WIKIPEDIA
The Free Encyclopedia

Trie

In computer science, a **trie** (/ˈtriː/, /ˈtraɪ/), also called **digital tree** or **prefix tree**,[1] is a type of *k*-ary search tree, a tree data structure used for locating specific keys from within a set. These keys are most often strings, with links between nodes defined not by the entire key, but by individual characters. In order to access a key (to recover its value, change it, or remove it), the trie is traversed depth-first, following the links between nodes, which represent each character in the key.

Unlike a binary search tree, nodes in the trie do not store their associated key. Instead, a node's position in the trie defines the key with which it is associated. This distributes the value of each key across the data structure, and means that not every node necessarily has an associated value.

All the children of a node have a common prefix of the string associated with that parent node, and the root is associated with the empty string. This task of storing data accessible by its prefix can be accomplished in a memory-optimized way by employing a radix tree.

Though tries can be keyed by character strings, they need not be. The same algorithms can be adapted for ordered lists of any underlying type, e.g. permutations of digits or shapes. In particular, a **bitwise trie** is keyed on the individual bits making up a piece of fixed-length binary data, such as an integer or memory address. The key lookup complexity of a trie remains proportional to the key size. Specialized trie implementations such as compressed tries are used to deal with the enormous space requirement of a trie in naive implementations.

Trie	
Type	tree
Invented	1960
Invented by	Edward Fredkin, Axel Thue, and René de la Briandais

Time complexity in big O notation

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(n)	O(n)
Insert	O(n)	O(n)
Delete	O(n)	O(n)

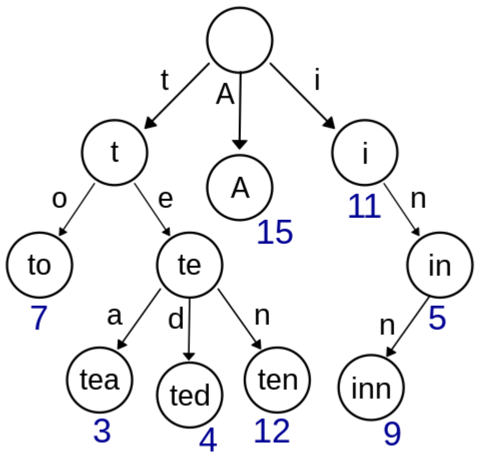


Fig. 1: A trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn". Each complete English word has an arbitrary integer value associated with it.

History, etymology, and pronunciation

The idea of a trie for representing a set of strings was first abstractly described by Axel Thue in 1912.[2][3] Tries were first described in a computer context by René de la Briandais in 1959.[4][3][5]:336

The idea was independently described in 1960 by Edward Fredkin,[6] who coined the term *trie*, pronouncing it /ˈtriː/ (as "tree"), after the middle syllable of *retrieval*.[7][8] However, other authors pronounce it /ˈtraɪ/ (as "try"), in an attempt to distinguish it verbally from "tree".[7][8][3]

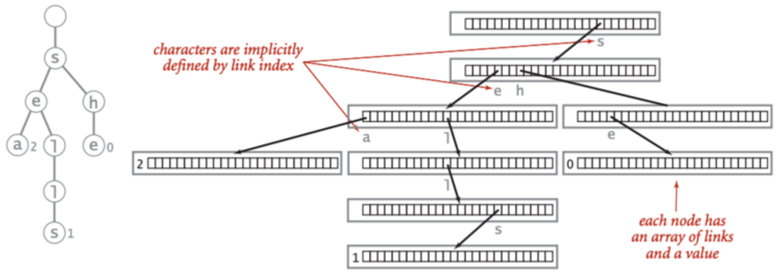
Overview

Tries are a form of string-indexed look-up data structure, which is used to store a dictionary list of words that can be searched on in a manner that allows for efficient generation of completion lists.[9][10]:1 A prefix trie is an ordered tree data structure used in the representation of a set of strings over a finite alphabet set, which allows efficient storage of words with common prefixes.[1]

Tries can be efficacious on string-searching algorithms such as predictive text, approximate string matching, and spell checking in comparison to a binary search trees.[11][8][12]:358 A trie can be seen as a tree-shaped deterministic finite automaton.[13]

Operations

Tries support various operations: insertion, deletion, and lookup of a string key. Tries are composed of **nodes** that contain *links* that are either references to other child suffix child nodes, or **nil** . Except for *root*, each node is pointed to by just one other node, called the *parent*. Each node contains **R** links, where **R** is the cardinality of the applicable alphabet, although tries have a substantial number of **nil** links. In most cases, the size of **Children** array is bitlength of the character encoding - 256 in the case of (unsigned) ASCII.[14]:732


characters are implicitly defined by link index
each node has an array of links and a value
s
e
h
a
l
s
2
0
1
Fig. 2: Trie representation of the string sets: sea, sells, and she.

