Modifications of Patricia trees for handling values with discontiguous masks with emphasis on internet routing applications

Joel M. Halpern
Network Systems Corporation
7600 Boone Avenue North
Brooklyn Park, MN 55428
jmh@anubis.network.com

October 15, 1991

Abstract

In many environments, information about values that are associated with masks must be maintained. This is particularly true in network routing, although uses in other contexts arise. Two recent work areas on this topic have been associated with the use of variable masks and the use of Patricia Trees for table management. A refinement on variable masks that has been suggested is to allow discontiguous masks. This paper lays out a set of restrictions to insure that discontiguous masks are well defined, and then provides an algorithm for maintaining a table of such values.

1 Introduction

In data management, one of the common situations is the maintenance of large tables of values from a much larger space. Binary trees are frequently used to maintain such values. Knuth [Knuth] documents a tree structure (invented by Donald Morrison [Morri]) that is particularly efficient when one does not wish to compare values at every node: the patricia tree. This article discusses some refinements and applications of this tree structure and the algorithms for manipulating it.

Tradidional patricia trees as developed have many elegant and useful properties, (many of which are discussed in [Sedge]) include tree shape invariance and a natural method for doing lexicographic traversal. However, the values at specific nodes are dependent on the order in which entries were inserted. This is no problem for normal data. However, in many areas rather than simple exact matches on data, one wishes to use matching with a mask, where a target value is found in the tree if it matches the tree entry's value when masked with that entry's mask.

This problem arises when one tries to use patricia trees to represent network addresses. For example, TCP/IP has addresses that may be matched on the full address, on just the network portion, or on some part in between. Van Jacobsen [VanJa] suggested that this problem could be overcome and that patricia trees would be useful for representing IP forwarding tables, since IP addresses tend to cluster into ranges and sets so that discrete bit testing is effective.

We have developed a simple enhancement to the patricia algorithm that permits the use of values with masks as entries in the table. The modified algorithm does have to perform more than one compare in certain situations. This is a reduction in efficiency, but not a severe reduction. The number of full address comparisons required for a lookup is still usually smaller than for conventional binary trees.

Recent work on IP addresses for TCP, and NSAPs for OSI, has suggested that routing and address assignments would be enhanced by permitting masks to become far more general than had previously been allowed. This paper will include a definition of valid sets of addresses and discontiguous masks, along with a more restrictive recommendation to permit bitwise processing of address, mask combinations. The modified patricia algorithm described here is usable for any situation where value matching is done under masks, when the value/mask sets meet the constraints given. The paper will also prove that the algorithm will give the desired results on any valid set of inputs.

1.1 Organization

Section 1 is the introduction, describing the objectives and contents of this paper.

Section 2 is a description of the original patricia algorithm, including the lookup and insertion code and descriptions of the general operation. Certain simple properties of patricia are proven here.

Section 3 is a description of the problems and solutions for looking up values with associated masks, where the masks are contiguous from the most significant bit. Subsection 3.1 is a review of the meaning of contiguous masks as used for TCP/IP addressing, and the interpretation of sets of such addresses and masks. Subsection 3.2 is the modification of the patricia algorithm to handle contiguous masked values. Certain properties of this are proven, including that it implements the interpretation of addresses described in section 3.1.

Section 4 describes the usage of values with discontiguous bit masks and the tree algorithms to maintain and look up information about them. Subsection 4.1 describes the general validity criteria for use with discontiguous masked values. After describing a method for verifying this condition, certain difficulties with using the general validity criteria are identified. This condition is then refined into one that allows for bitwise decidiability. Some perspective on the meaning of this condition is then provided. Subsection 4.2 then provides the full patricia algorithm to handle this case. The correctness of this algorithm is then proven.

Finally, in section 5, some areas for further work are outlined, including some complexity and flexibility questions.

Note that while this discussion focuses on IP addresses, it is applicable in other areas. The applicability to other kinds of hierarchical addresses follows naturally. There are other areas where masked values are maintained, and discontiguous masks might provide administrative flexibility. For a description of managing multilayered hierarchical address spaces efficiently using discontiguous masks, see the work of Paul Tsuchiya [Tsuch].

1.2 Coding Assumptions

Several algorithm excerpts are included in this article. The algorithms presented will be based on C code, simplified and modified for clarity. Bit-testing and comparison operations will all be done using macro calls, recognizable by their uppercase names. Of particular note is the macro HIGHER which will be used to compare two bit selections to see if the first one is for a node higher in the tree than the second one (ie if it selects a more significant bit). In the final algorithm this macro will use an explicit node depth counter, instead of using the bit selection to infer relative depth. The general data structures and macros used are given in figure 1. The type are generic TYPEDEFs, since those are implementation specific. relative depth.

```
typedef struct patricia_struct {
    BIT_SELECT bit;    /* what bit this node tests */

    KEY_TYPE   key;    /* The value */
    KEY_TYPE   mask;   /* The mask, for other than basic case */
    KEY_TYPE   common; /* bits common to all masks lower down */

    int        pos;    /* depth - for discontiguous case */

    struct patricia_struct *left, *right; /* child pointers */
} patricia_node;

#define TEST_BIT(select, value) /* is the selected bit set? */
#define NEXT_BIT(select) /* generate the next sequential bit selector */
#define MAX_BIT          /* The most significant bit selector */

#define HIGHER(select1, select2) /* is select1 for a node higher
                                    in the tree than select2? */
/* replace with ``pos1 < pos2'' for discontiguous case */

#define EQ_KEY(key1, key2) /* are two keys equal? */
#define MASK_KEY(key, mask) /* apply the mask to the key */
#define SUBSET(key1, key2) /* all bits set in key1 are set in key2 */
```

Figure 1: Patrica structure and Macros

2 Patricia

The standard patricia algorithm is a method for maintaining trees of values such that traversal is on the basis of testing selected bits. Bits are only tested if they are the best distinguishing bit between at least two values in the table.

When looking up a value in a patricia tree, as shown in figure 2, one traverses the tree, branching on the basis of the value of tested bits, until a pointer into the traversed portion of the tree is found. The specific value in that node is then compared with the value being looked up. If it does not match, the sought value is not in the table.

Note the use of up pointers into the table to terminate the search. There is an alternative, equivalent data structure which uses explicit leaf nodes. That would be called a patricia trie, instead of a tree. The two methods of storing and lookup are equivalent, and structures of one type can be transformed into the other. The two structures are shown in figure 3. The difference between the two approaches is a matter of implementors' preference. In later algorithms in this paper, we will use the values stored in a node during a reverse traversal of the node. If a patricia trie is desired, one could use a pointer to the leaf where the appropriate value is stored instead. This does not affect any re-arrangement or structuring issues. For this paper, we will concentrate specifically on patricia trees.

The code for performing a lookup in a patricia tree, as shown in figure 2, is very simple. The traversal is exactly as described above. When an up pointer is encountered, the search is finished and the result is returned. Note that this code does not check what value is found, but merely returns the node that matches the key on all tested bits. For normal lookups, the caller must compare the key in the returned node with the key sought. If the two do not match, the key being sought is not in the table. The reason for returning what is found, instead of doing the comparison in `patricia_find` is that `patricia_insert` will use `patricia_find`, and needs the node found.

