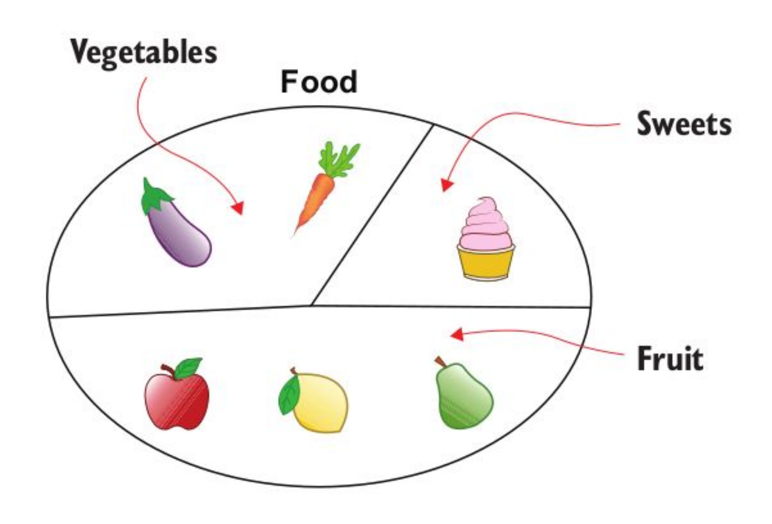
Disjoint sets
Sub-linear time processsing
Procesamiento de tiempo sublineal

By Pedro Méndez José Manuel

Disjoint set: Conjuntos disjuntos



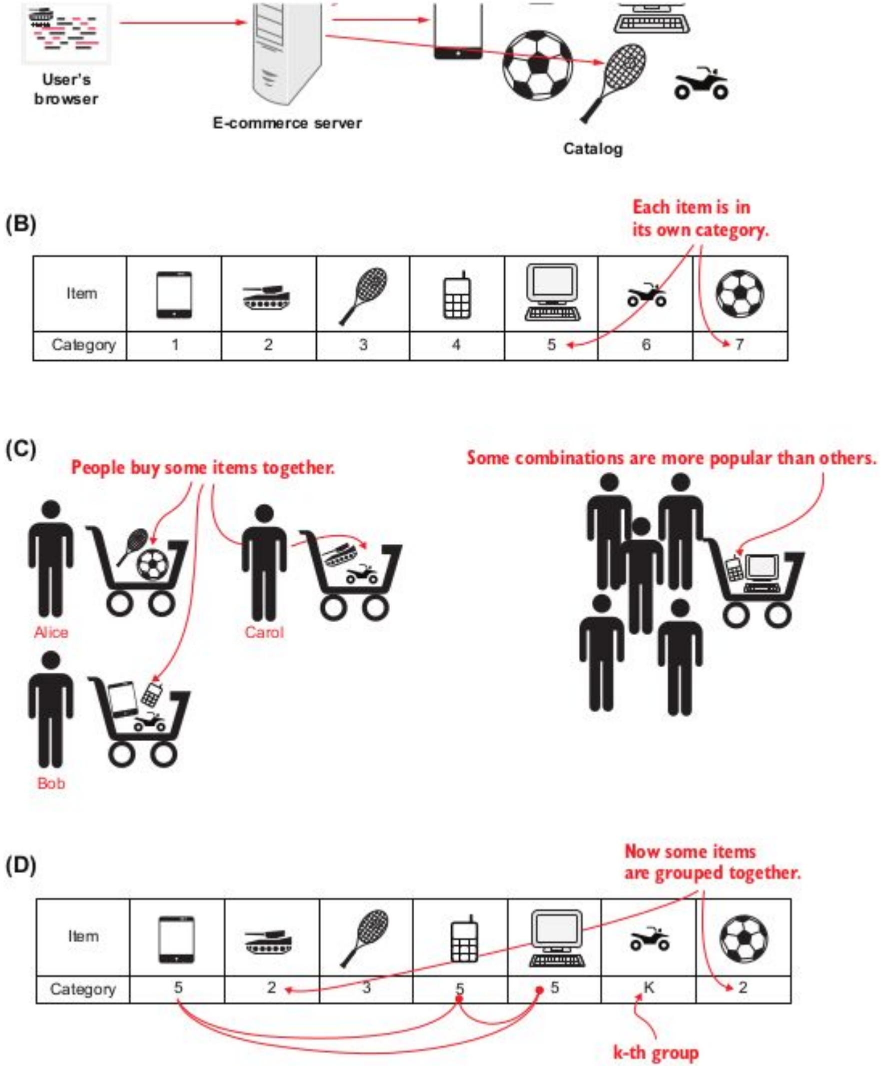
Utilizaremos un conjunto disjunto cada vez que queramos realizar una partición de un conjunto inicial (U) de objetos en grupos disjuntos.

Un conjunto disjunto es aquel que no comparte elementos con algún otro conjunto, teniendo así subconjuntos sin ningún elemento del universo en común entre ellos.

El problema de los subconjuntos distintos como recomendaciones no personalizadas

¿Recomendaciones Personalizadas? Si, lo que hace Twitta, YT, Netflix, etc… para mantenerte enganchado[ojo ahí ;)], esas recomendaciones que se dirigen a clientes individuales a partir de la información/datos previamente conocidos como compras anteriores o metadatos que muestran similitudes con otros usuarios.

Las **recomendaciones no personalizadas** son aquellas que no se dirigen a tí o alguien en particular, si no a todos (los consumidores, clientes) en general, si no tenemos ningún dato podemos hacer las asociaciones codificarlas o basadas en acciones realizadas por otros clientes.



