Data Structures
Fall 2018
Fundamentals of Data Structures

Luis de Marcos Ortega

luis.demarcos@uah.es

Contents

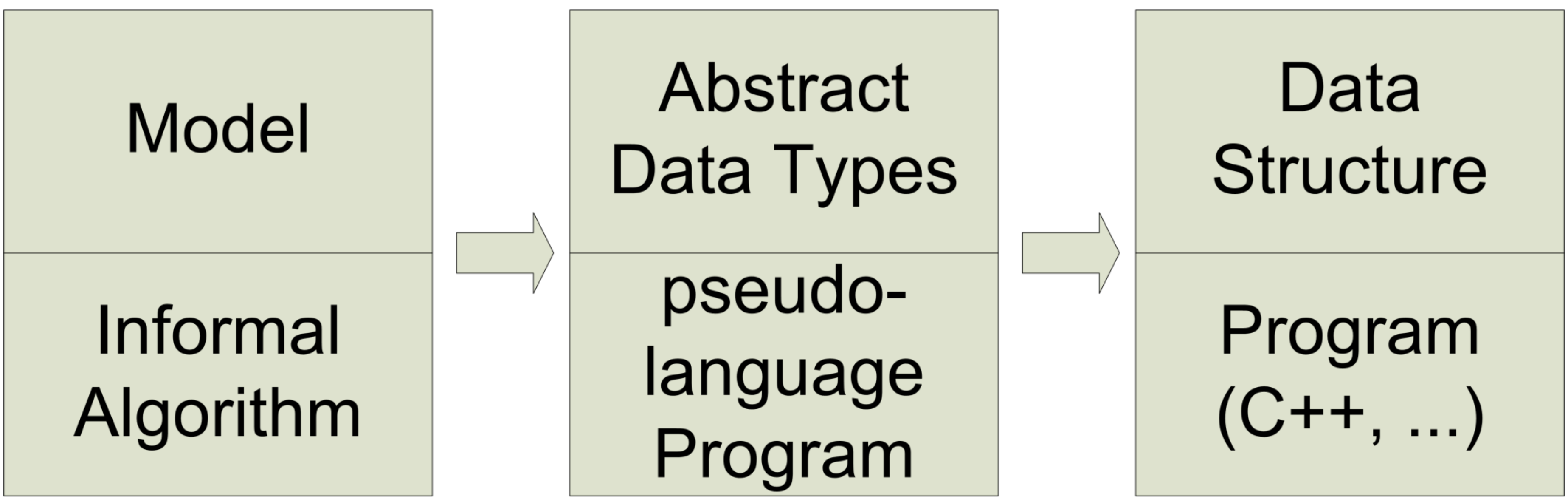
- Problem solving
- ADTs
- Pointers
- Big Oh notation
- Pseudocode

Bibliography

- Chapter 1 of:
  - A.V. AHO., J.E. HOPCROFT., J.D. ULLMAN. 1987. “Data Structures and Algorithms.” Addison-Wesley.

Problem Solving

- We use *programs* to solve real world *problems* (or at least try to help…)



Problem Solving

- *Data structures* are concerned with the *representation* and *manipulation* of data
- All programs represent and manipulate data in one way or another
- Data manipulation requires an *algorithm*

- PROGRAM = DATA STRUCTURE + ALGORITHM

Problem Solving

- Algorithm – A finite sequence of instructions to solve a problem.
- Each instruction has a clear meaning and can be performed with a finite amount of effort in finite lenght of time

Abstract Data Types

- An *Abstract Data Type* (*ADT*) is a *model* with a collection of *operations* defined on that model.
  - Example: Set of integers, together with operations of union, intersection and difference
- Operations can take as *operands* (arguments):
  - Instances of the ADT (e.g. a set of integers)
  - Instances of other objects (e.g. an integer, bool, …)
- At least one operand (or the result) is assumed to be of the ADT in question

Abstract Data Types

- Properties of ADTs:
  - *Generalization*. ADTs are independent of **any implementation**.
  - *Encapsulation*. Not neccesary to know the details. All the code will be in same section.
- Other advantages:
  - *Privacy* as details are hidden.
  - *Protection* as only the operations defined (as public) can be called.

