Homework 12

November 30, 2018

1 2-3 Tree

Submit your source program as one file by the name `TwoThreeTree.scala`.

For this homework, you will implement a 2-3-tree algorithm in Scala. A 2-3 tree is a balanced binary search tree so that the time complexity of searching an object within the tree takes $O(log(N))$ time, where N is the number of objects in the tree.

The algorithm is described at this webpage: `https://en.wikipedia.org/wiki/2-3_tree`

The main idea of 2-3 tree algorithm is that it is a search tree with three types of nodes:

- a leaf node with no data or subtree.
- a 2-node with a left subtree, a data value x, and a right subtree, where all values in left subtree are less than x, which is less than all values in right subtree.
- a 3-node with a left subtree, a data value x_1, a middle subtree, a data value x_2, a right subtree. The data values in the subtrees are ordered the same way as those of 2-node.

During computation, you also use a 4-node that contains 4 subtrees and 3 data values.

When you search for a data key, you look up the 2-3 tree in the way similar to a binary search tree.

When you insert a data value e to a tree t, you have several cases.

- t is a leaf, you return a 2-node.
- t is a 2-node
  - if the subtrees of t are all leaves, you return a 3-node.
  - otherwise, recursively insert e to a subtree of t, which returns t_1.
    - if t_1 is a 4-node, then convert t into a 3-node (using 4 subtrees of t_1 to make two 2-nodes as subtrees).

    * otherwise, keep t_1 as a subtree of t.

- t is a 3-node
  - if the subtrees of t are all leaves, you return a 4-node.
  - otherwise, recursively insert e into a subtree of t, which returns t_1.
    * if t_1 is a 4-node, then convert t into a 4-node (using 4 subtrees of t_1 to make two 2-nodes as subtrees).
    * otherwise, keep t_1 as a subtree of t.

Note that if after insertion, the root of the tree a 4-node, you should convert the 4-node into a 2-node with two 2-nodes as subtrees.

2 Design

You will define a trait `Tree[X]` to represent the abstract definition of the tree. You will also define four case classes `Leaf`, `TwoNode`, `ThreeNode`, and `FourNode` to represent the leaf and the internal nodes of the tree. A leaf is an empty tree and has no data. A 2-node contains 2 subtrees and a data value. A 3-node contains 3 subtrees and 2 data values. A 4-node contains 4 subtrees and 3 data values. The 4-node is a temporary data holder for computation purpose and is never part of the tree.

The `Tree` trait has the following methods.

- `contains(e: X): Boolean` that returns true if and only if the tree contains the argument `e`. This method is abstract and must be implemented by subclasses.
- `ins(e: X): Tree[X]` that returns the tree node after inserting the argument `e`. This method is abstract and must be implemented by the subclasses. This method may return a 4-node.
- `insert(e: X): Tree[X]` that returns the tree node after inserting the argument `e` by calling `ins(e)` and if it returns a 4-node, it converts it into a 2-node with two 2-nodes as subtrees.
- `height` that returns the height of the tree (which is the max height of the subtrees plus 1). This method is abstract and must be implemented by the subclasses.

Finally, you will define `toString` method in each case class to convert the tree to a string. You should print the leaf object as “L”. Note that the methods of `FourNode` should just throw exceptions since they are not supposed to be called. It is in fact a bad style to define a subclass `FourNode` that does not really belong to the `Tree` hierarchy but we keep it this way because it is closer to the algorithm as described.

