3

Internet Protocol (IP) Address Lookup

CONTENTS

3.1 Introduction ...................................................... 48
3.2 Basic Binary Tree ................................................ 50
3.3 Disjoint Trie ...................................................... 52
  3.3.1 Procedure to Build a Disjoint Trie ...................... 53
3.4 Path-Compressed Trie ........................................... 54
3.5 Multibit Trie .................................................... 55
3.6 Level-Compressed Trie .......................................... 56
3.7 Lulea Algorithm .................................................. 58
  3.7.1 Number of Prefix Levels ................................. 59
  3.7.2 Representation of Prefixes as a Bit Vector ............. 59
    3.7.2.1 Code Field and Maptatable .......................... 62
    3.7.2.2 Code Field ........................................... 63
    3.7.2.3 Lulea Matching Process .............................. 65
3.8 DIR-24-8 Scheme ................................................ 66
3.9 Bloom Filter-Based Algorithm ................................ 67
  3.9.0.4 IP Address Lookup Using a Bloom Filter ............ 68
3.10 Helix: A Parallel-Search Lookup Scheme .................... 71
3.11 Ternary Content-Addressable Memory ........................ 75
3.12 Additional Readings ............................................. 78
3.13 Exercises ......................................................... 78

Finding out the egress port a packet traversing a network node, or router, requires performing the longest prefix matching between the destination of the packet to a prefix in the forwarding table. This process is called Internet Protocol (IP) address lookup, or just IP lookup. It requires to access this table to retrieve the next-hop information (NHI), which is the IP address or the router’s egress port number where a packet is forwarded.

The forwarding table resides in the memory of the router. Every time a packet arrives in a router, memory storing the forwarding table is accessed to perform IP lookup. Unfortunately, memory is a component whose speed lags in comparison to the rates at which data can be transmitted through (optical) network links. This speed mismatch requires to perform lookup using the smallest number of memory accesses. In addition, to improve feasibility in building such schemes, the amount of required memory must be imple-
mentable. Furthermore, IP lookup mechanisms should be able to accommodate changes of the forwarding table caused by changes in routing tables, in an expeditious manner so as not to interrupt the lookup process.

3.1 Introduction

Classless Interdomain Routing (CIDR) [67] is an approach used to aggregate IP addresses as prefix. This aggregation of addresses has the objective of reducing the number of entries in a memory holding a forwarding table of an Internet router. CIDR uses a technique called supernetting to summarize a range of addresses represented as the combination of a prefix and prefix length. The prefix includes a range of addresses that are reachable by a given router port. Figure 3.1 shows an example of address aggregation at a router.

In this figure, different networks, which are represented as ranges of addresses or prefixes, are reachable through different ports. Those addresses that share common most significant bits (MSBs) and are reachable through the same router port may be aggregated.

With CIDR, addresses can be aggregated at arbitrary levels, and IP lookup becomes a matching to the longest prefix. Specifically, a forwarding table consists of a set of IP routes. Each IP route is represented by a \(<\text{prefix/prefix length}>\) pair. The prefix length indicates the number of significant bits in the prefix, starting from the MSB. To perform IP lookup, the forwarding table is searched to obtain the longest matching prefix from among all the possible prefixes that match the destination IP address of an incoming packet. Then, the output port information for the longest matched prefix is retrieved and used for forwarding the packet. For example, a forwarding table may have three IP routes: \(<12.0.54.8/32>\), \(<12.0.54.0/24>\), and \(<12.0.0.0/16>\). If an incoming packet has the destination IP address \(<12.0.54.2>\), prefix \(<12.0.54.0/24>\) is the longest matched and its associated output port is retrieved and used for forwarding the packet.

Although CIDR resolves the aggregation of addresses to minimize the number of entries in a forwarding table (and in turn, in the memory of a router), the search for the matching prefix becomes more complex than performing exact matching (as used in label switching).

The complexity of address lookup is high because: 1) prefix lengths may be large and the adoption of IPv6 may increase these lengths even further, and 2) the number of prefixes is increasingly large for version 4 of the IP protocol, IPv4, and it is expected to grow with IPv6. In 1), the longest matching prefix is selected; so this selection requires a search for prefixes in all different lengths and to keep records of the longest prefix that last matched the destination address.
Prefixes have to be carefully stored and searched so as to minimize the number of times the memory storing the prefixes is accessed. The following example illustrates part of the search problem, whose complexity is affected by the relationship between the address of an entry and the entry content. Table 3.1 has eight entries (or rows) and each entry holds a number as content. Let’s consider the contents of the table as a 20-number set, where each can take a value between 0 and 19. Note that the range of values an entry can have is larger than the table size. Let’s also consider that no entry has a duplicate. Let’s assume that the entry contents are the set \{1, 4, 7, 9, 10, 13, 18, and 19\}, where the entries are stored in ascending order and are consecutively stored in the table, where 1 is stored in Entry 0, 4 in Entry 1, and so on. A simple strategy to find a desired number in the table would be to perform a linear search. For example, if we were to find number 15, we would go from location 0 to 6 (7 locations) and find out that 15 is not in the set. So the problem is to find a specific content in the smallest number of trials.

One could think of starting the search from different locations of the list (e.g., in the middle of the table) to reduce the number of accesses to the table. This problem exacerbates as the range of the content values or the number of entries grows. For example, in IP lookup, the contents could be found to be over half a million prefixes. This number is being reported by the time this book is written, but the number continues to increase [2].

Back in our example, we could sort the contents by selecting them based on the tens and the units of an entry, such that we could sort the numbers as a decimal tree where the first children would be from numbers from 0 to 1 (as root tree), and the children of those nodes would be numbers from 0 to 9. Note that if the set of numbers increases, so does the number of children in the root tree. We use here the terms tree or trie interchangeably in this chapter.
TABLE 3.1
Example of searching in a table with nonrelated set members.

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>19</td>
</tr>
</tbody>
</table>

TABLE 3.2
Example of prefixes in a forwarding table.

