8-1-2012

Message Passing Algorithm for Different Problems Sum, Mean, Guide and Sorting in a Rooted Tree Network.

Sabaresh Nageswara Rao Maddula
University of Nevada, Las Vegas, maddulas@unlv.nevada.edu

MESSAGE PASSING ALGORITHMS FOR DIFFERENT PROBLEMS SUM, MEAN, GUIDE AND SORTING IN A ROOTED TREE NETWORK.

by

Sabaresh N Maddula

A Thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
August 2012

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Sabaresh N. Maddula

entitled

Message Passing Algorithms for Different Problems Sum, Mean, Guide and Sorting in a Rooted Tree Network.

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
School of Computer Science

Ajoy K. Datta, Committee Chair

Lawrence Larmore, Committee Member

Juyeon Jo, Committee Member

Emma Regentova, Graduate College Representative

Ronald Smith, Ph. D., Vice President for Research and Graduate Studies
and Dean of the Graduate College

August 2012

ABSTRACT

MESSAGE PASSING ALGORITHMS FOR DIFFERENT PROBLEMS SUM, MEAN, GUIDE AND SORTING PROBLEMS IN A ROOTED TREE NETWORK.

by

SABARESH N MADDULA
Dr . Ajoy K. Datta, Examination Committee Chair
School of Computer Science
University of Nevada, Las Vegas

In this thesis, we give message passing algorithms in distributed environment for five different problems of a rooted tree $\mathcal{T}$ having *n* nodes. In the first algorithm, every node has a value; the root calculates the sum of those values, and sends it to all the nodes in the network. In the second algorithm, the root computes the value of mean of values of all the nodes, and sends it to all nodes of the network. The third algorithm calculates the guide pairs. Guide pair of a node x is an ordered pair (pre_index(x), post_index(x)), where pre_index(x) and post_index(x) are the rank of x in the preorder and reverse postorder traversal of T. In the fourth algorithm, we compute the rank of all the nodes in the tree by considering the weight (value) present at every node. Finally, in the fifth algorithm, values present in the nodes are sorted in level order.

ACKNOWLEDGEMENTS

Firstly I would like to thank my guide, mentor and advisor Dr. Ajoy K Datta who is guiding, helping and supporting me throughout my thesis and entire masters program. I am indebted to him for his guidance and support.

I would like to thank Dr. Lawrence L Larmore, Dr.Ju-Yeon Jo, Dr. Emma Regentova for being as my committee members.

I would also like to thank Computer Science for supporting me financially with the Graduate Assistantship.

Finally I would thank my parents and family members for their support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In this thesis we give message passing algorithms in distributed environment for five different problems of a rooted tree $\mathcal{T}$ having n nodes. Initially the topology may not be a tree. But we make an assumption that the tree $\mathcal{T}$ is constructed using a self-stabilized silent algorithm. Any of these five algorithms do not assume the knowledge of n. We don't have any assumption that the nodes in the tree have unique ID's. But we assume that the tree $\mathcal{T}$ has a distinguished root process Root. We consider that every process P has a variable P.Parent, which points its neighbor process. If the node is a root node P.Parent points to itself.

The First algorithm SUM, all the process in the tree has a value V(P), it may be calculated by some algorithm or given by some application. Sum of all these values are calculated by the root i.e $\sum_P V(P)$ by sending all these messages to the root. This algorithm SUM can be used to combine the values which has an associative and commutative operations.

Our second algorithm MEAN calculates the mean of all the values present at every node of the tree i.e $1/n \sum_P V(P)$. Here root finds the number of nodes in the tree along with the sum and calculates the mean value.

The third algorithm GUIDE computes the guide pair value and it makes use of SUM algorithm for calculating the guide pair value. Guide pair for a process is written as P.guide = (P.pre_index, P.post_index). Figure 5.1 shows a rooted tree labeled with guide

pairs. Tree can be traversed easily by using the guide pair values. Partial order is defined on the guide pair values: we say (i,j) ▪ (k,l) if i ▪ k and j▪ l. For a process Q to be a member of subtree $\mathcal{T}$p rooted at P if and only if P.guide ▪ Q.guide.

Our fourth algorithm RANK computes the rank of every node in the tree based on the values V(P) present at every node in the tree. RANK uses the GUIDE algorithm guide pair values for traversing the tree.

The fifth algorithm LEVEL ORDER SORTING computes the rank of every node in the tree based on the values present at every node in the tree. Here the sorting is done in level order. This algorithm makes use of GUIDE algorithm for traversing the tree.

CHAPTER 2

BACKGROUND

In this chapter, we will give a brief explanation of Distributed Systems. In the first section, we will give explanation of distributed systems. Later in the second section, we will discuss about the message passing system.

2.1 Distributed Systems

A distributed system is a collection of some computing devices that can communicate with each other. It includes a wide range of computer systems, these systems range from a VLSI chip, to a tightly-coupled shared memory multiprocessor, to a local-area cluster of workstation, to the Internet. The motivation for using a distributed system may include inherently distributed computations, resource sharing, access to geographically remote data and resources, enhanced reliability, increased performance/cost ratio, and scalability. Every computer has a memory-processing unit, and the computers are connected by a communication network. These processors need to communicate with each other in order to achieve some level of coordination to complete a task. The different types of communication among these processors; Message Passing and Shared Memory. Shared memory systems are those in which there is a shared address space throughout the system. Communication among processors takes place via shared data variables and control variables. In Message passing systems, the processors communicate by sending and receiving messages through the links in the network.

2.2 Message Passing System

Here we will present a brief overview of message passing interface in distributed computing. Message passing is a method of communication used in parallel computing , interprocess communication and object-oriented programming. In this system process or nodes in the network communicate by sending and receiving messages to each other. Synchronization can be achieved by waiting for messages.

In message passing paradigm of communication sender can send messages to more number of recipients. The message can be sent in different forms signals, remote method invocation (RMI), data packets. We need to consider different conditions while designing a message passing system.

- Reliability of messages i.e. knowing if the message is sent or not.
- Guaranteed delivery of messages in order.
- Different scenarios are to be considered such as the messages are passed one-to-one(unicast), one-to-many(multicast), many-to-one(client to server), many-to-many(AllToAll).
- Different modes of communication whether it is a synchronous mode of communication or asynchronous mode of communication.

2.2.1 Message Passing Applications

Remote Method Invocation(RMI) and Distributed Object Systems like Cobra, Java RMI, SOAP, CTOS, .Net Remoting systems etc are different message passing

systems. This message passing system can be called as "shared nothing" system as this systems abstraction hides the underlying state changes which are used in message sending.

Message passing model based programming languages generally define messages as the(normally asynchronous) sending(copy) of data to the user(processer, node) at the other end. This kind of messaging is used in Web Services by SOAP. This is higher level version of datagram with an exception that the size of the message cannot be larger than size of the packet. They may or may not be reliable, secure, durable and transacted.

2.2.2 Modes of Message Passing

Synchronous and Asynchronous modes are the two different modes available in message passing systems. In synchronous mode of communication both the sender and receiver have to wait for each other for the transfer of messages. This mode of communication have different advantages.

- Reasoning about the program can be simplified in that there is synchronizing point between the sender and the receiver.
- In synchronized mode of communication buffer is not necessary.

In asynchronous mode of communication for message passing systems sender and the receiver need not wait for the messages. Sender can send the messages without the receiver waiting for the messages. This mode of communication in message passing systems have some disadvantages. If the buffer in asynchronous system is full this will

cause some problems. If the messages being sent are lost the communication will no longer be reliable in this system. If there is a situation weather to block the sender or discard the new messages then this may lead to deadlock. We can also implement synchronous mode of communication over the asynchronous mode of communication by making the messages to wait.

2.2.3 Message Passing and locks

Access to resources in asynchronous or concurrent systems can be controlled using this message passing systems. The resources can shared memory, database table rows, files in a disk. A process must initially acquire the lock in order to access the resources. If a process acquires a lock all other process will be blocked and they cannot access the resources anymore. Once the process with the lock is done with the usage of resources then it will release the lock and make the resources available to other processes. This message passing is one of the main alternative solution for mutual exclusion.

2.3 Message Passing System vs. Shared Memory Systems

Message passing and Shared memory are different communication systems in distributed environment. Their performance characteristics will vary in different ways. In Message passing system the task switching and per-process memory overhead is low, but the overhead of message passing itself is greater than for a procedure call. There are other overwhelming performance factors which shows better performance of message passing system. When we consider the scalability factor if there are more number of systems in a network then message passing system works better in that environment. We can apparently see that message passing system is the preferred way to increase the number

of processor managed in a multi processor system. The memory is being shared in a shared memory and multiple process share the same data. This may cause many concurrency issues. But in message passing system all the process communicate using messages and this keeps the process separated. Process cannot modify each other's data.

CHAPTER 3

3.SUM ALGORITHM

SUM, assumes each process in the tree is given a value *V (P),* it can be given by any application, or it can be computed by some other algorithm. SUM then computes $\sum_P V(P)$, the sum of all the values at all the processes. The algorithm SUM can actually be used to combine the values of any type which has an associative and commutative operation. For example, if every process is given a value belonging to any ordered type, we could use an appropriate variant of SUM to compute the maximum or the minimum value in *T*.

Suppose that there is a connected network with a single designated process Root, where all other processes are anonymous. Every process is given a value *V(P)*. This value *V (P),* can either be initialized at every process or it can be calculated. We can allow the application layer to change *V (P)* for any *P* at any time and this *V(P)* need not be stable.

We are given a commutative associative operator on values, which we call addition and write as "+." Of course, this operator could actually be ordinary addition; or it could be multiplication, minimum, or maximum, to name a few well-known possibilities. The sum problem is to compute the "sum" of the values of all the processes in the network, which we write as *V (P),* and to inform every process of this sum.

SUM algorithm in this section solves the sum problem, by finding the correct value of $\sum_P V(P)$ within O(diam) rounds, provided the values of the processes remain unchanged during that time. SUM is also self-stabilizing and silent, and works under the unfair daemon. If SUM has converged, every process knows the sum. If the application

layer changes *V (P)* for one or more processes *P*, then SUM will "wake up" and recalculate the sum, then SUM will become silent again after every process in the tree know the new sum.

3.1 General Overview of SUM

SUM consists of a bottom-up wave that computes the sum of the values of the processes in each subtree of the BFS tree, followed by a top-down wave that informs every process what the value of the sum is. Initially all the leaf nodes starts a bottom-up wave by sending a message V(P) to its parent. All the other processes calculate *Subtreesum(P)* and send *subtreesum* value to its parents. After the process receiving the values from all its children it finds the *Subtreesum(P)* and sends to its parent and this wave continues until it reaches the Root. Then root has the value P.sumall i.e the sum of all values *V(P)* in the network.

Now root starts a broadcast wave by sending a message *P.sumall* i.e *V(P)* to all its children and this wave sends the sum of all *V(P)* i.e SUM to all the process in the network.

3.2 Variables of SUM

Each process P has the following variables.

1. *P.subtreesum.* The correct value of *P.subtreesum* is the sum of the values of V (Q) for all

Q Tp , the subtree of the BFS tree rooted at P.

2. *P.sumall* . The correct value of *P.sumall* is the sum of the values of V (Q) for all processes

Q in the network.

We will assume that P can read the local names of its neighbors, so that if Q N(P), P can tell

whether *Q.parent = P*.

3.3 Functions of SUM

Each process *P* can compute the following functions.

1. *Chldrn(P) = {Q N(P) : Q.parent = P}*

2. *Subtreesum(P) = V (P) + Q Chldrn(P) Q.subtreesum*

If P is a leaf of the BFS tree, then *Subtreesum(P) = V (P).*

3. *Sumall (P) = P.subtreesum* if *P = Root*

P.parent.sumall otherwise

3.4 Legitimate Configuration of SUM

A configuration of SUM is defined to be legitimate if the

following conditions hold.

1. All values of *subtreesum* are correct, that is *P.subtreesum = Q TpV(Q).*

2. All values of *sumall* are correct, that is *P.sumall = V (Q).*

3.5 Messages of SUM

- send *P.subtreesum*. Each process *P* sends *P.subtreesum* to its parent where

Subtreesum(P) = V (P) + Q Chldrn(P) Q.subtreesum

- send *P.subtreesize.* Each process *P* sends *P.subtreesize* to its parent where

  Subtreesize(P) = 1 + $\sum_{Q \in Chldrn(P)} Q.subtreesize$

- *P*.receive *Q.subtreesum.* Process *P* receives *V(P)* from all its children.

- send *P.sumall(P).* Starting from Root every process sends the *sumall* to all its children where

  Sumall (P) = P.subtreesum if *P = Root*

  P.parent.Sumall otherwise

3.6 Explanation of Sum

Every node of the rooted tree *Tp* has a value *V(P).* Starting from the nodes every node sends its *subtreesum* to its parent. This is done in the form of a convergecast wave starting from the leaf nodes. This convergecast wave ends after reaching the Root node.

Now the Root node has the values of *subtreesum* of the tree. Root calculates the value of the *Sumall(P)* for the tree i.e *Sumall(P) = Subtreesum(P).* Root sends the *sumall* value to all the children. This is send in the form of a broadcast wave. This wave ends when it reaches the leaf nodes. Then every node in the tree has the Sum Value.

3.7 SUM ALGORITHM STEPS

A1 *P=Leaf*

Subtreesum(P)=V(P) *P* send *P.subtreesum* to *P.Parent*

A2 *QCp* send *Q.subtreesum* to *Q.Parent*

P.receive *Q.subtreesum*

P.receivedQ ←True

A3 *QCp* if *P.receivedQ*=True

ubtreesum(P)=V(P)+ QCp Q.subtreesum

send *P.subtreesum* to *P.Parent*

A4 *P=Root* *Sumall(P)=Subtreesum(P)*

QCp send *P.sumall*

to *Q*

A5 *P Root* receive *P.Parent.sumall*

Sumall(P)=P.Parent.sumall

CHAPTER 4

4 . MEAN ALGORITHM

MEAN, which computes the average of all the values present at different process. Every process in the network knows the value *V(P)*, the network has n processes . Here we will compute the mean value of *V* , namely *1/n* $\sum V(P)$, where n is the size of the network.

4.1 General Overview of MEAN

MEAN algorithm starts with a convergecast wave starting from the leafs sends messages *subtreesum*, *subtreesize* to its parents i.e. *V(P)*, 1 to its parent. Now all the process after receiving the messages from its children calculates *Subtreesum(P)*, *Subtreesize(P)* values and send those values to its parent and this wave continues until it reaches the root. Now the root has the values of *subtreesum* and *subtreesize* for the entire tree. Then Root calculates the value Mean *P.subtreesum/P.subtreesize*.

Now a broadcast wave starting from the root sends a message meanvalue to all its children and those process sends the meanvalue message to its children. This wave continues until it reaches the leaves. Now all the process in the tree has the value of Mean for the entire tree.

4.2 Variables of MEAN

Each process P has the following variables.

1. *P.subtreesum*, as in SUM.

2. *P.subtreesize*. The correct value of *P.subtreesize* is the number of processes in the subtree of the BFS tree rooted at *P*.

3. *P.meanvalue*. The correct value of *P.meanvalue* is the mean value of all the processes in the

network.

We will assume that P can read the local names of its neighbors, so that if $Q \in N(P)$, *P* can tell

whether *Q.parent = P*.

4.3 Functions of MEAN

Each process P can compute the following functions.

1. $Chldrn(P) = \{Q \in N(P) : Q.parent = P\}$

2. $Subtreesum(P) = V(P) + \sum_{Q \in Chldrn(P)} Q.subtreesum$

If *P* is a leaf of the BFS tree, then $Subtreesum(P) = V(P)$.

3. $Subtreesize(P) = 1 + \sum_{Q \in Chldrn(P)} Q.subtreesize$

If *P* is a leaf of the BFS tree, then $Subtreesize(P) = 1$.

4. $Meanvalue(P) = P.subtreesum/P.subtreesize$ if $P = Root$

$P.parent.meanvalue$ otherwise

4.4 Messages of MEAN

- send *P.subtreesum*. Each process *P* sends *subtreesum* to its parent where

  $Subtreesum(P) = V(P) + \sum_{Q \in Chldrn(P)} Q.subtreesum$

- send *P.subtreesize.* Each process *P* sends *subtreesize* to its parent where

  $Subtreesize(P) = 1 + \sum_{Q \in Chldrn(P)} Q.subtreesize$

- send *P.meanvalue.* Each process *P* sends *meanvalue* to all its children where

$$Meanvalue(P) = P.subtreesum/P.subtreesize$$

4.5 Legitimate Configuration of MEAN

A configuration of MEAN is defined to be legitimate if the following conditions hold.

1. All values of *subtreesum* are correct, that is $P.subtreesum = \sum_{Q \in Tp} V(P)$

2. All values of *subtreesize* are correct, that is *P.subtreesize* is the number of processes in *Tp* .

3. All values of *meanvalue* are correct, that is $P.meanvalue = 1/n \sum_{Q \in TV(Q)}$.

4.6 Explanation of Mean

Every node of the rooted tree *Tp* has a value *V(P)*. Starting from the leaf nodes every node sends its *subtreesum* and *subtreesize* to its parent. This is done in the form of a convergecast wave starting from the leaf nodes. This convergecast wave ends after reaching the Root node.

Now the Root node has the values of *subtreesum* and *subtreesize* of the tree. Root calculates the value of the Mean for the tree i.e *Meanvalue(P)=Subtreesum(P)/Subtreesize(P).* Root sends the Mean value to all the children. This is send in the form of a broadcast wave. This wave ends when it reaches the leaf node. Then every node in the tree has the Mean Value.

4.7 MEAN ALGORITHMS STEPS

A1 *P=Leaf* *P.subtreesum=V(P)*

P.subtreesize=1

Q send *P.subtreesum to P.Parent*

send *P.subtreesize to P.Parent*

A2 *Q Cp* send *Q.subtreesum to Q.Parent*

send *Q.subtreesize to Q.Parent*

P.receive *Q.subtreesum*

P.receive *Q.subtreesize*

P.receivedQ ← True

A3 *Q Cp* *Subtreesum(P)=V(P)+ Q Cp Q.subtreesum*

if *P*.received*Q*=True *Subtreesize(P)=1+ Q Cp Q.subtreesize*

send *P.subtreesum* to *P.Parent*

send *P.subtreesize* to *P.Parent*

A4 *P=Root* *Meanvalue(P) = Subtreesum(P)/Subtreesize(P)*

Q Cp send *P.meanvalue* to *Q*

A5 *P Root* receive *P.Parent.meanvalue*

Meanvalue(P)=P.Parent.meanvalue

5. GUIDE ALGORITHM

A rooted tree T is an ordered tree, if the children of every node are ordered together with an order (called a left-to-right order). Let *P1, P2, . . . Pm* be the children of the root of *T* in left-to-right order, and let *Ti* be the subtree rooted at *Pi*.

Preorder traversal of a tree *T* is defined, recursively, as follows.

1. Visit the root of *T* .

2. For each *i* from 1 to m in increasing order, visit the nodes of *Ti* in preorder.

Post-order traversal of a tree *T* is similarly defined.

1. For each i from 1 to m in increasing order, visit the nodes of *Ti* in postorder.

2. Visit the root of *T* .

Pre-order traversal is top-down, while post-order is bottom-up. However, we can also traverse *T*

is *reverse post-order*, which is top-down, as follows.

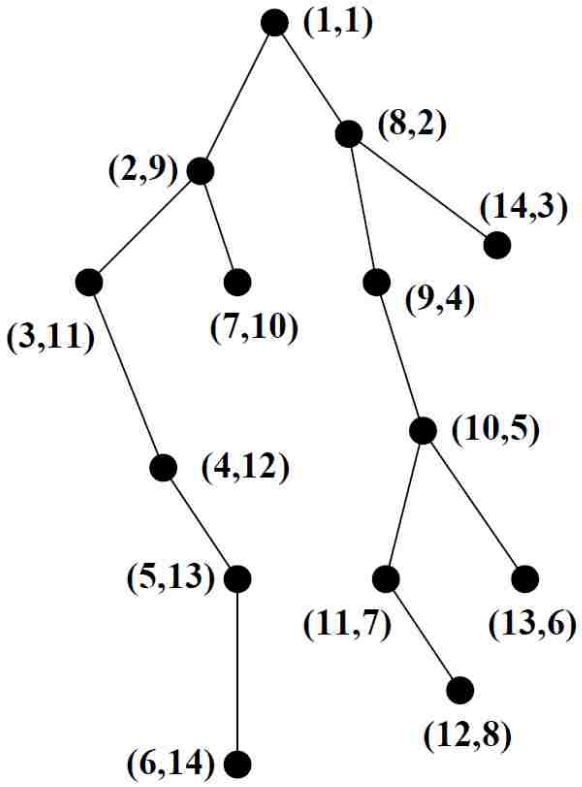
1. Visit the root of *T* .

2. For *i* from *m* to 1 in decreasing order, visit the nodes of T_i in reverse postorder.

In preorder traversal of *T* if a node x is the i^{th} node visited, we say that the preorder rank of x is *i*. The reverse postorder rank of x is *j* if the node x is the jth node visited in reverse post-order. We write *pre_index(x)* for the preorder rank and *post_index(x)* reverse postorder rank of x, respectively, We define the guide pair of x to be the ordered pair guide(x) = *(pre index (x), post index (x))*. Figure 5.0.1 shows an ordered tree where each process is labeled with its guide pair.

If *(i, j)* and *(k, l)* are guide pairs, we write *(i, j)* ▪ *(k, l)* if *i* ▪ *k* and *j* ▪ *l*. Thus, the set of guide

pairs is partially ordered.



(fig 5.0.1) An ordered tree, labeled with guide pairs.

5.1 General overview of GUIDE.

We now describe the algorithm, GUIDE, which computes the guide pairs of all processes in a rooted tree network. In the first phase of GUIDE is to compute the sizes of all subtrees, by applying Algorithm MEAN. Later in the second phase is to assign index to every children in the tree. Each child must know the index assigned to it by its parent.

This index is assigned in left to right order. Each process then computes the guide pair values starting from the root. Guide pair value of the root is (1,1).

In the first phase of GUIDE all the node in the computes the *subtreesize* by receiving the messages *subtreesize* from all its children i.e all the node starting from the leaf nodes sends message *subtreesize* to their parent. GUIDE uses MEAN algorithm for the calculation of *subtreesize*. After the parent receiving the message from its children it calculates *Subtreesize(P).* The values of *pre_index* and *post_index* of root are set to 1.

In the second phase, GUIDE computes the guide pair for each process. GUIDE uses *subtreesize* values for computing the guide pair values. Now every node assigns indexes to all its children and the children must know their indexes. Guide pair value of root is set to (1,1) GUIDE computes the values *P.chld_pre_predecessors[i]* and *P.chld_post_predecessors[i]* for each *i* in the range *1 . . . P.num_chld. Num_Preorder_Predecessors(i)* is calculated by adding the subtreesize of leftmost subtree of P.parent. Each process P then computes its guide pair value *(P.parent.chld_pre_predecessors[j] + 1, P.parent.chld_post_predecessors[j] + 1)* where *j* is the index of child P in its left-to-right order.

After GUIDE computing the values *P.child_pre_predecessors* and *P.child_post_predecessors*. If the index of the child is 1 then *Num_Preorder_Predecessors(P) = P.parent.pre_index* , this value is directly sent to the child and for all other children to *P.parent.chld_pre_predecessors[1]*. For all other children parent sends the message *P. chld_pre_predecessor[i], P.chld_post_predecessor[i]* to its children where *i* is the order of the children. And the guide pair of the node is calculated by

P.pre_index ← *1+P.parent.chld_pre_predecessors[P.my_order]*

P.post _index ← *1+P.parent.chld_post_predecessors[P.my_order]*

5.2 Variables of GUIDE

1. *P.subtreesize,* is the number of processes present in a tree T_P rooted at *P* .

Let *Chldrn(P) = {Q* ∈ *N(P) : Q.parent = P},* the children of P in the tree *T* . These children

are then ordered left-to-right, using the neighbor ordering of *P*, making *T* an ordered tree. The

following variables enable each process to know the local state of the ordered tree.

2. *P.num_chld,* an integer in the range *0 . . . δp,* whose correct value is the number of children

of *P* in the BFS tree *T* .

3. *P.child[i]* ∈ *N(P) U {*⊥*}* for *1* ≤ *i* ≤ *δp* . If *1* ≤ *i* ≤ *P.num_chld,* then *P.child[i]* is the *ith*

child in *P*'s local ordering of *N(P),* while *P.child[i] =* ⊥ *if i > P.num_chld.*

4. *P.my_order.* If *P* ≠ *Root,* then the correct value of *P.my_order* is *i* if *P.parent.child[i] = P,*

while the correct value of *Root.my_order* is ⊥.

5. *P.pre_index , P.post_index ,* integers, whose correct values are the preorder and reverse postorder ranks of *P*, respectively. We will write *P.guide = (P.pre_index , P.post_index).*

6. *P.chld_pre_predecessors[i]*, *P.chld_post_predecessors[i]*,integer, for *1* ≤ *i* ≤ *P.num_chld*.

The correct value of *P.chld_pre_predecessors* is the number of predecessors of *P.child[i]* in

the preorder traversal of *T* . The correct value of *P.chld_post_predecessors* is the number of

predecessors of *P.child[i]* in the reverse postorder traversal of *T* .

5.3 Messages

- send *subtreesize.* Each process *P* sends *subtreesize* to its parent where

$$subtreesize = 1 + \sum_{Q \in Chldrn(P)} Q.subtreesize$$

- send *P. chld_pre_predecessor[i]*. Each process *P* sends the value *chld_pre_predecessor[i]* to its children where *i* is the index of children. It is used to calculate the uide pair value.
- send *P.chld_post_predecessor[i]*. Each process *P* sends the value *chld_post_predecessor[i]* to its children where *i* is the index of children. It is used to calculate the guide pair value.

5.4 Explanation of GUIDE

Here we give a clear explanation how GUIDE computes the values *P.pre_index* for all the process. In the similar way GUIDE also computes the *P.post_index*. If P is the ith process visited in a preorder visitation of *T* . Then the correct value of *P.pre_index* is *i*. GUIDE computes the number of predecessors of *P,* i.e. the number of processes

visited before *P* is visited. Let us call that number *Num_Preorder_Predecessors(P)*. After the GUIDE computing the *Num_Preorder_Predecessors(i)* value it sends this value to the children of respective index where is the index and after the children receiving the message it calculates the *P.pre_index*

i.e. *P.pre_index* ← *Num_Preorder_Predecessors(P)*.

If P is the root process then *Num_Preorder_Predecessors(Root) = 0*, otherwise the value *Num_Preorder_Predecessors(P)* is calculated by the parent and stores the value in the variable *P.parent.chld_pre_predecessors[i]*, where *P* is the *i*th child of its parent in left-to-right order. For computing all these values of its children, P.parent must have computed its own value of *pre_index* , it should also know all its subtree sizes. If *i = 1*, then *Num_Preorder_Predecessors(P) = P.parent.pre_index* , because here its parent is the immediate predecessor of its leftmost child in the preorder traversal. So it can directly send the value *P.parent.pre_index = Num_Preorder_Predecessors(P).*Thus *P.parent.chld_pre_predecessors[1]* is equal to *P.parent.pre_index.* This value can be directly sent to the first child. But for *P.parent.chld_pre_predecessors[2]* is obtained by adding the size of the leftmost subtree of *P.parent* to *P.parent.chld_pre_predecessors[1]*, since all members of that subtree are predecessors of the second child of P.parent.

In general, the number of predecessors of P is equal to *P.parent.pre_index* of its parent plus the sum of the sizes of the leftmost *i −1* subtrees of its parent. Where *i* is the index. The values of *P.post_index* are computed from right to left, in a similar manner. After computing both the values *P.parent.chld_pre_predecessors[i]*, *P.parent.chld_post_predecessors[i]*. It sends these values to its children respectively. Then P computes the guide pair value by computing the below values.

P.pre_index ← *P.parent.chld_pre_predecessors[j] + 1*

P.post_index ← *P.parent.chld_post_predecessors[j] + 1*

5.5 GUIDE Algorithm Steps

A1 *P=Leaf* *subtreesize = 1*

send *P.subtreesize* to *P.Parent*

A2 ∀*Q*∈*Cp* receive *Q.subtreeize*

P.receivedQ ← True

A3 ∀*Q*∈*Cp* if *P*.received*Q*=True

Subtreesize(P)=1+ ∑*Q*∈*Cp Q.subtreesize*

A4 *P=Root* *P.pre_index* ← *1*

P.post_index ← *1*

A5 ∀*P* *P.chld_pre_predecessors[1]* ← *P.pre_index*

send *P.chld_pre_predecessors[1]*

P.chld_post_predecessors[P.num_chld] ← *P.post_index*

send *P.chld_post_predecessor[P.num_child]*

A6 ∀ *2*≤*i*≤ *P.num_child* *P.chld_pre_predecessor[i]* ←

P.child_pre_predecessors[i-1] + P.child[i-1].subtreesize

send *P. chld_pre_predecessor[i]*

A7 ∀ *Q* ∈ *Cp* receive

P.parent.chld_pre_predecessors[P.my_order]

Received *Q* = True

A8 received Q ← True *P.pre_index* ←

1+P.parent.chld_pre_predecessors[P.my_order]

A9 ∀ *1* ≤ *i* < *P.num_child* *P.chld_post_predecessor[i]*<-

P.child_post_predecessors[i+1] + P.child[i+1].subtreesize

send *P.chld_post_predecessor[i]*

A10 ∀ *Q* ∈ *Cp* Receive

P.parent.chld_post_predecessors[P.my_order]

Received *Q* = True

A11 If *P*.received *Q*← True *P.post _index* ←

1+P.parent.chld_post_predecessors [P.my_order]

CHAPTER 6

6. RANK ALGORITHM

Rank Ordering problem solves the ordering problem of a rooted tree. All the nodes in the tree *T* has a value *P.weight*. Based on this *weight* we need to find the rank of each process *P* if *P1, P2 . . . Pn* is the list of processes of *T* sorted according to their weight, the *i* is the rank of *Pi*.

6.1 Overview of RANK

When a new epoch starts all the Root node starts a broadcast wave by sending a message *status = 1* to all its children and changes its status to 1 from *status = 0 or 4*. When this broadcast wave has reached the leaf nodes then a new convergecast starts from leaf nodes sending message *status = 2* to its parent. This wave reaches the root node and it ends there. All the Rank computation goes when process are having *status = 2*. After the computation of Rank is done i.e after last down package is sent then the Root node starts a broadcast wave by sending a message *status = 3* to all its children. When this wave reaches the leaf nodes. A new convergecast wave starts from the leaf nodes by sending a message *status = 4* to its parents. A new epoch will begin after the convergecast wave.

When the status variable of all the process is 2. The actual computation of rank will take place. Initially all the leaf node process creates up-packages which is a combination of weight and guide pair value. Starting from the leaf nodes all the process sends this packages up until it reaches the root. There will be guard that makes sure that small weight packages goes first. After the package reaches the root. Root assigns a rank

to the process and includes in the down-package. The will be assigned with the help of a counter present in the root node. The combination of rank and guide pair is down-package. When this package reaches the home process i.e. the process having the guide pair value same as in the down-package that process will get assigned the rank. The process rank will be the rank present in the down-package.

6.2 Variables of RANK

Each process P has the following variables.

1. All the variables of GUIDE.

2. *P.up_pkg,* either of package type, or ▮▮ (undefined). If *P.up_pkg* is defined, its home process is some *Q*▪ *Tp.*

3. *P.down_pkg,* either of package type, or ▮▮ (undefined). If *P.down_pkg* is defined, its home process is some *Q*▪ *Tp* .

4. *P.started,* Boolean, which indicates that P has already generated an up-package during this epoch. (Of course, *P.up_pkg* may or may not still contain that up-package.)

5. *P.up_done,* Boolean, which indicates *Tp* contains no active up-package. (Of course, active p-packages whose home processes are in *Tp* could exist at processes above *P*.)

6. *P.status,* an integer in the range [0 . . . 4]. Status variables are used to control the order of computation, to correct errors.

7. Root contains an incrementing integer variable *counter,* which assigns the rank to packages. It is initialized to be 0 each time a new epoch begins.

6.3 Functions of RANK.

1. *Clean_State(P),* Boolean, which is true if *P* is in an initial, or "clean," state. Formally,

Clean_State(P) is true if all the following conditions hold:

(a) *P.up_pkg* = ∅

(b) *P.down_pkg* = ∅

(c) ¬*P.started*

(d) ¬*P.up done*

2. *Status_Error(P),* Boolean, which is true if *P* finds that its status is incorrect with those of its neighbors. Arbitrary initialization is the main reason for status error *Status_Error(P)* will eventually remain false all the time for all *P*. Formally, *Status-_Error(P)* holds if any one of the following conditions holds.

(a) *P.status* ∈ *{1, 3}* and *P.parent.status* ≠ *P.status*.

(b) *P.status* ∈ *{2, 4}* and *Q.status* ≠ *P.status* for some *Q* ∈ *Chldrn (P).*

(c) *P.status* ≠ *0* and *P.parent.status = 0.*

(d) *P.status* ∉*{0, 1}* and *Q.status = 0* for some *Q* ∈ *Chldrn(P).*

3. *Guide_Error(P),* is a Boolean variable, meaning that *P* can detect an error in the guide string of one of its packages. We say that a guide pair g is consistent with a process P if either g = *P.guide*, or there is some *Q* ∈ *Chldrn(P)* such that *g <= Q.guide.*

Formally, *Guide_Error(P)* is true if either of the following conditions holds:

(a) *P.up_pkg* ≠ ∅ and *P.up_pkg.guide* is not consistent with *P*.

(b) P.down_pkg ≠ ∅ and P.down_pkg.guide is not consistent with P.

4. *Error(P),* Boolean. Formally, *Error(P)* is true if any one of the following conditions holds:

(a) *Status_Error(P).*

(b) ¬*Clean_State(P)* ^ *(P.status = 1).*

(c) *Guide_Error(P)* ^ *(P.status = 2).*

(d) *P.up_done* ^ ¬*P.started* ^ *(P.status = 2).*

(e) *P.up_done* ^ *(P.status = 2),* and there is some *Q* ∈ *Chldrn(P)* such that ¬*Q.up_done.*

5. *Start_Pkg(P) = (P.weight, P.guide),* of package type, the up-package whose home process is *P,* and which *P* initiates.

6. *Up_Redundant(P),* Boolean, meaning that *P.up_pkg* is redundant. Formally,

If *P* ≠ *Root,* then *Up_Redundant(P)* holds provided the following three conditions hold:

(a) *P.up_pkg* ≠ ■,

(b) *P.parent.up_pkg* ≠ ■,

(c) *P.parent.up_pkg* ≥ *P.up_pkg*, which means that *P.parent* has already copied P.up_pkg.

Up_Redundant(Root) holds provided the following three conditions hold:

(a) *Root.up_pkg* ≠ ■,

(b) *Root.down_pkg* ≠ ■,

(c) *Root.down_pkg.guide = Root.up_pkg.guide.* This means that Root has already used its up-package to initialize a down-package. We note that *P* can evaluate *Up_Redundant(Q)* for any *Q* ∈ *Chldrn(P).*

7. *Down_Ready(P),* Boolean, meaning that *P.down_pkg* is redundant or undefined, and thus P can create or copy a new down-package.

Formally, if *P.down_pkg* = ■, then *Down_Ready(P)* holds.

If *P.down_pkg* ≠ ■ and *P.down_pkg.guide* ≠ ■ *P.guide,* then *Down_Ready(P)* holds if there is some *Q* ∈ *Chldrn(P)* such that *Q.down_pkg = P.down_pkg*.

If *P.down_pkg* ≠ ∎ and *P.down_pkg.guide = P.guide,* then *Down_Ready(P)* holds if *P.rank = P.down_pkg.value,* indicating that *P* has already copied the correct value of *P.rank.*

8. *Up_Done(P),* Boolean, meaning that *TP* contains no active up-packages. The value of *Up_Done(P)* is the correct value of *P.up_done*.

Formally, *Up_Done(P)* holds if the following three conditions hold:

(a) *P.started = 1*, meaning that P has already created an up-package.

(b) *Up_Redundant(P),* meaning that *P.up_pkg* is redundant.

(c) *Q.up_done* for all *Q* ▪ *Chldrn(P),* meaning that there are no active packages in any subtree of *P*.

9. *Can_Start(P),* Boolean, meaning that *P* can set *P.up_pkg* to *Start_ Pkg(P).*

Formally, *Can_Start(P)* is true if all the following conditions hold:

(a) ¬*P.started*, i.e., *P* has not get generated an up-package.

