  Trees in Scheme

What is a tree??

LINK/EDGE

Element/NODE

Other parts of a tree

|  |  |  |
| --- | --- | --- |
| **Other parts of a tree** | | |
| Root | internal nodes | leafs |
|  |  |  |
| Path | descendants | Node makeup |
|  |  |  |

~~Length~~ of a tree (skipped)

|  |
| --- |
| **Determining the Length (Depth) of a Tree** |
| A tree, with height and depth of each node  src: <http://stackoverflow.com/questions/2603692/what-is-the-difference-between-tree-depth-and-height> |

Relationships of a tree - Part 1

“A” – no parents

**“A” - 2 children “K” and “Q”**

“K” – Parent is “A”   “Q” – Parent is “A”

“K” - 1 child “Z”

“Z” – Parent is “K”

“Z” – no children Z’s ancestors are “K” and “A”

Other Info on Trees

* Trees are ***SHALLOW*** – they can hold many nodes with very few levels
* A height of 21 can hold 1048575 nodes

2height -1 = How many TOTAL nodes can be held by this tree

* ***Tree is inherently recursive structure, because each node in a tree can itself be viewed as the root of a smaller tree***.
  + or called a sub-tree

Types Basic of Trees

* also called classification

|  |  |  |
| --- | --- | --- |
| **Types of Trees** | | |
| Regular Tree | BST | Reg. BT |
| undefined number of links | two links | two links |

* Regular Tree
  + may have MANY links to MANY kids!!!
* Binary Search Tree
  + has only **AT MOST** 2 children
  + used in Binary Search
  + used in Tree Traversals (covered later)
  + built (usually) in order
    - left value is LESS than parent node data
    - right value is MORE than parent node data
* Regular Binary Tree **// ex. Which-Way books**
  + has only 2 children
  + may have a link back up to the parent
  + links may not follow BST link options
  + or data inside each node has a pattern
  + each like (left and right) is traveled depending on choice
    - yes = left link
    - no = right link

Representing a Tree (skipped)

* There are two ways to construct trees
  + Linked Lists
    - use links to connect to the other nodes in the tree
  + Array (K-ary)
    - only if we know the MAXIMUM number of children allowed

|  |  |  |
| --- | --- | --- |
| **Representing a Tree** | | |
| Linked List | “Helter Skelter” | Array (K-ary) |
|  | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | [0] | [1] | [2] | [3] | [4] | [5] | [6] | | 7 | 5 | 1 | 2 | 3 |  | 6 |   **// it is possible to have an empty element** |

// draw indices OVER middle tree

|  |
| --- |
| DATA |
| parent->link |
| left->link |
| right->link |

Binary Tree Traversals (skipped)

* the process of “visiting” or performing some operations on, each node of a tree is called TREE TRAVERSAL
* a traversal is to process each node ONCE
* There are 3 commonly used to traversals
  + preorder
  + inorder
  + postorder
  + **they differ in the sequence in which the nodes are visited**
* PREORDER/PREFIX

1. process root first
2. if (left child present, and not visited already) move to left, restart with Step 1 at new node
3. if (right child present, and not visited already) move to right, restart with Step 1 at new node
4. Else, go back up to parent, and start at step 3

* overall, the algorithm reaches the root first
* INORDER/INFIX

1. if (left child present, and not visited already) move to left, restart with Step 1 at new node
2. process root
3. if (right child present, and not visited already) move to right, restart with Step 1 at new node
4. Else go back up to parent, and start at step 2

* POSTORDER/POSTFIX

1. if (left child present, and not visited already) move to left, restart with Step 1 at new node
2. if (right child present, and not visited already) move to right, restart with Step 1 at new node
3. Else process root, go back to parent, and start at step 2

* overall, the algorithm reaches the root LAST

Practicing Tree Traversals (skipped)

* Guarantees that we will visit EACH node at least once
  + No man left behind!!!
* Always start at the root

|  |  |  |
| --- | --- | --- |
|  | Inorder  (L, ***P***, R) | <http://youtu.be/SLslZb1wZuA>  (sorry, boring, but good info) |
| Preorder  (***P***, L, R) |  |
| Postorder  (L, R, ***P***) |  |
|  | Inorder |  |
| Preorder |  |
| Postorder |  |
|  | Inorder | ORDERED TREE!!!  How was Inorder Helpful?? |
| Preorder | How was PreOrder Helpful?? |
| Postorder | How was PostOrder Helpful?? |

