ALGORITMOS Y ESTRUCTURA DE DATOS III

Sección TQ - 1er Semestre/2018

Unidad: *Búsqueda y Ordenación*

Árboles Binarios de Búsqueda *(BST)*

Prof. Cristian Cappo
7/03/2018

En esta clase

- Motivación del problema de búsqueda
- Concepto de diccionario
- Concepto de recuperación
- Árboles. Concepto. Representación.
- Árboles Binarios de Búsqueda
  - Usos y Tipos
  - Definición e implementaciones
  - Recorridos
  - Operaciones
  - Estructura de Dato en Java

Búsqueda

- Es una operación fundamental en computación, igual que la ordenación.
- ¿Cómo recupero rápidamente un valor de memoria dada una clave?
  - Cuando los datos son estáticos
  - Cuando los datos son dinámicos

Diccionario

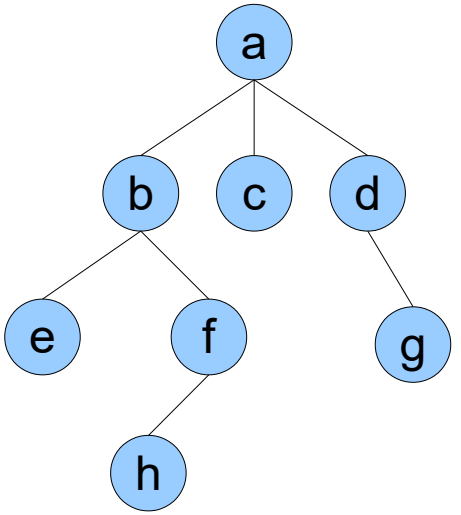
- Teniendo una clave, encuentro la información asociada a ésta
- Por ejemplo: un guía telefónica, un diccionario de palabras, tabla de símbolos (compiladores),etc.
- Nosotros consideraremos tres estructuras de datos para implementar diccionario: ***Árbol de Búsqueda Binario***, Árbol de Búsqueda Binario Balanceado y Tabla de Dispersión (Hash)

Diccionario

- Operaciones básicas dada una clave k y un diccionario D
  - Existencia (existe ***k*** en ***D***?)
  - Recuperación (retorna el elemento en ***D*** que coincide con ***k***)
  - Búsqueda asociada (retorna la información asociada en ***D*** a ***k***) (*datos satélites*)

Árboles

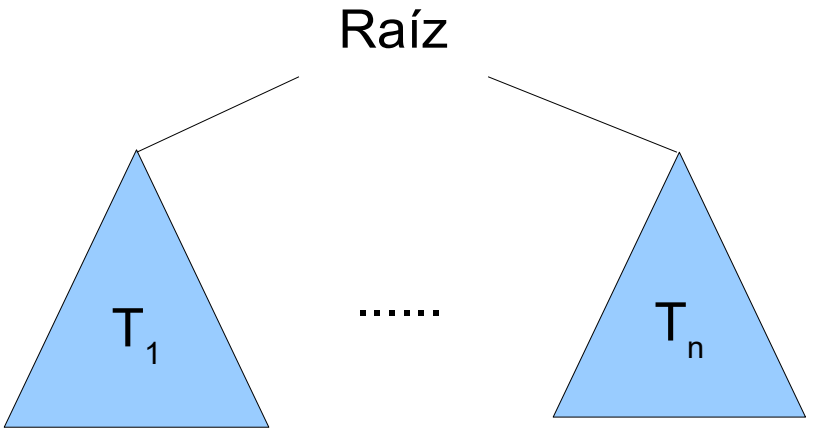
- Conjunto de nodos y aristas orientadas que conectan pares de nodos
- Consideramos árbol con raíz:
  - A cada nodo *c*, excepto la raíz, le llega una arista desde exactamente otro nodo *z*, al cual llamamos padre de *c*. Decimos que *c* es uno de los hijos de *z*.
  - Existe un único camino desde la raíz hasta el nodo. El número de aristas es la longitud del camino.
  - Nodos sin hijos= ***hojas***. Los demás=***nodos internos***.
    Hojas: c,e,h,g *Nodos internos*: a,b,d,f



