**EDA NOTES**

**TOPIC 1.-INTRODUCTION: DATA STRUCTURE AND ALGORITHMS IN JAVA**

1.- DATA STRUCTURES

A data structure (**EDA**) is a set of **operations** that define the “behaviour” of a data collection and its **representation** (in memory).

There are two levels of abstractions or components that are necessary for the description of a EDA:

* The **model** of an EDA: description of the operations, independently of its representation in memory.
* The **implementation** of an EDA: representation in memory of data and, based on that, implementation of the operations that define the model.

2.- DESIGN OF AN EDA IN JAVA

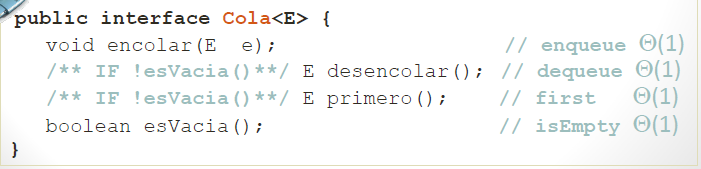
2.1.-JAVA HIERARCHY OF AN EDA

A data structure EDA is described in Java with a hierarchy composed of:

* An **interface** (“root” of the generic type) that describes its **model**.
* Every derived **class** from the root (via **implements**) of generic type, that describes one of its **implementations**.

2.2.- THE QUEUE HIERARCHY: FIFO (FIRST IN FIRST OUT)

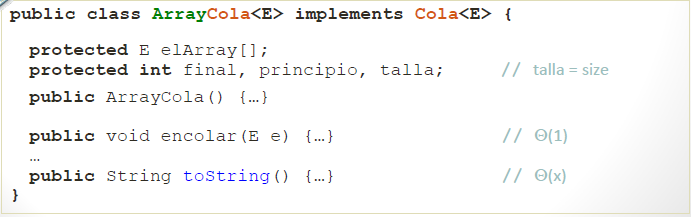
A **Queue** is a data collection organized with a **FIFO** policy.



2.3.- A DERIVED CLASS OF THE QUEUE HIERARCHY

If **ArrayCola<E>** is a class that implements the interface Cola<E>:

1. It uses a generic **array** to store data.
2. In order to implement efficiently the methods of the interface:
   * A circular **array** should be simulated.
   * It has attributes: **final**, **principio** and **talla**.
3. It overwrites the method toString().



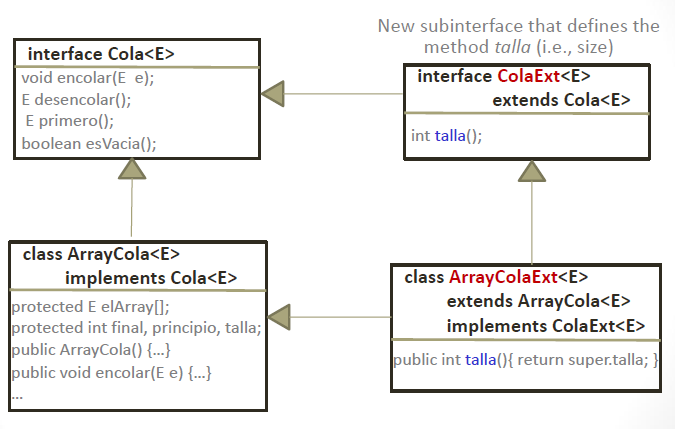
2.4.- THE STACK HIERARCHY: LIFO (LAST IN FIRST OUT)

A **Stack** is a data collection organized with a **LIFO** policy.

3.- USE OF THE JAVA HIERARCHY OF A DATA STRUCTURE: MODALITIES

The hierarchy of a data structure EDA can be used to design new classes:

* Via **composition** (it **has** A). Use if an object of an EDA for a concrete application.
* Via **inheritance** (it **is** A). The best solution consists in **extending** via **inheritance** the Java **hierarchy** of the **EDA**. Example:

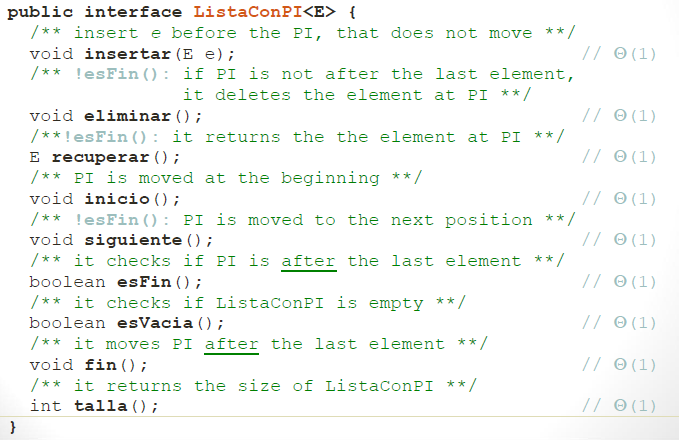


4.- LIST WITH ITERATOR

At a given moment only it is possible to access an element of the **List**, the element at the **Point of Interest (PI)**. The access to the element at the PI is possible in constant time. The position of the element in the List is not a parameter of the sequential access operations:

* At the **beginning** the PI is at the position of the **first** element of the List.
* The sequential access to the list of elements is possible moving the PI to the **next** element.
* To **insert**, **retrieve** or **delete** an element from the List, the **PI** does **not** **move**.
* If a List **is** **empty** or the sequential access has reached the end, there is no element at the PI.

A **ListaConPI** is a data collection that is accessed **sequentially**, the element of the List that is accessible at a given moment is the one at the PI.



