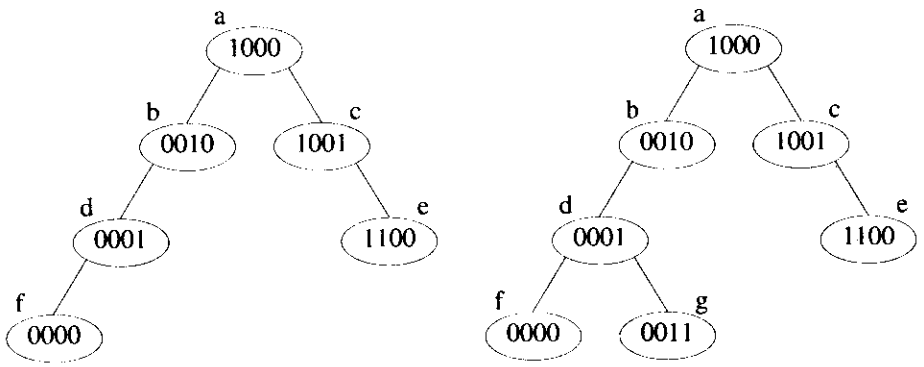
CHAPTER 12

Digital Search Structures

12.1 DIGITAL SEARCH TREES

12.1.1 Definition

A *digital search tree* is a binary tree in which each node contains one element. The element-to-node assignment is determined by the binary representation of the element keys. Suppose that we number the bits in the binary representation of a key from left to right beginning at one. Then bit one of 1000 is 1, and bits two, three, and four are 0. All keys in the left subtree of a node at level i have bit i equal to zero whereas those in the right subtree of nodes at this level have bit $i = 1$. Figure 12.1(a) shows a digital search tree. This tree contains the keys 1000, 0010, 1001, 0001, 1100, and 0000.



(a) Initial tree (b) After 0011 inserted

Figure 12.1: Digital search trees

12.1.2 Search, Insert and Delete

Suppose we are to search for the key $k = 0011$ in the tree of Figure 12.1(a). k is first compared with the key in the root. Since k is different from the key in the root, and since bit one of k is 0, we move to the left child, b, of the root. Now, since k is different from the key in node b, and bit two of k is 0, we move to the left child, d, of b. Since k is different from the key in node d and since bit three of k is one, we move to the right child of d. Node d has no right child to move to. From this we conclude that $k = 0011$ is not in the search tree. If we wish to insert k into the tree, then it is to be added as the right child of d. When this is done, we get the digital search tree of Figure 12.1(b).

The digital search tree functions to search and insert are quite similar to the corresponding functions for binary search trees. The essential difference is that the subtree to move to is determined by a bit in the search key rather than by the result of the comparison of the search key and the key in the current node. The deletion of an item in a leaf is done by removing the leaf node. To delete from any other node, the deleted item must be replaced by a value from any leaf in its subtree and that leaf removed.

Each of these operations can be performed in $O(h)$ time, where h is the height of the digital search tree. If each key in a digital search tree has *keySize* bits, then the height of the digital search tree is at most *keySize* + 1.

EXERCISES

1. Draw a different digital search tree than Figure 12.1 (a) that has the same set of keys.
2. Write the digital search tree functions for the search, insert, and delete operations. Assume that each key has *keySize* bits and that the function *bit* (k, i) returns the ith (from the left) bit of the key k. Show that each of your functions has complexity $O(h)$, where h is the height of the digital search tree.

12.2 BINARY TRIES AND PATRICIA

When we are dealing with very long keys, the cost of a key comparison is high. Since searching a digital search tree requires many comparisons between pairs of keys, digital search trees (and also binary and multiway search trees) are inefficient search structures when the keys are very long. We can reduce the number of key comparisons done during a search to one by using a related structure called *Patricia* (*P*ractical *a*lgorithm *t*o *r*etrieve *i*nformation *c*oded *i*n *a*lphanumeric). We shall develop this structure in three steps. First, we introduce a structure called a binary trie (pronounced try''). Then we transform binary tries into compressed binary tries. Finally, from compressed binary tries we obtain Patricia. Since binary tries and compressed binary tries are introduced only as a means of arriving at Patricia, we do not dwell much on how to manipulate these structures. A more general version of binary tries (called a trie) is considered in the next section.

12.2.1 Binary Tries

A *binary trie* is a binary tree that has two kinds of nodes: *branch nodes* and *element nodes*. A branch node has the two data members *leftChild* and *rightChild*. It has no *data* data member. An element node has the single data member *data*. Branch nodes are used to build a binary tree search structure similar to that of a digital search tree. This search structure leads to element nodes.

Figure 12.2 shows a six-element binary trie. Element nodes are shaded. To search for an element with key k, we use a branching pattern determined by the bits of k. The ith bit of k is used at level i. If it is zero, the search moves to the left subtree. Otherwise, it moves to the right subtree. To search for 0010 we first follow the left child, then again the left child, and finally the right child.

Observe that a successful search in a binary trie always ends at an element node. Once this element node is reached, the key in this node is compared with the key we are searching for. This is the only key comparison that takes place. An unsuccessful search may terminate either at an element node or at a 0 pointer.

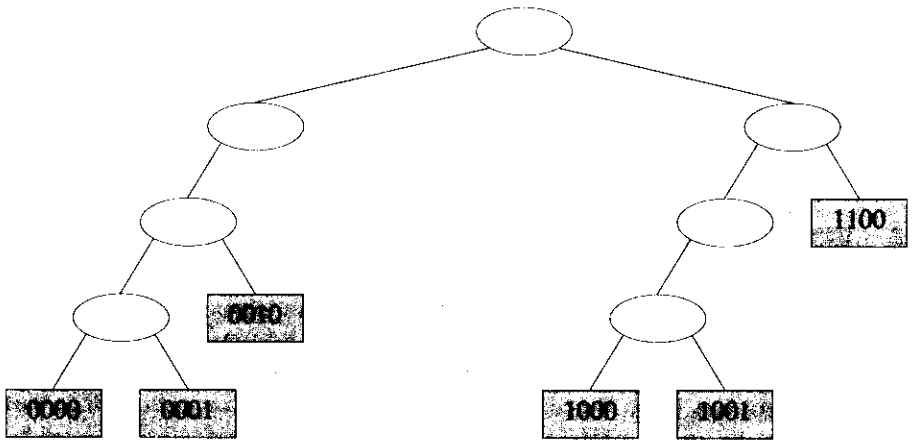


Figure 12.2: Example of a binary trie

12.2.2 Compressed Binary Tries

The binary trie of Figure 12.2 contains branch nodes whose degree is one. By adding another data member, *bitNumber*, to each branch node, we can eliminate all degree-one branch nodes from the trie. The *bitNumber* data member of a branch node gives the bit number of the key that is to be used at this node. Figure 12.3 gives the binary trie that results from the elimination of degree-one branch nodes from the binary trie of Figure 12.2. The number outside a node is its *bitNumber*. A binary trie that has been modified in this way to contain no branch nodes of degree one is called a *compressed binary trie*.

12.2.3 Patricia

Compressed binary tries may be represented using nodes of a single type. The new nodes, called *augmented branch nodes*, are the original branch nodes augmented by the data member *data*. The resulting structure is called *Patricia* and is obtained from a compressed binary trie in the following way:

(1) Replace each branch node by an augmented branch node.

(2) Eliminate the element nodes.

(3) Store the data previously in the element nodes in the *data* data members of the

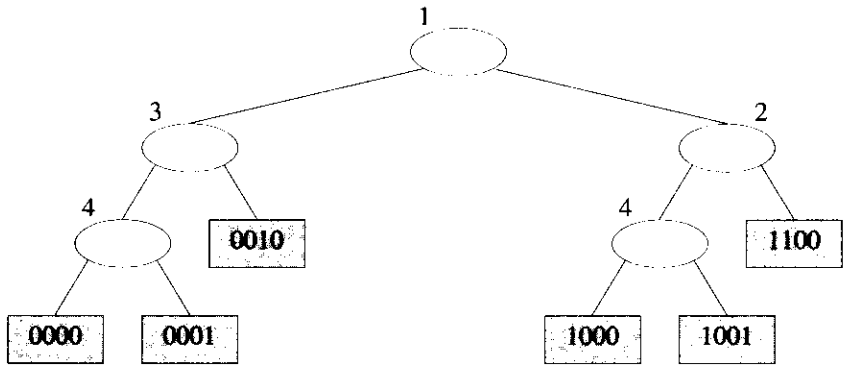


Figure 12.3: Binary trie of Figure 12.2 with degree-one nodes eliminated

augmented branch nodes. Since every nonempty compressed binary trie has one less branch node than it has element nodes, it is necessary to add one augmented branch node. This node is called the *header node*. The remaining structure is the left subtree of the header node. The header node has *bitNumber* equal to zero. Its *rightChild* data member is not used. The assignment of data to augmented branch nodes is done in such a way that the *bitNumber* in the augmented branch node is less than or equal to that in the parent of the element node that contained this data.

(4) Replace the original pointers to element nodes by pointers to the respective augmented branch nodes.

When these transformations are performed on the compressed trie of Figure 12.3, we get the structure of Figure 12.4. Let *root* be the root of Patricia. *root* is 0 iff the Patricia is empty. A Patricia with one element is represented by a header node whose left-child data member points to itself (Figure 12.5(a)). We can distinguish between pointers that pointed originally to branch nodes and those that pointed to element nodes by noting that, in Patricia, the former pointers are directed to nodes with a greater *bitNumber* value, whereas pointers of the latter type are directed to nodes whose *bitNumber* value either is equal to or less than that in the node where the pointer originates.

12.2.3.1 Searching Patricia

To search for an element with key k, we begin at the header node and follow a path determined by the bits in k. When an element pointer is followed, the key in the node reached is compared with k. This is the only key comparison made. No comparisons are

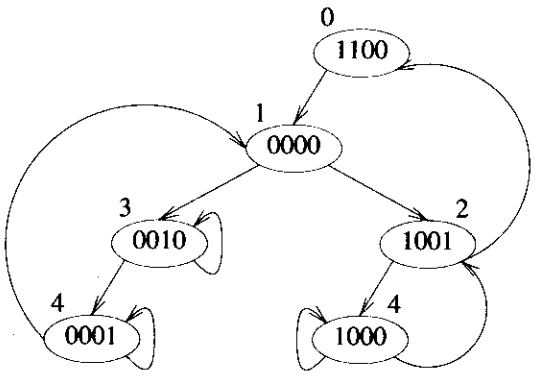


Figure 12.4: An example of Patricia

made on the way down. Suppose we wish to search for $k = 0000$ in the Patricia instance of Figure 12.4. We begin at the header node and follow the left-child pointer to the node with 0000. The bit-number data member of this node is 1. Since bit one of k is 0, we follow the left child pointer to the node with 0010. Now bit three of k is used. Since this is 0, the search moves to the node with 0001. The bit-number data member of this node is 4. The fourth bit of k is zero, so we follow the left-child pointer. This brings us to a node with bit-number data member less than that of the node we moved from. Hence, an element pointer was used. Comparing the key in this node with k, we find a match, and the search is successful.

Next, suppose that we are to search for $k = 1011$. We begin at the header node. The search moves successively to the nodes with 0000, 1001, 1000, and 1001. k is compared with 1001. Since k is not equal to 1001, we conclude that there is no element with this key.

The function to search Patricia tree t is given in Program 12.1. This function returns, a pointer to the last node encountered in the search. If the key in this node is k, the search is successful. Otherwise, t contains no element with key k. The function $bit(i,j)$ returns the jth bit (the leftmost bit is bit one) of i. The C declarations used to define a Patricia tree are:

```
typedef struct patriciaTree *patricia;
      struct {
          int bitNumber;
          element data;
          patricia leftChild, rightChild;
          } patriciaTree;
patricia root;
```

```
patricia search(patricia t, unsigned k)
{/* search the Patricia tree t; return the last node
    encountered; if k is the key in this last node, the
    search is successful */
   patricia currentNode, nextNode;
   if (!t) return NULL; /* empty tree */
   nextNode = t→leftChild;
   currentNode = t;
   while (nextNode→bitNumber > currentNode→bitNumber) {
      currentNode = nextNode;
      nextNode = (bit(k, nextNode→bitNumber)) ?
      nextNode→rightChild : nextNode→leftChild;
   }
   return nextNode;
}
```

Program 12.1: Searching Patricia

12.2.3.2 Inserting into Patricia

Let us now examine how we can insert new elements. Suppose we begin with an empty instance and wish to insert an element with key 1000. The result is an instance that has only a header node (Figure 12.5(a)). Next, consider inserting an element with key k = 0010. First, we search for this key using function *Search* (Program 12.1). The search terminates at the header node. Since 0010 is not equal to the key q = 1000 in this node, we know that 0010 is not currently in the Patricia instance, so the element may be inserted. For this, the keys k and q are compared to determine the first (i.e., leftmost) bit at which they differ. This is bit one. A new node containing the element with key k is added as the left-child of the header node. Since bit one of k is zero, the left child data member of this new node points to itself, and its right-child data member points to the header node. The bit number data member is set to 1. The resulting Patricia instance is shown in Figure 12.5(b).

