# AED: Árboles de Búsqueda Binaria

gabriel.spranger@utec.edu.pe
macarena.oyague@utec.edu.pe

Abril 2022

## Contents

# 1 Terminología

- **Árbol:** grafo acíclico con $n$ nodos y $n-1$ aristas.

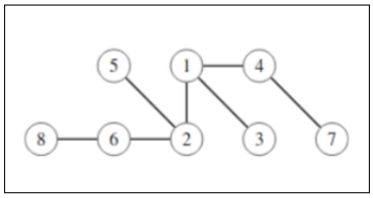



Figure 1: Un árbol.

- **Árbol Enraizado:** un árbol es *enraizado* si solo un nodo tiene grado 0 de entrada y todos los demás tienen grado 1 de entrada. Al nodo de grado 0 se le llama *raíz*. Asumiremos que cada árbol definido más abajo será un árbol enraizado.

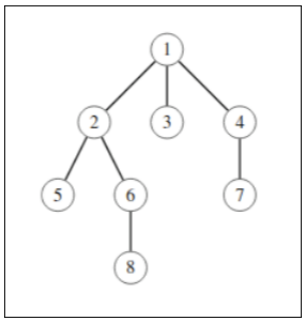



Figure 2: Un árbol enraizado.

- **Árbol Binario:** árbol donde cada nodo tiene máximo 2 nodos hijos, formando así dos subárboles (izquierdo y derecho).

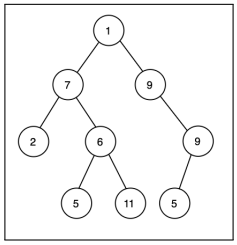



Figure 3: Un árbol binario. Notar que el arbol binario no guarda ningun orden entre sus nodos.

- **Árbol Binario Lleno:** árbol binario donde cada nodo tiene 0 o 2 hijos.

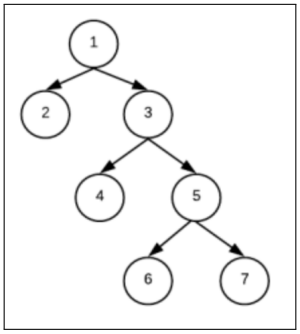



Figure 4: Un árbol binario lleno.

- **Nivel (de un nodo):** longitud del camino desde la raíz hasta el nodo.
- **Árbol Binario Completo:** árbol binario donde cada *nivel* está *lleno*, sin contar el último, el cual podría estar o no *lleno* y además los nodos de este último nivel están pegados hacia la izquierda.

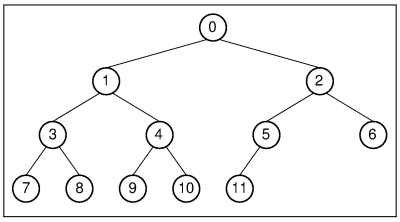



Figure 5: Un árbol binario completo.

- **Árbol Binario Perfecto:** árbol binario lleno donde solo las hojas son los nodos con 0 hijos.

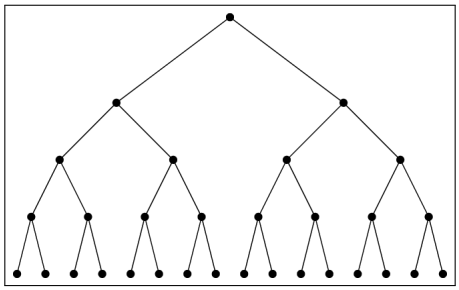


Figure 6: Un árbol binario perfecto.

- **Altura**

  - **de un Nodo:** longitud del camino **más largo** desde el nodo hasta una hoja.
  - **de un Árbol:** altura del nodo raíz.

- **Árbol Binario Balanceado:** árbol binario donde dado un nodo, la diferencia de las alturas de sus subárboles izquierdo y derecho, difieren por máximo 1. Una propiedad de los árboles binarios balanceados es: $h = \mathcal{O}(\lg n)$, donde $h$ es la altura del árbol y $n$ el número de nodos en el árbol.

