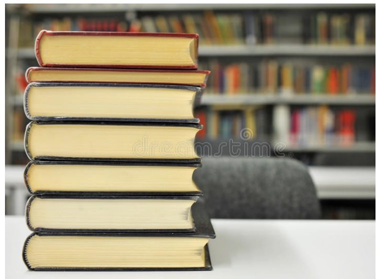
Data Structures and Algorithms

Instructor – Milan Lamichhane

Lecture 1

Data Structure

- Data Structures are systematic methods for organizing data in a computer, that can be used effectively.



Information and Meaning

- What is information?
- Measuring the quanities of information
  - The basic unit of information is the ***bit,*** whose values asserts one of two mutually exclusive possibilities
  - bit pattern to represent
    - Binary and decimal integers (38)
    - Real numbers (38.999)
    - Character strings (‘A’)
  - A method of interpreting a bit pattern is often called a ***data type***
  - For example, the bit string ***00100110*** can be interpreted as
    - the number 38 (binary)
    - the number 26 (BCD)
    - the character ‘&’

Information and Meaning

- Bits are grouped together into larger units such as ***bytes.***
- Several bytes are then grouped together into units called ***words.***
- Each unit (byte or word) is assigned a location (or address) in a computer memory.
- Every computer has a set of “native” data types
  - It is constructed with a mechanism for manipulating bit patterns consistent with the objects they represent.
  - A data type consists of the values it represents, and the operations defined upon it.
- It is the programmer’s responsibility to identify which data type to use and which operation to perform.

Information and Meaning

In C programming language, the declaration below,

int *x, y* ;

space is reserved at two locations for 2 different numbers, referenced by ***identifiers*** *x* and *y*.

Again, the compiler that is responsible for translating C programs into machine language will translate the “+” in the statement below into *integer addition*.

x = x+y;

Concept of Implementation

- The set of native data types that a particular computer can support is determinded by what functions have been wired into its hardware.
- Once the concept of “data type” is divorced from the hardware capabilities of the computer, a limitless number of data types can be considered.
- A data type becomes abstract concept defined by a set of logical properties.
- Once such an abstract data type is defined and the legal operations involving that type are specified, we may implement that data type.

Concept of Implementation

- ***Hardware implementation*** – circuitry necessary to perform the required operation is designed an constructed as part of a computer.
- ***Software implementation*** – a computer program consisting of already existing hardware instructions is written to interpret bit strings in the desired fashion and to perform the required operations.

Abstract Data Type

- An ***abstract data type(ADT)*** is a programmer-defined data type that specifies a set of data values and a collection of ***well-defined operations*** that can be performed on those values.
- Only the formal definition of the data type is important and **NOT** how it is implemented in binary form or in hardware.
- This is sometimes called, "***Separation of Interface and Implementation***".
- ***Information Hiding -*** how data is represented and how operations are implemented is completely irrelevant when a new Abstract Data Type (ADT) is defined.
- A useful tool for specifying the logical properties of a data type.

Abstract Data Type

String as an ADT

```
String Data Type:
    An string of characters like
    s = "Hello World"
    s = "Guido Van Rossum, 1993"

Operations:
    upper(s)            All characters to upper case
    lower(s)            All characters to lower case
    find(s,w)           Find a word w in s (return index)
    replace(s,w1,w2)    Replace sub word w1 with w2
```

Example code

```
s = "Hello World"
upper(s) = "HELLO WORLD"
lower(s) = "hello world"
find(s, "Wo") = 6
replace(s, "lo", " NEW") = "Hel NEW World"
```

Abstract Data Type

- Note that the term "***string of characters***" does not imply anything about its implementation (how English characters are represented).
- It can be implemented as a C array of characters terminated by a NULL character.
- It can be implemented like a Java or C++ String object.
- We may even decide to encode and compress the string if it size is too large.
- We can decide to break each string to chunks of 4K in different memory locations and keep a central table for accessing these chunks, etc.

Abstract Data Type

- Similarly, nothing on how the ***find()*** and ***replace()***algorithms should be implemented is mentioned!
- All we care is about how we <u>interface</u> with the string data type. (“what/how to do?” instead of “how it is done?”)
- All implementation issues are <u>irrelevant</u> to the ADT specification!

Abstract Data Type

- Rational Number as an ADT
- A rational number is a number that can be expressed as the quotient of two integers
- The operations defined are;
    - creation of a rational number from two integers
    - addition
    - multiplication
    - testing of equality

```
      /*value definition*/
abstract typedef <integer, integer> RATIONAL;
condition RATIONAL[1] != 0;

      /*operator definition*/
abstract RATIONAL makerational(a,b)
int a,b;
precondition  b != 0;
postcondition makerational[0] == a;
              makerational[1] == b;

abstract RATIONAL add(a,b)                /* written a + b */
RATIONAL a,b;
postcondition add[1] == a[1] * b[1];
              add[0] == a[0] * b[1] + b[0] * a[1];

abstract RATIONAL mult(a,b)               /* written a * b */
RATIONAL a,b;
postcondition mult[0] == a[0] * b[0];
              mult[1] == a[1] * b[1];

abstract equal(a,b)                       /* written a == b */
RATIONAL a,b;
postcondition equal == (a[0]*b[1] == b[0]*a[1]);
```

Abstract Data Type

- Implementation of Rational ADT in Python

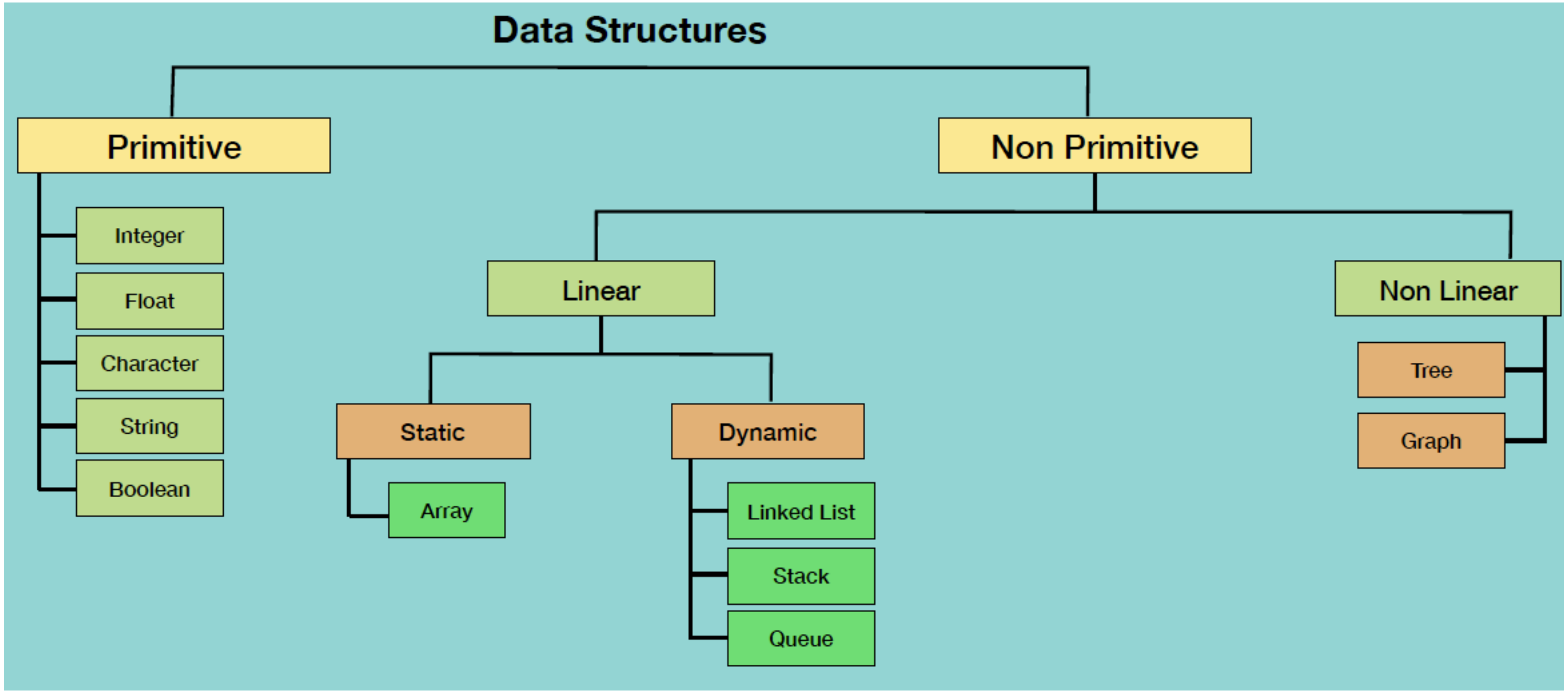
A Rational class →

```python
from math import gcd

class Rational:
    def __init__(self, numerator: int, denominator: int):
        if denominator == 0:
            raise ValueError("Denominator cannot be zero.")
        self.numerator = numerator
        self.denominator = denominator

    def __add__(self, other):
        if isinstance(other, Rational):
            new_numerator = self.numerator * other.denominator + other.numerator * self.denominator
            new_denominator = self.denominator * other.denominator
            return Rational(new_numerator, new_denominator)
        return NotImplemented

    def __sub__(self, other):
        if isinstance(other, Rational):
            new_numerator = self.numerator * other.denominator - other.numerator * self.denominator
            new_denominator = self.denominator * other.denominator
            return Rational(new_numerator, new_denominator)
        return NotImplemented

    def __mul__(self, other):
        if isinstance(other, Rational):
            new_numerator = self.numerator * other.numerator
            new_denominator = self.denominator * other.denominator
            return Rational(new_numerator, new_denominator)
        return NotImplemented

    def __truediv__(self, other):
        if isinstance(other, Rational):
            if other.numerator == 0:
                raise ValueError("Cannot divide by zero.")
            new_numerator = self.numerator * other.denominator
            new_denominator = self.denominator * other.numerator
            return Rational(new_numerator, new_denominator)
        return NotImplemented

    def __str__(self):
        return f"{self.numerator}/{self.denominator}"

    def __repr__(self):
        return f"Rational({self.numerator}, {self.denominator})"
```

Example usage

```python
r1 = Rational(3, 4)
r2 = Rational(5, 6)

print(f"R1: {r1}")              # Output: R1: 3/4
print(f"R2: {r2}")              # Output: R2: 5/6
print(f"R1 + R2: {r1 + r2}")    # Output: R1 + R2: 19/24
print(f"R1 - R2: {r1 - r2}")    # Output: R1 - R2: -1/24
print(f"R1 * R2: {r1 * r2}")    # Output: R1 * R2: 15/24
print(f"R1 / R2: {r1 / r2}")    # Output: R1 / R2: 18/20
```

Types of Data Structures



Python Primer

- Objects in Python
  - Identifiers, objects, and the assignment statement
  - Creating and using objects
  - Python’s built-in classes
- Expressions, operators, and precedence
- Control flow
  - Conditionals
  - Loops
- Functions

Python Primer

- Simple input and output
  - Console i/o
  - Files
- Exception Handling
- Iterators and generators
- Modules and the *import* statement

Object-Oriented Programming

- Goals, principles, and patterns
- Class definitions
  - Operator Overloading and Python's Special Methods
  - Iterators
- Inheritance
- Namespaces and object-orientation
- Shallow and deep copying