CSCE 608 Database Systems Project 2
Part 1: B Plus Tree

Texas A&M University
Luming Xu
UIN: 930001415

December 2020

Contents

1 Introduction

This is the first part of project 2, implementing a B Plus Tree that supports insert, delete, search, and range search. For execution guide and to verify that the implementation is correct, see README under the project folder.

GitHub Repository: `https://github.com/Abalagu/BPlusTree`

2 Data Generation

Although project states that a leaf node requires no attribute other than a search key, an additional attribute 'payload' as the string of the corresponding key is assigned to make search results more readable. To generate unique keys within range, `numpy.random.choice(range, size, replace=False)` is called before constructing the tree, which is equivalent to random permutation of all the elements within range, and take the first n size of elements.

3 Building B+ Tree

When building a B+ Tree, the program follows the constraint listed within the lecture nodes.

	Max. # pointers	Max. # keys	Min. # pointers	Min. # keys
Non-leaf	n+1	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil - 1$
Leaf	n+1	n	$\lfloor (n+1)/2 \rfloor + 1$	$\lfloor (n+1)/2 \rfloor$
Root	n+1	n	2	1

Suppose a list of values are given, there are two categories to build a B+ Tree, either dense or sparse. A dense tree is that each node is as full as possible, aiming at the upper bound of the constraint. A sparse tree is that each node is as empty as possible, aiming at the lower bound of the constraint.

For example, a tree of order 3 have constraints as follows:

1. NON LEAF, pointers [2-4], keys [1-3]',
2. LEAF, pointers [3-4], keys [2-3]',
3. ROOT, pointers [2-4], keys [1-3]'

The detailed procedure is as follows:

Firstly, build leaf nodes on tree height of 0. The idea is to divide the given keys to groups. Notice that a greedy method does not work here, because taking max keys in the last but one node may leave insufficient keys for the last node, which violates the above mentioned constraint. Suppose a list of 7 keys are given. A greedy method would results in three nodes, each having 3, 3, 1 keys. Notice that the last node has only 1 key, which is below the leaf minimum of 2 keys. Therefore, the correct way of key distribution is 3, 2, 2. If to build a sparse tree, a greedy methods would results in 2, 2, 2, 1, while the correct way of key distribution is 2, 2, 3.

To handle this slicing behavior, a function is implemented to output the above distribution given the number of keys. For dense tree, if the remaining keys are less than maximum + minimum, then let the last node take minimum

number of keys, and the last but one node takes the rest. For sparse tree, if the remaining keys are less than twice the minimum, then let this node take all the keys, since it cannot be further divided.

The procedure of building leaf layer can be summarized as:

1. create a leaf node with keys in range of the corresponding slicing, and assign payload for each key as `str(key)`, as is discussed in the data generation section.
2. connects each leaf node with its next node.
3. assign order to the node, and each node type as leaf.

Secondly, recursively build internal nodes on tree height of 1 and higher from lower level nodes. The recursion stops when it results in only one node, which would be the root of the tree. The difference between building a leaf node and an internal node is that in this case, grouping of lower layer nodes is based on the number of pointers, and the number of keys is obtained afterwards.

Following the example of order 3 B+ tree, now assume that there are 7 leaf nodes. To build a dense tree, divide them into groups of 3, 2, 2, each having 2, 1, 1 keys. To build a sparse tree, divide them into groups of 2, 2, 3, each having 1, 1, 2 keys. These are the nodes at height 1.

Here the output is three nodes, and another layer is required. For both sparse and dense tree, a single root with 3 pointers and 2 keys is sufficient, which resides in height 2 of the tree. This concludes the tree generation procedure.

4 Tree Operations

It is known that deletion may results in node underflow, and insertion may results in node overflow. If an insert or delete operation results in a node violating the constraint, leave it there and let its parent node fix the problem on recursion back to root. When inserting into a dense tree, it may cause a chaining effect of several node overflow, since inserting the newly split node to its parent may result in the parent node violating the constraint as well. This is the same for deleting a key in a sparse tree, resulting in possible chaining effect of node merge.

To address this complex behavior, the tree operations are implemented in a recursive manner. Notice that [2] states that the deletion rules are of the following priority:

1. Borrow from left
2. Merge with left
3. Borrow from right
4. Merge with right

However, in this implementation, borrow from either left or right is checked first, then consider merging with left or right, which is slightly different. Following the rule may maintain that a tree be left biased after several operations.

4.1 Insertion

When inserting a key into the tree makes the leaf node **L** overflow, it finishes execution at the leaf level and recurs to its parent node **P**. The parent node detects that the child node storing the new key overflows, it then splits this child node into two, **L, L'**. As a result, the original child holds the first half of its previous keys and payload, and the new child holds the second half. The new child is then inserted into the parent **P**, along with a new key as the minimal key value within this new child. This insertion may also cause overflow. In a similar way, this internal node recurs up and let its parent **Q** fix the problem by splitting node **P**.

