Introduction to Data Structures and Algorithms

As part of Units 1 and 2

CC4 Data Structures and Algorithms

Christine Gonzales

Table of Contents

- Concepts on algorithms and data structures
- Data types and structures
- Data structures
- Array address calculations

Concepts on Algorithms and Data Structures

Basic terminologies | Parts of a programming language | Concepts in algorithms and data structures

Basic Terminologies

- Algorithm
- Data structure
- Programming language

Basic Terminologies – Algorithm

- Finite structure of well-defined instructions
- Used to solve a problem
- Typically independent of the programming language
- Can be expressed in:
  - Formal language (English, Filipino, etc.)
  - Flowcharting
  - Pseudocode
  - Programming language

Basic Terminologies – Data Structure

- Process of organizing data in a computer for more efficient use
- Looks into:
  - Collection of data values
  - Relationships amongst data values,
  - Functions and operations applied to the data
- Expressed as an algorithm
  - All data structures are algorithms, but not all algorithms are data structures

Basic Terminologies – Programming Language

- Set of commands used to create a software program
- Used to properly illustrate the concepts in an algorithm and data structure

In this course, ***Java*** *shall be used as the programming language to express the algorithms .and data structures.*

Parts of a Programming Language

- Data types and objects (int, float, boolean, String)
- Expressions (assignment, printing)
- Operations (arithmetic, conditional, logical)
- Decision control structures (if, else if, else, switch)
- Iterative control structures (while, for, do-while)
- Arrays (one-dimensional, multi-dimensional)
- Methods (user-defined, parameters, return type)

Parts of a Programming Language

- Other parts:
  - Input
  - Classes and objects

Concepts in Algorithms and Data Structures

- Data
- Data type
- Basic operations

Concepts in Algorithms and Data Structures

Data

A data must have the following characteristics:

- Atomic – Define a single concept
- Traceable – Be able to be mapped to some data element
- Accurate – Should be unambiguous
- Clear and Concise – Should be understandable

Concepts in Algorithms and Data Structures

Data Type

- Classifies various types of data which help:
  - Determine the values that can be used
  - Type of operations that can be performed

Concepts in Algorithms and Data Structures

Basic Operations

- Data in data structures are processed by operations
- Largely depends on the frequency of the operation that needs to be performed
- Examples:
  - Search
  - Insertion
  - Deletion
  - Sorting
  - Merging

Data Types and Structures

What is a Data Type? | What is a Data Structure? | Forms of Data Structures

What is a Data Type?

- Attribute of data that tells the compiler / interpreter how the data is intended to be used
- Looks into what kind of data can be placed inside of the variable
- Types of data types:
  - Built-in
  - Derived
  - Data object – represents an object having a data (i.e. String)

What is a Data Type?

Built-in Data Type

- Programming language has built-in support
- Examples:
  - Integers
  - Boolean (true, false)
  - Floating (Decimal numbers)
  - Character and Strings

What is a Data Type?

Derived Data Type

- Implementation independent
- Normally built by the combination of primary or built-in data types and associated operations on them
- Examples (based on the one-dimensional array):
  - List
  - Array
  - Stack
  - Queue

What is a Data Structure?

- Collection of data type values
- Process of organizing data in a computer for more efficient use
- Looks into:
  - Collection of data values
  - Relationships amongst data values,
  - Functions and operations applied to the data
  - Expressed as an algorithm

