95.590X
Advanced Topics in Data Structures

A Graphical Java Implementation of PQ-Trees

Submitted by: Jon Harris (267 309)
Submitted on: April 24, 2002
Submitted to: Pat Morin

Table of Contents

Table of Contents 2
Overview of PQ-Trees 3
Overview of the Reduction operation 4
Bubble 4
Reduce 5
Clear 6
Overview of Methods used for Reduction 7
Implementation Issues 8
Performance Analysis 10
Appendix A – Detailed Description of the Template Matching Process 11
Notes: 11
Template L1: 12
Template P1: 12
Template P2: 13
Template P3: 13
Template P4: 16
Template P5: 18
Template P6: 20
Template Q1: 22
Template Q2: 23
Template Q3: 26
Appendix B – Profiler Performance Analysis Results 30
Appendix C – Notes on Compiling and Running the Java Implementation 34
References 34

Overview of PQ-Trees

The PQ-Tree data structure was first introduced by Booth and Lueker in 1976 in their paper entitled *Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms*. [Booth et al. 1976].

In their paper, Booth and Lueker describe PQ-Trees as a data structure for representing the permutations of a set U in which various subsets of U are constrained to occur consecutively. They present efficient algorithms for manipulating PQ-Trees, the most desirable property of these algorithms being that they require a number of steps which are linear in the size of their input (the subsets).

The fundamental elements of PQ-Trees are P-Nodes and Q-Nodes. P-Nodes allow their children to be permuted in any order, while Q-Nodes allow only a reversal of the (fixed) ordering of their children. These operations are referred to by the authors as *equivalence transformations*, and two PQ-Trees are deemed to be *equivalent* if one can be transformed into the other by applying zero or more equivalence transformations. The initial or *universal PQ-Tree* is created by adding all of the elements of U as P-Node (leaf) children of a root P-Node.

The authors describe the following conditions that PQ-Trees must satisfy in order to be deemed proper:

- Every element of U occurs precisely once in the PQ-Tree, always as a leaf Node.
- Every P-Node must have at least two children. (A P-Node with one child would serve no purpose and would lead to chaining).
- Every Q-Node must have at least three children. (A Q-Node with two children is essentially a P-Node).

The only operation that can be performed on PQ-Trees is the reduction operation. The reduction operation takes as input a subset S of U in which the elements are constrained to appear together. The reduction operation proceeds by performing template matching and replacement on all of the elements of S (the *pertinent* elements of U) and their parents until the lowest common ancestor of all these elements is encountered. This lowest common ancestor is referred to by the authors as the *root of the pertinent sub tree*. Pertinent PQ-Nodes are labelled as ‘full’ or ‘partial’ while non-pertinent PQ-Nodes are labelled as ‘empty’. P-Nodes and Q-Nodes are labelled as full when all of their children are pertinent, and Q-Nodes are labelled partial when some of their children are pertinent.

After the reduction operation is performed, the number of permissible permutations of the elements in the PQ-Tree is reduced, and the resulting (proper) PQ-Tree represents a class of permissible permutations of U. The resulting PQ-Tree will be equivalent to all other PQ-Trees within this class. If no valid reduction of a PQ-Tree is possible, the result will be the *null PQ-Tree* which consists of one P-Node as the root with zero children.

Overview of the Reduction operation

The reduction operation is the heart and soul of any PQ-Tree implementation. It consists of two phases, the bubbling up pass (referred to as *Bubble*) and the reduction pass (referred to as *Reduce*). The key to implementing the reduction operation efficiently is to avoid maintaining valid parent pointers for any empty nodes which are siblings of pertinent nodes. Otherwise, maintaining valid parent pointers for empty nodes could easily require processing the entire PQ-Tree during each and every reduction.

Bubble

Bubble is the first pass of the reduction operation, it is responsible for ensuring that every pertinent PQ-Node has a valid parent pointer at the start of the second pass. It also provides each pertinent PQ-Node with a count of the number of pertinent child nodes that it has (*pertinent child count*).

At the beginning of the bubbling up pass, each of the pertinent leaf PQ-Nodes is enqueued in a Queue data structure. A count of the number of blocked nodes is initialized to zero, and a count of the number of blocks of blocked nodes (*block count*) is also initialized to zero.

During the bubbling up pass, PQ-Nodes are repeatedly dequeued and processed until the following conditions are met:

- No more PQ-Nodes remain in the Queue.
- The root of the PQ-Tree is encountered.
- No more PQ-Nodes with valid parent pointers could be processed.

Initially, each PQ-Node that is dequeued is marked as 'blocked'. If it has a valid parent pointer, or can obtain a valid parent pointer from one of its immediate siblings, it is marked as 'unblocked'.

<u>If the PQ-Node is marked as 'blocked':</u>

Decrement the block count by the number of immediate siblings of the PQ-Node which are marked as 'blocked' and then increment it by one.

Increment the number of blocked nodes by one.

<u>If the PQ-Node is marked as 'unblocked':</u>

Find the maximal consecutive set of blocked siblings of the current PQ-Node (containing itself).
If this set has a size greater than zero, provide the current PQ-Node's parent pointer to each of the PQ-Nodes in the set and mark them as 'unblocked'. The

pertinent child count of the parent is incremented by one for each of these children. The number of blocked nodes is also decremented by one for each of these children, and then incremented by one.
Decrement the block count by the number of immediate siblings of the current PQ-Node which are marked as 'blocked'.

If the PQ-Node is not the root of the PQ-Tree, increment the pertinent child count of its parent once for itself, and enqueue the parent in the Queue (provided that the parent has not already been processed).

After the bubbling up pass is completed, a check is made to verify that the block count is not greater than one. A block count of zero means that no nodes were left blocked. A block count of one means that there is a consecutive block of blocked nodes which are children of the root of the pertinent sub tree. Since the root of the pertinent sub tree is not actually processed during the second pass, a pseudo-node can be assigned as the parent of these blocked nodes. (The reader is referred to the illustrations of Template Q3 of Appendix A for an example of the use of a pseudo-node.) A block count greater than one means that no reduction will be possible.

Reduce

Reduce is the second pass of the reduction operation, it is responsible for actually making the modifications to the PQ-Tree through the use of template matching and replacement. It also determines the number of pertinent leaves which are descendants of each node (*pertinent leaf count*)

At the beginning of the bubbling up pass, each of the pertinent leaf PQ-Nodes is enqueued in a Queue data structure and its pertinent leaf count is set to one.

During reduction pass, PQ-Nodes are repeatedly dequeued and processed until no more PQ-Nodes remain in the Queue, or a PQ-Node fails to match a template. If no template is matched for a PQ-Node, no reduction is possible.

As each PQ-Node is dequeued, its pertinent leaf count is checked against the total number of pertinent leaves. This allows the reduction pass to determine whether or not the current PQ-Node is the root of the pertinent sub tree.

<u>If the current PQ-Node IS NOT the root of the pertinent sub tree:</u>

Add the current Node's pertinent leaf count to the pertinent leaf count its parent.
Decrement the pertinent child count of its parent by one.
If the parent has a pertinent child count of zero (all descendants have been processed), enqueue it in the Queue.
Attempt to match (and replace) the current PQ-Node against the following templates:

- Template L1
- Template P1
- Template P3
- Template P5
- Template Q1
- Template Q2

<u>If the current PQ-Node IS the root of the pertinent sub tree:</u>

Attempt to match (and replace) the current PQ-Node against the following templates:

- Template L1
- Template P1
- Template P2
- Template P4
- Template P6
- Template Q1
- Template Q2
- Template Q3

(The reader is referred to Appendix A for a more detailed description of the Template Matching Process.)

Clear

Clear is the third pass of the reduction operation, it is used to reset all of the values of the pertinent PQ-Nodes back to their initial state from before the first two passes. This is implemented in a bottom-up fashion from the pertinent leaves up to the root of the pertinent sub tree. This pass could be eliminated by resetting the values of pertinent PQ-Nodes after they are processed during the reduction pass, but this would have prevented the visualization of the reduction process from being possible in the GUI of the implementation.

Overview of Methods used for Reduction

During the development of the implementation of PQ-Trees, it became apparent that several sequences of operations needed to be performed frequently during the reduction pass. The following is a summary of the more complicated methods that were implemented in the PQNode class to perform these sequences of operations.

absorbPartialChild(PQNode partialChild)
: Links the sibling pointers of endmost children of the partial child with the siblings of the partial child, and updates the full child list of the current node accordingly. Linkage is done in such a way that the children being linked to each other are either both pertinent, or both empty. The partial child is deleted.

addChild(PQNode newChild)
: Adds a new child node to this node, updating the full and partial child lists if necessary. If the current node is a Q-Node, the new child is added to the appropriate end of the list of siblings for the children, according to its label.

becomeChild(PQNode child)
: The current node assumes the identity of the child node, provided that only one child exists. All children of the child node except empty interior Q-Node children are updated to point to the current node as their parent. The child is deleted.

boolean checkFullAreAdjacent()
: Iterates the list of full children of the current node, and determines whether or not they occur as a consecutive sequence in the child list of a Q-Node.

mergePartialChildren(PQNode partialChild1, PQNode partialChild2)
: Links the sibling pointers of the pertinent nodes at the end of each of the partial children's child lists. The endmost children of the first partial child are updated to be the other two endmost children of the partial children. The full child lists of both partial children are merged into one. The second partial child is deleted.

removeChild(PQNode oldChild)
: Removes the old child node from this node, updating the full and partial child lists if necessary. The sibling pointers of the (both) sibling(s) of the old node are updated to maintain linkage.

replaceChild(PQNode oldChild, PQNode newChild)
: Essentially performs removeChild(oldChild) followed by addChild(newChild) except that the new child assumes the position of the old child in the list of siblings for the children of the current node.

(The reader is referred to the Java source code provided with the implementation for more details about the operations which PQ-Nodes can perform.)

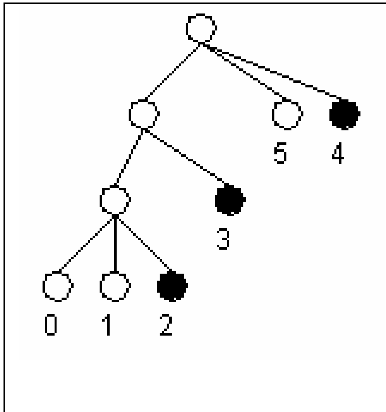
Implementation Issues

Implementing PQ-Trees efficiently gave rise to many issues. For the initial implementation, the main focus was getting the reduction operation working, without considering all of the optimizations that allow it to perform efficiently. Three Vectors (dynamic arrays) were maintained for each PQ-Node, one containing all the children, one containing all the full children, and one containing all of the partial children.

This Vector based implementation simplified the implementation of all of the methods mentioned in the previous section, yet it quickly became apparent that it was not efficient in terms of the number of child nodes that were processed during the reduction for each pertinent node. Removing or inserting elements from anywhere but the end of a Vector could easily require examining each of the nodes in a vector, and in the worst case, could require processing every node in a PQ-Tree during a reduction.

The solution was to store the various children of a PQ-Node in linked lists instead of Vectors. Each node now stores a reference to a single ‘access node’ for each of the linked lists, and traversal of the linked list can begin from this node. This added complexity to the methods from the previous section, but allowed them to run much more efficiently. Things were further complicated by the fact that the linked-list of child nodes for Q-Nodes needs to be reversible. This necessitated that children of Q-Nodes store a collection of up to two siblings instead of simply storing a left and right sibling. The SiblingVector class was implemented to handle this situation.

Another problem that was encountered was that during the bubbling up pass, the authors suggest that the parent of each pertinent node is enqueued if it is not already in the queue. This was discovered to be not quite correct however, since no node should ever be enqueued more than once during a single bubbling up pass. Enqueuing a node more than once would incorrectly increment the pertinent child count of its parent. The solution was to add a flag to each PQ-Node that determines whether or not is has already been processed during the bubbling up pass. The next figure illustrates this situation.

