CC4 Data Structures and Algorithms Midterms Notes

I. INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS

Basic Terminologies

A. Algorithm
1. Finite structure of well-defined instructions
2. Used to solve a problem
3. Typically, independent of the programming language
4. Can be expressed in:
    a. Formal language
    b. Flowcharting
    c. Pseudocode
    d. Programming language

B. Data Structure
1. Process of organizing data in a computer for more efficient use
2. Looks into:
    a. Collection of data values
    b. Relationships amongst data values
    c. Functions and operations applied to the data
3. Expressed as an algorithm
    a. All data structures are algorithms, but not all algorithms are data structures

C. Programming Language
1. Set of commands used to create a software program
2. Used to properly illustrate the concepts in an algorithm and data structure

Parts of a Programming Language

A. Data types and objects (int, float, Boolean, String)
B. Expressions (assignment, printing)
C. Operations (arithmetic, conditional, logical)
D. Decision control structures (if, else if, else, switch)
E. Iterative control structures (while, for, do-while)
F. Arrays (one-dimensional, multi-dimensional)
G. Methods (user-defined, parameters, return type)
H. Other parts:
1. Input
2. Classes and objects

Concepts in Algorithms and Data Structures

A. Data
1. A data must have the following characteristics:
2. **Atomic** – Define a single concept
3. **Traceable** – Be able to be mapped to some data element
4. **Accurate** – Should be unambiguous
5. **Clear and Concise** – Should be understandable

B. Data Type
1. Classifies various types of data which help:
    a. Determine the values that can be used
    b. Type of operations that can be performed

C. Basic Operations
1. Data in data structures are processed by operations
2. Largely depends on the frequency of the operation that needs to be performed
3. Examples:
    a. Search
    b. Insertion
    c. Deletion
    d. Sorting
    e. Merging

What is Data Type?

A. Attribute of data that tells the compiler/interpreter how the data is intended to be used
B. Looks into what kind of data can be placed inside of the variable
C. Types of data types:
1. Built-in
2. Derived

3. Data object – represents an object having a data (i.e. String)

Built-in Data Types

A. Programming language has built-in support
B. Examples:
   1. Integers
   2. Boolean (true, false)
   3. Floating (Decimal numbers)
   4. Character and Strings

Derived Data Type

A. Implementation independent
B. Normally built by the combination of primary or built-in data types and associated operations on them
C. Examples (based on the one-dimensional array):
   1. List
   2. Array
   3. Stack
   4. Queue

What is a Data Structure:

A. Collection of data type values
B. Process of organizing data in a computer for more efficient use
C. Looks into:
   1. Collection of data values
   2. Relationships amongst data values
   3. Functions and operations applied to the data
   4. Expressed as an algorithm

Forms of Data Structures

A. Linear
   1. Structure where the elements are stored sequentially
   2. Elements are connected to the previous and the next element
   3. As the elements are stored sequentially, so it can be traversed or accessed in a single run
   4. Examples:
      a. Array
      b. List

B. Tree
   1. Represent a hierarchical tree structure
   2. Contains a root value and subtrees if children (with a parent node)
   3. Represented as a set of linked nodes

C. Hash Table
   1. Data structure capable of mapping keys to values
   2. **Key** – labels the pair; used to pertain to the pair
   3. **Value** – data stored as the pair to the key
   4. Typically abstracted and enhanced with additional behaviors
   5. Example:
      a. Dictionary (Python)

D. Graphs
   1. Abstract data type that follows the principles of graph theory
   2. Structure is non-linear
   3. Consists of:
      a. **Nodes/Vertices** – points on the graph
      b. **Edges** – lines connecting each node

Array Address Calculations

A. One-Dimensional Array
   1. Also known as a list or just array
   2. A set of elements stored sequentially
   3. Uses the index as a pointer

