Lab 4 : Balanced Trees

Steven Robles *

University of Texas, El Paso, TX

March 26, 2019

This paper aims to demonstrate implementations of a balanced binary tree. The structure is based upon connected individual nodes with its items and and children information are contained in native Python lists. Within this lab, the attributes pertaining to this data structure are demonstrated through a serious of tasks which as evaluated based upon time performance in seconds.

;

Contents

I. Introduction

Binary tree structures are an essential part within data sciences. There are many different typed of binary tree structures with all having different properties. In this lab, we'll be focusing on a particular structure type, B-Trees. B-Trees is a type of structure where each node contains information within array. In this case, this lab will be using native python lists. Information stored within these lists include the nodes items and its children.

To demonstrate the different implementations and applications of B-Trees, the lab is 9 different parts. Each part demonstrates a specific property of the tree. They each complete a given task based upon a given intital B-Tree. Each task takes time to execute, which is recorded for evaluation of the performances towards the end of the lab.

*ID Number: 80678755

II. Design

This lab, as stated in the Introduction, consists of 9 separate parts. Some tasks over layed with other in such way that it is implemented together. Overall design challenges for this lab were minimal, since an extensive exercise was provided prior to staring this lab. Many essential components utilized in the exercise were implemented through the lab.

A. Part 1: Height of Tree

Calculating the height of the tree turned out to be one of the simpler tasks to accomplish in this lab. Since the properity of B-Tress state that it must always be balanced, the height of the tree will always be the depth level of any leaf node. It does not matter which path is derived from the root, as long as the destination is a leaf node with depth d.

B. Parts 2: Inserting Into Sorted Array

Another crucial property that payed a key role in this section and as well as other sections is that the items within a B-Tree are stored in order. This means that the values of the items increases from left to right. Even with this property in mind, completing this task proved to be on the most challenging part of the lab. A native python can be sorted if each an inorder transversing is utilized while appending item values. A solution was designed in a way that each of children of a node with values less than the given node is transversed before the node value itself. This was accomplished by constantly checking these two items after each iteration.

C. Parts 3 & 4: Smallest and Largest Items

This sections is shared between the second and third parts of the lab because both parts follow the same approach. Since the B-Tree is sorted, the locations of the smallest and largest items are already pre-known. This translates to always finding the smallest item being located in the left most item slot in the left most node of the tree. Similar, the largest item can be found in the right most item slot in the right most node of the tree. Having only differ slightly, the same logic approach was utilized for both parts, which is only transversing one of two directions.

D. Part 5: Nodes at Depth d

This part is presented some challenge when approaching the problem. The objective is to count the number of nodes at any give depth of d of the tree. The first challenge that presented itself is knowing that the all of the children of the parent node has been accessed and checked. This was solved by assuring that all of the children is accessed through a for loop of the node children lists. An interesting aspect of this problem in general is to implement a fail safe. The depth provided could exceed the height of the tree. In this case, a $-\infty$ value is returned signifying that the depth level has exceeded the tree's height.

E. Part 6: Printing Nodes at Depth d

The following section followed a similar logic approach to that of part 5. Each children is checked by comparing its depth level value. A key feature in this problem is that the transversal of the B-Tree must be from left to right, as the items must be printed in order. Once again, a fail save was implemented in case the desired depth level exceeds the height of the tree.

F. Parts 7 & 8: Full Nodes and Leafs

Determining the total amount of full nodes and leafs follow the same logic approach, that's is why this section includes both parts of the lab. Determining if a node is full or not is a very simple task, which is accomplished by checking the the number of items to the constructor value of 5. The only difference between both parts is that the second function checked an additional parameter, which is if the current node is a leaf or not. Both of these parts presented little to no challenges and are an effective demonstration of the properties of B-Trees.

G. Part 9: Return Depth d Given k

This part of the lab presented some challenges. It follows the same approach as parts 5 and 6 in terms of locating a depth of a node, but also includes search for item k. Since the fail safe should return a -1, means of acquiring the depth level of the item is founded required create solutions on what the function should return since it is iterative.

III. Results

After the successful implementations of the previous parts of the lab, each tasks were executed and their execution times were recorded. The times were recorded to evaluate the performance of each results. Below is the collection of the results of each task as well as the time trial results.

