Repaso

- Hasta ahora…

Estructuras de Datos y Algoritmos

- Buscar y Ordenar (binSearch – qSort & friends)
- Concepto general <- Diccionarios
- Árboles
  - ABB, AVL, 2-3/2-4, ARN
- Hashing
- Backtracking
- Sort O(n)

Diccionario: estructura de datos con las siguientes operaciones

Asociar un **valor** (p.ej., un archivo con la solución de la tarea 1) a una **clave** (p.ej., un rut o número de alumno)

… o **actualizar** el valor asociado a la clave (p.ej., cambiar el archivo)

Obtener el **valor** asociado a una **clave**

(… y para ciertos casos de uso)

Eliminar del diccionario una **clave** y su **valor** asociado

Un diccionario se puede implementar con diferentes estructuras de datos

El árbol binario de búsqueda (ABB)

Es una estructura de datos que guarda tuplas —pares *(key, value)*— organizadas en nodos de forma recursiva:

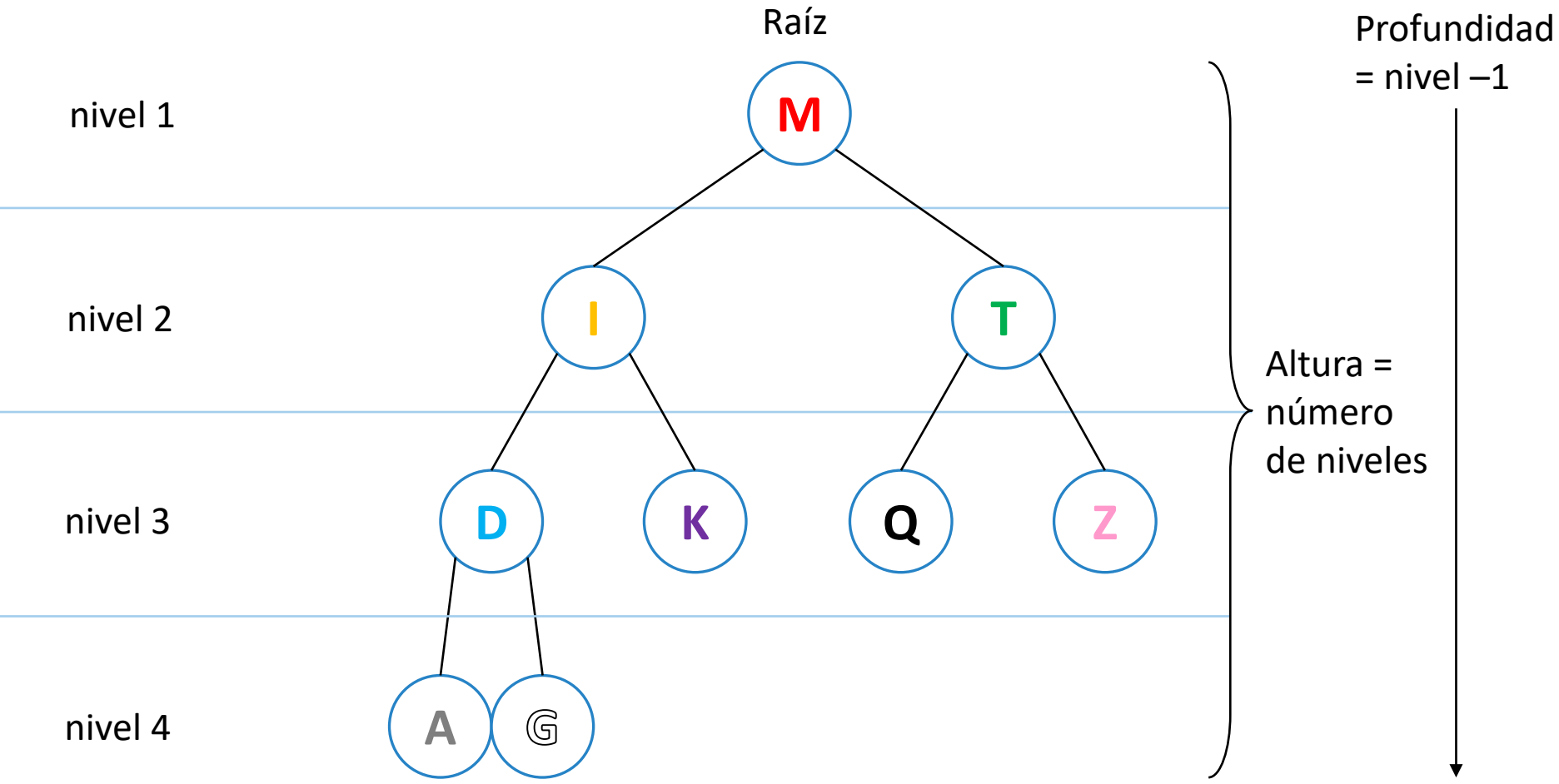
- en las figuras, mostramos sólo las *keys*

La raíz del árbol almacena una tupla y el resto se organiza recursivamente en uno o dos ABBs como hijos (izquierdo y/o derecho) de la raíz:

- la estrategia dividir para reinar aplicada a la estructura de datos

Propiedad ABB: Los *keys* menores que la raíz cuelgan del hijo izquierdo, y los *keys* mayores, del hijo derecho … **recursivamente**

Anatomía de un árbol binario (mostramos solo las *keys*)



Cada nodo A de un ABB va a tener 4 campos:

$A.\boldsymbol{key}$: clave del nodo (p.ej., el rut del estudiante)

$A.\boldsymbol{left}$: puntero al hijo izquierdo

$A.\boldsymbol{right}$: puntero al hijo derecho

$A.\boldsymbol{value}$: valor del nodo (p.ej., un archivo con la ficha académica del estudiante; en general, no lo incluimos en nuestros algoritmos)

```
search(A, k):
    if A = ∅  o  A.key = k:
        return A
    else if k < A.key:
        return search(A.left , k)
    else:
        return search(A.right, k)
```