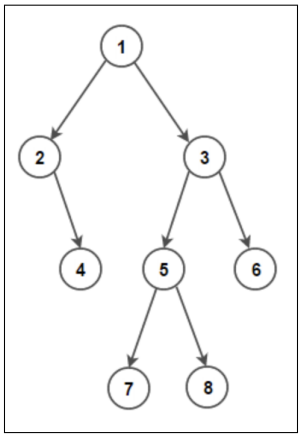



Figure 7: Un árbol binario balanceado.

- **Árbol Binario de Búsqueda:** árbol binario donde dado un nodo $p$, todos los nodos de su subárbol izquierdo son menores al valor de $p$ y todos los nodos de su subárbol derecho son mayores a $p$. Formalmente, la propiedad de árboles binarios de búsqueda dice que para cualquier nodo $x$ en un árbol binario de búsqueda, si un nodo $y$ es su hijo izquierdo, entonces $y < x$, mientras que si un nodo $z$ es su hijo derecho, entonces $z > x$. El objetivo de las operaciones INSERT y DELETE es que modifiquen el árbol de tal manera que se **mantenga** la propiedad de árboles binarios de búsqueda.

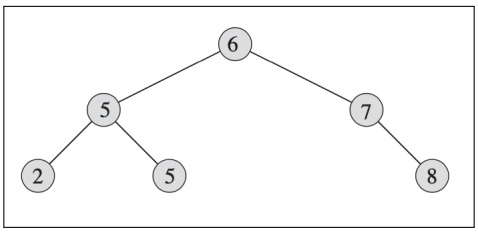



Figure 8: Un árbol binario de búsqueda.

- **Diámetro (de un árbol):** número de nodos en el camino más largo entre dos nodos hoja (un algoritmo usa DFS para hallarlo en $\mathcal{O}(n)$).
- **Profundidad:**
  - **de un Nodo:** nivel de un nodo.
  - **de un Árbol:** nivel del nodo raíz.
- **Orden (de un árbol):** número máximo de hijos que puede tener cualquier nodo de ese árbol. Un árbol binario es de orden 2.

# 2 Recorridos

## 2.1 Inorder

Como su nombre lo dice, el recorrido *inorder* visita todos los nodos de tal manera que se visitan en orden ascendiente ya que para cualquier nodo $x$ del árbol binario de búsqueda, su hijo izquierdo $y < x$ y su hijo derecho $z > x$. Sin esta propiedad, el recorrido *inorder* no visitaría los nodos de menor a mayor. Además, hay distintos tipos de árboles como los *heaps* cuyo recorrido *inorder* no tiene el mismo resultado que en los árboles binarios de búsqueda, dado que los *heaps* tienen otra **propiedad**.

Dado que el recorrido *inorder* visita los nodos de un árbol binario de búsqueda de menor a mayor, primero tiene que visitar el nodo izquierdo (ya que es menor), luego el nodo padre (ya que es mayor que el nodo izquierdo) y finalmente el nodo derecho (ya que es mayor al nodo padre).

Listing 1: Pseudocódigo del recorrido inorder para un árbol binario.

```
inorder(x):
1.   if x != null:
2.     inorder(x.left)
3.     // procesar x, quizas imprimirlo
4.     inorder(x.right)
```

## 2.2 Postorder

El recorrido *postorder* visita al padre **luego** (*post*) de visitar a los hijos. Tiene distintas aplicaciones dependiendo del uso que se le dé al árbol.

Listing 2: Pseudocódigo del recorrido postorder para un árbol binario.

```
postorder(x):
1.   if x != null:
2.     postorder(x.left)
3.     postorder(x.right)
4.     // procesar x, quizas imprimirlo
```

## 2.3 Preorder

El recorrido *preorder* visita al padre **antes** (*pre*) de visitar a los hijos. Tiene distintas aplicaciones dependiendo del uso que se le dé al árbol.

Listing 3: Pseudocódigo del recorrido preorder para un árbol binario.

```
preorder(x):
1.  if x != null:
2.    // procesar x, quizas imprimirlo
3.    preorder(x.left)
4.    preorder(x.right)
```

## 2.4 Depth-First Search (DFS)

Empieza desde un nodo $x$, escoge **el primer** hijo de $x$ (si es que $x$ tiene hijo y si ese hijo no ha sido visitado) y se repite el proceso hasta que el nodo $y$ que se está visitando ya no tenga más hijos. En dicho caso, el algoritmo *backtrackea*, es decir, retrocede hasta el padre de $y$ (llamémoslo $z$) y elige **el siguiente** hijo de $z$ (si es que tiene otro hijo y si no lo ha visitado aún, sino *backtrackea* de nuevo al padre de $z$).

