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.

```
public interface Cola<E> {
   void encolar(E  e);                    // enqueue Θ(1)
   /** IF !esVacia()**/ E desencolar();  // dequeue Θ(1)
   /** IF !esVacia()**/ E primero();     // first   Θ(1)
   boolean esVacia();                     // isEmpty Θ(1)
}
```

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().

```
public class ArrayCola<E> implements Cola<E> {

  protected E elArray[];
  protected int final, principio, talla;      // talla = size
  public ArrayCola() {…}

  public void encolar(E e) {…}                // Θ(1)
  …
  public String toString() {…}                // Θ(x)
}
```

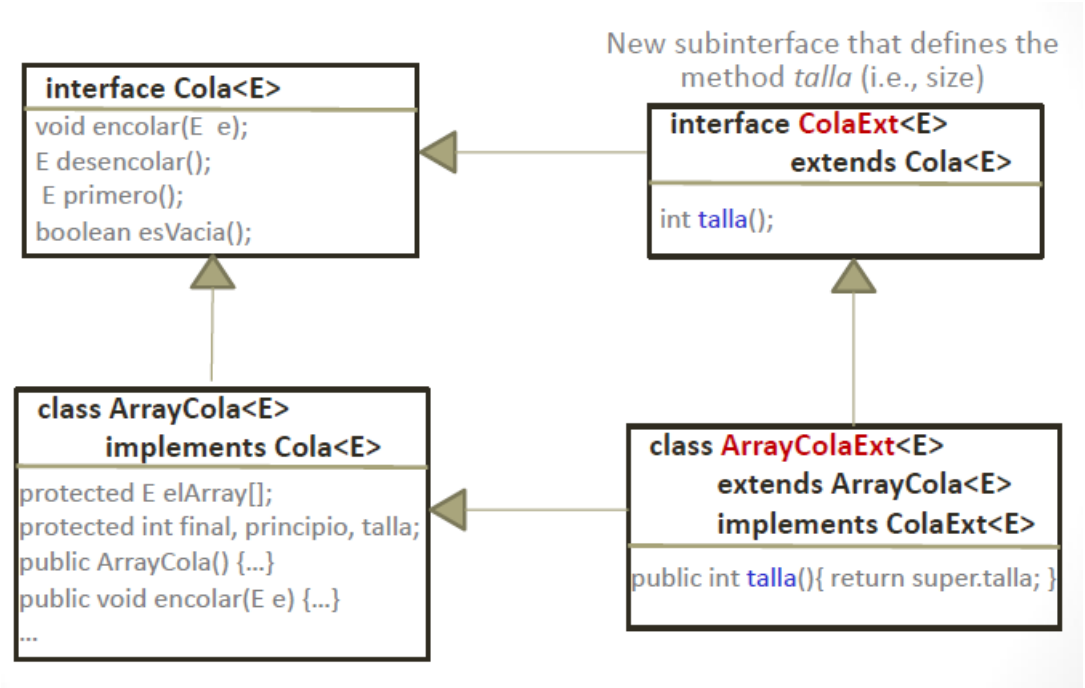
2.4.- THE STACK HIERARCHY: LIFO (LAST IN FIRST OUT)

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.

```
public interface ListaConPI<E> {
  /** insert e before the PI, that does not move **/
  void insertar(E e);                                    // Θ(1)
  /** !esFin(): if PI is not after the last element,
                it deletes the element at PI **/
  void eliminar();                                       // Θ(1)
  /**!esFin(): it returns the the element at PI **/
  E recuperar();                                         // Θ(1)
  /** PI is moved at the beginning **/
  void inicio();                                         // Θ(1)
  /** !esFin(): PI is moved to the next position **/
  void siguiente();                                      // Θ(1)
  /** it checks if PI is after the last element **/
  boolean esFin();                                       // Θ(1)
  /** it checks if ListaConPI is empty **/
  boolean esVacia();                                     // Θ(1)
  /** it moves PI after the last element **/
  void fin();                                            // Θ(1)
  /** it returns the size of ListaConPI **/
  int talla();                                           // Θ(1)
}
```

