The Concept of Ownership in Rust and Swift

The Concept of Ownership in Rust and Swift

By
Elaf A Alhazmi, M.Sc.

A Thesis
Submitted to The Department of Computing and Software
and the School of Graduate Studies
of McMaster University
in Partial Fulfillment of the Requirements
for the Degree
Master of Computer Science

McMaster University
©Copyright by Elaf A Alhazmi, August 2018
All Rights Reserved

MASTER OF COMPUTER SCIENCE (2018) McMaster University
COMPUTING AND SOFTWARE Hamilton, Ontario, Canada

TITLE: The Concept of Ownership in Rust and Swift

AUTHOR: Elaf A Alhazmi,
B.Sc (Computer Science)
King Abdulaziz University
Kingdom of Saudi Arabia

SUPERVISOR(s): Dr. Emil Sekerinski and Dr. Frantisek Franek

NUMBER OF PAGES: x, 206

To my beautiful wise Mother

Huda

To my strong gentle Father

Abdulrahman

To the greatest persevered

Elaf

Abstract

There is a great number of programming languages and they follow various paradigms such as imperative, functional, or logic programming among the major ones and various methodologies such as structured approach or object-oriented or object-centered approach. A memory management design of a programming language is one of the most important features to help facilitate reliable design. There are two wide-spread approaches in memory management: manual memory management and automatic memory management, known as Garbage Collection (GC for short). Recently, a third approach to memory management called *Ownership* has emerged. Ownership management is adapted in two recent languages Rust and Swift. Rust follows a deterministic syntax-driven memory management depending on static ownership rules implemented and enforced by the `rustc` compiler. Though the Rust approach eliminates to a high degree memory problems such as *memory leak*, *dangling pointer* and *use after free*, it has a steep learning costs. Swift also implements ownership concept in *Automatic Reference Counting* ARC. Though the ownership concept is adapted in Swift, it is not a memory safe language because of possibility of strong reference cycles. In this research, a demonstration of the ownership in Rust and Swift will be discussed and illustrated, followed by analysis of the consequences of memory issues related to each design. The comparison of Rust and Swift is based on *ownership*, *memory safety*, *usability* and *programming paradigm* in each language. As an illustration, an experiment to compare the elapsed times of two different structures and their algorithms, Binary Tree and Array are presented. The results illustrate and compare the performances of the same programs written in Rust, Swift, C, C++, and Java.

Acknowledgements

First and above all, I praise God (Allah) always and forever in allowing me to complete this research.

I would like to acknowledge several people without whom this thesis would not have been at all possible. I am grateful to so many for their guidance and support. I would like to offer my sincere gratitude and thanks to my supervisors Dr. Emil Sekerinski and Dr. Frantisek Franek for their enthusiasm, motivation, patience and immense knowledge in supervising me. Especially, their guidance and support helped me to clarify my thinking for this research and move forward.

I have very special thanks to my brother Adel who shared with me these years in Canada and discovered with me the happiness and burdens of adapting to a new country and lifestyle. I want also to express my gratitude and deepest appreciation to my siblings Ahoud, Emtenan and my lovely youngest brother Mohammad, who always prayed for my success. Similarly , I would like to thank my brother in law Abdulwahab for his guidance, advice and encouragement in completing this journey.

To my best friend, Harika, I sincerely thank you – your kindness and caring made my journey more wonderful. Finally, but by no means least, to my mother Huda and father Abdulrahman, I sincerely thank you – your love and praying always show me the impossible could be possible.

Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.2 The Purpose of the Research . . . 2

1.3 The Contributions of the Research . . . 3

1.4 Thesis Structure . . . 3

2 Literature Review **5**

3 Rust **10**

3.1 The Rust Programming Language . . . 11

3.2 Rust Memory Essentials . . . 11

3.2.1 Fixed and Dynamic Size Types . . . 15

3.2.2 References and Pointers . . . 17

3.3 The Rust Compiler `rustc` and the Ownership system . . . 21

3.3.1 Ownership . . . 21

3.3.2 Borrowing . . . 23

3.3.3 Borrow Checker . . . 25

3.3.4 Scopes . . . 32

3.3.5 Lifetimes . . . 34

3.4 Analysis of the Ownership System . . . 38

4 Swift **40**

4.1 Swift Programming Language 41

4.2 Swift Type Essentials 41

4.2.1 Value, Reference and Optional Types 41

4.3 Swift ARC and Ownership 43

4.3.1 MRR – ARC 43

4.3.2 Strong Reference Ownership 44

4.4 ARC and Memory Issues 46

4.4.1 Strong Reference Cycle 46

4.5 Lifetime Qualifiers 49

4.5.1 `Weak` Lifetime 49

4.5.2 `Unowned` Lifetime 52

4.5.3 Strong Dependent Lifetime 56

4.6 Analysis of Ownership in ARC 59

5 Comparison Between Swift and Rust **61**

5.1 Similarities and Differences 61

5.1.1 Ownership and Memory Safety 61

5.1.2 Usability 64

5.1.3 Programming Paradigms 65

6 Run-Time Experiment **66**

6.1 Methodology 66

6.1.1 The Targeted Feature 66

6.1.2 Languages Selection 67

6.1.3 Tasks Selection 67

6.2 The Experiment Setting 67

6.3 Permutations of a sequence of Numbers 68

6.3.1 Loop – Generating Permutations 69
6.3.2 Shuffle – Fisher-Yates algorithm 71
6.4 Binary Tree 72
6.4.1 Insert 73
6.4.2 Replace 78
6.4.3 Search 83
6.5 Array 85
6.5.1 Quick Sort 85
6.5.2 Binary Search 86
6.6 Discussion 88

7 Conclusion and Future Work 89

A Appendix 97
A.1 Array – Source Code 97
A.2 Tree - Source Codes 132
A.3 Rust - Rc<T> and RefCell<T> cyclic issue 200
A.4 Rust- smart pointers composition 202

List of Tables

6.1 Generating Permutations – Loop 69
6.2 Optimized – Loop 70
6.3 Shuffle 71
6.4 Optimized Shuffle 72
6.5 Insert 1,000,000 nodes – Binary Tree 73
6.6 Optimized Insert 1,000,000 nodes – Binary Tree 74
6.7 Average Insert 10,000,000 nodes – Binary Tree 74
6.8 Max Insert 10,000,000 nodes – Binary Tree 75
6.9 Min Insert 10,000,000 nodes – Binary Tree 75
6.10 Insert 50,000,000 nodes – Binary Tree (Minutes) 76
6.11 Avg Insert 100,000,000 nodes – Binary Tree 77
6.12 Avg Replace 10,000,000 – Binary Tree 79
6.13 Max Replace 10,000,000 – Binary Tree 80
6.14 Min Replace 10,000,000 – Binary Tree 81
6.15 Replace 50,000,000 nodes – Binary Tree (Minutes) 82
6.16 Avg Replace 100,000,000 nodes – Binary Tree 82
6.17 Search – Binary Tree 84
6.18 Optimized Search – Binary Tree 84
6.19 Quick Sort 85
6.20 Optimized Quick Sort 86

6.21 Binary Search – Array 87
6.22 Optimized Binary Search – Array 87

List of Figures

4.1 Weak reference – Customer Car objects 52

4.2 Unowned reference – Customer Phone objects 56

4.3 Unowned with Unwrap (!) operator – Building Room objects 58

6.1 Generating Permutations – Loop 70

6.2 Optimized – Loop 70

6.3 Shuffle 71

6.4 Optimized Shuffle 72

6.5 Insert 1,000,000 nodes – Binary Tree 73

6.6 Optimized Insert 1,000,000 – Binary Tree 74

6.7 Avg Insert 10,000,000 – Binary Tree 75

6.8 Max Insert 10,000,000 – Binary Tree 75

6.9 Min Insert 10,000,000 – Binary Tree 76

6.10 Insert 50,000,000 – Binary Tree (Minutes) 77

6.11 Insert 100,000,000 – Binary Tree 77

6.12 Binary Tree – random example 78

6.13 Binary Tree – selected random node 78

6.14 Binary Tree – recreate and replace 79

6.15 Avg Replace 10,000,000 – Binary Tree 79

6.16 Max Replace 10,000,000 – Binary Tree 80

6.17 Min Replace 10,000,000 – Binary Tree 81

6.18 Replace 50,000,000 – Binary Tree (Minutes) 82
6.19 Replace 100,000,000 – Binary Tree 83
6.20 Search – Binary Tree 84
6.21 Optimized Search – Binary Tree 84
6.22 Quick Sort 85
6.23 Optimized Quick Sort 86
6.24 Binary Search – Array 87
6.25 Optimized Binary Search – Array 87

Chapter 1

Introduction

Various features and capabilities of programming languages make programmers focusing more on reliability of coding [1]. Even though some programmers probably choose the most suitable language providing automatic management of resources instead of taking the risk of manual management; others prefer to use the conventional programming languages that have been used for decades since the experience of using these languages is robust and rich enough to control language issues manually. However; reliability of memory management is required during system design in order not only for the safety of the programs, but also to avoid serious issues during execution.

Memory management – the allocation and deallocation of dynamic memory during the execution of a program – is one of the most important aspects of programming languages. How memory resources are created, manged, and deallocated are issues requiring the programmers' attention. Two recent programming languages, Rust and Swift introduced a distinct language design for memory management based on the ownership concept. The ownership concept can not only be adapted for managing memory, it can be used for managing all kind of resources. The realizations of the ownership concept in both languages require new cognitive skills on the side of the programmer and hence incur a steep learning curve.

1.1 Motivation

Beside the object-oriented features, high order functions, and concurrency features in these languages, to start programming in Swift and Rust is distinct from other languages due to the influence of ownership rules. Programmers typically start programming as if they were using C, C++, or Java, but after finding many errors even in simple programs such as calculator, they will start realizing how strong the ownership techniques are and that they need to be understood precisely.

Introducing the details of ownership concept in Rust and Swift, we contribute to learning of these new languages by presenting their rules and techniques on simple to complex data structures. Besides introducing the rules of ownership in these languages, memory issues such as memory leaking (when an allocated memory is not reclaimed even though it is not needed anymore) and dangling pointers (when a pointer references an object that had been meanwhile deallocated) will be discussed and analyzed.

1.2 The Purpose of the Research

What is the best programming language? Most developers would not be able to give a uniform answer because each programming language has its own pros and cons. Also, the answers will differ based on many perspectives. For example, from a software engineering perspective, the simplicity and the performance could be the most important properties. On the other hand, from a programmer perspective, it could be the easiness of maintaining the software and fixing the errors.

The most critical property that impacts any new programming language is memory management. As mentioned previously, the ownership concept provides more efficient support to overcome the limitation of the conventional styles. Rust and Swift are two recent programming languages that support this concept. Our aim is to investigate their approaches to the ownership concept. In addition, this research examines the related memory management issues and compare it with some classical programming languages.

1.3 The Contributions of the Research

We believe that our efforts are significant. To the best of our knowledge, there is no other research comparing Rust and Swift with respect of memory management.

This thesis documents several key contributions made to the field of evaluating both Swift and Rust languages. Our contributions are as follow:

- We investigate the concept of ownership in Swift and Rust.
- We present the similarities and differences between these two languages based on ownership, memory safety, usability, and programming paradigms.
- We explore and present the memory issues in the concept of ownership management comparing to the management of conventional style: Manual and automatic memory management.
- We conduct experiments that compare these languages with the other three classical languages that support conventional styles.

1.4 Thesis Structure

This research focuses on the concept of ownership in Rust and Swift by discussing it in the following chapters. **Chapter 2** is a literature review of the field and focuses on the programming languages and memory management designs. Several programming languages are discussed in terms of performance and safety; Rust and Swift are relatively recent languages contributing new ideas to the field. **Chapter 3** illustrates the concept of ownership in Rust and introduces `rustc` compiler rules known as the ownership rules. Rust is a language designed and developed by Mozilla. In **Chapter 4**, Swift is introduced. It is a programming language supported by Apple and the transition of MRR, (ManualRetainRelease) to ARC, (Automatic Reference Counting) will be presented. Beside that, the concept of ownership in Swift will be discussed with the consequences of adapting ARC management. Then comparing Rust and Swift by presenting the similarities and

difference of memory management is discussed in **Chapter 5**.The similarities and difference will be summarized based on *ownership*, *memory safety*, *usability* and *paradigms*. **Chapter 6** presents experiments of performance evaluation of different algorithms and data structures – Array and Binary Tree – in five different programming languages: Java, C, C++ , Rust, and Swift. In addition, the performance and the consequences of using ownership in Swift and Rust are discussed. Finally, **Chapter 7** presents the conclusion and charts the future work.

Chapter 2

Literature Review

There have been many programming languages designed for creating applications. Some of these languages have vanished and some of them are still in use in industry, business, and academia. New specialized languages occur all the time, but for general programming languages it is a rare event. Thus the emergence of new general programming languages such as Swift and Rust that tried to overcome memory issues of the older programming languages attract interest of the academia as well the industry. Therefore, it became necessary to study and compare these emerging programming languages in order to understand their contributions. In general, there are many factors for evaluation of programming languages such as readability, writability, and reliability [1]. Furthermore, memory management and speed of execution are most important criteria for designing any complex software. This chapter is focused on memory management styles of various high-level programming languages. It is obvious, that all programs of non-trivial complexity need to allocate some space in memory to store data values and data structures, and deallocate the memory when no longer needed.

A number of authors have evaluated several programming languages based on many features. The following works are considered the most relevant to our efforts. The authors of [2] compare the graphics rendering performance, memory efficiency, and computational speed of three high-level languages: C, C++, and Java. Their results demonstrate that Java uses on average 2.5 to 3 times more memory than C/C++. Also, it is slower for graphics image animation as well as using a pure

numerical benchmark that simulated image pixel operations. However, previous studies [2] cannot be considered as conclusive because they only compared two programming paradigms: procedural and object-oriented.

Currently, the most common and hence most important programming paradigms are procedural, object-oriented, functional, and scripting. Some researchers classify programming languages based on the programming paradigm they use: for example, C# and Java are considered object-oriented languages, F# and Haskell are functional languages, while Python and Ruby are scripting languages, and C is one of languages classified as a procedural language [3]. However some programming language employ multiple paradigms. Such is the programming language Swift which exhibits aspects of object-oriented, imperative, and functional paradigms [4].

The study by the authors of [3] is more efficient as it is comparing more programming paradigms. They use also two programming languages representing each paradigm. They show how efficiently the studied programming languages use memory. They found C and Go languages to be more efficient than the other languages. The conclusion can be summarized as that procedural languages use less memory than language using other paradigms, however, there are no significant differences among languages in the same category.

The authors of [5] did an empirical comparison of seven languages: C, C++, Java, Perl, Python, Rexx, and Tcl. Their study was based on investigating several properties such as memory consumption, runtime, reliability and other. In terms of memory consumption, the result shows the memory consumption of a script program is about twice that of a C or C++ program.

There have been numerous studies showing the procedural languages such as C make the most economical usage of memory. However, the consequences of the procedural languages concerning the security of the data and the difficulties of relating data with real objects were mentioned by researchers in [6, 7]. Such limitations of procedural languages sparked the efforts to look for new

methods and approaches, in particular concerning the memory management.

There are two fundamental approaches to memory management for the majority of programming languages. The first approach is the manual or explicit memory management method which requires programmers to allocate and free memory manually. The second approach is automatic memory management which emerged in order to overcome the issues of the first type approach[8]. For instance, a programmer who uses the C language should implement memory management for each application manually. In contrast, a programmer who uses modern programming languages such as Java, C#, and Caml is not obliged to manage the memory manually because these languages offer the capability of managing the memory automatically via garbage collection.

Undoubtedly, the benefits of the automatic memory management are indisputable. However, the costs are high in runtime and space efficiency [2, 3]. The authors of [9] emphasized that object-oriented programming significantly increases dynamic memory usage, thereby causing overhead in terms of memory, performance, and power. Due to various reasons, there are only a handful of documented bad experiences of object-oriented technology. Also, the authors of [10] emphasized that the garbage collection will improve in the future.

Nowadays, there are several new programming languages designed with the aim to improve the weaknesses of the previous programming languages. In our research we study two recent languages, Rust and Swift. Rust was developed by Mozilla [11] and it was designed to overcome two obstacles in the previous languages to offer programmers both high-level safety and low-level control [12]. Swift was developed by Apple and programmers can use it since about 2014 [13]. In comparison to Rust, this language is easier to use for new programmers in in many ways it is also more flexible. In particular, Swift is easier to learn and use for a novice programmer than the previous languages such as C# and Java. In addition, Swift is designed to be faster and safer than Objective-C [14] that inspired it.

As mentioned previously, evaluating any new programming languages is a significant task. That is why it is a useful contribution to compare the existing programming languages. There are several studies that compare Swift and Rust with the existing mainstream languages in terms of various properties.

Comparing Rust with other languages, the research [15] discusses the benefits of Rust. They found Rust promising both a better performance and safety in comparison to C-like languages. Also, their results show Rust's safety features do not create significant barriers to implementing a high performance collector. In terms of Rust's safety, the authors of [16] demonstrated how Rust utilizes the ownership-based type system. However, this core type system uses libraries that internally use unsafe features.

Furthermore, the authors of [17] investigate the strengths of Rust in comparison with conventional languages such as C. They found Rust has ability to implemented powerful security and reliability mechanisms like Software fault isolation (SFI), Information flow control (IFC), and Automatic check pointing more efficiently than C. However, that is not a sufficient reason to enforce using Rust and abandon other languages such as C. The efficiency and performance of the current languages like C encourage use of these languages[18].

Regarding to conventional languages' weakness like lack of memory safety and integrated support for concurrency, these languages force developers to deal with memory and thread safety [19] . There are attempts for solving these problems by designing some methods in languages. For example, the authors of [20] present a static type system called *Real-Time Specification for Java* (RTSJ) in order to ensure memory safety. Their work was the first work that combined the benefits of region types and ownership types. However, in our search we do not focus on studying one language and how researches try to solve it weakness.

Comparing Swift with other languages, the authors of [4] compare Swift (as multi paradigm language) with Objective-C (as object-oriented paradigm). In their study they explore the difference between theses languages only for some specific properties. They found Swift to be easier, faster, and more secure fro program development than Objective-C. As a result, programmers need fewer characters in order to create the same code than Objective-C because of its simplified syntax. Furthermore, the authors of [14] emphasize that Swift's syntax is simpler because it does not use pointers and includes improvements in its data structures and in its syntax. However, the study [4] compares the two languages that were created by Apple. They did not consider other mainstream languages.

To the best of our knowledge, no prior studies have compared both of these two languages Swift and Rust for any evaluation criteria. For that, in our research, more specific research questions will be introduced and investigated in terms of memory management of these two languages comparing the other existing languages. In other words, we want to investigate the impact of the three concept in memory management which are ownership, garbage collectors, and manual memory management.

Chapter 3

Rust

Rust programming language is a system programming language developed by Mozilla research with help of more than 900 members of the community [21]. It is a compiled language that satisfies three major goals: safety, speed, and concurrency. It is a strongly statically typed language where types are checked at compile time. Most of memory issues and problems such as dangling pointers, buffer overflows, null pointers, segmentation faults, and data races are successfully addressed. Rust eliminates two aspects existing in other programming languages. First, Rust satisfies deterministic memory management without garbage collection by using the `rustc` compiler that has strict rules of checking of the ownership. Despite the need for additional analysis at compile time, Rust compiler satisfies it with a minimum run-time overhead, in simple terms, `rustc` is still a fast compiler despite additional analysis. The second primary facet of Rust is the elimination of data races; in other words, Rust eliminates the corruption of shared data through concurrent access .

Rust is a general purpose and multi-paradigm programming language exhibiting many aspects of imperative, structured, object-oriented, and functional languages with advanced techniques for concurrent programming. It must also be noted that Rust not only has a high safety because of its strong type system, but also has a low level control of resources comparable to C and C++. Like Swift discussed in the next chapter, Rust has an inference-type system to determine the type of variable from its initialization [22]. The compiler `rustc` is self hosted which means it is written in

Rust and can compile itself by using a previous version, where the first stable production version 1.0.0 dates to May 2015. Rust is portable to a wide variety of hardware and software platforms such as Linux, Mac OS X, Windows, FreeBSD, Android, and iOS with the LLVM compiler as the back-end.

3.1 The Rust Programming Language

In this section, we introduce the Rust programming language focusing on the essential data types. The rules of `rustc` compiler are discussed in details including a discussion on how novice programmers could struggle with the borrow checker that is one of the Rust compiler subsystem. Complex data types will be introduced to illustrate how more advanced programming code could be developed with the ownership rules taken into consideration. We explain why the difference between fixed and dynamic sized allocation in Rust is highly important for understanding of what behaviour is allowed in each case and the consecutive compiler rules that should be strictly followed.

The concept of ownership is not a unique principle in programming languages, however the feature that when a variable lifetime ends – it goes out of scope – the resource memory is freed is unique to Rust. In addition, memory safety and ownership rules are introduced and illustrated how they work simultaneously and coherently.

3.2 Rust Memory Essentials

In the Rust documentation [23], it is mentioned that there is a learning cost for learning a new programming language. Learning the difficult concept of borrow checker referred colloquially as “fighting with the borrow checker" is something common for Rust novices because `rustc` has many compiler rules related to ownership, borrowing, lifetime and mutability. Rust not only introduced a new programming language with new features and rules, but also a language requiring new cognitive programming skills.

One of the important concepts concerning the memory safety in Rust is *mutability* or *immutability* of a value. By default, all variables (and that includes references we will discuss later) are immutable meaning the value “stored” in the variable is immutable, i.e., cannot be changed. *Shadowing* a variable is an important concept often confused with mutability; it is using the name of already declared variable in another declaration which allows the value to be changed as well as the type of the variable. Thus the following code is legal, and though it seems that the variable x changed its value and hence is mutable, the additional (shadowed) declaration redefines the value of the variable and in between the declarations the variable is perfectly immutable.

```
fn main() {
// outer scope
let x = 10;
   {      // inner scope starts here
      let x = 12;
      println!("inner x = {}",x);
      let x = "McMaster";
      println!(" x now is {}", x);
      // inner scope end here
   }
   println!("outer x = {}",x);
//outer scope ends here
}
```

The output is

```
inner x = 12
x now is McMaster
outer x = 10
```

The above code also illustrates the concept of *scope* in Rust that we will discuss later. In the inner scope, the value and type of `x` (i.e., its binding) is redefined from integer 12 to string `"McMaster"` and it is alright. The `x` declared in the outer scope retains its own binding (integer 10).

The scope and ownership play an important part in preserving the lifetime of any binding. Simply stated, Rust frees a memory resource when a binding goes out of scope. The outer `x` has a longer lifetime than the inner `x`. Note that in Rust, often a *variable* is referred to as a *binding*, this being a more precise term. In the above code, there seems to be one variable name `x` with three different bindings. But in fact there are only two variables, the outer `x` with one binding, and the inner `x` with two different bindings.

In programming languages such as C and Java, the following statement `x=y=8;` would be acceptable in order to initialize `x` and `y` with the value 8. However, such a statement is not acceptable in Rust.

```
fn main() {
    let x = y = 8;
}
```

The compilation causes an error with the following message: `error[E0425]: cannot find value 'y' in this scope`. Why? The keyword `let` declares the variable `x` and its binding should be changed to the value and type of the right hand side. For that, the value and type of `y` is needed, but no `y` is declared in this scope.

Even if `y` were declared in some outer scope, the compiler might reject `y = 8` part if `y` was declared there immutable. It would be acceptable only of `y` was declared mutable in some outer scope. Which brings us to mutable variables declared by the reserved phrase `let mut`.

```
fn main() {
    let mut x = 6;
    let mut y = 8;
    let zero = 0;

    let x = y = zero;
    println!(" x = {:?}, y = {:?}, zero = {:?}", x, y , zero);
}
```

This code will be executed and the following output will be printed, where `()` signifies the *empty value* also known as the *unit value*.

```
x = (), y = 0, zero = 0
```

It should be noted that the *unit value* is not the *null value* which is not allowed in Rust for memory safety reasons. Even though null values are routinely used in many linked data structures such as Binary Tree or Linked List and various algorithms in other languages, Rust does not support it and one of the *enum* types defined in the standard library `std` is used instead.

A common error `error[E0384]: re-assignment of immutable variable ‘x‘` of novice programmers is when they violate the reassignment of an immutable variable as shown in the following example.

```
fn main() {
     let x = 2018;
     x += 1;  // error
     println!("x = {}", x);
}
```

Note: the commonly used operators increment ++ and decrement −− are not supported in Rust.

3.2.1 Fixed and Dynamic Size Types

Besides mutability, there is the question of the size of a binding for a variable: It is a fixed size binding or dynamic size binding. They decide where the memory allocation takes place. In most cases the fixed size binding is allocated on the stack, while the dynamic size binding is always allocated on the heap.

The behaviour of fixed and dynamic size data types are different. Each binding has exactly one single owner that controls its lifetime and the ownership ends when the owner goes out of scope. For dynamic size data types, when the owner goes out of scope, the memory on the heap is freed. The following code describes the *move semantic* action and its consequences of violating the compiler rules.

```
fn main() {
    let x = vec![1,2,3];
    let y = x;
    println!("x[0] = {}", x[0]);  // error
    println!("y[0] = {}", y[0]);
}
```

The code will cause the compile-time error `error[E0382]: use of moved value: 'x'`. This error illustrates one of the Rust strongest rules. To explain the previous code in details, `x` elements will be allocated contiguously on the heap and `x` will be allocated on the stack as a pointer holding the address of the heap elements. When `x` is assigned to `y`, the bitwise copy will copy the pointer `x` into the stack allocation represented by `y`, and this shallow copying will not copy the heap elements. One of significant Rust rules is to prevent data race by preventing two bindings pointing to the same memory resource. As a result, *move semantics* which means moving of the ownership

of x to y will be executed by the compiler and any attempts to use the previous owner, i.e., x, will cause compile error.

The previous example shows the behaviour of a dynamic size type which will not allow two owners pointing to the same address, but the behaviour of a fixed size type will be different – Array is a fixed data sized type and thus has the same behaviour as primitive data types:

```
fn main() {
    let x: [i32; 10] = [0; 10];
    let mut y = x;
    print!(" y = ");
    for i in &mut y {
        *i+=1;
        print!("{}  ", i);
    }

    print!("\n x = ");
    for i in &x {
        print!("{}  ", i);
    }
}
```

The Output

```
 y = 1  1  1  1  1  1  1  1  1  1
 x = 0  0  0  0  0  0  0  0  0  0
```

3.2.2 References and Pointers

A pointer is a variable that holds an address of a value. *Dereferencing* denoted by `*` is required to access the value it points to. Since its 2012 version, Rust only has two types of build-in pointers, in a big departure from many different types in the previous versions. The two types are *references* and *raw pointers*.

