

CS 1033: Programming Fundamentals

Part B: Beyond Programming Basics

BSc Engineering - Semester 1 (2020 Intake)

August – December 2021

Department of Computer Science and Engineering

Faculty of Engineering

University of Moratuwa

Sri Lanka

Compiled by:

Sanath Jayasena, Shehan Perera, Malaka Walpola, Chandana Gamage, Gayashan Amarasinghe, Kalana Wijegunaratna, Piyumal Demotte, Darshana Priyasad

© Dept of Computer Science & Engineering, University of Moratuwa

Acknowledgements:

All help and support provided to make this Course Note possible are acknowledged.

Sections of this note were based on:

- “*Introduction to C Programming*” Course Notes (2014 version) from the Dept. of Computer Science & Engineering, compiled by Sanath Jayasena, Shehan Perera, Malaka J. Walpola and Adeesha Wijayasiri and others.
- “*No Silver Bullet: Essence and Accidents of Software Engineering*” by Fred Brooks, in IEEE Computer, Vol. 20, No. 4 (April 1987) pp. 10-19.

Note: Feedback is very much appreciated for improvement of this document. Please email your feedback at sanath@cse.mrt.ac.lk.

Date of this Part B version: 31 August 2021

Table of Contents

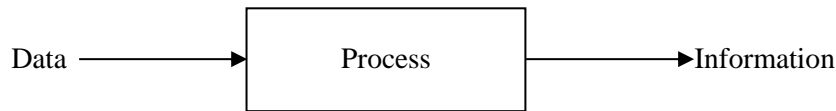
1	Data Representation	5
1.1	Introduction and Terminology	5
1.2	Number Systems	6
1.3	Decimal Number System	6
1.4	Binary Number System	7
1.5	Octal Number System	7
1.6	Hexadecimal Number System	7
1.7	Number Base Conversion.....	7
1.7.1	Decimal to Binary Conversion - Integers.....	7
1.7.2	Decimal to Binary Conversion - Fractions.....	8
1.7.3	Binary to Decimal Conversion - Integers.....	9
1.7.4	Binary to Decimal Conversion – Fractions.....	10
1.7.5	Decimal to Octal Conversion	10
1.7.6	Octal to Decimal Conversion	10
1.7.7	Decimal to Hexadecimal Conversion.....	10
1.7.8	Hexadecimal to Decimal Conversion.....	11
1.7.9	Binary to Octal Conversion.....	11
1.7.10	Octal to Binary Conversion.....	11
1.7.11	Binary to Hexadecimal Conversion	11
1.7.12	Hexadecimal to Binary Conversion	11
1.7.13	Octal \leftrightarrow Hexadecimal Conversion	12
1.8	Arithmetic Operations on Binary Numbers	12
1.8.1	Binary Addition.....	12
1.8.2	Binary Subtraction	12
1.8.3	Binary Multiplication.....	12
1.8.4	Binary Division.....	13
1.9	Representation of Data	13
1.10	Representation of Integers	14
1.10.1	Unsigned Integers	14
1.10.2	Signed Integers and Two's Complement Method	14
1.10.3	How to Obtain Two's Complement Representation.....	16
1.11	Representation of Real Numbers (Floating-point Representation)	16
1.11.1	The Concept of Floating-Point Representation.....	16
1.11.2	The IEEE Standard on Floating-Point Arithmetic	17
1.11.3	Floating-Point Arithmetic in Python.....	18
1.12	Representation of Non-numeric Data	20
1.12.1	ASCII	20

1.12.2	Unicode.....	20
2	Data Structures and Algorithms	23
2.1	Abstract Data Types (ADT)	23
2.2	Introduction to Arrays	23
2.3	Introduction to Stacks.....	24
2.3.1	Using a List as a Stack in Python.....	26
2.3.2	Stack Implementations in C	27
2.3.3	Using a Stack to Evaluate Mathematical Expressions	27
2.3.4	Towers of Hanoi.....	27
2.4	Introduction to Queues	29
2.4.1	Using a List as a Queue in Python	30
2.4.2	Queue using Dynamic Arrays in C	31
2.5	Introduction to Binary Trees	31
2.5.1	Binary Search Trees (BST)	32
2.5.2	Insertion into a Binary Search Tree	32
3	Introduction to Recursion.....	33
3.1	Recursion.....	33
3.2	Tree Traversal with Recursion	34
3.3	More Examples on Recursion	35
3.4	Recursion vs Iteration.....	37
4	Introduction to Algorithms with Sorting.....	38
4.1	Selection Sort	38
4.2	Bubble Sort.....	39
5	Introduction to Software Engineering.....	41
5.1	Introduction	41
5.2	Software	41
5.3	Software Engineer	41
5.4	Software Process	42
5.5	Quality of Software	42
5.6	Challenges in Software Engineering	43
6	Appendix.....	45
6.1	Declaring and Use of Arrays in C – Sample Code.....	45
6.2	Stack as a Fixed Size Array in C – Sample Code	46
6.3	Stack with Pointers and Structs in C – Sample Code.....	47
6.4	Queue with Dynamic Arrays in C – Sample Code.....	48
6.5	Recursive Tree Traversal in Python – Sample Code.....	50

1 Data Representation

1.1 Introduction and Terminology

We use computers to *process* input *data* and produce some useful *information*.



Note that data and information can be relative, i.e., information output from one stage of processing can be fed as input data for another stage of processing. Physical devices used to store and process data in computers are two-state devices, also called *binary* devices. A switch is a two-state device which can either be ON or OFF and its binary states can be represented by the two digits 0 and 1, each of which is known as a *bit* (the abbreviation for *binary digit*). The bit is the smallest unit of data in a computer. Data is stored in *memory*, *disks* and processor *registers* as strings of bits. A string of 8 bits is called a *byte* (also called an *octet*).

A range of units is used to measure amounts of data. Note that, in computing the prefix *kilo* has been used to refer to either $2^{10}=1,024$ or $10^3=1,000$ depending on the context. Similarly, the prefix *mega* has been used to refer to either $2^{20}=1,048,576$ or $10^6=1,000,000$. For example, powers of 1,024 have been used to specify memory capacities and by the operating system to specify file sizes and disk capacities; whereas, powers of 1,000 have been used by disk manufacturers to specify disk capacities and also in data communication/networking contexts. To avoid confusion, international standard bodies ISO and IEC have proposed a set of binary prefixes to unambiguously refer to powers of 1,024 and proposed that SI prefixes are to be used in the decimal sense, as shown in Table 1.1.

Table 1.1: SI and Binary Prefixes

SI prefixes	10^3 bytes = 1 kilo byte = 1 kB	10^6 bytes = 1 mega byte = 1 MB	10^9 bytes = 1 giga byte = 1 GB
Binary prefixes	2^{10} bytes = 1 kibi byte = 1 KiB	2^{20} bytes = 1 mebi byte = 1 MiB	2^{30} bytes = 1 giBi byte = 1 GiB

Data can be broadly classified into two categories *numeric* and *non-numeric* (Figure 1.1). Numeric data represent numbers, i.e., quantifiable and countable things. E.g., the number of students in a class, the percentage marks for CS1033 for a student, GPA of a student, interest rate of a bank. The name of a student, national identity card number, telephone numbers are examples for non-numeric data.

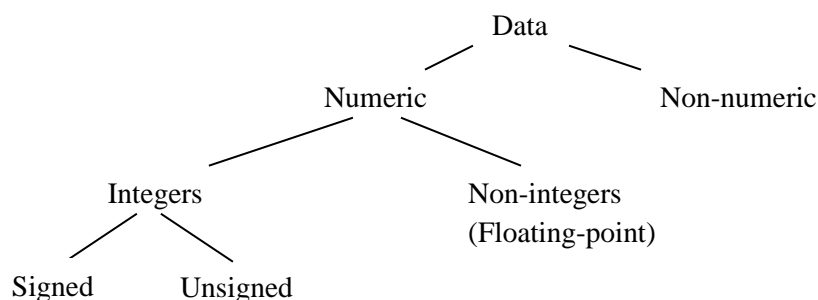


Figure 1.1: Classification of data

We will first discuss the concepts of number systems and representation of numeric data. Next we will discuss representation of non-numeric data.

1.2 Number Systems

Numbers can be represented using different *number systems*. Some of the common number systems are *decimal*, *binary*, *octal* and *hexadecimal*. Decimal is the number system used by humans while binary is the number system used in computers. The octal and hexadecimal systems are closely related to the binary system and therefore important.

The characteristic which distinguishes one number system from another is called the *base* (or *radix*) of a number system. The base is the number of different digits that can occur in each position when representing a number. In simple terms, this is the number of symbols (digits) in the system. It is not surprising that human number system is decimal whose base is ten, when nature provides us with ten fingers. As stated, computers have only two-state (or binary) switches which can be either ON or OFF. As a result computers use the binary number system whose base is two. Table 1.2 summarises different number systems and their symbols.

Table 1.2: Number bases

Number System	Base	Symbols (Digits)
Binary	2	0, 1
Octal	8	0, 1, 2, 3, 4, 5, 6, 7
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

1.3 Decimal Number System

The base of the decimal number system is 10 and the symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. The number 4,567 in base 10 means:

$$\begin{aligned}
 4567_{10} &= \text{four thousand five hundred and sixty seven} \\
 &= 4000 + 500 + 60 + 7 \\
 &= (4 \times 1000) + (5 \times 100) + (6 \times 10) + (7 \times 1) \\
 &= (4 \times 10^3) + (5 \times 10^2) + (6 \times 10^1) + (7 \times 10^0)
 \end{aligned}$$

Numbers are represented using the *positional notation* with which all of us are much familiar. In this notation the numeric value (weight) represented by a digit that appears in a specific position p is expressed by $\text{digit} \times \text{base}^p$. As we move one place to the left, the value of the digit is multiplied by the base (in this case by 10) and when we move to the right, the value is divided by the base (in this case 10). The number 512.49 in base 10 means:

$$\begin{aligned}
 512.49_{10} &= 500 + 10 + 2 + 0.40 + 0.09 \\
 &= (5 \times 100) + (1 \times 10) + (2 \times 1) + (4 \times 0.1) + (9 \times 0.001) \\
 &= (5 \times 10^2) + (1 \times 10^1) + (2 \times 10^0) + (4 \times 10^{-1}) + (9 \times 10^{-2})
 \end{aligned}$$

A number N in base b is written (or expressed) as:

$$N = a_n a_{n-1} a_{n-2} \dots a_1 a_0 . a_{-1} a_{-2} a_{-3} \dots a_{-m} \quad \text{--- (Equation 1.1)}$$

and its numeric value is defined as:

$$N = a_nb^n + a_{n-1}b^{n-1} + a_{n-2}b^{n-2} + \dots + a_1b^1 + a_0b^0 + a_{-1}b^{-1} + a_{-2}b^{-2} + a_{-3}b^{-3} + \dots + a_{-m}b^{-m} \quad \text{--- (Equation 1.2)}$$

The **a** in the above equations is the *digit* and **b** is the *base*. The positional notation employs the *radix point* (.) to separate the *integer* and *fractional* parts of the number. In the decimal system, we call it the *decimal point*.

1.4 Binary Number System

The base of the binary number system is 2 and numbers are represented using symbols 0 and 1. The number 1101.11 in base 2 means:

$$1101.01_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2})$$

Applying equations (1.1) and (1.2) **a** = {0, 1} and **b** = 2.

1.5 Octal Number System

The base of the octal number system is 8 and the symbols are 0, 1, 2, 3, 4, 5, 6 and 7. The number 456.41 in base 8 means:

$$475.01_8 = (4 \times 8^2) + (7 \times 8^1) + (5 \times 8^0) + (0 \times 8^{-1}) + (1 \times 8^{-2})$$

Applying equations (1.1) and (1.2) **a** = {0, 1, 2, 3, 4, 5, 6, 7} and **b** = 8.

1.6 Hexadecimal Number System

The base of the hexadecimal number system is 16 and numbers are represented using symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Since there are no digits beyond digit 9, the first 6 characters of the English alphabet are used to represent the remaining six digits. The number 1FA.4C in base 16 means:

$$\begin{aligned} 1FA.4C_{16} &= (1 \times 16^2) + (F \times 16^1) + (A \times 16^0) + (4 \times 16^{-1}) + (C \times 16^{-2}) \\ &= (1 \times 16^2) + (15 \times 16^1) + (10 \times 16^0) + (4 \times 16^{-1}) + (12 \times 16^{-2}) \end{aligned}$$

In this case **a** = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F} and **b** = 16.

1.7 Number Base Conversion

How do we convert a number from one number system to another? Suppose you want to add two numbers using the *Calculator* program on a computer. The two numbers are input to the calculator in decimal. However the processor of a computer can only understand binary numbers. So the calculator has to convert from decimal to binary before performing the addition. After the addition the final result which is in binary should be converted to decimal for displaying.

1.7.1 Decimal to Binary Conversion - Integers

To convert a decimal number to binary, perform long division by 2 (i.e., divide the number first and then repeatedly the quotient by 2, as long as possible) and record the remainder in the reverse order.

Example 1.1: Convert 8_{10} to binary.

$8/2$	=	4	$r = 0$	
$4/2$	=	2	$r = 0$	
$2/2$	=	1	$r = 0$	
$1/2$	=	0	$r = 1$	

When the remainder is read from bottom to up it becomes 1000 in binary. So $8_{10} = 1000_2$

Example 1.2: Represent 123_{10} in binary.

$$123/2 = 61 \quad r = 1$$

$$61/2 = 30 \quad r = 1$$

$$30/2 = 15 \quad r = 0$$

$$15/2 = 7 \quad r = 1$$

$$7/2 = 3 \quad r = 1$$

$$3/2 = 1 \quad r = 1$$

$$1/2 = 0 \quad r = 1$$

Then $123_{10} = 1111011_2$

Definition: Least Significant Bit (LSB) and Most Significant Bit (MSB)

Given a string of bits, the rightmost bit is called the *least significant bit* (LSB) because, according to the positional notation, it has the least value (weight, or numerical significance) among the bits in the bit string. The leftmost bit is called the *most significant bit* (MSB) because it has the most value.