<table>
<thead>
<tr>
<th>Prefix ID</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>*</td>
</tr>
<tr>
<td>P2</td>
<td>1*</td>
</tr>
<tr>
<td>P3</td>
<td>00*</td>
</tr>
<tr>
<td>P4</td>
<td>0001*</td>
</tr>
<tr>
<td>P5</td>
<td>1000*</td>
</tr>
<tr>
<td>P6</td>
<td>10001*</td>
</tr>
<tr>
<td>P7</td>
<td>11111*</td>
</tr>
<tr>
<td>P8</td>
<td>111011*</td>
</tr>
<tr>
<td>P9</td>
<td>100000*</td>
</tr>
<tr>
<td>P10</td>
<td>00010000*</td>
</tr>
</tbody>
</table>

Table 3.2 shows an example of a forwarding table. This table has 10 prefixes, with prefix lengths between 1 and 7 bits. Prefixes are represented by a string of 0s and 1s, marked at the end (right-most side) with a star mark (*). For instance, prefix 128/1 is represented as 1*.

3.2 Basic Binary Tree

In a simple form, prefixes can be stored in a binary tree representation (using 32 bits for IPv4 and 128 bits for IPv6). This figure shows the binary tree of the forwarding table (Table 3.2), where the 0 child is at the left side of the root and the 1 child is at the right. The figure shows the position of the 10 prefixes in the binary tree, where prefix P1 has prefix length 0 (i.e., root of the tree), or it is positioned on level 0, and prefix P10 is on level 7.

1Tree structures are covered in detail in [43].
The prefixes in the tree are stored, starting from the MSB to the least-significant bit (LSB). Furthermore, because we need to match any of the prefixes in the forwarding table, all entries are incorporated into the tree. Therefore, prefixes with common bits (bits with the same value at the same significant position) are shared. For example, prefixes 110* and 111* share the first two bits (11-) and the only different bit is their LSBs. So these two bits are represented as 11 linked to 0 for the first prefix and to 1 for the second. As this example shows, each position bit has only two possible combinations, or binary children. The default prefix, *, is the ancestor of all other prefixes (i.e., all address ranges overlap with the range represented by *). Therefore, * is the root of the tree.

Figure 3.2 shows the binary tree representation of Table 3.2. Each time a packet arrives, the destination of the packet is compared bit to bit in the path from the root (*), or MSB, to a leaf, or LSB, of the tree. The matching prefix is the longest of those that matches the destination address of the packet.

**FIGURE 3.2**
Binary tree of the prefixes in Table 3.2.

The binary tree of this table is stored in memory. Table 3.3 shows a simple implementation of the binary tree in Figure 3.2. In this representation, each memory location is assigned to a tree node (where the addresses of the table are the node numbers in the tree, not shown in the figure). A node may be represented with two children, the 0- and 1-child, as pointers. A node may be a parent node or prefix. If the node is a prefix a field for the prefix ID or NHI is also stored in that node. The memory address 0 stores the root of the tree. Each entry holds a flag indicating if the entry is a prefix or a non-prefix node, the Id of the node (or NHI), and two pointers to the address of the children of the node (children 0 and 1).

The table stores the 22 nodes of the tree. However, this table is built as a 32-entry (i.e., \(2^5\) memory block as the number of entry of memories come in
power-of-two number of entries) table, the number of entries is equal to the number of tree nodes. The memory used to represent prefixes of any length must be of a feasible size (as fast memories, such as Static Random-Access Memory, SRAM, are small). SRAM is a suitable candidate memory technology for building lookup tables as it is desirable to perform IP lookup in the smallest possible packet inter-arrival time.

### Table 3.3
An example of memory format storing the binary tree of Figure 3.2.

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Prefix flag</th>
<th>Prefix Id (NHId)</th>
<th>0-child</th>
<th>1-child</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>P1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>P2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>P3</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>-</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>-</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>-</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>P4</td>
<td>13</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>P5</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>17</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>-</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>-</td>
<td>19</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>P6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>20</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>P7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>21</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>P9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>P8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>P10</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The matching process in a binary tree may require as many memory accesses as the longest stored prefix. This is up to 32 memory accesses for IPv4 and 128 for IPv6. The matching complexity is said to be in the order of $O(W)$, where $W$ is the prefix length.

### 3.3 Disjoint Trie

The binary tree requires remembering the last matched prefix along the search for the longest prefix in the traverse from the root to the leaves. A register
may be common to keep this temporary record. This additional memory requirement may be avoided if a disjoint trie is used instead. The disjoint trie has the property that each node has either two children or none.

Figure 3.3 shows the disjoint trie of the binary tree in Figure 3.2. As the figure shows, prefixes of a disjoint tree are at the leaves. Therefore, each time a packet arrives in a router, the tree is examined, started from the root (*) and continuing towards the matching leaves, and the visited leaf holds the longest matching prefix. This tree uses a number of nodes equal to or larger than that of a regular binary tree, but it simplifies the matching process by matching a single prefix.

![Forwarding Table](image)

**FIGURE 3.3**
Disjoint binary tree from prefixes in Figure 3.2.

### 3.3.1 Procedure to Build a Disjoint Trie

The disjoint trie is built by completing a binary tree. This process consists of making each node have two children or no children. This means that a node that has one child has to be completed with another child. Such an added node may be either a prefix or an ancestor node of a prefix (or a subtree with a prefix). In such cases, the new added child will hold the prefix of the closest ancestor prefix. This technique is also called *leaf pushing* [161].

For example, let’s assume that the branch of a trie has the following two prefixes: $P_x$ 0010* and $P_y$ 00100* where 00100* is the only child (0 child) of 0010*. Therefore, 0010* must be completed by adding 1 child, or 00101*. Then 0010 is pushed from level four to level five of the trie as $P_x$ 00101*, and 0010 becomes an ancestor node (not a prefix) of these two prefixes that has two children.
The search complexity of matching on a disjoint tree is equivalent to that of a binary tree, this is, the worst-case scenario is 32 bits for IPv4 and 128 bits for IPv6. The prefix search complexity remains at $O(W)$. The amount of memory used for this trie is larger than the memory needed for a binary tree as the number of nodes increases as the tree is completed, but matching is achieved by reaching of the trie leaves as the trie no longer has internal prefixes (that could be matched).