The **nil** links within **Children** in **Node** emphasizes the following characteristics:[14]:734[5]:336

1. Characters and string keys are implicitly stored in the trie data structure representation, and include a character sentinel value indicating string-termination.
2. Each node contains one possible link to a prefix of strong keys of the set.

A basic structure type of nodes in the trie is as follows; **Node** may contain an optional **Value**, which is associated with each key stored in the last character of string, or terminal node.

```
structure Node
    Children Node[Alphabet-Size]
    Is-Terminal Boolean
    Value Data-Type
end structure
```

Searching

Searching a **Value** in a trie is guided by the characters in the search string key, as each node in the trie contains a corresponding link to each possible character in the given string. Thus, following the string within the trie yields the associated **Value** for the given string key. A **nil** link within search execution indicates the inexistence of the key.[14]:732-733

Following pseudocode implements the search procedure for a given string key (**key**) in a rooted trie (**x**).[15]:135

```
Trie-Find(x, key)
    for 0 ≤ i < key.length do
        if x.Children[key[i]] = nil then
            return false
        end if
        x := x.Children[key[i]]
    repeat
    return x.Value
```

In the above pseudocode, $\mathbf{x}$ and $\mathbf{key}$ correspond to the pointer of trie's root node and the string key respectively. The search operation, in a standard trie, takes $O(\mathrm{dm})$, m is the size of the string parameter $\mathbf{key}$, and d corresponds to the alphabet size.[16]:754 Binary search trees, on the other hand, take $O(m \log n)$ on the worst case, since the search depends on the height of the tree ($\log n$) of the BST (in case of balanced trees), where n and m being number of keys and the length of the keys.[12]:358

Tries occupy less space in comparison with BST if it encompasses a large number of short strings, since nodes share common initial string subsequences and stores the keys implicitly on the structure.[12]:358 The terminal node of the tree contains a non-nil Value, and it is a *search hit* if the associated value is found in the trie, and *search miss* if it is not.[14]:733

Insertion

Insertion into trie is guided by using the character sets as the indexes into the $\mathbf{Children}$ array until last character of the string key is reached.[14]:733-734 Each node in the trie corresponds to one call of the radix sorting routine, as the trie structure reflects the execution of pattern of the top-down radix sort.[15]:135

```
1  Trie-Insert(x, key, value)
2      for 0 ≤ i < key.length do
3          if x.Children[key[i]] = nil then
4              x.Children[key[i]] := Node()
5          end if
6          x := x.Children[key[i]]
7      repeat
8      x.Value := value
9      x.Is-Terminal := True
```

If a $\mathbf{nil}$ link is encountered prior to reaching the last character of the string key, a new $\mathbf{Node}$ is created, such along lines 3–5.[14]:745 $\mathbf{x.Value}$ gets assigned to input $\mathbf{value}$; if $\mathbf{x.Value}$ wasn't $\mathbf{nil}$ at the time of insertion, the value associated with the given string key gets substituted with the current one.

Deletion

Deletion of a key–value pair from a trie involves finding the terminal node with the corresponding string key, marking the terminal indicator and value to *false* and $\mathbf{nil}$ correspondingly.[14]:740

Following is a recursive procedure for removing a string key ($\mathbf{key}$) from rooted trie ($\mathbf{x}$).

```
Trie-Delete(x, key)
    if key = nil then
```

```
 3          if x.Is-Terminal = True then
 4              x.Is-Terminal := False
 5              x.Value := nil
 6          end if
 7          for 0 ≤ i < x.Children.length
 8              if x.Children[i] != nil
 9                  return x
10              end if
11          repeat
12          return nil
13      end if
14      x.Children[key[0]] := Trie-Delete(x.Children[key[0]], key[1:])
15      return x
```

The procedures begins by examining the **key**; **nil** denotes the arrival of a terminal node or end of string key. If terminal and if it has no children, the node gets removed from the trie (line 14 assign the character index to **nil**). However, an end of string key without the node being terminal indicates that the key does not exist, thus the procedure does not modify the trie. The recursion proceeds by incrementing **key**'s index.