<table>
<tr>
<td>
</td>
<td>In this example, the initial queue would contain the nodes {3,4,2} (in any order).

If node 3 were first in the queue it would be dequeued and its parent enqueued. the queue would contain the nodes {4, 2, 3.p}

Nodes 4, 2, and 3.p (parent of 3) would then be dequeued, and as a result the queue would contain the nodes {4.p, 2.p}

When 2.p is processed, it would be incorrect to re-enqueue 3.p (its parent), since it has already been processed.</td>
</tr>
</table>

The linked list child structure for PQ-Nodes allows sets of children from different PQ-Nodes to be joined without processing all of the nodes in the list. In the situation where the children of one Q-Node are reparented to a different Q-Node, the parent references of the empty children of the original Q-Node are not updated. This brings to light an interesting situation; we may want to get rid of the original Q-Node once all of its children have been reparented, but it cannot be completely destroyed since it is likely still referenced by some empty children. The solution to this situation is to flag the original Q-Node as deleted, without destroying it. The Java garbage collector is left with the responsibility of destroying this Q-Node when it is no longer referenced. After considerable thought, it was determined that there was no alternative to this approach that would not require processing all of the empty Q-Nodes that were previously children of the Q-Node. The undesirable side-effect of this solution is that extra memory will be consumed to store the deleted nodes, however in the worst possible case this will only result in an extra linear factor of the total number of nodes in the PQ-Tree. (The reader is referred to the illustrations of Templates P6, Q2 and Q3 in Appendix A for examples of situations where nodes are deleted but still referenced.)

The fact that children of Q-Nodes may have invalid parent references (i.e. to deleted nodes) required a slight modification to the bubbling up pass of the reduction operation. The bubbling up pass requires that we be able to retrieve the siblings of a PQ-Node which are blocked. The authors describe the blocked siblings of a node as being those which are actually marked as ‘blocked’. This is not entirely valid, since a sibling whose parent is deleted should be considered as a blocked node even if it has not been marked as ‘blocked’.

During the reduction pass, the re-labelling of ‘empty’ nodes as either ‘full’ or ‘partial’ required some special attention since the full or partial child lists of the parent whose child was re-labelled required updating. This was achieved through the use of the replaceChild method on the parent. When a node is re-labelled, it is replaced by a newly labelled version of itself with respect to the parent. A special flag was used with the replaceChild method to indicate that only the full and partial child lists should be updated.

The most difficult part of implementing the reduction pass was the correct implementation of the absorbPartialChild, addChild, mergePartialChildren, removeChild and replaceChild methods described in the previous section. It proved to be quite a challenge to ensure that all of the child lists were properly updated for each of these operations, given the diversity of the possible arrangements of full, partial or empty children of the PQ-Nodes involved in the operation. In the case of absorbPartialChild, addChild and mergePartialChild, it was even more challenging to ensure that the child lists were linked in the correct order.

In conclusion, the last challenging aspect of this implementation of PQ-Trees was the implementation of the graphical user interface. While not a requirement for a PQ-Tree implementation, the graphical user interface proved itself invaluable for debugging the implementation. An added benefit was the ease with which I was able to extract the

illustrations provided in Appendix A. The graphical user interface can also be used as a powerful, visual tool for educating people about PQ-Trees in the future.

Performance Analysis

As mentioned previously, PQ-Trees are designed in such a way that reductions require only that nodes in the pertinent sub tree be examined. During the testing of this implementation, the reduction operation did take approximately the same amount of time irrespective of the number of nodes in the tree, given the same constraints. A slight performance penalty is incurred by this implementation with larger PQ-Trees since it takes time to retrieve the pertinent leaves from the set of all leaves. This slow-down can be avoided if references to the pertinent leaf nodes are available without requiring retrieval.

How long the reduction operations takes is a factor on the number of pertinent PQ-Nodes in the pertinent sub tree during a given reduction. Nodes that match simple templates such as L0, P1 or Q1 will clearly be faster to execute than those which require many operations like P2-P6, Q2 and Q3.

Implementing a data structure in Java is not the logical choice for a memory (or speed) efficient implementation. During testing of this implementation, an 'Out of Memory Error' occurred whenever the number of initial nodes for the PQ-Tree surpassed 500,000. This limit may of course be increased by allocating more memory to the virtual machine, but the default allocation of 70 Megs of memory is already quite high.

The bulk of the memory consumption was by the PQNode objects. Memory savings could be achieved by restricting leaf nodes to have primitive data types as their data instead of any Object, but this would be somewhat crippling to the applicability of the implementation. Additional memory savings could likely be achieved by reducing the size of the primitive instance variables (i.e. int to short etc…) but the gains from this would be marginal.

A limited amount of performance analysis was carried out on this implementation with aid of a profiler tool. I tried out two different profilers, both of which were crippled by evaluation licenses, and managed to gain some insight into the performance of the implementation, although the tools were not intuitive to use.
(The reader is referred to Appendix B for a summary of the profiler output.)

Appendix A – Detailed Description of the Template Matching Process

The following is a sketch of the 10 templates which are employed during the reduction process. Each template has a pattern that a node must match, and a replacement which is applied when a match occurs.

Notes:

When grouping of PQ-Nodes occurs, a new grouping PQ-Node is only created when there are more than one child to group, otherwise the new node is replaced by the single PQ-Node which was to be grouped.

Here is a summary of notation used in the following template illustrations:

- The current PQ-Node being matched is marked with a Blue Dot.
- In the replacement, the position of the root of the replacement is marked with a Blue Dot. If the current PQ-Node was used as a grouping node, it is marked with two Blue Dots.
- P-Nodes are drawn as a Circle.
- Q-Nodes are drawn as a Rectangle.
- Full PQ-Nodes are filled in Black.
- Partial PQ-Nodes are filled in Grey.
- Empty PQ-Nodes are filled in White.
- Pseudo-nodes are filled in Green.
- PQ-Nodes with valid parent pointers are drawn in Black.
- PQ-Nodes without valid parent pointers are drawn in Red.
- PQ-Nodes whose parent is a pseudo-node are drawn in Green.

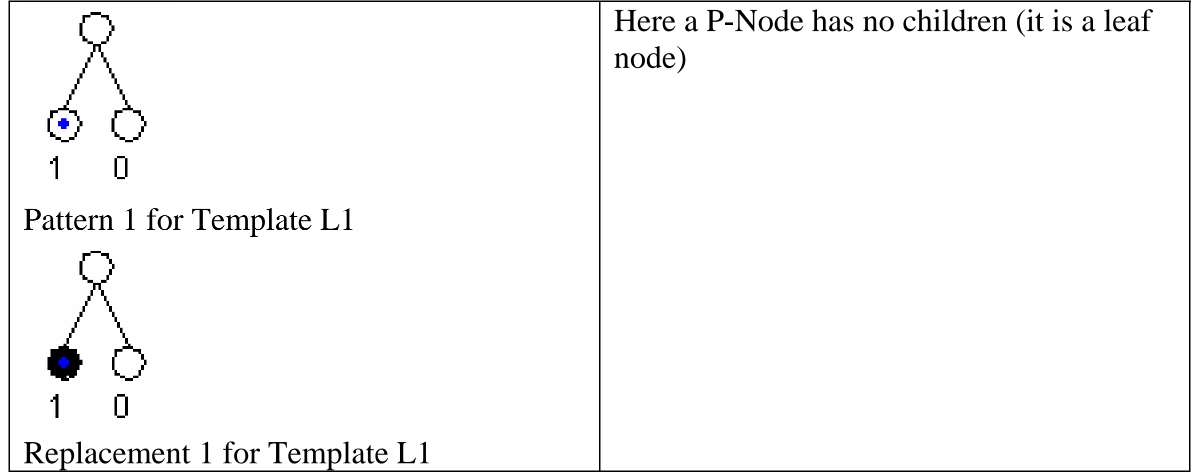
Template L1:

Pattern:

- A leaf node (P-Node) which is a part of the set S.

Replacement:

- Label the leaf node as 'full'.



1 0 Pattern 1 for Template L1 1 0 Replacement 1 for Template L1	Here a P-Node has no children (it is a leaf node)

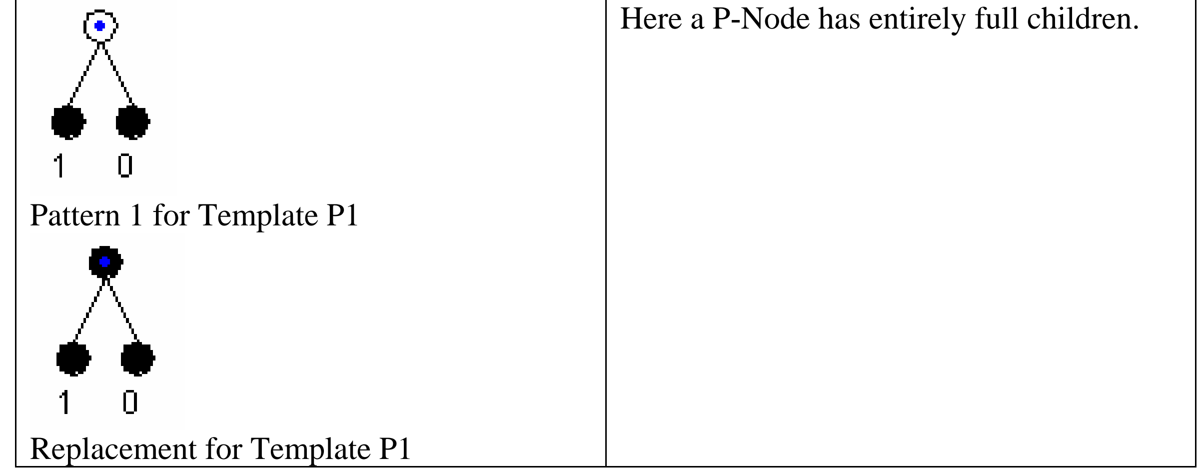
Template P1:

Pattern:

- A P-Node with more than one child node, all of which are labelled as 'full'.

Replacement:

- Label the current P-Node as 'full'.



1 0 Pattern 1 for Template P1 1 0 Replacement for Template P1	Here a P-Node has entirely full children.

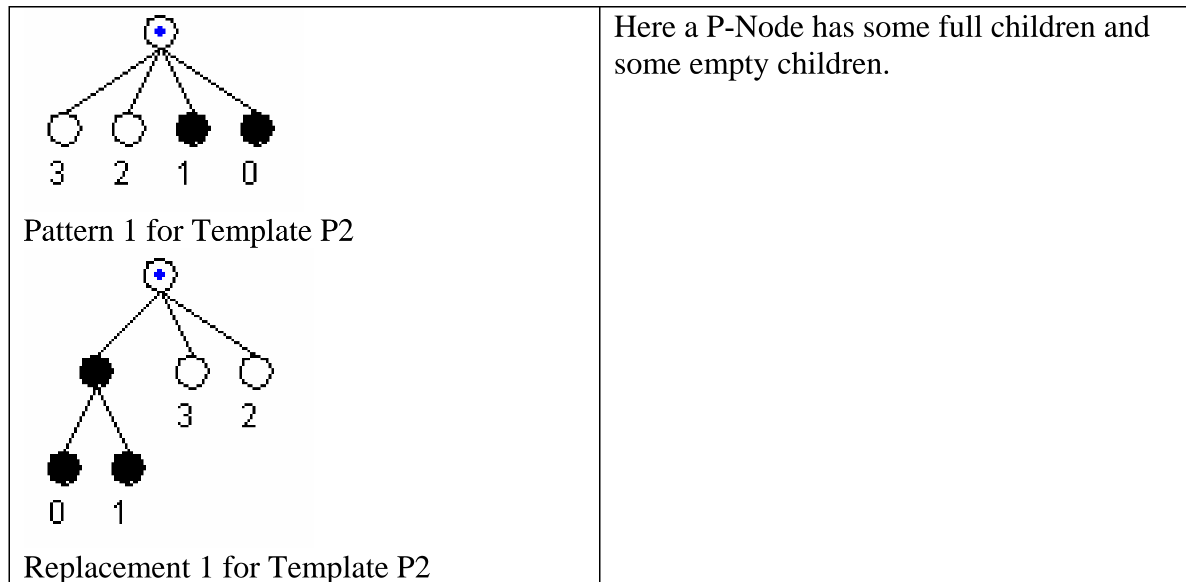
Template P2:

Pattern:

- A P-Node which is the root of the pertinent sub tree with more than one child node, some of which are labelled as ‘empty’ and at least one of which is labelled as ‘full’.

