An Applicative, $O(N \times \log_k(N) \times N(k-1)/k)$ Algorithm to Reconstruct Trees from Preorder Node Lists

Arthur Alexander Reyes
reyes@cse.uta.edu

Monica Nadkar
mvnadkar@yahoo.com

Technical Report CSE-2003-10

This report was also submitted to the 2003 International Conference on Functional Programming (ICFP 2003).
An Applicative, $O(N \cdot \log_k(N) \cdot N(k-1)/k)$ Algorithm to Reconstruct Trees from Preorder Node Lists\(^1\)

Arthur Alexander Reyes  
Assistant Professor  
Department of Computer Science & Engineering  
The University of Texas at Arlington  
Arlington, TX 76019-0015, USA  
reyes@cse.uta.edu

Monica Nadkar  
Graduate Student  
Department of Computer Science & Engineering  
The University of Texas at Arlington  
Arlington, TX 76019-0015, USA  
mvnadkar@yahoo.com

ABSTRACT  
We describe a functional pearl we discovered—an applicative, $O(N \cdot \log_k(N) \cdot N(k-1)/k)$ algorithm to reconstruct a tree from a preorder list of the tree’s nodes, where $N$ is the length of the preorder node list and $k$ is the maximum number of children (i.e., degree) a node can possess. Nodes in the preorder node list are annotated with their respective degrees. Imperative algorithms have complexity $O(N)$. It appears the cost of performing our reconstruction applicatively is the performance penalty factor of $\log_k(N) \cdot N(k-1)/k$ – when a new node is inserted into the growing tree, the tree must be traversed from root to the insertion point. In addition to parameters for the preorder list and the tree being reconstructed, our algorithm has a third parameter that keeps track of where the next node must be inserted in the growing tree. We believe the benefit of this algorithm’s applicative style is its elegance. We describe a simple, but rather fault-intolerant implementation in Erlang.

Categories and Subject Descriptors  
D.1.1 Applicative (Functional) Programming  
E.1 DATA STRUCTURES, Trees

General Terms  
Algorithms.

Keywords  
Erlang, preorder node list, tree algorithm

1. INTRODUCTION  
A common exercise in algorithms courses is to write algorithms to traverse a tree and generate a list of the tree’s nodes in preorder, inorder & postorder. Occasionally students are also tasked with writing algorithms to regenerate a tree from a list of the tree’s nodes given in preorder, inorder and postorder. Applicatively transforming a preorder node list into a tree is tricky because you need an extra, recursive data structure to keep track of where to insert the next node into the tree. This paper describes what we believe is a novel, applicative algorithm to reconstruct a tree from a preorder node list. Our algorithm maintains an index to keep track of where to insert the next node in the growing tree.

2. OVERVIEW OF THE ALGORITHM  
Consider the list $[5,0,0,4,0,3,0,2,1,0,0,0,0,0,0,2,1,0,0]$. Suppose each element in this list represents the number of children (i.e., the degree) of a node in a tree. “5” represents a node with five children. “0” represents a node with zero children (i.e., a leaf node). If we interpret this list to represent a preorder traversal of a tree, then Figure 1 shows the tree.

\[ \begin{array}{c} 
\text{Figure 1: Tree with Preorder Traversal} \\
[5,0,0,4,0,3,0,2,1,0,0,0,0,0,0,2,1,0,0] 
\end{array} \]

Let us represent trees by nested tuples. Our discussion uses the syntax of Erlang [AVWW1996]. The nested tuple corresponding to the example tree is therefore $\{5,0,0,\{4,0,\{3,0,\{2,\{1,0\},0\},0\},0,0\},\{2,\{1,0\},0\}\}$. Our algorithm takes as input the list $[5,0,0,4,0,3,0,2,1,0,0,0,0,0,2,1,0,0]$ and produces as output the tuple $\{5,0,0,\{4,0,\{3,0,\{2,\{1,0\},0\},0\},0,0\},\{2,\{1,0\},0\}\}$, for example.

