The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 280– Intermediate Data Structures and Algorithms

Assignment 6

Date Due: March 13, 2018, 10:00pm

Total Marks: 33

1 Submission Instructions

- Assignments must be submitted using Moodle.
- Responses to written (non-programming) questions must be submitted in a PDF file, plain text file (`.txt`), Rich Text file (`.rtf`), or MS Word's `.doc` or `.docx` files. Digital images of handwritten pages are also acceptable, provided that they are **clearly** legible.
- Programs must be written in Java.
- If you are using IntelliJ (or similar development environment), do not submit the Module (project). Hand in only those files identified in Section 4. Export your `.java` source files from the workspace and submit only the `.java` files. **Compressed archives are not acceptable.**
- No late assignments will be accepted. See the course syllabus for the full late assignment policy for this class.

2 Your Tasks

Question 1 (33 points):

In `lib280-asn6` you are provided with a fully functional 2-3 tree class called `TwoThreeTree280`. Recall that 2-3 trees are keyed dictionaries. As such, the `TwoThreeTree280` class implements the `KeyedBasicDict280` interface. This interface adds the methods `obtain(k)`, `delete(k)` and `has(k)`, and `set(x)` (replace the item whose key matches they key of `x` with the item `x`).

Presently, `TwoThreeTree280` does not implement `KeyedDict280` which adds additional operations including all of the methods in `KeyedLinearIterator280` which, in turn, includes all of the public operations on a cursor. Note that `KeyedDict280` is the same interface that is implemented by `KeyedChainedHashTable280` so you should be somewhat familiar with it from the previous assignment.

The task for this question is to extend the `TwoThreeTree280` to a class called `IterableTwoThreeTree280` which allows linear iteration over the keyed data items stored in the two-three tree in ascending key-order. We will achieve this by adding additional references to leaf nodes so that the leaf nodes form a bi-linked list. Note that adding this feature to a 2-3 tree results in exactly a B+ tree of order 3 (see textbook Section 17.1). We aren't going to call it a B+ tree class though, because we are implementing specifically a B+ tree of order 3, and higher-order B+ trees will not be supported. Our `IterableTwoThreeTree280` class will be exactly a B+ tree of order 3.

Figure 1 in the Appendix shows the differences between a 2-3 tree (without iteration) and a B+ tree of order 3 containing the same elements, with the linking of the leaf nodes to support iteration. The algorithms for insertion and deletion are the same in both kinds of tree, except that in the case of the B+ tree, references to/from the predecessor and successor leaf nodes in key-order have to be adjusted to maintain the bi-linked list of leaf nodes.

The full class hierarchy of `IterableTwoThreeTree280` is shown in Figure 2 of the Appendix. The hierarchy of tree node classes is shown in Figure 3 of the Appendix.

To implement the `IterableTwoThreeTree280`, the following tasks must be carried out:

1. Make an extension of `LeafTwoThreeNode280` that adds references to its predecessor and successor leaf nodes. **This has already been done for you in the class** `LinkedLeafTwoThreeNode280`.
2. Override the `TwoThreeTree280::createNewLeafNode()` method by adding a new protected method in `IterableTwoThreeTree280` that it returns a new `LinkedLeafTwoThreeNode280` object instead of a `TwoThreeNode280` object. **This has already been done for you.**
3. In `IterableTwoThreeTree280`, override the `insert` and `delete` methods of `TwoThreeTree280` with modified versions that correctly maintain the additional predecessor and successor references in the `LinkedLeafTwoThreeNode280`. Each leaf node should always point to the the leaf node immediately to the left of it (the predecessor) and to the right of it (the successor) even if they are not siblings. Of course, the leaf node with the smallest key has no predecessor and the leaf node with the largest key has no successor.

   In `IterableTwoThreeTree280`, the `insert` and `delete` methods from `TwoThreeTree280` already have been copied, and TODO comments have been inserted indicating where you need to add additional code to maintain the additional leaf node references. The comments also provide a few hints. You should not have to modify any of the existing code for `insert` or `delete`, just add new code to deal with the linking and unlinking of leaf nodes from their successors and predecessors. Maintaining these links is very similar to inserting and removing nodes from the middle of a doubly-linked list.
4. Implement the additional methods required by `KeyedDict280` (and, by extension, `KeyedLinearIterator280`). Some of these have been done for you, others have not. TODO comments in `IterableTwoThreeTree280`

indicate which methods you need to implement and maybe even a hint or two. In this class, the linear iterator allows positioning of the cursor along the leaf-level of the tree.

