… previamente en IIC2133

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	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(1)$

Ordenación en tiempo lineal

Consideremos una A secuencia de n naturales entre 0 y k

- Para todo $a_i \in A$, se tiene que $0 \leq a_i \leq k$
- Notemos que no necesariamente $n = k - 1$
- La secuencia A puede tener elementos repetidos

Propondremos un algoritmo de ordenación que **no compara**

- Para cada dato, contaremos cuántos datos son menores que él
- Esto nos indica la posición final de cada elemento

Si $k \in \mathcal{O}(n)$, entonces este algoritmo será $\Theta(n)$

El algoritmo `CountingSort()`

input : Arreglo $A[0 \dots n-1]$, natural k
output: Arreglo $B[0 \dots n-1]$

`CountingSort` (A, k)**:**

```
    B[0...n−1] ← arreglo vacío de n celdas
    C[0...k] ← arreglo vacío de k+1 celdas
    for i = 0...k :
        C[i] ← 0
    for j = 0...n−1 :
        C[A[j]] ← C[A[j]] + 1
    for p = 1...k :
        C[p] ← C[p] + C[p−1]
    for r = n−1...0 :
        B[C[A[r]] − 1] ← A[r]
        C[A[r]] ← C[A[r]] − 1
    return B
```

RadixSort

El algoritmo `RadixSort` ordena por dígito **menos significativo**

- Ordena por dígito n_{d-1}
- Luego, usando el mismo arreglo, ordena por dígito n_{d-2}, **con un algoritmo estable**
- Luego de ordenar k dígitos, los datos están ordenados si solo miramos el fragmento $n_{d-k} \cdots n_{d-1}$
- Se requieren solo d pasadas para ordenar la secuencia completa

```
RadixSort(A, d):
    for j = 0 ... d − 1 :
        StableSort(A, j)    ▷ algoritmo de ordenación estable por
                              j-ésimo dígito menos significativo
```

Dos implementaciones

Estas ideas tiene dos implementaciones

LSD string sort (*Least Significant Digit*)

- Si todos los strings son del mismo largo (patentes, IP's, teléfonos)
- Funciona bien si el largo es pequeño
- No recursivo

MSD string sort (*Most Significant Digit*)

- Si los strings tienen largo diferente
- Ordenamos con `CountingSort()` por primer caracter
- Recursivamente ordenamos subarreglos correspondientes a cada caracter (excluyendo el primero, que es común en cada subarreglo)
- Como `Quicksort`, puede ordenar de forma independiente
- **Pero** particiona en tantos grupos como valores del primer caracter

Cuidados de MSD

En la ejecución de MSD string sort se debe considerar

- Si un string s_1 es prefijo de otro s_2, s_1 es menor que s_2

$$\text{she} \leq \text{shells}$$

- Pueden usarse diferentes alfabetos
  - binario (2)
  - minúsculas (26)
  - minúsculas + mayúsculas + dígitos (64)
  - ASCII (128)
  - Unicode (65.536)
- Para subarreglos pequeños (e.g. $|A| \leq 10$)
  - cambiar a `InsertionSort` que *sepa* que los primeros k caracteres son iguales

Counting & Radix Sort

- Atributos generales
  - O(n)
- Harold H. Seward
  - July 24, 1930 – June 19, 2012
- Was a computer scientist, engineer, and inventor.
- Seward developed the radix sort and counting sort algorithms in 1954 at MIT.
- Also worked on the Whirlwind Computer and developed instruments that powered the guidance systems for the Apollo spacecraft and Polaris missile.

