RESULTADOS EJECUCIÓN WBT y WBT Mejorado vs AVL – CON PONDERACIÓN DE LOS DATOS +/-80%

Seed = 1234

Usage Balanced Tree ELS

{'weight': 20000, 'size': 1980, 'height': 13}

tree ELS 9.7837

left ELS 8.7944

right ELS 8.783540248297957

Improved Usage Balanced Tree ELS

{'weight': 20000, 'size': 1980, 'height': 13}

tree ELS 9.66465

left ELS 8.682

right ELS 8.657689227072488

Result

Cumulative heights WBT 1303120

Cumulative heights WBT improved 1272649

Cumulative heights AVL 1346962

Result - UBT

1437726 function calls (269207 primitive calls) in 0.501 seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)

1 0.000 0.000 0.501 0.501 <string>:1(<module>)

1 0.031 0.031 0.501 0.501 UBTvsAVL-TestBorrar.py:10(findtest)

134601 0.029 0.000 0.470 0.000 UsageBalancedTree.py:87(findnode)

1303120/134601 0.441 0.000 0.441 0.000 UsageBalancedTree.py:93(\_findnode)

1 0.000 0.000 0.501 0.501 {built-in method builtins.exec}

1 0.000 0.000 0.000 0.000 {built-in method builtins.len}

1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}

Result – UBT Improved

1407255 function calls (269207 primitive calls) in 0.516 seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)

1 0.000 0.000 0.516 0.516 <string>:1(<module>)

1 0.033 0.033 0.516 0.516 UBTvsAVL-TestBorrar.py:10(findtest)

134601 0.030 0.000 0.484 0.000 UsageBalancedTree.py:87(findnode)

1272649/134601 0.453 0.000 0.453 0.000 UsageBalancedTree.py:93(\_findnode)

1 0.000 0.000 0.516 0.516 {built-in method builtins.exec}

1 0.000 0.000 0.000 0.000 {built-in method builtins.len}

1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}

Result - AVL

1481568 function calls (269207 primitive calls) in 0.521 seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)

1 0.000 0.000 0.521 0.521 <string>:1(<module>)

134601 0.029 0.000 0.490 0.000 BinaryBalanceTreeAVL.py:76(findnode)

1346962/134601 0.460 0.000 0.460 0.000 BinaryBalanceTreeAVL.py:82(\_findnode)

1 0.031 0.031 0.521 0.521 UBTvsAVL-TestBorrar.py:10(findtest)

1 0.000 0.000 0.521 0.521 {built-in method builtins.exec}

1 0.000 0.000 0.000 0.000 {built-in method builtins.len}

1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}

RESULTADOS EJECUCIÓN WBT y WBT Mejorado vs AVL – SIN PONDERACIÓN DE LOS DATOS

Seed = 1234

Usage Balanced Tree ELS

{'weight': 20000, 'size': 1980, 'height': 13}

tree ELS 9.7837

left ELS 8.7944

right ELS 8.783540248297957

Imporoved Usage Balanced Tree ELS

{'weight': 20000, 'size': 1980, 'height': 13}

tree ELS 9.66465

left ELS 8.682

right ELS 8.657689227072488

Result

Cumulative heights WBT 780320

Cumulative heights WBT improved 773680

Cumulative heights AVL 794840

Result UBT

859525 function calls (158405 primitive calls) in 0.273 seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)

1 0.000 0.000 0.273 0.273 <string>:1(<module>)

1 0.016 0.016 0.273 0.273 UBTvsAVL-TestBorrar.py:10(findtest)

79200 0.017 0.000 0.257 0.000 UsageBalancedTree.py:87(findnode)

780320/79200 0.240 0.000 0.240 0.000 UsageBalancedTree.py:93(\_findnode)

1 0.000 0.000 0.273 0.273 {built-in method builtins.exec}

1 0.000 0.000 0.000 0.000 {built-in method builtins.len}

1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}

Result UBT Improved

852885 function calls (158405 primitive calls) in 0.316 seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)