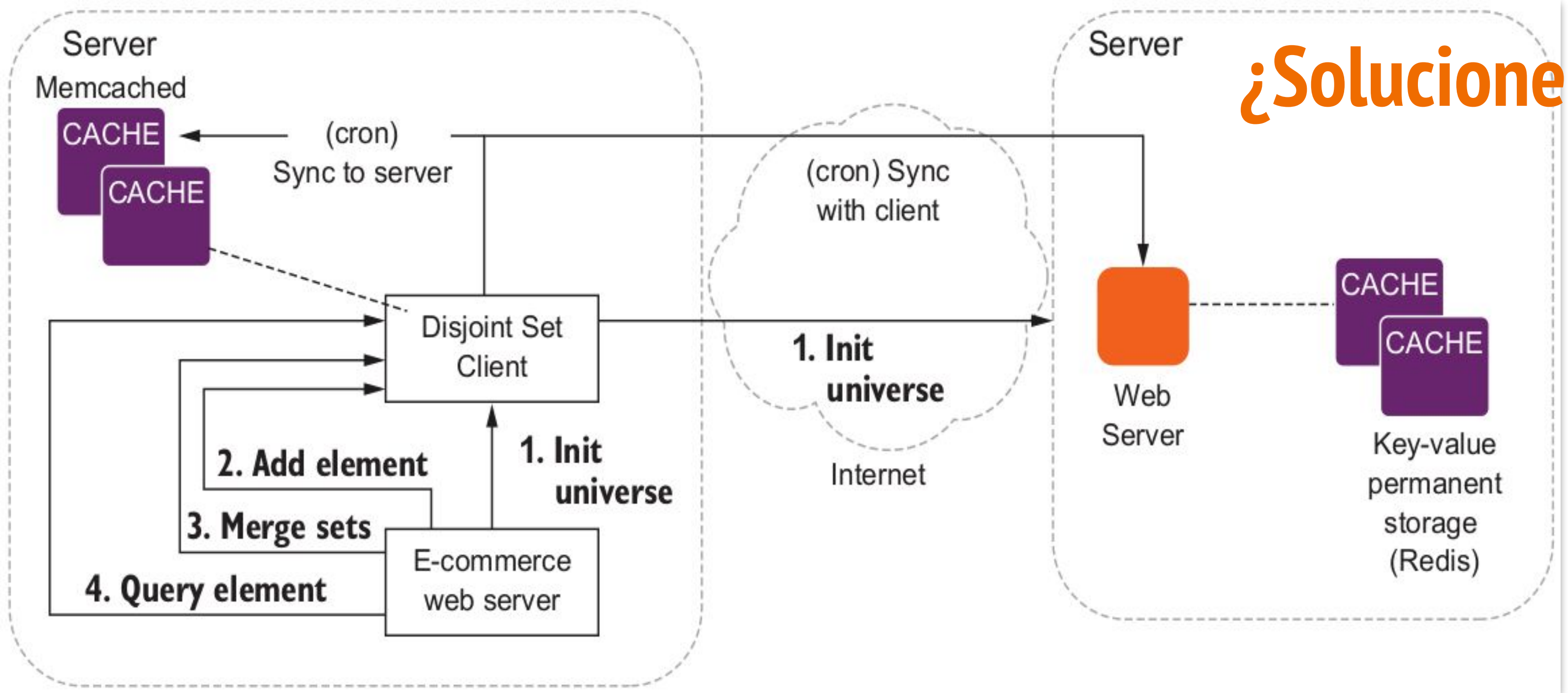
Para simplificar las cosas imaginemos que establecemos la regla: que si un artículo X y un artículo Y se compran juntos más de un umbral fijo durante la última hora, sus dos categorías se fusionan.

Por ejemplo, si un cliente compra un producto X, podemos sugerirle un artículo al azar de la misma categoría.

El punto de partida es que tenemos que partir de un enorme conjunto de elementos y dividirlo en grupos separados, osea, en grupos disjuntos.

Y por supuesto: **podremos añadir nuevos elementos** a nuestro catálogo todo el tiempo y tener **relaciones dinámicas**, por lo que tendremos que ser capaces de actualizar tanto la lista de artículos como los grupos.

¿Soluciones? 0.0



Antes de abordar nuestra posible solución es importante mencionar que nos limitaremos al caso agregativo, es decir, dos particiones pueden fusionarse en un conjunto más grande pero no lo contrario.

Necesitaremos escribir una clase que se encargue de todo el problema, llevando la cuenta de a qué subconjunto pertenece cada elemento y encapsulando toda la lógica en ella. Sin embargo primero tenemos que hablar sobre la durabilidad de nuestros datos ya que es importante en este caso.

Dependiendo del tamaño del catálogo, podríamos incluso encajar una estructura de datos de este tipo en la memoria, pero vamos a suponer, en cambio, que configuramos un servicio REST basado en un almacenamiento persistente tipo Memcached(Un almacén de claves-valores (no-SQL) utilizado como sistema de caché de objetos distribuidos.).

Estructura de Datos API: Disjoint Set y el contrato con el cliente.

Abstract data structure:

API

```
class DisjointSet {
  init(U);
  findPartition(x);
  merge(x, y);
  areDisjoint(x,y);
}
```

1. Un conjunto disjunto guarda las relaciones mutuas entre los elementos del universo U.
2. La relación ® está definida por el cliente.
3. ® tiene las propiedades reflexiva, simétrica y transitiva.

