CSE 214 : Data Structures (Fall 2023) – Assignment III

Instructor: Dr. Ritwik Banerjee
Computer Science, Stony Brook University

This programming assignment is entirely on the B-tree data structure, as discussed in our lectures. It is important to follow the specifications discussed in our lectures, because the literature on B-trees is not entirely consistent with the terminology and design. Following external sources may result in your implementation adhering to a different variation of this data structure, and be deemed incorrect by the grading process.

This assignment is due by 11:59 pm, Dec 5 (Tuesday).

Unlike the previous programming assignments, you do not have to account for interfaces here. In fact, the programming is comparatively much simpler, once you understand the B-tree data structure. *It is strongly advised that you create a plan for how to handle the details of the search and insert algorithms, before you jump into coding.*

1. Write the signature of a `BTree` class such that a `BTree` instance can contain nodes of any data type that allows for comparison. For example, a user should be able to use your class to create a `BTree<Integer>`, a `BTree<Double>`, or a `BTree<String>`. But they should *not* be able to create a `BTree<Exception>`. (5)

2. Write a constructor of your `BTree` class such that its only argument is the minimum degree of the tree. That is, the signature of the constructor must be (5)

   ```
   public BTree(int minimumDegree)
   ```

   You may choose to write additional constructors for your class, but any additional constructors must not be accessible outside the `BTree` class.

   Your constructor must also throw a `java.lang.IllegalArgumentException` if the argument is invalid. For example, trying to create a B-tree with a negative or zero minimum degree should throw an `IllegalArgumentException`.

3. Implement the addition of an element to the B-tree, as discussed in the lectures. In the code, the addition of an element must be accomplished by the user (i.e., the client code) by calling the following method: (50)

   ```
   /**
    * Adds the specified element to this B-tree. If the B-tree already contains this
    * element, this operation has no effect on the tree (i.e., it silently ignores
    * attempts to add duplicates).
    *
    * @param element The element to be added to this B-tree.
    * @throws OperationNotSupportedException If the add operation results in a call to add
    *                                        the element to a non-full node while the node
    *                                        is actually full.
    */
   public void add(E element) throws OperationNotSupportedException
   ```

   You are free to write additional helper methods so that the `add(E element)` method does not get too complicated. In fact, the pseudocode from our lectures did just that: it divided the logic in a way that has a separate method for adding an element into a non-full node. As part of adding an element to

the tree, you will also have to carefully handle node splitting, whenever necessary. Any such additional method must not be public, however.

This is the most important part of your assignment. Your `add` method needs to be correct in order to build a proper B-tree. Without that, your B-tree cannot be used to properly search for an element. There is partial credit in this question for some specific aspects of the addition operation (listed below). This is not an exhaustive list of "what to do for partial credit", but rather, a list of important things you should check. If some of the following are not working properly, it is an indicator that your code needs to be immediately debugged:

(a) correct splitting of the root node
(b) upon encountering a full node, it is always split
(c) the addition operation works correctly for at least the simplest `min` value of 2.

There are two methods that you must add in your `BTree` class:

```
public void addAll(Collection<E> elements) throws OperationNotSupportedException {
    for (E e : elements) this.add(e);
}
```

This is just a simple method so that you (and the grader) can create B-trees faster. As you can see, this method simply takes a collection of elements, and adds them one by one to the tree using *your* `add(E element)` method.

Second, we have a method in the `BTree` class to display the current instance of a B-tree. This method does not show the explicit parent-child relations between the nodes of a tree, but it does display the contents of each node, and prints out the entire tree one level at a time.

```
public void show() {
    String      nodesep = "    ";
    Queue<Node> queue1  = new LinkedList<>();
    Queue<Node> queue2  = new LinkedList<>();
    queue1.add(root); /* root of the tree being added */
    while (true) {
        while (!queue1.isEmpty()) {
            Node node = queue1.poll();
            System.out.printf("%s%s", node.toString(), nodesep);
            if (!node.children.isEmpty())
                queue2.addAll(node.children);
        }
        System.out.printf("%n");
        if (queue2.isEmpty())
            break;
        else {
            queue1 = queue2;
            queue2 = new LinkedList<>();
        }
    }
}
```

The `Node` data type used in the above code refers to the node type for the `BTree` class. Notice that this node type is not parameterized, even though `BTree` requires a parameter (as per Q.1 in this assignment). This gives an important clue about where the `Node` class should be written. The `show()` method must run exactly as it is provided. Which means, your code must abide by these three rules:

(1) the root of the tree must be called `root`,
(2) the child nodes of a node must be stored in a variable called `children`, and
(3) the `toString()` method of the `Node` class must return the string representation of its elements.

