WIKIPEDIA
The Free Encyclopedia
WIKIPEDIA

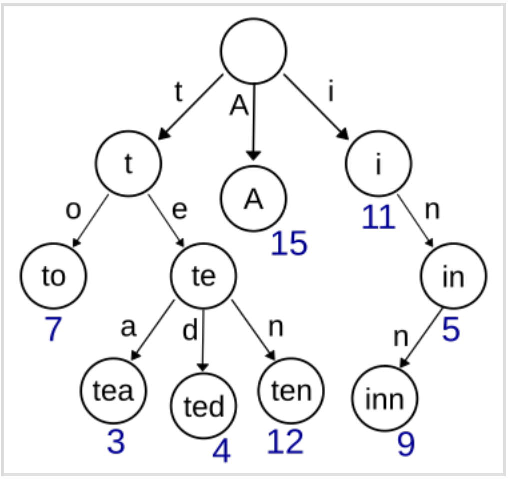
Trie

In computer science, a **trie** (/ˈtraɪ/, /ˈtriː/), also known as a **digital tree** or **prefix tree**,[1] is a specialized search tree data structure used to store and retrieve strings from a dictionary or set. Unlike a binary search tree, nodes in a trie do not store their associated key. Instead, each node's *position* within the trie determines its associated key, with the connections between nodes defined by individual characters rather than the entire key.

Trie

Type	Tree	
Invented	1960	
Invented by	Edward Fredkin, Axel Thue, and René de la Briandais	
Time complexity in big O notation		
Operation	**Average**	**Worst case**
Search	O(*n*)	O(*n*)
Insert	O(*n*)	O(*n*)
Delete	O(*n*)	O(*n*)
Space complexity		
Space	O(*n*)	O(*n*)

Tries are particularly effective for tasks such as autocomplete, spell checking, and IP routing, offering advantages over hash tables due to their prefix-based organization and lack of hash collisions. Every child node shares a common prefix with its parent node, and the root node represents the empty string. While basic trie implementations can be memory-intensive, various optimization techniques such as compression and bitwise representations have been developed to improve their efficiency. A notable optimization is the radix tree, which provides more efficient prefix-based storage.



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.

While tries commonly store character strings, they can be adapted to work with any ordered sequence of elements, such as permutations of digits or shapes. A notable variant is the **bitwise trie**, which uses individual bits from fixed-length binary data (such as integers or memory addresses) as keys.

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]

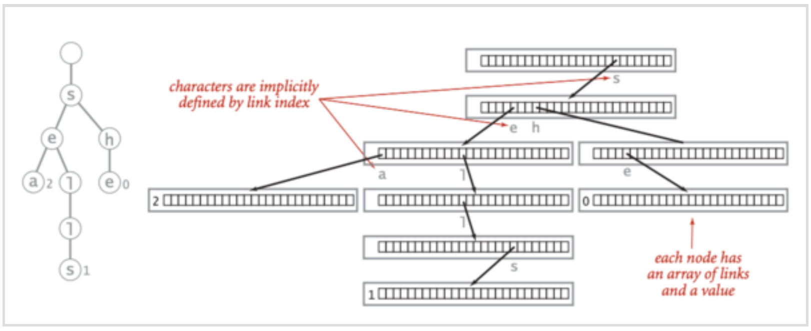
Overview

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 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, which either point to other suffix child nodes or *null*. As for every tree, each node but the root is pointed to by only one other node, called its *parent*. Each node contains as many links as the number of characters in the applicable alphabet (although tries tend to have a substantial number of null links). In some cases, the alphabet used is simply that of the character encoding—resulting in, for example, a size of 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
e
l
s
2
0
1
Trie representation of the string sets: sea, sells, and she.

The null links within the children of a node emphasize the following characteristics:[14]:734[5]:336

1. Characters and string keys are implicitly stored in the trie, 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 for 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 null link during the search indicates the inexistence of the key.[14]: 732-733

The following pseudocode implements the search procedure for a given string `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, `x` and `key` correspond to the pointer of trie's root node and the string key respectively. The search operation, in a standard trie, takes $O(\mathrm{dm})$ time, where m is the size of the string parameter key, and d corresponds to the alphabet size.[16]: 754 Binary search trees, on the other hand, take $O(m \log n)$ in 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

The trie occupies less space in comparison with a BST in the case of a large number of short strings, since nodes share common initial string subsequences and store the keys implicitly.[12]: 358 The terminal node of the tree contains a non-null 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 indexes to the children array until the 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

```
_  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 null link is encountered prior to reaching the last character of the string key, a new node is created (line 3).[14]: 745 The value of the terminal node is assigned to the input value; therefore, if the former was non-null at the time of insertion, it is substituted with the new value.

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 null correspondingly.[14]: 740

The following is a recursive procedure for removing a string `key` from rooted trie (`x`).