Replacement:

- Create a new P-Node labelled as ‘full’ and move all the full children to be children of the new P-Node. Add the new P-Node as a child of this P-Node.



Pattern 1 for Template P2 Replacement 1 for Template P2	Here a P-Node has some full children and some empty children.

Template P3:

Pattern:

- A P-Node which is not the root of the pertinent sub tree with more than one child node, some of which are labelled as ‘empty’ and at least one of which is labelled as ‘full’.

Replacement:

- Create a new Q-Node labelled as ‘partial’.
- Remove any children of the current P-Node which labelled ‘full’ and group them under a new P-Node which is labelled as ‘full’ and added as a child of the new Q-Node.
- If there is more than one child node labelled as ‘empty’, add the current P-Node as a child of the new Q-Node.
- If there is only one child labelled as ‘empty’, add it as a child of the new Q-Node and delete the current P-Node.

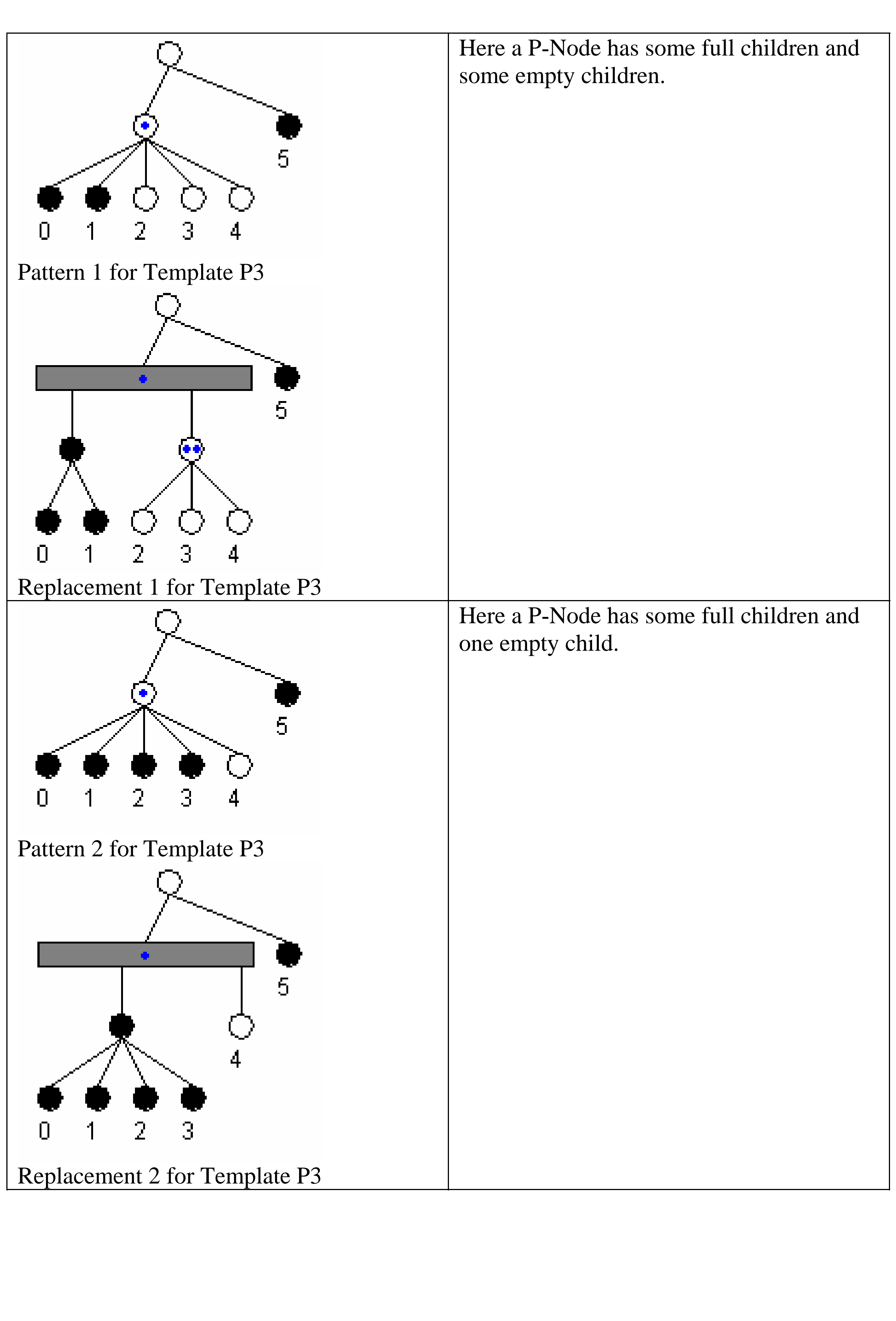
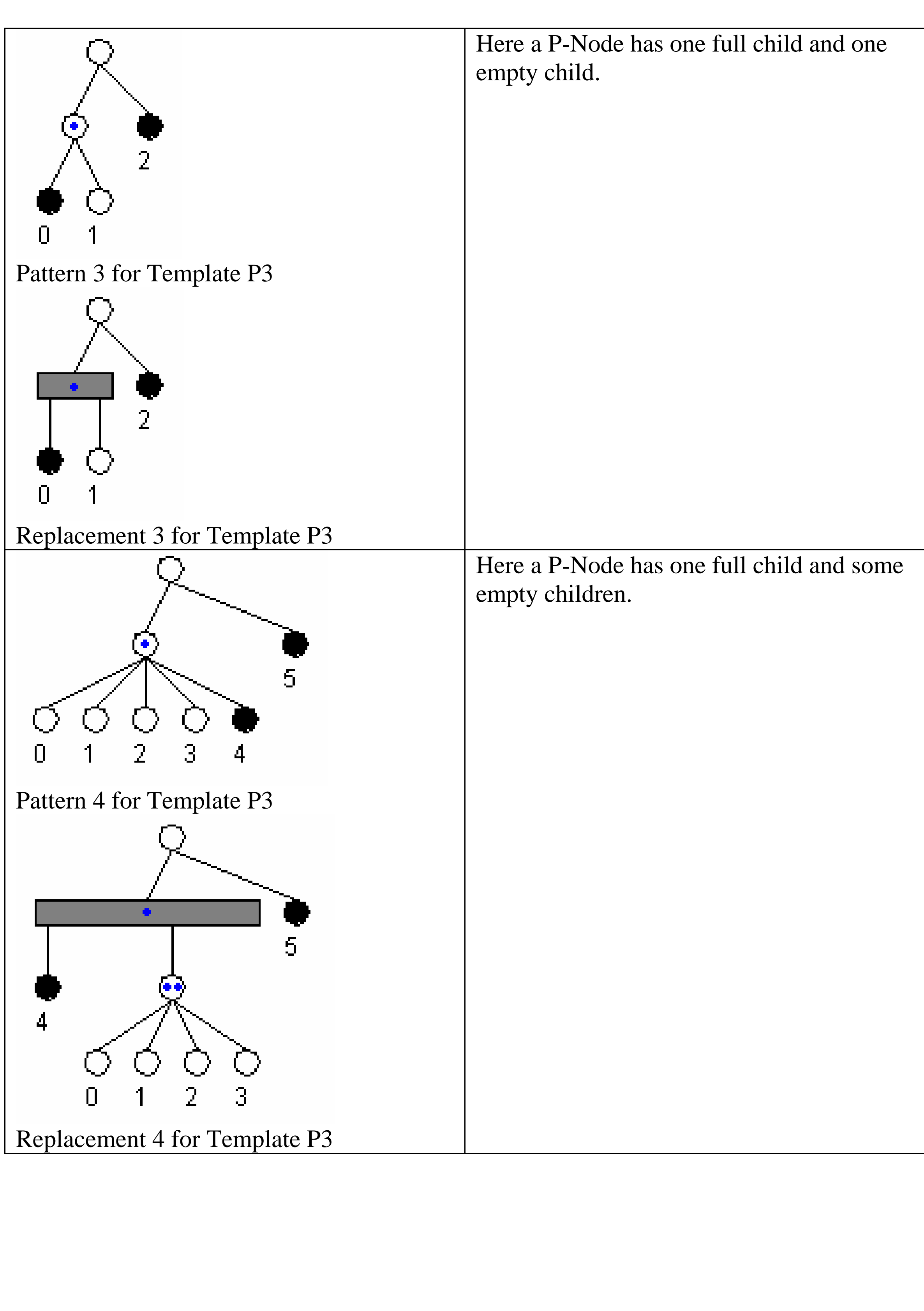


Figure	Description
Pattern 1 for Template P3 Replacement 1 for Template P3	Here a P-Node has some full children and some empty children.
Pattern 2 for Template P3 Replacement 2 for Template P3	Here a P-Node has some full children and one empty child.



Pattern 3 for Template P3 Replacement 3 for Template P3	Here a P-Node has one full child and one empty child.
Pattern 4 for Template P3 Replacement 4 for Template P3	Here a P-Node has one full child and some empty children.

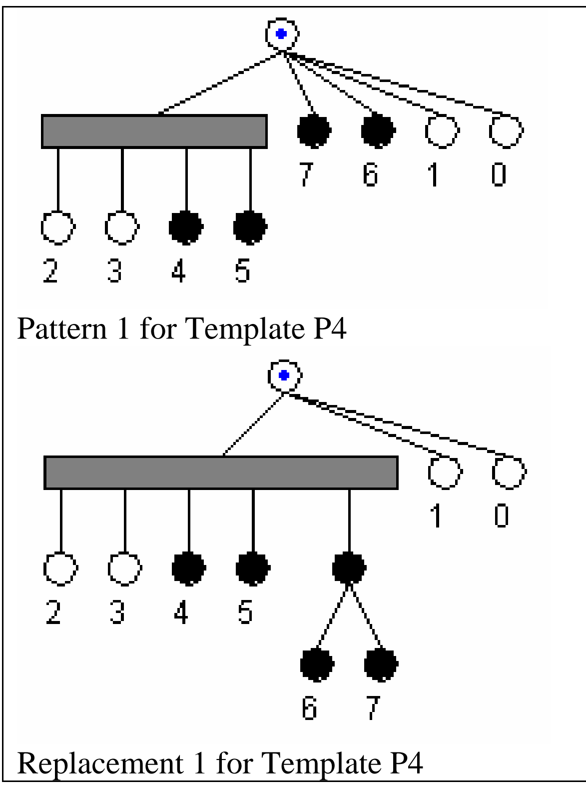
Template P4:

Pattern:

- A P-Node which is the root of the pertinent sub tree and has exactly one Q-Node child labelled as ‘partial’. The partial Q-Node child must have its full children adjacent to one end.

