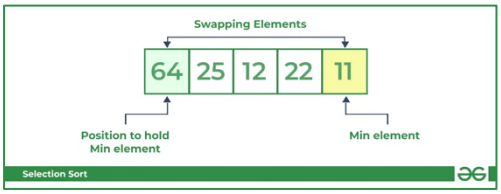
Ignacio Méndez Pérez
2023

Resumen EDD

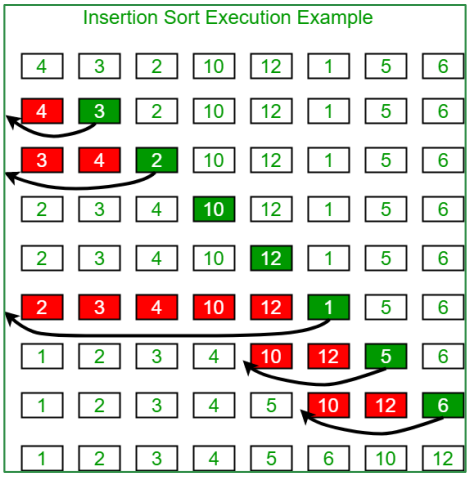
[Visualizador de algoritmos]

- **Selection Sort**: Buscamos el mínimo de una lista L y lo situamos al comienzo de esta.



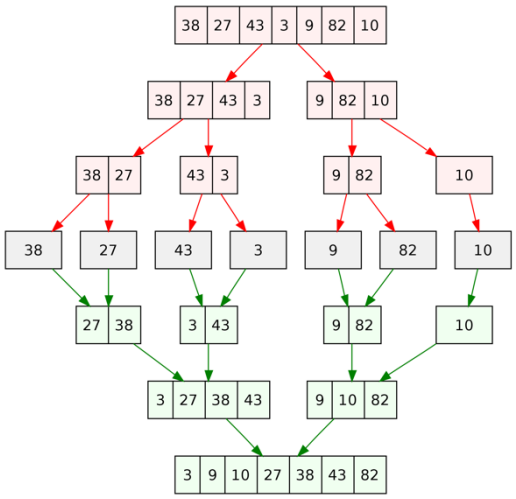
Mejor caso: $\mathcal{O}(n^2)$
Caso promedio: $\mathcal{O}(n^2)$
Peor caso: $\mathcal{O}(n^2)$
Memoria adicional: $\mathcal{O}(1)$

- **Insertion Sort**: Vamos de izquierda a derecha comprobando que los elementos estén ordenados. Si no lo están, cambiamos posiciones hacia la izquierda hasta que lo estén.



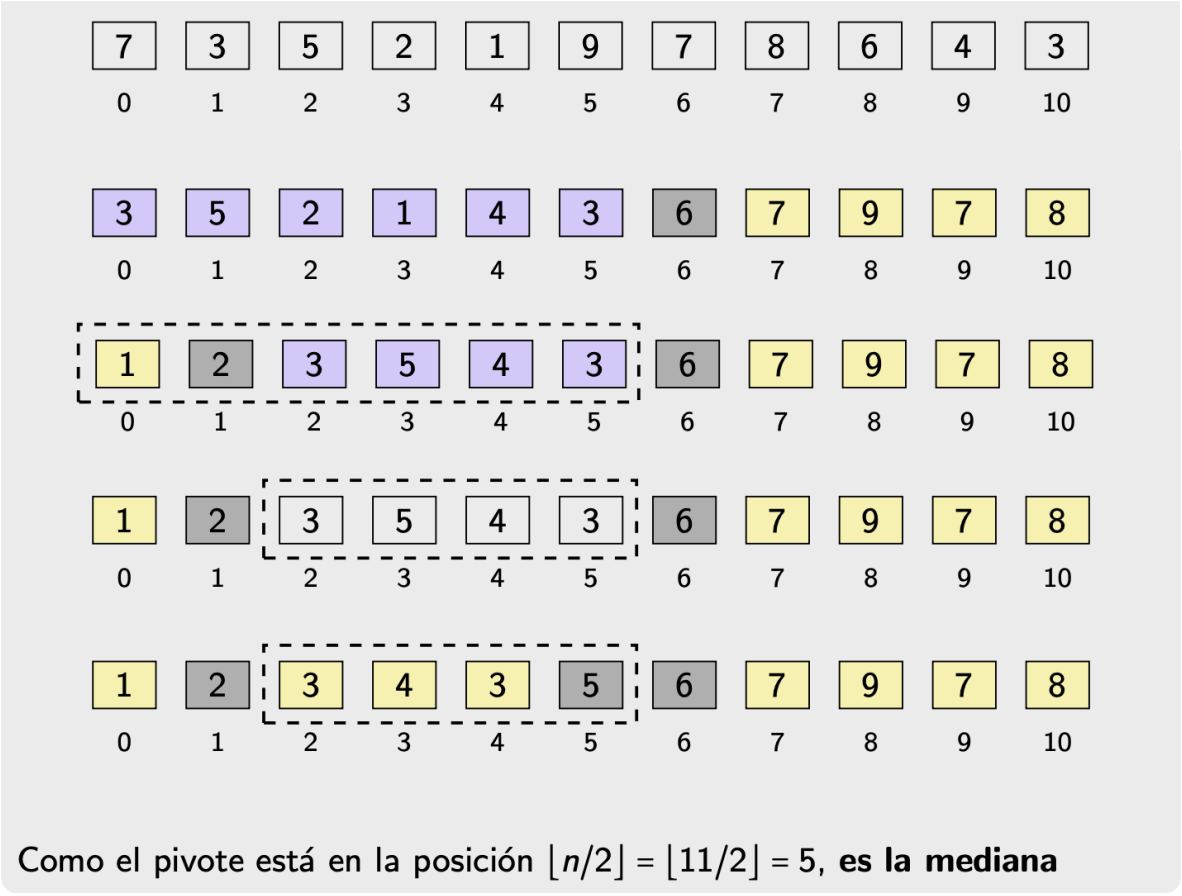
Mejor caso: $\mathcal{O}(n)$
Caso promedio: $\mathcal{O}(n^2)$
Peor caso: $\mathcal{O}(n^2)$
Memoria adicional: $\mathcal{O}(1)$

