Universidad de Costa Rica

Escuela de Ingeniería Eléctrica

IE 0217- Estructura de datos abstractos y algoritmos

Nombre del Estudiante: Sebastian Herrera Esquivel

Carne: B93851

Profesor: Juan Carlos Coto Ulate

Asunto: Proyecto Treap Tree Search

Indice:

Introduccion:

Árbol de búsqueda binaria (BST):

El árbol de búsqueda binaria es una estructura de datos de árbol binario basada en nodos. Cada nodo tiene una llave las cuales siguen unas reglas para estructurar el árbol. La primera es que todos las llaves a la izquierda de cualquier nodo deben ser menores. La segunda es que todas las llaves a la derecha de cualquier nodo deben ser mayores. Es un tipo de estructura de datos cuya función es almacenar información de forma organizada. Ahora que está organizado se puede usar la busqueda binaria la cual compara el valor del medio con el valor que estamos buscando. Si no son iguales, se elimina la mitad del arbol en la que la llave no puede estar y la búsqueda continúa en la mitad restante. De ahí se toma nuevamente el elemento del medio para compararlo con el valor que queremos, y se repite esto hasta encontrar la llave deseada.

Treap Tree Search:

Treap es una estructura de datos que combina la búsqueda binaria de árboles (BST) y el heap, (tree + heap) treap. Es un árbol binario cuyos nodos contienen dos valores, una llave y una prioridad, de modo que la llave sigue las mismas reglas BST y la prioridad es un valor aleatorio que viene del heap. Por lo tanto es un árbol de búsqueda binario equilibrado. La forma que tomará el árbol y su distribución de nodos es aleatorio igual que con con el caso de un árbol binario.

Ejemplo de un Treab Tree:

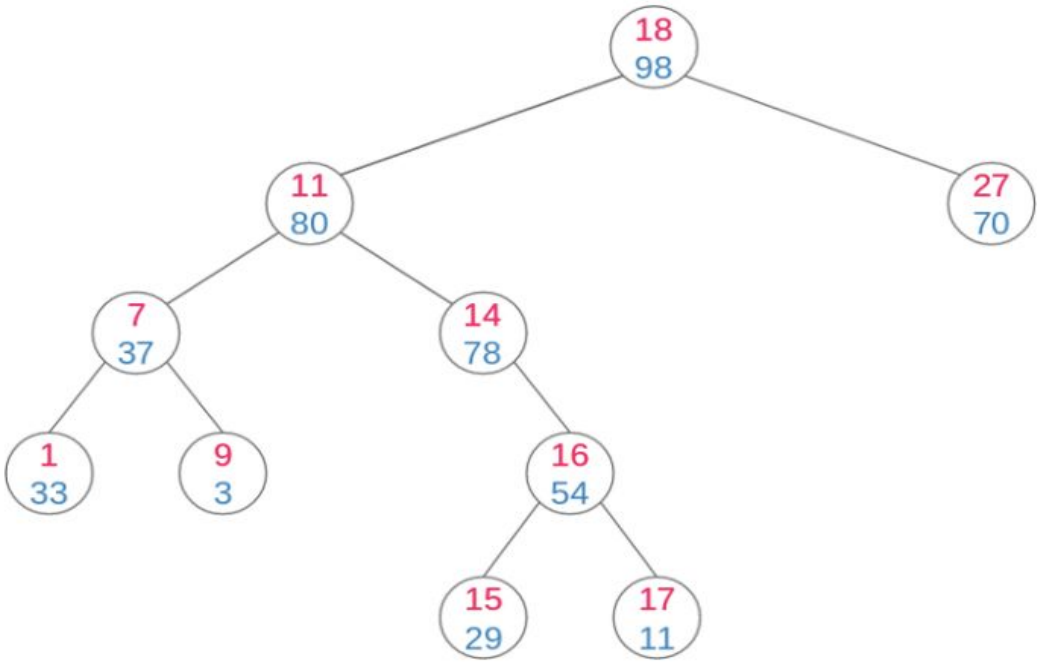


Figure 1: A example of a treap. The rose numbers are the keys (BST values) and the blue numbers are priorities given randomly (heap values).

Historia y Contexto sobre Treab Tree Search:

El treap fue descrito por primera vez por Raimund Seidel y Cecilia R. Aragón en 1989 como una derivación del árbol cartesiano. Lo introdujeron con la idea de que fuera un árbol en el cual cada nodo tuviera dos valores (x,y) como un plano cartesiano. La parte interesante es que la prioridad (posición en el árbol) es aleatoria y es otorgada por el heap. Gracias a esto la estructura del árbol siempre será aleatoria, con la raíz con la mayor prioridad y sus descendientes de menor llave a la izquierda y los de mayor a la derecha (manteniendo la propiedad BST).