5.- CLASSES RESTRICTED WITH COMPARABLE

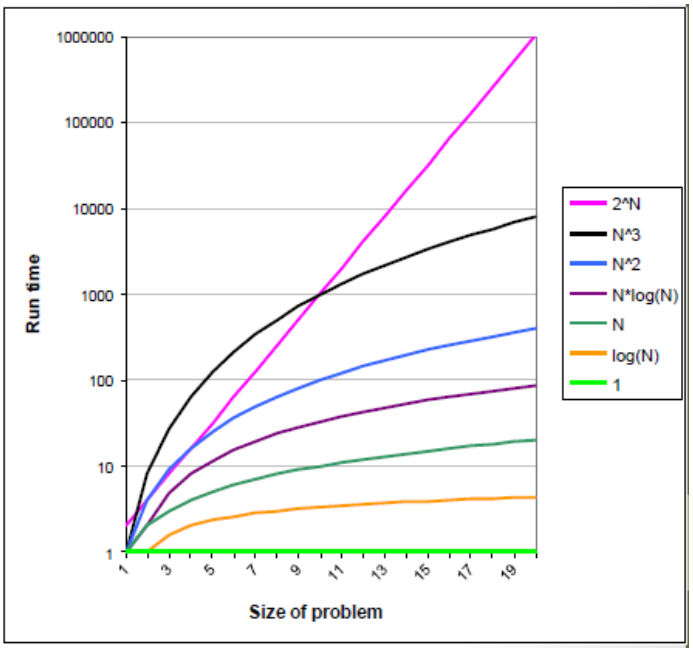
Java provides a criterion for a generic comparison, a method **compareTo(other)** that can be used the same way than **equals(other)**.

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

Name	Asymptotical notation
exponential	$\Theta(2^{size})$
cubic	$\Theta(size^3)$
quadratic	$\Theta(size^2)$
linear	$\Theta(size)$
logarithmic	$\Theta(\log size)$
constant	$\Theta(1)$



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.

Theorem 1: $T_{recMethod}(x) = a \cdot T_{recMethod}(x - c) + b$, with $b \geq 1$

- If $a = 1$, $T_{recMethod}(x) \in \Theta(x)$
- If $a > 1$, $T_{recMethod}(x) \in \Theta(a^{x/c})$

Theorem 2: $T_{recMethod}(x) = a \cdot T_{recMethod}(x - c) + b \cdot x + d$, with b and $d \geq 1$

- If $a = 1$, $T_{recMethod}(x) \in \Theta(x^2)$
- If $a > 1$, $T_{recMethod}(x) \in \Theta(a^{x/c})$

Theorem 3: $T_{recMethod}(x) = a \cdot T_{recMethod}(x/c) + b$, with $b \geq 1$

- If $a = 1$, $T_{recMethod}(x) \in \Theta(\log_c x)$
- If $a > 1$, $T_{recMethod}(x) \in \Theta(x^{\log_c a})$

Theorem 4: $T_{recMethod}(x) = a \cdot T_{recMethod}(x/c) + b \cdot x + d$, with b and $d \geq 1$

- If $a < c$, $T_{recMethod}(x) \in \Theta(x)$
- If $a = c$, $T_{recMethod}(x) \in \Theta(x \cdot \log_c x)$
- If $a > c$, $T_{recMethod}(x) \in \Theta(x^{\log_c a})$

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 $\theta(x*\log_2 x)$ applying theorem 4.

2.3.- MERGE SORT

Merge:

```
public static <T extends Comparable<T>>
  T[] merge(T[] a, T[] b) {
  T[] res = (T[]) new Comparable[a.length + b.length];
  int i = 0, j = 0, k = 0;
  while (i < a.length && j < b.length) {
    if (a[i].compareTo(b[j]) < 0) res[k++] = a[i++];
    else res[k++] = b[j++];
  }
  for (int r = i; r < a.length; r++) res[k++] = a[r];
  for (int r = j; r < b.length; r++) res[k++] = b[r];
  return res;
}
```

MergerSort:

```
private static <T extends Comparable <T>>
    void mergeSort(T[] v, int left, int right) {
    if (left < right) {
        int middle = (left + right) / 2;            // DIVIDE
        mergeSort(v, left, middle);                 // CONQUER
        mergeSort(v, middle + 1, right);            // CONQUER
        Merge(v, left, middle + 1, right);          // COMBINE
    }
}
```

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:

```
int posPivot = selectPivot(v, left, right);
T pivot = v[posPivot];
swap(v, posPivot, right);
int i = left, j = right - 1;
do {
    while (v[i].compareTo(pivot) < 0 ) i++;
    while (v[j].compareTo(pivot) > 0 ) j--;
    if (i < j) {
            swap(v, i, j); i++; j--;
    }
} while (i <= j);
swap(v, i, right);
```

QuickSort:

```
private static <T extends Comparable <T>>
    void quickSort(T[] v, int left, int right) {
    if (left < right) {
        int indexP = partition(v, left, right);      // DIVIDE
        quickSort(v, left, indexP - 1);              // CONQUER
        quickSort(v, indexP + 1, right);             // CONQUER
    }                                                // COMBINE
}
```

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

- Best case: **partition** divides the array into two balanced halves($\theta(x*\log_2 x)$).
- Worst case: **partition** divides it into completely imbalanced two parts ($\theta(x^2)$).