### 3.4 Path-Compressed Trie

Although the comparison of bit by bit in a binary tree is faster than a linear search in IP lookup, the worst case scenario still takes time; 32 memory accesses for IPv4. The path-compressed trie [117], also called Patricia trie, improves the search for a prefix by comparing not one but many bits at a time. The number of compared bits may be variable in this trie; the number compared at once is equal to the number of compressed bits for a node. This technique is called path compression. In the Patricia trie, unbifurcated branches of a tree may be compressed. For example, if a forwarding table contains prefixes $1^*$ and $111^*$, prefix $111^*$ can be represented as a compressed prefix $11^*$ with 1 bit compressed (indicated by the number of skipped bits, or skip field, or 1 bit) and by the value of the skipped bits, or segment="1" (the actual string of compressed bits). Figure 3.4 shows the Patricia trie of the binary tree in Figure 3.2. The compressed trie in this figure shows the same set of prefixes but in a smaller trie (i.e., a trie with a shorter height and a smaller number of nodes). However, the compressed bits and information indicating the compression must be stored for a node, indicating the skip and segment fields. In general, an entry of a Patricia trie uses additional space to store these two fields. That is, while the number of nodes is smaller (therefore, requiring a shorter memory), the memory required is wider than that of a binary trie. Reducing the tree height decreases the number of needed memory accesses.

In terms of performance, the Patricia trie performs lookup with a smaller number of memory accesses, on the average, than a binary trie. But in the worst-case scenario, the number of memory accesses may be equal to that of a binary trie. This worst-case scenario occurs when the path to the longest prefix in the table has bifurcations (two children) at each node. However, the possibility of this to occur may be small and may not occur in all possible paths.
### 3.5 Multibit Trie

Nodes of a trie can be compressed with a large number of bits, as the Patricia trie does, but rather using a constant number of bits, called *stride*. The multibit trie approach avoids the complexity of identifying the skip size and segment for each node. This approach also reduces the height of the trie by assigning a stride larger than one bit, and therefore, the search for a matching prefix would use a number of bits, equal to the stride size, at a time. In turn, the number of memory accesses is reduced. This approach is called a multibit trie [159]. In such a trie, prefixes are expanded to the levels that are proportional to the stride size, as Figure 3.5 shows it as a *n*-ary trie.

![Multibit Trie Diagram](image)

**FIGURE 3.5**
Multibit trie with a three-bit and constant stride.

The multibit trie may also be represented as a set of memory blocks, each holding all the children of a node. Figures 3.5 and 3.6 show an example of a
3-bit stride multibit trie. In these figures, the prefixes are pushed to the leaves when they fall into intermediate nodes. This approach is called multibit trie with leaf pushing. As the trie has a height of 7, the trie has three 3-bit levels, and prefixes have to be expanded to levels that are proportional to three: 3, 6, and 9.

**Speed and memory amount.** The multibit trie is faster than a binary tree for performing IP lookup. For strides of up to $k$ bits, the trie’s height is reduced from $W$, which is 32 bits for IPv4 (and 128 for IPv6), to $\lceil \frac{W}{k} \rceil$. Therefore, the search complexity is $O(\frac{W}{k})$, where $k$ can also be considered as the speedup factor. In this approach, the amount of memory used to store all prefixes of a forwarding table is larger than that used in a binary tree, as per the used prefix expansion, where in the worst case, a node is represented by $2^k$ children.

### 3.6 Level-Compressed Trie

The level-compressed trie [124], also known as LC trie, reduces a tree height and, therefore, the number of memory accesses needed to perform IP lookup. In addition, it reduces the required amount of memory to store the prefixes, in comparison with a multibit trie. The LC trie uses a combination of properties from the path-compressed, multibit trie, and disjoint tries. In fact, tree completion (as in a disjoint trie), bit expansion (as in a multibit trie), and path compression (as in a Patricia tree) techniques may be used to build an LC trie. This trie is a variable-stride multibit tree. It reduces the amount needed to build a tree by reducing unnecessary prefix expansions, by using variable stride sizes and path compression.

Figures 3.7, 3.8, and 3.9 show three steps in which an LC trie is built. First, a tree is completed, as Figure 3.7 shows.

Then, a stride size is selected to include a large number of prefixes at the selected prefix level (Figure 3.8). Once the stride is selected, the node is represented as a multibit trie.

After that step, the next compressed path is selected, as a new stride or as path compression. The main objective may be to reduce the tree height or memory. These steps are performed in the remaining uncompressed parts of the tree. Subtrees are compressed as a multibit or as a path-compressed trie separately from the compression of other subtrees. Figure 3.9 shows the resulting LC trie, where the stride size for the subtrees is three. Note that a stride size of four for the subtree at the (bottom) left would have reduced the number of memory accesses by one. However, at the expense using of more memory.

The largest number of memory accesses needed for this example tree is
FIGURE 3.6
Multibit trie representation. Each node has $2^{\text{stride}}$ children.
four as the root of the tree may need to be accessed for retrieving the stride size.

**Speed and memory amount.** The LC trie is very effective at reducing both memory size and number of memory accesses. As in the multibit trie, the stride is the number of bits that can be compared at once. The speed in which IP lookup is achieved depends on the stride sizes used. The lower the level where the prefixes are placed, the longer the stride can be achieved. The lookup complexity of the LC trie is similar to that of the multibit trie.

### 3.7 Lulea Algorithm

The Lulea algorithm, named after the university where it was developed, is a data structure that minimizes the amount of memory used to store NHIs of a forwarding table [49]. Lulea follows the representation of prefixes of a multibit trie but with fewer prefix levels and with a reduced number of replicated prefixes produced by expansion (i.e., leaf pushing). Lulea uses a codeword array, base index array, maptable, and NHI table. Each of these parts can be considered a memory block. The three first parts are used to determine which entry of the NHI table holds the matching prefix.