A reference is a pointer with highly restrictive syntax and semantics. It refers to a value, but it does not own it and the value it points to will not be dropped when the reference goes out of scope. That is why sometimes a reference is called a *borrowed pointer* because it "borrows" a value which it does not own. The syntax for a reference is `&T` or `&mut T`, indicating an immutable or mutable reference respectively. *Borrowing* is one of the fundamental concepts of Rust. Borrowing is needed for example when a binding needs to be passed to a function without moving the ownership.The two types of references in more details:

- *Reference type* (`&T`), where `T` is a generic type (by generic type is meant an arbitrary type). It should be noted that a reference type is an immutable reference type which means reading data is allowed, but not writing.
- *Mutable reference type* (`&mut T`), where `T` is a generic type, and this defines a mutable reference type. Mutable reference gives the opportunity to borrow the authority to mutate the data. Two points should be made in this case. First, the owner must be mutable in order to borrow it as a mutable reference. Second, the compiler `rustc` depends on *Read-Write Lock* pattern which is discussed in *Ownership* section.

A few comments on mutability of structures. Consider

```
struct Student {
   name: &'static str,
   age : u32,
}
```

```
fn main() {
  let s1 = Student {name: "Sara", age: 20};
  println!("Student's name is {},{} years old",s1.name,s1.age);
}
```

The Output:

```
Student's name is Sara, 20 years old
```

Mutable binding of field objects is not allowed in a structure definition even though mutable references are allowed. Second, structure binding is immutable by default. The following fragments of code illustrate this.

Mutable binding in structure is not supported at language level. The following code causes compile-time error: `Error: expected identifier, found keyword 'mut'`.

```
struct Circut {
   mut switch : bool,   //error
}
```

The following code shows that a structure data field could be bound to a mutable reference.

```
use std::string::String;

struct Student {
   name: String,
   age : u32,
 }

 struct StudentRef<'a> {
```

```
    name : &'a mut String,
    age : &'a mut u32,
  }

fn main() {
  let mut s1 = Student {name: String::from("Sara Ahmad"), age:
   20};
    {
      let s2 = StudentRef{name: &mut s1.name , age: &mut s1.age};
      *s2.name = String::from("Elaf Ahmad");
      *s2.age  = 21;
    }
println!("Student's name is {},{} years old",s1.name,s1.age);
}
```

The Output :

```
Student's name is Sara Ahmad, 21 years old
```

Declarations `*const T` and `*mut T` define so-called *raw pointers* in Rust, again an immutable and a mutable version respectively. Since much of memory safety in Rust come from compile-time enforced rules, raw pointers are inherently unsafe. Declaring a raw pointer is safe, but dereferencing may not be, the programmer must assure that the raw pointer points to a real value. Unless necessary, programmers should not use raw pointers, instead use various *smart pointers* defined in the standard library `std`; these are "wrapper type" abstractions. They are called smart pointers as they behave as the original smart pointers of C/C++ and the memory they reference is deallocated when a smart pointer goes out of scope.

- `Box<T>`

The simplest smart pointer. When set by allocation, it becomes the owner of the resource. Unless it passes the ownership, the resource will be deallocated using the user-provided destructor when the box goes out of scope. Box is not a wrapper and hence there is not run-time cost associated with using it.

- `Rc<T>`

  Reference counted smart pointer. There could be several owner pointers of this type for a resource and the resource is deallocated when the reference count reaches zero. In its implementation, it contains a shared global variable `refcount`, which is incremented each time an Rc is cloned, and decremented each time an Rc goes out of scope. The main responsibility of `Rc<T>` is to ensure that destructors are called for the shared data. On the other hand, the internal data is immutable, and if a cycle of references is created (by using RefCell<T>) [24], the memory will be leaked (i.e., never deallocated). Note that if we wanted data that does not leak when there is a reference cycle, we would need a garbage collector not based on reference counting. The Rc pointers are not thread-safe and thus cannot be shared by various threads – this alleviates the need for atomic instructions lowering the costs. The extra memory costs include allocation of two extra words for strong and weak (see next item) refcounts, while the computational costs include the code for updating the refcount whenever the pointer is cloned or goes out of scope.

- `Weak<T>`

  Similar to `Rc<T>`, it is is a non-owning and non-borrowed reference counted smart pointer. Though similar to a reference `&T`, it is not restricted in lifetime and can be held on to for the duration of the execution. This is useful for cyclic data structures.

Let us remark that there are other forms of smart pointers, especially so-called cells `Cell<T>` and `RefCell<T>`. `Cell<T>` allocates data of the cell on the stack and thus there is no danger of memory leaking and provides zero-cost interior mutability. `RefCell<T>` provides interior mutability with supporting borrow checker rules dynamically, however it does not move data in

and out of the cell. Then there are thread-safe smart pointers `Arc<T>` that are like `Rc<T>` with the reference count guaranteed to be atomic, and `Mutex<T>` and `RwLock<T>` are used to provide mutual-exclusion.

Much of Rust memory safety comes from compile-time checks, but raw pointers do not have such guarantees, and thus are unsafe to use.

3.3 The Rust Compiler `rustc` and the Ownership system

The Rust programming language compiler `rustc` focuses on three primary goals : Speed, Safety and Concurrency [25]. One of the most important facets of developing Rust programs is the ownership system organized into Ownership, Borrowing and Lifetime categories.

3.3.1 Ownership

Ownership is Rust's most unique feature in that the memory safety is provided without employing a garbage collector. Even though the concept of ownership is simple, the implementation of an ownership system affects all aspects of Rust. The compiler `rustc` enforces the following *ownership rules*:

1. Each value in Rust has a variable that is called its owner.
2. There can only be one owner at any time.
3. When the owner goes out of scope, the value is dropped.

Let us remark that the scoping in Rust is lexical scoping, i.e., defined as sections of the code – in simple terms, a scope of an item is the range within a program source code for which the item is valid. When a variable goes out of scope, Rust calls a special function `drop`, and it is where the programmer can put the code to deallocate the memory. This guarantees that there is no memory leak.

Smart pointers such as `Box` and `Rc` follow the ownership rules. Let us illustrate violations of some of the ownership rules.

- First consider the following code showing function `display` with a vector parameter (dynamic size type). The resource will be deallocated by the end of `display` function. A compile-time error `error: use of move value` will be generated caused if the owner of the vector is used after it has been passed to another function.

```
fn display(v:Vec<i32>) {
    for i in &v {
        println!("{} ", i);
    }
}


fn main() {
    let mut v = vec![1,2,3,4,5];
    display(v);
    v.push(6);    // error: use of moved value: `v`
}
```

- The following code shows the *scope* defined by a block `{ ... }` and the bindings when declared inside the scope will be cleaned up at the end of that scope.

```
fn main() {
    let v = vec!['a','b','c'];
    {
        let v2 = v;
        // more instructions
    }
```

```
    for i in &v {  // error:  use of moved value: `v`
        println!("{} ", i);
    }
}
```

3.3.2 Borrowing

When a reference is set, we refer to it as *borrowing*, the most common situation being when it is passed to a function. It is a useful strategy for the ownership system because it gives an opportunity to borrow a value with a specific authority and without the owner losing the ownership of the binding. Borrowing is associated strongly with the ownership rules.

The following example shows how `mystring` borrows the value of `string`:

```
let mut string = String::from("McMaster");
let mystring = & string;
```

In the next code, borrowing via passing of a parameter is shown (`display(&v)`). It shows basically the same `display` function as in the end of the previous section, but in this example the parameter `vector` will be borrowed to save the lifetime of the owner in the `main` function.

- Immutable borrowing

```
fn display(v:&Vec<i32>) {
    print!("v =   ");
    for i in v {
        print!("{}    ", i);
    }
    println!("    ");
}
```

```
fn main() {
    let mut v = vec![1,2,3,4,5];
    display(&v);
    v.push(6);
    display(&v);
}
```

The output

```
v =   1   2   3   4   5
v =   1   2   3   4   5   6
```

Note: if there were any attempt to mutate the vector inside the `display` function, a compile-time error would be caused: `error[E0596]: cannot borrow immutable borrowed content `*v` as mutable`.

- Mutable borrowing

```
fn display(v:&mut Vec<i32>) {
    v.push(6);
    v.push(7);
    v.push(8);
    print!("v =   ");
    for i in v {
        print!("{}   ", i);
    }
    println!("   ");
}
```

```
fn main() {
    let mut v = vec![1,2,3,4,5];
    display(&mut v);
}
```

The output

```
v =  1   2   3   4   5   6   7   8
```

3.3.3 Borrow Checker

The borrow checker is based on a strong set of rules enforced at compile time. The rules can be summarized as follows:

1. Any borrow must last for a scope no greater than that of the owner.
2. Only one of the following types of borrows is allowed, but not both of them
   (a) One or more (immutable) references (`&T`) to a resource,
   (b) Exactly one mutable reference (`&mut T`).

In order to illustrate the effectiveness of the borrow checker, the following attempts were tried for both fixed and dynamic data types.

Case 1: ***When multiple references*** (`& T`) ***are allowed.***

- Fixed size – stack allocation

  The following code shows three binding variables (`v1`, `v2`, and `v3`). Even though `v1` is the owner, `v2` and `v3` can be immutable references to `v1`. In other words, multiple immutable references can access an owner binding variable as *read only*. This is alright according to the rule 2(a):*One or more immutable references to a resource*.

```
fn main() {
    let v1 = 5;
    let v2 = &v1;
    let v3 = &v1;
    println!("v1 is {}, v2 is {} v3 is {}", v1, v2, v3);
    println!("Reference address");
    println!("v2 is {:p} v3 is {:p}", v2, v3);
}
```

- Dynamic size – heap allocation

  Even in dynamic size allocation, multiple immutable references are allowed to access an owned binding variable. It should be noted that the owner (`v1`) still exists int the scope and accessing the resource by the owner is allowed in this case.

```
fn main() {
    let v1 = vec![1,2,3,4,5];
    let v2 = &v1;
    let v3 = &v1;
    println!("v1[0] is {}, v2[0] is {} v3[0] is {}", v1[0],
   v2[0], v3[0]);
    println!("Reference address");
    println!("v2 is {:p} v3 is {:p}", v2, v3);
}
```

Case 2: ***Borrowing mutable reference from immutable owner***.

- Fixed size – stack allocation

  It is not only logically incorrect, but also the compiler `rustc` strictly disallows this case. If the

owner is immutable, its resource is not allowed to be borrowed as mutable. `error[E0596]: cannot borrow immutable local variable ‘v1‘ as mutable` will be caused during the compilation.

```
fn main() {
    let v1 = 5;
    let v2 = &mut v1;   // error
    println!("v1 is {}, v2 is {} ", v1, v2);
}
```

- Dynamic size – heap allocation

  The same error will be generated by the compiler if a mutable borrow is attempted with an immutable owner.

```
fn main() {
    let v1 = vec![1,2,3,4,5];
    let v2 = &mut v1;    // error
    println!("v1[0] is {}, v2[0] is {} ", v1[0], v2[0]);
}
```

Case 3: ***Borrowing immutable reference from an immutable reference of an immutable owner – nesting of borrowing.***

- Fixed size – stack allocation

  The immutable references are allowed to be nested as long as there is one mutable owner or exactly one mutable reference.

```
fn main() {
    let v1 = 5;
    let v2 = &v1;
```

```
    let v3 = &v2;
    println!("v1 is {}, v2 is {} v3 is {}", v1, v2, v3);
    println!("Reference address");
    println!("v2 is {:p} v3 is {:p}", v2, v3);
}
```

- Dynamic size – heap allocation

  The rules of compiler `rustc` are uniform for static or dynamic allocation types.

```
fn main() {
    let v1 = vec![1,2,3,4,5];
    let v2 = &v1;
    let v3 = &v2;
    println!("v1[0] is {}, v2[0] is {} v3[0] is {}", v1[0],
   v2[0], v3[0]);
    println!("Reference address");
    println!("v2 is {:p} v3 is {:p}", v2, v3);
}
```

Case 4: ***Two mutable references for a mutable owner***.

- Fixed size – stack allocation

  It strictly breaks the rule 2(b): *Exactly one mutable reference*. The following error will be produced: `"error[E0499]: cannot borrow ‘v1‘ as mutable more than once at a time`

```
fn main() {
    let mut v1 = 5;
    let v2 = &mut v1;
```

```
    let v3 = &mut v1; // error
}
```

- Dynamic size – heap allocation

  The same error will occur when the following code is executed.

```
fn main() {
    let mut v1 = vec![1,2,3,4,5];
    let v2 = &mut v1;
    let v3 = &mut v1;   //error
}
```

Note: the following code describes the same issue of more than one mutable reference even if a resource (`v2`) is a mutable reference of `v1`, which is the owner, and `v3` is a mutable reference of `v2`. The following error `error[E0596]: cannot borrow immutable local variable ‘v2‘ as mutable` will occur during the compile time.

```
fn main() {
    let mut v1 = vec![1,2,3,4,5];
    let v2 = &mut v1;
    let v3 = &mut v2;   // error
}
```

The compilation error suggest to mutate `v2`.

```
fn main() {
    let mut v1 = vec![1,2,3,4,5];
    let mut v2 = &mut v1;
    let v3 = &mut v2;
}
```

The compiler will accept the above code. It should be noted that, without accessing and printing the binding variables, programmers cannot find which binding variable is allowed to be in read or write mode.

```
fn main() {
    let mut v1 = vec![1,2,3,4,5];
    let mut v2 = &mut v1;
    let v3 = &mut v2;
    v3[0] = 10 ;
    println!("v3[0] = {} ",v3[0]);
}
```

To summarize the facts that can be derived from the above code is that `&mut T` can transfer the ownership from a binding variable (`v1`) to another (`v2`). If `&T` is needed, only the last `&mut T` can be used. The first resource (`v1`) is not applicable to be used by the compiler rules. In other words, `&mut T` will transfer the mutability and readability of a binding variable from one to another.

If a reference refers a primitive data type, static allocation and dereference (*) are needed. Note the following example showing a reference of a reference, so not only once, but dereferencing is required twice.

```
fn main() {
    let mut v1 = 5;
    let mut v2 = &mut v1;
    let v3 = &mut v2;
    *(*v3) = 10 ;
    println!("v3 = {} ",v3);
}
```

Case 5: ***The order of borrowing mutable reference and immutable reference for the same resource and vice verse***.

- Fixed size – stack allocation

```
fn main() {
    let mut v1 = 5;
    let v2 = &mut v1;
    *v2 = 10;    // correct

    let v3 = &v2;
    //*v2 = 10;  // error
    println!("v2 = {}, v3 = {}",v2,v3);
}
```

- Dynamic size – heap allocation

```
fn main() {
    let mut v1 = vec![1,2,3,4,5];
    let v2 = &mut v1;
    v2[0] = 10;    // correct

    let v3 = &v2;
    //v2[0] = 10;  // error
    println!("v2[0] = {}, v3[0] = {}",v2[0],v3[0]);
}
```

From the time of the transfer of the ownership (`&mut`) from `v1` to `v2`, `v2` has the authority to access and modify the content of the binding variable, if `v2` had been borrowed as immutable to a `v3`. No update is allowed to `v2`; in other words, `v2` lost the mutability permission.

The question is now what is the state of a binding variable if an immutable reference is borrowed before a mutable reference for the same resource.

```
fn main() {
  let mut v1 = 5;
  let  v2 = &v1;  // immutable reference

  let v3 = &mut v1;   // mutable reference
  println!("v2 = {}, v3 = {}",v2,v3);
}
```

The following error `error[E0502]: cannot borrow ‘v1‘ as mutable because it is also borrowed as immutable` will occur as the ordering of borrowing references is taken into consideration.

3.3.4 Scopes

Before getting to the third fundamental concept in Rust which is *lifetime*, programmers need to think about *scope*. As mentioned previously, Rust use static, or equivalently, lexical scoping. Thus a scope refers to a section in the code where an item such as variable is valid. In a sense, a named lifetime is a way to give a name to a scope.

The following example shows the lifetime of binding and clean-up when the end of the scope is reached.

- When a lifetime of scope is attempted to be used after it has been deallocated :

```
fn main() {
   let mut x = 1;
   {
      let mut y = 2;
      println!("Inner scope x = {}, y = {}",x,y);
   }
   // error: error[E0425]: cannot find value 'y' in this
   scope
   println!("Outer scope x ={}, y = {}", x,y);
}
```

- Borrowing is strongly associated with the lifetime of a resource. The authority of using borrowed resource is following the ownership rules and the resource of the owner will not be deallocated until the owner itself goes out of scope.

```
fn main() {
   let mut x = 10;
   {
      let mut y = &mut x;
      (*y) = 100;
      println!("y = {} ", y);
   }
   println!("x = {}", x);
}
```

It should be noted that a mutable borrowed resource will freeze the authority of the owner during its lifetime. In fact, the following compile-time error (*"error[E0502]: cannot borrow 'x' as*

immutable because it is also borrowed as mutable") will be caused if a binding of `x` is attempted to be used after it had been borrowed by `y`.

3.3.5 Lifetimes

One of the strongest features in Rust is the ownership system designed to prevent not only data races – when two or more pointers access the same memory location at the same time, where at least one of them is writing, and the operations are not synchronized, but also specifically the more dangerous problem of *dangling pointers* – when the value a pointer points to had been previously released. A lifetime is a construct the borrow checker of `rustc` uses to ensure that all borrows are valid. A variable's lifetime begins when it is created and ends when it is destroyed. Lifetimes and scopes are intertwined, but they are not the same thing. The following simple example illustrates the situation when the scopes and lifetimes coincide:

```
fn main() {                     // scope 1 starts
   let i = 5;                   // i lifetime starts
   {// scope 2 starts
      let x = &i;               // x lifetime starts
      println!("x = {}", x);
   }                            // scope 2 ends, x lifetime ends
   {                            // scope 3 starts
      let y = &i;               // y lifetime starts
      println!("y = {}", y);
   }                            // scope 3 ends, y lifetime ends
}                               // scope 1 ends, i lifetime ends
```

The borrow checker uses symbolic names representing lifetimes to determine how long a particular reference should be valid. The notation `<'a>` designates a name `a` for lifetime, and `<'a:'b>` designates the fact that the lifetime `a` should be the same or longer than the lifetime `b`.

```
// references op1 and op2 are coerced to have
// the same lifetime as  add()
fn add<'a>(op1: &'a i32, op2: &'a i32) -> i32 {
    op1 + op2
}

// reference with lifetime a is the parameter, it is coerced to
// a reference with lifetime b and returned -- the lifespan
// of op1 was shortened
fn which_is_op1<'a: 'b, 'b>(op1: &'a i32, op2: &'b i32) -> &'b
   i32 {
    op1
}

fn main() {
    let op1 = 1; // longer lifetime
    {
        let op2 = 2; // shorter lifetime
        println!("The sum is {}", add(&op1, &op2));
        println!("{} is the first operand", which_is_op1(&op1,
   &op2));
    };
}
```

Running the same code without the symbolic names for the lifespans gives compile-time error

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:5:42
```

```
  |
5 | fn which_is_op1(op1: &i32, op2: &i32) -> &i32 {
  |                                          ^ expected lifetime
   parameter
  |
  = help: this function's return type contains a borrowed value,
   but the signature does not say whether it is borrowed from
   `op1` or `op2`
```

The following code shows two different circumstances that could lead to dangling pointer errors and the ownership rules in Rust will find those types of errors at compile time.

- A compile-time error will be caused if a reference lives longer than the resource it refers to.

```
fn main() {
   let y : &i32;
   let x = 5;
   y = &x;
   println!("{} ", y);
}
```

The compiler will respond with

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:4:9
  |
4 |    y = &x;
  |         ^ borrowed value does not live long enough
...
7 | }
```

```
  | - `x` dropped here while still borrowed
  |
  = note: values in a scope are dropped in the opposite
   order they are created
```

It is a very illustrative example and it has everything to do with the order of variables being destroyed – in the reverse order of their declaration. Hence at the end of the scope, first x is deallocated, then y. However, when x is being destroyed, it is still borrowed by y. To make it work, we need to make the lifetime of x exceed the lifetime of y. In this particular case, just reordering the declarations will make the code compile alright.

```
fn main() {
   let x = 5;
   let y : &i32;
   y = &x;
   println!("{} ", y);
}
```

- A compile-time error will be caused if a resource points to invalid resource.

```
fn main()  {
   let r;
   {
      let i = 1;
      //error[E0597]: `i` does not live long enough
      r = &i;
   }
   // `i` dropped here while still borrowed
```

```
    println!("{}", r);
}
```

The variable *r* will live longer than *i*, but after *i* is deallocated, *r* would be dangling.

3.4 Analysis of the Ownership System

In Rust, as in garbage collected languages, the programmer never explicitly frees memory. It achieves this without run-time costs and without sacrificing safety. By the run-time costs we mean garbage collection and/or reference counting, while neglecting as usual the costs of manual deallocation (`free` in C or `delete` in C++). By the memory safety we mean the following guarantees

- No dangling pointers, via ownership types.
- No memory leaks.
- No use-after-free.
- No reads of uninitialized values.

The basic idea is that every object is owned by exactly one owner. But the ownership can be transferred, and once the ownership is transferred, the original owner is banned from accessing it. Rust ties the scope of returned values from functions to the scope of a borrowed argument. We can contrast it with C++ which makes the distinction between “transferred" and “lent" arguments explicit (passing by value and passing by reference) in some cases and doesn’t check for mistakes. Languages with a garbage collection typically hide this distinction. Note, that Rust has reference-counted smart pointers (`Rc<T>`) and plans in the future to add a garbage collector. In a sense the use of smart pointers is a throwback to C/C++ paradigm of programming. Rust’s ownership really improves local reasoning leading to a different paradigm of programming. As a result, Rust is a very fast language that lets you control memory directly with an absolute guarantee of safety.

The process of transferring ownership in Rust is called *moving*. After a move, the contents of the original variable are considered no longer valid or important, in fact Rust actually pretends the variable is "logically uninitialized" and is forbidden to be used, unless it is re-initialized. On the other hand, we need to consider under which circumstances we need to have multiple references to the same resource. Moving a value back and forth would not be appealing and correct. This is resolved in Rust be *borrowing*:

- Allows to have multiple references to a resource while still adhering to the "single owner" concept.
- References are similar to pointers in C/C++.
- A reference is an object as well. Mutable references are moved, immutable ones are copied. When a reference is dropped, the borrow ends, of course, subject to the lifetime rules.
- In the simplest case references behave as if we were moving the ownership back and forth without doing it explicitly.

Even though it seems that acquiring and dropping references work as if there was a garbage collection, everything is done at statically at compile-time with zero or negligible run-time costs. References, among other objects, have lifetimes and those can be different from the lifetimes of the borrow they represent. It stands to reason that the lifetime of a borrow should not exceed the lifetime of the owner, that is why programmers sometimes need to specify lifetime details explicitly.

Chapter 4

Swift

Memory management is one of the most critical components of a programming language. In fact, how memory management allocates a chunk of memory according to an application request and how this space is reclaimed when the resource is no more needed varies from one programming language design to another. In 2014, the Swift programming language was introduced at Apple's 2014 Worldwide Developers Conference (WWDC) [26]. Swift is a compiled programming language "designed by Apple for developing IOS and OS X applications. Swift-compiled programs will run on iOS7 or newer and OS X 10.9 (Mavericks) or newer" [27]. Furthermore, Swift is compatible with the Cocoa and Cocoa Touch frameworks[27, 28, 29]. As reported by Apple developers, Swift is a safe, fast, and interactive programming language [30]. Swift supports the concept of ownership to manage its dynamic reference types by using Automatic Reference Counting ARC.

In order to discuss ownership in Swift, there are few points that must be illustrated. The difference between value types and reference types will clarify the initial steps of finding how types are managed in Swift, the transition of "Manual Retain Release", (MRR) and "Automatic Reference Counting" (ARC) will introduce the history of memory management designs in Swift and what the weakness that are enhanced in ARC; in addition, the strong reference types and lifetime qualifiers should be discussed and how they strength the usability of Swift. Therefore, discussing the memory safety and ownership design will provide an approach to introducing memory issues.

4.1 Swift Programming Language

In this section, we introduce the Swift programming language focusing on various data types. Swift supports automatic reference counting, known as ARC, that will be discussed in details including the consequences of using ARC. We will indicate the reasons of using ARC as well. Ownership in Swift is presented in reference counting by using strong reference lifetime. Even though ownership in Swift is not too obvious, developers need to be aware of reference types and lifetimes concepts.

New programmers will enjoy using Swift, but the requirements and the rules of building a simple application in Swift are quite complex and are discussed in this section.

4.2 Swift Type Essentials

Swift is a general-purpose programming language that supports not only object-oriented paradigm, but also imperative and functional paradigms [4]. In fact, Swift developers emphasize that Swift is enjoyable, expressive, and safe language [30]. *Constants*, *variables* and *optional* types are supported beside classes and structures. The following sections will introduce them and how they are related to ownership and memory safety in Swift.

4.2.1 Value, Reference and Optional Types

Swift introduces two different methods for declaring constants and variables by using the keyword *let* or *var* respectively. Constants are not allowed to be mutated after initialization whereas variables have the choice to be mutated during their lifetime.

The difference between value types and reference types in Swift is highly important. In fact, *primitive* data types, *collections* [31], strings, structures, and enumerations are value types. On other hand, classes and closures are reference types. ARC will manage reference types using "reference

counting mechanism". Even though memory managements of structures and classes are different, there are significant similarities based on properties, declarations, and accessing syntax.

One important factor related to reference types is optional types. In fact, Swift supports the operation of optional chaining symbolized by `(?)` which accepts `nil` value. Invoking optional chain has two results: if the optional contains a value, then the property, method, or subscript call succeeds, otherwise `nil` value is returned indicating that the optional contains `nil`. To forcefully access optional types, the operation `(!)` is required. Optional values will be used extensively when lifetime qualifiers are needed. The following pieces of code describe class and structure declaration:

- **Structures are copy types**

```
struct Student_st {
var name : String;  var age: UInt
var major : String; var enrolled : Bool
}
```

- **Classes are reference types**

```
    class Student_cl {
    var name : String; var age: UInt
    var major : String; var enrolled : Bool

    init(n:String,a: UInt, m:String ,e:Bool) {
        name = n; age = a; major = m; enrolled = e
    }
}
```

It must be noted that Swift is not only a compiled language, but also a strongly typed language that needs an initialization process to be guaranteed before creating variables or instances. In fact, both structures and classes can use default value for initialization as the first choice. The second choice is to use `init()` method. Swift provides a special method that does not return a value and initial value of properties will be initialized. Unlike classes, structures have *memberwise initializer* [32] which is automatically generated to initialize structure properties. `deinit()` method is also provided to execute a custom code before an instance destruction [33, 34].