Esta manera de operar de DFS hizo que lo llamen **depth**-first porque primero trata de ir lo más profundo que pueda hasta que ya no hay más nodos que recorrer y es ahí cuando hace el *backtracking* para repetir el proceso con el siguiente nodo no visitado.

Listing 4: Pseudocódigo de DFS genérico para cualquier árbol.

```
dfs(x):
1.  for hijo in hijos de x:
2.      if hijo is not visited:
3.          dfs(hijo)
4.  mark x as visited
```

Haciendo un análisis laxo de la complejidad de esta rutinra, podemos decir que como cada nodo es visitado solo una vez, el algoritmo DFS corre en $\mathcal{O}(n)$, donde $n$ es el número de nodos en el árbol.

Notar que habíamos mencionado que DFS(X) visita **el primer** hijo de $x$, lo cual significa que el orden que se ha determinado para los hijos de un nodo $x$, determina el orden de recorrido de DFS. Por ejemplo, si tenemos un árbol binario de búsqueda donde dado un nodo $x$ **el primer** hijo de $x$ es el nodo izquierdo y **el segundo** hijo es el nodo derecho, DFS recorre el árbol binario de búsqueda tal cual lo hace PREORDER.

Una aplicación interesante de DFS es que si modelamos las esquinas de un laberinto como nodos y los caminos como aristas, entonces con DFS podemos hallar en tiempo lineal el camino que cruza el laberinto.

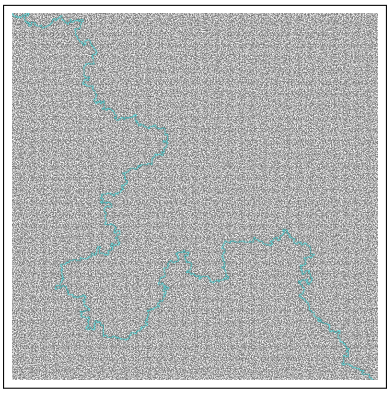


Figure 9: Laberinto solucionado mediante DFS. Tomado de este link.

## 2.5 Breadth-First Search (BFS)

Primero visita todos los nodos que tienen nivel 0, luego todos los de nivel 1, luego los de nivel 2 y así sucesivamente hasta el último nivel del árbol. También se le llama *level-order traversal*.

Listing 5: Pseudocódigo de BFS genérico para cualquier árbol.

```
bfs(x):
1.  encolar x
2.  while cola tiene nodos:
3.    nodo = desencolar un nodo (el primero de la cola)
4.    for hijo in hijos de nodo:
5.      if hijo is not visited:
6.        encolar hijo
7.    mark nodo as visited
```

De manera similar a DFS, como cada nodo es visitado solo una vez y también asumiendo que las operaciones de "encolar" y "desencolar" corren en $\mathcal{O}(1)$, podemos decir que BFS corre en $\mathcal{O}(n)$, donde $n$ es el número de nodos en el árbol.

Una aplicación interesante de BFS es que si lo corremos en un grafo y guardamos en cada nodo el nivel de dicho nodo, entonces estamos guardando el mínimo número de aristas desde el nodo donde se empezó el BFS hasta todos los demás nodos, es decir, un *shortest path tree*, pero ojo que este no es el mismo de Dijkstra, el cual toma en cuenta el peso de cada arista, porque BFS solo toma en cuenta el **número** de aristas.

# 3 Operaciones

Como se mencionó anteriormente, un árbol binario de búsqueda mantiene la siguiente propiedad: dado cualquier nodo $x$, si $y$ es el hijo izquierdo de $x$, entonces $y < x$ y si $z$ es el hijo derecho de $x$, entonces $z > x$. Esta propiedad hace que ciertas operaciones sobre el árbol binario de búsqueda corran en tiempo logarítmico **en el mejor caso** (en la sección PALABRAS FINALES se explica porqué se menciona "en el mejor caso"). Hay dos tipos de operaciones: consultas y mutaciones. La idea es que las mutaciones modifiquen la estructura de datos de tal manera que la propiedad descrita anteriormente se mantenga, de lo contrario se pierden los beneficios que esta propiedad trae (operaciones en tiempo logarítmico en el mejor caso).

