Árboles binarios de búsqueda

Clase 06

IIC 2133 - Sección 2

Prof. Mario Droguett

Sumario

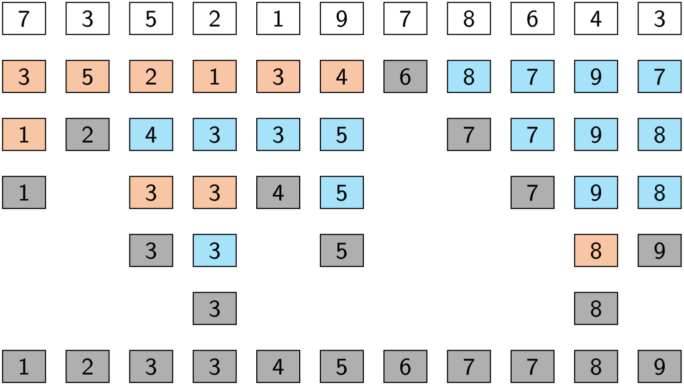
Repaso: `Quicksort`

input : Secuencia $A[0, \ldots, n-1]$, índices i, f
output: $\varnothing$

```
QuickSort (A, i, f):
    if i ≤ f :
        p ← Partition(A, i, f)
        Quicksort(A, i, p − 1)
        Quicksort(A, p + 1, f)
```

¿Cuál es la complejidad de `Quicksort` en los distintos casos?

Repaso: `Quicksort`



Repaso: Complejidad de tiempo de `Quicksort`

Tal como con `Median`, la complejidad depende de la elección del pivote: es **probabilística**

Mejor caso

- `Partition` genera sub-secuencias del *mismo* tamaño
- Ecuación de recurrencia $T(n) = n + 2T(n/2)$ (como en `MergeSort`)
- Complejidad $\mathcal{O}(n \log(n))$

Peor caso

- `Partition` genera sub-secuencias de tamaño 0 y $n - 1$
- Ecuación de recurrencia $T(n) = n + T(n - 1)$
- Complejidad $\mathcal{O}(n^2)$

¿Qué hay del caso promedio?

Repaso: Caso promedio de `Quicksort`

En nuestro análisis de **caso promedio** supondremos que el pivote queda en cualquiera de las n posiciones con igual probabilidad

```
Partition (A, i, f):
1     x ← índice aleatorio en
        {i, ..., f};   p ← A[x]
2     A[x] ⇄ A[f]
3     j ← i
4     for k = i ... f − 1 :
5         if A[k] < p :
6             A[j] ⇄ A[k]
7             j ← j + 1
8     A[j] ⇄ A[f]
9     return j
```

Contaremos el número de comparaciones de la línea 5 de `Partition`, pues mide cuántas iteraciones debe hacer este método

Definimos

$$C(n) := \begin{array}{l} \#\text{ comp. } A[k] < p \\ \text{en } \texttt{Quicksort} \end{array}$$

Encontraremos una ecuación de recurrencia para $C(n)$

Repaso: Caso promedio de `Quicksort`

`Partition` (A, i, f):

$x \leftarrow$ índice aleatorio en $\{i, \ldots, f\}$; $\quad p \leftarrow A[x]$

$A[x] \rightleftarrows A[f]$

$j \leftarrow i$

for $k = i \ldots f - 1$ **:**

    if $A[k] < p$ **:**

        $A[j] \rightleftarrows A[k]$

        $j \leftarrow j + 1$

$A[j] \rightleftarrows A[f]$

return j

- En la primera llamada se hacen $n - 1$ comparaciones, pues $i = 0$ y $f = n - 1$
- Supongamos que `Partition` termina retornando q, i.e. deja al pivote en $A[q]$
- Las llamadas recursivas de `Quicksort` se harán sobre los
  - q elementos a la izq
  - $n - q - 1$ elementos a la der
- Los llamados recursivos aportan

$$C(q) + C(n - q - 1)$$

Repaso: Caso promedio de `Quicksort`

Con lo anterior, las comparaciones ecuando el pivote queda en $A[q]$ son

$$n - 1 + C(q) + C(n - q - 1)$$

Para ver el caso promedio, ponderamos para todos los q posibles obteniendo

$$C(n) \;=\; n - 1 + \frac{1}{n}\sum_{q=0}^{n-1}\left(C(q) + C(n - q - 1)\right)$$

