Name: Kartik Jolapara **SAPID:** 60004200107

DIV: B/B1

ADBMS

Exp 2

Kartik Jolapara
60004200107
8/10/22 B1 ADBMS

Page No. 1
Date 8/10/22

Exp 2

Aim: Perform operation like searching, insertion, deletion on B-Tree and B+Tree

Theory:

① B-Tree

B-Tree is a self-balancing tree. B-Trees are useful when we are dealing with huge amounts of data that can't be fitted in main memory. When the number of keys is high, the data is read from the disk in forms of blocks. Disk access time is very high compared to the main memory access time.

The main idea of using B-Trees is to reduce the number of disk access. Most of the tree operations require (O(h)) disk access where h is the height of the tree. B-Tree is a fat tree meaning, the height of B-Tree is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to disk block size. Since the height of the B-Tree is kept low so total disk access for most of the operations are reduced significantly compared to balanced Binary search Tree like AVL Trees.

② Properties of B-Tree

→ All leaves are at the same level

→ B-Tree is defined by term minimum degree 't'. The value of 't' depends upon disk block size.

→ Every node except the root must contain atleast (t-1) keys. The root may contain minimum of 1 key.

→ All nodes may contain almost (2*t - 1) keys.
→ No. of children of a node is equal to no. of keys in it plus 1
→ All keys of a node are sorted in increasing order.
→ It grows & shrinks from root.
→ Insertion happens only at leaf node.

Time complexity of B-Tree:
search - O(logn)
Insert - O(logn)
Delete - O(logn)

⊛ B+ Tree

B+ Tree is an extension of B-Tree which allows efficient insertion deletion and search operations. In B-tree keys and records both can be stored in internal as well as leaf nodes. whereas, in B+ tree records can only be stored on leaf nodes and internal nodes can only store key values.

The leaf node of a B+ tree are linked together in the form of singly linked list to make search queries more efficient. B+ trees are used to store the large amount of data which can't be stored in main memory. Due to the fact that, size of main memory is always limited, the internal nodes of B+ tree are stored in main memory whereas, leaf nodes are stored in secondary memory.

⊙ properties of B+ tree

→ All leafs are at some level.
→ The root has atleast two children
→ Each node except root can have a maximum of m children and atleast m/2 children
→ Each node can contain a maximum of (m-1) keys and a maximum of ⌈m/2⌉ − 1 keys
→ keys are used for indexing
→ Data can be stored sequentially or directly

⊙ Time complexity

search − O(logn)
Insert − O(logn)
Delete − O(logn)

⊛ B-Tree v/s B+ Tree

B-Tree	B+ Tree
① Search keys cannot be repeatedly stored	① Redundant search keys can be present.
② Data can stored in the leaf node as well as internal nodes	② Data can only be stored on the leaf nodes
③ searching for some data is slower process some data can be found on internal nodes as well as leaf nodes	③ search is comparitively faster as data can only be found on the leaf nodes.
④ Deletion of internal nodes are complicated & time consuming	④ Deletion will never be a complexed process since some elements will be deleted from leaf

B Tree	B+ Tree
⑤ Leaf nodes can't be linked together	⑤ Leaf nodes are linked together to make search operations more efficient.

Conclusion: Thus, we successfully studied & compared various operations in B tree & B+ tree.

B Tree Code:

```
class BTreeNode:
  def __init__(self, leaf=False):
    self.leaf = leaf
self.keys = []     self.child
= []

class BTree:
  def __init__(self, t):
    self.root = BTreeNode(True)
    self.t = t

  def insert(self, k):     root = self.root
if len(root.keys) == (2 * self.t) - 1:
      temp = BTreeNode()
self.root = temp
temp.child.insert(0, root)
self.split_child(temp, 0)
self.insert_non_full(temp, k)
else:
      self.insert_non_full(root, k)
```

```
  def insert_non_full(self, x, k):
    i = len(x.keys) - 1
if x.leaf:
      x.keys.append((None, None))
while i >= 0 and k[0] < x.keys[i][0]:
x.keys[i + 1] = x.keys[i]         i -= 1
      x.keys[i + 1] = k
else:
      while i >= 0 and k[0] < x.keys[i][0]:
i -= 1       i += 1
      if len(x.child[i].keys) == (2 * self.t) - 1:
        self.split_child(x, i)
if k[0] > x.keys[i][0]:
          i += 1
self.insert_non_full(x.child[i], k)


  def split_child(self, x, i):
    t = self.t     y =
x.child[i]     z =
BTreeNode(y.leaf)
x.child.insert(i + 1, z)
    x.keys.insert(i, y.keys[t - 1])
    z.keys = y.keys[t: (2 * t) - 1]
    y.keys = y.keys[0: t - 1]
if not y.leaf:
      z.child = y.child[t: 2 * t]
```

```
      y.child = y.child[0: t - 1]

  def print_tree(self, x, l=0):
    print("Level ", l, " ", len(x.keys), end=":")
for i in x.keys:       print(i, end=" ")     print()
l += 1     if len(x.child) > 0:       for i in
x.child:         self.print_tree(i, l)

  def search_key(self, k, x=None):
if x is not None:
      i = 0       while i < len(x.keys) and k >
x.keys[i][0]:         i += 1       if i < len(x.keys)
and k == x.keys[i][0]:
        return (x, i)
elif x.leaf:
return None
else:
        return self.search_key(k, x.child[i])

else:
      return self.search_key(k, self.root)



def main():   B
= BTree(3)
```

```
  for i in range(10):
B.insert((i, 2 * i))


  B.print_tree(B.root)


  if B.search_key(8) is not None:
    print("\nFound")
  else:
    print("\nNot Found")



if __name__ == '__main__':
  main()
```

