

Árboles Binarios de Búsqueda

Análisis de Algoritmos

Semestre 2, año 2019

¿Por que es importante buscar?

- Tenemos grandes cantidades de datos, que están en constante crecimiento.
- La Web pesa aproximadamente 8ZB ($8ZB = 10^{21} \text{ Byte}$).

Tablas de Símbolos

- Asocia pares (llave, valor).
- Vector caso particular (llave = índice, valor = contenido).

Ejemplos:

Aplicación	Propósito	Llave	Valor
Diccionario	Buscar un término.	Palabra.	Definición.
Índice de Libro	Buscar una página.	Palabra.	Lista de páginas.
Cuentas Bancarias	Procesar una transacción.	Número de cuenta.	Detalles de la transacción.
Búsqueda en la Web	Buscar páginas Web.	Palabras claves (Keywords).	Lista de páginas

Estructuras de Datos

- Árboles Binarios de Búsqueda.
- Árboles Binarios de Búsqueda Balanceado.
- Árbol 2-3.
- Árbol Rojo-Negro.
- Heap.
- Tabla Hash.

Pros y Contras de diferentes estructuras de datos

Estructura	Pros	Contras
Lista enlazada	Buena para tablas de símbolos pequeñas.	Malo para tablas de símbolos grandes.
Arreglo ordenado	Óptimo en búsqueda y espacio.	Inserción lenta.
Árbol Binario de Búsqueda (BST)	Facil de implementar, inserción óptima.	Sin garantía de espacio (referencias).
Árbol Rojo-Negro	Óptimo en búsqueda e inserción.	Uso de espacio (referencias).
Tabla Hash	Búsqueda e inserción rápida.	Espacio y cálculo del valor hash.

Árboles Binarios de Búsqueda

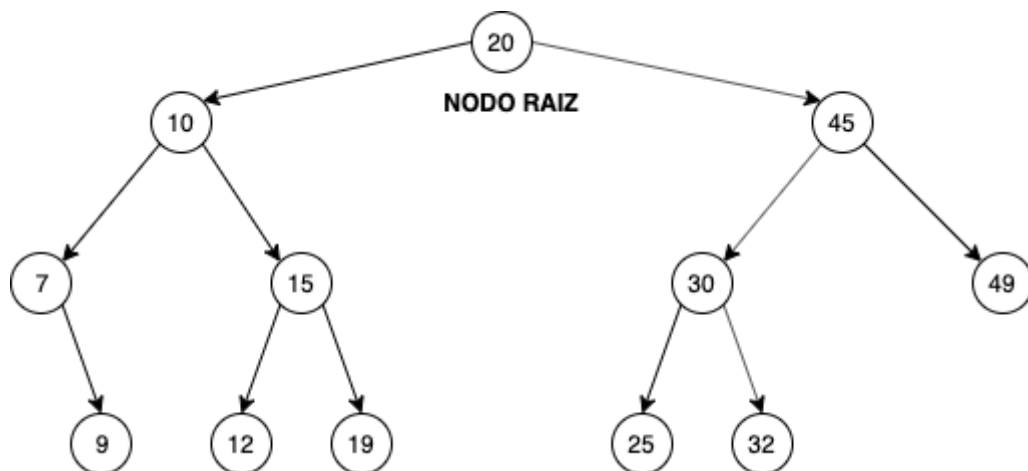
- Árbol binario:
 - Apuntado por solo un nodo (**nodo padre**).
 - Apunta como máximo a otros dos nodos (**nodos hijos**).
- Sub-árbol izquierdo: elementos menores.
- Sub-árbol derecho: elementos mayores.

El Árbol Binario de Búsqueda (BST, *Binary Search Tree*), es un tipo particular de árbol binario que satisface las condiciones básicas de estos, es decir, cada elemento es apuntado solo por un nodo, llamado **nodo padre** y apunta como máximo a otros dos nodos, llamados **nodos hijos**. De manera genérica, podemos decir que cada puntero apunta a un árbol binario.

