Análisis y Diseño de Algoritmos.

Sesión 5. 23 de septiembre de 2015.

Maestría en Sistemas Computacionales.

Por: Hugo Iván Piza Dávila.

¿Qué veremos hoy?

- Eliminación de la recursión: 3 formas generales.
  - Aplicado a búsqueda binaria y *mergesort.*
- Búsqueda Binaria con Interpolación.
- Búsqueda con Árboles Binarios.

La Recursión

- La recursión le da mayor poder de expresividad a una solución de software (más elegante y claro).
  - *"Repite las mismas instrucciones pero ahora con estos datos"*
  - Para el lector es más fácil entender la funcionalidad de un algoritmo recursivo que de uno iterativo.
  - Se parece a la definición matemática de funciones:

$$\text{Factorial}(n) \begin{cases} 1, & \text{si } n = 0 \\ n \times \text{Factorial}(n-1) & \text{si } n > 0 \end{cases}$$

La Recursión

- Sin embargo, tiene desventajas:
  1. Memoria insuficiente: cada llamada a una función es un apuntador que se guarda en la pila del programa.
     - Con la recursión se suele tener muchísimas funciones pendientes por acabar: desbordamiento de la pila.
     - Se va a dar en Quicksort con arreglos ordenados.
  2. Hay lenguajes que no soportan la recursión: sistemas embebidos.

Eliminar la recursión

- Hay que usar la recursión sólo cuando sea necesario (no existe una solución *iterativa* simple) o como 1er versión de un algoritmo.
- Para muchos algoritmos recursivos, existe una versión iterativa equivalente.
- Antes de convertir un algoritmo recursivo a su equivalente iterativo, identifiquemos 3 **formas generales** de recursión:
  1. Se realiza una llamada recursiva después de las operaciones con el espacio de búsqueda. Tal vez había varias llamadas posibles pero se seleccionó una.
  2. Dos (o más) llamadas recursivas después de las operaciones.
  3. Dos (o más) llamadas recursivas antes de las operaciones. En las primeras llamadas no se efectúan las operaciones: quedan en espera.

Forma General 1

- *tipo* función-recursiva(espacio de búsqueda, parámetros)
  1. ¿Puede acabar la recursión con éxito o fracaso? Devolver un valor o salir
  2. Operaciones con el espacio de búsqueda *(si aplica)*
  3. Modificación de parámetros (por cada caso posible)
  4. Llamar función-recursiva(espacio de búsqueda, nuevos parámetros)
- *tipo* función-iterativa(espacio de búsqueda, parámetros)
  - Repetir mientras el paso 1 sea falso:
    1. ¿Puedo terminar con éxito o fracaso?
    2. Operaciones con el espacio de búsqueda (*si aplica*)
    3. Modificación de parámetros (por cada caso posible)

Ejercicio 1

- Implementar la Forma General 1 con la Búsqueda Binaria para convertirlo en un algoritmo iterativo.
- El método es:
  - `int binarySearchIte(int[] array, int value)`

Forma general 2

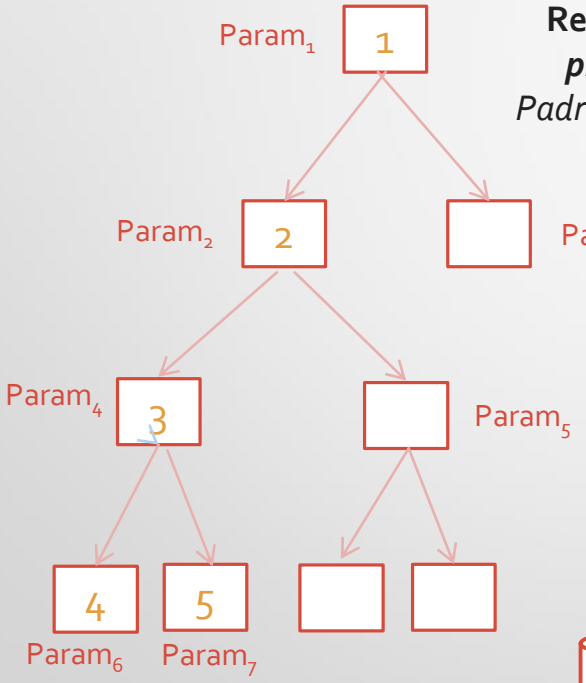
- *tipo* **función-recursiva**(espacio de búsqueda, parámetros)
  1. ¿Puede acabar la recursión con éxito o fracaso?
  2. Operaciones con el espacio de búsqueda
  3. Llamar **función-recursiva**(espacio de búsqueda, $parámetros_1$)
  4. Llamar **función-recursiva**(espacio de búsqueda, $parámetros_2$)
- *tipo* función-iterativa(espacio de búsqueda, parámetros)
  1. Crear una **pila de parámetros** y depositar los parámetros recibidos
  2. Mientras la **pila** no esté vacía
     a) Sacar de la pila los últimos parámetros depositados *(en orden inverso)*
     b) ¿Puedo ir a la siguiente iteración sin meter parámetros? *(como el fin de la recursión)*
     c) Operaciones con el espacio de búsqueda
     d) Meter a la pila $parámetros_2$, $parámetros_1$