4.3 Swift ARC and Ownership

4.3.1 MRR – ARC

Even though *reference counting* (or sometimes called *retain counting*) is the primary mechanism that is used in Objective-C and Swift for keeping track of the objects that are owned, reference counting has two models: *manual reference counting* which is known as *Manual Retain-Release* or **MRR** and *Automatic Reference Counting* or **ARC** [35, 36]. In ARC, the programmers do not have to think about retain and release operations that are required in MRR. In addition, ARC compiler will manage the lifetime of objects by inserting appropriate memory management method calls to perform object destruction automatically once the object is no more needed.

Reference counting is the mechanism that has been used in Objective-C an so supported by Apple, and ARC is the official standard adopted in Swift. When an object is created and assigned to a property, constant, or variable, the integer that is associated with the object will be increased by the number of properties, constants, or variables that hold a reference to the object. This integer associated with the object is called the *reference count* and the object will be reclaimed when its reference count reaches zero; in other words, there is no more property, constant, or variable that needs that object[27].

Therefore, ARC is "a compile-time system that keeps track of and manages the memory an application uses"[27]. Unlike MRR, ARC deallocation is managed automatically and so programmers do not have to worry about the objects destruction. However, full understanding of the concept of the ownership is required in order to write a safe code in Swift.

Writing a safe code in Swift mostly means avoiding memory leaks. Even though creating an object that is allocated a chunk of memory, and reclaiming that memory by ARC when that object is no more needed sounds elegant and straightforward, ownership and lifetime concepts are strongly required to manage this automatic handling. In fact, when an object is owned by a property, constant, or variable, that object will be alive as long as the object is needed. By default, a reference in Swift is *Strong*, see below. As a result, if there are multiple strong references to an object, there are multiple owners.

4.3.2 Strong Reference Ownership

The following *Student* class defines `name`, `age`, `major`, and `enrolled` properties. Four student objects will be created. To describe how the deallocation will be executed, owner variables will be optional types in order to assign *nil* value. If the reference count of an owner is zero, ARC will automatically execute the deinitialization method which only gives a hint to programmers that an object will be automatically reclaimed by the compiler not the `dinit()` method.

It is significant to note that, for the following code, `student2` and `student3` point to the same reference. That will clarify if the deallocation of an object will be executed if the two owners are no longer owning the reference.

- **Class declaration**

```
class Student {
    var name : String; var age: UInt
```

```
    var major : String; var enrolled : Bool

    init(n: String, a: UInt, m: String , e: Bool) {
                name = n; age = a; major = m; enrolled = e
    }
    deinit {
                print("Student \(self.name) has been
   deleted!")
    }
}
```

- **Object declaration**

```
var student1:Student? = Student(n: "Harika", a: 20, m:
   "computer science", e: true)
var student2:Student? = Student(n: "Elaf", a: 20 , m:
   "computer science", e: false)
var student3 = student2
var student4:Student? = Student(n: "Zahra", a : 19 , m:
   "Software Engineering", e: true)
```

- **Destructing**

```
student1 = nil; student2 = nil; student4 = nil; student3 =
   nil
```

- **Output**

```
Student Harika has been deleted!
Student Zahra has been deleted!
Student Elaf has been deleted!
```

4.4 ARC and Memory Issues

One of the primary safety of programming languages is to provide memory safety guarantees. Memory leaking and dangling pointer are the most common problems in software. Memory leaking is caused when a memory location is not reachable and deallocating it is not under control either explicitly or implicitly. In addition, dangling pointer is caused when a reference is pointing to an object that has been reclaimed.

Even though Swift developers claim that Swift is a safe language, Swift has memory issues and programmers are responsible to avoid them. Strong references cycle is one of the main reasons of causing memory leaking. If a programmer were not aware of how strong reference cycle is caused, then there is a high possibility of causing memory leak.

4.4.1 Strong Reference Cycle

The concept of ownership seems strong and it controls the lifetime of object from the creation to the destruction. Comparing reference counting to the Garbage collection (GC for short) seems

interesting. In fact, GC is a non-deterministic deallocation because when the GC will clean up the unreachable resource is not known. In contrast, reference counting, deallocates the reference types in deterministic style. Even though these styles require comparing the overhead, reference counting primary issue is the memory leaking.

The following examples of `Phone Customer` class relationships will illustrate how strong reference cycle causes memory leaking. It should be noted that each customer has a phone number and each phone number has an owner. Even though instance properties are optional and assigning the owner variables of `customer1` and `phone1` to `nil`, the dinitilization method has not be executed to guarantee the destruction. Therefore, the objects of customer and phone are still alive.

- **Class declarations**
  - **Phone Class**

```
class Phone {
    var number : UInt; var customer : Customer?
= nil

    init(n:UInt){
    number = n; print("\(number) has been
created!")
    }

    deinit {
    print("Phone number \(number) has been
deleted!")
    }
}
```

– **Customer Class**

```
class Customer {
    var name : String; var phone: Phone? = nil

    init(n: String) {
    name = n; print("Customer \(name) has been
created!")
    }

    deinit {
    print("Customer \(name) has been deleted!")
    }
}
```

- **Object declarations**

  – **Creating**

```
var customer1: Customer? = Customer (n: "Ahmad")
var phone1: Phone? = Phone(n: 1009367099)
```

  – **Assigning**

```
customer1!.phone = phone1 ; phone1!.customer =
customer1
```

- **Destructing**

```
customer1 = nil; phone1 = nil
```

- **Output**

  Any attempt to clean `customer1` and `phone1` reference types will not be applicable and the `deinit()` has not been executed because objects still life in memory.

4.5 Lifetime Qualifiers

Strong reference cycle is presented if the relationship between objects are not defined. Lifetime qualifiers such as `weak` and `unowned` reference types contribute to solve memory leaking issues. Swift is an object-oriented language and there is a relationship between classes and objects [37] such as *association*: Two objects are independent, *aggregation*: Two objects are dependent, or *composition*: Two object are strongly dependent. Therefore, Swift developers introduce the lifetime qualifiers as a reference types to support the compiler and overcome memory leaking. In fact, `weak` and *unowned* references are not only used to prevent strong reference cycle, but also help the compiler destruct objects automatically.

4.5.1 `Weak` Lifetime

Swift developers indicate that weak references could be used if other instances has shorter lifetime; in other words, if an instance should be reclaimed first. Weak reference is “a reference that does not keep a strong hold on the instance it refers to, and so does not stop ARC from disposing of the referenced instance"[38]. This relationship seems similar to the relationship of `Customer` and `Car` classes. Customer may own a car (strong reference) but a car may exist without an owner (weak reference). In a different way, this relationship describes two **independent** lifetimes which is could be described as *Association relationship*. To clarify the reference types of these instances, the following rules should be satisfied:

- Customer car instance should be an optional type
- Car's owner (customer instance) should be a weak and optional type

The following classes *Customer* and *Car* illustrates weak reference types:

- **Class declarations**
  - **Customer**

```
class Customer {
var name : String; var age  : Int
var car : Car?

init(name : String , age : Int) {
    self.name = name; self.age = age
    print( "Customer \(self.name) is created")
    }

deinit {
    print( "Customer \(self.name) is reclaimed")
    }
}
```

  - **Car**

```
class Car {
var number: UInt64
weak var  owner : Customer?
```

```
        init(number :UInt64) {
            self.number = number
            print("Car with \(number) is created")
        }
        deinit {
            print("Car \(number) is reclaimed")
        }
    }
```

- **Object declarations**
  - **Creating**

```
        var customer1:Customer? = Customer(name:
   "Ahmad", age: 34)
        var car1:Car? = Car(number: 11123334)
```

  - **Assigning**

```
        customer1!.car=car1
        car1!.owner = customer1
```

- **Destructing**

```
    customer1 = nil
    //car's owner value after Customer deleted
    print("Car's owner is  \(car1!.owner)")
```

- **Output**

```
Customer Ahmad is created

Car with 11123334 is created

Customer Ahmad is reclaimed

Car's owner is  nil
```

It can be seen that memory leaking will not be caused because weak references will not stop ARC from deallocating the instance and if an instance of weak reference has been deleted, `nil` value will be assigned to weak reference type. Note, the operation (`!`) is used to unwrap and access `car1` optional type.

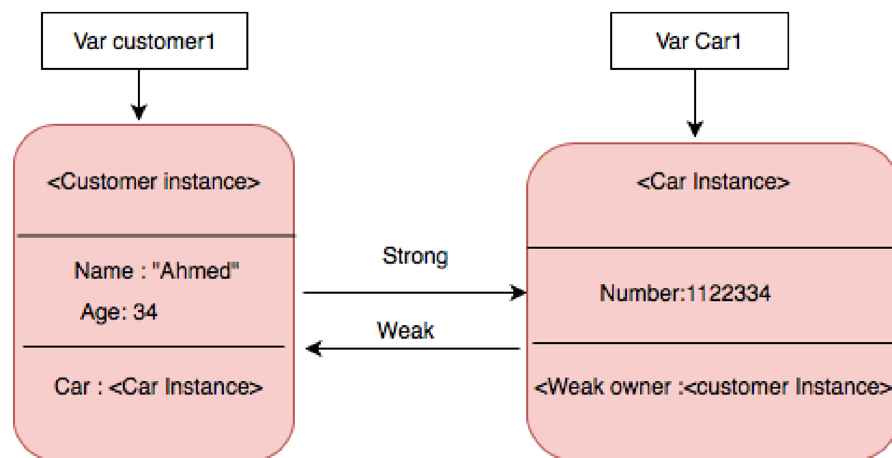

Figure 4.1: Weak reference – Customer Car objects

4.5.2 Unowned Lifetime

In some cases of relationships, objects may have dependent relationship. The lifetime of an object depends on another object; in other words, destructing one objects will cause a destruction of another. Swift developers provide `unowned` reference type where should be used when other instance has

the same lifetime or longer lifetime [38]. Unowned reference is expected also to have a value during the object lifetime.

There are two notes have been higlited by Swift developers: "Use an unowned reference only when you are sure that the reference always refers to an instance that has not been deallocated" and "If you try to access the value of an unowned reference after that instance has been deallocated, a runtime error will be executed."

To illustrate how to use `unowned` reference type, `Phone` and `Customer` classes will be used to describe dependent relationship. Let us consider a phone number company polices. Owning a number must require a customer to exist. As a result `Phone` cannot be alive if there is no customer owning the phone. The following `Customer` class has an optional phone instance but `Phone` class has (unowned) and non-optional customer instance. To clarify the reference types of these instances, the following rules should be satisfied:

- Customer phone instance should be optional type (strong reference)
- A phone of a customer should be unowned, **not optional type**: *when a phone is created a customer must be assigned*

The following classes `Customer` and `Car` illustrates weak reference types:

- **Class declarations**
  - **Customer class**

```
class Customer {
    var name : String;  var age  : Int
    var phone : Phone?

init(name : String , age : Int) {
```

```
            self.name = name; self.age = age
            print( "\(self.name) is created")
        }
        deinit {
            print( "\(self.name) is reclaimed")
        }
    }
```

– **Phone class**

```
        class Phone {
            var number : UInt64
            unowned let owner : Customer

        init (number:UInt64 , owner:Customer) {
            self.number = number; self.owner = owner
            print("\(number) of a \(owner.name)")
        }

        deinit {
            // Program will crush if owner name is
printed,
            // Customer will be deallocated first
            print("\(number), is cancelled")
    }
}
```

- **Object declaration and assigning**

```
var customer1 : Customer? = Customer
(name:"Ahmad",age:34)
customer1!.phone = Phone(number: 16470010011, owner:
customer1!)
```

- **Destructing**

```
customer1 = nil
```

- **Output**

```
Ahmad is created
16470010011 of a Ahmad
Ahmad is reclaimed
16470010011, is cancelled
```

Unowned reference type will not only solve the strong reference cycle but also provides a lifetime qualifier for illustrating dependent lifetimes. Unlike weak reference type that accepts to be `nil` during object lifetime, unowned reference type needs a reference value during its lifetime. Therefore, weak and unowned references show independent and dependent lifetimes respectively.

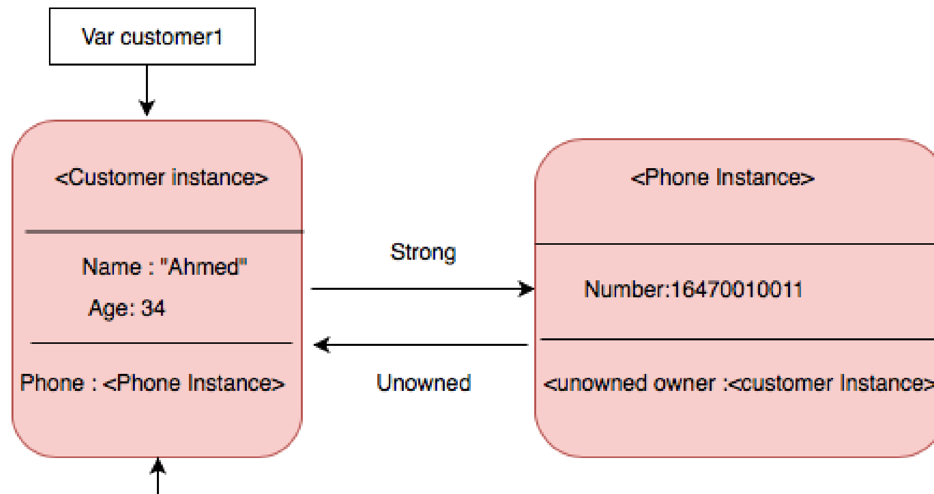

Figure 4.2: Unowned reference – Customer Phone objects

4.5.3 Strong Dependent Lifetime

The simplest illustration of relationship is *association* when two instances are related with independent lifetime and semantics. In fact, there is a kind of relationship which is even stronger than *aggregation* which is *composition*. For example, the relationship between a building and rooms are kind of composition relationship because the lifetime of rooms strongly depends on a building lifetime, as a result when a building is created the rooms will be created inside the building, and if the building is destroyed the rooms that belong to the building will be destroyed with it.

In fact, Swift developers combine two features *unowned reference* and *unwrapped optional property* to simulate this relationship[38]. The combination of these two features satisfies the guarantee that both properties have values instead of being `nil`. The following classes `Building` and `Room` will illustrate a composition relationship.

- **Class declarations**
  - **Building class**

```
class Building{
    let name : String
```

```
            var room : Room!

            init(name: String, n_rooms: UInt) {
            self.name = name
            self.room = Room(number: n_rooms,
building : self)
            print("\(self.name) creates \(n_rooms)
rooms")
            }

            deinit {
            print("\(self.name) is deleted")
            }
        }
```

– **Rooms class**

```
        class Room {
            let number : UInt
            unowned let building: Building

            init(number: UInt, building: Building) {
            self.number = number
            self.building = building
            }
        }
```

- **Object Declaration**

```
var b1:Building? = Building(name: "100 James", n_rooms:
3)
print("\(b1!.name) has \(b1!.room.number) rooms")
```

- **Destruction**

```
b1 = nil
```

- **Output**

```
100 James creates 3 rooms
100 James has 3 rooms
100 James is deleted
```

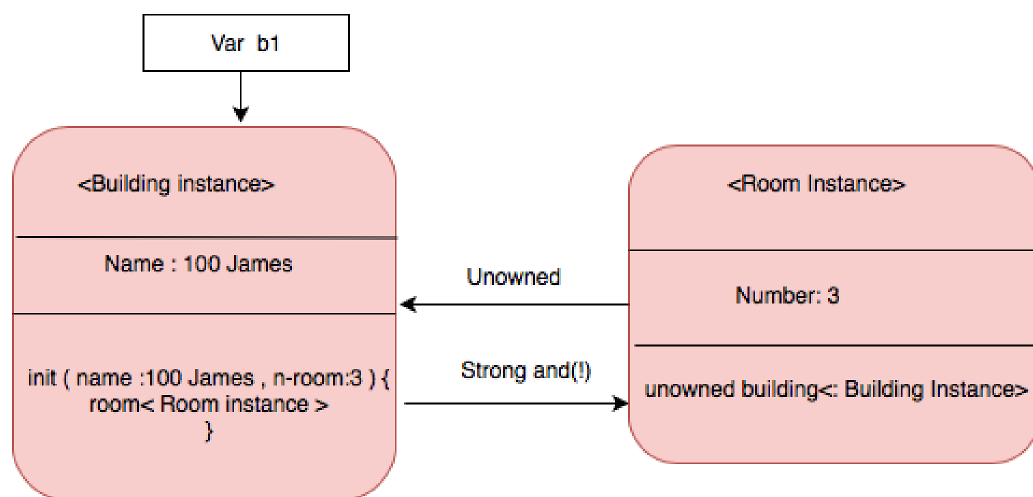

Figure 4.3: Unowned with Unwrap (!) operator – Building Room objects

The third type of relationship defines not only how Swift developers had overcome strong reference cycle, but also how objects relationship is composed. Swift programmers need to be a ware of memory management and Class Objects relationship. Even though Swift supports ARC to implicitly destruct objects in a deterministic way, the programmers responsibility still has to provide a reliable design.

4.6 Analysis of Ownership in ARC

In Swift, ARC supports implicit memory destruction. Swift has run-time cost which is comparable to Java. In addition, Swift does not support all memory safety guarantees even though Swift is a strongly typed language. In fact, the following memory issues still exist in Swift:

- Dangling pointers
- Memory leaks
- Use after free

The idea of reference counting, as explained in section *Swift ARC and Ownership*, accepts multiple owners, that are identified as strong references. Reference counting will free the memory if the last owner does not need the object and if a ref-count becomes zero. Reference counting mechanism needs developers to be precise during the programming, which puts ARC in the programmer's responsibility as well. `Weak` and `unowned` references are not only useful for avoiding memory leaking, but also to define accurate relationships. Where instance property should be declared is an important approach not only in Swift, but in all (pure) object-oriented languages. We used the term pure object-oriented language because Rust is considered not a pure object-oriented language, by the authors of the so-called The Gang of Four book [39].

Beside ownership and lifetime qualifiers, Swift needs programmers to be precise about optional types. Even though `nil` values or optional types exist in other programming languages, Swift has

rules to combine these types with lifetime qualifiers; for example an `unowned` reference needs to have a valid value, and declaring an `unowned` reference as optional type will not be accepted by the compiler. In addition, the mutability of types by using `let` and `var` is still designed in naive way. However, the contribution of those rules and conditions in MRR – ARC transition is significantly noticeable, Swift still has an issues in memory safety.

Swift supports optional types to help programmers to be specific if `nil` value is required beside the unwrap operator `!`. Swift as other conventional programming languages support `nil`, but swift gives the choice to the programmer to define `optional` when the value of type may be valid value or `nil`. Beside optional types, there is a choice for programmers to define the mutability of a type. In other words, values in Swift must have valid value to be usable and that forces the requirement of initialization step. In fact, immutable variables are known in Swift as a `Constants`, where the value will be initialized and will be in immutable state during it is lifetime; in contrast `Variables` hold values of types that are possibly mutable – i.e., changing the value of variable may be allowed during its lifetime.

Chapter 5

Comparison Between Swift and Rust

5.1 Similarities and Differences

In this chapter, we will summarize the similarities and differences of Swift and Rust features in terms of *ownership*, *memory safety*, *usability*, and *paradigms*.

5.1.1 Ownership and Memory Safety

As we have mentioned previously, the concept of ownership is presented in both Rust and Swift, but Rust's ownership system is wider and more solid than Swift's. The major difference is that Rust's system is static and can be enforced during the compilation and results in zero or near zero run-time costs, while Swift's system is dynamic and the run-time costs are significant. Rust's ownership rules enforced by its compiler `rustc` are designed not only to satisfy memory safety, but also to prevent data races: the mutability of binding is adapted so that it contributes to prevention of data race problems. Most of memory management issues such as memory leaks, dangling pointers, use-after-free are prevented statically at compile time. Unlike Rust, Swift supports mutability in a naive, simple way providing constants, variable or optional types. Even though mutability in Swift will contribute to declaring objects lifetimes, it does not prevent undesirable behaviour.

Java has a non-deterministic and implicit memory management by using GC, whereas C, and C++ have a deterministic, explicit memory management. Rust combines deterministic and implicit memory deallocation with no or negligible run-time overhead. It must also be noted that Rust is like C, a system programming language that supports low level control. A significant example can be shown in destruction. Deallocation in Rust depends on scope and lifetime of a binding. If an owner goes out of scope, a binding will be deallocated by the compiler. Rust supports traits, which represent behaviour or functionality of common types; trait is a concept similar to interface or super class in other languages. However, there is the `Drop` trait, which is an important trait for running a custom code once an owner is deallocated. `Drop` in Rust trait is similar to `dinit()` method in Swift. Rust provides not only `fn drop(&mut self)` method, but also `std::mem::drop` function to explicitly enforce dropping. Rust designers took care of the compiler to be smart enough to find out *double free* if a method is used instead of memory drop function [40], The following code illustrates using drop trait with memory drop function.

It should be noted that `Drop` trait helps to execute a code when a resource is deallocated, but it is not required to perform the action of cleaning, because Rust automatically deallocates a binding by assigning a code at compile time when a binding goes out of scope.

One of the primary safety aspects in Rust is to avoid double freeing. The following code illustrates how a double free can be caused in Rust: `fn drop(&mut self)` method as (drop trait implementation) can be used to explicitly enforce dropping a resource, but a run-time error will be caused, because resource freeing will be executed first explicitly and then implicitly when the resource goes out of scope. However, if an explicit dropping is required before a resource goes out of scope `std::mem::drop` function is required instead of the method `drop` in drop trait.

```
struct Student {
    name : String,
    G1: u32,
    G2: u32,
}

impl Drop for Student {
    fn drop(&mut self) {
        println!("Student '{}' has been dropped !", self.name);
    }
}
```

It should be noted that method use dot syntax to invoke a method and parameters will be sent to functions in parentheses ().

```
fn main() {
   let mut s1 = Student {name:String::from("Sara"), G1:80,
   G2:90};
   {
   drop(s1);
   //s1.drop();    // error:explicit use of destructor method
   }
}
```

The `rustc` compiler significantly prevents memory issues, while Swift memory management ARC shows number of memory issues. Even though `Rc<T>` is the only pointer in Rust, `Rc<T>` with `RefCell<T>` may cause strong reference cycle as mentioned in the documentation of *The Rust*

Programming Language [**?**]. However, if Rust borrow rules are violated in run-time with RefCell, Panic will be executed at run-time .

5.1.2 Usability

Comparing Rust and Swift in usability is still under research. Swift developers claim that Swift is a very expressive language. Learning Swift seems much easier than learning Rust based on our experience of this research: Rust is pretty challenging and the “learning costs" are pretty high. On the other hand, mastering Rust and/or Swift provides a new level of programming cognitive skills; for example, both Swift and Rust are considered object-oriented languages, but in different ways. Because classes are not supported in Rust, it is known as not a pure object-oriented language, even though methods and objects are supported in structures; while Swift is a pure object-oriented language because classes are supported, but it should be noted that pointers are treated as in Objective-C.

It has been mentioned previously, that in Rust, composition guarantees of pointers have different meanings. For example Rc<RefCell<vec<T>>> and Rc<vec<RefCell<T>>> have different composition guarantees.

Rc<RefCell<vec<T>>> is similar to &mut vec <T>; it means the vector is entirely mutable and the following notes should be considered:

- Vector is entirely mutable, push and pop are allowed
- Only one mutable borrow of the whole vector at a given scope
- The code cannot simultaneously work on different elements of the vector from different Rc bounds

On other hand, Rc<vec<RefCell<T>>> is similar to &mut [T]; it means the size of the vector is not changeable and the following remarks should be considered:

- Borrowing is allowed for the individual elements, but the overall vector is immutable
- It is allowed to borrow separate elements independently
- It is not allowed to apply push or pop to the vector

Therefore, Rust is much stricter than Swift. Using Rust or Swift put novice programmers in trade-off between effectiveness and efficiency.

5.1.3 Programming Paradigms

Both Rust and Swift are multi-paradigms languages. In fact, Rust and Swift are object-oriented, imperative, and functional languages.

Closures, pattern-matching, and Generics are provided in both languages. Rust is a system programming language, whereas Swift is considered a general purpose programming language. Swift is still under development and will need more time to reach maturity. In fact, Swift developers are still introducing and proposing new features that may be incorporated in the language soon [41, 42].

Therefore, Rust and Swift have similarities and differences, but Rust has been solidified and it is not only an interesting comprehensive language, but its memory management is starting to compete with mature mainstream programming languages such as C,C++, and Java.

Chapter 6

Run-Time Experiment

Our objective is to evaluate both the current Rust and Swift language implementations in comparison with other languages (C, C++ and Java). This chapter presents the Methodology. Then, it presents and describes the experiments . Finally, it presents the results and discussion.

6.1 Methodology

6.1.1 The Targeted Feature

We selected memory management as one of the factors that may affect the properties of any new programming language. Memory management is the process of controlling and coordinating computer memory, assigning portions called blocks to various running programs to optimize overall system performance.

Our targeted programming languages are Rust and Swift as both support the ownership concept in regard of the memory management. This concept aims to eliminate the vulnerabilities resulting from the conventional methods of memory management.

6.1.2 Languages Selection

As mentioned previously, this study focus on studying the concept of ownership in Rust and Swift. In addition to that, we selected three other languages C, C++, and Java. We chose the most mainstream languages that are still in widespread use today in industry and academia. C and C++ language support manually managed languages, whereas Java supports automatic memory management.

