## Análisis de Algoritmos

***NOTAS DE CLASE***

Objetivos al resolver un problema:

* Algoritmo fácil de entender y debuggear
* Uso eficiente de recursos (memoria) y velocidad

El tiempo de ejecución de un programa depende de:

* Los datos de entrada (cantidad)
* La calidad del código
* La rapidez de las instrucciones usadas
* La complejidad de tiempo del algoritmo base

El peor caso de un algoritmo es cuando se ejecutan todos los casos posibles, es decir, cuando se ejecuta por más tiempo. Se define el tiempo de ejecución del peor caso como T(N).

Cálculo del tiempo de ejecución de un programa:

* Reglas de la suma y producto en notación asintótica.
* El tiempo de ejecución de cada **proposición** de asignación puede tomarse como O(1).
* El tiempo de ejecución de una **secuencia de proposiciones** se determina por la regla de la suma.
* El tiempo de ejecución de una **proposición condicional if** es el costo de las proposiciones que se ejecutan condicionalmente, más el tiempo para evaluar la condición (se toma O(1)).
* El tiempo para ejecutar un **ciclo** es la suma sobre todas las iteraciones del ciclo, del tiempo de ejecución del cuerpo y del empleado para evaluar la condición de terminación. Normalmente este tiempo es el producto del número de iteraciones del ciclo por el mayor tiempo posible para una ejecución del cuerpo.

Def Algoritmo: Conjunto claramente especificado de instrucciones.

Cuando se proporciona un algoritmo:

1. Se verifica que es correcto
2. Se determina la cantidad de recursos (tiempo y espacio en memoria) > Analisis de Algoritmo

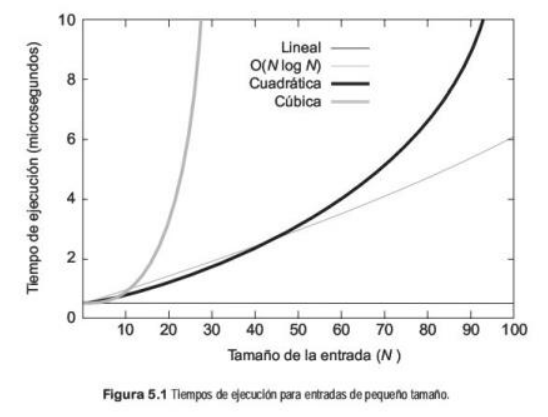
La cantidad de tiempo que demora un algoritmo en ejecutarse depende de la cantidad de entrada a procesar. El tiempo de ejecución es una función del tamaño de la entrada. El valor exacto de la función va a depender de:

* la velocidad de la máquina
* la calidad del compilador
* la calidad del programa
* entre otros

Hay 4 funciones comunes en el análisis de algoritmos:

1. **Lineal:**  El término dominante es una constante multiplicada por N
2. **O(N log N):** El termino dominante es N veces el logaritmo de N.
3. **Cuadrática:** El término dominante es una constante multiplicada por N^2
4. **Cúbica:** El término dominante es una constante multiplicada por N^3

Representan distintos tiempos de ejecución, en orden de preferencia decreciente, siendo la lineal la representación de los algoritmos más eficientes.

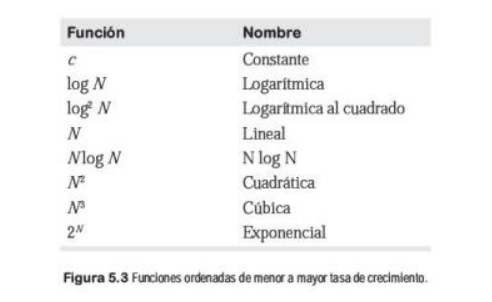


Se miden las **tasas de crecimiento** de las funciones en lugar de calcular su valor en puntos específicos. Para N suficientemente grandes, el valor de una función está determinado por su término dominante. Ademas, el valor exacto de la constante que multiplica al termino dominante no tiene significado si estamos comparando distintas maquinas, porque segun la maquina ese valor puede ser distinto.

La notación O (o notación de Landau) captura el término más dominante de una función y representa la tasa de crecimiento. El tiempo de ejecución de un algoritmo cuadrático se especifica como O(N^2) (de orden N cuadrado).

Para pequeños tamaños de entrada, los tiempos de ejecución suelen ser irrelevantes. Cuando eso sucede, es buena regla práctica utilizar el algoritmo que sea más simple.

