York University EECS 4101/5101 October 21, 2022

Homework Assignment #4
Due: October 28, 2022 at 11:59 p.m.

1. Suppose we want to be keep track of a collection of rooted trees of Nodes. The depth of a Node x, denoted $depth(x)$ is the distance from x to the root of its tree. We want to support the following three operations.

- CREATENODE(x) creates a new Node x in a tree by itself.
- SETPARENT(x, y) sets a Node x's parent to Node y. (This can only be done once on each Node x.)
- DEPTH(x) returns $depth(x)$.

Our implementation will use Node objects that each store the following four fields.

- *parent*: pointer to parent of the Node.
- *relative*: pointer to some Node in the same tree as this Node.
- *difference*: an integer variable. After every operation, each Node x must satisfy the following invariant. If $x.relative \neq$ null then $x.difference = depth(x) - depth(x.relative)$. If $relative =$ null, then $x.difference = depth(x)$.
- *rank*: an integer value that plays a similar role to the rank in a union-find data structure. Here, it is an overestimate of the height of the tree rooted at this node *defined by relative pointers* (not by *parent* pointers).

Below is an implementation with some blanks for you to fill in. Note the similarities with the code on page 530 of the textbook (here, *relative* plays the role of the p field of the text's code). This means that we are really storing two tree structures:

- the *actual trees* defined by SETPARENT, whose parent pointers are stored in the *parent* field, and
- *depth trees* similar to those that we use in the union-find data structure, whose parent pointers are stored in the *relative* field. These are just used to help us compute depths quickly.

Do not confuse these two tree structures. In particular, the depth of a node refers to depth in its actual tree. The data structure will satisfy an invariant that two Nodes are in the same actual tree if and only if they are in the same depth tree. (However, the actual tree and its corresponding depth tree might be entirely different shapes.) If you mention a tree in any of your answers, be clear about whether you mean an actual tree or a depth tree.

[12] **(a)** Fill in the blanks in the pseudocode given in Figure 1 so that the operations work correctly (and satisfy the invariant for the *difference* field mentioned above). For each blank, explain briefly how you came up with your answer.

[6] **(b)** Draw the actual trees and the depth trees after the following sequence of operations.
CREATENODE(A), CREATENODE(B), CREATENODE(C), CREATENODE(D),
CREATENODE(E), CREATENODE(F), CREATENODE(G), CREATENODE(H),
SETPARENT(A,B), SETPARENT(C,D), SETPARENT(E,F), SETPARENT(G,H),
SETPARENT(B,C), SETPARENT(F,G), SETPARENT(D,G), DEPTH(A)
Beside each Node in the depth tree, show the values of its *rank* and *difference* fields.

[3] **(c)** Give a good upper bound on the worst-case time for a single DEPTH operation when there are n Nodes in the data structure. Briefly explain why your answer is correct.

[3] **(d)** Give a good upper bound on the total time to do any sequence of m operations that contains n CREATENODE operations. Assume you start with an empty data structure. You do not have to give a complete proof of your bound, but you should briefly explain how you came up with it.

```
CreateNode(x)
    x.parent ← null
    x.relative ← null
    x.rank ← 0
    x.difference ← 0
end CreateNode
Find(x)
    ▷ helper function that follows relative pointers from x until reaching node y that has a null
    ▷ relative field (i.e., y is the root of x's depth tree); Find returns a pair ⟨y, depth(x) − depth(y)⟩.
    if x.relative ≠ null then
        ⟨root, d⟩ ← Find(x.relative)
        x.relative ← root            ▷ do path compression on relative pointers
        x.difference ←______________
        return ______________
    else
        return ⟨x, 0⟩
    end if
end Find
Link(r, s, d_y) ▷ makes one Node the relative of other, depending on ranks; d_y is used to update fields of nodes
    ▷ Precondition: r.relative = s.relative = null, i.e., r and s are roots of depth trees
    if r.rank > s.rank then
        s.relative ← r
        ______________ ← ______________
        ______________ ← ______________
    else
        r.relative ← s
        ______________ ← ______________
        if r.rank = s.rank then
            s.rank ← s.rank + 1
        end if
    end if
end Link
SetParent(x, y)
    ▷ Precondition: x.parent = null; i.e., once a Node has a parent, the parent cannot change.
    x.parent ← y
    ⟨r, d_x⟩ ← Find(x)
    ⟨s, d_y⟩ ← Find(y)
    Link(r, s, d_y)
end SetParent
Depth(x)
    ⟨root, d⟩ ← Find(x)
    return ______________
end Depth
```

Figure 1: Pseudocode