B. Two-Dimensional Array
   1. Also known as a matrix
   2. Stored sequentially in two dimensions

3. Data is stored "row and column wise"

C. Three-Dimensional Array

1. Think as a collection of 2D arrays

Address Calculations

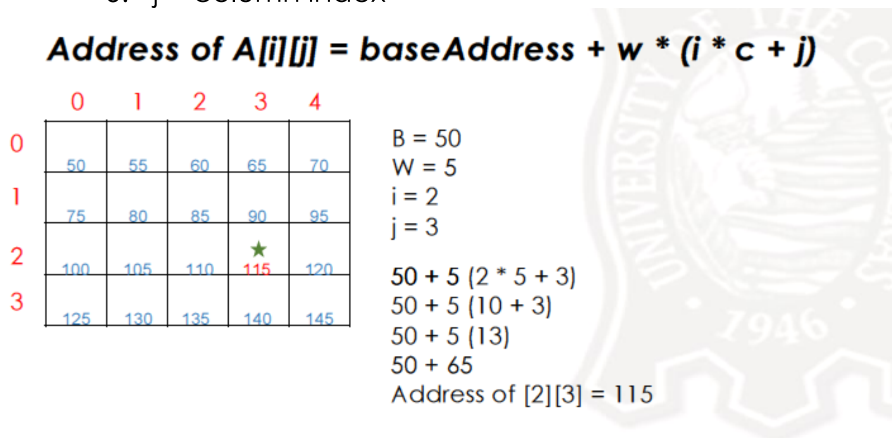
A. Two Ways:
   1. Row major system
   2. Column major system

Row Major System

A. All elements of the same rows are stored consecutively

B. Formula:
   1. Address of A[i] [j] = baseAddress + w*(i*c+j)

C. Which means:
   2. baseAddress = assigned address to A[0] [0]
   3. w = storage size of one element stored in the array
   4. I = row index
   5. c = number of columns
   6. j = column index

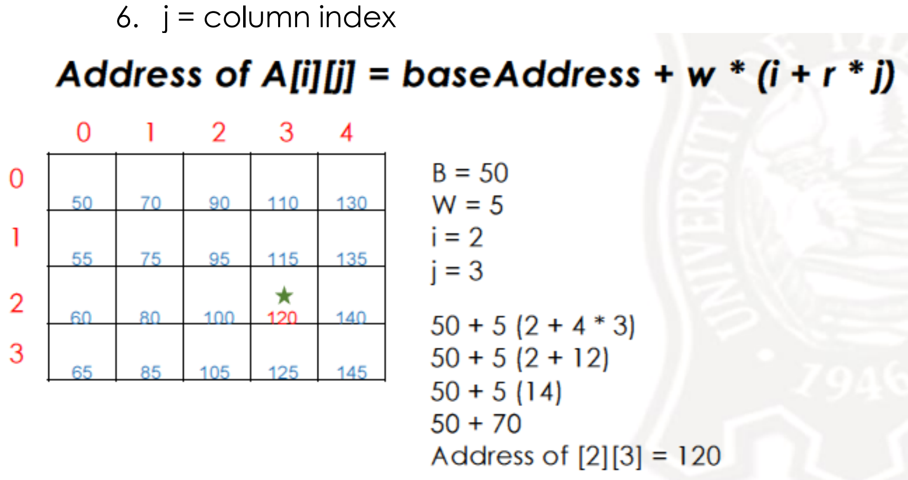


Column Major System

A. All elements of the same column are stored consecutively

B. Formula:
   1. Address of A[i] [j]=baseAddress + w*(i+r*j)

C. Where:
   2. baseAddress = assigned address to A[0] [0]
   3. w = storage size of one element stored in the array
   4. I = row index
   5. r = number of rows
   6. j = column index



II. STACKS AND QUEUES

Importance of Stacks and Queues

A. Both stacks and queues are an example of data structures

B. Most common way to arrange data in different algorithms of the same data type
   1. Expressions (infix, prefix, postfix)
   2. Binary search trees
   3. AVL trees

4. Graphs
5. Searching algorithm
6. Sorting algorithms

Applying Stacks and Queues

A. Usually implemented in one-dimensional array
    1. Can use other variations: list, linked-list, etc.
    2. Possible to use for any programming language
B. Can also be used in multidimensional arrays
    3. Not recommended
    4. Increases the time complexity and the space required

Stack	Queue
• Last-in first-out policy	• First-in first-out policy
• First element is at the bottom	• First element is at the beginning/front
• Last element is at the topmost part	• Last element is at the end of the queue
• Examples: pancakes, pringles, stack of books, etc.	• Examples: lines at the supermarket or jeepney

Stacks

A. Container based on the last-in-first-out (LIFO) policy
    1. New data is inserted at the last index (push)
    2. Data to be deleted starts off with the last index (pop)
B. Uses only one (1) pointer
    1. Starts off at array[-1]: empty
C. Maximum number of elements is the limit of the array
D. Practical examples:
    1. Pringles
    2. Pancake stack
    3. Stack of clothing

