EDA NOTES

TOPIC 5.- PRIORITY QUEUE AND HEAP. HEAP SORT

1.-INTRODUCTION

The Priority Queue (Cola de Prioridad) is a model for a data collection that allows to access to the element of highest priority.

```
public interface ColaPrioridad<E extends Comparable<E>> {
  // Insert x in the queue
  void insertar(E x);
  // IFF !esVacia(): if not empty return the element with highest
priority
  E recuperarMin();
  // IFF !esVacia(): return and delete the element with highest
priority
  E eliminarMin();
  // Return true if the queue is empty
  boolean esVacia();
}
```

2.-BINARY HEAP

2.1.-CHARACTERISTICS

Characteristics:

- Implementation based on an array.
- The average cost of **insertar** is a constant. The worst case is logarithmic.
- The average cost of **eliminarMin** is logarithmic. The worst case too.
- The cost of **recuperarMin** is constant.

2.2.-PROPERTIES

Structural property: a heap is a complete binary tree.

- Its height is maximum: $\lfloor log_2 N \rfloor$.
- The cost of the algorithms is in the worst case logarithmic.
- The complete binary trees allow for an array representation.

Sort property: in a min heap, the element of the node is always smaller or equal than its children.

2.3.-ARRAY REPRESENTATION OF A COMPLETE BINARY TREE

Data is stored in an **array** following the **traversal by level**. The root is in the position 1. Given the i-th node:

- Position of its left child: 2 * i (if 2 * I <= size).
- Position of its right child: 2 * i + 1 (if 2 * i + 1 <= size).
- Position of its father: i / 2 (if i != 1).

Every path from the root to a leaf is a sorted sequence. The root is the node with the smallest element. Each subtree of a Heap is also a Heap.

3.-THE CLASS MONTICULOBINARIO (HEAP)

```
public class MonticuloBinario<E extends Comparable<E>>
        implements ColaPrioridad<E> {
  // Attributes
  protected static final int CAPACIDAD_INICIAL = 50;
  protected E elArray[];
  protected int talla;

  // Constructor of an empty min-heap
  @SuppressWarnings("unchecked")
  public MonticuloBinario() {
    talla = 0;
    elArray = (E[]) new Comparable[CAPACIDAD_INICIAL];
  }
```

Steps of the method **insertar**:

1. The new element is inserted in the first available position of the array: *elArray[talla + 1]*.
2. The new element is compared with its predecessors and moved in order to have the sort property accomplished.

```
public void insertar(E x) {
  // do we have space in the array for another element?
  if (talla == elArray.length - 1) duplicarArray();
  // hole is the position where x will be inserted
  int hole = ++talla;
  // it is moved in order to have the sort property
  accomplished
  while (hole > 1 && x.compareTo(elArray[hole/2]) < 0) {
    elArray[hole] = elArray[hole/2];
    hole = hole/2;
  }
  // now we know in what position to insert the new
  element
  elArray[hole] = x;
}
```

Temporal costs of the method **insertar**:

- Worst case: the complexity is $O(\log_2 N)$ if the added element is the new minimum.
- Best case: when the element to insert is greater than its father (only one comparison).
- It has been empirically proofed that on average, 2.6 comparisons are needed to insert a new element, so it is constant complexity.

Steps of the method **eliminarMin**:

1. The minimum is in the root. The root is substituted by the last element of the Heap.
2. The new root is moved down via its children in order to accomplish the sort property.

The method **heapify** (hundir):

```
private void heapify(int hole) {
  E aux = elArray[hole];
  int child = hole * 2;
  boolean isHeap = false;
  while (child <= talla && !isHeap) {
    if (child != talla &&
        elArray[child+1].compareTo(elArray[child]) < 0)
        child++;    // We choose the smalles of the two children
    if (elArray[child].compareTo(aux) < 0) {  // we move down
      elArray[hole] = elArray[child];
      hole = child;
      child = hole*2;
    } else isHeap = true; // The sort property is accomplished
  }
  elArray[hole] = aux;
}
```