Lulea uses several strategies to reduce the memory amount: 1) a small number of prefix lengths (e.g., three in the original algorithm), 2) a prefix count, and 3) representing the leaves of a tree produced by expansion of prefixes, as disjoint subtrees, in a small number of combinations. Each of these strategies is described next.
3.7.1 Number of Prefix Levels

The Lulea algorithm is one of the first schemes that aims to represent all prefixes on a routing table on a few prefix levels. The selected levels are 16, 24, and 32. Any prefix with a prefix length smaller than 16 is expanded to that level, prefixes with prefix lengths between 17 and 23 are expanded to level 24, and similarly, prefixes with prefix lengths between 25 and 31 are expanded to level 32. The main level in the scheme is level 16. Levels 24 and 32 are accessed after level 16 is accessed. Figure 3.10 shows a general representation of the relocation of prefixes in the Lulea algorithm, for IPv4 tables. The description focuses on describing level 16 as this is used to build the maptable, which is reused for matching at levels 24 and 32.

3.7.2 Representation of Prefixes as a Bit Vector

To build the codeword and base-index arrays, and the contents of the maptable, prefixes in level 16 are represented as a bit vector. The bit vector is a string of bits, whose position represents the prefix value. The objective of the bit vector is to show the number of prefixes needed to be stored in the NHI. The bitmap has $2^{16}$ bits, numbered from bit 0 to bit $2^{16} - 1$. Here, each bit is dedicated to indicate whether its position holds a new prefix or not. For example, prefix 1000 0000 0000 0000* in binary, or 128.0/16 in decimal, occupies position $2^{15}$ in the bitmap with $2^{16}$ bits. This bit vector then represents prefixes from 0.0/16, as the bit in position 1, to prefix 255.255/16, as the bit in position $2^{15}$.

A bit in the bit vector is the leaf of the binary tree built with the prefixes
with lengths up to level 16. Figure 3.11 shows an example of how to set the bit values in the bit vector. This subtree has four levels, so it has 16 leaves; bits 0 to 15. Here, bit 0 is set to 1 as it is the first bit of a prefix expanded to four leaves. Bits 1 to 3 are also expansions of the same prefix, but since they are a duplicate of the prefix in bit 0, they are set to 0. Similar case is presented in bits 4 and 5. Because these two bits are expansions of the same prefix, bit 4 is set to 1 and bit 5 to 0. Bits 6, 12, and 13 are not expansions of a prefix but they are ancestors of prefixes, which have a longer prefix length. These bits are also set to 1. Therefore, nodes with no prefix and no descendants, as nodes that hold a duplicated prefix, are set to 0.

The number of bits set to 1 in the bit vector equal to the number of prefixes in the NHI table. By setting bits of duplicated prefixes to 0, the number of prefixes in the NHI table is kept small. For example, considering prefix length of four, if bits nodes 0001, 0010, and 0011 belong to the same prefix, the nodes are represented by the bits 1, 0, and 0, respectively (instead of representing them as 1, 1, and 1, respectively, as in a multibit trie). In this way, once a prefix, a bit of the bitmap, is accessed for lookup, the position of that prefix in the NHI table is addressed by the number of 1’s in the bitmap of the bits with a position smaller than or equal to the matching prefix.

Figure 3.11 shows an example of the representation of nodes in a bitmap.

It should be clear now that what Lulea does is to count the number of ones from bit 0 to the bit $x$, where $x$ corresponds to the matching prefix at level 16 in this case. However, storing the bit vector as described may not be efficient for counting all 1s, as multiple memory accesses would be needed. To overcome this issue, the scheme uses a codeword, base index array, and a maptable to store information about the bit-vector, and in turn, for the lookup process.
Prefixes are found at different levels and they can be projected onto the larger selected prefix level.

The bit vector is segmented into groups of small and equal number of bits, or *bit-masks*. The original Lulea algorithm uses 16-bit bit-masks. Therefore, level 16 comprises $2^{16} = 4096$ bit-masks. A codeword uses the same number of bits as a chunk and comprises two fields: 1) an *offset* and 2) a combination *code*. Therefore, the number of codewords in the codeword array is 4096, where each codeword corresponds to one and only one bit-mask. The offset field of codeword $i$ indicates the number of 1s in bit-masks $i - 3$, $i - 2$, and $i - 1$, where $i$ is the bit-mask of the matching (prefix) bit. The offset records the accumulated number of ones in each set of four bit-masks. The offset of bit-mask 0 is 0, as the offset is 6-bit wide. However, the count of 1s must include all those from bit-mask 0 to bit-mask 2 (three bit-masks) in the set of four. The counting of 1s in the last bit-mask is stored in the maptable.

To add the 1s up to bit-mask $2^{12} - 2$, a *base index* is used. Figure 3.13 shows an example of the use of the code, offset, and bit index. The length, in number of bits, of the base index is calculated by considering the largest number of 1s that may accumulate for the last possible bit-mask in the bit vector, or $2^{16} - 16$ 1s for level 16. Therefore, the base index is 16 bits long (i.e., $\lceil \log_2(2^{16} - 16) \rceil$). Because offset counts the number of ones in three bit-masks, a base index is allocated for each set of four consecutive bit-masks (this reduces the number of base indexes). Therefore, the base index array has $\frac{2^{16}}{4} = 1024$ base indexes. Therefore, the MSB 12 bits of a packet destination address are used to access a codeword and 10 bits for a base index.

Figure 3.13 shows the codeword and base index arrays.

As an example, let’s consider that a packet address matches the bit position pointer by the arrow in Figure 3.12. The arrow points to the bit corresponding to prefix 00000000 01011101* (or bit 92), which is in Bit-mask 5 (the sixth bit-mask from the left). Therefore, base index 1 counts the number of 1s in Bit-masks 0 to 3, offset 5 counts the number of 1s in Bit-mask 4, and code
in Bit-mask 5 counts the number of 1s from the beginning of the bit-mask to the bit corresponding to the prefix. The total number of ones is 22, which corresponds to prefix v.

### 3.7.2.1 Code Field and Maptable

The code field indicates the combination and position of the 1s in a bit-mask. The code is used to address an entry in the maptable. This entry provides this information by showing the accumulated number of 1s for each bit position in the corresponding bit-mask. Figure 3.14 shows the maptable for our example. The table has as many rows as the number of different codes and as many
columns as the number of bits in a bit-mask. Therefore, the code and the four LSB bits in a packet’s destination address are used to address the rows and columns of the table, respectively. In general, the \( \log_2(\text{bit-mask size}) \) LSB bits (e.g., 4 bits for level 16) of the string of packet address bits used for lookup (e.g., 16 bits for level 16) are used to address the maptable column.

