ESO207 Programming Assignment 2.1

Devansh Kumar Jha(200318) and Divyansh Gupta(200351)

2021–10-24

1 Data Structure Usage

Two-Three tree ADT controlled using a class **twth** and structure **twthnode**.

2 Strategy Used

Taking advantage of the fact that one of the trees has values strictly greater than the other we can design a merge algorithm to make a two three tree which displays the union of the two sets. Let's consider all values of T2 are greater than -

- **Height of trees are equal**
  A twonode is made with root of both trees as children and this is the root for the tree representing union of both sets.
- **Height of T1 is greater**
  An appropriate node X in the tree T1 (tree with greater height) is found and the root of tree T2 (tree with smaller height) is made children of it if possible otherwise a new node containing appropriate childrens is created and the same process is now done for parent of X. If we reach all the way to the root of T1 than procedure similar to first case is executed.
- **Height of T2 is greater**
  Similar procedure as the second case.

3 Structure Used

The structure used for the program **twthnode** is declared as follows -

Algorithm 1: Structure Declaration

```
struct twthnode {
        int type;
        /* 0 null 1 single 2 leaf 3 twonode 4 threenode          */
        int d1,d2;
        /* 2 values are stored and when not needed they are set to
    -1                                                           */
        struct twthnode* parent;
        struct twthnode* left;
        struct twthnode* middle;
        struct twthnode* right;
}
```

4 Pseudo Codes

Functions created for implementing the merge -

- **Merge(T1,T2)**
  Takes the two three trees T1 and T2 and returns a two three tree T which represents the union of these two trees.

- **insert(node,position,m,type)**
  Takes 4 quantities, the root of the tree to be added, the position of the node with appropriate height to add the tree, minimum value of interest(may be minimum of tree rooted at node or position) and a number to denote the category of union. It returns nothing just changes the two three tree it is working upon.

- **insert-part(node,position,m,type)**
  It is the main insert function which works recursively and performs the task as explained in the "Strategy Used" header. Returns either a triplet (n1,n2,m) where n1 and n2 represent root of trees and m is minimum value in tree n2 or NULL.

Algorithm 2: Merge(th1,th2)

Input : Two three trees th1 and th2

Output: A two three tree representing the Union of given trees

```
if th1 == NULL TREE then
    return th2
end if
if th2 == NULL TREE then
    return th1
end if
temp1 = th1.get()
temp2 = th2.get()
/* get() function on any two three tree returns the root of that
   tree.                                                          */
h1 = height of th1
h2 = height of th2
/* Heights are found by traversing the trees downwards till a leaf
   is encountered.                                                */
if h1 == h2 then
    th.root = twonode(th2.min(), temp1, temp2)
    return th /* min() function on any tree returns the minimum
       value stored in it by getting to the leftmost leaf.        */
else
    if h1 > h2 then
        position =
            node of tree th1 at height (h2 + 1) towards the rear end
            /* This is found using tree traversals in O(h) time.  */
        th1.insert(temp2, position, th2.min(), 2)
        /* "1" represents that the shorter tree is added towards the
           front.                                                 */
        /* "2" represents that the shorter tree is added towards the
           rear.                                                  */
        return th1
    else
        position =
          node of tree th2 at height (h1 + 1) towards the front end
        th2.insert(temp1, position, th2.min(), 1)
        return th2
    end if
end if
```

Algorithm 3: T.insert(node,position,m,type)

Input: Works for a two three tree T having a node position. node is the root of tree to be added, type shows the end at which addition takes place and m denotes the minimum value of tree at position for type=1 and minimum value of tree at node for type=2.

Output: It returns nothing, the changes made by this reflect for the tree T.

if $position == NULL$ **then**
  $position = node$
else
end if
if $position! = LEAF$ *and* $node! = NULL$ **then**
  $(n1, n2, min) = insert\_part(node, position, m, type)$
  if $n1! = NULL$ **then**
    $root = twonode(min, n1, n2)$
  else
  end if
else
end if
return

Algorithm 4: insert_part(node,pos,m,type)

Input: As in previous function

Output: Returns a triplet (n1,n2,m) where n1 and n2 are two nodes and m is the minimum value in tree rooted at n2.

```
if pos == twonode(a, α, β) then
    if type == 1 then
        pos = threenode(m, a, node, α, β)
        return (NULL,NULL,-1)
    else
        pos = threenode(a, m, α, β, node)
        return (NULL,NULL,-1)
    end if
else
end if
if pos == threenode(a, b, α, β, γ) then
    if type == 1 then
        k = twonode(m, node, α)
        pos = twonode(b, β, γ)
        if pos → parent == NULL then
            return (k,pos,a)
        else
            p = insert_part(k, pos → parent, a, type)
            return p
        end if
    else
        k = twonode(m, γ, node)
        pos = twonode(a, α, β)
        if pos → parent == NULL then
            return (pos,k,b)
        else
            p = insert_part(k, pos → parent, b, type)
            return p
        end if
    end if
else
end if
```

5 Runtime Complexity Analysis

- **insert_part**
  A single iteration of the recursive function involves structuring the various pointers and structure parameters so that the new node is added as a child for the position node if possible otherwise a new node is made for it. So a single iteration of *insert_part*() takes **O(1) time**. Either the function stops at this if position is a twonode otherwise moves to the parent in search of accomodation for the extra node created. At max the function continues till the root so if h(node) denotes height of node -
  Runtime Complexity = O(h(root))

- **insert**
  This function just checks some of the boundary cases while adding the tree rooted at node to the given tree T. All the boundary checks just take **O(1) time** and then the function calls *insert_part*() function. So if h(T) denotes the height for a tree T.
  Runtime Complexity = O(h(T))

- **Merge**
  The Merge function involves traversing the tree for finding it's height and minimum values. If we traverse along the leftmost path starting from the root till we get a leaf we can find both the height and minimum value of the tree. Clearly this requires **O(h(T)) time**. For getting the appropriate nodes we can follow either the leftmost or rightmost path downwards and then come the required number of steps up. Again this would require **O(h(th1)+h(th2)) time** where th1 and th2 are the given trees. Than the algorithm calls *insert*() function which again works in **O(h(T)) time**. So $overall\ iterations <= constant * (h(th1) + h(th2) + h(th1) + h(th2) + h(th2))$
  Runtime Complexity = O(h(th1)+h(th2))

So the overall time complexity for the running of algorithm Merge(T1,T2) would be **O(h(T1)+h(T2))**.