Part 4

—

Representing Rooted Forests

CSCI 3110 Code

Summer 2015

1 Introduction

Most of our graph algorithms construct and manipulate some kind of spanning tree or forest of the graph. Thus, it makes sense to have a representation for rooted trees; forests are just lists of rooted trees. In fact, as we will see, for some applications it is more useful to have the tree edges directed away from the root, while for others directing the edges toward the root is easier. So we provide representations for both cases.

2 Top-Down Trees and Forests

First we'll consider trees and forests where edges are directed away from the root. Thinking about the root being at the top of the tree, we call these "top-down" trees and forests. Since they're the more common types of trees and forests we'll manipulate in our algorithms, we still call the data types simply *Tree* and *Forest*, respectively. In theory, a tree is just a node together with a forest of child subtrees. However, we often want to remember which graph edge we used to get to a given vertex, so we augment this a bit:

```
newtype Forest    v e = Forest {trees :: [Tree v e]}
data    Tree      v e = Tree v [Subtree v e]
data    Subtree v e = Subtree e (Tree v e)
```

There are a number of standard orderings of the vertices of a tree, which are easily extended to a forest by concatenating the vertex orderings of its constituent trees. A *preordering* orders the vertices of the tree so that every node appears before all its descendants and the nodes in every subtree appear before the nodes in its right sibling subtree:

```
preOrderForest :: Forest v e → [v]
preOrderForest = concatMap preOrderTree ∘ trees
```

```
preOrderTree  :: Tree v e → [v]
preOrderTree t = go t [ ]
  where go (Tree v ws) vs = v : foldr go vs (map (λ(Subtree _ t') → t') ws)
```

In a *postordering*, every vertex needs to come *after* all its descendants and again the nodes in every subtree must appear before the nodes in its right sibling subtree:

```
postOrderForest :: Forest v e → [v]
postOrderForest = concatMap postOrderTree ∘ trees

postOrderTree  :: Tree v e → [v]
postOrderTree t = go t [ ]
  where go (Tree v ws) vs = foldr go (v : vs) (map (λ(Subtree _ t') → t') ws)
```

A *level ordering* finally orders the nodes of a tree so they appear level by level, left to right in each level:

```
levelOrderForest :: Forest v e → [v]
levelOrderForest = concatMap levelOrderTree ∘ trees

levelOrderTree  :: Tree v e → [v]
levelOrderTree t = levelOrder' [t]

levelOrder'  :: [Tree v e] → [v]
levelOrder' [ ] = [ ]
levelOrder' ts = map (λ(Tree v _) → v) ts ++
                 levelOrder' (concatMap (λ(Tree _ ws) → map (λ(Subtree _ t) → t) ws) ts)
```

To get convenient access to these ordering functions for trees and forests, we define a type class *TreeLike* here:

```
class TreeLike t v | t → v where
  preOrder  :: t → [v]
  postOrder :: t → [v]
  levelOrder :: t → [v]

instance TreeLike (Forest v e) v where
  preOrder  = preOrderForest
  postOrder = postOrderForest
  levelOrder = levelOrderForest

instance TreeLike (Tree v e) v where
  preOrder  = preOrderTree
  postOrder = postOrderTree
  levelOrder = levelOrderTree
```

3 Bottom-Up Trees and Forests

We represent a bottom-up tree or forest as a list of vertices. Each vertex stores its path to the root of its tree. Thanks to sharing of most of the structure of these lists (the path for a given node is composed of a single edge followed by the path for its parent), the internal structure of this representation is exactly the forest and thus uses linear space.

type *UpTree v e* = [(*v*, [(*e*, *v*)])]

So, for every vertex *v* an *UpTree* stores a pair consisting of *v* itself and a list of edge-vertex pairs representing the alternating sequence of edges and vertices that make up the path from *v* to the root. What we want to do with an *UpTree* depends a lot on the particular algorithm, so we provide only two primitive functions here: conversion from a *Forest* or *Tree* to an *UpTree* and back.

