the list. Only tail need to be changed in this case.
- Size of the list is increased by one.

Delete head node

Example 7.58:
```
func (list *DoublyCircularLinkedList) RemoveHead() (int, bool) {
	if list.count == 0 {
		fmt.Println("EmptyListError")
		return 0, false
	}

	value := list.head.value
	list.count--

	if list.count == 0 {
		list.head = nil
		list.tail = nil
		return value, true
	}
	next := list.head.next
	next.prev = list.tail
	list.tail.next = next
	list.head = next
	return value, true
}
```

Analysis: Delete node in a doubly circular linked list is just same as delete node in a circular linked list. Just few extra next reference need to be adjusted.

Delete tail node

Example 7.59:
```
func (list *DoublyCircularLinkedList) RemoveTail() (int, bool) {
	if list.count == 0 {
		fmt.Println("EmptyListError")
		return 0, false
	}

	value := list.tail.value
	list.count--
	if list.count == 0 {
		list.head = nil
		list.tail = nil
		return value, true
	}

	prev := list.tail.prev
	prev.next = list.head
	list.head.prev = prev
	list.tail = prev
	return value, true
}
```

Analysis: Delete node in a doubly circular linked list is just same as delete node in a circular linked list. Just few extra prev reference need to be adjusted.

Exercise

1. Insert an element at k^{th} position from the start of linked list. Return 1 if success and if list is not long enough, then return -1.
   Hint: Take a reference advance it K steps forward, and then inserts the node.

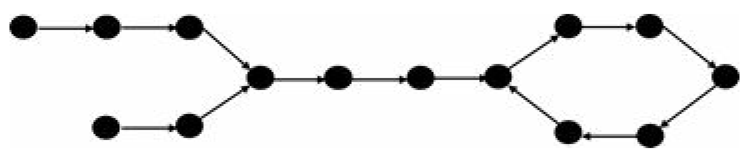
2. Insert an element at k^{th} position from the end of linked list. Return 1 if success and if list is not long enough, then return -1.
   Hint: Take a reference advance it K steps forward, then take another reference and advance both of them simultaneously, so that when the first reference reach the end of a linked list that is the point where you need to insert the node.

3. Consider there is a loop in a linked list, Write a program to remove loop if there is a loop in this linked list.

4. In the above SearchList program return, the count of how many instances of same value found else if value not found then return 0. For example, if the value passed is “4”. The elements in the list are 1,2,4,3 & 4. The program should return 2.

   Hint: In place of return 1 in the above program increment a counter and then return counter at the end.

5. Given two linked list head pointer and they meet at some point and need to find the point of intersection. However, in place of the end of both the linked list to be a null pointer there is a loop.



6. If linked list having a loop is given. Count the number of nodes in the linked list

7. We were supposed to write the complete code for the addition of polynomials using Linked Lists. This takes time if you do not have it by heart, so revise it well.

8. Given two linked lists. We have to find that whether the data in one is reverse that of data in another. No extra space should be used and traverse the linked lists only once.

9. Find the middle element in a singly linked list. Tell the complexity of your solution.
   Hint:-
   Approach 1: Find the length of linked list. Then find the middle element and return it.
   Approach 2: Use two reference one will move fast and one will move slow make sure you handle border case properly. (Even length and odd length linked list cases.)

10. Print list in reverse order.
    Hint: Use recursion.

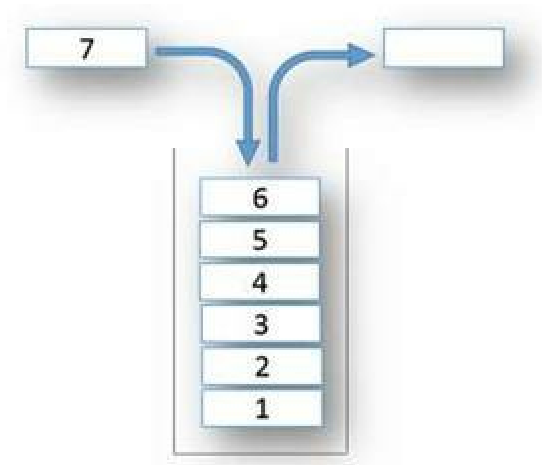
CHAPTER 8: STACK

Introduction

A stack is a basic data structure that organized items in last-in-first-out (LIFO) manner. Last element inserted in a stack will be the first to be removed from it.

The real-life analogy of the stack is "stack of plates". Imagine a stack of plates in a dining area everybody takes a plate at the top of the stack, thereby uncovering the next plate for the next person.

Stack allow to only access the top element. The elements that are at the bottom of the stack are the one that is going to stay in the stack for the longest time.



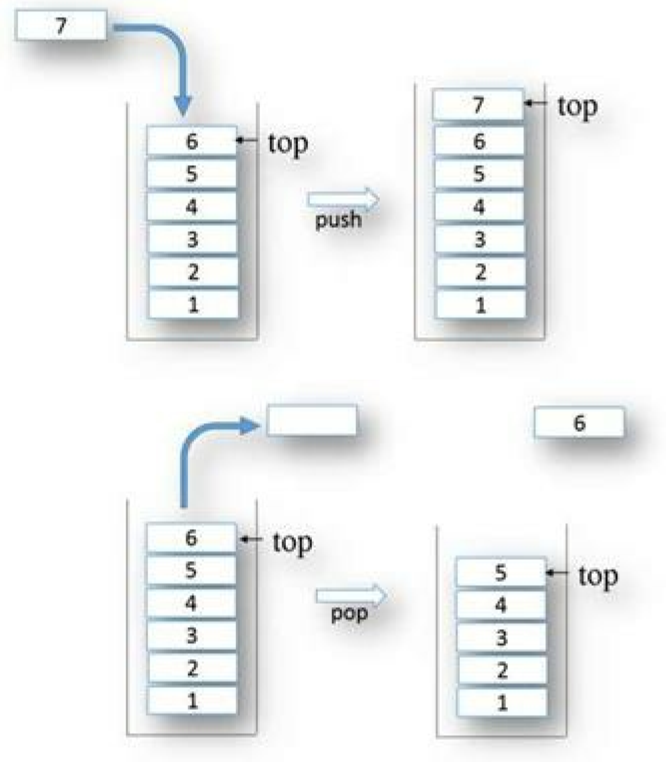
Computer science also has the common example of a stack. Function call stack is a good example of a stack. Function main() calls function foo() and then foo() calls bar(). These function calls are implemented using stack first bar() exists, then foo() and then finally main().

As we navigate from web page to web page, the URL of web pages are kept in a stack, with the current page URL at the top. If we click back button, then each URL entry is popped one by one.

The Stack Abstract Data Type

Stack abstract data type is defined as a class, which follows LIFO or last-in-first-out for the elements, added to it.
The stack should support the following operation:

1. Push(): which add a single element at the top of the stack
2. Pop(): which remove a single element from the top of a stack.
3. Top(): Reads the value of the top element of the stack (does not remove it)
4. isEmpty(): Returns 1 if stack is empty
5. Size(): returns the number of elements in a stack.



Add n to the top of a stack

```
void push(int  value);.
```

Remove the top element of the stack and return it to the caller function.

```
int  pop();
```

The stack can be implemented using a list or a linked list.

1. When stack is implemented using list, it takes care of its size automatically.
2. In a linked list, there is no limit on the number of elements

When a stack is implemented, using a list top of the stack is managed using an index variable called top.
When a stack is implemented using a linked list, push() and pop() is implemented using insert at the head of the linked list and remove from the head of the linked list.

Stack using slices

Implement a stack using slice.

Example 8.1:
```
type StackInt struct {
    s []int
}
```

If user does not provide the max capacity of the list. Then a list of 1000 elements is created.
The top is the index to the top of the stack.
Number of elements in the stack is governed by the “top” index and top is initialized to -1 when a stack is initialized. Top index value of -1 indicates that the stack is empty in the beginning.

```
func (s *StackInt) IsEmpty() bool {
    length := len(s.s)
    return length == 0
}
```

isEmpty() function returns 1 if stack is empty or 0 in all other cases. By comparing the top index value with -1.

```
func (s *StackInt) Length() int {
    length := len(s.s)
    return length
}
```

size() function returns the number of elements in the stack. It just returns "top+1". As the top is referring the list index of the stack top variable so we need to add one to it.

```
func (s *StackInt) Print() {
    length := len(s.s)
    for i := 0; i < length; i++ {
        fmt.Print(s.s[i], " ")
    }
    fmt.Println()
}
```

The print function will print the elements of the list.

```
func (s *StackInt) Push(value int) {
    s.s = append(s.s, value)
}
```

push() function checks whether the stack has enough space to store one more element, then it increases the "top" by one. Finally sort the data in the stack "data" list. In case, stack is full then "stack overflow" message is printed and that value will not be added to the stack and will be ignored.

```
func (s *StackInt) Pop() int {
    length := len(s.s)
    res := s.s[length-1]
    s.s = s.s[:length-1]
    return res
```

```
}
```

In the pop() function, first it will check that there are some elements in the stack by checking its top index. If some element is there in the stack, then it will store the top most element value in a variable "value". The top index is reduced by one. Finally, that value is returned.

```
func (s *StackInt) Top() int {
	length := len(s.s)
	res := s.s[length-1]
	return res
}
```

top() function returns the value of stored in the top element of stack (does not remove it)

```
func main() {
	s := new(StackInt)
	length := 10
	for i := 0; i < length; i++ {
		s.Push(i)
	}
	fmt.Println(s.Length())
	for i := 0; i < length; i++ {
		fmt.Print(s.Pop(), " ")
	}
	fmt.Println()
}
```

Analysis:

- The user of the stack will create a stack local variable.
- Use push() and pop() functions to add / remove variables to the stack.
- Read the top element using the top() function call.
- Query regarding size of the stack using size() function call
- Query if stack is empty using isEmpty() function call

Stack generic implementation

Example 8.2:

```
type Stack struct {
	s []interface{}
}

func (s *Stack) Push(value interface{}) {
	s.s = append(s.s, value)
}

func (s *Stack) Pop() interface{} {
	if s.IsEmpty() {
		return nil
	}
	length := len(s.s)
	res := s.s[length-1]
	s.s = s.s[:length-1]
	return res
}

func (s *Stack) Top() interface{} {
	length := len(s.s)
	res := s.s[length-1]
	return res
}
```

Stack using linked list

Example 8.3: Implement stack using a linked list.

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

type Stack struct {
	head *Node
	size int
}

func (s *Stack) Size() int {
	return s.size
}

func (s *Stack) IsEmpty() bool {
	return s.size == 0
}

func (s *Stack) Peek() (int, bool) {
	if s.IsEmpty() {
		fmt.Println("StackEmptyError")
		return 0, false
	}
	return s.head.value, true
}

func (s *Stack) Push(value int) {
	s.head = &Node{value, s.head}
	s.size++
}

func (s *Stack) Pop() (int, bool) {
	if s.IsEmpty() {
		fmt.Println("StackEmptyError")
		return 0, false
	}
	value := s.head.value
	s.head = s.head.next
	s.size--
	return value, true
}

func (s *Stack) Print() {
	temp := s.head
	fmt.Print("Value stored in stck are :: ")
	for temp != nil {
		fmt.Print(temp.value, " ")
		temp = temp.next
	}
```

```
    fmt.Println()
}
```

Analysis:

- Stack implemented using a linked list is simply insertion and deletion at the head of a singly linked list.
- In push() function, memory is created for one node. Then the value is stored into that node. Finally, the node is inserted at the beginning of the list.
- In pop() function, the head of the linked list starts pointing to the second node there by releasing the memory allocated to the first node (Garbage collection).

Problems in Stack

Balanced Parenthesis

Example 8.4: Stacks can be used to check a program for balanced symbols (such as {}, (), []). The closing symbol should be matched with the most recently seen opening symbol.
Example: {()} is legal, {() ({})} is legal, but {((} and {(}) are not legal

```
func IsBalancedParenthesis(expn string) bool {
	stk := new(Stack)
	for _, ch := range expn {
		switch ch {
		case '{', '[', '(':
			stk.Push(ch)
		case '}':
			val := stk.Pop()
			if val != '{' {
				return false
			}
		case ']':
			val := stk.Pop()
			if val != '[' {
				return false
			}
		case ')':
			val := stk.Pop()
			if val != '(' {
				return false
			}
		}
	}
	return stk.IsEmpty()
}

func main() {
	expn := "{()}[]"
	value := IsBalancedParenthesis(expn)
	fmt.Println("Given Expn:", expn)
	fmt.Println("Result aft1er isParenthesisMatched:", value)
}
```

Analysis:

- Traverse the input string when we get an opening parenthesis we push it into stack. Moreover, when we get a closing parenthesis then we pop a parenthesis from the stack and compare if it is the corresponding to the one on the closing parenthesis.
- We return false if there is a mismatch of parenthesis.
- If at the end of the whole staring traversal, we reached to the end of the string and the stack is empty then we have balanced parenthesis.

Infix, Prefix and Postfix Expressions

When we have an algebraic expression like A + B then we know that the variable is being added to variable B. This type of expression is called **infix** expression because the operator "+" is between operands A and operand B.

Now consider another infix expression A + B * C. In the expression there is a problem that in which order + and * works. Does A and B are added first and then the result is multiplied. Alternatively, B and C are multiplied first and then the result is added to A. This makes the expression ambiguous. To deal with this ambiguity we define the precedence rule or use parentheses to remove ambiguity.

So if we want to multiply B and C first and then add the result to A. Then the same expression can be written unambiguously using parentheses as A + (B * C). On the other hand, if we want to add A and B first and then the sum will be multiplied by C we will write it as (A + B) * C. Therefore, in the infix expression to make the expression unambiguous, we need parenthesis.

Infix expression: In this notation, we place operator in the middle of the operands.
< Operand > < operator > < operand >

Prefix expressions: In this notation, we place operator at the beginning of the operands.
< Operator > < operand > < operand >

Postfix expression: In this notation, we place operator at the end of the operands.
< Operand > < operand > < operator >

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + (B * C)	+ A * B C	A B C * +
(A + B) * C	* + ABC	A B + C *

Now comes the most obvious question why we need so unnatural Prefix or Postfix expressions when we already have infix expressions which words just fine for us. The answer to this is that infix expressions are ambiguous and they need parenthesis to make them unambiguous. While postfix and prefix notations do not need any parenthesis.

Infix-to-Postfix Conversion

Example 8.5:
```
func InfixToPostfix(expn string) string {
	fmt.Println(expn)
	stk := new(Stack)
	output := ""

	for _, ch := range expn {
		if ch <= '9' && ch >= '0' {
			output = output + string(ch)
		} else {
			switch ch {
			case '+', '-', '*', '/', '%', '^':
				for stk.IsEmpty() == false && precedence(ch) <= precedence(stk.Top().(rune)) {
					out := stk.Pop().(rune)
					output = output + " " + string(out)
				}
```

```
                stk.Push(ch)
                output = output + " "
            case '(':
                stk.Push(ch)
            case ')':
                for stk.IsEmpty() == false && stk.Top() != '(' {
                    out := stk.Pop().(rune)
                    output = output + " " + string(out) + " "
                }
                if stk.IsEmpty() == false && stk.Top() == '(' {
                    stk.Pop()
                }
            }
        }
    }
    for stk.IsEmpty() == false {
        out := stk.Pop().(rune)
        output = output + string(out) + " "
    }
    return output
}

func precedence(x rune) int {
    if x == '(' {
        return (0)
    }
    if x == '+' || x == '-' {
        return (1)
    }
    if x == '*' || x == '/' || x == '%' {
        return (2)
    }
    if x == '^' {
        return (3)
    }
    return (4)
}

func main() {
    expn := "10+((3))*5/(16-4)"
    value := InfixToPostfix(expn)
    fmt.Println("Infix Expn: ", expn)
    fmt.Println("Postfix Expn: ", value)
}
```

Analysis:

- Print operands in the same order as they arrive.
- If the stack is empty or contains a left parenthesis "(" on top, we should push the incoming operator in the stack.
- If the incoming symbol is a left parenthesis "(", push left parenthesis in the stack.
- If the incoming symbol is a right parenthesis ")", pop from the stack and print the operators until you see a left parenthesis ")". Discard the pair of parentheses.
- If the precedence of incoming symbol is higher than the precedence of operator at the top of the stack, then push it to the stack.

· If the incoming symbol has, an equal precedence compared to the top of the stack, use association. If the association is left to right, then pop and print the symbol at the top of the stack and then push the incoming operator. If the association is right to left, then push the incoming operator.
· If the precedence of incoming symbol is lower than the precedence of operator on the top of the stack, then pop and print the top operator. Then compare the incoming operator against the new operator at the top of the stack.
· At the end of the expression, pop and print all operators on the stack.

Infix-to-Prefix Conversion

Example 8.6:

```
func InfixToPrefix(expn string) string {
	expn = reverseString(expn)
	expn = replaceParanthesis(expn)
	expn = InfixToPostfix(expn)
	expn = reverseString(expn)
	return expn
}

func reverseString(in string) string {
	expn := []rune(in)
	lower := 0
	upper := len(expn) - 1
	for lower < upper {
		expn[lower], expn[upper] = expn[upper], expn[lower]
		lower++
		upper--
	}
	return string(expn)
}

func replaceParanthesis(str string) string {
	a := []rune(str)
	lower := 0
	upper := len(a) - 1
	for lower <= upper {
		if a[lower] == '(' {
			a[lower] = ')'
		} else if a[lower] == ')' {
			a[lower] = '('
		}
		lower++
	}
	return string(a)
}

func main() {
	expn := "10+((3))*5/(16-4)"
	value := InfixToPrefix(expn)
	fmt.Println("Infix Expn: ", expn)
	fmt.Println("Prefix Expn: ", value)
}
```

Analysis:
1. Reverse the given infix expression.
2. Replace '(' with ')' and ')' with '(' in the reversed expression.
3. Now, apply infix to postfix subroutine already discussed.
4. Reverse the generated postfix expression and this will give required prefix expression.

Postfix Evaluate

Write a postfixEvaluate() function to evaluate a postfix expression. Such as: 1 2 + 3 4 + *
Example 8.7:
```
func postfixEvaluate(expn string) int {
	stk := new(Stack)
	str := strings.Split(expn, " ")
	for _, tkn := range str {
		value, err := strconv.Atoi(tkn)
		if err == nil {
			stk.Push(value)
		} else {
			num1 := stk.Pop().(int)
			num2 := stk.Pop().(int)
			switch tkn {
			case "+":
				stk.Push(num1 + num2)
			case "-":
				stk.Push(num1 - num2)
			case "*":
				stk.Push(num1 * num2)
			case "/":
				stk.Push(num1 / num2)
			}
		}
	}
	return stk.Pop().(int)
}
func main() {
	expn := "6 5 2 3 + 8 * + 3 + *"
	value := postfixEvaluate(expn)
	fmt.Println("Given Postfix Expn: ", expn)
	fmt.Println("Result after Evaluation: ", value)
}
```

Analysis:
1) Create a stack to store values or operands.
2) Scan through the given expression and do following for each element:
   a) If the element is a number, then push it into the stack.
   b) If the element is an operator, then pop values from the stack. Evaluate the operator over the values and push the result into the stack.
3) When the expression is scanned completely, the number in the stack is the result.

Min stack

Design a stack in which get minimum value in stack should also work in **O(1)** Time Complexity.

Hint: Keep two stack one will be general stack, which will just keep the elements. The second will keep the min value.

1. Push: Push an element to the top of stack1. Compare the new value with the value at the top of the stack2. If the new value is smaller, then push the new value into stack2. Or push the value at the top of the stack2 to itself once more.
2. Pop: Pop an element from top of stack1 and return. Pop an element from top of stack2 too.
3. Min: Read from the top of the stack2, this value will be the min.

Palindrome string

Find if given string is a palindrome or not using a stack.
Definition of palindrome: A palindrome is a sequence of characters that is same backward or forward.
Eg. “AAABBBCCCBBBAAA”, “ABA” & “ABBA”

Hint: Push characters to the stack until the half-length of the string. Then pop these characters and then compare. Make sure you take care of the odd length and even length.

Depth-First Search with a Stack

In a depth-first search, we traverse down a path until we get a dead end; then we backtrack by popping a stack to get an alternative path.

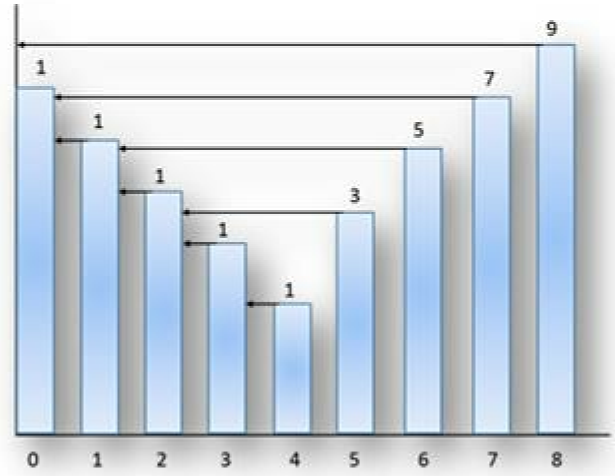
- Create a stack
- Create a start point
- Push the start point onto the stack
- While (value searching not found and the stack is not empty)
    - Pop the stack
    - Find all possible points after the one which we just tried
    - Push these points onto the stack

Stack using a queue

How to implement a stack using a queue. Analyze the running time of the stack operations.
See queue chapter for this.

Stock Span Problem

Given a list of daily stock price in a list A[i]. Find the span of the stocks for each day. A span of stock is the maximum number of days for which the price of stock was lower than that day.



Example 8.8: Approach 1

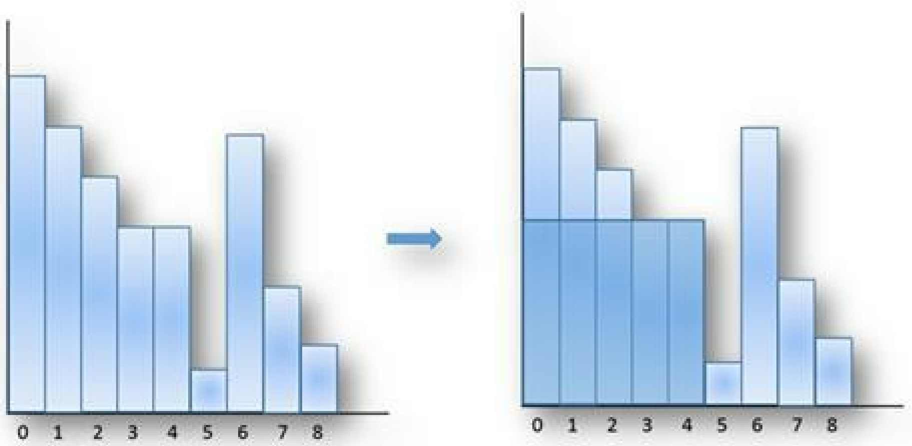
```
func StockSpanRange(arr []int) []int {
	n := len(arr)
	SR := make([]int, n)
	SR[0] = 1
	for i := 1; i < len(arr); i++ {
		SR[i] = 1
		for j := i - 1; (j >= 0) && (arr[i] >= arr[j]); j-- {
			SR[i]++
		}
	}
	return SR
}
```

Example 8.9: Approach 2

```
func StockSpanRange2(arr []int) []int {
	stk := new(Stack)
	n := len(arr)
	SR := make([]int, n)
	stk.Push(0)
	SR[0] = 1
	for i := 1; i < len(arr); i++ {
		for !stk.IsEmpty() && arr[stk.Top().(int)] <= arr[i] {
			stk.Pop()
		}
		if stk.IsEmpty() {
			SR[i] = (i + 1)
		} else {
			SR[i] = (i - stk.Top().(int))
		}
		stk.Push(i)
	}
	return SR
}
```

Get Max Rectangular Area in a Histogram

Given a histogram of rectangle bars of each one unit wide. Find the maximum area rectangle in the histogram.



Example 8.10: Approach 1

```
func GetMaxArea(arr []int) int {
```

```
    size := len(arr)
    maxArea := -1
    currArea := 0
    minHeight := 0
    for i := 1; i < size; i++ {
        minHeight = arr[i]
        for j := i - 1; j >= 0; j-- {
            if minHeight > arr[j] {
                minHeight = arr[j]
            }
            currArea = minHeight * (i - j + 1)
            if maxArea < currArea {
                maxArea = currArea
            }
        }
    }
    return maxArea
}
```

Approach 2: Divide and conquer

Example 8.11: Approach 3

```
func GetMaxArea2(arr []int) int {
    size := len(arr)
    stk := new(Stack)
    maxArea := 0
    Top := 0
    TopArea := 0
    i := 0
    for i < size {
        for (i < size) && (stk.IsEmpty() || arr[stk.Top().(int)] <= arr[i]) {
            stk.Push(i)
            i++
        }
        for !stk.IsEmpty() && (i == size || arr[stk.Top().(int)] > arr[i]) {
            Top = stk.Top().(int)
            stk.Pop()
            if stk.IsEmpty() {
                TopArea = arr[Top] * i
            } else {
                TopArea = arr[Top] * (i - stk.Top().(int) - 1)
            }

            if maxArea < TopArea {
                maxArea = TopArea
            }
        }
    }
    return maxArea
}
```

Uses of Stack

- Recursion can also be done using stack. (In place of the system stack)
- The function call is implemented using stack.
- Some problems when we want to reverse a sequence, we just push everything in stack and pop from it.
- Grammar checking, balance parenthesis, infix to postfix conversion, postfix evaluation of expression etc.

Exercise

1. Converting Decimal Numbers to Binary Numbers using stack data structure.
   Hint: store reminders into the stack and then print the stack.

2. Convert an infix expression to prefix expression.

3. Write an HTML opening tag and closing tag-matching program.
   Hint: parenthesis matching.

4. Write a function that will do Postfix to Infix Conversion

5. Write a function that will do Prefix to Infix Conversion

6. Write a palindrome matching function, which ignore characters other than English alphabet and digits. String "Madam, I'm Adam." should return true.

7. In the Growing-Reducing Stack implementation using list. Try to figure out a better algorithm which will work similar to Vector<> or ArrayDeque<>.

CHAPTER 9: QUEUE

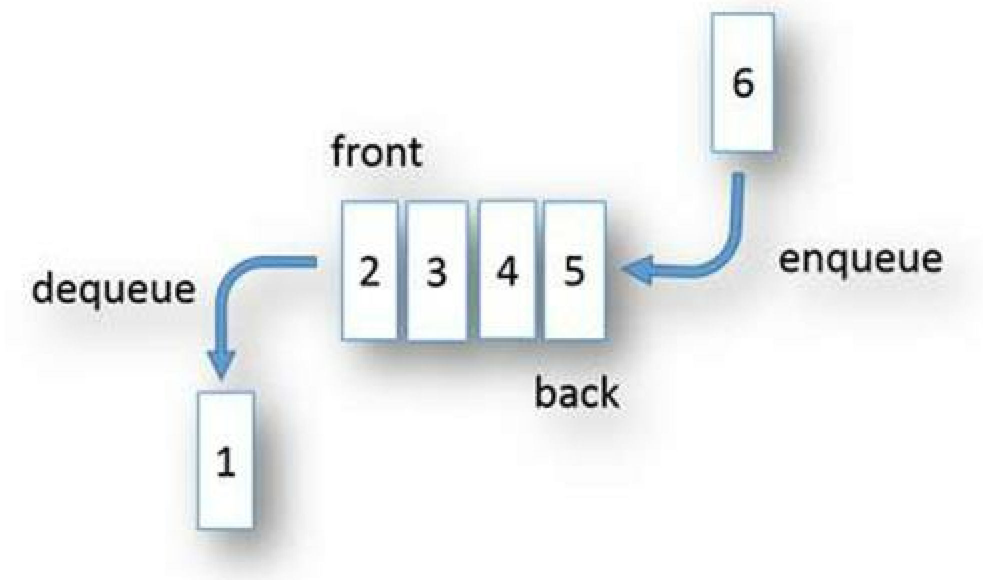
Introduction

A queue is a basic data structure that organized items in first-in-first-out (FIFO) manner. First element inserted into a queue will be the first to be removed. It is also known as "first-come-first-served".

The real life analogy of queue is typical lines in which we all participate time to time.

- We wait in a line of railway reservation counter.
- We wait in the cafeteria line.
- We wait in a queue when we call to some customer case.

The elements, which are at the front of the queue, are the one that stayed in the queue for the longest time.



Computer science also has common examples of queues. We issue a print command from our office to a single printer per floor. The print task are lined up in a printer queue. The print command that was issued first will be printed before the next commands in line.

In addition to printing queues, operating system is also using different queues to control process scheduling. Processes are added to processing queue, which is used by an operating system for various scheduling algorithms.

Soon we will be reading about graphs and will come to know about breadth-first traversal, which uses a queue.

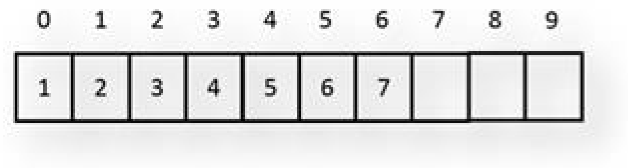
The Queue Abstract Data Type

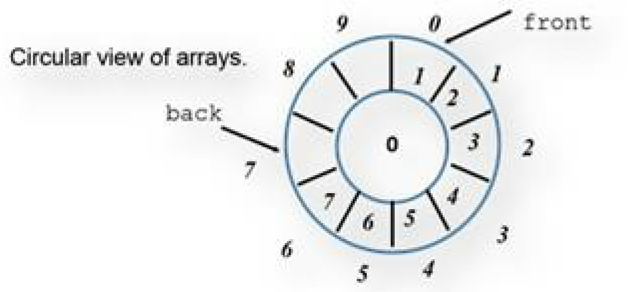
Queue abstract data type is defined as a class whose object follows FIFO or first-in-first-out for the elements, added to it.

Queue should support the following operation:

1. add(): Which add a single element at the back of a queue
2. remove(): Which remove a single element from the front of a queue.
3. isEmpty(): Returns 1 if the queue is empty
4. size(): Returns the number of elements in a queue.

Queue Using List





Example 9.1:

```
const capacity = 100

type Queue struct {
	size  int
	data  [capacity]int
	front int
	back  int
}

func (q *Queue) Add(value int) bool {
	if q.size >= capacity {
		return false
	}
	q.size++
	q.data[q.back] = value
	q.back = (q.back + 1) % (capacity - 1)
	return true
}

func (q *Queue) Remove() (int, bool) {
	var value int
	if q.size <= 0 {
		return 0, false
	}
	q.size--
	value = q.data[q.front]
	q.front = (q.front + 1) % (capacity - 1)
	return value, true
}

func (q *Queue) IsEmpty() bool {
	return q.size == 0
}
```

```
func (q *Queue) Size() int {
	return q.size
}
```

Analysis:

- Hear queue is created from a list of size 100.
- The number of element in queue to zero. By assigning front, back and size of queue to zero.
- Add() insert one element at the back of the queue.
- Remove() delete one element from the front of the queue.

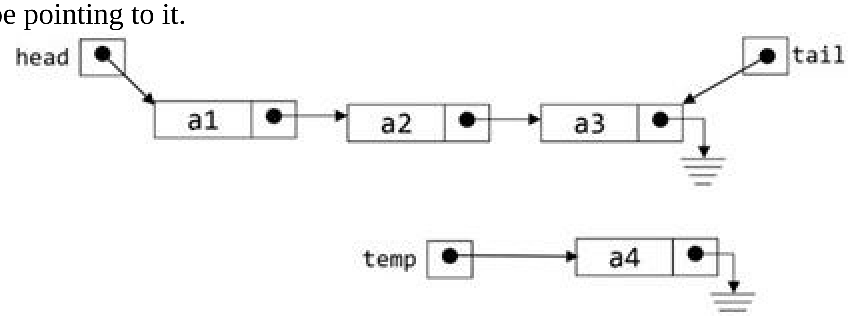
Queue Using linked list

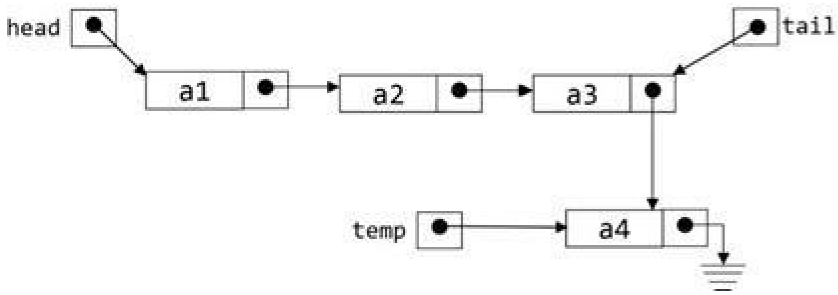
Example 9.2:

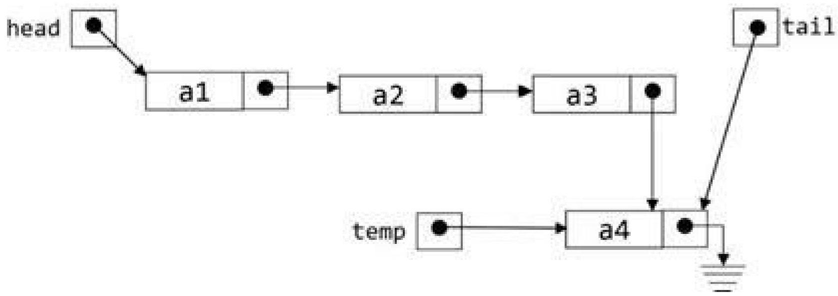
```
type Node struct {
    value int
    next  *Node
}

type Queue struct {
    head *Node
    tail *Node
    size int
}

func (q *Queue) Size() int {
    return q.size
}

func (q *Queue) IsEmpty() bool {
    return q.size == 0
}

func (q *Queue) Peek() (int, bool) {
    if q.IsEmpty() {
        fmt.Println("QueueEmptyError")
        return 0, false
    }
    return q.head.value, true
}

func (q *Queue) Print() {
    temp := q.head
    for temp != nil {
        fmt.Println(temp.value, " ")
        temp = temp.next
    }
}
```

Add

Enqueue into a queue using linked list. Nodes are added to the end of the linked list. Below diagram indicates how a new node is added to the list. The tail is modified every time when a new value is added to the queue. However, the head is also updated in the case when there is no element in the queue and when that first element is added to the queue both head and tail will be pointing to it.





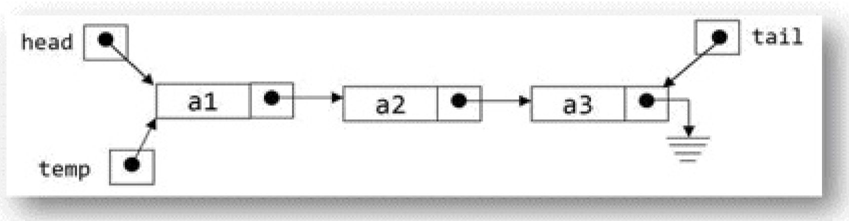


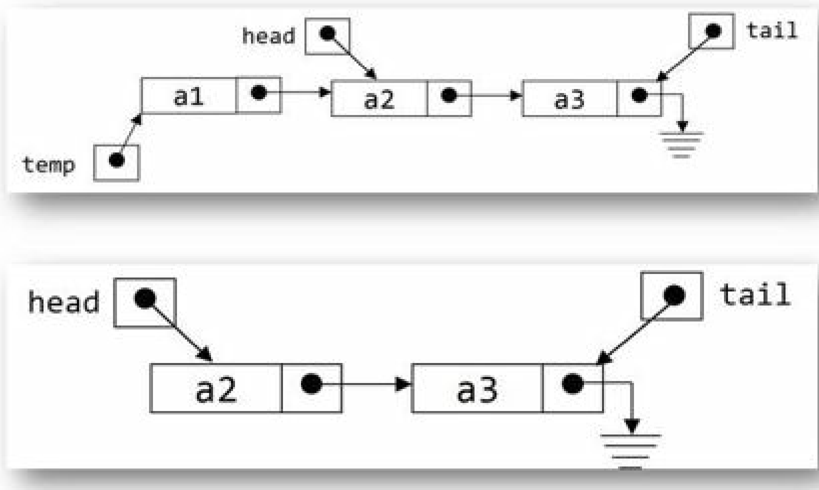
Example 9.3:

```
func (q *Queue) Add(value int) {
    temp := &Node{value}
    if q.head == nil {
        q.head = temp
        q.tail = temp
    } else {
        q.tail.next = temp
        q.tail = temp
    }
    q.size++
}
```

Analysis: add operation add one element at the end of the Queue (linked list).

Remove





In this we need the tail reference as it may be the case there was only one element in the list and the tail reference will also be modified in case of the remove.

Example 9.4:
```
func (q *Queue) Remove() (int, bool) {
    if q.IsEmpty() {
        fmt.Println("QueueEmptyError")
        return 0, false
    }

    value := q.head.value
    q.head = q.head.next
    q.size--
    return value, true
}
```

Analysis: Remove operation removes first node from the start of the queue (linked list).

Problems in Queue

Queue using a stack

How to implement a queue using a stack. You can use more than one stack.

Solution: We can use two stack to implement queue. We need to simulate first in first our using stack.

1. **Enqueue Operation**: new elements are added to the top of first stack.
2. **Dequeue Operation**: elements are popped from the second stack. When second stack is empty then all the elements of first stack are popped one by one and pushed into second stack.

Example 9.5:

```
type QueueUsingStack struct {
    stk1 Stack
    stk2 Stack
}

func (que *QueueUsingStack) Add(value int) {
    que.stk1.Push(value)
}

func (que *QueueUsingStack) Remove() int {
    var value int
    if que.stk2.IsEmpty() == false {
        value = que.stk2.Pop().(int)
        return value
    }

    for que.stk1.IsEmpty() == false {
        value = que.stk1.Pop().(int)
        que.stk2.Push(value)
    }

    value = que.stk2.Pop().(int)
    return value
}
```

Analysis: All add() happens to stack 1. When remove() is called remove happens from stack 2. When the stack 2 is empty then stack 1 is popped and pushed into stack 2. This popping from stack 1 and pushing into stack 2 revert the order of retrieval there by making queue behavior out of two stacks.

Stack using a Queue

Implement stack using a queue.

Solution 1: use two queue
Push: add new elements to queue1.
Pop: while size of queue1 is bigger than 1. Push all items from queue 1 to queue 2 except the last item. Switch the name of queue 1 and queue 2. Then return the last item.
Push operation is **O(1)** and Pop operation is **O(n)**

Solution 2: This same can be done using just one queue.

Push: add the element to queue.
Pop: find the size of queue. If size is zero then return error. Else, if size is positive then remove size- 1 elements from the queue and again add to the same queue. At last, remove the next element and return it.
Push operation is **O(1)** and Pop operation is **O(n)**

Solution 3: In the above solutions the push is efficient and pop is un efficient can we make pop efficient **O(1)** and push inefficient **O(n)**
Push: add new elements to queue2. Then add all the elements of queue 1 to queue 2. Then switch names of queue1 and queue 2.
Pop: remove from queue1

Reverse a stack

Reverse a stack using a queue
Solution 1:
- Pop all the elements of stack and add them into a queue.
- Then remove all the elements of the queue into stack
- We have the elements of the stack reversed.

Solution 2:
- Since dynamic list or [ ] list is used to implement stack in Go, we can iterate from both the directions of the list and swap the elements.

Reverse a queue

Reverse a queue-using stack
Solution:
- Dequeue all the elements of the queue into stack (append to the Go list [ ])
- Then pop all the elements of stack and add them into a queue. (pop the elements from the list)
- We have the elements of the queue reversed.

Breadth-First Search with a Queue

In breadth-first search, we explore all the nearest nodes first by finding all possible successors and add them to a queue.
- Create a queue
- Create a start point
- Enqueue the start point onto the queue
- while (value searching not found and the queue is not empty)
    - o Dequeue from the queue
    - o Find all possible points after the last one tried
    - o Enqueue these points onto the queue

Josephus problem

There are n people standing in a queue waiting to be executed. The counting begins at the front of the queue. In each step, k number of people are removed and again added one by one from the queue. Then the next person is executed. The execution proceeds around the circle until only the last person remains, who is given freedom.

Find that position where you want to stand and gain your freedom.

Solution:

- Just insert integer for 1 to k in a queue. (corresponds to k people)
- Define a Kpop() function such that it will remove and add the queue k-1 times and then remove one more time. (This man is dead.)
- Repeat second step until size of queue is 1.
- Print the value in the last element. This is the solution.

Exercise

1. Implement queue using dynamic memory allocation. Such that the implementation should follow the following constraints.
    a. The user should use memory allocation from the heap using new operator. In this, you need to take care of the max value in the queue.
    b. Once you are done with the above exercise and you are able to test your queue. Then you can add some more complexity to your code. In add() function when the queue is full, in place of printing, “Queue is full” you should allocate more space using new operator.
    c. Once you are done with the above exercise. Now in remove function once you are below half of the capacity of the queue, you need to decrease the size of the queue by half. You should add one more variable "min" to queue so that you can track what is the original value capacity passed at initialization() function. Moreover, the capacity of the queue will not go below the value passed in the initialization.

    (If you are not able to solve the above exercise, and then have a look into stack chapter, where we have done similar for stack)

2. Implement the below function for the queue:
    a. IsEmpty: This is left as an exercise for the user. Take a variable, which will take care of the size of a queue if the value of that variable is zero, isEmpty should return 1 (true). If the queue is not empty, then it should return 0 (false).
    b. Size: Use the size variable to be used under size function call. Size() function should return the number of elements in the queue.

3. Implement stack using a queue. Write a program for this problem. You can use just one queue.

4. Write a program to Reverse a stack using queue

5. Write a program to Reverse a queue using stack

6. Write a program to solve Josephus problem (algo already discussed.). There are n people standing in a queue waiting to be executed. The counting begins at the front of the queue. In each step, k number of people are removed and again added one by one from the queue. Then the next person is executed. The elimination proceeds around the circle until only the last person remains, who is given freedom. Find that position where you want to stand and gain your freedom.

7. Write a CompStack() function which takes reference to two stack as an argument and return true or false depending upon whether all the elements of the stack are equal or not. You are given isEqual(int, int) which will compare and return 1 if both values are equal and 0 if they are different.

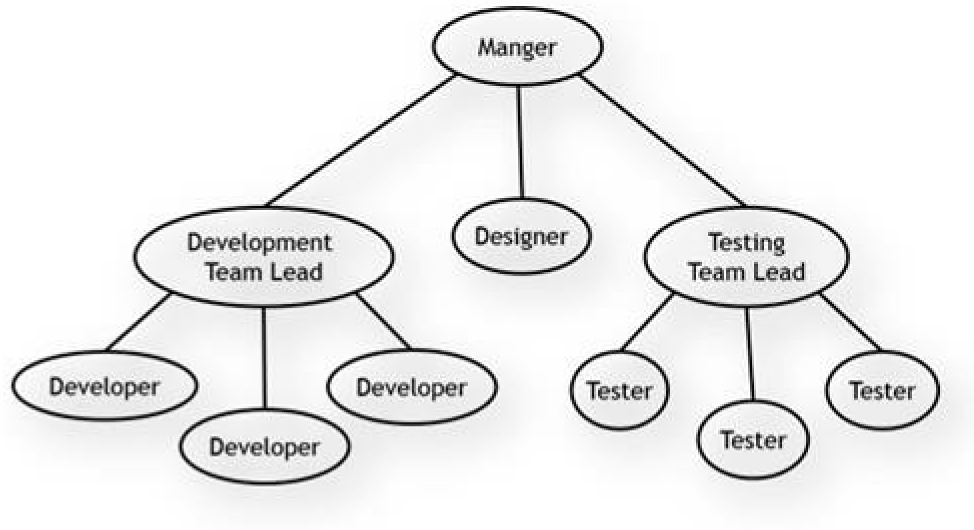
CHAPTER 10: TREE

Introduction

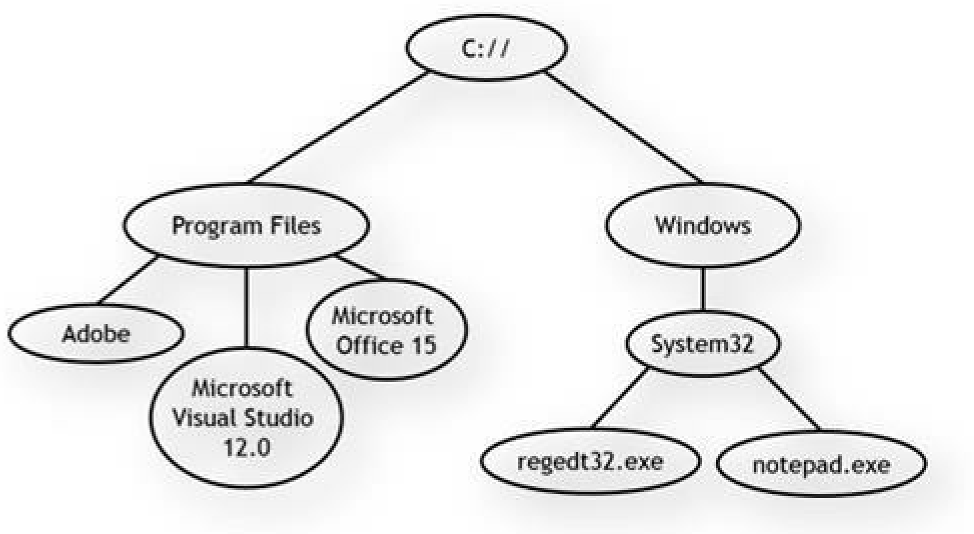
We have already read about various linear data structures like a list, linked list, stack, queue etc.
Both list and linked list have a drawback of linear time required for searching an element.

A tree is a nonlinear data structure, which is used to represent hierarchical relationships (parent-child relationship). Each node is connected by another node by directed edges.

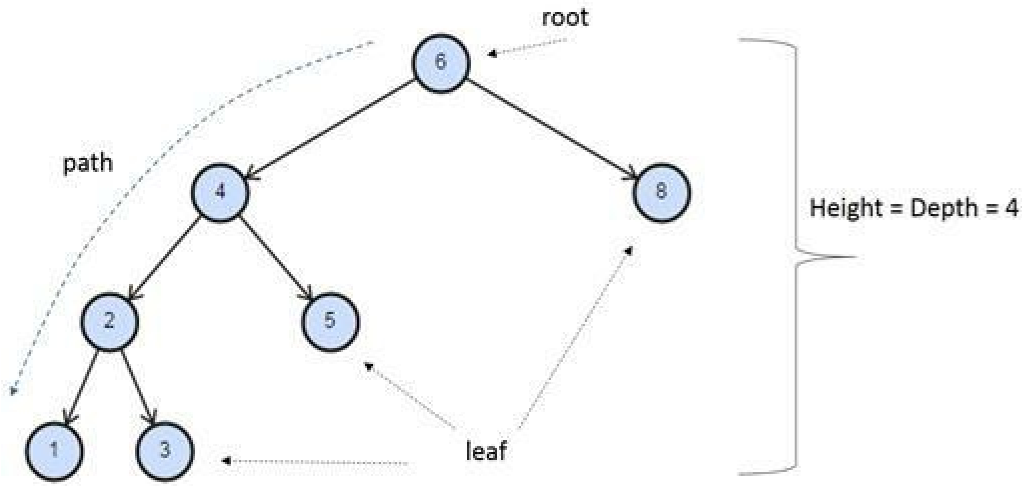
Example 1: Tree in organization



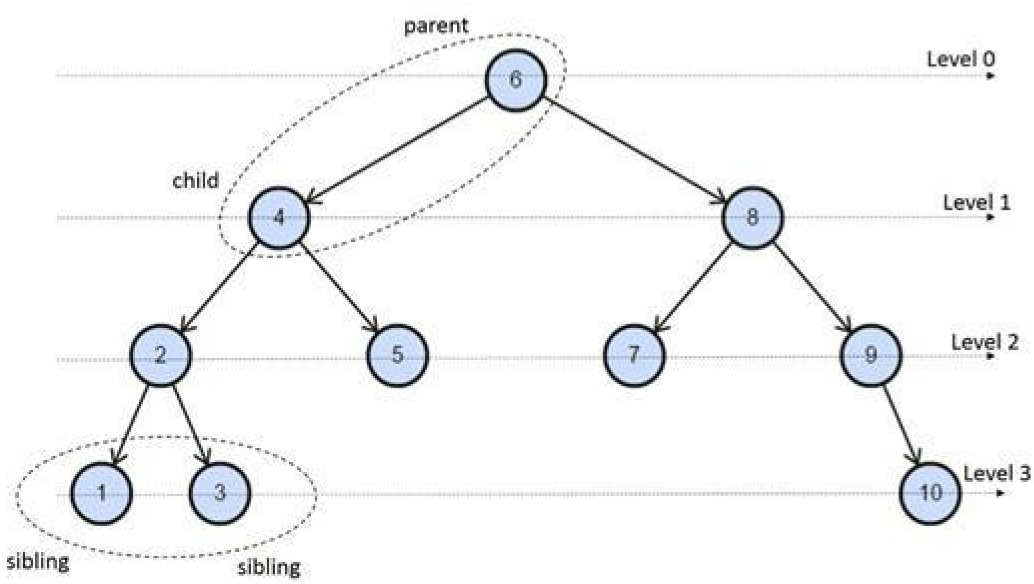
Example 2: Tree in a file system



Terminology in tree



Root: The root of the tree is the only node without any incoming edges. It is the top node of a tree.
Node: It is a fundamental element of a tree. Each node has data and two references that may point to null or its children
Edge: It is also a fundamental part of a tree, which is used to connect two nodes.
Path: A path is an ordered list of nodes that are connected by edges.
Leaf: A leaf node is a node that has no children.
Height of the tree: The height of a tree is the number of edges on the longest path between the root and a leaf.
The level of node: The level of a node is the number of edges on the path from the root node to that node.

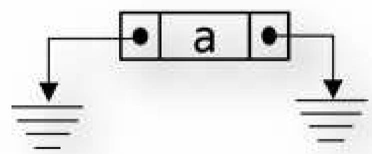


Children: Nodes that have incoming edges from the same node to be said to be the children of that node.
Parent: Node is a parent of all the child nodes that are linked by outgoing edges.
Sibling: Nodes in the tree that are children of the same parent are called siblings.
Ancestor: A node reachable by repeated moving from child to parent.

Binary Tree

A binary tree is a type tree in which each node has at most two children (0, 1 or 2), which are referred to as the left child and the right child.

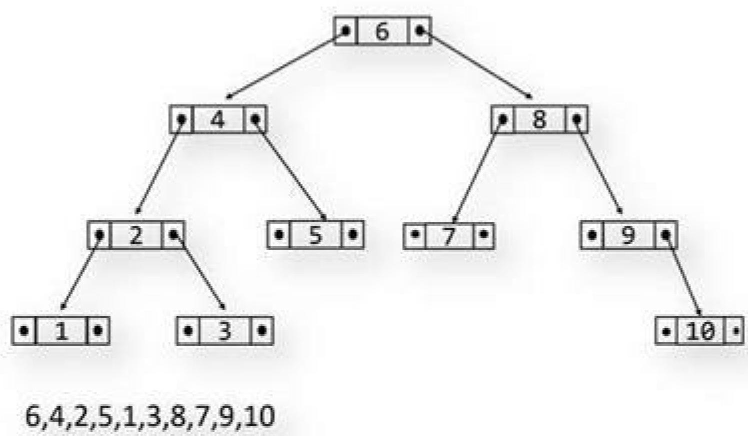
Below is a node of the binary tree with "a" stored as data and whose left child (lChild) and whose right child (rchild) both pointing towards null.



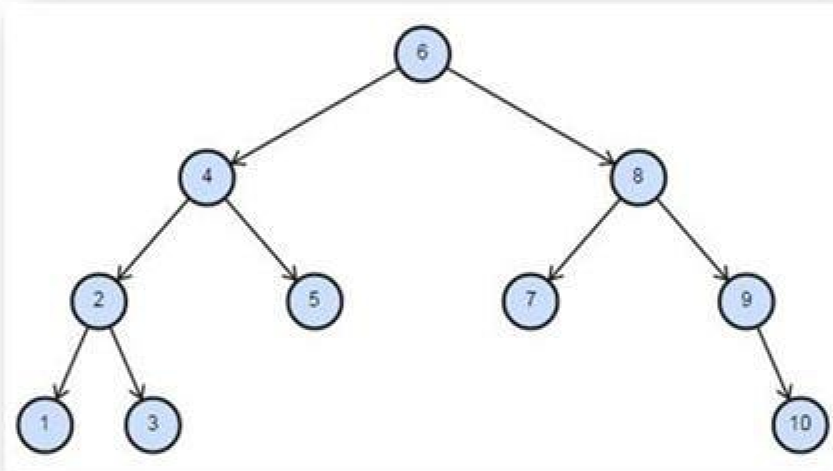
Below is a class definition used to define node.

```
type Node struct {
    value   int
    left, right *Node
}

type Tree struct {
    root *Node
}
```

Below is a binary tree whose nodes contains data from 1 to 10



In the rest of the book, binary tree will be represented as below:



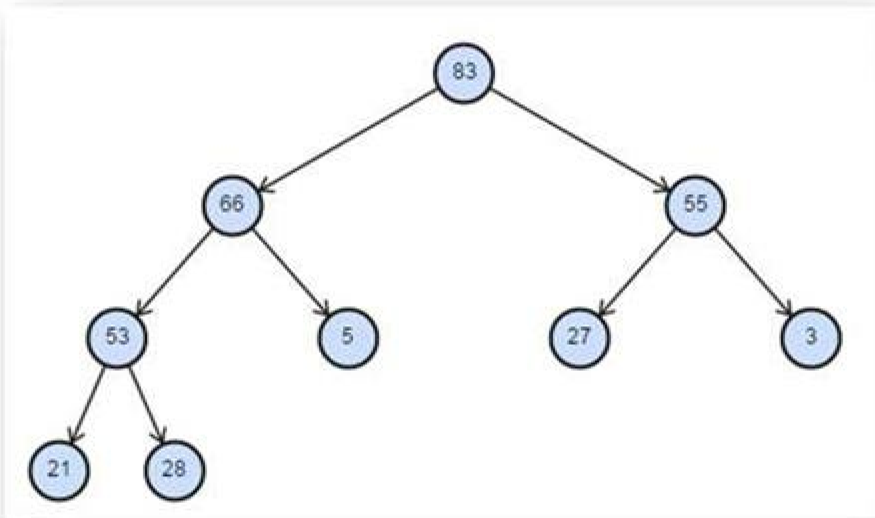
Properties of Binary tree are:

1. The maximum number of nodes on level i of a binary tree is 2^i , where i >= 1
2. The maximum number of nodes in a binary tree of depth k is 2^{k+1}, where k >= 1
3. There is exactly one path from the root to any nodes in a tree.
4. A tree with N nodes have exactly N-1 edges connecting these nodes.
5. The height of a complete binary tree of N nodes is $\log_2 N$.

Types of Binary trees

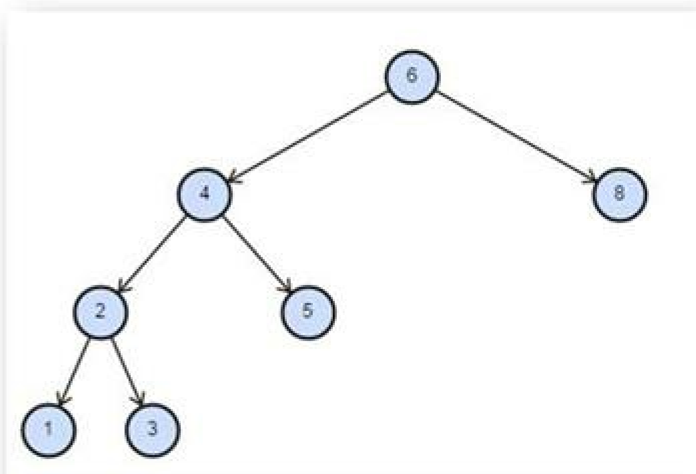
Complete binary tree

In a complete binary tree, every level except the last one is completely filled. All nodes in the left are filled first, then the right one. A binary heap is an example of a complete binary tree.



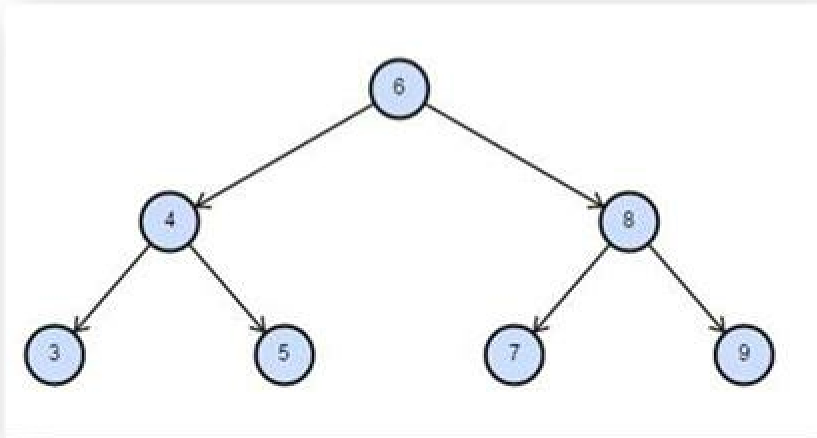
Full/ Strictly binary tree

The full binary tree is a binary tree in which each node has exactly zero or two children.



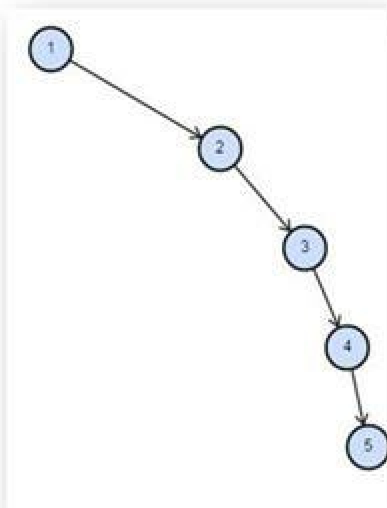
Perfect binary tree

The perfect binary tree is a type of full binary tree in which each non-leaf node has exactly two child nodes. All leaf nodes have identical path length and all possible node slots are occupied



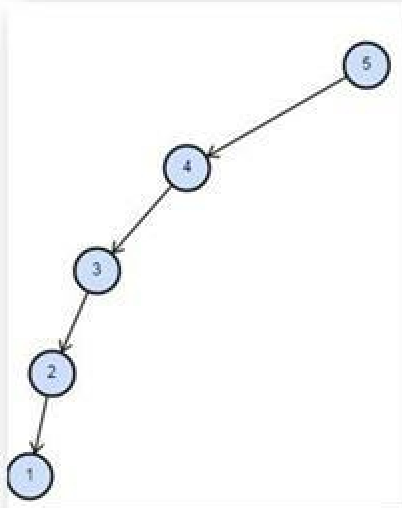
Right skewed binary tree

A binary tree in which either each node is have a right child or no child (leaf) is called as right skewed binary tree



Left skewed binary tree

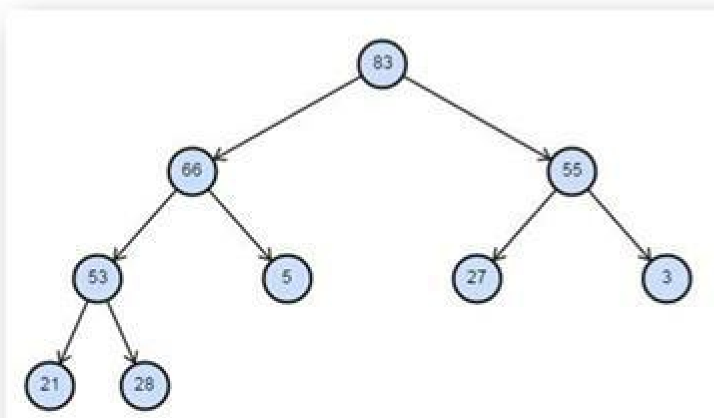
A binary tree in which either each node is have a left child or no child (leaf) is called as Left skewed binary tree



Height-balanced Binary Tree

A height-balanced binary tree is a binary tree such that the left & right subtrees for any given node differ in height by max one. AVL tree and RB tree are an example of height balanced tree we will discuss these trees later in this chapter.

Note: Each complete binary tree is a height-balanced binary tree



Problems in Binary Tree

Create a Complete binary tree

Create a binary tree given a list of values.

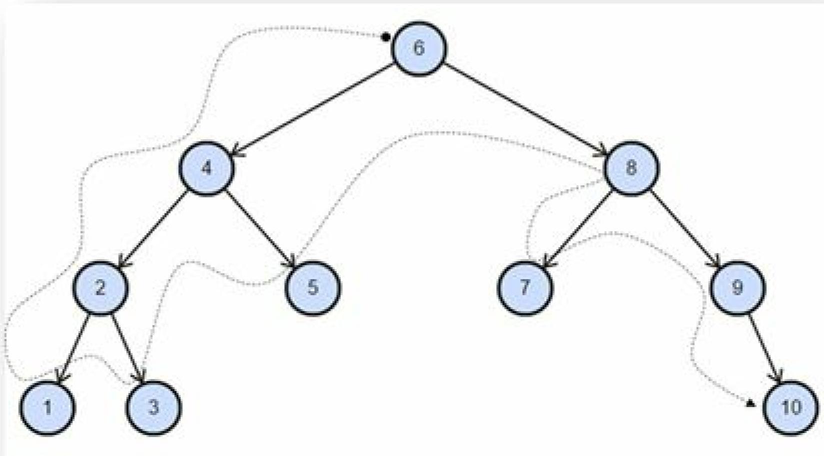
Solution: Since there is no order defined in a binary tree, so nodes can be inserted in any order so it can be a skewed binary tree. But it is inefficient to do anything in a skewed binary tree so we will create a Complete binary tree. At each node, the middle value stored in the array is assigned to node. The left of array is passed to the left child of the node to create left sub-tree and the right portion of array is passed to right child of the node to crate right sub-tree.

Example 10.1:
```
func LevelOrderBinaryTree(arr []int) *Tree {
	tree := new(Tree)
	tree.root = levelOrderBinaryTree(arr, 0, len(arr))
	return tree
}

func levelOrderBinaryTree(arr []int, start int, size int) *Node {
	curr := &Node{arr[start], nil, nil}

	left := 2*start + 1
	right := 2*start + 2

	if left < size {
		curr.left = levelOrderBinaryTree(arr, left, size)
	}
	if right < size {
		curr.right = levelOrderBinaryTree(arr, right, size)
	}
	return curr
}

func main() {
	arr := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
	t2 := LevelOrderBinaryTree(arr)
	t2.PreOrder()
}
```

Complexity Analysis: This is an efficient algorithm for creating a complete binary tree.
Time Complexity: O(n), Space Complexity: O(n)

Pre-Order Traversal

Traversal is a process of visiting each node of a tree. In Pre-Order Traversal parent is visited / traversed first, then left child and then right child. Pre-Order traversal is a type of depth-first traversal.



Solution: Preorder traversal is done using recursion. At each node, first the value stored in it is printed and then followed by the value of left child and right child. At each node its value is printed followed by calling printTree() function to its left and right child to print left and right sub-tree.

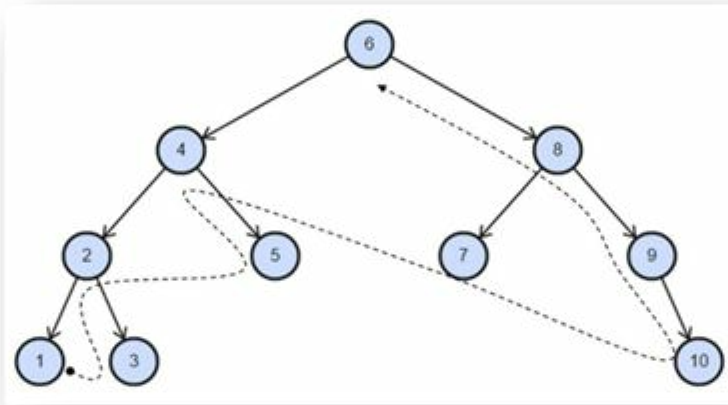
Example 10.2:
```
func (t *Tree) PrintPreOrder() {
	printPreOrder(t.root)
	fmt.Println()
}

func printPreOrder(n *Node) {
	if n == nil {
		return
	}
	fmt.Print(n.value, " ")
	printPreOrder(n.left)
	printPreOrder(n.right)
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Note: When there is an algorithm in which all nodes are traversed then complexity cannot be less than O(n). When there is a large portion of the tree, which is not traversed, then complexity reduces.

