Practical 2
Implementation of B+ Tree

Gahan M. Saraiya, 18MCEC10

18mcec10@nirmauni.ac.in

I. Introduction

Aim of this practical is to implement algorithm of B+ tree.
Supported Operation

- Insert single item
- Insertion in bulk
- Deletion
- Search
- Range Search

II. Implementation

```
#!usr/bin/python3
# coding=utf-8
"""
By Gahan Saraiya
B+ Tree Implementation :
 ↪  https://github.com/gahan9/DS_lab/blob/master/btree_implementation/bPlusTree.py

some terminology of python used for variables in this code:
_var => (convention only) underscore prefix is just a hint to programmer that a
 ↪  variable or method starting with a single underscore is intended for internal
 ↪  use
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 the class is extended later.
__var__ => magic methods
// => Integer division (5//2 returns 2)
"""

__author__ = "Gahan Saraiya"

```

```
__link__ =
    "https://github.com/gahan9/DS_lab/blob/master/btree_implementation/bPlusTree.py"

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

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
```

```
63          self.keys = []  # store keys/data values
64          self.children = []  # store child nodes (list of instances of BtreeNode);
             ↪  empty list if node is leaf node
65          self.parent = None
66
67      def __repr__(self):
68          return " | ".join(map(str, self.keys))
69
70      @property
71      def is_leaf(self):
72          return False
73
74      @property
75      def total_keys(self):
76          return len(self.keys)
77
78      @property
79      def is_balanced(self):
80          # return False if total keys exceeds max accommodated keys (degree - 1)
81          return self.total_keys <= self.degree - 1
82
83      @property
84      def is_full(self):
85          return self.total_keys >= self.degree
86
87      @property
88      def is_empty(self):
89          return self.total_keys < (self.degree + 1) // 2
90
91
92  class LeafNode(object):
93      """
94      Class : B+ Tree Leaf Node
95      represents leaf node in B+ tree
96      """
97      def __init__(self, degree=4):
98          self.degree = degree
99          self.keys = []  # data values
100         self.sibling = None  # sibling node to point
101         self.parent = None  # parent node - None for root node
102
103     def __repr__(self):
104         return " | ".join(map(str, self.keys))
105
106     @property
107     def is_leaf(self):
108         return True
109
```

```
110     @property
111     def total_keys(self):
112         return len(self.keys)
113 
114     @property
115     def is_balanced(self):
116         # return False if total keys exceeds max accommodated data (degree - 1)
117         return self.total_keys < self.degree
118 
119     @property
120     def is_full(self):
121         return not self.is_balanced
122 
123     @property
124     def is_empty(self):
125         return self.total_keys < math.floor(self.degree / 2)
126 
127 
128 class BPlusTree(object):
129     def __init__(self, degree=4):
130         self.degree = degree
131         self.__root = LeafNode(degree=degree)
132         self.__leaf = self.__root
133 
134     def search_key(self, start_node, value):
135         """
136 
137         :param start_node: get root or any non leaf node
138         :param value: value to be search
139         :return: most matching leaf node
140         """
141         if start_node.is_leaf:
142             _index = bisect_left(start_node.keys, value)
143             return _index, start_node
144         else:
145             _index = bisect_right(start_node.keys, value)
146             return self.search_key(start_node.children[_index], value)
147 
148     def search(self, start=None, end=None):
149         """
150 
151         :param start: specify start node to search range for
152         :param end: specify end node for range search
153         :return:
154         """
155         _result = []
156         node = self.__root
157         leaf = self.__leaf
```

```
158
159         if start is None:
160             while True:
161                 for value in leaf.keys:
162                     if value <= end:
163                         _result.append(value)
164                     else:
165                         return _result
166                 if leaf.sibling is None:
167                     return _result
168                 else:
169                     leaf = leaf.sibling
170         elif end is None:
171             _index, leaf = self.search_key(node, start)
172             _result.extend(leaf.keys[_index:])  # equivalent to _result + leaf
173             while True:
174                 if leaf.sibling is None:
175                     return _result
176                 else:
177                     leaf = leaf.sibling
178                     _result.extend(leaf.keys)
179         else:
180             if start == end:
181                 _index, _node = self.search_key(node, start)
182                 try:
183                     if _node.keys[_index] == start:
184                         _result.append(_node.keys[_index])
185                         return _result
186                     else:
187                         return _result