Replacement:

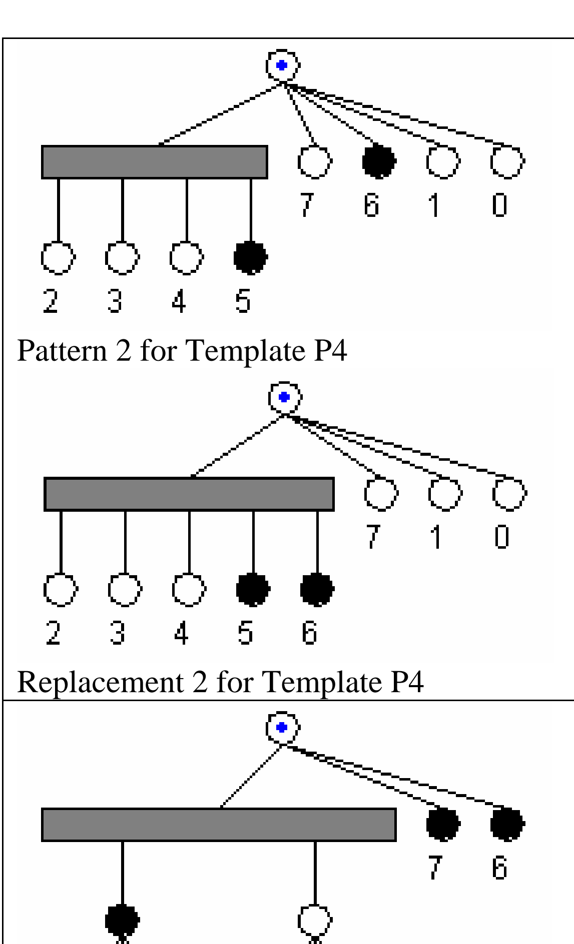
- Remove any children of the current P-Node which are labelled ‘full’ and group them under a new P-Node which is labelled as ‘full’ and added as a child of the partial Q-Node.



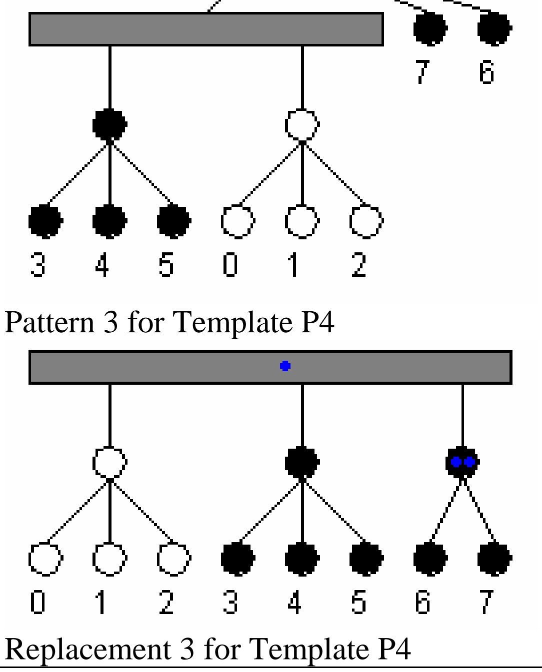
Pattern 1 for Template P4

Replacement 1 for Template P4

Here a P-Node has one partial child and some full children and some empty children.

<table>
<tr>
<td>


Pattern 2 for Template P4

Replacement 2 for Template P4
</td>
<td>Here a P-Node has one partial child and one full child and some empty children.</td>
</tr>
<tr>
<td>


Pattern 3 for Template P4

Replacement 3 for Template P4
</td>
<td>Here a P-Node has one partial child and some full children and no empty children.

Since the resulting P-Node would have only one child, it is replaced by the partial child.</td>
</tr>
</table>

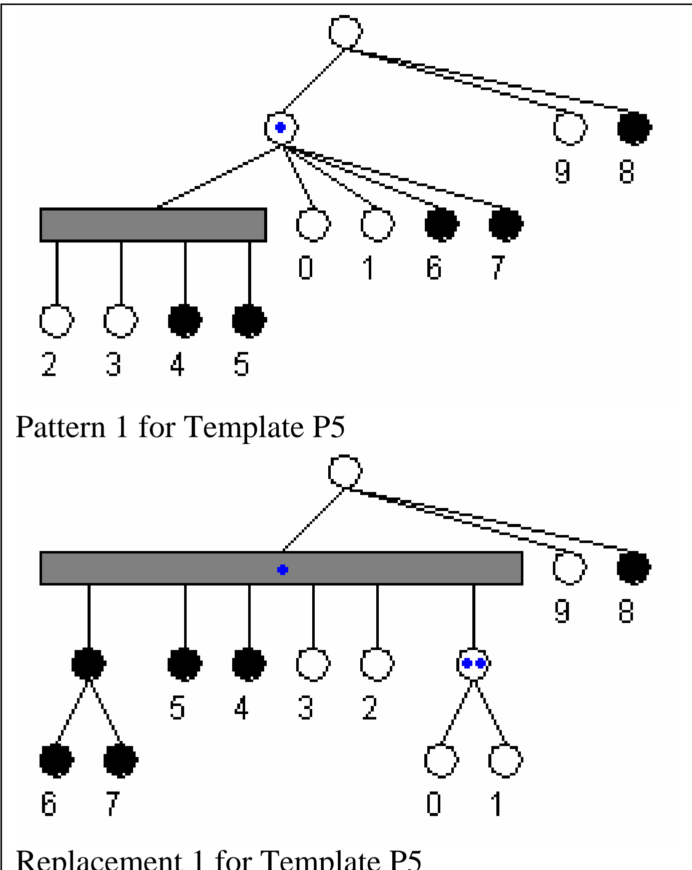
Template P5:

Pattern:

- A P-Node which is not the root of the pertinent sub tree and has exactly one Q-Node child labelled as ‘partial’. The partial Q-Node child must have its full children adjacent to one end.

Replacement:

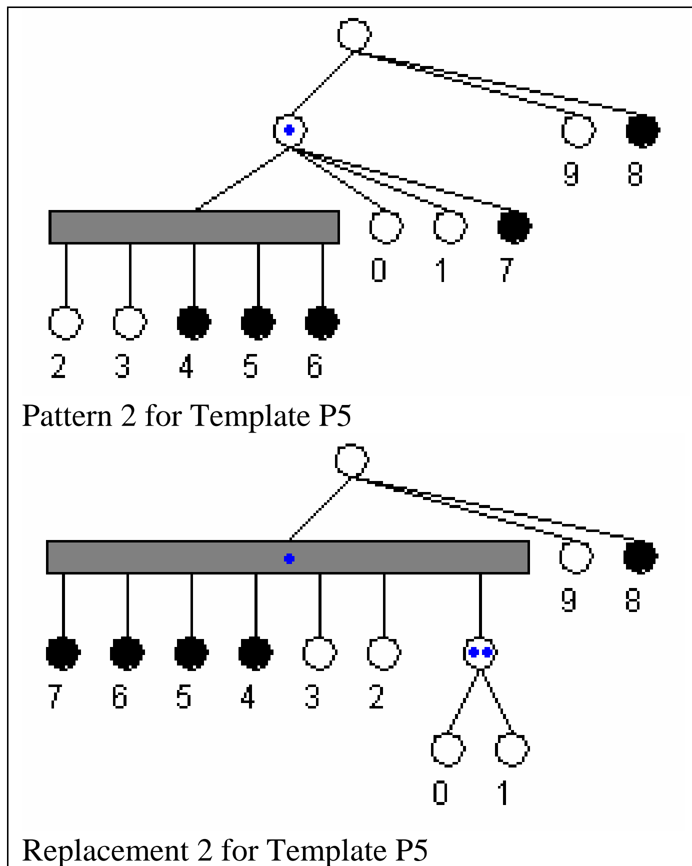
- Replace the current P-Node with the partial Q-Node child in the child list of the current P-Node’s parent.
- Remove any children of the current P-Node which are labelled ‘full’ and group them under a new P-Node which is labelled as ‘full’ and added as a child of the partial Q-Node.
- If there is more than one child node labelled as ‘empty’, add the current P-Node as a child of the partial Q-Node.
- If there is only one child labelled as ‘empty’, add it as a child of the partial Q-Node and delete the current P-Node.

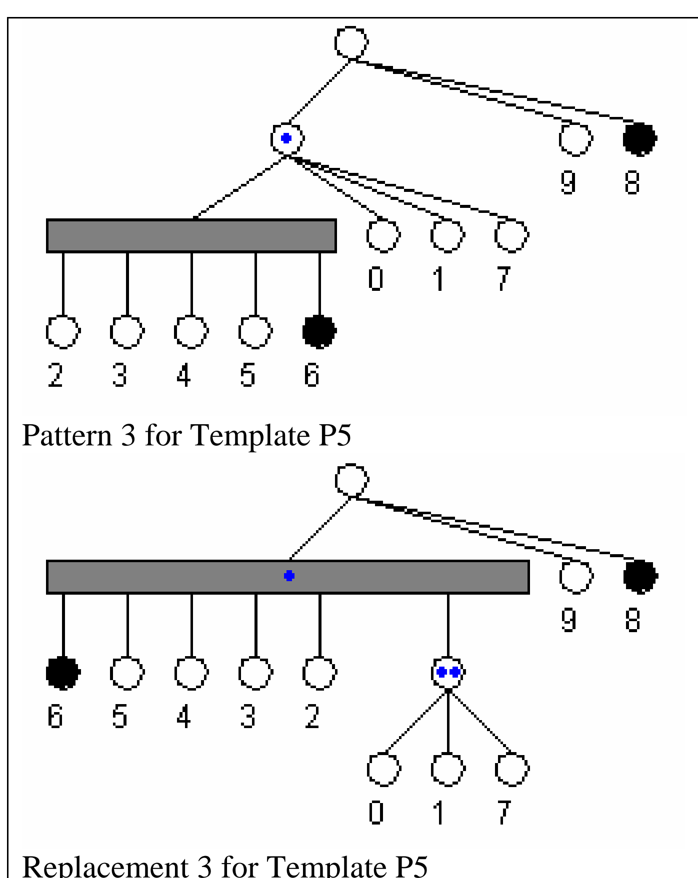
<table>
<tr>
<td>


Pattern 1 for Template P5

Replacement 1 for Template P5
</td>
<td>
Here a P-Node has one partial child and some empty children and some full children.

Since the resulting P-Node would have only one child, it is replaced by the partial child, and used for grouping the empty children under the partial child.
</td>
</tr>
</table>

 Pattern 2 for Template P5 Replacement 2 for Template P5	Here a P-Node has one partial child and some empty children and one full child. Since the resulting P-Node would have only one child, it is replaced by the partial child, and used for grouping the empty children under the partial child.

<table>
<tr>
<td>


Pattern 3 for Template P5
Replacement 3 for Template P5
</td>
<td>
Here a P-Node has one partial child and some empty children, but no full children.