Forma general 3

- *tipo* **función-recursiva**(espacio de búsqueda, parámetros)
  1. ¿Puede acabar la recursión con éxito o fracaso?
  2. Llamar **función-recursiva**(espacio de búsqueda, $parámetros_1$)
  3. Llamar **función-recursiva**(espacio de búsqueda, $parámetros_2$)
  4. Operaciones con el espacio de búsqueda
- *tipo* **función-iterativa**(espacio de búsqueda, parámetros)
  1. Crear una **pila de parámetros**, depositar los parámetros y una bandera *visitado* = *falso*
  2. Mientras la **pila** no esté vacía
     a) Sacar de la pila los últimos parámetros depositados *(en orden inverso)*
     b) ¿Puedo ir a la siguiente iteración sin meter parámetros? *(como el fin de la recursión)*
     c) ¿Los parámetros no han sido visitados? Meter a la pila parámetros, $parámetros_2$, $parámetros_1$
     d) En caso contrario, realizar operaciones con el espacio de búsqueda.

FG2 vs FG3

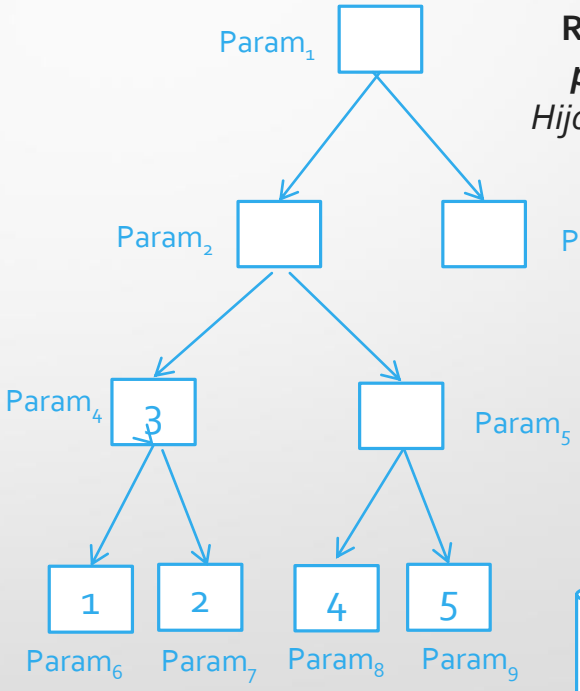
Forma General 2



Recorrido en *pre-orden:*
Padre, Hijo$_1$, Hijo$_2$

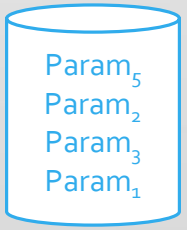
Parámetros pendientes por evaluar al ejecutar **5**:

Forma General 3



Recorrido en *post-orden:*
Hijo$_1$, Hijo$_2$, Padre

Parámetros pendientes por evaluar al ejecutar **5**:

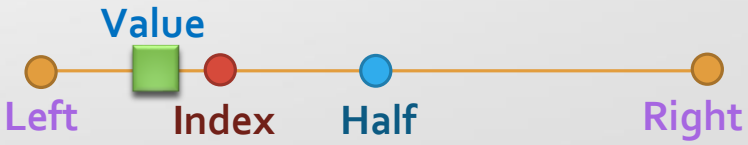


Ejercicio

- Aplicar la Forma General 3 a MergeSort
  - No se crearán nuevos arreglos. En su lugar, se utilizarán enteros que denotan los límites de los sub-arreglos.
  - El método merge() recibirá el arreglo original y los límites izquierdo y derecho de los dos sub-arreglos a mezclar.
    - El resultado de la mezcla se guardará en el arreglo original sabiendo que los dos sub-arreglos son adyacentes y excluyentes. No usar arreglos temporales.
  - No depositar <left, right, visited> en la pila si:
    - Left y right son iguales (arreglo trivial).
    - El sub-arreglo actual ya fue visitado: merge.
  - En otro caso, depositar en la pila:
    - Argumentos del sub-arreglo actual, indicando que ya fue visitado.
    - Argumentos de la mitad derecha que va de middle + 1 a right.
    - Argumentos de la mitad izquierda que va de left a middle.
  - Implementaciones de pilas: ArrayDeque (+ rápida), LinkedList, Stack (+ lenta).