Los algoritmos cuadráticos no son prácticos cuando el tamaño de entrada supera unos pocos miles, mientras que los algoritmos cúbicos dejan de ser prácticos para tamaños de unos pocos centenares. Ejemplo, no es práctico emplear un algoritmo de ordenación sencillo para un millón de elementos porque los algoritmos de ordenación más simples son algoritmos cuadráticos.



**Ejemplos de tiempos de ejecución en algunos algoritmos**  
  
1) Elemento mínimo de una matriz: Dada una matriz N, determinar el menor de ellos.

Se puede resolver de la siguiente forma:

1. Mantener una variable min que guarde el elemento mínimo
2. Inicializar min con el valor del primer elemento
3. Recorrer secuencialmente la matriz y actualizar min

El tiempo de ejecución de este algoritmo es O(N) porque se repite una cantidad fija de trabajo para cada elemento de la matriz. Es un algoritmo lineal.

2) Puntos más próximos en el plano: Dados N puntos en un sistema de coordenadas x-y, encontrar el par de puntos más próximo entre sí.

Se puede resolver de la siguiente forma:

1. Calcular la distancia entre cada pareja de puntos
2. Quedarse con la distancia mínima

Como cada uno de los N puntos puede emparejarse con N-1 puntos, nos da un total de N(N-1)/2 parejas. Por lo tanto hay aproximadamente N^2 pares de puntos. La entrada crece de manera cuadrática, por lo que el algoritmo requiere un tiempo cuadrático. Para mejorar el algoritmo se debería evitar tener que calcular todas las distancias.

**Reglas Generales para el Cálculo de O *(página 198, “Estructuras de Datos en Java”, Mark Allen Weiss)***



Existe un punto N0 tal que para todos los valores de N mayores a ese punto, T(N) está acotado por algún múltiplo de F(N). Se afirma que la tasa de crecimiento T(N) es menor o igual que la de F(N).



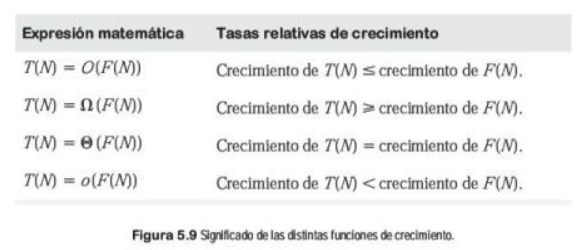
Afirma que la tasa de crecimiento T(N) es mayor o igual que la de F(N).



Afirma que la tasa de crecimiento de T(N) es igual a la tasa de crecimiento de F(N). El tiempo de ejecución está acotado por una función y esta cota no puede mejorarse porque también tiene como cota inferior otra función.



La tasa de crecimiento de T(N) es estrictamente inferior a la tasa de crecimiento de F(N). Es diferente de O porque O admite la posibilidad de que las tasas de crecimiento colindan.



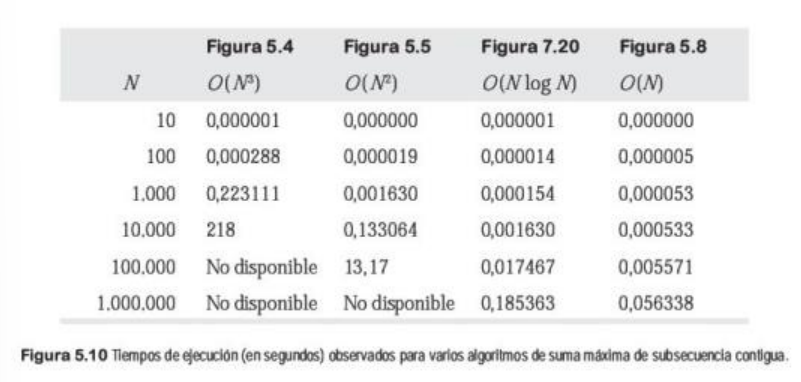
Tiempos de ejecución de bucles:

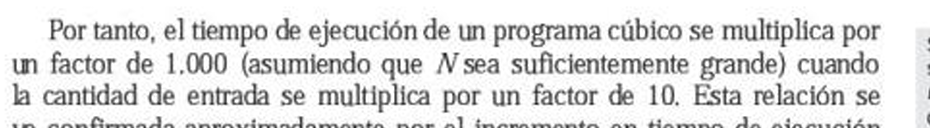
El tiempo de ejecución **de un bucle** es, como máximo, el tiempo de ejecución de las instrucciones contenidas dentro del bucle (incluyendo las comparaciones) multiplicado por el numero de iteraciones.