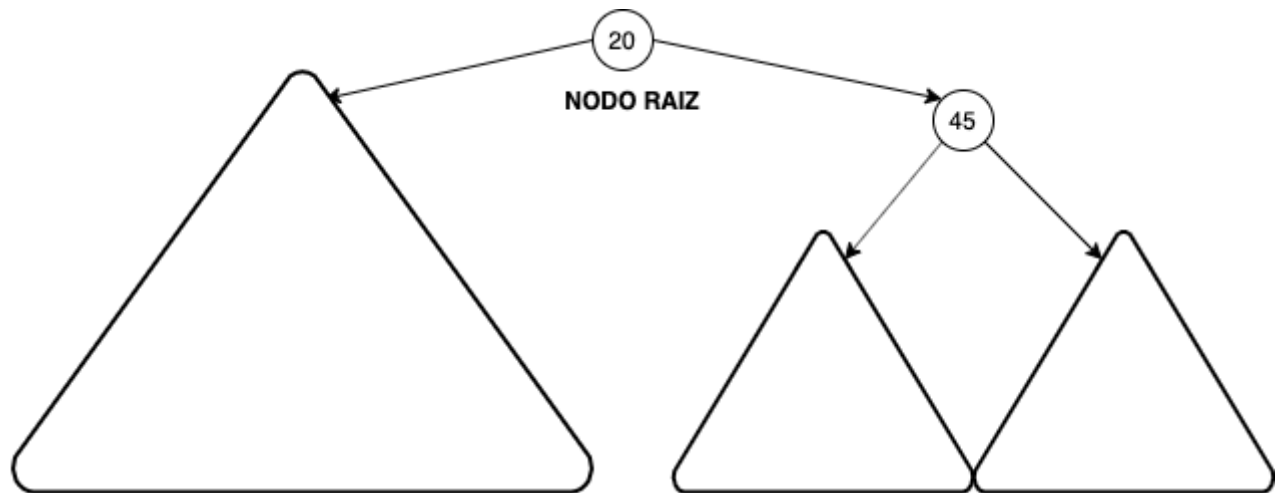
En el BST además se cumple la condición que: las llaves del sub-árbol izquierdo son menores que la llave de su **nodo padre** y las llaves del sub-árbol derecho son mayores que la llave de su **nodo padre**. La llave es dependiente del criterio de búsqueda que se establezca para cada nodo, para un mismo set de datos se pueden generar tantos BST diferentes como criterios existan.

Ejemplo: Si deseamos buscar alumnos, podemos ordenarlos (llaves) por RUT, o por apellido, o por número de matrícula, etc. y cada criterio generará un BST diferente.

Árboles Binarios de Búsqueda



Árboles Binarios de Búsqueda



Nodo árbol binario

Nodo
+ llave: int + valor: int + hijoIzq: Nodo + hijoDer: Nodo
+ __init__(int) + addChild(Nodo)

```
In [1]: class Nodo:
        def __init__(self, llave, valor):
            self.llave=llave
            self.valor=valor
            self.hijoIzq=None
            self.hijoDer=None

        def addChild(self, new_nodo):
            pass
```

```
In [2]: # Funciones complementarias, no se ven en las slides

# Indica el formato de retorno cuando se invoca str()
def __str__(self):
    return "Tree[%i]=%i"%(self.llave,self.valor)

Nodo.__str__=__str__

# Imprimir el arbol visual-friendly
def printTree(self, lvl=0, whoami="root"):
    print(" "*5*lvl+whoami+": "+str(self))
    if(self.hijoIzq != None):
        self.hijoIzq.printTree(lvl+1, "izq")
    if(self.hijoDer != None):
        self.hijoDer.printTree(lvl+1, "der")

Nodo.printTree=printTree
```

Búsqueda e Inserción

- Búsqueda:
 - Condiciones de término.
 - **HIT**: encuentra el nodo.
 - **MISS**: llega a un enlace vacío.
- Inserción:
 - Condiciones de inserción.
 - Cuando se hace **HIT** el valor es reemplazado.
 - Cuando se hace **MISS** la llave y el valor crean un nuevo nodo en esa posición.

Como vimos previamente, el principio de orden es parte estructural de un BST, por esto el recorrido para una búsqueda esta guiado por este principio. De esta manera, si la llave buscada es menor que la del nodo actual, el recorrido continúa por el hijo izquierdo, y si es mayor por el hijo derecho. Si en algún momento encontramos la llave buscada, diremos que hemos echo **HIT** con la llave. En caso contrario, si llegamos hasta la base del árbol, encontrandonos con una rama nula, diremos que hemos echo **MISS** con la llave.

