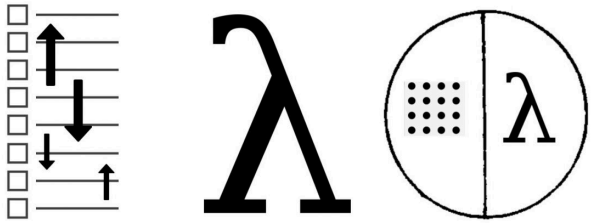
Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Wrocławska

Unia Europejska
Europejski Fundusz Społeczny

„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”



Programming paradigms

L07: Abstract data types

by Michał Szczepanik

Type vs data type

A type is a collection of values. For example, the Boolean type consists of the values true and false. The integers also form a type. An integer is a simple type because its values contain no subparts. A bank account record will typically contain several pieces of information such as name, address, account number, and account balance. Such a record is an example of an aggregate type or composite type. A data item is a piece of information or a record whose value is drawn from a type. A data item is said to be a member of a type.

A data type is a type together with a collection of operations to manipulate the type. For example, an integer variable is a member of the integer data type. Addition is an example of an operation on the integer data type.

Abstract data types

Abstract data types are mathematical models of a set of data values or information that share similar behavior or qualities and that can be specified and identified independent of specific implementations.

Abstract data types, or ADTs, are typically used in algorithms. An abstract data type is defined in term of its data items or its associated operations rather than by its implementation.

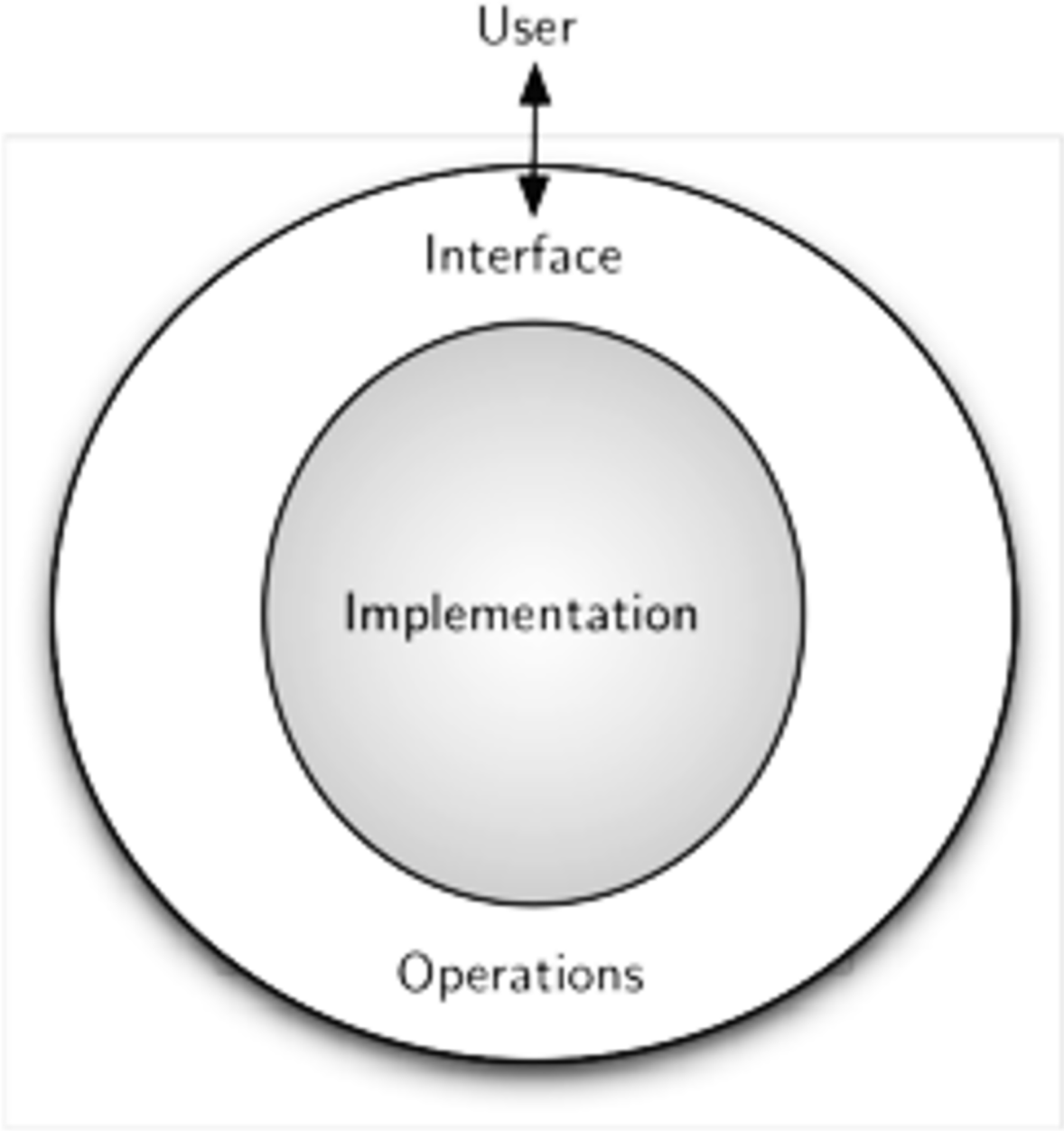
According to the NIST definition for abstract data types, an abstract data structure or type "is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations."