Inserting Elements in Stacks (Push)

A. Create a one-dimensional array
    1. Pointer is at -1
B. Place value to be inserted in another value
C. Locate the pointer
D. Iterate the value of the pointer
E. Place value to the array where is the index is pointer (array[pointer[)

Deleting Elements in Stacks (Pop)

A. Locate the pointer
B. Assign the value of the pointer to the index of the array
C. Remove the value at array[pointer]
D. Decrement the value of the pointer by 1

Queues

A. Container based on the first-in-first-out (FIFO) policy
    1. New data is inserted at the last index
    2. Data to be deleted starts off with the first index
B. Uses two (2) pointers
    3. One pointer is for the first element in the array
    4. Another pointer is for the space after the last element in the array
C. Maximum number of elements is array.length-1
    5. Queue is considered empty if both pointers are on the same position
    6. Arrays can be considered circular
D. Practical examples:
    7. People queuing on a line

Inserting Elements in Queues (Enqueue)

A. Create a one-dimensional array
B. Place the value to be inserted into a variable
C. Locate the value of the second pointer (position of the index number after the last element that was placed)
D. Place value to the array where is the index is the second pointer (array[pointer_two])

E. Iterate the second pointer

Deleting Elements in Queue (Dequeue)

A. Locate where the first pointer is (pointer that shows the earliest element inserted)
B. Remove the value at array [pointer_one]
C. Iterate first pointer

III. INTRODUCTION TO EXPRESSIONS

What are Expressions?

A. A combination of symbols used to express portions of mathematical equations
B. Symbols can be:
    1. Numbers (positive/negative, absolute/decimal)
    2. Variables (expressed in letters)
    3. Operators (arithmetic or comparison)
    4. Punctuations (grouping)
C. Mathematical expressions can be:
    1. Arithmetic (numbers only)
    2. Algebraic (numbers and constant)
    3. Comparison (uses comparison symbols)

Parts of an Expression

A. Operator
    1. Symbols(s) that decide(s) which operation is to be performed
    2. Arithmetic (+, -, *, /, %)
    3. Comparison (<, >, ==, <=, >=)

B. Operand
    4. Symbol(s) that represents an entity on which the operation is performed
    1. Numbers (0, -1, 1.2)
    2. Variables (a, b, x)
    3. Constants (log, e)

Expressions in this Unit

A. Limited to binary expressions
    1. Utilizes operations that would require two operands
    2. Arithmetic and comparison operations may be used
    3. Increment and decrement operations are not to be used
    4. Unary/ternary operators are possible for use in expressions
B. Limited to arithmetic operations
    1. Comparison operations may be used, and may appear in activities
    2. All examples will be limited to arithmetic operations and the equal sign (=)

Infix Expressions

What is an Infix Expression?

A. Usual way on how human express mathematical expressions
B. Notation where the operators are placed in-between the operands
C. Requires specific precedence and associativity rules

Precedence

A. Determines the order of what operation must be performed first in comparison to other operations
B. Parenthesis, Multiplication and Division, Addition and Subtraction

Associativity

A. Determines the order of what operation must be performed if they are in the same precedence
B. Left to right precedence

Prefix and Postfix Expressions

A. Infix expressions are easily understood for humans, but not for machines
    1. There are a lot of rules and regulations that cannot be easily translated into machine code
B. Prefix and Postfix expressions are notation that are understood better by machines
    1. All rules on precedence and associativity are removed
    2. All symbols can be placed on a stack
    3. Increases overall efficiency of the code
    4. Less understood by humans

Precedence Rule

A. **In-Stack Priority**
    1. The priority if the operator as an element of the stack

B. **In-Coming Priority**
    1. The priority of the operator as a current token

Postfix Expressions

What is a Postfix Expression?

A. Notation for expressions where:
    1. Operators are placed on the right side (after)
    2. Operands are placed on the left side (before)

B. Order of evaluation of expressions is always left to right

C. Bracket cannot be used to change the order

D. More commonly used for machine code
    1. Translation from infix expression is much easier

Steps – Infix to Postfix

A. Determine the order of operations using the riles from infix expressions
B. First operands that must be evaluated are placed at the leftmost portion of the notation
C. First operator that must be evaluated are placed after the operands
D. Next operators and/or operands are placed at the left side of the notation
E. Last operator must be at the rightmost side of the notation

Prefix Expressions

What is a Prefix Expression?