A es un nodo del árbol; en la llamada inicial, la raíz

k es la clave que buscamos

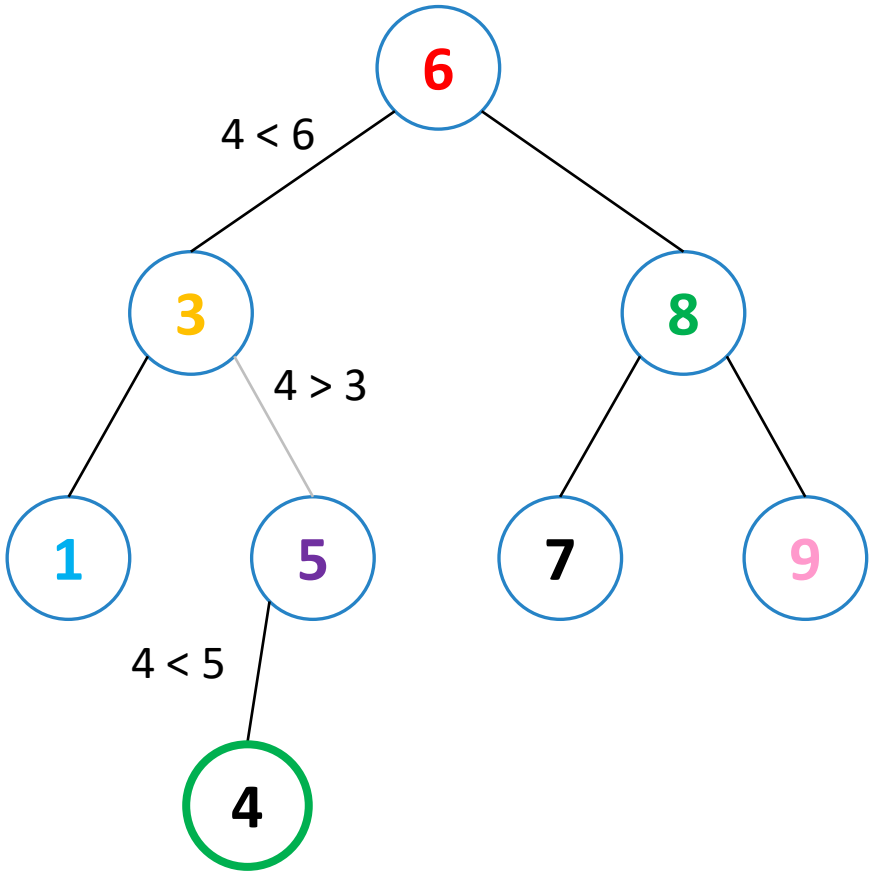
Operaciones que modifican el árbol

Insertar (un nodo con) una nueva *key* (y su *value* asociado) produce un cambio en la estructura del árbol

Eliminar (un nodo con) una *key* (y su *value* asociado) produce un cambio en la estructura del árbol

Ambas operaciones hay que realizarlas de modo de que, una vez terminadas, el árbol sea efectivamente un ABB → si es necesario, hay que restaurar la propiedad de ABB

Insert en ABB



$\boldsymbol{insert(A, k)}$:

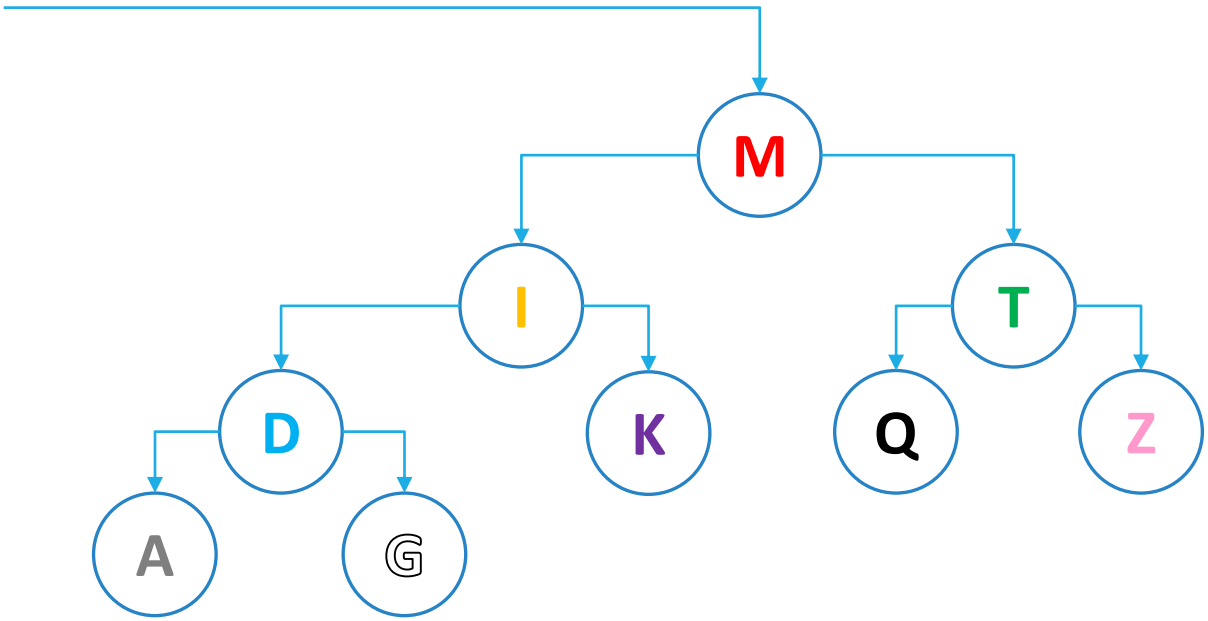
$\boldsymbol{B \leftarrow search(A, k)}$

—crear un nodo B

—conectar B al árbol

$\boldsymbol{B.key \leftarrow k}$

Este procedimiento de inserción nos asegura que el árbol resultante es efectivamente un ABB → no es necesario restaurar la propiedad de ABB




```
min(A):
    if A.left = ∅:
        return A
    else:
        return min(A.left)
```

```
max(A):
    if A.right = ∅:
        return A
    else:
        return max(A.right)
```

```
delete(A, k):
    D = search(A, k)
    if D es hoja:
        D = ∅
    else if D tiene un solo hijo H:
        D = H
    else:                                  —D tiene dos hijos
        R = min(D.right)                   —sucesor de D
        t = R.right                        —posiblemente ≠ ∅
        D.key = R.key
        D.value = R.value
        R = t
```

Antecesor y sucesor, en general

Si los nodos estuvieran ordenados en una lista según su key:

- El **sucesor** de un nodo es el siguiente en la lista
- El **antecesor** de un nodo es el anterior en la lista

¿Cómo podemos encontrar estos elementos dentro del árbol?