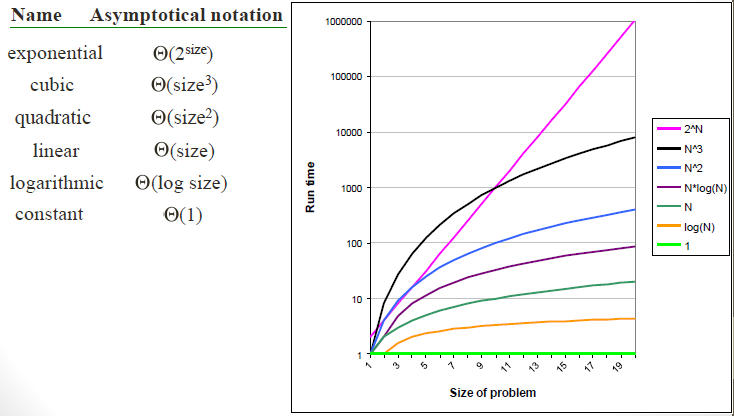
5.- CLASSES RESTRICTED WITH COMPARABLE

Java provides a criterion for a generic comparison, a method **compareTo(other)** that can be used the same way than **equals(other)**.

The method **compareTo()** is not defined in **Object**, thus, it cannot be used exactly as **equals()**. It is the only method of the interface **java.lang.Comparable**, the standard and generic model provided by Java for the comparison of two generic type objects.

**TOPIC 2.A- DIVIDE AND CONQUER. SORTING AND SELECTING**

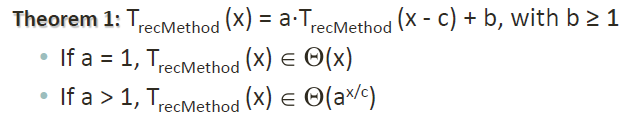
1.- ANALYSIS OF COMPLEXITY

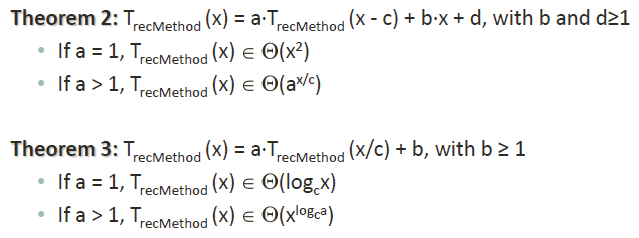


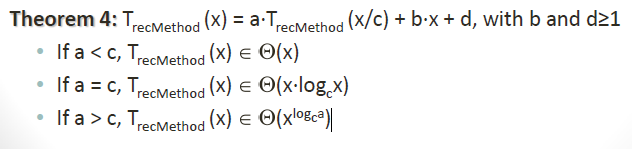
1.1.- RECURRENCE RELATIONS

The complexity of a recursive method depends on:

* The number of recursive invocations.
* The way the size of the problems decreases.
* The complexity of the calculations in each invocation.







Where:

* “a” is the number of recursive invocations.
* “x/c” or “x-c” is the decreased size of x.

2.- DIVIDE AND CONQUER

2.1.- INTRODUCTION

Divide & Conquer technique is based on the following steps:

* DIVIDE: a problem of size x is divided in N > 1 disjoint subproblems, with the size of the subproblems the most similar as possible.
* CONQUER: solve recursively each subproblem.
* COMBINE: combine the solutions of the subproblems in order to obtain the solution of the original problem.

2.2.- SORTING AN ARRAY

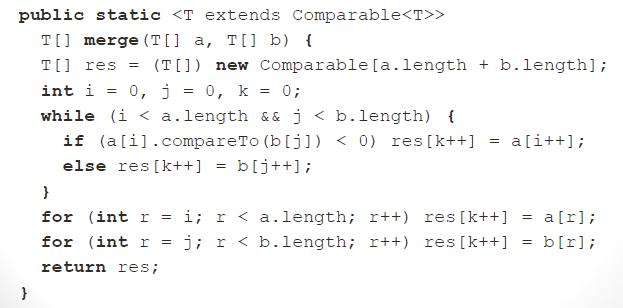
The easiest sorting algorithms (InsertionSort, SelectionSort and bubbleSort) have a quadratic complexity.

The methods QuickSort and MergeSort employ the D&C strategy in order to improve the efficiency:

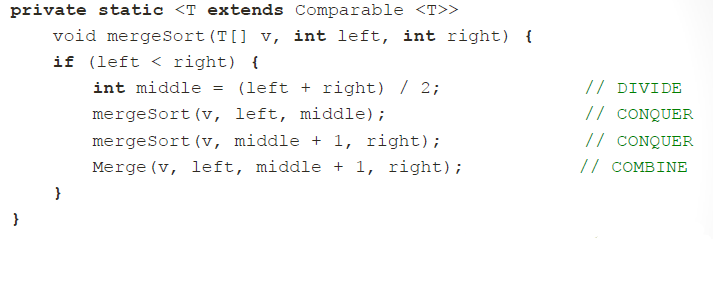
* The original problem is divided into subproblems (a=2) whose size is approximately the half of the original one (c=2).
* Divide and combine has a linear complexity.
* The complexity of both algorithms is (x\*log2 x) applying theorem 4.

2.3.- MERGE SORT

Merge:



MergerSort:



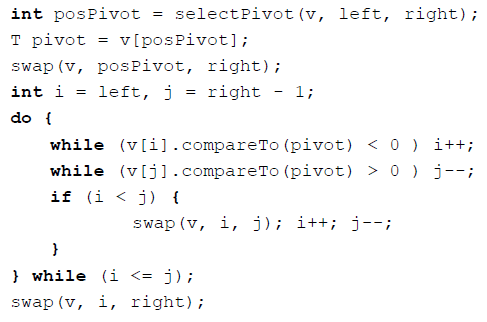
2.4.- QUICK SORT

Given an array “v”:

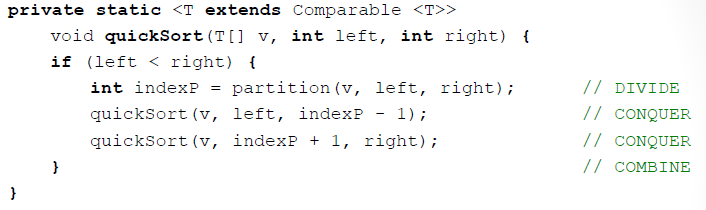
1. An element of the array is chosen (**pivot**).
2. Given the pivot, the elements of the array are organised in a way that the elements on its left are smaller and those on its right are greater.
3. We do the same with the subarrays on its left and right.

