# ABSTRACT DATA TYPES, CLASSES AND POINTERS

**About these slides:**

- Slides 1–13 replace slides 47–62 in the Lecture Notes package.
- Slides 14–15 replace slides 90–91.
- A number of small revisions will be made to slides 63–72 in lecture. In particular, change the word "module" to "class".

**Reading:**

- BHH, section 2.10 on pointers and linked-lists in Turing
- BHH, chapter 4 on data abstraction and modules





## The Ordered Set as an Abstract Concept

An **ordered set** is an abstract mathematical entity

- that is composed of a finite, non-negative number of elements
  (i.e., zero or more elements),
- whose elements come from an ordered domain, and
- that contains no two identical elements.

The operations that one can perform on an ordered set are:

**Size of:** returns the number of elements in an ordered set (0 if it's empty).

**Insert:** Given an item, adds it to the ordered set. Has no effect if the item already occurs in the ordered set.



**Retrieve:** Given an integer i between 1 and the size of an ordered set inclusive, returns the element of rank i in the ordered set, that is, the $i^{th}$ smallest value.

**Delete:** Given an integer i between 1 and the size of an ordered set inclusive, deletes the element of rank i in the ordered set, that is, the $i^{th}$ smallest value.

We have placed no constraints on how the ordered set might be implemented.

We have separated what an ordered set is from how it might be implemented.



## Abstract Data Types

When we specify data and operations that can be performed on it, we have defined an "abstract data type" (ADT).

We "implement" an ADT by:

- designing some data structure in our programming language that can represent the data, and
- providing subprograms that perform the operations.

There are *many* different ways to implement any ADT.

Each implementation has pros and cons. We must be aware of these so that we can recognize tradeoffs and make informed choices.

## Implementing the Ordered Set ADT

Decisions:

- data structure — a sorted array is one reasonable choice, since it is simple, and will permit some operations to be more efficient than if we used an unsorted array.
- subprograms — the algorithms will be quite simple.

**Question:** Could we implement the ordered set ADT using an unsorted array?

We will use a "class" to package up everything to do with implementing an ordered set.



## Classes

Classes are a programming language feature that support the distinction between an abstract data type and its implementation by:

- grouping together the declarations and subprograms that implement an abstract data type.
- specifying precisely what things inside the class a program that uses the class may access. This is called the **interface**. These things intuitively correspond to the abstract data type.
- preventing access to everything else within the class — ideally, all the implementation details.



## Advantages of Using Classes

- Programs that use a class need *only* know about the interface of the class. They can even be written before the rest of the class is completed.
- Programs that use a class cannot accidentally mess up any of the internal details (e.g., by changing a variable's value).
- When writing a large program, classes help break the task into smaller pieces (as subprograms do).
  "Divide and conquer" theme again.
- Classes can be easily reused or replaced.
- Classes can be in separate files from the programs that use them. In compiled languages, this means the class file has to be compiled just once, even if the program that uses it is recompiled many times.



## A Class of Ordered Sets

```
% File "orderedSet.t".
% For the sake of slide space, NOT well commented.

class orderedSet
    import elementType
    export sizeOf, insert, retrieve, delete

    const maxSize := 500
    var theSet : array 1 .. maxSize of elementType
    var numElements : int := 0

    function sizeOf : int
        result numElements
    end sizeOf

    procedure insert (element : elementType,
            var success : boolean)
        % Details omitted;
    end insert

    procedure retrieve (rank : int, var success : boolean,
            var answer : elementType)
        % Details omitted;
    end retrieve

    procedure delete (rank : int,
        var success : boolean)
        % Details omitted;
    end delete
end orderedSet
```