Output:

```
Level  1   2:(3, 6) (4, 8)
Level  1   4:(6, 12) (7, 14) (8, 16) (9, 18)

Found
```

B+ Tree

Code:

```
import math


class Node:     def
__init__(self, order):
```

```
        self.order = order
self.values = []        self.keys
= []        self.nextKey =
None        self.parent =
None        self.check_leaf =
False


    def insert_at_leaf(self, leaf, value, key):
        if (self.values):
            temp1 = self.values
for i in range(len(temp1)):
if (value == temp1[i]):
                    self.keys[i].append(key)
break                elif (value <
temp1[i]):
                    self.values = self.values[:i] + [value] + self.values[i:]
self.keys = self.keys[:i] + [[key]] + self.keys[i:]
break                elif (i + 1 == len(temp1)):
self.values.append(value)                   self.keys.append([key])
break        else:
            self.values = [value]
self.keys = [[key]]



class BplusTree:
```

```
    def __init__(self, order):
self.root = Node(order)
self.root.check_leaf = True

    def insert(self, value, key):
        value = str(value)
old_node = self.search(value)
old_node.insert_at_leaf(old_node,
value, key)


        if (len(old_node.values) == old_node.order):
            node1 = Node(old_node.order)
node1.check_leaf = True             node1.parent =
old_node.parent             mid = int(math.ceil(old_node.order /
2)) - 1             node1.values = old_node.values[mid + 1:]
node1.keys = old_node.keys[mid + 1:]             node1.nextKey
= old_node.nextKey             old_node.values =
old_node.values[:mid + 1]             old_node.keys =
old_node.keys[:mid + 1]             old_node.nextKey = node1
self.insert_in_parent(old_node, node1.values[0], node1)

    def search(self, value):         current_node =
self.root         while(current_node.check_leaf
== False):
            temp2 = current_node.values
for i in range(len(temp2)):
if (value == temp2[i]):
```

```
                    current_node = current_node.keys[i + 1]
break                   elif (value < temp2[i]):
                    current_node = current_node.keys[i]
break                   elif (i + 1 ==
len(current_node.values)):
current_node = current_node.keys[i + 1]
break
        return current_node

    def find(self, value, key):         l =
self.search(value)         for i, item in
enumerate(l.values):             if item ==
value:                  if key in l.keys[i]:
return True                  else:
return False         return False

    def insert_in_parent(self, n, value, ndash):
if (self.root == n):
            rootNode = Node(n.order)
rootNode.values = [value]
rootNode.keys = [n, ndash]
self.root = rootNode             n.parent
= rootNode             ndash.parent =
rootNode             return

        parentNode = n.parent
temp3 = parentNode.keys         for
```

```
i in range(len(temp3)):             if
(temp3[i] == n):
                parentNode.values = parentNode.values[:i] + \
                    [value] + parentNode.values[i:]
parentNode.keys = parentNode.keys[:i +
                                                  1] + [ndash] + parentNode.keys[i + 1:]
if (len(parentNode.keys) > parentNode.order):
                    parentdash = Node(parentNode.order)
parentdash.parent = parentNode.parent                    mid =
int(math.ceil(parentNode.order / 2)) - 1
parentdash.values = parentNode.values[mid + 1:]
parentdash.keys = parentNode.keys[mid + 1:]
value_ = parentNode.values[mid]                    if (mid == 0):
                        parentNode.values = parentNode.values[:mid + 1]
else:
                        parentNode.values = parentNode.values[:mid]
parentNode.keys = parentNode.keys[:mid + 1]                    for j
in parentNode.keys:                        j.parent = parentNode
for j in parentdash.keys:                        j.parent = parentdash
self.insert_in_parent(parentNode, value_, parentdash)


    def delete(self, value, key):
node_ = self.search(value)


        temp = 0
        for i, item in enumerate(node_.values):
if item == value:
```

```
        temp = 1

        if key in node_.keys[i]:
if len(node_.keys[i]) > 1:

node_.keys[i].pop(node_.keys[i].index(key))
elif node_ == self.root:                node_.values.pop(i)
node_.keys.pop(i)              else:
            node_.keys[i].pop(node_.keys[i].index(key))
del node_.keys[i]
node_.values.pop(node_.values.index(value))
self.deleteEntry(node_, value, key)          else:
          print("Value not in Key")
return      if temp == 0:
      print("Value not in Tree")
return

  def deleteEntry(self, node_, value, key):

    if not node_.check_leaf:
      for i, item in enumerate(node_.keys):
if item == key:
          node_.keys.pop(i)
break        for i, item in
enumerate(node_.values):          if item
== value:
```

```
                    node_.values.pop(i)
break

        if self.root == node_ and len(node_.keys) == 1:
            self.root = node_.keys[0]
node_.keys[0].parent = None
del node_             return
        elif (len(node_.keys) < int(math.ceil(node_.order / 2)) and node_.check_leaf
== False) or (len(node_.values) < int(math.ceil((node_.order - 1) / 2)) and
node_.check_leaf == True):

            is_predecessor = 0
parentNode = node_.parent
            PrevNode = -1
            NextNode = -1
            PrevK = -1             PostK = -1             for
i, item in enumerate(parentNode.keys):

                if item == node_:
if i > 0:
                        PrevNode = parentNode.keys[i - 1]
                        PrevK = parentNode.values[i - 1]

                    if i < len(parentNode.keys) - 1:
NextNode = parentNode.keys[i + 1]
                        PostK = parentNode.values[i]
```

```
            if PrevNode == -1:                  ndash = NextNode
value_ = PostK              elif NextNode == -1:                  is_predecessor
= 1                   ndash = PrevNode                  value_ = PrevK
else:                   if len(node_.values) + len(NextNode.values) <
node_.order:
                    ndash = NextNode
value_ = PostK                  else:
                    is_predecessor = 1
ndash = PrevNode
value_ = PrevK

            if len(node_.values) + len(ndash.values) < node_.order:
if is_predecessor == 0:                     node_, ndash = ndash,
node_                  ndash.keys += node_.keys                  if not
node_.check_leaf:
                    ndash.values.append(value_)
                else:
                    ndash.nextKey = node_.nextKey
ndash.values += node_.values

                if not ndash.check_leaf:
for j in ndash.keys:
j.parent = ndash
                self.deleteEntry(node_.parent, value_, node_)
del node_             else:                 if is_predecessor ==
1:                     if not node_.check_leaf:
```