188                 except IndexError:
189                     return _result
190             else:
191                 _index1, _node1 = self.search_key(node, start)
192                 _index2, _node2 = self.search_key(node, end)
193                 if _node1 is _node2:
194                     if _index1 == _index2:
195                         return _result
196                     else:
197                         _result.extend(_node1.keys[_index1:_index2])
198                         return _result
199                 else:
200                     _result.extend(_node1.keys[_index1:])
201                     node_ = _node1
202                     while True:
203                         if _node1.sibling == _node2:
204                             _result.extend(_node2.keys[:_index2 + 1])
205                             return _result
```

```
206                         else:
207                             try:
208                                 _result.extend(node_.sibling.keys)
209                                 node_ = node_.sibling
210                             except AttributeError:
211                                 return _result
212
213     def traverse(self, _node):
214         _result = []
215         _result.extend(_node.keys)
216         if getattr(_node, "sibling", None) is None:
217             return _result
218         for i in range(0, len(_node.sibling))[::-1]:
219             _result.extend(self.traverse(_node.sibling[i]))
220         while True:
221             pass
222
223     def pretty_print(self):
224         # print("B+ Tree:")
225         queue, height = deque(), 0
226         queue.append([self.__root, height])
227         while True:
228             try:
229                 node, height_ = queue.popleft()
230                 # print("adding node: {}".format(node))
231             except IndexError:
232                 return
233             else:
234                 if not node.is_leaf:
235                     print("Internal Node : {:} \theight >> {}".format(node.keys,
                            ↪ height_))
236                     if height_ == height:
237                         height += 1
238                     queue.extend([[i, height] for i in node.children])
239                 else:
240                     print("Leaf Node     : {} \theight >> {}".format([i for i in
                            ↪ node.keys], height_))
241
242     def insert(self, value):
243         # log("parent:{} leaf:{} node:{}\tkeys:{}\t
            ↪ children:{}".format(node.parent, node.is_leaf, node, node.keys,
            ↪ getattr(node, 'children', '0')))
244
245         def split_leaf_node(node):
246             log("splitting leaf node: {}".format(node.keys))
247             mid = self.degree // 2  # integer division in python3
248             new_leaf = LeafNode(self.degree)
249             new_leaf.keys = node.keys[mid:]
```

```
250             if node.parent is None:  # None and 0 are to be treated as different
                ↪ value
251                 parent_node = InternalNode(self.degree)  # create new parent for
                    ↪ node
252                 parent_node.keys, parent_node.children = [node.keys[mid]], [node,
                    ↪ new_leaf]
253                 node.parent = new_leaf.parent = parent_node
254                 self.__root = parent_node
255             else:
256                 _index = node.parent.children.index(node)
257                 node.parent.keys.insert(_index, node.keys[mid])
258                 node.parent.children.insert(_index + 1, new_leaf)
259                 new_leaf.parent = node.parent
260                 if not node.parent.is_balanced:
261                     split_internal_node(node.parent)
262             node.keys = node.keys[:mid]
263             node.sibling = new_leaf
264             log("{} --- {} --- {}".format(node, node.sibling,
                ↪ self.__root.children))
265
266         def split_internal_node(node_):
267             mid = self.degree // 2  # integer division in python3
268             new_node = InternalNode(self.degree)
269             new_node.keys = node_.keys[mid:]
270             new_node.children = node_.children[mid:]
271             new_node.parent = node_.parent
272             for child in new_node.children:
273                 child.parent = new_node  # assign parent to every new child of
                    ↪ current node
274             if node_.parent is None:  # again Note that None and 0 are not same
                ↪ but both treated as False in boolean
275                 # need to make new root if we are to split root node
276                 new_root = InternalNode(self.degree)
277                 new_root.keys = [node_.keys[mid - 1]]
278                 new_root.children = [node_, new_node]
279                 node_.parent = new_node.parent = new_root  # set parent of newly
                    ↪ created node
280                 self.__root = new_root  # set new ROOT node
281             else:
282                 # if node is not root internal node
283                 _index = node_.parent.children.index(node_)
284                 node_.parent.keys.insert(_index, node_.keys[mid - 1])
285                 node_.parent.children.insert(_index + 1, new_node)