Since the resulting P-Node would have only one child, it is replaced by the partial child, and used for grouping the empty children under the partial child.
</td>
</tr>
</table>

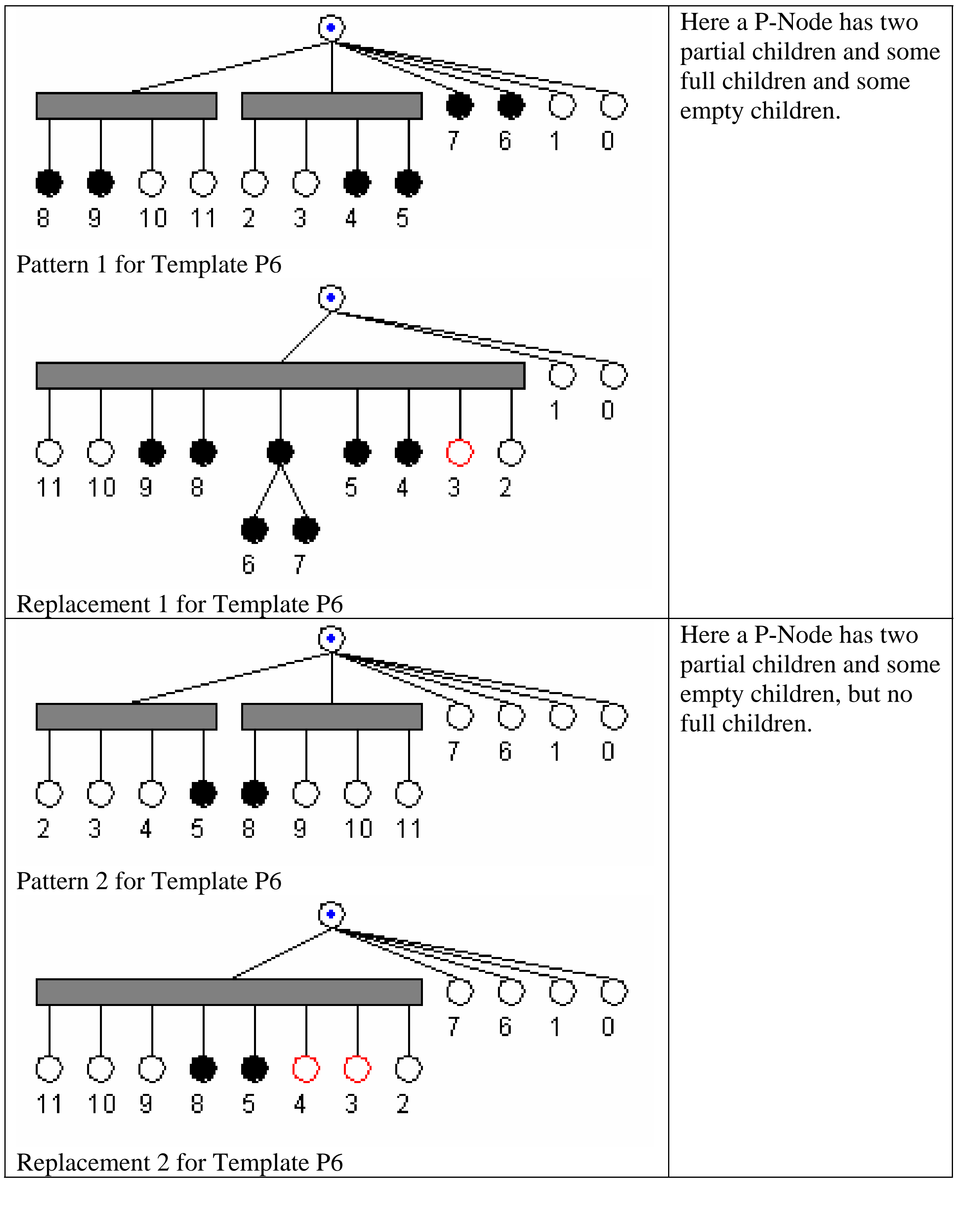
Template P6:

Pattern:

- A P-Node which is the root of the pertinent sub tree and has exactly two Q-Node children labelled as ‘partial’. Both partial Q-Node children must have their full children adjacent to one end.

Replacement:

- Remove any children of the current P-Node which are labelled ‘full’ and group them under a new P-Node which is labelled as ‘full’ and added as a child of the one of the partial Q-Node children.
- Merge the child lists of the two partial children, storing the result under on of the partial Q-Node children. The other partial Q-Node child is deleted.



Pattern 1 for Template P6

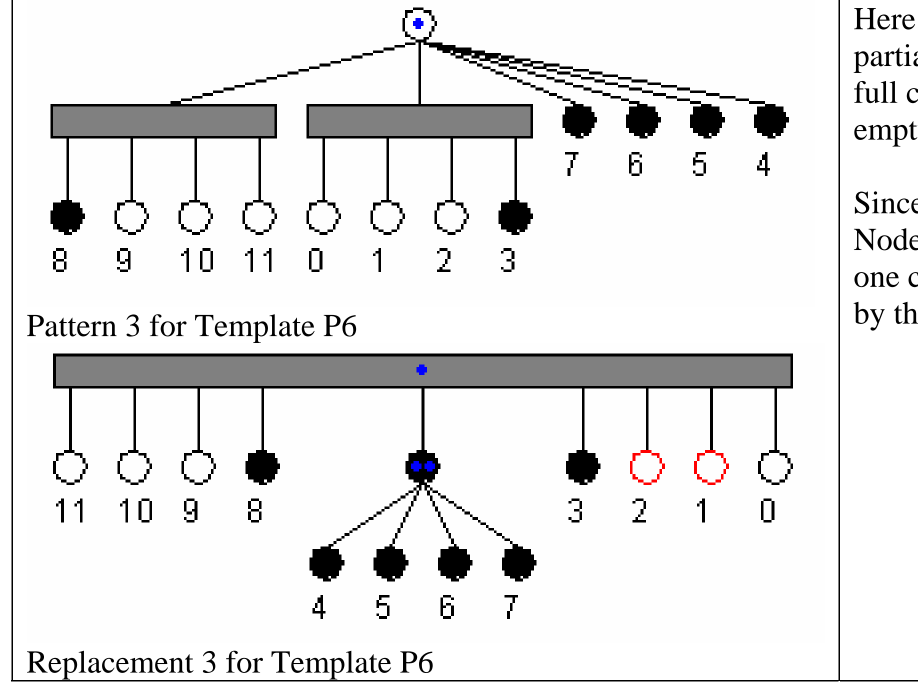
Replacement 1 for Template P6

Here a P-Node has two partial children and some full children and some empty children.

Pattern 2 for Template P6

Replacement 2 for Template P6

Here a P-Node has two partial children and some empty children, but no full children.



Pattern 3 for Template P6

Replacement 3 for Template P6

Here a P-Node has two partial children and some full children, but no empty children.

Since the resulting P-Node would have only one child, it is replaced by the partial child.

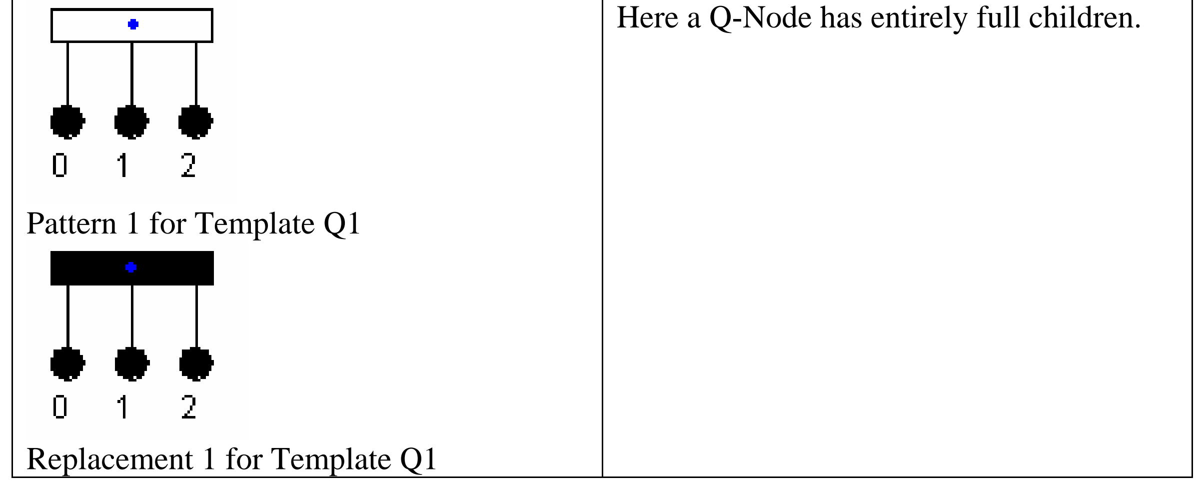
Template Q1:

Pattern:

- A Q-Node with more than one child node, all of which are labelled as ‘full’.

Replacement:

- Label the current Q-Node as ‘full’.



Pattern 1 for Template Q1

Replacement 1 for Template Q1

Here a Q-Node has entirely full children.

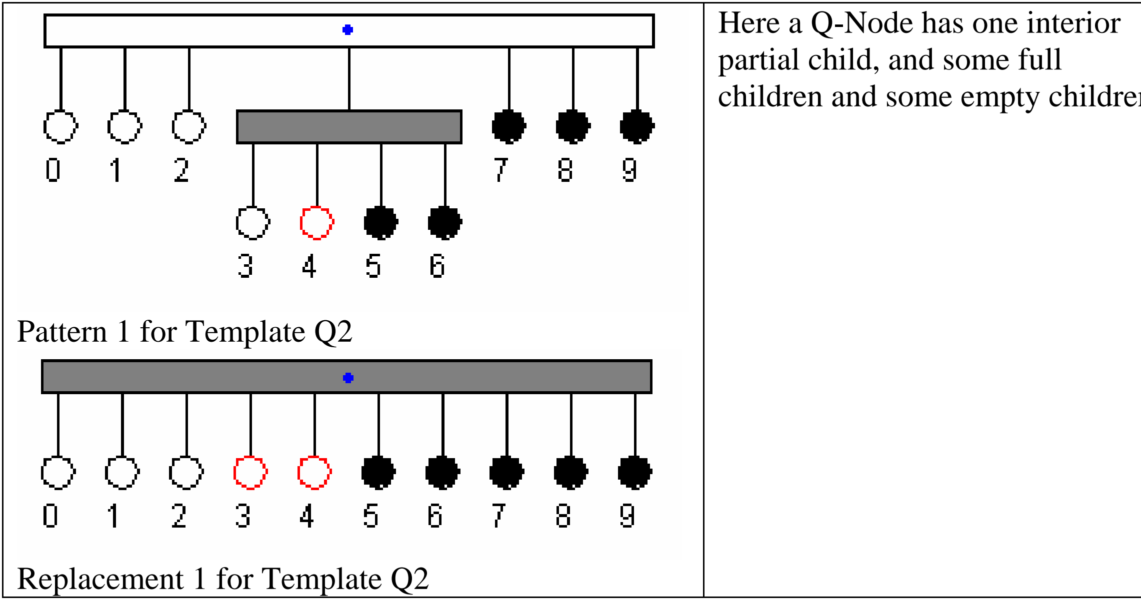
Template Q2:

Pattern:

- A Q-Node with zero or one Q-Node children labelled as 'partial'. The partial Q-Node child if it exists must have its full children adjacent to one end.