Además, notando que

$$\sum_{q=0}^{n-1} C(n - q - 1) \;=\; C(n-1) + C(n-2) + \cdots + C(0) \;=\; \sum_{q=0}^{n-1} C(q)$$

obtenemos la regla simplificada

$$C(n) \;=\; n - 1 + \frac{2}{n}\sum_{q=0}^{n-1} C(q)$$

Repaso: Caso promedio de `Quicksort`

Para la ecuación de recurrencia

$$C(n) = n - 1 + \frac{2}{n} \sum_{q=0}^{n-1} C(q)$$

tenemos dos casos base

- $C(0) = 0$, pues corresponde al caso base de `Quicksort`
- $C(1) = 0$, pues `Partition` no itera si hay un solo elemento

Repaso: Caso promedio de `Quicksort`

Amplificamos por n la recurrencia y la reescribimos para $n-1$

$$
\begin{aligned}
nC(n) &= n(n-1) + 2\sum_{q=0}^{n-1} C(q) \\
(n-1)C(n-1) &= (n-1)(n-2) + 2\sum_{q=0}^{n-2} C(q)
\end{aligned}
$$

Restando ambas ecuaciones obtenemos

$$
nC(n) = (n+1)C(n-1) + 2n - 2
$$

Dividimos por $n(n+1)$ y comenzamos una serie de inecuaciones

$$
\begin{aligned}
\frac{C(n)}{n+1} &= \frac{C(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \\
&\leq \frac{C(n-1)}{n} + \frac{2}{n+1}
\end{aligned}
$$

Repaso: Caso promedio de Quicksort

$$
\begin{aligned}
\frac{C(n)}{n+1} &\leq \frac{C(n-1)}{n} + \frac{2}{n+1} \\
&\leq \frac{C(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\leq \frac{C(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\dots \\
&\leq \frac{C(1)}{2} + \sum_{k=2}^{n} \frac{2}{k+1} \\
&\leq \sum_{k=1}^{n} \frac{2}{k} \leq \int_{1}^{n} \frac{2}{x} dx = 2\log(n)
\end{aligned}
$$

Concluimos que una cota superior para el número de comparaciones es

$$C(n) \leq 2(n+1)\log(n)$$

`Quicksort` es $\mathcal{O}(n\log(n))$ en el caso promedio

Mejoras en `Quicksort`

- Para sub-secuencias pequeñas (e.g. $n \leq 20$) podemos usar `InsertionSort`
  - A pesar de no tener una complejidad mejor
  - Eso es solo cuando hablamos asintóticamente
  - En la práctica, en instancias pequeñas tiene mejor desempeño
- Usar la mediana de tres elementos como pivote
  - *Informar* la elección de pivote
  - Dado A, escogemos 3 elementos $A[k_1], A[k_2], A[k_3]$
  - En $\mathcal{O}(1)$ encontramos la mediana entre ellos
- Particionar la secuencia en 3 sub-secuencias: menores, iguales y mayores
  - Mejora para caso con datos repetidos
  - Evita que `Partition` particione innecesariamente sub-secuencias en que todos los valores son iguales

Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Quick Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Heap Sort	?	?	?	?

El Misterio de EDD

- Hasta aquí, somos capaces de ordenar una secuencia usando diferentes algoritmos

- En determinados casos, alguno de ellos puede ser más adecuado
  - Por características del input
  - Por requisitos de memoria y tiempo

¿Cuál era el problema que motivó esta primera parte?

El Misterio de EDD

Dada una secuencia desordenada, nos interesa **buscar** un elemento

¿ Zallen Misterio $\in$

Apellido	Nombre
Alen	Misterio
Misterio	Misterio
Zalen	Berenice
Gonzalópez	D
Turing	Alan
Misterio	Yadran
Zeta	Hache
Ararán	Jota
Alenn	Cristina
...	...

pág. 1/376

?

El Misterio de EDD

Escogemos algún **algoritmo de ordenación**

```
QuickSort (A, i, f):
    if i ≤ f :
        p ← Partition(A, i, f)
        Quicksort(A, i, p − 1)
        Quicksort(A, p + 1, f)
```

El Misterio de EDD

Obtenemos la secuencia ordenada

¿ Zallen Misterio $\in$

Apellido	Nombre
Abarca	Yadran
Abusleme	Nicole
Arenas	Camila
Arenas	D
Bañados	Richard
Beterraga	Brócoli
Blanco	Ximena
Brahms	Johannes
Castillo	Raquel
...	...
	pág. 1/376

?

El Misterio de EDD

Usamos algún **algoritmo de búsqueda** para encontrar el elemento

```
BinarySearch (A, x, i, f):
    if f < i : return −1
    m ← ⌊(i + f) / 2⌋
    if A[m] = x : return m
    if A[m] > x :
        return BinarySearch (A, x, i, m − 1)
    return BinarySearch (A, x, m + 1, f)
```

¿Habrá otra forma de combinar **ordenación y búsqueda**?