Árboles binarios de búsqueda

Clase 08

IIC 2133 - Sección 2

Prof. Mario Droguett

Sumario

El Misterio de EDD

El Misterio de EDD

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

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

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

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

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 $i \leq f$ **:**

        $p \leftarrow \texttt{Partition}(A, i, f)$

        $\texttt{Quicksort}(A, i, p-1)$

        $\texttt{Quicksort}(A, p+1, f)$

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

  $m \leftarrow \left\lfloor \frac{i+f}{2} \right\rfloor$

  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

Objetivos de la clase

- Comprender la noción de diccionario y qué operaciones soporta

Objetivos de la clase

- Comprender la noción de diccionario y qué operaciones soporta
- Conocer los árboles binarios de 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

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

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

Una nueva estructura

Construiremos una estructura con nuevas características

Una nueva estructura

Construiremos una estructura con nuevas características

- Dada una **llave o clave**, queremos asociarle un **valor**

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

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

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

Diccionarios

Definición

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

Diccionarios

Definición

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

- **Asociar** un valor a una llave

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

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

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

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

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**

Diccionarios

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

- Primero buscamos la llave (si está o no está)

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*

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?

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**...

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

Limitaciones de arreglos y listas

En una **listas ligada** de llaves

Limitaciones de arreglos y listas

En una **listas ligada** de llaves

- No tenemos **acceso por índice** de forma eficiente

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 $\mathcal{O}(n)$

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

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 $\mathcal{O}(1)$

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 $\mathcal{O}(n)$

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 $\mathcal{O}(\log(n))$

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

Limitaciones de arreglos y listas

En una **listas ligada** de llaves

- Para insertar solo necesitamos reasignar (pocos) punteros

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 $\mathcal{O}(1)$

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

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

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 $\mathcal{O}(n)$

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 $\mathcal{O}(1)$

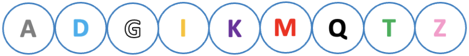
En un **arreglo** de llaves

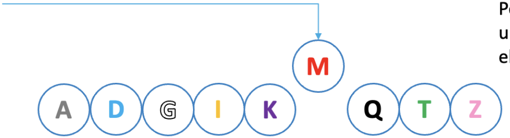
- Insertar un elemento puede gatillar un desplazamiento de datos
- En promedio, la inserción es $\mathcal{O}(n)$

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

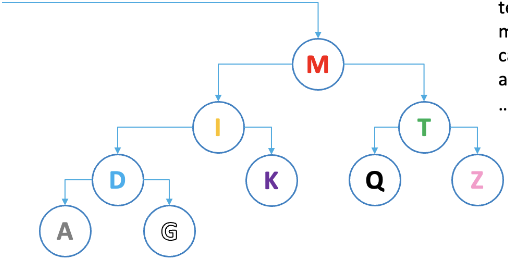
Modifiquemos las listas

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

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

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 un **nodo** que contiene una tupla (llave, valor)

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 un **nodo** que contiene una tupla (llave, valor)
2. El nodo puede tener hasta dos ABB's asociados mediante punteros

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

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

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 un **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**:

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 un **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.

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 un **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


divide and conquer…

Árboles binarios

Un árbol binario (de búsqueda o no) cumple que

Árboles binarios

Un árbol binario (de búsqueda o no) cumple que

- Cada nodo x tiene a lo más un **padre** $x.p$

Árboles binarios

Un árbol binario (de búsqueda o no) cumple que

- Cada nodo x tiene a lo más un **padre** $x.p$
- El nodo sin padre se conoce como **raíz**

Árboles binarios

Un árbol binario (de búsqueda o no) cumple que

- Cada nodo x tiene a lo más un **padre** $x.p$
- El nodo sin padre se conoce como **raíz**
- Cada nodo x tiene hasta dos punteros que (*apuntan*) a sub-árboles

Árboles binarios

Un árbol binario (de búsqueda o no) cumple que

- Cada nodo x tiene a lo más un **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

Árboles binarios

Un árbol binario (de búsqueda o no) cumple que

- Cada nodo x tiene a lo más un **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

Árboles binarios

Un árbol binario (de búsqueda o no) cumple que

- Cada nodo x tiene a lo más un **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
  - $x.p$ es puntero al padre (si tiene)

Árboles binarios

Un árbol binario (de búsqueda o no) cumple que

- Cada nodo x tiene a lo más un **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
    - $x.p$ es puntero al padre (si tiene)
- Un nodo sin punteros descendentes, i.e. sin hijos, se conoce como **hoja**