Una variación que pudo haber llegado a ser lo que conocemos como Treab Tree Search es la sugerencia de Aragón y Seidel de asignar prioridades más altas a los nodos que se acceden con mayor frecuencia. Esta modificación haría que el árbol perdiera su forma aleatoria; en cambio, es más probable que los nodos a los que se accede con frecuencia estén cerca de la raíz del árbol, lo que hace que las búsquedas sean más rápidas.

Como Funciona:

Como es una estructura de árbol, utiliza las reglas BST para estructurar el árbol de manera ordenada y permitir la búsqueda binaria, la cual permite reducir el tiempo de búsqueda. El heap por otro lado proporciona un valor aleatorio de prioridad en el orden en que se van seleccionando los nodos para la estructura del árbol (shuffle). Al juntar estas dos propiedades se logra disminuir la altura del árbol y que sea equilibrado. Esto es de suma importancia ya que erradica la posibilidad de que la estructura termine siendo una lista (con la máxima altura posible). Al erradicar esto la funcionalidad de esta estructura es bastante buena, permitiendo la manipulación de información de manera rápida y dinámica gracias a su estructura binaria balanceada.

Discusion:

Para que se utiliza:

“La estructura treap ha sido elogiada por su elegancia y eficiencia, y ahora se utiliza en aplicaciones de producción que van desde redes inalámbricas hasta asignación de memoria y operaciones rápidas de conjuntos agregados en paralelo”. En la sección de referencias hay código fuente y aplicaciones de la estructura. - Cecilia Aragon de su blog de la universidad de Wahsington.

Ventajas:

Recordemos que en los árboles binarios entre más ordenada esté la entrada de nodos más alto será el árbol y como peor caso obtendremos una lista. El barajar rompe el orden y disminuye su altura, normalmente se puede asumir que la altura h = log (n). Todo esto implica que si queremos usar una estructura de árbol eficiente tenemos que leer todas las entradas, verificar que no estén ordenadas y después barajarlas. Todo este proceso sería largo y tedioso y sobre todo gastaría tiempo y rendimiento. El treap, sin embargo, es una estructura de datos que puede ayudarnos a mezclar las claves de una manera inteligente después de cada inserción.La idea es que podamos insertar todas las claves de la peor forma posible (es decir, de forma ordenada), pero la

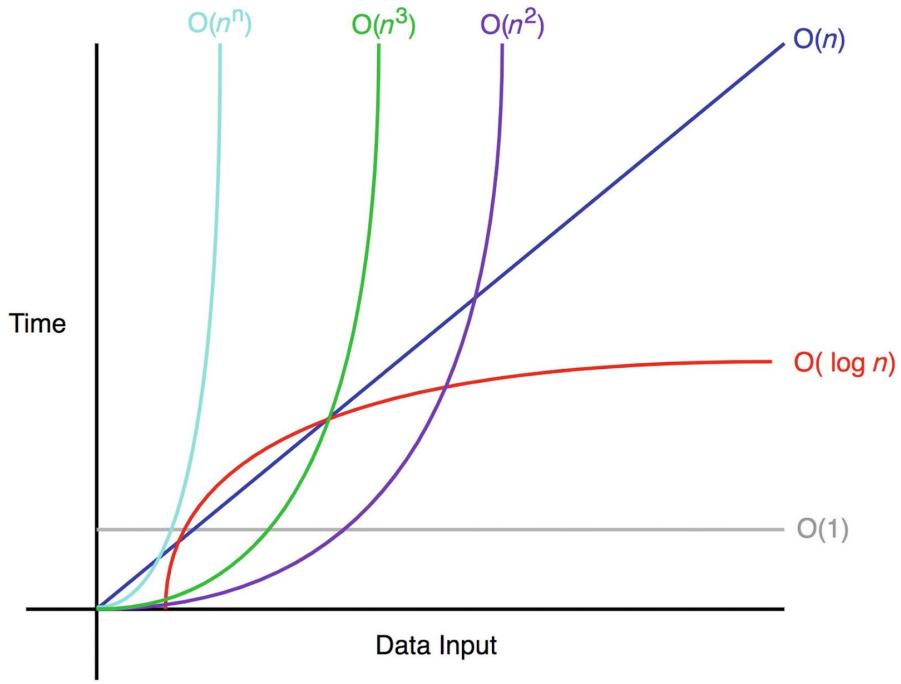
prioridad aleatoria se asegurará de mezclarlas “en tiempo real”. No sabemos el orden de las inserciones en ese tratamiento (ni siquiera sabemos cómo insertar elementos en un tratamiento todavía), pero podemos decir con certeza que el tratamiento simula un pedido específico. Cabe recalcar que los árboles treab tienen los mismos beneficios que un árbol binario normal como una almacenamiento de datos ordenado y una manipulacion de informacion bastante dinamica y rapida.