A good pivot divides the array in two subarrays of equal size, that is, it has to be the **median** of the array. To calculate the median has a high complexity. Therefore, as approximation the **median of three** is employed (as the leftmost element, the rightmost element and the element in the middle).

Partition:



QuickSort:



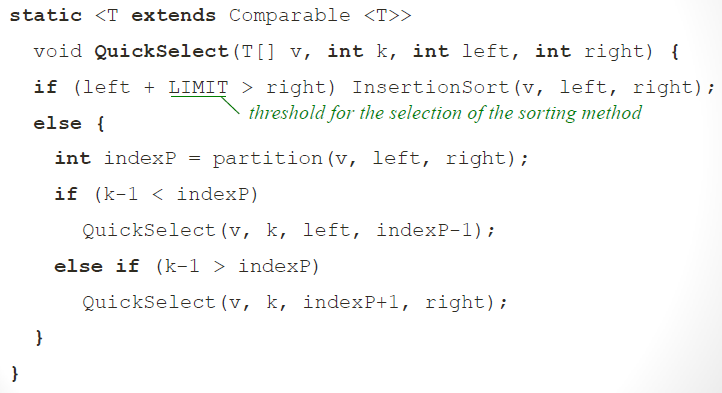
The complexity of QuickSort depends on the method **partition**:

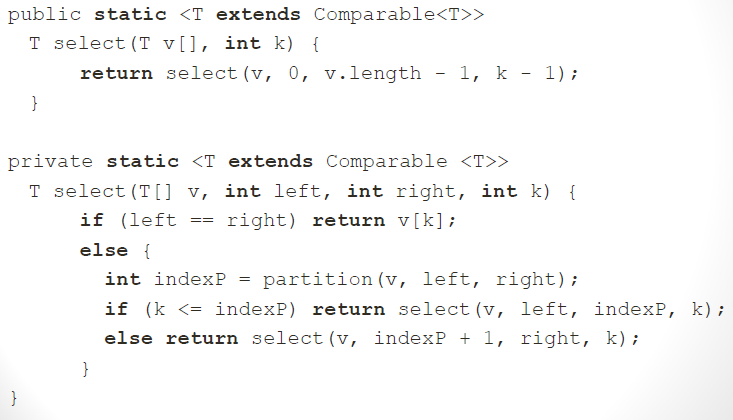
* Best case: **partition** divides the array into two balanced halves( (x\*log2 x) ).
* Worst case: **partition** divides it into completely imbalanced two parts ( (x2) ).

Anyway, the process of **partition** is more efficient than **merge**.

2.5.- QUICK SELECT

Find the k-th smallest element of an array in linear cost.





**TOPIC 2.B.- SELECTION ALGORITHM, BINARY SEARCH AND MASTER THEOREMS**

1.- SELECTION ALGORITHM

D&C solution:

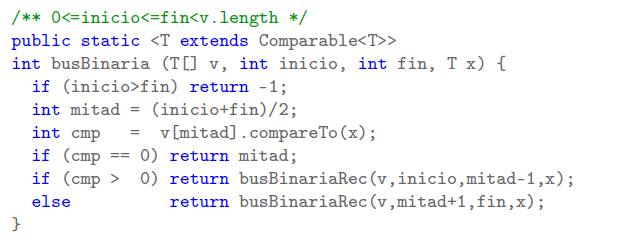
* Divide (partition): the array A[p…r] is partitioned in two subvectors A[p…q] and A[q+1…r], so that the elements of A[p…q] are less than or equal to the pivot and those of A[q+1…r] are greater or equal.
* Conquer: we search in the corresponding subvector doing recursive call to the algorithm.
* Combine: if k <= q, then k-th smallest item will be in A[p…q], otherwise, it will be in A[q+1…r].

Worst case: ordered vector in a non-decreasing way and we look for the greatest element. Cost: (x2).

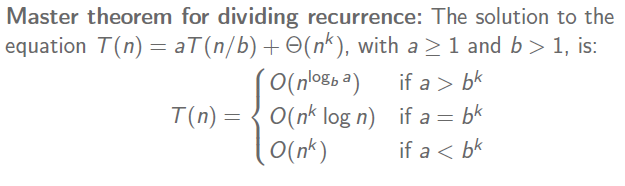
Best case: the item to look is the minor or the biggest and this acts as a pivot. Cost: (x)

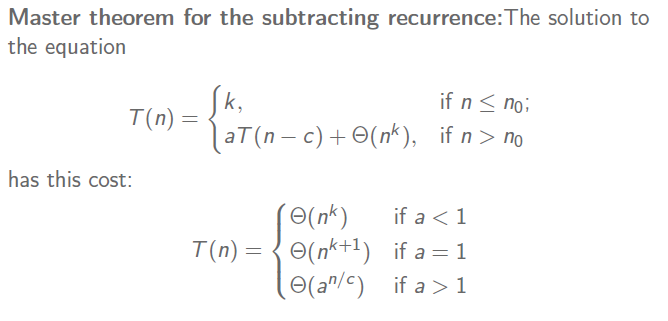
2.- BINARY SEARCH

Given a vector of size n, ordered in increasing order, find x. With D&C and taking advantage of the fact that the vector is ordered (log x).



