ICOM 4035 – Data Structures

[Lecture 1 - Introduction 3](#_Toc61164677)

[ Part I) Introduction 3](#_Toc61164678)

[ Abstract Data Type 3](#_Toc61164679)

[ Data Structure 3](#_Toc61164680)

[ Part II) Vector Class Creation 3](#_Toc61164681)

[ Part III) Vector Class Operations 3](#_Toc61164682)

[ Process to write code 3](#_Toc61164683)

[Lecture 2 – Abstract Factories 4](#_Toc61164684)

[ Part I) Multiple implementations for ADT 4](#_Toc61164685)

[ Part II) Using Interfaces to specify ADT 4](#_Toc61164686)

[ Part III) Using Abstract Factories to create instances 4](#_Toc61164687)

[ Benefit 5](#_Toc61164688)

[Lecture 3 - Bags 6](#_Toc61164689)

[ Part I) Role of collection classes and Bags 6](#_Toc61164690)

[ Collection 6](#_Toc61164691)

[ Bag / Multi-set 6](#_Toc61164692)

[ Part II) Design and implementation of the Bag ADT 6](#_Toc61164693)

[ Operations of bag 7](#_Toc61164694)

[ Part III) Iterating over a Bag 7](#_Toc61164695)

[ Part IV) Implementing a Dynamic Bag 8](#_Toc61164696)

[ Notes 8](#_Toc61164697)

[Lecture 4 - Sets 9](#_Toc61164698)

[ Part I) Sets using Java Generics in collections 9](#_Toc61164699)

[ Set 9](#_Toc61164700)

[ Example of applications 9](#_Toc61164701)

[ Part II) Design and Implementation of the Set ADT 9](#_Toc61164702)

[ Part III) Iterating over a Set 11](#_Toc61164703)

[ Part IV) Implementing a Dynamic Set 11](#_Toc61164704)

[ Object composition 11](#_Toc61164705)

[Lecture 5 – List ADT 12](#_Toc61164706)

[ Part I) Introduction 12](#_Toc61164707)

[1) List / Finite sequence 12](#_Toc61164708)

[ Part II) Design and Implementation of the List ADT using ArrayList DS 12](#_Toc61164709)

[ Part III) Iterating over a List 13](#_Toc61164710)

[Lecture 7 – Linked Lists 14](#_Toc61164711)

[ Part I) Introduction to the Linked List Data Structure 14](#_Toc61164712)

[ Part II) Design and implementation of Linked List Data Structure 14](#_Toc61164713)

[ Part III) Iterating over a Linked List 15](#_Toc61164714)

[Lecture 11 – MAP ADT 16](#_Toc61164715)

[ Part I) Introduction to the Map ADT 16](#_Toc61164716)

[ Part II) Design and implementation of a Map using Linked List 16](#_Toc61164717)

[Lecture 12 – Hashtables & Map ADT 18](#_Toc61164718)

[ Part I) Introduction to the Hash Table and its use for implementing a Map 18](#_Toc61164719)

[ Part II) Design and Implementation of a Hash Table using Separate Chaining 18](#_Toc61164720)

[ Part III) Design and Implementation of a Hash Table using Open Addressing 18](#_Toc61164721)

[Summary of Data Structures 19](#_Toc61164722)

[1) Bag / Multi-Set 19](#_Toc61164723)

[2) Set 19](#_Toc61164724)

[3) List / Finite SEQUENCE 20](#_Toc61164725)

[4) Stack 21](#_Toc61164726)

[5) Queue 21](#_Toc61164727)

