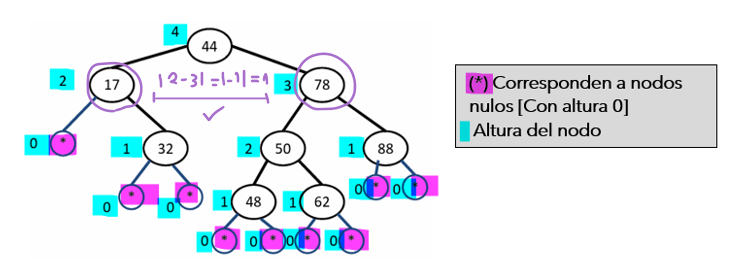


RESUMEN DE ARBOLES DE BUSQUEDA Y PROCESAMIENTO DE TEXTO:

* El arbol binario de busqueda permite implementar conjuntos y mapeos con un tiempo de operaciones buscar, inserter y eliminar con orden logaritmico en la cantidad de elementos en promedio 🡪  **[O(log(n)]**
* En el peor caso las operaciones tienen orden lineal en la cantidad de elementos (cuando las inserciones se realizaron en forma ascendente o descendente en cuyo caso el árbol degenera en una lista) 🡪 **[O(n)]**
* Hay estructuras alternativas que garantizan tiempo de acceso de orden logarítmico en la cantidad de elementos y se los conoce como árboles de búsqueda balanceados: **Árbol AVL, Árbol 2-3 y Árbol B**

**ARBOLES AVL:**

* Agregaremos una corrección al árbol binario de búsqueda para mantener una altura del árbol proporcional al logaritmo de la cantidad de nodos del árbol
* Recordemos que el tiempo de búsqueda, inserción y borrado en un árbol binario de búsqueda es lineal en la altura del árbol
* Entonces, si n=cantidad de elementos de un arbol T, tendríamos así que **T(n) = O(log2(n)) [dependiendo de n]**
* **Propiedad del balance de la altura: Para cada nodo interno v de T, las alturas de los hijos difieren en a lo sumo 1**
* Cualquier árbol binario de búsqueda que satisface esta propiedad se dice "árbol AVL**" [Propiedad que se le agrega al ABB]**



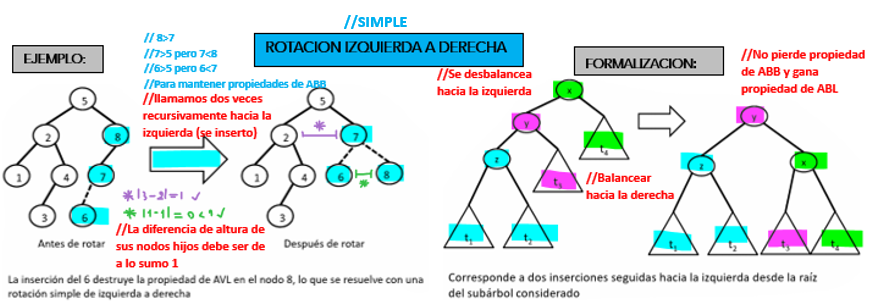
**ACLARACIONES:**

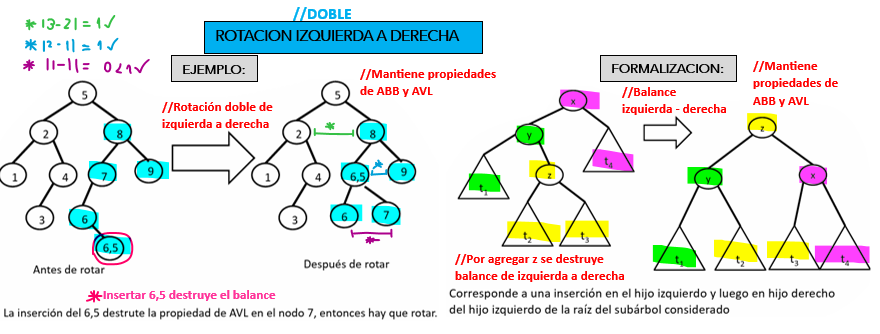
* Los nodos nulos tienen altura 0
* Como un AVL es un árbol binario de búsqueda, la operación de búsqueda no sufre alteraciones
* Las únicas operaciones para modificar son la inserción y eliminación, las cuales deben verificar que se cumpla la propiedad de balance al final la operación
* Luego de cada modificación en un nodo, rebalancearán los hijos de dicho nodo en O(1) por medio de las llamadas “rotaciones”
* Las modificaciones se hacen desde la hoja donde se insertó el nodo hacia la raíz siguiendo el camino de llamadas recursivas **[POS-ORDEN]**
* Las eliminaciones las haremos perezosas (lazy): marcaremos los datos eliminados con un booleano

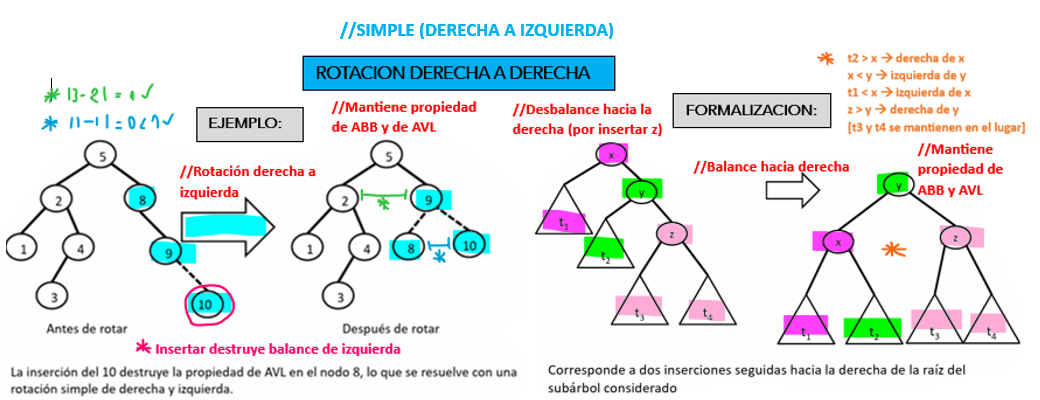
**ROTACIONES:**

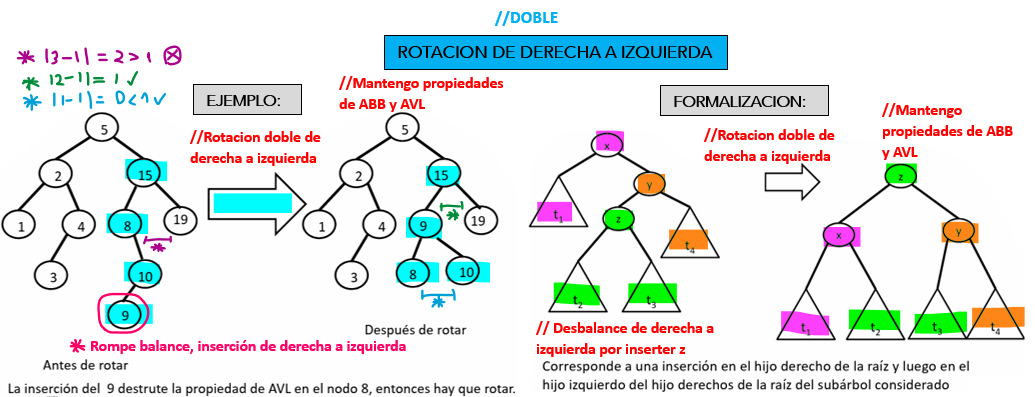
Son cuatro correspondientes a las cuatro combinaciones para la insercion de una clave a partir de un nodo raiz del subarbol considerado:

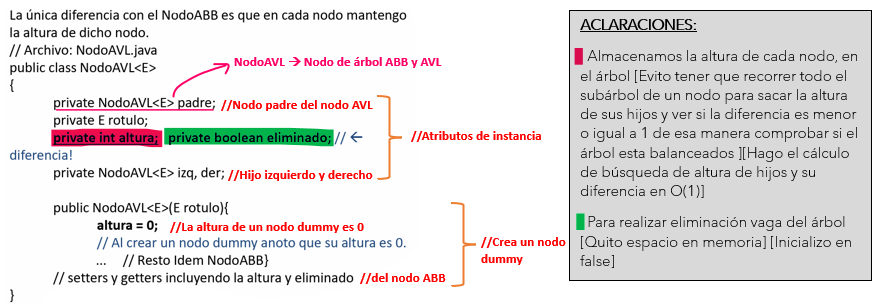
1. **IZQUIERDA-IZQUIERDA:** Rotacion simple de izquierda a derecha **[De un nodo hacemos dos llamados hacia la izquierda, en recursion]**
2. **IZQUIERDA-DERECHA:** Rotacion doble de izquierda a derecha **[Un llamado recursivo hacia la izquierda y otro hacia la derecha]**
3. **DERECHA-DERECHA:** Rotacion simple de derecha a izquierda **[Parados en un nodo hacemos dos llamados recursivos hacia la derecha, en inserccion]**
4. **DERECHA-IZQUIERDA:** Rotacion doble de derecha a izquierda **[Parados en un nodo hacemos llamado recursivo con el hijo derecho de v y luego con el hijo Izquierdo de v, en inserccion**

