ESO207_Assignment2

Jaya Gupta(200471) Pratyush Gupta(200717)

May 3, 2022

1 Question1 → Merge two 2-3 trees

1.1 Pseudo Code

```
function HeightMin(T)                    ▷ calculates the height and minimum element of the Tree T
    if T==leaf(v) then                                        ▷ if tree if leaf Node height=0
        return {0,v}
    end if
    {height,min}=HeightMin(T.firstchild)
     return {height+1,min}
end function



    \* r1,r2 root of respective trees
h1,h2 height of respective trees
min1,min2 minimum element of respective trees
returns{n1,n2,m} where n1 and n2 are 2-3 trees and if n2!=nil then m=min_element(n2)
it is given that elements(r1)<elements(r2) for all elements
* \

function sub_merge(r1,r2,h1,h2,min1,min2)

\* here we will traverse on the right side of tree1 as h1>h2 so tree2 will be
inserted at a given height level of tree1
* \

    if h1>h2 then                                     ▷ case when tree2 is smaller than tree1
        if h1 ≠ h2+1 then                ▷ when we dont reach the desired height for merging trees
            if r1 == twoNode(a,α,β) then                                          ▷ if twoNode
                {n1,n2,m}=sub_merge(β,r2,h1-1,h2,min1,min2)   ▷ h1 is decreased as we go down in tree1
                if n2 == nil then                   ▷ if n2==nil right child will be updated to n1
                    return (twoNode(a,α,n1),nil,0)
                else                                       ▷ if n2≠nil then make it a threeNode
                    return (threeNode(a,m,α,n1,n2),nil,0)
                end if
            end if

            if r1==threeNode(a,b,α,β,γ) then                                    ▷ if ThreeNode
                {n1,n2,m}=sub_merge(γ,r2,h1-1,h2,min1,min2)
                if n2==nil then                     ▷ if n2==nil right child will be updated to n1
                return (threeNode(a,b,$\alpha$,$\beta$,n1),nil,0)
            else                                  ▷ if n2≠nil then make it will be splitted into 2 twoNodes
                return (twoNode(a,$\alpha$,$\beta$),twoNode(m,n1,n2),b)
            end if
        end if
    end if

\* when we reach the desried height for insertion i.e. h1==h2+1
make r2 as the rightmost child of a twoNode
split a threeNode into twoNode and make r2 right child of second twoNode
* \

    if r1==twoNode(a,$\alpha$,$\beta$) then
        return (threeNode(a,min2,$\alpha$,$\beta$,r2),nil,0)
    end if
    if r1==threeNode(a,b,$\alpha$,$\beta$,$\gamma$) then
        return (twoNode(a,$\alpha$,$\beta$),twoNode(min2,$\gamma$,r2),b)
    end if
end if

   \* symmetric case when h2 > h1
in this case we only need to traverse on the left side of tree2 for merging tree1
* \

if h1<h2 then
    if h2 $\neq$ h1+1 then
        if r1 == twoNode(a,$\alpha$,$\beta$) then
            {n1,n2,m}=sub_merge(r1,$\alpha$,h1,h2-1,min1,min2)
            if n2 == nil then
                return (twoNode(a,n1,$\beta$),nil,0)
            else
                return (threeNode(m,a,n1,n2,$\beta$),nil,0)
            end if
        end if

        if r1==threeNode(a,b,$\alpha$,$\beta$,$\gamma$) then
            {n1,n2,m}=sub_merge(r1,$\alpha$,h1,h2-1,min1,min2)
            if n2==nil then
                return (threeNode(a,b,n1,$\beta$,n$\gamma$),nil,0)
            else
                return (twoNode(m,n1,n2),twoNode(b,$\beta$,$\gamma$),a)
            end if
        end if
    end if

    if r1==twoNode(a,$\alpha$,$\beta$) then
        return (threeNode(min2,a,r1,$\alpha$,$\beta$),nil,0)
    end if
    if r1==threeNode(a,b,$\alpha$,$\beta$,$\gamma$) then
        return (twoNode(min2,r1,$\alpha$),twoNode(b,$\beta$,$\gamma$),a)
    end if
end if
end function
```

```
function Merge(T1,T2)    ▷ takes trees to be merged and return root of merged tree
    if T1.root ==NULL then    ▷ if any or both of the tree are NULL
        return T2
    end if

    if T2.root == NULL then
        return T1
    end if

    {h1,min1}=HeightMin(T1.root)    ▷ calculate height and min element of each tree
    {h2,min2}=HeightMin(T2.root)

    if h1==h2 then    ▷ h1=h2,make a twoNode as parent(T1,T2) which will be new root
        return twoNode(min2,T1.root,T2.root)
    end if
{n1,n2,m}=sub_merge(T1.root,T2.root,h1,h2,min1,min2)
    if n2==nil then
        return n1
    else
        return twoNode(m,n1,n2)
    end if
end function
```

