Algoritmos y Estructuras de Datos
UT4-PD5

Santiago Blanco

10-05-2025

Ejercicio #1

Obtener menor clave

Precondiciones:

- El árbol no está vacío
- Es un árbol binario de búsqueda válido
- Cada nodo contiene una clave comparable
- Las claves deben ser comparables entre sí

Postcondiciones:

- Devuleve un elemento (clave) comparable no nula
- El elemento comparable 'clave' devuelto es menor que todos los demás en el árbol
- El árbol no se modifica

Lenguaje natural

El algoritmo inicia desde la raíz, y busca avanzar siempre por el hijo izquierdo del nodo, cuando se encuentra con que el hijo izquierdo del nodo es nulo, significa que llegó al nodo 'más a la izquierda' del árbol, es decir el más chico, y lo devuelve.

Pseudocódigo

```
menorClave() -> Comparable
COM
  SI hijoIzq == nulo ENTONCES
    RETORNAR etiqueta
  SINO
    RETORNAR hijoIzq.menorClave()
  FIN SI
FIN
```

Orden de tiempo de ejecución

El algoritmo recorre el árbol buscando el nodo más a la izquierda, por lo que el tiempo de ejecución depende de la altura del árbol -> $O(n)$

Obtener mayor clave

⚠️ Nótese que este algoritmo comparte toda la lógica con el anterior, por lo que resultan idénticos pero con sutiles modificaciones (copié y pegué todo)

Precondiciones:

- El árbol no está vacío
- Es un árbol binario de búsqueda válido
- Cada nodo contiene una clave comparable
- Las claves deben ser comparables entre sí

Postcondiciones:

- Devuleve un elemento (clave) comparable no nula

- El elemento comparable 'clave' devuelto es mayor que todos los demás en el árbol
- El árbol no se modifica

Lenguaje natural

El algoritmo inicia desde la raíz, y busca avanzar siempre por el hijo derecho del nodo, cuando se encuentra con que el hijo derecho del nodo es nulo, significa que llegó al nodo 'más a la derecha' del árbol, es decir el más grande, y lo devuelve.

Pseudocódigo

```
1 mayorClave() -> Comparable
2 COM
3   SI hijoDer == nulo ENTONCES
4     RETORNAR etiqueta
5   SINO
6     RETORNAR hijoIzq.mayorClave()
7   FIN SI
8 FIN
```

Orden de tiempo de ejecución

El algoritmo recorre el árbol buscando el nodo más a la derecha, por lo que el tiempo de ejecución depende de la altura del árbol -> $O(n)$

Obtener clave anterior

Precondiciones:

- El árbol no está vacío
- Es un árbol binario de búsqueda válido
- Cada nodo contiene una clave comparable
- Las claves deben ser comparables entre sí
- La clave que recibe no es nula

Postcondiciones:

- Si encontró la clave, devuleve un elemento (clave) comparable
  - El elemento comparable 'clave' devuelto es la clave del padre del nodo que contiene la etiqueta buscada.
- Si no encontró la clave o la clave está en la raíz, devuelve NULL
- El árbol no se modifica.

Lenguaje natural

Similar al algoritmo de búsqueda convencional. Primero verifica si el elemento buscado es igual al del hijo izquierdo del nodo actual. En caso positivo, retorna la clave del nodo actual (que es el padre). Repite el mismo proceso con el hijo izquierdo. Si no encontró la etiqueta en los hijos directos del nodo, busca recursivamente como en una búsqueda binaria.

Pseudocódigo

```
1 buscarAnterior(etiq: Comparable) -> Comparable
2 COM
3   SI hijoIzq <> NULO Y etiq = hijoIzq.etiqueta ENTONCES
4     RETORNAR etiqueta
5   FIN SI
6
```

```
  SI hijoDer <> NULO Y etiq = hijoDer.etiqueta ENTONCES
    RETORNAR etiqueta
  FIN SI

  SI etiq < etiqueta ENTONCES
    SI hijoIzq <> NULO ENTONCES
      RETORNAR hijoIzq.buscarAnterior(etiq)
    SINO
      RETORNAR NULO
    FIN SI
    SINO
      SI hijoDer <> NULO ENTONCES
          RETORNAR hijoDer.buscarAnterior(etiq)
      SINO
          RETORNAR NULO
      FIN SI
    FIN SI
  FIN SI
FIN
```

Orden de tiempo de ejecución

El algoritmo ejecuta una búsqueda binaria donde su tiempo de ejecución es de orden logarítmico $O(\log n)$ si el árbol está balanceado. Si no está balanceado y se degenera en una lista enlazada, entonces se trata de orden lineal $O(n)$.