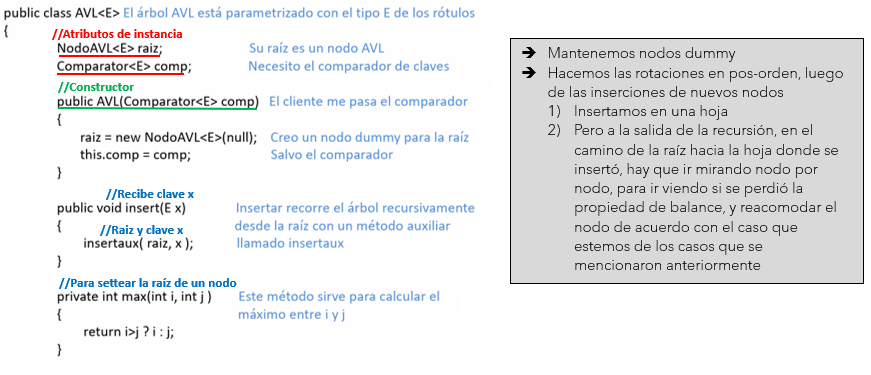


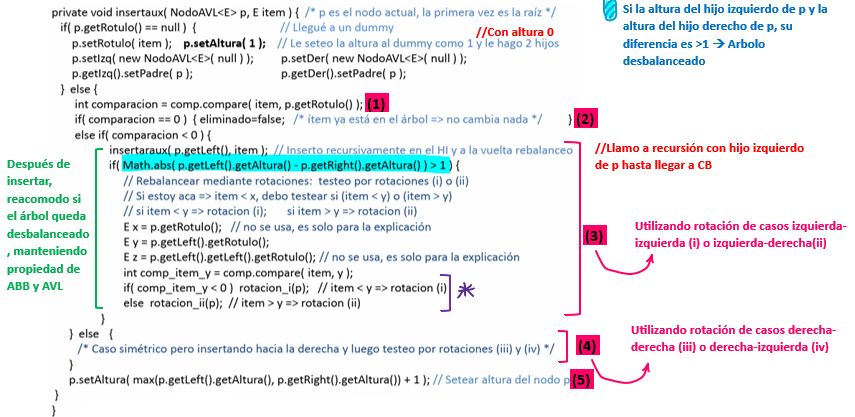


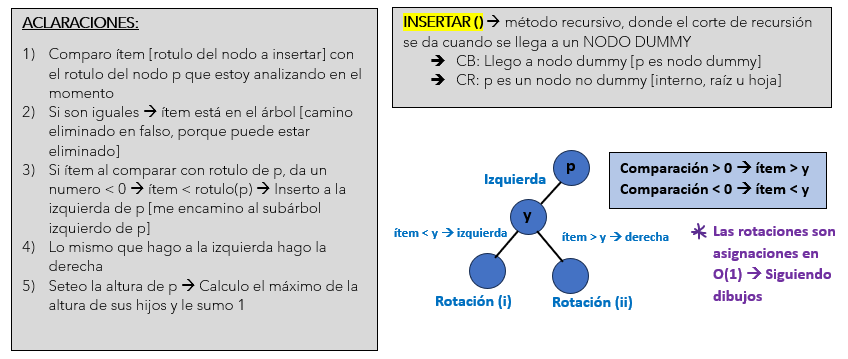




**IMPLEMENTACION JAVA DE LA INSERCION:**







**COMPLEJIDAD TEMPORAL DE LA INSERCION:**

* Las rotaciones se hacen en los nodos del camino desde la raiz hasta la hoja donde se inserto la nueva clave
* Como las rotaciones se implementan con asignaciones de referencias (posiciones), cada rotación se hace en tiempo constante
* La cantidad de rotaciones es del orden de la altura del árbol
* La altura es proporcional al logaritmo de la cantidad de nodos del árbol
* El tiempo de insertar es del orden del logaritmo de la cantidad de nodos del árbol

**//Por lo que siempre la busqueda, insercion y eliminacion posee tiempo O(log(n)), ya que esta balanceado**

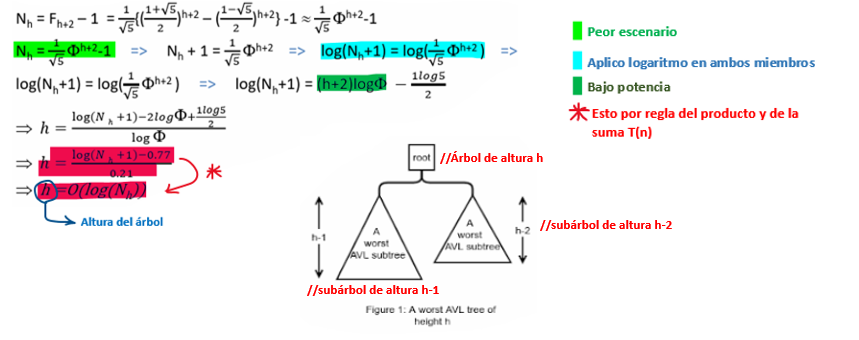
**EXPLICACION DE PORQUE LA ALTURA DEL AVL ES O(log(n)):**

* Como la diferencia de altura de los hijos de un nodo de un AVL es a lo sumo 1, se cumple que la cantidad de nodos de un arbol de altura “h” en el peor arbol desbalanceado se define recursivamente como:

Esta recurrencia se parece a Fibonacci:

* Sea 🡪 Conocido como la **RAZON DORADA o NUMERO AUREO**

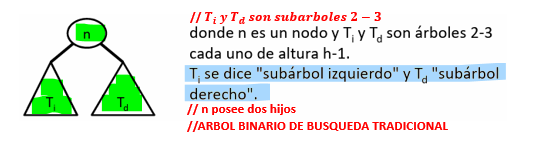
Se puede demostrar que:



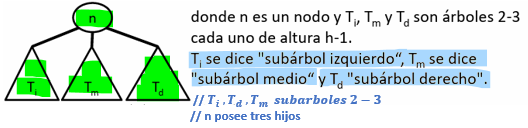
**ARBOLES 2-3:**

* Un “ARBOL 2-3” es un arbol tal que cada nodo interno (no hoja) tiene dos o tres hijos, y **todas las hojas estan al mismo nivel**
* DEFINICION RECURSIVA ARBOL 2-3: T es un arbol 2-3 de altura h si:

1. T es vacio 🡪 De altura 0
2. T es de la forma:

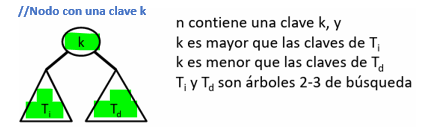


1. T es de la forma:

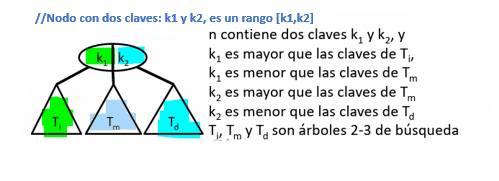


* PROPIEDAD: Si un árbol 2-3 no contiene ningún nodo con 3 hijos entonces su forma corresponde a un ARBOL BINARIO LLENO
* DEFINICION RECURSIVA DE ARBOL 2-3 DE BUSQUEDA: T es un arbol 2-3 de busqueda, si T es un arbol 2-3 tal que :

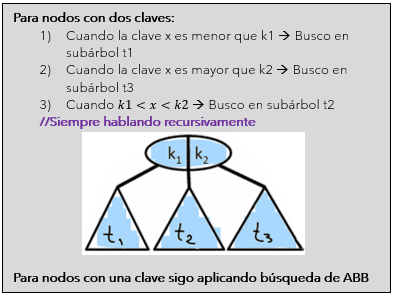
1. T es vacio
2. T es de la forma:

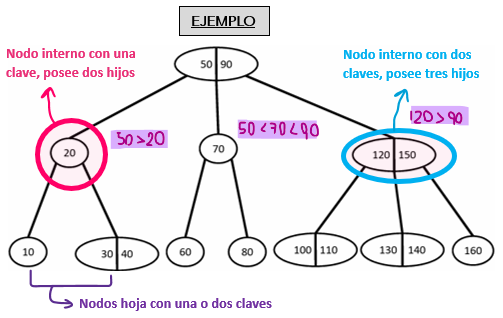


1. T es de la forma:

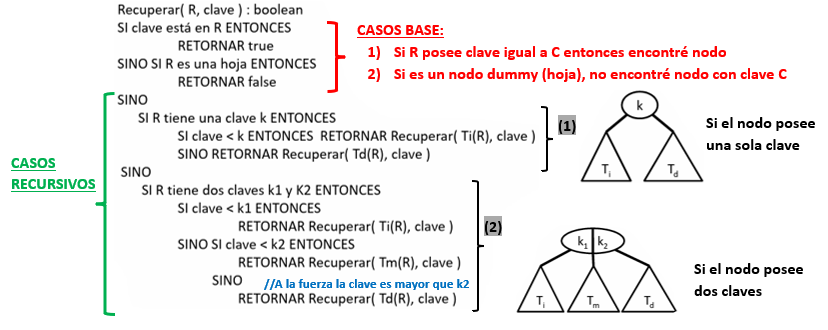


* REGLAS PARA UBICAR CLAVES EN LOS NODOS DE UN ARBOL 2-3 DE BUSQUEDA:

1. Si n tiene dos hijos, entonce contiene una clave
2. Si n tiene tres hijos, entonces contiene dos claves
3. Si n es una hoja, entonces contiene una o dos claves

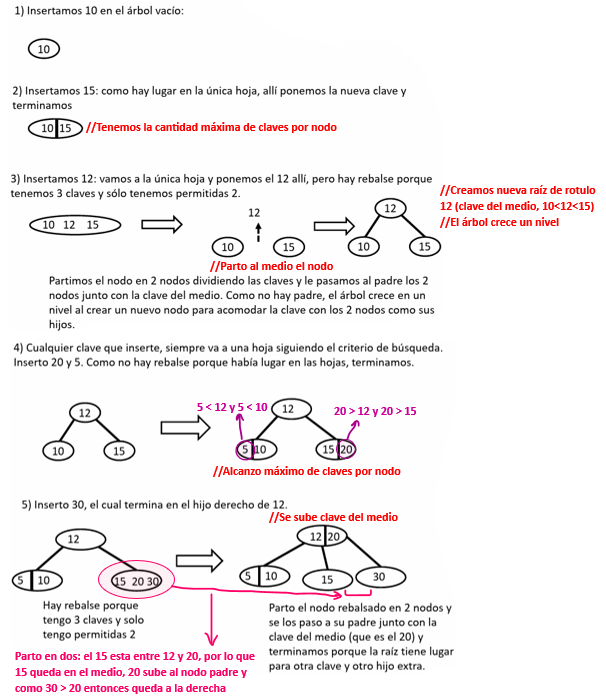


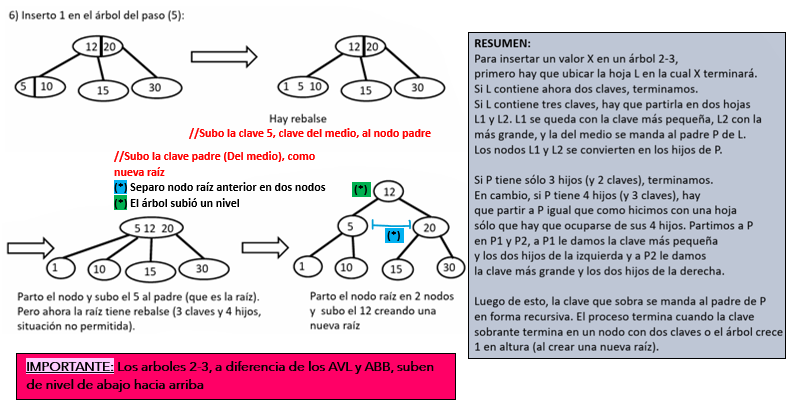
* BUSQUEDA EN UN ARBOL 2-3 [ALGORITMO]:

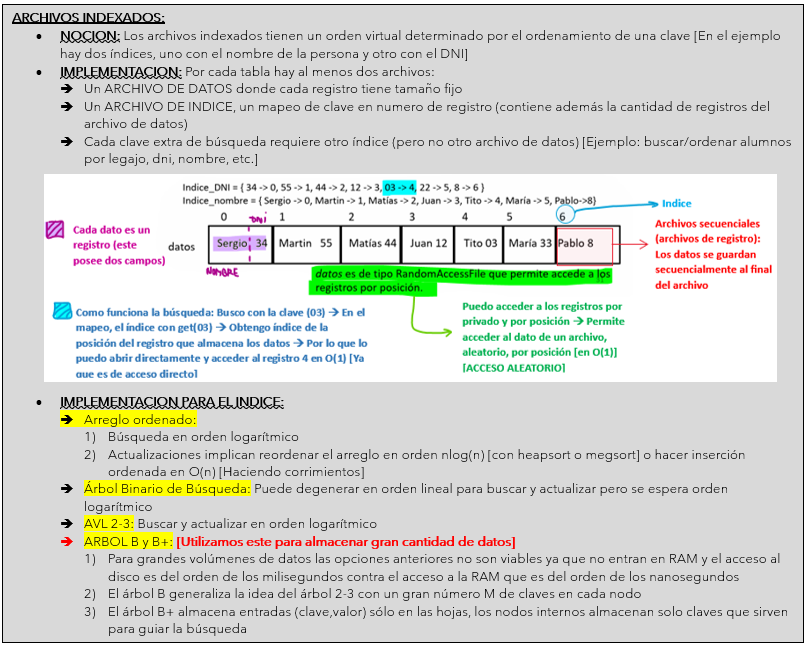


* INSERCION EN UN ARBOL 2-3:
* Igual que en el ABB, siempre se inserta en una hoja siguiendo el camino de busqueda desde la raiz del arbol hasta la hoja correspondiente (esa hoja ya va a tener por lo menos un hijo)
* Si la hoja ahora tiene 2 claves, terminamos
* Si la hoja ahora tiene 3 claves, se produce un rebalse (overflow) y se debe partir el nodo en dos nodos, la clave del medio sube al padre, quien queda a cargo de administrar ese hijo que se duplicon y ver donde ubicar esa clave **(1)**
* Si el padre tiene 1 clave y dos hijos, no hay problema porque ahora tendra 2 claves y 3 hijos
* Si el padre ya tenia 2 claves y 3 hijos, ahora pasaria a tener 3 claves y 4 hijos, lo cual no puede ocurrir 🡪 Entonces el proceso **(1)** se repite
* El proceso se termina cuando terminamos en un nodo intermedio (Con 2 claves o 3 hijos) o llegamos a crear una nueva raiz y el arbol crece un nivel