6.1.3 Tasks Selection

To compare the efficiency of memory management in these programming languages, we decided to measure the performance of **dynamic allocation and deallocation** and chose task requiring a frequent allocation and deallocation. The following list shows the data structures and the algorithms that were measured:

- Random Permutations of a sequence of Numbers
  - *Generate* a sequence and *Shuffle* it using Fisher Yates algorithm
- Binary Tree
  - *Insert a new node*, *Search for a node*, and *Replace an existing node with a new one* algorithms
- Array
  - *Quick Sort* and *Binary Search* for sorted linear list of numbers

6.2 The Experiment Setting

The following Environment and Platform Operating system are used for measurements:

- Environment

  - macOS Sierra version 10.12.6 with Processor: 2.9 GHz Intel Core i5

- Compilers
  - Java, java "1.8.0-161"
  - C, Apple LLVM version 8.1.0 (Clang – 802.0.42)
  - C++ , Apple LLVM version 8.1.0 (Clang – 802.0.42)
  - Rust, rustc 1.23.0 (766bdllc8 2018-1-1)
  - Swift , Apple swift 3.1 (swift lang-802.0.53) and Clang – 802.0.42)

- Time functions and measurements

  All-time results are unified in Milliseconds and the average time is computed of 28 runs

  - $gettimeofday()$, returns the current system time in a timeval struct with the time in seconds and microseconds (C, C++, Swift) [43][44]
  - $mach_absolute_time()$, works in OSx,(C, C++,Swift) : returns 'Mach absolute time unit', nanoseconds and it is CPU dependent. In fact, It is better than $gettimeofday()$.[43]
  - $nanoTime()$, returns wall clock time in nanoseconds (Java)[45]
  - $Instant$, A measurement of a monotonically nondecreasing clock. Opaque and useful only with Duration. It returns seconds and microseconds. (Rust) [46]

We opted for the simplicity's sake to measure the elapsed time [47] since the averaging should smooth out the various discrepancies of system executions not related to our experiment. Also, the purpose of the experiments was not any strict bench-marking, but an illustrations of the efficiency of the respective memory management approaches.

6.3 Permutations of a sequence of Numbers

This section shows the results of generating permutations of a sequence of numbers by two methods. The first subsection presents the result of a loop to generate permutations of million

numbers. The second subsection illustrates the result with using the Fisher–Yates shuffle algorithm.

6.3.1 Loop – Generating Permutations

Table 6.1 and Figure 6.1 present the performance of generating permutations Table 6.2 and Figure 6.2 display the performance of generating permutations with optimized loop. The results show Swift is significantly faster than Rust in generating 1,000,000 permutations. Also, with the optimized loop, the Rust program becomes faster. Without the optimized loop, Rust generated permutations in approximately 96.622 ms, while its performance has improved significantly with the optimized loop. It should be noted that only for Rust and Swift we experimented with **release** mode for the optimization.

In fact, implementation of list in Swift is Array (fixed size) and in Rust is Vector (dynamic size). C, C++ and Java have significantly better performance in comparison to Swift and Rust before optimization.

Loop	
Swift	24.1120668929403 ms
C	5.22859653571429 ms
C++	5.39247821428572 ms
Java	5.96084571428571 ms
Rust	96.62198175 ms

Table 6.1: Generating Permutations – Loop

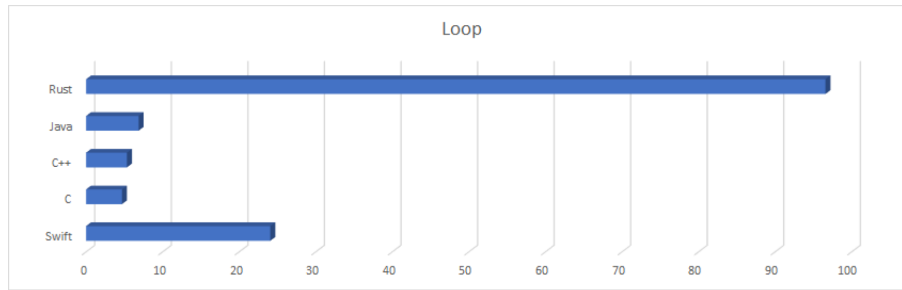

Figure 6.1: Generating Permutations – Loop

Loop	
Swift	0.822003071462469 ms
C	4.69449157142857 ms
C++	5.34256892857143 ms
Java	6.87084207142857 ms
Rust	2.86676953571429 ms

Table 6.2: Optimized – Loop

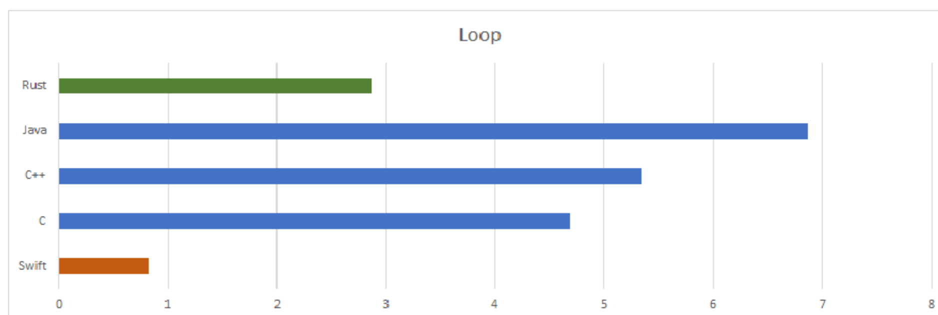

Figure 6.2: Optimized – Loop

6.3.2 Shuffle – Fisher-Yates algorithm

In order to generate a random permutation of a finite sequence, we used the Fisher–Yates shuffle algorithm. Table 6.3 and Figure 6.3 present the performance of generating permutations with this algorithm whereas Table 6.4 and Figure 6.4 present the performance of this algorithm with optimized shuffle.

The result shows Swift performance is better than Rust before optimization. However, after optimization, Rust speed time is sharply decreased in comparison to the same performance before optimization.

Shuffle	
Swift	220.900766785523 ms
C	1284.48449346429 ms
C++	1290.40107142857 ms
Java	178.316524464286 ms
Rust	590.934675142857 ms

Table 6.3: Shuffle

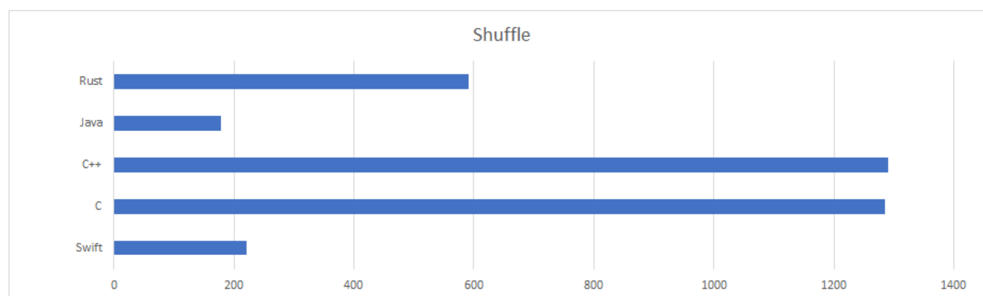

Figure 6.3: Shuffle

Shuffle	
Swift	93.9161121786705 ms
C	1284.48449346429 ms
C++	1290.40107142857 ms
Java	178.316524464286 ms
Rust	83.0169940357143 ms

Table 6.4: Optimized Shuffle

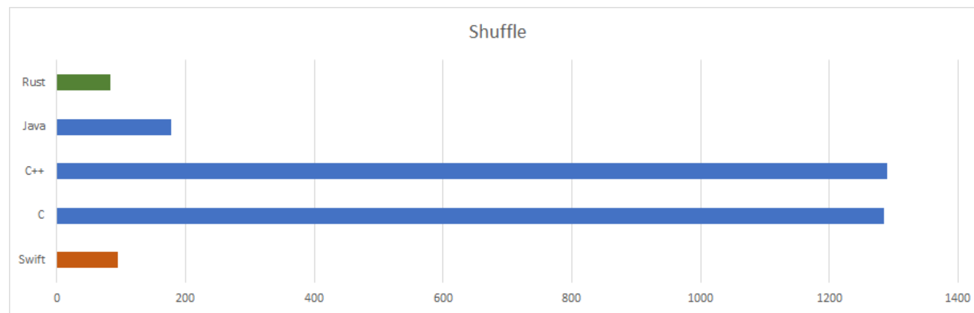

Figure 6.4: Optimized Shuffle

6.4 Binary Tree

This section, we used two approaches for measuring the performance of the algorithms using the data structure Binary Tree, the source code is in the Appendix A [48, 49]. The first approach was insertion – **number of** – nodes. We aimed to increase the number of nodes gradually to find out the memory management response in each selected language. The second approach is measuring the binary search node replacement by recreating an existing node. The replacement node will demonstrate the deallocation of old node by creating new one. In C and C++ , we have to deallocate a memory resource manually by free or delete it whereas Java, Rust and Swift will deallocate the old node implicitly by the memory management approach of each language. Finally, Binary search tree has been involved in the experiment by 1,000,000 random numbers. The results shows in the following sub-sections.

6.4.1 Insert

We measure the performance of five programs written in the selected languages for insertion starting with million nodes . Table 6.5 and Figure 6.5 present the performance of insert nodes before optimization while Table 6.6 and Figure 6.6 display the performance after optimization.

In C and C++, *malloc* and *new* were used respectively. In Swift and Java (object classes) were used to satisfy reference type allocation and `Box<T>` heap allocation smart pointers were used in Rust. It must also be noted that Rust unsafe pointers have never been used in these experiments. Swift's result begs the question "How fast is Swift?". The overhead is visible not only in the Swift program, but also in the Java program, but Java program has a much better in performance than the Swift program. It should be noted that Java is a non-deterministic language and when the garbage collector will engage is not possible to known. Comparing Rust and Swift performances, Rust is much faster than Swift even after the optimization.

Insert – Binary Tree	
Swift	9273.30323564288 ms
C	1026.59746275 ms
C++	932.358821428571 ms
Java	558.069735857143 ms
Rust	1655.36354007143 ms

Table 6.5: Insert 1,000,000 nodes – Binary Tree

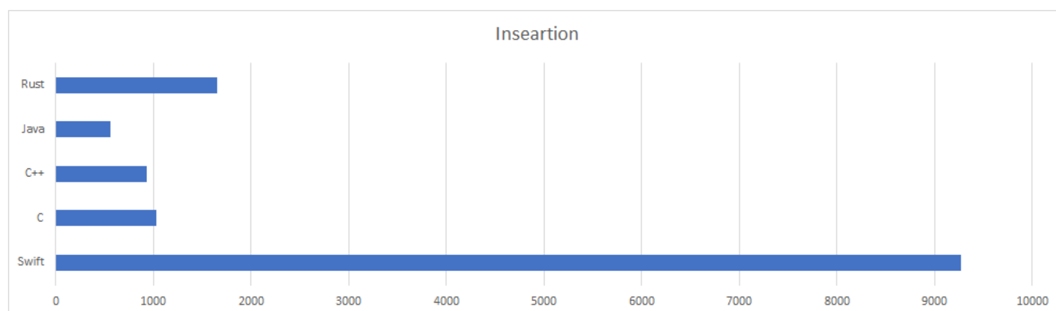

Figure 6.5: Insert 1,000,000 nodes – Binary Tree

Insert – Binary Tree	
Swift	5959.4845486429 ms
C	1026.59746275 ms
C++	932.358821428571 ms
Java	558.069735857143 ms
Rust	771.679387535714 ms

Table 6.6: Optimized Insert 1,000,000 nodes – Binary Tree

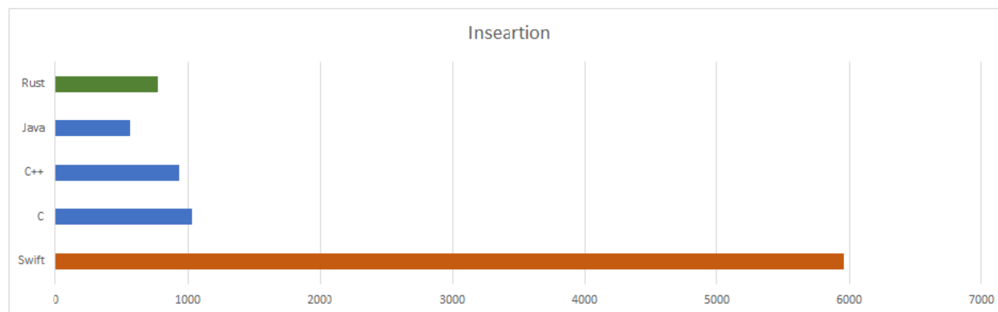

Figure 6.6: Optimized Insert 1,000,000 – Binary Tree

Increasing the number of allocations was used to observe the performance of the programs in the selected languages. Table 6.7 and Figure 6.7 present the average performance of 28 runs inserting 10,000,000 nodes in a binary tree. Table 6.8 and Figure 6.8 present the maximum performance of 28 runs of inserting 10,000,000 nodes. Beside average and maximum elapsed time, minimum elapsed times are presented in Table 6.9 and Figure 6.9.

Average Insert 10,000,000 nodes – Binary Tree	
Swift	114676.70168 ms
C	19932.7830675 ms
C++	20540.1428571429 ms
Java	8596.56452435715 ms
Rust	25134.8390402143 ms

Table 6.7: Average Insert 10,000,000 nodes – Binary Tree

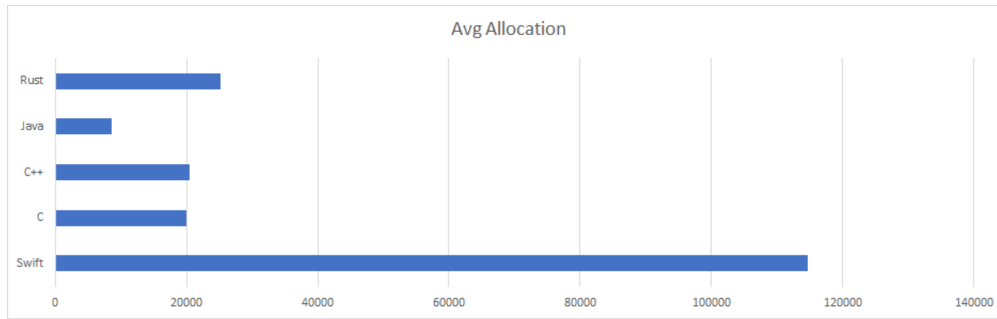

Figure 6.7: Avg Insert 10,000,000 – Binary Tree

Max Insert 10,000,000 nodes – Binary Tree	
Swift	129004.664062992 ms
C	69576.160086 ms
C++	75546.9 ms
Java	9553.740132 ms
Rust	25956.154696 ms

Table 6.8: Max Insert 10,000,000 nodes – Binary Tree

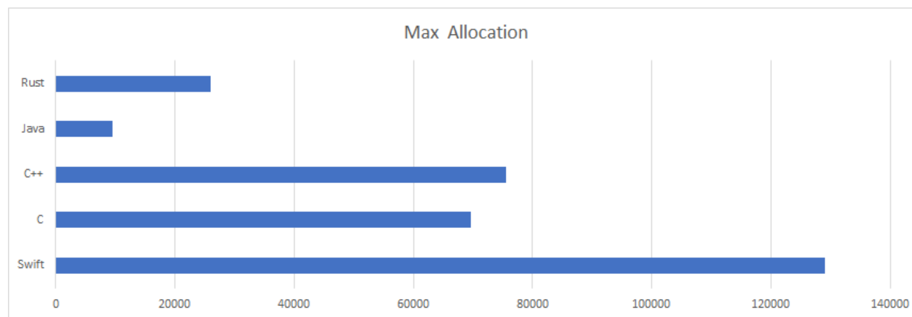

Figure 6.8: Max Insert 10,000,000 – Binary Tree

Min Insert 10,000,000 nodes – Binary Tree	
Swift	108086.51910001 ms
C	12713.426458 ms
C++	14191.5 ms
Java	7364.452619 ms
Rust	24599.767279 ms

Table 6.9: Min Insert 10,000,000 nodes – Binary Tree

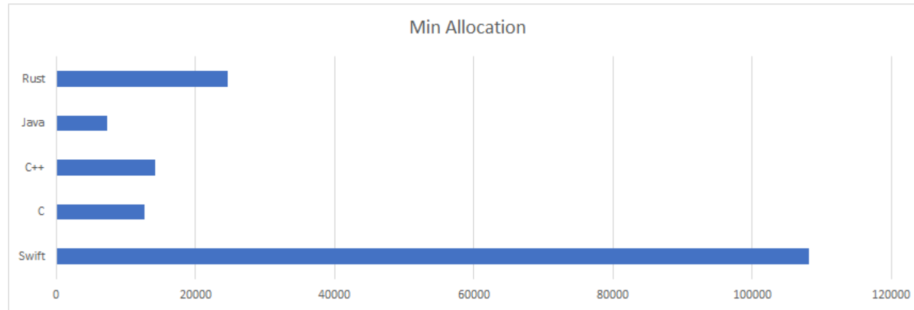

Figure 6.9: Min Insert 10,000,000 – Binary Tree

The experiment has been observed by increasing the number of nodes allocation to 50,000,000. In fact, manual memory management generated several errors and issues. For example, C gave "Segmentation fault: 11" run-time error approximately eight times out of ten attempts. One successful attempt shows, approximately (1010662.212087 ms) which is about 17 minutes to insert the desired number of nodes whereas Swift and Rust need (658737.423750013 ms) about 11 minutes and (163998.485697 ms) about 3 minutes respectively. It must also be noted that Rust allocation is much more significant than Swift in this part. However, Java still shows better performance with (81814.693349 ms) which is less than 2 minutes. On the other hand, C++ shows sharply slow performance in inserting 50 million nodes where (4.06993e+06 ms) about 68 minutes which is more than an hour. Table 6.10 and Figure 6.10 show the experiment of inserting 50 million nodes.

Insert 50,000,000 nodes – Binary Tree (Minutes)	
Swift	10.9789570625002 m
C	16.84437020145 m
C++	67.83216667 m
Java	1.36357822248333 m
Rust	2.73330809495 m

Table 6.10: Insert 50,000,000 nodes – Binary Tree (Minutes)

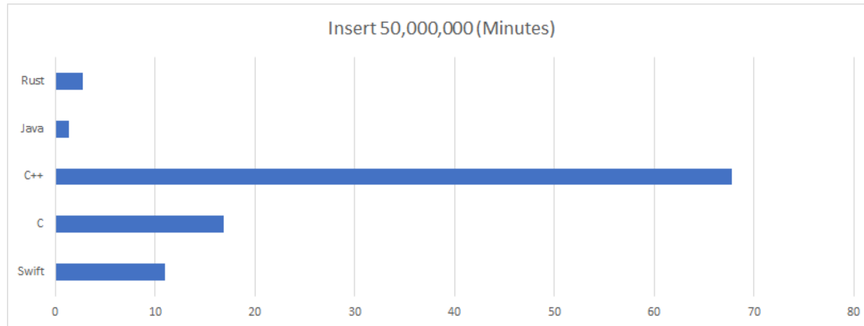

Figure 6.10: Insert 50,000,000 – Binary Tree (Minutes)

From the previous results, we can observe the influences of increasing the volume of the dynamic allocation. As a result, 100,000,000 nodes were used. In fact, that number of nodes became difficult to measure C and C++. Beside that, Java produced the error *"Exception in thread "main" java.lang.OutOfMemoryError: Java heap space"*. We succeeded only in the measurement for Swift and Rust by measuring the average of insertion of 100 million nodes, but the averaging was reduced to 5 attempts. Swift average time allocation needed about 23 minutes, while Rust needed about 6 minutes; in other words, allocation in Rust is approximately four times faster than Swift. Table 6.11 and Figure 6.11 present the results.

Avg Insert 100,000,000 nodes – Binary Tree		
Swift	1402980.5182958 ms	23.3830086382633 m
Rust	381652.289836 ms	6.36087149726666 m

Table 6.11: Avg Insert 100,000,000 nodes – Binary Tree

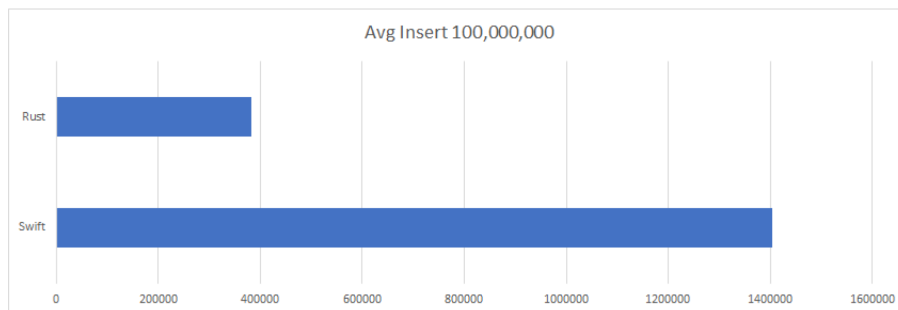

Figure 6.11: Insert 100,000,000 – Binary Tree

6.4.2 Replace

Beside allocation, deallocation is forced by replace of an existing node by a new one. The root replacement will be discarded. Figure 6.12, Figure 6.13 and Figure 6.14 show the idea of replacement. First, the binary tree has a random number of nodes, then suppose node with number 5 needs to be replaced. A new node of the same value will be created. The left and right nodes of the selected node will be assigned to a new node, and the parent node will be assigned to a new node as well. Finally, old node is required to be deleted explicitly in C and C++ while Java, Rust, and Swift the deallocation is handled implicitly.

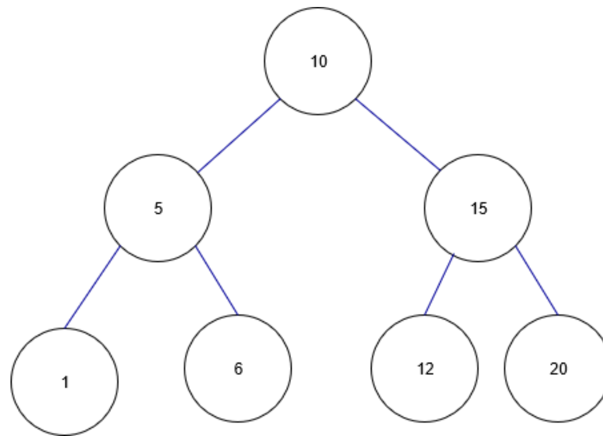

Figure 6.12: Binary Tree – random example

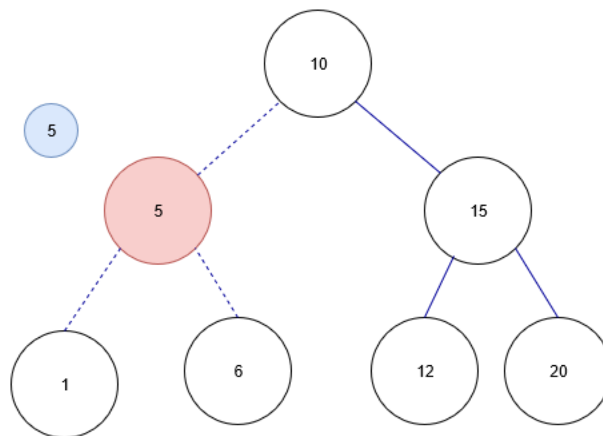

Figure 6.13: Binary Tree – selected random node

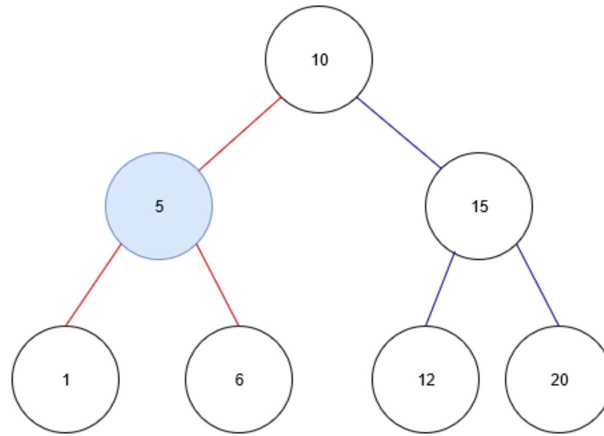

Figure 6.14: Binary Tree – recreate and replace

The average running of (replace) algorithm 28 times is measured with 10,000,000 nodes. In fact, Java shows the best performance among the other four selected languages whereas Rust has the slowest performance of deallocating 10 million nodes. In the deallocation part, Swift is performing faster than Rust and close to C performance while C++ is slower than C and Swift as well as Java. Table 6.12 and Figure 6.15 show the results.

Avg Replace – Binary Tree	
Swift	38865.051626678 ms
C	36973.6750028214 ms
C++	45142.8571428571 ms
Java	14797.2640756786 ms
Rust	48923.1860756072 ms

Table 6.12: Avg Replace 10,000,000 – Binary Tree

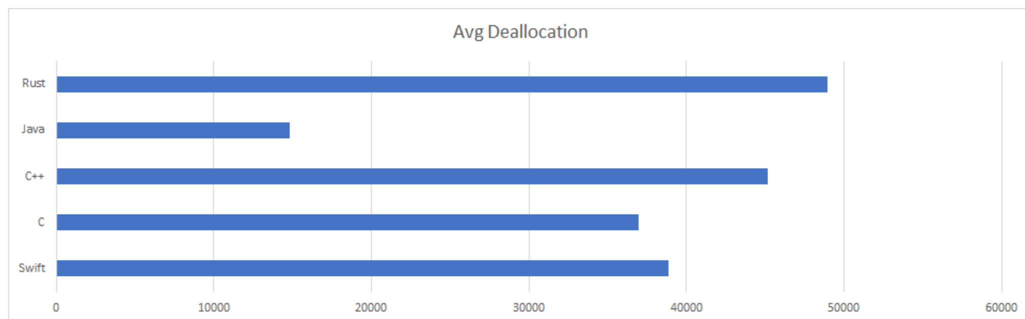

Figure 6.15: Avg Replace 10,000,000 – Binary Tree

The maximum time of measuring 28 running times shows C++ as the slowest language in deallocation. Even though Rust and C++ average measurements are close, the maximum measurement of each language shows a noticeable gap. C++ has stated the highest milliseconds for replacing 10 million nodes randomly. Similarly, Swift is showing a good performance compared to C with a maximum number of milliseconds, and Java still has the best performance among other languages. However, the unpredictable maximum measurement of replacement shows the effect when the number of nodes is increased.Table 6.13 and Figure 6.16 illustrate the results.

