Practical 3

> Implementation of B+ Tree Indexing for Database query processing
> Input-output for Select Query on exact Match, Range Query, Insert , delete Query.
> Analysis report

Implementation

```
#!usr/bin/python3
# coding=utf-8
"""
some terminology of python:
_var => (convention only) underscore prefix is just a hint to programmer that a variable or method starting with a
var_ => (convention only) to brake name conflict
__var => “dunders” (name mangling) rewrite the attribute name in order to avoid naming conflicts in subclasses.
            interpreter changes the name of the variable in a way that makes it harder to create collisions when t
"""
```

```
import math
import logging
import os
import random
from datetime import datetime
from bisect import bisect_right, bisect_left
from collections import deque

__author__ = "Gahan Saraiya"
DEBUG = False
LOG_DIR = "."
logger = logging.getLogger('bPlusTree')
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s [%(name)-8s - %(levelname)s]: %(message)s',
                    datefmt='[%Y-%d-%m_%H.%M.%S]',
                    filename=os.path.join(LOG_DIR, 'b_plus_tree.log'),
                    filemode='w')
ch = logging.StreamHandler()  # create console handler with a higher log level
ch.setLevel(logging.DEBUG)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s [%(name)-8s - %(levelname)s]: %(message)s')
ch.setFormatter(formatter)
# add the handlers to the logger
logger.addHandler(ch)


def log(*msg):
    if DEBUG:
        logger.debug(msg)
    else:
        pass


class InternalNode(object):
    """
    Class : B+ Tree Internal Node
    represents internal (non-leaf) node in B+ tree
    """
    def __init__(self, degree=4):
        """
        initialize B tree node
        :param degree: specify degree of btree  # default degree set to 4
        """
        self.degree = degree
        self.keys = []  # store keys/data values
        self.children = []  # store child nodes (list of instances of BtreeNode); empty list if node is leaf node
        self.parent = None

    def __repr__(self):
        return " | ".join(map(str, self.keys))

    @property
    def is_leaf(self):
        return False
```

```
    @property
    def total_keys(self):
        return len(self.keys)

    @property
    def is_balanced(self):
        # return False if total keys exceeds max accommodated keys (degree - 1)
        return self.total_keys <= self.degree - 1

    @property
    def is_full(self):
        return self.total_keys >= self.degree

    @property
    def is_empty(self):
        return self.total_keys < (self.degree + 1) // 2


class LeafNode(object):
    """
    Class : B+ Tree Leaf Node
    represents leaf node in B+ tree
    """
    def __init__(self, degree=4):
        self.degree = degree
        self.keys = []  # data values
        self.sibling = None  # sibling node to point
        self.parent = None  # parent node - None for root node

    def __repr__(self):
        return " | ".join(map(str, self.keys))

    @property
    def is_leaf(self):
        return True

    @property
    def total_keys(self):
        return len(self.keys)

    @property
    def is_balanced(self):
        # return False if total keys exceeds max accommodated data (degree - 1)
        return self.total_keys < self.degree

    @property
    def is_full(self):
        return not self.is_balanced

    @property
    def is_empty(self):
        return self.total_keys < math.floor(self.degree / 2)


class BPlusTree(object):
    def __init__(self, degree=4):
        self.degree = degree
        self.__root = LeafNode(degree=degree)
        self.__leaf = self.__root

    def search_key(self, start_node, value):
        """

        :param start_node: get root or any non leaf node
        :param value: value to be search
        :return: most matching leaf node
        """
        if start_node.is_leaf:
            _index = bisect_left(start_node.keys, value)
            return _index, start_node
        else:
            _index = bisect_right(start_node.keys, value)
            return self.search_key(start_node.children[_index], value)

    def search(self, start=None, end=None):
        """

        :param start: specify start node to search range for
        :param end: specify end node for range search
```

```
        :return:
        """
        _result = []
        node = self.__root
        leaf = self.__leaf

        if start is None:
            while True:
                for value in leaf.keys:
                    if value <= end:
                        _result.append(value)
                    else:
                        return _result
                if leaf.sibling is None:
                    return _result
                else:
                    leaf = leaf.sibling
        elif end is None:
            _index, leaf = self.search_key(node, start)
            _result.extend(leaf.keys[_index:])  # equivalent to _result + leaf
            while True:
                if leaf.sibling is None:
                    return _result
                else:
                    leaf = leaf.sibling
                    _result.extend(leaf.keys)
        else:
            if start == end:
                _index, _node = self.search_key(node, start)
                try:
                    if _node.keys[_index] == start:
                        _result.append(_node.keys[_index])
                        return _result
                    else:
                        return _result
                except IndexError:
                    return _result
            else:
                _index1, _node1 = self.search_key(node, start)
                _index2, _node2 = self.search_key(node, end)
                if _node1 is _node2:
                    if _index1 == _index2:
                        return _result
                    else:
                        _result.extend(_node1.keys[_index1:_index2])