La inserción recibe una llave y un valor comportandose de manera similar, pero con un proposito diferente. En este caso si hacemos un **HIT**, corresponde reemplazar el valor del nodo actual por el nuevo valor otorgado. Un **MISS** en el caso de la inserción nos indica que hemos encontrado el punto donde un nuevo nodo debe ser creado.

Mejor caso

El mejor caso de un BST se produce cuando como resultado obtenemos un *árbol perfectamente balanceado*. En este caso las operaciones de inserción y búsqueda tienen tiempos asociados de $O(\lg N)$. Esto es importante, porque muchas de las mejoras al BST tradicional, buscan mantener el balance en el árbol, para poder asegurar tiempos. Ejemplos de esto lo vemos en los árboles AVL, 2-3, Rojo-Negro, etc.

Buscar una llave

```
In [3]: def buscar(self, llave_buscada):
        if(self.llave == llave_buscada):
            return str(self)
        elif(llave_buscada < self.llave and self.hijoIzq == None):
            return False
        elif(llave_buscada < self.llave):
            return self.hijoIzq.buscar(llave_buscada)
        elif(llave_buscada > self.llave and self.hijoDer == None):
            return False
        elif(llave_buscada > self.llave):
            return self.hijoDer.buscar(llave_buscada)

        Nodo.buscar = buscar
```

Agregar nuevo nodo

```
In [4]: def addChild(self, new_nodo):
        if(self.llave == new_nodo.llave):
            self.valor = new_nodo.valor
            del new_nodo
            return False
        elif(new_nodo.llave < self.llave and self.hijoIzq == None):
            self.hijoIzq = new_nodo
            return True
        elif(new_nodo.llave < self.llave):
            return self.hijoIzq.addChild(new_nodo)
        elif(new_nodo.llave > self.llave and self.hijoDer == None):
            self.hijoDer = new_nodo
            return True
        elif(new_nodo.llave > self.llave):
            return self.hijoDer.addChild(new_nodo)

Nodo.addChild = addChild
```

Demo

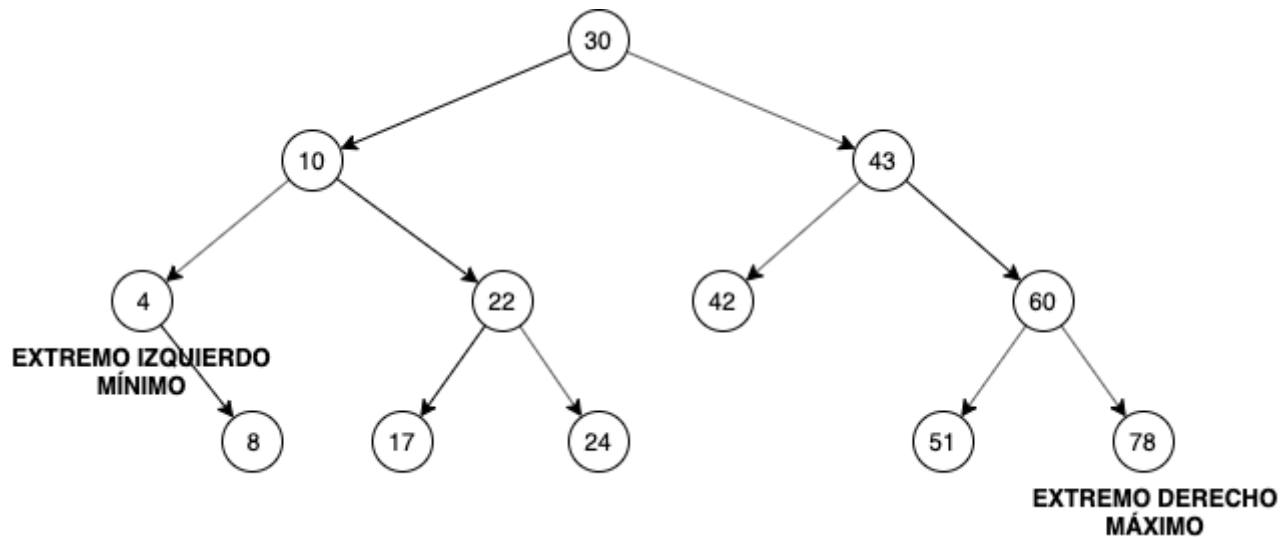
```
In [ ]:
```

Operaciones básicas de un BST

- Mínimo y máximo
- Piso y techo (floor and ceiling)
- Borrar mínimo y máximo (delete min and delete max)
- Borrar llave