Forms of Data Structures

- Linear
- Tree
- Hash
- Graphs

Forms of Data Structures

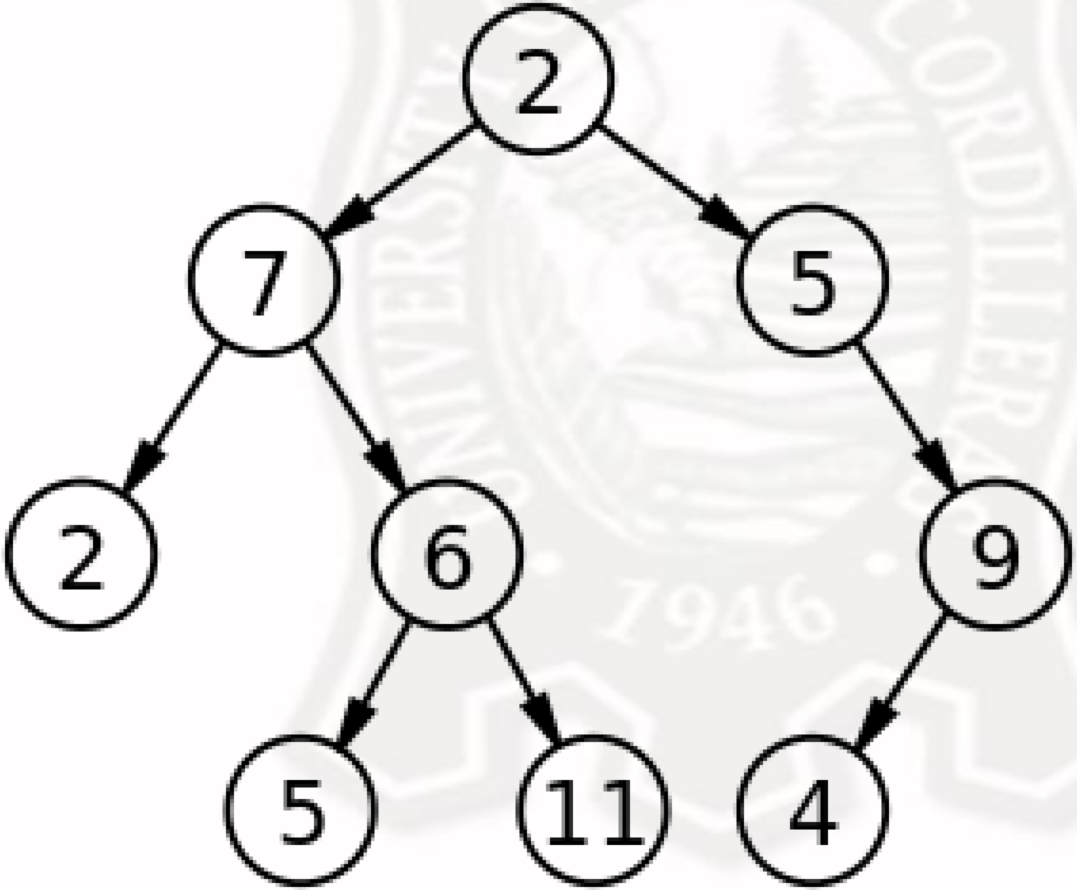
Linear

- Structure where the elements are stored sequentially
- Elements are connected to the previous and the next element
- As the elements are stored sequentially, so it can be traversed or accessed in a single run
- Examples:
  - Array
  - List

Forms of Data Structures

Tree

- Represent a hierarchical tree structure
- Contains a root value and subtrees of children (with a parent node)
- Represented as a set of linked nodes



Forms of Data Structures

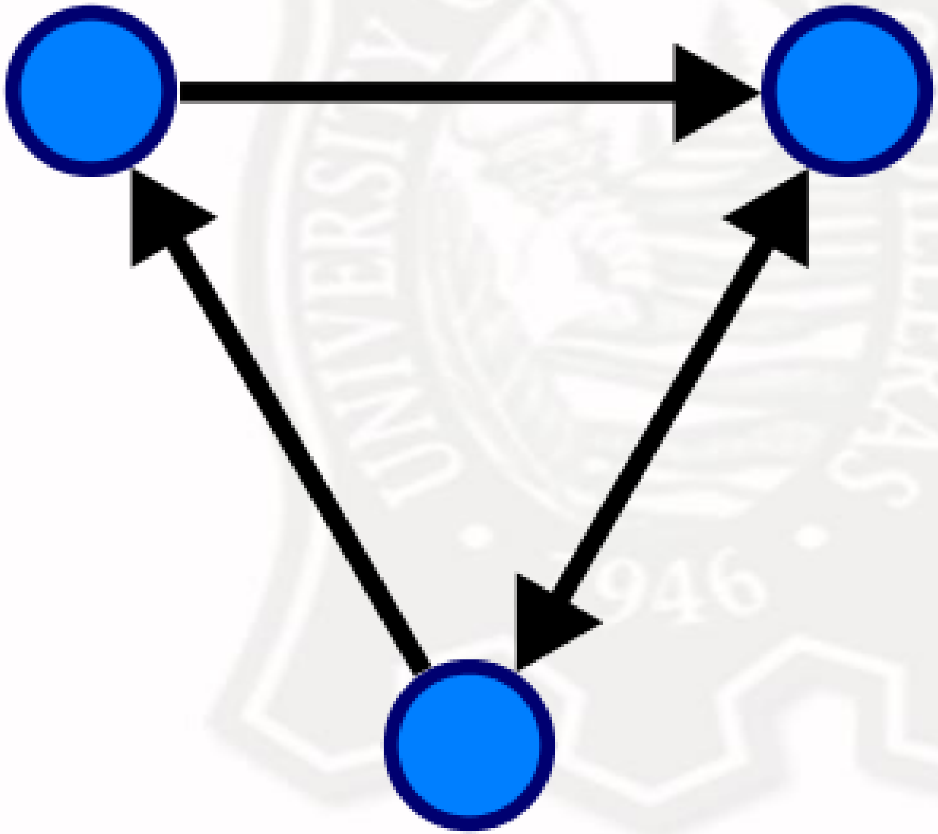
Hash Table

- Data structure capable of mapping keys to values
  - Key – labels the pair; used to pertain to the pair
  - Value – data stored as the pair to the key
- Typically abstracted and enhanced with additional behaviors
- Example:
  - Dictionary (Python)