  - *Profundidad de un nodo*: longitud del camino desde la raíz al nodo. (Ejemplos: la profundidad de **b** es 1, de **f** es 2, de **h** es 3, etc)
  - *Altura de un nodo*: longitud del camino del nodo hasta la hoja más profunda. La altura del árbol es la altura de la raíz. (Ejemplos: la altura de **b** es 2, de **a** es 3, de **d** es 1, etc)
  - *Tamaño de un nodo*: número de descendientes incluyéndose él. Tamaño del árbol es el tamaño de la raíz (Ejemplos: el tamaño de **b** es 4, el de **a** es 8, etc)

Árboles

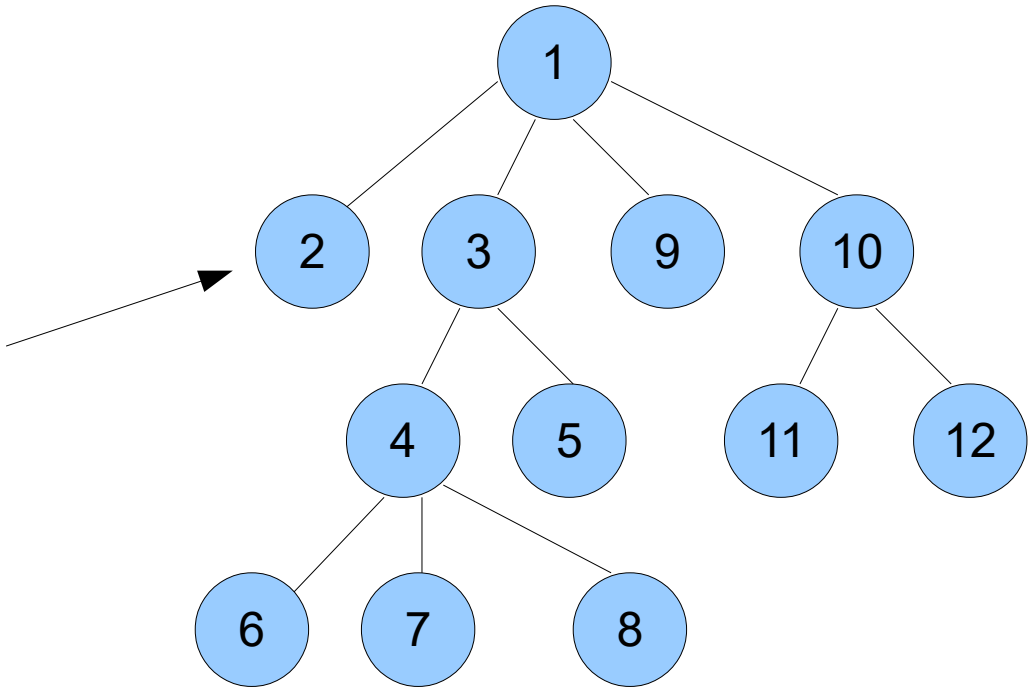
- ***Definición recursiva:*** colección de elementos que es vacío o consiste en un elemento raíz y cero o más subárboles no vacíos T_1, T_2, .. , T_n cada una de sus raíces conectada por medio de una arista a la raíz.



Implementaciones

- ***Primer hijo/Sigte Hermano:***

  Mantenemos los hijos de cada nodo en una lista enlazada. Cada nodo tiene dos referencias: uno a su hijo más a la izquierda y otra a su hermano de la derecha



¿Cuál sería la representación de punteros en memoria de este árbol?

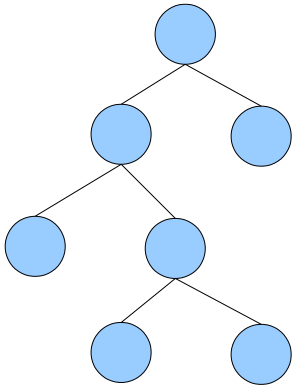
Diseñe en Java las clases necesarias para esta representación

