CI2612: Algoritmos y Estructuras de Datos II

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Árboles binarios de búsqueda

© 2017 Blai Bonet

Objetivos

- Árboles binarios de búsqueda
- Operaciones sobre árboles: query y modificación

© 2017 Blai Bonet

Árbol binario de búsqueda

Un árbol binario de búsqueda es una ED que implementa un conjunto dinámico y soporta las siguientes operaciones:

- `Search`
- `Minimum`
- `Maximum`
- `Predecessor`
- `Successor`
- `Insert`
- `Delete`

Como su nombre lo indica, los elementos son almacenados en un árbol binario. Asumimos que las claves obedecen un **orden total**

Las operaciones toman en el peor caso tiempo proporcional a la **altura del árbol**. Si el árbol es balanceado, su altura es $O(\log n)$ pero el árbol puede tener una estructura lineal y su altura ser $O(n)$, donde n es el número de elementos en el árbol

© 2017 Blai Bonet

Ejemplo de árbol binario de búsqueda

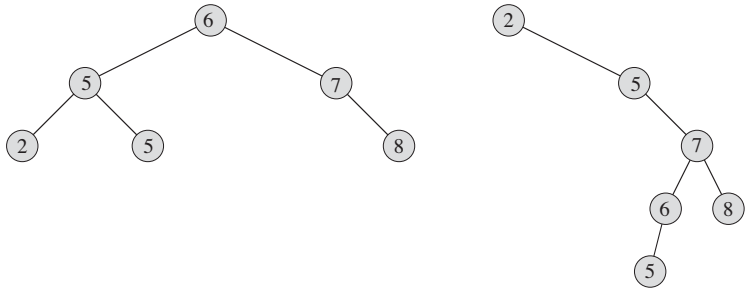

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2017 Blai Bonet

Árbol binario de búsqueda

Cada nodo del árbol almacena una clave junto a un apuntador a los datos satélites asociados a la clave

La **propiedad** que define al árbol como **árbol de búsqueda** es:

Para todo nodo x en el árbol:

- *si y es un nodo cualquiera en el subárbol izquierdo de x, entonces $y.key < x.key$*
- *si z es un nodo cualquiera en el subárbol derecho de x, entonces $x.key \leq z.key$*

Los nodos internos del árbol no están obligados a tener ambos hijos

© 2017 Blai Bonet

Otro ejemplo de árbol binario de búsqueda

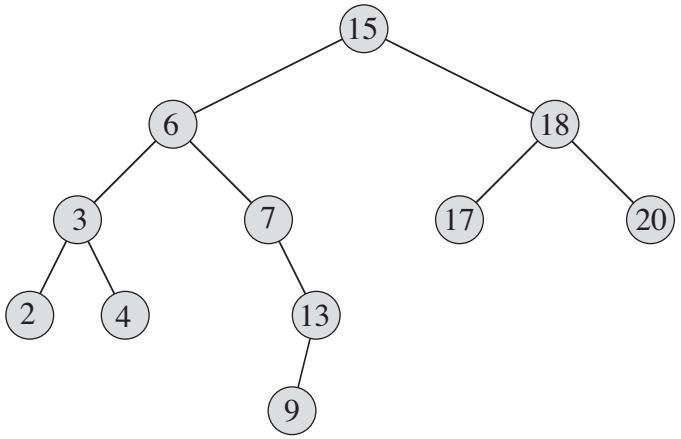

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2017 Blai Bonet

Recorrido del árbol

El simple hecho de que las claves satisfacen la propiedad de árbol de búsqueda permite imprimir todas las claves en el árbol de forma ordenada en tiempo $\Theta(n)$ con un **algoritmo recursivo**

```
1  Inorder-Tree-Walk(pointer x)
2      if x != null
3          Inorder-Tree-Walk(x.left)
4          Imprimir x.key
5          Inorder-Tree-Walk(x.right)
```

Esto es un **recorrido inorden**. Un **recorrido preorden** imprime la clave antes de visitar los hijos, y un **recorrido postorden** imprime la clave luego de hacer el recorrido de los hijos

© 2017 Blai Bonet

Operaciones query sobre árboles de búsqueda

Recuerde que una operación tipo query es una operación que **no cambia** el árbol. Las siguientes operaciones son tipo query:

- `Search`
- `Minimum`
- `Maximum`
- `Predecessor`
- `Successor`

© 2017 Blai Bonet

Operación de búsqueda de claves

Dada una clave k, tenemos que determinar si el árbol contiene dicha clave. En caso afirmativo, debemos devolver un apuntador al nodo que la contiene, y null si la clave no se encuentra en el árbol

```
1  Tree-Search(pointer x, key k)
2      if x == null || x.key == k then return x
3
4      if k < x.key
5          return Tree-Search(x.left, k)
6      else
7          return Tree-Search(x.right, k)
```