You are also being given a simple driver method to test your code:

```
public class BTreeRunner {
    public static void main(String[] args) throws OperationNotSupportedException {
        BTree<Integer> tree = new BTree<>(3); /* minimum degree is 3 */
        tree.addAll(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 11,
                                  12, 22, 19, 25, 100, 88, 64, 65, 16));
        tree.show();
    }
}
```

If your tree is properly constructed, the `show()` method should print

```
[9]
[3, 6]    [19, 64]
[1, 2]    [4, 5]    [7, 8]    [11, 12, 16]    [22, 25]    [65, 88, 100]
```

4. Recall that the search operation on a B-tree doesn't just return true/false. It actually returns the node where an element is found, along with the index of the element in that node's contents. To accomplish this, a data type is needed to encapsulate the node and the index together. Write a class called `NodeAndIndex` to do this. (10)

   The inner details of this class are completely up to you. However, it *is* important to have a standard string representation of the encapsulation. Thus, your implementation must override the `toString()` method defined in `java.lang.Object`. See the output of the search operation described in Q.5 for the format of this string representation.

   Another important thing to figure out in this question is where to define this `NodeAndIndex` class. Since client codes will use the search functionality of your B-tree, your implementation of the search operation will need to return an instance of the `NodeAndIndex` class. Therefore, this class needs to be public. On the other hand, any instance of this class must refer to a specific node, which in turn depends on an instance `BTree`. So, when you implement this class, ask yourself how you can make it public while still making sure that it depends on an instance of the `BTree` class.

5. Finally, your last task is to implement the search operation for the B-tree. This must be implemented such that a client code can find an element using the following method: (30)

```
/**
 * Search for the specified element in this B-tree. If the element is found, return
 * the node where it is located along with the index of the element in the contents
 * of that node.
 *
 * @param element The specified element being searched in this B-tree.
 * @return A {@link NodeAndIndex} instance, containing the node where the element was
 * found and its index in the node's contents. If the element was not found in this
 * tree, <code>null</code> is returned.
 */
public NodeAndIndex find(E element)
```

   Similar to the `add(E element)` method's implementation, you can have additional helper methods in your class (but such additional methods should not be `public`). You should, of course, use the given driver method to test your search operation. For example, we can do this with the same tree we used to check the `show()` method earlier:

```
public class BTreeRunner {
    public static void main(String[] args) throws OperationNotSupportedException {
```

```
        BTree<Integer> tree = new BTree<>(3);
        tree.addAll(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 11,
                                  12, 22, 19, 25, 100, 88, 64, 65, 16));
        System.out.println(tree.find(65));
    }
}
```

This should print: `<[65, 88, 100], 0>`.

Very important notes:

1. The correctness of your code will be tested with various test cases, and NOT by reading through your code. Throughout the semester, we have discussed the role of a client code. You should think of the grading code as a client code, which means that the grader will work under the assumption that the contractual instructions (e.g., return type, method signature, etc.) have been followed.
2. Use the driver `main` method given to you. This method is given to you so that you can test your own code exactly as it will be tested by the grading process.
3. Keep in mind that if your submitted code doesn't compile, it cannot be graded! It is better to submit code where *some* methods work properly and can be graded by the driver method (or, for the second part, the tree printing method) than to submit a lot of code that doesn't compile or doesn't abide by the provided guidelines.

What to submit?

Only the contents of your `src` folder, compressed into a single `.zip` file. Please remember to create this `.zip` even if all your code is in just a single `BTree.java` file. In particular, remember that your submission should not include any of the following:

- compiled `.class` files
- local settings such as `.idea` folder or `.iml` or `.classpath` files
- package directories.

This assignment is due on Brightspace by 11:59 pm of Dec 5 (Tuesday).