Programa 6:
Búsqueda en Grafos mediante *Depth First Search (BFS)*

Estructura de Datos y Algoritmos II

Autor: José Mauricio Matamoros de Maria y Campos

Entrega: Lunes 20 de Abril, 2020

1. Introducción

Los árboles de búsqueda son estructuras de datos que permiten realizar un número diverso de operaciones, tales como SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, y DELETE relativamente rápido, lo que hace posible usarlos como diccionarios o colas de prioridad.

Objetivo: El estudiante conocerá e identificará las características de la estructura no lineal árbol.

2. Antecedentes

Las operaciones básicas en un árbol binario toman un tiempo proporcional a la altura del árbol. Dado un árbol binario saturado con n nodos, estas operaciones se ejecutarán en $\Theta(\log n)$ en el peor caso. Sin embargo, cuando el árbol es una cadena lineal de n nodos, las mismas operaciones toman un tiempo $O(n)$ el peor caso.

Cuando el conjunto de datos está formado por valores aleatorios de dispersión homogénea, la altura esperada de un árbol binario de búsqueda construido será de $\Theta(\log n)$, por lo que aplicar una operaciones básica tomará en promedio $\Theta(\log n)$.

No obstante, en la práctica no siempre es posible garantizar que los datos tengan una dispersión aleatoria homogenea, por lo que es necesario diseñar variaciones de árboles de búsqueda que garanticen un buen desempeño en el peor caso con cada una de las operaciones básicas. Ejemplos de estas modificaciones son (a) los árboles rojinegros que tienen siempre tienen una altura $O(\log n)$; y (b) los árboles tipo B, optimizados para operar en almacenamiento secundario (disco) lo que los hace ideales para sistemas de archivos y bases de datos grandes.

2.1. Árbol biario de búsqueda

Un árbol binario de búsqueda está organizado como se muestra en la Figura 1a. Se puede representar al árbol mediante una estructura de datos vinculados en la que cada nodo es un objeto. Además de una *llave* y datos satelitales, cada nodo contiene los atributos *left* (izquierdo), *right* (derecho) y p que apuntan a los nodos hijos izquierdo, derecho y padre respectivamente. Si falta un elemento, el atributo tendrá un valor nulo. El nodo raíz es el único nodo en el árbol cuyo padre es nulo.

Las llaves en un árbol binario de búsqueda están almacenadas de tal forma de la que satisfacen la siguiente propiedad.

Propiedad 1. *Sea x un nodo en un árbol binario de búsqueda T. Si y es un nodo en el sub-árbol izquierdo de x entonces $y.key \leq x.key$. Si y es un nodo en el sub-árbol derecho de x entonces $y.key \geq x.key$.*

2.2. Recorrido en-orden

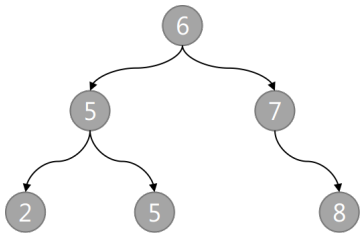
Esta propiedad permite imprimir todas las claves del árbol en orden mediante llamadas recursivas a un algoritmo llamado ***recorrido de árbol enorden*** (*inorder tree walk* en inglés). Este algoritmo se llama así porque imprime al padre en medio de los hijos izquierdo y derecho respectivamente. El algoritmo es como sigue:

Algoritmo 1 Recorrido de árbol enorden

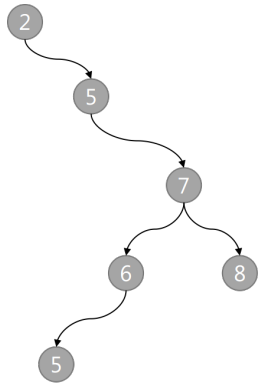
```
procedure INORDER-TREE-WALK(x)
    if x ≠ NIL then
        INORDER-TREE-WALK(x.left)
        PRINT(x.key)
        INORDER-TREE-WALK(x.right)
    end if
end procedure
```

De manera similar, un recorrido de árbol en ***preorden*** imprime la raíz antes de los valores en cualquier sub-árbol, y una caminata de árbol de ***postorden*** imprime la raíz después de los valores en sus sub-árboles.

Recorrer enorden un árbol binario de búsqueda de n nodos requiere a lo sumo de $\Theta(n)$.



(a) Árbol binario de búsqueda con 6 nodos y altura 2.



(b) Árbol binario de búsqueda menos eficiente con 6 nodos y altura 4 que contiene las mismas claves que Figura 1a.

