## **Evaluación asincrónica 2** de Estrategias Algorítmicas, ITESO, Luis Gatica

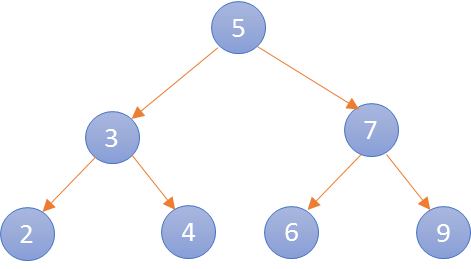
Nombre: Rodríguez Castro, Carlos Eduardo; Cordero Hernández, Marco Ricardo

# Parte 2: árboles

Para los siguientes ejercicios, asuma que la clase correspondiente a cada árbol contiene el tamaño (número de llaves actualmente en el árbol) y una referencia al nodo raíz. Considere que para cada tipo de árbol además de la clase árbol es necesaria una clase nodo. Los valores que almacenarán los nodos son enteros.

1. **Árboles Binarios de Búsqueda (ABB)**

1) Crea una función(es) que determine(n) si existe una ruta de la raíz a alguna hoja tal que el número de claves impares de los nodos visitados es igual a *x*.

Así se usaría el método:

boolean exists1 = *existsPath*(root, 1); // exists1 = false

boolean exists2 = *existsPath*(root, 2); // exists2 = true

boolean exists3 = *existsPath*(root, 3); // exists3 = true

boolean exists4 = *existsPath*(root, 4); // exists4 = false

private boolean **countOdd**(Node<Integer> *node*, int *goal*, int *count*) {

    if (*node*.key % 2 != 0) *count*++;

    if (*node*.left == null && *node*.right == null) {

        if (*count* == *goal*) return true;

        else return false;

    }

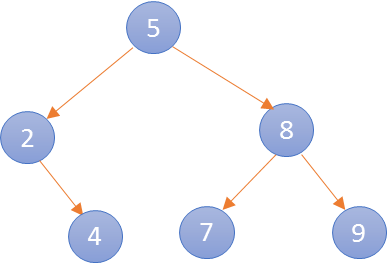
    return **countOdd**(*node*.left, *goal*, *count*) || **countOdd**(*node*.right, *goal*, *count*);

}

public boolean **existsPath**(Node<Integer> *root*, int *goal*) {

    return **countOdd**(*root*, *goal*, 0);

}

2) Crea una función(es) que determine(n) si un árbol binario es de búsqueda.

Así se usaría el método:

boolean isBST1 = *isBST*(root); // isBST1 = true

Si cambiamos, por ejemplo, a 7 por 3, ó a 4 por 1:

boolean isBST2 = *isBST*(root); // isBST2 = false

private Node<T> **maximum**(Node<T> *node*) {

    if (*node* == null) return null;

    while (*node*.right != null) *node* = *node*.right;

    return *node*;

}

public T **maximum**() {

    Node<T> max = **maximum**(root);

    return (max == null) ? null : max.key;

}

private Node<T> **minimum**(Node<T> *node*) {

    if (*node* == null) return null;

    while (*node*.left != null) *node* = *node*.left;

    return *node*;

}

public T **minimum**() {

    Node<T> min = **minimum**(root);

    return (min == null) ? null : min.key;

}

public boolean **isBST**(Node<T> *root*) {

    if (*root* == null) return true;

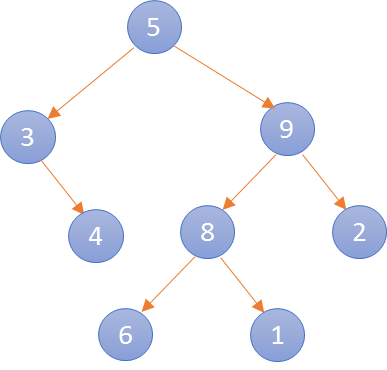
    if (*root*.left != null && **maximum**(*root*.left).key.**compareTo**(*root*.key) >= 0 ||

*root*.right != null && **minimum**(*root*.right).key.**compareTo**(*root*.key) <= 0 ||

    !**isBST**(*root*.left) || !**isBST**(*root*.right)) return false;

    return true;

}

3) Crea una función(es) que construya(n) una lista de listas, tal que la lista *k* contiene las claves de todos los nodos que viven el nivel *k*, de izquierda a derecha.