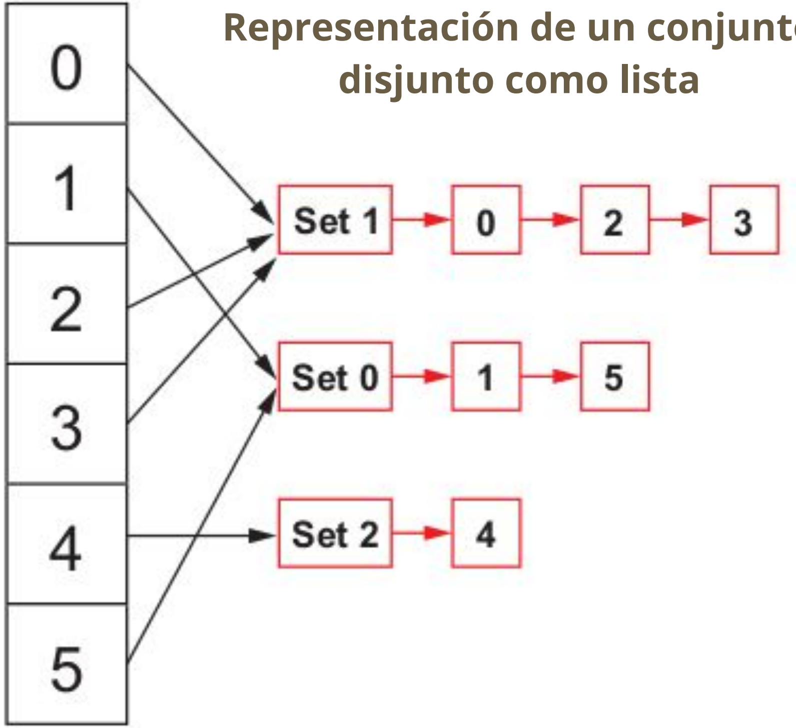
Algunas restricciones y **garantías**:

1. El universo U, el conjunto de todos los elementos posibles, es conocido de antemano y estático.
2. También asumimos que inicialmente cada elemento está en su propia partición.
3. Asumimos que los elementos de nuestro Universo U son los enteros entre 0 y n-1.

1. **Se pueden relacionar 2 elementos cualesquiera.**
2. **Si dos elementos se fusionan estos pertenecerán al mismo conjunto disjunto.**
3. **- Si tenemos los elementos x1,x2,..,xn, si fucionas x1 con x2, x2 con x3, y así sucesivamente, al final tendríamos que todos los elementos estarán en la misma partición**
4. **Si dos elementos no están en la misma partición, entonces no existe otro elemento que pertenezca a los conjuntos disjuntos de ambos elementos.**

Naïve Solution como primera solución

Representación de un conjunto disjunto como lista



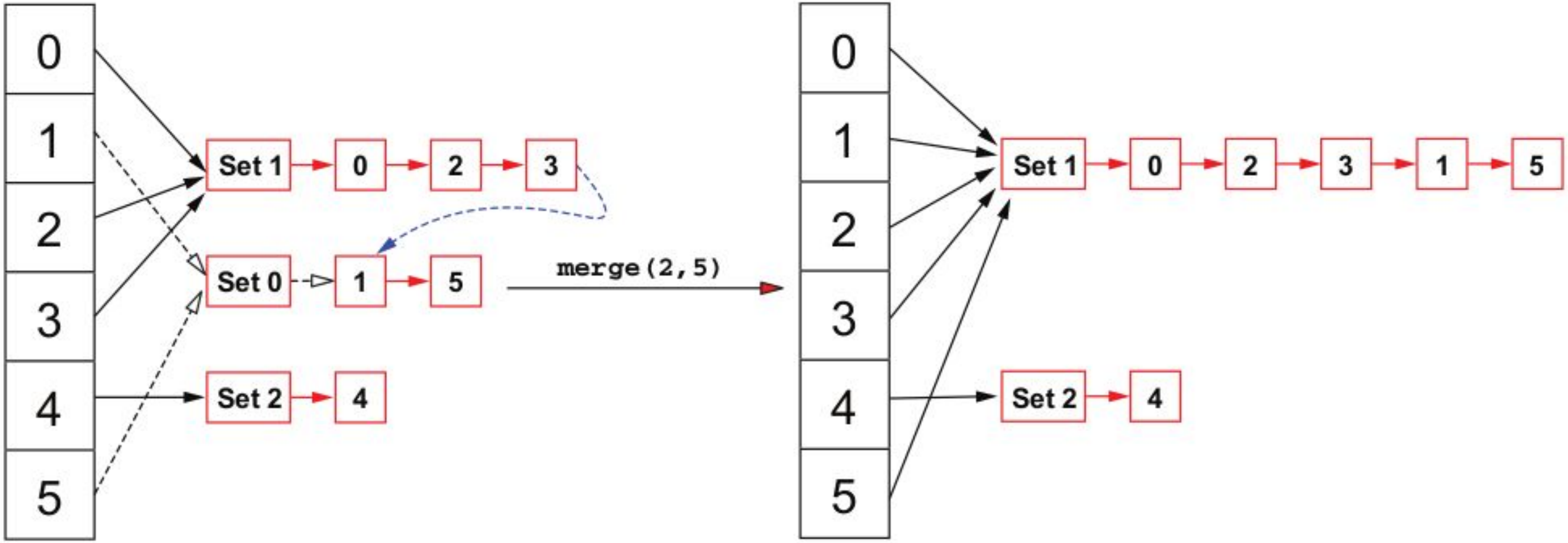
La solución más rápida que podemos implementar es representar cada partición como una lista ligada, por c/elemento en nuestro arreglo necesitaremos mantener la pista del apuntador de la cabeza de la lista.