First let's focus on converting an *UpTree* to a *Forest*. We can't reasonably convert it to a *Tree* because there is no guarantee that the *UpTree* is in fact a single tree. If we are certain of this in the application where we use this function, we can of course always convert the single-tree *Forest* into a *Tree* using *head* ∘ *trees*. In general, we know very little about the order in which nodes are listed in the *UpTree*, which makes efficient conversion to a *Forest* difficult. So, once again, we actually offer six functions for *UpTrees* whose vertices are of *Eq*, *Ord* or *Ix* type or for which we know mappings from vertices to values of such types. The costs of these functions are $O(n^2)$, $O(n \lg n)$, and $O(n)$, respectively. We define these functions using a more general function *toForest* that again relies on a dictionary to do its work:

```
toForestEq :: Eq v ⇒ UpTree v e → Forest v e
toForestEq = toForestIndexByEq id

toForestOrd :: Ord v ⇒ UpTree v e → Forest v e
toForestOrd = toForestIndexByOrd id

toForestIx :: Ix v ⇒ UpTree v e → Forest v e
toForestIx = toForestIndexByIx id

toForestIndexByEq  :: Eq k ⇒ (v → k) → UpTree v e → Forest v e
toForestIndexByEq f = toForest f makeEqDict eqDictFind

toForestIndexByOrd  :: Ord k ⇒ (v → k) → UpTree v e → Forest v e
toForestIndexByOrd f = toForest f makeOrdDict ordDictFind

toForestIndexByIx  :: Ix k ⇒ (v → k) → UpTree v e → Forest v e
toForestIndexByIx f = toForest f makeIxDict ixDictFind
```

```
toForest :: (v → k) → ([k] → [(k, Subtree v e)] → d) → (d → k → [Subtree v e]) →
            UpTree v e → Forest v e
toForest f makeDict findDict t = Forest $ map (λ(v, _) → Tree v (findDict d (f v))) roots
  where (roots, nonroots) = partition (null ∘ snd) t
        d                 = makeDict (map (f ∘ fst) t)
                                     (map (λ(v, (e, p): _) → (f p, Subtree e $ Tree v $ findDict d (f v)))
                                          nonroots)
```

Holy cow! Let's take this apart slowly. First we split the vertices of the *UpTree t* into two lists: *roots* and *nonroots*. Roots are those vertices without proper ancestors; non-roots are all other vertices. The dictionary d we construct stores for every node v the list of subtrees of the tree with root v. Thus, our final output is the list of trees containing for every root v a tree with root v and whose subtrees are exactly the ones associated with $f\ v$ in d. So how do we build this dictionary d? We apply *makeDict* to a list of pairs of the form $(f\ v, s)$ where s is a subtree of the tree with root v. Every non-root v with parent p creates such a pair. Specifically, if e is the parent edge of v, then the resulting pair is $(f\ p, s)$, where s is a *Subtree* with root edge e and root node v. Of course we also need to store the subtrees of this tree. We find those using *findDict d* $(f\ v)$ exploiting laziness once again, that is, we can store elements in d that are defined via lookups in d because these elements are evaluated only when we use them later.

Conversion from a *Tree* or *Forest* to an *UpTree* is much easier in comparison:

```
toUpTreeT :: Tree v e → UpTree v e
toUpTreeT t = tut t [ ]
  where tut (Tree v ws) as = (v, as) : concatMap (λ(Subtree e t) → tut t ((e, v) : as)) ws

toUpTreeF :: Forest v e → UpTree v e
toUpTreeF = concatMap toUpTree ∘ trees
```

In order not to have to distinguish between *Tree*s and *Forest*s, we add a *toUpTree* function to the *TreeLike* type class:

```
class TreeLike t v e | t → v, t → e where
  preOrder   :: t → [v]
  postOrder  :: t → [v]
  levelOrder :: t → [v]
  toUpTree   :: t → UpTree v e

instance TreeLike (Forest v e) v e where
  preOrder   = preOrderForest
  postOrder  = postOrderForest
  levelOrder = levelOrderForest
  toUpTree   = toUpTreeF
```

```
instance TreeLike (Tree v e) v e where
  preOrder   = preOrderTree
  postOrder  = postOrderTree
  levelOrder = levelOrderTree
  toUpTree   = toUpTreeT
```