- **Merge Sort**: Separamos la lista hasta que queden elementos solos. A partir de estas listas más pequeñas, elegimos el menor de cada una de estas listas para ir formando una nueva lista más grande ordenada. Formar una nueva lista toma $\mathcal{O}(n)$ y esto se repite $\log_2(n)$ veces (altura del árbol).



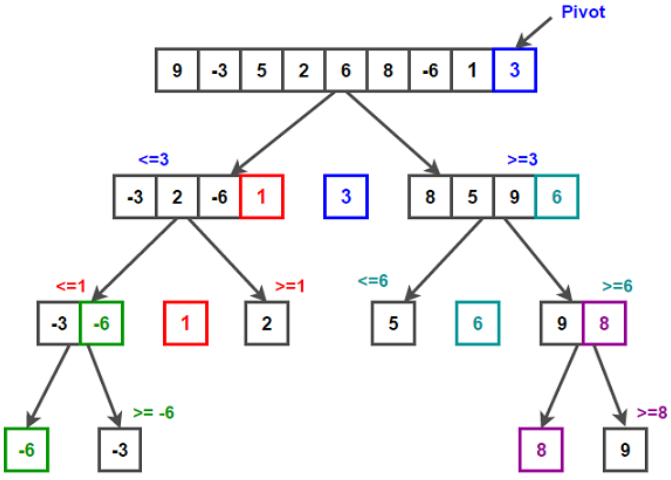
Mejor caso: $\mathcal{O}(n \cdot \log(n))$
Caso promedio: $\mathcal{O}(n \cdot \log(n))$
Peor caso: $\mathcal{O}(n \cdot \log(n))$
Memoria adicional: $\mathcal{O}(n)$

- **Median**: Elegimos un elemento x aleatorio en L. Situamos los elementos mayores a la derecha y menores a la izquierda. Es como la búsqueda binaria.



Como el pivote está en la posición $\lfloor n/2 \rfloor = \lfloor 11/2 \rfloor = 5$, **es la mediana**

- **QuickSort:** Elegimos un pivote $x \in L$ "aleatorio". Muchas veces se usa el último elemento de la lista. Los elementos menores a x los situamos a su izquierda, mientras que los elementos mayores a x los situamos a su derecha. Luego de este paso, el elemento x estará en el índice correcto (esta ordenado) de la lista. Repetimos el paso con ambas listas de la derecha e izquierda.



Mejor caso: $\mathcal{O}(n \cdot \log(n))$
Caso promedio: $\mathcal{O}(n \cdot \log(n))$
Peor caso: $\mathcal{O}(n^2)$
Memoria adicional: $\mathcal{O}(1)$

Ignacio Méndez Pérez
2023

- **Árbol de búsqueda binario (ABB)**: Busca representar un diccionario. Cada nodo x tiene un $x.key$, $x.value$, $x.left$, $x.right$ y $x.p$ (padre). Esta es una estructura recursiva (el subárbol de un nodo hijo es también un ABB).

```
Search (A, k):
1   if A = ∅ ∨ A.key = k :
2       return A
3   if k < A.key :
4       return Search(A.left, k)
5   return Search(A.right, k)
```

Buscar valor de llave, retorna ese sub-árbol

```
Min (A):
1   if A.left = ∅ :
2       return A
3   return Min(A.left)
```

Buscar valor del antecesor del nodo raíz

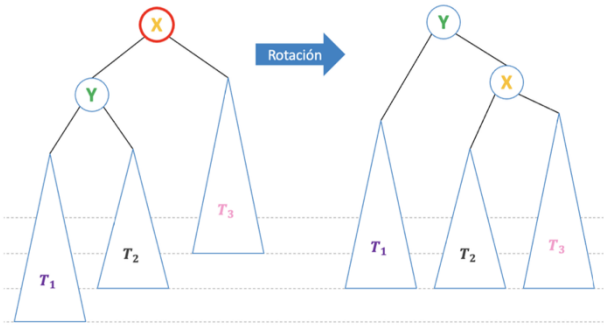
```
Max (A):
1   if A.right = ∅ :
2       return A
3   return Max(A.right)
```

Buscar valor del sucesor del nodo raíz

  - Para insertar un elemento, se va comparando si son menores/mayores a los hijos, hasta llegar a un nodo que no tenga hijo o tenga solo.
  - Si queremos eliminar un nodo raíz, entonces podemos reemplazarlo por su antecesor/sucesor.