\(^1\) Submitted to the 2003 International Conference on Functional Programming (ICFP)
Table 1 shows the steps our algorithm follows to perform this example transformation. Steps advance from the top of the table to the bottom of the table. The leftmost column is the node at the head of the preorder list, which is the first argument of our algorithm. The second column in the table is the state of the growing tree, which is the second argument of our algorithm. The third column in the table is the state of the insertion index data structure, which is the third argument of our algorithm. The symbolic constant “e” (i.e., an “atom” in Erlang) in the slots of a tree tuple means that the children have not yet been determined. Initially the tree is the empty tree “e” and the insertion index is the empty list [].

The algorithm consumes the head of the preorder list and replaces the empty (sub)tree at the index. If the head of the preorder list is “0”, the empty indexed subtree is replaced with a “0”. If the head of the preorder list is not “0”, the empty indexed subtree is replaced with a tuple. If the head of the preorder list is a number k > 0, then the empty indexed subtree is replaced with a (k+1)-tuple. The first element of such a tuple is the number k and all remaining k elements are each set to the empty tree.

The insertion index is a list of pairs. The insertion index represents a path from the root of the tree to an empty subtree. The insertion index denotes the first empty subtree that would be encountered during a preorder traversal of the tree. E.g., the insertion index ([(5,3), (4,2), (3,2), (2,1), (1,1)]) denotes the first child of the first child of the second child of the second child of the third child of the root. The first component of each pair in the insertion index is the degree k of that node. We retain this information to more conveniently detect when a node has had all its children instantiated.

The Erlang function shown in Figure 2, preOrdLstToTreAux (preorder list to tree—auxiliary) demonstrates recursively how to consume the preorder node list and generate the tree. This demonstration considers tree nodes with maximum number of children (degree) k = 5. A more general implementation of our algorithm would work for an unlimited maximum degree k.

The first clause (at line 1) is the base case. The base case occurs when the preorder node list is empty. For the base case, simply return the Tree that has been constructed thus far. The insertion indexIdx can be ignored (but a more robust implementation of the algorithm would verify that it is indeed an empty list).

As shown in line 2, if a 0 is at the head of the preorder node list, this represents a leaf node (i.e., a node with degree = 0). preOrdLstToTreAux calls itself with the preorder node list’s head removed, a “0” node is inserted into the growing Tree, and the insertion index is modified by the function rmvFldIncLst, which we will describe later.

If the node at the head of the preorder node list represents a non-leaf node (i.e., a node with degree k > 0), preOrdLstToTreAux calls itself with the preorder node list’s head removed, with Tree replaced with an appropriate call to our insert function, and the insertion indexIdx replaced by a modified insertion index that has a new pair appended. The pair appended to the insertion index tells the insert function that the next node must be inserted at the first empty child of the node just inserted. These details are shown in lines 3 through 7 below.

3. IMPLEMENTATION DETAILS

Table 1: "Animation" of the Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Head of Preorder List</th>
<th>Growing Tree Tuple</th>
<th>Insertion Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>e</td>
<td>[]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>[5, e, e, e, e, e]</td>
<td>[(5,1)]</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>[5, 0, e, e, e]</td>
<td>[(5,2)]</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>[5, 0, 0, e, e, e]</td>
<td>[(5,3)]</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>[5, 0, 0, 4, e, e, e, e]</td>
<td>[(5,3), (4,1)]</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>[5, 0, 0, 4, 0, e, e, e, e]</td>
<td>[(5,3), (4,2)]</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>[5, 0, 0, 4, 0, 3, e, e, e, e, e]</td>
<td>[(5,3), (4,2), (3,1)]</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>[5, 0, 0, 4, 0, 3, 0, e, e, e, e, e]</td>
<td>[(5,3), (4,2), (3,2)]</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>[5, 0, 0, 4, 0, 3, 0, 2, e, e, e, e, e]</td>
<td>[(5,3), (4,2), (3,2), (2,1)]</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>[5, 0, 0, 4, 0, 3, 0, 2, 1, e, e, e, e, e]</td>
<td>[(5,3), (4,2), (3,2), (2,1), (1,1)]</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>[5, 0, 0, 4, 0, 3, 0, 2, 1, 0, e, e, e, e, e]</td>
<td>[(5,3), (4,2), (3,2), (2,2)]</td>
</tr>
</tbody>
</table>

