Data Structures (CSE 214) – Homework III

Dr. Ritwik Banerjee
Computer Science, Stony Brook University

This homework document consists of 4 pages. Compared to the previous assignments, this assignment is much more focused and requires less coding. The data structure (B Tree) itself, however, is somewhat more complex. As mentioned in the syllabus, you should use **Java 17 LTS**, the latest long-term support release for the Java SE platform. As the development environment, you should use IntelliJ IDEA.

Since B-tree is a more complex than, say, linked lists or standard binary trees, we will restrict this assignment to only the very basic operations. The core requirements of the B-tree are given by the following interface, which has only two methods:

```
package cse214hw3;

/**
 * This is the interface providing the public contract for any B-tree implementation. It comprises of only
 * two methods:
 * <ol>
 *     <li>contains(E element)</li>
 *     <li>add(E element)</li>
 * </ol>
 * @param <E> the type of elements in this tree; which must implement the <code>java.lang.Comparable</code>
 *            interface (since B-trees require a totally ordered data type).
 */
public interface AbstractBTree<E extends Comparable<E>> {

    /**
     * Follows the search algorithm for B-trees to determine whether the specified element exists in this
     * B-tree. If no no element is found, <code>null</code> is returned.
     *
     * @param  element the specified element to be searched.
     * @return a node-index pair that consists of the node in this tree where the specified element is
     *         found, and the index of this element in that node; <code>null</code> if the element is
     *         not present in this tree.
     */
    NodeIndexPair<E> contains(E element);

    /**
     * Add the specified element to this B-tree, using the insertion algorithm for B-trees. If this element
     * already exists in this tree, then calling this method has no effect on the tree.
     *
     * @param element the specified element to be added to this tree.
     */
    void add(E element);
}
```

In the lectures, the code presented to you was written using array-based implementations for the B-tree and the B-tree node. Arrays, however, are not very well suited for working with generic programming in Java. The reason behind this is beyond the scope of our syllabus, but suffice it to say that if you choose to use arrays to store the values and the child nodes, you may have to be extra careful with typecasting in many

places in your code. It is highly recommended (not mandatory, however) that you use lists instead. After all, the constant-time retrieval of elements can be achieved through `ArrayLists`!

The next code given to you is the `NodeIndexPair` class, which is used as the return type of the `contains(E element)` method in the `AbstractBTree` interface:

```
package cse214hw3;

/**
 * A simple pair object, where one item of the pair is a B-tree <code>Node</code> <code>n</code>, and the
 * other item is an integer index. The integer index is meant to specify the position of an element in the
 * node <code>n</code>.
 *
 * @param <E> the element type of the node in this pair.
 * @apiNote   The parameter type is not bound by <code>java.lang.Comparable</code> so that any future use of
 *            this pair remains possible for other data structures that may not require total ordering.
 */
public class NodeIndexPair<E> {
    public final Node<E> nodeLocation;
    public final int     index;

    public NodeIndexPair(Node<E> n, int i) {
        this.nodeLocation = n;
        this.index        = i;
    }
}
```

Your first task is to get the `Node` class ready. This will be the building block of your B-tree. The internal implementation details are largely left to you, but it is very important that the following code (which you must include in your `Node` class) compiles and runs *exactly as provided*:

```
@Override
public String toString() {
    return toString(0);
}

// based on what toString() does, think about what 'elements' and 'children' can be
private String toString(int depth) {
    StringBuilder builder         = new StringBuilder();
    String        blankPrefix     = new String(new char[depth]).replace("\0", "\t");
    List<String>  printedElements = new LinkedList<>();
    for (E e : elements) printedElements.add(e.toString());
    String eString = String.join(" :: ", printedElements);
    builder.append(blankPrefix).append(eString).append("\n");
    children.forEach(c -> builder.append(c.toString(depth + 1)));
    return builder.toString();
}
```

1. Your `Node` class must have a method defined as follows, in order to determine if a node is a leaf node. (3)

   ```
   /**
    * @return <code>true</code> if and only if this node is a leaf node.
    */
   public boolean isLeaf()
   ```

2. Your `Node` class must have a method defined as follows, in order to determine if a node is a full node. (3)

   ```
   /**
    * @return <code>true</code> if and only if this node is a full node, i.e., it has the maximum
    *         possible number of elements permitted by the minimum degree of the B-tree.
    */
   public boolean isFull()
   ```

Your primary task is to implement the `BTree` data structure. The class signature must be as follows:

```
public class BTree<E extends Comparable<E>> implements AbstractBTree<E>
```

3. This class must have a public constructor with the following signature: (4)

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

   A call to the above constructor should create an empty B-tree with the specified minimum degree. Recall from the lectures that the minimum degree of a B-tree is the integer `MIN` such that any non-root node must have at least `MIN - 1` values and at most `2 * MIN - 1` values.