Complejidad de las operaciones

Todas las operaciones

buscar, *insertar* y *eliminar**

toman tiempo (o número de pasos) **proporcional a la *altura del árbol*** —el número máximo de niveles desde la raíz hasta la hoja "de más abajo"

… o la longitud de la rama más larga del árbol:

- la altura mínima de un ABB con *n* objetos es O(log*n*)
- … aunque en general podría ser O(*n*)

**insertar* y *eliminar* incluyen primero una búsqueda

ABBs balanceados

Para ABBs, podemos garantizar que las operaciones de diccionario —*buscar*, *insertar* y *eliminar*— tomen tiempo $O(\log n)$ en el peor caso:

es necesario mantenerlos balanceados

La **propiedad de balance** debe cumplir dos condiciones:

- asegurar que la altura de un árbol con *n* nodos sea $O(\log n)$
- ser fácil de mantener —p.ej., la complejidad de (re)balancear el árbol después de una inserción no puede ser mayor que $O(\log n)$

ABB **AVL**

Diremos que un ABB está **AVL-balanceado** si:

- las alturas de los hijos de la raíz difieren a lo más en 1 entre ellas
- cada hijo a su vez está **AVL-balanceado**

- **Después de insertar o eliminar**
  - Nos interesa conservar todas las propiedades de los ABB y además la propiedad de balance AVL:
    - en particular, el balance **debe ser restaurado** antes de que la operación —de inserción o eliminación— pueda considerarse completa

Agregamos a cada nodo *r* un **atributo de balance** (un campo adicional):

$$r.balance = -1 \,/\, 0 \,/\, +1$$

… dependiendo de si:

- el subárbol izquierdo es más alto (–1)
- ambos subárboles tienen la misma altura (0)
- el subárbol derecho es más alto (+1)

Luego de insertar, recorremos el árbol hacia arriba a lo largo de la **ruta de inserción**:

definimos ***X*** como la raíz del **primer** árbol desbalanceado que encontremos (o como el primer “nodo desbalanceado”),

… y ***Y*** como el hijo de ***X*** en la ruta de inserción

Hay 4 casos de desbalance, según la ruta de inserción desde *X*

1. Izquierda + izquierda (LL): rotación simple
2. Izquierda + derecha (LR): rotación doble
3. Derecha + izquierda (RL): rotación doble
4. Derecha + derecha (RR): rotación simple

Los casos 1 y 4 son simétricos entre ellos; lo mismo para 2 y 3

Una rotación tiene **costo constante**: **no depende** del número de nodos del árbol

Rotación simple: hay que cambiar 3 punteros

Rotación doble: hay que cambiar 5 punteros

Además, al hacer estas rotaciones para el ***X*** que definimos, se soluciona el desbalance de ***X*** **sin crear un nuevo desbalance**

… por lo que siempre **en el peor caso realizamos una sola rotación** (simple o doble) por cada inserción

El costo de una inserción sigue siendo $\mathbf{O}(h)$

Luego de insertar, debemos revisar hacia arriba la ruta de inserción buscando el primer desbalance

El peor caso es que no haya un desbalance y lleguemos hasta la raíz

Toda **la inserción sigue siendo $\mathbf{O}(h)$**, siendo h la altura del árbol (aunque el número de pasos individuales más o menos se duplicó)

Entonces, la pregunta relevante ahora es

¿cuál es la altura de un árbol AVL en función del número de nodos?

Altura de un AVL

La altura h de un árbol AVL de n nodos es $O(\log n)$, tanto en el caso del mayor número de nodos, como en el caso del menor número de nodos