Tree ADT in Scheme

1. We can use lists to represent a tree in scheme
2. every node has a ***datum*** and ***0 to more children***
3. An empty list will represent the empty tree or otherwise it will contain a root, left subtree and right subtree

(root left-subtree right-subtree more***?***) ; general form

* So a second element is the left subtree, third element is the right subtree and root is a singleton containing the node value.
* A simple function to construct the tree is

(define (make-node root left right more)

   (list root left right more))

Basic Scheme tree example

* ah, those ( ) come back to haunt us
* nested does make a difference.
* notice 3 answers
  + Scheme Code
  + Visually
  + Scheme Tree Visually

|  |  |
| --- | --- |
| **Coding a Tree in Scheme** | |
| Code | Visually |
| (make-node      'italy      (list           (make-node 'venezia '())           (make-node 'riomaggiore '())           (make-node 'firenze '())           (make-node 'roma '())      )  )  // notice the ‘()s, room for more nodes |  |
| (make-node ‘5  (list        (make-node        ‘3        (list (make-node ‘()              (make-node ‘4 ‘()        )        (make-node        ‘6        (list (make-node ‘()              (make-node ‘()        )        )  )  ) | SchemeTree |

|  |
| --- |
| As a Scheme Tree Visually |
| thanks Rohit Sandur - Su21’ |

|  |
| --- |
| 1. ***Draw*** how the Scheme **Lists** would handle this. (Should look like above) Answerb: |
| ‘ == null |

2. Notice I did not ask you to code above yet. Why? (Simple answer)

Scheme Tree Example 1

* constructor is “make-node”
  + takes two arguments, the datum and ***possible*** list of children
* datum function returns datum (duh) of the node
* children returns the list of children of the code

|  |
| --- |
| **Tree1.scm** |
| Constructor |
| (define (make-node datum children)    (**cons** datum children)) |
| Datum |
| (define (datum node)    (**car** node)) |
| Children |
| (define (children node)    (**cdr** node)) |

|  |
| --- |
| **The Tree and the accessing the tree with code** |
| world |

1. What does the World **node** (Scheme list) look like? (just the node) Answerb:

2. What would Italy’s node look like?

3. Use the code below to determine HOW the answers came to be.

|  |
| --- |
| **Running the Tree1.scm code** |
| ; (datum world-tree1)  (display (datum world-tree1))  **;WORLD**  ; (datum (car (children world-tree)))  (display (datum (car (children world-tree1))))  **;ITALY**  ; (datum (car (children (c**ad**r (children world-tree)))))  (display (datum (car (children (c**ad**r (children world-tree1))))))  **;CALIFORNIA**  (display   (datum (car (children (car (children (c**ad**r (children world-tree1))))))))  ; (datum (car (children (car (children (c**ad**r (children world-tree)))))))  **; Berkeley** |

1. Create a directory called Trees
2. Download the trees1.scm code by:
   1. (in UNIX)

wget <http://faculty.cse.tamu.edu/slupoli/notes/Scheme/code/trees/trees1.txt>

1. (in Windows)

<http://faculty.cse.tamu.edu/slupoli/notes/Scheme/code/trees/trees1.txt>

1. Run the code in Dr. Racket (might need to put #lang racket on top)

The rest (below) will take about 40 minutes, could start Pt 2 (but most done@home)

1. Draw two trees using the coded tree “worldtree1”. **(It’s not same as the picture on the previous page).** The first drawn tree as a normal tree, the second a tree using Scheme lists, you will need to save your work somehow
   1. if on a computer, save as a “.png”
   2. on paper, take a photo
2. The very last line of world-tree has a massive amount of parentheses (near Sudsbury). Change that line and move the parentheses to reflect a more block like structure. Run the program just to make sure it is still ok.
3. Determine how (in the code provided) California and Berkeley were displayed
   1. I want a written statement on how
4. Code a new country and 3 universities within that country.
   1. Run the program just to make sure it is still ok.
   2. display the children of the new country (will take several lines)
   3. capture the code and output
5. Add a new state to the US, and add 2 Universities
   1. Run the program just to make sure it is still ok.
   2. display the children of the new state
   3. capture the code and output
6. Whatever pics/code you created, email your teammate