Inserting new values in a table (the code given in figure 4) is somewhat more interesting. First, note that an empty patricia tree consists of a root node with no bit tested, a key of zero, and both left and right

```
patrica_node *patricia_find(KEY_TYPE key; patrica_node *root)
{
    patricia_node *parent, *trav;

    trav = root;

    do {
        /*
         * To detect the termination, we must find a pointer that
         * does not descend in the tree.  This requires saving
         * the current node before proceeding down in the direction
         * selected by the tested bit.
         */
        parent = trav;

        if (TEST_BIT(trav->bit, key))
            trav = trav->right;
        else
            trav = trav->left;
    } while (HIGHER(parent->bit, trav->bit));

    return (trav);
}
```

Figure 2: Original Patricia Lookup Code

pointers pointing to itself. The first entry will always go on the left-hand side of the root, and all values will go below that. The right-hand side of the root is not used.

Logically, the insertion process can be thought of as having three phases. Phase one (through the end of the first loop) is the bit selection phase. Phase two (through the end of the second loop) is the tree positioning. Phase three is the actual insertion. Note that the inserted (or found, in the case of duplication) node is returned so that additional information can be added to the node by the installing routine.

In phase one of the insertion, the tree is searched for a value that matches the key on all tested bits. This is always possible. If it is an exact match, we have a duplicate and the correct handling is up to the caller. If it is not a duplicate, then there is at least one bit on which the two values differ. We select the most significant.

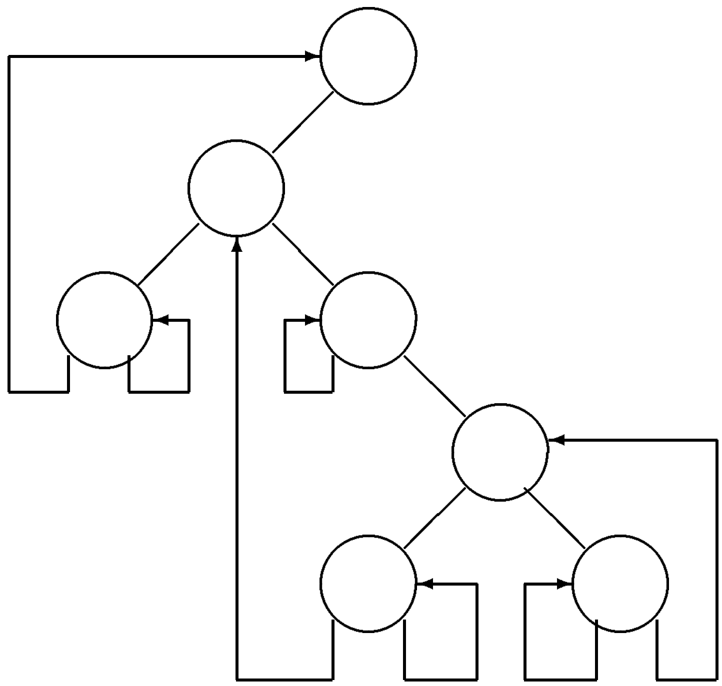
Phase two is needed because in basic patricia one always tests bits in order of significance. Therefore, we go through the tree to find where this bit should be tested. If this goes somewhere higher in the tree than the bottom, we will find a pair of nodes such that the child is testing a bit less significant than the decisive one. The new node goes between these two. If there is no such node, then since we are retracing the original lookup path, we will find an up pointer pointing to the same final node. At that point, the parent node is pointing to a node which, at least on the side to which we belong, has no children. We can insert the new node so that it is between the parent node and the up pointer.

Phase three is the insertion of the new node. Knowing the tested bit, we can easily insert the lower portion of the tree `trav` on the correct side of the new node. Similarly, testing the bit in the key selected by `parent` tells us which side of `parent` the new node goes on.

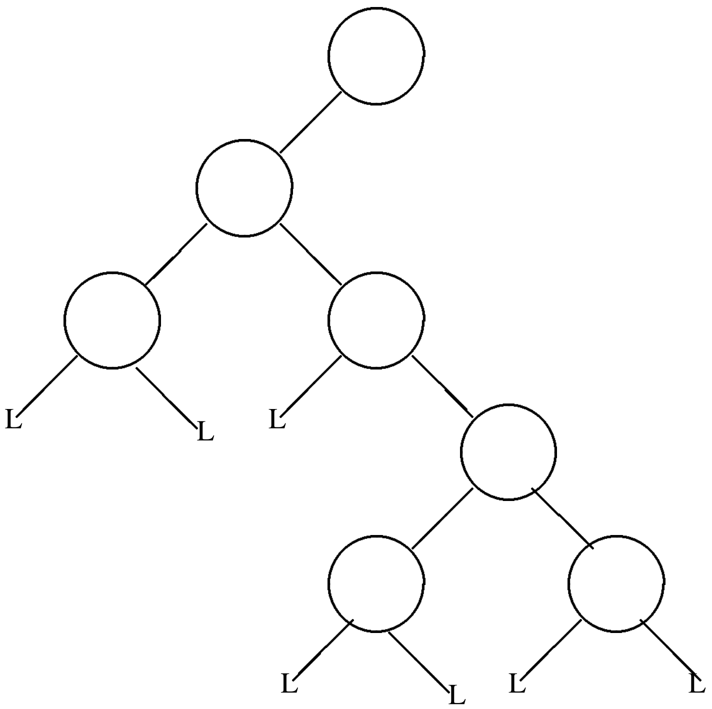
This algorithm produces a tree in which only significant bits are tested, and bits are tested in order of significance. There are some useful properties of patricia trees.

Proposition 1 *The upward pointers of the tree are uniformly spread. Specifically:*

- a *Given any non-root node in a patricia tree, there is exactly one pointer in the subtree rooted at that node that points above that node.*



Patricia Tree



Patricia Trie

Figure 3: Patricia Tree and Trie

```
patrica_node *patricia_insert(KEY_TYPE key; patrica_node *root)
{
    patricia_node *parent, *trav, *new_node;
    BIT_SELECT try_bit;

    trav = patricia_find(key, root);

    if (EQ_KEY(trav->key, key)) return (trav);

    try_bit = MAX_BIT;

    /*
     * Find the first bit by which the new key differs from
     * what was found.  There must be one.
     */
    while (TEST_BIT(key, try_bit) == TEST_BIT(trav->key, try_bit))
        try_bit = NEXT_BIT(try_bit)

    /*
     * now check whether, on the basis of this bit,
     * the new node belongs somewhere else in the tree.
     */
    trav = root;
    do {
        /* as in the lookup, save the parent */
        parent = trav;

        if (TEST_BIT(key, trav->bit))
            trav = trav->right;
        else
            trav = trav->left;
    } while (HIGHER(trav->bit, try_bit) &&
             HIGHER(parent->bit, trav->bit))

    /*
     * now insert the new node between parent and trav
     */
    new_node = alloc_node();
    new_node->key = key;
    new_node->bit = try_bit;

    if (TEST_BIT(key, try_bit)) {
        new_node->left = trav;
        new_node->right = new_node;
    } else {
        new_node->left = new_node;
        new_node->right = trav;
    }

    if (TEST_BIT(key, parent->bit))
        parent->right = new_node;
    else
        parent->left = new_node;

    return (new_node);
}
```

Figure 4: Original Patricia Insertion Code

b *Given any non-root node in a patricia tree, there is exactly one pointer in the subtree rooted at that node which points to that node.*

Proof: By Induction and case analysis:

1. When the first node is inserted in a tree, one of its two pointers is set to point to itself, and the other is set to point to the root. This meets proposition 1.