5. In the `main()` function, write a regression test to test the methods required by `KeyedDict280` (and, by extension, `KeyedLinearIterator280`). You to not need to explicitly test the insertion and deletion methods since testing of the methods from `KeyedLinearIterator280` will reveal any problems with the new leaf node linkages, but you will need to insert and delete items to create test cases.

   You must test all of the methods listed in the interfaces that are coloured blue in Figure 2 of the Appendix.

   Use instances of the local class called `Loot`, which has been defined in the main method, as the data items to insert into the tree for testing. This class implements the type of item depicted in Figure 1 in the Appendix consisting of the name of a magic item from a fantasy game, and its value in gold pieces. The item keys are the item names (strings).

Hint: The `toStringByLevel()` method you've been given prints not only the 2-3 tree's structure, but also displays current linear ordering of the nodes that results from following the successor links in the leaf nodes, beginning with the leftmost leaf node. This may be helpful for the debugging of step 2.

3 Files Provided

lib280-asn6: A copy of lib280 which includes:

- solutions to assignment 5;
- The`TwoThreeTree280` class and related node and position classes in the `lib280.tree` package for Question 1.
- Partially completed `IterableTwoThreeTree280` class in the in `lib280.tree` package for Question 1.

4 What to Hand In

IterableTwoThreeTree280.java: Your completed tree for Question 1.