Búsqueda Binaria con Interpolación

- Sólo para arreglos ordenados.
- Es una mejora de la búsqueda binaria
  - El nuevo espacio de búsqueda es, por lo general, más pequeño que la mitad del anterior.
  - La posición de corte **Index** sigue una *regla de tres* que relaciona la diferencia de valores con la diferencia de posiciones:
    - Right - Left $\rightarrow$ $List_{Right}$ - $List_{Left}$
    - **Index** - Left $\rightarrow$ Value - $List_{Left}$



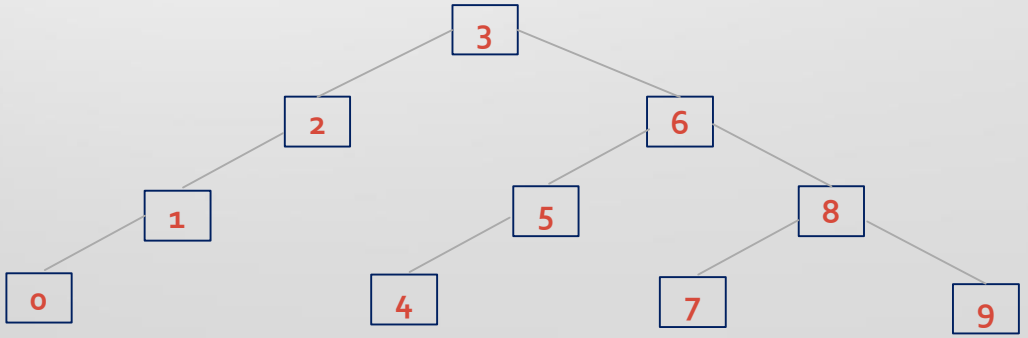
- Número máximo de iteraciones: $\log_2\log_2 N$
  - Complejidad temporal menor a logarítmica.

Búsqueda Binaria con Interpolación

- Algoritmo:
- Si el valor a buscar abandona el espacio de búsqueda.
  - Terminar con fracaso.
- Calcular el punto de corte.
- Si el punto de corte abandona el espacio de búsqueda.
  - Terminar con fracaso.
- Si el valor se encuentra en el punto de corte.
  - Terminar con éxito.
- Generar la recursión con el nuevo punto de corte de forma semejante a la búsqueda binaria.

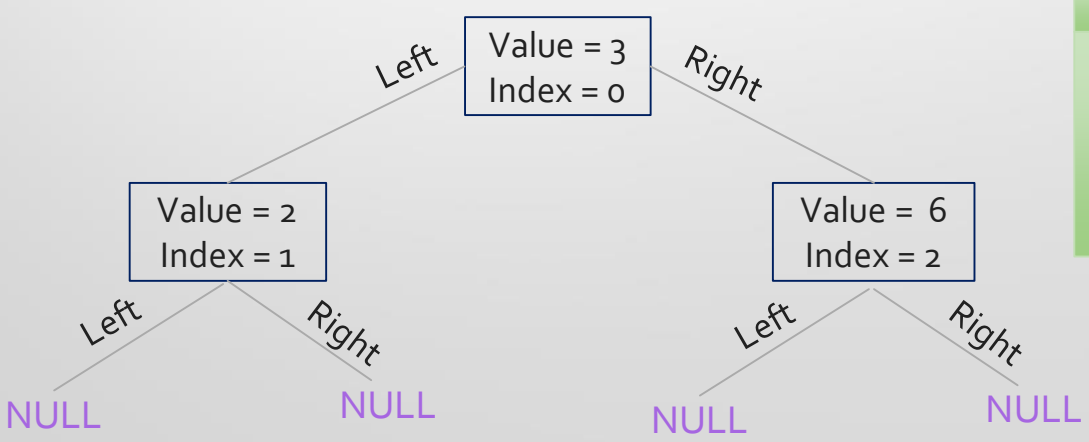
Arboles Binarios

- Para arreglos desordenados
- A partir de la lista, se construye un árbol binario.
  - No necesariamente balanceado ni alineado.
  - El hijo izquierdo siempre será menor que el padre.
  - El hijo derecho siempre será mayor (o igual) que el padre.
- Árbol binario de {3, 2, 6, 1, 5, 8, 0, 4, 7, 9}:



Estructura de un Árbol Binario

- Un árbol binario está compuesto de uno o más nodos.
- El nodo inicial es la raíz: almacena el primer elemento de la lista.
- A partir de cada nodo se desprenden uno o dos nodos, llamados hijos *izquierdo* y *derecho*.
- De los nodos *hoja* no se desprende ningún nodo.