- **Árboles AVL**: Es como un ABB, pero más "balanceado":

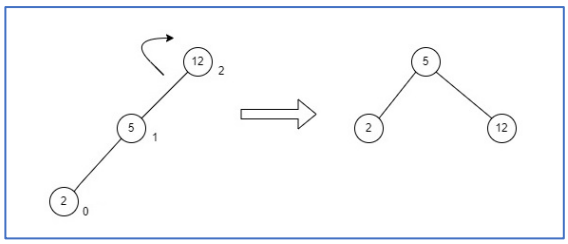
  - Asegura que un árbol con n nodos tenga una altura de $\mathcal{O}(\log n)$.
  - Es fácil de mantener.
  - Existen rotaciones para balancear árbol →

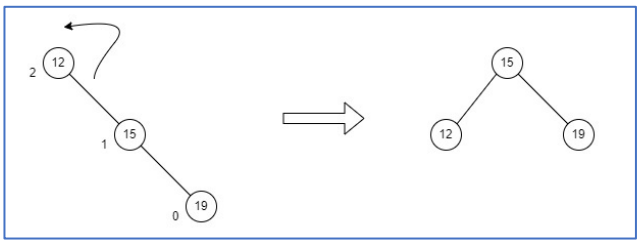


  - Para un nodo x del árbol, agregamos un atributo de balance ($x.balance$) con el cual decidiremos las rotaciones.
  - Tomará valor -1, 0 o 1 según sus hijos.
    - Si el subárbol izquierdo es 1 nivel más alto: $x.balance = -1$.
    - Si los hijos tienen la misma altura: $x.balance = 0$.
    - Si el subárbol derecho es 1 nivel más alto: $x.balance = 1$.

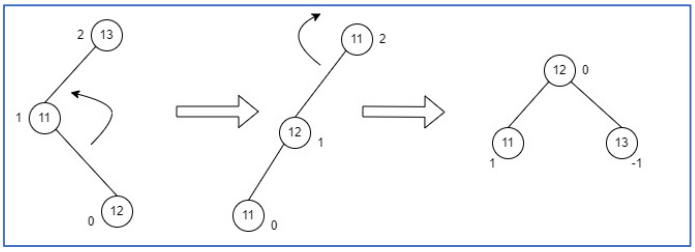
Hay 4 casos posibles de desbalance, según la ruta de inserción:

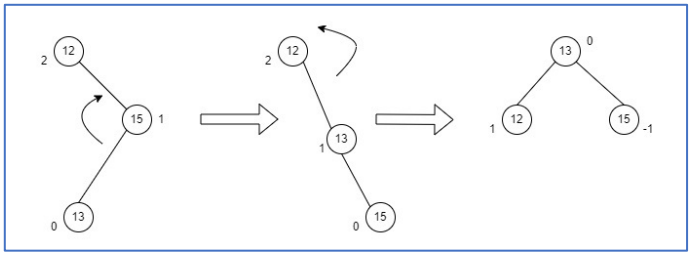
1. Izquierda + izquierda (LL) o Derecha derecha (RR): Rotación simple.



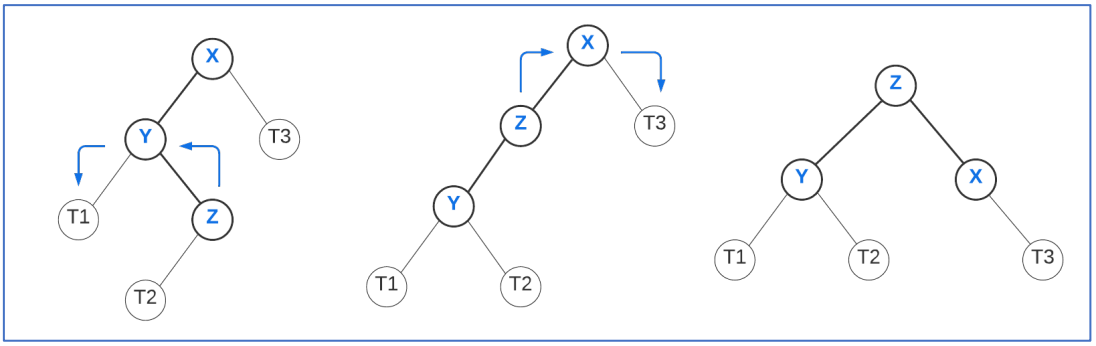


2. Izquierda + derecha (LR) o Derecha izquierda (RL): Rotación doble.





Ejemplo:



- Inserción → $\mathcal{O}(h) = \mathcal{O}(\log n)$.
- Detectar nodos que participan en rotación (si aplica) → $\mathcal{O}(h) = \mathcal{O}(\log n)$.
- Rotación → $\mathcal{O}(1)$.

