Brandeis University
Department of Computer Science
COSI 127B, Introduction to Database Systems

Programming Assignment #3

Out: March 26, 2018
Due: April 16, 2018

1 Introduction

PA1 and PA2 were designed to teach you about the skills involved in being a database administrator. PA3 will focus on the systems side of databases. You will be asked to implement in Java a B+-Tree that conforms to specific guidelines.

2 Background

A B+-Tree is a balanced tree where every path from the root of the tree to the leaf of the tree is the same length. For a B+-Tree of degree n, the following constraints must hold:

- At the root node, the maximum number of pointers is `n` and the minimum number of pointers is 2.
- At an internal node (not the root and not a leaf), the maximum number of pointers is `n` and the minimum number of pointers is `n/2`, rounded up.
- At leaf nodes, the maximum number of values stored is `n-1`, and the minimum number of values stored is `(n - 1)/2`, rounded up.

The specifics of B+-Trees have been covered in class.

2.1 The B+-Tree Classes

This project requires you to work with several Java classes.

- `Main`: this class provides two important methods: main, which you can use to test your B+-Tree at the command line, and buildTree, which will be called by the tests. You should not need to change this class.
- `BTree`: this class contains the outwards-facing functionality of the B+-Tree (insert and delete). It also contains a method called outputGraphviz, which you can use for rendering your tree graphically, if you so desire. You should not need to change this class.
- `Node`: this is the superclass of the two types of nodes in the B+-Tree (`InternalNode` and `LeafNode`). Many of the methods of this class are implemented for you and documented. Some of them will likely be very useful. Unlike in class, our nodes have pointers to both their next sibling (next) the their previous sibling (prev). You will need to implement the marked methods in this class.
- `InternalNode`: inherits from Node and represents the internal nodes of the tree. You will need to implement the marked methods in this class.

- `LeafNode`: inherits from Node and represents the nodes at the bottom of the tree. For this exercise we will ignore the file blocks that leaf nodes point to.
- `Reference`: contains information about references to nodes or to values

3 Task Descriptions

You will need to complete the implementations of 3 classes: `Node, InternalNode and LeafNode`. Classes `Reference, BTree and Main` can be left as is they are fully implemented. You only need to read and understand them.

Some of the important methods to be implemented by you along with descriptions are as follows:

- `InternalNode::search(int val)` returns `Reference`: This method will call `findPtrIndex(val)` (specified in class `Node`), which returns the array index i, such that the ith pointer in the node points to the subtree containing val. It will then call `search()` recursively on the node returned in the previous step.
- `InternalNode::insert(int val, Node p)` returns void: This method calls the following routines, which you will implement:
  - `findKeyIndex(val)`: (specified in the class `Node`), this method returns the array index i in the key array such that keys[i] is the largest key that is less than or equal to val.
  - `insertSimple()`: this method is called when the current node is not full
  - `redistribute()`: This method is called when the node is full. In class, algorithms presented involved creating a supernode and splitting it into two. In this implementation, you will instead create a new InternalNode, and distribute the keys and pointers in the full node (as well as the value and pointer to be inserted) so that the old node and the new node are each half full. redistribute() is the method you will call to trigger this movement of data between nodes. This method should return the key value to be inserted into the parent node as a result of the split.
- `LeafNode::insert(int val, Node p)` returns int: This method calls the following routines:
  - `findKeyIndex()`: (specified in the `Node` class)
  - `insertSimple()`: (called when the current node is not full)
  - `redistribute()`: This method is as defined like that in `InternalNode` except that it creates a new LeafNode and the key returned is the first key of the new node.
- `Node::delete(int i)` returns void: This method calls the following routine which you will need to define as:(all of the following methods are abstract methods and should be implemented in both `InternalNode` and `LeafNode` class)
  - `deleteSimple()`: this method removes the ith key and pointer from the key array and pointer array of the node.
  - `combinable(Node other)`: This method is used to test if this node can be combined with other node without splitting.
  - `combine()`: This method combines the current underfull node with the sibling to its right. This method should be called only if this node is `combinable()` with its right sibling.
  - `redistribute()`: If current node cannot be combined with one of its siblings, use this method to distribute its keys with its next sibling.
- `Node::redistribute()` returns int: This method moves some of the keys and pointers from this node to its next sibling. There are no parameters here because each node knows its next sibling. If p is the current node:
  - To redistribute keys and pointers of p with its next sibling, use `p.redistribute()`.

