A simple general purpose functional data structure

Gyanendra Aggarwal

1. Introduction

This data structure is very similar to Finger Trees (Ralf Hinze and Ross Paterson) in terms of its behavior but differs in actual implementation. This data structure also promises adding/removing of elements at the end points in amortized constant time. This data structure is general enough to support deque, ordered sequence (unique/non-unique), ordered map (unique/non-unique), priority queue etc.

This can be used in place of any sequence (list) that is either ordered by the position or value of its elements.

This has been implemented in Erlang as a behavior and it has 3 callback functions. These callback functions are provided by specific implementations. We use callback functions to customize this general purpose data structure to behave like deque, ordered sequence, ordered map and priority queue.

2. Callback function

These 3 callback functions, each specific implementation needs to provide.

measure : it takes 1 argument (entry) as input and returns a comparable value (measure of the entry).

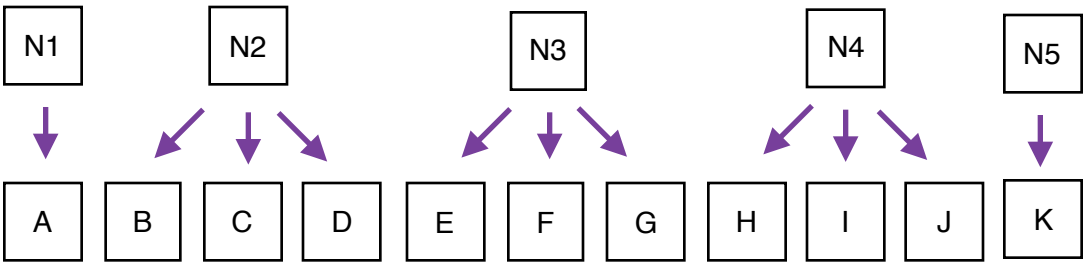
null : it takes no arguments as input and returns a comparable value (measure). Normally, we use this function to get an initial value.

add : it takes 2 arguments (2 measures) and returns a comparable value (measure).

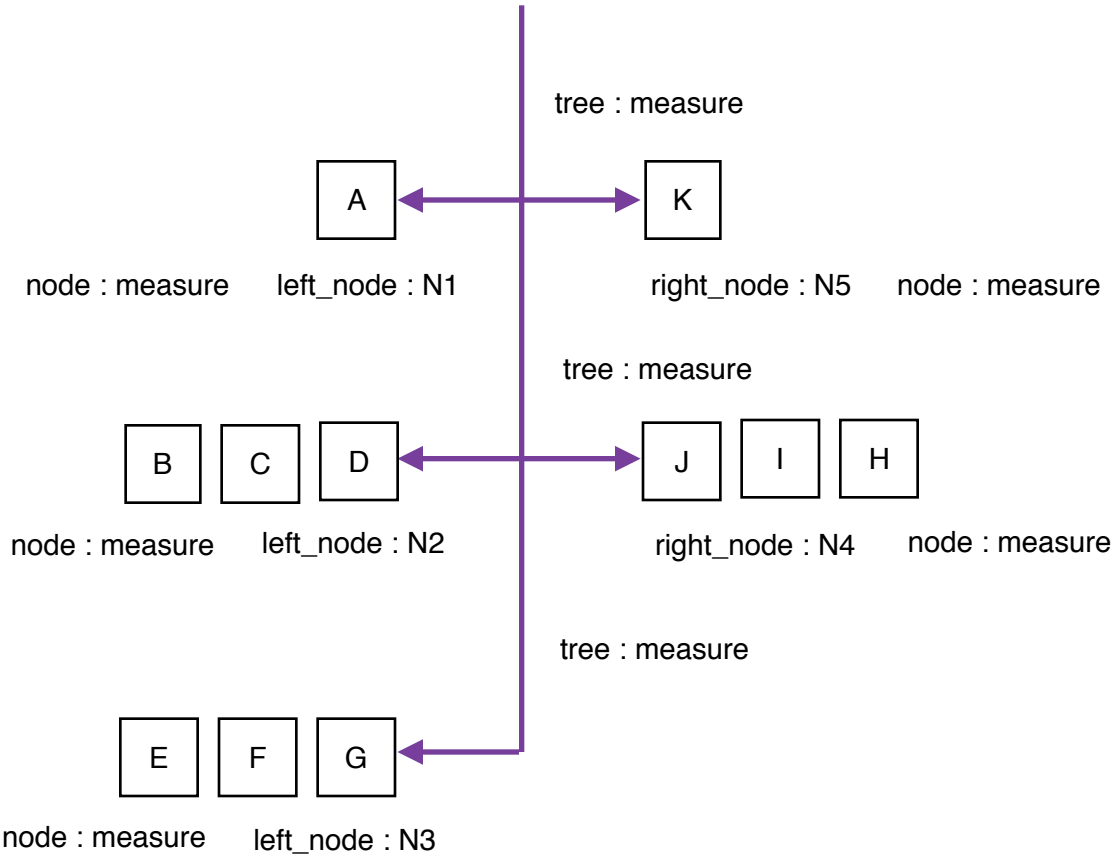
3. Properties

If we have an ordered sequence (list) as given below:

We group the elements of this sequence into groups of 3 elements into a node. Here 3 is the node size of a node and you can choose any node size. We have 11 elements and we make 5 nodes, although we could have managed it with 4 nodes.



This is a horizontal structure. We fold this structure from the middle and make it a vertical structure as given below:



Note : right_node has the list in reverse order

We keep the following information with nodes and trees:

Node :

type	: left_node l right_node
measured	: comparable value l undefined
list	: list of elements (must have at least 1 element)

Tree :

type	: to distinguish the trees of different type (deque, ordered_seq etc.)
node_size	: max. no. of elements a node can have
attribute	: this is an optional field, it is required to further distinguish the trees of same type (unique or non-unique seq/map) l undefined
measured	: comparable value l undefined
left_node	: left node of the tree l undefined
trunk	: sub-tree below the tree l undefined
right_node	: right node of the tree l undefined

This new data structure (tree) must maintain the following 5 properties

1. Top nodes of the tree (like N1, N5) will have minimum 1 element and maximum of *node_size* elements.
2. Number of elements in all other nodes will equal to *node_size.*
3. If a tree has a sub-tree (*trunk*) then it must have a left node and a right node
4. If a tree does not have a sub-tree (*trunk)* then it need not have both left and right node.
5. Tree must maintain a node order, top left node is the lowest node and next left node (if any) below the a left node is higher. Similarly, top right node is the highest node and next right node (if any) below the a right node is lower. Like in our example N1 is the lowest node and N5 is the highest node. The node order from low to high is N1, N2, N3, N4 and N5.

4. Operation supported

We support the functions on this data structure

cons_l	**:** This function will insert an element from the left side of the tree.
cons_r	**:** This function will insert an element from the right side of the tree.
uncons_l	**:** This function will remove 1 element from the left side of the tree.
uncons_r	**:** This function will remove 1 element from the right side of the tree.
to_list_l	**:** This function will create a list by traversing the tree from left.
to_list_r	**:** This function will create a list by traversing the tree from right.
from_list_l	**:** This function will create a tree from a list by inserting the elements from the left side of the tree.
from_list_r	**:** This function will create a tree from a list by inserting the elements from the right side of the tree.
split	**:** This function will split a tree (based on a predicate supplied) into 2 trees, one that satisfies the predicate and other that does not satisfy the predicate.
join	**:** This function joins 2 trees to form a single tree.

All these operations create a new tree with the exception of **to_list_l** and **to_list_r** and maintain the 5 properties mentioned above.