Post-Order Traversal

In Post-Order Traversal left child is visited / traversed first, then right child and then parent
Post-Order traversal is a type of depth-first traversal.



Solution: At each node, first the left child is traversed then right child and in the end, current node value is printed to the screen.

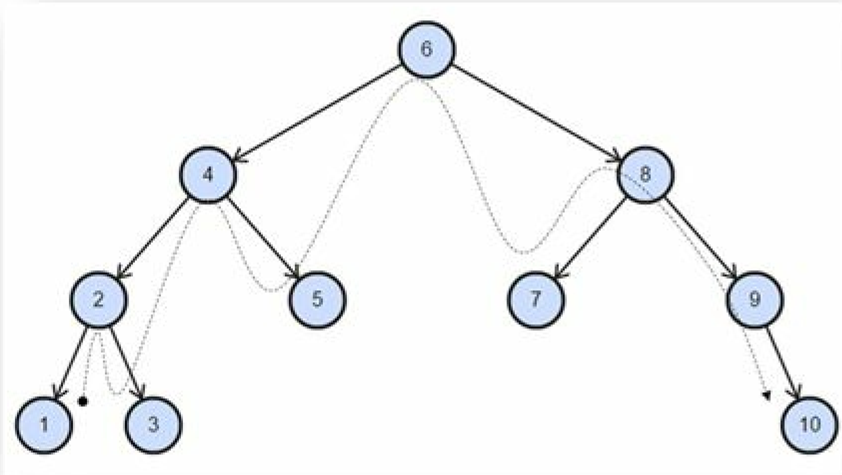
Example 10.3:
```
func (t *Tree) PrintPostOrder() {
	printPostOrder(t.root)
	fmt.Println()
}

func printPostOrder(n *Node) {
	if n == nil {
		return
	}
	printPostOrder(n.left)
	printPostOrder(n.right)
	fmt.Print(n.value)
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

In-Order Traversal

In In-Order Traversal, left child is visited / traversed first, then the parent value is printed and last right child is traversed.
In-Order traversal is a type of depth-first traversal. The output of In-Order traversal of BST is a sorted list.

Solution: In In-Order traversal first, the value of left child is traversed, then the value of node is printed to the screen and then the value of right child is traversed.



Example 10.4:

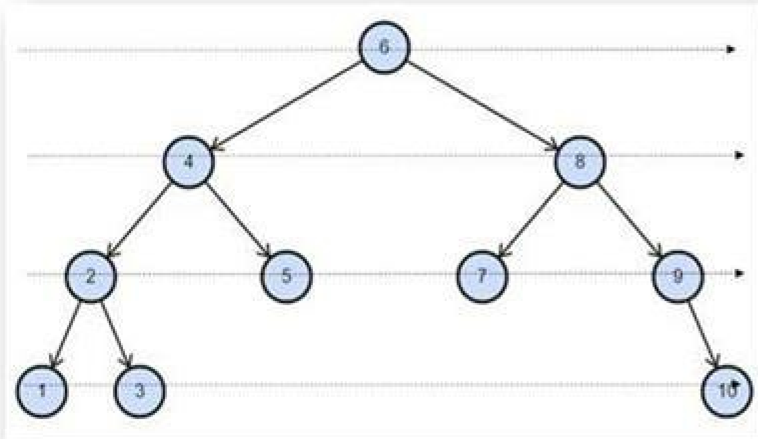
```
func (t *Tree) PrintInOrder() {
	printInOrder(t.root)
	fmt.Println()
}

func printInOrder(n *Node) {
	if n == nil {
		return
	}
	printInOrder(n.left)
	fmt.Print(n.value)
	printInOrder(n.right)
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Note: Pre-Order, Post-Order, and In-Order traversal are for all binary trees. They can be used to traverse any kind of a binary tree.

Level order traversal / Breadth First traversal

Write code to implement level order traversal of a tree. Such that nodes at depth k is printed before nodes at depth k+1.



Solution: Level order traversal or Breadth First traversal of a tree is done using a queue. At the start, the root node reference is added to queue. The traversal of tree happens until its queue is empty. When we traverse the tree, we first remove an element from the queue, print the value stored in that node and then its left child and right child will be added to the queue.

Example 10.5:

```
func (t *Tree) PrintBredthFirst() {
    que := new(queue.Queue)
    var temp *Node

    if t.root != nil {
        que.Put(t.root)
    }

    for que.Empty() == false {
        temp2, _ := que.Get(1)
        temp = temp2[0].(*Node)
        fmt.Print(temp.value, " ")
        if temp.left != nil {
            que.Put(temp.left)
        }
        if temp.right != nil {
            que.Put(temp.right)
        }
    }
    fmt.Println()
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Tree Depth

Solution: Depth is tree is calculated recursively by traversing left and right child of the root. At each level of traversal depth of left child is calculated and depth of right child is calculated. The greater depth among the left and right child is added by one (which is the depth of the current node) and this value is returned.

Example 10.6:

```
func (t *Tree) TreeDepth() int {
    return treeDepth(t.root)
}

func treeDepth(root *Node) int {
    if root == nil {
        return 0
    }
    lDepth := treeDepth(root.left)
    rDepth := treeDepth(root.right)

    if lDepth > rDepth {
        return lDepth + 1
    }
    return rDepth + 1
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Nth Pre-Order

Solution: We want to print the node, which will be at the nth index when we print the tree in PreOrder traversal. Therefore, we keep a counter to keep track of the index. When the counter is equal to index, then we print the value and return the Nth preorder index node.

Example 10.7:

```
func (t *Tree) NthPreOrder(index int) {
    var counter int
    nthPreOrder(t.root, index, &counter)
}

func nthPreOrder(node *Node, index int, counter *int) {
    if node != nil {
        (*counter)++
        if *counter == index {
            fmt.Println(node.value)
        }
        nthPreOrder(node.left, index, counter)
        nthPreOrder(node.right, index, counter)
    }
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Nth Post Order

Solution: We want to print the node that will be at the nth index when we print the tree in post order traversal. Therefore, we keep a counter to keep track of the index, but at this time, we will increment the counter after left child and right child traversal. When the counter is equal to index, then we print the value and return the nth post-order index node.

Example 10.8

```
func (t *Tree) NthPostOrder(index int) {
	var counter int
	nthPostOrder(t.root, index, &counter)
}

func nthPostOrder(node *Node, index int, counter *int) {
	if node != nil {
		nthPostOrder(node.left, index, counter)
		nthPostOrder(node.right, index, counter)
		(*counter)++
		if *counter == index {
			fmt.Println(node.value)
		}
	}
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Nth In Order

Solution: We want to print the node that will be at the nth index when we print the tree in in-order traversal. Therefore, we keep a counter to keep track of the index, but at this time, we will increment the counter after left child traversal but before the right child traversal. When the counter is equal to index, then we print the value and return the nth in-order index node.

Example 10.9:

```
func (t *Tree) NthInOrder(index int) {
	var counter int
	nthInOrder(t.root, index, &counter)
}

func nthInOrder(node *Node, index int, counter *int) {
	if node != nil {
		nthInOrder(node.left, index, counter)
		*counter++
		if *counter == index {
			fmt.Println(node.value)
		}
		nthInOrder(node.right, index, counter)
	}
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(1)

Copy Tree

Solution: Copy tree is done by copy nodes of the input tree at each level of the traversal of the tree. At each level of the traversal of nodes of the tree, a new node is created and the value of the input tree node is copied to it. The left child tree is copied recursively and then reference to new subtree is returned which will be assigned to the left child reference of the current new node. Similarly for the right child node too. Finally, the tree is copied.

Example 10.10:

```go
func (t *Tree) CopyTree() *Tree {
	Tree2 := new(Tree)
	Tree2.root = copyTree(t.root)
	return Tree2
}

func copyTree(curr *Node) *Node {
	var temp *Node
	if curr != nil {
		temp = new(Node)
		temp.value = curr.value
		temp.left = copyTree(curr.left)
		temp.right = copyTree(curr.right)
		return temp
	}
	return nil
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Copy Mirror Tree

Solution: Copy mirror image of the tree is done same as copy tree, but in place of left child pointing to the tree formed by left child traversal of input tree. This time left child points to the tree formed by right child traversal of the input tree. Similarly right child point to the tree formed by the traversal of the left child of the input tree.

Example 10.11:

```go
func (t *Tree) CopyMirrorTree() *Tree {
	tree := new(Tree)
	tree.root = copyMirrorTree(t.root)
	return tree
}

func copyMirrorTree(curr *Node) *Node {
	var temp *Node
	if curr != nil {
		temp = new(Node)
		temp.value = curr.value
		temp.right = copyMirrorTree(curr.left)
		temp.left = copyMirrorTree(curr.right)
		return temp
	}
	return nil
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Number of Element

Solution: Number of nodes at the right child and the number of nodes at the left child is added by one and we get the total number of nodes in any tree / sub-tree.

Example 10.12:

```
func (t *Tree) NumNodes() int {
    return numNodes(t.root)
}

func numNodes(curr *Node) int {
    if curr == nil {
        return 0
    }
    return (1 + numNodes(curr.right) + numNodes(curr.left))
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Number of Leaf nodes

Solution: If we add the number of leaf node in the right child with the number of leaf nodes in the left child, we will get the total number of leaf node in any tree or subtree.

Example 10.13:

```
func (t *Tree) NumLeafNodes() int {
    return numLeafNodes(t.root)
}

func numLeafNodes(curr *Node) int {
    if curr == nil {
        return 0
    }
    if curr.left == nil && curr.right == nil {
        return 1
    }
    return (numLeafNodes(curr.right) + numLeafNodes(curr.left))
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Identical

Solution: Two trees have identical values if at each level the value is equal.

Example 10.14:

```
func (t *Tree) IsEqual(t2 *Tree) bool {
    return isEqual(t.root, t2.root)
}

func isEqual(node1 *Node, node2 *Node) bool {
    if node1 == nil && node2 == nil {
        return true
    } else if node1 == nil || node2 == nil {
        return false
    } else {
        return ((node1.value == node2.value) &&
```

```
            isEqual(node1.left, node2.left) &&
            isEqual(node1.right, node2.right))
    }
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Free Tree

Solution: You just need to make the root of the tree point to nil. The system will do garbage collection and recover the memory assigned to the tree. Effectively you had done a single act and because of this action, the time complexity is constant.

Example 10.15:

```
func (t *Tree) Free() {
    t.root = nil
}
```

Complexity Analysis: Time Complexity: O(1), Space Complexity: O(1)

Print all the paths

Print all the paths from the roots to the leaf

Solution: Whenever we traverse a node, we add that node to the list. When we reach a leaf, we print the whole list. When we return from a function, then we remove the element that was added to the list when we entered this function.

Example 10.16:

```
func (t *Tree) PrintAllPath() {
    stk := new(Stack)
    printAllPath(t.root, stk)
}

func printAllPath(curr *Node, stk *Stack) {
    if curr == nil {
        return
    }
    stk.Push(curr.value)
    if curr.left == nil && curr.right == nil {
        stk.Print()
        stk.Pop()
        return
    }

    printAllPath(curr.right, stk)
    printAllPath(curr.left, stk)
    stk.Pop()
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Least Common Ancestor

Solution: We recursively traverse the nodes of a binary tree. We find any one of the input node for which we are searching a common ancestor then we return that node. When we get both the left and right as some valid reference location other than null, we will return that node as the common ancestor.

Example 10.17:
```
func (t *Tree) LCA(first int, second int) (int, bool) {
	ans := LCAUtil(t.root, first, second)
	if ans != nil {
		return ans.value, true
	}
	fmt.Println("NotFoundError")
	return 0, false
}

func LCAUtil(curr *Node, first int, second int) *Node {
	var left, right *Node

	if curr == nil {
		return nil
	}

	if curr.value == first || curr.value == second {
		return curr
	}

	left = LCAUtil(curr.left, first, second)
	right = LCAUtil(curr.right, first, second)

	if left != nil && right != nil {
		return curr
	} else if left != nil {
		return left
	} else {
		return right
	}
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Find Max in Binary Tree

Solution: We recursively traverse the nodes of a binary tree. We will find the maximum value in the left and right subtree of any node then will compare the value with the value of the current node and finally return the largest of the three values.

Example 10.18:
```
func (t *Tree) FindMaxBT() int {
	return findMaxBT(t.root)
}
```

```
func findMaxBT(curr *Node) int {
	if curr == nil {
		return math.MinInt32
	}

	max := curr.value
	left := findMaxBT(curr.left)
	right := findMaxBT(curr.right)
	if left > max {
		max = left
	}

	if right > max {
		max = right
	}
	return max
}
```

Search value in a Binary Tree

Solution: To find if some value is there in a binary tree or not is done using exhaustive search of the binary tree. First, the value of current node is compared with the value, which we are looking for. Then it is compared recursively inside the left child and right child.

Example 10.19:
```
func SearchBT(root *Node, value int) bool {
	var left, right bool
	if root == nil || root.value == value {
		return false
	}
	left = SearchBT(root.left, value)
	if left {
		return true
	}
	right = SearchBT(root.right, value)
	if right {
		return true
	}
	return false
}
```

Maximum Depth in a Binary Tree

Solution: To find the maximum depth of a binary tree we need to find the depth of the left tree and depth of right tree then we need to store the value and increment it by one so that we get depth of the given node.

Example 10.20:
```
func (t *Tree) TreeDepth() int {
	return treeDepth(t.root)
}

func treeDepth(root *Node) int {
```

```
    if root == nil {
        return 0
    }
    lDepth := treeDepth(root.left)
    rDepth := treeDepth(root.right)

    if lDepth > rDepth {
        return lDepth + 1
    }
    return rDepth + 1
}
```

Number of Full Nodes in a BT

Solution: A full node is a node that have both left and right child. We will recursively traverse the whole tree and will increase the count of full node as we find them.

Example 10.21:

```
func (t *Tree) NumFullNodesBT() int {
    return numFullNodesBT(t.root)
}

func numFullNodesBT(curr *Node) int {
    var count int
    if curr == nil {
        return 0
    }
    count = numFullNodesBT(curr.right) + numFullNodesBT(curr.left)
    if curr.right != nil && curr.left != nil {
        count++
    }
    return count
}
```

Maximum Length Path in a BT/ Diameter of BT

Solution: To find the diameter of BT we need to find the depth of left child and right child then will add these two values and increment it by one so that we will get the maximum length path (diameter candidate) which contains the current node. Then we will find max length path in the left child sub-tree. Will also find the max length path in the right child sub-tree. Finally, we will compare the three values and return the maximum value out of these this will be the diameter of the Binary tree.

Example 10.22:

```
func (t *Tree) MaxLengthPathBT() int {
    return maxLengthPathBT(t.root)
}

func maxLengthPathBT(curr *Node) int {
    var max, leftPath, rightPath, leftMax, rightMax int
    if curr == nil {
        return 0
```

```
    }
    leftPath = treeDepth(curr.left)
    rightPath = treeDepth(curr.right)
    max = leftPath + rightPath + 1
    leftMax = maxLengthPathBT(curr.left)
    rightMax = maxLengthPathBT(curr.right)
    if leftMax > max {
        max = leftMax
    }
    if rightMax > max {
        max = rightMax
    }
    return max
}
```

Sum of All nodes in a BT

Solution: We will find the sum of all the nodes recursively. sumAllBT() will return the sum of all the node of left and right subtree then will add the value of current node and will return the final sum.

Example 10.23:

```
func (t *Tree) SumAllBT() int {
    return sumAllBT(t.root)
}

func sumAllBT(curr *Node) int {
    var sum, leftSum, rightSum int
    if curr == nil {
        return 0
    }
    rightSum = sumAllBT(curr.right)
    leftSum = sumAllBT(curr.left)
    sum = rightSum + leftSum + curr.value
    return sum
}
```

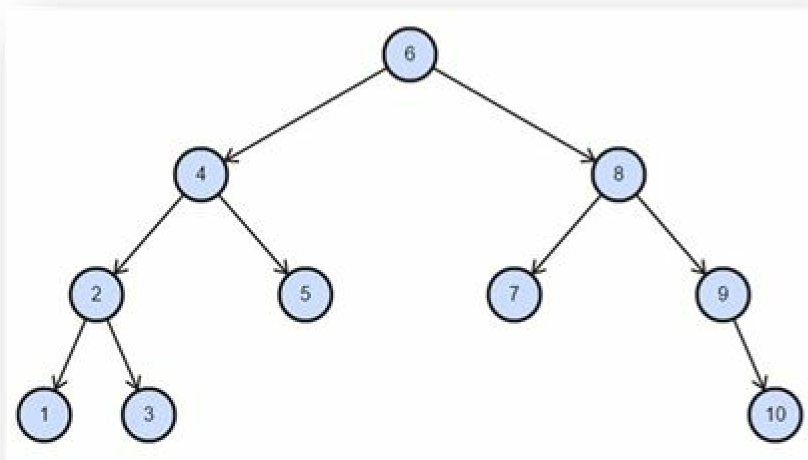
Binary Search Tree (BST)

A binary search tree (BST) is a binary tree on which nodes are ordered in the following way:

- The key in the left subtree is less than the key in its parent node.
- The key in the right subtree is greater the key in its parent node.
- No duplicate key allowed.

Note: there can be two separate key and value fields in the tree node. But for simplicity, we are considering value as the key. All problems in the binary search tree are solved using this supposition that the value in the node is key for the tree.

Note: Since binary search tree is a binary tree to all the above algorithm of a binary tree are applicable to a binary search tree.



Problems in Binary Search Tree (BST)

All binary tree algorithms are valid for binary search tree too.

Create a binary search tree from sorted list

Create a binary tree given list of values in a list in sorted order. Since the elements in the list are in sorted order and we want to create a binary search tree in which left subtree nodes are having values less than the current node and right subtree nodes have value greater than the value of the current node.

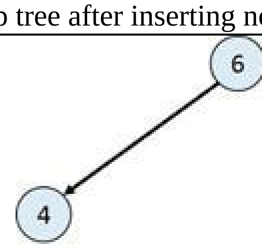
Solution: We have to find the middle node to create a current node and send the rest of the list to construct left and right subtree.

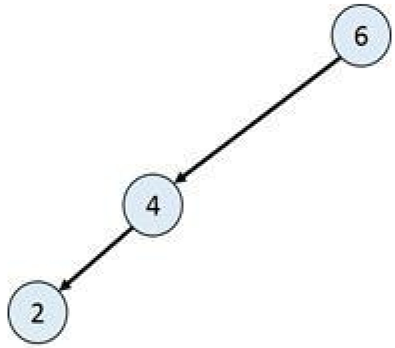
Example 10.24:

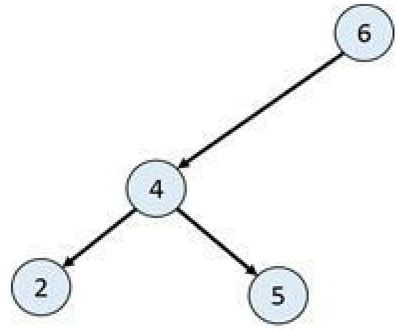
```
func CreateBinaryTree(arr []int) *Tree {
	t := new(Tree)
	size := len(arr)
	t.root = createBinaryTreeUtil(arr, 0, size-1)
	return t
}

func createBinaryTreeUtil(arr []int, start int, end int) *Node {
	if start > end {
		return nil
	}

	mid := (start + end) / 2
	curr := new(Node)
	curr.value = arr[mid]
	curr.left = createBinaryTreeUtil(arr, start, mid-1)
	curr.right = createBinaryTreeUtil(arr, mid+1, end)
	return curr
}

func main() {
	arr := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
	t := CreateBinaryTree(arr)
}
```

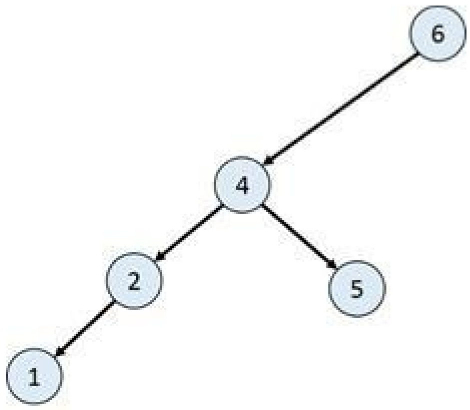
Insertion

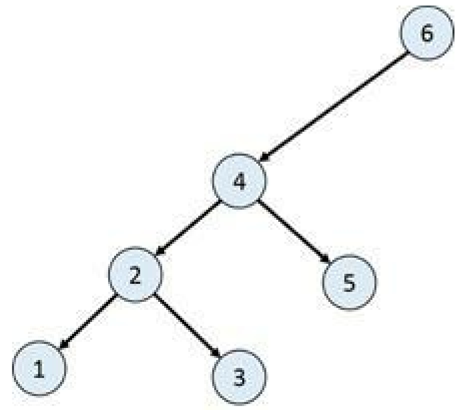
Nodes with key 6,4,2,5,1,3,8,7,9,10 are inserted in a tree. Below is step by step tree after inserting nodes in the order.

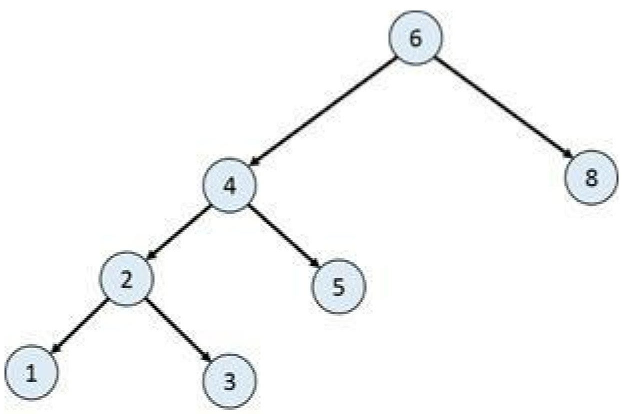
	

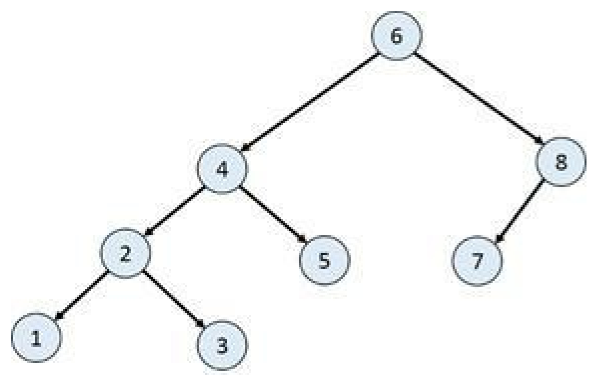

6
4
2

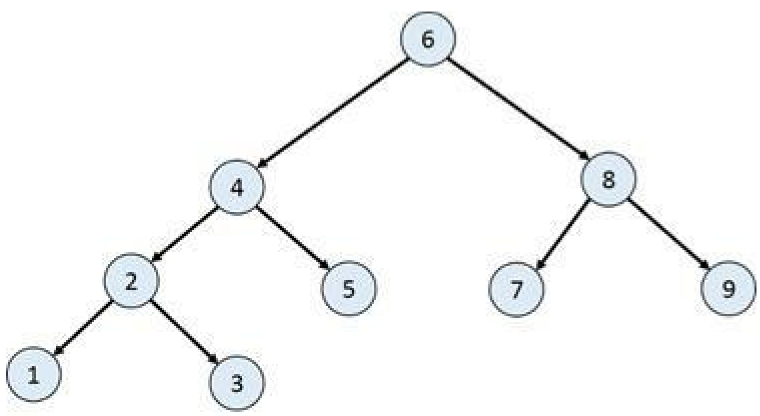

6
4
2
5

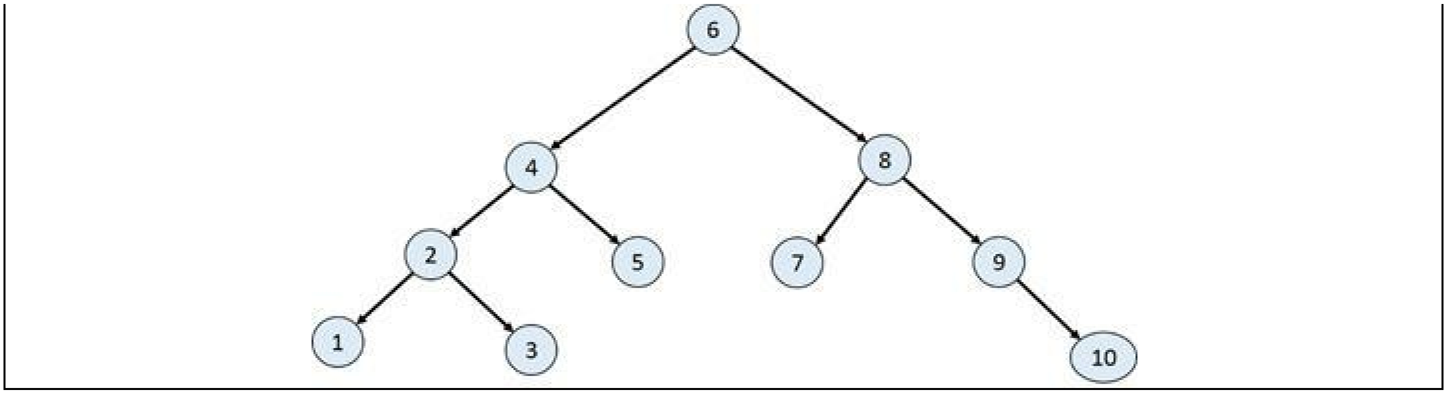

6
4
2
5
1


6
4
2
5
1
3


6
4
8
2
5
1
3


6
4
8
2
5
7
1
3


6
4
8
2
5
7
9
1
3



Solution: Smaller values will be added to the left child sub-tree of a node and greater value will be added to the right child sub-tree of the current node.

Example 10.25:
```
func (t *Tree) Add(value int) {
    t.root = addUtil(t.root, value)
}

func addUtil(n *Node, value int) *Node {
    if n == nil {
        n = new(Node)
        n.value = value
        return n
    }
    if value < n.value {
        n.left = addUtil(n.left, value)
    } else {
        n.right = addUtil(n.right, value)
    }
    return n
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Find Node

Solution: The value greater than the current node value will be in the right child sub-tree and the value smaller than the current node is in the left child sub-tree. We can find a value by traversing the left or right subtree iteratively.

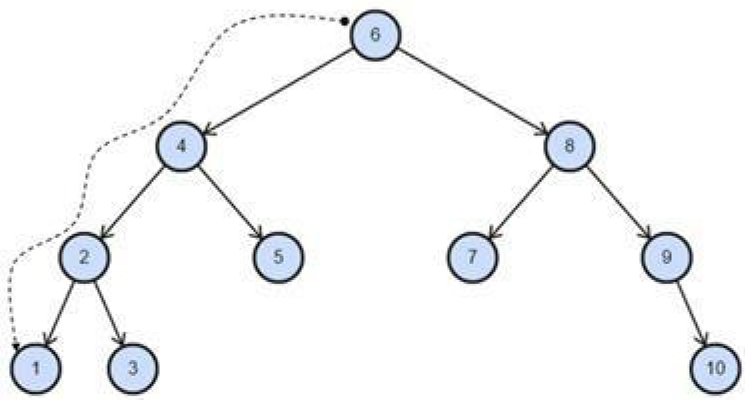
Example 10.26: Find the node with the value given.
```
func (t *Tree) Find(value int) bool {
    var curr *Node = t.root
    for curr != nil {
        if curr.value == value {
            return true
        } else if curr.value > value {
            curr = curr.left
        } else {
            curr = curr.right
        }
    }
```

```
    return false
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(1)

Find Min

Find the node with the minimum value.
Solution: left most child of the tree will be the node with the minimum value.



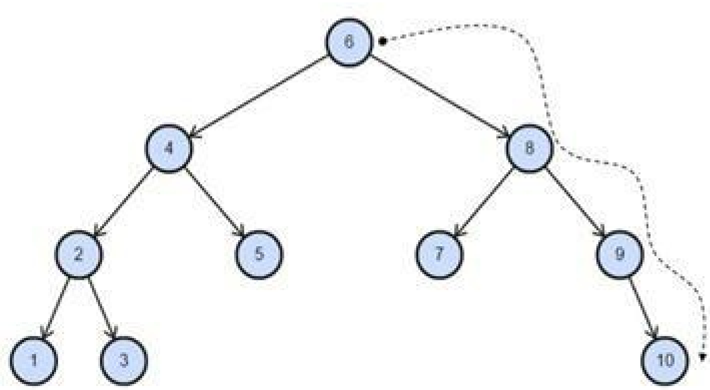
Example 10.27:

```
func (t *Tree) FindMin() (int, bool) {
    var node *Node = t.root
    if node == nil {
        fmt.Println("EmptyTreeError")
        return 0, false
    }
    for node.left != nil {
        node = node.left
    }
    return node.value, true
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(1)

Find Max

Find the node in the tree with the maximum value.



Solution: Right most node of the tree will be the node with the maximum value.

Example 10.28:

```
func (t *Tree) FindMax() (int, bool) {
	var node *Node = t.root
	if node == nil {
		fmt.Println("EmptyTreeError")
		return 0, false
	}
	for node.right != nil {
		node = node.right
	}
	return node.value, true
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(1)

Is tree a BST

Approach 1: At each node we check, max value of left subtree is smaller than the value of current node and min value of right subtree is greater than the current node.

Example 10.29:

```
func IsBST3(root *Node) bool {
	if root == nil {
		return true
	}
	if root.left != nil && FindMax(root.left).value > root.value {
		return false
	}
	if root.right != nil && FindMin(root.right).value <= root.value {
		return false
	}
	return (IsBST3(root.left) && IsBST3(root.right))
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

The above solution is correct but it is not efficient as same tree nodes are traversed many times.

Approach 2: A better solution will be the one in which we will look into each node only once. This is done by narrowing the range. We will be using an isBSTUtil() function which take the max and min range of the values of the nodes. The initial value of min and max will be INT_MIN and INT_MAX.

Example 10.30:

```
func (t *Tree) IsBST() bool {
	return IsBST(t.root, math.MinInt32, math.MaxInt32)
}

func IsBST(curr *Node, min int, max int) bool {
	if curr == nil {
		return true
	}
	if curr.value < min || curr.value > max {
```

```
        return false
    }
    return IsBST(curr.left, min, curr.value) && IsBST(curr.right, curr.value, max)
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n) for stack

Approach 3: Above method is correct and efficient but there is an easy method to do the same. We can do in-order traversal of nodes and see if we are getting a strictly increasing sequence

Example 10.31:

```
func (t *Tree) IsBST2() bool {
    var c int
    return IsBST2(t.root, &c)
}

func IsBST2(root *Node, count *int) bool {
    var ret bool
    if root != nil {
        ret = IsBST2(root.left, count)
        if !ret {
            return false
        }
        if *count > root.value {
            return false
        }
        *count = root.value
        ret = IsBST2(root.right, count)
        if !ret {
            return false
        }
    }
    return true
}
```

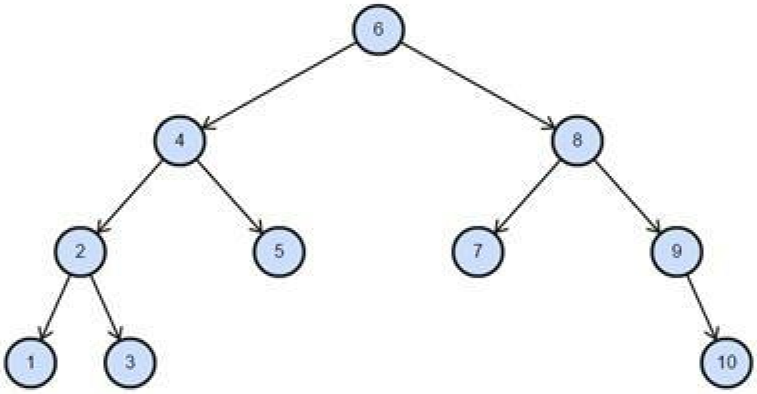
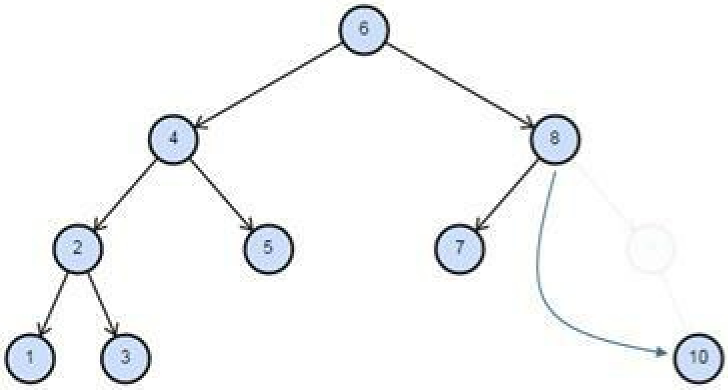
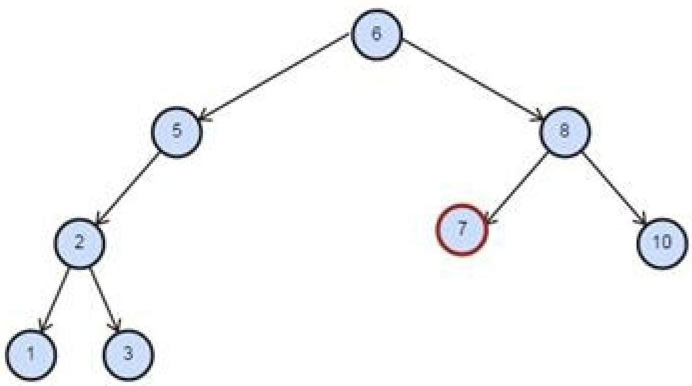
Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n) for stack

Delete Node

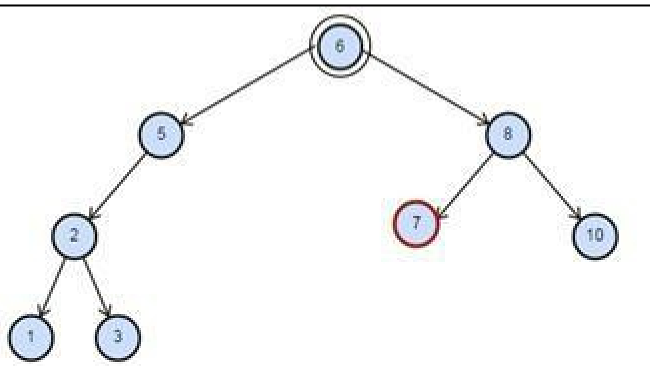
Description: Remove the node x from the binary search tree, making the necessary, reorganize nodes of binary search tree to maintain its properties.
There are three cases in delete node, let us call the node that need to be deleted as x.
Case 1: node x has no children. Just delete it (i.e. Change parent node so that it does not point to x)
Case 2: node x has one child. Splice out x by linking x's parent to x's child
Case 3: node x has two children. Splice out the x's successor and replace x with x's successor

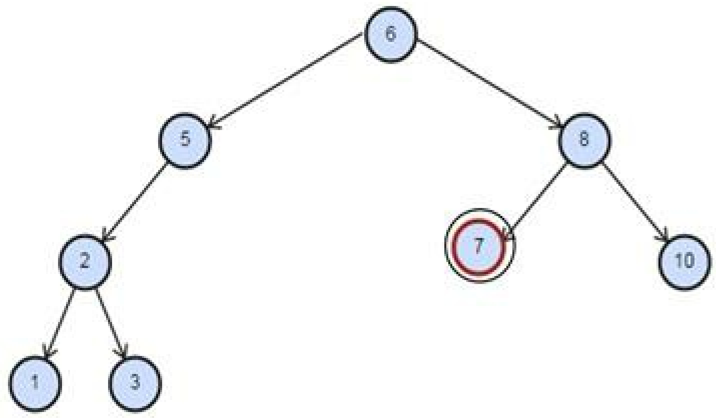
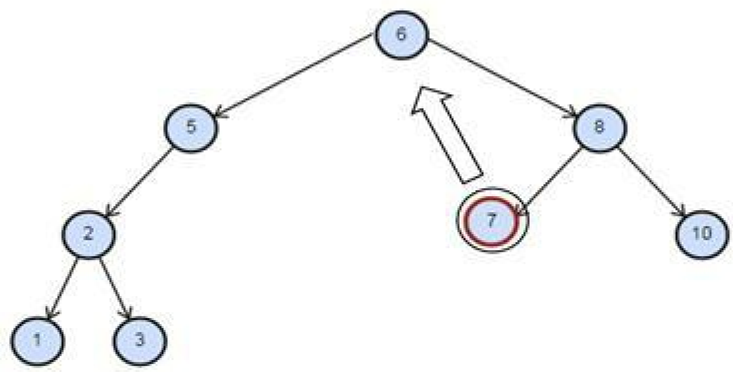
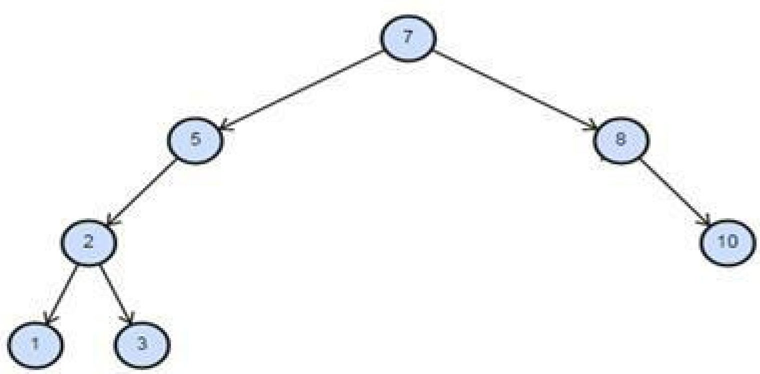
When the node to be deleted have no children
This is a trivial case, in which we directly delete the node and return null.

When the node to be deleted have only one child.
In this case, we save the child in a temp variable, then delete current node, and finally return the child.

	We want to remove node with value 9. The node has only one child.
	Right child of the parent of node with value 9 that is node with value 8 will point to the node with value 10.
	Finally, node with value 9 is removed from the tree.

When the node to be deleted has two children.

	We want to delete node with value 6. Which have two children.

	We had found minimum value node of the right child of node with value 6.
	Minimum value node value is copied to the node with value 6.
	Delete node with minimum value 7 is called over the right child tree of the node. Finally the tree with both the children is created.

Example 10.32:

```
func (t *Tree) DeleteNode(value int) {
    t.root = DeleteNode(t.root, value)
}

func DeleteNode(node *Node, value int) *Node {
    var temp *Node = nil
    if node != nil {
        if node.value == value {
            if node.left == nil && node.right == nil {
                return nil
            }
            if node.left == nil {
                temp = node.right
                return temp
            }
```

```
            if node.right == nil {
                temp = node.left
                return temp
            }
            maxNode := FindMax(node.left)
            maxValue := maxNode.value
            node.value = maxValue
            node.left = DeleteNode(node.left, maxValue)
        } else {
            if node.value > value {
                node.left = DeleteNode(node.left, value)
            } else {
                node.right = DeleteNode(node.right, value)
            }
        }
    }
    return node
}
```

Analysis: Time Complexity: O(n), Space Complexity: O(n)

Least Common Ancestor

In a tree T. The least common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants.

Example 10.33:

```
func (t *Tree) LcaBST(first int, second int) (int, bool) {
    return LcaBST(t.root, first, second)
}

func LcaBST(curr *Node, first int, second int) (int, bool) {
    if curr == nil {
        fmt.Println("NotFound")
        return 0, false
    }
    if curr.value > first && curr.value > second {
        return LcaBST(curr.left, first, second)
    }
    if curr.value < first && curr.value < second {
        return LcaBST(curr.right, first, second)
    }
    return curr.value, true
}
```

Trim the Tree nodes which are Outside Range

Given a range as min, max. We need to delete all the nodes of the tree that are out of this range.

Solution: Traverse the tree and each node that is having value outside the range will delete itself. All the deletion will happen from inside out so we do not have to care about the children of a node as if they are out of range then they already had deleted themselves.

Example 10.34:
```
func (t *Tree) TrimOutsidedataRange(min int, max int) {
	t.root = trimOutsidedataRange(t.root, min, max)
}

func trimOutsidedataRange(curr *Node, min int, max int) *Node {
	if curr == nil {
		return nil
	}
	curr.left = trimOutsidedataRange(curr.left, min, max)
	curr.right = trimOutsidedataRange(curr.right, min, max)
	if curr.value < min {
		return curr.right
	}
	if curr.value > max {
		return curr.left
	}
	return curr
}
```

Print Tree nodes which are in Range

Print only those nodes of the tree whose value is in the range given.
Solution: Just normal inorder traversal and at the time of printing we will check if the value is inside the range provided.

Example 10.35:
```
func (t *Tree) PrintDataInRange(min int, max int) {
	printDataInRange(t.root, min, max)
}

func printDataInRange(root *Node, min int, max int) {
	if root == nil {
		return
	}
	printDataInRange(root.left, min, max)
	if root.value >= min && root.value <= max {
		fmt.Print(root.value, " ")
	}
	printDataInRange(root.right, min, max)
}
```

Find Ceil and Floor value inside BST given key

Given a tree and a value we need to find the ceil value of node in tree which is smaller than the given value and need to find the floor value of node in tree which is bigger. Our aim is to find ceil and floor value as close as possible then the given value.

Example 10.36:
```
func (t *Tree) FloorBST(val int) int {
	curr := t.root
```

```
    floor := math.MaxInt32

    for curr != nil {
        if curr.value == val {
            floor = curr.value
            break
        } else if curr.value > val {
            curr = curr.left
        } else {
            floor = curr.value
            curr = curr.right
        }
    }
    return floor
}

func (t *Tree) CeilBST(val int) int {
    curr := t.root
    ceil := math.MinInt32

    for curr != nil {
        if curr.value == val {
            ceil = curr.value
            break
        } else if curr.value > val {
            ceil = curr.value
            curr = curr.left
        } else {
            curr = curr.right
        }
    }
    return ceil
}
```

Segment Tree

Segment tree is a binary tree that is used to make multiple range queries and range update in an array.

Examples of problems for which Segment Tree can be used are:
1. Finding the sum of all the elements of an array in a given range of index
2. Finding the maximum value of the array in a given range of index.
3. Finding the minimum value of the array in a given range of index (also known as Range Minimum Query problem)

Properties of Segment Tree:
1. Segment tree is a binary tree.
2. Each node in a segment tree represent an interval in the array.
3. The root of tree represent the whole array.
4. Each leaf node of represent a single element.

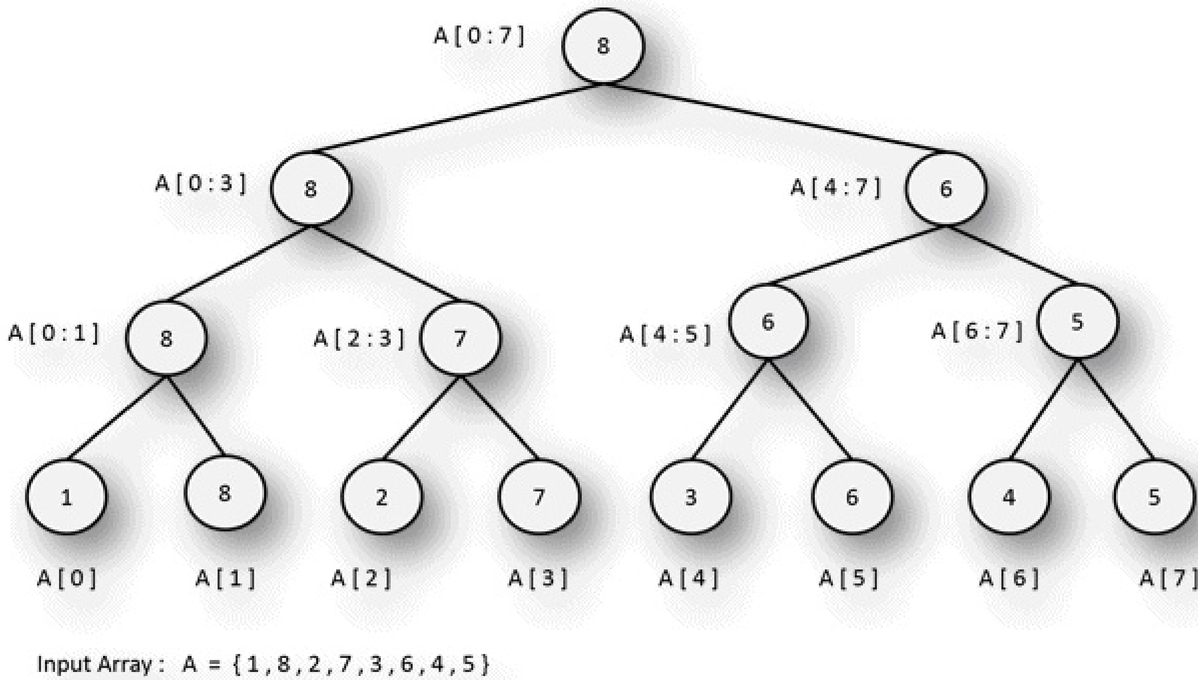
Note:- Segment tree solve problems, which can be solve in linear time by just scanning and updating the elements of array. The only benefit we are getting from segment tree is that it does update and query operation in logarithmic time that is more efficient than the linear approach.

Let us consider a simple problem:
Given an array of N numbers. You need to perform the following operations:
1. Update any element in the array
2. Find the maximum in any given range (i, j)

Solution 1:
Updating: Just update the element in the array, a[i] =x. Finding maximum in the range (i, j), by traversing through the elements of the array in that range.
Time Complexity of Update is O(1) and of Finding is O(n)

Solution 2: The above solution is good. However, can we improve performance of Finding?
The answer is yes. In fact we can do both the operation in O(log n) where n is the size of the array. This we can do using a segment tree.

Let us suppose we are given an input array A = {1, 8, 2, 7, 3, 6, 4, 5}. Moreover, the below diagram will represent the segment tree formed corresponding to the input array A.

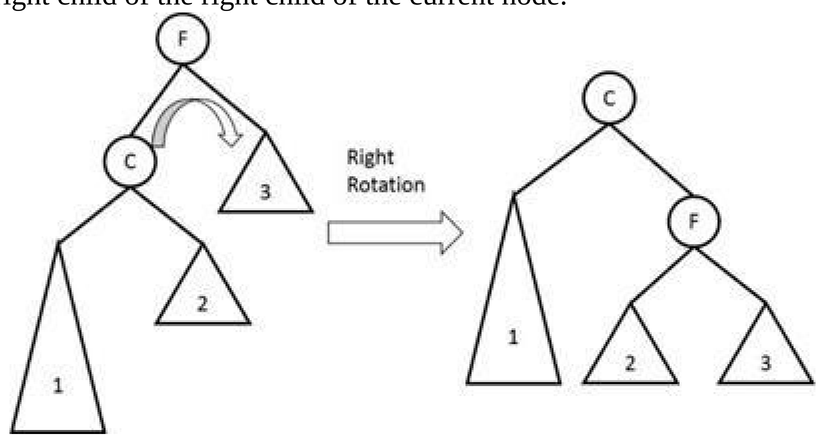

A[0:7]
8
A[0:3]
8
A[4:7]
6
A[0:1]
8
A[2:3]
7
A[4:5]
6
A[6:7]
5
1
8
2
7
3
6
4
5
A[0]
A[1]
A[2]
A[3]
A[4]
A[5]
A[6]
A[7]
Input Array : A = {1,8,2,7,3,6,4,5}

AVL Trees

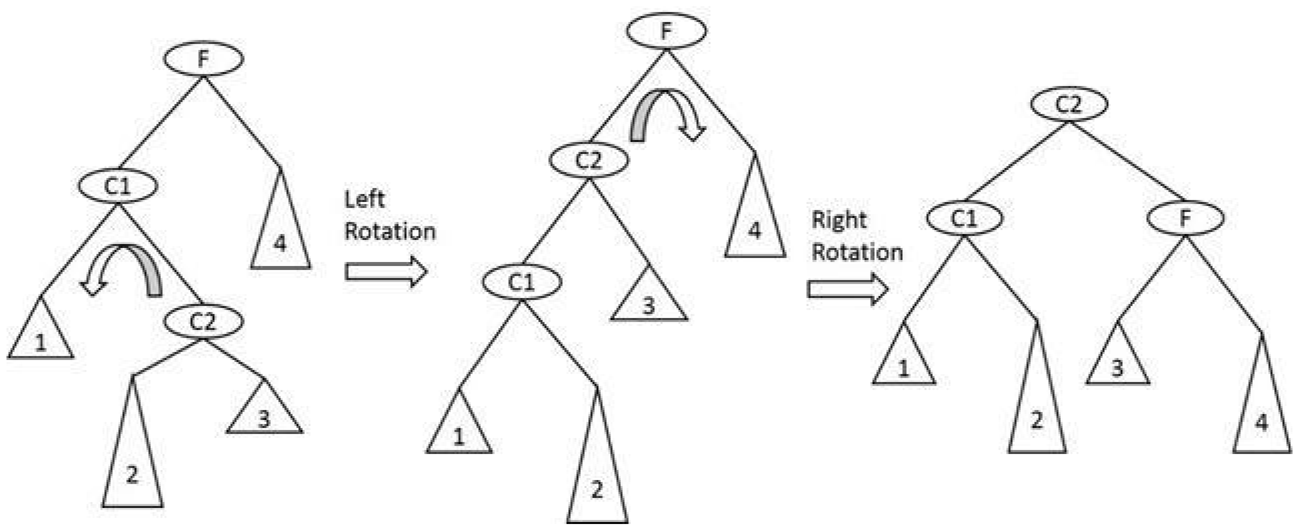
An AVL tree is a binary search tree (BST) with an additional property that the subtrees of every node differ in height by at most one. An AVL tree is a height balanced BST.

AVL tree is a balanced binary search tree. Adding or removing a node form AVL tree may make the AVL tree unbalanced. Such violations of AVL balance property is corrected by one or two simple steps called rotations. Let us assume that insertion of a new node had converted a previously balanced AVL tree into an unbalanced tree. Since the tree is previously balanced and a single new node is added to it, maximum the unbalance difference in height will be 2.

Therefore, in the bottom most unbalanced node there are only four cases:
Case 1: The new node is left child of the left child of the current node.
Case 2: The new node is right child of the left child of the current node.
Case 3: The new node is left child of the right child of the current node.
Case 4: The new node is right child of the right child of the current node.



Case 1 can be re-balanced using a single Right Rotation.
Case 4 is symmetrical to Case 1: can be re-balanced using a single Left Rotation



Case 2 can be re-balanced using a double rotation. First, rotate left than rotation right.
Case 3 is symmetrical to Case 2: can be re-balanced using a double rotation. First, rotate right than rotation left.

Time Complexity of Insertion: To search the location where a new node need to be added in done in O(log(n)). Then on the way back, we look for the AVL balanced property and fixes them with rotation. Since the rotation at each node is done in constant time, the total amount of word is proportional to the length of the path. Therefore, the final time complexity of insertion is O(log(n)).

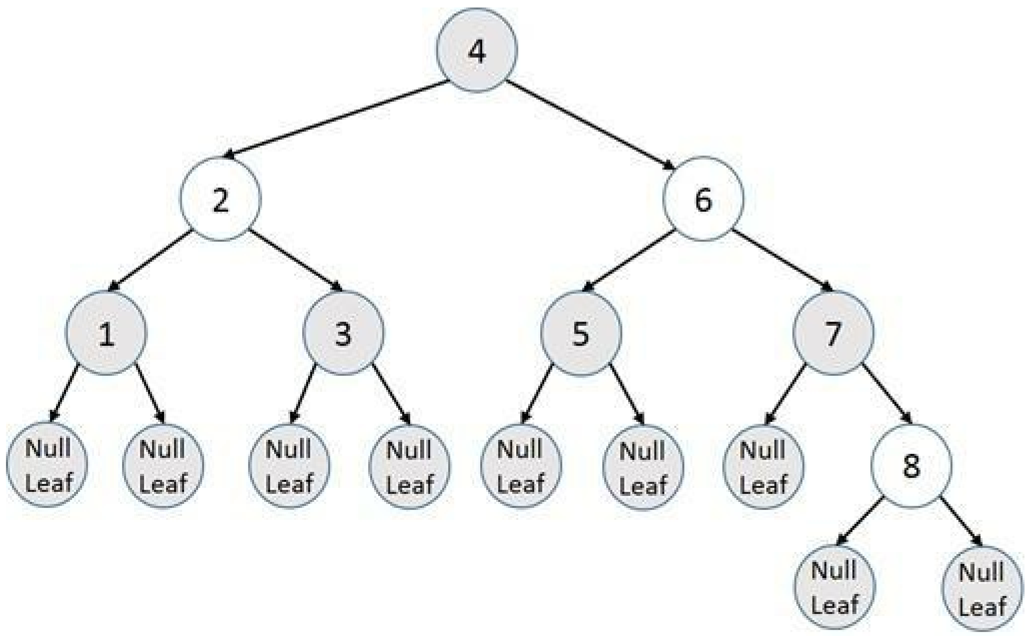
Red-Black Tree

The red-black tree contains its data, left and right children like any other binary tree. In addition to this its node also contains an extra bit of information which represents colour which can either red or black. Red-Black tree also contains a specialized class of nodes called NULL nodes. NULL nodes are pseudo nodes that exists at the leaf of the tree. All internal nodes has data associated with them.

Red-Black tree have the following properties:

1. Root of tree is black.
2. Every leaf node (NULL node) is black.
3. If a node is red then both of its children are black.
4. Every path from a node to a descendant leaf contains the same number of black nodes.

The first three properties are self-explanatory. The forth property states that, from any node in the tree to any leaf (NULL), the number of black nodes must be the same.



In the above figure, from the root node to the leaf node (NULL) the number of black node is always three nodes.

Like the AVL tree, red-black trees are also self-balancing binary search tree. Whereas the balance property of an AVL tree was a direct relationship between the heights of left and right subtrees of each node. In red-black trees, the balancing property is governed by the four rules mentioned above. Adding or removing a node form red-black tree may violate the properties of a red-black tree. The red-black properties are restored through recolouring and rotation. Insert, delete, and search operation time complexity is O(log(n))

Splay tree

A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time.
Elements of the tree are rearranged so that the recently accessed element is placed at the top of the tree. When an element is searched then we use standard BST search and then use rotation to bring the element to the top.

