The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 280– Intermediate Data Structures and Algorithms

Assignment 5

Date Due: August 8, 2022, 11:59pm

Total Marks: 63

General Instructions

- Assignments must be submitted using Canvas.
- Programs must be written in Java.
- VERY IMPORTANT: Canvas is very fragile when it comes to submitting multiple files. We insist that you package all of the files for all questions for the entire assignment into a ZIP archive file. This can be done with a feature built into the Windows explorer (Windows), or with the zip terminal command (LINUX and Mac). We cannot accept any other archive formats. This means no tar, no gzip, no 7zip, no RAR. Non-zip archives will not be graded. We will not grade assignments if these submission instructions are not followed.

1 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 **??** 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 in the tree. 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 interface allows positioning of the cursor along the leaf-level of the tree. The cursor can never be positioned at an internal node.

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. This is because you will need to insert and delete items to create test cases for those methods in `KeyedLinearIterator280`
   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.

Question 2 (30 points):

For this question you will be implementing a k-D tree. We begin with introducing some algorithms that you will need. Then we will present what you must do.

Helper Algorithms for Implementing k-dimensional Trees

As we saw in class, in order to build a k-D tree we need to be able to find the median of a set of elements efficiently. The "j-th smallest element" algorithm will do this for us. If we have an array of n elements, then finding the $n/2$-smallest element is the same as finding the median.

Below is a version of the j-th smallest element algorithm that operates on a subarray of an array specified by offsets *left* and *right* (inclusive). It places at offset j (where $left \leq j \leq right$) the element that belongs at offset j if the subarray were sorted. Moreover, all of the elements in the subarray smaller than that belonging at offset j are placed between offsets *left* and $j-1$ and all of the elements in the subarray larger than that element are placed between offsets $j+1$ and *right*, but there is no guarantee on the ordering of any of these elements! The only element guaranteed to be in its sorted position is the one that belongs at offset j. Thus, if we want to find the median element of a subarray of the array `list` bounded by offsets *left* and *right*, we can call

```
jSmallest(list, left, right, (left+right)/2)
```

The offset $(left + right)/2$ (integer division!) is always the element in the middle of the subarray between offsets *left* and *right* because the average of two numbers is always equal to the number halfway in between them.

The j-smallest algorithm is presented in its entirety on the next page.

```
Algorithm jSmallest(list, left, right, j)
   list - array of comparable elements
   left - offset of start of subarray for which we want the median element
   right - offset of end of subarray for which we want the median element
   j - we want to find the element that belongs at array index j
   To find the median of the subarray between array indices 'left' and 'right',
   pass in j = (right+left)/2.

   Precondition: left <= j <= right
   Precondition: all elements in 'list' are unique (things get messy otherwise!)
   Postcondition: the element x that belongs at offset j, if the subarray were
                  sorted, is at offset j.  Elements in the subarray
                  smaller than x are to the left of offset j and the
                  elements in the subarray larger than x are to the right
                  of offset j.

   if( right > left )
      // Partition the subarray using the last element, list[right], as a pivot.
      // The index of the pivot after partitioning is returned.
      // This is exactly the same partition algorithm used by quicksort.
      pivotIndex := partition(list, left, right)

      // If the pivotIndex is equal to j, then we found the j-th smallest
      // element and it is in the right place!  Yay!

      // If the position j is smaller than the pivot index, we know that
      // the j-th smallest element must be between left, and pivotIndex-1, so
      // recursively look for the j-th smallest element in that subarray:
      if j < pivotIndex
            jSmallest(list, left, pivotIndex-1, j)

      // Otherwise, the position j must be larger than the pivotIndex,
      // so the j-th smallest element must be between pivotIndex+1 and right.
      else if j > pivotIndex
            jSmallest(list, pivotIndex+1, right, j)

      // Otherwise, the pivot ended up at list[j], and the pivot *is* the
      // j-th smallest element and we're done.
```