Node

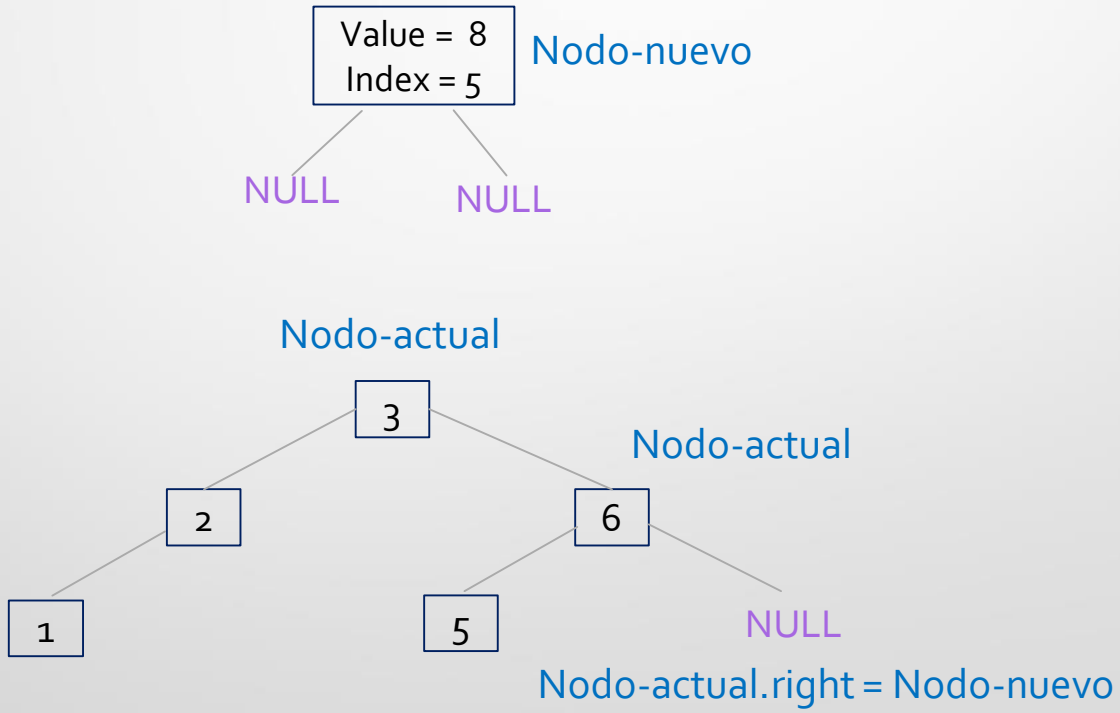
Value : int
Index : int
Left : Node
Right : Node

De la lista a un Árbol Binario

- Por cada elemento del arreglo diferente al primero:
  1. Crear un nuevo nodo con el valor y posición del elemento.
  2. Apuntar al nodo raíz.
  3. Si el nodo apuntado contiene el valor a buscar, devolver la posición que contiene.
  4. Si el nodo apuntado contiene un valor mayor al que se busca, apuntar ahora al hijo izquierdo. Si no, apuntar al derecho.
  5. Si no existe el hijo apuntado (NULL) insertar ahí el nuevo nodo.
  6. En caso contrario, regresar al paso 3.

De la lista a un Árbol Binario

Ejemplo. Del arreglo {3, 2, 6, 1, 5, 8, 0, 4, 7, 9}, insertar 8 en el AB.



Búsqueda en un Árbol Binario

- Implementación más intuitiva:
  - Método recursivo que recibe el nodo actual y el valor a buscar.
  - Devuelve el índice.
  - En la primera llamada se pasa el nodo raíz.

1. Si el nodo recibido es nulo, no se encontró [devolver -1]
2. Si el valor indicado por el nodo es igual al valor que se busca, devolver el índice indicado en el nodo.
3. Si el valor indicado por el nodo es menor al que se busca, repetir la búsqueda [de forma recursiva] con el nodo izquierdo al actual.
4. En otro caso, repetir la búsqueda con el nodo derecho al actual.