Forms of Data Structures

Graph

- Abstract data type that follows the principles of graph theory
- Structure is non-linear
- Consists of:
  - Nodes / Vertices – points on the graph
  - Edges – lines connecting each node

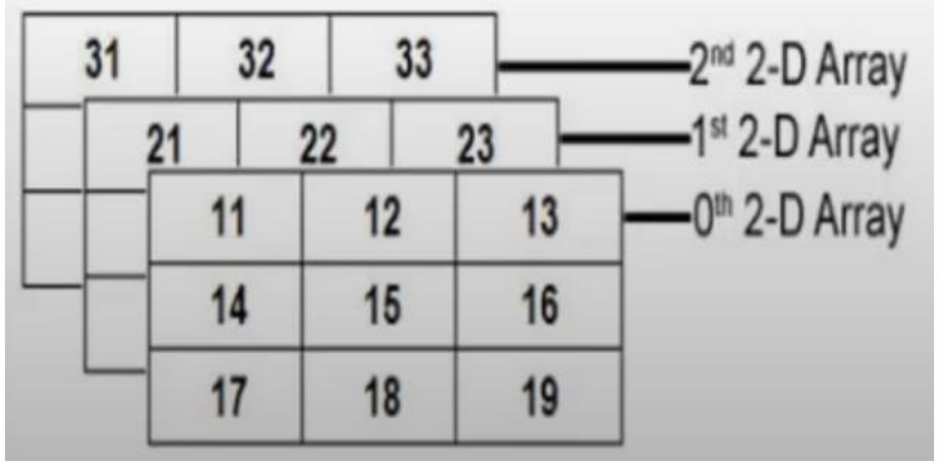


Array Address Calculations

Address calculations | Row major system | Column major system

Review: Multidimensional Array

- **One-Dimensional Array**
  - Also known as a list or just array
  - A set of elements stored sequentially
  - Uses the index as a pointer
- **Two-Dimensional Array**
  - Also known as a matrix
  - Stored sequentially in two dimensions
  - Data is stored "row and column wise"
- **Three-Dimensional Array**
  - Think as a collection of 2D arrays



Address Calculation

Two Ways:

- Row major system
  - All elements of the row are stored consecutively
- Column major system
  - All elements of the column are stored consecutively

	0	1	2
0	5	6	2
1	7	0	-3
2	4	1	3

Row	5	6	2	7	0	-3	4	1	3
Col	5	7	4	6	0	1	2	-3	3

Row Major System

- All elements of the same rows are stored consecutively
- Formula:

  Address of A[i][j] = baseAddress + w * (i * c + j)

- which means:
  - baseAddress = assigned address to A[0][0]
  - w = storage size of one element stored in the array
  - i = row index
  - c = number of columns
  - j = column index

Row Major System - Example

Address of A[i][j] = baseAddress + w * (i * c + j)

	0	1	2	3	4
0	50	55	60	65	70
1	75	80	85	90	95
2	100	105	110	★ 115	120
3	125	130	135	140	145

B = 50
W = 5
i = 2
j = 3

50 + 5 (2 * 5 + 3)
50 + 5 (10 + 3)
50 + 5 (13)
50 + 65
Address of [2][3] = 115

Column Major System

- All elements of the same columns are stored consecutively
- Formula:

$$\textit{Address of } A[i][j] = baseAddress + w * (i + r * j)$$

- where:
  - baseAddress = assigned address of A[0][0]
  - w = storage size of one element stored in the array
  - i = row index
  - r = number of rows
  - j = column index

Column Major System – Example

Address of A[i][j] = baseAddress + w * (i + r * j)

	0	1	2	3	4
0	50	70	90	110	130
1	55	75	95	115	135
2	60	80	100	★ 120	140
3	65	85	105	125	145

B = 50
W = 5
i = 2
j = 3

50 + 5 (2 + 4 * 3)
50 + 5 (2 + 12)
50 + 5 (14)
50 + 70
Address of [2][3] = 120

Stacks and Queues

Unit 3

CC4 Data Structures and Algorithms

Lovey Jenn A. Reformado

Table of Contents

Introduction to Stacks and Queues

Importance of Stacks and Queues | Applying Stacks and Queues | Stacks v. Queues

Importance of Stacks and Queues

- Both stacks and queues are an example of data structures
- Most common way to arrange data in different algorithms of the same data type
  - Expressions (infix, prefix, postfix)
  - Binary search trees
  - AVL trees
  - Graphs
  - Searching algorithms
  - Sorting algorithms

Applying Stacks and Queues

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

Stacks and Queues

Stack

- Last-in first-out policy
- First element is at the bottom
- Last element is at the topmost part
- Examples: pancakes, Pringles can, stack of books, etc.