Replacing other data structures

Replacement for hash tables

A trie can be used to replace a hash table, over which it has the following advantages:[12]:358

- Searching for a node with an associated key of size m has the complexity of $O(m)$, whereas an imperfect hash function may have numerous colliding keys, and the worst-case lookup speed of such a table would be $O(N)$, where N denotes the total number of nodes within the table.
- Tries do not need a hash function for the operation, unlike a hash table; there are also no collisions of different keys in a trie.
- Buckets in a trie, which are analogous to hash table buckets that store key collisions, are necessary only if a single key is associated with more than one value.
- String keys within the trie can be sorted using a predetermined alphabetical ordering.

However, tries are less efficient than a hash table when the data is directly accessed on a secondary storage device such as a hard disk drive that has higher random access time than the main memory.[6] Tries are also disadvantageous when the key value cannot be easily represented as string, such as floating point numbers where multiple representations are possible (e.g. 1 is equivalent to 1.0, +1.0, 1.00, etc.),[12]:359 however it can be unambiguously represented as a binary number in IEEE 754, in comparison to two's complement format.[17]

Implementation strategies

Tries can be represented in several ways, corresponding to different trade-offs between memory use and speed of the operations.[5]:341 Using a vector of pointers for representing a trie consumes enormous space; however, memory space can be reduced at the expense of running time if a singly linked list is used for each node vector, as most entries of the vector contains **nil**.[3]:495

Techniques such as *alphabet reduction* may alleviate the high space complexity by reinterpreting the original string as a long string over a smaller alphabet i.e. a string of *n* bytes can alternatively be regarded as a string of $2n$ four-bit units and stored in a trie with sixteen pointers per node. However, lookups need to visit twice as many nodes in the worst-case, although space requirements go down by a factor of eight.[5]:347–352 Other techniques include storing a vector of 256 ASCII pointers as a bitmap of 256 bits representing ASCII alphabet, which reduces the size of individual nodes dramatically.[18]

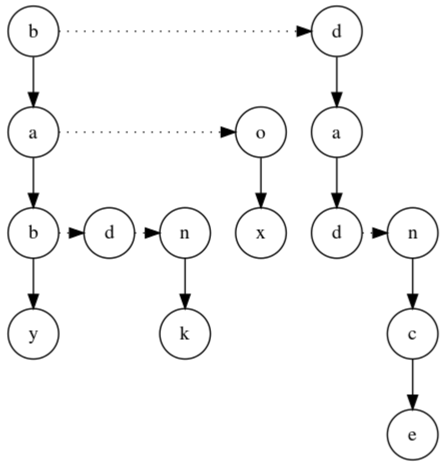


Fig. 3: A trie implemented as a left-child right-sibling binary tree: vertical arrows are `child` pointers, dashed horizontal arrows are `next` pointers. The set of strings stored in this trie is {`baby, bad, bank, box, dad, dance`}. The lists are sorted to allow traversal in lexicographic order.

Bitwise tries

Bitwise tries are used to address the enormous space requirement for the trie nodes in a naive simple pointer vector implementations. Each character in the string key set is represented via individual bits, which are used to traverse the trie over a string key. The implementations for these types of trie use vectorized CPU instructions to find the first set bit in a fixed-length key input (e.g. GCC's `__builtin_clz()` intrinsic function). Accordingly, the set bit is used to index the first item, or child node, in the 32- or 64-entry based bitwise tree. Search then proceeds by testing each subsequent bit in the key.[19]

This procedure is also cache-local and highly parallelizable due to register independency, and thus performant on out-of-order execution CPUs.[19]

Compressed tries

Radix tree, also known as a **compressed trie**, is a space-optimized variant of a trie in which nodes with only one child get merged with its parents; elimination of branches of the nodes with a single child results in better in both space and time metrics.[20][21]:452 This works best when the trie remains static and set of keys stored are very sparse within their representation space.[22]:3–16

One more approach is to "pack" the trie, in which a space-efficient implementation of a sparse packed trie applied to automatic hyphenation, in which the descendants of each node may be interleaved in memory.[8]