Abstract data types

An abstract data type (ADT) is the specification of a data type within some language, independent of an implementation. The interface for the ADT is defined in terms of a type and a set of operations on that type. The behavior of each operation is determined by its inputs and outputs. An ADT does not specify how the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as encapsulation.

A data structure is the implementation for an ADT. In an object-oriented language, an ADT and its implementation together make up a class. Each operation associated with the ADT is implemented by a member function or method. The variables that define the space required by a data item are referred to as data members. An object is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.

Abstract data types



ADT representation

ADTs are defined by an interface:

- Simplicity. Hiding the internal representation from the client means that there are fewer details for the client to understand.
- Flexibility. Because an ADT is defined in terms of its behavior, the lib programmer who implements one is free to change its underlying representation.
- Security. The interface boundary acts as a wall that protects the implementation and the client from each other. If a client program has access to the representation, it can change the values in the underlying data structure in unexpected ways

Example

The mathematical concept of an integer, along with operations that manipulate integers, form a data type. The int variable type is a physical representation of the abstract integer. The int variable type, along with the operations that act on an int variable, form an ADT. Unfortunately, the int implementation is not completely true to the abstract integer, as there are limitations on the range of values an int variable can store. If these limitations prove unacceptable, then some other representation for the ADT "integer" must be devised, and a new implementation must be used for the associated operations.

Example

An ADT for a list of integers might specify the following operations:

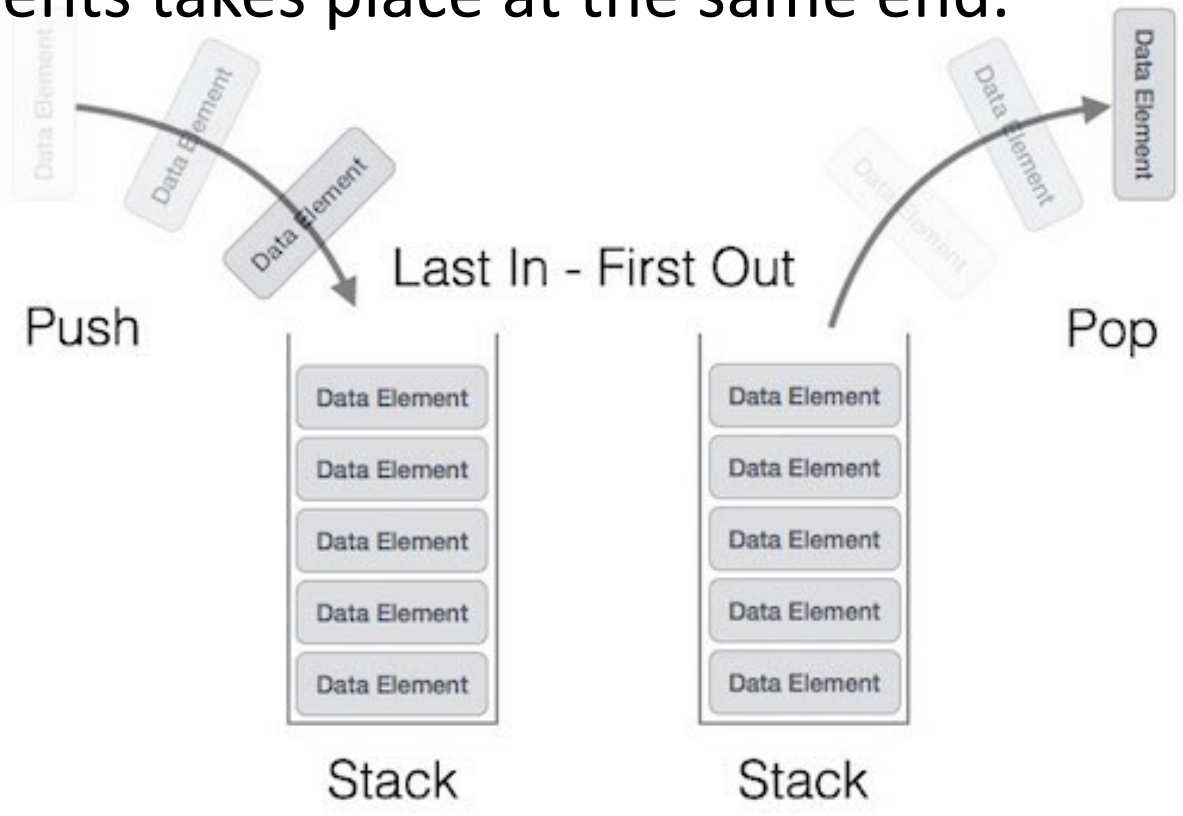
- Insert a new integer at a particular position in the list.
- Return True if the list is empty.
- Reinitialize the list.
- Return the number of integers currently in the list.
- Retrieve the integer at a particular position in the list.
- Delete the integer at a particular position in the list.