Para verificar que dos elementos estén en la misma partición solo necesitamos revisar si los dos están en la misma lista.

Para fusionar 2 particiones solo actualizamos el último apuntador de una lista a la cabeza de la otra lista.

Naïve Solution: Ejemplo de un merge

Fusión de 2 particiones(conjuntos disjuntos)



Naïve Solution: Constructor & Add

```
class DisjointSet
  #type HashMap[Element, List[Element]]
  partitionsMap

function DisjointSet(initialSet=[])
    this.partitionsMap ← new HashMap()
    for elem in initialSet do
      throw-if (elem == null or partitionsMap.has(elem))
      partitionsMap[elem] ← new Set(elem)
```

The constructor takes a list of elements as argument, but by default initializes the disjoint set with an empty set.

Creates a new map from elements to sets

Goes over each element in the argument list

Throws an exception if the element is `null` or a duplicate

Adds a mapping between current element and a new singleton set containing current element only

Takes an element and returns `true` iff[8] the element was added successfully, `false` otherwise

```
function add(elem)
  throw-if elem == null
  if partitionsMap.has(elem) then
    return false
  partitionsMap[elem] ← new Set(elem)
  return true
```

Checks that the element is valid

If the element is already in the data structure, returns `false` without updating anything

Otherwise just adds a mapping between the element and a newly created singleton[9] set and returns `true`

Naïve Solution: findPartition & areDisjoint

Takes an element and returns a `Set`, the partition (aka disjoint set) to which the element belongs

Checks that the element is valid

Returns the `Set` containing the argument

```
function findPartition(elem)
  throw-if (elem == null or not partitionsMap.has(elem))
  return partitionsMap[elem]
```

Takes two elements and returns `true` iff the elements are valid but don't belong to the same partition, `false` iff the elements are valid but do belong to the same partition. Notice that if either element is `null` or hasn't been added to this container, then this method will throw an error (because in turn `findPartition` will throw an error).

Retrieves the disjoint set to which `elem1` belongs. If the argument is invalid or not found, this call will throw an error.

Repeats the same operation for `elem2`

```
function areDisjoint(elem1, elem2)
  p1 ← this.findPartition(elem1)
  p2 ← this.findPartition(elem2)
  return p1 != p2
```

Compares the two sets, and checks if they are the same, and hence if the elements belong to the same partition

Naïve Solution: merge

Takes two elements, merges their partitions, and returns `true` iff the two elements were in two different partitions that now are merged, or `false` if they were already in the same partition

Retrieves the partitions to which `elem1` and `elem2` belong. If the argument is invalid or not found, these calls will throw.

Compares `p1` and `p2`, and if they are the same, there is nothing left to do. The elements are already in the same partition, and so no merge happens: `false` is returned.

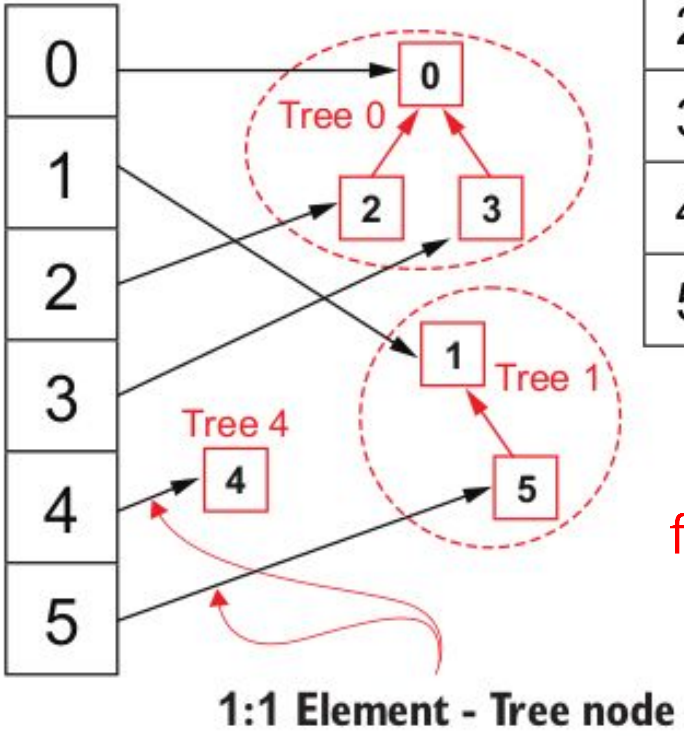
Loops over the elements in the first partition. For each element in `p1` do:

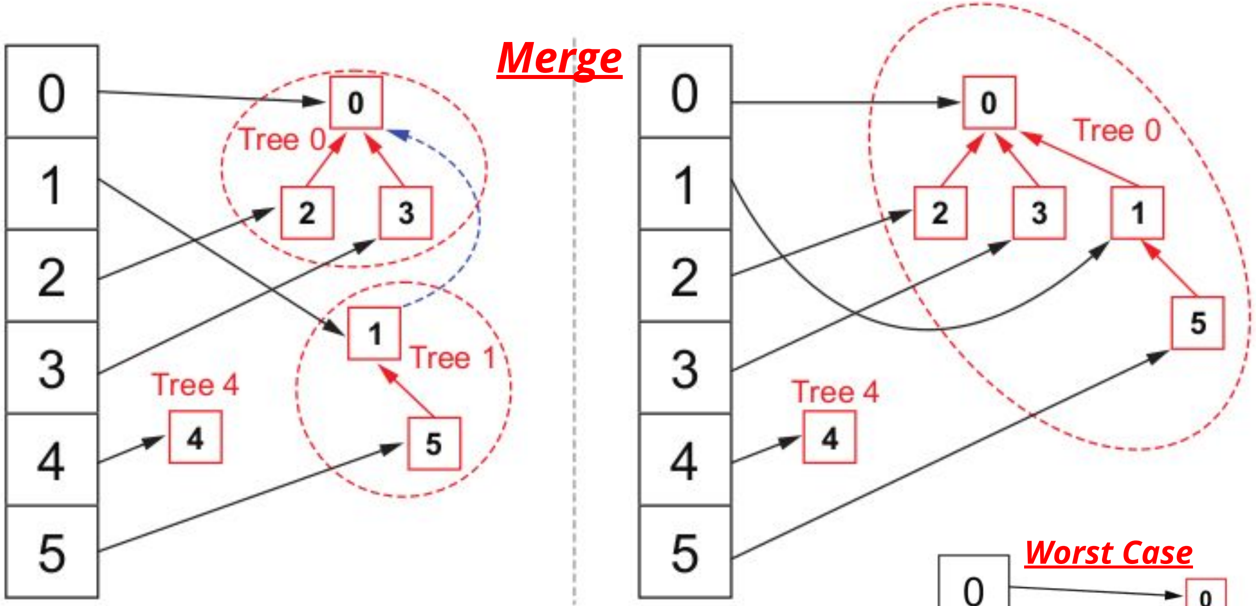
. . . add the element to `p2` . . .

. . . then update the mapping for that element, which now belongs to `p2`.

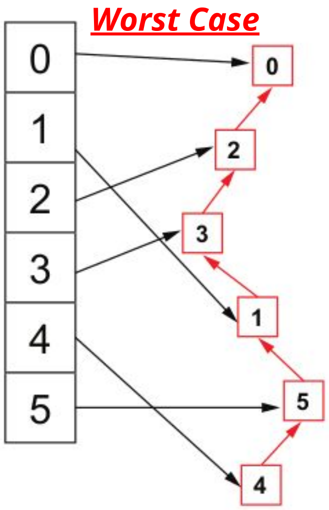
```
function merge(elem1, elem2)
  p1 ← this.findPartition(elem1)
  p2 ← this.findPartition(elem2)
  if p1 == p2 then
    return false
  for elem in p1 do
    p2.add(elem)
    this.partitions[elem] ← p2
  return true
```

Utilizando árboles como estructura





Para implementar el método findPartition en tiempo constante y tiempo lineal en el peor de los casos de nuestra fusión/merge



Utilizando árboles: findPartition(elem)

```
class DisjointSet
  #type HashMap[Element, Tree[Element]]
  parentsMap

  function findPartition(elem)
    throw-if (elem == null or not parentsMap.has(elem))
    parent ← this.parentsMap[elem]
    if parent != elem then
      parent ← this.findPartition(parent)
    return parent
```

Checks that the element is valid

Takes an element and returns another element, the one element at the root of the tree for the partition to which `elem` belongs

Retrieves the parent of the element

If the current element's parent is `elem` itself, then we've already gotten to the root of the tree; otherwise . . .

. . . we need to climb up recursively to the root, by looking for `parent`'s partition.

At this point, `parent` stores the root of the tree for the partition containing `elem`, so we can return it.

Utilizando árboles: merge

Takes two elements, merges their partitions, and returns `true` iff the two elements were in two different partitions that now are merged, `false` if they were already in the same partition

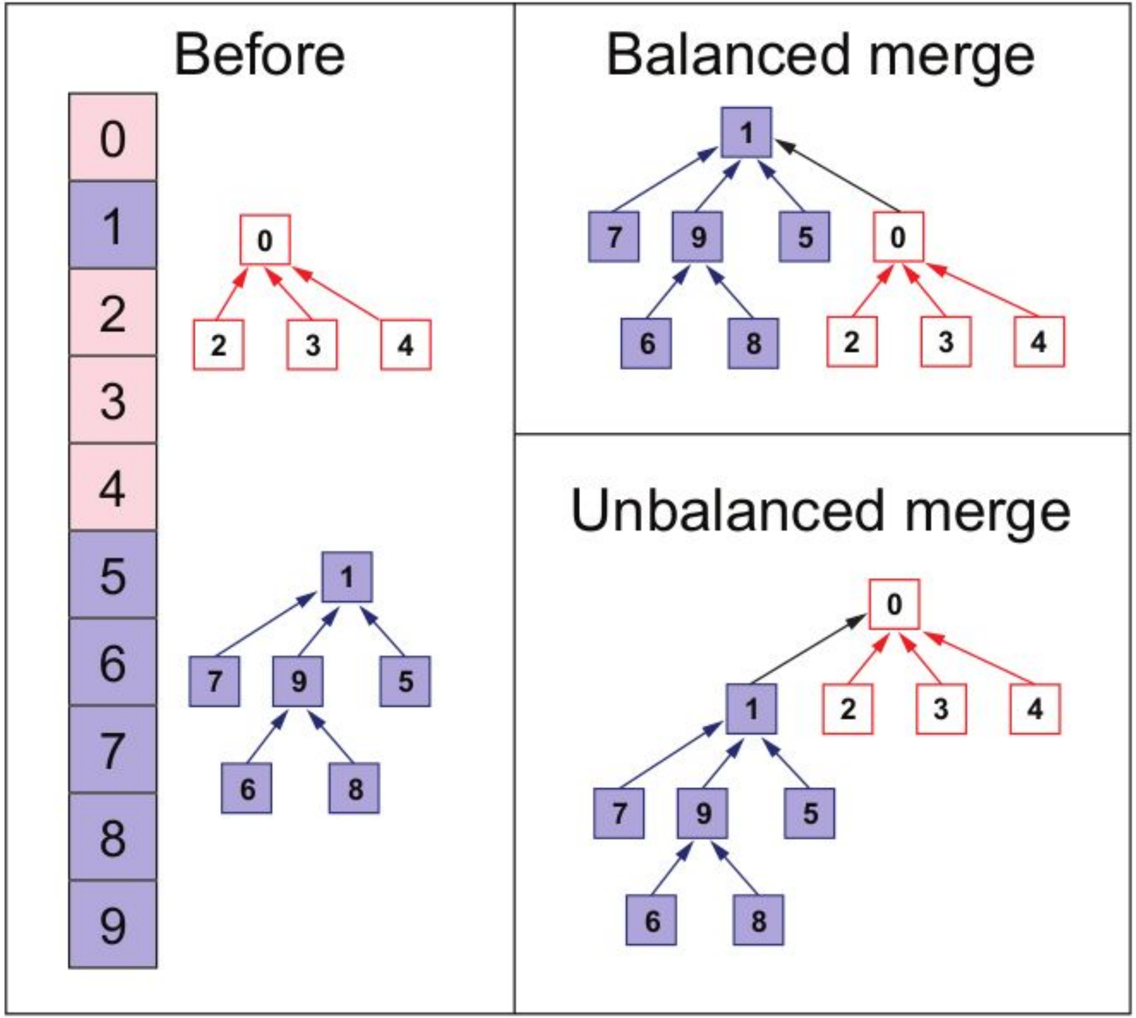
Retrieves the partitions to which `elem1` and `elem2` belong to. If the argument is invalid or not found, this call will throw.

```
function merge(elem1, elem2)
  p1 ← this.findPartition(elem1)
  p2 ← this.findPartition(elem2)
```

