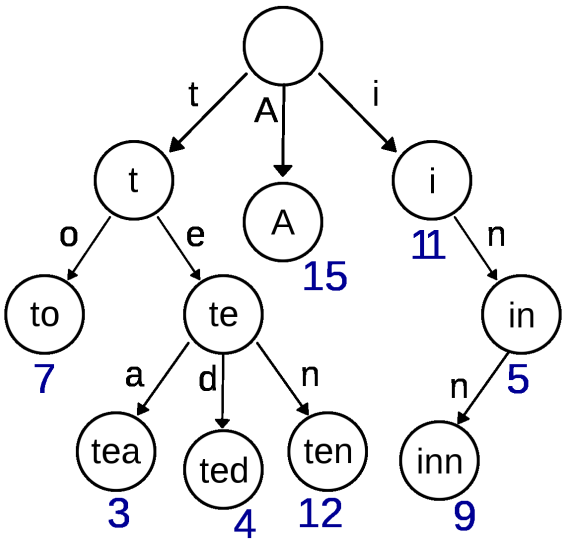
Trie

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 defines the key with which it is associated. 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 re**trie**val. Following the etymology, the inventor, Edward Fredkin, pronounces it English pronunciation: /ˈtriː/ "tree".[1][2] However, it is pronounced English pronunciation: /ˈtraɪ/ "try" by other authors.[1][2][3]



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

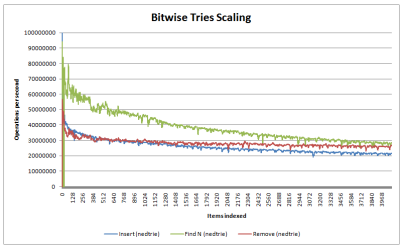
In the example shown, keys are listed in the nodes and values below them. Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a deterministic finite automaton, although the symbol on each edge is often implicit in the order of the branches.

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.)

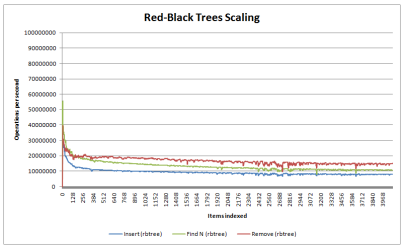
Though it is most common, tries need not be keyed by character strings. The same algorithms can easily be adapted to serve similar functions of ordered lists of any construct, e.g., permutations on a list of digits or shapes. In particular, a **bitwise trie** is keyed on the individual bits making up a short, fixed size of bits such as an integer number or pointer to memory.

Advantages relative to other search algorithms

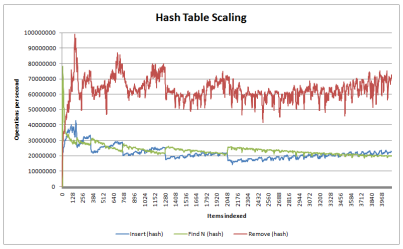
A series of graphs showing how different algorithms scale with number of items



Behavior of Fredkin-style tries as a function of size
(in this case, nedtries, which is an **in-place** implementation, and therefore has a much steeper curve than a dynamic memory based trie implementation)



Behavior of red-black trees as a function of size
(in this case, the BSD rbtree.h, which shows classic O(log N) behaviour)



Behavior of hash tables as a function of size
(in this case, uthash, which when averaged shows classic O(1) behaviour)

Unlike most other algorithms, tries have the peculiar feature that the code path, and hence the time required, is almost identical for insert, delete, and find operations. As a result, for situations where code is inserting, deleting and finding in equal measure, tries can handily beat binary search trees, as well as provide a better basis for the CPU's instruction and branch caches.

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 are more space-efficient when they contain a large number of short keys, since nodes are shared between keys with common initial subsequences.
- Tries facilitate longest-prefix matching.
- The number of internal nodes from root to leaf equals the length of the key. Balancing the tree is therefore of no concern.

The following are the main advantages of tries over hash tables:

- Tries support ordered iteration, whereas iteration over a hash table will result in a pseudorandom order given by the hash function (and further affected by the order of hash collisions, which is determined by the implementation).
- Tries facilitate longest-prefix matching, but hashing does not, as a consequence of the above. Performing such a "closest fit" find can, depending on implementation, be as quick as an exact find.
- Tries tend to be faster on average at insertion than hash tables because hash tables must rebuild their index when it becomes full - a very expensive operation. Tries therefore have much better bounded worst-case time costs, which is important for latency-sensitive programs.
- Since no hash function is used, tries are generally faster than hash tables for small keys.

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 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 necessary only 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.[5]

- Some keys, such as floating point numbers, can lead to long chains and prefixes that are not particularly meaningful. Nevertheless a bitwise trie can handle standard IEEE single and double format floating point numbers.

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. This is because an acyclic deterministic finite automaton can compress identical branches from the trie which correspond to the same suffixes (or parts) of different words being stored.

