udp Escuela de Informática y Telecomunicaciones
FACULTAD DE INGENIERÍA Y CIENCIAS

Tutoría para Examen de Grado

Sesión 1: Estructuras de Datos y algoritmos básicos sobre EDD

Docente: Mauricio Hidalgo Barrientos

Temario de la sesión

- Recordar los conceptos de Orden de complejidad
- Algoritmos (básicos) de Ordenamiento
- Recordar las EDD básicas: Listas enlazadas, Pilas y Colas
- Recordar otras estructuras de datos: Árbol binario y algoritmos de DFS, otras EDD del tipo árbol y generalidades sobre Hashing

Orden de Complejidad

Orden de Complejidad

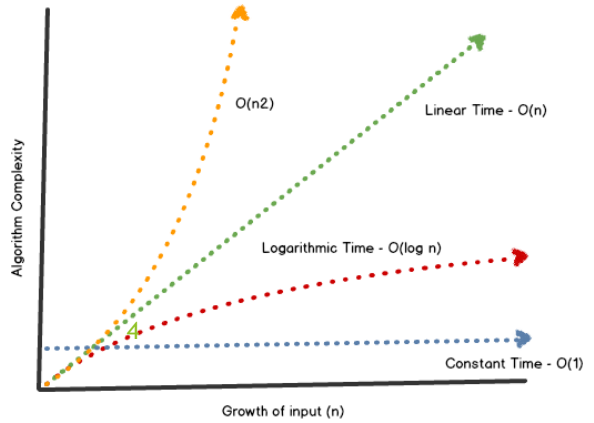
¿Qué es el Orden de Complejidad?

"En las ciencias de la computación, el orden de complejidad de un algoritmo cuantifica la cantidad de tiempo y/o espacio que requiere un algoritmo para ejecutar una función en virtud del largo de su input"

Notación Asintótica

La notación asintótica es una forma de expresar "el orden de complejidad" de un Algoritmo basados en su tasa de crecimiento. Es decir, despreciando los coeficientes constantes y menos significativos. En este ámbito, recordaremos 3 tipos de notación asintótica:

- *Notación Ω*
- *Notación Θ*
- *Notación O*



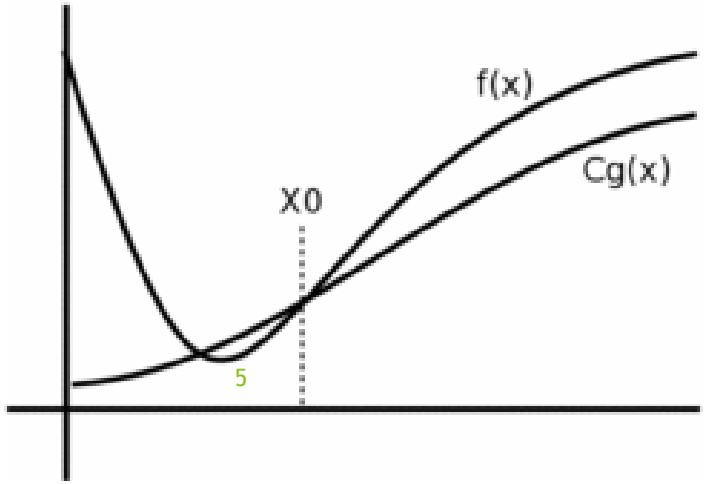
Orden de Complejidad: Notación Ω

La notación Ω, conocida como cota inferior asintótica es una función que:

- Sirve de “tope” inferior de otra función cuando el argumento tiende a infinito.
- Usualmente se utiliza la notación Ω(g(x)) para referirse a las funciones acotadas inferiormente por g(x).

Formalmente se define:

Notación Ω: Dadas $f: \mathbb{R} \to \mathbb{R}^+$ y $g: \mathbb{R} \to \mathbb{R}^+$, se dice que f es omega de g, denotado por $f \sim \Omega(g)$, si existen constantes c y n_0 tales que $f(n) \geq c \cdot g(n) \forall\, n \geq n_0$. En otras palabras, f está acotada inferiormente por $c \cdot g(n)$ para todo valor $n \geq n_0$.



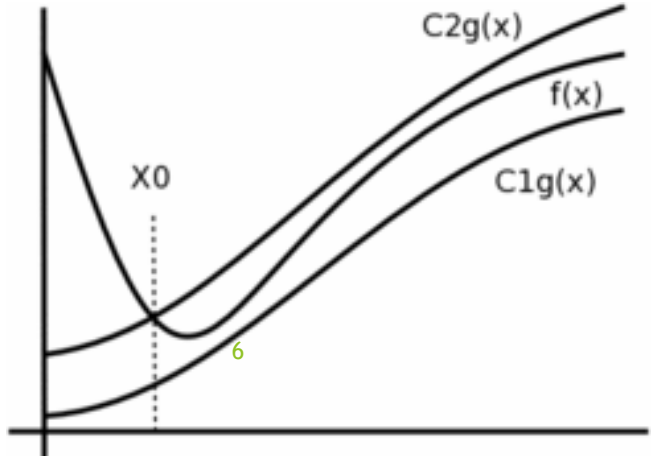
Orden de Complejidad: Notación Θ

La notación Θ, conocida como cota ajustada asintótica es una función que:

- Sirve de “tope” superior e inferior de otra función cuando el argumento tiende a infinito.
- Usualmente se utiliza la notación Θ(g(x)) para referirse a las funciones acotadas por la función g(x).

Formalmente se define:

Notación Θ: Dadas $f: \mathbb{R} \to \mathbb{R}^+$ y $g: \mathbb{R} \to \mathbb{R}^+$, si $f \sim O(g)$ y $f \sim \Omega(g)$ a la vez, se dice que $f \sim \Theta(g)$. Expresado de otro modo, $f \sim \Theta(g)$ si existen constantes c_1, c_2 y n_0 tales que $c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)\ \forall\ n \geq n_0$. Es decir, f está acotada inferiormente por $c_2 \cdot g(n)$ y superiormente por $c_1 \cdot g(n)$ para todo valor $n \geq n_0$.



Orden de Complejidad: Notación O

La notación O, conocida como cota superior asintótica, es una función que:

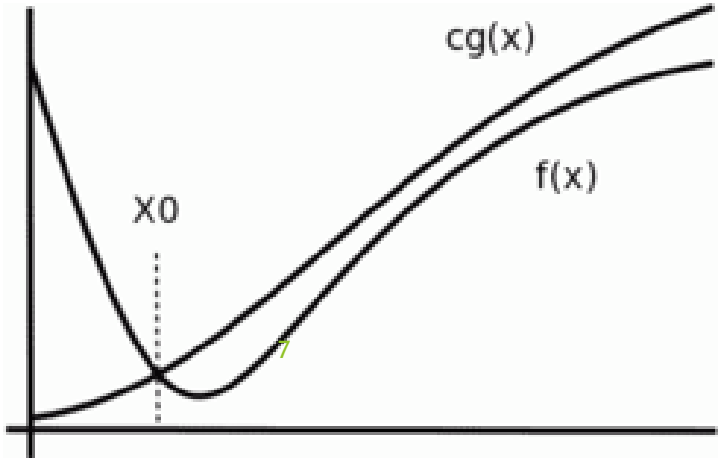
- Sirve de “tope” superior de otra función cuando el argumento tiende a infinito.
- Usualmente se utiliza la notación O(g(x)) para referirse a las funciones acotadas superiormente por g(x).

Formalmente se define:

Dadas $f: \mathbb{R} \to \mathbb{R}^+$ y $g: \mathbb{R} \to \mathbb{R}^+$, se dice que f es del orden de g, denotado por $f \sim O(g)$, si existen constantes c y n_0 tales que $f(n) \leq c \cdot g(n) \forall n \geq n_0$. En otras palabras, f está acotada inferiormente por $c \cdot g(n)$ para todo valor $n \geq n_0$.

