Motivation

- Most of the things we will study will *not refer to a specific* programming language.
- Things discussed during these classes can be applied when working in almost any programming language.

Pseudocode

Pseudocode

an artificial and informal language

"text-based" detail (algorithmic) design tool

helps programmers develop algorithms

- *standard* pseudocode statements

(only for this course:)

- *not standard* pseudocode statements

@ - notations for pseudocode not standard statement

ex.: *@ read the array x of length n*

Specifications of subalgorithms

Subalg. name(f.p.l.)

Description: …

Data: … Prec.: …

Rez.: … Post.: …

or

Subalg. name(f.p.l.)

Prec.: …

Post.: …

short

Specifications of subalgorithm

Function *funcname*(f.p.l.)

Description: …

Data: … Prec.: …

Rez.: funcname Post.: *funcname* = …

Pseudocode standard statements

```
if cond then
        …
else
        …
endif
```

```
for count ← iv , fv [, step] do
    …
endfor
```

```
repeat
    …
until cond
```

```
while cond do
        …
endwhile
```

```
read            lista_elem
print /write    lista_expr
```

. . .

- assignment operator: **:= or ←**

comments

- *// one line comment*
- */* more*
  than
  one line comment
  /

DS specification

- What (most) programming languages offer?
  languages provide basic data types

Convention:

We suppose that we have the next data types:

- *Integer*
- *Real*
- *Boolean*
- *Char*
- *String*

DS specification – notation conventions

Progr. languages provide d**ata structure builders**

Convention:
We suppose that we have
→ **static one-dimensional array**
array[firstindexvalue .. lastindexvalue] of TElement
x:***array***[fi..li] of TE means: $x_{fi},\ldots,x_{li}$
→ **tuple**

Example:

- Vector = Record
  n: Integer
  els: array[1..10] of TElement
  end

More notation conventions

- pointers:

p: ^DT declaration of a pointer **p**
to a data type DT

p: ↑DT

[p] access to value pointed by pointer **p**

Conventions: special note on pointers

Mechanism for working with pointers and dynamic variable

Pointer :

domain of values: {valid variable addresses} ∪ {NIL}

special value: NIL or NULL – no valid addresses

- pointer type specification: Ex: **p**:^Integer
- dynamic variable allocation Ex: ***new***(**p**)
  new(**p[20]**)
- dynamic variable deallocation Ex.: ***free***(**p**)
- value pointed by a pointer Ex.: [**p**]

Abstraction

Abstraction layers / levels in cs

Computer science commonly presents levels (layers) of abstraction;
each level represents a different model of the same information and processes

DT (in programming)

a particular type of information ← classification
A type denotes a domain (a set of values) and operations on those values.

Concrete data types

= available data types, ready to use !
The available data types vary from one programming language to another, but there are some that usually exist in one form or another.

ADT

ADT (abstract data type): = domain (of values)
and operations (specification)

abstraction = a way of hiding the implementation details

- data abstraction: domain of values; no indication about representation
- procedural abstraction: operation specification; no indication of how achieved
  mathematical constraints on the effects (and possibly cost)

Implementation

for an ADT => (concrete) DT

- can be implemented in many ways and in many programming languages
- ADTs are often implemented as classes:
  class interface declares procedures that correspond to the ADT operations.
  ➔ **Encapsulation**
  → **information hiding strategy**

Advantages

A major goal of software engineering: write reusable code!

Reduced complexity easy to learn and use

Abstraction provides a promise that any implementation of the ADT has certain properties and abilities.This is required in order to use the ADT (implementation).
The user does not need any technical knowledge of how the implementation works to use the ADT. In this way, the implementation may be complex but will be encapsulated in a simple interface when it is actually used.

Flexibility

Different implementations of an ADT are equivalent and may be used somewhat interchangeably in code that uses the ADT.

Localization of change

Code that uses an ADT object will not need to be edited if the implementation of the ADT is changed.

Efficiency

Different implementations of an ADT may be more efficient in different situations; it is possible to use each in the situation where they are preferable, thus increasing overall efficiency.

Data structure

Data structure: is a description of how data are organized.
For any data type, in a computer we will have a representation.