(b) *P.up_pkg* ≠ ∎ or *Up_Redundant(P).* This means that *P* is not holding an active up-package.

(c) For all *Q*▪ *Chldrn(P),* either ¬*Up_Redundant(Q)* or *Q.up_done.* This means that *P* can determine the smallest active up-package in *TQ*.

(d) For all *Q* ▪ *Chldrn(P),* either *Q.up_pkg > Start_Pkg(P)* or *Q.up_done*. This means that any active up-package in *TQ* is greater than *Start*▪ *Pkg(P).*

10. *Can_Copy_Up(P,Q),* Boolean, for *Q* ▪ *Chldrn(P). Can_Copy_Up(P,Q)* is true if *P* can copy *Q.up_pkg to P.up_pkg*.

Formally, *Can_Copy_Up(P,Q)* holds if all the following conditions hold:

(a) *Q.up_pkg* ≠ ∎ and ¬*Up_Redundant(Q).* This means that *Q* is holding an active up-package.

(b) *P.up_pkg* ≠ ∎ or *Up_Redundant(P)*. This means that *P* is not holding an active up-package.

(c) For all *R* ∈ *Chldrn(P),* either ¬*Up_Redundant(R)* or *R.up_done*. This means that *P* can determine the smallest active up-package in TR.

(d) For all *R* ∈ *Chldrn(P),* either *R.up_pkg* ≥ *Q.up_pkg* or *R.up_done.* This means that any active up-package in *TR* is greater than *Q.up_pkg*.

(e) *P.started,* or *Start_Pkg(P)* > *Q.up_pkg*. This means that *Start_Pkg(P)* has already gone up, or must wait for *Q.up_pkg* to go up first.

6.4 Messages of RANK

- send(status=0). Initially when the network has arbitrary initialization . If a process P find the status of a process incorrect. Then P starts sending the message send(status=0) to all its neighbours.
- send(status=1). A new epoch is initiated by the Root by changing its status to 1 and it sends a message send(status=1) to all its children. This message will end after reaching the leaf nodes.
- send(status=2). This message is initiated by the leaf nodes and it is send to its parent. It ends after reaching the Root node. In this epoch all the rank order processing is done.
- send(status=3). This message is initiated by the Root nodes when all the rank processing is done by the and this message ends when it reaches the leaf nodes.

- send(up_pkg). When there is an active up_pkg for a process then it sends the message to its parent. Guard ensures that the lower weight package is moved up.
- send(P.down_pkg). When there is an active down_pkg with a process then it sends the message to its children.

6.5 Explination of RANK

Initially the network is configured in a clean slate configuration i.e. for a process P *P.started = 0*, *P.up_done = 0*, and *P.up_pkg = P.down_pkg =* ∧, and *counter = 0*, the weight is given by the application. The values of the rank are not given since they are overwritten during the computation.

Flow of packages is the heart of Rank algorithm. Every package has an ordered pair *x = (x.value, x.guide),* where *x.value* is its weight value and *x.guide* is the guide pair value which is calculated using GUIDE algorithm. Every package has its own home process, Even if the package can be at any process in the chain between its home and the root. The guide pair of a package will always be same as the guide pair of its home process, but the value can either be the weight of its home process or the rank that RANK will assign to its home process.

Each process P initiates its flow of packages by creating an up-package whose value is *P.weight* and guide pair value. Initially all the leaf nodes creates the up-package value and send them to their parent. These packages are sent up until it reaches the root. This flow of packages is done in a way such that packages having the smaller weight reaches the root before the packages with larger weights. After the root receiving up-packages in order of their weights, it creates down-packages in order same as the up-packages. Suppose an up-package received by the root is created by the root is the *i*th

package then root creates a *i*th down-package with same guide pair value in the *i*th up-package. But its value will be *i* which will be the rank the process with that particular home process.

Once the root receives the up-package from a child, it creates a down-package with the same home process as the up-package, but the value in the down-package will be changed. It can be any number between the range *1 . . . n.* There will be a counter in the root which increments the value continuously for every down-package value. Starting from the first package having a value 1, second package having a value 2 and so on. Root sends this package to its children and this package will go down until it reaches home process.

The packages are guided to their home process with the help of their guide pair value. Once if the root knows that it has sent all its down-packages, then a broadcast wave is initiated which resets all the variables of *T* and a new epoch will be started.

In the computation of RANK, process P sends the package to its neighbor *Q*, still the package also remains at *P*. In the algorithm RANK, each process *P* can be home to at most one package, but when the packages are sent to their neighbors' the packages are getting redundant. We can eliminate that redundancy by defining a package variable currently held by a process (it need not be home process, it can be any process on the chain from its home to the root) is either active or redundant. If the package is redundant the package can be easily overwritten but if it is active package it cannot be overwritten.

If x is an up-package currently held by some process *Q* which is not the root, then x is redundant if x has already sent the package to *Q.parent*. If x is an up-package

currently held by the root, then x is redundant if the root has already created a down-package with the same guide pair as x. All other packages are considered as active.

If x is a down-package held by some process *Q* which is not its home process, then *Q* is considered as a redundant package if it has been sent to its child. After a process receiving its down-package that package is considered as redundant if the value in the package and the rank are same. This indicates that *P.rank* is correct before receiving its down-package or P has already received its down-package . All other down-packages are considered as active.

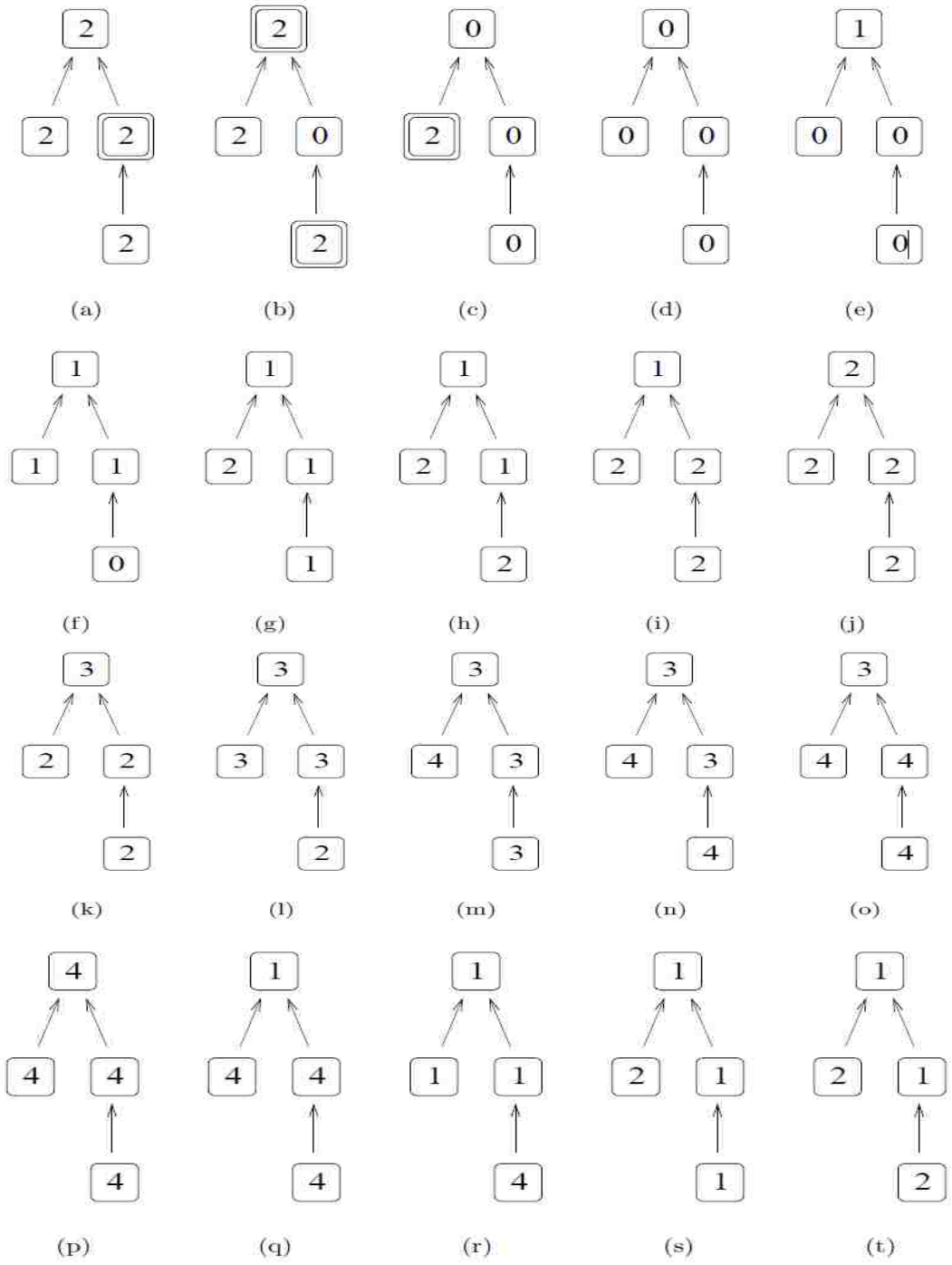
RANK is a distributed algorithm which is self-stabilizing, but not silent, it continuously computes the rank of all the process . Each cycle of this RANK computing is called as epoch. After each epoch is done variables except weight and rank for all the process will be reset. This variables will be used for the new epoch. If the epoch is not initialized arbitrarily and has a clean start, the rank value for each process will be computed correctly. All the new epochs after this will recalculate the rank and this will be same as in the previous calculation.

But if it configuration of network is initialized arbitrarily, the rank of the process may be calculated incorrectly. But eventually when a new epoch get started with a clean slate the correct values of rank can be computed.

The entire system is controlled by status variable. When a new epoch gets started, the root changes the status of every process from either 0 or 4 to 1 by sending a message to its children and all the node will change the status to 1, and all variables except rank and weight are set to their initial values. After the broadcast wave reaching the leaf nodes, a convergecast wave gets started from the leaf node sends messages to their

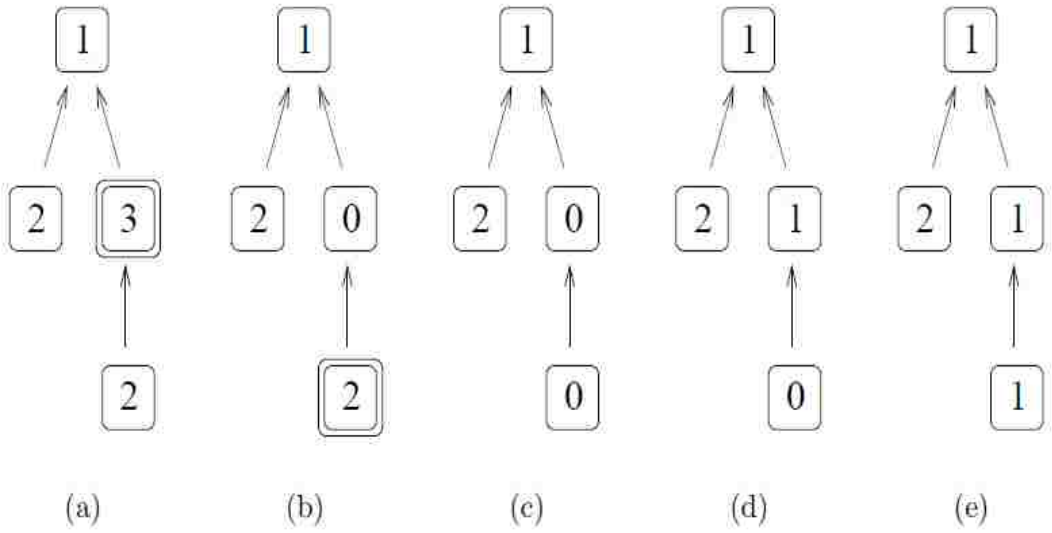
parents which changes the states of all the process to 2. Parent status will be changed to 2 after it receives message from all its children. Now the computation of the RANK begins when the status variable is 2 all the above discussed process will be done. After the root sending its last down-package, it initiates a broadcast wave by sending a message to its children which changes the status to 3 and this wave will be continued until it reaches the leaf nodes. The return convergecast wave then changes the status of all processes to 4 by sending a message to its parents from leaves until it reaches the root, and when this wave reaches the root, the new epoch begins.

Status zero can be used for error correction. If a process finds that the current epoch is having some error, then it changes its status to 0. This status 0 will be sent to all its neighbors until it finds the neighbors having status 1. If this status 0 reaches the root, then root creates a new epoch.



(6.7.1) Error correction diagrams.

Doubled box is used to show if there is any error in the diagram. In (a), all the process have status 2. (b) A process is found that its state is in error state, and it changes the status to 0. (c)Now this error state starts sending 0 message to all its neighbors (d) all the process changed its status to 0. (e) Since root has the status value 0 it begins a new epoch with a status 1.(g)With this new epoch the error in the process have been removed. Here the computation of RANK begins when all the process have the status 2. When Root knows that it is done with the RANK computation, it sends a broadcast wave with message 3 to all its children, as shown in (k). Then a new epoch begins (q).



(6.7.2) Changes the states to correction configuration.

(a) A process has a status error. (b) Process with the error state changes its status to 0, but this status 0 wave does not move in the up direction because its parent has status 1. But this status wave moves down, followed by the status 1 wave. (e) Here we can see that all the errors have remove and the RANK computation flows in a normal manner.

6.6 RANK Algorithm Steps

A1	If *Error(P),*	Then *P.Status* ← *0*
	∀*Q* ∈ *nebrs(P)*	send(status=0)
	∀*Q*	receive(status=0)

Initially the network may have any arbitrary initialization, it could be in any configuration. If some process detects that its state is erroneous , it initiates a resetting of the entire network by changing its status to 0 and send this message all its neighbors. This resetting continues until it finds a neighbor with status = 1.

A2	If *Root.status = 0 or 4* and	Then *Root.status* ← *1*
	∀*Chldrn(Root).Status = 0 or 4*	
	Root is set to Clean Slate.	
	∀*Q* ∈ *Chldrn(P)*	send(status=1)
	∀*Q*	recieve(status=1)

If the process in erroneous state is done with sending the status 0 to all its neighbors .If the has the status 0, then root starts sending a status message 1 to all its children with a broadcast wave. This broadcast wave stops after reaching the leaf nodes.

A3 If $P \neq Root$, $P.Parent.status = 1$ and Then

$\forall Q \in Chldrn(P)$ $P.receive(status = 1)$

$Q.Status = 0 \text{ or } 4$

Initial broadcast wave of a new epoch is being continued.

A4 $P = leaf\ node$, $\forall P \in Chldrn(Q)$ Then $P.status \leftarrow 2$

If Q.status = 1 P.send (status = 2)

Q.recieve(status =2)

A5 If $P.status = 1$ and $\forall Q \in Chldrn(P)$ Q.send(status=2)

$Q.status = 2$

$\forall P$ recieve(status=2)

All the children of a process have status 2 and the process is having status 1. Then the child sends a status message 2 to its children. This wave continues until it reaches the Root.

A6 If $P.status = 2$ and $Can\_start(P)$ Then

$P.up\_pkg.value \leftarrow P.weight$

$P.up\_pkg.guide \leftarrow P.guide$

We can write the statement of this action as $P.up\_pkg \leftarrow Start\_Pkg(P)$. When status variable value changes to 2. Then the actual computation of the rank starts. Starts from the leaf nodes the *up_pkg* values will be generated.

A7	If *P.status = 2* and *Can_Copy_Up(P,Q)* *Q* ∈ *Chldrn(P)*	Then *P.up_pkg* ← *Q.up_pkg* Q.send(*up_pkg*) P.recieve(*up_pkg*)

Q has an active up-package whose home is some process in T_Q, and package is sent to its parent. The guard ensures that the package moved up has a lower weight than any other active package in T_P. *Q* sends the *up_pkg* by sending a message *send(up_pkg)* and parent process receives the *recieve(up_pkg).*

A8	If *P.started = 1, Up_Redundant(P) = 1* And *Q.up_done = 1* ∀*Q* ∈ *Chldrn(P)*	Then *P.up_done* ← *1*

If P finds that T_P has no active up-package, then it sets *P.up_done* Boolean variable to true.

A9	If *Down_Ready(Root)* , *Root.up_Package* ≠ ⊥ ∧ *Up_Redundant(Root)*	Then *counter++* *Root.down_pkg.value* ← *counter* *Root.down_pkg.guide* ← *Root.up_pkg.guide*

Root has an active *up_pkg*, and *Root.down_pkg* is either ⊥ or redundant. Thus, Root is enabled to create a new down-package to send back down to the home process of its up-

package. If the value of counter is *i*, then *Root.up_pkg* is the *i*th up-package copied or created by Root, and its weight is the *i*th smallest weight in the network, and *i* will become the value of the down-package. Root sends the message send(*down_pkg*) to its children and its guided to the home process.

A10	If *P* ≠ *Root*, *Down_Ready(P)*, *P.parent.down_package* ≠ ∅ ¬*Down_Redundant(P.parent)* *P.parent.down_pkg.guide*	Then *P.down_pkg* ← *P.parent.down_pkg*
	Q = *parent(P)*	Q.send(*down_pkg*)
	P	receive(*P.down_pkg*)

P.parent holds an active down-package whose home is some process in T_P. *P.down_pkg* is either ∅ or redundant, then P.parent sends the message send(*down_pkg*). *P* is enabled to receive message recieve(*down_pkg*).

A11	If *P.down_pkg* ≠ ∅ *P.down_pkg.guide* = *P.guide* *P.down_pkg.value* ≠ *P.rank*	Then *P.rank* ← *P.down_pkg.value*

If *P.down_pkg.guide* is same as *P.guide* then *P* is its home process. Then *P* assigns the value to its rank from the value present in the *down_pkg*

A12	If *Root.up_done*,	Then *Root.status* ← *3*

Down_Ready(Root) and

Root.rank = Root.down_pkg.value

There can still be active down-packages below Root, but no up-package is active. Thus, Root is finished with its tasks for the current epoch.

A13	*R = Root*	Then *R.status* ←*3*
	S ∈ *Chldrn(R)*	*R.send(status=3)*
	R.status = 2	*receive(status=3)*
	S.status = 2	
	If *P* ≠ *Root,* and	*P.send(status = 3)*
	Q ∈ *Chldrn(P)*	*Q.receive(status=3)*
	P.status = 3	
	Q.status = 2	

Since *P*'s parent status is 3 and *P'*s status and its children status is 2 *P'*s parent will send a message send(status=3). *P* knows that its jobs for this epoch is done.

A14	If *P.status = 3*	*Q*.send(status=4)
	Q∈*Chldrn(P)*	*P*.receive(status =4)
	Q.status =4	

If all the children of *P* have status 4 its status is 3 the *P*'s status will change to 4 . i.e all the children of *P* will send a message send(status=4) then *P*'s status will change to 4.

CHAPTER 7

7. LEVEL ORDER SORTING

Given a rooted tree *T* where each node x has a "*value*", *x.value*. Let *r* be the root. We need to sort the values that every node is having. Finally every node will have a value *x.sortvalue*, its "sorted value". The set of all sorted values is the set of all values, but sorted in level order.

7.1 General overview of LEVEL ORDER SORTING

In LEVEL ORDER SORTING initially we need to find the level of each node i.e the distance of each node from the root. After each node having its level value it has to calculate the guide pair value. Guide pair is used for traversing the tree.

An ordered tree is a rooted tree *T* , together with an order (called a left-to-right order) on the children of every node. Let *P1, P2, . . . Pm* be the children of the root of *T* in left-to-right order, and let *Ti* be the subtree rooted at *Pi*.

The preorder traversal of *T* is defined, recursively, as follows.

1. Root node of *T* is visited first.

2. For each *i* from 1 to m in increasing order, visit the nodes of *Ti* in preorder.

Post-order traversal *T* is similarly defined.

1. For each *i* from 1 to m in increasing order, visit the nodes of *Ti* in postorder.

2. Root node of *T* is visited finally.

Pre-order traversal is top-down, while *postorder* is bottom-up. However, we can also traverse *T*

is *reverse postorder,* which is top-down, as follows.

1. Visit the root of *T* .

2. For *i* from m to 1 in decreasing order, visit the nodes of *Ti* in reverse postorder.

If a node x is the *i*th node of *T* visited in a preorder traversal of *T* , we say that the preorder rank of x is *i*. If a node x is the *j*th node of *T* visited in a reverse postorder traversal of *T* , we say that the reverse postorder rank of x is *j*. Write *pre_index (x)* and *post_index (x)* for the preorder rank and reverse postorder rank of *x*, respectively, We define the guide pair of x to be the ordered pair *guide(x) = (pre index (x), post index (x))*.

If *(i, j)* and *(k, l)* are guide pairs, we write *(i, j)* ▪ *(k, l)* if *i* ▪ *k* and *j* ▪ *l.* Thus, the set of guide pairs is partially ordered. After the Guide pair has been calculated we need to define *lev_guide* for every node.

x.lev_guide = (x.level, x.preindex, x.postindex).

After calculating the *lev_guide* we need to pipeline the values up to the Root such that the smaller values overtake the larger values. The values will reach the Root one at a time. In parallel we need to pipeline the *lev_guide* triple to r such that the smaller values will overtake the larger values and reach the Root. We use lexical ordering when we pipeline the *lev_guide* triplets. Now each triplet value will reach the root one at a time. Both one of the these *lev_guide* triplet and a value will reach the root at the time Both the value and *lev_guide* triple will be combined to form descending triple consisting of

one value and the guide pair. Now the Triple navigates down the tree until if finds the matching guide pair.

7.2 Variables of LEVEL ORDER SORTING

1. *P.subtreesize,* is the number of processes in the tree *Tp* .

Let *Chldrn(P) = {Q ∈ N(P) : Q.parent = P},* the children of *P* in the tree *T* . These children are then ordered left-to-right making *T* an ordered tree.

2. *P.num_chld,* an integer in the range *0 . . . δp,* whose correct value is the number of children of P in the BFS tree T .

3. *P.child[i] ∈ N(P) U {⊥}* for *1 ≤ i ≤ δp* . If *1≤ i ≤ P.num chld,* then *P.child[i]* is the *i*th *child in P's local ordering of N(P), while P.child[i] = ⊥ if i > P.num chld.*

4. *P.my_order.* If *P ≠ Root,* then the correct value of *P.my_order* is *i* if *P.parent.child[i] = P,* while the correct value of Root.my_order is ⊥.

5. *P.pre_index , P.post_index ,* integers, whose correct values are the preorder and reverse postorder ranks of *P*, respectively. We will write *P.guide = (P.pre_index , P.post_index).*

6. *P.chld_pre_predecessors[i], P.chld_post_predecessors[i],* integer, for all *1 ≤ i ≤ P.num_chld.* The correct value of *P.chld_pre_predecessors* is the number of predecessors of *P.child[i]* in the preorder traversal of *T* . The correct value of *P.chld_post_predecessors* is the number of predecessors of P.child[i] in the reverse postorder traversal of T .

7. *P.lev_guide,* it is triplet of a node P value containing one level value and guide pair.

P.lev_guide = (P.level, P.preindex, P.postindex).

8. *P.desc_pkg,* it is descending triplet formed by combing the value and the *lev_guide* triple.

For example, if the value *x.value* reaches *r* at the same time as the level guide triple *(y.level, y.preindex, y.postindex),* the descending triple *(x.value, y.preindex, y.postindex)* is created and then descends until it reaches y. When it reaches y, the assignment *y.sortvalue* ← *x.value* will

be executed.

9. *P.lev_done,* Boolean, which indicates *Tp* contains no active up-package.

10. *P.status,* an integer in the range [0 . . . 4]. Status variables are used to control the order of computation, to correct errors.

7.3 Functions of LEVEL ORDER RANKING.

1. *Clean_State(P),* Boolean, which is true if *P* is in an initial, or "clean," state. Formally, *Clean_State(P)* is true if all the following conditions hold:

(a) *P.lev_lev* = ∎.

(b) *P.desc_pkg* = ∎

(c) ¬*P.started.*

(d) ¬*P.lev_done*.

2. *Status_Error(P),* Boolean, which is true if P detects that its status is inconsistent with those of its neighbors. Status errors are always the result of arbitrary initialization; eventually, *Status_Error(P)* will remain false forever for all *P*. *Status_Error(P)* holds if any one of the following conditions holds.

(a) *P.status* ∈ *{1, 3}* and *P.parent.status* ∉ *P.status*.

(b) *P.status* ∈ *{2, 4}* and *Q.status* ∉ *P.status* for some *Q* ∈ *Chldrn (P)*.

(c) *P.status* ≠ *0* and *P.parent.status = 0.*

(d) *P.status* ∉ *{0, 1}* and *Q.status = 0* for some *Q* ∈ *Chldrn(P).*

3. *Guide_Error(P),* Boolean, meaning that P can detect an error in the guide string of one of its packages. We say that a guide pair g is consistent with a process *P* if either *g = P.guide,* or there is some *Q* ∈ *Chldrn(P)* such that *g* ≤ *Q.guide.*

Formally, *Guide_Error(P)* is true if either of the following conditions holds:

(a) *P.lev_guide* ≠ ▮▮ and *P.lev_guide.guide* is not consistent with *P*.

(b) *P.desc_pkg* ≠ ▮▮ and *P.desc_pkg.guide* is not consistent with *P*.

4. *Error(P),* Boolean. Formally, *Error(P)* is true if any one of the following conditions holds:

(a) *Status_Error(P).*

(b) ¬*Clean_State(P)* ^ *(P.status = 1).*

(c) *Guide_Error(P)* ^ *(P.status = 2).*

(d) *P.lev_done* ^ ¬*P.started* ^ *(P.status = 2).*

(e) *P.lev_done* ^ *(P.status = 2),* and there is some *Q* ∈ *Chldrn(P)* such that ¬*Q.lev done*.

5. *Start_Pkg(P) = (P.value, P.guide),* of package type, the level-guide triplet whose home process is *P,* and which *P* starts the package.

6. *lev_Redundant(P),* is a Boolean, indicating that *P.lev_guide* is redundant.

If *P* ≠ *Root,* then *lev_Redundant(P)* holds provided the following three conditions hold:

(a) *P.lev_guide* ≠ ▮▮,

(b) *P.parent.lev_guide* ≠ ▮▮,

(c) *P.parent.lev_guide* ≥ *P.lev_ guide,* which means that *P.parent* has already copied *P.lev_guide.*

l*ev_Redundant(Root)* holds provided the following three conditions hold:

(a) *Root.lev_redundant* ≠ ▮▮,

(b) *Root.desc_pkg* ≠ ▮▮,

(c) *Root.desc_pkg.guide = Root.lev_guide.guide.* This means that Root has already used its level guide triplet to initialize a down-package.

We note that *P* can evaluate *lev_Redundant(Q)* for any *Q* ▪ *Chldrn(P).*

7. *Desc_Ready(P),* Boolean, meaning that *P.desc_pkg* is not defined or redundant, and thus *P* can create or copy a new down-package.

Formally, if *P.desc_pkg* = ▮▮, then *Desc_Ready(P)* holds.

If *P.desc_pkg* ≠ ▮▮ and *P.desc_pkg.guide* ≠ ▮▮ *P.guide,* then *Desc_Ready(P)* holds if there is some *Q* ▪ *Chldrn(P)* such that *Q.desc_pkg = P.desc_pkg.*

If *P.desc_pkg* ≠ ▮▮ and *P.desc_pkg.guide = P.guide,* then *Desc_Ready(P)* holds if *P.sortedvalue = P.down_pkg.value*, indicating that *P* has already copied the correct value of *P.rank.*

8. *Lev_Done(P),* Boolean, meaning that *TP* contains no active *lev_guide* packages. The value of *Lev_Done(P)* is the correct value of *P.lev_done.*

Formally, *Lev_Done(P)* holds if the following three conditions hold:

(a) *P.started = 1*, meaning that P has already created an up-package.

(b) *Lev_Redundant(P),* meaning that *P.lev_guide* is redundant.

(c) *Q.lev_done* for all *Q* ▪ *Chldrn(P),* meaning that there are no active packages in any subtree of *P*.

9. *Can_Start(P),* Boolean, meaning that *P* can set *P.lev_guide* to *Start_ Pkg(P).*

Formally, *Can_Start(P)* is true if all the following conditions hold:

(a) ¬*P.started,* i.e., P has not get generated an up-package.

(b) *P.lev_guide* ≠ ∎ or *Lev_Redundant(P).* This means that *P* is not holding an active level guide triplet.

(c) For all *Q* ∈ *Chldrn(P),* either ¬*Lev_Redundant(Q)* or *Q.lev_done.* This means that *P* can determine the smallest active up-package in *TQ*.

(d) For all *Q* ∈ *Chldrn(P),* either *Q.lev_guide* > *Start_Pkg(P)* or *Q.Lev_done.* This means that any active up-package in *TQ* is greater than *Start_ Pkg(P).*

10. *Can_Copy_Up(P,Q),* Boolean, for *Q* ∈ *Chldrn(P). Can_Copy_Up(P,Q)* is true if *P* can send *Q.lev_guide* to *P.lev_guide.*

Formally, *Can_Copy_Up(P,Q)* holds if all the following conditions hold:

(a) *Q.lev_guide* ≠ ∎ and ¬*Lev_ Redundant(Q).* This means that *Q* is holding an active level-guide triplet.

(b) *P.lev_guide* ≠ ∎ or *Lev_Redundant(P).* This means that *P* is not holding an active level-guide triplet.

(c) For all *R* ∈ *Chldrn(P),* either ¬*Lev_Redundant(R)* or *R.lev_done.* This means that *P* can determine the smallest active level-guide triplet in TR.

(d) For all *R* ∈ *Chldrn(P),* either *R.lev_guide* ≥ *Q. lev_guide* or *R.lev_done.* This means that any active up-package in TR is greater than *Q. lev_guide*.

(e) *P.started,* or *Start_Pkg(P)* > *Q. lev_guide.* This means that *Start_Pkg(P)* has already gone up, or must wait for *Q. lev_guide* to go up first.

7.4 Messages of LEVEL ORDER SORTING

- send *subtreesize*. Each process P sends subtreesize to its parent where

$$Subtreesize(P) = 1 + \sum_{Q \in Chldrn(P)} Q.subtreesize$$

- send *P. chld_pre_predecessor[i]*. Each process *P* sends the value *chld_pre_predecessor[i]* to its children where is the index of children. It is used to calculate the guide pair value.
- send *P.chld_post_predecessor[i]*. Each process *P* sends the value *chld_post_predecessor[i]* to its children where *i* is the index of children. It is used to calculate the guide pair value.
- send *P.level.* Each process *P* starting for root sends the level values to its children.
- send *P.(lev_guide).* Each process *P* sends the triplet value *lev_guide* to its parent.
- send *P.(up_value)* Each process *P* sends the value to its parent until it reaches the root.
- send *P.(desc_pkg)* Starting from the root every process sends a triplet value containing the value and guide pair to its children until the guide pair matches the value of the nodes guide pair.
- send(status=0). Initially when the network has arbitrary initialization . If a process *P* find the status of a process incorrect. Then P starts sending the message send(status=0) to all its neighbours.

- send(status=1). A new epoch is initiated by the Root by changing its status to 1 and it sends a message send(status=1) to all its children. This message will end after reaching the leaf nodes.
- send(status=2). This message is initiated by the leaf nodes and it is send to its parent. It ends after reaching the Root node. In this epoch all the rank order processing is done.
- send(status=3). This message is initiated by the Root nodes when all the rank processing is done by the and this message ends when it reaches the leaf nodes.

7.5 Explanation of LEVEL ORDER SORTING

We now give an intuitive explanation of how the LEVEL ORDER SORTING algorithm works in finding the sorted values which are sorted in level order. The sorting will be done in phases. In the first phase values of the root i.e level value is .
set to zero , *pre_order* and *post_order* are set to 1. Root starts sending message *level +1* to its children and this message will pass until it reaches the leaf nodes. Now each node will have the level value i.e the distance from the root.

After the computation of level for each node it should calculate the guide pair value. LEVEL ORDER SORTING uses GUIDE algorithm for calculating the guide pair values. For calculating the guide pair every node should calculate the *subtreesize*. Starting from the leaf node every node sends a message *subtreesize* to its parent. Now the guide pair for root *pre_order* and *post_order* are set to *1*. *P.chld_pre_predecessors[1]* is set to *P.pre_index.* For all other children *P.chld_pre_predecessors[i]* is set as *P.child_pre_predecessors[i-1] + P.child[i-1].subtreesize.* Now every node will have its

child_pre_predecessors[i] and this node sends a message *child_pre_predecessors[i]* to its children matching its order. After each children receiving the message it calculates *pre_index i.e 1+P.parent.chld_pre_predecessors[P.my_order]*.

In the similar way it calculates the post_order. Now every node has the guide pair calculated.

Now *lev_guide* triplet is defined for each node it is the combination of level and guide pair. This lev_guide is pipelined until it reaches the root. Starting from the leaf node every node sends the message send *lev_guide* to its parent. Guard ensures that the smaller will reaches node first and *lev_guide* triplet are ordered in lexical order. In parallel we pipeline the values present at every until it reaches the root. Now root combines *lev_guide* triplet and value to form a *desc_pkg* triplet. . *P.desc_pkg,* it is descending triplet formed by combing the value and the *lev_guide* triple. For example, if the value *x.value* reaches *r* at the same time as the level guide triple *(y.level, y.preindex, y.postindex),* the descending triple *(x.value, y.preindex, y.postindex)* is created and then descends until it reaches y. When it reaches y, the assignment *y.sortvalue* ← *x.value* will be executed. This traversed down the tree until it reaches its home process.

