Author: *Nguyen, Abram*
Assignment: *Lab 4 Report*
Course: *CS 2302 - Data Structures*
Instructor: *Fuentes, Olac*
T.A.: *Nath, Anindita*

Introduction:

In this lab, I'm demonstrating the use of B-Trees. I have implemented the B-Tree methods using nodes, each called 'child', every 'child' holds an array of elements called 'item'. An 'item' array can hold up to 5 nodes, but no more than that. B-Trees are a sort of combination of trees and arrays. In my lab, I used a B-Tree to organize values and search for values, taking advantage of the structure of the B-Tree.

Proposed solution design and implementation:

1) To compute the height of the tree, I traversed the tree recursively. Since all leaves in a B-Tree are at the same depth, this method is much easier to write compared to binary trees. I chose to just traverse the right side of the tree. If I come to a leaf, return 0. Otherwise, I return 1 and add to that number the count of the rest of the tree.

2) Extracting the items of the B-Tree into a sorted list was a little more difficult to write than I thought it would be. I would just have to start at the left-most element of the B-Tree and move to the right along the B-Tree. In my code, my plan was to traverse each node from left to right, then I would append the items of each node to an array. If I reached a leaf, I would just append the items of the leaf to the array. Otherwise, I would recursively iterate through the entire tree in ascending order, adding elements to the array, through each node from left to right.

3) Returning the smallest element at given depth 'd' was easy to conceptualize and write in code. I traversed recursively to the left-most child of a node, because this will always be the smallest element in every depth. Every traversal/recursive call, I subtract 1 from 'd' to count down to 0. I have 3 base cases written out below. I return infinity to show that a minimum could not be found. If depth 'd' is reached successfully, I just return the left-most item of that depth. The order of these base cases matter because of the way I traverse through the B-Tree.
    a) If root is empty
    b) If depth 'd' is reached
    c) If the tree has no existing depth 'd'

4) Returning the largest element at given depth 'd' is very similar to returning the smallest element in the previous method. I traversed recursively to the right-most child of a node, this will be the largest element in every depth. Every recursive call, I subtract 1 from 'd' to count down to 0. I have 3 base cases written out below. I return -infinity to show that a

maximum could not be found. If depth ‘d’ is reached successfully, I just return the right-most item of that depth, the largest. The order of these base cases matter because of the way I traverse through the B-Tree.

a) If root is empty
b) If depth ‘d’ is reached
c) If the tree has no existing depth ‘d’

5) Counting the number of nodes at given depth ‘d’ turned out to be easier than I thought it would be once I got it down on paper. I used a for loop to iterate through every child of a node and add the recursive call’s return value to variable ‘count’. I decided whether to count 0 or 1 using 3 base cases. If the root is empty or depth ‘d’ cannot be reached, return 0. If depth ‘d’ is reached successfully, return 1 (count that node). The order of these base cases matter because of the way I traverse through the B-Tree.
    a) If root is empty
    b) If depth ‘d’ is reached
    c) If the tree has no existing depth ‘d’

6) To print all the items at given depth ‘d’, I used ‘d’ as a count down to 0, like in my previous methods. As long as ‘d’ is not 0, I recursively iterate through every node in the tree. Once ‘d’ is reached (d = 0), I print all the items in a node using a for loop. I take advantage of recursion to iterate through the entire tree up to depth ‘d’.

7) Counting the number of full nodes in the B-Tree wasn’t too difficult to plan or write out. I traversed through the tree very similarly to how I traversed the tree in a previous method ‘NodesAtDepth’. I traversed every node, but I only needed to compare the number of elements in each node to the ‘max_items’ value, which was 5 in this case. This was 1 of my 2 base cases for this recursive method.
    a) If a node contains at least 5 elements, count 1 full node
    b) If a leaf has been reached and no full node is found, return 0

8) Counting the number of full leaves in the B-Tree turned out to be similar to the previous method, counting the number of full nodes. I traversed through every node like before. If a node contained at least 5 items and had been a leaf, then I’d counted 1. This was the only base case necessary for this method.
    a) If a node contains at least 5 elements AND is a leaf node, count 1 full leaf node

