Programming Assignment 3: B+-Trees
CS127B - Introduction to Database Systems, Spring, 2014

Assignment Out: Monday, April 7th @ 5:00PM
Assignment Due: Wednesday, April 23rd @ 3:30PM (electronic submission to Latte)

1 Introduction

Having completed PA1 and PA2, you should now be well-versed in the DBA skill of database design. This assignment is intended to give you some experience with the systems-side of databases, and in particular, with B+-Tree indexes.

For PA2, you witnessed the tradeoff between query performance and maintenance overhead when constructing indexes. The purpose of PA3 is to give you a better understanding of this tradeoff by designing your own B+-Trees. You will be asked to implement a set of Java classes that together implement a B+-Tree. To make your life easier, you will be provided an API for the classes you need to implement. You need to understand what each method in the API is for and implement it according to the appropriate algorithm.

2 Background

A B+-Tree is a balanced tree in which every path from the root of the tree to the leaf of the tree is of the same length. Assuming a B+-Tree of Order n, every node must be populated according to the requirements listed below:

	Max	Min
root node	n pointers	2 pointers
non-root internal nodes	n pointers	$\lceil \frac{n}{2} \rceil$ pointers
leaf nodes	n - 1 keys	$\lceil \frac{n-1}{2} \rceil$keys

2.1 The B+-Tree Classes

An API (JavaDoc) for the classes required to implement B+-Trees can be found at Latte. The classes in the API include:

- **Main:** The user interface class, which interactively accepts commands (to insert, delete and print your index tree) and invokes the appropriate methods accordingly.
- **BTree:** The "main" class. All calls of the index (insertions, deletions, searches) are made via method calls on objects belonging to this class. This class also defines the method, "outputforGraphviz" which you will be able to call to generate a visualization of your index (for debugging purposes). You will not need to make any changes to this class.

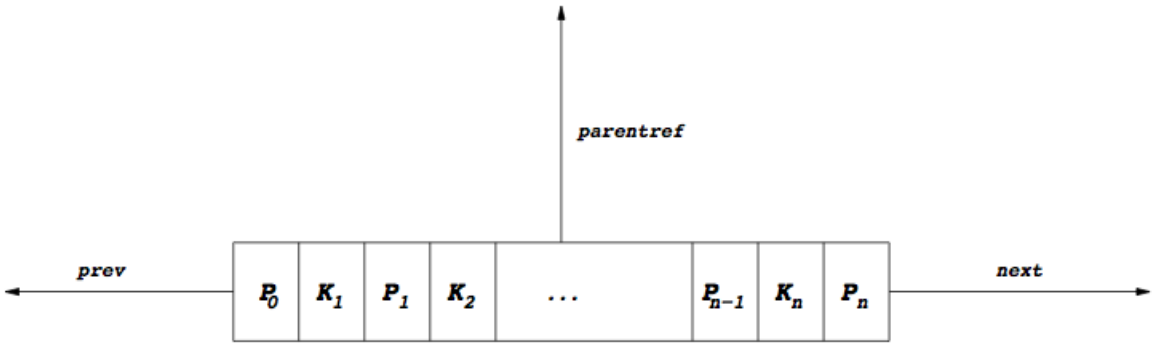

Figure 1: The node structure

- **Node:** Node is the superclass of both types of nodes in a B+-Tree: InternalNode and LeafNode. Figure 1 shows the structure of a Node. The interface for Node includes the following implemented methods:

  Node methods implemented for you:

  - int getLast()
  - int getKey(int i)
  - int getPtr(int i)
  - Node getPtr(int i)
  - Node getNext(); void setNext(Node n)
  - Node getPrev(); void setPrev(Node p)
  - Reference getParent(); void setParent(Reference r)
  - int maxkeys()
  - int minkeys()*
  - boolean full()
  - boolean siblings(Node other)
  - void setInitPtr(Node p)
  - boolean combinable(Node other)
  - void combine()*
  - int redistribute()*
  - void insert(int val, Node ptr)*
  - void insertSimple(int val, Node ptr, int i)*
  - void delete(int i)
  - void deleteSimple(int i)*
  - Reference search(int val)*

  Node methods you must implement:

  - void delete(int i)
  - int findKeyIndex(int val)
  - int findPtrIndex(int val)

  * = abstract methods implemented in subclasses, InternalNode and LeafNode. See JavaDocs for additional documentation.

  Note: The node structure is slightly different from the one presented in the class. In class, a node did not have a pointer pointing to its previous node. In this program, a node has both the pointer pointing to its previous sibling and the pointer pointing to its next sibling. Given a Node p, p.prev refers to p's previous Node and p.next refers to p's next Node.