ndashpm = ndash.keys.pop(-1)
ndashkm_1 = ndash.values.pop(-1)
node_.keys = [ndashpm] + node_.keys
node_.values = [value_] + node_.values
parentNode = node_.parent for i, item in
enumerate(parentNode.values):
if item == value_:
p.values[i] = ndashkm_1
break else:
ndashpm = ndash.keys.pop(-1)
ndashkm = ndash.values.pop(-1)
node_.keys = [ndashpm] + node_.keys
node_.values = [ndashkm] + node_.values
parentNode = node_.parent for i, item in
enumerate(p.values): if item == value_:
parentNode.values[i] = ndashkm
break else: if not
node_.check_leaf:
ndashp0 = ndash.keys.pop(0)
ndashk0 = ndash.values.pop(0) node_.keys
= node_.keys + [ndashp0] node_.values =
node_.values + [value_] parentNode =
node_.parent for i, item in
enumerate(parentNode.values): if item ==
value_:
parentNode.values[i] = ndashk0
break else:

ndashp0 = ndash.keys.pop(0)

ndashk0 = ndash.values.pop(0) node_.keys

= node_.keys + [ndashp0] node_.values =

node_.values + [ndashk0] parentNode =

node_.parent for i, item in

enumerate(parentNode.values): if item ==

value_:

parentNode.values[i] = ndash.values[0]

break

if not ndash.check_leaf:

for j in ndash.keys:

j.parent = ndash if not

node_.check_leaf: for j in

node_.keys: j.parent =

node_ if not

parentNode.check_leaf:

for j in parentNode.keys:

j.parent = parentNode

def printTree(tree):

lst = [tree.root]

level = [0] leaf

= None flag = 0

lev_leaf = 0

```
    node1 = Node(str(level[0]) + str(tree.root.values))

    while (len(lst) != 0):         x =
lst.pop(0)         lev = level.pop(0)
if (x.check_leaf == False):             for i,
item in enumerate(x.keys):
                print(item.values)
else:
            for i, item in enumerate(x.keys):
                print(item.values)
if (flag == 0):
lev_leaf = lev
leaf = x                 flag = 1



record_len = 3 bplustree =
BplusTree(record_len)
bplustree.insert('5', '33')
bplustree.insert('15', '21')
bplustree.insert('25', '31')
bplustree.insert('35', '41')
bplustree.insert('45', '10')

printTree(bplustree)

if(bplustree.find('45', '10')):
    print("Found") else:
```

print("Not found") **Output:**

```
['15', '25']
['35', '45']
['5']
Found
```