Example: In the bit string 1001011_2 (75_{10}), the rightmost bit (LSB) contributes 1 out of 75_{10} while the leftmost bit (MSB) contributes 64_{10} out of 75_{10} .

1.7.2 Decimal to Binary Conversion - Fractions

When numbers include fractions the integer and fractional portions can be converted separately. The integer portion is converted in the manner described above. Then the fraction is multiplied by 2 and the integer part of the result is noted. The integer (which will be either 1 or 0) is then stripped out from the result and the fraction of that result is multiplied again by 2. This process continues until the process ends or sufficient degree of precision has been reached. Consider the following example.

Example 1.3: Represent 0.5_{10} in binary.

$$0.5 \times 2 = 1.0$$

0.0×2 (this ends the process)

Therefore $0.5_{10} = 0.1_2$

Example 1.4: Represent 0.25_{10} in binary.

$$0.25 \times 2 = 0.50$$

$$0.50 \times 2 = 1.00$$

0.00×2 (this ends the process)

So $0.25_{10} = 0.01_2$ (read the integer portion of the result of each step, from top to bottom)

Example 1.5: Represent 0.1_{10} in binary.

0.1×2	=	0.2
0.2×2	=	0.4
0.4×2	=	0.8
0.8×2	=	1.6
0.6×2	=	1.2
0.2×2	=	0.4
0.4×2	=	0.8
0.8×2	=	1.6
0.6×2	=	1.2
0.2×2	=	0.4
0.4×2	=	0.8
0.8×2	=	1.6

this process never ends therefore approximately $0.1_{10} \cong 0.000110011001_2$.

The number 0.1_{10} cannot be accurately represented in binary with a finite number of bits. This can happen with some numbers. Also note that bit sequence “1100” is getting repeated; in such cases, we can represent the number as $0.0001\overline{1100}$, where the repeating pattern is marked with a line above it.

1.7.3 Binary to Decimal Conversion - Integers

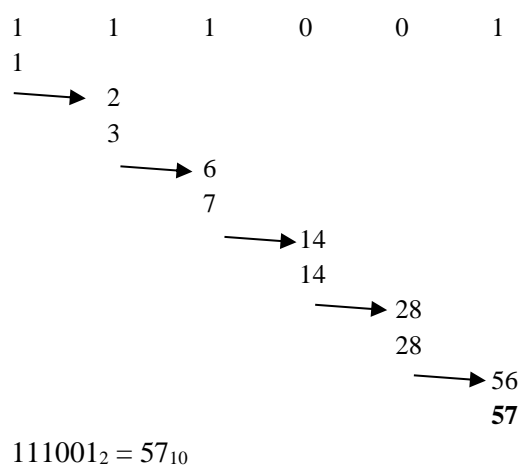
Binary to decimal conversion can be performed by adding together the necessary powers of two.

Example 1.6: Represent 101001_2 in decimal.

$$\begin{aligned}
 101001_2 &= (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\
 &= (1 \times 32) + (0 \times 16) + (1 \times 8) + (0 \times 4) + (0 \times 2) + (1 \times 1) \\
 &= 32 + 8 + 1 \\
 &= 41 \rightarrow 101001_2 = 41_{10}
 \end{aligned}$$

To perform the same conversion another *algorithm* can be used. First take the leftmost non-zero bit, double it and add the result to the bit on its right. Now take the result, double it and add the result to the next bit on the right. Continue in this way until the LSB has been added in.

Example 1.7: Represent 111001_2 in decimal.

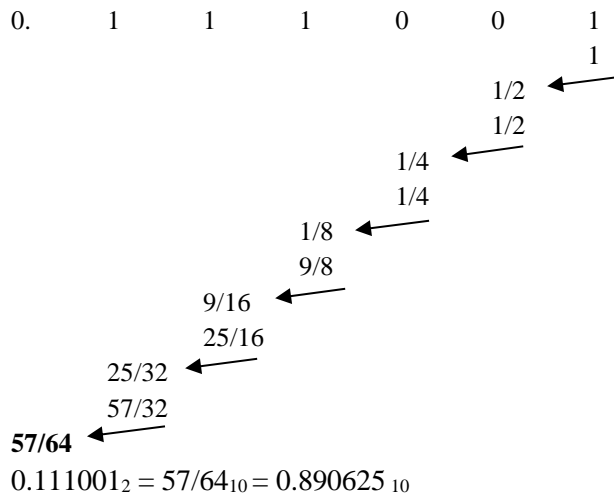


1.7.4 Binary to Decimal Conversion – Fractions

This conversion also can be performed by adding together the necessary powers of two (in this case, these are negative powers) as described in previous examples.

Another algorithm for converting binary fractions to their decimal equivalent is based on the fact that a bit in one position is worth half the value of the bit in the position left to it. Starting at the right most non-zero bit, take that bit and halve it. Now add the result to the next bit on its left. Halve this result and add it to the next bit on the left. Continue this process until the binary (radix) point is reached.

Example 1.8: Represent 0.111001_2 in decimal.



1.7.5 Decimal to Octal Conversion

The process of converting decimal numbers to octal is similar to decimal to binary conversion; instead of dividing by 2, we divide the decimal number by 8.

Example 1.9: Represent 123_{10} in octal.

$$\begin{array}{rclcl} 123/8 & = & 15 & r = \mathbf{3} \\ 15/8 & = & 1 & r = \mathbf{7} \\ 1/8 & = & 0 & r = \mathbf{1} \\ 123_{10} & = & 173_8 \end{array}$$

1.7.6 Octal to Decimal Conversion

This is similar to binary to decimal conversion, except that the powers of 8 are considered.

Example 1.10: Represent 432_8 in decimal.

$$\begin{aligned} 432_8 &= (4 \times 8^2) + (3 \times 8^1) + (2 \times 8^0) \\ &= (4 \times 64) + (3 \times 8) + (2 \times 1) \\ &= 256 + 24 + 2 \\ &= 282 \rightarrow 432_8 = 282_{10} \end{aligned}$$

The other method described for binary to decimal conversion can also be applied here.

1.7.7 Decimal to Hexadecimal Conversion

The conversion is similar to decimal to binary conversion; instead of dividing by 2 we divide the decimal number by 16.

Example 1.11: Represent 1234_{10} in hexadecimal.

$$1234/16 = 77 \quad r = \mathbf{2}$$

$$\begin{array}{rcl}
77/16 & = & 4 \quad r = \mathbf{13} = \mathbf{D} \\
4/16 & = & 0 \quad r = \mathbf{4} \\
1234_{10} & = & 4\mathbf{D}2_{16}
\end{array}$$

1.7.8 Hexadecimal to Decimal Conversion

Example 1.12: Represent $6A8_{16}$ in decimal.

$$\begin{aligned}
6A8_{16} &= (6 \times 16^2) + (A \times 16^1) + (8 \times 16^0) \\
&= (6 \times 256) + (A \times 16) + (8 \times 1) \\
&= 1536 + (10 \times 16) + 8 \\
&= 1536 + 160 + 8 \\
&= 1704 \rightarrow 6A8_{16} = 1704_{10}
\end{aligned}$$

1.7.9 Binary to Octal Conversion

Binary to octal conversion can be performed in two steps: first converting from binary to decimal and then converting the decimal value to octal. However, there is a much easier method that makes use of the fact that $8 = 2^3$. This means a number represented with 3 bits can be represented by a single octal digit.

To convert a given binary number to octal, group the bits into 3-bit blocks starting from the LSB and move towards the left. Replace each block of 3 bits with the corresponding octal digit.

Example 1.13: Convert 11010001_2 into octal

$$\begin{array}{rcl}
11 & | & 010 & | & 001 \\
011 & | & 010 & | & 001 \quad - \text{ may have to add extra bit at left as each block should have 3 bits} \\
\mathbf{3} & \mathbf{2} & \mathbf{1} \\
11010011_2 & = & 321_8
\end{array}$$

1.7.10 Octal to Binary Conversion

It is possible to convert an octal number into binary making use of the fact that $8 = 2^3$. This means that a number represented with an octal digit can be represented in binary by 3 bits. Therefore, to convert an octal number to binary, replace each octal digit with the corresponding 3 bits.

Example 1.14: Convert 276_8 into binary

$$\begin{array}{rcl}
2 & | & 7 & | & 6 \\
010 & | & 111 & | & 110 \quad - \text{ may have to add extra bit at left as each block should have 3 bits} \\
276_8 & = & 010111110_2
\end{array}$$

1.7.11 Binary to Hexadecimal Conversion

The method used in binary to octal conversion can also be applied here. Instead of forming groups of 3 bits starting from the LSB, we form 4-bit groups. This is because $16 = 2^4$.

Example 1.15: Convert 100111010011_2 into hexadecimal

$$\begin{array}{rcl}
1001 & | & 1101 & | & 0011 \\
\mathbf{9} & \mathbf{13} & \mathbf{3} \\
\mathbf{9} & \mathbf{D} & \mathbf{3} \\
100111010011_2 & = & 9D3_{16}
\end{array}$$

1.7.12 Hexadecimal to Binary Conversion

We can convert a hexadecimal number into a binary number by noting that $16 = 2^4$. We simply replace each hexadecimal digit with the corresponding 4-bit binary number.

Example 1.16: Convert $2AF_{16}$ into binary

2 | A | F

0010 | 1010 | 1111 - add extra 0 bits to the left of the leftmost block since each block should be 4 bits

$$2AF_{16} = 001010101111_2$$

1.7.13 Octal \leftrightarrow Hexadecimal Conversion

Octal to hexadecimal conversion (or vice-versa) can be performed in 2 steps: first convert from octal (or hexadecimal) into binary and then convert from binary to hexadecimal (or octal).

Example 1.17: Convert 1234_8 into hexadecimal

1 2 3 4
001 010 011 100 □ 001010011100

0010 | 1001 | 1100

2 | 9 | C

$$1234_8 = 29C_{16}$$

1.8 Arithmetic Operations on Binary Numbers

Arithmetic operations on binary numbers are performed in the same way as on decimal numbers. If all operands are not in the same number base, they should be converted into a common number base before performing any operation. We will discuss operations only with the binary number system.

1.8.1 Binary Addition

Adding two binary numbers is simple, instead of digits from 0 to 9 we have only 0 and 1 to deal with.

Example 1.18: Add 01010_2 and 10001_2

$$\begin{array}{r} 01010 \\ 10001 + \\ \hline 11011 \end{array}$$

1.8.2 Binary Subtraction

Binary subtraction is similar to decimal subtraction.

Example 1.19: Subtract 11110_2 from 11000111_2

$$\begin{array}{r} 11000111 \\ \underline{11110 -} \\ 10101001 \end{array}$$

1.8.3 Binary Multiplication

Binary multiplication is similar to decimal multiplication. Multiply the multiplicand by one bit of the multiplier at a time starting from the LSB of the latter. (Doing one such multiplication by a machine involves a series of left shifting and addition operations). As we move to the left of the multiplier this way, we shift the results of each new multiplication to the left by one bit. We continue this process for all the bits in the multiplier and at the end add the results of all multiplications to get the final result.

Example 1.20: Multiply 1110_2 by 1011_2

$$\begin{array}{r} 1110 \\ 1011 \times \\ \hline 1110 \\ 1110 \\ 1110 \end{array}$$

$$\begin{array}{r}
 0000 \\
 1110 \\
 \hline
 10011010
 \end{array}$$

$$1110_2 \times 1011_2 = 10011010_2$$

1.8.4 Binary Division

Binary division is similar to the decimal division.

Example 1.21: Divide 1110_2 by 10_2

$$\begin{array}{r}
 111 \\
 10 \overline{) 1110} \\
 \underline{10} \\
 11 \\
 \underline{10} \\
 10 \\
 \underline{10} \\
 0
 \end{array}$$

$$1110_2 \div 10_2 = 111_2$$

The same answer can be obtained by right shifting and subtracting.

1.9 Representation of Data

In the following, we will consider the representation of numeric data (i.e., numbers) and non-numeric data in computers separately. First, let us discuss something general and common to both.

Data are initially stored in memory and moved into *registers* in the processor for processing; after processing, the results are first stored in registers and then moved to memory. A register can be thought of as an array of locations each of which can store one bit. The number of bits a register can hold is limited. Fig. 1.2 shows a register that can store n bits.

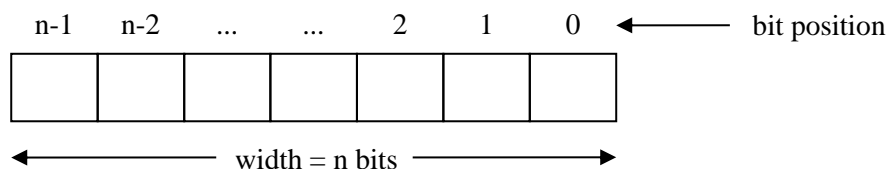


Figure 1.2: Bit positions in an n -bit register

A processor whose registers have a maximum width of n bits is called an *n -bit processor* and we say its *word size* is n . The word size is the maximum width of a data item that the processor can handle in a single operation.

In an n -bit register, the LSB has bit position 0 and the bit position of the MSB is $(n-1)$. Since each bit can represent two combinations (data items) as 0 and 1, an n -bit register can represent 2^n combinations (or distinct data items).

For example, in an 8-bit processor, the word size is 8 and will have registers that are 8-bits wide. Such a register can store $2^8=256$ distinct data items. Similarly, a 16-bit processor would have 16-bit wide registers that can represent $2^{16} = 65536$ distinct data items.

1.10 Representation of Integers

Numeric data (i.e., numbers) can be categorised as *integers* and *non-integers* (the latter is also referred to as *floating-point* numbers). Integers denote whole numbers while non-integers denote numbers with fractions. Let us consider integers in this section.