## 3.1 Consultas

### 3.1.1 Search

Dada la propiedad, si queremos encontrar un nodo con valor $x$, entonces empezamos desde la raíz y recursivamente preguntamos si $x$ es igual al valor de la raíz (retornar el valor que sea pertinente sea un booleano, el mismo nodo, etc), si $x$ es menor (llamar recursivamente pero ahora al nodo izquierdo como raíz) o si $x$ es mayor (llamar recursivamente pero ahora al nodo derecho como raíz). La recursión continua hasta que la llamada se haga sobre un nodo que es NULL.

Esta consulta depende de la **altura del árbol**, ya que SEARCH forma un camino simple desde la raíz hasta el nodo que se quiere encontrar, llegando en el peor caso (cuando el nodo no existe) hasta una hoja, es decir, hasta recorrer exactamente "altura del árbol" nodos. Por lo tanto, si la altura del árbol es logarítmica, entonces esta consulta correrá en $\mathcal{O}(\lg n)$.

### 3.1.2 Maximum

Dada la propiedad, el nodo con el valor máximo del árbol binario de búsqueda es la hoja representada por el último nodo de un camino simple desde la raíz, pero solo tomando el hijo derecho en cada paso. Esta consulta depende de la altura del árbol, dado que en el peor caso, el tamaño del camino simple para encontrar el nodo con el valor máximo es igual a la altura del árbol. Por lo tanto, si la altura del árbol es logarítmica, entonces esta consulta correrá en $\mathcal{O}(\lg n)$.

### 3.1.3 Minimum

Dada la propiedad, el nodo con el valor mínimo del árbol binario de búsqueda es la hoja representada por el último nodo de un camino simple desde la raíz, pero solo tomando el hijo izquierdo en cada paso. Esta consulta depende de la altura del árbol, dado que en el peor caso, el tamaño del camino simple para encontrar

el nodo con el valor mínimo es igual a la altura del árbol. Por lo tanto, si la altura del árbol es logarítmica, entonces esta consulta correrá en $\mathcal{O}(\lg n)$.

### 3.1.4 Successor

Imaginemos que imprimimos todos los valores del árbol en orden mediante un recorrido *inorder*, si ubicamos un nodo con valor $x$, su sucesor sería el nodo con el valor siguiente. En otras palabras, el sucesor de un nodo con valor $x$ es el nodo cuyo valor $y$ es el menor posible que cumpla que $y > x$. Por ejemplo, en el siguiente gráfico

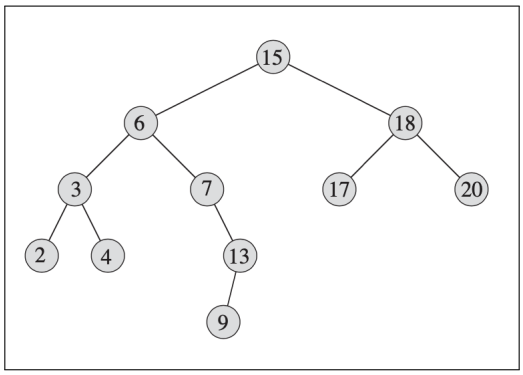



Figure 10: Un árbol binario de búsqueda.

el sucesor del nodo con valor 15 es el nodo con valor 17 (el que le sigue si fueran imprimidos *inorder*).

Para hallar el sucesor de un nodo con valor $x$, hay dos casos:

1. Si $x$ tiene un subárbol derecho, entonces el sucesor es MINIMUM($x.right$). Esto es porque sabemos que todos los nodos en el subárbol derecho de $x$ tienen un mayor valor y para hallar el sucesor nosotros queremos hallar el mínimo de todos esos nodos que son mayores a $x$.