286                 if not node_.parent.is_balanced:
287                     split_internal_node(node_.parent)
288             node_.keys = node_.keys[:mid - 1]
289             node_.children = node_.children[:mid]
290             return node_.parent
```

```
291
292         def insert_node(_node):
293             log("inserting : {} in node: {} having children: {}".format(value,
                ↪  _node.keys, getattr(_node, "children", "NULL")))
294             if _node.is_leaf:  # logic for leaf node
295                 log("node: {} is leaf".format(_node))
296                 _index = bisect_right(_node.keys, value)  # bisect and get index
                    ↪  value of where to insert value in node.keys
297                 _node.keys.insert(_index, value)
298                 if not _node.is_balanced:
299                     split_leaf_node(_node)
300                     log("----------- Tree status after split---------------")
301                     log(self.__root)
302                     log(self.__root.children)
303                     log(_node.parent.children)
304                     log(getattr(self.__root, "children", "NULL"))
305                 else:
306                     return
307             else:  # logic for internal node
308                 if not _node.is_balanced:
309                     self.insert(split_internal_node(_node))
310                 else:
311                     _index = bisect_right(_node.keys, value)
312                     log(_node.keys, _node.children, _index)
313                     insert_node(_node.children[_index])
314
315         insert_node(self.__root)
316
317     @staticmethod
318     def traverse_left_to_right(node, index):
319         if node.children[index].is_leaf:
320             node.children[index + 1].keys.insert(0,
                    ↪  node.children[index].keys[-1])
321             node.children[index].keys.pop()
322             node.keys[index] = node.children[index + 1].keys[0]
323         else:
324             node.children[index + 1].children.insert(0,
                    ↪  node.children[index].children[-1])
325             node.children[index].children[-1].parent = node.children[index + 1]
326             node.children[index + 1].keys.insert(0, node.keys[index])
327             node.children[index].children.pop()
328             node.children[index].keys.pop()
329
330     @staticmethod
331     def traverse_right_to_left(node, index):
332         if node.children[index].is_leaf:
333             node.children[index].keys.append(node.children[index + 1].keys[0])
```

```
334             node.children[index + 1].keys.remove(node.children[index +
                ↪ 1].keys[0])
335             node.keys[index] = node.children[index + 1].keys[0]
336         else:
337             node.children[index].children.append(node.children[index +
                ↪ 1].children[0])
338             node.children[index + 1].children[0].parent = node.children[index]
339             node.children[index].keys.append(node.keys[index])
340             node.keys[index] = node.children[index + 1].children[0]
341             node.children[index + 1].children.remove(node.children[index +
                ↪ 1].children[0])
342             node.children[index + 1].keys.remove(node.children[index +
                ↪ 1].keys[0])
343
344     def delete(self, delete_value):
345         def merge(node, index):
346             if node.children[index].is_leaf:
347                 node.children[index].keys = node.children[index].keys +
                    ↪ node.children[index + 1].keys
348                 node.children[index].sibling = node.children[index + 1].sibling
349             else:
350                 node.children[index].keys = node.children[index].keys +
                    ↪ [node.keys[index]] + node.children[
351                     index + 1].keys
352                 node.children[index].children = node.children[index].children +
                    ↪ node.children[index + 1].children
353             node.children.remove(node.children[index + 1])
354             node.keys.remove(node.children[index])
355             if node.keys:
356                 return node
357             else:
358                 node.children[0].parent = None
359                 self.__root = node.children[0]
360                 del node
361                 return self.__root
362
363         def delete_node(value, node):
364             log("deleting {} from node: {}".format(value, node))
365             if node.is_leaf:
366                 log("node is leaf")
367                 _index = bisect_left(node.keys, value)
368                 try:
369                     node_ = node.keys[_index]
370                 except IndexError:
371                     return False
372                 else:
373                     if node_ != value:
374                         return False
```

```
375                     else:
376                         node.keys.remove(value)
377                         return True
378             else:
379                 log("traversing internal node for deleting value")
380                 _index = bisect_right(node.keys, value)
381                 log("encountered index: {} having child values: {}".format(_index,