Integers can be further categorised as *unsigned* and *signed*. Unsigned numbers denote only the positive values while signed numbers denote both positive and negative values.

1.10.1 Unsigned Integers

Unsigned integers denote whole numbers which are only positive.

For an n -bit register, the following shows the range of unsigned integers that can be represented.

Possible number of values	$= 2^n$
Minimum (smallest integer)	$= 0$ (when all bits are zero)
Maximum (largest integer)	$= 2^n - 1$ (when all bits are one)

With a single byte (8 bits) we can represent $2^8 = 256$ unsigned integers, where zero (00000000_2) is the minimum and 255 (11111111_2) is the maximum. Similarly with 16 bits, we can represent $2^{16} = 65536$ unsigned integers, where 0 is the minimum and 65535 is the maximum.

Consider the following binary addition.

Example 1.22: Add 111110_2 and 11000111_2

$$\begin{array}{r} 00111110 \\ 11000111 + \\ \hline 100000101 \end{array}$$

In this case the 6-bit integer 111110_2 and the 8-bit integer 11000111_2 are added and the result is a 9-bit integer 100000101_2 . If the processor has only 8-bit registers it will not be able to store this 9-bit result in a register. It will only hold the last 8 bits; so the stored result will be 00000101_2 . (That is, the MSB of the correct result cannot be included in the register). In a situation in which the result of a computation is bigger than what a register can accommodate, we say an *overflow* has occurred.

1.10.2 Signed Integers and Two's Complement Method

Signed integers include both positive and negative integers. Signed integers in computers are represented using the two's *complement* method. In this method, the exact representation of an integer depends on the word size (i.e., the number of bit positions); this means, an integer will have seemingly different representations with different word sizes.

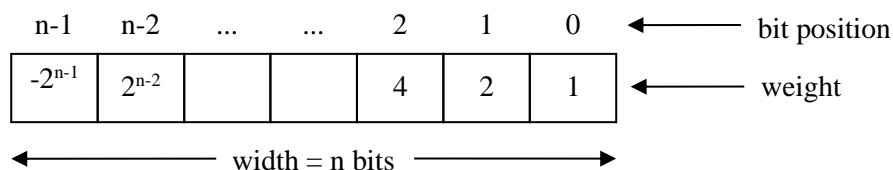


Figure 1.3: Weights of bit positions in two's complement method

As shown in Fig 1.3, in the two's complement method, the MSB determines the sign. If the MSB = 1, the integer is negative; if the MSB = 0, then the integer is either 0 or positive.

The MSB also has a positional value (weight); in an n-bit integer, the positional value of the MSB is -2^{n-1} .

Let $b_{n-1} b_{n-2} \dots b_2 b_1 b_0$ be an n-bit integer in two's complement form (each $b_i = 0$ or 1). Then the value, x , of this integer is given by the following:

$$x = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

With the above, now we can list the smallest and largest integers represented by n bits.

Minimum (smallest integer) $= -2^{n-1}$ (when MSB=1, all other bits are zero)

Maximum (largest integer) $= 2^{n-1} - 1$ (when MSB=0, all other bits are one)

The following Table 1.3 shows the signed integers in two's complement representation for n=4.

Table 1.3: Two's complement representation for n=4

Two's complement	Decimal value		Two's complement	Decimal value
0000	0		1000	-8
0001	1		1001	-7
0010	2		1010	-6
0011	3		1011	-5
0100	4		1100	-4
0101	5		1101	-3
0110	6		1110	-2
0111	7		1111	-1

For n=8, in a similar manner, we get the following:

zero	$= 0_{10}$	$= 0000\ 0000_2$
smallest integer	$= -128_{10}$	$= 1000\ 0000_2$
largest integer	$= 127_{10}$	$= 0111\ 1111_2$
minus 1	$= -1_{10}$	$= 1111\ 1111_2$

1.10.3 How to Obtain Two's Complement Representation

Let d be a positive decimal integer. We can obtain the two's complement representation b of $(-d)$ as follows.

Step 1: Decide the number of bits (n) to use.

Step 2: Get the binary representation b_1 of d (with MSB=0); note that n should be large enough, else increase n .

Step 3: Get the one's complement b_2 of b_1 . The one's complement of a binary number is obtained by flipping (complementing) every bit; that is, change every 0 to 1 and 1 to 0 in b_1 to obtain b_2 .

Step 4: Get the final result, the two's complement b of $(-d)$ as $b = b_2 + 1$.

Note that adding 1 to the one's complement of a number is equivalent to subtracting the number from 2^n .

Example 1.23: Find the two's complement form of -5_{10} for $n=4$.

Let's follow the above steps:

$+5_{10}$ in binary for $n=4 \rightarrow 0101_2$

one's complement of $0101_2 \rightarrow 1010_2$

Add 1 $\rightarrow 1010_2 + 1_2 = 1011_2$, which is the final result (two's complement of -5_{10})

Exercise: Obtain the two's complement form of -5_{10} for $n=8$ and verify that it is $1111\ 1011_2$.

Example 1.24: In Python, we can verify the two's complement form of -1 as follows.

```
>>> ~0
-1
>>> ~(-1)
0
```

1.11 Representation of Real Numbers (Floating-point Representation)

Real numbers (that is, non-integers) are more commonly referred to as *floating-point numbers* in our context because of the method of representation, which is called *floating-point*. We will discuss floating-point representation in this section.

Note that floating-point is not the only way to represent real numbers in computers. In *fixed-point* representation (not used in modern computers), a fixed number of digits appear before and after the radix point.

1.11.1 The Concept of Floating-Point Representation

In the floating-point representation, there is no fixed number of digits before or after the radix point. In fact, the radix point can *float* in the left or right direction. This is a method of representing *an approximation of a real number* in a way that can support a wide range of values.

The concept of the floating-point representation is to approximate a real number using a fixed number of digits called the *mantissa* (or the *significand*) and scale it using an *exponent* with respect to a *base*. Thus, this is similar in concept to the *scientific notation* of numbers.

In summary, a real number N can have its floating-point representation as follows:

$$N = \text{mantissa} \times \text{base}^{\text{exponent}}$$

Example 1.25: If we decide to use decimal floating-point representation with a 5 digit mantissa, the decimal point after the first digit from left and the base as 10, then the numbers 0.12345, 12345, 123450 and 123452 will be represented as follows:

- (a). 0.12345 $= 1.2345 \times 10^{-1}$
- (b). 12,345 $= 1.2345 \times 10^4$
- (c). 123,450 $= 1.2345 \times 10^5$
- (d). 123,452 $\approx 1.2345 \times 10^5$

In all cases (a)-(d) above, the mantissas are the same; the decimal (radix) point floats to the left or right based on the scaling given by the exponent. In cases (c) and (d), the exponents are the same too. Note that in case (d), due to the 5 digit limit in the mantissa, the value 123,452 is not exactly represented.

As can be seen from the above example, the *precision* is determined by the width of the mantissa. In the example, we use only 5 digits for the mantissa, i.e., there is no way to represent a 6th digit. As a result, the two numbers 123,450 and 123,452 have the same representation of 1.2345×10^5 . Being constrained to use a finite number of digits for the mantissa is a practical limitation we face in many situations, including, especially, in computers. Thus, as exact representation is not possible, this is a method of representing *an approximation of a real number*. Therefore, when we use computers for computations that involve real numbers, we have to be mindful of this important fact.

The scaling of the mantissa with the exponent gives us an expanded range of numbers to represent.

1.11.2 The IEEE Standard on Floating-Point Arithmetic

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation established by the *Institute of Electrical and Electronics Engineers* (IEEE). Almost all modern processors use the IEEE 754 standard. The standard addressed many problems found in the different floating point implementations that existed previously.

The IEEE standard defines:

- arithmetic formats: sets of binary and decimal floating-point data; e.g., finite numbers, infinities, and special "not a number" values ("NaN"s)
- interchange formats: encodings (bit strings) that may be used to exchange floating-point data
- rounding rules: properties to be satisfied when rounding numbers during arithmetic
- operations: arithmetic and other operations on arithmetic formats
- exception handling: indications of exceptional conditions (e.g., division by zero, overflow)

Among the basic formats defined in the IEEE standard, the most common two are the *single-precision* (32-bit) binary format, also known as *binary32*, and the *double-precision* (64-bit) binary format, also known as *binary64*.

Both representations above use three fields:

- sign bit (represents the sign of the number)
- exponent (in biased representation, implied base is 2)
- mantissa (represents a fraction, has an implied bit)

The precision of the basic formats is one bit more than the width of its mantissa. The extra bit of precision comes from an implied (hidden) leading "1" bit. The typical floating point number will be normalized such that the MSB of its mantissa will be a "1". If the leading bit is known to be "1", then it need not be encoded in the format. Table 1.4 shows the details of the two formats.

Table 1.4: Basic IEEE Floating-Point formats

	Number of Bits				Min Exponent	Max Exponent
	Sign	Mantissa	Exponent	Total		
Single-precision	1	23	8	32	-126	+127
Double-precision	1	52	11	64	-1022	+1023

Example 1.26: Here is how the number 0.15625_{10} will be represented in the IEEE single-precision format.

- First, we have $0.15625_{10} = 0.00101_2$.
- Next, let's express this binary number 0.00101_2 such that there is one "1" bit left of binary point, i.e., $0.00101_2 = 1.01_2 \times 2^{-3}$.
- In single precision, the exponent is biased by 127_{10} . Therefore, in this case the exponent (-3_{10}) will be represented in biased form as $(-3_{10} + 127_{10}) = 124_{10} = 01111100_2$.
- The mantissa will be, based on above, the fraction $.01_2$ (with an implied leading "1").

The overall representation of 0.15625_{10} in IEEE single-precision format is as shown in Fig. 1.4.

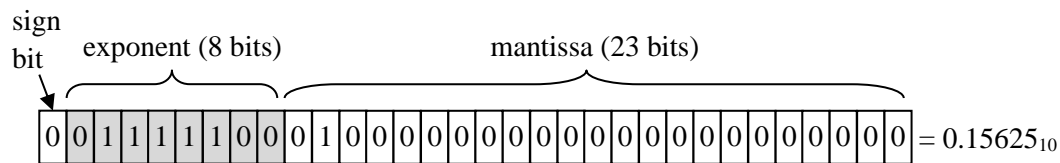


Figure 1.4: Representation of 0.15625_{10} in IEEE single-precision format

1.11.3 Floating-Point Arithmetic in Python

Python "float" types are in double-precision format, which is equivalent to "double" type in the C language (i.e., 64 bits, with 53 bits for the mantissa).

As we have seen, unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers we enter are only approximated by the binary floating-point numbers actually stored in the computer.

For example, the value stored internally when you enter the decimal number 0.1 is the binary fraction $0.00011001100110011001100110011001100110011001100110011010_2$ which is close to, but not exactly equal to, $1/10$.

It's easy to forget that the stored value is an approximation to the original decimal fraction, because of the way that floats are displayed at the interpreter prompt. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. If Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

That is more digits than most people find useful, so Python keeps the number of digits manageable by displaying a rounded value instead, as follows:

```
>>> 0.1
0.1
```

It's important to realize that this is, in a real sense, an illusion: the value in the machine is not exactly $1/10$, you're simply rounding the *display* of the true machine value. This fact becomes apparent as soon as you try to do arithmetic operations with these values.

Example 1.27: What will be the result if we add 0.1 and 0.2 in Python?

Here is the answer:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Note that the above is in the very nature of binary floating-point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

Example 1.28: What will be the result if we round the value 2.675 to two decimal places in Python?

Here is the answer:

```
>>> round(2.675, 2)
2.67
```

How could this be possible? The built-in `round()` function rounds to the nearest value, rounding ties away from zero. Since the decimal fraction 2.675 is exactly halfway between 2.67 and 2.68, you might expect the result here to be (a binary approximation to) 2.68. It's not, because when the decimal string 2.675 is converted to a binary floating-point number, it's again replaced with a binary approximation, whose exact value is:

```
2.67499999999999982236431605997495353221893310546875.
```

Since this approximation is slightly closer to 2.67 than to 2.68, it's rounded down.

Example 1.29: What will be the result if we sum ten values of 0.1 in Python? Since 0.1 is not exactly $1/10$, this may not yield exactly 1.0, as seen below.

```
>>> sum = 0.0
>>> for i in range(10):
    sum += 0.1

>>> sum
0.9999999999999999
```

You should be able to explain the following results we get from Python.

```
>>> .1 + .1 + .1 == .3
False
>>> 0.1 + 0.1 == 0.2
True
>>> 0.15 + 0.15 >= 0.1 + 0.2
False
```

1.12 Representation of Non-numeric Data

As stated, data such as the name of a student, the national identity card number and a telephone number are examples of non-numeric data. In these, the elements are symbols or characters, which can be: letters such as A-Z and a-z; numerical digits 0-9; symbols such as &, %, <, >, @, #, \$; and control characters such as; <CR>, <LF>, <BEL>, <ESC>, ¹. Other than these, we have to be able to represent characters (symbols) used in many languages (e.g., Sinhala, Tamil) as well.

The computer internally has only two “characters”: 0 and 1. So the question is: how can we represent our characters listed above using 0 and 1? Note that in this context there is no numerical value associated with any character; each is simply a symbol. A simple solution is to assign a unique pattern (code) of 0s and 1s for each of our characters (i.e., *encode* each using 0 and 1). Since there are many possible ways of doing this, we need a standard so that every computer can use the same encoding to enable sharing of data among computers.

The standards used for character encoding are ASCII and Unicode.

1.12.1 ASCII

American Standard Code for Information Interchange (ASCII) is the historically most widely used alphanumeric code set. It is used for representing uppercase and lower case English letters (i.e., Latin alphabet), digits and punctuations. There are 128 standard ASCII codes which are represented using 7-bits (also called 7-bit ASCII). In actual data storage and transmission 8-bits are used and the MSB is called the *parity*² bit. Selected set of ASCII codes are given in Table 1.5.