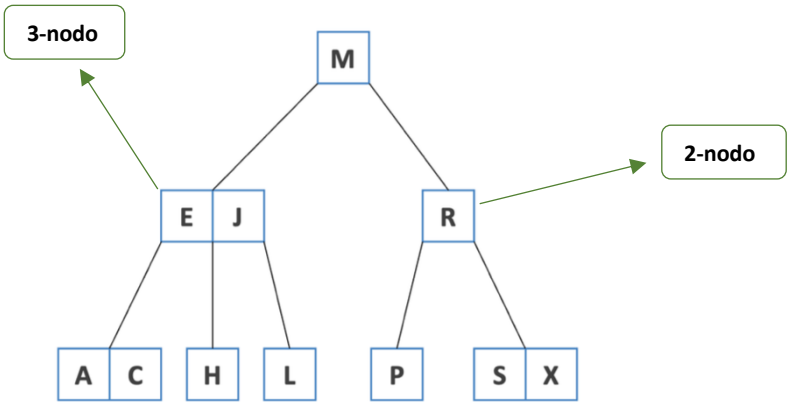
Total ↷ $\mathcal{O}(h) = \mathcal{O}(\log n)$.

- $\max_{A}(nodos) = \sum_{k=0}^{h-1} 2^k = 2^h - 1$.
- $\min_{A}(nodos) = m(h) = m(h-1) + m(h-2) + 1$.

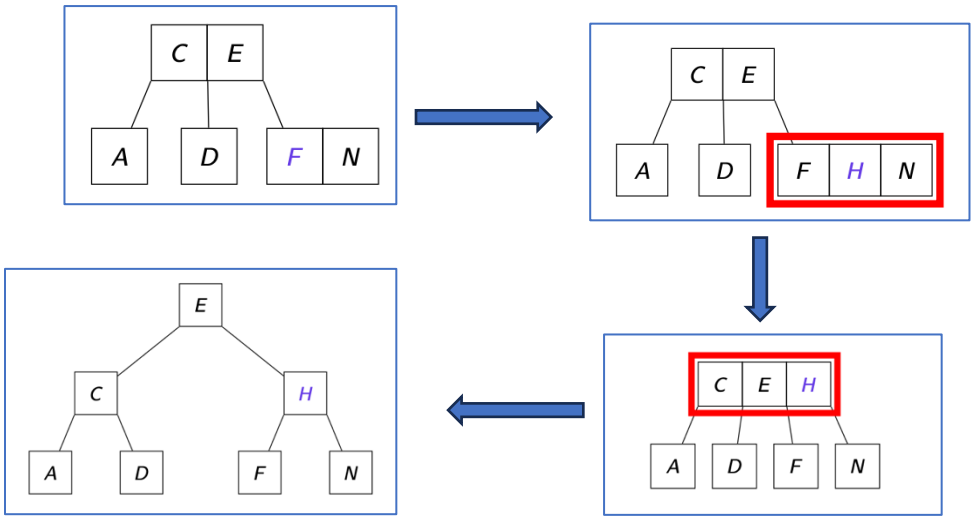
Uno de los hijos debe tener altura $h-1$ y otro $h-2$.

- **Árboles 2-3**:

Los nodos pueden no tener hijos, o tener exactamente:
- **2** hijos árboles 2-3 si es **2-nodo**
- **3** hijos árboles 2-3 si es un **3-nodo**



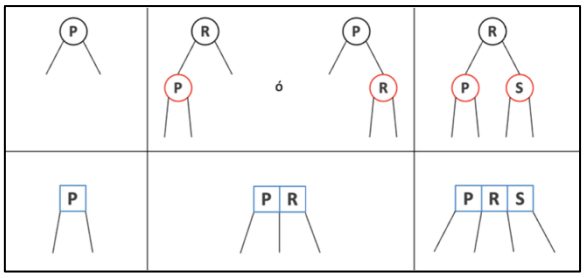
Ejemplo de inserción:



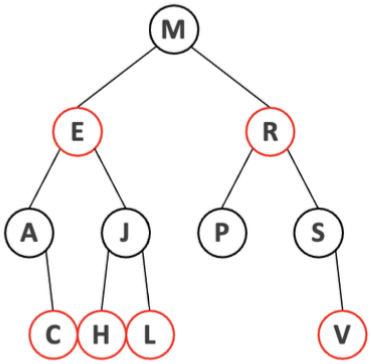
- **Árboles rojo-negro**:

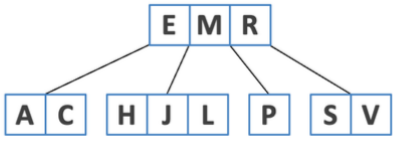
Es un ABB que cumple que:

1. Cada nodo es 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 desde un nodo cualquiera debe ser la misma.

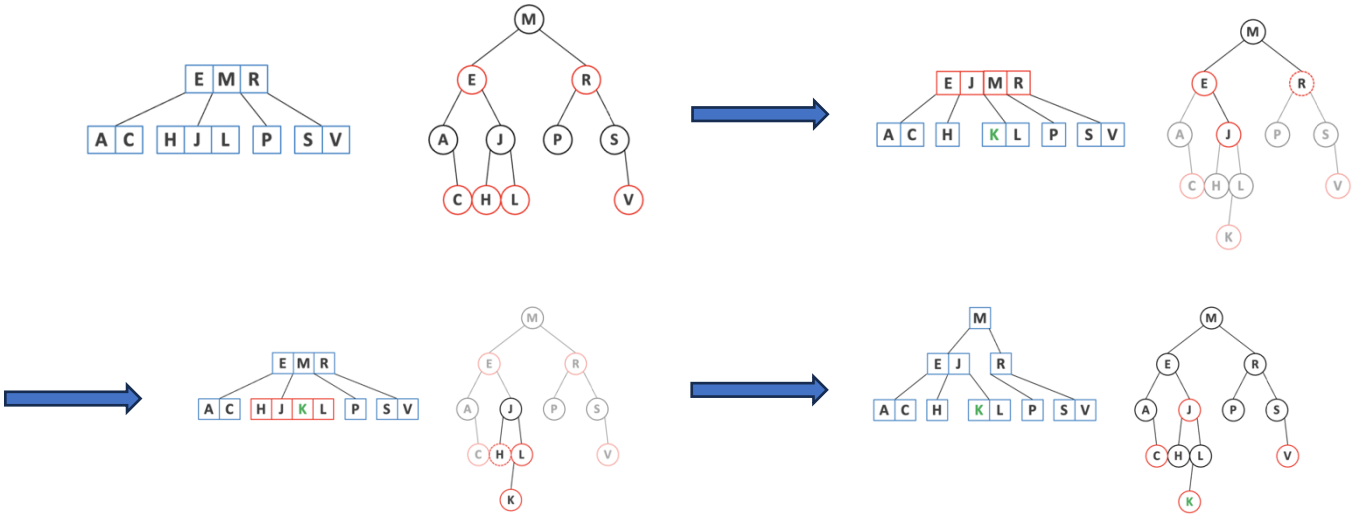


¡Tienen un árbol 2-4 equivalente!





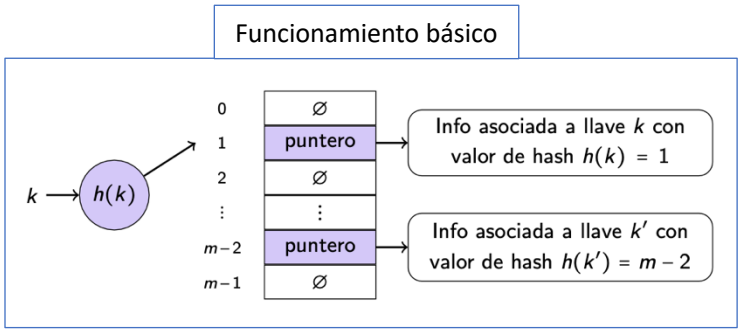
- Para realizar una inserción, simplemente podemos hacer una inserción en el árbol 2-4 equivalente, y a partir de ese construir el nuevo árbol rojo-negro.



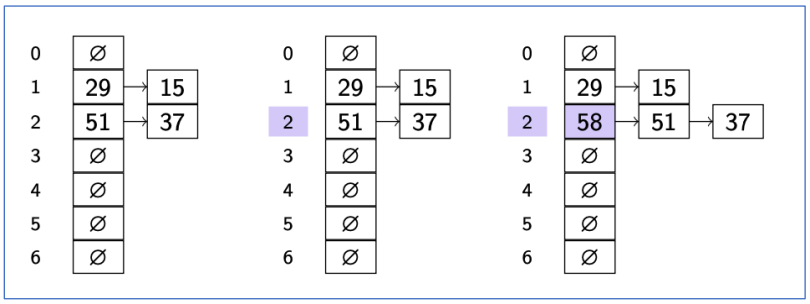
- Para eliminar un nodo, se realizan 2 etapas:
  - Se elimina el nodo tal como se hace en un ABB (buscando antecesor/sucesor).
  - Se recuperan las propiedades rojo-negro (coloración).

POR REVISAR LOS 5 CASOS DE ELIMINACIÓN EN ARBOL ROJO NEGRO (DE SER NECESARIO)

- **Diccionario**



- **Inserción con encadenamiento**: Cada puntero lleva a una lista de elementos.



- **Inserción con sondeo lineal**: Se busca la siguiente celda vacía.
- **Sondeo doble hashing**: Se busca primero en $h_1(k)$, después en $h_1(k) + h_2(k)$, luego en $h_1(k) + 2 \cdot h_2(k)$, etc.