From this description, the input and output of each operation should be clear, but the implementation for lists has not been specified

Stack

One of the simplest abstract data types is the stack.

The distinguishing characteristic of a stack is that the insertion or removal of elements takes place at the same end.



Stack

One of the simplest abstract data types is the stack. The operations new(), push(v, S), top(S), and popOff(S) may be defined with axiomatic semantics as following:

- new() returns a stack
- popOff(push(v, S)) = S
- top(push(v, S)) = v

where S is a stack and v is a value.

From these axioms, one may define equality between stacks, define a pop function which returns the top value in a non-empty stack, etc. For instance, the predicate isEmpty(S) may be added and defined with the following additional axioms:

- isEmpty(new()) = true
- isEmpty(push(v, S)) = false

Stack - implementation

A Stack can be implemented in several ways, some implementations use an Array and store the top reference to manipulate the stack of elements. It’s also possible to use a LinkedList taking advantage of the insertion and removal from the beginning is an O(1) operation.

We can achieve the O(1) either with an Array or LinkedList to the push and pop operations.

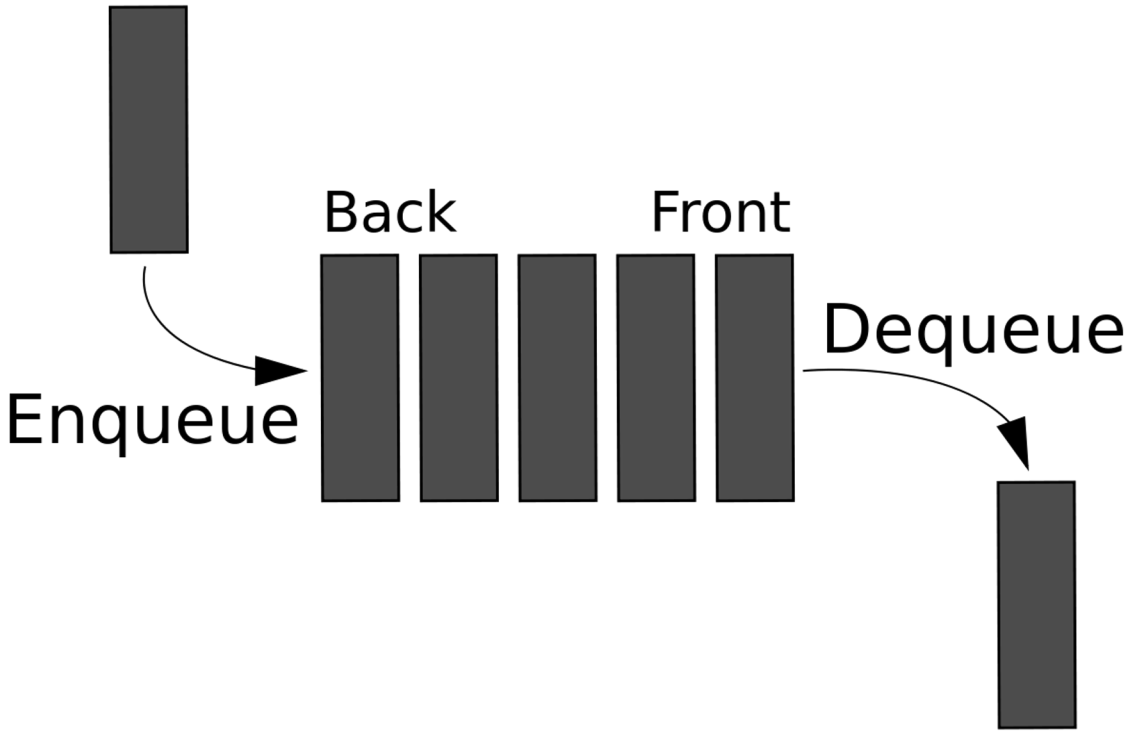
Queue