Árboles binarios

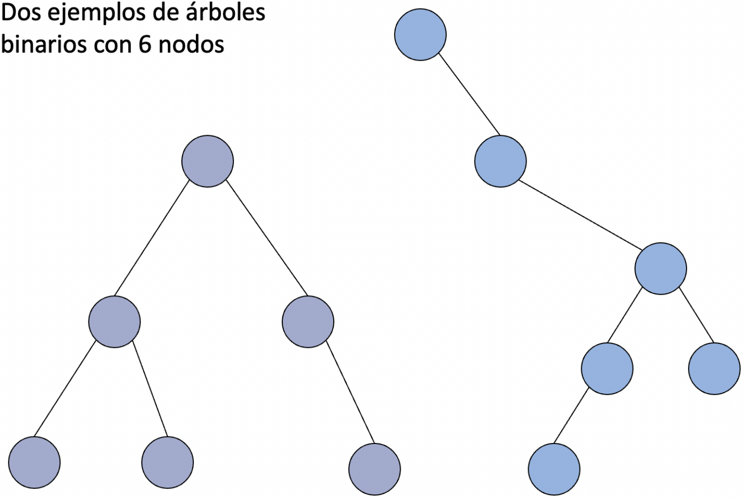
Un árbol binario (de búsqueda o no) cumple que

- Cada nodo x tiene a lo más un **padre** $x.p$
- El nodo sin padre se conoce como **raíz**
- Cada nodo x tiene hasta dos punteros que (*apuntan*) a sub-árboles
  - $x.\mathit{left}$ es un puntero al hijo izquierdo
  - $x.\mathit{right}$ es un puntero al hijo derecho
  - $x.p$ es puntero al padre (si tiene)
- Un nodo sin punteros descendentes, 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