Max Replace – Binary Tree	
Swift	47114.8262329996 ms
C	128577.741138 ms
C++	176892 ms
Java	16513.708287 ms
Rust	51305.765053 ms

Table 6.13: Max Replace 10,000,000 – Binary Tree

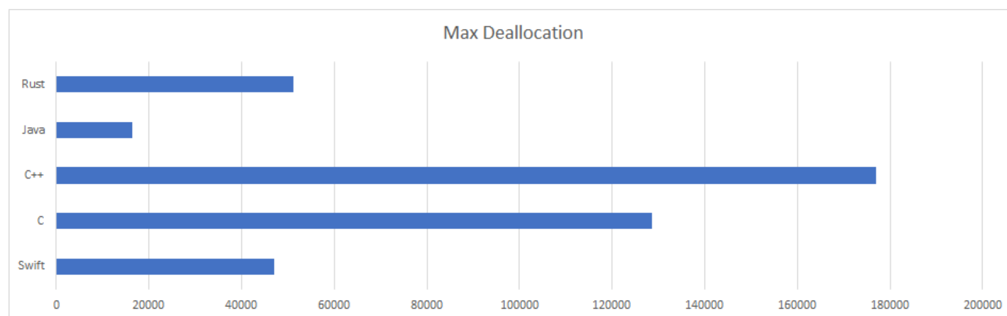

Figure 6.16: Max Replace 10,000,000 – Binary Tree

The minimum number of C++ is about 24 seconds, and the maximum number of C++ is about 177 seconds – 3 minutes. Similarly to the C language where the minimum is about 24 seconds, and the maximum is 128 second – 2 minutes. The gap between minimum and maximum is enormous. In contrast, Rust minimum is the highest compared to other languages, but the minimum number of Rust is about 45 seconds compared to the maximum time which is about 51 seconds. Swift minimum and maximum are 33 seconds and 47 seconds respectively, and Java minimum and maximum are 11

seconds and 16 seconds respectively. As a result, manual memory management shows a noticeable gap between minimum and maximum results whereas Swift, Rust, and Java are more stabilized. The following Table 6.14 and Figure 6.17 show the minimum measurements for the selected languages.

Min Replace – Binary Tree	
Swift	32980.7445499897 ms
C	24265.181879 ms
C++	24099.2 ms
Java	11471.549662 ms
Rust	45312.516293 ms

Table 6.14: Min Replace 10,000,000 – Binary Tree

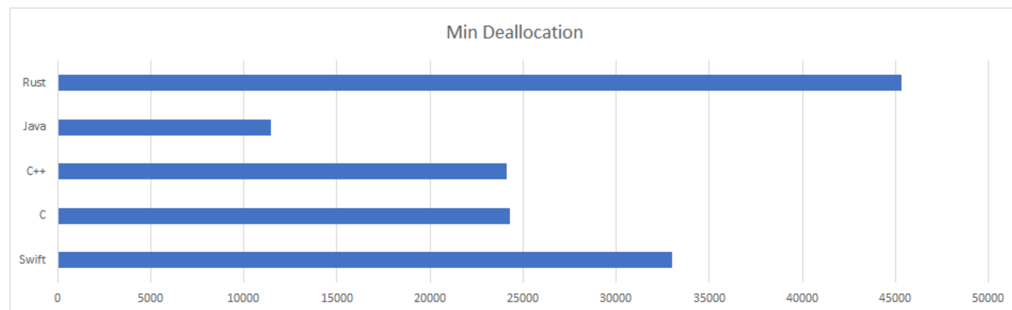

Figure 6.17: Min Replace 10,000,000 – Binary Tree

Similarly to allocation, deallocation – replace algorithm – was measured with 50,000,000 nodes. In C++, the deallocation is unspecified as the programs would not terminate correctly. Swift and Java show approximately the same performance (218031.831889004 ms) for about 4 minutes and (190440.675321 ms) about 3 minutes respectively. Swift in deallocation shows slightly better performance than Rust. In fact, Rust needs (323532.767015 ms) about 5 minutes, while Swift needs (218031.831889004 ms) about 4 minutes to replace 50 million nodes randomly. C and C++ show a sharp delay in – replace algorithm – when the number of nodes is increased. C has been completed by 50 minutes. Table 6.15 and Figure 6.18 show the results.

Replace 50,000,000 nodes – Binary Tree (Minutes)	
Swift	3.63386386481673 m
C	50.1281039668166 m
C++	Unspecified
Java	3.17401125535 m
Rust	5.39221278358333 m

Table 6.15: Replace 50,000,000 nodes – Binary Tree (Minutes)

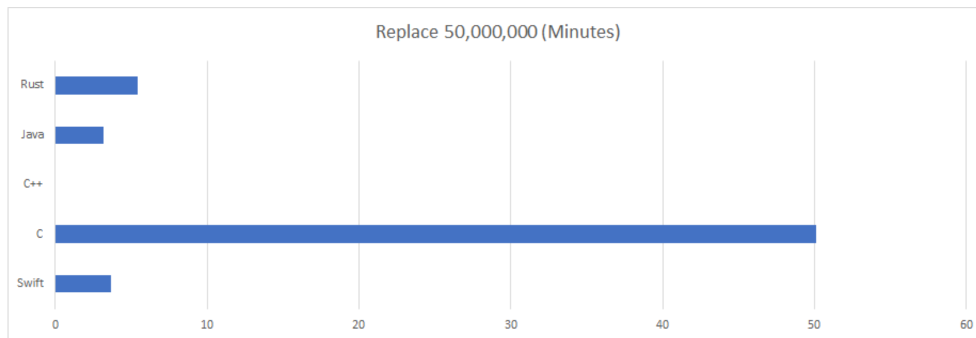

Figure 6.18: Replace 50,000,000 – Binary Tree (Minutes)

Then, 100,000,000 nodes were measured for the replace algorithms. Java, C, and C++ did not complete the experiment for the same reasons as when inserting 100 million nodes. Table 6.16 and Figure 6.19 show the average – Avg – performance of 28 times of – replace algorithm – running times. Swift completes the task with approximately 7 minutes and 12 minutes in Rust.Table 6.16 and Figure 6.19 show the result.

Avg Insert 100,000,000 nodes – Binary Tree		
Swift	436185.548425203 ms	7.26975914042005 m
Rust	727131.2228828 ms	12.1188537147133 m

Table 6.16: Avg Replace 100,000,000 nodes – Binary Tree

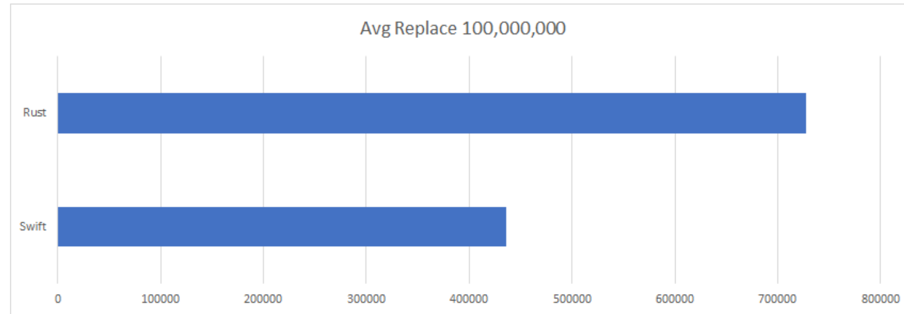

Figure 6.19: Replace 100,000,000 – Binary Tree

Therefore, conventional programming languages show a remarkable number of issues and restrictions related to memory management, while the recent programming languages – Rust and Swift – were able to manage a number of these issues, partially in Swift and fully in Rust, with the ability to handle large numbers of allocation and deallocation. However, we aimed to experience the run-time of different memory management designs with a finite number of nodes.

6.4.3 Search

Binary search tree is another attempt to check the performance of languages dynamically. We measure the performance of five selected languages in the Binary search tree of million nodes. Table 6.17 , Figure 6.20, Table 6.18 and Figure 6.21 present the performance before and after optimization, respectively.

Similar to Insert in binary tree, binary search tree in Rust is faster than Swift and after optimization, Rust shows significant performance gain in comparison to C and C++. Before optimization C, C++ and Java are performing faster than Swift and Rust. *Note: optimization is only applied to Swift and Rust*.

Search – Binary Tree	
Swift	4009.13879274999 ms
C	1219.26813632143 ms
C++	1227.66107142857 ms
Java	643.892571321429 ms
Rust	2286.76626814286 ms

Table 6.17: Search – Binary Tree

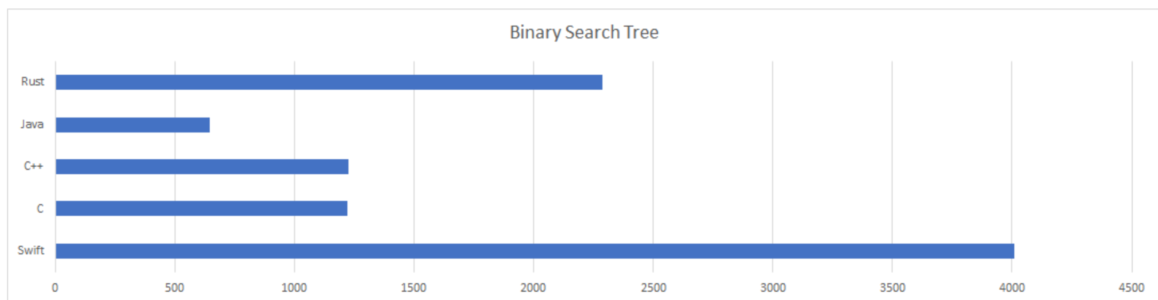

Figure 6.20: Search – Binary Tree

Search – Binary Tree	
Swift	1501.7445044288 ms
C	1219.26813632143 ms
C++	1227.66107142857 ms
Java	643.892571321429 ms
Rust	900.257434071429 ms

Table 6.18: Optimized Search – Binary Tree

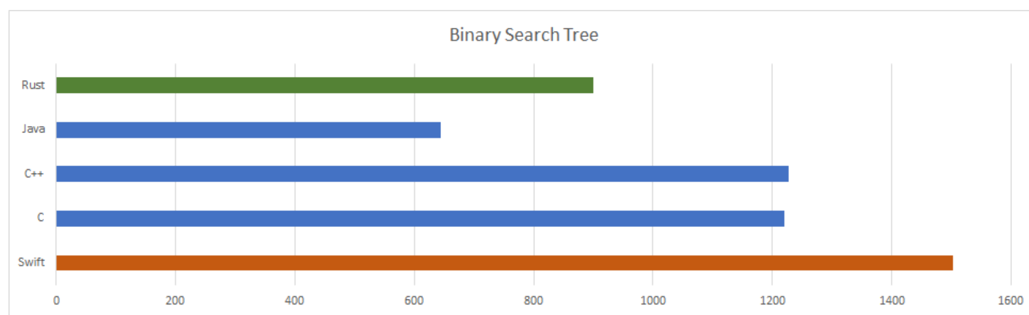

Figure 6.21: Optimized Search – Binary Tree

6.5 Array

This section presents the measuring of Quick sort and Binary search in five languages.The source code is in the Appendix A.

6.5.1 Quick Sort

In order to measure the performance, we selected one of the sort algorithms. Quick sort is a commonly used algorithm for sorting, it is faster in practice than other sort algorithms such as heapsort and merge sort. As the data structure to be sorted we chose Array.

Table 6.19 and Figure 6.22 present the performance of Quick sort before optimization while Table 6.20 and Figure 6.23 present the performance after optimization. The results show that the performance of Quick sort is quite similar between Swift and Rust. However, the performance of C,C++, and Java are significantly better than Swift and Rust before optimization.

Quick Sort	
Swift	1661.05913164286 ms
C	168.143928857143 ms
C++	182.263 ms
Java	114.041641642857 ms
Rust	1719.37826510714 ms

Table 6.19: Quick Sort

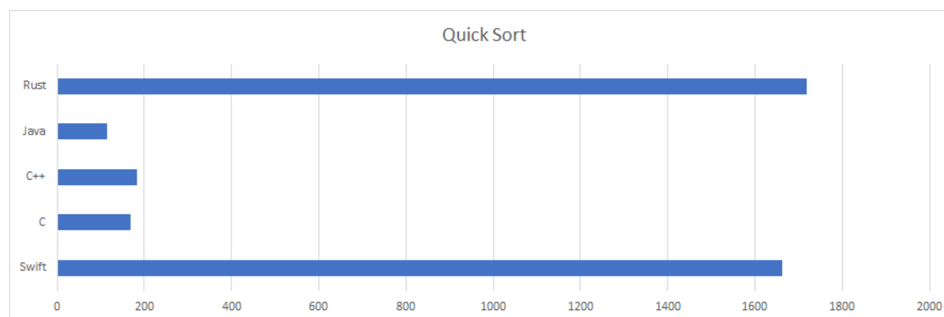

Figure 6.22: Quick Sort

Quick Sort	
Swift	114.153976678516 ms
C	168.143928857143 ms
C++	182.263 ms
Java	114.041641642857 ms
Rust	95.2023225714286 ms

Table 6.20: Optimized Quick Sort

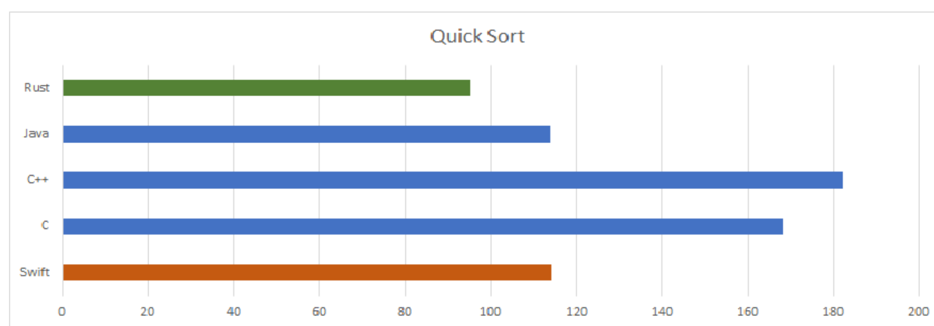

Figure 6.23: Optimized Quick Sort

6.5.2 Binary Search

Following the same procedure of the previous sections, we measured the performance of five languages specially in Binary search algorithm. Table 6.21, Figure 6.24, Table 6.22 and Figure 6.25 present the performance before and after optimization of binary search algorithms, respectively.

Binary Search for sorted array is similar to the performance of quick sort. Swift and Rust have approximately the same performance and after optimization Rust shows faster performing of binary search than Swift. Before optimization, the mainstream programming languages are better than Swift and Rust.

Binary Search – Sorted Array	
Swift	1056.32568567853 ms
C	419.856820678571 ms
C++	416.784464285714 ms
Java	204.298940071429 ms
Rust	907.54795625 ms

Table 6.21: Binary Search – Array

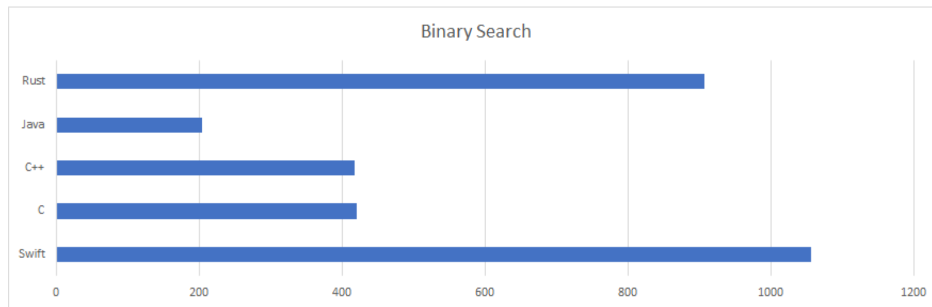

Figure 6.24: Binary Search – Array

Binary Search – Sorted Array	
Swift	408.007720821537 ms
C	419.856820678571 ms
C++	416.784464285714 ms
Java	204.298940071429 ms
Rust	221.840437571429 ms

Table 6.22: Optimized Binary Search – Array

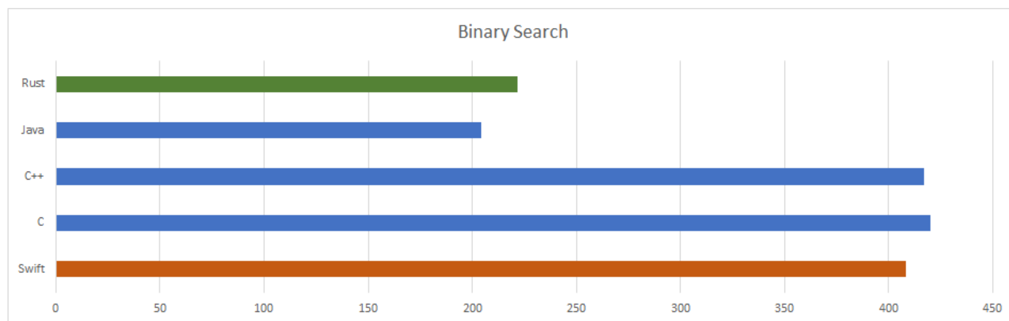

Figure 6.25: Optimized Binary Search – Array

6.6 Discussion

Evaluating performance of programs written in five different languages with different memory management was the purpose of this experiment. In the binary tree, the evaluation was balanced because classes in Swift are reference types, while `Option<Box<T>>` pointers, dynamic sized and heap allocated, are used in Rust. The same conditions were applied in C, C++, and Java by using *malloc* and class objects respectively. The performance of binary tree shows a remarkable significant comparison between Swift and Rust. In a binary tree, insert performance is noticeably faster than Swift, but Swift is faster than Rust in deallocation even with a large number of nodes. In addition, Java and Swift incur runtime costs, but Java still has a better performance with a restricted number of nodes, whereas Swift is capable of working with large number of nodes. Therefore, the dynamic performance in Rust and Swift compares favourably to the conventional programming languages. It must also be noted that Rust is more focused on memory safety than Swift. In addition, new programmers in Rust will have to adopt the new cognitive programming skills to be able writing abstract data types and its algorithms; it is seriously difficult to get successful compiled source codes – without taking into consideration the ownership, borrowing, mutability and lifetime rules.

In evaluating list either implemented as an array or vector on the experiment, the evaluation not only shows unbalanced results but, also the performances were varied. While Swift supports *array* as a value type and the size of the array is known at compile time; Rust array cannot be used because of *stack overflow* errors for 1,000,000 elements. On other hand, array in Swift has *use-on-write* optimization [50]. Sorting array of elements using the built-in functions provides significant performance in Swift[51, 52]. Even though Swift shows a significant speed of generating permutations using loops comparing to Rust, there is still a conflict between the implementation styles of vector and array allocation types. Therefore, the performance of Swift and Rust is not balanced, and it has been used for preparing the random numbers to support Binary tree structure. Besides that, the performance of quick sort and binary search of a sorted list is quite similar for the same reasons.

Chapter 7

Conclusion and Future Work

How to manage a memory resource is a significant issue in programming languages besides the safety and performance guarantees. Ownership rules were partially introduced in conventional programming language such as C++; for example $std :: unique_ptr$ and objects with shared ownership : $std :: shared_ptr$, but that is still not satisfying memory safety. It is always a question of whether to enhance old version of a language or introduce a full new aspect of management in a new language. Rust and Swift were introduce as new programming languages with novel memory management approaches.

Programmers using the mainstream programming languages such as C, C++, and Java have to pay steep costs for learning the new languages and developing new cognitive programming skills, especially for Rust. Rust is using composition pointers to satisfy the semantic guarantee of a specific requirement. The ownership and mutability are a strong part of resource declaration. The components *ownership, borrowing and lifetime* in rustc compiler implemented as a static sets of rules deal successfully with the typical memory issues be it memory leaking, dangling reference, or double dealocation. The readability of Rust code is a trade-off between effectiveness and efficiency. Unlike Swift, the syntax of Rust is pretty difficult such as the semantics and guarantees of composition pointers Rc<RefCell<vec<T>>> and Rc<vec<RefCell<T>>> are different. However, Rust provides high safety guarantees, comparing favourably with conventional programming languages.

Swift was designed to be a simple language solving the weakness of Objective-C. Swift is using ARC, Automatic Reference Counting, for memory management. ARC offers automatic memory deallocation. However, ARC still has an issue with strong reference cycles causing memory leaking. Even though ARC is an automatic memory management, Swift programmers need to be aware of lifetime qualifiers to resolve strong reference cycle issues.

Measurement comparisons of different programs working with different data structure and implementing various algorithms put Swift and Rust in a better perspective vis-a-vis conventional programming languages such as C, C++, and Java. Even though the results of simply comparing elapsed performance programs written in five different programming languages seems naive, they deserve to be taken seriously as they provide a simple gauge for comparison of the various language designs.

Comparing Rust and Swift to other programming languages is the next step of this research to answer questions concerning closures and concurrency. A more thorough evaluation needs not only more experiments, but also time as these languages are still under development. In order to find out the strength of ownership in memory design, a significant number of studies must be performed and experiences of developers must be assessed. Our research indicates that the ownership system as the base of the memory management merits further study and a serious consideration by developers in the industry and business.

Bibliography

[1] Robert W Sebesta and Soumen Mukherjee. *Concepts of programming languages*, volume 8. Addison-Wesley Reading, Massachusetts, 1999.

[2] Sudhir Sangappa, K Palaniappan, and Richard Tollerton. Benchmarking java against c/c++ for interactive scientific visualization. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 236–236. ACM, 2002.

[3] Sebastian Nanz and Carlo A Furia. A comparative study of programming languages in rosetta code. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 778–788. IEEE, 2015.

[4] Cristian González García, Jordán Pascual Espada, Begoña Cristina Pelayo García Bustelo, and Juan Manuel Cueva Lovelle. Swift vs. objective-c: A new programming language. *IJIMAI*, 3(3):74–81, 2015.

[5] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, (10):23–29, 2000.

[6] David Evans. Static detection of dynamic memory errors. In *ACM SIGPLAN Notices*, volume 31, pages 44–53. ACM, 1996.

[7] Michael R Genesereth. Software agents michael r. genesereth logic group computer science department stanford university. 1994.

[8] Narendran Sachindran and J Eliot B Moss. Mark-copy: Fast copying gc with less space overhead. In *ACM SIGPLAN Notices*, volume 38, pages 326–343. ACM, 2003.

[9] Julio CB Mattos and Luigi Carro. Object and method exploration for embedded systems applications. In *Proceedings of the 20th annual conference on Integrated circuits and systems design*, pages 318–323. ACM, 2007.

[10] Matthew Hertz and Emery D Berger. Automatic vs. explicit memory management: Settling the performance debate. *Technical Report CS*, 2004.

[11] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[12] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[13] James Goodwill and Wesley Matlock. *Beginning Swift Games Development for IOS*. Springer, 2015.

[14] Harwinder Singh. Speed performance between swift and objective-c.

[15] Yi Lin, Stephen M Blackburn, Antony L Hosking, and Michael Norrish. Rust as a language for high performance gc implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, pages 89–98. ACM, 2016.

[16] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66, 2017.

[17] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. System programming in rust: Beyond safety. *ACM SIGOPS Operating Systems Review*, 51(1):94–99, 2017.

[18] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. Oil and water? high performance garbage collection in java with mmtk. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 137–146. IEEE, 2004.

[19] Hans-J Boehm. Threads cannot be implemented as a library. In *ACM Sigplan Notices*, volume 40, pages 261–268. ACM, 2005.

[20] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee Jr, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *ACM SIGPLAN Notices*, volume 38, pages 324–337. ACM, 2003.

[21] Ivo Balbaert. *Rust Essentials*. Packt Publishing Ltd, 2015.

[22] The Rust Project Developers. Data types. `https://doc.rust-lang.org/book/2018-edition/ch03-02-data-types.html`, 2018. Accessed: 2018-03-15.

[23] The Rust Project Developers. Ownership. `https://doc.rust-lang.org/book/first-edition/ownership.html`, 2011. Accessed: 2018-03-25.

[24] The Rust Project Developers. Reference cycles can leak memory. `https://doc.rust-lang.org/book/second-edition/ch15-06-reference-cycles.html`, 2011. Accessed: 2018-04-16.

[25] The Rust Programming Language. Introduction. `https://doc.rust-lang.org/book/first-edition/index.html`, 2011. Accessed: 2018-03-16.

[26] the free encyclopedia Wikipedia. Swift (programming language). `https://en.wikipedia.org/wiki/Swift_(programming_language)`, 2018. Accessed: 2018-03-19.

[27] Waqar Malik. *Learn Swift on the Mac: For OS X and IOS*. Apress, 2015.

[28] Apple Inc. Foundation. `https://developer.apple.com/documentation/foundation`, 2018. Accessed: 2018-03-19.

[29] Apple Inc. Swift. https://developer.apple.com/documentation/swift, 2018. Accessed: 2018-03-19.

[30] Apple Inc. The swift programming language (swift 4.2). https://docs.swift.org/swift-book/, 2018. Accessed: 2018-03-19.

[31] Apple Inc. Collection types. https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html, 2018. Accessed: 2018-03-17.

[32] Apple Inc. Structures and classes. https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html#, 2018. Accessed: 2018-03-19.

[33] Apple Inc. Initialization. https://docs.swift.org/swift-book/LanguageGuide/Initialization.html, 2018. Accessed: 2018-03-19.

[34] Apple Inc. Deinitialization. https://docs.swift.org/swift-book/LanguageGuide/Deinitialization.html, 2018. Accessed: 2018-03-08.

[35] Apple Inc. Transitioning to arc release notes. https://developer.apple.com/library/content/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011226, 2013. Accessed: 2018-03-17.

[36] Apple Inc. Advanced memory management programming guide. https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html#//apple_ref/doc/uid/10000011i, 2012. Accessed: 2018-03-19.

[37] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William E. Lorensen, et al. *Object-oriented modeling and design*, volume 199. Prentice-hall Englewood Cliffs, NJ, 1991.

[38] Apple Inc. Automatic reference counting. https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html, 2018. Accessed: 2018-03-19.

[39] The Rust Programming Language. Characteristics of object-oriented languages. https://doc.rust-lang.org/book/second-edition/ch17-01-what-is-oo.html, 2011. Accessed: 2018-04-16.

[40] The Rust Programming Language. Running code on cleanup with the drop trait. https://doc.rust-lang.org/book/second-edition/ch15-03-drop.html, 2011. Accessed: 2018-03-11.

[41] GitHub Gankro/OwnershipTLDR. Swift ownership manifesto tl;dr. https://gist.github.com/Gankro/1f79fbf2a9776302a9d4c8c0097cc40e. Accessed: 2018-03-08.

[42] GitHub apple/swift. Ownership manifesto. https://github.com/apple/swift/blob/master/docs/OwnershipManifesto.md. Accessed: 2018-03-08.

[43] Nadeau Software Consulting (Dr. David R. Nadeau). C/c++ tip: How to measure elapsed real time for benchmarking. http://nadeausoftware.com/articles/2012/04/c_c_tip_how_measure_elapsed_real_time_benchmarking, 2012. Accessed: 2018-03-07.

