COMP 2140 Assignment 5: Leaf-based 2-3 Trees

Robert Guderian and Helen Cameron

Due: Friday, December 7, 2018 at 4:30 p.m.

Hand-in Instructions

Go to COMP2140 in UM Learn; then, under "Assessments" in the navigation bar at the top, click on "Assignments". You will find an assignment folder called "Assignment 5". Click the link and follow the instructions. Please note the following:

- Submit the ONE .java file that you are completing only. The .java file must contain all the source code that you wrote. The .java file must be renamed `A5<your last name><your first name>.java` (e.g., `A5CameronHelen.java`).
- Please do not submit anything else.
- We only accept homework submissions via UM Learn. Please DO NOT try to email your homework to the instructor or TA or markers — it will not be accepted.
- We reserve the right to refuse to grade the homework or to deduct marks if you do not follow instructions.
- **Assignments become late immediately after the posted due date and time.** Late assignments will be accepted up to 49 hours after that time, at a penalty of 2% per hour or portion thereof. After 49 hours, an assignment is worth 0 marks and will no longer be accepted.

How to Get Help

Your course instructor is helpful: For our office hours and email addresses, see the course website on UM Learn (on the right side of the front page).

For email, please remember to put "[COMP2140]" in the subject and use a meaningful subject, and to send from your UofM email account.

Course discussion groups on UM Learn: A discussion group for this assignment is available in the COMP 2140 course site on UM Learn (click on "Discussions" under "Communications"). Post questions and comments related to Assignment 5 there, and we will respond. Please do not post solutions, not even snippets of solutions there, or anywhere else. We strongly suggest that you read the assignment discussion group for helpful information.

Computer science help centre: The staff in the help centre can help you (but not give you assignment solutions!). See their website at http://www.cs.umanitoba.ca/undergraduate/computer-science-help-centre.php for location and hours. You can also email them at helpctr@cs.umanitoba.ca.

Programming Standards

When writing code for this course, follow the programming standards, available on this course's website on UM Learn. Failure to do so will result in the loss of marks.

Question

Remember: You will need to read this assignment many times to understand all the details of the code you need to write.

Goal: To compare the execution of insertions and searches in binary search trees (BSTs) and in leaf-based 2-3 trees.

Code you can use: You are permitted to use and modify the code your instructor gave you in class for BSTs and their nodes, and for leaf-based 2-3 trees and their nodes. You are NOT permitted to use code from any other source, nor should you show or give your code to any other student. Any other code you need for this assignment, you must write for yourself. We encourage you to write ALL the code for this assignment yourself, based on your understanding of BSTs and leaf-based 2-3 trees — you will learn the material much better if you write the code yourself.

This assignment is more like a lab than previous assignments were. File `A5.java` contains a nearly-complete program that reads in sequences of integers. For each sequence, the program times each of the following four steps:

- Inserting the sequence of integers into an empty binary search tree, in the order that the integers occur in the sequence.
- Then, searching for each of the integers in the sequence in the binary search tree.
- Inserting the sequence of integers into an empty leaf-based 2-3 tree, in the order that the integers occur in the sequence.
- Then, searching for each of the integers in the sequence in the leaf-based 2-3 tree.

After timing each of these steps, the program prints the number of elements in the sequence, the timing information and first 20 elements in each tree in sorted order (or the whole tree if there are no more than 20 elements), and then moves on to the next sequence.

You need to write the bodies for all the methods in the `BST` class. The headers for the methods and the BST node class are already provided — don't change them. You can add `private` helper methods to the BST class as needed.

You also need to write the bodies for methods in the `TwoThreeTree` class (details below).

The Nodes for the Leaf-Based 2-3 Tree

We have provided a `private TwoThreeNode` class inside the `TwoThreeTree` class. You may change or add constructors, but note that doing this may cause the debugging methods to stop working. You may NOT change any of the other already-written parts (details below). You are permitted to add more methods to this class as needed.

