Departamento de Informática
Universidad de Valladolid
Campus de Segovia

TOPIC 4: ABSTRACT DATA TYPES (ADTs)

ABSTRACT DATA TYPES (ADTs)

- Introduction
- A counterexample:
  - Euclide´s algorithm implementation without using ADTs.
- How to construct an ADT
- The same Euclide´s algorithm implementation using an ADT.

INTRODUCTION

- In computing, an **abstract data type** (**ADT**) is a specification of a set of data and their valid set of operations.
- In this sense abstract means that it is independent of the implementation.
- The definition can be mathematical, or programmed as an interface.
- If it is programmed then the interface provides a *constructor*, which returns an object, and several *operations*, which are functions accepting this object as an argument.

INTRODUCTION

- Users of an ADT are concerned with the interface, but not the implementation.

- The strength of an ADT is that the implementation is hidden from the user. Only the interface is published.

- This means that the ADT can be implemented in various ways, but as long as user programs are unaffected. (This supports the principle of information hiding, or protecting the program from design decisions that are subject to change.)

ADT vs IMPLEMENTATION

- There is a distinction, although sometimes subtle, between the abstract data type and the data structure used in its implementation.

- For example, a **List ADT** can be represented using an **array-based implementation** or a **linked-list implementation**.

- A **List** is an abstract data type with well-defined operations (add element, remove element, etc.) while a **linked-list** is a pointer-based data structure that can be used to create a representation of a List.

- The **linked-list** implementation is so commonly used to represent a List ADT, so that the terms are interchanged in common use.

A COUNTEREXAMPLE

- The storage data structure “**SET**” in Pascal presents a strong restriction with respect to the maximum number of elements that can be contained (the **SET type** in Turbo Pascal 7.0 can contain only 256 elements).

- The proposal involves to implement the **Eratosthenes´s sieve Algorithm** (find all the primes between 2 and *N*, removing the non-primes, where “N” may be any integer), using a new data “set” structure defined by the user, that overcomes the previous limitation.

- In this first approach the ADT scheme will not be considered.

CHOSING THE NEW DATA SET REPRESENTATION

- Eratosthenes's original algorithm was based on the idea of removing non-primes from a **list of integers.**
- Then, in order to represent the new Data Set, a growing ordered Linked list will be used.

ERATOSTHENES´S SIEVE ALGORITHM

- To find all the primes between 2 and *n*, Eratosthenes would proceed as follows:
  - First design step:

    Read N

    Generate the initial list containing the N-2 elements {2,...,N}

    Remove the non-prime numbers from list.

    Show the remaining numbers.
  - Generating the initial list:

    Create an empty list

    Add from 2 to N numbers to list.
  - Remove the non-prime numbers from list.

    Given e $\in$ (2, sqrt(N)), Remove from list all multiple of “e”.
  - Remove from list all multiple of “e”.

    coeficient:=2

    repeat

    Remove multiples of “e” from list as long as (e*coefficient) is less or equal than “N”

    coefficient:=coefficient+1

    Until (e>= sqrt(N))

```
PROGRAM Eratosthenessieve;
{Prec. Input must be an integer equal or greater than 2}
TYPE
    tset=^tNode;
    tNode=record
            elem:integer;
            next:tset
    end; {tNode}
VAR
    N,e,coeff:integer;
    initialset,aux,pprime,remaux:tset;
```

```
BEGIN
    writeln('maximum ordinal number of the set: '); {Read N}
    readln(N);
    new(initialset);
    aux:=initialset;
    aux^.elem:=2;
    for e:=3 to N do begin
            new(aux^.next);
            aux:=aux^.next; {linking the nodes of the list}
            aux^.elem:=e
    end; {for}
    aux^.next:=nil  {allocating the NIL value to the last node}
    {Removing non-prime numbers from initialset}
    {Pprime is a pointer that points there where initialset is pointing}
    Pprime:=initialset
```

```
repeat
      e:=Pprime^.elem;
      coeff:=2;
      aux:=Pprime;
      while (e*coeff≤N) and (aux^.next<>nil) do begin
            if (aux^.next^.elem)<(coeff*e) then
                  aux:=aux^.next
            else if aux^.next^.elem=coeff*e then begin
                  remaux:=aux^.next;
                  aux^.next:=aux^.next^.next;
                  Dispose(remaux);
                  coeff:=coeff+1
            end; {else if}
             else if aux^.next^.elem>coeff*e then
                  coeff:=coeff+1
      end {while}
      Pprime:=Pprime^.next
until (e≥sqtr(N)) or (Pprime=nil);
```

```
    {Show the remaining numbers on the screen}
    aux:=initialset;
    while (aux<>nil) do begin
        write(aux^.elem:4);
        aux:=aux^.next
    end; {while}
end. {Eratosthenessieve}
```

SOME CONSIDERATIONS ABOUT THE PREVIOUS IMPLEMENTATION

- The features of the algorithm and those ones related to data structure are mixed. That means:
  - The generated code is complex and difficult to read.
  - There is not any possibility to reuse the implemented data structure.
- All these problems can be overcome by using ADTs.

ADT DEFINITION

- An abstract data type may be defined as an abstract representation model of data consiting of three components:

1. A set of abstract objects.

2. A set of sintactic decriptions of operations which arguments are the abstract objects previously mentioned.

3. A complete semantic decription for each operation.

DEVELOPING AN ADT

- Identification of possible abstract objects.
- Identification of the basic operation related to these abstract objects.
- Operation specification.
- Choosing a good operation implementation.