Table 1.5 : Selected set of ASCII codes (in equivalent decimal and Hex values)

ASCII Decimal	ASCII Hex.	Symbol	ASCII Decimal	ASCII Hex.	Symbol	ASCII Decimal	ASCII Hex.	Symbol
0	0	NUL	48	30	0	65	41	A
1	1	SOH	49	31	1	66	42	B
2	2	STX	50	32	2	67	43	C
3	3	ETX	51	33	3	68	44	D
4	4	EOT	52	34	4	69	45	E
5	5	ENQ	53	35	5	70	46	F
6	6	ACK	54	36	6
7	7	BEL	55	37	7
8	8	BS	56	38	8	87	57	W
9	9	TAB	57	39	9	88	58	X
10	A	LF	58	3A	:	89	59	Y
11	B	VT	59	3B	;	90	5A	Z
12	C	FF	60	3C	<
13	D	CR	61	3D	=	97	61	a
14	E	SO	62	3E	>	98	62	b
15	F	SI	63	3F	?	99	63	c

1.12.2 Unicode

Although ASCII is heavily used it cannot represent more than 128 characters. However, there are hundreds of alphabets (character sets or *writing systems*) in the world, and they have many characters in them. So, it is not possible to use ASCII to represent characters in a language like Sinhala or Tamil.

¹ CR – Carriage Return, LF – Line Feed, BEL – Bell, ESC – Escape, DEL - Delete

² The parity bit is used to check errors in data transmission and storage.

(Many proprietary schemes got around this problem, but there is no standard way for this). Therefore, Unicode was adopted as a standard coding system to support a large number of writing systems.

The Unicode standard uses 32 bits to overcome the limitation of the number of characters. Unicode system can represent multiple character sets, including Sinhala and Tamil. The range 0x0D80 (in hexadecimal) to 0x0DFF is assigned for Sinhala characters, as shown in Figure 1.5. The range 0x0B80 to 0x0BFF is assigned for Tamil characters, as shown in Figure 1.6.

0D80		Sinhala						0DFF	
		0D8	0D9	0DA	0DB	0DC	0DD	0DE	0DF
0			ඌ 0D90	ඌ 0DA0	ඌ 0DB0	ඌ 0DC0	ඌ 0DD0		
1			ඌ 0D91	ඌ 0DA1	ඌ 0DB1	ඌ 0DC1	ඌ 0DD1		
2		ඌ 0D82	ඌ 0D92	ඌ 0DA2		ඌ 0DC2	ඌ 0DD2		ඌ 0DF2
3		ඌ 0D83	ඌ 0D93	ඌ 0DA3	ඌ 0DB3	ඌ 0DC3	ඌ 0DD3		ඌ 0DF3
4			ඌ	ඌ	ඌ	ඌ	ඌ		

Figure 1.5 : Representation of some Sinhala characters in Unicode

0B80		Tamil								0BFF	
		0B8	0B9	0BA	0BB	0BC	0BD	0BE	0BF		
0			ஐ 0B90		ர 0BB0	ீ 0BC0			ய 0BF0		
1					ற 0BB1	ு 0BC1			ள 0BF1		
2		ஃ 0B82	ஒ 0B92		ல 0BB2	ூ 0BC2			சு 0BF2		
3		ஃ 0B83	ஒ 0B93	ஃ 0BA3	ள 0BB3				உ 0BF3		

Figure 1.6 : Representation of some Tamil characters in Unicode

The Unicode standard describes how characters are represented by *code points*. A code point is an integer value, usually in base 16 (i.e., hexadecimal), as seen in Fig. 1.5 and 1.6. For e.g., the code points 0x0DC1 and 0x0BB1 represent the Sinhala and Tamil characters 'ඌ' and 'ஊ', respectively.

In Python source code, Unicode literals are written as strings prefixed with the ‘u’ or ‘U’ character. Specific code points can be written using the `\u` escape sequence, which is followed by four hex digits giving the code point. The `\U` escape sequence is similar, but expects 8 hex digits, not 4.

For example, in Python source code,

- `u'abcdefghijkl'` represents the string of characters 'abcdefghijkl'.
- `u'\u0dc1'` represents the Sinhala character 'ශ'.
- `u'\u0bb1'` represents the Tamil character '௩'.

Python has a built-in function, `chr(i)` which takes in an integer `i` and returns the string representing the corresponding Unicode character. Inversely, the built-in `ord(c)` function takes in one Unicode character `c` and returns an integer representing the Unicode code point of that character.

Here is an example Python session.

```
>>> s = chr(0x0dc1)
>>> print(s)
ශ
>>> print(u'\u0bb1')
௩
>>> hex(ord('௩'))
'0xd86'
```

2 Data Structures and Algorithms

The study of data structures and algorithms will help us to identify existing structures and algorithms that we can re-use to solve new problems. A certain solution that was used to solve a particular problem may be applicable to other problems. So, it would be beneficial to identify types of solutions that others have used so that we can adopt and use them to solve new problems that we have. In certain scenarios, the existing data structures and algorithms may not be enough to solve a problem and you may modify them to match your requirements. In this section, you will study the structures that hold data and algorithms that can be used to manipulate data.

2.1 Abstract Data Types (ADT)

When designing a solution for a problem we may want to think of the solution in terms of data structures that can be used to store the data that is relevant to the problem. If a suitable data structure is picked it will help you in solving the problem in an efficient manner.

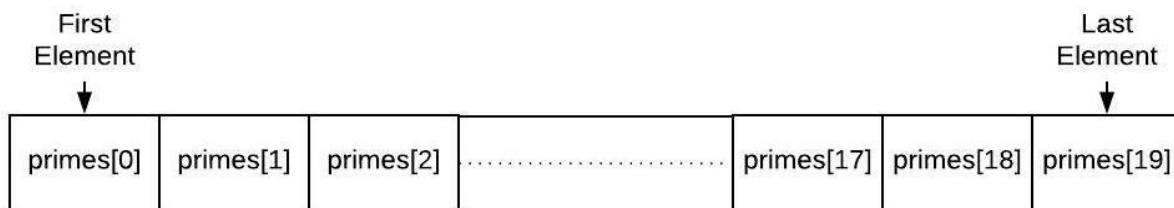
In computer science we use several well-known data structures that can be used to hold data. Arrays, Linked lists, Stacks, Queues, Trees, and Graphs are some of these data structures. All of these data structures have a way of storing data, a specific set of methods for accessing and manipulating data and the structure. All these methods are standard methods, and their operations are well defined. However, the programmers can implement them in different programming languages and different ways. These data structure specifications are called *abstract data types* (ADTs) as they are abstracted out at a higher level and are not specific to any one programming language. However, some programming languages will provide default implementations of these data structures as standard libraries. For example, Python has methods implemented in `list` data type that will allow them to be used as stacks or queues. In addition, Python has a “deque” data type, which implements the queue data structure.

When solving problems, it is beneficial for us to think of these abstract data types without having to worry about the implementation details. As an abstract data type is defined as a set of data items and fundamental operations on the data, when solving problems, we can simply assume we have such a data type available in coming up with the solution. When implementing the solution, we can use a library implementation of that data type if it is available in the programming language we selected or use our own implementation of the abstract data type. In the next subsections, we will introduce you to the ideas of some basic ADTs (stacks, queues, etc.) and how they can be implemented.

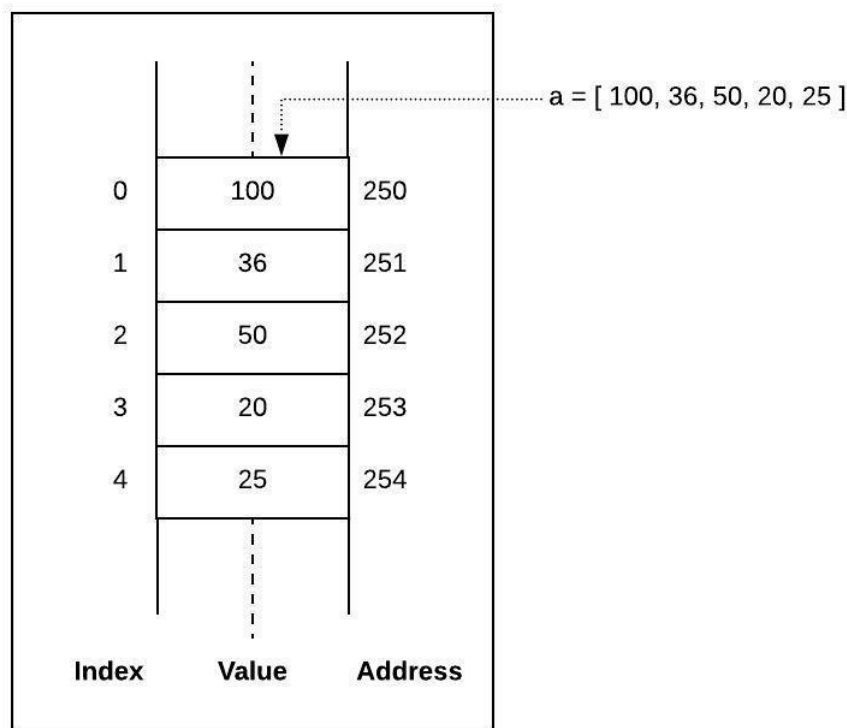
2.2 Introduction to Arrays

Array is a kind of data structure that can store a fixed-size sequential collection of elements of the same type. This makes it easier to calculate the position of each element by simply adding an offset to a base value or the memory location of the first element of the array.

Consider, you need to store the first 20 primes. Instead of declaring individual variables for each prime, such as `prime_1`, `prime_2`, ..., and `prime_20`, you declare one array variable such as “primes” and use `primes[0]`, `primes[1]`, and ..., `primes[19]` to represent individual variables as follows.



All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element. Arrays are often represented with diagrams that represent their memory use. The diagram below is one typical way to represent the memory used by an array of elements 100, 36, 50, 20 and 25.



In Python, lists can be used in place of arrays (Python does not have built-in support for a type or structure called ‘array’). Python lists are dynamic arrays where we can add a new element to the array simply by using `append()`, without considering the maximum length.

See the Appendix 6.1 on how to declare and use arrays in the C language.

2.3 Introduction to Stacks

A stack allows us to put things from the top and take things out from the top. All the activities of a stack take place at the top. A stack allows us to process the last item that is placed on the stack first. In other words it allows us to do Last-In-First-Out (LIFO) type of processing. A stack can be compared to a stack of books that we have on the table. When we want to take a look at the bottom we may be required to first remove the books at the top and get to the bottom of the stack of books. So we would be removing the books that we added last, first from the stack.

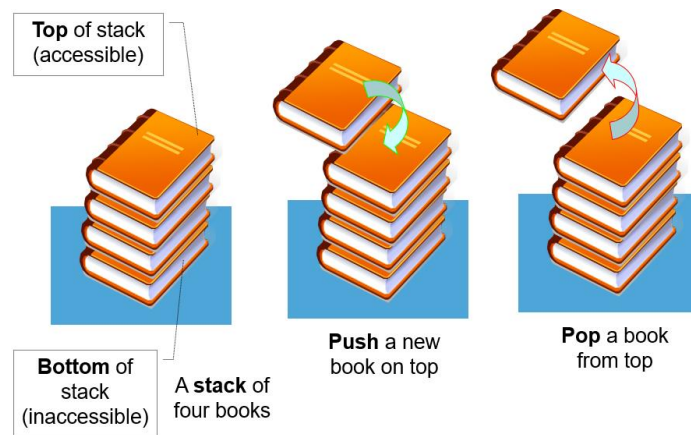
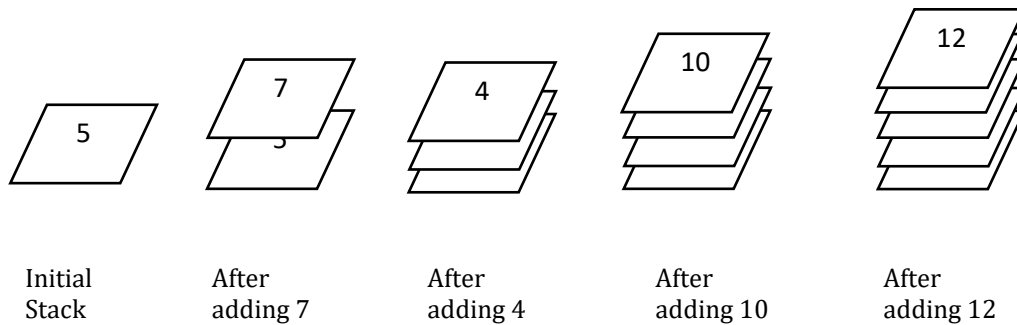
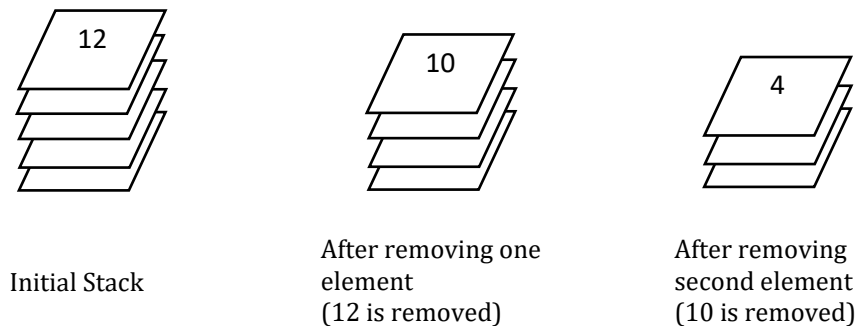


Figure 1 : Stack of books (Source: <https://visualgo.net/en/list?slide=4>)

Similar to the stack of books shown in the above picture, we can have a stack of numbers as well. In such a data structure, we can add numbers to the top (“push”) and remove numbers from the top (“pop”), one at a time. For example a stack of numbers initially contains number 5 and the resultant stack after adding numbers 7, 4, 10 and 12 is shown below.



