Universidad Autónoma de Yucatán

Maestría en Ciencias de la Computación

Paradigmas de Programación

Tarea Final

Autores:

Mario Herrera Almira

Dayan Bravo Fraga

8 de diciembre del 2022

**Descripción general del sistema**

El sistema utiliza los lenguajes de programación Python y Prolog para realizar una serie de operaciones sobre un árbol binario. Entre las operaciones que se desarrollaron se encuentran:

* Añadir un nuevo vértice al árbol.
* Determinar si un vértice existe en el árbol.
* Determinar los adyacentes de un vértice determinado.
* Recorridos en orden, preorden, en profundidad y a lo ancho.
* Determinar si un nodo es padre de otro.
* Determinar si un nodo es hoja.
* Determinar si un nodo es predecesor de otro.
* Determinar el nivel de un árbol.

**Documentación Técnica**

**Descripción de los métodos en Python**

**Método *insert:***

Este método recibe por parámetros un árbol y un vértice, y devuelve el árbol con el vértice añadido en la posición que le corresponda. El procedimiento es comparar el vértice nuevo con el vértice actual del árbol y moverse hacia la rama izquierda o derecha según corresponda. El procedimiento se repite hasta llegar a la posición correcta del árbol y el vértice nuevo se añade como una hoja en esa posición.

def insert(self, new\_data: float):  
 if new\_data < self.data:  
 if self.left is None:  
 self.left = Tree(new\_data)  
 self.left.parent = self  
 return  
 else:  
 self.left.insert(new\_data)  
 else:  
 if self.right is None:  
 self.right = Tree(new\_data)  
 self.right.parent = self  
 return  
 else:  
 self.right.insert(new\_data)

**Método *exists\_vertex:***

Este método recibe un vértice y un árbol por parámetros y devuelve verdadero si el vértice se encuentra contenido dentro del árbol o falso en caso contrario. El procedimiento consiste en ir comparando el vértice que se esta buscando con el vértice actual del grafo y moverse hacia la rama derecha o izquierda según corresponda hasta encontrar el vértice buscado o hasta que se llegue a una hoja del árbol y no se haya podido encontrar el vértice.

def exists\_vertex(self, vertex: 'Tree', tree: 'Tree') -> 'bool':  
 if not tree:  
 return False  
 elif tree == vertex:  
 return True  
 elif vertex < tree:  
 return self.exists\_vertex(vertex, tree.left)  
 else:  
 return self.exists\_vertex(vertex, tree.right)

**Método *adjacent\_vertexes:***

Este método recibe un vértice y un árbol por parámetros y devuelve los vértices hijos y el padre del vértice deseado. El procedimiento consiste en encontrar el vértice dentro del árbol y una vez encontrado de devuelven en una lista el hijo izquierdo, el derecho y el padre del vértice.

def adjacent\_vertexes(self, vertex: 'Tree', tree: 'Tree') -> list['Tree']:  
 if not tree:  
 return []  
 elif tree == vertex:  
 return [tree.left, tree.right, tree.parent]  
 elif vertex < tree:  
 return self.adjacent\_vertexes(vertex, tree.left)  
 else:  
 return self.adjacent\_vertexes(vertex, tree.right)

**Método *depth\_tour:***

Este método recibe un árbol por parámetros y lo recorre de manera que primero se impriman los vértices que estén más abajo en las ramas del árbol.

def depth\_tour(self, tree: 'Tree'):  
 if tree:  
 self.depth\_tour(tree.left)  
 self.depth\_tour(tree.right)  
 print(tree.data, end=", ")

**Método *broad\_tour:***

Este método recibe un árbol por parámetros y lo recorre de manera que se impriman los vértices en el orden de los niveles del árbol, por lo que primero se imprime la raíz, luego el siguiente nivel, luego el siguiente, hasta llegar al nivel más profundo del árbol.

def broad\_tour(self, nodes: list['Tree']):  
  
 if len(nodes) == 0:  
 return  
  
 for node in nodes:  
 if node is not None:  
 print(node.data.value, end=" ")  
 print()  
  
 children = []  
 for node in nodes:  
 if node is not None:  
 children.append(node.left)  
 children.append(node.right)  
  
 self.broad\_tour(children)

**Método *in\_order\_tour:***

Este método recorre el árbol de forma tal que se impriman todos los valores del árbol de manera ordenada de menor a mayor.

def in\_order\_tour(self):  
 if None is not self.left:  
 self.left.in\_order\_tour()  
 print(self.data.value)  
 if None is not self.right:  
 self.right.in\_order\_tour()

**Método *preorder\_tour:***

Este método recibe un árbol por parámetros y lo recorre de manera que se impriman los vértices que se van visitando de primeros, hace un recorrido parecido al de profundidad, pero imprime los vértices según los visita sin esperar a llegar el punto más bajo de la rama.

def preorder\_tour(self, tree: 'Tree'):  
 if tree:  
 print(tree.data.value)  
 self.preorder\_tour(tree.left)  
 self.preorder\_tour(tree.right)