Por lo tanto, en un árbol AVL de altura h

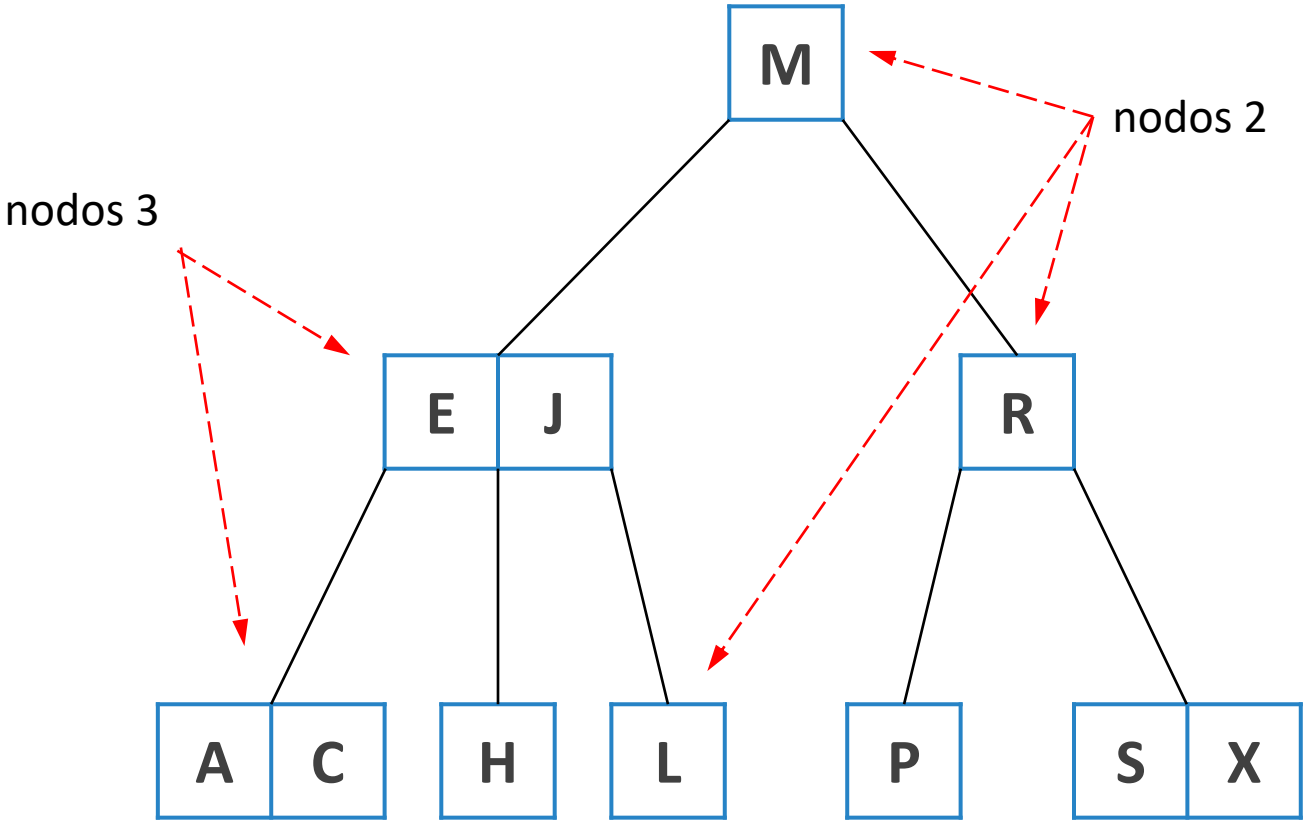
$$h \in O(\log n)$$

Árboles 2-3

Queremos un árbol de búsqueda en que el balance esté dado porque todas las hojas están a la misma profundidad

… y que esa profundidad sea $O(\log n)$, si el árbol almacena n claves

Ejemplo de árbol 2-3



En un **árbol 2-3** hay dos tipos de nodos

Nodo 2, con una clave y, si no es una hoja, exactamente 2 hijos

Nodo 3, con dos claves distintas y ordenadas y, si no es una hoja, exactamente 3 hijos

Esto permite que todas las hojas estén a la misma profundidad

… y que esa profundidad sea $\mathrm{O}(\log n)$, si el árbol almacena n claves:

- en un árbol 2-3, número de nodos ≤ número de claves almacenadas

Inserción en árboles 2-3

La inserción siempre se hace —inicialmente— en una hoja

Si un nodo está lleno (ya tiene dos claves) y debe recibir una tercera clave,

… entonces se hace subir la clave que habría quedado al medio —la clave mediana— al nodo padre

¡ El árbol sólo aumenta de altura cuando la raíz está llena y recibe una clave desde un hijo !

Altura de un árbol 2-3

El mejor caso es que todos los nodos sean nodos 3:

$$h = \log_3 n$$

El peor caso es que todos los nodos sean nodos 2:

$$h = \log_2 n$$

Por lo tanto,

$$h \in \Theta(\log n)$$

El costo de buscar o insertar es $O(2h) = O(h)$

Los árboles 2-3 son balanceados ... pero

Las operaciones en un árbol 2-3, particularmente al insertar una nueva clave, tienen mucho *overhead*:

- durante el recorrido desde la raíz a la hoja, es posible que haya que hacer dos comparaciones en cada nodo (nodos 3)
- cuando se llega a la hoja, si es un nodo 2, hay que convertirlo en un nodo 3
- si es un nodo 3, hay que convertirlo en dos nodos 2 y hacer subir la clave mediana al nodo padre
- si el nodo padre es un nodo 2, hay que convertirlo en un nodo 3; si es un nodo 3, hay que aplicar recursivamente el paso anterior

¿Será posible representar un árbol 2-3 como un ABB?

Nos interesa conservar toda la información del 2-3

Árboles rojo-negro

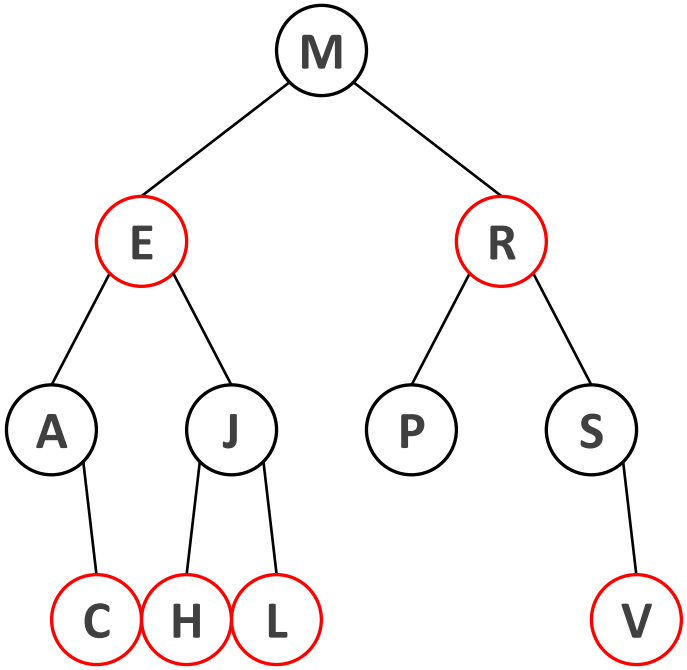
Un **árbol rojo-negro** es un ABB que cumple cuatro propiedades:

1) Cada nodo es ya sea **rojo** o **negro**

2) La raíz del árbol es **negra**

3) Si un nodo es **rojo**, sus hijos deben ser **negros**

4) La cantidad de nodos **negros** camino a cada hoja debe ser la misma



Las hojas nulas se consideran como nodos **negros**

Inserción en un árbol rojo-negro

Una inserción puede violar las propiedades del árbol rojo-negro (así como ocurre en un árbol AVL)

Debemos restaurarlas, usando rotaciones (como en un AVL) y **cambios de color** (en lugar de ajustar el balance del nodo)