Splay tree	Average Case	Worst Case
Space complexity	O(n)	O(n)
Time complexity search	O(log(n))	Amortized O(log(n))
Time complexity insert	O(log(n))	Amortized O(log(n))
Time complexity delete	O(log(n))	Amortized O(log(n))

Unlike the AVL tree, the splay tree is not guaranteed to be height balanced. What is guaranteed is that the total cost of the entire series of accesses will be cheap.

B-Tree

As we had already seen various types of binary tree for searching, insertion and deletion of data in the main memory. However, these data structures are not appropriate for huge data that cannot fit into main memory, the data that is stored in the disk.

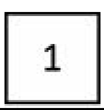
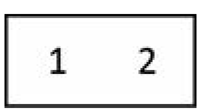
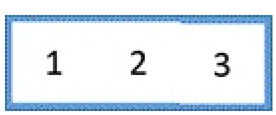
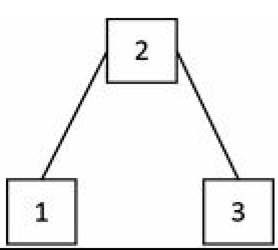
A B-tree is a self-balancing search tree that allows searches, insertions, and deletions in logarithmic time. The B-tree is a tree in which a node can have multiple children. Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write entire blocks (page) of data. The read - write operation from disk is very slow as compared with the main memory. The main purpose of B-Tree is to reduce the number of disk access. The node in a B-Tree is having a huge number of references to the children nodes. Thereby reducing the size of the tree. While accessing data from disk, it make sense to read an entire block of data and store into a node of tree. B-Tree nodes are designed such that entire block of data (page) fits into it. It is commonly used in databases and filesystems.

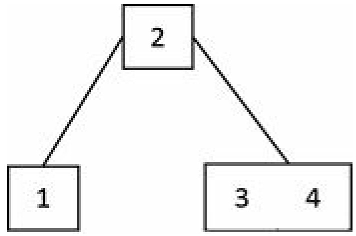
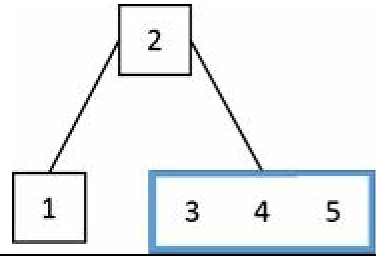
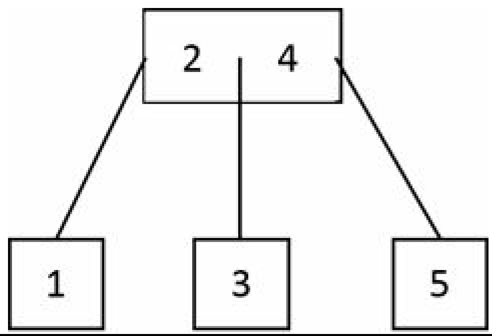
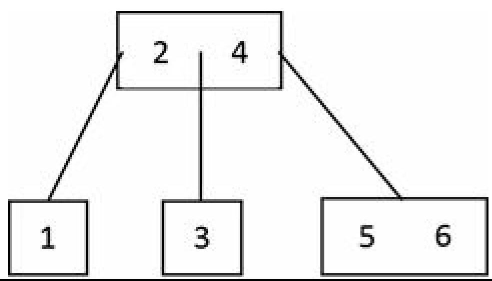
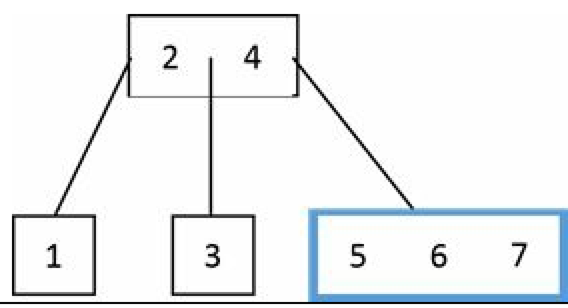
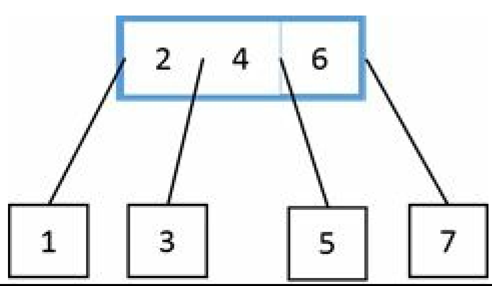
B-Tree of minimum degree d has the following properties:

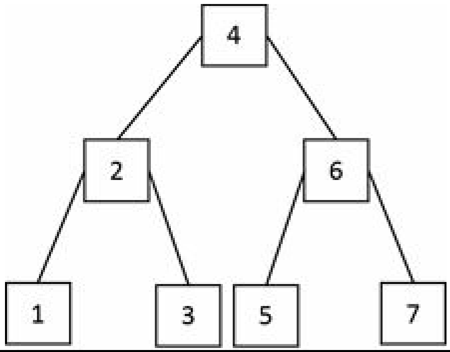
1. All the leaf nodes must be at same level.
2. All nodes except root must have at least (d-1) keys and maximum of (2d-1) keys. Root may contain minimum 1 key.
3. If the root node is a non-leaf node, then it must have at least 2 children.
4. A non-leaf node with N keys must have (N+1) number of children.
5. All the key values within a node must be in Ascending Order.
6. All keys of a node are sorted in ascending order. The child between two keys, K1 and K2 contains all keys in range from K1 and K2.

B-Tree	Average Case	Worst Case
Space complexity	O(n)	O(n)
Time complexity search	O(log(n))	O(log(n))
Time complexity insert	O(log(n))	O(log(n))
Time complexity delete	O(log(n))	O(log(n))

Below is the steps of creation of B-Tree by adding value from 1 to 7.

1		Insert 1 to the tree.	Stable
2		Insert 2 to the tree.	Stable
3		Insert 3 to the tree.	Intermediate
4		New node is created and data is distributed.	Stable

5		Insert 4 to the tree.	Stable
6		Insert 5 to the tree.	Intermediate
7		New node is created and data is distributed.	Stable
8		Insert 6 to the tree.	Stable
9		Insert 7 to the tree. New node is created and data is distributed.	Intermediate
10		After rearranging the intermediate node is, also have more than maximum number of keys.	Intermediate

11		New node is created and data is distributed. The height of the tree is increased.	Stable

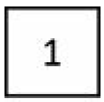
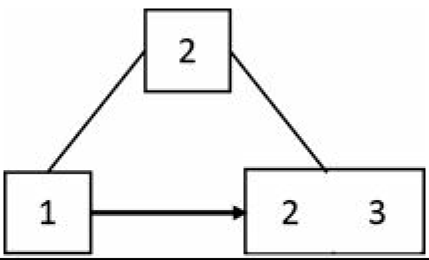
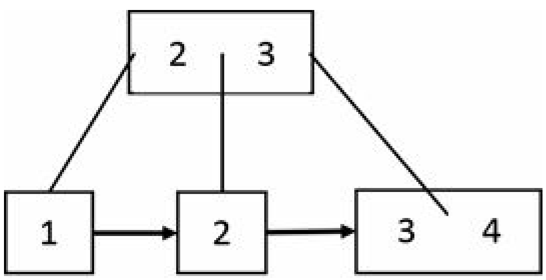
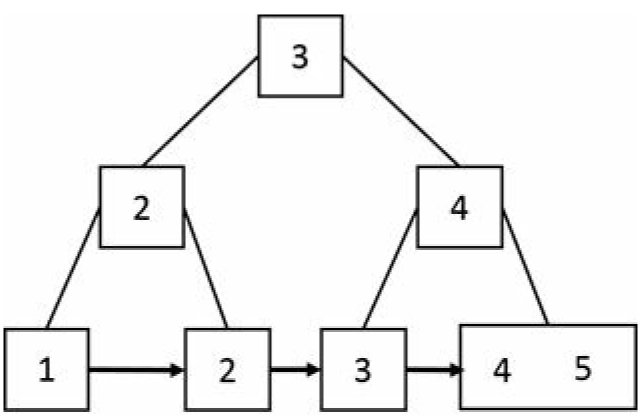
Note:- 2-3 tree is a B-tree of degree three.

B+ Tree

B+ Tree is a variant of B-Tree. The B+ Tree store records only at the leaf nodes. The internal nodes store keys. These keys are used in insertion, deletion and search. The rules of splitting and merging of nodes is same as B-Tree.

b-order B+ tree	Average Case	Worst Case
Space complexity	O(n)	O(n)
Time complexity search	$O(\log_b(n))$	$O(\log_b(n))$
Time complexity insert	$O(\log_b(n))$	$O(\log_b(n))$
Time complexity delete	$O(\log_b(n))$	$O(\log_b(n))$

Below is the B+ Tree created by adding value from 1 to 5.

1.		Value 1 is inserted to leaf node.
2.		Value 2 is inserted to leaf node.
3.		Value 3 is inserted to leaf node. Content of the leaf node passed the maximum number of elements. Therefore, node is split and intermediate / key node is created.
4.		Value 4 is further inserted to the leaf node. Which further split the leaf node.
5.		Value 5 is added to the leaf node the number of nodes in the leaf passed the maximum number of nodes that it can contain so it is divided into 2. One more key is added to the intermediate node, which make it also, passed maximum number of nodes it can contain, and finally divided and a new node is created.

B* Tree

The B* tree is identical to the B+ tree, except for the rules for splitting and merging of nodes. Instead of splitting a node into two halves when it overflows, the B* tree node tries to gives some of its records to its neighbouring sibling. If the sibling is also full, then a new node is created and records are distributed into three.

Exercise

1. Construct a tree given its in-order and pre-order traversal strings.
    - inorder: 1 2 3 4 5 6 7 8 9 10
    - pre-order: 6 4 2 1 3 5 8 7 9 10

2. Construct a tree given its in-order and post-order traversal strings.
    - inorder: 1 2 3 4 5 6 7 8 9 10
    - post-order: 1 3 2 5 4 7 10 9 8 6

3. Write a delete node function in Binary tree.

4. Write a function print depth first in a binary tree without using system stack
   Hint: you may want to keep another element to tree node like visited flag.

5. The worst-case runtime Complexity of building a BST with n nodes
    - $O(n^2)$
    - O(n * log n)
    - O(n)
    - O(logn)

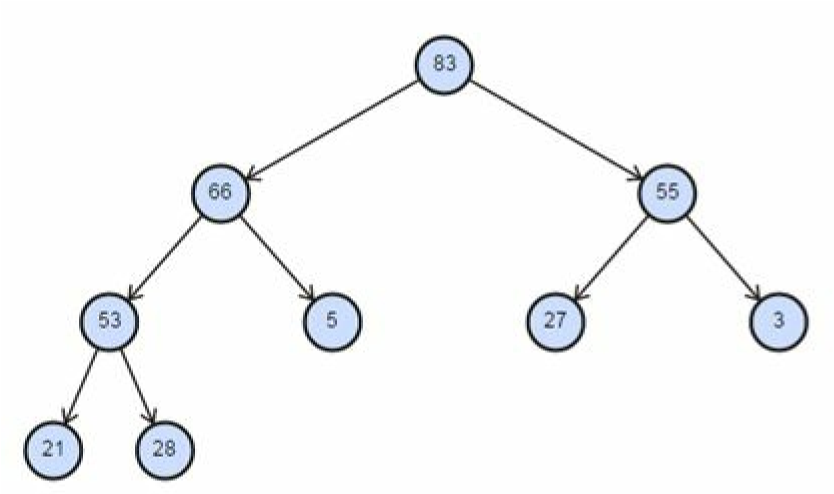
6. The worst-case runtime Complexity of insertion into a BST with n nodes is
    - O(n2)
    - O(n * log n)
    - O(n)
    - O(logn)

7. The worst-case runtime Complexity of a search of a value in a BST with n nodes.
    - $O(n^2)$
    - O(n * log n)
    - O(n)
    - O(logn)

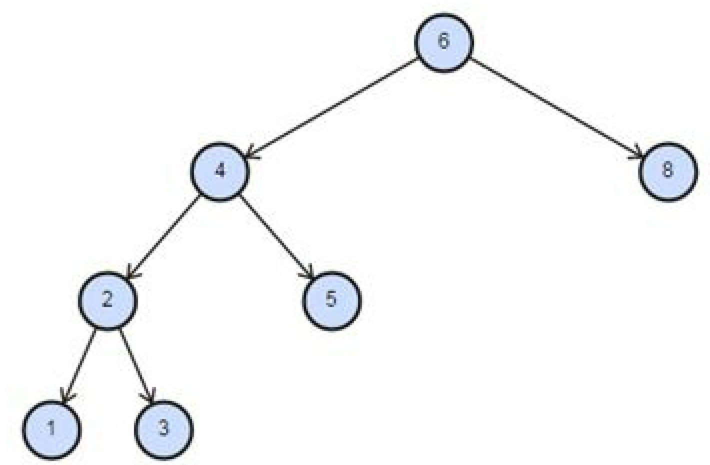
8. Which of the following traversals always gives the sorted sequence of the elements in a BST?
    - Preorder
    - Ignored
    - Postorder
    - Undefined

9. The height of a Binary Search Tree with n nodes in the worst case?
    - O(n * log n)
    - O(n)
    - O(logn)
    - O(1)
    -

10. Check whether a given Binary Tree is Complete or not
    - In a complete binary tree, every level except the last one is completely filled. All nodes in the left are filled first, then the right one.

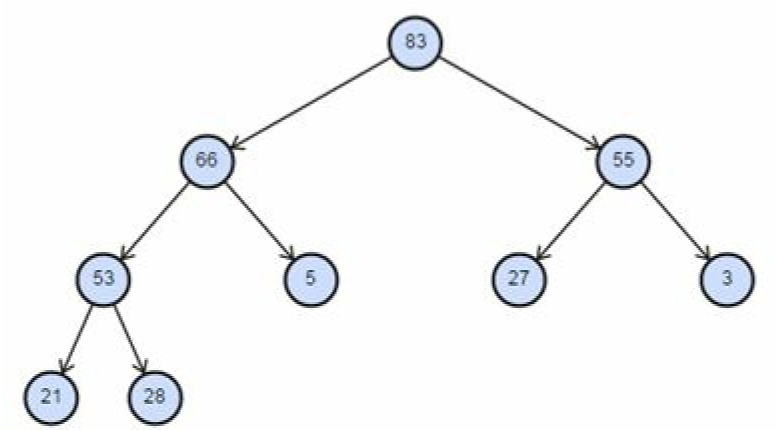


11. Check whether a given Binary Tree is Full/ Strictly binary tree or not. The full binary tree is a binary tree in which each node has zero or two children.



12. Check whether a given Binary Tree is a Perfect binary tree or not. The perfect binary tree- is a type of full binary trees in which each non-leaf node has exactly two child nodes.

13. Check whether a given Binary Tree is Height-balanced Binary Tree or not. A height-balanced binary tree is a binary tree such that the left & right subtrees for any given node differ in height by no more than one



14. Isomorphic: two trees are isomorphic if they have the same shape, it does not matter what the value is. Write a program to find if two given tree are isomorphic or not.

15. Try to optimize the above solution to give a DFS traversal without using recursion use some stack or queue.

16. This is an open exercise for the readers. Every algorithm that is solved using recursion (system stack) can also be solved using user defined or library defined stack. So try to figure out what all algorithms that are using recursion and try to figure out how you will do this same issue using user layer stack.

17. In a binary tree, print the nodes in zigzag order. In the first level, nodes are printed in the left to right order. In the second level, nodes are printed in right to left and in the third level again in the order left to right.
    Hint: Use two stacks. Pop from first stack and push into another stack. Swap the stacks alternatively.

18. Find nth smallest element in a binary search tree.
    Hint: Nth inorder in a binary tree.

19. Find the floor value of key that is inside a BST.

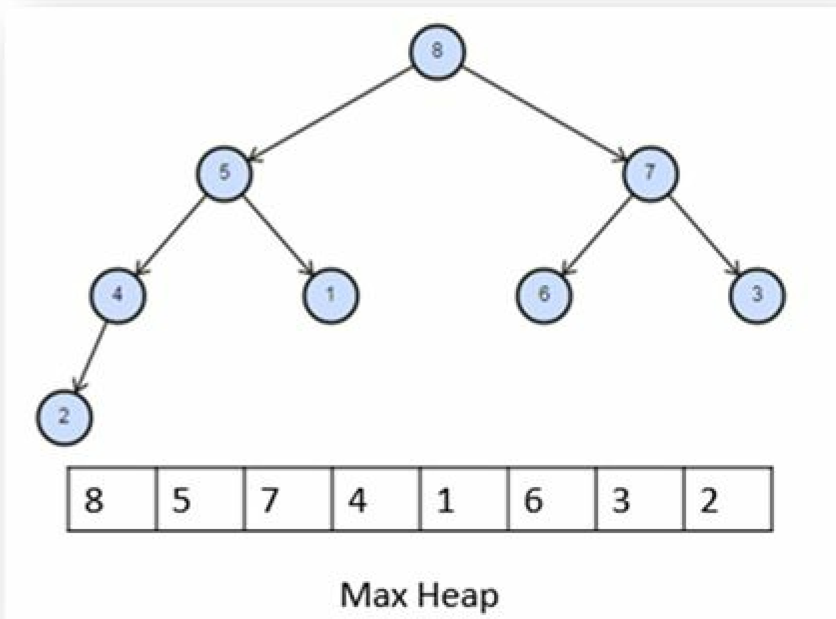
20. Find the Ceil value of key, which is inside a BST.

CHAPTER 11: PRIORITY QUEUE

Introduction

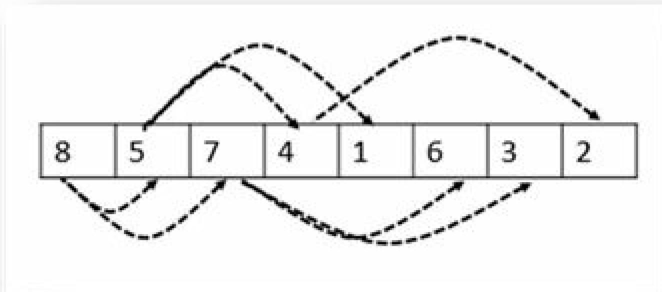
A Priority-Queue also knows as Binary-Heap, is a variant of queue. Items are removed from the start of the queue. However, in a Priority-Queue the logical ordering of objects is determined by their priority. The highest priority item are at the front of the Priority-Queue. When you add an item to the Priority-Queue, the new item can more to the front of the queue. A Priority-Queue is a very important data structure. Priority-Queue is used in various Graph algorithms like Prim's Algorithm and Dijkstra's algorithm. Priority-Queue is also used in the timer implementation etc.

A Priority-Queue is implemented using a Heap (Binary Heap). A Heap data structure is an array of elements that can be observed as a complete binary tree. The tree is completely filled on all levels except possibly the lowest. Heap satisfies the heap ordering property. In max-heap, the parent's value is greater than or equal to its children value. In min-heap, the parent's value is less than or equal to its children value. A heap is a complete binary tree so the height of tree with N nodes is always **O(logn)**.



A heap is not a sorted data structure and can be regarded as partially ordered. As you see from the picture, there is no relationship among nodes at any given level, even among the siblings.

Heap is implemented using an array. Moreover, because heap is a complete binary tree, the left child of a parent (at position x) is the node that is found in position 2x in the array. Similarly, the right child of the parent is at position 2x+1 in the array. To find the parent of any node in the heap, we can simply division. Given the index y of a node, the parent index will by y/2.

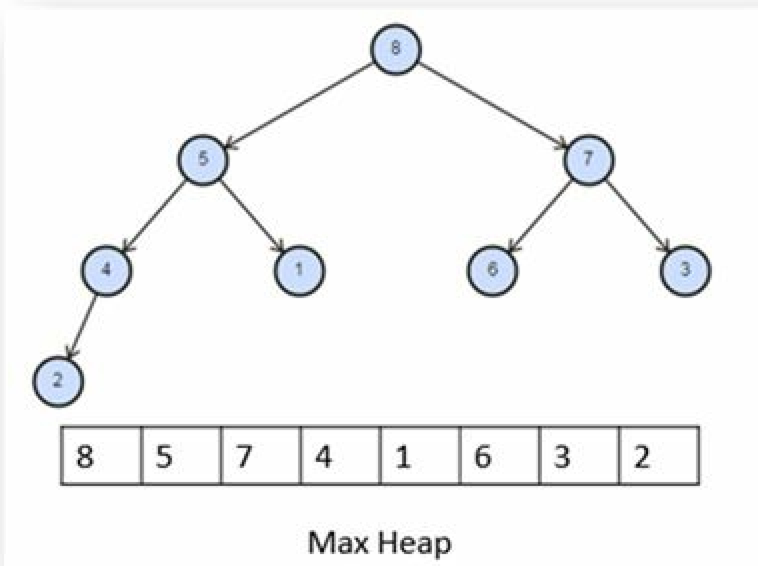

8
5
7
4
1
6
3
2

Types of Heap

There are two types of heap and the type depends on the ordering of the elements. The ordering can be done in two ways: Min-Heap and Max-Heap

Max Heap

Max-Heap: the value of each node is less than or equal to the value of its parent, with the largest-value element at the root.

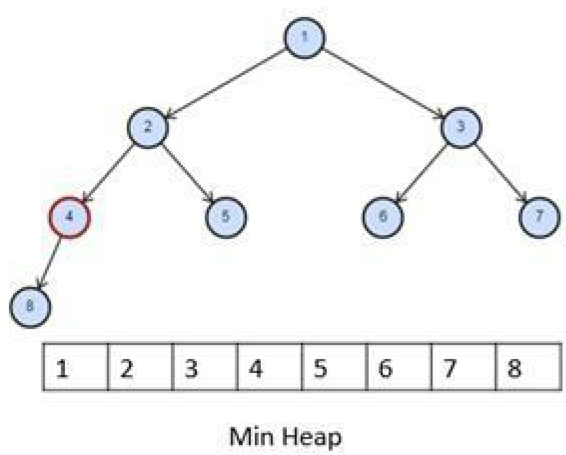


Max Heap Operations

Insert	O(logn)
DeleteMax	O(logn)
Remove	O(logn)
FindMax	O(1)

Min Heap

Min-Heap: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.



Use it whenever you need quick access to the smallest item, because that item will always be at the root of the tree or the first element in the list. However, the remainder of the list is kept partially sorted. Thus, instant access is only possible for the smallest item.

Min Heap Operations

Insert	O(logn)
DeleteMin	O(logn)
Remove	O(logn)
FindMin	O(1)

Throughout this chapter, the word "heap" will always refer to a max-heap. The implementation of min-heap is left for the user to do it as an exercise.

Heap ADT Operations

The basic operations of binary heap are as follows:

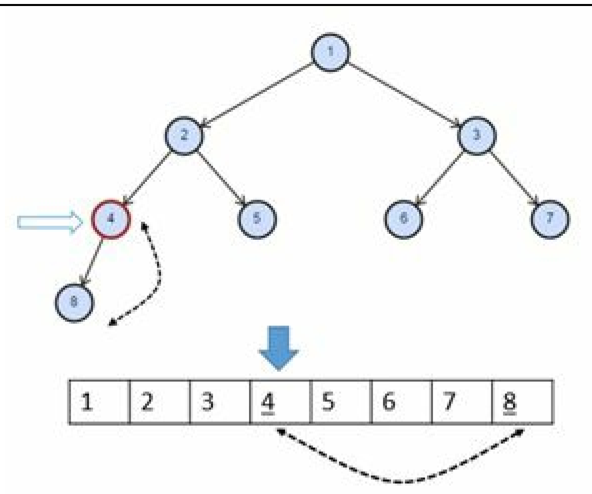
Binary Heap	Create a new empty binary heap	O(1)
Insert	Adding a new element to the heap	O(logn)
DeleteMax	Delete the maximum element form the heap.	O(logn)
FindMax	Find the maximum element in the heap.	O(1)
isEmpty	return true if the heap is empty else return false	O(1)
Size	Return the number of elements in the heap.	O(1)
BuildHeap	Build a new heap from the list of elements	O(logn)

Operation on Heap

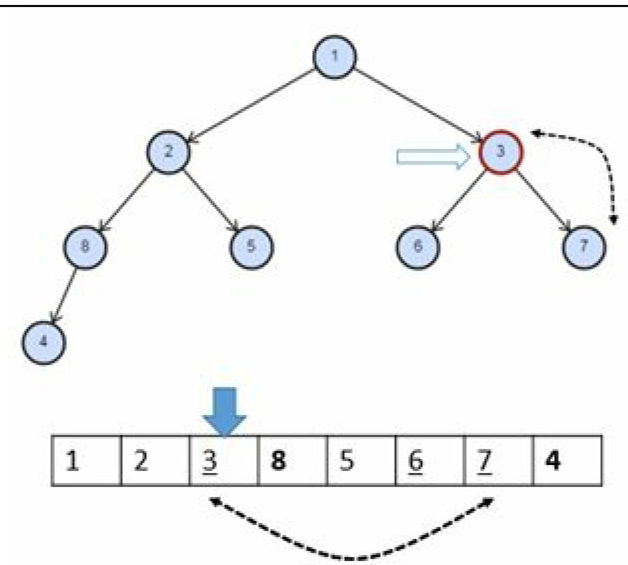
Create Heap from a list

Heapify is the process of converting a list into Heap. The various steps are:

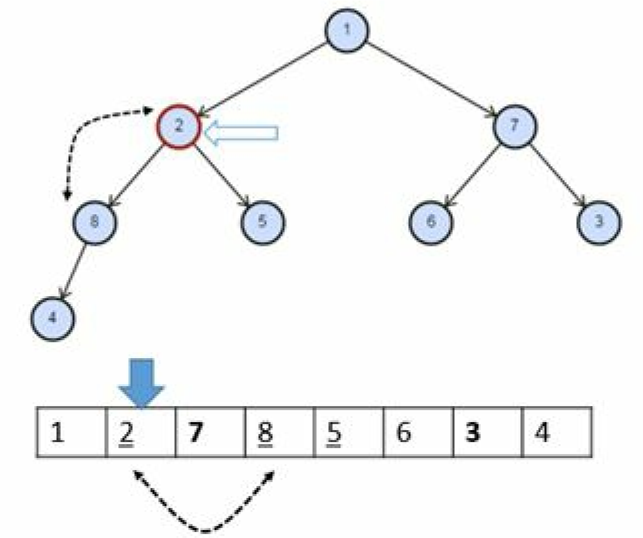
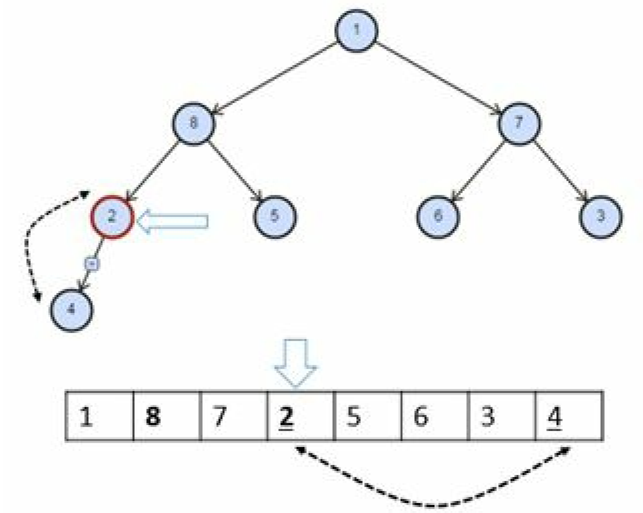
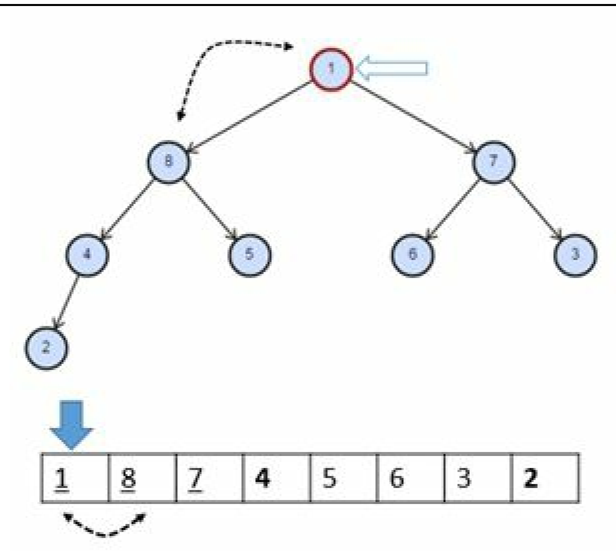
1. Values are present in list.
2. Starting from the middle of the list move downward towards the start of the list. At each step, compare parent value with its left child and right child. In addition, restore the heap property by shifting the parent value with its largest-value child. Such that the parent value will always be greater than or equal to left child and right child.
3. For all elements from middle of the list to the start of the list. We are doing comparisons and shift, until we reach the leaf nodes of the heap. The Time Complexity of build heap is **O(N)**.

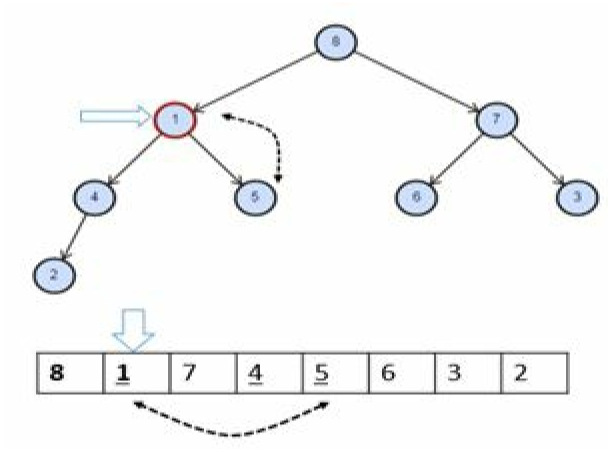
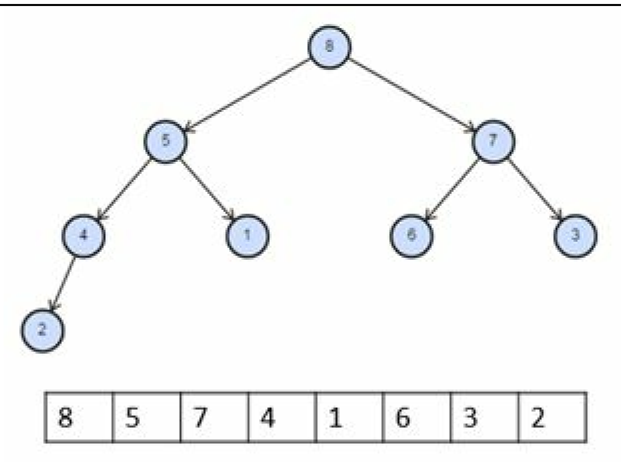


Given a list as input to create heap function. Value of index i is compared with value of its children nodes that is at index (i*2 + 1) and (i*2 + 2). Middle of list N/2 that is index 3 is comapred with index 7. If the children node value is greater than parent node then the value will be swapped.



Similarly, value of index 2 is compared with index 5 and 6. The largest of the value is 7 which will be swapped with the value at the index 2.

	Similarly, value of index 1 is compared with index 3 and 4 The largest of the value is 8 which will be swapped with the value at the index 1.
	Percolate down function is used to subsequently adjest the value replased in the previous step by comparing it with its children nodes.
	Now value at index 0 is comared with index 1 and 2. 8 is the largest value so it swapped with the value at index 0.
	Percolate down function is used to further compare the value at

	index 1 with its children nodes at index 3 and 4.
	In the end max heap is created.

Example 11.1:

```
type Heap struct {
    size  int
    arr   []int
    isMin bool
}

func NewHeap(arrInput []int, isMin bool) *Heap {
    size := len(arrInput)
    arr := []int{1}
    arr = append(arr, arrInput...)
    h := &Heap{size: size, arr: arr, isMin: isMin}
    for i := (h.size / 2); i > 0; i-- {
        h.proclateDown(i)
    }
    return h
}
```

```
func (h *Heap) proclateDown(parent int) {
    lChild := 2 * parent
    rChild := lChild + 1
    small := -1
    if lChild <= h.size {
        small = lChild
    }
    if rChild <= h.size && h.comp(lChild, rChild) {
```

```
            small = rChild
      }
      if small != -1 && h.comp(parent, small) {
            h.swap(parent, small)
            h.proclateDown(small)
      }
}

func NewHeap2(isMin bool) *Heap {
      arr := []int{1}
      h := &Heap{size: 0, arr: arr, isMin: isMin}
      return h
}

func (h *Heap) comp(i, j int) bool { // always i < j in use
      if h.isMin == true {
            return h.arr[i] > h.arr[j] // swaps for min heap
      }
      return h.arr[i] < h.arr[j] // swap for max heap.
}

func (h *Heap) swap(i, j int) {
      h.arr[i], h.arr[j] = h.arr[j], h.arr[i]
}

func (h *Heap) Empty() bool {
      return (h.size == 0)
}

func (h *Heap) Size() int {
      return h.size
}
func (h *Heap) Peek() (int, bool) {
      if h.Empty() {
            fmt.Println("Heap empty Error.")
            return 0, false
      }
      return h.arr[1], true
}
```

Initializing an empty Heap

Example 11.2:

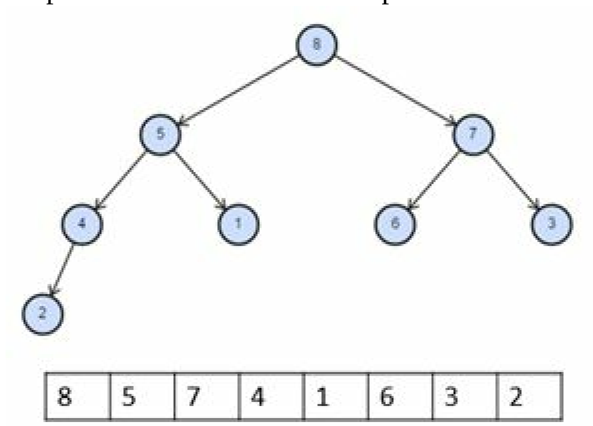
```
func (h *Heap) proclateUp(child int) {
      parent := child / 2
      if parent == 0 {
            return
      }
      if h.comp(parent, child) {
            h.swap(child, parent)
            h.proclateUp(parent)
      }
```

```
}
```

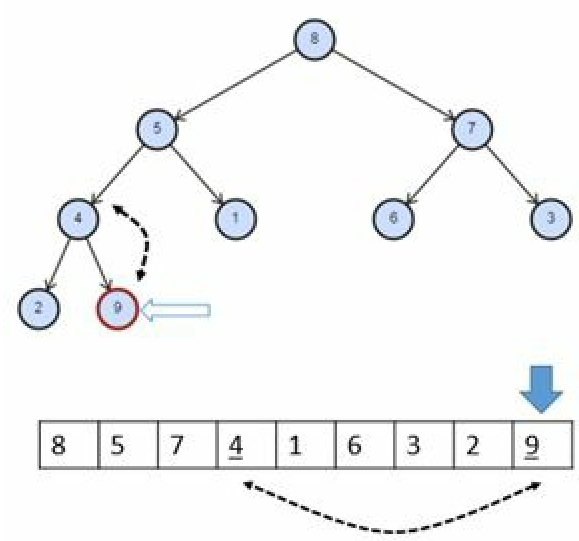
Enqueue / Insert

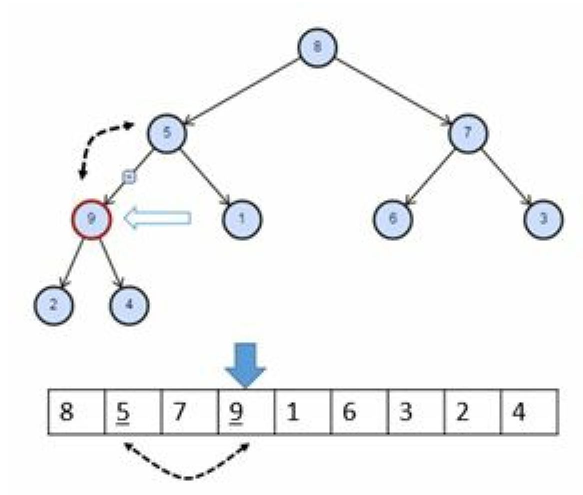
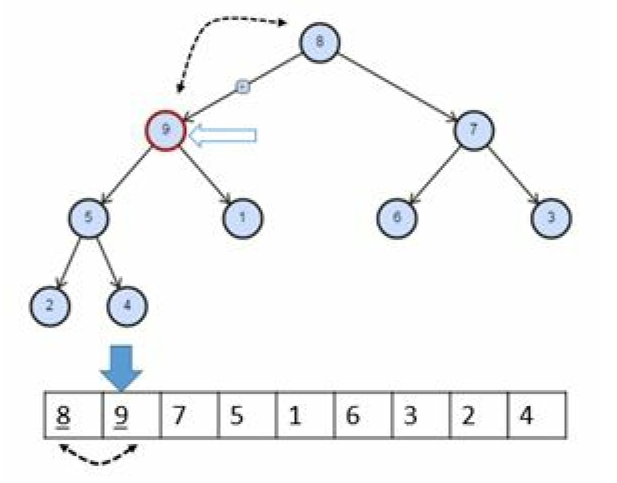
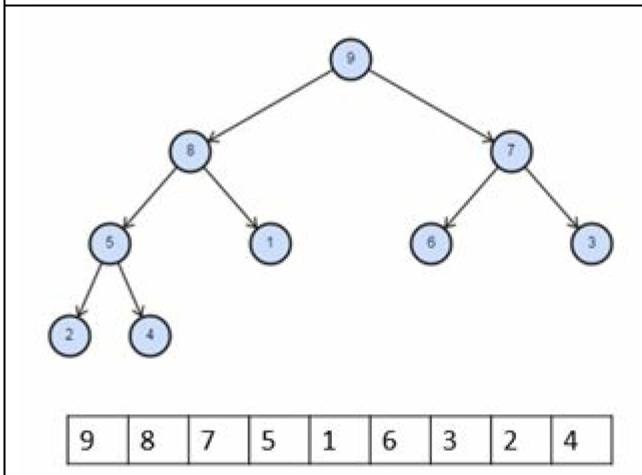
1. Add the new element at the end of the list. This keeps the structure as a complete binary tree, but it might no longer be a heap since the new element might have a value greater than its parent's value.
2. Swap the new element with its parent until it has value greater than its parent's value.
3. Step 2 will terminate when the new element reaches the root or when the new element's parent have a value greater than or equal to the new element's value.

Let us take an example of the Max heap created in the above example.



Let us take an example by inserting element with value 9 to the heap. The element is added to the end of the heap list. Now the value will be percolate up by comparing it with the parent. The value is added to index 8 and its parent will be (N-1)/2 = index 3.

	Since the value 9 is greater thn 4 it will be swapped with it.
	Percolate up is used and the value is moved up until heap property is

	satisfied.
	Now the value at index 1 is compared with index 0 and to satisfy heap property it is further swapped.
	Now finally max heap is created by inserting new node.

Example 11.3:

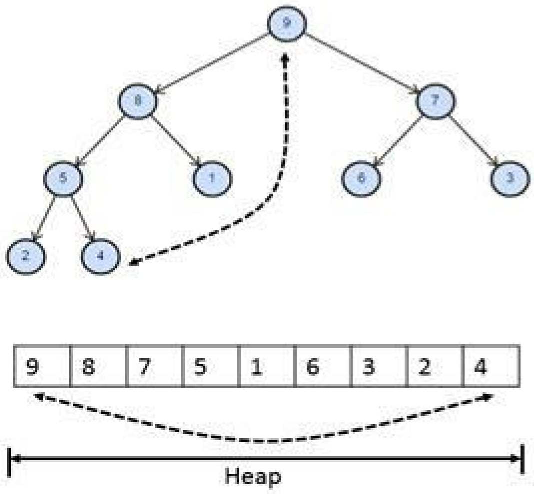
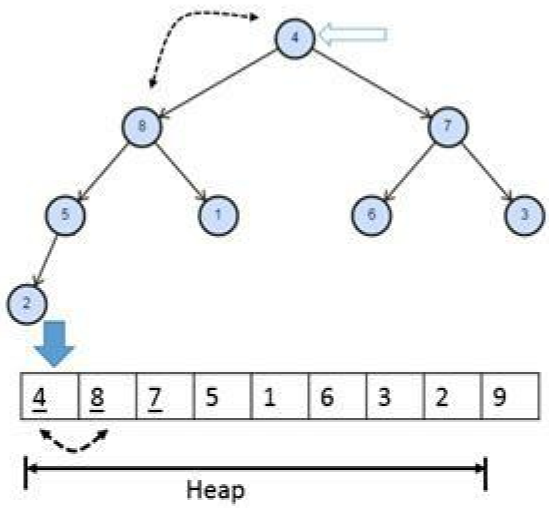
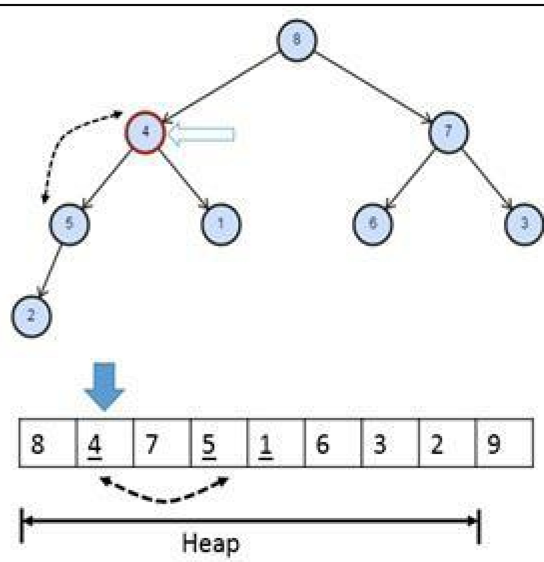
```
func (h *Heap) Add(value int) {
    h.size++
    h.arr = append(h.arr, value)
    h.proclateUp(h.size)
}
```

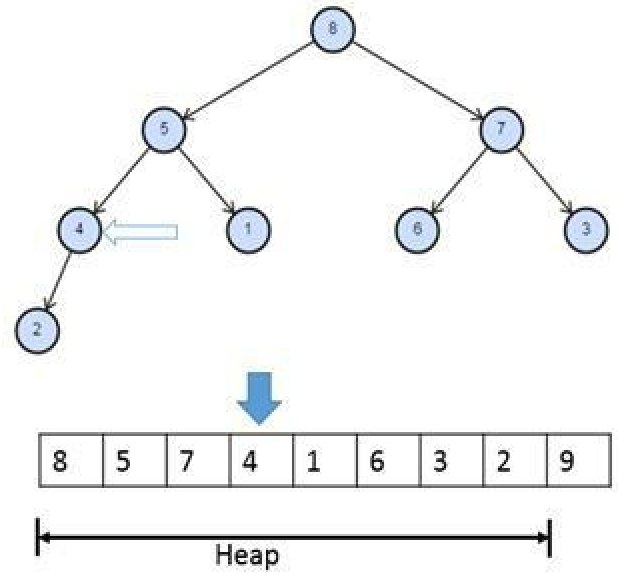
Dequeue / Delete

1. Copy the value at the root of the heap to the variable used to return a value.

2. Copy the last element of the heap to the root, and then reduce the size of heap by 1. This element is called the "out-of-place" element.
3. Restore heap property by swapping the out-of-place element with its largest-value child. Repeat this process until the out-of-place element reaches a leaf or its value is greater or equal to all its children.
4. Return the answer that was saved in Step 1.

To remove an element from heap its top & end value is swapped and size of heap is reduced by 1.

	Since value at end of the heap is copied to head of heap. Heap property is disturbed so we need to percolate down by comparing node with its children nodes and restore heap property.
	Percolate down continued by comparing with its children nodes.
	Percolate down

	Percolate down Complete

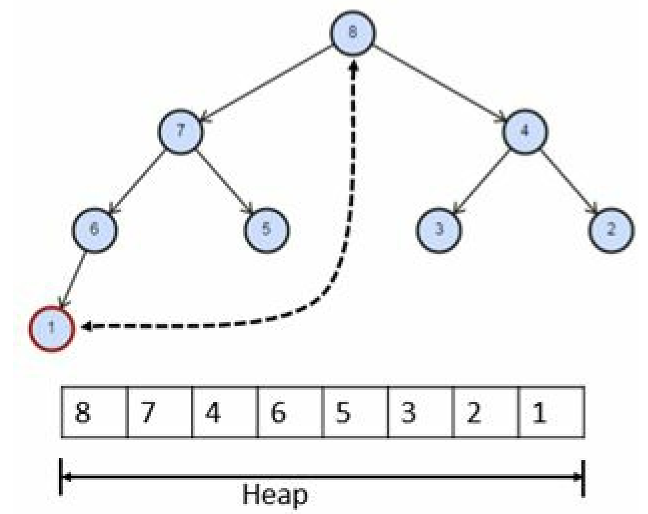
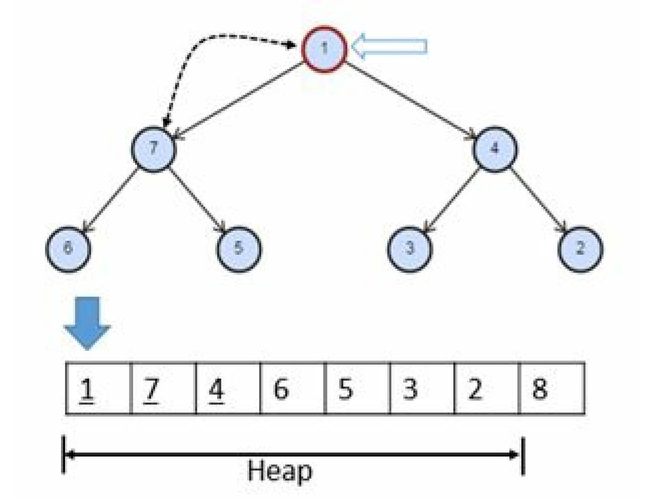
Example 11.4:

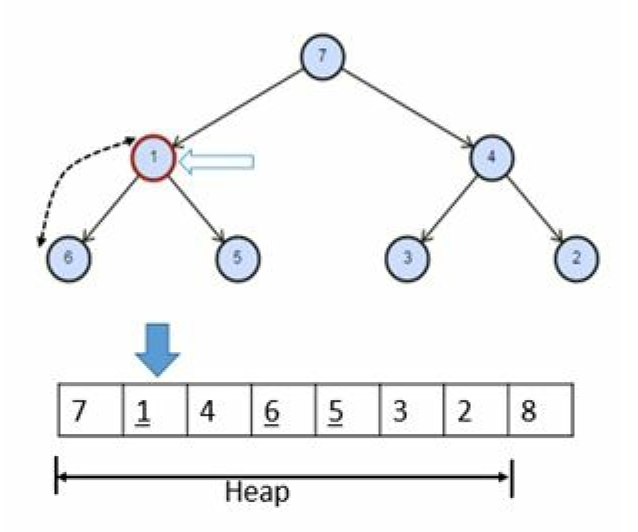
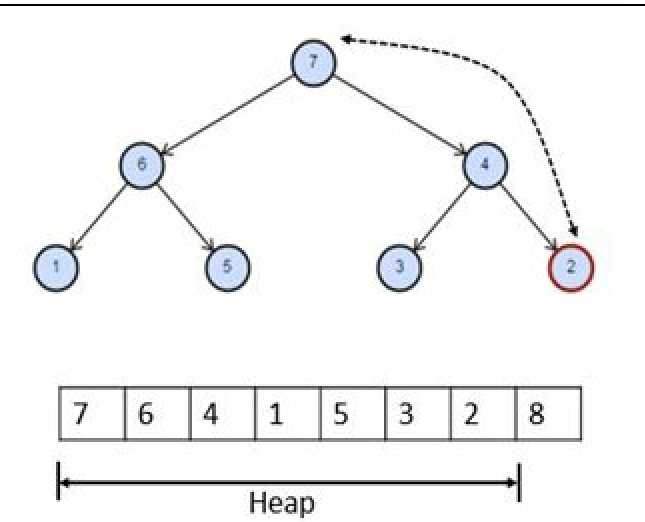
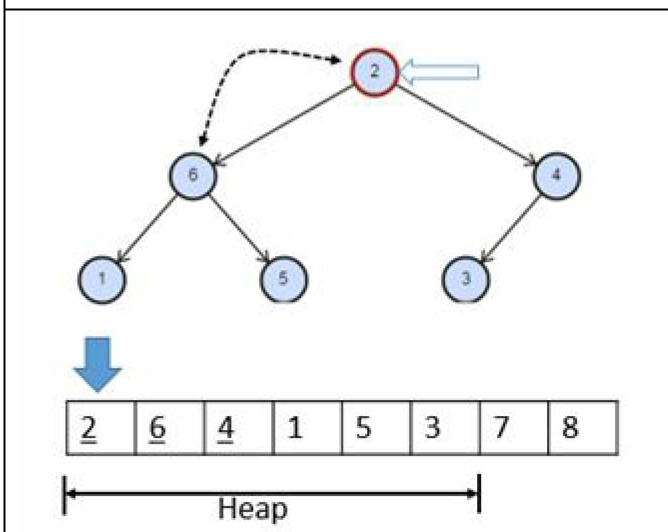
```
func (h *Heap) Remove() (int, bool) {
	if h.Empty() {
		fmt.Println("HeapEmptyError.")
		return 0, false
	}
	value := h.arr[1]
	h.arr[1] = h.arr[h.size]
	h.size--
	h.proclateDown(1)
	h.arr = h.arr[0 : h.size+1]
	return value, true
}

func main() {
	a := []int{1, 9, 6, 7, 8, -1, 2, 4, 5, 3}
	hp := NewHeap(nil, true)
	// hp := NewHeap(a, true)
	n := len(a)
	for i := 0; i < n; i++ {
		hp.Add(a[i])
	}
	for i := 0; i < n; i++ {
		fmt.Println("pop value :: ", hp.Remove())
	}
}
```

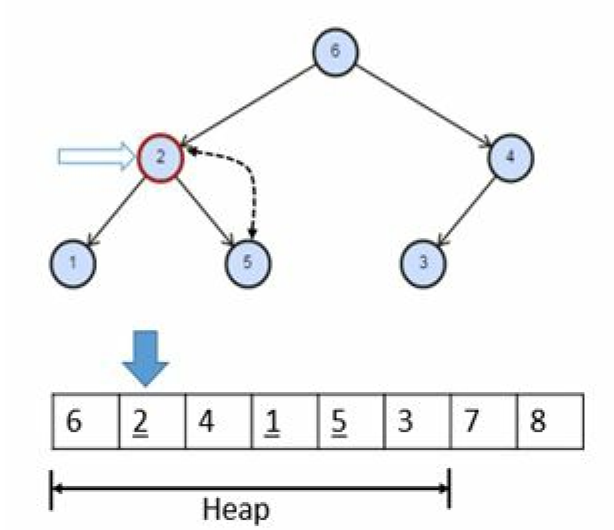
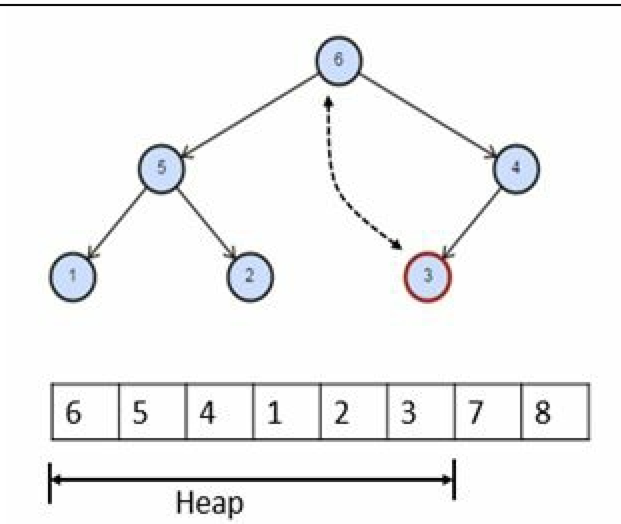
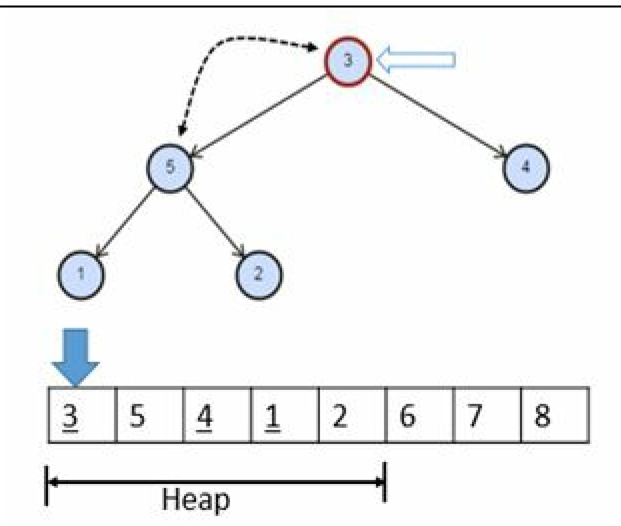
Heap-Sort

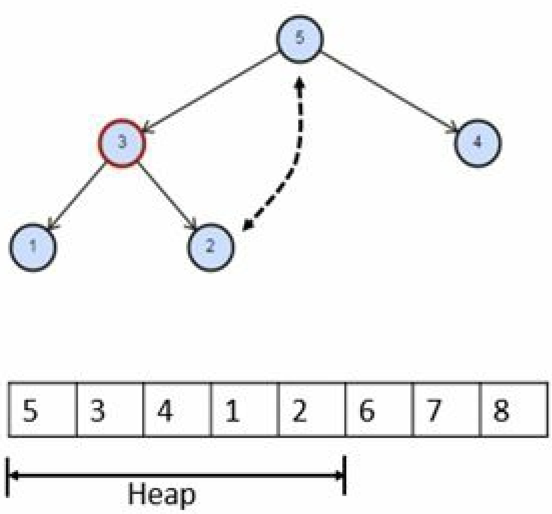
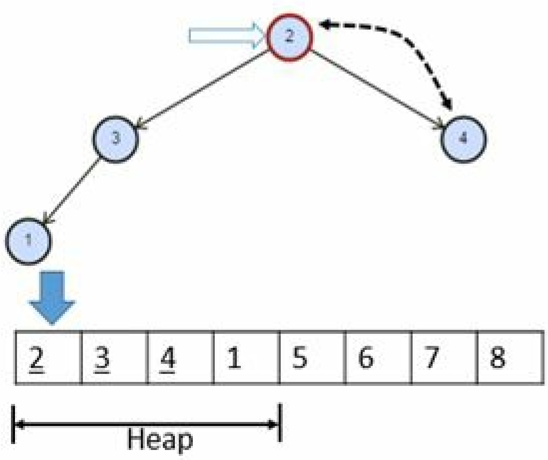
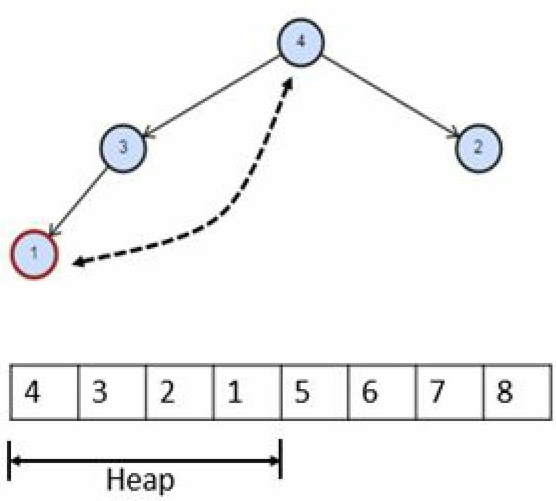
1. Use create heap function to build a max heap from the given list of elements. This operation will take **O(N)** time.
2. Dequeue the max value from the heap and store this value to the end of the list at location arr[size-1]
    a) Copy the value at the root of the heap to end of the list.
    b) Copy the last element of the heap to the root, and then reduce the size of heap by 1. This element is called the "out-of-place" element.
    c) Restore heap property by swapping the out-of-place element with its greatest-value child. Repeat this process until the out-of-place element reaches a leaf or it has a value that is greater or equal to all its children
3. Repeat this operation until there is just one element in the heap.

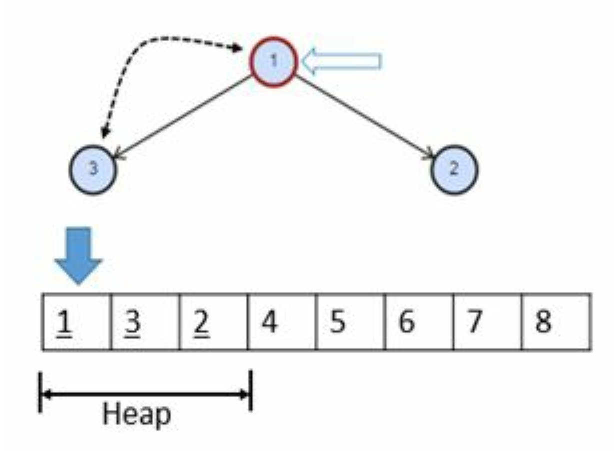
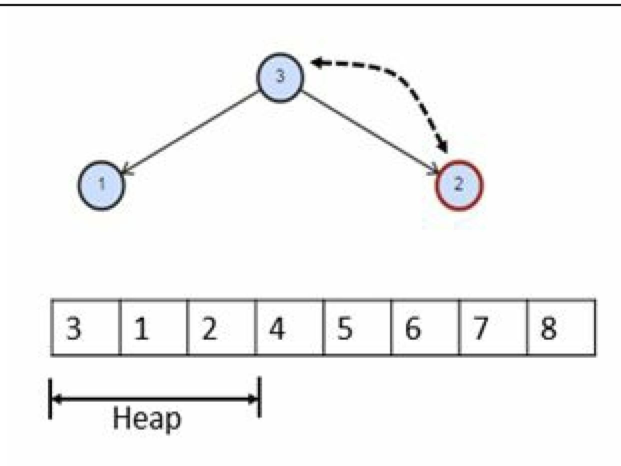
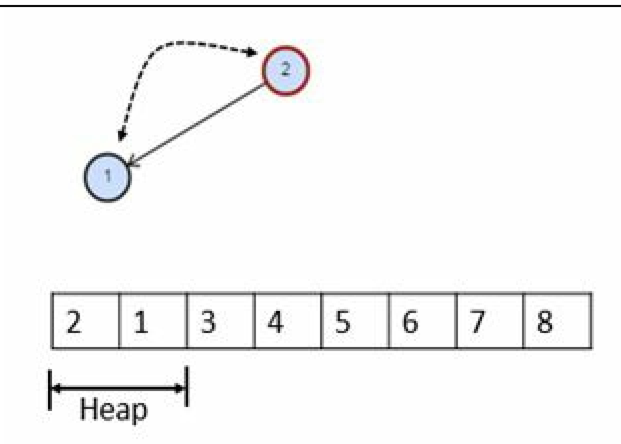
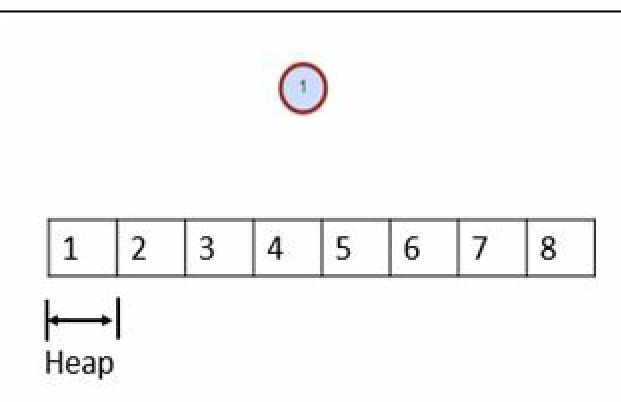
Let us take example of the heap that we had created at the start of the chapter. Heap sort is algorithm starts by creating a heap of the given list, which is done in linear time. Then at each step head of the heap is swapped with the end of the heap and the heap size is reduced by 1. Then percolate down is used to restore the heap property. Moreover, this same is done multiple times until the heap contain just one element.