2. When a new node is added at the bottom of the tree, it is added with a pointer to itself, and the up pointer, on its two sides. Thus it meets the all of the listed properties. Since no pointers were removed from the original tree, and no pointers pointing outside the subtree were added, adding a new node at the bottom of the tree does not affect the previous state, in which all containing subtrees met the proposition.

3. When a new node is added in the middle of the tree, it is added with one pointer pointing to itself, and the other pointer pointing to the subtree that was on this side of its parent node. As such, it does not affect the properties of the parent subtree. The properties are preserved if this subtree contains exactly one pointer that points above the new node (since the other pointer from the new node points to itself). This subtree was a direct subtree of the parent node. As such, by the induction process, it either contains exactly one pointer to the parent node, and none higher, or exactly one pointer to a node higher than the parent. In either case, it contains exactly one pointer to a node higher than the new node.

Note that this proposition implies that the two pointers, one pointing to the subtree root and one pointing above that in the tree, must be on different sides of the subtree root.

Proposition 2 *Given a node A with a child pointer B, that points somewhere at or above A in the tree, B points to a node along the path from the root to A.*

Proof: By Induction

1. A one-node tree has a single node A, whose child pointers point to A and to root. Thus meeting the proposition.

2. When a node is added to the tree, if it is inserted in the middle of the tree, the only pointers modified are the parent pointer (which was not and is not an up pointer,) and the child pointers of the new node. One of these child pointers points down, and the other points to the new node. Thus, with regard to changed pointers, the property is preserved. Since no nodes were removed from a path from the root to any node, the property is preserved for all other ndoes and pointers.

3. When a node is added to the bottom of a tree, it is added under a node that had an up pointer pointing along the path to the node. This becomes one of the two child pointers of the new node, and points along the path from root to the parent of this node, so points along the path from root to this node. The other child pointer points to this new node, hence it points along the path from the root to this node.

Proposition 3 *Given a node A, which tests bit b_A, one of its two child pointers represents the choice on b_A that agrees with A. All nodes under that subtree match A up through b_A.*

The immediate child of A on this side, B, matches A up to b_B. There is a path through the subtree rooted at A, P, which branches at each node according to the settings, in the value of A, of the bit selected at that node. For nodes on P, the values in these node match A up to the tested bit. For nodes off of this path, they match A up to, but not including, the bit tested at the branch point from the path.

Proof:

1. When a new node is added to the tree, it is always added with the pointer corresponding to the value of the tested bit in the node pointing to the new node.
2. When the first child node C is added under A on this side, it matches A on all tested bits. Therefore, it is compared with A to select a tested bit. If it is inserted below A, then it must select a tested bit beyond b_A, and therefore agree with A up to that bit, where it disagrees with A.
3. Any node that agrees with A on all tested bits up to A, and disagrees on some subsequent bit, will branch from the path P at some node C. It will agree with C on the bit tested there, and C will agree with A up to b_C. Therefore, the node node will be compared with a value that agrees with A up to b_C. If it differs before that, it will be inserted appropriately such that it will agree with A up to its tested bit. Otherwise it will be inserted under C, and agrees with A up to b_C, where it branches from path P.
4. Any node that agrees with A on all tested bits will be compared against A for bit selection, and will be inserted such that it agrees with A up to the tested bit (unless it differs before b_A, in which case it will be inserted above A.)

Note that the path P in the above proof is the path traversed when looking up A in the tree.

Proposition 4 Sorting*: Traversing the tree in depth-first order, left then right subtrees, and processing nodes only when they are encountered by up pointers, one will process nodes in lexicographic order.*

Proof: By induction and case analysis:

1. A tree with one node A in it meets the sorting constraint. The one node will test the most significant bit set in the value in that node, b_A. Traversing, we will reach the root, value 0, and then the node.
2. Adding a second node B, one of four cases can occur.
   (a) b_A is not set in B, but a more significant bit b_B is set. The first stage search will end at the root, bit selection will select b_B. The location process will insert B above A, with A under the zero branch, and B under the one branch. Traversal will now produce (root(0), A, B), which is correct. (0 is less than A and B, while the fact that B has a more significant bit set than A means that A is less than B.)
   (b) b_A is not set in B, and the first set bit, b_B, is less significant than b_A. The search will still terminate at the root, but the bit location procedure will put B under the zero branch of A. The new node will have the not-set branch pointing to root, and the set branch pointing to B. The traversal will produce root, B, A, which is correct since A has a more significant bit b_A set than the most significant set bit in B (b_B).
   (c) b_A is set in B, but a more significant bit, b_B, is also set in B (B is greater than A). The search will terminate at A, the bit selection will select b_B, and the insertion will insert B between root and A. A will be on the not-set side of the test of bit b_B, and B will be on the set side. The traversal will produce root, A (the two sides of node A), B.
   (d) b_A is set in B, which is the most significant set bit in B, and B differs from A on bit b_B. The search will terminate on A, the bit selection will select b_B, and the positioning will put B under the set branch of A. The pointers from node B will be set to reflect the relative settings of bit b_B in A and B. This will mean that the left side of A leads to the root. The right side of A leads to B. The left side of B leads to whichever value of A and B does not have bit b_B (the most significant bit on which they differ) set, and the right side will point to the one that has the bit set. Thus the traversal will produce proper results.

Thus, a two-node tree meets the proposition.

3. Given a tree that meets the proposition, when adding a new node C, one finds the node T which matches C on all bits tested in the path through the tree. T is pointed to by a node P, which tests bit b_P.

   (a) If C differs from T on a bit that is less significant than b_P, then it will be inserted in the tree under P, with pointers to itself and T. As in the insertion of the second node, it will be printed appropriately either before or after T. Given that T occurred in the proper place in the traversal, and C agrees with T up to the bits tested, it will occur in the proper place.

   (b) Otherwise, C differs from T on some bit, b_C, more significant than b_P, and will be placed in the tree along the path just traversed between some other two nodes, A and B.

   (c) Note that C agrees with T through bit b_C, which is less significant than b_A, so C clearly belongs in the subtree under A. Specifically, we know that there are no values in this subtree that differ from T on any bit between b_A and b_B, or they would be above B in the tree. Therefore, b_C is the sorting bit to correctly place C.

3 Masked Patricia

When manipulating a network address, or any other value where portions of a value have meaning at different parts of a hierararchy, one frequently finds that one needs a data structure to manage this information which can perform matches based on "relevant" portions of the address. For TCP/IP, this relevance was first defined based on the value in the address (referred to as Class A, B, and C addresses). In this, depending on the first one or two bits of the value, some one, two, or three bytes of the address denoted the network it was on, and the remainder denoted the specific host within the network. As structures grew more complex, provision for a scope of interpretation between the network and the full address was added, the subnet. This was identified by a mask. Recently there has been more work allowing variable length masks, and multiple masks within a subnet. Thus, managing this information begins to look like the general problem of managing masked information. Historically, all such masks have been contiguous, and set starting at the most significant possible bit. This is a simpler case than one of arbitrary, discontiguous bit masks. This section of the paper will treat this contiguous case (where the set bits start at the most significant bit in the value, and are contiguous from there for as much as is selected by the mask). The ensuing section will generalize this to arbitrary masks.