El tiempo de ejecución de las **instrucciones contenidas dentro de un grupo de bucles anidados** es igual al tiempo de ejecución de las instrucciones multiplicado por los tamaños de todos los bucles. (Cuadrático) (N^2)

El tiempo de ejecución de una **secuencia de bucles consecutivos** es igual al tiempo de ejecución del bucle dominante. (Lineal) (2N)

La diferencia de tiempo entre un bucle anidado en el que ambos indices vayan de 1 a N y dos bucles consecutivos no anidados pero que se ejecutan a lo largo del mismo rango de índices, es similar a la diferencia de espacio existente entre una matriz bidimensional y dos matrices unidimensionales. (N^2 - 2N = diferencia).











### El Logaritmo

Definición: Para cualquier B,N>0, logB N = K si B^K = N.

Papeles que desempeñan los logaritmos:

**El numero de bits requerido para representar numeros es logaritmico.**

16 bits representan 65.538 enteros

B bits representan 2^B enteros distintos

El numero B de bits requeridos para representar N enteros cumple 2^B > N

Entonces B > log N

Entonces el numero minimo de bits es log N.

**Duplicaciones consecutivas: partiendo de 1, solo podemos duplicar la cantidad repetidamente log N de veces hasta alcanzar N**

## Listas Enlazadas

Un objeto puede tener una referencia a otro objeto del mismo tipo. Las listas estan compuestas por **nodos**, donde cada nodo tiene una **referencia al próximo nodo** de la lista. La unidad de datos de cada nodo se llama **carga**. La referencia al próximo nodo del ultimo nodo es null. El primer nodo sirve como referencia a una lista entera.

public class Nodo {

int carga;

Nodo prox;

public Nodo () {

carga = 0;

prox = null;

}

public Nodo (int carga, Nodo prox) {

this.carga = carga;

this.prox = prox;

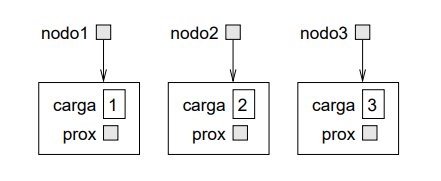
}

public String toString () {

return carga + ""; //+ “” convierte un entero a un string

}

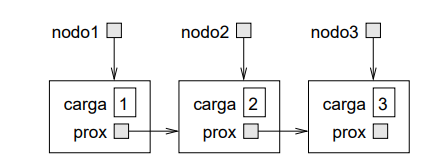
}



Nodo nodo1 = new Nodo (1, null);

Nodo nodo2 = new Nodo (2, null);

Nodo nodo3 = new Nodo (3, null);



nodo1.prox = nodo2;

nodo2.prox = nodo3;

nodo3.prox = null;

<https://youtu.be/WwfhLC16bis?si=VTK9X_uEf5CE0j15>

**Listas como colecciones**

Con una lista podemos ensamblar muchos objetos en una sola entidad llamada colección.

Para pasar una lista como parámetro a algún método, hay que pasar la referencia al primer nodo.

La forma de moverse a través de una lista se llama recorrido. Es común usar un iterador para referenciar a cada nodo sucesivamente.

public static void imprimirLista (Nodo lista) {

Nodo nodo = lista;

while (nodo != null) {

System.out.print (nodo);

nodo = nodo.prox;

}

System.out.println ();

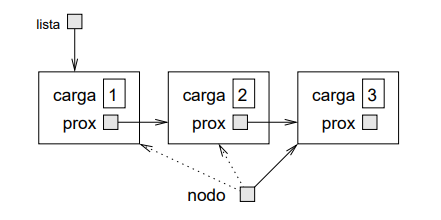
}

Dentro de imprimirLista hay una referencia al primer nodo de la

lista, pero no hay ninguna variable que se refiera a los demás nodos.

Hay que usar el valor de **prox** de cada nodo para llegar al próximo

nodo.



**Listas y recursión**

<https://replit.com/@Cotidib/Linked-List-Nodes>

**Listas infinitas**

Un nodo puede referenciar a un nodo anterior de la lista. Incluso también puede referenciarse a sí mismo. Si llamaramos *imprimirLista* en dicha lista, caería en un loop infinito. Son difíciles de trabajar pero algunas veces son útiles, como por ejemplo para representar números periódicos.