7.6 LEVEL ORDER SORTING Algorithm Steps

A1 *P=Leaf* *subtreesize = 1*

send *P.subtreesize* to *P.Parent*

A2 *Q Cp* receive *Q.subtreesize*

P.received*Q* ← True

A3 *Q Cp* *Subtreesize(P)=1+ Q Cp Q.subtreesize*

if *P*.received*Q* = True

A4 *P=Root* *P.pre_index* ← *1*

P.post_index ← *1*

P.level ← *0*

A5 *Q Cp* send *P.level + 1*

Q receive *P.level +1*

A6 *Q Cp* *P.chld_pre_predecessors[1]* ← *P.pre_index*

Q.my_order is 1 send *P.chld_pre_predecessors[1]*

P.chld_post_predecessors[P.num_chld] ← *P.post_index*

send P.chld_post_predecessor[P.num_child]

A7	$\forall Q \in Cp$	*Q.my_order is* $2 \le i \le$ *P.num_child*	*P.chld_pre_predecessor[i]* ← *P.child_pre_predecessors[i-1] + P.child[i-1].subtreesize* send *P. chld_pre_predecessor[i]*
A8	$\forall Q \in Cp$		Receive *P.parent.chld_pre_predecessors[P.my_order]* Received *Q* = True
A9	received *Q* ← True		*P.pre_index* ← *1+P.parent.chld_pre_predecessors[P.my_order]*
A10	$\forall 1 \le i < P.num\_child$		*P.chld_post_predecessor[i]* ← *P.child_post_predecessors[i+1] + P.child[i+1].subtreesize* send *P.chld_post_predecessor[i]*
A11	$\forall Q \in Cp$		Receive *P.parent.chld_post_predecessors[P.my_order]* Received *Q* = True
A12	If received *Q*←True		*P.post_index*← *1+P.parent.chld_post_predecessors [P.my_order]*

A13	If *Error(P),*	Then *P.Status* ← *0.*
	∀*Q* ∈ *nebrs(P)*	send(status=0)
	∀*Q*	receive(status=0)
A14	If *Root.status = 0 or 4* and	Then *Root.status* ← *1*
	∀*Chldrn(Root).Status = 0* or *4*	
	Root is set to Clean Slate.	
	∀*Q* ∈ *Chldrn(P)*	send(status=1)
	∀*Q*	recieve(status=1)
A15	If *P* ∉ *Root, P.Parent.status = 1* and	Then *P*.receive
	∀*Q* ∈ *Chldrn(P)*	
	P is set to clean slate	
	Q.Status = 0 or 4	
A16	*P = leaf node,* ∀*P* ∈ *Chldrn(Q)*	Then *P*.status ← 2
	If *Q.status = P.status =1*	*P*.send (status = 2)
		Q.recieve(status =2)
A17	If *P.status = 1* and ∀*Q* ∈ *Chldrn(P)*	*Q*.send(status=2)
	Q.status = 2	

	∀*P*	recieve(status=2)
A18	*P.status = 2* and *Can_start(P)*	*P.lev_guide.value* ← *P.value* *P.lev_guide.guide* ← *P.guide* *P.up_value* ← *P.value*
A19	If *P.status = 2 and Can_Copy_Up(P,Q)* Then *Q* ∈ *Chldrn(P)*	*P.lev_guide* ← *Q.lev_guide* send *Q.(lev_guide)* recieve *P.(lev_guide)* send *Q.(up_value)* recieve *P.(up_value)*
A20	If *P.started = 1, lev_Redundant(P) = 1* Then and *Q.lev_done = 1* ∀*Q* ∈ *Chldrn(P)*	*P.lev_done* ← *1*
A21	If *Desc_Ready(Root)* , *Root.up_Package*≠∅	Then *Root.desc_pkg.value*

	▬*Up_Redundant(Root)*	←*Root.value* *Root.desc_pkg.guide* ←*Root.lev_guide.guide*
A22	If *P* ▞ *Root*, *Desc_Ready(P)*, *P.parent.desc_package* ▞ ▙ ▬*Desc_Redundant(P.parent)* *P.parent.desc_pkg.guide* is consistent with *P* ▞*P* ▀ *Chldrn(Q)* ▞*P*	Then *P.desc_pkg* ← *P.parent.desc_pkg* send *Q.(desc_pkg)* recieve *P.(desc_pkg)*
A23	If *P.desc_pkg* ▞ ▙ *P.desc_pkg.guide=P.guide* *P.desc_pkg.value* ▞ *P.rank*	Then *P.sortvalue* ← *P.desc_pkg.value*
A24	If *Root.lev_done*, *Desc_Ready(Root) and* *Root.value = Root.desc_pkg.value*	Then *Root.status* ←*3*
A25	If *P* ▞ *Root*, and *P.parent.status = 3*	Then *P.status* ←*3*

	∀*Q* ∈ *Chldrn(P)*	
	P.status = 2	
	Q.status = 2	
	Desc_Ready(P)	
	Z=*Parent(P)*	send(status=3)
	∀*P*	receive(status=3)
A26	*P = Leaf*	*P.status* ←*4*
	P.Parent.status = 3	
A27	If *P.status = 3*	
	Q.Send(status=4)	
	∀*Q*=*Chldrn(P)*	*P*.Recieve(status=4)
	Q.status = 4	

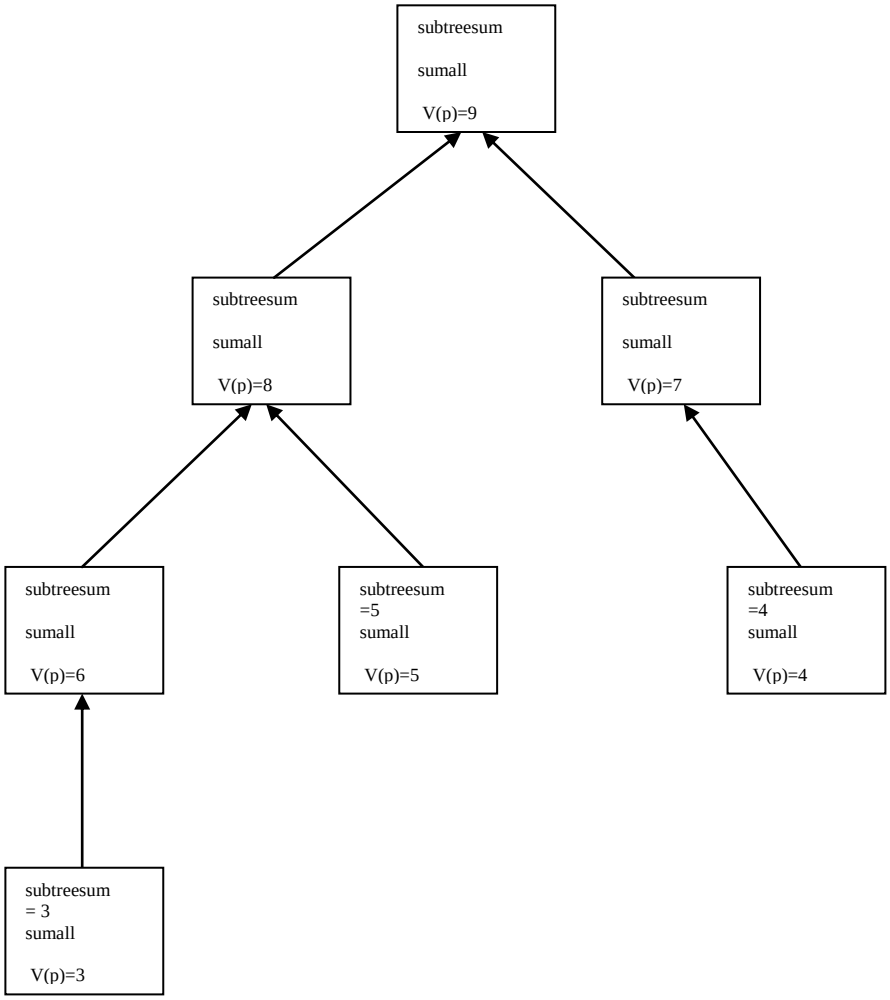
CHAPTER 8

CONCLUSION

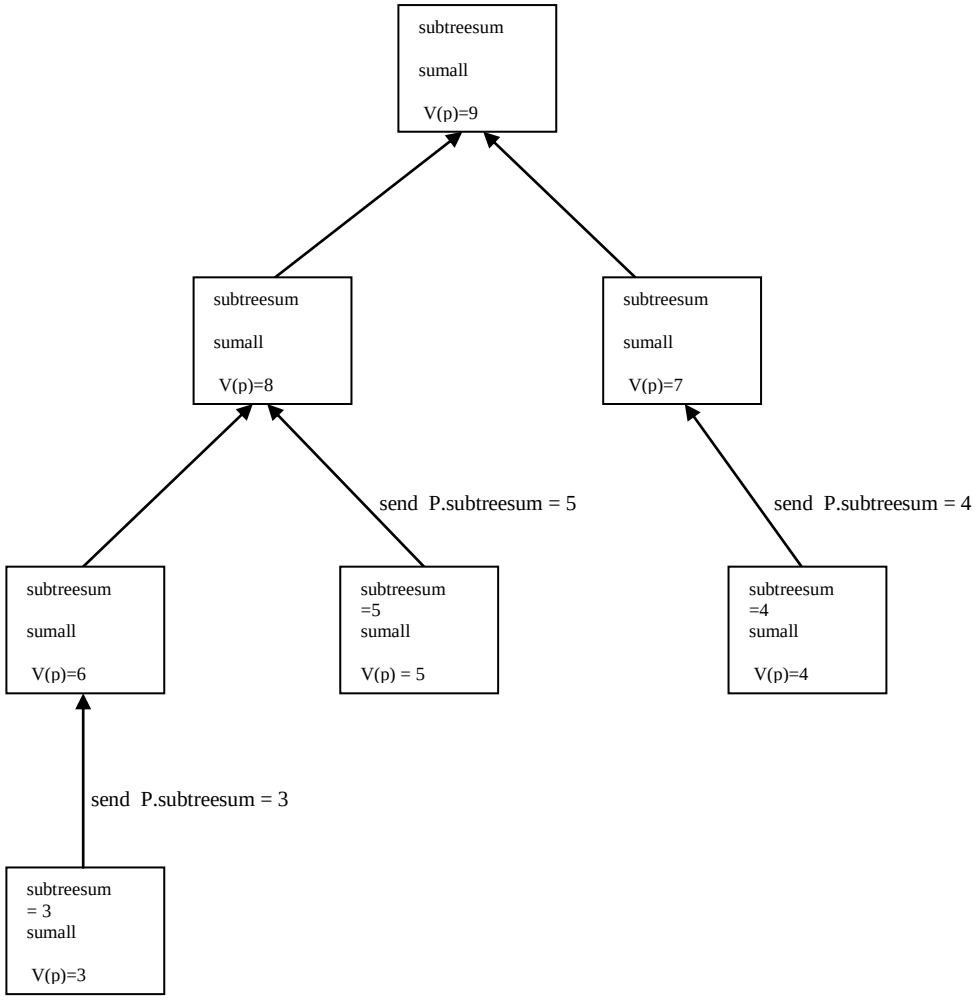
In this thesis we have developed different algorithms in message passing system by considering different performance factors. When we consider the scalability factor as the number of process in the network increases it can be handled easily in the message passing system. The pre-processing overhead and the buffer can be avoided in this system. The SUM algorithm can be used for combing the values present in the network using commutative and associative operations and finding the minimum and maximum values present in the network. The MEAN algorithm which is an extension of SUM algorithm can be used for finding the mean of all the values present in the network. The GUIDE algorithm can be used for finding the guide pair values which can be used for computing the guide pair values. These guide pair values can be used of traversing the tree. The RANK and LEVEL ORDER SORTING algorithms can be used to assigning rank and sorting the nodes present in the network. Different strategies were used for sorting the nodes in the tree.

APPENDIX A

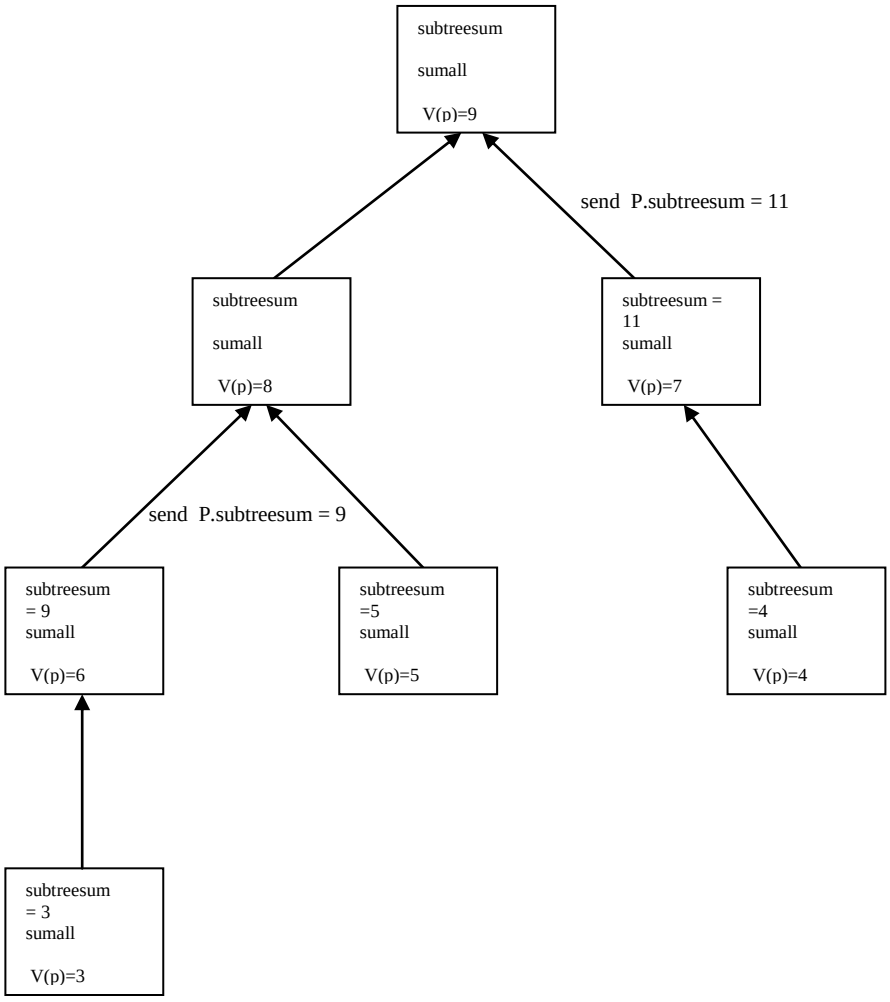
Diagrams of SUM



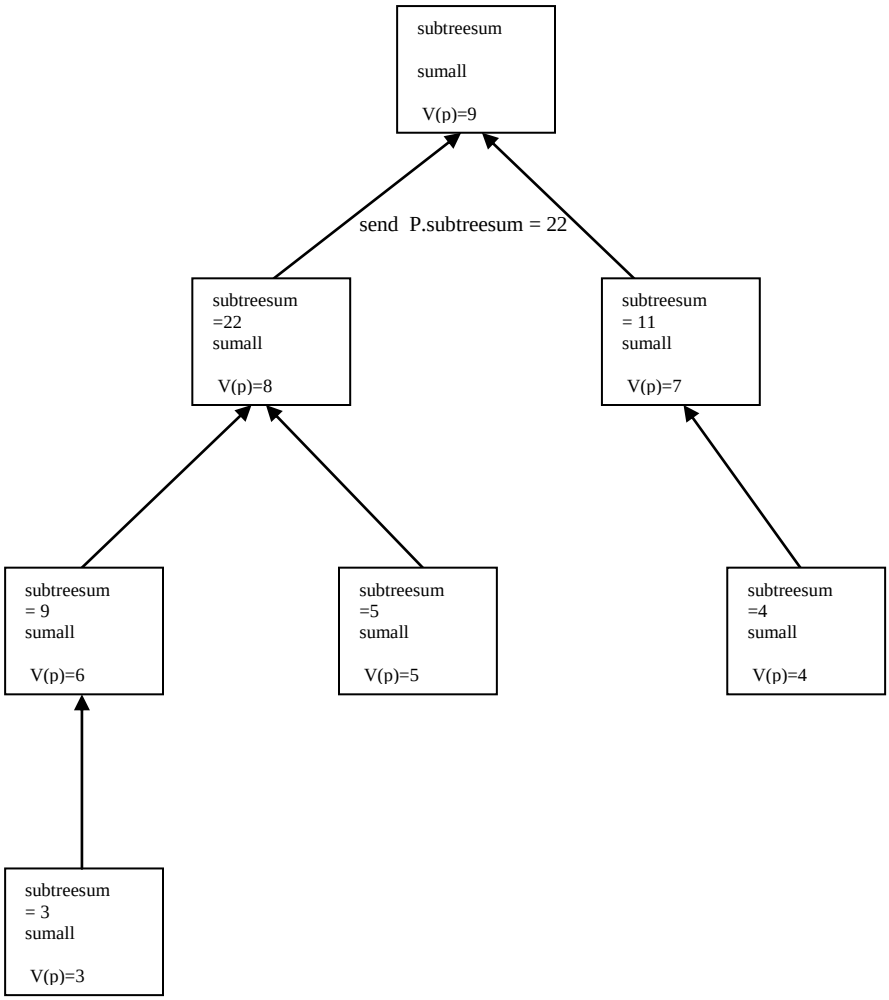
Initial configuration of the network with every node having the values V(P)



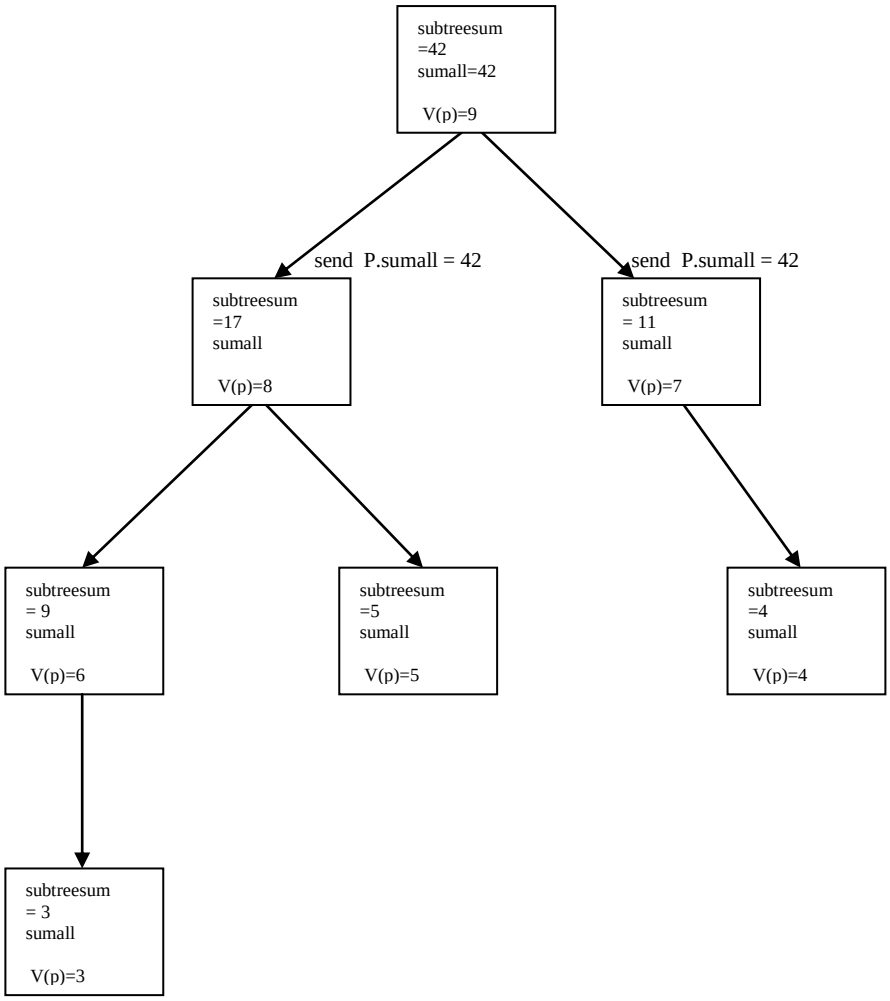
A converge cast wave starting from the leaf nodes starts sending the value V(P) which is Subtreesum(P) to its parent.



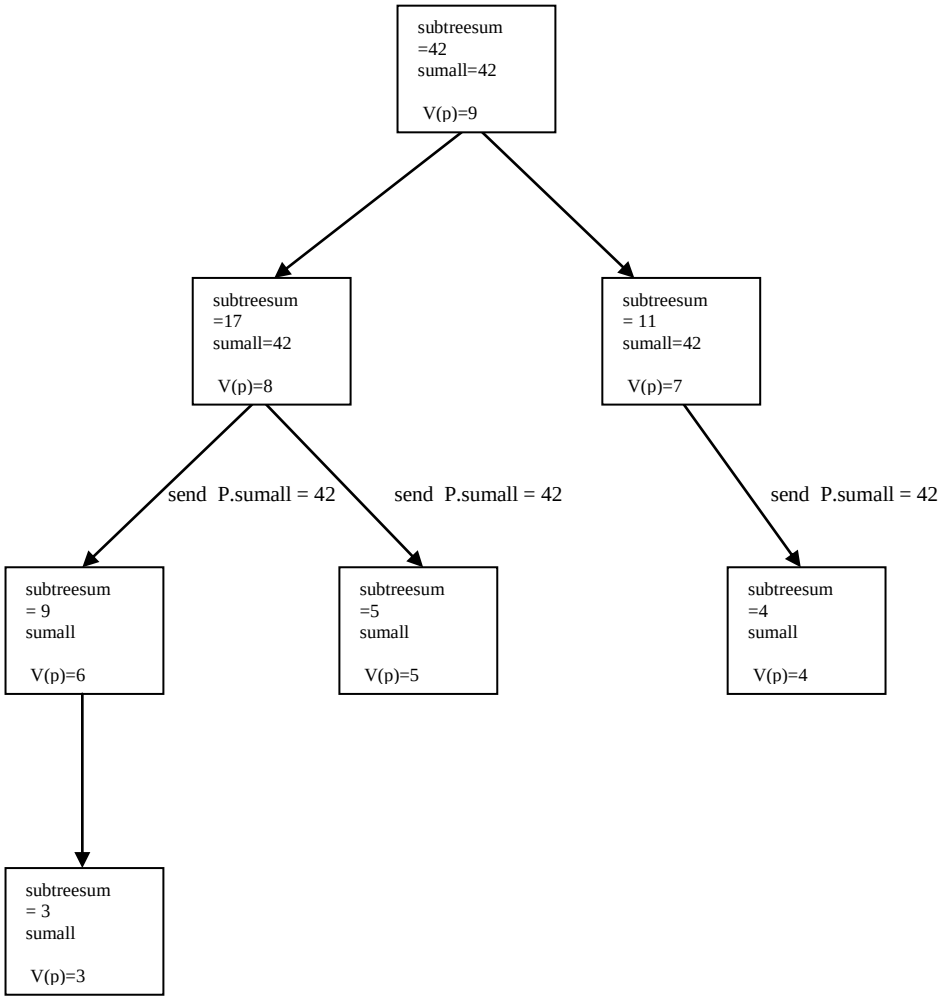
A nodes waits until it receives all the values V(P) from its children and then calculates the sum of all values along with its values and then it sends the value to its parent.



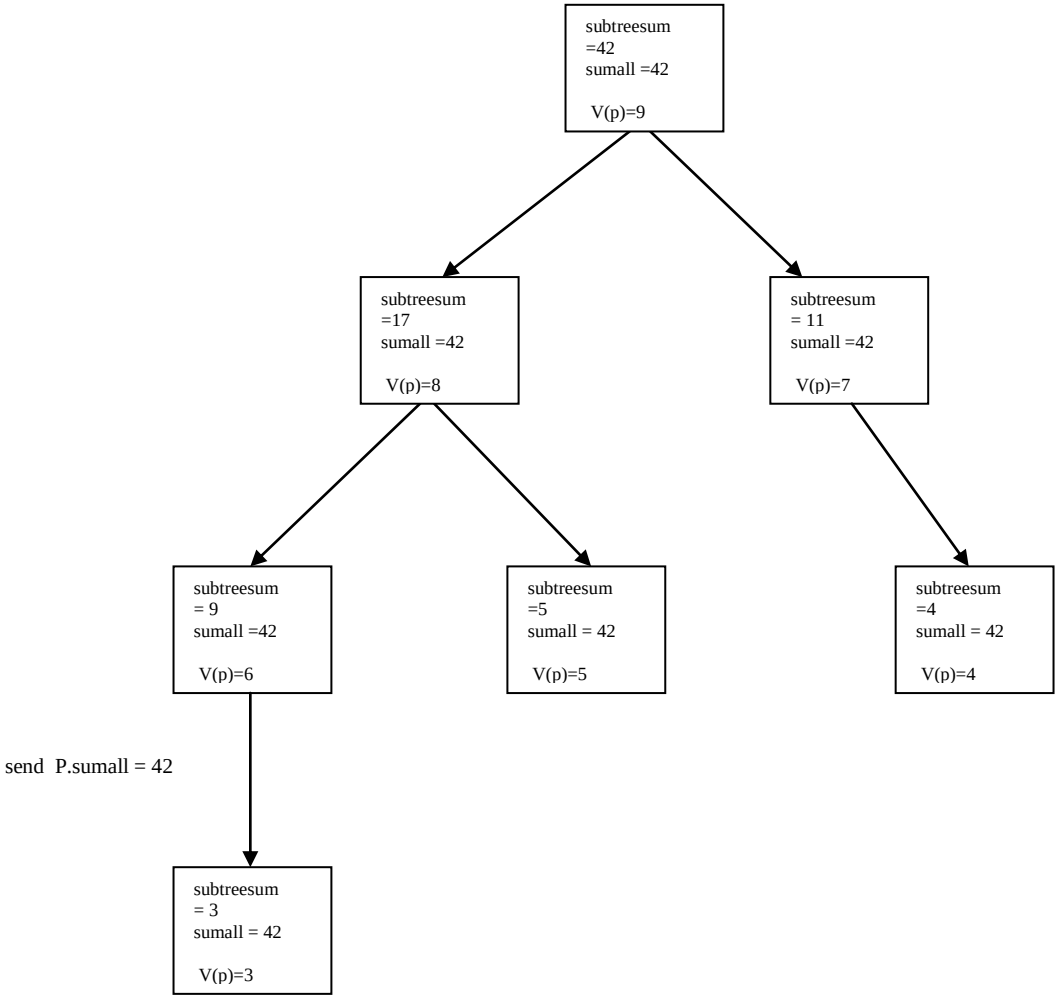
After the node receiving the values from its children it calculates the value Subtreesum(P) and send the value to its parent.



After the Root receiving the values from all its children it calculates the value sumall(P) and send the value to all its children.



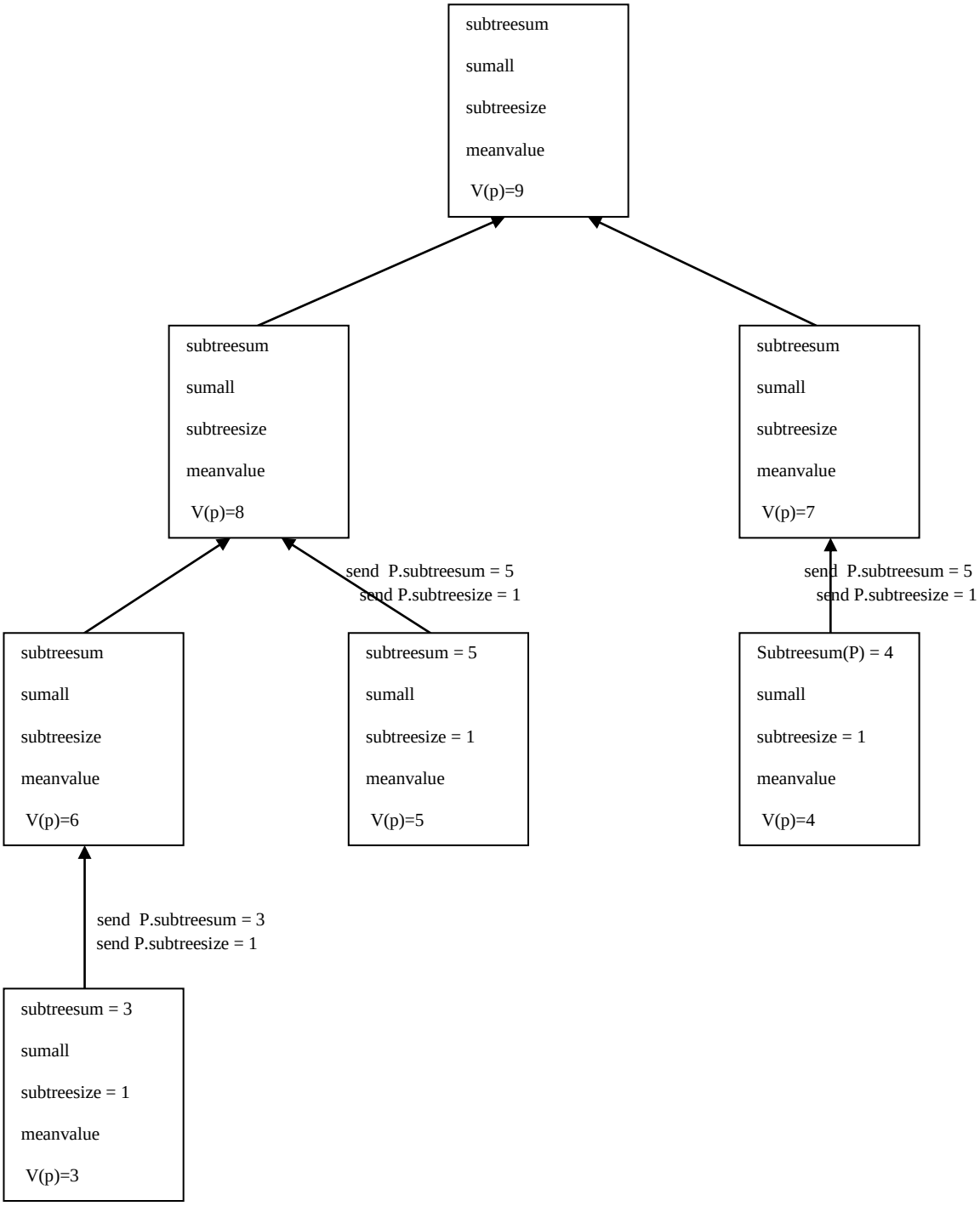
The broadcast wave of sending sumall(P) continues until it reaches the leaf nodes.



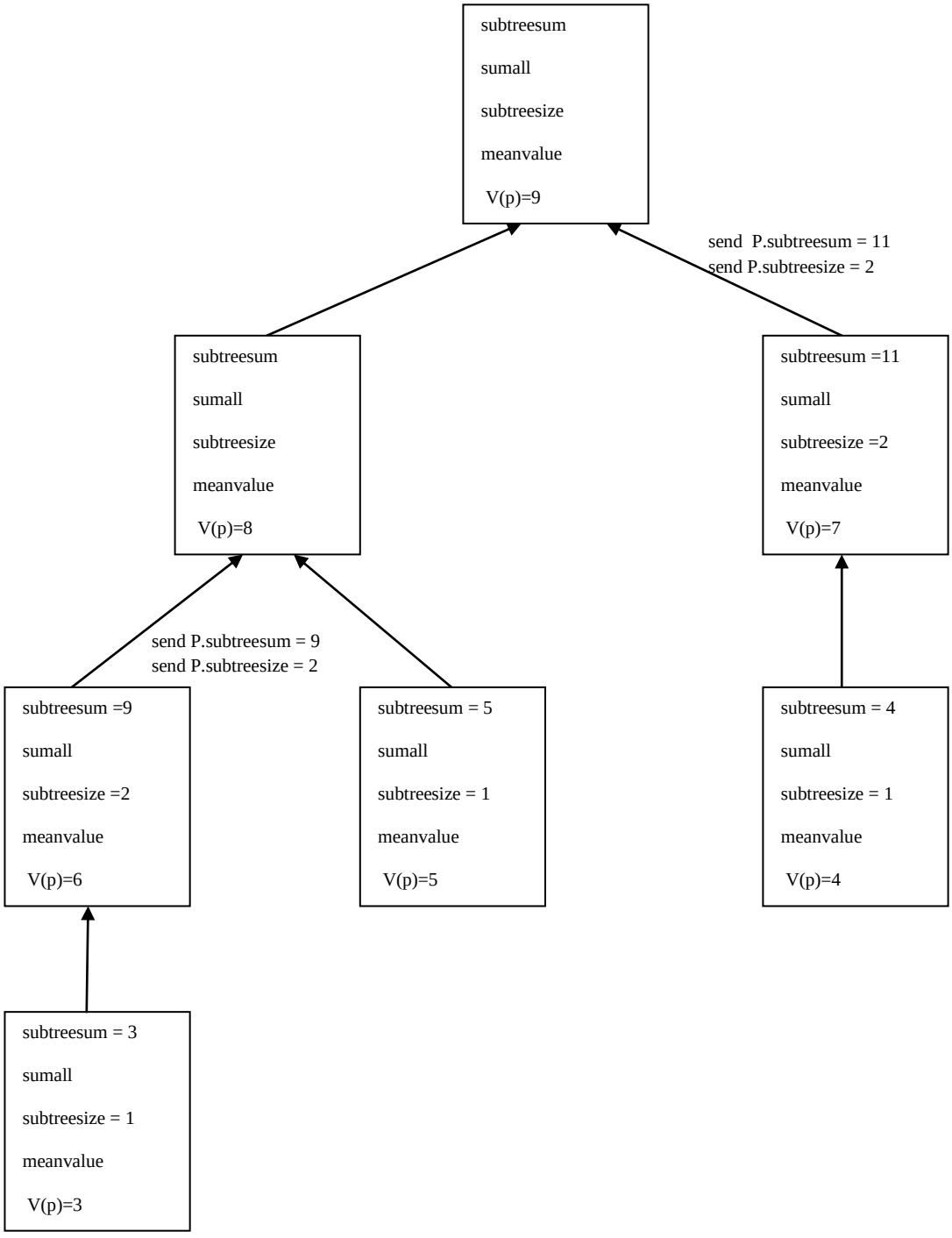
Now all the nodes in the tree has the value sumall(P).

APPENDIX B

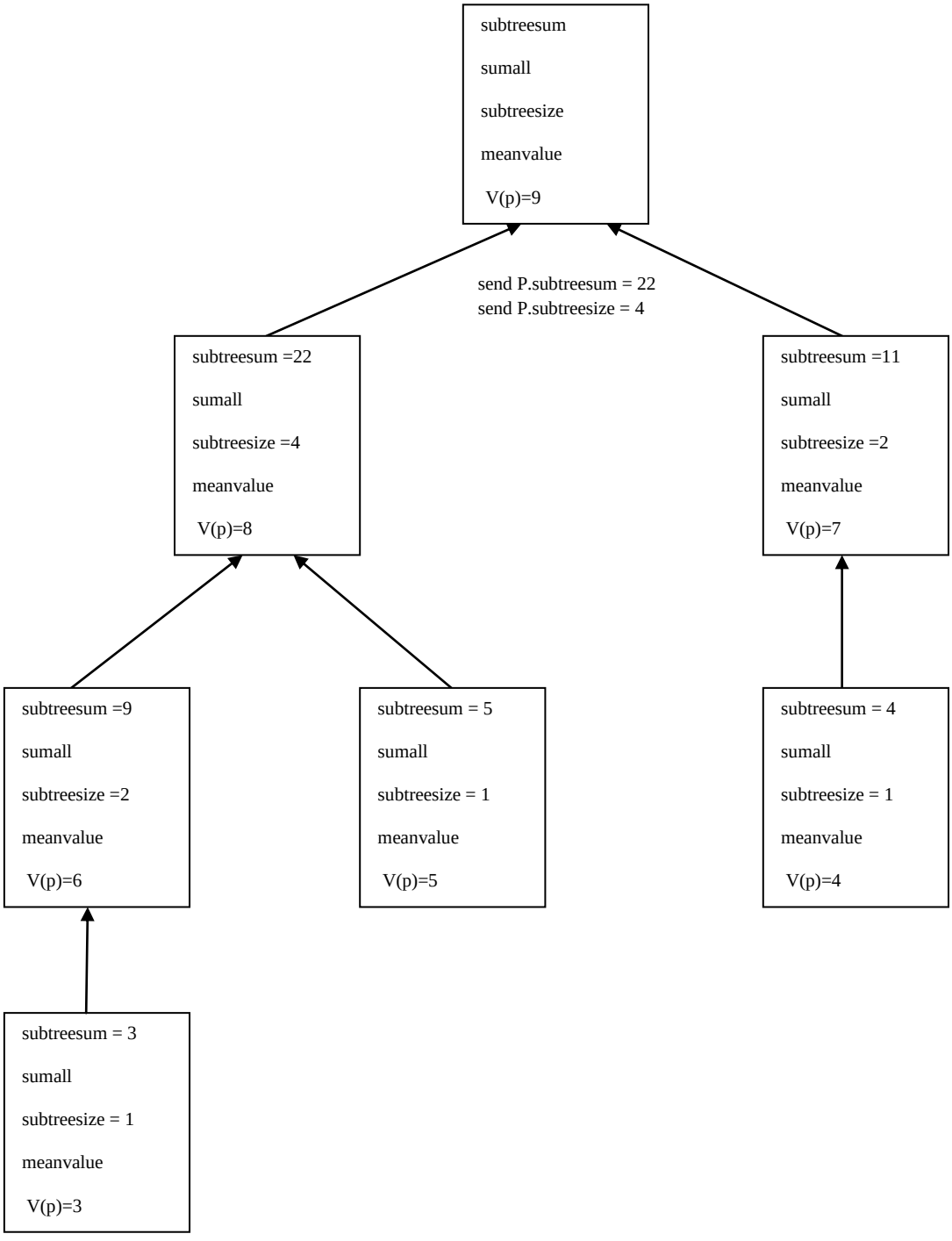
Diagrams of MEAN



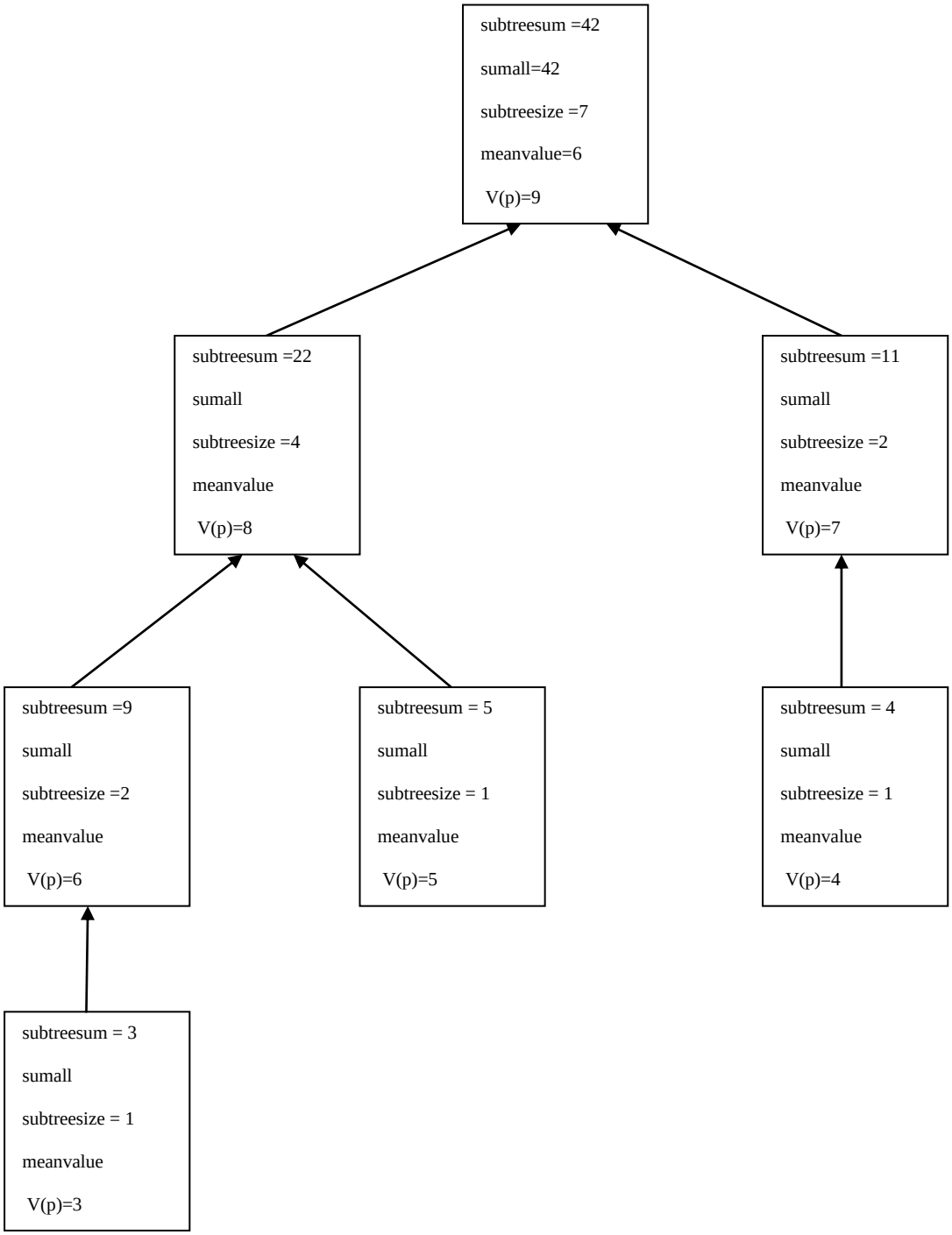
Initial configuration of the network with a convercast wave starting from the leaf nodes and sending the values Subtreesum and Subtreesize to its parent.