Es más fácil de ver si nos fijamos en el **árbol 2-3** equivalente

Equivalencia de árboles rojo-negro con los árboles 2-3 2-4

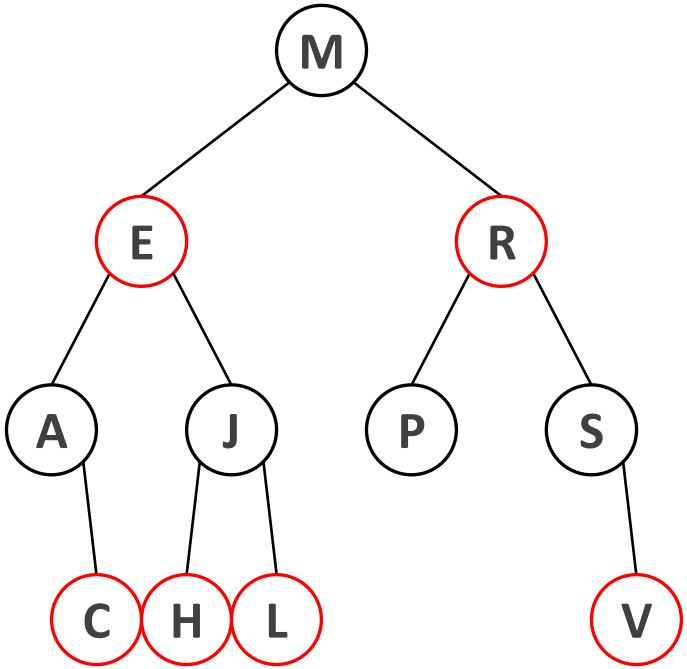
Bueno … no todos los árboles rojo-negro tienen un árbol 2-3 equivalente

…¡pero sí tienen un árbol 2-4 equivalente!

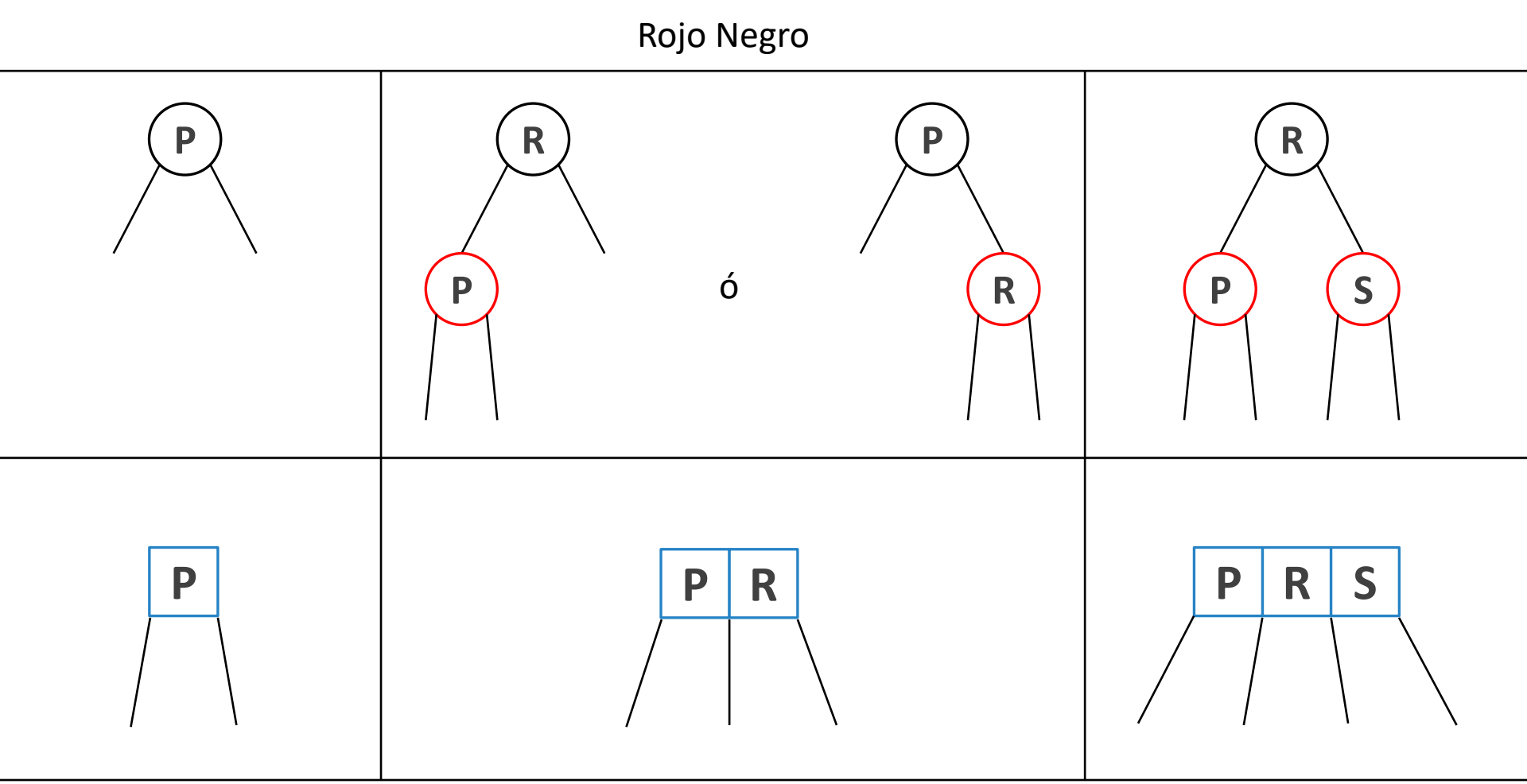
un **árbol 2-4** puede tener nodos 2 y nodos 3 (al igual que un árbol 2-3)

… y además puede tener **nodos 4**:

- 3 claves
- si no es una hoja, entonces 4 hijos



Equivalencia de los árboles rojo-negro con los **árboles 2-4**



¡Entonces hay que fijarse en el **árbol 2-4** equivalente!