```
 _  Trie-Delete(x, key)
 2      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 procedure begins by examining the `key`; null denotes the arrival of a terminal node or end of a string key. If the node is terminal it has no children, it is removed from the trie (line 14). 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

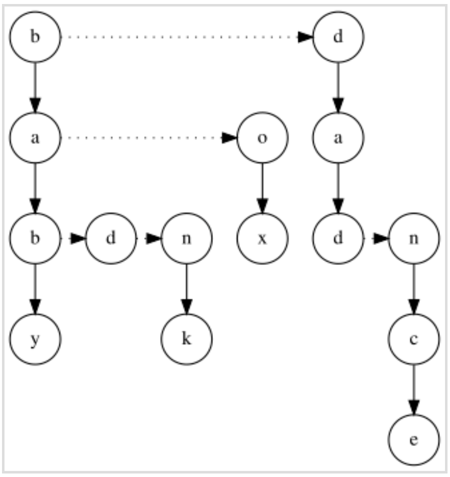
A trie can be used to replace a hash table, over which it has the following advantages:[12]: 358

- 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



A trie implemented as a left-child right-sibling binary tree: vertical arrows are `child` pointers, dotted 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.

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]

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]

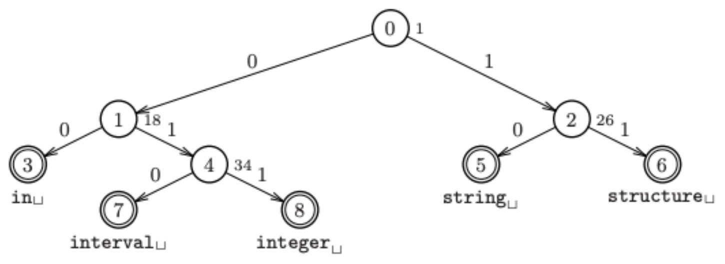
Compressed tries

Radix tree, also known as a **compressed trie**, is a space-optimized variant of a trie in which any node with only one child gets merged with its parent; elimination of branches of the nodes with a single child results in better metrics in both space and time.[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 the compressed binary trie that uses the 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



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	···

Patricia tree representation of the string set {`in`, `integer`, `interval`, `string`, `structure`}.

A representation of a Patricia tree is shown to the right. 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 $\boldsymbol{X}$.[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

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] accomplished by 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 fulltext 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
- Aho–Corasick algorithm

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.
8. Franklin Mark Liang (1983). *Word Hy-phen-a-tion By Com-put-er* (http://www.tug.org/docs/liang/liang-thesis.pdf) (PDF) (Doctor of Philosophy thesis). Stanford University. Archived (https://web.archive.org/web/20051111105124/http://www.tug.org/docs/liang/liang-thesis.pdf) (PDF) from the original on 2005-11-11. Retrieved 2010-03-28.
9. "Trie" (https://ds.cs.rutgers.edu/assignment-trie/). School of Arts and Science, Rutgers University. 2022. Archived (https://ghostarchive.org/archive/20220417170426/https://ds.cs.rutgers.edu/assignment-trie/) from the original on 17 April 2022. Retrieved 17 April 2022.
10. Connelly, Richard H.; Morris, F. Lockwood (1993). "A generalization of the trie data structure" (https://surface.syr.edu/eecs_techreports/162/). *Mathematical Structures in Computer Science*. **5** (3). Syracuse University: 381–418. doi:10.1017/S0960129500000803 (https://doi.org/10.1017%2FS0960129500000803). S2CID 18747244 (https://api.semanticscholar.org/CorpusID:18747244).
11. Aho, Alfred V.; Corasick, Margaret J. (Jun 1975). "Efficient String Matching: An Aid to Bibliographic Search" (https://doi.org/10.1145%2F360825.360855). *Communications of the ACM*. **18** (6): 333–340. doi:10.1145/360825.360855 (https://doi.org/10.1145%2F360825.360855). S2CID 207735784 (https://api.semanticscholar.org/CorpusID:207735784).
12. Thareja, Reema (13 October 2018). "Hashing and Collision". *Data Structures Using C* (https://global.oup.com/academic/product/data-structures-using-c-9780198099307) (2 ed.). Oxford University Press. ISBN 9780198099307.