Así se usaría el método:

List<List<Node>> nodeList = *getNodeListPerLevel*(root);

// nodeList = [ [5], [3, 9], [4, 8, 2], [6, 1] ]

int **depthBST**(Node *root*) {

    if (*root* == null) return -1;

    int lDepth = **depthBST**(*root*.left);

    int rDepth = **depthBST**(*root*.right);

    if (lDepth > rDepth) return lDepth + 1;

    else return rDepth + 1;

}

void **getNodesOnLevel**(Node<T> *node*, int *level*, List<T> *list*) {

    if (*node* == null) return;

    if (*level* == 0) {

*list*.**add**(*node*.key);

    } else {

**getNodesOnLevel**(*node*.left, *level* - 1, *list*);

**getNodesOnLevel**(*node*.right, *level* - 1, *list*);

    }

}

List<T> **getNodesOnLevel**(Node<T> *root*, int *level*) {

    List<T> list = new **ArrayList**<>();

**getNodesOnLevel**(*root*, *level*, list);

    return list;

}

List<List<Node>> **getNodeListPerLevel**(Node *root*) {

    int depth = **depthBST**(*root*);

    List<List<Node>> res = new **ArrayList**<>(depth);

    for (int i = 0; i <= depth; i++) {

        res.**add**(**getNodesOnLevel**(*root*, i));

    }

    return res;

}

1. **Árboles AVL**

Cada nodo contiene su dato o llave, referencia a sus hijos izquierdo y derecho, y un entero que representa la altura del nodo.

Escriba un método (puede escribir otros más y llamarlos) que reciba el árbol y determine si se cumplen todas las propiedades debidas:

1. A la izquierda de cada nodo sólo hay valores menores y a la derecha mayores. (O hijos nulos, en su defecto.)
2. El factor de balance de todos los nodos pertenece a {-1, 0, 1}

public class AVL<T extends Comparable<? super T>> {

    private int **height**(Node<T> *root*) {

        if (*root* == null) return -1;

        int hl = **height**(*root*.left);

        int hr = **height**(*root*.right);

        if (hl > hr) return hl + 1;

        else return hr + 1;

    }

    private boolean **isBalanced**(Node<T> *n*) {

        if (*n* == null) return true;

        int hl = **height**(*n*.left), hr = **height**(*n*.right);

        if (Math.**abs**(hl - hr) <= 1

            && **isBalanced**(*n*.left) && **isBalanced**(*n*.right))

            return true;

        return false;

    }

    public boolean **checkAVL**(Node<T> *root*) {

        if (*root* == null) return true;

*// Primera condición: BST*

        CheckBST<T> helper = new **CheckBST**<>(*root*);

        if (!(helper.**isBST**(*root*))) return false;

*// Segunda condición: balance*

        if (!(**isBalanced**(*root*))) return false;

        return true;

    }

}

1. **Árboles rojinegros**

Cada nodo contiene su dato o llave, referencia a sus hijos izquierdo y derecho, y un booleano que representa si el nodo es rojo.

Escriba un método (puede escribir otros más y llamarlos) que reciba el árbol y determine si se cumplen todas las propiedades debidas:

1. A la izquierda de cada nodo sólo hay valores menores y a la derecha mayores. (O hijos nulos, en su defecto.)
2. Propiedad raíz (*root property* en las diapositivas).
3. Propiedad roja
4. Propiedad negra

public class ARN<T extends Comparable<? super T>> {

    private boolean **redProperty**(NodeRN<T> *node*) {

        if (*node* == null) return true;

        if (*node*.left != null && *node*.isRed && *node*.left.isRed) return false;

        if (*node*.right != null && *node*.isRed && *node*.right.isRed) return false;

        return **redProperty**(*node*.left) && **redProperty**(*node*.right);

    }

    private int **countBlacks**(NodeRN<T> *node*, int *count*) {

        if (*node* == null) return *count*;

        if (!(*node*.isRed)) *count*++;

        int countL =  **countBlacks**(*node*.left, 0), countR = **countBlacks**(*node*.right, 0);

        return *count* + countL + countR;

    }

    private boolean **blackProperty**(NodeRN<T> *node*) {

        return (**countBlacks**(*node*.left, 0) - **countBlacks**(*node*.right, 0) == 0) ? true : false;

    }

    public boolean **isRB**(NodeRN<T> *root*) {

        if (*root* == null) return true;