Inserción en árboles rojo-negros

Los nodos siempre se insertan rojos

Si su padre es rojo, hay dos casos según el color del tío:

- Si el tío es negro, tenemos el aumento de grado en el nodo del 2-4
  - Se soluciona con rotaciones y cambios de color. No genera más conflictos.
- Si el tío es rojo, tenemos el caso en que el nodo del 2-4 rebalsa
  - Se soluciona cambiando colores. Puede generar el mismo caso hacia arriba.

Inserción en árboles rojo-negros

```
Mientras padre actual es rojo
        Si: tío actual es "rojo"                                  // caso K
                padre actual <- negro
                tío actual <- negro
                abuelo actual <- rojo
                actual <- abuelo actual
        Si no,  si actual es hijo "interior"                      //caso U
                        rotacion padre actual, actual
                        actual <- padre actual
                padre actual <- negro                             // caso Z
                abuelo actual <- rojo
                rotacion abuelo actual, actual
Raiz <- negro
```

… esto para un solo "lado" del árbol

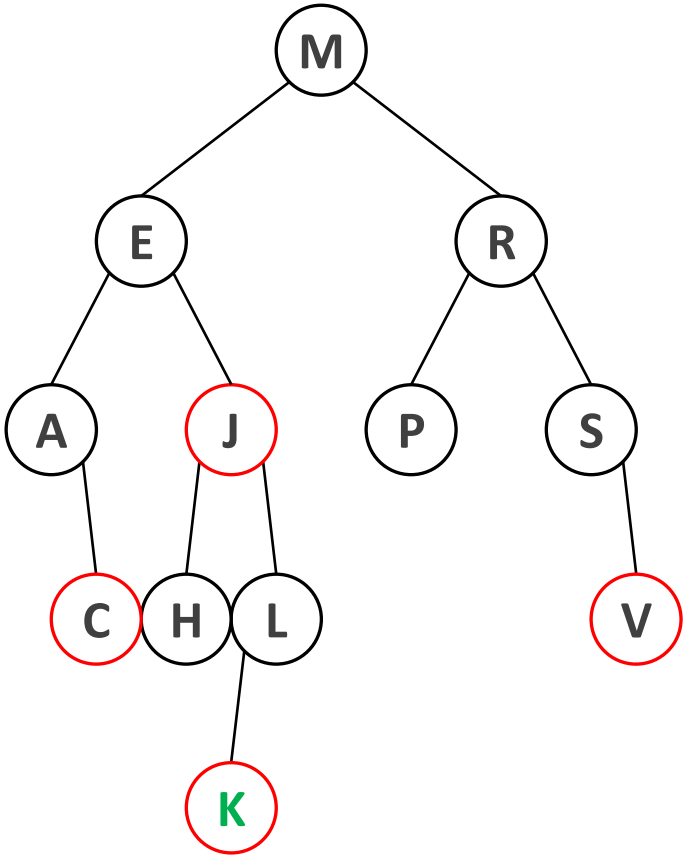
Eliminación en árboles Rojo-Negro

Primero:

- **Eliminar el nodo manteniendo el orden de las llaves**

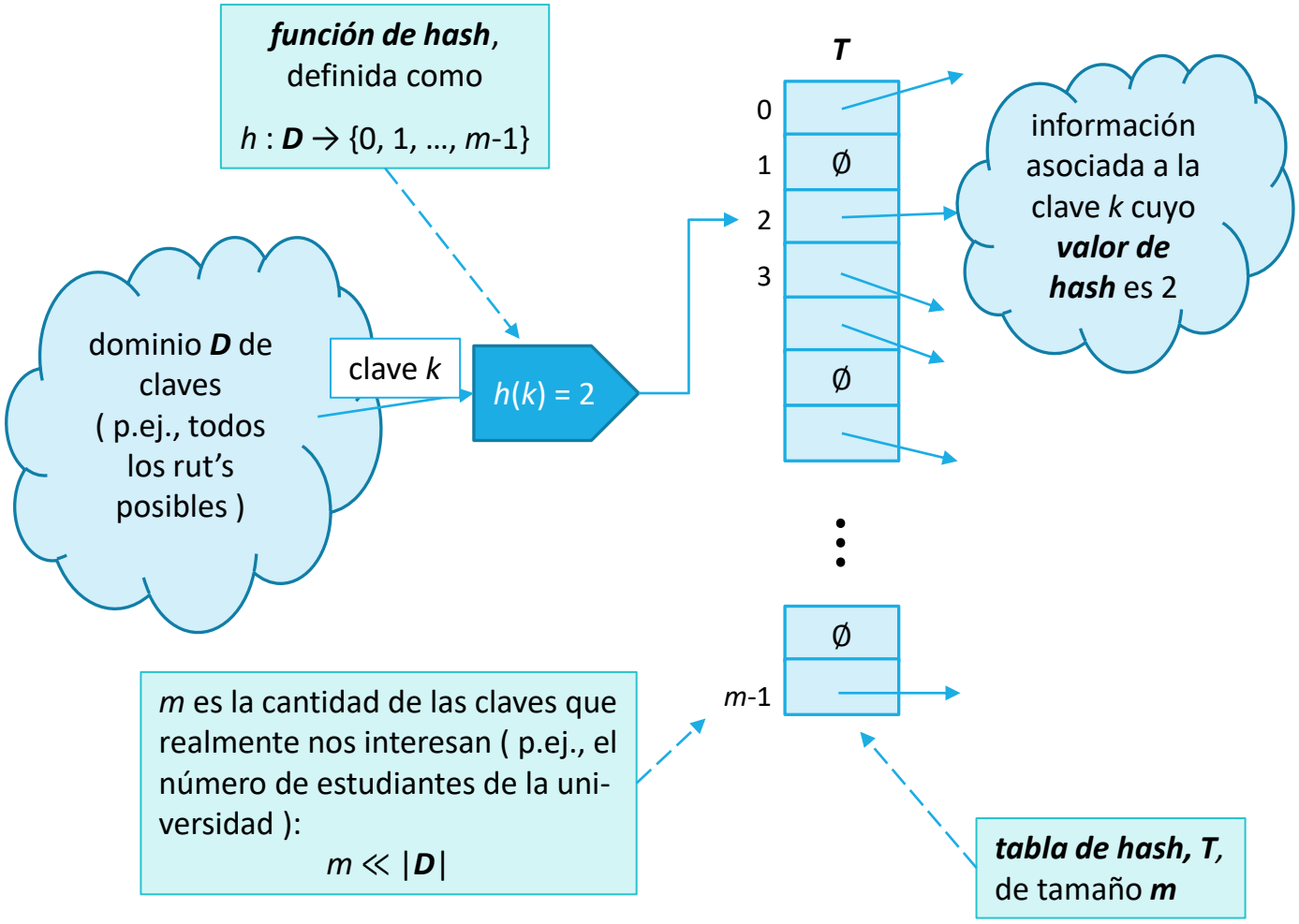
Luego:

- **Recuperar las propiedades rojo-negro**



H*ashing*:

- la clave no se usa directamente como índice
- - el índice **se calcula** a partir de la clave



Propiedades de hashing

Hashing se comporta “casi” como un arreglo:

- en un arreglo, buscar el dato con clave *k* consiste simplemente en mirar *T*[*k*] → es O(1) (diap. 7)
- en hashing, buscar el dato con clave *k* consiste en mirar *T*[*h*(*k*)] → es O(1) pero sólo **en promedio**

En hashing el orden relativo de las claves **no importa**:

- comparar claves entre ellas (para determinar cuál es mayor)

  … o, dada una clave, encontrar la clave predecesora o la sucesora

  … **no son** operaciones de diccionario (diap. 2)

(En este sentido, los ABBs son en realidad diccionarios con operaciones adicionales:

- ABB = diccionario + *cola de prioridades* **)**

Hashing: Colisiones

¿Qué hacemos si una nueva clave debería quedar en una casilla que ya está ocupada?
→**Colisión**: dos posibilidades

- Una posibilidad es usar **encadenamiento**: hacer una lista con las claves que van a una misma casilla

- Otra forma de manejar colisiones es **direccionamiento abierto:** buscar sistemática-mente una casilla vacía, distintas estrategias de sondeo.

Insertemos la clave 51:

$$h(51) = 51 \bmod 7 \ = 2$$

nil	15	37	nil	nil	nil	nil
0	1	2	3	4	5	6

Hashing: Búsqueda & Eliminación

→**Búsqueda**: dos posibilidades

- **Encadenamiento**: Para buscar (el dato con) la clave *k*, primero calculamos el valor de hash *h*(*k*) y miramos la casilla ***T***[*h*(*k*)] :

si ***T***[*h*(*k*)] está vacía, entonces la clave *k* no está en ***T***

si ***T***[*h*(*k*)] no está vacía, entonces apunta a una lista ligada de una o más claves, todas distintas entre ellas, pero que tienen el mismo valor de hash que la clave *k*
… por lo tanto, buscamos *k* en esta lista, p.ej., secuencialmente (*k* puede estar o no en la lista)

Hashing: Búsqueda & Eliminación

→**Búsqueda**: dos posibilidades

- **Direccionamiento abierto**: Para buscar (el dato con) la clave *k*, primero calculamos el valor de hash *h*(*k*) y miramos la casilla ***T***[*h*(*k*)] :

si ***T***[*h*(*k*)] está vacía, entonces la clave *k* no está en ***T***

si ***T***[*h*(*k*)] no está vacía, entonces verificamos el valor de la clave, si no corresponde avanzamos a la siguiente posición de sondeo hasta encontrar la clave o una posición vacía (no está la clave en ***T***)

Hashing: Búsqueda & Eliminación

→**Eliminación**: dos posibilidades

- E**ncadenamiento**: Para eliminar la clave *k* —suponiendo que la buscamos y la encontramos— simplemente la sacamos de la lista ligada

- D**ireccionamiento abierto:** no es simple manejar las eliminaciones…

Direccionamiento abierto: otras políticas de sondeo

Sondeo lineal (el que vimos recién): si $h(k) = H$, entonces

- buscamos en H, $H + 1$, $H + 2$, $H + 3$, …
- también puede ser H, $H + d$, $H + 2d$, $H + 3d$, …, en que d no tiene factores comunes con m

Sondeo cuadrático: si $h(k) = H$, entonces

- buscamos en H, $H + 1$, $H + 4$, $H + 9$, …

Doble hashing:

- ocupamos dos funciones de hash, $h_1(k)$ y $h_2(k)$
- buscamos en $h_1(k)$, $h_1(k) + h_2(k)$, $h_1(k) + 2h_2(k)$, $h_1(k) + 3h_2(k)$, …
- p.ej., $h_1(k) = k \bmod m$ y $h_2(k) = 1 + k \bmod (m–2)$, en que m y m-2 son *primos gemelos*

En cualquiera de estos casos, el problema al eliminar claves se manifiesta igual

¿Qué tan llena está la tabla?

Si una tabla de *m* casillas tiene almacenados *n* datos

… entonces definimos el **factor de carga** λ como

$$\lambda = \frac{n}{m}$$

- con encadenamiento, es aceptable $\lambda \approx 1$ (o incluso un número entero pequeño mayor que 1)
- con direccionamiento abierto, $\lambda > 0.5$ resulta en inserciones y búsquedas muy lentas

Rehashing

Cuando la tabla se empieza a llenar demasiado —las búsquedas e inserciones se empiezan a demorar más de lo aceptable—

… hay que construir otra tabla —aproximadamente el doble de grande— y definir una nueva función de hash para esta tabla

… y pasar todos los datos de la tabla original a la nueva tabla —calculando el nuevo valor de hash para cada uno

Esta es una operación cara —O(n)— pero infrecuente:

- tienen que haber habido O(n) inserciones en la tabla original

  … por lo que en esencia estamos agregando un costo constante a cada inserción (por eso la nueva tabla es el doble de grande)

Funciones de hash

Claves numéricas:

- la función de hash debe convertir la clave a un número entero entre 0 y $m-1$
- las claves pueden ser números enteros o bien números reales

Claves no numéricas:

- la función de hash debe, primero, convertir la clave a un número (entero)
- … y, luego, convertir este número a un número entero entre 0 y $m-1$