Tries are also well suited for implementing approximate matching algorithms, including those used in spell checking and hyphenation[2] 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 look up 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 find(node, key):
    for char in key:
        if char not in node.children:
            return None
        else:
            node = node.children[char]
    return node.value
```

A simple Ruby version:

```
class Trie
  def initialize
    @root = Hash.new
  end

  def build(str)
```

```
    node = @root
    str.each_char do |ch|
      node[ch] ||= Hash.new
      node = node[ch]
    end
    node[:end] = true
  end

  def find(str)
    node = @root
    str.each_char do |ch|
      return nil unless node = node[ch]
    end
    node[:end] && true
  end
end
```

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 lengths 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.

Bitwise tries

Bitwise tries are much the same as a normal character based trie except that individual bits are used to traverse what effectively becomes a form of binary tree. Generally, implementations use a special CPU instruction to very quickly find the first set bit in a fixed length key (e.g. GCC's __builtin_clz() intrinsic). This value is then used to index a 32 or 64 entry table which points to the first item in the bitwise trie with that number of leading zero bits. The search then proceeds by testing each subsequent bit in the key and choosing child[0] or child[1] appropriately until the item is found.

Although this process might sound slow, it is very cache-local and highly parallelizable due to the lack of register dependencies and therefore in fact performs excellent on modern out-of-order execution CPUs. A red-black tree for

example performs much better on paper, but is highly cache-unfriendly and causes multiple pipeline and TLB stalls on modern CPUs which makes that algorithm bound by memory latency rather than CPU speed. In comparison, a bitwise trie rarely accesses memory and when it does it does so only to read, thus avoiding SMP cache coherency overhead, and hence is becoming increasingly the algorithm of choice for code which does a lot of insertions and deletions such as memory allocators (e.g. recent versions of the famous Doug Lea's allocator (dlmalloc) and its descendents).

A reference implementation of bitwise tries in C and C++ useful for further study can be found at http://www.nedprod.com/programs/portable/nedtries/.

Compressing tries

When the trie is mostly static, i.e. all insertions or deletions of keys from a prefilled 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 of 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.[2] 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.

External links

- NIST's Dictionary of Algorithms and Data Structures: Trie [9]
- Trie implementation and visualisation in flash [10]
- Tries [11] by Lloyd Allison
- Using Tries [12] Topcoder tutorial
- An Implementation of Double-Array Trie [13]
- de la Briandais Tree [14]
- Discussing a trie implementation in Lisp [15]
- ServerKit "parse trees" implement a form of Trie in C [16]
- A simple implementation of Trie in Python [17]
- A Trie implemented in Ruby [18]
- A reference implementation of bitwise tries in C and C++ [19]
- A reference implementation in Java [20]
- A quick tutorial on TRIE in Java and C++ [21]
- A compact C implementation of Judy Tries [22]
- Data::Trie [23] and Tree::Trie [24] Perl implementations.

References

[1] Black, Paul E. (2009-11-16). "trie" (http://www.webcitation.org/5pqUULy24). *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology. Archived from the original (http://www.nist.gov/dads/HTML/trie.html) on 2010-05-19. .

[2] Franklin Mark Liang (1983). *Word Hy-phen-a-tion By Com-put-er* (http://www.webcitation.org/5pqOfzlIA) (Doctor of Philosophy thesis). Stanford University. Archived from the original (http://www.tug.org/docs/liang/liang-thesis.pdf) on 2010-05-19. . Retrieved 2010-03-28.

[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 0201896850.

[4] Bentley, Jon; Sedgewick, Robert (1998-04-01). "Ternary Search Trees" (http://web.archive.org/web/20080623071352/http://www.ddj.com/windows/184410528). *Dr. Dobb's Journal* (Dr Dobb's). Archived from the original (http://www.ddj.com/windows/184410528) on 2008-06-23. .

[5] Edward Fredkin (1960). "Trie Memory". *Communications of the ACM* **3** (9): 490–499. doi:10.1145/367390.367400.

[6] "Cache-Efficient String Sorting Using Copying" (http://www.cs.mu.oz.au/~rsinha/papers/SinhaRingZobel-2006.pdf) (PDF). . Retrieved 2008-11-15.