**//Como este proceso tiene [Tiempo de insercion]**

 **EJEMPLO DE INSERCION DE UN ARBOL 2-3:**

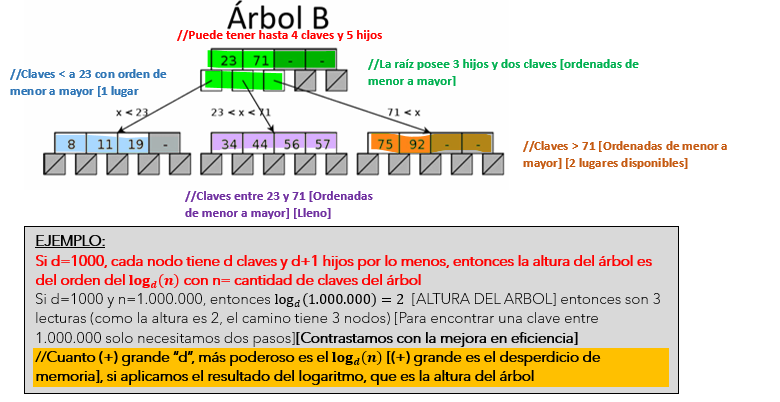




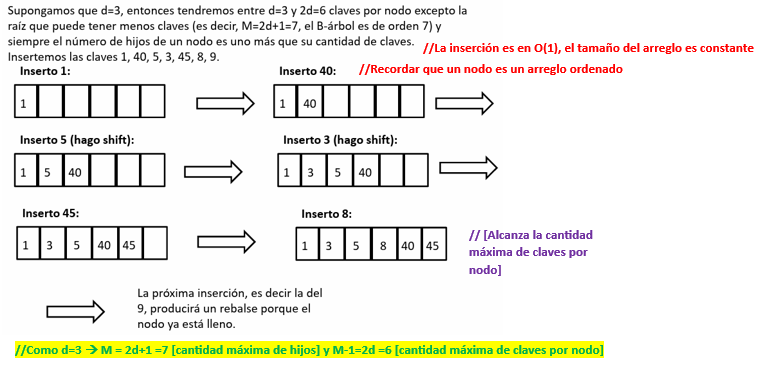
**ARBOL B (o M-ARIOS DE BUSQUEDA):**

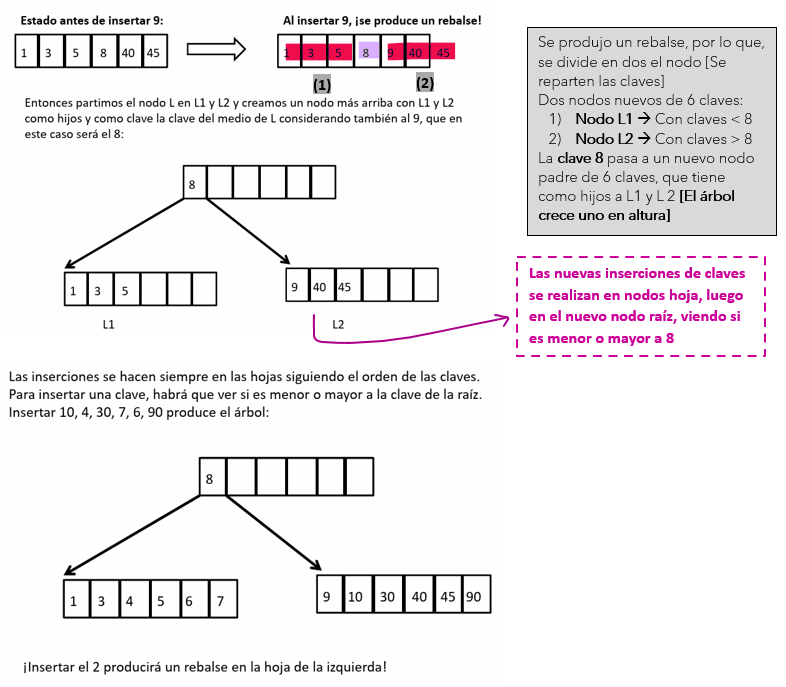
* Un árbol-B (B-tree) es un árbol balanceado (o autobalanceante) que mantiene los datos ordenados y permite realizar búsquedas, inserciones y eliminaciones en tiempo logarítmico
* Se puede ver también como una generalización del árbol 2-3 porque sus nodos pueden tener más de tres hijos
* A diferencia de los árboles 2-3, el árbol-B está optimizado para sistemas que leen y escriben grandes bloques de datos
* Los árboles-B son un buen ejemplo de una estructura de datos para memoria externa (en disco) y se usan comúnmente en bases de datos y sistemas de archivos
* Los árboles B se optimizan para manejar grandes volúmenes de datos
* Los árboles B se almacenan en disco y el tamaño del nodo coincide con un múltiplo entero del tamaño del sector del disco
* Se utilizan para implementar índices en bases de datos

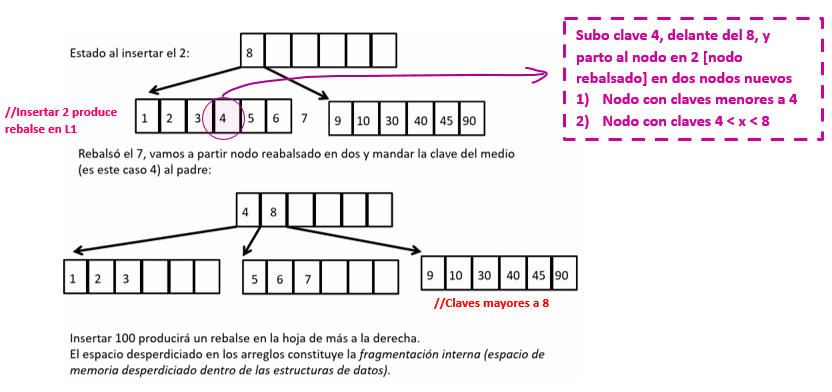
**Cualquier nodo excepto la raíz:**

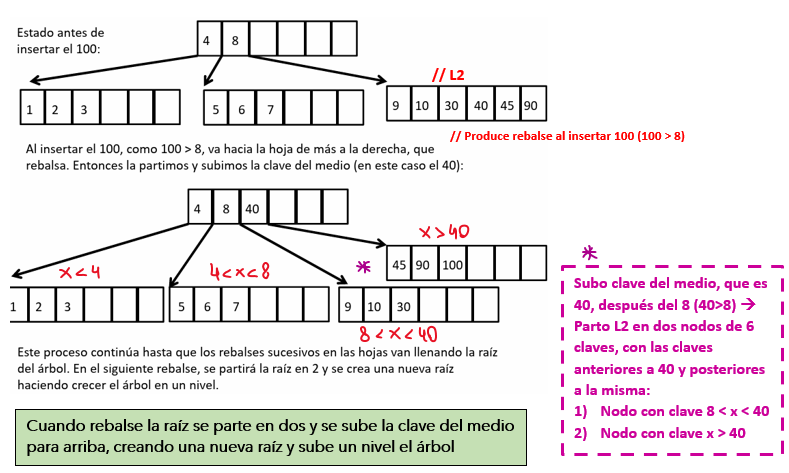
* **Menor caso: posee de claves y d+1 hijos**
* **Peor caso: 2d claves y 2d+1 hijos**
* El grado de ramificación d del árbol es un entero, indicando que un nodo tiene entre d y 2d claves y entre d+1 y 2d+1 hijos (excepto la raíz que puede tener menos de d claves y tiene por lo menos 1 clave y 2 hijos) [Cuando d=1 tenemos un árbol 2-3]
* Dependiendo de la formalización, al número 2d+1 se lo llama M (Tenemos M-1 claves y M hijos a lo sumo por nodo; en un árbol 2-3, M =3)
* **El tiempo de busqueda, insercion y eliminacion es el tiempo logaritmico puesto que es del orden de la altura del arbol**
* Generalmente hay espacio desperdiciado en memoria [FRAGMENTACION INTERNA]
* INSERCCION EN ARBOLES B:
* Comenzamos con un nodo vacío (el cual funciona como un arreglo ordenado) **[Cada nodo funciona como un arreglo ordenado]**
* Las claves se insertan (en forma lineal y ordenada) en el nodo hasta tener a lo sumo 2d = m-1 claves
* El nodo se parte en dos (cada uno con d claves): la clave del medio va al nodo de arriba (incrementando la altura del árbol cuando tal nodo de arriba no existiera) y los nodos que quedan se quedan cada uno con la mitad de las claves menores a d y mayores a d respectivamente

**EJEMPLO DE INSERCION EN ARBOL B:**

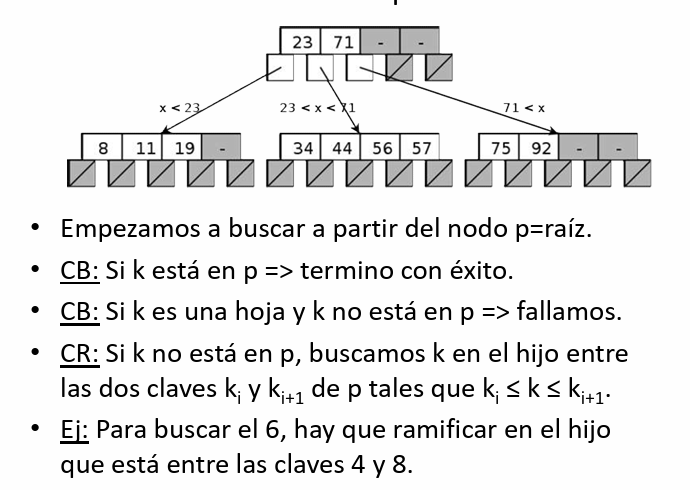








* BUSQUEDA DE K EN ARBOLES B (ALGORITMO DE BUSQUEDA DE UNA CLAVE):

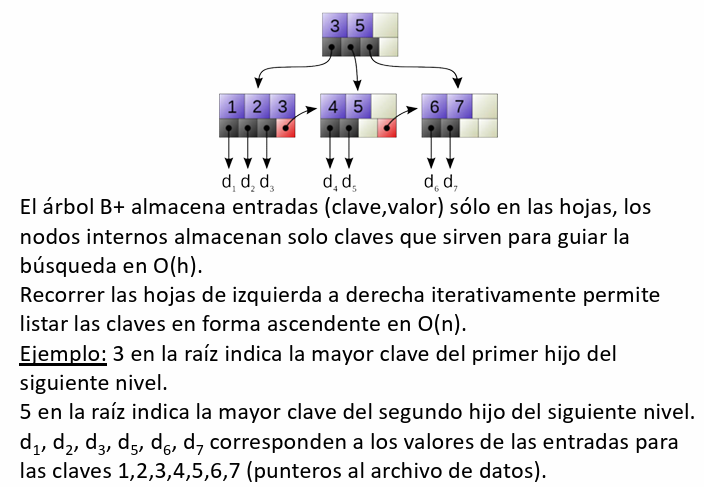


**Hago búsqueda binaria ya que los nodos poseen claves ordenadas**

**Realizo búsquedas binarias en cada nodo del arreglo de claves**

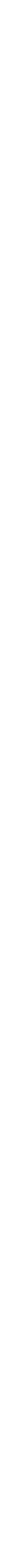
**Se realiza en O(1), ya que el tamaño del arreglo de claves de un nodo es constante**