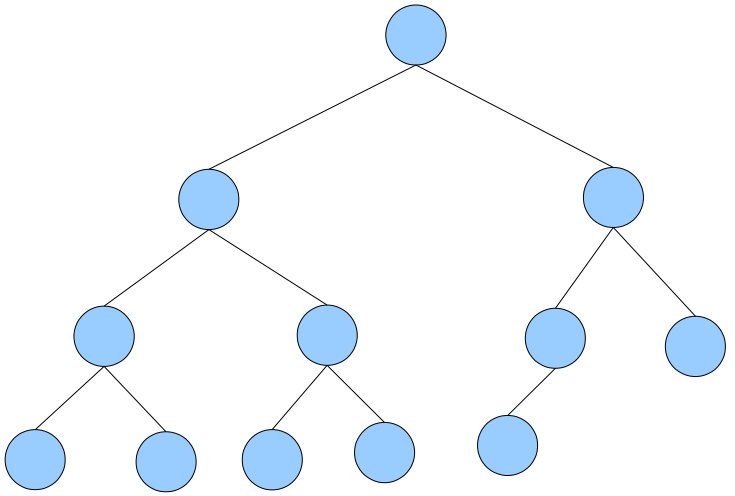
Árbol Binario

- Ningún nodo puede tener más de dos hijos. Hijo izquierdo e Hijo derecho.
- **Definición recursiva**: es una colección vacía o consta de una raíz de un hizo izquierdo y otro derecho que son árboles binarios.
- Ejemplos de uso:
  - Árboles de expresión (compiladores, evaluación de expresiones)
  - Codificación de Huffman (compresión)

Tipos de árboles binarios

- **Lleno**: todos los nodos tienen exactamente 0 o 2 hijos
- **Completo**: todos los niveles están completos a excepción del último que se completa de izquierda a derecha.





Recorridos

- Postorden : hijos, raiz
- Inorden u orden simétrico : hijos izq, raiz, hijos der
- Preorden : raiz, hijos
- Por niveles (amplitud) : raiz, hijos 1er nivel, hijos 2do nivel, etc

Árboles Binarios de Búsqueda
(BST (*Binary Search Tree*))

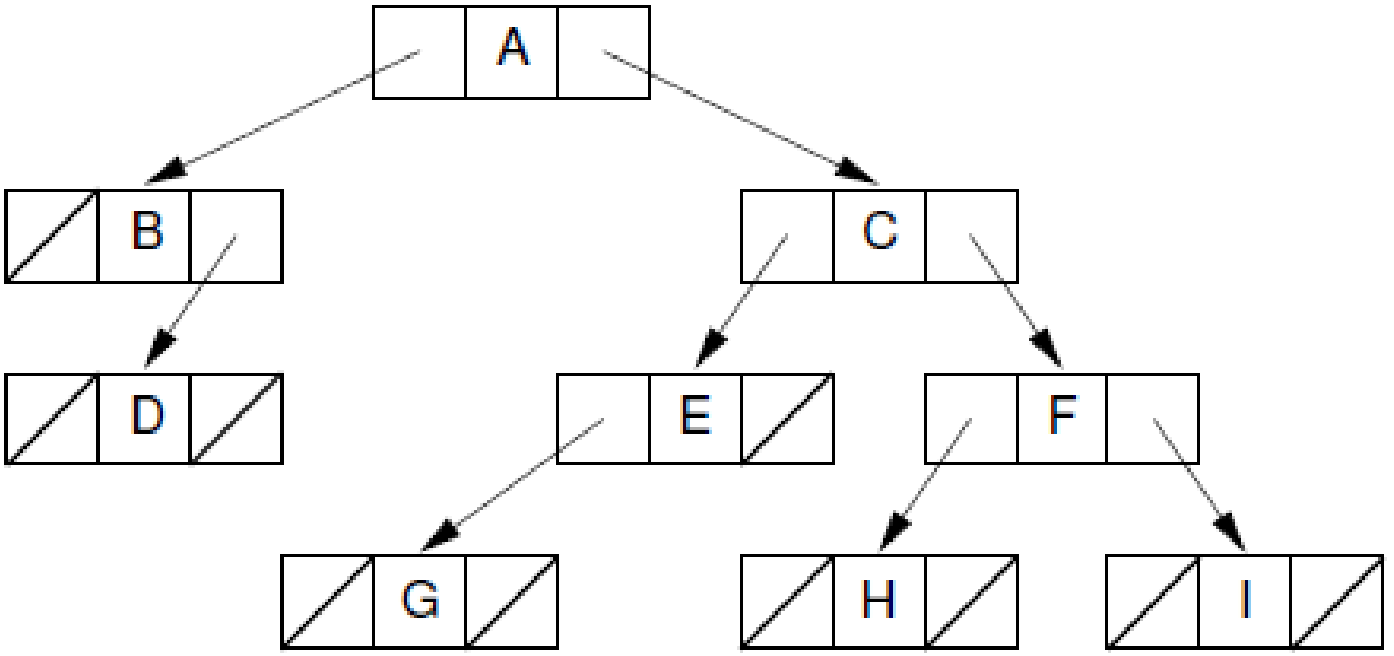
- Cada nodo tiene una clave comparable y datos asociados, que satisface la restricción que la clave en cualquier nodo es **mayor** que las claves de todos los nodos que se encuentran en el sub-árbol izquierdo y es **menor** que todas las claves del sub-árbol derecho. ***Esta es una propiedad del BST***
- Es un caso especial de árbol binario. Cada nodo posee: clave, datos satélite, puntero al hijo izquierdo, al hijo derecho y eventualmente un puntero al padre.

