

# CMSC 420: Coding Project 3

## B-Trees

### 1 Due Date and Time

Due to Gradescope by Sunday 19 March at 11:59pm. You can submit as many times as you wish before that.

### 2 Get Your Hands Dirty!

This document is intentionally brief and much of what is written here will be more clear once you start looking at the provided files and submitting.

### 3 Assignment

We have provided the template `btree.py` which you will need to complete. More specifically you will fill in the code details to manage insertion, deletion, and search for B-trees.

As a slight change from the previous two projects we have implemented a `Btree` class as well as a `Node` class. The `Btree` class stores only the  $m$  value as well as a pointer to the root `Node`. The `insert`, `delete`, and `search` functions are then implemented as methods of the `btree` class.

As is noted in the `btree.py` template you are welcome to make minor augmentations to the `Node` class, for example if you wish to put in a `parent` variable to keep track of a node's parent.

Please look at this file as soon as possible.

## 4 Details

The functions should do the following:

- `def insert(self, key: int):`  
Insert the key into the tree and rebalance as per B-tree rules. The key is guaranteed not to be in the tree.
- `def delete(self, key: int):`  
Delete the key from the tree and rebalance as per B-tree rules. The key is guaranteed to be in the tree.
- `def search(self, key: int):`  
Calculate the list of child indices followed on the path from the root to the node which includes the key. If the key is in the root, return []. The key is guaranteed to be in the tree.

## 5 Order of Operations

For insertion, follow these steps when correcting an overfull node:

1. Use a left rotation if possible and effective.
2. Use a right rotation if possible and effective.
3. Split.

For deletion, follow these steps when correcting an underfull node:

1. Use a right rotation if possible and effective.
2. Use a left rotation if possible and effective.
3. Merge with left sibling if possible and effective.
4. Merge with right sibling if possible and effective.

## 6 Additional Functions

You will probably want some helper functions as well as `Btree` class methods to handle the rotations, merging, and splitting.

## 7 What to Submit

You should only submit your completed `btree.py` code to Gradescope for grading. We suggest that you begin by uploading it as-is (it will run!), before you make any changes, just to see how the autograder works and what the tests look like. Please submit this file as soon as possible.

## 8 Testing

This is tested via the construction and processing of tracefiles.

- The first line in the tracefile is `initialize,m` which should initialize an instance of the `Btree` class with  $m$  and with root node `None`.
- Each remaining non-final line in a tracefile is either `insert,key` or `delete,key`. All together these lines result in the creation of a B-tree.
- The final line is either `dump` or `search,searchkey`.

You can see some examples by submitting the `btree.py` file as-is.

## 9 Local Testing

We have provided the testing file `test_btree.py` which you can use to test your code locally. Simply put the lines from a tracefile (either from the autograder or just make one up) into a file `whatever` and then run:

```
python3 test_btree.py -tf whatever
```