Solution of ESO207 Programming Assignment-2.1

Harshit Raj Shubhan R

October 24th 2021

1 About Team

- About Assignment

Name:	Programming Assignment 2.1
Due on:	October 25th 2021

- Team Member

Name:	Harshit Raj
Email:	harshitr20@iitk.ac.in
Roll:	200433

- Team Member

Name:	Shubhan R
Email:	shubhanr20@iitk.ac.in
Roll:	200971

- About Course

Course Code:	ESO207A
Name of course:	Data Structure & Algorithm
Offered by:	Prof. Anil Seth
Offered in:	Semester 2021-22-I

Assignment-2 is about implementing some operations of Union-Find-Split ADT used in algorithm to find largest common subsequence. Each set is represented as a 2-3 tree.

This file includes Pseudo codes and complexity analysis of the Algorithm. Working code has been submitted using C++ programming language.

2 Problem Statement

Instructions

Notation: Height of a tree T is denoted by $h(T)$. For a set $S, |S|$ stands for number of elements in S.
You are given $2-3$ trees T_1 and T_2, representing respectively finite sets S_1, S_2 of natural numbers. Further, it is given that for all $x \in S_1$ and for all $y \in S_2, x < y$.

(a) Write pseudo-code for the algorithm $Merge(T_1, T_2)$. $Merge(T_1, T_2)$ should return a $2-3$ tree representation of set $S_1 \cup S_2$. Your algorithm should take $O(h(T_1) + h(T_2))$ time. Justify time complexity of your algorithm.

(b) Implement your algorithm of part (a) as an executable function $Merge(T_1, T_2)$. You are not allowed to use any library function in this implementation.
To help you write this code, you may divide it into several smaller functions. A modular and hierarchical design is encouraged.

For allowing us to test your program easily, you need to design following two functions. You need to submit just the working programs for these functions. No pseudo code for these is required. You may also use library functions (for example queue data structure) for these programs.

- **fun** Extract(T).
  Input: T is a 2-3 tree.
  action of program: Extract(T) prints elements of the set represented by T is ascending order.

- **fun** MakeSingleton(x)
  Input: x is a number.
  Output: MakeSingleton(x) returns a 2-3 tree representing set $\{x\}$

Please test your program using following function Test.

```
T = MakeSingleton(1)
for i = 2 to 500
    T= Merge(T,MakeSingleton(i))

U = MakeSingleton(777)
for i = 778 to 1000
    U= Merge(U,MakeSingleton(i))
V=Merge(U,T)
Extract(V)
```

Executing Test() should output 1, 2, 3, ..., 500, 777, 778, ..., 1000.

3 Data Structure Used

In order to implement the 2-3 tree the following node was used. Each node has the following structure:

```
struct two3node {
    nodeType type
    int val         // only for leaf node—holds value
    int val1, val2  // it is min1, min2
    two3node *left, *mid, *right
}
```

The 2-3 tree we used here is just a pointer to the root node

3.1 Helper Functions

Following standard function were used during this algorithm

3.1.1 New Nodes

- `newNilNode()` creates and returns a new nil node.
- `newLeaf(v)` creates and returns a new leaf node with value v (aliased as `MakeSingleton`)
- `newTwoNode(rMin, lChild, rChild)` creates and returns a new two node with given parameters
- `newThreeNode(mMin, rMin, lChild, mChild, rChild)` creates and returns a new three node with given parameters

3.1.2 Height and Minimum value of Tree

`GetHeightAndMin(T)` returns height and value of minimum element of Tree. This is done by traversing to left child of each node until we reach the bottom most leaf node. The value of leaf node is minimum element of Tree and depth of that node is the height of tree.

3.1.3 Extract Set

`Extract(T)` prints elements of the set represented by T is ascending order. Implemented using level wise traversal algorithm with queue.

4 Tree Merge Algorithm

Algorithm for two 2-3 trees merge is mentioned in this section. During this we will be provided with two trees, all elements of second is strictly greater than first.

4.1 Pseudo code

`Merge` function returns the root of the merged tree. `InsertLeftNode` and `InsertRightNode` are two helper functions.

```
Merge(tree1, tree2) {
    if (tree1.type == nil) // when tree1 is empty tree
        return tree2
    if (tree2.type == nil) // when tree2 is empty tree
        return tree1

    (h1, min1) = GetHeightAndMin(tree1)
    (h2, min2) = GetHeightAndMin(tree2)

    if (h1 == h2)
        return newTwoNode(min2, tree1, tree2)
    else if (h1 > h2)
        // insert tree2 inside tree1
        (n1, n2, val) = InsertRightNode(tree1, tree2, h1, h2, min2);

        if (n2.type == nil) return n1;
        else return newTwoNode(val, n1, n2);
    else
        // insert tree1 inside tree2
        (n1, n2, val) = InsertLeftNode(tree1, tree2, h1, h2, min2);

        if (n2.type == nil) return n1;
        else return newTwoNode(val, n1, n2);
}
```

4.2 Helper Function for Merge

There are two helper function `InsertLeftNode` and `InsertRightNode` both in theory does exactly same thing. Both of them return two *trees* and a value, which is minimum element of second tree. To not create unused nodes, we have made a global nil node called `nilNode` which is used to return nil nodes in the function.