The method **eliminarMin**:

```
// IFF !esVacia(): delete and return the smallest element
public E eliminarMin() {
  E theMin = recuperarMin();
  // the root is substituted with the last element
  elArray[1] = elArray[talla--];
  // the new root is moved down (sort property)
  heapify(1);
  return theMin;
}

// IFF!esVacia(): return the smallest element
public E recuperarMin() {
  return elArray[1];
}
```

The method **buildHeap** (arreglarMonticulo): given a complete binary tree it allows to accomplish the sort property. It moves down all nodes in an inverse order than the traversal by levels. It has liner time complexity.

```
private void buildHeap() //arreglarMonticulo
{
for (int i = talla / 2; i > 0; i--)
    heapify(i);
}
```

4.- FAST SORTING WITH HEAP SORT

The cost of **HeapSort** is O(N * $\log_2$N). This algorithm is based on the properties of a heap:

1. All the elements of an array to be sorted are stored in a heap.
2. The smallest element is extracted (root) in an iterative way.

The most efficient way to insert the data of an array in a Heap is with the method **arreglarMonticulo** (buildHeap).

```
@SuppressWarnings("unchecked")     // Constructor
public MonticuloBinario(E v[]) {
  talla = v.length;                // Data is copied
  elArray = (E[]) new Comparable[talla+1];
  System.arraycopy(v, 0, elArray, 1, talla);
  buildHeap();                     // Sort property is accomplished
}
```

The cost of the constructor is O(N), where N is the size of the array.

Algorithm:

```
public class Ordenacion {
  public static <E extends Comparable<E>> void heapSort(E v[]) {
    // Creating the heap from the array
    MonticuloBinario<E> heap = new MonticuloBinario<E>(v);
    // Extracting data from the heap in a sorted way
    for (int i = 0; i < v.length; i++)
      v[i] = heap.eliminarMin();
  }
}
```

The temporal cost of this algorithm is $O(N * \log_2 N)$, and for the first k elements, $O(N + k * \log_2 N)$.

TOPIC 6.- MF-SETS

1.-INTRODUCTION

A **disjoint-set data structure** is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. Given a set C previously fixed, a relation R defined in C is a subset of the Cartesian product CxC such that aRb means (a, b) ∈ R. A relation is an equivalence relation if the following three properties are accomplished:

- Reflexive: aRa for each a ∈ C.
- Symmetric: aRb if bRa, for each a, b ∈ C.
- Transitive: aRb y bRc implies aRc, for each a, b, c ∈ C.

A set of elements can be partitioned in equivalence classes from the definition of an equivalence relation.

The problem of Union-Find (Merge-Find) has a search space the set of the possible partitions of a set C. The aim is to develop an efficient data structure in order to group together n different elements in a collection of k disjoint sets. There are two kind of operations:

- **Union (Merge)** of two disjoint sets.
- **Search (Find**) to what set a given element belongs to.

An MF-Set is a data structure of type set in which the elements are organised in disjoint sub-sets and the number of elements is fixed (no elements are added nor deleted).

```
public interface MFSet
{
 /* it returns the identifier of the set the element x belongs to */

 public int find (int x);

 /* it merges the sets x and y belong to */

 public void merge (int x, int y);
}
```

2.-REPRESENTATION OF MF-SETS

As an array:

- M is an array of int of size n = |C|.
- M[i] indicates directly the set of class the element i belongs to.
- The complexity of **find(i)** is O(1).
- The complexity of **merge(i, j)** is O(n).

As disjoint-set forests:

- Every **sub-set** is a **tree**:
  - The nodes of the tree are elements of the set.
  - **Every node refers to its father**.
  - **The root of the tree can be used to represent the partition** or class.
- The **MF-Set** is represented as a **collection of trees**, a **forest**.
- The trees are not necessarily binary trees but the representation is easy and only a reference to the father is needed: M = array[n] of int.
  - M[x] is the father of the element x.
  - If M[x] = x, x is the root of the tree of the forest.