3.1 Address Matching

When one specifies that values are matched based on masks, one must define what is a meaningful table entry, and what is a proper match.

A table entry A is a pair of values, V_A and M_A, where V_A is the value, and M_A is the appropriate mask. It should be noted that there can be no significant information in V_A outside of that portion selected by the mask M_A (i.e. $V_A \wedge M_A = V_A$).

Given a candidate value V_B, this is matched by an entry A if $V_B \wedge M_A = V_A$. This says that on that part of the address which A considers relevant, B matches A.

Given a collection of entries A_i, V_B could potentially be matched by several entries. This is permissible. Contiguous masks starting at the left can be viewed as prefixes, and this says that several of the prefixes of V_B (there are as many prefixes as there are bits in V_B) occur in the collection. It is then necessary to select one as the best match. Given the structure of masks, it is clear that what one wants is the longest matching prefix. This corresponds to the mask with the most bits set, or the mask with the highest numerical value. More formally, given A_i and A_j such thatV_B matches both of them as described in the previous paragraph, then the correct match is A_i if $M_{A_i} > M_{A_j}$, (with the masks considered as unsigned numeric quantities) and A_j if the obverse holds. (If $M_{A_i} = M_{A_j}$, then $i = j$ since there can not be two distinct entries with the same mask and the same value.)

Given the definitions above, there is one restriction on the range of meaningful values which should be recognized. It is not possible in this patricia structure to represent two node, a parent and a child, such that the portion of the child outside of the mask of the parent is zero. This is because, viewed as unmasked quantities, these two values are identical. Hence, there is no bit that can be tested in a value to distinguish them, so this algorithm will consider them to be the same. This restriction is already present in many address allocation schemes for other reasons. It should also be noted that unless some special ending indicator is used, the same restriction applies to variable length addresses and extensions.

3.2 Masked Tree Algorithm

Given the above definition of matching, the question is how to build and use a patricia tree that will produce the correct results.

The method is to produce a patricia tree such that one will find the sought node somewhere in the path one traverses down the tree. Then in the lookup procedure, if the correct value is not found, check back up the nodes traversed, until the correct one is found. While this gives up some of the reduction in comparsions that one gained with basic patricia, it is still much better than examining every node on the way down, particularly if comparisons are expensive, as with OSI NSAPs (160 bit addresses). The code to do this is in figure 5.

Conceptually, this starts by testing the best match, and then starts ignoring bits until it finds a value that matches appropriately. The `return(NULL)` is for an error case that should never occur, since, by structure, the root of the tree is pushed first, and that, because it has a value of zero and a mask of zero, will match anything. This gives an appropriate place for storing a global default result.

The more significant work is in the insertion routine. It is necessary to insert values in such a way that for two entries in the tree, A and B, if $V_A \wedge M_B = V_B$, then B is higher in the tree than A. Basically, one simply tests for this condition on the way down, and inserts where necessary.

Specifically, figure 6 shows the initial code traversing the tree. It is equivalent to the first part of the lookup routine.

Figure 7 shows what happens if the value being inserted is already present in the tree. The first check is for mask extension. As we show in proposition 9, if the mask is being extended, there is no need to rebuild the tree, so the mask can just be changed.

If it is being shortened, we must check to see if there is a different place where it must be inserted. Specifically, we search down the tree to see if there is a matching value above which this new one must be put. If not, just replace the mask and be done with it. Otherwise, record the location, using `parent` and `next`, and fall through to the code shown in figure 9.

If there is no exact match in the tree, then we must determine where to insert the new node in the table. As shown in figure 8, this is equivalent to the location logic for normal patricia, except that we have one extra condition, `EQ_KEY( MASK_KEY(trav->key, mask), key)`, which will terminate the traversal. This check says that we have found a node `trav` such that the new node must be inserted above it. This loop termination condition is then the condition for the tree rebuild code in figure 9. The reason we must force this position is that the new node is a prefix of the old node. Therefore, in order for the backtracking to work properly, the prefix (parent) of the node must be higher in the tree than the node. Otherwise, when looking for some other child value, the parent will not be where the backtrack will find it. (Obviously, it

```
    patricia_node *patricia_find (KEY_TYPE key, patricia_node *root)
    {
        patricia_node *parent, *trav;
        patricia_node *pm_stack[MAXBITS+1];
        patricia_node **pm_stack_ptr;

        /*
         * Standard Patricia search, and
         * remember the nodes visited along the way.
         */
        INITSTACK();
        trav = root;

        do {
            /*
             * save the parent to detect the up-pointer
             */
            parent = trav;

            PUSH(parent); /* push what we have passed */

            if (TEST_BIT(trav->bit, key))
                trav = trav->right;
            else
                trav = trav->left;
        } while (HIGHER(parent->bit, trav->bit);

        PUSH(trav); /* so it is tested first */

        /*
         * Search up the tree for the best match
         */

        while (!STACKEMPTY()) {
            trav = POP();
            if (EQ_KEY( MASK_KEY(key, trav->mask), trav->key))
                return (trav);
        }

    return (NULL);
}
```

Figure 5: Patricia lookup with Masks

```
patrica_node *patricia_insert(KEY_TYPE key;
                              KEY_TYPE mask;
                              patrica_node *root)
{
    patricia_node *parent, *trav, *next;
    BIT_SELECT try_bit;

    /*
     * we can not call patricia_find,
     * since we need the actual search termination with backtrack
     */
    trav = root;
    do {
        parent = trav;

        if (TEST_BIT(trav->bit, key))
            trav = trav->right;
        else
            trav = trav->left;
    } while (HIGHER(parent->bit, trav->bit));
```

Figure 6: Patricia Masked Insertion - Step 1 - lookup

```
    /*
     * If the new value is actually in the tree, we must
     * determine if masks have changed, and cope with that.
     */
    if (EQ_KEY(trav->key, key)) {
        /*
         * test for the simple case of identical or extending mask
         */
        if (mask >= trav->mask) {
            trav->mask = mask;
            return (trav);
        }

        /*
         * The mask is becoming shorter, check to see if
         * the node needs to be moved up.
         */

        next = root;
        do {
            /* save the parent */
            parent = next;

            if (TEST_BIT(key, next->bit))
                next = next->right;
            else
                next = next->left;
        } while (next != trav &&
                 !EQ_KEY( MASK_KEY(next->key, mask), key));

        if (next == trav) {
            /* there is no need to move it */
            trav->mask = mask;
            return (trav);
        }
    }
```