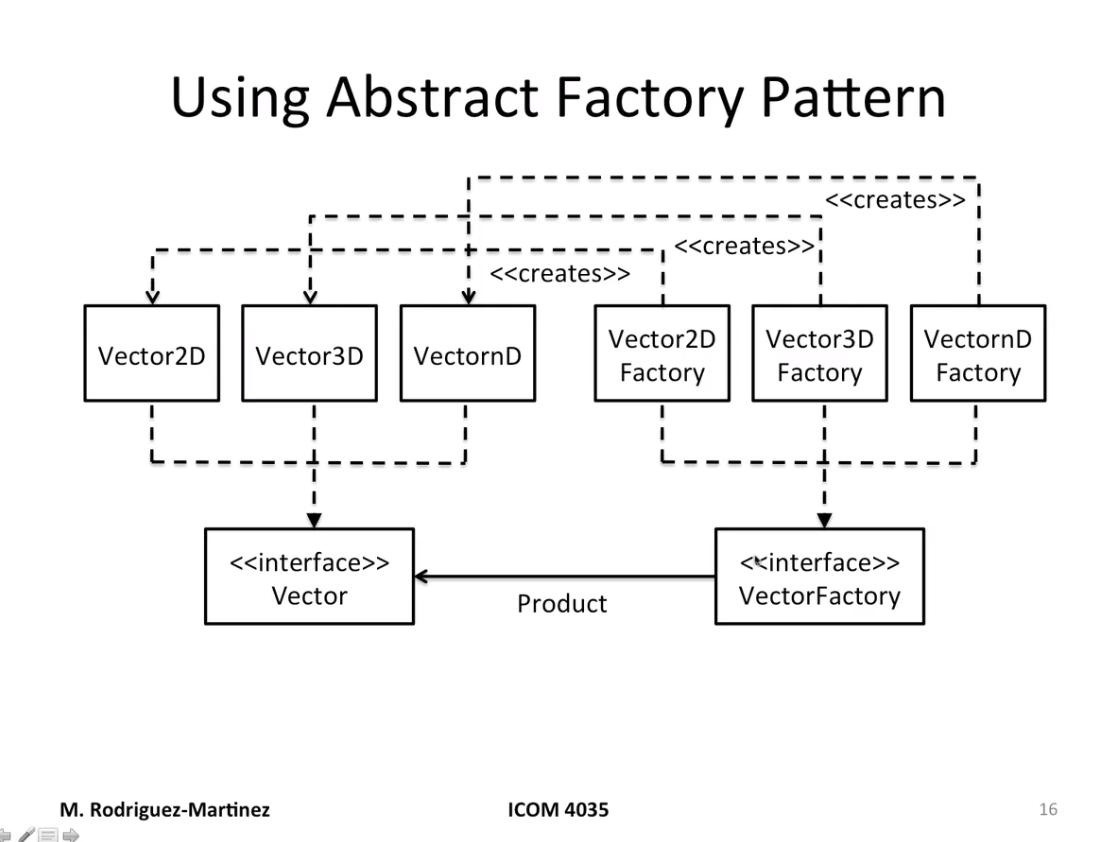
[6) Map 22](#_Toc61164728)

# Lecture 1 - Introduction

* Part I) Introduction
  + Abstract Data Type (ADT) = Mathematical Model that represents data
    1. Parts of an ADT
       - **Values** = data to be stored
         * Represents current state in model
       - **Operations** = actions that will be taken
         * Modifies the current state in model
         * Produces new information based on these values
  + Data Structure (DS) = Implementation of ADT using constructs of programming language
    1. Parts of a DS
       - **Variables and data types** – they provide storage for ADT values
       - **Methods** – functions that will implement instructions associated with operations for each ADT
    2. \*1 ADT can be implemented with multiple types DS
  + Example: 2-D Vector ADT
    1. **Values**
       - X coordinate (real number)
       - Y coordinate (real number)
    2. **Operations**
       - *Creation* – build a new vector at (a,b)
       - *Mutator* – (re-assign vector coordinates)
         * However, it’s better practice to keep the values immutable to prevent unexpected behavior
       - *Accessor* – (inspects the values of coordinates)
       - *Externalizer* – (convert to a string of characters)
       - *Mathematical Operations*
         * Magnitude
         * Sum
         * Substract
         * Dot Product
* Part II) Vector Class Creation
  + Values of implementation should be private since accessing and changing them should be the accessor’s and mutator’s job
* Part III) Vector Class Operations
  + Process to write code
    1. Define ADT values and operations
    2. Pick Data Structure to implement ADT
       - Chose basic types or other data structures to store values
       - Write member methods to implement operations
    3. Revise, test, revise, test

# Lecture 2 – Abstract Factories

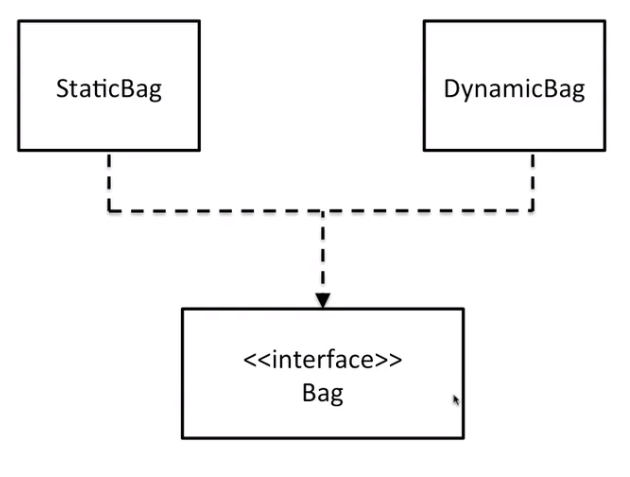
* Part I) Multiple implementations for ADT
  + Even though we implemented a 2D vector class, we cannot use 3D vectors or N dimensional vectors
    - This shows how an ADT can be implemented multiple ways
* Part II) Using Interfaces to specify ADT
  + An interface acts like a prototype that includes all the operations and values a specific implementation will have. However, it does not actually do anything
  + Classes will then implement the interfaces. Therefore, the classes will store the values and perform operations, adding its own peculiarities, restrictions, and methods.
    - 2D vector hast *getX* and *getY* methods, but the 3D vector additionally implements *getZ*
    - The *getCoordinate* method is hardcoded for 2 and 3 dimensions, but not for N dimensions
    - If trying to make operations by mixing types, an error will be thrown
* Part III) Using Abstract Factories to create instances
  + Can be used to control object creation
    - Instead of calling constructor in the code, the Abstract Factory class is in charge of making the new instance of the object
    - We don’t necessarily know which concrete class is used to implement the interface