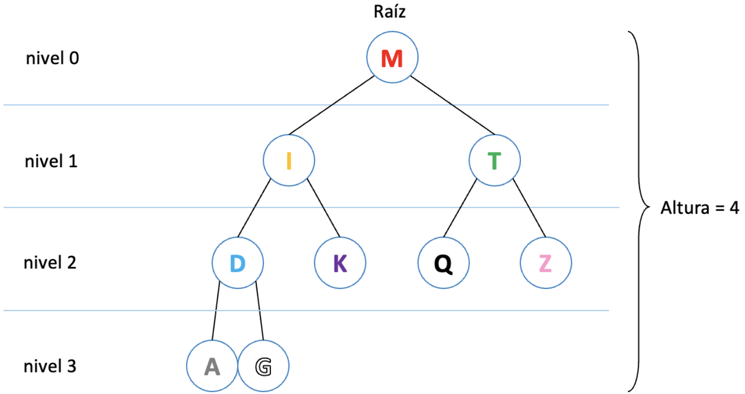
Anatomía de un árbol binario

Anatomía de un árbol binario

Por simplicidad, representaremos solo las llaves de los árboles

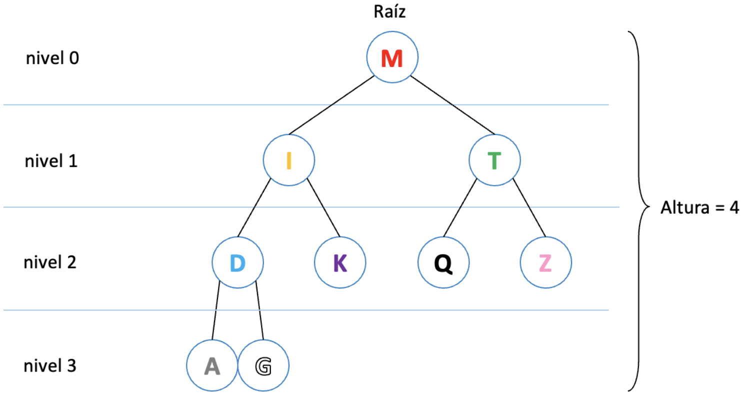
Anatomía de un árbol binario

Por simplicidad, representaremos solo las llaves de los árboles



Anatomía de un árbol binario

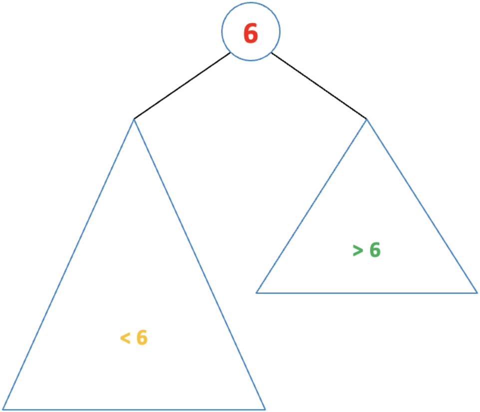
Por simplicidad, representaremos solo las llaves de los árboles



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

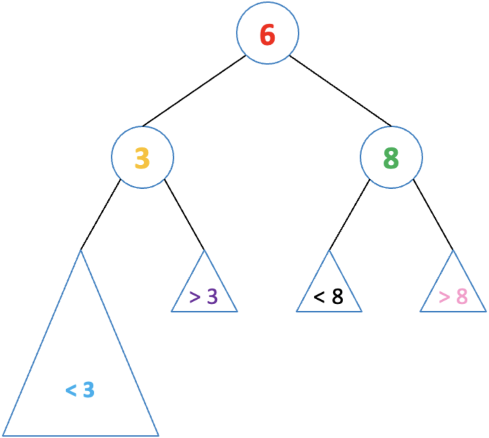
Anatomía de un árbol binario

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



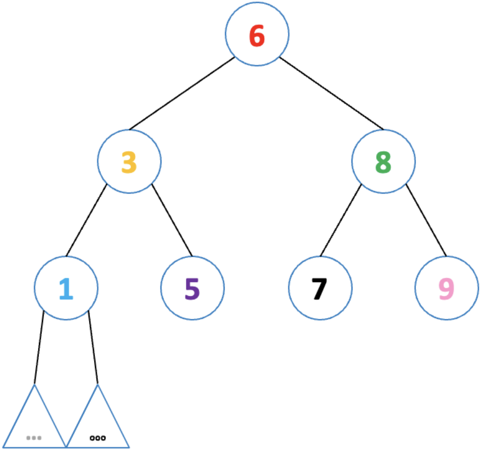
Anatomía de un árbol binario

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

Operaciones de un ABB

Recordemos nuestro objetivo al definir esta nueva estructura

- Queremos búsqueda rápida

Operaciones de un ABB

Recordemos nuestro objetivo al definir esta nueva estructura

- Queremos búsqueda rápida
- Para esto buscamos lograr un diccionario

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

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

Operaciones de un ABB

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

- $x.key$ es la llave del nodo

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

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

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

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
- $x.p$ el puntero al padre

Operaciones de un ABB

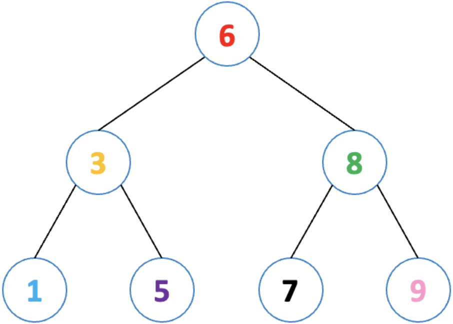
Completamos las definiciones de atributos de un **nodo** x en un ABB

- $x.key$ 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
- $x.p$ el puntero al padre