Notice that there is nothing returned by `jSmallest`, rather, it is the postcondition that is important. The postcondition is simply that the element of the subarray specified by `left` and `right` that belongs at index j if the subarray were sorted is placed at index j and that elements between $left$ and $j-1$ are smaller than the j-th smallest element and the elements between $j+1$ and *right* are larger than the j-th smallest element. There are no guarantees on ordering of the elements within these parts of the subarray except that they are smaller and larger than the the element at index j, respectively. *This means that if you invoke this algorithm with $j = (right + left)/2$ then you will end up with the median element in the median position of the subarray, all smaller elements to its left (though unordered) and all larger elements to its right (though unordered), which is just what you need to implement the tree-building algorithm!*

NOTE: for this algorithm to work on arrays of `NDPoint280` objects you will need an additional parameter d that specifies which dimension (coordinate) of the points is to be used to compare points.

An advantage of making this algorithm operate on subarrays is that you can use it to build the k-d tree without using any additional storage — your input is just one array of `NDPoint280` objects and you can do all the work without any additional arrays — just work with the correct subarrays.

You may have noticed that `jSmallest` uses the `partition` algorithm partition the elements of the subarray using a pivot. The pseudocode for the `partition` algorithm used by the `jSmallest` algorithm is given below. Note that in your implementation, you will, again, need to add a parameter *d* to denote which dimension of the *n*-dimensional points should be used for comparison of `NDPoint280` objects.

```
// Partition a subarray using its last element as a pivot.
Algorithm  partition(list, left, right)
list - array of comparable elements
left - lower limit on subarray to be partitioned
right - upper limit on subarray to be partitioned
Precondition: all elements in 'list' are unique (things get messy otherwise!)
Postcondition: all elements smaller than the pivot appear in the leftmost
               part of the subarray, then the pivot element, followed by
               the elements larger than the pivot.  There is no guarantee
               about the ordering of the elements before and after the
               pivot.
returns the offset at which the pivot element ended up


pivot = list[right]

swapOffset = left
for i = left to right-1
   if( list[i] <= pivot )
      swap list[i] and list[swapOffset]
      swapOffset = swapOffset + 1

swap list[right] and list[swapOffset]
return swapOffset;   // return the offset where the pivot ended up
```

Algorithm for Building the Tree

An algorithm for building a *k*-d tree from a set of *k*-dimensional points is given below. It is slightly more detailed than the version given in the lecture slides. It uses the `jSmallest` algorithm presented above.

```
Algorithm kdtree (pointArray, left, right, int depth)
pointArray - array of k-dimensional points
left - offset of start of subarray from which to build a kd-tree
right - offset of end of subarray from which to build a kd-tree
depth - the current depth in the partially built tree - note that the root
        of a tree has depth 0 and the $k$ dimensions of the points
        are numbered 0 through k-1.

if the specified subarray of pointArray is empty
    return null;
else
    // Select axis based on depth so that axis cycles through all
    // valid values. (k is the dimensionality of the tree)
    d = depth mod k;
    medianOffset = (left+right)/2

    // Put the median element in the correct position
    // This call assumes you have added the dimension d parameter
    // to jSmallest as described earlier.
    jSmallest(pointArray, left, right, d, medianOffset)

    // Create node and construct subtrees
    node = a new kD-tree node
    node.item = pointArray[medianOffset]
    node.leftChild = kdtree(pointArray, left, medianOffet-1, depth+1);
    node.rightChild = kdtree(pointArray medianOffset+1, right, depth+1);
    return node;
```

Your Tasks

Implementing the *k*-D Tree

Implement a *k*-D tree. You **must** use the `NDPoint280` class provided in the `lib280.base` package of `lib280-asn6` to represent your *k*-dimensional points. **You must design and implement both a node class (`KDNode280.java`) and a tree class (`KDTree280.java`).** Other than specific instructions given in this question, the design of these classes is up to you. You may use as much or as little of lib280 as you think is appropriate. You'll be graded in the **actual** appropriateness of your choices. You should aim to make the class fit into lib280 and its hierarchy of data structures, but you should not force things by extending classes inappropriately. You may use whatever private/protected methods you deem necessary.

A portion of the marks for this question will be awarded for the design/modularity/style of the implementation of your class. A portion of the marks for this question will be awarded for acceptable inline and javadoc commenting.