**ARBOL B+ (VARIANTE DE ARBOL B):**





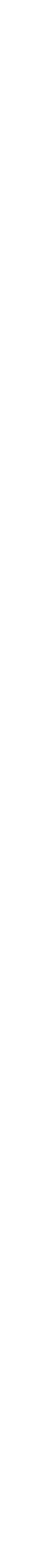
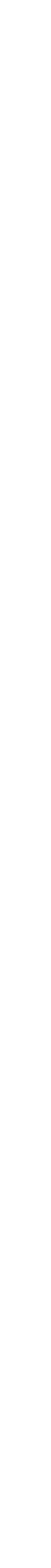
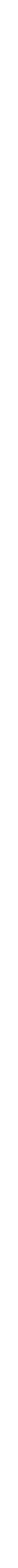
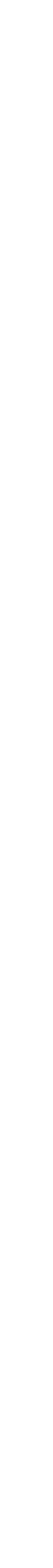
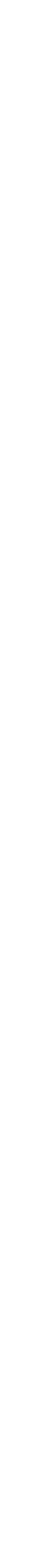
**Referencia a archivos de datos**



**Las claves indican los sectores de búsqueda**



**Enlazadas como en una lista, recorrer claves de (-) a (+) es hacer un bucle**



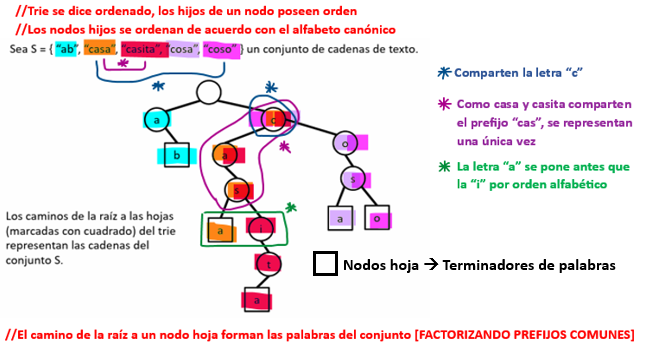
**PROCESAMIENTO DE TEXTOS:**

**TRIES:**

* Un trie es una estructura de datos que se usa para implementar conjuntos de strings, y mapeos y diccionarios de string en un tipo E
* Un trie es un árbol que factoriza prefijos comunes entre las cadenas almacenadas en el mismo
* Los caminos de la raíz a las hojas representan las palabras del conjunto o las claves del mapeo
* DEFINCION:
* Sea S un conjunto de s strings sobre un alfabeto
* Un trie para S es un árbol ordenado T tal que:

1. Cada nodo de T, excepto la raíz, está etiquetado con un carácter de
2. El orden los hijos de un nodo interno de T está determinado por el orden canónico de
3. T tiene s nodos externos, cada uno asociado con un string de S, tal que la concatenación de los rótulos de los nodos del camino de la raíz a una hoja v produce el string de S asociado a v

**EJEMPLO:**

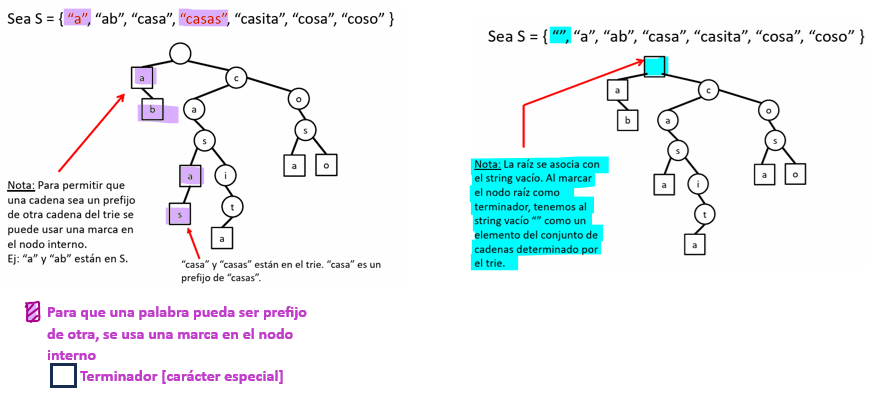


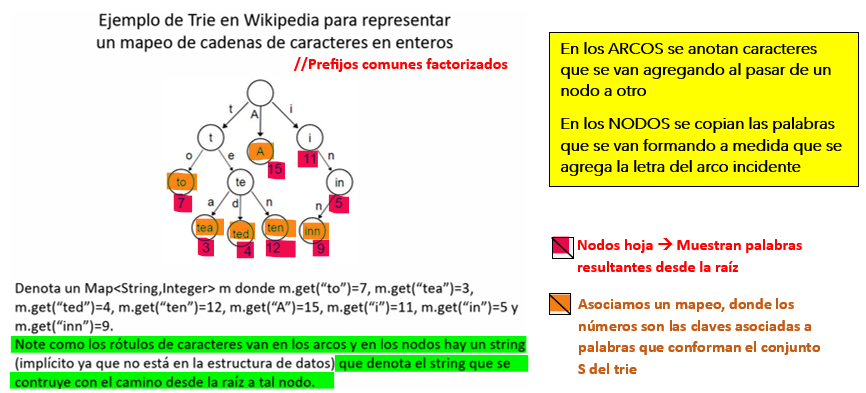
* PROPIEDADES DE TRIES: Un trie T almacenado:
* Una coleccion S de s strings de longitude total n (es decir, la suma de las longitudes de todos los strings del conjunto es n)
* Sobre un alfabeto de tamaño d cumple:

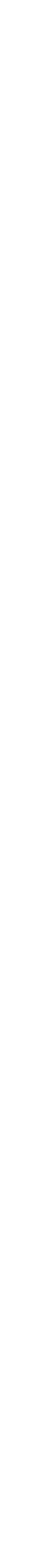
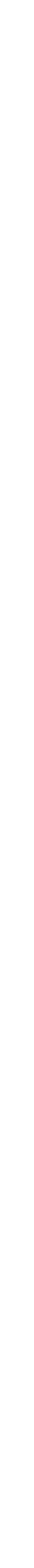
1. Cada nodo interno de T tiene a lo sumo d hijos
2. Ttieen s nodos externos
3. La altura de T es igual a la longitude del string mas largo de S
4. El numero de nodos de T es O(n) [Peor caso: Cuando no hay prefijos communes y el arbol tiene tantos nodos como caracteres totals tendria el conjunto [sino tiene menos]]

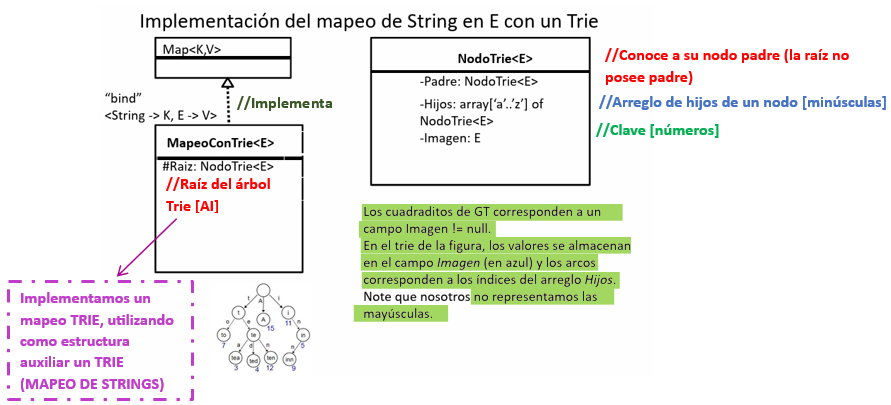
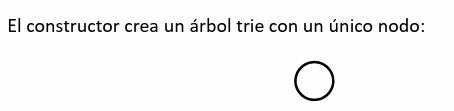
* Para definer un TRIE no se permitira que una palabra del trie sea prefijo de otra palabra del trie [segun GT] 🡪 Para solucionar esto agregaremos un caracter especial terminador

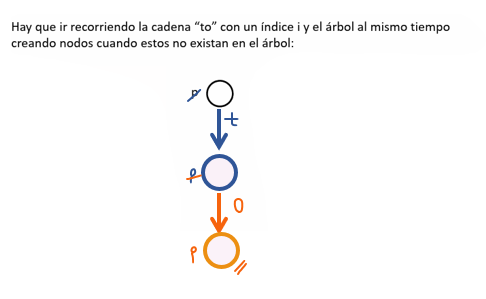
**EJEMPLOS:**

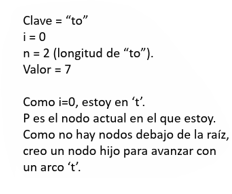
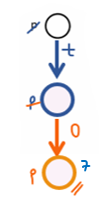