9) Searching for a key and returning its depth turned out to be a little more complicated to write out than I thought it would be. The concept seemed simple, though. First, I had to search for where the item exists, if at all. I used an iterator ‘i’ to help me traverse the B-Tree and provide a sort of direction on which child of a node to traverse to. The run time of this method of traversal is faster than the traversal of every single node and element. Using ‘i’, I traversed down the tree until I found that I’m at the end of the tree or I found ‘k’. If I can’t find ‘k’, I return -1. -1 is a number I would never return otherwise, if ‘k’

had been found. If ‘k’ was not found in the node but may still have been in the tree, I continued on to find ‘k’. If ‘k’ happens to be found, I would return my variable ‘depth’ plus 1.

<u>Experimental results:</u>

For most of my experiments, I decided to test my methods on 3 different B-Trees. I’ll show below the 3 lists of numbers that I’ve inserted into a B-Tree and experimented with.

These are the methods I’m testing using the 3 B-Trees:
(numbered corresponding to lab sheet)

1) Height(T)
2) Extract(T)
3) SmallestAtDepthD(T)
4) LargestAtDepthD(T)
5) NodesAtDepth(T,d)
6) PrintAtDepth(T,d)
9) SearchDepth(T,k)

I’m testing these other methods using different values:

7) FullNodes(T)
8) FullLeaves(T)

These are the 3 lists that I’ll insert into B-Trees to test methods 1, 2, 3, 4, 5, 6, and 9:

```
L = []
L = [50]
L = [6, 3, 16, 11, 7, 17, 14, 8, 5, 19, 15, 1, 2, 4, 18, 13, 9, 20, 10, 12, 21, 22, 0, -1, -2]
```

The first list is empty, the second will be a tree of only a single node, and the third is a tree of multiple nodes of varying sizes. I’ve included a picture of what the third list looks like in a B-Tree.

```
          22
          21
          20
          19
          18
     17
          16
          15
     14
          13
          12
          11
10
          9
          8
     7
          6
          5
          4
     3
          2
          1
          0
          -1
          -2
```

Results of testing methods 1, 2, 3, 4, 5, 6, and 9 on an **empty** tree ⇒ **expected results**

Since the B-Tree, in this case, is empty, there are a lot of results saying ‘0’ or a sort of default value. For example, in my methods returning smallest and largest values, they end up returning infinity and -infinity, respectively. Those are values that I’ve set the methods to return if nothing is found. When searching for the depths of different values, I tried searching for a negative number, 0, and a random value. Nothing was found, the method returned -1 each time.

```
Height of the B-Tree: 0

Extracted array of numbers: []

Smallest at depth -1 : inf
Smallest at depth 0 : inf
Smallest at depth 1 : inf

Largest at depth -1 : -inf
Largest at depth 0 : -inf
Largest at depth 1 : -inf

# of nodes at depth: -1 : 0
# of nodes at depth: 0 : 0
# of nodes at depth: 1 : 0

Items at depth -1 :
Items at depth 0 :
Items at depth 1 :

Depth of -10: -1
Depth of 0: -1
Depth of 3.5: -1
```

Results of testing methods 1, 2, 3, 4, 5, 6, and 9 on a tree containing **one node** ⇒ **expected results**

The B-Tree, in this case, contains only one node with one element. I only searched for numbers at depths -1, 0, and 1 just because the tree only has one level. I wanted to be sure that going outside depth 0 would yield expected results. -1 is an impossible depth of a tree, so I tested that in my cases to try and see if something interesting would happen, but nothing did. When searching for the depth of different values, I went with a similar approach to my previous experiment. I tested a negative number, 0, a number close to 50, and 50. Each case returned expected results, so I didn't find anything interesting there.

```
Height of the B-Tree: 0

Extracted array of numbers: [50]

Smallest at depth -1 : inf
Smallest at depth 0 : 50
Smallest at depth 1 : inf

Largest at depth -1 : -inf
Largest at depth 0 : 50
Largest at depth 1 : -inf

# of nodes at depth: -1 : 0
# of nodes at depth: 0 : 1
# of nodes at depth: 1 : 0

Items at depth -1 :
Items at depth 0 : 50
Items at depth 1 :

Depth of -10: -1
Depth of 0: -1
Depth of 50.1: -1
Depth of 50: 0
```