Your *k*-D tree ADT must support the following operations:

- Construct a new (balanced) *k*-D tree from a set of *k*-dimensional points (it must work for *any* $k > 0$).
- Perform a range search: given a pair of points $(a_1, a_2, \ldots a_k)$ and $(b_1, b_2, \ldots, b_k)$, $a_i <= b_i$ for all $i = 1 \ldots k$, return all of the points $(c_1, c_2, \ldots, c_k)$ such that $a_1 \leq c_1 \leq b_1, a_2 \leq c_2 \leq b_2, \ldots, a_k \leq c_k \leq b_k$.

*Note: your tree does **not** have to have operations that insert or remove individual NDPoints.*

In addition, you should write a test program that demonstrates the correctness of your tree. The test program should consist of two parts:

1. Show that your class can correctly build a *k*-D tree from a set of points. For *k*=2, display the set of *k*-dimensional points that you used as input (use between 8 and 12 elements), followed by a graphical representation of the built tree (similar to the `toStringByLevel()` output in the trees we've done previously). Do this again for one other value of *k*, between 3 and 5 (your choice).
2. For the second of the two trees you displayed in part 1, perform at least three range searches. For each search, display the query range, execute the range search, and then display the list of points in the tree that were found to be in range. A sample test program output is given below.

Implementation and Debugging Strategy

In order to implement the tree-building algorithm `kdtree` (shown in pseudocode, above) you first need to implement `jSmallest` which, in turn requires `partition`. It is **strongly** suggested that you implement and thoroughly test `partition` before trying to implement `jSmallest`. In turn, throughly test `jSmallest` before you implement `kdtree`. If you don't do this, I can tell you from experience that it will be a nightmare to debug. You need to be sure that each algorithm is correct before implementing the algorithms that depend on it, otherwise, if you run into a bug it will be very hard to determine in which method in the chain of dependent methods the bug is occurring. This is a fundamental principle that is crucial to designing complex software systems. Make sure each piece is correct before relying on it later.

Keep in mind that the algorithms presented above as abstract pseudocode do not necessarily translate line-for-line into Java code. Likely much of the pseudocode you've seen up to this point *does* translate in a line-by-line fashion, but you need to get used to pseudocode that doesn't, as this is the entire point of using pseudocode, such that it is language independent and omits implementation details while still conveying what the algorithm is supposed to do.

Sample Output

```
Input 2D points:
(5.0, 2.0)
(9.0, 10.0)
(11.0, 1.0)
(4.0, 3.0)
(2.0, 12.0)
(3.0, 7.0)
(1.0, 5.0)

The 2D lib280.tree built from these points is:

          3: (9.0, 10.0)
     2: (5.0, 2.0)
          3: (11.0, 1.0)
1: (4.0, 3.0)
          3: (2.0, 12.0)
     2: (3.0, 7.0)
          3: (1.0, 5.0)
Input 3D points:
(1.0, 12.0, 0.0)
(18.0, 1.0, 2.0)
(2.0, 13.0, 16.0)
(7.0, 3.0, 3.0)
(3.0, 7.0, 5.0)
(16.0, 4.0, 4.0)
(4.0, 6.0, 1.0)
(5.0, 5.0, 17.0)

               4: (5.0, 5.0, 17.0)
          3: (16.0, 4.0, 4.0)
               4: -
     2: (7.0, 3.0, 3.0)
          3: (18.0, 1.0, 2.0)
1: (4.0, 6.0, 1.0)
          3: (2.0, 13.0, 16.0)
     2: (1.0, 12.0, 0.0)
          3: (3.0, 7.0, 5.0)
```

```
Looking for points between (0.0, 1.0, 0.0) and (4.0, 6.0, 3.0).
Found:
(4.0, 6.0, 1.0)

Looking for points between (0.0, 1.0, 0.0) and (8.0, 7.0, 4.0).
Found:
(7.0, 3.0, 3.0)
(4.0, 6.0, 1.0)

Looking for points between (0.0, 1.0, 0.0) and (17.0, 9.0, 10.0).
Found:
(16.0, 4.0, 4.0)
(7.0, 3.0, 3.0)
(3.0, 7.0, 5.0)
(4.0, 6.0, 1.0)
```