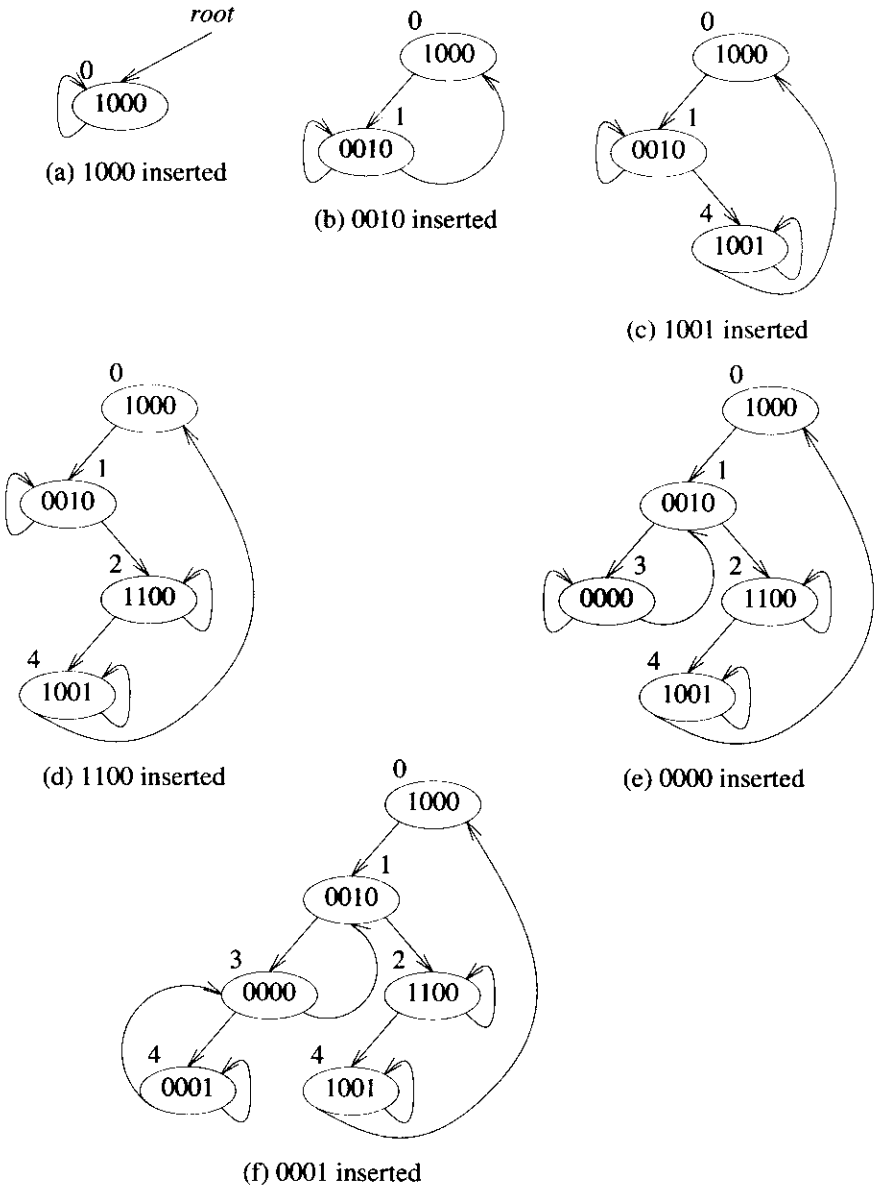


Figure 12.5: Insertion into Patricia

Suppose that the next element to be inserted has $k = 1001$. The search for this key ends at the node with $q = 1000$. The first bit at which k and q differ is bit $j = 4$. Now we search the instance of Figure 12.5(b) using only the first $j - 1 = 3$ bits of k. The last move is from the node with 0010 to that with 1000. Since this is a right-child move, a new node containing the element with key k is to be inserted as the right child of 0010. The bit-number data member of this node is set to $j = 4$. As bit four of k is 1, the right-child data member of the new node points to itself and its left-child data member points to the node with q. Figure 12.5(c) shows the resulting structure.

To insert $k = 1100$ into Figure 12.5(c), we first search for this key. Once again, $q = 1000$. The first bit at which k and q differ is $j = 2$. The search using only the first $j - 1$ bits ends at the node with 1001. The last move is a right child move from 0010. A new node containing the element with key k and bit-number data member $j = 2$ is added as the right child of 0010. Since bit j of k is one, the right-child data member of the new node points to itself. Its left-child data member points to the node with 1001 (this was previously the right child of 0010). The new Patricia instance is shown in Figure 12.5(d). Figure 12.5(e) shows the result of inserting an element with key 0000, and Figure 12.5(f) shows the Patricia instance following the insertion of 0001.

The preceding discussion leads to the insertion function *insert* of Program 12.2. Its complexity is seen to be $O(h)$ where h is the height of t. h can be as large as $\min\{keySize + 1, n\}$ where *keySize* is the number of bits in a key and n is the number of elements. When the keys are uniformly distributed the height is $O(\log n)$. We leave the development of the deletion procedure as an exercise.

```
void insert(patricia *t, element theElement)
{/* insert theElement into the Patricia tree *t */
   patricia current, parent, lastNode, newNode;
   int i;
   if (!(*t)) {/* empty tree */
      MALLOC(*t, sizeof(patriciaTree));
      (*t)→bitNumber = 0; (*t)→data = theElement;
      (*t)→leftChild = *t;
   }
   lastNode = search(*t,theElement.key);
   if (theElement.key == lastNode→data.key) {
      fprintf(stderr, "The key is in the tree. Insertion
                       fails.\n");
      exit(EXIT_FAILURE);
   }
   /* find the first bit where theElement.key and
      lastNode→data.key differ */
```

```
    for (i = 1; bit(theElement.key,i) ==
                bit(lastNode→data.key,i); i++);

    /* search tree using the first i-1 bits */
    current = (*t)→leftChild; parent = *t;
    while (current→bitNumber > parent→bitNumber &&
                               current→bitNumber < i) {
       parent = current;
       current = (bit(theElement.key, current→bitNumber)) ?
                  current→rightChild : current→leftChild;
    }

    /* insert theElement as a child of parent */
    MALLOC(newNode, sizeof(patriciaTree));
    newNode→data = theElement; newNode→bitNumber = i;
    newNode→leftChild = (bit(theElement.key,i)) ?
                              current: newNode;
    newNode→rightChild = (bit(theElement.key,i)) ?
                               newNode : current;
    if (current == parent→leftChild)
       parent→leftChild = newNode;
    else parent→rightChild = newNode;
}
```

Program 12.2: Insertion function for Patricia

EXERCISES

1. Write the binary trie functions for the search, insert, and delete operations. Assume that each key has *keySize* bits and that the function *bit* (k, i) returns the ith (from the left) bit of the key k. Show that each of your functions has complexity $O(h)$, where h is the height of the binary trie.
2. Write the compressed binary trie functions for the search, insert, and delete operations. Assume that each key has *keySize* bits and that the function *bit* (k, i) returns the ith (from the left) bit of the key k. Show that each of your functions has complexity $O(h)$, where h is the height of the compressed binary trie.
3. Write a function to delete the element with key k from a Patricia. The complexity of your function should be $O(h)$, where h is the height of the Patricia instance. Show that this is the case.

12.3 MULTIWAY TRIES

12.3.1 Definition

A multiway trie (or, simply, trie) is a structure that is particularly useful when key values are of varying size. This data structure is a generalization of the binary trie that was developed in the preceding section.

A *trie* is a tree of degree $m \geq 2$ in which the branching at any level is determined not by the entire key value, but by only a portion of it. As an example, consider the trie of Figure 12.6 in which the keys are composed of lowercase letters from the English alphabet. The trie contains two types of nodes: *element*, and *branch*. In Figure 12.6, element nodes are shaded while branch nodes are not shaded. An element node has only a *data* field; a branch node contains pointers to subtrees. In Figure 12.6, each branch node has 27 pointer fields. The extra pointer field is used for the blank character (denoted b). This character is used to terminate all keys, as a trie requires that no key be a prefix of another (see Figure 12.7).

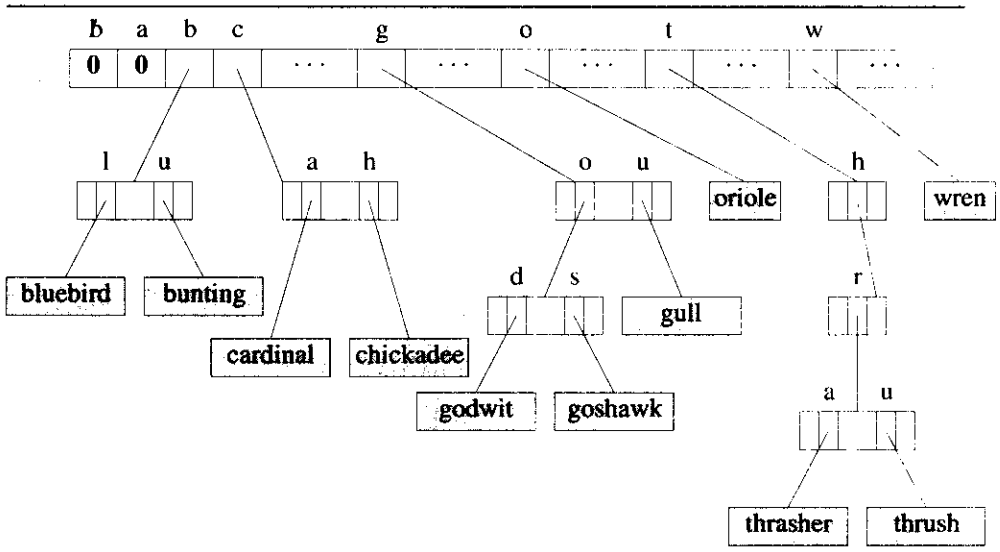


Figure 12.6: Trie created using characters of key value from left to right, one at a time

At the first level all key values are partitioned into disjoint classes depending on their first character. Thus, *root* →*child* [*i*] points to a subtrie containing all key values beginning with the *i*th letter. On the *j*th level the branching is determined by the *j*th character. When a subtrie contains only one key value, it is replaced by a node of type

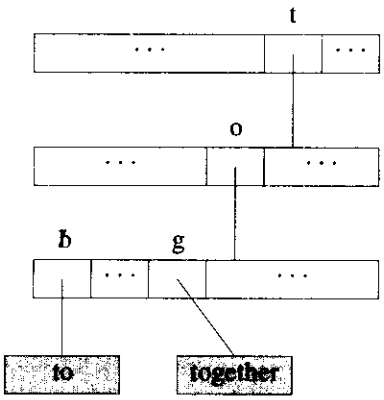


Figure 12.7: Trie showing need for a terminal character (in this case a blank)

element. This node contains the key value, together with other relevant information, such as the address of the record with this key value.

As another example of a trie, suppose that we have a collection of student records that contain fields such as student name, major, date of birth, and social security number (SS#). The key field is the social security number, which is a nine digit decimal number. To keep the example manageable, assume that we have a total of five elements. Figure 12.8 shows the name and SS# fields for each of these five elements.

Name	SS#
Jack	951-94-1654
Jill	562-44-2169
Bill	271-16-3624
Kathy	278-49-1515
April	951-23-7625

Figure 12.8: Five elements (student records)

To obtain a trie representation for these five elements, we first select a radix that will be used to decompose each key into digits. If we use the radix 10, the decomposed digits are just the decimal digits shown in Figure 12.8. We shall examine the digits of the key field (i.e., SS#) from left to right. Using the first digit of the SS#, we partition the

elements into three groups—elements whose SS# begins with 2 (i.e., Bill and Kathy), those that begin with 5 (i.e., Jill), and those that begin with 9 (i.e., April and Jack). Groups with more than one element are partitioned using the next digit in the key. This partitioning process is continued until every group has exactly one element in it.

The partitioning process described above naturally results in a tree structure that has 10-way branching as is shown in Figure 12.9. The tree employs two types of nodes—*branch nodes* and *element nodes*. Each branch node has 10 children (or pointer) fields. These fields, *child* [0:9], have been labeled $0,1,\cdots,9$ for the root node of Figure 12.9. *root.child* [*i*] points to the root of a subtrie that contains all elements whose first digit is *i*. In Figure 12.9, nodes *A,B,D,E,F*, and *I* are branch nodes. The remaining nodes, nodes *C,G,H,J*, and *K* are element nodes. Each element node contains exactly one element. In Figure 12.9, only the key field of each element is shown in the element nodes.

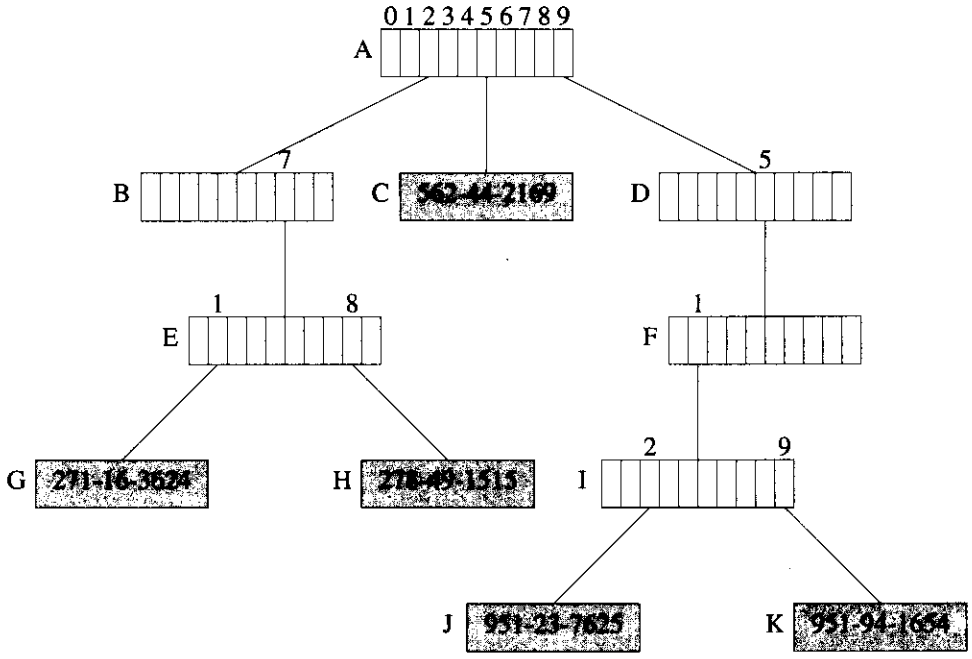


Figure 12.9: Trie for the elements of Figure 12.8

12.3.2 Searching a Trie

To search a trie for a key, x, we must break x into its constituent characters and follow the branches determined by these characters. The function *search* (Program 12.3) assumes that $p \rightarrow u$. *key* is the key represented in node p if p is an element node and that a blank has been appended to the search key before invocation. The function invocation is *search* (t,*key*, 1). *search* uses the function *getIndex* (*key*,i) which performs the ith level sampling of the key. In the case of left to right single character sampling, this function extracts the ith character of the key and converts it to an integer index that tells us which pointer field of the branch node to use.

```
triePointer search(triePointer t, char *key, int i)
{/* search the trie t, return NULL if there is no
    element with this key, otherwise return a pointer
    to the node with the matching element */
   if (!t) return NULL; /* not found */
   if (t->tag == data)
      return ((strcmp(t->key,key)) ? NULL : t);
   return search(t->child[getIndex(key,i)], key, i+1);
}
```

Program 12.3: Searching a trie

Analysis of *search*: The *search* function is straightforward and we may readily verify that the worst case search time is $O(l)$, where l is the number of levels in the trie (including both branch and element nodes). □

12.3.3 Sampling Strategies

Given a set of key values to be represented in an index, the number of levels in the trie will depend on the strategy or key sampling technique used to determine the branching at each level. This can be defined by a sampling function, *sample* (x,i), which appropriately samples x for branching at the ith level. In the trie of Figure 12.6 and in the search function of Program 12.4, the sampling function is *sample* (x,i) = ith character of x. Some other choices for this function are

(1) *sample* $(x,i) = x_{n-i+1}$

(2) *sample* $(x,i) = x_{r(x,i)}$ for $r(x,i)$ a randomization function

$$(3) \quad sample(x,i) = \begin{cases} x_{i/2} \text{ if } i \text{ is even} \\ x_{n-(i-1)/2} \text{ if } i \text{ is odd} \end{cases}$$

where $x = x_1 x_2 \cdots x_n$.