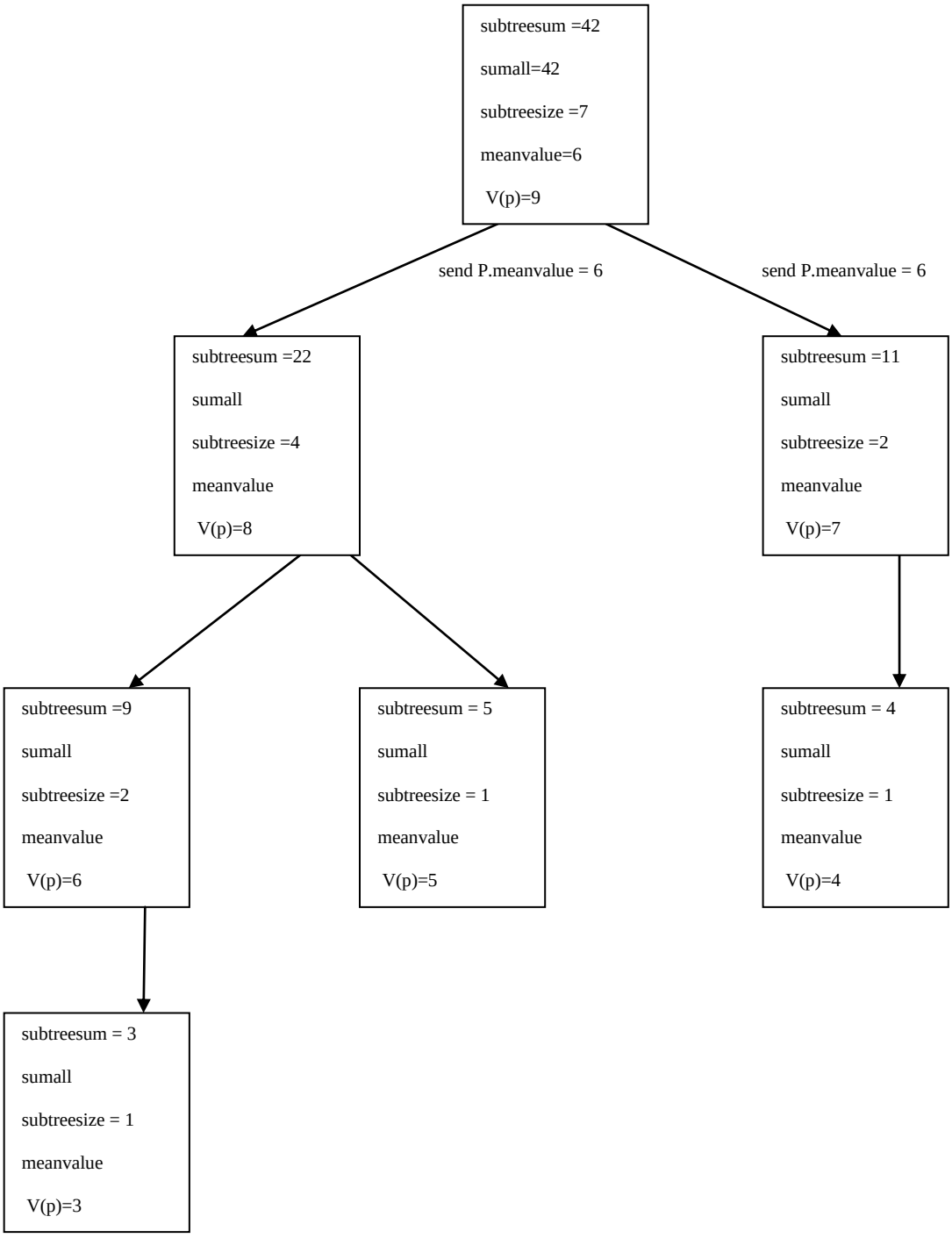
A nodes waits until it receives all the values P.subtreesum and P.subtreesize from its children and then calculates the sum of all values along with its values and then it sends the value to its parent.



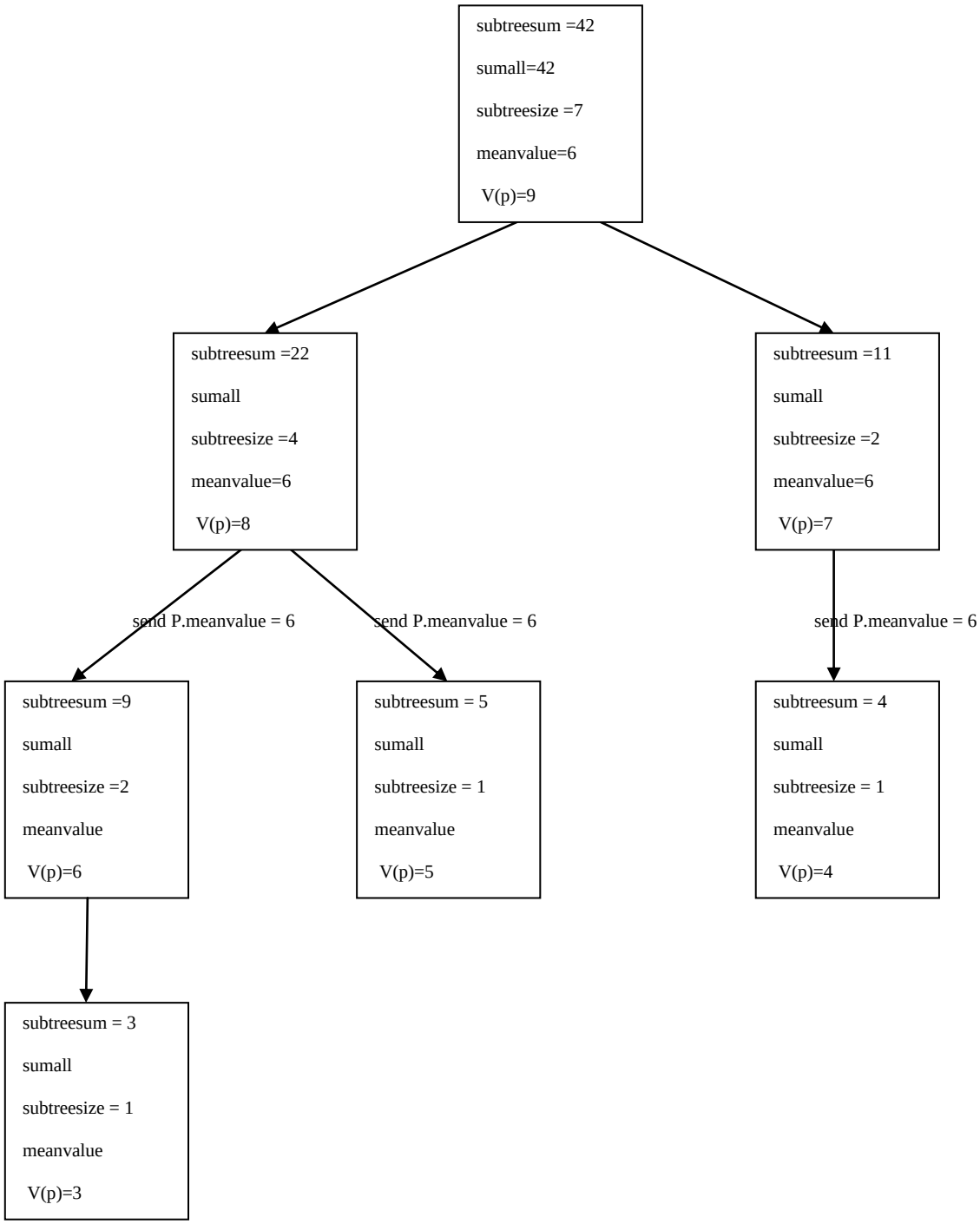
After the node receiving all the values from its children it calculates the value P.subtreesum and P.subtreesize and send the value to its parent.



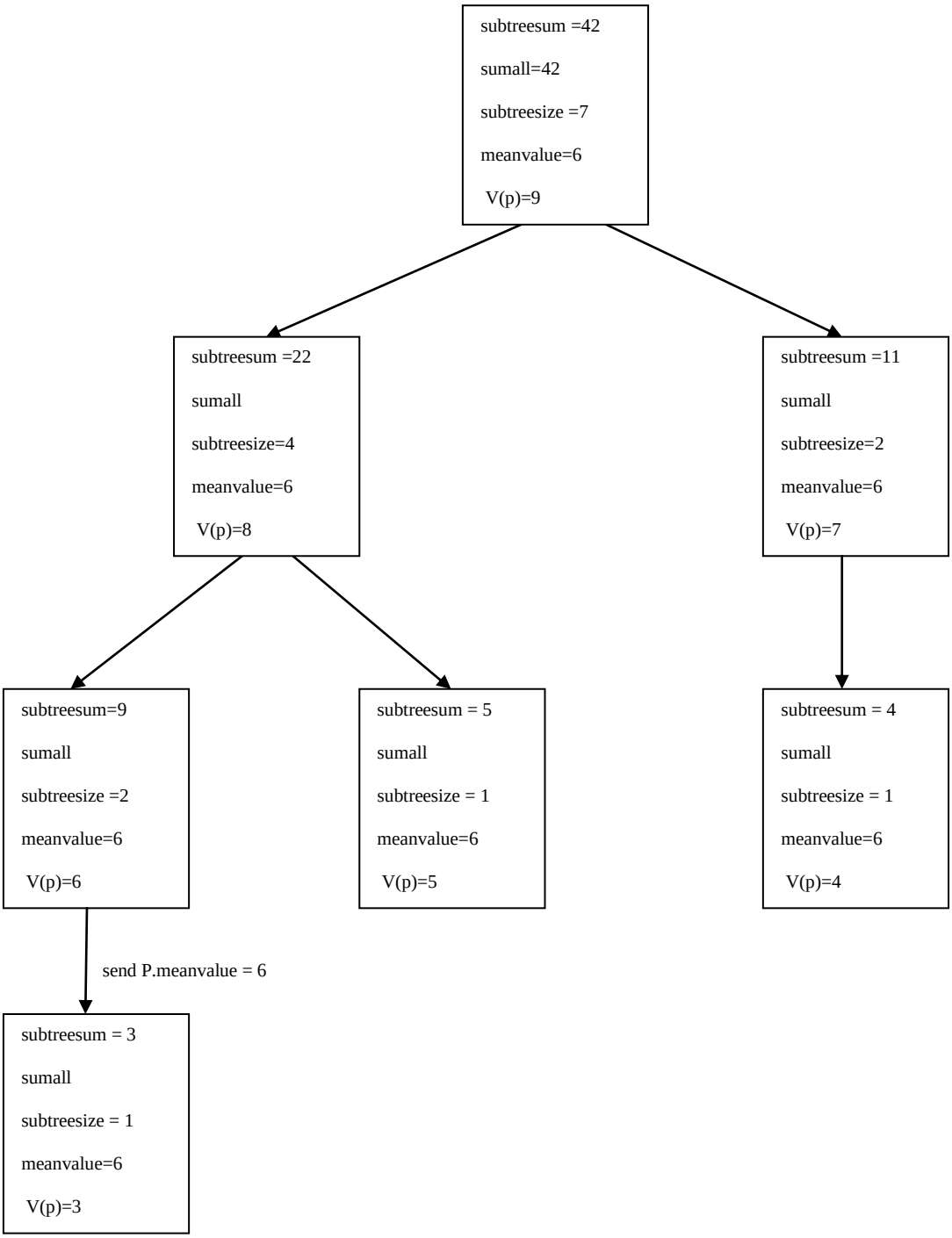
After the Root receiving all the values from its children it calculates the Mean value.



Root sends the mean value to all its children by starting a broadcast wave.



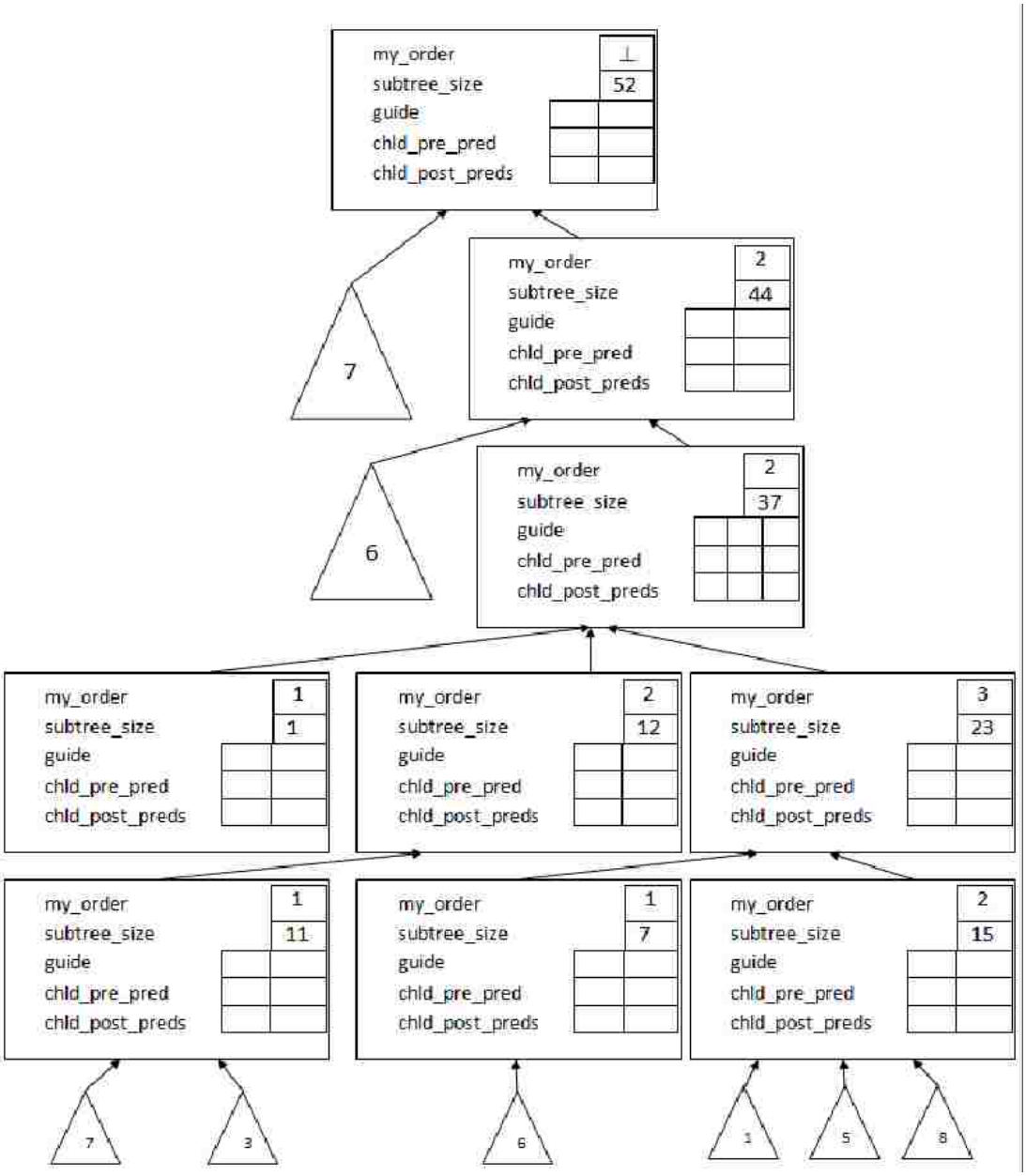
Broadcast wave of sending the mean value to its children continues until it reaches the leaf nodes.



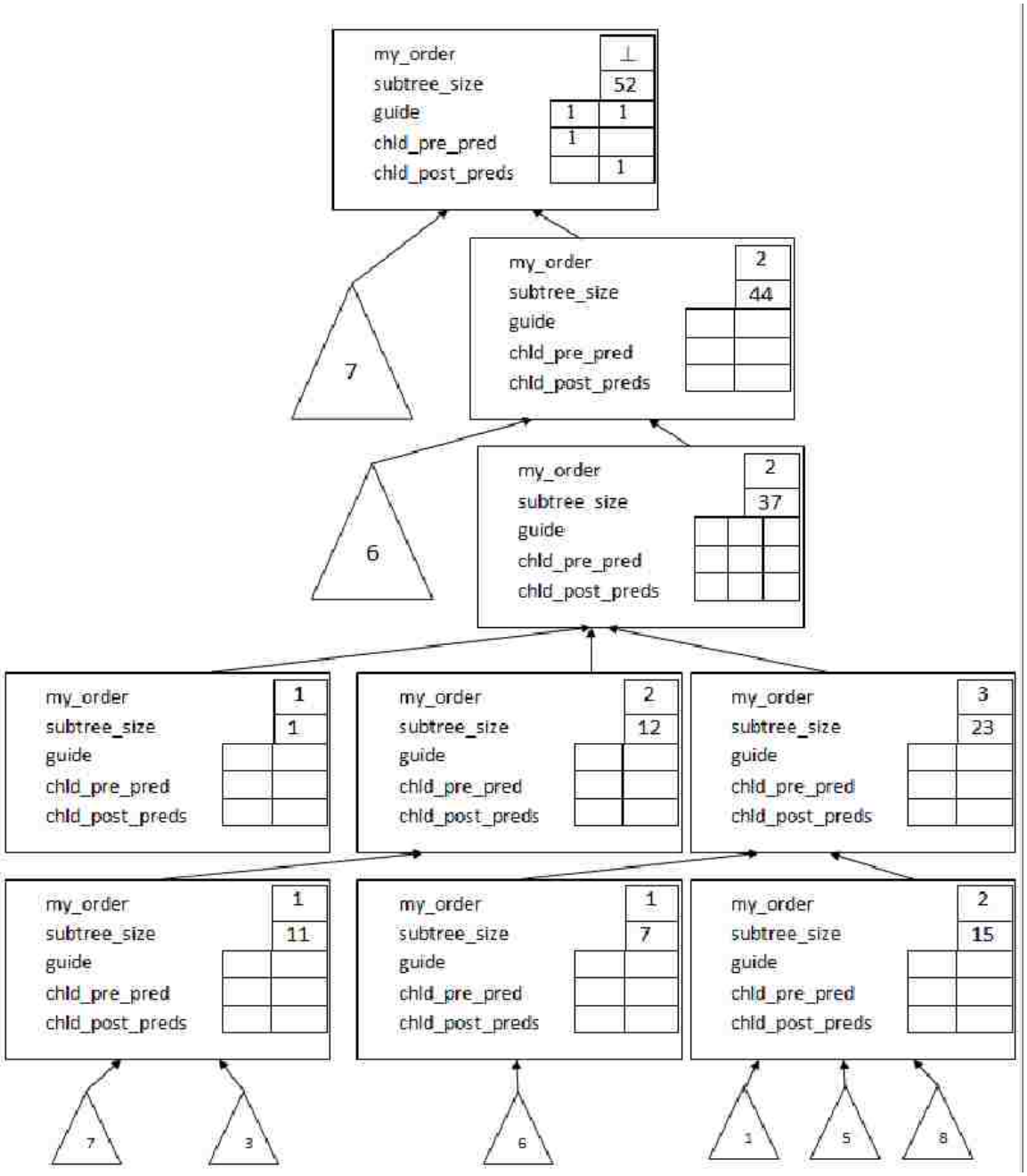
After the broadcast wave reaches the leaf node all the node in the tree has the mean value.

APPENDIX C

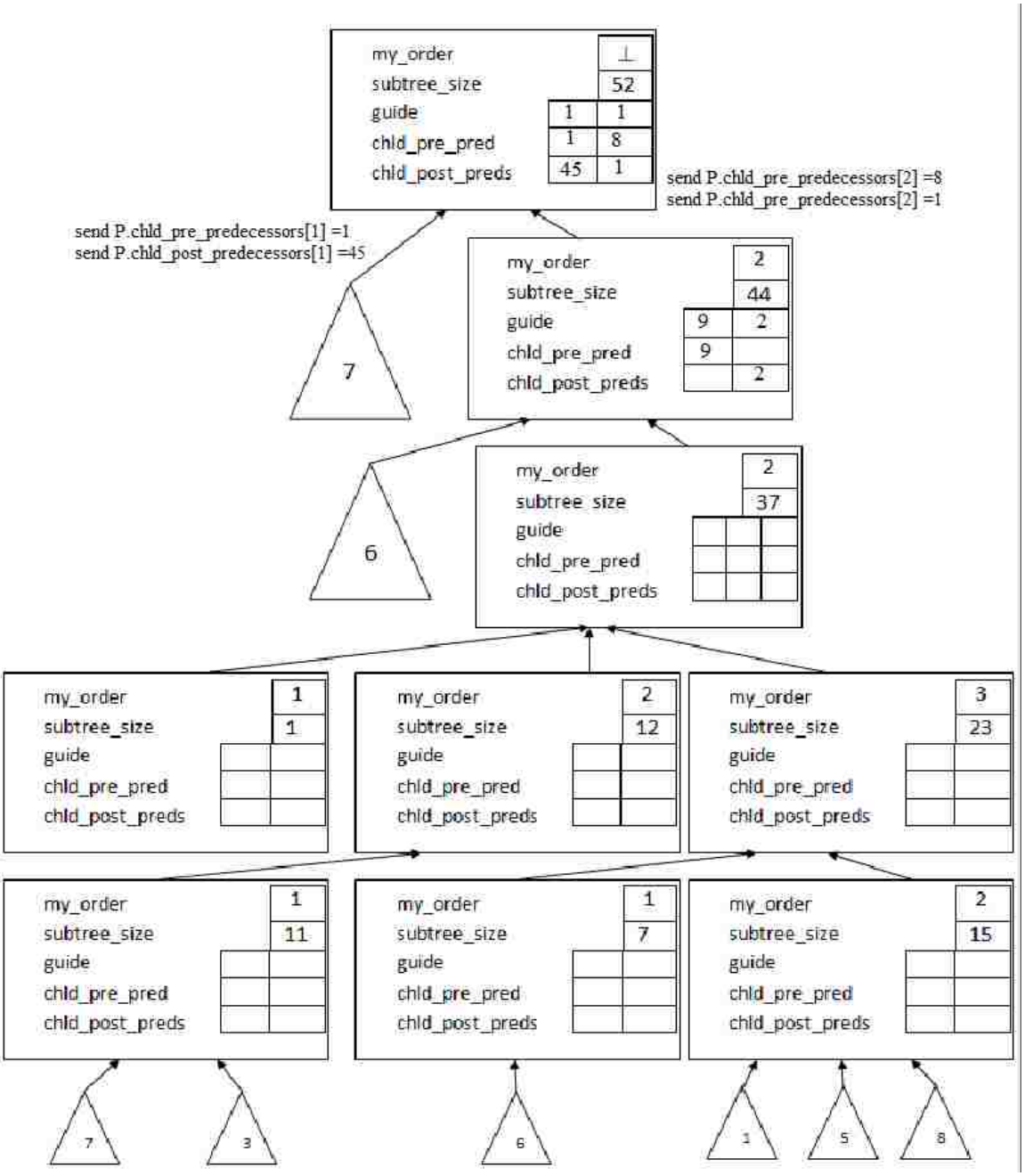
Diagrams of GUIDE Algorithm



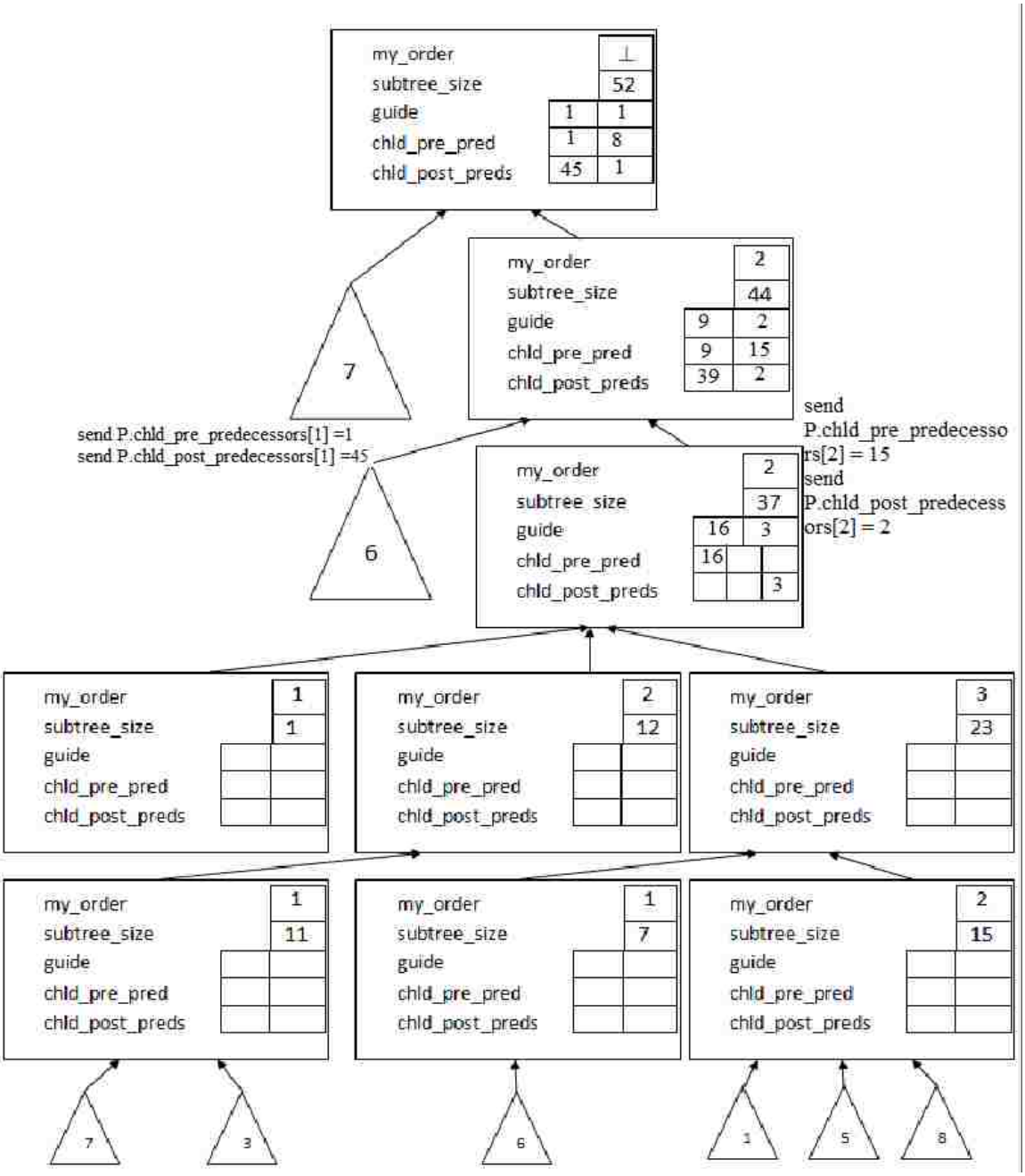
This is the initial configuration of the network



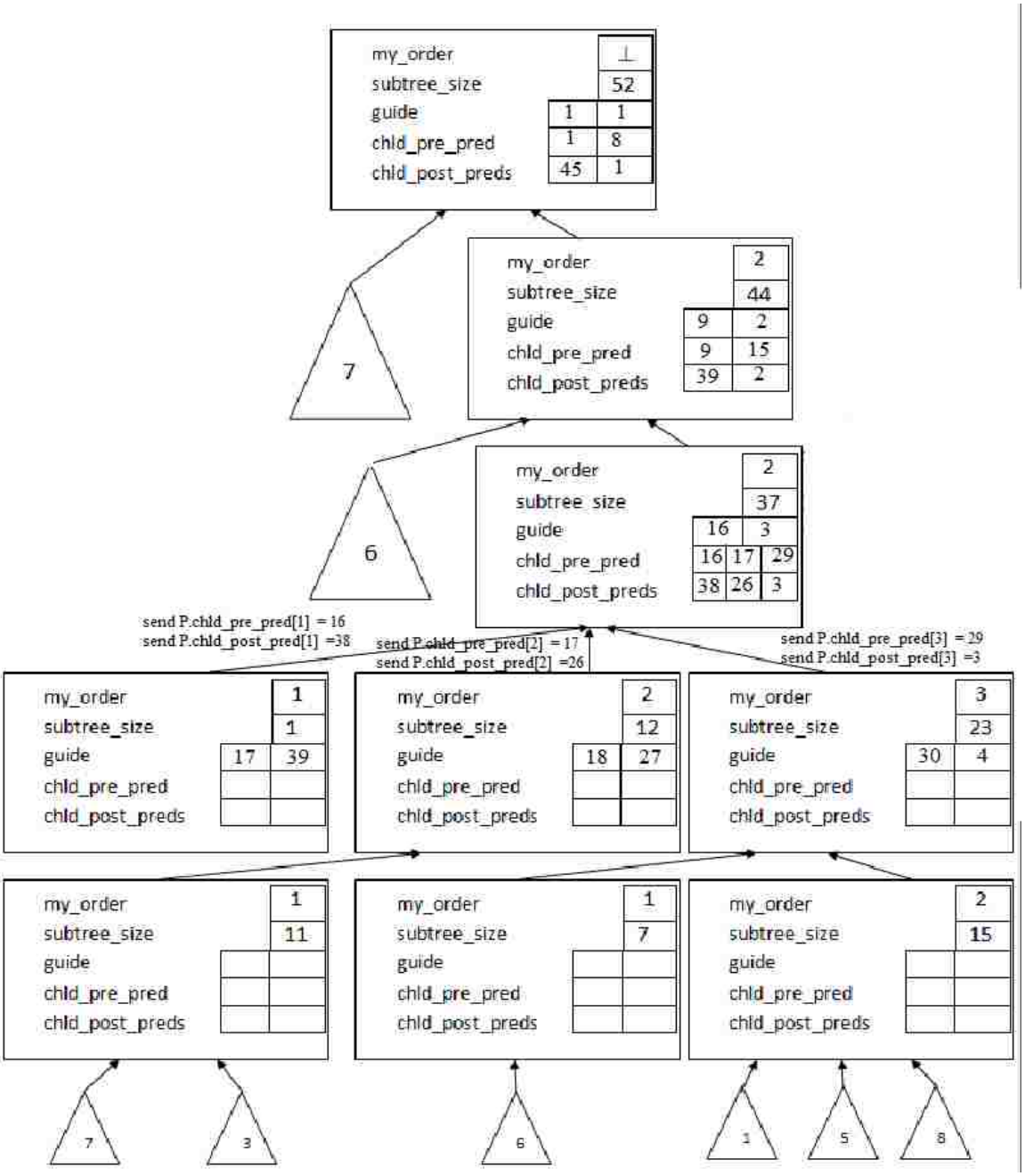
Root has assigned the guide pair value (1,1)



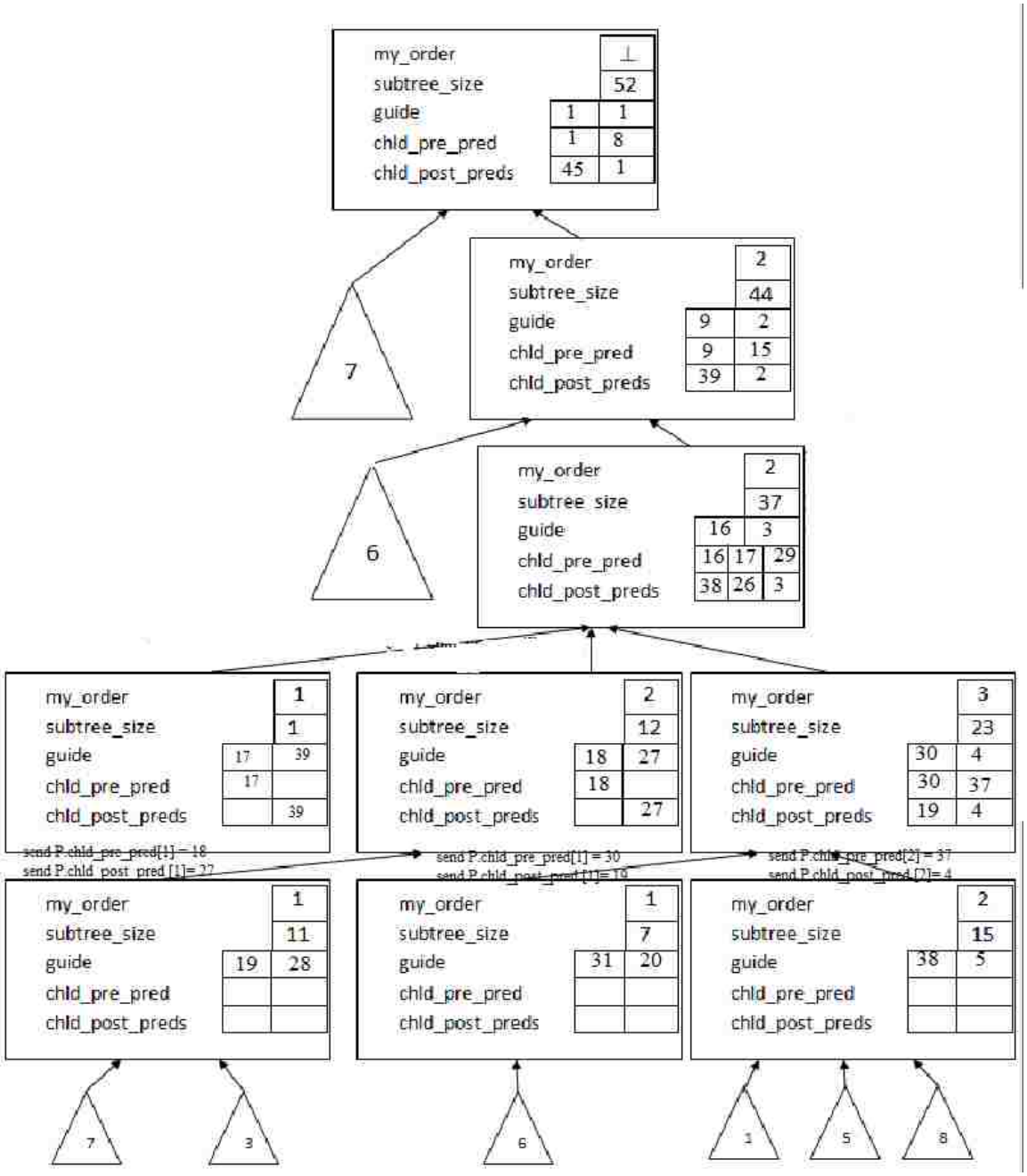
Root calculated the pre and post predecessors of its children and forwarded the to their children