Figure 1: B-Tree

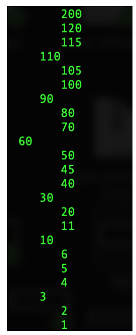


Figure 2: Part 1

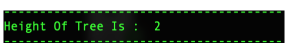


Figure 3: Part 2

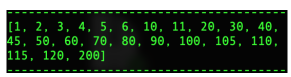


Figure 4: Part 3

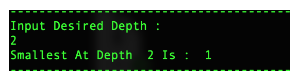


Figure 5: Part 4

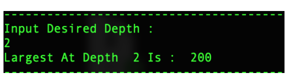


Figure 6: Part 5

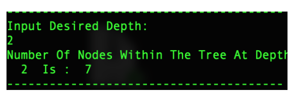


Figure 7: Part 6

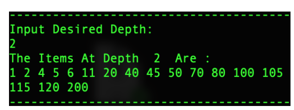


Figure 8: Part 7

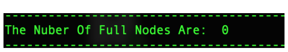


Figure 9: Part 8

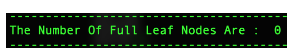


Figure 10: Part 9

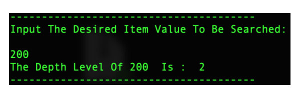
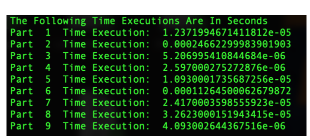


Figure 11: Time Trials



IV. Discussion

Overall, implementing tasks to the B-Trees turns into simpler challenges if you keep in mind that the nodes are just connected python native lists. The performances are effective in a way that all of the items are already sorted, so previous sorting algorithms used for BST are no longer needed.

V. Source Code

```python
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 03/24/2019
Discreption: Lab 3:
        This is the main program of which lab 4 is built. It realies on the attached file to complete the
        given tasks. This prgram mainly focuses on the user interface and timing the task executions.
"""

from B_Tree import BuildTree
from B_Tree import Height
from B_Tree import SortList
from B_Tree import SmallestAtD
from B_Tree import LargestAtD
from B_Tree import CountAtD
from B_Tree import PrintAtD
from B_Tree import CountFullNodes
from B_Tree import FullLeaves
from B_Tree import DepthAtK

import timeit

#sets loop value and builds the b-tree
T = BuildTree()
loop = True

#the while loops exeecuetes the user interface a
while loop:
  print("Hello! Do you want to proceed with?\n1.  Find Height Of Tree\n2.  Extract Items Into Sorted List")
  print("3.  Print Smallest At Depth D\n4.  Print Largest At Depth D\n5.  Count Nodes In Tree At Depth D")
  print("6.  Print Items At Depth D\n7.  Return Full Nodes\n8.  Return Full Leaf Nodes\n9.  Find Depth Of Item K\n10.  Time Trials")
  choice = input("11. Exit\n")
  try: #the try and except functions to convert input into an integer unless the input isnt a number
    choiceNum = int(choice)
  except ValueError:
    choiceNum = -1
  print("----------------------------------------")
  if (choiceNum == 1): #outputs the height of the tree
    print("Height Of Tree Is : ",Height(T))

  elif(choiceNum == 2): #extracts items into a sorted list.
    L = []
    SortList(T, L)
    print(L)


  elif(choiceNum == 3): #prints smallest at depth d
    dIn = input("Input Desired Depth : \n")
    d = int(dIn)
    print("Smallest At Depth ", d, "Is : ", SmallestAtD(T, d))

  elif(choiceNum == 4): # Prints Largest at depth d
    dIn = input("Input Desired Depth : \n")
    d = int(dIn)
    print("Largest At Depth ", d, "Is : ", LargestAtD(T, d))

  elif(choiceNum == 5): #count the number of nodes in the tree
    dIn = input("Input Desired Depth: \n")
    d = int(dIn)
```

```
    print("Number Of Nodes Within The Tree At Depth ", d," Is : ", CountAtD(T, d))

  elif(choiceNum == 6): #prints items at depth d
    dIn = input("Input Desired Depth: \n")
    d = int(dIn)
    print("The Items At Depth ", d," Are : ")
    PrintAtD(T, d)
    print()

  elif(choiceNum == 7): #returns the number of full nodes within the tree
    print("The Nuber Of Full Nodes Are: ", CountFullNodes(T))

  elif(choiceNum == 8): #returns the number of full leaf nodes
    print("The Number Of Full Leaf Nodes Are : ", FullLeaves(T))


  elif(choiceNum == 9): #Finds the depth level of node containing the item k
    kIn = input("Input The Desired Item Value To Be Searched: \n")
    k = int(kIn)
    print("The Depth Level Of",k," Is : ", DepthAtK(T, k))

  elif(choiceNum == 10): #exits the program
    dIn = input("Input Desired Depth For Test: \n")
    d = int(dIn)
    kIn = input("Input Desired K For Test: \n")
    k = int(dIn)

    times = []

    #part 1
    start = timeit.default_timer() # starts timer
    Height(T)
    stop = timeit.default_timer() # ends timer
    times.append(stop - start)

    #part 2
    arr=[]
    start = timeit.default_timer() # starts timer
    SortList(T, arr)
    stop = timeit.default_timer() # ends timer
    times.append(stop - start)

    #part 3
    start = timeit.default_timer() # starts timer
    SmallestAtD(T,d)
    stop = timeit.default_timer() # ends timer
    times.append(stop - start)

    #part 4
    start = timeit.default_timer() # starts timer
    LargestAtD(T, d)
    stop = timeit.default_timer() # ends timer
    times.append(stop - start)

    #part 5
    start = timeit.default_timer() # starts timer
    CountAtD(T,d)
    stop = timeit.default_timer() # ends timer
    times.append(stop - start)

    #part 6
    start = timeit.default_timer() # starts timer
    PrintAtD(T,d)
    stop = timeit.default_timer() # ends timer
    times.append(stop - start)

    #part 7
    start = timeit.default_timer() # starts timer
    CountFullNodes(T)
    stop = timeit.default_timer() # ends timer
```

```
    times.append(stop - start)

    #part 8
    start = timeit.default_timer() # starts timer
    FullLeaves(T)
    stop = timeit.default_timer() # ends timer
    times.append(stop - start)

    #part 9
    start = timeit.default_timer() # starts timer
    DepthAtK(T,k)
    stop = timeit.default_timer() # ends timer
    times.append(stop - start)

    print()
    print("The Following Time Executions Are In Seconds")
    for i in range(9):
      print("Part ", i+1 , " Time Execution: ", times[i])


  elif(choiceNum == 11): #exits the program
    print("Goodbye!")
    loop = False

  else:       #allows for re-entry incase of missed input
    print("Try Again")
  print("----------------------------------------")
```

