```python
# -*- coding: utf-8 -*-
"""
Created on Mon Jun 18 20:26:14 2018

@author: Tawanda Vera
"""
from __future__ import (nested_scopes, generators, division, absolute_import,
                        with_statement, print_function, unicode_literals)
import bisect
import random
import unittest

################################################################################
# Problem 1:  Make Changes to the B-Tree Code

"""
  Description: This adopted code provides the BTree class, based on support from
    the BTreeNode class.  The BTreeNode class is also implemented in this
    module.
"""
################################################################################
# Given an implementation of a B-Tree code structure, continue to use Python
# to do the following:
# Part A: Implement the function __print_tree, this function should print all
# keys in the B-Tree in increasing order.
# For example.. the output of the test example
# (after completing all insert operations) in the main part in the code is:
#  B-Tree: 0 1 2 3 4 5 6 7 8 9

"""Answering Part A can be done by creating a tree t = BTree(10) with a t
of 10 and then printing t. Alternatively, the we can Transverse the B-Tree and
get the same result. Transversing is prefered as it will assist in answering
the search functions.TThe part of transversing the B-Tree code was very
challenging
"""

# Part B:
# Implement the function  def search(x, k, nil=None) which should return a
# pointer to the node with key=k in the B-tree T where x is the root node in
# the B-tree T.

# Part C:
# Update def search(x, k, nil=None) function to print all keys which would be
# compared to the value k before returning a pointer to the node in B-Tree T
# with key=k.
###############################################################################

# Part A: Given an implementation of the following B-Tree code structure:

#==============================================================================
class _BNode(object):
    __slots__ = ["tree", "contents", "children"]

    def __init__(self, tree, contents=None, children=None):
        self.tree = tree
        self.contents = contents or []
        self.children = children or []
```

```python
    if self.children:
        assert len(self.contents) + 1 == len(self.children)
        "one more child than data item required"


def __repr__(self):
    name = getattr(self, "children", 0) and "Branch" or "Leaf"
    return "<%s %s>" % (name, ", ".join(map(str, self.contents)))

def lateral(self, parent, parent_index, dest, dest_index):

    if parent_index > dest_index:
        dest.contents.append(parent.contents[dest_index])
        parent.contents[dest_index] = self.contents.pop(0)
        if self.children:
            dest.children.append(self.children.pop(0))

    else:
        dest.contents.insert(0, parent.contents[parent_index])
        parent.contents[parent_index] = self.contents.pop()
        if self.children:
            dest.children.insert(0, self.children.pop())

def shrink(self, ancestors):
    parent = None
    if ancestors:
        parent, parent_index = ancestors.pop()
        # try to lend to the left neighboring sibling
        if parent_index:
            left_sib = parent.children[parent_index - 1]
            if len(left_sib.contents) < self.tree.order:
                self.lateral(
                        parent, parent_index, left_sib, parent_index - 1)
                return

        # try the right neighbor
        if parent_index + 1 < len(parent.children):
            right_sib = parent.children[parent_index + 1]
            if len(right_sib.contents) < self.tree.order:
                self.lateral(
                        parent, parent_index, right_sib, parent_index + 1)
                return
    center = len(self.contents) // 2
    sibling, push = self.split()


    if not parent:
        parent, parent_index = self.tree.BRANCH(
                self.tree, children=[self]), 0
        self.tree.root = parent
    # pass the median up to the parent
    parent.contents.insert(parent_index, push)
    parent.children.insert(parent_index + 1, sibling)
    if len(parent.contents) > parent.tree.order:
        parent.shrink(ancestors)

def grow(self, ancestors):
```

```
        parent, parent_index = ancestors.pop()
        minimum = self.tree.order // 2
        left_sib = right_sib = None

        # try to borrow from the right sibling
        if parent_index + 1 < len(parent.children):
            right_sib = parent.children[parent_index + 1]
            if len(right_sib.contents) > minimum:
                right_sib.lateral(parent, parent_index + 1, self, parent_index)
                return

        # try to borrow from the left sibling
        if parent_index:
            left_sib = parent.children[parent_index - 1]
