



# Recuperación de Información Multimedia

## Repaso de Estructuras de Datos

**CC5213 – Recuperación de Información Multimedia**

Departamento de Ciencias de la Computación

Universidad de Chile

Juan Manuel Barrios – <https://juan.cl/mir/> – 2019



# **Estructuras de Datos Básicas**

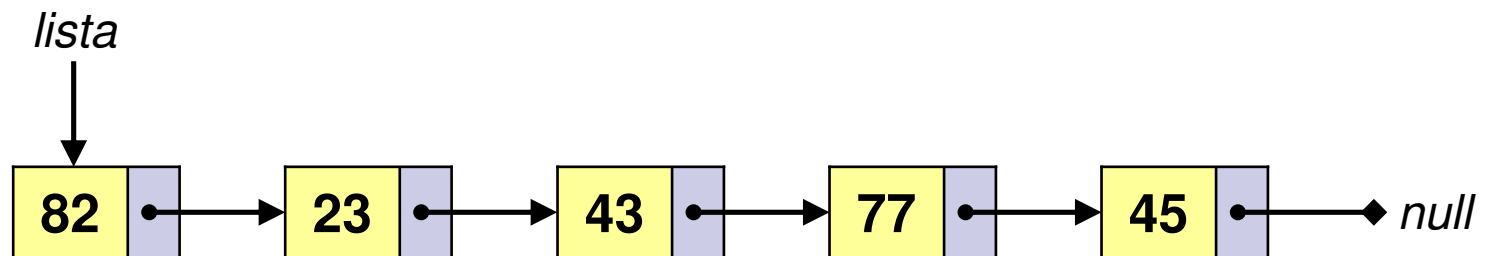
# Arreglo

- Secuencia contigua de elementos de un mismo tipo.
- Cada elemento se referencia por su posición (número entero).
- Tamaño fijo (usualmente)
- Permite leer cualquier elemento en tiempo constante (independiente del número de elementos).
- Costo lineal para insertar un elemento en cierta posición (desplazar elementos hacia la derecha).

Elementos	<b>82</b>	<b>23</b>	<b>43</b>	<b>77</b>	<b>45</b>	<b>76</b>	<b>47</b>	<b>97</b>	<b>24</b>
Índices	0	1	2	3	4	5	6	7	8

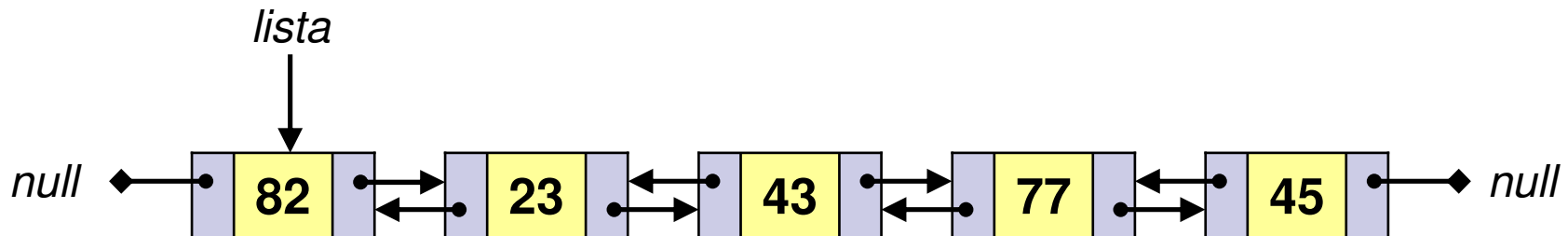
# Lista Enlazada (Linked List)

- Serie de objetos (nodos) conectados entre sí secuencialmente por referencias o punteros.
- Cada nodo se compone de un elemento y una referencia al nodo siguiente.
- Costo constante para leer, insertar o eliminar el primer elemento de la lista.
- Acceder un elemento ubicado en la posición  $i$  tiene un costo proporcional a  $i$ .



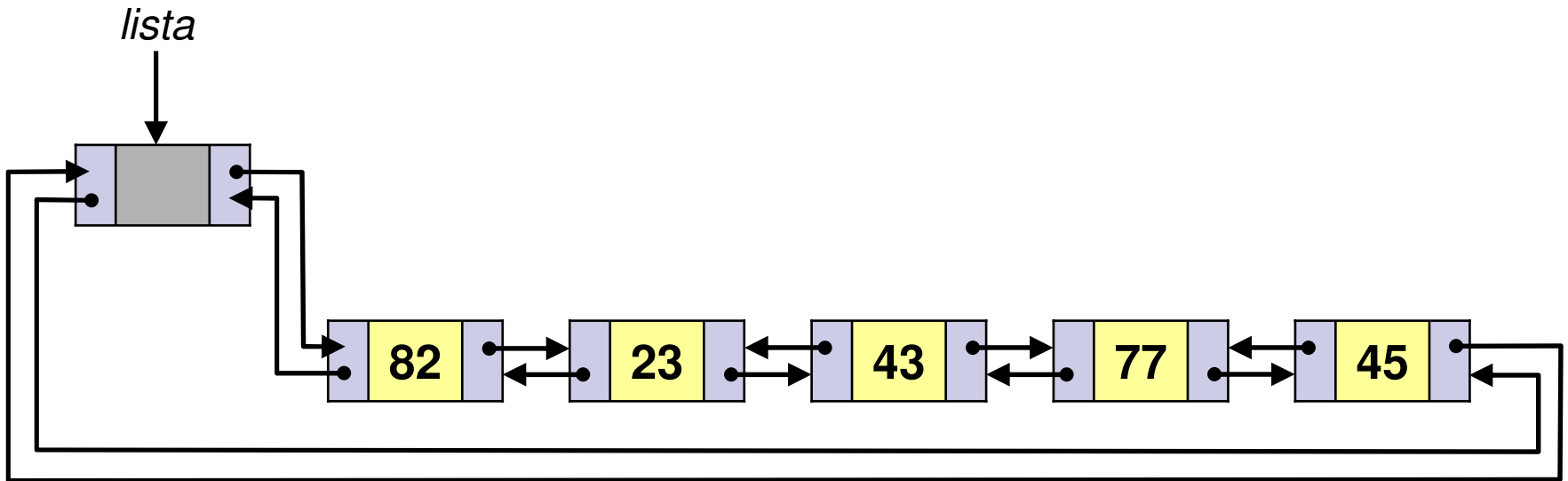
# Lista Enlazada (Linked List)

- Lista doblemente enlazada.
  - Cada nodo además referencia al nodo anterior.
  - Simplifica modificar la lista en una posición dada.



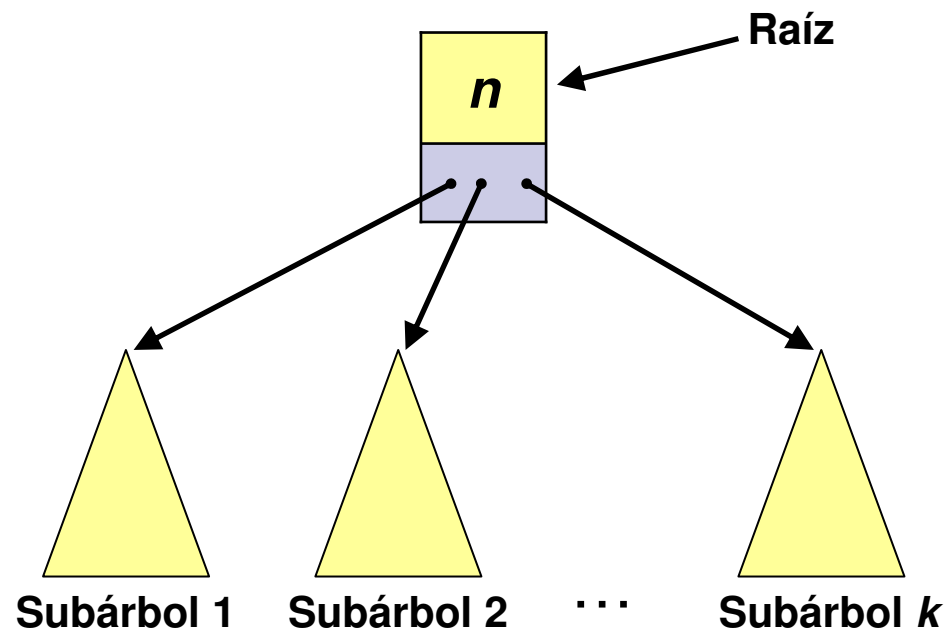
# Lista Enlazada (Linked List)

- Lista con cabecera.
  - Se utiliza un nodo sin datos para iniciar la lista.
  - Simplifica acceder al final de la lista.



# Árbol

- Conjunto de valores ordenados jerárquicamente.
- Cada nodo se compone de un elemento y una lista de referencias a sus nodos hijos.
  - Árbol k-ario: cada nodo tiene a lo más  $k$  hijos.