Results of testing methods 1, 2, 3, 4, 5, 6, and 9 on a tree containing **various nodes** ⇒ **expected results**

The B-Tree, in this case, contains several nodes of varying sizes. The tree contains 25 elements, has 9 nodes, and has a height of 2. These results look more like the results I got in the main lab project. When searching for specific nodes, I've made sure to check on depths that the tree doesn't actually have. Depths -1 and 3 returned expected results. The valid depths 0, 1, and 2 returned expected results, too. When searching for the depth of certain values, I included a little more variety in the test cases. I searched for the number at the root as well as several other values at varying depths. The value I knew did not exist in the tree wasn't found.

```
Height of the B-Tree: 2

Extracted array of numbers: [-2, -1, 0,
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15, 16, 17, 18, 19, 20, 21, 22]

Smallest at depth -1 : inf
Smallest at depth 0 : 10
Smallest at depth 1 : 3
Smallest at depth 2 : -2
Smallest at depth 3 : inf

Largest at depth -1 : -inf
Largest at depth 0 : 10
Largest at depth 1 : 17
Largest at depth 2 : 22
Largest at depth 3 : -inf

# of nodes at depth: -1 : 0
# of nodes at depth: 0 : 1
# of nodes at depth: 1 : 2
# of nodes at depth: 2 : 6
# of nodes at depth: 3 : 0

Items at depth -1 :
Items at depth 0 : 10
Items at depth 1 : 3 7 14 17
Items at depth 2 : -2 -1 0 1 2 4 5 6 8 9
11 12 13 15 16 18 19 20 21 22
Items at depth 3 :

Depth of 10:     0
Depth of 0:      2
Depth of -2:     2
Depth of 2.1:   -1
Depth of 7:      1
Depth of 12:     2
```

```
          22
          21
          20
          19
          18
     17
          16
          15
     14
          13
          12
          11
10
          9
          8
     7
          6
          5
          4
     3
          2
          1
          0
          -1
          -2
```

These are the lists I used to fill in B-Trees and test methods 'FullNodes' and 'FullLeaves' on:

```
L = []
L = [50]
L = [-5, 5, 2, 50, 10]
L = [50, 49, 48, 47, 46, 45, 44, 43, 51, 52, 53, 54, 55,
     56, 42, 41, 40, 50.5, 49.5, 48.5, 45.5, 46.5, 47.5]
L = [16, 11, 7, 17, 14, 8, 19, 15, 1, 2, 4, 18,
     13, 9, 20, 10, 12, 21, 22, 0, -1, -2]
```

Results of testing methods 7 and 8, ‘FullNodes’ and ‘FullLeaves’, on an **empty** tree ⇒ expected results

```
# of full nodes: 0
# of full leaves: 0
```

Results of testing methods 7 and 8, ‘FullNodes’ and ‘FullLeaves’, on an tree containing **one node that is full** ⇒ expected results

```
 50
 10
 5
 2
 -5

####################
# of full nodes: 1
# of full leaves: 1
```

Results of testing methods 7 and 8, ‘FullNodes’ and ‘FullLeaves’, on an tree containing **only full leaf nodes** ⇒ expected results

```
    56
    55
    54
    53
    52
51
    50.5
    50
    49.5
    49
    48.5
48
    47.5
    47
    46.5
    46
    45.5
45
    44
    43
    42
    41
    40
```

```
# of full nodes: 4
# of full leaves: 4
```

Results of testing methods 7 and 8, ‘FullNodes’ and ‘FullLeaves’, on an tree containing **nodes of different sizes** ⇒ expected results

```
     22
     21
     20
     19
     18
17
     16
     15
14
     13
     12
     11
10
     9
     8
7
     4
     2
1
     0
     -1
     -2
```

```
# of full nodes: 1
# of full leaves: 1
```

Conclusion:

I learned a lot about the B-Trees data structure by manipulating and playing with them in this lab. This is the first time I've worked with B-Trees. B-Trees seem like a fairly straightforward concept, but can be harder to code, in my opinion. I had most trouble just iterating through the trees at the beginning of learning the concept. After getting through this lab and experimenting with my methods, I now have a better understanding of B-Trees as a concept and as a data structure. I think I'd be able to solve problems involving B-Trees in the future.