For each of these functions, one may easily construct key value sets for which the particular function is best (i.e., it results in a trie with the fewest number of levels). The trie of Figure 12.6 has five levels. Using function (1) on the same key values yields the trie of Figure 12.10, which has only three levels. An optimal sampling function for this data set will yield a trie that has only two levels (Figure 12.11). Choosing the optimal sampling function for any particular set of values is very difficult. In a dynamic situation, with insertion and deletion, we wish to optimize average performance. In the absence of any further information on key values, the best choice would probably be function (2).

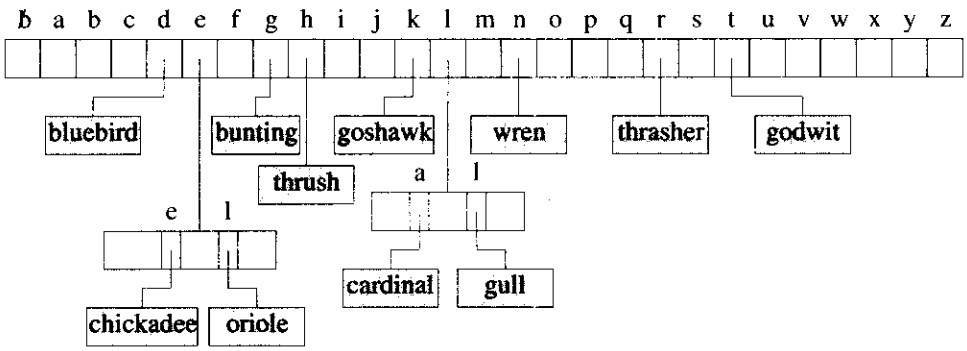


Figure 12.10: Trie constructed for data of Figure 12.6 sampling one character at a time, from right to left

Although all our examples of sampling have involved single-character sampling we need not restrict ourselves to this. The key value may be interpreted as consisting of digits using any radix we desire. Using a radix of 27^2 would result in two-character sampling. Other radixes would give different samplings.

The maximum number of levels in a trie can be kept low by adopting a different strategy for element nodes. These nodes can be designed to hold more than one key value. If the maximum number of levels allowed is l, then all key values that are synonyms up to level $l - 1$ are entered into the same element node. If the sampling function is chosen correctly, there will be only a few synonyms in each element node. The element node will therefore be small and can be processed in internal memory. Figure 12.12 shows the use of this strategy on the trie of Figure 12.6 with $l = 3$. In further

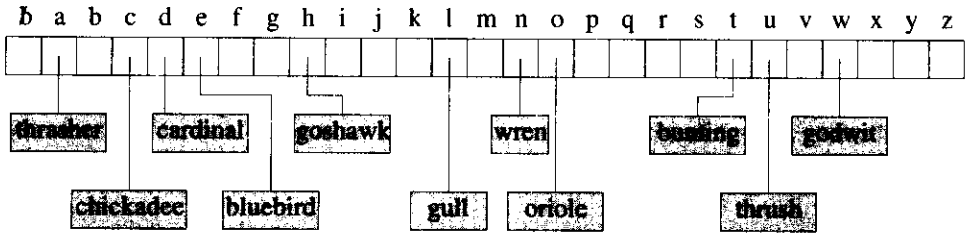


Figure 12.11: An optimal trie for the data of Figure 12.6 sampling on the first level done by using the fourth character of the key values

discussion we shall, for simplicity, assume that the sampling function in use is *sample* (x,i) = ith character of x and that no restriction is placed on the number of levels in the trie.

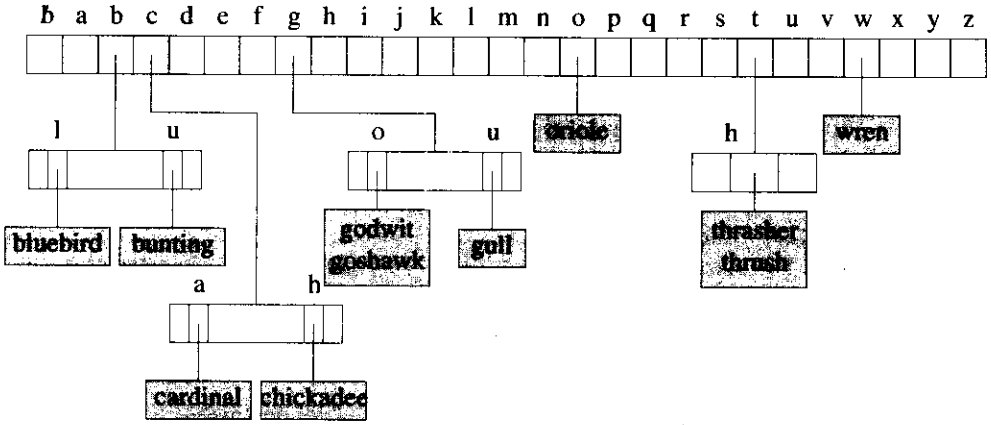


Figure 12.12: Trie obtained for data of Figure 12.6 when number of levels is limited to 3; keys have been sampled from left to right, one character at a time

12.3.4 Insertion into a Trie

Insertion into a trie is straightforward. We shall illustrate the procedure by two examples and leave the formal writing of the algorithm as an exercise. Let us consider the trie of Figure 12.6 and insert into it the keys bobwhite and bluejay. First, we have x = bobwhite and we attempt to search for bobwhite. This leads us to node σ, where we discover that $\sigma.link['o'] = 0$. Hence, x is not in the trie and may be inserted here (see Figure 12.13). Next, x = bluejay, and a search of the trie leads us to the element node that contains bluebird. The keys bluebird and bluejay are sampled until the sampling results in two different values. This happens at the fifth letter. Figure 12.13 shows the trie after both insertions.

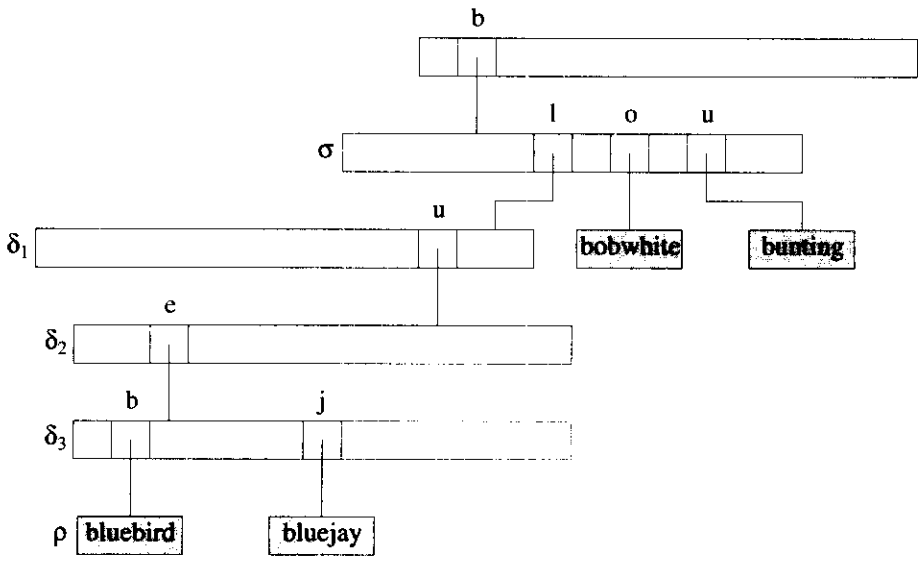


Figure 12.13: Section of trie of Figure 12.6 showing changes resulting from inserting bobwhite and bluejay

12.3.5 Deletion from a Trie

Once again, we shall not present the deletion algorithm formally but will look at two examples to illustrate some of the ideas involved in deleting entries from a trie. From the

trie of Figure 12.13, let us first delete bobwhite. To do this we set $\sigma.link['o']$ equal to 0. No other changes need to be made. Next, let us delete bluejay. This deletion leaves us with only one key value in the subtrie, δ_3. This means that the node δ_3 may be deleted, and ρ can be moved up one level. The same can be done for nodes δ_1 and δ_2. Finally, the node σ is reached. The subtrie with root σ has more than one key value. Therefore, ρ cannot be moved up any more levels, and we set $\sigma.link['l']$ equal to ρ. To facilitate deletions from tries, it is useful to add a *count* data member in each branch node. This data member contains the number of children the node has.

As in the case of binary tries, we can define compressed tries in which each branch node has at least two children. In this case, each branch node is augmented to have an additional data member, *skip*, that indicates the number of levels of branching that have been eliminated (alternately, we can have a data member, *sample*, that indicates the sampling level to use).

12.3.6 Keys With Different Length

As noted earlier, the keys in a trie must be such that no key is a prefix of another. When all keys are of the same length, as is the case in our SS# example of Figure 12.9, this property is assured. But, when keys are of different length, as is the case with the keys of Figure 12.6, it is possible for one key to be a prefix of another. A popular way to handle such key collections is to append a special character such as a blank or a # that doesn't appear in any key to the end of each key. This assures us that the modified keys (with the special character appended) satisfy the no-prefix property.

An alternative to adding a special character at the end of each key is to give each node a *data* field that is used to store the element (if any) whose key exhausts at that node. So, for example, the element whose key is 27 can be stored in node E of Figure 12.9. When this alternative is used, the search strategy is modified so that when the digits of the search key are exhausted, we examine the *data* field of the reached node. If this *data* field is empty, we have no element whose key equals the search key. Otherwise, the desired element is in this *data* field.

It is important to note that in applications that have different length keys with the property that no key is a prefix of another, neither of the just-mentioned strategies is needed.

12.3.7 Height of a Trie

In the worst case, a root-node to element-node path has a branch node for every digit in a key. Therefore, the height of a trie is at most $numberofdigits+1$.

A trie for social security numbers has a height that is at most 10. If we assume that it takes the same time to move down one level of a trie as it does to move down one level

of a binary search tree, then with at most 10 moves we can search a social-security trie. With this many moves, we can search a binary search tree that has at most $2^{10} - 1 = 1023$ elements. This means that, we expect searches in the social security trie to be faster than searches in a binary search tree (for student records) whenever the number of student records is more than 1023. The breakeven point will actually be less than 1023 because we will normally not be able to construct full or complete binary search trees for our element collection.

Since a SS# is nine digits, a social security trie can have up to 10^9 elements in it. An AVL tree with 10^9 elements can have a height that is as much as (approximately) $1.44\log_2(10^9+2) = 44$. Therefore, it could take us four times as much time to search for elements when we organize our collection of student records as an AVL tree rather than as a trie!

12.3.8 Space Required and Alternative Node Structures

The use of branch nodes that have as many child fields as the radix of the digits (or one more than this radix when different keys may have different length) results in a fast search algorithm. However, this node structure is often wasteful of space because many of the child fields are **NULL**. A radix r trie for d digit keys requires $O(rdn)$ child fields, where n is the number of elements in the trie. To see this, notice that in a d digit trie with n element nodes, each element node may have at most d ancestors, each of which is a branch node. Therefore, the number of branch nodes is at most dn. (Actually, we cannot have this many branch nodes, because the element nodes have common ancestors like the root node.)

We can reduce the space requirements, at the expense of increased search time, by changing the node structure. Some of the possible alternative structures for the branch node of a trie are considered below.

A chain of nodes.
Each node of the chain has the three fields *digitValue*, *child*, and *next*. Node E of Figure 12.9, for example, would be replaced by the chain shown in Figure 12.14.

Figure 12.14: Chain for node E of Figure 12.9

The space required by a branch node changes from that required for r

children/pointer fields to that required for $2p$ pointer fields and p digit value fields, where p is the number of children fields in the branch node that are not **NULL**. Under the assumption that pointer fields and digit value fields are of the same size, a reduction in space is realized when more than two-thirds of the children fields in branch nodes are **NULL**. In the worst case, almost all the branch nodes have only 1 field that is not **NULL** and the space savings become almost $(1-3/r)*100\%$.

A (balanced) binary search tree.
Each node of the binary search tree has a digit value and a pointer to the subtrie for that digit value. Figure 12.15 shows the binary search tree for node E of Figure 12.9.

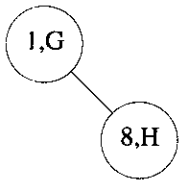


Figure 12.15: Binary search tree for node E of Figure 12.9

Under the assumption that digit values and pointers take the same amount of space, the binary search tree representation requires space for $4p$ fields per branch node, because each search tree node has fields for a digit value, a subtrie pointer, a left child pointer, and a right child pointer. The binary search tree representation of a branch node saves us space when more than three-fourths of the children fields in branch nodes are **NULL**. Note that for large r, the binary search tree is faster to search than the chain described above.