Examples

ADT for a LIST of integers:

- Operations (informal language):
  1. Make a list empty
  2. Get the first element of the list and return null if the list is empty
  3. Get the next member of the list and return null if there is no next member
  4. Insert an integer into the list

Examples

ADT for a LIST of integers (II):

- Operations (pseudo-code):
  1. `func makenull (newlist:LIST)`
  2. `func first (mlist:LIST) ret f:int`
  3. `func next (mlist:LIST) ret n:int`
  4. `func insert (mlist:LIST, e:int) ret pos:int`

Examples

- Date:

```
func create (day, month, year:natural) ret d:date

func increase (ini_d:date; num_days:int) ret final_d:date

func difference (ini_d, fin_d:date) ret diff:int

func week_day (d:date) ret day:1..7
```

Abstract Data Types

- The developer of an ADT must write a *specification* (spec):
  - Consisting of a name and operations
  - The only part visible to the user
- but also its *implementation*:
  - Hidden to the user

Abstract Data Types

- The emphasis is on the operations
  - An ADT is not just a collection of data
- An ADT is an unambiguous *abstraction*:
  - Showing details of the specification (few)
  - Hiding details of implementation (many)

Abstract Data Types

- *Functional notation* to define specifications of ADTs. It includes:
  - Name and type of the ADT
  - Constants and its types
    - e.g. *0:->natural* or *T,F:->bool*
  - Operations: name, return value and arguments
    - e.g. *_+_:natural natural->natural*

Example I

- An ADT for natural numbers with add and substract operations:

```
spec NATURALS
   genre natural
   operations
         0:->natural {a constant}
         nxt:natural->natural {unary op}
         _+_:natural natural->natural
endspec
```

Abstract Data Types

- Functional notation. *Keywords*:
  - For the specification
    - spec – for the name of the specification
    - uses – indicates other specifications used
    - genre – new data types defined in the present spec
    - operations – using prefix notation and _ to denote arguments

Example II

```
spec BOOLEANS
    genres bool
    operations
        T,F:->bool
        ¬:bool->bool
        _∧_, _∨_:bool bool→bool
endspec
```

Example III

```
spec INTEGERS
    genres integer
    operations
        0:->int
        next, prev:int->int
        _+_, _-_:int int->int
endspec
```

Abstract Data Types (Nomenclature)

- ADT = model + operations
- *Data Type* – Type of a variable. Either:
  - ADT
  - Primitive data type (integer, real, bool, …)
- *Data Structure* – Implementation of the ADT on a specific programming language

Abstract Data Types (Nomenclature)

- *Cell* – Basic building block of a *Data Structure*

- Some basic *Data Structures* provided by most programming languages:
  - Array – sequence of cells of a given datatype
  - Record – Collection of cells of (possibly) dissimilar datatypes
  - Files – sequence of values of a particular type (non-random access)

Pointers and Cursors

- Pointer – A cell whose value indicates another cell
  - The value is usually a memory address
- Operations
  - Declaration: *ptr: ^int* or *ptr: pointer to int*
  - Memory allocation: *allocate(p)*
  - Disposal: *dispose(p)*
  - Derefence (access the object pointed): *p^ := 33*

Pointers and Cursors

- Cursor – An integer-valued cell used as a pointer to another object. As a method of connection, the cursor is essentially the same as a pointer, but a cursor can be used in languages that DO NOT have explicit pointer types (e.g. Java)