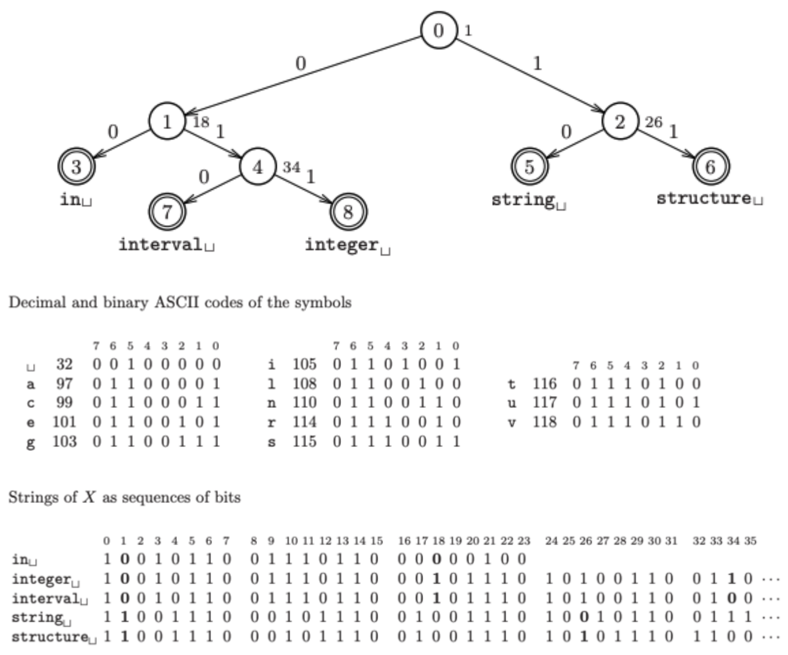
Patricia trees

Patricia trees are a particular implementation of compressed binary trie that utilize binary encoding of the string keys in its representation.[23][15]:140 Every node in a Patricia tree contains an index, known as a "skip number", that stores the node's branching index to avoid empty subtrees during traversal.[15]:140-141 A naive implementation of a trie consumes immense storage due to larger number of leaf-nodes caused by sparse distribution of keys; Patricia trees can be efficient for such cases.[15]:142[24]:3

A representation of a Patricia tree with string keys $\{in, integer, interval, string, structure\}$ is shown in figure 4, and each index value adjacent to the nodes represents the "skip number" - the index of the bit with which branching is to be decided.[24]:3 The skip number 1 at node 0 corresponds to the position 1 in the binary encoded ASCII where the leftmost bit differed in the key set X.[24]:3-4 The skip number is crucial for search, insertion, and deletion of nodes in the Patricia tree, and a bit masking operation is performed during every iteration.[15]:143

Applications

Trie data structures are commonly used in predictive text or autocomplete dictionaries, and approximate matching algorithms.[11] Tries enable faster searches, occupy less space, especially when the set contains large number of short strings, thus used in spell checking, hyphenation applications and longest prefix match algorithms.[8][12]:358 However, if storing dictionary words is all that is required (i.e. there is no need to store metadata associated with each word), a minimal deterministic acyclic finite state automaton (DAFSA) or radix tree would use less storage space than a trie. This is because DAFSAs and radix trees can compress identical branches from the trie which correspond to the same suffixes (or parts) of different words being stored. String dictionaries are also utilized in natural language processing, such as finding lexicon of a text corpus.[25]:73



Decimal and binary ASCII codes of the symbols

		7	6	5	4	3	2	1	0
␣	32	0	0	1	0	0	0	0	0
a	97	0	1	1	0	0	0	0	1
c	99	0	1	1	0	0	0	1	1
e	101	0	1	1	0	0	1	0	1
g	103	0	1	1	0	0	1	1	1
i	105	0	1	1	0	1	0	0	1
l	108	0	1	1	0	0	1	0	0
n	110	0	1	1	0	0	1	1	0
r	114	0	1	1	1	0	0	1	0
s	115	0	1	1	1	0	0	1	1
t	116	0	1	1	1	0	1	0	0
u	117	0	1	1	1	0	1	0	1
v	118	0	1	1	1	0	1	1	0

Strings of X as sequences of bits

	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31	32 33 34 35	
in␣	1 **0** 0 1 0 1 1 0	0 1 1 1 0 1 1 0	0 0 **0** 0 0 1 0 0			
integer␣	1 **0** 0 1 0 1 1 0	0 1 1 1 0 1 1 0	0 0 **1** 0 1 1 1 0	1 0 1 0 0 1 1 0	0 1 **1** 0	···
interval␣	1 **0** 0 1 0 1 1 0	0 1 1 1 0 1 1 0	0 0 **1** 0 1 1 1 0	1 0 1 0 0 1 1 0	0 1 **0** 0	···
string␣	1 **1** 0 0 1 1 1 0	0 0 1 0 1 1 1 0	0 1 0 0 1 1 1 0	1 0 **0** 1 0 1 1 0	0 1 1 1	···
structure␣	1 **1** 0 0 1 1 1 0	0 0 1 0 1 1 1 0	0 1 0 0 1 1 1 0	1 0 **1** 0 1 1 1 0	1 1 0 0	···

Fig. 4: Patricia tree representation of string keys: in, integer, interval, string, and structure.

