CSE 12 Getting Started HW8 --- Binary Tree

Disclaimer

Most of the code if not all that is presented in this note should be interoperated as pseudocode!

Key Idea: Boxing

Review your notes if it is not clear to you.

But basically, you want to "pass in" an output parameter, thus, you would need to create an object to "box" the data field that it contains. Because the object is dynamically allocated, after the stack frame of a function call is destroyed, the box still persists in the memory. Thus, you still have access to this parameter that you passed in, thus it became an output parameter.

Some examples that you have already done: homework 5. You were returning `Long` instead of `long`.

Why do we need it in this PA?

We would like to use PointerInParent as an output parameter. But why?

Because the child is in charge of restructuring the tree. Therefore, the parent shall put this pointer that it is holding into a box and give it to its child, then the parent let its child to do its work, then the parent open the box and get the updated result of the pointer.

This pointer is either `left` or `right` that the parent is holding to its child, and after the update, it will point to the newly updated `left` or `right`. Again, the update comes from the fact that the child might restructure the nodes.

Driver.java

As usual.

Tree.java

Tree

Tree Ctor

Basically same as hw7

isEmpty ()

Determine a way to do it, and stick with it. (It would be a good practice to just call this method)

insert (Whatever *element*)

Special case: your tree is empty.

General case: Create a box of `root`, call `root.insert(element, box)` to perform (delegate) insertion. Then you unbox it by saying `root = box.pointer`.

lookup (Whatever *element*)

Special case: empty tree.

Delegate lookup to root.

remove (Whatever *element*)

Special case: empty tree

General case: Create a box of `root` and call `root.remove(element, box, false)`, don't forget to reassign the root to what is stored in your box after removal.

Note: The above code is bad practice, you should never name your box box, you should give it explicit names: rootBox

jettison ()

- Check whether the tree is empty or not
  - Empty:
    - Jettison the tree
  - Not empty:
    - Call `jettisoAllTnodes` on `root`, and jettison the tree.

TNode

TNode ()

Similar if not exact to hw7

jettisonTNodeAndData

Similar idea to hw5

jettisonTNodeOnly

Similar idea to hw5

jettisonAllTNodes

Go back to your hw7, instead of recurse on the input argument, you recurse on the caller

Insert (Whatever *element*, PointerBox *pointerInParentBox*)

This time it is a recursive method. You will recurse on the calling object.

Utilize the `.equals` and `.isLessThan` methods to determine which why to go.

Scenarios:

Duplicate insertion: Similar to what you shall do in hw7

Going left:

Check whether there is a left to go, if not, you should place a new TNode here.

If you have a left to go, you want to do the following things:

```
// Put the child pointer into a box
PointerBox leftBox = new PointerBox (left);
// Calling insert on the left
result = left.insert (element, leftBox);
// Update your left by the resulted pointer
left = leftBox.pointer;
```

Going right:

Same as left but replace left by right.

IMPORTANT: Make sure your do left.insert (element, leftBox) and right.insert (element, rightBox).

At the end, you need to call `setHeightAndBalance` with the `pointerInParent` box as the input .

The `SHAB` method will restructure the tree, thus, the pointer to the child in the parent might be changed after the restructure that is happening in the child. (child = this). That is why you would need to pass it in as an input parameter.

Return indicate whether the insert is succeed or not. (Should always be able to insert).

lookup (Whatever element)

This is again similar to insert. You recursively find the place and return the resulted data. Return `null` if you failed to find one. (Again, when will you fail to find one?)

remove (Whatever *element*, ParentBox *pointerInParentBox*, boolean *fromSHB*)

Similar to lookup, you will recursively find where is the element to remove.

If there is no such an element, you should return `null`.

If the to be removed element is found, then you need to do some works.

Scenarios:

Leaf:

```
pointerInParentBox.pointer = null;
```

One-child:

```
pointerInParentBox.pointer = existedChild; // assign the pointer to the child that exists
```

Think: One-child and leaf cases can be combined into one case if you like.

Two-child:

You need to call `replaceAndRemoveMin` in this case to maintain the binary tree structure, and you pass in the `targetTNode`. Think: What is the `targetTNode`?

Recall, `RARM` will go right once and keep going left to find the smallest element that is larger than the current one. The going right portion should happen in here, and going left will be recursively executed in `RARM`. You should write your own code to perform the going right step in remove.

Don't forget to reassign `right` by the resulted that you get from opening the box after you called `RARM`.

After your perform `RARM`, you might "corrupted" the height and balance of the tree, so you should call `SHAB` after you performed this step.

However, you shouldn't call it if the call is from `SHAB` because otherwise you will be doing an infinite recursion and ended up removing everything from the tree.

In all the three scenarios above, don't forget to `jettisonTNodeOnly` for the node that you removed.

And as you go back up the tree, don't forget to call `SHAB` to restructure the tree. Also think about when do you need to call this. If you just removed an item, do you need to call `SHAB`?

replaceAndRemoveMin (TNode *targetTNode*, ParentBox *pointerInParentBox*)

Recursively going left until you don't have a left child, set the `targetTNode` to hold the data of the current `TNode` that you are on, update the `pointerInParentBox.point` and jettison the current `TNode`.

Think: what is the new value that you should assign to your `pointerInParentBox.point`.

setHeightAndBalance (ParentBox *pointerInParentBox*)

This is the function that is used to maintain the efficiency of the binary tree data structure.

You first calculate height and balance.

Then if the absolute value of balance is greater than `THRESHOLD`, you shall remove the remove the current node and reinsert it back (make use of a box that you created). This is a key feature, so I will not provide code here, but you should draw pictures and think about what should be the actual process. Also, refer the the visualized videos!

General Notes:

This PA is known to be very difficult and hard to debug, so please **start early!!!!!!!!!**