ESO207A Programming Assignment-2

Akhil Agrawal 200076
Aditya Tanwar 200057

October 2021

Height of a tree T is denoted by $h(T)$. For a set S, $|S|$ stands for number of elements in S.

Q1 (Marks 15 + 20) 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 $\forall\, x \in S_1$ and $\forall\, y \in S_2$, $x < y$. Write pseudo-code to merge the two trees and justify time complexity analysis of the written algorithm.

pseudo-code

Algorithm 1: Height(T)

Data: T `/* A 2-3 tree */`
Result: Returns the height of T
$n \leftarrow T.root$
if $n == NULL$ **then**
  return 0
else
  $h \leftarrow 0$
  while $n \neq leaf(x)$ **do**
    $h \leftarrow h + 1$
    $n \leftarrow n.firstChild$
  end while
  return h
end if

Algorithm 2: Min(n)

Data: n `/* A node in a 2-3 tree */`

Result: The minimum value of any leaf contained in the sub-tree at n

$t \leftarrow n.root$
while $t \neq leaf(x)$ **do**
  $t \leftarrow t.firstChild$
end while
return x `/* The same` x `as in` $t \neq leaf(x)$ `*/`

Algorithm 3: MergeNodeLeft(n_1, n_2, Δh)

`/* Two nodes, with` n_1 `to be inserted on the left of the sub-tree rooted at` n_2 `and,` Δh `being the difference in the heights of the two nodes */`

Data: n_1, n_2, Δh

Result: Returns two nodes χ_1, χ_2 and a number x denoting the minimum in χ_2 if it is not $NULL$, and $-$ otherwise

if $\Delta h == 1$ **then**
  if $n_2 == twoNode(b,\ \alpha,\ \beta)$ **then**
    return $(threeNode(Min(n_2),\ b,\ n_1,\ \alpha,\ \beta),\ NULL,\ -)$
  else if $n_2 == threeNode(b,\ c,\ \alpha,\ \beta,\ \gamma)$ **then**
    return $(twoNode(Min(n_2),\ n_1,\ \alpha),\ twoNode(c,\ \beta,\ \gamma),\ b)$
  end if
else
  if $n_2 == twoNode(b,\ \alpha,\ \beta)$ **then**
    let $(\chi_1,\ \chi_2,\ x) = MergeNodeLeft(n_1,\ \alpha,\ \Delta h - 1)$
    if $\chi_2 == NULL$ **then**
      return $(twoNode(b,\ \chi_1,\ \beta),\ NULL,\ -)$
    else
      return $(threeNode(x,\ b,\ \chi_1,\ \chi_2,\ \beta),\ NULL,\ -)$
    end if
  else if $n_2 == threeNode(b,\ c,\ \alpha,\ \beta,\ \gamma)$ **then**
    let $(\chi_1,\ \chi_2,\ x) = MergeNodeLeft(n_1,\ \alpha,\ \Delta h - 1)$
    if $\chi_2 == NULL$ **then**
      return $(threeNode(b,\ c,\ \chi_1,\ \beta,\ \gamma),\ NULL,\ -)$
    else
      return $(twoNode(x,\ \chi_1,\ \chi_2),\ twoNode(c,\ \beta,\ \gamma),\ b)$
    end if
  end if
end if

Algorithm 4: MergeNodeRight(n_1, n_2, Δh)

```
/* Two nodes, with n2 to be inserted on the right of the sub-tree rooted
at n1 and, Δh being the difference in the heights of the two nodes */
```

Data: n_1, n_2, Δh

Result: Returns two nodes χ_1, χ_2 and a number x denoting the minimum in χ_2 if it is not $NULL$, and $-$ otherwise

if $\Delta h == 1$ **then**
  if $n_1 == twoNode(b,\ \alpha,\ \beta)$ **then**
    return $(threeNode(b,\ Min(n_2),\ \alpha, \beta,\ n_2),\ NULL,\ -)$
  else if $n_1 == threeNode(b,\ c,\ \alpha,\ \beta,\ \gamma)$ **then**
    return $(twoNode(b,\ \alpha,\ \beta),\ twoNode(Min(n_2),\ \gamma,\ n_2),\ c)$
  end if
else
  if $n_1 == twoNode(b,\ \alpha,\ \beta)$ **then**
    let $(\chi_1,\ \chi_2,\ x) = MergeNodeRight(\beta,\ n_2,\ \Delta h - 1)$
    if $\chi_2 == NULL$ **then**
      return $(twoNode(b,\ \alpha,\ \chi_1),\ NULL,\ -)$
    else
      return $(threeNode(b,\ x,\ \alpha,\ \chi_1,\ \chi_2),\ NULL,\ -)$
    end if
  else if $n_1 == threeNode(b,\ c,\ \alpha,\ \beta,\ \gamma)$ **then**
    let $(\chi_1,\ \chi_2,\ x) = MergeNodeRight(\gamma,\ n_2,\ \Delta h - 1)$
    if $\chi_2 == NULL$ **then**
      return $(threeNode(b,\ c,\ \alpha,\ \beta,\ \chi_1),\ NULL,\ -)$
    else
      return $(twoNode(b,\ \alpha,\ \beta),\ twoNode(x,\ \chi_1,\ \chi_2),\ c)$
    end if
  end if
end if

Algorithm 5: Merge(T_1, T_2)

```
/* Two 2-3 trees with each element in T_1 < each element in T_2 */
```

Data: T_1, T_2

Result: Returns the root of the tree obtained on merging T_1 and T_2

if $T_1 == NULL$ **then**
  return $T_2.root$
else if $T_2 == NULL$ **then**
  return $T_1.root$
else
  $n_1 \leftarrow T_1.root$, $n_2 \leftarrow T_2.root$
  if $Height(T_1) == Height(T_2)$ **then**
    return $twoNode(Min(n_2), n_1, n_2)$
  else if $Height(T_1) > Height(T_2)$ **then**
    let $(x_1, x_2, x) = MergeNodeRight(n_1, n_2, Height(T_1) - Height(T_2))$
    if $x_2 == NULL$ **then**
      return x_1
    else
      return $twoNode(x, x_1, x_2)$
    end if
  else
    /* $Height(T_1) < Height(T_2)$ */
    let $(x_1, x_2, x) = MergeNodeLeft(n_1, n_2, Height(T_2) - Height(T_1))$
    if $x_2 == NULL$ **then**
      return x_1
    else
      return $twoNode(x, x_1, x_2)$
    end if
  end if
end if

Complexity Analysis

Complexity of each function is presented below-

- **Height(T)**: Calculates the height of a `tree`by going down to a $leaf$ by repeatedly choosing the $first\,child$. A $leaf's$ height is taken to be 0. As it travels down the tree only once, its complexity is directly proportional to $h(T)$, and hence, it is in $\mathcal{O}(h(T))$.
- **Min(n)**: Calculates the minimum value stored in a `tree/ sub-tree` rooted at node n by repeatedly choosing the $first\,child$ (i.e., the $left$-$most\,child$), until it hits a $leaf$ node, when it simply returns the value stored in it. This function makes use of a property of `2-3 trees` which is that the leaves contain values in increasing order from left to right. Again, as it travels down the tree only once, its complexity is proportional to $h(T)$, and hence, it is in $\mathcal{O}(h(T))$.
- **MergeNodeLeft(n_1, n_2, Δh)**: Merges the `tree` rooted at n_1 on the left of n_2 at an appropriate height difference (Δh). The return statements are akin to that of **Insert** in the lectures.
  The candidate height difference is chosen to be 1, because at this difference in height, n_1 has the same height as the *children* of n_2, so it can be added as a child of n_2 at this difference. The results of this addition (of child) are propagated upwards to the root of the `tree` containing n_2 in a manner identical to **Insert** in the lectures.
  The function makes some assumptions which are listed below-
  - Each element stored in $n_1 <$ Each element stored in n_2
  - Height of $n_1 <$ Height of n_2
  - $n_1 \neq NULL$

  In essence, the function keeps choosing the $left$-$most\,child$ of n_2 until a certain Δh, carries some constant time instructions and **Min(n_2)** function at max a constant number of times, to insert n_1 as a child of n_2 at this Δh, then recursively propagates this insertion upto the root, making use of only constant time functions in its journey upwards. The function thus, traverses from the root of a `tree`to its left-most child at most a constant number of times. The time complexity can thus be summarised as follows-

$$\begin{aligned}
&Time(\text{Traversal to } n_2 \text{ for } \Delta h = 1) + Time(\text{Min}(n_2)) \\
&+ Time(\text{Insertion at } \Delta h = 1) \\
&+ Time(\text{Updation in the journey back to root}) \\
&\leq c_1 \cdot h(T_2) + c_2 \cdot h(T_2) + c_3 + c_4 \cdot h(T_2) \\
&\leq c_0 \cdot h(T_2) = \mathcal{O}(h(T_2))
\end{aligned}$$