Compares `p1` and `p2`, and if they are the same there is nothing left to do. The elements are already in the same partition, and so no merge happens: `false` is returned.

```
  if p1 == p2 then
    return false
  this.parentsMap[p2] ← p1
  return true
```

Sets the parent of `p2` to be equal to `p1`, so that now both `p1` and `p2` have the same parent, but also all elements in `p2` will ultimately share `p1` as the root of their tree



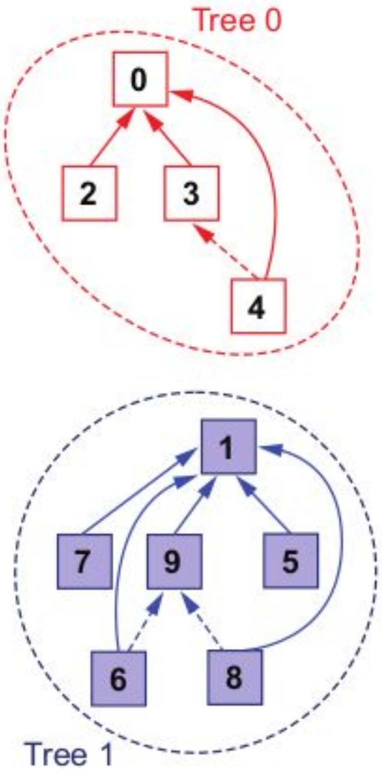
Heurística para mejorar el tiempo de ejecución

El siguiente paso en nuestra búsqueda de un rendimiento óptimo es asegurarnos de que findPartition sea logarítmica incluso en el peor de los casos.

Llevando fácilmente la cuenta del rango (tamaño) de cada árbol, para que cuando hagamos merge de dos árboles, nos aseguraremos de establecer como hijo el árbol con el menor número de nodos.

	Size	Root
0	3	0
1	6	1
2	1	0
3	2	0
4	1	0
5	1	1
6	1	1
7	1	1
8	1	1
9	3	1

Heurística para mejorar el tiempo de ejecución: Comprensión de rutas

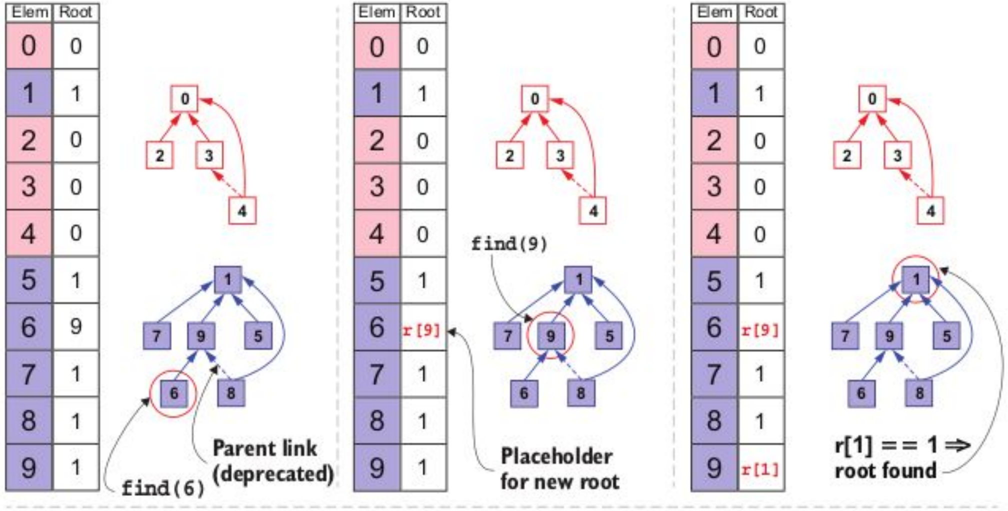


Podemos hacer algo mejor que solo tener árboles equilibrados y métodos de tiempo logarítmico.

