2-Trees

Bruno MARTIN,
University of Nice - Sophia Antipolis
mailto:Bruno.Martin@unice.fr
http://deptinfo.unice.fr/~bmartin/mathmods.html

Glossary on Tree

A tree is a nonempty collection of connected elements: the **nodes**

One of the elements is distinguished: the **root**

The nodes below (above) a node are its **descendants** (**ancestors**)

Each node has exactly one ancestor : its **parent**

The nodes directly below a node are its **children**

A node with no children is called a **leaf** or a **terminal node** or an **external node**

A node with children is a **nonterminal node** or an **internal node**

Trees

Trees are intimately connected with **recursion** : a tree is either a **single element** or a **root element** connected to a set of trees.

Extensive use in **computer science**:

- to represent the syntactic structure of source programs
- to decribe **arithmetic expressions** in programs.

Of common use for both **sorting** and **searching** because the **running time** of **searching** an element among N can be **logarithmic**.

Glossary on pathes

A **path** from n_1 to n_k is the sequence $n_1, \ldots, n_k$ such that n_i is the parent of n_{i+1} (n_1 an ancestor of n_k and n_k a descendant of n_1)

The **length** of a path equals the number of nodes in the path -1

The **height** of a **node** is the length of a longest path from the node to a leaf

The **height** of a **tree** is the height of the root

The **depth** of a **node** is the length of the unique path from the root to that node

Example of length and height of paths

```
                    O       ^                   ^
                  / | \     | depth of X        |
                 /  |  \    | is 1              |
       ^        N   O   X   v       ^           |
       |       /\   |   |           | path of   | tree
height |      /  \  |   |           | length 1  | height
of N   |     O    O O   O           v           | is 3
is 2   |     |                                  |
       |     |                                  |
       v     O                                  v
```


Binary trees

A **binary tree** is an ordered tree with three types of nodes : **leaves**, **unary nodes** and **binary nodes**

A binary tree is **strictly binary** if its **internal nodes** have **exactly two** children

A strictly binary tree is **full** when nodes completely fill every level, except possibly the last one

Glossary

There is exactly **one path** between the **root** and some **node** (otherwise it is a graph)

Any node is the root of a **subtree**

The nodes in a tree are divided into **levels** : nodes with same depth

In an **ordered** (oriented) tree the children of each node are ordered from left-to-right

A *n*-**ary** tree is a tree where the internal nodes have **at most** *n* **children**

Properties on Trees

[1] A tree with N nodes has $N-1$ edges

- Each node except the root has a unique parent connected by one edge

[2] A strictly binary tree with I internal nodes has $E = I + 1$ leaves

- **By induction** for $I = 0$: a strictly binary tree with no internal nodes has **one** leaf: the root
- For $I > 0$, a tree with I internal nodes has k internal nodes in its left subtree and $I - k - 1$ nodes in its right subtree. Since $0 < k < I - 1$ by induction hypothesis : the left subtree has $k + 1$ leaves and the right $I - k$ ...

Properties on Trees

For a binary tree with N nodes we have

$$\log_2(N) \leq height \leq N-1$$

Consider all the binary trees of height h:

- The one with the **minimum number of nodes** is the tree reduced to a path from the root where each parent has only a child : $h = N-1$; so for a binary tree $h \leq N-1$
- The one with the **maximum number of nodes** is the binary tree with all levels filled with nodes:
  - 2^0 on the root level, 2^1 on the first level, 2^i on level i^{st} and 2^h on the last level
  - $N = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ nodes
  - $N < 2^{h+1} \Rightarrow \log_2(N) \leq h$

A Ruby Tree

```
class BinaryTree
  class Node
    attr_reader :left, :right, :value

    def initialize()
     @left, @right, @value = nil, nil, nil
    end
  end
  attr_reader : root

  def initialize
   @root = nil
  end
end
```


Representing Binary Trees

A data structure for a binary tree is done with 2 `ruby` classes:

- one for the tree
- one for the nodes with **two links per node** (`left` + `right`) and a **field** for the **information** about the node's `value`

For **leaves** the **two links are nil**.

Traversing Binary Trees

Problem: How to **traverse** a tree i.e. how to systematically visit every node. 4 ways to proceed according to the **order** in which the root and the two children are visited
Suppose your tree is an arithmetic expression

- **preorder traversal:**
  ```
  visit the root,
  visit the left subtree,
  visit the right subtree
  ```
  you visit the expression in the prefix manner
- **inorder traversal:**
  ```
  visit the left subtree,
  visit the root,
  visit the right subtree
  ```
  you visit the expresion in an infix manner

Traversing Binary Trees

- **postorder traversal:**
  ```
  visit the left subtree,
  visit the right subtree,
  visit the root
  ```
  you visit the expresion in a postfix manner
- **level-order traversal:**
  ```
  visit the levels from top to bottom,
  in each level visit the nodes from left to
  right
  ```