2. Si $x$ **no** tiene subárbol derecho, el sucesor de $x$ es **el primer** ancestro de $x$ (llamémosle $z$) donde $x$ se encuentra en el subárbol izquierdo de $z$, es decir, $x$ es el nodo con el mayor valor del subárbol izquierdo de $z$. Esto es porque $x$ es el máximo valor del subárbol izquierdo de su sucesor. Para hallar el primer ancestro que tiene a $x$ en su subárbol izquierdo, mantenemos dos punteros, uno hacia $x$ y otro hacia su padre y "subimos ambos punteros" mientras el de más abajo sea el hijo derecho del de más arriba. Cuando esta condición no se cumpla, el puntero de más arriba será el sucesor de $x$. Como en el peor caso se recorrería toda la altura del árbol, este algoritmo corre en $\mathcal{O}(h)$.

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5       x = y
6       y = y.p
7  return y
```

Figure 11: Imagen tomada del Cormen. Muestra el pseudocódigo de ambos casos para hallar el sucesor de un nodo $x$.

### 3.1.5 Predecessor

La idea del predecesor es el reflejo del sucesor. El predecesor de un nodo con valor $x$ es el nodo cuyo valor $y$ es el mayor posible que cumpla que $y < x$. Por ejemplo, en el siguiente gráfico

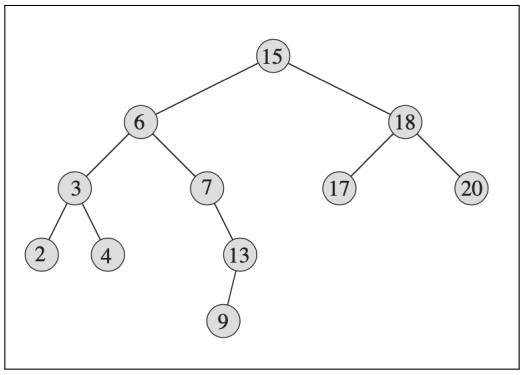



Figure 12: Un árbol binario de búsqueda.

el predecesor del nodo con valor 6 es el nodo con valor 4 (el que le precede si fueran imprimidos *inorder*).

Para hallar el predecesor de un nodo con valor $x$, hay dos casos:

1. Si $x$ tiene un subárbol izquierdo, entonces el predecesor es MAXIMUM($x.left$). Esto es porque sabemos que todos los nodos del subárbol izquierdo de $x$ tienen un menor valor y para hallar el predecesor nosotros queremos hallar el máximo de todos esos nodos que son menores a $x$.

2. Si $x$ **no** tiene subárbol izquierdo, el predecesor de $x$ es **el primer** ancestro de $x$ (llamémosle $z$) donde $x$ se encuentra en el subárbol derecho $z$, es decir, $x$ es el nodo con menor valor del subárbol derecho de $z$. Notar

que la operación de predecesor es casi una imagen reflejada de TREE-SUCCESSOR. Como en el peor caso se recorrería toda la altura del árbol, este algoritmo corre en $\mathcal{O}(h)$.

## 3.2 Mutaciones

### 3.2.1 Insert

Para insertar un elemento, usamos un algoritmo muy similar al SEARCH. Solo que si el valor que estamos insertando es igual al valor de un nodo existente en el árbol, retornamos inmediatamente (el INSERT falla*). Si no es igual, entonces nos vamos hacia la izquierda o derecha dependiendo de si el valor que insertamos es menor o mayor, respectivamente. Al final, cuando el último nodo en el camino que estamos formando tiene a su hijo izquierdo o derecho como NULL, insertamos el nuevo nodo en la posición correspondiente.

Como en el peor caso tendríamos que recorrer toda la altura del árbol para llegar al nodo hoja para poder insertar el nuevo nodo, esta operación también depende de la altura del árbol, es decir, corre en $\mathcal{O}(h)$, donde $h$ es la altura del árbol.