A binary trie.
Figure 12.16 shows the binary trie for node E of Figure 12.9. The space required by a branch node represented as a binary trie is at most $(2*\lceil \log_2 r \rceil + 1)p$.

A hash table.
When a hash table with a sufficiently small loading density is used, the expected time performance is about the same as when the node structure of Figure 12.9 is used. Since we expect the fraction of **NULL** child fields in a branch node to vary from node to node and also to increase as we go down the trie, maximum space efficiency is obtained by consolidating all of the branch nodes into a single hash table. To accomplish this, each node in the trie is assigned a number, and each parent to child pointer is replaced by a triple of the form (*currentNode*,*digitValue*,*childNode*). The numbering scheme for

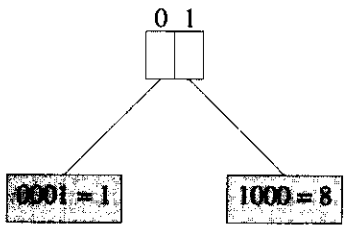


Figure 12.16: Binary trie for node *E* of Figure 12.9

nodes is chosen so as to easily distinguish between branch and element nodes. For example, if we expect to have at most 100 elements in the trie at any time, the numbers 0 through 99 are reserved for element nodes and the numbers 100 on up are used for branch nodes. The element nodes are themselves represented as an array *element* [100]. (An alternative scheme is to represent pointers as tuples of the form (*currentNode*,*digitValue*,*childNode*,*childNodeIsBranchNode*), where *childNodeIsBranchNode* = **true** iff the child is a branch node.)

Suppose that the nodes of the trie of Figure 12.9 are assigned numbers as given in Figure 12.17. This number assignment assumes that the trie will have no more than 10 elements.

Node	A	B	C	D	E	F	G	H	I	J	K
Number	10	11	0	12	13	14	1	2	15	3	4

Figure 12.17: Number assignment to nodes of trie of Figure 12.9

The pointers in node *A* are represented by the tuples (10,2,11),(10,5,0), and (10,9,12). The pointers in node *E* are represented by the tuples (13,1,1) and (13,8,2).

The pointer triples are stored in a hash table using the first two fields (i.e., the *currentNode* and *digitValue*) as the key. For this purpose, we may transform the two field key into an integer using the formula *currentNode* * *r*+*digitValue*, where *r* is the trie radix, and use the division method to hash the transformed key into a home bucket. The data presently in element node *i* is stored in *element* [*i*].

To see how all this works, suppose we have set up the trie of Figure 12.9 using the hash table scheme just described. Consider searching for an element with key 278-49-1515. We begin with the knowledge that the root node is assigned the number 10. Since

the first digit of the search key is 2, we query our hash table for a pointer triple with key (10,2). The hash table search is successful and the triple (10,2,11) is retrieved. The *childNode* component of this triple is 11, and since all element nodes have a number 9 or less, the child node is determined to be a branch node. We make a move to the branch node 11. To move to the next level of the trie, we use the second digit 7 of the search key. For the move, we query the hash table for a pointer with key (11,7). Once again, the search is successful and the triple (11,7,13) is retrieved. The next query to the hash table is for a triple with key (13,8). This time, we obtain the triple (13,8,2). Since, *childNode* = 2 < 10, we know that the pointer gets us to an element node. So, we compare the search key with the key of *element*[2]. The keys match, and we have found the element we were looking for.

When searching for an element with key 322-16-8976, the first query is for a triple with key (10,3). The hash table has no triple with this key, and we conclude that the trie has no element whose key equals the search key.

The space needed for each pointer triple is about the same as that needed for each node in the chain of nodes representation of a trie node. Therefore, if we use a linear open addressed hash table with a loading density of α, the hash table scheme will take approximately $(1/\alpha - 1)*100\%$ more space than required by the chain of nodes scheme. However, when the hash table scheme is used, we can retrieve a pointer in O(1) expected time, whereas the time to retrieve a pointer using the chain of nodes scheme is $O(r)$. When the (balanced) binary search tree or binary trie schemes are used, it takes $O(\log r)$ time to retrieve a pointer. For large radixes, the hash table scheme provides significant space saving over the scheme of Figure 12.9 and results in a small constant factor degradation in the expected time required to perform a search.

The hash table scheme actually reduces the expected time to insert elements into a trie, because when the node structure of Figure 12.9 is used, we must spend $O(r)$ time to initialize each new branch node (see the description of the insert operation below). However, when a hash table is used, the insertion time is independent of the trie radix.

To support the removal of elements from a trie represented as a hash table, we must be able to reuse element nodes. This reuse is accomplished by setting up an available space list of element nodes that are currently not in use.

12.3.9 Prefix Search and Applications

You have probably realized that to search a trie we do not need the entire key. Most of the time, only the first few digits (i.e., a prefix) of the key is needed. For example, our search of the trie of Figure 12.9 for an element with key 951-23-7625 used only the first four digits of the key. The ability to search a trie using only the prefix of a key enables us to use tries in applications where only the prefix might be known or where we might desire the user to provide only a prefix. Some of these applications are described below.

Criminology: Suppose that you are at the scene of a crime and observe the first few characters *CRX* on the registration plate of the getaway car. If we have a trie of registration numbers, we can use the characters *CRX* to reach a subtrie that contains all registration numbers that begin with *CRX*. The elements in this subtrie can then be examined to see which cars satisfy other properties that might have been observed.

Automatic Command Completion: When using an operating system such as Unix or Windows (command prompt), we type in system commands to accomplish certain tasks. For example, the Unix and DOS command *cd* may be used to change the current directory. Figure 12.18 gives a list of commands that have the prefix *ps* (this list was obtained by executing the command *ls /usr/local/bin/ps** on a Unix system).

ps2ascii	ps2pdf	psbook	psmandup	psselect
ps2epsi	ps2pk	pscal	psmerge	pstopnm
ps2frag	ps2ps	psidtopgm	psnup	pstops
ps2gif	psbb	pslatex	psresize	pstruct

Figure 12.18: Commands that begin with "ps"

We can simplify the task of typing in commands by providing a command completion facility which automatically types in the command suffix once the user has typed in a long enough prefix to uniquely identify the command. For instance, once the letters *psi* have been entered, we know that the command must be *psidtopgm* because there is only one command that has the prefix *psi*. In this case, we replace the need to type in a 9 character command name by the need to type in just the first 3 characters of the command!

A command completion system is easily implemented when the commands are stored in a trie using ASCII characters as the digits. As the user types the command digits from left to right, we move down the trie. The command may be completed as soon as we reach an element node. If we fall off the trie in the process, the user can be informed that no command with the typed prefix exists.

Although we have described command completion in the context of operating system commands, the facilty is useful in other environments:

(1) A web browser keeps a history of the URLs of sites that you have visited. By organizing this history as a trie, the user need only type the prefix of a previously used URL and the browser can complete the URL.

(2) A word processor can maintain a collection of words and can complete words as you type the text. Words can be completed as soon as you have typed a long enough prefix to identify the word uniquely.

(3) An automatic phone dialler can maintain a list of frequently called telephone numbers as a trie. Once you have punched in a long enough prefix to uniquely identify the phone number, the dialler can complete the call for you.

12.3.10 Compressed Tries

Take a close look at the trie of Figure 12.9. This trie has a few branch nodes (nodes *B*,*D*, and *F*) that do not partition the elements in their subtrie into two or more nonempty groups. We often can improve both the time and space performance metrics of a trie by eliminating all branch nodes that have only one child. The resulting trie is called a *compressed trie*.

When branch nodes with a single child are removed from a trie, we need to keep additional information so that trie operations may be performed correctly. The additional information stored in three compressed trie structures is described below.

12.3.10.1 Compressed Tries with Digit Numbers

In a *compressed trie with digit numbers*, each branch node has an additional field *digitNumber* that tells us which digit of the key is used to branch at this node. Figure 12.19 shows the compressed trie with digit numbers that corresponds to the trie of Figure 12.9. The leftmost field of each branch node of Figure 12.19 is the *digitNumber* field.

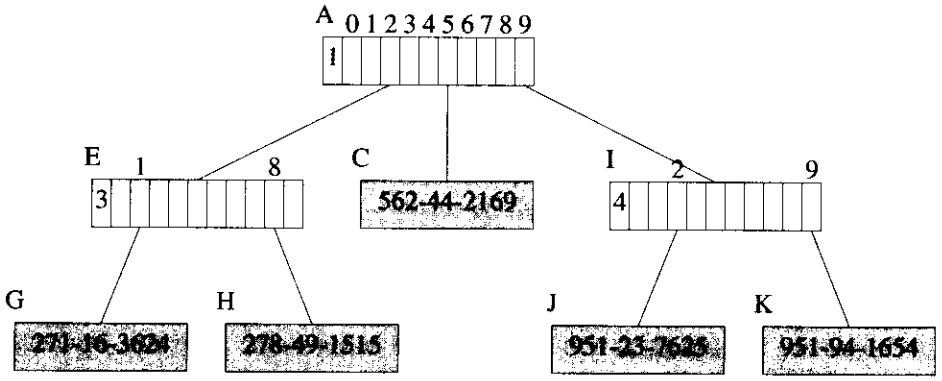


Figure 12.19: Compressed trie with digit numbers

12.3.10.2 Searching a Compressed Trie with Digit Numbers

A compressed trie with digit numbers may be searched by following a path from the root. At each branch node, the digit, of the search key, given in the branch node's *digitNumber* field is used to determine which subtrie to move to. For example, when searching the trie of Figure 12.19 for an element with key 951-23-7625, we start at the root of the trie. Since the root node is a branch node with *digitNumber*=1, we use the first digit 9 of the search key to determine which subtrie to move to. A move to node *A.child*[9]=*I* is made. Since, *I.digitNumber*=4, the fourth digit, 2, of the search key tells us which subtrie to move to. A move is now made to node *I.child*[2]=*J*. We are now at an element node, and the search key is compared with the key of the element in node *J*. Since the keys match, we have found the desired element.

Notice that a search for an element with key 913-23-7625 also terminates at node *J*. However, the search key and the element key at node *J* do not match and we conclude that the trie contains no element with key 913-23-7625.

12.3.10.3 Inserting into a Compressed Trie with Digit Numbers

To insert an element with key 987-26-1615 into the trie of Figure 12.19, we first search for an element with this key. The search ends at node *J*. Since, the search key and the key, 951-23-7625, of the element in this node do not match, we conclude that the trie has no element whose key matches the search key. To insert the new element, we find the first digit where the search key differs from the key in node *J* and create a branch node for this digit. Since, the first digit where the search key 987-26-1615 and the element key 951-23-7625 differ is the second digit, we create a branch node with *digitNumber*=2. Since, digit values increase as we go down the trie, the proper place to insert the new branch node can be determined by retracing the path from the root to node *J* and stopping as soon as either a node with digit value greater than 2 or the node *J* is reached. In the trie of Figure 12.19, this path retracing stops at node *I*. The new branch node is made the parent of node *I*, and we get the trie of Figure 12.20.

Consider inserting an element with key 958-36-4194 into the compressed trie of Figure 12.19. The search for an element with this key terminates when we fall of the trie by following the pointer *I.child*[3]=*NULL*. To complete the insertion, we must first find an element in the subtrie rooted at node *I*. This element is found by following a downward path from node *I* using (say) the first non **NULL** link in each branch node encountered. Doing this on the compressed trie of Figure 12.19, leads us to node *J*. Having reached an element node, we find the first digit where the element key and the search key differ and complete the insertion as in the previous example. Figure 12.21 shows the resulting compressed trie.

Because of the possible need to search for the first non **NULL** child pointer in each branch node, the time required to insert an element into a compressed tries with

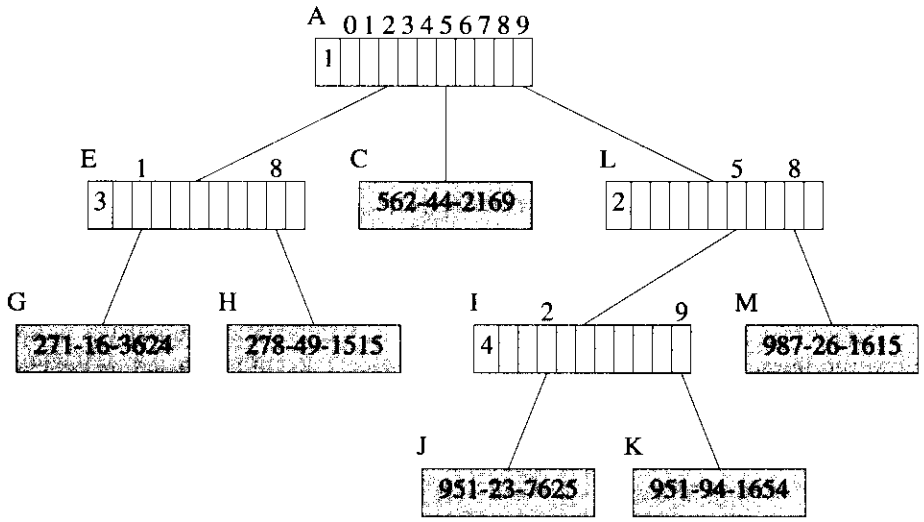


Figure 12.20: Compressed trie following the insertion of 987-26-1615 into the compressed trie of Figure 12.19

digit numbers is O(rd), where r is the trie radix and d is the maximum number of digits in any key.

12.3.10.4 Deleting an Element from a Compressed Trie with Digit Numbers

To delete an element whose key is k, we do the following:

(1) Find the element node X that contains the element whose key is k.

(2) Discard node X.

(3) If the parent of X is left with only one child, discard the parent node also. When the parent of X is discarded, the sole remaining child of the parent of X becomes a child of the grandparent (if any) of X.