*// Primera condición: BST*

        CheckBST<T> helper = new **CheckBST**<>(*root*);

        if (!(helper.**isBST**(*root*))) return false;

*// Segunda condición: raíz negra*

        if (*root*.isRed) return false;

*// Tercera condición: propiedad roja*

        if (!(**redProperty**(*root*))) return false;

*// Cuarta condición: propiedad negra*

        if (!(**blackProperty**(*root*))) return false;

        return true;

    }

}

1. **Árboles B**

La clase árbol además contiene un entero para el parámetro inmutable *t*. Cada nodo contiene los siguientes atributos: *parent*, *keys*, *children*, *n* (número de llaves actual), *isLeaf*.

Escriba un método (pueden escribir otros más y llamarlos) que reciba el árbol y determine si se cumplen las propiedades debidas:

1. Las llaves de todos los nodos siguen un orden no descendente (AKA: el orden "de siempre").
2. A la “izquierda” de cada llave sólo hay hijos con valores menores y a la “derecha” sólo hijos con valores mayores. (O hijos nulos, en su defecto.)
3. Todos los nodos (salvo, en su defecto, la raíz) contienen *t-1* llaves como mínimo.
4. Todos los nodos contienen *2t-1* llaves como máximo.
5. Todos los nodos internos tienen *n+1* hijos.
6. Todos los nodos internos tienen i*sLeaf* falso.
7. Todos los nodos hoja tienen *isLeaf* verdadero.

import java.util.List;

public class BTree<T extends Comparable<? super T>> {

*// Primera condición*

    private boolean **orderedKeys**(NodeBTree<T> *node*) {

        if (*node* == null) return true;

        for (int i = 0; i < *node*.key.**size**() - 1; i++) {

            if (*node*.key.**get**(i).**compareTo**(*node*.key.**get**(i + 1)) > 0) return false;

        }

        for (NodeBTree<T> child : *node*.children) {

            if (!(**orderedKeys**(child))) return false;

        }

        return true;

    }

*// Segunda condición*

    private boolean **holdsOrder**(NodeBTree<T> *node*) {

        if (*node* == null) return true;

        for (int i = 0; i < *node*.key.**size**(); i++) {

            T currVal = *node*.key.**get**(i);

            for (int j = 0; j < *node*.children.**size**(); j++) {

                List<T> currChild = *node*.children.**get**(j).key;

                for (int k = 0; k < currChild.**size**(); k++) {

                    if (j <= i && currChild.**get**(i).**compareTo**(currVal) >= 0) return false;

                    if (j > i && currChild.**get**(i).**compareTo**(currVal) <= 0) return false;

                }

            }

        }

        for (NodeBTree<T> child : *node*.children) {

            if (!(**holdsOrder**(child))) return false;

        }

        return true;

    }

*// Tercera condición*

    private boolean **minimumKeys**(NodeBTree<T> *node*, int *mn*) {

        if (*node* == null || *node*.children == null) return true;

        for (NodeBTree<T> child : *node*.children) {

            if (child.key.**size**() < *mn*) return false;

            if (!(**minimumKeys**(child, *mn*))) return false;

        }

        return true;

    }

*// Cuarta condición*

    private boolean **maximumKeys**(NodeBTree<T> *node*, int *mx*) {

        if (*node* == null || *node*.children == null) return true;

        for (NodeBTree<T> child : *node*.children) {

            if (child.key.**size**() < *mx*) return false;

            if (!(**maximumKeys**(child, *mx*))) return false;

        }

        return true;

    }

*// Quinta condición*

    private boolean **nChilds**(NodeBTree<T> *node*) {

        if (*node* == null || *node*.children == null) return true;

        if (*node*.children.**size**() < (*node*.n + 1)) return false;

        for (NodeBTree<T> child : *node*.children) {

            if (!(**nChilds**(child))) return false;

        }

        return true;

    }

*// Sexta condición*

    private boolean **nodeNotLeaf**(NodeBTree<T> *node*) {

        if (*node* == null) return true;

        if (*node*.children != null && *node*.isLeaf == true) return false;

        else {

            for (NodeBTree<T> child : *node*.children) {

                if (!(**nodeNotLeaf**(child))) return false;

            }

            return true;