[7] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, Richard E. Watson (2000). "Incremental Construction of Minimal Acyclic Finite-State Automata" (http://www.mitpressjournals.org/doi/abs/10.1162/089120100561601). *Computational Linguistics* (Association for Computational Linguistics) **26**: 3. doi:10.1162/089120100561601. Archived from the original (http://www.pg.gda.pl/~jandac/daciuk98.ps.gz) on 2006-03-13. . Retrieved 2009-05-28. "This paper presents a method for direct building of minimal acyclic finite states automaton which recognizes a given finite list of words in lexicographical order. Our approach is to construct a minimal automaton in a single phase by adding new strings one by one and minimizing the resulting automaton on-the-fly"

[8] Ulrich Germann, Eric Joanis, Samuel Larkin (2009). "Tightly packed tries: how to fit large models into memory, and make them load fast, too" (http://www.aclweb.org/anthology/W/W09/W09-1505.pdf) (PDF). *ACL Workshops: Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*. Association for Computational Linguistics. pp. 31–39. . "We present Tightly Packed Tries (TPTs), a compact implementation of read-only, compressed trie structures with fast on-demand paging and short load times. We demonstrate the benefits of TPTs for storing n-gram back-off language models and phrase tables for statistical machine translation. Encoded as TPTs, these databases require less space than flat text file representations of the same data compressed with the gzip utility. At the same time, they can be mapped into memory quickly and be searched directly in time linear in the length of the key, without the need to decompress the entire file. The overhead for local decompression during search is marginal."

[9] http://www.nist.gov/dads/HTML/trie.html

[10] http://blog.ivank.net/trie-in-as3.html

[11] http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/Trie/

[12] http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=usingTries

[13] http://linux.thai.net/~thep/datrie/datrie.html

[14] http://tom.biodome.org/briandais.html

[15] http://groups.google.com/group/comp.lang.lisp/browse_thread/thread/01e485291d150938/9aacb626fa26c516

[16] http://serverkit.org/apiref-wip/node59.html

[17] http://kb.pyground.com/2011/05/trie-data-structure.html

[18] http://scanty-evidence-1.heroku.com/past/2010/5/10/ruby_trie/
[19] http://www.nedprod.com/programs/portable/nedtries/
[20] http://www.superliminal.com/sources/TrieMap.java.html
[21] http://www.technicalypto.com/2010/04/trie-in-java.html
[22] http://code.google.com/p/judyarray
[23] http://search.cpan.org/~hammond/data-trie-0.01/Trie.pm
[24] http://search.cpan.org/~avif/Tree-Trie-1.7/Trie.pm

- de la Briandais, R. (1959). "File Searching Using Variable Length Keys". *Proceedings of the Western Joint Computer Conference*: 295–298.

Article Sources and Contributors

Trie *Source*: http://en.wikipedia.org/w/index.php?oldid=490438047 *Contributors*: 97198, Adityasinghhhhhh, AlanUS, Altenmann, Andreas Kaufmann, Antaeus Feldspar, Anupchowdary, Atthaphong, Base698, Bignose, Blahedo, Bleakgadfly, Booyabazooka, Bryan Derksen, Bsdaemon, Chenopodiaceous, Coding.mike, Conversion script, Cowgod14, Cutelyaware, CyberSkull, Cybercobra, Danielx, Danny Rathjens, Dantheox, David Eppstein, Dbenbenn, Dcoetzee, Deborahjay, Deineka, Denshade, Diannaa, Diego Moya, Diomidis Spinellis, Doradus, Drewnoakes, Drpaule, Dscholte, Dysprosia, Edlee, Edward, Electrum, EmilJ, Enrique.benimeli, Eug, Francis Tyers, Fredrik, FuzziusMaximus, Gaius Cornelius, Gdr, Gerbrant, GeypycGn, Giftlite, Gmaxwell, Graham87, Ham Pastrami, Headbomb, Honza Záruba, Hugowolf, Ivan Kuckir, Jbragadeesh, JeffDonner, Jim baker, Jludwig, Jmacglashan, Johnny Zoo, JustinWick, Jyoti.mickey, KMeyer, Kaimiddleton, Kate, Kwamikagami, Leaflord, Let4time, LiDaobing, Loreto, Malbrain, Matt Gies, MattGiuca, Meand, Micahcowan, MichaelPloujnikov, Mostafa.vafi, MrOllie, Nad, Ned14, Neurodivergent, Nosbig, Otus, Para15000, Pgan002, Piet Delport, Pombredanne, Qwertyus, Raanoo, Rjwilmsi, Rl, Runtime, Sepreece, Sergio01, Shoujun, Simetrical, Slamb, Sperxios, Stephengmatthews, Stillnotelf, Superm401, Svick, TMott, Taral, Teacup, Tobias Bergemann, Tr00st, Watcher, WillNess, Wolfkeeper, X7q, Yaframa, 17אנונימי, مانې, 165 anonymous edits

Image Sources, Licenses and Contributors

Image:trie example.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Trie_example.svg *License*: Public Domain *Contributors*: Booyabazooka (based on PNG image by Deco). Modifications by Superm401.

File:BitwiseTreesScaling.png *Source*: http://en.wikipedia.org/w/index.php?title=File:BitwiseTreesScaling.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Ned14

File:RedBlackTreesScaling.png *Source*: http://en.wikipedia.org/w/index.php?title=File:RedBlackTreesScaling.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Ned14

File:HashTableScaling.png *Source*: http://en.wikipedia.org/w/index.php?title=File:HashTableScaling.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Ned14

License

Creative Commons Attribution-Share Alike 3.0 Unported
//creativecommons.org/licenses/by-sa/3.0/