## With all the details:

```
% File "orderedSet.t".
% For the sake of slide space, NOT well commented.

class orderedSet
    import elementType
    export sizeOf, insert, retrieve, delete

    const maxSize := 500
    var theSet : array 1 .. maxSize of elementType
    var numElements : int := 0

    function sizeOf : int
        result numElements
    end sizeOf

    % (Not exported; just used by insert.)
    % Insert at given position in the set array.
    % Move things down, if necessary, to make room.
    % Precondition: 1 <= position <= numElements + 1
    %               and numElements < maxSize.
    procedure insertAt (position : int, element : int)
        for decreasing i : numElements .. position
            theSet (i + 1) := theSet (i)
        end for
        theSet (position) := element
        numElements += 1
    end insertAt
```



```
    % (Not exported; just used by insert.)
    % If the value does not occur in the set, "found" is
    % set to false, and "position" is set to the position
    % of the 1st element greater than "value",
    % (or numElements if there is no element greater).
    % Preconditions: numElements >= 1
    %                theSet[1..numElements] is sorted
    %                in non-decreasing order
    procedure binarySearch (value : elementType,
            var found : boolean, var position : int)

        var first, last, middle : int
        first := 1
        last := numElements
        loop
            exit when first = last
            middle := (first + last) div 2
            if value <= theSet (middle) then
                last := middle
            else
                first := middle + 1
            end if
        end loop
        position := first
        if theSet (first) = value then
            found := true
        else
            found := false
        end if

    end binarySearch
```



```
    % Will not insert duplicates.
    procedure insert (element : elementType,
            var success : boolean)

        if numElements < maxSize then
            success := true
            if numElements = 0 then
                numElements := 1
                theSet (numElements) := element
            else
                var found : boolean
                var position : int
                binarySearch (element, found, position)
                if not found then
                    if theSet (position) < element then
                        % position must be numElements, so
                        % we must be inserting after the
                        % current end of set.
                        insertAt (position + 1, element)
                    else
                        insertAt (position, element)
                    end if
                end if
            end if
        else
            success := false
        end if

    end insert
```



```
    procedure retrieve (rank : int, var success : boolean,
            var answer : elementType)
        if rank >= 1 and rank <= numElements then
            answer := theSet (rank)
            success := true
        else
            success := false
        end if
    end retrieve

    procedure delete (rank : int,
        var success : boolean)
        if rank >= 1 and rank <= numElements then
            for i : rank .. (numElements - 1)
                theSet (i) := theSet (i + 1)
            end for
            numElements -= 1
            success := true
        else
            success := false
        end if
    end delete

end orderedSet
```

### Syntax for using the class

```
% File "main.t".  Just demonstrates syntax for using the
% class.  Does not thoroughly test the class.

type elementType : int
include "orderedSet.t"

var MySet : ^orderedSet  % will point to an ordered set
new MySet                % creates an ordered set

loop
    var num : int
    var success : boolean
    put "What number to insert?  (0 to exit):  " ..
    get num
    exit when num = 0
    MySet->insert (num, success)
end loop
put "There are ", MySet->sizeOf, " elements.", skip

loop
    var rank : int
    var exists : boolean
    var num : elementType
    put "What rank element to retrieve?  (0 to exit):  " ..
    get rank
    exit when rank = 0
    MySet->retrieve (rank, exists, num)
    if exists then
        put "The ", rank, "th element by rank is ", num
    else
        put "There is no element of rank ", rank, "."
    end if
end loop
```



## Linked List Implementation of the Ordered Set ADT

```
% File "orderedSet.t".
% Implements the ordered set ADT with a
% sorted linked list.

class orderedSet
    import elementType
    export sizeOf, insert, retrieve, delete

    % All new data structure:
    type node :
        record
            entry : elementType
            next : ^node
        end record
    var front : ^node := nil
    var numElements : int := 0

    function sizeOf : int
        % This function doesn't happen to change.
        result numElements
    end sizeOf

    procedure insert (element : elementType,
            var success : boolean)
        % All new details omitted;
        % Does NOT use binarySearch and insertAt.
        % Why not??
    end insert
```



```
    procedure retrieve (rank : int, var success : boolean,
            var answer : elementType)
        % All new details omitted.
    end retrieve

    procedure delete (rank : int,
            var success : boolean)
        % All new details omitted.
    end delete

end orderedSet
```

**Exercise:** Fill in the details of the class

Remember that the main program will work just as well with this version of the class.