![FIGURE 3.12](image)

**FIGURE 3.12**
Example of the values of offset and base index fields.

Table 3.4 shows the NHI table used in Lulea to identify the matching prefix. The number of prefixes (and entries) in this table is equal to the number of 1s in the bitmap.

**3.7.2.2 Code Field**

The authors of the Lulea algorithm found that the number of combinations in a 16-bit bit-mask is smaller than \( 2^{16} \). As the expansion of prefixes is similar to completing a tree, where each node has either two or no children (see the disjoint trie), the number of actual combinations that can be produced in prefix expansion is smaller than \( 2^l \), where \( l \) is the prefix level. Specifically, the number of combinations are

![FIGURE 3.14](image)

**FIGURE 3.14**
Example of maptable for bitmap of Figure 3.12.

Table 3.4 shows the NHI table used in Lulea to identify the matching prefix. The number of prefixes (and entries) in this table is equal to the number of 1s in the bitmap.
for bit-masks with $2^n$ bits. For example, if the bit-mask is 16-bit wide, $n = 4$ as $2^4 = 16$, and $a(4) = 677$ (an improvement to reduce this number of codes has been considered [49]). After considering the all-0s combination ($677 + 1$), to represent 678 combinations, we need only 10 bits ($\lceil\log_2(678)\rceil = 10$) instead of the original 16 bits. Then, each combination of 0s and 1s in a bit-mask with 16 bits is represented as one of the $2^{10}$ combinations. This shows that the code provides information of the combination of 1s and 0s in a bit-mask but with a smaller number of bit than those in the bit-mask. In general, the size of the code, $|code| = \lceil\log_2(a(n))\rceil$, in number of bits.

Figure 3.15 shows the combinations of complete tries with $n = 1$ and 2. These examples shows that the possible combinations are only two and five, respectively.
3.7.2.3 Lulea Matching Process

The lookup process in Lulea uses the destination IP address to identify the corresponding NHI in the NHI table. For this, the MSB 12 bits of the address are used to find the corresponding bit-mask and the MSB 10 bits for accessing the corresponding base index. The LSB 4 bits of the level (16 in this case) are used as partial address to the maptable. Figure 3.16 shows the a summary of the bits needed from the packet destination address to access the code and NHI tables.

For example, let’s consider IP lookup is performed for a packet with des-
Interconnections for Computer Communications and Packet Networks

66

... in the bit vector used as an example (Figure 3.13). Using the MSB 16 bits part of the address, 0.72, matches Codeword 5 and Base index 1. Codeword 5 has offset = 0 (as it is the first bit-mask of a set of four), the code is r4 (or row 4 of the maptable), and base index = 14. The maptable entry addressed by row 4 and column 9 stores the number 2, so the address in NHI=0+14+2=16. The reader can count the number of 1s to where the *Example* arrow in Figure 3.16 and this is the same number. The 16th NHI prefix is matched and it may provide the next hop address for this packet. This prefix corresponds to prefix p in Table 3.4 and Figure 3.12.

For level 16, Lulea performs IP lookup in three memory accesses (where the codeword and base index arrays are accessed simultaneously). A similar process is performed for other larger prefix levels. A corresponding number of memory accesses is added to the lookup time if the matches continues to the other levels. Forwarding table updates are costly in this scheme, as codewords, base indexes, maptable, and NHI table may need to be all updated.

### 3.8 DIR-24-8 Scheme

The DIR-24-8 scheme aims to represent prefixes in two levels (or lengths) [75], similar to the Lulea algorithm. This scheme performs controlled prefix expansion to represent all prefixes on either level 24 or 32. The adoption of two levels greatly simplifies the implementation of the IP lookup engine. The scheme uses a memory block for each level. Level 24 uses a memory large enough to represent all possible prefixes. All prefixes with length smaller than 24 are expanded to that level. Prefixes between levels 25 and 31 are expanded to level 32. As the number of prefixes at level 32 is too vast, only a few 256-address intervals are represented. Each of these intervals, or subtries, are the descendant of a prefix at levels 25 to 30. Figure 3.17 shows the idea of representing these intervals at level 32. The total number of intervals at level 32 depends on the number of prefixes larger than 24 bits in the forwarding table. Because it is difficult to estimate the memory needed for an unknown number of prefixes, the level-32 memory is given a fixed size, and therefore, the number of prefixes is limited to those that fit in that memory size.

DIR-24-8 uses two memory blocks, called TBL24 and TBLLong, where the first one stores prefixes on level 24, or pointers directing the search to level 32, and TBLLong stores prefixes on level 32. Figure 3.18 shows these two memories and the supporting logic. An entry (memory location) holds a prefix bit to indicate whether the entry is a prefix. If the bit is set to one, the stored information is the NHI; otherwise it is a pointer to TBLLong.

**Speed and memory amount.** DIR-24-8 performs IP lookup in up to two memory accesses (if the prefix is not matched at TBL24, it is matched in TBLLong). The lookup complexity of DIR-24-8 is O(1) as the search on TBL24 is
FIGURE 3.17
Prefix levels in DIR24-8 scheme and the prefix intervals created by prefix expansion.

A single memory access, where the 24 MSB bits of a packet destination address is used as memory address, and the process is similar in TBLlong, however, using instead 32 bits. The number of maximum number of pointers is $2^{12}$ and it may need to copy several replicates of NHI for expanded prefixes. The table update is simple as the small number of levels simplifies the identification of entries to be updated but it may require accessing several entries (as a by product of prefix expansion). As can be deduced from Figure 3.18, the total memory amount for this scheme is $(2^{24} + 2^{20})W$, where $W$ is the number of bits used in an entry. The entry stores the prefix bit and the NHI. Figure 3.19 shows an example for interval 123, to where some prefixes are expanded.