	We had started with max heap. The maximum value as the first element of the Heap list is swapped with the last element of the list. Now the largest value is at the end of the list. Then we will reduce the size of the heap by one.
	Since 1 is at the top of the heap. Moreover, heap property is lost we will use Percolate down method to regain the heap property.

	Percolate down cont.
	Since heap property is regained. Then we will copy the first element of the heap array to the second last position.
	Heap size is further reduced and percolate down cont.

<table>
<tr><td>
</td><td>Percolate down cont.</td></tr>
<tr><td>
</td><td>Again swap.</td></tr>
<tr><td>
</td><td>Size of heap reduced by 1 and percolate down.</td></tr>
<tr><td></td><td></td></tr>
</table>

	Again swap.
	Size of heap reduced and percolate down.
	Again swap.
	Size of heap reduced and percolate down.

	
	Again swap.
	Again swap.
	End.
	Final list, which is sorted in increasing order.

1	2	3	4	5	6	7	8

Example 11.5:

```
func HeapSort(arrInput []int) {
      hp := NewHeap(arrInput, true)
      n := len(arrInput)
      for i := 0; i < n; i++ {
            arrInput[i] = hp.Remove()
      }
}

func main() {
      a := []int{1, 9, 6, 7, 8, -1, 2, 4, 5, 3}
      HeapSort(a)
      fmt.Println(a)
}
```

Data structure	List
Worst Case Time Complexity	O(nlogn)
Best Case Time Complexity	O(nlogn)
Average Time Complexity	O(nlogn)
Space Complexity	O(1)

Note: Heap-Sort is not a Stable sort and do not require any extra space for sorting a list.

Uses of Heap

1. **Heapsort**: One of the best sorting methods being in-place and log(N) time complexity in all scenarios.

2. **Selection algorithms**: Finding the min, max, both the min and max, median, or even the kth largest element can be done in linear time (often constant time) using heaps.

3. **Priority Queues**: Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm. A heap is a useful data structure when you need to remove the object with the highest (or lowest) priority. Schedulers, timers

4. **Graph algorithms**: By using heaps as internal traversal data structures, run time will be reduced by polynomial order. Examples of such problems are Prim's minimal

5. Because of the lack of references, the operations are faster than a binary tree. In addition, some more complicated heaps (such as binomial) can be merged efficiently, which is not easy to do for a binary tree.

Problems in Heap

Kth Smallest in a Min Heap

Just call DeleteMin() operation K-1 times and then again call DeleteMin() this last operation will give Kth smallest value. Time Complexity O(KlogN)

Kth Largest in a Max Heap

Just call DeleteMax() operation K-1 times and then again call DeleteMax () this last operation will give Kth smallest value. Time Complexity O(KlogN)

100 Largest in a Stream

There are billions of integers coming out of a stream some getInt() function is providing integers one by one. How would you determine the largest 100 numbers?

Solution: Large hundred (or smallest hundred etc.) such problems are solved very easily using a Heap. In our case, we will create a min heap.

1. First from 100 first integers builds a min heap.
2. Then for each coming integer compare if it is greater than the top of the min heap.
3. If not, then look for next integer. If yes, then remove the top min value from the min heap, insert the new value at the top of the heap, use procolateDown, and move it to its proper position down the heap.
4. Every time you have largest 100 values stored in your head

Merge two Heap

How can we merge two heaps?

Solution: There is no single solution for this. Let us suppose the size of the bigger heap is N and the size of the smaller heap is M.

1. If both heaps are comparable size, then put both heap lists in same bigger Lists. Alternatively, in one of the Lists if they are big enough. Then apply CreateHeap() function which will take theta(N+M) time.
2. If M is much smaller than N then add() each element of M list one by one to N heap. This will take O(MlogN) the worst case or O(M) best case.

Get Median function

Give a data structure that will provide median of given values in constant time.

Solution: We will be using two heaps, one min heap and other max heap. Max heap will contain the first half of data and min heap will contain the second half of the data. Max heap will contain the smaller half of the data and its max value that is at the top of the heap will be the median contender. Similarly, the Min heap will contain the larger values of the data and its min value that is at its top will contain the median contender. We will keep track of the size of heaps. Whenever we insert a value to heap, we will make sure that the size of two heaps differs by max one element, otherwise we will pop one element from one and insert into another to keep them balanced.

Example 11.6:

```
type MedianHeap struct {
    minHeap *Heap
    maxHeap *Heap
}
```

```go
func NewMedianHeap() *MedianHeap {
	min := NewHeap(nil, true)
	max := NewHeap(nil, false)

	return &MedianHeap{
		minHeap: min,
		maxHeap: max,
	}
}

func (h *MedianHeap) insert(value int) {
	empty := h.maxHeap.Empty()

	if empty {
		h.maxHeap.Add(value)
	} else {
		top := h.maxHeap.Peek()
		if top >= value {
			h.maxHeap.Add(value)
		} else {
			h.minHeap.Add(value)
		}
	}

	// size balancing
	if h.maxHeap.Size() > h.minHeap.Size()+1 {
		value := h.maxHeap.Remove()
		h.minHeap.Add(value)
	}

	if h.minHeap.Size() > h.maxHeap.Size()+1 {
		value := h.minHeap.Remove()
		h.maxHeap.Add(value)
	}
}

func (h *MedianHeap) getMedian() (int, bool) {
	if h.maxHeap.Size() == 0 && h.minHeap.Size() == 0 {
		fmt.Println("HeapEmptyError")
		return 0, false
	}

	if h.maxHeap.Size() == h.minHeap.Size() {
		val1 := h.maxHeap.Peek()
		val2 := h.minHeap.Peek()
		return (val1 + val2) / 2, true
	} else if h.maxHeap.Size() > h.minHeap.Size() {
		val1 := h.maxHeap.Peek()
		return val1, true
	} else {
		val2 := h.minHeap.Peek()
		return val2, true
	}
```

```
}
```

Is Min Heap

Example 11.7: Given a list, find if it is a binary Heap is Min Heap

```
func IsMinHeap(arr []int) bool {
	size := len(arr)
	for i := 0; i <= (size-2)/2; i++ {
		if 2*i+1 < size {
			if arr[i] > arr[2*i+1] {
				return false
			}
		}
		if 2*i+2 < size {
			if arr[i] > arr[2*i+2] {
				return false
			}
		}
	}
	return true
}
```

Is Max Heap

Example 11.8: Given a list find if it is a binary Heap Max heap

```
func IsMaxHeap(arr []int) bool {
	size := len(arr)
	for i := 0; i <= (size-2)/2; i++ {
		if 2*i+1 < size {
			if arr[i] < arr[2*i+1] {
				return false
			}
		}
		if 2*i+2 < size {
			if arr[i] < arr[2*i+2] {
				return false
			}
		}
	}
	return true
```

Analysis: If each parent value is greater than its children value then heap property is true. We will traverse from start to half of the array and compare the value of index node with its left child and right child node.

Traversal in Heap

Heaps are not designed to traverse to find some element they are made to get min or max element fast. Still if you want to traverse a heap just traverse the list sequentially. This traversal will be level order traversal. This traversal will have linear Time Complexity.

Deleting Arbiter element from Min Heap

Again, heap is not designed to delete an arbitrary element, but still if you want to do so. Find the element by linear search in the heap list. Replace it with the value stored at the end of the Heap value. Reduce the size of the heap by one. Compare the new inserted value with its parent. If its value is smaller than the parent value, then percolate up. Else if its value is greater than its left and right child then percolate down. Time Complexity is **O(logn)**

Deleting Kth element from Min Heap

Again, heap is not designed to delete an arbitrary element, but still if you want to do so. Replace the kth value with the value stored at the end of the Heap value. Reduce the size of the heap by one. Compare the new inserted value with its parent. If its value is smaller than the parent value, then percolate up. Else if its value is greater than its left and right child then percolate down. Time Complexity is **O(logn)**

Print value in Range in Min Heap

Linearly traverse through the heap and print the value that are in the given range.

Priority Queue generic implementation.

Let us look into priority queue implementation, which internally implements heap. We will Add() items and their priority to the queue.
While removing items form priority queue using Remove() function, highest priority item is removed first and so on.

Example 11.9:

```
type PQueue struct {
	items []*item
	Count int
	isMin bool
}

type item struct {
	value interface{}
	priority int
}

func newItem(value interface{}, priority int) *item {
	return &item{
		value:    value,
		priority: priority,
	}
}

func NewPQueue(isMin bool) *PQueue {
	items := make([]*item, 1)
	items[0] = nil // Heap queue first element should always be nil
	return &PQueue{
		items: items,
		Count: 0,
		isMin: isMin,
	}
}

func (pq *PQueue) comp(i, j int) bool { // always i < j in use
	if pq.isMin == true {
		return pq.items[i].priority > pq.items[j].priority // swaps for min heap
	}
	return pq.items[i].priority < pq.items[j].priority // swap for max heap.
}

func (pq *PQueue) proclateDown(position int) {
	lChild := 2 * position
	rChild := lChild + 1
	small := -1
	if lChild <= pq.Count {
		small = lChild
	}
	if rChild <= pq.Count && pq.comp(lChild, rChild) {
		small = rChild
	}
```

```
	if small != -1 && pq.comp(position, small) {
		pq.items[position], pq.items[small] = pq.items[small], pq.items[position]
		pq.proclateDown(small)
	}
}

func (pq *PQueue) proclateUp(position int) {
	parent := position / 2
	if parent == 0 {
		return
	}
	if pq.comp(parent, position) {
		pq.items[position], pq.items[parent] = pq.items[parent], pq.items[position]
		pq.proclateUp(parent)
	}
}

func (pq *PQueue) Add(value interface{}, priority int) {
	item := newItem(value, priority)
	pq.items = append(pq.items, item)
	pq.Count++
	pq.proclateUp(pq.Count)
}

func (pq *PQueue) Print() {
	n := pq.Count
	for i := 1; i <= n; i++ {
		fmt.Print(*(pq.items[i]), " ")
	}
	fmt.Println()
}

func (pq *PQueue) Remove() (interface{}, bool) {
	if pq.IsEmpty() {
		fmt.Println("Heap Empty Error.")
		return 0, false
	}
	value := pq.items[1].value
	pq.items[1] = pq.items[pq.Count]
	pq.items = pq.items[0:pq.Count]
	pq.Count--
	pq.proclateDown(1)
	return value, true
}

func (pq *PQueue) IsEmpty() bool {
	return (pq.Count == 0)
}

func (pq *PQueue) Len() int {
	return pq.Count
}
```

```
func (pq *PQueue) Peek() (interface{}, bool) {
	if pq.IsEmpty() {
		fmt.Println("Heap empty Error.")
		return 0, false
	}
	return pq.items[1].value, true
}

func main() {
	pq := NewPQueue(true)
	pq.Add("banana", 2)
	pq.Add("apple", 1)
	pq.Add("orange", 4)
	pq.Add("mango", 3)
	fmt.Println(pq.Len())

	for pq.IsEmpty() == false {
		fmt.Println(pq.Remove())
	}
}
```

Output:

```
4
apple true
banana true
mango true
orange true
```

Analysis:
In this implementation, the elements of heap have two fields, first field of type interface{} which will store data and second field is of type integer which is priority. Elements will be removed according to their priority order.

Heap operations like Add(), Remove(), proclateUp(), procolateDown() are already discussed in Heap section.

Priority Queue using heap from container.

Example 11.10:

```
import "container/heap"

type Item struct {
	value interface{}
	priority int
}

type ItemList []*Item

func (lst ItemList) Len() int {
	return len(lst)
}

func (lst ItemList) Less(i, j int) bool {
	return lst[i].priority < lst[j].priority
}

func (lst ItemList) Swap(i, j int) {
	lst[i], lst[j] = lst[j], lst[i]
}

func (lst *ItemList) Push(val interface{}) {
	item := val.(*Item)
	*lst = append(*lst, item)
}

func (lst *ItemList) Pop() interface{} {
	old := *lst
	n := len(old)
	item := old[n-1]
	*lst = old[0 : n-1]
	return item
}

type PQueue struct {
	pq ItemList
}

func NewPQueue() *PQueue {
	queue := new(PQueue)
	queue.pq = make(ItemList, 0)
	heap.Init(&queue.pq)
	return queue
}

func (queue *PQueue) Init() {
	queue.pq = make(ItemList, 0)
	heap.Init(&queue.pq)
```

```go
}

func (queue *PQueue) Add(value interface{}, priority int) {
	heap.Push(&queue.pq, &Item{value: value, priority: priority})
}

func (queue *PQueue) Remove() interface{} {
	return heap.Pop(&queue.pq).(*Item).value
}

func (queue *PQueue) Len() int {
	return queue.pq.Len()
}

func (queue *PQueue) IsEmpty() bool {
	return queue.pq.Len() == 0
}

func main() {
	pq := NewPQueue()
	// pq := new(PQueue)
	// pq.Init()
	pq.Add("banana", 2)
	pq.Add("apple", 1)
	pq.Add("orange", 4)
	pq.Add("mango", 3)
	fmt.Println(pq.Len())

	for pq.IsEmpty() == false {
		fmt.Println(pq.Remove())
	}
}
```

Exercise

1. What is the worst-case runtime Complexity of finding the smallest item in a min-heap?

2. Find max in a min heap.
   Hint: normal search in the complete list. There is one more optimization you can search from the mid of the list at index N/2

3. What is the worst-case time Complexity of finding the largest item in a min-heap?

4. What is the worst-case time Complexity of deleteMin in a min-heap?

5. What is the worst-case time Complexity of building a heap by insertion?

6. Is a heap full or complete binary tree?

7. What is the worst time runtime Complexity of sorting a list of N elements using heapsort?

8. Given a sequence of numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9
   a. Draw a binary Min-heap by inserting the above numbers one by one
   b. Also draw the tree that will be formed after calling Dequeue() on this heap

9. Given a sequence of numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9
   a. Draw a binary Max-heap by inserting the above numbers one by one
   b. Also draw the tree that will be formed after calling Dequeue() on this heap

10. Given a sequence of numbers: 3, 9, 5, 4, 8, 1, 5, 2, 7, 6. Construct a Min-heap by calling CreateHeap function.

11. Show a list that would be the result after the call to deleteMin() on this heap

12. Given a list: [3, 9, 5, 4, 8, 1, 5, 2, 7, 6]. Apply heapify over this to make a min heap and sort the elements in decreasing order?

13. In Heap-Sort once a root element has been put in its final position, how much time, does it take to re-heapify the structure so that the next removal can take place? In other words, what is the Time Complexity of a single element removal from the heap of size N?

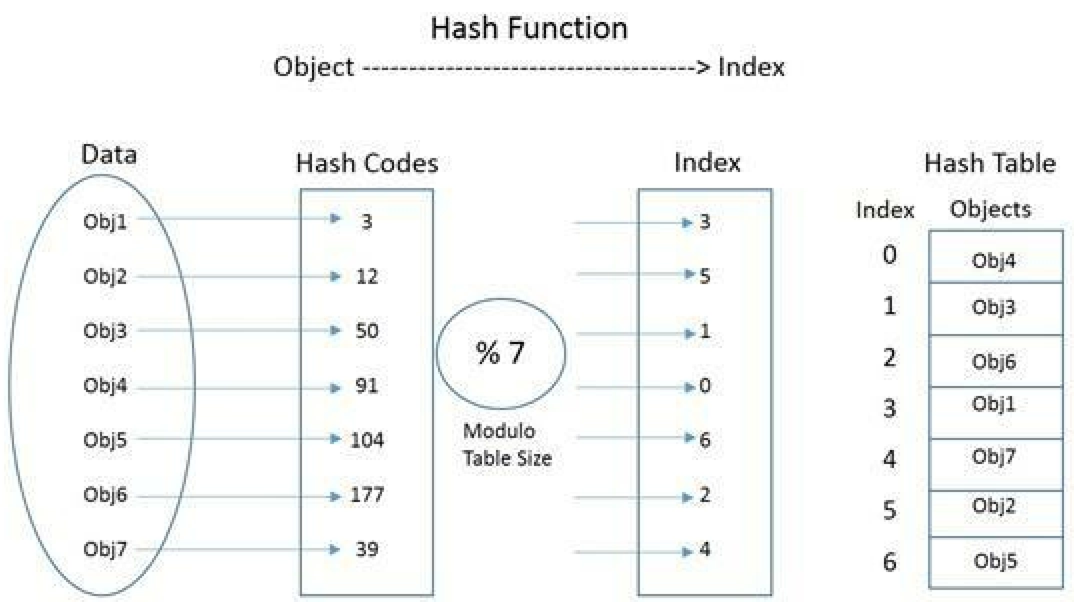
14. What do you think the overall Time Complexity for heapsort is? Why do you feel this way?

CHAPTER 12: HASH-TABLE

Introduction

In the searching chapter, we have looked into various searching techniques. Consider a problem of searching a value in an array. If the array is not sorted then we have no other option, but to look into each element one by one so the searching Time Complexity will be **O(n)**. If the array is sorted then we can search the value in **O(logn)** logarithmic time using binary search.

What if the possible location / index of the value that we are looking in the array is returned by a magic function in constant time? We can directly go into that location and tell whether the value we are searching for is present or not in just **O(1)** constant time. Such a function is called a Hash function.



The process of storing objects using a hash function is as follows:

1. Create a list of size M to store objects; this list is called Hash-Table.
2. Find a hash code of an object by passing it through the hash function.
3. Take module of hash code by the size of Hashtable to get the index of the table where objects will be stored.
4. Finally store these objects in the designated index.

The process of searching objects in Hash-Table using a hash function is as follows:

1. Find a hash code of the object we are searching for by passing it through the hash function.
2. Take module of hash code by the size of Hashtable to get the index of the table where objects are stored.
3. Finally, retrieve the object from the designated index.

Hash-Table

A Hash-Table is a data structure that maps keys to values. Each position of the Hash-Table is called a slot. The Hash-Table uses a hash function to calculate an index of a list. We use the Hash-Table when the number of keys actually stored is small relatively to the number of possible keys.

Hash-Table Abstract Data Type (ADT)

ADT of Hash-Table contains the following functions:

1. Insert(x), add object x to the data set.
2. Delete(x), delete object x from the data set.
3. Search(x), search object x in data set.

Hash Function

A hash function is a function that generates an index in a table for a given object.
An ideal hash function that generate a unique index for every object is called the perfect hash function.

Example 12.1: Most simple hash function

```
func (ht *HashTable) ComputeHash(key int) int {
    return key % ht.tableSize
}
```

There are many hash functions. The above function is a very simple hash function. Various hash generation logics will be added to this function to generate a better hash.

Collisions

When a hash function generates the same index for the two or more different objects, the problem known as the collision. Ideally, hash function should return a unique address for each key, but practically it is not possible.

Properties of good hash function:

1. It should provide a uniform distribution of hash values. A non-uniform distribution increased the number of collisions and the cost of resolving them.
2. Choose a hash function, which can be computed quickly and returns values within the range of the Hash-Table.
3. Chose a hash function with a good collision resolution algorithm which can be used to compute alternative index if the collision occurs.
4. Choose a hash function, which uses the necessary information provided in the key.
5. It should have high load factor for a given set of keys.

Load Factor

Load factor = Number of elements in Hash-Table / Hash-Table size

Based on the above definition, Load factor tells whether the hash function is distributing the keys uniformly or not. Therefore, it helps in determining the efficiency of the hashing function. It also works as decision parameter when we want to expand or rehash the existing Hash-Table entries.

Collision Resolution Techniques

Hash collisions are practically unavoidable when hashing large number of objects. Techniques that are used to find the alternate location in the Hash-Table is called collision resolution. There are a number of collision resolution techniques to handle the collision in hashing.

Most common and widely used techniques are:

- Open addressing
- Separate chaining

Hashing with Open Addressing

When using linear open addressing, the Hash-Table is represented by a one-dimensional list with indices that range from 0 to the desired table size-1.

One method of resolving collision is the look into a Hash-Table and find another free slot the hold the object that have caused the collision. A simple way is to move from one slot to another in some sequential order until we find a free space. This collision resolution process is called Open Addressing.

Linear Probing

In Linear Probing, we try to resolve the collision of an index of a Hash-Table by sequentially searching the Hash-Table free location. Let us suppose, if k is the index retrieved from the hash function. If the kth index is already filled then we will look for (k+1) %M, then (k+2) %M and so on. When we get a free slot, we will insert the object into that free slot.

Example 12.2: The resolver function of linear probing

```
func (ht *HashTable) ResolverFun(index int) int {
    return index
}
```

Quadratic Probing

In Quadratic Probing, we try to resolve the collision of the index of a Hash-Table by quadratic ally increasing the search index free location. Let us suppose, if k is the index retrieved from the hash function. If the kth index is already filled then we will look for (k+1^2) %M, then (k+2^2) %M and so on. When we get a free slot, we will insert the object into that free slot.

Example 12.3: The resolver function of quadratic probing

```
func (ht *HashTable) ResolverFun(index int) int {
    return index * index
}
```

Table size should be a prime number to prevent early looping should not be too close to 2powN

Linear Probing implementation

Example 12.4: Below is a linear probing collision resolution Hash-Table implementation.

```
const (
    EmptyNode byte = iota
    LazyDeleted
    FillledNode
)

type HashTable struct {
    Arr      []int
    Flag    []byte
    tableSize int
}

func (ht *HashTable) Init(tSize int) {
    ht.tableSize = tSize
```

```
    ht.Arr = make([]int, (tSize + 1))
    ht.Flag = make([]byte, (tSize + 1))
}
```

Table list size will be 50 and we have defined two constant values EMPTY_NODE and LAZY_DELETED.

```
func (ht *HashTable) ComputeHash(key int) int {
    return key % ht.tableSize
}
```

This is the most simple hash generation function, which just take the modulus of the key.

```
func (ht *HashTable) ResolverFun(index int) int {
    return index
}
```

When the hash index is already occupied by some element the value will be placed in some other location to find that new location resolver function is used.

Hash-Table has two component one is table size and other is reference to list.

Example 12.5:

```
func (ht *HashTable) Add(value int) bool {
    hashValue := ht.ComputeHash(value)
    for i := 0; i < ht.tableSize; i++ {
        if ht.Flag[hashValue] == EmptyNode || ht.Flag[hashValue] == LazyDeleted {
            ht.Arr[hashValue] = value
            ht.Flag[hashValue] = FillledNode
            return true
        }
        hashValue += ht.ResolverFun(i)
        hashValue %= ht.tableSize
    }
    return false
}
```

An insert node function is used to add values to the list. First hash is calculated. Then we try to place that value in the Hash-Table. We look for empty node or lazy deleted node to insert value. In case insert did not success, we try new location using a resolver function.

Example 12.6:

```
func (ht *HashTable) Find(value int) bool {
    hashValue := ht.ComputeHash(value)
    for i := 0; i < ht.tableSize; i++ {
        if ht.Flag[hashValue] == EmptyNode {
            return false
        }
        if ht.Flag[hashValue] == FillledNode && ht.Arr[hashValue] == value {
            return true
        }
        hashValue += ht.ResolverFun(i)
        hashValue %= ht.tableSize
    }
    return false
}
```

Find node function is used to search values in the array. First hash is calculated. Then we try to find that value in the Hash-Table. We look for over desired value or empty node. In case we find the value that we are looking, then we return that value or in case it is not found we return -1. We use a resolver function to find the next probable index to search.

Example 12.7:
```
func (ht *HashTable) Remove(value int) bool {
    hashValue := ht.ComputeHash(value)
    for i := 0; i < ht.tableSize; i++ {
        if ht.Flag[hashValue] == EmptyNode {
            return false
        }
        if ht.Flag[hashValue] == FillledNode && ht.Arr[hashValue] == value {
            ht.Flag[hashValue] = LazyDeleted
            return true
        }
        hashValue += ht.ResolverFun(i)
        hashValue %= ht.tableSize
    }
    return false
}
```

Delete node function is used to delete values from a Hashtable. We do not actually delete the value we just mark that value as LAZY_DELETED. Same as the insert and search we use resolverFun to find the next probable location of the key.

Example 12.8:
```
func (ht *HashTable) Print() {
    fmt.Println("\nValues Stored in HashTable are::")
    for i := 0; i < ht.tableSize; i++ {
        if ht.Flag[i] == FillledNode {
            fmt.Println("Node at index [", i, " ] :: ", ht.Arr[i])
        }
    }
}

func main() {
    ht := new(HashTable)
    ht.Init(1000)
    ht.Add(89)
    fmt.Println("89 found : ", ht.Find(89))
    ht.Remove(89)
    fmt.Println("89 found : ", ht.Find(89))
    ht.Print()
}
```

Print method print the content of hash table. Main function demonstrating how to use hash table.

Quadratic Probing implementation.

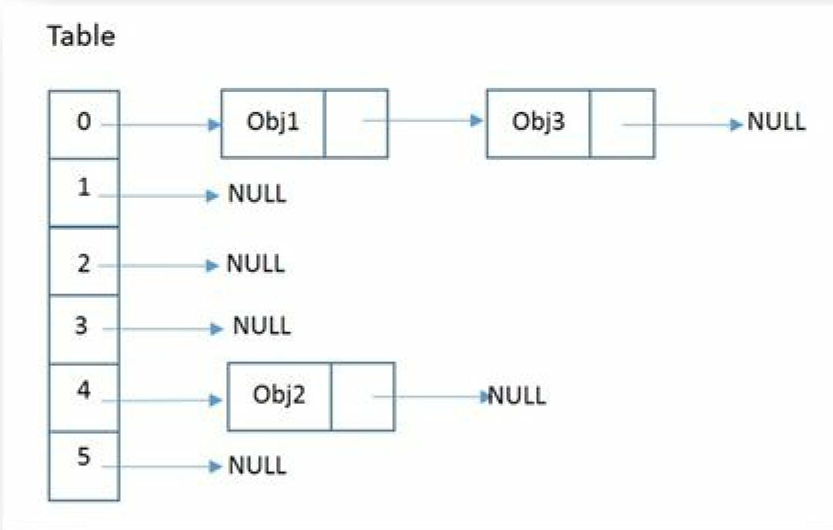
Everything will be same as linear probing implementation only resolver function will be changed.

```
func (ht *HashTable) ResolverFun(index int) int {
	return index * index
}
```

}

Hashing with separate chaining

Another method for collision resolution is based on an idea of putting the keys that collide in a linked list. This method is called separate chaining. To speed up search we use Insertion-Sort or keeping the linked list sorted.



Separate Chaining implementation

‘**Example 12.9:** Below is separate chaining implementation of hash tables.

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

type HashTableSC struct {
	listArray [](*Node)
	tableSize int
}

func (h *HashTableSC) Init() {
	h.tableSize = 101
	h.listArray = make([](*Node), h.tableSize)

	for i := 0; i < h.tableSize; i++ {
		h.listArray[i] = nil
	}
}

func (h *HashTableSC) ComputeHash(key int) int {
	hashValue := key
	return hashValue % h.tableSize
}

func (h *HashTableSC) Add(value int) {
	index := h.ComputeHash(value)
	temp := new(Node)
```

```
    temp.value = value
    temp.next = h.listArray[index]
    h.listArray[index] = temp
}

func (h *HashTableSC) Remove(value int) bool {
    index := h.ComputeHash(value)
    var nextNode, head *Node
    head = h.listArray[index]
    if head != nil && head.value == value {
        h.listArray[index] = head.next
        return true
    }
    for head != nil {
        nextNode = head.next
        if nextNode != nil && nextNode.value == value {
            head.next = nextNode.next
            return true
        }
        head = nextNode
    }
    return false
}

func (h *HashTableSC) Print() {
    for i := 0; i < h.tableSize; i++ {
        head := h.listArray[i]
        if head != nil {
            fmt.Print("\nValues at index :: ", i, " Are :: ")
        }
        for head != nil {
            fmt.Print(head.value, " ")
            head = head.next
        }
    }
    fmt.Println()
}

func (h *HashTableSC) Find(value int) bool {
    index := h.ComputeHash(value)
    head := h.listArray[index]
    for head != nil {
        if head.value == value {
            return true
        }
        head = head.next
    }
    return false
}

func main() {
    ht := new(HashTableSC)
    ht.Init()
```

```
    ht.Add(89)
    ht.Print()
    fmt.Println("89 found : ", ht.Find(89))
}
```

Note: It is important to note that the size of the “skip” must be such that all the slots in the table will eventually be occupied. Otherwise, part of the table will be unused. To ensure this, it is often suggested that the table size being a prime number. This is the reason we have been using 11 in our examples.

Problems in Hashing

Anagram solver

An anagram is a word or phrase formed by reordering the letters of another word or phrase.

Example 12.10: Two words are anagram if they are of same size and their characters are same.

```
func IsAnagram(str1 string, str2 string) bool {
	size1 := len(str1)
	size2 := len(str2)
	if size1 != size2 {
		return false
	}
	cm := make(Counter)
	for _, ch := range str1 {
		cm.Add(ch)
	}
	for _, ch := range str2 {
		cm.Remove(ch)
	}
	return (len(cm) == 0)
}
```

Remove Duplicate

Solution: We can use a second list or the same list, as the output list. In the below example Hash-Table is used to solve this problem.

Example 12.11: Remove duplicates in a list of numbers

```
func RemoveDuplicate(str string) string {
	input := []rune(str)
	hs := make(Set)
	var output []rune
	for _, ch := range input {
		if hs.Find(ch) == false {
			output = append(output, ch)
			hs.Add(ch)
		}
	}
	return string(output)
}
```

Find Missing

Example 12.12: There is a list of integers we need to find the missing number in the list.

```
func FindMissing(arr []int, start int, end int) (int, bool) {
	hs := make(Set)
	for _, i := range arr {
		hs.Add(i)
	}
	for curr := start; curr <= end; curr++ {
		if hs.Find(curr) == false {
			return curr, true
```

```
        }
    }
    return 0, false
}
```

All the elements in the list is added to a HashTable. The missing element is found by searching into HashTable and final missing value is returned.

Print Repeating

Example 12.13: Print the repeating integer in a list of integers.

```
func PrintRepeating(arr []int) {
    hs := make(Set)
    fmt.Print("Repeating elements are :: ")
    for _, val := range arr {
        if hs.Find(val) {
            fmt.Print(val, "  ")
        } else {
            hs.Add(val)
        }
    }
    fmt.Println()
}
```

All the values to the hash table when some value came which is already in the hash table then that is the repeated value.

Print First Repeating

Example 12.14: Same as the above problem in this we need to print the first repeating number. Caution should be taken to find the first repeating number. It should be the one number that is repeating. For example, 1, 2, 3, 2, 1. The answer should be 1 as it is the first number, which is repeating.

```
func PrintFirstRepeating(arr []int) {
    size := len(arr)
    hs := make(Counter)

    for i := 0; i < size; i++ {
        hs.Add(arr[i])
    }
    for i := 0; i < size; i++ {
        hs.Remove(arr[i])
        if hs.Find(arr[i]) {
            fmt.Println("First Repeating number is : ", arr[i])
            return
        }
    }
}
```

Add values to the count map the one that is repeating will have multiple count. Now traverse the list again and see if the count is more than one. So that is the first repeating.

Exercise

1. Design a number (ID) generator system that generate numbers between 0-99999999 (8-digits).
   The system should support two functions:
   a. int getNumber();
   b. int requestNumber();
   getNumber() function should find out a number that is not assigned, then marks it as assigned and return that number. requestNumber() function checks the number is assigned or not. If it is assigned returns 0, else marks it as assigned and return 1.

2. Given a large string, find the most occurring words in the string. What is the Time Complexity of the above solution?
   Hint:-
   a. Create a Hashtable which will keep track of <word, frequency>
   b. Iterate through the string and keep track of word frequency by inserting into Hash-Table.
   c. When we have a new word, we will insert it into the Hashtable with frequency 1. For all repetition of the word, we will increase the frequency.
   d. We can keep track of the most occurring words whenever we are increasing the frequency we can see if this is the most occurring word or not.
   e. The Time Complexity is **O(n)** where n is the number of words in the string and Space Complexity is the O(m) where m is the unique words in the string.

3. In the above question, What if you are given whole work of OSCAR WILDE, most popular playwrights in the early 1890s.
   Hint:-
   a. Who knows how many books are there, let us assume there is a lot and we cannot put everything in memory. First, we need a Streaming Library so that we can read section by section in each document. Then we need a tokenizer that will give words to our program. In addition, we need some sort of dictionary let us say we will use HashTable.
   b. What you need is - 1. A streaming library tokenizer, 2. A tokenizer 3. A hashmap
      Method:
      1. Use streamers to find a stream of the given words
      2. Tokenize the input text
      3. If the stemmed word is in hash map, increment its frequency count else adds a word to hash map with frequency 1
   c. We can improve the performance by looking into parallel computing. We can use the map-reduce to solve this problem. Multiple nodes will read and process multiple documents. Once they are done with their processing, then we can use reduce to merge them.

4. In the above question, What if we wanted to find the most common PHRASE in his writings.
   Hint: - We can keep <phrase, frequency> Hash-Table and do the same process of the 2nd and 3rd problems.

5. Write a hashing algorithm for strings.
   Hint: Use Horner's method

```
func hornerHash(key []rune, tableSize int) int {
  size := len(key)
  h := 0
  for i := 0; i < size; i++ {
      h = (32*h + (int)(key[i])) % tableSize
  }
  return h
}
```

6. Pick two data structures to use in implementing a Map. Describe lookup, insert, & delete operations. Give time & Space Complexity for each. Give pros & cons for each.

Hint:-

a) Linked List
   I. Insert is **O(1)**
   II. Delete is **O(1)**
   III. Lookup is **O(1)** auxiliary and **O(N)** worst case.
   IV. Pros: Fast inserts and deletes, can use for any data type.
   V. Cons: Slow lookups.

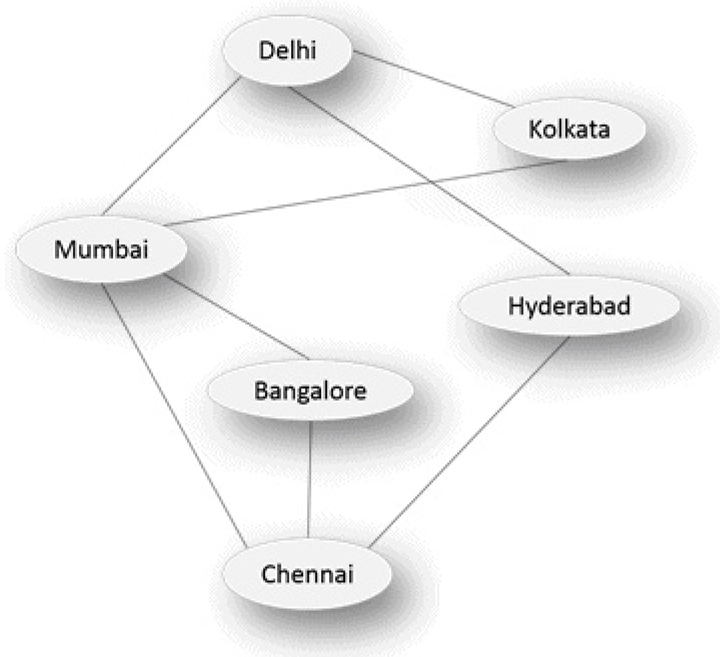
b) Balanced Search Tree (RB Tree)
   I. Insert is **O(logn)**
   II. Delete is **O(logn)**
   III. Lookup is **O(logn)**
   IV. Pros: Reasonably fast inserts/deletes and lookups.
   V. Cons: Data needs to have order defined on it.

CHAPTER 13: GRAPHS

Introduction

In this chapter, we will study about Graphs. Graphs can be used to represent many interesting things in the real world. Flights from cities to cities, rods connecting various town and cities. Even the sequence of steps that we take to become ready for jobs daily, or even a sequence of classes that we take to become a graduate in computer science. Once we have a good representation of the map, then we use a standard graph algorithms to solve many interesting problems of real life.

The flight connection between major cities of India can also be represented by the below graph. Each node is a city and each edge is a straight flight path from one city to another. You may want to go from Delhi to Chennai, if given this data in good representation to a computer, through graph algorithms the computer may propose shortest, quickest or cheapest path from soured to destination.



Google map that we use is also a big graph of lots of nodes and edges. That suggest shortest and quickest path to the user.

Graph Definitions

A Graph is represented by G where G = (V, E), where V is a finite set of points called **Vertices** and E is a finite set of **Edges**.

Each **edge** is a tuple (u, v) where u, v ∈ V. There can be a third component weight to the tuple. **Weight** is cost to go from one vertex to another.

Edge in a graph can be directed or undirected. If the edges of graph are one way, it is called **Directed graph** or **Digraph**. The graph whose edges are two ways are called **Undirected graph** or just graph.

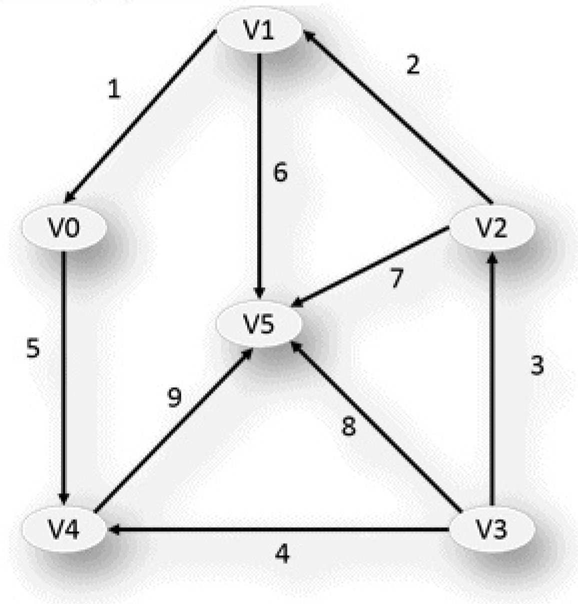
A **Path** is a sequence of edges between two vertices. The length of a path is defined as the sum of the weight of all the edges in the path.

Two vertices u and v are **adjacent** if there is an edge whose endpoints are u and v.

In the below graph:

V = {V1, V2, V3, V4, V5, V6, V7, V8, V9},

$$E = \left\{\begin{matrix}(V1, V0,1), (V2, V1,2), (V3, V2,3), (V3, V4,4), (V5, V4,5), \\ (V1, V5, 6), (V2, V5,7), (V3, V5,8), (V4, V5,9)\end{matrix}\right\}$$



The **in-degree** of a vertex v, denoted by indeg(v) is the number of incoming edges to the vertex v. The **out-degree** of a vertex v, denoted by outdeg(v) is the number of outgoing edges of a vertex v. The **degree** of a vertex v, denoted by deg(v) is the total number of edges whose one endpoint is v.

deg(v) = Indeg (v) + outdeg (v)

In the above graph
deg(V4) =3, indeg(V4) =2 and outdeg(V4) =1

A **Cycle** is a path that starts and ends at the same vertex and include at least one vertex.

An edge is a **Self-Loop** if two if its two endpoints coincide. This is a form of a cycle.

A vertex v is **Reachable** from vertex u or “u reaches v” if there is a path from u to v. In an undirected graph if v is reachable from u then u is reachable from v. However, in a directed graph it is possible that u reaches v but there is no path from v to u.

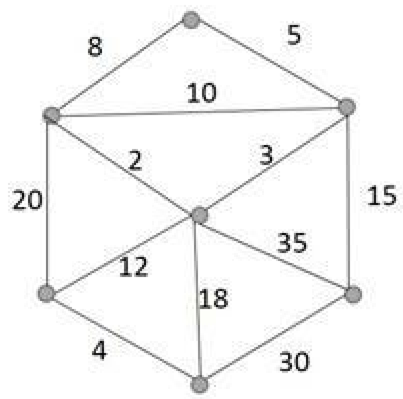
A graph is **Connected** if for any two vertices there is a path between them.

A **Forest** is a graph without cycles.

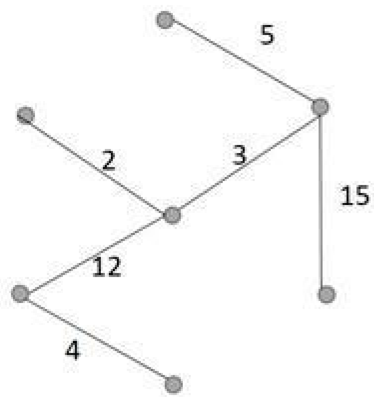
A **Sub-Graph** of a graph G is a graph whose vertices and edges are a subset of the vertices and edges of G.
A **Spanning Sub-Graph** of G is a graph that connects all the vertices of G.

A **tree** is an acyclic connected graph.

A **Spanning tree** of a graph is a tree that connects all the vertices of the graph. Since a Spanning-Tree is a tree, so it should not have any cycle.



Graph



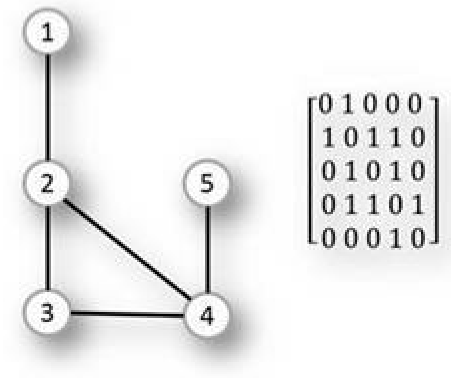
Spanning Tree

Graph Representation

In this section, we introduce the data structure for representing a graph. In the below representations we maintain a collection to store edges and vertices of the graph.

Adjacency Matrix

One of the ways to represent a graph is to use two-dimensional matrix. Each combination of row and column represent a vertex in the graph. The value stored at the location row v and column w is the edge from vertex v to vertex w. The nodes that are connected by an edge are called adjacent nodes. This matrix is used to store adjacent relation so it is called the Adjacency Matrix. In the below diagram, we have a graph and its Adjacency matrix.



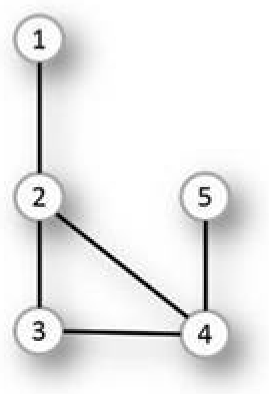
In the above graph, each node has weight 1 so the adjacency matrix has just 1s or 0s. If the edges are of different, weights that that weight will be filled in the matrix.

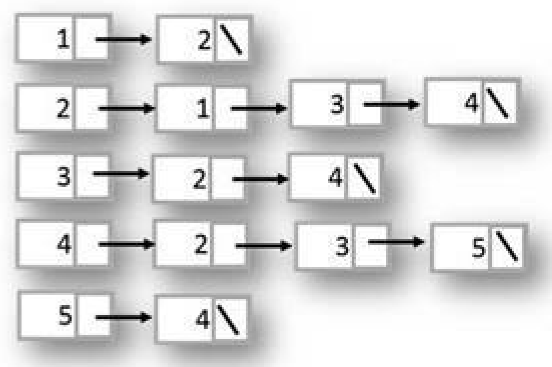
Pros: Adjacency matrix implementation is simple. Adding/Removing an edge between two vertices is just **O(1)**. Query if there is an edge between two vertices is also **O(1)**

Cons: It always consumes $O(V^2)$ space, which is an inefficient way to store when a graph is a sparse.
Sparse Matrix: In a huge graph, each node is connected with fewer nodes. So most of the places in adjacency matrix are empty. Such matrix is called sparse matrix. In most of the real world problems adjacency matrix is not a good choice for sore graph data.

Adjacency List

A more space efficient way of storing graph is adjacency list. In adjacency list of references to a linked list node. Each reference corresponds to vertices in a graph. Each reference will then point to the vertices that are connected to it and store this as a list.
In the below diagram node 2 is connected to 1, 3 and 4. Therefore, the reference at location 2 is pointing to a list that contain 1, 3 and 4.





The adjacency list helps us to represent a sparse graph. An adjacency list representation also allows us to find all the vertices that are directly connected to any vertices by just one link list scan. In all our programs, we are going to use the adjacency list to store the graph.

Example 13.1: adjacency list representation of an undirected graph

```
type Edge struct {
    source      int
    destination int
    cost        int
    next        *Edge
}

type EdgesList struct {
    head *Edge
}

type Graph struct {
    count   int
    VertexList [](*EdgesList)
}

func (g *Graph) Init(cnt int) {
    g.count = cnt
    g.VertexList = make([]*EdgesList, cnt)
    for i := 0; i < cnt; i++ {
        g.VertexList[i] = new(EdgesList)
        g.VertexList[i].head = nil
    }
}

func (g *Graph) AddEdge(source int, destination int, cost int) {
```

```go
	edge := &Edge{source, destination, cost, g.VertexList[source].head}
	g.VertexList[source].head = edge
}

func (g *Graph) AddEdgeUnweighted(source int, destination int) {
	g.AddEdge(source, destination, 1)
}

func (g *Graph) AddBiEdge(source int, destination int, cost int) {
	g.AddEdge(source, destination, cost)
	g.AddEdge(destination, source, cost)
}

func (g *Graph) AddBiEdgeUnweghted(source int, destination int) {
	g.AddBiEdge(source, destination, 1)
}

func (g *Graph) Print() {
	for i := 0; i < g.count; i++ {
		ad := g.VertexList[i].head
		if ad != nil {
			fmt.Print("Vertex ", i, " is connected to : ")
			for ad != nil {
				fmt.Print("[", ad.destination, ad.cost, "] ")
				ad = ad.next
			}
			fmt.Println()
		}
	}
}
```

Graph traversals

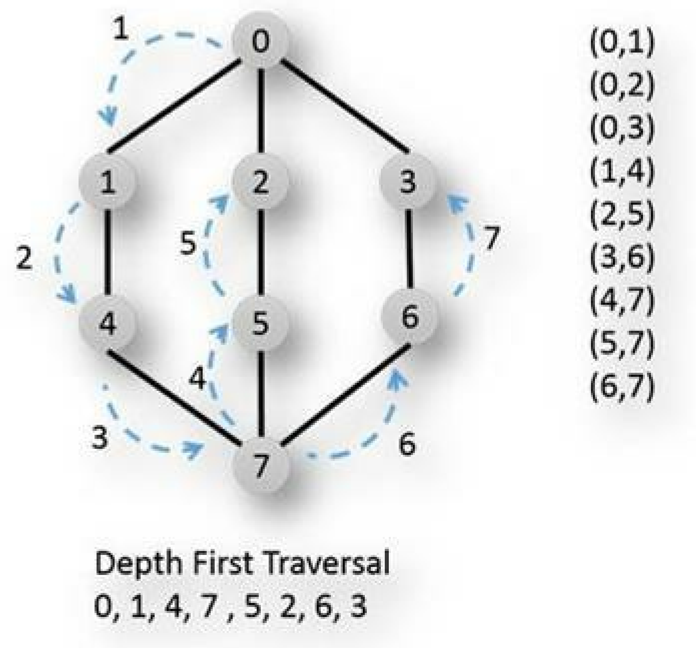
The **Depth first search (DFS**) and **Breadth first search (BFS**) are the two algorithms used to traverse a graph. These same algorithms can also be used to find some node in the graph, find if a node is reachable etc.

Traversal is the process of exploring a graph by examining all its edges and vertices.
A list of some of the problems that are solved using graph traversal are:

1. Determining a path from vertex u to vertex v, or report an error if there is no such path.
2. Given a starting vertex s, finding the minimum number of edges from vertex s to all the other vertices of the graph.
3. Testing of a graph G is connected.
4. Finding a spanning tree of a Graph.
5. Finding if there is some cycle in the graph.

Depth First Traversal

The DFS algorithm we start from starting point and go into depth of graph until we reach a dead end and then move up to parent node (Backtrack). In DFS, we use stack to get the next vertex to start a search. Alternatively, we can use recursion (system stack) to do the same.



Algorithm steps for DFS

1. Push the starting node in the stack.
2. Loop until the stack is empty.
3. Pop the node from the stack inside loop call this node current.
4. Process the current node. //Print, etc.
5. Traverse all the child nodes of the current node and push them into stack.
6. Repeat steps 3 to 5 until the stack is empty.

Stack based implementation of DFS

Example 13.2:

```
func (g *Graph) DFSStack() {
	count := g.count
	visited := make([]int, count)
	var curr int
	stk := new(stack.Stack)
	for i := 0; i < count; i++ {
		visited[i] = 0
	}
	visited[1] = 1
	stk.Push(1)

	for stk.Len() != 0 {
		curr = stk.Pop().(int)
		head := g.VertexList[curr].head
		for head != nil {
			if visited[head.destination] == 0 {
```

```
                visited[head.destination] = 1
                stk.Push(head.destination)
            }
            head = head.next
        }
    }
}
```

Recursion based implementation of DFS

Example 13.3:

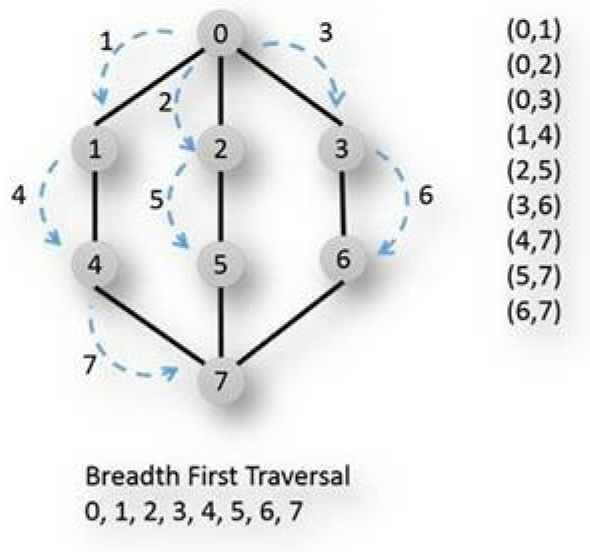
```
func (g *Graph) DFS() {
    count := g.count
    visited := make([]int, count)
    for i := 0; i < count; i++ {
        visited[i] = 0
    }
    for i := 0; i < count; i++ {
        if visited[i] == 0 {
            visited[i] = 1
            g.DFSRec(i, visited)
        }
    }
}

func (g *Graph) DFSRec(index int, visited []int) {
    head := g.VertexList[index].head
    fmt.Println(index)
    for head != nil {
        if visited[head.destination] == 0 {
            //fmt.Println(index)
            visited[head.destination] = 1
            g.DFSRec(head.destination, visited)
        }
        head = head.next
    }
}
```

Breadth First Traversal

In BFS algorithm, a graph is traversed in layer-by-layer fashion. The graph is traversed closer to the starting point. The queue is used to implement BFS.

Algorithm steps for BFS

1. Push the starting node into the Queue.
2. Loop until the Queue is empty.
3. Remove a node from the Queue inside loop, and call this node current.
4. Process the current node.//print etc.
5. Traverse all the child nodes of the current node and push them into Queue.
6. Repeat steps 3 to 5 until Queue is empty.



Example 13.4:

```
func (g *Graph) BFS() {
	count := g.count
	visited := make([]int, count)
	for i := 0; i < count; i++ {
		visited[i] = 0
	}
	for i := 0; i < count; i++ {
		if visited[i] == 0 {
			g.BFSQueue(i, visited)
		}
	}
}

func (g *Graph) BFSQueue(index int, visited []int) {
	var curr int
	que := new(queue.Queue)
	visited[index] = 1
	que.Enqueue(index)

	for que.Len() != 0 {
		curr = que.Dequeue().(int)
		head := g.VertexList[curr].head
		for head != nil {
```

```
                if visited[head.destination] == 0 {
                    visited[head.destination] = 1
                    que.Enqueue(head.destination)
                }
                head = head.next
            }
        }
}
```

A runtime analysis of DFS and BFS traversal is O(n+m) time, where n is the number of edges reachable from source node and m is the number of edges incident on s.

The following problems have O(m+n) time performance:

1. Determining a path from vertex u to vertex v, or report an error if there is no such path.
2. Given a starting vertex s, finding the minimum number of edges from vertex s to all the other vertices of the graph.
3. Testing of a graph G is connected.
4. Finding a spanning tree of a Graph.
5. Finding if there is some cycle in the graph.

Problems in Graph

Determining a path from vertex u to vertex v

IF there is a path from u to v and we are doing DFS from u then v must be visited. Moreover, if there is no path then report an error.

Example 13.5:

```
func (g *Graph) PathExist(source int, destination int) bool {
	count := g.count
	visited := make([]int, count)
	for i := 0; i < count; i++ {
		visited[i] = 0
	}
	visited[source] = 1
	g.DFSRec(source, visited)
	return visited[destination] != 0
}
```

Given a starting vertex s, finding the minimum number of edges from vertex s to all the other vertices of the graph

Look for single source shortest path algorithm for each edge cost as 1 unit.

Testing of a graph G is connected.

IF there is a path from u to v and we are doing DFS from u then v must be visited. Moreover, if there is no path then report an error.

Example 13.6:

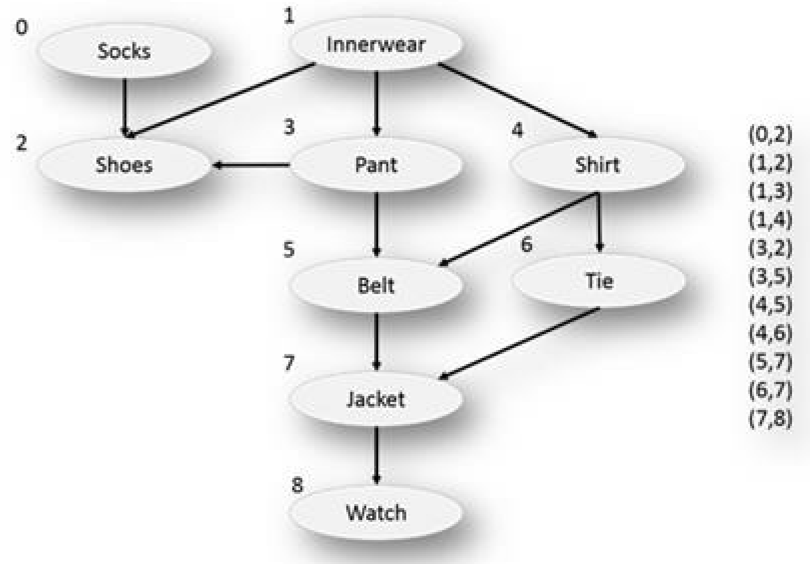
```
func (g *Graph) IsConnected() bool {
	count := g.count
	visited := make([]int, count)
	for i := 0; i < count; i++ {
		visited[i] = 0
	}
	visited[0] = 1
	g.DFSRec(0, visited)
	for i := 0; i < count; i++ {
		if visited[i] == 0 {
			return false
		}
	}
	return true
}
```

Finding if there is some cycle in the graph.

Modify DFS problem and get this done.

Directed Acyclic Graph

A Directed Acyclic Graph (DAG) is a directed graph with no cycle. A DAG represent relationship, which is more general than a tree. Below is an example of DAG, this is how someone becomes ready for work. There are N other real life examples of DAG such as coerces selection to being graduated from college



Topological Sort

A topological sort is a method of ordering the nodes of a directed graph in which nodes represent activities and the edges represent dependency among those tasks. For topological sorting to work it is required that the graph should be a DAG which means it should not have any cycle. Just use DFS to get topological sorting.

Example 13.7:

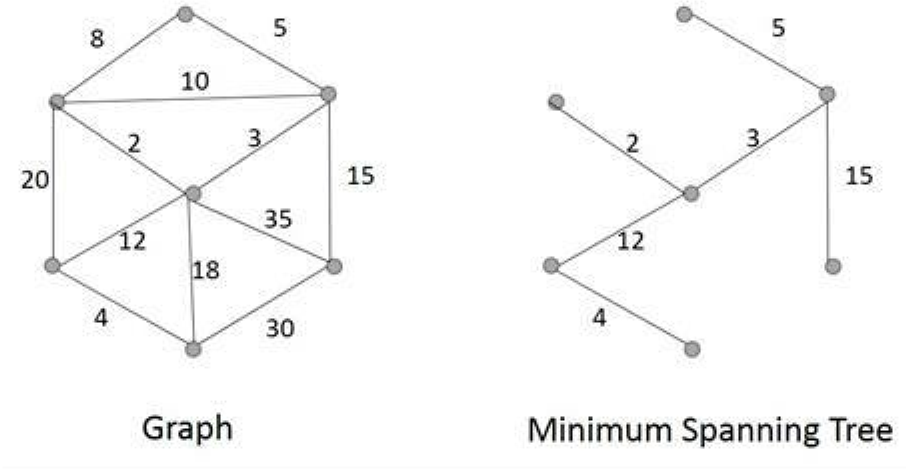
```
func (g *Graph) TopologicalSort() {
	fmt.Print("Topological order of given graph is : ")
	var count = g.count
	stk := new(stack.Stack)
	visited := make([]int, count)
	for i := 0; i < count; i++ {
		visited[i] = 0
	}
	for i := 0; i < count; i++ {
		if visited[i] == 0 {
			visited[i] = 1
			g.TopologicalSortDFS(i, visited, stk)
		}
	}
	for stk.Len() != 0 {
		fmt.Print(stk.Pop().(int), " ")
	}
	fmt.Println()
}

func (g *Graph) TopologicalSortDFS(index int, visited []int, stk *stack.Stack) {
	head := g.VertexList[index].head
	for head != nil {
		if visited[head.destination] == 0 {
			visited[head.destination] = 1
			g.TopologicalSortDFS(head.destination, visited, stk)
		}
		head = head.next
	}
	stk.Push(index)
}
```

Topology sort is DFS traversal of topology graph. First, the children of node are added to the stack then only the current node is added. So the sorting order is maintained. Reader is requested to run some examples to understand this algo.

Minimum Spanning Trees (MST)

A <u>Spanning Tree</u> of a graph G is a tree that contains all the edges of the Graph G.
A <u>Minimum Spanning Tree</u> is a spanning-tree whose sum of length/weight of edges is minimum as possible.

For example, if you want to setup communication between a set of cities, then you may want to use the least amount of wire as possible. MST can be used to find the network path and wire cost estimate.



Prim's Algorithm for MST

Prim's algorithm grows a single tree T, one edge at a time, until it becomes a spanning tree.
We initialize T with zero edges and U with single node. Where T is spanning tree edges set and U is spanning tree vertex set.

At each step, Prim's algorithm adds the smallest value edge with one endpoint in U and other not in us. Since each edge adds one new vertex to U, after n − 1 additions, U contain all the vertices of the spanning tree and T becomes a spanning tree.

Example 13.8:

```
// Returns the MST by Prim's Algorithm
// Input: A weighted connected graph G = (V, E)
// Output: Set of edges comprising a MST

Algorithm Prim(G)
    T = {}
    Let r be any vertex in G
    U = {r}
    for i = 1 to |V| - 1 do
        e = minimum-weight edge (u, v)
            With u in U and v in V-U
        U = U + {v}
        T = T + {e}
    return T
```

Prim's Algorithm using a priority queue (min heap) to get the closest fringe vertex
Time Complexity will be O(m log n) where n vertices and m edges of the MST.

Example 13.9:

```
func (g *Graph) Prims() {
	count := g.count
	previous := make([]int, count)
	dist := make([]int, count)
	que := new(PQueue)
	que.Init()
	source := 1
	for i := 0; i < count; i++ {
		previous[i] = -1
		dist[i] = Infinite
	}

	dist[source] = 0
	edge := &Edge{source, source, 0, nil}
	que.Push(edge, edge.cost)

	for que.Len() != 0 {
		edge = que.Pop().(*Edge) // Pop from PQ
		if dist[edge.destination] < edge.cost {
			continue
		}
		dist[edge.destination] = edge.cost
		previous[edge.destination] = edge.source
		adl := g.VertexList[edge.destination]
		adn := adl.head
		for adn != nil {
			if previous[adn.destination] == -1 {
				edge = &Edge{adn.source, adn.destination, adn.cost, nil}
				que.Push(edge, edge.cost)
			}
			adn = adn.next
		}
	}
	// Printing result.
	for i := 0; i < count; i++ {
		if dist[i] == Infinite {
			fmt.Println(" edge id ", i, "  prev ", previous[i], " distance : Unreachable")
		} else {
			fmt.Println(" edge id ", i, "  prev ", previous[i], " distance : ", dist[i])
		}
	}
}
```

Kruskal’s Algorithm

Kruskal’s Algorithm repeatedly chooses the smallest-weight edge that does not form a cycle.
Sort the edges in non-decreasing order of cost: $c(e1) \leq c(e2) \leq \cdots \leq c(em)$.
Set T to be the empty tree. Add edges to tree one by one, if it does not create a cycle.

Example 13.10:

```
// Returns the MST by Kruskal’s Algorithm
// Input: A weighted connected graph G = (V, E)
```

```
// Output: Set of edges comprising a MST

Algorithm Kruskal(G)
    Sort the edges E by their weights
    T = { }
    while |T | + 1 < |V | do
        e = next edge in E
        if T + {e} does not have a cycle then
            T = T + {e}
    return T
```

Kruskal’s Algorithm is O(E log V) using efficient cycle detection.

Shortest Path Algorithms in Graph

Single Source Shortest Path

For a graph G= (V, E), the single source shortest path problem is to find the shortest path from a given source vertex s to all the vertices of V.

Single Source Shortest Path for unweighted Graph.

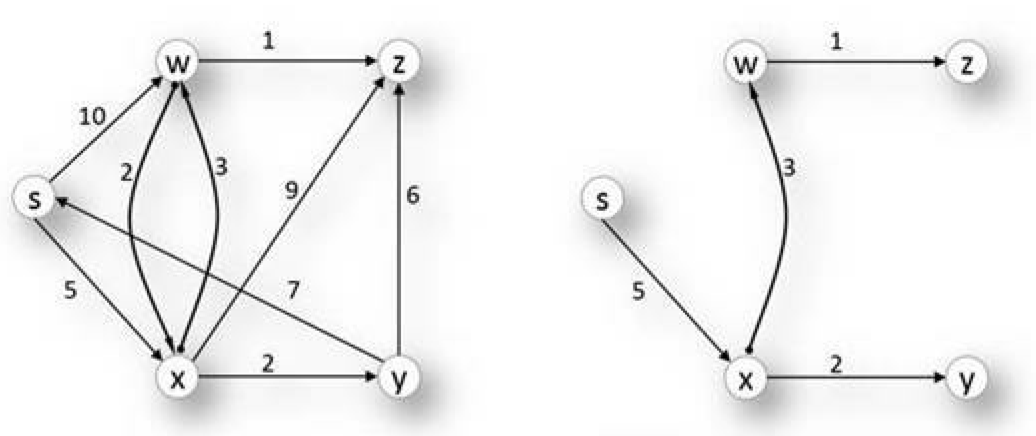
Find single source shortest path for unweighted graph or a graph with all the vertices of same weight.

Example 13.11:

```
func (g *Graph) ShortestPath(source int) {
	var curr int
	count := g.count
	distance := make([]int, count)
	path := make([]int, count)
	que := new(queue.Queue)
	for i := 0; i < count; i++ {
		distance[i] = -1
	}
	que.Enqueue(source)
	distance[source] = 0
	path[source] = source
	for que.Len() != 0 {
		curr = que.Dequeue().(int)
		head := g.VertexList[curr].head
		for head != nil {
			if distance[head.destination] == -1 {
				distance[head.destination] = distance[curr] + 1
				path[head.destination] = curr
				que.Enqueue(head.destination)
			}
			head = head.next
		}
	}
	for i := 0; i < count; i++ {
		fmt.Println(path[i], " to ", i, " weight ", distance[i])
	}
}
```

Dijkstra’s algorithm

Dijkstra’s algorithm for single-source shortest path problem for weighted edges with no negative weight. Given a weighted connected graph G, find shortest paths from the source vertex s to each of the other vertices. Dijkstra’s algorithm is similar to prims algorithm. It maintains a set of nodes for which shortest path is known.



Single-Source shortest path

The algorithm starts by keeping track of the distance of each node and its parents. All the distance is set to infinite in the beginning as we do not know the actual path to the nodes and parents of all the vertices are set to null. All the vertices are added to a priority queue (min heap implementation)
At each step algorithm takes one vertex from the priority queue (which will be the source vertex in the beginning). Then update the distance list corresponding to all the adjacent vertices. When the queue is empty, then we will have the distance and parent list fully populated.

Example 13.12:

```
// Solves SSSP by Dijkstra's Algorithm
// Input: A weighted connected graph G = (V, E)
// with no negative weights, and source vertex v
// Output: The length and path from s to every v

Algorithm Dijkstra(G, s)
for each v in V do
    D[v] = infinite   // Unknown distance
    P[v] = null //unknown previous node
    add v to PQ //adding all nodes to priority queue

D[source] = 0 // Distance from source to source

while (PQ is not empty)
    u = vertex from PQ with smallest D[u]
    remove u from PQ
    for each v adjacent from u do
        alt = D[u] + length ( u , v)
        if  alt < D[v] then
            D[v] = alt
            P[v] = u
Return D[] , P[]
```

Time Complexity will be O(|E|log|V|)

Note: Dijkstra's algorithm does not work for graphs with negative edges weight.
Note: Dijkstra's algorithm is applicable to both undirected and directed graphs.

Example 13.13:

```
func (g *Graph) Dijkstra(source int) {
	count := g.count
	previous := make([]int, count)
	dist := make([]int, count)
	que := new(PQueue)
	que.Init()
	for i := 0; i < count; i++ {
		previous[i] = -1
		dist[i] = Infinite
	}

	dist[source] = 0
	edge := &Edge{source, source, 0, nil}
	que.Push(edge, edge.cost)

	for que.Len() != 0 {
		edge = que.Pop().(*Edge) // Pop from PQ
		if dist[edge.destination] < edge.cost {
			continue
		}
		dist[edge.destination] = edge.cost
		previous[edge.destination] = edge.source
		adl := g.VertexList[edge.destination]
		adn := adl.head
		for adn != nil {
			if previous[adn.destination] == -1 {
				edge = &Edge{adn.source, adn.destination, adn.cost + dist[adn.source], nil}
				que.Push(edge, edge.cost)
			}
			adn = adn.next
		}
	}
	// Printing result.
	for i := 0; i < count; i++ {
		if dist[i] == Infinite {
			fmt.Println(" edge id ", i, "  prev ", previous[i], " distance : Unreachable")
		} else {
			fmt.Println(" edge id ", i, "  prev ", previous[i], " distance : ", dist[i])
		}
	}
}
```

Bellman Ford Shortest Path

The bellman ford algorithm works even when there are negative weight edges in the graph. It does not work if there is some cycle in the graph whose total weight is negative.

Example 13.14:

```
func (g *Graph) BellmanFordShortestPath(source int) {
	count := g.count
	distance := make([]int, count)
	path := make([]int, count)
```

```
    for i := 0; i < count; i++ {
        distance[i] = Infinite
    }
    distance[source] = 0
    path[source] = source
    for i := 0; i < count-1; i++ {
        for j := 0; j < count; j++ {
            head := g.VertexList[j].head
            for head != nil {
                newDistance := distance[j] + head.cost
                if distance[head.destination] > newDistance {
                    distance[head.destination] = newDistance
                    path[head.destination] = j
                }
                head = head.next
            }
        }
    }
    for i := 0; i < count; i++ {
        fmt.Println(path[i], " to ", i, " weight ", distance[i])
    }
}
```

All Pairs Shortest Paths

Given a weighted graph G(V, E), the all pair shortest path problem is to find the shortest path between all pairs of vertices u, v ϵ V. Execute n instances of single source shortest path algorithm for each vertex of the graph.

The complexity of this algorithm will be $O(n^3)$

Exercise

1. In the various path-finding algorithms, we have created a path array that just store immediate parent of a node, print the complete path for it.

2. All the functions are implemented considering as if the graph is represented by adjacency list. To write all those functions for graph representation as adjacency matrix.

3. Given a start string, end string and a set of strings, find if there exists a path between the start string and end string via the set of strings.

   A path exists if we can get from start string to end the string by changing (no addition/removal) only one character at a time. The restriction is that the new string generated after changing one character has to be in the set.

   Start: "cog"
   End: "bad"
   Set: ["bag", "cag", "cat", "fag", "con", "rat", "sat", "fog"]
   One of the paths: "cog" -> "fog" -> "fag" -> "bag" -> "bad"

CHAPTER 14: STRING ALGORITHMS

Introduction

A String is a sequence of Unicode character. String is an immutable type variable. Strings are declared using double quotes.

```
s := "hello, World!"
```

Strings are immutable so you cannot change its content once created. You need to first convert into a slice of rune then do the changes and in the end convert it back to string.

Example 14.1:

```
func main() {
	s := "hello, World!"
	r := []rune(s)
	r[0] = 'H'
	s2 := string(r)
	fmt.Println(s2)
}
```

Below is table, which explain some of the operations on string.

```
mystring := “Hello World!”
```

Expression	Value	Explanation
len(mystring)	12	Used to find the number of characters in mystring
“hello”+“world”	“helloworld”	Two strings are concatenated into a single string
“world” == “hello” “world” == “world”	False True	Equality can be tested using “==” sign
“a” < “b” “b” < “a”	True False	Unicode value can also be used to compare
mystring[0]	“h”	Indexing: String are indexed same as array.
mystring[1:4]	"ell"	Slicing

String Matching

Every word processing program has a search function in which you can search all occurrences of any particular word in a long text file. For this, we need string-matching algorithms.

Brute Force Search

We have a pattern that we want to search in the text. The pattern is of length m and the text is of length n. Where m < n.

The brute force search algorithm will check the pattern at all possible value of "i" in the text where the value of "i" range from 0 to n-m. The pattern is compared with the text, character by character from left to right. When a mismatch is detected, then pattern is compared by shifting the compare window by one character.

Example 14.2:

```
func BruteForceSearch(text string, pattern string) int {
	j := 0
	n := len(text)
	m := len(pattern)
	for i := 0; i <= n-m; i++ {
		j = 0
		for j < m && pattern[j] == text[i+j] {
			j++
		}
		if j == m {
			return i
		}
	}
	return -1
}
```

Worst case Time Complexity of the algorithm is O(m*n), we got the pattern at the end of the text or we did not get the pattern at all**.**
Best case Time Complexity of this algorithm is O(m), The **average case** Time Complexity of this algorithm is O(n)

Robin-Karp algorithm

Robin-Karp algorithm is somewhat similar to the brute force algorithm. Because the pattern is compared to each portion of text of length m. Instead of pattern at each position a hash code is compared, only one comparison is performed. The hash code of the pattern is compared with the hash code of the text window. We try to keep the hash code as unique as possible.

The two features of good hash code are:

- The collision should be excluded as much as possible. A collision occurs when hash code matches, but the pattern does not.
- The hash code of text must be calculated in constant time.

Hash value of text of length m is calculated. Each time we exclude one character and include next character. The portion of text that will be compared move as a window of characters. For each window calculation of hash is done in constant time, one member leaves the window and a new number enters a window.

Multiplication by 2 is same as left shift operation. Multiplication by 2^{m-1} is same as left shift m-1 times. We want this multiple times so just store it in variable pow(m) = 2^{m-1}

We do not want to do big multiplication operations so modular operation with a prime number is used.

Example 14.3:
```
func RobinKarp(text string, pattern string) int {
	n := len(text)
	m := len(pattern)
	prime := 101
	powm := 1
	TextHash := 0
	PatternHash := 0
	i, j := 0, 0
	if m == 0 || m > n {
		return -1
	}

	for i = 0; i < m-1; i++ {
		powm = (powm << 1) % prime
	}

	for i = 0; i < m; i++ {
		PatternHash = ((PatternHash << 1) + (int)(pattern[i])) % prime
		TextHash = ((TextHash << 1) + (int)(text[i])) % prime
	}

	for i = 0; i <= (n - m); i++ {
		if TextHash == PatternHash {
			for j = 0; j < m; j++ {
				if text[i+j] != pattern[j] {
					break
				}
			}
			if j == m {
				return i
			}
		}
		TextHash = (((TextHash - (int)(text[i])*powm) << 1) + (int)(text[i+m])) % prime
		if TextHash < 0 {
			TextHash = (TextHash + prime)
		}
	}
	return -1
}
```

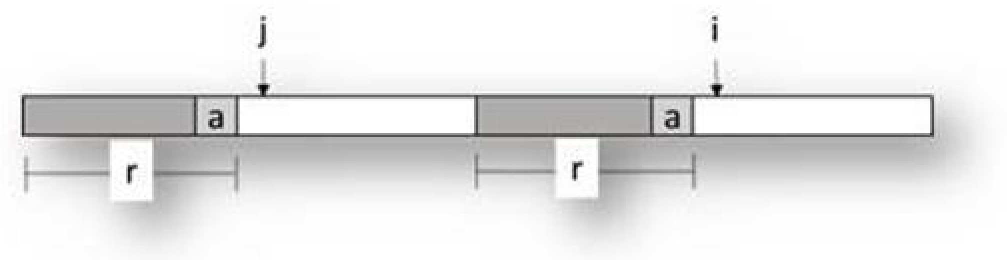
Knuth-Morris-Pratt algorithm

There is an inefficiency in the brute force method of string matching. After a shift of the pattern, the brute force algorithm forgotten all the information about the previous matched symbols. This is because of which its worst case Time Complexity is O(mn).

The Knuth-Morris-Pratt algorithm make use of this information that is computed in the previous comparison. It never re compares the whole text. It uses preprocessing of the pattern. The preprocessing takes O(m) time and whole algorithm is O(n)

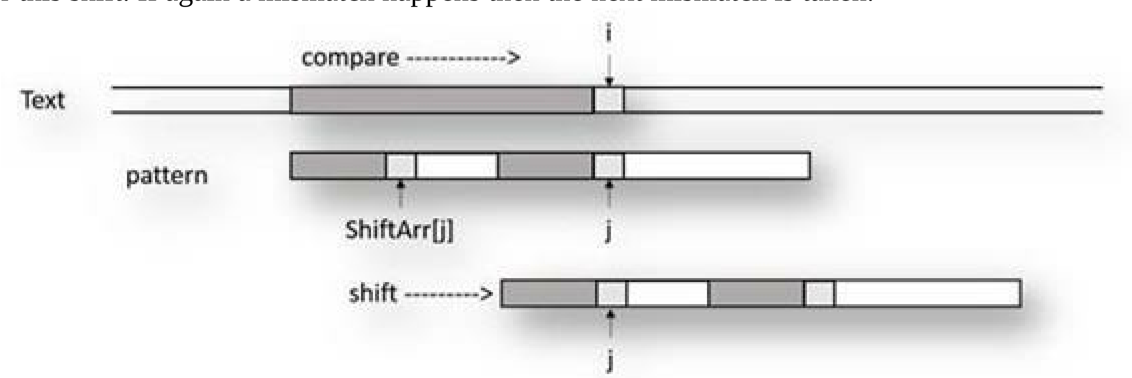
Preprocessing step: we try to find the border of the pattern at a different prefix of the pattern.

A **prefix** is a string that comes at the start of a string.
A **proper prefix** is a prefix that is not the complete string. Its length is less than the length of the string.
A **suffix** is a string that comes at the end of a string.
A **proper suffix** is a suffix that is not the complete string. Its length is less than the length of the string.
A **border** is a string that is both proper prefix and a proper suffix.



Example 14.4:
```
func KMPPreprocess(pattern string, ShiftArr []int) {
    m := len(pattern)
    i := 0
    j := -1
    ShiftArr[i] = -1
    for i < m {
        for j >= 0 && pattern[i] != pattern[j] {
            j = ShiftArr[j]
        }
        i++
        j++
        ShiftArr[i] = j
    }
}
```

We have to loop outer loop for the text and inner loop for the pattern when we have matched the text and pattern mismatch, we shift the text such that the widest border is considered and then the rest of the pattern matching is resumed after this shift. If again a mismatch happens then the next mismatch is taken.



Example 14.5:
```
func KMP(text string, pattern string) int {
    i, j := 0, 0
    n := len(text)
    m := len(pattern)
    ShiftArr := make([]int, m+1)
```

```
    KMPPreprocess(pattern, ShiftArr)
    for i < n {
        for j >= 0 && text[i] != pattern[j] {
            j = ShiftArr[j]
        }
        i++
        j++
        if j == m {
            return (i - m)
        }
    }
    return -1
}
```

Example 14.6: Use the same KMP algorithm to find the number of occurrences of the pattern in a text.

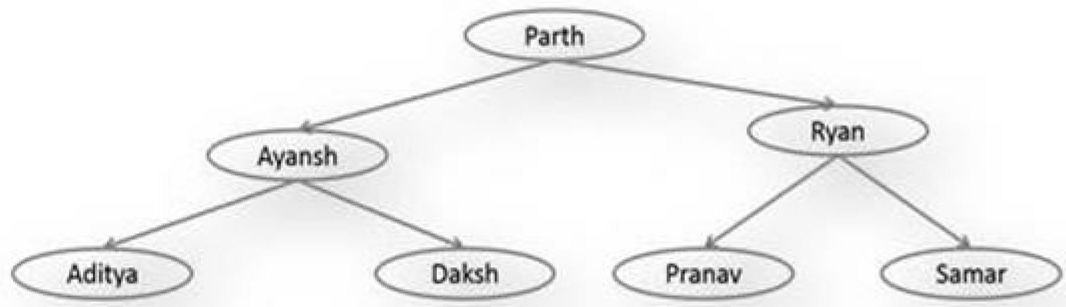
```
func KMPFindCount(text string, pattern string) int {
    i, j := 0, 0
    count := 0
    n := len(text)
    m := len(pattern)
    ShiftArr := make([]int, m+1)
    KMPPreprocess(pattern, ShiftArr)
    for i < n {
        for j >= 0 && text[i] != pattern[j] {
            j = ShiftArr[j]
        }
        i++
        j++
        if j == m {
            count++
            j = ShiftArr[j]
        }
    }
    return count
}
```

Dictionary / Symbol Table

A symbol table is a mapping between a string (key) and a value that can be of any type. A value can be an integer such as occurrence count, dictionary meaning of a word and so on.

Binary Search Tree (BST) for Strings

Binary Search Tree (BST) is the simplest way to implement symbol table. Simple strcmp() function can be used to compare two strings. If all the keys are random, and the tree is balanced. Then on an average key lookup can be done in O(logn) time.



BINARY SEARCH TREE AS DICTIONARY

Below is an implementation of binary search tree to store string as key. This will keep track of the occurrence count of words in a text.

Example 14.7:

```
type Node struct {
    value  string
    count  int
    lChild *Node
    rChild *Node
}

type StringTree struct {
    root *Node
}

func (t *StringTree) print() {
    t.printUtil(t.root)
}

func (t *StringTree) printUtil(curr *Node) {
    if curr != nil {
        fmt.Println(" value is ::", curr.value)
        fmt.Println(" count is :: ", curr.count)
        t.printUtil(curr.lChild)
        t.printUtil(curr.rChild)
    }
}
```

```
func (t *StringTree) Insert(value string) {
	t.root = t.insertUtil(value, t.root)
}

func (t *StringTree) insertUtil(value string, curr *Node) *Node {
	var compare int
	if curr == nil {
		curr = new(Node)
		curr.value = value
	} else {
		compare = strings.Compare(curr.value, value)
		if compare == 0 {
			curr.count++
		} else if compare == 1 {
			curr.lChild = t.insertUtil(value, curr.lChild)
		} else {
			curr.rChild = t.insertUtil(value, curr.rChild)
		}
	}
	return curr
}

func (t *StringTree) freeTree() {
	t.root = nil
}

func (t *StringTree) Find(value string) bool {
	ret := t.findUtil(t.root, value)
	fmt.Println("Find ", value, " Return ", ret)
	return ret
}

func (t *StringTree) findUtil(curr *Node, value string) bool {
	var compare int
	if curr == nil {
		return false
	}
	compare = strings.Compare(curr.value, value)
	if compare == 0 {
		return true
	}

	if compare == 1 {
		return t.findUtil(curr.lChild, value)
	}
	return t.findUtil(curr.rChild, value)
}

func (t *StringTree) frequency(value string) int {
	return t.frequencyUtil(t.root, value)
}
```

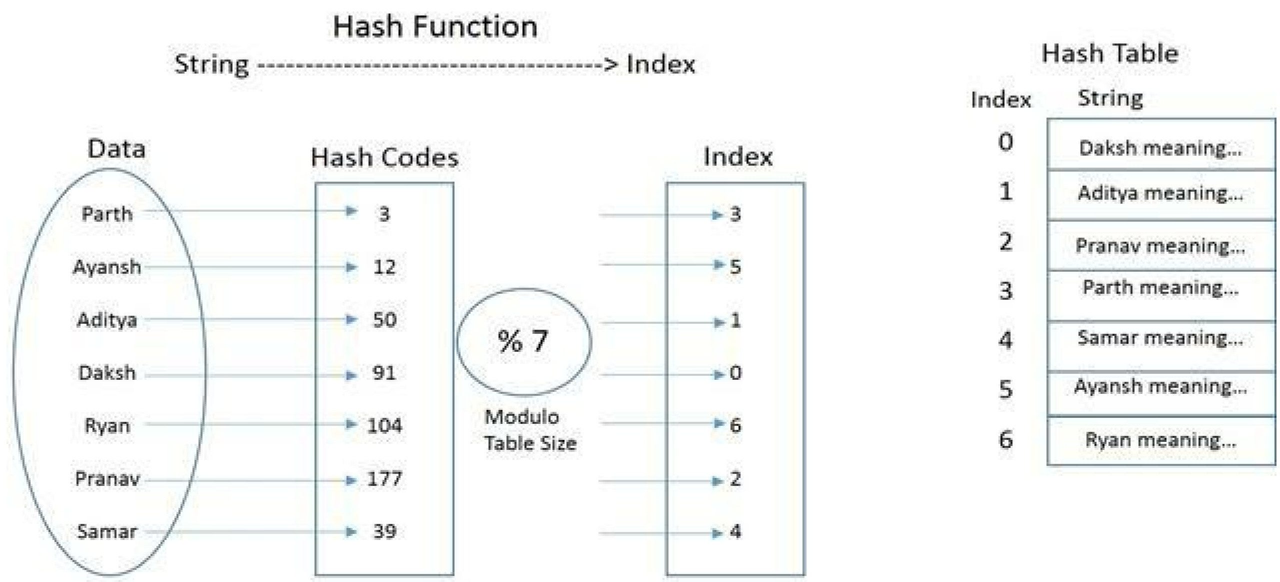
```
func (t *StringTree) frequencyUtil(curr *Node, value string) int {
	var compare int
	if curr == nil {
		return 0
	}
	compare = strings.Compare(curr.value, value)

	if compare == 0 {
		return curr.count
	}
	if compare > 0 {
		return t.frequencyUtil(curr.lChild, value)
	}
	return t.frequencyUtil(curr.rChild, value)
}

func main() {
	tt := new(StringTree)
	tt.Insert("banana")
	tt.Insert("apple")
	tt.Insert("mango")
	fmt.Println("Search results for apple, banana, grapes and mango :")
	tt.Find("apple")
	tt.Find("banana ")
	tt.Find("grapes")
	tt.Find("mango")
}
```

Output:

```
Find  apple  Return  true
Find  banana  Return  true
Find  grapes  Return  false
Find  mango  Return  true
```

Hash-Table

The Hash-Table is another data structure that can be used for symbol table implementation. Below Hash-Table diagram, we can see the name of that person is taken as the key, and their meaning is the value of the search. The first key is converted into a hash code by passing it to appropriate hash function. Inside hash function the size of Hash-Table is also passed, which is used to find the actual index where values will be stored. Finally, the value, which is meaning of name is stored in the Hash-Table, or you can store a reference to the string which store meaning can be stored into the Hash-Table.



Hash-Table has an excellent lookup of O(1).

Let us suppose we want to implement autocomplete the box feature of Google search. When you type some string to search in google search, it propose some complete string even before you have done typing. BST cannot solve this problem as related strings can be in both right and left subtree.

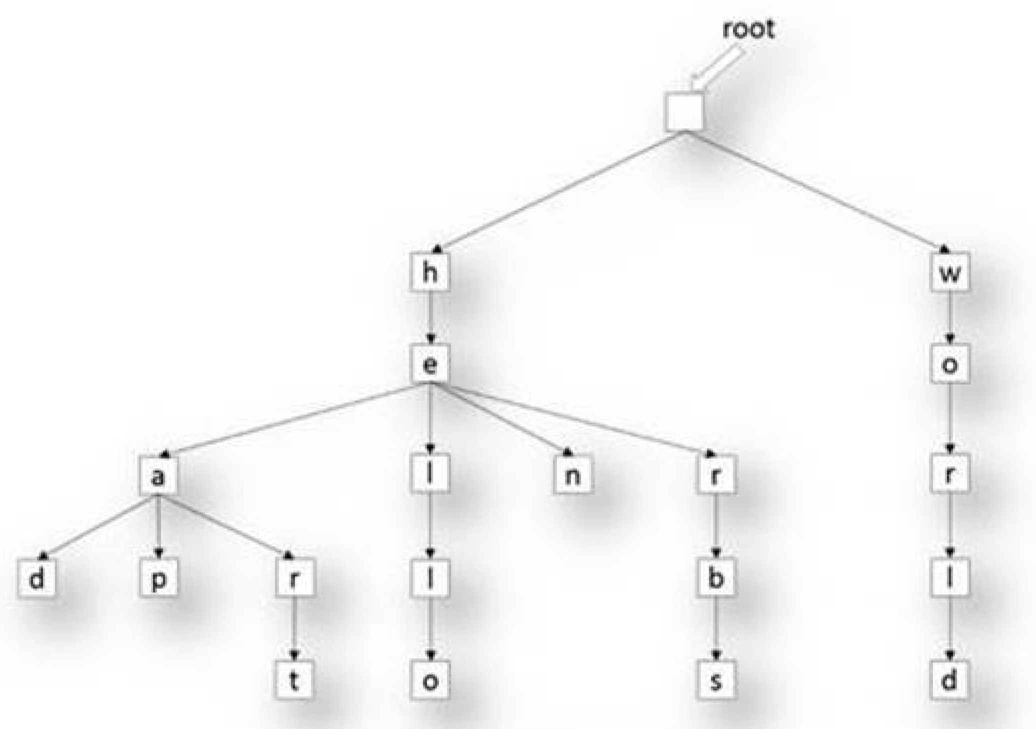
The Hash-Table is also not suited for this job. One cannot perform a partial match or range query on a Hash-Table. Hash function transforms string to a number. Moreover, a good hash function will give a distributed hash code even for partial string and there is no way to relate two strings in a Hash-Table.

Trie and Ternary Search tree are a special kind of tree that solves partial match and range query problem well.

Trie

Trie is a tree, in which we store only one character at each node. The final key value pair is stored in the leaves. Each node has R children, one for each possible character. For simplicity purpose, let us consider that the character set is 26, corresponds to different characters of English alphabets.

Trie is an efficient data structure. Using Trie, we can search the key in O(M) time. Where M is the maximum string length. Trie is also suitable for solving partial match and range query problems.



TRIE DICTIONARY

Example 14.8:
```
package main

import (
	"fmt"
	"strings"
)

type Node struct {
	isLastChar bool
	child   [26](*Node)
}

type Trie struct {
	root *Node
}

func (t *Trie) Insert(s string) {
	if s == "" {
		return
	}

	str := strings.ToLower(s)
	t.root = t.InsertUtil(t.root, str, 0)
}

func (t *Trie) InsertUtil(curr *Node, str string, index int) *Node {
	if curr == nil {
		curr = new(Node)
	}
```

```
    if len(str) == index {
        curr.isLastChar = true
    } else {
        curr.child[str[index]-'a'] = t.InsertUtil(curr.child[str[index]-'a'], str, index+1)
    }
    return curr
}

func (t *Trie) Remove(s string) {
    if s == "" {
        return
    }

    str := strings.ToLower(s)
    t.RemoveUtil(t.root, str, 0)
}

func (t *Trie) RemoveUtil(curr *Node, str string, index int) {
    if curr == nil {
        return
    }

    if len(str) == index {
        if curr.isLastChar {
            curr.isLastChar = false
        }
        return
    }
    t.RemoveUtil(curr.child[str[index]-'a'], str, index+1)
}

func (t *Trie) Find(s string) bool {
    if s == "" {
        return false
    }
    str := strings.ToLower(s)
    return t.FindUtil(t.root, str, 0)
}

func (t *Trie) FindUtil(curr *Node, str string, index int) bool {
    if curr == nil {
        return false
    }
    if len(str) == index {
        return curr.isLastChar
    }
    return t.FindUtil(curr.child[str[index]-'a'], str, index+1)
}

func main() {
    t := new(Trie)
    a := "apple"
    b := "app"
```

```
    c := "appletree"
    d := "tree"
    t.Insert(a)
    t.Insert(d)
    fmt.Println(t.Find(a))
    fmt.Println(t.Find(b))
    fmt.Println(t.Find(c))
    fmt.Println(t.Find(d))
}
```

Output:

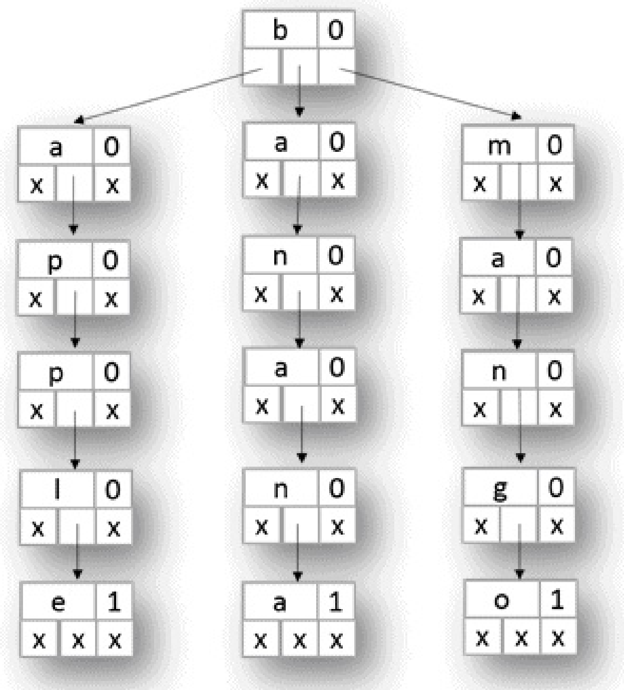
```
true
false
false
true
```

Ternary Search Trie/ Ternary Search Tree

Tries having a very good search performance of O(M) where M is the maximum size of the search string. However, tries having very high space requirement. Every node Trie contain references to multiple nodes, each reference corresponds to possible characters of the key. To avoid this high space requirement Ternary Search Trie (TST) is used.

A TST avoid the heavy space requirement of the traditional Trie while still keeping many of its advantages. In a TST, each node contains a character, an end of key indicator, and three references. The three references are corresponding to current char hold by the node(equal), characters less than and character greater than.

The Time Complexity of ternary search tree operation is proportional to the height of the ternary search tree. In the worst case, we need to traverse up to 3 times that many links. However, this case is rare. Therefore, TST is a very good solution for implementing Symbol Table, Partial match and range query.



Ternary Search Tree

Example 14.9:
```
type Node struct {
	data 		byte
	isLastChar 	bool
	left, equal, right *Node
}
type TST struct {
	root *Node
}

func (t *TST) Insert(word string) {
	t.root = t.insertUtil(t.root, word, 0)
}

func (t *TST) insertUtil(curr *Node, word string, wordIndex int) *Node {
	if curr == nil {
		curr = new(Node)
		curr.data = word[wordIndex]
	}
	if word[wordIndex] < curr.data {
		curr.left = t.insertUtil(curr.left, word, wordIndex)
	} else if word[wordIndex] > curr.data {
		curr.right = t.insertUtil(curr.right, word, wordIndex)
	} else {
		if wordIndex < len(word)-1 {
			curr.equal = t.insertUtil(curr.equal, word, wordIndex+1)
		} else {
			curr.isLastChar = true
		}
```

```
    }
    return curr
}

func (t *TST) findUtil(curr *Node, word string, wordIndex int) bool {
    if curr == nil {
        return false
    }
    if word[wordIndex] < curr.data {
        return t.findUtil(curr.left, word, wordIndex)
    } else if word[wordIndex] > curr.data {
        return t.findUtil(curr.right, word, wordIndex)
    } else {
        if wordIndex == len(word)-1 {
            return curr.isLastChar
        }
        return t.findUtil(curr.equal, word, wordIndex+1)
    }
}

func (t *TST) Find(word string) bool {
    ret := t.findUtil(t.root, word, 0)
    fmt.Print(word, " :: ")
    if ret {
        fmt.Println(" Found ")
    } else {
        fmt.Println("Not Found ")
    }
    return ret
}

func main() {
    tt := new(TST)
    tt.Insert("banana")
    tt.Insert("apple")
    tt.Insert("mango")
    fmt.Println("Search results for apple, banana, grapes and mango :")
    tt.Find("apple")
    tt.Find("banana")
    tt.Find("mango")
    tt.Find("grapes")
}
```

Output:

```
Search results for apple, banana, grapes and mango :
apple ::  Found
banana ::  Found
mango ::  Found
grapes :: Not Found
```

Problems in String

Regular Expression Matching

Implement regular expression matching with the support of ‘?’ and ‘*’ special character.
‘?’ Matches any single character.
‘*’ Matches zero or more of the preceding element.

Example 14.10:

```
func matchExpUtil(exp string, str string, i int, j int) bool {
	if i == len(exp) && j == len(str) {
		return true
	}
	if (i == len(exp) && j != len(str)) || (i != len(exp) && j == len(str)) {
		return false
	}
	if exp[i] == '?' || exp[i] == str[j] {
		return matchExpUtil(exp, str, i+1, j+1)
	}
	if exp[i] == '*' {
		return matchExpUtil(exp, str, i+1, j) || matchExpUtil(exp, str, i, j+1) || matchExpUtil(exp, str, i+1, j+1)
	}
	return false
}

func matchExp(exp string, str string) bool {
	return matchExpUtil(exp, str, 0, 0)
}
```

Order Matching

Given a long text string and a pattern string. Find if the characters of pattern string are in the same order in text string.
Eg. Text String: ABCDEFGHIJKLMNOPQRSTUVWXYZ Pattern string: JOST

Example 14.11:

```
// Match if the pattern is present in the source text.
func match(source string, pattern string) int {
	iSource := 0
	iPattern := 0
	sourceLen := len(source)
	patternLen := len(pattern)
	for iSource = 0; iSource < sourceLen; iSource++ {
		if source[iSource] == pattern[iPattern] {
			iPattern++
		}
		if iPattern == patternLen {
			return 1
		}
	}
	return 0
}
```

Unique Characters

Write a function that will take a string as input and return 1 if it contain all unique characters else return 0.

Example 14.12:

```
func isUniqueChar(str string) bool {
	mp := make(map[byte]int)
	size := len(str)
	for i := 0; i < size; i++ {
		c := str[i]
		if mp[c] != 0 {
			fmt.Println("Duplicate detected!")
			return false
		}
		mp[c] = 1
	}
	fmt.Println("No duplicate detected!")
	return true
}
```

Permutation Check

Example 14.13: Function to check if two strings are permutation of each other.

```
func isPermutation(s1 string, s2 string) bool {
	count := make(map[byte]int)
	length := len(s1)
	if len(s2) != length {
		fmt.Println(s1, "&", s2, "are not permutation")
		return false
	}

	for i := 0; i < length; i++ {
		ch := s1[i]
		count[ch]++
		ch = s2[i]
		count[ch]--
	}
	for i := 0; i < length; i++ {
		ch := s1[i]
		if count[ch] != 0 {
			fmt.Println(s1, "&", s2, "are not permutation")
			return false
		}
	}
	fmt.Println(s1, "&", s2, "are permutation")
	return true
}
```

Palindrome Check

Example 14.14: Find if the string is a palindrome or not

```
func isPalindrome(str string) bool {
	i := 0
```

```
    j := len(str) - 1
    for i < j && str[i] == str[j] {
        i++
        j--
    }
    if i < j {
        fmt.Println("String is not a Palindrome")
        return false
    }
    fmt.Println("String is a Palindrome")
    return true
}
```

Time Complexity is **O(n)** and Space Complexity is **O(1)**

Power function

Example 14.15: Function which will calculate x^n, Taking x and n as argument.

```
func pow(x int, n int) int {
    var value int
    if n == 0 {
        return 1
    } else if n%2 == 0 {
        value = pow(x, n/2)
        return (value * value)
    } else {
        value = pow(x, n/2)
        return x * value * value
    }
}
```

String Compare function

Write a function strcmp() to compare two strings. The function return values should be:

a) The return value is 0 indicates that both first and second strings are equal.
b) The return value is negative indicates the first string is less than the second string.
c) The return value is positive indicates that the first string is greater than the second string.

Example 14.16:

```
func strcmp(a string, b string) int {
    index := 0
    len1 := len(a)
    len2 := len(b)
    minlen := len1
    if len1 > len2 {
        minlen = len2
    }

    for index < minlen && a[index] == b[index] {
        index++
    }
```

```
    if index == len1 && index == len2 {
        return 0
    } else if len1 == index {
        return -1
    } else if len2 == index {
        return 1
    }
    return (int)(a[index]) - (int)(b[index])
}
```

Reverse String

Example 14.17: Reverse all the characters of a string.

```
func reverseString(a string) string {
    chars := []rune(a)
    reverseStringUtil(chars)
    return string(chars)
}

func reverseStringUtil(a []rune) {
    lower := 0
    upper := len(a) - 1
    for lower < upper {
        a[lower], a[upper] = a[upper], a[lower]
        lower++
        upper--
    }
}
```

Reverse Words

Example 14.18: Reverse order of words in a string sentence.

```
func reverseStringRange(a []rune, lower int, upper int) {
    for lower < upper {
        a[lower], a[upper] = a[upper], a[lower]
        lower++
        upper--
    }
}

func reverseWords(str string) string {
    length := len(str)
    upper := -1
    lower := 0
    arr := []rune(str)
    for i := 0; i < length; i++ {
        if arr[i] == ' ' {
            reverseStringRange(arr, lower, upper)
            lower = i + 1
            upper = i
        } else {
            upper++
```

```
        }
    }
    reverseStringRange(arr, lower, upper)
    reverseStringRange(arr, 0, length-1)
    return string(arr)
}
```

Print Anagram

Example 14.19: Given a string as character list, print all the anagram of the string.

```
func printAnagram(a string) {
    n := len(a)
    printAnagramUtil([]rune(a), n, n)
}

func printAnagramUtil(a []rune, max int, n int) {
    if max == 1 {
        fmt.Println(string(a))
    }
    for i := -1; i < max-1; i++ {
        if i != -1 {
            a[i], a[max-1] = a[max-1], a[i]
        }
        printAnagramUtil(a, max-1, n)
        if i != -1 {
            a[i], a[max-1] = a[max-1], a[i]
        }
    }
}
```

Shuffle String

Example 14.20: Write a program to convert list ABCDE12345 to A1B2C3D4E5

```
func shuffle(arr string) string {
    ar := []rune(arr)
    n := len(ar) / 2
    count := 0
    k := 1
    var temp rune
    for i := 1; i < n; i = i + 2 {
        k = i
        temp = ar[i]
        for true {
            k = (2 * k) % (2*n - 1)
            temp, ar[k] = ar[k], temp
            count++
            if i == k {
                break
            }
        }
        if count == (2*n - 2) {
            break
```

```
        }
    }
    return string(ar)
}
```

Exercise

1. Given a string, find the longest substring without reputed characters.

2. The function memset() copies ch into the first 'n' characters of the string

3. Serialize a collection of string into a single string and de serializes the string into that collection of strings.

4. Write a smart input function, which take 20 characters as input from the user. Without cutting some word.
   User input: “Harry Potter must not go”
   First 20 chars: “Harry Potter must no”
   Smart input: “Harry Potter must”

5. Write a code that returns if a string is palindrome and it should return true for below inputs too.
   Stella won no wallets.
   No, it is open on one position.
   Rise to vote, Sir.
   Won't lovers revolt now?

6. Write an ASCII to integer function, which ignore the non-integral character and give the integer. For example, if the input is “12AS5” it should return 125.
7. Write code that would parse a Bash brace expansion.
   Example: the expression "(a, b, c) d, e" and would output all the possible strings: ad, bd, cd, e

8. Given a string write a function to return the length of the longest substring with only unique characters

9. Replace all occurrences of "a" with "the"

10. Replace all occurrences of %20 with ' '.
    E.g. Input: www.Hello%20World.com
    Output: www.Hello World. com

11. Write an expansion function that will take an input string like "1..5,8,11..14,18,20,26..30" and will print "1,2,3,4,5,8,11,12,13,14,18,20,26,27,28,29,30"

12. Suppose you have a string like "Thisisasentence". Write a function that would separate these words. Moreover, will print whole sentence with spaces.

13. Given three string str1, str2 and str3. Write a complement function to find the smallest sub-sequence in str1 which contains all the characters in str2 and but not those in str3.

14. Given two strings A and B, find whether any anagram of string A is a sub string of string B.
    For eg: If A = xyz and B = afdgzyxksldfm then the program should return true.

15. Given a string, find whether it contains any permutation of another string. For example, given "abcdefgh" and "ba", the function should return true, because "abcdefgh" has substring "ab", which is a permutation of the given string "ba".

16. Give an algorithm which removes the occurrence of “a” by “bc” from a string? The algorithm must be in-place.

17. Given a string "1010101010" in base2 convert it into string with base4. Do not use an extra space.

18. In Binary Search tree to store strings, delete() function is not implemented implement the same.

19. If you implement delete() function, then you need to make changes in find() function. Do the needful.

CHAPTER 15: ALGORITHM DESIGN TECHNIQUES

Introduction

In real life when we are asked to do some work, we try to correlate it with our experience and then try to solve it. Similarly, when we get a new problem to solve. We first try to find the similarity of the current problem with some problems for which we already know the solution. Then solve the current problem and get our desired result.

This method provides following benefits:

1) It provides a template for solving a wide range of problems.
2) It provides us the idea of the suitable data structure for the problem.
3) It helps us in analyzing, space and Time Complexity of algorithms.

In the previous chapters, we have used various algorithms to solve different kind of problems. In this chapter, we will read about various techniques of solving algorithmic problems.

Various Algorithm design techniques are:

1) Brute Force
2) Greedy Algorithms
3) Divide-and-Conquer, Decrease-and-Conquer
4) Dynamic Programming
5) Reduction / Transform-and-Conquer
6) Backtracking and Branch-and-Bound

Brute Force Algorithm

Brute Force is a straightforward approach of solving a problem based on the problem statement. It is one of the easiest approaches to solve a particular problem. It is useful for solving small size dataset problem.

Some examples of brute force algorithms are:

- Bubble-Sort
- Selection-Sort
- Sequential search in a list
- Computing pow(a, n) by multiplying a, n times.
- Convex hull problem
- String matching
- Exhaustive search: Traveling salesman, Knapsack, and Assignment problems

Greedy Algorithm

In greedy algorithm, solution is constructed through a sequence of steps. At each step, choice is made which is locally optimal. Greedy algorithms are generally used to solve optimization problems. We always take the next data to be processed depending upon the dataset which we have already processed and then choose the next optimum data to be processed. Greedy algorithms does not always give optimum solution.

Some examples of brute force algorithms are:

- Minimal spanning tree: Prim’s algorithm, Kruskal’s algorithm
- Dijkstra’s algorithm for single-source shortest path problem
- Greedy algorithm for the Knapsack problem
- The coin exchange problem
- Huffman trees for optimal encoding

Divide-and-Conquer, Decrease-and-Conquer

Divide-and-Conquer algorithms involve basic three steps, first split the problem into several smaller sub-problems, second solve each sub problem and then finally combine the sub problems results to produce the result.

In divide-and-conquer the size of the problem is reduced by a factor (half, one-third, etc.), While in decrease-and-conquer the size of the problem is reduced by a constant.

Examples of divide-and-conquer algorithms:

- Merge-Sort algorithm (using recursion)
- Quicksort algorithm (using recursion)
- Computing the length of the longest path in a binary tree (using recursion)
- Computing Fibonacci numbers (using recursion)
- Quick-hull

Examples of decrease-and-conquer algorithms:

- Computing pow(a, n) by calculating pow(a, n/2) using recursion.
- Binary search in a sorted list (using recursion)
- Searching in BST
- Insertion-Sort
- Graph traversal algorithms (DFS and BFS)
- Topological sort
- Warshall’s algorithm (using recursion)
- Permutations (Minimal change approach, Johnson-Trotter algorithm)
- Computing a median, Topological sorting, Fake-coin problem (Ternary search)

Consider the problem of exponentiation Compute x^n

Brute Force:	n-1 multiplications
Divide and conquer:	T(n) = 2*T(n/2) + 1 = n-1
Decrease by one:	T (n) = T (n-1) + 1 = n-1
Decrease by constant factor:	T (n) = T (n/a) + a-1 = (a-1) n = n when a = 2

Dynamic Programming

While solving problems using Divide-and-Conquer method, there may be a case when recursively sub-problems can result in the same computation being performed multiple times. This problem arises when there are identical sub-problems arise repeatedly in a recursion.

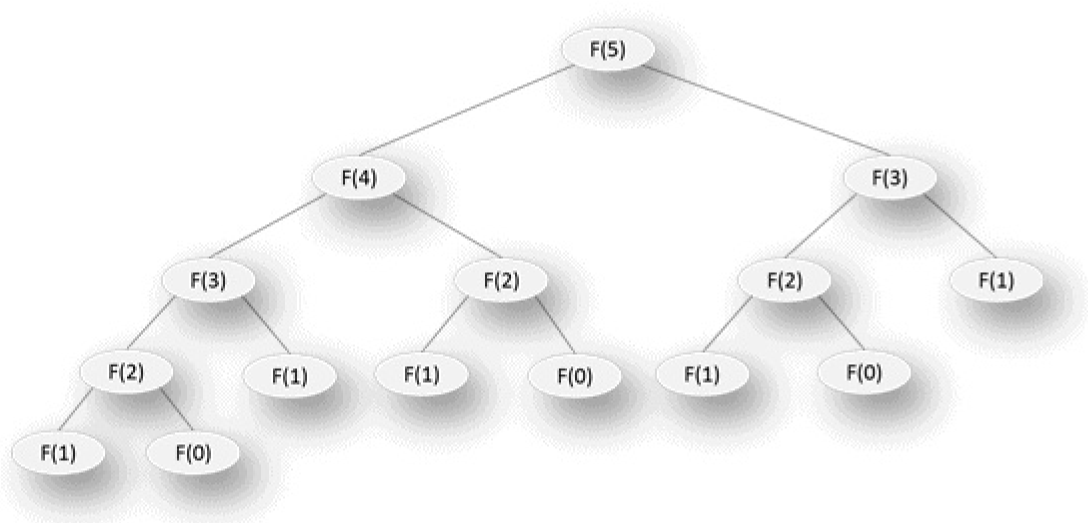
Dynamic programming is used to avoid the requirement of repeated calculation of same sub-problem. In this method, we usually store the result of sub - problems in a table and refer that table to find if we have already calculated the solution of sub - problems before calculating it again.

Dynamic programming is a bottom up technique in which the smaller sub-problems are solved first and the result of these are sued to find the solution of the larger sub-problems.

Examples:

- Fibonacci numbers computed by iteration.
- Warshall's algorithm for transitive closure implemented by iterations
- Floyd's algorithms for all-pairs shortest paths

```
func fibonacci(n int) int {
    if n <= 1 {
        return n
    }
    return fibonacci(n-1) + fibonacci(n-2)
}
```



Using divide and conquer the same sub problem is solved again and again, which reduce the performance of the algorithm. This algorithm has an exponential Time Complexity and linear Space Complexity.

```
func fibonacci(n int) int {
    first := 0
    second := 1
    temp := 0

    if n == 0 {
        return first
    } else if n == 1 {
        return second
    }
```

```
    for i := 2; i <= n; i++ {
        temp = first + second
        first = second
        second = temp
    }
    return temp
}
```

Using this algorithm, we will get Fibonacci in linear Time Complexity and constant Space Complexity.

Reduction / Transform-and-Conquer

These methods works as two-stage procedure. First, the problem is transformed into a known problem for which we know optimal solution. In the second stage, the problem is solved.

The most common types of transformation are sort of a list. For example, Given a list of numbers finds the two closest number.

The brute force solution for this problem will take distance between each element in the list and will try to keep the minimum distance pair; this approach will have a Time Complexity of $O(n^2)$

Transform and conquer solution, will be first sort the list in **O(nlogn)** time and then find the closest number by scanning the list in another **O(n)**. Which will give the total Time Complexity of **O(nlogn)**.

Examples:

- Gaussian elimination
- Heaps and Heapsort

Backtracking

In real life, let us suppose someone gave you a lock with a number (three digit lock, number range from 1 to 9). Moreover, you do not have the exact password key for the lock. You need to test every combination until you got the right one. Obviously, you need to test starting from something like “111”, then “112” and so on. You will get your key before you reach “999”. Therefore, what you are doing is backtracking.

Suppose the lock produce some sound “click” correct digit is selected for any level. If we can listen to this sound such intelligence/ heuristics will help you to reach your goal much faster. These functions are called Pruning function or bounding functions.

Backtracking is a method by which solution is found by exhaustively searching through large but finite number of states, with some pruning or bounding function that will narrow down our search.
For all the problems like (NP hard problems) for which there does not exist any other method we use backtracking.

Backtracking problems have the following components:

1. Initial state
2. Target / Goal state
3. Intermediate states
4. Path from the initial state to the target / goal state
5. Operators to get from one state to another
6. Pruning function (optional)

The solving process of backtracking algorithm starts with the construction of state’s tree, whose nodes represents the states. The root node is the initial state and one or more leaf node will be our target state. Each edge of the tree represents some operation. The solution is obtained by searching the tree until a Target state is found.

Backtracking uses depth-first search:

1) Store the initial state in a stack
2) While the stack is not empty, repeat:
3) Read a node from the stack.
4) While there are available operators, do:
    a. Apply an operator to generate a child
    b. If the child is a goal state – return solution
    c. If it is a new state, and pruning function does not discard it push the child into the stack.

There are three monks and three demons at one side of a river. We want to move all of them to the other side using a small boat. The boat can carry only two persons at a time. Given if on any shore the number of demons will be more than monks then they will eat the monks. How can we move all of these people to the other side of the river safely?

Same as the above problem there is a farmer who has a goat, a cabbage and a wolf. If the farmer leaves, goat with cabbage, goat will eat the cabbage. If the farmer leaves wolf alone with goat, wolf will kill the goat. How can the farmer move all his belongings to the other side of the river?

You are given two jugs, a 4-gallon one and a 3-gallon one. There are no measuring markers on jugs. A tap can be used to fill the jugs with water. How can you get 2 gallons of water in the 4-gallon jug?

Branch-and-bound

Branch and bound method is used when we can evaluate cost of visiting each node by a utility functions. At each step, we choose the node with lowest cost to proceed further. Branch-and bound algorithms are implemented using a priority queue. In branch and bound, we traverse the nodes in breadth-first manner.

A* Algorithm

A* is sort of an elaboration on branch-and-bound. In branch-and-bound, at each iteration we expand the shortest path that we have found so far. In A*, instead of just picking the path with the shortest length so far, we pick the path with the shortest estimated total length from start to goal, where the total length is estimated as length traversed so far plus a heuristic estimate of the remaining distance from the goal.

Branch-and-bound will always find an optimal solution, which is shortest path. A* will always find an optimal solution if the heuristic is correct. Choosing a good heuristic is the most important part of A* algorithm.

Conclusion

Usually a given problem can be solved using a number of methods; however, it is not wise to settle for the first method that comes to our mind. Some methods result in a much more efficient solution than others do.

For example, the Fibonacci numbers calculated recursively (decrease-and-conquer approach), and computed by iterations (dynamic programming). In the first case, the complexity is **$O(2^n)$,** and in the other case, the complexity is **$O(n)$**.

Another example, consider sorting based on the Insertion-Sort and basic bubble sort. For almost sorted files, Insertion-Sort will give almost linear complexity, while bubble sort sorting algorithms have quadratic complexity.

So the most important question is how to choose the best method?
First, you should understand the problem statement.
Second by knowing various problems and their solutions.

CHAPTER 16: BRUTE FORCE ALGORITHM

Introduction

Brute Force is a straightforward approach of solving a problem based on the problem statement. It is one of the easiest approaches to solve a particular problem. It is useful for solving small size dataset problem.

Many times, other algorithm techniques can be used to get a better solution of the same problem.

Some examples of brute force algorithms are:

- Bubble-Sort
- Selection-Sort
- Sequential search in a list
- Computing pow (a, n) by multiplying a, n times.
- Convex hull problem
- String matching
- Exhaustive search
- Traveling salesman
- Knapsack
- Assignment problems

Problems in Brute Force Algorithm

Bubble-Sort

In Bubble-Sort, adjacent elements of the list are compared and are exchanged if they are out of order.

```
// Sorts a given list by Bubble Sort
// Input: A list A of orderable elements
// Output: List A[0..n - 1] sorted in ascending order

Algorithm BubbleSort(A[0..n - 1])
    sorted = false
    while !sorted do
        sorted = true
        for j = 0 to n - 2 do
            if A[j] > A[j + 1] then
                swap A[j] and A[j + 1]
                sorted = false
```

The Time Complexity of the algorithm is $\Theta(n^2)$

Selection-Sort

The entire given list of N elements is traversed to find its smallest element and exchange it with the first element. Then, the list is traversed again to find the second element and exchanged it with the second element. After N-1 passes, the list will be fully sorted.

```
//Sorts a given list by selection sort
//Input: A list A[0..n-1] of orderable elements
//Output: List A[0..n-1] sorted in ascending order

Algorithm SelectionSort (A[0..n-1])
    for i = 0 to n - 2 do
        min = i
        for j = i + 1 to n - 1 do
            if A[j] < A[min]
                min = j
        swap A[i] and A[min]
```

The Time Complexity of the algorithm is $\Theta(n^2)$

Sequential Search

The algorithm compares consecutive elements of a given list with a given search keyword until either a match is found or the list is exhausted.

```
Algorithm SequentialSearch (A[0..n], K)
    i = 0
    While A [i] ≠ K do
        i = i + 1
    if  i < n
        return i
```

```
    else
        return -1
```

Worst case Time Complexity is Θ (n).

Computing pow (a, n)

Computing a^n (a > 0, and n is a nonnegative integer) based on the definition of exponentiation.
N-1 multiplications are required in brute force method.

```
// Input: A real number a and an integer n = 0
// Output: a power n

Algorithm Power(a, n)
    result = 1
    for i = 1 to n do
        result = result * a
    return result
```

The algorithm requires Θ (n)

String matching

A brute force string matching algorithm takes two inputs, first text consists of n characters and a pattern consist of m character (m<=n). The algorithm starts by comparing the pattern with the beginning of the text. Each character of the patters is compared to the corresponding character of the text. Comparison starts from left to right until all the characters are matched or a mismatch is found. The same process is repeated until a match is found. Each time the comparison starts one position to the right.

```
// Input: A list T[0..n - 1] of n characters representing a text
// a list P[0..m - 1] of m characters representing a pattern
// Output: The position of the first character in the text that starts the first
// matching substring if the search is successful and -1 otherwise.