13. Daciuk, Jan (24 June 2003). *Comparison of Construction Algorithms for Minimal, Acyclic, Deterministic, Finite-State Automata from Sets of Strings* (https://link.springer.com/chapter/10.1007/3-540-44977-9_26). International Conference on Implementation and Application of Automata. Springer Publishing. pp. 255–261. doi:10.1007/3-540-44977-9_26 (https://doi.org/10.1007%2F3-540-44977-9_26). ISBN 978-3-540-40391-3.
14. Sedgewick, Robert; Wayne, Kevin (3 April 2011). *Algorithms* (https://algs4.cs.princeton.edu/home/) (4 ed.). Addison-Wesley, Princeton University. ISBN 978-0321573513.
15. Gonnet, G. H.; Yates, R. Baeza (January 1991). *Handbook of algorithms and data structures: in Pascal and C* (https://dl.acm.org/doi/book/10.5555/103324) (2 ed.). Boston, United States: Addison-Wesley. ISBN 978-0-201-41607-7.
16. Patil, Varsha H. (10 May 2012). *Data Structures using C++* (https://global.oup.com/academic/product/data-structures-using-c-9780198066231). Oxford University Press. ISBN 9780198066231.
17. S. Orley; J. Mathews. "The IEEE 754 Format" (http://mathcenter.oxford.emory.edu/site/cs170/ieee754/). Department of Mathematics and Computer Science, Emory University. Archived (https://web.archive.org/web/20220328093853/http://mathcenter.oxford.emory.edu/site/cs170/ieee754/) from the original on 28 March 2022. Retrieved 17 April 2022.
18. Bellekens, Xavier (2014). "A Highly-Efficient Memory-Compression Scheme for GPU-Accelerated Intrusion Detection Systems". *Proceedings of the 7th International Conference on Security of Information and Networks - SIN '14*. Glasgow, Scotland, UK: ACM. pp. 302:302–302:309. arXiv:1704.02272 (https://arxiv.org/abs/1704.02272). doi:10.1145/2659651.2659723 (https://doi.org/10.1145%2F2659651.2659723). ISBN 978-1-4503-3033-6. S2CID 12943246 (https://api.semanticscholar.org/CorpusID:12943246).
19. Willar, Dan E. (27 January 1983). "Log-logarithmic worst-case range queries are possible in space O(n)" (https://www.sciencedirect.com/science/article/abs/pii/0020019083900753). *Information Processing Letters*. **17** (2): 81–84. doi:10.1016/0020-0190(83)90075-3 (https://doi.org/10.1016%2F0020-0190%2883%2990075-3).
20. Sartaj Sahni (2004). "Data Structures, Algorithms, & Applications in C++: Tries" (https://www.cise.ufl.edu/~sahni/dsaac/enrich/c16/tries.htm). University of Florida. Archived (https://web.archive.org/web/20160703161316/http://www.cise.ufl.edu/~sahni/dsaac/enrich/c16/tries.htm) from the original on 3 July 2016. Retrieved 17 April 2022.
21. Mehta, Dinesh P.; Sahni, Sartaj (7 March 2018). "Tries". *Handbook of Data Structures and Applications* (https://www.routledge.com/Handbook-of-Data-Structures-and-Applications/Mehta-Sahni/p/book/9780367572006) (2 ed.). Chapman & Hall, University of Florida. ISBN 978-1498701853.