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.

```
static <T extends Comparable <T>>
  void QuickSelect(T[] v, int k, int left, int right) {
  if (left + LIMIT > right) InsertionSort(v, left, right);
                            threshold for the selection of the sorting method
  else {
    int indexP = partition(v, left, right);
    if (k-1 < indexP)
      QuickSelect(v, k, left, indexP-1);
    else if (k-1 > indexP)
      QuickSelect(v, k, indexP+1, right);
  }
}
```

```
public static <T extends Comparable<T>>
  T select(T v[], int k) {
      return select(v, 0, v.length - 1, k - 1);
  }

private static <T extends Comparable <T>>
  T select(T[] v, int left, int right, int k) {
      if (left == right) return v[k];
      else {
        int indexP = partition(v, left, right);
        if (k <= indexP) return select(v, left, indexP, k);
        else return select(v, indexP + 1, right, k);
      }
}
```

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: $\theta(x^2)$.

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

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 $\theta(\log x)$.

```
/** 0<=inicio<=fin<v.length */
public static <T extends Comparable<T>>
int busBinaria (T[] v, int inicio, int fin, T x) {
  if (inicio>fin) return -1;
  int mitad = (inicio+fin)/2;
  int cmp   =  v[mitad].compareTo(x);
  if (cmp == 0) return mitad;
  if (cmp >  0) return busBinariaRec(v,inicio,mitad-1,x);
  else          return busBinariaRec(v,mitad+1,fin,x);
}
```

3.- MASTER THEOREMS

Master theorem for dividing recurrence: The solution to the equation $T(n) = aT(n/b) + \Theta(n^k)$, with $a \geq 1$ and $b > 1$, is:

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^k) & \text{if } a < b^k \end{cases}$$

Master theorem for the subtracting recurrence: The solution to the equation

$$T(n) = \begin{cases} k, & \text{if } n \leq n_0; \\ aT(n-c) + \Theta(n^k), & \text{if } n > n_0 \end{cases}$$

has this cost:

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < 1 \\ \Theta(n^{k+1}) & \text{if } a = 1 \\ \Theta(a^{n/c}) & \text{if } a > 1 \end{cases}$$

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.

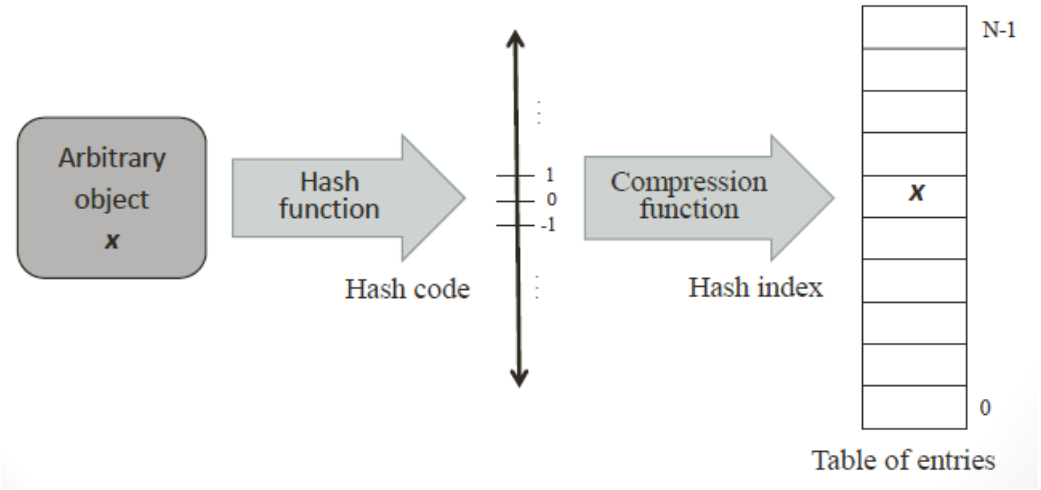
The basic operation of a **Map** is searching by key (or name) in a collection of entries.

```
public interface Map<C, V> {
  // Add the entry (c,v) and return the old value that this key had
  // (or null if it had no associated value)
  V insertar(C c, V v); // insert

  // Delete the entry with key c and return its associated value
  // (or null if there is no key with the key c)
  V eliminar(C c); // delete

  // Search for the key c and return its associated value
  // (or null if there is no key with the key c)
  V recuperar(C c); // retrieve

  // Return true the Map ia empty
  boolean esVacio(); // isEmpty

  // Return the number of entries of Map
  int talla(); // size

  // Return a List with Point of INterest with the keys of all entries
  // of the Map
  ListaConPI <C> claves(); // keys
}
```

