Chapter 5

Tries

A *trie* (pronounced "try") is a data structure that supports efficient FIND, INSERT, and DELETE operations for keys that can be represented by a unique string. In fact, the time complexity of *any* of these operations is $O(1)$ with respect to the number of keys present in the trie. The factor that determines the worst-case time it takes to perform any of these operations on a trie is the length of the key. For example, if we used a trie to implement a telephone directory, it would take the same amount of time to find "Ellard, Dan" whether we were looking in the Harvard telephone book or a telephone book for the entire United States. It could take twice as long to look up "Titmouse, Barneswoggle", however, simply because this name is twice the length of "Ellard, Dan".

5.1 Tries as an Abstract Data Type

As an abstract data type, a trie supports three fundamental operations:

- **FIND**(k)

  Determine whether k is present in the trie. Depending on the application of the ADT, **FIND** may either simply indicate whether or not k is present, or it may return a value v associated with the key k.

- **INSERT**(k, v)

  Add the key k to the trie. Depending on the application, a value v may or may not be supplied.

- **DELETE**(k)

  Remove the key k from the trie.

The domain of k (that is, the set of types that satisfy the constraints on k) is the set of finite strings from a specific alphabet. The domain of v is unrestricted.

The notion of strings used here is more general than the definition of strings in C, and is worth investigating. A *string* is defined to be to be a finite, ordered sequence of elements (called *characters* of a finite set (called the *alphabet*). In C, for example, the alphabet is the set of all possible `char`s, with the exception of the `char` 0, which is used to denote the end of a C string but is not considered part of the string), and each character is simply a non-zero `char`. However, other kinds of strings are quite useful and common (in the context of tries, but also in other contexts). For example, strings of bits (which have an alphabet of $\{0, 1\}$, and each character is a single bit) or strings of decimal numbers are often useful. For all of the examples in this section, we will use strings that have an alphabet consisting of the lowercase letters 'a' through 'z'. In the code samples, we will use the C convention of using a zero (`'\0'`) character to mark the end of a string, but note again that this character is not considered to be part of the string itself.

Note that the keys must be unique; while it is relatively easy to adapt most of the data structures that we have seen so far to work with multiple elements that have the same key, the same is not true for tries.[1]

As we will see later, most implementations of tries work best when the alphabet of their strings is small, although from the perspective of the ADT, there is no limitation placed on the size of the alphabet, as long as it is finite.

Tries also readily support some other operations, which will be discussed in section 5.6.

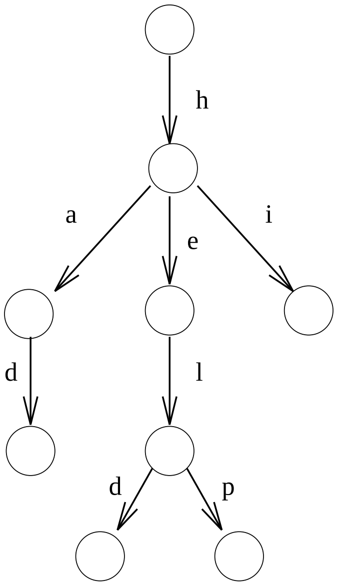
5.2 Tries as a Data Structure

As we have seen, when we search for a key in a binary search tree, at each node we visit, we compare the key in that node to the key that we are searching for. All that we care about is whether the key in the node is less than, equal to, or greater than the key we are searching for, however– the precise value of the node's key is not particularly interesting, unless it is an *exact* match.

If you think carefully about it, you will realize that this manner of search potentially throws away quite a bit of information at each comparison. The path from the root to any

[1]It is possible to represent duplicate keys in a trie, but the result is a more complicated data structure, whose design is left as an exercise.

Figure 5.1: A simple trie containing the keys "had", "held", "help", and "hi".