* + Benefit
    - You can have as many vector implementations as I need
    - Main program might not need to be changed
      * Factory class may read out of a configuration file form command line or properties file
    - Code is based on the Vector interface (only one concept to deal with)
  + It’s basically an interface with a unique method *newInstance* that validates input values for the factory and then returns new object instance if everything turned out alright

# Lecture 3 - Bags

* Part I) Role of collection classes and Bags
  + Collection = Abstract Data Type whose purpose is to store data items for us
    - They should keep object instances organized and ready to use
  + Benefits
    - *Variable size*
      * Size can be dynamically changed without need of recompiling
    - *Implementation can be changed*
      * Arrays are not necessarily required. Maybe Linked Lists, Hash Tables, etc. can be used
    - *Information hiding*
      * Controls access to items without exposing specific details that might reveal vulnerabilities
    - *Reuse* (Don’t Repeat Yourself | DRY)
      * No need to make the same allocation or same operations to find things around. The collection will make sure to access information in an efficient way with the same pattern
  + Bag / Multi-set
    - **Unordered** collection of things
    - Repetitions are allowed
    - They can store whatever data item the application needs
    - Static Bag
      * Maximum size of bag is fixed at build-time
    - Dynamic Bag
      * Maximum size of bag can grow to accommodate elements
    - How do we implement it?
      * Array of type Object
      * ADT Bag is an interface
      * Static Bag or Dynamic bags will implement the Bag interface
* Part II) Design and implementation of the Bag ADT



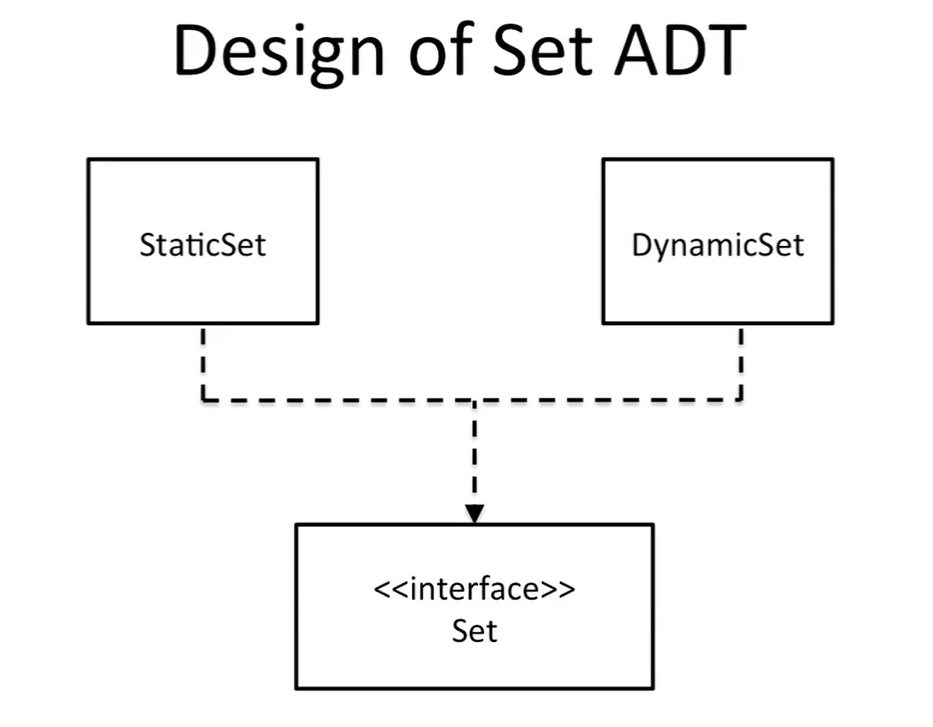
* + Operations of bag (9)
    - Add new element (add)
      * Return true if element was added or false if it could not be added
      * For static bag, we need to check if there is enough space
      * Dynamic bags grow as needed
    - Remove **one** copy of an element (erase)
      * Return true if erased or false if not found
      * If found, remove that specific copy and swap it with the last element (so the currentSize index points to the end of the array again)
        + Making swap is not a problem since the multi-set is not ordered
        + The previously last element should become null to prevent memory leaks
    - Remove **all** copies of an element (eraseAll)
      * for(int result = 0; multiset.erase(value); result++); return(result);
    - Count copies of element(count)
      * Scan array and accumulate each time you see target value
        + In Java the *equals* method must be used to compare objects
    - Clear bag (clear)
      * Loop through array, setting everything to null and setting currentSize to 0
    - Test for element membership (isMember)
      * return(multiset.count(value) > 0);
    - Get Bag size (size)
      * return(currentSize);
    - Test if empty(isEmpty)
      * return(multiset.size == 0);
    - Iterate over all stored values
* Part III) Iterating over a Bag
  + In order to iterate over a bag, we can make the Bag an iterable
    - The Bag interface will extend the Java Iterable interaface
    - Bag can then return an iterator
* Part IV) Implementing a Dynamic Bag
  + Using ArrayList / C++ Vector logic (duplicate size once you run out)
  + Possibly using Linked List
  + Alternatives
    - Copy all methods from static bag, but change the add method
      * NO!
      * Has a lot of space for bugs than can cross over from the static bag
    - Use inheritance to make static bag a parent class for dynamic bag and change add method
      * Better, but you need to make array of elements and currentSize protected instead of private
      * Always aim to have private fields in the class
    - Object composition
      * Static bag is private object instance of dynamic bag, which maps every method to the static’s bag except the add method, which makes a new bag and overwrites the original static bag, making it twice the original size
* Notes
  + References that are unused MUST be set to null
    - Otherwise, unused references will stay there without the garbage collector knowing what to do with it
  + It’s always good practice to write operations in terms of previous ones
    - If there’s a bug somewhere, it’s localized in only one place
  + Factories
    - The factory interface will have 1 method called *newInstance* which takes in any important parameters like maxSize in the case of Bags
    - The classes that implement this interface will simply return a new instance of the specific object the factory was made for