3.- MASTER THEOREMS





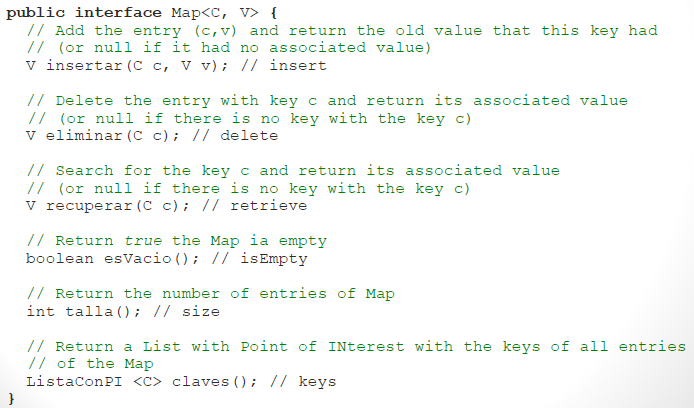
**TOPIC 3.-MAPS AND HASH TABLES**

1.- THE MAP MODEL

The **Map** model is designed to ease data search in a collection. The data that are stored in a **Map** are key-value pairs, where:

* The search is carried out depending on the **key**.
* The **value** is the information associated at the key that we aim at retrieving.

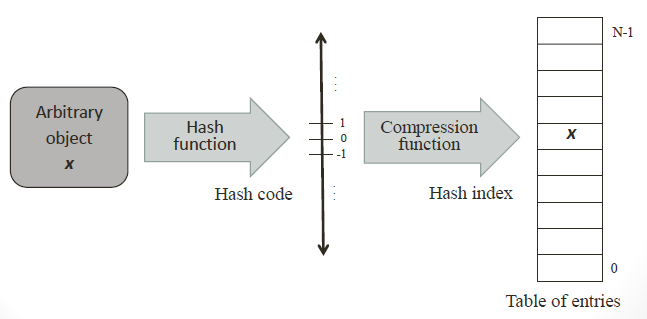
The basic operation of a **Map** is searching by key (or name) in a collection of entries.



2.- HASH TABLE

2.1.- THE CONCEPT OF HASH

Data structure designed for the implementation of Maps that has operations: *retrieve*, *insert* and *delete* in constant time.



2.2.- HASH FUNCTION – SIMPLE METHOD

A **hash** **function** is a function that converts an entry in an integer (hash code) appropriate to index the table in which this entry will be stored.

**Simple** **method**: sum of the components. The sum of components is not a good hash function since it is easy that two different entries have the same hash code (**collision**).

2.3.- POLYNOMIAL HASH FUNCTION

In order to improve the quality of the hash function, it is possible to use polynomial functions, that is, to weight the position of each character of the key.

2.4.- THE METHOD HASHCODE OF JAVA

Every class that is going to be used as key in a Map must overwrite properly the **hashCode()** method.

2.5.- COMPRESSION FUNCTIONS

The **hash code** can be a value greater than the size of the array. I can be also a negative number.

A **compression function** is a function that converts a *hash code* in a **hash index** between 0 and the size of the *array* minus one.

2.6.- COLLISIONS

The hash function returns always the same value for the same entry. If two entries are different, then the hash function should return two different values. Although this is not strictly necessary, this feature improves the efficiency of hash tables. Even with a good hash function, collisions can occur, then we need efficient methods to solve collisions:

* Open addressing.
* Separate chaining.

In **open** **addressing**, if we are going to insert an element in a specific position which is already taken, we search for an alternative position:

* The **linear** **exploration** solves a collision searching sequentially, starting from *hashIndex* until the next free position in the table.
* The **quadratic exploration** solves a collision checking the positions.

In **separate** **chaining**, all the entries which collide in the same position are stored in a linked list. Each list is called **bucket**.

2.7.- LOAD FACTOR

The performance of a hash table is measured in terms of its **load** **factor**, which is defined as the average length of its buckets: actualSizeOfTable / sizeOfArray. Therefore, the efficiency of a hash table depends on:

* The quality of its **hash** **function**.
* Its **load** **factor**.
* Its method to **solve** **collisions**.

2.8.- REHASHING

The number of collisions can grow too much if the load factor is very high. The **rehashing** consists in increasing the size of the hash table, reducing its occupancy rate.

3.- IMPLEMENTATION

EntradaHash:

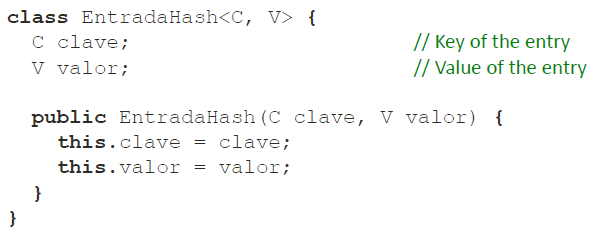
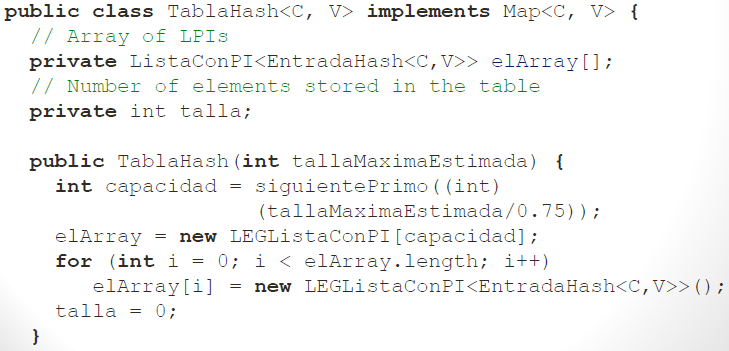
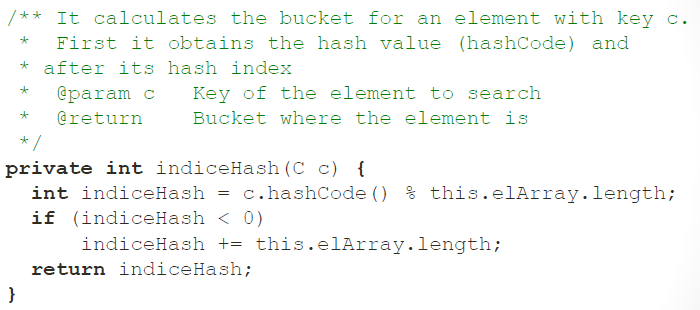
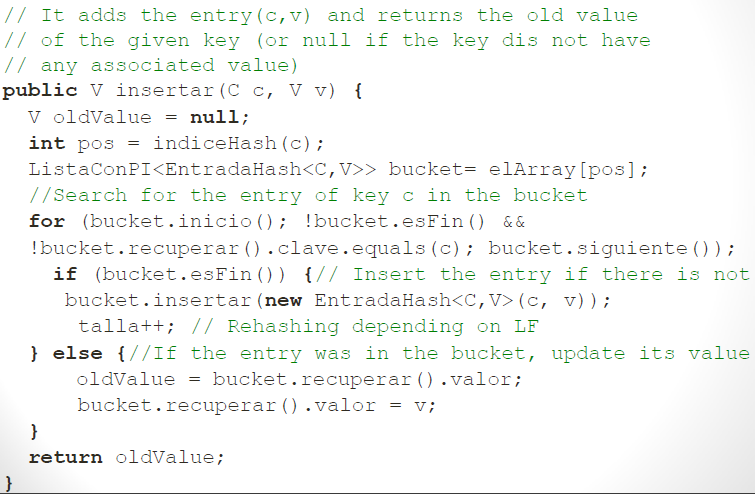
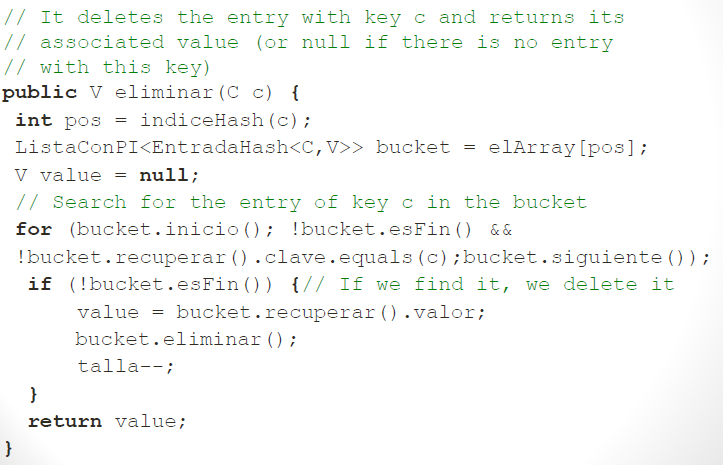


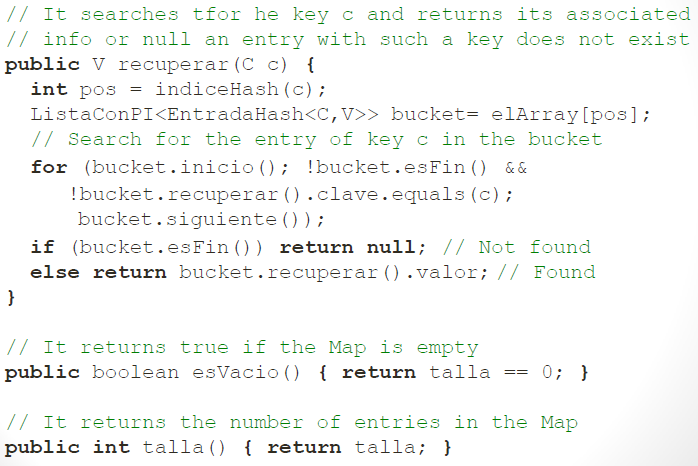
Tabla hash (it is highly recommended that the **size** of the array is a **prime** **number**):











**TOPIC 4.-TREE, BINARY TREE AND BINARY SEARCH TREE**

1.- CONCEPTS OF TREES

**Linear data structures** allow to describe sets of data that allow relationships of successor. **Trees** allow to represent hierarchical structures among data sets.

Sometimes data of a collection have hierarchical relationships that is not possible to model with a linear representation.

A **tree** is a hierarchical structure that is possible to define through a set of **nodes** (one is the **root** of the tree) and a set of **edges** such that:

* Each node H, with the exception of the root, is linked to a unique node P via an edge. P is the node **father** and H is the **child**.
* A node without children is a **leaf**.
* A node that is not a leaf is an **inner** **node**.
* The **degree** is the number of its children.

In a tree there is a unique **path** from the root to each node. The number of edges of a path gives its **length**. The **depth** of a node is the length of the path from the root to the node:

* The depth of the root is 0.
* All the nodes at the same depth belong to the same **level**.

The **height** of a node is the length of the path from the node to its deepest leaf. The height of a tree is the height of its root.

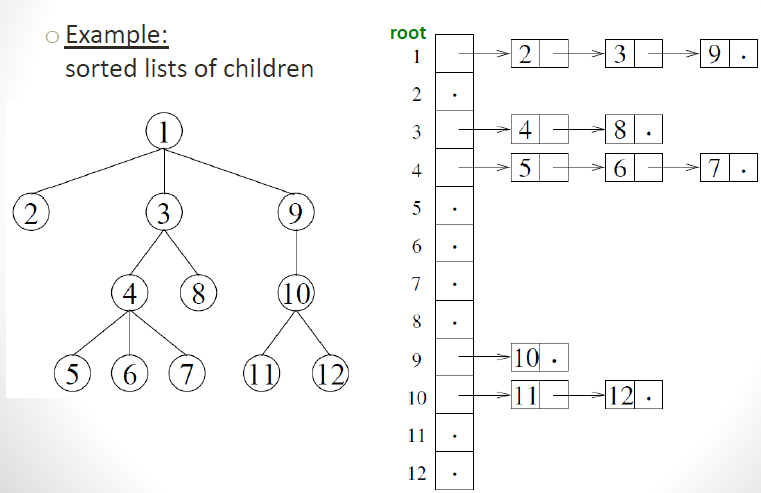
A tree is:

* An empty set.
* A root and zero or more not empty sub-trees where each of its roots are linked via an edge to the root.