To remove the element with key 951-94-1654 from the compressed trie of Figure 12.21, we first locate the node K that contains the element that is to be removed. When this node is discarded, the parent I of K is left with only one child. Consequently, node I

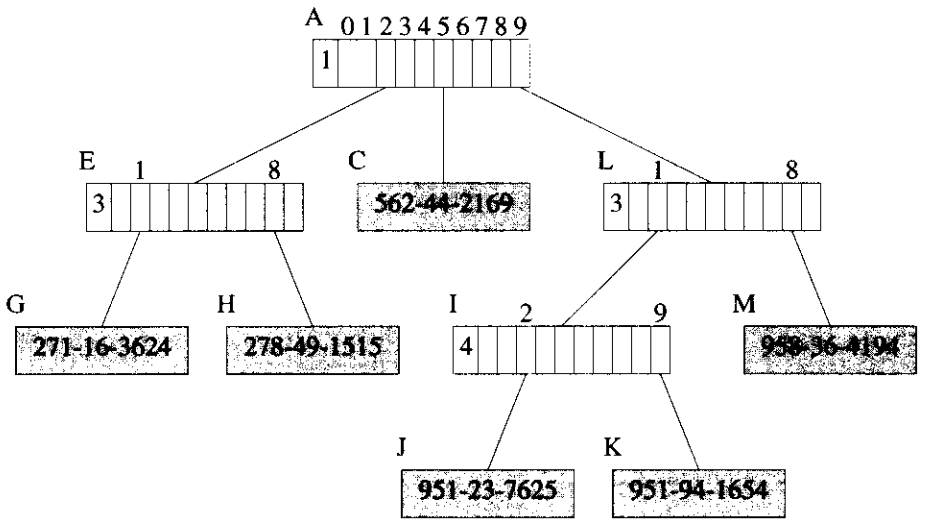


Figure 12.21: Compressed trie following the insertion of 958-36-4194 into the compressed trie of Figure 12.19

is also discarded, and the only remaining child J of node I is the made a child of the grandparent of K. Figure 12.22 shows the resulting compressed trie.

Because of the need to determine whether a branch node is left with two or more children, removing a d digit element from a radix r trie takes $O(d+r)$ time.

12.3.11 Compressed Tries with Skip Fields

In a *compressed trie with skip fields*, each branch node has an additional field *skip* which tells us the number of branch nodes that were originally between the current branch node and its parent. Figure 12.23 shows the compressed trie with skip fields that corresponds to the trie of Figure 12.9. The leftmost field of each branch node of Figure 12.23 is the skip field.

The algorithms to search, insert, and remove are very similar to those used for a compressed trie with digit numbers.

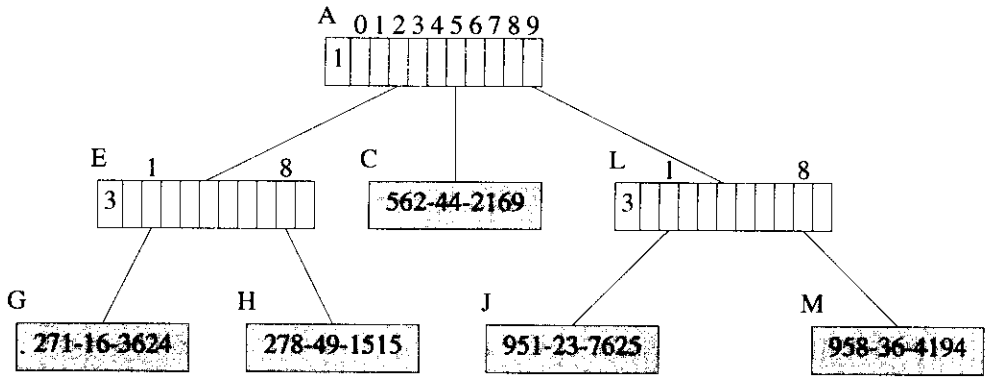


Figure 12.22: Compressed trie following the removal of 951-94-1654 from the compressed trie of Figure 12.21

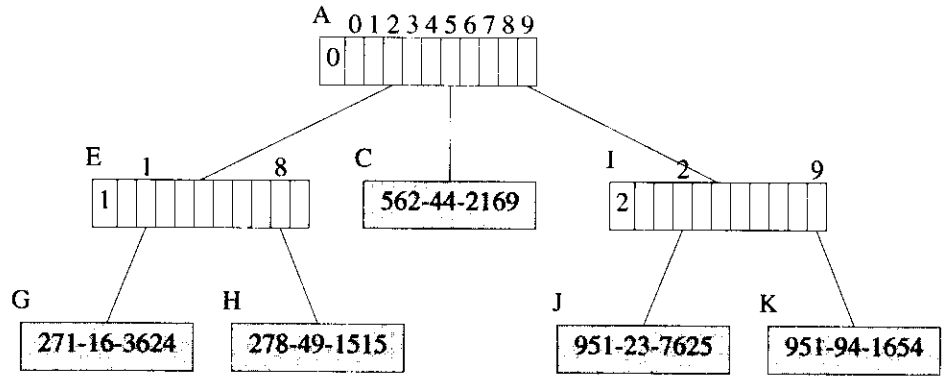


Figure 12.23: Compressed trie with skip fields

12.3.12 Compressed Tries with Labeled Edges

In a *compressed trie with labeled edges*, each branch node has the following additional information associated with it: a pointer/reference *element* to an element (or element

node) in the subtrie, and an integer *skip* which equals the number of branch nodes eliminated between this branch node and its parent. Figure 12.24 shows the compressed trie with labeled edges that corresponds to the trie of Figure 12.9. The first field of each branch node is its *element* field, and the second field is the *skip* field.

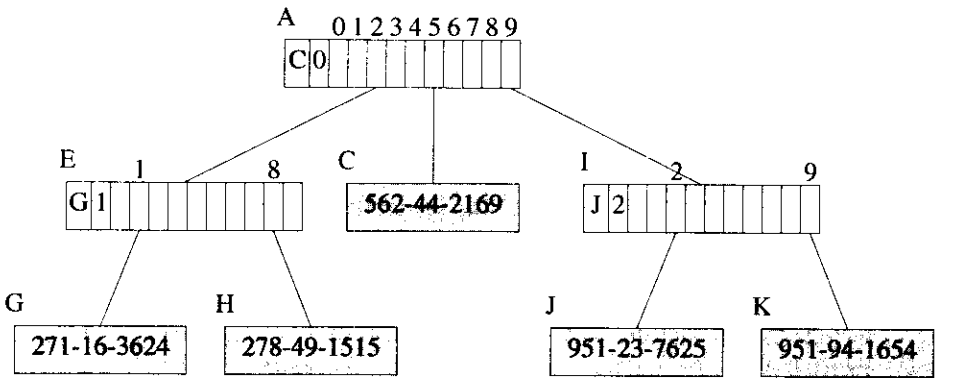


Figure 12.24: Compressed trie with labeled edges

Even though we store the "label" with branch nodes, it is convenient to think of this information as being associated with the edge that comes into the branch node from its parent (when the branch node is not the root). When moving down a trie, we follow edges, and when an edge is followed. we skip over the number of digits given by the *skip* field of the edge information. The value of the digits that are skipped over may be determined by using the *element* field.

When moving from node *A* to node *I* of the compressed trie of Figure 12.24, we use digit 1 of the key to determine which child field of *A* is to be used. Also, we skip over the next 2 digits, that is, digits 2 and 3, of the keys of the elements in the subtrie rooted at *I*. Since all elements in the subtrie *I* have the same value for the digits that are skipped over, we can determine the value of these skipped over digits from any of the elements in the subtrie. Using the *element* field of the edge label, we access the element node *J*, and determine that the digits that are skipped over are 5 and 1.

12.3.12.1 Searching a Compressed Trie with Labeled Edges

When searching a compressed trie with labeled edges, we can use the edge label to terminate unsuccessful searches (possibly) before we reach an element node or fall off the trie. As in the other compressed trie variants, the search is done by following a path

from the root. Suppose we are searching the compressed trie of Figure 12.24 for an element with key 921-23-1234. Since the *skip* value for the root node is 0, we use the first digit 9 of the search key to determine which subtrie to move to. A move to node *A.child*[9]=*I* is made. By examining the edge label (stored in node *I*), we determine that, in making the move from node *A* to node *I*, the digits 5 and 1 are skipped. Since these digits do not agree with the next two digits of the search key, the search terminates with the conclusion that the trie contains no element whose key equals the search key.

12.3.12.2 Inserting into a Compressed Trie with Labeled Edges

To insert an element with key 987-26-1615 into the compressed trie of Figure 12.24, we first search for an element with this key. The search terminates unsuccessfully when we move from node *A* to node *I* because of a mismatch between the skipped over digits and the corresponding digits of the search key. The first mismatch is at the first skipped over digit. Therefore, we insert a branch node *L* between nodes *A* and *I*. The *skip* value for this branch node is 0, and its *element* field is set to reference the element node for the newly inserted element. We must also change the *skip* value of *I* to 1. Figure 12.25 shows the resulting compressed trie.

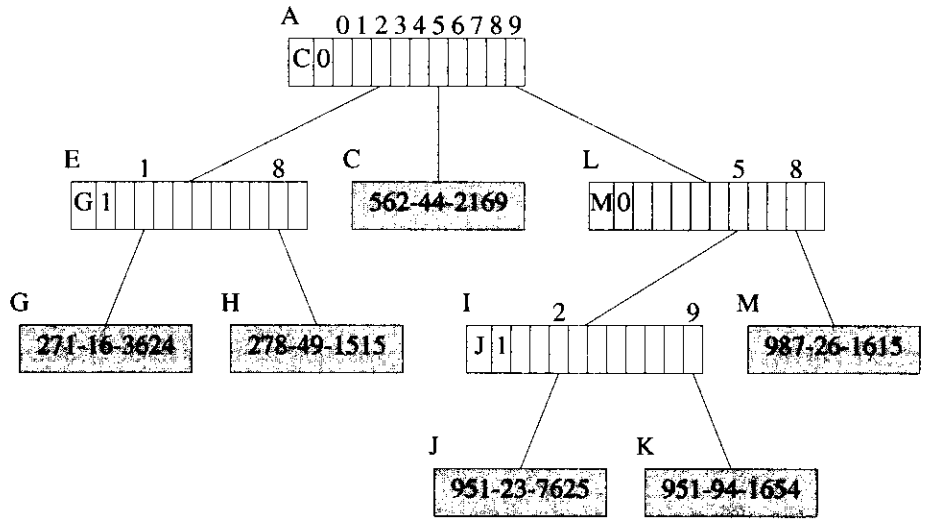


Figure 12.25: Compressed trie (with labeled edges) following the insertion of 987-26-1615 into the compressed trie of Figure 12.24

Suppose we are to insert an element with key 958-36-4194 into the compressed trie of Figure 12.25. The search for an element with this key terminates when we move to node *I* because of a mismatch between the digits that are skipped over and the corresponding digits of the search key. A new branch node is inserted between nodes *A* and *I* and we get the compressed trie that is shown in Figure 12.26.

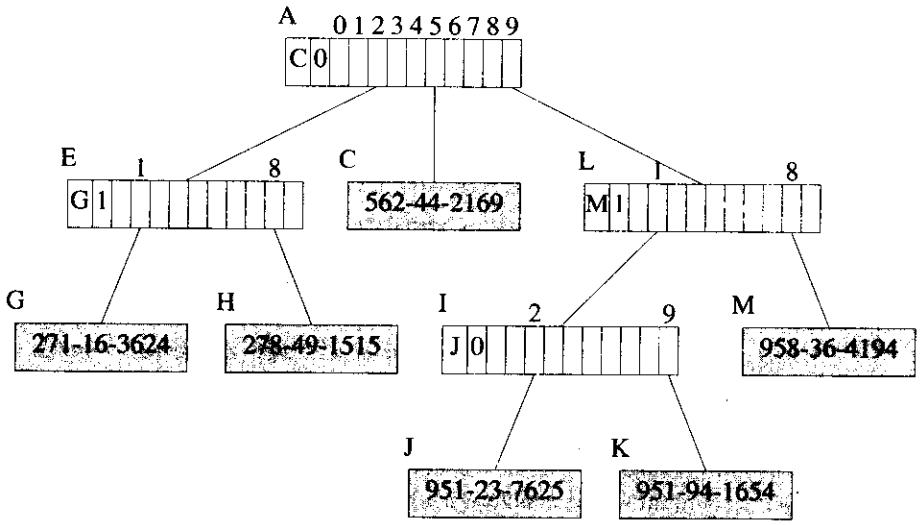


Figure 12.26: Compressed trie (with labeled edges) following the insertion of 958-36-4194 into the compressed trie of Figure 12.24

The time required to insert a d digit element into a radix r compressed trie with labeled edges is $O(r+d)$.

12.3.12.3 Deleting an Element from a Compressed Trie with Labeled Edges

This is similar to removal from a compressed trie with digit numbers except for the need to update the *element* fields of branch nodes whose *element* field references the removed element.

12.3.13 Space Required by a Compressed Trie

Since each branch node partitions the elements in its subtrie into two or more nonempty groups, an n element compressed trie has at most $n-1$ branch nodes. Therefore, the space required by each of the compressed trie variants described by us is $O(nr)$, where r is the trie radix.

When compressed tries are represented as hash tables, we need an additional data structure to store the nonpointer fields of branch nodes. We may use an array for this purpose.

EXERCISES

1. (a). Draw the trie obtained for the following data:

   AMIOT, AVENGER, AVRO, HEINKEL, HELLDIVER, MACCHI, MARAUDER, MUSTANG, SPITFIRE, SYKHOI

   Sample the keys from left to right one character at a time.

   (b) Using single-character sampling, obtain a trie with the fewest number of levels.