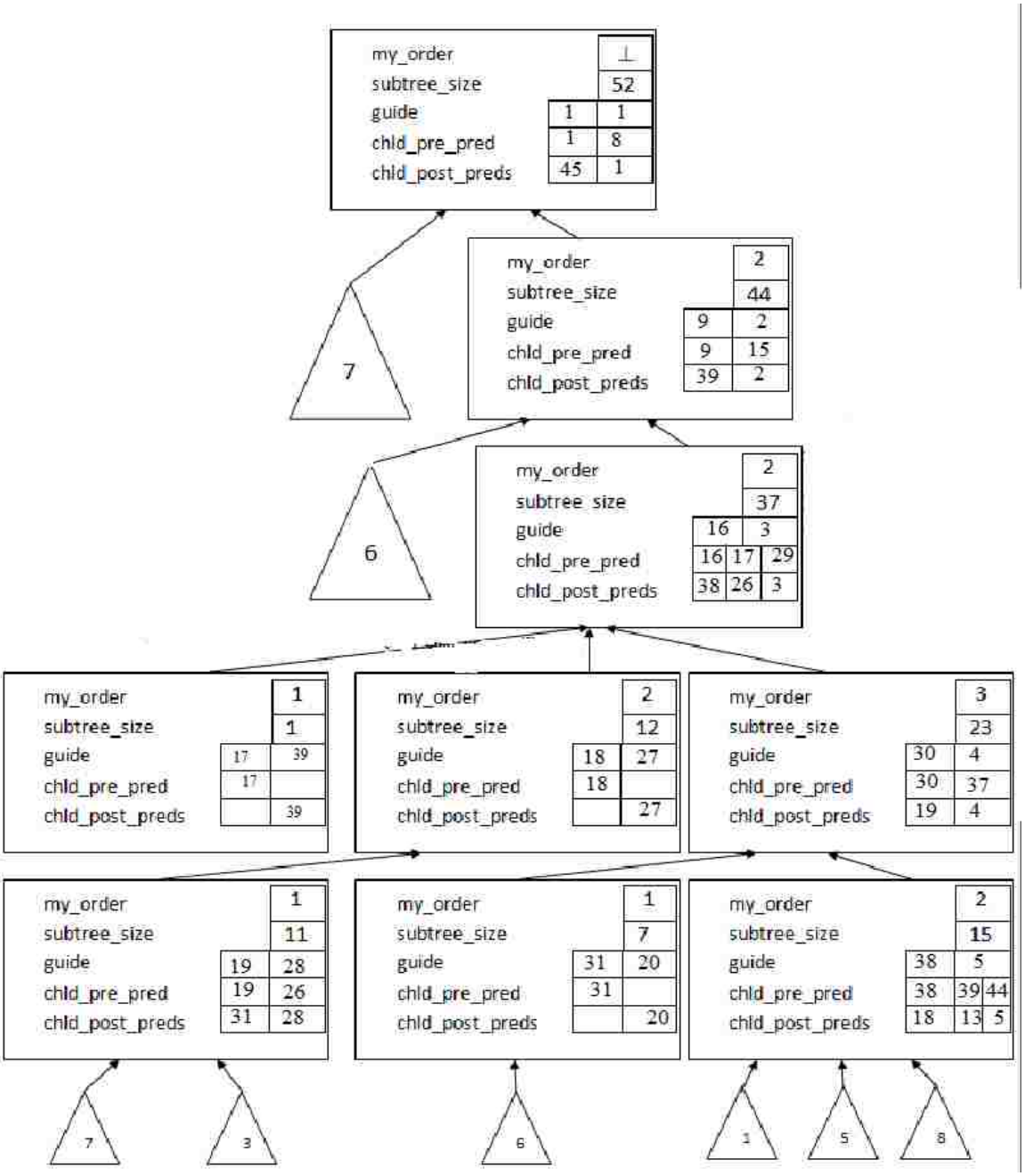
Roots children calculates the guide pair values and send the values to its children



Node (16,3) computes the guide pair values and sends the pre and post predecessor values to its children



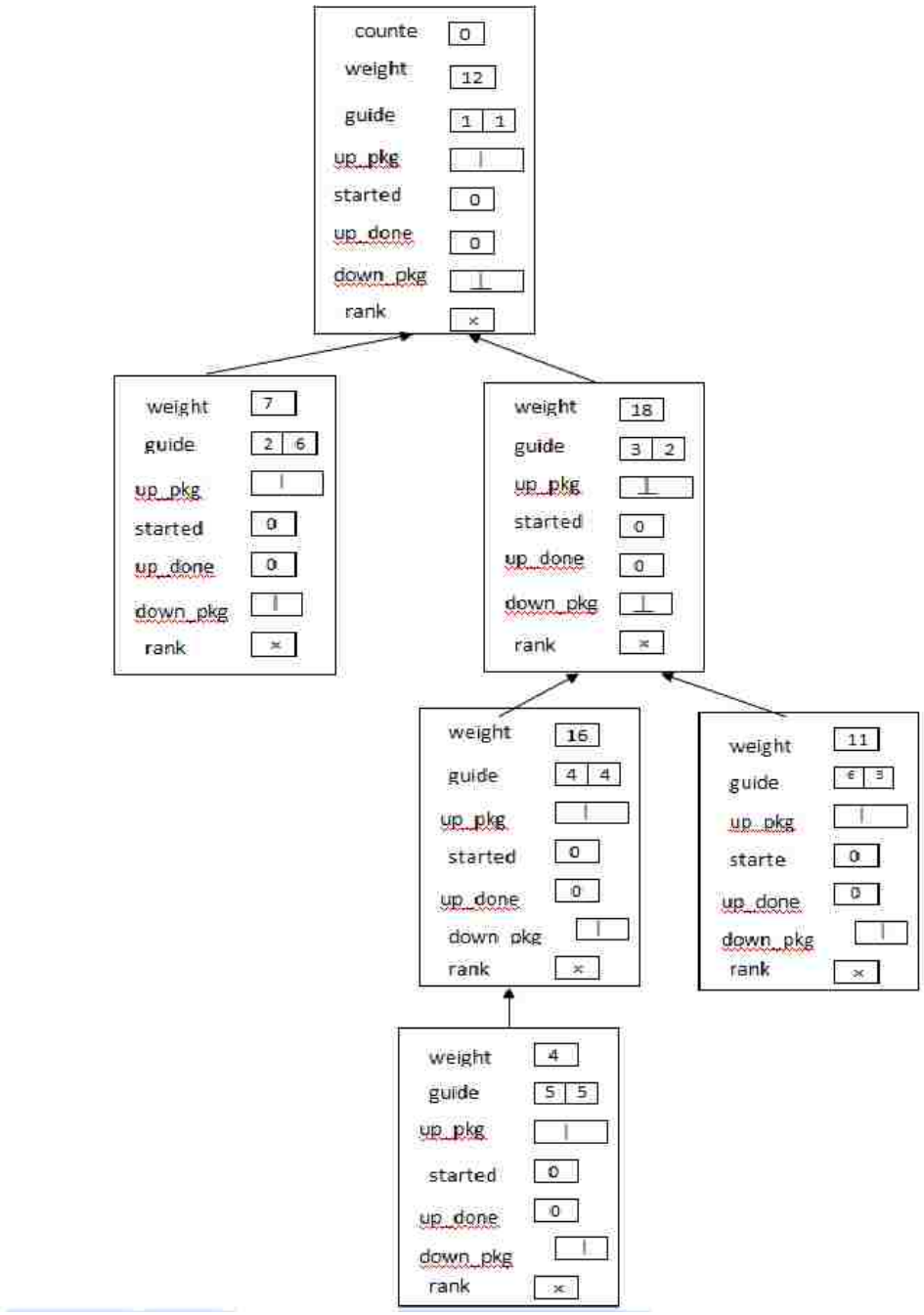
Node (17,39), (18,27), (30,24) computes the guide pair values and sends the pre and post predecessor values to their children



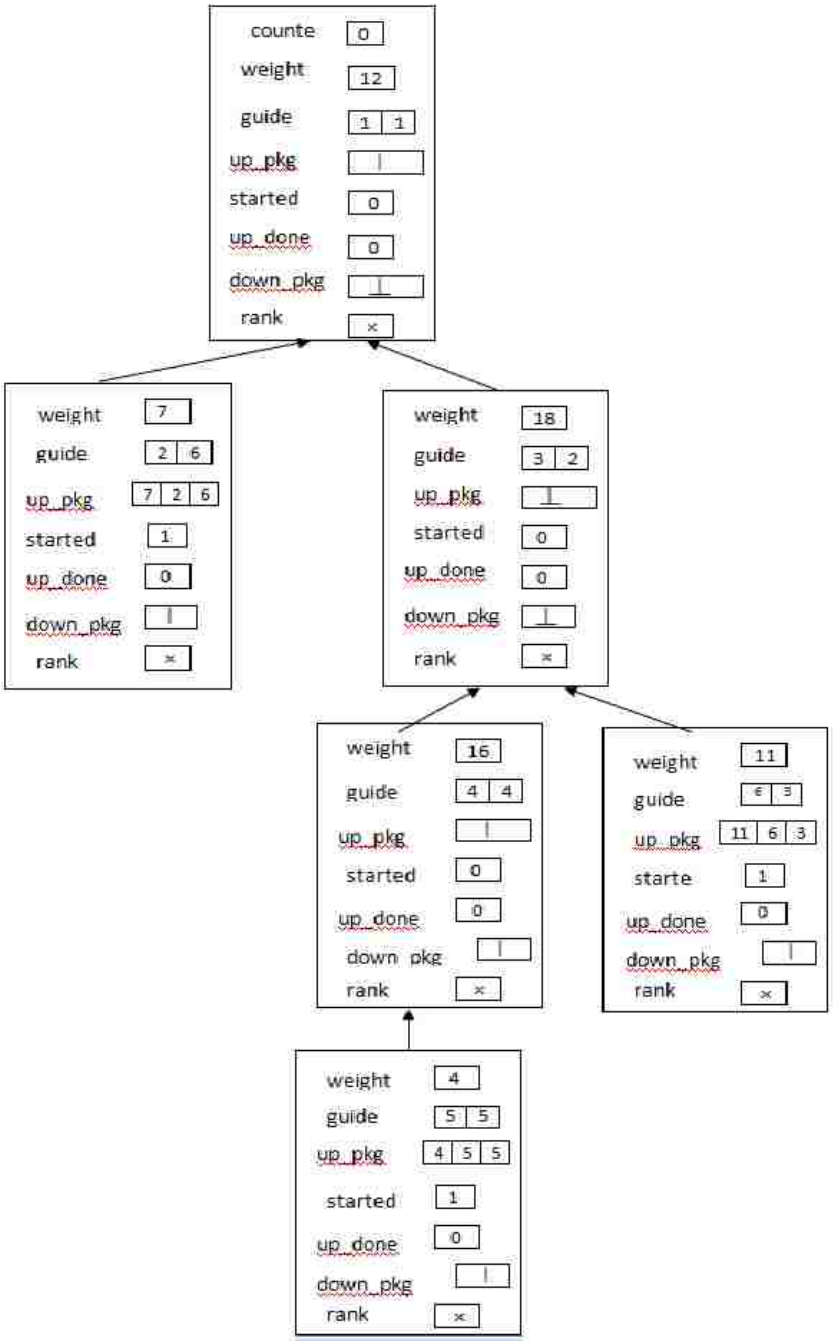
All the nodes have the guide pair values.

APPENDIX D

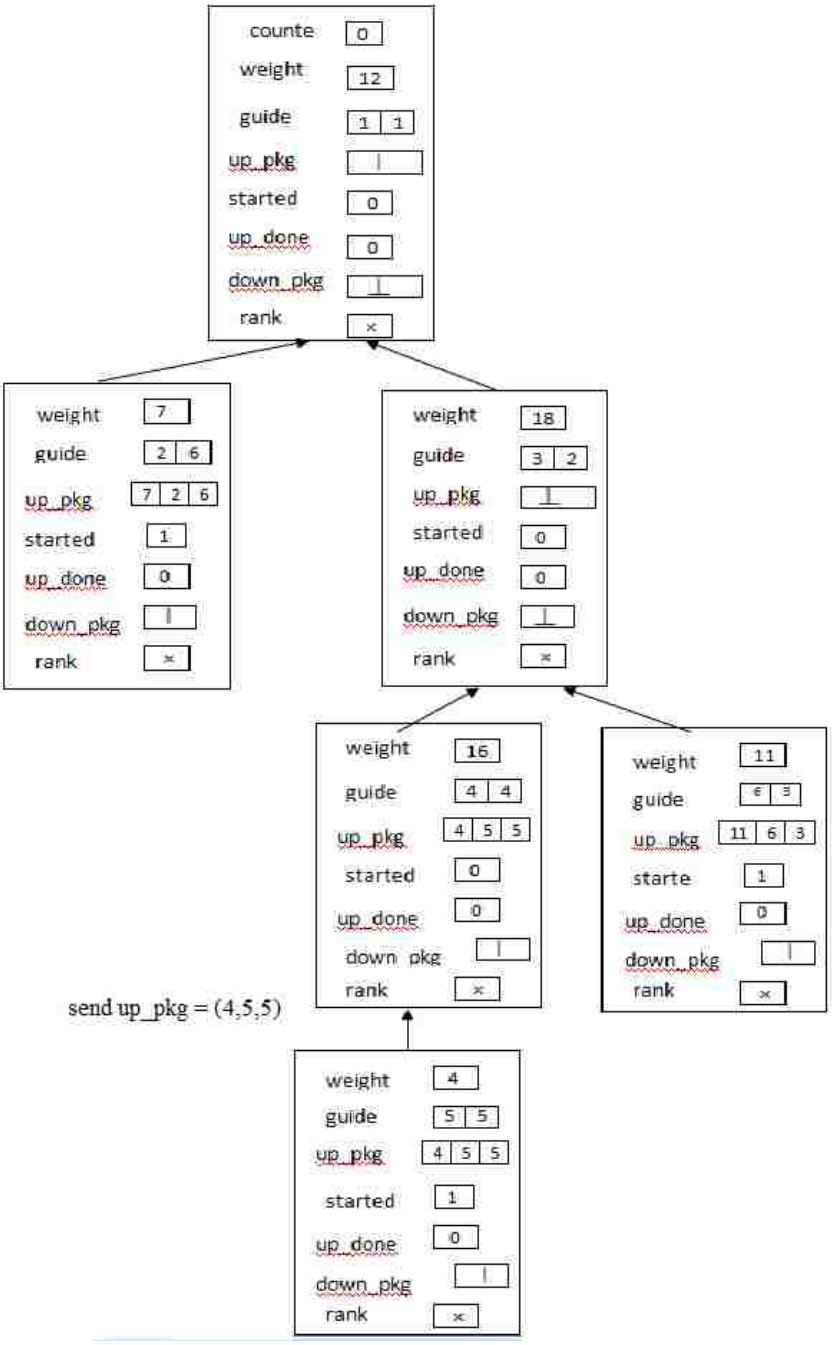
DIAGRAMS OF RANK



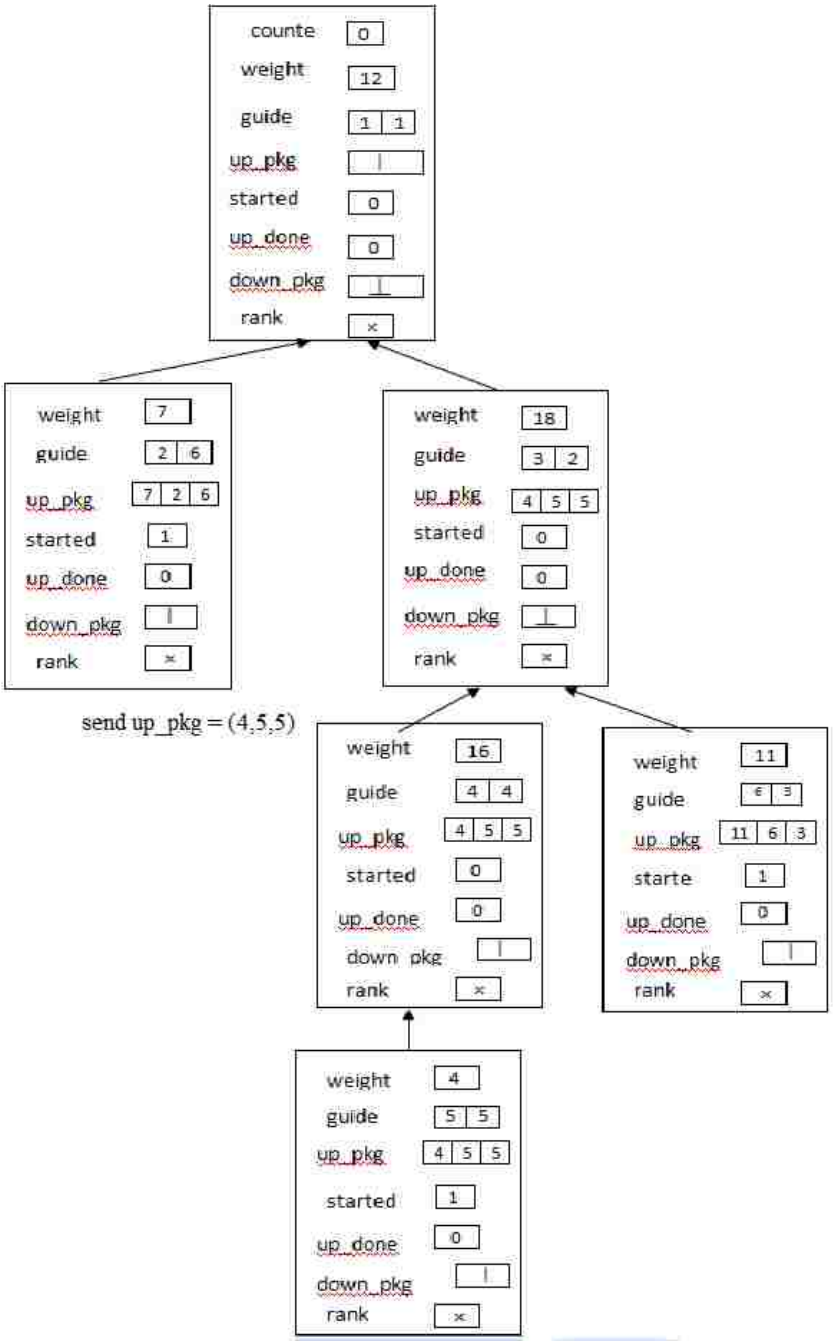
Initial configuration of the network at the beginning of an epoch



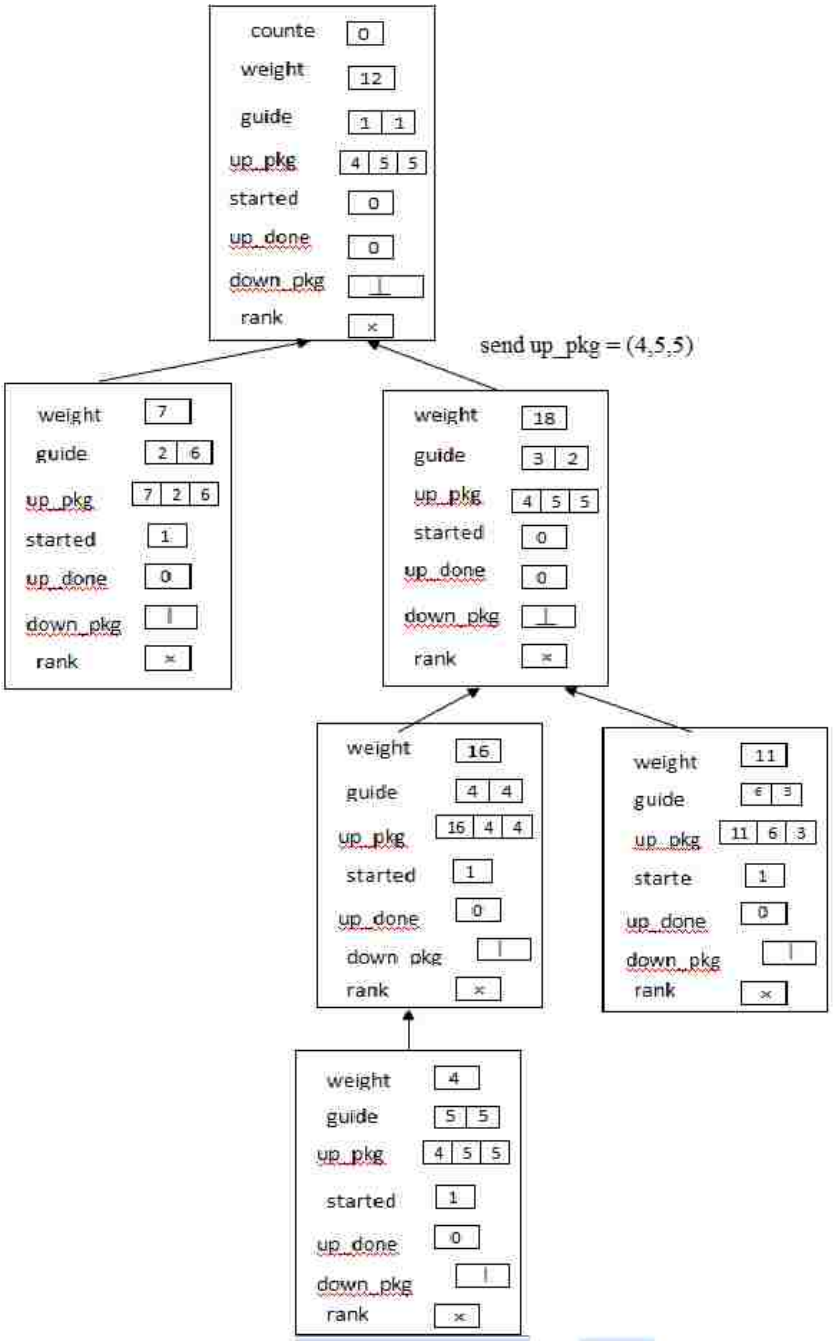
The three leaf processes, P(2,6), P(5,5) and P (6,3) starts their up-packages. P4,4 must wait until it is sure that all packages with smaller weights have been sent.



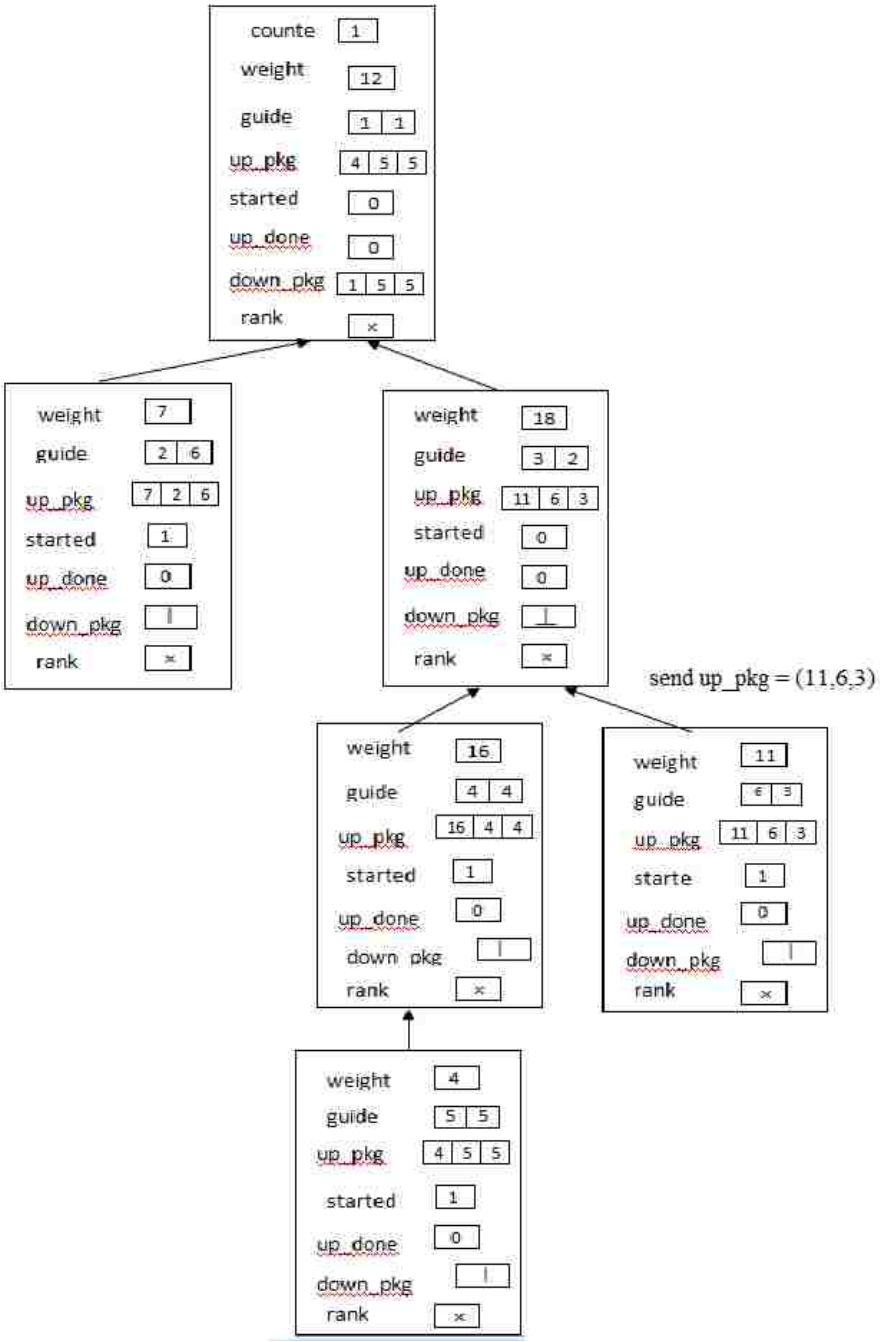
Only one package sends the up_pkg to its parent and the rest are blocked as they don't know if they are smallest or not.



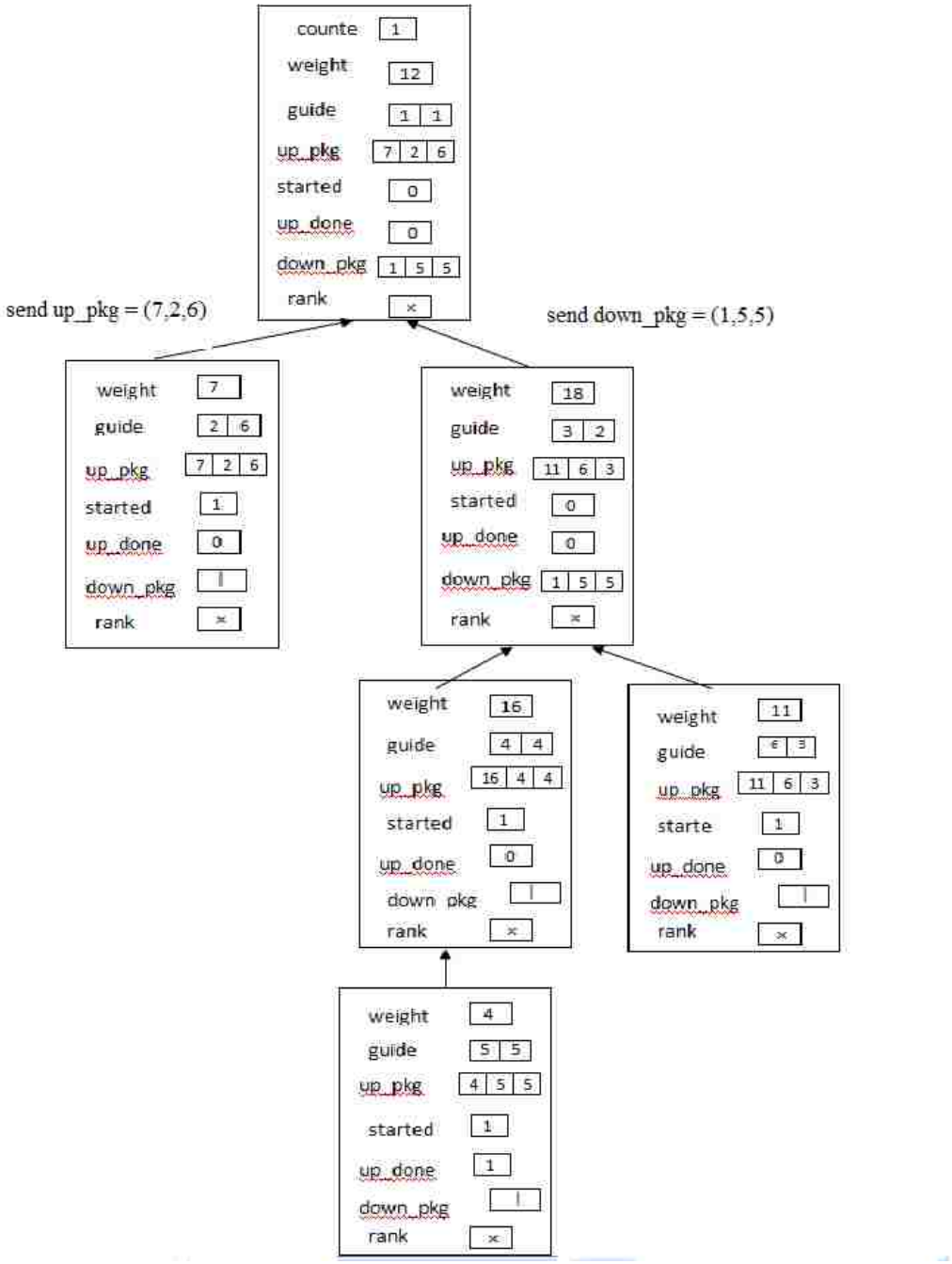
The package whose home process is $P_{5,5}$ sends its up_pkg up. $P_{4,4}$ is redundant now.



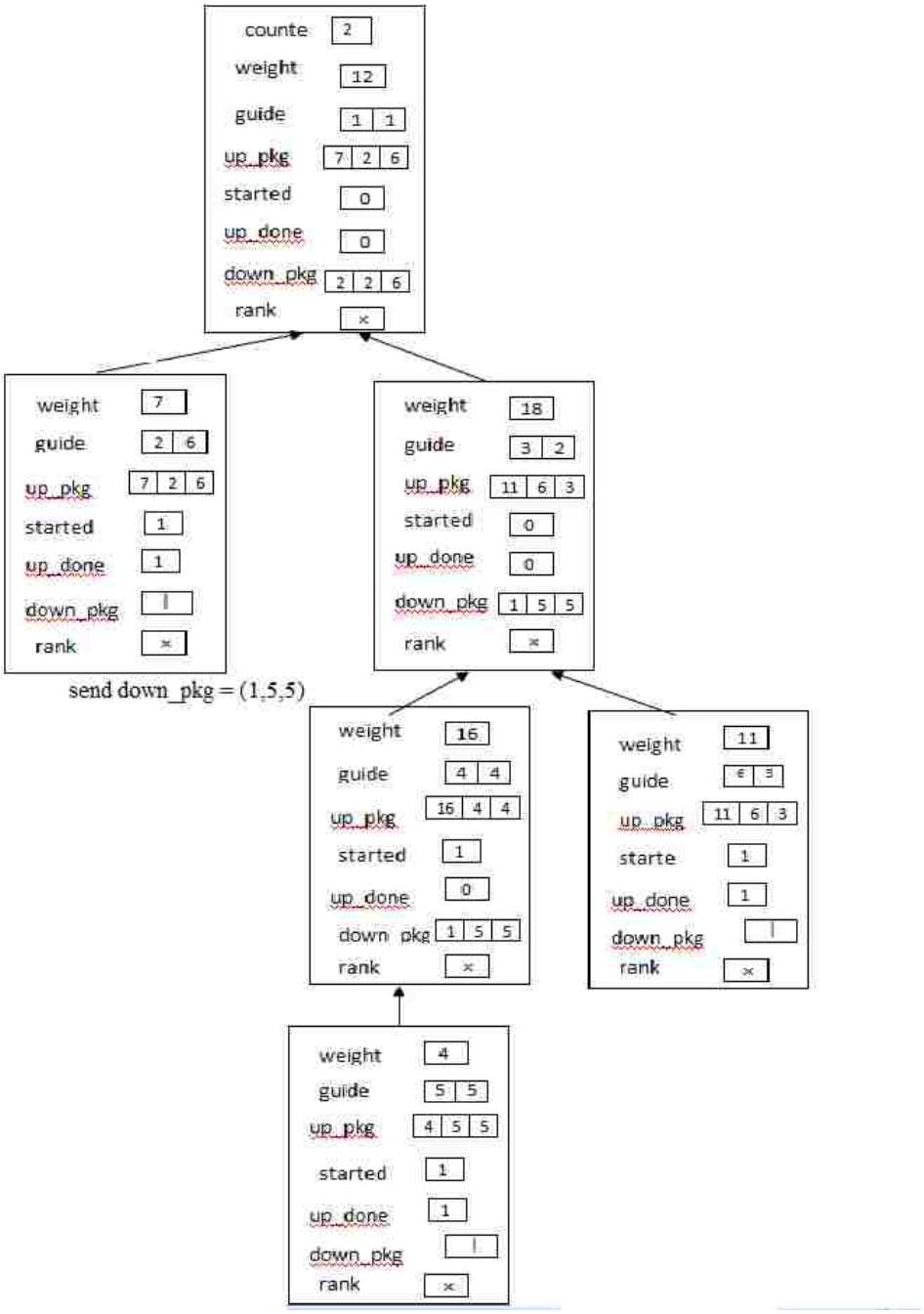
$P_{4,4}$ makes its up_pkg ready . Root has received the first up-package, and thus it will assign rank 1 to the process $P_{5,5}$



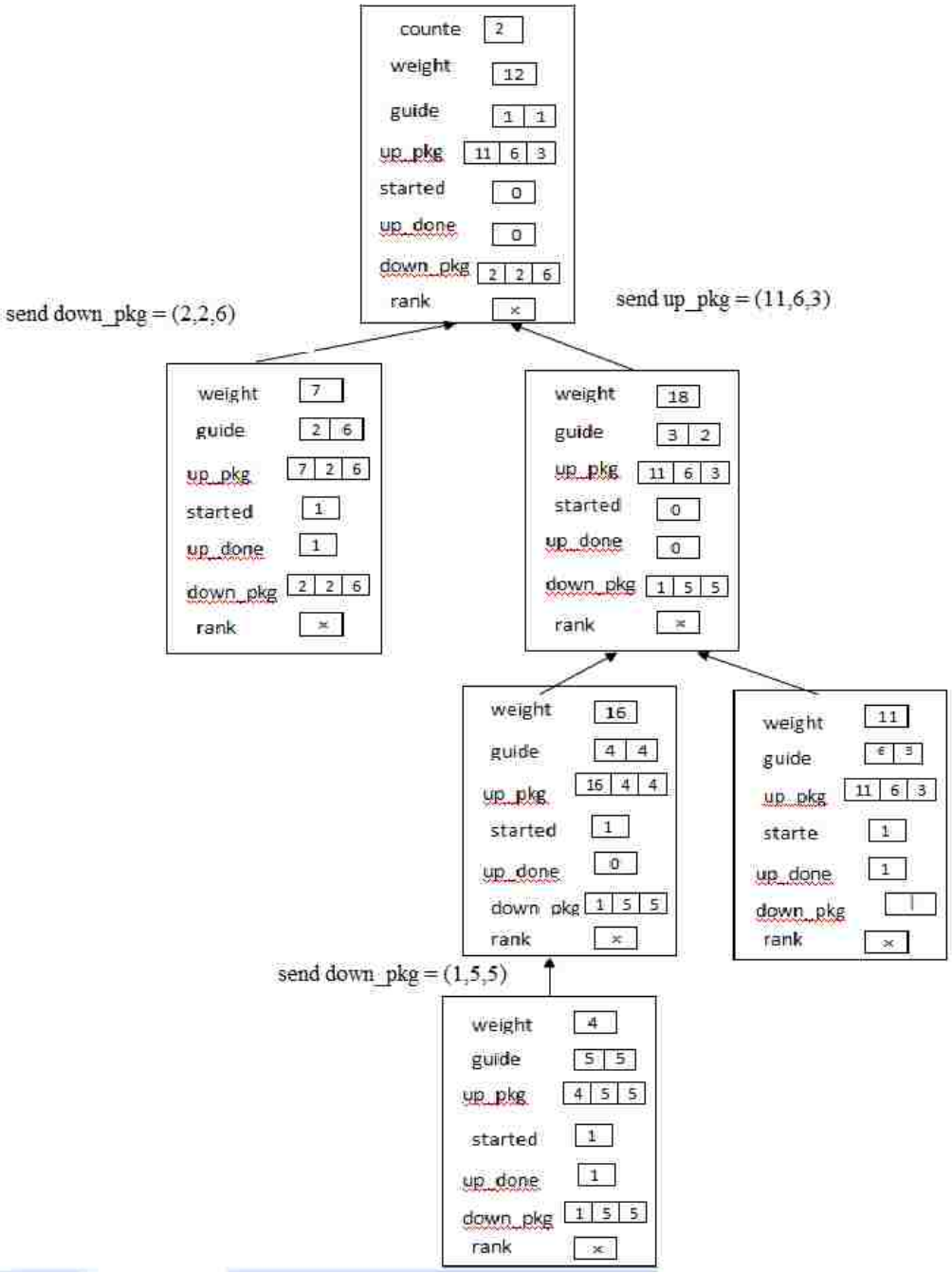
The up-package whose home is $P_{4,4}$ is still at $P_{4,4}$, and cannot be send to its parent, since there is another up-package in the way. Root creates the first down-package, in this step; counter increments to 1, and the new down-package carries value 1, indicating that $P_{5,5}$ has the smallest weight.