Algorithm BruteForceStringMatch (T[0..n - 1], P[0..m - 1])
    for i = 0 to n - m do
    j = 0
    while j < m and P[j] = T[i + j] do
    j = j + 1
    if j = m then
        return i
    return -1
```

In the worst case, the algorithm is O(mn).

Closest-Pair Brute-Force Algorithm

The closest-pair problem is to find the two closest points in a set of n points in a 2-dimensional space.
A brute force implementation of this problem computes the distance between each pair of distinct points and find the smallest distance pair.

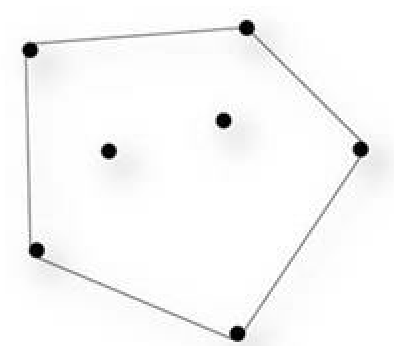
```
// Finds two closest points by brute force
// Input: A list P of n >= 2 points
// Output: The closest pair

Algorithm BruteForceClosestPair(P)
    dmin = infinite
    for i = 1 to n - 1 do
        for j = i + 1 to n do
            d = (xi - xj)^2 + (yi - yj)^2
            if d < dmin then
                dmin = d
                imin = i
                jmin = j
    return imin, jmin
```

In the Time Complexity of the algorithm is $\Theta(n^2)$

Convex-Hull Problem

Convex-hull of a set of points is the smallest convex polygon containing all the points. All the points of the set will lie on the convex hull or inside the convex hull. Illustrate the rubber-band interpretation of the convex hull. The convex-hull of a set of points is a subset of points in the given sets.

How to find this subset?
Answer: The rest of the points of the set are all on one side.



Two points (x1, y1), (x2, y2) make the line ax + by = c
Where a = y2-y1, b = x1-x2, and c = x1y2 - y1x2

And divides the plane by ax + by - c < 0 and ax + by - c > 0
So we need to only check ax + by - c for the rest of the points

If we find all the points in the set lies one side of the line with either all have ax + by - c < 0 or all the points have ax + by - c > 0 then we will add these points to the desired convex hull point set.
For each of n (n -1) /2 pairs of distinct points, one needs to find the sign of ax + by - c in each of the other n - 2 points.
What is the worst-case cost of the algorithm: O(n3)

```
Algorithm ConvexHull
  for i=0 to n-1
    for j=0 to n-1
      if (xi,yi) !=(xj,yj)
        draw a line from (xi,yi) to (xj,yj)
```

```
for k=0 to n-1
  if(i!=k and j!=k)
      if ( all other points lie on the same side of the
                         line (xi,yi) and (xj,yj))
        then add  (xi,yi) to (xj,yj) to the convex hull set
```

Exhaustive Search

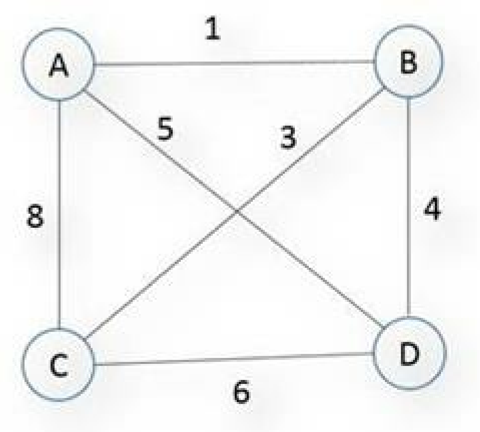
Exhaustive search is a brute force approach applies to combinatorial problems.
In exhaustive search, we generate all the possible combinations. See if the combinations satisfy the problem constraints and then finding the desired solution.
Examples of exhaustive search are:

- Traveling salesman problem
- Knapsack problem
- Assignment problem

Traveling Salesman Problem (TSP)

In the traveling salesman problem we need to find the shortest tour through a given set of N cities that salesperson visits each city exactly once before returning to the city where he started.

Alternatively, finding the shortest Hamiltonian circuit in a weighted connected graph. A cycle that passes through all the vertices of the graph exactly once.



Tours where A is starting city:

Tour	Cost
A→B→C→D→A	1+3+6+5 = 15
A→B→D→C→A	1+4+6+8 = 19
A→C→B→D→A	8+3+4+5 = 20
A→C→D→B→A	8+6+4+1 = 19
A→D→B→C→A	5+4+3+8 = 20
A→D→C→B→A	5+6+3+1 = 15

```
Algorithm TSP
    Select a city
    MinTourCost = infinite
    For ( All permutations of cities ) do
        If( LengthOfPathSinglePermutation < MinTourCost )
            MinTourCost = LengthOfPath
```

Total number of possible combinations = (n-1)!
Cost for calculating the path: Θ(n)
So the total cost for finding the shortest path: Θ(n!)

Knapsack Problem

Given an item with cost C1, C2,..., Cn, and volume V1, V2,..., Vn and knapsack of capacity Vmax, find the most valuable (max ∑Cj) that fit in the knapsack (∑Vj ≤ Vmax).

The solution is one of all the subset of the set of object taking 1 to n objects at a time, so the Time Complexity will be $O(2^n)$

```
Algorithm KnapsackBruteForce
    MaxProfit = 0
    For ( All permutations of objects ) do
        CurrProfit = sum of objects selected
        If( MaxProfit < CurrProfit )
            MaxProfit = CurrProfit
            Store the current set of objects selected
```

Conclusion

Brute force is the first algorithm that comes into mind when we see some problem. They are the simplest algorithms that are very easy to understand. However, these algorithms rarely provide an optimum solution. Many cases we will find other effective algorithm that is more efficient than the brute force method. This is the most simple to understand the kind of problem solving technique.

CHAPTER 17: GREEDY ALGORITHM

Introduction

Greedy algorithms are generally used to solve optimization problems. To find the solution that minimizes or maximizes some value (cost/profit/count etc.).

In greedy algorithm, solution is constructed through a sequence of steps. At each step, choice is made which is locally optimal. We always take the next data to be processed depending upon the dataset which we have already processed and then choose the next optimum data to be processed.

Greedy algorithms does not always give optimum solution. For some problems, greedy algorithm gives an optimal solution. For most, they do not, but can be useful for fast approximations.

Greedy is a strategy that works well on optimization problems with the following characteristics:
1. Greedy choice: A global optimum can be arrived at by selecting a local optimum.
2. Optimal substructure: An optimal solution to the problem is made from optimal solutions of sub problems.

Some examples of brute force algorithms are:
Optimal solutions:

- Minimal spanning tree:
  - Prim's algorithm,
  - Kruskal's algorithm
- Dijkstra's algorithm for single-source shortest path
- Huffman trees for optimal encoding
- Scheduling problems

Approximate solutions:

- Greedy algorithm for the Knapsack problem
- Coin exchange problem

Problems on Greedy Algorithm

Coin exchange problem

How can a given amount of money N be made with the least number of coins of given denominations D= {d1… dn}?

The Indian coin system {5, 10, 20, 25, 50,100}

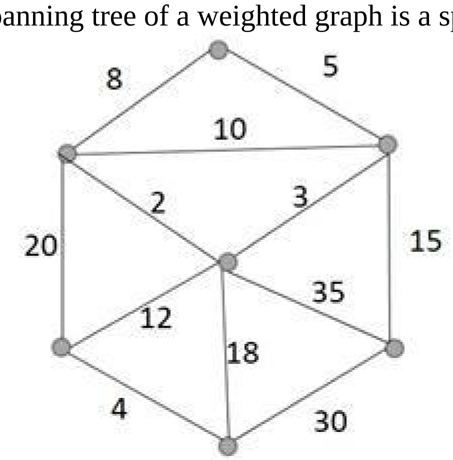
Suppose we want to give change of a certain amount of 40 paisa.

We can make a solution by repeatedly choosing a coin ≤ to the current amount, resulting in a new amount. In the greedy algorithm, we always choose the largest coin value possible without exceeding the total amount.
For 40 paisa: {25, 10, and 5}
The optimal solution will be {20, 20}
The greedy algorithm did not give us optimal solution, but it gave a fair approximation.

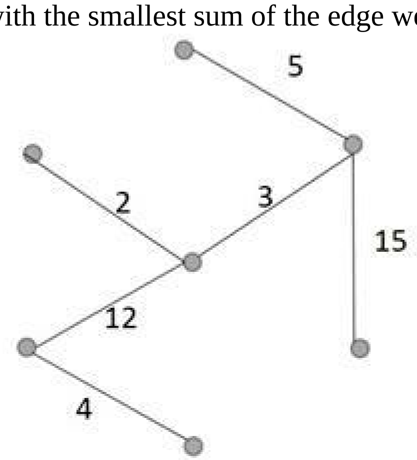
```
Algorithm  MAKE-CHANGE (N)
    C = {5, 20, 25, 50, 100}      // constant denominations.
    S = {}                  // set that will hold the solution set.
    Value = N
    WHILE Value != 0
        x = largest item in set C such that x < Value
        IF no such item THEN
            RETURN  "No Solution"
        S = S + x
        Value = Value - x
    RETURN S
```

Minimum Spanning Tree

A spanning tree of a connected graph is a tree containing all the vertices.
A minimum spanning tree of a weighted graph is a spanning tree with the smallest sum of the edge weights.



Graph



Minimum Spanning Tree

Prim’s Algorithm

Prim’s algorithm grows a single tree T, one edge at a time, until it becomes a spanning tree.
We initialize T with zero edges and U with single node. Where T is spanning tree edges set and U is spanning tree

vertex set.

At each step, Prim's algorithm adds the smallest value edge with one endpoint in U and other not in us.
Since each edge adds one new vertex to U, after n − 1 additions, U contain all the vertices of the spanning tree and T becomes a spanning tree.

```
// Returns the MST by Prim's Algorithm
// Input: A weighted connected graph G = (V, E)
// Output: Set of edges comprising a MST

Algorithm Prim(G)
    T = {}
    Let r be any vertex in G
    U = {r}
    for i = 1 to |V| - 1 do
        e = minimum-weight edge (u, v)
            With u in U and v in V-U
        U = U + {v}
        T = T + {e}
    return T
```

Prim's Algorithm using a priority queue (min heap) to get the closest fringe vertex
Time Complexity will be O(m log n) where n vertices and m edges of the MST.

Kruskal's Algorithm

Kruskal's Algorithm is used to create minimum spanning tree. Spanning tree is created by choosing smallest weight edge that does not form a cycle. And repeating this process until all the edges from the original set is exhausted.

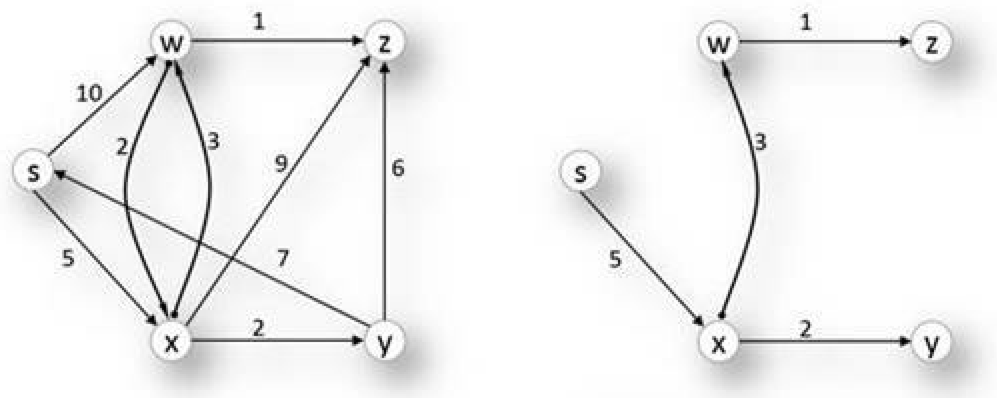
Sort the edges in non-decreasing order of cost: c (e1) ≤ c (e2) ≤ · · · ≤ c (em).
Set T to be the empty tree. Add edges to tree one by one if it does not create a cycle. (If the new edge form cycle then ignore that edge.)

```
// Returns the MST by Kruskal's Algorithm
// Input: A weighted connected graph G = (V, E)
// Output: Set of edges comprising a MST

Algorithm Kruskal(G)
    Sort the edges E by their weights
    T = {}
    while |T | + 1 < |V | do
        e = next edge in E
        if T + {e} does not have a cycle then
            T = T + {e}
    return T
```

Kruskal's Algorithm is O(E log V) using efficient cycle detection.

Dijkstra's algorithm for single-source shortest path problem

Dijkstra's algorithm for single-source shortest path problem for weighted edges with no negative weight. It determine the length of the shortest path from the source to each of the other nodes of the graph. Given a weighted graph G, we need to find shortest paths from the source vertex s to each of the other vertices.



Single-Source shortest path

The algorithm starts by keeping track of the distance of each node and its parents. All the distance is set to infinite in the beginning as we do not know the actual path to the nodes and parents of all the vertices are set to null. All the vertices are added to a priority queue (min heap implementation)

At each step algorithm takes one vertex from the priority queue (which will be the source vertex in the beginning). Then update the distance list corresponding to all the adjacent vertices. When the queue is empty, then we will have the distance and parent list fully populated.

```
// Solve SSSP by Dijkstra's Algorithm
// Input: A weighted connected graph G = (V, E)
// with no negative weights, and source vertex v
// Output: The length and path from s to every v

Algorithm Dijkstra(G, s)
    for each v in V do
        D[v] = infinite   // Unknown distance
        P[v] = null // Unknown previous node
        add v to PQ // Adding all nodes to priority queue
    D[source] = 0 // Distance from source to source
    while (PQ is not empty)
        u = vertex from PQ with smallest D[u]
        remove u from PQ
        for each v adjacent from u do
            alt = D[u] + length ( u , v)
            if  alt < D[v] then
                D[v] = alt
                P[v] = u
    Return D[] , P[]
```

Time Complexity will be O(|E|log|V|).

Note: Dijkstra's algorithm does not work for graphs with negative edges weight.

Note: Dijkstra's algorithm is applicable to both undirected and directed graphs.

Huffman trees for optimal encoding

Encoding is an assignment of bit strings of alphabet characters.

There are two types of encoding:

· Fixed-length encoding (eg., ASCII)
· Variable-length encoding (eg., Huffman code)

Variable length encoding can only work on prefix free encoding. Which means that no code word is a prefix of another code word.

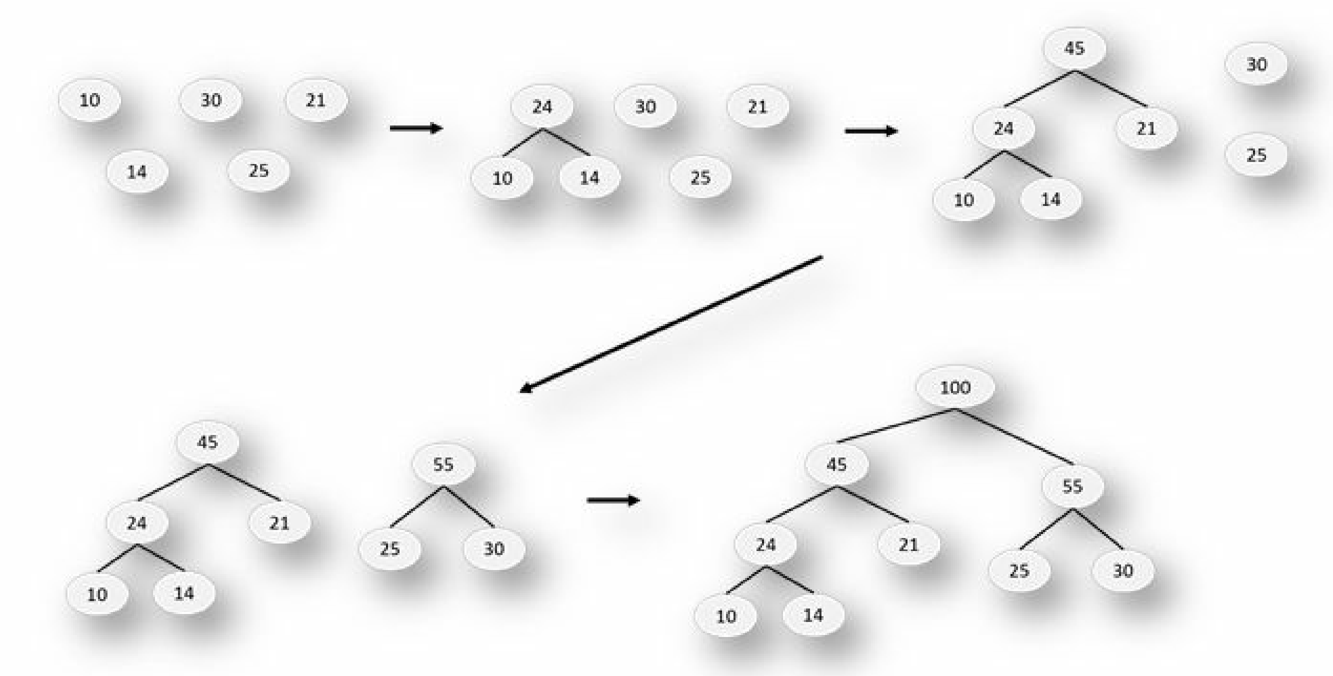
Huffman codes are the best prefix free code. Any binary tree with edges labeled as 0 and 1 will produce a prefix free code of characters assigned to its leaf nodes.

Huffman's algorithm is used to construct a binary tree whose leaf value is assigned a code, which is optimal for the compression of the whole text need to be processed. For example, the most frequently occurring words will get the smallest code so that the final encoded text is compressed.

Initialize n one-node trees with words and the tree weights with their frequencies. Join the two binary tree with smallest weight into one and the weight of the new formed tree as the sum of weight of the two small trees. Repeat the above process N-1 times and when there is just one big tree left you are done.

Mark edges leading to left and right subtrees with 0's and 1's, respectively.

Word	Frequency
Apple	30
Banana	25
Mango	21
Orange	14
Pineapple	10



Word	Value	Code
Apple	30	11
Banana	25	10
Mango	21	01
Orange	14	001
Pineapple	10	000

It is clear that more frequency words gets smaller Huffman's code.

// Computes optimal prefix code.

```
// Input: List W of character probabilities
// Output: The Huffman tree.

Algorithm Huffman(C[0..n - 1], W[0..n - 1])
    PQ = {} // priority queue
    for i = 0 to n - 1 do
        T.char = C[i]
        T.weight = W[i]
        add T to priority queue PQ

    for i = 0 to n - 2 do
        L = remove min from PQ
        R = remove min from PQ
        T = node with children L and R
        T.weight = L.weight + R.weight
        add T to priority queue PQ
    return T
```

The Time Complexity is **O(nlogn)**.

Activity Selection Problem

Suppose that activities require exclusive use of common resources, and you want to schedule as many activities as possible.
Let S = {a1,..., an} be a set of n activities.

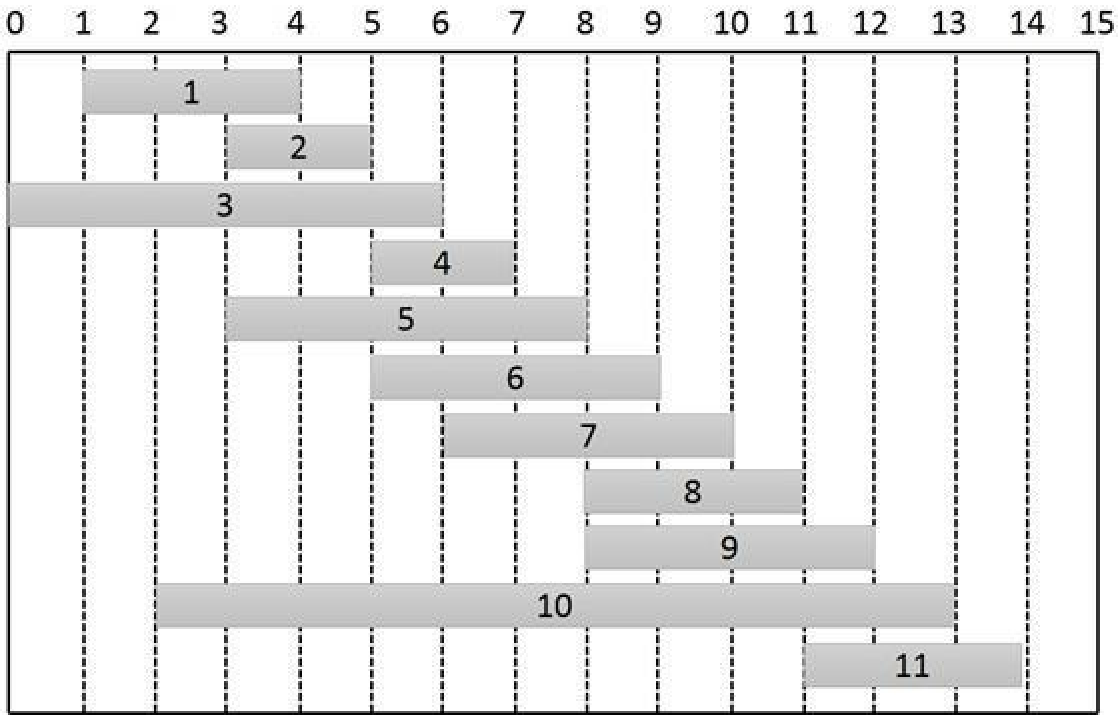
Each activity ai needs the resource during a time period starting at si and finishing before fi, i.e., during [si, fi).
The optimization problem is to select the non-overlapping largest set of activities from S.

We assume that activities S = {a1,..., an} are sorted in finish time f1 ≤ f2 ≤ ... fn-1 ≤ fn (this can be done in $\Theta(n \lg n)$).

Example: Consider these activities:

I	1	2	3	4	5	6	7	8	9	10	11
S[i]	1	3	0	5	3	5	6	8	8	2	11
F[i]	4	5	6	7	8	9	10	11	12	13	14

Here is a graphic representation:



We chose an activity that start first, and then look for the next activity that starts after it is finished. This could result in {a4, a7, a8}, but this solution is not optimal.
An optimal solution is {a1, a3, a6, a8}. (It maximizes the objective function of a number of activities scheduled.)
Another one is {a2, a5, a7, a9}. (Optimal solutions are not necessarily unique.)
How do we find (one of) these optimal solutions? Let us consider it as a dynamic programming problem.
We are trying to optimize the number of activities. Let us be greedy!

- The more time left after running an activity, the subsequent activities we can fit in.
- If we choose the first activity to finish, the more time will be left.
- Since activities are sorted by finish time, we will always start with a_1.
- Then we can solve the single sub problem of activity scheduling in this remaining time.

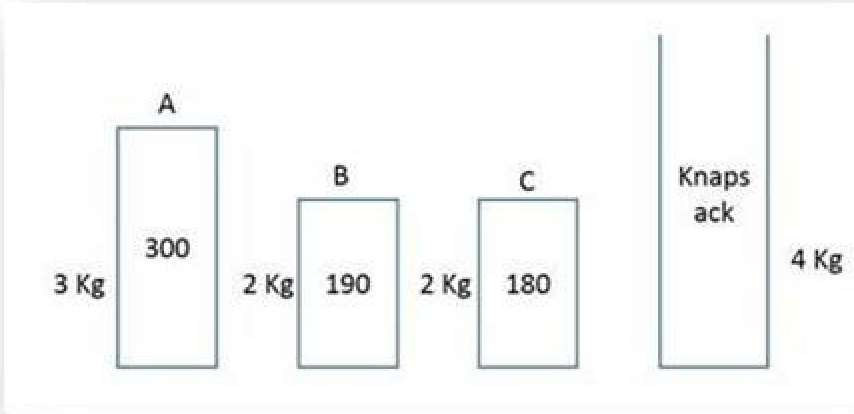
```
Algorithm ActivitySelection(S[], F[], N)
    Sort S[] and F [] in increasing order of finishing time
    A = {a1}
    K = 1
    For m = 2 to N do
        If S[m] >= F[k]
            A = A + {am}
            K = m
    Return A
```

Knapsack Problem

A thief enters a store and sees a number of items with their cost and weight mentioned. His Knapsack can hold a max weight. What should he steal to maximize profit?

Fractional Knapsack problem

A thief can take a fraction of an item (they are divisible substances, like gold powder).



The fractional knapsack problem has a greedy solution one should first sort the items in term of cost density against weight. Then fill up as much of the most valuable substance by weight as one can hold, then as much of the next most valuable substance, etc. Until W is reached.

Item	A	B	C
Cost	300	190	180
Weight	3	2	2
Cost/weight	100	95	90

For a knapsack of capacity of 4 kg.
The optimum solution of the above will take 3kg of A and 1 kg of B.

```
Algorithm FractionalKnapsack(W[], C[], Wk)
    For i = 1 to n do
        X[i] = 0
    Weight = 0
    //Use Max heap
    H = BuildMaxHeap(C/W)
    While Weight < Wk do
        i = H.GetMax()
        If(Weight + W[i] <= Wk) do
            X[i] = 1
            Weight = Weight + W[i]
        Else
            X[i] = (Wk – Weight)/W[i]
            Weight = Wk
    Return X
```

0/1 Knapsack Problem

A thief can only take or leave the item. He cannot take a fraction.
A greedy strategy same as above could result in empty space, reducing the overall cost density of the knapsack.

In the above example, after choosing object A there is no place for B or C so there leaves empty space of 1kg. Moreover, the result of the greedy solution is not optimal.
The optimal solution will be when we take object B and C. This problem can be solved by dynamic programming that we will see in the coming chapter.

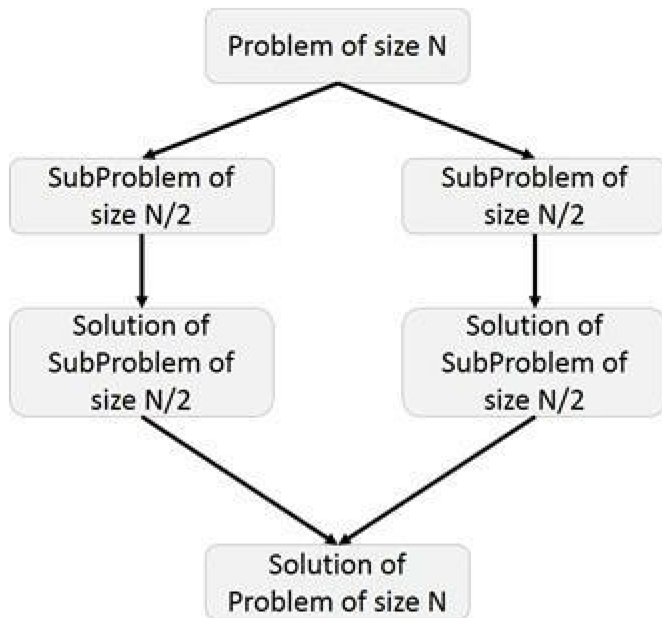
CHAPTER 18: DIVIDE-AND-CONQUER, DECREASE-AND-CONQUER

Introduction

Divide-and-Conquer algorithms works by recursively breaking down a problem into two or more sub-problems (divide), until these sub problems become simple enough so that can be solved directly (conquer). The solution of these sub problems is then combined to give a solution of the original problem.

Divide-and-Conquer algorithms involve basic three steps
1. Divide the problem into smaller problems.
2. Conquer by solving these problems.
3. Combine these results together.

In divide-and-conquer the size of the problem is reduced by a factor (half, one-third etc.), While in decrease-and-conquer the size of the problem is reduced by a constant.



Divide-and-Conquer algorithms

Examples of divide-and-conquer algorithms:
- Merge-Sort algorithm (recursion)
- Quicksort algorithm (recursion)
- Computing the length of the longest path in a binary tree (recursion)
- Computing Fibonacci numbers (recursion)
- Convex Hull

Examples of decrease-and-conquer algorithms:
- Computing POW (a, n) by calculating POW (a, n/2) using recursion
- Binary search in a sorted list (recursion)
- Searching in BST
- Insertion-Sort
- Graph traversal algorithms (DFS and BFS)
- Topological sort
- Warshall’s algorithm (recursion)
- Permutations (Minimal change approach, Johnson-Trotter algorithm)
- Fake-coin problem (Ternary search)
- Computing a median

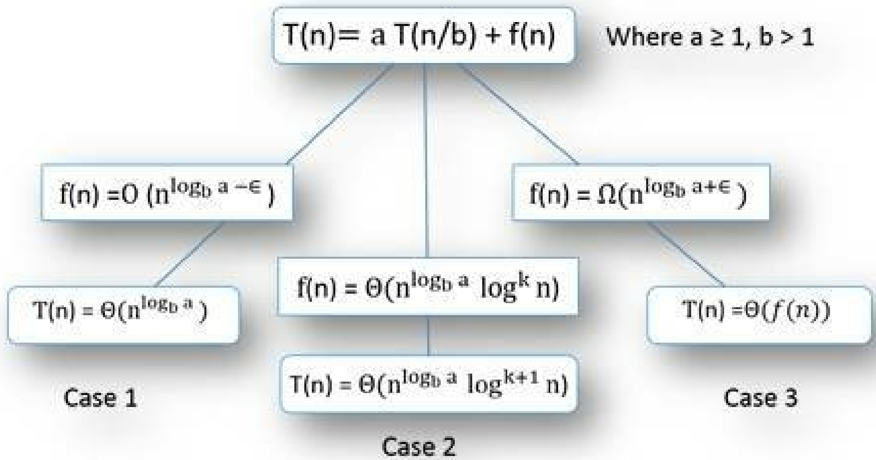
General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$

- Where $a \geq 1$ and $b > 1$.
- "n" is the size of a problem.
- "a" is a number of sub-problem in the recursion.
- “n/b” is the size of each sub-problem.
- "f(n)" is the cost of the division of the problem into sub problem or merge of the results of sub-problem to get the final result.

Master Theorem

The master theorem solves recurrence relations of the form:
T(n) = aT(n/b) + f (n)



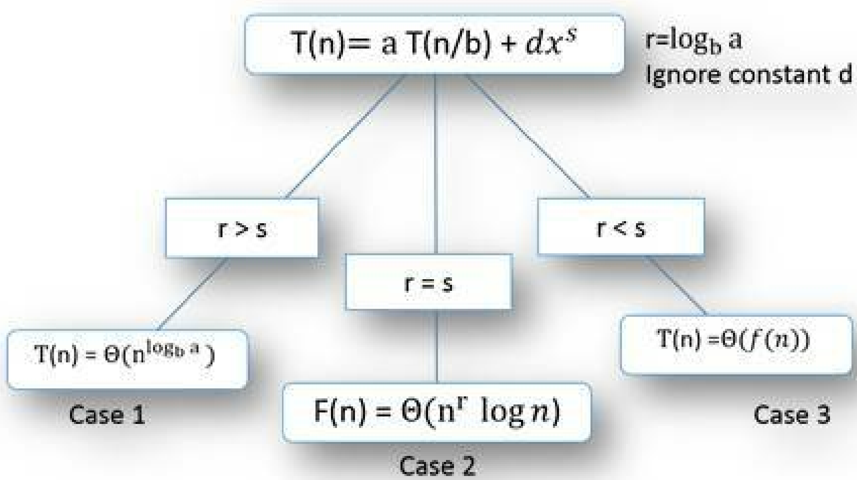
It is possible to determine an asymptotic tight bound in these three cases:

Case 1: when $f(n) = \mathbf{O}(\mathbf{n}^{\log_b a - \epsilon})$ and constant Є > 1, then the final Time Complexity will be:
$T(n) = \boldsymbol{\Theta}(\mathbf{n}^{\log_b a})$

Case 2: when $f(n) = \boldsymbol{\Theta}(\mathbf{n}^{\log_b a} \mathbf{log}^{k}\mathbf{n})$ and constant k ≥ 0, then the final Time Complexity will be:
$T(n) = \boldsymbol{\Theta}(\mathbf{n}^{\log_b a} \mathbf{log}^{k+1}\mathbf{n})$

Case 3: when $f(n) = \boldsymbol{\Omega}(\mathbf{n}^{\log_b a + \epsilon})$ and constant Є > 1, Then the final Time Complexity will be:
T(n) = Θ(f(n))

Modified Master theorem: This is a shortcut to solving the same problem easily and fast. If the recurrence relation is in the form of $T(n) = a\,T(n/b) + \mathbf{dx^s}$



Example 1: Take an example of Merge-Sort, T(n) = 2T(n/2) + n
Sol:-
$\text{Log}_b a = \log_2 2 = 1$

$f(n) = n = \Theta(n^{\log_2 2} \log^0 n)$

Case 2 applies and $T(n) = \Theta(n^{\log_2 2} \log^{0+1} n)$
$T(n) = \Theta(n \log(n))$

Example 2: Binary Search $T(n) = T(n/2) + O(1)$
Sol:-
$\text{Log}_b a = \log_2 1 = 0$

$f(n) = 1 = \Theta(n^{\log_2 1} \log^0 n)$

Case 2 applies and $T(n) = \Theta(n^{\log_2 1} \log^{0+1} n)$
$T(n) = \Theta(\log(n))$

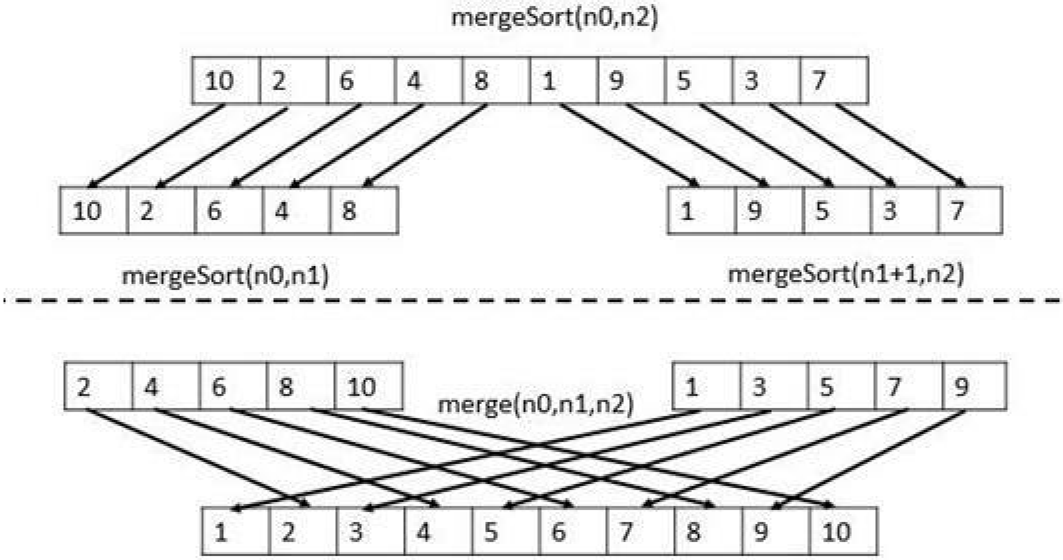
Example 3: Binary tree traversal $T(n) = 2T(n/2) + O(1)$
Sol:-
$\text{Log}_b a = \log_2 2 = 1$

$f(n) = 1 = O(n^{\log_2 2 - 1})$

Case 1 applies and $T(n) = \Theta(n^{\log_2 2})$
$T(n) = \Theta(n)$

Problems on Divide-and-Conquer Algorithm

Merge-Sort algorithm




```
// Sorts a given list by mergesort
// Input: A list A of orderable elements
// Output: List A[0..n − 1] in ascending order

Algorithm Mergesort(A[0..n − 1])
    if n ≤ 1 then
        return;
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B)
    Mergesort(C)
    Merge(B, C, A)
```

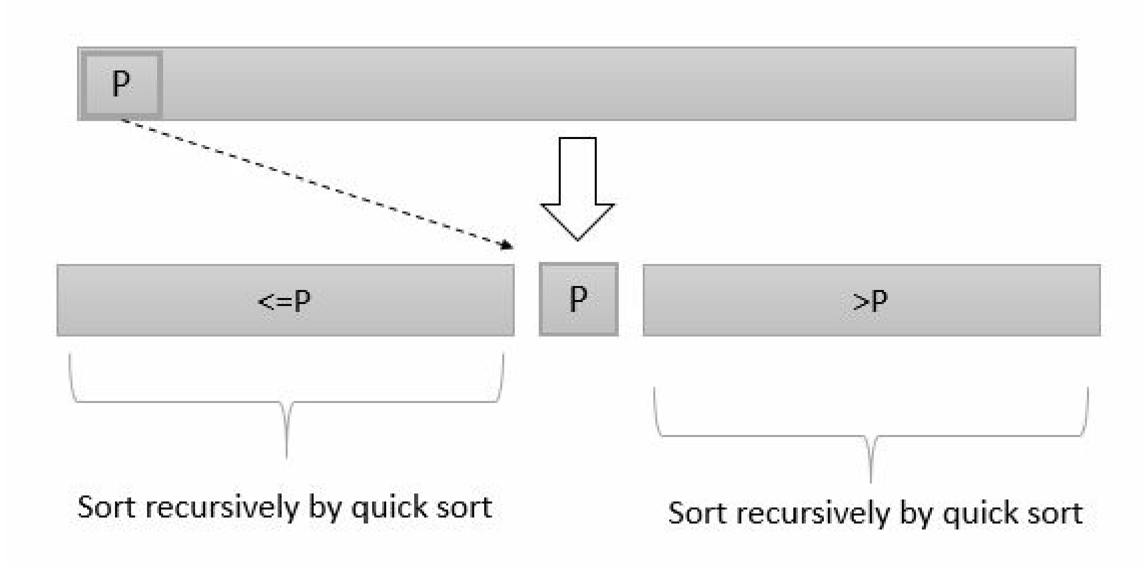
```
// Merges two sorted arrays into one list
// Input: Sorted arrays B and C
// Output: Sorted list A

Algorithm Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
    i = 0
    j = 0
    for k = 0 to p + q − 1 do
        if i < p and (j = q or B[i] ≤ C[j]) then
            A[k] = B[i]
            i = i + 1
        else
            A[k] = C[j]
            j = j + 1
```

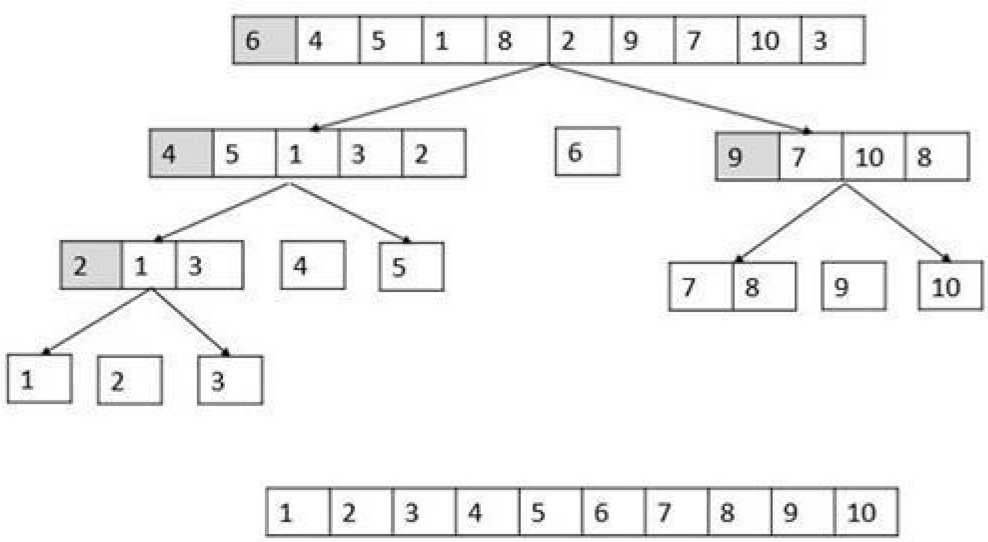
Time Complexity: **O(nlogn)** & Space Complexity: **O(n)**

The Time Complexity of Merge-Sort is **O(nlogn)** in all 3 cases (worst, average and best) as Merge-Sort always divides the list into two halves and take linear time to merge two halves.

It requires the equal amount of additional space as the unsorted list. Hence, it is not at all recommended for searching large unsorted lists.

Quick-Sort





```
// Sorts a subarray by quicksort
// Input: An subarray of A
// Output: List A[l..r] in ascending order

Algorithm Quicksort(A[l..r])
    if l < r then
        p ← Partition(A[l..r]) // p is index of pivot
        Quicksort(A[l..p − 1])
        Quicksort(A[p + 1..r])
```

```
// Partitions a subarray using A[..] as pivot
// Input: Subarray of A
// Output: Final position of pivot
```

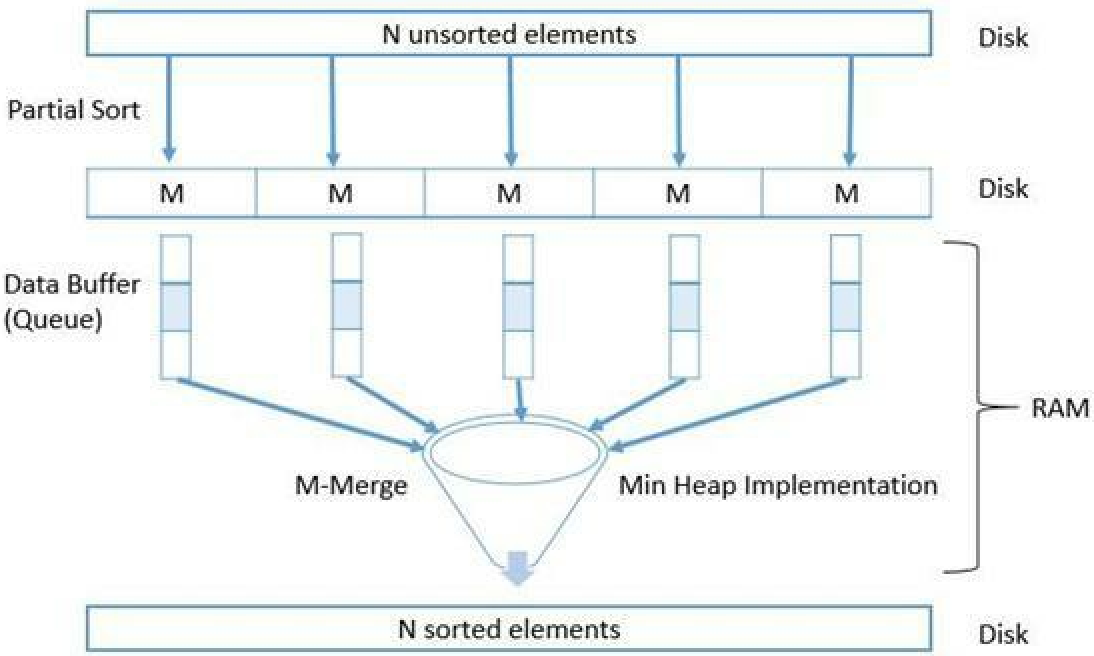
```
Algorithm Partition(A[], left, right)
    pivot = A[left]
    lower = left
    upper= right
    while lower < upper
        while A[lower] <= pivot
            lower = lower + 1
        while A[upper] > pivot
            upper = upper – 1
        if lower < upper then
            swap A[lower] and A[upper]
    swap A[lower] and A[upper] //upper is the pivot position
    return upper
```

Worst Case Time Complexity: **O(n^2),**
Average Time Complexity: **O(nlogn),**
Space Complexity: **O(nlogn),** The space required by Quick-Sort is very less, only **O(nlogn)** additional space is required.

Quicksort is not a stable sorting technique, so it might change the occurrence of two similar elements in the list while sorting.

External Sorting

External sorting is also done using divide and conquer algorithm.



Binary Search

We get the middle point from the sorted list and start comparing with the desired value.
Note: Binary search requires the list to be sorted otherwise binary search cannot be applied.

```
// Searches a value in a  sorted list using binary search
// Input: An sorted list A and a key K
```

```
// Output: The index of K or −1

Algorithm BinarySearch(A[0..N − 1], N, K) // iterative solution
    low = 0
    high = N-1
    while low <= high do
        mid = ⌊ (low + high)/2 ⌋
        if K = A[mid] then
            return mid
        else if A[mid] < K
            low = mid + 1
        else
            high = mid - 1
    return −1
```

```
// Searches a value in a  sorted list using binary search
// Input: An sorted list A and a key K
// Output: The index of K or −1

Algorithm BinarySearch(A[], low, high, K) //Recursive solution
    If low > high
        return -1
    mid = ⌊ (low + high)/2 ⌋
    if K = A[mid] then
        return mid
    else if A[mid] < K
        return BinarySearch(A[],mid + 1, high, K)
    else
        return BinarySearch(A[],low, mid - 1, K)
```

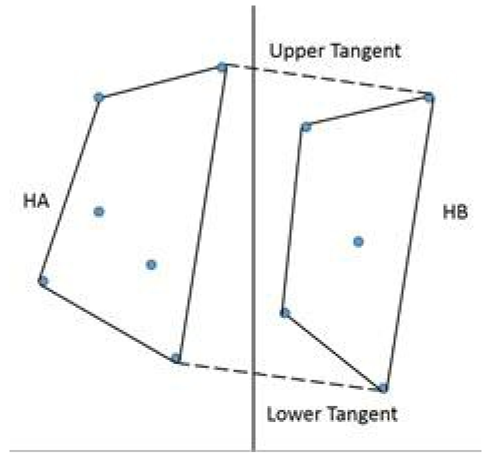
Time Complexity: **O(logn)**. If you notice the above programs, you see that we always take half input and throwing out the other half. So the recurrence relation for binary search is T (n) = T (n/2) + c. Using a divide and conquer master theorem, we get T (n) = **O(logn)**.
Space Complexity: **O(1)**

Power function

```
// Compute Nth power of X using divide and conquer using recursion
// Input: Value X and power N
// Output: Power( X, N)
Algorithm Power( X, N)
    If N = 0
        Return 1
    Else if N % 2 == 0
        Value = Power(X, N/2)
        Return Value * Value
    Else
        Value = Power(X, N/2)
        Return Value * Value * X
```

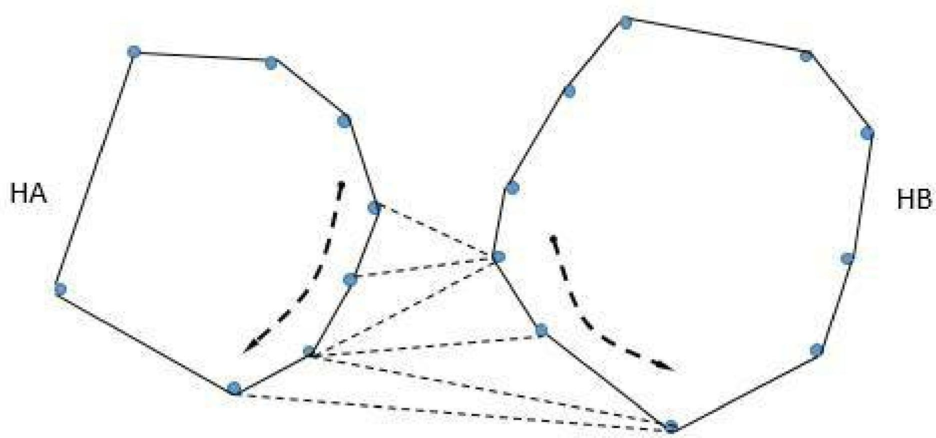
Convex Hull

Sort points by X-coordinates. Divide points into equal halves A and B. Recursively compute HA and HB. Merge HA and HB to obtain CH




```
LowerTangent(HA, HB)
    A = rightmost point of HA
    B = leftmost point of HB
    While ab is not a lower tangent for HA and HB do
        While ab is not a lower tangent to HA do
            a = a − 1 (move a clockwise)
        While ab is not a lower tangent to HB do
            b = b + 1 (move b counterclockwise)
    Return ab
```

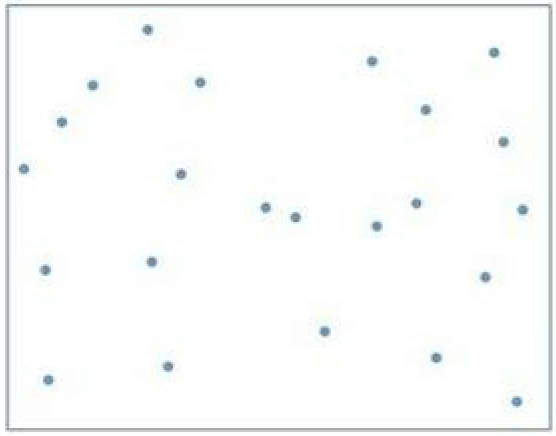
Similarly find upper tangent and combine the two hulls.



Initial sorting takes **O(nlogn)** time
Recurrence relation T (N) = 2T (N/2) + **O(N)**
Where, **O(N)** time for tangent computation inside merging
Final Time Complexity will be T (N) = **O(nlogn)**.

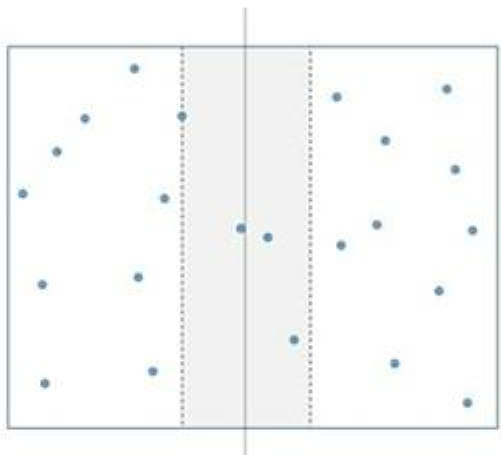
Closest Pair

Given N points in 2-dimensional plane, find two points whose mutual distance is smallest.



A brute force algorithm takes every point and find its distance with all the other points in the plane. In addition, keep track of the minimum distance points and minimum distance. The closest pair will be found in $O(n^2)$ time.

Let us suppose there is a vertical line, which divide the graph into two separate parts (let us call it left and right part). The brute force algorithm, we will notice that we are comparing all the points in the left half with the points in the right half. This is the point where we are doing some extra work.



To find the minimum we need to consider only three cases:

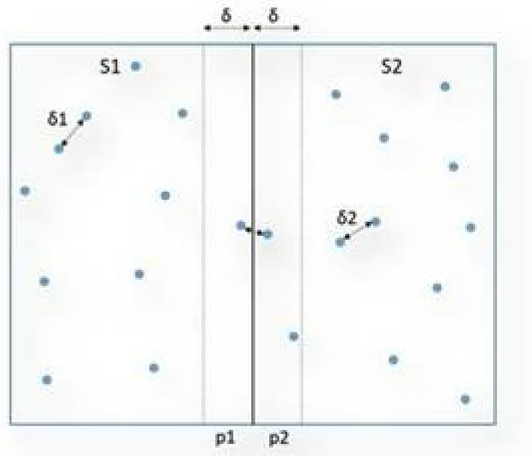
1) Closest pair in the right half
2) Closest pair in the left half.
3) Closest pair in the boundary region of the two halves. (Gray)

Every time we will divide the space S into two parts S1 and S2 by a vertical line. Recursively we will compute the closest pair in both S1 and S2. Let us call minimum distance in space S1 as δ1 and minimum distance in space S2 as δ2.

We will find δ = min (δ1, δ2)

Now we will find the closest pair in the boundary region. By taking one point each from S1 and S2 in the boundary range of δ width on both sides.

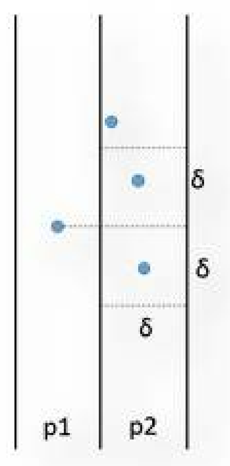
The candidate pair of point (p, q) where p Є S1 and q Є S2.



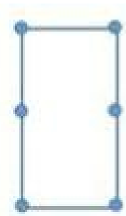
We can find the points that lie in this region in linear time O(N) by just scanning through all the points and finding that all points lie in this region.

Now we can sort them in increasing order in Y-axis in just **O(nlogn)** time. Then scan through them and get the minimum in just one linear pass. Closest pair cannot be far apart from each other.

Let us look into the next figure.



Then the question is how many points we need to compare. We need to compare the points sorted in Y-axis only in the range of δ. Therefore, the number of points will come down to only 6 points.



By doing this, we are getting equation.
$T(N) = 2T(N/2) + N + N\log N + 6N = O(n(\log n)^2)$

Can we optimize this further?
Yes

Initially, when we are sorting the points in X coordinate we are sorting them in Y coordinate too.
When we divide the problem, then we traverse through the Y coordinate list too, and construct the corresponding Y coordinate list for both S1 and S2. And pass that list to them.

Since we have the Y coordinate list passed to a function the δ region points can be found sorted in the Y coordinates in just one single pass in just **O(N)** time.

$T(N) = 2T(N/2) + N + N + 6N =$ **O(nlogn)**

```
// Finds closest pair of points
// Input: A set of n points sorted by coordinates
// Output: Distance between closest pair

Algorithm ClosestPair(P)
    if n < 2 then
        return ∞
    else if n = 2 then
        return distance between pair
    else
        m = median value for x coordinate
        δ1 = ClosestPair(points with x < m)
        δ2 = ClosestPair(points with x > m)
        δ = min(δ1, δ2)
        δ3 = process points with m −δ < x < m + δ
        return min(δ, δ3)
```

First pre-process the points by sorting them in X and Y coordinates. Use two separate lists to keep these sorted points.

Before recursively solving sub-problem pass the sorted list for that sub-problem.

CHAPTER 19: DYNAMIC PROGRAMMING

Introduction

While solving problems using Divide-and-Conquer method, there may be a case when recursively sub-problems can result in the same computation being performed multiple times. This problem arises when there are identical sub-problems arise repeatedly in a recursion.

Dynamic programming is used to avoid the requirement of repeated calculation of same sub-problem. In this method, we usually store the result of sub - problems in some data structure (like a table) and refer it to find if we have already calculated the solution of sub - problems before calculating it again.

Dynamic programming is applied to solve problems with the following properties:
1. Optimal Substructure: An optimal solution constructed from the optimal solutions of its sub-problems.
2. Overlapping Sub problems: While calculating the optimal solution of sub problems same computation is repeated repeatedly.

Examples:

- Fibonacci numbers computed by iteration.
- Assembly-line Scheduling
- Matrix-chain Multiplication
- 0/1 Knapsack Problem
- Longest Common Subsequence
- Optimal Binary Tree
- Warshall's algorithm for transitive closure implemented by iterations
- Floyd's algorithms for all-pairs shortest paths
- Optimal Polygon Triangulation
- Floyd-Warshall's Algorithm

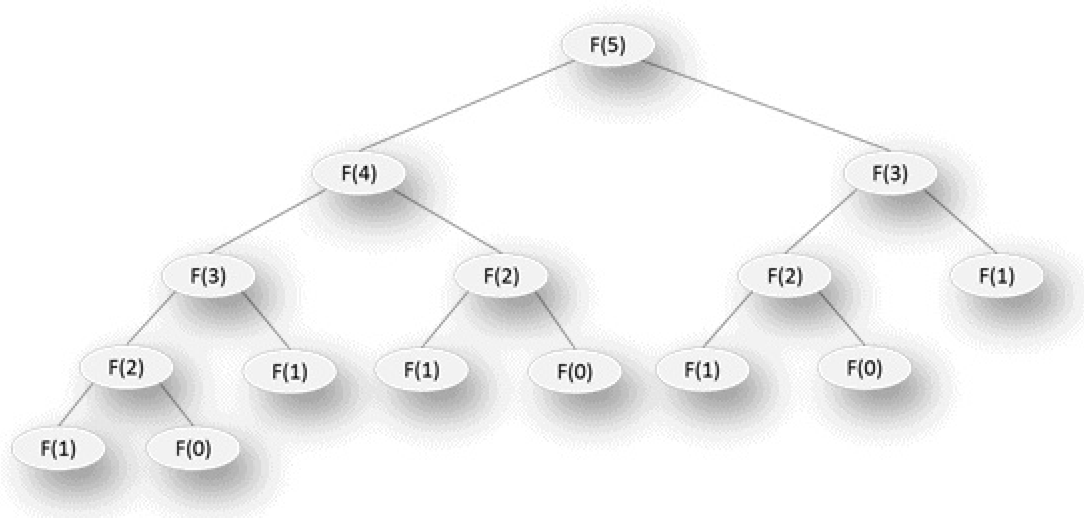
Steps for solving / recognizing if DP applies.

1) Optimal Substructure: Try to find if there is a recursive relation between problem and sub-problem.
2) Write recursive relation of the problem. (Observe Overlapping Sub problems at this step.)
3) Compute the value of sub problems in a bottom up fashion and store this value in some table.
4) Construct the optimal solution from the value stored in step 3.
5) Repeat step 3 and 4 until you get your solution.

Problems on Dynamic programming Algorithm

Fibonacci numbers

```
func fibonacci(n int) int {
	if n <= 1 {
		return n
	}
	return fibonacci(n-1) + fibonacci(n-2)
}
```



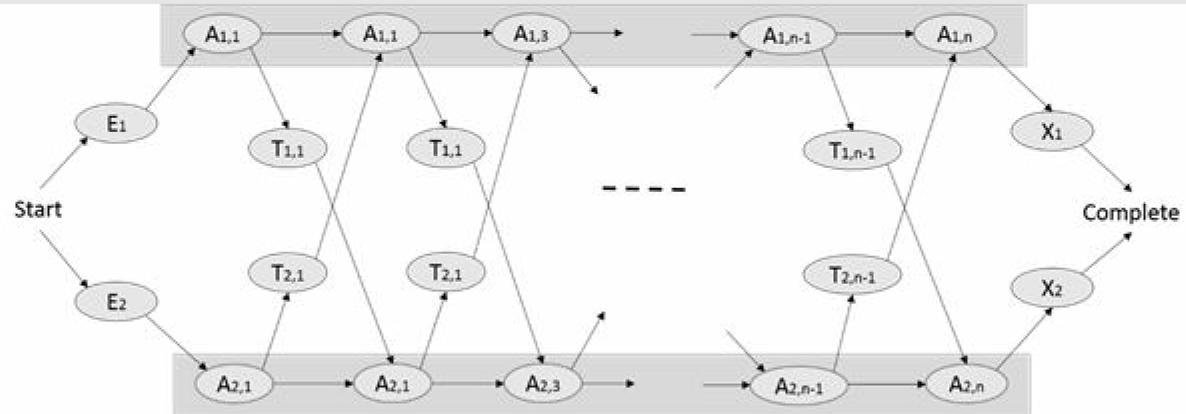
Using divide and conquer same sub-problem is solved again and again, which reduce the performance of the algorithm. This algorithm has an exponential Time Complexity.

Same problem of Fibonacci can be solved in linear time if we sort the results of sub problems.

```
func fibonacci2(n int) int {
	first := 0
	second := 1
	temp := 0

	if n == 0 {
		return first
	} else if n == 1 {
		return second
	}
	for i := 2; i <= n; i++ {
		temp = first + second
		first = second
		second = temp
	}
	return temp
}
```

Using this algorithm, we will get Fibonacci in linear Time Complexity and constant Space Complexity.

Assembly-line Scheduling



We consider the problem of calculating the least amount of time necessary to build a car when using a manufacturing chain with two assembling lines, as shown in the figure

The problem variables:

- e[i]: entry time in assembly line i
- x[i]: exit time from assembly line i
- a[i,j]: Time required at station S[i,j] (assembly line i, stage j)
- t[i,j]: Time required to transit from station S[i,j] to the other assembly line

Your program must calculate:

- The least amount of time needed to build a car
- The list of stations to traverse in order to assemble a car as fast as possible.

The manufacturing chain will have no more than 50 stations.

If we want to solve this problem in the brute force approach, there will be in total 2^n Different combinations so the Time Complexity will be O(2^n)

Step 1: Characterizing the structure of the optimal solution

To calculate the fastest assembly time, we only need to know the fastest time to $S1_n$ and the fastest time to $S2_n$, including the assembly time for the nth part. Then we choose between the two exiting points by taking into consideration the extra time required, x1 and x2. To compute the fastest time to $S1_n$ we only need to know the fastest time to $S1_{n-1}$ and to $S2_{n-1}$. Then there are only two choices.

Step 2: A recursive definition of the values to be computed

$$f1[j] = \begin{cases} e1 + a1,1 & \text{if } j = 1 \\ \min(f1[j-1] + a1,j, \quad f2[j-1] + t2,j-1 + a1,j) & \text{if } j \geq 2 \end{cases}$$

$$f2[j] = \begin{cases} e2 + a2,1 & \text{if } j = 1 \\ \min(f2[j-1] + a2,j, \quad f1[j-1] + t1,j-1 + a2,j) & \text{if } j \geq 2 \end{cases}$$

Step 3: Computing the fastest time finally, compute f* as

Step 4: Computing the fastest path compute as li[j] as the choice made for fi[j] (whether the first or the second term gives the minimum). Also, compute the choice for f* as l*.