            if len(left_sib.contents) > minimum:
                left_sib.lateral(parent, parent_index - 1, self, parent_index)
                return

        # consolidate with a sibling - try left first
        if left_sib:
            left_sib.contents.append(parent.contents[parent_index - 1])
            left_sib.contents.extend(self.contents)
            if self.children:
                left_sib.children.extend(self.children)
            parent.contents.pop(parent_index - 1)
            parent.children.pop(parent_index)
        else:
            self.contents.append(parent.contents[parent_index])
            self.contents.extend(right_sib.contents)
            if self.children:
                self.children.extend(right_sib.children)
            parent.contents.pop(parent_index)
            parent.children.pop(parent_index + 1)

        if len(parent.contents) < minimum:
            if ancestors:
                # parent is not the root
                parent.grow(ancestors)
            elif not parent.contents:
                # parent is root, and its now empty
                self.tree.root = left_sib or self

    def split(self):
        center = len(self.contents) // 2
        median = self.contents[center]
        sibling = type(self)(
                self.tree,
                self.contents[center + 1:],
                self.children[center + 1:])
        self.contents = self.contents[:center]
        self.children = self.children[:center + 1]
        return sibling, median

    def insert(self, index, item, ancestors):
        self.contents.insert(index, item)
        if len(self.contents) > self.tree.order:
            self.shrink(ancestors)
```

```python
    def remove(self, index, ancestors):
        minimum = self.tree.order // 2
        if self.children:
            # try promoting from the right subtree first,
            # but only if it won't have to resize
            additional_ancestors = [(self, index + 1)]
            descendent = self.children[index + 1]
            while descendent.children:
                additional_ancestors.append((descendent, 0))
                descendent = descendent.children[0]
            if len(descendent.contents) > minimum:
                ancestors.extend(additional_ancestors)
                self.contents[index] = descendent.contents[0]
                descendent.remove(0, ancestors)
                return

            # fall back to the left child
            additional_ancestors = [(self, index)]
            descendent = self.children[index]
            while descendent.children:
                additional_ancestors.append(
                        (descendent, len(descendent.children) - 1))
                descendent = descendent.children[-1]
            ancestors.extend(additional_ancestors)
            self.contents[index] = descendent.contents[-1]
            descendent.remove(len(descendent.children) - 1, ancestors)
        else:
            self.contents.pop(index)
            if len(self.contents) < minimum and ancestors:
                self.grow(ancestors)
#===============================================================================

class BTree:
    BRANCH = LEAF = _BNode
    def __init__(self, order):
        self.order = order
        self.root = self._bottom = self.LEAF(self)

    def _path_to(self, item):
        current = self.root
        ancestry = []
        while getattr(current, "children", None):
            index = bisect.bisect_left(current.contents, item)
            ancestry.append((current, index))
            if index < len(current.contents)  and current.contents[index] == item:
                return ancestry
            current = current.children[index]

        index = bisect.bisect_left(current.contents, item)
        ancestry.append((current, index))
        present = index < len(current.contents)
        present = present and current.contents[index] == item
        return ancestry
#===============================================================================
                            # Part B:
#===============================================================================
```

```python
    # Implement the function  def search(x, k, nil=None) which should
    # return a pointer to the node with key=k in the B-tree T where x is
    # the root node in the B-tree T.
    """I understood that the problem required a depth first traversal of the
    B-Tree. Therefore, attempts were made on the code to get item 'k' with its
    ancestors using the __present__ method. The method is boolen class which
    assert the presents of the ancestors. The attempt is also made with the
    __contains__ attribute of the tree. This gives an output of the branch or
    root(x), the leaf (k) and the nil index."""