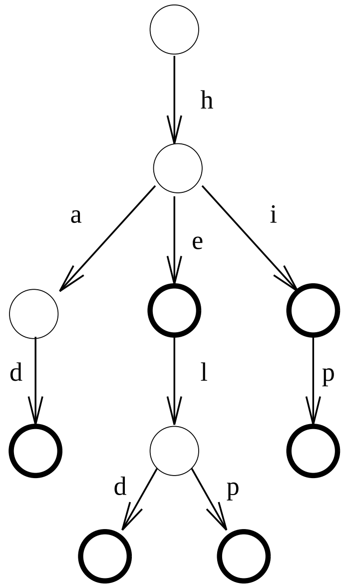
node in a binary search tree may represent a considerable amount of information about the contents of the node (and may, in some cases, specify it completely).

A *trie* is a data structure that stores the information about the contents of each node in the path from the root to the node, rather than the node itself. For example, consider the tree shown in figure 5.1. In this tree, each path between the root and a leaf represents a key, so this trie contains the keys "had", "held", "help", and "hi". Note that the nodes themselves are unlabeled. Instead, each *transition* between two nodes is labeled with a single character from a key.

However, this manner of representing keys has a fatal flaw; it is impossible to represent any key that is a prefix of another. This problem is fixed by introducing a special marker on each node that tells whether or not it represents the end of a key. In this text, a node represented by a light circle is an "internal" node, while a node represented by a dark circle represents the end of a key. In the code that is used to express the algorithms in this section, a node that represents the end of a key has a non-zero value in its `endOfKey` field. The trie shown in figure 5.2 contains the same keys as the trie in figure 5.1, with the addition of "he" and "hip".

The algorithm for determining whether or not a particular word is present in the trie is very simple, and is given in algorithm 5.1.

Figure 5.2: A trie containing the keys "had", "he", "held", "help", "hi", and "hip".



If the trie is used to store information about the keys (in addition to the keys themselves), then algorithm 5.1 can be modified to return this information instead of simply FOUND and NOT_FOUND.

The algorithm for inserting a new key is more involved. Like the algorithm for FIND, it traverses the trie until it either reaches the end of the string, or it discovers that the trie does not contain the string. In the first case, all that is necessary is to mark the end node as being the end of a key (and depositing a value into this node, if appropriate). In the second case, the trie must be extended with new nodes that represent the remainder of the string. A recursive algorithm for INSERT is shown in algorithm 5.2.

The algorithm for deleting a key is similar to insert, but has more special cases to consider. Like insert, it traverses the tree until the end of the key string is reached. If the end node has children, then this node cannot be removed, but all that needs to be done is to remove the marker that tells that it is the end of a key. If the node does not have any children, however, then it can be removed, as can the entire sequence of single-child, non-end-of-key nodes that lead to this node.

For example, in figure 5.3, deleting "are" causes nodes 3 and 4 to be deleted, but not 1 and 2 (because these represent part of the other keys represented by the trie). Deleting "at" from the original trie cannot delete any nodes, but simply removes the `endOfKey` marker on

Algorithm 5.1 (Trie FIND) *An algorithm to determine whether a specified key is present in a trie. Returns FOUND if the key is in the trie, NOT_FOUND otherwise.*

```
int trieFind (trie_t *root, char *key)
{
        int     i       = 0;
        trie_t  *curr   = root;
        int     key_len = strlen (key);

        if (root == NULL)
                return (NOT_FOUND);

        for (i = 0; i < key_len; i++) {
                curr = trieFindKthChild (curr, key[i]);
                if (curr == NULL)
                        return (NOT_FOUND);
        }

        if (trieIsEndOfKey (curr))
                return (FOUND);
        else
                return (NOT_FOUND);
}
```