IMPORTANTE

Durante esta sesión nos vamos a referir principalmente a esta notación



Propiedades de la Notación O

1. $O(f) + O(g) = O(f + g)$	Regla de la suma
2. $O(k \cdot f) = k \cdot O(f) = O(f), k\ cte.$	Constantes multiplicativas
3. $O(f) \cdot O(g) = O(f \cdot g)$	Regla de la multiplicación
4. $f \geq g \Rightarrow O(f + g) = O(f)$	Cotas
5. $f \geq g \Rightarrow (h \sim O(g)\ entonces\ h \sim O(f))$	Transitividad

IMPORTANTE

Las propiedades de esta notación son igualmente válidas en las otras notaciones (en la mayoría de los casos)

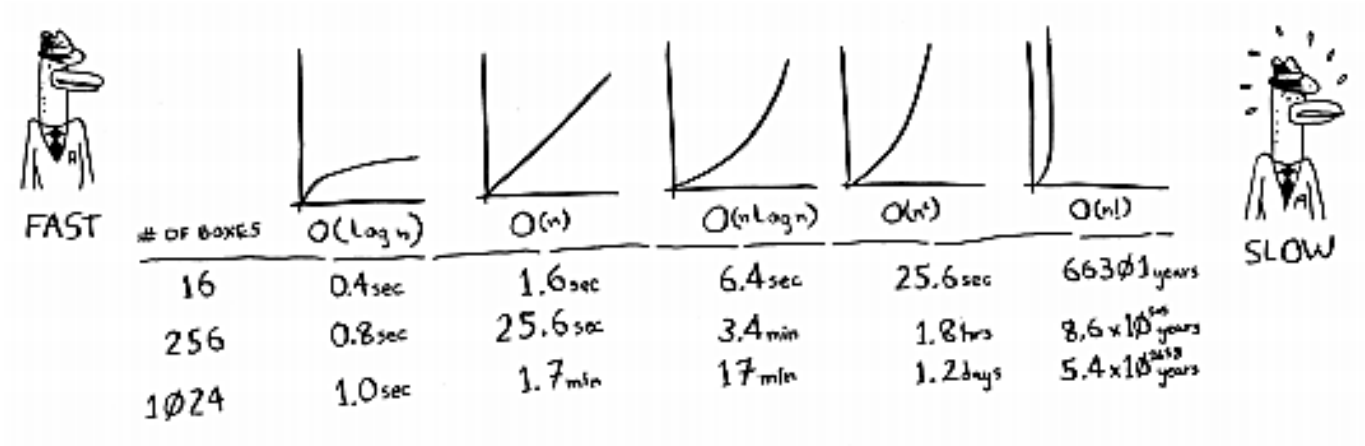


Tabla de Orden de Complejidad

Los órdenes más utilizados en análisis de algoritmos, en orden creciente, son los siguientes (donde “c” representa una constante y “n” el tamaño de la entrada):

notación	nombre
$O(1)$	orden constante
$O(\log \log n)$	orden sublogarítmica
$O(\log n)$	orden logarítmica
$O(n \cdot \log n)$	orden lineal logarítmica
$O(\sqrt{n})$	orden sublineal
$O(n)$	orden lineal o de primer orden
$O(n^2)$	orden cuadrática o de segundo orden
$O(n^3)$, ...	orden cúbica o de tercer orden, ...
$O(n^c)$	orden potencial fija
$O(c^n)$, n > 1	orden exponencial
$O(n!)$	orden factorial
$O(n^n)$	orden potencial exponencial

Fuente: *https://es.wikipedia.org/wiki/Cota_superior_asint%C3%B3tica*

Tiempo constante – O(c)

Un algoritmo es considerado de Tiempo Constante (escrito como tiempo O(c)) se da cuando el tamaño del input no cambia lo que se demora en realizar el algoritmo. Esto es lo ideal.

Ejemplos: Operaciones simples y Comparaciones

```
int index = 5;
int item = list[index];
if (condition true) then
   perform some operation that runs in constant time
else
   perform some other operation that runs in constant time
for i = 1 to 100
   for j = 1 to 200
      perform some operation that runs in constant time
```

Tiempo Logarítmico - O(log n)

Los algoritmos que se ejecutan en un tiempo logarítmico son aquellos donde cada nueva operación tomará la mitad de tiempo con respecto a la anterior. Esto se considera óptimo.

Ejemplo: Búsqueda binaria

```
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1
    return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

Tiempo Lineal - O(n)

Un algoritmo se llama lineal cuando el tiempo que se demora en ejecutar es dependiente del largo del input. Esto se considera normal.

Ejemplo: Copiar y Buscar (en lista), Operaciones que necesiten analizar todos los elementos de un conjunto.

```
int Maximo(int v[], int Tam)
{
   int i, Max;

   Max = v[0];
   for (i=0; i<Tam; i++)
     if (v[i] > Max)
        Max = v[i];

   return Max;
}
```

0	1	2	3	4
7	12	-1	24	11

i	*Max*
	7
0	7
1	12
2	12
3	24
4	24

Tiempo Polinomial - $O(n^k)$

Se le llama Tiempo Polinomial cuando un algoritmo es limitado superiormente por una expresión polinomial según el tamaño del input. Esto ocurre con mucha frecuencia.

Ejemplo: Algunos algoritmos de ordenamiento, Algoritmos de Factorización

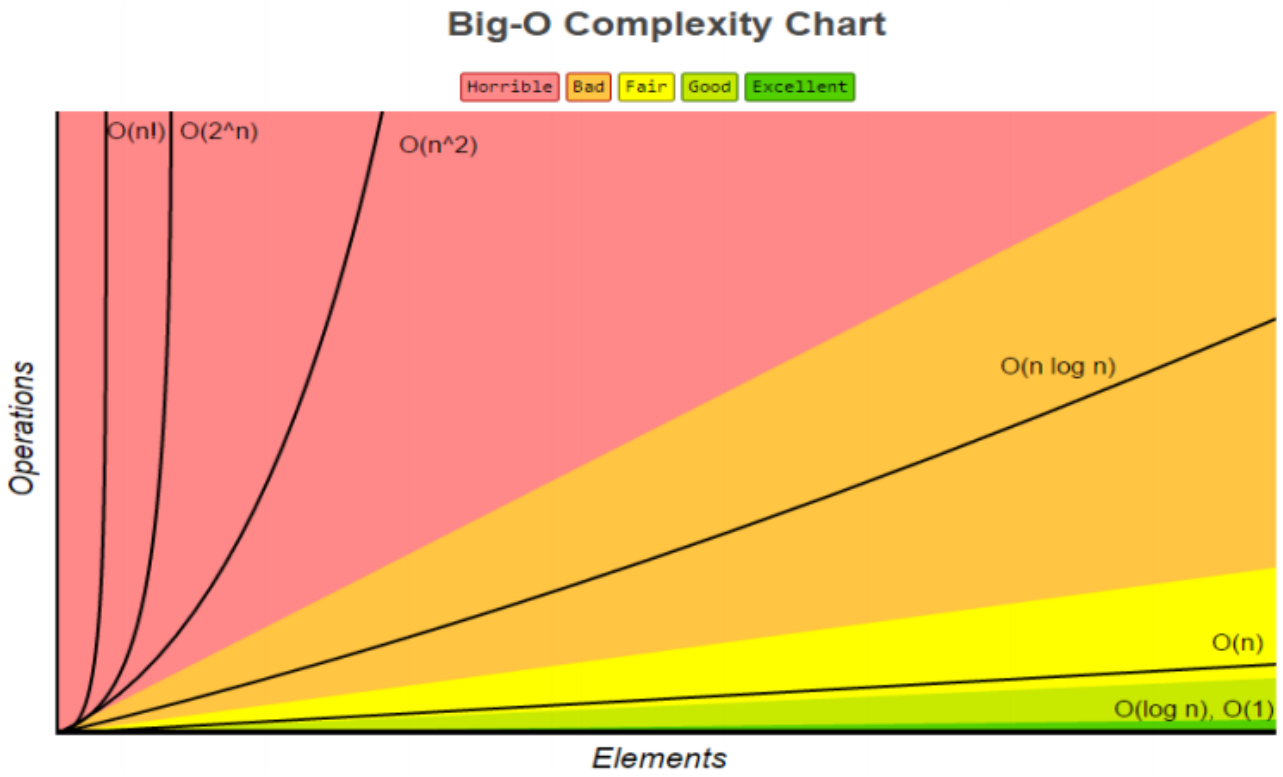
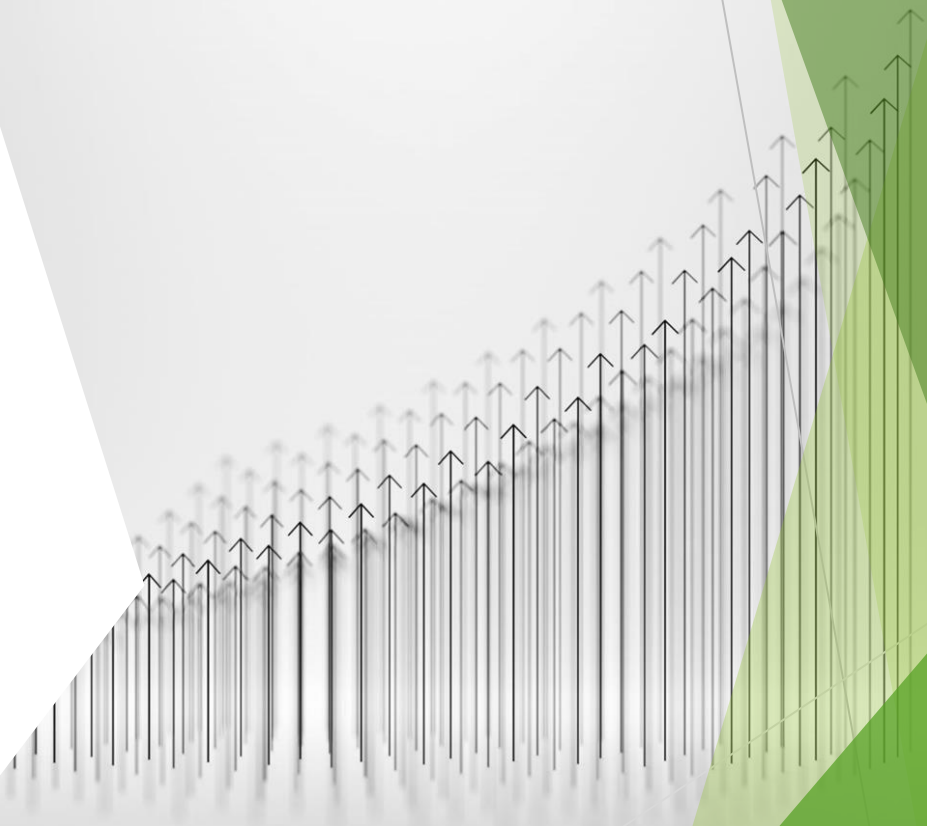


Tabla de Complejidad para algunas EDD

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Algoritmos de Ordenamiento



Definiciones iniciales

Algoritmo de ordenamiento

Los algoritmos de ordenamiento son un conjunto de instrucciones que toman un arreglo o lista como entrada y organizan los elementos en un orden particular.

Invariante

Se conoce como invariante a una condición que se sigue cumpliendo después de la ejecución de determinadas instrucciones (tanto antes como después de estas instrucciones) permaneciendo sin variación.