    def __present__(self, item, ancestors):
        last, index = ancestors[-1]
        return index < len(last.contents) and last.contents[index] == item

    def insert(self, item):
        current = self.root
        ancestors = self._path_to(item)
        node, index = ancestors[-1]
        while getattr(node, "children", None):
            node = node.children[index]
            index = bisect.bisect_left(node.contents, item)
            ancestors.append((node, index))
        node, index = ancestors.pop()
        node.insert(index, item, ancestors)

    def remove(self, item):
        current = self.root
        ancestors = self._path_to(item)
        if self.__present__(item, ancestors):
            node, index = ancestors.pop()
            node.remove(index, ancestors)
        else:
            raise ValueError("%r not in %s" % (item, self.__class__.__name__))

    def __contains__(self, item):
        return item, self._path_to(item)

#==============================================================================
                                # Part C:
#==============================================================================

    # Update def search(x, k, nil=None) function to print all keys which
    # would be compared to the value k before returning a pointer to the node
    # in B-Tree T with key=k.
    """ The _iter_ method updates the items position in the structure by
    recursively comparing the node (k) with all keys before returning a pointer
    to the node in Bulkmethod below. Part C is answered by the iter method and
    the build_bulkloaded_leaves class method which prints all the keys.
    In this module the bulk class method is hidden to improve efficiency of the
    code."""

    def __iter__(self):
        def _recurse_(node):
            if node.children:
                for child, item in zip(node.children, node.contents):
                    for child_item in _recurse_(child):
                        yield child_item
```

```
                yield item
            for child_item in _recurse_(node.children[-1]):
                yield child_item
        else:
            for item in node.contents:
                yield item
    for item in _recurse_(self.root):
        yield item

def print_order(self):
    """Print an level-order representation."""
    current = self.root
    this_level = current
    while this_level:
        next_level = []
        output = ""

    for node in this_level:
                if node.children:
                    next_level.extend(node.children)
                    output += str(node.keys) + " "
    print(output)

#==============================================================================
     # Print Tree Structure
def __repr__(self):
    def recurse(node, accum, depth):
        accum.append(("  " * depth) + "=|=" + repr(node))
        for node in getattr(node, "children", []):
            recurse(node, accum, depth + 1)
    accum = []
    recurse(self.root, accum, 0)
    return " \n ".join(accum)

@classmethod
def bulkload(cls, items, order):
    tree = object.__new__(cls)
    tree.order = order
    leaves = tree._build_bulkloaded_leaves(items)
    tree._build_bulkloaded_branches(leaves)
    return tree
#=========================Part C===============================================
def _build_bulkloaded_leaves(self, items):
    minimum = self.order // 2
    leaves, seps = [[]], []
    for item in items:
        if len(leaves[-1]) < self.order:
            leaves[-1].append(item)
        else:
            seps.append(item)
            leaves.append([])
    if len(leaves[-1]) < minimum and seps:
        last_two = leaves[-2] + [seps.pop()] + leaves[-1]
        leaves[-2] = last_two[:minimum]
        leaves[-1] = last_two[minimum + 1:]
        seps.append(last_two[minimum])
    return [self.LEAF(self, contents=node) for node in leaves], seps
```

```python
#==============================================================================
    def _build_bulkloaded_branches(self, leaves, seps):
        minimum = self.order // 2
        levels = [leaves]
        while len(seps) > self.order + 1:
            items, nodes, seps = seps, [[]], []
            for item in items:
                if len(nodes[-1]) < self.order:
                    nodes[-1].append(item)
                else:
                    seps.append(item)
                    nodes.append([])
            if len(nodes[-1]) < minimum and seps:
                last_two = nodes[-2] + [seps.pop()] + nodes[-1]
                nodes[-2] = last_two[:minimum]
                nodes[-1] = last_two[minimum + 1:]
                seps.append(last_two[minimum])
            offset = 0
            for i, node in enumerate(nodes):
                children = levels[-1][offset:offset + len(node) + 1]
                nodes[i] = self.BRANCH(self, contents=node, children=children)
                offset += len(node) + 1
            levels.append(nodes)
        self.root = self.BRANCH(self, contents=seps, children=levels[-1])