Replacement:

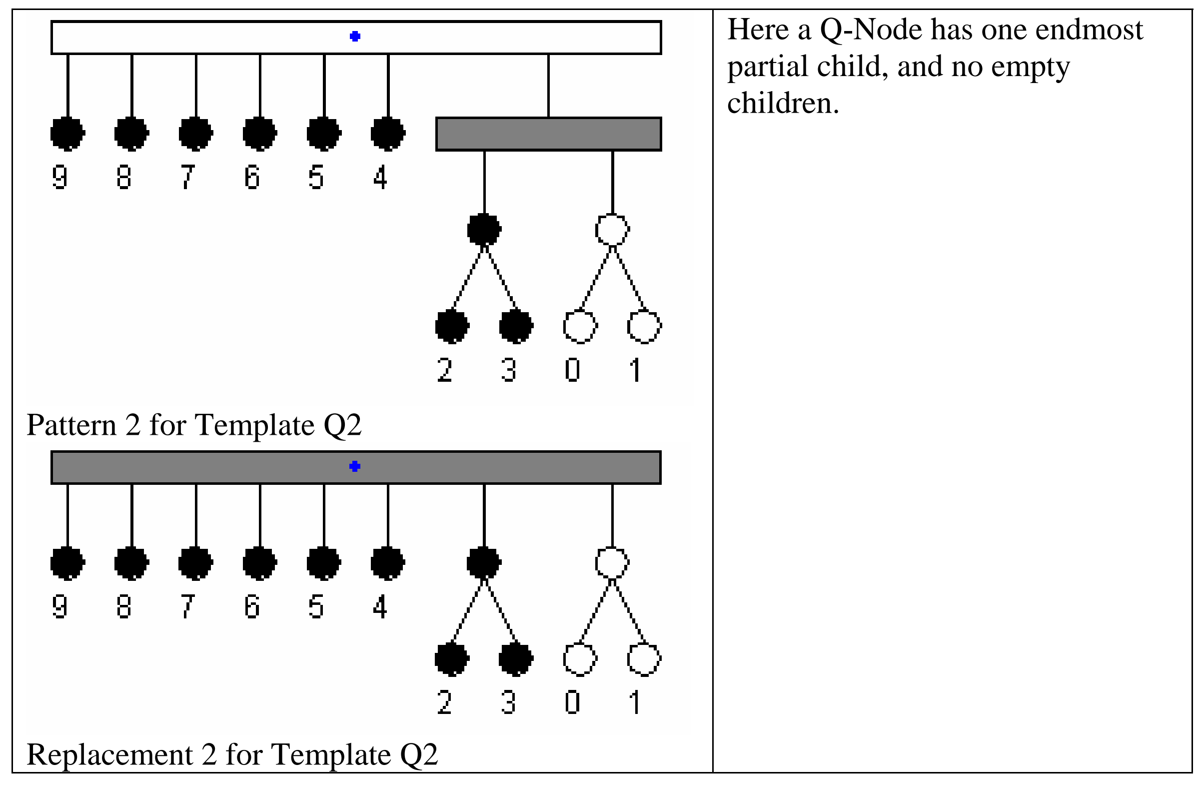
- Absorb the children of the partial Q-Node child (if it exists) into the child list of the current Q-Node.
- Label the current Q-Node child as 'partial'.



Pattern 1 for Template Q2

Replacement 1 for Template Q2

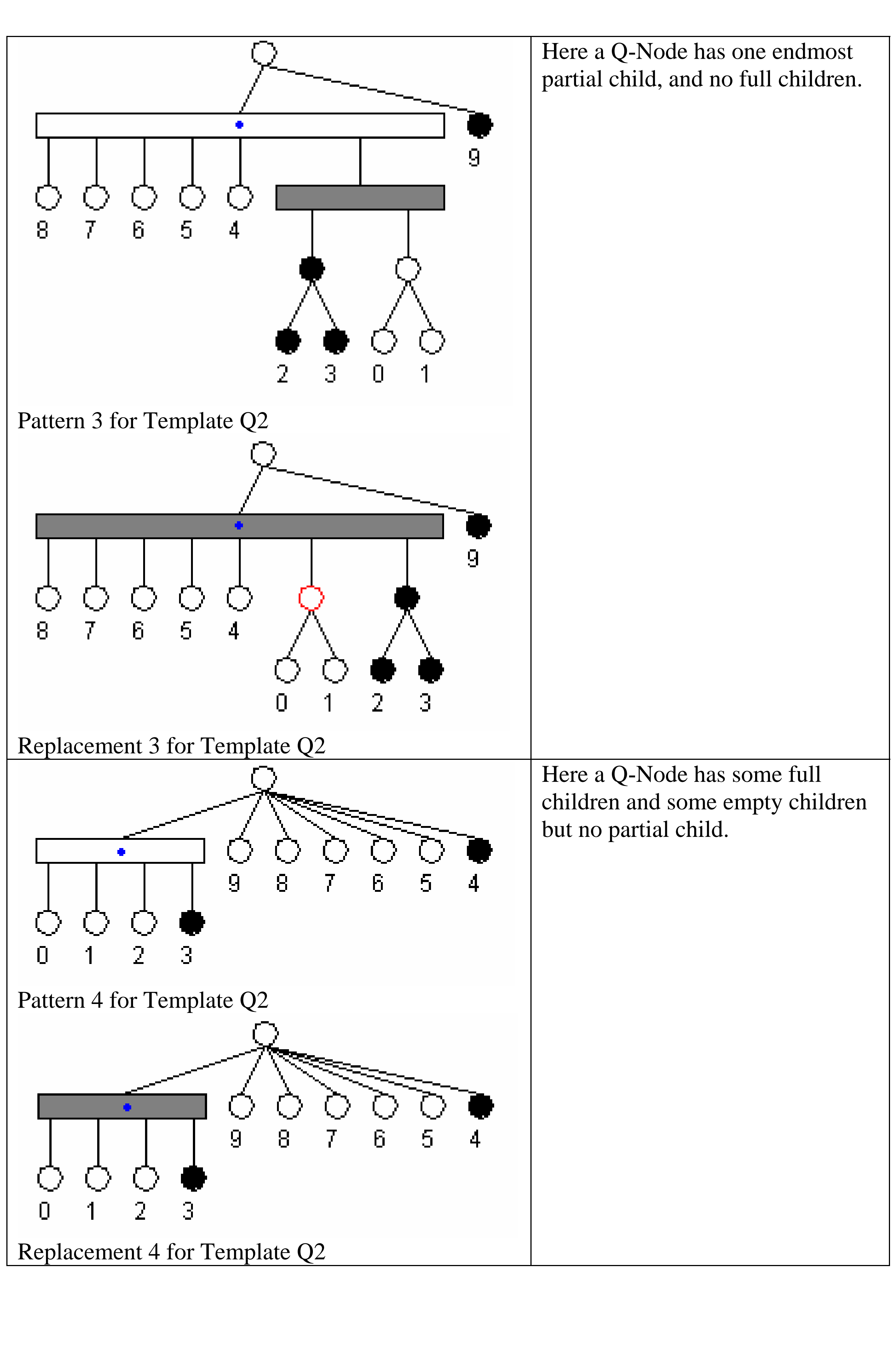
Here a Q-Node has one interior partial child, and some full children and some empty children.



Pattern 2 for Template Q2

Replacement 2 for Template Q2

Here a Q-Node has one endmost partial child, and no empty children.



Pattern 3 for Template Q2

Replacement 3 for Template Q2

Here a Q-Node has one endmost partial child, and no full children.

Pattern 4 for Template Q2

Replacement 4 for Template Q2

Here a Q-Node has some full children and some empty children but no partial child.

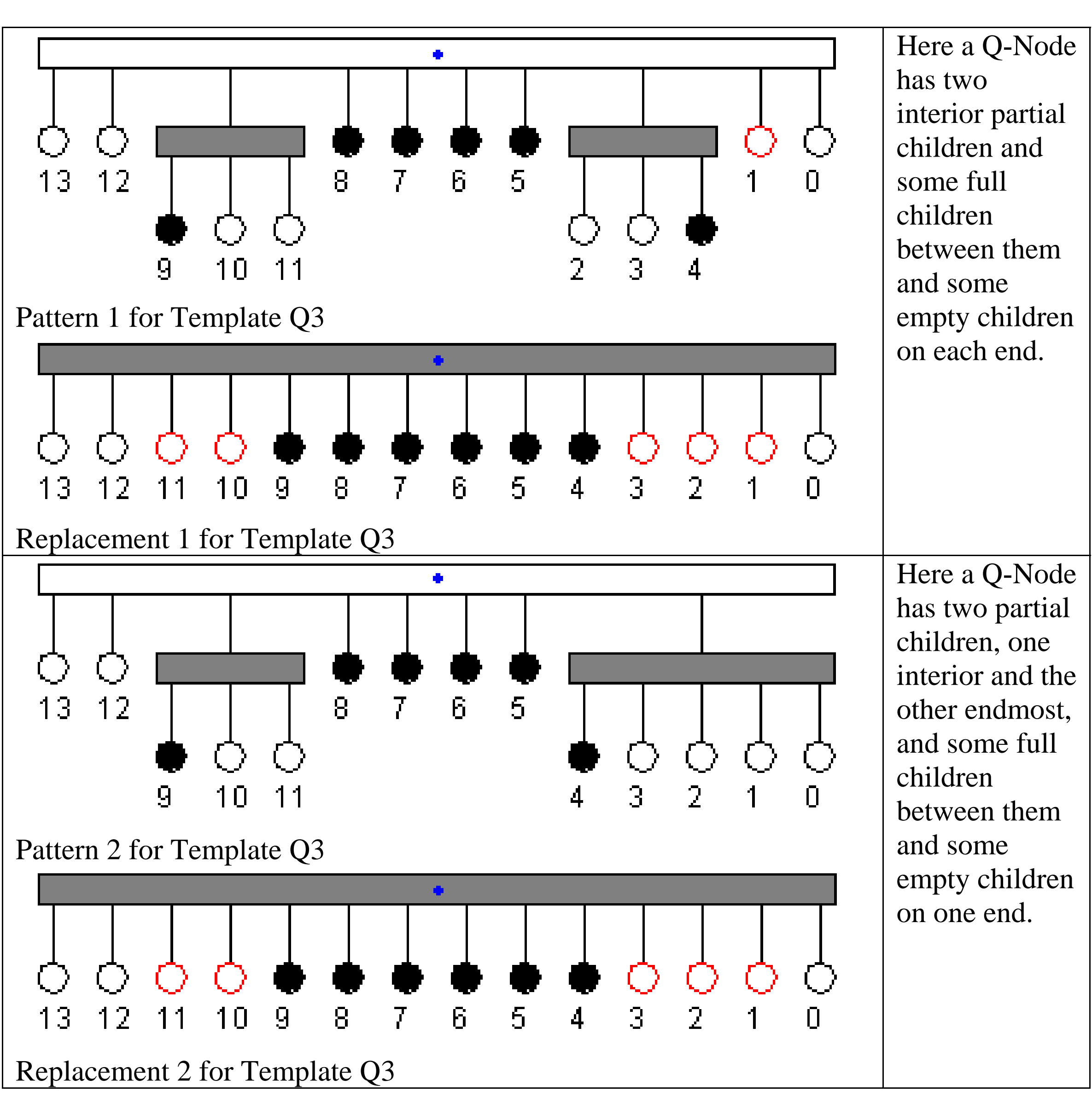
Template Q3:

Pattern:

- A Q-Node with zero, one or two Q-Node children labelled as ‘partial’ that is a pseudo-node or that has only partial or empty children at its ends. The partial Q-Node children if they exist must have its full children adjacent to one end.

Replacement:

- Absorb the children of the partial Q-Node child (if they exist) into the child list of the current Q-Node.
- Label the current Q-Node child as ‘partial’.
- If the current Q-Node is a pseudo-node, delete current Q-Node.