A. Notation where the operators are written before the operands

B. Operators act on the two nearest values on the right
    1. Technically evaluated from left to right
    2. The order changes depending whether or not elements to the right would be used

Steps – Infix to Prefix

A. Determine the order of operations using the rules from infix expressions
B. First operands that must evaluated are placed at the rightmost portion of the notation
C. First operator that must be evaluated are placed before the operands
D. Next operators and/or operands are placed at the right side of the notation
E. Last operator must be at the leftmost side of the notation

IV. INTRODUCTION TO ALGORITHMS

Priori and Posteriori Estimates

A. Originally used for statistics and partial differential equations

B. In computer science, they are used to analyze algorithms and test their effectiveness
    1. Priori estimate/priori analysis – analysis of algorithms
    2. Posteriori estimate/ posteriori testing – testing of program codes

Piori Analysis

A. Focuses on analyzing an algorithm
    1. Determine the efficiency of the algorithm
    2. Determine the time and space
    3. Done before it is coded into a program

B. Independent of language
    1. Usually expressed in pseudocode

C. Independent on hardware
    1. Worst case scenario is usually measured

Posteriori Testing

A. Done on a program
    1. Usually focuses on testing the efficiency of the program
    2. Also looks into watch time and bytes
    3. There are other factors that can change the efficiency

B. Dependent on the:
    1. Language
    2. Hardware (usually space)
    3. Operating system

Asymptotic Notation
What is Asymptotic Notation?

A. Language that analyzes an algorithm's running time
B. Done by identifying its behavior as the input size for the algorithm increases
C. Usually used to measure time, but can also measure space

Type of Asymptotic Notations

A. **Big-O notation (O)**
    1. Identifies the worst-case scenario complexity of an algorithm
B. **Omega notation (Ω)**
    1. Identifies the best-case scenario complexity of an algorithm
C. **Theta notation (Θ)**
    1. Identifies the average-case scenario complexity of an algorithm

Frequency Count and Big-O Notation
What is the Frequency Count Method?

A. Method to determine the time (usually) and space of an algorithm
B. Can be known by ***assigning one unit of time/space for each statement***
C. If any statement is repeating, the frequency is calculated and the time taken is computed

Example – Frequency Count Method

```
                                   Frequency Count
K = 500;                               1
for (j=1; j<=K; j++)                   1 + k + 1 + k
    x= x+1;                            k
n=200;                                 1

Substituting actual value for k.
```

Freq. count. = 1505
= O(n)

Parts:
Initialization – j=1
Condition – j<=K
Iteration – j++
Body/Statement – x= x+1;

Statement = 1
K = 500; - 1
j=1 – 1
j<=K – n-1+1
j++ - n-1+1+1
x= x+1; - n-1+1+1
n=200 – 1

Add Everything
1+1+n+n+1+n+1+1
3n+5
3(500)+5
1505

Big O Notation
What is the Big O Notation?

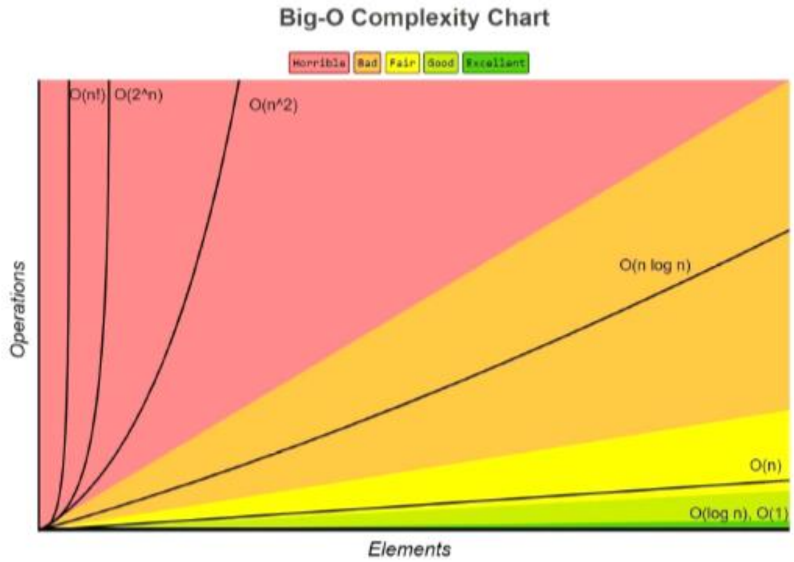
A. Type of asymptotic notation
B. Used to determine the efficiency and **complexity** of an algorithm
    1. Average
    2. Best
    3. Worst case – usually used
C. Looks into the complexity in terms of the input size
D. Used for priori analysis
E. Frequency count method must be done first to properly identify the complexity