Ejemplo de una entrada, azul (priorities) y rosa (llave):

Figure 4: The nodes of the treap sorted based on their priority values (in blue).

Comportamiento en términos de complejidad

De modo que la altura del árbol casi siempre será h = log (n). La complejidad de tiempo esperado de la búsqueda, inserción y eliminación o cualquier otra operación es O (Log n).



Visualizando la gráfica de la big O notation podemos ver que esta estructura se desempeña bastante bien cuando la cantidad de información que se manipula es bastante grande, de hecho la mayoría de usos derivan de esa premisa. En cambio otras estructuras con otro comportamiento y complejidad de tiempo tendrán una mejor eficiencia cuando se trata de manipulaciones de menor cantidad de información.

Por ejemplo, buscar personas en una guía telefónica es O (log n). No es necesario consultar a todas las personas de la guía telefónica para encontrar la correcta; en su lugar, puede simplemente dividir y conquistar buscando en función de dónde está su nombre alfabéticamente, y en cada sección solo necesita explorar un subconjunto de cada sección antes de encontrar el número de teléfono de alguien.

Ejemplo:

(Insertar un nodo y buscar un nodo)

```
#Sebastian Herrera Esquivel
#Treap Tree Search
#Implementaciones Insertar y buscar

import random
import sys

#Aqui se crean los nodos.
#Las keys son la prioridad y el value es el valor BST
class node:
```

```
 def __init__(self, value, key=None, left_c = None,
right_c = None):
   self.value = value
   self.left_c = left_c
   self.right_c = right_c
   self.key = key
   if self.key == None:
     random.randint(0, sys.maxsize)
   else:
     self.key = key

class binary_search_tree:
 #Se construye el arbol sin nada en la raiz
 def __init__(self):
   self.root = None
  #Se crea una raiz si no hay o se toma la raiz como
padre para agregar el nodo en la posicion correcta.
 def insert(self, value):
   if self.root == None:
     self.root = node(value)
   else:
     self._insert(value, self.root)

 #Proceso recursivo de inserccion tomando en cuenta
las prioridades
```

```
 #primero se agrega el valor siguiendo BST, para luego
con la prioridad rotar el arbol. (No se cambia el
padre antes de rotar)

 def _insert(self, value, actual_parentN):
   if value < actual_parentN.value:
     if actual_parentN.left_c == None:
       actual_parentN.left_c = node(value)

     if actual_parentN.key >
actual_parentN.left_c.key:
       self._rotate_right(actual_parentN)
     else:
       self._insert(value, actual_parentN.left_c)

   elif value > actual_parentN.value:
     if actual_parentN.right_c == None:
       actual_parentN.right_c = node(value)
     if actual_parentN.key >
actual_parentN.right_c.key:
       self._rotate_left(actual_parentN)
     else:
       self._insert(value, actual_parentN.right_c)

   else: #para no repetir valores
     pass
```

```
 def _rotate_left(self, actual_parentN):
     copy_parent = node(actual_parentN.value,
actual_parentN.key, actual_parentN.left_c,
actual_parentN.right_c.left_c)

     actual_parentN.key = actual_parentN.right_c.key
     actual_parentN.value =
actual_parentN.right_c.value
     actual_parentN.right_c =
actual_parentN.right_c.right_c
     actual_parentN.left_c = copy_parent

 def _rotate_right(self, actual_parentN):
     copy_parent = node(actual_parentN.key,
actual_parentN.value, actual_parentN.left_c.right_c,
actual_parentN.right_c)

     actual_parentN.key = actual_parentN.left_c.key
     actual_parentN.value =
actual_parentN.left_c.value
     actual_parentN.left_c =
actual_parentN.left_c.left_c
     actual_parentN.right_c = copy_parent
  def find(self, value):
   if self.root!= None:
```

```
      return self._find(value, self.root)
    else:
      return None

  def _find(self, value, actual_parentN):
    if value == actual_parentN.value:
      return actual_parentN
    elif value < actual_parentN.value and
actual_parentN.left_child != None:
      return self._find(value,
actual_parentN.left_child)
    elif value > actual_parentN.value and
actual_parentN.right_child != None:
      return self._find(value,
actual_parentN.right_child)


def fill (tree):
  for n in range(20):
    cur_elem = n
    tree.insert(cur_elem)
  return tree

tree = binary_search_tree()
tree = fill (tree)
```

```
print (str(tree.find(10)))
```