Figure 7: Masked Insertion - Step 2 - Found match

```
else {

    try_bit = MAX_BIT;

    /*
     * Find the first bit by which the new key differs from
     * what was found.  There must be one.
     */
    while (TEST_BIT(key, try_bit) == TEST_BIT(trav->key, try_bit))
        try_bit = NEXT_BIT(try_bit)

    /*
     * Now check whether, on the basis of this bit,
     * the new node belongs somewhere else in the tree.
     *
     * In addition to the standard checks, we test to see if
     * there is a node which must be below the new one, defining
     * an insert position just above it.
     */
    trav = root;
    do {
        /* as in the lookup, save the parent */
        parent = trav;

        if (TEST_BIT(key, trav->bit))
            trav = trav->right;
        else
            trav = trav->left;
    } while (HIGHER(trav->bit, try_bit) &&
             HIGHER(parent->bit, trav->bit) &&
             !EQ_KEY( MASK_KEY(trav->key, mask), key));
}
```

Figure 8: Masked Insertion - Step 3 - Locate Insertion

```
/*
 * Now test to see if we are doing an unusual middle insert.
 * If so, we must remove the lower subtree, add the node, and
 * then re-insert the subtree.
 */
if (EQ_KEY( MASK_KEY(trav->key, mask), key)) {
    /*
     * Find where the pointer from parent would have pointed,
     * if there had been no subtree under it.
     */
   if (TEST_BIT(key, parent->bit) {
       /* we are in the right subtree */
       if (TEST_BIT(parent->key, parent->bit))
           /* parent's right would have pointed to parent */
           pp = parent;
       else
           pp = find_up(parent->right, parent->bit);
    } else {
        /* we are in the left subtree */
        if (!TEST_BIT(parent->key, parent->bit))
            /* parent's left would have been parent */
            pp = parent;
        else
            pp = find_up(parent->left, parent->bit);
    }

        /* determine which bit we differ from pp on */
    try_bit = MAX_BIT;
    while (TEST_BIT(key, try_bit) == TEST_BIT(pp->key, try_bit))
        bit = NEXT_BIT(bit);

    /* Create new node */
    n = Create_Node (key, mask, try_bit);
    if (TEST_BIT(key, try_bit)) {
        n->left = pp;
        n->right = n;
    } else {
        n->left = n;
        n->right = pp;
    }
```

Figure 9: Masked Insertion - Step 4 - Tree Rebuild / Part 1

does not matter when looking for this precise value, since it is in the tree.)

If it is determined that the new node must be inserted in a special location, then potentially the entire subtree under that will need to be rebuilt, since the structure will change. Figures 9 and 10 show this work.

In figure 9, having decided we are rebuilding, it is necessary to determine sufficient information to rebuild. One uses as an anchor the parent under which we are rebuilding. Only one side must be rebuilt, and that side is identified for work.

The rebuild process consists of creating the tree as it would have been without the subtree which is being rebuilt, but with the new node. All of the nodes from the subtree are then re-inserted into the new tree. To create this stub tree for the insertion, it is necessary to create a new node, with two child pointers. One pointer will point to the new node, while the other either points to the parent, or points higher up in the tree. Proposition 1 insures that there is exactly one pointer. To simplify operations, the algorithm separates the case where the up-pointer points to the parent and where it points higher. The case where it points to the parent is detectable by a simple bit test. The routine `find_up` is used otherwise, to find out where the up-pointer points. It recursively searches the tree looking for that pointer. In either case, given the correct

```
        /*
         * Determine which side of p we go on,
         * And if there is a subtree there,
         * Rebuild it under the new node.
         */
        if (!TEST_BIT(key, parent->bit) &&
                HIGHER(parent->left->bit, parent->bit))
            Walk_Default (trav, Install_Node, n);
        else if (!TEST_BIT(key, parent->bit) &&
                     HIGHER(parent->right->bit < parent->bit))
            Walk_Default (trav, Install_Node, n);

        /*
         * Hook-up the rebuilt tree
         */

        if (TEST_BIT(key, parent->bit))
            parent->right = n;
        else
            parent->left = n;

        /*
         * Free all nodes in the old subtree trav.
         */

        Walk_Default (trav, Free_Node, NULL);
    }
```

Figure 10: Masked Insertion - Step 4 - Tree Rebuild / Part 2

value for this child pointer, it is easy to build the new node to use to hold the new value and serve as a root for the tree rebuilding.

The node pointed to by this child pointer is used to select which bit to test in the new node. This produces the correct results since if the subtree had not been present, that higher node is the one that would have been used by the conventional portion of the patricia code to select a bit to test. Also, the positioning of the child node which is being displaced (the node which forced the early insertion to start this section) insures that a sufficiently significant bit is required for comparison so the new node does not belong higher in the tree.

In figure 10, having built the new node, the code checks to see if there is a subtree in place. If so, the nodes in it are copied into the tree under the new node, one at a time. By doing the traversal in depth first order, no additional recursion should be required. Then, one simply inserts the new subtree into the tree, and gets rid of the old nodes.

Finally, figure 11 is simply the classic case of inserting a new node in a patricia tree after properly selecting a bit.

Given this code, there are several properties that must be verified. One must insure that if a new node is being inserted, that it will always be forced to a correct place when necessary. Then, it must be shown that the insertion still produces a correct patricia tree.

Proposition 5 *Given a masked value in a patricia tree, the bit selected for testing will always be within the mask of the value.*

Proof: Patricia always selects, as the bit for testing in a new node, the most significant bit on which the new value differs from the comparison target. By the algorithm, the comparison target will never be a value that matches the new address on all masked bits. (If it were, the algorithm would force the new node above that one, and select a different node for the bit selection comparison). Since masks are contiguous,

```
    else { /* normal case */
        /*
         * now insert the new node between parent and trav
         */
        next = alloc_node();
        next->key = key;
        next->bit = try_bit;

        if (TEST_BIT(key, try_bit)) {
            new_node->left = trav;
            new_node->right = new_node;
        } else {
            new_node->left = new_node;
            new_node->right = trav;
        }

        if (TEST_BIT(key, parent->bit))
            parent->right = new_node;
        else
            parent->left = new_node;

        return (next);
    }
}
```

Figure 11: Mask Insertion - Step 5 - Normal insert

starting at the most significant bit, and the two values do not match on all bits which are within the mask, the selected bit will be within the mask.

Proposition 6 *When inserting a new value A in the tree, if there is a node B such that $V_A \wedge M_B = V_B$ (A is an extension of B and must be inserted below B), then the traversal for locating the insertion point for A will pass through node B, and A will be inserted below B.*

Proof: Bits are tested in order of significance. By proposition 5, the bit tested in node *B* must be within the mask of *B*. Since *A* matches *B* that far, and since propostion 2 says that *B* is along the path to finding *B*, *A* must go through that node and down further. In fact, all values that agree with *B* on the bits tested that far will be below *B*. Therefore, the final target selected for comparison with *A* will either be *B*, or a node below *B* that agrees with *B* on all tested bits.

Proposition 7 *When inserting a new value A in the tree, if there is a node B such that $V_B \wedge M_A = V_A$ (A goes above B), then the traversal for locating the insertion point for A will select a point above B, and along the path from the root to B.*

Proof: Given that all tested bits for the new value will be within the mask, either the new value will attempt to go below the old value (in which case it will go above it), or it will be inserted naturally at a higher spot in the path. Specifically, it will go above all nodes that test values outside its mask. Therefore, it will be on the path leading to the longer old value.