Objetivos de la clase

- Comprender la noción de diccionario y qué operaciones soporta
- Conocer los árboles binarios de búsqueda
- Comprender las propiedades básicas de un ABB
- Identificar la utilidad de los ABB
- Comprender los algoritmos que implementan sus operaciones básicas

Sumario

Una nueva estructura

- Construiremos una estructura con nuevas características
- Dada una **llave o clave**, queremos asociarle un **valor**
- Si la llave no está en la EDD, lo sabemos de forma eficiente
- Si la llave está en la EDD, también lo sabemos de forma eficiente
- Podemos agregar, modificar y eliminar **pares llave-valor** de forma eficiente

Diccionarios

Definición

Un **diccionario** es una estructura de datos con las siguientes operaciones

- **Asociar** un valor a una llave
- **Actualizar** el valor asociado a una llave
- **Obtener** el valor asociado a una llave
- En ciertos casos, **eliminar** de la estructura una asociación llave-valor

Ejemplos

- RUT como llave y nombre como valor
- RUT como llave y (nombre, apellido, edad,...) como valor

Diccionarios

Uno de los objetivos centrales de un diccionario es facilitar la **búsqueda**

- Primero buscamos la llave (si está o no está)
- Buscar = *buscar eficientemente*

¿Cómo almacenamos las llaves para lograr búsqueda eficiente?

Hasta ahora tenemos dos opciones: **arreglos** y **listas**... ¿cumplen nuestro objetivo?

Limitaciones de arreglos y listas

En una **listas ligada** de llaves

- No tenemos **acceso por índice** de forma eficiente
- La búsqueda, incluso en el caso ordenado, es $\mathcal{O}(n)$

En un **arreglo** de llaves

- Hay **acceso por índice** en $\mathcal{O}(1)$
- La búsqueda en general es $\mathcal{O}(n)$
- Para el caso ordenado, podemos lograrla en $\mathcal{O}(\log(n))$

¿Qué punto débil tienen los arreglos comparados con las listas?

Limitaciones de arreglos y listas

En una **listas ligada** de llaves

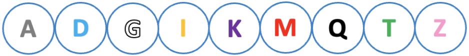
- Para insertar solo necesitamos reasignar (pocos) punteros
- La inserción de un nuevo elemento es $\mathcal{O}(1)$

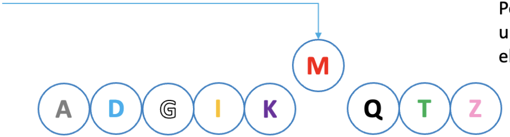
En un **arreglo** de llaves

- Insertar un elemento puede gatillar un desplazamiento de datos
- En promedio, la inserción es $\mathcal{O}(n)$

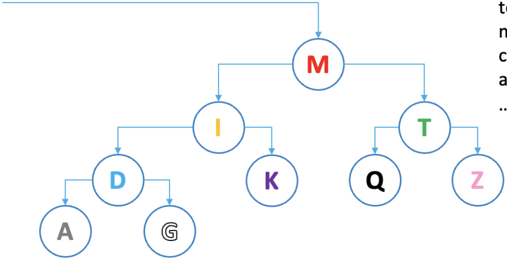
¿Podemos construir una EDD con buen desempeño en ambas operaciones?

Modifiquemos las listas





Podemos tener un puntero a un elemento más o menos en el centro de la lista



... y ese elemento puede tener punteros a elementos más o menos en el centro de cada una de las dos sublistas, a su izquierda y a su derecha;
... y así recursivamente

Árboles binarios de búsqueda

Definición

Un **árbol binario de búsqueda (ABB)** es una estructura de datos que almacena **pares (llave, valor)** asociándolos mediante punteros según una estrategia recursiva

1. Un ABB tiene una **nodo** que contiene una tupla (llave, valor)
2. El nodo puede tener hasta dos ABB's asociados mediante punteros
   - Hijo izquierdo
   - Hijo derecho

y que además, satisface la **propiedad ABB**: las llaves menores que la llave del nodo están en el sub-árbol izquierdo, y las llaves mayores, en el sub-árbol derecho.