The insertion index denotes the first empty subtree that would be encountered during a preorder traversal of the tree. E.g., the insertion index ([(5,3), (4,2), (3,2), (2,1), (1,1)]) denotes the first child of the first child of the second child of the second child of the third child of the root. The first component of each pair in the insertion index is the degree k of that node. We retain this information to more conveniently detect when a node has had all its children instantiated.
1. preOrdLstToTreAux([], Tree, Idx) -> Tree;
2. preOrdLstToTreAux([0|T], Tree, Idx) -> preOrdLstToTreAux(T, insert(0, Tree, Idx), lists:append(Idx, [[1,1]]));
3. preOrdLstToTreAux([1|T], Tree, Idx) -> preOrdLstToTreAux(T, insert({1,e}, Tree, Idx), lists:append(Idx, [[2,1]]));
4. preOrdLstToTreAux([2|T], Tree, Idx) -> preOrdLstToTreAux(T, insert({2,e,e}, Tree, Idx), lists:append(Idx, [[3,1]]));
5. preOrdLstToTreAux([3|T], Tree, Idx) -> preOrdLstToTreAux(T, insert({3,e,e,e}, Tree, Idx), lists:append(Idx, [[4,1]]));
6. preOrdLstToTreAux([4|T], Tree, Idx) -> preOrdLstToTreAux(T, insert({4,e,e,e,e}, Tree, Idx), lists:append(Idx, [[5,1]])).

The node insertion function insert is defined in Figure 3. Read calls to insert as “insert the node (first argument) into the subtree (second argument) at the insertion index (third argument).” A more general implementation of our algorithm would collapse most of its clauses below into one, but we show explicit clauses below to make our algorithm easier to understand.

Line 1’s clause is the base case. Given a node, an empty subtree, and an empty insertion index, we simply return the node. If the Tree matches the pattern e, then the insertion index has to be empty. A more robust implementation of our algorithm would check this.

Line 2’s clause considers what to do when given a node, a subtree that matches the pattern [1,A], and an insertion index with head {1,1}; we simply return a Tree of the form {1,insert(Node,A,T)}, where T is the tail of the insertion index. The head of the insertion index can only be {1,1} in this case, because the current node has degree k = 1 and only child 1 of the node can exist to be traversed further.

Lines 3 and 4 consider what to do when given a node, a subtree that matches the pattern [2,A,B], and two possible insertion index head values. If the head of the insertion index is {2,1}, then we return a tree in which the Node is inserted into the first child A of the subtree [2,A,B]. If the head of the insertion index is {2,2}, then we return a tree in which the Node is inserted into the second child B of the subtree [2,A,B]. Lines 5—16 are analogous.

1. insert(Node,e, []) -> Node;
2. insert(Node,{1,A}, [{1,1}|T]) -> {1,insert(Node,A,T)};
3. insert(Node,{2,A,B}, [{2,1}|T]) -> {2,insert(Node,A,T),B};
4. insert(Node,{2,A,B}, [{2,2}|T]) -> {2,A,insert(Node,B,T)};
5. insert(Node,{3,A,B,C}, [{3,1}|T]) -> {3,insert(Node,A,T),B,C};
6. insert(Node,{3,A,B,C}, [{3,2}|T]) -> {3,A,insert(Node,B,T),C};
7. insert(Node,{3,A,B,C}, [{3,3}|T]) -> {3,A,B,insert(Node,C,T)};
8. insert(Node,{4,A,B,C,D}, [{4,1}|T]) -> {4,insert(Node,A,T),B,C,D};
9. insert(Node,{4,A,B,C,D}, [{4,2}|T]) -> {4,A,insert(Node,B,T),C,D};
10. insert(Node,{4,A,B,C,D}, [{4,3}|T]) -> {4,A,B,insert(Node,C,T),D};
11. insert(Node,{4,A,B,C,D}, [{4,4}|T]) -> {4,A,B,C,insert(Node,D,T)};
12. insert(Node,{5,A,B,C,D,E}, [{5,1}|T]) -> {5,insert(Node,A,T),B,C,D,E};
13. insert(Node,{5,A,B,C,D,E}, [{5,2}|T]) -> {5,A,insert(Node,B,T),C,D,E};
14. insert(Node,{5,A,B,C,D,E}, [{5,3}|T]) -> {5,A,B,insert(Node,C,T),D,E};
15. insert(Node,{5,A,B,C,D,E}, [{5,4}|T]) -> {5,A,B,C,insert(Node,D,T),E};
16. insert(Node,{5,A,B,C,D,E}, [{5,5}|T]) -> {5,A,B,C,D,insert(Node,E,T)}.