- **MergeNodeRight(n_1, n_2, Δh)**: Similar to **MergeNodeLeft**, this function merges the `tree` rooted at n_2 to the right of `tree` rooted at n_1 at an appropriate height difference Δh. The function calls itself recursively on the right most child of n_1, each time decreasing Δh by 1, until a height difference of $\Delta h = 1$ is reached. This is taken as the base case for the recursion, as at this height, n_2 can be inserted as a child of n_1. The insertion process of a node is similar to that of **Insert** in the lectures. The function makes some assumptions which are listed below-

- Each element stored in $n_1 <$ Each element stored in n_2
- Height of $n_2 <$ Height of n_1
- $n_2 \neq NULL$

So the function reaches the height difference, $\Delta h = 1$, by using operations that take constant time at each height. At $\Delta h = 1$ it calculates the minimum value in the `tree` rooted at n_2. The results of this insertion are propagated up to the root of the `tree`in a manner similar to that of **Insert** in the lectures, while making use of only constant time instructions in the journey up. Thus total time taken can be summarised as:

$$\begin{aligned}
&Time(\text{Traversal to } n_1 \text{ for } \Delta h = 1) + Time(\text{Min}(n_2)) \\
&+ Time(\text{Insertion at } \Delta h = 1) \\
&+ Time(\text{Updation in the journey back to root}) \\
&\leq c_1 \cdot h(T_1) + c_2 \cdot h(T_2) + c_3 + c_4 \cdot h(T_1) \\
&\leq c_0 \cdot h(T_1) + c_1 \cdot h(T_2) = \mathcal{O}(h(T_1) + h(T_2))
\end{aligned}$$

- **Merge(T_1, T_2)**: Merges the `trees` T_1 & T_2, based on their heights. The possibilities that arise in doing so, and their corresponding time complexities are listed below-
  - If either of the trees are $NULL$, the other one is returned. This takes care of the assumptions of the functions it calls subsequently as well. $\mathcal{O}(1)$
  - If both of them have the same height, a new node is made, with roots of the trees as its children. To define the parameters of the new *root* node, **Min(T_2)** is called once. $\mathcal{O}(h(T_2))$
  - The only case that remains is when their heights are different, or in other words, when the difference in their heights is at least 1. In such a case, the tree with the lower height is inserted into the tree with the higher height. **Height(T)** is called to calculate their heights. According to which tree has the taller height, **MergeNodeLeft** or **MergeNodeRight** is called. The complexity is summarised below-

$$\begin{aligned}
&Time(\text{Height}(T_1) + Time(\text{Height}(T_2)) \\
&+ (Time(\text{MergeNodeLeft}) \text{ or } Time(\text{MergeNodeRight})) \\
&\leq a_0 \cdot h(T_1) + a_0 \cdot h(T_2) + ((a_1 \cdot h(T_2)) \text{ or } (a_2 \cdot h(T_1) + a_3 \cdot h(T_2)) \\
&\leq a \cdot (h(T_1) + h(T_2)) \\
&= \mathcal{O}(h(T_1) + h(T_2))
\end{aligned}$$

Concluding for this function, we can take an upper limit on all the possible cases to arrive at this time complexity-

$$\begin{aligned}
&k_1 + k_2 \cdot h(T_2) + k_3 \cdot (h(T_1) + h(T_2)) \\
&\leq k_0 \cdot (h(T_1) + h(T_2)) \\
&= \mathcal{O}(h(T_1) + h(T_2))
\end{aligned}$$

Note

- The convention used for representation of different kinds of nodes is the same as that in lectures, but mentioned here again for completeness-
  - leaf(x): The leaf stores the value x in it.
  - twoNode(min_{child_2}, $child_1$, $child_2$)
  - threeNode(min_{child_2}, min_{child_3}, $child_1$, $child_2$, $child_3$)
- It is assumed that the time taken to construct a $twoNode$ or a $threeNode$ with all the parameters available, takes constant time, as does checking if a node is a $twoNode$ or a $threeNode$.
- The **Time** notation used in complexity analysis simply denotes a generic dependence of the function on its parameters.
- The T_1 and T_2 introduced in complexity analysis of **MergeNodeLeft** and **MergeNodeRight** refer to the complete tree that contain the nodes n_1 and n_2 respectively.
- The actual code written and submitted has some subtle differences.
  Most of them are there in favour of reduced code size, and easier debugging and use, for example, writing a single function for **MergeNode** instead of separate ones for $left$ and $right$.
  Others are there due to constraints in language, for example, a variable of struct type was returned in some functions where multiple values had to be returned, as C++ does not allow multiple values to be returned at once.
  Regardless, the algorithm used and the resultant time complexity remain the exact same.

□□□