ESO207 Programming Assignment-2.1

Suyash Srivastava (201031) and Mohan Krishna (200590)

Hackerrank ID: sumokrisri

25^{th} October, 2021

1 Merging two trees in $O(h(T_1) + h(T_2))$

In our program, 2-3 Trees are represented by the structure TREE, which consists of a node pointer ROOT (the root of the tree) and TREE pointers Left, Middle, Right (the left, middle and right children subtrees of the tree). We call trees with NULL *parent* field, proper trees. Throughout our program, a 2-3 Tree with only two child subtrees has right child NULL. A leaf has all 3 children subtrees (left) NULL.

Algorithm 1: Merging into the left 2-3 Tree (Helper function)

Input: 2-3 Trees, *ptr* and T_2, such that ptr is the rightmost subtree at height $h(T_2)$ of a proper tree T_1
Output: 2-3 Tree, T_3 = Merge (T_1, T_2)
(Let *Ancestor* denote $ptr \rightarrow parent$)

1. *[Degenerate Case]* If *ptr* is a proper tree, return a TREE with left child *ptr*, middle child T_2 and right child NULL.
2. *[If Ancestor has only two children]* Make T_2 the right child subtree of *Ancestor* and return T_1.
3. *[If Ancestor has three children]* Remove *ptr* from the children of *Ancestor*, Create a TREE T_{temp} with left child *ptr*, middle child T_2 and right child NULL, and return MergeIntoLeft (*Ancestor*, T_{temp}) .

Algorithm 2: Merging into the right 2-3 Tree (Helper function)

Input: 2-3 Trees, T_1 and ptr, such that ptr is the leftmost subtree at height $h(T_1)$ of a proper tree T_2
Output: 2-3 Tree, T_3 = Merge (T_1, T_2)
(Let *Ancestor* denote $ptr \rightarrow parent$)

1. *[Degenerate Case]* If ptr is a proper tree, return a TREE with left child T_1, middle child ptr and right child NULL.

2. *[If Ancestor has only two children]* Make T_1 the left child subtree of *Ancestor* and return T_2.

3. *[If Ancestor has three children]* Remove ptr from the children of *Ancestor*, Create a TREE T_{temp} with left child T_1, middle child ptr and right child NULL, and return MergeIntoLeft (T_{temp}, *Ancestor*).

Algorithm 3: Merging 2-3 Trees

Input: 2-3 Trees, T_1 and T_2, representing the Sets S_1 and S_2 respectively
Output: 2-3 Tree T_3 representing the set $S_1 \cup S_2$

1. *[Degenerate cases]* If T_1 is NULL, return T_2. If T_2 is Nil return T_1.

2. *[Calculate heights]* Calculate the heights h_1 and h_2 of T_1 and T_2 respectively.

3. *[Calling helper functions]*

   (a) *[If $h_1 \geq h_2$]* Starting from the root of T_1, we traverse downwards to the rightmost subtree ptr at depth $h_1 - h_2$.
   Return MergeIntoLeft (ptr, T_1).

   (b) *[If $h_1 < h_2$]* Starting from the root of T_2, we traverse downwards to the rightmost subtree ptr at depth $h_2 - h_1$.
   Return MergeIntoRight (T_1, ptr).

1.1 Pseudo Code:

min(ptr) returns minimum value in the TREE rooted at ptr.

height (ptr) returns height of tree rooted at ptr, works in $O(log(n))$ time where n is the number of nodes in the tree

CreateTwoTree ($ptr1$, $ptr2$) returns a Tree with left child subtree $ptr1$ and middle child subtree $ptr2$