The last implementation detail we will discuss is the function rmvFldIncLst. rmvFldIncLst modifies the insertion index when a leaf (node with degree k = 0) is inserted into the tree. rmvFldIncLst stands for “remove filled nodes from the insertion index and increment any remaining last node's index.” Manipulation of the insertion index is straightforward. There are two cases: inserting a leaf node when the subtree is full, and inserting a leaf node when the subtree is not full.

rmvFldIncLst(L) -> lists:reverse(rmvFldIncLstAux(lists:reverse(L))).

The auxiliary function is defined in Figure 4. Line 1 is the base case. Modifying an empty insertion index list results in an empty insertion index list.

When a leaf node is inserted into a subtree that is full (line 2), the last pair of the insertion index will have the form {k,k}. The last pair and all last pairs of the form {k,k} will be removed.

Recall that modifications to the insertion index are performed at the end of the list or from the last element to the first element. Because pattern matching for Erlang lists (i.e., [H|T]) is stated in terms of the head element H and tail list T, it is convenient to first reverse the insertion index list L, modify it via a pattern-matching auxiliary function, and reverse the modified list again.

When a leaf node is inserted into a subtree that is not full (line 3), the last pair of the insertion index will have the form [k,m], where k > m. If any such pair remains, it will have the value of m incremented by one.
4. ANALYSIS OF WORST-CASE EXECUTION TIME GROWTH

4.1 rmvFldIncLst
We assume the built-in Erlang function \texttt{lists:reverse} executes in time proportional to its input. \texttt{rmvFldIncLst} calls \texttt{lists:reverse} twice. \texttt{rmvFldIncLstAux} executes in time proportional to its input as well. Therefore \texttt{rmvFldIncLst} executes in time proportional to its input. The input to \texttt{rmvFldIncLst} is the insertion index list. In the worst case, the insertion index list will be the same size as the tree’s height. We will show later that in the worst-case, the tree’s height grows as \( O(\log k(N)) \), where \( k \) is the largest node degree encountered and \( N \) is the number of nodes.

\texttt{rmvFldIncLst} is called once by \texttt{preOrdLstToTreAux} for every leaf inserted into the tree. In the worst case, the number of leaves in a tree with \( N \) nodes of degree \( k \) is \( N(k-1)/k \).

An Mathematical proof for the same is shown below:

Number of leaves in a \( k \)-ary tree with depth \( H \) is given by \( k^H \).

Total Nodes \( N \) in the tree:

\[
N = \sum_{M=0}^{M=H} k^M = \frac{k^{H+1}}{k-1}
\]

Therefore height \( H \) of a completely filled \( k \)-ary tree with \( N \) nodes is:

\[
N(k-1) = k^{H+1} - 1,
\]

\[
\log_k(N(k-1)+1) = H + 1,
\]

\[
H = \log_k(N(k-1)+1) - 1
\]

As there are \( k^H \) leaves,

\[
k^H = k \log_k(N(k-1)+1)
\]

\[
k^H = \frac{k^{log_k(N(k-1)+1)}}{k}
\]

\[
k^H = \frac{N(k-1)+1}{k}
\]

Therefore the worst-case number of times \texttt{rmvFldIncLst} is called for a \( k \)-ary tree with \( N \) nodes varies as \( O(N(k-1)/k) \).