When such chaining effect ripples up to the top, making the root node overflow, the root **R** then needs to split to **R, R'**. The tree then creates a new root **T** with key as the minimum key from the tree under the node **R'**, and two pointers **R, R'** holding the original information.

4.1.1 Split Behavior Difference

Notice that the splitting behavior is different between an internal node and a leaf node. When splitting a leaf node, copy the middle key value up to its parent node; when splitting an internal node, move the middle key value up to its parent node. [4]This video starting at 3:51 explains such a difference as: "leaf stores information associated with the respective keys; when storing an internal node, it associates no actual information and does not store redundant information." However, below discusses the fact that the split behavior of internal node is mandatory, not optional, because there is no way that the resulting two internal nodes after split conforms to the constraint while keeping all the keys.

Suppose the i-th leaf node **I** of an internal node **P** overflows after an insertion, and splits to node **I** and **I'**.

The middle key value of a leaf represents an actual data, and thus should be kept. Although there are other valid key values to be inserted into the parent keys that do not violate B+Tree rules, such as any value in interval $(max(\mathbf{I}.keys), min(\mathbf{I'}.keys)]$, copying $\mathbf{I'}.keys[0]$, which is the smallest key in **I'** node, would be the easiest to do.

However, when splitting an internal node, it is forced to leave the middle key value out for insertion to the parent node. Suppose a tree is of order 3, internal nodes pointer size should be between [2,4], key size should be between [1,3]. Also suppose an internal node **N** at height 1 is already full with 4 pointers and 3 keys, and it is a child node of its parent node **P** at height 2. Suppose an insertion to one of **N**'s child triggers its split, and node **N** receives a new child, which as a chaining effect makes it split as well. Notice that now the internal node **N** has 5 pointers and 4 keys. To properly split the internal node

to two groups, the 5 child pointers need to be distributed, see discussion on node distribution of either dense or sparse. The split of internal node **N** should result in **N** with 3 pointers and 2 keys, and **N'** with 2 pointers and 1 key. Notice that the total number of keys from **N** and **N'** is 3, which is 1 key short of the 4 keys when **N** is in overflow state before splitting. Node **P**, the parent of node **N**, now needs a key that divides **N** and **N'**. However, the missing is not of proper value, and the new key should be the smallest **leaf key** from the tree rooted under **N'**.

4.1.2 Alternative Approach

Note that redistribution can solve overflow problem as well, by moving a minimal key to the left sibling, or a maximal key to the right sibling, given that one of them have empty space. This may results in less splitting of the tree. However, redistribution may not work in scenario where its left and/or right siblings are full. If a node has no left nor right sibling, then it is the root node, and the root should split and connects to a new root when full. At this moment, only split function is implemented.

4.1.3 Bias

Left biasing: when the node overflows after an insertion, it splits to two nodes, such that left has more keys than the right. Right biasing: when splitting, right has more keys than the left. When making a sparse tree, the left is as sparse as possible, which is right biased; when making a dense tree, the left is as full as possible, which is left biased.

4.2 Deletion

When deleting a key in a node, the number of keys within the node may drop below the minimum constraint. Similar to insertion, such error caused by the operation leaves for the parent node to fix. There are two ways of fixing node underflow implemented in this program, redistribution and merge.

1. If either its left or right sibling has more than minimum of keys, then redistribute one to the node to satisfy the constraint.

2. If either its left or right sibling is not full, try merge with the sibling.

3. If it has no left or right sibling, then it can only be the root node. In this case, the root node has only one child, and that child should be elevated to be the new root. After becoming the root, the key and pointer constraints are lifted, since it only requires 1 key and 2 pointers.

A node must fall into one of the three cases discussed above, thus it solves any problem caused by deletion.

4.2.1 Redistribution

Redistribution fixes the problem by borrowing a key from the left or right sibling of a node, if either of them has more than minimum amount of keys. There is a slight difference between redistribution among leaf nodes and internal nodes. For leaf nodes, when borrowing from the left sibling, take the largest key and the corresponding payload and put to the front; when borrowing from the right sibling, take the smallest key and the corresponding payload and put to the end of the list. Also when borrowing from right sibling, one should update the corresponding key from its parent, since the smallest key serves as a search division between nodes, and it has changed due to redistribution.

4.2.2 Merge

Merge also fixes the problem, and it is put to lower priority than redistribution. As a result, a node only merges with its sibling when the sibling hold its minimal amount of keys, because otherwise a key will be borrowed. Merge may cause overflow in the resulting node, but it leaves to its parent to fix when it recurs upward.

5 Experiments

The experiments print out the state of the node before and after any change. For sparse trees, insertion would normally only change the leaf node, while deletion would trigger a series of merge/redistribution from leaf upward. In contrast, for dense trees, deletion would normally only affect the leaf node, while insertion would trigger a series of node splitting from leaf upward.