- DS describe a possible representation for a domain

Two ways to think about a DS

- **logic** DS:
  specification of how informations of different types are grouped together
  - elements that form the DS and their relations
  - based on other DT and using ***DS builders***
- **physical representation: the way data is stored in computer memory**

data structure classification (based on physical representation)

- static:
  during execution, it occupies a fixed zone of memory
- semi-static:
  the memory area where the elements of the structure are stored – is fixed
  elements can change their place
- dynamic:
  occupies a memory zone that changes during program execution,
  according to the needs of dimension change

ADT and DS

For an ADT	in programs, we have:
- domain:	➔ data structure (representation)
- operations:	➔ algorithms for operations

Remark:
For a given ADT we can consider more than one DS to implement it. Each data structure has specific benefits as well as its drawbacks.

ADT	DS
Fixed-sized	static
Variable-sized	dynamic

ADT, DS and levels of abstraction

In practice,
we can have different levels of logic specification of structure used for storing data
(similar with stepwise refinement for algorithms)

Example:

ADT domain	collection of elements of type TE
DS:	
refinement 0	list of elements of type TE
refinement 1	singly linked list of elements of type TE
refinement 2	singly linked list with dynamically allocated nodes containing elements of type TE

Data structure: the lowest level of logic data structure specification
Convention: DS - described by using basic types and DS builders (see conventions)

Remarks
1. Refinements involves (permits) some levels of abstraction. For example, we can study the: "single linked list" regardless of the fact that it is dynamically allocated or it has a semi-static representation.
2. The fundamental building blocks for data structures are provided by languages. (arrays, records, pointers/references)

Why DS and ADT ?

Algorithm efficiency strongly depends on the chosen DS for the concrete problem.

> *Selecting the most appropriate data structure to store your application's data is extremely important. Your choice of data structure affects the operation and performance of your application -- sometimes with little consequence, sometimes dramatically.*
> *http://java.sun.com/docs/books/performance/1st_edition/html/JPAlgorithms.fm.html*

Understanding **how to choose** the **best** algorithm or **data structure** for a particular task is one of the keys to writing high-performance software.

Basic concepts – element type

EqualityComparable

A type is EqualityComparable if objects of that type can be compared for equality

We denote this type: TE

- Element Type that can be compared for equality
- has a default value;

Alternative name: TElement

Notation: ε_{TE} – the default element from the set

TE as an ADT:

$\mathcal{D}_{TE}$= {e | e –element}

operations:

Subalg. assignment(e1,e2)

Desc.: assign value from one element to another

Prec.: e1 $\in \mathcal{D}_{TE}$ *here "$\in$" and "=" are the math. signs*

Post.: e2 = e1 *with mathematical significance*

Funct. isEqual (e1,e2)

Desc.: test if that 2 values are equal

Prec.: e1 $\in \mathcal{D}_{TE}$, e2 $\in \mathcal{D}_{TE}$

Post.: isEqual = true if e1=e2

false if e1<>e2

Subalg. initDefault(e)

Desc.: initialize with the default element

Prec.: -

Post.: e= ε_{TE}

Notations:

assignment	operator:	:= or ←
equalityTest	operator:	=

Remarks:

- we consider TE a general (abstract) Type (that can have specialization)
- basic data types are a kind of TE
- Specification of the type of an element:

  e $\in \mathcal{D}_{TE}$

  e: TE

  We can read/print an element by using *read*/*print (*pseudocode*)*

LessThanComparable

A type is LessThanComparable if it is ordered.

- it must be possible to compare two elements of that type
  by using operators < , = and <
  - <, > - is a partial ordering relation
  - = is a equivalence relation
  - any 2 elements are in one of the 3 relations: <,=,>

Example:

<, =, > - mathematical relations for numbers

<, =, > - given by the "dictionary" order of strings (of chars)

Remark:

Only `operator <=` *is fundamental;*
the other inequality operators are essentially syntactic sugar.