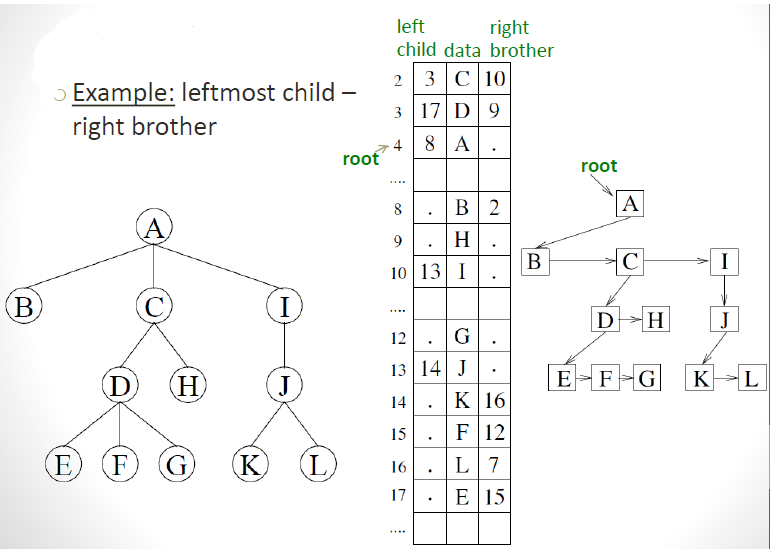
2.- GENERIC TREES

Representation of generic trees:

* Lists (sorted) of children.



* Leftmost child – right brother.



3.-BINARY TREES

A **binary** **tree** is a tree where each node has a maximum of two children. Properties:

* The maximum number of nodes at level **i** is **2i**.
* In a tree of height H, the maximum number of nodes is 2H+1-1.
* The maximum number of leaves is 2H.
* The maximum number if nodes is 2H-1.

A **binary** **tree** is **full** if all its levels are complete. Properties:

* H = .
* N = 2H+1-1.

A **complete** **binary** **tree** has all its levels complete, except maybe the last one, where all the leaves are leftmost as possible. Properties:

* H <= then, it is a **balanced** **tree**.
* 2H <= N <= 2H+1-1.

A tree is **balanced** if the difference between the heights of the left and right sub-trees of each of its nodes is as maximum equal to 1.

4.- TRAVERSAL OPERATIONS

In a traversal **by** **levels** of a binary tree, the nodes are visited by level and, in each level, from left to right.

In a traversal **by** **depth**, the nodes can be visited in the following orders:

* **Pre-Order**:
  1. Root.
  2. Left sub-tree.
  3. Right sub-tree.
* **In-Order**:
  1. Left sub-tree.
  2. Root.
  3. Right sub-tree.
* **Post-Order**:
  1. Left sub-tree.
  2. Right sub-tree.
  3. Root.

5.- BINARY SEARCH TREES

Data structure that can be used for the implementation of dictionaries and priority queues. It is a generalisation of the binary search. It allows for implementing efficiently operations such as search, search min, search max, predecessor and successor. It allows also for an efficient implementation of the insert and delete operations.

A binary tree is a **binary search tree** (BST) if:

* The data of its **left** sub-tree are **smaller** than the root.
* The data of its **right** sub-tree are **greater** than the root.
* The left and right sub-trees are also binary search trees.

If a binary search tree is visited in-order the result is a sorted sequence of its elements.

NodoABB:

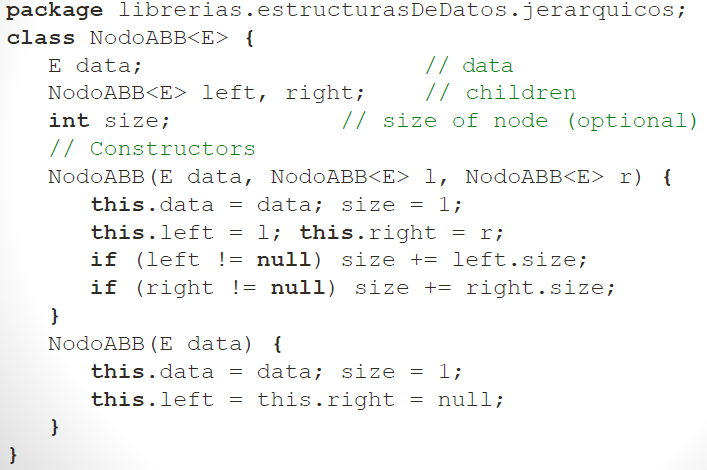
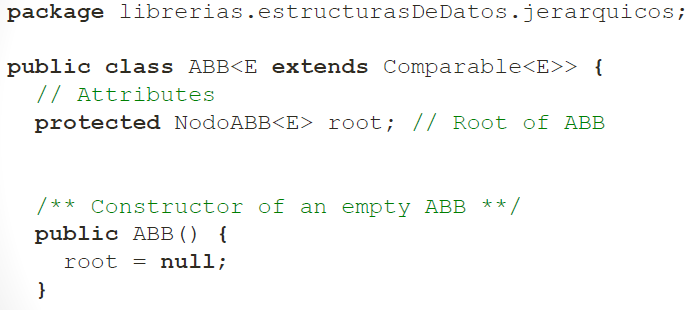
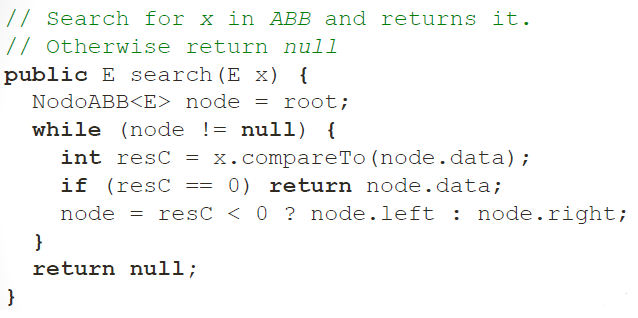
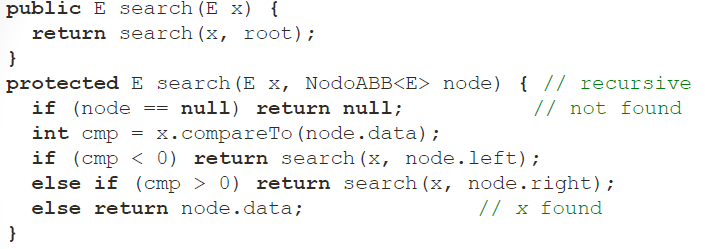