                        return _result
                else:
                    _result.extend(_node1.keys[_index1:])
                    node_ = _node1
                    while True:
                        if _node1.sibling == _node2:
                            _result.extend(_node2.keys[:_index2 + 1])
                            return _result
                        else:
                            try:
                                _result.extend(node_.sibling.keys)
                                node_ = node_.sibling
                            except AttributeError:
                                return _result

    def traverse(self, _node):
        _result = []
        _result.extend(_node.keys)
        if getattr(_node, "sibling", None) is None:
            return _result
        for i in range(0, len(_node.sibling))[::-1]:
            _result.extend(self.traverse(_node.sibling[i]))
        while True:
            pass

    def pretty_print(self):
        # print("B+ Tree:")
        queue, height = deque(), 0
        queue.append([self.__root, height])
        while True:
            try:
                node, height_ = queue.popleft()
                # print("adding node: {}".format(node))
            except IndexError:
                return
```

```
            else:
                if not node.is_leaf:
                    print("Internal Node : {:} \theight >> {}".format(node.keys, height_))
                    if height_ == height:
                        height += 1
                    queue.extend([[i, height] for i in node.children])
                else:
                    print("Leaf Node     : {} \theight >> {}".format([i for i in node.keys], height_))

    def insert(self, value):
        # log("parent:{} leaf:{} node:{}\tkeys:{}\t children:{}".format(node.parent, node.is_leaf, node, node.keys

        def split_leaf_node(node):
            log("splitting leaf node: {}".format(node.keys))
            mid = self.degree // 2  # integer division in python3
            new_leaf = LeafNode(self.degree)
            new_leaf.keys = node.keys[mid:]
            if node.parent is None:  # None and 0 are to be treated as different value
                parent_node = InternalNode(self.degree)  # create new parent for node
                parent_node.keys, parent_node.children = [node.keys[mid]], [node, new_leaf]
                node.parent = new_leaf.parent = parent_node
                self.__root = parent_node
            else:
                _index = node.parent.children.index(node)
                node.parent.keys.insert(_index, node.keys[mid])
                node.parent.children.insert(_index + 1, new_leaf)
                new_leaf.parent = node.parent
                if not node.parent.is_balanced:
                    split_internal_node(node.parent)
            node.keys = node.keys[:mid]
            node.sibling = new_leaf
            log("{} --- {} --- {}".format(node, node.sibling, self.__root.children))

        def split_internal_node(node_):
            mid = self.degree // 2  # integer division in python3
            new_node = InternalNode(self.degree)
            new_node.keys = node_.keys[mid:]
            new_node.children = node_.children[mid:]
            new_node.parent = node_.parent
            for child in new_node.children:
                child.parent = new_node  # assign parent to every new child of current node
            if node_.parent is None:  # again Note that None and 0 are not same but both treated as False in boole
                # need to make new root if we are to split root node
                new_root = InternalNode(self.degree)
                new_root.keys = [node_.keys[mid - 1]]
                new_root.children = [node_, new_node]
                node_.parent = new_node.parent = new_root  # set parent of newly created node
                self.__root = new_root  # set new ROOT node
            else:
                # if node is not root internal node
                _index = node_.parent.children.index(node_)
                node_.parent.keys.insert(_index, node_.keys[mid - 1])
                node_.parent.children.insert(_index + 1, new_node)
                if not node_.parent.is_balanced:
                    split_internal_node(node_.parent)
            node_.keys = node_.keys[:mid - 1]
            node_.children = node_.children[:mid]
            return node_.parent

        def insert_node(_node):
            log("inserting : {} in node: {} having children: {}".format(value, _node.keys, getattr(_node, "childre
            if _node.is_leaf:  # logic for leaf node
                log("node: {} is leaf".format(_node))
                _index = bisect_right(_node.keys, value)  # bisect and get index value of where to insert value in
                _node.keys.insert(_index, value)
                if not _node.is_balanced:
                    split_leaf_node(_node)
                    log("----------- Tree status after split---------------")
                    log(self.__root)
                    log(self.__root.children)
                    log(_node.parent.children)
                    log(getattr(self.__root, "children", "NULL"))
                else:
                    return
            else:  # logic for internal node
                if not _node.is_balanced:
                    self.insert(split_internal_node(_node))
                else:
                    _index = bisect_right(_node.keys, value)
                    log(_node.keys, _node.children, _index)
```

```
                    insert_node(_node.children[_index])

        insert_node(self.__root)

    @staticmethod
    def traverse_left_to_right(node, index):
        if node.children[index].is_leaf:
            node.children[index + 1].keys.insert(0, node.children[index].keys[-1])
            node.children[index].keys.pop()
            node.keys[index] = node.children[index + 1].keys[0]