La estrategia **dividir para conquistar** aplicada a una EDD

Árboles binarios

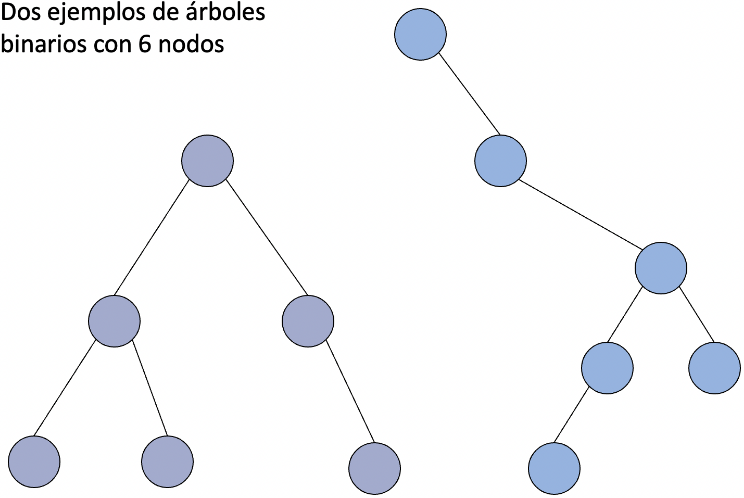
Un árbol binario (de búsqueda o no) cumple que

- Cada nodo x es apuntado a lo más por un nodo **padre** $x.p$
- El nodo sin padre se conoce como **raíz**
- Cada nodo x tiene hasta dos punteros que (*apuntan*) a sub-árboles
  - $x.left$ es un puntero al hijo izquierdo
  - $x.right$ es un puntero al hijo derecho
- Un nodo sin punteros, i.e. sin hijos, se conoce como **hoja**

¿Necesariamente un árbol binario tiene
nodos con la misma cantidad de hijos?

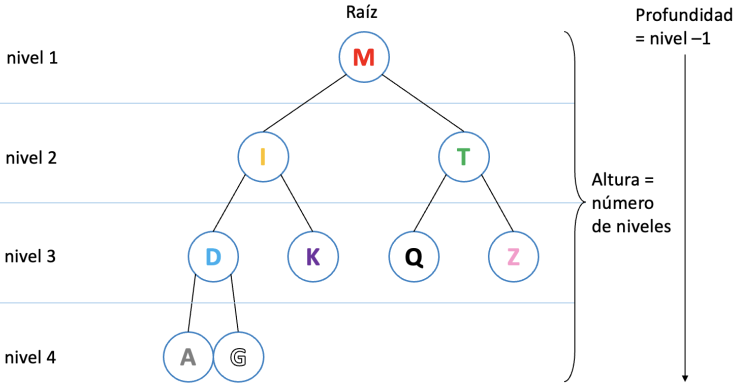
Árboles binarios

Dos ejemplos de árboles binarios con 6 nodos



Anatomía de un árbol binario

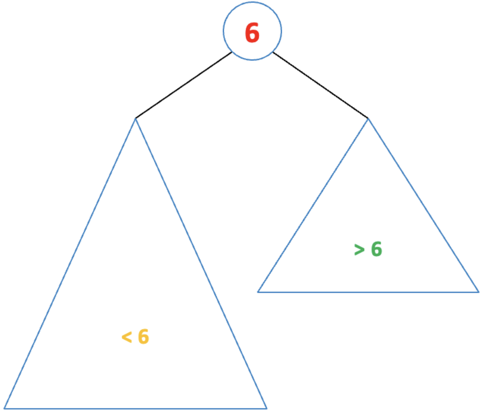
Por simplicidad, representaremos solo las llaves de los árboles



Notemos que ir de una hoja a la raíz toma tiempo $\mathcal{O}(\textit{profundidad})$