2 Files Provided

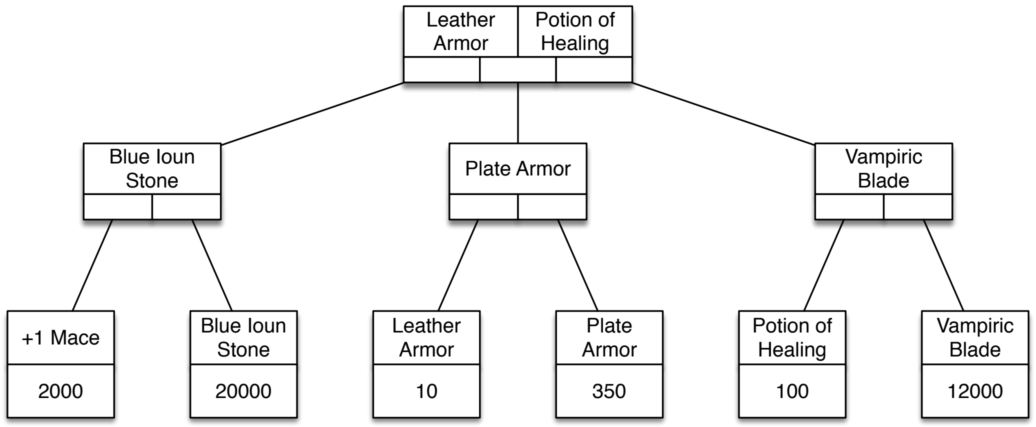
lib280-asn6: A copy of lib280 which includes:

- 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.
- the `NDPoint280` class in the `lib280.base` package for representing *n*-dimensional points for question 2;

3 What to Hand In

IterableTwoThreeTree280.java: Your completed tree for Question 1.
KDNode280.java: The node class for your *k*-D tree from Question 1.
KDTree280.java: Your *k*-D tree class for Question 1.
a7q1.txt/doc/pdf: The console output from your test program for question 1, cut and paste from the IntelliJ console window.

Appendix



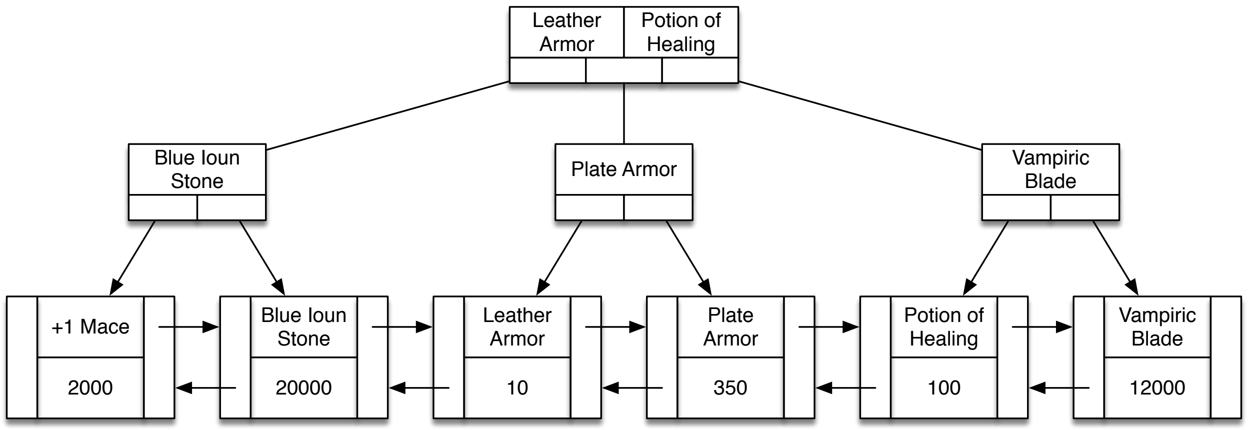


Figure 1: Top: a 2-3 tree which does not support a linear iterator; Bottom: a B+ tree of order 3 containing the same elements. Here the keys are strings (describing magical items in a fantasy game world) and the keyed data items contain the item name and an integer (representing the value, in gold pieces,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

