Pair Programming

You are required to work with your assigned partner on this assignment. You must observe the pair programming guidelines outlined in the course syllabus — failure to do so will be considered a violation of the Davidson Honor Code. *Collaboration across teams is prohibited and at no point should you be in possession of any work that was completed by a person other than you or your partner.*

1 Introduction

On this assignment, you will implement a file-backed in-memory B+Tree and integrate it into MicroDB.

1.1 Material

You received with this assignment a MicroDB implementation with a B+Tree skeleton incorporated into the file structure[1]. You will complete the skeleton, which consists of files in `indexes.bptree` submodule.

Attention! You are implementing a data structure that is different from what you saw in CSC 221. Any new node in your B+Tree created in your program should be obtained from the BPNodeFactory, and it will have a **number** that never changes associated to it. The BPNodeFactory does that association. If you want to follow a link between a parent and a child, the parent stores only the **numbers** of the children, and you should ask the BPNodeFactory for the actual (memory) node when you need it. Your initial BPNodeFactory code is not doing much, but later on in the assignment you will implement a disk-backed structure similar to the BufferManager in MicroDB.

2 (20 pts) Search

Due: Nov 17

In the file `BPTree.java`, implement the get() method, which returns the leaf node in which a new key-value pair should be inserted, and the get() method, which returns the value associated with a particular key. The get() method should call find(), obtain the leaf node associated with the search key, and then look for the value associated with that key inside that leaf node. If the key is not present, get() should return **null**.

[1] When the skeleton code for a project has 3,500 lines of code, we should call it "boneyard code." – HM.

The logic for both find() and get() is outlined in our textbook in Sec. 14.3.2. The textbook describes the logic of find() and get() in a single method there called find(), but we separate these two methods here (calling them find() and get()).

In summary, there are two methods to implement:

- get() in `BPTree.java`. Use the implemented functions `more()`, `less()`, or `equal()` to help you navigate through the tree.
- find() in `BPTree.java`. This function only calls `get()` to find a terminal node and then goes through the node to see if the key of interest is there.

The full find/get algorithm is described in Sec. 14.3.2 of our textbook.

3 (30 pts) Insertion

Due: Nov 17

In the file `BPTree.java`, implement the insert() method for the B+Tree, outlined in our textbook in Sec. 14.3.3.1. There are four methods to implement in this section:

- insert() in `BPTree.java`
- insertOnParent() in `BPTree.java`
- splitLeaf() in `BPNode.java`
- splitInternal() in `BPNode.java`

Here are some overall pointers for implementing these methods:

- The insert() method should find the location where to insert the new key/value pair by calling find(), implemented above. Call that node insertPlace.
- After you insert the key-value pair in insertPlace, you should check if the node has just overflown. The arrays storing keys/values/children in `BPNode.java` contain each one more entry than the maximum allowed, because you might prefer to add the key-value or key-child pair **first**, and **only then** split the node.
- If a split is required, you should use the splitLeaf() method over insertPlace. This method is not implemented, so you will have to complete its implementation. The method should return a SplitResult object such that:
  1. The left field contains a pointer to the left node in the split operation;

2. The right field contains a pointer to the right node in the split operation;
3. The keyDivider field contains the first key of the right node after the split operation.

- If you split the node, you are required to insert the first key of the right split into the parent node. implement the insertOnParent() method similarly to what has been described in the book. If you are required to split an internal node, you should use the splitInternal () method over such internal node. This method is not implemented, so you will have to complete its implementation. The method should return a SplitResult object such that:

  1. The left field contains a pointer to the left node in the split operation;
  2. The right field contains a pointer to the right node in the split operation;
  3. The keyDivider field contains the key that will subsequently be inserted once again in the parent node, recursively. The key inserted in the parent node will have its associated pointer referring to the right-side of the previously splitted node.

  You are going to need to create new nodes for the root as you insert more and more key-value pairs. **Important.** Any new node created in your program should be obtained from the BPNodeFactory. We are doing this because later on you will modify this class to behave similarly to the `BlockManager` of MicroDB, effectively saving the index in a file!

The full insertion algorithm is described in Sec. 14.3.3.1 of our textbook. You can implement these pointer changes manually or with the help of the `insertChild()` function.

4 (25 pts) Load and Save

Due: Dec 7 (Work not allowed on Thanksgiving week)

You should implement load() and save() methods in `BPNode.java`. These methods should "translate" between the disk representation and the in-memory representation of a BPNode.

- The save() method receives a 512-sized ByteBuffer object, and it should fill this buffer with all the information that is required to fully load a BPNode later on. You `cannot` store pointers in the buffer. If you need to store references to other nodes, store the **numbers** associated with each node, as you obtain from BPNodeFactory's getNumber () method. You will need to pass a reference to the BPTree's BPNodeFactory to this method, so feel free to add the appropriate parameters for this function.