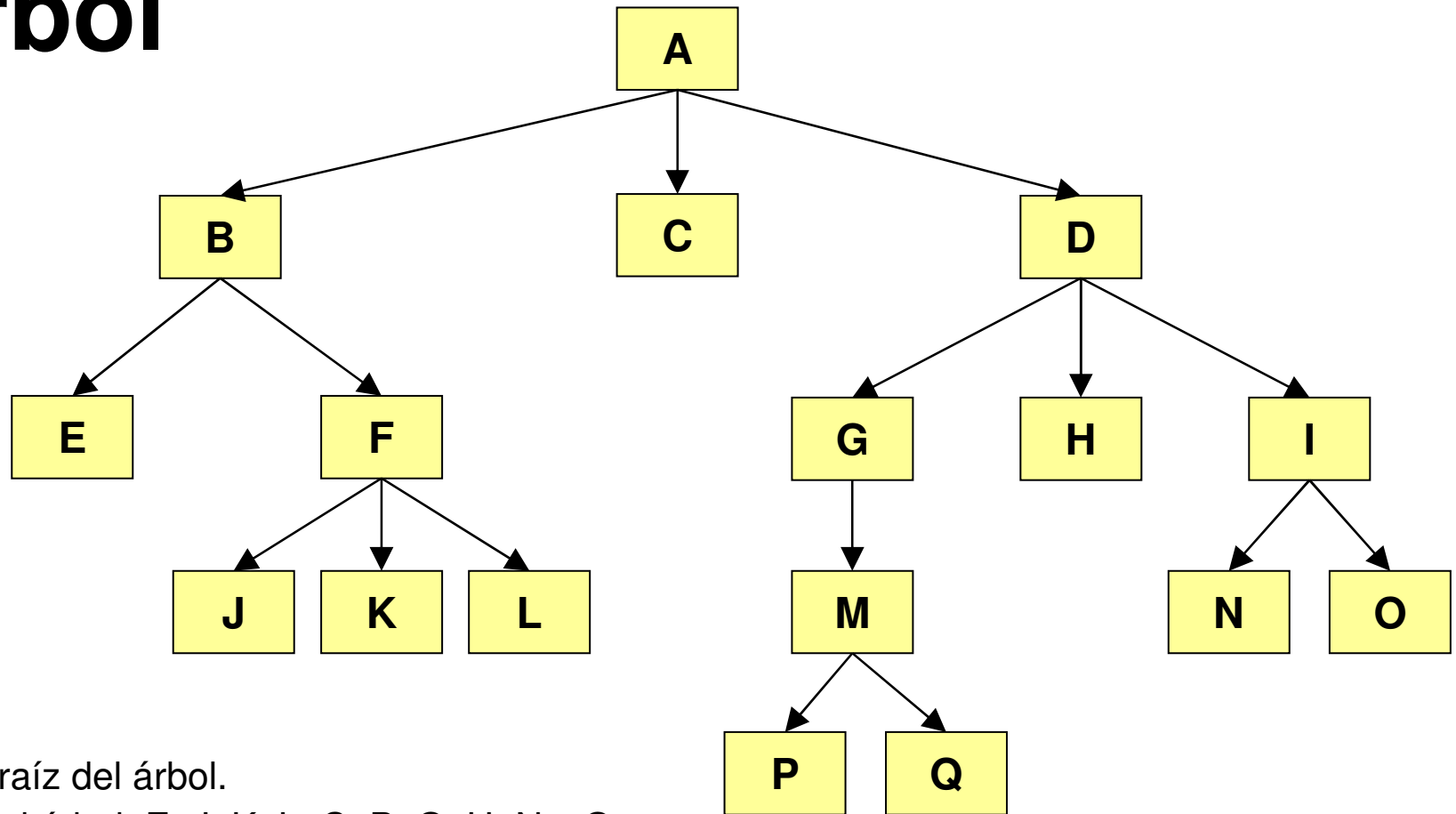


# Árbol

- Camino: Secuencia de pasos de un padre hacia sus descendientes.
- **En los árboles existe un único camino entre el nodo raíz y cualquier otro nodo del árbol.**
- Nodo interno: Nodo con al menos un hijo.
- Nodo externo (hojas): Nodo sin hijos.
- Profundidad de un nodo: Largo del camino entre la raíz y el nodo.
- Altura de un nodo: Largo máximo del camino entre el nodo y todos sus nodos descendientes.



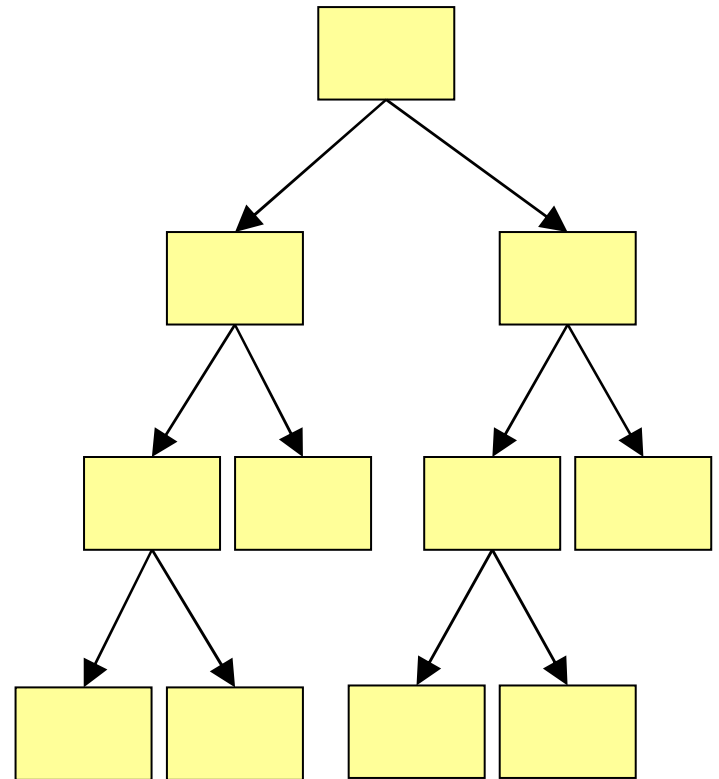
# Árbol



- A es la raíz del árbol.
- Hojas del árbol: E, J, K, L, C, P, Q, H, N y O.
- El largo del camino desde A a J es 3.
- L no es descendiente de D (no existe un camino entre D y L)
- La profundidad de C es 1, de F es 2 y de Q es 4.
- La altura del árbol es 4.

# Árbol Binario

- Árbol donde cada nodo tiene a lo más dos hijos (izquierdo y derecho)
- Un árbol binario de altura  $h$  tiene a lo más  $2^h$  hojas.
- Un árbol binario de  $t$  hojas tiene altura  $h \geq \log_2 t$ .





# **Tipos de Datos Abstractos (TDA)**



# Tipos de Datos Abstractos (TDA)

- Modelo de una componente que permite realizar operaciones sobre datos
- Cada operación tiene bien definida su entrada y salida (una “interfaz”)
- Se puede implementar con diferentes estructuras de datos, cada una con ventajas y desventajas para las distintas operaciones.
- Ejemplos:
  - Cola, Pila, Lista, Cola de Prioridad, Diccionario



# TDA: Cola (Queue)

- Lista de elementos donde se insertan elementos la final y se extraen desde el principio (First In, First Out).
- Operaciones:
  - ☐ Insertar elemento (enqueue)
  - ☐ Obtener elemento sin eliminarlo (front o peek)
  - ☐ Eliminar elemento (dequeue)
  - ☐ Preguntar si esta vacía (isEmpty)



# TDA: Pila (Stack)

- Lista de elementos donde sólo se puede extraer el último insertado (Last In, First Out).
- Operaciones:
  - ☐ Insertar elemento (push)
  - ☐ Obtener elemento sin eliminarlo (top o peek)
  - ☐ Eliminar elemento (pop)
  - ☐ Preguntar si esta vacía (isEmpty)



# TDA: Lista (List)

- Contiene secuencias de valores, con posibles repeticiones.
- Operaciones:
  - Agregar elemento (al inicio, al final, en posición  $k$ )
  - Obtener elemento (al inicio, al final, en posición  $k$ )
  - Eliminar elemento (al inicio, al final, en posición  $k$ )
  - Buscar posición del elemento  $x$
  - Cantidad de elementos en la lista



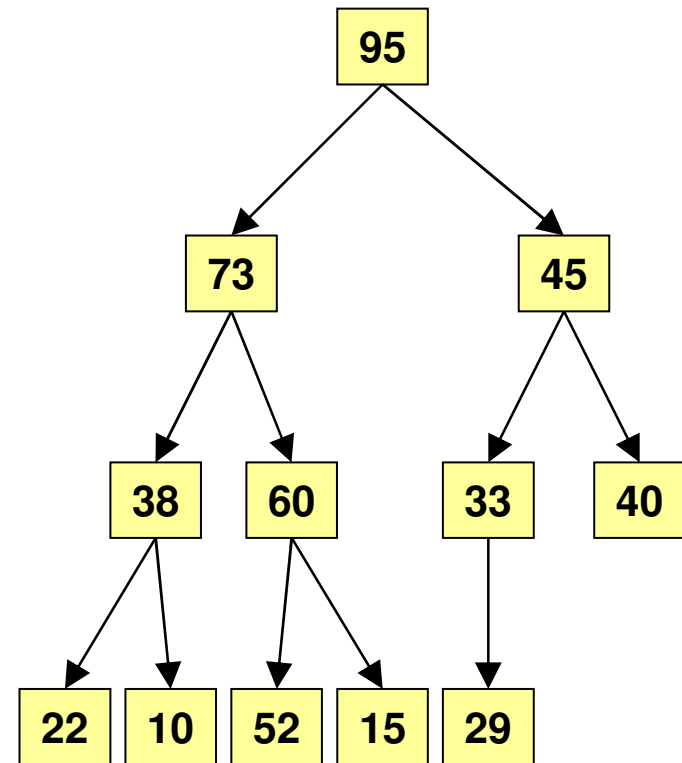
# TDA: Cola de Prioridad (Priority Queue)

- Lista donde cada elemento tiene asociado un valor de prioridad y se extraen elementos de mayor a menor prioridad.
- Operaciones:
  - ☐ Insertar elemento con su prioridad
  - ☐ Obtener elemento de mayor prioridad sin eliminarlo
  - ☐ Eliminar elemento de mayor prioridad
  - ☐ Preguntar si esta vacía
- Al momento de creación se puede definir usar orden inverso (extraer elementos con menor prioridad primero).