Operations with MF-Sets:

```
/* MFSet where the operations find and merge were not optimised  */

public class ForestMFSet implements MFSet {
 protected int size;
 protected int n_partitions;
 protected int mfset[];

 /* Constructor: it creates a MFSet of a given size and it initialises it  */

 public ForestMFSet (int n) {
  size= n;
  n_partitions = size;

  mfset = new int[size];
  for (int i = 0;i<size;++i) mfset[i] = i;
 }
```

```
/* it returns the identifier of the set the element x belongs to */
public int find (int x) {
   while (mfset[x] != x) x = mfset[x];
   return x;
  }

  /* it merges the sets x and y belong to*/
  public void merge (int x, int y) {
    int rx = find(x);
    int ry = find(y);

    if(rx != ry) {
      n_partitions--;        // partitions are merged
      mfset[rx] = ry;        // x is appended to y
    }
  }
}
```

A sequence of m operations Merge and Find has a complexity O(m * n) in the worst case.

3.-IMPROVING EFFICIENCY

Complexity can be improved if the height of the tree is reduced: m operations Merge and Find will have a quasi linear complexity with m:

- **Union by rank**.
- **Path compression**.

With these improvements the complexity of the operations is practically constant.

3.1.-UNION BY RANK

Merge is done in a way that the root of the tree whose height is smaller becomes the child of the tree whose height is higher. If the heights are different, the height after the merge is the height of the higher tree. In case that the merge is with two trees of the same height, the final height is increased by 1. The value of the height of each tree of the forest is kept, for instance, in the array, the value associated to the root is usually a negative number.

If MF-Set[i] < 0, i is the root of the tree and |MF-Set[i]| - 1 is the height of the tree.

```
/* MFSet where the complexity of merge and find operations has been
improved with union by rank (based on height). The height will be
never greater than O(log(n)) */

public class MFSetUnionByRank extends ForestMFSet {

 /* the constructor creates a MFSet of a given size and initialises it */
 public MFSetUnionByRank(int n) {
  size = n;
  n_partitions = size;

  mfset = new int[size];
  for(int i = 0;i<size;++i)
   mfset[i] = -1;  // the negative value indicates the height for a root
 }

 /* It returns the identifier of the set the element x belongs to */
public int find (int x) {
  while (mfset[x] >= 0)
   x = mfset[x];
  return x;
 }

 /** It merges the sets x and y belong to; the set with smaller height is
merged with the other one */
 public void merge (int x, int y) {
  int rx = find(x); int ry = find(y);
  if (rx != ry) {
   n_partitions--;                                  // union of partitions
   if (mfset[rx] == mfset[ry]) {                    // same height
    mfset[rx] = ry;                                 //x is appended to y
    mfset[ry]--;                                    // the height of y is updated
   }
   else if (mfset[rx] < mfset[ry]) mfset[ry] = rx;  //y is appended to x
   else mfset[rx] = ry;                             // x is appended to y
  }
 }
}
```

3.2.-PATH COMPRESSION

The effect of the path compression on Find is that each node of the path from x to the root refers directly to the root of the tree.

```
/* MFSet where the complexity of the operations of the class
MFSetNormal is optimised reducing the height, when the find
method is invoked */

public class MFSetPathCompression extends ForestMFSet {

 /* The constructor creates a MFSet of a given size and initialises it
 public MFSetPathCompression(int n) {
  super(n);
 }

 /* It returns the identifier of the set the element x belongs to ;
moreover, all the elements of a set are directly linked to the root */

 public int find (int x) {
  int rx = x;

   while (mfset[rx] != rx) rx = mfset[rx];   // it finds the root of the set

   while (mfset[x] != rx) {
    int tmp = x;
    x = mfset[x];
    mfset[tmp] = rx;                        // it is directly linked to the root
   }

   return rx;
 }
}
```

3.3.- COMBINING STRATEGIES

It may happen that path compression diminishes the height of the tree. In this case, the height stored in the array does not necessarily represent the real height of the tree and it is an upper-bound of the real height O(log n) that in practice is much less due to the path compression.

```
 /* It returns the identifier of the set the element x belongs to ; all the
elements of the set of x are linked directly with the father */

 public int find (int x) {
  int rx = x;

  // it finds the root of the set
  while (mfset[rx] >=  0) rx = mfset[rx];

  while (mfset[x] != rx) {
   int tmp = x;
   x = mfset[x];
   mfset[tmp] = rx;  // ithe element is linked directly with the root
  }
  return rx;
 }
}
```

TOPIC 7.- GRAPHS

1.-INTRODUCTION

Binary relation between data of the collection:

- A relation R over a set S is defined as a set of pairs (a, b) / a, b ∈ S
- If (a, b) ∈ R, it can be written as "a R b" and it denotes that a is related with b.