# Lecture 4 - Sets

* Part I) Sets using Java Generics in collections
  + Set
    - Unordered collection of things
    - Repetitions are NOT allowed
    - Can be implemented with
      * *Static*-size Set = Fixed number of items (set at build-time)
      * *Dynamic*-size Set = Variable number of items (size can be increased at run-time)
  + Example of applications
    - Tracking of family members
    - Keeping courses enrolled in a given semester
    - Assign meetings to a room
    - Any application that needs to collect items that cannot be repeated more than once
  + Using arrays of type Object to implement ADTs is unsafe
    - Mixing Strings, Doubles and Integers (for example) can introduce bugs in the program
    - Makes the implementation either less stable
  + It’s better to use Generics (equivalent to C++ Templates)
    - It lets us take in as a parameter the datatype to be stored in the collection
    - **One** implementation works for any type associated with the container
      * Only objects of a particular type can be used
    - In Java
      * E elements[] = (E[]) new Object[arraySize];

1. Make the elements array of type E
2. Typecast an array of Objects of size *arraySize*

* Part II) Design and Implementation of the Set ADT



* + Add new element
    - Return true if element was successfully added, or false if it is already present
    - For a static set, it is necessary to check if there is enough space
    - Dynamic sets need to grow as neededs
    - Steps

1. Search in set to see if element is already present
2. Add element to position *currentSize* and increment it
   * Remove a copy of an element
     + Return true if erased or false if not found
     + Works the same for both static and dynamic sets
     + Steps
3. Search for element in set
4. Replace the element with the last element
5. Decreased *currentSize* while setting the element in the last position to null
   * Test for element membership
     + Determine if an element is inside set
     + Iterate through set until element is found, and return true if found
   * Clear set
     + Iterate though set, turning everything into null and setting *currentSize* to 0
   * Get set size
     + Return *currentSize*
   * Test if empty
     + Check if *currentSize* is 0
   * Compute Union
     + Given two sets, their union is another set which includes all elements of both sets
     + There are no duplicates
     + Steps
6. Copy Set 1 to the union set
7. Iterate over Set 2, adding every element to the union set
   * Compute Difference
     + Enables to check what elements are in Set 1, but not on Set 2
     + Steps
8. Make an empty Set 3
9. Iterate over every element in Set 1. If element is not present in Set 2, add to Set 3
   * Compute Intersections
     + Given two sets, their union is another set which only includes all elements in common
     + There are no duplicates
   * Test if subset
     + Iterate over Set 1 and see if every element is also in S2
   * Iterate over all stored values