2. Explain how a trie could be used to implement a spelling checker.
3. Explain how a trie could be used to implement an auto-command completion program. Such a program would maintain a library of valid commands. It would then accept a user command, character by character, from a keyboard. When a sufficient number of characters had been input to uniquely identify the command, it would display the complete command on the computer monitor.
4. Write an algorithm to insert a key value x into a trie in which the keys are sampled from left to right, one character at a time.
5. Do Exercise 4 with the added assumption that the trie is to have no more than six levels. Synonyms are to be packed into the same element node.
6. Write an algorithm to delete x from a trie under the assumptions of Exercise 4. Assume that each branch node has a *count* data member equal to the number of element nodes in the subtrie for which it is the root.
7. Do Exercise 6 for the trie of Exercise 5.
8. In the trie of Figure 12.13 the nodes δ_1 and δ_2 each have only one child. Branch nodes with only one child may be eliminated from tries by maintaining a *skip* data member with each node. The value of this data member equals the number of characters to be skipped before obtaining the next character to be sampled. Thus, we can have $skip[\delta_3] = 2$ and delete the nodes δ_1 and δ_2. Write algorithms to search, insert, and delete from tries in which each branch node has a *skip* data

member.

9. Assume that the branch nodes of a compressed trie are represented using a hash table (one for each node). Each such hash table is augmented with a count and skip value as described above. Describe how this change to the node structure affects the time and space complexity of the trie data structure.

10. Do the previous exercise for the case when each branch node is represented by a chain in which each node has two data members: *pointer* and *link*, where *pointer* points to a subtrie and *link* points to the next node in the chain. The number of nodes in the chain for any branch node equals the number of non-0 pointers in that node. Each chain is augmented by a skip value. Draw the chain representation of the compressed version of the trie of Figure 12.6.

12.4 SUFFIX TREES

12.4.1 Have You Seen This String?

In the classical *substring search* problem, we are given a string S and a pattern P and are to report whether or not the pattern P occurs in the string S. For example, the pattern P = *cat* appears (twice) in the string $S1$ = *The big cat ate the small catfish.*, but does not appear in the string $S2$ = *Dogs for sale.*.

Researchers in the human genome project, for example, are constantly searching for substrings/patterns (we use the terms substring and pattern interchangeably) in a gene databank that contains tens of thousands of genes. Each gene is represented as a sequence or string of letters drawn from the alphabet A,C,G,T. Although most of the strings in the databank are around 2000 letters long, some have tens of thousands of letters. Because of the size of the gene databank and the frequency with which substring searches are done, it is imperative that we have as fast an algorithm as possible to locate a given substring within the strings in the databank.

We can search for a pattern P in a string S using Program 2.16. The complexity of such a search is $O(|P|+|S|)$, where $|P|$ denotes the length (i.e., number of letters or digits) of P. This complexity looks pretty good when you consider that the pattern P could appear anywhere in the string S. Therefore, we must examine every letter/digit (we use the terms letter and digit interchangeably) of the string before we can conclude that the search pattern does not appear in the string. Further, before we can conclude that the search pattern appears in the string, we must examine every digit of the pattern. Hence, every pattern search algorithm must take time that is linear in the lengths of the pattern and the string being searched.

When classical pattern matching algorithms are used to search for several patterns $P_1, P_2, \cdots, P_k$ in the string S, $O(|P_1| + |P_2| + \cdots + |P_k| + k|S|)$ time is taken (because $O(|P_i|+|S|)$ time is taken to seach for P_i). The suffix tree data structure that we are about to study reduces this complexity to $O(|P_1| + |P_2| + \cdots + |P_k| + |S|)$.

Of this time, $O(|S|)$ time is spent setting up the suffix tree for the string S; an individual pattern search takes only $O(|P_i|)$ time (after the suffix tree for S has been built). Therefore once the suffix tree for S has been created, the time needed to search for a pattern depends only on the length of the pattern.

12.4.2 The Suffix Tree Data Structure

The *suffix tree* for S is actually the compressed trie for the nonempty suffixes of the string S. Since a suffix tree is a compressed trie, we sometimes refer to the tree as a trie and to its subtrees as subtries.

The (nonempty) suffixes of the string S = *peeper* are *peeper*, *eeper*, *eper*, *per*, *er*, and *r*. Therefore, the suffix tree for the string *peeper* is the compressed trie that contains the elements (which are also the keys) *peeper*, *eeper*, *eper*, *per*, *er*, and *r*. The alphabet for the string *peeper* is *e*,*p*,*r*. Therefore, the radix of the compressed trie is 3. If necessary, we may use the mapping $e \rightarrow 0$, $p \rightarrow 1$, $r \rightarrow 2$, to convert from the letters of the string to numbers. This conversion is necessary only when we use a node structure in which each node has an array of child pointers. Figure 12.27 shows the compressed trie (with labeled edges) for the suffixes of *peeper*. This compressed trie is also the suffix tree for the string *peeper*.

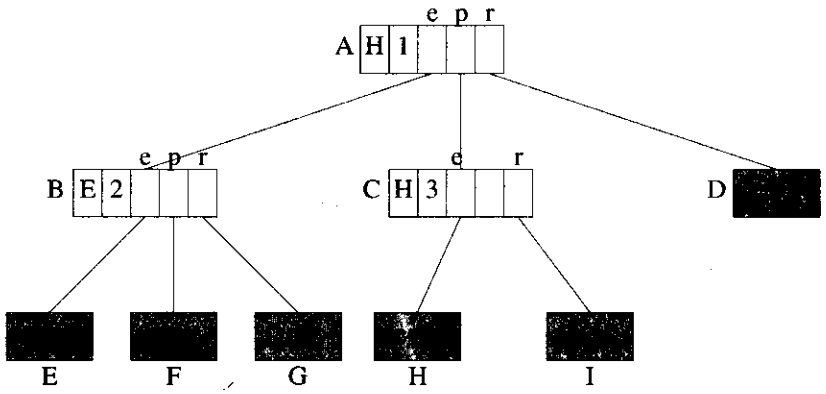


Figure 12.27: Compressed trie for the suffixes of *peeper*

Since the data in the element nodes *D*–*I* are the suffixes of *peeper*, each element node need retain only the start index of the suffix it contains. When the letters in *peeper* are indexed from left to right beginning with the index 1, the element nodes *D*–*I* need

only retain the indexes 6, 2, 3, 5, 1, and 4, respectively. Using the index stored in an element node, we can access the suffix from the string S. Figure 12.28 shows the suffix tree of Figure 12.27 with each element node containing a suffix index.

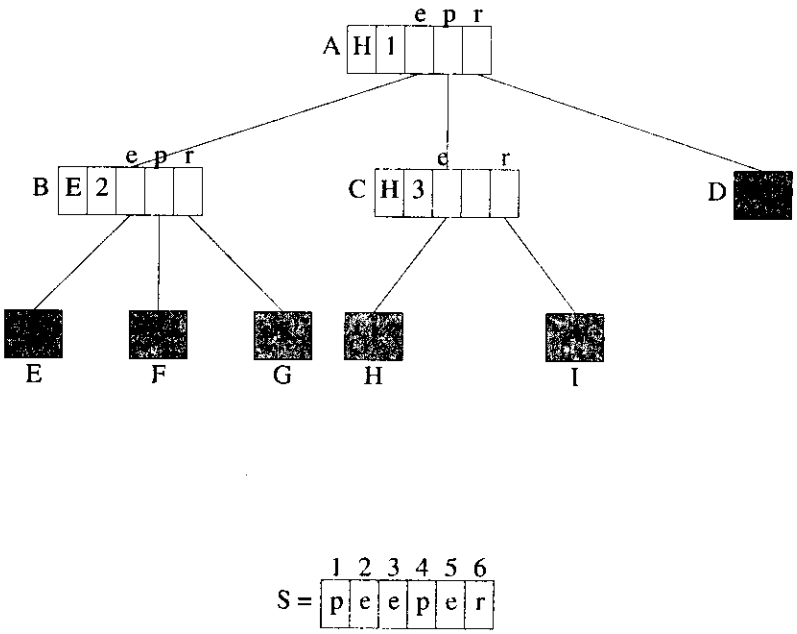


Figure 12.28: Modified compressed trie for the suffixes of *peeper*

The first component of each branch node is a reference to an element in that subtrie. We may replace the element reference by the index of the first digit of the referenced element. Figure 12.29 shows the resulting compressed trie. We shall use this modified form as the representation for the suffix tree.

When describing the search and construction algorithms for suffix trees, it is easier to deal with a drawing of the suffix tree in which the edges are labeled by the digits used in the move from a branch node to a child node. The first digit of the label is the digit used to determine which child is moved to, and the remaining digits of the label give the digits that are skipped over. Figure 12.30 shows the suffix tree of Figure 12.29 drawn in this manner.

In the more humane drawing of a suffix tree, the labels on the edges on any root to element node path spell out the suffix represented by that element node. When the digit number for the root is not 1, the humane drawing of a suffix tree includes a header node

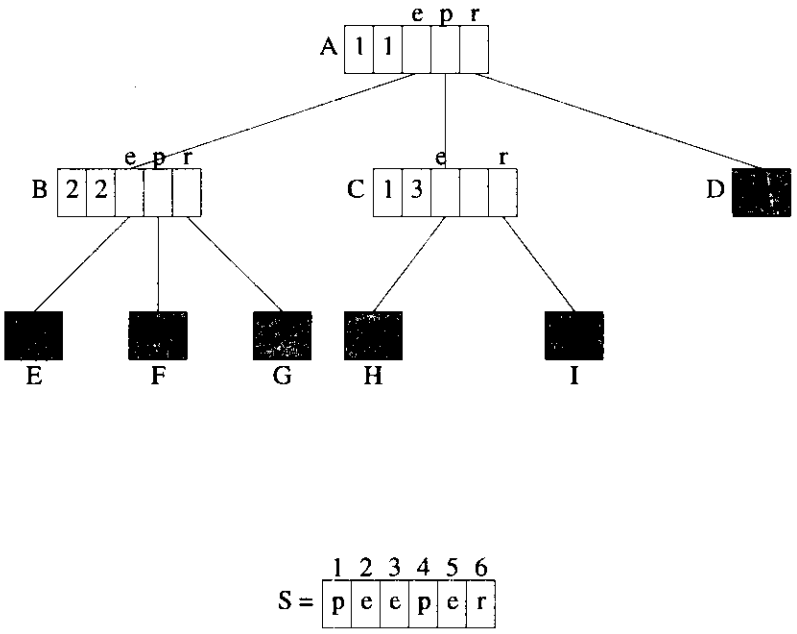


Figure 12.29: Suffix tree for *peeper*

with an edge to the former root. This edge is labeled with the digits that are skipped over.

The string *represented* by a node of a suffix tree is the string formed by the labels on the path from the root to that node. Node *A* of Figure 12.30 represents the empty string ε, node *C* represents the string *pe*, and node *F* represents the string *eper*.

Since the keys in a suffix tree are of different length, we must ensure that no key is a proper prefix of another. Whenever the last digit of string *S* appears only once in *S*, no suffix of *S* can be a proper prefix of another suffix of *S*. In the string *peeper*, the last digit is *r*, and this digit appears only once. Therefore, no suffix of *peeper* is a proper prefix of another. The last digit of *data* is *a*, and this last digit appears twice in *data*. Therefore, *data* has two suffixes *ata* and *a* that begin with *a*. The suffix *a* is a proper prefix of the suffix *ata*.

When the last digit of the string *S* appears more than once in *S* we must append a new digit (say #) to the suffixes of *S* so that no suffix is a prefix of another. Optionally, we may append the new digit to *S* to get the string *S#*, and then construct the suffix tree for *S#*. When this optional route is taken, the suffix tree has one more suffix (#) than the suffix tree obtained by appending the symbol # to the suffixes of *S*.

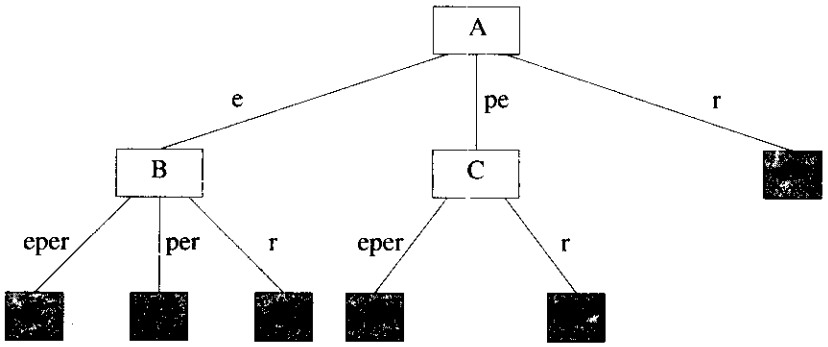


Figure 12.30 A more humane drawing of a suffix tree

12.4.3 Let's Find That Substring (Searching a Suffix Tree)

But First, Some Terminology

Let $n=|S|$ denote the length (i.e., number of digits) of the string whose suffix tree we are to build. We number the digits of S from left to right beginning with the number 1. $S[i]$ denotes the ith digit of S, and *suffix* (i) denotes the suffix $S[i]\cdots S[n]$ that begins at digit i, $1\le i\le n$.

On With the Search

A fundamental observation used when searching for a pattern P in a string S is that P appears in S (i.e., P is a substring of S) iff P is a prefix of some suffix of S.

Suppose that $P=P[1]\cdots P[k] = S[i]\cdots S[i+k-1]$. Then, P is a prefix of *suffix* (i). Since *suffix* (i) is in our compressed trie (i.e., suffix tree), we can search for P by using the strategy to search for a key prefix in a compressed trie.

Let's search for the pattern $P = per$ in the string $S = peeper$. Imagine that we have already constructed the suffix tree (Figure 12.30) for *peeper*. The search starts at the root node A. Since $P[1] = p$, we follow the edge whose label begins with the digit p. When following this edge, we compare the remaining digits of the edge label with successive digits of P. Since these remaining label digits agree with the pattern digits, we reach the branch node C. In getting to node C, we have used the first two digits of the pattern. The third digit of the pattern is r, and so, from node C we follow the edge whose label begins with r. Since this edge has no additional digits in its label, no additional digit comparisons are done and we reach the element node I. At this time, the digits in the pattern have been exhausted and we conclude that the pattern is in the string. Since an element node

is reached, we conclude that the pattern is actually a suffix of the string *peeper*. In the actual suffix tree representation (rather than in the humane drawing), the element node I contains the index 4 which tells us that the pattern $P = per$ begins at digit 4 of *peeper* (i.e., $P = suffix(4)$). Further, we can conclude that *per* appears exactly once in *peeper*; the search for a pattern that appears more than once terminates at a branch node, not at an element node.