Árboles Binarios de Búsqueda (II)

- ¿Es posible tener claves duplicadas?
- La condición de ubicación de las claves permite listar en forma ordenada las claves aplicando un recorrido **INORDEN** del árbol

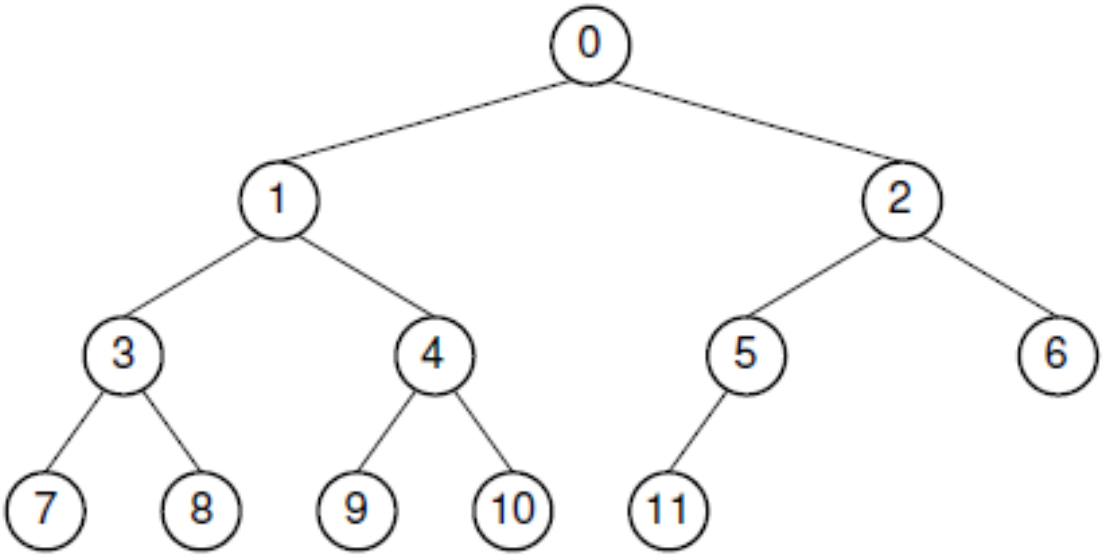
Implementaciones de BST

- Basada en nodos y punteros



Implementaciones de BST

- Basadas en arreglos (implica un árbol binario completo)



Posición	0	1	2	3	4	5	6	7	8	9	10	11
Padre	–	0	0	1	1	2	2	3	3	4	4	5
Hijo Der	1	3	5	7	9	11	–	–	–	–	–	–
Hijo Izq	2	4	6	8	10	–	–	–	–	–	–	–
Hno. Izq.	–	–	1	–	3	–	5	–	7	–	9	–
Hno. Der.	–	2	–	4	–	6	–	8	–	10	–	–

BST (Recorrido inorden)

```
Recorrido-In-Orden ( X )
    si ( X ≠ NULL )
      Recorrido-In-Orden ( X.IZQ )
      Imprimir(X)
      Recorrido-In-Orden ( X.DER )
```

Donde la primera llamada es *Recorrido-In-Orden(T.Raiz)*, asumiendo que **T** es un **BST**

Teorema

Si X es la raíz de un sub-árbol de tamaño n, entonces la llamada Recorrido-In-Orden(X) toma un tiempo de *O(n)*

Demostración del teorema anterior

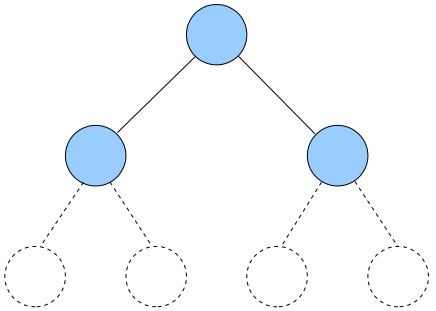
Dado *T(n)* que denota el tiempo de *Recorrido-In-Orden(X)* cuando es llamado en la raíz de un sub-árbol de tamaño *n*. *Recorrido-In-Orden* toma una pequeña y constante cantidad de tiempo en un sub-árbol vacío (para `X = NULL`) y así *T(0)* = *c* para alguna constante positiva *c*.