A **directed graph** (dg) is a pair G = (V, E):

- V is a finite set of **vertices** (or nodes).
- E is a set of directed **edges**, where an edge is an ordered pair of vertices (u, v): u -> v.

A **non directed graph** (ndg) is a pair G = (V, E):

- V is a finite set of **vertices**.
- E is a set of non directed **edges**, where an edge is a pair of non directed vertices (u, v) = (v, u), u != v: u – v.

A **labelled graph** is a graph G = (V, E) where a function is defined f: E -> L, with L is a set whose components are called **labels**.

A **weighted graph** is a labelled graph with real numbers.

Being G = (V, E) a graph, if (u, v) ∈ E, we say that the vertex v is adjacent to the vertex u. In a non directed graph the relation is symmetrical.

The **degree of a vertex** in a non directed graph, is the number of its incident edges (or adjacent vertices). The degree of a vertex in a directed graph is the sum of its outdegree and its indegree. The **degree of a graph** is the maximum degree of its vertices.

A **path of length** k from u to u' in a graph G = (V, E) is a sequence of vertices $<v_0, v_1,\ldots,v_k>$ such that:

- V_0 = u and v_k = u'.
- ∀ i : 1…k : $(v_{i-1}, v_i) \in E$.
- The length k of the path is the number of edges.
- The length of the path with weights is the sum of the weights of the edges of the path.

If there exists a path P from u to u', we say that u' is **reachable** from u via P.

A **cycle** is a path $<v_0, v_1,\ldots,v_k>$:

- Starting and ending in the same vertex ($v_0 = v_k$).
- Containing at least an edge.

A path or cycle is **simple** if all its vertices are different. A **loop** is a cycle of length 1. A graph is **acyclic** if it does not contain cycles.

The **connected components** in a non directed graph are the equivalence classes of vertices under the relation "being reachable". A non directed graph is connected if ∀ u, v ∈ V, v is reachable from u. That is if it has just one connected component.

The **strongly connected components** in a directed graph are the equivalence classes of vertices under the relation of "being mutually reachable". A directed graph is strongly connected if ∀ u, v ∈ V, v is reachable from u.

2.-REPRESENTATION OF GRAPHS

There exist two forms for representing a graph:

- If the graph is **disperse** ($|E| <<< |V|^2$): **adjacency lists**.
- If the graph is **dense** ($|E| \approx |V|^2$): **adjacency matrix**.

2.1.- ADJACENCY MATRIX

A graph G = (V, E) is represented as a matrix of |V|x|V| of elements of type boolean:

- If (u, v) ∈ E -> G[u, v] = true, otherwise, false.
- Spatial cost $O(|V|^2)$.
- Time for access O(1).

2.2.-ADJACENCY LISTS

A graph G = (V, E) is represented as an **array** of |V| **lists** of vertices:

- G[v], v ∈ V, is the list of the adjacent vertices to v.

2.3.-IMPLEMENTATION

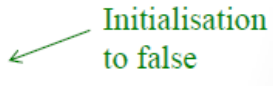
```
class Adjacent {
  int target; // Vertex target of the edge
  double weight;     // weight of the edge
  public Adjacent(int d, double p){ target = d; weight = p;}
  public String toString(){ return target + "(" + weight + ")";}
}
```

```
public abstract class Graph {
  // It returns the number of vertices of the graph
  public abstract int numVertices();

  // It returns the number of edges of the graph
  public abstract int numEdges();

  // It checks whether the edge (i,j) exists
  public abstract boolean existEdge(int i, int j);

  // It retrieves the weight of the edge (i,j)
  public abstract double weightEdge(int i, int j);

  // It returns a list with the adjacent vertices of vertx i
  public abstract ListWithPI<Adjacent> adjacentsOf(int i);

// It adds the edge (i,j) to a non weighted graph
public abstract void insertEdge(int i, int j);

// It adds the edge (i,j) with weight p to a weighted graph
public abstract void insertEdge(int i, int j, double p);
```

