Data Structures

Lecture 01: Data Structures and ADTS

Data Structures

Data Structures

- Building blocks of a program
- A data structure is a systematic way of organizing a collection of data
- It is a mathematical or logical model of particular organization of data items
- It is used to represent a mathematical model underlying an ADT
- Every data structure needs a variety of algorithms for processing the data in it:
  - algorithms for insertion, deletion, retrieval, etc.
- So, it is natural to study data structures in terms of
  - properties
  - organization
  - operations

Why data structures? (1/2)

Aren't primitive types, like integers, floats and characters, and simple arrays enough?

- Yes, since the memory model of a computer is simply an array of Booleans

Why data structures? (2/2)

- But, this model . . .
  - is *conceptually inadequate*, since information is usually expressed in the form of *highly structured data*
  - makes it *difficult to describe complex algorithms*, since the basic operations are too primitive
  - makes it *difficult to reason about the efficiency of algorithms*, since computer instructions are not operating on the right level of abstraction

Classification of Data Structure

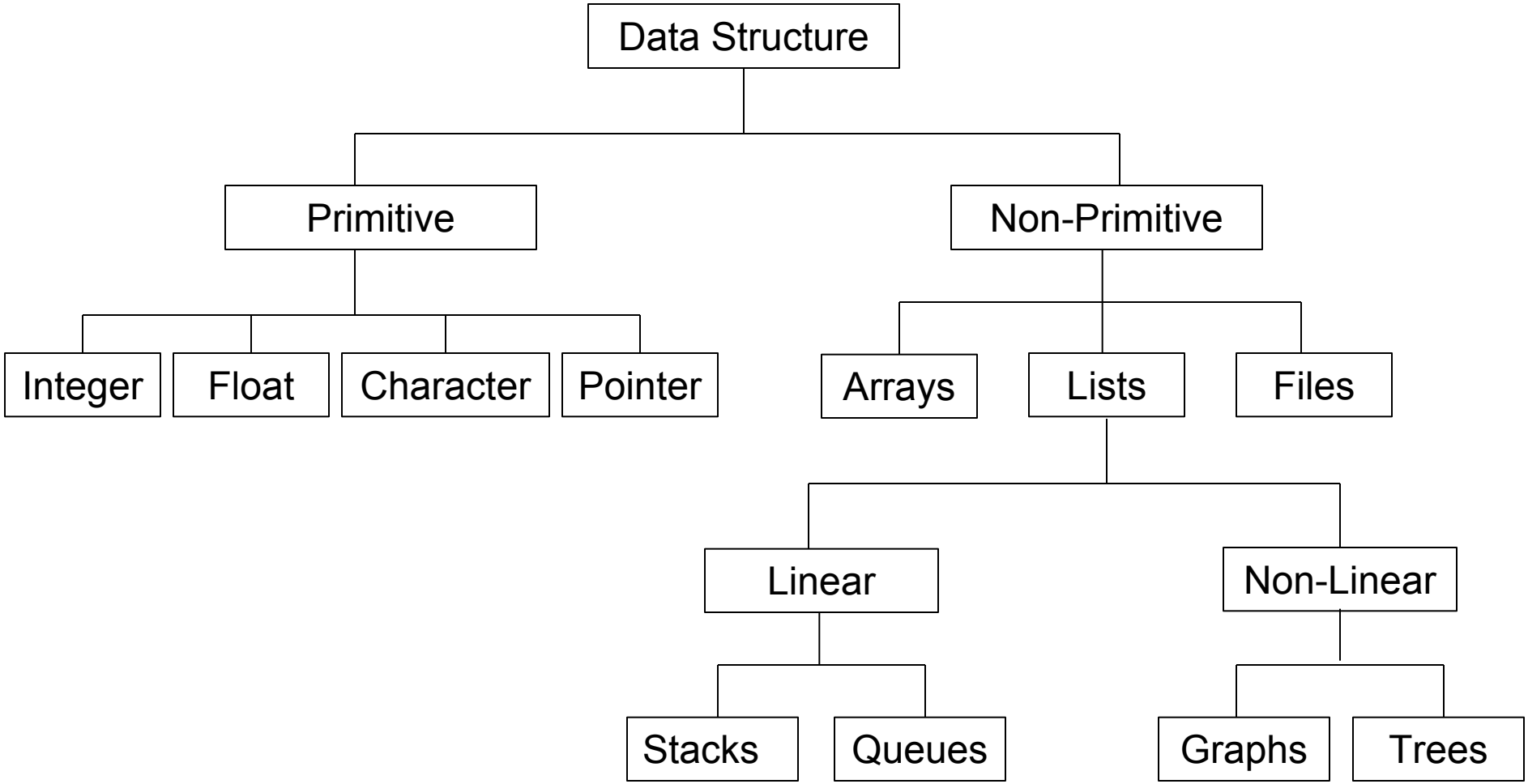

Fig: Classification of Data Structure

Data Types

- A data type is a *collection of values* and *a set of operations* on those values
- Example
  - the type **int** in C consists of the values {0, +1, -1, +2, -2, …, MAXINT, MININT}
  - the operations on **int**s includes the arithmetic operators +, -, *, and / along with operators for testing equality and inequality

Atomic and Structured Types

- Atomic types are those whose values are single entities and can't be subdivided
  - Types such as int, double, char in C
- Structured (Composite) types are made up of *simpler data types* and there is a *structure*, a set of rules for putting the individual types together
  - Arrays, pointers and structures allows to create structured types in C

Abstract Data Types (1/2)

- An abstract data type (ADT) is a mathematical construct *of a data type* that is organized in such a way that the specification of the values and the specification of the operations on the values is *separated* from the representation of the values and the implementation of the operations
- **ADT:** ***A mathematical model with a collection of operations defined on that model***

Abstract Data Types (2/2)

- It is a tool for specifying the logical properties of a data types