4.2 insert
\texttt{preOrdLstToTreAux} calls \texttt{insert} at most once for each node in the preorder node list. Once called by \texttt{preOrdLstToTreAux}, \texttt{insert} calls itself recursively a number of times proportional to the current height of the tree. We believe \texttt{insert} takes \( O(\log(N)) \), where \( k \) is the largest node degree considered and \( N \) is the number of nodes. An mathematical proof for the same is given below:

Consider a complete \( k \)-ary tree with \( N \) nodes. The root has \( k \) children at depth 1, each of which has \( k \) children at depth 2 and so on and so forth. Since the number of nodes on the successive levels follows a geometric progression (1, \( k^2 \), \( k^3 \), ...), the number of nodes at level \( M \) is \( k^M \).

Therefore total Nodes \( N \) in the tree:

\[
N = \sum_{M=0}^{M=H} k^M = \frac{k^{H+1}}{k-1}
\]

where \( H \) is the height of the tree.

Therefore height \( H \) of a completely filled \( k \)-ary tree with \( N \) nodes is:

\[
N(k-1) = k^{H+1} - 1,
\]

\[
\log_k(N(k-1)+1) = H + 1,
\]

\[
H = \log_k(N(k-1)+1) - 1
\]

As \( \texttt{insert} \) function inserts in time proportional to the height, it takes \( O(\log_k(N)) \).

4.3 preOrdLstToTreAux
\texttt{preOrdLstToTreAux} is called at most once for every node in the preorder node list. If there are \( N \) nodes, \texttt{preOrdLstToTreAux} will be called at most \( N \) times.

4.4 SUMMARY
By multiplying the worst-case growth factors, the expected time complexity of our algorithm is \( O(\log_k(N)) \), where \( k \) is the maximum node degree and \( N \) is the number of nodes.

5. RELATED WORK
We were not able to find a published account of an applicative algorithm that reconstructs a tree from a preorder node list. However, we were able to locate a published account of imperative solutions to very similar problems. The work most closely related to us is Standish’s [Standish] algorithm of binary tree reconstruction from preorder enumeration of its nodes. The algorithm involves the use of two bits per node, where information of the node’s left descendant and right sibling is stored. He calls this a “marked preorder node sequence”. Standish’s algorithm’s worst-case execution time growth is \( O(N) \), where \( N \) is the number of nodes.

Our preorder nodes lists are what Standish calls an “arithmetic tree representation”, because each node contains its degree. If we convert the arithmetic tree representation, i.e. tree encoded
using arithmetic facts like the degree of nodes, as we do in our algorithm to two bits representation (information regarding left descendant and right sibling) then we can use Standish’s algorithm to reconstruct the tree. One can think of our algorithm as an applicative solution to exercises 3.19 & 3.20 in [Standish]. Standish states that working with a preorder node list is more difficult than his “left descendant and right sibling” representation because “finding the extent of a subtree rooted at a given node requires computation.” By maintaining the insertion index, our algorithm makes this computation simple.

Perlis [Perlis] developed methods for manipulating trees represented in preorder list with weights (i.e., the number of nodes below them) using concise expressions in the programming language APL.

6. REFERENCES


7. APPENDIX: IMPLEMENTATION IN ERLANG

% Copyright 2003 Arthur Alexander Reyes. All rights reserved.
% author: Arthur Alexander Reyes
% date: 2003/01/16

% This module applicatively models the general activity of
% transforming a list of tree node arities given in preorder back into
% the original tree. This model assumes the list given as the actual
% parameter is well formed.

% Trees are represented by tuples. The first element of each tuple is
% a number signifying the number of children @ that tree/node/branch.

-module(preOrdLstToTre).
-export([preOrdLstToTre/1, runTst/0]).