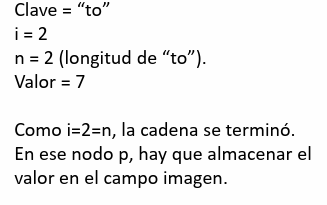
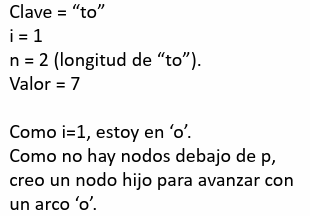




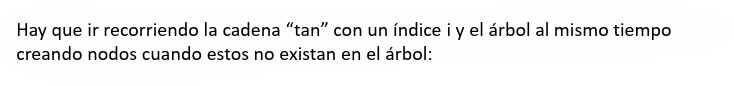
* IMPLEMENTACION (UML) Y EXPLICACION GRAFICA DE ALGUNOS METODOS:
* EXPLICACION GRAFICA DEL FUNCIONAMIENTO DEL METODO PUT():

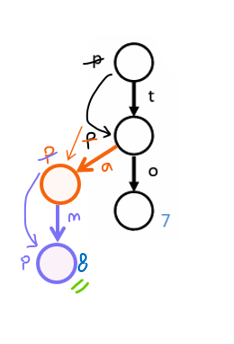
1. **PUT (“TO”,7) EN EL ARBOL:**

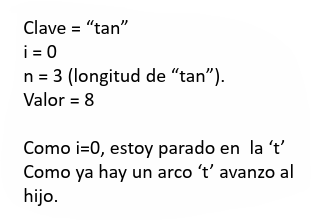
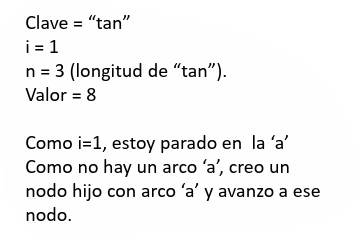
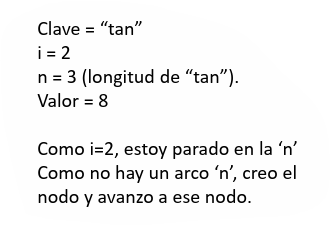
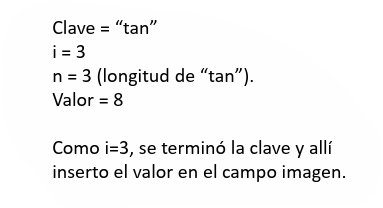




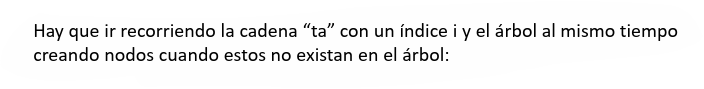
1. **PUT(“TAN”,8) EN EL ARBOL:**

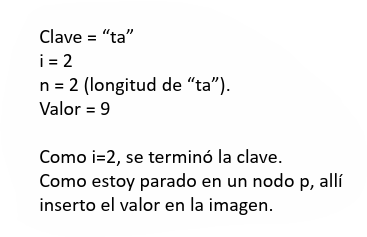
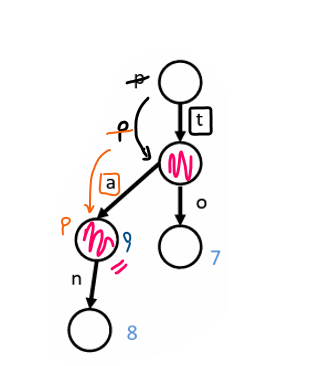


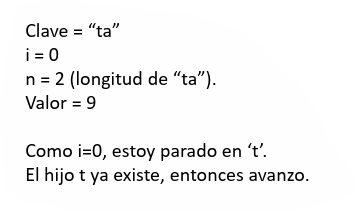
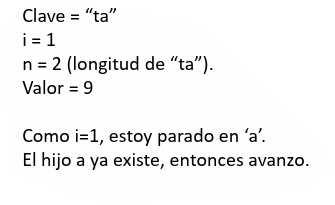






1. **PUT(“TA”,9) EN EL ARBOL:**

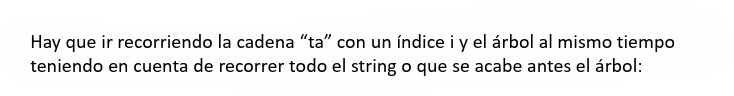


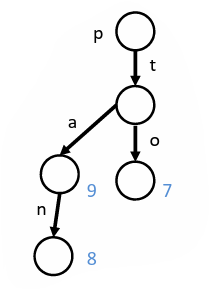
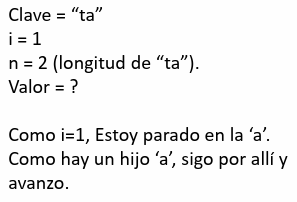
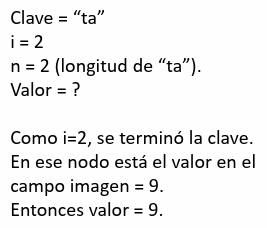


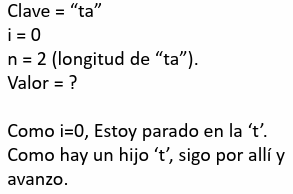
**//La clave me indica la terminación del camino a recorrer para obtener el string deseado**

* EXPLICACION GRAFICA DEL FUNCIONAMIENTO DEL METODO GET():

1. **GET(“TA”) DEL ALBOL:**







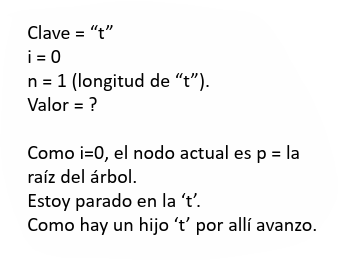
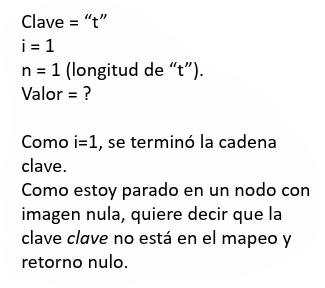
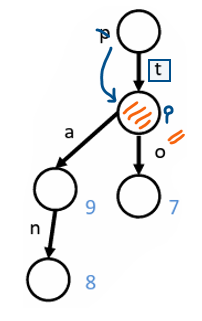


**//Consumí toda la clave, termina recorrido, retorna 9**



1. **GET(“T”) DEL ARBOL:**

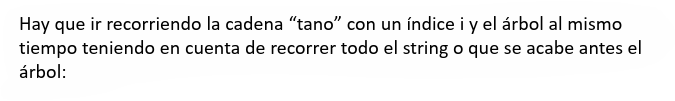


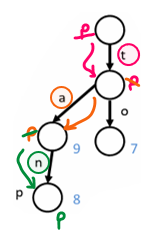


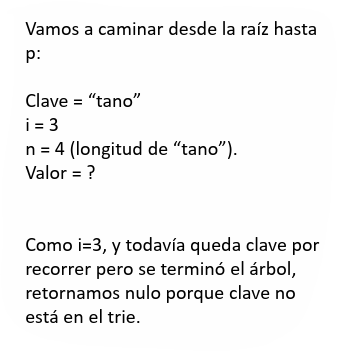
**//Cuando quiero retornar el valor de un string que no pertenece a el conjunto S**

**//Voy a caer en un nodo seteado en NULL [sin clave] [EL ARBOL NO SE TERMINO]**

1. **GET(“TANO”) DEL ARBOL:**



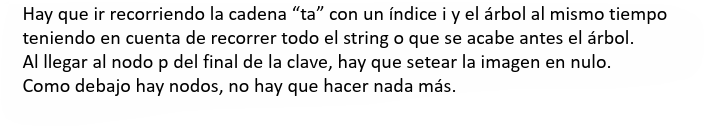




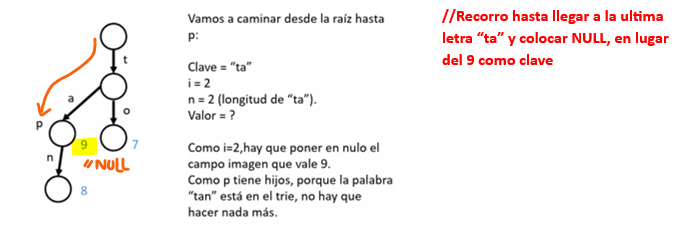
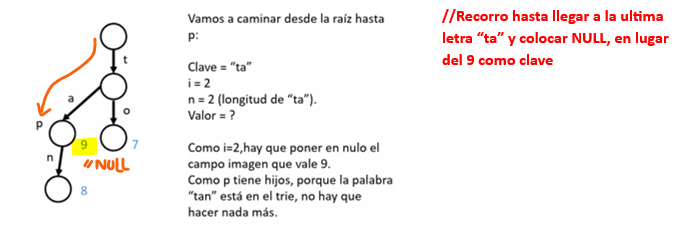
**//Si quiero retornar el valor de una palabra y resulta que se termina árbol [caigo en nodo hoja], retorno NULL como valor; para indicar que la palabra no existe al conjunto S**

