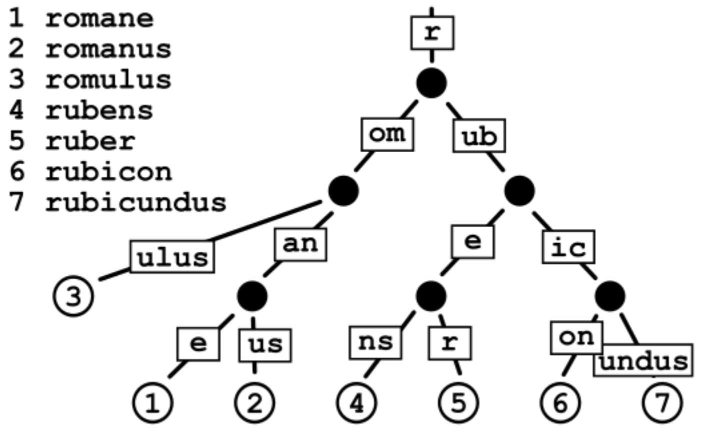
WIKIPEDIA

Radix tree

In computer science, a **radix tree** (also **radix trie** or **compact prefix tree**) is a data structure that represents a space-optimized trie in which each node that is the only child is merged with its parent. The result is that the number of children of every internal node is at least the radix *r* of the radix tree, where *r* is a positive integer and a power *x* of 2, having $x \geq 1$. Unlike in regular tries, edges can be labeled with sequences of elements as well as single elements. This makes radix trees much more efficient for small sets (especially if the strings are long) and for sets of strings that share long prefixes.



An example of a radix tree

Unlike regular trees (where whole keys are compared *en masse* from their beginning up to the point of inequality), the key at each node is compared chunk-of-bits by chunk-of-bits, where the quantity of bits in that chunk at that node is the radix *r* of the radix trie. When the *r* is 2, the radix trie is binary (i.e., compare that node's 1-bit portion of the key), which minimizes sparseness at the expense of maximizing trie depth—i.e., maximizing up to conflation of nondiverging bit-strings in the key. When *r* is an integer power of 2 greater or equal to 4, then the radix trie is an *r*-ary trie, which lessens the depth of the radix trie at the expense of potential sparseness.

As an optimization, edge labels can be stored in constant size by using two pointers to a string (for the first and last elements).[1]

Note that although the examples in this article show strings as sequences of characters, the type of the string elements can be chosen arbitrarily; for example, as a bit or byte of the string representation when using multibyte character encodings or Unicode.

Contents

Applications

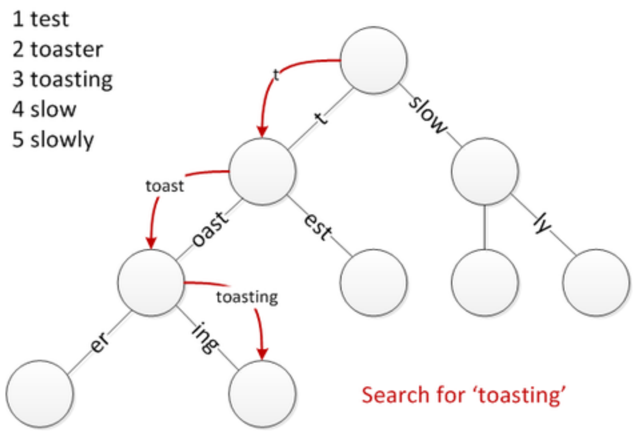
Radix trees are useful for constructing associative arrays with keys that can be expressed as strings. They find particular application in the area of IP routing,[2] where the ability to contain large ranges of values with a few exceptions is particularly suited to the hierarchical organization of IP addresses.[3] They are also used for inverted indexes of text documents in information retrieval.

Operations

Radix trees support insertion, deletion, and searching operations. Insertion adds a new string to the trie while trying to minimize the amount of data stored. Deletion removes a string from the trie. Searching operations include (but are not necessarily limited to) exact lookup, find predecessor, find successor, and find all strings with a prefix. All of these operations are O(k) where k is the maximum length of all strings in the set, where length is measured in the quantity of bits equal to the radix of the radix trie.

Lookup

The lookup operation determines if a string exists in a trie. Most operations modify this approach in some way to handle their specific tasks. For instance, the node where a string terminates may be of importance. This operation is similar to tries except that some edges consume multiple elements.



Finding a string in a Patricia trie

The following pseudo code assumes that these classes exist.