La primera llamada es con x igual a un apuntador a la raíz del árbol. La búsqueda toma tiempo $O(h)$ donde h es la altura del árbol

© 2017 Blai Bonet

Máximos y mínimos

Por la propiedad de árbol de búsqueda, la clave mínima debe ser la clave "más a la izquierda" mientras que la clave máxima debe ser la clave "más a la derecha"

```
1  Tree-Minimum(pointer x)
2      while x != null && x.left != null
3          x = x.left
4      return x
5
6  Tree-Maximum(pointer x)
7      while x != null && x.right != null
8          x = x.right
9      return x
```

La primera llamada es con x igual a un apuntador a la raíz del árbol. Ambas operaciones toman tiempo $O(h)$ donde h es la altura del árbol

© 2017 Blai Bonet

Sucesor

Si todas las claves son distintas, el sucesor del nodo x con clave $x.key$ es el nodo que contenga la **menor clave** entre todos los nodos que tienen **clave mayor estricta** a $x.key$ (i.e. a la derecha de x)

```
1  Tree-Successor(pointer x)                          % se asume x != null
2      % caso de subárbol derecho no vacío
3      if x.right != null then return Tree-Minimum(x.right)
4
5      % caso de subárbol derecho vacío
6      y = x.p                                        % y es el padre de x
7      while y != null && x == y.right    % mientras x sea hijo derecho
8          x = y
9          y = y.p
10     return y
```

`Tree-Successor` toma tiempo $O(h)$ donde h es la altura del árbol

© 2017 Blai Bonet

Correctitud del cálculo del sucesor (1 de 2)

Suponga que x tiene subárbol derecho

Por la propiedad de árbol de búsqueda:

- Todo nodo en el subárbol derecho de x tiene clave $\geq x.key$
- ¿Existen otras claves mayores a $x.key$? ¿Cuáles?

  Si x es un **descendiente por hijo izquierdo** de un nodo y, entonces $x.key < y.key$ y todo nodo que esté en el subárbol derecho de y también tendrá clave mayor estricta a $x.key$

  Sin embargo, $y.key$ y **todas las claves** en $y.right$ tendrán claves mayores estrictas a todas las claves en el subárbol derecho de x

Como el sucesor es el menor de todas las claves mayores a $x.key$, el sucesor es **el menor en** $x.right$

Esto establece la correctitud cuando x tiene subárbol derecho

© 2017 Blai Bonet

Correctitud del cálculo del sucesor (2 de 2)

Ahora suponga que x no tiene subárbol derecho

Como vimos $x.key < y.key$ ssi:

- x es un **descendiente** de y por hijo izquierdo, ó
- x **desciende** de un nodo z por hijo izquiero y y **pertenece** a $z.right$. En este caso, $x.key < z.key \leq y.key$

Sea A el conjunto de ys tal que x desciende de y por hijo izquierdo:

- Si $A = \emptyset$, entonces x no tiene sucesor
- Si $A \neq \emptyset$, el sucesor de x es el mínimo en A
- Para todo $y, z \in A$, y desciende de z por hijo izquierdo ó vice versa. En el primer caso, $y.key < z.key$
- **Conclusión:** El sucesor de x es el nodo en A **"mas cercano"** a x

Esto establece la correctitud cuando x no tiene subárbol derecho

© 2017 Blai Bonet

Inserción

La operación de inserción modifica el árbol añadiendo un nodo nuevo

La modificación debe ser tal que la propiedad de árbol de búsqueda se mantiene en el nuevo árbol

El procedimiento de inserción **recibe** un nodo z con la nueva clave a ser insertada y con $z.left = z.right =$ null

Una vez insertado, z aparece como **hoja del árbol**

© 2017 Blai Bonet

Idea para la inserción

La idea es realizar una búsqueda de la clave $z.key$ a ser insertada para así determinar quién debe ser el **padre** del nuevo elemento z

Una vez determinado el padre y, el nodo z es insertado como hijo izquierdo o derecho dependiendo si $z.key < y.key$ ó $y.key \leq z.key$