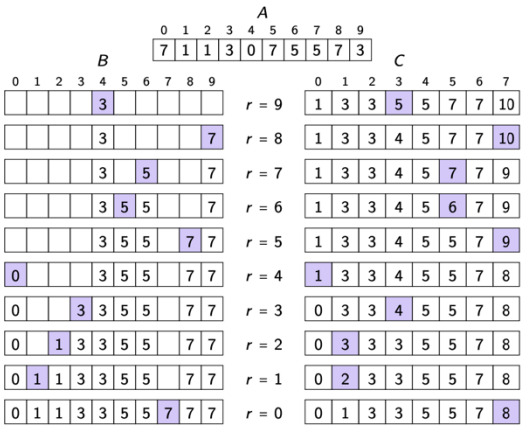
- **Método de la multiplicación**: Tomar $0 < A < 1$:

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

  - Se extra la parte decimal de $A \cdot k$.
  - El valor de m no es crítico (m es el tamaño de la tabla).
  - Tomar m como potencia de 2 simplifica los cálculos, pero es conveniente que sea primo.

- **Ordenación lineal**:

  - Counting Sort:




```
input  : Arreglo A[0...n-1], natural k
output: Arreglo B[0...n-1]

CountingSort (A, k):
1   B[0...n-1] ← arreglo vacío de n celdas
2   C[0...k] ← arreglo vacío de k+1 celdas
3   for i = 0...k :
4       C[i] ← 0
5   for j = 0...n-1 :
6       C[A[j]] ← C[A[j]] + 1
7   for p = 1...k :
8       C[p] ← C[p] + C[p-1]
9   for r = n-1...0 :
10      B[C[A[r]] - 1] ← A[r]
11      C[A[r]] ← C[A[r]] - 1
12  return B
```

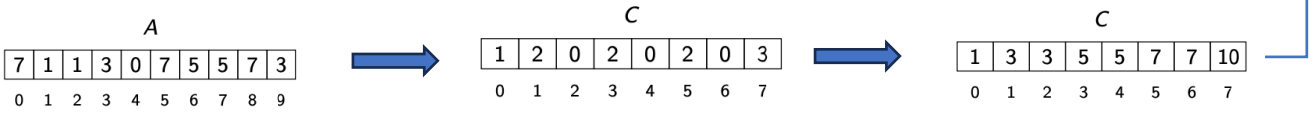
Inicializamos lista de frecuencias (parten en 0)

Creamos la lista de frecuencias C. El elemento $C[i]$ indica cuántas veces existe el elemento i en A.

Pasamos la lista de frecuencias a frecuencia acumuladas

Iteramos la lista A en sentido contrario, y asignamos a la tabla B lo que nos dice la tabla de frecuencia acumulada.

Complejidad $\mathcal{O}(n + k)$. Donde n es el largo de la lista y k el valor máximo que pueden tomar. Si $k \in \mathcal{O}(n)$ entonces Counting Sort es $\mathcal{O}(n)$.



Ignacio Méndez Pérez
2023

- **CSP**: Un problema de satisfacción de restricciones es una tripleta (X, D, C) tal que:
  - $X = \{x_1, \dots, x_n\}$ es un conjunto de **variables**.
  - $D = \{D_1, \dots, D_n\}$ es un conjunto de **dominios** respectivos.
  - $C = \{C_1, \dots, C_m\}$ es un conjunto de **restricciones**.

  Problema de las 8 reinas:
  - $X = \{x_1, \dots, x_8\}$. Donde x_i es la reina en la fila i.
  - $D = \{B, \dots, B\}$ con $B = \{1, \dots, 8\}$.
  - $C =$ No se deben "matar".

- **Backtracking**:

```
input : Conjunto de variables sin asignar X, dominios D,
        restricciones R

isSolvable(X, D, R):
1   if X = ∅ : return true                          → Ya asigné todas las variables, por lo tanto terminé.
2   x ← alguna variable de X
3   for v ∈ D_x :                                   → Iteramos sobre todos las posibles opciones para mi siguiente variable
4       if x = v no rompe R :                       → Si al tomar la opción no se rompen las restricciones.
5           x ← v
6           if isSolvable(X − {x}, D, R) :          → Seguimos probando la siguiente variable
7               return true
8           x ← ∅                                   → Se genera el backtrack en caso de que la llamada retorne false.
9   return false
```

  - Tiene una complejidad de $\mathcal{O}(K^n)$ en donde $K = |D_i|$ y n es la cantidad de variables.

- **Grafos**: Los podemos codificar y almacenar en:

  1. **Listas de adyacencias** en que la celda i de la lista tiene una lista con los vecinos j tales que (i, j) es arista del grafo.
  2. **Matriz de adyacencias** en que la celda $[i][j]$ indica si la arista (i, j) existe en el grafo.

  - Detección de ciclos: Si el nodo Y que vamos a visitar ya fue visitado:
    1. Si estamos en un nodo posterior a Y, **hay ciclo**.
    2. Si no, entonces **esta arista no forma ciclo**.