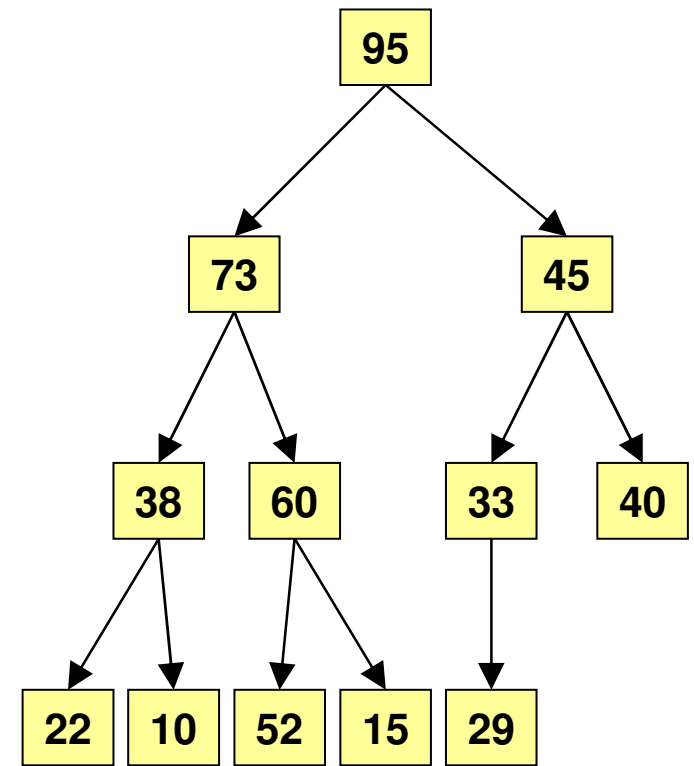
# Estructura de Datos: Heap

- Permite implementar una Cola de Prioridad.
- Árbol binario que cumple dos propiedades:
  1. Cada nodo cumple que su valor es mayor o igual al valor de sus dos nodos hijos (max-heap), o menor o igual a sus dos hijos (min-heap)
  2. Todos sus niveles están completos, excepto el último que puede estar parcialmente ocupado de izquierda a derecha.



# Heap

- La primera propiedad permite mantener en la raíz el máximo elemento (max-heap) o el mínimo (min-heap).
- Para eliminar la raíz se traslada el valor del último nodo a la raíz y mediante intercambios padre-hijo se corrige la propiedad.
- Para insertar un nuevo elemento se agrega un nodo al final del último nivel y mediante intercambios padre-hijo se corrige la propiedad.

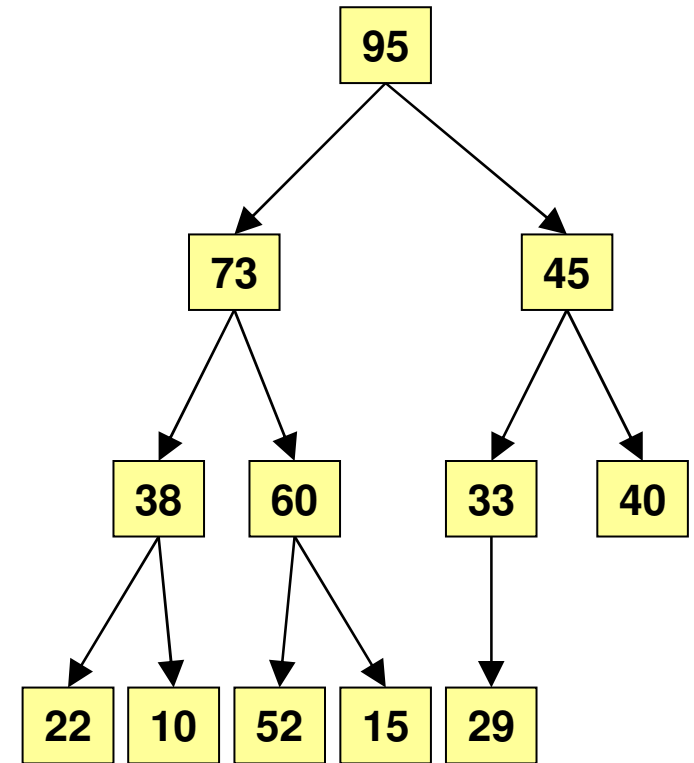


# Heap

- La segunda propiedad permite que los nodos del árbol se puedan almacenar en un arreglo.

□ Para el nodo  $A[i]$ :

- Nodo Padre:  $A[(i-1)/2]$
- Nodos Hijos:  $A[2i+1]$  y  $A[2i+2]$



<b>A</b>	<b>95</b>	<b>73</b>	<b>45</b>	<b>38</b>	<b>60</b>	<b>33</b>	<b>40</b>	<b>22</b>	<b>10</b>	<b>52</b>	<b>15</b>	<b>29</b>
<i>Indice</i>	0	1	2	3	4	5	6	7	8	9	10	11



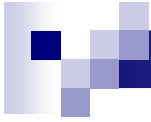
# TDA: Diccionario (Dictionary)

- Permite almacenar elementos y preguntar por la existencia de ellos. No se permiten repeticiones.
- Operaciones:
  - ☐ Insertar elemento
  - ☐ Buscar elemento
  - ☐ Eliminar elemento



# TDA: Map

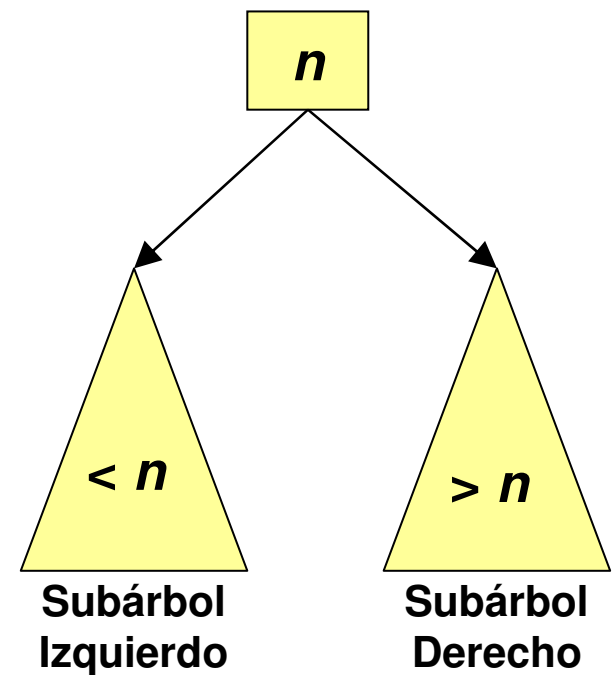
- Similar a un Diccionario, excepto que además del elemento se puede almacenar un valor asociado.
- Se conoce como Map o Associative Array ya que almacena pares (llave-valor).
- Operaciones:
  - ☐ Insertar llave y su valor asociado
  - ☐ Buscar llave y retornar su valor asociado
  - ☐ Eliminar llave y su valor asociado



# **Estructuras de Datos para implementar el TDA Diccionario**

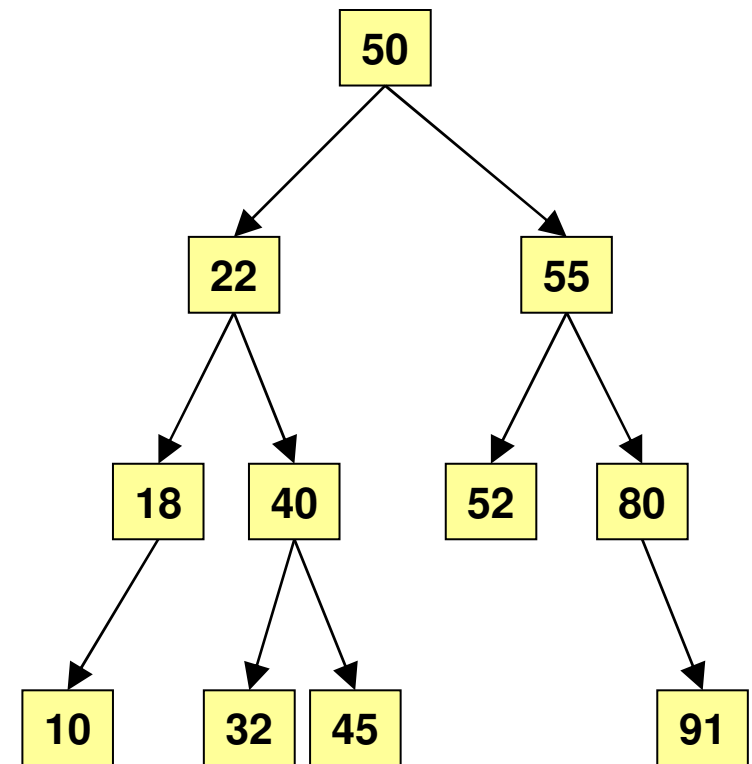
# Árbol de Búsqueda Binaria (BST)

- Permite implementar el TDA Diccionario.
- Árbol binario que cumple que en todos sus nodos:
  - Los elementos en el subárbol izquierdo son menores al nodo.
  - Los elementos en el subárbol derecho son mayores al nodo.