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:

```
class EntradaHash<C, V> {
  C clave;                          // Key of the entry
  V valor;                          // Value of the entry

  public EntradaHash(C clave, V valor) {
    this.clave = clave;
    this.valor = valor;
  }
}
```

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

```
public class TablaHash<C, V> implements Map<C, V> {
  // Array of LPIs
  private ListaConPI<EntradaHash<C,V>> elArray[];
  // Number of elements stored in the table
  private int talla;

  public TablaHash(int tallaMaximaEstimada) {
    int capacidad = siguientePrimo((int)
                    (tallaMaximaEstimada/0.75));
    elArray = new LEGListaConPI[capacidad];
    for (int i = 0; i < elArray.length; i++)
       elArray[i] = new LEGListaConPI<EntradaHash<C,V>>();
    talla = 0;
  }
```

```
/** It calculates the bucket for an element with key c.
 *  First it obtains the hash value (hashCode) and
 * after its hash index
 *  @param c   Key of the element to search
 *  @return    Bucket where the element is
 */
private int indiceHash(C c) {
  int indiceHash = c.hashCode() % this.elArray.length;
  if (indiceHash < 0)
      indiceHash += this.elArray.length;
  return indiceHash;
}
```

```
// It adds the entry(c,v) and returns the old value
// of the given key (or null if the key dis not have
// any associated value)
public V insertar(C c, V v) {
  V oldValue = null;
  int pos = indiceHash(c);
  ListaConPI<EntradaHash<C,V>> bucket= elArray[pos];
  //Search for the entry of key c in the bucket
  for (bucket.inicio(); !bucket.esFin() &&
  !bucket.recuperar().clave.equals(c); bucket.siguiente());
    if (bucket.esFin()) {// Insert the entry if there is not
     bucket.insertar(new EntradaHash<C,V>(c, v));
      talla++; // Rehashing depending on LF
  } else {//If the entry was in the bucket, update its value
      oldValue = bucket.recuperar().valor;
      bucket.recuperar().valor = v;
  }
  return oldValue;
}
```

```
// It deletes the entry with key c and returns its
// associated value (or null if there is no entry
// with this key)
public V eliminar(C c) {
 int pos = indiceHash(c);
 ListaConPI<EntradaHash<C,V>> bucket = elArray[pos];
 V value = null;
 // Search for the entry of key c in the bucket
 for (bucket.inicio(); !bucket.esFin() &&
 !bucket.recuperar().clave.equals(c);bucket.siguiente());
  if (!bucket.esFin()) {// If we find it, we delete it
      value = bucket.recuperar().valor;
      bucket.eliminar();
      talla--;
  }
  return value;
}
```

```java
// It searches tfor he key c and returns its associated
// info or null an entry with such a key does not exist
public V recuperar(C c) {
  int pos = indiceHash(c);
  ListaConPI<EntradaHash<C,V>> bucket= elArray[pos];
  // Search for the entry of key c in the bucket
  for (bucket.inicio(); !bucket.esFin() &&
     !bucket.recuperar().clave.equals(c);
      bucket.siguiente());
  if (bucket.esFin()) return null; // Not found
  else return bucket.recuperar().valor; // Found
}

// It returns true if the Map is empty
public boolean esVacio() { return talla == 0; }

// It returns the number of entries in the Map
public int talla() { return talla; }
```

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.