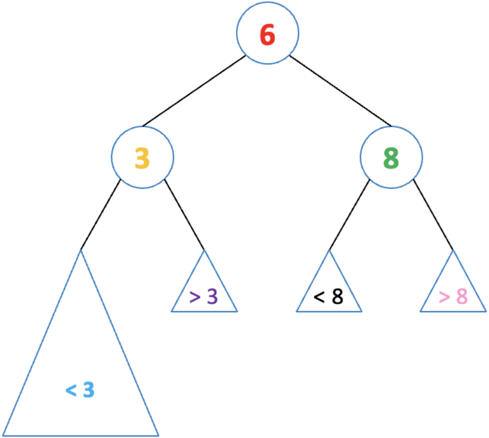
Anatomía de un árbol binario

No olvidemos la estructura recursiva: los hijos son ABB's



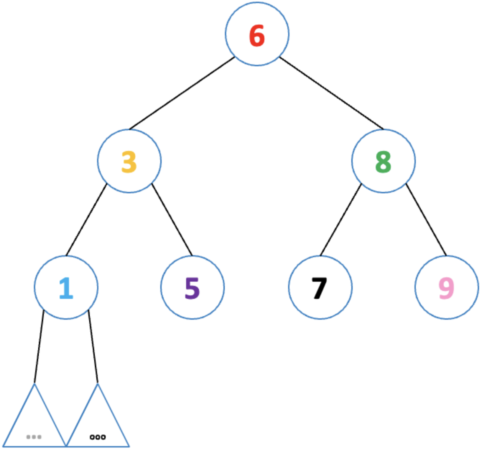
Anatomía de un árbol binario

No olvidemos la estructura recursiva: los hijos son ABB's



Anatomía de un árbol binario

No olvidemos la estructura recursiva: los hijos son ABB's



Sumario

Operaciones de un ABB

- Recordemos nuestro objetivo al definir esta nueva estructura
- Queremos búsqueda rápida
- Para esto buscamos lograr un diccionario
- Queremos garantizar operaciones eficientes para búsqueda, inserción, modificación y eliminación
- A través de la definición concreta de estas operaciones para un ABB mostraremos que un ABB nos sirve como **diccionario**

Operaciones de un ABB

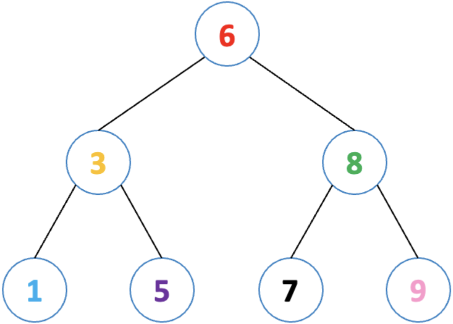
Completamos las definiciones de atributos de un **nodo** x en un ABB

- $x.key$ es la llave del nodo
- $x.value$ es su valor
- $x.left$ el puntero a su hijo izquierdo
- $x.right$ el puntero a su hijo derecho

En general no incluiremos $x.value$ en los algoritmos.
Solo será un espacio de almacenamiento

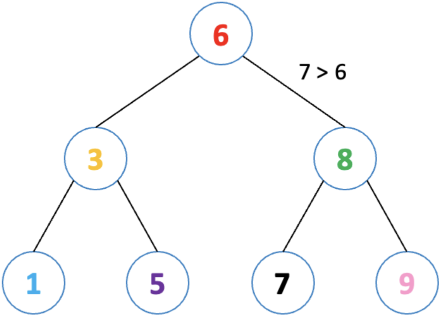
Ejemplo de búsqueda

Nos interesa encontrar el nodo con llave 7. Solo conocemos el nodo raíz



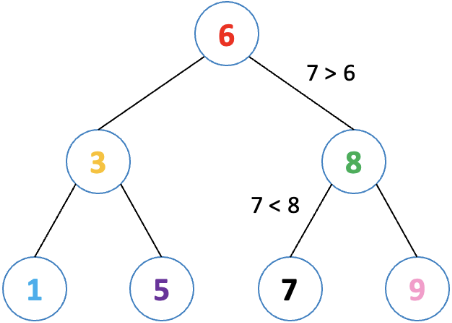
Ejemplo de búsqueda

Comparamos con la llave raíz y sabemos que, si está, debe estarlo en el sub-árbol derecho



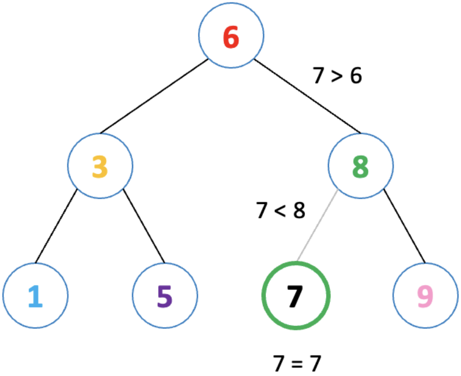
Ejemplo de búsqueda

Recursivamente, repetimos para la raíz del sub-árbol detectado y determinamos que hay que revisar el sub-árbol izquierdo



Ejemplo de búsqueda

Al revisar la raíz de este nuevo sub-árbol, encontramos la llave buscada



Operación de búsqueda

Proponemos el siguiente algoritmo de búsqueda en ABB's

input : Árbol binario de búsqueda A, llave buscada k
output: Árbol binario de búsqueda, o $\varnothing$ si no se encuentra

`Search` (A, k)**:**
    if $A = \varnothing \lor A.key = k$ **:**
        return A
    if $A = \varnothing \lor A.key = k$ **:**
        return A
    if $k < A.key$ **:**
        return `Search` $(A.left, k)$
    return `Search` $(A.right, k)$