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

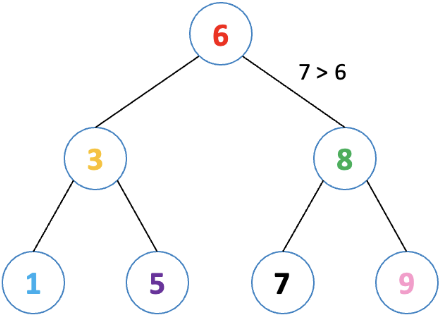
Ejemplo de búsqueda

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



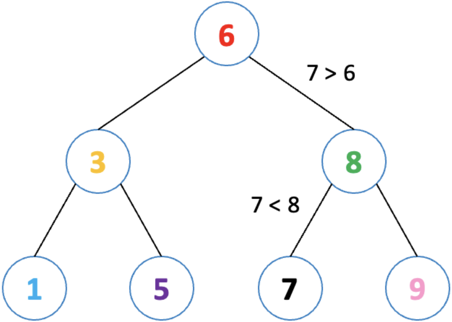
Ejemplo de búsqueda

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



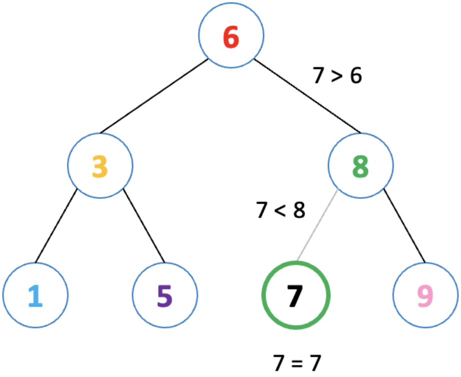
Ejemplo de búsqueda

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

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 = ∅ ∨ A.key = k :
        return A
    if k < A.key :
        return Search(A.left, k)
    return Search(A.right, k)
```

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 $k < A.key$ **:**
        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

Pensemos ahora en modificar el árbol

Operaciones para modificar un ABB

Pensemos ahora en modificar el árbol

- **Insertar** un nodo con una nueva llave produce un cambio en la estructura del árbol

Operaciones para modificar un ABB

Pensemos ahora en modificar el árbol

- **Insertar** un nodo con una nueva llave produce un cambio en la estructura del árbol
- De igual forma, **eliminar** un nodo también lo hace

Operaciones para modificar un ABB

Pensemos ahora en modificar el árbol

- **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**

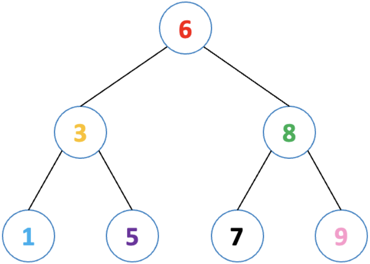
Operaciones para modificar un ABB

Pensemos ahora en modificar el árbol

- **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

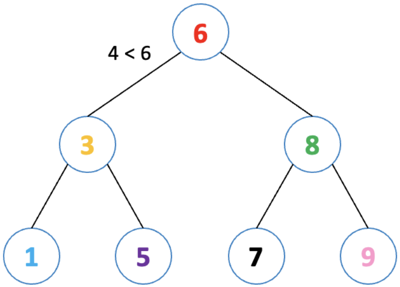
Ejemplo de inserción

Insertemos un nodo con llave 4



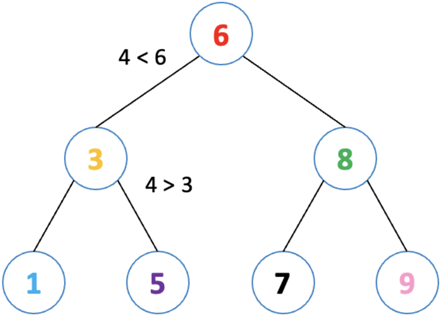
Ejemplo de inserción

Comparamos llaves para determinar en qué posición debe ser insertado

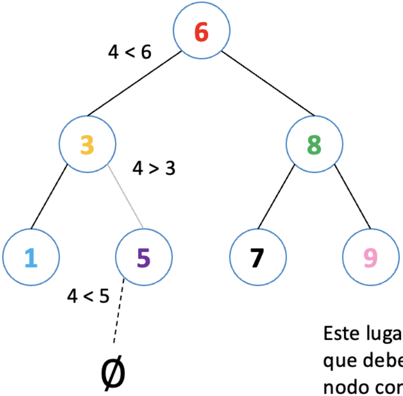


Ejemplo de inserción

Comparamos llaves para determinar en qué posición debe ser insertado



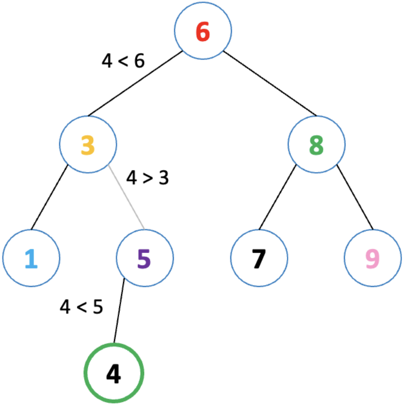
Ejemplo de inserción



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



Un cambio a la búsqueda

Modificaremos la búsqueda para saber quién es el padre del nodo encontrado

input : ABB A, ABB padre p, llave buscada k
output: Tupla con ABB encontrado y su padre

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

Un cambio a la búsqueda

Modificaremos la búsqueda para saber quién es el padre del nodo encontrado

input : ABB A, ABB padre p, llave buscada k
output: Tupla con ABB encontrado y su padre