Queue

- First-in first-out policy
- First element is at the beginning / front
- Last element is at the end of the queue
- Examples: lines at the supermarket or jeepney

Stacks

Introduction to Stacks | Inserting Elements in Stacks | Deleting Elements in Stacks | Example

Stacks

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

Inserting Elements in Stacks (Push)

- Create a one-dimensional array
  - Pointer is at -1
- Place value to be inserted in another variable
- Locate the pointer
- Iterate the value of the pointer
- Place value to the array where is the index is pointer (array[pointer])

Deleting Elements in Stacks (Pop)

- Locate the pointer
- Assign the value of the pointer to the index of the array
- Remove the value at array[pointer]
- Decrement the value of the pointer by 1

Stacks – Example

0	1	2	3	4	5	6	7

Values: 8, -1, 5, 7, 2, 6, 3, 4, 9, 10

Pointer:

Values – push:

Values – pop:

Queues

Introduction to Queues | Inserting Elements in Queues | Deleting Elements in Queues | Queues as a Circular Structure | Examples

Queues

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

Inserting Elements in Queues (Enqueue)

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

Deleting Elements in Queues (Dequeue)

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

Queues as a Circular Structure

- In theory, the elements of the queue does not change positions
  - Index numbers don't change if the first element is deleted
- Queue is a circular tape structure
  - Elements may be placed at earlier indices provided that they are empty, and the last index of the array was occupied
  - Done to maximize the array
  - Pointers would go back to [0] after array.length-1

Queues - Example

0	1	2	3	4	5	6	7

Values: 8, -1, 5, 7, 2, 6, 3, 4, 9, 10

Pointer1:

Pointer2:

Values – insert:

Values – delete:

Introduction to Expressions

As part of Week 3, Unit 3

CC4 Data Structures and Algorithms

Lovely Jenn A. Reformado

Table of Contents

Introduction to Expressions

What are expressions? | Parts of an expression | Expressions in this unit

What are Expressions?

- A combination of symbols used to express portions of mathematical equations
- Symbols can be:
  - Numbers (positive / negative, absolute / decimal)
  - Variables (expressed in letters)
  - Operators (arithmetic or comparison)
  - Punctuations (groupings)
- Mathematical expressions can be:
  - Arithmetic (numbers only)
  - Algebraic (numbers and constants)
  - Comparison (uses comparison symbols)

Parts of an Expression

Operator

- Symbol(s) that decide(s) which operation is to be performed
- Arithmetic (+, -, *, /, %)
- Comparison (<, >, ==, <=, >=)

Operand

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

Expressions in this Unit

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

Infix Expressions

What is an infix expression? | Examples of infix expressions | Precedence and associativity | Importance of prefix and postfix expressions

What is an Infix Expression?

- Usual way on how humans express mathematical expressions
- Notation where the operators are placed in-between the operands
- Requires specific precedence and associativity rules

Examples of Infix Expressions

- a + b
- c + d * 2
- 3 – e + 1
- (x + y) * (z – 5)

Precedence and Associativity

Precedence

- Determines the order of what operation must be performed first in comparison to other operations
- Parenthesis, Multiplication and Division, Addition and Subtraction
- Example: c + d * 2, (x + y) * (z – 5)

Precedence and Associativity

Associativity

- Determines the order of what operation must be performed if they are in the same precedence
- Left to right precedence
- Example: 3 – e + 1

Prefix and Postfix Expressions

- Infix expressions are easily understood for humans, but not for machines
  - There are a lot of rules and regulations that cannot be easily translated into machine code
- Prefix and postfix expressions are notations that are understood better by machines
  - All rules on precedence and associativity are removed
  - All symbols can be placed on a stack
  - Increases overall efficiency of the code
  - Less understood by humans

Postfix Expressions

What is a postfix expression? | Steps – infix to postfix | Postfix expression – examples

What is a Postfix Expression?