“Si la lista no contiene ciclos, entonces estos métodos terminan”. A este tipo de afirmaciones se las llama **precondición.**

## Pilas

Un **tipo de dato abstracto** especifica un conjunto de operaciones (o métodos) y la semantica de las operaciones (qué es lo que hacen) pero no especifica la implementación de las mismas.

**El TAD Pila**

Una pila es una colección, es una estructura de datos que contiene múltiples elementos. Las pilas pueden hacer las siguientes operaciones:  
  
*constructor:* Crear una pila vacía

*apilar (oush) :* Agregar un nuevo elemento al final de la pila

*despilar (pop) :* Quitar y devolver un item de la pila. El item devuelto es el ultimo agregado.

*estaVacia (isEmpty) :* Responder si la pila está vacía

Una pila es una estructura LIFO, Last In First Out.

Una pila es una estructura de datos genérica, lo cual significa que podemos agregar cualquier tipo de ítems a ella. En la implementación de Java, sin embargo, sólo podemos agregar objetos y no valores de tipos nativos.

<https://replit.com/@Cotidib/Stacks#src/main/java/Main.java>

**Implementando TAD Pila usando arreglos**

Un TAD separa los intereses del proveedor, que escribe el código que implementa el TAD, del cliente, que es quien lo usa. El proveedor solo se preocupa de que la implementación sea correcta. El cliente asume que la implementación es correcta.

public class Stack {

Object[] arreglo; //contiene los elementos de la pila

int indice; //siguiente espacio disponible

public Stack () {

this.arreglo = new Object[128];

this.indice = 0;

}

public boolean isEmpty () {

return indice == 0;

}

public void push (Object elem) {

if (indice < arreglo.length)redimensionar ();

arreglo[indice] = elem;

indice++;

}

public Object pop () {

indice--;

return arreglo[indice];

}

private void redimensionar () {

Object[] nuevoArreglo = new Object[arreglo.length \* 2];

// asumimos que el arreglo anterior estaba lleno

for (int i=0; i<arreglo.length; i++) {

nuevoArreglo[i] = arreglo[i];

}

arreglo = nuevoArreglo;

}

}

Al final de redimensionar, reemplazamos el arreglo viejo con el nuevo. Esto es una **postcondición** de redimensionar (algo que tiene que ser cierto cuando el metodo esta completo).

## Colas y Colas de Prioridad

El TAD Cola y el TAD Cola de Prioridad tienen el mismo conjunto de operaciones y sus interfaces son la misma. La diferencia esta en la semantica de las operaciones: una cola usa FIFO (First in First Out) mientras que la Cola de Prioridad usa Encolado con Prioridades.

Una cola es una colección de elementos. Se pueden usar cualquiera de las formas básicas de almacenar elementos, incluyendo arreglos y listas. Un objeto cola contiene una única variable de instancia, que es la lista que la implementa.

El TAD Cola se define con las siguientes operaciones:

constructor: Crear una nueva cola, vacía.

agregar: Quitar un elemento de la cola y devolverlo. El elemento que se devuelve es el primero en haber sido agregado.

estaVacia: Verifica si la cola está vacía.  
  
import java.util.LinkedList;  
public class Cola {

private LinkedList lista;

public Cola() {

lista = new LinkedList ();

}

public boolean estaVacia() {

return lista.isEmpty();

}

public void agregar(Object obj) {

lista.addLast (obj);

}

public Object quitar() {

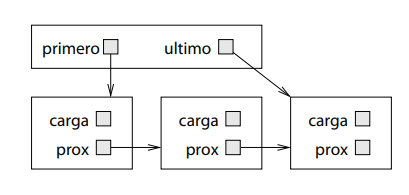
return list.removeFirst ();

}

}

**Cola Enlazada**

Para hacer una implementación de Cola que pueda efectuar todas las operaciones en tiempo constante, se puede implementar una cola enlazada que es similar a una lista enlazada. La diferencia es que la cola mantiene una referencia tanto al primero como al último nodo.