ADT SPECIFICATION

- The appropriate language for specifying an abstracta data type is the mathematical one.
- The syntactic definition will be expressed in terms of the operation headers (identifier and argument description),
- whereas the semantic one describes the meaning of these operations using mathematical expressions.

ADT OPERATION CATEGORIES

- Operations on an ADT fall into four categories. These categories are:
  - **Constructors** - create an instance of the ADT
  - **Interrogators** - return information about an instance without modifying the instance
  - **Manipulators** - modify the properties of an instance without returning any information about it
  - **Destructors** - de-allocate storage space, close any open documents, and release system resources

SUPPORTING ABSTRACT DATA TYPE IMPLEMENTATION

- In order to implement an abstract data type is neccessary a framework that supports encapsulation mechanism and therefore information hiding:

  - Encapsulation involves modullarity and therefore reusability
  - Information hiding involves protecting the interface with respect to changes if the design decision is changed.

- Turbo Pascal 7.0 is provided of such mechanism by means of the UNITs, in order to satisfy as much as possible both of them.

THE “SET” ABSTRACT DATA TYPE

TYPE

tSet=Abstract

{The abstract object of this ADT is a set of integers}

- The related operations are:
  - **Createset(iset):** Creates an empty set of integers.
  - **Addelem(elem,iset):** adds “elem” (an integer) to iset.
  - **Removeelem(elem,iset):** Removes “elem” (an integer) to iset.

THE “SET” ABSTRACT DATA TYPE

– **Belong(elem,iset):** Determines whether “elem” belongs to iset or not.

– **showset(iset):** Shows all the elements of iset on the screen.

– **Emptyset(iset):** Determines whether iset is empty or not.

OPERATION SPECIFICATION

PROCEDURE Createset(var iset: tset);

{Returns iset:=Ø}

PROCEDURE Addelem (elem:integer; var iset:tset);

{Returns iset:=iset ∪ [elem]}

PROCEDURE Removeelem(elem:integer; var iset:tset);

{Returns iset:=iset / [elem]}

FUNCTION Belong(elem:integer; iset:tset):boolean;

{Returns True if (elem∈iset), otherwise False}

PROCEDURE showset(iset:tset);

{Shows all the elements of iset on the screen}

FUNCTION Emptyset(iset:tset):boolean;

{Returns True if Conj:=∅, otherwise False}

THE “SET” ABSTRACT DATA TYPE IMPLEMENTATION

```
UNIT IntegerSet;
  {Implementation by means of an ascending ordered linked list without
   repetition}
INTERFACE
TYPE
   tElem:integer;
   tset=^tListNode;
   tListNode=record
          info:tElem;
          sig:tset;
   end; {tListNode}
```

THE “SET” ABSTRACT DATA TYPE IMPLEMENTATION

PROCEDURE Createset(var iset: tset);

{Efecto. iset:=Ø}

PROCEDURE Addelem(elem:integer; var iset:tset);

{Efecto. iset:=iset ∪ [elem]}

PROCEDURE Removeelem(elem:integer; var iset:tset);

{Efecto. iset:=iset / [elem]}

FUNCTION Belong(elem:integer; iset:tset):boolean;

{Dev. True if (elem∈iset) otherwise False}

PROCEDURE showset(iset:tset);

{Shows all the elements of iset on the screen}

FUNCTION Emptyset(iset:tset):boolean;

{Dev. True if Conj:=∅ otherwise False}

```
IMPLEMENTATION
PROCEDURE Createset(var iset: tset);
Begin
    Createlist(iset)
End; {Createset}

PROCEDURE Addelem(elem:integer; var iset:tset);
VAR
    insert:boolean;
Begin
        if  find(elem,iset)=nil then begin
                        insert:=true
          else
                        insert:= false;
        if insert then
                   Orderedinsert(elem,iset)
End; {Addelem}
```

```
PROCEDURE Removeelem(elem:integer; var iset:tset);
VAR
    Remove:boolean;
Begin
        if  Find(elem,iset)=nil then begin
                    remove:=false
           else
                    remove:= true;
        if remove then
                Delete(elem,iset);
End; {Removeelem}

FUNCTION Belong(elem:integer; iset:tset):boolean;
Begin
        Belong:=(Find(elem,iset)<>nil);
End; {Belong}
```

```
PROCEDURE Showset (iset:tset);
Begin
    if Emptyset(iset) then
            write('[ ]');
    else begin
            write('[',);
            View(iset);
            writeln(']')
    end; {else}
End; {showset}

FUNCTION Emptyset(iset:tset):boolean;
Begin
    Emptyset:=Emptylist(iset)
End; {Emptyset}

+  all linked list operations

End. {Set_of_Integers}
```

- And now the TYPE declaration is changed by a USES declaration:

```
TYPE
    tconj: Abstracto

por:

USES
    Integerset
```

```
PROGRAM Eratosthenessieve;
{Prec. integer >=2}
USES
    Integerset;
VAR
    N,e,coeff:integer;
    iset:tset;
BEGIN
    writeln('maximum number: '); {reading maximum set ordinal number}
    readln(N);
    Createset(iset);
    {initializing a new set of integers}
```

```
    for e:=2 to N do begin
        Addelem(e,iset);
    for e:=2 to Trunc (sqrt(N)) do
        if  Belong(e,iset) then begin
                coef:=2;
                repeat
                    Removeelem(e*coeff,iset);
                    coeff:=coeff+1;
                until (e*coeff>N)
        end; {if}
    Showset(iset)
End. {Eratosthenessieve}
```