Pattern 1 for Template Q3

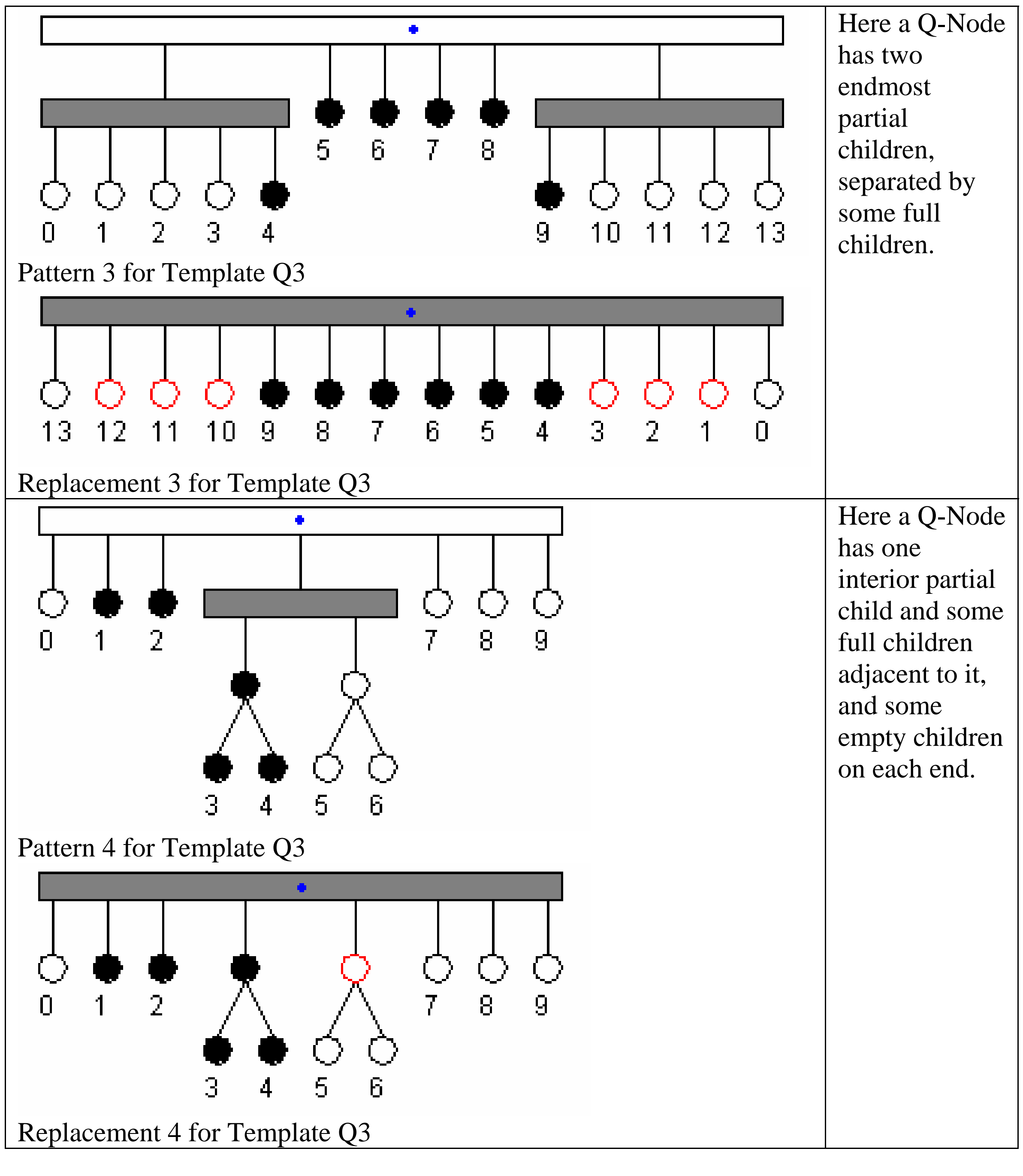
Replacement 1 for Template Q3

Here a Q-Node has two interior partial children and some full children between them and some empty children on each end.

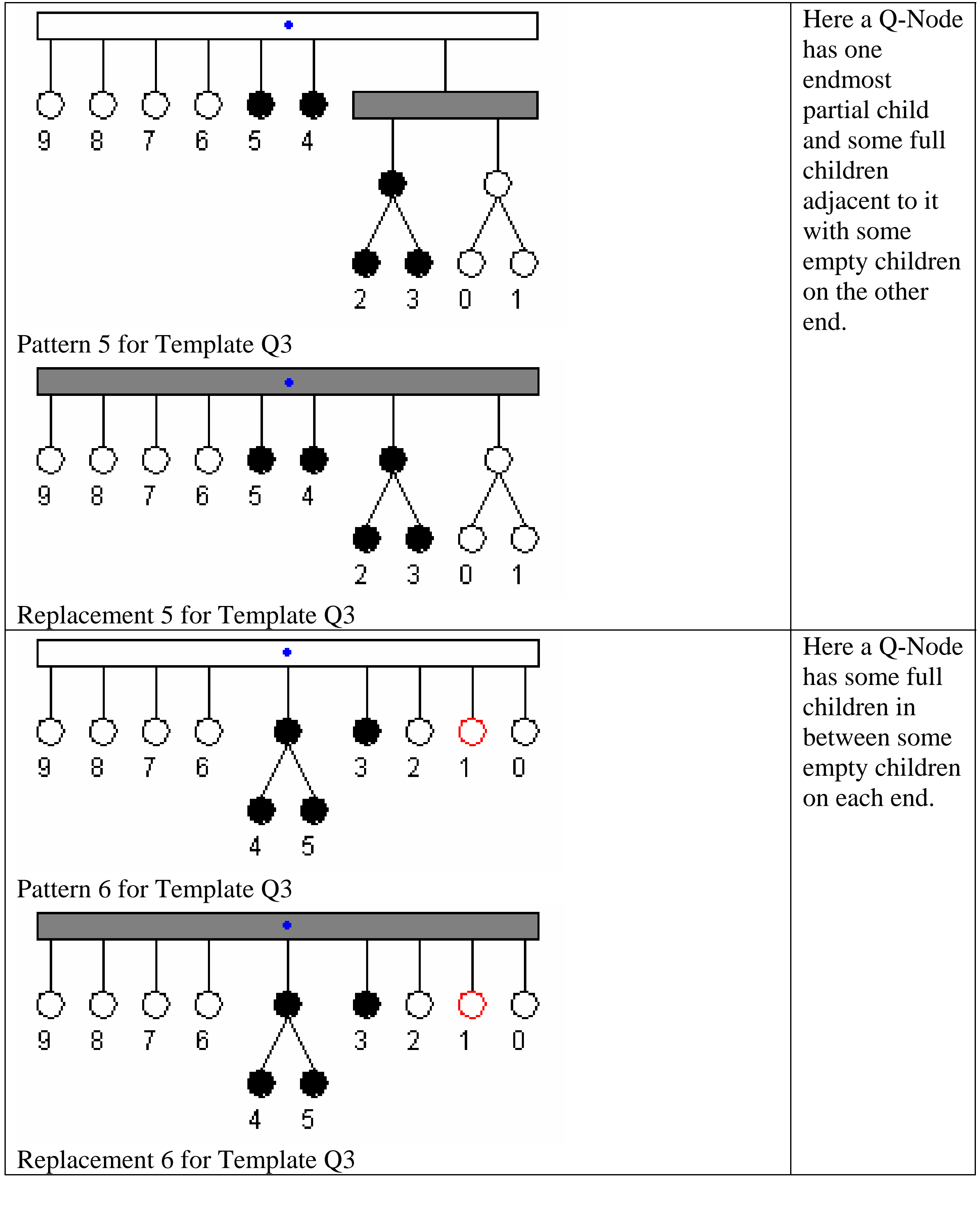
Pattern 2 for Template Q3

Replacement 2 for Template Q3

Here a Q-Node has two partial children, one interior and the other endmost, and some full children between them and some empty children on one end.



Pattern 3 for Template Q3 Replacement 3 for Template Q3	Here a Q-Node has two endmost partial children, separated by some full children.
Pattern 4 for Template Q3 Replacement 4 for Template Q3	Here a Q-Node has one interior partial child and some full children adjacent to it, and some empty children on each end.



Pattern 5 for Template Q3

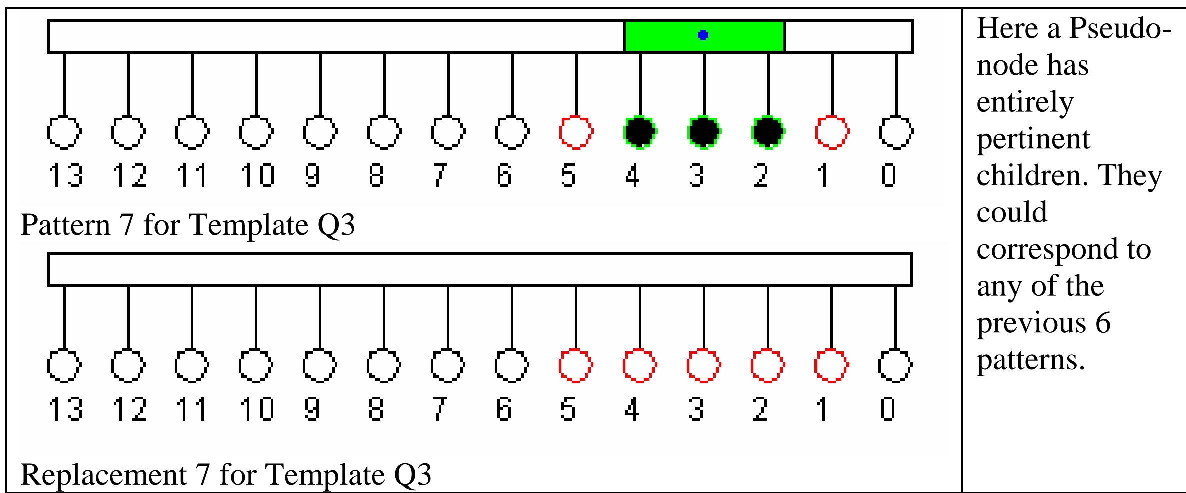
Replacement 5 for Template Q3

Here a Q-Node has one endmost partial child and some full children adjacent to it with some empty children on the other end.

Pattern 6 for Template Q3

Replacement 6 for Template Q3

Here a Q-Node has some full children in between some empty children on each end.



Pattern 7 for Template Q3

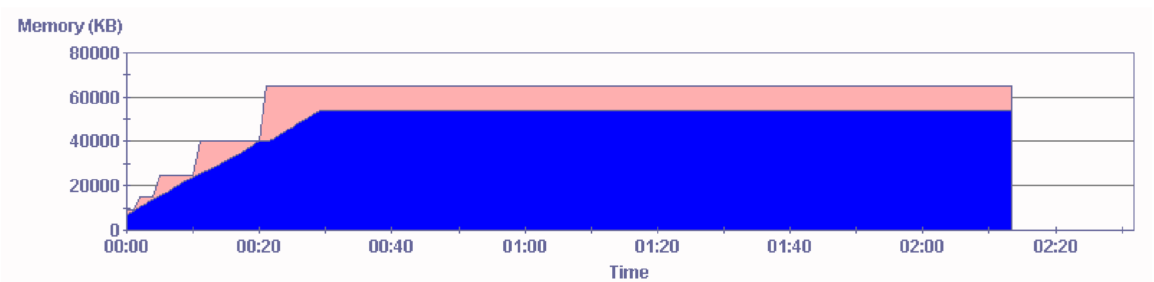
Replacement 7 for Template Q3

Here a Pseudo-node has entirely pertinent children. They could correspond to any of the previous 6 patterns.

<u>Note:</u>
In many of the templates illustrated above, not all possible combinations of full and empty children of a PQ-Node being matched to a pattern are shown.
(Ex: The pattern for a PQ-Node with a partial child and all full children is shown, but not the same case when the PQ-Node has a partial child and all empty children.)

Appendix B – Profiler Performance Analysis Results

The following figure shows the amount of memory that was consumed by the java virtual machine throughout a series of test cases for a PQ-Tree with 300,000 leaves. The bulk of the memory is taken up by the storage for the 300,000 PQ-Node objects. In order to achieve better performance and clarity, the graphical user interface was disabled during these tests.



The following table gives a summary of how much CPU time was spent on each of the methods used in the PQ-Tree implementation for the above test cases. The bulk of the CPU time was taken for initializing and resetting the PQ-Tree and adding the leaf nodes to the children.