1.2 Time Complexity Analysis

- Height and minimum element of the tree is calculated by traversing on the left side of the given tree recursively until we reach the leaf node and increment the height at each level by one.

  Let Height_Min function is called on tree T of height h and the time it takes is T(h).
  So by recursive relation :
  $$T(h) = c + T(h-1)$$
  because we go down in the tree so height decreases by 1 on each recursive call.
  Further continuing in the same manner, we get
  $$T(h) = c*h + T(0)$$

- The HeightMin(T) function clearly works in $\mathcal{O}(h(T))$ time.

- So the time required for computing the heights of the trees is $\mathcal{O}(h(T_1) + h(T_2))$.

- The sub_merge function only calls itself recursively when the difference between the heights provided is greater than 1. Hence the depth of recursion is $\mid h(T_1) - h(T_2) \mid$.

- If the heights of the trees are provided to the Merge function beforehand then the function operates in $\mathcal{O}(\mid h(T_1) - h(T_2) \mid)$ since the runtime of the statements other than the recursive calls can be bounded by a finite constant.

  Proof by induction on Tree T1 and T2 of height h1 and h2.
  $$\text{Let } \Delta h = \mid h(T_1) - h(T_2) \mid$$
  Let sub_merge function take time T(Δh).

  So by recursive relation we can write :
  $$T(\Delta h) = C1 + T(\Delta h - 1)$$
  beacause we go down in the tree until we reach the point where the height difference of trees is 1.

All other steps take constant time in sub_merge.
Continuing in the same way :

$$T(\Delta h) = C1(\Delta h - 1) + T(1)$$

where C1 and T(1) are constant
So overall time taken T' for execution of Merge function is

$$T' = c*h1 + c*h2 + 2*T(0) + C1(\Delta h - 1) + T(1)$$

for height calculation and sub_merge function.

Let us take case of h2 > h1 (other case is symmetrical)

$$T' = c*h1 + c*h2 + 2*T(0) + C1(h2 - h1 - 1) + T(1)$$

$$T' = (c + C1)*h1 + (c - C1) *h2 + 2*T(0) -C1 + T(1)$$

$$c2 = max((c + C1), (c - C1))$$

By clubbing the terms and setting appropriate bounds we get

$$T' = c2*(h1 + h2) + c3$$

Clearly the time taken by Merge function is $\mathcal{O}(h(T_1) + h(T_2))$.

- However in this case, the runtime is $\mathcal{O}(h(T_1) + h(T_2))$.