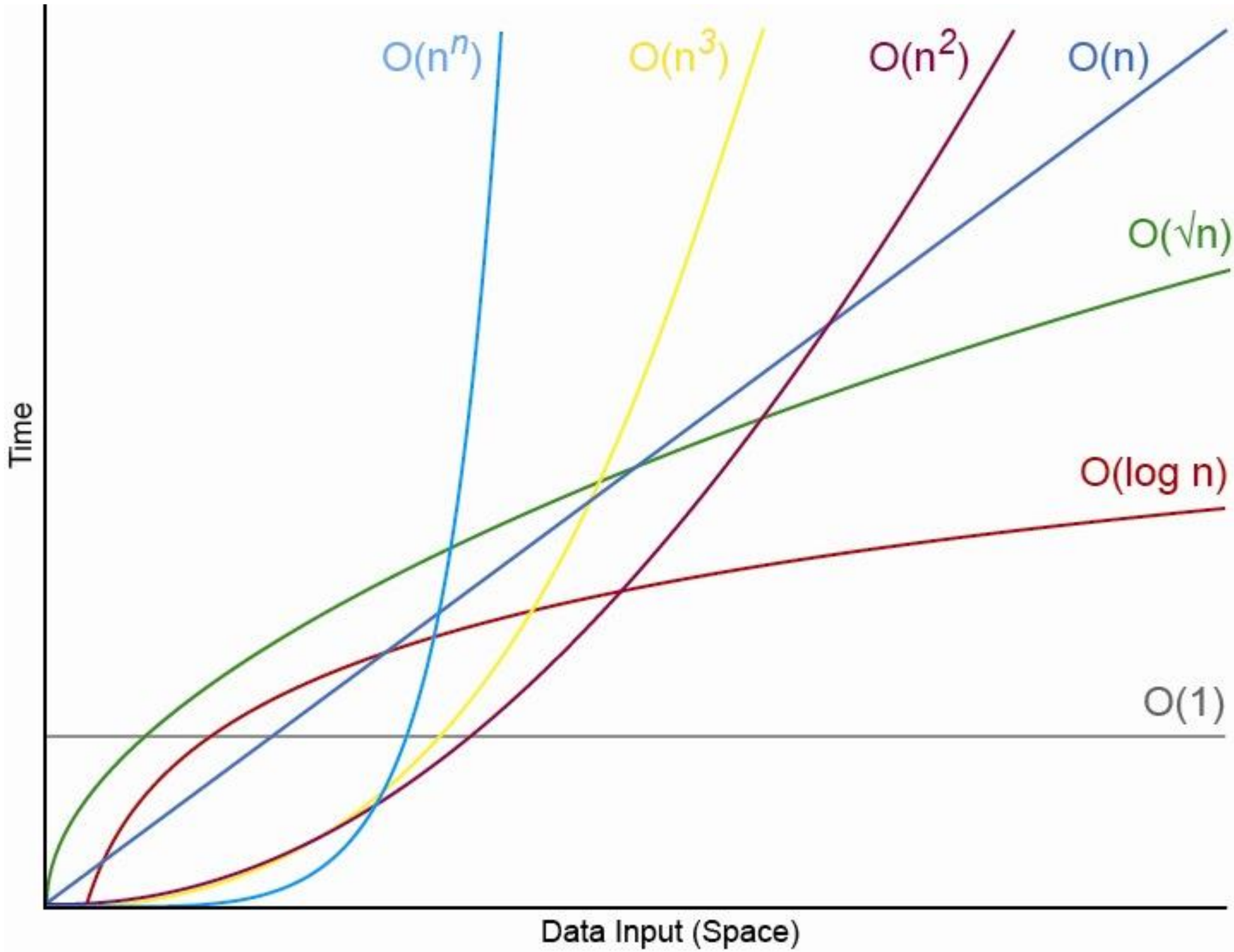
Big Oh Notation

- Big O notation is used to talk about *running time* T(n) of a program as a function of the *size* (n) *of the input*.
- For example, when we say that the running time of a program is $O(n^2)$, read "big oh *n* squared" or just "oh of *n* squared," we mean that there are positive constants *c* and n_0 such that for n equal or greater to n_0, we have $T(n) \leq cn^2$.

Big Oh Notation

- A program whose running time is *O(f(n))* is said to have a *growing rate f(n)* in the *worst case*.
- We shall assume that programs can be evaluated by comparing their running time functions, with constants of proportionality neglected.
- Example: Which program would be better?
  - P1 with $f(n)=100n^2$
  - P2 with $f(n)=5n^3$

Big Oh Notation



Big Oh Notation

- General rules for the analysis of programs:
  1. The running time of a simple statement (e.g. assignment, read, write, …) is O(1).
  2. The running time of a sequence of disjoint statements is the largest running time of any statement in the sequence.
  3. The running time a conditional statement is the running time of the largest branch (if-then or else).
  4. The running time of a loop is the sum, over all times around the loop of the time to execute the body (e.g. O(n) if the body is O(1) and the loop is executed n times).