- Notation for expressions where:
  - Operators are placed on the right side (after)
  - Operands are placed on the left side (before)
- Order of evaluation of expressions is always left to right
- Brackets cannot be used to change the order
- More commonly used for machine code
  - Translation from infix expression is much easier

Steps – Infix to Postfix

- Determine the order of operations using the rules from infix expressions
- First operands that must be evaluated are placed at the leftmost portion of the notation
- First operator that must be evaluated are placed after the operands
- Next operators and/or operands are placed at the left side of the notation
- Last operator must be at the rightmost side of the notation

Postfix Expression - Examples

- a + b = **a b +**
- c + d * 2 = **c d 2 * +**
- 3 – e + 1 = **3 e – 1 +**
- (x + y) * (z – 5) = **x y + z 5 -***

Prefix Expressions

What is a prefix expression? | Steps – infix to prefix| Prefix expression – examples

What is a Prefix Expression?

- Notation where the operators are written before the operands
- Operators act on the two nearest values on the right
  - Technically evaluated from left to right
  - The order changes depending whether or not elements to the right would be used

Steps – Infix to Prefix

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

Prefix Expression - Examples

- a + b = **+ a b**
- c + d * 2 = **+ c * d 2**
- 3 – e + 1 = **- 3 + e 1**
- (x + y) * (z – 5) = ***+ x y – z 5**

Introduction to Expressions

Unit 3

CC4 Data Structures and Algorithms

Christine T. Gonzales

Table of Contents

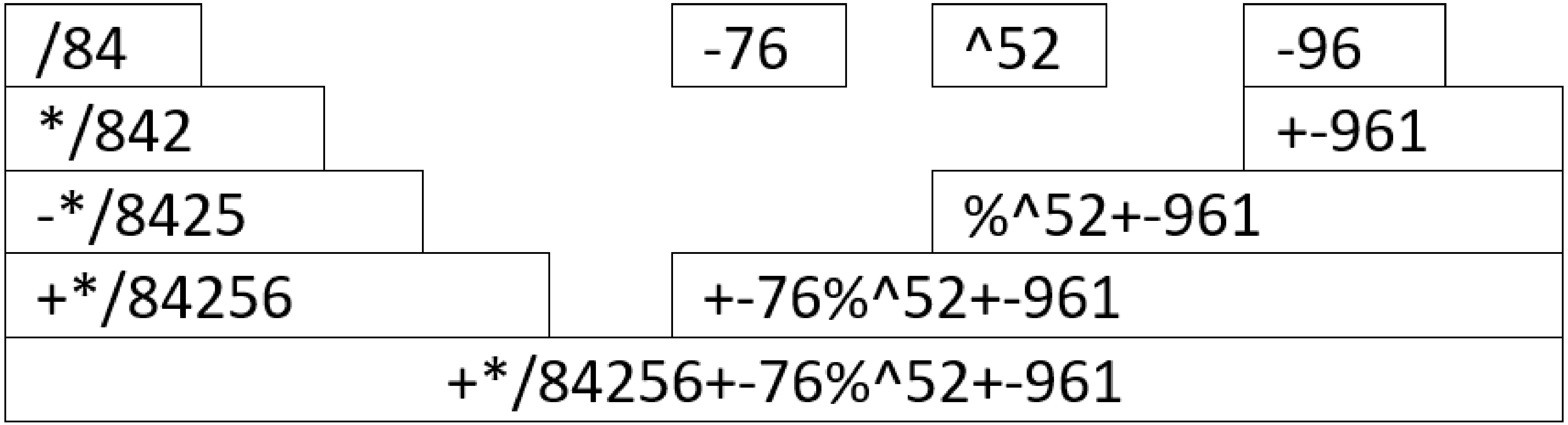
- Infix to Prefix Conversion
- Infix to Postfix Conversion
- Infix to Postfix Using Stacks

Infix to Prefix Conversion

Infix–Prefix Example

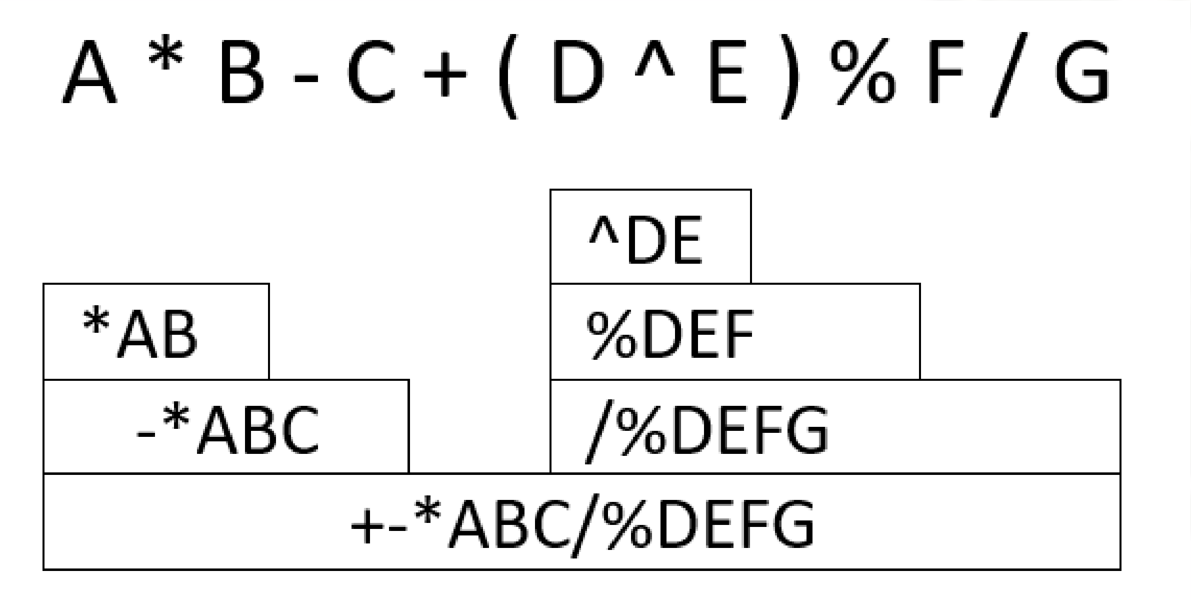
Infix: <Operand 1> **<Operator>** <Operand 2>
Prefix: **<Operator>** <Operand 1> <Operand 2>

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



Infix–Prefix Example

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

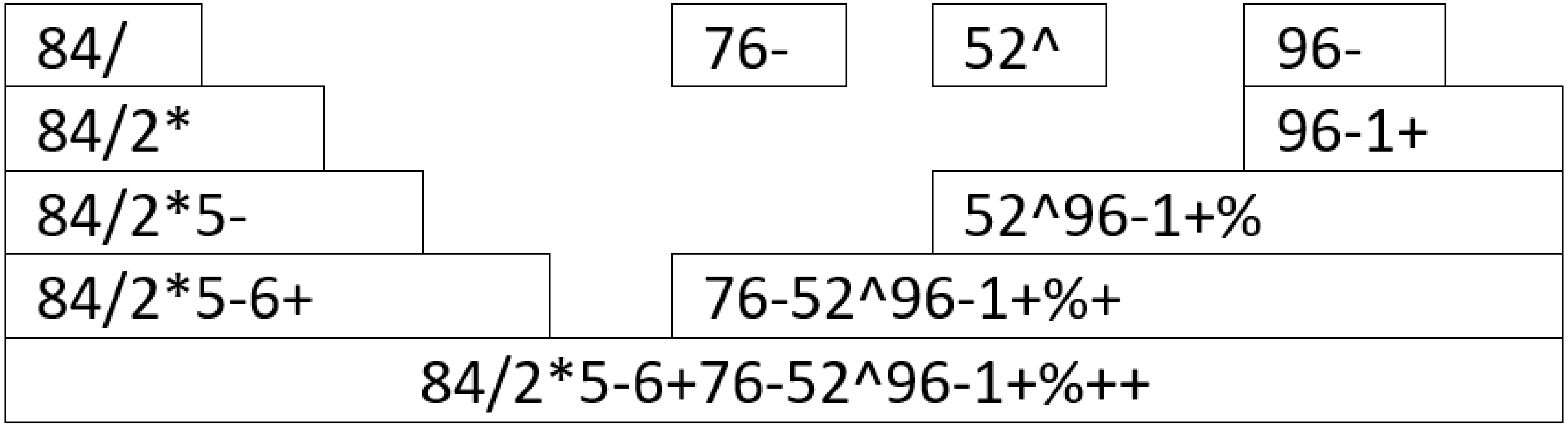


Infix to Postfix Conversion

Infix–Postfix Example

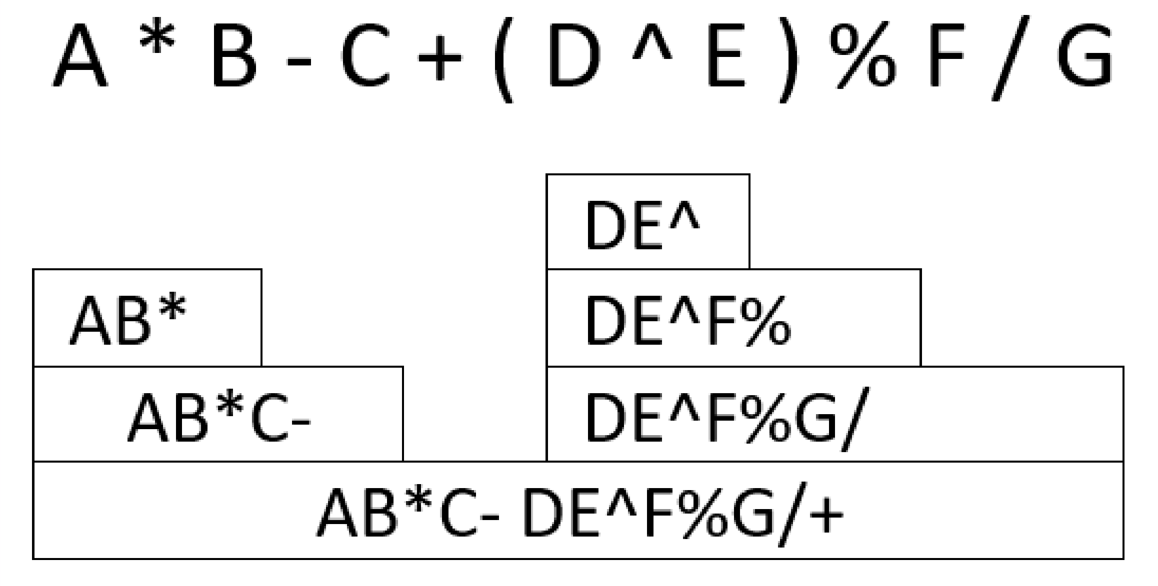
Infix: <Operand 1> **<Operator>** <Operand 2>
Postfix: <Operand 1> <Operand 2> **<Operator>**

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



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+

Introduction to Algorithms

As part of Unit 1

CC4 Data Structures and Algorithms

Christine T. Gonzales

Table of Contents

Priori and Posteriori Estimates

Priori analysis | Posteriori testing

Priori and Posteriori Estimates

- Originally used for statistics and partial differential equations
- In computer science, they are used to analyze algorithms and test their effectiveness
  - Priori estimate / priori analysis – analysis of algorithms
  - Posteriori estimate / posteriori testing – testing of program codes

Priori Analysis

- Focuses on analyzing an algorithm
  - Determine the efficiency of the algorithm
  - Determine the time and space
  - Done before it is coded into a program
- Independent of language
  - Usually expressed in pseudocode
- Independent on hardware
  - Worst case scenario is usually measured

Posteriori Testing

- Done on a program
  - Usually focuses on testing the efficiency of the program
  - Also looks into watch time and bytes
  - There are other factors that can change the efficiency
- Dependent on the:
  - Language
  - Hardware (usually space)
  - Operating system

Asymptotic Notation

What is asymptotic notation? | Types of asymptotic notations

What is Asymptotic Notation?

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

Types of Asymptotic Notations

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

Frequency Count and Big-O Notation

What is the Frequency Count method? | Examples of frequency count method | What is the Big-O Notation? | Complexity chart | Examples of Big-O Notation

What is the Frequency Count Method?

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

Example – Frequency Count Method

Code	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. = 1504
             = O(1)
```

What is the Big O Notation?

- Type of asymptotic notation
- Used to determine the efficiency and **complexity** of an algorithm:
  - Average
  - Best
  - Worst case – usually used
- Looks into the complexity in terms of the input size
- Used for priori analysis
- Frequency count method must be done first to properly identify the complexity

Big O Notation

General Rules to Determine:

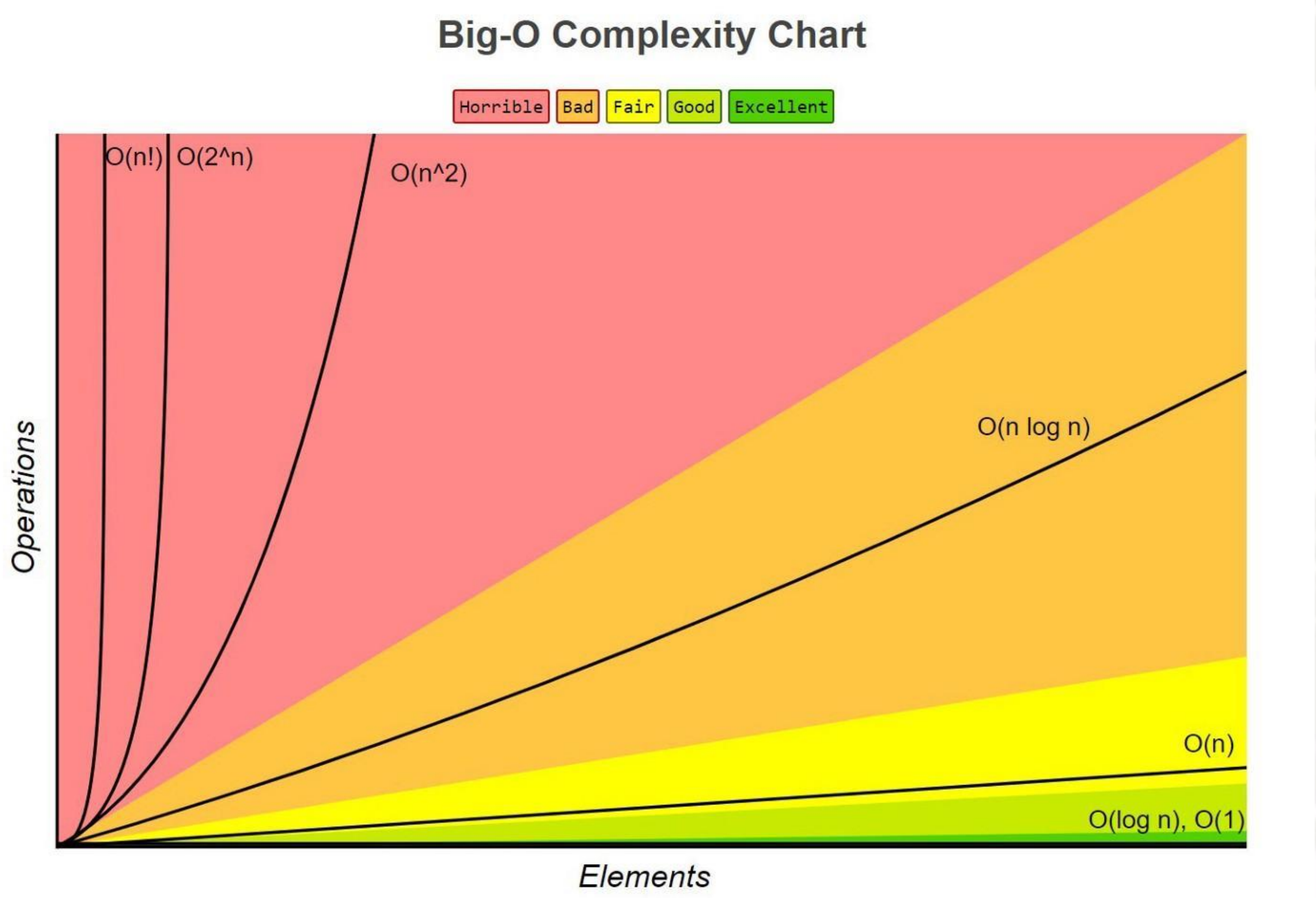
- Ignore constants
- Certain terms dominate others:

O(1) < O (log n) < O(n) < O(n log n) <

O(n2) < O(2n) < O(n!)

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

Big O Complexity Chart



Big O Measure of Efficiency

- Measure of efficiency for n = 10,000

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	Microsecond
Linear	$O(n)$	10,000	Seconds
Linear Logarithmic	$O(n \log n)$	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 Complexity Chart

- Looks into the time that is taken to finish a set of operations
- Usually states the efficiency of the algorithm
  - “Horrible” notations can be faster depending on the number of operations
  - “Good” notations are usually better on shorter operations, but worse at larger ones

Big O Complexity Chart

- Constant time
- Linear time
- Quadratic time

Big O Complexity Chart

Constant Time

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

Big O Complexity Chart

Linear Time

- The more operations, the slower the time
- Better for shorter operations
- Notation(s): O(log N), O(N), O(N log N)
- Sample operations:
  - Decision control structures
  - One-layered iterative control structures

Big O Complexity Chart

Quadratic Time

- The more operations, the shorter the time
- Better for longer operations
- Notation(s): O(N2), O(2N), O(N!)
- Sample operations:
  - Nested iterative control structures
  - Recursive loops
  - Permutations

Big O Notation

- O(1)
  - Constant; most instructions are executed once or at most only a few times
- O(log n)
  - 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
- O(n)
  - Linear; proportional to the size of N

Big O Notation

- O(n log n)
  - Occurs in algorithms that solve a problem by breaking it up into smaller sub-problems, solve them independently and then combining the solution
- O(n^2)
  - Quadratic; can be seen in algorithms that process all pairs of data items.
- O(n^k)
  - Polynomial; algorithms that process polynomial of data items.

Big O Notation

- O(2^n)
  - Exponential; brute-force situation
- O(n!)
  - Factorial

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

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

Big O Notation

N	P1	P2
1	__	__
5	__	__
10	__	__
15	__	__
20	__	__
30	__	__

Substitute values in N and determine which among the two algorithms is more efficient and faster.

Big O Notation

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

P2 is faster and more efficient for N <= 20, but for N>20, P1 proves to be faster than P2.

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;
}
```

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];
          }
     }
}
```