[44] Microsoft. Queryperformancecounter function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx, 2018. Accessed: 2018-03-19.

[45] Nadeau Software Consulting (Dr. David R. Nadeau). Java tip: How to get cpu, system, and user time for benchmarking. http://nadeausoftware.com/articles/2008/03/java_tip_how_get_cpu_and_user_time_benchmarking, 2012. Accessed: 2018-03-07.

[46] The Rust Programming Language. Struct std::time::instant. https://doc.rust-lang.org/beta/std/time/struct.Instant.html, 2011. Accessed: 2018-03-19.

[47] David B Stewart. Measuring execution time and real-time performance. In *Embedded Systems Conference (ESC)*, volume 141, 2001.

[48] GitHub. Simple binary tree in rust. https://gist.github.com/DanielKeep/dcfadf1f4ea451e3cf6c, 2018. Accessed: 2017-06-8.

[49] GitHub. Rust binary tree worked example. https://gist.github.com/aidanhs/5ac9088ca0f6bdd4a370, 2018. Accessed: 2017-06-8.

[50] Apple Inc. Array. https://developer.apple.com/documentation/swift/array. Accessed: 2018-03-13.

[51] Kostiantyn Koval. *Swift High Performance*. Packt Publishing Ltd, 2015.

[52] Bikramjit Singh and Ramanjot Kaur. Raising performance of iphone using swift language over other programming languages. 2017.

Appendix A

Appendix

This Appendix contains the source codes of Array and Binary Tree.

A.1 Array – Source Code

- Swift

```
/**
 COPYRIGHT (c) 2018 Elaf Alhazmi . All Rights Reserved.
 SUPERVISED BY Dr.Emil Sekerinski and Dr.Frantisek Franek
 Swift Code Array
 */

#if os(Linux)
    import SwiftGlibc
    import Glibc
#endif
import Foundation
```

```
func shuffle (a: inout [Int], count_num: Int) {
    var r:Int
    var temp:Int
    srandom(UInt32(time(nil)))
    for i in 0..<a.count {
        #if os(Linux)
            r =  Int(random() % (count_num))
        #else
            r = Int(arc4random_uniform(UInt32(count_num)))
        #endif
        temp = a[i]
        a[i] = a[r]
        a[r] = temp
    }
}

func partition (a: inout [Int] ,start: Int , end: Int) ->
   Int {
    let x = a[end]
    var i = start
    for j in start..<end {
        if (a[j] <= x) {
            let temp = a[i]
            a[i] = a[j]
            a[j] = temp
            i = i+1
        }
```

```
        }
        let temp = a[i]
        a[i] = a[end]
        a[end] = temp

        return i
    }

    func quick_sort (a: inout [Int] , start: Int , end: Int ) {
        if (start < end) {
            let q = partition (a:&a , start: start , end: end)
            quick_sort (a:&a , start:start ,end: q-1)
            quick_sort (a:&a , start:q+1 ,end:end)
        }

    }

    func binarySearch (a: inout [Int] , key : Int) -> Int
    {
        var start = 0
        var end = a.count-1
        while (start <= end) {
            let mid = (start + end)/2
            if( key == a[mid]) {
                return mid
            }
            if (key < a[mid]) {
```

```
                end = (mid-1)
            }
            else {
                start = (mid+1)
            }
        }
        return -1
    }


    /**
     on OSX, mach_absolute_time() is used for measuring elpased
       time
     it is monotonic
     */
    func MAT() -> Double
    {
        var timeBaseInfo = mach_timebase_info_data_t()
        mach_timebase_info(&timeBaseInfo)
        return Double(mach_absolute_time() *
       UInt64(timeBaseInfo.numer)/
       (UInt64(timeBaseInfo.denom)))/1000000.0
    }
    /**
     Main performs four main tasks
     1.Generating Permutaion numbers using dynaimcally allocated
       array from [0-1000000)
     2.Using Fisher Yates algorithm to shuffle permutation
```

```
   numbers order
 3.Sorting array elaments using Quick Sort
 4.Binary search algorithm for sorted array
 */

let Num_of_Nodes = 1000000
var array_of_Rand_Num = Array(repeating:0, count:
   Num_of_Nodes)

/**
 Generating permutaion numbers by using loop in dynamic
   array allocation
 */
print("Loop : Generating Permutation random numbers \n")
print("Elapsed (MAT)")
print("----------------\n")
var start_MAT_loop1 = MAT()
for index in 0..<array_of_Rand_Num.count {
    array_of_Rand_Num[index] = index
}
var end_MAT_loop1 = MAT()
let diff_MAT_loop1 = end_MAT_loop1 - start_MAT_loop1;
print("\(diff_MAT_loop1) ms\t\t \n")

/**
 Shuflle permutaion numbers using Fisher Yates Algorithm
 */
```

```
print("Shuffle : Shuffle Permutation random numbers \n")
print("Elapsed (MAT)")
print("-----------\n")
var start_MAT_shuffle1 = MAT()
shuffle(a:&array_of_Rand_Num,count_num:Num_of_Nodes)
var end_MAT_shuffle1 = MAT()
let diff_MAT_shuffle1 = end_MAT_shuffle1 -
   start_MAT_shuffle1;
print("\(diff_MAT_shuffle1)\t\t \n")

/**
 Quick Sort Algorithm
 */
print("Quick Sort\n")
print("Elapsed (MAT) \t\t")
print("----------------\n")
var start_MAT_quick1 = MAT()
quick_sort(a:&array_of_Rand_Num, start:0 ,
   end:(array_of_Rand_Num.count-1))
var end_MAT_quick1 = MAT()
let diff_MAT_quick1 = end_MAT_quick1 - start_MAT_quick1;
print("\(diff_MAT_quick1)\t\t \n")

/**
 Binary Search Algorithm for Sorted Array
 */
print("Binary Search\n")
```

```
print("Elapsed (MAT) ")
print("------------\n")
srandom(UInt32(time(nil)))
var r:Int ;
var c:Int ;
var found = 0;
var not_found = 0;
var start_MAT_binary1 = MAT()
for _ in 0..<array_of_Rand_Num.count {
    #if os(Linux)
        r =  Int(random() % (Num_of_Nodes))
    #else
        r = Int(arc4random_uniform(UInt32(Num_of_Nodes)))
    #endif
    c = binarySearch (a:&array_of_Rand_Num, key:r)
    if (c != -1){
        found = found + 1;
    }else {
        not_found = not_found + 1;
    }
}
var end_MAT_binary1 = MAT()
let diff_MAT_binary1 = end_MAT_binary1 - start_MAT_binary1;
print("\(diff_MAT_binary1)\t\t \n")
print("Found = \(found) Not Found = \(not_found)")
```

- C

```
/**
COPYRIGHT (c) 2018 Elaf Alhazmi . All Rights Reserved.
SUPERVISED BY Dr.Emil Sekerinski and Dr.Frantisek Franek
C Code -- Array
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <mach/mach_time.h>

void shuffle(int *a, int count_num );
void quick_sort(int *a, int start, int end_index );
int partition_list(int *a, int start, int end_index);
int binarySearch(int *a , int key , int count_num) ;
double MAT ();

void shuffle(int *a, int count_num )
{
    int r;
    int temp;
    for (int i = count_num -1; i > 0; --i)
    {
        srandom(time(NULL));
        r = random() % (i+1) ;
```

```
        temp = a[i];
        a[i] = a[r];
        a[r] = temp ;
    }
}

void quick_sort(int *a, int start, int end_index )
{
    if (start < end_index)
    {
        int q = partition_list(a, start, end_index);
        quick_sort(a, start, q - 1);
        quick_sort(a, q + 1, end_index);
    }
}

int partition_list(int *a, int start, int end_index)
{
    int x = a[end_index];
    int i = start;
    int temp ;
    for (int j = start; j < end_index; j++)
    {
        if (a[j] <= x)
        {
            temp = a[i];
            a[i] = a[j];
```

```
                a[j] = temp ;
                i = i + 1;
            }
        }
         temp = a[i];
         a[i] = a[end_index];
         a[end_index] = temp ;
        return i;
    }

    int binarySearch(int *a , int key , int count_num)
    {
       int start =0;
       int end = count_num -1 ;
       while ( start <= end)
       { int mid = (start + end) / 2 ;
         if (key == a[mid] )
       {
         return mid ;
       }
         if (key < a[mid])
       {
         end = mid -1 ;
       }
         else
       {
         start = mid + 1 ;
```

```
    }
     }
 return -1 ;
}

/**
 on OSX, mach_absolute_time() is used for measuring elpased
   time
 it is monotonic
 */
double MAT ()
{
  mach_timebase_info_data_t start_timeBase_loop1;
    mach_timebase_info(&start_timeBase_loop1);
    return (((mach_absolute_time() *
   (double)start_timeBase_loop1.numer )/ (double)
   start_timeBase_loop1.denom ));
}

/**
 Main Function performs four main tasks
    1.Generating Permutation numbers -- array from
   [0-1000000)
    2.Using Fisher Yates algorithm to shuffle permutation
   numbers order
    3.Sorting array elaments using Quick Sort
    4.Binary search algorithm for sorted array
```

```
 */

int main()
{
    int Num_of_Nodes = 1000000;
    int *array_of_Rand_Num ;
    array_of_Rand_Num = (int *)
  malloc(Num_of_Nodes*sizeof(int));

    /**
     Generating permutation numbers
     */
    printf("Loop : Generating Permutaions of Numbers from 0
  - %i \n",Num_of_Nodes);
    printf("Elapsed (MAT) \n");
    printf("----------------\n");
    double elapsedTime_MAT_loop1 ;
    double start_timeBase_loop1 , end_timeBase_loop1;
    start_timeBase_loop1 = MAT() ;
    int i=0;
    while (i< Num_of_Nodes)
    {
        *(&array_of_Rand_Num[i]) = i;
        i++;
    }
    end_timeBase_loop1 = MAT ();
    elapsedTime_MAT_loop1 = (end_timeBase_loop1 -
```

```
start_timeBase_loop1)/ 1000000.0;
 printf("%fms\n",elapsedTime_MAT_loop1);

 /**
  Shuflle permutation numbers using Fisher Yates Algorithm
  */
 printf("\n\nShuffle : Shuffle Permutaion numbers \n");
 printf("Elapsed (MAT)\n");
 printf("---------------\n");
 double elapsedTime_MAT_shuffle1 ;
 double start_timeBase_shuffle1 , end_timeBase_shuffle1;
 start_timeBase_shuffle1 = MAT();
 shuffle (array_of_Rand_Num , Num_of_Nodes) ;
 end_timeBase_shuffle1 = MAT ();
 elapsedTime_MAT_shuffle1 = (end_timeBase_shuffle1 -
start_timeBase_shuffle1)/ 1000000.0;
 printf("%fms\n",elapsedTime_MAT_shuffle1);

 /**
  Quick Sort Algorithm
  */
 printf("\n\nQuick Sort \n");
 printf("Elapsed (MAT) \n");
 printf("--------------\n");
 double elapsedTime_MAT_quick1 ;
 double start_timeBase_quick1 , end_timeBase_quick1;
 start_timeBase_quick1 = MAT() ;
```

```
 quick_sort (array_of_Rand_Num, 0 , Num_of_Nodes-1);
 end_timeBase_quick1 = MAT ();
 elapsedTime_MAT_quick1 = (end_timeBase_quick1 -
start_timeBase_quick1)/ 1000000.0;
 printf("%fms\n", elapsedTime_MAT_quick1);

 /**
  Binary Search Algorithm for Sorted Array
  */
 printf("\n\n Binary Search Sort \n");
 printf("Elapsed (MAT) \n");
 printf("--------------\n");
 double elapsedTime_MAT_binary1 ;
 double start_timeBase_binary1 , end_timeBase_binary1;
 start_timeBase_binary1 = MAT() ;

 int c = 0;
 int found = 0;
 int not_found = 0;
 srandom(time(NULL));
 int r;
 for (int j = 0 ; j< Num_of_Nodes ; j++)
 {
     r = random() % Num_of_Nodes ;
     c = binarySearch ( array_of_Rand_Num , r ,
Num_of_Nodes);
     if (c != -1) {
```

```
            found++ ;
        }
        else {
            not_found++ ;
        }
    }
    end_timeBase_binary1 = MAT ();
    elapsedTime_MAT_binary1 = (end_timeBase_binary1 -
   start_timeBase_binary1)/ 1000000.0;
    printf("%fms \n", elapsedTime_MAT_binary1);
    printf("Found%i \n" ,found);
    printf("Not Found%i \n", not_found);
    free(array_of_Rand_Num);
    return 0;
}
```

- C++

```
    /**
 COPYRIGHT (c) 2018 Elaf Alhazmi . All Rights Reserved.
 SUPERVISED BY Dr.Emil Sekerinski and Dr.Frantisek Franek
 C++ Code -- Array
 */
#include <iostream>
#include <time.h>
#include <stdlib.h>
#include <sys/time.h>
```

```
#include <mach/mach_time.h>

using namespace std;

void Shuffle(int *a, int count_num);
void quick_sort(int *a, int start, int end_index );
int partition_list (int *a, int start, int end_index);
int binarySearch(int *a, int key, int size_of_array);
double MAT ();

void Shuffle(int *a, int count_num )
{
    int r;
    for (int i = count_num -1; i > 0; --i)
    {
        srandom(time(NULL));
        r = random() % (i+1) ;
        swap (a[i],a[r]);
    }
}

void quick_sort(int *a, int start, int end_index )
{
    if (start < end_index)
    {
        int q = partition_list(a, start, end_index);
        quick_sort(a, start, q - 1);
```

```
            quick_sort(a, q + 1, end_index);
        }
    }

    int partition_list(int *a, int start, int end_index)
    {
        int x = a[end_index];
        int i = start;
        for (int j = start; j < end_index; j++)
        {
            if (a[j] <= x)
            {
                swap (a[i], a[j]);
                i = i + 1;
            }
        }
        swap(a[i], a[end_index]);

        return i;
    }
    int binarySearch(int *a, int key, int size_of_array)
    {
         int start_num = 0;
            int end_num = size_of_array - 1;
            while (start_num <= end_num) {
                int mid_num = (start_num + end_num) / 2;
                if (key == a[mid_num]) {
```

```
                return mid_num;
            }
            if (key < a[mid_num]) {
                end_num = mid_num - 1;
            } else {
                start_num = mid_num + 1;
            }
        }
        return -1;
}
/**
 on OSX, mach_absolute_time() is used for measuring elpased
   time
 it is monotonic
 */
double MAT ()
{
  mach_timebase_info_data_t start_timeBase_loop1;
    mach_timebase_info(&start_timeBase_loop1);
    return (((mach_absolute_time() *
   (double)start_timeBase_loop1.numer )/ (double)
   start_timeBase_loop1.denom));
}

/**
 Main Function performs four main tasks
 1.Generating Permutation numbers using array from
```

```
   [0-1000000)
  2.Using Fisher Yates algorithm to shuffle permutation
   numbers order
  3.Sorting array elaments using Quick Sort
  4.Binary search algorithm for sorted array
  */

int main()
{
    int Num_of_Nodes = 1000000;
    int *array_of_Rand_Num ;
    array_of_Rand_Num = new int [Num_of_Nodes];

    /**
     Generating permutation numbers
     */
    cout<<"Loop : Generating Permutation Numbers from 0 -
   "<<Num_of_Nodes<<" \n";
    cout<<"Elapsed (MAT) \n";
    cout<<"----------------\n";
    double elapsedTime_MAT_loop1 ;
    double start_timeBase_loop1 , end_timeBase_loop1;
    start_timeBase_loop1 = MAT() ;
    for (int i=0 ; i< Num_of_Nodes ; i++)
    {
        array_of_Rand_Num [i] = i;
    }
```

```
 end_timeBase_loop1 = MAT ();
 elapsedTime_MAT_loop1 = (end_timeBase_loop1 -
start_timeBase_loop1)/ 1000000.0;
 cout<<elapsedTime_MAT_loop1<<" ms"<<endl ;

 /**
  Shuflle permutation numbers using Fisher Yates Algorithm
  */
 cout <<"\n\nShuffle : Shuffle Permutation number \n";
 cout<<"Elapsed (MAT) \n";
 cout<<"-------------------\n";
 double elapsedTime_MAT_shuffle1 ;
 double start_timeBase_shuffle1 , end_timeBase_shuffle1;
 start_timeBase_shuffle1 = MAT() ;
 Shuffle(array_of_Rand_Num, Num_of_Nodes);
 end_timeBase_shuffle1 = MAT ();
 elapsedTime_MAT_shuffle1 = (end_timeBase_shuffle1 -
start_timeBase_shuffle1)/ 1000000.0;
 cout<<elapsedTime_MAT_shuffle1<<" ms"<<endl ;

 /**
  Quick Sort Algorithm
  */
 cout<<"\n\nQuick Sort \n ";
 cout<<"Elapsed (MAT) \n";
 cout<<"-----------------\n";
```

```
double elapsedTime_MAT_quick1 ;
double start_timeBase_quick1 , end_timeBase_quick1;
start_timeBase_quick1 = MAT() ;
quick_sort(array_of_Rand_Num,0, Num_of_Nodes -1);
end_timeBase_quick1 = MAT ();
elapsedTime_MAT_quick1 = (end_timeBase_quick1 -
start_timeBase_quick1)/ 1000000.0;
cout<<elapsedTime_MAT_quick1<<" ms"<<endl ;

/**
 Binary Search Algorithm for Sorted Array
 */
cout<< "\n\nBinary Search \n ";
cout<<"Elapsed (MAT) \n";
cout<<"--------------------\n";
double elapsedTime_MAT_binary1 ;
double start_timeBase_binary1 , end_timeBase_binary1;
int c = 0;
int found = 0;
int not_found = 0;
start_timeBase_binary1 = MAT() ;
srandom(time(NULL));
int r;
for (int i = 0; i < Num_of_Nodes; i++)
{
    r = random() % Num_of_Nodes ;
    c = binarySearch (array_of_Rand_Num , r
```

```
,Num_of_Nodes);
        if (c != -1) {
            found++;
        } else {
            found++;
            not_found++;
        }
    }
    end_timeBase_binary1 = MAT ();
    elapsedTime_MAT_binary1 = (end_timeBase_binary1 -
start_timeBase_binary1)/ 1000000.0;
    cout<<elapsedTime_MAT_binary1<<" ms"<<endl ;
    cout<<"Found = " << found <<endl;
    cout<<"Not Found = " << not_found <<endl;
    delete [] array_of_Rand_Num;
    array_of_Rand_Num = NULL ;
    return 0;
}
```

- Java

```
/**
 * COPYRIGHT (c) 2018 Elaf Alhazmi . All Rights Reserved.
 * SUPERVISED BY Dr.Emil Sekerinski and Dr.Frantisek Franek
 * Java code - Array
 */

import java.time.Instant;
```

```
import java.util.Date;
import java.util.Random;

public class Array_measurment_java {

    public static void main(String[] args) {
        int Num_of_Nodes = 1000000;
        int[] array_of_Rand_Num = new int[Num_of_Nodes];

        /**
         Generating permutation numbers
         */
        System.out.println("Loop : Generating Permutation
   Numbers from 0 - " + Num_of_Nodes);
        System.out.println("Elapsed Time (NT)");
        System.out.println ("----------------");
        double start_NT_loop1 = System.nanoTime( );
        for (int i = 0; i < Num_of_Nodes; i++) {
            array_of_Rand_Num[i] = i;
        }
        double end_NT_loop1 = System.nanoTime( );
        double diff_NT_loop1 = end_NT_loop1 - start_NT_loop1;
        System.out.println ((diff_NT_loop1)/1000000.0D + "
   ms\t\t\t");

        /**
         Shuflle permutation numbers Fisher Yates Algorithm
```

```
         */
        System.out.println("\n\nShuffle : Shuffle
Permutation number");
        System.out.println("Elapsed Time (NT)");
        System.out.println ("----------------");
        double start_NT_shuffle1 = System.nanoTime( );
        shuffle(array_of_Rand_Num, Num_of_Nodes);
        double end_NT_shuffle1 = System.nanoTime( );
        double diff_NT_shuffle1 = end_NT_shuffle1 -
start_NT_shuffle1;
        System.out.println ((diff_NT_shuffle1)/1000000.0D +
" ms\t\t\t");

        /**
         Quick Sort Algorithm
         */
        System.out.println("\n\nQuick Sort");
        System.out.println("Elapsed Time (NT)");
        System.out.println ("----------------");
        double start_NT_quick1 = System.nanoTime( );
        quick_sort(array_of_Rand_Num, 0,
(array_of_Rand_Num.length - 1));
        double end_NT_quick1 = System.nanoTime( );
        double diff_NT_quick1 = end_NT_quick1 -
start_NT_quick1;
        System.out.println ((diff_NT_quick1)/1000000.0D + "
ms\t\t\t" );
```

```
/**
 Binary Search Algorithm for Sorted Array
 */
System.out.println("Binary Search");
System.out.println("Elapsed Time (NT)");
System.out.println ("----------------");
double start_NT_binary1 = System.nanoTime( );
Random random_num = new Random();
int c = 0;
int found = 0;
int not_found = 0;
for (int j = 0; j < Num_of_Nodes; j++) {
    c = binarySearch(array_of_Rand_Num,
random_num.nextInt(Num_of_Nodes));
    if (c != -1) {
        found++;
    } else {
        not_found++;
    }
}
double end_NT_binary1 = System.nanoTime( );
double diff_NT_binary1 = end_NT_binary1 -
start_NT_binary1;
System.out.println ((diff_NT_binary1)/1000000.0D + "
ms\t\t\t");
System.out.println("\nFound = " + found);
```

```
        System.out.println("Not Found = " + not_found);

    }

    public static int partition(int[] a, int start, int end)
    {
        int x = a[end];
        int i = start;
        for (int j = start; j < end; j++) {
            if (a[j] <= x) {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
                i = i + 1;
            }
        }
        int temp = a[i];
        a[i] = a[end];
        a[end] = temp;

        return i;

    }

    public static void quick_sort(int[] a, int start, int
    end) {
        if (start < end) {
```

```
            int q = partition(a, start, end);
            quick_sort(a, start, q - 1);
            quick_sort(a, q + 1, end);
        }
    }

    public static void shuffle(int[] a, int count_num) {
        int r;
        int temp;
        for (int i = count_num - 1; i > 0; --i) {
            Random r1 = new Random();
            r = r1.nextInt(count_num);
            temp = a[i];
            a[i] = a[r];
            a[r] = temp;
        }
    }

    public static int binarySearch(int[] a, int key) {

        int start = 0;
        int end = a.length - 1;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (key == a[mid]) {
                return mid;
            }
```

```
                if (key < a[mid]) {
                    end = mid - 1;
                } else {
                    start = mid + 1;
                }
            }
            return -1;
        }
}
```

- Rust

```
/**
 COPYRIGHT (c) 2018 Elaf Alhazmi . All Rights Reserved.
 SUPERVISED BY Dr.Emil Sekerinski and Dr.Frantisek Franek
 Rust Code - Vector
 */

extern crate rand;
extern crate time;
use std::time::{Duration, Instant};
use time::PreciseTime;
use rand::{thread_rng, Rng};

fn shuffle_x(a: &mut[i32], count_num: usize)
{
   let mut r :usize;
```

```
   let mut temp;
   let mut rng = thread_rng();
   for i in 1..count_num
   {
    r = rng.gen_range(0, i);
    temp = a[i];
         a[i] = a[r];
         a[r] = temp;
   }
}

fn quick_sort(a: &mut[i32], start: usize , end: usize) {
   if start >= end {
   return
   }
   else{
   let  q = partition ( a , start , end);
   if q > 0 {
   quick_sort(a,start, q-1);
   }
   quick_sort(a,q+1,end);
   }
}

fn partition(a: &mut[i32], start: usize , end: usize) ->
   usize {
   let  x = a[end];
```

```
    let mut i =  start ;
    for j in start..end {
       if a[j] <= x {
       let temp = a[i];
       a[i] = a[j];
       a[j] =temp;
       i=i+1;
       }
    }
   let temp = a[i];
    a[i] = a[end];
    a[end] =temp;
    return i;
 }

 fn binarySearch (a: &mut[i32] , key: i32 ) -> i32 {
    let mut start = 0 ;
    let mut end = a.len() -1 ;

    while start <= end
    {
       let mut mid: usize = (start + end) / 2  as usize ;
       if  key == a[mid] {
       return mid as i32 ;
       }
       if key < a[mid]  {
       end = mid -1 ;
```

```
        }
        else {
        start = mid + 1 ;
        }
    }
     return -1 ;
}

/**
 Main Function performs four main tasks
    1.Generating Permutation numbers from [0-1000000)
    2.Using Fisher Yates algorithm to shuffle permutation
   numbers order
    3.Sorting array elaments using Quick Sort
    4.Binary search algorithm for sorted array
 */
fn main() {
    const Num_of_Nodes:usize = 1000000;
    let mut array_of_Rand_Num = vec![0; Num_of_Nodes];

     /**
     Generating permutaion numbers
     */
    println!("Loop : Generating Permutation Numbers from 0 -
   {} \n",Num_of_Nodes);
    println!("Elapsed Time (Instant) \t Elapsed Time
   (Precise) \n");
```

```
 println!("------------------------------\n");
let start_Precise_loop1 = PreciseTime::now();
let start_instant_loop1 = Instant::now();
for i in 0..Num_of_Nodes {
array_of_Rand_Num[i] = i as i32;
}
let end_instant_loop1 = start_instant_loop1.elapsed();
let end_Precise_loop1 = PreciseTime::now();
let diff_instant_loop1 = (end_instant_loop1.as_secs() as
f64)*1000.0 + (end_instant_loop1.subsec_nanos() as f64 /
1000000.0);
println!(" {} ms \t\t {} s",diff_instant_loop1,
start_Precise_loop1.to(end_Precise_loop1));

/**
  Shuflle permutation numbers using Fisher Yates Algorithm
  */
println!("Shuffle : Shuffle Permutation Numbers");
 println!("Elapsed Time (Instant) \t Elapsed Time
(Precise) \n");
 println!("---------------------------------\n");
let start_Precise_shuffle1 = PreciseTime::now();
let start_instant_shuffle1 = Instant::now();
shuffle_x (&mut array_of_Rand_Num , Num_of_Nodes );
let end_instant_shuffle1 =
start_instant_shuffle1.elapsed();
let end_Precise_shuffle1 = PreciseTime::now();
```

```
let diff_instant_shuffle1 =
(end_instant_shuffle1.as_secs() as f64)*1000.0 +
(end_instant_shuffle1.subsec_nanos() as f64 / 1000000.0);
println!(" {} ms \t\t {} s",diff_instant_shuffle1,
start_Precise_shuffle1.to(end_Precise_shuffle1));
let mut size_of_array = Num_of_Nodes;
if size_of_array > 0
{
size_of_array = size_of_array -1 ;
}


/**
  Quick Sort Algorithm
  */
println!("Quick Sort");
 println!("Elapsed Time (Instant) \t Elapsed Time
(Precise) \n");
 println!("-------------------------------\n");
let start_Precise_quick1 = PreciseTime::now();
let start_instant_quick1 = Instant::now();
quick_sort (&mut array_of_Rand_Num , 0 , size_of_array) ;
let end_instant_quick1 = start_instant_quick1.elapsed();
let end_Precise_quick1 = PreciseTime::now();
let diff_instant_quick1 = (end_instant_quick1.as_secs()
as f64)*1000.0 + (end_instant_quick1.subsec_nanos() as
f64 / 1000000.0);
println!(" {} ms \t\t {} s",diff_instant_quick1,
```

```
start_Precise_quick1.to(end_Precise_quick1));

/**
  Binary Search Algorithm for Sorted Array
  */
println!("Binary Search");
 println!("Elapsed Time (Instant) \t Elapsed Time
(Precise) \n");
 println!("--------------------------------\n");
let mut rng = thread_rng();
let mut found = 0 ;
let mut not_found = 0;
let start_Precise_binary1 = PreciseTime::now();
let start_instant_binary1 = Instant::now();
for _ in 0..Num_of_Nodes {
let n: i32 = rng.gen_range(0, Num_of_Nodes as i32);
let  c = binarySearch (&mut array_of_Rand_Num , n) ;
if c != -1
{found = found +1;}
else
{ not_found = not_found + 1; }
}
let end_instant_binary1 = start_instant_binary1.elapsed();
let end_Precise_binary1 = PreciseTime::now();
let diff_instant_binary1 = (end_instant_binary1.as_secs()
as f64)*1000.0 + (end_instant_binary1.subsec_nanos() as
f64 / 1000000.0);
```

```
    println!(" {} ms \t\t {} s",diff_instant_binary1,
    start_Precise_binary1.to(end_Precise_binary1));
    println! ("\nFound ={} and not Found ={}" ,
    found,not_found ) ;
}
```