```
Search(A, p, k):
    if A = ∅ ∨ A.key = k :
        return (A, p)
    if k < A.key :
        return Search(A.left, A, k)
    return Search(A.right, A, k)
```

Si retorna $(A, \varnothing)$, sabemos que A es la raíz

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

`Insert` (A, k, v)**:**

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

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

$\texttt{Insert}\ (A, k, v)$**:**

  $(B, p) \leftarrow \texttt{Search}(A, \varnothing, k)$ $\triangleright$ `versión que indica el padre`

  if $B = \varnothing$ **:**

    $B \leftarrow$ nodo vacío

    $B.key \leftarrow k$

    Conectar B al padre p en la posición adecuada

  $B.value \leftarrow v$

Este algoritmo mantiene la propiedad ABB al insertar

Operación de eliminación

La eliminación es un poco más compleja

Operación de eliminación

La eliminación es un poco más compleja

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

Operación de eliminación

La eliminación es un poco más compleja

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

- Lo borramos

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

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

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...

Operación de eliminación

La eliminación es un poco más compleja

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

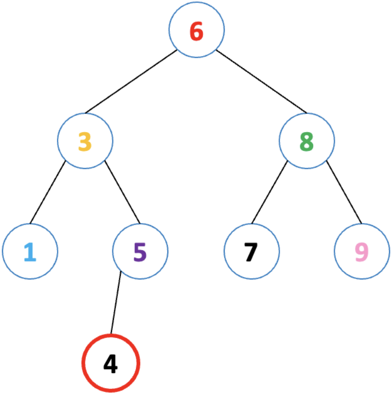
- Lo borramos
- Si tenía un hijo, el hijo lo reemplaza
- Es claro que se mantiene la propiedad ABB

En caso contrario...

¿Se puede reemplazar por otro árbol?

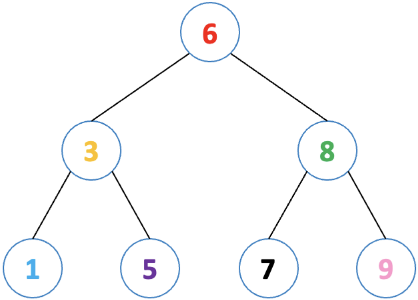
Ejemplo de eliminación

Si queremos eliminar el nodo con llave 4



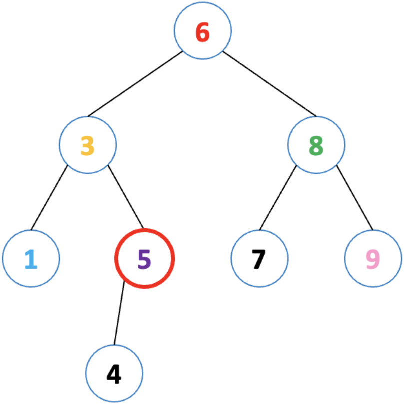
Ejemplo de eliminación

Simplemente se elimina y se preserva la propiedad ABB