        else:
            node.children[index + 1].children.insert(0, node.children[index].children[-1])
            node.children[index].children[-1].parent = node.children[index + 1]
            node.children[index + 1].keys.insert(0, node.keys[index])
            node.children[index].children.pop()
            node.children[index].keys.pop()

    @staticmethod
    def traverse_right_to_left(node, index):
        if node.children[index].is_leaf:
            node.children[index].keys.append(node.children[index + 1].keys[0])
            node.children[index + 1].keys.remove(node.children[index + 1].keys[0])
            node.keys[index] = node.children[index + 1].keys[0]
        else:
            node.children[index].children.append(node.children[index + 1].children[0])
            node.children[index + 1].children[0].parent = node.children[index]
            node.children[index].keys.append(node.keys[index])
            node.keys[index] = node.children[index + 1].children[0]
            node.children[index + 1].children.remove(node.children[index + 1].children[0])
            node.children[index + 1].keys.remove(node.children[index + 1].keys[0])

    def delete(self, delete_value):
        def merge(node, index):
            if node.children[index].is_leaf:
                node.children[index].keys = node.children[index].keys + node.children[index + 1].keys
                node.children[index].sibling = node.children[index + 1].sibling
            else:
                node.children[index].keys = node.children[index].keys + [node.keys[index]] + node.children[
                    index + 1].keys
                node.children[index].children = node.children[index].children + node.children[index + 1].children
            node.children.remove(node.children[index + 1])
            node.keys.remove(node.children[index])
            if node.keys:
                return node
            else:
                node.children[0].parent = None
                self.__root = node.children[0]
                del node
                return self.__root

        def delete_node(value, node):
            log("deleting {} from node: {}".format(value, node))
            if node.is_leaf:
                log("node is leaf")
                _index = bisect_left(node.keys, value)
                try:
                    node_ = node.keys[_index]
                except IndexError:
                    return False
                else:
                    if node_ != value:
                        return False
                    else:
                        node.keys.remove(value)
                        return True
            else:
                log("traversing internal node for deleting value")
                _index = bisect_right(node.keys, value)
                log("encountered index: {} having child values: {}".format(_index, node.children[_index]))
                if _index <= len(node.keys):
                    # print(node.children[_index].is_leaf)
                    # print(node.children[_index], node.children[_index].total_keys, node.children[_index].degree
                    if not node.children[_index].is_empty:
                        return delete_node(value, node.children[_index])
                    elif not node.children[_index - 1].is_empty:
                        self.traverse_left_to_right(node, _index - 1)
                        return delete_node(value, node.children[_index])
                    else:
                        return delete_node(value, merge(node, _index))
        delete_node(delete_value, self.__root)
```

```
def _test():
    # test_lis = [0, 1, 11, 1, 2, 22, 13, 14, 4, 5, 23, 1, 51, 12, 31]
    test_lis = [10, 1, 159, 200, 18, 90, 8, 17, 9]
    # test_lis = range(10)
    b = BPlusTree(degree=4)
    for val in test_lis:
        b.insert(val)
        print("----------- B+ TREE AFTER INSERT : {:3d} -----------".format(val))
        b.pretty_print()
    # print("searching range.....")
    # result = b.search_range(1, 12)
    search_start, search_end = 1, 23
    print("----- Searching in batch for {} to {} -----".format(search_start, search_end))
    print("Result*: {} \n (*distinct values)".format(b.search(search_start, search_end)))
    for delete_val in [200, 18, 9]:
        print("----- DELETING {} -----".format(delete_val))
        b.delete(delete_val)
        print("----------- B+ TREE AFTER DELETING : {:3d} -----------".format(delete_val))
        b.pretty_print()


if __name__ == "__main__":
    from collections import OrderedDict
    choices = OrderedDict({
        1: "Insert",
        2: "Batch Insert",
        3: "Delete",
        4: "Search",
        5: "Search Range",
        6: "Terminate"
    })
    degree = input("Enter Degree of tree[4]: ")
    b = BPlusTree(degree=int(degree) if degree else 4)
    while True:
        print("\n".join("{} {}".format(key, val) for key, val in choices.items()))
        choice = int(input("Enter Choice: "))
        if choice == 1:
            val = int(input("Enter number to insert: "))
            b.insert(val)
            b.pretty_print()
        elif choice == 2:
            _values = map(int, input("Enter numbers (space separated): ").split())
            for val in _values:
                b.insert(val)
                b.pretty_print()
        elif choice == 3:
            val = int(input("Enter number to delete: "))
            b.delete(val)
            b.pretty_print()
        elif choice == 4:
            val = int(input("Enter number to search: "))
            b.search(val, val)
            b.pretty_print()
        elif choice == 5:
            start = int(input("Enter start number of range: "))
            end = int(input("Enter end number of range: "))
            b.search(start, end)
            b.pretty_print()
        else:
            print("Thanks for using the service!!")
            break
```