3.-GRAPH TRAVERSALS

Depth First Search is like the **PreOrder** traversal (root, left, right), but vertices do not have to be visited twice.

```
public abstract class Graph {
  // DFS needs the following attributes
  protected boolean visited[]; // To not repeat vertices
  protected int orderVisit;  // Order of visit of vertices

  // DFS returns an array with visited vertices
    public int[] toArrayDFS() {
    int res[] = new int[numVertices()];
    visited = new boolean[numVertices()];
    orderVisit = 0;
    for (int  i = 0; i < numVertices(); i++)
      if (!visited[i]) toArrayDFS(i, res);
    return res;
  }
```



```
// Recursive method for DFS
protected void toArrayDFS(int source, int res[]) {
  // Source vertex is added and marked as visited
  res[orderVisit++] = source;
  visited[source] = true;
  // Adjacent vertices of source are visited
  ListaConPI<Adjacent> l = adjacentsOf(source);
  for (l.inicio(); !l.esFin(); l.siguiente()) {
    Adjacent a = l.recuperar();
    if (!visited[a.target]) toArrayDFS(a.target, res);
  }
}
```

Breadth First Search is like the **traversal by levels** of a tree.

```
public abstract class Graph {
  ... // Apart from the attributes visited and orderVisit,
      // BFS needs an auxiliar queue
      // (the algorithm is iterative)
  protected Queue<Integer> q;

  // BFS
  public int[] toArrayBFS() {
    int res[] = new int[numVertices()];
    visited = new boolean[numVertices()];
    orderVisit = 0;
    q = new ArrayQueue<Integer>();
    for (int  i = 0; i < numVertices(); i++)
      if (!visited[i]) toArrayBFS(i, res);
    return res;
  }
```

```
protected void toArrayBFS(int source, int res[]
  res[orderVisit++] = source;
  visited[source] = true;
  q.encolar(source);
  while (!q.esVacia()) {
    int u = q.desencolar().intValue();
    ListaConPI<Adjacent> l = adjacentsOf(u);
    for (l.inicio(); !l.esFin(); l.siguiente())
      Adjacent a = l.recuperar();
      if (!visited[a.target]) {
        res[orderVisit++] = a.target;
        visited[a.target] = true;
        q.encolar(a.target);
      }
    }
  }
}
```

4.-IMPLEMENATION

```
// Implementation of a Directed Graph
public class GraphDirected extends Graph {

  // Nomber of Vertices and Edges
  protected int numV, numE;
   // The array of list fo adjeacents of each vertex
  protected ListaConPI<Adjacent> elArray[];

  // Construction of the graph (number of vertices)
  public GraphDirected(int numVertices) {
    numV = numVertices;
    numE = 0;
    elArray = new ListaConPI[numVertices];
    for (int i = 0; i < numV; i++)
      elArray[i] = new LEGListaConPI<Adjacent>();
  }
```

```
// Getters (in Spanish: consultores)
public int numVertices() { return numV; }
public int numEdges()  { return numE; }

public ListaConPI<Adjacent> adjacentsOf(int i) {
  return elArray[i];
}

public boolean existEdge(int i, int j) {
  ListaConPI<Adjacent> l = elArray[i];
  boolean isin = false;
  for (l.incio(); !l.esFin()&& !isin; l.siguiente())
    if (l.recuperar().target == j) isin = true;
  return isin;
}
public double weightEdge(int i, int j) {
  ListaConPI<Adjacent> l = elArray[i];
  for (l.inicio(); !l.esFin(); l.siguiente())
    if (l.recuperar().target == j)
      return l.recuperar().weight;
  return 0.0;
}
// Insert edge
public void insertaEdge(int i, int j) {
  insertEdge(i, j, 1.0);  // Its weight is 1.0 by default
}
public void insertEdge(int i, int j, double p) {
  if (!existEdge(i,j)) {
    elArray[i].insert (new Adjacentj,p));
    numE++;
  }
}
```

5.-MINIMUM SPANNING TREE (KRUSKAL)

An undirected graph is connected if each pair of vertices is connected via a path. An acyclic undirected and connected graph is a tree. A spanning tree of a graph (V, E) is a tree (V', E') such that:

- V' = V.