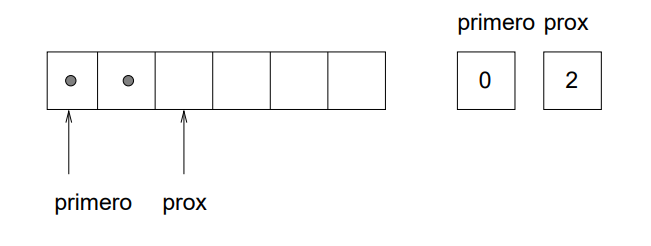


<https://replit.com/@Cotidib/Colas#src/main/java/ColaEnlazada.java>

**Buffer Circular**

La palabra Buffer es un nombre general para un espacio de almacenamiento temporal. La implementación de un buffer circular es similar a la implementación de la pila. Los elementos de la cola se almacenan en un arreglo y usamos índices para mantener un registro de donde estamos ubicados.

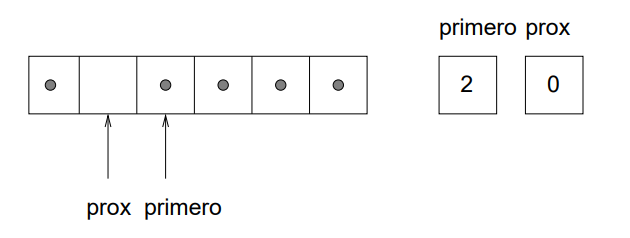
En la pila había un único índice que apunta al siguiente espacio disponible. En la implementación de cola, hay dos índices, el que apunta al espacio del arreglo que contiene el primer elemento y el que apunta al siguiente espacio disponible.



LOS INDICES NO SON REFERENCIAS, SOLO SON ENTEROS

Cuando no hay “próximo espacio disponible,” es decir que prox no tiene a dónde apuntar, una posibilidad es redimensionar el arreglo. Pero en ese caso el arreglo sólo continuaría creciendo independientemente de cuántos elementos haya efectivamente en la cola. Una mejor solución es dar la vuelta hasta el principio del arreglo y **reutilizar los espacios disponibles** ahí. Esta “vuelta” es la razón por la cual se conoce a esta implementación

como un buffer circular.