Sorting

Lexicographic sorting of a set of string keys can be implemented by building a trie for the given keys and traversing the tree in pre-order fashion;[26] this is also a form of radix sort.[27] Tries are also fundamental data structures for burstsort, which is notable for being the fastest string sorting algorithm as of 2007,[28] accompanied for its efficient use of CPU cache.[29]

Full-text search

A special kind of trie, called a suffix tree, can be used to index all suffixes in a text to carry out fast full-text searches.[30]

Web search engines

A specialized kind of trie called a compressed trie, is used in web search engines for storing the indexes - a collection of all searchable words.[31] Each terminal node is associated with a list of URLs—called occurrence list—to pages that match the keyword. The trie is stored in the main memory, whereas the occurrence is kept in an external storage, frequently in large clusters, or the in-memory index points to documents stored in an external location.[32]

Bioinformatics

Tries are used in Bioinformatics, notably in sequence alignment software applications such as BLAST, which indexes all the different substring of length *k* (called k-mers) of a text by storing the positions of their occurrences in a compressed trie sequence databases.[25]:75

Internet routing

Compressed variants of tries, such as databases for managing Forwarding Information Base (FIB), are used in storing IP address prefixes within routers and bridges for prefix-based lookup to resolve mask-based operations in IP routing.[25]:75

See also

- Suffix tree
- Hash trie
- Hash array mapped trie
- Prefix hash tree
- Ctrie
- HAT-trie

References

1. Maabar, Maha (17 November 2014). "Trie Data Structure" (https://bioinformatics.cvr.ac.uk/trie-data-structure/). CVR, University of Glasgow. Archived (https://web.archive.org/web/20210127130913/https://bioinformatics.cvr.ac.uk/trie-data-structure/) from the original on 27 January 2021. Retrieved 17 April 2022.
2. Thue, Axel (1912). "Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen" (https://archive.org/details/skrifterutgitavv121chri/page/n11/mode/2up). *Skrifter Udgivne Af Videnskabs-Selskabet I Christiania*. **1912** (1): 1–67. Cited by Knuth.
3. Knuth, Donald (1997). "6.3: Digital Searching". *The Art of Computer Programming Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley. p. 492. ISBN 0-201-89685-0.
4. de la Briandais, René (1959). *File searching using variable length keys* (https://web.archive.org/web/20200211163605/https://pdfs.semanticscholar.org/3ce3/f4cc1c91d03850ed84ef96a08498e018d18f.pdf) (PDF). Proc. Western J. Computer Conf. pp. 295–298. doi:10.1145/1457838.1457895 (https://doi.org/10.1145%2F1457838.1457895). S2CID 10963780 (https://api.semanticscholar.org/CorpusID:10963780). Archived from the original (https://pdfs.semanticscholar.org/3ce3/f4cc1c91d03850ed84ef96a08498e018d18f.pdf) (PDF) on 2020-02-11. Cited by Brass and by Knuth.
5. Brass, Peter (8 September 2008). *Advanced Data Structures* (https://www.cambridge.org/core/books/advanced-data-structures/D56E2269D7CEE969A3B8105AD5B9254C). UK: Cambridge University Press. doi:10.1017/CBO9780511800191 (https://doi.org/10.1017%2FCBO9780511800191). ISBN 978-0521880374.
6. Edward Fredkin (1960). "Trie Memory" (https://doi.org/10.1145%2F367390.367400). *Communications of the ACM*. **3** (9): 490–499. doi:10.1145/367390.367400 (https://doi.org/10.1145%2F367390.367400). S2CID 15384533 (https://api.semanticscholar.org/CorpusID:15384533).
7. Black, Paul E. (2009-11-16). "trie" (https://xlinux.nist.gov/dads/HTML/trie.html). *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology. Archived (https://web.archive.org/web/20110429080033/http://xlinux.nist.gov/dads/HTML/trie.html) from the original on 2011-04-29.