Mínimo y máximo

En un BST el elemento más a la izquierda es el elemento *mínimo*. Este elemento no necesariamente se encuentra en el último nivel, ya que por construcción si giramos hacia la derecha, los valores van creciendo. Análogamente, el elemento en el extremo derecho es el *máximo*.



Mínimo

```

In [5]: def minimo(self):
        if(self.hijoIzq != None):
            return self.hijoIzq.minimo()
        else:
            return "min:"+str(self)

Nodo.minimo = minimo

```

Máximo

```

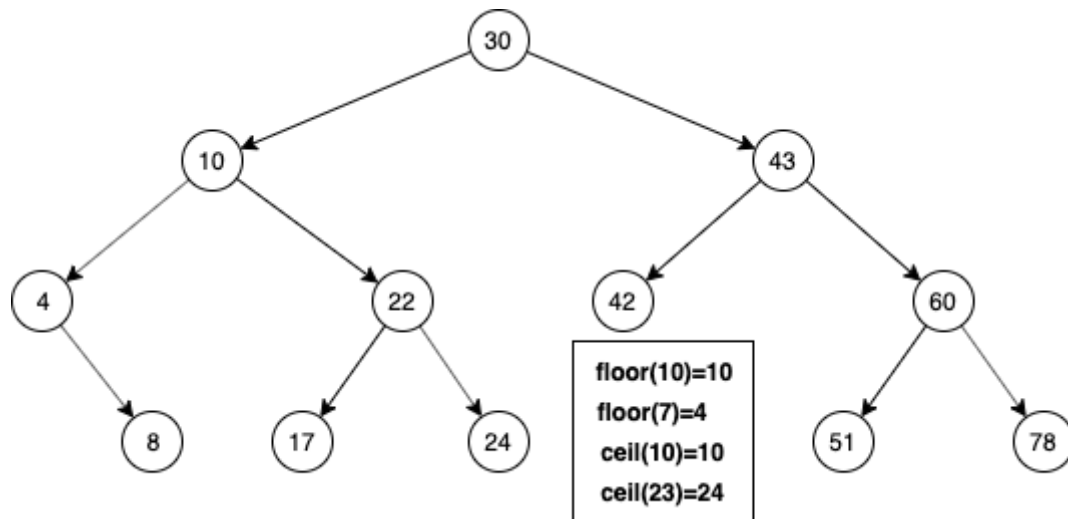
In [6]: def maximo(self):
        if(self.hijoDer != None):
            return self.hijoDer.maximo()
        else:
            return "max:"+str(self)

Nodo.maximo = maximo

```

Piso y techo

La función piso (floor) recorre el árbol buscando la llave entregada, si la encuentra retorna el valor, sino retorna el valor de la llave inmediatamente inferior. De manera análoga, la función techo (ceil) retorna el valor si encuentra la llave entregada, sino retorna el valor de la llave inmediatamente superior.



Piso (floor)

```

In [7]: def floor(self, llave_buscada):
        if(self.llave == llave_buscada):
            return self.llave
        elif(llave_buscada < self.llave and self.hijoIzq == None):
            return False if self.llave > llave_buscada else self.llave
        elif(llave_buscada < self.llave):
            prev_llave = self.hijoIzq.floor(llave_buscada)
            prev_llave = self.llave if prev_llave == False else prev_llave
            return False if prev_llave > llave_buscada else prev_llave
        elif(llave_buscada > self.llave and self.hijoDer == None):
            return False if self.llave > llave_buscada else self.llave
        elif(llave_buscada > self.llave):
            prev_llave = self.hijoDer.floor(llave_buscada)
            prev_llave = self.llave if prev_llave == False else prev_llave
            return False if prev_llave > llave_buscada else prev_llave

        Nodo.floor = floor
  