Este algoritmo decide si hay un ciclo **partiendo desde X**

input : Grafo G, nodo $X \in V(G)$

`cycleAfter`(G, X):

```
1  if X.color = gris : return true
2  if X.color = negro : return false
3  X.color ← gris
4  for Y tal que X → Y :
5      if cycleAfter(G, Y) : return true
6  X.color ← negro
7  return false
```

Este algoritmo decide si hay un ciclo **en el grafo**

input : Grafo G

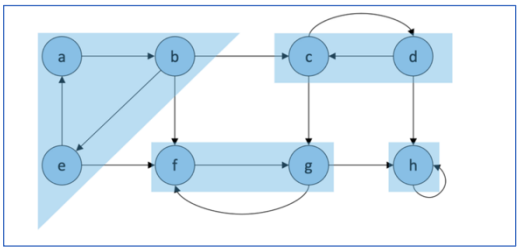
`isCyclic`(G):

```
1  for X ∈ V(G) :
2      if X.color ≠ blanco :
3          continue
4      if CycleAfter(G, X) :
5          return true
6  return false
```

- Este algoritmo sigue una estrategia de **búsqueda por profundidad (DFS)**.

- **Componentes fuertemente conectados**:

  Definición
  Sea G un grafo dirigido. Una componente fuertemente conectada (CFC) es un conjunto maximal de nodos $C \subseteq V(G)$ tal que dados $u, v \in C$, existe un camino dirigido desde u hasta v.



- Un grafo **transpuesto** se obtiene simplemente al invertir todas las aristas.

REPASAR EN PROFUNDIDAD CLASE DE DFS (DE SER NECESARIO)

- **Algoritmos codiciosos**

REPASAR EN PROFUNDIDAD ALGORITMOS GREEDY (DE SER NECESARIO)

- **Programación dinámica**:

Formalicemos las ideas anteriores

- Sea Ω_j la solución al problema con charlas $\{1, \dots, j\}$ y sea $opt(j)$ su ganancia total. **Objetivo final:** obtener Ω_n con valor $opt(n)$
- Para cada $1 \le j \le n$, hay dos casos
  - Si $j \in \Omega_j$, entonces $opt(j) = v_j + opt(b(j))$
  - Si $j \notin \Omega_j$, entonces $opt(j) = opt(j-1)$
- Para saber si $j \in \Omega_j$, comparamos las dos opciones

$$opt(j) = \max\{v_j + opt(b(j)),\ opt(j-1)\} \qquad (\bigstar)$$

input : natural $0 \le j \le n$
output: ganancia óptima

`Opt`(j):

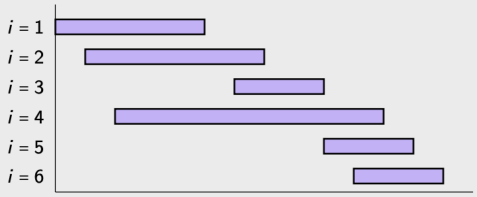
```
1  if j = 0 :
2      return 0
3  else:
4      return max{v_j + Opt(b(j)), Opt(j − 1)}
```

v_j = Ganancia de escoger la conferencia j.

Dadas las charlas $\{1, 2, \dots, 6\}$ ordenadas por f_i, añadimos

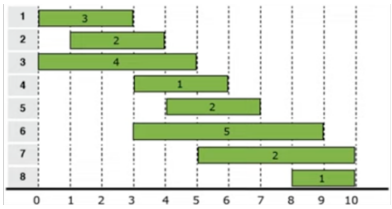
$$b(i) := \begin{cases} j, & j \text{ es la charla que termina más tarde antes de } s_i \\ 0, & \text{no hay tal charla} \end{cases}$$

- $i = 1$, $[0,5)$, $v_1 = 2$, $b(1) = 0$
- $i = 2$, $[1,7)$, $v_2 = 4$, $b(2) = 0$
- $i = 3$, $[6,9)$, $v_3 = 4$, $b(3) = 1$
- $i = 4$, $[2,11)$, $v_4 = 7$, $b(4) = 0$
- $i = 5$, $[9,12)$, $v_5 = 2$, $b(5) = 3$
- $i = 6$, $[10,13)$, $v_6 = 1$, $b(6) = 3$



Ignacio Méndez Pérez
2023

Ejemplo:

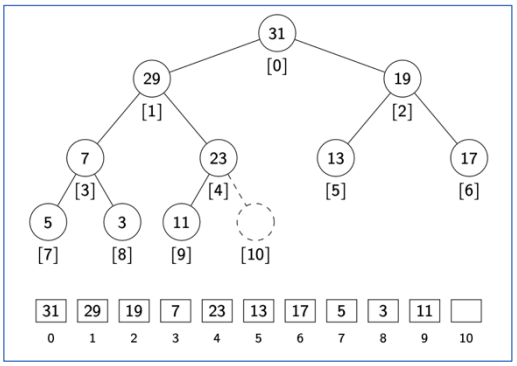


j	1	2	3	4	5	6	7	8
v_j	3	2	4	1	2	5	2	1
$b(j)$	0	0	0	1	2	1	3	5
$OPT(j)$	3	3	4	4	5	8	8	8

- **Heaps**: Un heap binario H es un árbol binario tal que:
  - $H.left$ y $H.right$ son heaps binarios
  - $H.value > H.left.value$.
  - $H.value > H.right.value$.

  Mantener los heaps balanceados permite:

  - Minimizar la altura del árbol representado.
  - Implementar el heap de forma **compacta** en un arreglo.