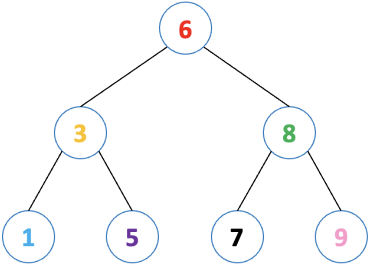
El llamado inicial es `Search`$(root, k)$ para la raíz *root* del árbol

Operaciones para modificar un ABB

- **Insertar** un nodo con una nueva llave produce un cambio en la estructura del árbol
- De igual forma, **eliminar** un nodo también lo hace
- Ambas operaciones pueden afectar la **propiedad ABB**
- Nuestra propuesta de algoritmos para estar operaciones debe **restaurar la propiedad ABB** si se incumple

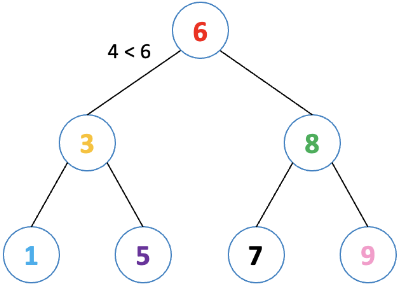
Ejemplo de inserción

Insertemos un nodo con llave 4



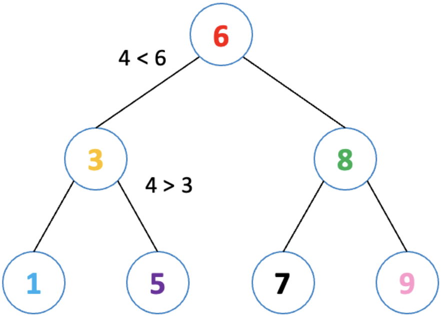
Ejemplo de inserción

Comparamos llaves para determinar en qué posición debe ser insertado

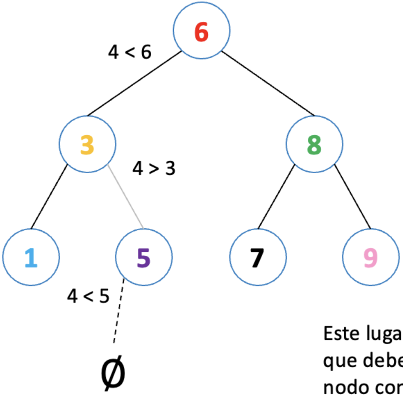


Ejemplo de inserción

Comparamos llaves para determinar en qué posición debe ser insertado



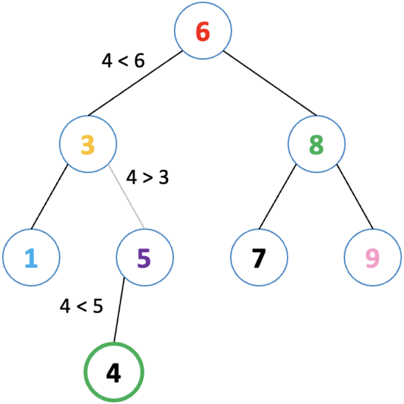
Ejemplo de inserción



Este lugar vacío es el lugar en que debería haber estado el nodo con clave 4 si hubiera estado en el árbol

Ejemplo de inserción

Dado que, para $x.key = 5$ se tiene $x.left = \varnothing$, lo reemplazamos con la llave indicada



Operación de inserción

Proponemos el siguiente algoritmo de inserción de valores según llave ABB's

input : Árbol binario de búsqueda A, llave k, valor v
output: $\varnothing$

`Insert` (A, k, v):

$\quad B \leftarrow \texttt{Search}(A, k)$ $\quad \triangleright$ `versión que indica el padre`
$\quad$ **if** $B = \varnothing$ **:**
$\quad\quad B \leftarrow$ nodo vacío
$\quad\quad B.key \leftarrow k$
$\quad\quad$ Conectar B al padre en la posición adecuada
$\quad B.value \leftarrow v$

Este algoritmo mantiene la propiedad ABB al insertar

Operación de eliminación

La eliminación es un poco más compleja

Si el nodo a eliminar es hoja o tiene solo un hijo

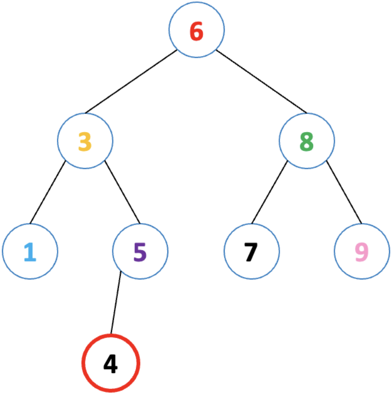
- Lo borramos
- Si tenía un hijo, el hijo lo reemplaza
- Es claro que se mantiene la propiedad ABB

En caso contrario...