Now if we remove two items from the above stack, the resultant stack is shown below.



A more descriptive diagram of the above stack operations are given below.

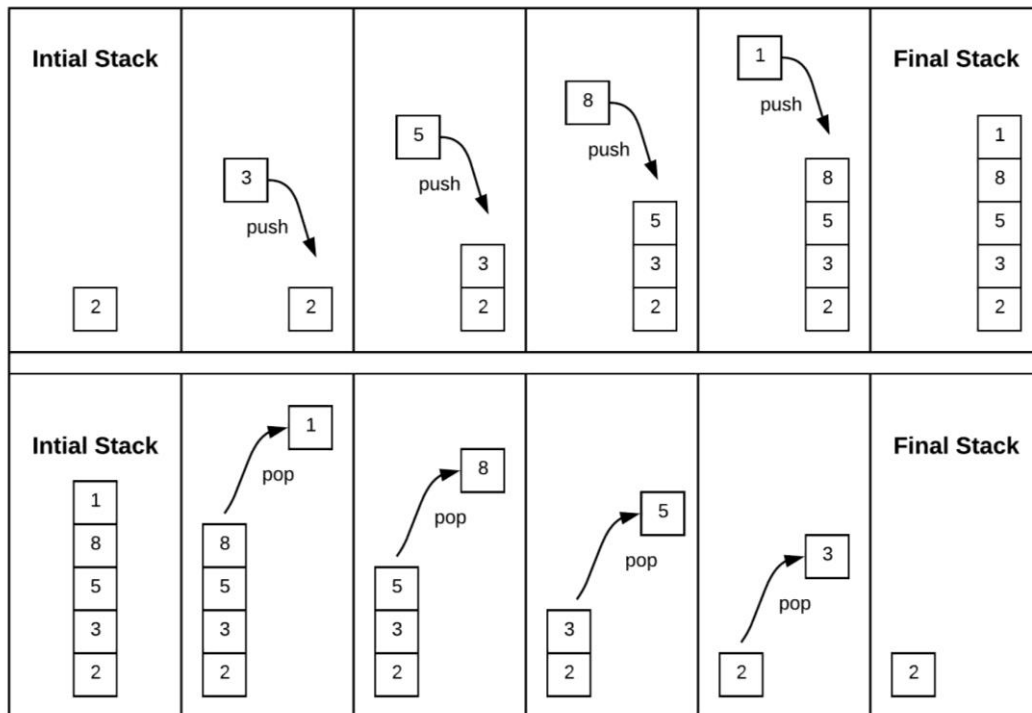


Figure 2 : Stack Operations Illustration

As we discussed earlier each data structure will have a set of operations that can be used on the data structure such as adding elements to the structure and removing things from the structure. In the case of a stack, those operations and the requirements are given below.

- **stack.push(element):** This will place the element on the top of the stack.
- **stack.pop():** Will remove the top element in the stack and return it provided that the stack is not empty.

2.3.1 Using a List as a Stack in Python

In Python, a fixed size list can be used to create a stack. Since it doesn't need to specify a specific data type for a Python variable, the stack can also be in any data type.

The list data type in Python can be used as a stack using the two methods `append()` and `pop()`. The `append` method adds the specified element to the end of the list. As the new elements have to be added to the top of the stack, then we can assume the end of the list as the top of the stack. For example;

- If `my_stack` is `[1, 2, 3, 4]`, element 4 will be considered as the top element of the stack `my_stack`.
- If we call `my_stack.append(5)`, it will create the list (stack) `[1, 2, 3, 4, 5]` where 5 is now at the end of the list (top of the stack).

Recall that the stack is a *last-in-first-out* structure. Therefore, if we are removing an element from the stack we have to remove the top most element of the stack. As of our representation, the top most element of the stack is the last element of the list. The `pop()` method, if no parameter is specified, removes the last element of the list. Thus, we can simply use the `pop()` method in the list as the `pop` method of our stack. For example, `my_stack.pop()` will remove the newly added element (5) from the stack (list) and return it.

2.3.2 Stack Implementations in C

A stack can be implemented in C with either arrays or linked lists. Array based stack can be implemented in two ways, one as a fixed size stack and other method with pointers and structs.

Refer Appendix 6.2 for a sample C implementation using fixed size arrays. Additional methods, **isEmpty()** and **top()** are used to determine whether the stack is empty or not before popping an item and to return the top object without removing it, respectively.

An implementation of an integer stack with structs and pointers in C is given in Appendix 6.3.

2.3.3 Using a Stack to Evaluate Mathematical Expressions

Stacks are used in many different situations in computing, and, as discussed earlier, the execution of programs (especially recursive programs) uses a stack memory. Another use of stack is for evaluation of postfix expressions. Postfix expression (or expression in Reverse Polish notation) is an expression in which the operator follows all its operands. A few mathematical expressions and their postfix representations are shown below.

Mathematical Expression (Infix Notation)	Postfix Representation
$2 - 3 + 4$	$2\ 3\ -\ 4\ +$
$2 + 3 * 4$	$2\ 3\ 4\ *\ +$
$2 * (3 + 4)$	$2\ 3\ 4\ +\ *$
$2 * 3 + 4 * 5$	$2\ 3\ *\ 4\ 5\ *\ +$
$(2 + 3) * (4 + 5)$	$2\ 3\ +\ 4\ 5\ +\ *$
$2 + 3 * 4 + 5$	$2\ 3\ 4\ *\ +\ 5\ +$

Postfix expressions can be evaluated very easily using a stack. What has to be done is scan the expression from left to right and;

- If we find an operand, we will have to store it in a stack.
- If we find an operation, we will have to retrieve the required number of operands from the stack, then perform the operation and then store the result back in the stack.

At the end, the stack should have a single value and that is the value of the given mathematical expression in postfix notation.

2.3.4 Towers of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and a stack of **n** disks. The objective is to move the entire disk stack to another rod, obeying the following simple rules:

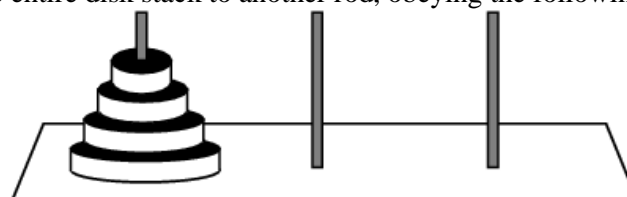


Figure Tower of Hanoi (source: <http://mathworld.wolfram.com/TowerofHanoi.html>)

- Only one disk can be moved at a time.

- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack ,i.e., a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk

Graphical illustration of solving the Tower of Hanoi problem where $n=3$ is given in Fig. 4.

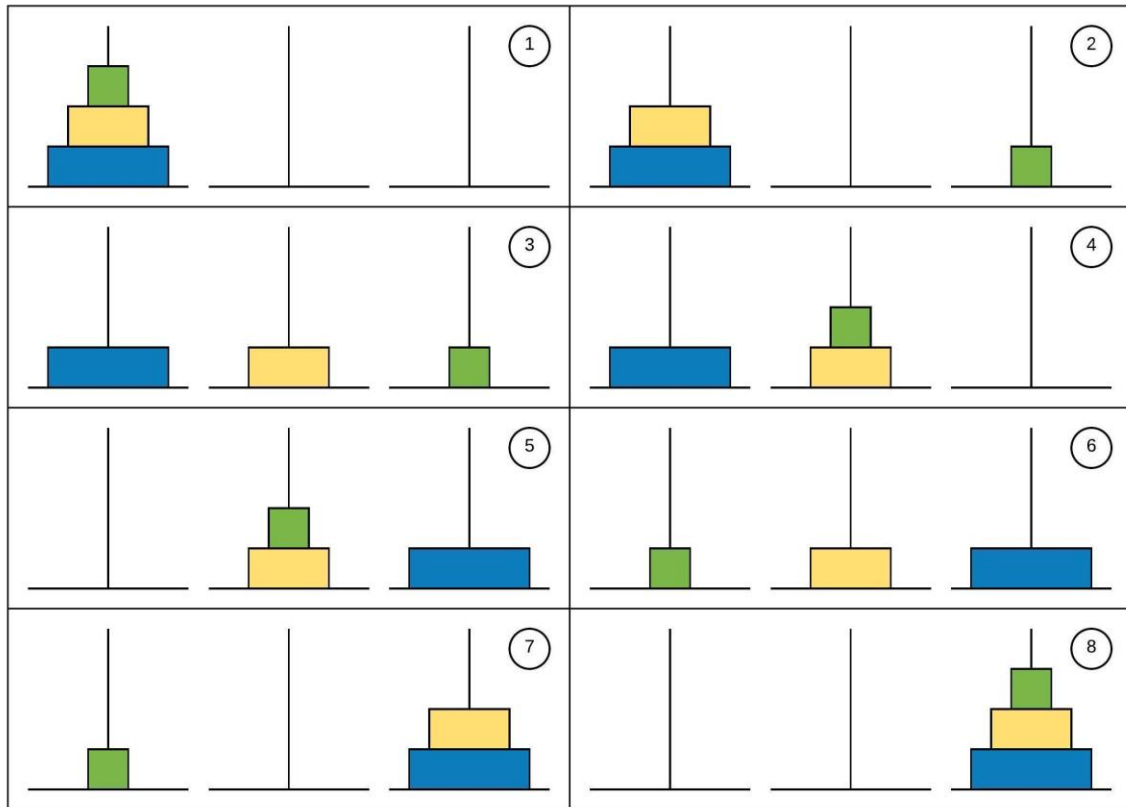


Figure 4 : Sample Solution to Tower of Hanoi

Exercise 4.1 Develop a C program to model Tower of Hanoi problem for the given scenario in Figure 4 using stacks. You may use any of the given implementations for stack.

Exercise 4.2 Develop a Python program to model Tower of Hanoi problem for the given scenario in Figure 4 using stacks.

Exercise 4.3 Develop a Python program that can take a postfix mathematical expression as the input (you can assume the mathematical expression is available in a list) and evaluate it and give the answer as the output.

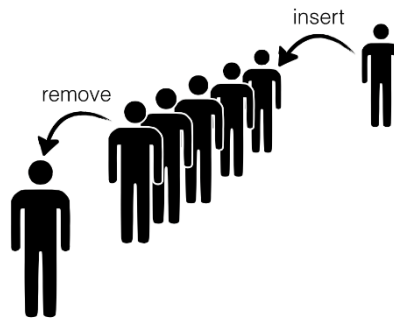
Exercise 4.4 Develop a Python program that can be used to enter a list of integers into a stack and then add the top two elements of the stack and push the total to the stack until there is only one element in the stack. At the end of each execution the total of the given set of integers should be the only remaining element in the stack should be displayed to the user.

Exercise 4.5 Develop a Python program for ABC company to resolve its parking problem. Employee vehicles are parked in a narrow road where two cars cannot pass each other. The road has only one entry with the other end having a dead-end. First car to be parked will be the last car that can

be removed. When a car other than the last car to be parked needs to be taken out all the cars that were parked after that would need to be removed. Write a program that can be used to maintain the list of cars that are in the park at any given time. When a car needs to be removed the owner would need to collect car keys from others who have parked behind him. The program should give a list of cars that needs to be removed in order to take a specific car out of the park. The program should also maintain the remaining list of cars. You can assume that the number plate of a car can be represented using an integer number.

Exercise 4.6 Develop a Python program that can take a mathematical expression as the input (you can take that the mathematical expression is available in a list) and check if it has matching parenthesis.

2.4 Introduction to Queues



(Image Source: <http://ucsd-progsys.github.io/liquidhaskell-tutorial/09-case-study-lazy-queues.html>)

Figure 5: A queue

Queue is another data structure which allows us to input items at the back and take items out from the front. A queue can be compared to the queue in the canteen or the queue in the bank, where the people get served in the order they joined the queue. Queue is a First-In-First-Out (FIFO) structure, which allows us to process the items in the order they are received. In computer science queues are used in applications like printer queues to manage printing jobs and operating systems for process scheduling.

A queue has to support two basic operations, inserting and removing elements from the queue.

- The insertion operation is called “enqueue” and it has to insert the specified item to the end of the queue.
- The removal operation is called “dequeue” and it removes the first item in the queue.

The following are the main operations on a queue. Front and Rear are two additional operations.

- Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
- Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
- Front: Get the front item from the queue.
- Rear: Get the last item from queue.

A more descriptive diagram of queue operations is given below.