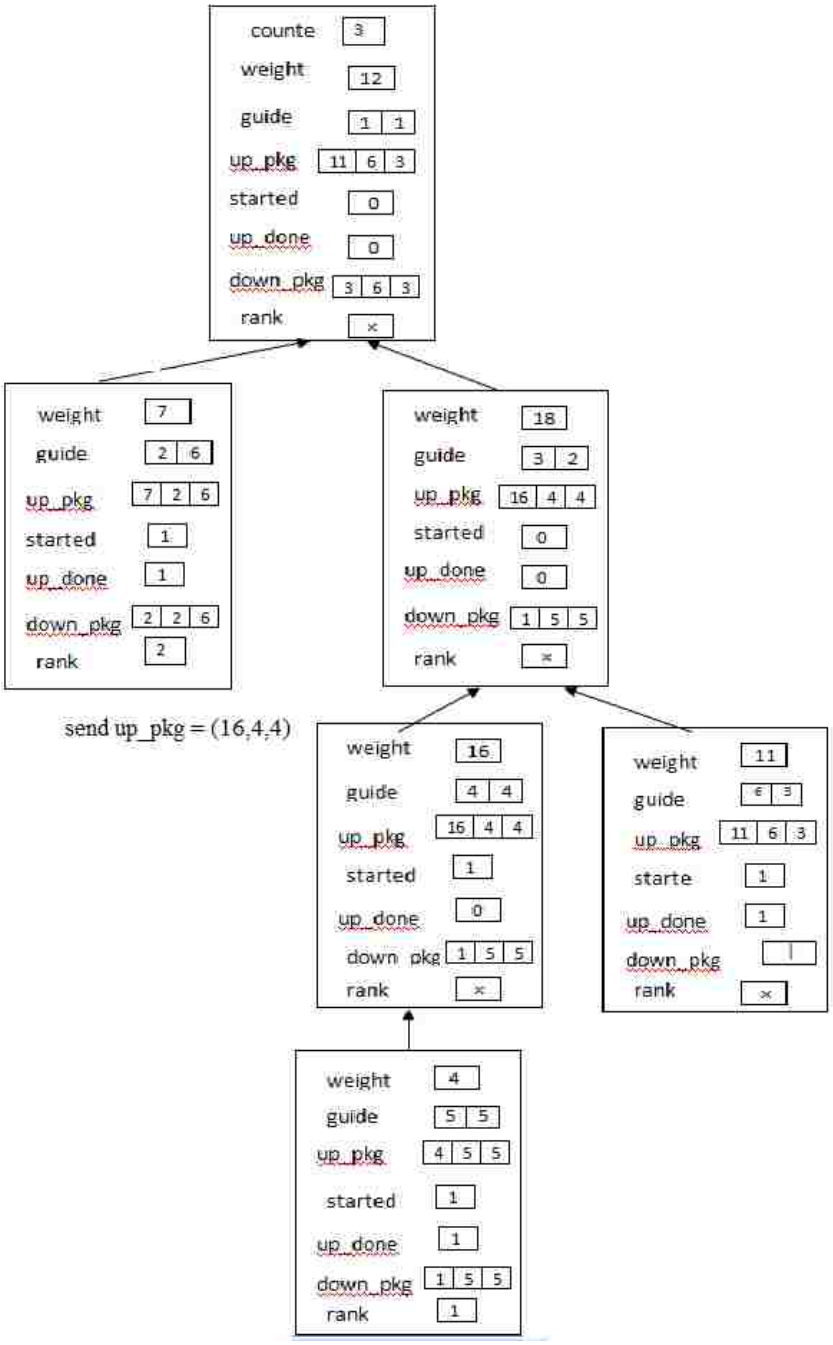
The $P_{4,4}$.up pkg still cannot be sent up, because $P_{6,3}$ has sent its up_pkg to the process $P_{3,2}$. The down_pkg whose home is $P_{5,5}$ is sent down to $P_{3,2}$.



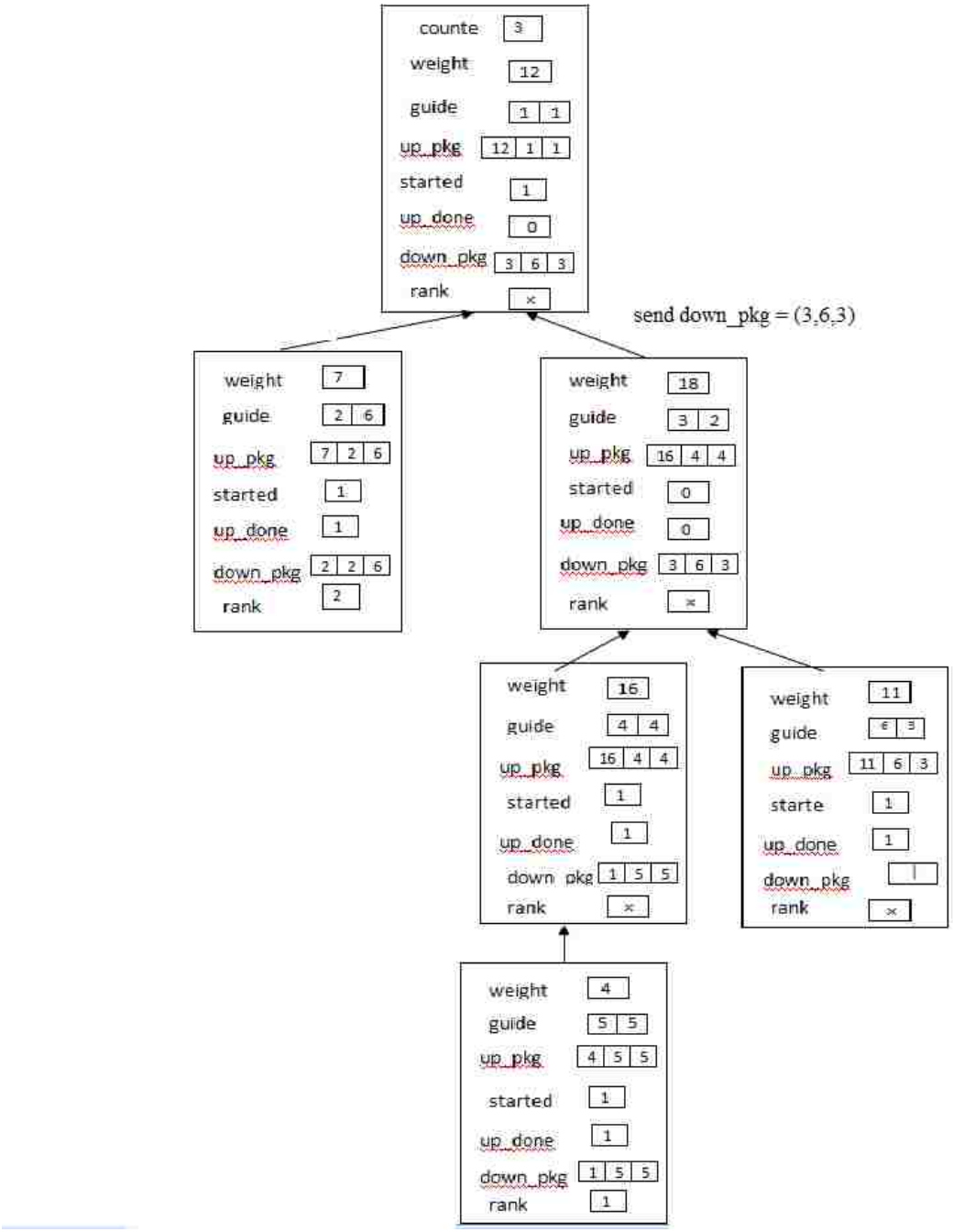
The up_pkg whose home is $P_{4,4}$ still cannot send it up. $P_{3,2}$ send its down_pkg to the process $P_{4,4}$



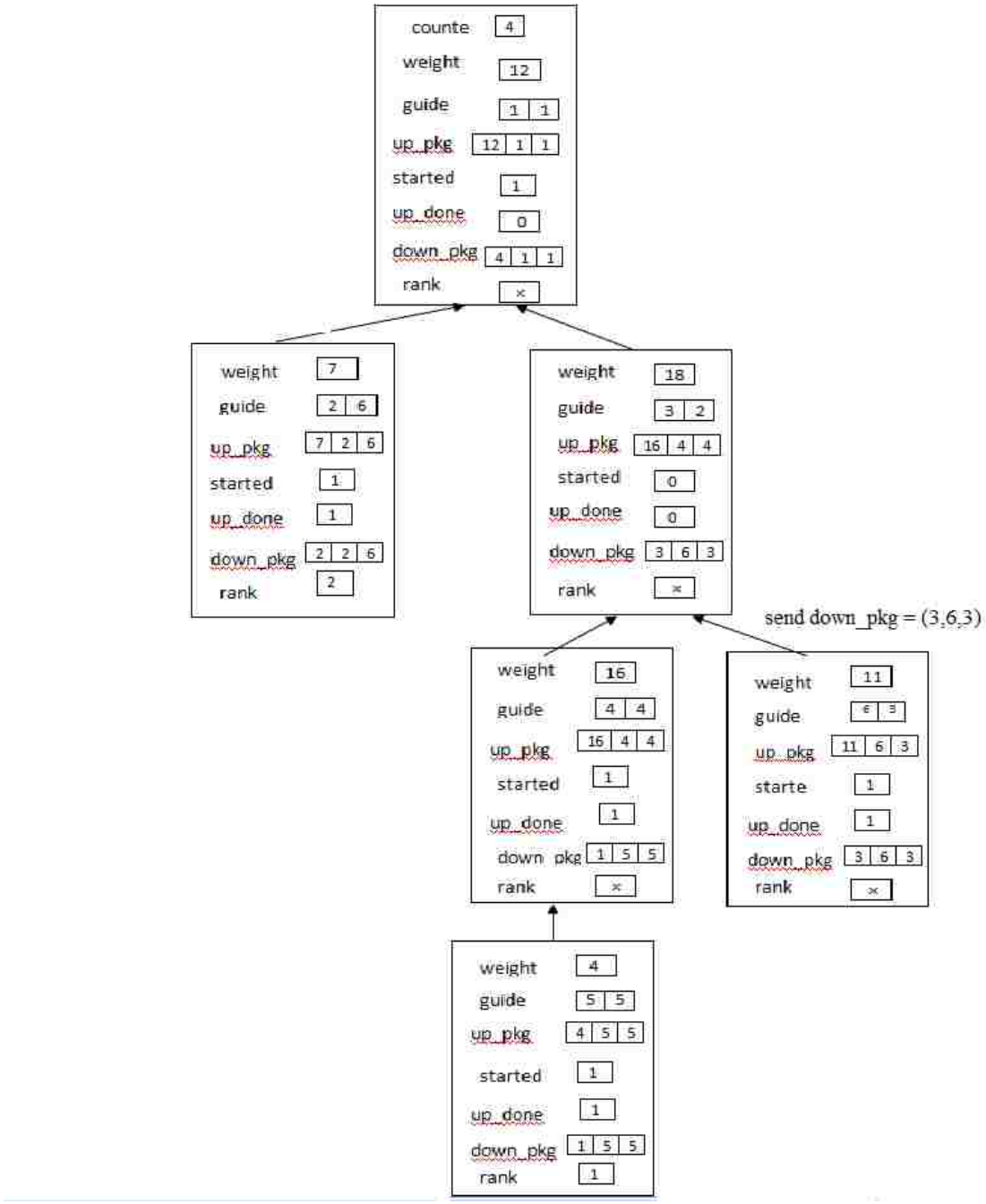
The down_pkg whose homes are $P_{2,6}$ and $P_{5,5}$ are sent to their home processes. The up_pkg whose home is $P_{4,4}$ is still stuck at $P_{4,4}$.



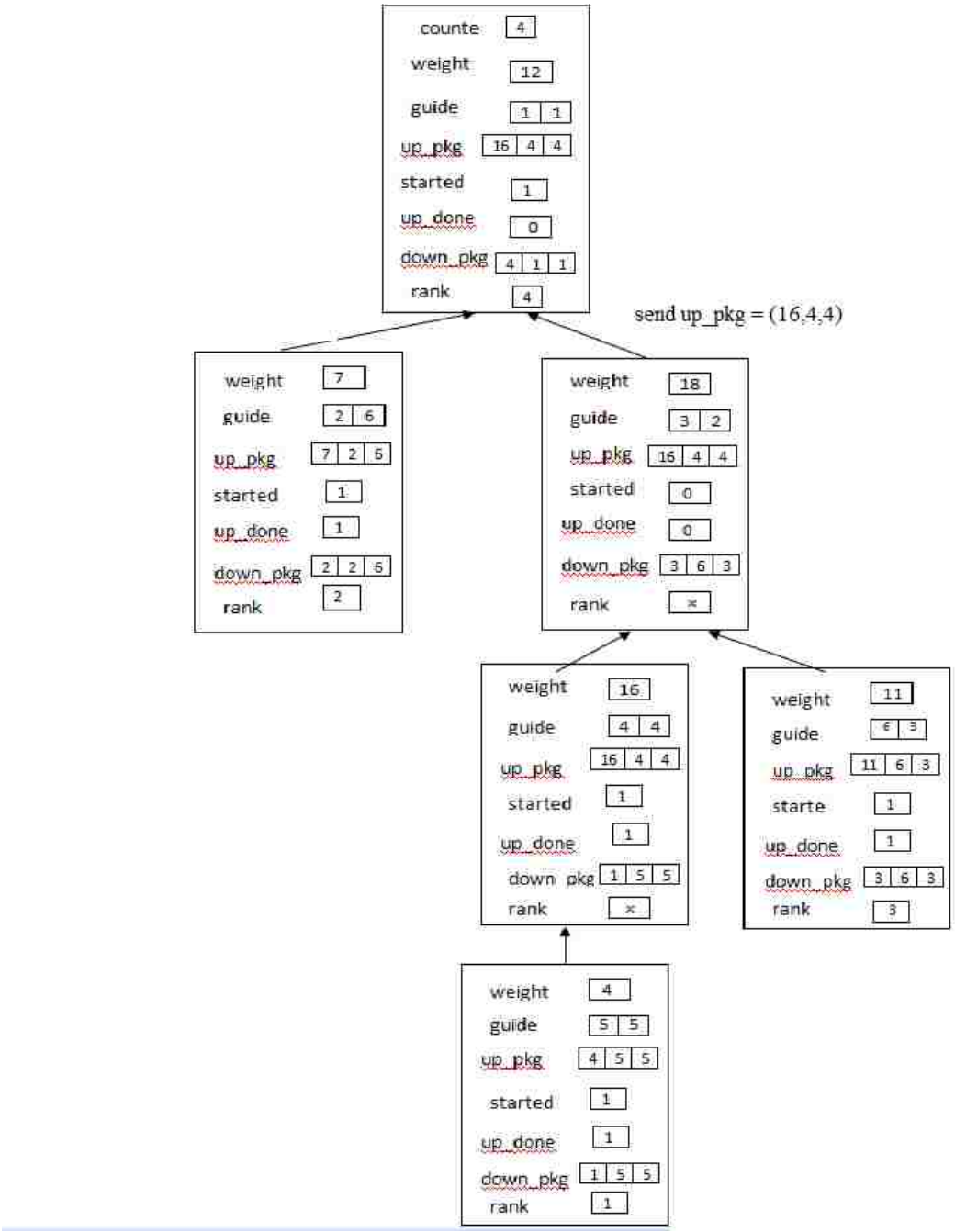
The up_pkg whose home is $P_{4,4}$ sends it to process $P_{3,2}$. To indicate that there are no more active up-packages in its subtree, $P_{4,4}$.up_done changes to 1. $P_{2,6}$ and $P_{5,5}$ copy their values of rank from their down-packages.



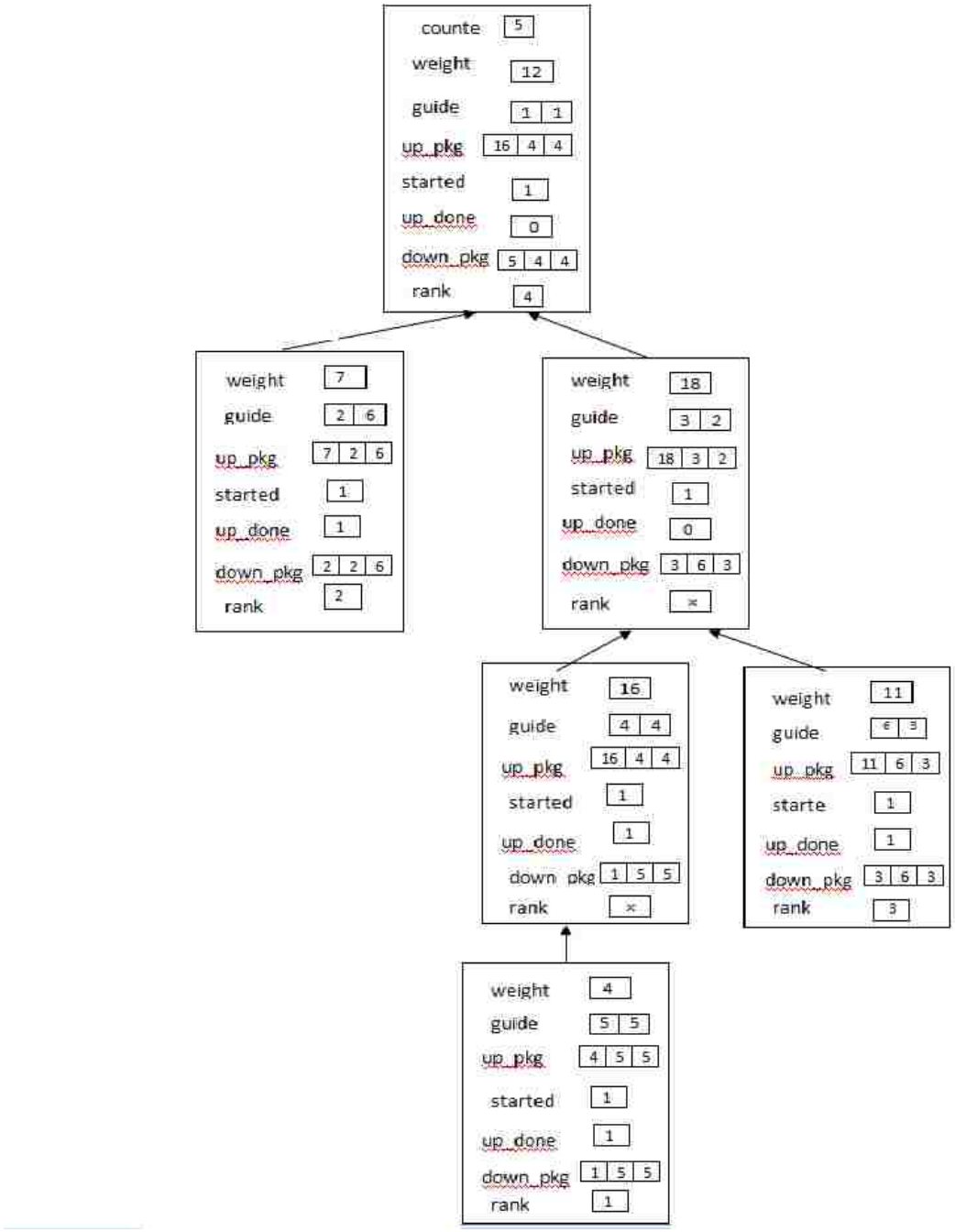
The up_pkg whose home $P_{4,4}$ is now stuck at $P_{3,2}$.



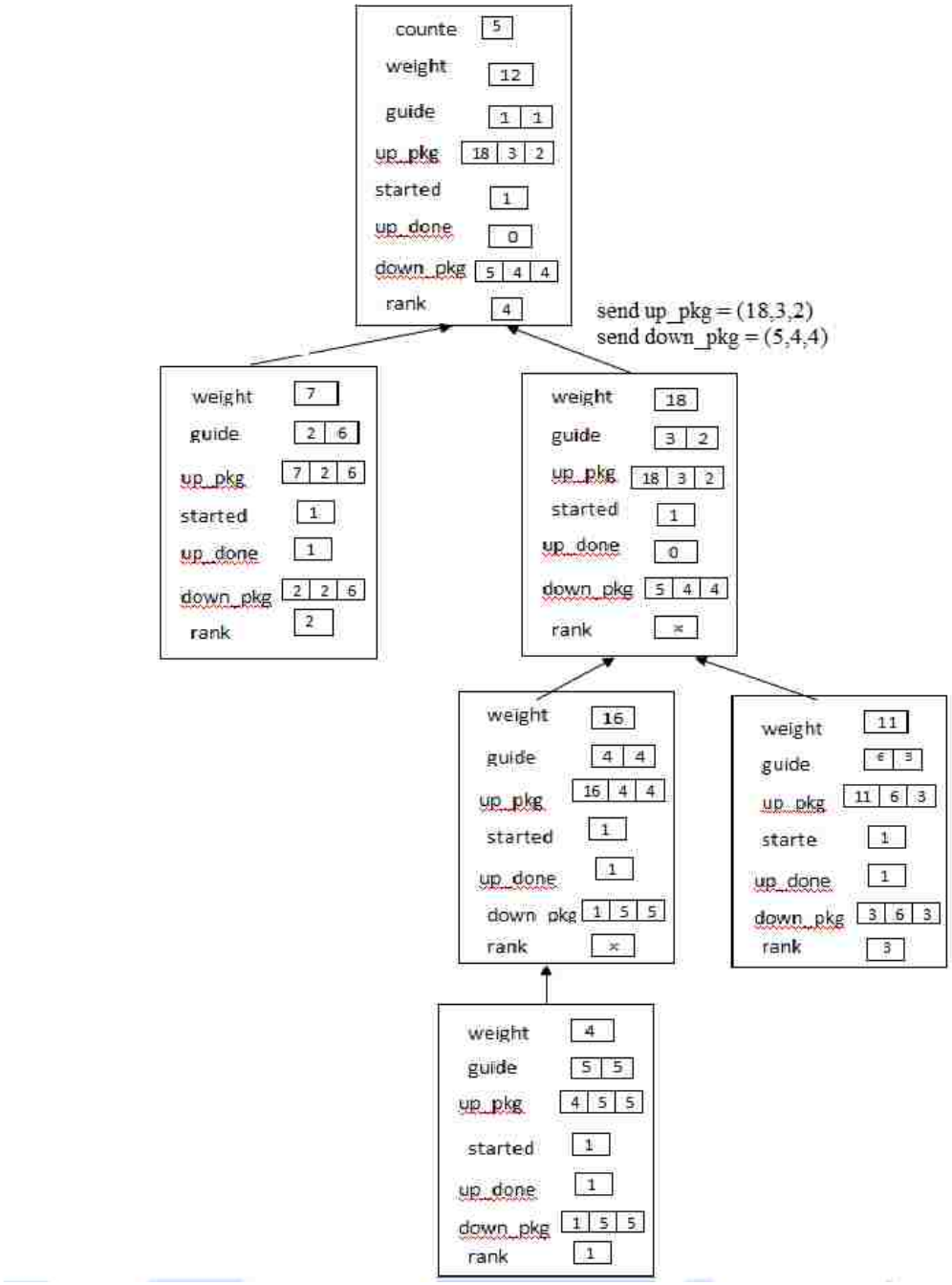
The up-package whose home is $P_{4,4}$ is still stuck at $P_{2,3}$. The process $P_{3,2}$ sends the down_pkg to its home process which is $P_{6,3}$.



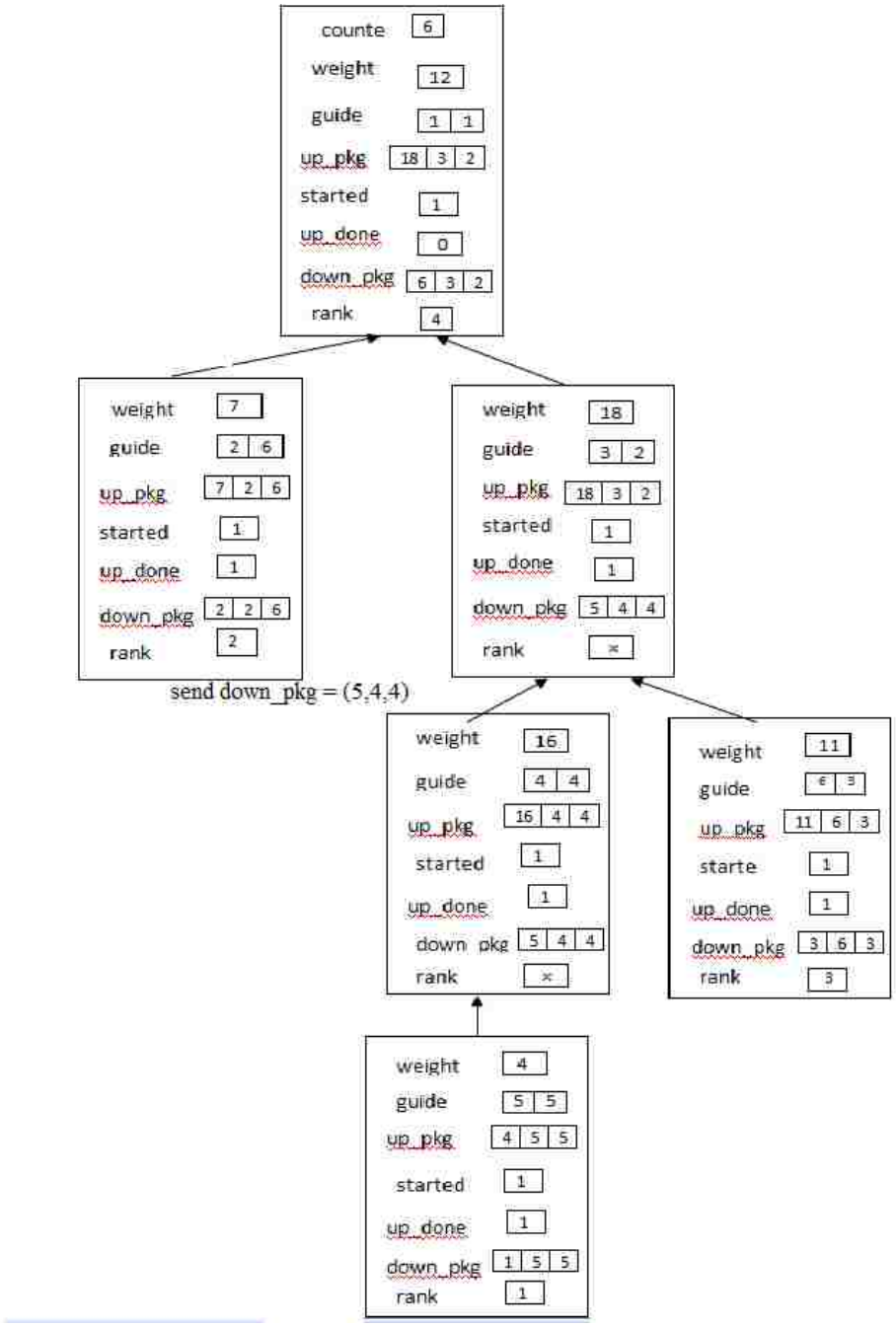
The up_pkg whose home is $P_{4,4}$ is sent to Root by the process $P_{3,2}$.



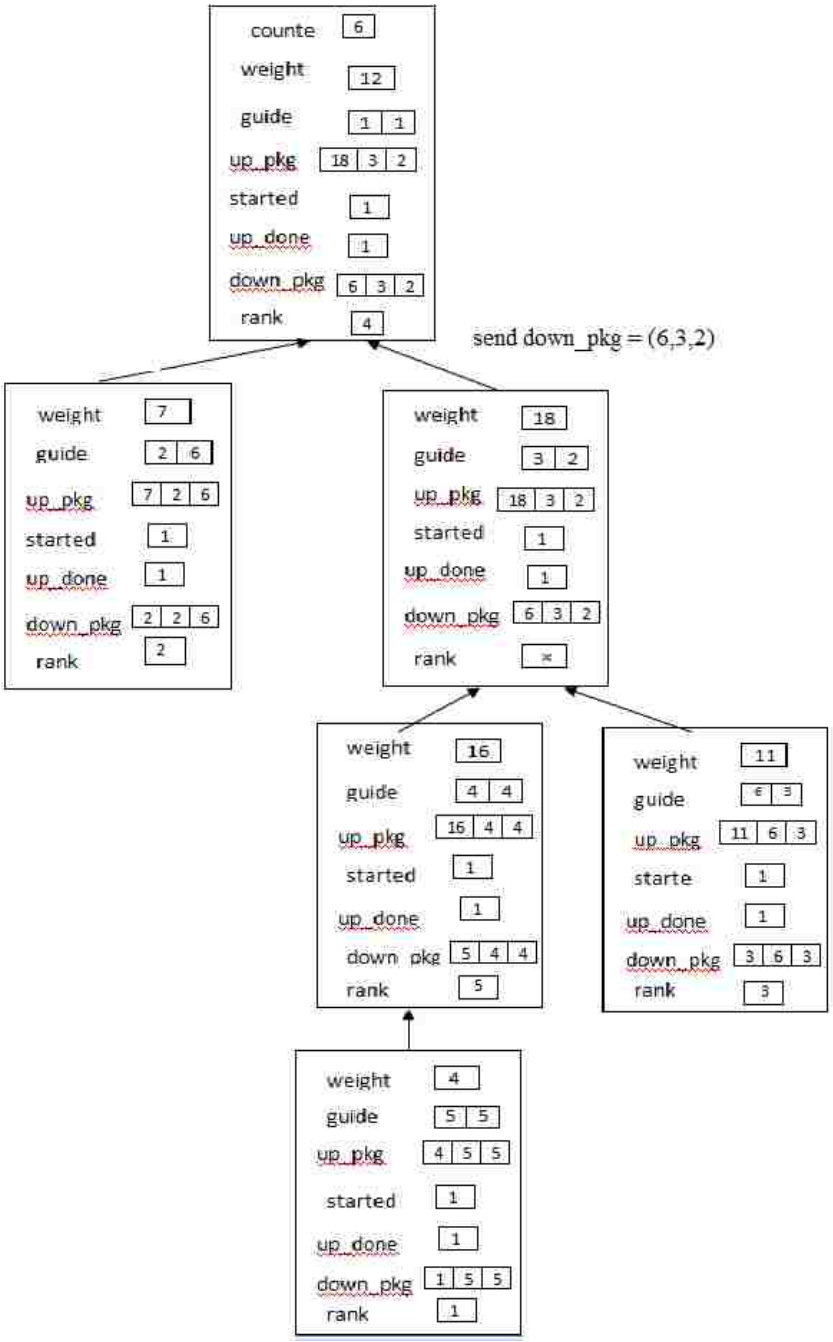
Root creates the down_pkg whose home process is $P_{4,4}$.



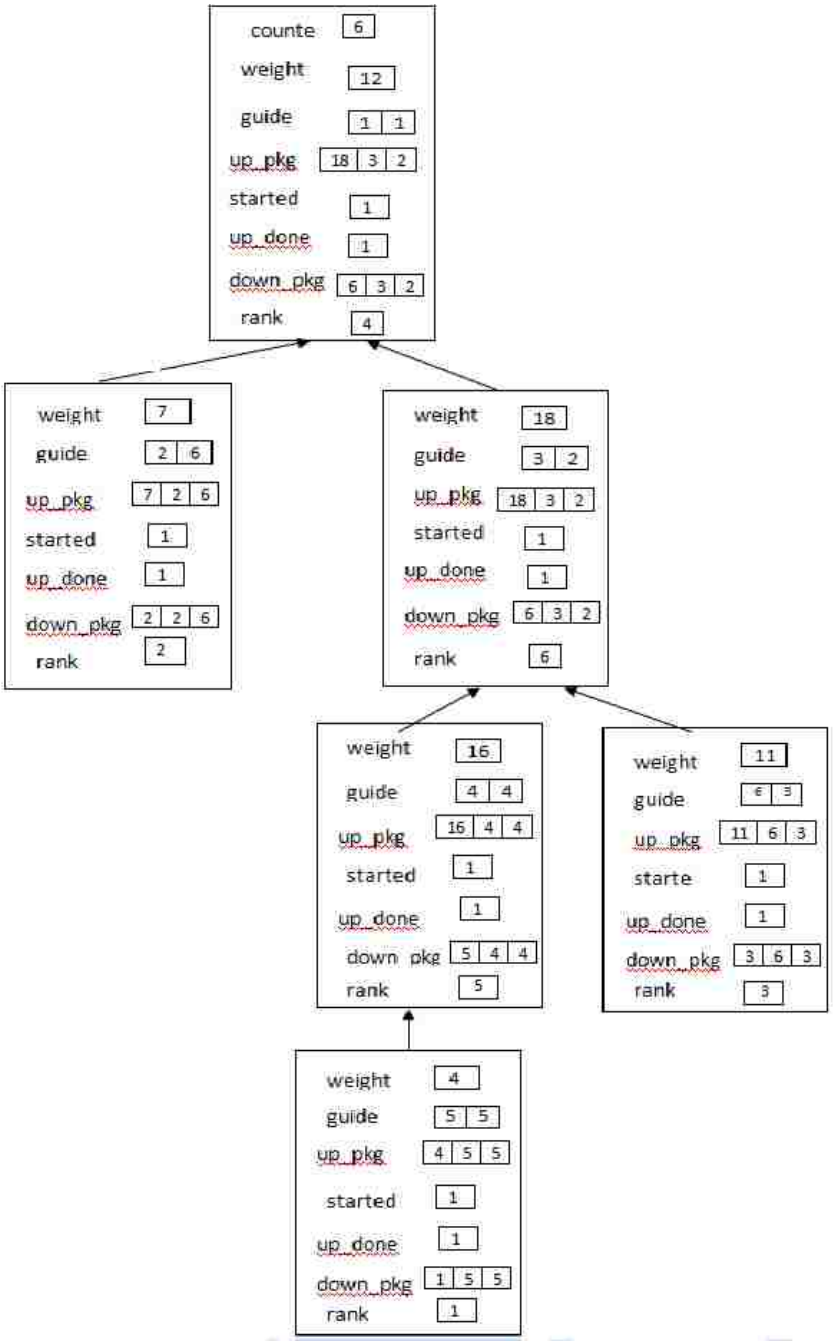
The down_pkg whose home is $P_{4,4}$ is sent to $P_{2,3}$ by the Root.



The down_pkg whose home is $P_{4,4}$ is sent to $P_{4,4}$ by $P_{3,2}$.