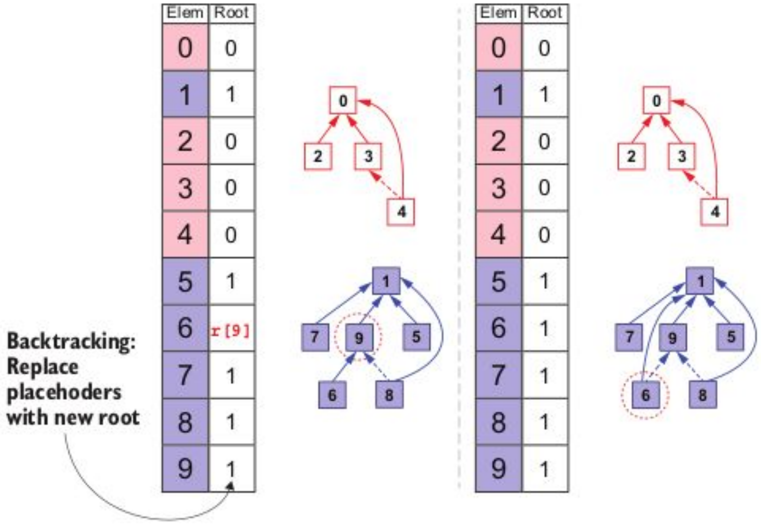
Para cada nodo de los árboles, en lugar de almacenar un enlace a su padre, podemos almacenar uno a la raíz del árbol.

Total,no necesitamos llevar un historial de las fusiones realizadas ya que sólo necesitamos saber en el momento actual cuál es la raíz de la partición de un elemento

Heurística para mejorar el tiempo de ejecución: <u>Comprensión de rutas</u>



¿Qué ocurre si no actualizamos inmediatamente los punteros padre en los nodos del árbol establecidos como hijo? La próxima vez que ejecutemos findPartition en uno de los elementos de ese árbol digamos x, tendremos que recorrer el árbol desde x hasta su primera(antigua) raíz xR y luego desde xR hasta la nueva raíz R del arbol(nuevo) completo



Tenemos que tener en cuenta que los punteros de los elementos del árbol antiguo podrían haber estado sincronizados antes de la fusión o podrían no haberse actualizado nunca. Sin embargo, como de todos modos tendremos que subir por el árbol, podremos volver sobre nuestros pasos desde la raíz, R hasta x y actualizar los punteros de la raíz para todos esos elementos.

Sin embargo, como consecuencia de realizar estos pasos extra, tendremos que para la próxima vez que ejecutemos findPartition solo necesitaremos un único paso para encontrar su raíz! Es decir, en tiempo constante podremos acceder a nuestra raíz.

¿Y el análisis?

¿Cuántas veces tendremos que actualizar los punteros raíz, en promedio, para una sola operación o, en un análisis amortizado, durante un cierto número k de operaciones?

Sólo hay que saber que se ha demostrado que el tiempo de ejecución amortizado para ***m*** llamadas a findPartition y merge sobre un conjunto de ***n*** elementos requerirá ***O(m * Ack(n))*** accesos al array.

Y ***Ack(n)*** es una aproximación de la función inversa de Ackermann, esta función crece tan lentamente que puede considerarse una constante.

Y aúnque todavía no se sabe si este es el límite más bajo para nuestra estructura, se ha demostrado que ***O(m*InvAck(m, n))*** es un límite inferior estricto, donde ***InvAck(m, n)*** es la verdadera función inversa de Ackermann.

😎 **En pocas palabras, hemos logrado obtener un límite constante amortizado para todas las operaciones en esta estructura de datos.** 🏆

Implementación balanceada con la heurística compresión de la ruta

```
class Info
  function Info(elem)
    throw-if elem == null
    this.root ← elem
    this.rank ← 1

class DisjointSet
  #type HashMap[Element, Info]
  parentsMap

    function findPartition(elem)
      throw-if (elem == null or not parentsMap.has(elem)))
      info ← this.parentsMap[elem]
      if (info.root == elem) then
        return elem
      info.root ← this.findPartition(info.root)
      return info.root
```

The constructor for the Info class just takes an element of the disjoint set.

Validates the argument

The rank of the subtree is initially 1, because it only contains one element.

Takes an element and return another element, the one element at the root of the tree for the partition to which elem belongs

Checks that the element is valid

Retrieves the info node stored for current element

If the element's root is the element itself, then we already got to the root of the tree, and we can return it.

Otherwise, we need to climb up recursively to the root, but meanwhile we can update the root link for the current element so that it points to the actual root of the tree.

Esta clase Info modela la información asociada a un nodo del árbol de particiones. Es sólo un contenedor de dos valores: la raíz del árbol, el rango del árbol enraizado en el elemento actual.

Almacenaremos directamente el índice del propio elemento, que utilizaremos como clave en el HashMap.

Como se describió al principio de la sección, cuando se utiliza la heurística de compresión de rutas, no actualizamos la raíz de todos los elementos en merge , pero sí la actualizamos en findPartition.

Implementación balanceada con la heurística compresión de la ruta: *merge*

Takes two elements, merges their partitions, and returns `true` iff the two elements were in two different partitions that now are merged, `false` if they were already in the same partition

Retrieves the roots of the trees to which `elem1` and `elem2` belong to. If its argument is invalid or not found, these calls will throw.