```
def maximo(lista):
    "Devuelve el elemento merximo de la lista o None si estar vacía."
    if not len(lista):
        return None
    max_elem = lista[0]
    for elemento in lista:
        if elemento > max_elem:
            max_elem = elemento
    return max_elem
```

*¿Por qué el valor de **max_elem** se considera el invariante?*

IMPORTANTE: *Parte de la sesión fue extraída de http://interactivepython.org/ para fines educacionales*

Bubble Sort

Es un algoritmo de ordenamiento que compara todos los elementos de un array (excepto el ultimo) con su vecino de la derecha e intercambia de posición si están “fuera de orden”. Esto es:

- Mueve siempre el mayor hacia “a la derecha”
- El último elemento siempre será el mayor (el segundo mayor queda penúltimo… y así sucesivamente)
- Compara los elementos y los intercambia si están fuera de orden

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
print(alist)
bubbleSort(alist)
print(alist)
```

```
[54, 26, 93, 17, 77, 31, 44, 55, 20]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Análisis del Bubble Sort

```
for passnum in range(len(alist)-1,0,-1):
    for i in range(passnum):
        if alist[i]>alist[i+1]:
```

Considerando **n = len(alist) = tamaño del arreglo**

1. *El loop exterior es ejecutado **n-1** veces (técnicamente es n veces)*
2. *Cada vez que el loop exterior se ejecuta, el loop interior es ejecutado*
3. *El loop interior se ejecuta **n-1** veces la primera vez y decrementa linealmente (en una unidad) para cada iteración*

Luego, tenemos que:

1. *En promedio el loop interior se ejecuta **n/2** veces por cada loop exterior (razonando sobre el punto 3 antes visto)*
2. *En el loop interior la comparación **siempre** se realiza y ello implica tiempo constante (al igual que el intercambio de valores)*

*El resultado de las operaciones sería de **n * n/2 + k**, en notación implica **O($n^2/2 + k$) = O(n^2)***

Invariante del Bubble Sort

Dado que el loop cambia está para “intercambiar elementos”, es importante encontrar un invariante en dicha iteración. Razonando:

1. En bubble sort, los numeros mayores quedan al final, y una vez puestos no se vuelven a mover.
2. La variable ***passnum*** comienza en el último índice del arreglo y hasta 0

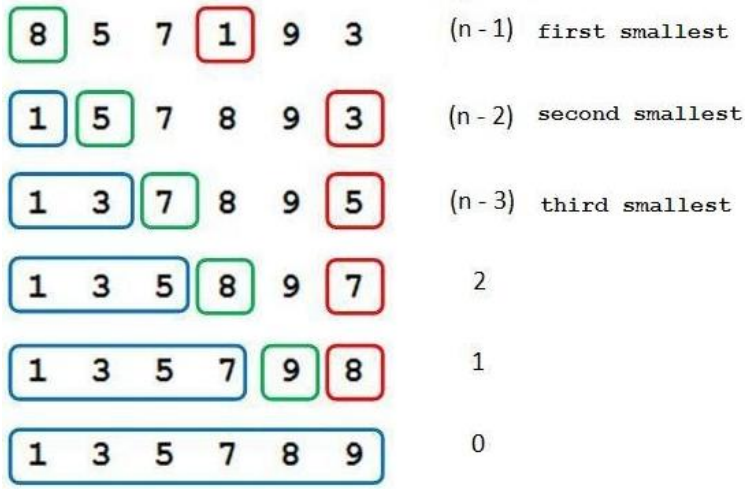
Luego, nuestro invariante consta de todos los elementos a la derecha de ***passnum*** pues están en la posición correcta y ***nunca se moverán*** de su posición. Esto es:

for all j > outter, if i < j, then a[i] <= a[j]

Selection Sort

Es un algoritmo de ordenamiento que compara todos los elementos de un arreglo buscando el mínimo elemento entre una posición i y el final de la lista e intercambia dicho mínimo con el elemento de dicha posición i-ésima.

```
def selectionSort(alist):
    for outter in range(len(alist)-1,0,-1):
        positionOfMax=0
        for inner in range(1, outter +1):
            if alist[inner ]>alist[positionOfMax]:
                positionOfMax = inner

        temp = alist[outter]
        alist[outter ] = alist[positionOfMax]
        alist[positionOfMax] = temp

alist = [54,26,93,17,77,31,44,55,20]
print(alist)
selectionSort(alist)
print(alist)
```




```
[54, 26, 93, 17, 77, 31, 44, 55, 20]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Análisis del Selection Sort

```
for outter in range(len(alist)-1,0,-1):
    positionOfMax=0
    for inner in range(1, outter +1):
        if alist[inner ]>alist[positionOfMax]:
            positionOfMax = inner

    temp = alist[outter]
    alist[outter ] = alist[positionOfMax]
    alist[positionOfMax] = temp
```

1. *El loop externo se ejecuta **n-1** veces (técnicamente es n veces)*
2. *El loop interno se ejecuta **n/2** veces en promedio*
3. *La operación en el loop interior es constante(intercambiar dos elementos)*
4. *El tiempo requerido es sería: **n*(n/2)***

*Lo anterior implica que se debe considerar como **O(n^2)***

Invariante del Selection Sort

Para el loop interior:

1. Este loop busca dentro del arreglo, incrementando interior desde su valor inicial **exterior+1** hasta **a.largo-1**
2. Mientras el loop continua, **min** se establece como el menor número encontrado hasta ahora.