22. Jan Daciuk; Stoyan Mihov; Bruce W. Watson; Richard E. Watson (1 March 2000). "Incremental Construction of Minimal Acyclic Finite-State Automata" (https://direct.mit.edu/coli/article/26/1/3/1628/Incremental-Construction-of-Minimal-Acyclic-Finite). *Computational Linguistics*. **26** (1). MIT Press: 3–16. arXiv:cs/0007009 (https://arxiv.org/abs/cs/0007009). Bibcode:2000cs........7009D (https://ui.adsabs.harvard.edu/abs/2000cs........7009D). doi:10.1162/089120100561601 (https://doi.org/10.1162%2F089120100561601).
23. "Patricia tree" (https://xlinux.nist.gov/dads/HTML/patriciatree.html). National Institute of Standards and Technology. Archived (https://web.archive.org/web/20220214182428/https://xlinux.nist.gov/dads/HTML/patriciatree.html) from the original on 14 February 2022. Retrieved 17 April 2022.
24. Crochemore, Maxime; Lecroq, Thierry (2009). "Trie". *Encyclopedia of Database Systems* (https://link.springer.com/referencework/10.1007/978-0-387-39940-9). Boston, United States: Springer Publishing. Bibcode:2009eds..book.....L (https://ui.adsabs.harvard.edu/abs/2009eds..book.....L). doi:10.1007/978-0-387-39940-9 (https://doi.org/10.1007%2F978-0-387-39940-9). ISBN 978-0-387-49616-0 – via HAL (open archive).
25. Martinez-Prieto, Miguel A.; Brisaboa, Nieves; Canovas, Rodrigo; Claude, Francisco; Navarro, Gonzalo (March 2016). "Practical compressed string dictionaries" (https://www.sciencedirect.com/science/article/abs/pii/S0306437915001672). *Information Systems*. **56**. Elsevier: 73–108. doi:10.1016/j.is.2015.08.008 (https://doi.org/10.1016%2Fj.is.2015.08.008). ISSN 0306-4379 (https://search.worldcat.org/issn/0306-4379).
26. Kärkkäinen, Juha. "Lecture 2" (https://www.cs.helsinki.fi/u/tpkarkka/opetus/12s/spa/lecture02.pdf) (PDF). University of Helsinki. "The preorder of the nodes in a trie is the same as the lexicographical order of the strings they represent assuming the children of a node are ordered by the edge labels."
27. Kallis, Rafael (2018). "The Adaptive Radix Tree (Report #14-708-887)" (https://www.ifi.uzh.ch/dam/jcr:27d15f69-2a44-40f9-8b41-6d11b5926c67/ReportKallisMScBasis.pdf) (PDF). *University of Zurich: Department of Informatics, Research Publications*.
28. Ranjan Sinha and Justin Zobel and David Ring (Feb 2006). "Cache-Efficient String Sorting Using Copying" (https://people.eng.unimelb.edu.au/jzobel/fulltext/acmjea06.pdf) (PDF). *ACM Journal of Experimental Algorithmics*. **11**: 1–32. doi:10.1145/1187436.1187439 (https://doi.org/10.1145%2F1187436.1187439). S2CID 3184411 (https://api.semanticscholar.org/CorpusID:3184411).
29. J. Kärkkäinen and T. Rantala (2008). "Engineering Radix Sort for Strings". In A. Amir and A. Turpin and A. Moffat (ed.). *String Processing and Information Retrieval, Proc. SPIRE*. Lecture Notes in Computer Science. Vol. 5280. Springer. pp. 3–14. doi:10.1007/978-3-540-89097-3_3 (https://doi.org/10.1007%2F978-3-540-89097-3_3). ISBN 978-3-540-89096-6.
30. Giancarlo, Raffaele (28 May 1992). "A Generalization of the Suffix Tree to Square Matrices, with Applications" (https://epubs.siam.org/doi/abs/10.1137/S0097539792231982). *SIAM Journal on Computing*. **24** (3). Society for Industrial and Applied Mathematics: 520–562. doi:10.1137/S0097539792231982 (https://doi.org/10.1137%2FS0097539792231982). ISSN 0097-5397 (https://search.worldcat.org/issn/0097-5397).

31. Yang, Lai; Xu, Lida; Shi, Zhongzhi (23 March 2012). "An enhanced dynamic hash TRIE algorithm for lexicon search". *Enterprise Information Systems*. **6** (4): 419–432. Bibcode:2012EntIS...6..419Y (https://ui.adsabs.harvard.edu/abs/2012EntIS...6..419Y). doi:10.1080/17517575.2012.665483 (https://doi.org/10.1080%2F17517575.2012.665483). S2CID 37884057 (https://api.semanticscholar.org/CorpusID:37884057).
32. Transier, Frederik; Sanders, Peter (December 2010). "Engineering basic algorithms of an in-memory text search engine" (https://dl.acm.org/doi/10.1145/1877766.1877768). *ACM Transactions on Information Systems*. **29** (1). Association for Computing Machinery: 1–37. doi:10.1145/1877766.1877768 (https://doi.org/10.1145%2F1877766.1877768). S2CID 932749 (https://api.semanticscholar.org/CorpusID:932749).

External links

- NIST's Dictionary of Algorithms and Data Structures: Trie (https://xlinux.nist.gov/dads/HTML/trie.html)