# Imprimir un ABB

```
class Nodo {  
    int valor;  
    Nodo izquierda, derecha;  
}  
void imprimir() {  
    imprimirRecursivo(raiz);  
}  
void imprimirRecursivo(Nodo nodo) {  
    if (nodo == null)  
        return;  
    imprimirRecursivo(nodo.izquierda);  
    print(nodo.valor);  
    imprimirRecursivo(nodo.derecha);  
}
```





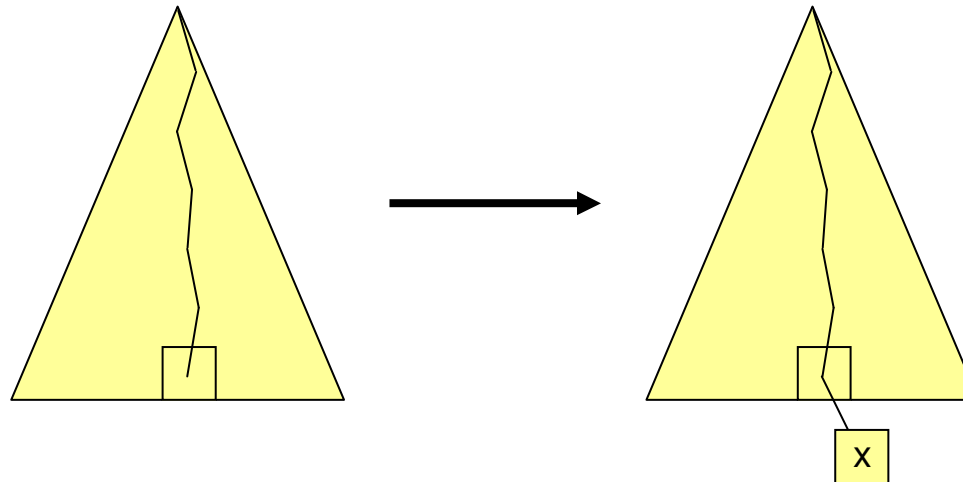


# Buscar en un ABB

```
class Nodo {
    int valor;
    Nodo izquierda, derecha;
}
boolean buscar(int valorBuscado) {
    return buscarRecursoivo(raiz, valorBuscado);
}
boolean buscarRecursoivo(Nodo nodo, int valorBuscado) {
    if (nodo == null) {
        return false;
    } else if (nodo.valor == valorBuscado) {
        return true;
    } else if (valorBuscado < nodo.valor) {
        return buscarRecursoivo(nodo.izquierda, valorBuscado);
    } else {
        return buscarRecursoivo(nodo.derecha, valorBuscado);
    }
}
```

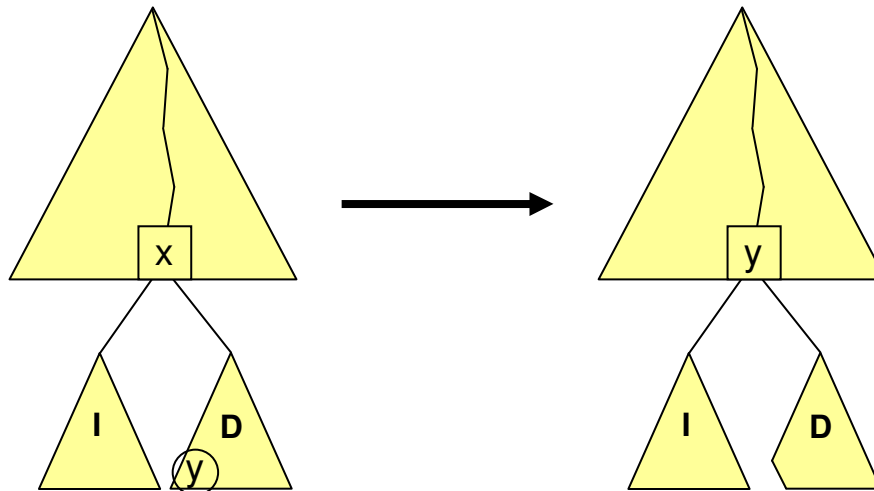
# Insertar en un ABB

- Buscar el valor **x** en el árbol hasta llegar a una hoja (**x** no existe), insertar **x** como hijo de esa hoja



# Eliminar en un ABB

- Buscar el nodo **x** que contiene el valor:
  - Si **x** no tiene hijos se elimina (padre de **x** referencia a *null*)
  - Si **x** tiene un hijo, el padre de **x** referencia al nodo hijo de **x**
  - Si **x** tiene dos hijos se busca en el subárbol derecho el nodo mínimo **y**, se elimina el nodo **y** (no puede tener dos hijos), se reemplaza el valor en **x** por el valor en **y**



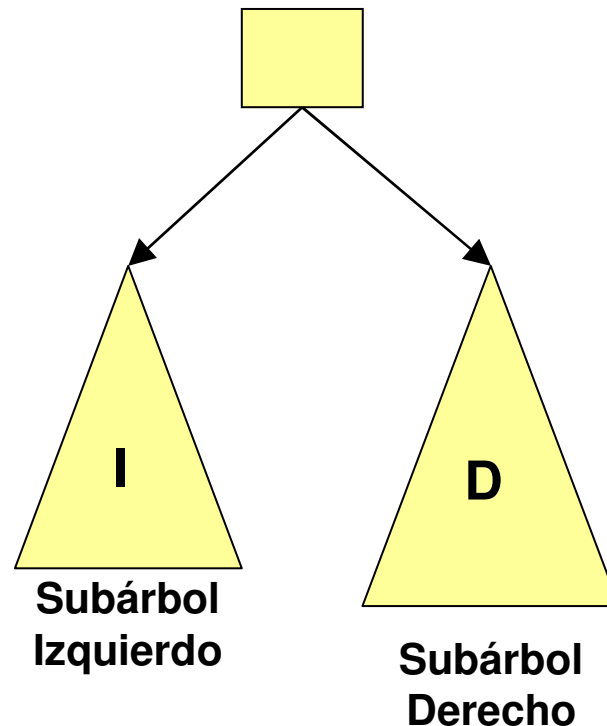


# Árboles Balanceados

- Mantienen reducida la altura del árbol para que las operaciones de búsqueda tengan costo logarítmico, incluso en el peor caso
- Ejemplos:
  - Árbol AVL
  - Árbol 2-3
  - Árbol B
  - Árbol Digital (Trie)

# Árbol AVL

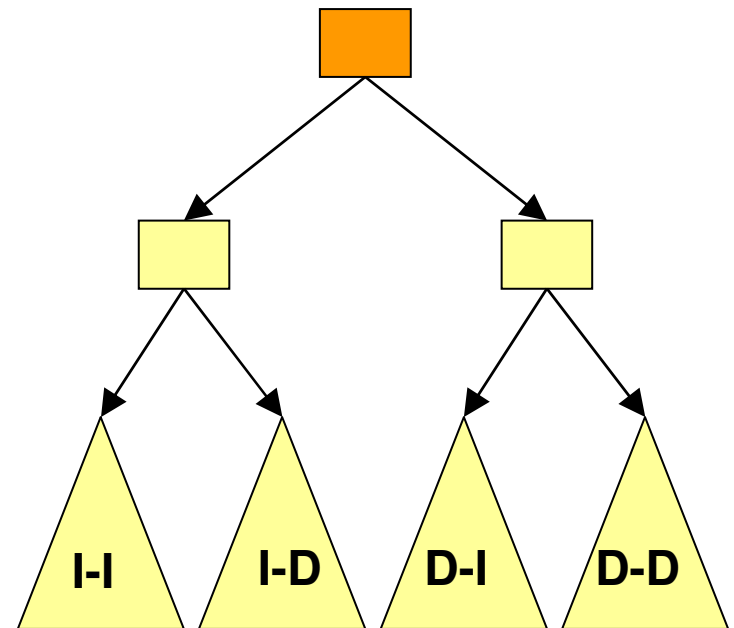
- Condición: En todo nodo del árbol se cumple que la diferencia de altura de sus dos subárboles es menor o igual que 1



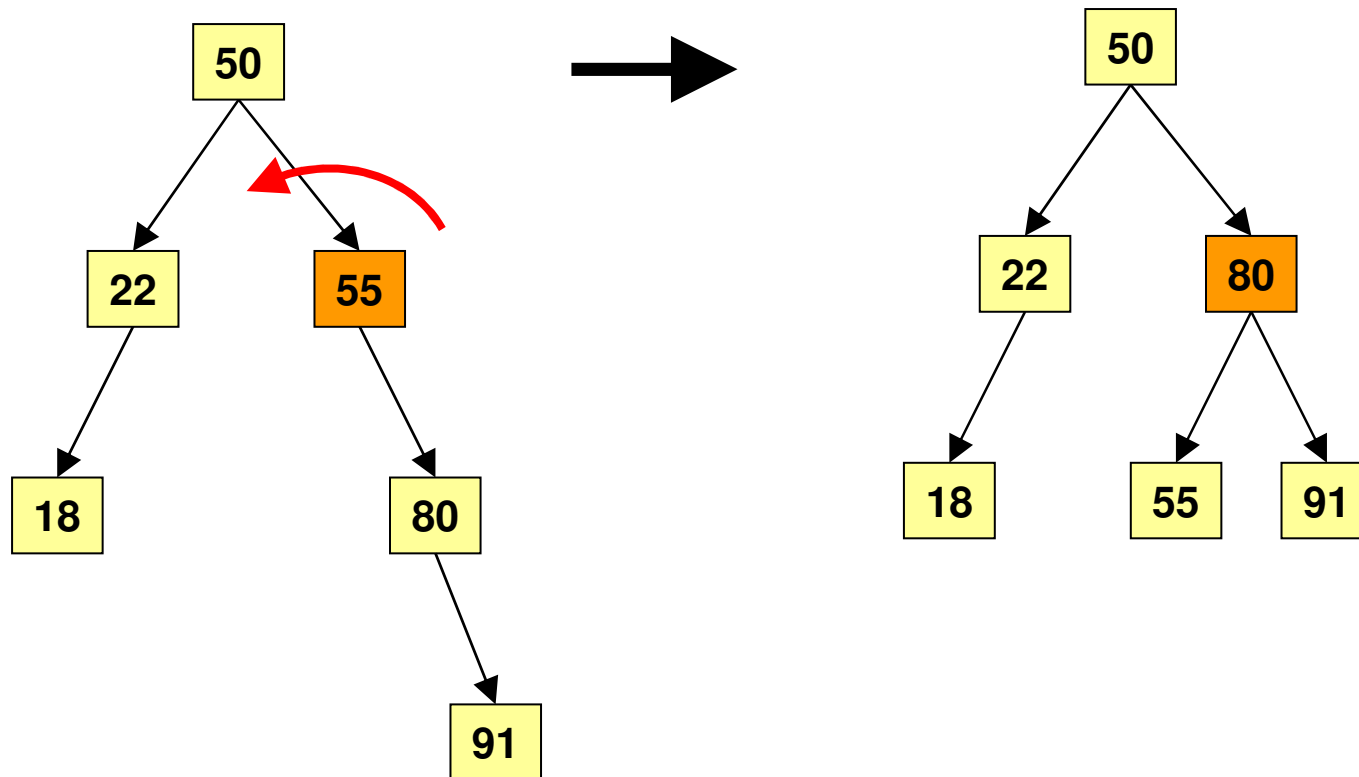
$$| \text{altura}(\mathbf{I}) - \text{altura}(\mathbf{D}) | \leq 1$$