**Método *get\_level:***

Este método recorre el árbol viendo en cada posición el nivel del subárbol de la izquierda y el nivel del subárbol de la derecha hasta llegar a las hojas que tienen nivel cero y en ese momento comienza a subir quedándose con el mayor nivel entre los dos subárboles más uno hasta llegar nuevamente a la raíz y de esa forma se obtiene el nivel completo del árbol.

def get\_level(self) -> int:  
 left\_level: int = 0 if self.left is None else self.left.get\_level()  
 right\_level: int = 0 if self.right is None else self.right.get\_level()  
 return max(left\_level, right\_level) + 1

**Método *is\_leaf:***

Este método recibe un vértice por parámetros y se apoya en el método *get\_vertex* para encontrar ese vértice en el árbol, una vez encontrado entonces se devuelve verdadero si el vértice no tiene hijos o falso en caso contrario.

def is\_leaf(self, vertex: 'Tree') -> bool:  
 found = self.get\_vertex(vertex, self)  
 if found:  
 if found.right is None and found.left is None:  
 return True  
 else:  
 return False

**Descripción de los métodos en Prolog**

**Método *leaf:***

Este método recibe un vértice y el árbol, y devuelve verdadero en caso de que el vértice sea una hoja del árbol o falso en caso contrario. El procedimiento consiste en recorrer el árbol buscando el vértice deseado y cuando se encuentre se verifica si este vértice tiene hijos o no.

% is a leaf node  
% leaf(Leaf, Tree)  
leaf(Leaf, tree(Leaf, nil, nil)) :- !.  
leaf(Leaf, tree(\_, Left, \_)) :- leaf(Leaf, Left), !.  
leaf(Leaf, tree(\_, \_, Right)) :- leaf(Leaf, Right), !.

**Método *insert:***

Este método recibe un vértice y un árbol, y devuelve el árbol con el vértice insertado en la posición que corresponde. El procedimiento consiste en comparar el vértice nuevo con el vértice actual del árbol y moverse hacia la rama izquierda o derecha según corresponda. Se repite este procedimiento hasta llegar a la posición correcta del árbol y el vértice nuevo se añade como una hoja en esa posición.

% Insert (Insert a node)  
% insert(Node, Tree, NewTree)  
insert(Node, tree(Parent, nil, Right ), tree(Parent, tree(Node, nil, nil), Right)) :- Node < Parent, !.  
insert(Node, tree(Parent, Left, nil ), tree(Parent, Left, tree(Node, nil, nil))) :- Node > Parent, !.  
insert(Node, tree(Parent, Left, Right), tree(Parent, NewLeft, Right)) :- Node < Parent, insert(Node, Left, NewLeft), !.  
insert(Node, tree(Parent, Left, Right), tree(Parent, Left, NewRight)) :- Node > Parent, insert(Node, Right, NewRight), !.

**Método *preorder:***

Este método recibe un árbol y devuelve una lista con los nodos en el orden en que se fueron recorriendo. Este va recorriendo el árbol como si fuera en profundidad, pero analiza cada nodo en el momento que lo visita sin esperar a llegar al nivel más profundo del árbol.

% preorder  
% preorder(Tree, List)  
preorder(nil, []) :- !.  
preorder(tree(Node, Left, Right), [Node|List]) :- preorder(Left, LeftList), preorder(Right, RightList), append(LeftList, RightList, List), !.

**Método *inorder:***

Este método recibe un árbol por parámetros y devuelve una lista con los vértices del árbol ordenados de menor a mayor. El procedimiento consiste en bajar lo más que se pueda por una rama luego imprimir el nodo de la izquierda del subárbol, luego la raíz del subárbol y por último el nodo de la derecha del subárbol.

% Inorder  
% inorder(Tree, List)  
inorder(nil, []) :- !.  
inorder(tree(Node, Left, Right), List) :- inorder(Left, LeftList), inorder(Right, RightList), append(LeftList, [Node|RightList], List), !.

**Método *postorder:***

Este método hace un recorrido en profundidad en el árbol que recibe por parámetros y devuelve una lista con la secuencia de valores ubicados de forma que primero se ponen los valores que están más al fondo de la rama izquierda del subárbol, luego los que están más al fondo de la rama derecha del subárbol y por último el nodo raíz del subárbol.

postorder(nil, []) :- !.  
postorder(tree(Node, Left, Right), List) :- postorder(Left, LeftList), postorder(Right, RightList), append(LeftList, RightList, TempList), append(TempList, [Node], List), !.

**Método *level:***

Este método devuelve la cantidad de niveles que tiene un árbol que recibe por parámetros. Si el árbol está vacío entonces devuelve cero, sino entonces devuelve el nivel máximo entre el subárbol de la izquierda y el de la derecha más 1. Finalmente se devolvería toda la suma que sería la cantidad de niveles totales del árbol.

level(0, nil) :- !.  
level(Level, tree(\_, Left, Right)) :- level(LeftLevel, Left), level(RightLevel, Right), max(LeftLevel, RightLevel, MaxLevel), Level is MaxLevel + 1.