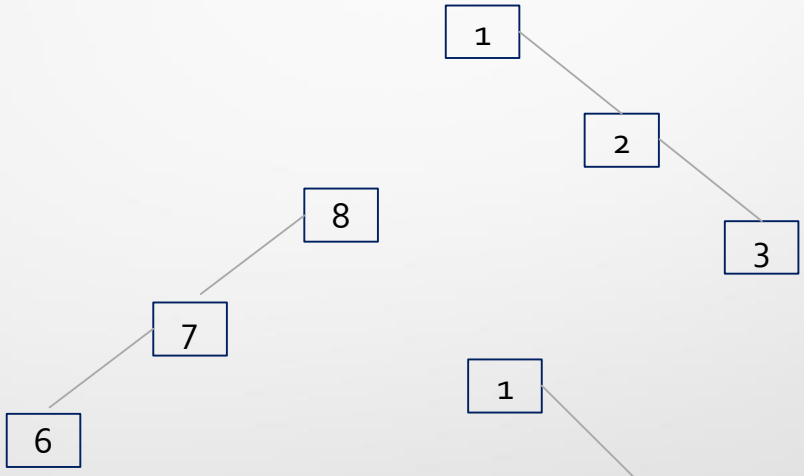
Análisis de la búsqueda con AB

- El número máximo de brincos que se realizan para llegar a la posición de cada elemento *i* es el número de niveles del árbol.
- Un árbol binario balanceado de N nodos tiene $\log_2 N$ niveles.
  - La complejidad temporal es quasi-lineal ($N \cdot \log N$)
  - Una búsqueda secuencial tiene menor complejidad ($N < N \cdot \log N$)
- ¿Dónde está la ventaja?
  - Una vez creado el AB, la búsqueda de un elemento tiene **complejidad logarítmica**, según estudios: **2ln N**.
  - En la práctica, el árbol binario se crea una vez y se utiliza muchas.

Peores casos del AB

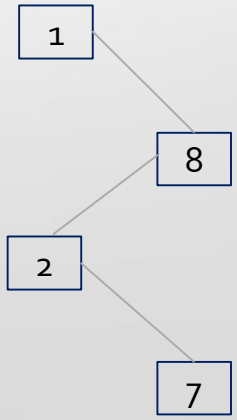
- Existen tres peores casos con complejidad lineal:

1. Lista ordenada.



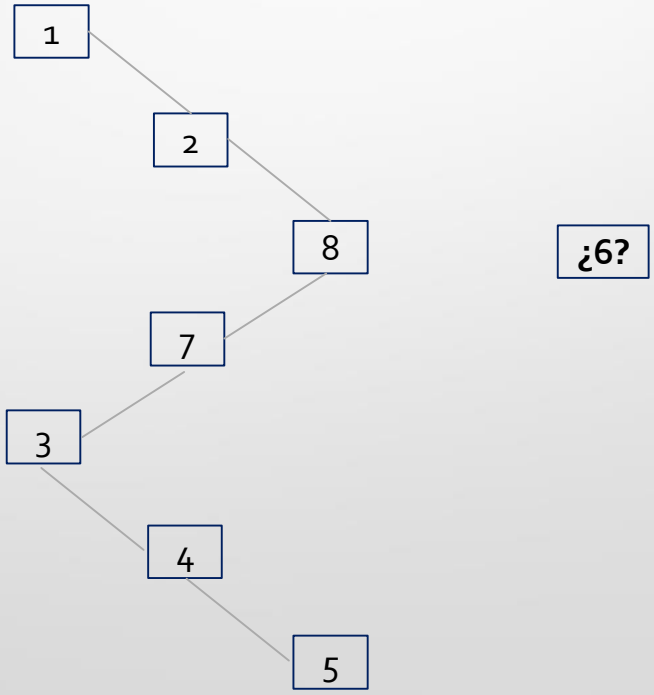
2. Lista invertida.

3. Lista que alterna menor con mayor.



Peores casos del AB

- En general, cuando muchos nodos tienen un hijo.
  - Se tienen segmentos de lista ordenados o invertidos.



Ejercicios

- Implementar la búsqueda Binaria con Interpolación y comprobar *a posteriori* que su complejidad es $\approx \log_2\log_2 N$.
- Implementar la búsqueda con Árbol Binario.
- Comprobar que con arreglos aleatorios la complejidad temporal es logarítmica y con arreglos ordenados, invertidos y alternados es lineal.