General Rules to Determine:

A. Ignore constants
B. Certain terms dominate others:
    O(1) < O(log n) < O(n) < O(n log n) < O(n2) < O(2n) < O(n!)

C. Ignore low-order terms when the high-order terms are present

Big O Complexity Chart



A. Looks into the time that is taken to finish a set of operations
B. Usually states the efficiency of the algorithm
   1. "Horrible" notation can be faster depending on the number of operations
   2. "Good" notations are usually better on shorter operations, but worse at larger ones

Constant Time

A. Independent of input size
B. It doesn't matter what the operation is, it will have the same complexity
C. Notation(s): O(1)
D. Sample operations:
   1. Assignment to a variable
   2. Printing

Linear Time

A. The more operations, the slower the time
B. Better for shorter operations
C. Notation(s): O(log N), O(N), O(N log N)
D. Sample operations:
   1. Decision control structures
   2. One-layered iterative control structures

Quadratic Time

A. The more operations, the shorter the time
B. Better for longer operations
C. Notation(s): O(N2), O(2N), O(N!)
D. Sample operations:
   1. Nested iterative control structures
   2. Recursive loops
   3. Permutations

Big O Measure of Efficiency

- Measure of efficiency for n = 10,000

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	O(logn)	14	Microsecond
Linear	O(n)	10,000	Seconds
Linear Logarithmic	O(nlogn)	140,000	Seconds
Quadratic	$O(n^2)$	$10{,}000^2$	Minutes
Polynomial	$O(n^k)$	$10{,}000^k$	Hours
Exponential	$O(c^n)$	$2^{10{,}000}$	Intractable
Factorial	O(n!)	10,000!	Intractable

Big O Notation

A. O(1)
   1. Constant; most instructions are executed once or at most only a few times
B. O(log n)
   1. Program slightly slower as N grows; normally in programs that solve a big problem by transforming it into a small problem, cutting the size by some constant factor
C. O(n)
   1. Linear; proportional to the size of N
D. O(n log n)
   1. Occurs in algorithms that solve a problem by breaking it up into smaller sub-problems, solve them independently and then combining the solution
E. O(n^2)
   1. Quadratic; can be seen in algorithms that process all pairs of data items
F. O(n^k)
   1. Polynomial; algorithms that process polynomial of data items
G. O(2^n)
   1. Exponential; brute-force situation
H. O(n!)
   1. Factorial

Example: given 2 algorithms performing the same task on N inputs, which is faster and efficient?

P1 | 10n | O(n)   P2 | $n^2/2$ | O(n^2)

N	P1	P2
1	10	0.5
5	50	12.5
10	100	50
15	150	112.5
20	200	200
30	300	450

Example – Big O Notation

```
public static int returnSum (int a[], n){
    int s = 0;
    for (int i = 0; i < n; i++){
        s = s + a[i];
    }
    return s;
}
```

Part

s = 0; - 1
int i = 0; -1
I < n – n-0
I++ - n-0+1
s = s+a[i]; - n-0+1
return s; - 1

Add everything

1+1+n+n+1+n+1+1
3n = 5
O(n)

Example – Big O Notation

```
for(j=1; j<=n; j++){
    for(k=1; k<=n; k++){
        c[j][k] = 0;
        for(l=1; l<=n; l++){
            c[j][k] = c[j][k] * b[l][k];
        }
    }
}
```

Inner part
1=1 – 1
1<=n – n-1+1
1++ - n-1+1+1
c[j][k] = c[j][k] * b[l][k]; - n-1+1+1