Algorithm 5.2 (Trie INSERT) *An algorithm to insert a key into a trie.*

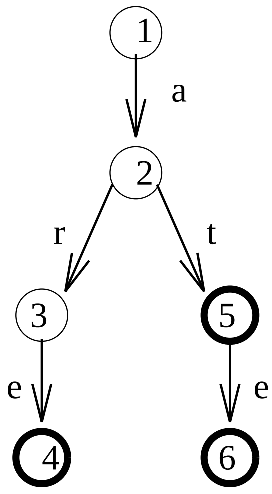
```
trie_t *trieInsert (trie_t *root, char *key)
{

        if (root == NULL)
                root = trieCreate ();

        if (key[0] == '\0') {
                trieEndOfKey (root, END_OF_KEY);
        }
        else {
                trie_t *child = trieInsert (trieFindKthChild (root, key[0]),
                                key + 1);
                trieInsertKthChild (root, key[0], child);
        }

        return (root);
}
```

Figure 5.3: A trie containing the keys "are", "at", and "ate".



node 5. Deleting "ate" from the original trie removes node 6, but cannot remove node 5 (even though node 5 has only a single child), because node 5 represents the end of a key.

5.3 Implementing Tries

The most obvious way of implementing a trie is as an n-ary tree, where n is the number of elements in the alphabet. In addition to the child pointers in each node, each node also contains a flag that indicates whether or not it is an end-of-key node, and the value associated with the key (if any). For example, if we wanted to construct a trie for the alphabet of the 26 lowercase letters, where each key has an integer associated with it, the C definition shown in figure 5.4 for the `trie_t` structure could be used.

Each letter of the alphabet maps to one of the indices of the `children` array, and writing the code to navigate through the resulting tree is quite simple.

However, from an engineering point of view, this simplistic approach can be disastrous for many applications, because the typical number of children of each node is far fewer than the size of the alphabet. For example, in figure 5.2, each node has, on average, 0.9 children, although this representation must always allocate enough space for each node to have 26 children.

For example, on the HP computers, `sizeof(trie_t) == 112`, so if we used this structure to represent the six keys in the trie illustrated in figure 5.2, this structure would require 1120

Figure 5.4: A 26-ary tree node to implement a trie of strings of lowercase letters.

```
#define MAX_CHILDREN    26      /* enough for 'a' through 'z' */

typedef struct  _trie_t {
        struct  _trie_t *children[MAX_CHILDREN];
        int             value;
        int             endOfKey;
} trie_t;
```

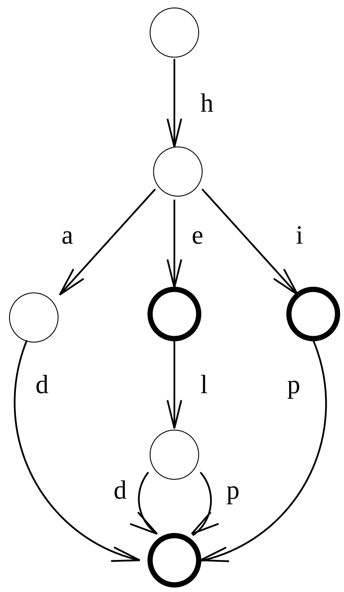
bytes of memory to represent six strings which have a combined length of only 24 bytes!

As another example, a trie consisting of the 20,000 lower-case words in my computer's small spell-checking dictionary requires 62,000 nodes to represent, for a total of nearly 7 megabytes of memory, even though the dictionary itself is only 200,000 bytes in length. Thus, a trie to represent this small dictionary requires nearly 35 times more memory than the dictionary itself.

Seven megabytes might not seem like a very large amount of memory by today's standards, but note the factor of 35 increase in memory consumption– by picking a larger problem, we can soon soak up enough memory to cause concern on any system. Depending on the nature of the keys represented by the trie, this factor may grow or shrink, but a few empirical experiments suggest that this factor is actually *lower* than many typical sets of keys!

If we expand our representation to allow uppercase letters and punctuation, then the memory requirements would increase even further. However, this might not be necessary, at least in the case of the dictionary; only a small percentage of the words in the dictionary (at least, the spell-checking dictionary I have) contain uppercase letters, and nearly all of these uppercase letters appear as the first letter in a word. Therefore, it would be possible to handle nearly all words that contain an uppercase letter by adding a special case to the way that the root node of our trie is treated. Punctuation is more troublesome, however, since the position of punctuation in words is not as conveniently localized.