Figura 1: Árboles binarios de búsqueda. Para cualquier nodo x, las claves en el sub-árbol a la izquierda de x son como máximo $x.key$, y las claves en el sub-árbol a la derecha de x son al menos $x.key$. Nótese que diferentes árboles pueden representar el mismo conjunto de valores. El tiempo de ejecución de la mayría de las operaciones considerando el peor caso será siempre proporcional a la altura del árbol.

2.3. Búsqueda

a menudo se necesita buscar una llave en un árbol. Además de la operación SEARCH (buscar), un árbol binario de búsqueda pude soportar otro tipo de operaciones como a) MINIMUM (mínimo), b) MAXIMUM (máximo), c) SUCCESSOR (sucesor) y d) PREDECESSOR (predecesor); cada una de los cuales tomará a lo más $\Theta(h)$ en un árbol binario de búsqueda de altura h.

Para buscar una llave, se usa el siguiente procedimiento:

Algoritmo 2 Búsqueda

```
procedure TREE-SEARCH(x, k)
    if x == NIL or k == x.key then
        return x
    end if
    if k < x.key then
        TREE-SEARCH(x.left, k)
    else
        TREE-SEARCH(x.right, k)
    end if
end procedure
```

Árboles muy grandes podrían generar un error de recursividad, por lo que a veces es preferible eliminar la misma y hacer la búsqueda en un poleo como sigue:

Algoritmo 3 Búsqueda iterativa

```
procedure ITERATIVE-TREE-SEARCH(x, k)
    while x ≠ NIL and k ≠ x.key do
        if k < x.key then
            x = x.left
        else
            x = x.right
        end if
    end while
    return x
end procedure
```

2.4. Máximo y mínimo

Siempre es posible encontrar en un árbol binario de búsqueda aquel elemento cuya llave es un mínimo siguiendo todos los apuntadores izquierdos desde la raíz hasta encontrar un nulo. El procedimiento mostrado a continuación devolverá un apuntador al mínimo del sub-árbol con raíz no nula x.

Algoritmo 4 Mínimo

```
procedure TREE-MINIMUM(x, k)
    while x.left ≠ NIL do
        x = x.left
    end while
    return x
end procedure
```

La Propiedad 1 garantiza que TREE-MINIMUM sea correcto. si un nodo x no tiene sub-árbol izquierdo, de inmediato x es el menor elemento en la secuenciaya que todos los elementos a la derecha tienen una llave mayor

a x. Si, por el contrario, x tuviere un sub-árbol izquierdo, los elementos menores o iguales que x siempre se encontrarán a la izquierda pues ningún elemento a la izquierda puede ser mayor que su padre.

Luego entonces, el algoritmo para encontrar al máximo será simétrico y ambos procedimientos se ejecutarán en $O(h)$.

Las operaciones de inserción y eliminación hacen que cambie el conjunto dinámico representado por el árbol. La estructura de datos debe modificarse para reflejar este cambio, pero de tal manera que la Propiedad 1 siga siendo válida. Si bien modificar el árbol para insertar un nuevo elemento es relativamente sencillo, la eliminación de datos es algo compleja.

2.5. Inserción

Para insertar un valor v en un árbol binario de búsqueda T, se usa el procedimiento TREE-INSERT. Este procedimiento toma un nodo z tal que $z.key = v, z.left = NIL$ y $z.right = NIL$. Entonces modifica T de tal manera que z es insertado en la posición correcta. El algoritmo se muestra a continuación.

Algoritmo 5 Inserción

```
procedure TREE-INSERT(T, z)
    y = NIL
    x = T.root
    while x ≠ NIL do
        y = x
        if z.key < x.key then
            x = x.left
        else
            x = x.right
        end if
    end while
    if y == NIL then
        T.root = z                    ▷ El arbol T está vacío
    else if z.key < y.key then
        y.left = z
    else
        y.right = z
    end if
end procedure
```

Al igual que los procedimientos TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT comienza en la raíz del árbol a partir del cual el apuntador x sigue una ruta simple hacia abajo en busca de NIL (nulo) que reemplazar con z. El procedimiento mantiene siempre al padre de x almacenado en y. Después de la inicialización, el ciclo while (líneas 3–7) mueve hacia abajo del árbol a los apuntadores, moviéndose hacia la izquierda o derecha según convenga y hasta que x apunta a nulo, es decir, la posición donde deseamos colocar el elemento de entrada z. Sin embargo, cuando x es nulo, la búsqueda ha avanzado un paso más allá del nodo que necesita ser cambiado. Es por esto que se necesita el apuntador y, pues es en éste donde se insertará z. Las líneas 8–13 modifican los apuntadores pertinentes que llevan a la inserción del nodo z.