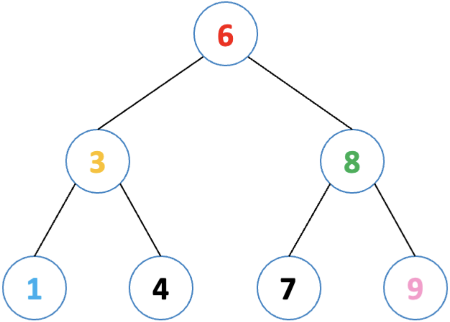
Ejemplo de eliminación

Si queremos eliminar el nodo con llave 5



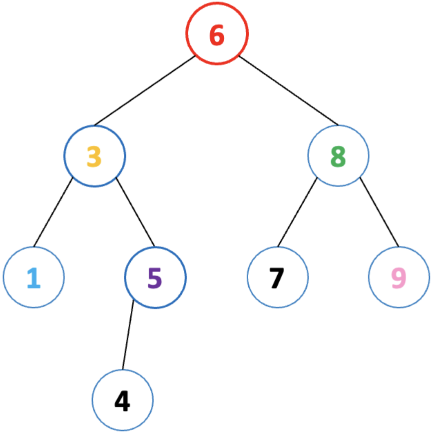
Ejemplo de eliminación

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



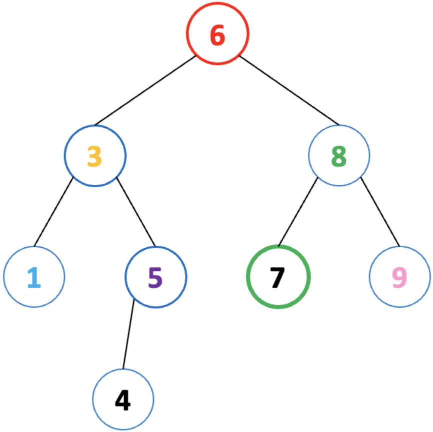
Ejemplo de eliminación

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



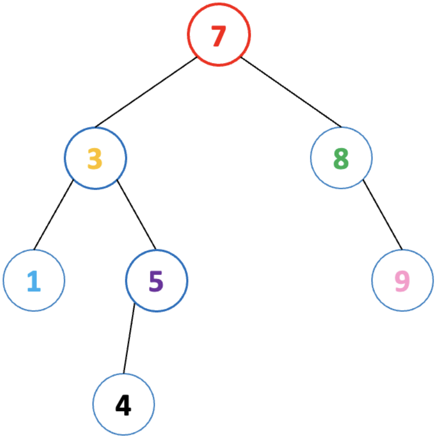
Ejemplo de eliminación

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



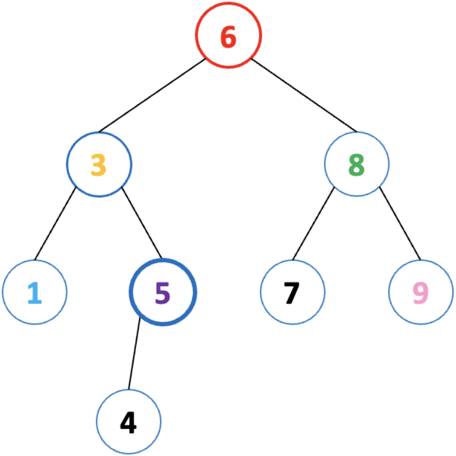
Ejemplo de eliminación

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



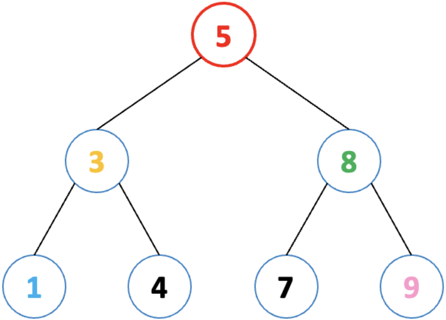
Ejemplo de eliminación

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



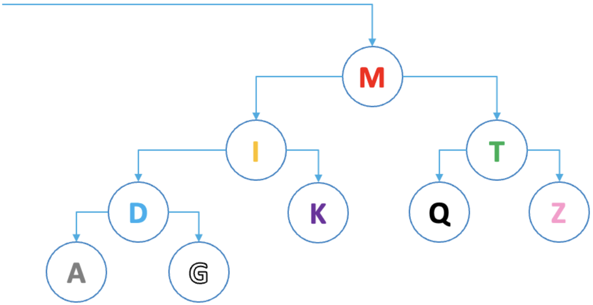
Ejemplo de eliminación

Y reubicamos su hijo con llave 4



Operación de eliminación

Nos interesa encontrar el sucesor/antecesor del nodo extraído



`Min` (A):

if $A.left = \varnothing$ **:**

  return A

return `Min`$(A.left)$

`Max` (A):

if $A.right = \varnothing$ **:**

  return A

return `Max`$(A.right)$

Operación de eliminación

Proponemos el siguiente algoritmo que preserva la propiedad ABB

`Delete` (A, k)**:**

$(D, p) \leftarrow \texttt{Search}(A, \varnothing, k)$ $\triangleright$ `Permite saber el padre de` D

if $D \neq \varnothing$ **:**

    if *D es hoja* **:** $D \leftarrow \varnothing$ y se elimina la referencia en p

    elif *D tiene un solo hijo H* **:** $D \leftarrow H$ y se actualiza p

    else:

        $R \leftarrow \texttt{Min}(D.right)$

        $t \leftarrow R.right$

        $D.key \leftarrow R.key$

        $D.value \leftarrow R.value$

        $R \leftarrow t$

Operación de eliminación

Proponemos el siguiente algoritmo que preserva la propiedad ABB