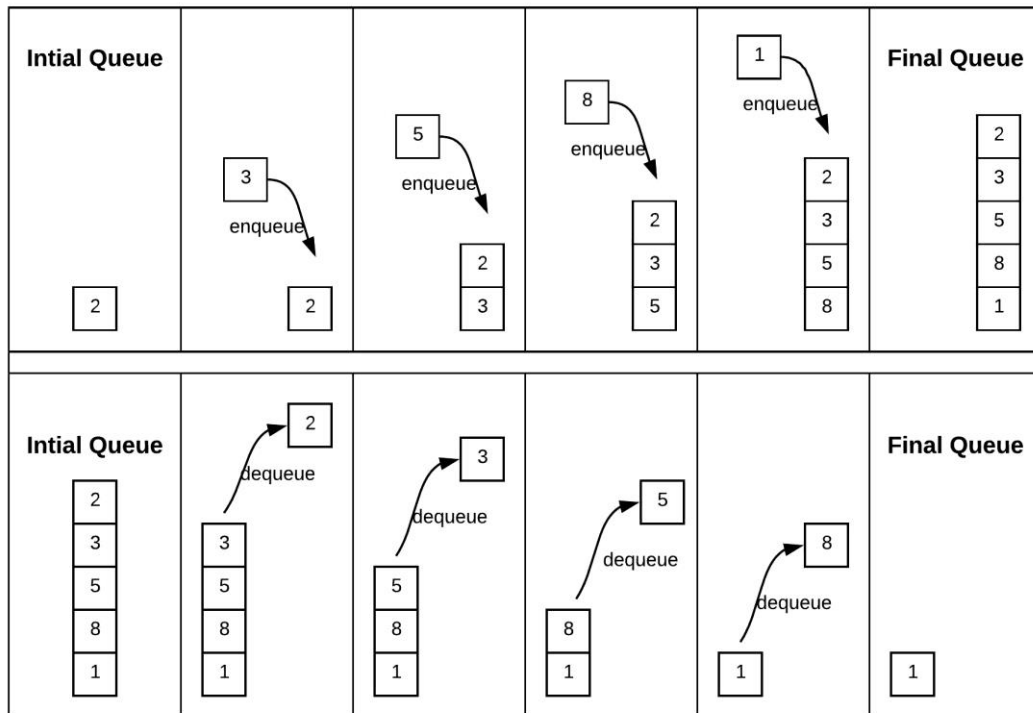
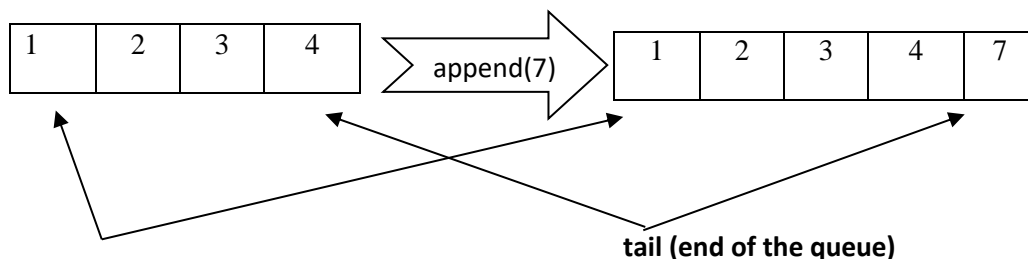


Figure 6: Queue operations

2.4.1 Using a List as a Queue in Python

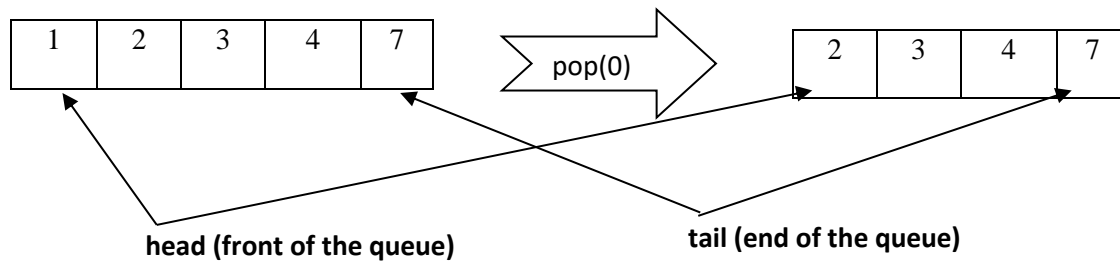
Similar to the stack, a queue can be implemented using the list data type in Python with the two methods `append()` and `pop()`. The `append()` method adds the specified element to the end of the list, which is the perfect match for the queue. Thus, the end of the list is considered as the end of the queue. For example;

- If `my_queue` is `[1, 2, 3, 4]`, 4 will be considered as the last element (end) of the queue `my_queue`.
- If we call `my_queue.append(7)`, it will create the list (queue) `[1, 2, 3, 4, 7]` where 7 is now at the end of the list (queue).



As a queue is a *first-in-first-out* structure, if we are removing an element from the queue we have to remove the first element of the queue. As of our representation, the first element of the queue is the first element of the list. Thus, we can use the `pop()` method with parameter 0 to remove the first element in the list, which will be the dequeue operation. For example, `my_queue.pop(0)` will remove the

first element (1) from the queue (list) and return it. However, it should be noted that using a list as a queue is not efficient in Python and the data type *deque* is more suited for a queue.



2.4.2 Queue using Dynamic Arrays in C

A sample C implementation of “Queue” using dynamic arrays is given in Appendix 6.3.

2.5 Introduction to Binary Trees

A binary tree is made of nodes, where each node contains a “left” reference, a “right” reference, and a data element. The data element is sometimes called the “key”. The topmost node in the tree is called the “root”.

Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called the “parent”. On the other hand, each node can be connected to a maximum of 2 nodes below, called “children”. Nodes with no children are called “leaves” or “leaf nodes”. Nodes which are not leaves are called “internal nodes”. Nodes with the same parent are called “siblings”.

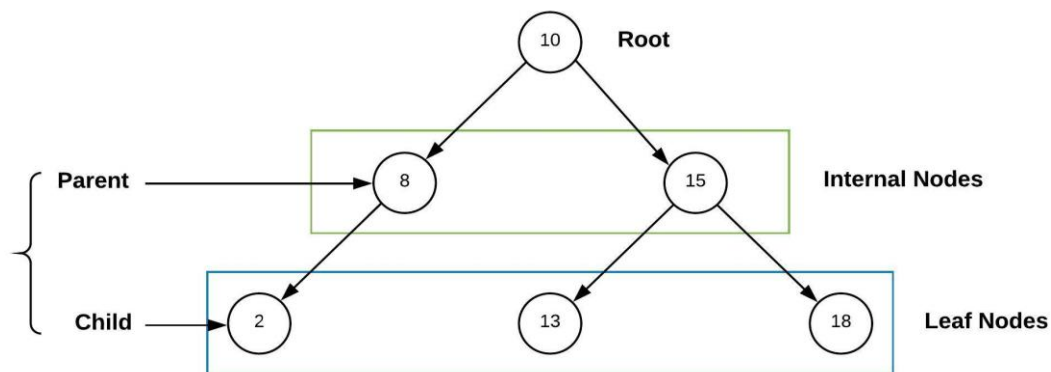


Figure 7: A binary tree

In the binary tree shown in Fig. 7, the data elements (keys) stored are 2, 8, 10, 13, 15 and 18.

More tree terminology:

- The depth of a node is the number of edges from the root to the node.
- The height of a node is the number of edges from the node to the deepest leaf.
- The height of a tree is a height of the root.
- A full binary tree is a binary tree in which each node has exactly zero or two children.
- A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.

2.5.1 Binary Search Trees (BST)

A binary search tree (BST) is a binary tree in which all the nodes have the following property (which is known as the *binary search tree property* or the BST property):

- The left child of a node x has a key less than or equal to the key of node x .
- The right child of a node x has a key greater than or equal to the key of node x .

Fig. 7 above shows a binary search tree, and we can see every node in it satisfies the BST property.

Thus, a BST divides the sub-trees of a node into two segments: the *left sub-tree* and the *right sub-tree* and the BST property can be described as:

$$\text{left_subtree (keys)} \leq \text{node (key)} \leq \text{right_subtree (keys)}$$

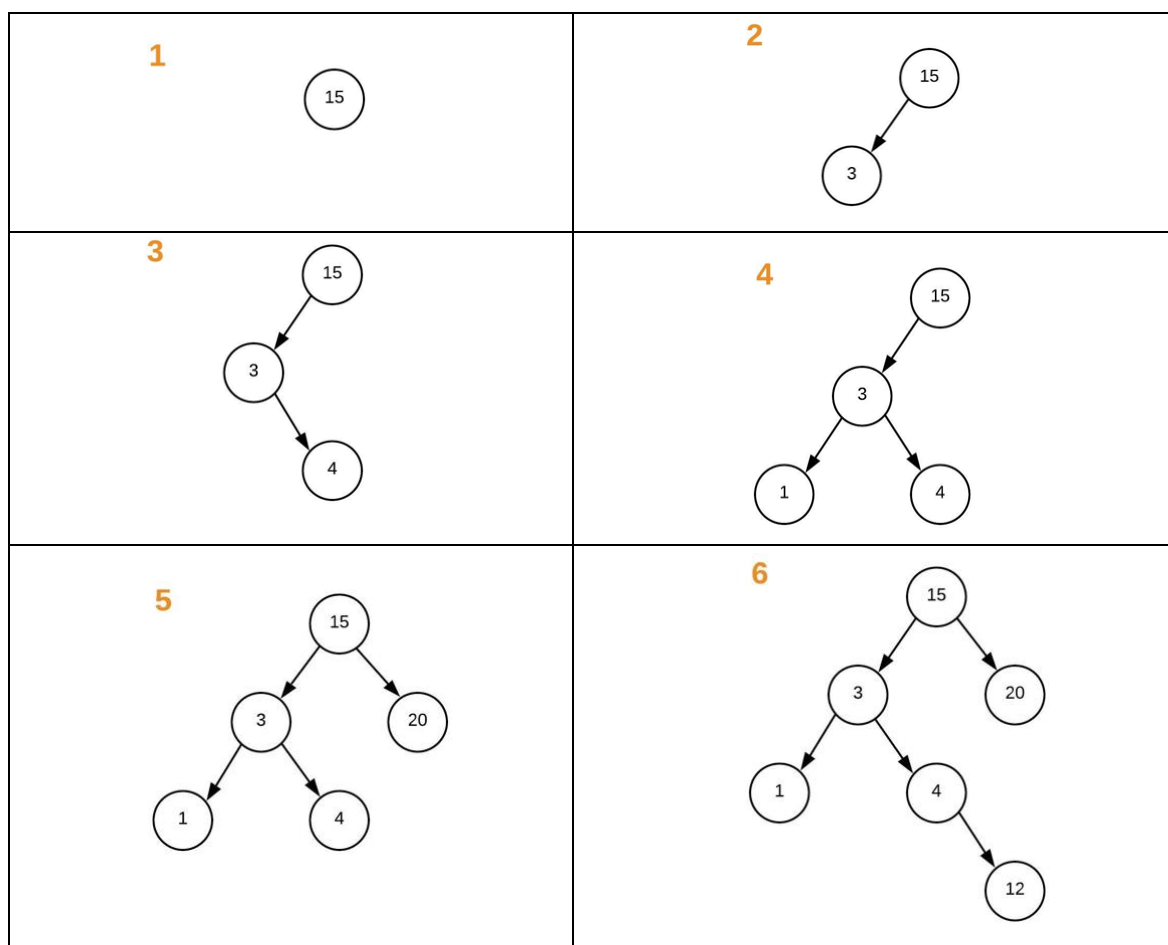
You can visualize BST operations at: <https://www.cs.usfca.edu/~galles/visualization/BST.html>

2.5.2 Insertion into a Binary Search Tree

Let us suppose we want to insert the following six numbers as keys, one at a time in the given order, into an initially empty BST.

15, 3, 4, 1, 20, 12

The following sequence of steps 1-6 shows the BST after insertion of each key.



3 Introduction to Recursion

3.1 Recursion

In this section we study recursion or the process of a **sub-program calling itself** to complete a given task. There are certain problems that can be easily solved using recursive programs. Certain characteristics of the problem or the solution make it more natural for us to think of the solution in a recursive manner. Recursion is a very powerful tool in programming languages. With recursive programmes we try to solve the problem by reducing the problem to a smaller problem at each calling iteration until we reach a state where we do not need to go any further to solve the problem (we get a very simple problem). As you go through this section, you would realize that thinking using recursion will help you come up with solutions to certain types of problems.

An algorithm or a function that calls itself directly or indirectly to solve a smaller version of its task is a recursive algorithm/function. The process of doing that is called recursion. In a recursive algorithm, recursion occurs until a final call which does not require further recursion, which is called the terminating condition(s). When we say the recursive call is to solve a smaller problem, what we mean is the recursive call is closer to the terminating condition(s) compared to the original function call.

Many definitions in computer science and mathematics are expressed recursively. For example the factorial function can be expressed recursively and we can use the factorial function as an example to learn about recursive function calls.

Factorial of n , or $n!$, is defined recursively as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

When we look at the above recursive definition we can observe two important aspects

1. Calculating the factorial of zero (0) is easy and straightforward. The answer is 1 and it is readily available. More importantly, the answer does not require any recursive calls. This is the terminating condition for this recursive function.
2. When n is greater than zero (0), factorial of n is defined using factorial of $(n-1)$. This indicates that to calculate factorial of n , we have to first calculate factorial of $(n-1)$. This needs a recursive call. Also, it is important to note that when we need to calculate factorial of n , the recursive call is to calculate factorial of $(n-1)$, which is closer to the terminating condition of calculating factorial of zero (0).

In general, the above two characteristics are present in any recursive algorithm and they can be stated as.

1. The task has a terminal case that is not recursive.
2. The task is defined in terms of itself by allowing us to work on a smaller task of the same nature.

Using the recursive definition we can calculate the factorial of n as n multiplied by factorial of $(n-1)$. Then to compute the result for factorial of n we must first know the result of factorial of $(n-1)$. As the definition is recursive to calculate the factorial of $(n-1)$, we need to compute $(n-1) \times (n-2)!$. For example factorial of 3 ($3!$) is defined in terms of $3 \times 2!$. The following figure shows the recursive calls and the steps involved in computing factorial of 3 ($3!$).

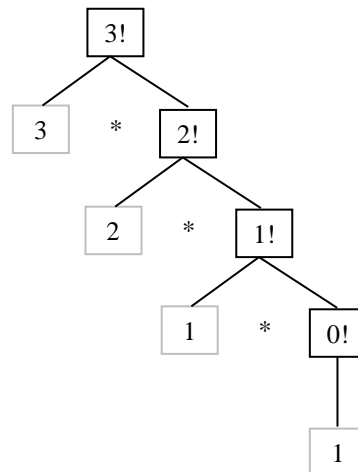


Figure 3.1 Recursion Tree Illustrating the Computing of 3!