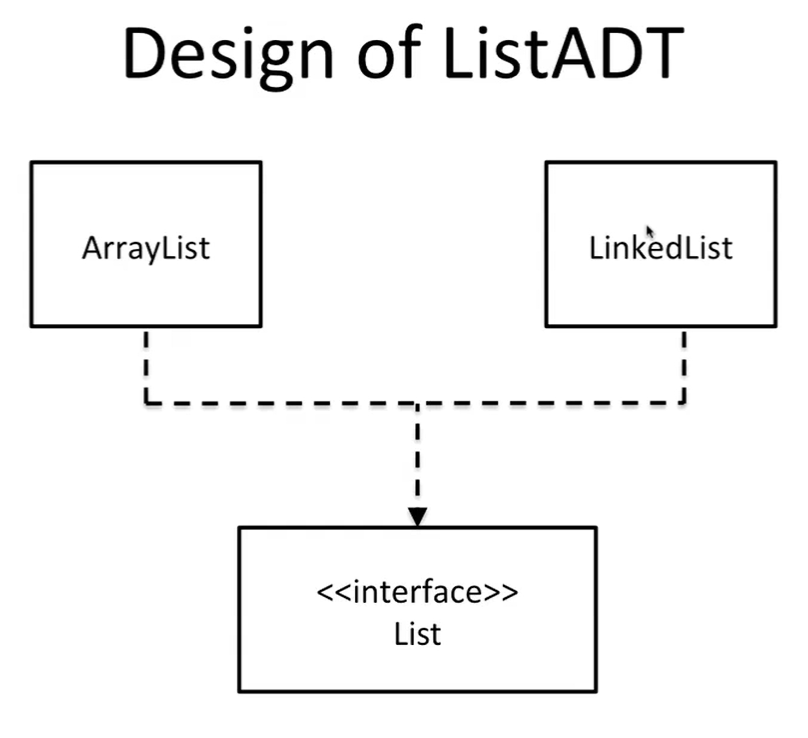
* Part III) Iterating over a Set
* Part IV) Implementing a Dynamic Set
  + Object composition
    - Using another object’s instance and methods to implement your class, tweaking some of the methods to implement your functionality

# Lecture 5 – List ADT

* Part I) Introduction
  + 1. List / Finite sequence
    2. Collection of elements with a notion of an assigned position for each element
    3. It has an order associated with the element’s specific position within the list
    4. They allow repetitions
  + Example Applications
    1. Applications where keeping the order of items is important
    2. Waiting list
    3. To do list
    4. Items in an agenda
    5. Questions in a test
    6. Favorites in a phone number list
  + Variations
    1. Static-size list = Maximum amount of elements list can contain is fixed at build-time
    2. Dynamic-size list = Maximum number of elements is increased during run-time
* Part II) Design and Implementation of the List ADT using ArrayList DS
  + Operations
    1. Add
       - At the end by just indicating the value
       - At a specific index by also indicating the index, followed by the value
    2. Remove
       - Erase first copy of the value OR
       - Erase element at specific position OR
       - All copies of the value
       - Return true or false depending on operation success
    3. Get
       - Returns element at given index
    4. Set
       - Returns old value as it is being replaced
    5. First
       - Returns first element
    6. Last
       - Returns last element
    7. firstIndex
       - Returns the index of the first appearance of a value
       - Return -1 if not found
    8. lastIndex
       - Returns the index of the last appearance of a value
       - Return -1 if not found
    9. Size
       - Returns list size
    10. isEmpty
        - Returns Boolean specifying if list is empty
    11. Contains
        - Returns Boolean if particular element is found
    12. Clear
        - Empty list
  + Él enfatiza tanto el principio DRY, pero no lo aplica mucho aquí
  + He also emphasizes checking for null parameters,
* Part III) Iterating over a List
  + - * List-ordered collection of items
    1. Order based on numeric position within list
    2. ArrayList – dynamic list whose size can be changed as needed (by doubling the size of previous array)