4.2.1 Insert Node to Left

```
InsertLeftNode(r1, r2, h1, h2, min2) {
    if (h1 == h2)
        return (r1, r2, min2)

    // call recursively till you are at same level

    (n1, n2, val) = InsertLeftNode(r1, r2.left, h1, h2 - 1, min2)
    //height of new r2 is h2-1

    if (n2.type == nil)
        r2.left = n1    // update left node
        return (r2, nilNode, 0)
    else
        if (r2.type == twonode)
            return (newThreeNode(val, r2.val1, n1, n2, r2.right),
                    nilNode, 0)
        else if (r2.type == threenode)
            return (newTwoNode(val, n1, n2),
                    newTwoNode(r2.val2, r2.mid, r2.right),
                    r2.val1)

    return (nilNode, nilNode, 0)
}
```

4.2.2 Insert Node to Right

```
InsertRightNode(r1, r2, h1, h2, min2) {
    if (h1 == h2)
```

```
        return (r1, r2, min2)

    (n1, n2, val) = InsertRightNode(r1.right, r2, h1 - 1, h2, min2)
    //height of new r1 is h1-1

    if (n2.type == nil)
        r1.right = n1
        return (r1, nilNode, 0)
    else
        if (r1.type == twonode)
            return (newThreeNode(r1.val1, val, r1.left, n1, n2),
                    nilNode, 0)
        else if (r1.type == threenode)
            return (newTwoNode(r1.val1, r1.left, r1.mid),
                    newTwoNode(val, n1, n2),
                    r1.val2)

    return (nilNode, nilNode, 0)
}
```

4.3 Mathematical Completeness

Our algorithm goes along the lines of `InsertRoot` function explained in the lectures.

- If $h1 > h2$, we invoke `InsertRightNode` which finds the right most node of $tree1$ at appropriate height $(h2 + 1)$ and inserts $tree2$ there.
- If $h1 < h2$, we use `InsertLeftNode` which finds the left most node of $tree2$ at appropriate height $(h1 + 1)$ and inserts $tree1$ there.
- If $h1 == h2$, we can not insert one tree into the other, so we create a new two node with $tree1$ and $tree2$ as the left and right child respectively.

and return it.
In `InsertRightNode`, we keep track of what happens to a node - whether it remains as an intact node or splits into two. Here we use the words node and tree interchangeably as every node is a tree. We keep recursively calling till we reach the right most node of tree1 at height h2. Each iteration of this function returns 3 values `(node1, node2, val)` where *val* is minimum element in tree of $node2$.

- If the node in current iteration doesn't split and is just modified, it returns `(modifiedNode, nilNode, -)`.
- If is splits, it returns `(leftSplitNode, rightSplitNode, min_of_rightSplitNode)`.
- The *base case* is when both the trees are at same height and it returns `(node of tree1 at height h2, root of tree2, minimum value of tree2)`.

Similarly for the explanation of `InsertLeftNode`, where we recursively call till the left most node of $tree2$ at height h1 and return appropriate nodes.

4.4 Complexity

Calculation of complexity of merge function is as follows:

$$O(Merge) = O(1) + O(GetHeightAndMin(tree1)) + O(GetHeightAndMin(tree2)) + \\ O(InsertLeftNode) \text{ or } O(InsertRightNode))$$

To get Height and Minimum element we just have to traverse from root to leaf once which will take $O(h(tree))$ time.

Complexity of `InsertLeftNode` is $O(recursionDepth)$ and recursion base case is achieved when height of both parameters matches. Hence, the recursion depth is difference in height of trees. This implies $O(InsertLeftNode) = O(|h1 - h2|)$

Similarly, Complexity of `InsertRightNode` is $O(recursionDepth) = O(|h1 - h2|)$

$$\implies O(Merge) = O(h1) + O(h2) + O(|h1 - h2|)$$

$$\implies O(Merge) = O(h1 + h2)$$

Final complexity of the algorithm is $O(h1 + h2)$, which is to be expected and is also indicated in the problem statement.

5 Footnotes

- Entire code of this assignment could be found here.
- The programming was done in C++ by implementing 2-3 tree.