Luego, nuestra invariante es:

for all i such that outter <= i <= inner, a[min] <= a[i]

Para el loop externo:

1. El loop cuenta desde outer = 0
2. Por cada loop, el valor mínimo queda guardado como a[outer]

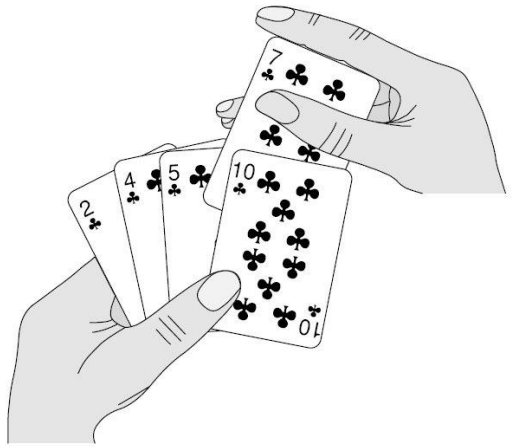
Luego, nuestra invariante es:

for all i <= outter, if i < j then a[i] <= a[j]

Insertion Sort

Es un algoritmo que cuando hay k elementos ordenados de menor a mayor:

- Toma el elemento k+1 y se compara con todos los elementos ya ordenados
  - *Se detiene cuando encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o*
  - *cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño).*
- En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementc "



```
def insertionSort(alist):
    for outer in range(1,len(alist)):
        minVal = alist[outer]
        i = outer - 1
        while(i>=0):
            if(minVal < alist[i]):
                alist[i+1] = alist[i]
                alist[i] = minVal
                i = i - 1
            else:
                break


alist = [54,26,93,17,77,31,44,55,20]
print(alist)
insertionSort(alist)
print(alist)
```

```
[54, 26, 93, 17, 77, 31, 44, 55, 20]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Análisis del Insertion Sort

```
for outer in range(1,len(alist)):
    minVal = alist[outer]
    i = outer - 1
    while(i>=0):
        if(minVal < alist[i]):
            alist[i+1] = alist[i]
            alist[i] = minVal
            i = i - 1
        else:
            break
```

1. *Recorremos una vez el loop exterior, insertando la totalidad de los **n** elementos. Esto es de factor **n**.*
2. *En promedio, hay **n/2** elementos ya ordenados por lo que el loop interior compara (y mueve) la mitad de ellos. Esto da un segundo factor de **n/4***

*Por lo tanto, el tiempo requerido para una insertion sort de un arreglo de **n** elementos es proporcional a **$n^2/4$**. Lo anterior implica que, obviando constantes, se debe considerar como **$O(n^2)$***

Invariante del Insertion Sort

El loop exterior es de forma:

```
for outer in range(1,len(alist)):
```

La invariante es que todos los elementos a la izquierda de **outer** están ordenados el uno con el otro, lo que implica que:

For all i < outer, j < outer, if i < j then a[i] <= a[j]

Esto no quiere decir que están en su posición definitiva ya que los elementos sobrantes pueden ser agregados. Lo anterior significa:

1. Buscar el lugar correcto del elemento
2. Hacer espacio para el elemento (al cambiar otros elementos) insertando el elemento

Estructuras de Datos básicas

Definiciones iniciales

Estructura de Datos

Es una forma de organizar un conjunto de datos elementales con el objetivo de facilitar su manipulación.

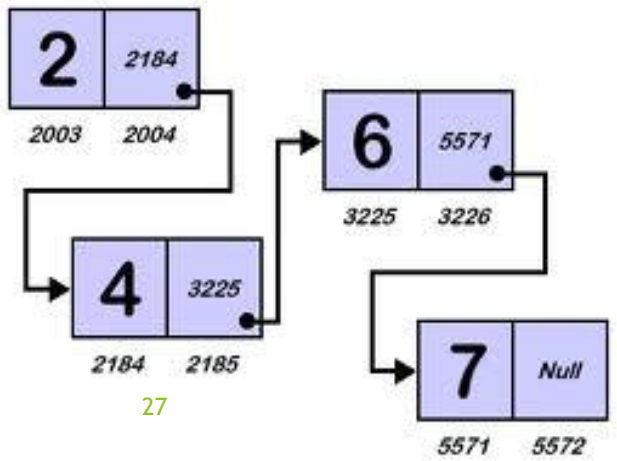
Lista enlazada

Es una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una o dos referencias, enlaces o punteros (punteros) al nodo anterior o posterior.

Nodo

Contenedor (de datos) que posee dos partes fundamentales:

- Espacio para la información
- Puntero(s)

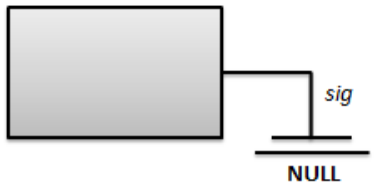


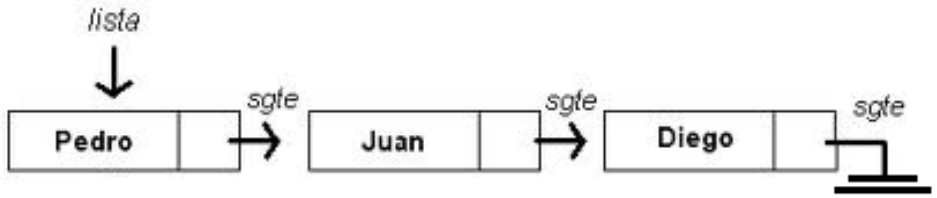
Listas enlazadas simples

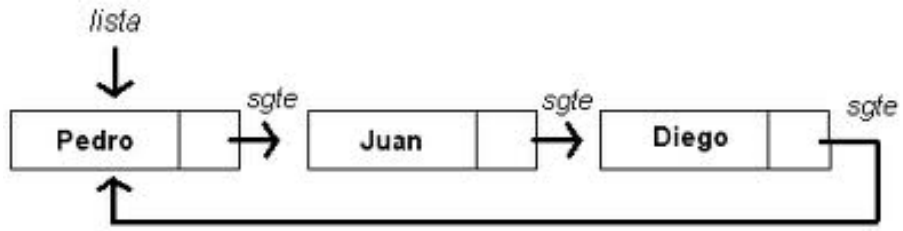
Es aquella que tiene un enlace por nodo. Dependiendo de si es lista simple o si es circular simple este enlace:

- ***Simple:*** Apunta al siguiente nodo en la lista, o al valor NULL o a la lista vacía, si es el último nodo.
- ***Circular:*** Apunta al siguiente nodo de la lista o al primero si es el último nodo.

```
#Clase Nodo simple con puntero
class Nodo(object):
    def __init__(self,elemento):
        #Atributo que tendrá el Nodo - Puede tener más
        self.__elemento=elemento
        #Puntero que servirá para "unir" a los Nodos
        #cuando se construya la lista
        self.__pSig = None

    def getElemento(self):
        return self.__elemento
```





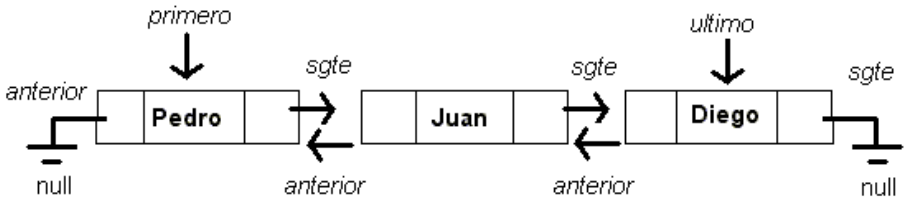


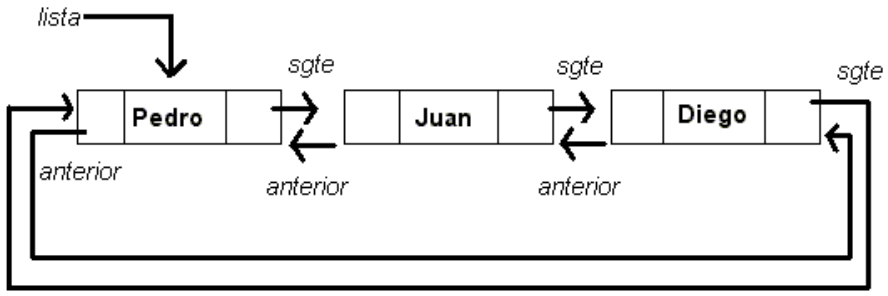
Listas doblemente enlazadas

Es aquella que tiene dos enlaces por nodo. Dependiendo si es o no circular, estos enlaces:

- ***Doble:*** Apuntan al siguiente y al anterior nodo en la lista. El anterior del primero es nulo y el siguiente del último es nulo.
- ***Circular doble:*** Apuntan al siguiente y al anterior nodo en la lista. El anterior del primero es el último y el siguiente del último es el primero.