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 03/24/2019
Discreption: Lab 3:
        The following program is desinged to explore methods involving the b-trees. Each
    part of the lab
        achieves a specific purpose with results shown.
"""

import math

class BTree(object):
    # Constructor
    def __init__(self,item=[],child=[],isLeaf=True,max_items=5):
        self.item = item
        self.child = child
        self.isLeaf = isLeaf
        if max_items <3: #max_items must be odd and greater or equal to 3
            max_items = 3
        if max_items%2 == 0: #max_items must be odd and greater or equal to 3
            max_items +=1
        self.max_items = max_items

def FindChild(T,k):
    for i in range(len(T.item)):
        if k < T.item[i]:
            return i
    return len(T.item)

def InsertInternal(T,i):
    if T.isLeaf:
        InsertLeaf(T,i)
    else:
        k = FindChild(T,i)
        if IsFull(T.child[k]):
            m, l, r = Split(T.child[k])
            T.item.insert(k,m)
            T.child[k] = l
            T.child.insert(k+1,r)
```

```
            k = FindChild(T,i)
        InsertInternal(T.child[k],i)

def Split(T):
    mid = T.max_items//2
    if T.isLeaf:
        leftChild = BTree(T.item[:mid])
        rightChild = BTree(T.item[mid+1:])
    else:
        leftChild = BTree(T.item[:mid],T.child[:mid+1],T.isLeaf)
        rightChild = BTree(T.item[mid+1:],T.child[mid+1:],T.isLeaf)
    return T.item[mid], leftChild,  rightChild

def InsertLeaf(T,i):
    T.item.append(i)
    T.item.sort()

def IsFull(T):
    return len(T.item) >= T.max_items

def Insert(T,i):
    if not IsFull(T):
        InsertInternal(T,i)
    else:
        m, l, r = Split(T)
        T.item =[m]
        T.child = [l,r]
        T.isLeaf = False
        k = FindChild(T,i)
        InsertInternal(T.child[k],i)


def PrintD(T,space):
    # Prints items and structure of B-tree
    if T.isLeaf:
        for i in range(len(T.item)-1,-1,-1):
            print(space,T.item[i])
    else:
        PrintD(T.child[len(T.item)],space+'   ')
        for i in range(len(T.item)-1,-1,-1):
            print(space,T.item[i])
            PrintD(T.child[i],space+'   ')
# ************* Below are the lab parts   *************