Edge

- *Node* targetNode
- *string* label

Node

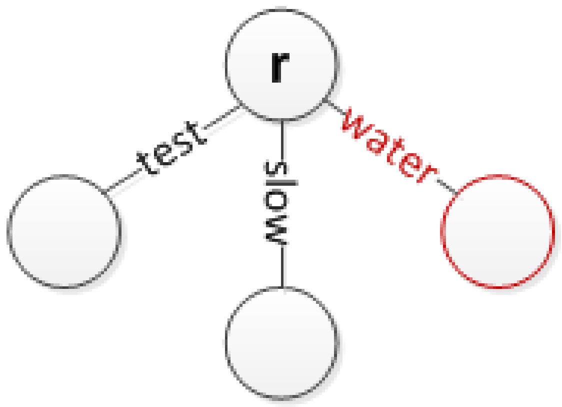
- *Array of Edges* edges
- *function* isLeaf()

```
function lookup(string x)
{
  // Begin at the root with no elements found
  Node traverseNode := root;
  int elementsFound := 0;

  // Traverse until a leaf is found or it is not possible to continue
  while (traverseNode != null && !traverseNode.isLeaf() && elementsFound < x.length)
  {
    // Get the next edge to explore based on the elements not yet found in x
    Edge nextEdge := select edge from traverseNode.edges where edge.label is a prefix of x.suffix(elementsFound)
      // x.suffix(elementsFound) returns the last (x.length - elementsFound) elements of x

    // Was an edge found?
    if (nextEdge != null)
    {
      // Set the next node to explore
      traverseNode := nextEdge.targetNode;

      // Increment elements found based on the label stored at the edge
      elementsFound += nextEdge.label.length;
    }
    else
    {
      // Terminate loop
      traverseNode := null;
    }
  }

  // A match is found if we arrive at a leaf node and have used up exactly x.length elements
  return (traverseNode != null && traverseNode.isLeaf() && elementsFound == x.length);
}
```

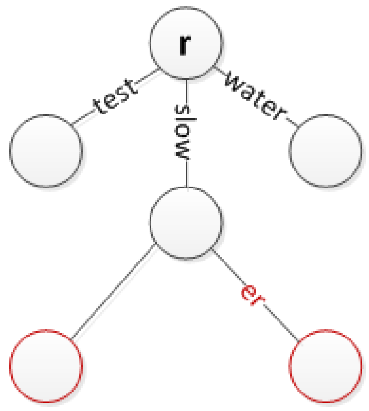
Insertion

To insert a string, we search the tree until we can make no further progress. At this point we either add a new outgoing edge labeled with all remaining elements in the input string, or if there is already an outgoing edge sharing a prefix with the remaining input string, we split it into two edges (the first labeled with the common prefix) and proceed. This splitting step ensures that no node has more children than there are possible string elements.

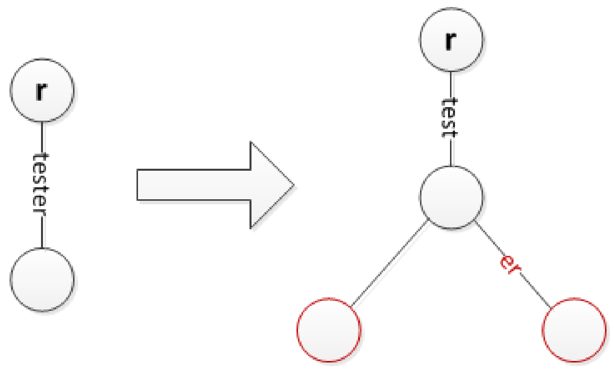
Several cases of insertion are shown below, though more may exist. Note that r simply represents the root. It is assumed that edges can be labelled with empty strings to terminate strings where necessary and that the root has no incoming edge. (The lookup algorithm described above will not work when using empty-string edges.)



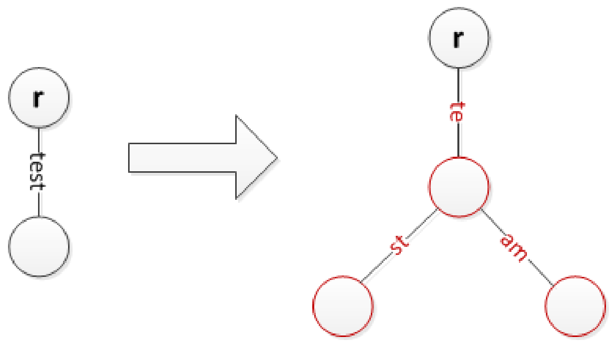
Insert 'water' at the root



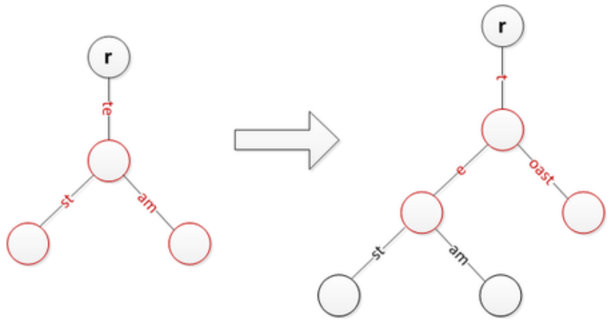
Insert 'slower' while keeping 'slow'