```
#Clase Nodo con dos punteros
class Nodo(object):
    def __init__(self,elemento):
        #Atributo que tendrá el Nodo - Puede tener más
        self.__elemento=elemento
        #Dos punteros que servirán para "unir" los Nodos
        #cuando se construya la lista. Siguiente y Anterior.
        self.__pSig = None
        self.__pAnt = None

    def getElemento(self):
        return self.__elemento
```





Colas - Queues

Es una lista ordenada o estructura de datos en la que el modo de acceso a sus elementos es de tipo FIFO.

El Nodo para Colas posee:

- Uno o más atributos (sus elementos)
- Un atributo adicional: Un puntero hacia “atrás” o hacia “adelante” (solo uno)

Respecto a la cola esta posee dos nodos centinela:

- Elemento primero
- Elemento último.

Hablemos un poco sobre las operaciones sobre las Colas.

Pilas - Stacks

Es una lista ordenada o estructura de datos en la que el modo de acceso a sus elementos es de tipo LIFO.

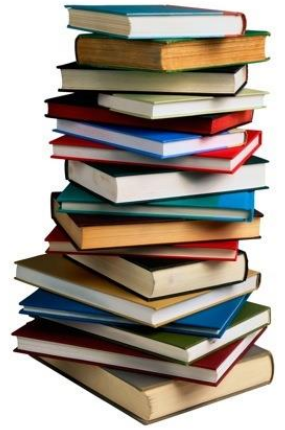
El Nodo para Pilas posee:

- Uno o más atributos (sus elementos)
- Un atributo adicional: Un puntero hacia “abajo” o hacia “arriba” (solo uno).

Respecto a la pila esta posee un nodo centinela:

- Elemento “tope”

Hablemos un poco sobre las operaciones sobre las Pilas.



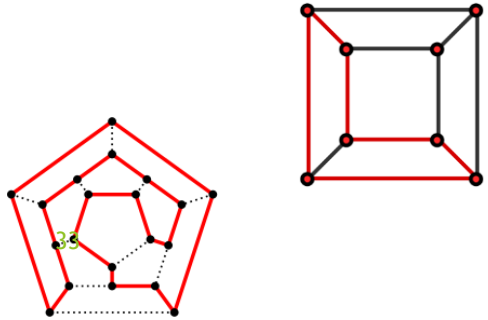
Estructuras de Datos avanzadas

Definiciones iniciales

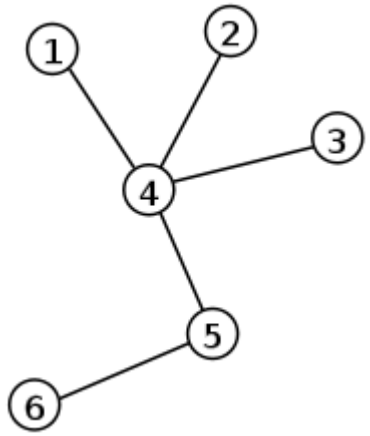
Teoría de Grafos

Un grafo G=(V,E) es una pareja ordenada en la que V es un conjunto no vacío de vértices y E es un conjunto de aristas. Estos poseen:

- *Vértice: El vértice es el elemento (nodo) que compone un grafo (un nodo). Puede tener asociado un valor N que representa su Grado.*
- *Grado: Es la cantidad de conexiones que posee un vértice.*
- *Camino: Es el conjunto de vértices interconectados a través de una "ruta" de aristas. Es decir, el recorrido que se hace para llegar de un nodo X a un nodo Y en un Grafo. La cantidad de vértices intermedios determinan "el largo" del camino (distancia entre dos vértices específicos).*
- *Arista: Es la línea que une los vértices de un grafo. Esta puede ser dirigida (de origen a destino) o no dirigida). Entre las aristas, tenemos:*
  - *Aristas adyacentes: Aristas que convergen en el mismo vértice.*
  - *Aristas paralelas: Aristas cuyo si el vértice inicial y final son el mismo.*
  - *Aristas cíclicas: Comienza y termina en el mismo vértice.*



Tipos de Grafo

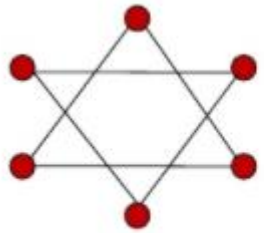


Grafo Simple

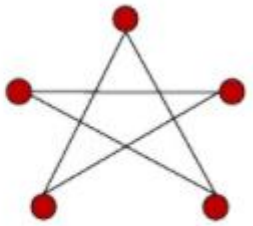
- Es aquel donde al menos una arista une dos vértices cualesquiera.
- No existe “loop” ni más de una arista entre dos vértices.

Grafo Conexo

- Es aquel donde cada par de vértices está conectado por un camino.



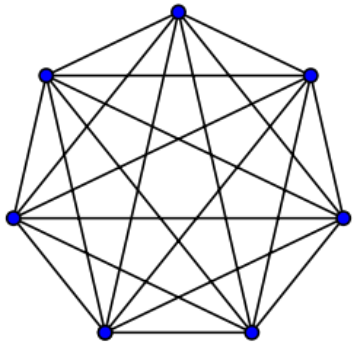
Grafo inconexo



Grafo conexo

Grafo Completo

- Es aquel donde todos los pares de vértices tienen una arista que los une.
- Posee n(n-1)/2 aristas en función de sus n vértices.



Árbol Binario

Es una estructura de datos en la cual cada nodo puede tener un hijo izquierdo y un hijo derecho. No pueden tener más de dos hijos (de ahí el nombre "binario").

- Si algún hijo tiene como referencia a null, es decir que no almacena ningún dato, entonces este es llamado un nodo externo.
- En el caso contrario el hijo es llamado un nodo interno

Un árbol binario está compuesto por cero o más nodos donde cada nodo contiene:

- Valor o contenedor
- Referencia al hijo izquierdo (left) (puede ser nulo)
- Referencia al hijo derecho (right) (puede ser nulo)

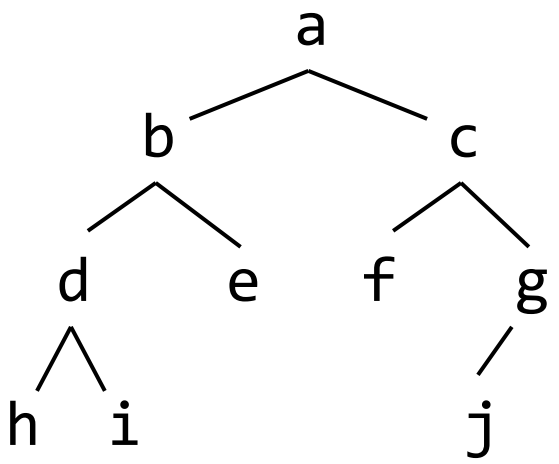
IMPORTANTE

- Un árbol binario puede estar vacío, pero si no está vacío, tiene un nodo raíz (root)
- Cada nodo del árbol puede ser alcanzado desde la raíz siguiendo una única ruta

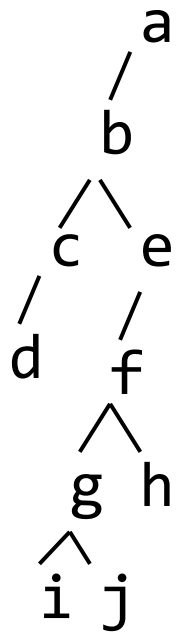
Árbol Binario