# Lecture 7 – Linked Lists

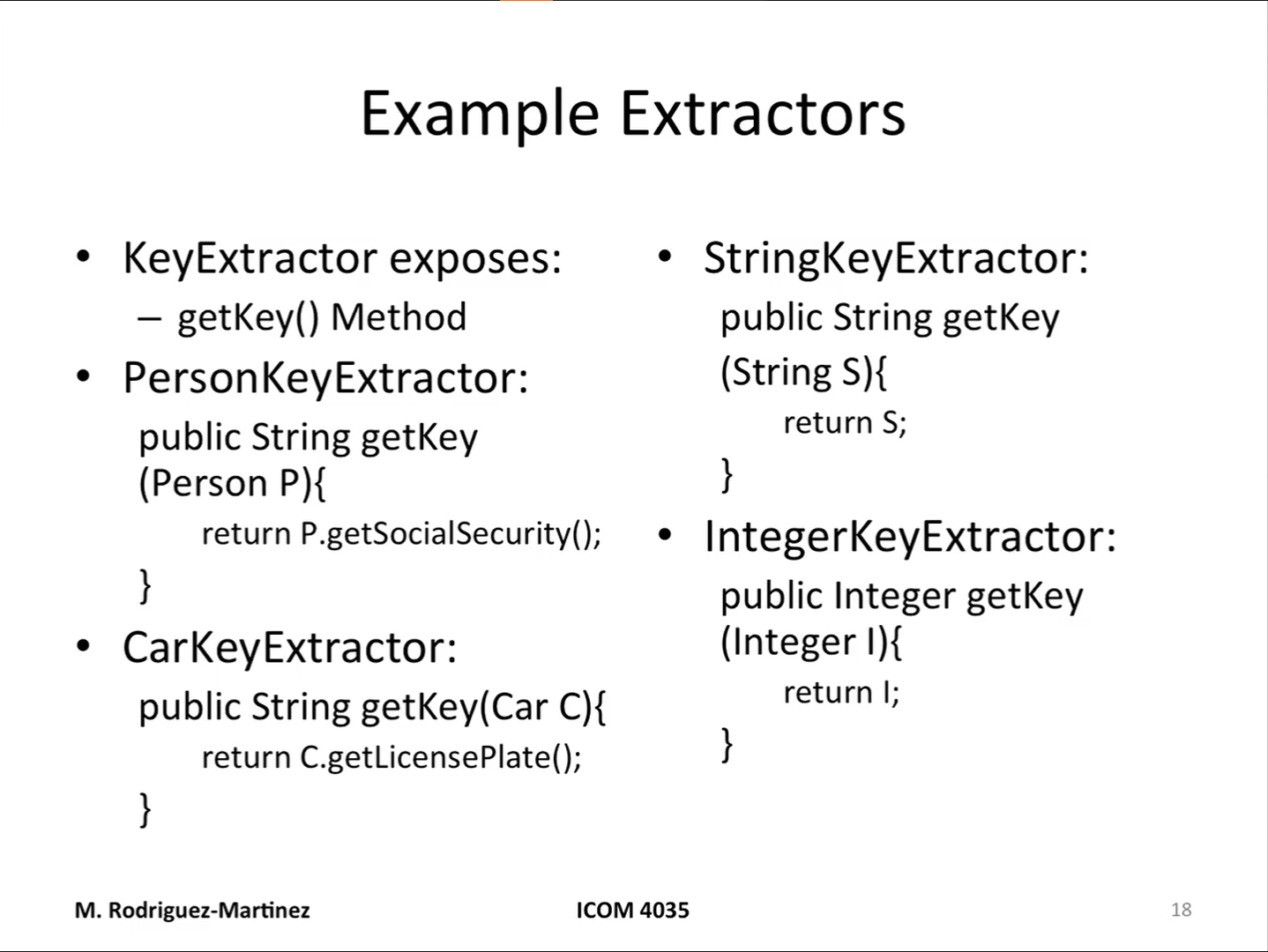
* Part I) Introduction to the Linked List Data Structure
  + Linked List
    - List that is fully dynamically allocated at runtime
    - There is no need to pre-allocate space for the list
      * Rather, elements are chained together in sequence
    - Every entry of a linked list is called a node, which has
      * Value Field
      * Reference to next node in the chain
    - Starts with header node
      * Null value
      * Points to first element of the linked list
    - Nodes in a Linked List could be scattered all around the memory
  + Features of a Linked List
    - Supports all List ADT operations
    - Small memory footprint since only nodes are allocated as needed
    - No limit to capacity (other than RAM size)
    - No need to reallocate or copy elements
    - Variations
      * Singly linked list
        + Nodes only point to the next element
      * Doubly linked list
        + Nodes point to both the next and the previous element
      * Circular Doubly linked list
        + Nodes form a circular chain
* Part II) Design and implementation of Linked List Data Structure



* + Add new element at the end of list
    - Simply add new node at the end of the list
    - Iterate through the linked list while the next node is not null
  + Add new element at position i
    - First make sure that the index i is valid
    - Use a counter with a for loop to iterate until you reach the predecessor of the node you want to add
  + Remove a copy of an element
  + Remove element at position i
  + Remove all copies of an element
  + Clear List
  + Test for element membership
  + Get List size
  + Get first element in list
  + Get last element in list
  + Get element at position i
  + Set element at position i
  + Get first index of element x
  + Get last index of element x
  + Test if empty
  + Iterate over all stored values
* Part III) Iterating over a Linked Listw