```

Techo (ceil)


```
In [8]: def ceil(self, llave_buscada):
        if(self.llave == llave_buscada):
            return self.llave
        elif(llave_buscada < self.llave and self.hijoIzq == None):
            return False if self.llave < llave_buscada else self.llave
        elif(llave_buscada < self.llave):
            prev_llave = self.hijoIzq.ceil(llave_buscada)
            prev_llave = self.llave if prev_llave == False else prev_llave
            return False if prev_llave < llave_buscada else prev_llave
        elif(llave_buscada > self.llave and self.hijoDer == None):
            return False if self.llave < llave_buscada else self.llave
        elif(llave_buscada > self.llave):
            prev_llave = self.hijoDer.ceil(llave_buscada)
            prev_llave = self.llave if prev_llave == False else prev_llave
            return False if prev_llave < llave_buscada else prev_llave

        Nodo.ceil = ceil
```

Demo

In []:

Borrar (delete)

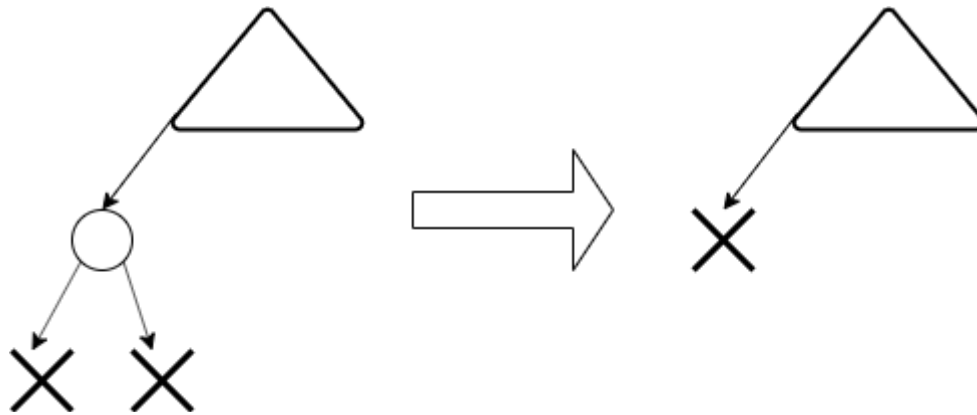
En esta sección consideramos tanto el borrado general de una llave, como el de los valores mínimos y máximos que son casos particulares del primero. El borrado de una llave, recibe la llave a borrar y elimina ese nodo si lo encuentra, sino no hace nada. El borrado del mínimo y el máximo, se posiciona en estos valores y los borra, la diferencia con el borrado general es que este no tiene un caso contrario, pues siempre se encontrará un valor mínimo y uno máximo dentro del árbol.

Otro punto importante a considerar es que un nodo no puede ser simplemente borrado sin revisar a sus hijos, pues estos no pueden ser eliminados también, si este fuera el caso, no sería un borrado sino una *poda* de un sub-arbol. Para ello se consideran ciertos casos que nos permiten lidiar con este problema.

- Tipos de borrado
 - Borrar llave
 - Borrar mínimo/máximo
- Casos de borrado
 - Nodo sin hijos
 - Nodo con 1 hijo
 - Nodo con 2 hijos

