

## Deliverables

There are two source-code deliverables:

1. `BPTree.java`
2. `BPNode.java`

## 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<sup>1</sup>. 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.

---

<sup>1</sup>When the skeleton code for a project has 3,500 lines of code, we should call it “boneyard code.” – HM.

## 2 (15 pts) Search

Due: Thursday, Apr 18 23:55

In the file `BPTree.java`, implement the `find()` 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**.

The logic for both `find()` and `get()` is outlined in our textbook in Sec. 11.3.2. Our 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:

- `find()` in `BPTree.java`
- `get()` in `BPTree.java`

The full find/get algorithm is described in Sec. 11.3.2 of our textbook.

## 3 (20 pts) Insertion

Due: Thursday, Apr 18 23:55

In the file `BPTree.java`, implement the `insert()` method for the B+Tree, outlined in our textbook in Sec. 11.3.3.1. 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 overflowed. 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. 11.3.3.1 of our textbook.

## 4 (20 pts) Load and Save

Due: Thursday, Apr 18 23:55

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 the 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 (20 pts) Disk-Backing the B+Tree

Due: Wednesday, May 1 23:55

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`.

## 6 (15 pts) Integrating with MicroDB: Creating and Populating the Index

Due: Wednesday, May 1 23:55

In `BPTreeIndex.java`, implement the constructor by creating and populating a B+Tree (insert that as a class attribute) that maps keys of type `K` to `RecordLocation` objects, which are defined in the “microdb.indexes” package.

You should create a single attribute in the class, a `BPTree<K, RecordLocation>`. In this B+Tree, as usual, keys and values are stored in the disk using a string representation. The `BPTreeIndex` constructor receives from the user a `loadKey` function, that maps from Strings to objects of type `K`, which could be repassed for the constructor of `BPTree`. The `loadValue` function, that maps from String to objects of type `RecordLocation`, should be specified by *you*. It should be a function mapping strings such as “(1, 3)”, representing a record located at block 1, record 3, into a **new** `RecordLocation` object having 1 and 3 in their appropriate fields.

## 6.1 Testing

**To test your disk-backing.** You can run the `main()` method from `BPTree.java`. **Warning:** if your `BPNodeManager` has a very low capacity (something smaller than 15), changes in memory done to nodes that end up being evicted could be “lost”, causing corruption in the data structure. This situation would only occur if the buffer manager has a capacity smaller than, say, 15 objects. In real life, with a buffer manager holding a couple hundred megabytes worth of nodes. If you maintain your `BPNodeManager.CAPACITY` over 10 nodes or so, you should not encounter any problems. If you want to test having a `BPNodeManager` with a very small capacity, **every time you evict the node to the disk make sure that you save it first!**.

**To test your index.** The file `TestBP.java` has been included in your skeleton code. This test file creates an instance of `BPTreeIndex` and an instance of `SecondaryIndexMaterialized` and compares the output from both, with some random queries. The output should be the same in all cases.

## 7 (10 pts) Style

The code will also be graded on style according to the following guidelines. The Java files are well-commented (using Javadoc), well-indented and concise.

Good luck,  
- Hammurabi