1 0.000 0.000 0.316 0.316 <string>:1(<module>)

1 0.020 0.020 0.316 0.316 UBTvsAVL-TestBorrar.py:10(findtest)

79200 0.018 0.000 0.296 0.000 UsageBalancedTree.py:87(findnode)

773680/79200 0.279 0.000 0.279 0.000 UsageBalancedTree.py:93(\_findnode)

1 0.000 0.000 0.316 0.316 {built-in method builtins.exec}

1 0.000 0.000 0.000 0.000 {built-in method builtins.len}

1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}

Result AVL

874045 function calls (158405 primitive calls) in 0.315 seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)

1 0.000 0.000 0.315 0.315 <string>:1(<module>)

79200 0.017 0.000 0.297 0.000 BinaryBalanceTreeAVL.py:76(findnode)

794840/79200 0.280 0.000 0.280 0.000 BinaryBalanceTreeAVL.py:82(\_findnode)

1 0.018 0.018 0.315 0.315 UBTvsAVL-TestBorrar.py:10(findtest)

1 0.000 0.000 0.315 0.315 {built-in method builtins.exec}

1 0.000 0.000 0.000 0.000 {built-in method builtins.len}

1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}

{'weight': 20000, 'size': 1980, 'height': 13}

RESULTADOS EJECUCIÓN WBT y WBT Mejorado vs AVL – SIN PONDERACIÓN DE LOS DATOS

Seed = 10

Cumulative heights WBT 783080

Cumulative heights WBT improved 779960

Cumulative heights AVL 793760

RESULTADOS EJECUCIÓN WBT y WBT Mejorado vs AVL – SIN PONDERACIÓN DE LOS DATOS

Seed = 912

Cumulative heights WBT 777760

Cumulative heights WBT improved 770760

Cumulative heights AVL 798600

RESULTADOS EJECUCIÓN WBT y WBT Mejorado vs AVL – CON PONDERACIÓN DE LOS DATOS +/- 80%

Seed = 10

Cumulative heights WBT 1286056

Cumulative heights WBT improved 1260781

Cumulative heights AVL 1323770

RESULTADOS EJECUCIÓN WBT y WBT Mejorado vs AVL – CON PONDERACIÓN DE LOS DATOS +/- 80%

Seed = 912

Cumulative heights WBT 1288092

Cumulative heights WBT improved 1249620

Cumulative heights AVL 1340801

Comparative Performance Table



CONCLUSIONES

* Sin ponderación, anti-intuitivamente, el rendimiento del WBT es mejor que el del AVL aunque en proporciones pequeñas (rondando entre el 2% y el 3% en profundidad total de las búsquedas y consecuentemente de llamadas recursivas a “findnode”)
* Con ponderación el rendimiento, como cabía esperar, mejora, aunque no tan sustancialmente como previsto, encontrándose en el entorno del 4% de la profundidad y el número de llamadas.
* La versión mejorada UBT obtiene un rendimiento superior. Respecto al AVL el rendimiento del UBT es sin proporcionalidad más cercano al 3%, mientras que con ponderación la mejora respecto a AVT es entorno al 5% y hasta del 7% según el caso
* Con un procedimiento de ponderación más ajustado al objetivo de este árbol de uso (con mayor diferencia de peso entre unos nodos y otros) los renidmientos mejoran entorno al 20%

ANEX: CODIGO MODULO PRUEBA

import BinaryBalanceTreeAVL as avl

import UsageBalancedTree as ubt

import random as rd

import cProfile

import copy

# --------------------------------------------------------------------------------

# Function to test performance of WBT vs AVL bulk data access with cProfile

# --------------------------------------------------------------------------------

def findtest(listaprueba, tree):

for ele in range(len(listaprueba)):

tree.findnode(listaprueba[ele])

# --------------------------------------------------------------------------------

# Function to test performance of AVL insert function with cProfile

# --------------------------------------------------------------------------------

def testinsertAVL(listanodos, tree):

for ele in range(len(listanodos)):

tree.insert(listanodos[ele])