```
FASTEST-WAY(a, t, e, x, n)
    f1[1] ← e1 + a1,1
    f2[1] ←e2 + a2,1
    for j ← 2 to n
        do if f1[j - 1] + a1,j ≤ f2[j - 1] + t2,j-1 + a1,j
            then f1[j] ← f1[j - 1] + a1, j
```

```
            l1[j] ← 1
        else f1[j] ← f2[j - 1] + t2,j-1 + a1,j
            l1[j] ← 2
        if f2[j - 1] + a2,j ≤ f1[j - 1] + t1,j-1 + a2,j
            then f2[j] ← f2[j - 1] + a2,j
            l2[j] ← 2
        else f2[j] ∞ f1[j - 1] + t1,j-1 + a2,j
            l2[j] ← 1
    if f1[n] + x1 ≤ f2[n] + x2
        then f* = f1[n] + x1
        l* = 1
    else f* = f2[n] + x2
        l* = 2
```

Matrix chain multiplication

Same problem is also known as Matrix Chain Ordering Problem or Optimal-parenthesization of matrix problem.

Given a sequence of matrices, M = M1,…, Mn. The goal of this problem is to find the most efficient way to multiply these matrices. The guild is not to perform the actual multiplication, but to decide the sequence of the matrix multiplications, so that the result will be calculated in minimal operations.

To compute the product of two matrices of dimensions pXq and qXr, pqr number of operations will be required. Matrix multiplication operations are associative in nature. Therefore, matrix multiplication can be done in many ways.
For example, M1, M2, M3 and M4, can be fully parenthesized as:
(M1· (M2· (M3·M4)))
(M1· ((M2·M3)· M4))
((M1·M2)· (M3·M4))
(((M1·M2)· M3)· M4)
((M1· (M2·M3))· M4)

For example,
Let M1 dimensions are 10 × 100, M2 dimensions are 100 × 10, and M3 dimensions are 10 × 50.
((M1·M2)· M3) = (10*100*10) + (10*10*50) = 15000
(M1· (M2·M3) = (100*10*50) + (10*100*50) = 100000
Therefore, in this problem we need to parenthesize the matrix chain so that total multiplication cost is minimized.

Given a sequence of n matrices M1, M2,… Mn. And their dimensions are p0, p1, p2,…, pn.
Where matrix Ai has dimension pi − 1 × pi for 1 ≤ i ≤ n. Determine the order of multiplication that minimizes the total number of multiplications.

If you try to solve this problem using the brute - force method, then you will find all possible parenthesization. Then will compute the cost of multiplication. Then will pick the best solution. This approach will be exponential in nature.
There is an insufficiency in the brute force approach. Take an example of M1, M2,…, Mn. When you have calculated that ((M1·M2) · M3) is better than (M1· (M2·M3) so there is no point of calculating then combinations of (M1· (M2·M3) with (M4, M5…. Mn).

Optimal substructure:
Assume that M (1, N) is the optimum cost of production of the M1,…, Mn.

A list p [] to record the dimensions of the matrices.
P [0] = row of the M1

p[i] = col of Mi   1<=i<=N

For some k
M(1,N) = M(1,K) + M(K+1,N) + p0*pk*pn

If M (1, N) is minimal then both M (1, K) & M (K+1, N) are minimal.

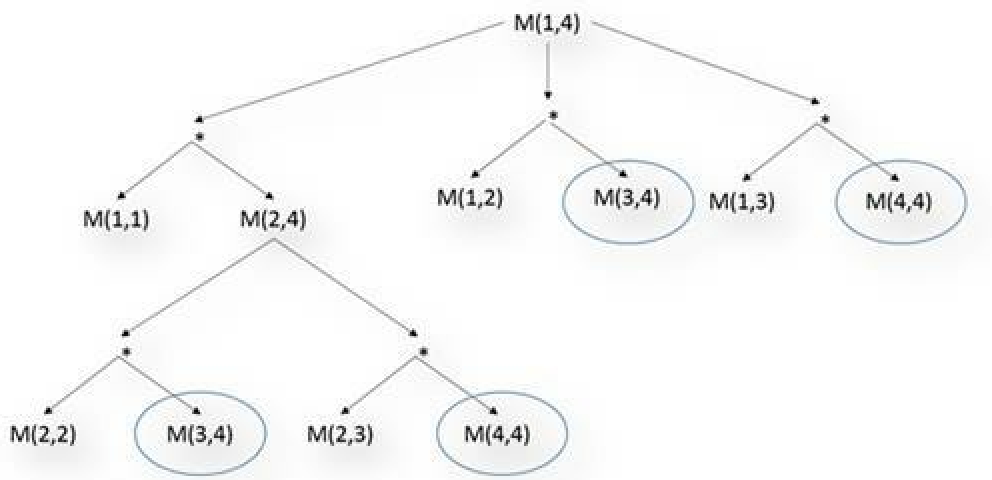
Otherwise, if there is some M'(1, K) is there whose cost is less than M (1.. K), then M (1.. N) can't be minimal and there is a more optimal solution possible.

For some general i and j.
M(i,j) = M(i,K) + M(K+1,j) + pi-1*pk*pj

Recurrence relation:

M(i,j) = $\begin{cases} 0 & \text{if } i = j \\ \min\{M(i,k) + M(k,j) + pi - 1 * pk * pj\} & i \le k < j \end{cases}$

Overlapping Sub problems:



Directly calling recursive function will lead to calculation of same sub-problem multiple times. This will lead to exponential solution.

```
Algorithm MatrixChainMultiplication(p[])
    for i := 1 to n
        M[i, i] := 0;
        for l = 2 to n // l is the moving line
            for i = 1 to n − l +1
                j = i + l − 1;
                M[i, j] =  min{M(i,k) + M(k,j) + pi − 1 * pk * pj}
i ≤ k < j
```

Time Complexity will O(n^3)
Constructing optimal parenthesis Solution

Use another table s[1..n, 1..n]. Each entry s[i, j] records the value of k such that the optimal parenthesization of Mi Mi+1...Mj splits the product between Mk and Mk+1.

```
Algorithm MatrixChainMultiplication(p[])
```

```
    for i := 1 to n
        M[i, i] := 0;
    for l = 2 to n // l is the moving line
        for i = 1 to n − l +1
            j = i + l − 1;
            M[i, j] = min {M(i,k) + M(k,j) + pi − 1 * pk * pj }
i ≤ k < j
            S[i, j] = k for min {M(i,k) + M(k,j) + pi − 1 * pk * pj }
i ≤ k < j
Algorithm MatrixChainMultiplication(p[])
    for i := 1 to n
    M[i, i] := 0;
    for l = 2 to n // l is the moving line
        for i = 1 to n − l +1
            j = i + l − 1;
            for k = i to j
                if( (M(i,k) + M(k,j) + pi − 1 * pk * pj ) <  M[i,j]
                    M[i, j] =  (M(i,k) + M(k,j) + pi − 1 * pk * pj )
                    S[i, j] =  k
```

```
Algorithm PrintOptimalParenthesis(s[], i, j)
    If i = j
        Print Ai
    Else
        Print "("
        PrintOptimalParenthesis(s[], i,s[i, j])
        PrintOptimalParenthesis(s[], s[i, j],j)
        Print ")"
```

Longest Common Subsequence

Let X = {x1, x2,…., xm} is a sequence of characters and Y = {y1, y2,…, yn} is another sequence.
Z is a subsequence of X if it can be driven by deleting some elements of X. Z is a subsequence of Y if it can be driven by deleting some elements form Y. Z is LCS of it is subsequence to both X and Y, and length of all the subsequence is less than Z.

Optimal Substructure:

Let X = < x1, x2, ..., xm > and Y = < y1, y2, ..., yn > be two sequences, and let Z = < z1, z2, ..., zk > be a LCS of X and Y.

- If xm = yn, then zk = xm = yn ⇒ Zk−1 is a LCS of Xm−1 and Yn−1
- If xm != yn, then:
    - zk != xm ⇒ Z is an LCS of Xm−1 and Y.
    - zk != yn ⇒ Z is an LCS of X and Yn−1.

Recurrence relation
Let c[i, j] be the length of the longest common subsequence between X = {x1, x2,…., xi} and Y = {y1, y2,…, yj}.
Then c[n, m] contains the length of an LCS of X and Y

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } xi = yj \\ \max(c[i-1, j], c[i, j-1] & \text{otherwise} \end{cases}$$

```
Algorithm LCS(X[], m, Y[], n)
    for i = 1 to m
        c[i,0] = 0
    for j = 1 to n
        c[0,j] = 0;
    for i = 1 to m
        for j = 1 to n
            if X[i] == Y[j]
                c[i,j] = c[i-1,j-1] + 1
                b[i,j] = ↖
            else
                if c[i-1,j] ≥ c[i,j-1]
                    c[i,j] = c[i-1,j]
                    b[i,j] = ↑
                else
                    c[i,j] = c[i,j-1]
                    b[i,j] = ←
```

```
Algorithm PrintLCS(b[],X[], i, j)
    if i = 0
        return
    if j = 0
        return
    if b[i, j] = ↖
        PrintLCS (b[],X[], i − 1, j − 1)
        print X[i]
    else if b[i, j] = ↑
        PrintLCS (b[],X[], i − 1, j)
    else
        PrintLCS (b[],X[], i, j − 1)
```

Coin Exchanging problem

How can a given amount of money N be made with the least number of coins of given denominations D= {d1… dn}?

For example, Indian coin system {5, 10, 20, 25, 50,100}. Suppose we want to give change of a certain amount of 40 paisa.

We can make a solution by repeatedly choosing a coin ≤ to the current amount, resulting in a new amount. The greedy solution always choose the largest coin value possible.
For 40 paisa: {25, 10, and 5}

This is how billions of people around the globe do change every day. That is an approximate solution of the problem. But this is not the optimal way, the optimal solution for the above problem is {20, 20}

Step (I): Characterize the structure of a coin-change solution.
Define C [j] to be the minimum number of coins we need to make a change for j cents.

If we knew that an optimal solution for the problem of making change for j cents used a coin of denomination di, we would have:
C[j] = 1+C[j − di]

Strep (II): Recursively defines the value of an optimal solution.

$$c[j] = \begin{cases} \text{infinite} & \text{if } j < 0 \\ 0 & \text{if } j = 0 \\ 1 + \min(c[j - di]) \ \ 1 \le i \le k & \text{if } j \ge 1 \end{cases}$$

Step (III): Compute values in a bottom-up fashion.

```
Algorithm CoinExchange(n, d[], k)
    C[0] = 0
    for j = 1 to n do
        C[j] = infinite
    for i = 1 to k do
        if j < di and 1+C[j − di] < C[j] then
            C[j] = 1+C[j − di]
    return C
```

Complexity: O(nk)

Step (iv): Construct an optimal solution
We use an additional list Deno[1.. n], where Deno[j] is the denomination of a coin used in an optimal solution.

```
Algorithm CoinExchange(n, d[], k)
    C[0] = 0
    for j = 1 to n do
        C[j] = infinite
    for i = 1 to k do
        if j < di and 1+C[j − di] < C[j] then
            C[j] = 1+C[j − di]
            Deno[j] = di
    return C
```

```
Algorithm PrintCoins( Deno[], j)
    if j > 0
        PrintCoins (Deno, j −Deno[j])
    print Deno[j]
```

CHAPTER 20: BACKTRACKING

Introduction

Suppose there is a lock, which produce some “click” sound when correct digit is selected for any level. To open it you just need to find the first digit, then find the second digit, then find the third digit and done. This will be a greedy algorithm and you will find the solution very quickly.

However, let us suppose the lock is some old one and it creates same sound not only at the correct digit, but also at some other digits. Therefore, when you are trying to find the digit of the first ring, then it may product sound at multiple instances. So at this point you are not directly going straight to the solution, but you need to test various states and in case those states are not the solution you are looking for, then you need to backtrack one step at a time and find the next solution. Sure, this intelligence/ heuristics of click sound will help you to reach your goal much faster. These functions are called Pruning function or bounding functions.

Problems on Backtracking Algorithm

N Queens Problem

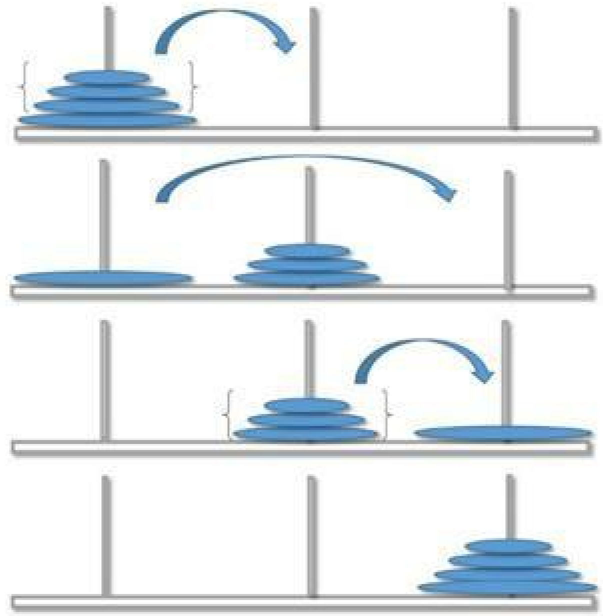
There are N queens given, you need to arrange them in a chessboard on NxN such that no queen should attach each other.

```
func NQueens(n int) {
    Q := make([]int, n)
    NQueensUtil(Q, 0, 8)
}

func NQueensUtil(Q []int, k int, n int) {
    if k == n {
        Print(Q, n)
        return
    }
    for i := 0; i < n; i++ {
        Q[k] = i
        if Feasible(Q, k) {
            NQueensUtil(Q, k+1, n)
        }
    }
}

func Absolute(n int) int {
    if n < 0 {
        return -1 * n
    }
    return n
}

func Feasible(Q []int, k int) bool {
    for i := 0; i < k; i++ {
        if Q[k] == Q[i] || Absolute(Q[i]-Q[k]) == Absolute(i-k) {
            return false
        }
    }
    return true
}

func Print(Q []int, n int) {
    for i := 0; i < n; i++ {
        fmt.Print(" ", Q[i])
    }
    fmt.Println()
}

func main() {
    NQueens(8)
}
```

Tower of Hanoi

The Tower of Hanoi puzzle, disks need to be moved from one pillar to another such that any large disk cannot rest above any small disk.

This is a famous puzzle in the programming world; its origins can be tracked back to India.
"There is a story about an Indian temple in Kashi Viswanathan which contains a large room with three timeworn posts in it surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient Hindu prophecy, have been moving these disks, in accordance with the immutable rules of the Brahma the creator of universe, since the beginning of time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the prophecy, when the last move of the puzzle will be completed, the world will end."   ;) ;) ;)



```
func TowersOfHanoi(num int) {
    fmt.Println("The sequence of moves involved in the Tower of Hanoi are :")
    TOHUtil(num, "A", "C", "B")
}

func TOHUtil(num int, from string, to string, temp string) {
    if num < 1 {
        return
    }

    TOHUtil(num-1, from, temp, to)
    fmt.Println("Move disk ", num, " from peg ", from, " to peg ", to)
    TOHUtil(num-1, temp, to, from)
}

func main() {
    TowersOfHanoi(3)
}
```

CHAPTER 21: COMPLEXITY THEORY AND NP COMPLETENESS

Introduction

Computational complexity is the measurement of how much resources are required to solve some problem.

There are two types of resources:
1. Time: how many steps are takes to solve a problem
2. Space: how much memory is takes to solve a problem.

Decision problem

Much of Complexity theory deals with decision problems. A decision problem always has a yes or no answer.

Many problems can be converted to a decision problem that have answered as yes or no. For example:

1. Searching: The problem of searching element can be a decision problem if we ask, “Find if a particular number is there in the list ?”.

2. Sorting of list and find if the list is sorted you can make a decision problem, “Is the list is sorted in increasing order?”.

3. Graph coloring algorithms: this is also can be converted to a decision problem. Can we do the graph coloring by using X number of colors?

4. Hamiltonian cycle: Is there is a path from all the nodes, each node is visited exactly once and come back to the starting node without breaking?

Complexity Classes

Problems are divided into many classes such that how difficult to solve them or how difficult to find if the given solution is correct or not.

Class P problems

The class P consists of a set of problems that can be solved in polynomial time. The complexity of a P problem is $O(n^k)$ where n is input size and k is some constant (it cannot depend on n).

Class P Definition: The class P contains all decision problems for which a Turing machine algorithm leads to the "yes/no" answer in a definite number of steps bounded by a polynomial function.
For example:
Given a sequence a1, a2, a3…. an. Find if a number X is in this list.
We can search, the number X in this list in linear time (polynomial time)

Another example:
Given a sequence a1, a2, a3…. an. If we are asked to sort the sequence.
We can sort and array in polynomial time using Bubble-Sort, this is also linear time.

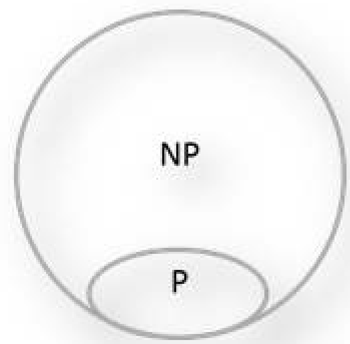
Note: **O(logn)** is also polynomial. Any algorithm, which has complexity less than some O (nk), is also polynomial.

Some problem of P class is:
1. Shortest path
2. Minimum spanning tree
3. Maximum problem.
4. Max flow graph problem.
5. Convex hull

Class NP problems

Set of problems for which there is a polynomial time checking algorithm. Given a solution if we can check in a polynomial time if that solution is correct or not then, the problem is NP problem.

Class NP Definition: The class NP contains all decision problems for which, given a solution, there exists a polynomial time “proof” or “certificate” that can verify if the solution is the right “yes/no” answer



Note: There is no guarantee that you will be able to solve this problem in polynomial time. However, if a problem is an NP problem, then you can verify an answer in polynomial time.

NP does not means “non-polynomial”. Actually, it is “Non-Deterministic Polynomial” type of problem. They are the kind of problems that can be solved in polynomial time by a Non-Deterministic Turing machine. At each point, all the possibilities are executed in parallel. If there are n possible choices, then all n cases will be executed in parallel. We do not have non-deterministic computers. Do not confuse it with parallel computing because the number of CPU is limited in parallel computing it may be 16 core or 32 core, but it cannot be N-Core.

In short NP problems are those problems for which, if a solution is given. We can verify that solution (if it is correct or not) in polynomial time.

Boolean Satisfiability problem

A Boolean formula is satisfied if there exist some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1.

$$(\mathbf{A1} \vee \mathbf{A2} \ldots) \wedge (\mathbf{A2} \vee \mathbf{A4}..) \ldots \wedge (.. \vee \mathbf{AN})$$

There are in total N Different Boolean Variables A1, A2… AN. There are an M number of brackets. Each bracket has K variables.

There is N variable so the number of solutions will be 2^n

And to verify if the solutions really evaluate the equation to 1 will take total (2^n * km) steps

Given solution of this problem you can find if the formula satisfies or not in KM steps.

Hamiltonian cycle

Hamiltonian cycle is a path from all the nodes of a graph, each node is visited exactly once and come back to the starting node without breaking.

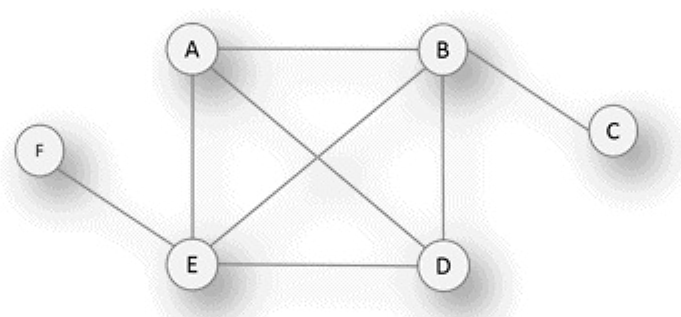
Is an NP problem, if you have a solution to it, then you just need to see if all the nodes are there in the path and you came back to where you started and you are done? The checking is done in linear time and you are done.

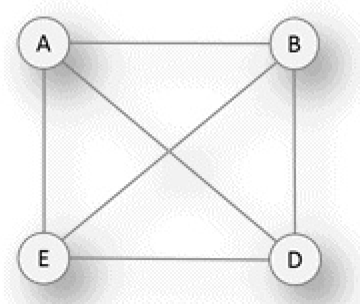
Determining whether a directed graph has a Hamiltonian cycle does not have a polynomial time algorithm. O(n!)

However, if someone have given you a sequence of vertices, determining whether that sequence forms a Hamiltonian cycle can be done in polynomial time (Linear time).
Hamiltonian cycles are in NP

Clique Problem

In a graph given is there is a clique of size K or more. A clique is a subset of nodes, which are fully connected to each other.
This problem is NP problem. Given a set of nodes, you can very easily find out whether it is a clique or not.
For example:





Clique of size 4

Prime Number

Finding Prime number is NP. Given a solution, it is easy to find if it is a Prime or not in polynomial time. Finding prime numbers is important as cryptography heavily uses prime numbers.

```
func isPrime(n int) bool {
    if( n < 1){
        return false
    }
    i := 2
    for (i*i <= n) {
        if(n%i == 0){
            return false
        }
    i += 1
    }
    return true
}
```

Checking will happen until the square root of number so the Time Complexity will be O(√n). Hence, prime number finding is an NP problem as we can verify the solution in polynomial time.

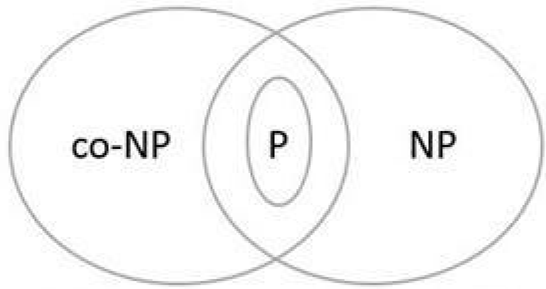
Graph theory have wonderful set of problems

- Shortest path algorithms?
- Longest path is NP complete.
- Eulerian tours is a polynomial time problem.
- Hamiltonian tours is a NP complete

Class co-NP

Set of problems for which there is a polynomial time checking algorithm. Given a solution, if we can check in a polynomial time if that solution is incorrect the problem is co-NP problem.

Class co-NP Definition: The class co-NP contains all decision problems such that there exists a polynomial time proof that can verify if the problem does not have the right “yes/no” answer.



Relationship between P, NP and co-NP

Class P is Subset of Class NP

All problems that are P also are NP ($\mathbf{P} \subseteq \mathbf{NP}$). Problem set P is a subset of problem set NP.

Searching

If we have some number sequence a1, a2, a3…. an. We already know that searching a number X inside this list is of type P.

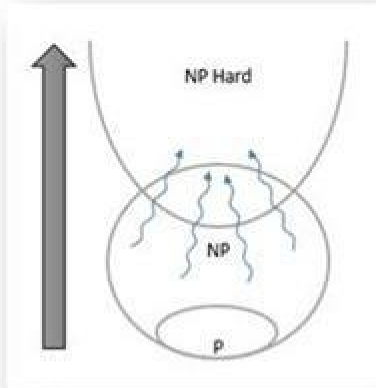
If it is given that number X is inside this sequence, then we can verify by looking into every entry again and find if the answer is correct in polynomial time (linear time.)

Sorting

Another example of sorting a number sequence, if it is given that the list b1, b2, b3.. bn is a sorted then we can loop through this given list and find if the list is really sorted in polynomial time (linear time again.)

NP–Hard:

A problem is NP-Hard if all the problems in NP can be reduced to it in polynomial time.

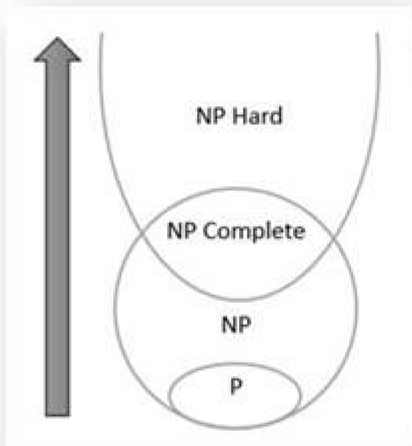


NP–Complete Problems

Set of problem is NP-Complete if it is an NP problem and an NP-Hard problem.
It should follow both the properties:

1) Its solutions can be verified in a polynomial time.
2) All problems of NP are reduced to NP complete problems in polynomial time.

You can always reduce any NP problem into NP-Complete in polynomial time. In addition, when you get the answer to the problem, then you can verify this solution in polynomial time.



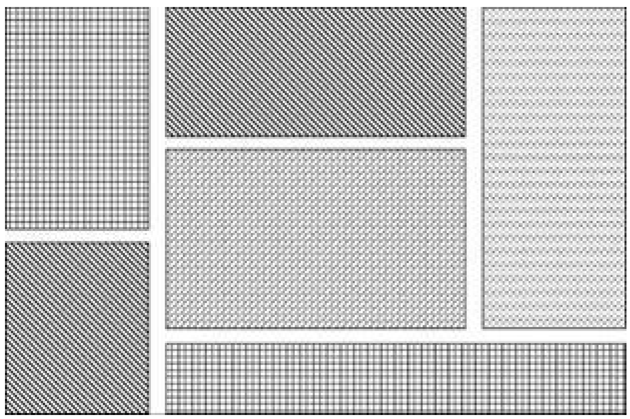
Any NP problem is polynomial reduced to NP-Complete problem, if we can find a solution to a single NP-Complete problem in polynomial time, then we can solve all the NP problems in polynomial time. However, so far no one is able to find any solution of NP-Complete problem in polynomial time.

P ≠ NP

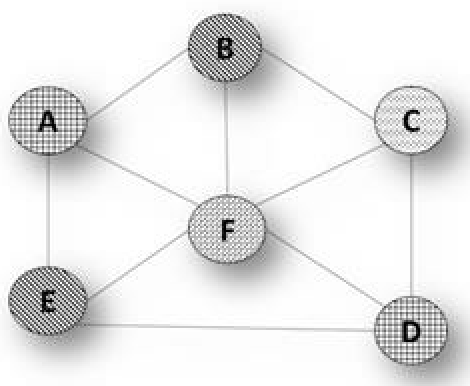
Reduction

It is a process of transformation of one problem into another problem. The transformation time should be polynomial. If a problem A is transformed into B and we know the solution of B in polynomial time, then A can also be solved in polynomial time.
For example,
Quadratic Equation Solver: We have a Quadratic Equation Solver, which solves equation of the form $ax^2 + bx + c = 0$. It takes Input a, b, c and generate output r1, r2.
Now try to solve a linear equation 2x+4=0. Using reduction second equation can be transformed to the first equation.
$2x+4 = 0$
$X^2 + 2x + 4 = 0$

ATLAS: We have an atlas and we need to color maps so that no two countries have the same color. Let us suppose below is the various countries. Moreover, different pattern represents different color.



We can see that same problem of atlas coloring can be reduced to graph coloring and if we know the solution of graph coloring then same solution can work for atlas coloring too. Where each node of the graph represents one country and the adjacent country relation is represented by the edges between nodes.



- The sorting problem reduces (≤) to Convex Hull problem.
- SAT reduces (≤) to 3SAT

Traveling Salesman Problem (TSP)

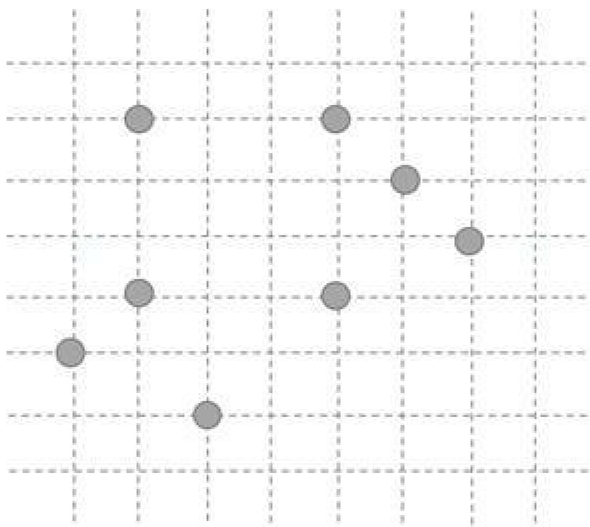
The traveling salesman problem tries to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.
Alternatively find the shortest Hamiltonian circuit in a weighted connected graph. A cycle that passes through all the vertices of the graph exactly once.

```
Algorithm TSP
Select a city
MinTourCost = infinite
For ( All permutations of cities ) do
    If( LengthOfPathSinglePermutation < MinTourCost )
        MinTourCost = LengthOfPath
```

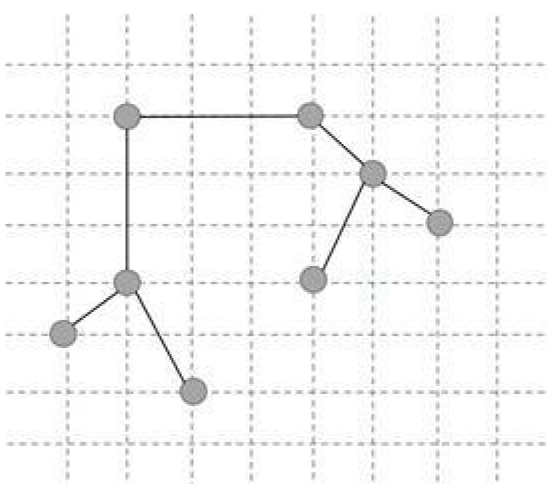
Total number of possible combinations = (n-1)!
Cost for calculating the path: $\Theta(n)$
So the total cost for finding the shortest path: $\Theta(n!)$

It is an NP-Hard problem there is no efficient algorithm to find its solution. Even if some solution is given, it is equally hard to verify that this is a correct solution or not. However, some approximate algorithms can be used to find a good solution. We will not always get the best solution, but will get a good solution.

Our approximate algorithm is based on the minimum spanning tree problem. In which we have to construct a tree from a graph such that every node is connected by edges of the graph and the total sum of the cost of all the edges it minimum.

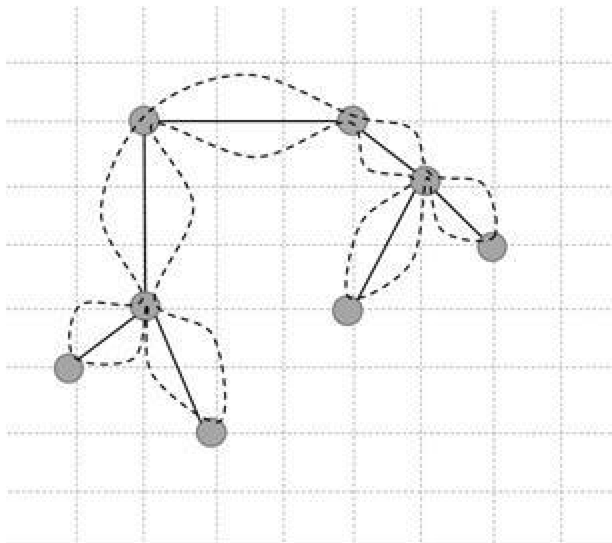


In the above diagram, we have a group of cities (each city is represented by a circle.) Which are located in the grid and the distance between the cities is same as per the actual distance. And there is a path from each city to another city which is a straight path from one to another.

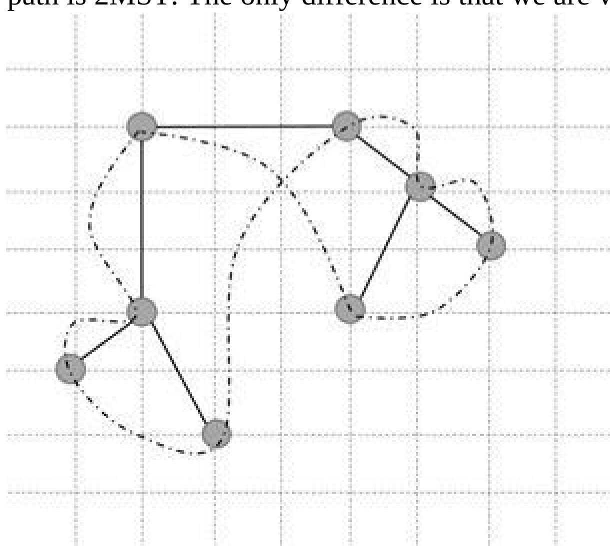


We have made a minimum spanning tree for the above city graph.

What we want to prove that the shortest path in a TSP will always be greater than the length of MST. Since in MST all nodes are connected to the next node, which is also the minimum distance from the group of node. Therefore, to make it a parth without repeating the nodes we need to go directly from one node to other without following MST. At that point, when we are not following MST we are choosing an edge, which is grater, then the edges provided by MST. So TSP path will always be greater than or equal to MST path.



Now let us take a path from starting node and traverse each node on the way given above and then come back to the starting node. The total cost of the path is 2MST. The only difference is that we are visiting many nodes multiple times.



Now let us change our traversal algorithm so that it will become TSP in our traversal, we did not visit an already visited node we will skip them and will visit the next unvisited node. In this algorithm, we will reach the next node by as shorter path. (The sum of the length of all the edges of a polygon is always greater than a single edge.) Ultimately, we will get the TSP and its path length is no more than twice the optimal solution. Therefore, the proposed algorithm gives a good result.

End Note

Nobody has come up with such a polynomial-time algorithm to solve a NP-Complete problem. Many important algorithms depends upon it. However, at the same time nobody has proven that no polynomial time algorithm is possible. There is a million US dollars for anyone who can do either solve any NP Complete problem in polynomial time. The whole economy of the world will fall as most of the banks depends on public key encryption will be easy to break if P=NP solution is found.

CHAPTER 22: INTERVIEW STRATEGY

Introduction

Success in tech interview depends on so many factors, your non-technical skills, your technical skills, etc. Above all the interviewers should be convinced that they would enjoy working with you.

Resume

A good resume is one that communicate your skills and accomplishments in a clear and effective way.

A good resume format has the following attributes:

1. Multiple Columns: Multiple columns make it easier for someone to skim your company name, positions, collage, and other key facts.
2. Brief: Interviewer is going to spend about 30 Sec reading your resume. You should just focus on the highlights. One page is all you need, but if you are 10+ years of experience, then you can justify two pages.
3. No Junk: No objective, No oath, Summary section/Key skills section may be fine, if your resume is short and concise then you do not need a summary section.
4. Use Tables: You can use tables, but it should not waste space.
5. Highlights: highlights should be short. Keep your highlights to one-liner.
6. Neat: Keep your resume neat and clean. Use appropriate Fonts and Formatting. Bold to represent highlights and maybe italics in some places.

Nontechnical questions

Prepare for various non-technical questions. The first thing to do is to prepare answers of any question that is related to your resume. The interviewer is going to look into it and ask a few questions to get an idea about you. So go through all the past/current job and projects and make sure you know about them and your role.

These questions may be like:

1. What was the most challenging activity you have done in project ABC?
2. What did you learn from project ABC?
3. What are your responsibilities in the current job?
4. What was the most interesting thing you have done in your current job?
5. Which course in university did you like most and why?

Technical questions

Solving a technical question is not just about knowing the algorithms and designing a good software system. The interviewer wants to know you approach towards any given problem.

Many people make mistakes, as they do not ask clarifying questions about a given problem. They assume many things and begin working with that. Well the truth is the interviewer to actually expect you to ask constraints questions. There is lot of data that is missing that you need to collect from your interviewer before beginning to solve a problem.

For example, Let us suppose the interviewer ask you to give a best sorting algorithm.
Some interviewee will directly jump to Quick-Sort **O(nlogn)**. Oops, mistake you need to ask many questions before beginning to solve this problem.

Questions:
1. What kind of data we are going to sort? Are they integers?
2. How much data are we going to sort?
3. What exactly is this data about?
4. What kind of data-structure used to hold this data?
5. Can we modify the given data-structure? And more…?

Answer:
1. Yes, they are integers.
2. May be thousands.
3. They store a person's age.
4. Data are given in the form of some array.
5. No, you cannot modify the data structure provided.

Ok from the first answer, we will deduce that the data is integer. The data is not so big it just contains a few thousand entries. The third answer is interesting from this we deduce that the range of data is 1-150. Data is provided in an array. From fifths answer we deduce that we have to create our own data structure and we cannot modify the array provided. So finally, we conclude, we can just use bucket sort to sort the data. The range is just 1-150 so we need just 151-capacity integral array. Data is under thousands so we do not have to worry about data overflow and we get the solution in linear time **O(N)**.

CHAPTER 23: SYSTEM DESIGN

System Design

The section we will look into questions in which interviewer asks to design a high-level architecture of any software system.

Note: - This is an advance chapter It may be that the user is not able to understand it completely. I would suggest that give it some time read the chapter and try to read online. The more time you give to this chapter the better understanding you will get. It may also help if you give multiple rounds of reading.

There are two kinds of questions in this and which will be asked depends on the type of companies. The first kind of questions is to design some kind of elevator system, valet parking system, etc. In this, the interviewer just wants to test how well you are able to design a system, especially how well your classes are interacting.

The Second kind of system design problems is more interesting, in which the interviewer asks you to design some kind of website or some kind of service or some API interface. For example, design google search engine or design some feature of Facebook like how friends mapping is done on Facebook, design a web-based game that allows 4 people play poker etc. They are interesting one and in this, the interviewer can ask about scalability aspect.

Now comes a question to our mind, how would you design google search engine in 10-15 minutes? Well, the answer is you cannot. It took many days if not years by a group of a smart engineer to design google search engine. The interviewer is expecting a Higher-level architecture of the system that can address the given Use-Cases and Constraints of the problem in hand. There is no single right solution. The same problem can be solved in a number of ways. The most important thing is that you should be able to justify your solution.

System Design Process

Let us look into a 5 Steps approach for solving system design problems:

1. Use Cases Generation
2. Constraints and Analysis
3. Basic Design
4. Bottlenecks
5. Scalability

Use Cases

Just like algorithm design problems, the system design questions are also most likely weakly defined. There is so much information that is missing and without this, the design is impossible. So first thing in the design process is to gather all the possible use cases. You should ask questions to the interviewer to find the use case of the system. The interviewer wants to see your **requirement gathering capability**. Same as algorithm questions never assume things, which are not stated.

Constraints and Analysis

This is the step in which you will define various constraints of the system and then analyze them. Your system design will depend on the analysis that you do in this step. In this step, you need to find answers to questions like. How many users will be using the system? What kind of data that we are going to store? Etc.

Basic Design

In this step, you will design the most basic design of the system. Draw your main components and make connections between them. In this step, you need to design a system with the supposition that there is no memory limitation and all data can fit in one single machine. You should be able to justify your idea. In this step, you need to handle all the use-cases.

Bottlenecks Analysis

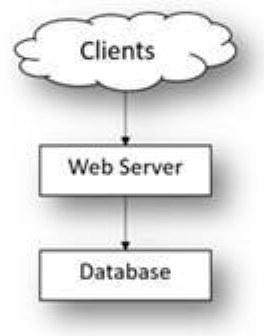
In this step, you will find the one or more bottlenecks on the basic design you had proposed. The "Scalability Theory" given below will help to identify the bottlenecks. You need to know the below theory which experts had developed over time. In this step, you will consider how much data your proposed system can handle, memory limitations etc.

Scalability

In this step, you will remove all the bottlenecks of the system and you are done. There may be multiple iterations between "Bottlenecks analysis" and "Scalability" until we reach our final solution. We will be reading various concepts like Vertical scaling, Horizontal scaling, Load-Balancer, Redundancy and Caching in this chapter. "Scalability Theory" given below will help you to understand these concepts.

Scalability Theory

In this section, we will be designing a generic web server, which will be handling a large number of requests. You can imagine it as some sort of website like Facebook in which large number of users are accessing it.



Vertical scaling

Vertical scaling means that you scale by adding more resources (Higher speed CPU, More RAM etc.) To your existing machine.

Vertical scaling has its own limit it can help you to handle more load, but until its limit is reached, then we have to go for horizontal scaling.

Horizontal scaling

Horizontal scaling means that you scale by adding more machines to your pool of resources.

Distribute the request by distributing the request among more than one web server. In doing this we need to have a load balancer, which will distribute the request among the servers.

Load Balancer (Application layer)

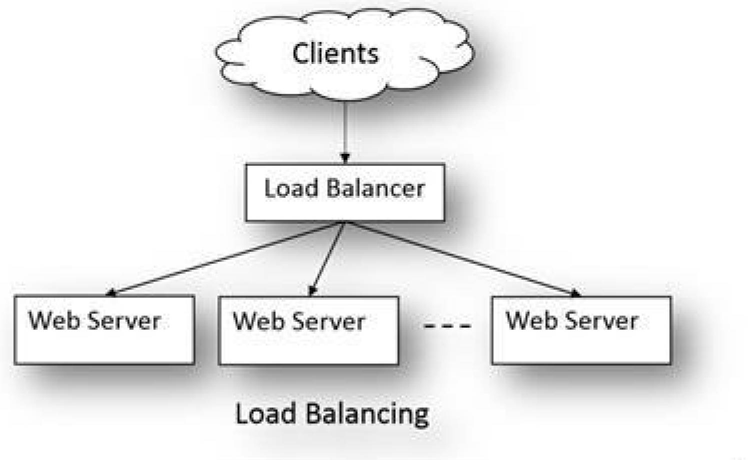
Load balancer has to decide which server should serve the next request. So distributing the load can be made using different strategies:

1. **Round Robbin**: Round robin is the way of distributing requests in a sequential fashion. The request is sent to the server 1 then the next request is sent to server 2 and so on until we reach the end of the server list. Then when we reach the end, it is sent again to server 1. Round robin has a problem that a server, which is already busy, may get another request. Round robin also has a problem with sticky sessions. We want that a request to be sent to the same server the next time.

2. Another approach is to select server corresponds to the hash value of the data. Find the hash value of the data, mod the hash value by the number of servers. Assign the job to a machine whose value we got after mod. Stick session problem is already solved in hash value approach. However, the problem of uneven load distribution is there, there is possible to have a more load sent to a server, which is already busy.

3. May be the load balancer know, how much load each server has or how busy each the server is. Moreover, will send the next request to the least busy server.

4. The server can be a specialized one serving image some serving video and some serving other data.



Problems of load Balancing

Consider a customer who had selected some items in his buy cart on Amazon. When he selects another item then it should be added to the same cart so it should be sent to the same server. In addition, the user profile that is saved to one server and if the user request reaches the other server his profile will be empty this is also not a good idea.

This problem can be solved by making the load balancer decide that a particular user request would always go to the same server. The user profile and cart details should be saved in some database.

Stick session: same sessions should lend to the same server. How to get this done. The first approach is that we store the IP address returned by load balancing into a cookie and then use this IP address in the subsequent requests. However, this reveals the IP address of the server to the world that we do not want. Therefore, another solution is that we use some session id that is a number that the load balancer knows that belongs to which server. By this, we are preventing our servers being exposed to the outer world and prevent it from being attacked.

Load Balancer (Database layer)

1. The most basic approach is **Round Robin**. Data is distributed in a circular fashion. First, data go to the first database, the second will go to the second database and so one. Each database server had an equal load. However, it has a disadvantage that the data lookup is complex. And need a large lookup table.

2. Another approach is to divide the data in such a way that all the data will go to the first machine until it reaches its maximum capacity. When maximum capacity is reached, then data goes to the second machine and so on. This approach has an advantage that only the required number of machines is used. However, it has a disadvantage that the data lookup is complex. And need a large lookup table.

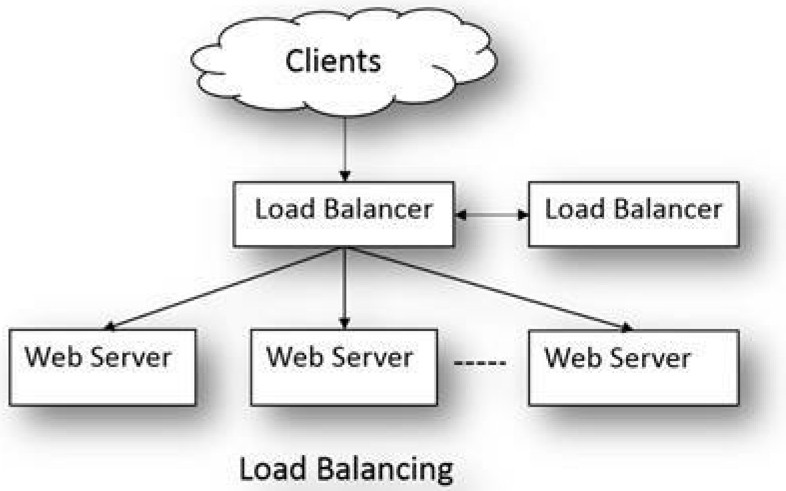
3. Another approach is to select database corresponds to the hash value of the data. Find the hash value of the data. Mod the hash value by the number of databases. The data are then stored in the database value we got after modulus. For has a value approach we do not require any lookup table. We can find the database, which is storing the data, by finding the hash value. However, the problem of uneven distribution of data is there, there is possible to have a more data sent to a database, which has already reached its maximum capacity. In this case, we need to find a better load-balancing key or split the data from the database into a number of databases.

4. In the hash value, based distribution of data there is no relation between the data that is stored in a particular database. Information about the data can be used to make the database accessible faster. For example, in social networking like Facebook, if someone who lives in India is more likely to have friends from India. And someone who lives in the USA is more likely to have friends in the USA.

5. Perhaps location aware (approach 4) and the hash value based (approach 3) distribution of data may be the best approach to keep the data so that it can take advantage of both the approaches. Country code and user ID can be used to get the location of the database.

Redundancy

There is one problem in our system, there is a redundancy in the servers but our load balancer is now our single point of failure. We add a secondary load balancer in case the primary load balancer dies, then secondary load balancer becomes primary and then all the requests will be handled by it.



Raid (Redundant List of Inexpensive Disk): Raid is a technology to create redundancy in the databases. Multiple hard drives are used to replicate data, thereby proving redundancy.

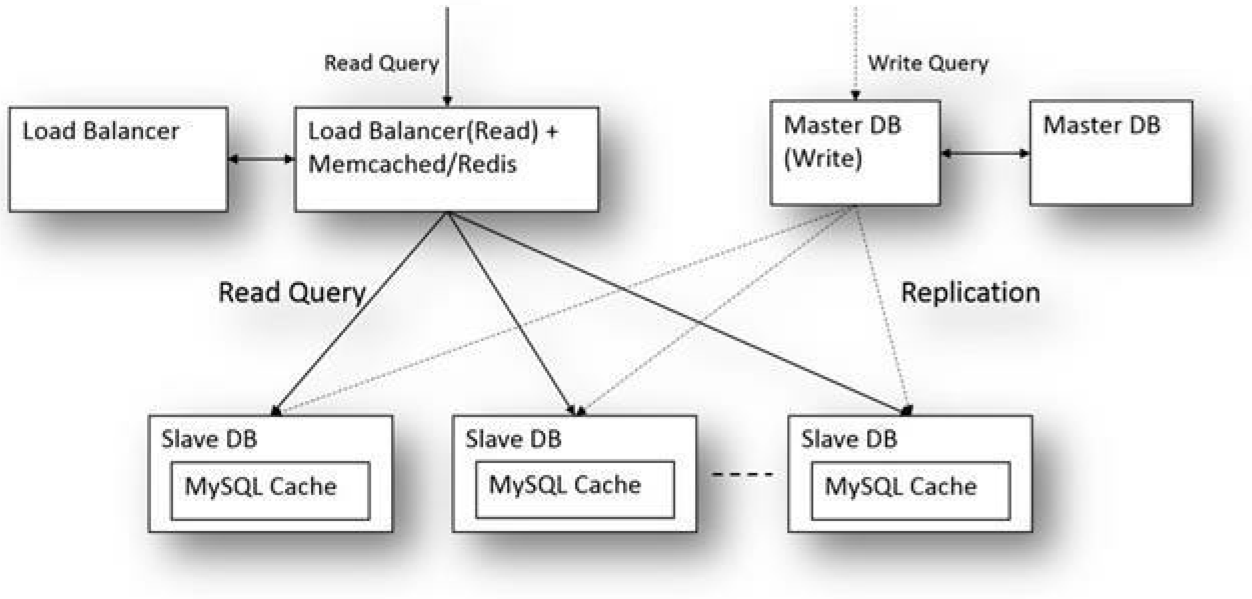
Caching

A cache is a simple key-value store and it should reside as a buffering layer between your application and your data storage. Whenever your application wants to read, it first tries to retrieve the data from your cache. Only if data is not present in the cache, then only it tries to get the data from the main database.
Caching improves application performance by storing portion of data in memory for low-latency access. We need databases and access to the database is slow, so we use multiple types of caching to make our system faster. Database servers itself does caching so do the other entities in between the user and the database.

Memcached: It is a server software, what it does is it kept whatever you access in memory. It can run on the same server as the webserver or it can run on a separate machine all together.

Redis: It is a data server based on **NoSQL**, which is a key-value data store. Data is stored as the value with respect to corresponding key. This data is later retrieved by the use of the key. Redis is used for caching it is best to store the whole object as one instance so that the data can be accessed in parallel and data expiration will flush out the whole object.

There is a problem since ram is finite, then the cache will get full. The expired object will be removed so everything that is accessed then its expiry will be reset and if there is an object that is not used for some time then it was deleted. Cache is more important when the website that we are designing is more read heavier than a write.

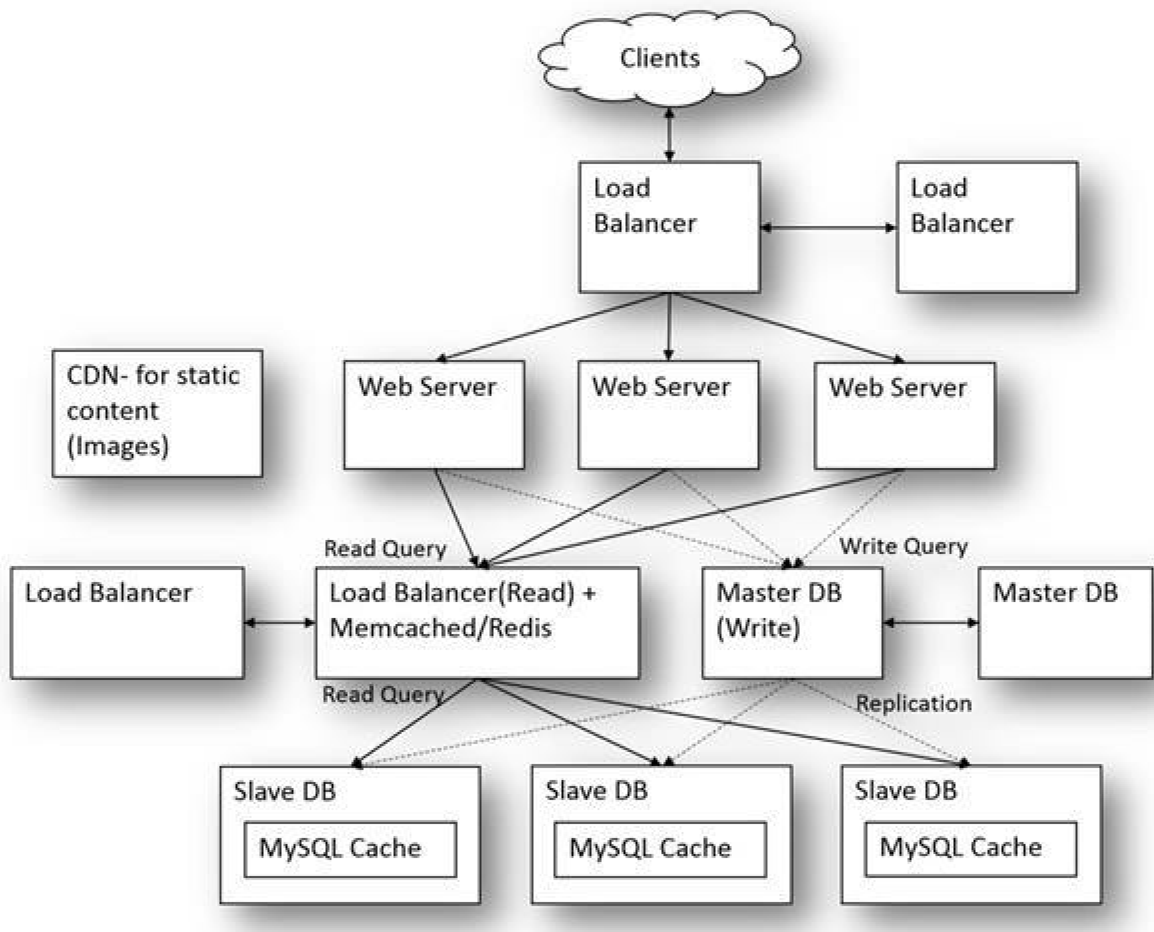


A complete web server implementation

The summary of the above system.

1. The Web-Servers of scalable web service is hidden behind a load balancer. The load balancer evenly distributes load across all the servers.
2. The user should get the same result from web-server regardless which server is actually serving the request. Therefore, every server should be identical to each other. Servers should not contain any data like session information or user profile.
3. Session need to be stored in a centralized data store (DB) which is accessible to all the servers. Data can be stored in some external database. Redundancy in the database is provided by raid technology.
4. The database is slow, so we need a cache. In-memory based cache like Redis or Memcached.
5. However, the cache has a problem of expiring. When a table changes, then the cache is outdated.
6. For Memcached there are two options:
    a) We can save queries to the DB
    b) We can save the whole object that will keep us close to web-server.
7. CDN (Content delivery networks) can be used to provide a pre-processed web page.

Below diagram will give you a complete picture of the whole system.


Clients
Load
Balancer
Load
Balancer
CDN- for static
content
(Images)
Web Server
Web Server
Web Server
Read Query
Write Query
Load Balancer
Load Balancer(Read) +
Memcached/Redis
Master DB
(Write)
Master DB
Read Query
Replication
Slave DB
MySQL Cache
Slave DB
MySQL Cache
Slave DB
MySQL Cache

Design simplified Facebook

Design simplified Facebook where people can add other people as friends. In addition, where people can post messages and that messages are visible on their friend's page. The design should be such that it can handle 10 million of people. There may be, on an average 100 friends each person has. Every day each person posts some 10 messages on an average.

Use Case

1. A user can create their own profile.
2. A user can add other users to his friend list.
3. Users can post messages to their timeline.
4. The system should display posts of friends to the display board/timeline.
5. People can like a post.
6. People can share their friends post to their own display board/timeline.

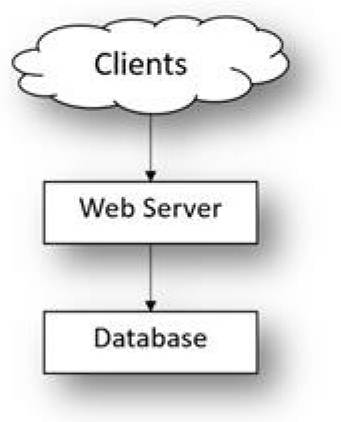
Constraints

1. Consider a whole network of people as represented by a graph. Each person is a node and each friend relationship is an edge of the graph.
2. Total number of distinct users / nodes: 10 million
3. Total number of distinct friend's relationship / edges in the graph: 100 * 10 million
4. Number of messages posted by a single user per day: 10
5. Total number of messages posted by the whole network per day: 10 * 10 million

Basic Design

Our system architecture is divided into two parts:

1. First, the web server that will handle all the incoming requests.
2. The second database, which will store the entire person's profile, their friend relations and posts.



First, three requirements creating a profile, adding friends, posting messages are written some information to the database. While the last operation is reading data from the database.

The system will look like this:

1. Each user will have a profile.
2. There will be a list of friends in each user profile.
3. Each user will have their own homepage where his posts will be visible.

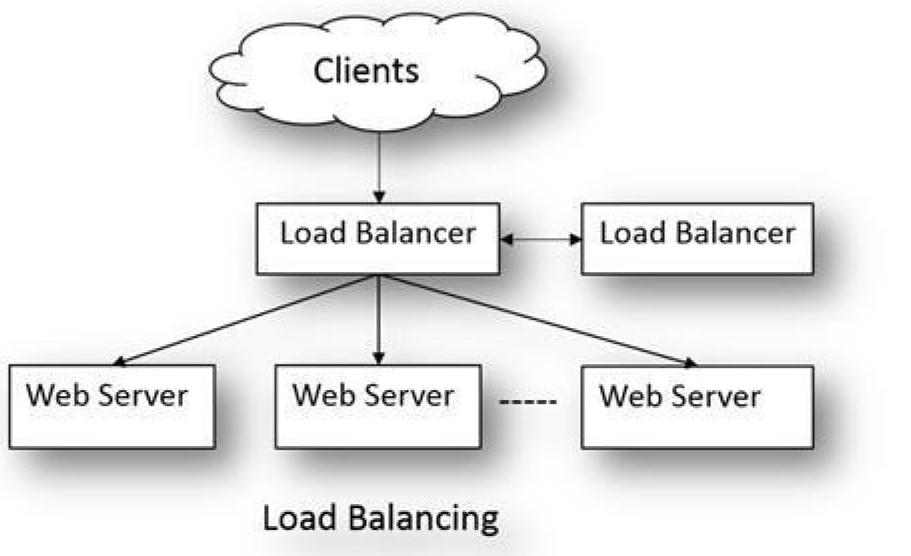
A user can like any post of their friend and that likes will reflect on the actual message shared by his friend.
If a user shares some post, then this post will be added to the user home page and all the other friends of the user will see this post as a new post.

Bottleneck

A number of requests posted per day is 100 million. Approximate some 1000 request are posted per second. There will be an uneven distribution of load so the system that we will design should be able to handle a few thousand requests per seconds.

Scalability

Since there is, a heavy load we need horizontal scaling many web servers will be handling the requests. In doing this we need to have a load balancer, which will distribute the request among the servers.



This approach gives us a flexibility that when the load increases, we can add more web servers to handle the increased load.

These web servers are responsible for handling new post added by the user. They are responsible for generating various user homepage and timeline pages. In our diagram, the client is the web browser, which is rendering the page for the user.

We need to store data about user profile, Users friend list, User-generated posts, User like statues to the posts.

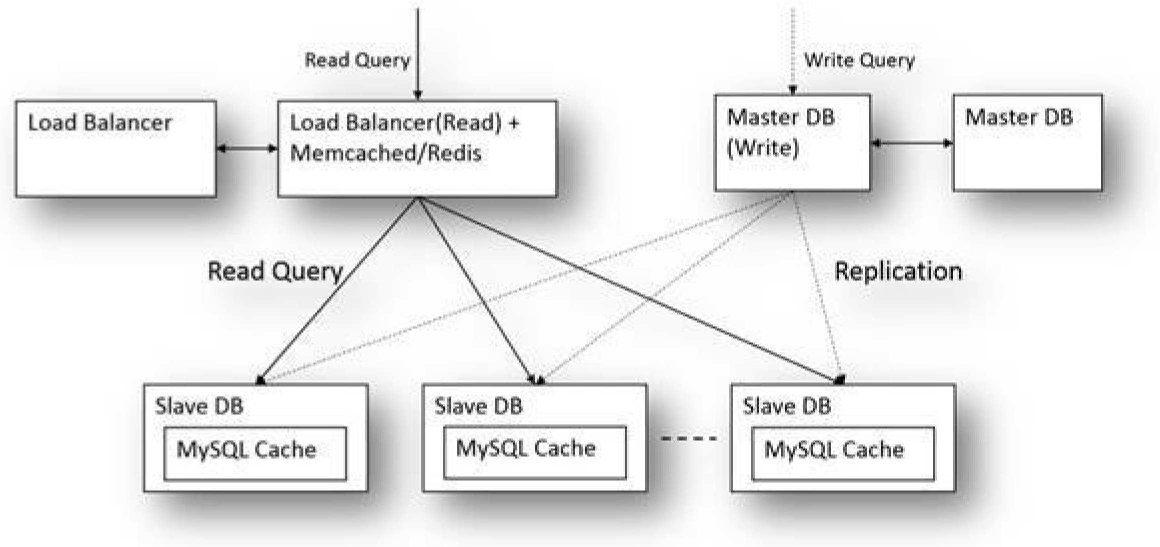
Let us find out how much storage we need to store all this data. The total number of users 10 million. Let us suppose each user is using Facebook for 5 to 6 years, so the total number of posts that a user had produced in this whole time is approximately 20,000 million or 20 billion. Let us suppose each message consists of 100 words or 500 characters. Let us assume each character take 2 bytes.

Total memory required = 20 * 500 * 2 billion bytes.
= 20,000 billion bytes
= 20, 000 GB
= 20 TB

1 gigabyte (GB) = 1 billion bytes

1000 gigabytes (GB) = 1 Terabytes

Most of the memory is taken from the posts and the user profile and friend list will take nominal as compared with the posts. We can use a relational database like SQL to store this data. Facebook and twitter are using a relational database to store their data.



Responsiveness is key for social networking site. Databases have their own cache to increase their performance. Still database access is slow as databases are stored on hard drives and they are slower than RAM. Database performance can be increased by replication of the database. Requests can be distributed between the various copies of the databases.

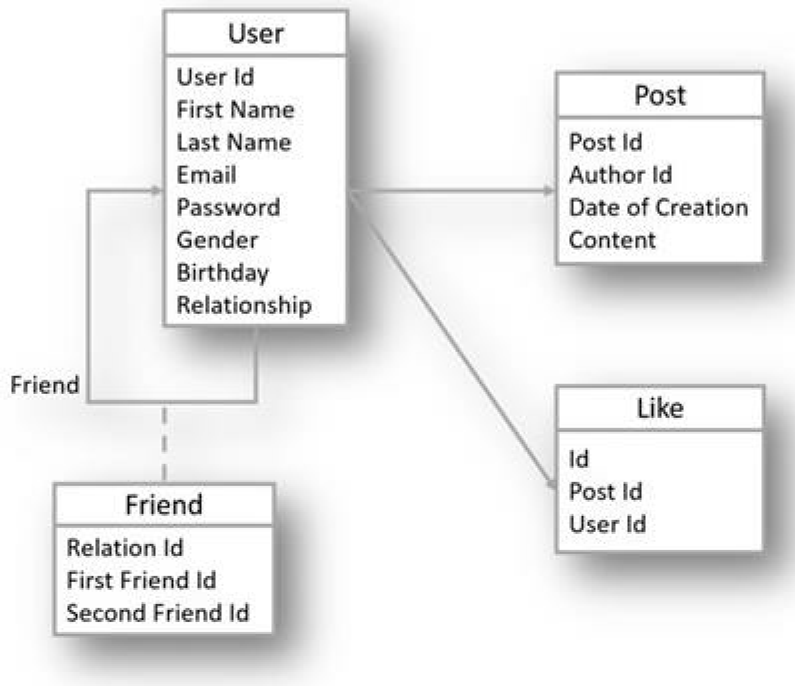
Also, there will be more reads then writes in the database so there can be multiple slave DB which are used for reading and there can be few master DB for writing. Still database access is slow to we will use some caching mechanism like Memcached in between application server and database. Highly popular users and their home page will always remain in the cache.

There may be the case when the replication no longer solves the performance problem. In addition, we need to do some Geo-location based optimization in our solution.

Again, look for a complete diagram in the scalability theory section. If it were asked in the interview how you would store the data in the database.

The schema of the database can look like:

Table Users · User Id · First Name · Last Name · Email · Password · Gender · Birthday · Relationship	Table Posts · Post Id · Author Id · Date of Creation · Content
Table Friends · Relation Id · First Friend Id · Second Friend Id	Table Likes · Id · Post Id · User Id


User
User Id
First Name
Last Name
Email
Password
Gender
Birthday
Relationship
Post
Post Id
Author Id
Date of Creation
Content
Like
Id
Post Id
User Id
Friend
Friend
Relation Id
First Friend Id
Second Friend Id

Design Facebook Friends suggestion function

Design a system to implement a friend suggestion functionality of Facebook, with millions of users. The algorithm should suggest all the friends of the immediate friends as a proposed list to add as friends.

Use Case

The system should suggest friends of the friends as suggested new friends.

Constraint

Millions of user’s lot of data with billions of relations.

Basic Design

Forget about millions of users. Just consider there are only a few persons and they are connected with each other as friends.

Consider that people are represented by vertices of graphs and their friendship relation is represented by edges.

Since there are only a few people, then we can keep everything in memory and find the friend suggestion using Breadth First Traversal.
We just need to find the nodes, which are just 2 degrees apart from the starting node.

Bottleneck

Since there are millions of users, we cannot have everything in memory. Since there are millions of users, we cannot keep the data on one machine. One friends’ profiles may lie on many different machines.

Scalability

Since there are millions of users, their user profile is distributed among many different database servers. User profiles can be distributed depending upon Geo-Location. The Indian users profile will lie in a server located in India and US citizen's profile lie in the server located in the US.

Each user will have corresponding User Id associated with them. Some portion of ID can be used to get Geo location of the user. Another portion of user id can find the user profile on that server.

The user profile is not that frequently updated so there is more read than write. So single master writer - multiple slave reader architecture is most suitable for this application.

The application server can process the data; it can do the optimization to query less from the database by accumulating user list to be processed.

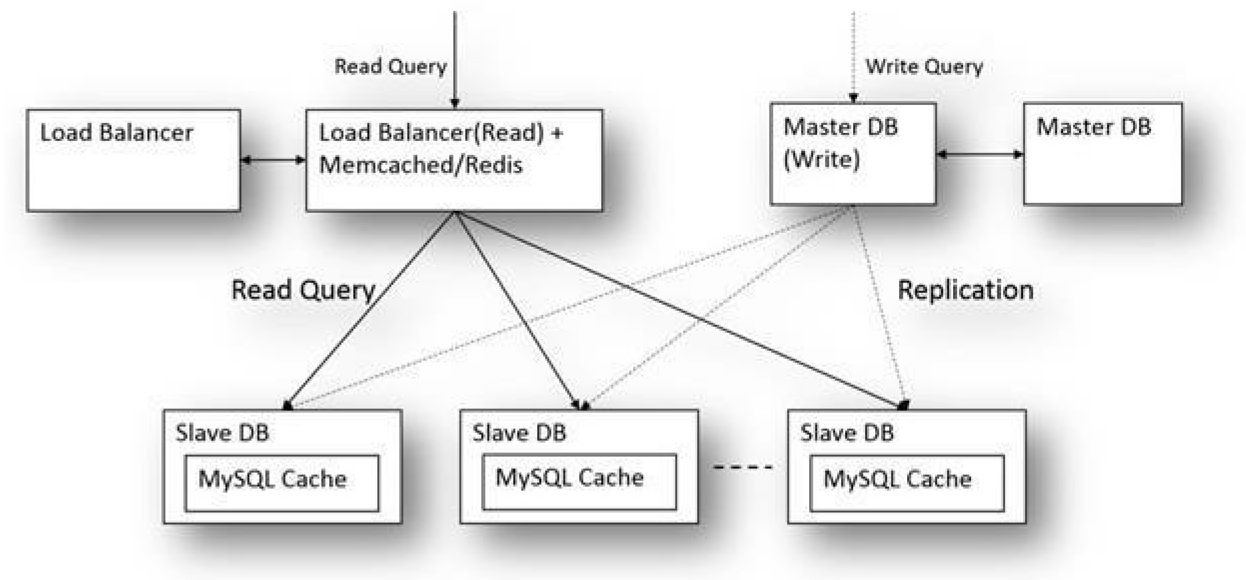
```
class system :
    personIdToMachineIdMap = {}
    machineIdToMachineMap = {}