```
class Hoja(object):
    def __init__(self,peso):
        self.pIzq = None
        self.pDer = None
        self.peso = peso
        #No tiene más datos... por ahora

    def getPeso(self):
        return self.peso


    def getHoja(self):
        return self
```

```
class Arbol(object):
    def __init__(self):
        self.raiz = None

    def getVacio(self):
        if(self.raiz == None):
            return True
```



Árbol balanceado



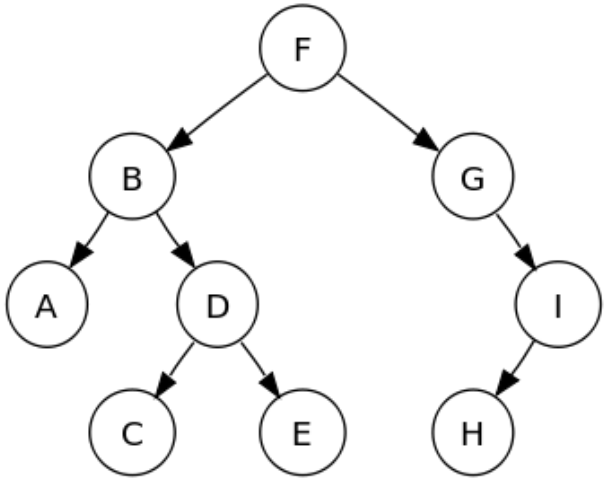
Desbalanceado

Inserción en Árbol Binario

```
def insertarHoja(self,hojaNueva):
#hojaNueva es un elemento del tipo Hoja que posee un valor en
#su atributo "peso"
    if(self.getVacio()==True):
        self.raiz = hojaNueva

    else:
        seguir = True
        temp = self.raiz
        while(seguir):

            if(temp.getPeso()>=hojaNueva.getPeso() and temp.pIzq == None):
                temp.pIzq = hojaNueva
                seguir = False

            elif(temp.getPeso()>=hojaNueva.getPeso()):
                temp = temp.pIzq

            if(temp.getPeso()<hojaNueva.getPeso() and temp.pDer == None):
                temp.pDer = hojaNueva
                seguir = False