Cantidad de nodos por nivel

Lenguaje natural

- Se recibe como parámetros el árbol (o nodo raiz) y el nivel donde se desea contar la cantidad de nodos.
- Se inicia desde el nodo raíz, y se verifica recursivamente la cantidad de nodos por nivel en cada subárbol. Se recorre el árbol en POSTORDEN, porque visita primero el subárbol izquierdo, luego el derecho y finalmente procesa el nodo actual.
- En cada llamada se disminuye el nivel hasta que se llegue al nivel objetivo →CASO BASE
- Al final se retorna la suma de la cantidad de nodos de cada subárbol

Precondiciones

- Se recibe un árbol binario válido y no vacío
- Se recibe un nivel de tipo entero y mayor o igual a cero

Postcondiciones

- Se retorna la cantidad de nodos en el nivel especificado
- Si el nivel que recibe como parámetro es mayor que el nivel máximo del árbol, se retorna 0

Pseudocódigo

```
contarNodosNivel(nodo: TElemento<T>, nivel: int) -> int
COM
  SI nodo == nulo ENTONCES
    RETORNAR 0 // O(1)
  FIN SI

```

```
  nodosIzq <- contarNodosNivel(nodo.izq, nivel -1)
  nodosDer <- contarNodosNivel(nodo.der, nivel -1)

  SI nivel == 0 ENTONCES
    RETORNAR 1 // O(1)
  FIN SI

  RETORNAR nodosIzq + nodosDer // O(1)
FIN
```

Análisis de tiempo de ejecución

Dado que el algoritmo recorre todos los nodos en Postorden como los anteriores, se trata de un tiempo de ejecución de orden líneal $\longrightarrow O(n)$

Listar hojas con nivel

Lenguaje Natural

El algoritmo recibe como parámetro un nodo y un número de nivel entero. Sus casos base son:
1. Si el nodo es nulo, no hace nada y termina el algoritmo
2. Si el nodo es hoja, imprime por pantalla la etiqueta del nodo junto con su nivel.

El algoritmo recorre el árbol en preorden y en cada llamada a sí mismo aumenta en 1 el nivel.

Precondiciones
- El algoritmo se ejecuta sobre un árbol binario válido.
- Los nodos hoja referencian a NULL tanto en su hijo derecho como izquierdo

Postcondiciones
- Identifica las hojas como aquellos nodos que no tienen hijos, es decir que, en sus valores izquierdo y derecho referencian a NULL.
- Imprime la etiqueta correspondiente a todos los nodos hoja junto con su nivel (profundidad) en el árbol, siendo 0 el nivel del nodo raíz y aumentando a medida que se avanza sobre los hijos de los nodos.

```
listarHojas(nodo: TElemento<T>, nivel: int)
COM
  SI nodo == NULL ENTONCES
    RETORNAR // O(1)
  FIN SI

  SI nodo.izq == NULL Y nodo.der == NULL ENTONCES
    IMPRIMIR nodo.etiqueta + " - " + nivel // O(1)
    RETORNAR // O(1)
  FIN SI

  listarHojas(nodo.izq, nivel+1)
  listarHojas(nodo.der, nivel+1)
FIN
```

Análisis de tiempo de ejecución

El algoritmo recorre todo el árbol en preorden, buscando los nodos sin hijos. En consecuencia, tiene un tiempo de ejecución lineal $\rightarrow O(n)$.

Verificar si un árbol es de búsqueda

Lenguaje Natural

Obtiene mediante un método auxiliar la lista de nodos del árbol en inorden y los almacena en un array. Posteriormente recorre la lista y si detecta que un nodo es menor que el anterior, retorna falso indicando que el árbol binario no es de búsqueda. Caso contrario, retorna verdadero.

Precondiciones

- Se ejecuta sobre un árbol binario válido

Postcondiciones

- Retorna verdadero si
  1. Todos los nodos en el subárbol izquierdo tienen valores menores que el valor del nodo raíz.
  2. Todos los nodos en el subárbol derecho tienen valores mayores que el valor del nodo raíz.
- Retorna falso si no se cumplen dichas condiciones.
- El árbol no se modifica

Pseudocódigo

```
esDeBusqueda() -> boolean
COM
  elementosInorden <- Nuevo ArrayList O(1)
  inOrden(elementosInorden) // O(n)

  cont <- 1
  MIENTRAS cont < elementosInorden.size() HACER // O(n)
    SI elementos[cont] <= elementos[cont-1] ENTONCES // O(1)
      RETORNAR FALSO // O(1)
    FIN SI
  FIN MIENTRAS

  RETORNAR VERDADERO // O(1)
FIN

-----------------------------------------
// METODO AUXILIAR - RECORRIDA INORDEN

inOrden(unaLista: ArrayList)
COM
  SI hijoIzq <> NULO ENTONCES // O(1)
    hijoIzq.inOrden(unaLista)
  FIN SI

  unaLista.add(this.getDatos()) // O(1)

  SI hijoDer <> NULO ENTONCES // O(1)
    hijoDer.inOrden(unaLista)
  FIN SI
FIN
```

Análisis de tiempo de ejecución

El método auxiliar y recursivo **inOrden()** tiene un tiempo de ejecución de orden $O(n)$, siendo n la cantidad de nodos del árbol. Dado que el método principal llama a este para luego recorrer la lista que modifica, el orden total es $O(2n) = O(n)$.