Appendix:

```
"""
        Author:             Nguyen, Abram
        Assignment:         Lab 4
        Course:             CS 2302 - Data Structures
        Instructor:         Fuentes, Olac
        T.A.:               Nath, Anindita
        Last modified:      March 24, 2019

        Purpose of program: The purpose of this program is to demonstrate the attributes
                               and uses of B-Trees as a data structure. This program
                               handles B-Trees in different ways to showcase its use.
        """
        import math
        # Code to implement a B-tree
        # Programmed by Olac Fuentes
        # Last modified February 28, 2019

        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):
           # Determines value of c, such that k must be in subtree T.child[c], if k is in the BTree
           for i in range(len(T.item)):
```

```python
        if k < T.item[i]:
            return i
    return len(T.item)

def InsertInternal(T,i):
    # T cannot be Full
    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):
    #print('Splitting')
    #PrintNode(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)
```

```python
        T.item =[m]
        T.child = [l,r]
        T.isLeaf = False
        k = FindChild(T,i)
        InsertInternal(T.child[k],i)

def Print(T):
    # Prints items in tree in ascending order
    if T.isLeaf:
        for t in T.item:
            print(t,end=' ')
    else:
        for i in range(len(T.item)):
            Print(T.child[i])
            print(T.item[i],end=' ')
        Print(T.child[len(T.item)])

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+'    ')

"""
###############################################################################
# Programmed by Abram Nguyen ##################################################
# Last modified March 24, 2019 ################################################
###############################################################################
"""
# Compute the height of the tree ##############################################
def Height(T):
    if T.isLeaf:
        return 0
    return 1 + Height(T.child[-1]) #add 1 and iterate to next child

# Extract the items in the B-tree into a sorted list. #########################
def Extract(T, L):
    if T.isLeaf:
```

```python
        for t in T.item:
            L += [t] #append items of leaf to list
    else:
        for i in range(len(T.item)):
            Extract(T.child[i], L) #traverse children recursively
            L += [T.item[i]] #append items to list
        Extract(T.child[len(T.item)], L) #last child

# Return the minimum element in the tree at a given depth d. ##################
def SmallestAtDepthD(T,d):
    if not T.item: #check if root is empty first
        return math.inf
    if d == 0: #reached depth 'd'
        return T.item[0]
    if T.isLeaf or d < 0: #if tree has no depth 'd'
        return math.inf
    return SmallestAtDepthD(T.child[0],d-1) #traverse to 'd', left most child

# Return the maximum element in the tree at a given depth d. ##################
def LargestAtDepthD(T,d):
    if not T.item: #check if root is empty first
        return -math.inf
    if d == 0: #reached depth 'd'
        return T.item[-1]
    if T.isLeaf or d < 0: #if tree has no depth 'd'
        return -math.inf
    return LargestAtDepthD(T.child[-1],d-1) #traverse to 'd', right most child

# Return the number of nodes in the tree at a given depth d.
def NodesAtDepth(T, d):
    if not T.item: #check if root is empty first
        return 0
    if d == 0: #reached depth 'd'
        return 1
    if T.isLeaf or d < 0: #if tree has no depth 'd'
        return 0
    count = 0
    for i in range(len(T.child)): #for every child:
        count += NodesAtDepth(T.child[i],d-1) #count each child/node
    return count

# Print all the items in the tree at a given depth d. #########################
```

```python
def PrintAtDepth(T,d):
   if d == 0: #reached depth 'd'
       for i in range(len(T.item)): #print every element of item