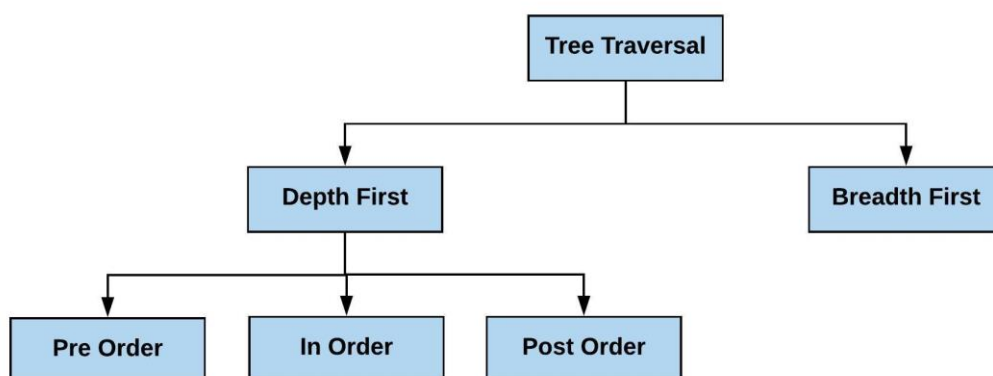
As you can see in the above figure it shows the number of steps that would be taken to reach the terminal case 0!. When we reach the terminal condition we compute the value for the terminal case and then substitute that value to compute 1!. We can continue this process until we are able to compute the answer for our original computation 3!.

You should now realize that in the above structure, the recursive function calls go from top to the bottom, from 3! down to 0!. Then the solution to the problem is calculated (or formulated) from bottom to top, starting from answer to 0! all the way up to the original problem of 3!. This is the nature of the recursive functions work. As you try out the examples in the lab, make sure you pay attention to this behaviour.

The above structure shown in figure 3.1 is called a recursion tree. Recursion tree is a convenient way to visualize what happens during recursion. The recursive calls will occur in a top-down manner in a recursion tree and the answer will be generated in a bottom-up manner in a recursion tree.

3.2 Tree Traversal with Recursion

Linear data structures like Arrays, Stacks, and Queues have only one logical way to traverse them. Contrast to that, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



InOrder (tree) <ul style="list-style-type: none"> • Traverse the left subtree, i.e., call InOrder(left-subtree) • Visit the root. • Traverse the right subtree, i.e., call InOrder(right-subtree)
PreOrder (tree) <ul style="list-style-type: none"> • Visit the root. • Traverse the left subtree, i.e., call PreOrder(left-subtree) • Traverse the right subtree, i.e., call PreOrder(right-subtree)
PostOrder (tree) <ul style="list-style-type: none"> • Traverse the left subtree, i.e., call PostOrder(left-subtree) • Traverse the right subtree, i.e., call PostOrder(right-subtree) • Visit the root.

A sample Python code for recursive tree traversal is given in Appendix 6.5.

3.3 More Examples on Recursion

Example 3.1 Write a Python function to compute the factorial of a given positive integer.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

You should be able to see the resemblance between the recursive definition of factorial and the Python program. In the Python program, **first we have checked whether the problem instance given to it is a problem which represents a terminating condition** (by checking whether the n is zero). **If that is the case, the answer is available, and the function returns without any recursive calls. If it is not the terminating condition, the *factorial* function calls itself with parameter $(n-1)$ and returns the answer of that call multiplied by n .** The structure of the program is very similar (if not identical) to the definition.

We may want to add a condition to check and see if the integer that is given to the function is a positive integer. Otherwise, the function will continue to execute without meeting the terminal condition. In that case the function will continue until we reach a condition called “stack overflow” which indicates that the machine has run out of space to call any more functions. Each time we call a function the calling function is put on a special memory area called the stack. A stack, as we have seen, is a special data structure that can be used to put things and take things only from the top. In other words, the first item that we put into the stack will be the last item to be taken out of the stack. In the case of recursive function calls, all the unresolved function calls are put on the stack until the function reaches the terminal condition and be able to evaluate it. Once it is evaluated, we will go through the stack of function calls and try to remove all the function calls on the stack starting from the top and coming down to the bottom of the stack.

Example 3.2 Compute the n^{th} power of a given value x (i.e., compute x^n).

We can define the requirement as a recursive function.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}$$

This definition can be transformed into a Python code in a straightforward manner and the resultant code is shown below.

```
def myexp(x, n):
    if n == 0:
        return 1
    else:
        return x * myexp(x, n-1)
```

You would have realized that by redefining the function we can reduce the number of recursive calls required for the computation. $x^n = x(x^{n/2})^2$ when n is odd and $x^n = (x^{n/2})^2$ when n is even while assuming $n/2$ will be an integer division where the largest possible integer which is less than or equal to $n/2$ (floor function). There is no change to the terminating condition. The modified definition for the n^{th} power of a given value x is shown below.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot (x^{(n/2)})^2 & \text{if } n > 0 \text{ and } n \text{ is odd} \\ (x^{(n/2)})^2 & \text{if } n < 0 \text{ and } n \text{ is even} \end{cases}$$

With this observation we can reduce the number of recursive calls by half. The following code describes the implementation.

```
def myexp_optimised(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        halfPower = myexp_optimised(x, n/2)
        return halfPower * halfPower
    else:
        halfPower = myexp_optimised(x, n/2)
        return x * halfPower * halfPower
```

Exercise 3.1 Develop a Python function to find the greatest common divisor (gcd) of two positive integers. The following algorithm was developed by Euclid. Use the algorithm to develop a Python program to find the gcd.

```
gcd(x, y)
    if x > y
        d ← gcd(x-y, y)
    else if y > x then
        d ← gcd(x, y-x)
    else d ← x
    return d
```

Exercise 3.2 Develop a recursive Python function to find the minimum in a given list of numbers.

Exercise 3.3 Develop a recursive Python function to print a list of characters in the reverse order.

Exercise 3.4 A principal P of Rs. 3,000 is deposited in a bank that compounds interests at a yearly rate of R=10%. If A_n is the amount present after n years;

$$A_0 = P = 3000$$

$$A_1 = A_0 + A_0R = (1+R) A_0 = (1 + 0.1) 3,000 = 3,300$$

$$A_2 = A_1 + A_1R = (1+R) A_1 = (1 + 0.1) 3,300 = 3,630$$

.....

Develop a recursive Python function with parameters P, R, and n to compute the amount in the bank at the end of n years.

Exercise 3.5 Develop a recursive Python function to find all the even numbers in a list of numbers and print them.

Exercise 3.6 Develop a recursive Python function to find the sum of a given list of numbers and print the answer.

Exercise 3.7 Develop a Python program to flatten a list. Flattening a list means making a list which contains lists within it a simple list which only contains simple data items. You can do this by including the simple data items in the nested lists to the outside list in a recursive manner. Simply stated, you have to take a list which has nested lists and transform it into a single list with all the elements. Number of nesting levels are unlimited. Some sample inputs and expected outputs are shown below.

Table 3.1: Sample inputs and corresponding outputs for a program to flatten a list

Sample Input	Expected Output
[0, 1, 2, 3, 4, 5, 6, 7]	[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, [31, 32], 4, 5, [61, 62], 7]	[0, 1, 2, 31, 32, 4, 5, 61, 62, 7]
[0, 1, 2, [31, 32, [331, 332, 333], 34, 35], 4, 5, 6, 7]	[0, 1, 2, 31, 32, 331, 332, 333, 34, 35, 4, 5, 6, 7]
[0, 1, 2, [31, 32, [331, 332, 333], 34, 35], 4, 5, [61, 62], 7]	[0, 1, 2, 31, 32, 331, 332, 333, 34, 35, 4, 5, 61, 62, 7]

3.4 Recursion vs Iteration

We have seen that we can solve certain problems by using recursive solutions very easily using a very few lines of code. These solutions are easy to develop and also simple, elegant and easy to understand. But they may not be the most efficient way to solve certain problems. As described earlier each recursive call will add an entry to the run time stack with all the required information for the function to continue later. In cases where there are large numbers of recursive calls with a lot of temporary information it could lead to issues with the available amount of memory for the run time stack. It is important to note that all recursive programs can also be solved using an equivalent iterative algorithm where we use a loop control structure such as a while loop to go through the required number of iterations. So it is important to look into the requirements of the problem and the environment where we are executing the program when deciding the type of approach we should take to solve a given problem.

4 Introduction to Algorithms with Sorting

To understand a collection of data sometimes we may want to arrange the data values starting from the smallest to the largest. A sorted list of data items also allows us to use a **binary search** to find an item in the list quickly rather than going through all the items. In a binary search we will go to the middle of the list and compare the value there with the search value. Based on the comparison we will decide to go to the first half of the list or the second half. We will repeat this until we find the value, or we can confirm that the value does not exist in the list. Binary search is one example of the use of a sorted list. We cannot expect data to be given in sorted order all the time. So, in data processing we need to sort data most of the time and it takes a lot of compute time to sort large lists of data items. It is important to implement efficient sorting algorithms. There are different types of sorting algorithms with different levels of efficiency. In this section we will look into two sorting algorithms as examples and the use of sorting to solve different types of problems.

4.1 Selection Sort

The idea behind the selection sort is to find the correct place for each of the data items starting from the smallest item. In the first step we find the smallest item in the list and exchange that data item with the first item. With the initial step we have identified and placed the first item in the list. Now we do the same thing for the rest of the list until we have gone through the entire list. This may not look like the most efficient way to sort the data, but we will look at it as a very simple sorting algorithm to understand and implement.

Example 4.1 Write a selection sort algorithm that can sort a given list of integers.

To solve the sorting problem, we can use 3 different concepts that we used earlier when we studied the fundamentals of Python programming. They are the ability to find the smallest value in a list, swapping two values and iterating through a list of items. The following code segments show the basic outline for a selection sort implementation.

```
def getMinIndex(L, i, n):    # will return the min value
                             # in array L, within index i to n

    minI = i
    for k in range(i+1, n):
        if(L[minI] > L[k]):
            minI = k
    return minI

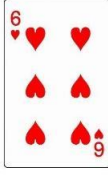
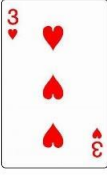
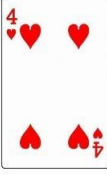

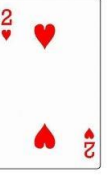

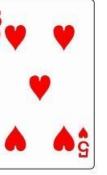
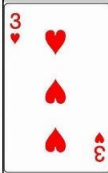
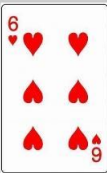


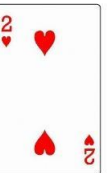


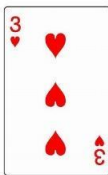

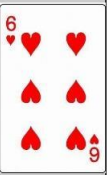
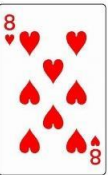

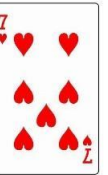

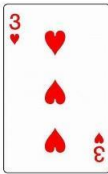
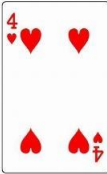
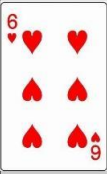
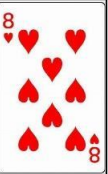
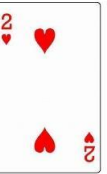





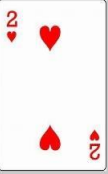



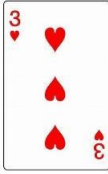
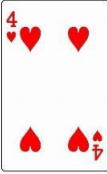
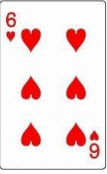




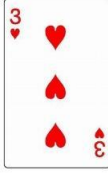

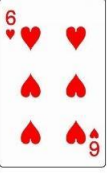
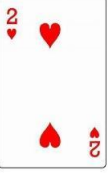



def selectionSort(L):

    n = len(L)
    for k in range (0, n-1):    # Loop for n-1 iterations
        minK = getMinIndex(L, k, n)    # Get index of min
        (L[k], L[minK]) = (L[minK], L[k])    # Swap items
```

4.2 Bubble Sort

The idea behind the Bubble Sort is simple and is similar to the selection sort. In Bubble Sort, we repeatedly step through the list and compare each pair of adjacent items in the list. In each comparison if the two items are in the wrong order, we swap the two items to correct the error. We repeat this process until the array is sorted. Bubble sort gets its name from the fact that this pairwise comparison and swapping process will bubble up the largest items in the list to the end of the list in each iteration.

Graphical representation of the first bubble sort iteration is given below.

Initial Set							
Iteration 1							
Step 1							
Step 2							
Step 3							
Step 4							
Step 5							
Step 6							

Python code for bubble sort is given below.

```
def bubbleSort(L):  
    n = len(L)  
    not_sorted = True  
    while not_sorted:  
        not_sorted = False  
        for k in range (0, n-1):  
            if(L[k] > L[k+1]):  
                (L[k], L[k+1]) = (L[k+1], L[k])  
                not_sorted = True
```

Exercise 4.7 Develop a program to input a student index number as an integer and the GPA as a float value and then sort the list based on the GPA. At the end the list of students should be listed with the student with the largest GPA first in descending order.

Exercise 4.8 Develop a recursive program to do a quick sort for a given list of integers.

Exercise 4.9 Develop a program that can be used to enter student information such as name, index number, current GPA, date of registration (year, month, day), and National Identity Card Number. Once the student details are entered the program should be able to sort the details according to the index number and display all the details for each student.

Exercise 4.10 Develop a program that can be used to enter details of books such as the title, author, ISBN and the year of publication. Once the book details are entered the program should sort the data on the ISBN and write the data to a file.

Exercise 4.11 Develop a program to read the data file generated by the previous exercise and allow the user to add new records to the file. The data should always be written in the sorted order of index numbers. The program should also allow the user to display the list of books in the collection and display an individual book based on the user input of index number if it exists in the collection.

5 Introduction to Software Engineering

5.1 Introduction

We now live in a world that depends on complex computer-based systems. For example, most of the banking activities are computerised and provide services without relying on any paper-based accounting systems. We see the use of computerised systems in many other fields. Most of the electrical or electronic devices also have computer programs that control them. So we are in a world that needs to build and maintain software in a cost-effective manner.

