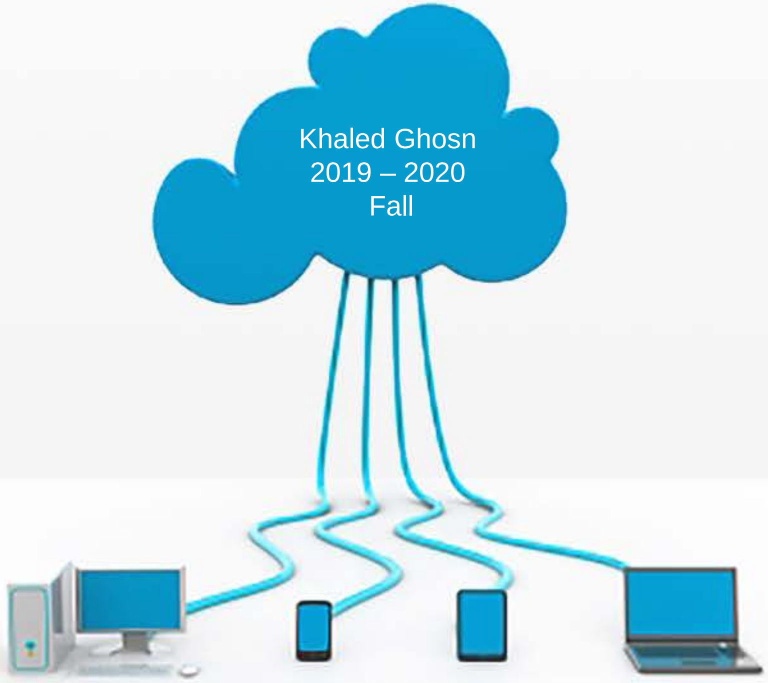
ARTS, SCIENCES & TECHNOLOGY UNIVERSITY IN LEBANON

AUL

Abstract Data Types & Lists

ADT, Lists, Array-List



Abstract Data Type

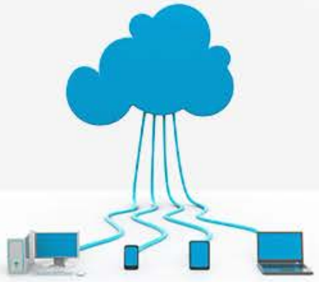
ADT

An Abstract Data Type (ADT) is a data structure with a set of operations

- Operations specify how the ADT behaves, but does not reveal how they are implemented
- In many cases, there are more than one way to implement an ADT

Examples of ADTs:

- Lists
- Stacks
- Queues
- Trees & Search Trees
- Hash Tables
- …



Abstract Data Type

ADT

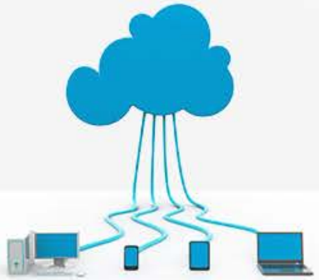
Separate the notions of:

- ✓ *Specification*

  what kind of thing we're working with, and what operations can be performed on it

- ✓ *Implementation*

  how the thing and its operations are actually implemented



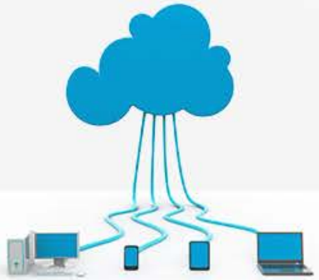
Abstract Data Type

Benefits of using ADTs

- Code is easier to understand and modify

- Implementations of ADTs can be changed (e.g., for efficiency) without requiring changes to the program that uses the ADTs

- ADTs can be reused in future programs.



Abstract Data Type

ADT example

String:

Spec:

assignment to “characters”, length, concatenation,

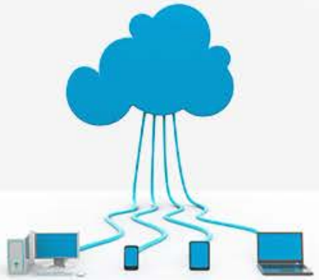
get-character-at-position-i, …

Implementation:

ordered sequence of elements of type char

or int (i.e. their ASCII)…

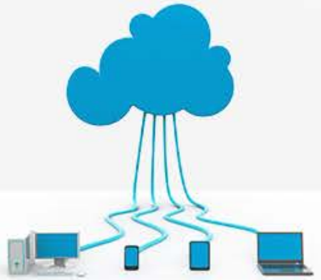
the sequence could be an array, hashed value, linked list, …



Abstract Data Type

ADT in Practice

- Each ADT is a class
- Specs' operations => public methods
- Implementation is isolated with private attributes inaccessible to the outside.