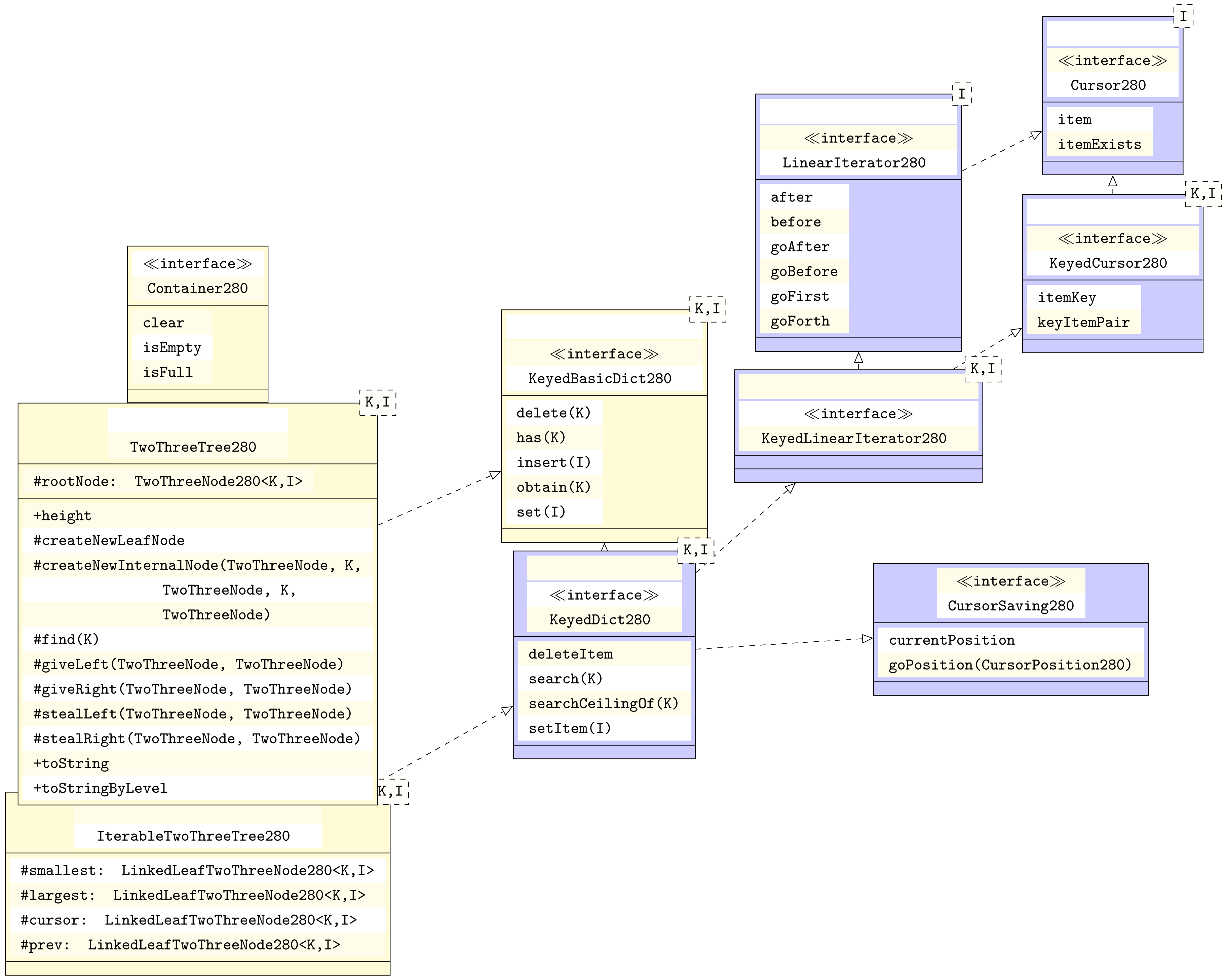


Figure 2: Class hierarchy for `IterableTwoThreeNode280`. For methods, only type names of parameters are shown.