#Part 1: Printing the height of the tree
def Height(T):
    if T.isLeaf:
        return 0
    return 1+ Height(T.child[-1])

#Part 2: Appending a list through inorder transversal
def SortList(T, arr):
    if not T.isLeaf:
        track = 0
        #this for loop goes through a node's children inorder including themselves
        for i in range(len(T.child)):
            if i == FindChild(T, T.item[track]):
                arr.append(T.item[track])
                track +=1
            SortList(T.child[i], arr)
    else:
        #prints the node otherwise
        for i in range(len(T.item)):
            arr.append(T.item[i])
    return

#Part 3: Returns the smallest item at depth
def SmallestAtD(T, d):
    if d == 0:
        return T.item[0]
```

```python
    if T.isLeaf:
        return -math.inf
    return SmallestAtD(T.child[0], d-1)

#Part 4: Returns the largest item at depth
def LargestAtD(T, d):
    if d == 0:
        return T.item[-1]
    if T.isLeaf:
        return -math.inf
    return LargestAtD(T.child[-1], d-1)

#Part 5: Returns the number of nodes at the given depth
def CountAtD(T, d):
    if d == 0:
        return 1 #only returns a value of one if the depth value is 0
    if T.isLeaf:
        return -math.inf #returns a negative inf. if depth exceeds height
    count = 0
    for i in range(len(T.child)):
        count+=CountAtD(T.child[i], d-1)
    return count

#Part 6: Prints all items at given depth
def PrintAtD(T, d):
    if d == 0:
        for i in range(len(T.item)):
            print(T.item[i], end = ' ')
        return 1
    if T.isLeaf:
        print("Out Of Range", end = ' ')
        return -1

    else:
        check = 1
        for i in range(len(T.child)):
            if check > 0:
                check = PrintAtD(T.child[i], d-1)
    return 1

#Part 7: Returns the number of nodes that are full
def CountFullNodes(T):
    if len(T.item) == T.max_items:
        return 1
    count = 0
    for i in range(len(T.child)):
        count += CountFullNodes(T.child[i])
    return count

#Part 8: Returns the number of leaf nodes that are full
def FullLeaves(T):
    if len(T.item) == T.max_items and T.isLeaf: #also checks if its a leaf
        return 1
    count = 0
    for i in range(len(T.child)):
        count += CountFullNodes(T.child[i])
    return count

#Part 9: Prtins the level of which item k is found
def DepthAtK(T, k):
    if k in T.item:
        return 0
    if T.isLeaf:
        return -1 #returns a -1 if k is not within the tree.
    count = DepthAtK(T.child[FindChild(T,k)],k)
    if count >-1:
        return count +1
    return count

#the folloiwng definition is set to create the initial b-tree
```

```python
def BuildTree():
    L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200,
    2, 45, 6]
    T = BTree()
    for i in L:
        Insert(T,i)
    PrintD(T,'')
    return T
```

VI. Academic Dishonesty

Scholastic Dishonesty

Any student who commits an act of scholastic dishonesty is subject to discipline. Scholastic dishonesty includes, but not limited to cheating, plagiarism, collusion, the submission for credit of any work or materials that are attributable to another person.

- **Cheating**
  - Copying form the test paper of another student
  - Communicating with another student during a test
  - Giving or seeking aid from another student during a test
  - Possession and/or use of unauthorized materials during tests (i.e. Crib notes, class notes, books, etc)
  - Substituting for another person to take a test
  - Falsifying research data, reports, academic work offered for credit
- **Plagiarism**
  - Using someone's work in your assignments without the proper citations
  - Submitting the same paper or assignment from a different course, without direct permission of instructors
- **Collusion**
  - Unauthorized collaboration with another person in preparing academic assignments

Sign: ______________________ Date: 03/26/2019