Insert 'test' which is a prefix of 'tester'



Insert 'team' while splitting 'test' and creating a new edge label 'st'



Insert 'toast' while splitting 'te' and moving previous strings a level lower

Deletion

To delete a string x from a tree, we first locate the leaf representing x. Then, assuming x exists, we remove the corresponding leaf node. If the parent of our leaf node has only one other child, then that child's incoming label is appended to the parent's incoming label and the child is removed.

Additional operations

- Find all strings with common prefix: Returns an array of strings which begin with the same prefix.
- Find predecessor: Locates the largest string less than a given string, by lexicographic order.
- Find successor: Locates the smallest string greater than a given string, by lexicographic order.

History

Donald R. Morrison first described what he called "Patricia trees" in 1968;[4] the name comes from the acronym **PATRICIA**, which stands for "*Practical Algorithm To Retrieve Information Coded In Alphanumeric*". Gernot Gwehenberger independently invented and described the data structure at about the same time.[5] PATRICIA trees are radix trees with radix equals 2, which means that each bit of the key is compared individually and each node is a two-way (i.e., left versus right) branch.

Comparison to other data structures

(In the following comparisons, it is assumed that the keys are of length k and the data structure contains n members.)

Unlike balanced trees, radix trees permit lookup, insertion, and deletion in $O(k)$ time rather than $O(\log n)$. This does not seem like an advantage, since normally $k \geq \log n$, but in a balanced tree every comparison is a string comparison requiring $O(k)$ worst-case time, many of which are slow in practice due to long common prefixes (in the case where comparisons begin at the start of the string). In a trie, all comparisons require constant time, but it takes m comparisons to look up a string of length m. Radix trees can perform these operations with fewer comparisons, and require many fewer nodes.

Radix trees also share the disadvantages of tries, however: as they can only be applied to strings of elements or elements with an efficiently reversible mapping to strings, they lack the full generality of balanced search trees, which apply to any data type with a total ordering. A reversible mapping to strings can be used to produce the required total ordering for balanced search trees, but not the other way around. This can also be problematic if a data type only provides a comparison operation, but not a (de)serialization operation.

Hash tables are commonly said to have expected O(1) insertion and deletion times, but this is only true when considering computation of the hash of the key to be a constant-time operation. When hashing the key is taken into account, hash tables have expected O(k) insertion and deletion times, but may take longer in the worst case depending on how collisions are handled. Radix trees have worst-case O(k) insertion and deletion. The successor/predecessor operations of radix trees are also not implemented by hash tables.

Variants

A common extension of radix trees uses two colors of nodes, 'black' and 'white'. To check if a given string is stored in the tree, the search starts from the top and follows the edges of the input string until no further progress can be made. If the search string is consumed and the final node is a black node, the search has failed; if it is white, the search has succeeded. This enables us to add a large range of strings with a common prefix to the tree, using white nodes, then remove a small set of "exceptions" in a space-efficient manner by *inserting* them using black nodes.

The **HAT-trie** is a cache-conscious data structure based on radix trees that offers efficient string storage and retrieval, and ordered iterations. Performance, with respect to both time and space, is comparable to the cache-conscious hashtable.[6][7] See HAT trie implementation notes at [1] (https://code.google.com/p/hat-trie)

The **adaptive radix tree** is a radix tree variant that integrates adaptive node sizes to the radix tree. One major drawback of the usual radix trees is the use of space, because it uses a constant node size in every level. The major difference between the radix tree and the adaptive radix tree is its variable size for each node based on the number of child elements, which grows while adding new entries. Hence, the adaptive radix tree leads to a better use of space without reducing its speed.[8][9][10]

See also

- Prefix tree (also known as a Trie)
- Deterministic acyclic finite state automaton (DAFSA)
- Ternary search tries
- Acyclic deterministic finite automata
- Hash trie
- Deterministic finite automata
- Judy array
- Search algorithm
- Extendible hashing
- Hash array mapped trie
- Prefix hash tree
- Burstsort
- Luleå algorithm
- Huffman coding