Add everything
1+n+n+1+n+1
3n+3

Middle part
k=1 – 1
k<=n – n-1+1
k++ - n-1+1+1
c[j][k] = 0; - n-1+1+1

Add everything
1+n+n+1+n+1
3n+3

Multiply inner part and middle part
(3n+3)(3n+3)
$9n^2$+9n+9n+9
$9n^2$+18n+9

Outer part
j=1 – 1
j<=n – n-1+1
j++ - n-1+1+1
statement – n-1+1+1

Add everything
1+n+n+1+n+1
3n+3

Multiply the calculated parts to Outer part
($9n^2$+18n+9)(3n+3)
$27n^3$+$27n^2$+$54n^2$+54n+27n+27
$27n^3$+$81n^2$+81n+27
O(n^3)

Big O Notation Basing

Output		Big O Notation
$3\underline{n^2}+8$	=	O(n^2)
$5\underline{n^3}+3n+2$	=	O(n^3)
$9\underline{n}+3$	=	O(n)

Introduction to Expressions

Unit 3

CC4 Data Structures and Algorithms

Christine T. Gonzales

Table of Contents

Infix to Prefix Conversion

Infix–Prefix Example

Infix: <Operand 1> **<Operator>** <Operand 2>
Prefix: **<Operator>** <Operand 1> <Operand 2>

8 / 4 * 2 - 5 + 6 + (7 - 6 + 5 ^ 2 % (9 - 6 + 1))

/84
*/842
-*/8425
+*/84256

-76

^52

-96
+-961
%^52+-961
+-76%^52+-961

+*/84256+-76%^52+-961

Infix–Prefix Example

Infix: <Operand 1> **<Operator>** <Operand 2>
Prefix: **<Operator>** <Operand 1> <Operand 2>