# Árbol AVL

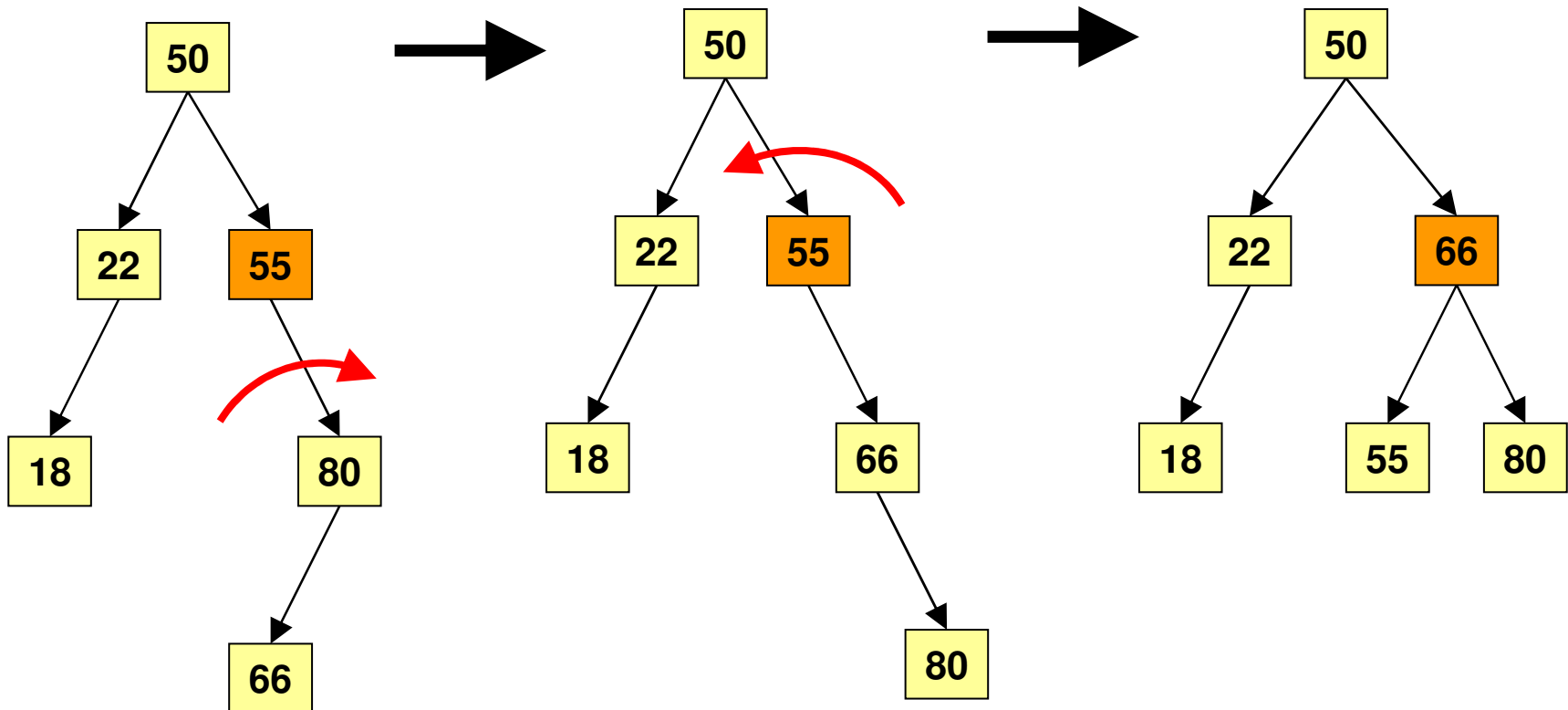
- Cuando un nodo deja de cumplir la propiedad se puede corregir por medio de “rotaciones” de los nodos
- Se debe determinar el sub-subárbol más largo (el que causa el problema):
  - Si es **I-I** o **D-D** (por afuera) hacer rotación simple
  - Si es **I-D** o **D-I** (por adentro) hacer rotación doble



# Ejemplo Rotación Simple



# Ejemplo Rotación Doble

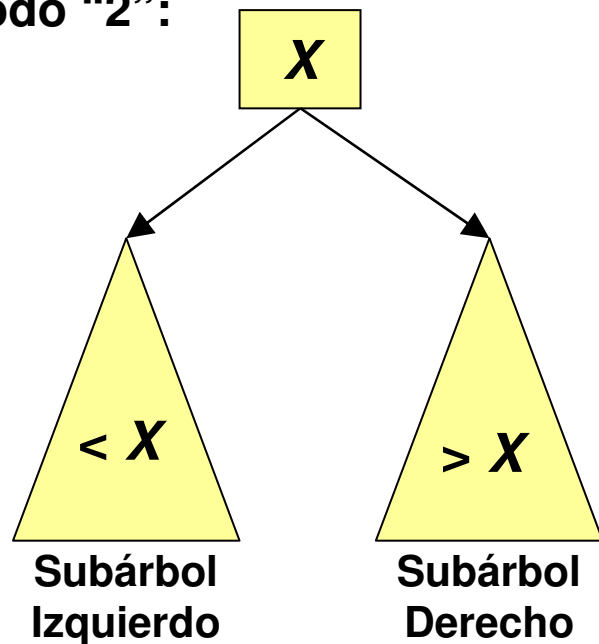




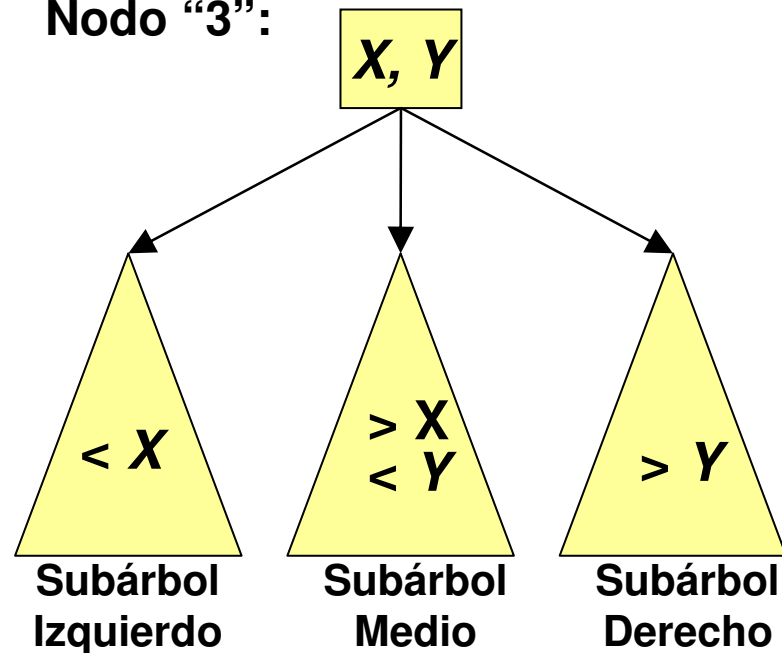
# Árbol 2-3

- El árbol puede contener dos tipos de nodos:
  - Nodos de 2 hijos y un valor (un nodo binario)
  - Nodos de 3 hijos y dos valores

Nodo “2”:



Nodo “3”:

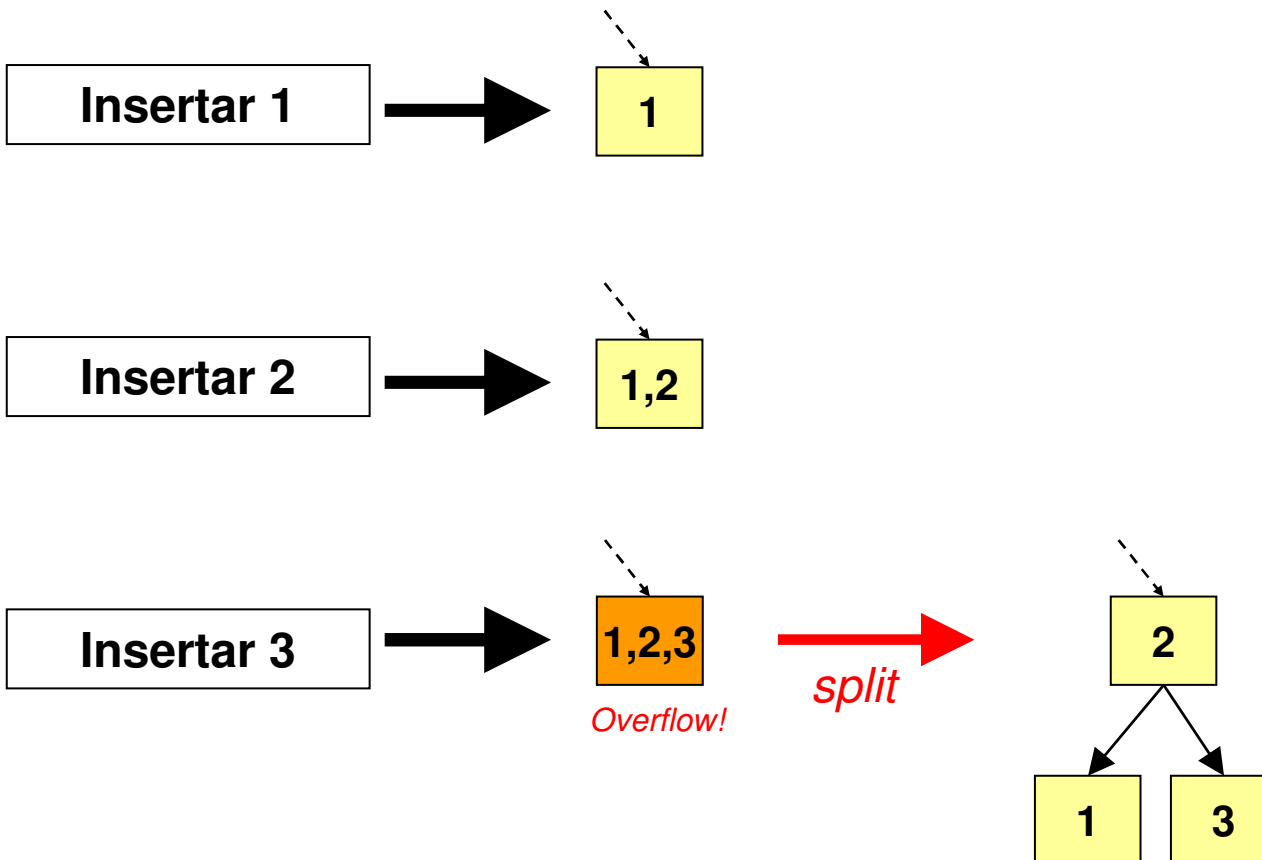




# Inserción en Árbol 2-3

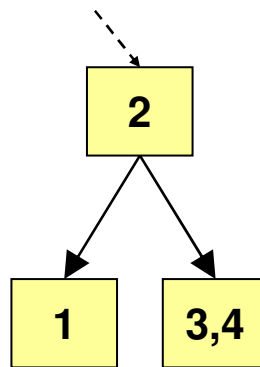
- Para insertar un valor  $X$ , primero hacer una búsqueda del elemento. Si se encuentra, no hay que hacer nada
- Si no se encuentra se agrega el elemento al último nodo visitado  $n$  (hoja):
  - Si  $n$  es nodo “2”, se agrega  $X$  y ahora es nodo “3”
  - Si  $n$  es nodo “3”, se agrega  $X$  y ocurre un *overflow*
    - Un *split* divide  $n$  en un nodo “2” con el menor valor, otro nodo “2” con el mayor valor y el valor central  $C$  sube un nivel y se agrega al nodo padre
    - Si el nodo padre es “2”, se agrega  $C$  y ahora es nodo “3”
    - Si el nodo padre es “3”, se agrega  $C$  y hace *overflow*, se continúa el split recursivamente hasta posiblemente crear un nuevo nodo raíz

# Inserción en Árbol 2-3

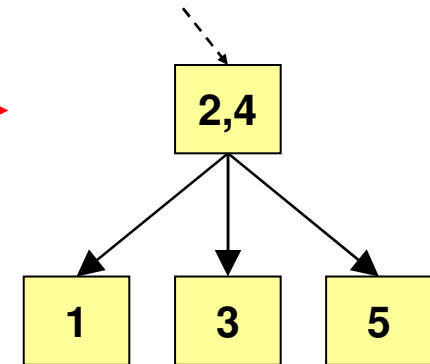
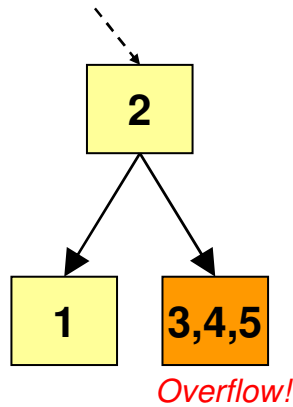
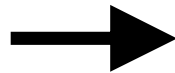




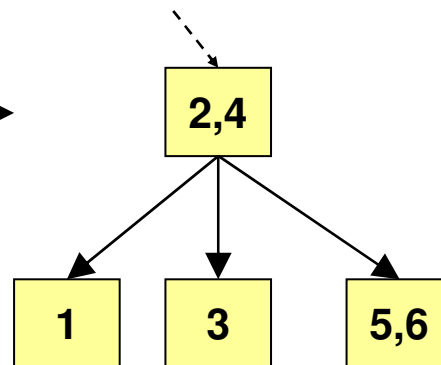
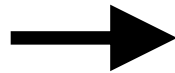
Insertar 4



Insertar 5

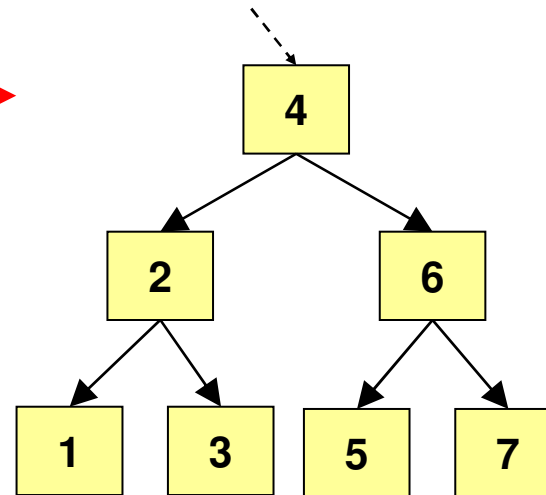
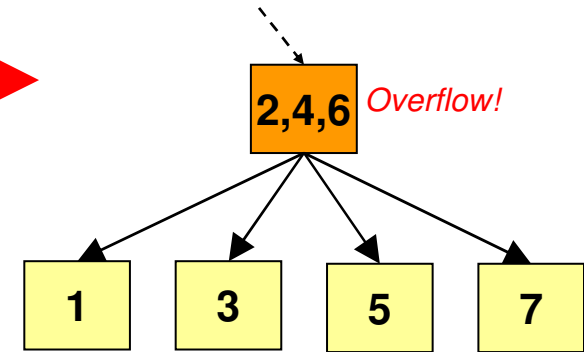
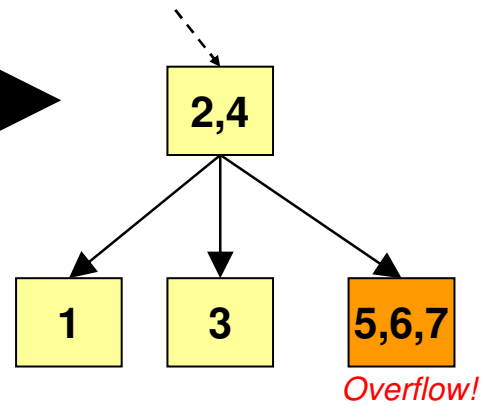


Insertar 6





Insertar 7





# Árbol 2-3

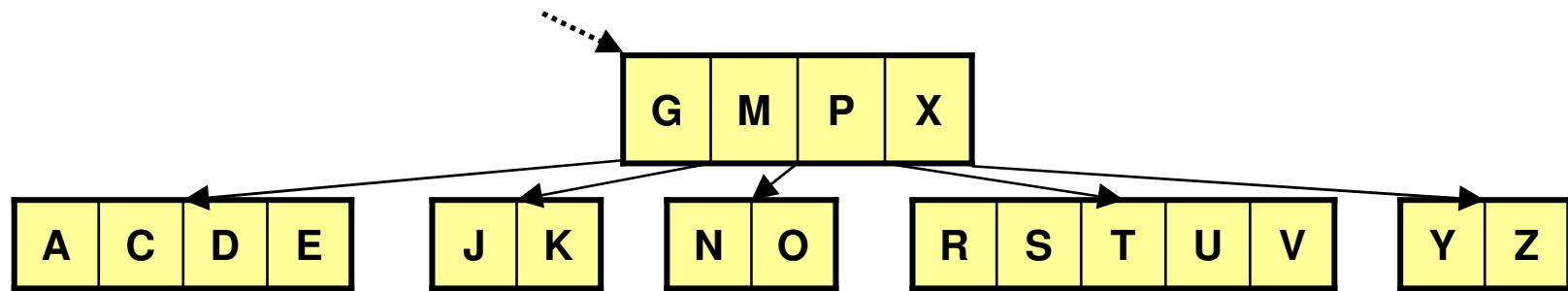
- El árbol 2-3 es perfectamente balanceado
  - Para cada nodo todos sus subárboles tienen la misma altura
  - Todos los caminos desde la raíz a alguna hoja tienen el mismo largo
- Los costos de búsqueda, inserción y eliminación tienen un costo logarítmico



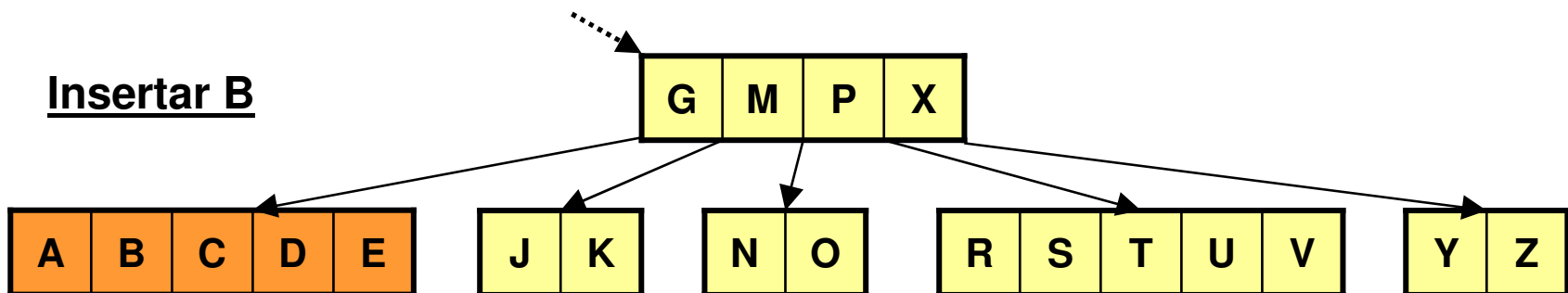
# Árbol B

- Cada nodo puede tener una cantidad máxima de  $p$  elementos ordenados (orden  $p$ )
  - Generalización del árbol 2-3 (donde  $p=3$ )
  - El split crea dos nodos cada uno con  $p/2$  elementos y rebalsa un elemento al padre.
  - El tamaño de  $p$  se fija según un tamaño de página de disco.
- Árbol B\*: Aumenta el nivel de ocupación mínimo. Mueve elementos entre nodos hermanos. El split agrupa dos nodos completos y crea tres, para lograr una ocupación mínima de 66%.
- Árbol B+: Los datos sólo se guardan en las hojas. Los nodos internos sólo contienen separadores para la búsqueda.

