1. Arrays y Listas

1.1. Ordenamiento

Cuando necesite ordenar un vector o una lista, utilice los métodos .sort() que tiene Java. El algoritmo que utilizan es QuickSort y su tiempo de ejecución es de $O(n \log n)$.

```
1  public static void main(String[] args) {
2    String v[] = {"hola", "z", "mundo", "0", "aaaa", "
         ↪ quick", "sort", "1a"};
3
4    ArrayList<Integer> a = new ArrayList<Integer>();
5    a.add(9); a.add(0); a.add(-1); a.add(25); a.add(7); a
         ↪ .add(1);
6
7    //Ambos utilizan QuickSort
8    //Collections.sort() tambien ordena LinkedList
9    Arrays.sort(v);
10   Collections.sort(a);
11
12   for(int i=0; i<v.length; i++){
13     System.out.print(v[i] + " ");
14   }
15
16   //Esto llama la funcion .size() y .get() cada vez
17   //En ArrayList .get() es O(1) pero en LinkedList es O
         ↪ (n),
18   //por lo cual hacer esto es fatal
19   for(int i=0; i<a.size(); i++){
20     System.out.print(a.get(i) + " ");
21   }
22
23   //Otra forma de hacerlo, pero no se tiene acceso a
         ↪ los indices de cada elemento
24   for(int k : a){
25     System.out.print(k + " ");
26   }
27 }
```

2. Mapas

Guardan pares (clave, valor). El HashMap no pone las claves en ningún orden en particular. TreeMap ordena las claves de acuerdo a su orden natural. LinkedHashMap pone las claves en el orden en que se ingresen.

Las operaciones .put(), .get() y .containsKey() son $O(1)$ en HashMap y LinkedHashMap, y $O(\log n)$ en TreeMap.

```
1  public static void main(String args[]){
2    HashMap<String, Integer> map = new HashMap<String,
         ↪ Integer>();
3    //TreeMap<String, Integer> map = new TreeMap<String,
         ↪ Integer>();
4    //LinkedHashMap<String, Integer> map = new
         ↪ LinkedHashMap<String, Integer>();
5
6    String s = "tres tristes tigres tragaban trigo en un
         ↪ trigal en tres tristes trastos";
7    String palabras[] = s.split(" ");
8
9    for(int i=0; i<palabras.length; i++){
10     if(!hash.containsKey(palabras[i])){
11       hash.put(palabras[i], 1);
12     }else{
13       hash.put(palabras[i], hash.get(palabras[i])+1);
14     }
15   }
16
17   //Obtener un elemento
18   System.out.println(hash.get("tres"));
19
20   //Recorrer el mapa
21   for(Entry<String, Integer> e : hash.entrySet()){
22     System.out.println(e.getKey() + ": " + e.getValue
         ↪ ());
23   }
24 }
```

3. Grafos

3.1. BFS y DFS

Recorren un grafo a partir de un nodo origen y visitan todos los nodos alcanzables desde éste. El siguiente ejemplo está con DFS pero funciona igual con BFS.

Ambos algoritmos tienen un tiempo de ejecución de $O(n+m)$ donde n es el número de nodos y m es el número de aristas del grafo.

```
static ArrayList<Integer> g[];
static boolean seen[];

public static void main(String[] args) {
  int nodes = 10;

  seen = new boolean[nodes];

  g = new ArrayList[nodes];
  for(int i = 0; i < nodes; i++){
    g[i] = new ArrayList<Integer>();
  }

  int s = 0;

  //Visita SOLO los nodos que son alcanzables desde el
      ↪ nodo 's'
  dfs(s);

  //Con el vector 'seen' vemos cuales son estos nodos
  for(int i=0; i<nodes; i++){
    if(seen[i]){
      //'i' es alcanzable desde 's'
    }
  }

  //Si queremos visitar todos los nodos
  for(int u=0; u<nodes; u++){
    if(!seen[u]){
      //Si no hemos visitado 'u', hacer DFS en 'u'
      //Esto visitara todos los nodos alcanzables desde
          ↪ 'u'
      dfs(u);
    }
  }
}

private static void dfs(int source){
  seen[source] = true;
  int adyLen = g[source].size();
  for(int i=0; i<adyLen; i++){
    int v = g[source].get(i);
    if(!seen[v]){
      dfs(v);
    }
  }
}

private static void bfs(int source){
  Queue<Integer> queue = new LinkedList<Integer>();

  seen[source] = true;
  queue.add(source);

  while(!queue.isEmpty()){
    source = queue.poll();
    int adyLen = g[source].size();
    for(int i=0; i<adyLen; i++){
      int v = g[source].get(i);
      if(!seen[v]){
        seen[v] = true;
        queue.add(v);
      }
    }
  }
}
```