Representaciones binarias de las claves

La idea es que la función de hash dependa de todos los bits de la clave

P.ej., para una tabla de tamaño 2^m, John von Newmann (aparentemente) sugirió elevar la clave k — supongamos de n bits en binario— al cuadrado

... y tomar los m bits del medio del patrón de $2n$ bits resultante:

$$h(k) = (n^2/2^r) \bmod 2^m$$

... en que $r = n - m/2$ es el número de bits ignorados en el extremo derecho de n^2

Si $m = 16$ y $k = 1234567_{10}$
→ $k^2 = 1524155677489_{10}$
= 010110001011011110110000011111101100110001$_2$
... en que **1111011000001111$_2$** = 62991_{10}
→ $h(1234567) = 62991$

Números primos en hashing

… y si, p.ej., $m = 128 (= 2^7)$, entonces el valor de la función va a depender sólo de los 7 bits menos significativos de la clave

Más aún, si *m* es un múltiplo de 2, 3, 5 o cualquiera otra constante pequeña

… entonces la distribución de las claves en la tabla no va a ser muy uniforme (al menos para ciertos conjuntos de claves):

- p.ej., si *m* es par y las claves *k* son números pares, entonces *k* mod *m* es par
- ... y la mitad de a tabla no va a ser usada

***m* debería ser un número primo cercano al tamaño que queremos para la tabla**

Números enteros: Método de multiplicación

Sea A un número entre 0 y 1:

$$h(k) = \lfloor m \cdot (A \cdot k \bmod 1) \rfloor$$

Es decir, multiplicamos *k* por *A* y extraemos la parte fraccional del producto (lo que queda a la derecha del punto decimal)

… éste valor lo multiplicamos por *m* y finalmente tomamos el piso del resultado

En este caso (a diferencia del anterior), el valor de *m* no es crítico

… y, de hecho, una forma de simplificar el cálculo es que *m* sea una potencia de 2

La técnica algorítmica *backtracking*

Dadas variables $x_1, \cdots, x_n$ con dominios finitos $d_1, \cdots, d_n$

… y un conjunto de restricciones R

… encontrar una asignación para cada x que respete R

Modelación de un problema para poder resolverlo mediante backtracking

¿Cuáles son las variables?

¿Cuáles son sus dominios?

¿Cuáles son las restricciones?

```
is solvable(X, D, R):
    if X = ∅, return true
    x ← alguna variable de X
    for v ∈ D_x:
        if x = v viola R, continue
        x ← v
        if is solvable(X − {x}, D, R):
            return true
        x ← ∅
    return false
```

Backtracking en pseudo código.

$all - solutions(X, D, R)$:

$if\ X = \emptyset,\ return\ true$

$x \leftarrow$ alguna variable de X sin asignar

$for\ v \in D_x$:

$if\ x = v$ viola R, $continue$

$x \leftarrow v$

$if\ all - solutions(X, D, R)$:

X es una asignación valida

$x \leftarrow \emptyset$

$return\ false$

Mejoras a Backtracking

- Podas
- Propagación
- Heurísticas

Para cada mejora veremos:

1. Contexto
2. Definición
3. Ubicación en pseudo código

$is\ solvable(X, D, R)$:

$if\ X = \emptyset$, $return\ true$

$x \leftarrow$ la mejor variable de X

$for\ v \in D_x$, de mejor a peor:

$if\ x = v$ no es válida, $continue$

$x \leftarrow v$, propagar

$if\ is\ solvable(X - \{x\}, D, R)$:

$return\ true$

$x \leftarrow \emptyset$, propagar

$return\ false$

Sorts en tiempo O(*n*)

`countingSort`:
No compara los datos que está ordenando

Suponemos que cada uno de los *n* datos es un número entero en el rango 0 a *k*, con k entero

… esta es información con la que no contábamos antes:

si *k* es O(*n*), entonces *countingSort* corre en tiempo $\Theta(n)$

Determinamos, para cada dato *x*, el número de datos menores que *x*:

- esto permite ubicar a *x* directamente en su posición final en el arreglo de salida
- hay que manejar el caso en que varios datos tengan el mismo valor

```
countingSort(data, tmp, k):
    sea count[0..k] un nuevo arreglo
    n = data.length
    for i = 0 … k:
        count[i] = 0
    for j = 1 … n:
        count[data[j]] = count[data[j]]+1
    for p = 1 … k:
        count[p] = count[p]+count[p-1]
    for r = n … 1:
        tmp[count[data[r]]] = data[r]
        count[data[r]] = count[data[r]]-1
```

Este algoritmo es (claramente) $\Theta(k+n)$

Si k es $O(n)$, entonces `countingSort` es $\Theta(n)$

Es estable

RadixSort

```
radixSort(a, d):
    for j = 1 … d:
        usando una ordenación estable,
        ordenar el arreglo a según el dígito j
```

Si **a** contiene *n* números de *d* dígitos,

… en que cada dígito puede tomar hasta *k* valores posibles,

… entonces `radixSort` toma tiempo $\Theta(d(n+k))$ en ordenar los *n* números:

si *d* es constante y $k = O(n)$, entonces *radixSort* es $\boldsymbol{\Theta(n)}$

El algoritmo es útil para ordenar strings, cuando todos son del mismo largo: LSD string sort

MSD string sort : Strings de largos diferentes

Usamos countingSort para ordenar los strings según el primer carácter

… luego, recursivamente, ordenamos los subarreglos correspondientes a cada carácter (excluyendo el primer carácter, que es el mismo para cada string en el subarreglo)

Así como *quicksort*, *MSD string sort* particiona el arreglo en sub-arreglos que pueden ser ordenados independientemente,

… pero lo particiona en **un subarreglo para cada posible valor del primer carácter**, en lugar de las dos particiones de *quicksort*

Precauciones

Fin del string:

- “she” es menor que “shells”

Alfabeto:

- binario (2), minúsculas (26), minúsculas + mayúsculas + dígitos (64), ASCII (128), Unicode (65,536)

Subarreglos pequeños:

- p.ej., tamaño ≤ 10
- cambiar a un *insertionSort* que sepa que los *p* primeros caracteres de los strings que está ordenando son iguales