Al igual que las otras operaciones primitivas, TREE-INSERT tiene una complejidad en tiempo de $O(h)$ para un árbol de altura h.

2.6. Eliminado

El procedimiento para borrar un nodo z de un árbol binario de búsqueda T toma como argumentos apuntadores a T y z. En este caso, es necesario tomar desiciones dependiendo de las situaciones que se presenten:

- Si z no tiene hijo izquierdo, entonces se reemplaza z con el hijo derecho, que puede o no ser nulo.
- Si z tiene exactamente un nodo hijo a la izquierda, entonces z se reemplaza con éste.
- En otro caso, z tiene dos nodos hijos. Es necesario encontrar un sucesor y para z en el sub-árbol derecho tal que y no tenga hijos izquierdos. El nodo y se transplanta entonces de su posición para reemplazar a z, dando lugar a dos sub casos.
  - Si y es el hijo derecho de z, entonces se reemplaza z con y, sin tocar a los hijos de éste.
  - De otro modo, y está en alguna parte del sub-árbol derecho pero no es el hijo derecho de z. En este caso, primero se reemplaza y con su propio hijo derecho y después se reemplaza z con y.

Para poder mover sub-árboles es necesario definir primero una subrutina a la que se denominará como TRANSPLANT, que reemplace un sub-árbol como hijo de su padre con otro sub-árbol. Cuando TRANSPLANT reemplaza el sub-árbol con raíz u con un sub-árbol de raíz v, el nodo padre de u se vuelve el nodo padre de v, y el padre de u termina teniendo v como hijo.

Algoritmo 6 Transplante

```
procedure TRANSPLANT(T, u, v)
    if u.p == NIL then
        T.root = v
    else if u == u.p.left then
        u.p.left = v
    else
        u.p.right = v
    end if
    if v ≠ NIL then
        v.p = u.p
    end if
end procedure
```

Las líneas 1–2 operan cuando u es la raíz de T, pues de otro modo u estará a la izquierda o derecha de su nodo padre. Las líneas 3–4 actuaizan $u.p.left$ si u es un hijo izquierdo, y la línea 5 actuaiza $u.p.right$ si u es un hijo derecho. Se permite que v sea nulo, y las líneas 6–7 si éste no lo fuere. Nótese que TRANSPLANT no intenta actualizar $v.left$ ni $v.right$, ya que esto es responsabilidad del procedimiento que llama a TRANSPLANT.

Con la subrutina TRANSPLANTa mano, es posible definir una rutina para eliminar al nodo z del arbol T.

Algoritmo 7 Eliminado

```
procedure TREE-DELETE(T, z)
    if z.left == NIL then
        TRANSPLANT(T, z, z.right)
    else if z.right == NIL then
        TRANSPLANT(T, z, z.left)
    else
        y = TREE-MINIMUM(z.right)
        if y.p ≠ z then
            TRANSPLANT(T, y, y.right)
            y.right = z.right
            y.right.p = y
        end if
        TRANSPLANT(T, z, y)
        v.left = z.left
        v.left.p = y
    end if
end procedure
```

El procedimiento TREE-DELETE ejecuta los cuatro casos como sigue. Las líneas 1–2 operan cuando el nodo z no tiene hijo izquierdo (o ningún hijo) y las líneas 3–4 cuando z tiene hijo izquierdo pero no hijo derecho. Las líneas 5–12 operan en los dos casos restantes, cuando z tiene dos hijos. La línea 5 encuentra el nodo y que es el sucesor de z. Como z tiene un sub-árbol derecho no vacío, su sucesor debe ser el nodo en ese sub-árbol con la llave más pequeña, por lo que es posible llamar a TREE-MINIMUM($z.right$). Tal como se mencionó anteriormente, el nodo y encontrado no tiene hijo izquierdo, por lo que es posible transportarlo y reemplazar a z con éste. Si y fuere hijo de z entonces las líneas 10–12 reemplazarán z como hijo de su padre por y, y reemplazarán al hijo izquierdo de y con el hijo izquierdo de z. Si, por el contrario, y no fuere hijo de z, las líneas 7–9 reemplazarán y como hijo de su padre con el hijo derecho de y, y convertirán al hijo derecho de z en el hijo derecho de y. Así, las líneas 10–12 reemplazan a z como hijo de su padre con y y reemplazan al hijo izquierdo de y con el hijo izquierdo de z.