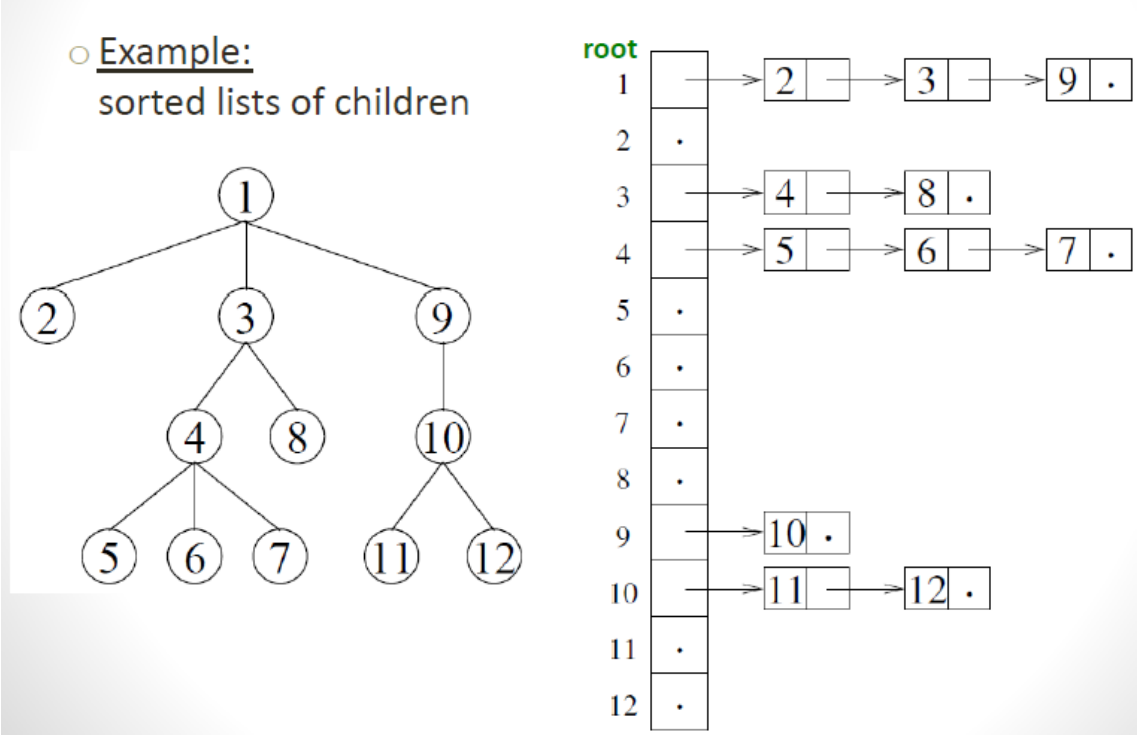
A tree is:

- 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.