  - To redistribute keys and pointers with ps previous sibling, use `(p.prev).redistribute()`.In this case, ps previous sibling moves some of its keys and pointers to p.
  - To split a full node, p, into two nodes:
    1. create a new node, p' (p' is between p and p.next).
    2. call `p.redistribute()` to move keys and pointers from p to p.

- `Node::combine()` returns void: given a node, p, `p.combine()` moves all keys and pointers of ps next node to p. If p is the current node to combine with ps next node, use `p.combine()`. If p is the current node to combine with ps previous node, use `(p.prev).combine()`. This call moves all keys and pointers from p to the previous sibling of p.

4 Getting Started

4.1 Get the files

First, get the files from LATTE. You should have a folder called CS127_PA3, which can be imported as an Eclipse project.

4.2 Implement the API

All code that you will be working with is found inside the "src" directory. You should implement the methods mentioned in the previous sections and any methods with a "ADD CODE HERE" note. You can also create any other methods as you like. Use the API and notes from the previous section as a guide for your implementation.

There are several requirements for your program:

- If a node is underfull and can be merged with either of its siblings, merge it with the sibling to its right (next sibling).
- When a node is split into two nodes, the first node should have at least as many keys and pointers as the second node.
- Every key in an internal node must also appear in a leaf node. This means that each time you delete a key from a leaf node, you should update the internal nodes if necessary.
- When a node becomes underfull after a deletion, if it can be combined with one sibling and redistributed with another sibling, choose combining with its sibling. In other words, in your delete() method, the process that deals with an underfull node should be:

```
if (siblings(next) &$ combinable(next)){
// combine with next sibling
} else if (siblings(prev) && combinable(prev)){
// combine with previous sibling
} else if (siblings(next)){
// redistribute with next sibling
}else if (siblings(prev)){
//redistribute with previous sibling
}
```

4.3 Debugging

The folder data contains 5 example command sets that can be feed to your program via standard in. Notice that each contains the o and p commands, which will cause the tree to be printed out to the terminal and for a GraphViz file called tree.dot to be created. You can turn tree.dot into a graphical representation of your tree like this:

```
dot -Tpng tree.dot >tree.png
```

You can then compare the tree generated by your code to the correct answers, which are stored in the results folder.

4.4 Testing

The JUnit tests ran by `AllProvidedTests` will use the `buildTree` method of the `Main` class in order to construct a tree. This tree will be tested against the correct values. A good submission should pass every test.

4.5 Submitting

If you would like to leave us any notes, such as reasons a test is failing, for possible partial credit, please leave them as comments in Main.java.

To submit, simply export your project from Eclipse by right-clicking on your project folder, selecting Export, and then choosing the General ->Archive File option. Please name your submission with your first name and your last name followed by _CS127PA3.zip. For example, ZhanLi_CS127PA3.zip. Use LATTE to submit your ZIP archive.

Do not simply send your src folder. Do not extract each of your source files individually. Do not use .rar or .7z. Do not package the Eclipse exported archive inside of another archive. Using the export feature in Eclipse (as described) will ensure that we will be able to grade your submission.

5 Tips

- Start early.
- As a first step, understand the insertion and deletion algorithms. Try a few on paper. Do Not Code anything until you're convinced how this works. You will only waste time.
- It's possible to get a correct output for one test, while failing others. Be sure to run all of the tests provided, as well as any others that you generate.
- Pay close attention to the project requirements in section 4.2 of the project handout.
- For debugging purposes you may find it helpful to modify the default print functions associated with LeafNodes and Internal Nodes. Feel free to do so.