We denote this type: TCE (Type: Comparable Element)
It is a kind of TE, that is it has all operations of TE and *some* more
Alternative name: TLessThanComparable

TCE as an ADT:

$\mathcal{D}_{TCE}$= {e | e – less than comparable element}

operations:

@all operations of TE

Funct. compare(e1,e2)

Desc.: compare 2 elements

Prec.: e1 $\in \mathcal{D}_{TE}$, e2 $\in \mathcal{D}_{TE}$

Post.: compare =		
	-1	if e1 < e2
	0	if e1 = e2
	1	if e1 > e2

Note:

sometimes, we will use operators: <,=,>,<=, >=
instead of calling function *compare*

Container

A Container
is an object that stores a finite number of other objects

- its *elements*
- *owns them*

methods for accessing its elements

- usually use ***iterator***

  0, 1 or more than one iterator may be active at any one time

In general:

no guarantee that the elements are stored in any definite order
the order might be different upon each iteration

- Empty container : $\varnothing_{\text{Container}}$

ADT …

Container

size the number of elements it contains ADT

area the total number of bytes that it occupies DS

(memory used)

the sum of the elements' areas plus *whatever* overhead

A variable sized container

insert and/or remove elements

A fixed size container constant size throughout the container's lifetime

(*In some fixed-size container types, the size is determined at compile time.*)

- static
- *semi-static*
- dynamic

Other classification

- homogeneous container *elements of same type (vector)*
- heterogeneous container *elements with different types (tuple)*

Operations

Think about properties of data type you model !

For containers, we have to consider:

- create / initialize, destroy *(create empty)*

  (*copy*, *assignment*)
- get

  set: controlled access to all relevant components
- test some properties
- conversion to/from other types

… (other) specific operations

NO: read/write (Why? Argue!)

Set

- no duplicate elements
- no order is guaranteed

In Mathematics: are unchanging
In computer science:
can change over time (grow, shrink)

Small examples:

- {a, b} = {b, a}
- {a} **no**: {a, a}

Bag

- allow duplicate elements
- no order is guaranteed

Small examples:

- {a, b} = {b, a}
- {a} <> {a, a}

Terminology:	bag	(*Smalltalk*)
	multiset	(*C++ STL)*
	collection	*(Java.util)*

(Linear) List

- elements are arranged in a strict linear order

Terminology: **Sequence containers** (C++ STL)
List (Java.util)

List as ADT: Uniform formal approach

Position – give the position of elements in list

- open to many possible instantiations of Position

ADT List: two parameters

- (i) TE - the type of the constitutive elements
- (ii) Position - the type of elements' positions

specialized ADTs List are obtained by instantiation of the type Position
IndexedList, *SinglyLinkedList*, *DoublyLinkedList*.

What is a Vector?

Stack

container

insertions/extractions are made following a fixed (predefined) strategy:

LIFO: Last In First Out

Remarks:

- **iteration** is ***not specific*** to stack
  stack with iteration → **extension**
  by default: work with stack that only have specific operation

---

True about stack (formal axioms)

- newly created stack is empty;
- after pushing an item to a newly created stack, it becomes nonempty;
- peek returns the most recently pushed item;
- stack remains untouched, after a pair of push and pop commands, executed one after another and with the same element.

Stack

Operations

- init // create , createEmpty, initEmpty
- destroy
- push
- pop
- isEmpty //empty
- peek
- isFull

Queue

- a container
- in which insertions/extractions are made following a fixed (predefined) strategy

  FIFO: First In First Out

Remarks:

- **iteration** is ***not specific*** to stack

  queue with iteration → **extension**

  by default: work with queue that only have specific operation

Queue

Operations

- init // create , createEmpty, initEmpty
- destroy
- enqueue // push, push_back , add, insert
- dequeue // pop, pop_front, extract,remove

- isEmpty //empty

- isFull
- peek

Deque

double ended queue

- insertions/extractions can be made ***from both ends*** **(LIFO + FIFO)**

… operations

Deque – name for operations

Operation	Ada	C++ STL	Java.util
insert at back	append	***push_back***	offerLast
insert at front	appendleft	***push_front***	offerFirst
remove last	pop	***pop_back***	pollLast
remove first	popleft	***pop_front***	pollFirst
examine last		back	***peekLast***
examine first		front	***peekFirst***

Map & Multimap

Elements:

key

value (mapped value)

- keys are not unique
- there is no limit on the number of elements with the same key

Other terms:

- associative array, dictionary

Unique associative container

Multiple associative container

no order is guaranteed

Map

- **operations**
  specific: based on keys

add …, k; v // … *put, reassign*
remove …; k
  // → value

search…, k // *get, searchByKey*
  // → value
  containsKey
  // → boolean
---
  containsValue
not usual : searchByValue(m,v) // → key;

Map

- **other operations**

create
destroy

isEmpty , size

keys // *key*Set
values // *valueMultiset*
pairs

getIterator // iterator