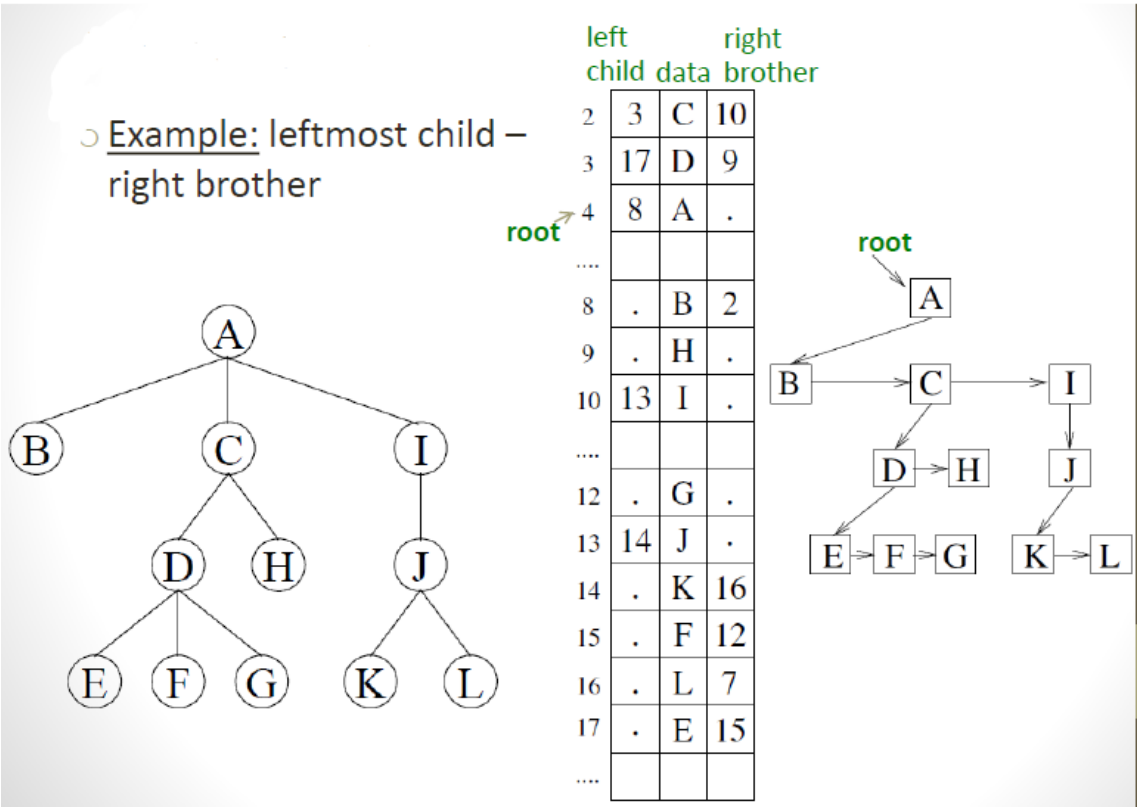
2.- GENERIC TREES

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 $\mathbf{2}^{i}$.
- In a tree of height H, the maximum number of nodes is 2^{H+1}-1.
- The maximum number of leaves is 2^{H}.
- The maximum number if nodes is 2^{H}-1.

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

- H = $\lfloor log_2 N \rfloor$.
- N = 2^{H+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 <= $\lfloor log_2 N \rfloor$ then, it is a **balanced tree**.
- 2^{H} <= N <= 2^{H+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:

```
package librerias.estructurasDeDatos.jerarquicos;
class NodoABB<E> {
   E data;                      // data
   NodoABB<E> left, right;      // children
   int size;                 // size of node (optional)
   // Constructors
   NodoABB(E data, NodoABB<E> l, NodoABB<E> r) {
      this.data = data; size = 1;
      this.left = l; this.right = r;
      if (left != null) size += left.size;
      if (right != null) size += right.size;
   }
   NodoABB(E data) {
      this.data = data; size = 1;
      this.left = this.right = null;
   }
}
```

ABB:

```
package librerias.estructurasDeDatos.jerarquicos;

public class ABB<E extends Comparable<E>> {
  // Attributes
  protected NodoABB<E> root; // Root of ABB


  /** Constructor of an empty ABB **/
  public ABB() {
    root = null;
  }
```

```java
// Search for x in ABB and returns it.
// Otherwise return null
public E search(E x) {
  NodoABB<E> node = root;
  while (node != null) {
    int resC = x.compareTo(node.data);
    if (resC == 0) return node.data;
    node = resC < 0 ? node.left : node.right;
  }
  return null;
}
```

Recursive version of search:

```java
public E search(E x) {
  return search(x, root);
}
protected E search(E x, NodoABB<E> node) { // recursive
  if (node == null) return null;           // not found
  int cmp = x.compareTo(node.data);
  if (cmp < 0) return search(x, node.left);
  else if (cmp > 0) return search(x, node.right);
  else return node.data;                   // x found
}
```

```java
// Return the number of elements in ABB
 public int size() {
   return size(root);
}

protected int size(NodoABB<E> node) {
  if (node == null) return 0;
  else return node.size;
}


public boolean isEmpty() {
  return root == null;
}
```

```java
// Update x in ABB;if x is not in ABB, insert it
public void insert(E x) {
  root = insert(x, root);
}

protected NodoABB<E> insert(E x, NodoABB<E> node) {
  if (node == null) return new NodoABB<E>(x);
  int cmp = x.compareTo(node.data);
  if (cmp < 0) node.left = insert(x, node.left);
  else if (cmp > 0) node.right = insert(x, node.right);
  else node.data = x;
  node.size = 1 + size(node.left) + size(node.right);
  return node;
}
```

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.

```java
// Return the smallest
public E retrieveMin() {
  if (root == null) return null;
  return retriveMin(root).data;
}

protected NodoABB<E> retrieveMin(NodoABB<E> node) {
  if (node.left == null) return node;
  else return retrieveMin(node.left);
}
```

```java
public E deleteMin() {
  E min = retrieveMin();
  if (min != null) root = deleteMin(root);
  return min;
}

protected NodoABB<E> deleteMin(NodoABB<E> node) {
  if (node.left == null) return node.right;
  node.left = deleteMin(node.left);
  node.size--;
  return node;
}
```