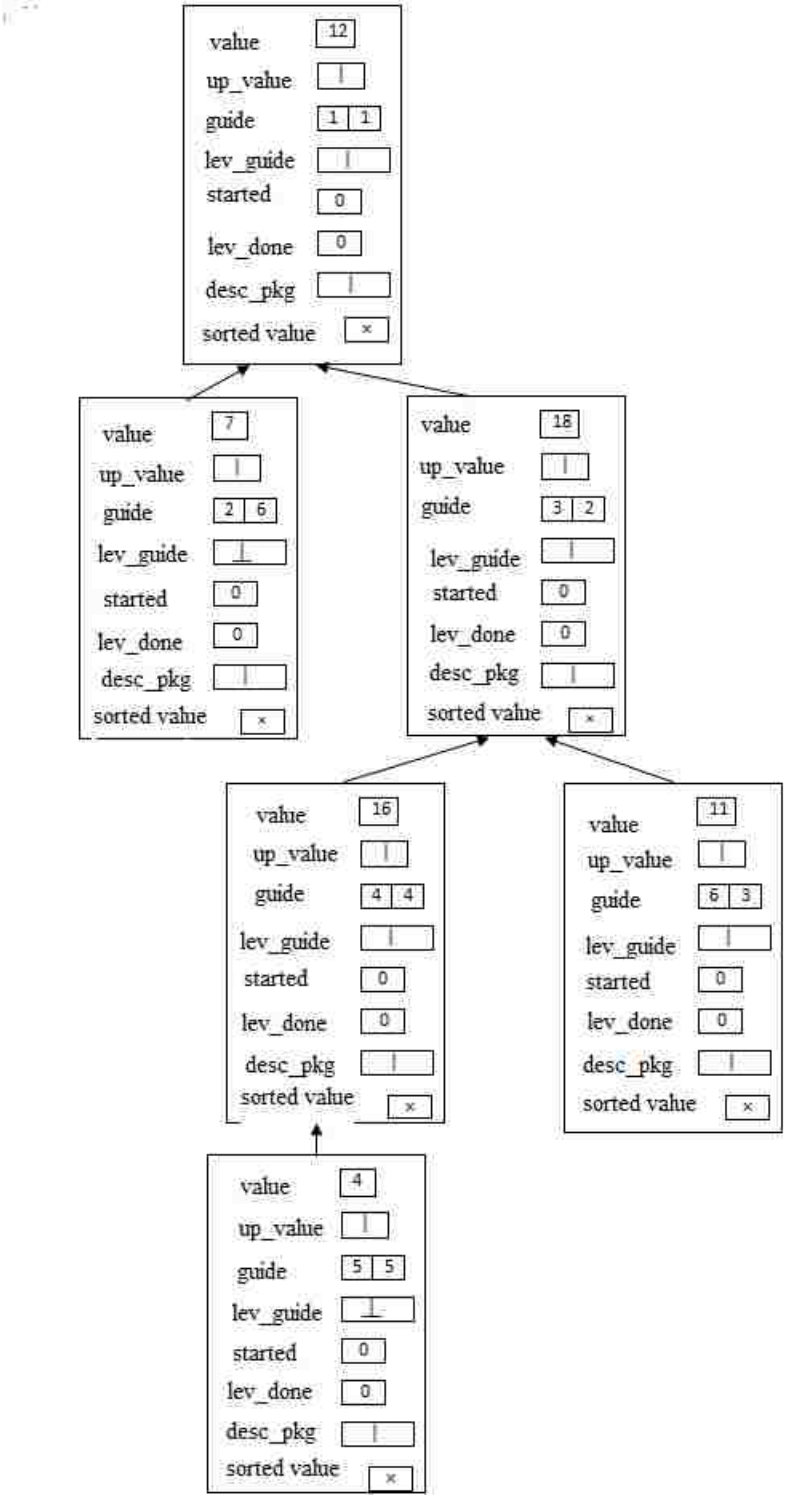
Process $P_{4,4}$ assigns rank to itself from its down_pkg.



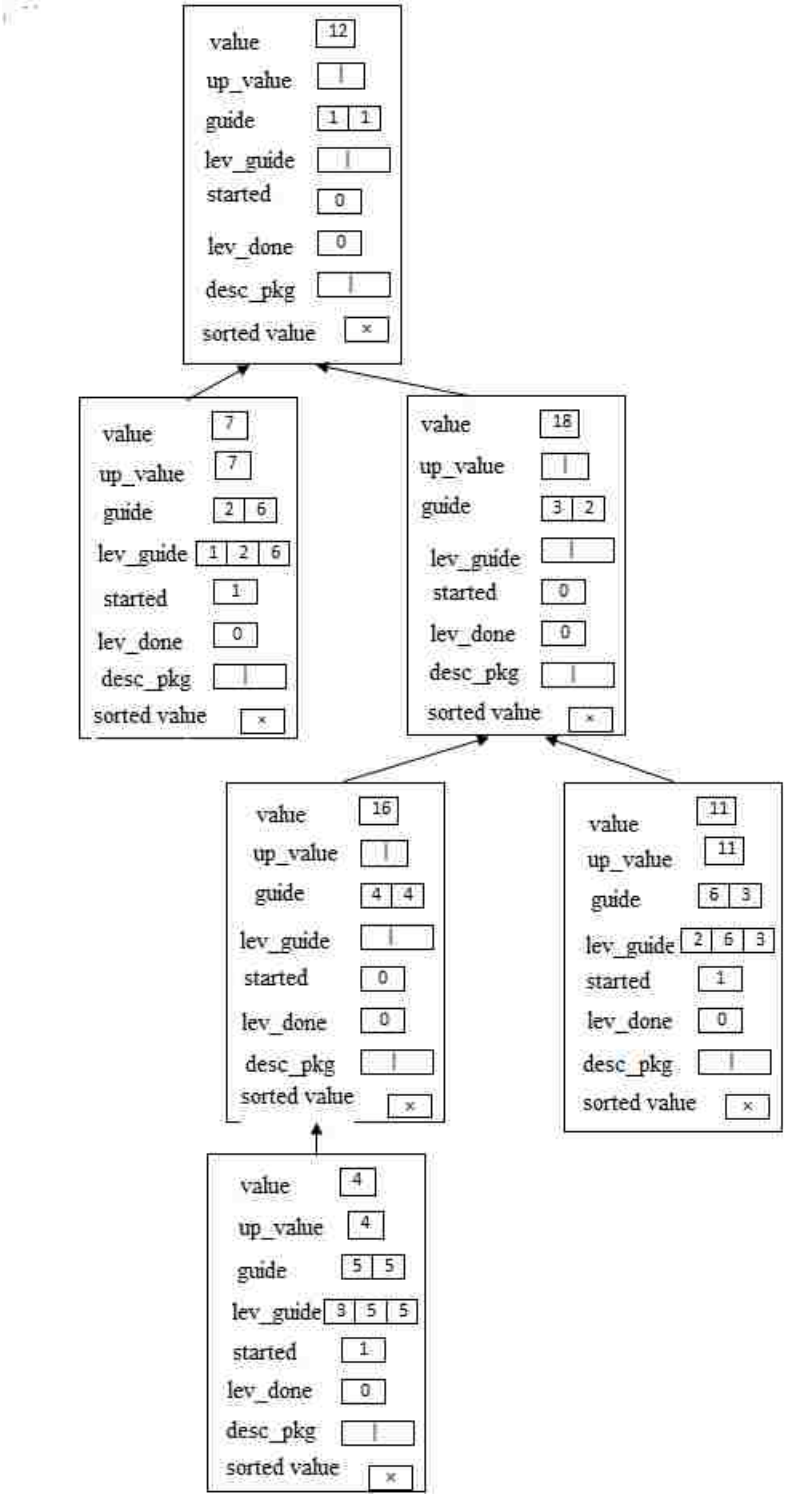
Now all the nodes in the tree has its rank values.

APPENDIX E

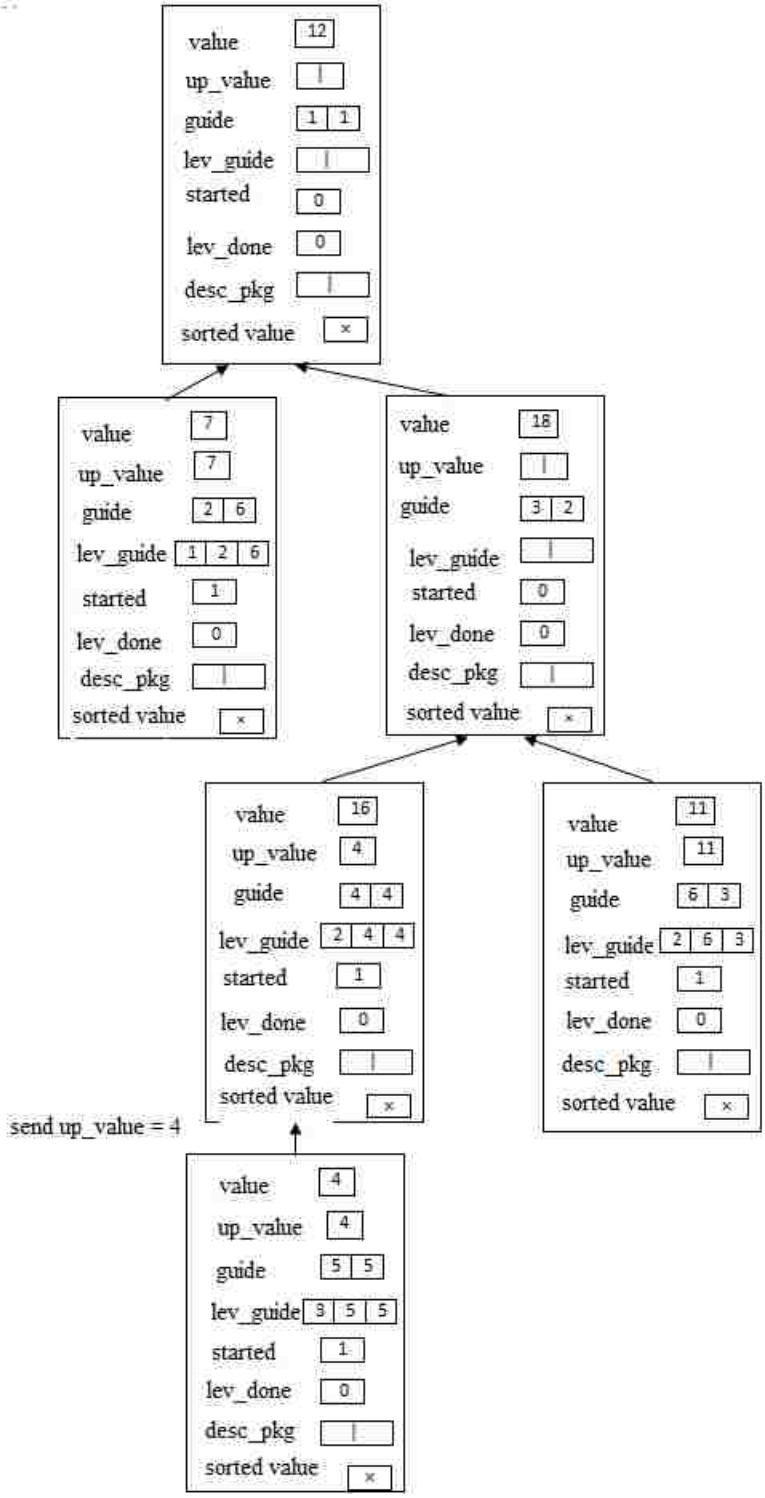
Diagrams of LEVEL ORDER SORTING



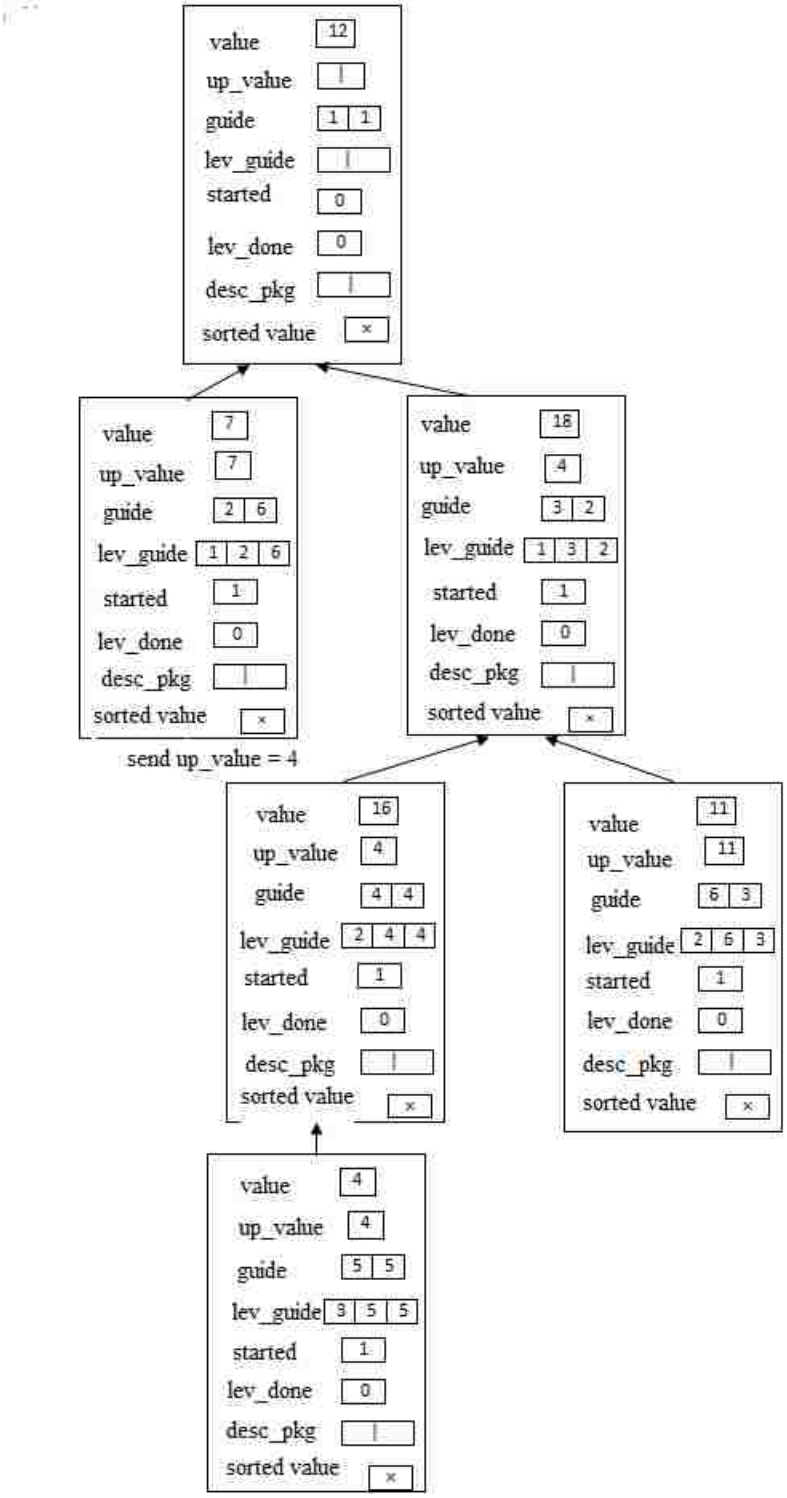
Initial configuration of the network at the beginning of and epoch



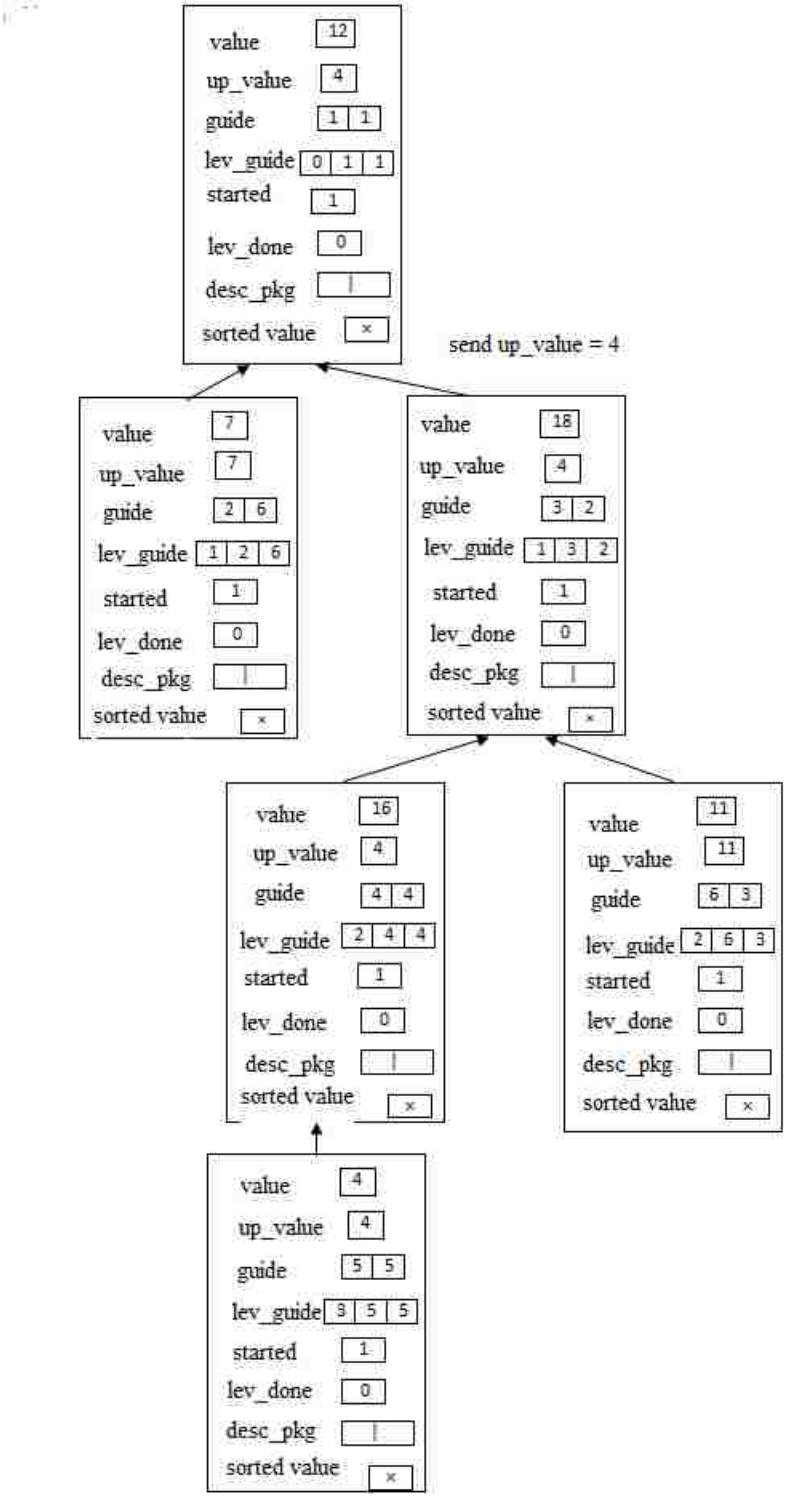
The three leaf processes, $P_{2,6}$, $P_{5,5}$ and $P_{6,3}$ starts their lev_guide and up_value. P4,4 must until it is sure that all packages with smaller weights have been sent up.



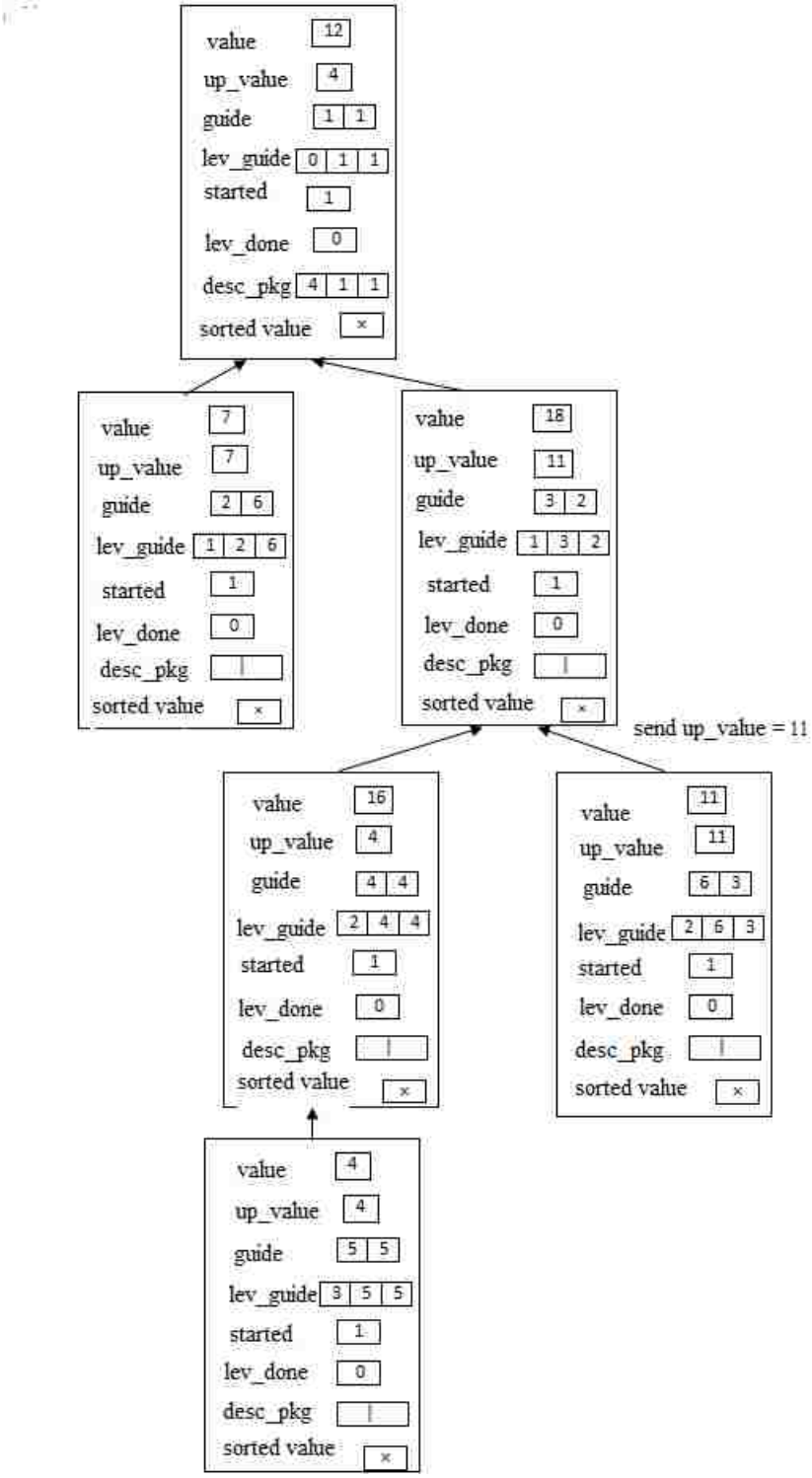
Only one process sends the up_value to its parent and the rest are blocked as they don't know which is the smallest value. The process $P_{4,4}$ makes its lev_guide value ready.



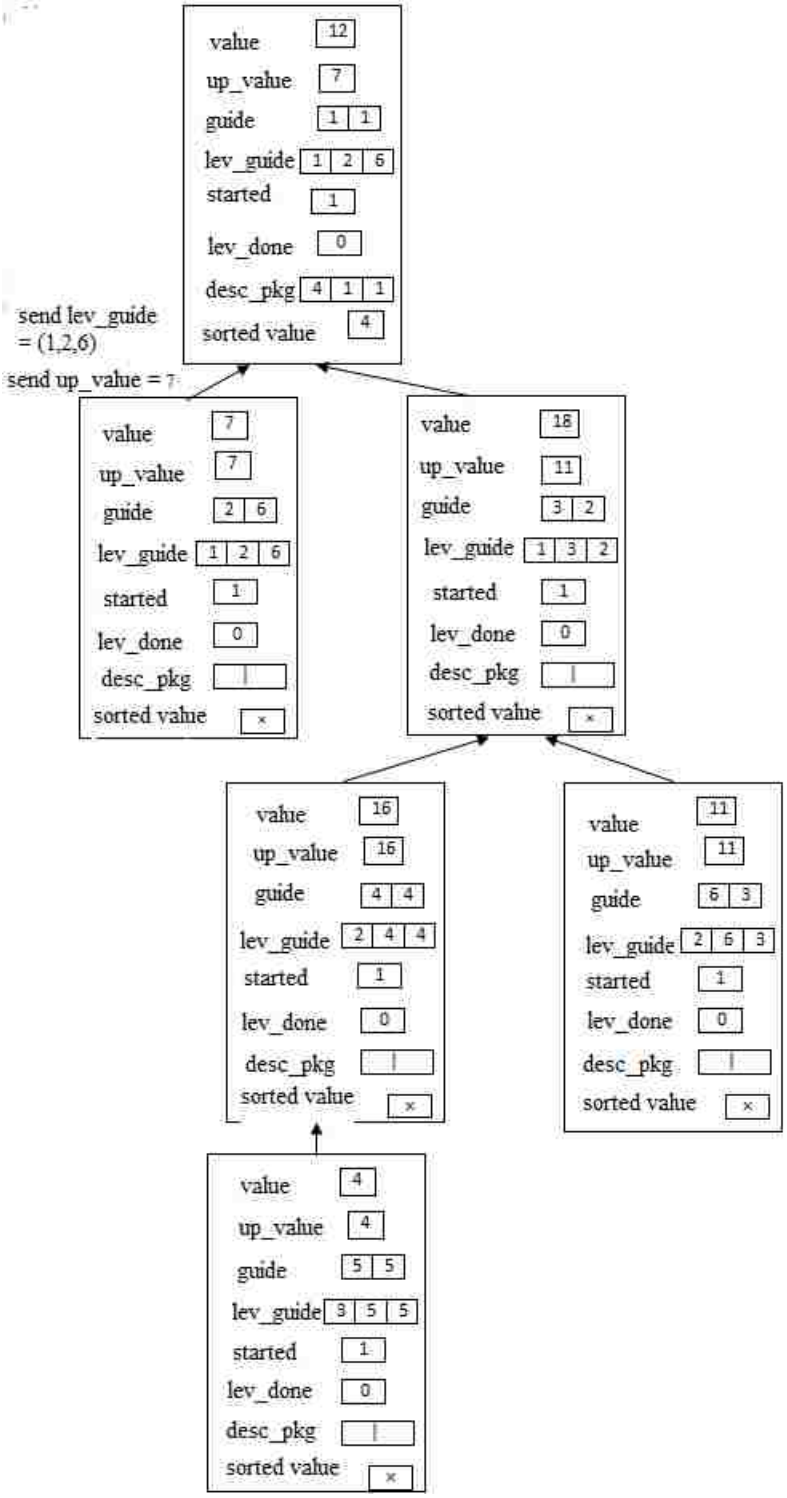
The package whose home process is $P_{5,5}$ sends its up_value up. $P_{4,4}$ is redundant now. The process $P_{3,2}$ makes its lev_guide value ready.



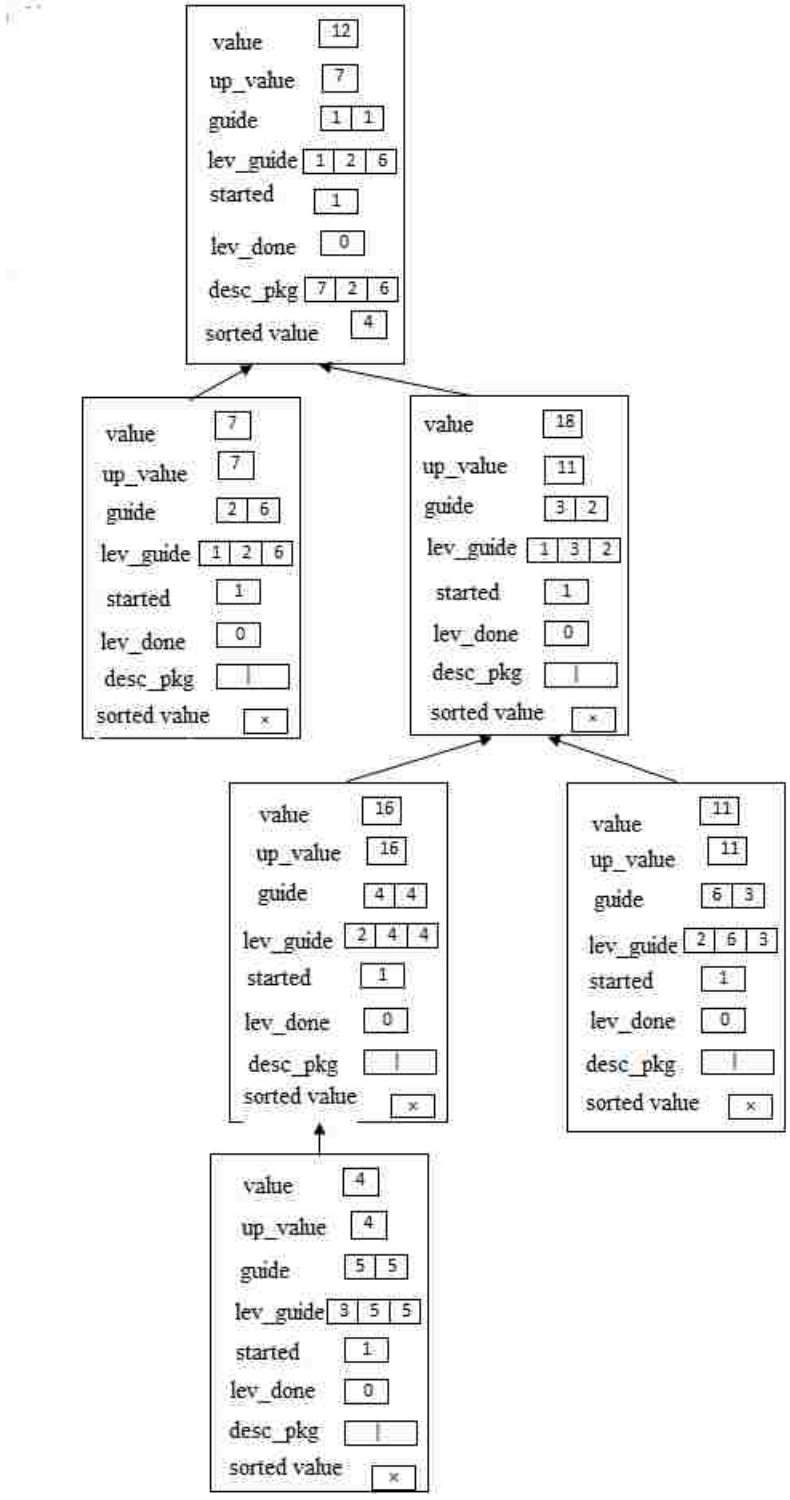
The package whose home process is $P_{5,5}$ sends its up_pkg up to the root. The root creates it lev_guide value.



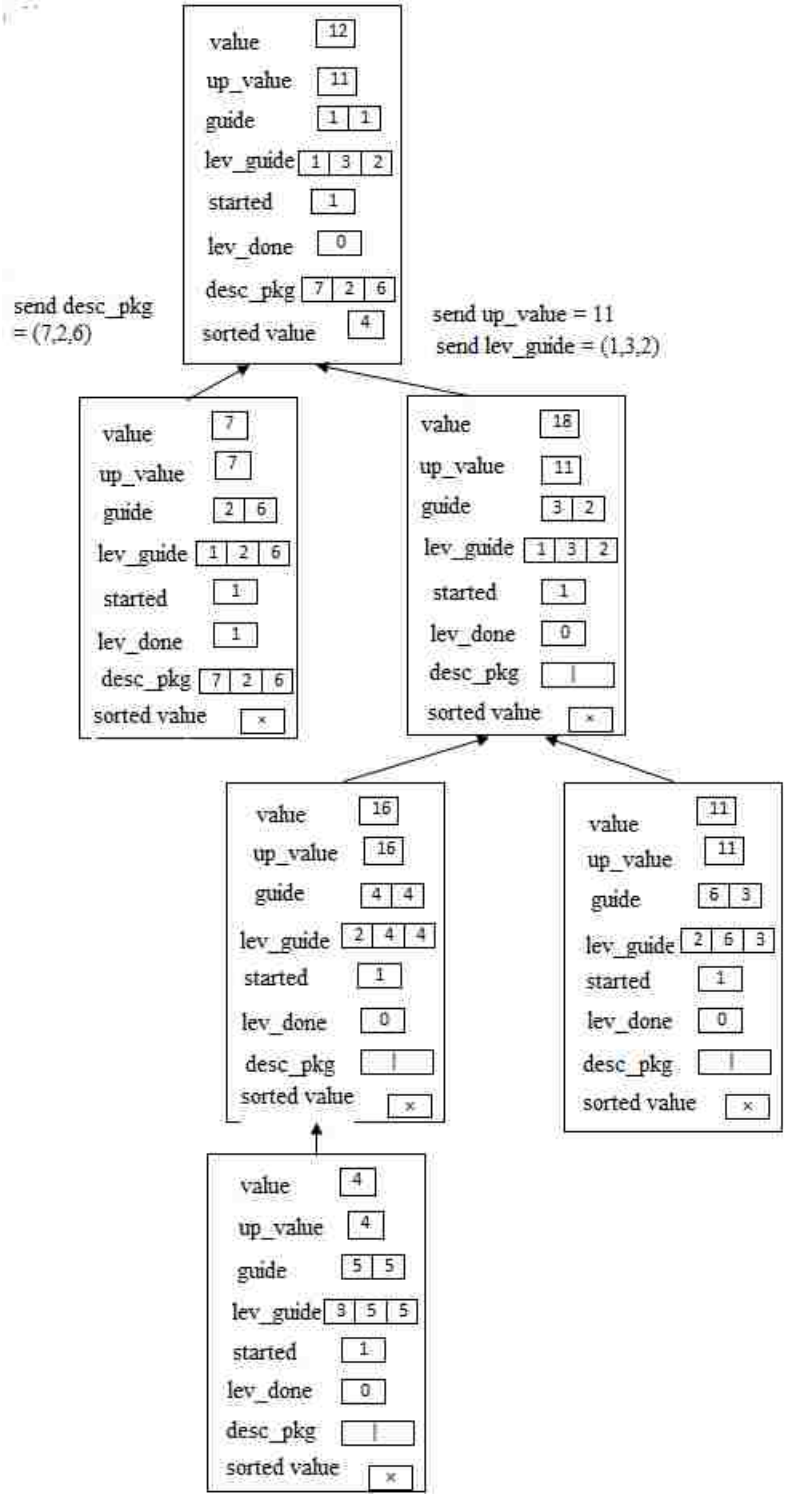
The Root creates the desc_pkg value.



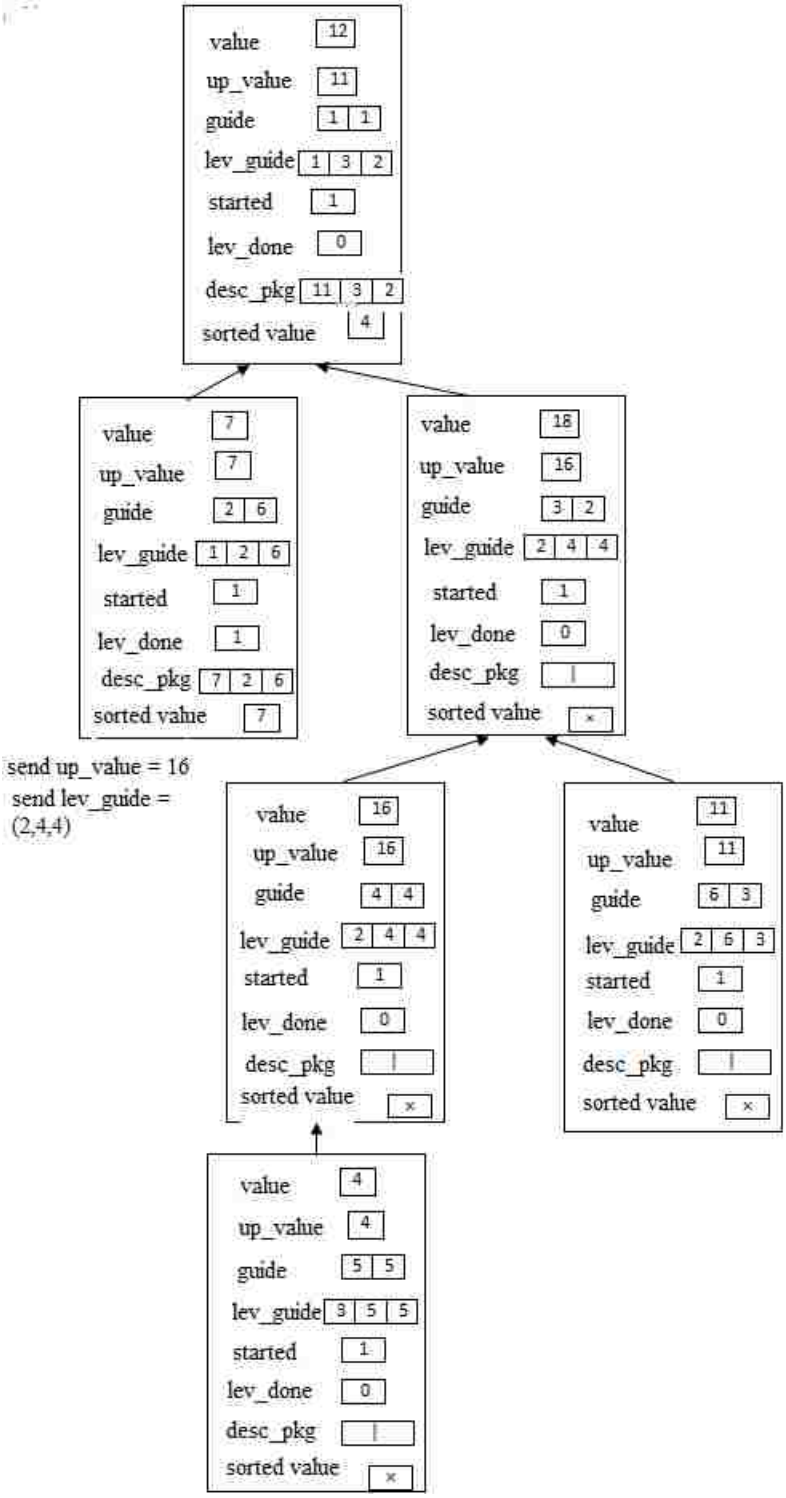
The Root assigns a value to its sorted value. $P_{2,6}$ sends its up_value, lev_guide values to its parent.