Conclusiones:

Como sabemos el árbol de búsqueda binaria es una estructura de datos de árbol binario basada en nodos donde cada nodo tiene una llave y un valor, además de seguir las reglas BST y llaves aleatorias del heap. Sin alguna duda este algoritmo puede ser implementado en una gran variedad de aplicaciones y usos, que con su tiempo de búsqueda tan eficiente lograra mejorar cualquier aplicación. Es un algoritmo que vale la pena comprender pero que tan bien puede dar dolores de cabeza si no se afronta con los pasos correctos. Me gustaría resaltar la forma en cómo afronte este desafío y lograr comprender el algoritmo para poder implementar el algoritmo. Primero habrá que tener un conocimiento y una idea de cómo se implementa la búsqueda binaria y la organización con respecto a los nodos que van entrando a un árbol. Después de tener una buena idea de estas implementaciones se podría incluir la forma en cómo se van dando llaves random a los nodos para que el árbol se estructure a partir de esto. Por último lo hay y más complicado son las rotaciones que se implementan en la inserción de nodos. Para comprender estas rotaciones es necesario tener en cuenta el nivel micro y macro de las afectaciones que produce rotar dos nodos ya que pueden modificar una gran parte del árbol si están cerca del centro y poco si son hojas. Un consejo que daría para visualizar mejor estas rotaciones es dibujar todo el proceso de rotación (en mi experiencia esto aclaro todas mis dudas con respecto al algoritmo). Sin duda un desafío que vale la pena tomar para mejorar en nuestro ambito laboral y conocimiento personal.

Referencias:

Introduction To Algorithms - Page 297 Thomas H.. Cormen, Thomas H Cormen, Charles E Leiserson · 2001. Recuperado 13 de diciembre de 2020, de https://books.google.co.cr/books?id=NLngYyWFl_YC&pg=PA297&dq=treap+tree&hl=en&sa=X&ved=2ahUKEwisie_q8cntAhUKmVkKHV8pDF4Q6AEwAnoECAMQAg#v=onepage&q=treap%20tree&f=false

Martínez, Conrado; Roura, Salvador (1997), "Randomized binary search trees", *Journal of the ACM*, Recuperado 13 de diciembre de 2020 http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.243

Aragon, Cecilia R.; Seidel, Raimund (1989), "Randomized Search Trees" (PDF), *Proc. 30th Symp. Foundations of Computer Science (FOCS 1989)*, Washington, D.C. Recuperado 13 de diciembre de 2020, de http://faculty.washington.edu/aragon/pubs/rst89.pdf

Seidel, Raimund; Aragon, Cecilia R. (1996), "Randomized Search Trees", *Algorithmica* Recuperado 13 de diciembre de 2020, de *http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.8602*

Cecilia Aragon | Professor, University of Washington. (s. f.). Treaps. Recuperado 14 de diciembre de 2020, de https://faculty.washington.edu/aragon/treaps.html

Código fuente y aplicaciones (Referencias otogadas por Cecilia R. Aragón):

https://www.drdobbs.com/windows/treaps-in-java/184410231

Treaps in Java, en Dr. Dobbs 'Journal, por Stefan Nilsson, incluido el código fuente de Java.

http://www.cs.cmu.edu/~scandal/treaps.html
Operaciones rápidas de conjuntos paralelos utilizando treaps de G. Blellock y M. Reid-Miller.

https://www.cs.cornell.edu/courses/cs312/2003sp/lectures/lec26.html
Implementación de conjuntos ordenados usando treaps, conferencia de la Universidad de Cornell de CS312, Estructuras de datos y programación funcional.

https://www.codeproject.com/Articles/8184/Treaps-in-C
Treaps en C # (incluye código fuente) por Roy Clem.

https://people.ksp.sk/~kuko/gnarley-trees/Intro.html
Animación Java de varios tipos de árboles, incluidos los treaps de Kubo Kovac.