# Ejemplo B-Tree de orden 5



Insertar B

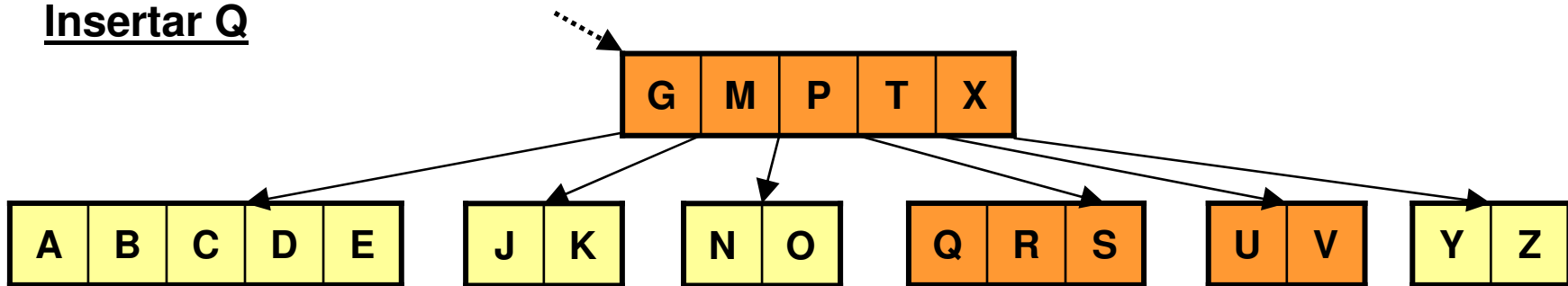


Insertar Q...



# Ejemplo B-Tree de orden 5

Insertar Q

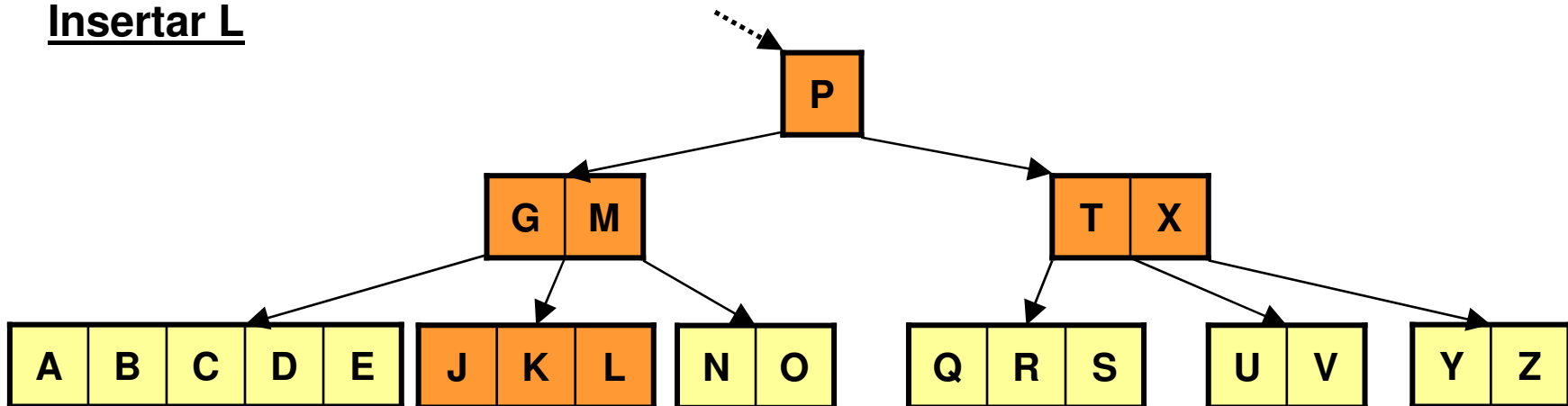


- Al agregar Q se obtiene un nodo QRSTUV que hace overflow
- Se hace split en dos nodos QRS y UV y el elemento T se agrega al padre

Insertar L...

# Ejemplo B-Tree de orden 5

Insertar L

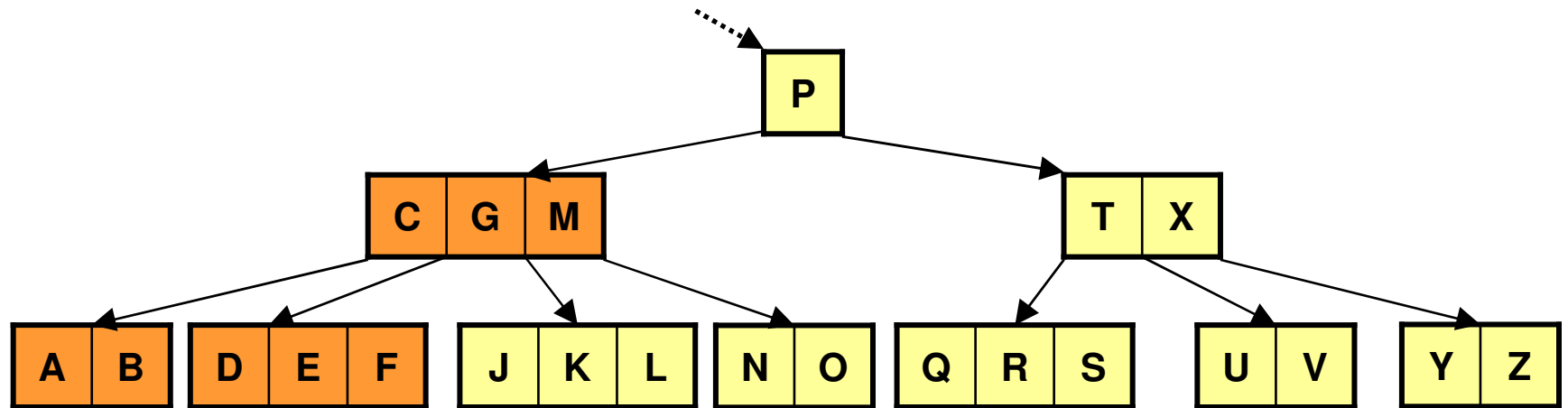


- Variante: Para evitar volver hasta la raíz con splits encadenados, mientras está bajando si se encuentra con un nodo completo dividirlo de inmediato

Insertar F...

# Ejemplo B-Tree de orden 5

## Insertar F



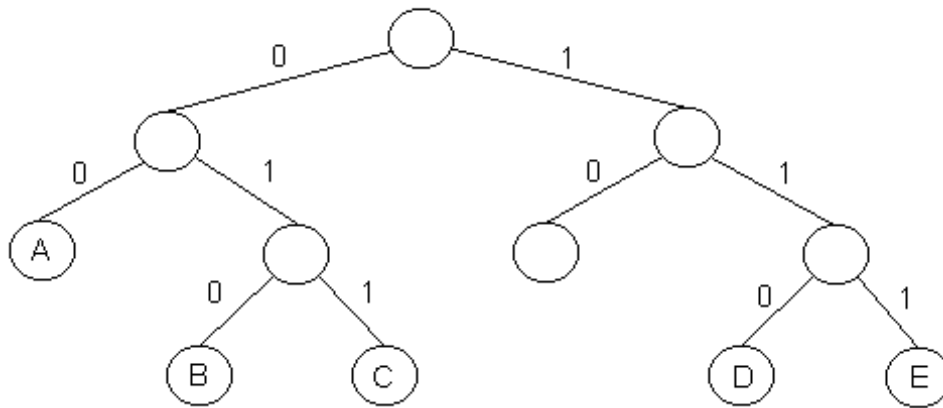
- Se obtiene un nodo ABCDEF que se divide en dos y C es el elemento que se agrega al padre



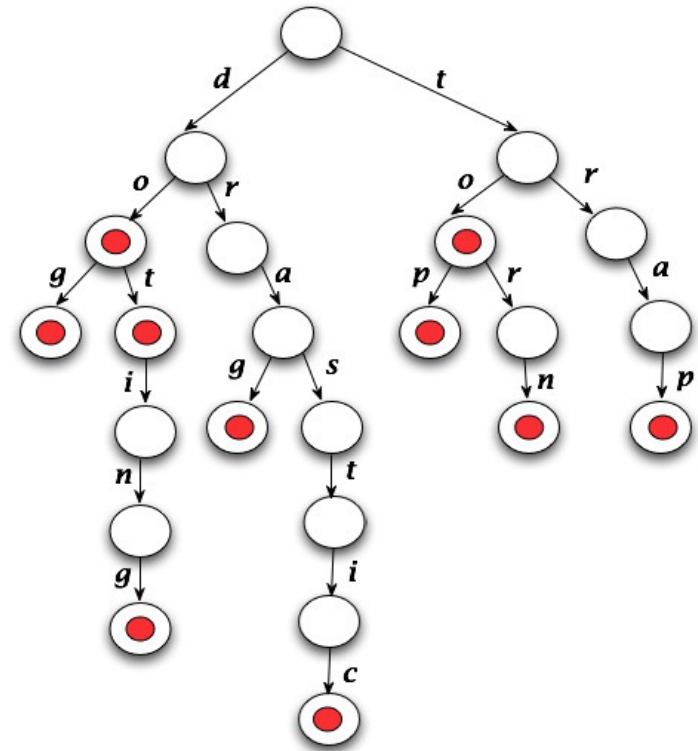
# Árbol Digital (Trie)

- Árbol digital o Árbol de prefijos o Trie
- Los elementos se representan como secuencias de símbolos
  - Strings son secuencias de caracteres: “hola” = h, o, l, a
  - Números enteros son secuencias de bits: 14 = 1, 1, 1, 0
- Todas las secuencias de símbolos se guardan en un único árbol compartiendo prefijos comunes
- Las secuencias corresponden a los caminos desde la raíz a cada nodo
- Cada nodo marca si su secuencia es válida o no
- Variante: Comprimir caminos, representando cadenas de símbolos (Árbol Patricia)

