

# Data Engineering

Harsh Tomar B21AI049

## Implementing a Generic B+ Tree

### Attachments

- `BPlusTree` Implementation Code File
  - The code files are divided into different header and c files to provide better implementation and readability of Code.
- Test Suite with Meaningful Test Cases
- We can interact with our code and make a B+ Tree by running the `main_bpt.c` file.
- Result of Test Suite can be found by running the `test_suite.c` file.

## What are B+ Trees?

### B-Tree

A B-Tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in **logarithmic time**. The B-Tree generalizes the Binary Search Tree, allowing for **nodes** with more than two children. Unlike other Self-Balancing Binary Search Trees, the B-Tree is well suited for storage systems that read and write relatively large blocks of data, such as Databases and File Systems.

### B+-Tree

A **B+ tree** is an **m-ary tree** with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves.<sup>1</sup>([https://en.wikipedia.org/wiki/B%2B\\_tree#cite\\_note-Navathe-1](https://en.wikipedia.org/wiki/B%2B_tree#cite_note-Navathe-1)) The root may be either a leaf or a node with two or more children.

A B+ tree can be viewed as a **B-tree** in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves.

(The definition is taken from Wikipedia)

## Implementing Generic B+ Tree

# Design Decisions

- Insert < key > < value > pairs in the Tree
- Duplicate Keys are handled by Maintaining a linked-list of duplicate values. For example, if we add the key 9 two times, we get 9 -> 9

## Challenges

- Handling Insertion and Deletion Algorithms
- Inserting Duplicate Keys
- Deleting Duplicate Keys
- Printing the Tree
- Searching for Duplicate Keys

## Code

### Class / Structure of a Node

#### Non Leaf Nodes

Minimum and Maximum number of entries  $n$  is bounded by the order  $d$  of the B+-Tree:

$$d \leq n \leq 2 * d$$

for root node

$$rootnode : (1 \leq n \leq 2 * d)$$

- A node contains  $n+1$  pointers.
- Pointers  $p_i$  points to a subtree in which all the key values  $k$  are such that  $k_i \leq k < k_{i+1}$
- Ex ( $p_0$  points to a subtree with key values  $< k_1$ ,  $p_n$  points to a subtree with key values  $\geq k_n$ )

#### Leaf Nodes

B+-Tree Leaf nodes contain pointers to data records. A leaf node entry with key value  $k$  is denoted as  $k^*$  as before.

### Question 1

Insert the keys based on the sequences provided below in the B+ - Tree and show the tree after each insertion.

Insertion sequence for my roll number (odd) (B21AI049 => 49):

2, 8, 24, 12, 49, 54, 42, 20, 8, 4

### Terminal Output

```
./"main_bpt"
hush@LAPTOP-G6QAR288 > > > > B_Plus_Tree_Implementation_Assignment
10 master ./"main_bpt" 16:43:28
B+ Tree of Order 4.
Enter any of the following commands after the prompt > :
    i <k> -- Insert <k> (an integer) as both key and value).
    i <k> <v> -- Insert the value <v> (an integer) as the value of key <k>
(an integer).
    f <k> -- Find the value under key <k>.
    p <k> -- Print the path from the root to key k and its associated value
.
    d <k> -- Delete key <k> and its associated value.
    x -- Destroy the whole tree. Start again with an empty tree of the sam
e order.
    t -- Print the B+ tree.
    l -- Print the keys of the leaves (bottom row of the tree).
    s -- Print the Tree in the format [leftchildvalues], root, [right_child
_values]...    q -- Quit. (Or use Ctl-D or Ctl-C.)
> 
```

Now, we insert the required sequence of keys

```
> i 2
2 |
> i 8
2 8 |
> i 24
2 8 24 |
> i 12
12 |
2 8 | 12 24 |
> i 49
12 |
2 8 | 12 24 49 |
> i 54
12 49 |
2 8 | 12 24 | 49 54 |
> i 42
12 49 |
2 8 | 12 24 42 | 49 54 |
> i 20
12 24 49 |
2 8 | 12 20 | 24 42 | 49 54 |
> i 8
12 24 49 |
2 8 | 12 20 | 24 42 | 49 54 |
> i 4
12 24 49 |
2 4 8 | 12 20 | 24 42 | 49 54 |
> 
```

Final Tree Structure

```
12 24 49 |
2 4 8 | 12 20 | 24 42 | 49 54 |
```

Final Leaf Level Structure

```
> l
2 (2)4 (4)8 (8->8) | 12 (12)20 (20) | 24 (24)42 (42) | 49 (49)54 (54)
```

Output of `toString()` :

```
2 4 8 | 12 20 | 24 42 | 49 54 |
> s
[2, 4, 8] 12 , [12, 20] 24 , [24, 42] 49 , [49, 54]
```

We can confirm that the tree-structure is printed in `[left_child_values], root, [right_child_values], ...` fashion.

## Question 2

Create a `toString()` function to return B+-Tree in string format such as "[left child values], root, [right child values], [left child values], root, [right child values].....".

The code for the `toString()` function can be found in `to_string.c` file.

Output of `to_string` function is also used in verifying the `test_cases` . Further, we can also use output of Search functions to verify different test cases.

---

## Question 3

Print the results of search and deletion of the last two digits of your roll number (B21AI049 => 49) in the above tree and report the final B+ - Tree.

```
> t
12 24 49 |
2 4 8 | 12 20 | 24 42 | 49 54 |
> f 49
Record at 0x55d757e41cc0 -- key 49, value 49.
> p 49
[12 24 49] 3 ->
Leaf [49 54] ->
Record at 0x55d757e41cc0 -- key 49, value 49.
>
```

## Searching for 49

In the above tree, when we search for the key 49 by using the command `f 49` we get that the key and it's value is found in the tree.

`p 49` prints the path from the root to the key 49, as we can confirm that from the root node we pick the 3rd (0 indexing) child to get the Leaf `[49, 54]` .

## Deleting 49

```
> t
12 24 49 |
2 4 8 | 12 20 | 24 42 | 49 54 |
> f 49
Record at 0x55d757e41cc0 -- key 49, value 49.
> p 49
[12 24 49] 3 ->
Leaf [49 54] ->
Record at 0x55d757e41cc0 -- key 49, value 49.
> d 49
Do you want to delete the whole key? type y for yes: y
12 24 |
2 4 8 | 12 20 | 24 42 54 |
> t
12 24 |
2 4 8 | 12 20 | 24 42 54 |
> f 49
Record not found under key 49.
> t
12 24 |
2 4 8 | 12 20 | 24 42 54 |
```

As we can observe, the key 49 has been removed from the tree, and now we can't search for it as well.

Tree Before Deleting

```
> t
12 24 49 |
2 4 8 | 12 20 | 24 42 | 49 54 |
```

ToString Before Deleting

```
> s
[2, 4, 8] 12 , [12, 20] 24 , [24, 42] 49 , [49, 54]
```

Tree after Deleting

```
> t
12 24 |
2 4 8 | 12 20 | 24 42 54 |
```

ToString after Deleting

```
2 4 8 | 12 20 | 24 42 54 |  
> s  
[2, 4, 8] 12 , [12, 20] 24 , [24, 42, 54]  
<
```

---

ThankYou :D