- **InternalNode:** **InternalNode** inherits from **Node**. You will need to implement the following methods:
  - int minkeys()
  - boolean combinable(Node other)
  - void combine()
  - int redistribute()
  - void insert(int val, Node ptr)
  - insertSimple(int val, Node ptr, int i)
  - deleteSimple(int i)
  - Reference search(int val)

  (All non-null pointers must reference internal nodes or leaf nodes.)
- **LeafNode:** **LeafNode** inherits from **Node**. You will need to implement the following methods:
  - int minkeys()
  - boolean combinable(Node other)
  - void combine()
  - int redistribute()
  - void insert(int val, Node ptr)
  - insertSimple(int val, Node ptr, int i)
  - deleteSimple(int i)
  - Reference search(int val)

  For this exercise we will ignore the file blocks that leaf nodes. Therefore all pointers in leaf nodes are set to null.
- **Reference:** A **Reference** object denotes a location within a node that "points to" another node. For example, Node::getParent(), when called on some node, n, will return:
  1. The parent node of n in the index (p),
  2. The integer, i, such that the i^{th} pointer in p points to n.

  Reference objects are also returned by searches on nodes. For example, a search for some value, v will return the leaf node that contains v and an integer denoting the position of v in the leaf node. Because searches can fail, a **Reference** object also includes a Boolean attribute which is set to **false** if returned by a search that failed, and true otherwise.

3 Your Tasks

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.

Methods to be implemented 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 will call **findKeyIndex(val)** (specified in the class **Node**), which 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. It returns a **Reference** referring to the current node and an array index in the key array. If the ith key in keys array matches **val**, match is set to true in the **Reference** returned, otherwise it is set to **false**.

- **InternalNode::insert(int val, Node p) returns void**
  This method calls the following routines, which you will implement:
  - **findKeyIndex(val):** (in the class **Node**)
  - **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 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:
  - **deleteSimple():** this method removes the i^{th} key and pointer from the key array and pointer array of the node. The method is implemented in
  - **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 it's right. This method should be called only if this node is **combinable()** with it's right sibling.
  - **redistribute():** If current node cannot be combined with one of its siblings, use this method to distribute its keys with it's 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().
  - Tor edistribute keys and pointers with p's previous sibling, use (p.prev).redistribute().
    In this case, p's 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 p's next node to p.
  If **p** is the current node:
  - To combine with **p's** next node, use **p.combine()**.
  - To combine with **p's** previous node, use **(p.prev).combine()**. This call moves all keys and pointers from **p** to the previous sibling of **p**.

4 Getting Started

1. **Get the files**
   By now you have already obtained and unzipped the PA3 files for this project. An overview of the files found in this directory is shown below:

```
README.html
gentree
project_handout.pdf (this file)
src (directory)
    Node.java
    LeafNode.java
    InternalNode.java
    BTree.java
    Main.java
doc (directory)
    javadocs (html files)
data(directory)
    data1
    data2
    data3
    data4
    data5
results (directory)
    data1.pdf
    data2.pdf
    data3.pdf
    data4.pdf
    data5.pdf
```

2. **Implement the API**
   All code that you will be working with is found inside the "src" directory. You should implements the methods mentioned in the previous sections. Use the API and notes from the previous section as a guide for your implementation. If a method has a "ADD CODE HERE" note, you should implement that method too.

   There are several requirements for your program:

   - If a node is underfull and can be merged with either of it's siblings, merge it with the sibling to it's 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
}
```

3. **Test your program**
   Run your B+-Tree program with the test files (data1..5) as input and compare the B+- Trees generated by your program with the B+-Trees on the web.
   The command to run your program with the test file data1 is:

```
$ java Main < ../data/data1
```

4. You may generate random input sequences with gentree for additional test cases. **Note that you must be logged in to a Linux-based cs server at Brandeis to use gentree**. To visualize the BTree you should run your program, create a tree with insertions and deletions, and type 'o' when you want to see the tree. To see a text-based version of the tree, type 'p'. Note that in this case the nodes may not be aligned properly. Use as a guide that siblings are connected with a '–'. Feel free to modify the various Print() functions to suit your debugging needs.

```
$ java Main
n3
i10
i20
i30
...
i32
p
o
```

   The usage of gentree is:

```
$ ./gentree [degree] [percent of insert[0..1]] [number of ops]
```

   For example, the command:

```
$ ./gentree 5 0.75 100
```

   creates an order 5 tree, 75% of the operations being insertions, and 100 operations (in- sertions and deletions) in all.

5 To hand in

You need to submit a digital copy of your code to Latte by the due date listed at the top of this document. When submitting code, please either zip your documents or create a tar file using the following command (from within the PA3 directory).

```
tar cvzf YourName_pa3.tar.gz src YourName_ReadMe.txt
```

Your compressed file should include the "src" directory as well as a text file containing a paragraph explaining the status of your program. In other words, does your program work and produce results in data1 to data 5? If not, what do you think the problem is?

Good Luck!