Abstract Data Type

Contents

Public or **External** part consists of:

- The conceptual picture

  the user's view of what the object looks like, how the structure is organized

- The conceptual operations

  what the user can do to the ADT

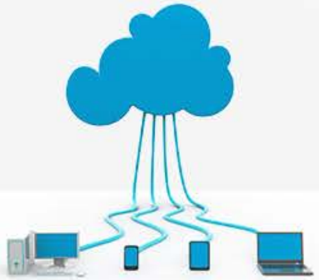
Private or **Internal** part consists of:

- The representation

  how the structure is actually stored => private attributes

- The implementation of the operations

  the actual code

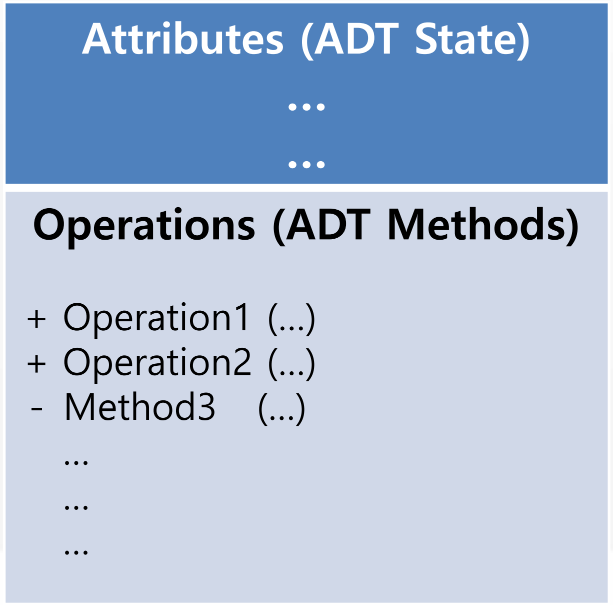


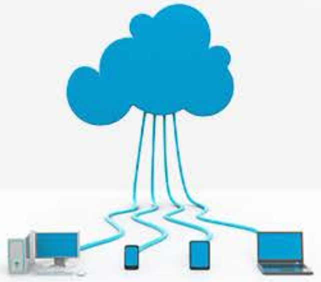
Abstract Data Type

Operations

In general, there are many possible operations that could be defined for each ADT; however, they often fall into these categories:

1. initialize
2. add data
3. access data
4. remove data





Lists

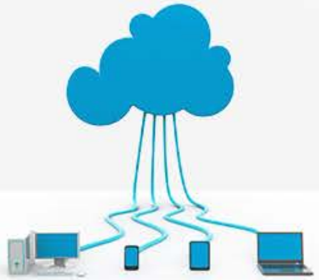
Definition

What is a list?

- An **ordered sequence** of elements
  - collection of items such as: A1, A2, ..., AN
- Elements may be of arbitrary, but the same type (i.e., all ints, all doubles etc.)
- Each element is accessible by a 0-based **index**
- A list has a **size** (number of elements that have been added)
- Elements can be added to the front, back, or elsewhere

E.g.:

- 2, 6, 1, 2, 3
- Days of Week = (S, M, T, W, Th, F, Sa)
- Months = (Jan, Feb, Mar, Apr, … , Nov, Dec)

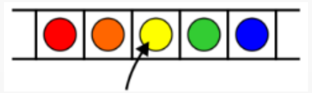


Lists

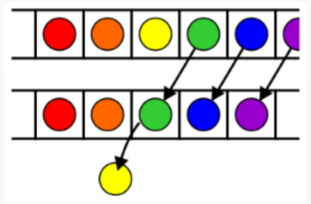
Operations

Operations at the k^{th} entry of the list include:

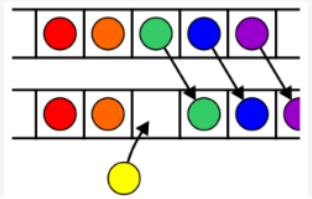
Access to the object



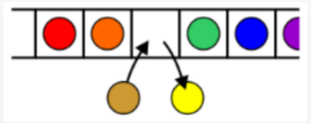
Erasing an object

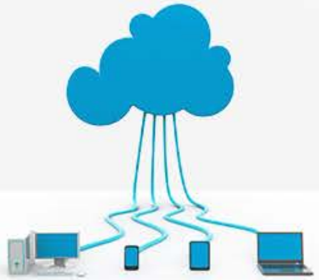


Insertion of a new object



Replacement of the object





Lists

Operations

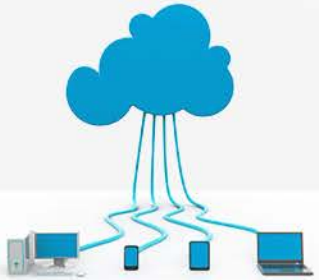
We may always find additional operations

- Too much operations => confusion

However, we keep on a minimum, complete and relevant set of operations

Not easy to achieve:

- ADTs are simply updated and extended in certain cases



Lists

Operations

Operation	Description
List ( )	create an empty list (constructor)
void *Add* (item)	add an element to the end of the list
void *Add* (item, index)	add an element "item" at position "index" in the list, moving the i tems originally in positions one place to the right to make room - error if "index" is less than 0 or greater than size( )
boolean *Contains* (item)	return true if "item" is in the list
int *Size* ( )	return the number of items in the list
boolean *isEmpty* ( )	return true if the list is empty
object *Get* (index)	return the item at position "index" in the list error if "index" is less than 0 or greater than or equal to size( )
object *Remove* (index)	remove and return the item at position "index" in the list, moving the items originally in positions "index+1" through size( ) one p lace to the left to fill in the gap - error if "index" less than 0 or greater than or equal to size()

Lists

List example: Abstract Strings

A specialization of an Abstract List is an Abstract String:

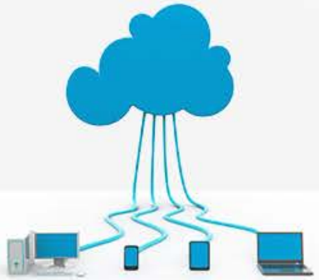
- The entries are restricted to *characters* from a finite *alphabet*
- This includes regular strings “Hello world!”

The restriction using an alphabet emphasizes specific operations that would seldom be used otherwise

- Substrings, matching substrings, string concatenations

It also allows more efficient implementations

- String searching/matching algorithms
- Regular expressions



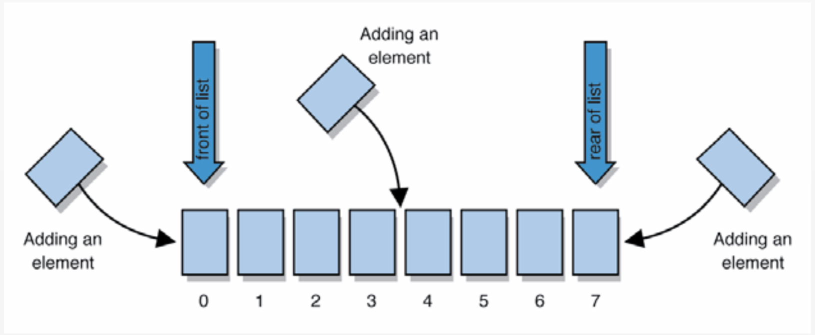
Lists

Implementations

Two types of implementation:

- **Array-Based List**
- **Linked List**

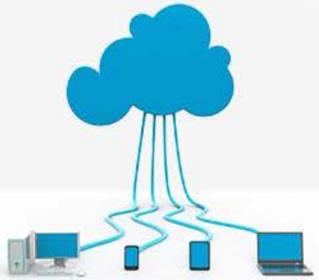
We will compare worst case running time of ADT operations with different implementations



Array-List

Definition

- An Array-List is an implementation of List, backed by an array
- An Array-List is a **dynamic** data structure, meaning items can be added and removed from the list
- An Array-List is implemented as a resizable array
  - The Array-List instance has a capacity, which is the size of the array used to store the elements in the list
  - As more elements are added to Array List, its size is increased dynamically
- It's elements can be accessed directly by using the get and set methods, since it is essentially an array.



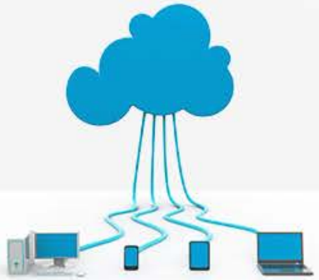
Array-List

Disadvantages

- One disadvantage of using **arrays** to store data is that arrays are **static structures** and therefore cannot be easily extended or reduced to fit the data set

- Another drawback of arrays is that if you delete an element from the middle and want no holes in your array (e.g. (1, 2, 4, null) instead of (1, 2, null, 4)), you will need to shift everything after the deleted element

- Arrays are also expensive to maintain new insertions and deletions

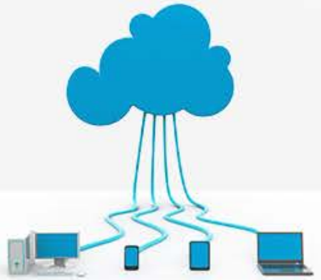


Array-List

Exercise 1

Write the following methods:

1. **void makeEmpty ( )**
2. **boolean isSorted ( )**
3. **object RemoveFrom (int index)**
4. **int IndexOf (object)**

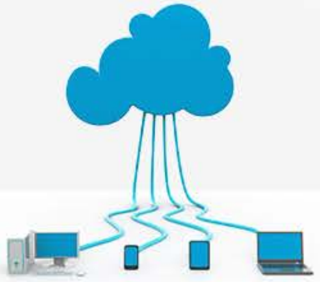


Array-List

Exercise 2

For each of the following **Array-List operations**, calculate the rate of growth in terms of **Big-*Oh*** notation

1. **Add**
2. **Remove**
3. **getElement**
4. **RemoveFrom**
5. **Print**
6. **Contains**
7. **AddTo**



Array-List

Challenging Exercise

Solve exercise P-7.58 page 304 in the textbook