Proposition 8 *If the mask associated with a value A is being changed, there is no possibility of a node lower than this one needing to be moved up.*

Proof: By Contradiction:

If there is a value *B* that must be moved up in the tree above *A* with the new mask, then $V_A \wedge M_B = V_B$, but this condition is not affected by the mask associated with *A*, hence would have been true before the change in mask, and therefore does not hold.

Proposition 9 *If the mask associated with a value A is being extended, then there is no possiblity that A need be moved further up the tree.*

Proof: By Contradiction:

1. If A must be moved up the tree, then there is a value B higher in the tree, such that $V_B \wedge M_A = V_A$ on the new M_A, but not under the old M_A.

2. Denoting the old mask for A as O_A, we know that under the old mask, $V_B \wedge O_A \neq V_A$

3. Given that V_A and V_B are the same in both expressions, the difference must lie in bits which are in V_B and M_A but not O_A. However, there are no such bits in V_A, hence there can be no equality.

Proposition 10 *When inserting a new node, using the up-pointer from the relevant subtree of a parent is equivalent to inserting the new node into the tree without said subtee.*

Proof: This property is actually a simple corrollary of proposition 1. Given that the up pointer in a subtree is either a pointer to the tree root or the unique up pointer (as the proof of the proposition shows), this up pointer is simply the pointer that started (or would have started) out as the pointer on this side of the root when it was the entire subtree. Therefore, using this pointer for comparison is equivalent to what the traversal would have found had there been no subtree.

4 Discontiguous Masking Patricia

Once one has started working with masked addresses, the obvious question is whether the masks need be contiguous and set at the most significant bit. There have been proposals suggesting utility in allowing noncontiguous masks for hierarchical network address management in some cases.

In looking at this alternative, we will be talking about a set of addressing entities, $\{A_i\}$, each consisting of an address value, V_i, and a mask, M_i. We will define what constitutes a valid set of nodes, and what consitutes a meaningful set of nodes.

4.1 Topologies of Masks and Addresses

In this section, we will discuss the meaning of masked address and will assume that our set of information consists of all possible information. This is distinct from the tree manipulation, described later, which must work on arbitrary subsets of this information.

In discussing the validity of address, one examines what entries match a target value. Matching here is defined the same way as for contiguous masked values, i.e. V match V_A under mask M_A if $V \wedge M_A = V_A$.

The other condition we talk about is one address being a parent of another. Specifically, A is a parent of B if $V_B \wedge M_A = V_A$ and M_A is a subset of M_B. That is, if the two entries match on the parents mask, and all bits from the parents mask are set in the child. (The subset criterion can be stated as $M_A \wedge M_B = M_A$.)

The first question is when a set $\{A_i\}$ is valid. The simple answer is that it is valid if, for any possible address value V, it is possible to define a uniquely correct node A. As with the contiguous case, it is permissible for a value to be matched by several entries. Even the selection criteria is the same (although the numeric simplification is not). If address V matches two nodes, A_i and A_j, then we wish to choose the one with the most (by count) bits set. However, unlike the contiguous case, the uniqueness of this selection is not automatically met. It is possible, if sufficient care is not taken in defining the addresses and masks, to have a value match several entries which have overlapping masks. It may then be the case that two (or

```
A: Mask=11001     A1: Mask=11111     A2: Mask=11111
   Addr=01xx0         Addr=01100         Addr=01010

B: Mask=10110     B1: Mask=11111     B2: Mask=11111
   Addr=1x01x         Addr=11010         Addr=10011

C: Mask=01101     C1: Mask=11111     C2: Mask=11111
   Addr=x11x1         Addr=11101         Addr=01111

D: Mask=01110     D1: Mask=11111     D2: Mask=11111
   Addr=x011x         Addr=10110         Addr=00110

E: Mask=10011     E1: Mask=11111     E2: Mask=11111
   Addr=1xx00         Addr=11000         Addr=10100
```

Figure 12: A set of addresses and masks

more) or the masks have the same number of bits set, and we would not be able to select a single best match. Thus, a basic criterion for validity is "don't do that".

The property as described above is not easily tested. There is another way to state the property. Given that the goal is to insure comparability of M_i when two V_i match a target V under their respective masks, the condition can be stated as a relationship among entries in the table.

Given two nodes A_i and A_j, either one is a parent of the other (i.e., $V_i \wedge M_j = V_j$ and M_j is a subset of M_i, or the obverse), or there is a bit set in both masks on which the two values differ (i.e. $V_i \wedge (M_i \wedge M_j) \neq V_j \wedge (M_i \wedge M_j)$). This would mean that no sought value could match both entries. Hence, if a candidate value matches two or more entries, they must be in a parent child relationship, and therefore there will be a unique matching entry with the most bits set in its mask.

This criterion will give a well formed set of addresses, with proper structural decidability. It follows that the addresses can be arranged into a containment tree, with child addresses under their parent. (Given that a child has a larger mask than a parent, no node can have two distinct parents. Each parent has a mask that is smaller than the childs, so if there were two parents, they would not have a bit in both masks on which they disagree.) The direct arrangement of children under parents produces an n-ary tree, since a given parent may have multiple children.

Given the logical tree of address containment, the next problem is to determine whether there is practical decidability among items. Given that one can compare entries only one at a time, and that binary trees are much simpler to handle then arbitrary n-ary trees, it is desirable to transform the containment tree into a binary one. Given that we are interested in patricia trees, based on individual bit testing, the question is how to make it practical to do that. Obviously, a set of addresses from the same mask can easily be arranged into a binary tree for bitwise testing. Unfortunately, it is possible to have multiple masked values such that one can not tell which order to test the bits. For example, on a five bit address, on might have the masks and addresses given in figure 12. (Here x indicates bits whose setting is irrelevant, as it is outside of the mask. These bits will be zero in a stored value, but they are irrelevant to the analysis.) In this example, no matter what bit one starts with, there are two different parents which do not have that bit in their mask. Specifically, they each have a child with each of the two possible values on that bit. Thus, one can not form a bitwise nesting tree with this set of addresses.

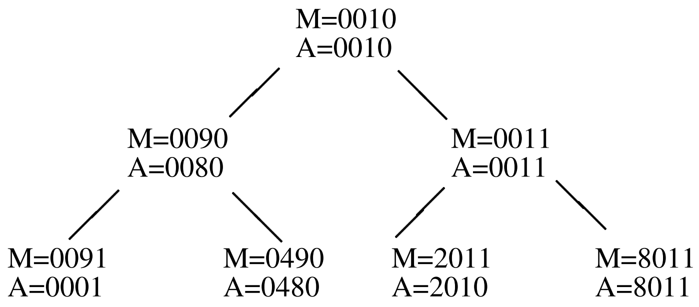


Figure 13: A tree of discontiguous masks

If one restricts slightly the way in which discontiguous addresses are used, it is practical and possible to use them. If one asserts that it must be possible to build a bitwise decidable tree, on is asserting that there must be a well defined order of bit significance. Not the same order for every address, but a well defined order. Thus, there is one bit that is most significant. If that bit is set, some specific other bit is the next most significant. If the most significant bit is not set, some bit (the same or different) is the next most significant bit. This does not imply that all intermediate masks must represent valid address, only that the represent valid significance. If one adds several bits to a mask at once, then one is simply asserting that independent of settings within that sequence, significance flows in the mathematical direction. This is illustrated in figure 13. (The masks and values are shown.) The ability to infer bit significance from the binary tree is critical in this application.