Para *n>0*, suponga que *Recorrido-In-Orden* es llamado de un nodo *X* donde el sub-árbol izquierdo tiene *k* nodos y el sub-árbol derecho tiene *n-k-1* nodos. El tiempo de *Recorrido-In-Orden(X)* es
T(n) = *T(k)* + *T(n-k-1)* + *d* para alguna constante *d* que refleja el tiempo de ejecutar la función en cada recursión (por ejemplo *imprimir*)

Usando sustitución mostraremos que *T(n)* = *O(n)* probando que
T(n) = *(c+d) n* + *c* => *c* = tiempo para los punteros NULL y *d* = tiempo para cada nodo no NULL.

Demostración del teorema anterior (cont.)

La ecuación tentativa siempre tiene alguna fundamentación.



Teorema: el número de sub-árboles vacíos en un árbol binario es uno más que el nro. de nodos del árbol.

Prueba: por definición, cada nodo en un árbol T binario tiene dos hijos, para un total de 2n hijos en un árbol de n nodos. Cada nodo, excepto la raíz, tiene un padre, para un total de n-1 nodos con padre. En otras palabras, hay *n-1* nodos hijos no vacíos. Ya que el nro. total de hijos es *2n*, el resto *n+1* hijos deben estar vacíos.

Así podemos decir que el nro. de nodos vacíos es *n+1*. Verificar cada nodo vacío lleva un tiempo c y el procesarlo un tiempo d. Así el tiempo total para n nodos es $T(n) = (c+d)n + c$

Demostración del teorema anterior (cont.)

Recurrencia : T(n) = T(k) + T(n-k-1) + d
Supuesto : T(n) = (c+d)n + c

Reemplazamos el supuesto en la recurrencia.

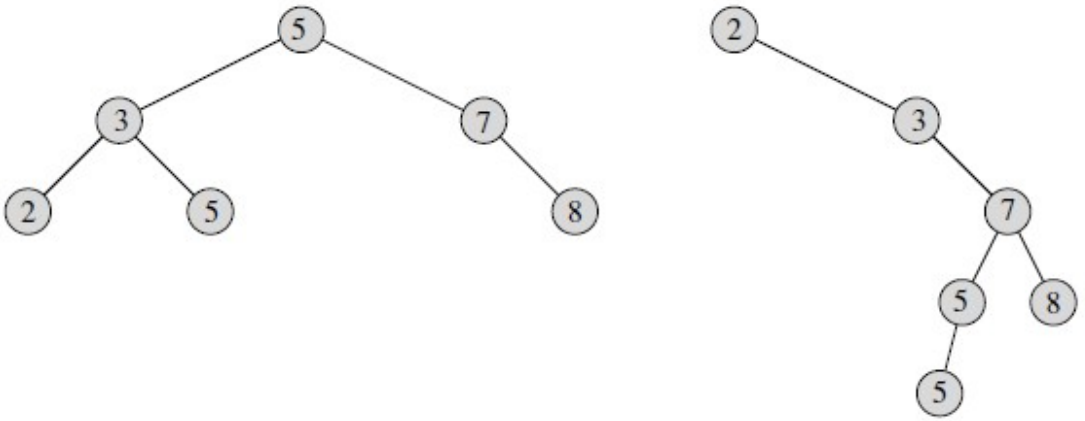
Para n=0 tenemos que *(c+d).0 + c = c = T(0)*

Para n>0 tenemos

$$
\begin{aligned}
T(n) &= T(k) + T(n-k-1) + d \\
&= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\
&= (c+d)n + c - (c+d) + c + d \\
&= (c+d)n + c \quad \blacksquare
\end{aligned}
$$

Operaciones en BST

- Inserción, búsqueda, borrado, mínimo, máximo,sucesor, predecesor.
- Toman un tiempo proporcional a la altura (porque?). Para un BST lleno y completo con n nodos, llevaría *O(log n)*