Si el árbol es vacío, el nodo z pasa a ser la raíz del árbol

© 2017 Blai Bonet

Ejemplo de inserción en árbol binario de búsqueda

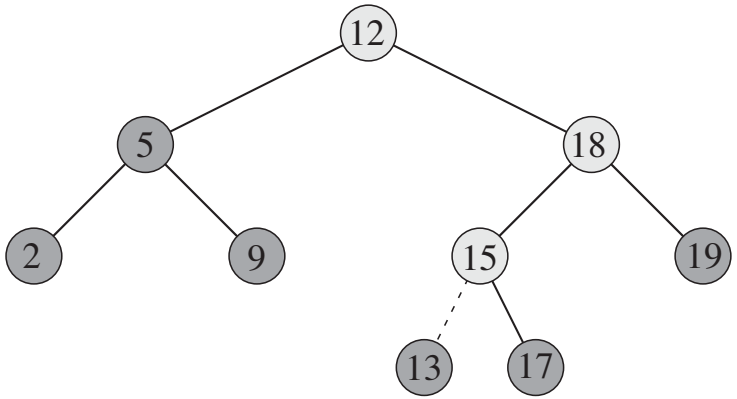

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2017 Blai Bonet

Inserción

```
1  Tree-Insert(tree T, pointer z)
2      y = null
3      x = T.root
4      while x != null                          % buscamos donde insertar
5          y = x
6          if z.key < x.key
7              x = x.left
8           else
9              x = x.right
10
11     % asignar y como padre del nuevo nodo z
12     z.p = y
13     if y == null                                % el árbol está vacío
14         T.root = z
15     else if z.key < y.key
16         y.left = z
17     else
18         y.right = z
```

La inserción toma tiempo $O(h)$ donde h es la altura del árbol

© 2017 Blai Bonet

Eliminación

Al eliminar un nodo z del árbol, consideramos 3 casos:

1. Si z es una hoja, la eliminación es fácil ya que solo debemos cambiar un apuntador para remover z

2. Si z tiene un solo hijo, **"elevamos"** el hijo para que reemplace a z

3. Si z tiene dos hijos, buscamos el sucesor y de z (el cual está en $z.right$) subárbol derecho de z) para que reemplace a z manteniendo la propiedad del árbol de búsqueda

   Diferenciamos cuando y es un hijo **derecho** o **izquierdo**. Cuando y es hijo derecho, y tiene que ser **necesariamente** el hijo derecho de z

   En ambos caso, el sucesor y de z **no tiene hijo izquierdo**

© 2017 Blai Bonet

Segundo caso de eliminación

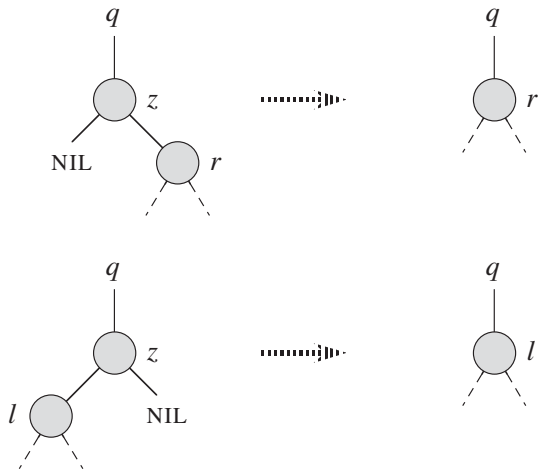

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2017 Blai Bonet

Tercer caso de eliminación

El sucesor y de z es un hijo derecho:

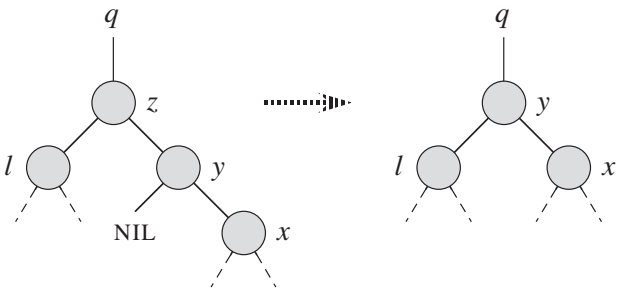

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2017 Blai Bonet

Tercer caso de eliminación

El sucesor y de z es un hijo izquierdo:

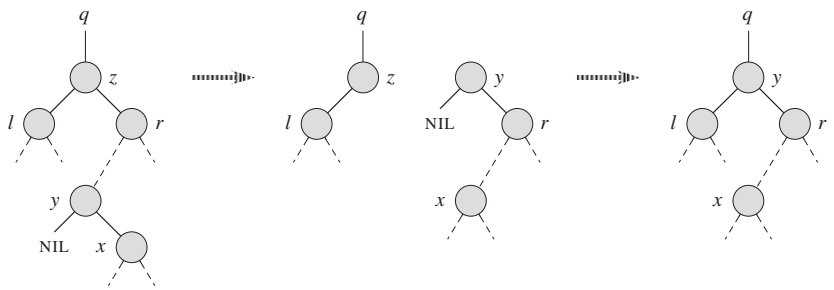

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2017 Blai Bonet

Transplante de nodos

El reemplazo de un nodo u por otro nodo v en el árbol T se implementa con la subrutina `Transplant(T, u, v)`:

```
1  Transplant(tree T, pointer u, pointer v)
2      if u.p == null                                  % u es nodo raíz
3          T.root = v
4      else if u == u.p.left                           % u es hijo izquierdo
5          u.p.left = v
6      else                                            % u es hijo derecho
7          u.p.right = v
8
9      % modificar apuntador del padre en nodo transplantado
10     if v != null then v.p = u.p
```

Esta rutina corre en **tiempo constante**

© 2017 Blai Bonet

Eliminación

```
1  Tree-Delete(tree T, pointer z)
2      if z.left == null
3          % caso 1 y 2 (z sin subárbol izquierdo)
4          Transplant(T, z, z.right)
5      else if z.right == null
6          % caso 2 (z sin subárbol derecho)
7          Transplant(T, z, z.left)
8      else
9          % caso 3 (z tiene ambos hijos)
10         y = Tree-Minimum(z.right)                   % y es sucesor de z
11         if y.p != z
12             % caso 3: y es hijo izquierdo
13             Transplant(T, y, y.right)
14             y.right = z.right
15             y.right.p = y
16         Transplant(T, z, y)
17         y.left = z.left
18         y.left.p = y
```

`Tree-Delete(T,z)` corre en tiempo $O(h)$ porque llama a `Tree-Minimum`

© 2017 Blai Bonet

Resumen

- Los árboles binarios de búsqueda son una ED que permite implementar todas las operaciones sobre conjuntos dińamicos en tiempo proporcional a la altura del árbol
- Si la altura es pequeña, $O(\log n)$, un árbol binario de búsqueda es una ED eficiente
- Sin embargo, en el peor caso, un árbol con n elementos puede tener altura $O(n)$
- Más adelante veremos los **árboles rojo y negro** que son un tipo de árbol binario de búsqueda en donde la altura está garantizada de ser $O(\log n)$

© 2017 Blai Bonet

Ejercicios (1 de 7)

1. Calcule la complejidad de `Inorder-Tree-Walk` planteando una recurrencia y resolviendo con sustitucion
2. (12.1-1) Considere las claves $\{1, 4, 5, 10, 16, 17, 21\}$. Dibuje árboles binarios de búsqueda de alturas 2, 3, 4, 5 y 6
3. (12.1-2) ¿Cuál es la diferencia entre las propiedades para árbol binario de búsqueda y min-heap? ¿Se puede utilizar la propiedad de min-heap para imprimir las claves de forma ordenada en tiempo $O(n)$ (donde n es el número de nodos en el heap)? Justifique
4. (12.1-3) Escriba un algoritmo iterativo que realice un recorrido inorden del árbol (Ayuda: una posible solución utiliza una pila para simular la recursión. Un mejor solución no utiliza pilas)

© 2017 Blai Bonet

Ejercicios (2 de 7)

5. (12.1-4) Implemente recorridos preorden y postorden sobre un árbol de búsqueda
6. (12.1-5) Argumente que como para ordenar n elementos se requieren de $\Omega(n \log n)$ comparaciones, entonces cualquier algoritmo que construya un árbol de búsqueda a partir de n claves debe tomar tiempo $\Omega(n \log n)$ en el peor caso
7. Argumente la correctitud de `Tree-Search`
8. Escriba una versión iterativa de `Tree-Search`
9. (12.2-2) Escriba versiones recursivas de `Tree-Minimum` y `Tree-Maximum`
10. (12.2-3) Escriba el procedimiento `Tree-Predecessor`

© 2017 Blai Bonet

Ejercicios (3 de 7)

11. (12.2-1) Suponga que tenemos un árbol binario de búsqueda cuyas claves están entre 1 y 1000, y que realizamos una búsqueda del elemento 363 sobre el árbol. ¿Cuál de las siguientes secuencias de nodos examinados durante la búsqueda es imposible?
    a) 2, 252, 401, 398, 330, 344, 397, 363
    b) 924, 220, 911, 244, 898, 258, 362, 363
    c) 925, 202, 911, 240, 912, 245, 363
    d) 2, 399, 387, 219, 266, 382, 381, 278, 363
    e) 935, 278, 347, 621, 299, 392, 358, 363
12. (12.2-5) Muestre que si un nodo en un árbol de búsqueda tiene dos hijos, entonces su sucesor no tiene hijo izquierdo y su predecesor no tiene hijo derecho