*Esto depende de cada uno, pero si hacemos el INSERT de esta manera, nos aseguramos que el árbol no tendrá duplicados (y encima los elementos mantienen un orden) y por lo tanto podríamos usar la implementación de este árbol como un *set* o conjunto.

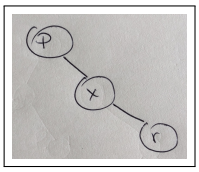
### 3.2.2 Delete

Para eliminar un nodo $x$ con un valor específico hay tres casos:

1. $x$ **no tiene hijos:** simplemente removemos el nodo y actualizamos el padre para que ya no apunte a ese nodo sino que ahora apunte a NULL.

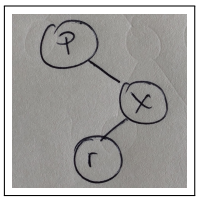
2. $x$ **tiene un solo hijo:** el hijo toma la posición del padre. Esto mantiene la propiedad del árbol binario de búsqueda. Para explicar el porqué, hay cuatro casos (notar que en todos los casos $x$ no necesariamente es el único hijo de $p$ y que $r$ podría o no tener hijos):

   - $x$ es un hijo derecho y $x$ tiene un solo hijo derecho:



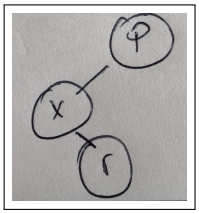
Por la propiedad sabemos que $x > p$ y que $r > p$ (por transitividad y porque sino $r$ no estuviera en el subárbol derecho de $p$). Por lo tanto, si eliminamos $x$ y lo reemplazamos por $r$, se sigue cumpliendo la propiedad dado que $r > p$.

- $x$ es un hijo derecho y $x$ tiene un solo hijo izquierdo:



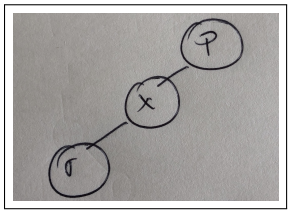

Por la propiedad sabemos que $x > p$ y que $r > p$ (sino no estuviera en el subárbol derecho de $p$). Por lo tanto, si eliminamos $x$ y lo reemplazamos por $r$, se sigue cumpliendo la propiedad dado que $r > p$.

- $x$ es un hijo izquierdo y $x$ tiene un solo hijo derecho:




Por la propiedad sabemos que $x < p$ y que $r < p$ (sino no estuviera en el subárbol izquierdo de $p$). Por lo tanto, si eliminamos $x$ y lo reemplazamos por $r$, se sigue cumpliendo la propiedad dado que $r < p$.

- $x$ es un hijo izquierdo y $x$ tiene un solo hijo izquierdo:




Por la propiedad sabemos que $x < p$ y que $r < p$ (por transitividad y porque sino $r$ no estuviera en el subárbol izquierdo de $p$). Por lo tanto, si eliminamos $x$ y lo reemplazamos por $r$, se sigue cumpliendo la propiedad dado que $r < p$.

3. $x$ **tiene dos hijos:** encontramos el sucesor $y$ de $x$ el cual está en el subárbol derecho de $x$, ya que sabemos que $x$ tiene hijo derecho. Luego reemplazamos $x$ por $y$ y eliminamos el nodo $x$ el cual ahora está en la posición de $y$. No obstante, hay algunos pormenores que dan nacimiento a dos casos:

   (a) **El sucesor $y$ es hijo derecho de $x$**: como $y$ es el sucesor de $x$, entonces sabemos que $y$ no tiene hijo izquierdo (ver definición de SUCCESSOR). Hacemos que el padre de $y$ sea el padre de $x$ y que el hijo izquierdo de $y$ sea el hijo izquierdo de $z$ (y también actualizamos los punteros del padre de $x$ para que ahora su hijo sea $y$ y el puntero del hijo izquierdo de $x$ para que su padre sea $y$).

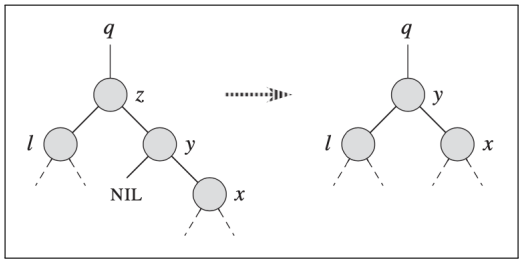



Figure 13: Notar que en este ejemplo $z$ es $x$.