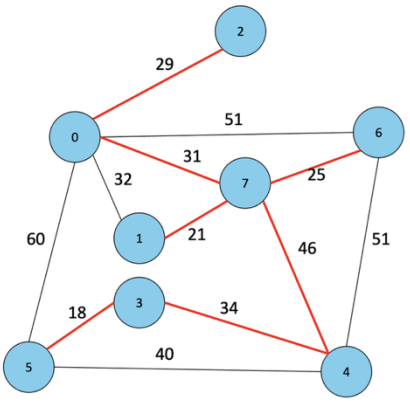


  - Al extraer, sacamos el elemento más prioritario (raíz) y reemplazamos por el menos prioritario (última hoja). Luego intercambiamos con el hijo más prioritario hasta que el heap este nuevamente balanceado.

- **Heapsort**: Tomamos un arreglo, lo pasamos a árbol binario, y lo ordenamos para que quede como un "max" heap. Luego, hacemos extracción del nodo raíz (elemento mayor) y lo ponemos en nuestra lista ordenada. Este paso se repite hasta que no queden nodos.

- **MST**: Dado un grafo no dirigido G, un subgrafo $T \subseteq G$ se dice un **árbol de cobertura mínimo** o **MST** de G si:

  1. T es un árbol.
  2. $V(T) = V(G)$.
  3. No existe otro MST T' para G con menor costo total.



Ignacio Méndez Pérez
2023

- **Algoritmo de Prim**:
  1. Agrego un nodo arbitrario a mis set $\{ Node \}$.
  2. Dentro de todos los posibles nodos no visitados que pueden alcanzar los nodos del set, elijo el que tiene menor costo, y agrego el nodo a mi set.
  3. Repito el paso 2 hasta visitar todos los nodos.

```
Prim(s):
1   Q ← cola de prioridades vacía
2   T ← lista vacía
3   for u ∈ V − {s} :
4       d[u] ← ∞;  π[u] ← ∅;  Insert(Q, u)
5   d[s] ← 0;  π[s] ← ∅;  Insert(Q, s)
6   while Q no está vacía :
7       u ← Extract(Q)
8       T ← T ∪ {(π[u], u)}
9       for v ∈ α[u] :
10          if v ∈ Q :
11              if d[v] > cost(u, v) :
12                  d[v] ← cost(u, v)
13                  π[v] ← u
14  return T
```

- **Algoritmo de Kruskal**:
  1. Elijo la arista con menor costo y la agrego a mi set $\{ edge \}$.
  2. Si no forma un ciclo, la agrego al set.
  3. Repito el paso hasta conectar todos los nodos. Es decir, estén en el mismo árbol.

```
Kruskal(G):
1   E ← E ordenada por costo, de menor a mayor
2   T ← lista vacía
3   for e ∈ E :
4       if Agregar e a T no forma ciclo :
5           T ← T ∪ {e}
6   return T
```

- **Algoritmo de Dijkstra**:
  1. Comienzo visitando un nodo arbitrario s con costo 0 ($d[s] = 0$). Los demás nodos comienzan con costo ∞ $d[u \in V - \{s\}] = \infty$.
  2. Calculo los costos que hay hacia los otros nodos desde el nodo que visite. Guardo estos con sus nodos correspondientes.
  3. Visito el nodo con menor costo.
  4. Repito pasos 2 y 3 actualizando los costos correspondientes, hasta visitar todos los nodos.
  5. Es un algoritmo *greedy*.

```
Dijkstra(s):
    for u ∈ V − {s} :
        u.color ← blanco;  d[u] ← ∞;  π[u] ← ∅
    s.color ← gris;  d[s] ← 0;  π[s] ← ∅
    Q ← cola de prioridades vacía
    Insert(Q, s)
    while Q no está vacía :
        u ← Extract(Q)
        for v ∈ α[u] :
            if v.color = blanco ∨ v.color =gris :
                if d[v] > d[u] + cost(u, v) :
                    d[v] ← d[u] + cost(u, v);  π[v] ← u
                if v.color =blanco :
                    v.color ← gris;  Insert(Q, v)
        u.color ← negro
```

- **Algoritmo de Bellman-Ford**:
  1. Comienzo desde un nodo arbitrario s con costo 0 ($d[s] = 0$). Los demás nodos comienzan con costo ∞ $d[u \in V - \{s\}] = \infty$.
  2. El algoritmo posee $|V| - 1$ iteraciones.
  3. Por cada una de estas $|V| - 1$ iteraciones, visito todas las aristas $(u, v) \in E$.
  4. Si la suma de llegar al nodo v con el costo de (u, v) es menor al costo que tenía guardado anteriormente, actualizo el costo de llegar a ese nodo.
  5. NO es un algoritmo *greedy*.

```
BellmanFord(s):
1   for u ∈ V :
2       d[u] ← ∞;  π[u] ← ∅
3   d[s] ← 0
4   for k = 1 . . . |V| − 1 :
5       for (u, v) ∈ E :
6           if d[v] > d[u] + cost(u, v) :
7               d[v] ← d[u] + cost(u, v)
8               π[v] ← u
```