The notion of *Software Engineering* was first proposed in 1968 as a response to the “software crisis”. The software crisis resulted from the introduction of computer hardware based on integrated circuits. This made it possible for users to claim the power of computing by using computer programs. Computer software was initially developed in an ad-hoc manner without any thought about the process or techniques. The initial objective was to get the programs to work. But with the rapid increase in the complexity of programs and the demand for new types of software, most of the software products that were produced were of poor quality, requiring a lot of maintenance that would cost a lot. Also, most of the software projects were not completed on time and to the original budget. To address these issues the practitioners came out with engineering techniques and practices that are part of software engineering. Over the years with the experience of building larger complex software systems the field of software engineering has evolved into a mature discipline of engineering. This is a very brief introduction to software engineering.

Software engineering is the study and application of engineering (i.e., systematic, disciplined, quantifiable) approach to the design, development, operation, and maintenance of software. A key aim of the engineering approach is to build the *right software, defect free, within budget and on time*. Here, the *right software* means *software that meets customer expectations*.

5.2 Software

A software product refers to all the related things that are required to make a software system operate properly. This includes a number of programs, configuration files that are used to setup and install the system, system documentation that explains the structure of the system, user documentation that explains how the system should be used, effective user support system and bug-fixes and updates. A software engineer is involved in building software products that can be used by customers.

5.3 Software Engineer

Software engineering is an engineering discipline that is concerned with all aspects of software production from the initial stage of writing the requirements to maintaining it while being used. An engineer is expected to make things work by applying methods, theories and tools appropriately. Engineers may be required to discover solutions to problems where there are no applicable theories and methods available. Engineers are also expected to work within budget constraints when devising solutions. The process of software engineering therefore does not only include writing code; it also includes other important activities such as requirement study, project management, planning, design, testing, documentation and maintenance.

In fulfilling these tasks, software engineers have to collaborate with many others and maintain effective means of communication throughout a project life cycle. Unlike in some other engineering disciplines, it is common to have tens or even hundreds of software engineers (perhaps distributed across the world and working in different time zones) to collaborate and work as a team in a software project. For this

reason, other than the technical skills, effective teamwork and communication skills are also much desired in a software engineer.

5.4 Software Process

A software process is the set of activities that are associated with the development of a software product. There are different software process models that are used depending on the type of product that is developed. There are four fundamental activities that are common to all types of software development processes.

- **Software specification** where customers and engineers define the software to be produced and the constraints that would be there when we use it.
- **Software development** where the software is designed and developed.
- **Software validation** where the software is checked to ensure that it is what the customer requires.
- **Software evolution** where the software is modified to adapt it to changing customer and market requirements.

We need to select an appropriate software process depending on the type of project we want to achieve. Most of the software processes are based on the following general paradigms (also called *software process models*).

- **Waterfall** development where we do the specification, development, validation, and evolution as clear separate phases, sequentially. We complete one phase and move to the next phase based on the outcome of the previous phase.
- **Iterative and incremental** development is where we try to incrementally develop the system by completing smaller parts of the system at each iteration. At each iteration we do a small amount of requirements specification, design and development for the specification and some validation. We continue to iterate through until we complete the project.
- **Component based** approach is where we try to find existing components that can be used to combine and complete a system. The development process concentrates on integrating these parts rather than having to develop them from scratch.

When we want to develop a real time system to control an unmanned air vehicle (UAV) we may have all the requirements specifications and we may use a waterfall approach to develop the system. In case of building an e-commerce application to sell customized t-shirts online we may want to go for an iterative approach to develop.

5.5 Quality of Software

It is important for us to be able to measure the quality of software so that we may be able to compare different software products that can do similar things. Unlike in most other engineering disciplines the product produced in software development is not tangible. In order to measure the quality, the attributes that we could consider range from actual functionalities to behavioural characteristics while it executes to the quality and organization of the code. The following is a list of different aspects that we can check to measure the quality of software.

Maintainability: Software changes are inevitable. So, a good software product should be written in such a way so that making changes that are required due to new user requirements is not difficult.

Dependability: Dependable Software will not cause physical or economic damage in the event of failures or other unusual activities. Dependable software is secure and reliable.

Efficiency: In developing good quality software we should make every effort to avoid wasting system resources such as memory, processor time and disk space when we execute them.

Usability: It should be possible for the intended users of the software to use the software without undue effort. To achieve this, we should have properly designed user interfaces with adequate documentation.

Robustness: The extent to which software can continue to operate correctly despite invalid inputs.

5.6 Challenges in Software Engineering

As stated before, unlike in most other engineering disciplines, the products produced in software development are not tangible. This fundamental characteristic gives rise to several challenges and complexities.

Fred Brooks, a pioneering figure in the field, writing about the process of software development stated that: "...*There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity...*" (in *No Silver Bullet: Essence and Accidents of Software Engineering*, Computer, 1987). Brooks makes a distinction between *accidental complexity* and *essential complexity* in software engineering. According to him, the main difficulties in software engineering are twofold:

- Essence: inherent in the nature of software and difficult to overcome.
- Accidents: not inherent in software and may be overcome.

The inherently difficult aspects of software, as described by Brooks, are the following (Brooks, 1987):

- **Complexity:** Software are more complex for their size than perhaps any other human construct because no two parts are alike. If they are, we make the two similar parts into a subroutine. In this respect, software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound. A scaling-up of a software is not merely a repetition of the same elements in larger sizes, it is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly. Many of the classic problems of developing software derive from this essential complexity and its nonlinear increases with size. From the complexity comes the difficulty of communication among team members, which leads to product flaws, cost overruns and delays.
- **Conformity:** Computer scientists/software engineers are not alone in facing complexity. Physics deals with terribly complex objects even at the fundamental particle level. The physicist labors on, however, in a firm faith that there are unifying principles to be found. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity, forced by the many human institutions and systems to which his interfaces must conform. These differ from interface to interface, and from time to time, not because of necessity but only because they were designed by different people.
- **Changeability:** Software is constantly subject to pressures for change. Of course, so are buildings, cars, and computers. But manufactured things are infrequently changed after manufacture; they are

superseded by later models, or essential changes are incorporated into later-serial-number copies of the same basic design. Call-backs of automobiles are really quite infrequent; field changes of computers somewhat less so. Both are much less frequent than modifications to software. This is so because (i) the software of a system embodies its function, and the function is the part that most feels the pressures of change, (ii) software can be changed more easily--it is pure thought-based, infinitely malleable. Buildings do in fact get changed, but the high costs of change is understood by all. As a software product is found to be useful, people try it in new use cases at the edge of or beyond the original domain. The pressures for extended function come chiefly from users who like the basic function and invent new uses for it. Software is embedded in a cultural matrix of applications, users, laws, and machines. These all change continually, and their changes inexorably force change upon the software product.

- **Invisibility:** Software is invisible and not-visualizable. Geometric abstractions are powerful tools. The floor plan of a building helps both architect and client evaluate spaces, traffic flows, views. Contradictions and omissions become obvious. Scale drawings of mechanical parts and stick-figure models of molecules, although abstractions, serve the same purpose. A geometric reality is captured in a geometric abstraction. The reality of software is not inherently embedded in space. Hence, it has no ready geometric representation in the way that land has maps, silicon chips have diagrams, and computers have connectivity schematics. As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general graphs superimposed one upon another. The several graphs may represent the flow of control, the flow of data, patterns of dependency, time sequence, name-space relationships. These graphs are usually not even planar, much less hierarchical.

In addition to the above, due to the complexity of software, *testing of software* itself is a challenge. There is no general way to guarantee a software product is defect free (i.e., we cannot fully rule out defects). A well-known saying attributed to *Dijkstra*, another key figure in the field, is: "*Testing can show presence of bugs, not their absence*". Defects in software can be in specifications, design, or coding.

As another perspective to understand the challenges in software engineering, one can consider the notion of *wicked problems* which are considered impossible to solve because of incomplete, contradictory, and changing requirements that are often difficult to recognize. Software engineering is frequently used to solve wicked problems. A wicked problem is *a problem that could be clearly defined only by solving it, or by solving part of it*. In some software projects, it is impossible to see all the issues until actually building the software or part of it. We cannot define the whole solution in the beginning, as the world changes and customers, developers and users constantly learn. This means too much early planning can be bad; thus a process like the waterfall model will not work and iterative / incremental approaches could be the only way.

Building software is not writing or hacking code to get something working. It is an engineering discipline that requires a proper process.

6 Appendix

6.1 Declaring and Use of Arrays in C – Sample Code

An array can be declared as follows in the C language.

```
#include <stdio.h>

int main()
{

    int n[10]; /* n is an array of 10 integers */
    int i, j;

    /* initialize elements of array n to 0 */
    for (i = 0; i < 10; i++)
    {
        n[i] = i + 100; /* element at i set to i + 100 */
    }

    /* output each array element's value */
    for (j = 0; j < 10; j++)
    {
        printf("Element[%d] = %d\n", j, n[j]);
    }

    return 0;
}
```

Sometimes, the size of the array we declared may be insufficient. To solve this issue, we can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

There are 4 library functions defined under `<stdlib.h>` that enable dynamic memory allocation in C programming. They are `malloc()`, `calloc()`, `realloc(z)` and `free()`.

The `malloc()` function reserves a block of memory of the specified number of bytes. And, it returns a pointer of type `void` which can be casted into a pointer of any form. Since the dynamically allocated memory created with either `calloc()` or `malloc()` doesn't get freed on its own, we must explicitly use `free()` to release the space. If the dynamically allocated memory is insufficient or more than required, we can change the size of previously allocated memory using `realloc()` function.

Consider we need to create an array with the size given as a user input (not pre-fixed value, hence dynamic) and then change the allocated size of the array to another value given as another user input. After all the executions are over, we have to free the allocated memory manually in this case.

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

int main()
```

```

{
    int *ptr, i, n1, n2;
    printf("Enter size of array: ");
    scanf("%d", &n1);

    /* allocate memory for n1 number of integers */
    ptr = (int *)malloc(n1 * sizeof(int));
    printf("Addresses of previously allocated memory: ");

    for (i = 0; i < n1; ++i)
        printf("%u\n", ptr + i);

    printf("\nEnter new size of array: ");
    scanf("%d", &n2);

    /* reallocate memory for n1 number of integers */
    ptr = realloc(ptr, n2 * sizeof(int));
    printf("Addresses of newly allocated memory: ");

    for (i = 0; i < n2; ++i)
        printf("%u\n", ptr + i);

    /* release memory used by the array */
    free(ptr);
    return 0;
}

```

6.2 Stack as a Fixed Size Array in C – Sample Code

```

#include <stdio.h>
#define SIZE 10

int stack[SIZE];
int top = -1;

void push(int value)
{
    if (top < SIZE - 1)
    {
        if (top < 0)
        {
            stack[0] = value;
            top = 0;
        }
        else
        {
            stack[top + 1] = value;
            top++;
        }
    }
    else
    {
        printf("Stackoverflow!!!!\n");
    }
}

```

```

int pop()
{
    if (top >= 0)
    {
        int n = stack[top];
        top--;
        return n;
    }
}

int top()
{
    return stack[top];
}

int isempty()
{
    return top < 0;
}

void display()
{
    int i;
    for (i = 0; i <= top; i++)
    {
        printf("%d\n", stack[i]);
    }
}

int main()
{
    push(4);
    push(8);
    display();
    pop();
    display();
    return 0;
}

```

6.3 Stack with Pointers and Structs in C – Sample Code

```

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

struct Stack
{
    int top;
    unsigned capacity;
    int *array;
};

```

```

struct Stack *createStack(unsigned capacity)
{
    struct Stack *stack = (struct Stack *)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int *)malloc(stack->capacity * sizeof(int));
    return stack;
}

int isFull(struct Stack *stack)
{
    return stack->top == stack->capacity - 1;
}

int isEmpty(struct Stack *stack)
{
    return stack->top == -1;
}

void push(struct Stack *stack, int item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
    printf("%d pushed to stack\n", item);
}

int pop(struct Stack *stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

int main()
{
    struct Stack *stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    printf("%d popped from stack\n", pop(stack));

    return 0;
}

```

6.4 Queue with Dynamic Arrays in C – Sample Code

```

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

struct Queue
{
    int front, rear, size;
    unsigned capacity;

```



```

    int *array;
};

struct Queue *createQueue(unsigned capacity)
{
    struct Queue *queue = (struct Queue *)malloc(sizeof(struct Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1; // This is important, see the enqueue
    queue->array = (int *)malloc(queue->capacity * sizeof(int));
    return queue;
}

int isFull(struct Queue *queue)
{
    return (queue->size == queue->capacity);
}

int isEmpty(struct Queue *queue)
{
    return (queue->size == 0);
}

void enqueue(struct Queue *queue, int item)
{
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
    printf("%d enqueued to queue\n", item);
}

int dequeue(struct Queue *queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1) % queue->capacity;
    queue->size = queue->size - 1;
    return item;
}

int front(struct Queue *queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->front];
}

int rear(struct Queue *queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->rear];
}

```

```

int main()
{
    struct Queue *queue = createQueue(1000);
    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);
    enqueue(queue, 40);
    printf("%d dequeued from queue\n\n", dequeue(queue));
    printf("Front item is %d\n", front(queue));
    printf("Rear item is %d\n", rear(queue));
    return 0;
}

```

6.5 Recursive Tree Traversal in Python – Sample Code

```

class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

def printInorder(root):
    if root:
        printInorder(root.left)
        print(root.val),
        printInorder(root.right)

def printPostorder(root):
    if root:
        printPostorder(root.left)
        printPostorder(root.right)
        print(root.val),

def printPreorder(root):
    if root:
        print(root.val),
        printPreorder(root.left)
        printPreorder(root.right)

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print("Preorder traversal of binary tree is")
printPreorder(root)
print("\nInorder traversal of binary tree is")
printInorder(root)
print("\nPostorder traversal of binary tree is")
printPostorder(root)

```