To form a non-empty tree, you will need to implement the insertion algorithm discussed in the lectures. This algorithm must be used when a user calls the `add(E element)` method in your `BTree` class. You can use as many helper methods as you like, which may be called from the public `add(E element)` method. These additional methods, however, must be private. The rubric for the insertion algorithm's implementation is divided into many sub-parts, as follows:

4. Your program is able to form a B-tree of a single node (the root node), with the number of elements never exceeding the upper bound of `2 * MIN - 1`. (5)

5. Your program is able to form a B-tree where, upon attempting to add an element to a tree with a single full node (i.e., the root node is full, and it is the only node in the tree), the tree gets modified properly and successfully adds the new element. (10)

6. If a B-tree has a root node with more than one child, a new element can be added successfully and in its proper location, when none of the nodes in the tree are full. (10)

7. When the attempt to add a new element leads to a full leaf node, the tree is modified properly as per the insertion algorithm, and the new element is added in its correct place after the tree is modified. (10)

8. Your program properly handles the situation where a new element being added will be added to a leaf node, but another node in the path leading to that leaf node is full. (Recall that the insertion algorithm always starts at the root, and there is always a unique path from the root to the node in which an element will be inserted.) (10)

9. Your program can handle *any* data type being the elements of the B-tree, as long as the data type provides a natural order. (10)

10. Your program correctly splits a B-tree of depth more than 2 (i.e., the leaf nodes are grandchildren of the root node, or even further down), if required by the insertion of a new element. (10)

11. Your program does not allow addition of duplicate values, and fails "silently" as required by the given interface documentation. (5)

It is VERY important to note that all trees will be verified based on their human-readable string representation. This is why the `toString()` method was provided for the `Node` class. The idea is that if you call `toString()` on the root node, the entire tree will be printed in the intended format. For this to happen, you must include the following in your `BTree.java`:

```
@Override
public String toString() {
    return root.toString();
}
```

This code must compile and work *exactly as provided*.

12. Your next task is to implement the B-tree search algorithm. Compared to the insertion algorithm, this is relatively simpler. The search algorithm must be as discussed in the lectures, and take $\theta(\log_k N)$ time for a B-tree of N elements with branching factor k. (20)

If you have successfully implemented the insertion and search algorithms, and your code is compatible with the `toString()` method codes given to you here, the following driver method can be used for testing.

```
@SafeVarargs
private static <T extends Comparable<T>> void addAllInThisOrder(BTree<T> theTree, T... items) {
    for (T item : items)
        theTree.add(item);
}

public static void main(String[] args) {
    BTree<Integer> integerBTree = new BTree<>(3);
    addAllInThisOrder(integerBTree, 10, 20, 30, 40, 50);
    System.out.println(integerBTree);

    integerBTree.add(60);
    integerBTree.add(70);
    integerBTree.add(80);
    integerBTree.add(90);
    System.out.println(integerBTree);

    NodeIndexPair<Integer> foundNumber = integerBTree.contains(80);
    System.out.printf("Element %d found at index %d of the following node:\n%s%n",
                          80,
                          foundNumber.index,
                          foundNumber.nodeLocation);
}

/* Correct output below: */
10 :: 20 :: 30 :: 40 :: 50

30 :: 60
        10 :: 20
        40 :: 50
        70 :: 80 :: 90

Element 80 found at index 1 of the following node:
70 :: 80 :: 90
```

- Please keep in mind these homework-related points mentioned in the syllabus.
- **What to submit?** The complete codebase (including classes and interfaces that were already given to you, in a package called "cse214hw3") as a single `.zip` file. Your zip file, once extracted, must not contain any further packages or subfolders.
  Deviations from the expected submission format carries varying degrees of score penalty (depending on the amount of deviation).

Submission Deadline: Nov 23 (Wednesday), 11:59 pm