trunk(ptr) traverses upwards from the root node of ptr till it reaches a node which has NULL parent field (i.e., the node which is the root of the tree of which ptr is a subtree)

```
MergeIntoLeft (ptr,Second)
{//Make Second a right-sibling of ptr
        ancestor = ptr.parent
        if (ptr is not root)
        {
            if (ancestor has 3 children)
            {
                Second = ancestor.right
                ancestor.right = nil
                return MergeIntoLeft (ancestor, CreateTwoTree
                (ptr , Second)) //insert recursively
            }
            //Ancestor has 2 children,
            //directly add second as right sibing
            ancestor.right = Second
            ancestor.root.rt = min (ptr)
            Second.parent = ancestor
            return trunk(ancestor)
        //returns the proper tree of which ancestor is a subtree
        }
        return CreateTwoTree (ptr, Second)
}
```

```
MergeIntoRight (First, ptr)
{//Make First a left sibling of ptr
        Ancestor = ptr.parent
        if (ptr is not root)
        {
            if (Ancestor has 3 children)
            {
                temp = Ancestor.left
                temp.parent = nil
                Ancestor.left = Ancestor.middle
                Ancestor.middle = Ancestor.right
                Ancestor.right = nil
    return MergeIntoRight (CreateTwoTree (First, Ancestor.left),
        Ancestor)
            //insert recursively
            }
            //Ancestor has 2 children add First as leftmost child.
            Ancestor.right =  Ancestor.middle
            Ancestor.middle = Ancestor.left
            Ancestor.left = First
            First.parent = Ancestor
            Ancestor.root.rt = Ancestor.root.mid
            Ancestor.root.mid = min (Ancestor.middle)
            return trunk( Ancestor)
        }
        return CreateTwoTree (First, ptr)
}
```

```
Merge (First, Second)
{//Merge such that the leaves corresponding to A (in resultant tree)
 //are on the left side of leaves corresponding to B
        if (First==nil) return Second;
        if (Second==nil) return First;
        hA = height (First)
        hB = height (Second)
        i=0
        if (hA < hB)
 //make root of "First" child of appropriate node in Second
        {
            ptr = Second
            while (i < hB - hA)
             {
                 i=i+1
                 ptr = (ptr.left)
             }
            return MergeIntoRight (First, ptr)
        }
        ptr = First //Opposite
        while (i < hA - hB)
        {
            i=i+1
            if(ptr has 2 children)
             ptr=ptr.middle
            else
             ptr=ptr.right
        }
        return MergeIntoLeft (ptr, Second)
}
```

1.2 Runtime Analysis

Let input trees be T_1 and T_2.

In the degenerate cases of T_1 or T_2 being NULL, Merge takes O(1) time. Otherwise, **Merge** calls height() twice, and as **height(T)** take **O(h(T))** time, both these calls together take time **O(h(T_1) +h(T_2))**.

If height(T_2) is greater than height(T_1)then **MergeIntoRight** is called, else **MergeIntoLeft** is called.

The while loop in **Merge** (to traverse downwards) is executed $|h_2 - h_1|$ times. Hence, it takes O($|h_2 - h_1|$) time, where $|x|$ denotes the absolute value of x .

MergeIntoLeft and **MergeIntoRight** take constant time in the best case, which is when the node in larger tree where we have to insert the smaller tree (Make root of smaller tree a child of node of larger tree) has two children only. In the worst case, **MergeIntoLeft** or **MergeIntoRight** calls itself at most $|h_1 - h_2|$ times, this happens when, while traversing upwards, it always encounters nodes which already have two siblings(parent has 3 children), until it reaches the root of the tree.

Because there are no loops and **createTree** takes constant time, function at i^{th} recursive call takes time
t_i=t_{i+1} + c_{i+1}
where each c_i is bounded from above by a fixed constant, independent of the input.
In base case, there are two possibilities:
1) We create a new root, which takes constant time.
2) We insert one tree as a third child of a node in the other tree as described above. This takes O($|h_2 - h_1|$), because we call trunk which takes O($|h_2 - h_1|$) time.
We assume the worst case t_n:O($|h_2 - h_1|$)

Therefore, the total time taken in **MergeIntoLeft** or **MergeIntoRight** is:
$t_0 = t_1 + c_1 = t_2 + c_1 + c_2 = ... = t_n + (c_1 + c_2 + ... + c_n)$
$\leq$ k * $|h_2 - h_1|$+$max(c_1, .., c_n) * |h_2 - h_1|$
$\leq$k*$|h_2 - h_1|$+c*$|h_2 - h_1|$=O($|h_2 - h_1|$)
c is a constant independent of the input.

Hence total time taken by **Merge** is bounded by time taken in executing **height()** twice, which is **O(h(T_1) + h(T_2))**.