5.1 Validity Check

A validity check function is implemented that checks the following from a node.

1. The number of keys should be in range of the constraint.
2. If it is a leaf node, the number of payload should be in range, and is of the same size as the number of keys.
3. If it is an internal node, the number of child should be in range, and is one more than the size of the keys.
4. The keys within a node should be sorted.
5. All its child should be at the same height.
6. Each key at position i should be larger than the max leaf key under the left sub-tree(tree rooted under the i-th child), and no larger than the min leaf key under the right sub-tree(tree rooted under i+1-th child).
7. Check that all the child till their leaf nodes satisfy the above condition.

8. If calling on a root node, check the following:
    - Traversing both top-down and by sequence pointer results in the same list of leaf nodes.
    - Traversing both top-down and by sequence pointer results in the same list of leaf keys.
    - Searching each leaf key retrieves the correct result

5.2 Log

Here attaches a sample of the complete log output.

```
STARTING TEST ON TREE:
order: 13, option: dense, 10000 keys, 3 height

INSERTING KEY: 117239
BEFORE INSERTION:
NodeType.LEAF in memory b1f08, 13 keys, 0 pointers, 13 payload, at height
     0, next->af388
keys=[117145, 117148, 117172, 117188, 117208, 117210, 117225, 117242,
    117262, 117269, 117270, 117274, 117281]

AFTER INSERTION:
NodeType.LEAF in memory b1f08, 14 keys, 0 pointers, 14 payload, at height
     0, next->af388
keys=[117145, 117148, 117172, 117188, 117208, 117210, 117225, 117239,
    117242, 117262, 117269, 117270, 117274, 117281]

INSERTION OVERFLOW
NODE BEFORE SPLIT:
NodeType.LEAF in memory b1f08, 14 keys, 0 pointers, 14 payload, at height
     0, next->af388
keys=[117145, 117148, 117172, 117188, 117208, 117210, 117225, 117239,
    117242, 117262, 117269, 117270, 117274, 117281]

NODE AFTER SPLIT:
NodeType.LEAF in memory b1f08, 7 keys, 0 pointers, 7 payload, at height
    0, next->4da08
keys=[117145, 117148, 117172, 117188, 117208, 117210, 117225]

NEW NODE:
NodeType.LEAF in memory 4da08, 7 keys, 0 pointers, 7 payload, at height
    0, next->af388
keys=[117239, 117242, 117262, 117269, 117270, 117274, 117281]

NODE BEFORE INSERT:
NodeType.NON_LEAF in memory aacc8, 13 keys, 14 pointers, 0 payload, at
    height 1, next->None
keys=[116466, 116570, 116652, 116746, 116844, 116937, 117038, 117145,
    117283, 117415, 117581, 117745, 117882]
```

```
NODE AFTER INSERT:
NodeType.NON_LEAF in memory aacc8, 14 keys, 15 pointers, 0 payload, at
    height 1, next->None
keys=[116466, 116570, 116652, 116746, 116844, 116937, 117038, 117145,
    117239, 117283, 117415, 117581, 117745, 117882]
```

6 Helpful Resources

During the writing of B+ Tree functionalities, there are several references that I have come across.

- [1]B+ Tree online visualization tool , which animates the merge and split process. However, the degree definition and the split/merge behavior is different from what is described in the lecture.
- [2]A video that describes B+ Tree borrow and merge behavior. Notice that this implementation is slightly different from what is described in the video, especially for borrowing a key from a sibling of a node. A sibling is generally defined as an adjacent node that is rooted under the same direct parent node. However, the video author describes that a node may borrow from a node that is adjacent to it, even if it is rooted differently.
- [3]A video that describes split behavior, and introduced the concept of left-biased and right-biased tree.

7 Other Thoughts

During the writing of this implementation, the most crucial function that I've written is the one that verifies if a node, as well as all the nodes rooted under it, is valid, by calling `node.is_valid()`. Without the extensive use of this function, it would be impossible to detect problems early, and I would obtain no confidence in my code. However, it becomes less effective at the stage of implementing merge, split, and redistribution, due to the recursive nature of these functions. One cannot poke into the ongoing chaining effect unless using single step debugging. At this stage, it depends more on the fact that one should be clear about the control flow of the functionality, and test driven development would be very inefficient.

While professor Jianer Chen discourages the use of Python as the project language, I should comment that this implementation uses no other external library than random number generator from numpy. I chose Python merely for my proficiency in the it, and I really hope that the effort put into this project is appreciated.

References

[1] B+ Tree Operation Online Visualization https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

[2] B+Tree Deletions by Stephan Burroughs https://www.youtube.com/watch?v=QrbaQDSuxIM

[3] 5.29 B+ tree insertion — B+ tree creation example — Data structure https://www.youtube.com/watch?v=DqcZLulVJ0M&t=606s

[4] B+ Trees Basics 2 (insertion) by Douglas Fisher https://youtu.be/_nY8yR6iqx4?t=231