#==============================================================================
# Test to check if the algorithms meets the B-Tree constraints

class BTreeTests(unittest.TestCase):

    def test_additions(self):

        bt = BTree(20)

        l = range(2000)

        for i, item in enumerate(l):
            bt.insert(item)
            self.assertEqual(list(bt), l[:i + 1])

    def test_bulkloads(self):
        bt = BTree.bulkload(range(2000), 20)
        self.assertEqual(list(bt), range(2000))

    def test_removals(self):
        bt = BTree(20)
        l = range(2000)
        map(bt.insert, l)
        rand = l[:]
        random.shuffle(rand)

        while l:
            self.assertEqual(list(bt), l)
            rem = rand.pop()
            l.remove(rem)
            bt.remove(rem)
        self.assertEqual(list(bt), l)
```

```python
    def test_insert_regression(self):
        bt = BTree.bulkload(range(2000), 50)
        for i in range(100000):
            bt.insert(random.randrange(2000))

#==============================================================================

if __name__ == '__main__':

    #unittest.main()
    print("\n==============Part A ========================\n")

    print("B-Tree with minimum degree of 10:")
    b = BTree(10)

    for i in range(0,10):

        b.insert(i)

    print (b)

    print("\n======================================\
          \n")

    print("B-Tree with minimum degree of 3:")
    bt = BTree(3)

    for i in range(0,10):

        bt.insert(i)

    print (bt)

#===============================================================================

    print("\n==============Part B ========================\n")

    print("Return a pointer to the node with key=k in the B-tree T where x \
          is the root node in the B-tree T (For min degree 10) \
          search item 4: \n")

    sb = b.__contains__(4)

    print(sb)

    print("\n===================================================\
          \n")

    print("\n Return a pointer to the node (leaf) with key=k in the B-tree T\
          where x is the root node (Branch) in the B-tree T (For min degree 3)\
          search item 4: \n")

    sbt = bt.__contains__(4)

    print(sbt)
```

```
    print("\n=====================================================\
          \n")
#==============================================================================
# Applications in Quant Finance
#===============================================================================
"""
Decision trees are a supervised classification technique that utilise a tree
structure to partition the feature space into recursive subsets via a
"decision" at each node of the tree. This process continues until there is
no more predictive power to be gained by partitioning. A decision tree
provides a naturally interpretable classification mechanism when compared to
the more "black box" opaque approaches of discriminant analysers and are
particularly appealing to alpha generation modelling because of it.

With Machine Learning we are able to create a large quantity of classifiers
from the same base model and train them all with varying parameters. Then
combine the results of the prediction in an average to hopefully obtain a
prediction accuracy that is greater than that brought on by any of the
individual constituents. One of the most widespread ensemble methods is
that of a Random Forest, which takes multiple decision tree learners
 (usually tens of thousands or more) and combines the predictions.
Because computers excel at quickly and accurately manipulating, storing,
 and retrieving data, databases are often maintained electronically using
 a database management system. It is not uncommon for a database to contain
 millions of records requiring many gigabytes of storage. For examples,
 TELSTRA, an Australian telecommunications company, maintains a customer
 billing database with 51 billion rows and 4.2 terabytes of data.
 In order for a database to be useful and usable, it must support
 the desired operations, such as retrieval and storage, quickly.
 Because databases cannot typically be maintained entirely in memory,
 b-trees are often used to index the data and to provide fast access.
 For example, searching an unindexed and unsorted database containing n
 key values will have a worst case running time of O(n); if the same data
 is indexed with a b-tree, the same search operation will run in O(log n).
 To perform a search for a single key on a set of one million keys (1,000,000)
 , a linear search will require at most 1,000,000 comparisons. If the same
 data is indexed with a b-tree of minimum degree 10, 114 comparisons will be
 required in the worst case. Clearly, indexing large amounts of data can
 significantly improve search performance. Although other balanced tree
 structures can be used, a b-tree also optimizes costly disk accesses that
 are of concern when dealing with large data sets.
 """
 #=============================================================================
```