- E’ ⊆ E.

Kruskal’s algorithm:

1. Store the edges in a priority queue.
2. Start from a graph without edges.
3. While |E| < |V| - 1 do:
    - Retrieve and delete from the priority queue the edge with the smallest cost.
    - Insert the edge in the graph if not cycles occur.

```
public class Edge
        implements Comparable< Edge> {
  int source, target;
  double weight;
  public Edge(int o, int d, double p) {
    source = o;
    target = d;
    weight = p;
  }
  public int compareTo(Edge p) {
    if (weight < p. weight) return -1;
    return weight > p. weight ? 1 : 0;
  }
}
```

```
public Edge[]  Kruskal() {
 Edge[] res = new Edge[numVertices()-1];
 PriorityQueue<Edge> qPrior = new MinHeap<Edge>();
 MFset m = new ForestMFset(numV);
 // The priority queue with the edges of the graph is created
 for (int i = 0; i < numVertices(); i++){
  ListaConPI<Adjacent> l = adjacentsOf(i);
  for (l.inicio(); !l.esFin(); l.siguiente()){
      Adjacent a = l.recuperar();
      qPrior.insert(new Edge (i, a.target, a.weight));
  }
 }
 int numE = 0; // Construction of the minimum spanning tree
 while (numE < numVertices() - 1 && !qPrior.isEmpty()) {
   Edge a = qPrior.deleteMin();
   if (m.find(a.source) != m.find(a.target)) {
     m.merge(a.source, a.target); res[numE++] = a;
   }
 }
 return res;
}
```

6.-SHORTEST PATH PROBLEM (DIJKSTRA)

The **weight of a path** is the sum of the weights of the edges that it passes through.

Dijkstra calculates the shortest paths from a vertex to the rest of vertices. It requires the weights of the edges to be positive. It stores the information in two arrays:

- **distanceMin**: it stores the minimum distance from the source vertex to the rest of vertices.
- **pathMin**: for each vertex it stores the previous vertex in the shortest path from the source vertex.

Pseudocode:

```
void dijkstra(int vSource) {
  pathMin[v] = -1,    ∀v∈V      // Initialisations
  distanceMin[v] = ∝, ∀v∈V
  distanceMin[vSource] = 0
  qPrior ← (vSource,0)
  while qPrior ≠ ∅{  // While there are vertices to explore
    v ← qPrior   // Next vertex to explore: the one with shortest distance
    if !visited[v]{   // Repetitions are avoided
      visited[v] = true
      for each a ∈ adjacentsOf(v){// The vertices are explored
        w = a.target                    // adjacent of v
        weightW = a.weight
        // Is it the best way to reach w through v?
        if distanceMin[w] > distanceMin[v] + weightW {
          distanceMin[w] = distanceMin[v] + weightW;
          parthMin[w] = v;
          qPrior ← (w, distanceMin[w])
}}}}}
```

7.-TOPOLOGICAL ORDERS

A **topological sorting** is a linear sorting of a given directed acyclic graph, preserving the original partial sorting.

```
// It returns an array with the topological sorting of the codes
  of the vertices
public int[] toArrayTopologic() {
  visited = new boolean[numVertices()];
  Pila<Integer> pVExplored = new ArrayPila<Integer>();
  // Traversal of vertices
  for (int vSource = 0; vSource < numVertices(); vSource++)
    if (!visited[vSource])
        topologicalSorting(vSource, pVExplored);
  // Result of topological sorting is copied in an array
  int res[] = new int[numVertices()];
  for (int i = 0; i < numVertices(); i++)
    res[i] = pVExplored.desapilar();
  return res;
}
protected void topological sorting(int source,
                Stack<Integer> pVExplored) {
   visited[source] = true;
   // The adjacent vertices are explored
   ListaConPI<Adjacent> aux = adjacentsOf(source);
   for (aux.inicio(); !aux.esFin(); aux.siguiente()) {
     int target = aux.recuperar().target;
     if (!visited[target])
        topologicalSorting(target, pVExplored);
   }
   // the vertex is pushed
  pVExplored.apilar(source);
}
```

$$T_{\text{topologicalSorting}}(|V|, |E|) \in O(|V| + |E|)$$