    getMachine(machineId):
        # return machine from id.
```

```
getPerson(personId):
    machienId = personIdToMachienIdMap[personId]
    m = machineIdToMachineMap[machienId]
    return m.getPersonWithId(personId)
```

Optimization: Reduced the number of jumps by first finding the list of friends whose profile is on the same machine. Then send the find next degree friends query that will return the list of next level friends. By doing, this work is distributed among various machines. Finally, the result of the various queries will be merged, and then friend list is suggested.

Better result: You can calculate the degree of the friends with the friend list. The person who is a friend of many of my friends is more likely to be my friend than the person who is just a friend of one of my friends. We need to keep track of the friend reference counts by keeping Hash-Table for the friend list and make the count 1 whenever we find a new person otherwise increase the count by 1.

If we want to take advantage of caching, then we need to add some database cache in between.



There can be multiple web servers, which will be querying the databases. Also multiple users who are accessing their Facebook profile and each one of them is proposed with new friends list so the final architecture is again same as the one proposed in the complete web server implemented in scalability theory.

Design a shortening service like Bitly

Use Case

Basic use case:

1. Shortening takes a URL and returns a short URL.
2. Redirection takes a short URL and redirects to the original URL.
3. Custom URL.
4. High availability of the system.

Additional use cases:

1. Analytics
2. Automatic link expiration.
3. Manual link removal.
4. Specific company URL.
5. UI or just API

Requirement Analysis/ Math
Firs we need to find the usage pattern.
You can directly ask this data from the interviewer or you can derive it using some data that the interviewer provides. Let us suppose that the interviewer tells that there will be 1 billion requests per month. In addition, out of these 10% times, it is a new request and 90% of the time, it is a redirection of the already shortened URL. Let us write down the data that we get.

1. 1BN requests per month
2. 10% are for new URL/shortening and 90% are for redirection.
3. New URLs per month is 100MLN
4. Requests per second 1BN/ (30*24*3600) = 385. Roughly, you can assume it 400 requests per seconds.
5. Total number of URLs stored in 5 years.
   5* 12* 100 MLN = 6BN URLs in 5 years.
6. Let us suppose the space required by each URL is 500bytes.
7. Let us suppose the space required by each Hash code for corresponding URLs is 6byte long.
8. Total data we need to store in five years. 3TBs for all the URLs and 36gb for hashes
   6,000,000,000 * 500 bytes = 3 terabytes
   6,000,000,000 * 6 bytes = 36 gigabytes
9. New data write requests per second: 40 * (500+6): 20k

Basic design

Web server: provide the website for the Bitly service where users can generate the short URL.
Application Server: provides the following services:

1. Shortening service
2. Redirection service
3. Key = Hash Function (URL)

Database Server:

1. Keep track of hash to URL mapping.
2. Works like a huge Hash-Table stores the new mapping and retrieves old mapping given key.

Bottleneck

1. Traffic is not much
2. Data storage can be a problem.

Scalability

Application Server:
1. Start with the single machine.
2. Test how far it takes up.
3. We do a vertical scaling for some time.
4. Add load balancer and a cluster of machines to handle spikes and to increase availability.

Data Storage:
1. Billions of objects
2. Each object is small
3. There is no relationship between objects
4. Reads are more than write.
5. 3TBs of URLs and 36GB of hash.

MySQL:
1. Widely used
2. A mature technology
3. Clean scaling paradigms (master/slave, master/ master)
4. Used by Facebook, google, twitter etc.
5. Index lookup is very fast.

Mappings: <Hash, URL>
1. Use only MySQL table with two fields.
2. Create a unique index on the hash we want to hold it in memory to speed up lookups.
3. Vertical scaling of MySQL for a while
4. Partition of data into many partitions
5. Master-slave (read from slave and write to master.)
6. Eventually, partition the data by taking the first character of the hash mod the number of partitions.

Stock Query Server

Implement a stock query service that can provide an interface to get stock price information like open price, close price, highest price, lowest price etc. You should provide an interface that will be used to enter these data and interface to read this data.

Use Case

There will be two interfaces to this system.

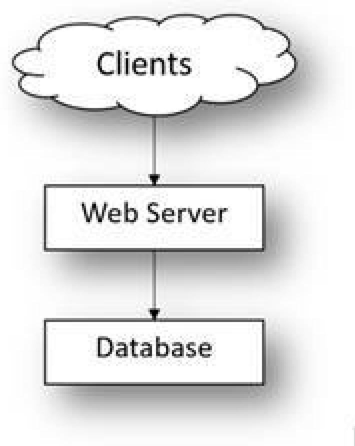
1) First interface to add daily stock price information to the system.
2) Second interface to read stock price information giving the date and stock id as input.

Constraints

Let us suppose the system will be used by thousands of users. For each stock, there will be only one write operation per day. However, for every stock there will be multiple read operations per day. Therefore, the application is more read heavy then write heavy. The solution should be flexible enough so that if new data fields need to be added to the stock they can easily be added to the system. The solution provided should be secure.

Basic Design

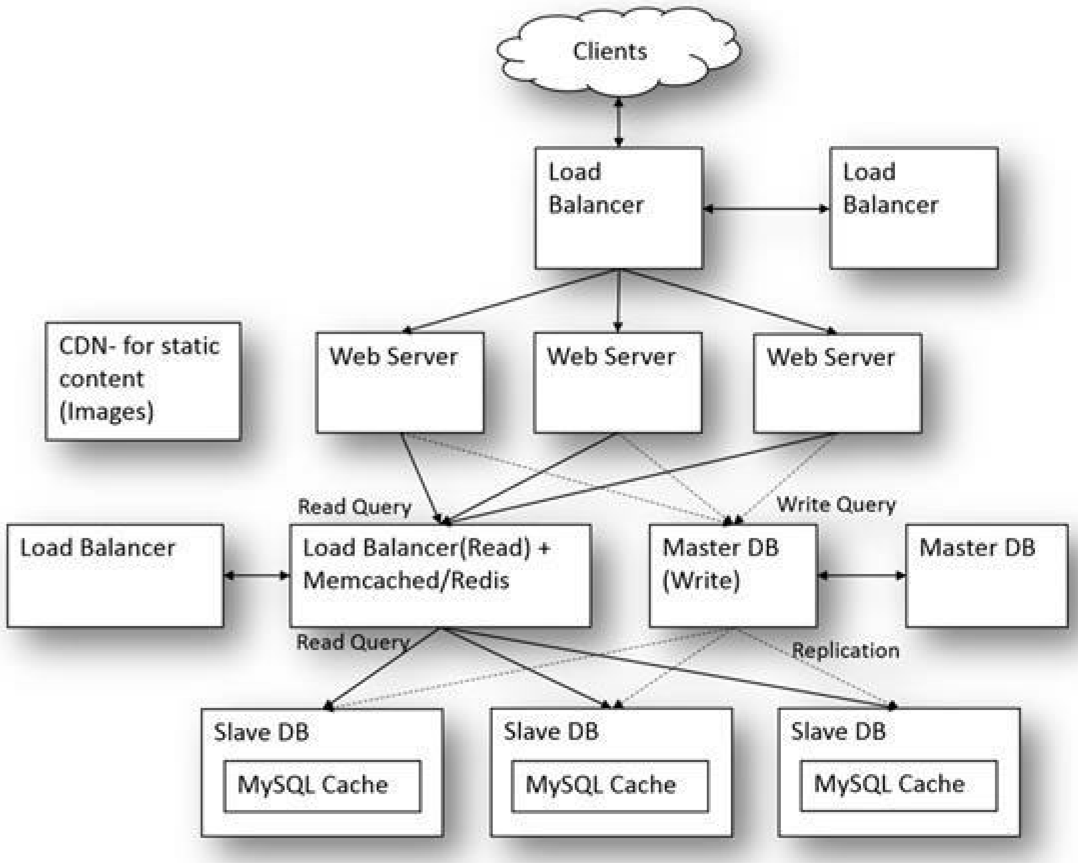
We can use a database like SQL to store stock data. Client can access the database using the web server interface. Below diagram will show the basic architecture.



In the above architecture, the user can access the database using web service. Any number of flexibility can be provided for the use. For example, what is the max price of some stock in 6 months etc.? At the same time, the user does not have access to the data they should not have. We can provide different access of read and write depending on the normal users or administrator. Well-defined rolling back, backing up data and security features are provided by the SQL database. The above architecture is easily extendable to use with a website or some mobile application.

Scalability

Since we have 1000's of users, then having a single web server and a single database is not extendable. We need to distribute data among N number of Databases, which sit behind some load balancer. In addition, multiple N number of web server which will sit behind some load balancer. Each of the load balancers needs to be provided with some redundancy, as they will be a single point of failure. Finally, the solution will look like below diagram. (For details, see scalability theory explained earlier in this chapter)


Clients
Load Balancer
Load Balancer
CDN- for static content (Images)
Web Server
Web Server
Web Server
Read Query
Write Query
Load Balancer
Load Balancer(Read) + Memcached/Redis
Master DB (Write)
Master DB
Read Query
Replication
Slave DB
MySQL Cache
Slave DB
MySQL Cache
Slave DB
MySQL Cache

Design a basic search engine Database

You are given millions of URLs; how would you create your database. So that given a query string of words, how to find the URLs, which contain all the words of the query string. The words can come in any order.

Use Case

1) We are given a list of millions of URLs.
2) The user of the system will provide query string. In addition, we need to return the URLs, which contain all the words of the query string.
3) It is some kind of search engine so we can pre-process the data and make our database.

Requirement Analysis

In the requirement step, you need to find out how many users are going to use this search engine. In our case, let us suppose there are not many users who are going to use our system so the sour main concern is a database.
Maybe we have N number of machines that can be used to fast our data pre-processing.

Basic Design

In this step, we will make the basic design so let us make a working system with just a few URLs. How would you find the required URL from the given URLs, which contains all the words of the input query string?
We can make a Hash-Table in which words are the keys and document ids are values.
“Hello” -> {url1, url2, url3}
“World” -> {url2, url4, url5}
To search the document, which contains “hello world”, we can find the intersection of the two lists. In addition, url2 is the result.

Bottleneck

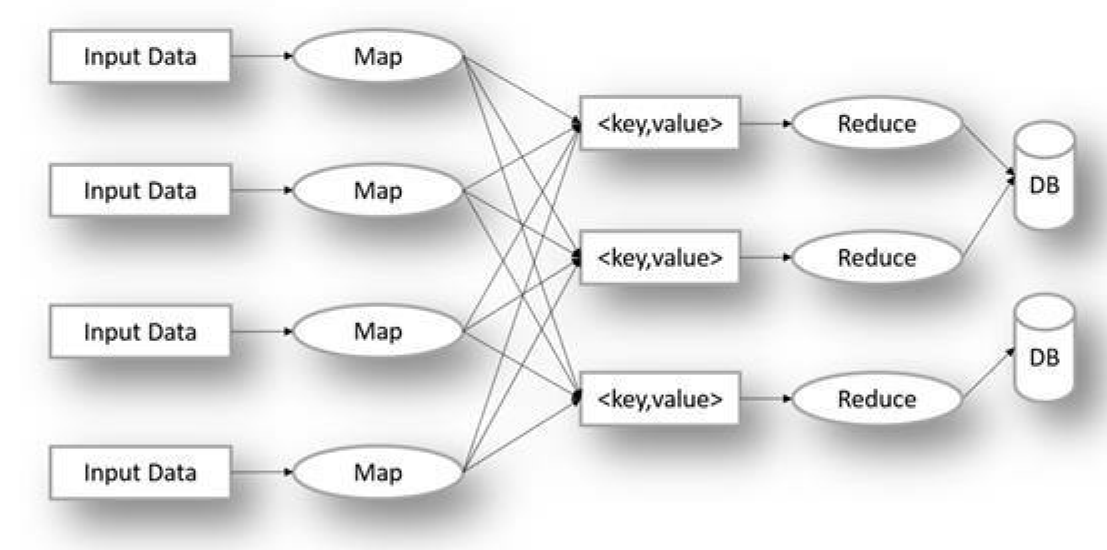
In this step, we will look back to our original problem in which pre-processing of millions of URLs. There may be a number of different words so it may not be possible to keep the whole Hash-Table on a single machine. Therefore, we need to divide the Hash-Table and keep it on a separate machine.
We need to retrieve the URLs that match a given word efficiently. So that we can find the intersection.
Pre-processing all the millions, URLs by single machine will be slow. We need to find a way to parallel process pre-processing step.

Scalability

Let us look into the problem of keeping the Hash-Table in different databases. One solution is to divide the words alphabetically. We can make tables corresponding to each word. Each database contains tables of words under some range. For example, DB1 contains all the words, which start with alphabet “a”, and DB2 contains all the words, which starts with the alphabets “b” and so on. Data is stored in the database. When a database reaches its maximum capacity, the data is stored to next machine and a tree kind of structure can be made. Finding the list of URLs corresponding to some word is easy, we can go to the corresponding database and find the table and get all the data of that table. Finally, we can take the intersection of the result of various words. In addition, the result will be given as output.

Processing of the millions of URLs with a single machine is slow. Therefore, we can divide a bunch of URL processing among an N number of machines, each URL processing is independent of each other and the final Hash-Table of the URLs can be finally combined. This approach of processing independent data and finally combining their result is used in MapReduce.

MapReduce: A MapReduce divides the input dataset into independent chunks, which are processed in parallel. Then their output is combined to get the result.



Design a basic search engine Caching

Given a search engine database implementation which supports QuerySearch() function which will return the best list of URLs based on the words of the query. This time you need to design the web server implementation of this such that there are N number of the web servers, which are responding to the user queries. Any web server can be picked at random. QuerySearch() is a heavy operation so you need to design, caching for this system so that database access is reduced.

Use Case

1. The user of the system will provide query string and system will respond with the proper list of URLs corresponding to his request.
2. Given that, the database operations are very heavy we need to minimize them by caching the queries at the web server.
3. We need to keep the frequent queries in the cache and stale queries need to be removed from the cache.
4. We need to have some proper refresh mechanism for each query.

Basic Design

Let us lust forget about an N number of machines and assume that QuerySearch() operation happens on a single machine. Now we would like to cache queries. Each query will consist of some string. In addition, the result of a query is a list of URLs.

We need to have a quick cache lookup so that we can get the result of the Query from the cache if it is present there. Also, need to have some proper refresh mechanism for each query.
The Hash-Table is most effective to keep the cache. By using a hash, a table lookup is fast. However, if the cache is filled how you would remove the least used data from the cache.

A linked list can be used to remove the old data. You can keep a double linked list to manage the old data removal. Whenever a data is accessed, it can be moved to the front of the linked list and the removal can happen from the end of the linked list.

Taking advantage of both the solutions, we can keep the cache in a linked list and add its reference to the Hash-Table.

Now the last problem of how to remove the data upon the expiry of it. For example, most frequently accessed query result will always remain in the linked list even though that result is changed and if it is accessed again from the database then it will give some updated result. For this, we need to have some TTL (time to live) associated with each query depending upon the result of URLs we get from each query. For example, some weather or current news related queries should have a TTL of days. On the other hand, some historic data should have a long TTL. The TTL can be derived from how frequently the URLs are changing in the query result.

Bottleneck

There are N different web servers. In addition, any particular query can be served by any server.
Data access should be fast.

Scalability

The various solutions that we can think about are:

Approach 1:
Servers can have their own cache. If some query is sent to Machine1 it will catch it in its cache when the same query is sent to it again it will return it from its cache. However, if some query is sent to Machine1 first, it will cache it and if the same query is sent to Machine2, it will again do a database lookup and cache it to its own cache. This implementation is suboptimal as it is doing more number of database lookups than what is actually required.

Approach 2:
Another approach is that each machine stores identical cache. Whenever some database access happened then the same cache is updated by all the web servers. This approach has a drawback that whenever a data is updated in cache same cache update is fired to all the N web servers. Another disadvantage is that all the cache stores the same data so we are wasting precious cache space.

Approach 3:
In this approach, we will divide our cache such that each web server holds a different part of the cache. When a query reach to some web server it knows which webserver actually holds the cache for this query or at least knows that which server is supposed to keep a cache of the particular query.
To do this we need a hash-based approach. We find the server, which serves the query, by just finding the hash (query) percentage N.

When a query request come to some web server, it will find the webserver corresponding to this query by applying the formula. It will ask the QuerySearch() function to that particular server. That server will in turn will query the database if required or provide the result from its own cache.
Now, regarding the cache expiration and old cache removal. We are keeping the TTL corresponding to each query so there can be a thread running which looks for the expired data and remove it from the cache.
In addition, combination of linked list and Hash-Table is used to keep the cache to get rid of less accessed data when the cache is almost full.

As a further improvement, we can think of some sort of Geo location aware webserver selection and cache policy so that query related to India is more supposed to be done in India and query related to china is supposed to come more from china.

Duplicate integer in millions of documents

Given millions of documents with all distinct numbers, find the number, which occurs multiple times.

Basic Design

Consider there are just a few numbers, and we want to find the duplicate numbers.

The first approach is to find keep a sorted list of the numbers and see if the next read number matches with some number in the list.

Another better approach is to find a hash value corresponding to number and add that the number to a Hash-Table.

Constraint

Millions of documents and there is no range of number so we cannot keep everything in memory.

Scalability

We can find the hash value for all the integers and then add that integer to its corresponding hash value file or database. If there is some duplicate, then they will fall in the same file. In the first pass, various files are created and integers are distributed.

In the second pass all, the data of the individual files can be loaded into memory and sorted to find if there is some duplicate value.

We can use the same technique explained above to process the various documents of integer in parallel by different machines and then combine their output to get our result faster.

Zomato

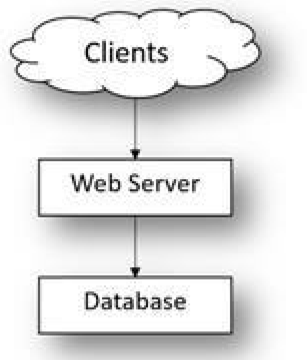
Use Case

1. Given a location, the list of restaurants in that locality needs to be displayed.
2. Given a restaurant name that restaurant's rating, review, and menu need to be displayed
3. There should be some option to find if a delivery option is there in the restaurant.
4. There should be some option to select a restaurant on veg/non-veg category.
5. There should be some option to select restaurants, which serve alcohol.
6. The user should be able to add reviews, add personal ratings to the restaurants.
7. The user has some account or can access as guest.
8. Users/Admin should be able to add new restaurants to the system.

Constraints

1. A number of queries per second suppose 100 queries per second.
2. There are more reads than writes.
3. 90% of the time there is read operation and only 10% of the time there is a write operation.
4. 100 * 60 * 60 * 24 = 8,640,000

Basic Design



The scalability will be same as that of the examples explained above in the case of basic Facebook. Same concept of redundancy, load balancing, scalability etc.

Abstract Design

Each restaurant has some restaurant id associated with it.

1. Data of the restaurant can be Name, Address, Rating, Review List, Veg-Nonveg, and Alcohol etc.
2. The region is a field in the address.
3. Search: When a user does a region based query all the entries of the restaurants in that region need to be displayed to the user.
4. Search: User should be able to search specific restaurant.
5. Add Review/ Rating: When users assess a restaurant, then he should be able to add reviews and rating for the restaurant.
6. Obviously the images are stored in CDN

Application Service layer

1. Start with fewer machines
2. All load balancer + a cluster of machines over time.

3. Traffic spike handling.
4. High availability.

Data Storage Layer:
1. Thousands of restaurants.
2. There are no relationships between the object.
3. Reads are more than writing.
4. Relational database option is MySQL
5. Widely used.
6. Clear scaling paradigms. (Master-Master replication, Master-Slave replication)
7. Index lookups are very fast.

One optimization that we can assign an id to restaurants the id can be derived from the locality so that it would be easy to find restaurants in that locality.

YouTube

Scenarios

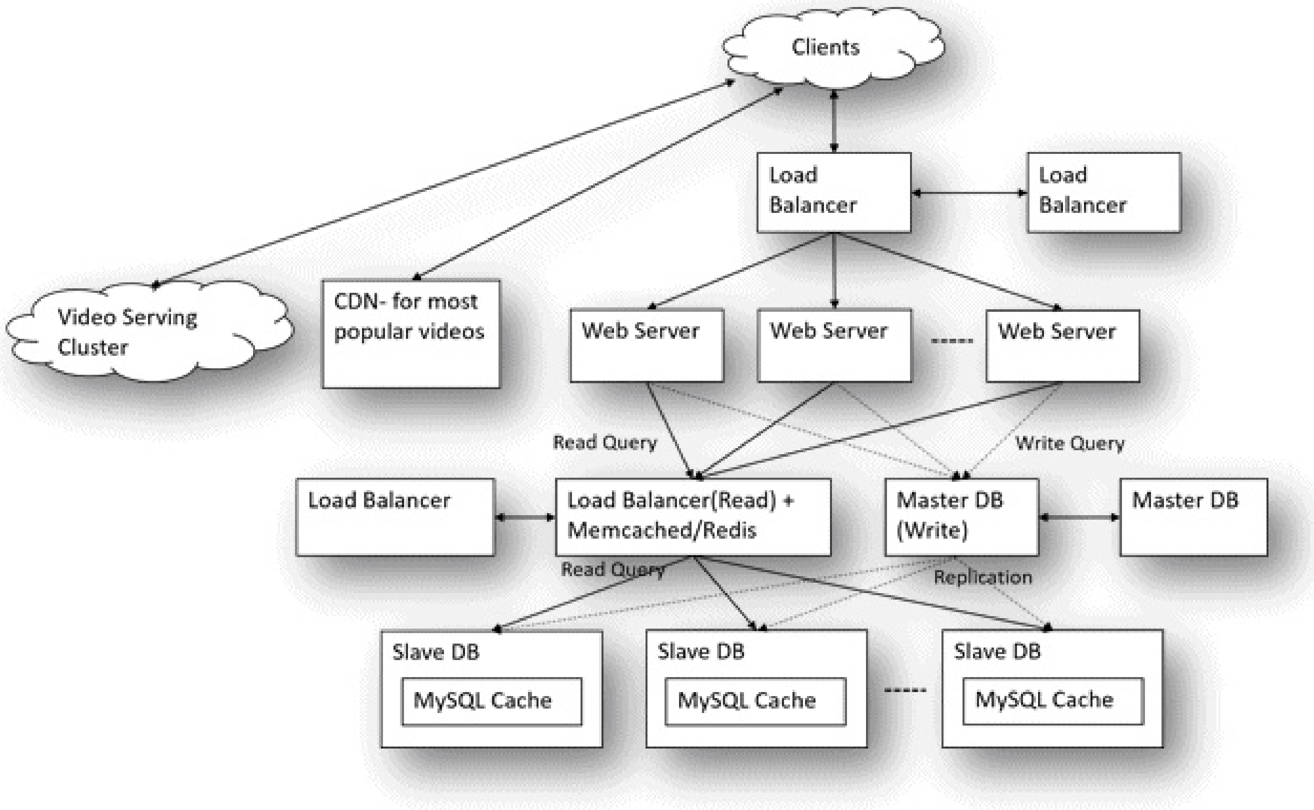
1. Users have some profile according to which content is shown.
2. Content thumbnails are shown when the user opens the YouTube web page.
3. When the user clicks on some thumbnail, then that video is played on flash player.

Constraints

1. Millions of users are going to use this service.
2. 200 million video requests served per day.
3. More reads than writes.

Design

1. YouTube is supposed to serve huge number of videos for which it has video serving clusters. A single video can be served from multiple servers in clusters and from multiple clusters thereby distributing the disk read which increases the performance of the system.
2. The most popular videos are served from CDN. CDN is more close to the user, which reduce the response time. This also reduce the load to the video serving clusters.
3. The rest of the metadata of the video is served from other servers, as the user is not much interested in the metadata.
4. The rest of the application will be same, as it will have an application server, database servers, load balancer, caching etc.
5. There is more read than write so the master server topology will be used. Therefore, there can be a single master for writing and multiple slaves for reading.
6. Master data is replicated to slaves. Since slaves are same as master then the master is down, then slaves can be promoted to make as master.
7. Page to be displayed to the user depends on his subscribed pages, History etc.
8. Information can be cached in the Memcached implemented near the database load balancer.



Clients
Load Balancer
Load Balancer
Video Serving Cluster
CDN- for most popular videos
Web Server
Web Server
Web Server
Read Query
Write Query
Load Balancer
Load Balancer(Read) + Memcached/Redis
Master DB (Write)
Master DB
Read Query
Replication
Slave DB
MySQL Cache
Slave DB
MySQL Cache
Slave DB
MySQL Cache

Design IRCTC

Scenario

1. The user should be able to query trains between two stations.
2. The query should be based on the date, quota, from and to the station.
3. The user should be able to see the availability in the train list retrieved from the above query.
4. The user should be able to book tickets for the available seats.

Constraints

1. There will be a huge number of people requesting the service. Let us suppose 0.01 percent of the population use the service daily once.
2. Geo-Redundancy should be provided.
3. More read query/ request then writes requests.

Design

1. The basic architecture from the scalability theory topic can be used here too.
2. There should be a huge number of servers, which are serving the users.
3. There should be multiple servers at multiple geo locations to provide geo-redundancy. The database should be replicated at these multiple geo locations. There may be multiple servers in one particular zone too.
4. There is a huge number of read query, the user generally does a large number of query to find the seat he wants to book. There will be more reads then write so master-slave.
5. Queries can be cached; little old data is ok.
6. All the search will be served by slave servers.
7. When we book a ticket then transaction goes directly to the master server. Locks on train number can be taken to prevent race conditions. Once a lock is acquired then only you can book a ticket. Some counts can be used to avoid unnecessary locking and some counter can be used for this.
8. Each station has a quota in train and seats are allocated from that quota.
9. Each physical train will have two train ids one when trans go from source to destination station and one when it comes back. So in the system, there will be two train ids. Keeping these two separate ids will make the query easier to implement.
10. When final charting is done then each seat is swapped for the empty slots and we try to find the request from source station whose destination is also in that slot. The first fit is allotted that seat.
11. A load balancer is used to distribute traffic.
12. There may be multiple booking servers which ask for a booking token to the master server. Master server allocate a token for that server and reserve it for some time. When all the user information is filled and payment is done then only it allocates real seat depending upon user preference.
13. Slave server will handle user request till the end. Final booking request with the user payment and his complete information will go to the master server and the corresponding ticket will be booked.

Alarm Clock

How would you design an alarm clock?

Use Case

Alarm clock should have all the functions of clock. Should be able to show time.
The User can set the alarm time
The User can reset the alarm.
The User can set the alarm.

Constrains

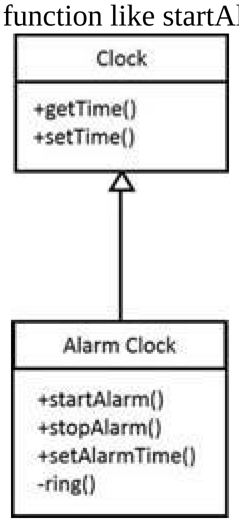
The Granularity of alarm can be 15 min.

Test Case

Set the alarm at some time 6:00AM and set it.
The Alarm should work at 6:00AM
Stop the alarm, then alarm should stop ringing.

Design

There can be a clock class, which manages time and shoe time to the screen. It has functions getTime() and setTime(). Alarm Clock extends Clock, and have some more function like startAlarm(), stopAlarm(), setAlarmTime(), ring()



Implementation

A timer entry will run for granularity of 15 min. Alternatively, one min depends on customer A timer entry will run for granularity of 15 min. Alternatively, one min depends on customer requirement.
It will check if current time is equal to alarm time. If true and the alarm is on then ring.
If start Alarm is called, it will set alarm on
If stop alarm is called, it will set alarm off.

Design for Elevator of a building

Scenarios

A typical lift has buttons (Elevator buttons) inside the cabin to let the user who got the lift to select his/her desired floor. Similarly, each floor has buttons (Floor buttons) to call the lift to go floors above and a floor below respectively. The buttons illuminate indicating the request is accepted. In addition, the button stops illuminating when the lift reaches the requested floor.

Use cases

User

- Presses the floor button to call the lift
- Presses the elevator button to move to the desired floor

Floor Button & Elevator Button

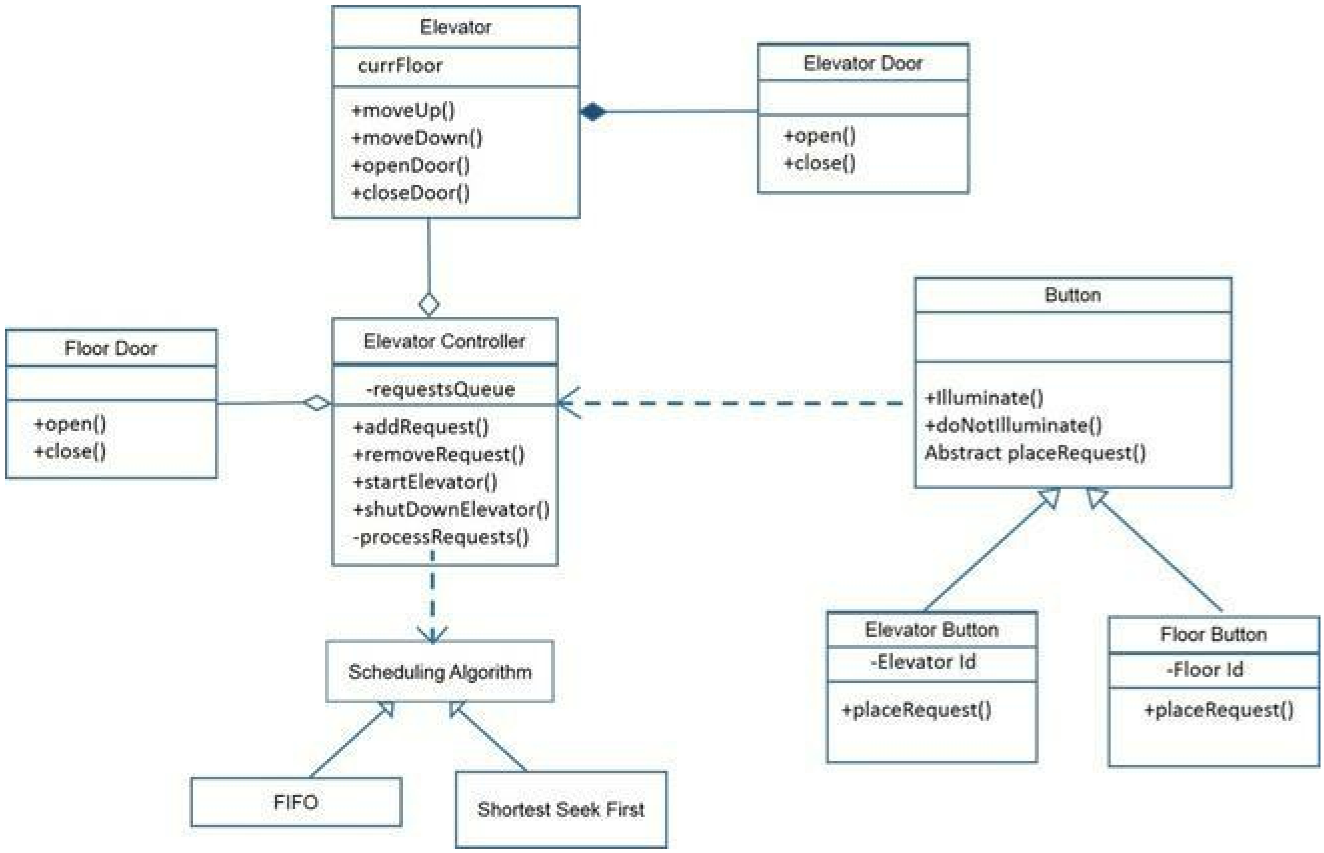
- Illuminates when pressed by user
- Places an elevator request when pressed

Elevator

- Moves up/down as per instruction
- Opens/closes the door

Design

Each button press results in an elevator request which has to be served. Each of these requests is tracked at a centralized place. Elevator Requests, the class that stores, elevator requests can use different algo to schedule the elevator requests. The elevator is managed by a controller class, which we call Elevator Controller. Elevator controller class provide instructions to the elevator. Elevator controller reads the next elevator request to be processed and served.



The button is an abstract class defining common behavior like illuminate, doNotIlluminate. FloorButton, Elevator Button extends Button type and define placeRequest () method which is invoked when a button is pressed. When a floor button or elevator button is presses a requests is added to a common queue.
ElevatorController reads the next request and instruct next action to the elevator.

How can we extend this to multiple elevators?

In the single elevator scenario, there is a single elevator and an elevator controller and a common server where the floor requests and the elevator button request are stored. Which are processed as per the scheduling algorithm.
To extend this to multiple elevator scenarios there will still be single elevator controller. Floor based requests can be served by any elevator whereas elevator button requests will be served only by the elevator to whom the button belongs.

FloorButton's placeRequest() adds a request to the common queue, which is accessed by the elevator controller thereby assigning the request to one of the elevators. ElevatorButton's placeRequest adds a request to the elevator directly as it is supposed to serve it. Elevator controller will be running various algorithms like shortest seek etc. to decide which lift is supposed to handle which request.

Valet parking system

Design a valet parking system.

Use Case

The requirements of the valet parking system should be:
1. Given a Parking lot having a fixed number of slots
2. Where a car can enter the slot if there is a free slot and then it will be given the direction of the free slot.
3. When exiting the car has to pay the fees for the duration of the time it uses parking.

Constraints

1. Parking slots come in multiple sizes- small, mid and large
2. Three types of vehicles, small, mid, large
3. A small vehicle can park in a small, medium, or large spot
4. A medium vehicle can park in a medium or large spot
5. A large vehicle can park only in a large spot

Design & Implementation

The parking lot will have the following interface

```
parkingLot:
    unreservedMap = {};
    reservedMap = {}

    reserveSpace(Space):
    # It will find if there is space in the unreserved
    # map If yes, then we will pick that element and
    # put into the reserved map with the current time
    # value.

    unreserveSpace(Space):
    # It will find the entry in reserve map.
    # If value found then we will pick that Element and
    # put into the unreserved map. And return the
    # charge units with the current time values
```

OO design for a McDonalds shop

Let us start with the description of how the McDonalds shop works.

1. In a McDonalds shop, the Customer selects the burger and directly places the order with the cashier.
2. In a McDonalds shop, the Customer waits for the order ready notification. Customer upon being notified that the order is ready collects the burger himself.

There are three different actors in our scenario and below is the list of actions they do.

Customer

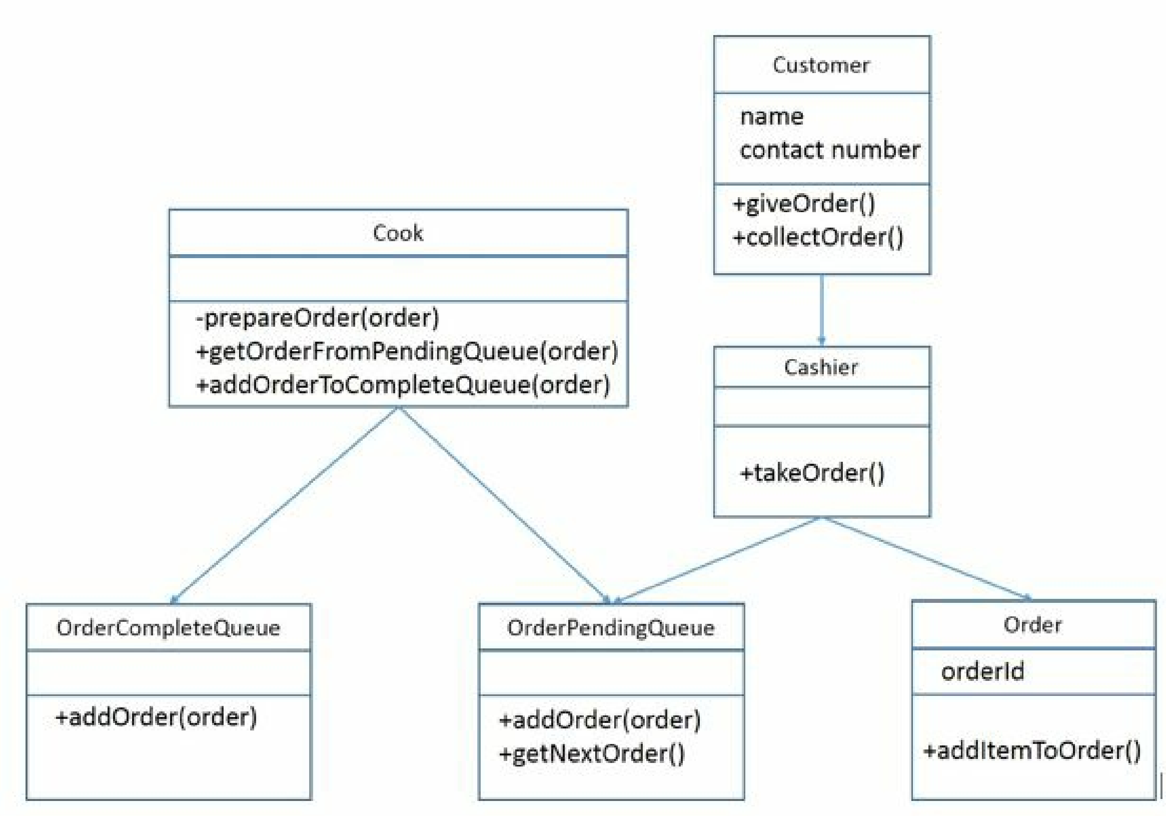
1. Pays the cash to the cashier and places his order, get a token number and receipt
2. Waits for the intimation that order for his token is ready
3. Upon intimation/ notification, he collects the burger and enjoys his drink

Cashier

1. Takes an order and payment from the customer
2. Upon payment, creates an order and places it into the order queue
3. Provide token and receipt to the customer

Cook

1. Gets the next order from the queue
2. Prepares the burger
3. Places the burger in the completed order queue
4. Places a notification that order for token is ready



Object oriented design for a Restaurant

Let us describe how the restaurant works.

1. In a restaurant, the waiter takes order from the customer.
2. The waiter waits for the order to be ready and once ready serves the dishes to the customer.

These are the different actors in the model and I have listed the different actions against each actor

Customer

1. Selects the dish from the menu and call upon a waiter
2. Places the order
3. Enjoys his meal once the dish is served on his plate
4. Ask for the bill
5. Pays for the services

Waiter

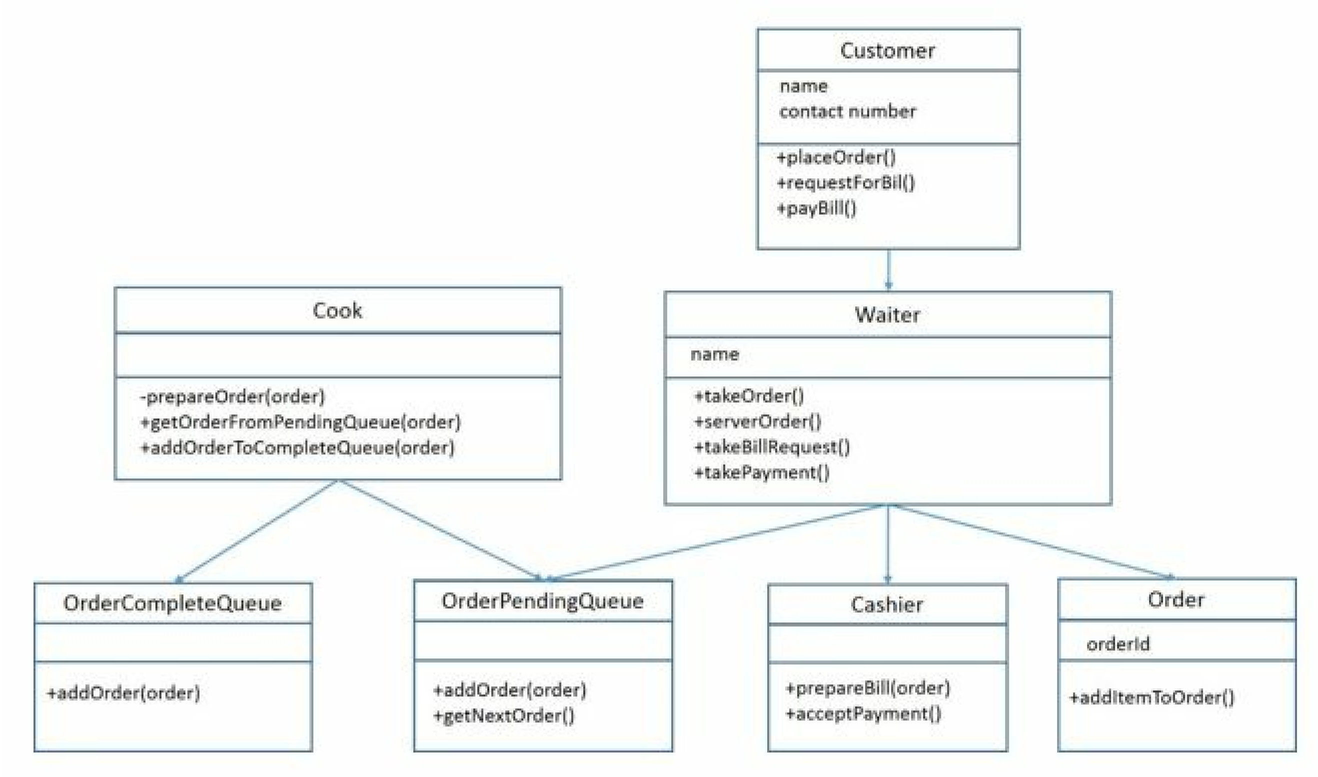
1. Responds to the customers call on the tables he is waiting
2. Takes the customer's order
3. Places the order in the pending order queue
4. Waits for the order ready notifications
5. Once notification is received, collects the dish and serves the dish to the corresponding customer
6. Receives the bill request from customer
7. Asks the Cashier to prepare the bill
8. Gives the bill to the customer and accepts the payment

Cashier

1. Accepts the prepared bill request from the waiter for the given order details
2. Prepares the bills and hands it over to the waiter
3. Accepts the cash from the waiter towards the order

Cook

1. Gets the next order from the pending order queue
2. Prepares the dish and push the order to finished order queue
3. Sends a notification that the order is ready



Class diagram for the Restaurant.

Object oriented design for a Library system

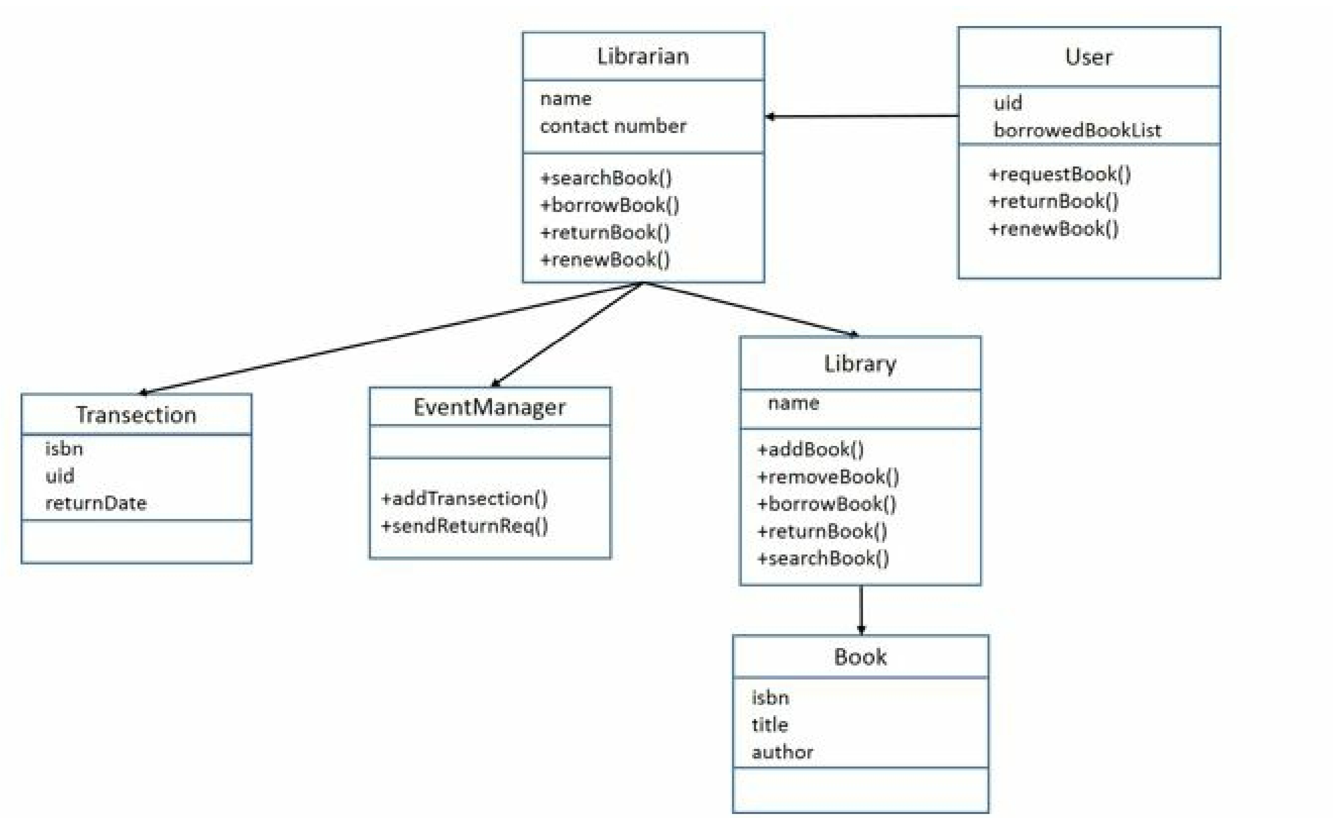
A library has a set of books, which the users can borrow for a certain period and return. Users may choose to renew the return date if they feel they need more time to read the book.

The typical user actions with this online library would be
1. Sign in/register
2. Search books
3. Borrow books
4. Renew books
5. Return books
6. View his profile

The online library must keep track of the different books in the library currently available for users to borrow and the books already borrowed by users. Put it simply the inventory should be managed.

The various components of the system:
1. User
2. Librarian
3. Library
4. Book
5. Transection
6. Event Manager

The below class diagram, which depicts how these components inter-operates.



The User either interacts with the Librarian, the user request, return or renews a book. The Librarian will search for the book if the book is available in the Library then issue it to the user. A Transection will be created and added to the Event Manager. Event Manager will support add transaction and send return request interface. Once the book is overdue then the event manager will send an indication to the student that the book needs to be returned. When the book is renewed then the library state is not changed but the Transection detail is renewed at the Event Manager.

Suggest a shortest path

Use Case

The user had some coordinate by searching the coordinate from the name.
Show the whole map considering the coordinate as its center.
Suggest the shortest path between two points.

Constraints

All paths are positive in cost. For simplicity, I am considering all paths are for vehicle only, no pedestrian (pedestrian can walk in either direction even in one-way road.)

Design

The whole city map is stored as a graph in google.
We need to find the map by looking into the objects, which are in the distance shown by the browser.
The same path is stored as directed graph. Moreover, the graph that needs to be rendered depends on the zoom level. The preferred algorithm is a* for this application to get the shortest path.
Weight = h(x, y) + g(x, y)

Exercise

1. Design a system to implement social networking like Facebook, with millions of users. How would you find the connection between two people?

2. Autocomplete in www.booking.com. Design autocomplete feature for www.booking.com.

3. Instagram, Instagram is an online mobile-based photo sharing, video sharing service, which enables users to take pictures, and video upload them to the server and share them on social networking sites like Facebook or Twitter.

4. Monolithic Website, assume you have a monolithic website and you are asked to architect the website
   Hint: - Discuss whole scalability theory section here.

5. Trip Advisor: URL's are parsed; content is collected from various services, and then applied to a template.

6. Cinchcast: Live audio streaming for business to do conferences.

7. BlogTalkRadio : Audio social network

8. Client based recommendation feature: How would you design a client based recommendation feature (based on customer history) on the product detail page? Design Customers who viewed item A also view item B and item C in an online shopping portal.

9. Car renting system: Design a car renting system, including reserving a car, checking in and checking out. Consider all the cases: reserve a car, then check out successfully; reserve a car, but the car is sold out before you check out.
   <u>Test Cases:</u>
   a) Try to reserve a car for more than one person
   b) Try to reserve a car that is sold out
   c) Verify the checkout process. After checking out a particular, you should be able to reserve it for another customer.
   d) Try to reserve the same car for different customers in different dates

10. Online cab booking system (like Uber)
    <u>Admin Module</u>
    a) Admin should be able to add new driver / taxi details.
    b) Should be able to calculate the amount that needs to be paid to the drivers. Monthly, weekly or daily.

    <u>User Module</u>
    a) Should be able to choose from and to location.
    b) Available Taxies type, along with fare details.
    c) Select a Taxi type
    d) Book the taxi.
    e) A confirmation message for the booking.

    <u>Driver Module</u>
    a) A driver should be able to register as a driver to Uber.
    b) When a job is displayed to the driver, he should be able to accept the job.
    c) When the driver reaches to the customer then he should be able to start a trip.
    d) When the driver had taken the customer to the desired location then he should stop the trip.
    e) The driver should collect the fare based on the amount displayed in the app.
    f) The driver should be able to give customer feedback.

Note: Just assume 2 minutes is equal to 1 KM.

11. Online teaching system
    a) In an online teaching system, there are n number of teachers and each one teaches only one subject to any number of students.
    b) A student can join to any number of teachers to learn those subjects.
    c) Each student can give one preference through which he can get updates about the subject or class timings etc.
    d) Those preferences can be through SMS or Twitter/Facebook or Email etc.
    e) Design above system and draw the diagram for above.
12. Design an online Auction system (similar to e-bay). Functionalities include enlisting a product for auction by bid owner, placing the bid for a product by bidders, Bid winner selection, Notification of bid winner etc.).

13. Customer Order Booking System

    <u>Admin Module</u>
    a) Should be able to add/edit/delete item, along with quantity, price, and unit.
    b) Should be able to see all orders.

    <u>Customer Module</u>
    a) Should be able to enter his/her details for shipping, along will basic information like name, email, contact etc.
    b) Can choose item, quantity
    c) Automatically payable price should be generated as per selected item and quantity.
    d) Should be able to confirm the order.
    e) After confirmation can see order confirmation report along with order number, which will be, system generated.

14. Online Movie Booking System

    <u>Admin Module</u>
    a) Should be able to enter all movies, which have been released, and about to release in next week with all possible details like theatre location, price, show timings and seats.
    b) Should be able to delete movies, which are no longer in the theatre.
    c) Can see a number of booked tickets and remaining tickets for single theatre or for all theatre.

    <u>User Module</u>
    a) User should be able to check all ongoing movies in theatre along with locations, availability of seats, price, and show timings
    b) The user should be able to check all upcoming movies for next week too.
    c) All movies those are running on theatre should be available for booking (one ticket or more than one ticket can be booked).
    d) After booking user should see the confirmation message of booking.

APPENDIX

Appendix A

Algorithms	Time Complexity
Binary Search in a sorted array of N elements	O(logN)
Reversing a string of N elements	O(N)
Linear search in an unsorted array of N elements	O(N)
Compare two strings with lengths L1 and L2	O(min(L1, L2))
Computing the Nth Fibonacci number using dynamic programming	O(N)
Checking if a string of N characters is a palindrome	O(N)
Finding a string in another string using the Aho-Corasick algorithm	O(N)
Sorting an array of N elements using Merge-Sort/Quick-Sort/Heap-Sort	O(N * LogN)
Sorting an array of N elements using Bubble-Sort	O(N!)
Two nested loops from 1 to N	O(N!)
The Knapsack problem of N elements with capacity M	O(N * M)
Finding a string in another string – the naive approach	O(L1 * L2)
Three nested loops from 1 to N	$O(N^3)$
Twenty-eight nested loops … you get the idea	$O(N^{28})$
Stack	
Adding a value to the top of a stack	O(1)
Removing the value at the top of a stack	O(1)
Reversing a stack	O(N)
Queue	
Adding a value to end of the queue	O(1)
Removing the value at the front of the queue	O(1)
Reversing a queue	O(N)
Heap	
Adding a value to the heap	O(logN)
Removing the value at the top of the heap	O(logN)
Hash	
Adding a value to a hash	O(1)
Checking if a value is in a hash	O(1)