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.

```
// Delete the node with x
public void delete(E x) {
  root = delete(x, root);
}

protected NodoABB<E> delete(E x, NodoABB<E> node) {
  if (node == null) return node;     // x not found
  int cmp = x.compareTo(node.data);
  if (cmp < 0) node.left  = delete(x, node.left);
  else if (cmp > 0) node.right = delete(x, node.right);
  else {           // x found -> we delete the node
   if (node.right == null) return node.left; // 1 child
   if (node.left == null) return node.right; // 1 child
   node.data = retrieveMin(node.right).data; // 2 children
   node.right = deleteMin(node.right);
  }
  node.size = 1 + size(node.left) + size(node.right);
  return node;
}
```

```
public String preOrder() {
  return preOrder(root);
}
private String preOrder(NodoABB<E> actual) {
  if (actual == null) return "";
  return actual.data.toString() + "\n" +
   preOrder(actual.left) + preOrden(actual.right);
}
```

```
public String byLevels() {
  if (root == null) return "";
  Cola<NodoABB<E>> q = new ArrayCola<NodoABB<E>>();
  q.enqueue(root);
  String res = "";
  while (!q.isEmpty()) {
   NodoABB<E> actual = q.dequeue();
   res += actual.data.toString() + "\n";
   if (actual.left != null) q.enqueue(actual.left);
   if (actual.right != null) q.enqueue(actual.right);
  }
  return res;
}
```

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.

```
/* Return the successor of e in ABB, null otherwise */

public E successor(E e) {
  E successor = null;
  NodoABB<E> aux = this.root;
  while (aux != null) {
    int resC = aux.data.compareTo(e);
    if (resC > 0) {
      successor = aux.data;
      aux = aux.left;
    } else aux = aux.right;
  }
  return successor;
}
```

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:

Average complexity	search(x)	insert(x)	min ()	deleteMin()
Linked list / Array	Θ(N)	Θ(1)	Θ(N)	Θ(N)
LEG (sorted)	Θ(N)	Θ(N)	Θ(1)	Θ(1)
Array (sorted)	Θ(log N)	Θ(N)	Θ(1)	Θ(1)
ABB	Θ(log N)	Θ(log N)	Θ(log N)	Θ(log N)

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:

```
class EntradaMap<C extends Comparable<C>,E>
      implements Comparable<EntradaMap<C,E>> {
  C key;
  E value;
  public EntradaMap(C c, E e) {key = c; value = e;}
  public EntradaMap(C c) { this(c, null); }
  public boolean equals(Object x) {
    return ((EntradaMap<C,E>)x).key.equals(this.key);
  }
  public int compareTo(EntradaMap<C,E> x) {
    return this.key.compareTo(x.key);
  }
  public String toString() {
    return this.key + " => " + this.value;
  }
}
```

ABBMap:

```
public class ABBMap<C extends Comparable<C>,V>
        implements Map<C,V> {
  private ABB<EntradaMap<C,V>> abb;
  public ABBMap() { abb = new ABB<EntradaMap<C,V>>(); }
  public V retrieve(C c) {
   EntradaMap<C,V> e;
   e = abb.retrieve(new EntradaMap<C,V>(c));
   return e == null ? null : e.value;
  }
  public V insert(C c, V v) {
   EntradaMap<C,V> newentry = new EntradaMap<C,V>(c,v);
   EntradaMap<C,V> prev = abb.insert(newentry);
   return prev == null ? null : prev.value;
  }
  public V delete(C c) {
   EntradaMap<C,V> prev = abb.delete(new EntradaMap<C,V>(c));
   return prev == null ? null : prev.value;
  }
```

7.- THE CLASS ABBCOLAPRIORIDAD

```
public class ABBColaPrioridad<E extends Comparable<E>>
        extends ABB<E>
        implements ColaPrioridad<E> {
  public boolean esEmpty() { return super.isEmpty(); }
  public void insert (E x) { // insert with duplicates
    root = insert(x, root);
  }
  protected NodoABB<E> insert(E x, NodoABB<E> node) {
    if (node == null) return new NodoABB<E>(x);
    int cmp = x.compareTo(node.data);
    if (cmp <= 0) node.left = insert(x, node.left);
    else node.right = insert(x, node.right);
    node.size++;
    return node;
  }
}
```