            elif(temp.getPeso()<hojaNueva.getPeso()):
                temp = temp.pDer

def setRaiz(self,raizNueva):
    self.raiz = raizNueva
```

Algoritmos de Recorrido en Profundidad

Actualmente se conocen 3 algoritmos de recorrido en profundidad.



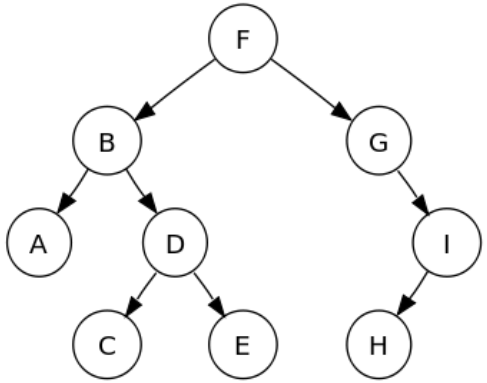
Preorden

En este tipo de recorrido:

- Se realiza cierta acción (por ejemplo un print) sobre la raíz,
- posteriormente se trata el subárbol izquierdo y cuando se haya concluido,
- el subárbol derecho.

Esto se hace recursivamente. Otra forma para entender el recorrido con este método seria seguir el orden: nodo raíz, nodo izquierda, nodo derecha.

Algoritmos de Recorrido en Profundidad



Postorden

En este tipo de recorrido:

- Se trabaja el subárbol izquierdo,
- después el subárbol derecho y
- al final la hoja raíz.

Otra forma para entender el recorrido con este método seria seguir el orden: nodo izquierda, nodo derecha, nodo raíz.

Inorden

En este tipo de recorrido:

- Se trabaja el subárbol izquierdo,
- después el nodo raíz y
- por último el subárbol derecho.

Otra forma para entender el recorrido con este método seria seguir el orden: nodo izquierda, nodo raíz, nodo derecha.

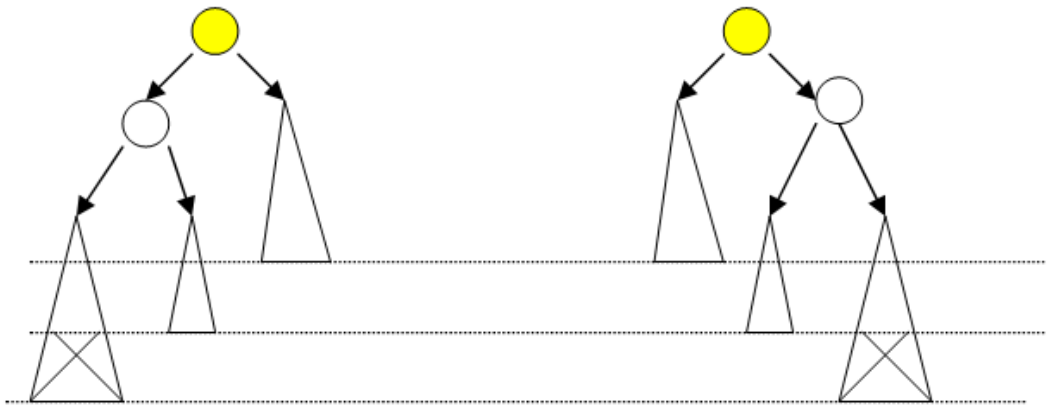
Árboles AVL

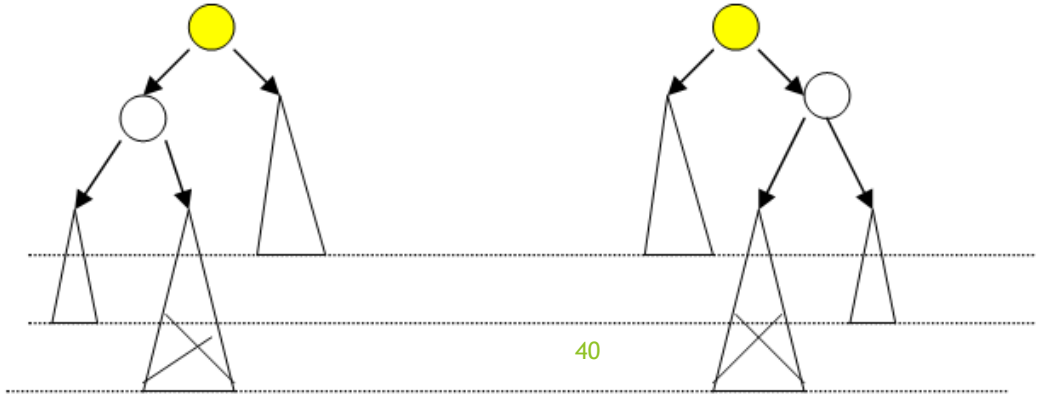
Los árboles AVL son árboles binarios “autobalanceados”. Esto se hace a través del factor de balance (alto del subárbol derecho menos el alto del subárbol izquierdo).

Factor de Balance = h(subárbol derecho) - h(subárbol izquierdo)

En definitiva

Todos los AVL tiene un factor de balance igual a: -1, 0, o 1



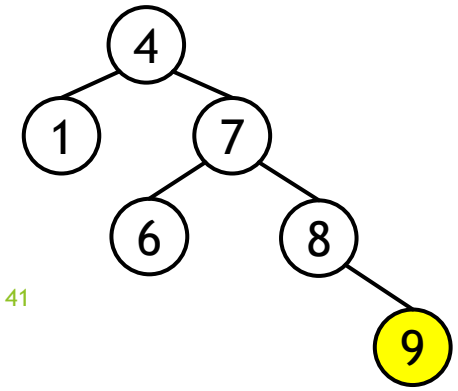


Rotación simple en AVL

Caso 1: Rotación Simple a la Izquierda

Se hace cuando se desbalancea el árbol por la inserción de un nodo la hoja externa del lado derecho. Considerando un árbol de Raíz R y cuyo hijo izquierdo es I y el derecho es D, tenemos que:

- Formamos un nuevo árbol cuya raíz sea la raíz del hijo derecho
- Luego, como hijo derecho colocamos el hijo derecho de D
- y como hijo izquierdo construimos un nuevo árbol que tendrá como raíz, la raíz R del árbol original, el hijo izquierdo de D será el hijo derecho y el hijo izquierdo del árbol original será el hijo izquierdo del nuevo subárbol.

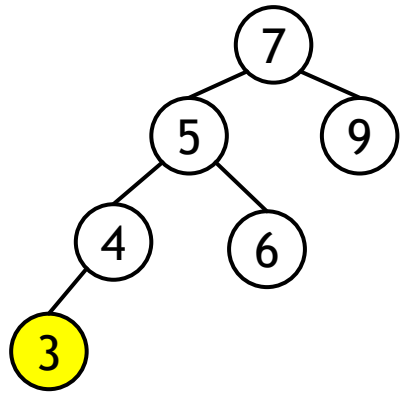


Rotación simple en AVL

Caso 2: Rotación Simple a la Derecha

Se hace cuando se desbalancea el árbol por la inserción de un nodo la hoja externa del lado izquierdo. Considerando un árbol de Raíz R y cuyo hijo izquierdo es I y el derecho es D, tenemos que:

- Formamos un nuevo árbol cuya raíz sea la raíz del hijo Izquierdo
- Luego, como hijo izquierdo colocamos el hijo izquierdo de I
- y como hijo derecho construimos un nuevo árbol que tendrá como raíz, la raíz R del árbol original, el hijo derecho de I será el hijo izquierdo y el hijo derecho del árbol original será el hijo derecho del nuevo subárbol (subárbol derecho).



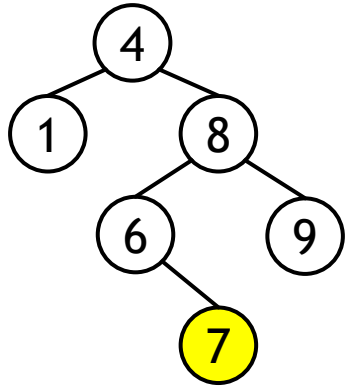
Rotación Doble en AVL

Caso 3: Rotación Doble de Izquierda a Derecha

Se hace cuando se desbalancea el árbol por la inserción de un nodo (izquierdo o derecho) como hijo del hijo izquierdo del hijo derecho de la raíz del árbol.

Considerando un árbol de Raíz R y cuyo hijo izquierdo es I y el derecho es D, tenemos que:

- Hacemos una Rotación Simple a la Derecha sobre el subárbol derecho y, luego
- Hacemos una Rotación Simple a la Izquierda sobre el árbol completo



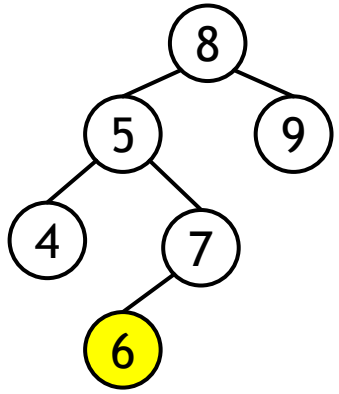
Rotación Doble en AVL

Caso 4: Rotación Doble de Derecha a Izquierda

Se hace cuando se desbalancea el árbol por la inserción de un nodo (izquierdo o derecho) como hijo del hijo derecho del hijo izquierdo de la raíz del árbol.

Considerando un árbol de Raíz R y cuyo hijo izquierdo es I y el derecho es D, tenemos que:

- Hacemos una Rotación Simple a la Izquierda sobre el subárbol izquierdo y, luego
- Hacemos una Rotación Simple a la Derecha sobre el árbol completo

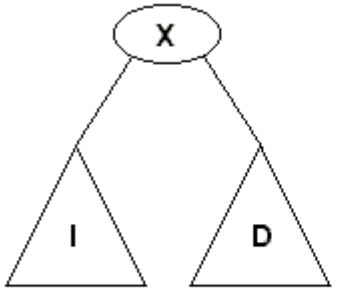


Arboles 2-3

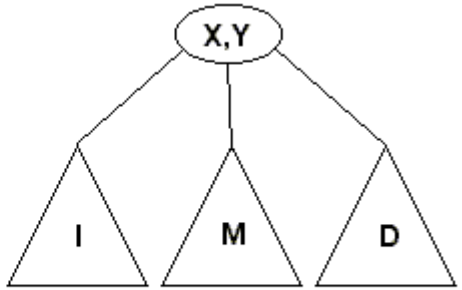
Es un árbol en donde los nodos interiores pueden contener hasta 2 elementos y, además, pueden tener 2 o 3 hijos, dependiendo de cuántos elementos posea el nodo. Por ello, siempre estarán “perfectamente” balanceados.

¿Por qué están “perfectamente” balanceados?

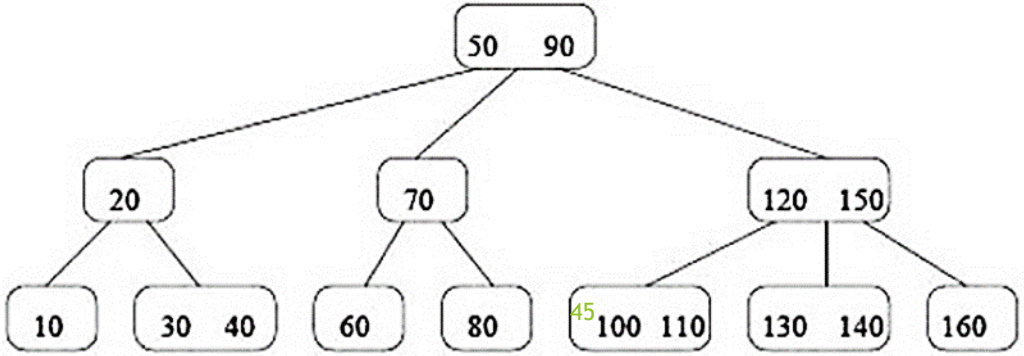
Una propiedad que poseen los árboles 2-3 es que todas sus hojas están a la misma profundidad.



I < X < D



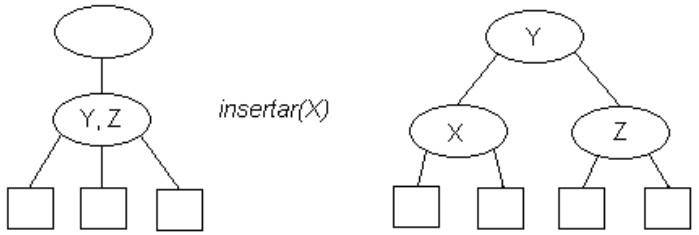
I < X < M < Y < D



Insertar elementos en Arboles 2-3

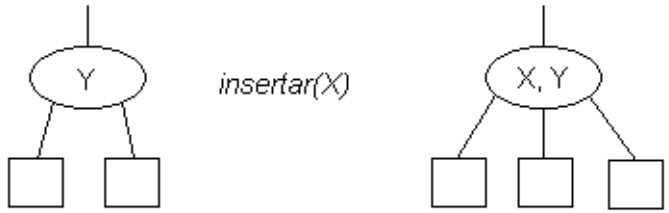
Caso 1: El nodo donde se insertará el dato posee dos datos (datos Y y Z) y tres enlaces

En este caso se obtendrá un nodo padre con dos hijos donde se respeta que X < Y < Z donde X, Y y Z son los datos "de peso" de los nodos. Gráficamente:



Caso 2: El nodo donde se insertará el dato posee un dato (dato Y) y dos enlaces

En este caso se obtendrá un nodo padre con tres hijos Nulos y con dos elementos donde se respeta que si X < Y los datos ordenados serán X,Y respectivamente. Por ejemplo:



Árbol Digital

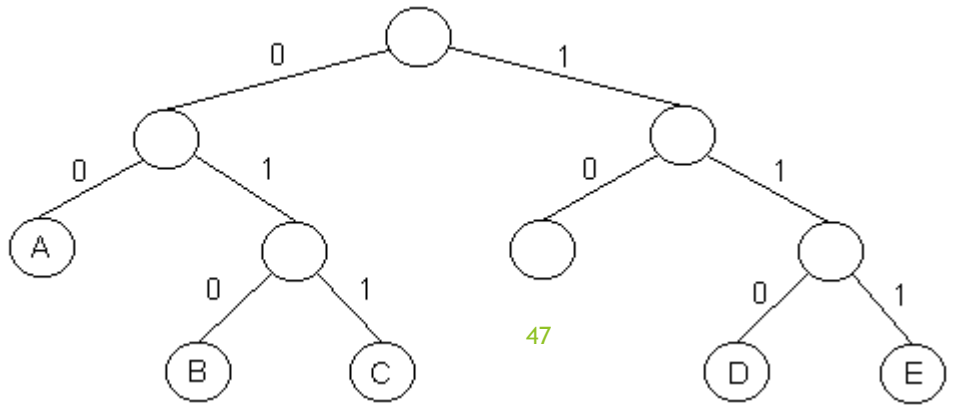