Appendix

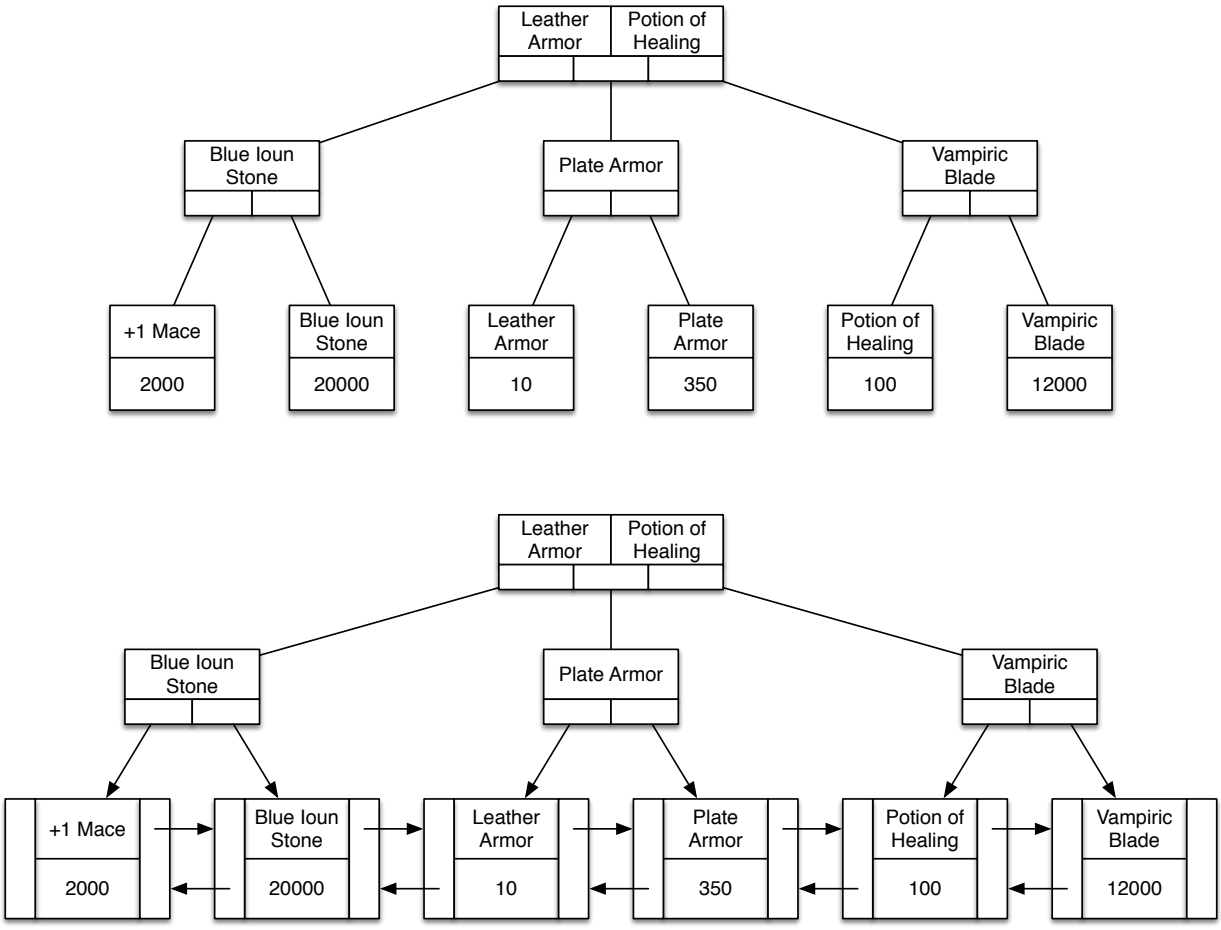

Figure 1: Top: a 2-3 tree which does not support a linear iterator; Bottom: a B+ tree of order 3 containing the same elements. Here the keys are strings (describing magical items in a fantasy game world) and the keyed data items contain the item name and an integer (representing the value, in gold pieces, of the object). Note that the trees are the same except for the extra linkages of the leaf nodes.

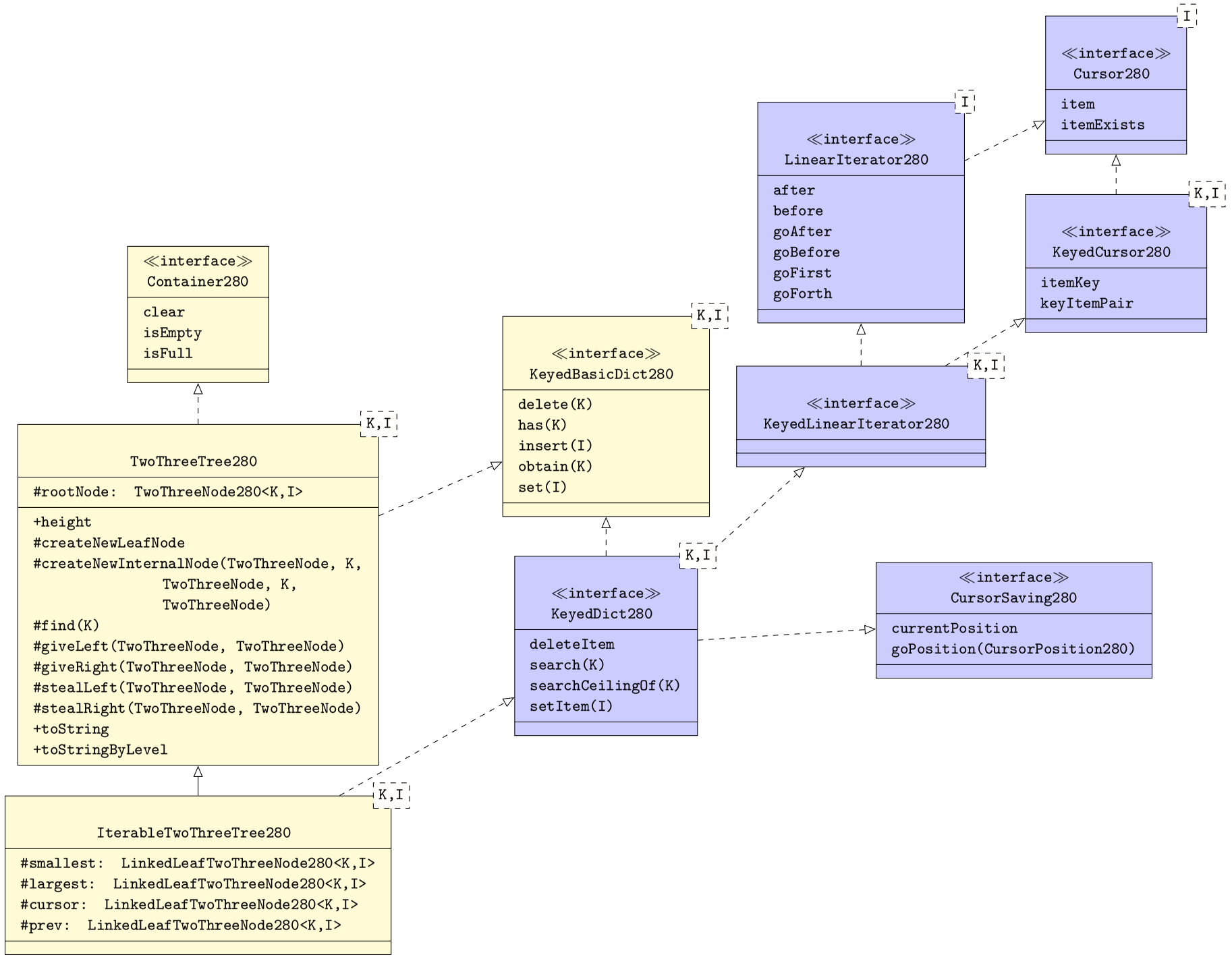

Figure 2: Class hierarchy for `IterableTwoThreeNode280`. For methods, only type names of parameters are shown.