Compares `r1` and `r2`, and if they are the same there is nothing left to do. The elements are already in the same partition, and so no merge happens: `false` is returned.

```
function merge(elem1, elem2)
  r1 ← this.findPartition(elem1)
  r2 ← this.findPartition(elem2)
  if r1 == r2 then
    return false
  info1 ← this.parentsMap[r1]
  info2 ← this.parentsMap[r2]
  if info1.rank >= info2.rank then
    info2.root ← info1.root;
    info1.rank += info2.rank;
  else
    info1.root ← info2.root;
    info2.rank += info1.rank;
  return true
```

At this point, we know we need to merge the two partitions, so it looks for the info nodes for both roots.

Checks if the first tree has a larger rank (more elements) or vice versa. The smallest tree will become a subtree of the root of the largest tree.

Sets the root of the smallest tree to the other root

Updates the rank of the root (of both trees, now). It's not worth updating the rank of the other (former) root, because it will never be checked again.

Como se describió al principio de la sección, cuando se utiliza la heurística de compresión de rutas, no actualizamos la raíz de todos los elementos en merge , pero sí la actualizamos en findPartition

Aplicaciones

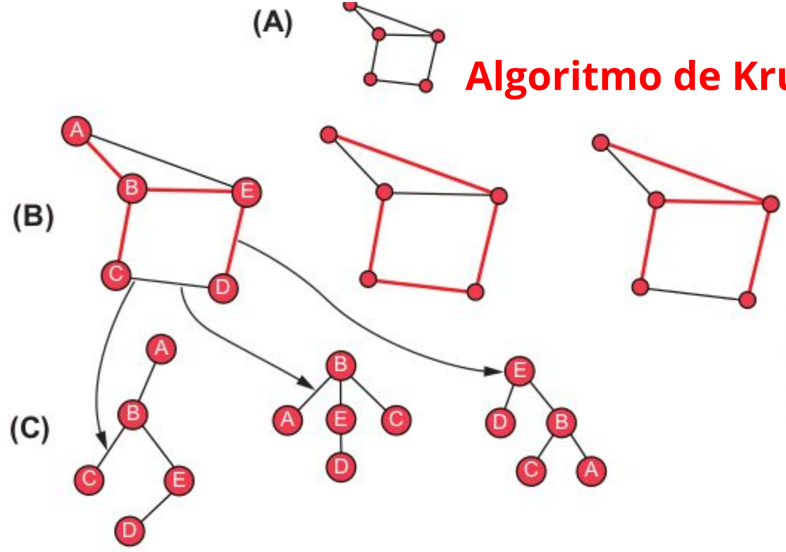
Componentes conectados

Creates a new disjoint set where each vertex of the graph is initially in a different partition

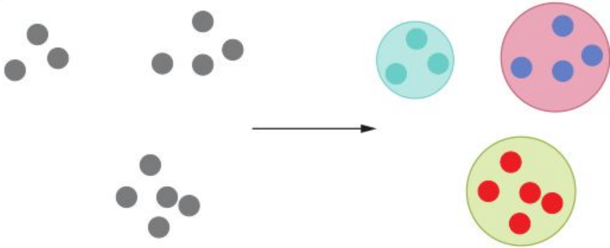
```
disjointSet = new DisjointSet(graph.vertices)
for edge in graph.edges do
  disjointSet.merge(edge.source, edge.destination)
```

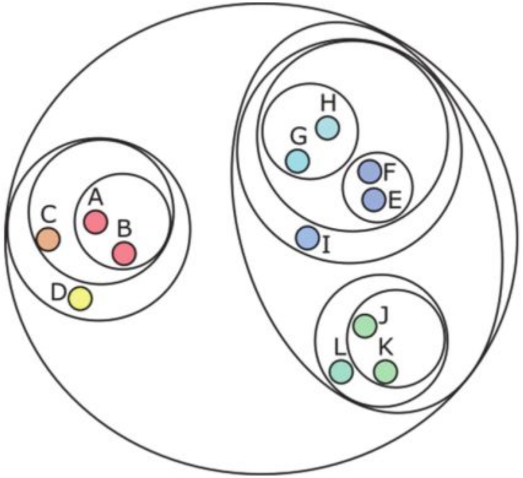
Loops over each edge in the graph

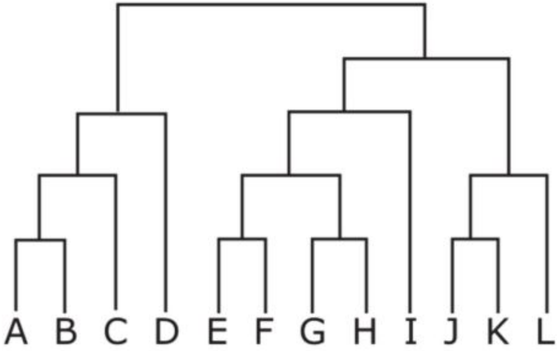
Algoritmo de Kruskal



Clustering







Resumen

1. Podemos construir soluciones cada vez más complejas y eficientes para resolverlo.
2. A veces podemos conformarnos con una implementación subóptima si es lo suficientemente eficiente y el rendimiento no es crítico.
3. Probablemente podríamos conformarnos con la solución ingenua de tiempo lineal, pero esto depende de lo que queramos resolver.
4. Conocemos un límite inferior teórico para el tiempo de ejecución de las operaciones de un conjunto disjunto, pero no sabemos si existe un algoritmo que se ejecute con ese límite, o incluso cualquier otro algoritmo más rápido que los que conocemos.

5. La función inversa de Ackermann, cuyo valor no será mayor que 5 para cualquier número entero que quepa en un ordenador, modela el orden de magnitud del tiempo de ejecución de nuestra operación merge en los conjuntos disjuntos.