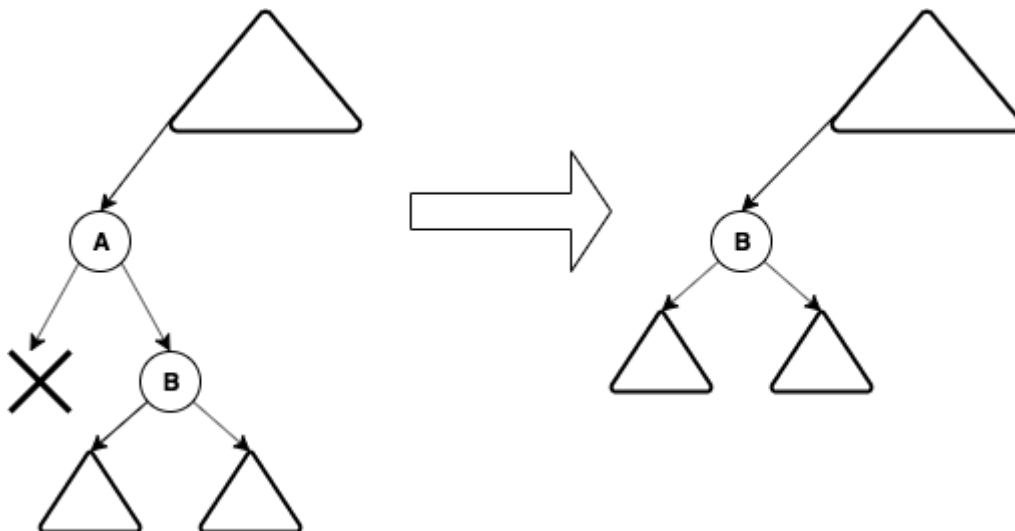
Borrado de un nodo sin hijos

Es el caso mas sencillo, en este caso, el nodo simplemente se elimina del árbol y el enlace del padre del nodo eliminado se deja nulo.



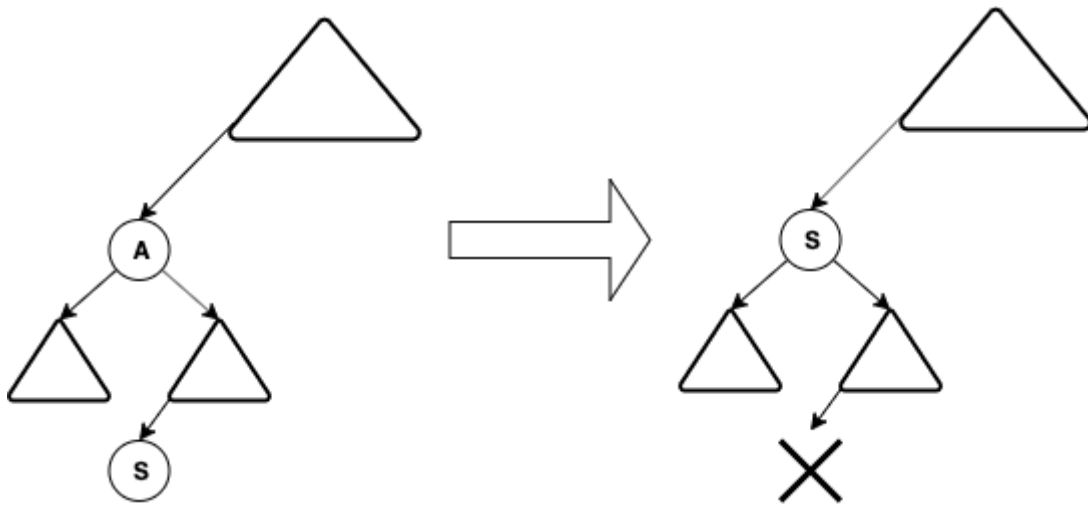
Borrado de un nodo con 1 hijo

Por construcción, permutar el único hijo (ya sea izquierdo o derecho) por la raíz de su sub-árbol mantendrá la restricción básica de un BST. Por esto, eliminamos el nodo solicitado, no sin antes guardar la referencia a su hijo para luego posicionarlo en el lugar del nodo borrado.



Borrado de un nodo con 2 hijos

Para este caso, buscamos el sucesor (S) del nodo a eliminar (A). El nodo S, es el *nodo mínimo* del hijo derecho. Este nodo S no afecta las restricciones ya que es el más cercano a A.



Tarea

Para el código presentado, implementar las funciones borrar mínimo (deleteMin), borrar máximo (deleteMax) y borrar llave (deleteKey). Para ello modifique las funciones auxiliares proporcionadas a su conveniencia, sin que ellas afecten el funcionamiento del resto del programa.





<https://github.com/moyarzunsil/AAIgo.git> (<https://github.com/moyarzunsil/AAIgo.git>)



moyarzunsil@unap.cl (<mailto:moyarzunsil@unap.cl>)



<https://twitter.com/moyarzunsilva> (<https://twitter.com/moyarzunsilva>)