Data Structures

INTERVIEW QUESTIONS

Tries

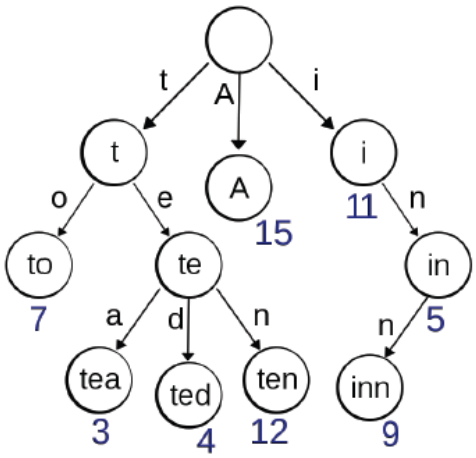
Test Your Caliber
e-University Home

List Of Topics

Trie

In computer science, a **trie**, or prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest. The term trie comes from "retrieval." Following the etymology, the inventor, Edward Fredkin, pronounces it English pronunciation: /ˈtriː/ "tree". However, it is pronounced English pronunciation: /ˈtraɪ/ "try" by other authors.

A trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn".



In the example shown, keys are listed in the nodes and values below them. Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a deterministic finite automaton, although the symbol on each edge is often implicit in the order of the branches. It is not necessary for keys to be explicitly stored in nodes. (In the figure, words are shown only to illustrate how the trie works.) Though it is most common, tries need not be keyed by character strings. The same algorithms can easily be adapted to serve similar functions of ordered lists of any construct, e.g., permutations on a list of digits or shapes.

Advantages relative to binary search tree

The following are the main advantages of tries over binary search trees (BSTs):

- Looking up keys is faster. Looking up a key of length *m* takes worst case O(*m*) time. A BST performs O(log(*n*)) comparisons of keys, where *n* is the number of elements in the tree, because lookups depend on the depth of the tree, which is logarithmic in the number of keys if the tree is balanced. Hence in the worst case, a BST takes O(*m*log *n*) time. Moreover, in the worst case log(*n*) will approach *m*. Also, the simple operations tries use during lookup, such as array indexing using a character, are fast on real machines.
- Tries can require less space when they contain a large number of short strings, because the keys are not stored explicitly and nodes are shared between keys with common initial subsequences.
- Tries facilitate longest-prefix matching, helping to find the key sharing the longest possible prefix of characters all unique.

Applications

As replacement of other data structures

As mentioned, a trie has a number of advantages over binary search trees.[4] A trie can also be used to replace a hash table, over which it has the following advantages:

- Looking up data in a trie is faster in the worst case, O(m) time, compared to an imperfect hash table. An imperfect hash table can have key collisions. A key collision is the hash function mapping of different keys to the same position in a hash table. The worst-case lookup speed in an imperfect

Pragnya Meter
Next Chapter
e-University Search
Search

Related Jobs

hash table is O(N) time, but far more typically is O(1), with O(m) time spent evaluating the hash.

- There are no collisions of different keys in a trie.
- Buckets in a trie which are analogous to hash table buckets that store key collisions are only necessary if a single key is associated with more than one value.
- There is no need to provide a hash function or to change hash functions as more keys are added to a trie.
- A trie can provide an alphabetical ordering of the entries by key.

Tries do have some drawbacks as well:

- Tries can be slower in some cases than hash tables for looking up data, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random access time is high compared to main memory.
- It is not easy to represent all keys as strings, such as floating point numbers - a straightforward encoding using the bitstring of their encoding leads to long chains and prefixes that are not particularly meaningful.

Dictionary representation

A common application of a trie is storing a dictionary, such as one found on a mobile telephone. Such applications take advantage of a trie's ability to quickly search for, insert, and delete entries; however, if storing dictionary words is all that is required (i.e. storage of information auxiliary to each word is not required), a minimal acyclic deterministic finite automaton would use less space than a trie. Tries are also well suited for implementing approximate matching algorithms, including those used in spell checking and hyphenation software.