A queue is a collection of entities that are maintained in a sequence and can be modified by the addition of entities at one end of the sequence and removal from the other end of the sequence. By convention, the end of the sequence at which elements are added is called the back, tail, or rear of the queue and the end at which elements are removed is called the head or front of the queue.

Queue

Operations:

- isFull(), This is used to check whether queue is full or not
- isEmpry(), This is used to check whether queue is empty or not
- insert(x), This is used to add x into the queue at the rear end
- delete(), This is used to delete one element from the front end of the queue
- size(), this function is used to get number of elements present into the queue



Queue - implementation

There are several efficient implementations of FIFO queues:

- A doubly linked list has O(1) insertion and deletion at both ends, so it is a natural choice for queues.
- A deque implemented using a modified dynamic array

List

A list or sequence is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a tuple or finite sequence; the (potentially) infinite analog of a list is a stream

List

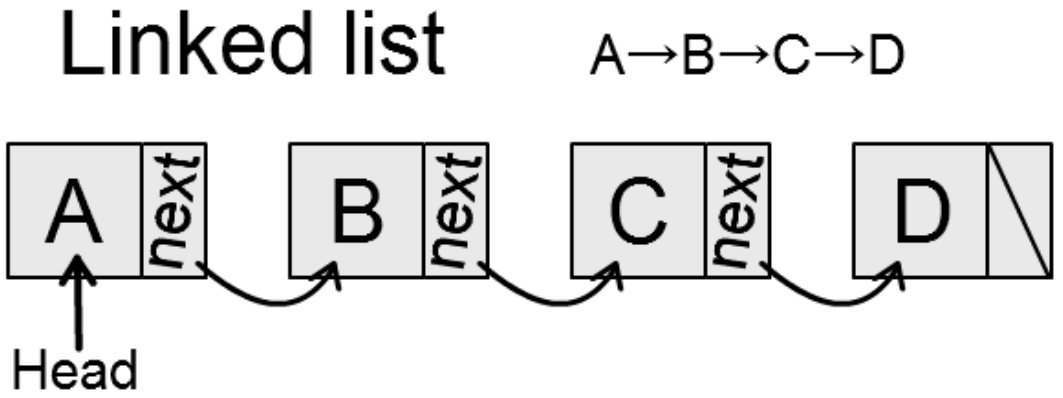
Operations:

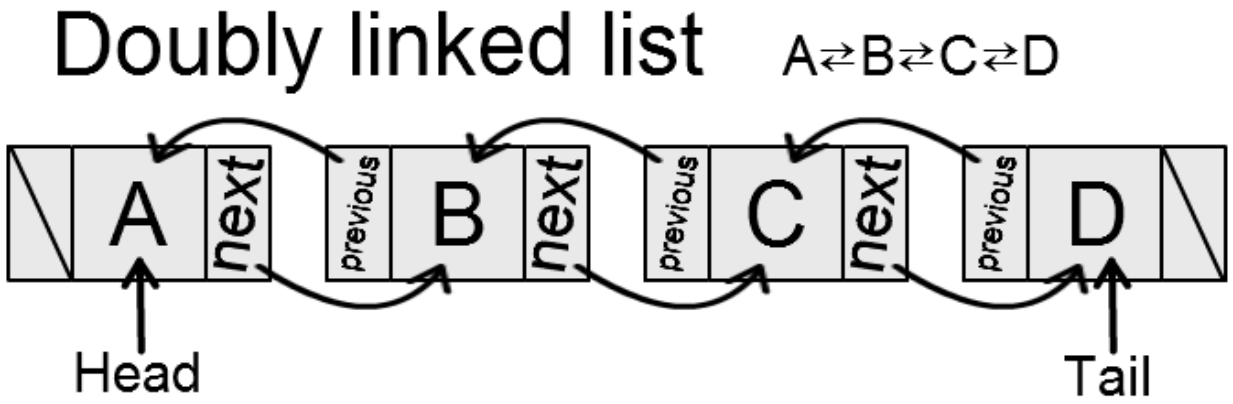
- size() – this function is used to get number of elements present into the list
- insert(x) – this function is used to insert one element into the list
- remove(x) – this function is used to remove given element from the list
- get(i) – this function is used to get element at position i
- replace(x, y) – this function is used to replace x with y value