* EXPLICACION GRAFICA DEL FUNCIONAMIENTO DEL METODO REMOVE():

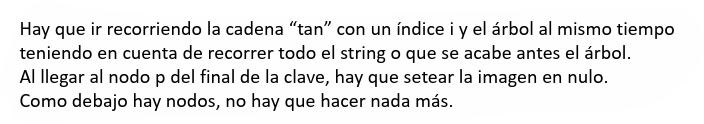
1. **REMOVE(“TA”) DEL ARBOL:**

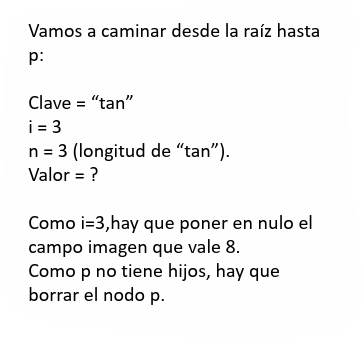
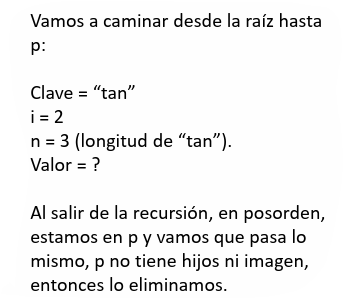
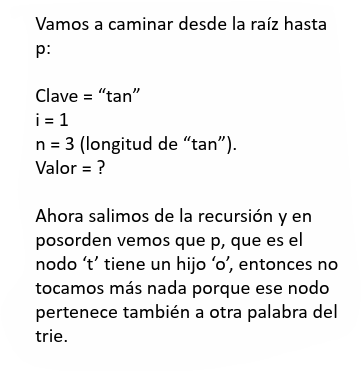
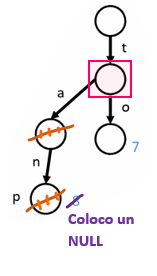


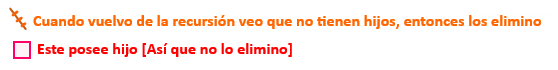


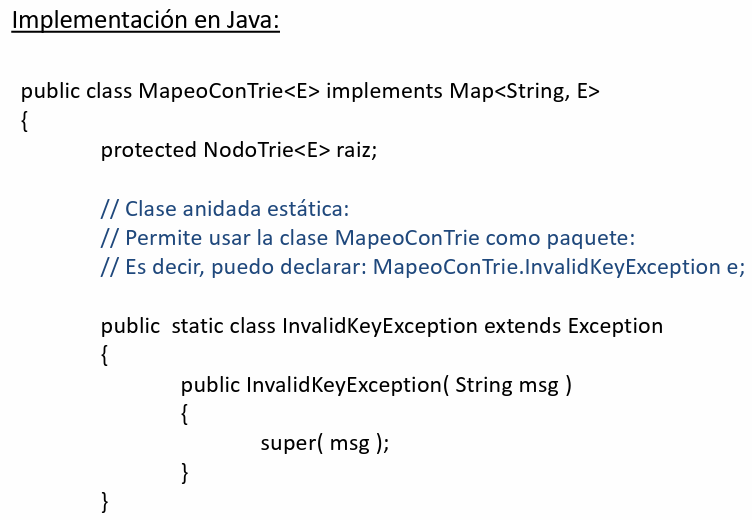
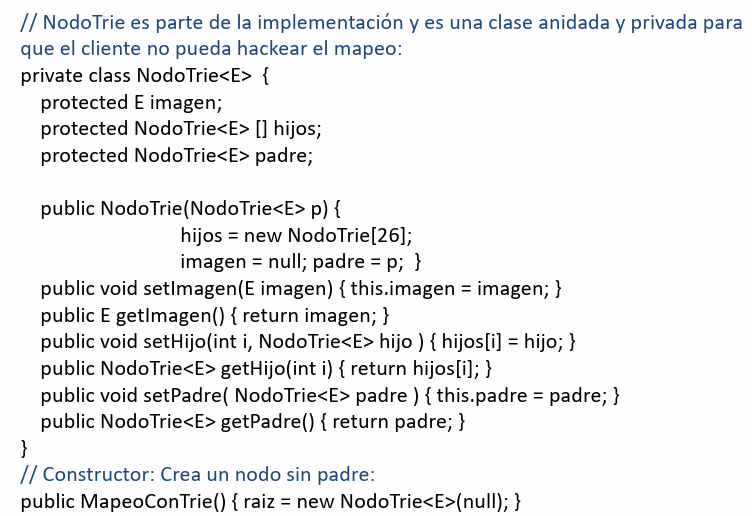


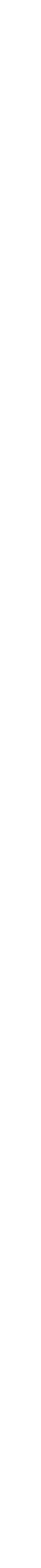
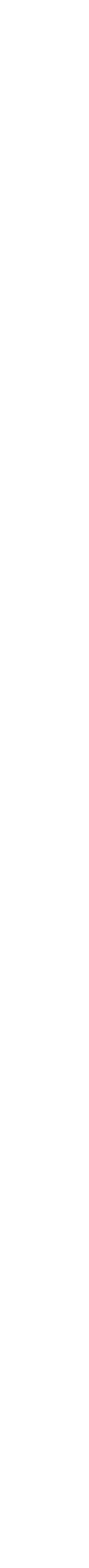
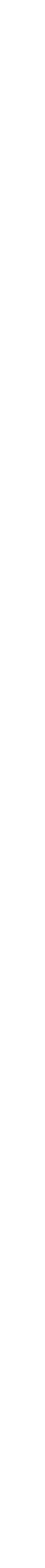
1. **REMOVE(“TAN”) DEL ARBOL:**

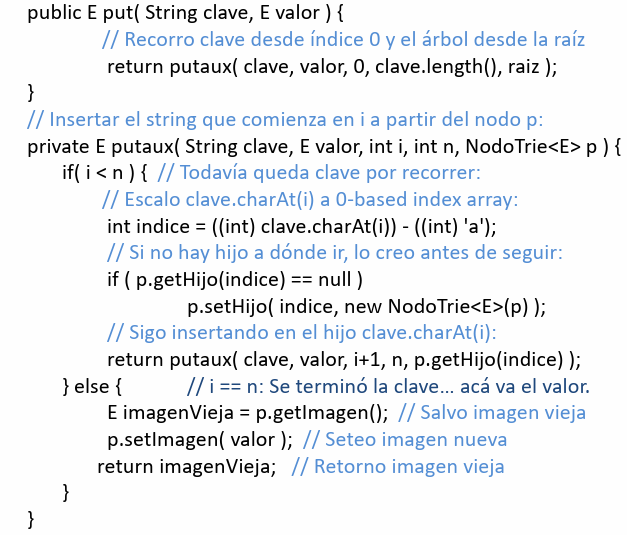
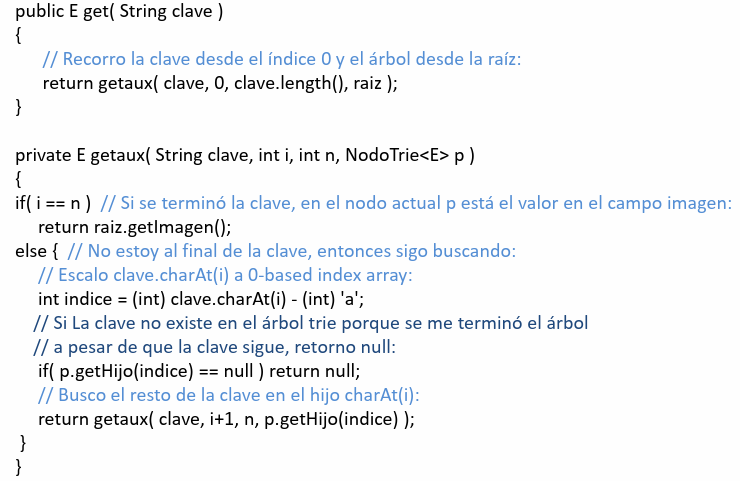


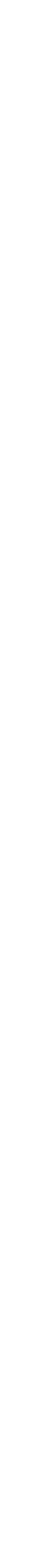
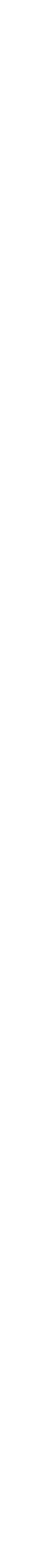