References

1. Morin, Patrick. "Data Structures for Strings" (http://cg.scs.carleton.ca/~morin/teaching/5408/notes/strings.pdf) (PDF). Retrieved 15 April 2012.
2. "rtfree(9)" (https://www.freebsd.org/cgi/man.cgi?query=rtfree&apropos=0&sektion=9&manpath=FreeBSD%2011-current&format=html). *www.freebsd.org*. Retrieved 2016-10-23.
3. Knizhnik, Konstantin. "Patricia Tries: A Better Index For Prefix Searches" (http://www.ddj.com/architect/208800854), *Dr. Dobb's Journal*, June, 2008.
4. Morrison, Donald R. Practical Algorithm to Retrieve Information Coded in Alphanumeric (http://portal.acm.org/citation.cfm?id=321481)
5. G. Gwehenberger, Anwendung einer binären Verweiskettenmethode beim Aufbau von Listen. (http://cr.yp.to/bib/1968/gwehenberger.html) Elektronische Rechenanlagen 10 (1968), pp. 223–226
6. Askitis, Nikolas; Sinha, Ranjan (2007). *HAT-trie: A Cache-conscious Trie-based Data Structure for Strings* (http://portal.acm.org/citation.cfm?id=1273749.1273761&coll=GUIDE&dl=). *Proceedings of the 30th Australasian Conference on Computer science*. **62**. pp. 97–105. ISBN 1-920682-43-0.

7. Askitis, Nikolas; Sinha, Ranjan (October 2010). "Engineering scalable, cache and space efficient tries for strings" (http://www.springerlink.com/content/86574173183j6565/). *The VLDB Journal*. **19** (5): 633–660. doi:10.1007/s00778-010-0183-9 (https://doi.org/10.1007%2Fs00778-010-0183-9).
8. Kemper, Alfons; Eickler, André (2013). *Datenbanksysteme, Eine Einführung*. **9**. pp. 604–605. ISBN 978-3-486-72139-3.
9. "armon/libart · GitHub" (https://github.com/armon/libart). *GitHub*. Retrieved 17 September 2014.
10. http://www-db.in.tum.de/~leis/papers/ART.pdf

External links

- Algorithms and Data Structures Research & Reference Material: PATRICIA (http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/PATRICIA/), by Lloyd Allison, Monash University
- Patricia Tree (https://xlinux.nist.gov/dads/HTML/patriciatree.html), NIST Dictionary of Algorithms and Data Structures
- Crit-bit trees (http://cr.yp.to/critbit.html), by Daniel J. Bernstein
- Radix Tree API in the Linux Kernel (https://lwn.net/Articles/175432/), by Jonathan Corbet
- Kart (key alteration radix tree) (http://code.dogmap.org/kart/), by Paul Jarc

Implementations

- FreeBSD Implementation (https://github.com/freebsd/freebsd/blob/master/sys/net/radix.h), used for paging, forwarding and other things.
- Linux Kernel Implementation (https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/lib/radix-tree.c), used for the page cache, among other things.
- GNU C++ Standard library has a trie implementation (https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/trie_based_containers.html)
- Java implementation of Concurrent Radix Tree (https://code.google.com/p/concurrent-trees/), by Niall Gallagher
- C# implementation of a Radix Tree (http://paratechnical.blogspot.com/2011/03/radix-tree-implementation-in-c.html)
- Practical Algorithm Template Library (https://code.google.com/p/patl/), a C++ library on PATRICIA tries (VC++ >=2003, GCC G++ 3.x), by Roman S. Klyujkov
- Patricia Trie C++ template class implementation (http://www.codeproject.com/KB/string/PatriciaTrieTemplateClass.aspx), by Radu Gruian
- Haskell standard library implementation (http://hackage.haskell.org/packages/archive/containers/latest/doc/html/Data-IntMap.html) "based on big-endian patricia trees". Web-browsable source code (http://hackage.haskell.org/packages/archive/containers/latest/doc/html/src/Data-IntMap.html).
- Patricia Trie implementation in Java (https://code.google.com/p/patricia-trie/), by Roger Kapsi and Sam Berlin
- Crit-bit trees (https://github.com/agl/critbit) forked from C code by Daniel J. Bernstein
- Patricia Trie implementation in C (http://cprops.sourceforge.net/gen/docs/trie_8c-source.html), in libcprops (http://cprops.sourceforge.net)
- Patricia Trees : efficient sets and maps over integers in (http://www.lri.fr/~filliatr/ftp/ocaml/ds) OCaml, by Jean-Christophe Filliâtre
- Radix DB (Patricia trie) implementation in C (https://github.com/balena/radixdb), by G. B. Versiani

Retrieved from "https://en.wikipedia.org/w/index.php?title=Radix_tree&oldid=785457367"