List - implementation

Lists are typically implemented as:

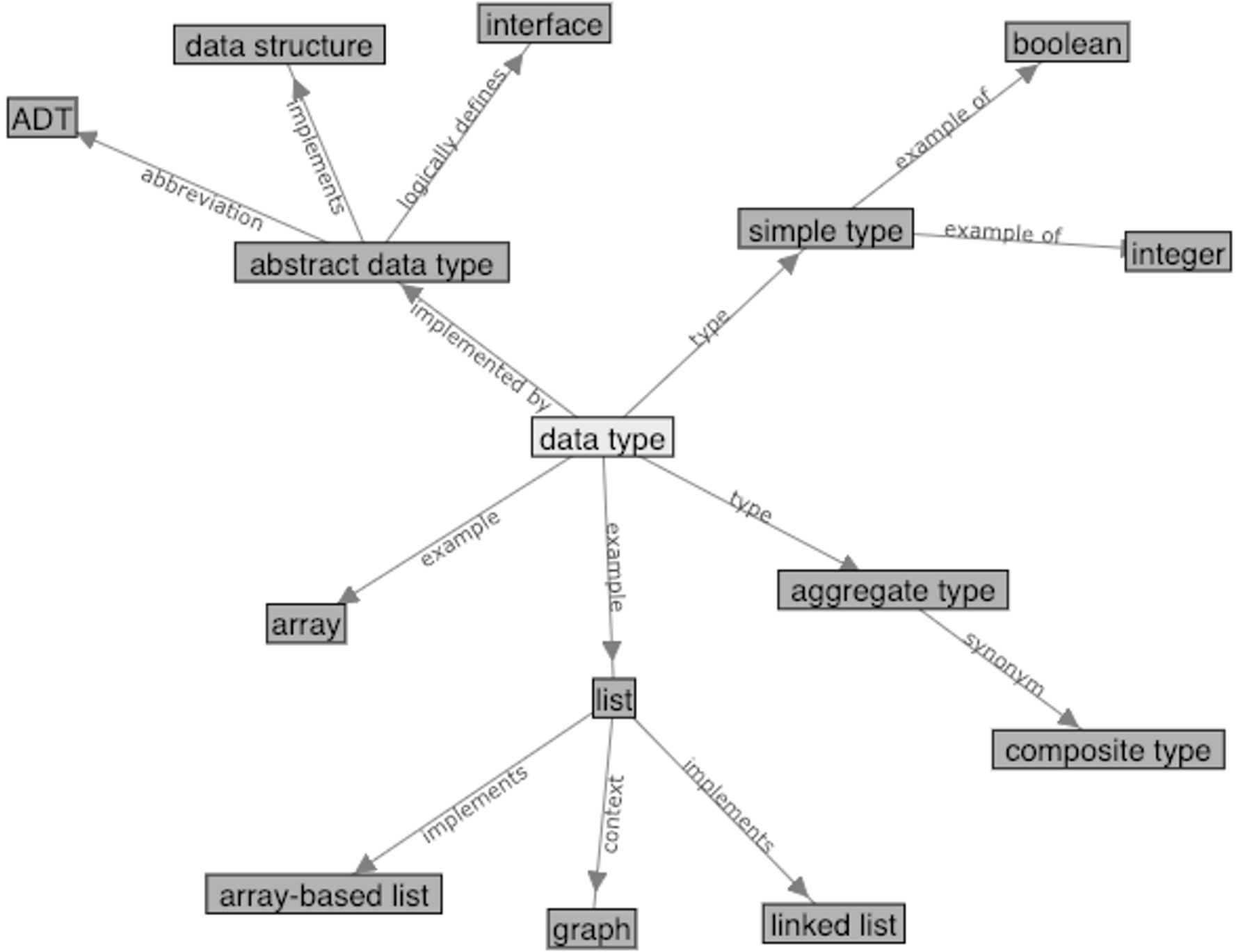
- linked lists (either singly or doubly linked),
- arrays, usually variable length or dynamic arrays.





Other types of ADT

- Container
- Set
- Map
- Graph
- Tree


data structure
interface
boolean
ADT
implements
logically defines
example of
abbreviation
simple type
example of
integer
abstract data type
implemented by
type
data type
example
example
type
array
aggregate type
synonym
list
composite type
implements
context
implements
array-based list
graph
linked list

Thank you for your attention