% preOrdLstToTre(PreorderList) -> Tree
% Initialize Tree to e (the atom representing the empty tree or
% branch) & Idx to [].
preOrdLstToTre(PreorderList) -> preOrdLstToTreAux(PreorderList,e,[]).

% The test driver appears below.

tstPreOrdLstToTre() ->
[ preOrdLstToTre([5,0,0,0,0,0,0,0]) == {5,0,0,0,0,0},
  preOrdLstToTre([5,0,0,0,0,0,1,0]) ==
    {5,0,0,0,0,0,1,0},
  preOrdLstToTre([5,0,0,0,0,1,1,1,1,1,1,0]) ==
    {5,0,0,0,0,1,1,1,1,1,1,0},
  preOrdLstToTre([5,1,0,1,0,1,0,1,0,1,0,1,0]) ==
    {5,1,0,1,0,1,0,1,0,1,0,1,0},
  preOrdLstToTre([5,4,3,2,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0]) ==
    {5,4,3,2,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
  preOrdLstToTre([5,0,0,0,4,0,0,0,3,0,0,2,0,1,0]) ==
    {5,0,0,4,0,3,0,2,1,0,0,0,0,0,0,2,1,0,0},
  preOrdLstToTre([5,0,0,4,0,3,0,2,1,0,0,0,0,0,0,2,1,0,0]) ==
    {5,0,0,4,0,3,0,2,1,0,0,0,0,0,0,2,1,0,0} ].

% preOrdLstToTreAux(PreorderList,Tree,InsertionIndex) -> Tree'}
% The InsertionIndex Idx is a list of pairs. The first element of each pair is the number of arguments @ the insertion node. The second element of each pair is the location in the node where the new node is to be inserted. Notice the value of the second component can never exceed the value of the first component. Thus 
% \[ \{3,1\}, \{4,2\} \] is an insertion index that states that the new node is to be inserted in the second position of the first node.

% When the current insertion of a 0 will fill up all the slots @ the current node, this is denoted by an Idx matching the pattern 
% \[ \ldots,\{N,N\} \]. After inserting a 0 @ the last slot of a node, the Idx must have the last pair removed, and the new last pair must have the second component incremented by 1.

% When the current insertion of a 0 will not fill up all the slots @ the current node, this is denoted by an Idx matching the pattern 
% \[ \ldots,\{N,M\} \] when N >= M. After inserting a 0 @ a slot that is not the last slot in a node, the Idx must have the second component of the last pair incremented by 1.

\[
\text{preOrdLstToTreAux}([], \text{Tree}, \text{Idx}) \rightarrow \text{Tree};
\text{preOrdLstToTreAux}([0|T], \text{Tree}, \text{Idx}) \rightarrow \text{preOrdLstToTreAux}(T, \text{insert}(0, \text{Tree}, \text{Idx}), \text{rmvFldIncLst}(\text{Idx}));
\text{preOrdLstToTreAux}([1|T], \text{Tree}, \text{Idx}) \rightarrow \text{preOrdLstToTreAux}(T, \text{insert}(1, \{e\}, \text{Tree}, \text{Idx}), \text{lists:append}(\text{Idx}, \{1,1\}));
\text{preOrdLstToTreAux}([2|T], \text{Tree}, \text{Idx}) \rightarrow \text{preOrdLstToTreAux}(T, \text{insert}(2, \{e,e\}, \text{Tree}, \text{Idx}), \text{lists:append}(\text{Idx}, \{2,1\}));
\text{preOrdLstToTreAux}([3|T], \text{Tree}, \text{Idx}) \rightarrow \text{preOrdLstToTreAux}(T, \text{insert}(3, \{e,e,e\}, \text{Tree}, \text{Idx}), \text{lists:append}(\text{Idx}, \{3,1\}));
\text{preOrdLstToTreAux}([4|T], \text{Tree}, \text{Idx}) \rightarrow \text{preOrdLstToTreAux}(T, \text{insert}(4, \{e,e,e,e\}, \text{Tree}, \text{Idx}), \text{lists:append}(\text{Idx}, \{4,1\}));
\text{preOrdLstToTreAux}([5|T], \text{Tree}, \text{Idx}) \rightarrow \text{preOrdLstToTreAux}(T, \text{insert}(5, \{e,e,e,e,e\}, \text{Tree}, \text{Idx}), \text{lists:append}(\text{Idx}, \{5,1\})).
\]