Algorithms

We can describe trie lookup (and membership) easily. Given a recursive trie type, storing an optional value at each node, and a list of children tries, indexed by the next character, (here, represented as a Haskell data type):

data Trie a = Trie { value :: Maybe a , children :: [(Char,Trie a)] }

We can lookup a value in the trie as follows:

find :: String -> Trie a -> Maybe a find [] t = value t find (k:ks) t = case lookup k (children t) of Nothing -> Nothing

Just t' -> find ks t'

In an imperative style, and assuming an appropriate data type in place, we can describe the same algorithm in Python (here, specifically for testing membership). Note that children is map of a node's children; and we say that a "terminal" node is one which contains a valid word.

```
def is_key_in_trie(node, key):
if not key: # base case: key is an empty
string
return node.is_terminal() # we have a match if the node
is terminal
c = key[0] # first character in key
if c not in node.children: # no child node?
return False # key is not empty, so  there
is no match
child = node.children[c] # get child node
tail = key[1:] # the rest of the key after
first character
return is_key_in_trie(child, tail) # recurse
```

Sorting

Lexicographic sorting of a set of keys can be accomplished with a simple trie-based algorithm as follows:

- Insert all keys in a trie.
- Output all keys in the trie by means of pre-order traversal, which results in output that is in lexicographically increasing order. Pre-order traversal is a kind of depth-first traversal. In-order traversal is another kind of depth-first traversal that is more appropriate for outputting the values that are in a binary search tree rather than a trie.

This algorithm is a form of radix sort.

A trie forms the fundamental data structure of Burstsort; currently (2007) the fastest known, memory/cache-based, string sorting algorithm.[6]

A parallel algorithm for sorting N keys based on tries is O(1) if there are N processors and the length of the keys have a constant upper bound. There is the potential that the keys might collide by having common prefixes or by being identical to one another, reducing or eliminating the speed advantage of having multiple processors operating in parallel.

Full text search

A special kind of trie, called a suffix tree, can be used to index all suffixes in a text in order to carry out fast full text searches.

Compressing tries

When the trie is mostly static, i.e. all insertions or deletions of keys from a prefilled trie

When the trie is mostly static, i.e. all insertions or deletions of keys from a prebuilt trie are disabled and only lookups are needed, and when the trie nodes are not keyed by node specific data (or if the node's data is common) it is possible to compress the trie representation by merging the common branches [7] . This application is typically used for compressing lookup tables when the total set of stored keys is very sparse within their representation space.

For example it may be used to represent sparse bitsets (i.e. subsets of a much fixed enumerable larger set) using a trie keyed by the bit element position within the full set, with the key created from the string of bits needed to encode the integral position of each element. The trie will then have a very degenerate form with many missing branches, and compression becomes possible by storing the leaf nodes (set segments with fixed length) and combining them after detecting the repetition of common patterns or by filling the unused gaps.

Such compression is also typically used, in the implementation of the various fast lookup tables needed to retrieve Unicode character properties (for example to represent case mapping tables, or lookup tables containing the combination of base and combining characters needed to support Unicode normalization). For such application, the representation is similar to transforming a very large unidimensional sparse table into a multidimensional matrix, and then using the coordinates in the hyper-matrix as the string key of an uncompressed trie.

The compression will then consist into detecting and merging the common columns within the hyper matrix to compress the last dimension in the key; each dimension of the hypermatrix stores the start position within a storage vector of the next dimension for each coordinate value, and the resulting vector is itself compressible when it is also sparse, so each dimension (associated to a layer level in the trie) is compressed separately.

Some implementations do support such data compression within dynamic sparse tries and allow insertions and deletions in compressed tries, but generally this has a significant cost when compressed segments need to be split or merged, and some tradeoff has to be made between the smallest size of the compressed trie and the speed of updates, by limiting the range of global lookups for comparing the common branches in the sparse trie. The result of such compression may look similar to trying to transform the trie into a directed acyclic graph (DAG), because the reverse transform from a DAG to a trie is obvious and always possible, however it is constrained by the form of the key chosen to index the nodes. Another compression approach is to "unravel" the data structure into a single byte array [8].

This approach eliminates the need for node pointers which reduces the memory requirements substantially and makes memory mapping possible which allows the virtual memory manager to load the data into memory very efficiently. Another compression approach is to "pack" the trie . Liang describes a space-efficient implementation of a sparse
packed trie applied to hyphenation, in which the descendants of each node may be interleaved in memory.