**Método *find:***

Este método busca un vértice en un árbol y lo devuelve en caso de que lo logre encontrar. Como es un árbol binario de búsqueda entonces se busca de manera ordenada preguntando en cada paso si tiene que mover a la izquierda o a la derecha hasta que encuentre el nodo o se quede sin opciones.

% Find (Ordered Search)  
% find(Node, Tree, NodeTree).  
find(Node, tree(Node, Left, Right), tree(Node, Left, Right)) :- !.  
find(Node, tree(Parent, Left, \_), NodeTree) :- Node < Parent, find(Node, Left, NodeTree), !.  
find(Node, tree(Parent, \_, Right), NodeTree) :- Node > Parent, find(Node, Right, NodeTree), !.

**Método *parent:***

Este método verifica si un nodo es padre directo de otro y nos devuelve verdadero en caso de que sí o falso en caso contrario. La estrategia consiste en encontrar el nodo que el usuario especifique como padre y luego verificar si el nodo que el usuario especificó como hijo se encuentra entre los hijos del nodo padre.

% Parent (Direct Parent)  
% parent(Parent, Child, Tree)  
parent(Parent, Child, tree(Parent, tree(Child, \_, \_), \_)) :- !.  
parent(Parent, Child, tree(Parent, \_, tree(Child, \_, \_))) :- !.  
parent(Parent, Child, tree(Root, Left, \_)) :- Child < Root, parent(Parent, Child, Left), !.  
parent(Parent, Child, tree(Root, \_, Right)) :- Child > Root, parent(Parent, Child, Right), !.

**Método *predecessor:***

Este método verifica si un nodo es predecesor de otro, esto lo hace encontrando el nodo que el usuario identificó como predecesor y luego verificando si el otro nodo se encuentra dentro del subárbol que tiene como raíz al nodo predecesor.

% Predecessor (Is Predecessor)  
% predecessor(Predecessor, Child, Tree)  
predecessor(Predecessor, Child, tree(Predecessor, Left, \_)) :- Child < Predecessor, find(Child, Left, \_), !.  
predecessor(Predecessor, Child, tree(Predecessor, \_, Right)) :- Child > Predecessor, find(Child, Right, \_), !.  
predecessor(Predecessor, Child, tree(\_, Left, \_)) :- predecessor(Predecessor, Child, Left), !.  
predecessor(Predecessor, Child, tree(\_, \_, Right)) :- predecessor(Predecessor, Child, Right), !.

**Documentación del Usuario**

Cuando el usuario ingresa al sistema se le muestra un menú con todas las opciones disponibles:

1. Añadir un vértice:
   1. Añade al árbol un nuevo vértice determinado por el usuario.
2. Buscar un vértice:
   1. Busca en el árbol un vértice determinado por el usuario.
3. Busacr adyacentes a un vértice:
   1. Muestra al usuario los nodos hijos y el padre de un vértice determinado.
4. Recorrer en orden:
   1. Muestra al usuario la secuencia de valores del árbol ordenados de menor a mayor.
5. Recorrer en preorden:
   1. Muestra al usuario la secuencia de valores del árbol como si fuera en profundidad pero analisando cada valor en el momento en que lo visita sin esperar a llegar al nivel más profundo del árbol.
6. Recorrer en profundidad:
   1. Muestra al usuario la secuencia de valores del árbol utilizando el recorrido en profundidad que muestra primero los elementos más profundos de cada rama y por último la raiz.
7. Recorrer a lo ancho:
   1. Muestra al usuario la secuencia de valores del árbol recorrido por niveles empezando de arriba hacía abajo.
8. Determinar si un nodo es padre de otro:
   1. Muestra al usuario si un nodo determinado es padre de otro nodo o no, ambos nodos indicados por el usuario.
9. Determinar si un nodo es hoja:
   1. Muestra al usuario si un nodo determinado es una hoja de árbol o no.
10. Determinar si un nodo es predecesor de otro:
    1. Muestra al usuario si un nodo se encuentra dentro del subárbol donde la raíz es otro nodo específico.
11. Determinar el nivel del árbol:
    1. Muestra al usuario cuantos niveles tiene el árbol.
12. Salir:
    1. Finaliza el programa.

El usuario puede seleccionar la opción que desee y el sistema seguidamente pasará a pedirle al usuario que le proporcione los datos necesarios y por último ejecutará el algoritmo en los lenguajes que estén disponibles.

El siguiente gráfico muestra el proceso de interacción principal del usuario con el sistema, donde se observa que primero se muestran todas las opciones disponibles, luego se pasa a ejecutar los algoritmos tanto en Python como en Prolog si se encuentra implementado en ambos lenguajes. Puedo que el método se encuentre implementado en uno solo de los lenguajes, pero en cualquiera de los dos casos se pasa a pedirle los datos al usario necesarios para ejecutar el algoritmo. Luego se procede a ejecutar el método deseado y por último se le muestra los resultados al usuario y se regresa al inicio del menú para que el usuario seleccione de nuevo la opción que desee.