# --------------------------------------------------------------------------------

# Random node list generation for tree creation

# --------------------------------------------------------------------------------

lista = []

n = 10000

rd.seed(10)

for i in range(n):

code = rd.randint(0,2000)

lista.append(code)

# --------------------------------------------------------------------------------

# AVL and WBT Tree Creation... with time performance check for AVL

# --------------------------------------------------------------------------------

mytree = ubt.WBTree()

mytreeAVL = avl.AVLTree()

listanodos=[]

for i in range(len(lista)):

listanodos.append(avl.AVLNode(lista[i]))

cProfile.run("testinsertAVL(listanodos, mytreeAVL)")

for i in range(len(lista)):

nodovalue = ubt.WBTNode(lista[i])

mytree.insert(nodovalue)

# --------------------------------------------------------------------------------

# WBT - ELS (Expected Length of Search)

# --------------------------------------------------------------------------------

# Just after tree creation, without rebalance

print(mytree.branchweight(mytree.top))

print ("tree ELS:", mytree.branchels(mytree.top))

print ("left ELS", mytree.branchels(mytree.top.left))

print ("right ELS:", mytree.branchels(mytree.top.right))

# We choose a number of nodes (n). Those will be accessed a random number of times.

# This will increase the weight of those nodes

n = 200 # 10% of the nodes

for i in range(200):

b = rd.choice(lista)

for i in range(rd.randint(10,100)):

mytree.findnode(b)

# and we rebalance the tree with the new weights and print the new ELS

mybacktree = copy.deepcopy(mytree)

mytree.rebalance(False)

print(mytree.branchweight(mytree.top))

print ("tree ELS", mytree.branchels(mytree.top))

print ("left ELS", mytree.branchels(mytree.top.left))

print ("right ELS", mytree.branchels(mytree.top.right))

mybacktree.rebalance(True)

print(mytree.branchweight(mytree.top))

print ("tree ELS", mybacktree.branchels(mybacktree.top))

print ("left ELS", mybacktree.branchels(mybacktree.top.left))

print ("right ELS", mybacktree.branchels(mybacktree.top.right))

print (mybacktree)

# --------------------------------------------------------------------------------

# Preparing the final test data randomly

# --------------------------------------------------------------------------------

listatree = mytree.treetolist()

listatest = []

for i in range(len(listatree)):

nodo = mytree.findnode(rd.choice(lista))

#

# To get a more accurate result of the performance test WBT vs AVL (improved performance of WBT) the list of

# elements should be evaluated in regards with the weight of each node (more entries for the "heaviest" nodes)

# For that, we process the initial list data and generate as many requests as random(0,5 - 1,5) times the weight

# The next two sentences could do the trick when preparing the data set:

numqueries = 1-rd.randint(-80, 80)/100

for i in range(int(5\*nodo.weight\*numqueries)): listatest.append(nodo.value)

#

# Alternatively we have done a even distribution of loads, which should benefit AVL performance vs WBT

#for i in range(40): listatest.append(nodo.value)

rd.shuffle(listatest)

# For performance reference we calculate the cumulative depth of all the nodes accessed.

# Idealy the lowest total depth (WBT vs AVL) should get the best performance

# The different would be the number of times in excess the find method is accessed

dep = 0

depback= 0

depavl = 0

for i in range(len(listatest)):

noda=mytree.findnode(listatest[i])

deptmp = mytree.depth(noda)

dep+=deptmp

nodi=mybacktree.findnode(listatest[i])

deptmp = mybacktree.depth(nodi)

depback+=deptmp

nodo = mytreeAVL.findnode(listatest[i])

depabbtmp = mytreeAVL.depth(nodo)

depavl += depabbtmp

print ("Cumulative heights WBT", dep)

print ("Cumulative heights WBT improved", depback)

print ("Cumulative heights AVL", depavl)

# Final bulk find execition for performance test

cProfile.run("findtest(listatest,mytree)")

cProfile.run("findtest(listatest,mybacktree)")

cProfile.run("findtest(listatest,mytreeAVL)")