DSC 395T Algorithms and Data Structures — Spring 2022	April 1
B-Trees	Programming Assignment #2

In this assignment, you will implement a B-Tree data structure, which will give you experience in creating your own specialized data structure as well as the challenge of finding and fixing code bugs.

1 B-Trees

Recall from lecture that B-Trees are an important data structure that are used to store large (and at times distributed) dictionaries, such as filesystems and databases. Thus, B-Trees support the three Dictionary operations:

- `find(key) -> value`: Retrieve the value associated with the given key, if one exists.
- `insert(key, value)`: Insert the given key into the dictionary and associate with it the given value.
- `delete(key)`: Delete the given key and its associated value.

To minimize disk accesses, B-Trees are short and wide m-ary trees, where m is an integer ≥ 2 that is chosen to fit as many keys and pointers into a node as possible, where the size of each node is defined by the size of a disk block. Similarly, the constant l is chosen to pack as many data items as possible into a disk block.

Recall the definition of a B-Tree:

- Each node has at most m children
- Each non-leaf and non-root node has at least $\lceil \frac{m}{2} \rceil$ children
- A non-leaf node with k children contains $k - 1$ comparable keys
- The root, if it isn't a leaf, has at least 2 children
- Each leaf node holds between $\lceil \frac{l}{2} \rceil$ and l data items
- If the leaf is the root, it can have a minimum of zero data items
- All leaves reside at the same level

2 Your Assignment

Building on the starter code in `tree.py`, create a working implementation of a B-Tree dictionary that supports the `find` and `insert` methods. To simplify your task, we are not asking you to implement the `delete` method. We also are not allowing duplicate keys in the B-Tree; inserts with the same key should overwrite old values.

The main, overarching goal of this assignment is to be able to grasp the concept of disk vs. memory. You will not be able to store the nodes of your B-Tree in memory, as your natural instincts would lead you to do. Instead, the point of a B-Tree is to have each node reside in its own disk block, which means that the nodes can only keep their structure by storing the disk block addresses of other nodes. For example, a node could be stored in disk block 1, whereas its children are stored in blocks 2, 3, and 4. The node's data would consist of two keys (in your assignment, of type `KT`) as well as a list of addresses of its three children: `[2, 3, 4]`. The only data that you will be allowed to store in memory is the B-Tree object itself, which provides an interface for the dictionary operations. However, this object can only store three things: the value of m, the value of l, , and the disk address of its root node. All other information about the tree must be stored on disk.

2.1 Interacting with the Disk

To avoid making you go through the cumbersome process of obtaining and writing to disk blocks, we have abstracted away the disk interface through the `disk.py` module. You should not touch any of the code in this file, and all you will need from it is `from disk import DISK`, which has already been added for you. This `DISK` variable is a singleton disk instance, which provides the following methods:

- `DISK.new() -> address`: Allocates a new disk block and returns its address. Use this to create an address for a new node.
- `DISK.write(address, BTreeNode)`: Writes the given BTreeNode to the disk at the given address.
- `DISK.read(address) -> BTreeNode`: Returns the node that you last wrote to that address.

2.2 Correctness

Be sure that your B-Tree implementation meets the following requirements:

- Many items can be inserted in a variety of orderings and with a variety of m-values and l-values without causing an error.
- Each item that is inserted can be subsequently found using `find`.
- Your B-Tree's data can be entirely recovered from everything that is written to disk, as well as the m-value, the l-value, and the address of the root.
- Your B-Tree should store keys in ascending order, so that it is possible to lookup a child address in logarithmic time using binary search.

You should again extend upon the given test cases in `tests/test_btrees.py`. Since there is no GUI, you will need to rely on writing test cases to convince yourself that your code works as expected. Run the test suite with `poetry run pytest -v`. To run your code in the interactive Python interpreter, run `poetry run python3` outside of the `tests` and `py_btrees` folders, and import your BTree code with `>>> from py_btrees.btree import BTree`.

3 Karma

One obvious way to challenge yourself is to implement the `delete` method.

4 What to Turn In

Use the Canvas website to submit a single ZIP file that contains your report and source code.

Source code: Make sure all classes are in the correct files. Also make sure that your submission is using the correct directory structure. (The same directory structure provided by the starter code.)

Acknowledgments. This assignment was created by Calvin Lin, Leo Orshansky, James Rayman, and Elvin Yang.