3.9 Bloom Filter-Based Algorithm

Bloom Filter Theory. A Bloom filter is a space-efficient data structure for membership queries [19] and is widely used in various applications because of its compactness and simplicity. A Bloom filter is an $m$-bit array that contains abstract information about a set. For a given set $S = \{x_1, x_2, \ldots, x_n\}$, programming a Bloom filter starts from initializing every bit of the filter to zero. For element $x_j$, where $1 \leq j \leq n$, $k$ hash indices $h_i(x_j)$ for $1 \leq i \leq k$ are obtained, where $0 \leq h_i(x_j) < m$. To program element $x_j$ into a Bloom filter, the $k$ bits indicated by $h_i(x_j)$ are set to one. This procedure is repeated for every element included in the set $S$. 

[Diagram of IP Address Lookup and Bloom Filter Concept]

<Diagram>
Querying for the membership of a given input $y$ also uses the same $k$ hash functions, $h_i(y)$ for $1 \leq i \leq k$. The $k$ bits indicated by the hash indices are checked. If all $k$ bits are one, $y$ is considered a member of $S$. If any of the $k$ bits are zero, $y$ is definitely not a member of $S$.

Bloom filters may have false positives, but no false negatives. For $n$ elements, the false-positive rate $f$ of an $m$-bit Bloom filter is calculated as Equation 3.2 [118].

$$f = \left[ 1 - \left( 1 - \frac{1}{m} \right)^{kn} \right]^k \approx \left( 1 - e^{-kn/m} \right)^k \quad (3.2)$$

The optimal number of hash functions for an $m$-bit Bloom filter of $n$ elements can be calculated as follows [118]:

$$k_{opt} = \frac{m}{n} \ln 2 \quad (3.3)$$

Figure 3.20 shows an example of a Bloom filter. The Bloom filter is programmed by $S = \{x_1, x_2, x_3, x_4\}$, and queried by inputs $y$ and $z$. The Bloom filter produces a negative result for $y$ and a false-positive result for $z$.

3.9.0.4 IP Address Lookup Using a Bloom Filter

In a binary trie, a node cannot exist without ancestor nodes, except for the root. This principle is used by a Bloom filter to improve the performance in a
Internet Protocol (IP) Address Lookup

FIGURE 3.19
Example of interval 123 with expanded prefixes on TBLlong.

FIGURE 3.20
An example of a Bloom filter.

search on a binary trie [118]. The Bloom filter may be implemented using on-chip memory, while a trie is usually implemented in off-chip memory because of its larger memory requirements. The Bloom filter is used to reduce the number of off-chip memory accesses by determining the best matching level holding the longest matching prefix for a given packet destination address.

In this approach, nodes of a binary trie are programmed in an on-chip Bloom filter and stored in an off-chip hash table. The Bloom filter is queried, while linearly increasing the access level, until a negative result is produced. The negative result means that there is no node detected in the current and
longer levels. Therefore, the level of the last positive outcome is the candidate level storing the best matching prefix. Using the prefix with a length equal to the candidate level, the off-chip hash table is probed to verify the result. Because the search procedure of this approach starts from the candidate level, the matching process may finish right after the matching prefix is found. If a matching node is not found because of the Bloom filter’s false positive, the search procedure goes to a smaller level. This process is called back-tracking.

The number of back-tracking processes depends on the rate of false positives. Assuming that the best matching prefix is on level $L$, if a false positive at level $L+1$ and a negative at level $L+2$ are produced, a single back-tracking process takes place. Similarly, if false positives on levels $L+1$ and $L+2$ and a negative outcome occur at level $L+3$, two back-tracking processes take place. Hence, the probability of the number of back-tracking processes are calculated by Equation 3.2, where $b_j$ represents that back-tracking has occurred $j$ times and $f_i$ is the false-positive rate of a node in level $i$.

$$\text{Prob}(b_j) = \prod_{i=L+1}^{L+j} f_i$$

(3.4)

The probability that multiple back-tracking processes occur rapidly decreases because this probability is the product of the false-positive probability for each level. However, even though a positive result occurs, the positive result does not guarantee that the best matching prefix is on the level, because empty internal nodes may exist. If an internal node is encountered, the back-tracking process also takes place. Precomputing the best matching prefix, and storing the corresponding information for each internal node can easily solve this problem. Figure 3.21 shows the IP address lookup algorithm using a Bloom filter for the example forwarding table in Table 3.5. It is assumed that each gray-colored node in Figure 3.21 inherits the longest matching prefix from its direct ancestor prefix node.

### TABLE 3.5

Example of a forwarding table.

<table>
<thead>
<tr>
<th>IP prefix</th>
<th>prefix ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>00*</td>
<td>P0</td>
</tr>
<tr>
<td>010*</td>
<td>P1</td>
</tr>
<tr>
<td>1*</td>
<td>P2</td>
</tr>
<tr>
<td>110101*</td>
<td>P3</td>
</tr>
<tr>
<td>1101*</td>
<td>P4</td>
</tr>
<tr>
<td>111*</td>
<td>P5</td>
</tr>
<tr>
<td>11111*</td>
<td>P6</td>
</tr>
</tbody>
</table>

Note that a hash entry is accessed either for the level with the last positive result when the Bloom filter produces a negative or by a back-tracking process.
when the Bloom filter produces a false positive. If a node at level $L$ has both children, the result is neither a negative nor a false positive at level $L + 1$. Hence internal nodes with both children (denoted with dotted lines in Figure 3.21) are not necessarily stored in the hash table, while every node in a trie is programmed in the Bloom filter.

Algorithm 1 describes the operation of this lookup scheme. For example, the search procedure for the input address of 110100 is as follows: The Bloom filter is queried for lengths 1 through 5 and it produces positive results. At level 6, the Bloom filter produces a negative result. Hence the hash table is accessed at level 5 using the prefix of 11010, which is the level of the last positive result, and the search is over by returning the precomputed longest matching prefix $P4$.