# Lecture 11 – MAP ADT

* Part I) Introduction to the Map ADT
  + Map
    - Collection of items that can be retrieved through a key
      * Stores key-value pairs
      * Key represents a unique identifier for each item
      * Each item has its key
    - Repetitions are not allowed (therefore, old values are overwritten)
      * Multiple values with the same key is called a Dictionary
    - Similar to mathematical functions
    - Key
      * Attribute that uniquely identifies values stored
      * Frequently are randomly generated to prevent privacy issues from using phone number or SSN as key for maps
    - Value
      * Data to be stored in the map (simple or complex objects)
  + Operations in a Map
    - size() – number of elements in map
    - isEmpty() – determine if the map is empty
    - get(key) – returns the value associated with a key
    - put(key, value) – adds a new value to the map with a given key
    - remove(key) – removes the value with the given key from map
    - makeEmpty() – removes all values from map
    - containsKey() – determines if there is a value with the given key
    - getKeys() – returns a list with all keys in the map
    - getValues() – returns a list with all values in the map
  + Example applications
    - Non-relational data-base systems (CouchDB, MongoDB, Hive, etc.)
    - Processing web data with JSON format
* Part II) Design and implementation of a Map using Linked List
  + Each node is a value of the map
  + In practice, keys are extracted from the values in the map (instead of having a particular field for key in the Linked List node)
  + Methods to extract keys from values
    - Value can implement interface with method to get key (a “Keyable” interface of sorts)
      * The problem is that existing objects such a String, Integer, etc. do not implement such methods, and we would have to re-implement them
    - Map includes object that knows how to extract key from value (Key extractor)
      * KeyExtractor class that exposes getKey() method



* + Operations
    - get()
      * Returns object that has a specific keys
      * Linearly search (using getKey) in Linked List until element with particular key is found
        + If empty, return null
    - put(key, value)
      * If other value in map has that same key, erase it first
      * Create new element
      * Insert node to Singly Linked List at position 0
      * Change size of map
    - remove(key)
      * Linearly search (using getKey) for node with key in Linked List
      * When found, erase element from linked list
      * Change size of map
    - containsKey(key)
      * Use the get function
      * If result is None, key is not in map
    - getKeys()
      * Create new list and add key of every value
    - getValues()
      * Create new list and add value of ever node
    - size()
      * Returns Linked List current size
    - isEmpty()
      * Check if size == 0
    - makeEmpty()
      * Call clear() on the linked list

# Lecture 12 – Hashtables & Map ADT

* Part I) Introduction to the Hash Table and its use for implementing a Map
  + Hashing is a method to map a given key into a specific position within an Array List
    - Array List size is typically chosen to be a prime number to better distribute values uniformly
    - Hashing maps the key into a hash code (which might be a HUGE number)
    - For this reason, we can map hash codes to Array List positions using modulo operator to get a value of 0 to len(Array List) - 1
  + Methods to compute hash codes
    - Take integer and shift it M times, and add the integer back
    - Sum character’s ascii number to a collector variable
    - Polynomial
      * hashCode = sum([ord(char) \* (k \*\* i) for i, char in enumerate(string)])
    - Cyclic
      * hashCode = 0
      * for i in range(len(string)):
        + hashCode = (hashCode << 5) | (hashCode >> 27)
        + hashCode += ord(string[i])
  + Vocabulary
    - **Entries** are called “buckets”
    - **Load Factor** (λ) = size of table / number of buckets
      * As λ gets to 70%, table is expanded and rehashed
* Part II) Design and Implementation of a Hash Table using Separate Chaining
  + Each Bucket is a Linked List
  + Hash Collisions are managed by adding to end of bucket’s Linked List
  + Buckets have average size of n/N (amount of elements over capacity of Hash Table)
    - Assuming hash functions distribute keys uniformly through the table
  + Operations involve
    - Hashing key ( O(1) )
    - Searching / inserting / deleting in bucket ( O(n/N) )
    - Therefore, the average cost is: O( 1 + n/N )
  + The operations are the same as the map, but are implemented in a different way
* Part III) Design and Implementation of a Hash Table using Open Addressing
  + Wenas

# Summary of Data Structures

## Bag / Multi-Set

* Unordered collection of elements where repetitions are allowed. Can be implemented as
  + Static bag (bag capacity cannot be changed)
  + Dynamic bag (every time maximum capacity is reached, previous bag size is doubled)

\*Note:

Length = amount of elements it has

Capacity = maximum amount of elements it can hold

|  |  |
| --- | --- |
| Operations | Time Complexity |
| Add new element | O(1) |
| Remove a copy of an element | O(n) |
| Remove all copies of an element | O(n) |
| Clear bag | O(n) |
| Count copies of element | O(n) |
| Test for element membership | O(n) |
| Get bag size | O(1) |
| Test if empty | O(1) |
| Iterate over all stored values |  |

## Set

* Unordered collection of elements where repetitions are NOT allowed. Can be implemented as
  + Static Set (capacity of set cannot be changed)
  + Dynamic Set (every time maximum capacity is reached, previous set size is doubled)

|  |  |
| --- | --- |
| Operations | Time Complexity |
| Add new element | O(n) |
| Remove a copy of an element | O(n) |
| Test for element membership | O(n) |
| Clear set | O(n) |
| Get set size | O(1) |
| Test if empty | O(1) |
| Compute Union | O(n2) |
| Compute Difference | O(n2) |
| Compute Intersections | O(n2) |
| Test if subset | O(n2) |
| Iterate over all stored values |  |