4.2 Discontiguous Algorithm

Given the description of the interpretation of discontiguous bit masks, one can see how to apply masked patricia to this case.

The basic concept is that the logic from masked patricia is applied, interpreting bits in the order of significance implied by the address tree.

One nice effect of this interpretation is that the lookup code does not change and so remains simple and efficient. The code is exactly the same as that shown in figure 5. This is a reflection of the fact that all the complexity of handling discontiguous masks is analysed and resolved at address insertion time.

THerefore, the insertion has several related problems to deal with. All of these are caused by the need for the table to be built with insufficient information. If one started with the entire address space of information, it would be trivial to insert information properly. However, one does not. So, the first problem is what order to test bit significance.

It turns out that using the information inherent in the masks one has, plus a property already proven, one can easily get the correct results.

It was shown earlier that when a value is inserted, all the bits tested in reaching it must be within our mask. Therefore, when inserting a value in the tree, we can infer that we have gone too far if the bit being tested falls outside the mask being added. One simply adds the new value into the tree at the point where this was found.

Given a proper but incomplete patricia tree, one can infer some of the bit significance. Specifically, one can infer the relevant bit significance to insure that all parent/child relationships are correct. Looking at the full address tree, in order of bit significance, one observes that the most significant bit occurs in *every* bit mask. In the side of the tree selected by that bit being set, *every* child has the next most significant bit in their mask. (in figure 13, 0010 is in every mask, 0080 is in every mask where the values do not have bit 0010 set, and 0001 is in the mask for all values which do have 0010 set.) Thus significance can be inferred from commonality of bits in masks.

So, in our incomplete patricia tree, we simply keep the mask of bits common to all children of this node. This mask may be equal to one's parent, or distinct. All bits added by the mask that were not in the parent's common mask are less significant than the bits in the common mask.

When one is inserting a node, therefore, one uses the sequence of common masks to infer the correct "most significant bit of difference". The case above, where a node is forced into position because the tested bit is outside its mask, represents a case where we have learned significant new information about the significance of bits in the mask.

The code to add these tests and interpretations is shown in Appendix A.

Note that in the earlier algorithms we used the bit mask information to indicate an up-pointer in the tree. Given that bits have significance in an almost random order, this is clearly not practical. Instead one must add a counter to the tree nodes, which will record the depth in the tree of the individual node. This can be used for detecting up-pointers, and must be updated whenever a new node is added to the tree. The routines `Walk_PostFix` and `Increase_pos` are used to do this.

The other item to note is that in some circumstances, particularly when adding a node at a spot because the tested bits are outside its mask, the inferring of new bit ordering may result in pushing the new node down in the rebuilt tree. This is detectable by the inability to select a bit already in the common portion to use for the new node. When this happens, the tree for rebuilding must be structured such that the rearrangements can occur.

By insuring that all tested bits are inside a mask, and by forcing a parent higher than a child if we find one, we automatically achieve the ordering required for the lookup algorithm. All properties proven earlier for masked patricia apply to this algorithm.

5 Further Work

There are some areas for further work in this algorithm. A sampling of these is given here.

It may be possible to improve the efficiency of the backtracking portion of the lookup algorithm by recording on the way down whether the candidate being sought matches the value in the node on the tested bit. Alternatively, the same gain may be had by examining the bit on the way back, before doing the full comparison. (Obviously, this efficiency only applies if bit tests are much faster than full entry comparisons.)

Complexity studies of the tree restructuring algorithm are of use. It is clear in the worst case that one has to rebuild the entire tree when a new entry which completely changes the bit significance inference is added. For perfomance analysis, the probability of needing to restructure a certain portion of the tree would be helpful. This depends on the model of address allocation and information spread which one uses.

Complexity studies of the cost of the backtracking, building on such work as [Szpan], would help determine what other efficiencies are needed, and what the true costs of this approach are.

The current structure works when the address space is bitwise decidable. It is possible (as shown in example 12) to create meaningful sets which are not bitwise decidable. To handle such cases would require node duplication. This is possible, but several questions arise. Primarily, the question of how to detect when a duplicate is needed, where the duplicate is to be inserted, and how often this would occur.

The algorithm presented here does not include a deletion routine. While the basic structure remains after deletion, deltions from even simple patricia trees are complicated. This algorithm neglects that because in many cases either no deletion is required, marking nodes as deleted is sufficient (due to the likelihood of reappearence), or marking as deleted until a restructure provides a time to remove a node is effective.

These possiblities suggest that deletion is not a necessary operation, but further analysis of how and when to perform deletion would be useful.