![Figure 3.21](image)

**Figure 3.21**
Improving trie search using a Bloom filter.

### 3.10 Helix: A Parallel-Search Lookup Scheme

As discussed in the previous sections, IP lookup schemes based on binary trees require multiple memory accesses and that makes it challenging to use them in implementations for routers with high-speed ports. To overcome this, Helix is a scheme that performs both IP lookup and table updates for IPv4 and IPv6 prefixes in a single memory access [145]. Helix achieves this lookup speed using very small amounts of memory. Helix uses a parallel prefix search [49, 140, 191] at the different prefix lengths and the helicoidal properties of binary trees. These properties are based on observations that if a binary tree
Algorithm 1 Address Lookup Procedure in Bloom Filter-based Approach

Function: Search dstAddr

1. find the BML using Bloom filter
2. for $i \leftarrow \text{shortestLen to longestLen}$ do
   1. if $\text{queryBF}$(dstAddr, $i$) is positive then
      1. $\text{BML} \leftarrow i$;
      2. else
         1. break;
   2. end if
3. end for
4. access hash table
5. for $i \leftarrow \text{BML to shortestLen}$ do
   1. $\text{node} \leftarrow \text{queryHT}$(dstAddr, $i$);
   2. if $\text{node exists}$ then
      1. return node.info;
   3. else
      1. continue; // back tracking
   4. end if
6. end for

is considered with plasticity features, it could be “folded” to make a smaller tree. The helicoidal properties of a binary tree allow folding the tree (i.e., information is not lost after the tree is folded).

Parallel search means here that Helix searches for a matching prefix in all different prefix tables, simultaneously. Parallel search has been of interest in several schemes [49, 140]. Among the matched prefixes at the different levels, the longest prefix is selected.

Figure 3.22 shows the Helix’s lookup process for IPv4 prefixes, as an example. In this figure, there is a prefix table for each nonempty prefix level. A matched prefix, if any, is output by the corresponding level table(s), and the selector selects the longest matching prefix.

A nonempty prefix table is one that has one or more prefixes. Prefixes are kept in their original levels; prefix lengths remain unchanged. In this scheme, there is a prefix table for each nonempty prefix level, where each table is implemented as a memory block. Note that keeping the prefixes in their original length has the following benefits: 1) the number of prefixes in the binary tree does not increase and 2) it is simple to make table updates.

Small tree levels may be stored without any change (e.g., level 8 may be a memory with 256 entries and the level-8 prefixes are stored as they appear in a routing table). Large levels may need to be processed further.

A forwarding table has as many different NHIs as the number of the ports, $k$, of the router hosting the table. Table 3.2 shows an example of a forwarding table, which is used as an example throughout this section to describe the proposed scheme and the helicoidal properties. Herein we use the prefix
 identification label instead of the NHI for simplicity. Figure 3.23 shows the prefixes listed in Table 3.10 as a binary tree. The height of this tree is six, and the tree is also referred to as having six levels. Therefore, prefix length $y$ of prefix $x$ is referred to as the level of the tree where the prefix is located. In this example, prefix $a$ ($x = 0^*$) is located at level 1 ($y = 1$), and prefix $l$ ($x = 111111^*$) is located at level 6 ($y = 6$).

**FIGURE 3.22**
Parallel-level lookup process.

**FIGURE 3.23**
Binary tree of the prefixes in Table 3.10.

This binary tree, as shown in Figure 3.23, has six levels but only five levels hold prefixes; levels 1, 3, 4, 5, and 6. Therefore, five prefix tables are provisioned. The table for level 6 requires 64 memory locations and holds prefixes $f$, $g$, $j$, and $l$.

The memory length increases exponentially as the level increases [145]. Therefore, the memory length is the dominant factor in the determination of the amount of used memory [11, 124].

Helix minimizes the number of bits used to address the location of prefix
TABLE 3.6
Prefixes of an example forwarding table.

<table>
<thead>
<tr>
<th>IP Prefix</th>
<th>Prefix ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*</td>
<td>a</td>
</tr>
<tr>
<td>000*</td>
<td>b</td>
</tr>
<tr>
<td>111*</td>
<td>c</td>
</tr>
<tr>
<td>0000*</td>
<td>d</td>
</tr>
<tr>
<td>00001*</td>
<td>e</td>
</tr>
<tr>
<td>000001*</td>
<td>f</td>
</tr>
<tr>
<td>000110*</td>
<td>g</td>
</tr>
<tr>
<td>1110*</td>
<td>h</td>
</tr>
<tr>
<td>1111*</td>
<td>i</td>
</tr>
<tr>
<td>111001*</td>
<td>j</td>
</tr>
<tr>
<td>11101*</td>
<td>k</td>
</tr>
<tr>
<td>111111*</td>
<td>l</td>
</tr>
</tbody>
</table>

$x$ in a prefix table by splitting the prefix into two parts: the MSB portion or family root ($FR_x$) of $x$ and the LSB portion or descendant bits ($D_x$) of $x$. Figure 3.24 shows these two portions of bits in a prefix, separated between bits of level $T$ and $T + 1$. $FR_x$ is stored with the corresponding NHI in the location addressed by $D_x$.

![FIGURE 3.24](image)

Family root and descendant bits in a prefix after a torsion.

The line between $FR_x$ and $D_x$ in the figure, called the torsion level ($L_T$), indicates the bit where the original prefix is split. The selection of $L_T$ determines the size of $FR_x$ and $D_x$, in number of bits, and subsequently the size of the memory amount used for prefix level $v$.

As an example, let’s consider level 6 of the example binary tree after applying a torsion on level 3 ($L_T = 3$). In this case prefixes on level 6, f, g, j, and l, are represented as $FR_f = 000$, $FR_g = 000$, $FR_j = 111$, and $FR_l = 111$, while $D_f = 001$, $D_g = 110$, $D_j = 001$, and $D_l = 111$. Since $D_x$ is used to address the location of prefixes, we see that multiple prefixes may share the
memory location. In our example, prefixes \( f \) and \( j \) share the same location. These prefixes are said to be conjoined. Figures 3.25(a) and 3.25(b) show a prefix table with \( f \) and \( g \) sharing the same memory location.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>( FR_x )</th>
<th>Descendant</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>000</td>
<td>001</td>
</tr>
<tr>
<td>( g )</td>
<td>000</td>
<td>110</td>
</tr>
<tr>
<td>( j )</td>
<td>111</td>
<td>001</td>
</tr>
<tr>
<td>( l )</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

**FIGURE 3.25**
(a) \( FR_x \) and \( Dx \) parts of prefixes at level 6 after \( L_T = 3 \). (b) Reduced prefix table of level 6.

Figure 3.26(a) shows the same tree and additional nodes to help visualize the representation of a torsion. The additional nodes are shown with a dashed-line circumference. Figure 3.26(b) shows the tree after the torsion.