                     ↪ node.children[_index]))
382                 if _index <= len(node.keys):
383                     # print(node.children[_index].is_leaf)
384                     # print(node.children[_index],
                         ↪ node.children[_index].total_keys,
                         ↪ node.children[_index].degree / 2,
                         ↪ node.children[_index].is_empty)
385                     if not node.children[_index].is_empty:
386                         return delete_node(value, node.children[_index])
387                     elif not node.children[_index - 1].is_empty:
388                         self.traverse_left_to_right(node, _index - 1)
389                         return delete_node(value, node.children[_index])
390                     else:
391                         return delete_node(value, merge(node, _index))
392         delete_node(delete_value, self.__root)
393
394 def _test():
395     # test_lis = [0, 1, 11, 1, 2, 22, 13, 14, 4, 5, 23, 1, 51, 12, 31]
396     test_lis = [10, 1, 159, 200, 18, 90, 8, 17, 9]
397     # test_lis = range(10)
398     b = BPlusTree(degree=4)
399     for val in test_lis:
400         b.insert(val)
401         print("----------- B+ TREE AFTER INSERT : {:3d} -----------".format(val))
402         b.pretty_print()
403     # print("searching range.....")
404     # result = b.search_range(1, 12)
405     search_start, search_end = 1, 23
406     print("----- Searching in batch for {} to {} -----".format(search_start,
         ↪ search_end))
407     print("Result*: {} \n (*distinct values)".format(b.search(search_start,
         ↪ search_end)))
408     for delete_val in [200, 18, 9]:
409         print("----- DELETING {} -----".format(delete_val))
410         b.delete(delete_val)
411         print("----------- B+ TREE AFTER DELETING : {:3d}
             ↪ -----------".format(delete_val))
412         b.pretty_print()
413
414
415 if __name__ == "__main__":
```

```
416     from collections import OrderedDict
417     choices = OrderedDict({
418         1: "Insert",
419         2: "Batch Insert",
420         3: "Delete",
421         4: "Search",
422         5: "Search Range",
423         6: "Terminate"
424     })
425     degree = input("Enter Degree of tree[4]: ")
426     b = BPlusTree(degree=int(degree) if degree else 4)
427     while True:
428         print("\n".join("{} {}".format(key, val) for key, val in
            ↪  choices.items()))
429         choice = int(input("Enter Choice: "))
430         if choice == 1:
431             val = int(input("Enter number to insert: "))
432             b.insert(val)
433             print("----------- B+ TREE AFTER INSERT : {:3d}
                ↪  -----------".format(val))
434             b.pretty_print()
435         elif choice == 2:
436             _values = map(int, input("Enter numbers (space separated):
                ↪  ").split())
437             for val in _values:
438                 b.insert(val)
439                 print("----------- B+ TREE AFTER INSERT : {:3d}
                    ↪  -----------".format(val))
440                 b.pretty_print()
441         elif choice == 3:
442             val = int(input("Enter number to delete: "))
443             b.delete(val)
444             print("----------- B+ TREE AFTER DELETING : {:3d}
                ↪  -----------".format(val))
445             b.pretty_print()
446         elif choice == 4:
447             val = int(input("Enter number to search: "))
448             result = b.search(val, val)
449             print("Result*: {} \n (*distinct values)".format(b.search(result)))
450         elif choice == 5:
451             search_start = int(input("Enter start number of range: "))
452             search_end = int(input("Enter end number of range: "))
453             result = b.search(search_start, search_end)
454             print("----- Searching in batch for {} to {}
                ↪  -----".format(search_start, search_end))
455             print("Result*: {} \n (*distinct
                ↪  values)".format(b.search(search_start, search_end)))
456             # b.pretty_print()
```

```
457         else:
458             print("Thanks for using the service!!")
459             break
```

Output

```
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 -----
Result¹: [10, 25, 30, 50]
```

III. Summary

✓ all leaves at the same lowest level

✓ all nodes at least half full (except root)

✓ Supports Range Query

✗ sequential search overhead may rise if large number of record in result of range query

Let f be the degree of tree and n be the total number of data then

Space Complexity

	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. which is the ***worst-case height*** of a B+ tree is:

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

Let f be the degree of tree and n be the total number of data then

Time Complexity

	Time Complexity	**Remarks**
height	$O(\log_f n)$	
Root	$O(f \log_f n)$	linear search inside each nodes
search	$O(\log_2 f \log_f n)$	binary search inside each node
insert	$O(\log_f n)$	if splitting not require
insert	$O(f \log_f n)$	if splitting require
insert	$O(\log_f n)$	if merge not require
insert	$O(f \log_f n)$	if merge require