A Discontiguous Masked Patricia Insert

```
PNODE *P_Insert (addr, mask, ptree, recommend)
KEY_TYPE addr;
KEY_TYPE mask;
patricia_node *ptree;
BIT_SELECT recommend;
{
    patricia_node *n, *p, *pp, *t, *up, *pc, *splice;
    BIT_SELECT bit;
    KEY_TYPE prev_common;
    PNODE_STACK pm_stack;
    PNODE_STACK_PTR pm_stack_ptr, pm_trav;
    int clone_parent;

    /*
     * Apply the mask to the address
     * This should be an irrelevant operation, but protects against
     * incorrect calls where the address has bits set outside the mask.
     */

    addr = MASK_KEY(addr, mask);

    /*
     * Lookup the address.
     */

    PNODE_INITSTACK(pm_stack_ptr, pm_stack);
    t = ptree;

    do {
        p = t;
        PNODE_PUSH(p, pm_stack_ptr);
        t = (TEST_BIT(addr, t->bit)) ? t->right : t->left;
    } while (TEST_BIT(mask, t->bit) &&
             p->pos < t->pos /* t is deeper than p */ &&
             !EQ_KEY(MASK_KEY(t->key, mask), addr));

    if (EQ_KEY(addr, t->key) && TEST_BIT(mask, t->bit)) {

        /*
         * The new mask is usable without re-ordering anything.
         * It may change 'common' higher in the tree, but by the descent
         * procedure, it will not invalidate any checks.
         */
        t->mask = mask;
        prev_common = t->common;
        t->common = mask;
        if (t->left->pos > t->pos) /* There is a left subtree */
            t->common = MASK_KEY(t->common, t->left->common);
        if (t->right->pos > t->pos) /* There is a right subtree */
            t->common = MASK_KEY(t->common, t->right->common);
```

```
        n = t; /* for return */
    } else {
        if (t->pos <= p->pos) /* t is higher in the tree than p */
            up = t;
        else /* we need to find the up pointer */ {
            if (TEST_BIT(addr, p->bit)) /* we belong on p's right */ {
                if (TEST_BIT(p->key, p->bit)) /* which is p */ {
                    up = p;
                } else {
                        up = Find_Up (p->right, p->pos);
                }
            } else /* we go on the left */ {
                if (!TEST_BIT(p->key, p->bit)) /* which is p */ {
                    up = p;
                } else {
                        up = Find_Up (p->left, p->pos);
                }
            }
        }

        /*
         * Now find the most significant bit on which we differ
         * from the located node.
         * Significance is defined by the tree of common masks.
         */
        PNODE_STACKBOT(pm_trav, pm_stack);
        while (!PNODE_STACKTOP(pm_trav, pm_stack_ptr)) {
            prev_common = MASK_KEY(mask, PNODE_STACKGET(pm_trav)->common);
            if (!EQ_KEY(MASK_KEY(addr, prev_common),
                        MASK_KEY(up->addr, prev_common)))
                /* there is a bit in this mask on which we differ */
                break;
            PNODE_STACKNEXT(pm_trav);
        }

        if (!PNODE_STACKTOP(pm_trav, pm_stack_ptr)) {
            prev_common = PNODE_STACKGET(pm_trav)->common;
            clone_parent = 0;
        } else {
            prev_common = mask; /* use our mask */
            if (t->pm_pos > p->pm_pos) /* there is a down pointer */ {
                prev_common = MASK_KEY(prev_common, t->common);
                clone_parent = 0;
                /* now check whether we can use this */
                if (EQ_KEY(MASK_KEY(addr, prev_common),
                           MASK_KEY(up->addr, prev_common))) {
                    prev_common = mask;
                    clone_parent = 1;
                }
            } else {
                clone_parent = 1; /* probably irrelevant */
            }
        }
```

```
/*
 * Having found a mask under which addr and up->pm_addr differ,
 * now select a bit.  We use the easiest one to find.
 */
/*
 * We will first see if the recommended bit can be used, since
 * That presumably works well with the subtree.
 * If it is in the mask we have selected, then we can check to
 * see if it distinguishes the values.
 *
 * The recommended bit is used when re-inserting nodes, to try
 * to use the same bits as before, reducing the occurence of
 * recursive rebuilds.
 * These may occur because the discontiguous mask inferrence may
 * Take two steps to reach the final configuration.
 */
if (TEST_BIT(prev_common, recommend) &&
     (TEST_BIT(addr, recommend) != TEST_BIT(up->addr, recommend))) {
    bit = recommend;
} else {
    /* just find the first distinguishing bit */
    firstbit(&bit, prev_common); /*find first set bit */
    while (TEST_BIT(addr, bit) == TEST_BIT(up->key, bit)) {
        nextbit(&bit, prev_common); /* find next set bit */
    }
}


n = Create_Node(addr, mask, bit);

/*
 * Now figure out where to put the node.
 * Put it either under p, or above the first node whose
 * tested bit is not in prev_common.
 * prev_common defines the significance of our bit.
 */

pp = ptree;

do {
    t = pp;
    pp = TEST_BIT(addr, pp->bit) ? pp->right : pp->left;
} while (t != p /* not to bottom of our space */ &&
         TEST_BIT(prev_common, pp->bit));

/* now insert between t and pp */
n->pm_pos = t->pm_pos + 1;

/*
 * This preserves the common information we needed above,
 * for future use.
 */
```

```
if (pp->pos > t->pos&& !clone_parent)
    APPLY_MASK(n->common, pp->common);

if (pp->pos <= t->pos /* leaf insertion */ ||
    t != p /* simple significance */) {
    n->left = TEST_BIT(addr, n->bit) ? pp : n;
    n->right = TEST_BIT(addr, n->bit) ? n : pp;
    splice = n;
    if (pp->pos > t->pos /* middle insertion */)
        /* then increase the subtree depth */
        Walk_PostFix (pp, Increase_pos);
} else { /* there is a subtree, which we were forced above */
    /* we already found the up pointer from this tree */

    n->pm_left = TEST_BIT(addr, n->bit) ? up : n;
    n->pm_right = TEST_BIT(addr, n->bit) ? n : up;

    if (clone_parent) {
        pc = Alloc_Node();
        *pc = *t;
        pc->common = MASK_KEY(pc->common, n->common);
        if (pc->left == pp) {
            pc->left = n;
            Walk_Default(pp, Install_Node, pc);
            splice = pc->left;
        } else {
            pc->right = n;
            Walk_Default(pp, Install_Node, pc);
            splice = pc->right;
        }
        /*
         * The up pointer may have been moved and
         * Replaced with a pointer to pc.  Fix it.
         */
        (void *)replace_ptr(splice, pc, up);
        Free_Node(pc);
    } else {
        Walk_Default(pp, Install_Node, n);
        splice = n;
    }
}

/*
 * while addr may not be the address in splice,
 * it does select the correct side.
 */
if (TEST_BIT(addr, t->bit))
    t->right = splice;
else
    t->left = splice;

if (pp->pos > t->pos && t == p)
    Walk_Default(pp, Free_Node, NULL);
```

```
        prev_common = t->common;
        APPLY_MASK(t->common, n->common);

        /* clear all the lower stuff off the stack */
        while (!PNODE_STACKEMPTY(pm_stack_ptr, pm_stack)) {
            p = PNODE_POP(pm_stack_ptr);
            if (p == t) /* we have cleared t off the stack */
                break;
        }
    }

    if (!EQ_KEY(prev_common, t->common) &&
        !PNODE_STACKEMPTY(pm_stack_ptr, pm_stack)) {
        /*
         * common bits have changed,
         * so propogate the changes up the tree
         */
        if (SUBSET(t->common, prev_common)) { /* bits have shrunk */
            do {
                p = PNODE_POP(pm_stack_ptr);
                prev_common = p->common;
                p->common = MASK_KEY(p->common, t->common);
                t = p;
            } while (!PNODE_STACKEMPTY(pm_stack_ptr, pm_stack) &&
                     !EQ_KEY(prev_common, t->common));
        } else { /* bits have grown */
            do {
                p = PNODE_POP(pm_stack_ptr);
                prev_common = p->common;
                p->common = p->mask;
                p->common = MASK_KEY(p->common, t->common);
                if (TEST_BIT(t->key, p->bit)) { /* p is on right */
                    if (p->left->pos > p->pos) /* left is down */
                        p->common = MASK_KEY(p->common, p->left->common);
                } else if (p->right->pos > p->pos) /* right is down */
                    p->common = MASK_KEY(p->common, p->right->common);
                t = p;
            } while (!PNODE_STACKEMPTY(pm_stack_ptr, pm_stack) &&
                     !EQ_KEY(prev_common, t->common));
        }
    }

    return (n); /* success */
}
```

References

[Knuth] Donald E.Knuth *The Art of Computer Programming, Volume 3/Sorting and Searching*. Addison-Wesley Publishing Company, 1973.

[Morri] Donald R. Morrison *Journal of the Association for Computing Machinery*, vol. 15, pp. 514-534, 1968.

[Sedge] Robert Sedgewick *Algorithms in C* Addison-Wesley Publishing Company, 1990.

[Tsuch] Paul F. Tsuchiya *Efficient Routing Hierarchies Using Multiple Addresses* Bell Communications Research Paper

[Szpan] Wojciech Szpankowski "Patricia Tries Again Revisited" Journal of the Association for Computing Machinery, Vol. 37, No. 4, pp. 691-711, October 1990.

[VanJa] Van Jacobsen *new (?) routing lookup algorithm* Private Correstpondence via Electronic Mail, October 1988