A.2 Tree - Source Codes

- Swift

```
/**
 COPYRIGHT (c) 2018 Elaf Alhazmi . All Rights Reserved.
 SUPERVISED BY Emil Sekerinski and Frantisek Franek
 Swift Code - Binary Tree
 */
#if os(Linux)
    import SwiftGlibc
    import Glibc
#endif
import Foundation

func shuffle (a: inout [Int], count_num: Int) {
    var r:Int
    var temp:Int
    srandom(UInt32(time(nil)))
    for i in 0..<a.count {
        #if os(Linux)
            r =  Int(random() % (count_num))
        #else
            r = Int(arc4random_uniform(UInt32(count_num)))
        #endif
        temp = a[i]
        a[i] = a[r]
        a[r] = temp
```

```
        }
    }

    /**
     Node Structure
     */
    class Node {
        var key : Int
        var left : Node?
        var right : Node?

        init ()
        {
            self.key = 0
            self.left = nil
            self.right = nil
        }
        init (value : Int)
        {
            self.key = value
            self.left = nil
            self.right = nil
        }

        /*deinit
         {
         print("Node is deleted!\n");
```

```
        }*/
    }

    /**
     Swift class to create Binary Tree Abstract Data Type
     and its operations
     */
    class BT {
        var root : Node?
        var created : Int = 0 ;
        var not_created : Int = 0 ;

        init ()
        {
            root = nil
        }

        /**
         Function insert adds new node in tree
         Note: "value" is the key value for new node
         */
        func insert (node: Node? , value : Int)
        {
            let temp =  Node()
            temp.key = value
            temp.left = nil
            temp.right = nil
```

```
if (root == nil)
{
    root = Node()
    root?.key = value
    root?.left = nil
    root?.right = nil
    created = created + 1
    return
}
else
{
    if (value < (node!.key))
    {
        if(node?.left != nil)
        {
            insert(node:node!.left, value: value)
        }
        else
        {
            node!.left = temp
            created = created + 1
            return
        }
    }
    if (value > node!.key)
    {
        if (node?.right != nil)
```

```
                {
                    insert(node:node!.right, value: value)
                }
                else
                {
                    node!.right = temp
                    created = created + 1
                    return
                }
            }
            if ( value == node!.key)
            {
                not_created = not_created + 1
                return
            }
        }
    }//end insert function

    /**
     Function search checks if "value" is exist in Tree
     */
    func search (node: Node? , value: Int) -> Bool
    {
        if( root == nil )
        {
            print(" Root is nil")
            return false
```

```
        }
        if (node == nil)
        {
            return false
        }
        else
        {
            if ( node!.key == value)
            {
                return true
            }
            if (value < node!.key)
            {
                return search(node:node?.left, value: value)
            }
            else
            {
                return  search (node:node?.right, value:
    value)
            }
        }
     }// end search function

     /**
      Function replace node by creating new node
    (deallocation purposes)
      */
```

```
func replace (node: Node? , item: Int)
{
    var current:Node? ; var parent:Node?; var temp:Node?
    current = root
    parent = root
    temp = Node();

    while(current != nil)
    {
        parent = current;
        if(item < current!.key) {
            current = current!.left;

            if (item == current?.key)
            {
                self.swap_node(parent: parent, current:
current, temp: temp);
                break;
            }
        }
        else if (item > current!.key)
        {
            current = current!.right;
            if (item == current?.key)
            {
                self.swap_node(parent: parent, current:
current, temp: temp);
```

```
                    break;
                }
            }
            else
            {
                // in case if the Node is root
                // no replacement will be occured
                break;
            }

        }// end while
        current = nil
        parent = nil

    }// end replace function

    /**
     Function swap_node by replacing old node by new node
     */
    func swap_node(parent: Node? , current: Node? ,
   temp:Node?)
    {
        if (current === root){
            return;
        }
        if (parent!.key > current!.key) {
            parent!.left = temp;
```

```
                temp!.left = current!.left;
                temp!.right = current!.right;
                current!.left = nil;
            }
            else if (parent!.key < current!.key)
            {
                parent!.right = temp;
                temp!.right = current!.right;
                temp!.left = current!.left;
                current!.left = nil;
                current!.right = nil;
            }
        }

        func inorder(item: Node?)
        {
            if ( item != nil  )
            {
                inorder(item:item!.left)
                print (" value \(item!.key)")
                inorder(item:item!.right)
            }
        }//end function inorder
    }// End class BT

    /**
     on OSX, mach_absolute_time() is used for measuring elpased
```

```
   time
 it is monotonic
 */
func MAT() -> Double
{
    var timeBaseInfo = mach_timebase_info_data_t()
    mach_timebase_info(&timeBaseInfo)
    return Double(mach_absolute_time() *
   UInt64(timeBaseInfo.numer)/
   (UInt64(timeBaseInfo.denom)))/1000000.0
}

/**
 Main Function performs five main tasks
 1.Generating Permutation numbers from [0-Num_of_Nodes)
 2.Using Fisher Yates algorithm to shuffle permutation
   numbers order
 3.Insert - Num_of_Nodes - in Binary Tree
 4.Search - Num_of_Nodes - in Binary Tree
 5.Replace - Num_of_Nodes - in Binary Tree
 */

let Num_of_Nodes = 10000000
var array_of_Rand_Num = Array(repeating:0, count:
   Num_of_Nodes)

/**
```

```
 Generating permutation numbers
 */
print("Loop : Generating Permutation random numbers \n")
print("Elapsed (MAT)")
print("--------------------------------\n")
var start_MAT_loop1 = MAT()
for index in 0..<array_of_Rand_Num.count {
    array_of_Rand_Num[index] = index
}
var end_MAT_loop1 = MAT()
let diff_MAT_loop1 = end_MAT_loop1 - start_MAT_loop1;
print("\(diff_MAT_loop1)\t\t \n")

/**
 Shuflle permutation numbers Fisher Yates Algorithm
 */
print("Shuffle : Shuffle Permutation random numbers \n")
print("Elapsed (MAT)")
print("----------------------\n")
var start_MAT_shuffle1 = MAT()
shuffle(a:&array_of_Rand_Num,count_num:Num_of_Nodes)
var end_MAT_shuffle1 = MAT()
let diff_MAT_shuffle1 = end_MAT_shuffle1 -
   start_MAT_shuffle1;
print("\(diff_MAT_shuffle1)\t\t \n")
var bt = BT()
```

```
/**
 Insert
 */
print("Insert\n")
print("Elapsed (MAT) \t\t")
print("----------------------\n")
var start_MAT_insert1 = MAT()
for  i in 0...(Num_of_Nodes-1)
{
    bt.insert(node:bt.root, value: array_of_Rand_Num[i])
}
var end_MAT_insert1 = MAT()
let diff_MAT_insert1 = end_MAT_insert1 - start_MAT_insert1;
print("\(diff_MAT_insert1)\t\t \n")

/**
 Search
*/
print("Binary Search Tree\n")
print("Elapsed (MAT) \t\t ")
print("-------------------------\n")
var r:Int ;
var found = 0;
var not_found = 0;
srandom(UInt32(time(nil)))
var start_MAT_binary1 = MAT()
for _ in 0...(Num_of_Nodes-1)
```

```
{
    #if os(Linux)
        r =  Int(random() % (Num_of_Nodes))
    #else
        r = Int(arc4random_uniform(UInt32(Num_of_Nodes)))
    #endif
    let b = bt.search(node:bt.root, value:r)
    if (b == true){
        found = found + 1;
    }else {
        not_found = not_found + 1;
    }
}
var end_MAT_binary1 = MAT()
let diff_MAT_binary1 = end_MAT_binary1 - start_MAT_binary1;
print("\(diff_MAT_binary1)\t\t \n")


/**
 Replace
 */
print("Binary Tree - Replacement\n")
print("Elapsed (MAT) \t\t ")
print("-------------------------\n")
var z:Int ;
srandom(UInt32(time(nil)))
var start_MAT_replace1 = MAT()
for _ in 0...(Num_of_Nodes-1)
```

```
{
    #if os(Linux)
        z =  Int(random() % (Num_of_Nodes))
    #else
        z = Int(arc4random_uniform(UInt32(Num_of_Nodes)))
    #endif
    bt.replace(node:bt.root, item:z)
}

var end_MAT_replace1 = MAT()
let diff_MAT_replace1 = end_MAT_replace1 -
   start_MAT_replace1;
print("\(diff_MAT_replace1)\t\t \n")
print ("Nodes created \(bt.created)")
print ("Nodes not created \(bt.not_created)")
```

- C

```
/**
 COPYRIGHT (c) 2018 Elaf Alhazmi . All Rights Reserved.
 SUPERVISED BY Dr.Emil Sekerinski and Dr.Frantisek Franek
 C Code - Binary Tree
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#include <sys/time.h>
#include <mach/mach_time.h>

void shuffle(long *a, long count_num );
double MAT ();

typedef struct BTNode Node;
struct BTNode
{
    long key;
    Node *left;
    Node *right;
};
void swap_node ( Node *root, Node *parent, Node *current ,
   Node *temp);

void shuffle(long *a, long count_num )
{
    int r;
    int temp;
    for (int i = count_num -1; i > 0; --i)
    {
        srandom(time(NULL));
        r = random() % (i+1) ;
        temp = a[i];
        a[i] = a[r];
        a[r] = temp ;
```

```
    }
}

//Global Variables
long Nodes_created = 0 ;
long Nodes_not_created =0;

/**
 Function insert() returns inserted Node pointer
 Note:   if the value n is already exist in Binary Tree,
 new node will not be dublicated.
 Param:  Node *root, long n
 */
Node* insert (Node *root, long n )
{
    if (root == NULL)
    {
        root= malloc(sizeof(Node));
        root->key = n ;
        root->left = NULL ;
        root->right = NULL ;
        Nodes_created= Nodes_created + 1 ;
    }
    else
    {
        if (n == root->key)
        {
```

```
            Nodes_not_created = Nodes_not_created + 1;
            return root;
        }
        else if (n < root->key)
        {
            root->left = insert (root->left, n);
        }
        else
        {
            root->right = insert (root->right, n);
        }
    }
    return root ;
}

/**
 Function search returns int, either 0 or 1
 Note:  search function returns 1
 if  item is found or 0 if not
 Param :  long item, Node *par,
 */
int search (long item, Node *par)
{
    int flag = 0;
    if (par != NULL)
    {
        if (par->key == item)
```

```
            {
                return 1;
            }
            else
            {
                if(item  < par->key)
                {
                    flag = search (item, par->left);
                }
                else
                {
                    flag = search (item, par->right);
                }
            }
        }
        return flag;
    }

    /**
     Function replace
     Param :  long item, Node *root,
     */
    void replace (long item, Node *root)
    {
        Node *parent, *current, *temp;
        current = root ;
        parent = root ;
```

```
temp= (Node *)malloc(sizeof(Node));
temp->key = item ;
temp->left = NULL ;
temp->right = NULL ;

while(current != NULL)
{
    parent = current;
    if (item < current->key) {
        current = current->left;
        if(item == current->key) {
          swap_node( root, parent, current, temp);
            break;
        }
    }
    else if (item > current->key) {
        current = current->right;
        if(item == current->key) {
           swap_node(root, parent, current, temp);
            break;
        }
    }
    else {
        break;
    }
}
```

```
}

/**
 Function swap_node
 Param :  Node *root,Node *parent, Node *current, Node *temp
 */
void swap_node ( Node *root, Node *parent, Node *current ,
   Node *temp)
{
    if (current == root) {
        return;
    }

    if (parent->key > current->key) {

        parent->left = temp;
        temp->left = current->left;
        temp->right = current->right;
        free(current);

    }
    else{
        if ( parent->key < current->key){
        parent->right = temp;
        temp->right = current->right;
        temp->left = current->left;
        free(current);
```

```
        }
    }
}

/**
 Function inorder
 Param :  Node *temp, Node *current, Node *temp
 */
void inorde_node ( Node *temp)
{
    if (temp!= NULL) {
        inorde_node(temp->left);
        printf("value %ld \n",temp->key);
        inorde_node(temp->right);
    }
}

/**
 on OSX, mach_absolute_time() is used for measuring elpased
   time
 it is monotonic
 */
double MAT ()
 {
 mach_timebase_info_data_t start_timeBase_loop1;
 mach_timebase_info(&start_timeBase_loop1);
 return (((mach_absolute_time() *
```

```
   (double)start_timeBase_loop1.numer )/ (double)
   start_timeBase_loop1.denom ));
  }

/**
 Main Function performs five main tasks
 1.Generating Permutation numbers from [0-Num_of_Nodes)
 2.Using Fisher Yates algorithm to shuffle permutation
   numbers order
 3.Insert - Num_of_Nodes - in Binary Tree
 4.Search - Num_of_Nodes - in Binary Tree
 5.Replace - Num_of_Nodes - in Binary Tree
 */

int main()
{
    long Num_of_Nodes = 10000000;
    long *array_of_Rand_Num ;
    array_of_Rand_Num = (long
   *)malloc(Num_of_Nodes*sizeof(long));

    /**
     Generating permutation numbers
     */
    printf("Loop : Generating Permutation Numbers from 0 -
   %ld \n",Num_of_Nodes);
    printf("Elapsed (MAT) \n");
```

```
    printf("--------------\n");
    double elapsedTime_MAT_loop1 ;
    double start_timeBase_loop1 , end_timeBase_loop1;
    start_timeBase_loop1 = MAT() ;
    int i=0;
    while (i< Num_of_Nodes)
    {
        *(&array_of_Rand_Num[i]) = i;
        i++;
    }
    end_timeBase_loop1 = MAT ();
    elapsedTime_MAT_loop1 = (end_timeBase_loop1 -
   start_timeBase_loop1)/ 1000000.0;
    printf("%f ms  \n",elapsedTime_MAT_loop1);

    /**
     Shuflle permutation numbers using Fisher Yates Algorithm
     */
    printf("\n\n Shuffle \n");
    printf("Elapsed (MAT) \n");
    printf("--------------\n");
    double  elapsedTime_MAT_shuffle1 ;
    double start_timeBase_shuffle1 , end_timeBase_shuffle1;
    start_timeBase_shuffle1 = MAT() ;
    shuffle (array_of_Rand_Num , Num_of_Nodes) ;
    end_timeBase_shuffle1 = MAT ();
    elapsedTime_MAT_shuffle1 = (end_timeBase_shuffle1 -
```

```
start_timeBase_shuffle1)/ 1000000.0;
 printf("%f ms \n", elapsedTime_MAT_shuffle1);

 /**
  Insert
  */
 Node *root = NULL ;
 printf("\n\nInsertion \n");
 printf("Elapsed (MAT) \n");
 printf("--------------\n");
 double elapsedTime_MAT_insert1;
 double start_timeBase_insert1 , end_timeBase_insert1;
 int j =0;
 start_timeBase_insert1 = MAT() ;
 while (j< Num_of_Nodes)
  {
     root = insert (root,array_of_Rand_Num[j]);
     j++;
  }
  end_timeBase_insert1 = MAT ();
  elapsedTime_MAT_insert1 = (end_timeBase_insert1 -
start_timeBase_insert1)/ 1000000.0;
  printf("%f ms \n", elapsedTime_MAT_insert1);

  /**
  Search
  */
```

```
printf("\n\n Binary Search Tree \n");
printf("Elapsed (MAT) \n");
printf("--------------\n");
double elapsedTime_MAT_binary1;
double start_timeBase_binary1 , end_timeBase_binary1;
int  k=0;
int c;
start_timeBase_binary1 = MAT() ;
int c_found = 0;
int c_not_found = 0 ;
srandom(time(NULL));
long r;
while (k< Num_of_Nodes)
{
r=(random() % Num_of_Nodes);
    replace(r, root);
c = search (r, root );
if ( c == 1 ) {
c_found++ ;
}
else {
c_not_found++ ;
}
k++;
}
end_timeBase_binary1 = MAT ();
elapsedTime_MAT_binary1 = (end_timeBase_binary1 -
```

```
start_timeBase_binary1)/ 1000000.0;
  printf("%f ms \n",elapsedTime_MAT_binary1);
  printf("Found is %i \n",c_found);
  printf("Not Found is %i \n",c_not_found);

  /**
  Replace
  */
  printf("\n\n Binary Tree Replacement \n");
  printf("Elapsed (MAT) \n");
  printf("--------------\n");
  double elapsedTime_MAT_replace1;
  double start_timeBase_replace1 , end_timeBase_replace1;
  int  z=0;
  start_timeBase_replace1 = MAT() ;
  srandom(time(NULL));
  long r2;
  while (z< Num_of_Nodes)
  {
  r2=(random() % Num_of_Nodes);
  replace(r2, root);
  z++;
  }
  end_timeBase_replace1 = MAT ();
  elapsedTime_MAT_replace1 = (end_timeBase_replace1 -
start_timeBase_replace1)/ 1000000.0;
  printf("%f ms \n",elapsedTime_MAT_replace1);
```

```
    free(array_of_Rand_Num);
    return 0;
}
```

- C++

```
/**
 COPYRIGHT (c) 2018 Elaf Alhazmi . All Rights Reserved.
 SUPERVISED BY Dr.Emil Sekerinski and Dr.Frantisek Franek
 C++ Code - Binary Tree
 */

#include <iostream>
#include <time.h>
#include <stdlib.h>
#include <sys/time.h>
#include <mach/mach_time.h>

using namespace std;

struct node
{
    long key;
    struct node *left;
    struct node *right ;
}*root ;
```

```
//Global Variables
long Nodes_created = 0;
long Nodes_not_created = 0;

class BT
{
    public :

    void insert_node (node*, node*);
    void search_node (long, node* );
    void replace(long, node* );
    void swap_node (node* ,node* ,node* );
    void inorder (node*);
    BT ()
    {
        root = NULL ;
    }
};

/**
 Function insert_node() inserts new node in tree
 Param:  Node *root, Node *newnode
 */
void BT::insert_node(node *tree, node *newnode)
{
    node *parent, *current ;
    if (root == NULL)
```

```
{
    root = new node ;
    root->key = newnode->key ;
    root->left = NULL ;
    root->right =NULL ;
    Nodes_created++;
}
else
{
    current = tree ;
    while( tree != NULL)
    {
        parent = current;
        if (newnode->key == current->key)
        {
            Nodes_not_created= Nodes_not_created + 1;
            break;
        }
        if (newnode->key < current->key)
        {
            current = current->left;
            if(current == NULL)
            {
                parent->left= newnode;
                newnode->left= NULL ;
                newnode->right = NULL;
                ++Nodes_created;
```

```
                    break;
                }
            }
            else
            {
                current=current->right;
                if(current == NULL)
                {
                    parent->right = newnode ;
                    newnode->left= NULL ;
                    newnode->right = NULL;
                    ++Nodes_created;
                    break;
                }
            }
        }
    }
}

/**
 Function search_node checking if item is exist in Tree
 Param :  long item, Node *par,
 */
void BT::search_node(long item, node *par)
{
    if (root == NULL)
    {
```

```
            cout<<"Root is NULL "<< endl;
            return;
        }
        else
        {
            if ( par != NULL)
            {
                if (item == par->key)
                {
                    return;
                }
                if (item < par->key)
                {
                    search_node(item, par->left);
                }
                else
                {
                    search_node(item, par->right);
                }
            }
            else
            {
                return;
            }
        }
    }
```

```
/**
 Function replace
 Param :  long item, Node *par,
 */
void BT::replace(long item, node *par)
{
    node *parent, *current, *temp;
    current = root ;
    parent = root ;
    temp = new node ;
    temp->key = item;
    temp->left = NULL ;
    temp->right =NULL ;

    while (current != NULL)
    {
        parent = current;
        if (item < current->key) {
            current = current->left;
            if(item == current->key) {
                swap_node(parent, current, temp);
                break;
            }
        }
        else if (item > current->key) {
            current = current->right;
```

```
                if (item == current->key) {
                    swap_node(parent, current, temp);
                    break;
                }
            }
            else {
                break;
            }
        }
    }

    void BT::swap_node (node *parent ,node *current,node *temp)
    {
        if (current == root) {
            return;
        }
        if (parent->key > current->key) {
            parent->left = temp;
            temp->left = current->left;
            temp->right = current->right;
            delete current;
        }
        else if (parent->key < current->key)
        {
            parent->right = temp;
            temp->right = current->right;
            temp->left = current->left;
```

```
            delete current;
        }
    }

    void BT::inorder(node *temp)
    {
        if ( temp != NULL  )
        {
            inorder(temp->left);
            cout<<" value" << temp->key<<"\n";
            inorder(temp->right);
        }
    }

    void Shuffle(long *a, long count_num )
    {
        int r;
        for (int i = count_num -1; i > 0; --i)
        {
            srandom(time(NULL));
            r = random() % (i+1) ;
            swap (a[i],a[r]);
        }
    }

    /**
     on OSX, mach_absolute_time() is used for measuring elpased
```

```
  time
 it is monotonic
 */
double MAT ()
{
    mach_timebase_info_data_t start_timeBase_loop1;
    mach_timebase_info(&start_timeBase_loop1);
    return (((mach_absolute_time() *
   (double)start_timeBase_loop1.numer )/ (double)
   start_timeBase_loop1.denom ));
}

/**
 Main Function performs five main tasks
 1.Generating Permutation numbers from [0-Num_of_Nodes)
 2.Using Fisher Yates algorithm to shuffle permutation
   numbers order
 3.Insert - Num_of_Nodes - in Binary Tree
 4.Search - Num_of_Nodes - in Binary Tree
 5.Replace - Num_of_Nodes - in Binary Tree
 */

int main()
{
    long Num_of_Nodes = 10000000;
    long *array_of_Rand_Num ;
    array_of_Rand_Num = new  long [Num_of_Nodes];
```

```
 /**
  Generating permutation numbers
  */
 cout<<"\nLoop : Generating Permutation Numbers from 0 -
"<<Num_of_Nodes<<" \n";
 cout<<"Elapsed (MAT) \n";
 cout<<"--------------\n";
 double elapsedTime_MAT_loop1 ;
 double start_timeBase_loop1 , end_timeBase_loop1;
 start_timeBase_loop1 = MAT() ;
 for (int i=0 ; i< Num_of_Nodes ; i++)
 {
     array_of_Rand_Num [i] = i;
 }
 end_timeBase_loop1 = MAT ();
 elapsedTime_MAT_loop1 = (end_timeBase_loop1 -
start_timeBase_loop1)/ 1000000.0;
 cout<<elapsedTime_MAT_loop1<<" ms"<<endl ;

 /**
  Shuflle permutation numbers Fisher Yates Algorithm
  */
 cout <<"\n\nShuffle : Shuffle Permutation number \n";
 cout<<"Elapsed (MAT) \n";
 cout<<"-----------------\n";
 double elapsedTime_MAT_shuffle1 ;
```

```
 double start_timeBase_shuffle1 , end_timeBase_shuffle1;
 start_timeBase_shuffle1 = MAT() ;
 Shuffle(array_of_Rand_Num, Num_of_Nodes);
 end_timeBase_shuffle1 = MAT ();
 elapsedTime_MAT_shuffle1 = (end_timeBase_shuffle1 -
start_timeBase_shuffle1)/ 1000000.0;
 cout<< elapsedTime_MAT_shuffle1<<" ms"<<endl ;

 /**
  Insert
  */
 BT bst;
 node *temp;
 cout<<"\n\n Insert \n";
 cout<<"Elapsed (MAT) \n";
 cout<<"--------------\n";
 double elapsedTime_MAT_insert1 ;
 double start_timeBase_insert1 , end_timeBase_insert1;
 start_timeBase_insert1 = MAT() ;
 for (long i=0; i<Num_of_Nodes ; i++)
  {
     temp = new node;
     temp->key= array_of_Rand_Num[i];
     bst.insert_node(root,temp);
  }
  end_timeBase_insert1 = MAT ();
  elapsedTime_MAT_insert1 = (end_timeBase_insert1 -
```

```
start_timeBase_insert1)/ 1000000.0;
  cout<<elapsedTime_MAT_insert1<<" ms"<<endl ;

 /**
   Search
  */
  cout<< "\n\nBinary Search Tree \n";
  cout<<"Elapsed (MAT) \n";
  cout<<"--------------\n";
  double elapsedTime_MAT_binary1 ;
  double start_timeBase_binary1 , end_timeBase_binary1;
  start_timeBase_binary1 = MAT() ;
  srandom(time(NULL));
  int r;
  for (long j=0 ; j<Num_of_Nodes; j++)
  {
  r = random() % Num_of_Nodes ;
  bst.search_node( r,root);
  }
  end_timeBase_binary1 = MAT ();
  elapsedTime_MAT_binary1 = (end_timeBase_binary1 -
start_timeBase_binary1)/ 1000000.0;
  cout<<elapsedTime_MAT_binary1<<" ms"<<endl ;

 /**
  Replace
  */
```

```
    cout<< "\n\nBinary Tree - replacement \n";
    cout<<"Elapsed (MAT) \n";
    cout<<"--------------------\n";
    double elapsedTime_MAT_replace1 ;
    double start_timeBase_replace1 , end_timeBase_replace1;
    start_timeBase_replace1 = MAT() ;
    srandom(time(NULL));
    int r2;
    for (long j=0 ; j<Num_of_Nodes; j++)
    {
    r2 = random() % Num_of_Nodes ;
    bst.replace( r2,root);
    }
    end_timeBase_replace1 = MAT ();
    elapsedTime_MAT_replace1 = (end_timeBase_replace1 -
  start_timeBase_replace1)/ 1000000.0;
    cout<<elapsedTime_MAT_replace1<<" ms"<<endl ;
    delete [] array_of_Rand_Num ;
    return 0;
}
```