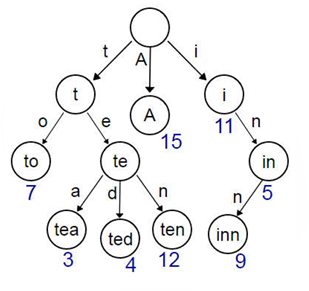
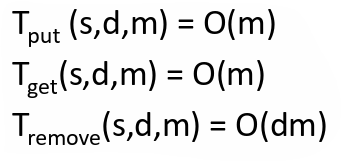




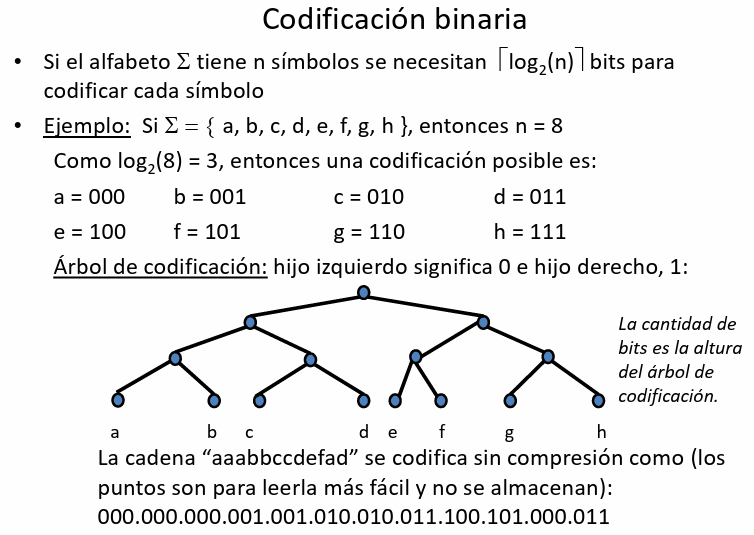






* COMPLEJIDAD TEMPORAL (METODOS):
* Sea un conjunto S implementado con un trie T sobre un alfabeto
* Sea s=cardinal de S, d=cardinal de (Tamaño del alfabeto), m=largo de un string a procesar (se puede acotar por la altura del arbol)
* COMPRESION DE DATOS:
* La compresión de datos consiste en la reducción del volumen de información tratable (procesar, transmitir o grabar)
* En principio, con la compresión se pretende transportar la misma información, pero empleando menor cantidad de espacio
* **TIPOS DE COMPRESION:**

1. **COMPRESION SIN PERDIDA DE INFORMACION:**

* Los datos antes y despues de comprimirlos son exactos en la compresion sin perdida
* En el caso de la compresion sin perdida una mayor compresion solo implica mas tiempo de proceso
* Se utiliza principalmente en la compresion de texto
* 

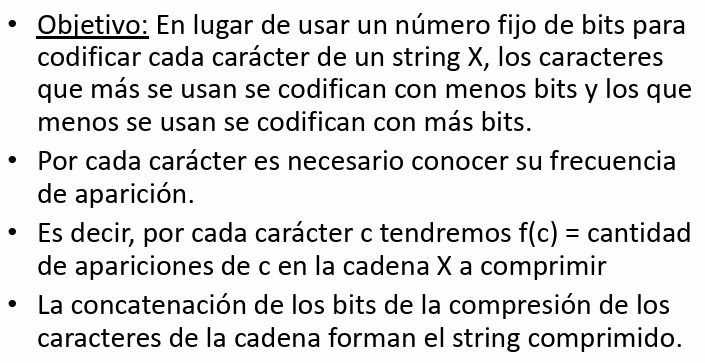
**EJEMPLO (Problema o Enunciado):** Dado un string X definido sobre un alfabeto , deseamos codificar X en forma eficiente [en forma binaria, es decir, obtener un string Y formado solo por 0 y 1]

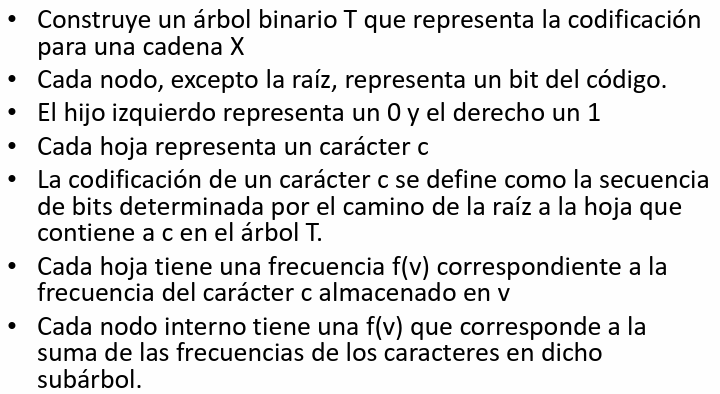
**//Voy tres veces a la izquierda**



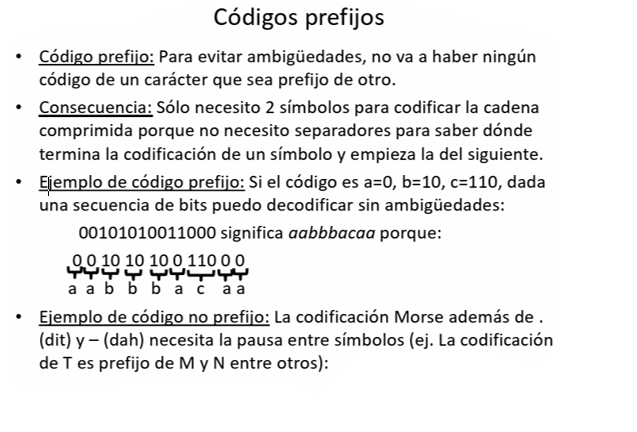
1. **COMPRESION CON PERDIDA DE INFORMACION:**

* Puede eliminar datos para reducer aun mas el tamaño, con lo que se suele reducir la Calidad
* Hay que tener en cuenta que una vez realizada la compresion, no se puede obtener la señal original, aunque si una aproximacion cuya semejanza con la original dependera del tipo de compresion
* Se utiliza principalmente en la compresion de imagenes, videos y sonidos [EJEMPLO: Formatos jpg, mpeg, mp3]

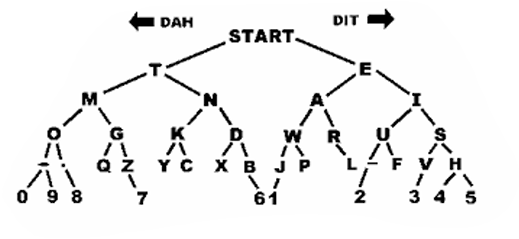
**CODIFICACION DE HUFFMAN:**

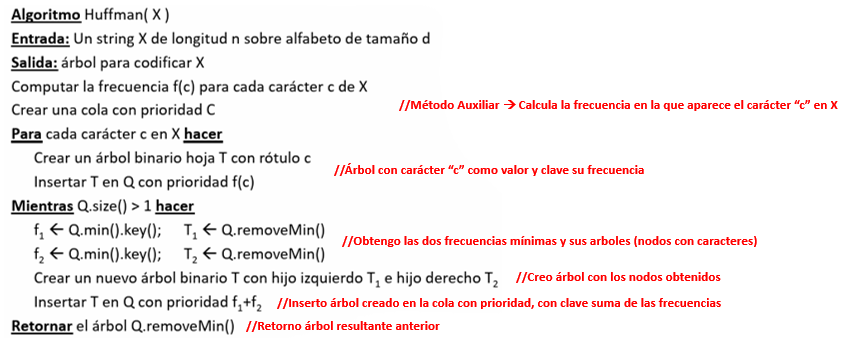
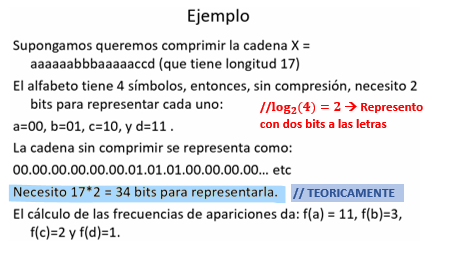
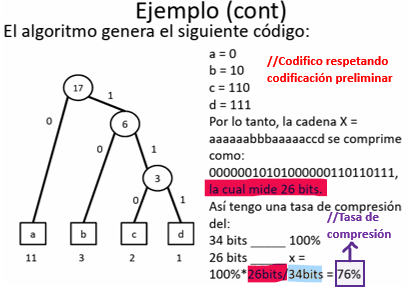


**Idea de codificación de Huffman**









**ALGORITMO DE HUFFMAN**