        }

    }

*// Séptima condición*

    private boolean **leafNotNode**(NodeBTree<T> *node*) {

        if (*node* == null) return true;

        if (*node*.children == null && *node*.isLeaf != true) return false;

        else {

            for (NodeBTree<T> child : *node*.children) {

                if (!(**leafNotNode**(child))) return false;

            }

            return true;

        }

    }

    public boolean **isBTree**(NodeBTree<T> *root*) {

*// Primera condición: nodos ascendentes*

        if (!(**orderedKeys**(*root*))) return false;

*// Segunda condición: orden tipo BST*

        if (!(**holdsOrder**(*root*))) return false;

*// Tercera condición: mínimo de llaves*

        if (!**minimumKeys**(*root*, *root*.n - 1)) return false;

*// Cuarta condición: máximo de llaves*

        if (!**maximumKeys**(*root*, (2 \* *root*.n) - 1)) return false;

*// Quinta condición: número de hijos*

        if (!(**nChilds**(*root*))) return false;

*// Sexta condición: nodos no son hojas*

        if (!(**nodeNotLeaf**(*root*))) return false;

*// Séptima condición: hojas no son nodos*

        if (!(**leafNotNode**(*root*))) return false;

        return true;

    }

}

1. **Arboles Digitales de Búsqueda (*DST*)**

La clase árbol además contiene un entero que indica la longitud de bits con la que se está trabajando.

Cada nodo contiene su dato o llave, referencia a sus hijos izquierdo y derecho.

Escriba un método (pueden escribir otros más y llamarlos) que reciba el árbol y determine si se cumple que todos los nodos en el árbol coinciden con su ruta de inserción (considerando la longitud de bits indicada en el atributo respectivo de la clase árbol). Observe que la raíz puede ser cualquier valor. Por ejemplo, en el siguiente árbol (*casi* correcto), considerando una longitud de 4 bits las claves *5* y *4* sí son aptas para su ruta de inserción, pero *3* no.

7 [0111]

5 [0101]

3 [0011]

4 [0100]

public class DSTree<T extends Comparable<? super T>> {

    private boolean **value**(int *n*, int *pos*) { *// Obtiene bits de derecha a izquierda*

        return (((*n* >> *pos*) & 1) == 0) ? true : false;

    }

    private boolean **check**(Node<Integer> *node*, int *bitLen*, int *level*) {

        if (*node* == null) return true;

        if (*node*.left != null && !(**value**((Integer)*node*.left.key, *bitLen* - (*level* + 1)))) return false;

        if (*node*.right != null && **value**((Integer)*node*.right.key, *bitLen* - (*level* + 1))) return false;

        return **check**(*node*.left, *bitLen*, *level* + 1) && **check**(*node*.right, *bitLen*, *level* + 1);

    }

    public boolean **isDST**(Node<Integer> *root*, int *bitLen*) {

        return **check**(*root*, *bitLen*, 0);

    }

}

1. **Árboles *Radix***

La clase árbol además contiene un entero que indica la longitud de bits con la que se está trabajando.

Cada nodo contiene su dato o llave, referencia a sus hijos izquierdo y derecho. Observe que el dato o llave puede ser nulo.

Escriba un método (pueden escribir otros más y llamarlos) que reciba el árbol y determine si se cumple que todos los nodos en el árbol coinciden con su ruta de inserción (considerando la longitud de bits indicada en el atributo respectivo de la clase árbol). Observe que la condición anterior implica que todos los nodos internos están vacíos (no tienen llave). Por ejemplo, en el siguiente árbol (*casi* correcto), considerando una longitud de 4 bits las llaves *4* y *11* sí son aptas para su ruta de inserción, pero *6* no.

4 [0100]

0

1

1

1

1

6 [0110]

0

0

0

1

1

11 [1011]

# Entregables y rúbrica

Código correcto en cada caso:

1. Árboles ABB
   * 1: 5%
   * 2: 5%
   * 3: 5%
2. Árboles AVL: 15%
3. Árboles rojinegros: 20%
4. Árboles B: 20%
5. Árboles digitales de búsqueda: 20%
6. Árboles radix: 10%

**Valor porcentual por cada parte:**

* Parte 1 (análisis divide y vencerás): 50%
* Parte 2 (árboles): 50%

Por ejemplo: si un estudiante obtiene todos los puntos de la primera parte, pero sólo la mitad de los puntos de la segunda, su puntaje final sería: 50% + 25% = 75% del total de la evaluación asincrónica.