- An ADT only defines a formal description what the operations do
- An ADT is *not concerned* with implementation details
  - It may not even be possible to implement a particular ADT on a particular piece of hardware or using a particular software system

ADTs Operations

- The operations can take as operands *not only* instances of the ADT being defined but other types of operands or instances of other ADTs
- The result of an operation can be other than an instance of that ADT
- At least one operand, or the result, is of the ADT in question

ADT Example

- Sets of integers, together with the operations of union, intersection, and set difference, form a simple example of an ADT
- Lists, Stacks, Queues, Sets, Trees, Graphs are some of the examples of ADTs
- Object-oriented languages such as C++ and Java provide explicit support for expressing ADTs by means of *classes*

Data Structures: Implementation of an ADT

- An ADT consists of two parts:
  - A value definition (definition clause + condition clause)
  - An operator definition (header + pre + post conditions)

- An implementation chooses a data structure to represent the ADT; each data structure is built up from the basic types of the underlying programming language using the available data structuring facilities

Data Structures in C

- Arrays and structures are two important data structuring facilities that are available in C
- For example, one possible implementation for variable S of type SET would be an array that contained the members of S

The ADT List

- Except for the first and last items, each item has a unique predecessor and a unique successor
- Head or front do not have a predecessor
- Tail or end do not have a successor
- Items are referenced by their position within the list

The ADT List

- Specifications of the ADT operations
  - Define the contract for the ADT list
  - Do not specify how to store the list or how to perform the operations
- ADT operations can be used in an application without the knowledge of how the operations will be implemented

The ADT List

- ADT List operations
  - Create an empty list
  - Determine whether a list is empty
  - Determine the number of items in a list
  - Add an item at a given position in the list
  - Remove the item at a given position in the list
  - Remove all the items from the list
  - Retrieve (get) item at a given position in the list

The ADT Sorted List

- The ADT sorted list
  - Maintains items in sorted order
  - Inserts and deletes items by their values, not their positions

The End