# Árbol Digital (Trie)



A = 00100  
B = 01000  
C = 01111  
D = 11000  
E = 11101





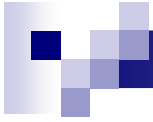
# Árbol Digital (Trie)

- Para strings una opción es usar un árbol de 26 hijos por nodo (o hasta 256 hijos si se usa un byte como símbolo)
  - Muy baja ocupación de los nodos
  - Una lista enlazada de hijos evita tener un array fijo por nodo, pero hace más lento avanzar por el árbol
- Opción 1: Reducir el tamaño del alfabeto: representar bytes como pares de 4 bits (nodos con 16 hijos aunque caminos con el doble de largo)
- Opción 2: Calcular un nuevo alfabeto: Huffman Codes obtiene una representación binaria de cada símbolo según su probabilidad de aparición, generando cadenas más cortas para los más probables (ej: a=10, x=110101)

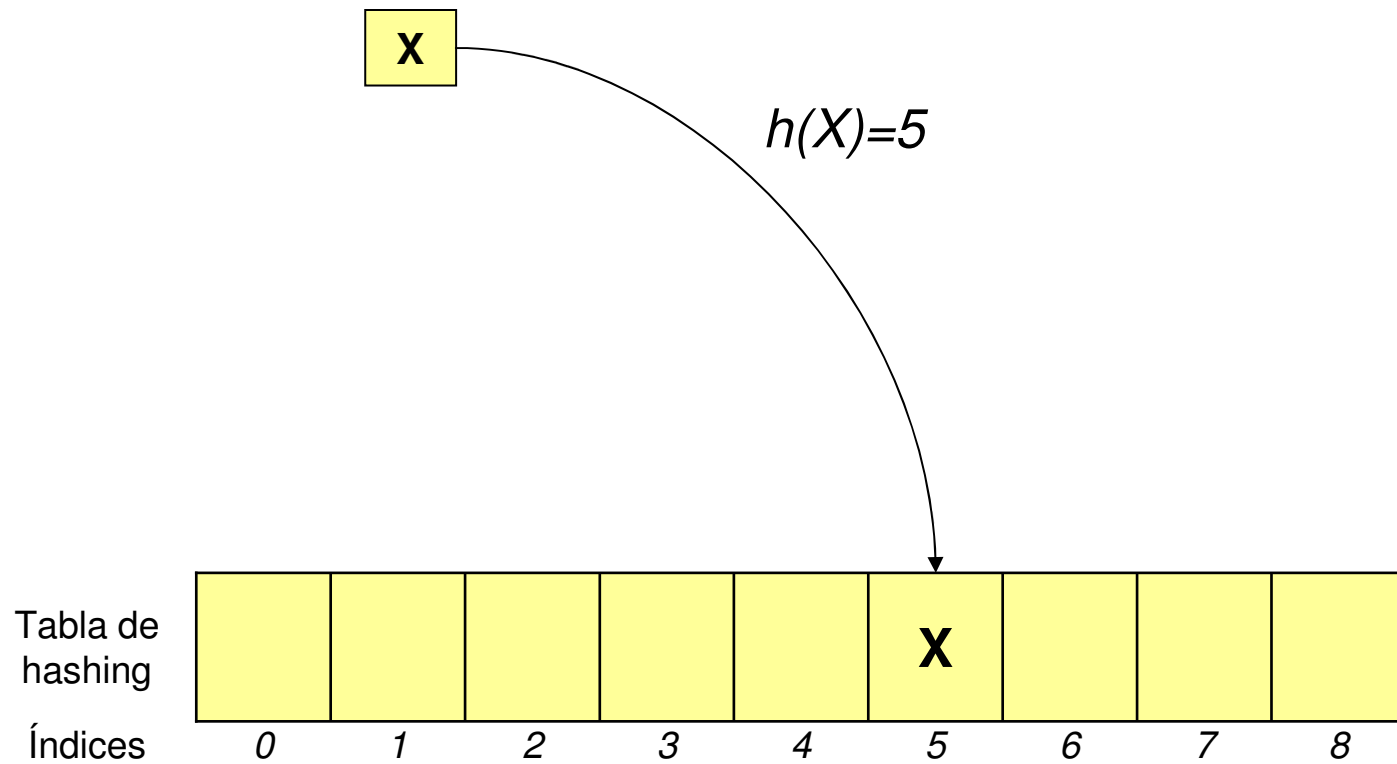


# Hashing

- Guardar todos los valores en un arreglo de tamaño fijo  $m$  predefinido
- Definir una función  $h$  (función de hash) que para cada elemento  $X$  calcula un valor  $h(X)$  como un entero entre 0 y  $m-1$
- El valor  $h(X)$  es la posición donde almacenar  $X$  en la tabla
- Propiedades deseadas para  $h$ :
  - Debe depender de todo el contenido de  $X$
  - Debe retornar valores uniformemente distribuidos entre 0 y  $m-1$  (función pseudoaleatoria)



# Hashing



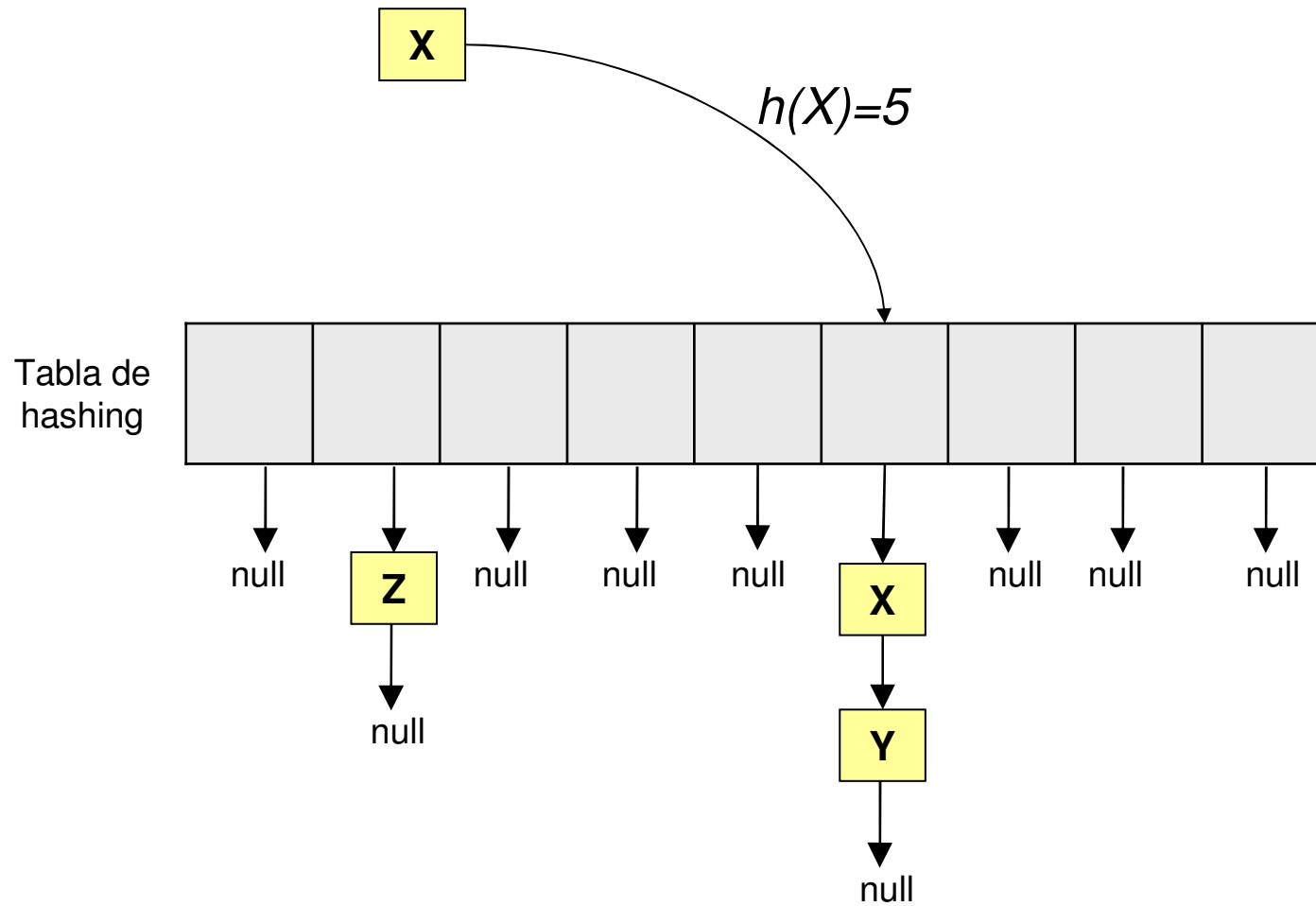




# Hashing

- Cuando  $X \neq Y$  pero  $h(X) = h(Y)$  se dice que ocurre una colisión
  - Encadenamiento: La tabla de hashing es un arreglo de listas enlazadas con todos los elementos asignados a la misma ubicación
    - Factor de carga: cantidad de elementos / tamaño de la tabla de hashing
  - Direccionamiento abierto: No usar espacio extra, si no que intentar con otra función de hash (por ejemplo, con  $h(X) + 1$ )

# Encadenamiento

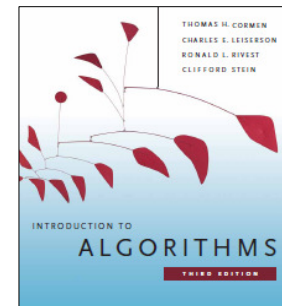


# Material de Estudio

- **Introduction to Algorithms 3rd edition.**

Cormen et al. 2009.

- ☐ Cap 6 (heaps)
- ☐ Cap 10 (queue, tree)
- ☐ Cap 11 (hashtable)
- ☐ Cap 12 (BST)
- ☐ Cap 18 (B-Tree)



- **Material del curso CC3001-Algoritmos y Estructuras de Datos**

- ☐ <https://users.dcc.uchile.cl/~bebustos/apuntes/cc3001/>