Cada línea del procedimiento TREE-DELETE, incluidas las llamadas a la subrutina TRANSPLANT, requiere tiempo constante con excepción de la llamada a la subrutina TREE-MINIMUM (línea 5) que toma tiempo $O(h)$. Por lo tanto, el procedimiento TREE-DELETE tendrá una complejidad de $O(h)$ en un árbol de altura h.

3. Desarrollo:

A partir de la plantilla del Apéndice A, realice un programa que implemente los métodos `Tree.inorder_walk`, `Tree.preorder_walk`, `Tree.posorder_walk`,`Tree.insert`, y `Tree.delete`.

A continuación, ejecute los siguientes apartados en secuencia. Escriba la salida del programa en cada paso.

- [0 puntos] Ejecute el programa `tree.py` con el archivo de ejemplo `example.tree` y escriba la salida obtenida al recorrer el árbol enorden.
- [2 puntos] Escriba la salida obtenida al buscar la llave 17.
- [2 puntos] Escriba la salida obtenida al buscar la llave 5.
- [0 puntos] Elimine el nodo con la llave 17.
- [0 puntos] Inserte un nuevo nodo con la llave 5.
- [0 puntos] Inserte un nuevo nodo con la llave -1.
- [0 puntos] Escriba la salida obtenida al recorrer el árbol enorden.
- [2 puntos] Escriba la salida obtenida al recorrer el árbol pre orden.
- [2 puntos] Escriba la salida obtenida al recorrer el árbol post orden.
- [2 puntos] Escriba la salida obtenida al imprimir el máximo y el mínimo del sub-árbol con llave 5.

A. Archivo `tree.py`

tree.py

```
class Node:
  # Constructor
  def __init__(self, key=None):
    # the key of the node
    self.key = key

    # The children of the node
    self.p = None       # Parent node
    self.left = None
    self.right = None
  #end def

  def __str__(self):
    return '{}'.format(self.key)
  #end def

  def __repr__(self):
    return '{} => ({}, {})'.format(self
    .key, self.left, self.right)
  #end def
#end class


class Tree:
  # Constructor
  def __init__(self, root=None):
    if (root is None) or isinstance(
    root, Node):
      self.root = root
    else:
      self.root = Node(root)
  #end def

  # Methods
  def inorder_walk(self):
    def inorder_walk_rec(node):
      if node is not None:
        inorder_walk_rec(node.left)
        print('{} '.format(node.key),
    end='')
        inorder_walk_rec(node.right)
    # end def inorder_walk_rec
    inorder_walk_rec(self.root)
    print()
  # end def

  def preorder_walk(self):
    pass
  # end def

  def posorder_walk(self):
    pass
  # end def

  def max(self, node=None):
    if node is None:
      node = self.root
    pass
  # end def

  def min(self, node=None):
    if node is None:
      node = self.root
    pass
  # end def

  def iterative_search(self, key):
    x = self.root
    while (x is not None) and (key != x
    .key):
      x = x.left if key < x.key else x.
    right
    return x
  # end def

  def insert(self, key):
    pass
  #end def

  def delete(self, key):
    pass
  #end def

  def __str__(self):
    return repr(self.root)
  #end def

  # Loads a preorder-encoded tree from
    a file
  @staticmethod
  def from_file( file_path ):
    cc = [ -1 ]
    parts = []
    def read_next():
      cc[0] +=1
      if cc[0] < len(parts) and parts[
    cc[0]] != 'None':
        nxt = Node(int(parts[cc[0]]))
        nxt.left = read_next()
        nxt.right = read_next()
      else:
        nxt = None
      return nxt
    #end read_next

    with open(file_path, 'r') as fp:
      text = fp.read()
    parts = re.split(r'[\s\n]+', text)
    root = read_next()
    return Tree(root)
  #end def
#end class

def main():
  tree = Tree.from_file('example.tree')
  tree.inorder_walk()
  print(tree.iterative_search(17))
  print(tree.iterative_search(5))
  tree.delete(17)
  tree.insert(5)
  tree.insert(-1)
  tree.inorder_walk()
  tree.preorder_walk()
  tree.posorder_walk()
  print(tree.max(tree.iterative_search
    (5)))
  print(tree.min(tree.iterative_search
    (5)))


if __name__ == "__main__":
  main()
```