Figure 5.5: A trie with a single shared leaf.



5.4 Reducing Memory Requirements: Eliminating Nodes

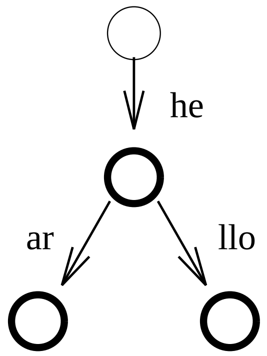
5.4.1 Special Case: Leaf Nodes

One observation is that in our implementation, leaf nodes are particularly wasteful: leaf nodes have no children, but require as much space as nodes that have 26 children.

If the only thing that we are interested in storing in the trie is a set of keys (with no associated values), then leaf nodes contain no useful information except that their `endOfKey` value is non-zero. This suggests an immediate optimization: represent *all* of the leaf nodes with a single leaf. This idea is illustrated in figure 5.5, which once again represents the same keys as the trie in figure 5.2.

If we do wish to associate values with each key, then unique leaf nodes are necessary– but they only need to be large enough to hold the value, and can omit the pointers to 26 children.

Figure 5.6: A patricia representing "he", "hello", and "hear".



5.4.2 Patricia Tries

One variation of tries is called a *Patricia Tree* (or simply *Patricia*).[2] The actual patricia data structure is somewhat different from what I describe here; I am focusing on a specific idea used by patricias that is particularly useful in our effort to reduce the memory consumption of tries. See Knuth, *The Art of Computer Programming, vol 3*, or Lewis and Denenberg, *Data Structures and Their Algorithms* for a full exposition of patricias.

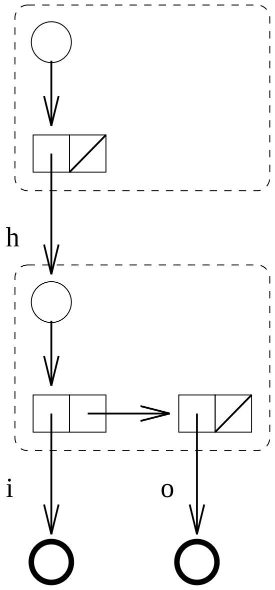
In a patricia, sequences of nodes that all have only one child and do not represent the end of a key are collapsed into a single transition. We can apply this idea to our tries by adding the possibility that each transition represents several characters of a string, instead of just one. An example of such a trie, representing the keys "he", "hello", and "hear" is shown in figure 5.6. Note that if only keys are stored in this kind of trie, then the trick of combining leaf nodes can be used again here, although this is not illustrated in figure 5.6.

At first glance, this would seem to greatly complicate the search algorithm, since during a search, at each node we might have to look at an unknown number of characters of the key we are searching for instead of single character. However, in actuality this does not complicate matters very much. Since only sequences of nodes that have one child are collapsed, no such collapsed sequence can possibly be a prefix of another. Therefore, although each transition might represent many characters, only one transition (the transition that begins with the next character in the key) can possibly be valid at each step. Therefore, the search is straightforward and efficient.

INSERT and DELETE in a trie constructed this way is slightly more complicated. When a key is deleted, it might present an opportunity to collapse a sequence of transitions, and

[2] Patricia stands for "Practical Algorithm To Retrieve Information Coded in Alphanumeric".

Figure 5.7: A de la Briandais tree containing “hi” and “ho”.



when a key is added it may be necessary to uncollapse a transition.

5.5 Reducing Memory Requirements: Smaller Nodes

There are two ways to reduce the memory consumption of a trie– reduce the number of nodes (as the techniques mentioned above attempt to do), or reduce the amount of memory required by each node. In this section, we will explore a technique that attempts to do the latter.

The problem is that the array of pointers to children is typically very sparse. We have seen sparse arrays before, so we know an approach to this problem: instead of storing the array explicitly as an ordinary C array, we can store the array as a linked list. This technique is generally called a *de la Briandais tree* (after Rene de la Briandais, who developed many of the ideas behind the practical implementation and application of tries). An example of a de la Briandais tree is shown in figure 5.7.