- The load() method receives a 512-sized ByteBuffer object, and it should fully load a BPNode from it. You can obtain references from the node numbers stored in a buffer by calling BPNodeFactory's getNode() method. Since the code in BPNode is generic, the load method should receive two parameters to convert from the inevitable string representation of keys and values into actual objects of type K and V. Call these parameters loadKey – of Java type Function<String, K> – and loadValue – of java type Function<String, V>.

- To test load() and save(), create a new BPNode, and a new 512-sized ByteBuffer. Save node *a* into the buffer by using a.save(buffer) and then load node *b* from t he buffer by using b.load(buffer). The code sample below demonstrates your test workflow. After this operation, the two nodes should have the same contents. Test with leaf and internal nodes.

  Look into `Block.java` from MicroDB for inspiration!

Here is some test code that should work with the load/save functionality:

```
// First node is the root
BPNode<String, Integer> node1 = testIndex.root;
// Second node is some leaf
BPNode<String, Integer> node2 = testIndex.find(testIndex.root, "a");

ByteBuffer buffer1 = ByteBuffer.allocate(512);
ByteBuffer buffer2 = ByteBuffer.allocate(512);

// Save the internal node (the root) and the leaf
node1.save(buffer1, testIndex.nodeFactory);
node2.save(buffer2, testIndex.nodeFactory);

// Create two empty nodes where you'll load
// the internal node (the root) and the leaf
BPNode<String, Integer> newNode1 = new BPNode<>(null, false);
BPNode<String, Integer> newNode2 = new BPNode<>(null, true);

newNode1.load(buffer1, testIndex.nodeFactory, k -> k, s -> Integer.
    parseInt(s));
newNode2.load(buffer2, testIndex.nodeFactory, k -> k, s -> Integer.
```

```
    parseInt(s));

// Print the original and loaded ones for comparison
System.out.println("Original root: " + node1 + ", parent = " + node1.
    parent + ", next = " + node1.next);
System.out.println("Original leaf: " + node2 + ", parent = " + node2.
    parent + ", next = " + node2.next);

System.out.println("Loaded root: " + newNode1 + ", parent = " +
    newNode1.parent + ", next = " + newNode1.next);
System.out.println("Loaded leaf: " + newNode2 + ", parent = " +
    newNode2.parent + ", next = " + newNode2.next);
```

5 (25 pts) Disk-Backing the B+Tree

Due: Dec 7 (Work not allowed on Thanksgiving week)

Your current implementation of BPNodeFactory currently creates and maintains all BPNodes in memory. As the BPTree code requests for nodes (based on their numbers), a simple map returns the corresponding nodes associated with a particular number.

You must change the BPNodeFactory class to implement a disk-backed memory factory for BPNode. Here are the components of your implementation:

- Implement a readNode() method, that receives a node number x, and reads a chunk of `DISK_SIZE` bytes into a ByteBuffer, corresponding to the in-disk node representation. This chunk should be the x-th chunk (0-based indexing) of `DISK_SIZE` bytes counting from the beginning of the file. After this chunk is read, create a new BPNode in memory, and call that node's *load()* method. Look into `Relation.java` for inspiration for readNode().

- Implement a writeNode() method, that receives a node object, and creates a ByteBuffer of `DISK_SIZE` bytes in memory, calling the node's *save()* method to populate this buffer. Then, write the buffer to its appropriate position in the file (based on the node's number). Look into `Relation.java` for inspiration for writeNode().

- Implement the create() method, that:

  1. Allocates a new BPNode object, associating it with a new node number;

2. Inserts a mapping from the new node number to a NodeTimestamp object, a class that contains nodes and timestamps, in a similar way that `BlockManager.java` does with its own BlockTimestamp object. This mapping is done at a HashMap called nodeMap.
3. Adds the NodeTimestamp object associated with the new node into a priority queue (use "utils.DecentPQ"), in a similar way that `BlockManager.java` does with its own priority queue.

- Implement the getNode() method, which receives a node number, and
  1. If the node is loaded in memory (check nodeMap), return that node, but making sure to update the timestamp associated with it. Look into `BlockManager.java` for inspiration.
  2. Otherwise, read the node from disk, using readNode(), and setup a *NodeTimestamp* objects in the same way as create() does.
- Implement the evict() method, which removes the least recently used node from nodeMap (and from the priority queue), but making sure to update the *disk* with the contents of the evicted node. Use the writeNode() method to update the disk with the in-memory representation of the evicted BPNode.

Good luck,
- Hammurabi