  In 3 and 4 the time to evaluate the condition is neglected providing that it is O(1).

Example

```
void bubbleSort(int *n,int length)//Bubble sort
{
    int i,j;
    for(i=0;i<n.length;i++)
    {
        for(j=0;j<i;j++)
        {
            if(n[i]>n[j])
            {
                int temp=n[i]; //swap
                n[i]=n[j];
                n[j]=temp;
            }
        }
    }
}
```

Big Oh Notation

- *Rule of thumb* for calculating the running time of a program intuitively:
  - $O(n^k)$ where k would be the number of nested loops on the worst branch (case) of the program

- This rule doesn't work always!!!
  - Recursive calls
  - Loops do not execute *n* times:
    - Loops not proportional or unrelated to the length of the input of the program

Big Oh Notation

- We will be using the big-oh notation to denote the ***worst case*** running time of operations
- Three cases:
  - Worst case
  - Best case
  - Average case
- Best and average are beyond the scope of this course and will only be mentioned sometimes

Big Oh Notation

- Big Oh notation is also used to denote the ***complexity*** of problems
  - In this case a given Oh would mean that no known algorithm can do better
  - e.g. Sorting by comparison complexity is O(n log n)
- Therefore remember that big oh can denote:
  - Efficiency (running time)
  - Complexity
- Having different meanings

Pseudocode

- We will use the abstract imperative language (aka pseudocode) as an informal high-level description of the operating principles of our programs.

Pseudocode

- Comments between curly brackets {comment…}
- Assignment

  x:=v

- In what follows:
  - c1, c2, …, cn are supposed to be conditions
  - P1, P2, …, Pn are supposed to be programs (i.e. arbitrary sequences of sentences)
- Case

  case

  c1 → P1

  c2 → P2

  c3 → P3

  endcase

Pseudocode

- Conditonals

  if c2 *then* P1 *else* P2 *endif*
- Loops

  while c1 *do* P1 *endwhile*

  for i=initialvalue *to* endvalue *step* p *do* P *endfor*
- Basic I/O

  read()

  write()

  error()

Pseudocode

- Primitive data types:
  - *void:* empty
  - *bool*
  - *nat* (natural), *int* (integers), and *real*
  - *char* for characteres
  - *enum*{value_1, … , value_k} for enumrated:
  - Ranges: i..j (e.g. index:1..10)
  - vectors: v[i,j]
  - records

```
record
        field1: type
        …
        fieldn: type
endrecord
```

Pseudocode

- Classes

  class classname

  private member

  public member

  …

  endclass

- Members can be either attributes or methods

Pseudocode

- Methods

  returntype [classname::]name(arg_1:type, …, arg_n:type)

  var x_1:type, …, x_n:type

  P {code including one or more ‘*return*’}

  Endmethod
- Declaring objects as with any other datatype

  var objectname: classname
- Accesing members

  Attributes: objectname.member

  Methods: objectname.member(arguments)

  this can be used to refer to the present class (emphasize)

Pseudocode

- Pointers

  Declaration: *p: ^datatype* or *p: pointer to datatype*

  Memory allocation: *allocate(p)*

  Disposal: *dispose(p)*

  Access the object pointed using *p^*

Pseudocode

- Functions

  returntype name(arg_1:type, …, arg_n:type)

      var x_1:type, …, x_n:type

      P {code including one or more ‘*return*’}

  endfunc

- Procedures

  proc name(arg_1:type, …, arg_n:type)

      var x_1:type, …, x_n:type

      P {code}

  endproc

Pseudocode

- Specifications

```
spec NAME
    uses LIST_OF_SPECS_USED
    parameter
        ...
    endparameter
    genres NAME_OF_TYPE/S
    operations
        ...
    private operations
        ...
endspec
```

Some shortcuts

- Operations → Ops
- Specification → Spec

Fundamentals of Data Structures

Luis de Marcos Ortega

luis.demarcos@uah.es