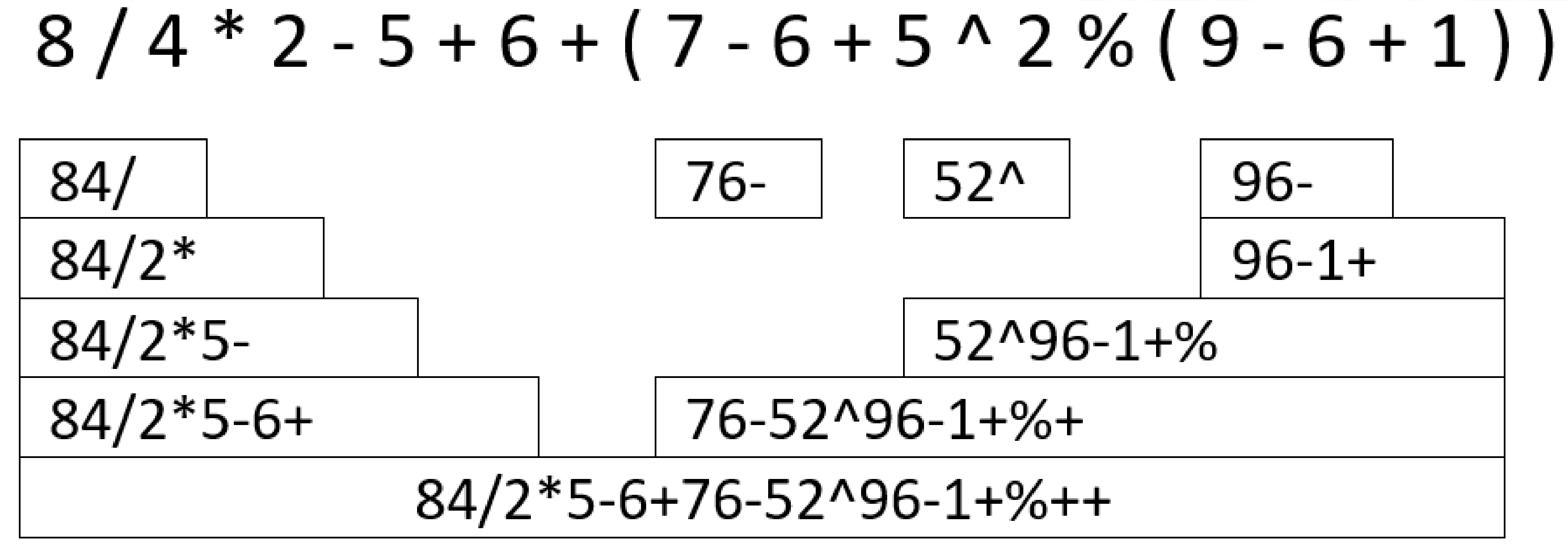
A * B - C + (D ^ E) % F / G

	^DE
*AB	%DEF
-*ABC	/%DEFG
+-*ABC/%DEFG	

Infix to Postfix Conversion

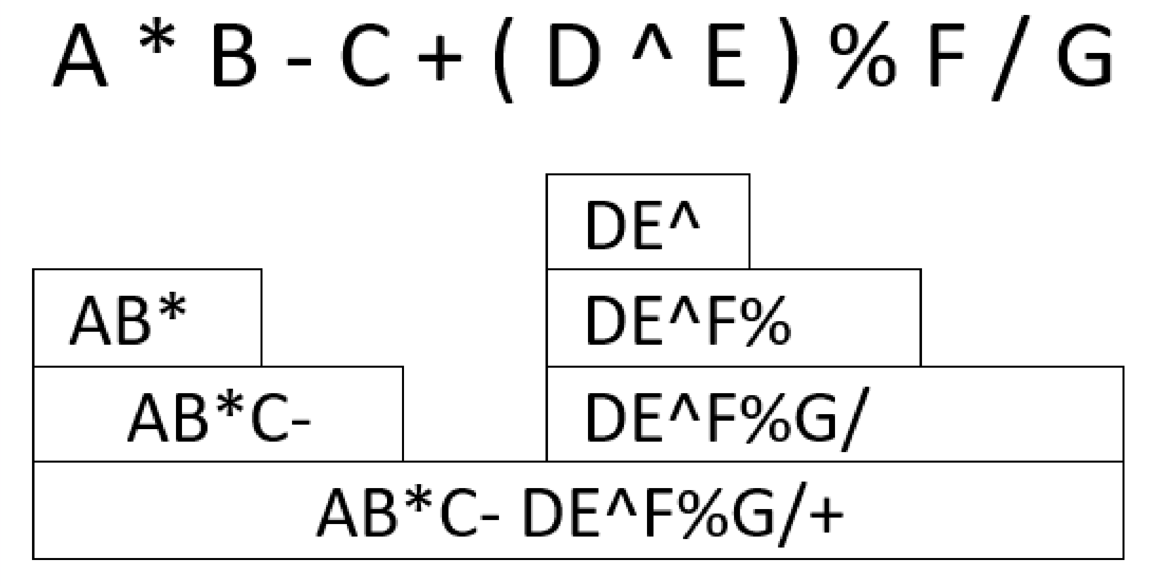
Infix–Postfix Example

Infix: <Operand 1> **<Operator>** <Operand 2>
Postfix: <Operand 1> <Operand 2> **<Operator>**



Infix–Postfix Example

Infix: <Operand 1> **<Operator>** <Operand 2>
Postfix: <Operand 1> <Operand 2> **<Operator>**



Infix to Postfix Using Stacks

Precedence Rule | Conversion

Precedence Rule

IN-Stack Priority

- The priority if the operator as an element of the stack

IN-Coming Priority

- The priority of the operator as current token

SYMBOL	ISP	ICP
)	--	--
^	3	4
*, /, %	2	2
+, -	1	1
(	0	4

Precedence Rule

Rule:

- Operators are taken out of the stack (POP) as long as the ISP is greater than or equal to the ICP of the new operator

Conversion

Example: 8/4*2-5+6+(7-6+5^2%(9-6+1))

Token	Stack	Output
8	#	8
/	#/	8
4	#/	84
*	#*	84/
2	#*	84/2
-	#-	84/2*
5	#-	84/2*5

Conversion

+	#+	84/2*5-
6	#+	84/2*5-6
+	#+	84/2*5-6+
(	#+(	84/2*5-6+
7	#+(	84/2*5-6+7
-	#+(-	84/2*5-6+7
6	#+(-	84/2*5-6+76
+	#+(+	84/2*5-6+76-

Conversion

5	#+(+	84/2*5-6+76-5
^	#+(+^	84/2*5-6+76-5
2	#+(+^	84/2*5-6+76-52
%	#+(+%	84/2*5-6+76-52^
(	#+(+%(	84/2*5-6+76-52^
9	#+(+%(	84/2*5-6+76-52^9
-	#+(+%(-	84/2*5-6+76-52^9
6	#+(+%(-	84/2*5-6+76-52^96

Conversion

+	#+(+%(+	84/2*5-6+76-52^96-
1	#+(+%(+	84/2*5-6+76-52^96-1
)	#+(+%	84/2*5-6+76-52^96-1+