© 2017 Blai Bonet

Ejercicios (4 de 7)

13. (12.2-7) Una forma alternativa de realizar un recorrido inorden de un árbol con n elementos consiste en primero llamar a `Tree-Minimum` y luego realizar $n-1$ llamadas a `Tree-Successor`. Muestre que este algoritmo corre en tiempo $\Theta(n)$

14. (12.2-8) Muestre que dado un nodo x en un árbol binario de búsqueda de altura h, si realizamos k llamadas a `Tree-Successor` partiendo desde x, para generar los k sucesores de x, el tiempo total es $\Theta(h+k)$

15. (12.2-9) Sea T un árbol binario de búsqueda donde todas las claves son distintas, x una hoja de T y y el padre de x. Muestre que $y.key$ es la menor clave en T que es mayor a $x.key$ ó $y.key$ es la mayor clave en T que es menor a $x.key$

16. (12.3-1) Escriba una versión recursiva de `Tree-Insert`

© 2017 Blai Bonet

Ejercicios (5 de 7)

17. (12.3-2) Suponga que construimos un árbol binario de búsqueda T realizando inserciones repetidas y que no se elimina ninguna clave del árbol. Argumente que el número de nodos examinados durante una busqueda de una clave k en el árbol es igual a 1 mas el número de nodos examinados cuando la clave k se insertó por primera vez

18. (12.3-3) n elementos pueden ser ordenados si primero construimos un árbol binario de búsqueda realizando n inserciones y luego hacemos un recorrido inorden del árbol. Calcule el tiempo de corrida de este algoritmo de ordenamiento en el mejor y peor caso

19. (12.3-4) ¿És la operación de eliminación commutativa? Es decir, ¿és igual eliminar x y luego y igual a primero eliminar y y luego x?

20. (12.3-6) Cuando el nodo z a eliminar en `Tree-Delete` tiene dos hijos, se puediera elegir a y como el predecesor de z en lugar de su sucesor. Implemente esta estrategia, y una estrategia que randomiza la selección del predecesor y sucesor cuando el nodo z a eliminar tenga dos hijos

© 2017 Blai Bonet

Ejercicios (6 de 7)

21. (12-2) Árboles de prefijos

    Dadas cadenas $a = a_0 \dots a_p$ y $b = b_0 \dots b_q$, donde cada a_i y b_j están en conjunto ordenado, decimos que a es **lexicográficamente menor** a b ssi:

    a) existe entero j, con $0 \le j \le \min\{p, q\}$, tal que $a_j < b_j$ y $a_i = b_i$ para todo i con $0 \le i < j$, ó

    b) $p < q$ y $a_i = b_j$ para todo $0 \le i \le p$.

    Por ejemplo, si hablamos de cadenas de bits, $10100 < 10110$ y $10100 < 101000$. El árbol de prefijos mostrado a continuación almacena los strings 1011, 10, 011, 100 y 0

    Al buscar una clave $a = a_0 \dots a_p$, nos movemos a la izquierda en un nodo a profundidad i si $a_i = 0$, y nos movemos a la derecha si $a_i = 1$

    Sea S un conjunto de cadenas, y n la suma de las longitudes de las cadenas en S. Muestre como usar un árbol de prefijos para ordenar las cadenas en S en tiempo $\Theta(n)$. Para el ejemplo de la figura, la salida debería ser 0, 011, 10, 100, 1011

© 2017 Blai Bonet

Ejercicios (7 de 7)

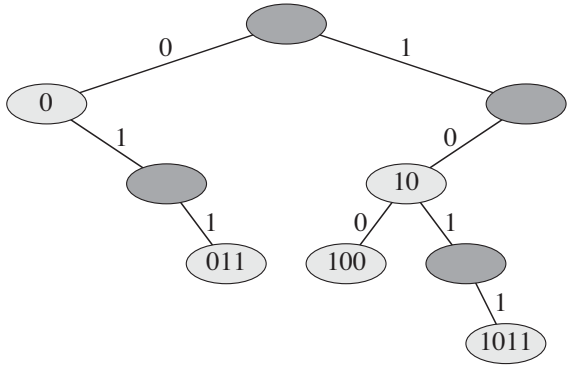

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Árbol de prefijos para 1011, 10, 011, 100 y 0. Las cadenas almacenadas se corresponden con los nodos claros. En tales nodos no hace falta guardar la cadenas ya que ellas está dadas por los caminos desde la raíz hasta los nodos

© 2017 Blai Bonet