% \text{rmvFldIncLst}(\text{IndexList}) \rightarrow \text{IndexList}'
% remove filled nodes from the last to the first & increment the last node's index.

\[
\text{rmvFldIncLst}(L) \rightarrow \text{lists:reverse}(\text{rmvFldIncLstAux}(\text{lists:reverse}(L))).
\]

% Inserts in time proportional to the height of the tree.
\[
\text{insert}(\text{Node}, e, []) \rightarrow \text{Node};
\text{insert}(\text{Node}, 1, \{1,1\} | T) \rightarrow 1, \text{insert}(\text{Node}, A, T);\]
\[
\text{insert}(\text{Node}, 2, A, B, \{2,1\} | T) \rightarrow 2, \text{insert}(\text{Node}, A, T, B);\]

insert (Node, {2, A, B}, [(2, 2) | T]) -> (2, A, insert (Node, B, T));
insert (Node, {3, A, B, C}, [(3, 1) | T]) -> {3, insert (Node, A, T), B, C};
insert (Node, {3, A, B, C}, [(3, 2) | T]) -> {3, A, insert (Node, B, T), C};
insert (Node, {3, A, B, C}, [(3, 3) | T]) -> {3, A, B, insert (Node, C, T)};
insert (Node, {4, A, B, C, D}, [(4, 1) | T]) -> {4, insert (Node, A, T), B, C, D};
insert (Node, {4, A, B, C, D}, [(4, 2) | T]) -> {4, A, insert (Node, B, T), C, D};
insert (Node, {4, A, B, C, D}, [(4, 3) | T]) -> {4, A, B, insert (Node, C, T), D};
insert (Node, {4, A, B, C, D}, [(4, 4) | T]) -> {4, A, B, C, insert (Node, D, T)};
insert (Node, {5, A, B, C, D, E}, [(5, 1) | T]) -> {5, insert (Node, A, T), B, C, D, E};
insert (Node, {5, A, B, C, D, E}, [(5, 2) | T]) -> {5, A, insert (Node, B, T), C, D, E};
insert (Node, {5, A, B, C, D, E}, [(5, 3) | T]) -> {5, A, B, insert (Node, C, T), D, E};
insert (Node, {5, A, B, C, D, E}, [(5, 4) | T]) -> {5, A, B, C, insert (Node, D, T), E};
insert (Node, {5, A, B, C, D, E}, [(5, 5) | T]) -> {5, A, B, C, D, insert (Node, E, T)}.

tstInsert() -> % fix index repn.
[insert(0,e,[[]]) == 0,
 insert(0,[1,e],[[1,1]]) == [1,0],
 insert(0,[1,{1,e}],[[1,1],[1,1]]) ==
 {1,[1,0]},
 insert(0,[2,e,e],[[2,1]]) == [2,0,e],
 insert(0,[2,e,e],[[2,2]]) == [2,e,0],
 insert(0,[2,{1,e},e],[[2,1],[1,1]]) ==
 {2,[1,0],e},
 insert(0,[2,e,{{1,e}},{2,2}],[{1,1}]) ==
 {2,e,[1,0]},
 insert(0,[2,{{1,e}},{2,2}]) ==
 {2,{{1,e}},0},
 insert([{1,e},e,[]]) == [1,e],
 insert([{1,e},{1,e},{{1,e}}]) ==
 {1,[1,e]}].

runTst() ->
[ tstrmvFldInclstAux(),
 tstrmvFidIncLst(),
 tstrvstInsert(),
 tstrvstPreOrdLstToTre()
].

% abbreviations
% aux auxiliary
% fld filled
% inc increment
% ins insert
% lst list
% rmv remove
% tst test
% tst test
% ord order
% tre tree