- Si las claves son randómicamente insertadas se espera un tiempo *O(log n)*
- Un BST perfectamente balanceado tiene $2^{h+1} – 1$ nodos para algun *n*>=1 y una altura *h*

ED básico para un BST

```
// BST (define la cáscara o contenedor - container )
public class BST<T extends Comparable<T>>
{
    private class NodoBST<T> { // Nodo del BST
        public T dato = null;
        public NodoBST<T> izq = null;
        public NodoBST<T> der = null;

        public NodoBST (T dato)
        {
          this.dato = dato;
        }
    }

    private NodoBST<T> raiz = null;

    public BST(){
    }

    // operaciones sobre el BST
    // (agregar, eliminar, buscar, imprimir, etc)
    ...
```

ED básico para un BST

```
    /* Codigo para agregar al BST */

    public void agregar (T dato)
    {
       raiz = priv_agregar (raiz, dato);
    }

    private NodoBST<T> priv_agregar (NodoBST<T> n_actual, T dato)
    {
       if ( n_actual == null )
          return ( new NodoBST<T>(dato) );

       int comparacion = dato.compareTo(n_actual.dato);

       if ( comparacion < 0 )
          n_actual.izq = priv_agregar(n_actual.izq,dato);
       else
          n_actual.der = priv_agregar(n_actual.der,dato);
       return n_actual;
    }

    ...
} /* Fin de la clase BST */
```

Ejercicios de clase (1)

Ejercicio #0
Para el conjunto de claves {1,4,5,10,16,17,21}, dibuje los BSTs con altura 2, 3, 4, 5, 6.

Ejercicio #1
Grafique un árbol binario de búsqueda para la secuencia (92, 88, 15, 39, 80, 23, 40, 2, 10, 18) y luego para la secuencia (192, 40, 18, 39, 480, 93, 20, 4, 89, 18). Para cada caso indique la altura del árbol y luego escriba el resultado de recorrer el árbol en preorden, postorden, orden simétrico y por niveles.

Ejercicio #2
Dado un árbol binario de altura 5. ¿Cuántos nodos como mínimo debe tener para ser considerado lleno y completo? Dibujar.

Ejercicio #3
Suponga que tenemos números entre 1 y 1000 en un BST y que queremos buscar el número 363. ¿Cuál de las siguientes secuencias corresponde a la búsqueda en un BST?

- 2, 252, 401, 398, 330, 344, 397, 363.
- 924, 220, 911, 244, 898, 258, 362, 363.
- 925, 202, 911, 240, 912, 245, 363.
- 2, 399, 387, 219, 266, 382, 381, 278, 363.
- 935, 278, 347, 621, 299, 392, 358, 363.

Ejercicios de clase (2)

Ejercicio #4

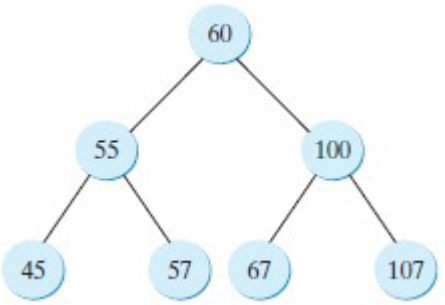
Dada la implementación BST mostrada en clase, proponga un método recursivo que retorne la altura del mismo.

```
int height ();
```

Indique cual sería la ecuación de recurrencia de su algoritmo. Analice su solución y encuentre el tiempo de ejecución de su algoritmo con la debida fundamentación.

Ejercicio #5

Escriba un método que reciba un elemento del BST y retorne, como una lista, el camino desde la raiz hasta ese elemento. Por ejemplo en el BST de abajo, si el parámetro fuese 107, entonces se retorna (60,100,107). Si el parámetro fuese 45 retornaría (60,55,45).



Bibliografía

- Clifford A. Shaffer. Data Structures and Algorithm Analysis. Edition 3.2 (Java Edition) . 2012. (Cap. 5)
- T. Cormen, C. Leiserson, R. Rivest y C. Stein. Introduction to algorithms. Second edition. MIT Press. 2009. (Cap 12)
- R. Sedgewick y K. Wayne. Algorithms. Addison Wesley. 2011 (Cap 3 (3.2 y 3.3))
- Estructura de Datos en Java, Mark Allen Weiss. 2000. (Cap. 17 y 18)