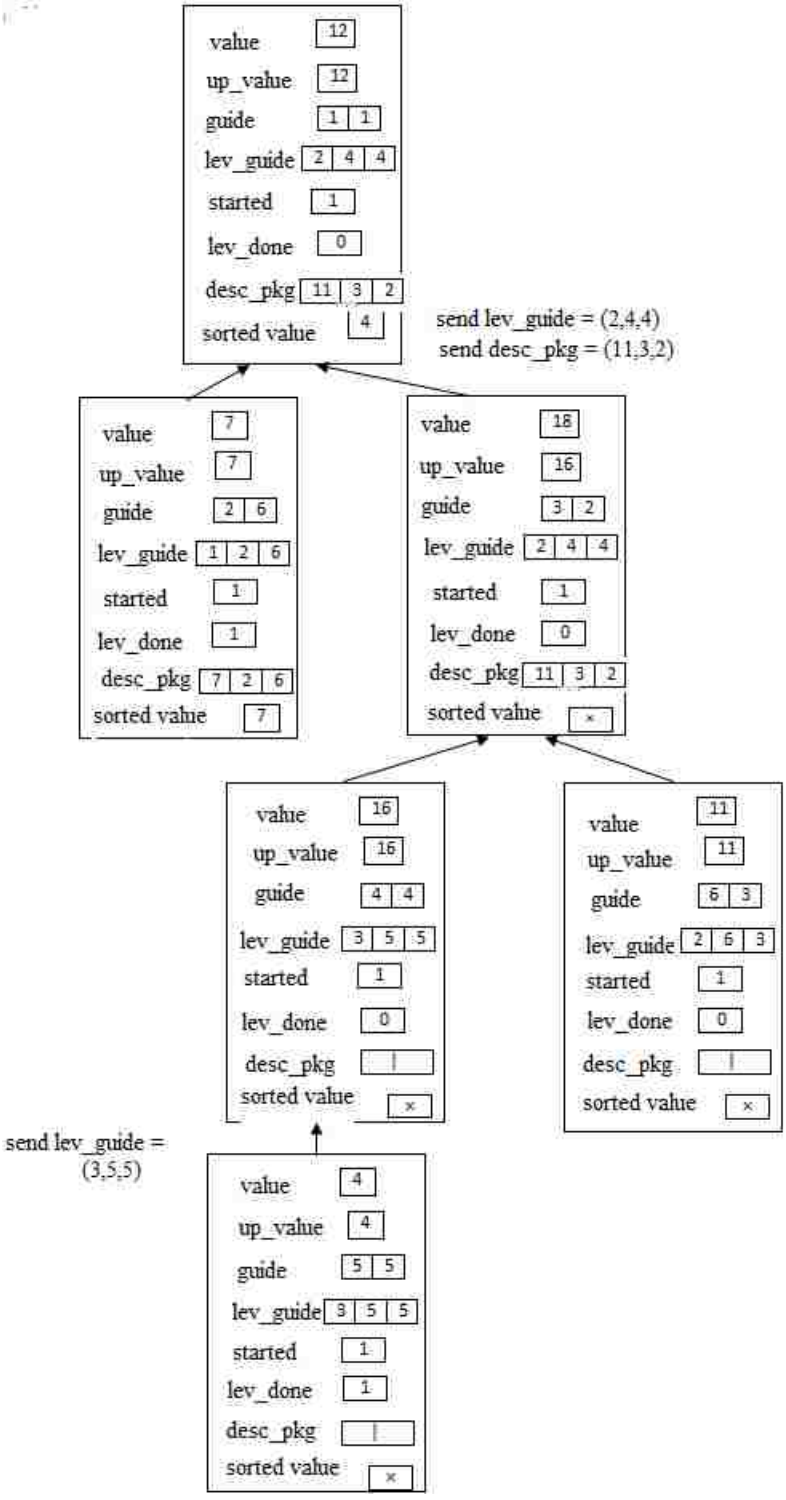
The Root creates desc_pkg value for the process $P_{2,6}$.



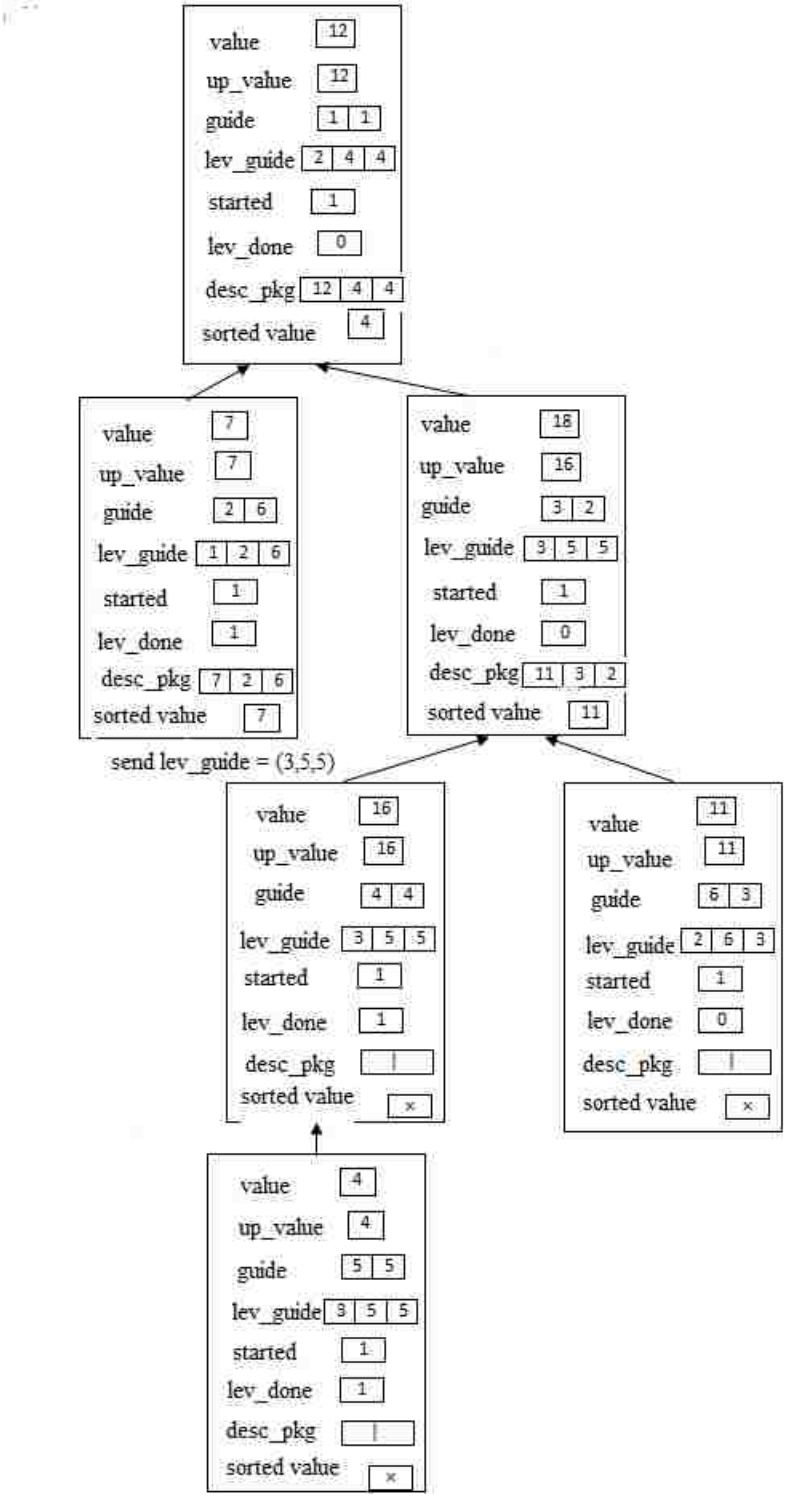
$P_{2,6}$ recieves its desc_pkg value. $P_{3,2}$ sends its up_value and lev_guide values to the Root.



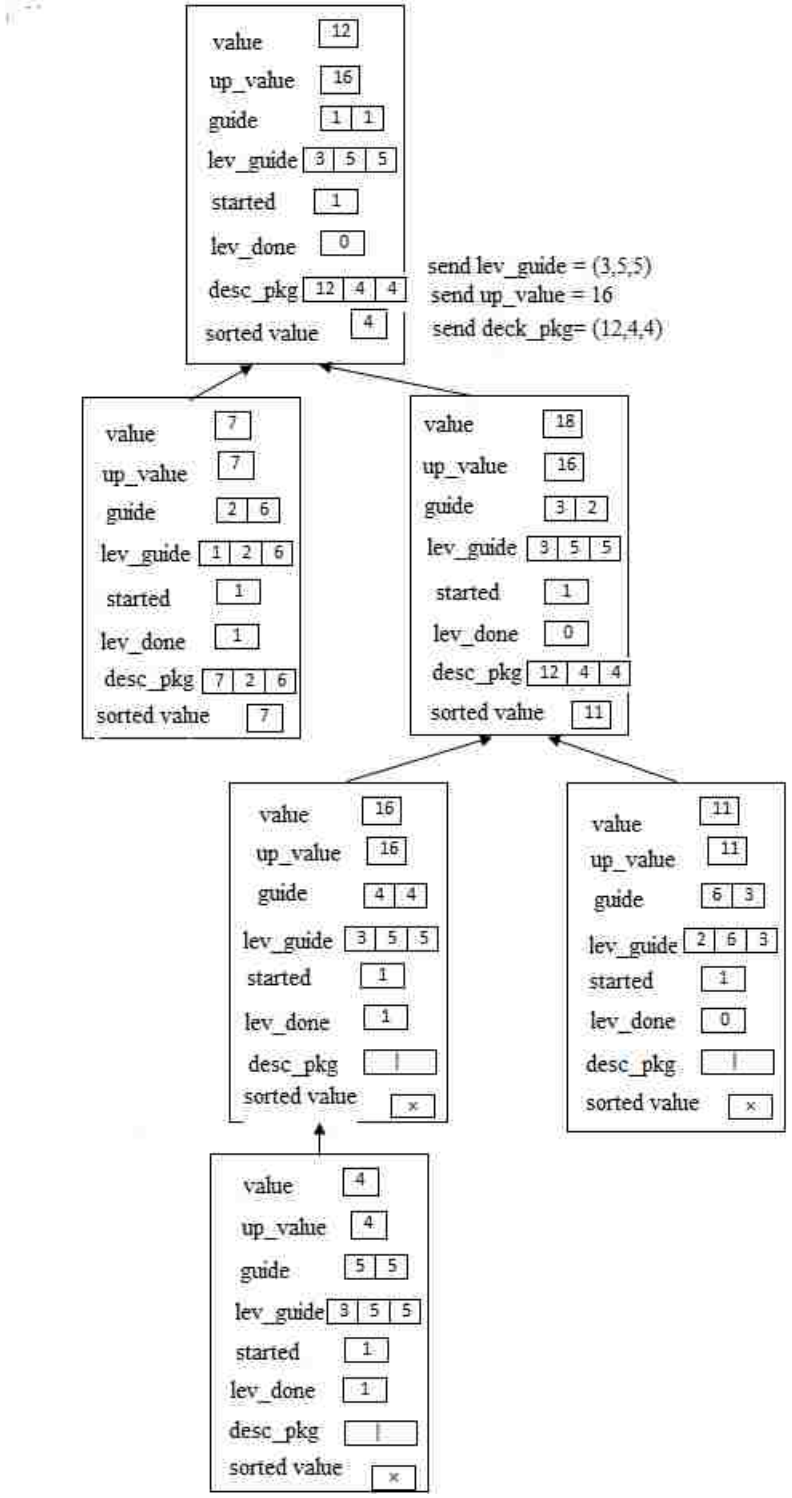
$P_{2,6}$ assigns the value to sorted value from its desc_pkg. Root creates a desc_pkg value for the process $P_{3,2}$.



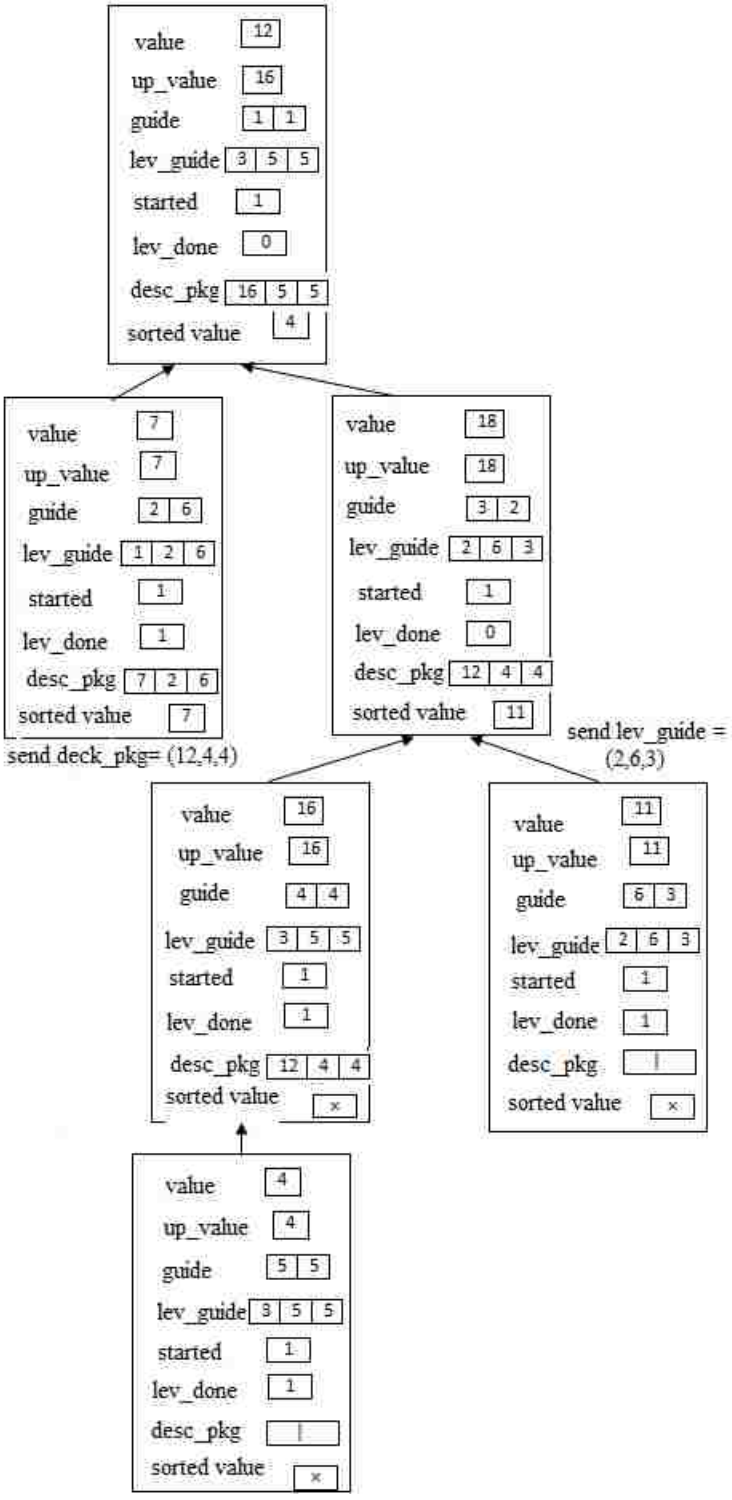
$P_{3,2}$ recieves its desc_pkg and sends the lev_guide value to the Root.



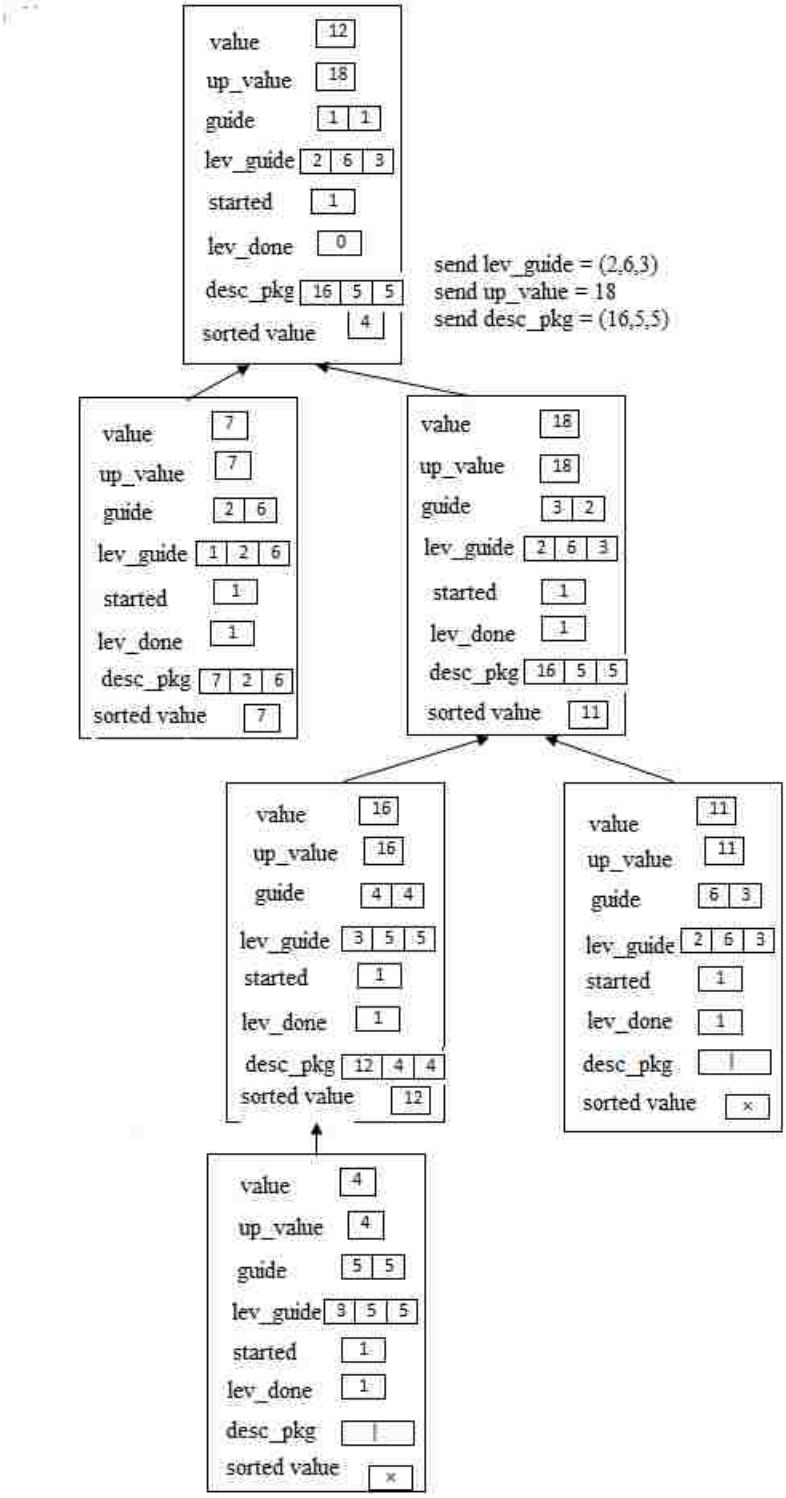
$P_{3,2}$ assigns its sorted value from its decs_pkg. Root creates a desc_pkg value for the process $P_{4,4}$



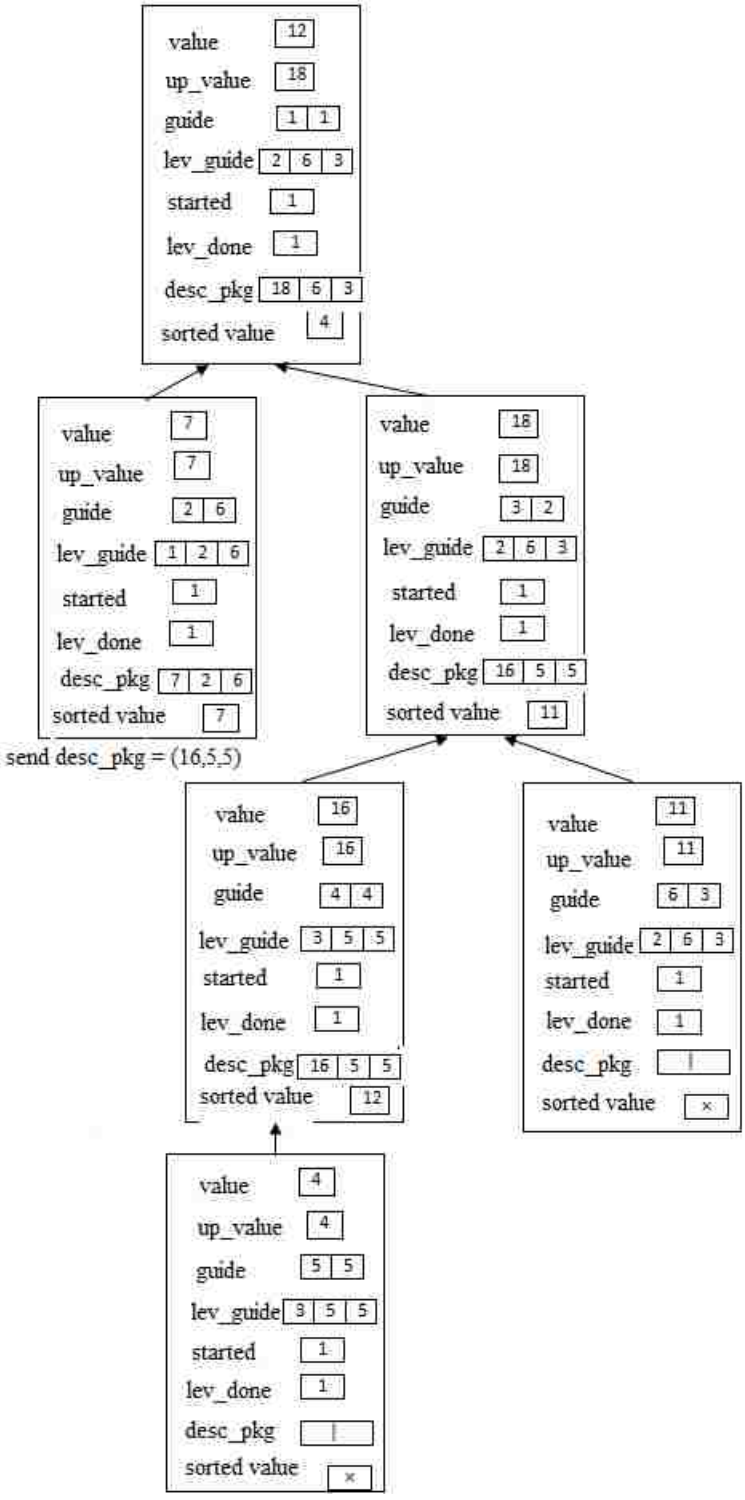
Root receives lev_guide value whose home process is $P_{5,5}$ and up_value from $P_{3,2}$. Root sends desc_pkg value whose home process is $P_{4,4}$.



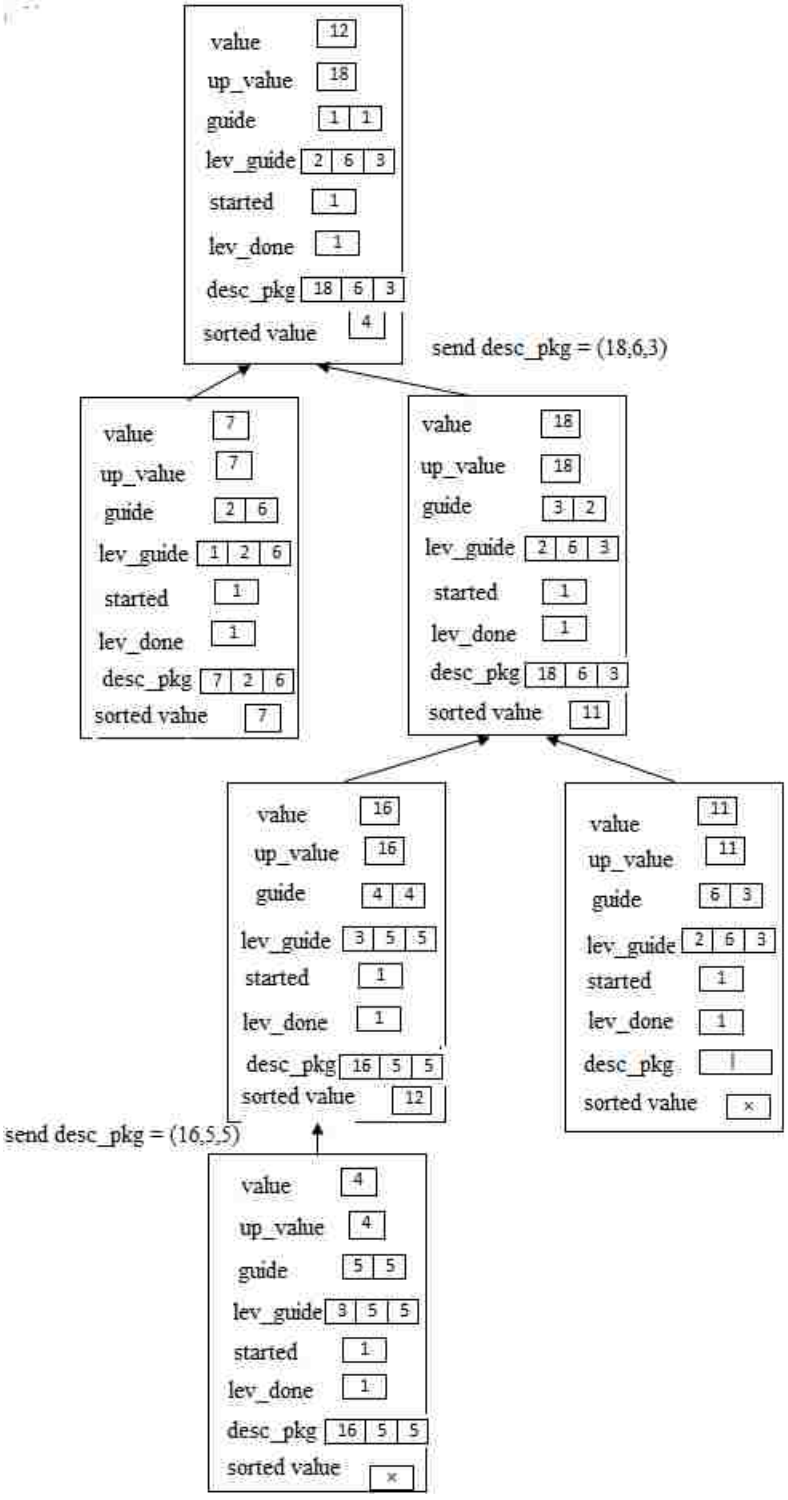
Root creates desc_pkg $P_{5,5}$.



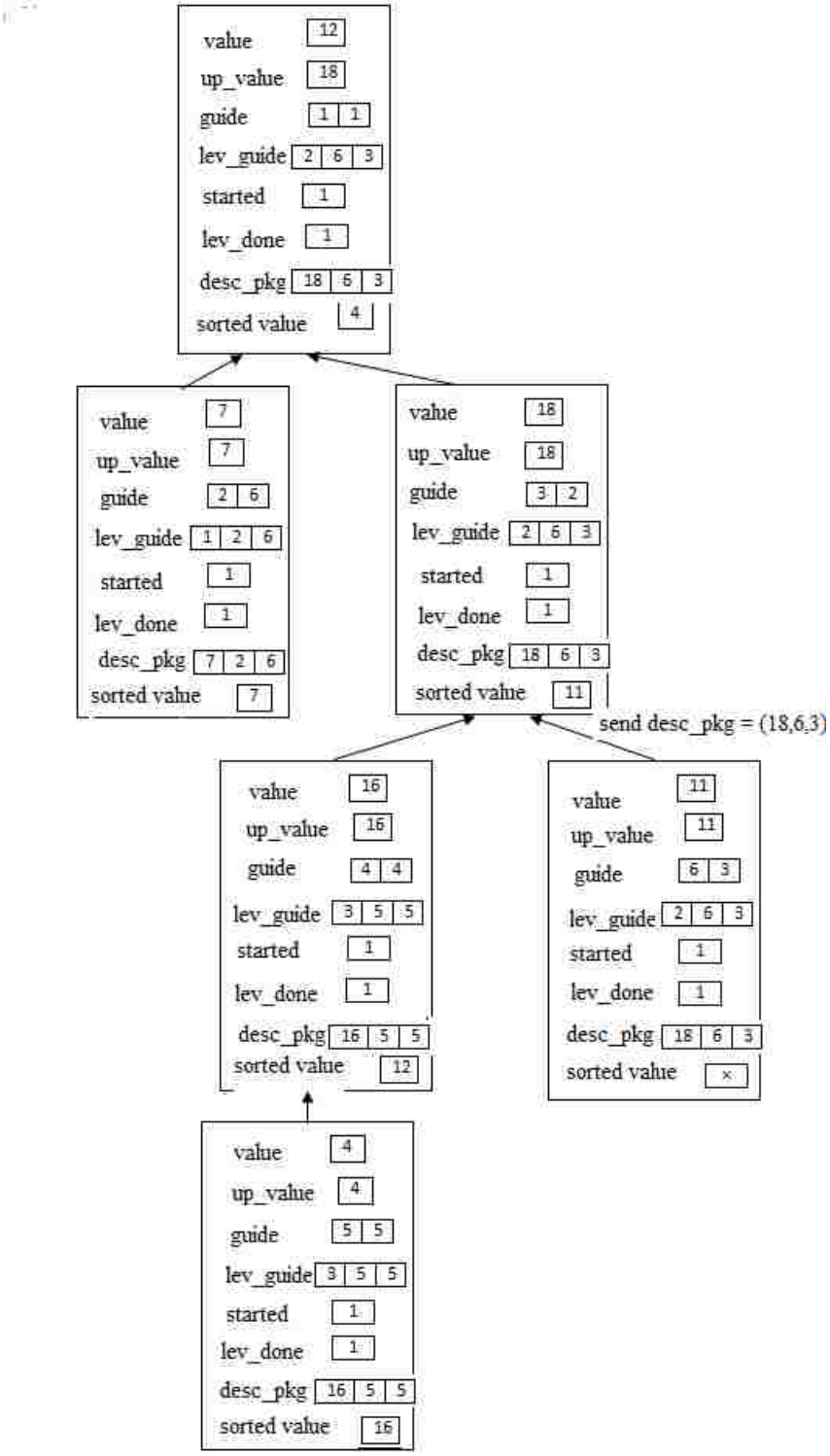
P3,2 sends its up_value and desc_pkg to the Root. Root send desc_pkg whose home process is $P_{5,5}$



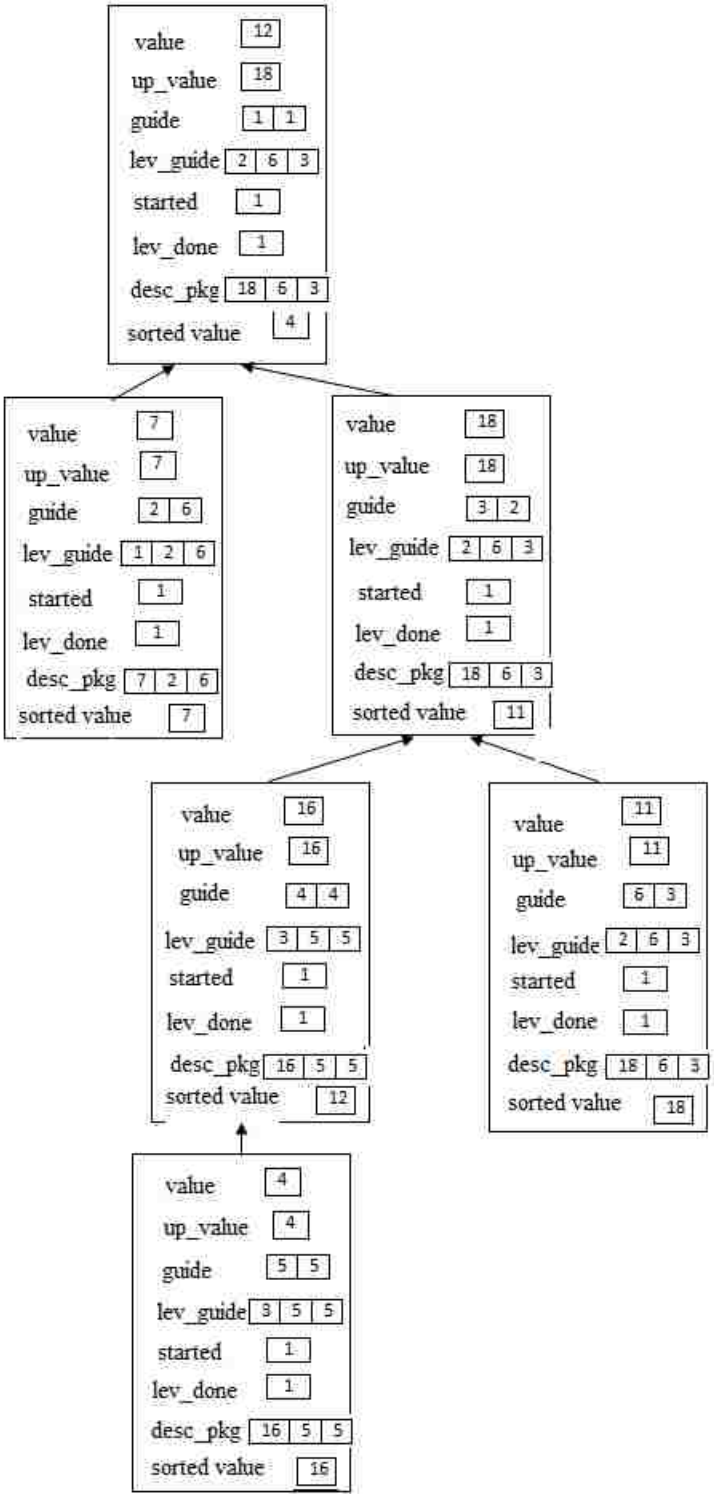
Root generates desc_pkg value for the process $P_{6,3}$. $P_{3,2}$ sends its desc_pkg to its child $P_{4,4}$



Root sends desc_pkg value for the process $P_{6,3}$. $P_{5,5}$ receives its desc_pkg.



$P_{5,5}$ assigns value to its sorted value from its desc_pkg value. $P_{6,3}$ receives its desc_pkg value.



Finally all the process in the tree has its sorted values.

BIBILOGRAPHY

[1] Dr. Ajoy K. Datta and Dr. Lawrence L. Larmore , Some Problems on a Rooted Tree Network.

[2] Rafael B Avila and Caciano Mochodo, Message Passing over Shared Memory.

[3] Angela C.Sodan, Message Passing over Shared Memory programming models.

VITA

Graduate College
University of Nevada, Las Vegas

Sabaresh N Maddula

Home Address :

1555 E Rochelle Ave, #Apt 252,
Las Vegas, Nevada - 89119

Degrees:

Bachelor of Science in Computer Science, 2006

Jawaharlal Nehru University, Hyderabad

Thesis Title: Message Passing Algorithms for different problems Sum, Mean, Guide and Sorting in a rooted tree network.

Thesis Examination Committee:

Chair Person, Dr. Ajoy K. Datta, Ph.D.

Committee Member, Dr. Lawrence L. Larmore,Ph.D.

Committee Member, Dr. Ju-Yeon Jo, Ph.D.

Committee Member, Dr. Emma Regentova, Ph.D.