Advanced Trees in Scheme

* Trees2.scm has less typing, but more features
* also includes mutual recursion

1. Download the trees2.scm code by:
   1. (in UNIX)

wget <http://faculty.cse.tamu.edu/slupoli/notes/Scheme/code/trees/trees2.txt>

1. (in Windows)

<http://faculty.cse.tamu.edu/slupoli/notes/Scheme/code/trees/trees2.txt>

1. Run the code in Dr. Racket
2. Write down what extra functionality does world-tree2 contain?
   1. (versus the first tree code given)
3. Draw ***some*** of the tree ***using lists***, you will need to save your work somehow
   1. is the structure REALLY any different than was in Tree1?
   2. if on a computer, save as a “.png”
   3. on paper, take a photo
4. Looking at United States and San Francisco, what is common in their code. Why is that? (something simple, **super dumb**) Why did I have to code it that way?
5. What is the left-most and right-most values respectfully?
6. Create the ***recursive*** function far-left (not politically motivated) that will find the left-most value.
7. To be fair and politically balanced, create the function far-right
8. What code would we use to find US, California, and Berkley respectfully?
9. Write depth, a procedure that takes a tree as argument and returns the largest number of nodes connected through parent-child links. That is, a leaf node has depth 1; a tree in which all the children of the root node are leaves has depth 2. Our world tree has depth 4 (because the longest path from the root to a leaf is, for example, world, country, state, city).
10. Write count-nodes, a procedure that takes a tree as argument and returns the total number of nodes in the tree. (Earlier we counted the number of leaf nodes.)

Mutual Recursion uses in Trees

* Note that count-leaves calls count-leaves-in-forest, and count-leaves-in-forest calls count-leaves.
* is often a useful technique for dealing with trees
* typical recursion sequentially moves through a list or sentence
  + with each recursive call taking us one step to the right
* three different models to help you think about how the shape of a tree gives rise to a mutual recursion
  + first model 🡪 “count-leaves”/”count-leaves-in-forest”
    - we want to count the leaves of a tree
    - unless the argument is a very shallow tree, this will involve counting the leaves of all of the children of that tree.
    - want is a straightforward sequential recursion over the list of children
      * (But we're given the wrong argument: the tree itself, not its list of children)
    - need an initialization procedure, count-leaves, whose job is to extract the list of children and invoke a helper procedure, count-leaves-in-forest, with that list as argument
  + second model 🡪 “count-leaves-in-forest”
    - easier to state but less rigorous
    - two-dimensional nature of trees, in order to visit every node we have to be able to move in two different directions
    - from a given node we have to be able to move down to its children, but from each child we must be able to move across to its next sibling.
    - The job of ***count-leaves-in-forest*** is to move from left to right through a list of children.
      * It does this using the more familiar kind of recursion, in which it invokes itself directly.
      * The downward motion happens in count-leaves, which moves down one level by invoking children
* third model
  + also based on the two-dimensional nature of trees
  + what if each node in the tree has at most one child
    - In that case, count-leaves could move from the root down to the single leaf with a structure very similar to the actual procedure, but carrying out a sequential recursion:

(define (count-leaf tree)

  (if (leaf? tree)

      1

      (count-leaf (child tree))))

* the trouble with this is that at each downward step there isn't a single "next" node
* Instead of a single path from the root to the leaf, there are multiple paths from the root to many leaves
* we must "clone" count-leaves as many times as there are children
* Count-leaves-in-forest is the factory that manufactures the clones.
  + for each child and accumulates their results.

Answers

2nd Tree as a Scheme List:

|  |
| --- |
|  |

What does the World node look like?

|  |
| --- |
|  |

What does Italy look like?

|  |
| --- |
|  |

World-Tree1 Exercises

|  |  |  |
| --- | --- | --- |
| #1 |  | |
| Normal Tree | Tree as a List | |
| Thanks Jonathon Hillier F’18 | Thanks Luke Grammar F’18 | |
| #3 | |  |
| Thank you Katherine Ruff F’18 | | |
| #4 | | |
| (make-node 'Japan                             (list (make-node 'Nihon '())                                   (make-node 'Tokyo '())                                   (make-node 'Kwansei '())                              ) | | |

|  |
| --- |
| #5 |
| (make-node 'texas                             (list (make-node 'college\_station '())                                   (make-node 'austin '())                                   )                             )                  ) |

Sources

<https://people.eecs.berkeley.edu/~bh/ssch18/trees.html>