¿Se puede reemplazar por otro árbol?

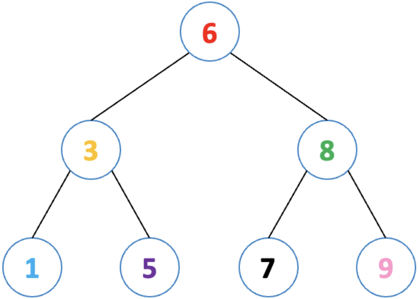
Ejemplo de eliminación

Si queremos eliminar el nodo con llave 4



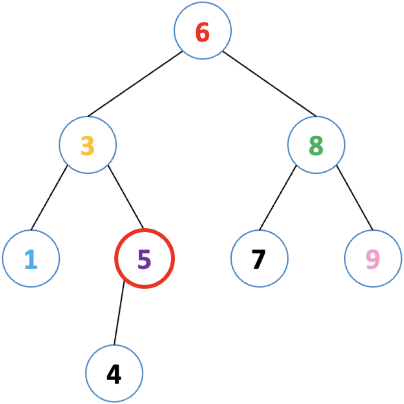
Ejemplo de eliminación

Simplemente se elimina y se preserva la propiedad ABB



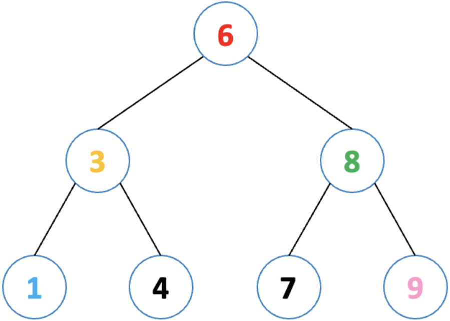
Ejemplo de eliminación

Si queremos eliminar el nodo con llave 5



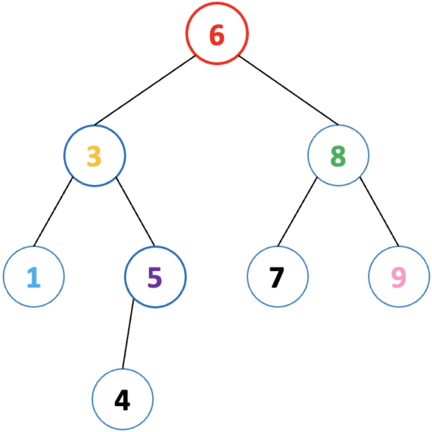
Ejemplo de eliminación

Se reemplaza por su único hijo y se preserva la propiedad ABB



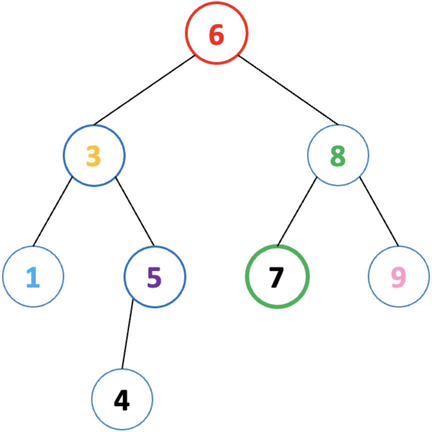
Ejemplo de eliminación

Si queremos eliminar el nodo con llave 6, estamos en problemas



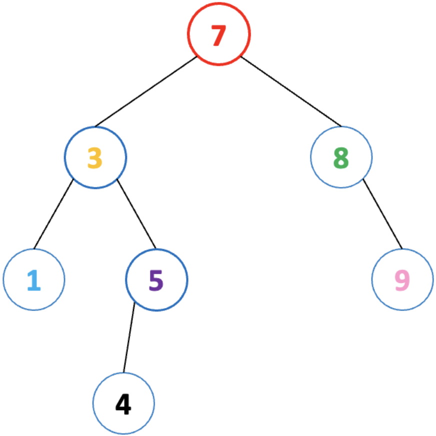
Ejemplo de eliminación

Podemos reemplazarlo por el nodo con llave 7 (**su sucesor**)