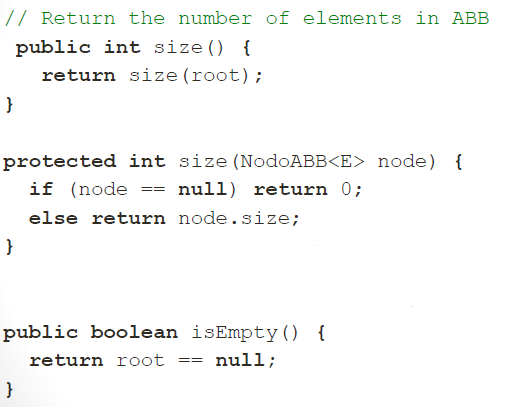
ABB:

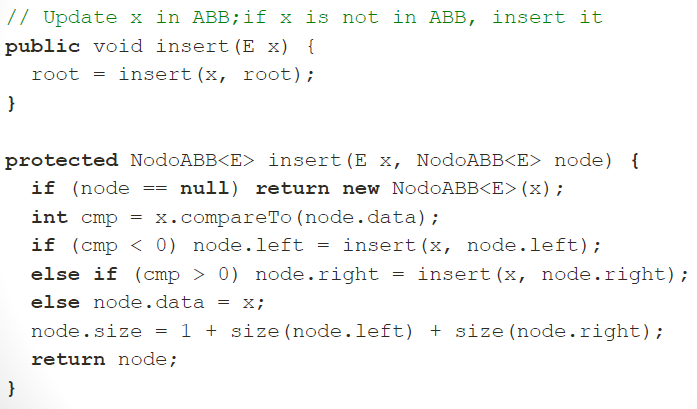




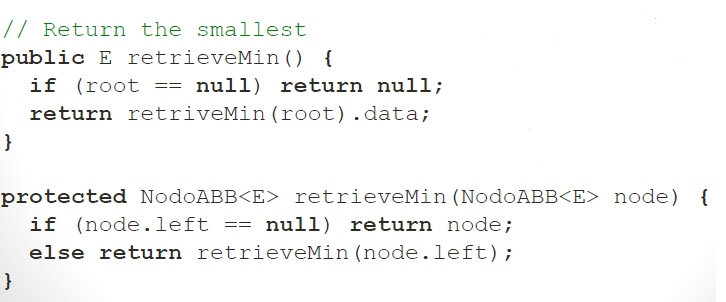
Recursive version of search:

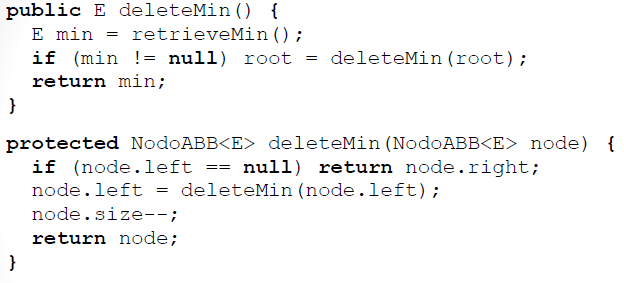






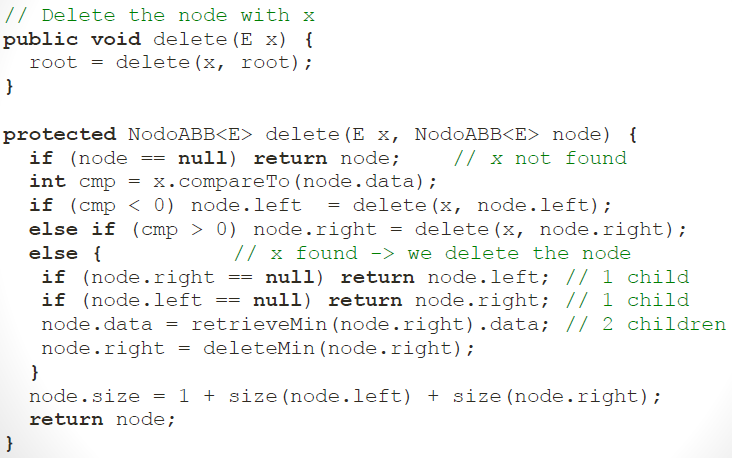
The smallest in an ABB does not have left child and it does not belong to any subtree of any node. For the greatest is the symmetric case.