Son árboles donde los elementos se almacenan en los nodos internos de igual forma que en los árboles binarios, pero sus llaves están dadas por bits y, en base a ello, se dan sus ramificaciones. Entonces, tenemos que los elementos se representan como una secuencia de bits de la forma:

X = b1b2b3…bn

Luego, la posición de inserción de un elemento ya no depende de su valor, sino de su representación binaria en un **alfabeto definido**. Por ello, **no todas las hojas contendrán elementos.**

Codificación:

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



Búsqueda en un Árbol Digital

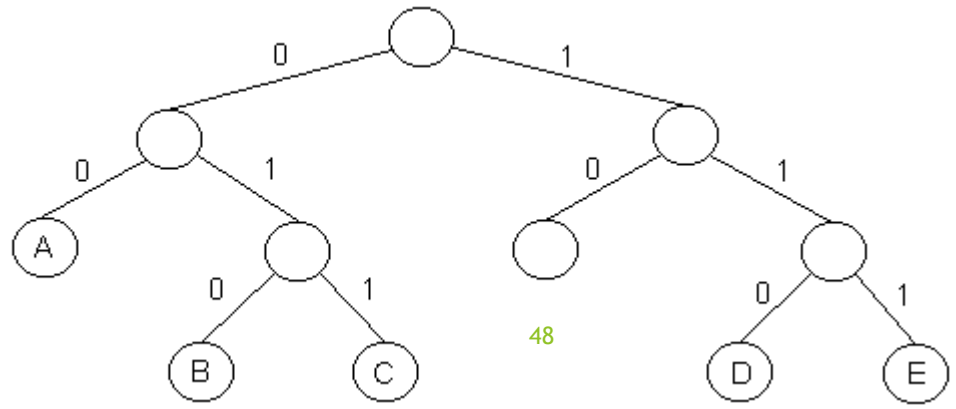
Para buscar un elemento X en un árbol digital se procede de la siguiente manera:

- ✓ Se examinan los bits bi del elemento X, partiendo desde b0 en adelante.
- ✓ Si bi = 0 se avanza por la rama izquierda y se examina el siguiente bit, bi+1.
- ✓ Si bi = 1 se avanza por la rama derecha y se examina el siguiente bit.

El proceso termina cuando se llega a una hoja, único lugar posible en donde puede estar insertado X.

Codificación:

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



Generalidades sobre el Hashing

Una tabla hash o mapa hash es una estructura de datos que asocia llaves o claves con valores.

Premisa principal

No existe una fórmula "única" para las funciones de Hash. Sin embargo, tenemos 3 pasos fundamentales:

- ***Paso 1***

Representar la llave de manera numérica en un conjunto determinado (siempre que no sea de por sí un número de dicho conjunto)*.

Ejemplo

El string “mauricio” en binario sería:

“01101101011000010111010101110010011010010110001101101001011011111”

Generalidades sobre el Hashing

- ***Paso 2***

Separar nuestro “numero” para tener componentes más pequeños y facilitar la operación con ellos.

01101101 | 01100001 | 01110101 | 01110010 | 01101001 | 01100011 | 01101001 | 01101111

Algunas operaciones que podemos realizar incluyen la suma, invertir o cualquiera que podamos generar con el conjunto que tenemos.

- ***Paso 3***

Dividir por un número primo (representado en el alfabeto) y usar el resultado como dirección.

NOTA

Sin perjuicio de lo anterior, se pueden realizar muchas operaciones adicionales según sea el caso. Por ejemplo: Transformar el resultado (en este caso binario) a otro conjunto (e.g. hex)

Hashing Extendible

Este esquema de hashing funciona, en promedio, cuando se almacenan N = O(MB) datos en total, es decir, unos miles de veces del tamaño de la memoria principal.

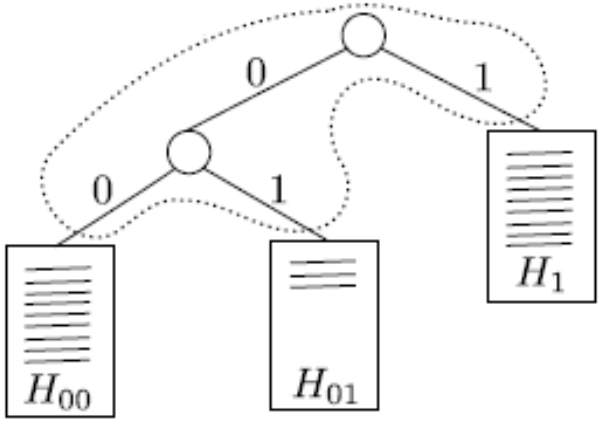
Inicialmente, la estructura es una única página H en disco donde los datos se insertan de cualquier manera, al costo de O(1) I/Os . Entonces:

1. Las búsquedas se hacen leyendo la página H y buscando secuencialmente la clave que se desea.

2. Una vez que esta página se llena, la siguiente inserción provoca que la dividamos en dos: H0 y H1. Para ello, releemos cada dato “y” de H y calculamos h(y).
    a. Según el primer bit de h(y) sea 0 o 1, insertamos el elemento en H0 o H1, respectivamente.
    b. Luego, creamos un nodo en memoria principal con dos hijos: el izquierdo apunta a la página de disco donde almacenamos H0, y el derecho a la de H1.

Hashing Extendible

3. Supongamos que, más adelante, la página de H0 rebalsa por una inserción. Recorreremos todos los elementos y de H0 y consideraremos el segundo bit de h(y) para separar los elementos en dos hojas, H00 y H01. La hoja de H0 se reemplazaría entonces por un nodo interno, cuyos hijos izquierdo y derecho serían, respectivamente, H00 y H01.

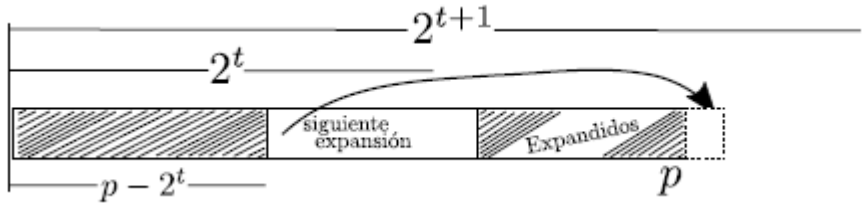


¿A qué se parece esto?

Hashing Lineal

Este esquema de hashing funciona para resolver colisiones ya que el direccionamiento (abierto) se basa en:

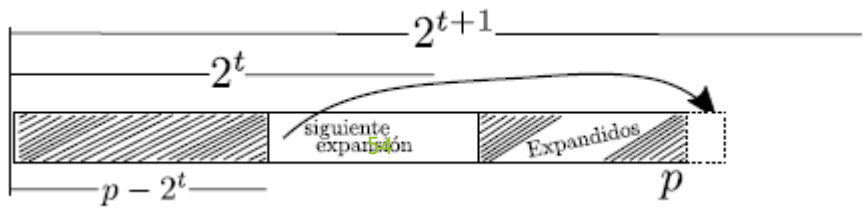
1. Hacer un recorrido dentro de la tabla a partir del índice de la tabla en donde se produce la colisión.
2. Se recorre todo hasta encontrar una posición que este vacía y almacenar (la clave) en ese lugar dependiendo si lo que se busca es almacenar.
3. De no encontrar ningún espacio con el índice buscado o espacio para almacenar se regresará a la posición inicial como si se tratara de un ciclo.
4. Si debemos solucionar una colisión… ¿Qué podemos hacer? ¿Expandir? ¿Usar listas?



Hashing Lineal: Solución de Colisiones

Pensemos que el archivo de hashing (en disco) tuviera siempre 2^t páginas. Luego:

1. Un elemento "x" y está guardado en la página número h(y) mod 2^t (es decir, los t bits más bajos de h(y)).
2. Si algunas páginas rebalsan durante la inserción (por colisión):
   a. Creamos una lista enlazada de "páginas de rebalse".
   b. Si, luego de un rebalse, notamos que el costo de búsqueda (es decir, 1 más el largo promedio de las listas de rebalse) se ha hecho "demasiado alto", expandimos la tabla. Expandir significa duplicar su tamaño a 2^{t+1}.
   c. Cada página i, con $0 \leq i \leq 2^t$, se recorre y sus elementos y se reinsertan en la página h(y) mod 2^{t+1}. Esto significa que una parte de los elementos se quedan en la página i, mientras que otros se insertan en la página $i + 2^t$.



Cierre de la sesión