Traversing a linked list to find a child pointer (or discover that there is not one) is a considerably more expensive operation than a simple array lookup, so tries implemented in this manner are not as time-efficient as our array-based tries. The decrease in speed is proportional, in the worst case, to the size of the character set in use– for our example application, where there are 26 possible characters, we might need to examine as many as 25 linked list elements at each node before we find the child we are looking for, or determine that it does not exist. However, this slowdown is proportional *in the worst case* to the size of the alphabet, and is not related in any way to the number of keys in the trie or their lengths.

Nodes that have many children may actually require more memory to represent using linked lists than arrays, since the linked list cells themselves require more memory than the array elements, but if the trie is sparse then the savings can be dramatic.

5.6 More Trie Operations

The FIND, INSERT, and DELETE operations shown here can also be implemented via a hash table, as we will learn shortly. A properly constructed hash table has the same properties with respect to FIND, INSERT, and DELETE– all run in time proportional to the length of the key (since the hash function takes time proportional to the length of the key, and this time will dominate the operations in a good hash table), not the number of keys. Hash tables also are easy to modify so that they can deal with non-unique keys, which is extremely practical in some applications. Best of all, hash tables are relatively easy to implement and understand, and good implementations generally consume far less memory than tries. Given all of these properties, why would anyone still choose a trie?

- **Tries eliminate the need to choose a good hashing function.**

  If the hashing function is chosen poorly, or the data is pathologically bad, the degenerate behavior of hash tables is bad. There is no bad data set for a trie; all data sets are equally good, as far as time-complexity is concerned.

  On the other hand, some data sets may soak up more memory than others (although even in the worst case, memory requirements are bounded by an amount proportional to the total length of the keys).

- **A trie can preserve (and represent) an implicit ordering of the keys, based on an ordering of the characters of the alphabet.**

  This means that a trie can be used to sort a set of keys in much the same was a binary tree can– but a trie does not exhibit the degenerate $O(n^2)$ worst case behavior that results from inserting keys into a binary tree in a "bad" order. This ordering also means that it is possible to easily find the lexical predecessor or successor of any key, as well as several other properties of a key (see exercise 2 for an example).

- **Other information can be coded into the trie along with the keys.**

  Finally, there are algorithms that encode information related to the keys in the tree itself. For example, information about the prefix of every key is implicit in the trie, so the question of whether there exist any keys that begin with a particular substring is easy to answer. This sort of functionality is often used for command or filename completion. It is also absolutely indispensable to many adaptive coding techniques; for example the Lempel-Ziv compression algorithm is very easy to implement with a trie.[3]

  It is possible to construct hashing functions that allow efficient "prefix checking", as we will see in a later chapter, but it is difficult to construct one that can out-perform a trie.

5.7 Exercises

1. Write an algorithm that prints out all of the keys stored in a trie. Not counting the action of printing each key, it should run in time $O(n)$, where n is the number of nodes in the trie. (Printing the keys will require time $O(L)$, where L is the total length of all of the keys.) An auxillary data structure can be used, but its size must not be greater than $O(l)$, where l is the length of the longest key in the trie.

2. Using a patricia, as described in this chapter, show that inserting a new key can cause at most one node to be uncollapsed, and that the deletion of one key can cause at most one node to be collapsed.

3. Write an algorithm that takes a key k and finds all the keys in a trie that differ from k in exactly one character position. The algorithm should run in time $O(l)$, where l is the length of the key, and be independent of the number of keys in the trie.

   There are, of course, only $O(l)$ such keys possible, so a time complexity of $O(l)$ should be the worst case. Your goal is to rule out as many possibilities as you can as early as possible.

[3]The Lempel-Ziv algorithm would make a fascinating topic for a final paper.

4. What is the time complexity of using a trie to sort a set of keys? (Hint– first choose how to measure the the "size" of the problem.)