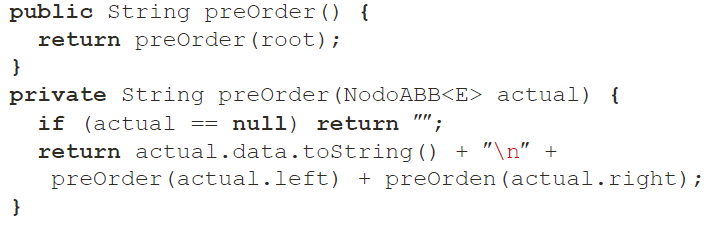


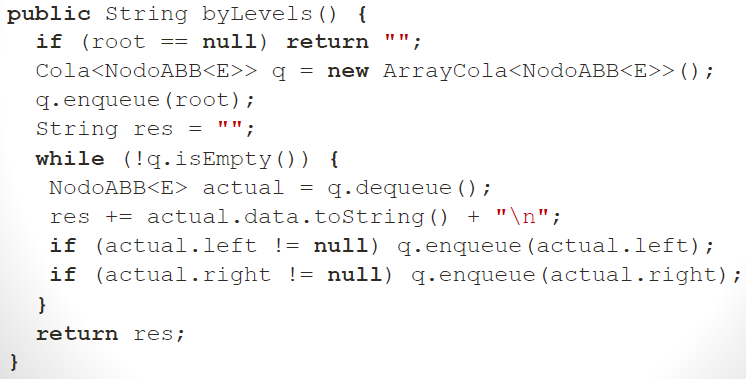


To **delete** in an ABB there are **three** possible cases:

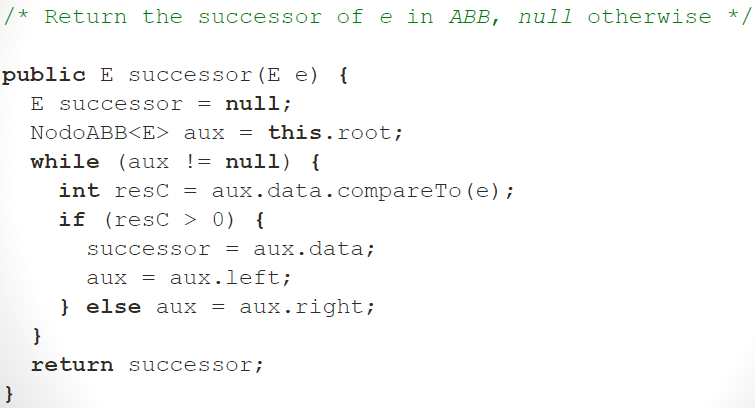
1. The node to be deleted does not have children. Just delete this node (null).
2. The node to be deleted has a child. Its child takes its position.
3. The node to be eliminated has two children. The smallest of its right subtree takes its position.





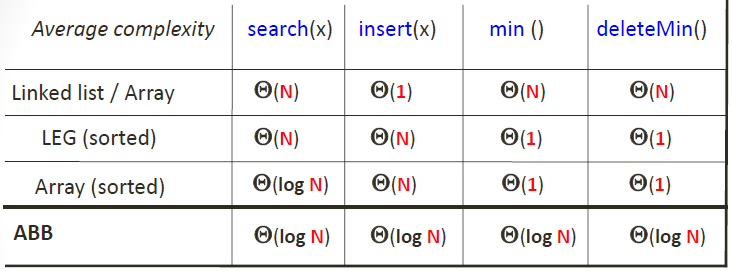


If a node has the right sub-tree, the successor of the node is the min of its right sub-tree. Otherwise, the successor is the closest ancestor. The successor of a node is the next visited node in an in-order transversal of the tree.



The complexity of the operations in an ABB depends on the height of the tree (H). The height H is between Ω(log2n) and O(n). In the worst case (ABB imbalanced), the complexity is linear.

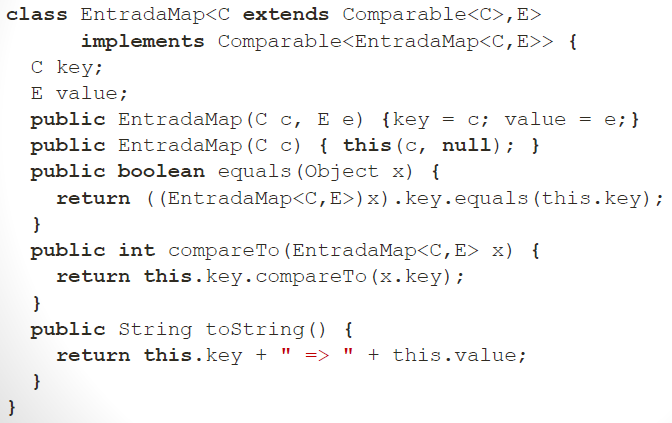
Complexity of operations:



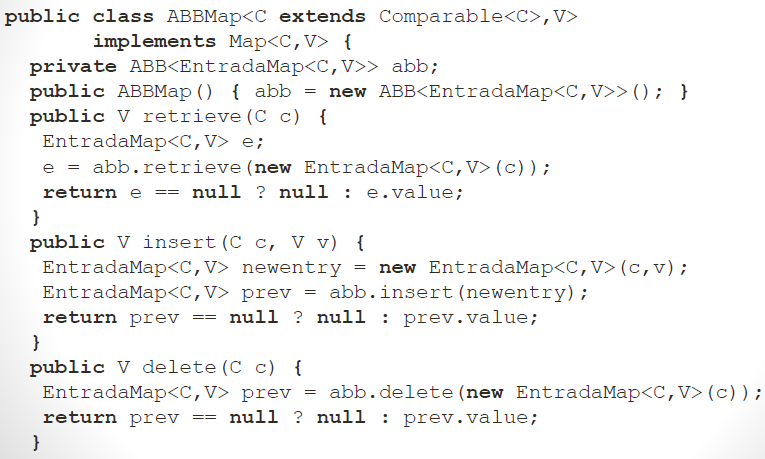
6.- THE CLASS ABBMAP

The model **Map** allows searching for **key** obtaining the associated **value** to the given entry. An entry is a pair (key, value). Two entries with the same key are not allowed.

EntradaMap:



ABBMap:



7.- THE CLASS ABBCOLAPRIORIDAD