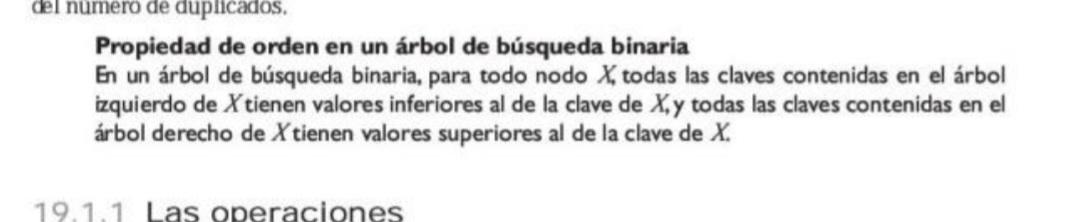


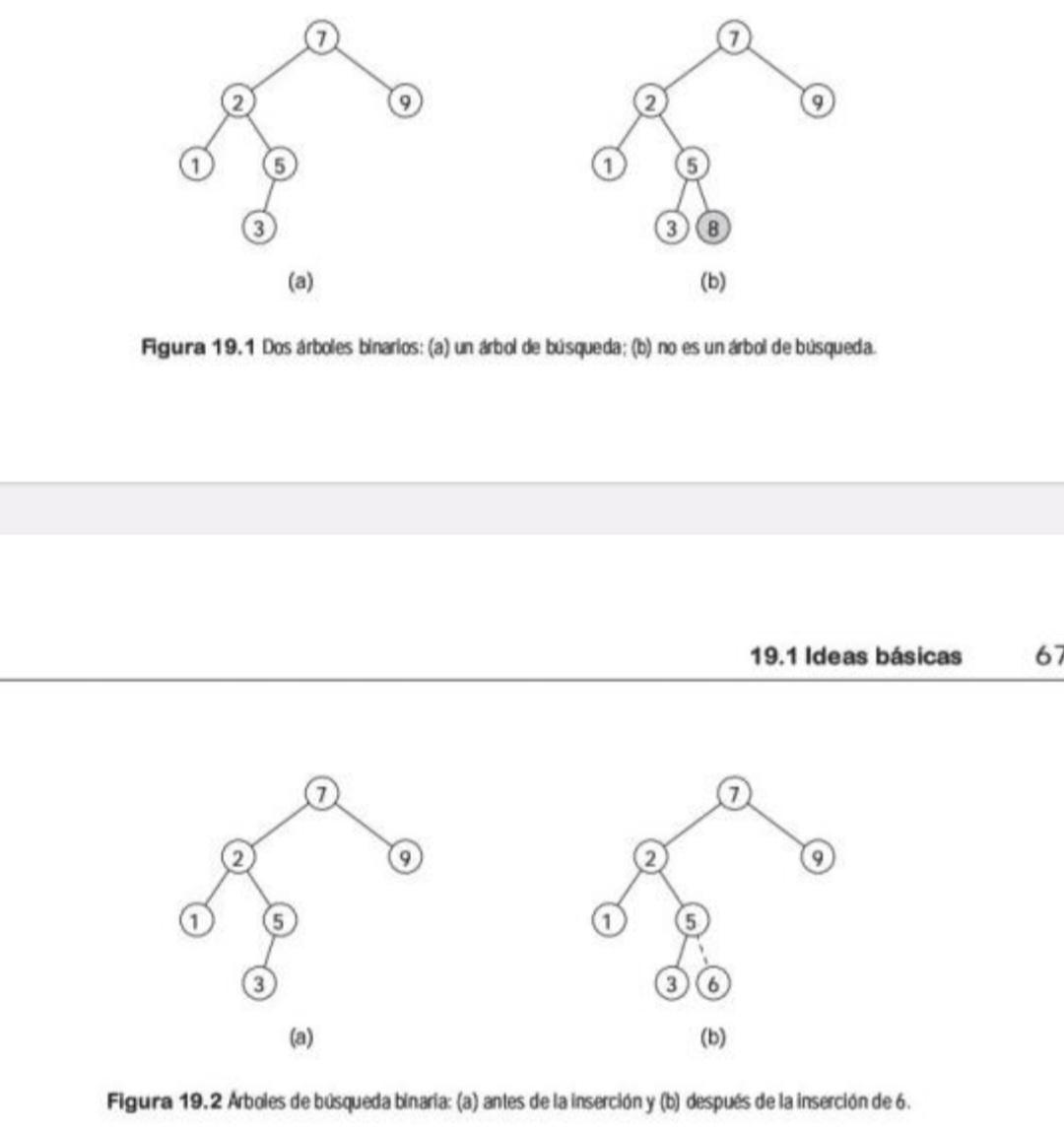
Hay aún un espacio disponible en el arreglo, pero la cola está llena porque si agregamos otro elemento, entonces tendríamos que incrementar prox de modo tal que prox == primero, en cuyo caso ¡parecería que la cola estuviera vacía!. ¿Y qué deberíamos hacer si el arreglo está lleno? En ese caso, redimensionar el arreglo.

<https://replit.com/@Cotidib/Colas#src/main/java/Cola.java>

# Arboles Binarios

Para grandes cantidades de datos, el tiempo de acceso lineal en las listas enlazadas es muy grande. Como alternativa para inserción y borrado se usan los Árboles de Búsqueda Binaria donde el tiempo de ejecución para la mayoría de los casos es O(logN) y el tiempo para el peor caso es O(N).





Para hacer una operación para **encontrar el elemento más pequeño**, comenzamos en la raíz y vamos bifurcando repetidamente a la izquierda hasta que deje de haber un hijo izquierdo. El punto en que nos detenemos es el elemento más pequeño. Para hacer una operación para **encontrar el elemento más grande** es similar, ya que que bifurca hacia la derecha. El coste de las operaciones es proporcional al número de nodos en el camino de la búsqueda. El coste tiende a ser logarítmico.

La **eliminación** de elementos es la más costosa. Tiene el problema de que la eliminación de un nodo puede dejar desconectadas parte del árbol y en ese caso se tiene que volver a asociar siguiendo la propiedad del orden de un árbol de busqueda binaria.

Eliminacion de elementos (Descripción en lenguaje natural):

* Si el nodo es una hoja se puede eliminar directamente ya que no deja partes del árbol desconectadas
* Si el nodo tiene un solo hijo, se ajusta el enlace para apuntar desde el padre del nodo hacia el hijo del nodo, eliminando el nodo
* Si el nodo es la raíz, se elimina directamente
* Si un nodo tiene dos hijos, se reemplaza el elemento del nodo por el elemento más pequeño del subárbol derecho. Esto es posible porque el nodo mínimo de un árbol no tiene un hijo izquierdo.