```
Delete (A, k):
    (D, p) ← Search(A, ∅, k)    ▷ Permite saber el padre de D
    if D ≠ ∅ :
        if D es hoja :  D ← ∅ y se elimina la referencia en p
        elif D tiene un solo hijo H :  D ← H y se actualiza p
        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

Antecesor y sucesor en general

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

Antecesor y sucesor en general

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

Ya tenemos algoritmos recursivos para esto

Antecesor y sucesor en general

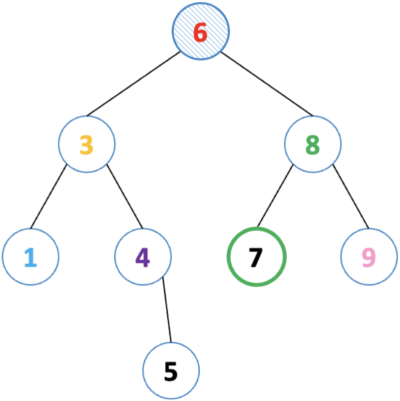
¿Qué tan fácil es determinar el sucesor y antecesor de un nodo?

Ya tenemos algoritmos recursivos para esto

¿Y si los tuviéramos en una lista ordenados?

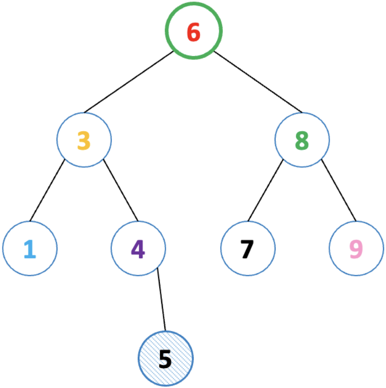
Antecesor y sucesor en general

Ya sabemos que es *fácil* encontrarlos 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

Objetivos de la clase

Objetivos de la clase

- ☐ Comprender la noción de diccionario y qué operaciones soporta

Objetivos de la clase

- ☐ Comprender la noción de diccionario y qué operaciones soporta
- ☐ Conocer los árboles binarios de 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

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

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