2 Question 2$\rightarrow$ Split and Repair

2.1 PseudoCode

```
procedure Repair($T, height, left_side$)
    if left_side then
        if T == leaf(v) then
            return (T,1)
        end if
        if T.children == 0 then
            return (nil,0)
        end if
        if T.children == 1 then
            return Repair(T.child1,height-1,left_side)
        end if
        if T == twonode(a,$\alpha$,$\beta$) then
            if $\alpha$ == leaf(v) then
                return (T,2)
            end if
            let trial $\leftarrow$ Repair($\beta$,height-1,left_side)
            return Merge($\alpha$,trial.T,height-1,trial.h,a)
        end if
        if T == threenode(a,b,$\alpha$,$\beta$,$\gamma$) then
            if $\alpha$ == leaf(v) then
                return (T,2)
            end if
            let trial $\leftarrow$ Repair($\gamma$,height-1,left_side)
            T.children $\leftarrow$ 2
            return (Merge(T,trial.T,height,trial.h,b),height)
        end if
    end if
    if not left_side then
        if T == leaf(v) then
            return (T,1)
        end if
        if T.children==0 then
            return (nil,0)
        end if
        if T.children == 1 then
            return Repair(T.child1,height-1,left_side)
        end if
        if T == twonode(a,$\alpha$,$\beta$) then
            if $\beta$==leaf(v) then
                return (T,2)
            end if
            let trial $\leftarrow$ Repair($\alpha$,height-1,left_side)
            return Merge(trial.T,$\beta$,trial.h,height-1,a)
        end if
        if T == threenode(a,b,$\alpha$,$\beta$,$\gamma$) then
            if $\beta$ == leaf(v) then
                return (T,2)
            end if
            let trial $\leftarrow$ Repair($\alpha$,height-1,left_side)
            $\alpha \leftarrow \beta$
            $\beta \leftarrow \gamma$
            let num ← a
            a ← b
            **return** (Merge(trial.T,T,trial.h,height,num),height)
        **end if**
    **end if**
**end procedure**
**function** Split(T,x,T1,T2)
    (curr, currT1, currT2) ← (T, T1, T2)
    **while** curr.child1 $\neq$ leaf(v) **do**
        **if** curr == twonode(a,$\alpha$,$\beta$) **then**
            **if** x is less than a **then**
                currT1.children ← 1
                currT1.child1 ← $\alpha$
                currT2.child1 ← $\alpha$
                currT2.child2 ← $\beta$
                curr ← $\alpha$
                currT1 ← currT1.child1
                currT1 ← currT1.child1
            **else**
                currT2.children ← 1
                currT2.child1 ← $\beta$
                currT1.child1 ← $\alpha$
                currT1.child2 ← $\beta$
                curr ← $\beta$
                currT1 ← currT1.child2
                currT1 ← currT1.child1
            **end if**
        **end if**
    **end while**
**end function**
```

2.2 Time Complexity Analysis

We will show the time complexity for repairing the left tree after splitting is $\mathcal{O}(h(T))$. The proof for the right tree is similar.

- The following statement holds after splitting: *For every node which has children, it is only the rightmost child which is in need of repairing..* So this damaged *rightmost* path has some nodes which only have a single child.

- Let us define a *singleton* path as a continuous sub-sequence of the *rightmost* path such that all of the included nodes have a single child. Let there be k such paths with each path having a length of l_i.

- Whenever the merge function is called, the difference of the height of the arguments is :

$$\begin{cases} 0 \; or \; 1 & root \; of \; right \; tree \; has \; more \; than \; one \; child \\ l & otherwise \end{cases}$$

  where l is the length of the singleton path starting with the root of the right tree.

- We will show this using induction. The base case is when one the children is a leaf. The rightmost child may either be a leaf or *nil*. The height difference is 1 or 0 respectively. The base case, therefore , holds.

  - *Case* 1 : *node has more than one child*

    The node is produced by merging its children. Since it has more than one chil, its leftmost child is a 2-3 tree of height at least $h-1$, if the height of the node in consideration is h. After merging the children of the given node, the height of the 2-3 tree can be h or $h-1$. So the statement holds.

  - *Case* 2 : *node has one child* In this case, we keep going down the *rightmost* path till we encounter a node with more than one child or *nil*. The difference between the heights of the trees to be merged will be bounded by l , where l is the length of this singleton path.

- The Merge function will run in $\mathcal{O}(\mid h(T_1) - h(T_2) \mid)$ since the heights of the trees is provided as input in every call. For a singleton path of length l the runtime will be the sum of the time required for the recursive calls to repair and the call to merge.

- The depth of recursion and the height difference of the trees to be merged are both l. The merge function willl take time $\alpha.l$ and the traversal will take $\beta.l$ where α and β are constants. So runtime will be $\mathcal{O}(l)$.

- For the other case, the entire operation will take constant time since the height difference will be less than or equal to 1.

- IF there are m nodes in that path part of a singleton path, and n nodes which have more than one child, then the total runtime will be $\mathcal{O}(m)+\mathcal{O}(n)$, which is the same as $\mathcal{O}(m+n)$. Since $m+n = h(T)$, the runtime of the Repair function is $\mathcal{O}(h(T))$