   Esto funciona por que dado que $l < x$ y $y > x$, si $l$ se convierte en hijo izquierdo de $y$, entonces $l < y$ y se sigue cumpliendo la propiedad. Sea $x$ hijo derecho o izquierdo, esto se mantiene.

   (b) **El sucesor $y$ no es hijo derecho de $x$**: encontramos el sucesor $y$ de $x$ el cual está en el subárbol derecho de $x$, ya que sabemos que $x$ tiene hijo derecho. Luego, reemplazamos $x$ por $y$ y borramos $x$ (el cual ahora está en la posición de $y$), y como $x$ no tiene hijo izquierdo (dado que $y$ no lo tenía porque es el sucesor) aplica el caso 2 de remover (cuando el nodo solo tiene 1 hijo).

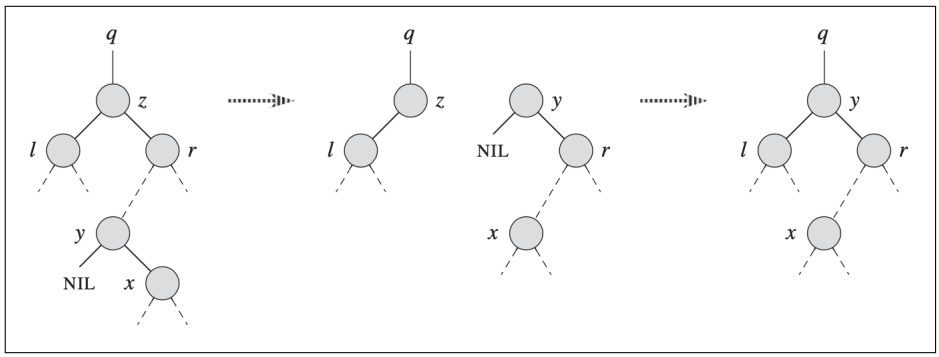



Figure 14: Notar que en este ejemplo $z$ es $x$ y que las aristas sólidas representan hijos directos y las aristas puntadas significan que pueden haber más nodos y aristas ahí.

*La explicación de por qué funciona este algoritmo queda de ejercicio. La pista ya la pueden haber adivinado: apoyarse de la propiedad de los árboles binarios de búsqueda.

# 4 Palabras Finales

- Se mencionó que la altura de un arbol binario de búsqueda es **en el mejor caso** logarítmica. Esto es porque los algoritmos de inserciones y deleciones que hemos visto, no se aseguran de que la altura del árbol se mantenga logarítmica. Como están las operaciones de mutación ahorita, se podría insertar 1, 2, 3, 4, 5, 6, 7, 8, 9, ... y el árbol binario de búsqueda resultante sería básicamente una lista enlazada simple y por lo tanto la búsqueda correría en $\mathcal{O}(n)$, la inserción en $\mathcal{O}(n)$, la deleción también en $\mathcal{O}(n)$, etc. Es por ello que existen árboles que se balancean al cabo de cada operación de mutación. Por ejemplo, los árboles AVL que veremos más adeltante modifican el algoritmo de inserción y deleción para que la altura del árbol se mantenga balanceado y por lo tanto todas sus operaciones corran el $\mathcal{O}(\lg n)$. Otro árbol que usa otros algoritmos para obtener el mismo resultado es el Red-Black Tree.

- La complejidad de cada caso de delete varía dependiendo de qué recibe como input la rutina DELETE. Si el DELETE recibe un puntero al nodo que se desea borrar entonces los casos 1 y 2 corren el $\mathcal{O}(1)$ porque solo se requiere actualización de punteros, mientras que el caso 3 correría en $\mathcal{O}(h)$, donde $h$ es la altura del árbol, dado que todo el trabajo sería actualización de punteros ($\mathcal{O}(1)$) excepto por la llamada a MINIMUM o MAXIMUM, los cuales corren en $\mathcal{O}(h)$. Esto da una complejidad resultante de $\mathcal{O}(h)$ para el DELETE. Sin embargo, si se recibe el valor del nodo que se desea eliminar en vez de un puntero a ese nodo, entonces cada caso corre en $\mathcal{O}(h)$ dado

que primero se tiene que buscar con SEARCH dicho nodo y luego aplicar el algoritmo de DELETE. Esto también da una complejidad resultante de $\mathcal{O}(h)$.

# 5 Referencias


- http://mate.cucei.udg.mx/matdis/6arb/6arb2.htm
- Introduction to Algorithms (Cormen) Capítulo 12: https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf