Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Čeština
Deutsch
Español
فارسی
Français
한국어
Հայերեն
Italiano
עברית
Lietuvių
Nederlands
日本語
Norsk bokmål
Polski
Português
Русский
Slovenčina
Српски / srpski
Suomi
Türkçe
Українська
Tiếng Việt
中文
Edit links

Radix sort

From Wikipedia, the free encyclopedia

In computer science, **radix sort** is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. A positional notation is required, but because integers can represent strings of characters (e.g., names or dates) and specially formatted floating point numbers, radix sort is not limited to integers. Radix sort dates back as far as 1887 to the work of Herman Hollerith on tabulating machines.[1]

Radix sort

Class	Sorting algorithm
Data structure	Array
Worst case performance	$O(wn)$
Worst case space complexity	$O(w + N)$

Most digital computers internally represent all of their data as electronic representations of binary numbers, so processing the digits of integer representations by groups of binary digit representations is most convenient. Two classifications of radix sorts are least significant digit (LSD) radix sorts and most significant digit (MSD) radix sorts. LSD radix sorts process the integer representations starting from the least digit and move towards the most significant digit. MSD radix sorts work the other way around.

LSD radix sorts typically use the following sorting order: short keys come before longer keys, and keys of the same length are sorted lexicographically. This coincides with the normal order of integer representations, such as the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.

MSD radix sorts use lexicographic order, which is suitable for sorting strings, such as words, or fixed-length integer representations. A sequence such as "b, c, d, e, f, g, h, i, j, ba" would be lexicographically sorted as "b, ba, c, d, e, f, g, h, i, j". If lexicographic ordering is used to sort variable-length integer representations, then the representations of the numbers from 1 to 10 would be output as 1, 10, 2, 3, 4, 5, 6, 7, 8, 9, as if the shorter keys were left-justified and padded on the right with blank characters to make the shorter keys as long as the longest key for the purpose of determining sorted order.

Contents [hide]

Efficiency [edit]

The topic of the efficiency of radix sort compared to other sorting algorithms is somewhat tricky and subject to quite a lot of misunderstandings. Whether radix sort is equally efficient, less efficient or more efficient than the best comparison-based algorithms depends on the details of the assumptions made. Radix sort complexity is $O(wn)$ for n keys which are integers of word size w. Sometimes w is presented as a constant, which would make radix sort better (for sufficiently large n) than the best comparison-based sorting algorithms, which all perform $O(n \log n)$ comparisons to sort n keys. However, in general w cannot be considered a constant: if all n keys are distinct, then w has to be at least $\log n$ for a random-access machine to be able to store them in

memory, which gives at best a time complexity $O(n \log n)$.[2] That would seem to make radix sort at most equally efficient as the best comparison-based sorts (and worse if keys are much longer than $\log n$).

The counter argument is the comparison-based algorithms are measured in number of comparisons, not actual time complexity. Under some assumptions the comparisons will be constant time on average, under others they will not. Comparisons of randomly-generated keys takes constant time on average, as keys differ on the very first bit in half the cases, and differ on the second bit in half of the remaining half, and so on, resulting in an average of two bits that need to be compared. In a sorting algorithm the first comparisons made satisfies the randomness condition, but as the sort progresses the keys compared are clearly not randomly chosen anymore. For example, consider a bottom-up merge sort. The first pass will compare pairs of random keys, but the last pass will compare keys that are very close in the sorting order.

The deciding factor is how the keys are distributed. The best case for radix sort is that they are taken as consecutive bit patterns. This will make the keys as short as they can be, still assuming they are distinct. This makes radix sort $O(n \log n)$, but the comparison based sorts will not be as efficient, as the comparisons will not be constant time under this assumption. If we instead assume that the keys are bit patterns of length $k \log n$ for a constant $k > 1$ and base-two logarithm, and that they are uniformly random, then radix sort will still be $O(n \log n)$, but so will the comparison based sorts, as the "extra" length makes even the keys that are consecutive in the sorted result differ enough that comparisons are constant time on average. If keys are longer than $O(\log n)$, but random, then radix sort will be inferior. There are many other assumptions that can be made as well, and most require careful study to make a correct comparison.

Least significant digit radix sorts [edit]

A Least significant digit (LSD) RADIX sort is a fast stable sorting algorithm which can be used to sort keys in integer representation order. Keys may be a string of characters, or numerical digits in a given 'radix'. The processing of the keys begins at the least significant digit (i.e., the rightmost digit), and proceeds to the most significant digit (i.e., the leftmost digit). The sequence in which digits are processed by an LSD radix sort is the opposite of the sequence in which digits are processed by a most significant digit (MSD) radix sort.

An LSD radix sort operates in O(*nk*) time, where *n* is the number of keys, and *k* is the average key length. This kind of performance for variable-length keys can be achieved by grouping all of the keys that have the same length together and separately performing an LSD radix sort on each group of keys for each length, from shortest to longest, in order to avoid processing the whole list of keys on every sorting pass.

A radix sorting algorithm was originally used to sort punched cards in several passes. A computer algorithm was invented for radix sort in 1954 at MIT by Harold H. Seward. In many large applications needing speed, the computer radix sort is an improvement on (slower) comparison sorts.

LSD radix sorts have resurfaced as an alternative to high performance comparison-based sorting algorithms (like heapsort and mergesort) that require O($n \cdot \log n$) comparisons, where *n* is the number of items to be sorted. Comparison sorts can do no better than O($n \cdot \log n$) execution time but offer the flexibility of being able to sort with respect to more complicated orderings than a lexicographic one; however, this ability is of little importance in many practical applications.

Definition [edit]

Each key is first figuratively dropped into one level of buckets corresponding to the value of the rightmost digit. Each bucket preserves the original order of the keys as the keys are dropped into the bucket. There is a one-to-one correspondence between the buckets and the values that can be represented by the rightmost digit. Then, the process repeats with the next neighbouring more significant digit until there are no more digits to process. In other words:

1. Take the least significant digit (or group of bits, both being examples of radices) of each key.
2. Group the keys based on that digit, but otherwise keep the original order of keys. (This is what makes the LSD radix sort a stable sort.)
3. Repeat the grouping process with each more significant digit.

The sort in step 2 is usually done using bucket sort or counting sort, which are efficient in this case since there are usually only a small number of digits.

An example [edit]

Original, unsorted list:

170, 45, 75, 90, 802, 2, 24, 66

Sorting by least significant digit (1s place) gives:

170, 90, 802, 2, 24, 45, 75, 66

Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.

Sorting by next digit (10s place) gives:

802, 2, 24, 45, 66, 170, 75, 90

Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

It is important to realize that each of the above steps requires just a single pass over the data, since each item can be placed in its correct bucket without having to be compared with other items.

Some radix sort implementations allocate space for buckets by first counting the number of keys that belong in each bucket before moving keys into those buckets. The number of times that each digit occurs is stored in an array. Consider the previous list of keys viewed in a different way:

170, 045, 075, 090, 002, 024, 802, 066

The first counting pass starts on the least significant digit of each key, producing an array of bucket sizes:

2 (bucket size for digits of 0: 170, 090)
2 (bucket size for digits of 2: 002, 802)
1 (bucket size for digits of 4: 024)
2 (bucket size for digits of 5: 045, 075)
1 (bucket size for digits of 6: 066)

A second counting pass on the next more significant digit of each key will produce an array of bucket sizes:

2 (bucket size for digits of 0: 002, 802)
1 (bucket size for digits of 2: 024)
1 (bucket size for digits of 4: 045)
1 (bucket size for digits of 6: 066)
2 (bucket size for digits of 7: 170, 075)
1 (bucket size for digits of 9: 090)

A third and final counting pass on the most significant digit of each key will produce an array of bucket sizes:

6 (bucket size for digits of 0: 002, 024, 045, 066, 075, 090)
1 (bucket size for digits of 1: 170)
1 (bucket size for digits of 8: 802)

At least one LSD radix sort implementation now counts the number of times that each digit occurs in each column for all columns in a single counting pass. (See the external links section.) Other LSD radix sort implementations allocate space for buckets dynamically as the space is needed.

Iterative version using queues [edit]

A simple version of an LSD radix sort can be achieved using queues as buckets. The following process is repeated for a number of times equal to the length of the longest key:

1. The integers are enqueued into an array of ten separate queues based on their digits from right to left. Computers often represent integers internally as fixed-length binary digits. Here, we will do something analogous with fixed-length decimal digits. So, using the numbers from the previous example, the queues for the 1st pass would be:
   0: 170, 090
   1: none
   2: 802, 002
   3: none
   4: 024
   5: 045, 075
   6: 066
   7–9: none
2. The queues are dequeued back into an array of integers, in increasing order. Using the same numbers,

the array will look like this after the first pass:

170, 090, 802, 002, 024, 045, 075, 066

3. For the second pass:

   Queues:

   0: 802, 002
   1: none
   2: 024
   3: none
   4: 045
   5: none
   6: 066
   7: 170, 075
   8: none
   9: 090

   Array:

   802, 002, 024, 045, 066, 170, 075, 090
   (note that at this point only 802 and 170 are out of order)

4. For the third pass:

   Queues:

   0: 002, 024, 045, 066, 075, 090
   1: 170
   2–7: none
   8: 802
   9: none

   Array:

   002, 024, 045, 066, 075, 090, 170, 802 (sorted)

While this may not be the most efficient radix sort algorithm, it is relatively simple, and still quite efficient. During all tests on 100M or fewer random 64-bit integers, qsort algorithm behaves faster. The following code in C is based on the counting sort algorithm provided in Cormen et al.,[3] with some improvements concerning memory allocation.

Most significant digit radix sorts [edit]

A most significant digit (MSD) radix sort can be used to sort keys in lexicographic order. Unlike a least significant digit (LSD) radix sort, a most significant digit radix sort does not necessarily preserve the original order of duplicate keys. An MSD radix sort starts processing the keys from the most significant digit, leftmost digit, to the least significant digit, rightmost digit. This sequence is opposite that of least significant digit (LSD) radix sorts. An MSD radix sort stops rearranging the position of a key when the processing reaches a unique prefix of the key. Some MSD radix sorts use one level of buckets in which to group the keys. See the counting sort and pigeonhole sort articles. Other MSD radix sorts use multiple levels of buckets, which form a trie or a path in a trie. A postman's sort / postal sort is a kind of MSD radix sort.

Recursion [edit]

A recursively subdividing MSD radix sort algorithm works as follows:

1. Take the most significant digit of each key.
2. Sort the list of elements based on that digit, grouping elements with the same digit into one bucket.
3. Recursively sort each bucket, starting with the next digit to the right.
4. Concatenate the buckets together in order.

Recursive forward radix sort example [edit]

Sort the list:
170, 045, 075, 090, 002, 024, 802, 066

1. Sorting by most significant digit (100s place) gives:
   Zero hundreds bucket: 045, 075, 090, 002, 024, 066
   One hundreds bucket: 170
   Eight hundreds bucket: 802

2. Sorting by next digit (10s place) is only needed for those numbers in the zero hundreds bucket (no other buckets contain more than one item):
   Zero tens bucket: 002
   Twenties bucket: 024
   Forties bucket: 045
   Sixties bucket: 066
   Seventies bucket: 075
   Nineties bucket: 090

Sorting by least significant digit (1s place) is not needed, as there is no tens bucket with more than one number. Therefore, the now sorted zero hundreds bucket is concatenated, joined in sequence, with the one hundreds bucket and eight hundreds bucket to give:
002, 024, 045, 066, 075, 090, 170, 802

This example used base ten digits for the sake of readability, but of course binary digits or perhaps bytes might make more sense for a binary computer to process.

In-place MSD radix sort implementations [edit]

Binary MSD radix sort, also called binary quicksort, can be implemented in-place by splitting the input array into two bins - the 0s bin and the 1s bin. The 0s bin is grown from the beginning of the array, whereas the 1s bin is grown from the end of the array. The 0s bin boundary is placed before the first array element. The 1s bin boundary is placed after the last array element. The most significant bit of the first array element is examined. If this bit is a 1, then the first element is swapped with the element in front of the 1s bin boundary (the last element of the array), and the 1s bin is grown by one element by decrementing the 1s boundary array index. If this bit is a 0, then the first element remains at its current location, and the 0s bin is grown by one element. The next array element examined is the one in front of the 0s bin boundary (i.e. the first element that is not in the 0s bin or the 1s bin). This process continues until the 0s bin and the 1s bin reach each other. The 0s bin and the 1s bin are then sorted recursively based on the next bit of each array element. Recursive processing continues until the least significant bit has been used for sorting.[4][5] Handling signed integers requires treating the most significant bit with the opposite sense, followed by unsigned treatment of the rest of the bits.

In-place MSD binary-radix sort can be extended to larger radix and retain in-place capability. Counting sort is used to determine the size of each bin and their starting index. Swapping is used to place the current element into its bin, followed by expanding the bin boundary. As the array elements are scanned the bins are skipped over and only elements between bins are processed, until the entire array has been processed and all elements end up in their respective bins. The number of bins is the same as the radix used - e.g. 16 bins for 16-Radix. Each pass is based on a single digit (e.g. 4-bits per digit in the case of 16-Radix), starting from the most significant digit. Each bin is then processed recursively using the next digit, until all digits have been used for sorting.[6][7]

Neither in-place binary-radix sort nor n-bit-radix sort, discussed in paragraphs above, are stable algorithms.

Stable MSD radix sort implementations [edit]

MSD Radix Sort can be implemented as a stable algorithm, but requires the use of a memory buffer of the same size as the input array. This extra memory allows the input buffer to be scanned from the first array element to last, and move the array elements to the destination bins in the same order. Thus, equal elements will be placed in the memory buffer in the same order they were in the input array. The MSD-based algorithm uses the extra memory buffer as the output on the first level of recursion, but swaps the input and output on the next level of recursion, to avoid the overhead of copying the output result back to the input buffer. Each of the bins are recursively processed, as is done for the in-place MSD Radix Sort. After the sort by the last digit has been completed, the output buffer is checked to see if it is the original input array, and if it's not, then a single copy is performed. If the digit size is chosen such that the key size divided by the digit size is an even number, the copy at the end is avoided.[8]

Hybrid approaches [edit]

Radix sort, such as two pass method where counting sort is used during the first pass of each level of recursion, has a large constant overhead. Thus, when the bins get small, other sorting algorithms should be used, such as insertion sort. A good implementation of Insertion sort is fast for small arrays, stable, in-place, and can significantly speed up Radix Sort.

Application to parallel computing [edit]

Note that this recursive sorting algorithm has particular application to parallel computing, as each of the bins can be sorted independently. In this case, each bin is passed to the next available processor. A single processor would be used at the start (the most significant digit). By the second or third digit, all available processors would likely be engaged. Ideally, as each subdivision is fully sorted, fewer and fewer processors would be utilized. In the worst case, all of the keys will be identical or nearly identical to each other, with the result that there will be little to no advantage to using parallel computing to sort the keys.

In the top level of recursion, opportunity for parallelism is in the Counting sort portion of the algorithm. Counting is highly parallel, amenable to the parallel_reduce pattern, and splits the work well across multiple cores until reaching memory bandwidth limit. This portion of the algorithm has data-independent parallelism. Processing each bin in subsequent recursion levels is data-dependent, however. For example, if all keys were of the same value, then there would be only a single bin with any elements in it, and no parallelism would be available. For random inputs all bins would be near equally populated and a large amount of parallelism opportunity would be available.[9]

Note that there are faster sorting algorithms available, for example optimal complexity O(log(n)) are those of the Three Hungarians and Richard Cole[10][11] and Batcher's bitonic merge sort has an algorithmic complexity of O($\log^2(n)$), all of which have a lower algorithmic time complexity to radix sort on a CREW-PRAM. The fastest known PRAM sorts were described in 1991 by David Powers with a parallelized quicksort that can operate in O(log(n)) time on a CRCW-PRAM with n processors by performing partitioning implicitly, as well as a radixsort that operates using the same trick in O(k), where k is the maximum keylength.[12] However, neither the PRAM architecture or a single sequential processor can actually be built in a way that will scale without the number of constant fanout gate delays per cycle increasing as O(log(n)), so that in effect a pipelined version of Batcher's bitonic mergesort and the O(log(n)) PRAM sorts are all O($\log^2(n)$) in terms of clock cycles, with Powers acknowledging that Batcher's would have lower constant in terms of gate delays than his Parallel quicksort and radix sort, or Cole's merge sort, for a keylength-independent sorting network of O(n$\log^2(n)$).[13]

Incremental trie-based radix sort [edit]

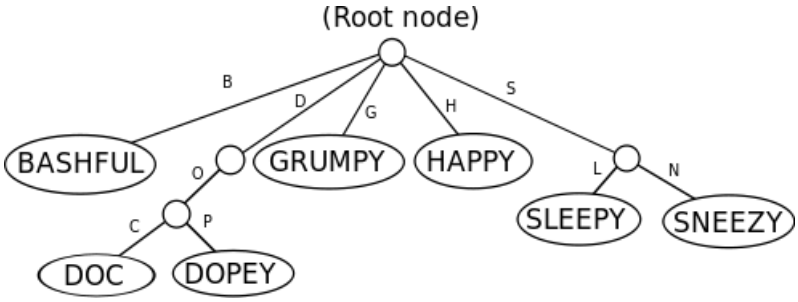
Another way to proceed with an MSD radix sort is to use more memory to create a trie to represent the keys and then traverse the trie to visit each key in order. A depth-first traversal of a trie starting from the root node will visit each key in order. A depth-first traversal of a trie, or any other kind of acyclic tree structure, is equivalent to traversing a maze via the right-hand rule.

A trie essentially represents a set of strings or numbers, and a radix sort which uses a trie structure is not necessarily stable, which means that the original order of duplicate keys is not necessarily preserved, because a set does not contain duplicate elements. Additional information will have to be associated with each key to indicate the population count or original order of any duplicate keys in a trie-based radix sort if keeping track of that information is important for a particular application. It may even be desirable to discard any duplicate strings as the trie creation proceeds if the goal is to find only unique strings in sorted order. Some people sort a list of strings first and then make a separate pass through the sorted list to discard duplicate strings, which can be slower than using a trie to simultaneously sort and discard duplicate strings in one pass.

One of the advantages of maintaining the trie structure is that the trie makes it possible to determine quickly if a particular key is a member of the set of keys in a time that is proportional to the length of the key, k, in O(k) time, that is *independent* of the total number of keys. Determining set membership in a plain list, as opposed to determining set membership in a trie, requires binary search, O(k *log(n)*) time; linear search, O(kn) time; or some other method whose execution time is in some way dependent on the total number, n, of all of the keys in the worst case. It is sometimes possible to determine set membership in a plain list in O(k) time, in a time that is independent of the total number of keys, such as when the list is known to be in an arithmetic sequence or some other computable sequence.

Maintaining the trie structure also makes it possible to insert new keys into the set incrementally or delete keys from the set incrementally while maintaining sorted order in O(k) time, in a time that is independent of the total number of keys. In contrast, other radix sorting algorithms must, in the worst case, re-sort the entire list of keys each time that a new key is added or deleted from an existing list, requiring O(kn) time.

Snow White analogy [edit]

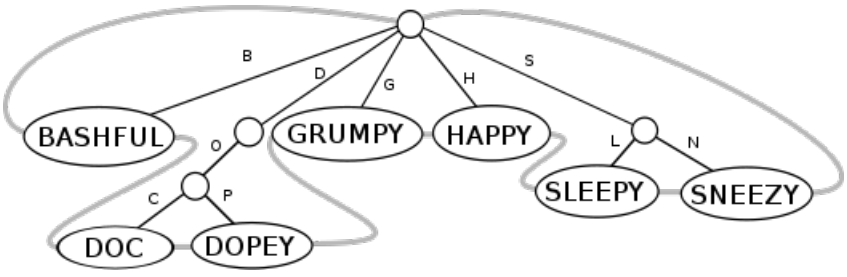


If the nodes were rooms connected by hallways, then here is how Snow White might proceed to visit all of the dwarfs if the place were dark, keeping her right hand on a wall at all times:

1. She travels down hall B to find Bashful.
2. She continues moving forward with her right hand on the wall, which takes her around the room and back up hall B.
3. She moves down halls D, O, and C to find Doc.
4. Continuing to follow the wall with her right hand, she goes back up hall C, then down hall P, where she finds Dopey.
5. She continues back up halls P, O, D, and then goes down hall G to find Grumpy.
6. She goes back up hall G, with her right hand still on the wall, and goes down hall H to the room where Happy is.
7. She travels back up hall H and turns right down halls S and L, where she finds Sleepy.
8. She goes back up hall L, down hall N, where she finally finds Sneezy.
9. She travels back up halls N and S to her starting point and knows that she is done.

These series of steps serve to illustrate the path taken in the trie by Snow White via a depth-first traversal to visit the dwarfs by the ascending order of their names, Bashful, Doc, Dopey, Grumpy, Happy, Sleepy, and Sneezy. The algorithm for performing some operation on the data associated with each node of a tree first, such as printing the data, and then moving deeper into the tree is called a pre-order traversal, which is a kind of depth-first traversal. A pre-order traversal is used to process the contents of a trie in ascending order. If Snow White wanted to visit the dwarfs by the descending order of their names, then she could walk backwards while following the wall with her right hand, or, alternatively, walk forward while following the wall with her left hand. The algorithm for moving deeper into a tree first until no further descent to unvisited nodes is possible and then performing some operation on the data associated with each node is called post-order traversal, which is another kind of depth-first traversal. A post-order traversal is used to process the contents of a trie in descending order.

The root node of the trie in the diagram essentially represents a null string, an empty string, which can be useful for keeping track of the number of blank lines in a list of words. The null string can be associated with a circularly linked list with the null string initially as its only member, with the forward and backward pointers both initially pointing to the null string. The circularly linked list can then be expanded as each new key is inserted into the trie. The circularly linked list is represented in the following diagram as thick, grey, horizontally linked lines:

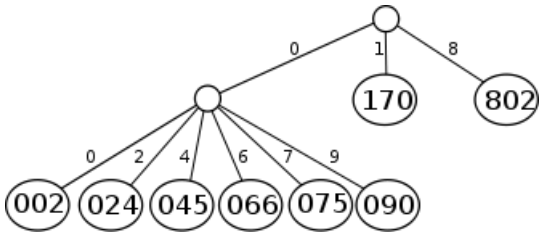


If a new key, other than the null string, is inserted into a leaf node of the trie, then the computer can go to the last preceding node where there was a key or a bifurcation to perform a depth-first search to find the lexicographic successor or predecessor of the inserted key for the purpose of splicing the new key into the circularly linked list. The last preceding node where there was a key or a bifurcation, a fork in the path, is a parent node in the type of trie shown here, where only unique string prefixes are represented as paths in the trie. If there is already a key associated with the parent node that would have been visited during a movement *away* from the root during a right-hand, forward-moving, depth-first traversal, then that immediately ends the depth-first search, as that key is the predecessor of the inserted key. For example, if Bashful is inserted into the trie, then the predecessor is the null string in the parent node, which is the root node in this case. In other words, if the key that is being inserted is on the leftmost branch of the parent node, then any string contained in

the parent node is the lexicographic predecessor of the key that is being inserted, else the lexicographic predecessor of the key that is being inserted exists down the parent node's branch that is immediately to the left of the branch where the new key is being inserted. For example, if Grumpy were the last key inserted into the trie, then the computer would have a choice of trying to find either the predecessor, Dopey, or the successor, Happy, with a depth-first search starting from the parent node of Grumpy. With no additional information to indicate which path is longer, the computer might traverse the longer path, D, O, P. If Dopey were the last key inserted into the trie, then the depth-first search starting from the parent node of Dopey would soon find the predecessor, "Doc", because that would be the only choice.

If a new key is inserted into an internal node, then a depth-first search can be started from the internal node to find the lexicographic successor. For example, if the literal string "DO" were inserted in the node at the end of the path D, O, then a depth-first search could be started from that internal node to find the successor, "DOC", for the purpose of splicing the new string into the circularly linked list.

Forming the circularly linked list requires more memory but allows the keys to be visited more directly in either ascending or descending order via a linear traversal of the linked list rather than a depth-first traversal of the entire trie. This concept of a circularly linked trie structure is similar to the concept of a threaded binary tree. This structure will be called a circularly threaded trie.



When a trie is used to sort numbers, the number representations must all be the same length unless you are willing to perform a breadth-first traversal. When the number representations will be visited via depth-first traversal, as in the above diagram, the number representations will always be on the leaf nodes of the trie. Note how similar in concept this particular example of a trie is to the recursive forward radix sort example which involves the use of buckets instead of a trie. Performing a radix sort with the buckets is like creating a trie and then discarding the non-leaf nodes.

See also [edit]

- IBM 80 series Card Sorters
- Spaghetti sort

References [edit]

1. ^ US 395781 and UK 327
2. ^ Nilsson, Stefan (1 April 2000). "The fastest sorting algorithm?". *Dr Dobb's Journal* **311**: 38–45.
3. ^ Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Third Edition. MIT Press, 2009, pp. 195.
4. ^ R. Sedgewick, "Algorithms in C++", third edition, 1998, p. 424-427
5. ^ V. J. Duvanenko, "In-Place Hybrid Binary-Radix Sort", Dr. Dobb's Journal, 1 October 2009
6. ^ V. J. Duvanenko, "In-Place Hybrid N-bit-Radix Sort", Dr. Dobb's Journal, November 2009
7. ^ V. J. Duvanenko, "Parallel In-Place Radix Sort Simplified", Dr. Dobb's Journal, January 2011
8. ^ V. J. Duvanenko, "Stable Hybrid N-bit-Radix Sort", Dr. Dobb's Journal, January 2010
9. ^ V. J. Duvanenko, "Parallel In-Place N-bit-Radix Sort", Dr. Dobb's Journal, August 2010
10. ^ A. Gibbons and W. Rytter, "Efficient Parallel Algorithms". Cambridge University Press, 1988.
11. ^ H. Casanova et al, "Parallel Algorithms". Chapman & Hall, 2008.
12. ^ David M. W. Powers, Parallelized Quicksort and Radixsort with Optimal Speedup, *Proceedings of International Conference on Parallel Computing Technologies*. Novosibirsk. 1991.
13. ^ David M. W. Powers, Parallel Unification: Practical Complexity, Australasian Computer Architecture Workshop, Flinders University, January 1995

External links [edit]

- Demonstration and comparison of Radix sort with Bubble sort, Merge sort and Quicksort implemented in JavaScript
- Article about Radix sorting IEEE floating-point numbers with implementation.

  Faster Floating Point Sorting and Multiple Histogramming with implementation in C++

- Pointers to radix sort visualizations
- USort library contains tuned implementations of radix sort for most numerical C types (C99)
- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Section 5.2.5: Sorting by Distribution, pp. 168–179.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 8.3: Radix sort, pp. 170–173.
- BRADSORT v1.50 source code
- Efficient Trie-Based Sorting of Large Sets of Strings, by Ranjan Sinha and Justin Zobel. This paper describes a method of creating tries of buckets which figuratively burst into sub-tries when the buckets hold more than a predetermined capacity of strings, hence the name, "Burstsort".
- Open Data Structures - Java Edition - Section 11.2 - Counting Sort and Radix Sort
- Open Data Structures - C++ Edition - Section 11.2 - Counting Sort and Radix Sort

v · t · e	**Sorting algorithms** [hide]
Theory	Computational complexity theory · Big O notation · Total order · Lists · Inplacement · Stability · Comparison sort · Adaptive sort · Sorting network · Integer sorting
Exchange sorts	Bubble sort · Cocktail sort · Odd–even sort · Comb sort · Gnome sort · Quicksort · Stooge sort · Bogosort
Selection sorts	Selection sort · Heapsort · Smoothsort · Cartesian tree sort · Tournament sort · Cycle sort
Insertion sorts	Insertion sort · Shellsort · Splaysort · Tree sort · Library sort · Patience sorting
Merge sorts	Merge sort · Cascade merge sort · Oscillating merge sort · Polyphase merge sort · Strand sort
Distribution sorts	American flag sort · Bead sort · Bucket sort · Burstsort · Counting sort · Pigeonhole sort · Proxmap sort · **Radix sort** · Flashsort
Concurrent sorts	Bitonic sorter · Batcher odd–even mergesort · Pairwise sorting network
Hybrid sorts	Block sort · Timsort · Introsort · Spreadsort · JSort
Other	Topological sorting · Pancake sorting · Spaghetti sort

Categories: Sorting algorithms | Stable sorts