Notice: The implementation of first three traversals is done by recursion. The last traversal is not recursive at all : it is not a stack based but a queue based strategy

Implementation of Preorder Traversal

```
def preOrder(node)
   puts node.value
   if node.left != nil
     preOrder(node.left)
   end
   if node.right != nil
     preOrder(node.right)
   end
 end


   print "Post-Order Traversal of tree\n"
   postOrder(@root)
```


Example of Binary Tree Traversing

```
   +
  / \
 2   *
    / \
   3   +
      / \
     10  5
```

Traverse the previous tree in preorder, inorder and postorder, print the information contained in the node

Implementation of Inorder Traversal

```
def inOrder(node)
  if node.left != nil
    inOrder(node.left)
  end
  p node.value
  if node.right != nil
    inOrder(node.right)
  end
end


  print "In-Order Traversal of tree\n"
  inOrder(@root)
```

Implementation of Postorder Traversal

```
def postOrderNode
  if @left != nil
    @left.postOrderNode
  end
  if @right != nil
    @right.postOrderNode
  end
  p node.value
end

  def postOrder
    return if @root.nil?
    @root.postOrderNode
  end
```

```
def postOrderNode
   @left.postOrderNode if @left
   @right.postOrderNode if @right
   print(@value, " ")
end
```


Binary Search Trees

For searching we associate a **key value** to each internal node
For any node, all nodes with **smaller keys** are in the **left subtree**
All nodes with **larger keys** are in the **right subtree**
To find a node with a given key `key` we run a recursive **search** from the root node with the searched `key`

Recursive calls

When a recursive call is executed, the "current environment" is saved in the execution stack and restored at the exit of the procedure

The **amount of memory** taken by the **execution stack during** the **traversal** of the tree is proportionnal to the **height** of that **tree** though the memory management is hidden to the programmer
⇒ the **analysis of the height** of the tree is important for the performance of the program

Implementation of Searching in a Binary Tree

```
def SearchNode?(key)
  if @value == key
    puts "FOUND"
    elsif @value < key
      if @right != nil
        @right.SearchNode?(key)
        else return "NOT FOUND"
      end
    elsif @left != nil
       @left.SearchNode?(key)
    else  return "NOT FOUND"
  end
end
```

```
def Search?(key)
  return if @root.nil?
  @root.SearchNode?(key)
end
```

Example of Insertion on Binary Tree

A binary tree is built by inserting node one by one at the good position

We insert A, S, E, A, R, C, H, I, N, G in a binary tree

```
  A      A       A        A        A         A
 / \    / \     / \      / \      / \       / \
           S       S        S        S         S
                  / \      / \      / \       / \
                 E        E        E         E        ...
                / \      / \      / \       /   \
                        A        A   R     A     R
                       / \      / \ / \   / \   / \
                                             C
```


Insertion Tree Part

```
def insert( value )
   if @root.nil?
     @root = Node.new( value )
   else
     @root.insertNode( value )
   end
end
```


Insertion Node Part

```
def insertNode( value )
  if value >= @value then # insert right
    if @right.nil?
      @right = Node.new( value )
    else
      @right.insertNode( value )
    end
  else # insert left
    if @left.nil?
      @left = Node.new( value )
    else
      @left.insertNode( value )
    end
  end
end
```


The shape of the trees

The **shape of the tree** and the **number of steps** to build it depends on the **order** in which the **keys** have been **inserted**

```
With keys in increasing order,                  A
the right subtree of the root                  / \
is reduced to a single path                       B
                                                 / \
Inserting A B C D               =>                  C
                                          D        / \
                                         / \          D
With keys in decreasing order,          C            / \
the left subtree of the root           / \
is reduced to a single path           B
                                     / \
Inserting D C B A               =>  A
                                   / \
```

Performance of Insertion and Searching on Binary Tree

Strongly depends on the shape of the tree

- When we insert the N nodes in order
  - The tree is reduced to a single path of length $N-1$
  - We must then examine $i-1$ nodes before inserting node i
  - The **insertion** takes N comparisons ($O(N)$)
  - The **unsuccessful search** takes N comparisons ($O(N)$)
- When the tree is **balanced**
  - The **unsuccessful search** takes $\log(N)$ comparisons (because of the tree height)
  - We must examine $\log(i-1)$ nodes before inserting node i
  - The **insertion** takes $\log(N)$ comparisons

Balanced Trees

For binary tree searching, there is a general technique that enables us to **guarantee** that the worst case will not occur
This technique is called **Balancing** and is used as the basis for several different "balanced-tree" algorithms:

- The AVL Tree (Adelson Velskii and Landis)
- Top-Down 2-3-4 Trees
- Red-Black Trees
- B-tree (an extension of 2-3-4 trees)