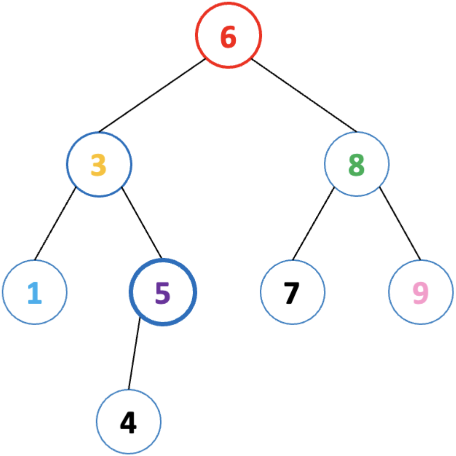
Ejemplo de eliminación

Y dado que no tenía hijos, no hay que hacer más modificaciones



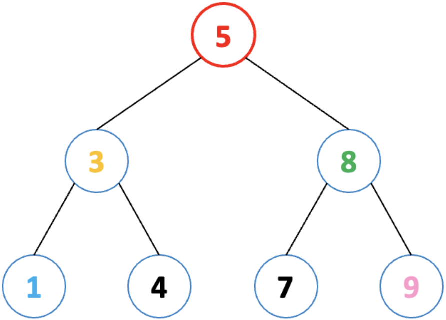
Ejemplo de eliminación

De forma alternativa, podemos reemplazarlo por el nodo con llave 5 (**su antecesor**)



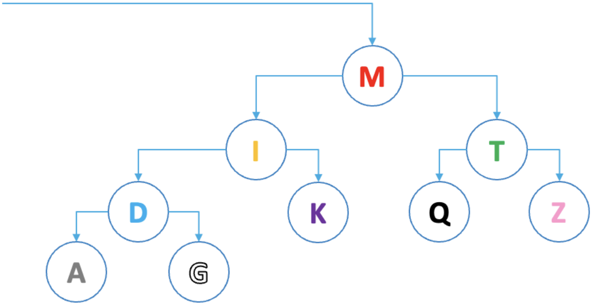
Ejemplo de eliminación

Y reubicamos su hijo con llave 4



Operación de eliminación

Nos interesa encontrar el sucesor/antecesor del nodo extraído




```
Min (A):
    if A.left = ∅ :
        return A
    return Min(A.left)
```

```
Max (A):
    if A.right = ∅ :
        return A
    return Max(A.right)
```

Operación de eliminación

Proponemos el siguiente algoritmo de eliminación que preserva la propiedad ABB

```
Delete (A, k):
    D ← Search(A, k)        ▷ Permite saber el padre de D
    if D ≠ ∅ :
        if D es hoja :  D ← ∅
        elif D tiene un solo hijo H :  D ← H
        else:
            R ← Min(D.right)
            t ← R.right
            D.key ← R.key
            D.value ← R.value
            R ← t
```

Notemos que al borrar un nodo, se debe eliminar la referencia desde su padre

Antecesor y sucesor en general

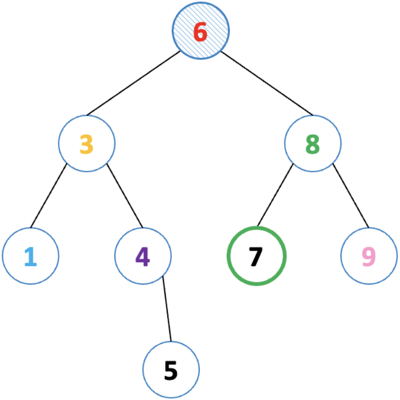
¿Qué tan fácil es deterinar el sucesor y antecesor de un nodo?

Ya tenemos algoritmos recursivos para esto

¿Y si los tuviéramos en una lista ordenados?

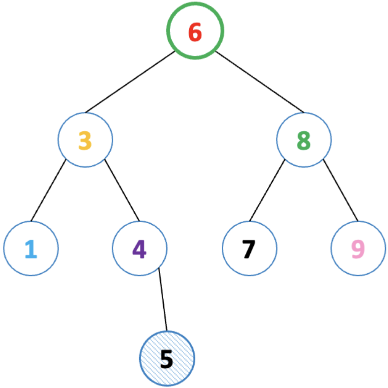
Antecesor y sucesor en general

Ya sabemos que es *fácil* entontrarlos preguntando por la raíz



Antecesor y sucesor en general

Pero ya no tenemos acceso a `Min` y `Max` si preguntamos por un nodo no raíz



Sumario

Ideas al cierre

- Un diccionario es la abstracción de una estructura con eficiencia en inserción y búsqueda
- Los árboles binarios de búsqueda intentan lograr este objetivo
- No hemos demostrado bajo qué circunstancias un ABB logra eficiencia en búsqueda
- Es posible buscar en un ABB usando sus propiedades
- Es posible modificar los ABB para mantener su propiedad característica