Results

```
Enter Degree of tree[4]: 4
1 Insert
2 Batch Insert
3 Delete
4 Search
5 Search Range
6 Terminate
Enter Choice: 1
Enter number to insert: 5
Leaf Node     : [5]     height >> 0
1 Insert
2 Batch Insert
3 Delete
4 Search
5 Search Range
6 Terminate
Enter Choice:
```

```
Leaf Node     : [5]     height >> 0
1 Insert
2 Batch Insert
3 Delete
4 Search
5 Search Range
6 Terminate
Enter Choice: 2
Enter numbers (space separated): 12 200 89 1
Leaf Node     : [5, 12]     height >> 0
Leaf Node     : [5, 12, 200]    height >> 0
Internal Node : [89]    height >> 0
Leaf Node     : [5, 12]     height >> 1
Leaf Node     : [89, 200]   height >> 1
Internal Node : [89]    height >> 0
Leaf Node     : [1, 5, 12]  height >> 1
Leaf Node     : [89, 200]   height >> 1
1 Insert
2 Batch Insert
3 Delete
4 Search
5 Search Range
6 Terminate
Enter Choice:
```

```
1 Insert

2 Batch Insert

3 Delete

4 Search

5 Search Range

6 Terminate

Enter Choice: 1

Enter number to insert: 450

---------- B+ TREE AFTER INSERT : 450 ----------

Internal Node : [50, 200, 320]  height >> 0

Leaf Node     : [10, 25, 30]    height >> 1

Leaf Node     : [50, 110]   height >> 1

Leaf Node     : [200, 300]  height >> 1

Leaf Node     : [320, 400, 450]     height >> 1

1 Insert

2 Batch Insert

3 Delete

4 Search

5 Search Range

6 Terminate
```

```
1 Insert

2 Batch Insert

3 Delete

4 Search

5 Search Range

6 Terminate

Enter Choice: 3

Enter number to delete: 320

---------- B+ TREE AFTER DELETING : 320 ----------

Internal Node : [50, 200, 320]  height >> 0

Leaf Node     : [10, 25, 30]    height >> 1

Leaf Node     : [50, 110]   height >> 1

Leaf Node     : [200, 300]  height >> 1

Leaf Node     : [400, 450]  height >> 1

1 Insert

2 Batch Insert

3 Delete

4 Search

5 Search Range

6 Terminate

Enter Choice: 5

Enter start number of range: 10

Enter end number of range: 50

----- Searching in batch for 10 to 50 -----

Result*: [10, 25, 30, 50]
```

Analysis

- all leaves at the same lowest level
- all nodes at least half full (except root) Let f be the degree of tree and n be the total number of data then

	Max # pointers	Max # keys	Min # pointers	Min # keys
Non-leaf	f	$f - 1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil - 1$
Root	f	$f - 1$	2	1
Leaf	f	$f - 1$	$\lfloor f/2 \rfloor$	$\lfloor f/2 \rfloor$

- Number of disk accesses proportional to the height of the B-tree.
- The ***worst-case height*** of a B+ tree is:

$$h \propto \log_f \frac{n+1}{2} \sim O(\log_f n)$$

	Time Complexity	Remarks
height	$O(\log_fn)$	
search	$O(f\log_fn)$	linear search inside each nodes
search	$O(\log_2f\log_fn)$	binary search inside each node
insert	$O(\log_fn)$	if splitting not require
insert	$O(f\log_fn)$	if splitting require
insert	$O(\log_fn)$	if merge not require
insert	$O(f\log_fn)$	if merge require