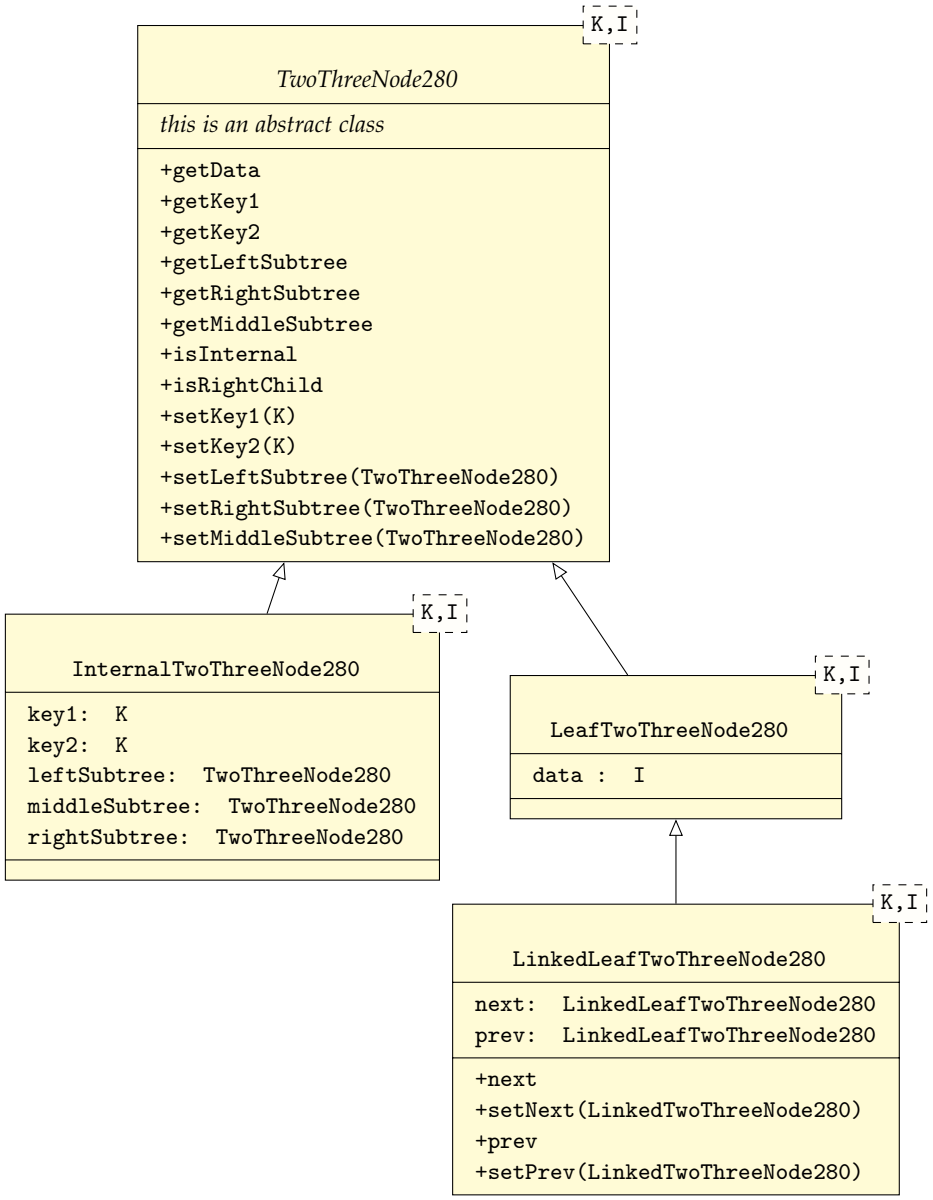

Figure 3: UML Class Hierarchy for 2-3 Tree Nodes in lib280. Every method that might be needed for either an internal or a leaf node is defined in the common abstract ancestor class `TwoThreeTree280` (note: because it is abstract, it cannot be instantiated). Subclasses `InternalTwoThreeNode280` and `LeafTwoThreeNode280` contain the data needed for the respective types of nodes, and definitions of each method appropriate to that type of node. Inherited methods that don't make sense for a particular type of node (e.g. `getData()` on an internal node) are defined to throw exceptions. The actual type of a reference to a TwoThreeNode can be determined by calling `isInternal` which is defined by internal nodes to return true and is defined by leaf nodes to return false. The `LinkedLeafTwoThreeNode280` extends the leaf node class to add predecessor and successor references to maintain the bi-linked list of leaf nodes in the B+ tree of order 3.