Name	Calls	Cumulative Time	Method Time	Cumulative Objects	Method Objects	Source
.Root.	1	87,806	0	813,418	0	
Vector.removeElement(Object)	73	10	10	0	0	Vector.java
Vector.setElementAt(Object, int)	7	0	0	0	0	Vector.java
Vector.contains(Object)	5	0	0	0	0	Vector.java
Vector.addElement(Object)	400,460	1,412	1,412	6	6	Vector.java
Vector.<init>(int)	352	10	10	352	352	Vector.java
Vector.<init>()	628	10	10	628	628	Vector.java
Vector.elementAt(int)	2,801,344	4,366	4,366	0	0	Vector.java
String.concat(String)	674	30	30	1,348	1,348	String.java
String.equals(Object)	3	0	0	0	0	String.java
String.<init>()	76	0	0	76	76	String.java
String.<init>(String)	60	0	0	0	0	String.java
PrintStream.print(String)	120	60	60	120	120	PrintStream.java
PrintStream.println()	60	10	10	60	60	PrintStream.java
PrintStream.println(String)	137	40	40	274	274	PrintStream.java
AbstractList.iterator()	120	0	0	128	128	AbstractList.java
Integer.intValue()	1	0	0	0	0	Integer.java
Integer.valueOf(String)	1	0	0	1	1	Integer.java
Integer.toString()	188	0	0	376	376	Integer.java
Integer.<init>(int)	400,000	1,192	1,192	0	0	Integer.java
ClassLoader.checkPackageAccess(Class,	10	0	0	0	0	ClassLoader.java

ProtectionDomain)						
ClassLoader.loadClassInternal(String)	14	20	20	301	301	ClassLoader.java
StringBuffer.append(String)	519	0	0	230	230	StringBuffer.java
StringBuffer.append(int)	135	0	0	271	271	StringBuffer.java
StringBuffer.append(long)	68	0	0	207	207	StringBuffer.java
StringBuffer.toString()	196	0	0	196	196	StringBuffer.java
StringBuffer.<init>()	196	0	0	196	196	StringBuffer.java
System.gc()	7	0	0	0	0	System.java
System.currentTimeMillis()	136	0	0	0	0	System.java
.main.	1	87,806	190	813,418	1,773	
.Signal Dispatcher.	1	0	0	0	0	
.CompileThread0.	1	0	0	0	0	
PQController.test7(PQController)	1	9,373	0	553	12	PQController.java
PQController.test6(PQController)	1	9,133	0	1,079	22	PQController.java
PQController.test5(PQController)	1	9,313	0	1,182	26	PQController.java
PQController.test4(PQController)	1	9,454	0	476	12	PQController.java
PQController.test3(PQController)	1	9,373	0	1,434	28	PQController.java
PQController.test2(PQController)	1	9,223	0	538	12	PQController.java
PQController.test1(PQController)	1	9,243	0	1,435	36	PQController.java
PQController.getLeafAt(int)	188	0	0	0	0	PQController.java
PQController.resetTree()	7	64,813	0	35	0	PQController.java
PQController.reduceTree(Vector)	60	270	0	5,719	126	PQController.java
PQController.update()	358	20	10	0	0	PQController.java
PQController.update(boolean)	358	10	10	0	0	PQController.java
PQController.init(boolean)	1	0	0	0	0	PQController.java
PQController.main(String[])	1	87,616	0	811,645	35	PQController.java
PQController.<init>(int, boolean)	1	22,492	10	804,875	15	PQController.java
PQTree.templateQ3(PQNode, PQTree)	3	0	0	1	0	PQTree.java
PQTree.templateQ2(PQNode, PQTree)	25	0	0	0	0	PQTree.java
PQTree.templateQ1(PQNode, PQTree)	25	10	0	0	0	PQTree.java
PQTree.templateP6(PQNode, PQTree)	16	10	0	0	0	PQTree.java
PQTree.templateP5(PQNode, PQTree)	22	0	0	2	0	PQTree.java
PQTree.templateP4(PQNode, PQTree)	34	10	10	38	2	PQTree.java
PQTree.templateP3(PQNode, PQTree)	49	0	0	189	27	PQTree.java
PQTree.templateP2(PQNode, PQTree)	60	0	0	26	26	PQTree.java
PQTree.templateP1(PQNode, PQTree)	110	0	0	0	0	PQTree.java
PQTree.templateL1(PQNode, PQTree)	298	30	0	2	0	PQTree.java
PQTree.reduce(PQTree, Vector, PQController)	60	120	20	1,476	144	PQTree.java
PQTree.bubble(PQTree, Vector, PQController)	60	50	0	2,185	377	PQTree.java
PQTree.setTemplateTimeString(String)	60	0	0	0	0	PQTree.java
PQTree.setTemplateMatchString(String)	418	0	0	0	0	PQTree.java
PQTree.isNullTree()	60	0	0	0	0	PQTree.java
PQTree.getTemplateTimeString()	1	0	0	0	0	PQTree.java
PQTree.getTemplateMatchString()	1	0	0	0	0	PQTree.java
PQTree.reduction(PQTree, Vector, PQController)	60	190	10	4,085	64	PQTree.java
PQTree.getLeafAt(int)	188	0	0	0	0	PQTree.java
PQTree.resetTree()	7	64,813	9,153	35	21	PQTree.java
PQTree.<init>(int)	1	22,482	10,405	804,796	800,008	PQTree.java

PQNode.labelAsEmpty()	386	0	0	0	0	PQNode.java
PQNode.isPartial()	3,200,795	7,431	7,431	0	0	PQNode.java
PQNode.isFull()	3,201,883	8,012	8,012	0	0	PQNode.java
PQNode.checkPartialAreAtEnds()	1	0	0	0	0	PQNode.java
PQNode.checkFullAreAdjacentTo(PQNode)	2	0	0	0	0	PQNode.java
PQNode.absorbPartialChild(PQNode)	6	0	0	1	0	PQNode.java
PQNode.getEndMostChildren()	1	0	0	0	0	PQNode.java
PQNode.checkEndMostAreEmptyOrPartial()	24	0	0	0	0	PQNode.java
PQNode.checkFullAreAdjacent()	55	0	0	0	0	PQNode.java
PQNode.childrenAreFull()	23	10	0	0	0	PQNode.java
PQNode.isPseudoNode()	1	0	0	0	0	PQNode.java
PQNode.isQNode()	1,910	0	0	0	0	PQNode.java
PQNode.mergePartialChildren(PQNode, PQNode)	12	10	0	0	0	PQNode.java
PQNode.checkFullAreEndMost()	30	0	0	0	0	PQNode.java
PQNode.removeChild(PQNode, boolean)	697	20	20	0	0	PQNode.java
PQNode.removeChild(PQNode)	181	0	0	0	0	PQNode.java
PQNode.becomeChild(PQNode)	1	0	0	0	0	PQNode.java
PQNode.hasOnlyOneChild()	30	0	0	0	0	PQNode.java
PQNode.getPartialChild(int)	49	0	0	0	0	PQNode.java
PQNode.removeOnlyEmptyChild()	14	0	0	0	0	PQNode.java
PQNode.replaceChild(PQNode, PQNode)	31	0	0	0	0	PQNode.java
PQNode.replaceChild(PQNode, PQNode, boolean)	31	0	0	0	0	PQNode.java
PQNode.removeOnlyFullChild()	43	0	0	0	0	PQNode.java
PQNode.labelAsPartial()	52	0	0	0	0	PQNode.java
PQNode.moveFullChildrenTo(PQNode)	28	0	0	0	0	PQNode.java
PQNode.getNumEmptyChildren()	69	0	0	0	0	PQNode.java
PQNode.getNumPartialChildren()	1	0	0	0	0	PQNode.java
PQNode.getNumFullChildren()	1	0	0	0	0	PQNode.java
PQNode.getNumChildren()	85	0	0	0	0	PQNode.java
PQNode.isPNode()	3,202,097	6,219	6,219	0	0	PQNode.java
PQNode.labelAsFull()	233	30	0	2	0	PQNode.java
PQNode.hasChildren()	298	0	0	0	0	PQNode.java
PQNode.getPertinentLeafCount()	1	0	0	0	0	PQNode.java
PQNode.setPertinentLeafCount(int)	438	0	0	0	0	PQNode.java
PQNode.isDeleted()	1	0	0	0	0	PQNode.java
PQNode.isBlocked()	44	0	0	0	0	PQNode.java
PQNode.pseudoNode()	2	0	0	0	0	PQNode.java
PQNode.convertToQNode()	29	0	0	58	29	PQNode.java
PQNode.setQueued()	118	0	0	0	0	PQNode.java
PQNode.setPertinentChildCount(int)	480	10	10	0	0	PQNode.java
PQNode.getPertinentChildCount()	1	0	0	0	0	PQNode.java
PQNode.isUnblocked()	530	0	0	0	0	PQNode.java
PQNode.getSiblings()	1	0	0	0	0	PQNode.java
PQNode.setUnblocked()	249	0	0	0	0	PQNode.java
PQNode.getUnblockedSiblings()	254	0	0	508	254	PQNode.java
PQNode.getBlockedSiblings()	254	10	0	508	254	PQNode.java

PQNode.setBlocked()	254	0	0	0	0	PQNode.java
PQNode.addChild(PQNode)	3,200,210	58,714	10,155	146	0	PQNode.java
PQNode.addChild(PQNode, boolean)	3,200,725	48,570	26,919	150	73	PQNode.java
PQNode.setParent(PQNode)	26	0	0	0	0	PQNode.java
PQNode.delete()	25	0	0	0	0	PQNode.java
PQNode.isEmpty()	441	0	0	0	0	PQNode.java
PQNode.init(boolean)	400,090	691	691	0	0	PQNode.java
PQNode.clear()	372	10	0	0	0	PQNode.java
PQNode.clear(boolean)	386	10	10	0	0	PQNode.java
PQNode.getParent()	1	0	0	0	0	PQNode.java
PQNode.toString()	188	0	0	376	0	PQNode.java
PQNode.<clinit>()	1	230	0	4,685	4	PQNode.java
PQNode.<init>()	65	0	0	0	0	PQNode.java
PQNode.<init>(Object)	400,000	1,813	1,122	0	0	PQNode.java
SiblingVector.replaceSibling(PQNode, PQNode)	7	0	0	0	0	SiblingVector.java
SiblingVector.removeSibling(PQNode)	7	0	0	0	0	SiblingVector.java
SiblingVector.addSibling(PQNode)	130	0	0	0	0	SiblingVector.java
SiblingVector.nextSibling(PQNode)	3	0	0	0	0	SiblingVector.java
SiblingVector.siblingAt(int)	436	0	0	0	0	SiblingVector.java
SiblingVector.<init>()	73	0	0	73	0	SiblingVector.java
Color.<clinit>()	1	230	230	4,662	4,662	Color.java
.Reference Handler.	1	0	0	0	0	
.Finalizer.	1	0	0	0	0	
Queue.dequeue()	552	0	0	0	0	Queue.java
Queue.enqueue(Object)	604	10	0	655	606	Queue.java
Queue.size()	1	0	0	0	0	Queue.java
Queue.<init>()	120	0	0	0	0	Queue.java
AbstractList$Itr.hasNext()	496	0	0	0	0	AbstractList.java
AbstractList$Itr.next()	376	10	10	0	0	AbstractList.java
Node.getNext()	1	0	0	0	0	Queue.java
Node.setNext(Node)	408	0	0	0	0	Queue.java
Node.getElement()	1	0	0	0	0	Queue.java
Node.<init>(Object, Node)	604	0	0	0	0	Queue.java
.Thread-0.	1	0	0	0	0	

Appendix C – Notes on Compiling and Running the Java Implementation

To run the application simply type:
java –jar PQTree.jar

The main class which starts execution of the application is called PQController, so if you want to run the application after recompiling the source code type:

java PQController

Running the application without specifying any parameters causes usage information to be displayed to the console.

For help with operating the graphical user interface, click the help button therein.

Also, you may wish to change some of the constants which are defined in the file DebugInterface.java.

References

Kellogg S. Booth, George S. Lueker. *Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-tree Algorithms*. Journal of Computer and System Sciences, 13(3) pp. 335-379, 1976