Now, let us search for the pattern $P = eeee$. Again, we start at the root. Since the first character of the pattern is e, we follow the edge whose label begins with e and reach the node B. The next digit of the pattern is also e, and so, from node B we follow the edge whose label begins with e. In following this edge, we must compare the remaining digits *per* of the edge label with the following digits *ee* of the pattern. We find a mismatch when the first pair (p,e) of digits are compared and we conclude that the pattern does not appear in *peeper*.

Suppose we are to search for the pattern $P = p$. From the root, we follow the edge whose label begins with p. In following this edge, we compare the remaining digits (only the digit e remains) of the edge label with the following digits (there aren't any) of the pattern. Since the pattern is exhausted while following this edge, we conclude that the pattern is a prefix of all keys in the subtrie rooted at node C. We can find all occurrences of the pattern by traversing the subtrie rooted at C and visiting the information nodes in this subtrie. If we want the location of just one of the occurrences of the pattern, we can use the index stored in the first component of the branch node C (see Figure 12.29). When a pattern exhausts while following the edge to node X, we say that node X has been reached; the search terminates at node X.

When searching for the pattern $P = rope$, we use the first digit r of P and reach the element node D. Since the the pattern has not been exhausted, we must check the remaining digits of the pattern against those of the key in D. This check reveals that the pattern is not a prefix of the key in D, and so the pattern does not appear in *peeper*.

The last search we are going to do is for the pattern $P = pepe$. Starting at the root of Figure 12.30, we move over the edge whose label begins with p and reach node C. The next unexamined digit of the search pattern is p. So, from node C, we wish to follow the edge whose label begins with p. Since no edge satisfies this requirement, we conclude that *pepe* does not appear in the string *peeper*.

12.4.4 Other Nifty Things You Can Do with a Suffix Tree

Once we have set up the suffix tree for a string S, we can tell whether or not S contains a pattern P in $O(|P|)$ time. This means that if we have a suffix tree for the text of Shakespeare's play "Romeo and Juliet," we can determine whether or not the phrase *wherefore art thou* appears in this play with lightning speed. In fact, the time taken will be that needed to compare up to 18 (the length of the search pattern) letters/digits. The search time is independent of the length of the play.

Some other interesting things you can do at lightning speed are described below.

Find all occurrences of a pattern P.
This is done by searching the suffix tree for P. If P appears at least once, the search terminates successfully either at an element node or at a branch node. When the search terminates at an element node, the pattern occurs exactly once. When we terminate at a branch node X, all places where the pattern occurs can be found by visiting the element nodes in the subtrie rooted at X. This visiting can be done in time linear in the number of occurrences of the pattern if we

(a) Link all of the element nodes in the suffix tree into a chain, the linking is done in lexicographic order of the represented suffixes (which also is the order in which the element nodes are encountered in a left to right scan of the element nodes). The element nodes of Figure 12.30 will be linked in the order E,F,G,H,I,D.

(b) In each branch node, keep a reference to the first and last element node in the subtrie of which that branch node is the root. In Figure 12.30, nodes A, B, and C keep the pairs (E,D), (E,G), and (H,I), respectively. We use the pair (*firstInformationNode*,*lastInformationNode*) to traverse the element node chain starting at *firstInformationNode* and ending at *lastInformationNode*. This traversal yields all occurrences of patterns that begin with the string spelled by the edge labels from the root to the branch node. Notice that when (*firstInformationNode*,*lastInformationNode*) pairs are kept in branch nodes, we can eliminate the branch node field that keeps a reference to an element node in the subtrie (i.e., the field *element*).

Find all strings that contain a pattern P.
Suppose we have a collection $S1, S2, \cdots, Sk$ of strings and we wish to report all strings that contain a query pattern P. For example, the genome databank contains tens of thousands of strings, and when a researcher submits a query string, we are to report all databank strings that contain the query string. To answer queries of this type efficiently, we set up a compressed trie (we may call this a *multiple string suffix tree*) that contains the suffixes of the string $S1\$S2\$...\$Sk\#$, where $\$$ and $\#$ are two different digits that do not appear in any of the strings $S1, S2, \cdots, Sk$. In each node of the suffix tree, we keep a list of all strings Si that are the start point of a suffix represented by an element node in that subtrie.

Find the longest substring of S **that appears at least** $m>1$ **times.**
This query can be answered in $O(|S|)$ time in the following way:

(a) Traverse the suffix tree labeling the branch nodes with the sum of the label lengths from the root and also with the number of information nodes in the subtrie.

(b) Traverse the suffix tree visiting branch nodes with element node count $\geq m$

Determine the visited branch node with longest label length.

Note that step (a) needs to be done only once. Following this, we can do step (b) for as many values of m as is desired. Also, note that when $m = 2$ we can avoid determining the number of element nodes in subtries. In a compressed trie, every subtrie rooted at a branch node has at least two element nodes in it.

Find the longest common substring of the strings S and T.
This can be done in time $O(|S|+|T|)$ as below:

(a) Construct a multiple string suffix tree for S and T (i.e., the suffix tree for $S\$T\#$).

(b) Traverse the suffix tree to identify the branch node for which the sum of the label lengths on the path from the root is maximum and whose subtrie has at least one information node that represents a suffix that begins in S and at least one information node that represents a suffix that begins in T.

EXERCISES

1. Draw the suffix tree for $S = ababab\#$.
2. Draw the suffix tree for $S = aaaaaa\#$.
3. Draw the multiple string suffix tree for $S1 = abba$, $S2 = bbbb$, and $S3 = aaaa$.

12.5 TRIES AND INTERNET PACKET FORWARDING

12.5.1 IP Routing

In the Internet, data packets are transported from source to destination by a series of routers. For example, a packet that originates in New York and is destined for Los Angeles will first be processed by a router in New York. This router may forward the packet to a router in Chicago, which, in turn, may foward the packet to a router in Denver. Finally, the router in Denver may forward the packet to Los Angeles. Each router moves a packet one step closer to its destination. A router does this by examining the destination address in the header of the packet to be routed. Using this destination address and a collection of forwarding rules stored in the router, the router decides where to send the packet next.

An Internet router table is a collection of rules of the form (P, NH), where where P is a prefix and NH is the next hop; NH is the next hop for packets whose destination address has the prefix P. For example, the rule $(01*, a)$ states that the next hop for packets whose destination address (in binary) begins with 01 is a. In IPv4 (Internet Protocol

version 4), destination addresses are 32 bits long. So, P may be up to 32 bits long. In IPv6, destination addresses are 128 bits long and so, P may be up to 128 bits in length.

It is not uncommon for a destination address to be matched by more than 1 rule in a commercial router table. In this case, the next hop is determined by the matching rule that has the longest prefix. So, for example, suppose that (01*,a) and (0100*,b) are the only two rules in our router table that match a packet whose destination address begins with the bit sequence 0100. The next hop for this packet is b. In other words, packet forwarding in the Internet is done by determining the longest matching-prefix.

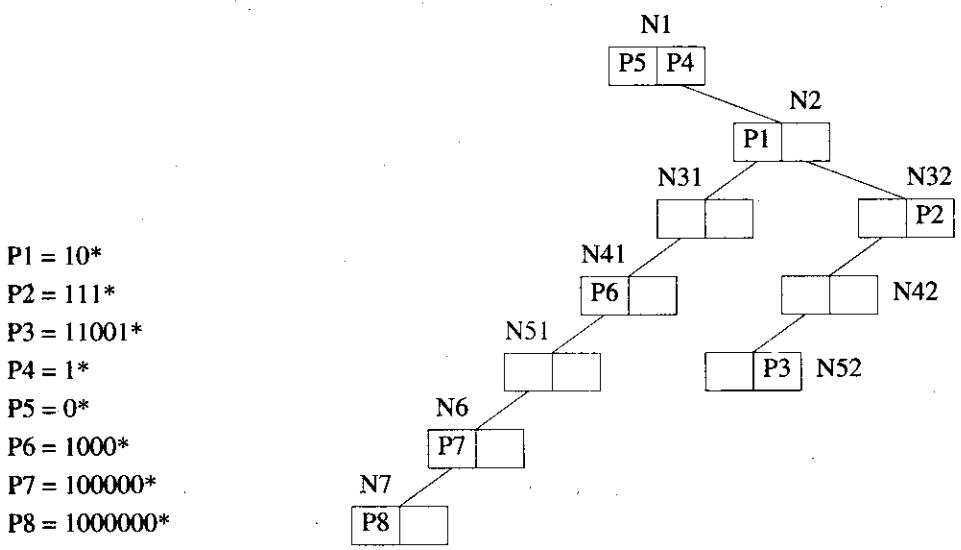
Although Internet router tables are dynamic in practice (i.e., the rule set changes in time; rules are added and deleted as routers come online and go offline), data structures for Internet router tables often are optimized for the search operation—given a destination address, determine the next hop for the longest matching-prefix.

12.5.2 1-Bit Tries

A *1-bit trie* is very similar to a binary trie. It is a tree-like structure in which each node has a left child, left data, right child, and right data field. Nodes at level l of the trie store prefixes whose length is l. If the rightmost bit in a prefix whose length is l is 0, the prefix is stored in the left data field of a node that is at level l; otherwise, the prefix is stored in the right data field of a node that is at level l. At level i of a trie, branching is done by examining bit i (bits are numbered from left to right beginning with the number 1) of a prefix or destination address. When bit i is 0, we move into the left subtree; when the bit is 1, we move into the right subtree. Figure 12.31(a) gives a set of 8 prefixes, and Figure 12.31(b) shows the corresponding 1-bit trie. The height of a 1-bit trie is $O(W)$, where W is the length of the longest prefix in the router table. Note that $W \leq 32$ for IPv4 tables and $W \leq 128$ for IPv6 tables. Note also that there is no place, in a 1-bit trie, to store the prefix * whose length is zero. This doesn't lead to any difficulty as this prefix matches every destination address. In case a search of a 1-bit trie fails to find a matching prefix, the next-hop associated with * is used.

For any destination address d, all prefixes that match d lie on the search path determined by the bits of d. By following this search path, we may determine the longest matching-prefix in $O(W)$ time. Further, prefixes may be inserted/deleted in $O(W)$ time. The memory required by a 1-bit trie is $O(nW)$, where n is the number of rules in the router table.

Although the algorithms to search, insert and delete using a 1-bit trie are simple and of seemingly low complexity, $O(W)$, the demands of the Internet make the 1-bit trie impractical. Using trie-like structures, most of the time spent searching for the next hop goes to memory acceses. Hence, when analyzing the complexity of trie data structures for router tables, we focus on the number of memory accesses. When a 1-bit trie is used, it may take us up to W memory accesses to determine the next hop for a packet. Recall that $W \leq 32$ for IPv4 and $W \leq 128$ for IPv6. To keep the Internet operating smoothly, it



(a) 8 prefixes

(b) Corresponding 1-bit trie

Figure 12.31: Prefixes and corresponding 1-bit trie

is necessary that the next hop for each packet be determined using far fewer memory accesses than W. In practice, we must determine the next hop using at most (say) 6 memory accesses.

12.5.3 Fixed-Stride Tries

Since the trie of Figure 12.31(b) has a height of 7, a search into this trie may make up to 7 memory accesses, one access for each node on the path from the root to a node at level 7 of the trie. The total memory required for the 1-bit trie of Figure 12.31(b) is 20 units (each node requires 2 units, one for each pair of (child, data) fields). We may reduce the height of the router-table trie at the expense of increased memory requirement by increasing the branching factor at each node, that is, we use a multiway trie. The *stride* of a node is defined to be the number of bits used at that node to determine which branch to take. A node whose stride is s has 2^s child fields (corresponding to the 2^s possible values for the s bits that are used) and 2^s data fields. Such a node requires 2^s memory

units. In a *fixed-stride trie* (FST), all nodes at the same level have the same stride; nodes at different levels may have different strides.

Suppose we wish to represent the prefixes of Figure 12.31(a) using an FST that has three levels. Assume that the strides are 2, 3, and 2. The root of the trie stores prefixes whose length is 2; the level two nodes store prefixes whose length is 5 (2 + 3); and level three nodes store prefixes whose length is 7 (2 + 3 + 2). This poses a problem for the prefixes of our example, because the length of some of these prefixes is different from the storeable lengths. For instance, the length of P5 is 1. To get around this problem, a prefix with a nonpermissible length is expanded to the next permissible length. For example, P5 = 0* is expanded to P5a = 00* and P5b = 01*. If one of the newly created prefixes is a duplicate, natural dominance rules are used to eliminate all but one occurrence of the prefix. For instance, P4 = 1* is expanded to P4a = 10* and P4b = 11*. However, P1 = 10* is to be chosen over P4a = 10*, because P1 is a longer match than P4. So, P4a is eliminated. Because of the elimination of duplicate prefixes from the expanded prefix set, all prefixes are distinct. Figure 12.32(a) shows the prefixes that result when we expand the prefixes of Figure 12.31 to lengths 2, 5, and 7. Figure 12.32(b) shows the corresponding FST whose height is 3 and whose strides are 2, 3, and 2.