               print(T.item[i], end = " ")
   else:
       for j in range(len(T.child)): #traverse to every child
           PrintAtDepth(T.child[j],d-1)


# Return the number of nodes in the tree that are full.
def FullNodes(T):
   if len(T.item) >= T.max_items: #if full node is found
       return 1
   if T.isLeaf: #full node not found here
       return 0
   count = 0
   for i in range(len(T.child)):
       count += FullNodes(T.child[i]) #traverse through every child, keep count
   return count


# Return the number of leaves in the tree that are full.
def FullLeaves(T):
   if len(T.item) >= T.max_items: #if full node is found and...
       if T.isLeaf: #...full node is a leaf
           return 1
   count = 0
   for i in range(len(T.child)):
       count += FullLeaves(T.child[i]) #traverse through every child, keep count
   return count



# Given a key k, return the depth at which it is found in the tree, ###########
#   or -1 if k is not in the tree.
def SearchDepth(T, k):
   i = 0
   #search a node for k w/ iterator, add 1 to i until/while...
   while i < len(T.item) and T.item[i] < k:
       i += 1

   if len(T.item) == i or T.item[i] > k: #k not found in node and...
       if T.isLeaf: #...k not found in tree
           return -1
       else: #k not found in node but IS found in tree:
           depth = SearchDepth(T.child[i], k)
```

```
            if depth >= 0: #if depth is valid (not negative)
                return depth+1 #i of element 1 is 0, so adjust depth by 1
            return -1
    return 0

"""
################################################################################
################################# METHOD CALLS #################################
################################################################################
"""


#L = []
#L = [-5, 5, 2, 50, 10]
#L = [50, 49, 48, 47, 46, 45, 44, 43, 51, 52, 53, 54, 55, 56, 42, 41, 40, 50.5, 49.5, 48.5, 45.5,
46.5, 47.5]
#L = [16, 11, 7, 17, 14, 8, 19, 15, 1, 2, 4, 18, 13, 9, 20, 10, 12, 21, 22, 0, -1, -2]
L = [6, 3, 16, 11, 7, 17, 14, 8, 5, 19, 15, 1, 2, 4, 18, 13, 9, 20, 10, 12, 21, 22, 0, -1, -2]


T = BTree()
for i in L:
    print('Inserting',i)
    Insert(T,i)
    PrintD(T,'')
    #Print(T)
    print('\n####################################')



# CALCULATE HEIGHT OF B-TREE -----------------------------------------------------
print("Height of the B-Tree:", Height(T), "\n")


# EXTRACT INTO SORTED ARRAY ------------------------------------------------------
A = []
Extract(T, A)
print("Extracted array of numbers:", A, "\n")


# SMALLEST AT DEPTH ---------------------------------------------------------------
for i in range(5):
    i -= 1
    print("Smallest at depth", i, ":", SmallestAtDepthD(T,i))
print()


# LARGEST AT DEPTH ---------------------------------------------------------------
```

```
for i in range(5):
   i -= 1
   print("Largest at depth", i, ":", LargestAtDepthD(T,i))
print()

# COUNT NODES AT EVERY DEPTH --------------------------------------------------
for i in range(5):
   i -= 1
   print("# of nodes at depth:", i, ":", NodesAtDepth(T, i))
print()

# PRINT ITEMS AT DEPTH --------------------------------------------------------
for i in range(5):
   i -= 1
   print("Items at depth", i, ":", end=" ")
   PrintAtDepth(T,i)
   print()
print()

# COUNT FULL NODES ------------------------------------------------------------
print("# of full nodes:", FullNodes(T))

# COUNT FULL LEAVES -----------------------------------------------------------
print("# of full leaves:", FullLeaves(T))
print()

# DEPTHS OF VARIOUS VALUES ----------------------------------------------------
print("Depth of 10:    ", SearchDepth(T, 10))
print("Depth of 0:     ", SearchDepth(T, 0))
print("Depth of -2:    ", SearchDepth(T, -2))
print("Depth of 2.1:  ", SearchDepth(T, 2.1))
print("Depth of 7:     ", SearchDepth(T, 7))
print("Depth of 12:    ", SearchDepth(T, 12))
################################################################################

#End of program
```

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.

- Abram Nguyen