3 Driver code

If you run the driver code below,

```
object TwoThreeTree {
  def main(arg: Array[String]) {
     val input = List(3, 4, 2, 10, 9, 1, 5, 6, 11, 12, 13, 14, 15).map(i=>OrderedInt(i))

     val t = input.foldLeft[Tree[OrderedInt]](Leaf())((l, i) => {
       val x = l.insert(i);
       println(x.height + " " + x);
       x
     })

     println(t.contains(OrderedInt(5)))
     println(t.contains(OrderedInt(14)))
     println(t.contains(OrderedInt(7)))
     println(t.contains(OrderedInt(17)))

     val input2 = List('a','c','d','g','e','z','r','k','l','p','y').map(i=>OrderedChar(i))

     val t2 = input2.foldLeft[Tree[OrderedChar]](Leaf())((l, i) => {
       val x = l.insert(i);
       println(x.height + " " + x);
       x
     })
     println(t2.contains(OrderedChar('d')))
     println(t2.contains(OrderedChar('z')))
     println(t2.contains(OrderedChar('m')))
     println(t2.contains(OrderedChar('s')))
  }
}
```

you should get the following output:

```
1 (L, 3, L)
1 (L, 3, L, 4, L)
2 ((L, 2, L), 3, (L, 4, L))
2 ((L, 2, L), 3, (L, 4, L, 10, L))
2 ((L, 2, L), 3, (L, 4, L), 9, (L, 10, L))
2 ((L, 1, L, 2, L), 3, (L, 4, L), 9, (L, 10, L))
2 ((L, 1, L, 2, L), 3, (L, 4, L, 5, L), 9, (L, 10, L))
3 (((L, 1, L, 2, L), 3, (L, 4, L)), 5, ((L, 6, L), 9, (L, 10, L)))
3 (((L, 1, L, 2, L), 3, (L, 4, L)), 5, ((L, 6, L), 9, (L, 10, L, 11, L)))
3 (((L, 1, L, 2, L), 3, (L, 4, L)), 5, ((L, 6, L), 9, (L, 10, L), 11, (L, 12, L)))
3 (((L, 1, L, 2, L), 3, (L, 4, L)), 5, ((L, 6, L), 9, (L, 10, L), 11, (L, 12, L, 13, L)))
3 (((L, 1, L, 2, L), 3, (L, 4, L)), 5, ((L, 6, L), 9, (L, 10, L)), 11, ((L, 12, L), 13, (L, 14, L)))
3 (((L, 1, L, 2, L), 3, (L, 4, L)), 5, ((L, 6, L), 9, (L, 10, L)), 11, ((L, 12, L), 13, (L, 14, L, 15, L)))
true
true
false
false
1 (L, a, L)
1 (L, a, L, c, L)
2 ((L, a, L), c, (L, d, L))
2 ((L, a, L), c, (L, d, L, g, L))
2 ((L, a, L), c, (L, d, L), e, (L, g, L))
```

```
2 ((L, a, L), c, (L, d, L), e, (L, g, L, z, L))
3 (((L, a, L), c, (L, d, L)), e, ((L, g, L), r, (L, z, L)))
3 (((L, a, L), c, (L, d, L)), e, ((L, g, L, k, L), r, (L, z, L)))
3 (((L, a, L), c, (L, d, L)), e, ((L, g, L), k, (L, l, L), r, (L, z, L)))
3 (((L, a, L), c, (L, d, L)), e, ((L, g, L), k, (L, l, L, p, L), r, (L, z, L)))
3 (((L, a, L), c, (L, d, L)), e, ((L, g, L), k, (L, l, L, p, L), r, (L, y, L, z, L)))
true
true
false
false
```

where the leaves are printed as `L` and a 2-node of the form `TwoNode(left, data, right)` is printed as `(left, data, right)`. The 3-node is printed similarly.

4 Type parameter and upper bound

Your trait `Tree` has a type parameter `X` with upper bound `Ordered[X]`. As a result, the type parameter `X` of all subclasses of `Tree` must have the upper bound `Ordered[X]` as well.

You should notice that the test code in the main method inserts `Int` to a tree but `Int` object in Scala is not a subtype of `Ordered[Int]`. I have provided a case class `OrderedInt` that wraps an Int and extends `Ordered[OrderedInt]` so that we can convert each `Int` value to a wrapped object `OrderedInt` before storing them in a tree. The provided case class `OrderedChar` is for similar purpose.