- Java
  - BT.java

```
/**
 COPYRIGHT (c) 2018 Elaf Alhazmi . All Rights Reserved.
```

```
 SUPERVISED BY Dr.Emil Sekerinski and Dr.Frantisek Franek
 Class to implement Binary Tree Abstract Data Type
 */
public class BT {
    public BTNode root ;
    public long Nodes_created = 0;
    public long Nodes_not_created = 0;
    public long s = 0;

    /**
     * Default constructor for Binary Tree (BT)
     * Note: creating empty BT with null pointer
     */
    public BT ()
    {
        root = null ;
    }

    /**
     * Check if BT is empty
     * @return root with null pointer
     */
    public boolean isEmpty ()
    {
        return root == null ;
    }
```

```
/**
 * Insert New BTNode to the current BT
 * @param n is integer value, key value of the new
Node
 */
public void insert (int n)
{
    BTNode parent, current , temp;
    if ( root == null)
    {
        root = new BTNode (n);
        ++Nodes_created;
    }
    else
    {
        temp = new BTNode ( n);
        current = root;
        while (root != null)
        {
            parent = current ;
            if (n == current.getKey())
            {
                ++Nodes_not_created;
                break ;
            }
            else
            {
```

```
                    if (n < current.getKey())
                    {
                        current = current.getLeft();
                        if(current == null)
                        {
                            parent.setLeft(temp);
                            ++Nodes_created;
                            break;
                        }
                    }
                    else
                    {
                        current = current.getRight();
                        if (current == null)
                        {
                            parent.setRight(temp);
                            Nodes_created++;
                            break;
                        }
                    }
                }
            }
        }
    }

    /**
     * Binary search Tree
```

```
    * @param item is  integer value,used for searching
    * @param par is  BTNode
    * @return true if it is found, otherwise return
false
    */
  public boolean search (int item , BTNode par)
  {
      boolean flag = false;
      if (root == null )
      {
         flag = false;
      }
      else
      {
          if(par != null)
          {
              if(par.getKey() == item)
              {
                  flag =true;

                  return flag;
              }
              if(item < par.getKey() )
              {
                  this.search(item, par.left);
              }
              else
```

```
                {
                    this.search(item, par.right);
                }
            }
            else
            {
                return flag;
            }
        }
        return flag;
    }

    /**
     * Replace
     * @param item is  integer value,used for searching
     * @param par is  BTNode
     */
     public void replase (int item , BTNode par)
     {
        BTNode current , parent , temp;
        current = root;
        parent = root;
        temp = new BTNode(item);

        while(current != null)
        {
            parent = current;
```

```
                if(item < current.key) {
                    current = current.left;
                    if (item == current.key)
                {
                    this.swap_node(parent, current, temp);
                    break;
                 }
                }
                else if (item > current.key)
                 {
                        current = current.right;
                        if (item == current.key)
                {
                    this.swap_node(parent, current, temp);
                    break;
                 }
                  }
                else
                {
                    break;
                }
            }// end while
          }

       public void swap_node (BTNode parent , BTNode
    current , BTNode temp)
       {
```

```
        if (current == root){
            return;
        }
        else {
         if (parent.key > current.key)
          {
              parent.left = temp;
              temp.left = current.left;
              temp.right = current.right;
              current = null;
          }
         else if (parent.key < current.key)
          {
            parent.right = temp;
            temp.right = current.right;
            temp.left = current.left;
            current = null;
          }
     }
  }

/**
 * Printing BT Nodes by using in-order traverser
 * @param r is BTNode
 */
public void inorder(BTNode r)
{
```

```
            if ( r != null )
            {
                inorder (r.getLeft());
                System.out.println(r.getKey());
                s += 1;
                inorder (r.getRight());
            }
        }
}
```

– BTNode.java

```
/**
 COPYRIGHT (c) 2018 Elaf Alhazmi . All Rights Reserved.
 SUPERVISED BY Dr.Emil Sekerinski and Dr.Frantisek Franek
 Class to declare Binary Tree Abstract Data Type - Node
 @author Elaf Alhazmi
 */

public class BTNode {
    BTNode left, right ;
    int key ;

    /**
     * Default constructor for BTNode
     */
```

```
public BTNode()
{
    left = null ;
    right = null;
    key = 0 ;
}

/**
 * Construct a Node with a value
 * @param n is integer value that is associated with
a Node
 */
public BTNode (int n)
{
    left = null ;
    right = null;
    key = n ;
}

/**
 * Set left Node reference of the current Node
 * @param l is BTNode, linked to the left of current
BTNode
 */
 public void setLeft (BTNode l)
 {
  left = l;
```

```
    }

    /**
     * Set right Node reference of the current Node
     * @param r is BTNode, linked to the right of
current BTNode
     */
    public void setRight (BTNode r)
    {
        right = r;
    }

    /**
     * Get left Node reference of the current Node
     * @return the left BTNode of the current BTNode
     */
    public BTNode getLeft ()
    {
        return left ;
    }

    /**
     * Get right Node reference
     * @return the right BTNode of the current BTNode
     */
    public BTNode getRight ()
    {
```

```
        return right ;
    }

    /**
     * Edit the integer value of a Node
     * @param n is integer value, editing the key value
  of BTNode
     */
    public void setKey (int n)
    {
        key = n;
    }

    /**
     * Return the integer value of a Node
     * @return key value of current BTNode
     */
    public int getKey ()
    {
        return key ;
    }
}
```

– BTree.java

```
import java.util.Random;
```

```
/**
 * COPYRIGHT (c) 2018 Elaf Alhazmi . All Rights Reserved.
 * SUPERVISED BY BY Dr.Emil Sekerinski and Dr.Frantisek
   Franek
 * Java BTree class
 */
class BTree {

    public static void main(String[] args) {

        BT bt = new BT();
        int Num_of_Nodes = 50000000;
        int[] array_of_Rand_Num = new int[Num_of_Nodes];
        System.out.println("Loop : Generating
   Permutation Numbers from 0 - " + Num_of_Nodes);
        System.out.println("Elapsed Time (NT)  ");
        System.out.println ("-----------------");
        double start_NT_loop1 = System.nanoTime( );

        for (int i = 0; i < Num_of_Nodes; i++) {
            array_of_Rand_Num[i] = i;
        }
        double end_NT_loop1 = System.nanoTime( );
        double diff_NT_loop1 = end_NT_loop1 -
   start_NT_loop1;
        System.out.println ((diff_NT_loop1)/1000000.0D );
        System.out.println("\n\nShuffle : Shuffle
```

```
Permutation number");
          System.out.println("Elapsed Time (NT)  ");
          System.out.println ("-----------------");
          double start_NT_shuffle1 = System.nanoTime( );
          shuffle(array_of_Rand_Num, Num_of_Nodes);
          double end_NT_shuffle1 = System.nanoTime( );
          double diff_NT_shuffle1 = end_NT_shuffle1 -
start_NT_shuffle1;
          System.out.println
((diff_NT_shuffle1)/1000000.0D );
          System.out.println("\n\nInsert");
          System.out.println("Elapsed Time (NT)");
          System.out.println ("----------------");
          double start_NT_insert1 = System.nanoTime( );

          for (int i = 0; i < Num_of_Nodes; i++) {
              bt.insert(array_of_Rand_Num[i]);
          }
          double end_NT_insert1 = System.nanoTime( );
          double diff_NT_insert1 = end_NT_insert1 -
start_NT_insert1;
          System.out.println("\n\nBinary Search Tree");
          System.out.println("Elapsed Time (NT)  ");
          System.out.println ("-----------------");
          double start_NT_binary1 = System.nanoTime( );
          Random r2 = new Random();
          boolean c;
```

```
        int c_found =0 ;
        int c_not_found = 0;
        for (int j = 0; j < Num_of_Nodes; j++) {
            c = bt.search(r2.nextInt(Num_of_Nodes),
bt.root);
            if (c == true)
            { c_found++;}
            else
            { c_not_found++ ;}
        }
        double end_NT_binary1 = System.nanoTime( );
        double diff_NT_binary1 = end_NT_binary1 -
start_NT_binary1;
        System.out.println
((diff_NT_binary1)/1000000.0D);
        System.out.println("\n\nBinary Tree -
Replacement");
        System.out.println("Elapsed Time (NT) ");
        System.out.println ("-----------------");
        double start_NT_replace1 = System.nanoTime( );
        Random r3 = new Random();

        for (int j = 0; j < Num_of_Nodes; j++) {
            bt.replase(r3.nextInt(Num_of_Nodes),
bt.root);
        }
        double end_NT_replace1 = System.nanoTime( );
```

```
        double diff_NT_replace1 = end_NT_replace1 -
    start_NT_replace1;
        System.out.println
    ((diff_NT_replace1)/1000000.0D);
        System.out.println ("Nodes created = "+
    bt.Nodes_created);
    }

    public static void shuffle(int[] a, int count_num) {
        int r;
        int temp;
        for (int i = count_num - 1; i > 0; --i) {
            Random r1 = new Random();
            r = r1.nextInt(count_num);
            temp = a[i];
            a[i] = a[r];
            a[r] = temp;
        }
    }
}
```

- Rust

```
/**
 COPYRIGHT (c) 2018 Elaf Alhazmi . All Rights Reserved.
 SUPERVISED BY Dr.Emil Sekerinski and Dr.Frantisek Franek
 Rust Code - Binary Tree
```

```
*/
extern crate rand;
extern crate time;
use rand::{thread_rng, Rng};
use std::fmt;
use time::PreciseTime;
use std::time::Instant;

fn shuffle_x(a: &mut[i32], count_num: usize)
{
   let mut r :usize;
   let mut temp;
   let mut rng = thread_rng();
   for i in 1..count_num
   {
      r = rng.gen_range(0, i);
       temp = a[i];
         a[i] = a[r];
         a[r] = temp;
   }
}

/**
Node Structure
*/
#[derive(Debug)]
pub struct Node<T>{
```

```
    key: T,
    left : Option<Box<Node<T>>>,
    right: Option<Box<Node<T>>>,
}

/**
Tree Structure
*/
#[derive(Debug)]
pub struct Tree<T> {
    root : Option<Box<Node<T>>>,
    size : usize,
}

/**
Node Implementation
*/
impl<T> Node<T> {
    fn new( key: T) -> Self {
    Node { key : key , left : None, right : None}
    }
}

macro_rules! fmt_node {
     ($fmt:expr, $node:expr) => {
        if let Some(ref node) = $node.as_ref() {
             try!(node.fmt($fmt));
```

```
            }
        };
    }

    impl<T: std::fmt::Display> fmt::Display for Tree<T>  {
        fn fmt(&self, fmt: &mut fmt::Formatter) -> Result<(),
      fmt::Error> {
            try!(write!(fmt, "["));
            fmt_node!(fmt, self.root);
            try!(write!(fmt, "]"));
             Ok(())
      }
     }

    impl<T: std::fmt::Display> fmt::Display for Node<T> {
         fn fmt(&self, fmt: &mut fmt::Formatter) -> Result<(),
      fmt::Error> {
            fmt_node!(fmt, self.left);
            try!(write!(fmt, "{}, ", self.key));
            fmt_node!(fmt, self.right);
            Ok(())
         }
    }

    /**
    Tree Implementation
    */
```

```
impl<T:Ord> Tree<T> {
   pub fn new () -> Self {
   Tree { root : None, size: 0}
   }
   /**
   Function insert() adds new node in tree
   */
   pub fn insert ( &mut self , key: T) {
      match self.root.as_mut() {
      None => {
         self.root = Some (Box::new (Node :: new(key)));
         self.size +=1;
         }, //end None
      Some(_) => {
         let mut next = self.root.as_mut().unwrap();
         loop {
            let cur = next;
            let target = {
               let elem_ref = &key;
               if cur.key == *elem_ref {
                  return;
               } // end if
               if cur.key < *elem_ref
                 {&mut cur.right }
               else
                 {&mut cur.left}
            }; // end target
```

```
            if target.is_some() {
               next = target.as_mut().unwrap();
               continue;
               } // end if
            *target= Some(Box::new(Node::new(key)));
            self.size +=1;
            return;
               } // end loop
            } // end some
         }// end match
      }// end insert

      /**
      Function search checking if value is exist in Tree
      */
      pub fn search( &mut self , key: T)-> bool {
         let mut flag = false;
         match self.root.as_mut() {
            None => {
               println! ("No Elements in Tree");
            }//end None
            Some(_) => {
                 let mut next = self.root.as_mut().unwrap();
               loop {
                  let cur = next;
                   let target = {
                     let elem_ref = &key;
```

```
                    if cur.key == *elem_ref {
                       flag = true;
                       break;
                    } // end if
                    if cur.key < *elem_ref {
                       &mut cur.right
                    } // end if
                    else {
                       &mut cur.left
                    }//end else
                 }; //end target
                 if target.is_some() {
                    next = target.as_mut().unwrap();
                    continue;
                 } // end if
                 break;
              } // end loop
           }//end Some
        }// end mathch
     return flag;
     }// end search function

     /**
     Function replace new node of existing one
     */
     pub fn replace  ( &mut self ,key: T) {
        match self.root.as_mut() {
```

```
None => {
   println! ("No Elements in Tree");
}//end None
Some(_) => {
   {
      let mut parent = self.root.as_ref();
      {
         // Root checking
         if parent.unwrap().key == *(&key) {
         return
         } // end if
      }
   }

   {
      let mut next = self.root.as_mut().unwrap();
      loop {
         let mut cur = next;
         let mut target = {
            if cur.left.is_some() {
               if (cur.left).as_mut().unwrap().key
== *(&key) {
                  let mut new_node =
Some(Box::new(Node::new(key)));
                  {
                     let mutate_new_node =
new_node.as_mut().unwrap();
```

```
                                {
                                    mutate_new_node.left =
(cur.left.as_mut().unwrap()).left.take();
                                }// end mini left moving

                                {
                                        {
                                            mutate_new_node.right
= (cur.left.as_mut().unwrap()).right.take();
                                        }// end mini right moving
                                }
                            }
                            cur.left = new_node;
                            break;
                        } // end left replacement
                    }
                    if cur.right.is_some() {
                        if (cur.right).as_mut().unwrap().key
== *(&key) {
                            let mut new_node =
Some(Box::new(Node::new(key)));
                            {
                                let mutate_new_node =
new_node.as_mut().unwrap();
                                {
                                    mutate_new_node.left =
(cur.right.as_mut().unwrap()).left.take();
```

```
                        }// end mini left moving

                        {
                           {
                              mutate_new_node.right =
(cur.right.as_mut().unwrap()).right.take();
                           }// end mini right moving
                         }
                      }
                      cur.right = new_node;
                     break;
                  } // end if
               }

               if cur.key < *(&key) {
                  &mut cur.right
               }
               else {
               if cur.key > *(&key) {
                  &mut cur.left
               }
               else {
                  break;
               }
               }
            }; // end target
            if target.is_some() {
```

```
                        next = target.as_mut().unwrap();
                        continue;
                        } // end if
                        else {
                           break;
                        }
                        return;
                     } // end loop
                  }
            }//end Some
         }// end match
      }// end search function
   } // end ord

   /**
    Main Function performs five main tasks
    1.Generating Permutation numbers from [0-Num_of_Nodes)
    2.Using Fisher Yates algorithm to shuffle permutation
      numbers order
    3.Insert - Num_of_Nodes - in Binary Tree
    4.Search - Num_of_Nodes - in Binary Tree
    5.Replace - Num_of_Nodes - in Binary Tree
    */
   fn main() {
      const Num_of_Nodes:usize = 10000000;
      let mut array_of_Rand_Num = vec![0; Num_of_Nodes];
```

```
/**
  Generating permutation numbers
  */
println!("Loop : Generating Permutation Numbers from 0 -
{} \n",Num_of_Nodes);
 println!("Elapsed Time (Instant) \t Elapsed Time
(Precise) \n");
 println!("----------------------------------\n");
let start_Precise_loop1 = PreciseTime::now();
let start_instant_loop1 = Instant::now();
for i in 0..Num_of_Nodes {
array_of_Rand_Num[i] = i as i32;
}
let end_instant_loop1 = start_instant_loop1.elapsed();
let end_Precise_loop1 = PreciseTime::now();
let diff_instant_loop1 = (end_instant_loop1.as_secs() as
f64)*1000.0 + (end_instant_loop1.subsec_nanos() as f64 /
1000000.0);
println!(" {} ms \t\t {} s",diff_instant_loop1,
start_Precise_loop1.to(end_Precise_loop1));

/**
  Shuflle permutation numbers using Fisher Yates Algorithm
  */
println!("Shuffle : Shuffle Permutation Numbers");
 println!("Elapsed Time (Instant) \t Elapsed Time
(Precise) \n");
```

```
 println!("-------------------------------------");
let start_Precise_shuffle1 = PreciseTime::now();
let start_instant_shuffle1 = Instant::now();
shuffle_x (&mut array_of_Rand_Num , Num_of_Nodes );
let end_instant_shuffle1 =
start_instant_shuffle1.elapsed();
let end_Precise_shuffle1 = PreciseTime::now();
let diff_instant_shuffle1 =
(end_instant_shuffle1.as_secs() as f64)*1000.0 +
(end_instant_shuffle1.subsec_nanos() as f64 / 1000000.0);
println!(" {} ms \t\t {} s",diff_instant_shuffle1,
start_Precise_shuffle1.to(end_Precise_shuffle1));


/**
  Insert
  */
let mut tree = Tree::new();
println!("Insertion, Binary Tree");
 println!("Elapsed Time (Instant) \t Elapsed Time
(Precise) \n");
 println!("------------------------------------------");
let start_Precise_insert1 = PreciseTime::now();
let start_instant_insert1 = Instant::now();
for i in 0..Num_of_Nodes  {
   tree.insert(array_of_Rand_Num[i]);
}
let end_instant_insert1 = start_instant_insert1.elapsed();
```

```
let end_Precise_insert1 = PreciseTime::now();
let diff_instant_insert1 = (end_instant_insert1.as_secs()
as f64)*1000.0 + (end_instant_insert1.subsec_nanos() as
f64 / 1000000.0);
println!(" {} ms \t\t {} s",diff_instant_insert1,
start_Precise_insert1.to(end_Precise_insert1));

/**
  Binary Search Tree using random numbers from 0-1000000
  */
println!("Binary Search - Tree");
 println!("Elapsed Time (Instant) \t Elapsed Time
(Precise) \n");
 println!("----------------------------------------");
let mut rng = thread_rng();
let mut found = 0 ;
let mut not_found = 0;
let mut t ;
let mut n: i32 = rng.gen_range(0, Num_of_Nodes as i32);
let start_Precise_binary1 = PreciseTime::now();
let start_instant_binary1 = Instant::now();
for _ in 0..Num_of_Nodes  {
   t = tree.search(n);
   if  t == true {
   found = found + 1 ;
   }
   else {
```

```
   not_found= not_found + 1 ;
   }
   n = rng.gen_range(0, Num_of_Nodes as i32) ;
}
let end_instant_binary1 = start_instant_binary1.elapsed();
let end_Precise_binary1 = PreciseTime::now();
let diff_instant_binary1 = (end_instant_binary1.as_secs()
as f64)*1000.0 + (end_instant_binary1.subsec_nanos() as
f64 / 1000000.0);
println!(" {} ms \t\t {} s",diff_instant_binary1,
start_Precise_binary1.to(end_Precise_binary1));
println! ("Found is {} and Not Found is {}", found,
not_found);


/**
  Binary Tree - replacement -
  */
println!("Binary Tree - replace");
 println!("Elapsed Time (Instant) \t Elapsed Time
(Precise) \n");
 println!("-----------------------------------------");
let mut rng_replace = thread_rng();
let mut n: i32 = rng_replace.gen_range(0, Num_of_Nodes as
i32);
let start_Precise_replace1 = PreciseTime::now();
let start_instant_replace1 = Instant::now();
```

```
        for _ in 0..Num_of_Nodes  {
            tree.replace(n);
            n = rng_replace.gen_range(0, Num_of_Nodes as i32) ;
        }
        let end_instant_replace1 =
        start_instant_replace1.elapsed();
        let end_Precise_replace1 = PreciseTime::now();
        let diff_instant_replace1 =
        (end_instant_replace1.as_secs() as f64)*1000.0 +
        (end_instant_replace1.subsec_nanos() as f64 / 1000000.0);
        println!(" {} ms \t\t {} s",diff_instant_replace1,
        start_Precise_replace1.to(end_Precise_replace1));
        println!(" Size {} ", tree.size);
        //println! ("{:} " , tree);
    }
```

A.3 Rust - Rc<T> and RefCell<T> cyclic issue

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};
use std::fmt::Display;

#[derive(Debug)]
struct Node {
    value: i32,
```

```
    parent: RefCell<Option<Rc<Node>>>,
    // Weak reference should be used instead of Rc
    next: RefCell<Option<Rc<Node>>>,
}

 impl Drop for Node {
    fn drop(&mut self) {
        println!("Dropping! Node value {}", self.value);
    }
}

fn main() {
   let L1 = Rc::new(Node {
   value: 1,
   parent: RefCell::new(None),
   next: RefCell::new(None),
   });
   let L2 = Rc::new(Node {
        value: 5,
        parent: RefCell::new(None),
        next: RefCell::new(Some(Rc::clone(&L1))),
    });
   println!("L1 next = {:?}", L1.next);
   println!("L2 next = {:?}\n ", L2.next);
   // Link L2 to L1
   // Hide the following instruction
   // to avoid stack overflow at run-time (cyclic issue)
```

```
    *L1.parent.borrow_mut() = Some(Rc::clone(&L2));
    // Causes Stack overflow
    // Because L2 points to L1 parent
    println!("L1 next = {:?}", L1.next);
    println!("L2 next = {:?}", L2.next);
}
```

A.4 Rust- smart pointers composition

- **Rc <RefCell<vec<T> > >**

```
use std::rc::{Rc, Weak};
use std::cell::{RefCell, RefMut};
use std::ops::DerefMut;
use std::mem::replace;

fn main()
{
    let fpointer = Rc::new(RefCell::new(vec![1,2,3,4]));
    let s1 = fpointer.clone();
    let s2 = fpointer.clone();
    let w1 = Rc::downgrade(&fpointer);
    // Try with and without {} Scope delimitar
    // Without the value is "borrowed"
    /*{let mut bmut = fpointer.borrow_mut();
    bmut.push(55);}*/
    println!("fpointer before adding new number  \n =
```

```
    {:?}\n", fpointer);
    //(*fpointer.borrow_mut()).push(10);
    //(*fpointer.borrow_mut()).push(20);
    {
        let mut bmut = fpointer.borrow_mut();
        //let mut bmut1 = fpointer.borrow_mut();    // error
    (two mutuble borrowed at given time)
        //println!("bmut  = {:?} ", bmut);
        bmut[0] = 1000;
        bmut.push(10);
        bmut.push (20)
    }

    {
        let mut bmut = fpointer.borrow_mut();
        //println!("bmut  = {:?} ", bmut);
        bmut.push(100);
        bmut.push (200)
    }
    println!("fpointer after adding new number \n = {:?}\n",
    fpointer);
    println!("Strong pointers = {}",
    Rc::strong_count(&fpointer));
    println!("Weak pointers   = {}",
    Rc::weak_count(&fpointer));
}
```

– The Output

```
fpointer before adding new number
 = RefCell { value: [1, 2, 3, 4] }

fpointer after adding new number
 = RefCell { value: [1000, 2, 3, 4, 10, 20, 100, 200] }

Strong pointers = 3
Weak pointers   = 1
```

- **Rc<vec<RefCell<T> > >**

```
use std::rc::{Rc, Weak};
use std::cell::{RefCell, RefMut};
use std::ops::DerefMut;
use std::mem::replace;

fn main()
{
let spointer = Rc::new(vec![RefCell::new(1),
   RefCell::new(2), RefCell::new(3)]);
let s1 = spointer.clone();
let s2 = spointer.clone();
let w1 = Rc::downgrade(&spointer);
println!("spointer after adding new number \n = {:?}\n",
   spointer);
```

```
{
   let mut xref = spointer[0].borrow_mut();
   let mut xref1 = spointer[1].borrow_mut();
   *xref = 3;
   *xref1 = 30;
}

{
/* error[E0599]: no method named `borrow_mut` found for type
  `std::rc::Rc<std::vec::Vec<std::cell::RefCell<{integer}>>>`
  in the current scope */
   //let mut xref = spointer.borrow_mut();
   //println!(" xref\n = {:?}\n", spointer);
}

// Error
//error[E0596]: cannot borrow immutable borrowed content as
   mutable
//{*((spointer[2]).borrow_mut()) = 300;}  // error
{*((spointer[2]).borrow_mut() as RefMut<i32>) = 300;} // it
   works
println!("spointer after adding new number \n = {:?}\n",
   spointer);
println!("Strong pointers = {}",
   Rc::strong_count(&spointer));
println!("Weak pointers   = {}", Rc::weak_count(&spointer));
}
```

- The Output

```
spointer after adding new number
 = [RefCell { value: 1 }, RefCell { value: 2 }, RefCell
   { value: 3 }]

spointer after adding new number
 = [RefCell { value: 3 }, RefCell { value: 30 }, RefCell
   { value: 300 }]

Strong pointers = 3
Weak pointers   = 1
```