This figure also shows the torsion on level 3 of the example binary tree, the resulting subtrees below the torsion level, and the representation of conjoined prefixes on level 6 of the resulting tree. The nodes below \( L_T \) form subtrees rooted to their string of ancestor nodes; from the root (*) on level 0 to \( L_T \). This string of bits is \( FR_x \). The remaining string of bits is \( Dx \).

For every prefix level, a torsion level is selected as to reduce the tree height such that the number of entries would fit in a memory block and would be fetched in a single memory access. The use of the Helix data structure then reduces the size of a binary tree, and in turn, it achieves optimum lookup speed and small memory use. Helix makes binary trees perform comparably to lookup engines based on expensive ternary content addressable memories (TCAMs).

### 3.11 Ternary Content-Addressable Memory

The implementation of the schemes presented above is based on RAM. However, associative memory is a memory that instead of matching the address input it matches the content. The use of this memory may bring additional benefits, such as embedded multibit search for matching prefixes and higher resolution speed. Such memory is called content addressable memory (CAM).
A version of a CAM where masks can be used to mark bits as a “don’t care” value, or a third value, is called Ternary CAM (TCAM).

In short, TCAMs receive an input, the content that needs to be matched, and it outputs secondary information associated with it. The associated information could be the NHI of a prefix, or the address of where the NHI is in an additional RAM block. Prefixes of different lengths may be stored in a TCAM. Therefore, when a packet destination is input as the content to a TCAM, multiple prefixes with different lengths may match, but only the longest prefix is of interest.

To output a single match, a TCAM matches the first content input into it. That is, longer prefixes are stored in the highest priority positions (or input first), and they are followed by prefixes with immediately shorter lengths, and so on. This storing process for a TCAM solves the matching priority operation required by IP lookup, but also makes it complex to update changes.
Internet Protocol (IP) Address Lookup

in prefixes or routing tables [153]. Figure 3.27 shows a basic example of a TCAM storing prefixes. In the TCAM, there is a priority encoder that assigns selection priority to the prefix on the top. Therefore, longer prefixes are placed at the top of the memory. Once the prefix is matched, the information associated with the entry or the ID of the prefix may be output. In the example the figure shows, packet destination address 133.210.111.7 matches prefix 133.210.111/24, which is associated with next hop address 113.45.22.14. This address is then output by the TCAM.

IP lookup by a TCAM is achieved in a single memory access, which is the optimum lookup time. In the lookup process, when a packet arrives, the destination address is input to the TCAM and all the contents of the TCAM are compared to it.

With these features, TCAMs are a practical alternative to building IP lookup engines [154, 158, 189].

Although the lookup speed of TCAMs is very fast, there are still some issues that need to be considered. One of them is that all entries are compared at once. This comparison of a large number of prefixes requires a large number of comparators. The additional logic takes some room and limits the number of entries that can be stored. Also, access to all memory elements and comparison of their content consume power. Therefore TCAMs are then power-demanding systems.

Several schemes have been developed to overcome some of these issues [133, 189]. However, much of the complexity on addressing these issues is that TCAMs are integrated systems.

FIGURE 3.27
Example of prefixes in a TCAM.
TABLE 3.7
Forwarding table for Exercise 1.

<table>
<thead>
<tr>
<th>IP Prefix</th>
<th>Prefix ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>P1</td>
</tr>
<tr>
<td>1*</td>
<td>P2</td>
</tr>
<tr>
<td>01*</td>
<td>P3</td>
</tr>
<tr>
<td>000*</td>
<td>P4</td>
</tr>
<tr>
<td>0110*</td>
<td>P5</td>
</tr>
<tr>
<td>0111*</td>
<td>P6</td>
</tr>
<tr>
<td>1001*</td>
<td>P7</td>
</tr>
<tr>
<td>1101*</td>
<td>P8</td>
</tr>
<tr>
<td>01001*</td>
<td>P9</td>
</tr>
<tr>
<td>11111*</td>
<td>P10</td>
</tr>
<tr>
<td>111000*</td>
<td>P11</td>
</tr>
</tbody>
</table>

3.12 Additional Readings
Search for simplification of IP lookup complexity has captured the interest of the research community for many years. The number of literature works is quite extensive. Therefore, there is a very long list of surveys of IP lookup algorithms for the interested reader [28, 152, 162, 179]. Other books may also list several schemes [29, 175].

3.13 Exercises
1. Build the binary tree representation from prefixes in Table 3.7.
2. From the forwarding table in Figure 3.7, show it as a disjoint tree, and indicate the largest number of memory accesses that may be needed to match a prefix.
3. Show a path-compressed tree (Patricia tree) of Table 3.7.
4. Indicate the largest number of memory accesses to match a prefix in Exercise 3.
5. Show a Lulea bitmap for level 4 of the forwarding table (or binary tree) of Table 3.7. How many bits can you use for index and code word codes?
6. Show a level-compressed tree (LC-Trie) of prefixes in Table 3.7 with initial stride of 3.
7. Show the multibit trie of the prefixes in Table 3.7 with stride 3. What is the largest number of memory accesses needed for this table?

8. Show the binary tree of the following forwarding table: P1 (*), P2 (1*), P3 (00*), P4 (01*), P5 (1101*), P6 (00101*), P7 (00111*), P8 (111010*), P9 (1110111*), P10 (11101010*).

9. Show the disjoint tree of the forwarding table in Exercise 8.

10. Show the path-compressed tree of the forwarding table in Exercise 8.

11. Show the Lulea data structure, including codeword and base index arrays, maptable, and NHI table for level 7 of the forwarding table in Exercise 8. Use 8 bits as bit-mask size.

12. Find the torsion level, other than level 7, that would require the smallest amount of memory to represent the prefixes using the Helix data structure of the list prefixes below. Indicate the memory size (in length and width) and the total amount (in bits) if 8 bits are used to represent the NHI of a prefix.

List of prefixes: P1 (0000001), P2 (0000111), P3 (0011011), P4 (0100111), P5 (0110011), P6 (0111101), P7 (1000111), P8 (1010001), P9 (1011011), P10 (1100111), P11 (1111011), and P12 (1111101).