The instance members are already provided for the leaf-based 2-3 tree nodes. The instance members are all declared `public` so you do not have to use accessors and mutator methods in the leaf-based 2-3 tree class — although you can if you want to (and they are already provided for you, too). The instance members are the following:

- `int[] key;` contains the data item (an integer, in this application) if the node is a leaf or either one or two index values if the node is an interior node. This array is allocated to be size 1 for a leaf and size 2 for an interior node.
- `TwoThreeNode[] child;` is `null` (i.e., no array is allocated) if the node is a leaf because leaves have no children and a leaf never becomes an interior node. It is an array of size three for an interior node, since the interior node will have either two or three children.

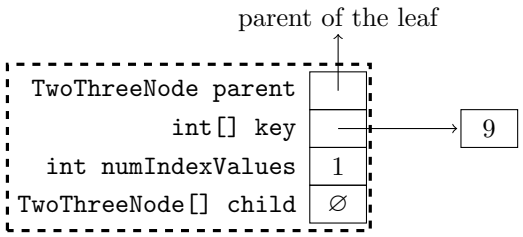
- `int numIndexValues;` is the number of index values stored in an interior node (either one or two). Furthermore, the number of children of an interior node is `numIndexValues+1`.

  When you split an interior node or add a new index value and a new child to an interior node you have to update `numIndexValues`.

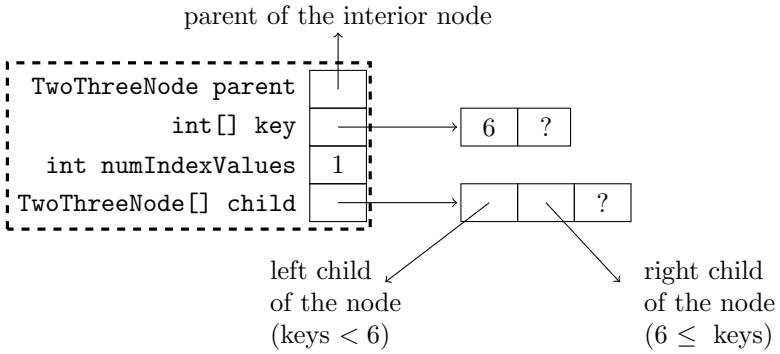
- `TwoThreeNode parent;` is a pointer to the parent of this node (to make the "split and push up" process in insertions easier). When you create a node, you have to set its `parent` pointer correctly. Watch out — when you split a node and move some children to the new node, you have to change the `parent` pointers of those children.

Remember: Only leaves store data items. Interior nodes contain index values that should only be used to guide a search to the correct leaf.

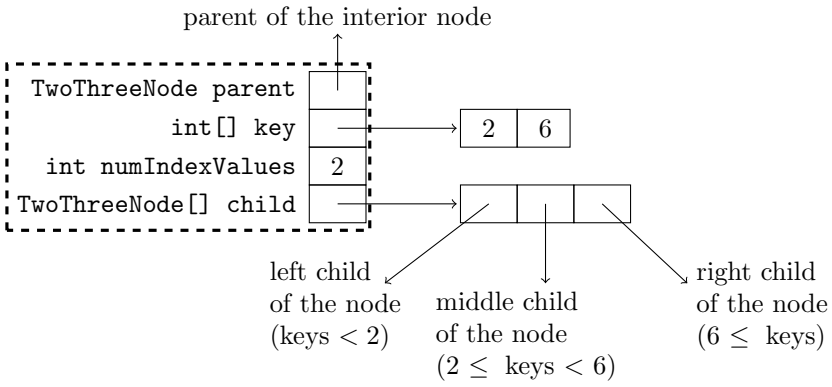
Node Constructor 1: The first `TwoThreeNode` constructor creates a leaf — it is passed a data item (an integer) and a pointer to the new leaf's parent. If we have a leaf containing the data item 9, for example, then it is implemented as follows:



Node Constructor 2: The second `TwoThreeNode` constructor creates an interior node with one index value and two children — it is passed an index value (an integer) and pointers to its parent, its left child and its right child. If we have such an interior node containing the index value 6, for example, then it is implemented as follows:



Of course, an interior node may later gain another index value and child during an insertion below it — if a sequence of split-and-push-up fixes reach all the way up to this node. If the above interior node receives another index value 2 and another child, for example, then it will become the following:



Notice that the index values are in sorted order, and so are the children.