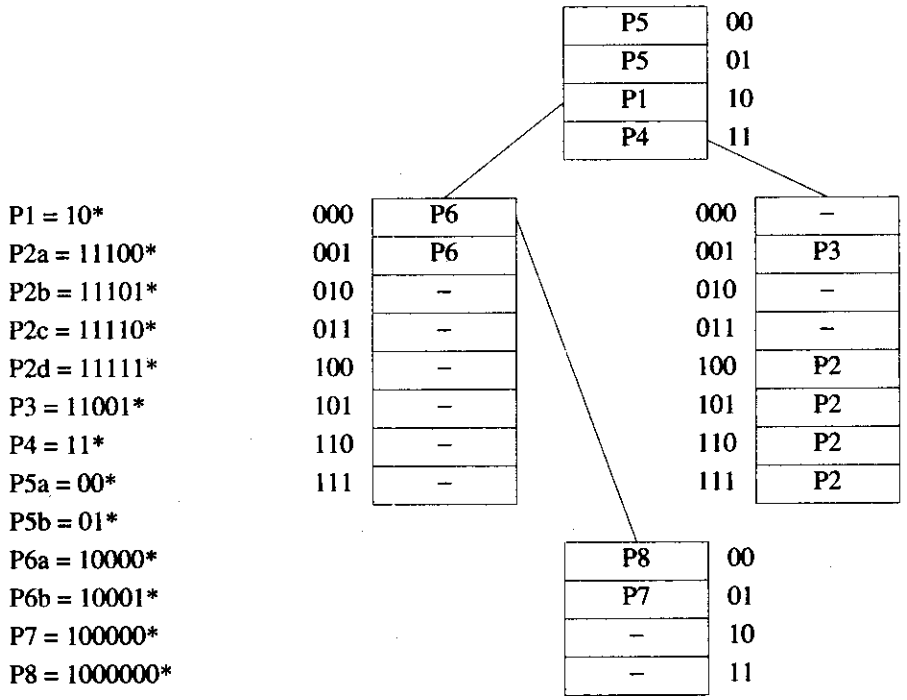
Since the trie of Figure 12.32(b) can be searched with at most 3 memory accesses, it represents a time performance improvement over the 1-bit trie of Figure 12.31(b), which requires up to 7 memory references to perform a search. However, the space requirements of the FST of Figure 12.32(b) are more than that of the corresponding 1-bit trie. For the root of the FST, we need 8 fields or 4 units; the two level 2 nodes require 8 units each; and the level 3 node requires 4 units. The total is 24 memory units.

We may represent the prefixes of Figure 12.31(a) using a one-level trie whose root has a stride of 7. Using such a trie, searches could be performed making a single memory access. However, the one-level trie would require $2^7 = 128$ memory units.

In the *fixed-stride trie optimization* (FSTO) problem, we are given a set P of prefixes and an integer k. We are to select the strides for a k-level FST in such a manner that the k-level FST for the given prefixes uses the smallest amount of memory.

For some P, a k-level FST may actually require more space than a $(k-1)$-level FST. For example, when $P = \{00^*, 01^*, 10^*, 11^*\}$, the unique 1-level FST for P requires 4 memory units while the unique 2-level FST (which is actually the 1-bit trie for P) requires 6 memory units. Since the search time for a $(k-1)$-level FST is less than that for a k-level tree, we would actually prefer $(k-1)$-level FSTs that take less (or even equal) memory over k-level FSTs. Therefore, in practice, we are really interested in determining the best FST that uses at most k levels (rather than exactly k levels). The *modified* MSTO problem (MFSTO) is to determine the best FST that uses at most k levels for the given prefix set P.

Let O be the 1-bit trie for the given set of prefixes, and let F be any k-level FST for this prefix set. Let $s_1, \cdots, s_k$ be the strides for F. We shall say that level j, $1 \le j \le k$, of F covers levels $a, \cdots, b$ of O, where $a = \sum_{1}^{j-1} s_q + 1$ and $b = \sum_{1}^{j} s_q$. So, level 1 of the FST of



(a) Expanded prefixes (b) Corresponding fixed-stride trie

Figure 12.32: Prefix expansion and fixed-stride trie

Figure 12.32(b) covers levels 1 and 2 of the 1-bit trie of Figure 12.31(b). Level 2 of this FST covers levels 3, 4, and 5 of the 1-bit trie of Figure 12.31(b); and level 3 of this FST covers levels 6 and 7 of the 1-bit trie. We shall refer to levels $e_u = \sum_{1}^{u} s_q$, $1 \leq u \leq k$ as the *expansion levels* of O. The expansion levels defined by the FST of Figure 12.32(b) are 1, 3, and 6.

Let *nodes* (i) be the number of nodes at level i of the 1-bit trie O. For the 1-bit trie of Figure 12.31(a), *nodes* (1:7) = [1,1,2,2,2,1,1]. The memory required by F is $\sum_{1}^{k} nodes(e_q) * 2^{s_q}$. For example, the memory required by the FST of Figure 12.32(b) is $nodes(1)*2^2 + nodes(3)*2^3 + nodes(6)*2^2 = 24$.

Let $T(j,r)$ be the best (i.e., uses least memory) FST that uses *at most* r expansion levels and covers levels 1 through j of the 1-bit trie O. Let $C(j,r)$ be the cost (i.e., memory requirement) of $T(j,r)$. So, $T(W,k)$ is the best FST for O that uses at most k expansion levels and $C(W,k)$ is the cost of this FST. We observe that the last expansion level in $T(j,r)$ covers levels $m+1$ through j of O for some m in the range 0 through $j-1$ and the remaining levels of this best FST define $T(m,r-1)$. So,

$$C(j,r) = \min_{0 \le m < j} \{C(m,r-1) + nodes(m+1) * 2^{j-m+1}\},\ j \ge 1,\ r > 1 \tag{12.1}$$

$$C(0,r) = 0 \text{ and } C(j,1) = 2^j,\ j \ge 1 \tag{12.2}$$

Let $M(j,r)$, $r > 1$, be the smallest m that minimizes

$$C(m,r-1) + nodes(m+1) * 2^{j-m+1},$$

in Eq. 12.1. Eqs. 12.1 and 12.2 result in an algorithm to compute $C(W,k)$ in $O(kW^2)$. The $M(j,r)$s may be computed in the same amount of time while we are computing the $C(j,r)$s. Using the computed M values, the strides for the optimal FST that uses at most k expansion levels may be determined in an additional $O(k)$ time.

12.5.4 Variable-Stride Tries

In a *variable-stride trie* (VST) nodes at the same level may have different strides. Figure 12.33 shows a two-level VST for the 1-bit trie of Figure 12.31. The stride for the root is 2; that for the left child of the root is 5; and that for the root's right child is 3. The memory required by this VST is 4 (root) + 32 (left child of root) + 8 (right child of root) = 44.

Since FSTs are a special case of VSTs, the memory required by the best VST for a given prefix set P and number of expansion levels k is less than or equal to that required by the best FST for P and k.

Let r-VST be a VST that has at most r levels. Let $Opt(N,r)$ be the cost (i.e., memory requirement) of the best r-VST for a 1-bit trie whose root is N. The root of this best VST covers levels 1 through s of O for some s in the range 1 through $height(N)$ and the subtries of this root must be best $(r-1)$-VSTs for the descendents of N that are at level $s+1$ of the subtree rooted at N. So,

$$Opt(N,r) = \min_{1 \le s \le height(N)} \{2^s + \sum_{M \in D_{s+1}(N)} Opt(M,r-1)\},\ r > 1 \tag{12.3}$$

where $D_s(N)$ is the set of all descendents of N that are at level s of N. For example,

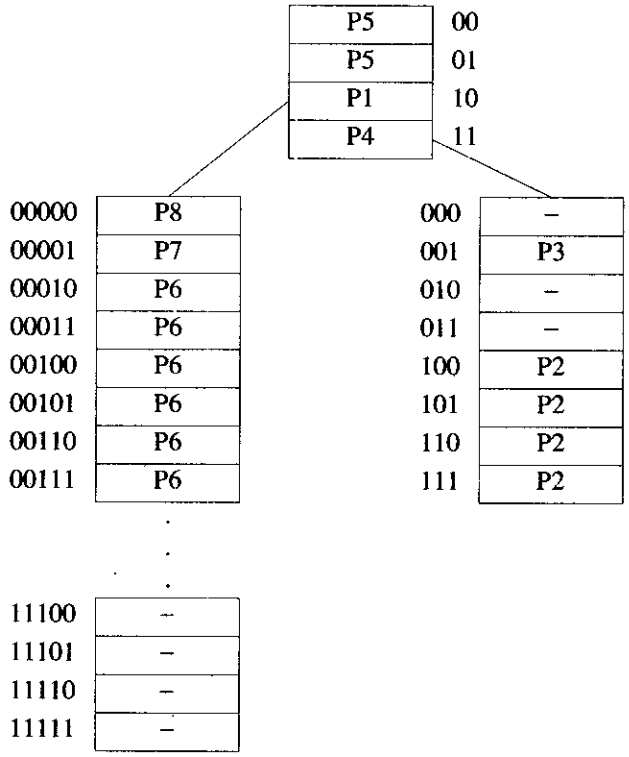


Figure 12.33: Two-level VST for prefixes of Figure 12.31(a)

$D_2(N)$ is the set of children of N and $D_3(N)$ is the set of grandchildren of N. *height* (N) is the maximum level at which the trie rooted at N has a node. For example, in Figure 12.31(b), the height of the trie rooted at N1 is 7. When r=1,

$$Opt\,(N, 1)=2^{height\,(N)}. \tag{12.4}$$

Let

$$Opt\,(N,s,r) = \sum_{M \in D_s(N)} Opt\,(M,r),\ s > 1,\ r > 1,$$

and let $Opt\,(N, 1,r)=Opt\,(N,r)$. From Eqs. 12.3 and 12.4, it follows that:

$$Opt(N,1,r) = \min_{1 \le s \le height(N)} \{2^s + Opt(N,s+1,r-1)\},\ r > 1 \tag{12.5}$$

and

$$Opt(N,1,1) = 2^{height(N)}. \tag{12.6}$$

For $s > 1$ and $r > 1$, we get

$$\begin{aligned} Opt(N,s,r) &= \sum_{M \in D_s(N)} Opt(M,r) \\ &= Opt(LeftChild(N),s-1,r) \\ &\quad + Opt(RightChild(N),s-1,r). \end{aligned} \tag{12.7}$$

For Eq. 12.7, we need the following initial condition:

$$Opt(null,*,*) = 0 \tag{12.8}$$

For an n-rule router table, the 1-bit trie O has $O(nW)$ nodes. So, the number of $Opt(*,*,*)$ values is $O(nW^2k)$. Each $Opt(*,s,*)$, $s > 1$, value may be computed in $O(1)$ time using Eqs. 12.7 and 12.8 provided the Opt values are computed in postorder. The $Opt(*,1,*)$ values may then be computed in $O(W)$ time each using Eqs. 12.5 and 12.6. Therefore, we may compute $Opt(R,k) = Opt(R,1,k)$, where R is the root of O, in $O(nW^2k)$ time. If we keep track of the s that minimizes the right side of Eq. 12.5 for each pair (N,r), we can determine the strides of all nodes in the optimal k-VST is an additional $O(nW)$ time.

EXERCISES

1. (a) Write a C function to compute $C(j,r)$ for $0 \le j \le W$ and $1 \le r \le k$ using Eqs. 12.1 and 12.2. Your function should compute $M(j,r)$ as well. The complexity of your function should be $O(kW^2)$. Show that this is the case.

   (b) Write a C function that determines the strides of all levels in the best FST that has at most k levels. Your function should use the M values computed in part (a). The complexity of your function should be $O(k)$. Show that this is the case.

2. (a) Write a C function to compute $Opt(N,s,r)$ for $1 \le s \le W$, $1 \le r \le k$ and all nodes N of the 1-bit trie O. You should use Eqs. 12.5 through 12.8. Your function should compute $S(N,r)$, which is the s value that minimizes the right side of Eq. 12.5, as well. The complexity of your function should be $O(nW^2k)$, where n is the number of rules in the router table. Show that this is the case.

   (b) Write a C function that determines the strides of all nodes in the best k-VST for O. Your function should use the S values computed in part (a). The complexity of your function should be $O(nW)$. Show that this is the case.

12.6 REFERENCES AND SELECTED READINGS

Digital search trees were first proposed by E. Coffman and J. Eve in *CACM*, 13, 1970, pp. 427-432. The structure Patricia was developed by D. Morrison. Digital search trees, tries, and Patricia are analyzed in the book *The Art of Computer Programming: Sorting and Searching*, Second Edition, by D. Knuth, Addison-Wesley, Reading, MA, 1998.

You can learn more about the genome project and genomic applications of pattern matching from the following Web sites: http://www.nhgri.nih.gov/HGP/ (NIH's Web site for the human genome project); http://www.ornl.gov/TechResources/Human_Genome/home.html (Department of Energy's Web site for the human genomics project); and http://merlin.mbcr.bcm.tmc.edu:8001/bcd/Curric/welcome.html; (Biocomputing Hypertext Coursebook).

Linear time algorithms to search for a single pattern in a given string can be found in most algorithm's texts. See, for example, the texts: *Computer Algorithms*, by E. Horowitz, S. Sahni, and S. Rajasekeran, Computer Science Press, New York, 1998 and *Introduction to Algorithms*, Second Edition, by T. Cormen, C. Leiserson, R. Rivest and C. Stein, McGraw-Hill Book Company, New York, 2002.

For more on suffix tree construction, see the papers: "A space economical suffix tree construction algorithm," by E. McCreight, *Journal of the ACM*, 23, 2, 1976, 262-272; "Fast string searching with suffix trees," by M. Nelson, *Dr. Dobb's Journal*, August 1996. and "Suffix trees and suffix arrays," by S. Aluru, in *Handbook of data structures and applications*, D. Mehta and S. Sahni, editors, Chapman & Hall/CRC, 2005.

You can download code to construct a suffix tree from http://www.ddj.com/ftp/1996/1996.08/suffix.zip.

The use of fixed- and variable-stride tries for IP router tables was first proposed in the paper "Faster IP lookups using controlled prefix expansion," by V. Srinivasan and G. Varghese, *ACM Transactions on Computer Systems*, Feb., 1999. Our dynamic programming formulations for fixed- and variable-stride tries are from "Efficient construction of multibit tries for IP lookup," by S. Sahni and K. Kim, *IEEE/ACM Transactions*

on Networking, 2003. For more on data structures for IP router tables and packet classification, see "IP router tables," by S. Sahni, K. Kim and H. Lu and "Multi-dimensional packet classification," by P. Gupta, in *Handbook of data structures and applications*, D. Mehta and S. Sahni, editors, Chapman & Hall/CRC, 2005.

INDEX