## List / Finite SEQUENCE

* Ordered collection of elements where the order is associated with the element’s numeric position within the list. Repetitions are allowed, and it can be implemented as
  + Static-size List (capacity of list cannot be changed)
    - Array
  + Dynamic-size List (every time the capacity is reached, previous list size is doubled)
    - Array List
    - Linked List

|  |  |  |
| --- | --- | --- |
| Operations | Array List | Linked List |
| Add new element at end | O(1) | O(n) OR O(1) with dummy tail |
| Add new element at position i | O(n) | O(n) |
| Remove first copy of an element | O(n) | O(n) |
| Remove element at position i | O(n) | O(n) |
| Remove all copies of an element | O(n) | O(n) |
| Get element at position i | O(1) | O(n) |
| Set element at position i | O(1) | O(n) |
| Get first element | O(1) | O(1) |
| Get last element | O(1) | O(n) OR O(1) with dummy tail |
| Get first index of element x | O(n) | O(n) |
| Get last index of element x | O(n) | O(n) |
| Get list size | O(1) | O(1) |
| Test if empty | O(1) | O(1) |
| Test for element membership | O(n) | O(n) |
| Clear list | O(n) | O(n) |
| Iterate over all stored values |  |  |

1. Stack
   * Collection of elements with restriction on access. These are:
     + Elements are added/removed from the top
     + There is no notion of specific position for element other than the element at the top. **Notion of vertical orientation**
     + Repetitions are allowed
   * Last-In First-Out (LIFO) list
   * It can be implemented with any kind of list

|  |  |  |
| --- | --- | --- |
| Operations | Array List | Linked List |
| Add new element at top of stack (push) | O(n)  If reallocation is needed | O(1) |
| Remove element at top of stack (pop) | O(1) | O(1) |
| Inspect element at top of stack (top) | O(1) | O(1) |
| Get stack size | O(1) | O(1) |
| Test if empty | O(1) | O(1) |
| Clear stack | O(n) | O(n) |

1. Queue
   * Collection of elements with restriction on access. These are:
     + Elements are added at the end and removed from the front
     + There is no notion of specific position for elements other than element at the front.
     + Repetitions are allowed
   * First-In First-Out (FIFO) list
   * It can be implemented with
     + Doubly Linked List
     + Circular array
       - Array model where
         * The element that follows the last index is the first one
         * The element before the first index is the last one
       - Using the modulo operator, keep all used array slots in contiguous segment and unused in another segment

|  |  |  |
| --- | --- | --- |
| Operations | Circular Array | Linked List |
| Add element at end of queue (enqueue) | O(n)  If reallocation is needed | O(1) |
| Remove element from beginning (dequeue) | O(1) | O(1) |
| Inspect element at beginning (front) | O(1) | O(1) |
| Get queue size | O(1) | O(1) |
| Test if empty | O(1) | O(1) |
| Clear queue | O(n) | O(n) |

1. Map
   * Collection of items that can be retrieved through a key
     + Stores key-value pairs
     + Key represents a unique identifier for each item
     + Each item has its key
   * Repetitions are NOT allowed
   * Can be implemented with:
     + Singly-Linked List
     + Hash Table

|  |  |  |
| --- | --- | --- |
| Operations | Linked List | Hash Table |
| Add new value to the map with a given key (overwrites old one) | O(n) | O(1) or O(n) in hash collisions |
| Remove value with the given key from map | O(n) | O(1) or O(n) in hash collisions |
| Get value associated with a key | O(n) | O(1) or O(n) in hash collisions |
| Determines if there is a value with a given key | O(n) | O(1) or O(n) in hash collisions |
| Return a list of all keys in map | O(n) | O(n) |
| Return a list of all values in map | O(n) | O(n) |
| Get number of values in map | O(1) | O(1) |
| Test if map is empty | O(1) | O(1) |
| Test for element membership | O(n) | O(1) or O(n) in hash collisions |
| Clear map | O(n) | O(n) |

1. Binary Search Tree
   * Unbalanced Binary Tree where, for any node N in T
     + Every node V < N that is a descendant of N will be located on the left sub-tree rooted at N
     + Every node V >= N that is a descendant of N will be located on the right sub-tree rooted at N
   * Can be used to implement the Map and Set ADTs

|  |  |
| --- | --- |
| Operations | Recursive Implementation |
| Add new element with given key | O(height) |
| Remove element with a given key | O(height) |
| Get value of element with given key | O(height ) |
| Test for element membership | O(height) |
| Get BST size | O(1) |
| Test if empty | O(1) |
| Clear BST | O(n) |
| Iterate over all stored values |  |