Methods to test if the node is a leaf or an interior node:

- The method `isLeaf()` returns `true` if the calling node is a leaf and `false` otherwise.
- The method `isInteriorNode()` returns `true` if the calling node is an interior node and `false` otherwise.

Debugging methods: Also provided are two methods that might help you debug your leaf-based 2-3 tree implementation:

- Method `parentChildPointersOK()` checks node "this" and all its descendants (in a traversal). It returns `true` only if the children of each interior node points back at their parent with their `parent` pointers; otherwise, it prints a error message and returns `false`.
- Method `valuesOK` also checks node "this" and all its descendants (in a traversal). It returns `true` only if each index value `key[i]` in an interior node is:
  - Greater than the largest index value stored in the child to its left (`child[i]`),
  - Less than or equal to the smallest index value stored in the child to its right (`child[i+1]`),
  - Greater than the largest data item stored in any leaf descendant of the child to its left, and
  - Less than or equal to the smallest data item stored in any leaf descendant of its child to its right.

You can call either of these debugging methods directly on a `TwoThreeNode` to check just the calling node and its descendants. Alternatively, you can call method `treeOK()` in the `TwoThreeTree` class to check the entire calling tree — it calls both these methods on the root of the tree (see details below).

You are expected to use the `TwoThreeNode` class and its methods appropriately in your implementation of the leaf-based 2-3 tree (details below).

The Leaf-Based 2-3 Tree

In the `TwoThreeTree` class, we have already provided:

- The instance member, `root`, which is a pointer to the root node; and
- The constructor, which creates an empty tree; and
- The method `treeOK()`, which returns `true` only if the `TwoThreeNode` debugging methods `parentChildPointersOK()` and `valuesOK()` (described above) both return `true`.

You need to write the following method bodies. You are permitted to add other necessary methods to this class and the `TwoThreeNode` class, but you are NOT permitted to change any of the already-written code anywhere in the file.

- The `private` helper method `searchToLeaf()` is passed `searchKey`, the data item to search for. This method must return a pointer to the **leaf** where the search ends.

  This method does NOT check if the leaf contains `searchKey`. Its job is to correctly use `searchKey` to navigate down the tree to the leaf where `searchKey` would be stored if `searchKey` is present anywhere in the tree.

  If the tree is empty, this method should return null.

  This method should not change anything in the tree.

  This method should be used in some of the other methods you must write.

- The `public` method `search()` is passed `searchKey`, the data item to search for. This method must return `true` if the data item is in the tree, and `false` if it is not.

  This method should not change anything in the tree.

  Remember that data items are stored only in the leaves, so you must go all the way to a leaf to decide whether the data item exists in the tree or not.

- The `public` method `insert()` is passed a new data item, `newKey`, to be inserted into the 2-3 tree.

  This method should not insert `newKey` if it already exists in some leaf of the tree — no duplicates!

  If `newKey` does not exist in any leaf, then this method should insert it and perform any split-and-push-up fixes that are needed. The end result should be a valid 2-3 tree that contains all the data items it contained before, plus `newKey`.

  Hint: Write `private` helper methods to make this method (and any other) readable. Don't write huge blobs of code in one method.

- Write the `public` method `printInOrder()`, which prints some or all of the data items stored in the 2-3 tree. It is passed `numToPrint` and should print (in order) the first (smallest) `numToPrint` data items stored in the 2-3 tree. If there are fewer than `numToPrint` data items in the tree, simply print all of the data items in the tree.

  You will need to do an inorder traversal of the 2-3 tree to print the required number of data items.

  Remember that data items are stored only in the leaves — the index values in interior nodes should NOT be printed. So in a leaf, "visit the node" means "print the data item stored in this leaf", but in an interior node, "visit the node" means do nothing. In an interior node, you simply recursively do an inorder traversal of each of its children in turn.