3.2. Shortest Hop

Modificación de BFS que calcula el camino más corto desde un nodo origen 'S' a todos los demás. Sólo funciona cuando el peso de todas las aristas es 1. Su tiempo de ejecución es el mismo de BFS: $O(n+m)$.

```
static ArrayList<Integer> g[];
static boolean seen[];
static int dist[];

```

```
public static void main(String[] args) {
  int nodes = 10;

  seen = new boolean[nodes];
  dist = new int[nodes];

  g = new ArrayList[nodes];
  for(int i = 0; i < nodes; i++){
    g[i] = new ArrayList<Integer>();
  }

  int s = 0;
  shortestHop(s);
  //Despues de llamar este metodo, en
  //dist[i] esta la distancia mas corta (s,i)
}

public static void shortestHop(int s){
  int n = g.length;

  //Distancia "infinita" hacia todos los nodos
  //Distancia 0 hacia el nodo de origen
  for(int i=0; i<n; i++){
    dist[i] = Integer.MAX_VALUE;
  }
  dist[s] = 0;

  //BFS "modificado"
  LinkedList<Integer> queue = new LinkedList<Integer>()
    ↪ ;

  seen[s] = true;
  queue.add(s);

  while(!queue.isEmpty()){
    s = queue.poll();
    int adyLen = g[s].size();
    for(int i=0; i<adyLen; i++){
      int w = g[s].get(i);
      if(!seen[w]){
        seen[w] = true;
        queue.add(w);
        //Lo unico que cambia es que se calcula el dist
          ↪ [w]
        dist[w] = dist[s] + 1;
      }
    }
  }
}
```

3.3. Ordenamiento Topológico

Todo grafo dirigido acíclico (DAG) tiene un ordenamiento topológico. Esto significa que para todas las aristas (u,v), 'u' aparece en el ordenamiento antes que 'v'. Visualmente es como si se pusieran todos los nodos en línea recta y todas las aristas fueran de izquierda a derecha, ninguna de derecha a izquierda. En realidad es una modificación de DFS y su tiempo de ejecución es el mismo: $O(n+m)$. El método retorna falso si detecta un ciclo en el grafo.

```
static ArrayList<Integer> g[];
static int seen[];
static LinkedList<Integer> topoSort;

public static void main(String[] args) {
  int nodes = 10;

  seen = new int[nodes];

  g = new ArrayList[nodes];
  for(int i = 0; i < nodes; i++){
    g[i] = new ArrayList<Integer>();
  }

  topoSort = new LinkedList<Integer>();

  boolean sinCiclo = true;

  //Es necesario hacer el ciclo para visitar todos los
    ↪ nodos
  for(int u=0; u<nodes; u++){
    if(seen[u] == 0){
      sinCiclo = dfs(u);
    }
  }
```

```
25
26      if(sinCiclo){
27        //La lista 'topoSort' contiene los nodos en su
              ↪ orden topologico
28      }else{
29        //Hay un ciclo
30      }
31    }
32
33    private static boolean dfs(int u){
34      //DFS "modificado" para hacer ordenamiento topologico
35      //Se marca 'u' como 'gris'
36      seen[u] = 1;
37      int adyLen = g[u].size();
38      boolean sinCiclo = true;
39      for(int i=0; i<adyLen; i++){
40        int v = g[u].get(i);
41        if(seen[v] == 0){
42          return dfs(v);
43        }else if(seen[v] == 1){
44          //Hay un ciclo, retorna falso
45          sinCiclo = false;
46        }
47      }
48      //Se agrega el nodo 'u' al inicio de esta lista y se
            ↪ marca 'negro'
49      seen[u] = 2;
50      topoSort.addFirst(u);
51      return sinCiclo;
52    }
```